Proofs are Programs, Computation, and Constructive Mathematics

Yuriy Dupyn

2016

In this mathematical essay we motivate dependent function/pair types in set theoretic settings, and then we apply them in interpreting mathematical **proofs as programs**.
We construct Euclid's proof of infinitude of primes to gain intuition for **proofs as programs** and to illustrate a few points concerning **constructive mathematics**.
We conclude with a discussion about **the law of excluded middle** and **axiom of choice**.

The purpose of this essay is not to study a particular formal system —there are many freely available resources that do just that; its purpose is to convince the reader that **proofs as programs** are worthwhile to study, and give her basic intuition for this concept.
We do not discuss the nature of equality in type theory.
Some of the ideas are based on
15-819 Homotopy Type Theory lectures by Robert Harper,
and Five Stages of Accepting Constructive Mathematics lecture by Andrej Bauer

Conventions and notation:

Instead of writing $x \in A$ we write $x : A$,
In set-theoretic context $A \to B$ is the set of all mappings from A to B;
together $f : A \to B$ means $f \in (A \to B)$.
Partition P of a set X is a set of non-empty subsets of X whose union is X.
Also $A \to B \to C = A \to (B \to C)$.

Proposition	Types
$\top$	1
$a \wedge b$	$a \times b$
$a \Rightarrow b$	$a \to b$
$\bot$	0
$a \vee b$	$a + b$
$\forall x : A . B_x$	$\prod_{x:A} B_x$
$\exists x : A . B_x$	$\sum_{x:A} B_x$
Bool	2
Proof of proposition	Program of type
Logic	Type Theory

is a special case of

Dependent Function Type

We begin by discussing set-theoretic notions of cartesian product and disjoint union.

Let A, B, C be sets, and $a : A$, $b : B$, $c : C$.
Then we can form the **triple** $\langle a, b, c \rangle$ consisting of a, b, and c in that order.
We denote the set of all such triples by $A \times B \times C$.

We can generalize this idea to form infinite **sequences** of elements.
Let $r_1, r_2, r_3, \ldots : \mathbb{R}$, then we wish to form the sequence

$$rs := \langle r_0, r_1, r_2, \ldots \rangle.$$

We usually denote the set of such sequences indexed by natural numbers by $\mathbb{R}^{\mathbb{N}}$. But what do we mean by the dots

$$\ldots\ ?$$

Formally one can view the sequence **rs** as a function

$$rs : \mathbb{N} \longrightarrow \mathbb{R}.$$

So we can define

$$\mathbb{R}^{\mathbb{N}} := \{ s : \mathbb{N} \to \mathbb{R} \mid \text{true} \}.$$

There is no need to restrict ourselves to sequences of real numbers; each component can come from a different set.
Let X_n be a set for each $n : \mathbb{N}$; then the set of all sequences made from $X_1, X_2, \ldots$ in that order is

the cartesian product

$$\prod_{n:\mathbb{N}} X_n := \left\{ s : \mathbb{N} \longrightarrow \bigcup_{n:\mathbb{N}} X_n \;\middle|\; \begin{array}{l} \text{for all } n : \mathbb{N}, \\ s\,n : X_n \end{array} \right\}.$$

Ofcourse this generalizes further; we can replace $\mathbb{N}$ by arbitrary set. Given set I and set X_i for each $i : I$, we can define

$$\prod_{i:I} X_i := \left\{ f : I \to \bigcup_{i:I} X_i \;\middle|\; \begin{array}{l} \text{for all } i : I, \\ f\,i : X_i \end{array} \right\}.$$

This definition will give us a finer way to restrict functions.

Consider the following progression:

(i.) f is a function.

Here we don't know anything about f;
its inputs and outputs might be anything, i.e. its an untyped function.

(ii.) f is a function s.t.

$$\operatorname{dom} f = A.$$

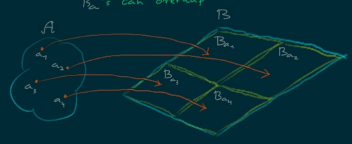
Now we know that inputs outside of A can be ignored; but we still don't know anything about output behaviour of f.

(iii.) $f : A \longrightarrow B$.

Here inputs/outputs are both bounded;
this is the usual notion of typed function.

We can go one step further.
Suppose B is the union of $\{ B_a \mid a : A \}$.

in the following picture it is assumed that $\{B_a \mid a : A\}$ partitions B, but in general B_a's can overlap



Let $f : \prod_{a:A} B_a$.

This means that $f : A \longrightarrow B$,
and for all $a : A$,
$f\,a : B_a$.

We define a more natural notation for the type:

$$\text{for all } a : A \,\bullet\, B_a := \prod_{a:A} B_a.$$

So

(iv.) $f : \text{for all } a : A \,.\, B_a$.

We say that f is dependently typed function.
since the outputs are bounded by (depend on) individual inputs.
Note that from dependent function type we can recover the usual function type, since

$$A \longrightarrow B = \text{for all } \_ : A \,.\, B,$$

i.e. $f : A \to B$ is a function whose outputs depend trivially on inputs.

example Let A be a set, and
$\mathrm{List}\ A$ be the set of all finite sequences over A.

We might want to have an operation of appending an element to a list:

$$\mathrm{cons} : A \longrightarrow \mathrm{List}\ A \longrightarrow \mathrm{List}\ A.$$

$$\mathrm{cons}\ x\ [x_1, \ldots, x_n] := [x, x_1, \ldots, x_n].$$

then we could define an operation that decomposes a list into its parts:

$$\mathrm{decompose} : \mathrm{List}\ A \longrightarrow A \times \mathrm{List}\ A$$

$$\mathrm{decompose}\ [x_1, x_2, \ldots, x_n] := \langle x_1, [x_2, \ldots, x_n] \rangle,$$

ex.
for $A = \mathbb{N}$,

$$\mathrm{decompose}\ [1, 2, 3] = \langle 1, [2, 3] \rangle.$$

But such a function is ill-defined;
what should be the result of

$$\mathrm{decompose}\ [\,]\ ?$$

Can we make the type of decompose more precise?

