Journal of Applied Logic 10 (2012) 32–51

Contents lists available at SciVerse ScienceDirect

Journal of Applied Logic

www.elsevier.com/locate/jal

Translating regular expression matching into transducers ☆

Yuto Sakuma [a], Yasuhiko Minamide [a,*], Andrei Voronkov [b]

[a] Department of Computer Science, University of Tsukuba, Tsukuba 305-8573, Japan
[b] University of Manchester, Oxford Road, Manchester, M13 9PL, United Kingdom

ARTICLE INFO

Article history:
Available online 12 November 2011

Keywords:
Regular expression
Semantics
Transducer
Monad
Program analysis

ABSTRACT

Regular expression matching is an essential tool in string manipulating programs and plays crucial roles in scripting languages. We focus on regular expression matching based on the strategy of Perl and develop a translation from regular expression matching into transducers. The representation makes it possible to apply the theory of formal languages in static analysis and verification of string manipulating programs.

We first formulate the semantics of regular expression matching as a nondeterministic parser by using the composition of the list and output monads. Then, we transform the nondeterministic parser into deterministic one by introducing lookahead. The deterministic parser is formulated with the option monad instead of the list monad and derived through equational reasoning involving monads. From the definition of the deterministic parser, we can easily construct transducers through transducers with regular lookahead. We have implemented the translation and conducted experiments on regular expressions found in several popular PHP programs.

1. Introduction

Regular expression matching is an essential tool in string manipulating programs. It is used not only for checking whether a string is in the language of a regular expression, but also for extracting a substring matching against a part of the regular expression. Regular expression matching (and replacement) are ubiquitous and extensively used for crucial checks and operations for security in programs written in scripting languages such as Perl and PHP. Therefore, it is important to precisely analyze regular expression matching in static analysis and verification for scripting languages.

The representation of string manipulating primitives is crucial in static analysis and verification for scripting languages. We represented string manipulating primitives by *transducers* in the PHP string analyzer described in [17]. A transducer is basically an automaton with output, a generalization of Mealy machines [16], and has been deeply studied in the theory of formal languages. That uniform representation of string manipulating primitives makes it possible to directly apply the theory of formal languages. The same representation is also used for automated test case generation in [31]. However some of the string manipulating primitives may be impossible or difficult to represent by transducers. This is especially true for commonly used regular expression matching, due to its rather involved semantics. Therefore, our PHP string analyzer previously adopted a very coarse approximation of regular expression matching.

In this paper, we focus on regular expression matching based on the strategy of Perl, and develop a *precise* translation of regular expression matching into transducers. We obtain transducers that simulate the behaviour of regular expression

☆ An earlier version of this paper appeared in the Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2010.

* Corresponding author.
E-mail address: minamide@cs.tsukuba.ac.jp (Y. Minamide).

1570-8683/$ – see front matter
doi:10.1016/j.jal.2011.11.003

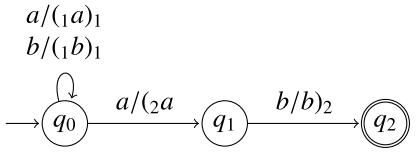
matching in Perl without any approximation. Since regular expression matching in most other scripting languages is based on that in Perl, the translation is applicable to other languages including PHP. The translation supports most commonly-used features of regular expressions in Perl excluding some exotic features such as backreference and atomic grouping. Adapting the precise translation, we can improve the precision of the static analysis based on transducers. The translation also has another interesting application: we can check the equivalence of regular expressions used for matching by deciding the equivalence of transducers obtained by the translation.

To explain our goal by an example, let us consider the following program written in Perl.

```
$var = "abaab";
if ($var =~ m/(a|b)*(ab)/) {
print $1;      ==> a
print $2;      ==> ab
}
```

It matches string `abaab` with regular expression $(a|b)^*(ab)$. The matching not only checks whether `abaab` is in $L((a|b)^*(ab))$, but also stores the last substring matching each grouped subexpression (that is, an expression enclosed in parentheses). The substrings matching against $(a|b)$ and (ab) are captured in `$1` and `$2`, respectively. Since $(a|b)$ matches a, b, a in this order, a is stored in `$1`.

We represent this capturing behaviour of regular expression matching by a transducer that annotates the input with *indexed parentheses*. For example, we build a transducer T for $(a|b)^*(ab)$ so that $T(abaab) = ({}_1a)_1({}_1b)_1({}_1a)_1({}_2ab)_2$. The following transducer realizes this.



Actually, translating regular expression matching directly into transducers is highly non-trivial. Instead, we translate regular expression matching into transducers with regular lookahead, and then use a result that every transducer with regular lookahead can be converted into one without lookahead.

The first step in our development is a rigorous description of the semantics of regular expression matching. The difficulty in catching the semantics is caused by ambiguities in regular expressions. For example, the regular expression $a|ab$ can match a substring of ab in two different ways: the first letter a matches against a in $a|ab$ or the whole string matches against ab in $a|ab$. There are several possible strategies to disambiguate regular expression matching; their semantics was studied by Vansummeren [28].

In this paper, we focus on the disambiguation strategy of Perl. This strategy is stated as follows. In alternation $r_1|r_2$, the expression r_1 has a higher priority than r_2. Then, the repetition r^* is interpreted as $rr^*|\epsilon$. To capture this strategy of Perl precisely, we formulate its semantics as a nondeterministic parser by using the list monad. The disambiguation strategy is represented by using the convention that the first element in a list has the highest priority. The semantics of capturing is given by extending the list monad by composing it with the output monad.

To obtain the construction of transducers from the semantics, we introduce a deterministic version of the parser as an intermediate step. The deterministic version chooses a branch of nondeterministic choices by using lookahead as transducers with regular lookahead. It is formulated with the option monad instead of the list monad and derived from the nondeterministic parser through equational reasoning involving monads.

Finally, we develop our translation from regular expression matching into transducers with regular lookahead. The construction is a refinement of the Thompson's standard construction of ϵ-NFA [27] and obtained from the definition of the deterministic parser. We show the correctness of our construction by proving the correspondence with the deterministic parser. From a transducer with regular lookahead, we can obtain one without lookahead by eliminating lookaheads with an existing technique.

We have implemented the translation from regular expression matching into transducers without lookahead, and conducted experiments on regular expressions found in several popular PHP programs. Although the construction has exponential complexity, the experimental results do not show the exponential blow-up in practice.

It should be noted that Perl-style regular expression matching without backreference can be implemented so that it is executed in linear time [3,5]. Thus, the translation to transducers will not be suitable as an implementation of matching. However, it can be the basis of the analysis of matching where we need to apply the theory of automata and transducers.

An earlier version of this paper appeared in the Proceedings of the 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2010 [18]. The semantics of the regular expression matching in this paper is extensively revised so that it is modularized by applying the output monad as a monad transformer: the conference version only utilized the list monad.

This paper is organized as follows. Section 2 reviews monads in the theory of programing languages and transducers with and without regular lookahead. We formulate the semantics of regular expression matching as a nondeterministic parser by using the list monad in Section 3. Then, we derive a deterministic parser using the option monad from the nondeterministic parser in Section 4. The nondeterministic and deterministic parsers for the problematic case are discussed in Section 5. Section 6 introduces the construction of transducers corresponding to the deterministic parser and proves its correctness. In Section 7 we describe our implementation and show experimental results. Finally, we discuss related works and conclude.

2. Preliminaries

We review monads in the theory of programming languages and transducers with and without regular lookahead.

2.1. Monad

We briefly review monads in the theory of programming languages, and introduce concrete monads, monad morphisms, and monad transformers used in this paper. Please refer the papers [19,20,29,30] for more details on monads.

A *monad* M is a type constructor M with the following polymorphic functions:

$$\begin{aligned} &\mathit{unit}_M :: \alpha \to \alpha\, M \\ &\mathit{bind}_M :: \alpha\, M \to (\alpha \to \beta\, M) \to \beta\, M \end{aligned}$$

where we adopt the postfix notation $\alpha\, M$ for type constructors. We write $m \gg\!=_M f$ for $\mathit{bind}_M\, m\, f$. These functions must satisfy the following monad laws.

$$\begin{aligned} (\mathit{unit}_M\, x) \gg\!=_M f &= f\, x \\ m \gg\!=_M (\lambda x.\mathit{unit}_M\, x) &= m \\ (m \gg\!=_M f) \gg\!=_M g &= m \gg\!=_M (\lambda x.\, f\, x \gg\!=_M g) \end{aligned}$$

We often omit the subscripts of the unit and bind functions if they are not significant or clear from the contexts.

The semantics of various constructs in programming languages can be naturally described by choosing a suitable monad. We first review the (power) set monad that can be used to represent nondeterministic computation. We write $\alpha\, \mathit{set}$ for the type consisting of sets over α. Then, the unit and bind functions on the set monad are defined as follows.

$$\begin{aligned} \mathit{unit}\, x &= \{x\} \\ m \gg\!= f &= \bigcup_{x \in m} f\, x \end{aligned}$$

The set monad also forms a monoid with the identity $\emptyset$ and the binary operation $\cup$.

Although the set monad is suitable to describe nondeterministic computation, it is not expressive enough to describe the semantics of regular expression matching. We need to consider priority between possible choices. Thus, we actually use the list monad [29] in Section 3.2 to formulate the semantics where we use the convention that the first element in a list has the highest priority. We write a list consisting of $v_1, v_2, \ldots, v_n$ as $[v_1, v_2, \ldots, v_n]$ and the concatenation of the list l_1 and l_2 as $l_1 +\!+ l_2$. Then, the unit and bind functions of the list monad are given as follows:

$$\begin{aligned} \mathit{unit}\, x &= [x] \\ m \gg\!= f &= \mathit{concat}\, (\mathit{map}\, f\, m) \end{aligned}$$

where the functions *map* and *concat* are defined in the standard manner.

$$\begin{aligned} \mathit{map}\, f\, [v_1, v_2, \ldots, v_n] &= [f\, v_1, f\, v_2, \ldots, f\, v_n] \\ \mathit{concat}\, [v_1, v_2, \ldots, v_n] &= v_1 +\!+ v_2 +\!+ \cdots +\!+ v_n \end{aligned}$$

The list monad is closely related to the set monad and also forms a monoid with $[\,]$ and $+\!+$, which correspond to $\emptyset$ and $\cup$ on the set monad.

To relate monads we apply monad morphisms in this paper. A polymorphic function h from $\alpha\, M_1$ to $\alpha\, M_2$ is called a *monad morphism* from M_1 to M_2 if the following holds [30].

$$\begin{aligned} h\, (\mathit{unit}_{M_1}\, x) &= \mathit{unit}_{M_2}\, x \\ h\, (m \gg\!=_{M_1} f) &= \big(h\, m \gg\!=_{M_2} \lambda x.h\, (f\, x)\big) \end{aligned}$$

For example, the function *set* converting lists into sets below is a monad morphism from the list monad to the set monad.

$$\mathit{set}\big([v_1, v_2, \ldots, v_n]\big) = \{v_1, v_2, \ldots, v_n\}$$

This monad morphism also satisfies the following properties.

$$\begin{aligned} \mathit{set}\big([\,]\big) &= \emptyset \\ \mathit{set}(m_1 +\!+ m_2) &= \mathit{set}(m_1) \cup \mathit{set}(m_2) \end{aligned}$$

It is a morphism on the monoid structures over the list and set monads. This morphism is used when we ignore priority represented by the list monad.

