Let-Binding with Regular Expressions in Lambda Calculus

Takuya Ohata, Shin-ya Nishizaki*
Department of Computer Science, Tokyo Institute of Technology, 2-12-1-W8-69, Ookayama, Meguro, Tokyo 152-8552, Japan.

* Corresponding author. Tel.: +81-3-5734-3506; email: nisizaki@cs.titech.ac.jp
Manuscript submitted October 1, 2015; accepted December 10, 2015.
doi: 10.17706/jsw.11.2.220-229

Abstract: We often give proper names to variables in programs based on their types, usages, and means, and the regularity and there are several kinds of conventions for variable-naming in programming languages.

For example, we use variables *i*, *j*, *k* or *i*1, *i*2, *i*3 for thevariables of integer type. In this paper, we propose let-binding mechanism by which you can declare multiple variables simultaneously using regular expressions. We formalize this variable binding mechanism in the framework of the lambda calculus: we propose a lambda calculus with the regular expression let-bindings and a simple type system to the calculus in the style of Curry. We then study the calculus and the type system from the theoretical viewpoint.

Key words: Programming language design, functional programming language, regular expression, variable declaration.

1. Introduction

In this section, we would like to introduce several backgrounds of our research.

1.1. Regular Expression

The *regular expressions* [1] consist of constant symbols and operator symbols and denotes sets of strings. Suppose that a finite set Σ of alphabets is given. The Constant symbols of the regular expression are ∅, ϵ , and a ($\in\Sigma$). The operator symbols are $\cdot$, | , and $*$.

Regular expressions are defined inductively by the following grammar.

- Constant symbol ∅is a regular expression, which denotes the empty set of strings ∅;
- Constant symbol ϵ is a regular expression, denotes a singleton set of the empty string ϵ;
- Constant symbol a is a regular expression, denotes a singleton set of string consisting of only one character $a\in\Sigma$.
- If R and S are regular expressions, then $R \cdot S$ is a regular expression, called *concatenation,* which denotes a set of strings

$$R \cdot S = \{\alpha\beta \mid \alpha \in |R| \; and \; \beta \in |S|\}.$$

- If R and S are regular expressions, the$R \mid S$ is a regular expression, called *alternation,* which denotes a union of two sets of strings

$$R \mid S = |R| \cup |S|.$$

- If R is a regular expression, the $R*$ is a regular expression, called *Kleene star,* which denotes the smallest superset of $|R|$ that contains ϵ and is closed under concatenation.

For thesake of simplicity, we write concatenation $R \cdot S$ as RS in the later part of this paper. For example,

$$(a \mid b)\ c$$

denotes a set {*ac, bc* } and $((a|b)c)*$ a set

$$\{\epsilon, ac, acac, \ldots, bc, bcbc, \ldots, ac, acbc, acbcac, \ldots\}$$

We also use a Unix-style notation such as [0-9], which means a set consisting of the digits 0,1,2, …, 9. A set of the alpha-numeric characters is represented by [a-zA-A-Z0-9]. The notation[^] represents a set of single characters that are not contained within the brackets. For example, [^0-9] denotes a set of the characters except digits.

Pattern matching with regular expressions has been incorporated into text editors since 1960's. Many programming languages have provided regular expression facilities. In scripting languages such as Perl, JavaScript and Ruby, you can write regular expressions using the language's syntax and in the other languages, using the standard library. For example, you can match a regular expression (\d+):(\d+)(\d+) with a string 11:45:14 as

result = "11:45:14".match(/(*d*+):(*d*+):(*d*+)/);

where *d* means a set of digits 0,…,9. If the pattern match succeeds, then you can extract each matched substring within parentheses referring. For example, you can get the second matched substring thorough an expression RegExp.$2, whose value is "45".

1.2. Variable Declaration and Variable Binding

A *variable declaration* specifies the variable, which makes the existence and data type of the variable know to the compiler. For example, a fragment of C language's source program

```
int i;
int sum=0;
for(i=0; i<10; i++){
   sum += i;
}
```

i and *j* are declared as variables of type int and sum is initialized as 10 simultaneously with its declaration.

