Lambda Calculus

FOCS, Fall 2020

Computation as algebra

Turing machines and production grammars are fundamentally computer science models — they can be used to define computation as decision problems over strings of symbols

This week - a model of computation originally due to logicians (Church 1936) to capture computability for natural number functions $N^k \to N$

Intuition: to compute is to simplify an algebraic expression until you get a result

Expressions of the λ calculus

λ calculus expressions (also called terms) are one of the following:

- identifiers (x, y, z, …)
- abstractions <x → M> where x is an identifier and M is a term
- applications (M N) where M, N are terms

We assume that application associates to the left, so that we can drop parentheses around applications in some situations, so that M N P = (M N) P

Simplification rule

A single simplification rule captures how abstraction interacts with application

$$(<x \to M>\ N) = M\{{}^{N}/x\}$$

where $M\{{}^{N}/x\}$ is M in which every *free occurrence* of x is replaced by N

- A free occurrence of x is a use of x that is not in the body of any <x → N>
- Need to make sure free occurrences of identifiers in N are not *captured*:
    - rename non-free occurrences in M to new identifiers if needed

We are allowed to simplify anywhere in an expression

Simplification rule

A single simplification rule captures

(<x → M> N) = M{^{N}/x}

where M{^{N}/x} is M in which every *fr*

- A free occurrence of x is a use
- Need to make sure free occurr
  - rename non-free occurrences in M

We are allowed to simplify anywher

One x is free in <y → x y>

One x is free in <y → x <x → y>>

Two x's are free in

<y → <z → x <x → x x> x> y>

Two x's and one y are free in

<y → <z → x <x → x x> x>> y

No identifier is free in:

<x → <y → <z → x x y>>>

Examples

$\langle x \to x \rangle \langle y \to y \rangle = \langle y \to y \rangle$

Examples

$\langle x \to x \rangle \langle y \to y \rangle \quad = \quad \langle y \to y \rangle$

$(\langle x \to \langle y \to x\ y \rangle\rangle\ z_1)\ z_2 \quad = \langle y \to z_1\ y \rangle\ z_2$

Examples

$\langle x \rightarrow x \rangle \ \langle y \rightarrow y \rangle \quad = \quad \langle y \rightarrow y \rangle$

$(\langle x \rightarrow \langle y \rightarrow x\ y \rangle \rangle\ z_1)\ z_2 \quad = \langle y \rightarrow z_1\ y \rangle\ z_2$

$= z_1\ z_2$

Examples

$\langle x \to x\rangle\ \langle y \to y\rangle \quad = \quad \langle y \to y\rangle$

$$
\begin{aligned}
(\langle x \to \langle y \to x\ y\rangle\rangle\ z_1)\ z_2 \quad &= \langle y \to z_1\ y\rangle\ z_2 \\
&= z_1\ z_2
\end{aligned}
$$

$(\langle x \to \langle y \to x\ y\rangle\rangle\ \langle z \to z\rangle)\ \langle x \to \langle y \to x\rangle\rangle = \langle y \to \langle z \to z\rangle\ y\rangle\ \langle x \to \langle y \to x\rangle\rangle$

Examples

$$<x \to x> <y \to y> \quad = \quad <y \to y>$$

$$\begin{aligned} (<x \to <y \to x\ y>> z_1)\ z_2 &= <y \to z_1\ y>\ z_2 \\ &= z_1\ z_2 \end{aligned}$$

$$\begin{aligned} (<x \to <y \to x\ y>> <z \to z>)\ <x \to <y \to x>> &= <y \to <z \to z>\ y>\ <x \to <y \to x>> \\ &= <z \to z>\ <x \to <y \to x>> \end{aligned}$$

Examples

$\langle x \to x \rangle \; \langle y \to y \rangle \quad = \quad \langle y \to y \rangle$

$$(\langle x \to \langle y \to x\ y \rangle\rangle\ z_1)\ z_2 \quad = \langle y \to z_1\ y \rangle\ z_2$$
$$= z_1\ z_2$$

$$(\langle x \to \langle y \to x\ y \rangle\rangle\ \langle z \to z \rangle)\ \langle x \to \langle y \to x \rangle\rangle \;=\; \langle y \to \langle z \to z \rangle\ y \rangle\ \langle x \to \langle y \to x \rangle\rangle$$
$$= \langle z \to z \rangle\ \langle x \to \langle y \to x \rangle\rangle$$
$$= \langle x \to \langle y \to x \rangle\rangle$$

Examples

$\langle x \to x\rangle\ \langle y \to y\rangle \quad = \quad \langle y \to y\rangle$

$$(\langle x \to \langle y \to x\ y\rangle\rangle\ z_1)\ z_2 \quad = \langle y \to z_1\ y\rangle\ z_2$$
$$= z_1\ z_2$$

$$(\langle x \to \langle y \to x\ y\rangle\rangle\ \langle z \to z\rangle)\ \langle x \to \langle y \to x\rangle\rangle = \langle y \to \langle z \to z\rangle\ y\rangle\ \langle x \to \langle y \to x\rangle\rangle$$
$$= \langle z \to z\rangle\ \langle x \to \langle y \to x\rangle\rangle$$
$$= \langle x \to \langle y \to x\rangle\rangle$$

$$(\langle x \to \langle y \to y\ x\rangle\rangle\ \langle z \to z\rangle)\ \langle x \to \langle y \to x\rangle\rangle = \langle y \to y\ \langle z \to z\rangle\rangle\ \langle x \to \langle y \to x\rangle\rangle$$