In order to formulate the semantics of capturing, we apply the output monad where the type constructor *out* is defined as $\alpha\, out = \alpha \times \tau$. The following associated functions constitute a monad for *out* when the type τ is a monoid with binary operation $\cdot$ and identity e.

$$\begin{aligned} unit\, x &= (x, e) \\ (x, v) \gg\!= f &= \mathsf{let}\ (y, v') = f\, x\ \mathsf{in}\ (y, v \cdot v') \end{aligned}$$

The semantics of output is formulated by the following function *out*:

$$out :: \alpha\, out \to (\alpha \to \tau) \to \alpha\, out$$

$$out\, (x, v)\, g = (x, v \cdot g\, x)$$

where $g\, x$ generates the output for x, and it is combined with the existing output v.[1] The output monad is not only a monad, but also a *monad transformer* that can be composed with any monad [15]. If a type constructor M is a monad, then $\alpha\, out\, M$ is also a monad for the parameter α. The unit, bind, and output functions for the composed monad are given as follows:

$$\begin{aligned} unit\, x &= unit_M\, (x, e) \\ m \gg\!= f &= m \gg\!=_M \left(\lambda(x, v).\, f\, x \gg\!=_M \left(\lambda(x', v').unit_M\, (x', v \cdot v')\right)\right) \\ out\, m\, g &= m \gg\!=_M \left(\lambda m'.unit_M\, (out_{out}\, m'\, g)\right) \end{aligned}$$

This monad transformer is used to extend the semantics of regular expression matching based on the list monad for capturing in a modular manner.

2.2. Transducer

We briefly review transducers in this section. The detailed description of the theory of transducers can be found in [1, 25].

A *transducer* T is a structure $\langle Q, \Sigma_i, \Sigma_o, \Delta, I, F\rangle$ where Q is a finite set of states, Σ_i and Σ_o are input and output alphabets, I and F are the sets of initial and final states, and $\Delta \subseteq Q \times \Sigma_i^* \times \Sigma_o^* \times Q$ is the set of transitions. We write $q \xrightarrow[T]{w/v} q'$ if $(q, w, v, q') \in \Delta$. It denotes that T may read input w and write output v at the state q and change its state to q'. It is extended to zero or more steps of transition, $q \xrightarrow[T]{w/v}{}^* q'$, as follows:

$$q \xrightarrow[T]{\epsilon/\epsilon}{}^* q$$

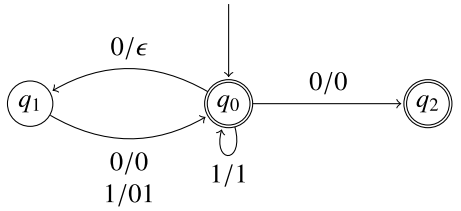
$$q \xrightarrow[T]{w_1 w_2/v_1 v_2}{}^* q'' \quad \text{if } q \xrightarrow[T]{w_1/v_1} q' \text{ and } q' \xrightarrow[T]{w_2/v_2}{}^* q''$$

where ϵ denotes the empty word. Then, the language (or behaviour) of T is defined as follows:

$$|T| = \left\{ (w, v) \mid \exists q_i \in I.\, \exists q_f \in F.\, q_i \xrightarrow[T]{w/v}{}^* q_f \right\}$$

We say that a transducer T is functional if $|T|$ represents a partial function. The equivalence of two transducers is undecidable in general, but it is decidable for functional transducers [25].

Example 1. The following transducer replaces all non-overlapping occurrences of 00 with 0 in an input word over $\{0, 1\}$. A label w/v above (or below) an arrow denotes that w is the input and v is the output.



Although this transducer is nondeterministic, it is functional and actually represents a total function.

[1] This definition of *out* is different from one in [29]. We adopt this definition to simplify the semantics of capturing we will discuss in Section 3.3.

Although transducers can express many of string manipulating operations, it is rather difficult to precisely model regular expression matching directly. Thus, we employ *transducers with regular lookahead* in the study of top-down tree transducers [7]. A transducer T with regular lookahead is a structure $\langle Q, \Sigma_i, \Sigma_o, \Delta, I, F\rangle$ as transducers without lookahead. The only difference is the set of transitions Δ: $\Delta \subseteq Q \times \Sigma_i^* \times \Sigma_o^* \times Q \times Reg(\Sigma_i)$ where $Reg(\Sigma_i)$ is the set of regular languages over Σ_i. The transition relation is extended to the form $q \xrightarrow[T]{w/v|w'} q'$, which denotes that T may read input w and write output v at the state q and change its state to q' *if the rest of the input is* w' *after reading* w. The definition of transition is revised as follows:

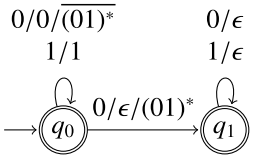
$$q \xrightarrow[T]{w/v|w'} q' \quad \text{if } (q, w, v, q', R) \in \Delta \text{ and } ww' \in R$$

The transition $(q, w, v, q', R) \in \Delta$ can be taken only when the rest of input ww' is in the lookahead set R. The relation is extended to zero or more steps of transition as follows.

$$q \xrightarrow[T]{\epsilon/\epsilon|w'}{}^{*} q \quad \text{for any } w' \in \Sigma^*$$

$$q \xrightarrow[T]{w_1w_2/v_1v_2|w'}{}^{*} q'' \quad \text{if } q \xrightarrow[T]{w_1/v_1|w_2w'}{}^{*} q' \text{ and } q' \xrightarrow[T]{w_2/v_2|w'}{}^{*} q''$$

Example 2. The following transducer replaces trailing $(01)^+$ with ϵ for an input word over $\{0, 1\}$. The lookahead $(01)^*$ and its complement $\overline{(01)^*}$ are utilized to check whether the rest of the input is a repetition of 01. A label $w/v/r$ above (or below) an arrow denotes that w is the input, v is the output, and $L(r)$ is the lookahead set.

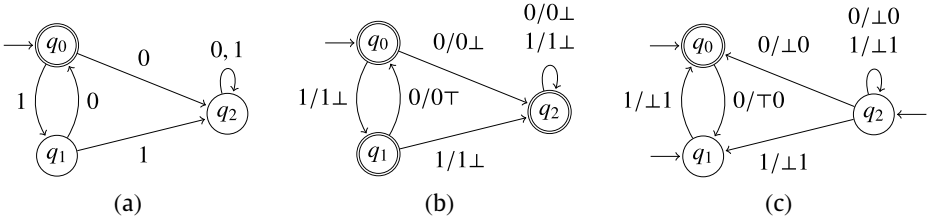


We can convert a transducer T with regular lookahead into one without lookahead. We decompose T into two transducers without lookahead and compose them to obtain the transducer without lookahead [7].

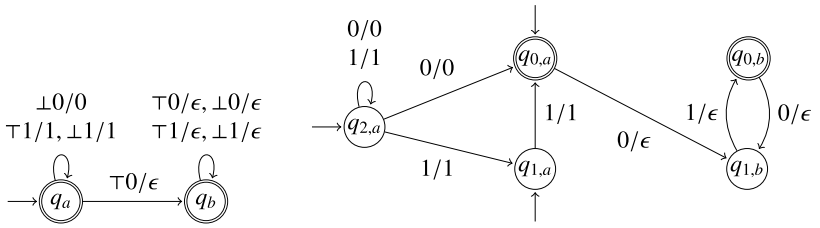
The first transducer preprocesses an input from the end to annotate the input. We call it a *preprocessing transducer*. Assume there are lookaheads $r_1, r_2, \ldots, r_k$ in the original transducer. We first construct a DFA (Deterministic Finite Automaton) A_i accepting the reverse of each r_i and construct the product $A_1 \times A_2 \times \cdots \times A_k$. Since the input word is preprocessed from the end, each DFA checks whether each postfix is in the language of the lookahead. Then we can enrich the DFA with output to insert annotation in every position in the input word.

The second transducer is almost identical to the original transducer with regular lookahead. However, instead of checking the rest of the input for a lookahead, it checks the annotation inserted by the preprocessing transducer.

Example 3. We illustrate the construction of the transducer without lookahead corresponding to Example 2. The following diagrams illustrate the construction of the preprocessing transducer, which inserts letter $\top$ or $\bot$.



(a) is a DFA accepting the reverse of $(01)^*$, *i.e.* $(10)^*$. (b) is obtained by modifying the transitions of (a) such that the letter $\top$ is inserted when the destination of the transition is the final state of the DFA (*i.e.*, q_0) and otherwise $\bot$ is inserted. All the states are now the final states. (c) is the reverse of (b): the initial and final states are changed to the final and initial states, respectively, and the transition is reversed.



(d) (e)

The transducer (d) simulates the transducer of Example 2 by checking the inserted letters. By composing (c) and (d), we obtain the transducer (e) without lookahead equivalent to the transducer in Example 2. The composition is basically the product construction.

Consider the state complexity of the transducer without lookahead. The size of the preprocessing transducer is in $O(2^{d(|r_1|+|r_2|+\cdots+|r_k|)})$ for some constant d. Then, by composing the preprocessing transducer with the original transducer T, we obtain a transducer without lookahead of size $O(|T| \cdot 2^{d(|r_1|+|r_2|+\cdots+|r_k|)})$.

3. Semantics of regular expression matching

We formulate the semantics of regular expression matching as a nondeterministic parser using the list monad according to the Perl-compatible strategy. The semantics of capturing is formulated by extending the list monad by composing it with the option monad. We also prove the soundness of the nondeterministic parser in the sense that it is correct if we ignore its disambiguation strategy.

3.1. Syntax of regular expressions

We discuss the regular expression matching for a fixed alphabet denoted by Σ. We call elements of Σ *letters* and elements of Σ^* *words*. The empty word is written as ϵ.

In the concrete syntax of regular expressions in programming languages, parentheses are not only used to group an expression, but also to capture a matched subword in a variable. To distinguish the parentheses for capturing and to simplify their semantics, we introduce a *capturing* expression $(r)_x$ indexed by a named variable x. Then, the syntax of regular expressions is defined as follows:

$$
\begin{array}{rll}
r ::= & \epsilon & \text{(empty word)} \\
| & c & \text{(letter)} \\
| & r_1 r_2 & \text{(concatenation)} \\
| & r_1 | r_2 & \text{(alternation)} \\
| & r^* & \text{(greedy repetition)} \\
| & r^{*?} & \text{(lazy repetition)} \\
| & (r)_x & \text{(capturing)}
\end{array}
$$

where $c \in \Sigma$. Although it is custom to write alternation as $r_1 + r_2$ in the theory of regular expressions, we write $r_1|r_2$ because it is closer to the concrete syntax in programming languages. There are two variants of repetition: *greedy* repetition r^* and *lazy* repetition $r^{*?}$. The expressions r^* and $r^{*?}$ are basically equivalent to $rr^*|\epsilon$ and $\epsilon|rr^{*?}$, respectively. Please note that an alternation $r_1|r_2$ is not commutative and the first alternative r_1 has a higher priority. For a capturing expression $(r)_x$, a subword matched with r is assigned to x.

The language represented by a regular expression r is defined in the standard manner as follows:

$$
\begin{aligned}
L(\epsilon) &= \{\epsilon\} \\
L(c) &= \{c\} \\
L(r_1 r_2) &= L(r_1)L(r_2) \\
L(r_1|r_2) &= L(r_1) \cup L(r_2) \\
L\big(r_1^*\big) &= L(r_1)^* \\
L\big(r_1^{*?}\big) &= L(r_1)^* \\
L\big((r_1)_x\big) &= L(r_1)
\end{aligned}
$$

where the greedy and lazy repetition has no difference and the capturing is ignored.

3.2. Nondeterministic parser

We formalize the semantics of regular expression matching as a nondeterministic parser where nondeterminism is represented by the list monad. The disambiguation strategy is represented by using the convention that the first element in a list has the highest priority. This formalization is inspired by the multithreaded implementation of regular expression matching discussed in [4].

We formalize the semantics of regular expression matching as the nondeterministic parser $\mathcal{N}[\![r]\!]$. We first ignore the capturing construct and postpone its discussion to the next subsection. Then, the nondeterministic parser $\mathcal{N}[\![r]\!]$ is defined as the function that computes the possible rest of words after the parsing of r as follows.

$$
\begin{aligned}
\mathcal{N}[\![r]\!] &:: \Sigma^* \to \Sigma^* \mathit{list} \\
\mathcal{N}[\![\epsilon]\!]w &= \mathit{unit}\, w \\
\mathcal{N}[\![c]\!]w &= \begin{cases} \mathit{unit}\, w' & \text{if } w = cw' \\ [\,] & \text{otherwise} \end{cases} \\
\mathcal{N}[\![r_1 r_2]\!]w &= \mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\mathcal{N}[\![r_2]\!]w' \\
\mathcal{N}[\![r_1|r_2]\!]w &= \mathcal{N}[\![r_1]\!]w +\!+ \mathcal{N}[\![r_2]\!]w \\
\mathcal{N}[\![r_1^*]\!]w &= \big(\mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\mathcal{N}[\![r_1^*]\!]w'\big) +\!+ \mathit{unit}\, w \\
\mathcal{N}[\![r_1^{*?}]\!]w &= \mathit{unit}\, w +\!+ \big(\mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\mathcal{N}[\![r_1^{*?}]\!]w'\big)
\end{aligned}
$$

Since r_1 has a higher priority in $r_1|r_2$, then $\mathcal{N}[\![r_1]\!]w$ must be at the left of $+\!+$. Repetition r_1^* and $r_1^{*?}$ are interpreted as $r_1 r_1^*|\epsilon$ and $\epsilon|r_1 r_1^{*?}$ as expected.

