Electronic Notes in Theoretical Computer Science 75 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume75.html 19 pages

Regular Expression Types for Strings in a Text Processing Language (Extended Abstract)

Naoshi Tabuchi Eijiro Sumii Akinori Yonezawa [1]

Department of Computer Science,
Graduate School of Information Science and Technology,
University of Tokyo

Abstract

We present λ^{re}, a minimal functional calculus with *regular expression types* for strings, in order to establish a theoretical foundation of using regular expressions as types of strings in text processing languages. The major technical novelties in this paper (with respect to other work such as XDuce) are (1) the use of regular expression *effects* to statically analyze the shape of the output of an even diverging program and (2) the treatment of `as`-patterns in non-tail positions. We expect that our approach will be of help for bringing the merits of static typing into scripting languages such as Perl, Python, and Ruby without sacrificing too much of their expressiveness and flexibility.

1 Introduction

Background

Scripting languages such as Perl [18], Python [10], and Ruby [11,17] are widely believed to be useful for so-called rapid application development, in particular for manipulating strings or text data as in web programming. However, while it may indeed be easier to just *write* programs in these languages, it actually tends to be more difficult to *debug* and *maintain* such programs. A typical experience of programming in those languages is as follows. (1) The programmer writes code at a tremendous speed. (2) He or she runs the program and gets a runtime error—or, worse, no output at all. (3) He/she spends a lot of time in pain on debugging. Furthermore, it is also likely that bugs are found only after the program is deployed because of some boundary cases that were overlooked in testing. Under open environments, bugs of this kind can

[1] E-mail: {tabee,sumii,yonezawa}@yl.is.s.u-tokyo.ac.jp

©2003 Published by Elsevier Science B. V.

lead to critical security holes, a major example being cross-site scripting vulnerabilities [1] of CGI programs that forgot to escape HTML meta-characters in the input (and the output).

In general, an effective approach to preventing such dynamic errors is static typing. However, standard type systems as in ML are not so effective solutions to the above problems, not only because they are often incompatible with legacy programs (or legacy *programmers*), but also because all text data are usually given a single type, `string`, in ordinary type systems. Of course, it is also possible to write a program that parses the input into some more structural data type, operates on values of that data type, and pretty-prints the result into the output. However, doing so would spoil the ease of text processing in such scripting languages as above.

Our Approach

In order to mitigate this dilemma between handiness and robustness, we propose *regular expression types for strings*, that is, refining the type of strings by the shape of the strings as regular expressions. For example, the string constants `a`, `aa`, `aaa` can all have type $\mathtt{a}^*$. For another example, the recursive function $f(n) = \mathsf{if}\ n \leq 0\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \mathtt{a} \mathbin{\hat{}} f(n-1) \mathbin{\hat{}} \mathtt{b}$, where ε denotes the empty string (that is, the string of length 0) and $\hat{}$ denotes string concatenation, may have type $\mathsf{int} \to \mathtt{a}^*\mathtt{b}^*$. We expect that, in more complex programs, this form of information about the shape of strings would be of help to programmers for debugging and maintenance of programs that process text data.

Why do we adopt regular expressions rather than, say, context-free grammars? If we adopted "context-free grammar types," then the codomain of the function f in the above example could be a more precise type $\{\mathtt{a}^n\mathtt{b}^n \mid n = 0, 1, 2, \ldots\}$, for instance. One reason why we have chosen regular expressions is that they have concise syntax and semantics with which people are just familiar. A more important reason is that the class of regular languages is closed for many basic operations such as intersection, union, difference, and quotients. Also, most fundamental problems about regular languages—such as equivalence and inclusion—are decidable. These good properties of regular languages are essential for effective type checking in our type system. In contrast, other formal languages are not so nice – for instance, it is well-known that equivalence (and therefore inclusion) of context-free languages is undecidable [5].

Our Contributions

In order to present the above ideas in a formal yet concise manner, we define a minimal functional calculus, λ^{re}, with regular expression types for strings. Although this calculus is extremely small, we believe that many of the results can be extended to more practical languages, thanks to the flexibility of regular expressions. For instance, any expression that evaluates to a string

can at worst be typed .* (meaning "any string"), and in many cases one can do better.

It is not a new idea to use regular expressions or other formal languages to describe shapes of data or behavior of programs. The most relevant work that we know of is XDuce [8,6], a programming language for processing XML documents, which gives regular expression types to XML documents as (labeled) *trees*. Theoretically, since strings are just a special case of trees (consider the encoding of lists by cons cells in Scheme), their work may seem to subsume ours. Nevertheless, we think that the present work deserves a separate paper for the following two reasons. (1) Our focus on strings rather than trees in general significantly simplifies the technicalities and thereby clarifies the presentation. (2) We also extend some of their results, as we shall see in detail in the following sections. The major extensions are (i) regular expression types for *effects* and (ii) as-patterns in non-tail positions.

This paper does not give a full type inference/reconstruction algorithm for the type system (although Section 4 presents a partial type inference method for variable bindings in regular expression pattern matching and Section 6 gives a rough idea toward a full type inference scheme), let alone its efficient implementation. They are important in practice in order for programmers to benefit from the type system without too much extra effort. However, these are complex issues by themselves and left for future work at this moment.

Overview

The rest of this paper is organized as follows. Section 2 gives the syntax of λ^{re} and Section 3 defines its operational semantics. Section 4 presents the type system and Section 5 shows examples of programming in λ^{re}. Section 6 concludes with discussion of related work and future work.

Because of the limitation on length, details of some proofs are omitted. They are available on the Web at `http://www.yl.is.s.u-tokyo.ac.jp/~tabee/xperl/`.

2 Syntax

The syntax of terms in λ^{re} is given in Figure 1, assuming a countably infinite number of variables x, y, z, f, g, h, etc. The first three forms, where $\mathsf{fix}(f, x, M)$ denotes a possibly recursive function f defined as $f(x) = M$, are standard. We write $\lambda x.\, M$ for $\mathsf{fix}(f, x, M)$ where f does not appear free in M and $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$ for $(\lambda x.\, M_2)M_1$. We furthermore write $M_1; M_2$ for $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2$ where x does not appear free in M_2. The definition of free variables is also standard and therefore omitted.

As primitives for string manipulation, we take the following four constructs: (1) constant strings s for having strings at all, (2) string concatenation $M_1 \hat{} M_2$ for constructing a new string from old ones, (3) pattern matching $\mathsf{match}\ M\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n$ over strings by regular expressions

$$
\begin{array}{lll}
M \text{ (term)} ::= & & \\
\quad x & & \text{(variable)} \\
\quad \mid \mathsf{fix}(f, x, M) & & \text{(recursive function)} \\
\quad \mid M_1 M_2 & & \text{(function application)} \\
\quad \mid s & & \text{(constant string)} \\
\quad \mid M_1 \hat{} M_2 & & \text{(string concatenation)} \\
\quad \mid \mathsf{match}\ M\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n & & \text{(pattern matching)} \\
\quad \mid \mathsf{print}\ M & & \text{(output)} \\
P \text{ (pattern)} ::= & & \\
\quad s & & \text{(constant string)} \\
\quad \mid (P_1 \mid P_2) & & \text{(choice)} \\
\quad \mid x\ \mathsf{as}\ P & & \text{(variable binding)} \\
\quad \mid P^* & & \text{(repetition)} \\
\quad \mid P_1 P_2 & & \text{(sequence)}
\end{array}
$$

Fig. 1. Syntax of Terms and Patterns

for destructing strings, and (4) printing $\mathsf{print}\ M$ of strings to the output as a side effect. As we shall see in Section 4, this is the target of our regular expression effect system.