Examples

$\langle x \to x \rangle \langle y \to y \rangle \quad = \quad \langle y \to y \rangle$

$(\langle x \to \langle y \to x\ y \rangle\rangle\ z_1)\ z_2 \quad = \langle y \to z_1\ y \rangle\ z_2$

$= z_1\ z_2$

$(\langle x \to \langle y \to x\ y \rangle\rangle\ \langle z \to z \rangle)\ \langle x \to \langle y \to x \rangle\rangle = \langle y \to \langle z \to z \rangle\ y \rangle\ \langle x \to \langle y \to x \rangle\rangle$

$= \langle z \to z \rangle\ \langle x \to \langle y \to x \rangle\rangle$

$= \langle x \to \langle y \to x \rangle\rangle$

$(\langle x \to \langle y \to y\ x \rangle\rangle\ \langle z \to z \rangle)\ \langle x \to \langle y \to x \rangle\rangle = \langle y \to y\ \langle z \to z \rangle\rangle\ \langle x \to \langle y \to x \rangle\rangle$

$= \langle x \to \langle y \to x \rangle\rangle\ \langle z \to z \rangle$

Examples

$$<x \to x> <y \to y> \quad = \quad <y \to y>$$

$$\begin{aligned}
(<x \to <y \to x\ y>>\ z_1)\ z_2 &= <y \to z_1\ y>\ z_2 \\
&= z_1\ z_2
\end{aligned}$$

$$\begin{aligned}
(<x \to <y \to x\ y>>\ <z \to z>)\ <x \to <y \to x>> &= <y \to <z \to z>\ y>\ <x \to <y \to x>> \\
&= <z \to z>\ <x \to <y \to x>> \\
&= <x \to <y \to x>>
\end{aligned}$$

$$\begin{aligned}
(<x \to <y \to y\ x>>\ <z \to z>)\ <x \to <y \to x>> &= <y \to y\ <z \to z>>\ <x \to <y \to x>> \\
&= <x \to <y \to x>>\ <z \to z> \\
&= <y \to <z \to z>>
\end{aligned}$$

Normal forms

An expression is in *normal form* if no simplification is possible

- it has no subexpression of the form <x → M> N

An expression has a normal form if you can simplify it to a normal form

You can think of an expression in normal form as a result

Existence of normal forms

Not every expression has a normal form

<x → x x> <x → x x>

Existence of normal forms

Not every expression has a normal form

<x → x x> <x → x x> = <x → x x> <x → x x>

Existence of normal forms

Not every expression has a normal form

<x → x x> <x → x x> = <x → x x> <x → x x>

= <x → x x> <x → x x>

Existence of normal forms

Not every expression has a normal form

<x → x x> <x → x x> = <x → x x> <x → x x>

= <x → x x> <x → x x>

= <x → x x> <x → x x>

Existence of normal forms

Not every expression has a normal form

<x → x x> <x → x x> = <x → x x> <x → x x>
= <x → x x> <x → x x>
= <x → x x> <x → x x>
= …

Church-Rosser Theorem: If an expression has a normal form, every simplification that yields a normal form yields the same normal form

In English: order of simplification is irrelevant, but some orders may not terminate

Abbreviations

I'm generally going to write:

$$<x\ y \to M> \quad \text{for} \quad <x \to <y \to M>>$$

$$<x\ y\ z \to M> \quad \text{for} \quad <x \to <y \to <z \to M>>>$$

...

So that for example:

$$\begin{aligned} <x\ y \to M>\ N &= <x \to <y \to M>>\ N \\ &= <y \to M\{^N/x\}> \end{aligned}$$

$$\begin{aligned} <x\ y\ z \to M>\ N &= <x \to <y \to <z \to M>>>\ N \\ &= <y \to <z \to M\{^N/x\}>> \\ &= <y\ z \to M\{^N/x\}> \end{aligned}$$

Encoding Booleans

The lambda calculus only has identifiers and functions - no other "types" of values

We can encode other types of values - if it looks, swims, and quacks like a duck…

Encoding Booleans

The lambda calculus only has identifiers and functions - no other "types" of values

We can encode other types of values - if it looks, swims, and quacks like a duck…

true = <x y → x> (= <x → <y → x>>)

false = <x y → y> (= <x → <y → y>>)

Encoding Booleans

The lambda calculus only has identifiers and functions - no other "types" of values

We can encode other types of values - if it looks, swims, and quacks like a duck…

true = <x y → x> (= <x → <y → x>>)

false = <x y → y> (= <x → <y → y>>)

if = <c t e → (c t) e> (= <c → <t → <e → (c t) e>>>)

Encoding Booleans

The lambda calculus only has identifiers and functions - no other "types" of values

We can encode other types of values - if it looks, swims, and quacks like a duck…

true = <x y → x> (= <x → <y → x>>)

false = <x y → y> (= <x → <y → y>>)

if = <c t e → (c t) e> (= <c → <t → <e → (c t) e>>>)