Although the definition above is quite intuitive, it is *not well-defined* in the *problematic* case (when $\epsilon \in L(r_1)$ for r_1^* and $r_1^{*?}$). In that case, $\mathcal{N}[\![r_1]\!]w$ contains w and thus $\mathcal{N}[\![r_1^*]\!]w$ recursively calls $\mathcal{N}[\![r_1^*]\!]w$. Hence, the definition above is well-defined only when $\epsilon \notin L(r_1)$ for r_1^* and $r_1^{*?}$. In this section we assume $\epsilon \notin L(r_1)$ for r_1^* and $r_1^{*?}$, and postpone the discussion of the problematic case to Section 5.

We formulate the soundness of $\mathcal{N}[\![r]\!]$ by using the so-called left quotient of a word. The *left quotient* of w by w', written as ${w'}^{-1}w$, is a word defined as follows:

$${w'}^{-1}w = \begin{cases} w'' & \text{if } w = w'w'' \\ \text{undefined} & \text{if } w' \text{ is not a prefix of } w \end{cases}$$

It is naturally extended to $L^{-1}w$ for $L \subseteq \Sigma^*$.

$$L^{-1}w = \{w'' \mid \exists w' \in L.\, w = w'w''\}$$

Then, the nondeterministic parser $\mathcal{N}[\![r]\!]$ is sound in the sense that it is correct if we ignore its disambiguation strategy. Priority represented by the list monad is ignored by applying the monad morphism *set*. The theorem is proved by induction on the lexicographic order of the structure of r and the length of w. Each case is easily proved by the properties of the monad morphism *set*.

Theorem 1. $\mathit{set}(\mathcal{N}[\![r]\!]w) = L(r)^{-1}w$.

Proof. We only show the case $r = r_1^*$ where $\epsilon \notin L(r_1)$.

$$
\begin{aligned}
\mathit{set}\big(\mathcal{N}[\![r_1^*]\!]w\big) &= \mathit{set}\big(\big(\mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\mathcal{N}[\![r_1^*]\!]w'\big) +\!+ \mathit{unit}\, w\big) \\
&= \bigcup_{v \in \mathit{set}(\mathcal{N}[\![r_1]\!]w)} \mathit{set}\big(\mathcal{N}[\![r_1^*]\!]v\big) \cup \{w\} \\
&= \bigcup_{v \in L(r_1)^{-1}w} \big(L\big(r_1^*\big)^{-1}v\big) \cup \{w\} \quad \big(\text{by I.H., } |v| < |w| \text{ from } \epsilon \notin L(r_1)\big) \\
&= L\big(r_1 r_1^*\big)^{-1}w \cup \{w\} \\
&= L\big(r_1^*\big)^{-1}w \qquad \square
\end{aligned}
$$

3.3. Extension for capturing

We formulate the semantics of capturing by extending the nondeterministic parser $\mathcal{N}[\![r]\!]$ with the output monad. To formulate the semantics of $(r)_x$, captured words need to be recorded to an environment as the side effect of matching. Let *Var* be the set of variables that can be used in capturing $(r)_x$. We represent the environment *Env* as a partial function from

Var to Σ^*. The set of environments *Env* constitutes a monoid for $\bot$ and $\uplus$ where $\bot(x)$ is undefined for any x and infix operator $\uplus$ is defined as follows:

$$(\rho_1 \uplus \rho_2)(x) = \begin{cases} \rho_2(x) & \text{if } \rho_2(x) \text{ is defined} \\ \rho_1(x) & \text{otherwise} \end{cases}$$

Then, we apply the output monad for *Env* to formulate the behaviour of capturing. Actually, we compose the list monad with the output monad as we discussed in Section 2.1 and obtain the monad of type $(\alpha \times \mathit{Env})\,\mathit{list}$. The following definition of the functions *unit*, *bind*, and *out* are obtained as a result of the monad composition.

$$\begin{aligned} \mathit{unit}\, x &= \big[(x, \bot)\big] \\ m \gg\!= f &= m \gg\!=_{\mathit{list}} \big(\lambda(w, \rho).\, f\, w \gg\!=_{\mathit{list}} \big(\lambda(w', \rho').\big[(w', \rho \uplus \rho')\big]\big)\big) \\ \mathit{out}\, m\, g &= \mathit{map}\,\big(\lambda(v, \rho).(v, \rho \uplus g\, v)\big)\, m \end{aligned}$$

Then, we extend $\mathcal{N}[\![r]\!]$ as a function from Σ^* to $(\Sigma^* \times \mathit{Env})\,\mathit{list}$. The definitions for the constructs except for capturing remain the same except that the unit and bind functions for the composed monad must be used. Then, the semantics of capturing is formulated by the function *out* as follows:

$$\mathcal{N}[\![(r_1)_x]\!]\, w = \mathit{out}\,\big(\mathcal{N}[\![r_1]\!]\, w\big)\,\big(\lambda w'.\bot[x \mapsto w w'^{-1}]\big)$$

where the right quotient of a word $w w'^{-1}$ is defined as follows.

$$w w'^{-1} = \begin{cases} w'' & \text{if } w = w'' w' \\ \text{undefined} & \text{if } w' \text{ is not a postfix of } w \end{cases}$$

Since the function *out* overrides the previous value of x, the semantics above correctly models the behaviour of actual implementations of matching where the last word matched against r is assigned to x for $(r)_x$.

Example 4. The following illustrates the semantics of repetition and capturing.

$$\begin{aligned} \mathcal{N}[\![(a|b)_x{}^*]\!]ab &= \big[\big(\epsilon, [x \mapsto b]\big), \big(b, [x \mapsto a]\big), (ab, \bot)\big] \\ \mathcal{N}[\![(a|b)_x{}^{*?}]\!]ab &= \big[(ab, \bot), \big(b, [x \mapsto a]\big), \big(\epsilon, [x \mapsto b]\big)\big] \end{aligned}$$

To formulate the soundness for the extended semantics, we lift the monad morphism *set* to the composed monad as follows.

$$\mathit{set}\big(\big[(w_1, \rho_1), (w_2, \rho_2), \ldots, (w_n, \rho_n)\big]\big) = \{w_1, w_2, \ldots, w_n\}$$

This is also a monoid morphism from the list monoid with $[\,]$ and $+\!+$ to the set monoid with $\emptyset$ and $\cup$. From these facts on the lifted *set*, it is easily shown that Theorem 1 also holds for the extended parser for capturing.

It is guaranteed that $w w'^{-1}$ in the definition of $\mathcal{N}[\![(r_1)_x]\!]w$ is well-defined as follows. The function $(\lambda w'.\bot[x \mapsto w w'^{-1}])$ is applied to a word $w' \in \mathit{set}(\mathcal{N}[\![r_1]\!]w)$. Theorem 1 implies that w' is a postfix of w. Thus, $w w'^{-1}$ is well-defined.

3.4. Other features in regular expression matching

There are other features in implementations of regular expression matching. Although we have not developed their translation into transducers, their semantics can be described by extending the nondeterministic parser.

We consider the following three features found in many implementations of regular expression matching.

$$r ::= \cdots \mid \backslash x \mid (r_1)^{\mathsf{atomic}} \mid (r_1)^{\mathsf{lookahead}}$$

- Backreference $\backslash x$ only matches the subword that matched with $(r)_x$ most recently.
- Atomic grouping $(r_1)^{\mathsf{atomic}}$ discards the branches of nondeterministic parsing except for the first.
- Lookahead $(r_1)^{\mathsf{lookahead}}$ is almost the same as $(r_1)^{\mathsf{atomic}}$. However, $(r_1)^{\mathsf{lookahead}}$ matches the empty word and restarts the matching.

The detailed explanation of these features can be found in [9].

Among these features, the semantics of atomic grouping and lookahead can be formalized by introducing the function *atomic* for the list monad.

$$\begin{aligned} \mathit{atomic}\, m &= \mathsf{case}\ m\ \mathsf{of}\ e :: l \Rightarrow [e] \mid [\,] \Rightarrow [\,] \\ \mathcal{N}[\![(r_1)^{\mathsf{atomic}}]\!]\, w &= \mathit{atomic}\,\big(\mathcal{N}[\![r_1]\!]\, w\big) \\ \mathcal{N}[\![(r_1)^{\mathsf{lookahead}}]\!]\, w &= \mathit{atomic}\,\big(\mathcal{N}[\![r_1]\!]\, w\big) \gg\!= \lambda v.\mathit{unit}\, w \end{aligned}$$

On the other hand, we cannot describe $\mathcal{N}[\![\backslash x]\!]$ with the output monad since it must *look up* the environment to use the subword that matched with $(r)_x$. However we can formulate the semantics of $\mathcal{N}[\![\backslash x]\!]$ by the state (and list) monad of type $Env \to (\alpha \times Env)\ list$.

$$\mathcal{N}[\![\backslash x]\!]w \;=\; \lambda\rho.\text{if } \rho(x)^{-1}w \text{ is defined then } \left[\left(\rho(x)^{-1}w, \rho\right)\right] \text{ else } [\,]$$

Please note that it is impossible to translate backreference into transducers because the language of a regular expression containing backreferences may not be regular.

4. Deterministic parser using lookahead

We derive a deterministic parser using lookahead from the nondeterministic one. The option monad is used to represent the failure of the matching in the deterministic parser. Although the parser in Section 3.2 is highly nondeterministic, we can choose the appropriate branch from possible choices if we can look ahead the rest of the word. The deterministic parser we obtain in this section basically coincides with the linear time parser by Frisch and Cardelli [10]. However, their representation is quite different from ours and we derive it from a different semantics in a more intuitive manner.

We formulate a deterministic parser $\mathcal{D}[\![r]\!]_{r_c}$ for regular expression r and continuation regular expression r_c: r_c is the regular expression that the rest of the word obtained by $\mathcal{D}[\![r]\!]_{r_c}$ will be matched with. We formulate $\mathcal{D}[\![r]\!]_{r_c}$ by using the option monad: the type $\alpha\ option$ has constructors *None* of type $\alpha\ option$ and *Some* of type $\alpha \to \alpha\ option$. The unit and bind functions for the option monad are defined as follows:

$$\begin{aligned} unit\, x &= Some\, x \\ m \gg\!= f &= \text{case } m \text{ of } Some\, w \Rightarrow f\, w \mid None \Rightarrow None \end{aligned}$$

We derive the definition of $\mathcal{D}[\![r]\!]_{r_c}$ by using the following corollary of Theorem 1. The function of *filter* has type $(\alpha \to bool) \to \alpha\ list \to \alpha\ list$ and is defined in the standard manner.

Corollary 1. *filter* $(\lambda w'.w' \in L(r_c))\ (\mathcal{N}[\![r]\!]w) \neq [\,]$ *iff* $w \in L(rr_c)$.