The syntax of patterns is similar to that of standard regular expressions except for variable bindings $x\ \mathsf{as}\ P$. Like as-patterns in ML, it first matches an input string s against the pattern P and then binds s to the variable x. We write $var(P)$ for the set of all variables that appear in P.

Standard data structures such as lists and pairs can easily be encoded in this language. See section 5 for examples of such encoding.

3 Operational Semantics

The meaning of terms in λ^{re} is defined in Figure 2 as a small-step, reduction semantics—rather than big-step, evaluation semantics—in order to treat effects of diverging programs as well as terminating ones. It is given by a relation $M_1 \xrightarrow{s} M_2$, meaning "term M_1 reduces to term M_2 in one step, printing string s to the output." The evaluation order is fixed to be call-by-value and left-to-right for simplifying the treatment of effects.

Rules (R-App), (R-Cat), and (R-Ctx) represent standard meanings of (possibly recursive) function application, string concatenation, and evaluation con-

$$
\begin{array}{rll}
v \text{ (value)} & ::= & \mathsf{fix}(f, x, M) \\
& \mid & s \\
C[\,] \text{ (evaluation context)} & ::= & [\,]M \\
& \mid & v[\,] \\
& \mid & [\,] \mathbin{\hat{}} M \\
& \mid & v \mathbin{\hat{}} [\,] \\
& \mid & \mathsf{match}\ [\,]\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n \\
& \mid & \mathsf{print}\ [\,]
\end{array}
$$

$$\frac{}{\mathsf{fix}(f, x, M)v \xrightarrow{\varepsilon} [v/x][\mathsf{fix}(f, x, M)/f]M}(\text{R-App})$$

$$\frac{}{s_1 \mathbin{\hat{}} s_2 \xrightarrow{\varepsilon} s_1 + s_2}(\text{R-Cat}) \qquad \frac{}{\mathsf{print}\ s \xrightarrow{s} \varepsilon}(\text{R-Print})$$

$$\frac{\forall i < m.\, s \rhd P_i \Rightarrow \bot \quad s \rhd P_m \Rightarrow \theta \quad \theta \neq \bot}{\mathsf{match}\ s\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n \xrightarrow{\varepsilon} \theta M_m}(\text{R-Match})$$

$$\frac{M_1 \xrightarrow{s} M_2}{C[M_1] \xrightarrow{s} C[M_2]}(\text{R-Ctx})$$

Fig. 2. Operational Semantics of Terms

texts. Rule (R-Print) says that the term $\mathsf{print}\ s$ prints the string s to the output and reduces to a dummy value (the empty string ε). Rule (R-Match) defines the meaning of pattern matching in λ^{re}, using the relation $s \rhd P \Rightarrow \theta$. Intuitively, $s \rhd P \Rightarrow \theta$ means that matching the string s against the pattern P either fails (if $\theta = \bot$) or yields the substitution θ from variables to strings (if $\theta \neq \bot$). Thus, (R-Match) says that the term $\mathsf{match}\ s\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n$ reduces to θM_m where P_m is the *first* pattern that matches the string s (and yields a substitution θ).

The pattern matching relation $s \rhd P \Rightarrow \theta$ is defined in Figure 3. We write $s \rhd P$ if $s \rhd P \Rightarrow \theta$ for some $\theta \neq \bot$. Intuitively, the rules describe a (naive) pattern matching algorithm in a bottom-up manner. Basically, it is defined by structural induction on the syntax of the pattern P. Rules (M-Const-Succ) and (M-Const-Fail) are for constant strings s, (M-Choice-Fst) and (M-Choice-Snd) are for choices $P_1 \mid P_2$, (M-Bind) is for variable bindings $x\ \mathsf{as}\ P$, and

$$\frac{}{s \rhd s \Rightarrow \emptyset}\text{(M-Const-Succ)} \qquad \frac{s \neq s'}{s \rhd s' \Rightarrow \bot}\text{(M-Const-Fail)}$$

$$\frac{s \rhd P_1 \Rightarrow \theta \quad \theta \neq \bot}{s \rhd P_1 \mid P_2 \Rightarrow \theta}\text{(M-Choice-Fst)}$$

$$\frac{s \rhd P_1 \Rightarrow \bot \quad s \rhd P_2 \Rightarrow \theta}{s \rhd P_1 \mid P_2 \Rightarrow \theta}\text{(M-Choice-Snd)}$$

$$\frac{s \rhd P \Rightarrow \theta}{s \rhd x \text{ as } P \Rightarrow \theta \uplus \{x \mapsto s\}}\text{(M-Bind)} \qquad \frac{s \rhd PP^* \mid \varepsilon \Rightarrow \theta}{s \rhd P^* \Rightarrow \theta}\text{(M-Rep)}$$

$$\frac{s_2^{-1}s_1 \rhd P \Rightarrow \theta}{s_1 \rhd s_2 P \Rightarrow \theta}\text{(M-Seq-Const-Succ)}$$

$$\frac{s_2^{-1}s_1 \text{ not exist}}{s_1 \rhd s_2 P \Rightarrow \bot}\text{(M-Seq-Const-Fail)}$$

$$\frac{s \rhd P_1 P_3 \mid P_2 P_3 \Rightarrow \theta}{s \rhd (P_1 \mid P_2) P_3 \Rightarrow \theta}\text{(M-Seq-Choice)}$$

$$\frac{y \notin var(P_1 P_2) \quad s_1 \rhd P_1(y \text{ as } P_2) \Rightarrow \theta \uplus \{y \mapsto s_2\}}{s_1 \rhd (x \text{ as } P_1) P_2 \Rightarrow \theta \uplus \{x \mapsto s_1 s_2^{-1}\}}\text{(M-Seq-Bind)}$$

$$\frac{s \rhd (P_1 {P_1}^* \mid \varepsilon) P_2 \Rightarrow \theta}{s \rhd {P_1}^* P_2 \Rightarrow \theta}\text{(M-Seq-Rep)} \qquad \frac{s \rhd P_1(P_2 P_3) \Rightarrow \theta}{s \rhd (P_1 P_2) P_3 \Rightarrow \theta}\text{(M-Seq-Seq)}$$

Fig. 3. Operational Semantics of Patterns

(M-Rep) is for repetitions P^*. The notation $\theta \uplus \{x \mapsto s\}$ in (M-Bind) denotes the extension θ' of θ such that $\theta'(x) = s$ and $\theta'(y) = \theta(y)$ for $y \neq x$ (with $\bot \uplus \{x \mapsto s\}$ being $\bot$), while $\emptyset$ denotes the identity substitution.

Note that the rule (M-Choice-Snd) applies only when (M-Choice-Fst) does not, which leads to the so-called *first-match* policy. Note also that (M-Rep) expands P^* to $PP^* \mid \varepsilon$ rather than $\varepsilon \mid PP^*$. In combination with the first-

match policy above, this leads to the so-called *longest-match* policy.

