Programs and Proofs

Mechanizing Mathematics with Dependent Types

Ilya Sergey

IMDEA Software Institute

ilyasergey.net/pnp-2014

Foreword

Formal Mathematics
as a branch
of Computer Science

Computer Science

Computer Science

- Computation and complexity
- Information and coding theory
- Data structures and algorithms
- Programming languages
- Formal methods and logics
- Security and cryptography
- Computer networks
- Databases
- Artificial Intelligence
- Computer graphics
- Computer/Human interaction
- Computer architecture
- Software Engineering
- ...

Mathematics

Mathematics

— *is what mathematicians do.*

Formal Mathematics

Formal Mathematics

makes rigorous statements…

Formal Mathematics

makes rigorous statements…

… and proves them.

What is a *proof* ?

Poor definition

A proof is sufficient evidence
or an argument for the truth of a proposition.

Poor definition

A proof is sufficient evidence
or an argument for the truth of a proposition.

Better definition

Better definition

A proof is a sequence of statements,

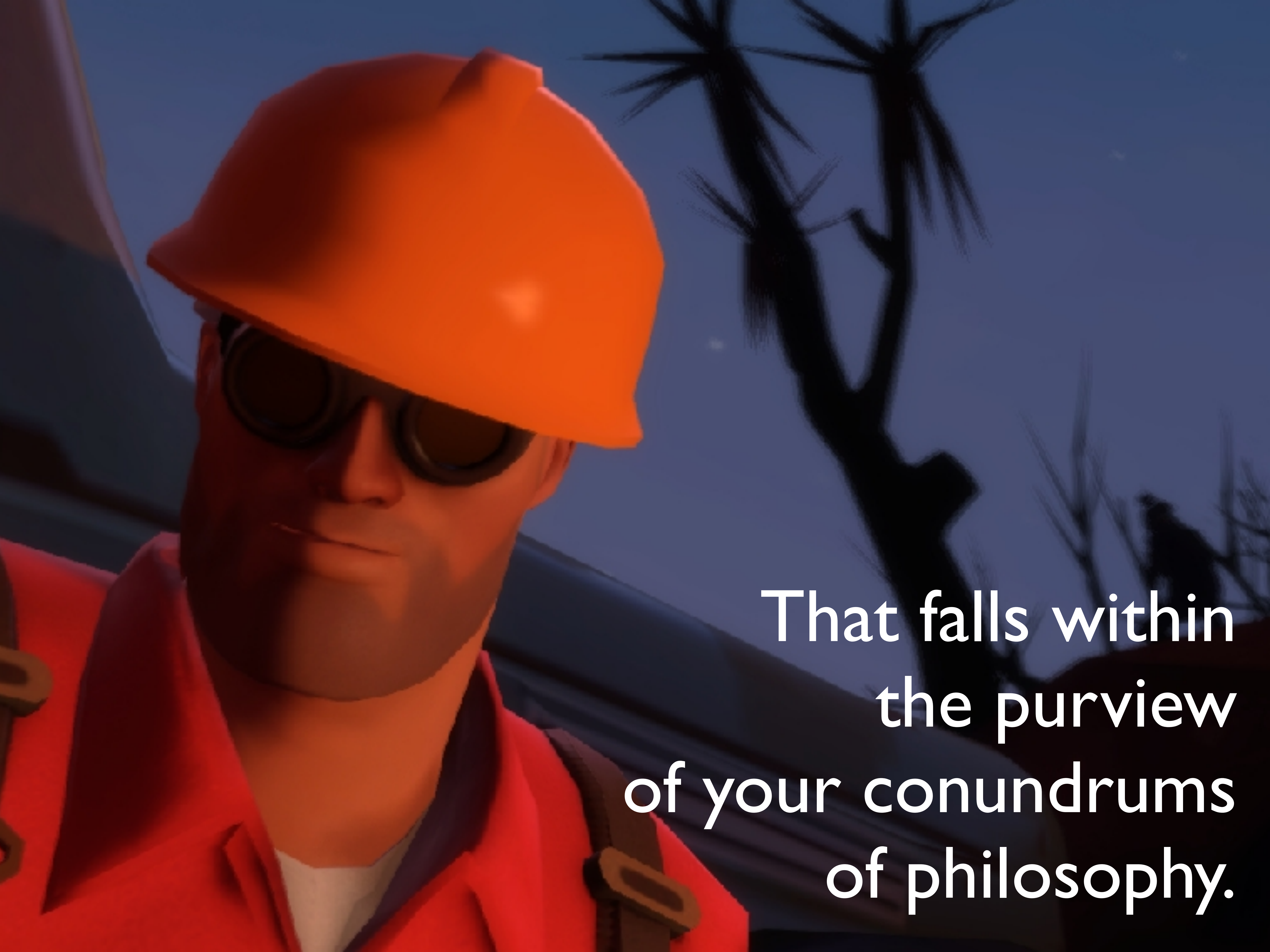
Better definition

A proof is a sequence of statements,
each of which is either validly
derived from those preceding it
or is an axiom or assumption,

Better definition

A proof is a sequence of statements,
each of which is either validly
derived from those preceding it
or is an axiom or assumption,
and the *conclusion* of which, is the statement
of which the *truth* is thereby established.

What is the *truth*?


That falls within
the purview
of your conundrums
of philosophy.

A proposition is considered to be *true*
(in a given set of assumptions and axioms)
if a proof of it can be *constructed*.

A proposition is considered to be *true*
(in a given set of assumptions and axioms)
if a proof of it can be *constructed*.

(so the truth *constructive* is *relative*)

What about *falsehood*?

What about *falsehood*?

A proposition is *false*
if no proof can be *derived* for it.

How do we *construct* proofs?

How do we *construct* proofs?

By using *rules* and *hypotheses*.

A hypothesis

Assuming that the proposition *A* is true, we can derive that *A* is true.

A hypothesis

Assuming that the proposition *A* is true, we can derive that *A* is true.

In formal logical notation: $\boxed{A \vdash A}$

Implication introduction