Let us consider $\mathcal{N}[\![r_1|r_2]\!]w$. From the corollary, if $w \in L(r_1r_c)$, it is guaranteed that the matching w against r_1 leaves at least one word w' such that $w' \in L(r_c)$. Otherwise, it does not leave such a word. Thus, we divide the definition of $\mathcal{D}[\![r_1|r_2]\!]_{r_c}w$ into two cases, $w \in L(r_1r_c)$ or $w \notin L(r_1r_c)$, and obtain the following definition.

$$\mathcal{D}[\![r_1|r_2]\!]_{r_c}w = \begin{cases} \mathcal{D}[\![r_1]\!]_{r_c}w & \text{if } w \in L(r_1r_c) \\ \mathcal{D}[\![r_2]\!]_{r_c}w & \text{if } w \notin L(r_1r_c) \end{cases}$$

By applying the similar case analysis, we derive the following definition of $\mathcal{D}[\![r]\!]_{r_c}w$. (The domain of $\mathcal{D}[\![r]\!]_{r_c}$ is $L(rr_c)$ from the corollary above.)

$$\begin{aligned} \mathcal{D}[\![r]\!]_{r_c} &:: \Sigma^* \to \Sigma^*\ option \\ \mathcal{D}[\![\epsilon]\!]_{r_c}w &= \begin{cases} unit\, w & \text{if } w \in L(r_c) \\ None & \text{if } w \notin L(r_c) \end{cases} \\ \mathcal{D}[\![c]\!]_{r_c}w &= \begin{cases} unit\, w' & w = cw' \text{ and } w' \in L(r_c) \\ None & \text{otherwise} \end{cases} \\ \mathcal{D}[\![r_1r_2]\!]_{r_c}w &= \mathcal{D}[\![r_1]\!]_{r_2r_c}w \gg\!= \lambda w'.\mathcal{D}[\![r_2]\!]_{r_c}w' \\ \mathcal{D}[\![r_1^*]\!]_{r_c}w &= \begin{cases} \mathcal{D}[\![r_1]\!]_{r_1^*r_c}w \gg\!= \lambda w'.\mathcal{D}[\![r_1^*]\!]_{r_c}w' & \text{if } w \in L(r_1r_1^*r_c) \\ \text{if } w \in L(r_c) \text{ then } unit\, w \text{ else } None & \text{if } w \notin L(r_1r_1^*r_c) \end{cases} \\ \mathcal{D}[\![r_1^{*?}]\!]_{r_c}w &= \begin{cases} unit\, w & \text{if } w \in L(r_c) \\ \mathcal{D}[\![r_1]\!]_{r_1^*r_c}w \gg\!= \lambda w'.\mathcal{D}[\![r_1^{*?}]\!]_{r_c}w' & \text{if } w \notin L(r_c) \end{cases} \end{aligned}$$

For $\mathcal{D}[\![r_1r_2]\!]_{r_c}$, the continuation for r_1 must be r_2r_c instead of r_c because the rest of the word is matched with r_2r_c.

In order to formulate the correctness of the deterministic parser, we introduce functions *first* and *find* defined as follows:

$$\begin{aligned} first &:: \alpha\ list \to \alpha\ option \\ find &:: (\alpha \to bool) \to \alpha\ list \to \alpha\ option \\ first\, l &= \text{case } l \text{ of } x :: l' \Rightarrow Some\, x \mid [\,] \Rightarrow None \\ find\, p\, m &= first\, (filter\, p\, m) \end{aligned}$$

These functions are not monad morphisms, but they satisfy the following properties with respect to the bind function.

Lemma 1. $filter\ p\ (m \gg\!=_{list}\ f) = m \gg\!=_{list}\ \lambda x.(filter\ p\ (f\ x))$.

Lemma 2. $first\ (m \gg\!=_{list}\ f) = find\ (\lambda x.f\ x \neq [\,])\ m \gg\!=_{option}\ \lambda x.first\ (f\ x)$.

The nondeterministic parser $\mathcal{N}[\![r]\!]$ uses the convention that the first element of the list has the highest priority. Then, the deterministic parser $\mathcal{D}[\![r]\!]_{r_c}$ is correct in the following sense.

Theorem 2. $\mathcal{D}[\![r]\!]_{r_c}\, w = find\ (\lambda w'.w' \in L(r_c))\ (\mathcal{N}[\![r]\!]\, w)$.

Proof. By induction on the lexicographic order of the structure of r and the length of w. We only show the case of $r = r_1^*$ where $\epsilon \notin L(r_1)$. We have $w \in L(r_1 r_1^* r_c)$ iff $filter\ (\lambda w'.w' \in L(r_c))\ \mathcal{N}[\![r_1 r_1^*]\!]\, w \neq [\,]$ by Corollary 1.

Case $w \in L(r_1 r_1^* r_c)$:

$$
\begin{aligned}
&find\ \big(\lambda w'.w' \in L(r_c)\big)\ \mathcal{N}[\![r_1^*]\!]\, w \\
&\quad = find\ \big(\lambda w'.w' \in L(r_c)\big)\ \mathcal{N}[\![r_1 r_1^*]\!]\, w \\
&\quad = first\ \big(filter\ \big(\lambda w'.w' \in L(r_c)\big)\ \big(\mathcal{N}[\![r_1]\!]\, w \gg\!= \lambda w'.\mathcal{N}[\![r_1^*]\!]\, w'\big)\big) \\
&\quad = first\ \big(\mathcal{N}[\![r_1]\!]\, w \gg\!= \lambda w'.\big(filter\ \big(\lambda w'.w' \in L(r_c)\big)\ \mathcal{N}[\![r_1^*]\!]\, w'\big)\big) \quad \text{(by Lemma 1)} \\
&\quad = find\ \big(\lambda v.filter\ \big(\lambda w'.w' \in L(r_c)\big)\ \big(\mathcal{N}[\![r_1^*]\!]\, v\big) \neq [\,]\big)\ \mathcal{N}[\![r_1]\!]\, w \\
&\qquad \gg\!= \lambda v.first\ \big(filter\ \big(\lambda w'.w' \in L(r_c)\big)\ \big(\mathcal{N}[\![r_1^*]\!]\, v\big)\big) \quad \text{(by Lemma 2)} \\
&\quad = find\ \big(\lambda v.v \in L\big(r_1^* r_c\big)\big)\ \mathcal{N}[\![r_1]\!]\, w \\
&\qquad \gg\!= \lambda v.find\ \big(\lambda w'.w' \in L(r_c)\big)\ \big(\mathcal{N}[\![r_1^*]\!]\, v\big) \quad \text{(by Corollary 1)} \\
&\quad = \mathcal{D}[\![r_1]\!]_{r_1^* r_c}\, w \gg\!= \lambda w'.\mathcal{D}[\![r_1^*]\!]_{r_c}\, w' \quad \text{(by I.H.)}
\end{aligned}
$$

Case $w \notin L(r_1 r_1^* r_c)$:

$$
\begin{aligned}
&find\ \big(\lambda w'.w' \in L(r_c)\big)\ \mathcal{N}[\![r_1^*]\!]\, w \\
&\quad = find\ \big(\lambda w'.w' \in L(r_c)\big)\ (unit\ w) \\
&\quad = \begin{cases} unit\ w & \text{if } w \in L(r_c) \\ None & \text{if } w \notin L(r_c) \end{cases}
\end{aligned}
$$

□

To extend the deterministic parser for capturing, we compose the option monad with the output monad as the nondeterministic parser. The option monad composed with the output monad for type Env has type $(\alpha \times Env)\ option$ for type parameter α. The functions $unit$, $bind$, and out are extended as before. Then, the definition of the $\mathcal{D}[\![(r_1)_x]\!]_{r_c}$ is given as follows:

$$\mathcal{D}[\![(r_1)_x]\!]_{r_c}\, w = out\ \big(\mathcal{D}[\![r]\!]_{r_c}\, w\big)\ \big(\lambda w'.\bot[x \mapsto w w'^{-1}]\big)$$

The correctness of $\mathcal{D}[\![r]\!]_{r_c}$ is shown by lifting the functions $filter$ and $find$ on the list and option monads to those composed with the output monad as follows.

$$
\begin{aligned}
&filter' &&:: (\alpha \to bool) \to (\alpha \times Env)\ list \to (\alpha \times Env)\ list \\
&find' &&:: (\alpha \to bool) \to (\alpha \times Env)\ list \to (\alpha \times Env)\ option \\
&filter'\ p\ m &&= filter\ (p \circ fst)\ m \\
&find'\ p\ m &&= first\ (filter'\ p\ m)
\end{aligned}
$$

where $fst\ (x, y) = x$. The lifted functions $filter'$ and $find'$ satisfy Lemmas 1 and 2 on the composed monads. This is sufficient to extend the correctness proof.

5. The problematic case

As we discussed earlier, if we have $\epsilon \in L(r)$ for r^*, the nondeterministic parser in Section 3.2 causes non-terminating unfolding $r^* = rr^*|\epsilon$. Thus, implementations of regular expression matching must avoid this unfolding. There are two strategies to avoid this non-terminating unfolding found in implementations of regular expression matching. The first strategy taken by Perl is to stop unfolding when ϵ matches with r unfolded from r^*. This strategy can be formulated in our nondeterministic parser $\mathcal{N}[\![r]\!]$ as follows:

$$\mathcal{N}[\![r_1^*]\!]w = \left(\mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\text{if } w = w' \text{ then } \mathit{unit}\, w \text{ else } \mathcal{N}[\![r_1^*]\!]w'\right) +\!+ \mathit{unit}\, w$$
$$\mathcal{N}[\![r_1^{*?}]\!]w = \mathit{unit}\, w +\!+ \left(\mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\text{if } w = w' \text{ then } \mathit{unit}\, w \text{ else } \mathcal{N}[\![r_1^*]\!]w'\right)$$

When r_1 matches ϵ, the parser stops the unfolding of r_1^*.

The other strategy is to restrict the matching of r for r^*: only nonempty words match with r for r^*. This strategy is taken by JavaScript [6] and the regular expression library RE2 [24], and is studied by Frisch and Cardelli [10]. This strategy is formulated by slightly revising the definition: [ ] instead of $\mathit{unit}\, w$ in the then-branch.

$$\mathcal{N}[\![r_1^*]\!]^{\mathsf{JS}}w = \left(\mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\text{if } w = w' \text{ then } [\,] \text{ else } \mathcal{N}[\![r_1^*]\!]w'\right) +\!+ \mathit{unit}\, w$$

Example 5. Let us consider $(\epsilon|a)^*$.

$$\begin{aligned}\mathcal{N}[\![(\epsilon|a)^*]\!]a &= [a] +\!+ \mathcal{N}[\![(\epsilon|a)^*]\!]\epsilon +\!+ [a]\\ &= [a,\epsilon,\epsilon,a]\\ \mathcal{N}[\![(\epsilon|a)^*]\!]^{\mathsf{JS}}a &= [\epsilon,a]\end{aligned}$$

In this paper, we focus on the first strategy because this strategy is taken by most scripting languages including Perl, PHP, and Python.[2]

To develop the deterministic parser that corresponds to the nondeterministic parser of the above definition, we first introduce a nondeterministic parser $\mathcal{N}[\![r]\!]'$ that consumes at least one letter. It can be recursively defined by using $\mathcal{N}[\![r]\!]$ as follows:

$$\begin{aligned}\mathcal{N}[\![\epsilon]\!]'w &= [\,]\\ \mathcal{N}[\![c]\!]'w &= \begin{cases}\mathit{unit}\, w' & \text{if } w = cw'\\ [\,] & \text{otherwise}\end{cases}\\ \mathcal{N}[\![r_1r_2]\!]'w &= \mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\text{if } w' = w \text{ then } \mathcal{N}[\![r_2]\!]'w' \text{ else } \mathcal{N}[\![r_2]\!]w'\\ \mathcal{N}[\![r_1|r_2]\!]'w &= \mathcal{N}[\![r_1]\!]'w +\!+ \mathcal{N}[\![r_2]\!]'w\\ \mathcal{N}[\![r_1^*]\!]'w &= \mathcal{N}[\![r_1]\!]'w \gg\!= \lambda w'.\mathcal{N}[\![r_1^*]\!]w'\\ \mathcal{N}[\![r_1^{*?}]\!]'w &= \mathcal{N}[\![r_1]\!]'w \gg\!= \lambda w'.\mathcal{N}[\![r_1^{*?}]\!]w'\\ \mathcal{N}[\![(r_1)_x]\!]'w &= \mathit{out}\left(\mathcal{N}[\![r_1]\!]'w\right)\left(\lambda w'.\bot\left[x \mapsto w{w'}^{-1}\right]\right)\end{aligned}$$

In order to parse a word for r_1r_2, it first parses a word for r_1. If $w' \neq w$, it applies $\mathcal{N}[\![r_2]\!]$ because one letter has already been consumed. Otherwise, it applies $\mathcal{N}[\![r_2]\!]'$ instead of $\mathcal{N}[\![r_2]\!]$, in order to consume at least one letter. The definition of $\mathcal{N}[\![r_1^*]\!]'$ can be understood by considering the following equation.

$$\begin{aligned}\mathcal{N}[\![r_1^*]\!]'w &= \mathcal{N}[\![r_1]\!]'w \gg\!= \lambda w'.\mathcal{N}[\![r_1^*]\!]w'\\ &= \mathcal{N}[\![r_1]\!]w \gg\!= \lambda w'.\text{if } w' = w \text{ then } [\,] \text{ else } \mathcal{N}[\![r_1^*]\!]w'\end{aligned}$$

If $\mathcal{N}[\![r_1]\!]w$ unfolded from $\mathcal{N}[\![r_1^*]\!]'w$ consumes no letter, it stops unfolding in Perl's strategy, and thus $\mathcal{N}[\![r_1^*]\!]'w$ consumes no letter and returns []. This behaviour is equivalent to the definition of $\mathcal{N}[\![r_1^*]\!]'w$. The correctness of $\mathcal{N}[\![r]\!]'$ is formulated by the *filter* function.

