

YuchiKaml

Yuchiki

2018 Dec.

Contents

1	Introduction	1
2	YuchiKaml Language	1
2.1	Syntax	1
2.1.1	Expression	1
2.1.2	Actual Notation	2
2.1.3	Comment	3
2.1.4	Syntax Sugar	3
2.2	Semantics	3
2.2.1	Value	3
2.2.2	Evaluation Relation	4
2.2.3	Valid Execution	5
3	YuchiKaml Interpreter	5
3.1	Usage	6
3.2	Preprocess	6
3.3	Known Bugs and Issues	6

1 Introduction

YuchiKaml is a toy language. and YuchiKaml interpreter is an implementation of interpreter of YuchiKaml. Both are created in order to get accustomed to Sprache, a C#Parser Combinator Library. In this article, we introduce both the language and the interpreter.

2 YuchiKaml Language

YuchiKaml is a dynamic typed language with-ML like surface grammar.

2.1 Syntax

In this section, we define the syntax of YuchikiML.

2.1.1 Expression

First, we define the expressions of YuchikiML.

Definition 1 (surface expression). A *surface expression* of YuchiKaml is defined by the following BNF equations:

$$\begin{aligned}
e ::= & () \mid x \mid n \mid \text{true} \mid \text{false} \mid s \mid (e) \\
& \mid e \ e \mid !e \\
& \mid e * e \mid e / e \\
& \mid e + e \mid e - e \\
& \mid e \leq e \mid e < e \mid e \geq e \mid e > e \\
& \mid e = e \mid e \neq e \\
& \mid e \& e \\
& \mid e \parallel e \\
& \mid e \triangleright e \mid e \gg e \\
& \mid \text{if } e \text{ then } e \text{ else } e \mid \text{let } x \ \tilde{a} = e \text{ in } e \mid \text{let rec } f \ a_1 \ \tilde{a} = e \text{ in } e \mid \lambda x. e \\
& \mid e ; e
\end{aligned}$$

The operators defined in earlier rows have stronger precedences than the operators defined in later rows. For example, $1 + 2 * 3$ is not parsed as $(1 + 2) * 3$, but $1 + (2 * 3)$.

Example 1 (GCD). This is an example of a YuchiKaml source code of a program which calculates the greatest common divisor of 120 and 45.

Listing 1: gcd

```

let rec gcd m n =
  if m > n then gcd (m - n) n
  else if m < n then gcd m (n - m)
  else m
in gcd 120 45

```

2.1.2 Actual Notation

In real source codes, the symbols above are notated as follows:

\leq	<code><=</code>
\geq	<code>>=</code>
\neq	<code>!=</code>
$\&$	<code>&&</code>
\parallel	<code> </code>
\triangleright	<code> ></code>
\gg	<code>>></code>
$\lambda x. e$	<code>\x-> e</code>

2.1.3 Comment

YuchiKaml has two kinds of comments.

Line Comment: Characters from `//` to the end of line are ignored.

Bracket Comment: Characters from `(*` to `*)` are ignored.

2.1.4 Syntax Sugar

Some of the expressions shown above are syntactic sugars. We show the list and syntax sugars and how they are desugared.

$$\begin{aligned}
e_1 \triangleright e_2 &::= e_2 \ e_1 \\
e_1 \gg e_2 &::= \lambda x. \ g(fx) \\
\text{let } f \ a_1 \cdots a_n = e_1 \text{ in } e_2 &::= \text{let } f = \lambda a_1. \cdots \lambda a_n. \ e_1 \text{ in } e_2 \\
\text{let rec } f \ a \ b_1 \cdots b_n = e_1 \text{ in } e_2 &::= \text{let rec } f \ a = \lambda b_1. \cdots \lambda b_n. \ e_1 \text{ in } e_2 \\
e_1 ; e_2 &::= \text{let } _ = e_1 \text{ in } e_2
\end{aligned}$$

Definition 2 (expression). We call a desugared surface expression as just *expression*.

The semantics of expressions, introduced in a later section, is defined for *expressions*, not *surface expressions*.

2.2 Semantics

Then we define the semantics of expressions. We assume that the behaviour of built-in functions are given. That is, built-in functions are assumed deterministic.

2.2.1 Value

Values of expressions are listed as below:

$$\begin{aligned}
v(\text{value}) &::= n \mid b \mid s \mid \text{cl} \mid f_b \\
\rho(\text{valuation}) &\in \text{Var} \not\rightarrow \text{Val} \\
f_b(\text{built-in function}) &\in \text{His} \times \text{Val} \not\rightarrow \text{Val} \\
\text{cl}(\text{closure}) &::= (x, e, \rho) \\
h(\text{history}) &::= \langle (f_1, v_1); \cdots ; (f_n, v_n) \rangle
\end{aligned}$$

Here Var is the set of the variables and Val is the set of the values.

Note 1. We assume that all the built-in functions return the same value for the same input, if it does not diverge. This is enough to see whether different two ways of execution of a program is equivalent or not, given a situation.

2.2.2 Evaluation Relation

We have to define the behaviour of evaluation process of expressions clearly. To this end, we define the *evaluation* process of expression by a big-step semantics shown below.