One solution is as follows.
Let for all $n : \mathbb{N}$,

$$\mathrm{List}\ n\ A := A^n$$

Then

$$\mathrm{decompose} : \text{for all } n : \mathbb{N} \,\bullet\, [\mathrm{List}\ (n+1)\ A \longrightarrow A \times \mathrm{List}\ n\ A].$$

$$\mathrm{decompose}\ n\ [x_1, x_2, \ldots, x_n] := \langle x_1, [x_2, \ldots, x_n] \rangle,$$

which is a compact way of saying that

$$\mathrm{decompose} : \mathbb{N} \longrightarrow \mathrm{List}\ A \setminus \{[\,]\} \longrightarrow A \times \mathrm{List}\ A$$

s.t. for all $n : \mathbb{N}$, $xs : \mathrm{List}\ A \setminus \{[\,]\}$,

$$\begin{array}{l} \text{if } \mathit{length}\ xs = n + 1 \\ \text{then} \left[\begin{array}{l} \quad \mathit{length}\ ys = n \\ \text{where } \langle \_, ys \rangle = \mathrm{decompose}\ xs \end{array} \right]. \end{array}$$

Note that we can write

$$\text{for all } n : \mathbb{N} \,\bullet\, [\mathrm{List}\ (n+1)\ A \longrightarrow A \times \mathrm{List}\ n\ A].$$

as

$$\text{for all } n : \mathbb{N} \,\bullet\, [\text{for all } \_ : \mathrm{List}\ (n+1)\ A \,.\, A \times \mathrm{List}\ n\ A].$$

example Let $s : \mathbb{N} \longrightarrow \mathbb{N}$ be the successor function,
and $k : \mathbb{N}$.
The following equations define unique function

$$\mathrm{add}_k : \mathbb{N} \longrightarrow \mathbb{N} :$$

$$\mathrm{add}_k\ 0 = k, \quad \mathrm{add}_k\ (s\,n) = s\,(\mathrm{add}_k\ n).$$

Note that $\mathrm{add}_k\ n = k + n$.

This is justified by recursion theorem on $\mathbb{N}$.

Theorem. Let A be a set, $a : A$, $g : A \to A$,
then
exists unique $f : \mathbb{N} \longrightarrow A$ s.t.

$$f\,0 = a,$$

for all $n : \mathbb{N}$,

$$f\,(s\,n) = g\,(f\,n).$$

In the example above we set $A = \mathbb{N}$, $a = 0$, $g = s : \mathbb{N} \to \mathbb{N}$.
In the proof of the recursion theorem we might use the following function

$$\mathrm{rec} : \text{for all } n : \mathbb{N}.\, [([0..n] \longrightarrow A) \longrightarrow ([0..s\,n] \longrightarrow A)]$$

$$\mathrm{rec}\ n\ f_n := f_{sn} \quad \text{where } f_{sn}\,k := \begin{array}{l} \text{if } k \le n \\ \text{then } f_n\,k \\ \text{else } g\,(f_n\,n) \end{array}$$ note that here $k = s\,n$.

Given an approximation of f we can improve on it with rec.
Thinking of functions as tables of values, we have

$$\mathrm{rec}\ 0\ \boxed{\begin{array}{c|c} 0 & a \end{array}} = \boxed{\begin{array}{c|c} 0 & a \\ \hline 1 & g\,a \end{array}}, \quad \mathrm{rec}\ 1 \circ = \boxed{\begin{array}{c|c} 0 & a \\ \hline 1 & g\,a \\ \hline 2 & g\,(g\,a) \end{array}} \quad \mathrm{rec}\ 2 \circ = \boxed{\begin{array}{c|c} 0 & a \\ \hline 1 & g\,a \\ \hline 2 & g\,(g\,a) \\ \hline 3 & g\,(g\,(g\,a)) \end{array}}.$$

example Let $\mathbb{F}$ be a field;
composition of linear maps has the following type:

$$\underbrace{\text{for all } \mathbb{V}_0, \mathbb{V}_1, \mathbb{V}_2}_{} : \mathrm{Vec}\,\mathbb{F} \,\bullet\, [(\mathbb{V}_1 \xrightarrow{\text{Lin}} \mathbb{V}_2) \longrightarrow (\mathbb{V}_0 \xrightarrow{\text{Lin}} \mathbb{V}_1) \longrightarrow (\mathbb{V}_0 \xrightarrow{\text{Lin}} \mathbb{V}_2)].$$

shortcut for
for all $\mathbb{V}_0 : \mathrm{Vec}\,\mathbb{F}$. for all $\mathbb{V}_1 : \mathrm{Vec}\,\mathbb{F} \ldots$

Dependent Pair Type

(*) Let A and B be two disjoint sets,
and $g : A \to X$, $h : B \to X$.
Let us define a new function f by
for all $a : A$,
$f\,a := g\,a$,
for all $b : B$,
$f\,b := h\,b$.

What is the domain of inputs of f?
Since A and B are disjoint we have
$$f : A \cup B \to X.$$

Now suppose we have the same setting as in (*)
except that A and B need not be disjoint.
Function f determined by equations
$$\text{for all } a : A,\ f\,a := g\,a,$$
$$\text{for all } b : B,\ f\,b := h\,b.$$
may become ill-defined;
for suppose we have $x : A \cap B$ with the property
$$g\,x \neq h\,x.$$
Then what becomes of $f\,x$?
To resolve this we need to consider
$x : A$ as **different** from $x : B$.

One solution is to **wrap** elements of A and B
in different containers; one for A and the other for B.
We'll have for each $a : A$, the element $\mathsf{L}a : \mathsf{L}A$,
and for each $b : B$, the element $\mathsf{R}b : \mathsf{R}B$.