Theorem 3. $\mathcal{N}[\![r]\!]'w = \mathit{filter}\,(\lambda w'.w' \neq w)\,\mathcal{N}[\![r]\!]w$.

In order to treat the problematic case, we reformulate $\mathcal{D}[\![r]\!]_{r_c}$ by introducing $\mathcal{D}[\![r]\!]'_{r_c}$ that corresponds to $\mathcal{N}[\![r]\!]'$. The revised definition of $\mathcal{D}[\![r]\!]_{r_c}$ and $\mathcal{D}[\![r]\!]'_{r_c}$ is given in Fig. 1 where $\widehat{L}$ denotes $L - \{\epsilon\}$ for $L \subseteq \Sigma^*$. The definitions for the cases not appearing in the figure, e.g. $\mathcal{D}[\![r_1r_2]\!]_{r_c}$, remain the same. Let us consider the definition of $\mathcal{D}[\![r_1r_2]\!]'_{r_c}w$.

- If $w \in \widehat{L(r_2)}L(r_c)$, it is guaranteed that $\mathcal{D}[\![r_2]\!]'_{r_c}w$ succeeds. Thus, we first apply $\mathcal{D}[\![r_1]\!]_{r_2r_c}w$, and then choose $\mathcal{D}[\![r_2]\!]'_{r_c}w$ or $\mathcal{D}[\![r_2]\!]_{r_c}w$ depending on whether $\mathcal{D}[\![r_1]\!]_{r_2r_c}w$ consumed at least one letter or not. Please note that if $w \in L(r_1r_2r_c)$, then $\mathcal{D}[\![r_1]\!]_{r_2r_c}w = \mathit{Some}\, w'$ and $w' \in L(r_2r_c)$ for some w'. Then, both $\mathcal{D}[\![r_2]\!]_{r_c}w'$ and $\mathcal{D}[\![r_2]\!]'_{r_c}w'$ are guaranteed to succeed.
- If $w \notin \widehat{L(r_2)}L(r_c)$, the matching of r_1 must consume at least one letter. Thus, we obtain the definition.

The correctness of the deterministic parser is extended as follows. Although we need a finer analysis to treat if-expressions in $\mathcal{N}[\![r^*]\!]w$ and $\mathcal{N}[\![r^{*?}]\!]w$, the proof is basically a straightforward extension of Theorem 2.

[2] The behaviour of problematic regular expressions are not described in the manuals of these languages as far as we know. We checked the behaviour by executing matching of several problematic regular expressions in these languages. The behaviour in JavaScript is described in [6].

$$\mathcal{D}[\![r_1^*]\!]_{r_c} w = \begin{cases} \mathcal{D}[\![r_1]\!]_{r_1^* r_c} w \gg\!= \lambda w'.\text{if } w = w' \text{ then } \mathit{unit}\, w' \text{ else } \mathcal{D}[\![r_1^*]\!]_{r_c} w' & \text{if } w \in L(r_c) \wedge w \in L(r_1 r_1^* r_c) \\ \mathit{unit}\, w & \text{if } w \in L(r_c) \wedge w \notin L(r_1 r_1^* r_c) \\ \mathcal{D}[\![r_1]\!]'_{r_1^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r_1^*]\!]_{r_c} w' & \text{if } w \notin L(r_c) \end{cases}$$

$$\mathcal{D}[\![r_1^{*?}]\!]_{r_c} w = \begin{cases} \mathit{unit}\, w & \text{if } w \in L(r_c) \\ \mathcal{D}[\![r_1]\!]'_{r_1^{*?} r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r_1^{*?}]\!]_{r_c} w' & \text{if } w \notin L(r_c) \end{cases} \quad \mathcal{D}[\![\epsilon]\!]'_{r_c} w = \mathit{None} \quad \mathcal{D}[\![c]\!]'_{r_c} w = \begin{cases} \mathit{unit}\, w' & \text{if } w = cw' \wedge w' \in L(r_c) \\ \mathit{None} & \text{otherwise} \end{cases}$$

$$\mathcal{D}[\![r_1|r_2]\!]'_{r_c} w = \begin{cases} \mathcal{D}[\![r_1]\!]'_{r_c} w & \text{if } w \in \widehat{L(r_1)} L(r_c) \\ \mathcal{D}[\![r_2]\!]'_{r_c} w & \text{if } w \notin \widehat{L(r_1)} L(r_c) \end{cases}$$

$$\mathcal{D}[\![r_1 r_2]\!]'_{r_c} w = \begin{cases} \mathcal{D}[\![r_1]\!]_{r_2 r_c} w \gg\!= \lambda w'.\text{if } w = w' \text{ then } \mathcal{D}[\![r_2]\!]'_{r_c} w' \text{ else } \mathcal{D}[\![r_2]\!]_{r_c} w' & \text{if } w \in \widehat{L(r_2)} L(r_c) \\ \mathcal{D}[\![r_1]\!]'_{r_2 r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r_2]\!]_{r_c} w' & \text{if } w \notin \widehat{L(r_2)} L(r_c) \end{cases}$$

$$\mathcal{D}[\![r_1^*]\!]'_{r_c} w = \mathcal{D}[\![r_1]\!]'_{r_1^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r_1^*]\!]_{r_c} w'$$

$$\mathcal{D}[\![r_1^{*?}]\!]'_{r_c} w = \mathcal{D}[\![r_1]\!]'_{r_1^{*?} r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r_1^{*?}]\!]_{r_c} w'$$

$$\mathcal{D}[\![(r_1)_x]\!]'_{r_c} w = \mathit{out}\left(\mathcal{D}[\![r]\!]'_{r_c} w\right)\left(\lambda w'.\bot[x \mapsto w w'^{-1}]\right)$$

Fig. 1. Deterministic parser for the problematic case.

Theorem 4.

$$\mathcal{D}[\![r]\!]_{r_c} w = \mathit{find}\left(\lambda w'.w' \in L(r_c)\right)\left(\mathcal{N}[\![r]\!] w\right)$$
$$\mathcal{D}[\![r]\!]'_{r_c} w = \mathit{find}\left(\lambda w'.w' \in L(r_c)\right)\left(\mathcal{N}[\![r]\!]' w\right)$$

The semantics based on the deterministic parser is also useful to prove the generic equivalence of regular expressions. Let us consider the Example 5 again. We find that the semantics of $(\epsilon|a)^*$ is rather similar to that of $a^{*?}$.

$$\mathcal{N}[\![a^{*?}]\!] a = [a, \epsilon]$$

Actually, we can show that $(\epsilon|r)^*$ and $r^{*?}$ are equivalent in the following sense [3]: $\mathcal{D}[\![(\epsilon|r)^*]\!]_{r_c} w = \mathcal{D}[\![r^{*?}]\!]_{r_c} w$ for any regular expression r and any $w \in \Sigma^*$.[4] It is shown by induction on the length of w.

Case: $w \in L(r_c)$. We have $\mathcal{D}[\![r^{*?}]\!]_{r_c} w = \mathcal{D}[\![(\epsilon|r)^*]\!]_{r_c} w = \mathit{unit}\, w$.
Case: $w \notin L(r_c)$.

$$\begin{aligned} \mathcal{D}[\![r^{*?}]\!]_{r_c} w &= \mathcal{D}[\![r]\!]'_{r^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r^{*?}]\!]_{r_c} w' \\ &= \mathcal{D}[\![r]\!]'_{r^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![(\epsilon|r)^*]\!]_{r_c} w' \quad \text{(by I.H.)} \\ &= \mathcal{D}[\![\epsilon|r]\!]'_{r^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![(\epsilon|r)^*]\!]_{r_c} w' \\ &= \mathcal{D}[\![\epsilon|r]\!]'_{(\epsilon|r)^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![(\epsilon|r)^*]\!]_{r_c} w' \\ &= \mathcal{D}[\![(\epsilon|r)^*]\!]_{r_c} w \end{aligned}$$

Although the equivalence of regular expression matching can be mechanically decided by the translation to transducers as we will describe later, it is not applicable to generic equivalence such as above that includes metavariables over regular expressions.

6. Construction of transducers

We construct a transducer with regular lookahead corresponding to regular expression r based on the definition of $\mathcal{D}[\![r]\!]_{r_c}$. Since a transducer with regular lookahead can be converted into one without lookahead, we can precisely represent regular expression matching with a transducer without lookahead. We first present our construction, and then prove the correctness of the construction.

[3] This shows that lazy repetition can be expressed by greedy repetition. This fact is not discussed elsewhere as far as we know.
[4] This equivalence does not hold for the nondeterministic parser: $\mathcal{N}[\![(\epsilon|r)^*]\!] w \neq \mathcal{N}[\![r^{*?}]\!] w$. In order to discuss the equivalence based on the nondeterministic parser we need to consider the equivalence relation on list obtained by $x +\!+ y +\!+ x \approx x +\!+ y$. We leave the investigation of this approach to the future work.