Properties desired:

if true z_1 z_2 = z_1

if false z_1 z_2 = z_2

Encoding Booleans

The lambda calculus only has identifiers a

We can encode other types of values - if i

true = <x y → x>   (= <x → <y → x>>

false = <x y → y>   (= <x → <y → y>>

if = <c t e → (c t) e>   (= <c → <t → <e –

Properties desired: **if true** z_1 z_2 = z_1
if false z_1 z_2 = z_2

if true z_1 z_2
= <c t e → (c t) e> **true** z_1 z_2
= <t e → (**true** t) e> z_1 z_2
= <e → (**true** z_1) e> z_2
= (**true** z_1) z_2
= (<x y → x> z_1) z_2
= <y → z_1> z_2
= z_1

Encoding Booleans

The lambda calculus only has identifiers a

We can encode other types of values - if i

true = <x y → x> (= <x → <y → x>>

false = <x y → y> (= <x → <y → y>>

if = <c t e → (c t) e> (= <c → <t → <e –

Properties desired: **if true** z_1 z_2 = z_1
if false z_1 z_2 = z_2

if false z_1 z_2
= <c t e → (c t) e> **false** z_1 z_2
= <t e → (**false** t) e> z_1 z_2
= <e → (**false** z_1) e> z_2
= (**false** z_1) z_2
= (<x y → y> z_1) z_2
= <y → y> z_2
= z_2

Encoding pairs

A pair is a data structure holding two values, from which you can extract out either

Encoding pairs

A pair is a data structure holding two values, from which you can extract out either

pair = <x y → <s → (s x) y>>

Encoding pairs

A pair is a data structure holding two values, from which you can extract out either

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Encoding pairs

A pair is a data structure holding two values, from which you can extract out either

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z_1
snd (**pair** z_1 z_2) = z_2

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z

snd (**pair** z_1 z_2) =

fst (**pair** z_1 z_2)

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) =
snd (**pair** z_1 z_2) =

fst (**pair** z_1 z_2)
= **fst** (<x y → <s → (s x) y>> z_1 z_2)

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z
snd (**pair** z_1 z_2) =

fst (**pair** z_1 z_2)
= **fst** (<x y → <s → (s x) y>> z_1 z_2)
= **fst** (<y → <s → (s z_1) y>> z_2)

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z
snd (**pair** z_1 z_2) =

fst (**pair** z_1 z_2)
= **fst** (<x y → <s → (s x) y>> z_1 z_2)
= **fst** (<y → <s → (s z_1) y>> z_2)
= **fst** <s → (s z_1) z_2>

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \rightarrow <s \rightarrow (s\ x)\ y>>$

fst $= <p \rightarrow p\ <x\ y \rightarrow x>>$

snd $= <p \rightarrow p\ <x\ y \rightarrow y>>$

Properties desired: **fst** (**pair** $z_1\ z_2$) = z

snd (**pair** $z_1\ z_2$) =

fst (**pair** $z_1\ z_2$)

= **fst** $(<x\ y \rightarrow <s \rightarrow (s\ x)\ y>>\ z_1\ z_2)$

= **fst** $(<y \rightarrow <s \rightarrow (s\ z_1)\ y>>\ z_2)$

= **fst** $<s \rightarrow (s\ z_1)\ z_2>$

= $<p \rightarrow p\ <x\ y \rightarrow x>>\ <s \rightarrow (s\ z_1)\ z_2>$

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \to <s \to (s\ x)\ y>>$

fst $= <p \to p\ <x\ y \to x>>$

snd $= <p \to p\ <x\ y \to y>>$

Properties desired: **fst** (**pair** $z_1\ z_2$) = z

snd (**pair** $z_1\ z_2$) =

fst (**pair** $z_1\ z_2$)

= **fst** $(<x\ y \to <s \to (s\ x)\ y>>\ z_1\ z_2)$

= **fst** $(<y \to <s \to (s\ z_1)\ y>>\ z_2)$

= **fst** $<s \to (s\ z_1)\ z_2>$

$= <p \to p\ <x\ y \to x>>\ <s \to (s\ z_1)\ z_2>$

$= <s \to (s\ z_1)\ z_2>\ <x\ y \to x>$

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \to <s \to (s\ x)\ y>>$

fst $= <p \to p\ <x\ y \to x>>$

snd $= <p \to p\ <x\ y \to y>>$

Properties desired: **fst** (**pair** $z_1\ z_2$) = z

snd (**pair** $z_1\ z_2$) =

fst (**pair** $z_1\ z_2$)

$= \mathbf{fst}\ (<x\ y \to <s \to (s\ x)\ y>>\ z_1\ z_2)$

$= \mathbf{fst}\ (<y \to <s \to (s\ z_1)\ y>>\ z_2)$

$= \mathbf{fst}\ <s \to (s\ z_1)\ z_2>$

$= <p \to p\ <x\ y \to x>>\ <s \to (s\ z_1)\ z_2>$

$= <s \to (s\ z_1)\ z_2>\ <x\ y \to x>$

$= (<x\ y \to x>\ z_1)\ z_2$

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z
snd (**pair** z_1 z_2) =

fst (**pair** z_1 z_2)
= **fst** (<x y → <s → (s x) y>> z_1 z_2)
= **fst** (<y → <s → (s z_1) y>> z_2)
= **fst** <s → (s z_1) z_2>
= <p → p <x y → x>> <s → (s z_1) z_2>
= <s → (s z_1) z_2> <x y → x>
= (<x y → x> z_1) z_2
= <y → z_1> z_2