L for left
R for right

If $x : A \cap B$, then we'll declare that $\mathsf{L}x \neq \mathsf{R}x$,
so $\mathsf{L}A$ and $\mathsf{R}B$ are disjoint.

Then we redefine f with
for all $a : A$,
$f(\mathsf{L}a) := g\,a$,
for all $b : B$,
$f(\mathsf{R}b) := h\,b$;

we get $f : \mathsf{L}A \cup \mathsf{R}B \to X$.

We'll define for any two sets A, B
$$A + B := \mathsf{L}A \cup \mathsf{R}B.$$

What if we had three sets A, B, C,
how could we define $A + B + C$?
First attempt might be to define
1. $A + B + C := (A+B) \cup \mathsf{R}C = (A+B)+C$

equivalently we could have used $A + (B + C)$

or
to think up new wrappers and set,
for example,
2. $A + B + C := \mathsf{W}_1 A \cup \mathsf{W}_2 B \cup \mathsf{W}_3 C$.

We'll choose a variant of 2.,
since, as we'll see, 1. doesn't generalize well.

define Let $\mathbf{n} := \{0, 1, \dots, n-1\}$,
so in particular $\mathbf{2} = \{0, 1\}$, $\mathbf{1} = \{1\}$, $\mathbf{0} = \{\}$.

define Let $A_0, A_1, \dots, A_{n-1}$ be sets.
Given $k : \mathbf{n}$, our wrapper for $x : A_k$
will be $\langle k, x\rangle$.
We define
$$\sum_{k:\mathbf{n}} A_k = A_0 + A_1 + \dots + A_{n-1} := \bigcup_{k:\mathbf{n}} \{\langle k, x\rangle \mid x : A_k\}.$$

In particular
$$\sum_{k:\mathbf{2}} A_k = A + B = \{\langle 0, a\rangle \mid a : A\} \cup \{\langle 1, b\rangle \mid b : B\},$$
$$\sum_{k:\mathbf{1}} A_k = \{\langle 0, a\rangle \mid a : A\}$$ ← which is basically A
$$\sum_{k:\mathbf{0}} A_k = \{\}.$$

Such definition generalizes by replacing $\mathbf{n}$ with arbitrary set.

define Let I be a set, and X_i is a set for each $i : I$.
We set

disjoint union

here the order is important

$$\sum_{i:I} X_i := \bigcup_{i:I} \{\langle i, x\rangle \mid x : X_i\} = \{\langle i, x\rangle \mid i : I,\ x : X_i\}.$$

Note that
$$\sum_{n:3} A_n = \{0\}\times A_1 \cup \{1\}\times A_1 \cup \{2\}\times A_2,$$
$$\sum_{n:3} A = \{0\}\times A \cup \{1\}\times A \cup \{2\}\times A = 3 \times A,$$
$$\sum_{n:\mathbb{N}} A_n = \bigcup_{n:\mathbb{N}} (\{n\}\times A_n),$$
$$\sum_{n:\mathbb{N}} A = \mathbb{N} \times A.$$

And in general suppose B is the union of $\{B_a \mid a : A\}$,
then
$$\sum_{a:A} B_a = \bigcup_{a:A} (\{a\}\times B_a) \subseteq A \times B.$$

So we can view elements in $\sum_{a:A} B_a$ as dependent pairs,
i.e. given $\langle x, y\rangle : \sum_{a:A} B_a$ we can deduce that $y : B_x$;
second component depends on the first.
That's why disjoint union is also called **dependent pair type**.

Of course we can make the second component
depend **trivially** on the first,
and in this way we recover the usual cartesian product
$$\sum_{\_:A} B = A \times B.$$

We get the concept of dependent triple, ..., n-tuple for free.

Let $\quad p : \sum_{a:A} \left[\sum_{b:B} C_{a,b}\right]$,
then $\quad p = \langle x, q\rangle$, where $x : A$, and $q : \sum_{b:B} C_{x,b}$ depends on x;
so $\quad q = \langle y, z\rangle$, where $y : B$, and $z : C_{x,y}$ depends on both x and y;

and finally we can think that
$$\langle x, y, z\rangle : \sum_{a:A} \left[\sum_{b:B} C_{a,b}\right].$$

Or
$$\langle x, y, z, w\rangle : \sum_{a:A} \left[\sum_{b:B} \left[\sum_{c:C} D_{a,b,c}\right]\right],$$ and so on.

Recall that $\sum_{b:\mathbf{2}} A_b = A_0 + A_1$,
so we can recover **binary** disjoint union from $\mathbf{2}$ and general dependent pair type.

Hence **dependent pair type** generalizes both **products** and **sums**.

We can think of dependent pair types as interfaces or specifications;
and objects of such a type **implement** the specification.

example Suppose we wanted a structure with one unary and one binary operation
with a distinguished element — think groups;
the specification might look like the following:

give me X : **Set** with $eq : X \times X \to \mathrm{Bool}$
$f : X \times X \to X$
$i : X \to X$
$e : X$.

[think that $\mathrm{Bool} := \mathbf{2}$,
and **Set** is the class of all sets]

The following implements the specification above:

take $\mathbb{N}$: **Set** with $(=) : \mathbb{N} \times \mathbb{N} \to \mathrm{Bool}$
$(+) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$(-) : \mathbb{N} \to \mathbb{N}$
$0 : \mathbb{N}$

In our language of **dependent pair types** we can write the specification as
$$\sum_{X:\mathbf{Set}} \Big[[X \times X \to \mathrm{Bool}] \times [X \times X \to X] \times [X \to X] \times [X] \Big],$$
and to give a possible implementation is to specify a member of this type,
which is a pair
$$\langle \mathbb{N}, \mathit{structure}\rangle \quad \text{where} \quad \mathit{structure} := \langle (=), (+), (-), 0\rangle.$$

Later we'll have a way of encoding properties like associativity into the specification.