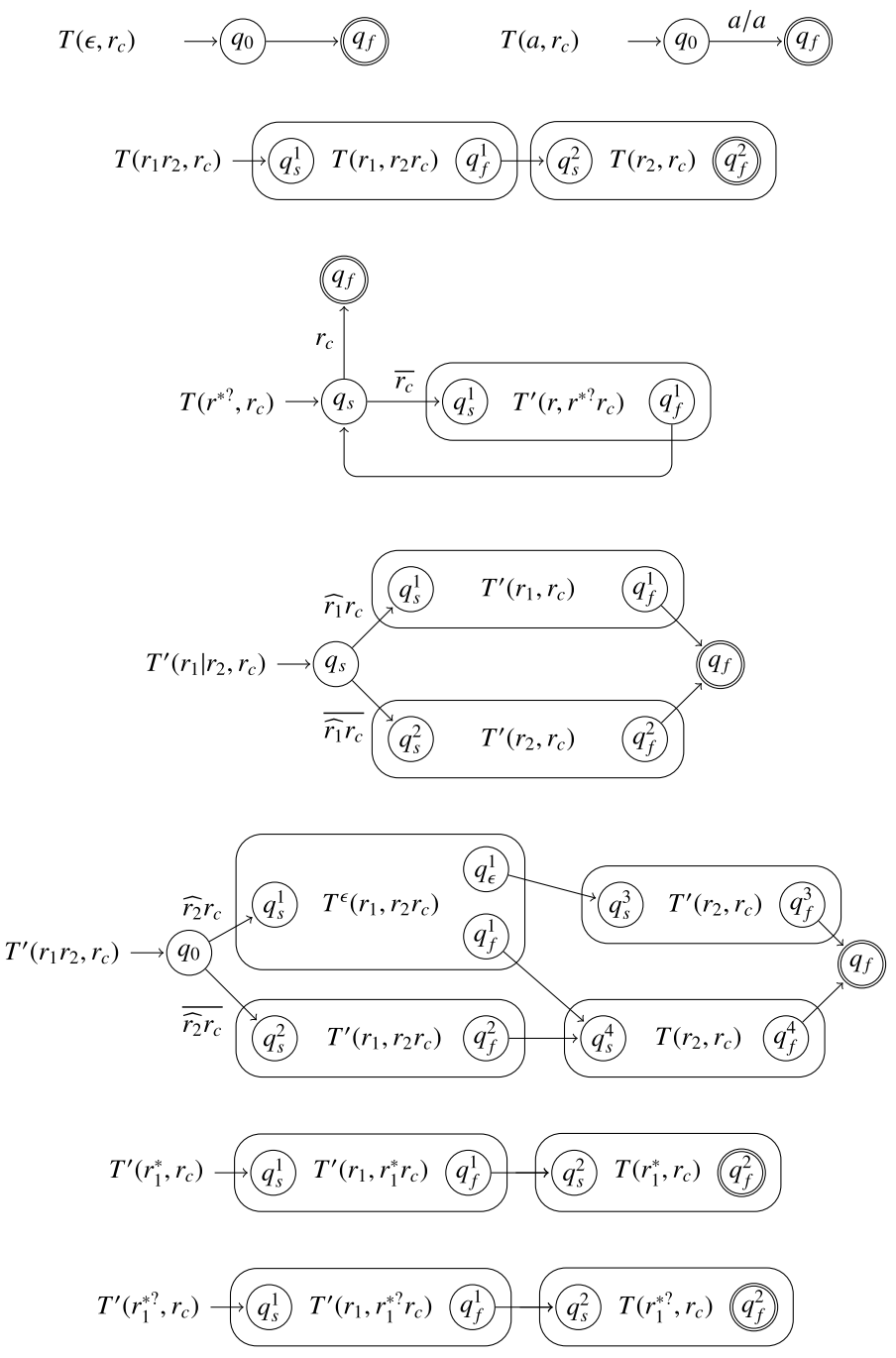


Fig. 2. Construction of transducers.

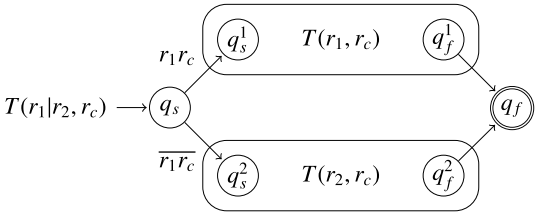
6.1. Construction

We construct transducers $T(r, r_c)$ and $T'(r, r_c)$ for regular expression r and continuation regular expression r_c. The construction is derived from the definition of $\mathcal{D}[\![r]\!]_{r_c}$ and $\mathcal{D}[\![r]\!]'_{r_c}$, and a refinement of Thompson's construction of ϵ-NFA from a regular expression [27]. In the transition diagrams in this section, we adapt the following convention: label r on a transition abbreviates $\epsilon/\epsilon/L(r)$ and a transition without any label denotes ϵ-transition without output and lookahead.

We illustrate the construction for some cases below, and the rest of the construction is given in Fig. 2. We first consider $T(r_1|r_2, r_c)$ that simulates $\mathcal{D}[\![r_1|r_2]\!]_{r_c}$. The definition of $\mathcal{D}[\![r_1|r_2]\!]_{r_c}$ was given as follows:

$$\mathcal{D}[\![r_1|r_2]\!]_{r_c} w = \begin{cases} \mathcal{D}[\![r_1]\!]_{r_c} w & \text{if } w \in L(r_1 r_c) \\ \mathcal{D}[\![r_2]\!]_{r_c} w & \text{if } w \notin L(r_1 r_c) \end{cases}$$

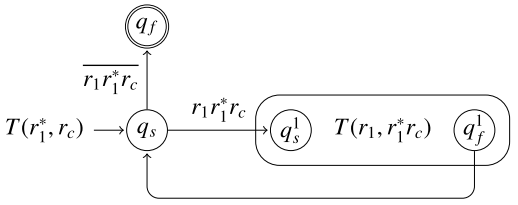
From this definition, we derive the following construction of $T(r_1|r_2, r_c)$.



The only difference from the standard construction is lookahead to simulate the case-analysis in the definition $\mathcal{D}[\![r_1|r_2]\!]_{r_c}$.

Secondly, we consider the construction for r_1^* where $\epsilon \notin L(r_1)$. The construction of the transducer is derived as follows:

$$\mathcal{D}[\![r_1^*]\!]_{r_c} w = \begin{cases} \mathcal{D}[\![r_1]\!]_{r_1^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r_1^*]\!]_{r_c} w' & \text{if } w \in L(r_1 r_1^* r_c) \\ \text{if } w \in L(r_c) \text{ then } \mathit{unit}\ w \text{ else } \mathit{None} & \text{if } w \notin L(r_1 r_1^* r_c) \end{cases}$$

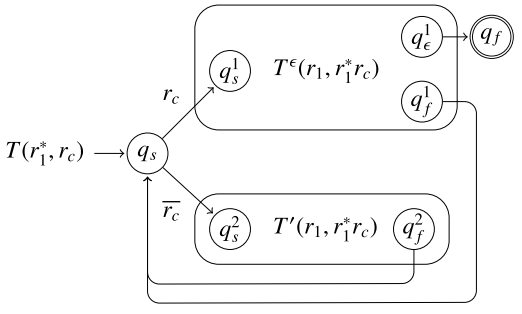


This is also a simple refinement of the standard construction. We again need lookahead to simulate the case-analysis in $\mathcal{D}[\![r_1^*]\!]_{r_c}$. The case distinction of $w \in L(r_c)$ in $\mathcal{D}[\![r_1^*]\!]_{r_c} w$ does not appear in the construction because the check corresponding to it is realized by the transducer that the rest of the word is applied to.

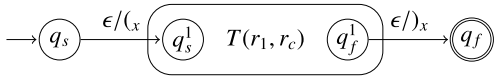
The construction diverts from the standard construction when we consider the problematic case. Let us consider the construction for r_1^* when $\epsilon \in L(r_1)$. In the definition of the deterministic parser, we have lookahead of the form $r_1 r_1^* r_c$. However, this lookahead is redundant by the following reason: if $w \in L(r_c)$, then we have $w \in L(r_1 r_1^* r_c)$ from $\epsilon \in L(r_1)$. Then, the definition of $\mathcal{D}[\![r_1^*]\!]_{r_c}$ can be simplified as follows:

$$\mathcal{D}[\![r_1^*]\!]_{r_c} w = \begin{cases} \mathcal{D}[\![r_1]\!]_{r_1^* r_c} w \gg\!= \lambda w'.\text{if } w = w' \text{ then } \mathit{unit}\ w' \text{ else } \mathcal{D}[\![r_1^*]\!]_{r_c} w' & \text{if } w \in L(r_c) \\ \mathcal{D}[\![r_1]\!]'_{r_1^* r_c} w \gg\!= \lambda w'.\mathcal{D}[\![r_1^*]\!]_{r_c} w' & \text{if } w \notin L(r_c) \end{cases}$$

In order to derive the construction for the case $w \in L(r_c)$, we introduce a variant, $T^\epsilon(r, r_c)$, of $T(r, r_c)$. It is because we have to check if $\mathcal{D}[\![r_1]\!]_{r_1^* r_c}$ has consumed at least one letter. The transducer $T^\epsilon(r, r_c)$ has two final states: q_ϵ^1 and q_f^1. If no letter is consumed in $T(r, r_c)$, then the transition reaches q_ϵ^1, and otherwise q_f^1. This transducer can be constructed by taking the product of $T(r, r_c)$ and an automaton that distinguishes the empty word from nonempty words. With $T^\epsilon(r_1, r_c)$, we can translate the definition above as follows:



The construction for capturing $(r_1)_x$ is shown below: each subword matched with $(r_1)_x$ is enclosed by $(_x$ and $)_x$ in the output of the transducer.



The transitions $\epsilon/(_x$ and $\epsilon/)_x$ have no lookahead and abbreviate $\epsilon/(_x/\Sigma^*$ and $\epsilon/)_x/\Sigma^*$, respectively. The construction of $T'((r_1)_x, r_c)$ is basically the same.

It is clear from the construction that the transducers $T(r, r_c)$ and $T'(r_1, r_c)$ always output a balanced word. Let G be a context-free grammar with the following production rules.

$$S \to \epsilon \mid cS \mid (_x S)_x S$$

where $c \in \Sigma$ and x is any variable occurring in r. Then, $q_s \xrightarrow[T(r,r_c)]{w/v|w'} q_f$ implies $v \in L(G)$ where q_s and q_f are the start and the final states of $T(r, r_c)$, respectively.

By the decomposition of the output using the grammar G, we can extract the environment from the output of the transducer. Let α be the homomorphism eliminating all parentheses indexed by variables.

$$\alpha(c) = \begin{cases} c & \text{if } c \in \Sigma \\ \epsilon & \text{otherwise} \end{cases}$$

It has type $\Sigma \to \Sigma^*$, and is naturally extended to type $\Sigma^* \to \Sigma^*$. The following lemma guarantees that the output produced by the transducers is identical to the input if we ignore indexed parentheses.

Lemma 3. *Let q_s and q_f be the start and the final states of $T(r, r_c)$ or $T'(r, r_c)$, respectively. If $q_s \xrightarrow[T(r,r_c)]{w/v|w'}{}^* q_f$ or $q_s \xrightarrow[T'(r,r_c)]{w/v|w'}{}^* q_f$, then $\alpha(v) = w$.*

For $v \in L(G)$, the environment π_v extracted from v is defined as follows.

$$\pi_v = \begin{cases} \bot & \text{if } v = \epsilon \\ \pi_{v'} & \text{if } v = cv' \\ \pi_{v_1} \uplus \bot[x \mapsto \alpha(v_1)] \uplus \pi_{v_2} & \text{if } v = (_x v_1)_x v_2 \end{cases}$$

Then, the construction of the transducers is correct in the following sense where ϵ is applied as the initial continuation.

Theorem 5. *Let q_s and q_f be the start and the final states of $T(r, \epsilon)$, respectively.*

(a) *If $\mathcal{D}[\![r]\!]_\epsilon w = \mathit{Some}(\epsilon, \rho')$, then $q_s \xrightarrow[T(r,\epsilon)]{w/v|\epsilon}{}^* q_f$ and $\rho' = \pi_v$ for some v.*

(b) *If $q_s \xrightarrow[T(r,\epsilon)]{w/v|\epsilon}{}^* q_f$, then $\mathcal{D}[\![r]\!]_\epsilon w = \mathit{Some}(\epsilon, \pi_v)$.*

The correctness above is proved by the following generalized lemma. In the propositions (b) and (b′), the hypothesis $w' \in L(r_c)$ is necessary because $T(\epsilon, r_c)$ and $T(a, r_c)$ in Fig. 2 do not check that the rest of the word is in $L(r_c)$. For the theorem above, we have $\epsilon \in L(\epsilon)$ for $q_s \xrightarrow[T(r,\epsilon)]{w/v|\epsilon}{}^* q_f$.

Lemma 4. *Let q_s and q_f be the start and the final states of $T(r, r_c)$ and $T'(r, r_c)$, respectively.*

(a) *If $\mathcal{D}[\![r]\!]_{r_c} w = \mathit{Some}(w', \rho')$, then $q_s \xrightarrow[T(r,r_c)]{ww'^{-1}/v|w'}{}^* q_f$ and $\rho' = \pi_v$ for some v.*

(b) *If $q_s \xrightarrow[T(r,r_c)]{w/v|w'}{}^* q_f$ and $w' \in L(r_c)$, then $\mathcal{D}[\![r]\!]_{r_c} ww' = \mathit{Some}(w', \pi_v)$.*

(a′) *If $\mathcal{D}[\![r]\!]'_{r_c} w = \mathit{Some}(w', \rho')$, then $q_s \xrightarrow[T'(r,r_c)]{ww'^{-1}/v|w'}{}^* q_f$ and $\rho' = \pi_v$ for some v.*

(b′) *If $q_s \xrightarrow[T'(r,r_c)]{w/v|w'}{}^* q_f$ and $w' \in L(r_c)$, then $\mathcal{D}[\![r]\!]'_{r_c} ww' = \mathit{Some}(w', \pi_v)$.*

Proof. By induction on the lexicographic order of the structure of r and the length of w. We only show two main cases of (a).

