LAMBDA CALCULUS: AN INTRODUCTION

Chandreyee Chowdhury

LISTS

A list may contain

- Nothing (empty)
- One thing
- Multiple things

List contains 2 values

- make_pair = λleft. λright. λf. f(left)(right)
- get_left = λpair. pair(true)
- get_right = λpair. pair(false)

A list will have the form # (empty, (head, tail))

null=make_pair(true)(true)

is_empty = get_left

is_empty(null) would return true

LISTS

null=make_pair(true)(true)

(prepend(two)(single_item_list))

prepend = λitem. λl. make_pair(false)

non-empty lists

- *[2, 1] ==> (empty=false, (2,(1,null)))*
- *# (false, (1,null))*
  - single_item_list = prepend(one)(null)
- *# (false, (3,(2,(1,null))))*

LISTS

```
hours_day_1=prepend(2)(null)
hours_per_day=prepend(three)(prepend(two)(hours_day_1))
head(tail(hours_per_day)
```

PREDECESSOR

the general strategy will be to create a pair (n,n-1) and then pick the second element of the pair as the result

- make_pair = λleft. λright. λf. f(left)(right)

make_pair (true)=left

make_pair (false)=right

Φ combinator generates from the pair (n, n-1) (which is the argument p in the function) the pair (n + 1, n)

Φ =λp. λf. f (succ(p true))(p true)

(succ(p true))(p true)

(zero,zero)→(one,zero)→(two,one)→(three,two)→…

The predecessor of a number n is obtained by applying n times the function to the pair (f zero zero) and then selecting the second member of the new pair

Pred=λn. n Φ

PREDECESSOR FUNCTION

PRED := λnfx.n (λgh.h (g f)) (λu.x) (λu.u)

PRED 1 <=>

(λnfx.n (λgh.h (g f)) (λu.x) (λu.u)) (λhy. h y) <=>

λfx. (λhy. h y) (λgh.h (g f)) (λu.x) (λu.u) <=>

λfx. (λgh.h (g f)) (λu.x) (λu.u) <=>

λfx. (λu.u) ((λu.x) f) <=>

λfx. (λu.u) x<=>

λfx. x <=>

0

ENCODING NATURAL NUMBERS IN LAMBDA CALCULUS

What can we do with a natural number?

- we can iterate a number of times

A natural number is a function that given an operation f and a starting value s, applies f a number of times to s:

$1 =_{def} \lambda f.\ \lambda s.\ f\ s$

$2 =_{def} \lambda f.\ \lambda s.\ f\ (f\ s)$

and so on

$0 =_{def} \lambda f.\ \lambda s.\ s$

COMPUTING WITH NATURAL NUMBERS

$$1 \equiv \lambda sz.s(z)$$
$$2 \equiv \lambda sz.s(s(z))$$

```
var successor = n => f => x => f(n(f)(x));
```

The successor function

successor n $=_{def}$ $\lambda n.\lambda f.\lambda x.f(nfx)$

Successor of 0 (S0) is $_{def}$ $(\lambda nfx.f(nfx))\ (\lambda sz.z)$

$$\lambda yx.y((\lambda sz.z)yx) = \lambda yx.y((\lambda z.z)x) = \lambda yx.y(x) \equiv 1$$

Successor of 1 (S1) is $_{def}$ $(\lambda wyx.y(wyx))\ (\lambda sz.s(z))$

Addition

$_{def}$ $\lambda m.\lambda n.\lambda f.\lambda x.m(f)(n(f)(x))$

2S3 is $_{def}$ $(\lambda sz.s(s(z)))(\lambda wyx.y(wyx))(\lambda uv.u(u(u(v))))$

ADDITION/MULTIPLICATION

(int X int) $\rightarrow$ int ---not pure lambda calculus

(int $\rightarrow$ int) $\rightarrow$ int ----pure form (using currying)