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \rightarrow <s \rightarrow (s\ x)\ y>>$

fst $= <p \rightarrow p\ <x\ y \rightarrow x>>$

snd $= <p \rightarrow p\ <x\ y \rightarrow y>>$

Properties desired: **fst** (**pair** $z_1\ z_2$) = z
snd (**pair** $z_1\ z_2$) =

fst (**pair** $z_1\ z_2$)
$= \textbf{fst}\ (<x\ y \rightarrow <s \rightarrow (s\ x)\ y>>\ z_1\ z_2)$
$= \textbf{fst}\ (<y \rightarrow <s \rightarrow (s\ z_1)\ y>>\ z_2)$
$= \textbf{fst}\ <s \rightarrow (s\ z_1)\ z_2>$
$= <p \rightarrow p\ <x\ y \rightarrow x>>\ <s \rightarrow (s\ z_1)\ z_2>$
$= <s \rightarrow (s\ z_1)\ z_2>\ <x\ y \rightarrow x>$
$= (<x\ y \rightarrow x>\ z_1)\ z_2$
$= <y \rightarrow z_1>\ z_2$
$= z_1$

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z
snd (**pair** z_1 z_2) =

snd (**pair** z_1 z_2)

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \to <s \to (s\ x)\ y>>$

fst $= <p \to p <x\ y \to x>>$

snd $= <p \to p <x\ y \to y>>$

Properties desired: **fst** (**pair** $z_1\ z_2$) =
snd (**pair** $z_1\ z_2$) =

snd (**pair** $z_1\ z_2$)
= **snd** $(<x\ y \to <s \to (s\ x)\ y>>\ z_1\ z_2)$

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z

snd (**pair** z_1 z_2) =

snd (**pair** z_1 z_2)

= **snd** (<x y → <s → (s x) y>> z_1 z_2)

= **snd** (<y → <s → (s z_1) y>> z_2)

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \to <s \to (s\ x)\ y>>$

fst $= <p \to p <x\ y \to x>>$

snd $= <p \to p <x\ y \to y>>$

Properties desired: **fst** (**pair** z_1 z_2) = z

snd (**pair** z_1 z_2) =

snd (**pair** z_1 z_2)

= **snd** $(<x\ y \to <s \to (s\ x)\ y>>\ z_1\ z_2)$

= **snd** $(<y \to <s \to (s\ z_1)\ y>>\ z_2)$

= **snd** $<s \to (s\ z_1)\ z_2>$

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) =

snd (**pair** z_1 z_2) =

snd (**pair** z_1 z_2)

= **snd** (<x y → <s → (s x) y>> z_1 z_2)

= **snd** (<y → <s → (s z_1) y>> z_2)

= **snd** <s → (s z_1) z_2>

= <p → p <x y → y>> <s → (s z_1) z_2>

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \to <s \to (s\ x)\ y>>$

fst $= <p \to p\ <x\ y \to x>>$

snd $= <p \to p\ <x\ y \to y>>$

Properties desired: **fst** (**pair** $z_1\ z_2$) $= z$

snd (**pair** $z_1\ z_2$) $=$

snd (**pair** $z_1\ z_2$)

$= \textbf{snd}\ (<x\ y \to <s \to (s\ x)\ y>>\ z_1\ z_2)$

$= \textbf{snd}\ (<y \to <s \to (s\ z_1)\ y>>\ z_2)$

$= \textbf{snd}\ <s \to (s\ z_1)\ z_2>$

$= <p \to p\ <x\ y \to y>>\ <s \to (s\ z_1)\ z_2>$

$= <s \to (s\ z_1)\ z_2>\ <x\ y \to y>$

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \to <s \to (s\ x)\ y>>$

fst $= <p \to p\ <x\ y \to x>>$

snd $= <p \to p\ <x\ y \to y>>$

Properties desired: **fst** (**pair** $z_1\ z_2) = z$

snd (**pair** $z_1\ z_2) =$

snd (**pair** $z_1\ z_2$)

$= \textbf{snd}\ (<x\ y \to <s \to (s\ x)\ y>>\ z_1\ z_2)$

$= \textbf{snd}\ (<y \to <s \to (s\ z_1)\ y>>\ z_2)$

$= \textbf{snd}\ <s \to (s\ z_1)\ z_2>$

$= <p \to p\ <x\ y \to y>>\ <s \to (s\ z_1)\ z_2>$

$= <s \to (s\ z_1)\ z_2>\ <x\ y \to y>$

$= (<x\ y \to y>\ z_1)\ z_2$

Encoding pairs

A pair is a data structure holding two val

pair $= <x\ y \rightarrow <s \rightarrow (s\ x)\ y>>$

fst $= <p \rightarrow p\ <x\ y \rightarrow x>>$

snd $= <p \rightarrow p\ <x\ y \rightarrow y>>$

Properties desired: **fst** (**pair** $z_1\ z_2) =$

snd (**pair** $z_1\ z_2) =$

snd (**pair** $z_1\ z_2$)

$= \textbf{snd}\ (<x\ y \rightarrow <s \rightarrow (s\ x)\ y>>\ z_1\ z_2)$

$= \textbf{snd}\ (<y \rightarrow <s \rightarrow (s\ z_1)\ y>>\ z_2)$

$= \textbf{snd}\ <s \rightarrow (s\ z_1)\ z_2>$

$= <p \rightarrow p\ <x\ y \rightarrow y>>\ <s \rightarrow (s\ z_1)\ z_2>$

$= <s \rightarrow (s\ z_1)\ z_2>\ <x\ y \rightarrow y>$

$= (<x\ y \rightarrow y>\ z_1)\ z_2$

$= <y \rightarrow y>\ z_2$

Encoding pairs

A pair is a data structure holding two val

pair = <x y → <s → (s x) y>>

fst = <p → p <x y → x>>

snd = <p → p <x y → y>>

Properties desired: **fst** (**pair** z_1 z_2) = z

snd (**pair** z_1 z_2) =

snd (**pair** z_1 z_2)

