The (untyped) λ-calculus

Theory for Programming Languages
MASTER IN FORMAL METHODS FOR SW ENGINEERING
UCM / UPM / UAM

November 12, 2021

Foundations

Exercise 1. a) Classify the following terms according to α-equivalence:
$\lambda x.x\ y$, $\lambda x.x\ z$, $\lambda y.y\ z$, $\lambda z.z\ z$, $\lambda z.z\ y$, $\lambda f.f\ y$, $\lambda f.f\ f$, $\lambda y.\lambda x.x\ y$, $\lambda z.\lambda y.y\ z$.
b) Provide an α-equivalent term where each abstraction uses a different variable name:
$\lambda x.((x\ (\lambda y.x\ y))(\lambda x.x))(\lambda y.y\ x)$.

Exercise 2. Normalize the following term: $(\lambda x.(\lambda y.x\ y))\ y$.

Exercise 3. The *de Bruijn index notation* is a way of avoiding the problems related to substitution and variable capture in Church's original presentation of the λ-calculus, thus facilitating its mechanized treatment. The key idea is to replace variable names by numbers denoting the *depth* of the scope of that variable. For example, the familiar terms $\lambda x.x$, $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$ are represented as $\lambda 1$, $\lambda\lambda 2$ and $\lambda\lambda 1$ in de Bruijn's notation. Free variables are represented by numbers higher than the maximum depth in its location. For example, $\lambda\lambda 3$ is a possible representation for $\lambda x.\lambda y.w$.

a) Represent the term $(\lambda x.\lambda y.\lambda z.x\ z\ y)(\lambda x.\lambda y.x)$ in de Bruijn's notation.
b) Explain how β-reduction of terms in de Bruijn notation can be implemented.
c) Apply your ideas to the application in a).

Exercise 4. *Combinators* can be seen as λ-terms without free variables — although they were actually proposed independently from the λ-calculus. Given the combinators $\mathsf{S} = \lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$, $\mathsf{K} = \lambda x.\lambda y.x$ and $\mathsf{I} = \lambda x.x$, prove the equivalence $\mathsf{SKK} = \mathsf{I}$.

Exercise 5. Combinator systems are commonly presented as equational theories where a *combinator base* is defined using oriented equational rules and new combinators are created by means of application alone — no abstraction is required. For example, the aforementioned combinators would be defined by the equations $\mathsf{S}mno = mo(no)$, $\mathsf{K}ab = a$ and $\mathsf{I}x = x$. As variables only appear in definitions where no confusion of scopes can happen, combinators solve in a natural way many of the nuissances associated with variable names in Church's formulation of the λ-calculus.

Take as base the following two combinators: $\mathsf{B}fgx = f(gx)$ and $\mathsf{M}x = xx$. Using B and M alone prove the existence of a *narcissistic* combinator n such that $nn = n$.

Constructive mathematics in the λ-calculus

Exercise 6. Using Church's encoding of booleans in the pure λ calculus, define normalized λ-terms CONJ, DISJ and NEG to represent conjunction, disjunction and negation, respectively. (Hint: define a λ-term COND which behaves as an *if-then-else* and apply β-reduction.)

Exercise 7. Using Church's encoding of natural numbers define addition, multiplication and exponentiation.

Exercise 8. Devise an encoding for pairs in the λ calculus. Provide λ-terms PAIR (a pair constructor) and the projections FST and SND. What properties would be required in order to check the correctness of the encoding?

Exercise 9. Define a predecessor function and subtraction for Church numerals. (Hint: You may use the pairs just defined.)

Exercise 10. Imagine that the list $[a_1, a_2 \ldots a_n]$ is represented in the lambda calculus by the term

$$\lambda f.\lambda x.f\ a_1\ (f\ a_2(\ldots f\ a_n\ x))$$

Define:

a) NIL, the empty list constructor,
b) APP, a function to concatenate two lists,
c) HD, which returns the first element of a nonempty list, and
d) a function ISEMPTY to check whether a list is empty..

The λ-calculus in Haskell

(or your language of choice)

Exercise 11. Using Haskell, define functions `boolChurch` and `boolUnchurch` which translate Haskell booleans into Church booleans and vice versa. Use them to check the correctness of your solutions to exercise 6.

Exercise 12. Analogously to the previous exercise, define Haskell functions to convert between Haskell and Church natural numbers, and check the correctness of your solutions to exercise 7.

Exercise 13. Define a Haskell datatype to represent the abstract syntax of the λ-calculus.

Exercise 14. Based on the previous exercise, define a Haskell function that obtains the free variables of a lambda term.

Exercise 15. Based on the previous exercises, define a Haskell function that implements *capture avoiding substitution.*

Exercise 16. Based on the previous exercises, define a Haskell function that implements β-reduction (one step).

Exercise 17. Based on the previous exercises, define a Haskell function that reduces a lambda term into β-normal form when possible .