Thinking of dependent pair types as specifications the Σ notation seems very unnatural.
We can define new notation, for example
$$\text{give } a : A \text{ with } B_a := \sum_{a:A} B_a,$$
but this is too verbose; instead we'll often use
$$\text{exists } a : A.\ B_a := \sum_{a:A} B_a.$$

[pick $a : A.\ B_a$ is also a good choice]

Actually, it mainly comes from the relation of dependent pair type
to the existential quantifier in logic; this relation will be discussed in due time.

Proofs are Programs

In classical logic we can think of the proposition φ as denoting its **truth value**, so in a certain sense

$$\varphi : \mathrm{Bool}$$

Predicate on $\mathbb{N}$ that says that a particular number is a perfect square can be thought of as a function

$$\mathrm{PerfectSquare} : \mathbb{N} \longrightarrow \mathrm{Bool}$$

$$\mathrm{PerfectSquare}\ n := \text{true if exists } k : \mathbb{N}.\ k^2 = n \text{ else false}.$$

Or a relation on $\mathbb{N}$ is a function

$$(=) : \mathbb{N} \times \mathbb{N} \longrightarrow \mathrm{Bool}.$$

In **intuitionistic logic** such an interpretation is **completely wrong**.
When one encounters **intuitionistic logic** for the first time, the greatest difficulty lies in resisting the temptation to think of propositions, predicates, and relations as
boolean-valued functions.

Consider a programmer
who spent his entire life writing in **imperative** languages;
he has trouble giving up thinking of a program as a sequence of statements
when learning **functional** programming, and this hinders him greatly.
A novice has an advantage over a programmer cursed with imperative thinking.

In the same way a mathematician trained in classical mathematics has trouble adjusting to intuitionistic logic.
So let us forget the boolean thinking and begin.

First we give intuition for propositions, predicates, and relations without saying what a proof is; that will come latter.
We can think of a proposition φ as the collection of all of the proofs of φ:

$$\varphi = \{ p \mid p \text{ is a proof of } \varphi \}$$

So '$p : \varphi$' becomes 'p is a proof of φ'.
Do not think of the above as a definition;
the purpose of this equality is to guide our future definitions.
When $\{ p \mid p \text{ is a proof of } \varphi \}$ is **inhabited**, that is it has at least one proof,
then we say that φ is **true** or **has a proof**; if it's **empty**, then φ is **false** or **refuted**.

To understand predicates recall **dependent pair type** as a specification.
We think of the predicate PerfectSquare on $\mathbb{N}$ as a specification
for those natural numbers that are perfect squares;
so if you want to say that you have a number that satisfies the predicate,
you have to construct that number together with the proof that it has the desired property.
So we set

$$\mathrm{PerfectSquare} := \text{exists } n : \mathbb{N}. \underbrace{\left\{ p \;\middle|\; \begin{matrix} p \text{ is a proof that} \\ \text{exists } k : \mathbb{N} \text{ s.t.} \\ k^2 = n \end{matrix} \right\}},$$

this is a collection of proofs, so it's a proposition

and if you want 4 to satisfy PerfectSquare, you have to construct
a proof p that proves that 4 is a perfect square (2 would have to be involved);
then you may state $\langle 4, p \rangle : \mathrm{PerfectSquare}$.

Similarly, we can think of relations as specifications

$$(=) := \text{exists } \langle m, n \rangle : \mathbb{N} \times \mathbb{N}. \left\{ p \;\middle|\; \begin{matrix} p \text{ is a proof of} \\ m \text{ is equal to } n \end{matrix} \right\}.$$

Ofcourse when doing mathematics, we won't use such cumbersome notation.
Instead of writing

$\langle 4, p \rangle : \mathrm{PerfectSquare}$		PerfectSquare 4 by p
$\langle \langle 2+2, 4 \rangle, q \rangle : (=)$	we'll write	$2+2=4$ since q.
$\langle 2, p \rangle : \mathrm{Even}$		p, so Even 2.

When its obvious that $\langle a, p \rangle : P$,
we'll just write $P\,a$, suppressing p.
Also given a predicate P on X, and $a : X$,
we'll interpret $P\,a$ as the collection of all proofs p s.t. $\langle a, p \rangle : P$.
i.e. $\quad P\,a = \{ p \mid p \text{ is a proof s.t. } \langle a, p \rangle : P \}$.
Similarly for a relation R we have $R\,a\,b = \{ p \mid p \text{ is a proof s.t. } \langle a, b, p \rangle : R \}$.
Note that Even 2 is inhabited, but Even 1 is uninhabited.

So, what are **proofs**?
Proofs are a special kind of **programs**.
But what is a **program**, how do we **make** one — how is **programming** done?
Program is a template for **computation**;
computation is a continuing process of **construction** and **deconstruction**.
For example we could create a **program** (pattern of **computation**)
that when **applied** to a **computation** that **constructs** a natural number n greater than 1,
produces a **computation** that **constructs** the least prime number p that divides n;
this is done by **constructing** 2, and dividing n by 2, (which is a form of **deconstruction**)

- if the division leaves a remainder,
  then continue the **process**
  else 2 is the desired result

  branching is a form of **construction/deconstruction**

constructing 3, and dividing n by 3;

- if the division leaves a remainder,
  then continue the **process**
  else 3 is the desired result

constructing 4, and dividing n by 4;

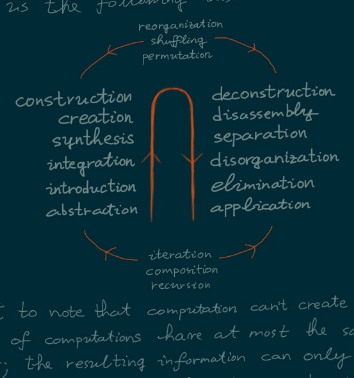
- if the division leaves a remainder,
  then continue the **process**
  else 4 is the desired result

  this is redundant, since 4 is not a prime, but nothing forbids us to create redundancy; and you have to realize that deciding if a number is prime is also a computation, and this program can be used to decide primality, so its not that redundant after all.

