

Lambda Calculus: Datatype encodings

10/08/2021

Agenda

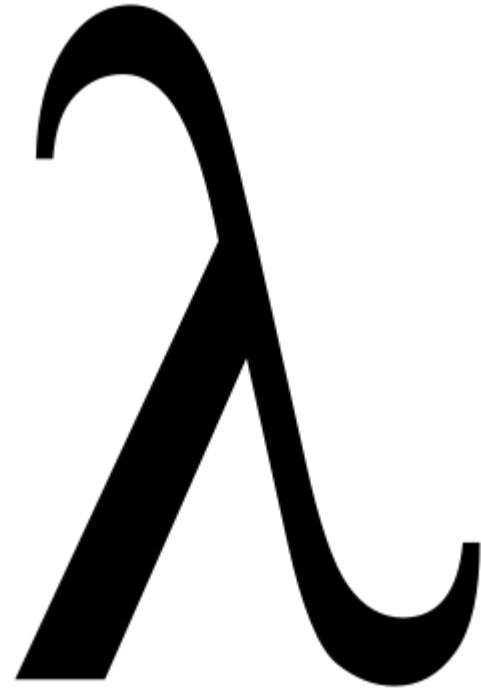
Alpha-renaming

Booleans

Pairs

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 $\text{^-}_\text{(ツ)}_\text{/}^{\text{-}}$

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 $\text{^-}_\text{(ツ)}_\text{/}^-$

Working with the Lambda Calculus

- Use alpha- / beta-reductions to simplify as much as possible:
 - $=a>$
 - $=b>$
- No more redexes
 - $(\lambda x \rightarrow E1) E2$
- Use Elsa definitions
 - $=d>$

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 FALSE = \x y → y

let ITE = \b x y → b x y

let NOT = \b → ITE b FALSE TRUE

let AND = \b1 b2 → ITE b1 b2 FALSE

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!

Stepping through **AND (NOT TRUE) TRUE**, static!

```
AND (NOT TRUE) TRUE
=d> AND ((\b → ITE b FALSE TRUE) TRUE) TRUE
=b> AND (ITE TRUE FALSE TRUE) TRUE
=d> AND ((\b x y → b x y) TRUE FALSE TRUE) TRUE
=b> AND ((\x y → TRUE x y) FALSE TRUE) TRUE
=b> AND ((\y → TRUE FALSE y) TRUE) TRUE
=b> AND (TRUE FALSE TRUE) TRUE
=d> AND ((\x y → x) FALSE TRUE) TRUE
=b> AND ((\y → FALSE) TRUE) TRUE
=b> AND FALSE TRUE
=d> (\b1 b2 → ITE b1 b2 FALSE) FALSE TRUE
=b> (\b2 → ITE FALSE b2 FALSE) TRUE
=b> ITE FALSE TRUE FALSE
=d> (\b x y → b x y) FALSE TRUE FALSE
=b> (\x y → FALSE x y) TRUE FALSE
=b> (\y → FALSE TRUE y) FALSE
=b> FALSE TRUE FALSE
=d> (\x y → y) TRUE FALSE
=b> (\y → y) FALSE
=b> FALSE
```

Pairs

```
let PAIR = \x y → (\b → ITE b x y)
```

```
let FST  = \p   → p TRUE  -- call w/ TRUE, get first value
```

```
let SND  = \p   → p FALSE -- call w/ FALSE, get second value
```

Pairs

Store structure in a function!

```
let PAIR = \x y → (\b → ITE b x y)
```

```
let FST  = \p → p TRUE  -- call w/ TRUE, get first value
```

```
let SND  = \p → p FALSE -- call w/ FALSE, get second value
```

A Note on Parentheses

Where are the implicit parentheses?

`SND (MKPAIR ITE FALSE)`

A. `SND ((MKPAIR ITE) FALSE)`

B. `SND (MKPAIR (ITE FALSE))`

A Note on Parentheses

Where are the implicit parentheses?

SND (MKPAIR ITE FALSE)

A. **SND ((MKPAIR ITE) FALSE)**

B. SND (MKPAIR (ITE FALSE))

A Note on Parentheses

instead of	we write
$\backslash x \rightarrow (\backslash y \rightarrow (\backslash z \rightarrow E))$	$\backslash x \rightarrow \backslash y \rightarrow \backslash z \rightarrow E$
$\backslash x \rightarrow \backslash y \rightarrow \backslash z \rightarrow E$	$\backslash x \ y \ z \rightarrow E$
$((E1 \ E2) \ E3) \ E4$	$E1 \ E2 \ E3 \ E4$

Stepping through `SND (MKPAIR ITE TRUE)`, live!

Stepping through `SND (MKPAIR ITE TRUE)`, static!

```
SND (PAIR ITE TRUE)
=d> SND ((\x y b → b x y) ITE TRUE)
=b> SND ((\y b → b ITE y) TRUE)
=b> SND (\b → b ITE TRUE)
=d> (\p → p FALSE) (\b → b ITE TRUE)
=b> (\b → b ITE TRUE) FALSE
=b> ((FALSE ITE) TRUE)
=d> ((\x y → y) ITE TRUE)
=b> (\y → y) TRUE
=b> TRUE
```

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)))

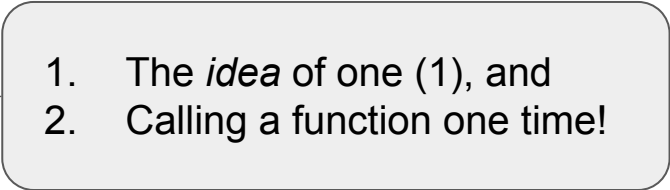
let FIVE = \f x → f (f (f (f (f x))))

let SIX = \f x → f (f (f (f (f (f x))))))

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))))))
```

- 
1. The *idea* of one (1), and
 2. Calling a function one time!

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

ADD ONE ZERO, Live!

ADD ONE ZERO, Static!

ADD ONE ZERO

=d> (\n m → n INC m) ONE ZERO

=b> (\m → ONE INC m) ZERO

=b> ONE INC ZERO

=d> (\f x → f x) INC ZERO

=b> (\x → INC x) ZERO

=b> INC ZERO

=d> (\n f x → f (n f x)) ZERO

=b> \f x → f (ZERO f x)

=d> \f x → f ((\f x → x) f x)

=a> \f x → f ((\g y → y) f x)

=b> \f x → f ((\y → y) x)

=b> \f x → f x

=d> ONE

Writing Definitions

- Factorial!
 - a. Replace the definition of **STEP** with a suitable lambda-term so that the given reductions are valid.
 - b. Replace the definition of **FACT** (factorial) with a suitable lambda-term so that the given reductions are valid.

STEP and **FACTORIAL**, Live!

STEP and FACTORIAL, Static!

-- 1st element: Index (Count)

-- 2nd element: Accumulator

```
let STEP = \p → PAIR (INC (FST p)) (MUL (FST p) (SND p))
```

-- Apply STEP n times, then get the accumulator

```
let FACT = \n → SND (n STEP (PAIR ONE ONE))
```

Q & A