CSCI 365 Problem Set 10: Lambda Calculus

Due Friday, 19 April 2024

Specification

To receive credit for this problem set:

- Complete at least 9 out of 13 exercises.
- Submit a text file with the definitions of your lambda calculus terms, in a format suitable for loading into the lambda calculus evaluator (*i.e.* the output of running `save filename.lc` from within the lambda calculus evaluator).
- Optionally, submit a PDF with your response to exercise 13.

Instructions

While working on this problem set you should use the command-line λ-calculus interpreter, which is available from the course website. Download and unzip, then execute `stack run` at a command prompt from within the unzipped directory. Please try it early and let me know if you have trouble getting it installed!

Note that files `bool.lc` and `nat.lc` are available with some of the definitions for Booleans and natural numbers that we went over in class.

If you wish to typeset any lambda calculus terms in LaTeX (though this is not required), you can use the following commands:

```
\newcommand{\lam}[2]{\ensuremath{\lambda #1. \, #2}}
\newcommand{\app}[2]{\ensuremath{#1 \; #2}}
```

For example,

```
\lam{x}{\lam{y}{\app{x}{y}}}
```

produces

$$\lambda x.\, \lambda y.\, x\ y.$$

These commands ensure proper spacing after the period of a λ and between terms in an application.

Natural numbers

Recall from lecture that we can represent natural numbers in the λ-calculus by their *Church encoding*, that is, the natural number n is represented by the λ-calculus term

$$\lambda s.\, \lambda z.\, s\ (s\ \ldots (s\ z))$$

where the s is repeated n times. In other words, a natural number is *represented by its own fold*, that is, a function which takes as arguments a function s and starting value z, and applies s to z a certain number of times.

We will abbreviate Church-encoded natural numbers as n_λ. For example,

$$3_\lambda = \lambda s.\, \lambda z.\, s\ (s\ (s\ z)).$$

In order to test your natural number functions in the λ-calculus evaluator, you will want to evaluate things like, *e.g.*, `plus two three S Z` instead of just `plus two three`. The reason is that reduction gets "stuck" when the outermost term constructor is a λ. In order to "fully reduce" a Church-encoded number like `plus two three`, you can apply it to some arguments, in this case, just two free variables `S` and `Z` to stand in for successor and zero.

Exercise 1 Define a λ-calculus term *exp* that exponentiates Church numerals, that is,

$$exp\ m_\lambda\ n_\lambda \equiv (m^n)_\lambda.$$

Exercise 2 Define a λ-calculus term *iszero* that decides whether a Church numeral is zero. That is, when applied to a Church numeral, it should evaluate to an appropriate Church-encoded boolean.

Exercise 3 Now define *iseven*, which tests whether a Church numeral is even.

Church lists

Exercise 4 Define λ-calculus terms *nil* and *cons* which represent the constructors for (Church-encoded) lists.

Remember that we will encode lists as their own folds! You may find it helpful to write out the recipe in Haskell first.

Exercise 5 Define a λ-calculus term *sum* such that, for example,

$$sum\ (cons\ 3_\lambda\ (cons\ 1_\lambda\ (cons\ 4_\lambda\ nil))) \equiv 8_\lambda.$$

Exercise 6 Define a λ-calculus term *filter* which works similarly to Haskell's standard `filter` function.

Church pairs and subtraction

Exercise 7 Define λ-calculus terms *pair*, *fst*, and *snd* such that

$$fst\ (pair\ x\ y) \equiv x$$

(and similarly for *snd*).

Hint:

Exercise 8 Define a λ-calculus term *pred* such that when n is positive, *pred* applied to n_λ is equivalent to $(n-1)_\lambda$ (*pred* applied to zero can just yield zero).

This problem is tricky! If you are stuck, feel free to ask me for a hint. Do not try to find hints online.

Exercise 9 Now define a λ-calculus term *sub* that subtracts Church numerals (truncating at zero in the case of subtracting a larger number from a smaller).

Exercise 10 Define a λ-calculus term *gte* that tests whether one Church numeral is greater than or equal to another. That is, for example,

$$gte\ 5_\lambda\ 4_\lambda \equiv true.$$

Recursion and the Y combinator

Watch this video lecture to learn about `fix` / the Y combinator, which allows us to encode arbitrary recursion in the λ-calculus:
`https://www.youtube.com/watch?v=tNhTu7uRKfY`

Exercise 11 Using the Y combinator, define a λ-calculus term *factorial* which computes the factorial of any Church numeral.

Exercise 12 Look up the *Ackermann function* and encode it as a λ-calculus term, using the Y combinator.

Further Exploration

Exercise 13 Take a look at *An Unsolvable Problem of Elementary Number Theory* by Alonzo Church[1], available from `https://www.jstor.org/stable/2371045`. This was not the very first paper to introduce the λ-calculus, but it is one of the first where the λ-calculus is more or less recognizable as we use it today.

Skim through the beginning of the paper and *either*:

- Read section 1 and footnote 3, and explain the significance of footnote 3. (Note also that Turing's paper introducing the Turing Machine—`https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf`—was published later in the same year, 1936.)
- *Or*, read section 2 and explain how Church's notation and operations I and II correspond to things we have discussed in class. (Church's operation III is called η-conversion, which we did not discuss.)

[1] Alonzo Church. American Journal of Mathematics, Vol. 58, No. 2 (Apr., 1936), pp. 345–363