An *evaluation relation* is a four-term relation of the form $\rho \vdash h; e \longrightarrow h'; v$. The evaluation rules of YuchiKaml are shown below:

$$\begin{array}{c}
\frac{}{\rho \vdash h; x \longrightarrow h; \rho(x)} \quad (\text{E-VAR}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; (x, e'_1, \rho') \quad \rho \vdash h'; e_2 \longrightarrow h''; v_2}{\rho \vdash h; e_1 \ e_2 \longrightarrow h'''; v'_1} \quad (\text{E-APP}) \\
\\
\frac{\rho \vdash h; e \longrightarrow h'; b}{\rho \vdash h; !e \longrightarrow h'; \llbracket ! \rrbracket b} \quad (\text{E-NOT}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 * e_2 \longrightarrow h''; n_1 \llbracket * \rrbracket n_2} \quad (\text{E-MUL}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 / e_2 \longrightarrow h''; n_1 \llbracket / \rrbracket n_2} \quad (\text{E-DIV}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 + e_2 \longrightarrow h''; n_1 \llbracket + \rrbracket n_2} \quad (\text{E-PLUS}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 - e_2 \longrightarrow h''; n_1 \llbracket - \rrbracket n_2} \quad (\text{E-MINUS}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 \leq e_2 \longrightarrow h''; n_1 \llbracket \leq \rrbracket n_2} \quad (\text{E-LEQ}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 < e_2 \longrightarrow h''; n_1 \llbracket < \rrbracket n_2} \quad (\text{E-LT}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 \geq e_2 \longrightarrow h''; n_1 \llbracket \geq \rrbracket n_2} \quad (\text{E-GEQ}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; n_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; n_2}{\rho \vdash h; e_1 > e_2 \longrightarrow h''; n_1 \llbracket > \rrbracket n_2} \quad (\text{E-GT}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; v_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; v_2}{\rho \vdash h; e_1 = e_2 \longrightarrow h''; v_1 \llbracket = \rrbracket v_2} \quad (\text{E-EQ})
\end{array}$$

$$\begin{array}{c}
\frac{\rho \vdash h; e_1 \longrightarrow h'; v_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; v_2}{\rho \vdash h; e_1 \neq e_2 \longrightarrow h''; v_1 \llbracket \neq \rrbracket v_2} \quad (\text{E-NEQ}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; b_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; b_2}{\rho \vdash h; e_1 \& e_2 \longrightarrow h''; v_1 \llbracket \& \rrbracket v_2} \quad (\text{E-AND}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; b_1 \quad \rho \vdash h'; e_2 \longrightarrow h''; b_2}{\rho \vdash h; e_1 \parallel e_2 \longrightarrow h''; v_1 \llbracket \parallel \rrbracket v_2} \quad (\text{E-OR}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; \text{true} \quad \rho \vdash h'; e_2 \longrightarrow h''; v_2}{\rho \vdash h; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow h''; v_2} \quad (\text{E-IF-TRUE}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; \text{false} \quad \rho \vdash h'; e_3 \longrightarrow h''; v_3}{\rho \vdash h; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow h''; v_3} \quad (\text{E-IF-FALSE}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; v_1 \quad \rho \cup \{x \mapsto v_1\} \vdash h'; e_2 \longrightarrow h''; v_2}{\rho \vdash h; \text{let } x = e_1 \text{ in } e_2 \longrightarrow h''; v_2} \quad (\text{E-LET}) \\
\\
\frac{\mu X. \rho \cup \{f \mapsto (x, e_1, X)\} \vdash h; e_2 \longrightarrow h'; v_2}{\rho \vdash h; \text{let rec } f \ x = e_1 \text{ in } e_2 \longrightarrow h'; v_2} \quad (\text{E-LETREC}) \\
\\
\frac{\rho \vdash h; e_1 \longrightarrow h'; f_b \quad \rho \vdash h'; e_2 \longrightarrow h''; v_2}{\rho \vdash h; e_1 \ e_2 \longrightarrow \langle h'', (f_b, v_2) \rangle; (f_b h'' v_2)} \quad (\text{E-APPBUILTIN}) \\
\\
\frac{}{\rho \vdash h; \lambda x. e \longrightarrow h; (x, e, \rho)} \quad (\text{E-ABS})
\end{array}$$

We write $\llbracket e \rrbracket$ for (h, v) such that $\emptyset \vdash \langle \rangle; e \longrightarrow h; v$.

2.2.3 Valid Execution

The evaluation process shown above is just one of the ways of possible evaluations. Real Implementation of YuchiKaml may be done in a more optimized way, or in a completely different way which gives the result equivalent to that of the process in Section 2.2.2. We define the valid ways of executions using the evaluation relation.

Definition 3 (valid execution). Assume that the behaviour of built-in functions are already given. An execution of a YuchiKaml expression e , which has the history h of built-in function calls and the return value v , is *valid* if $(h, v) = \llbracket e \rrbracket$.

3 YuchiKaml Interpreter

In this section, We introduce the YuchiKaml Interpreter in the linked repository.

3.1 Usage

YuchiKaml *file_name*: Execute *file_name*

YuchiKaml **-help**: Show Help

YuchiKaml **-test**: Run test

YuchiKaml **-verbose** Run the interpreter in verbose mode.

For further infomation, see help with YuchiKaml **-help**.

3.2 Preprocess

include<library_name>: Include library\library_name.

include"library_name": Include source_code's_directory\library_name.

3.3 Known Bugs and Issues

The parser cannot parse variables having keywords in its prefix.

The parser cannot parse variables like ifa, thena, and elsey.