Functional Programming
Question:

Consider the set of functions

\[ f : \mathbb{N} \rightarrow \mathbb{N} \cdot \]

Are all these function computable?

Answer: no!

Why? because the set of functions

\[ S = \{ f : \mathbb{N} \rightarrow \mathbb{N} \} \]

is uncountable. (and infinite)

What this means is that there is no bijection between \( S \) and \( \mathbb{N} \).

Example of countable sets:

- the set of even numbers
- the set \( \mathbb{Q} \) of rational numbers
example of non countable sets: the set \( \mathbb{R} \) of reals.

Thm by Cantor (inventor of set theory)
suppose given a set \( A \)
there is no bijection
between the set \( A \)
and the set \( \mathcal{P}(A) \) of subsets of \( A \).

Proof: by a diagonal argument.

(please have a look at it)

related to the Russell paradox

Logicomix
Suppose given a function

\[ A \xrightarrow{f} \mathcal{P}(A) \]

\underline{Claim:} \( f \) is not surjective.

[subjective \( f: A \to B \) means that for all \( b \in B \), there exists \( a \in A \) such that \( f(a) = b \)]

\[ A \xrightarrow{f} \mathcal{P}(A) \]

we want to find \( B \in \mathcal{P}(A) \)

such that there is no \( a \in A \)

such that \( f(a) = B \).
We define $B$ as

$$B = \{ a \in A \mid a \notin f(a) \}$$

Every property such as $a \notin f(a)$ defines a subset
What we have:

\[ \forall a \in A, \ a \in B \iff a \notin f(a) \]

by definition of B.

Hence, suppose that f is surjective.

in that case,

there exists a \( a \in A \)

such that \( f(a) = B \).

We may ask ourselves:

is a an element of B?

\[ a \in B \iff a \notin B \]

this means that there is

a contradiction. hence f is not surjective.
Example: the empty set $\emptyset$

$$P(\emptyset) = \{\emptyset\} \quad \text{one element}$$

$$P(\{\emptyset\}) = \{\emptyset, \{\emptyset\}\} \quad \text{two elements}$$

If $A$ contains $n$ elements,
then $P(A)$ contains $2^n$ elements.

In particular, by Cantor's theorem,
$\mathbb{N}$ is not in bijection with $P(\mathbb{N})$.

$$\forall U \in P(A), \quad \exists \mu \in \{\text{true, false}\}$$

$$\mu(a) = \text{true when } a \in U$$

$$\mu(a) = \text{false when } a \notin U$$

the characteristic function of $U$.

$$P(A) \cong \text{functions from } A \text{ to } \Omega$$
clearly there are more functions
\[ N \rightarrow N \]
as there are functions
\[ N \rightarrow \Omega \]
(in fact there is a bijection between the two sets)

\underline{Computability}: imagine that we construct
\underline{a machine} which can compute any function \( f : N \rightarrow N \).
then we would be able to count these functions.

Hence, since the set of functions \( S \)
is not countable, \( \text{(some of these functions are not computable).} \)
In the 1930's, the Church-Turing hypothesis states that a function $f: \mathbb{N} \rightarrow \mathbb{N}$ is computable if and only if it is computable by a Turing machine.

This is arguably the simplest of the three calculi which can be expressed using a term of the $\lambda$-calculus.
What is the λ-calculus?

It is a “pure” calculus of functions.

What is important in a function?

1. We can apply a function \( f \) to an argument \( x \) in order to obtain a value

\[ f(x) \]

I will write this (very formally)

\[ \text{App}(f, x) \]

application node function

the argument

argument

function
1. We should have variables $x, y, z$

so we suppose given a countable set (infinite)

Var of variables.

2. We should have the ability to construct functions by picking a variable $x$.

in a term of the calculus

Informally: $x \mapsto \sin(x)$

function term
We thus have an operator
\[ \lambda x \]
which plays the role of
\[ x \mapsto x \]

Example:

1. \( \lambda x . x \)

is the identity function
which behaves as follows:

\[ \text{App}(\lambda x . x, P) = P \]

any term of the \( \lambda \)-calculus

informally, \( I = \lambda x . x \)
is the function which returns \( P \) when given the argument \( P \).

\[ M = \lambda f. \lambda x. \text{App}(f, x) \]

this the function (informally)

\[ f \mapsto (x \mapsto f(x)) \]

we give a second argument \( f \)

give a first argument \( x \)

\[ \text{App} (\text{App}(M, f), x) = \text{App}(f, x) \]

I will come back to this equation later.
We can use the tree notation:

\[
P = \text{App} \quad \text{App} \quad \text{App} \quad \lambda f \quad \lambda x \quad \text{App} \quad f \quad x
\]

\[
M = \text{App} \quad \lambda x \quad \lambda f \quad \text{App} \quad f \quad x
\]
3. \( K = \lambda x. \lambda y. y \)

( called a \( \lambda \)-term term of the \( \lambda \)-calculus)

\[ K \xrightarrow{\text{App}} \text{App}(\text{App}(K,p),Q) \quad \xrightarrow{\text{rewrites}} \quad Q \]

- \( KPQ \)
- \( \text{App}(\text{App}(K,p),Q) \)

\( K \) erases the first argument, returns the second argument.
in programming (in Haskell) K behaves a second projection

\[ \text{projtwo}(x, y) = y. \]

4 \[ \Delta = \lambda x . \text{App}(x, x) \]

the behaviour of \( \Delta \):

\[ \Delta \xrightarrow{\text{App}} P \]

\[ \text{App}(\Delta, P) \xrightarrow{} \text{App}(P, P) \]

\[ \Delta P \xrightarrow{} PP \]

the duplicator!
\[ \Delta : x \mapsto \text{App}(x, x) \uparrow \]
\[ x \in \text{Var} \]

In the \( \lambda \)-calculus, every term is a function and also every term is an argument. We should think of it as a purely formal calculus.

Later in the course:

the typed \( \lambda \)-calculus

a more "reasonable" situation
$\text{App}(\Delta, \Delta) \Rightarrow \text{App} \Rightarrow \text{App} \Rightarrow \text{App} \Rightarrow \ldots$

This is the story of a duplicator duplicating itself!

It is possible to construct $\lambda$-terms which never reach a result/value by rewriting.

Symbolic transformation invented in the 1930's
related to "referential transparency" in functional programming—the fact that the meaning of a program (\(\equiv\) term of the \(\lambda\)-calculus) does not change by rewriting it into another program.

6 as a matter of fact, the \(\lambda\)-calculus is so powerful that it is possible to define a \(\lambda\)-term \(Y\) such that

\[
\text{App}(Y,M) \xrightarrow{\text{rewrites}} \text{App}(M,\text{App}(Y,M))
\]
Definition of the fixpoint $Y$

by Turing:

$$W = \lambda w \lambda x. \text{App}(x, \text{App}(\text{App}(w, w), x))$$

$$Y = W$$

$$W = \lambda w \lambda x. \text{App}(x, \text{App}(\text{App}(w, w), x))$$

$$Y = \lambda \text{App} \lambda w \lambda x. \text{App}(x, \text{App}(\text{App}(w, w), x))$$
Y is called a fixpoint operator because the intuition is that it produces a fixpoint of the function M.

Def. a fixpoint of a function

\[ f : A \rightarrow A \]

is an element \( a \in A \) such that

\[ f(a) = a \]
Here:

\[ M(YM) = YM \]

fixpoint of \( M \)

in fact

\[ YM \xrightarrow{\text{rewrites}} M(YM) \]

"meaning" of \( YM \) same as "meaning" of \( M(YM) \)

discovered by Church and Turing