If, assuming that the proposition A is true, we can derive the proof of the proposition B, then we can derive the implication $A \Rightarrow B$.

Implication introduction

If, assuming that the proposition A is true, we can derive the proof of the proposition B, then we can derive the implication $A \Rightarrow B$.

In formal logical notation:

$$\boxed{\dfrac{A \vdash B}{\vdash A \Rightarrow B}}$$

Modus ponens

The proposition *A* is true, and, moreover, *A* being true implies that *B* is true; then we can derive that *B* is true.

Modus ponens

The proposition A is true, and, moreover, A being true implies that B is true; then we can derive that B is true.

$$\boxed{\dfrac{\vdash A \qquad \vdash A \Rightarrow B}{\vdash B}}$$

Conjunction introduction/elimination

From "A is true" and "B is true", we can derive that $A \wedge B$ is true as well.

Conjunction introduction/elimination

From "A is true" and "B is true", we can derive that $A \wedge B$ is true as well.

$$\boxed{\dfrac{\vdash A \qquad \vdash B}{\vdash A \wedge B}}$$

Conjunction introduction/elimination

From "A is true" and "B is true", we can derive that $A \wedge B$ is true as well.

$$\boxed{\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}}$$

From "$A \wedge B$ is true" we can derive that A is true and that B is true.

Conjunction introduction/elimination

From "*A* is true" and "*B* is true", we can derive that $A \wedge B$ is true as well.

$$\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}$$

From "$A \wedge B$ is true" we can derive that *A* is true and that *B* is true.

$$\frac{\vdash A \wedge B}{\vdash A} \qquad \frac{\vdash A \wedge B}{\vdash B}$$

Disjunction introduction/elimination

From “A is true” or “B is true”, we can derive that $A \vee B$ is true.

Disjunction introduction/elimination

From "A is true" or "B is true", we can derive that $A \vee B$ is true.

$$\boxed{\dfrac{\vdash A}{\vdash A \vee B}} \qquad \boxed{\dfrac{\vdash B}{\vdash A \vee B}}$$

Disjunction introduction/elimination

From "A is true" or "B is true", we can derive that $A \vee B$ is true.

$$\boxed{\frac{\vdash A}{\vdash A \vee B}} \qquad \boxed{\frac{\vdash B}{\vdash A \vee B}}$$

From "$A \vee B$ is true", "$A \Rightarrow C$" and "$B \Rightarrow C$", we can derive that C is true (*case analysis*).