$\vdots$

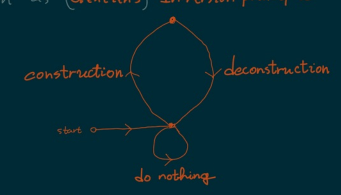
this process stops after a finite number of steps.
Note that **iteration** above can be thought of as a **construction/deconstruction** itself.

This gives us the following associations:



Its important to note that computation can't create something from nothing;
all results of computations have at most the same information content
as its inputs; the resulting information can only be rearranged
or restructured, but no new information can be gained.
Cost is implicit in the inputs; this is called **no free lunch principle**.
We'll return to this when discussing the law of excluded middle.
But programs can easily lose information.
For example there is little difficulty in constructing computations that never finish.

If we want proofs to be useful, they can't diverge towards infinity;
they have to prove their proposition in finite number of steps.
This is guaranteed by another principle.
Suppose someone gives us a collection of finished constructions,
and we combine them into a proof. Then our principle
says that if we wish, we can disassemble the proof
into its constituent parts completely, so we get back all of the
constructions that were given to us.
This is known as (Gentzen's) **Inversion principle** illustrated below



Inversion and **No free lunch principles** give us
the **Principle of conservation of information** for **proofs**.

To get a solid intuition for the nature on computation,
and to gain enlightment in general read

Structure and Interpretation of Computer Program
by Harold Abelson and Gerald Jay Sussman with Julie Sussman.

Now, armed with these principles,
we'll define proofs and their propositions/types.

define Type $\top$ is the **trivially true** proposition.
construction $\top$ has only one proof,
we'll denote it by $\langle\rangle$;
deconstruction Proof $\langle\rangle$ has no internal structure,
so there is nothing to deconstruct.

We can think that $\top = 1$.

define Let φ, ψ be propositions, then
their **conjunction** $(\varphi \wedge \psi)$ is a proposition.
construction Given proofs $p : \varphi$, $q : \psi$, then $\langle p, q \rangle : \varphi \wedge \psi$.
deconstruction Only proofs of $\varphi \wedge \psi$ are of the form above
If $c : \varphi \wedge \psi$, then there are $p : \varphi$, $q : \psi$ s.t. $c = \langle p, q \rangle$
We can denote the components of c without giving them explicit names by
$\mathrm{fst}\ c := p$, $\quad \mathrm{snd}\ c := q$.

Think that $\varphi \wedge \psi = \varphi \times \psi$,
i.e. proof of $(\varphi \wedge \psi)$ is a proof of φ together with a proof of ψ.

define Let φ, ψ be propositions, then
their **implication** $(\varphi \Rightarrow \psi)$ is a proposition.
construction Assume that for each proof $p : \varphi$,
we have precisely one proof $q_p : \psi$.
This determines a function

$$f : \varphi \longrightarrow \psi$$
$$f\,p := q_p$$

Then f is a proof of $(\varphi \Rightarrow \psi)$.
deconstruction If f is a proof of $(\varphi \Rightarrow \psi)$,
then f is a function $f : \varphi \longrightarrow \psi$;
in particular given $p : \varphi$, we get $f\,p : \psi$.

Think $(\varphi \Rightarrow \psi) = \overbrace{\varphi \longrightarrow \psi}^{\text{function space}}$,
i.e. a proof of $(\varphi \Rightarrow \psi)$ transforms any proof of φ into a proof of ψ.

define $\bot$ is a proposition; we call it **contradiction**.
construction Proposition $\bot$ has no proof, so there is nothing to construct.
deconstruction If we had a proof of $\bot$,
then we would have a proof of every proposition.
Such a hypothetical deconstruction is valid,
since there isn't a proof of $\bot$, so we can't get anything for free from it.
This is called the **Principle of explosion** or **From contradiction everything follows**.

Think $\bot = \{\}$.

define Let φ, ψ be propositions, then
their **disjunction** $(\varphi \vee \psi)$ is a propositon.
construction i. given $p : \varphi$, then $\langle 0, p \rangle : \varphi \vee \psi$;
ii. given $q : \psi$, then $\langle 1, q \rangle : \varphi \vee \psi$.
deconstruction given $d : \varphi \vee \psi$, then
either there exists $p : \varphi$ s.t.
$\langle 0, p \rangle = d$,
or there exists $q : \psi$ s.t.
$\langle 1, q \rangle = d$.

Think $\varphi \vee \psi = \varphi + \psi$,
i.e. a proof of $(\varphi \vee \psi)$ is a proof of φ or a proof of ψ.

define Let A be a type, and for each $a : A$ we have a proposition ψ_a.
Then $(\text{for all } x : A.\ \psi_x)$ is a proposition
construction For each $a : A$ let us construct $q_a : \psi_a$.
Then we have a function

$$f : \prod_{x:A} \psi_x,$$
$$f\,a := q_a,$$

and f is a proof for all $x : A.\ \psi_x$.
deconstruction If f is a proof for all $x : A.\ \psi_x$.
then f is dependently typed function

$$f : \prod_{x:A} \psi_x.$$

Think $\text{for all } x : A.\ \psi_x = \prod_{x:A} \psi_x$.

define Let A be a type, and for each $a : A$ we have a proposition ψ_a.
Then $(\text{exists } x : A.\ \psi_x)$ is a proposition.
construction Let $a : A$, and $q : \psi_a$,
then $\langle a, q \rangle$ is a proof of $\text{exists } x : A.\ \psi_x$.
deconstruction If $c : \text{exists } x : A.\ \psi_x$
then there is $a : A$,
there is $q : \psi_a$ s.t.
$c = \langle a, q \rangle$.

Think $\text{exists } x : A.\ \psi_x = \sum_{x:A} \psi_x$. If $a : A$, we name $P := \text{exists } x : A.\ \psi_x$, then $P\,a := \psi_a$.

define Let φ be a proposition, we define the **negation** of φ as

$$\neg\varphi := \varphi \Rightarrow \bot.$$

So we can think that $\neg\varphi = \varphi \longrightarrow \bot$;
hence to have a proof of $\neg\varphi$ is to have
a general way of turning every proof of φ into a proof of $\bot$.