In programming languages, variable binding is the association of data with variables. In functional languages, such as Haskell [2], Standard ML [3], and Scheme [4], typical binding of variables appears in let-expressions. For example, in a Scheme's expression

```
(let ((i 1)
(j 2))
(+ i j))
```

variables *i* and *j* are bound to 1 and 2, respectively. In many procedural programming languages, a type of a variable is determined by its variable declaration. On the other hand, in typed functional languages such as Standard ML [3] and Haskell [2], a type of a variable is determined by type inference provided by a complier. The compiles knows theexistence of the variables used in a program by tracking variable bindings and therefore variables bindings play a role of variable declaration.

1.3. Naming Convention

In programming, a naming convention is a set of rules for choosing the character sequence to be used for identifiers which denote variables, types and functions in program source code. Naming conventions are explicitly given as guideline in programming language communities and development teams. Naming conventions are also shared with unwritten rules in mathematics. For example, i,j,k are used for indices of matrices' components in linear algebra, such as

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \left(a_{ij}\right)_{i,j\in\{1,\dots,n\}}$$

but *i*, *j*, *k* should not be used for representing matrices themselves.In programming, names of variables often hint their types, usages, and meanings. Some people recommend that we should use descriptive names for global variables and short names for local variables [5].

1.4. Research Motivation

When you write a program, you have to decide names of variables, paying attention to their data types. For example, for variables of the integer type, you should adopt names such as *i*, *j*, *k*, preferably. In this paper, we propose anew mechanism of variable declaration in typed programming languages, which enables us to relate variable names to their types effectively and systematically using aregular expression.

First, we propose a simply-typed lambda calculus with regular expression bindings, called λ REG. To put it concretely, we extend the lambda calculus by adding let-expressions in which you can describe bound variables using regular expressions. The extended lambda calculus gives us a theoretical prototype of thevariable declaration with regular expressions. We give formal semantics to the calculus λREG both by defining reduction relation and by giving atranslation of λREG to the traditional lambda-calculus. We then study several theoretical properties.

1.5. Related Works

The regular expressions are incorporated into many programming languages; especially, scripting languages such as Perl and JavaScript [6] provide the regular expressions as a part of the languages' syntax. A regular expression in such languages, the matched text are a string data. On the other hand, the regular expressions in our calculus are matched with the variable identifiers.

Recently, the lambda calculus with regular types [7] is proposed, in which the regular expressions are introduced into the type system and the expressiveness of the types is extended. For example, a type $\alpha^* \to \beta$means

$$\alpha \to \alpha \to \cdots \to \beta,$$

intuitively. The regular expressions are a part of types and are matched with types, which is clearly different with the approach in this paper.

2. Lambda Calculus with Regular Expression Bindings

In this section, we propose the lambda calculus with regular expressions, λREG. We first formulate it as an untyped calculus and give a simple type theory [8] to the system.

2.1. Untyped Lambda Calculus with Regular Expression Bindings

We assume that we have a countable set \kwd{Var} of strings, whose elements are called *variables*.

Definition 1 (Terms and Values of λREG) Terms of λREG are defined inductively by the following grammar:

M	::=	c	Constant
	|	x	Variable
	|	$\$n$	$-variable
	|	$\lambda x.M$	Lambda abstraction
	|	(MN)	Function application
	|	$(let\ p = N\ in\ L)$	Pattern let-expression

The constant *c* represents a primitive data or a data constructor. The variable *x* is similar to the one of the lambda calculus.The $-variable *$n* represents an identifier designating the result of pattern matching with a regular expression, which will be explained intuitively in the following example. The lambda abstraction $\lambda x.M$ and the function application (MN) are similar to the those of the lambda calculus. The pattern let-expression $(let\ p = N\ in\ L)$ is an extension of the let-expression of the functional programming language [8].

The subset of the terms, the set of *values*, is defined by the following grammar, which is the set of evaluation results:

V	::=	c
	|	$\lambda x.M$

where *p* is a regular expression. In this paper, we describe regular expressions by the following syntax.

p	::=	ϵ	Empty
	|	a	Constant symbol
	|	pq	Concatenation
	|	$p \mid q$	Alternation
	|	$p*$	Kleene star

where a means a character.

We also use Unix-like notations of regular expressions for convenience. For example, [0-9] means

$$(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$$

and $p+$ means $pp*$

We use *L, M, N* for terms and *U, V, W* for values.We present an example of pattern binding in the following.

