COMPSCI 3MI3 : Project 2 - Untyped λ-Calculus

Fall 2021

Nicholas Moore

Project Due: Sunday October 31th at 11:59 PM

When complete, submit the code for this part as ULC.hs through the Avenue dropbox.

Part 1: Implementing Untyped λ-Calculus

1. (5 points) **Term Definitions**
   The grammar of the untyped λ-Calculus is given below.

   $\langle t \rangle ::= \langle Var \rangle$
   $\quad | \quad \lambda \, \langle Var \rangle \, . \, \langle t \rangle$
   $\quad | \quad \langle t \rangle \, \langle t \rangle$

   $\langle Var \rangle ::=$ A | B | C | D | E | F | G | H | I | J | K | M | N | O | P | Q | R | S | U | V | W | X | Y | Z

   Unlike our terms of UAE, in λ-Calculus, we need to be able to handle something close to the idea of algebraic variables. So, we will add the following restriction. In our Haskell implementation, what we mean by "variable" is one element from a set of labels. This set contains every letter of the standard English alphabet. You are allowed to skip a few of these if you are using them in your term data type.

   Implement this grammar as a recursive datatype in Haskell. Both data types will need to derive Eq, and it would be convenient for them to also derive Show.

2. (5 points) **Free Variables**
   Implement a function which takes in a term as defined above, and returns a list containing the free variables in that term. The rules for finding these are as follows:

   $$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.t_1) &= FV(t_1) \setminus \{x\} \\ FV(t_1 t_2) &= FV(t_1) \cup FV(t_2) \end{aligned}$$

   This will require use of Haskell lists. You may find some of the functions in `Data.List` useful:

   - `union`
   - `\\`
   - `nub` (as a last resort!)

   For more information, check the documentation at `https://hoogle.haskell.org/`

3. (5 points) **Relabelling**
   In order to implement our rules of substitution, we will need a way to relabel terms. In general, terms that differ only in the names of bound variables are interchangeable in all contexts. Write a function in Haskell which, given a term, the name of a variable to replace, and the name of a variable to replace it with, replaces each occurrence of the first variable with the second.

   You may assume all occurrences of the variable we are replacing are bound. You may also assume that, if a subexpression contains a lambda abstraction over the same variable name, that we are replacing that variable name too.

4. (10 points) **Substitution**
   The rules of substitution are given as follows:

Definition [Substitution]:

$$\begin{array}{llll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & [x \mapsto s]t_1\ [x \mapsto s]t_2 &
\end{array}$$

Write a function in Haskell named `sub` which performs substitution over the terms defined in question 1. This function should accept as arguments:

- The term the substitution is being performed on
- The variable label being substituted out.
- The term that is being substituted in.

Your function should also implement the meta-rule implicit in the above substitution rules, that if the abstracted variable is in the set of free variables of the term being substituted in, it must be relabelled with a label that does not conflict with any of the other labels. The function you wrote in the previous part should help with that.

The function should output the term produced as a result of the substitution operation.

5. (5 points) **Normal Forms**
Write a Haskell function, `isNF` which accepts as an argument a term of our calculus, and outputs a Boolean value (that is, a Haskell `Bool`, not a Church Boolean). This function must determine whether or not a term is in normal form, according to the call by value evaluation strategy.

6. (5 points) **Small Step Operational Semantics**
Consider the operational semantics of the call by value evaluation strategy of λ-Calculus.

$\rightarrow$ *(untyped)*

Syntax

$$\begin{array}{lll}
t ::= & & \textit{terms:} \\
 & x & \textit{variable} \\
 & \lambda x.t & \textit{abstraction} \\
 & t\ t & \textit{application} \\
 & & \\
v ::= & & \textit{values:} \\
 & \lambda x.t & \textit{abstraction value}
\end{array}$$

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

Using these operational semantics, write a function, `ssos`. This function will take a term as input, and will output the result of a single step of our call by value evaluation strategy.

7. (2 points) **Full Evaluation** Write a function, `eval`, which, when applied to a term, fully evaluates it.

Part 2: Implementing Some Functions

When you have completed this section, submit the code as ULC2.hs to the Avenue dropbox.

8. **Boolean Functions**

(a) (1 point) **Logical Not**
Write a Haskell function which uses your newly developed λ-Calculus to implement a logical not operation over Church Booleans. This function should take a term of our calculus, and return a term of our calculus. You are not allowed to use pattern matching for this question!

(b) (1 point) **Logical And**
Repeat the above for a logical and operator. This function should take two terms of our calculus, and return a third.

(c) (1 point) **Logical Or**
Repeat the above for a logical or operator. This function should take two terms of our calculus, and return a third.

9. **Numeric Functions**

(a) (2 points) **Numeric Sum**
Write a Haskell function which uses your newly developed λ-Calculus to implement addition over church numerals. This function should take two terms of our calculus and return a third. You are not allowed to use pattern matching for this question, you have to do it exclusively in λ-Calculus!

(b) (2 points) **Numeric Times**
Repeat the above for a numeric multiplication operator. This function should take two terms of our calculus, and return a third.

(c) (3 points) **Predecessor**
Repeat the above for a predecessor function over church numerals. This function should take one term of our calculus, and return another.