The **law of excluded middle** (LEM) states that
for every proposition φ.

$$\varphi \vee \neg\varphi \text{ is true.}$$

At first look it seems like a reasonable thing to accept,
but if you try to construct its proof, you will fail!
That doesn't mean that its refuted.
In fact for every prop. φ. $\neg\neg(\varphi \vee \neg\varphi)$ is true, and we'll prove it later.
Now we'll show that if we were to accept LEM, then **No free lunch principle** is violated.
Let $n := 2^{74207281} - 1$. Then by LEM, p is a prime or it isn't.
So either we have a proof of 'n is a prime' or we have a proof of 'n is not a prime' for free,
which is ridiculous.

Infinitude of Primes

To gain intuition for Proofs as Programs we'll construct Euclid's proof of infinitude of primes in a semi-formal manner.
We assume that $\mathbb{N}$ is equipped with usual relations and operations, and

$$\mathrm{fct} : \mathbb{N} \longrightarrow \mathbb{N}$$

is the factorial function, or more precisely its an **implementation** of the factorial function,

define Let $\mathrm{Even} := \text{exists } n.\ \text{exists } k : \mathbb{N}.\ (n = 2 \cdot k)$.

Instead of writing the above we'll often use the following notation:

define Let $n : \mathbb{N}$, then $\mathrm{Even}\ n := \text{exists } k : \mathbb{N}.\ (n = 2 \cdot k)$.

Interpret this as a friendlier shorthand for the first definition.

define Let $m, n : \mathbb{N}$, then

$$m \mid n := \text{exists } k : \mathbb{N}.\ n = k \cdot n.$$

i.e. $(\mid) := \text{exists } m : \mathbb{N}.\ \text{exists } n : \mathbb{N}.\ \text{exists } k : \mathbb{N}.\ n = k \cdot n.$

The type $\neg(m \mid n)$ is abbriviated as $m \nmid n$.

We define

$$(\text{if } \varphi \text{ then } \psi) := \varphi \Rightarrow \psi.$$

define Given $p : \mathbb{N}$ we set

$\mathrm{Prime}\ p := p > 1,$
  for all $d : \mathbb{N}$
   if $d > 1,\ d \mid p$
   then $d = p$.

Now we'll define a function that computes the least prime divisor of its input; ofcourse there is no such thing for 1 because our definition of primes does not admit 1.
We could have made the function dependently typed, but for our purposes this is not needed, so we'll set it to 1, so atleast $(\mathrm{leastPrimeDivisor}\ 1) \mid 1$.

define
$\mathrm{leastPrimeDivisor} : \mathbb{N} \longrightarrow \mathbb{N}$
$\mathrm{leastPrimeDivisor}\ 0 := 2$
$\mathrm{leastPrimeDivisor}\ 1 := 1$
$\mathrm{leastPrimeDivisor}\ n := \left[\ \mathrm{iter}\ 2 \ \textbf{where}\ \ \mathrm{iter} : \mathbb{N} \longrightarrow \mathbb{N};\ \ \mathrm{iter}\ k := \text{if } k \mid n \text{ then } k \text{ else } \mathrm{iter}\ (k+1)\ \right].$

We'll assume the following properties of leastPrimeDivisor without proof:

i. for all $n : \mathbb{N}$.
  if $n \neq 1$,
  then $\mathrm{Prime}\ (\mathrm{leastPrimeDivisor}\ n)$;

ii. for all $n : \mathbb{N}$.
  $(\mathrm{leastPrimeDivisor}\ n) \mid n$.

Instead of the traditional

Theorem x. φ.
Proof. … ■

we'll write

proof $x : \varphi$
 … ■

to emphasize the importance of proofs over propositions.

proof primeSuccessor:
 for all $n : \mathbb{N}$.
  exists $p : \mathbb{N}$.
   $\mathrm{Prime}\ p$,
   $n < p$.

We have n, we create $q := \mathrm{fct}\ n + 1$,
and compute $p := \mathrm{leastPrimeDivisor}\ q$.
Then we'll show that
 for all $0 < k < n$.
  $p \nmid k$,
and conclude that $n < p$.

Let $n : \mathbb{N}$.
Define $q := \mathrm{fct}\ n + 1$.
Note that $q > 1$.
Let $p := \mathrm{leastPrimeDivisor}\ q$.
Note that $\mathrm{Prime}\ p$.

proof helper:
 for all $k : \mathbb{N}$,
  if $0 < k \leq n$
  then $p \nmid k$

Let $k : \mathbb{N}$ s.t. $0 < k \leq n$.
Seeking a contradiction, assume $p \mid k$.
Note that since $0 < k \leq n$, we have
 $k \mid \mathrm{fct}\ n$.
Then by transitivity of $\mid$ we have
 $p \mid \mathrm{fct}\ n$.
So $p \nmid (\mathrm{fct}\ n + 1)$.
But $q = \mathrm{fct}\ n + 1$, and $p = \mathrm{leastPrimeDivisor}\ q$.
so $p \mid q$ and $p \nmid q$,
a contradiction. ■

proof loop:
 for all $k : \mathbb{N}$
  if $0 < k \leq n$
  then $k < p$.

define Let $k : \mathbb{N}$, set
 $P\,k = \text{if } 0 < k \leq n \text{ then } k < p.$

Its sufficient to prove that
 for all $k : \mathbb{N}.\ P\,k$

Let $k : \mathbb{N}$, and
 $\mathrm{IndH} : [\text{for all } k' : \mathbb{N}\ \text{ if } k' < k \text{ then } P\,k']$