When P is a sequence P_1P_2, the rules furthermore separate into six cases (M-Seq-...) according to the form of P_1, in order to circumvent non-deterministically dividing the input string s (into two strings s_1 and s_2, so that s_1 matches P_1 and s_2 matches P_2). Each of the six rules rewrites a sequence pattern P_1P_2 according to the meaning of its first half P_1, so that P_1 becomes "smaller" in the premise than in the conclusion (except for the case of repetition). The notations $s_1^{-1}s_2$ and $s_1s_2^{-1}$ in (M-Seq-Const-...) and (M-Seq-Bind) denote the string s (if there is any) such that $s_1s = s_2$ or $s_1 = ss_2$, respectively.

Example 3.1 Matching the string $\mathtt{a}$ against pattern $(x \text{ as } \mathtt{a}^*)\mathtt{a}^*$ yields a substitution $x \mapsto \mathtt{a}$ as below. Note how the first- and longest-match policies are working in the derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon \rhd \mathtt{a}(\mathtt{a}^*(y \text{ as } \mathtt{a}^*)) \Rightarrow \bot}\,(\text{M-Seq-Const-Fail})}{\varepsilon \rhd (\mathtt{aa}^*)(y \text{ as } \mathtt{a}^*) \Rightarrow \bot}\,(\text{M-Seq-Seq}) \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon \rhd \mathtt{aa}^* \Rightarrow \bot}\,(\text{M-Seq-Const-Fail}) \quad \dfrac{}{\varepsilon \rhd \varepsilon \Rightarrow \emptyset}\,(\text{M-Const-Succ})}{\varepsilon \rhd \mathtt{aa}^* \mid \varepsilon \Rightarrow \emptyset}\,(\text{M-Choice-Snd})}{\varepsilon \rhd \mathtt{a}^* \Rightarrow \emptyset}\,(\text{M-Rep})}{\varepsilon \rhd y \text{ as } \mathtt{a}^* \Rightarrow y \mapsto \varepsilon}\,(\text{M-Bind})}{\varepsilon \rhd \varepsilon(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Seq-Const-Succ})}{\varepsilon \rhd (\mathtt{aa}^*)(y \text{ as } \mathtt{a}^*) \mid \varepsilon(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Choice-Snd})}{\varepsilon \rhd (\mathtt{aa}^* \mid \varepsilon)(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Seq-Choice})}{\varepsilon \rhd \mathtt{a}^*(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Seq-Rep})}{\mathtt{a} \rhd \mathtt{a}(\mathtt{a}^*(y \text{ as } \mathtt{a}^*)) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Seq-Const-Succ})}{\mathtt{a} \rhd (\mathtt{aa}^*)(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Seq-Seq})}{\mathtt{a} \rhd (\mathtt{aa}^*)(y \text{ as } \mathtt{a}^*) \mid \varepsilon(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Choice-Fst})}{\mathtt{a} \rhd (\mathtt{aa}^* \mid \varepsilon)(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Seq-Choice})}{\mathtt{a} \rhd \mathtt{a}^*(y \text{ as } \mathtt{a}^*) \Rightarrow y \mapsto \varepsilon}\,(\text{M-Seq-Rep})}{\mathtt{a} \rhd (x \text{ as } \mathtt{a}^*)\mathtt{a}^* \Rightarrow x \mapsto \mathtt{a}}\,(\text{M-Seq-Bind})$$

□

By the way, there is a possibility that the above pattern matching algorithm falls into an "infinite loop" because of empty repetition patterns, that is, patterns of the form P^* where P matches ε. To put it formally, it is impossible in the above (inductive) rules to derive, say, $\varepsilon \rhd \varepsilon^* \Rightarrow \emptyset$. Fortunately, such repetition patterns can be precluded by pre-processing without changing their meanings. This technique is already studied in the literature. See [4, Section 4], for example. Thus, in the rest of this paper, we assume a priori that such patterns never appear.

A careful reader may also wonder why we did not define $s \rhd P \Rightarrow \bot$ just as "there exists no substitution θ such that $s \rhd P \Rightarrow \theta$." However, doing so makes it impossible to distinguish non-termination and failure of pattern matching. In addition, it even makes the relation *ill-defined* at all, since the rule (M-Choice-Snd) becomes non-inductive because of the premise $s \rhd P_1 \Rightarrow \bot$. (In fact, the definition of pattern matching in [6] has this problem.)

$$
\begin{aligned}
\tau \ \text{(type)} \quad &::= \tau_1 \xrightarrow{T} \tau_2 \mid T \\
T \ \text{(string type)} \ &::= s \mid (T_1 \mid T_2) \mid T^* \mid T_1 T_2 \\
&\quad \mid \ T_1 \cap T_2 \mid T_1 \cap \overline{T_2} \mid T_1^{-1} T_2 \mid T_1 T_2^{-1}
\end{aligned}
$$

$$
\begin{aligned}
[\![s]\!] &= \{s\} \\
[\![T_1 \mid T_2]\!] &= [\![T_1]\!] \cup [\![T_2]\!] \\
[\![T^*]\!] &= \{s_1 + \cdots + s_n \mid s_i \in [\![T]\!] \ (i = 1, \ldots, n), n \geq 0\} \\
[\![T_1 T_2]\!] &= \{s_1 + s_2 \mid s_1 \in [\![T_1]\!] \wedge s_2 \in [\![T_2]\!]\} \\
[\![T_1 \cap T_2]\!] &= [\![T_1]\!] \cap [\![T_2]\!] \\
[\![T_1 \cap \overline{T_2}]\!] &= [\![T_1]\!] \setminus [\![T_2]\!] \\
[\![T_1^{-1} T_2]\!] &= \{s' \mid s \in [\![T_1]\!] \wedge s + s' \in [\![T_2]\!]\} \\
[\![T_1 T_2^{-1}]\!] &= \{s \mid s + s' \in [\![T_1]\!] \wedge s' \in [\![T_2]\!]\}
\end{aligned}
$$

Fig. 4. Syntax and Semantics of Types

4 Type System

Types τ in λ^{re} are defined as in Figure 4. A function type $\tau_1 \xrightarrow{T} \tau_2$ denotes functions that take an argument of type τ_1, prints a string of type T to the output, and possibly returns a result of type τ_2. A string type T denotes a set $[\![T]\!]$ of strings. It is either regular expressions such as a constant string s, choice $T_1 \mid T_2$, repetition T^*, and sequence $T_1 T_2$, or operations on them such as intersection $T_1 \cap T_2$, difference $T_1 \cap \overline{T_2}$, left quotient $T_1^{-1} T_2$, and right quotient $T_1 T_2^{-1}$. It is known that the set of regular languages are closed with respect to these operations [5]. Nevertheless, it is useful to have them in the *syntax* of types in order to give the typing rules. Note that the syntax of types subsumes that of patterns except for variable bindings. We write $novar(P)$ for the type obtained by removing all variables in P. That is, $novar(s) = s$, $novar(P_1 \mid P_2) = novar(P_1) \mid novar(P_2)$, $novar(x \ \mathsf{as} \ P) = novar(P)$, $novar(P^*) = novar(P)^*$, and $novar(P_1 P_2) = novar(P_1) novar(P_2)$. We also extend the intersection $\cap$ and choice $\mid$ operations from strings types T to types τ in general, by defining those operations on function types $\tau_1 \xrightarrow{T} \tau_2$ as follows. (Note that they do not necessarily mean a union or intersection of *sets* – they are merely operations on types, defined as below.)