Disjunction introduction/elimination

From "A is true" or "B is true", we can derive that $A \vee B$ is true.

$$\boxed{\dfrac{\vdash A}{\vdash A \vee B}} \qquad \boxed{\dfrac{\vdash B}{\vdash A \vee B}}$$

From "$A \vee B$ is true", "$A \Rightarrow C$" and "$B \Rightarrow C$", we can derive that C is true (*case analysis*).

$$\boxed{\dfrac{\vdash A \vee B \quad \vdash A \Rightarrow C \quad \vdash B \Rightarrow C}{\vdash C}}$$

Universal quantification

If c is an *arbitrary* element of X, and $A(c)$ is true, then $\forall x \in X, A(x)$ is true (*universal generalization*).

Universal quantification

If c is an *arbitrary* element of X, and $A(c)$ is true, then $\forall x \in X, A(x)$ is true (*universal generalization*).

$$\frac{\vdash A(c) \qquad c \in X \text{ is arbitrary}}{\vdash \forall x \in X, A(x)}$$

Universal quantification

If c is an *arbitrary* element of X, and $A(c)$ is true, then $\forall x \in X, A(x)$ is true (*universal generalization*).

$$\frac{\vdash A(c) \qquad c \in X \text{ is arbitrary}}{\vdash \forall x \in X, A(x)}$$

If c is an arbitrary element of X and $\forall x \in X, A(x)$ is true, then $A(c)$ is true (*instantiation*).

Universal quantification

If c is an *arbitrary* element of X, and $A(c)$ is true, then $\forall x \in X, A(x)$ is true (*universal generalization*).

$$\frac{\vdash A(c) \qquad c \in X \text{ is arbitrary}}{\vdash \forall x \in X, A(x)}$$

If c is an arbitrary element of X and $\forall x \in X, A(x)$ is true, then $A(c)$ is true (*instantiation*).

$$\frac{\vdash \forall x \in X, A(x) \qquad c \in X}{\vdash A(c)}$$

Existential quantification

If c is a *particular* element of X, and $A(c)$ is true, then $\exists x \in X, A(x)$ is true (*existential introduction*).

Existential quantification

If c is a *particular* element of X, and $A(c)$ is true, then $\exists x{\in}X, A(x)$ is true (*existential introduction*).

$$\frac{\vdash A(c) \qquad c \in X \text{ is fixed}}{\vdash \exists x{\in}X, A(x)}$$

Existential quantification

If c is a *particular* element of X, and $A(c)$ is true, then $\exists x \in X, A(x)$ is true (*existential introduction*).

$$\frac{\vdash A(c) \qquad c \in X \text{ is fixed}}{\vdash \exists x \in X, A(x)}$$

If $\exists x \in X, A(x)$ is true, and for any $c \in X$, $A(c)$ implies B, then B is true (*generalized case analysis*).

Existential quantification

If c is a *particular* element of X, and $A(c)$ is true, then $\exists x{\in}X, A(x)$ is true (*existential introduction*).

$$\frac{\vdash A(c) \qquad c \in X \text{ is fixed}}{\vdash \exists x{\in}X, A(x)}$$

If $\exists x{\in}X, A(x)$ is true, and for any $c{\in}X$, $A(c)$ implies B, then B is true (*generalized case analysis*).

$$\frac{\vdash \forall x{\in}X, (A(x) \Rightarrow B) \qquad \vdash \exists x{\in}X, A(x)}{\vdash B}$$

Falsehood

Falsehood

Special proposition: **False.**

Falsehood

Special proposition: **False.**

No introduction rule.

Falsehood

Special proposition: **False.**

No introduction rule.

Assuming falsehood, we can derive a proof of any statement.

Falsehood

Special proposition: **False.**

No introduction rule.

Assuming falsehood, we can derive a proof of any statement.

$$\boxed{\frac{\vdash \textbf{False}}{\vdash A}}$$

Falsehood

A is *false* ($\neg A$) if, assuming *A* is true, we can derive the proof of **False**.

$$\boxed{\neg A \stackrel{\text{def}}{=} A \Rightarrow \textbf{False}}$$

Falsehood

Any statement can be proved to be true, assuming a *contradiction* ($A \wedge \neg A$).

$$\frac{\vdash A \wedge \neg A}{\vdash B}$$

Falsehood

Any statement can be proved to be true, assuming a *contradiction* ($A \wedge \neg A$).

$$\frac{\vdash \mathrm{A} \qquad \vdash \neg \mathrm{A}}{\vdash B}$$

Falsehood

Any statement can be proved to be true, assuming a *contradiction* ($A \wedge \neg A$).

$$\frac{\vdash \mathrm{A} \qquad \vdash \mathrm{A} \Rightarrow \textbf{False}}{\vdash B}$$

Falsehood

Any statement can be proved to be true, assuming a *contradiction* ($A \land \neg A$).

$$\frac{\vdash \textbf{False}}{\vdash B}$$

Falsehood

A system of hypotheses and rules (axioms) is *consistent* if no proof of **False** can be derived in it.