$\lambda x.\lambda y.x+y$

Evaluate $((\lambda x.\lambda y.y\ x)\ 3)$

Multiplication

def $\lambda m.\lambda n.\lambda f.\lambda x.m(n(f))(x)$

SUBTRACTION AND COMPARISON

Subtraction: m-n

- λm. λn. nPRED m

Comparison

- *greaterOrEqual*=λn. λm. isZero(subtract n m)
- *lessOrEqual*= λn. λm. isZero(subtract m n)
- *areEqual*= λn. λm. AND (*greaterOrEqual n m*) (*lessOrEqual n m*)

POLYMORPHISM

Functions that allow arguments of many types, such as this identity function, are known as **polymorphic operations**

- $((\lambda x \,.\, x)\ E) = E$

define Twice $= \lambda f \,.\, \lambda x \,.\, f\ (f\ x)$

- If D is any domain, the syntax (or signature) for Twice can be described as
  - Twice : $(D \rightarrow D) \rightarrow D \rightarrow D$
- Given the square function, sqr : $N \rightarrow N$ where N stands for the natural numbers, it follows that
- (Twice sqr) : $N \rightarrow N$
- Is twice a higher order function?
- *define* FourthPower = Twice sqr.

The mechanism that allows functions to be defined to work on a number of types of data is also known as **parametric polymorphism**

DIVISION

```
if(a>=b) then
     return 1+ (a-b)/b);
else
  return 0
```

if_then_else=$_{def}$ *λcond.λthen_do. λelse_do.* Cond (*then_do*) (*else_do*)

- *a/b*
  - *if a>=b then 1+ (a-b)/b else 0*

divide=λa. λb. if_then_else(*greaterOrEqual a b*) (succ ((zero)

divide= λa. λb. if_then_else(*greater b a*) (zero) (succ (*self* (*subtract a b*) *b*)

- divide seven three
- if_then_else(*greaterOrEqual seven three*) (succ(*self* (*subtract seven three*) *three*) (zero)
- (succ(*self* (*subtract seven three*) *three*)
- *(succ* (if_then_else(*greaterOrEqual four three*) (*succ*(*self* (*subtract four three*) *three*) (zero)))
- *(succ((succ(self (subtract four three) three)))*
- *(succ((succ(*if_then_else(*greaterOrEqual one three*) (*succ*(*self* (*subtract one three*) *three*) (zero))))
- *(succ(succ(*zero)))

LITTLE BIT OF CREATIVITY + LITTLE BIT OF ELEGANCE

Self application

- sa = λx. x x

This function takes an argument x, which is apparently a function

- Loop : (λx. x x) (λx. x x)
- Ω=(λx.(λx. x x) (λx. x x)) (λx. x x)
- The Omega Combinator is just the simplest function which infinitely recurs without calling itself.
- Y = λf. (λx. f (x x)) (λx. f (x x))

https://youtu.be/BC8ZAMwfwi4

Y COMBINATOR

Yt=t(Yt)=t(t(Yt))=...

Y combinator can be defined as

- `Y = λt. (λx. t (x x)) (λx. t (x x))`

`Yz=(λt. (λx. t (x x)) (λx. t (x x)))z`

`=(λx. z (x x)) (λx. z (x x))`

```
Yz =(λx. z (x x)) (λg. z (g g))
     = z (λg. z (g g)) (λg. z (g g))
      =z(Yz)
      =z((λg. z (g g)) (λh. z (h h)))
      =z(z((λh. z (h h)) (λh. z (h h))))
      =z(z(Yz)...
```

Y COMBINATOR

Y combinator can be defined as

- Y = λf. (λ x. f(x x)) (λx. f (x x))