$$
\begin{aligned}
(\tau_1 \xrightarrow{T_1} \tau_1') \mid (\tau_2 \xrightarrow{T_2} \tau_2') &= (\tau_1 \cap \tau_2) \xrightarrow{T_1 \mid T_2} (\tau_1' \mid \tau_2') \\
(\tau_1 \xrightarrow{T_1} \tau_1') \cap (\tau_2 \xrightarrow{T_2} \tau_2') &= (\tau_1 \mid \tau_2) \xrightarrow{T_1 \cap T_2} (\tau_1' \cap \tau_2')
\end{aligned}
$$

Alternatively, one can think of introducing $\tau_1 \cap \tau_2$ and $\tau_1 \mid \tau_2$ in the *syntax* of types and thereby having real intersection types (that is, overloading) and

$$\frac{[\![T_1]\!] \subseteq [\![T_2]\!]}{T_1 \leq T_2}(\text{S-Str}) \qquad \frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2' \quad T_1 \leq T_2}{\tau_1 \xrightarrow{T_1} \tau_2 \leq \tau_1' \xrightarrow{T_2} \tau_2'}(\text{S-Fun})$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau, \varepsilon}(\text{T-Var}) \quad \frac{\Gamma, f : \tau_1 \xrightarrow{T} \tau_2, x : \tau_1 \vdash M : \tau_2, T' \quad T' \leq T}{\Gamma \vdash \mathsf{fix}(f, x, M) : \tau_1 \xrightarrow{T} \tau_2, \varepsilon}(\text{T-Fix})$$

$$\frac{\Gamma \vdash M_1 : \tau_2 \xrightarrow{T} \tau_1, T_1 \quad \Gamma \vdash M_2 : \tau_2', T_2 \quad \tau_2' \leq \tau_2}{\Gamma \vdash M_1 M_2 : \tau_1, T_1 T_2 T}(\text{T-App})$$

$$\frac{}{\Gamma \vdash s : s, \varepsilon}(\text{T-Const}) \qquad \frac{\Gamma \vdash M_1 : T_1, T_1' \quad \Gamma \vdash M_2 : T_2, T_2'}{\Gamma \vdash M_1 \hat{} M_2 : T_1 T_2, T_1' T_2'}(\text{T-Cat})$$

$$\frac{\left.\begin{array}{c} \Gamma \vdash M : T_1, T' \\ T_m \rightsquigarrow P_m \Rightarrow \Gamma_m \\ \Gamma, \Gamma_m \vdash M_m : \tau_m, T_m' \\ T_{m+1} = T_m \cap \overline{novar(P_m)} \end{array}\right\} m = 1, \ldots, n \qquad T_{n+1} \leq \emptyset}{\begin{array}{l} \Gamma \vdash \mathsf{match}\ M\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n : \\ \qquad \tau_1 \mid \ldots \mid \tau_n, T'(T_1' \mid \ldots \mid T_n') \end{array}}(\text{T-Match})$$

$$\frac{\Gamma \vdash M : T, T'}{\Gamma \vdash \mathsf{print}\ M : \varepsilon, T'T}(\text{T-Print})$$

Fig. 5. Typing Rules for Terms

union types. A similar idea is already studied in [3]. It may be possible to incorporate their work into ours, but we refrain from doing so for now in favor of simplicity.

The typing rules are given in Figure 5 and 6. For the sake of expressiveness and precision, we define a subtyping relation $T_1 \leq T_2$ between string types T_1 and T_2 by inclusion between their denotations as in rule (S-Str). Thanks to their regularity, this inclusion is decidable (although its efficient implementation is beyond the scope of the present paper). This subtyping relation between string types is extended to $\tau_1 \leq \tau_2$ for types in general by a standard

$$\frac{(\llbracket T \rrbracket, P) \in \Pi \quad \forall x \in dom(\Gamma) = var(P).\, \Gamma(x) = \emptyset}{\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma}\text{(P-Mem)}$$

$$\frac{}{\Pi \vdash T \rightsquigarrow s \Rightarrow \emptyset}\text{(P-Const)}$$

$$\frac{\Pi \vdash T \rightsquigarrow P_1 \Rightarrow \Gamma_1 \quad \Pi \vdash T \cap \overline{novar(P_1)} \rightsquigarrow P_2 \Rightarrow \Gamma_2 \quad \forall x \in dom(\Gamma) = dom(\Gamma_1) = dom(\Gamma_2).\, \Gamma(x) = \Gamma_1(x) \mid \Gamma_2(x)}{\Pi \vdash T \rightsquigarrow P_1 \mid P_2 \Rightarrow \Gamma}\text{(P-Choice)}$$

$$\frac{\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow x \text{ as } P \Rightarrow \Gamma \uplus \{x : \Gamma(P) \cap T\}}\text{(P-Bind)}$$

$$\frac{\Pi \uplus \{(\llbracket T \rrbracket, P^*)\} \vdash T \rightsquigarrow P P^* \mid \varepsilon \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow P^* \Rightarrow \Gamma}\text{(P-Rep)}$$

$$\frac{\Pi \vdash s^{-1}T \rightsquigarrow P \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow sP \Rightarrow \Gamma}\text{(P-Seq-Const)}$$

$$\frac{\Pi \vdash T \rightsquigarrow P_1 P_3 \mid P_2 P_3 \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow (P_1 \mid P_2) P_3 \Rightarrow \Gamma}\text{(P-Seq-Choice)}$$

$$\frac{\forall (\llbracket T \rrbracket, P) \in \Pi.\, y \notin var(P) \quad y \notin var(P_1 P_2) \quad \Pi \vdash T_1 \rightsquigarrow P_1(y \text{ as } P_2) \Rightarrow \Gamma \uplus \{y : T_2\}}{\Pi \vdash T_1 \rightsquigarrow (x \text{ as } P_1) P_2 \Rightarrow \Gamma \uplus \{x : \Gamma(P_1) \cap T_1 T_2^{-1}\}}\text{(P-Seq-Bind)}$$

$$\frac{\Pi \uplus (\llbracket T \rrbracket, {P_1}^* P_2) \vdash T \rightsquigarrow (P_1 {P_1}^* \mid \varepsilon) P_2 \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow {P_1}^* P_2 \Rightarrow \Gamma}\text{(P-Seq-Rep)}$$

$$\frac{\Pi \vdash T \rightsquigarrow P_1 (P_2 P_3) \Rightarrow \Gamma}{\Pi \vdash T \rightsquigarrow (P_1 P_2) P_3 \Rightarrow \Gamma}\text{(P-Seq-Seq)}$$

Fig. 6. Typing Rules for Patterns

subtyping rule (S-Fun) for functions.