= **snd** (<x y → <s → (s x) y>> z_1 z_2)

= **snd** (<y → <s → (s z_1) y>> z_2)

= **snd** <s → (s z_1) z_2>

= <p → p <x y → y>> <s → (s z_1) z_2>

= <s → (s z_1) z_2> <x y → y>

= (<x y → y> z_1) z_2

= <y → y> z_2

= z_2

Encoding natural numbers

Church numerals : the encoding of *n* takes a function and applies it *n* times

Encoding natural numbers

Church numerals : the encoding of *n* takes a function and applies it *n* times

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))> (*n* times)

Encoding natural numbers

Church numerals : the encoding of *n* takes a function and applies it *n* times

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))> (*n* times)

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **true**>

Encoding natural num

succ 2

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

succ 2
= <n → <f x → (n f) (f x)>> **2**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

succ 2
= <n → <f x → (n f) (f x)>> **2**
= <f x → (**2** f) (f x)>

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

succ 2

= <n → <f x → (n f) (f x)>> **2**

= <f x → (**2** f) (f x)>

= <f x → (<f x → f (f x)> f) (f x)>

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

succ 2

= <n → <f x → (n f) (f x)>> **2**

= <f x → (**2** f) (f x)>

= <f x → (<f x → f (f x)> f) (f x)>

= <f x → <x → f (f x)> (f x)>

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

succ 2

= <n → <f x → (n f) (f x)>> **2**

= <f x → (**2** f) (f x)>

= <f x → (<f x → f (f x)> f) (f x)>

= <f x → <x → f (f x)> (f x)>

= <f x → f (f (f x))>

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

succ 2
= <n → <f x → (n f) (f x)>> **2**
= <f x → (**2** f) (f x)>
= <f x → (<f x → f (f x)> f) (f x)>
= <f x → <x → f (f x)> (f x)>
= <f x → f (f (f x))>
= **3**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 0

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 0
= <n → (n <x → **false**>) **true**> **0**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 0

= <n → (n <x → **false**>) **true**> **0**

= (**0** <x → **false**>) **true**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 0

= <n → (n <x → **false**>) **true**> **0**

= (**0** <x → **false**>) **true**

= (<f x → x> <x → **false**>) **true**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 0

= <n → (n <x → **false**>) **true**> **0**

= (**0** <x → **false**>) **true**

= (<f x → x> <x → **false**>) **true**

= <x → x> **true**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 0
= <n → (n <x → **false**>) **true**> **0**
= (**0** <x → **false**>) **true**
= (<f x → x> <x → **false**>) **true**
= <x → x> **true**
= **true**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 2

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 2
= <n → (n <x → **false**>) **true**> **2**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 2
= <n → (n <x → **false**>) **true**> **2**
= (**2** <x → **false**>) **true**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 2

= <n → (n <x → **false**>) **true**> **2**

= (**2** <x → **false**>) **true**

= (<f x → f (f x)> <x → **false**>) **true**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 2

= <n → (n <x → **false**>) **true**> **2**

= (**2** <x → **false**>) **true**

= (<f x → f (f x)> <x → **false**>) **true**

= <x → <x → **false**> (<x → **false**> x)> **true**

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>

1 = <f x → f x>

2 = <f x → f (f x)>

3 = <f x → f (f (f x))>

…

n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 2

= <n → (n <x → **false**>) **true**> **2**

= (**2** <x → **false**>) **true**

= (<f x → f (f x)> <x → **false**>) **true**

= <x → <x → **false**> (<x → **false**> x)> **true**

= <x → **false**> (<x → **false**> **true**)

Encoding natural num

Church numerals : the encoding o

0 = <f x → x>
1 = <f x → f x>
2 = <f x → f (f x)>
3 = <f x → f (f (f x))>
…
n = <f x → f (f (... f (f x) …))>

succ = <n → <f x → (n f) (f x)>>

zero? = <n → (n <x → **false**>) **tru**

zero? 2
= <n → (n <x → **false**>) **true**> **2**
= (**2** <x → **false**>) **true**
= (<f x → f (f x)> <x → **false**>) **true**
= <x → <x → **false**> (<x → **false**> x)> **true**
= <x → **false**> (<x → **false**> **true**)
= **false**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**>

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

= <m n → (m **succ**) n> **2 3**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

= <m n → (m **succ**) n> **2 3**

= <n → (**2 succ**) n> **3**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

= <m n → (m **succ**) n> **2 3**

= <n → (**2 succ**) n> **3**

= (**2 succ**) **3**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

= <m n → (m **succ**) n> **2 3**

= <n → (**2 succ**) n> **3**

= (**2 succ**) **3**

= (<f x → f (f x)> **succ**) **3**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

= <m n → (m **succ**) n> **2 3**

= <n → (**2 succ**) n> **3**

= (**2 succ**) **3**

= (<f x → f (f x)> **succ**) **3**

= <x → **succ** (**succ** x)> **3**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

