Assignment 7

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Wednesday, November 4, 2020

Submission:

1. Submit the following files via https://handins.ccs.neu.edu/courses/119:

   - `Assignment07.hs`
   - `Church.hs`
   - `Compiler.hs`
   - `Syntax.hs`
   - `Eval.hs`

2. This assignment is meant to be worked on and submitted in pairs, but you can choose to work on your own. Note, that you need to have a team on Handins to be able to submit (a singleton team or a pair).

3. At the very top, the file should contain a preamble following this template.

   ```
   {- |
   Module      :  Assignment07
   Description :  Assignment 7 submission for CS 4400.
   Copyright   :  (c) <your name>

   Maintainer  :  <your email>
   -}

   module Assignment07 where
   ```

   The file should contain a `main` function which runs all tests in the submission, as well as additional definitions required by this assignment.

4. Every top-level definition must include a purpose statement (for functions) and a type signature, followed by one or more defining equations.

5. Double-check that you have named everything as required and that functions required by this assignment have the correct type signatures.

6. Make sure your file loads into GHCi or can be compiled by GHC without any errors. **Your grade might be reduced by up to 50% if your code does not compile and run.**

Purpose: To experiment with translating a higher-level language to a minimal core calculus and to practice working with Church encodings.

Overview

In this assignment, we will perform an experiment: we will take a subset of protoScheme and compile it into pure lambda calculus. This entails: translating values (booleans and numbers, restricted to naturals), translating arithmetic and boolean operations, an if conditional, and finally, translating function calls, and global value and function definitions.

We will limit ourselves to a subset of protoScheme outlined below. However, do not modify your `Syntax` or `Eval` module by removing things. This experiment will exist in parallel to your main implementation.

(a) values: only integers ≥ 0 (i.e., natural numbers) and booleans

(b) expressions:

```
<Expr> ::= <Variable>
         | (+ <Expr> <Expr>)
         | (- <Expr> <Expr>)
         | (* <Expr> <Expr>)
         | (let (<Variable> <Expr>) <Expr>)
         | (if <Expr> <Expr> <Expr>)
         | (and <Expr> <Expr>)
         | (or <Expr> <Expr>)
         | (not <Expr>)
         | (< <Expr> <Expr>)
         | (> <Expr> <Expr>)
         | (= <Expr> <Expr>)
         | (<Variable> <Expr> <Expr>*)
```

(c) programs: global definitions with an expression

The goal is to have enough in place to be able to compile a factorial program and run it by applying normal reduction to it.

You are expected to refer to the following notes provided online:

- "Introduction to Lambda Calculus", https://vesely.io/teaching/CS4400f20/l/10/10.pdf
- "Programming in Pure Lambda Calculus", https://vesely.io/teaching/CS4400f20/l/13/13.pdf

Do ask questions. This assignment doesn't require you to write a huge amount of code, but some of the concepts might feel a little difficult at first. The deadline for this assignment allows you to

Pack

The assignment pack contains the following:

`Lambda.hs` Syntax of pure lambda calculus.

`Reduce.hs` Normal order reduction for pure lambda calculus. Contains the functions `normalize` and `normalizeWithCount`, which you will use in this assignment.

`Church.hs` Church encodings of values and operations. **You will need to complete this file.**

`Compiler.hs` Compilation from protoScheme expressions/programs into pure lambda calculus. **You will need to modify this file.**

`SimpleTests{Color}.hs` No changes.

`SExpression.hs` No changes.

`Parser.hs` No changes.

Questions

1. [In `Church.hs`] Complete the implementation of `toChurchBool`, a function converting Haskell booleans to Church-encoded booleans, and its reverse, `fromChurchBool`, a function for converting a Church boolean (in normal form) into its Haskell counterpart:

   ```
   toChurchBool :: Bool -> Lambda
   fromChurchBool :: Lambda -> Maybe Bool
   ```

   The function `fromChurchBool` should assume that the `Lambda` term is normalized, that is, the term will be in the canonical form presented in the lecture (or the notes) for true or false. However, due to alpha-equivalence, do you should not assume anything about the names of bound variables. For any `Lambda` term that is not a canonical Church boolean, the function should return `Nothing`.

   The two conversion functions should satisfy the following property, for all `b`

   ```
   fromChurchBool (toChurchBool b) == Just b
   ```

   Write tests as `test_bools`

2. [In `Church.hs`] Implement a function converting an `Integer` x, where $x \geq 0$ to a Church numeral and its reverse, a conversion from a Church numeral to an integer.

   ```
   toNumeral :: Integer -> Lambda
   fromNumeral :: Lambda -> Maybe Integer
   ```

The function `fromNumeral` should assume that the `Lambda` term is normalized, that is, the Church numeral will be in the canonical form presented in the lecture (or the notes). However, do not assume anything about the names of bound variables. For any `Lambda` term that is not a canonical Church numeral, the function should return `Nothing`.

These operations should satisfy the following, for all `i` ≥ 0:

```
fromNumeral (toNumeral i) == Just i
```

Write tests as `test_numerals`

3. [In `Church.hs`] Complete the tests and definitions of Church-encoded value operations. Writing *at least* one test for each, in the style of the example provided for `cplus`. Write the tests in the respective `test_` definition, replacing `undefined`.
4. [In `Compiler.hs`] Complete the `compile :: Expr -> L.Lambda` function which will translate a protoScheme expression into pure lambda calculus. You only need to cover the above constructs. For all other cases, return `Nothing`.
5. [In `Compiler.hs`] Complete the `compileProgram` function, which should take a protoScheme program with global definitions and translate it into a pure lambda expression. Global definitions can be translated the same way let expressions are translated. Function definitions should be recursive, using the fixed point combinator, which we will discuss in class this week.
6. [In `Compile.hs`] Complete the function `factorialProgram :: Integer -> String`, which generates the source code for a protoScheme factorial program, calculating the factorial of the given number. Write a test which generates a program for a low but interesting number (greater than 2), parses it into a protoScheme expression, compiles it into pure lambda calculus, runs it using `normalize`, (provided in `Reduce.hs`) and converts the result back into an integer. Note: this will be *slow*. Like, *very* slow. Make sure you test your compiler with simpler functions first.
7. Run the program with 4 as the parameter using `normalizeWithCount`. This function will return the normalized lambda term (which should represent the factorial of 4) paired up with the total number of reduction steps it took to normalize the lambda expression. In `Assignment07.hs`, write the number of steps taken to reduce factorial of 4 as a global variable `fact4StepsBefore`. Modify your `compileProgram` function so that it normalizes function bodies (using `normalize`). This is a simple optimization step, optimizing the function body before it runs on actual arguments. Rerun the factorial calculation with `normalizeWithCount` and write the number of steps as `fact4StepsAfter` in `Assignment07.hs`. If you have time, try the above experiment with the factorial of 5 and add your results as `fact5StepsBefore` and `fact5StepsAfter`.