A typing judgment $\Gamma \vdash M : \tau, T$ means that "under type environment Γ, term M has type τ and effect T." The typing rules are standard except for conditions about effects and pattern matching. As for effects, consider rule (T-Print) for example. The term $\mathsf{print}\ M$ first evaluates M to a string s, prints s to the output, and then returns the empty string ε. Thus, if M has a string type T and effect T', then $\mathsf{print}\ M$ has the string type ε and effect $T'T$. Conditions about effects in other rules are also based on similar reasoning. (The condition $T' \leq T$ in rule (T-Fix) may seem unnecessary, but it is actually necessary for the typing of almost all recursive functions—either terminating or diverging—with effects. Consider $\vdash \mathsf{fix}(f, x, \mathsf{print}\ \mathtt{a}; f\ x) : \varepsilon \xrightarrow{\mathtt{a}^*} \emptyset$, for instance.[2])

As for pattern matching, see rule (T-Match). First, the input string M must have a string type T_1. Then, this input string of type T_1 is matched against the first pattern P_1. If this pattern matching succeeds, it will yield some variable bindings. In order to calculate the type of those bound variables, we introduce an auxiliary relation $T \rightsquigarrow P \Rightarrow \Gamma$, meaning that "if a string of type T matches pattern P, then each variable x thus bound will have type $\Gamma(x)$." In this way, the type environment Γ_1 is calculated from T_1 and P_1 as $T_1 \rightsquigarrow P_1 \Rightarrow \Gamma_1$, according to which the first body M_1 is typed as $\Gamma, \Gamma_1 \vdash M_1 : \tau_1, T_1'$. (By the way, if there is no string of type T_1 that matches P_1 at all, that is, if the intersection $T_1 \cap \mathit{novar}(P_1)$ of T_1 with the type of strings that match P_1 is empty, then this pattern matching never succeeds and is therefore redundant. Although this does not affect type soundness, we may choose to *warn* of such redundancy in the implementation for the sake of programmers' convenience. We can not take it as a type *error*, however, because doing so would break the type preservation property below.) Furthermore, because of our first-match policy, an input string that matched the first pattern P_1 will not be an input string for the second pattern P_2. Thus, the type T_2 of input strings for P_2 is calculated as the difference $T_1 \cap \overline{\mathit{novar}(P_1)}$ of T_1 and (the type of strings that match) P_1. This line of typing is repeated for each pattern P_m and each body M_m for $m = 1, \ldots, n$. Last, in order for the pattern matching to be exhaustive, the type T_{n+1} of input strings that matched no patterns must be empty. Note that an empty type (that is, a type whose denotation is an empty set) is different from type ε (that is, a type whose denotation is the singleton set of the empty string). We write $\emptyset$ for an empty type. It never matters *which* empty type we choose, but $\varepsilon \cap \overline{\varepsilon}$ is one.

The typing relation $T \rightsquigarrow P \Rightarrow \Gamma$ for patterns is defined similarly to their operational semantics $s \rhd P \Rightarrow \theta$. Again, the rules can be read in a bottom-up manner as an algorithm that takes a pattern P with the type T of input strings and calculates the resulting type environment Γ. Since a string type denotes a (regular) set of strings, these typing rules for patterns are indeed similar

[2] In this example, by the way, $\mathtt{a}^*$ does not match the actual effect $\mathtt{a}^\infty$ (the infinite sequence of a). This does not break the effect soundness property stated below, since it is defined in terms of each finite *prefix* of an infinite reduction sequence.

to the operational semantics of patterns (cf. Figure 3). However, besides the obvious distinction that those rules deal with string types instead of individual strings, there are the following two major differences.

One is that a "memoization" technique is necessary for guaranteeing termination of this type inference algorithm. Specifically, we generalize the relation to $\Pi \vdash T \rightsquigarrow P \Rightarrow \Gamma$ where Π is a set of pairs $(\llbracket T \rrbracket, P)$ of (the denotation of) a string type T and a pattern P. We abbreviate this relation to $T \rightsquigarrow P \Rightarrow \Gamma$ when $\Pi = \emptyset$. Intuitively, Π remembers repetition patterns that have "already appeared" during the typing, along with the corresponding input string type. This is represented by rules (P-Rep) and (P-Seq-Rep). Then, if the derivation comes across such a pair again, it can give a type environment with the type of all variables empty, as represented by rule (P-Mem). Without these, typing of *any* pattern that contains a repetition would diverge.[3]

The other is that the typing relation itself does *not* mean success of pattern matching; it just tells what strings are bound to each variable *when* the pattern matching succeeds. This is most apparent in rule (P-Const), which implies $\emptyset \vdash \mathtt{a}^* \rightsquigarrow \mathtt{b} \Rightarrow \emptyset$, for example. It means "matching a string of type $\mathtt{a}^*$ against the (constant string) pattern $\mathtt{b}$ yields no variable binding," but this pattern matching never succeeds, of course. This may seem unpleasant but is inevitable, in particular for the typing (P-Choice) of choice patterns $P_1 \mid P_2$, where it is too restrictive to always require that the input string really matches the first pattern P_1. Consider, for instance, the use of an "any" pattern $.^*$ for default cases like $\mathsf{match}\ M\ \mathsf{with}\ P \Rightarrow M_1 \mid x\ \mathsf{as}\ .^* \Rightarrow M_2$, where . is an abbreviation of $\mathtt{a} \mid \mathtt{b} \mid \ldots$ with $\{\mathtt{a}, \mathtt{b}, \ldots\}$ being the set of all characters. Even though the input string for $.^*$ can not be an arbitrary string, it is quite natural to use the "any" pattern here, so we should not take it as a type error. Nevertheless, we *do* want the *type* of x to reflect the fact that it is never bound to strings that match P_1. This is achieved in rule (P-Bind) by taking the intersection of the input string type T and the range $\Gamma(P)$ of the pattern P under the type environment Γ, where $\Gamma(P)$ is a type defined as $\Gamma(s) = s$, $\Gamma(P_1 \mid P_2) = \Gamma(P_1) \mid \Gamma(P_2)$, $\Gamma(x\ \mathsf{as}\ P) = \Gamma(x) \cap \Gamma(P)$, $\Gamma(P^*) = \Gamma(P)^*$, and $\Gamma(P_1 P_2) = \Gamma(P_1)\Gamma(P_2)$. The same idea is also adopted in (P-Seq-Bind). (In previous work [6], by contrast, it was not possible to give a precise type to as-patterns in such non-tail positions. Since the theory of regular trees is quite similar to that of regular strings, we conjecture that our approach is also applicable to that work.) Note that these rules extend type environments only conservatively by using $\uplus$ and thus require *linearity* of variables in a pattern.

For the sake of convenience, we may implicitly substitute a type T with another type T' when $\llbracket T \rrbracket = \llbracket T' \rrbracket$. This never affects the result of typing.

[3] Formally, even when (P-Mem) is applicable, we do *not* actually forbid applying the other rules, because the typing rules are inductively defined and each type derivation is anyway guaranteed to be finite. This choice does not affect the result of typing. Of course, from an implementation viewpoint, it would be reasonable to apply (P-Mem) whenever possible.

Example 4.1 Matching strings of type $\mathtt{aa}^*$ against pattern $(x \text{ as } \mathtt{a}^*)(y \text{ as } \mathtt{a}^*)$ binds x to a string of type $\mathtt{aa}^*$ and y to ε. That is, the judgment $\mathtt{aa}^* \rightsquigarrow (x \text{ as } \mathtt{a}^*)(y \text{ as } \mathtt{a}^*) \Rightarrow x : \mathtt{aa}^*, y : \varepsilon$ can be derived as below. Note how the typing reflects our first-match and longest-match policies.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta}{\Pi_1 \vdash \mathtt{aa}^* \rightsquigarrow (\mathtt{aa}^*)(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Seq-Seq)} \quad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Pi_1 \vdash \emptyset \rightsquigarrow \mathtt{a}^* \Rightarrow \emptyset}\ \text{(P-Rep)}}{\Pi_1 \vdash \emptyset \rightsquigarrow y \text{ as } \mathtt{a}^* \Rightarrow y : \emptyset}\ \text{(P-Bind)}}{\Pi_1 \vdash \emptyset \rightsquigarrow z \text{ as } (y \text{ as } \mathtt{a}^*) \Rightarrow y : \emptyset, z : \emptyset}\ \text{(P-Bind)}}{\Pi_1 \vdash \emptyset \rightsquigarrow \varepsilon(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \emptyset, z : \emptyset}\ \text{(P-Seq-Const)}}{\Pi_1 \vdash \mathtt{aa}^* \rightsquigarrow (\mathtt{aa}^*)(z \text{ as } (y \text{ as } \mathtt{a}^*)) \mid \varepsilon(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Choice)}}{\Pi_1 \vdash \mathtt{aa}^* \rightsquigarrow (\mathtt{aa}^* \mid \varepsilon)(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Seq-Choice)}}{\emptyset \vdash \mathtt{aa}^* \rightsquigarrow \mathtt{a}^*(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Seq-Rep)}}{\emptyset \vdash \mathtt{aa}^* \rightsquigarrow (x \text{ as } \mathtt{a}^*)(y \text{ as } \mathtt{a}^*) \Rightarrow x : \mathtt{aa}^*, y : \varepsilon}\ \text{(P-Seq-Bind)}$$

