

Lambda Calculus: Datatype encodings

4/9/2021

Agenda

Alpha-renaming

Booleans

Pairs / List

Numbers

Q & A



Alpha-renaming

- Renaming a formal argument

- $\lambda a \rightarrow a \quad =a> \quad \lambda b \rightarrow b \quad =a> \quad \lambda q \rightarrow q$

- Only rename the *free* variables

- Rename to make expressions clearer!

Poll

What is NOT a valid alpha renaming of:

$\lambda f. x \rightarrow ((\lambda f. \rightarrow f) x)$

A. $\lambda g. x \rightarrow ((\lambda f. \rightarrow f) x)$

B. $\lambda f. y \rightarrow ((\lambda f. \rightarrow f) y)$

C. $\lambda g. x \rightarrow ((\lambda f. \rightarrow g) x)$

D. $\lambda f. x \rightarrow ((\lambda g. \rightarrow g) x)$

E. No clue $\sim \lambda_(\ツ)_/\sim$

Poll

What is NOT a valid alpha renaming of:

$\lambda f. x \rightarrow ((\lambda f. \rightarrow f) x)$ *↪ All free fs*

A. $\lambda g. x \rightarrow ((\lambda f. \rightarrow f) x)$

B. $\lambda f. y \rightarrow ((\lambda f. \rightarrow f) y)$

C. $\lambda g. x \rightarrow ((\lambda f. \rightarrow g) x)$

D. $\lambda f. x \rightarrow ((\lambda g. \rightarrow g) x)$

E. No clue $\sim \lambda_(\text{ツ})_/\sim$

Working with the Lambda Calculus

- Use alpha / beta reductions to simplify as much as possible

- No more redexes

- $(\lambda x \rightarrow E1) E2$

No more redexes AKA "Normal form"

- Use Elsa ~~statements~~ / definitions

= >

Booleans

```
let TRUE  = \x y -> x      -- Returns its first argument
let FALSE = \x y -> y      -- Returns its second argument
let ITE   = \b x y -> b x y -- Applies condition to branches
                        -- (redundant, but improves readability)
```

Thinking about booleans

```
let TRUE  = \x y -> x           let NOT = \b      -> ITE b FALSE TRUE
let FALSE = \x y -> y           let AND  = \b1 b2 -> ITE b1 b2 FALSE
let ITE   = \b x y -> b x y     let OR   = \b1 b2 -> ITE b1 TRUE b2
```

```
(ITE (NOT TRUE) (TRUE) (FALSE))
```

- Expand *as necessary*, not all at once!

Stepping through AND (NOT TRUE) TRUE, live!

Pairs

Diagram illustrating the construction of a pair (a, b) and its accessors `fst` and `snd`.

The pair (a, b) is shown with arrows pointing to `fst` and `snd`. Below, the function `MKPAIR` is defined to store the structure in a function:

```
let MKPAIR = \x y -> ((\b -> ITE b x y))
```

The expression `((\b -> ITE b x y))` is highlighted and labeled as an **accessor**.

The accessors are then defined:

```
let FST = \p -> p TRUE -- call w/ TRUE, get first value
let SND = \p -> p FALSE -- call w/ FALSE, get second value
```

Poll

function!

Where are the implicit parentheses?



bool

SND (MKPAIR ITE FALSE)

A. SND ((MKPAIR ITE) FALSE)

B. SND (MKPAIR (ITE FALSE))

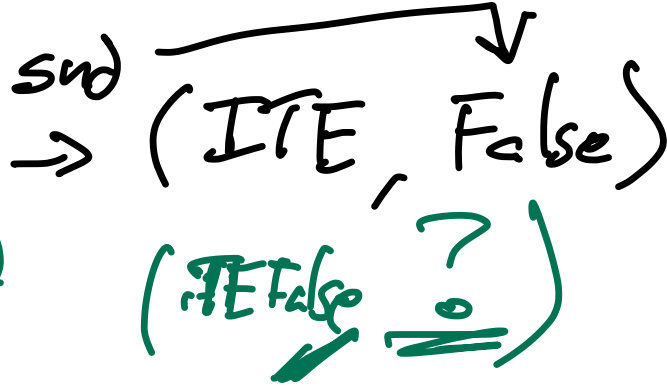
Poll

Where are the implicit parentheses?