By mathematical induction its sufficient to prove that
 $P\,k$

Assume $0 < k \leq n$. We need to prove $k < p$.
Note that $k = 1$ or $k > 1$.
Case $k = 1$. Then $1 < p$, since $\mathrm{Prime}\ p$. ■
Case $k > 1$.
 Then, since $k - 1 < k$,
  we get from $\mathrm{IndH}(k-1) : (\text{if } k-1 < k \text{ then } P\,(k-1))$
  that $P\,(k-1)$.
 But $0 < k-1 \leq n$, so $k - 1 < p$, i.e.
  $0 < k \leq p$.
 Note helper k gets us $p \nmid k$.
 So $k < p$. ■
So in both cases $k < p$. ■

Now we'll show that $n < p$
Case $n = 0$.
 Then $q = \mathrm{fct}\ 0 + 1 = 2$, and $p = 2$.
 So $n < p$. ■
Case $n > 0$.
 Then $0 < n \leq n$,
 so $\mathrm{loop}\ n : (\text{if } 0 < n \leq n \text{ then } n < p)$ gives us
  $n < p$. ■
In both cases $n < p$. ■

Note that from primeSuccessor we can extract an algorithm that given n as input computes prime $p > n$; explicitly

$$p = \mathrm{leastPrimeDivisor}\ (\mathrm{fct}\ n + 1).$$

Even though the algorithm is computationally infeasible for large n, atleast we have an algorithm; in **classical** mathematics we don't even get that.
This demonstrates that **constructive** mathematics gives **first-class status** to **proofs** and **algorithms** as mathematical objects; actually the former is a special case of the latter.
Proofs of propositions are more important than propositions, which reflects the categorical way of thinking that morphisms are more important than objects.
Since proofs are true mathematical objects, we can give them names, and construct them wherever we like it — for example in a body of another proof.
So we do not need to say that certain proofs are lemmas, because it will be obvious that they serve the role of a helper.
And when the main proof is finished, they won't pollute the namespace.

Ofcourse we didn't have to use leastPrimeDivisor to compute a prime factor of a number; instead we could have proven

for all $f : \mathbb{N} \longrightarrow \mathbb{N}$.
 if for all $n : \mathbb{N}$
   if $n \neq 1$ then $\mathrm{Prime}\ (f\,n)$,
   $f\,n \mid n$,
 then for all $n : \mathbb{N}$.
   exists $p : \mathbb{N}$.
    $\mathrm{Prime}\ p$
    $n < p$

taking a **hypothetical** factorization algorithm as input.
This shows that in **constructive mathematics** we do not loose the **axiomatic freedom**.

Just to demonstrate as we promised that we have a specification language powerful enough to encode properties such as associativity of an operation, we provide definitions for common algebraic structures.

define $\mathrm{Monoid} := \text{exists } M : \mathrm{Type}.$
 exists $(\cdot) : M \times M \longrightarrow M$.
 exists $e : M$.
 for all $\langle x, y, z \rangle : M \times M \times M$.
  $(x \cdot y) \cdot z = x \cdot (y \cdot z)$,
 for all $x : M$.
  $x \cdot e = x$,
 for all $x : M$.
  $e \cdot x = x$.

We can think of Monoid as ternary relation, so we can write $\mathrm{Monoid}\ \mathrm{String}\ (+\!+)\ \varepsilon$ for the proposition
 for all $\langle x, y, z \rangle : \mathrm{String} \times \mathrm{String} \times \mathrm{String}$.
  $(x +\!+ y) +\!+ z = x +\!+ (y +\!+ z)$,
 for all $x : \mathrm{String}$.
  $x +\!+ \varepsilon = x$,
 for all $x : \mathrm{String}$.
  $\varepsilon +\!+ x = x$.

This makes the following possible.

define $\mathrm{Group} := \text{exists } G : \mathrm{Type}.$
 exists $(\cdot) : G \times G \longrightarrow G$.
 exists $e : G$.
 exists $i : G \longrightarrow G$.
 $\mathrm{Monoid}\ G\ (\cdot)\ e$
 for all $x : G$.
  $x \cdot (i\,x) = e$.

define $\mathrm{Ring} := \text{exists } R : \mathrm{Type},$
 $(+) : R \times R \longrightarrow R$,
 $0 : R$,
 $(-) : R \longrightarrow R$
 $(\cdot) : R \times R \longrightarrow R$,
 $1 : R$.
 $\mathrm{Group}\ (+)\ 0\ (-)$,
 for all $\langle x, y \rangle : R \times R$.
  $x + y = y + x$,
 $\mathrm{Monoid}\ (\cdot)\ 1$,
 for all $\langle x, y, z \rangle : R \times R \times R$.
  $x \cdot (y + z) = x \cdot y + x \cdot z$,
 for all $\langle x, y, z \rangle : R \times R \times R$.
  $(x + y) \cdot z = x \cdot z + y \cdot z$.

syntactic sugar for
 exists …
 exists …
 ⋮

Law of Excluded Middle
Axiom of Choice

define $\mathrm{LEM} := \textbf{for all } \varphi : \mathrm{Prop}. (\varphi \vee \neg\varphi)$.

Even though we do not use LEM,
it doesn't mean that we **refute** it.
Now we'll show that refuting LEM
leads to a contradiction.

proof $\mathrm{not}^2\mathrm{LEM}$:

$$\textbf{for all } \varphi : \mathrm{Prop}. \neg\neg(\mathrm{LEM}\ \varphi).$$

Let $\varphi : \mathrm{Prop}$. Note that

$$\neg\neg(\mathrm{LEM}\ \varphi) = \neg(\mathrm{LEM}\ \varphi) \longrightarrow \bot.$$

Let $f : \neg(\mathrm{LEM}\ \varphi)$, now its sufficient to show $\bot$.