where Δ is the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Pi_2 \vdash \varepsilon \rightsquigarrow \mathtt{a}^* \Rightarrow \emptyset}\ \text{(P-Rep)}}{\Pi_2 \vdash \varepsilon \rightsquigarrow y \text{ as } \mathtt{a}^* \Rightarrow y : \varepsilon}\ \text{(P-Bind)}}{\Pi_2 \vdash \varepsilon \rightsquigarrow z \text{ as } (y \text{ as } \mathtt{a}^*) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Bind)}}{\Pi_2 \vdash \varepsilon \rightsquigarrow \varepsilon(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Seq-Const)}}{\Pi_2 \vdash \mathtt{a}^* \rightsquigarrow (\mathtt{aa}^*)(z \text{ as } (y \text{ as } \mathtt{a}^*)) \mid \varepsilon(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Choice)}}{\Pi_2 \vdash \mathtt{a}^* \rightsquigarrow (\mathtt{aa}^* \mid \varepsilon)(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Seq-Choice)}}{\Pi_1 \vdash \mathtt{a}^* \rightsquigarrow \mathtt{a}^*(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Seq-Rep)}}{\Pi_1 \vdash \mathtt{aa}^* \rightsquigarrow \mathtt{a}(\mathtt{a}^*(z \text{ as } (y \text{ as } \mathtt{a}^*))) \Rightarrow y : \varepsilon, z : \varepsilon}\ \text{(P-Seq-Const)}}{\Pi_2 \vdash \mathtt{a}^* \rightsquigarrow \mathtt{a}^*(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \emptyset, z : \emptyset}\ \text{(P-Mem)}}{\Pi_2 \vdash \mathtt{a}^* \rightsquigarrow \mathtt{a}(\mathtt{a}^*(z \text{ as } (y \text{ as } \mathtt{a}^*))) \Rightarrow y : \emptyset, z : \emptyset}\ \text{(P-Seq-Const)}}{\Pi_2 \vdash \mathtt{a}^* \rightsquigarrow (\mathtt{aa}^*)(z \text{ as } (y \text{ as } \mathtt{a}^*)) \Rightarrow y : \emptyset, z : \emptyset}\ \text{(P-Seq-Seq)}$$

with $\Pi_1 = \{([\![\mathtt{aa}^*]\!], \mathtt{a}^*(z \text{ as } (y \text{ as } \mathtt{a}^*)))\}$ and $\Pi_2 = \Pi_1 \cup \{([\![\mathtt{a}^*]\!], \mathtt{a}^*(z \text{ as } (y \text{ as } \mathtt{a}^*)))\}$. □

Soundness of the above type system is proved as follows by standard arguments with a few properties about pattern matching.

Theorem 4.2 (Type and Effect Soundness) *If $\vdash M : \tau, T$ and $M \xrightarrow{s_1} M_1 \xrightarrow{s_2} M_2 \xrightarrow{s_3} \ldots \xrightarrow{s_n} M_n$, then $s_1 s_2 s_3 \ldots s_n$ is a prefix of some $s \in [\![T]\!]$. Furthermore, if M_n is irreducible (that is, $M_n \xrightarrow{s} M'$ for no s and M'), then M_n is a value of a subtype of τ (that is, $M_n = v$ with $\vdash v : \tau', \varepsilon$ and $\tau' \leq \tau$ for some v and τ') and $s_1 s_2 s_3 \ldots s_n \in [\![T]\!]$.*

Proof. Immediate from the progress and type preservation lemmas below. □

Lemma 4.3 (Progress) *If $\vdash M : \tau, T$ and M is not a value, then M is reducible (that is, $M \xrightarrow{s} M'$ for some s and M').*

Proof. Induction on the derivation of $\vdash M : \tau, T$ with the following two lemmas for the case of (T-Match). □

Lemma 4.4 (Determinacy and Termination of Pattern Matching) *If $s \rhd P \Rightarrow \theta_1$ and $s \rhd P \Rightarrow \theta_2$, then $\theta_1 = \theta_2$. In addition, if $s \in [\![novar(P)]\!]$, then $s \rhd P$. Conversely, if $s \notin [\![novar(P)]\!]$, then $s \rhd P \Rightarrow \bot$.*

Lemma 4.5 (Exhaustiveness of Pattern Matching) *If* $\Gamma \vdash \mathsf{match}\ s\ \mathsf{with}\ P_1 \Rightarrow M_1 \mid \ldots \mid P_n \Rightarrow M_n : \tau, T$, *then* $s \rhd P_i$ *for some* $1 \leq i \leq n$.

Lemma 4.6 (Type Preservation) *If* $\Gamma \vdash M : \tau, T$ *and* $M \xrightarrow{s} M'$, *then* $\Gamma \vdash M' : \tau', T'$ *for some* τ' *and* T' *with* $\tau' \leq \tau$ *and* $sT' \leq T$.

Proof. Induction on the derivation of $\Gamma \vdash M : \tau, T$ with the following two lemmas for the cases of (T-App) and (T-Match), respectively. □

Lemma 4.7 (Substitution) *If* $\Gamma, x : \tau' \vdash M : \tau, T$ *and* $\Gamma \vdash v : \tau', \varepsilon$, *then* $\Gamma \vdash [v/x]M : \tau, T$.

Lemma 4.8 (Soundness and Completeness of Type Inference for Pattern Matching) *If* $T \rightsquigarrow P \Rightarrow \Gamma$, *then* $[\![\Gamma(x)]\!] = \{\theta(x) \mid s \in [\![T]\!] \wedge s \rhd P \Rightarrow \theta \wedge \theta \neq \bot\}$ *for any* $x \in var(P) = dom(\Gamma)$.

The last lemma is much trickier than it may seem (the $\supseteq$ direction, in particular). For instance, if we allowed (possibly) empty repetition patterns such as ε^*, this property would not hold with a counterexample being $\mathtt{a} \rightsquigarrow \varepsilon^*(y\ \mathsf{as}\ \mathtt{a}) \Rightarrow y : \emptyset$, which could break type soundness (though it actually does not, fortunately, because we have precluded such patterns in Section 3 without loss of generality). Accordingly, the proof is far from trivial, involving auxiliary inductive relations to state the invariant which Π keeps. Details of the proof are given in the aforementioned manuscript.

5 Programming Examples

This section gives a few tiny examples of programming (and typing) in λ^{re}.