Example 1 Consider a pattern let-expression

$$\text{let } i([0-9]+) = (\text{succ } \$1) in\ (add\ i5\ i100)$$

where *i*([0-9]+) is a regular expression and succ a successor function. The string matched with the parenthesized part *i*([0-9]+) is referred by $1 like regular expressions in scripting languages such as Perl and PHP.

The patterni([0-9]+) is matched to variables appearing in the let's body, that is, i5 and i100. The $-variable $1 is substituted with 5 and 100, respectively.

We may consider the term as

let i5 = (inc 5)

and i100 = (inc 100)
in
(add i5 i100)

Next, we introduce a reduction relation as an operational semantics of λREG, after preparing a matching operation match.

Definition 2 (Function much) Suppose that p is a regular expression and M a term of λREG.Let $x_1, \dots, x_n$ be the variables that occurs freely in M and can be matched with p. Substitutionsσ_i $(i = 1, \dots, n)$are supposed to give matching between variables x_i and the patterns p, respectively. Function match is defined as follows:

$$\text{match}(p, M) = \big((x_1, \sigma_1), \dots, (x_n, \sigma_n)\big).$$

For example, let p be a pattern i([0-9]+) and a term M (add i5 i100). Then,

$$\text{match}(p, M) = ((\text{i5}, [\$1 \mapsto 5]), (\text{i100}, [\$1 \mapsto 100]))$$

The pattern p is matched to variables i5 and i100 through substitutions [$1↦ 5] and [$1 ↦ 100], respectively.

If let p' be h([0-9]+)w([0-9]+) and M' (add $h161w62$ $y170w75$), then
$\text{match}(p', M') = ((\text{h161w62}, [\$1 \mapsto 161, \$2 \mapsto 62]), (h170\text{w75}, [\$1 \mapsto 170, \$2 \mapsto 75]))$

We give an operational semantics as a reduction relation, or small-step semantics.

Next, we give an operational semantics to the calculus λREG as a small-step semantics.

Definition 3 (Reduction $M \to N$) We define a reduction relation as a binary relation $M \to N$ between terms M and N inductively by the following rules.

$$\frac{}{x \to x}\ \text{Var} \qquad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}\ \text{Lam}$$

$$\frac{M \to M'}{(MN \to M'N)}\ \text{AppL} \qquad \frac{N \to N'}{(MN \to M'N)}\ \text{AppL} \qquad \frac{M \to M'}{(\lambda x.M)N \to M\left[x \to N\right]}\ \text{Beta}$$

$$\frac{\text{match}(p, N) = ((x_1, \sigma_1), \dots\dots (x_n, \sigma_n))}{\text{let } p{=}N \text{ in } L \to L\left[x_1 \to \sigma_1(N), \dots\dots x_n \to \sigma_n(N)\right]}\ \text{Let3}$$

Readers should be noticed that instantiation of regular expressions is provided by rule \textrm{Let3} using the function match. We show an example of reduction sequence in λREG.

Example 2 (Reduction) Consider a term

Let i([0-9]+) = (inc $1) in (add i5 i100)

As already explained, we have

$$\text{match}\big(i([0-9]+), (\text{add } i5\ \text{i100})\big) = ((i5, \sigma_1), (\text{i100}, \sigma_2))$$

where $\sigma_1 = [\$1 \mapsto 5]$ and $\sigma_2 = [\$1 \mapsto 100]$.

$$\begin{aligned} &\text{let } i([0-9]+) = (\text{inc } \$1) \text{in } (\text{add i5 i100}) \\ \to\ &(\text{add i5 i100})[\text{i5} \mapsto \sigma_1(\text{inc } \$1)][\text{i100} \mapsto \sigma_2(inc\ \$1)] \\ &= (\text{add i5 i100})[\text{i5} \mapsto (\text{inc } 5)][\text{i100} \mapsto (\text{inc } 100)] \\ &= \text{add } (\text{inc } 6)(inc\ 100) \end{aligned}$$

A type system of λREG is introduced based on the simple type system of the lambda calculus. The syntax of the types is similar to the usual simply-typed lambda calculus.

Definition 4 (Types) Types of λREG*A, B, ...*are defined inductively by the following grammar.

$$A ::= \alpha \mid c \mid (A \to B)$$

where c means a constant type. Concretely, we consider a numeral type num, a string type string, etc.