Yf=f(Yf)=f(f(Yf))=…

$$\textbf{define} \quad \text{factorial} \quad = \quad \lambda n.\, \text{if } (= n\ 1)\ 1 \quad (*\ n\ (\text{factorial } (-\ n\ 1)))$$

```
T= λn. If_then_else(isZero n)one (mult n(Fact(pred n))
```

$$\begin{aligned} &\textbf{define} \quad \text{factorial} &&= \underline{T}\ \text{factorial} \\ &\textbf{define} \quad \underline{T} &&= \lambda f.\, \lambda n.\, \text{if } (= n\ 1)\ 1 \\ & && \qquad (*\ n\ (f\ (-\ n\ 1)) \end{aligned}$$

$$\begin{aligned} (\mathbf{Y}\ \underline{T})\ 1 &= \underline{T}\ (\mathbf{Y}\ \underline{T})\ 1 \\ &= \text{if } (= 1\ 1)\ 1\ (*\ 1\ (\mathbf{Y}\ \underline{T}\ (-\ 1\ 1))) && \beta\text{-reduction} \\ &= 1 && \text{calculating arithmetic} \end{aligned}$$

Fact:=Y (λ f. λ n. `If_then_else` (isZero n) one (mult n(f(pred n))

CALCULATING FACTORIAL

```
T= (λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n))
Fact=YT  Fact 2= (YT) two
=T(YT)two
= (λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n)) (YT) two
= mult two (YT(one))
=mult two (T(YT)one)
```

```
divide=λa. λb. if_then_else(greaterOrEqual a b) (succ (self (subtract a b) b) (zero)
```

DIVISION AGAIN!

```
D:= λf. λa. λb. if_then_else(greaterOrEqual a b) (succ(f (subtract a b) b) (Zero)

YD=D(YD)

YD five two

=> D(YD) five two

=> succ(YD three two)

=> succ(D(YD) three two)

=>…

=>succ(succ(YD one two))

…

=>succ(succ(zero))
```

SUMMATION

To compute sum of natural numbers from 0 to n

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i.$$

$$\mathsf{R} \equiv (\lambda rn.\mathsf{Z}n0(n\mathsf{S}(r(\mathsf{P}n))))$$

```
T= (λ f. λ n. If_then_else (isZero n) zero
Summation=YT  Summation three= (YT) three
```

- Zn0 : if n==0 then the result of the sum is 0 else the successor
- function is applied n times through the recursive call (r)

TAIL RECURSION

Tail recursion is a situation where a recursive call is the last thing a function does before returning, and the function either returns the result of the recursive call or (for a procedure) returns no result. A compiler can recognize tail recursion and replace it by a more efficient implementation.

RECURSIVE SUM

```
function recsum(x) {
    if (x === 0) {
        return 0;
    } else {
        return x + recsum(x - 1);
    }
}
```

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
tailrecsum(0, 15)
15
```

```
function tailrecsum(x, running_total = 0) {
    if (x === 0) {
        return running_total;
    } else {
        return tailrecsum(x - 1, running_total + x);
    }
}
```

- If the continuation is empty and there are no backtrack points, nothing need be placed on the stack; execution can simply jump to the called procedure, without storing any record of how to come back. This is called LAST–CALL OPTIMIZATION

- A procedure that calls itself with an empty continuation and no backtrack points is described as TAIL RECURSIVE, and last–call optimization is sometimes called TAIL–RECURSION OPTIMIZATION

FACTORIAL

```
Fact(acc,n) {
   return n==1?acc:Fact(acc*n,n-1);
}
```

```
T= (λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n))
```

```
T= (λ f. λ n. λ acc. If_then_else (isZero n) acc
                                   (f(pred n) (mult n acc))
```

INTRODUCING TYPES

- Even though the lambda calculus is untyped, a large majority of the lambda terms that we look at can be given types
- In fact, looking at the types of the terms provides insight into the kind of functions these terms represent
- So, wherever possible, we mention the types of the functions. We use capital letters A, B, . . . to represent arbitrary types and the → symbol to represent function types.
- A → B represents the type of functions from A to B, i.e., functions that given A-typed arguments, return B-typed results.
- We use a bracketing convention to parse type expressions with multiple → symbols

Simple types: $A, B ::= \iota \mid A \to B \mid A \times B \mid 1$

TYPE DEFINITION

- The base types are things like the type of integers or the type of Booleans
- The type A → B is the type of functions from A to B.
- The type A × B is the type of pairs <x, y>, where x has type A and y has type B
- The type 1 is a one-element type.
  - You can think of 1 as an abridged version of the booleans, in which there is only one boolean instead of two.
  - You can think of 1 as the “void” or “unit” type in many programming languages: the result type of a function that has no real result.

INTRODUCING TYPES

We are going to construct functions to represent typed objects

In general, an object will have a type and a value

We need to be able to:

- i) construct an object from a value and a type
- ii) select the value and type from an object
- iii) test the type of an object

We will represent an object as a type/value pair