Booleans

By using the strings `t` and `f` (of length 1) as true and false, the boolean algebra can be encoded into λ^{re}, where a conditional expression $\mathsf{if}\ M_0\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$ is encoded by a pattern matching $\mathsf{match}\ M_0\ \mathsf{with}\ \mathtt{t} \Rightarrow M_1 \mid \mathtt{f} \Rightarrow M_2$. Thus, type bool can be defined to be `t` | `f`. In combination with recursive functions, this makes λ^{re} Turing-complete.

Lists

Let T be any string type whose denotation contains no string that includes the character , (comma). Then, lists of strings of this type can be represented by strings of type $(T,)^*$ as follows: the empty list is represented by ε; the cons of an element s and (the representation of) a list ℓ is represented by $s\,\hat{}\,,\hat{}\,\ell$; destruction of a list ℓ can be implemented by pattern matching like $\mathsf{match}\ \ell\ \mathsf{with}\ \varepsilon \Rightarrow M_1 \mid (x\ \mathsf{as}\ T),(y\ \mathsf{as}\ .^*) \Rightarrow M_2$, where x is the head ("car") and y is the tail ("cdr") of ℓ. Even though the pattern for y is written as $.^*$, its type is automatically and precisely inferred as $(T,)^*$ provided that ℓ is typed $(T,)^*$. Note that this is an example of the use of as-patterns in non-tail

positions, which is one of the technical advantages of the present type system to previous work [6].

Filtering of List

Simple filtering like "grep" over (the above encoding of) lists can be implemented in the following obvious way, as far as the filtering pattern T' is static (that is, given at compile-time).

$$\mathsf{fix}(f, x, \mathsf{match}\ x\ \mathsf{with}$$
$$\varepsilon \Rightarrow \varepsilon \mid (y\ \mathsf{as}\ T'),(z\ \mathsf{as}\ .^*) \Rightarrow ,\hat{}\, y \hat{}\, f(z) \mid T,(z\ \mathsf{as}\ .^*) \Rightarrow ,\hat{}\, f(z))$$

As expected, this filtering function can be typed as $(T,)^* \xrightarrow{\varepsilon} ((T \cap T'),)^*$.

Parsing and Pretty-Printing

Let us assume (without formal definitions, just for the sake of brevity) that we have natural numbers and standard operations on them as primitives. Then, the conversion between natural numbers and their string representations can be programmed as terms

$$\mathsf{let}\ \mathit{dig2chr} =$$
$$\lambda d.\,\mathsf{if}\ d = 9\ \mathsf{then}\ \mathtt{9}\ \mathsf{else}\ \mathsf{if}\ d = 8\ \mathsf{then}\ \mathtt{8}\ \mathsf{else}\ \ldots\ \mathsf{if}\ d = 1\ \mathsf{then}\ \mathtt{1}\ \mathsf{else}\ \mathtt{0}\ \mathsf{in}$$
$$\mathsf{fix}(\mathit{nat2str}, n,$$
$$\mathsf{if}\ n < 10\ \mathsf{then}\ \mathit{dig2chr}(n)\ \mathsf{else}\ \mathit{nat2str}(n\ \mathsf{div}\ 10) \hat{}\, \mathit{dig2chr}(n\ \mathsf{mod}\ 10))$$

and

$$\mathsf{let}\ \mathit{chr2dig} = \lambda c.\,\mathsf{match}\ c\ \mathsf{with}\ \mathtt{9} \Rightarrow 9 \mid \ldots \mid \mathtt{0} \Rightarrow 0\ \mathsf{in}$$
$$\mathsf{fix}(\mathit{str2int}, s, \mathsf{match}\ s\ \mathsf{with}$$
$$(d\ \mathsf{as}\ .) \Rightarrow \mathit{chr2dig}(d)$$
$$\mid (d\ \mathsf{as}\ .)(t\ \mathsf{as}\ .^*) \Rightarrow \mathit{chr2dig}(d) \times 10 + \mathit{str2int}(t))$$

of types $\mathsf{nat} \xrightarrow{\varepsilon} \iota^*$ and $\iota^* \xrightarrow{\varepsilon} \mathsf{nat}$, respectively, where $\iota = \mathtt{0} \mid \ldots \mid \mathtt{9}$. Note, however, that our type system is not strong enough to type the codomain of *nat2str* as $\mathtt{0} \mid (\iota \cap \overline{\mathtt{0}})\iota^*$ excluding actually impossible results such as `00`. Doing so would require dependent types over nat.

Verification of User Input

Suppose that a string *instr* is typed $.^*$ and stands for some user input, and that the programmer expects it to be the representation of a date in the format of DD/MM/YYYY. Then, a partial verification of this input string

can be implemented by the following function with $\iota = \mathtt{0} \mid \ldots \mid \mathtt{9}$.

$$\mathsf{let}\ \mathit{verify_input} = \lambda s.\, \mathsf{match}\ s\ \mathsf{with}\ \iota/\iota\iota/\iota\iota\iota \Rightarrow s \mid .^* \Rightarrow \mathsf{print}\ \mathtt{ERR}\ \mathsf{in}$$
$$\mathit{verify_input}\ \mathit{instr}$$

The function *verify_input* is typed $.^* \stackrel{\varepsilon|\mathtt{ERR}}{\to} \iota\iota/\iota\iota/\iota\iota\iota\iota \mid \varepsilon$. This tells us that *verify_input* returns only a string of the correct format or an empty string, and that it may also output the message `ERR` as an effect. In addition, the rule (T-Match) prevents the programmer from forgetting error handling (the "default" case).

Filtering of Infinite Stream

Let us assume that a function *read_line* of type $\varepsilon \stackrel{\varepsilon}{\to} .^*$ inputs one line from an infinite stream. (Its effect is ε because our present effect system concerns only the *output* of a program.) Then, a filtering program that escapes HTML meta-characters (< and >) can be implemented as follows:

$$\begin{array}{l}
\mathsf{let}\ \mathit{escline} = \mathsf{fix}(\mathit{el}, s, \mathsf{match}\ s\ \mathsf{with} \\
\quad (s_1\ \mathsf{as}\ .^*)\mathtt{<}(s_2\ \mathsf{as}\ .^*) \Rightarrow (\mathit{el}\ s_1)\ \hat{}\ \&\mathtt{lt;}\ \hat{}\ (\mathit{el}\ s_2) \\
\quad \mid (s_1\ \mathsf{as}\ .^*)\mathtt{>}(s_2\ \mathsf{as}\ .^*) \Rightarrow (\mathit{el}\ s_1)\ \hat{}\ \&\mathtt{gt;}\ \hat{}\ (\mathit{el}\ s_2) \\
\quad \mid .^* \Rightarrow s)\ \mathsf{in} \\
\mathsf{fix}(\mathit{main}, x, \mathsf{print}\ (\mathit{escline}\ (\mathit{read_line}\ \varepsilon)); \mathit{main}\ x)
\end{array}$$

The functions *el* and *main* can be typed as $.^* \stackrel{\varepsilon}{\to} (. \cap \overline{< \mid >})^*$ and $\varepsilon \stackrel{(.\cap\overline{<|>})^*}{\to} \emptyset$ respectively. This is an example of typing of diverging programs.

6 Conclusions

We have presented λ^{re}, a tiny text-processing language with a regular expression type (and effect) system for strings.