Definition 5 (Type Assignment) A type assignment

$$\{x_1 : A_1, \ldots, x_n : A_n\}$$

is a partial mapping of variables $x_1, \ldots, x_n$ to types $A_1, \ldots, A_n$. We will use meta-variables $\Gamma, \Gamma', \ldots$ If type assignment Γ maps x to A, we write $\{x : A\} \in \Gamma$. We write an extension of Γadding correspondence between x and A, as $\Gamma\{x : A\}$.

Definition 6 (Typing Rules) Type judgement $\Gamma \vdash M : A$is a ternary relation among typing assignment Γ, term M, and type A defined inductively by the following rules.

$$\frac{\{x : A\} \in \Gamma}{\Gamma \vdash x : A}\ \text{Var} \qquad \frac{\Gamma\{x : A\} \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \text{Lam} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}\ \text{App}$$

$$\frac{\Gamma \vdash \sigma_i(M) : B_i\ (i = 1, \ldots, n) \quad \Gamma\{x_1 : B_1\} \cdots \{x_n : B_n\} \vdash N : A}{\Gamma \vdash (\text{let } p = M \text{ in } N) : A}\ \text{Let}$$

where $\text{match}(p, N) = \big((x_1, \sigma_1), \ldots, (x_n, \sigma_n)\big)$.

We next show an example of typing derivation.

Example 3 (Typing)

We consider typing of a term

$$\text{let } (\text{i}[0-9]+) = \lambda x.\$1 \text{ in } (\lambda y.\text{i}5)(\lambda z.\text{i}100)$$

$$\text{By Rule Var, we have } \Gamma\{x : B\} \vdash \text{i}100 : A. \tag{1.1}$$

$$\text{By RuleLam, (1.1) derives } \Gamma \vdash \lambda x.\text{i}100 : B \to A \text{ and therefore } \Gamma \vdash \sigma_1(\lambda x.\$1) : B \to A. \tag{1}$$

$$\text{By Rule Var, we have } \Gamma\{\text{i}5 : \text{A}\} \vdash \text{i}5 : A. \tag{2.1}$$

By Rule Lam, (2.1) derives $\Gamma \vdash \lambda x.\text{i}5 : \text{B} \to \text{A}$, and therefore

$$\Gamma \vdash \sigma_2(\lambda x.\$1) : B \to A. \tag{2}$$

$$\text{By Rule Var, we have } \Gamma\{\text{i}5 : A\}\{\text{i}100 : A\}\{y : D \to A\} \vdash \text{i}5 : A. \tag{3.1.1}$$

$$\text{Base Rule Lam and (3.1.1), we have } \Gamma\{\text{i}5 : A\}\{\text{i}100 : A\} \vdash \lambda\, \text{y.i}5 : (D \to A) \to A. \tag{3.1}$$

$$\text{By Rule Var, we have } \Gamma\{z : D\}\{i5 : A\}\{\text{i}100 : \text{A}\} \vdash \text{i}100 : A. \tag{3.2.1}$$

By Rule Lam and (3.1.1), we have

$$\Gamma\{\text{i}5 : A\}\{\text{i}100 : A\} \vdash \lambda z.\text{i}100 : D \to A. \tag{3.2}$$

$$\text{By (3.1) and (3..2), } \Gamma\{\text{i}5 : A\}\{\text{i}100 : A\} \vdash (\lambda y.\text{i}5)(\lambda z.\text{i}100) : A. \tag{3}$$

By (1), (2), and (3), we have $\Gamma \vdash \text{let } (\text{i}[0-9]+) = \lambda x.\$1 \text{ in } (\lambda y.\text{i}5)(\lambda z.\text{i}100): A$ (*)

The above reasoning can be written as the following derivation tree.

$$\dfrac{\dfrac{\dfrac{(1.1)}{(1)} \quad \dfrac{(2.1)}{(2)}}{} \quad \dfrac{\dfrac{(3.1.1)}{(3.1)} \quad \dfrac{(3.2.1)}{(3.2)}}{(3)}}{(*)}$$

3. Translation of λREG into the Lambda Calculus

In this section, we introduce a translation of λREG into the usual lambda calculus and discuss its theoretical properties.The translation of λREG gives a suggestion that a programming language with regular expression bindings can be implemented as a preprocessor.