= <m n → (m **succ**) n> **2 3**

= <n → (**2 succ**) n> **3**

= (**2 succ**) **3**

= (<f x → f (f x)> **succ**) **3**

= <x → **succ** (**succ** x)> **3**

= **succ** (**succ 3**)

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

plus 2 3

= <m n → (m **succ**) n> **2 3**

= <n → (**2 succ**) n> **3**

= (**2 succ**) **3**

= (<f x → f (f x)> **succ**) **3**

= <x → **succ** (**succ** x)> **3**

= **succ** (**succ 3**)

= …

= **5**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

times 2 3

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

= <n → (**2** (**plus** n)) **0**> **3**

Arithmetic

plus = <m n → (m **succ)** n>

times = <m n → (m (**plus** n)) **0**

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

= <n → (**2** (**plus** n)) **0**> **3**

= (**2** (**plus 3**)) **0**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**>

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

= <n → (**2** (**plus** n)) **0**> **3**

= (**2** (**plus 3**)) **0**

= (<f x → f (f x)> (**plus 3**)) **0**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

= <n → (**2** (**plus** n)) **0**> **3**

= (**2** (**plus 3**)) **0**

= (<f x → f (f x)> (**plus 3**)) **0**

= <x → (**plus 3**) ((**plus 3**) x)> **0**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

= <n → (**2** (**plus** n)) **0**> **3**

= (**2** (**plus 3**)) **0**

= (<f x → f (f x)> (**plus 3**)) **0**

= <x → (**plus 3**) ((**plus 3**) x)> **0**

= (**plus 3**) ((**plus 3**) **0**)

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

= <n → (**2** (**plus** n)) **0**> **3**

= (**2** (**plus 3**)) **0**

= (<f x → f (f x)> (**plus 3**)) **0**

= <x → (**plus 3**) ((**plus 3**) x)> **0**

= (**plus 3**) ((**plus 3**) **0**)

= **plus 3** (**plus 3 0**)

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**

times 2 3

= <m n → (m (**plus** n)) **0**> **2 3**

= <n → (**2** (**plus** n)) **0**> **3**

= (**2** (**plus 3**)) **0**

= (<f x → f (f x)> (**plus 3**)) **0**

= <x → (**plus 3**) ((**plus 3**) x)> **0**

= (**plus 3**) ((**plus 3**) **0**)

= **plus 3** (**plus 3 0**)

= …

= **6**

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**>

pred = <n → ??? >

Arithmetic

plus = <m n → (m **succ**) n>

times = <m n → (m (**plus** n)) **0**>

pred = <n → **snd** (n **F** (**pair 0 0**))>
where **F** = <p → **pair** (**succ** (**fst** p)) (**fst** p)>

Key property: **F** (**pair** *i j*) = **pair** (*i* + *1*, *i*)

So *n* **F** (**pair 0 0**) = **F** (**F** (... **F** (**pair 0 0**) …)) = **pair** (*n*, *n* - *1*)
and **snd** (*n* **F** (**pair 0 0**)) = *n* - *1*

Recursion

What about computing the sum of all numbers between 0 and *n* ?

Tempting:

sumto = <n → **if** (**zero?** n) **0** (**plus** n (**sumto** (**pred** n)))>

Recursion

What about computing the sum of all numbers between 0 and *n* ?

Tempting:

> **sumto** = <n → **if** (**zero?** n) **0** (**plus** n (**sumto** (**pred** n)))>

That's not a mathematical definition!
The right-hand side of a definition cannot contain the term you're defining

But we've written stuff like that before, in OCaml.

So what gives? If it's not a valid definition, what is it?

Arithmetical example

If I write:

$$x = 3x + 1$$

how do you read this in mathematics?

Arithmetical example

If I write:

$$x = 3x + 1$$

how do you read this in mathematics?

It's an equation. It tells you that whatever x is, it has the property that it's equal to one more than its own tripled value

How can you find an x with that property? You solve the equation:

$$x = 3x + 1 \quad \Rightarrow \quad x = -1/2$$

Lambda calculus equations

Similarly, we're going to interpret

sumto = <n → **if** (**zero?** n) **0** (**plus** n (**sumto** (**pred** n)))>

as an **equation**

g = <n → **if** (**zero?** n) **0** (**plus** n (g (**pred** n)))>

We're looking for a term in the lambda calculus that satisfies the above equation

Solving equations = finding fixed points!

Consider again the arithmetic equation

$$x = 3x + 1$$

Define function F as:

$$F(x) = 3x + 1$$

Every solution x_0 of $x = 3x + 1$ has the property that $F(x_0) = x_0$

Conversely, if $F(x_0) = x_0$, then x_0 is a solution of $x = 3x + 1$

Solutions of $x = 3x + 1$ are *fixed points* of the function $F(x) = 3x + 1$

Solving lambda calculus equations

Back to

g = <n → **if** (**zero?** n) **0** (**plus** n (g (**pred** n)))>

Solving this equation means finding a fixed point of the function:

F_{sumto}(g) = <n → **if** (**zero?** n) **0** (**plus** n (g (**pred** n)))>

or equivalently:

F_{sumto} = <g → <n → **if** (**zero?** n) **0** (**plus** n (g (**pred** n)))>>

Theorem: Every function in the lambda calculus has a fixed point

Fixed point combinators

Define

$\Theta \quad = \langle y \to y\ ((\Theta_0\ \Theta_0)\ y)\rangle$ where $\Theta_0 = \langle x\ y \to y\ ((x\ x)\ y)\rangle$