Besides XDuce, Igarashi and Kobayashi's *resource usage analysis* [9] (and its predecessors) is also relevant to our work. It is a generic framework to statically analyze (and verify) the pattern of access in a program to various kinds of resources such as files and memory regions. Our regular expression effect system can be seen as a simple instance of their framework. However, type checking in their system is undecidable in general because their access pattern language has recursion, with which equivalence and inclusion become undecidable. On the other hand, if explicit type annotations are given for bound variables in fix, type checking in our system is decidable because our string types are regular. There is also plenty of work in the context of type systems for concurrent programs which makes use of regular expression-like type languages that (more or less) resemble our effect language. Nierstrasz

proposed *regular types* to infer and check the behavior of objects [16]. A similar idea is adopted by Nielson and Nielson in their study on concurrent ML [14,15].

Other relevant work is refinement types [2], which allow a kind of subtyping among data types in ML. While the relationship between our regular expression types and refinement types is not clear yet, we conjecture that neither subsumes the other since refinement types require declaration of data types and insertion of tags by the programmer (as in ordinary ML). More accurate comparison of these type systems is left for future work.

In CDuce [3] and a recent version of XDuce [7], the problem of calculating the precise types of non-tail pattern-variables has been solved independently. However, it seems that their settings are different from ours in several points. In CDuce, types and patterns are represented as trees, not strings. Although it is straightforward to encode strings as trees, strings require *more* equivalence than their encodings in trees, which differentiates the problem. For example, $((\mathtt{a},\mathtt{b}),\mathtt{c})$ and $(\mathtt{a},(\mathtt{b},\mathtt{c}))$ are different as trees but $(\mathtt{ab})\mathtt{c}$ and $\mathtt{a}(\mathtt{bc})$ are the same strings. In XDuce, a precise type inference algorithm is recently implemented that directly calculates the intersection of a pattern and an input type automaton. However, the current version of XDuce employs non-deterministic pattern matching rather than first-match, which would require modifications to the algorithm [Haruo Hosoya, personal communication, January 2003].

This work is far from complete yet. In particular, an efficient algorithm for type checking and—more importantly—type reconstruction in λ^{re} is essential in practice but unavailable at present. A major technical problem here is that standard type reconstruction by constraint solving introduces recursion and thereby requires more expressive solutions than regular expressions (context-free grammars, at least). Recall, for example, the recursive function $f(n) = \mathsf{if}\ n \leq 0\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \mathtt{a} \mathbin{\hat{}} f(n-1) \mathbin{\hat{}} \mathtt{b}$ in Section 1. From its type reconstruction, we obtain a constraint like $\varepsilon \mid \mathtt{a}\alpha\mathtt{b} \leq \alpha$, which has no least solution within regular expressions because all of $\mathtt{a}^*\mathtt{b}^* \geq (\varepsilon \mid \mathtt{aa}^*\mathtt{b}^*\mathtt{b}) \geq (\varepsilon \mid \mathtt{ab} \mid \mathtt{aaa}^*\mathtt{b}^*\mathtt{bb}) \geq (\varepsilon \mid \mathtt{ab} \mid \mathtt{aabb} \mid \mathtt{aaaa}^*\mathtt{b}^*\mathtt{bbb}) \geq \ldots$ are valid solutions. We are currently working on a type reconstruction algorithm which gives "modest" approximate solutions, incorporating similar work [12,13] from the domain of natural language processing.

Another direction of future work is to incorporate *input* effects in a "dual" manner to the output effects. A typing judgment will then be like $\Gamma \vdash M : \tau, T_i, T_o$, meaning that "under type environment Γ, the term M evaluates to a value of type τ and outputs a string of type T_o, *provided that* the input is a string of type T_i." For example, letting $\mathsf{input}()$ be the primitive for input, the input effect T_i of the program $\mathsf{match}\ \mathsf{input}()\ \mathsf{with}\ \mathtt{y} \Rightarrow \mathsf{print}\ \mathtt{YES} \mid \mathtt{n} \Rightarrow \mathsf{print}\ \mathtt{NO}$ would be $\mathtt{y} \mid \mathtt{n}$ with its output effect T_o being $\mathtt{YES} \mid \mathtt{NO}$. That is, this program does not "go wrong" *as far as* its input is either `y` or `n`. In this way, the input effect of a program represents the form of input with respect to which the program runs without an error such as match failure.

Acknowledgement

We would like to thank Haruo Hosoya, Atsushi Igarashi, Naoki Kobayashi, the members of Yonezawa's group in the University of Tokyo, and the anonymous reviewers for their suggestions of various improvements.

This work was partially supported by the Ministry of Education, Culture, Sports, Science and Technology under Grants-in-Aid for Scientific Research, Priority Areas Research (B) (2), 12133203.

References

[1] *CERT Advisory CA-2000-02*, `http://www.cert.org/advisories/CA-2000-02.html` (2000).

[2] Freeman, T. and F. Pfenning, *Refinement types for ML*, in: *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation* (1991), pp. 268–277.

[3] Frisch, A., G. Castagna and V. Benzaken, *Semantic subtyping*, in: *Proceedings of the LICS 2002 17th Annual IEEE Symposium on Logic in Computer Science* (2002), pp. 137–146.

[4] Harper, R., *Proof-directed debugging*, Journal of Functional Programming **9** (1999), pp. 463–469.

[5] Hopcroft, J. E., R. Motwani and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation," Addison-Wesley Publishing, 2000, 2nd edition.

[6] Hosoya, H. and B. C. Pierce, *Regular expression pattern matching for XML*, Journal of Functional Programming (2003), to appear. Extended abstract in *Proceedings of the POPL 2001 Symposium on Principles of Programming Languages*, pages 67–80, 2001.

[7] Hosoya, H. and B. C. Pierce, *XDuce: A typed XML processing language*, ACM Transactions on Internet Technology (2003), to appear.

[8] Hosoya, H., J. Vouillon and B. C. Pierce, *Regular expression types for XML*, in: *International Conference on Functional Programming*, 2000, pp. 11–22.

[9] Igarashi, A. and N. Kobayashi, *Resource usage analysis*, in: *Proceedings of the POPL 2001 Symposium on Principles of Programming Languages*, 2002, pp. 331–342.

[10] Lutz, M., "Programming Python," O'Reilly & Associates, 2001, 2nd edition.

[11] Matsumoto, Y. M. and K. Ishituka, "The Ruby Programming Language," Addison-Wesley Publishing, 2002.

[12] Mohri, M. and M.-J. Nederhof, *Regular approximation of context-free grammars through transformation*, in: J.-C. Junqua and G. V. Noord, editors, *Robustness in Language and Speech Technology*, Kluwer Academic Publishers, 2001 .

[13] Nederhof, M.-J., *Regular approximation of CFLs: A grammatical view*, in: H. C. Bunt and A. Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, Kluwer Academic Publishers, 2000 .

[14] Nielson, F. and H. R. Nielson, *From CML to process algebras*, in: *Proceedings of the CONCUR '93 International Conference on Concurrency Theory*, Lecture Notes in Computer Science **715** (1993), pp. 493–508.

[15] Nielson, H. R. and F. Nielson, *Higher-order concurrent programs with finite communications topology*, in: *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, New York, NY, 1994, pp. 84–97.

[16] Nierstrasz, O., *Regular types for active objects*, in: O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, Prentice-Hall, 1995 pp. 99–121.

[17] Thomas, D., A. Hunt and D. Thomas, "Programming Ruby: A Pragmatic Programmer's Guide," Addison-Wesley Publishing, 2000.

[18] Wall, L. et al., "Programming Perl," O'Reilly & Associates, 2000, 3rd edition.