Definition 7 (Translation of λREG) A translation trans(-) of terms of λREG into λ-terms is defined inductively by the following equations.

$$\begin{aligned}
&\text{trans}(x) = x,\\
&\text{trans}(\lambda x.M) = \lambda x.\text{trans}(M),\\
&\text{trans}(MN) = \big(\text{trans}(M)\text{trans}(N)\big), \text{trans}(N),\\
&\text{trans}(\text{let } p = N \text{ in } L) = \text{trans}(L)\big[x_1 \mapsto \text{trans}\big(\sigma_1(N)\big), \dots, x_n \mapsto \text{trans}\big(\sigma_n(N)\big)\big],
\end{aligned}$$

where $\text{match}(p, L) = \big((x_1, \sigma_1), \dots, (x_n, \sigma_n)\big)$.

The constructs except the pattern let-expressions are not changed by the translation trans.

We first show soundness of the translation \$\trans\$ with respect to the reduction: if $M \to M'$, then $\text{trans}(M) \to_\beta^* \text{trans}(M')$. In order to demonstrate the soundness property, we prepare several lemmas.

Lemma 1 (Substitution Lemma) For terms *M, N* and variables $x_i (i = 1, \dots, n)$. it holds that

$$\begin{aligned}
&\text{trans}(M[\text{x}_1 \mapsto \sigma_1(N), \dots, x_n \mapsto \sigma_n(N)])\\
&= \text{trans}(M)[x_1 \mapsto \text{trans}\big(\sigma_1(N)\big), \dots, x_n \mapsto \text{trans}(\sigma_n(N))]
\end{aligned}$$

(*Proof)* This lemma is proved by induction on the structure of term *M*. In the following, we show a proof of the case of the pattern let-expression; the other cases are easy.

$$\begin{aligned}
&\text{Let } M \text{ be } (\text{let } p' = N' \text{ in } L'). \quad \text{trans}\left((\text{let } p' = N' \text{in } L')\overline{[x_n \mapsto \sigma_n(N)]}\right)\\
&= \text{trans}\left(\left(\text{let } p' = \left(N'\overline{[x_n \mapsto \sigma_n(N)]}\right) \text{ in } \left(L'\overline{[x_n \mapsto \sigma_n(N)]}\right)\right)\right)\\
&= \left(L'\overline{[x_n \mapsto \sigma_n(N)]}\right)\overline{\left[x'_{n'} \mapsto \sigma'_{n'}\left(N'\overline{[x_n \mapsto \sigma_n(N)]}\right)\right]}\\
&= L'\overline{[x'_{n'} \mapsto \sigma'_{n'}(N')[x_n \mapsto \sigma_n(N)]}\\
&= \text{trans}\left(\text{let } p' = N' in\, L'\overline{[x_n \mapsto \sigma_n(N)]}\right)\\
&= \text{trans}(M)\overline{[x_n \mapsto \sigma_n(N)]}
\end{aligned}$$

(*End of Proof*)

Theorem 1 (Soundness of Translation Trans)

For a term *M*, if $M \to M'$, then

$$\text{trans}(M) \rightarrow_{\beta}^{*} \text{trans}(M')$$

(*Proof*) We prove this proof by induction on the structure of $\rightarrow$. Due to lack of space, we focus on thecaseof App and Let3.

Case of App. Let M and M' be $((\lambda x.L)N)$ and $L[x \mapsto N]$, respectively. We then have

$$\begin{aligned}&\text{trans}\big((\lambda x.L)N\big)\\&= \text{trans}(\lambda x.L)\text{trans}(N)\\&= \big(\lambda x.\text{trans}(L)\big)\text{trans}(N)\\&\rightarrow_{\beta} \text{trans}(L)[x \mapsto \text{trans}(N)]\end{aligned}$$

On the other hand,

$$\begin{aligned}&\text{trans}(L[x \mapsto N])\\&= \text{trans}(L)[x \mapsto \text{trans}(N)].\end{aligned}$$

since Lemma 1. Hence, we have

$$\text{trans}\big((\lambda x.L)N\big) \mapsto_{\beta} \text{trans}(L[x \mapsto N]).$$