$\Theta\ F = ?$

Fixed point combinators

Define

$\Theta \quad = \langle y \to y\ ((\Theta_0\ \Theta_0)\ y)\rangle \qquad$ where $\Theta_0 = \langle x\ y \to y\ ((x\ x)\ y)\rangle$

$\Theta\ F = \langle y \to y\ ((\Theta_0\ \Theta_0)\ y\rangle\ F$

Fixed point combinators

Define

$$\Theta \quad = <y \to y\ ((\Theta_0\ \Theta_0)\ y)> \qquad \text{where}\ \ \Theta_0 = <x\ y \to y\ ((x\ x)\ y)>$$

$$\Theta\ F = <y \to y\ ((\Theta_0\ \Theta_0)\ y>\ F$$

$$= F\ ((\Theta_0\ \Theta_0)\ F)$$

Fixed point combinators

Define

$$\Theta \quad = \langle y \rightarrow y\ ((\Theta_0\ \Theta_0)\ y)\rangle \qquad \text{where} \quad \Theta_0 = \langle x\ y \rightarrow y\ ((x\ x)\ y)\rangle$$

$$\begin{aligned} \Theta\ F &= \langle y \rightarrow y\ ((\Theta_0\ \Theta_0)\ y\rangle\ F \\ &= F\ ((\Theta_0\ \Theta_0)\ F) \\ &= F\ ((\langle x\ y \rightarrow y\ ((x\ x)\ y)\rangle\ \Theta_0)\ F) \end{aligned}$$

Fixed point combinators

Define

$\Theta \quad = \langle y \to y\ ((\Theta_0\ \Theta_0)\ y)\rangle \qquad \text{where} \quad \Theta_0 = \langle x\ y \to y\ ((x\ x)\ y)\rangle$

$$\begin{aligned}\Theta\ F &= \langle y \to y\ ((\Theta_0\ \Theta_0)\ y\rangle\ F \\ &= F\ ((\Theta_0\ \Theta_0)\ F) \\ &= F\ ((\langle x\ y \to y\ ((x\ x)\ y)\rangle\ \Theta_0)\ F) \\ &= F\ ((\langle y \to y\ ((\Theta_0\ \Theta_0)\ y)\ F)\end{aligned}$$

Fixed point combinators

Define

$\Theta \quad = <y \to y\ ((\Theta_0\ \Theta_0)\ y)> \qquad$ where $\ \Theta_0 = <x\ y \to y\ ((x\ x)\ y)>$

$$
\begin{aligned}
\Theta\ F &= <y \to y\ ((\Theta_0\ \Theta_0)\ y>\ F \\
&= F\ ((\Theta_0\ \Theta_0)\ F) \\
&= F\ ((<x\ y \to y\ ((x\ x)\ y)>\ \Theta_0)\ F) \\
&= F\ ((<y \to y\ ((\Theta_0\ \Theta_0)\ y)\ F) \\
&= F\ (\Theta\ F)
\end{aligned}
$$

Fixed point combinators

Define

$$\Theta \quad = <y \to y\ ((\Theta_0\ \Theta_0)\ y)> \qquad \text{where}\ \ \Theta_0 = <x\ y \to y\ ((x\ x)\ y)>$$

$$
\begin{aligned}
\Theta\ F &= <y \to y\ ((\Theta_0\ \Theta_0)\ y>\ F \\
&= F\ ((\Theta_0\ \Theta_0)\ F) \\
&= F\ ((<x\ y \to y\ ((x\ x)\ y)>\ \Theta_0)\ F) \\
&= F\ ((<y \to y\ ((\Theta_0\ \Theta_0)\ y)\ F) \\
&= F\ (\Theta\ F)
\end{aligned}
$$

$\Rightarrow$ $\Theta\ F$ is a fixed point of F!

Fixed point combinators

Define

$$\Theta \quad = <y \to y\ ((\Theta_0\ \Theta_0)\ y)>$$

wher

$$\begin{aligned}
\Theta\ F &= <y \to y\ ((\Theta_0\ \Theta_0)\ y>\ F \\
&= F\ ((\Theta_0\ \Theta_0)\ F) \\
&= F\ ((<x\ y \to y\ ((x\ x)\ y)>\ \Theta_0)\ F) \\
&= F\ ((<y \to y\ ((\Theta_0\ \Theta_0)\ y)\ F) \\
&= F\ (\Theta\ F)
\end{aligned}$$

$\Rightarrow$ $\Theta\ F$ is a fixed point of F!