Case $r = r_1^*$ where $\epsilon \in L(r_1)$:

The following holds for some w'', ρ'', and ρ'''.

$$\mathcal{D}[\![r_1]\!]_{r_1^* r_c} w = \mathit{Some}\left(w'', \rho''\right) \tag{1}$$

$$\text{if } w = w'' \text{ then } \mathit{unit}\, w'' \text{ else } \mathcal{D}[\![r_1^*]\!]_{r_c} w'' = \mathit{Some}\left(w', \rho'''\right) \tag{2}$$

$$\rho' = \rho'' \uplus \rho''' \tag{3}$$

We only show the case where $w \in L(r_c)$. The other case is proved in a similar manner.
Let q_s^0 and q_f^0 be the start and the final states of $T(r_1, r_1^* r_c)$. By induction hypothesis on (1),

$$q_s^0 \xrightarrow[T(r_1, r_1^* r_c)]{}{}^{*}\ q_f^0 \quad \text{with } ww''^{-1}/v_1|w''$$

and $\rho'' = \pi_{v_1}$. Let q_s^1, q_ϵ^1, and q_f^1 be the start and the final states of $T^\epsilon(r_1, r_1^* r_c)$.

Subcase $w = w''$: From (2) and (3), we have $w = w' = w''$, $\rho''' = \bot$, and $\rho' = \rho''$. By construction of $T^\epsilon(r_1, r_1^* r_c)$, we have

$$q_s^1 \xrightarrow[T^\epsilon(r_1, r_1^* r_c)]{}{}^{*}\ q_\epsilon^1 \quad \text{with } ww'^{-1}/v_1|w'$$

Thus, $q_s \xrightarrow[T^\epsilon(r_1^*, r_c)]{}{}^{*}\ q_f$ with $ww'^{-1}/v_1|w'$ and $\rho' = \pi_{v_1}$.

Subcase $w \neq w''$. We have $|w''| < |w|$ from (1). By construction of $T^\epsilon(r_1, r_1^* r_c)$, we have $q_s^1 \xrightarrow[T^\epsilon(r_1, r_1^* r_c)]{}{}^{*}\ q_f^1$ with $ww''^{-1}/v_1|w''$. Since $|w''| < |w|$, we also have $q_s \xrightarrow[T(r_1^*, r_c)]{}{}^{*}\ q_f$ with $w''w'^{-1}/v_2|w'$ and $\rho''' = \pi_{v_2}$ by induction hypothesis. From these and the construction of $T(r_1^*, r_c)$, we have

$$q_s \xrightarrow[T(r_1^*, r_c)]{}{}^{*}\ q_f \quad \text{with } ww'^{-1}/v_1 v_2|w'$$

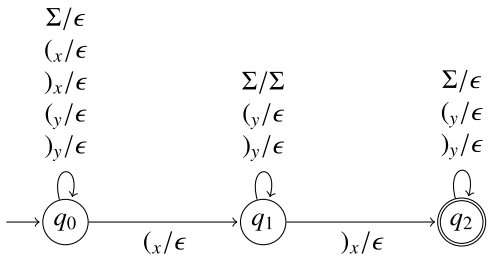
and $\rho' = \rho'' \uplus \rho''' = \pi_{v_1} \uplus \pi_{v_2} = \pi_{v_1 v_2}$.

Case $r = (_1)_x$: Let f be $\lambda v.\bot[x \mapsto w v^{-1}]$. We have $\mathcal{D}[\![r_1]\!]_{r_c} w = \mathit{Some}(w', \rho'')$ and $\rho' = \rho'' \uplus \bot[x \mapsto w w'^{-1}]$ for some ρ''. Then, $q_s^1 \xrightarrow[T(r_1, r_c)]{}{}^{*}\ q_f^1$ with $ww'^{-1}/v_1|w'$ and $\rho'' = \pi_{v_1}$ by induction hypothesis. By the construction, $q_s \xrightarrow[T((r_1)_x, r_c)]{}{}^{*}\ q_f$ with $ww'^{-1}/(_x v_1)_x|w'$. Finally, the following holds by Lemma 3.

$$\pi_{(_x v_1)_x} = \pi_{v_1} \uplus \bot\left[x \mapsto \alpha(v_1)\right] = \rho'' \uplus \bot\left[x \mapsto w w'^{-1}\right] = \rho' \qquad \square$$

6.2. Application of the transducer

In applications of the transducer constructed from regular expression matching, we actually need the value $\rho(x)$ for the environment ρ obtained by the matching instead of the word v that represents capturing with parentheses and satisfies $\rho = \pi_v$. Actually, $\rho(x)$ can easily be obtained from v by applying the transducer that extracts the last subword enclosed with $(_x$ and $)_x$. The transducer T_x below realizes this. Note that Σ on the arrow denotes all letters in Σ, and y is a variable such that $y \neq x$.



We assume that there is no nested capturing for the same variable in a regular expression.[5]

[5] In the concrete syntax of regular expressions, capturing is not explicitly named, but named by its position. Thus, this condition is implicitly satisfied. It is also possible to extend the transducer for nested capturing because the depth of nesting is determined by the regular expression.

One of the practical applications of transducers obtained from regular expressions is the equivalence checking of two regular expressions r_1 and r_2. It can be decided by using $T(r_1, \epsilon)$, $T(r_2, \epsilon)$, and T_x. We say that r_1 and r_2 are equivalent with respect to x if $L(r_1) = L(r_2)$ and the following holds for any $w \in \Sigma^*$.

- $\mathcal{D}[\![r_1]\!]_\epsilon w = \mathit{Some}(\epsilon, \rho_1)$ and $\mathcal{D}[\![r_2]\!]_\epsilon w = \mathit{Some}(\epsilon, \rho_2)$ implies $\rho_1(x) = \rho_2(x)$.

This equivalence can be decided by checking equivalence between $T_x \circ T(r_1, \epsilon)$ and $T_x \circ T(r_2, \epsilon)$. It is decidable because they are functional transducers [25].

Example 6. The following are the regular expressions matching a pseudo comment of the C language shown in [9]. Because `*` at the start and end of a C comment is a special character of regular expressions, `*` is replaced with `x` to simplify the regular expressions. The second regular expression is obtained by optimizing the first one by unrolling the repetition as described in [9]: the second one is much faster than the first one. If we have lazy repetition, we can use the third one, which is faster than the second in recent versions of Perl [9]. We have checked the equivalence of these expressions by using our implementation discussed in Section 7. The equivalence is checked by putting each regular expression `r` in `.*?(r).*` to extract the substring matching with `r`.

```
/x([^x]|x+[^/x])*x+/
/x[^x]*x+([^/x][^x]*x+)*/
/x.*?x/
```

6.3. State complexity

For the non-problematic case, the number of the states in $T(r, \epsilon)$ is clearly in $O(|r|)$. There are at most $O(|r|)$ lookaheads in $T(r, \epsilon)$. Thus, the size of transducer without lookahead is in $O(|r| \cdot 2^{c|r|^2})$ as we discussed in Section 2.2. However, we can do better for $T(r, \epsilon)$: all lookahead regular languages appearing in $T(r, \epsilon)$ can be accepted by some state of ϵ-NFA constructed from r. Frisch and Cardelli [10] adopted similar construction. By constructing the preprocessing transducer from this ϵ-NFA, we obtain the transducer without lookahead for r of size $O(|r| \cdot 2^{c|r|})$.

For the problematic case, the number of the states in $T(r, \epsilon)$ can grow exponentially with respect to the depth of the repetition. We can inductively show that the numbers of states of $T(r, r_c)$ and $T'(r, r_c)$ are not more than $4^{|r|}$, and the size of $T^\epsilon(r, r_c)$ is not more than $2 \cdot 4^{|r|}$. In $T(r, \epsilon)$, we have lookaheads of the form $\widehat{L(r')}L(r'_c)$. These languages cannot be represented by the states of ϵ-NFA constructed from r. However, it is easily shown that they are represented by the states of an ϵ-NFA of size $O(|r|^2)$. Then, we obtain the transducer without lookahead for r of size $O(4^{|r|} \cdot 2^{c|r|^2})$.

7. Implementation and experimental results

We have implemented the translation from a regular expression to a transducer for the subset of Perl-style regular expressions [23]. The subset includes not only the features described in the Section 3.1, but also other most commonly-used features such as `r+`, `|r?|`, `and`character classes. Those extended features of regular expressions in Perl are supported by a translation into regular expressions defined in Section 3.1. The following are some examples of the translation.

```
/(a|b)+/      ==>  (a|b)(a|b)*
/a|b/i        ==>  (a|A)|(b|B)
/(a|b){1,3}/  ==>  (a|b)((a|b)((a|b)|ϵ)|ϵ)
```

In regular expression `/a|b/i`, `i` is a modifier that makes the regular expression matching case-insensitive. The regular expression `/(a|b){1,3}/` matches the repetition of `a` or `b` at least one but not more than three times. Anchors in regular expressions `^` and `$` are translated in the following manner: `/^r$/` is treated as just `r`, and `/r/` is treated as `.*?r.*` where `.` matches any character. This correctly simulates the behaviour of matching where a regular expression matches the leftmost subword that matches with the expression.

For our experiments, we have collected regular expressions used to capture matching substrings from five popular PHP programs: PHP-Fusion, phpMyAdmin, SquirrelMail, TorrentFlux, XOOPS. The regular expression matching in PHP is based on the PCRE (Perl Compatible Regular Expression) library [22]. We obtained 212 regular expressions from those programs by excluding regular expression matching where a regular expression is dynamically constructed by string operations. Among 212 regular expressions, only one contains a feature not supported by our implementation: negative lookbehind assertion. A negative lookbehind assertion is similar to lookahead, but it succeeds if the word behind the current position does not match the assertion. By excluding this, we have 211 regular expressions in our test set.

The following summarizes basic facts about our test set.

- No regular expression contains r^* or $r^{*?}$ where $\epsilon \in L(r)$.
- Lazy repetition $r^{*?}$ appears only as `.*?`.

Table 1
Detailed results for some regular expressions.

	Regular expression size	Without elimination		With elimination	
		Lookaheads	States	Lookaheads	States
r_1	29	6	75	4	21
r_2	40	7	91	3	91
r_3	47	10	78	3	78
r_4	51	10	136	3	136
r_5	53	8	114	2	55
r_6	61	5	252	4	252
r_7	79	8	237	6	235
r_8	104	8	381	4	330
r_9	117	17	1529	4	1463
r_{10}	154	12	677	0	76
r_{11}	173	15	522	4	316

	Regular expression
r_1	`|^([^:]+)://([^:/]+)(:[\d]+)*(.*)|`
r_2	`/charset\s*=\s*"?([a-z0-9\-\.\_]+)"?/i`
r_3	`/^(?:(?:vedr|sv|re|aw|fw|fwd|\[\w\]):\s*)*\s*(.*)$/si`
r_4	`/([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})/`
r_5	`/^\\*\s+([0-9]+)\s+FETCH.*UID\s+([0-9]+)\s+/iAU`
r_6	`'<textinput(|[^>]*[^/])>(.*?)</textinput>'si`
r_7	`/^.*(content-transfer-encoding:)\s*(\w+-?(\w+)?).*/i`
r_8	`@(.*)([[:space:]](LIMIT (.*)|PROCEDURE (.*)| FOR UPDATE|LOCK IN SHARE MODE))@i`
r_9	`/\* NAMESPACE +(\( *\(.+\) *\)|NIL) +(\( *\(.+\) *\)|NIL) + \(\( *\(.+\) *\)|NIL)/i`
r_{10}	`@^SHOW[[:space:]]+(VARIABLES|(FULL[[:space:]]+)?` `PROCESSLIST|STATUS|TABLE|GRANTS|CREATE|LOGS|DATABASES|FIELDS)@i`
r_{11}	`/^create\s+(?:temp|temporary)?trigger\s+(?:if\s+not\s+exists\s+)?` `.*(before|after)?\s+(insert|update|delete)/Uims`

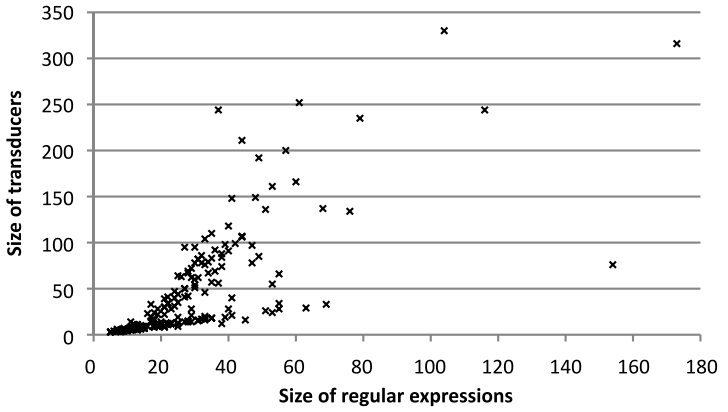


Fig. 3. Size of the transducers.