Case of Let 3. Let M and M' be (let p=N in L) and $L[x_1 \mapsto \sigma_1(N), \dots, x_n \mapsto \sigma_n(N)]$,respectively. Moreover, we suppose that

$$\text{match}(p, L) = \big((x_1, \sigma_1), \dots, (x_n, \sigma_n)\big).$$

Then,

$$\begin{aligned}&\text{trans}(\text{let } p = N \text{ in } L)\\&= \text{trans}(L)\big[x_1 \mapsto \text{trans}\big(\sigma_1(N)\big), \dots, x_n \mapsto \text{trans}\big(\sigma_n(N)\big)\big]\\&= \text{trans}(L[x_1 \mapsto \sigma_1(N), \dots, x_n \mapsto \sigma_n(N)]\end{aligned}$$

since Lemma 1.

(*End of Proof*)

We next present invariance theorem of typing with respect to the translation: $\Gamma \vdash M: A$ if and only if $\Gamma \vdash \text{trans}(M): A$. Before proving the theorem, we prepare the following lemma.

Lemma 2

$$\Gamma\{x_1: A_1\} \cdots \{x_n: A_n\} \vdash M: A \text{and } \Gamma \vdash N_i: A_i (i = 1, \dots, n)$$

if and only if$\Gamma \vdash M[x_1 \mapsto N_1, \dots, x_n \mapsto N_n] : A$

This lemma is proved straight-forwardly by induction on the structure of term *M.*

Theorem 2 (Invariance of Typing) For a term M and a type A, $\Gamma \vdash M: A$if and only if $\Gamma \vdash \text{trans}(M): A$. (*Proof*) We prove this proof by induction on the structure of M. Due to lack of space, we focus on the case that M is a pattern binding (let p=N in L).

(Necessity) Suppose that $\text{match}(p, L) = \big((x_1, \sigma_1), \dots, (x_n, \sigma_n)\big)$and

$$\frac{\Gamma \vdash \sigma_i(N) : A_i \ (i = 1, \dots, n) \quad \Gamma\{x_1 : A_1\} \cdots \{x_n : A_n\} \vdash L : A}{\Gamma \vdash (\text{let } p = N \text{ in } L) : A} \text{ Let}$$

By the induction hypothesis, we have

$$\Gamma \vdash \text{trans}\big(\sigma_1(N)\big) : A_i \text{ and } \Gamma\{x_1: A_1\} \cdots \{x_n: A_n\} \vdash \text{trans}(L): A$$

By Lemma 2, we have

$$\Gamma \vdash \text{trans}(L)[x_1 \mapsto trans(\sigma_1(N_1), \ldots, x_n \mapsto \text{trans}(\sigma_n(N_n))] : A,$$

that is,

$$\Gamma \vdash \text{trans}(\text{let } p = N \text{ in } L) : A.$$

(Sufficiency) Suppose that match $(p, L) = ((x_1, \sigma_1), \ldots, (x_n, \sigma_n))$ and $\Gamma \vdash \text{trans}(\text{let } p = N \text{ in } L) : A$. Since

$$\text{trans}(\text{let } p = N \text{ in } L) = \text{trans}(L)[x_1 \mapsto \text{trans}(\sigma_1(N_1)), \ldots, x_n \mapsto \text{trans}(\sigma_n(N_n))],$$

we have

$$\Gamma \vdash \text{trans}(\sigma_i(N_i)) : A_i,$$

and

$$\Gamma\{x_1 : A_1\} \ldots \{x_n : A_n\} \vdash L : A.$$

By the induction hypothesis,

$$\Gamma \vdash \sigma_i(N_i) : A_i,$$

and

$$\Gamma\{x_1 : A_1\} \cdots \{x_n : A_n\} \vdash L : A.$$

By Lemma 2,

$$\Gamma \vdash L[x_1 \mapsto \sigma_1(N_1), \ldots, x_n \mapsto \sigma_n(N_n)] : A$$

this is,

$$\Gamma \vdash (\text{let } p = N \text{ in } L) : A.$$

(*End of Proof*)

4. Concluding Remarks

We proposed a simply-typed lambda calculus with regular expression bindings, λREG, by extending the lambda calculus by adding let-expressions in which bound variables are specified by regular expressions. We provide semantics to the calculus λREG both by defining reduction relation and by giving atranslation of λREG to the traditional lambda-calculus [9]. The former formulated our intuitive semantics and the latter gives us a theoretical basis for compilation. We then study several theoretical properties between the two semantics.