Another famous fixed point combinator you might have heard of:

$$Y = <f \to (<x \to f\ (x\ x)>\ <x \to f\ (x\ x)>)>$$

and you can derive:

$$Y\ F = F\ (Y\ F)$$

Defining **sumto**

Given:

F_{sumto} = <g → <n → **if** (**zero?** n) **0** (**plus** n (g (**pred** n)))>>

You can define:

sumto = Θ F_{sumto}

= Θ <g → <n → **if** (**zero?** n) **0** (**plus** n (g (**pred** n)))>>

Claim: **sumto** *n* simplifies to the sum of all first *n* natural numbers

$$
\begin{aligned}
\textbf{sumto 3} &= (\Theta\ F_{\text{sumto}})\ \mathbf{3} \\
&= \ldots = (F_{\text{sumto}}\ (\Theta\ F_{\text{sumto}}))\ \mathbf{3} \\
&= (<h \rightarrow <n \rightarrow \textbf{if}\ (\textbf{zero?}\ n)\ \mathbf{0}\ (\textbf{plus}\ n\ (h\ (\textbf{pred}\ n)))>>\ (\Theta\ F_{\text{sumto}}))\ \mathbf{3} \\
&= <n \rightarrow \textbf{if}\ (\textbf{zero?}\ n)\ \mathbf{0}\ (\textbf{plus}\ n\ ((\Theta\ F_{\text{sumto}})\ (\textbf{pred}\ n)))>\ \mathbf{3} \\
&= \textbf{if}\ (\textbf{zero?}\ 3)\ \mathbf{0}\ (\textbf{plus 3}\ ((\Theta\ F_{\text{sumto}})\ (\textbf{pred 3}))) \\
&= \ldots = \textbf{plus 3}\ ((\Theta\ F_{\text{sumto}})\ (\textbf{pred 3})) \\
&= \ldots = \textbf{plus 3}\ ((\Theta\ F_{\text{sumto}})\ \mathbf{2}) \\
&= \ldots = \textbf{plus 3}\ ((F_{\text{sumto}}\ (\Theta\ F_{\text{sumto}}))\ \mathbf{2}) \\
&= \textbf{plus 3}\ ((<h \rightarrow <n \rightarrow \textbf{if}\ (\textbf{zero?}\ n)\ \mathbf{0}\ (\textbf{plus}\ n\ (h\ (\textbf{pred}\ n)))>>\ (\Theta\ F_{\text{sumto}}))\ \mathbf{2}) \\
&= \textbf{plus 3}\ (<n \rightarrow \textbf{if}\ (\textbf{zero?}\ n)\ \mathbf{0}\ (\textbf{plus}\ n\ ((\Theta\ F_{\text{sumto}})\ (\textbf{pred}\ n)))>\ \mathbf{2}) \\
&= \textbf{plus 3}\ (\textbf{if}\ (\textbf{zero? 2})\ \mathbf{0}\ (\textbf{plus 2}\ ((\Theta\ F_{\text{sumto}})\ (\textbf{pred 2})))) \\
&= \ldots = \textbf{plus 3}\ (\textbf{plus 2}\ ((\Theta\ F_{\text{sumto}})\ (\textbf{pred 2}))) \\
&= \ldots = \textbf{plus 3}\ (\textbf{plus 2}\ ((\Theta\ F_{\text{sumto}})\ \mathbf{1}))
\end{aligned}
$$

...

…

$= \text{plus}\ 3\ (\text{plus}\ 2\ ((\Theta\ F_{sumto})\ 1))$

$= \ldots = \text{plus}\ 3\ (\text{plus}\ 2\ ((F_{sumto}\ (\Theta\ F_{sumto}))\ 1))$

$= \text{plus}\ 3\ (\text{plus}\ 2\ ((<h \to <n \to \text{if}\ (\text{zero?}\ n)\ 0\ (\text{plus}\ n\ (h\ (\text{pred}\ n)))>>\ (\Theta\ F_{sumto}))\ 1))$

$= \text{plus}\ 3\ (\text{plus}\ 2\ (<n \to \text{if}\ (\text{zero?}\ n)\ 0\ (\text{plus}\ n\ ((\Theta\ F_{sumto})\ (\text{pred}\ n)))>\ 1))$

$= \text{plus}\ 3\ (\text{plus}\ 2\ (\text{if}\ (\text{zero?}\ 1)\ 0\ (\text{plus}\ 1\ ((\Theta\ F_{sumto})\ (\text{pred}\ 1)))))$

$= \ldots = \text{plus}\ 3\ (\text{plus}\ 2\ (\text{plus}\ 1\ ((\Theta\ F_{sumto})\ (\text{pred}\ 1))))$

$= \ldots = \text{plus}\ 3\ (\text{plus}\ 2\ (\text{plus}\ 1\ ((\Theta\ F_{sumto})\ 0)))$

$= \ldots = \text{plus}\ 3\ (\text{plus}\ 2\ (\text{plus}\ 1\ ((F_{sumto}\ (\Theta\ F_{sumto}))\ 0)))$

$= \text{plus}\ 3\ (\text{plus}\ 2\ (\text{plus}\ 1\ ((<h \to <n \to \text{if}\ (\text{zero?}\ n)\ 0\ (\text{plus}\ n\ (h\ (\text{pred}\ n)))>>\ (\Theta\ F_{sumto}))\ 0)))$

$= \text{plus}\ 3\ (\text{plus}\ 2\ (\text{plus}\ 1\ (<n \to \text{if}\ (\text{zero?}\ n)\ 0\ (\text{plus}\ n\ ((\Theta\ F_{sumto})\ (\text{pred}\ n)))>\ 0)))$

$= \text{plus}\ 3\ (\text{plus}\ 2\ (\text{plus}\ 1\ (\text{if}\ (\text{zero?}\ 0)\ 0\ (\text{plus}\ 0\ ((\Theta\ F_{sumto})\ (\text{pred}\ 0))))))$

$= \ldots = \text{plus}\ 3\ (\text{plus}\ 2\ (\text{plus}\ 1\ 0)) = \ldots = 6$