Falsehood

A system of hypotheses and rules (axioms) is *consistent* if no proof of **False** can be derived in it.

That, is it does not contain *paradoxes*.

Falsehood

A system of hypotheses and rules (axioms) is *consistent* if no proof of **False** can be derived in it.

That, is it does not contain *paradoxes*.

Example 1: Naïve set theory is inconsistent (*Russel's paradox*).

Falsehood

A system of hypotheses and rules (axioms) is *consistent* if no proof of **False** can be derived in it.

That, is it does not contain *paradoxes*.

Example 1: Naïve set theory is inconsistent (*Russel's paradox*).

Example 2: Zermelo–Fraenkel set theory is consistent (see *axiom schema of specification*).

How do we *check* proofs?

How do we *check* proofs?

By verifying *each* inference step.

This might be difficult

This might be difficult

(but not as difficult as to construct a proof)

This might be difficult

(but not as difficult as to construct a proof)

… but still

Some proofs are too critical

- Hardware correctness
- Software correctness
- Integrity of cryptographic protocols

Some proofs are too large

Some proofs are too large

- *Fermat's Last Theorem* (stated in 1637, proved in 1993)
  - The proof is about 150 pages of handwritten math

Some proofs are too large

- *Fermat's Last Theorem* (stated in 1637, proved in 1993)
  - The proof is about 150 pages of handwritten math
- *Odd order theorem* (stated in 1911, proved in 1962)
  - The proof is about 250 pages of printed text

Some proofs are too large

- *Fermat's Last Theorem* (stated in 1637, proved in 1993)
  - The proof is about 150 pages of handwritten math
- *Odd order theorem* (stated in 1911, proved in 1962)
  - The proof is about 250 pages of printed text
- *Four colour theorem* (proved in 1976)
  - 1936 special cases are discharged via *a program*

Can we use computers to check our proofs?

Can we use computers to check our proofs?

Yes

Can we use computers to check our proofs?

Yes

In fact, this is what
some programmers do every day.

Programs

Programs

=

Data Types + Functions

Programming Languages

- Haskell
- ML
- F#
- Lisp
- Scala
- Agda
- Coq
- Perl
- Python
- Ruby
- C++
- Java
- C#
- …

Functional Programming Languages

- Haskell
- ML
- F#
- Lisp
- Scala
- Agda
- Coq

Functional Programming Languages

- Haskell
- ML
- F#
- Lisp
- Scala
- Agda
- Coq

- Data types define *immutable* values
- Functions are *values* as well
- Programs are *pure* functions

Statically-typed Functional Programming Languages

- Haskell
- ML
- F#
- Scala
- Agda
- Coq

Statically-typed Functional Programming Languages

- Haskell
- ML
- F#
- Scala
- Agda
- Coq

- Every value has a *type*
- *Type* defines a *set* of values
- Type of a program — its specification

A type with one element

```
Datatype unit := tt.
```

A type with one element

```
Datatype unit := tt.
```

$$unit \stackrel{\text{def}}{=} \{\ tt\ \}$$

A type with two elements

```
Datatype bool := true | false.
```

A type with two elements

```
Datatype bool := true | false.
```

$$\mathit{bool} \stackrel{\text{def}}{=} \{\, \mathit{true}, \mathit{false} \,\}$$

A type with no elements

```
Datatype empty := .
```

A type with no elements

```
Datatype empty := .
```

$$\mathit{empty} \stackrel{\text{def}}{=} \{\ \}$$

A type defined recursively

```
Datatype nat := 0 | .+1 of nat.
```

A type defined recursively

$\boxed{\texttt{n} \in \texttt{nat} \Rightarrow \texttt{n.+1} \in \texttt{nat}}$

```
Datatype nat := 0 | .+1 of nat.
```