```
def make_obj type value = λs.(s type value)
```

EXTRACTING TYPE AND/OR VALUE

```
def selectSecond=λfirst.λsecond.second
def value obj =obj selectSecond
```

```
def selectFirst=λfirst. λsecond. first
def type obj=obj selectFirst
```

we can use these functions to define a (type, value) pair and then access the type

```
def myObj ⟨type⟩ ⟨value⟩=λs.(s ⟨type⟩⟨value⟩)

type myObj=myObj selectFirst
          =λs.(s ⟨type⟩⟨value⟩) selectFirst
          =(selectFirst ⟨type⟩⟨value⟩)
          =(λfirst.λsecond.first ⟨type⟩ ⟨value⟩)
          =(λsecond.⟨type⟩) ⟨value⟩
          =⟨type⟩
```

TYPE BOOLEAN

We will represent the boolean type as one:

def bool_type = one

Constructing a boolean type involves preceding a boolean value with bool_type:

def MAKE_BOOLEAN = make_obj bool_type

which expands as:

λvalue. λs.(s bool_type value)

We can now construct the typed booleans TRUE and FALSE from the untyped versions by:

def TRUE = MAKE_BOOLEAN true

which expands as:

λs.(s bool_type true)

TYPE BOOLEAN

def FALSE = MAKE_BOOLEAN false

which expands as:

λs.(s bool_type false)

The test for a boolean type involves checking for bool_type:

def isbool = istype bool_type

This definition expands as:

λobj.(equal (type obj) bool_type)

SELF APPLICATION IN TYPED LAMBDA CALCULUS

- Even though self-application allows calculations using the laws of the lambda calculus, what it means conceptually is not at all clear
- We can see some of the problems by just trying to give a type to sa = $\lambda x.\ x\ x$.
- Suppose the argument x is of type A.
- But, since x is being applied as a function to x, the type of x should be of the form $A \rightarrow \ldots$.
- How can x be of type A as well as $A \rightarrow B \ldots$?
- Is there a type A such that $A = (A \rightarrow B)$?
- In traditional mathematics (set theory), there is no such type.
- The concept of "domains" which can be used to represent types (instead of traditional sets)
- This led to the development of an elegant theory of domains, which serves as the foundation for the mathematical meaning of programming languages.

OBJECTS IN LAMBDA CALCULUS

- Self application is used very fundamentally in implementing object-oriented programming languages. Suppose we have an object x with a method m.
- We might invoke this method by writing something like x.m(y).
- Inside the method m, there would be references to keywords like “self” or “this” which are supposed to represent the object x itself.
- One way of solving the problem is to translate the method m into a function m’ that takes two arguments: in addition to the proper argument y, the object on which the method is being invoked. So, the definition of m’ looks like:
- m' = λ self. λ y. . . . the body of m . . .

OBJECTS IN LAMBDA CALCULUS

- The object x has a collection of such functions encoding the methods.
- The method call x.m(y) is then translated as x.m’(x)(y).
- This is a form of self application.
- The function m’, which is a part of the structure x, is applied to the structure x itself.

EXPRESSIVENESS OF LAMBDA CALCULUS

The λ-calculus can express

- data types (integers, booleans, lists, trees, etc.)
- branching (using booleans)
- recursion

This is enough to encode Turing machines

Encodings can be done

But programming in pure λ-calculus is painful

- add constants (0, 1, 2, …, true, false, if-then-else, etc.)
- add types