SND (MKPAIR ITE FALSE)

A. SND ((MKPAIR ITE) FALSE)

B. SND (MKPAIR (ITE FALSE))



Application binds LEFT

Live

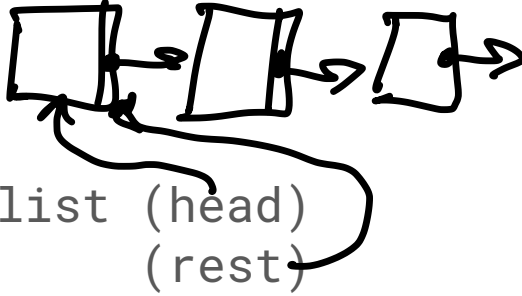
Where are the implicit parentheses?

```
SND (MKPAIR ITE) FALSE)
```

Example: List

API:

- Get head element of list (head)
- Get rest of list (rest)



Let cons = $\lambda x \text{ xs} \rightarrow (\lambda b \rightarrow \text{ITE } b \text{ x xs})$

Let nil = $\lambda b \rightarrow \text{FALSE}$

Example: List -- Poll



API:

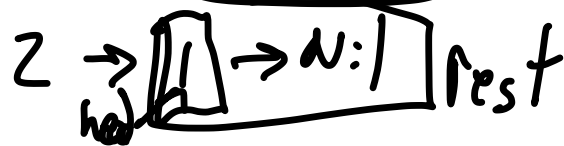
- Get head element of list (head)
- Get rest of list (rest)

Let `cons` = `\x xs -> (\b -> ITE b x xs)`

Let `nil` = `\b -> FALSE`

\uparrow
`HEAD (CONS ONE NIL) == ONE`

\downarrow
`HEAD (REST (CONS TWO (CONS ONE NIL))) == ONE`



What is HEAD?

- A. `\p -> p TRUE`
- B. `\p -> TRUE p`
- C. `\p -> p FALSE`
- D. `\p -> \f x -> f x`
- E. `????`

Example: List -- Poll

API:

- Get head element of list (head)
- Get rest of list (rest)



Let cons = \x xs -> (\b -> ITE b x xs)

Let nil = \b -> FALSE

ITE b head rest

HEAD (CONS ONE NIL) == ONE

HEAD (REST (CONS TWO (CONS ONE NIL))) == ONE

What is HEAD?

A. \p -> p TRUE

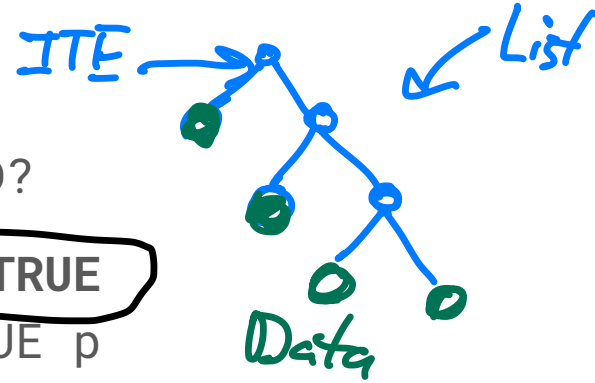
B. \p -> TRUE p

C. \p -> p FALSE

D. \p -> \f x -> f x

E. ????

head (Cons (Cons 1 Nil) Nil)



$$ITE = (\lambda b \ x y \Rightarrow b \boxed{x} y) \text{ True}$$

$$\text{True} = \lambda x y \Rightarrow x \quad (\lambda x y \Rightarrow \text{True} \ x y)$$

$$\text{False} = \lambda x y \Rightarrow y \quad (\lambda x y \Rightarrow x) \cong \text{True}$$

List Live

Numbers

- Implemented for a purpose: to count or do something X times
- Church Numerals

```
let ZERO  = \f x -> x
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> f (f (f (f (f x))))
let SIX   = \f x -> f (f (f (f (f (f x))))))
```

Using Add

let INC = \n f x -> f (n f x)

let ADD = \n m -> n INC m

let ZERO = \f x -> x

let ONE = \f x -> f x

ONE represents both:

- The idea of one (1), AND,
- iterating a function 1 time!

Add, Live!

Q & A