A type defined recursively

$\boxed{\texttt{n} \in \texttt{nat} \Rightarrow \texttt{n.+1} \in \texttt{nat}}$

$$\texttt{\textbf{Datatype} nat := 0 |} \overbrace{\texttt{.+1 \textbf{of} nat.}}$$

$$nat \stackrel{\text{def}}{=} \{\, 0, (0.+1), (0.+1.+1), \ldots \,\}$$

A type defined recursively

$$\boxed{\texttt{n} \in \texttt{nat} \Rightarrow \texttt{n.+1} \in \texttt{nat}}$$

```
Datatype nat := 0 | .+1 of nat.
```

$$nat \stackrel{\text{def}}{=} \{\, 0, \underbrace{(0.{+}1)}_{\mathbf{1}}, \underbrace{(0.{+}1.{+}1)}_{\mathbf{2}}, \ldots \,\}$$

A parametrized type

```
Datatype prod A B := pair of A & B
```

A parametrized type

```
Datatype prod A B := pair of A & B
```

$$A \times B \stackrel{\text{def}}{=} \{\ (a, b) \mid a \in A, b \in B\ \}$$

Another parametrized type

```
Datatype sum A B := inl of A | inr of B
```

Another parametrized type

```
Datatype sum A B := inl of A | inr of B
```

$$A + B \stackrel{\text{def}}{=} \{\ (a, 1)\ |\ a \in A\ \} \cup \{\ (b, 2)\ |\ b \in B\ \}$$

Parametrized recursive type

```
Datatype list A := Nil | Cons of A & (list A).
```

Parametrized recursive type

```
Datatype list A := Nil | Cons of A & (list A).
```

$$\mathit{list}\ A \stackrel{\text{def}}{=} \mathit{Nil} \cup \{\ \mathit{Cons}(\mathrm{a}, \mathrm{l}) \mid \mathrm{l} \in \mathit{list}\ A\ \}$$

A simple function

```
Function negate : bool -> bool :=
  fun b => match b with
    | true  => false
    | false => true
  end.
```

A simple function

negate ∈ bool → bool

```
Function negate : bool -> bool :=
  fun b => match b with
    | true  => false
    | false => true
  end.
```

A recursive function

```
Function even : nat -> bool :=
  fun n => match n with
    | n'.+1 => negate (even n')
    | 0     => true
  end.
```

A recursive function

$\boxed{\texttt{even} \in \texttt{nat} \to \texttt{bool}}$

```
Function even : nat -> bool :=
  fun n => match n with
    | n'.+1 => negate (even n')
    | 0     => true
  end.
```

An ill-typed function

```
Function even : nat -> bool :=
  fun n => match n with
    | n'.+1 => negate (even n')
    | 0     => Nil
  end.
```

An ill-typed function

```
Function even : nat -> bool :=
  fun n => match n with
    | n'.+1 => negate (even n')
    | 0     => Nil
  end.
```

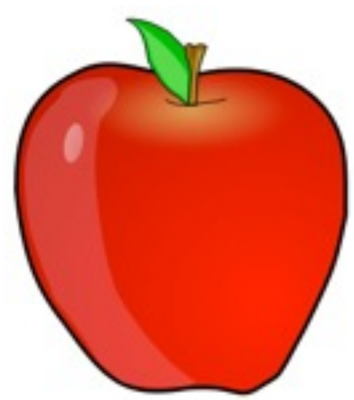
```
Wrong type: bool expected, but list ? found.
```

Types are program specifications

A type-checking algorithm ensures that each *value* has an appropriate *type*, i.e., that it belongs to the *corresponding set.*

