Encoding Data in Lambda Calculus: An Introduction

Frank(Peng) Fu

September 26, 2017

Abstract

Lambda calculus is a formalism introduced by Alonzo Church in the 1930s for his research on the foundations of mathematics. It is now widely used as a theoretical foundation for the functional programming languages (e.g. Haskell, OCaml, Lisp). I will first give a short introduction to lambda calculus, then I will discuss how to encode natural numbers using the encoding schemes invented by Alonzo Church, Dana Scott and Michel Parigot. Although we will mostly focus on numbers, these encoding schemes also works for more general data structures such as lists and trees. If time permits, I will talk about the type theoretical aspects of these encodings.

1 Introduction to Lambda Calculus

Lambda calculus was invented by Alonzo Church, a lot of early results are due to him and his students. Currently, the definitive reference for lambda calculus is the book by Henk Barendregt [1].

Definition 1 (Lambda Calculus) *The set of lambda term Λ is defined inductively as following.*

- $x \in \Lambda$ *for any variable* x.
- *If* $e \in \Lambda$, *then* $\lambda x.e \in \Lambda$.
- *If* $e_1, e_2 \in \Lambda$, *then* $e_1\ e_2 \in \Lambda$.

Some computer scientists express lambda terms as: $e, n\ ::=\ x \mid e_1\ e_2 \mid \lambda x.e$. Lambda terms are almost symbolic, except we only consider lambda terms modulo *alpha-equivalence*, i.e. we view $\lambda x.e$ as the same term as $\lambda y.[y/x]e$, where y does not occurs in e.

Definition 2 *Beta-reduction:* $(\lambda x.e_1)\ e_2 \rightsquigarrow [e_2/x]e_1$, *where* $[e_2/x]e_1$ *means the result of replacing all the variable* x *in* e_1 *by* e_2.

Note that we allow beta-reduction occurs anywhere inside a lambda term. Let us see some examples.

Example 1 $(\lambda x.x\ x)\ (\lambda x.x) \rightsquigarrow (\lambda x.x)\ (\lambda x.x) \rightsquigarrow \lambda x.x$
$(\lambda x.x\ x)\ (\lambda x.x\ x) \rightsquigarrow (\lambda x.x\ x)\ (\lambda x.x\ x) \rightsquigarrow ...$
$(\lambda x.(\lambda y.y)\ x)\ (\lambda x.x\ x) \rightsquigarrow (\lambda x.x)\ (\lambda x.x\ x) \rightsquigarrow (\lambda x.x\ x)$

The beta-reduction can be intuitively understood as performing computation step by step, and sometimes computation diverges. If a lambda term can not be further reduced by beta-reduction, then it is in *normal form*.

Theorem 1 (Church-Rosser/Confluence) *If* $e \rightsquigarrow^* e_1$ *and* $e \rightsquigarrow^* e_2$, *then there exist a* e' *such that* $e_1 \rightsquigarrow^* e'$ *and* $e_2 \rightsquigarrow^* e'$. *Here* $\rightsquigarrow^*$ *means performing reduction in one or more steps.*

2 Lambda Encoded Numbers

One reason that lambda calculus is interesting is that we can use lambda term to represent algorithm and let the beta-reduction to *calculate* the result.

Definition 3 (Fixpoint combinator) *A fixpoint combinator in lambda calculus is a term n such that $n\ e \rightsquigarrow^* e\ (n\ e)$ for any lambda term e.*

Example 2 (Turing's Fixpoint) *Let A be $\lambda x.\lambda y.y\ (x\ x\ y)$. Then $A\ A$ is a fixpoint combinator. Let e by any lambda term. $A\ A\ e = (\lambda x.\lambda y.y\ (x\ x\ y))\ A\ e \rightsquigarrow e\ (A\ A\ e)$.*

We usually write fix to means a fixpoint combinator, and compuationally it behaves as $\mathsf{fix}\ e \rightsquigarrow^* e\ (\mathsf{fix}\ e)$.

Fixpoint combinator allows us to represent recursive definition such as $a = G\ a$, (where G is a lambda term) by a lambda term $a = \mathsf{fix}\ G$

2.1 Scott Encoded Numbers

Scott encoded numbers, or more generally, Scott encoding, is attributed to Dana Scott[1].

Definition 4 (Scott numerals)
Zero $\mathsf{Z} := \lambda z.\lambda s.z$
Successor $\mathsf{S} := \lambda n.\lambda z.\lambda s.s\ n$
Pattern matching $\mathsf{caseN} := \lambda n.\lambda u.\lambda f.n\ u\ f$

The normal form of natural numbers in Scott encoding looks like: $\mathsf{one} := \lambda z.\lambda s.s\ \mathsf{Z}, \mathsf{two} := \lambda z.\lambda s.s\ \mathsf{one}, \mathsf{three} := \lambda z.\lambda s.s\ \mathsf{two}, \ldots$. With the pattern matching, we can define predecessor function for natural number: $\mathsf{pred} := \lambda n.\mathsf{caseN}\ n\ \mathsf{Z}\ (\lambda n'.n')$.