Note $\neg(\mathrm{LEM}\ \varphi) = (\varphi \vee \neg\varphi) \longrightarrow \bot$,

so we get

$$h_1 : \varphi \longrightarrow \bot, \qquad h_2 : \neg\varphi \longrightarrow \bot \text{ s.t.}$$

$$\textbf{for all } q : \varphi.\ h_1 q = f q, \qquad \textbf{for all } q : \neg\varphi.\ h_2 q = f q.$$

Note that $\varphi \to \bot = \neg\varphi$, hence we get

$$h_2\, h_1 : \bot. \quad \blacksquare$$

How to make sense of this?
In contstructive math we forbid global use of LEM.
It doesn't mean we can never say that
$\varphi \vee \neg\varphi$ for some specific φ.

Local use is fine provided we've proven $\varphi \vee \neg\varphi$.

In the absence of the *law of excluded middle*
the proposition $\textbf{for all } n : \mathbb{N}.\ \mathrm{Prime}\ p \textbf{ or } \neg\mathrm{Prime}\ p$

becomes non-trivial, since to have its proof is
to have an algorithm that for each $n : \mathbb{N}$
computes a proof of $\mathrm{Prime}\ p$ **or** $\neg\mathrm{Prime}\ p$,
i.e. computes a proof of $\mathrm{Prime}\ p$
or computes a proof of $\neg\mathrm{Prime}\ p$.

When proving a theorem that requires
global excluded middle, we can include
it as an additional assumption; then ofcourse
we are proving a weaker theorem,
but that's okay – we can think that this is done for every theorem classically.

This has an interesting corollary:

i. Classical theorems **are weaker** than constructive theorems;
Constructive theorems **are stronger** than classical theorems;

ii. In constructive mathematics we loose nothing
with respect to classical mathematics;

iii. 'Taking the Principle of The Excluded Middle from the mathematician
is the same as prohibiting the boxer the use of his fist' is **false**

Note that if we have a proof of

$$\textbf{for all } x : A.\ \textbf{exists } y : B.\ P\,x\,y,$$

then we can transform it into a proof of

$$\textbf{exists } f : A \longrightarrow B.\ \textbf{for all } x : A.\ P\,x\,(f\,y),$$

which looks like *the axiom of choice*.

We cheat here by not qantifying over A, B, and P

proof constructiveChoice:
$$\textbf{if for all } x : A.\ \textbf{exists } y : B.\ P\,x\,y$$
$$\textbf{then exists } f : A \longrightarrow B.\ \textbf{for all } x : A.\ P\,x\,(f\,x).$$

Let $h : \textbf{for all } x : A.\ \textbf{exists } y : B.\ P\,x\,y.$

Note that the second component would have been a proof of $P\,a\,b$.

define $f : A \longrightarrow B$,
$$f\,a := [\,b \textbf{ where } \langle b, \_\rangle := h\,a\,]$$

Let $a : A$; then we have
$$\langle f\,a, p\rangle = h\,a$$
where p is a proof of $P\,a\,(f\,a)$.

define $g : \textbf{for all } x : A.\ P\,x\,(f\,x)$
$$g\,x := [\,p \textbf{ where } \langle \_, p\rangle := h\,x\,].$$

Then $\langle f, g\rangle : \textbf{exists } f : A \longrightarrow B.\ \textbf{for all } x : A.\ P\,x\,(f\,x).$ $\blacksquare$

But since our interpretation of *exists* is
much stronger than in classical mathematics,
the proof is not surprising.
The equivalent of *the axiom of choice*
in constructive setting would be

$$\mathrm{AC} := \textbf{if for all } x : A.\ \neg\neg\,\textbf{exists } y : B.\ P\,x\,y$$
$$\textbf{then exists } f : A \longrightarrow B.\ \textbf{for all } x : A.\ P\,x\,(f\,x),$$

which is much stronger and unprovable.
Now we'll demonstrate that LEM is just a special case of AC;
to do that we'll need a kind of inverse for constructiveChoice.

proof $\mathrm{constructiveChoice}^{-1}$:
$$\textbf{if exists } f : A \longrightarrow B.\ \textbf{for all } x : A.\ P\,x\,(f\,x)$$
$$\textbf{then for all } x : A.\ \textbf{exists } y : B.\ P\,x\,y.$$

Let $\langle f, g\rangle : \textbf{exists } f : A \longrightarrow B.\ \textbf{for all } x : A.\ P\,x\,(f\,x)$,

and $a : A$. Then
$$f\,a : B,$$
and $P\,a\,(f\,a)$. $\blacksquare$

Now we are ready.

proof $\_ :$ **if** AC **then** LEM

Understanding that
$$\textbf{exists } b : \mathrm{Bool}.\ A_b = A_{false} \vee A_{true}$$
is crucial here.

In this proof we make the following convention:
Given $\varphi : \mathrm{Prop}$, we denote
$$\varphi_{false} := \neg\varphi, \quad \varphi_{true} := \varphi.$$

Then $\varphi \vee \neg\varphi = \textbf{exists } b : \mathrm{Bool}.\ \varphi_b$

actually it should be $\neg\varphi \vee \varphi = \textbf{exists } b : \mathrm{Bool}.\ \varphi_b$
but that is irrelevant for the understanding of the proof.

Let choice : AC. Note that
$$\mathrm{not}^2\mathrm{LEM} : \textbf{for all } \varphi : \mathrm{Prop}.\ \neg\neg(\textbf{exists } b : \mathrm{Bool}.\ \varphi_b),$$

so we get
$$\text{choice not}^2\mathrm{LEM} : \textbf{exists } f : \mathrm{Prop} \longrightarrow \mathrm{Bool}.\ \textbf{for all } \varphi : \mathrm{Prop}.\ \varphi_{(f\,\varphi)}.$$

So
$$\mathrm{constructiveChoice}^{-1}(\text{choice not}^2\mathrm{LEM}) : \textbf{for all } \varphi : \mathrm{Prop}.\ \textbf{exists } b : \mathrm{Bool}.\ \varphi_b,$$

i.e. $\textbf{for all } \varphi : \mathrm{Prop}. (\varphi \vee \neg\varphi)$, which is LEM. $\blacksquare$