Types are program specifications

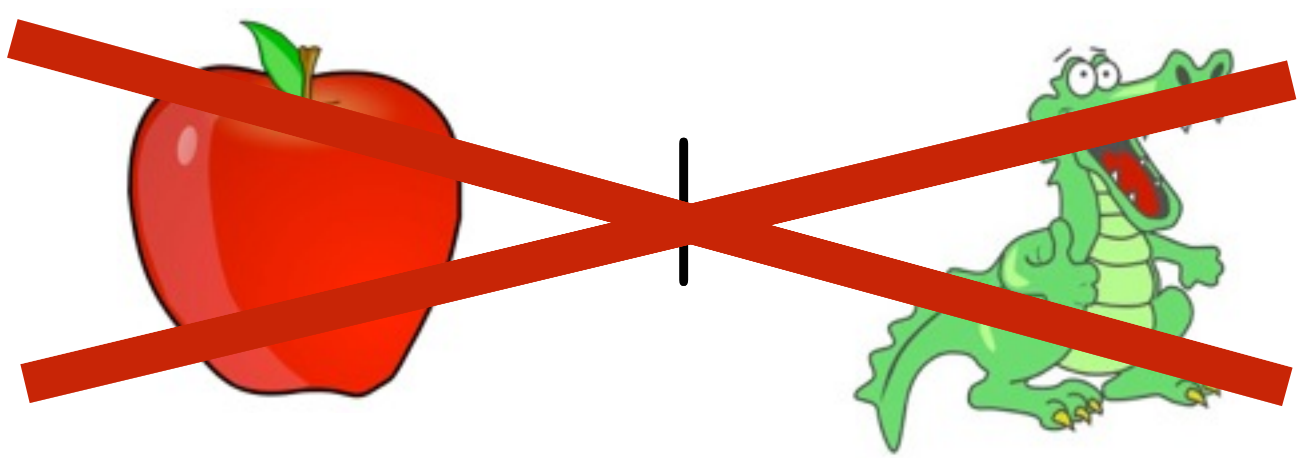
A type-checking algorithm ensures that each *value* has an appropriate *type*, i.e., that it belongs to the *corresponding set.*



+

Types are program specifications

A type-checking algorithm ensures that each *value* has an appropriate *type*, i.e., that it belongs to the *corresponding set.*



Given a type **A**, can we construct a program (value) **p**, such that **p** is an element of type **A**?

Given a type **A**, can we construct a program (value) **p**, such that **p** is an element of type **A**?

In other words: can we *inhabit* the type **A**?

unit

```
unit
```

```
tt
```

bool

```
bool
```

```
true
```

bool

false

nat

`nat`

0

nat

1

nat

2014

empty

empty

???

```
A -> A
```

```
A -> A
```

```
fun (a: A) => a
```

```
A -> A
```

```
fun (a: A) => a
```

$$\frac{A \vdash A}{\vdash A \Rightarrow A}$$

```
A -> (A -> B) -> B
```

```
A -> (A -> B) -> B
```

```
fun (a: A)
    (f : A -> B) => f a
```

```
A -> (A -> B) -> B
```

```
fun (a: A)
    (f : A -> B) => f a
```

$$\frac{\vdash A \qquad \vdash A \Rightarrow B}{\vdash B}$$

```
(A × B) -> A
```

$$(A \times B) \to A$$

```
fun (a: A × B) =>
  match a with
  | pair a b => a
  end
```

```
(A × B) -> A
```

```
fun (a: A × B) =>
  match a with
  | pair a b => a
  end
```

$$\frac{\vdash A \land B}{\vdash A}$$

```
(A × B) -> B
```

```
(A × B) -> B
```

```
fun (a: A × B) =>
  match a with
  | pair a b => b
  end
```

```
(A × B) -> B
```

```
fun (a: A × B) =>
  match a with
  | pair a b => b
  end
```

$$\frac{\vdash A \wedge B}{\vdash B}$$

```
A -> B -> (A × B)
```

```
A -> B -> (A × B)
```

```
fun (a: A)(b: B) => pair a b
```

```
A -> B -> (A × B)
```

```
fun (a: A)(b: B) => pair a b
```

$$\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}$$

```
A -> (A × B)
```

$$A \to (A \times B)$$

???

```
A -> A + B
```

```
A -> A + B
```

```
fun (a: A) => inl a
```

```
A -> A + B
```

```
fun (a: A) => inl a
```

$$\frac{\vdash A}{\vdash A \vee B}$$

```
(A + B) -> (A -> C) -> (B -> C) -> C
```

```
(A + B) -> (A -> C) -> (B -> C) -> C
```

```
fun (x: A + B)(f: A -> B)(g: B -> C) =>
  match x with
  | inl a => f a
  | inr b => g b
  end
```

```
(A + B) -> (A -> C) -> (B -> C) -> C
```

```
fun (x: A + B)(f: A -> B)(g: B -> C) =>
   match x with
   | inl a => f a
   | inr b => g b
   end
```

$$\frac{\vdash A \vee B \qquad \vdash A \Rightarrow C \qquad \vdash B \Rightarrow C}{\vdash C}$$

```
empty -> A
```

```
empty -> A
```

```
fun (x: empty) => match x with end
```

```
empty -> A
```

```
fun (x: empty) => match x with end
```

$$\frac{\vdash \mathbf{False}}{\vdash A}$$

To show that a *type* **A** is inhabited, it is sufficient to construct a *program* **p : A** using *datatype constructors*, *case-analysis* and *function application*.