One of the future direction of our research is theincorporation of the regular expression bindings and the unification via thefirst-classenvironment [10].

The other kinds of the semantics of λREGare also interesting. For example, the abstract machine semantics [11]-[13]. The compatibility of the regular expression binding with the other programming paradigm such as object-oriented programming [14]-[16].

Acknowledgement

This paper is based on `Variable Bindings with Regular Expressions'' by T. Ohata and Shin-ya Nishizaki, appeared in the Proceedings of theInternational Conference on Advances in Information Technology and Mobile Communication 2013, and we extend it essentially adding the demonstration of the theoretical properties and supplementing the introduction and concluding remarks, essentially.

This work was supported by Grants-in-Aid for Scientific Research (C) (24500009).

References

[1] Ullman, J. D., Hopcroft, J. E., & Motwani, R. (2006). *Introduction to Automata Theory, Languages and Computation*, Pearson.

[2] Jones, S. P. (2003). *Haskell 98 Languages and Libraries: The Revised Report*. Cambridge University Press.

[3] Milner, R., Harper, R., & MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. The MIT Press. Sperber, M., Dybvig, R. K., Flatt, M., & Van, S. A. (2010). *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press.
[4] Kernighan, B. W., & Pike, R. (1998). *The Practice of Programming*. Addison-Wesley.
[5] ECMAscript 2015 language specification, the 6th edition. (2015). Retrieved from http://www.ecma-international. org/ecma-262/6.0/index.html
[6] Dundua, B., Florido, M., & Kutsia, T. (2015). Lambda calculus with regular types. *Proceedings of the 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, to appear.*
[7] Gunter, C. A. (1992). *Semantics of Programming Languages Structures and Techniques*. The MIT Press.
[8] Milner, R., Harper, R., & MacQueen, D. (1997). *The Definition of Standard ML (Revised)*. The MIT Press.
[9] Nishizaki, S. (2012). Incorporating first-order unification into functional language via first-class environments. *Proceedings of the SPIT 2012 Second International Joint Conference*.
[10] Nishizaki, S., Narita, K., & Ueda, T. (2015). Simplification of abstract machine for functional language and its theoretical investigation. *Journal of Software*, *10(10)*, 1148–1159.
[11] Narita, K., & Nishizaki, S. (2011). A parallel abstract machine for the rpc calculus. *Proceedings of the International Conference on Informatics Engineering and InformationScience – ICIEIS 2011. Communicationsin Computer and Information Science* (pp. 320–332).
[12] Nomura, K., & Nishizaki, S. (2014). Simple abstract machine with delimited continuations. *Proceedings of International Conference on Advances in Communication, Network, and Computing, Advances in Engineering and Technology Series* (pp. 371–380).
[13] Abadi, M., & Cardelli, L. (1996). *A Theory of Objects*. Springer-Verlag.
[14] Nishizaki, S., & Ikeda, R. (2012). Typed and untyped object calculi with first-class continuations. *Journal of Software Engineering*, *1*, 1–10.
[15] Matsumoto, S., & Nishizaki, S. (2013). An object calculus with remote method invocation. *Proceedings of the Second Workshop on Computation: Theory and Practice, WCTP2012, Proceedings in Information and Communication Technology*.

Takuya Ohata received his bachelor's degree in computer science from Tokyo Institute of Technology in 2008. His bachelor's thesis is entitled ``Regulatory compliance checking by model checking.'' In 2010, he received his master's degree in computer science with the master's thesis ``Variable declarations using regular expressions.'' He majored in Software Engineering and was interested in the theory of programming languages, formal methods, and system verification using model checkers.

He is now working in DENSO Corporation.

Shin-ya Nishizaki is an associate professor of computer science at Tokyo Institute of Technology, Japan, where he leads a research group on formal theory on software systems. He received his bachelor's, master's and doctorate degrees from Kyoto University, in mathematical sciences. Before joining Tokyo Institute of Technology in1998, Dr. Nishizaki held appointments in computer science as Associate Professor at Chiba University for 2 years and assistant professor at Okayama University for 2 years.