Lambda Calculus

Data Type Encodings

Zheng Guo **2020/04/10**

Overview

- Questions about homework
- Review and exercise on data type encodings

Homework Questions

- `stack test` fail
  - piazza post for solution
- beta-reduction hint
  - use `=d>` when it is needed
- homework partners
  - contact me (zhg069@eng.ucsd.edu) if you haven’t got a partner
  - register on Canvas if you already have a partner

Boolean Encodings

- `TRUE = \x y -> x`
- `FALSE = \x y -> y`
- `ITE = \b x y -> b x y`
- define binary `XOR` function, which returns `TRUE` exactly when one of its arguments is `TRUE`
- `XOR = \b1 b2 -> b1 (NOT b2) b2`

Boolean Encodings

- `TRUE = \x y -> x`
- `FALSE = \x y -> y`
- `ITE = \b x y -> b x y`
- prove De Morgan's laws for all possible values of `b1, b2`:
  - `NOT (AND b1 b2) = OR (NOT b1) (NOT b2)`

```
eval de_morgan_left:
  NOT (AND b1 b2)
  =d> (\b -> b FALSE TRUE) (AND b1 b2)
  =b> (AND b1 b2) FALSE TRUE
  =d> (\b1 b2 -> b1 b2 FALSE) b1 b2 FALSE TRUE
  =*> b1 b2 FALSE FALSE TRUE

eval de_morgan_right:
  OR (NOT b1) (NOT b2)
  =d> (\b1 b2 -> b1 TRUE b2) (NOT b1) (NOT b2)
  =*> (NOT b1) TRUE (NOT b2)
  =d> ((\b -> b FALSE TRUE) b1) TRUE ((\b -> b FALSE TRUE) b2)
  =*> (b1 FALSE TRUE) TRUE (b2 FALSE TRUE)
  =a> b1 FALSE TRUE TRUE (b2 FALSE TRUE)
```

```
-- b1 = TRUE, b2 = TRUE
eval de_morgan_left:
  TRUE TRUE FALSE FALSE TRUE
  =*> TRUE FALSE TRUE
  =*> FALSE

eval de_morgan_right:
  TRUE FALSE TRUE TRUE (TRUE FALSE TRUE)
  =*> FALSE TRUE (TRUE FALSE TRUE)
  =*> FALSE TRUE FALSE
  =*> FALSE
```

```
-- b1 = TRUE, b2 = FALSE
eval de_morgan_left:
  TRUE FALSE FALSE FALSE TRUE
  =*> FALSE FALSE TRUE
  =*> TRUE

eval de_morgan_right:
  TRUE FALSE TRUE TRUE (FALSE FALSE TRUE)
  =*> FALSE TRUE (FALSE FALSE TRUE)
  =*> FALSE TRUE TRUE
  =*> TRUE
```

```
-- b1 = FALSE, b2 = TRUE
eval de_morgan_left:
  FALSE TRUE FALSE FALSE TRUE
  =*> FALSE FALSE TRUE
  =*> TRUE

eval de_morgan_right:
  FALSE FALSE TRUE TRUE (TRUE FALSE TRUE)
  =*> TRUE TRUE (TRUE FALSE TRUE)
  =*> TRUE TRUE FALSE
  =*> TRUE
```

```
-- b1 = FALSE, b2 = FALSE
eval de_morgan_left:
  FALSE FALSE FALSE FALSE TRUE
  =*> FALSE FALSE TRUE
  =*> TRUE

eval de_morgan_right:
  FALSE FALSE TRUE TRUE (FALSE FALSE TRUE)
  =*> TRUE TRUE (FALSE FALSE TRUE)
  =*> TRUE TRUE TRUE
  =*> TRUE
```

Pair Encodings

- `PAIR = \x y -> \b -> b x y`
- `FST = \p -> p TRUE`
- `SND = \p -> P FALSE`
- define binary `SWAP` function, which swaps two elements in the pair
- `SWAP = \p -> PAIR (SND p) (FST p)`

Number Encodings

- `ZERO = \f x -> x`
- `ONE = \f x -> f x`
- `INC = \n -> \f x -> f (n f x) / \n -> \f x -> n f (f x)`
- `ADD = \n m -> n INC m`
- `MULT = \n m -> n (ADD m) ZERO`

Number Encodings

- `ZERO = \f x -> x`
- `ONE = \f x -> f x`
- `ADD = \n m -> ??`
- `MULT = \n m -> ??`
- `INC = \n -> ??`

Number Encodings

- `ZERO = \f x -> x`
- `ONE = \f x -> f x`
- `ADD = \n m -> ??`
- `MULT = \n m -> ??`
- define `MULT` without using `ADD`?

Number Encodings

- `ZERO = \f x -> x`
- `ONE = \f x -> f x`
- `ADD = \n m -> ??`
- `MULT = \n m -> ??`
- define `MULT` without using `ADD`?
  - `\n m -> \f x -> n (m f) x`

Number Encodings

- `ZERO = \f x -> x`
- `ONE = \f x -> f x`
- `ADD = \n m -> ??`
- `MULT = \n m -> ??`
- `POWER = \n m -> ?? (define n^m)`

Number Encodings

- `ZERO = \f x -> x`
- `ONE = \f x -> f x`
- `ADD = \n m -> ??`
- `MULT = \n m -> ??`
- `POWER = \n m -> ?? (define n^m)`
  - `\n m -> m (MULT n) ONE`
  - `\n m -> m n`

```
let POWER = \n m -> m n

eval power :

  POWER TWO THREE

  =d> (\n m -> m n) TWO THREE

  =b> (\m -> m TWO) THREE

  =b> THREE TWO

  =d> (\f x -> f (f (f x))) TWO

  =*> \x -> TWO (TWO (TWO x))

  =d> \x -> TWO (TWO ((\f x -> f (f x)) x))

  =a> \f -> TWO (TWO ((\f x -> f (f x)) f))

  =b> \f -> TWO (TWO (\x -> f (f x)))

  =d> \f -> TWO ((\f x -> f (f x)) (\x -> f (f x)))

  =b> \f -> TWO (\x -> (\x -> f (f x)) ((\x -> f (f x)) x))

  =b> \f -> TWO (\x -> (\x -> f (f x)) (f (f x)))

  =b> \f -> TWO (\x -> f (f (f (f x))))

  =d> \f -> (\f x -> f (f x)) (\x -> f (f (f (f x))))

  =b> \f -> (\x -> (\x -> f (f (f (f x)))) ((\x -> f (f (f (f x)))) x))

  =b> \f -> (\x -> (\x -> f (f (f (f x)))) (f (f (f (f x)))))

  =b> \f -> (\x -> f (f (f (f (f (f (f (f x))))))))

  =a> \f x -> f (f (f (f (f (f (f (f x)))))))
```

Number Encodings

- ZERO = \f x -> x
- ONE = \f x -> f x
- ADD = \n m -> ??
- MULT = \n m -> ??
- FACT = \n -> ?? (factorial function: 1*2*…*n, using pairs)

```
def factorial(n):
   def helper(n, acc):
   if n == 0:
      return (n, acc)
   else:
      return helper(n-1, acc * n)
   _, ret = helper(n, 1)
   return ret
```

Number Encodings

- `ZERO = \f x -> x`
- `ONE = \f x -> f x`
- `ADD = \n m -> ??`
- `MULT = \n m -> ??`
- `FACT = \n -> ?? (factorial function: 1*2*…*n, using pairs)`
  - `let STEP = \p -> PAIR (INCR (FST p)) (MULT (FST p) (SND p))`
  - `let FACT = \n -> SND (n STEP (PAIR ONE ONE))`