To show that a *type* **A** is inhabited, it is sufficient to construct a *program* **p : A** using *datatype constructors*, *case-analysis* and *function application*.

To prove a *proposition* **A** it is sufficient to construct a *proof* **p** of **A** using *assumptions*, *axioms* and *derived inference rules*.

Curry-Howard correspondence

Curry-Howard correspondence

Propositions = Types

Curry-Howard correspondence

Propositions = Types

Proofs = Programs

Curry-Howard correspondence

Axioms are
datatype constructors

Curry-Howard correspondence

Axioms are
datatype constructors

Inference rules are
functions

Curry-Howard correspondence

Curry-Howard correspondence

True	`unit`

Curry-Howard correspondence

True	`unit`
False	`empty`

Curry-Howard correspondence

True	`unit`
False	`empty`
$A \wedge B$	`A × B`

Curry-Howard correspondence

True	`unit`
False	`empty`
$A \wedge B$	`A × B`
$A \vee B$	`A + B`

Curry-Howard correspondence

True	`unit`
False	`empty`
$A \wedge B$	`A × B`
$A \vee B$	`A + B`
$A \Rightarrow B$	`A → B`

Curry-Howard correspondence

Curry-Howard correspondence

modus ponens	function application

Curry-Howard correspondence

modus ponens	function application
a hypothesis	function argument

Curry-Howard correspondence

modus ponens	function application
a hypothesis	function argument
introduction rule	datatype constructor

Curry-Howard correspondence

modus ponens	function application
a hypothesis	function argument
introduction rule	datatype constructor
elimination rule	pattern matching

Programs-as-proofs are *constructive*

A program of type **A → B**,
taken as a proof, specifies how to
derive a proof of its result type **B**
from the proof of **A** *algorithmically*.

Programs-as-proofs are *constructive*

A program of type **A → B**,
taken as a proof, specifies how to
derive a proof of its result type **B**
from the proof of **A** *algorithmically*.

This is not always the case
in the *classical logic*.

Non-constructive Axioms

Non-constructive Axioms

- Excluded middle: $A \vee \neg A$
  - Neither a proof of A, nor of $\neg A$ is required;

Non-constructive Axioms

- Excluded middle: $A \vee \neg A$
  - Neither a proof of *A*, nor of $\neg A$ is required;
- Double negation: $((A \Rightarrow \textbf{\textit{False}}) \Rightarrow \textbf{\textit{False}}) \Rightarrow A$
  - Again, the proof of *A* is not required.

Non-constructive Axioms

- Excluded middle: $A \vee \neg A$
  - Neither a proof of A, nor of $\neg A$ is required;
- Double negation: $((A \Rightarrow \textbf{\textit{False}}) \Rightarrow \textbf{\textit{False}}) \Rightarrow A$
  - Again, the proof of A is not required.

Assuming these axioms keeps Curry-Howard correspondence *consistent* as a formal system.

What about quantifiers?

Statically-typed functional programming languages

- Haskell
- ML
- F#
- Scala
- Agda
- Coq

- Every value has a *type*
- *Type* defines a *set* of values
- Type of a program — its specification

Dependently-typed functional programming languages

- Agda
- Coq

- Every value has a *type*
- *Type* defines a *set* of values
- Type of a program — its specification

Dependently-typed functional programming languages

- Agda
- Coq

- Every value has a *type*
- *Type* defines a *set* of values
- Type of a program — its specification
- Types can *depend* on values

Function Type

```
A -> B
```

Function Type

```
A -> B

bool -> nat
```

Function Type

```
A -> B
```

```
bool -> nat
```

```
fun b =>
  match b with
  | true  => 0
  | false => 1
  end
```

Dependent Function Type

```
Π(x: A). B(x)
```

Dependent Function Type

```
Π(x: A). B(x)
```

```
Π(b: bool). if b then nat else unit
```

