Bonus Assignment 4: Functional programming

Deadline: December 30th, 2022, 23:59

1 Introduction

In this assignment, you will combine and adapt Assignment 2 and 3, to create a simple calculator in the style of a functional programming language. The program you write for this assignment is obtained by reusing and integrating your programs submitted for the previous assignments, and specified below.

This assignment is a bonus assignment, and as such does not have the same strict requirements as before. Your submission will be judged either as passing or as failing: a bonus point is awarded for submissions that are in line with the spirit of this assignment.

2 Interface

The program is compilable and works on the command line. It accepts one command line argument, namely the file from which it reads.

Input. The program reads a string of characters from the file named by the first command line argument (see Assignment 3 for reading and parsing expressions in the simply-typed lambda calculus).

Process. After parsing the input into an abstract syntax tree, the program performs type checking with respect to the type O (see Assignment 3 for type checking). The type checking process halts, and if type checking was succesful, the checked expression is converted to the untyped lambda calculus. If type checking was unsuccesful, the program exits and may report an understandable error message. Otherwise, given the converted expression in the untyped lambda calculus, the program repeatedly β-reduces or α-renames the current expression (if possible), and the program terminates when normal form is reached (see Assignment 2 for working with the untyped lambda calculus). Then the normal form is converted back into an expression of the simply-typed lambda calculus and the result is output.

Conversion. Details of the (back and forth) conversion are provided below.

3 Grammar

The input file is analyzed using the following Backus-Naur grammar:

$\langle\text{expr}\rangle ::= \langle\text{lvar}\rangle \mid$ `'('` $\langle\text{expr}\rangle$ `')'` $\mid$ `'\'` $\langle\text{lvar}\rangle$ `'^'` $\langle\text{type}\rangle\ \langle\text{expr}\rangle \mid \langle\text{expr}\rangle\ \langle\text{expr}\rangle$

$\langle\text{type}\rangle ::=$ `'O'` $\mid$ `'('` $\langle\text{type}\rangle$ `')'` $\mid \langle\text{type}\rangle$ `'->'` $\langle\text{type}\rangle$

where $\langle\text{lvar}\rangle$ stands for any variable name that starts with a lowercase letter. A variable name is alphanumerical. The only basic type provided is the type O. All the function types are built from it or other function types, e.g. $(O \to O)$ and $(O \to (O \to O))$ and $((O \to O) \to O)$ et cetera.

The following variables are special, in the following sense: the names of these variables must not be used as names following a lambda abstraction, and their types are assumed to be fixed in the environment. One may think of these special variables as constants.

$\Gamma = z : O,\ s : O \to O,\ p : O \to O \to O$

4 Conversion

Expressions of the simply-typed lambda calculus are converted as follows:

$$\begin{aligned} z &\longrightarrow \lambda x \lambda f(x) \\ s &\longrightarrow \lambda a \lambda x \lambda f(a(f(x))f) \\ p &\longrightarrow \lambda a \lambda b \lambda x \lambda f(a(b(x)f)f) \\ x &\longrightarrow x \\ (\lambda x^T\ M) &\longrightarrow (\lambda x\ M) \\ (M\ N) &\longrightarrow (M\ N) \end{aligned}$$

Put in words: the constants are replaced by the given associated untyped lambda expressions. Any other variable remains the same, and the type annotations in lambda abstractions are erased. The conversion must be performed recursively, that is, the subexpressions (i.e. body of an abstraction, or left and right operands of an application) must be converted too.

The expressions are converted backwards by recognizing (after α-renaming) the following pattern:

$$\begin{aligned} \lambda x \lambda f(x) &\longrightarrow z \\ \lambda x \lambda f(f(x)) &\longrightarrow s(z) \\ \lambda x \lambda f(f(f(x))) &\longrightarrow s(s(z)) \\ \lambda x \lambda f(f(f(f(x)))) &\longrightarrow s(s(s(z))) \end{aligned}$$

5 Example

The following example shows some input, that is type checked, the conversion forward, the reduction process, and the conversion backwards.

Input. $p(s\ z)(s\ (s\ z))$
Type checking. $\Gamma \vdash p(s\ z)(s\ (s\ z)) : O$, which follows from

- $\Gamma \vdash p(s\ z) : O \to O$, which follows from
  - $\Gamma \vdash p : O \to O \to O$ from context
  - $\Gamma \vdash s\ z : O$, which follows from
    - $\Gamma \vdash s : O \to O$ from context
    - $\Gamma \vdash z : O$ from context
- $\Gamma \vdash s\ (s\ z) : O$, which follows from
  - $\Gamma \vdash s : O \to O$ from context
  - $\Gamma \vdash s\ z : O$ same as above

Reduction (not showing all steps).
$p(s\ z)(s\ (s\ z))$
$p(s\ z)(\lambda x \lambda f(f(f\ x)))$
$p(\lambda x \lambda f(f\ x))(\lambda x \lambda f(f(f\ x)))$
$\lambda a \lambda b \lambda x \lambda f(a(b(x)f)f)(\lambda x \lambda f(f\ x))(\lambda x \lambda f(f(f\ x)))$
$\lambda b \lambda x \lambda f((\lambda x \lambda f(f\ x))(b(x)f)f)(\lambda x \lambda f(f(f\ x)))$
$\lambda b \lambda x \lambda f((\lambda f(f(b(x)f)))f)(\lambda x \lambda f(f(f\ x)))$
$\lambda b \lambda x \lambda f(f(b(x)f))(\lambda x \lambda f(f(f\ x)))$
$\lambda x \lambda f(f((\lambda x \lambda f(f(f\ x)))(x)f))$
$\lambda x \lambda f(f((\lambda f(f(f(x))))f))$
$\lambda x \lambda f(f(f(f(x))))$
Output. $s(s(s(z)))$