- Greedy repetition r^* is mostly used in the very simple form where r matches some set of characters. The regular expressions r_1 and r_3 in Table 1 are only the exceptions in our test set.

The transducer constructed by our translation often contains redundant lookaheads. For example, consider the regular expression $r_1|r_2$. If $L(r_1 r_c) \cap L(r_2 r_c) = \emptyset$, it is clear that eliminating lookaheads in $T(r_1|r_2, r_c)$ does not change the behaviour of the transducer. The lookaheads in the construction of repetition can also be eliminated if $L(r_1 r_1^* r_c) \cap L(r_c) = \emptyset$. We have implemented this to eliminate the redundant lookahead from the transducer.

Fig. 3 shows the sizes of the transducers obtained by applying our translation to the regular expressions in the test set. We measure the size of a regular expression after translating it into a basic regular expression. There is one result excluded from Fig. 3, which does not fit in the graph. For the regular expression r_9 of size 117 in Table 1, we obtained a transducer of size 1463. Although our translation has an exponential state complexity, the experimental results do not generally show the exponential blow-up.

Table 1 shows the detailed results for some interesting regular expressions. The results without and with the lookahead elimination above are shown in the table. The lookahead elimination often reduces the number of lookaheads in the transducers. However, it is not so effective for reducing the size of the transducer. It will be partly because the prepro-

cessing transducer is minimized in our implementation. The preprocessing transducer is deterministic, and thus can be minimized [21].

We have also conducted preliminary experiments for problematic regular expressions. For example, we obtained the transducers of size 11 and 4 for $(a^*b^*c^*)^*$ and a^{***}, respectively. For $a^{**\cdots*}$, we observed the exponential blow-up in the size of transducers as the depth of repetition increases.

8. Related work

Frisch and Cardelli formulated the semantics of regular expression matching based on the tree language of a regular expression and the ordering between trees [10]. Although their semantics is quite elegant, it is less intuitive to understand the semantics of existing implementations of regular expression matching. Furthermore, it cannot be extended for features such as atomic grouping and lookahead. The disambiguation strategy they considered for the problematic case differs from ours as we discussed in Section 5: in their disambiguation strategy, r in r^* cannot match ϵ. They developed a linear time matching algorithm: $O(|w| \times |r|)$ for a word w and a regular expression r. For the case of non-problematic regular expressions, the deterministic parser in Section 4 basically coincides with their parser. On the other hand, for the problematic case, we have obtained a deterministic parser that is different from theirs in a non-trivial manner. However, our construction is not fully satisfactory because the number of states in $T(r, r_c)$ can be exponential with respect to $|r|$. This contrasts with their linear time parser that can handle the problematic case. We would like to investigate whether the construction for the problematic case of Frisch and Cardelli can be revised for the Perl-compatible semantics in future work.

Xi showed a backtrack-based implementation of regular expression matching in a functional programming language [32]. This implementation is closely related to our nondeterministic parser in a sense that if we exchange the list monad in our parser with the monad for backtracking, we basically obtain his implementation. The semantics of the repetition he considered is that of JavaScript.

Vansummeren investigated the various disambiguation strategies of regular expression matching and formulated their semantics as deductive systems [28]. He considered two disambiguation strategies: POSIX and the first and longest match. It is discussed that the disambiguation of Perl is obtained by revising some rules of the first and longest match. However, the revision does not correctly handle the problematic case we discussed in this paper. The semantics given by Vansummeren is also limited in the sense that it does not consider capturing inside repetition.

Fischer, Huch, and Wilke have developed a Haskell library for regular expression matching that supports weighted regular expressions [8]: weight can be an arbitrary semiring and it is shown that it can be used to simulate the leftmost longest strategy. They started from the specification of weighted regular expression matching and derived an efficient implementation. Although their derivation of the efficient implementation is interesting, it is not clear whether it can be revised for the Perl-compatible strategy. It is because weight should be a semiring, but the Perl-compatible strategy cannot be directly described by a semiring because the addition of the semiring corresponding to alternation must be commutative.

Brabrand and Thomsen proposed typed and unambiguous pattern matching on strings and implemented it in Java [2]. They presented a syntax-directed analysis of ambiguity and introduced a restriction operator r_1/r_2 to resolve ambiguity of regular expressions. The use of restriction operator is closely related to our use of lookahead when we obtain the deterministic parser. They also introduced disambiguation directives to resolve ambiguity in a regular expression.

Laurikari introduced NFAs with tagged transitions for an efficient implementation of regular expression matching with capturing for the POSIX-compliant strategy [14], where each transition has a priority to simulate a disambiguation strategy and tags are utilized to store the start and end positions of a subword matching with a capturing expression. NFAs with tagged transitions are similar to our usage of transducers in the sense that it outputs the information of capturing during matching. However, they do not directly correspond to well-investigated extension of automata such as transducers. Furthermore, the semantics of regular expression matching is not formally discussed in the paper.

Several axiomatizations of the equivalence of regular expressions have been proposed by Salomma [26], Kozen [13], Grabmayer [11], Henglein and Nielsen [12]. Although these axiomatizations make it possible to prove the equivalence of regular expressions in the standard sense, they are not adequate for deciding the equivalence of regular expression matching in the presence of a disambiguation strategy and capturing. For example, the alternation is commutative if we only consider the language of a regular expression, but it is not commutative in general for regular expression matching with capturing and disambiguation strategies. The following three axioms of the weak equivalence do not hold for Perl-style regular expressions.

$$r_1|r_2 = r_2|r_1 \qquad r_0(r_1|r_2) = r_0r_1|r_0r_2 \qquad r^* = rr^*|\epsilon$$

9. Conclusion

We have formulated the semantics of the regular expression matching as a nondeterministic parser by using the list and output monad, and derived the deterministic parser by equational reasoning. We then derived the construction of transducers with regular lookahead, which can be converted into those without lookahead by an existing technique. We have implemented our translation and applied it to regular expressions found in popular PHP programs. The regular expressions obtained from those programs show that problematic regular expressions rarely appear in programs and repetitions are

mostly used in a very simple form. The experimental results on the test set do not show exponential blow-up and suggest that it will be applicable in analysis and verification of string-manipulating programs.

Checking equivalence of the regular expression matching is an interesting application of the translation. Although the equivalence of regular expressions is easily decided in the theory regular languages, it is non-trivial in the presence of the disambiguation strategy and capturing. It is sometimes necessary to optimize a regular expression into more efficient one in a non-trivial manner. Such an optimization can be verified by our translation.

The use of monads makes the semantics and equational reasoning on it modular. It is basically an application of modular interpreters based on monads [15,29]. However, it is especially well-suited for regular expression matching based on the Perl-compatible strategy and the equational reasoning based on it is smoothly conducted in this paper.

We are planning to extend our translation for other constructs of regular expressions such as atomic and lookahead expressions. The main difficulty in treating these features is that $L(r)$ cannot be defined in the standard manner: e.g. $L(r) \neq L((r)^{\mathsf{atomic}})$ in general.

References

[1] J. Berstel, Transductions and Context-Free Languages, Teubner Studienbucher, 1979.
[2] C. Brabrand, J.G. Thomsen, Typed and unambiguous pattern matching on strings using regular expressions, in: Proceedings of 12th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP), 2010, pp. 243–254.
[3] R. Cox, Regular expression matching can be simple and fast, http://swtch.com/~rsc/regexp/, 2007.
[4] R. Cox, Regular expression matching: the virtual machine approach, http://swtch.com/~rsc/regexp/, 2009.
[5] R. Cox, Regular expression matching in the wild, http://swtch.com/~rsc/regexp/, 2010.
[6] ECMAScript language specification, 5th ed., http://www.ecmascript.org/, 2009.
[7] J. Engelfriet, Top-down tree transducers with regular look-ahead, Mathematical Systems Theory 10 (1) (1977) 289–303.
[8] S. Fischer, F. Huch, T. Wilke, A play on regular expressions, in: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, 2010, pp. 357–368.
[9] J.E.F. Friedl, Mastering Regular Expressions, 3rd ed., O'Reilly, 2006.
[10] A. Frisch, L. Cardelli, Greedy regular expression matching, in: Proceedings of the 31st International Colloquium on Automata, Languages and Programming, in: LNCS, vol. 3142, 2004, pp. 618–629.
[11] C. Grabmayer, Using proofs by coinduction to find "traditional" proofs, in: Proceedings of 1st Conference on Algebra and Coalgebra in Computer Science, in: LNCS, vol. 3629, 2005, pp. 175–193.
[12] F. Henglein, L. Nielsen, Regular expression containment: Coinductive axiomatization and computational interpretation, in: Conference Record of POPL 2011: 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2011, pp. 385–398.
[13] D. Kozen, Kleene algebra with tests, Information and Computation 19 (3) (1997) 427–443.
[14] V. Laurikari, Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions, in: Proceedings of the 7th International Symposium on String Processing and Information Retrieval, IEEE, 2000, pp. 181–187.
[15] S. Liang, P. Hudak, M.P. Jones, Monad transformers and modular interpreters, in: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1995, pp. 333–343.
[16] G.H. Mealy, A method for synthesizing sequential circuits, Bell System Technical Journal 34 (5) (1955) 1045–1079.
[17] Y. Minamide, Static approximation of dynamically generated Web pages, in: Proceedings of the 14th International World Wide Web Conference, ACM Press, 2005, pp. 432–441.
[18] Y. Minamide, Y. Sakuma, A. Voronkov, Translating regular expression matching into transducers, in: Proceedings of International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (2010), 2011, pp. 107–115.
[19] E. Moggi, Computational lambda-calculus and monads, in: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS), 1989, pp. 14–23.
[20] E. Moggi, An abstract view of programming languages, Course Notes, 1989.
[21] M. Mohri, Minimization algorithms for sequential transducers, Theoretical Comput. Sci. 234 (1–2) (2000) 177–201.
[22] PCRE – Perl compatible regular expressions, http://www.pcre.org/.
[23] perlre – Perl regular expressions, http://perldoc.perl.org/perlre.html.
[24] re2: an efficient, principled regular expression library, http://code.google.com/p/re2/.
[25] J. Sakarovitch, Elements of Automata Theory, Cambridge University Press, 2009.
[26] A. Salomma, Two complete axiom systems for the algebra of regular events, J. ACM 13 (1966) 148–169.
[27] K. Thompson, Regular expression search algorithm, Communications of the ACM 11 (6) (1968) 419–422.
[28] S. Vansummeren, Type inference for unique pattern matching, ACM Transactions on Programming Languages and Systems 28 (3) (2006) 389–428.
[29] P. Wadler, The essence of functional programming, in: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1992, pp. 1–14.
[30] P. Wadler, Comprehending monads, Mathematical Structures in Computer Science 2 (1992) 461–493.
[31] G. Wassermann, D. Yu, et al. Dynamic test input generation for web applications, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, 2008, pp. 249–260.
[32] H. Xi, Dependent types for program termination verification, in: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, 2001, pp. 231–242.