Definition 5 (Addition) *A typical recursive definition of addition:*

```
add n m = case n of
            Z -> m
           (S n') -> S (add n' m)
```

To obtain the Scott encoded addition, observe the followings.

```
add  = \ n . \ m.
          caseN n
                m
                (\ n' . S (add n' m))

G = \ r . \ n . \ m .
                caseN n
                      m
                      (\ n' . S (r n' m))

add = G add

add' = fix G
```

[1] Who himself did not remember this encoding according to folklore.

2.2 Church Encoded Numbers

Definition 6
pair $:= \lambda a.\lambda b.\lambda p.p\ a\ b$
p1 $:= \lambda p.p\ (\lambda x.\lambda y.x)$
p2 $:= \lambda p.p\ (\lambda x.\lambda y.y)$

If we have type, then $\mathsf{pair} : A \to B \to A \times B, \mathsf{p1} : A \times B \to A, \mathsf{p2} : A \times B \to B$.

Definition 7 (Church numerals)
Zero $\mathsf{Z} := \lambda z.\lambda s.z$
Successor $\mathsf{S} := \lambda n.\lambda z.\lambda s.s\ (n\ z\ s)$
Iteration $\mathsf{iterN} := \lambda n.\lambda u.\lambda f.n\ u\ f$

The normal form of each Church numeral looks like: $\mathsf{one} := \lambda z.\lambda s.s\ z, \mathsf{two} := \lambda z.\lambda s.s\ (s\ z), \mathsf{three} := \lambda z.\lambda s.s\ (s\ (s\ z)), ...$
Note that iterN $3\ u\ f \rightsquigarrow^* f\ (f\ (f\ u))$. One nice(?) thing about Church encoding is that it allows programming without using fixpoint combinators.

Definition 8 (Addition) *The addition function we seen has this property:*
$\mathsf{add}\ n\ m \rightsquigarrow^* \underbrace{(\mathsf{S}...(\mathsf{S}}_{n}\ m))$
Thus we can use iterator to represent addition. $\mathsf{add} := \lambda n.\lambda m.\mathsf{iterN}\ n\ m\ \mathsf{S}$.

Note that the number of beta-reduction steps that this addition takes is proportioned to the input n. Rosser has a much clever constant time definition of addition, which is $\mathsf{add} := \lambda n.\lambda m.\lambda z.\lambda s.n\ (m\ z\ s)\ s$. Rosser addition take four beta-reduction steps for any number n, m (assuming n, m are in normal forms).

Definition 9 (Predecessor) *Since in Church encoding, we do not have the pattern matching like Scott encoding, how are we going to represent predecessor? Idea: use pair and iteration:* $(0,0) \mapsto (0,1) \mapsto (1,2)...$*, i.e. to get the predecessor of* m*, we can iterate* m *times the function* $(n, n') \mapsto (n', n'+1)$ *over* $(0,0)$*, and project the left element on the result. This is due to Kleene.*
So $\mathsf{pred} := \lambda m.\mathsf{p1}\ (\mathsf{iterN}\ m\ (\mathsf{pair}\ \mathsf{Z}\ \mathsf{Z})(\lambda r.\mathsf{pair}\ (\mathsf{p2}\ r)\ (\mathsf{S}\ (\mathsf{p2}\ r))))$
Note that the number of beta-reduction steps that this predecessor function takes is proportioned to the size of input number m*, unlike Scott encoded predecessor, which take three steps.*

2.3 Parigot Encoded Numbers

Parigot encoding was proposed to enable constant time predecessor function while allowing programming with the iterators.

Definition 10 (Parigot numerals)
Zero $\mathsf{Z} := \lambda z.\lambda s.z$
Successor $\mathsf{S} := \lambda n.\lambda z.\lambda s.s\ n\ (n\ z\ s)$
Primitive recursion $\mathsf{recN} := \lambda n.\lambda u.\lambda f.n\ u\ f$

The normal form of each Church numeral looks like: $\mathsf{one} := \lambda z.\lambda s.s\ \mathsf{Z}\ z, \mathsf{two} := \lambda z.\lambda s.s\ \mathsf{one}\ (s\ \mathsf{Z}\ z), \mathsf{three} := \lambda z.\lambda s.s\ \mathsf{two}\ (s\ \mathsf{one}\ (s\ \mathsf{Z}\ z)), ...$ Note that $\mathsf{recN}\ 3\ u\ f \rightsquigarrow^* f\ 2\ (f\ 1\ (f\ 0\ u))$.

Definition 11
Addition $\mathsf{add} = \lambda n.\lambda m.\mathsf{recN}\ n\ m\ (\lambda a.\lambda r.\mathsf{S}\ r)$
Multiplication $\mathsf{mult} = \lambda n.\lambda m.\mathsf{recN}\ n\ \mathsf{zero}\ (\lambda a.\lambda r.\mathsf{add}\ m\ r)$
Predecessor $\mathsf{pred} = \lambda n.n\ \mathsf{Z}\ (\lambda a.\lambda r.a)$
Factorial $\mathsf{fac} = \lambda n.n\ \mathsf{one}\ (\lambda a.\lambda r.\mathsf{mult}\ (\mathsf{S}\ a)\ r)$

References

[1] HP Barendregt. The lambda calculus: Its syntax and semantics, volume 103 of studies in logic and the foundations of computer science, 1981.