Dependent Function Type

```
Π(x: A). B(x)
```

```
Π(b: bool). if b then nat else unit
```

```
fun b =>
  match b with
  | true  => 0
  | false => tt
  end
```

Pair Type

`A × B`

Pair Type

$$A \times B$$

```
Datatype prod A B := pair of A & B
```

Dependent Pair Type

`Σ(x: A). P(x)`

Dependent Pair Type

```
Σ(x: A). P(x)
```

```
Datatype sigma A (P: A -> B):=
      ex (x: A) of P x
```

Dependent Pair Type

$$\Sigma(x: A).\ P(x)$$

```
Datatype sigma A (P: A -> B):=
      ex (x: A) of P x
```

Every value of type `sigma A P` contains
a value x of type A (*witness*)
and a value of type P(x).

Curry-Howard correspondence

Curry-Howard correspondence

$\forall x \in A, P(x)$	$\Pi(x{:}A).\ P(x)$

Curry-Howard correspondence

$\forall x \in A, P(x)$	$\Pi(x{:}A).\ P(x)$
$\exists x \in A, P(x)$	$\Sigma(x{:}A).\ P(x)$

An example of dependent type

```
Π(P : nat -> Prop).
  P(0) ->
 (Π(n : nat). P(n) -> P(n.+1)) ->
  Π(n : nat). P n
```

An example of dependent type

```
∀(P ∈ nat ⇒ Prop).
  P(0) ⇒
 (∀(n ∈ nat). P(n) ⇒ P(n + 1)) ⇒
  ∀(n ∈ nat). P n
```

An example of dependent type

```
∀(P ∈ nat ⇒ Prop).
  P(0) ⇒
 (∀(n ∈ nat). P(n) ⇒ P(n + 1)) ⇒
  ∀(n ∈ nat). P n
```

```
Function nat_ind (P : nat -> Prop)
                 (f0 : P 0)
                 (fn : Π(n : nat), P n -> P (n.+1)) :=
 fun (n : nat) =>
   match n with
    | 0 => f0
    | n'.+1 => fn n' (nat_ind P f0 fn n0)
   end.
```

Dependently-typed functional programming languages

- Agda
- Coq

Proof Assistants

- Agda
- Coq

Proof Assistants

- Agda
- Coq

- *Not* Turing-complete
  - type-checking should terminate
- Proofs can be built interactively using *scripts*
- “Boring” proofs can be automated

Interactive proof construction

(in Coq)

Interactive proof construction

(in Coq)

```
Theorem counterexample (A: Type) (P: A → Prop) :
   (∃x: A, ¬P x) → ¬(∀ x, P x).
```

Interactive proof construction

(in Coq)

```
Theorem counterexample (A: Type) (P: A → Prop) :
    (∃x: A, ¬P x) → ¬(∀ x, P x).
Proof.
 case => x H1 H2.
 by apply : H1 (H2 x).
Qed.
```

Interactive proof construction

(in Coq)

```
Function counterexample :=
  fun (A : Type) (P : A -> Prop) (hyp : ∃x : A, ¬ P x) =>
  (fun F : ∀(x : A) (p : (fun x0 : A => ¬ P x0) x),
              (fun _ : (∃x0 : A, ¬ P x0) => ¬ (∀x0 : A, P x0))
                (ex_intro (fun x0 : A => ¬ P x0) x p) =>
   match hyp as e
     return ((fun _ : (∃x : A, ¬ P x) => ¬ (∀x : A, P x)) e)
   with
   | ex x x0 => F x x0
   end) (fun (x : A) (H1 : ¬ P x) (H2 : ∀x0 : A, P x0) => H1 (H2 x)).
```

Current advances

- *Four colour theorem* — mechanised in Coq in 2005
- *CompCert* — fully verified C compiler (2006)
- *Odd order theorem* — mechanised in Coq in 2013
- *Keppler's conjecture* — formally proved in 2014
- *sel4* — formally verified OS kernel (2014)

To take away

- Formal *proofs* are functional *programs* in disguise;
- *propositions* are program *types*;
- writing a proof = constructing a program;
- proving is still *tricky* and takes a mathematician’s insight;
- proof assistants help to automate the “boring” parts of mechanised formal proof constructions…
- … and the rest is a huge fun.

“A mathematician is expected to sit at his computer and think.”

Hari Seldon