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: $A \vdash A$
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:

\[
\frac{A \vdash B}{\vdash A \Rightarrow B}
\]
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.

\[
\vdash A \quad \vdash A \Rightarrow B \\
\hline
\vdash B
\]
Conjunction introduction/elimination

From “A is true” and “B is true”, we can derive that $A \land B$ is true as well.
Conjunction introduction/elimination

From “A is true” and “B is true”, we can derive that $A \land B$ is true as well.

\[ \vdash A \quad \vdash B \quad \quad \quad \vdash A \land B \]
Conjunction introduction/elimination

From “$A$ is true” and “$B$ is true”, we can derive that $A \land B$ is true as well.

\[
\begin{array}{c}
\vdash A \\
\vdash B \\
\hline
\vdash A \land B
\end{array}
\]

From “$A \land B$ 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 \( A \land B \) is true as well.

\[ \frac{\vdash A \quad \vdash B}{\vdash A \land B} \]

From “\( A \land B \) is true” we can derive that \( A \) is true and that \( B \) is true.

\[ \frac{\vdash A \land B}{\vdash A} \]
\[ \frac{\vdash A \land B}{\vdash B} \]
Disjunction introduction/elimination

From “A is true” or “B is true”, we can derive that $A \lor B$ is true.
Disjunction introduction/elimination

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

\[
\begin{align*}
\dashv A \\
\hline \\
\dashv A \lor B
\end{align*}
\]

\[
\begin{align*}
\dashv B \\
\hline \\
\dashv A \lor B
\end{align*}
\]
Disjunction introduction/elimination

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

\[
\begin{array}{c}
\vdash A \\
\hline
\vdash A \lor B \\
\end{array}
\quad
\begin{array}{c}
\vdash B \\
\hline
\vdash A \lor B \\
\end{array}
\]

From “$A \lor B$ 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 $A \lor B$ is true.

\[
\begin{align*}
&\vdash A \\
\hline
&\vdash A \lor B
\end{align*}
\]

\[
\begin{align*}
&\vdash B \\
\hline
&\vdash A \lor B
\end{align*}
\]

From “$A \lor B$ is true”, “$A \Rightarrow C$” and “$B \Rightarrow C$”, we can derive that C is true (case analysis).

\[
\begin{align*}
&\vdash A \lor B \quad \vdash A \Rightarrow C \quad \vdash B \Rightarrow C \\
\hline
&\vdash C
\end{align*}
\]
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).

\[
\vdash A(c) \quad 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).

\[
\because A(c) \quad c \in X \text{ is arbitrary} \\
\therefore \forall x \in X, A(x)
\]

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).

\[
\begin{array}{c}
\vdash A(c) \quad \text{\( c \in X \) is arbitrary} \\
\hline
\vdash \forall x \in X, A(x)
\end{array}
\]

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

\[
\begin{array}{c}
\vdash \forall x \in X, A(x) \quad \text{\( c \in X \)} \\
\hline
\vdash A(c)
\end{array}
\]
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).

\[
\begin{array}{c}
\vdash A(c) \quad c \in X \text{ is fixed}
\\
\hline
\vdash \exists x \in X, A(x)
\end{array}
\]
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).

\[
\begin{array}{c}
\vdash A(c) \quad c \in X \text{ is fixed} \\
\hline
\vdash \exists x \in X, A(x)
\end{array}
\]

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).

\[
\begin{align*}
\vdash A(c) & \quad c \in X \text{ is fixed} \\
\hline
\vdash \exists x \in X, A(x)
\end{align*}
\]

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).

\[
\begin{align*}
\vdash \forall x \in X, (A(x) \Rightarrow B) & \quad \vdash \exists x \in X, A(x) \\
\hline
\vdash B
\end{align*}
\]
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.

\[ \vdash \text{False} \]

\[ \vdash \text{A} \]
A is false (\(\neg A\)) if, assuming A is true, we can derive the proof of False.

\[\neg A \overset{\text{def}}{=} A \Rightarrow \text{False}\]
Falsehood

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

\[
\begin{align*}
\vdash A \land \neg A \\
\hline
\vdash B
\end{align*}
\]
Falsehood

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

\[
\begin{align*}
\neg A & \quad \neg A \\
\hline
\vdash B
\end{align*}
\]
Falsehood

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

\[
\vdash A \quad \vdash A \Rightarrow \text{False} \\
\hline
\vdash B
\]
Falsehood

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

\[ \vdash \text{False} \]

\[ \vdash B \]
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

\textbf{Datatype} unit := \texttt{tt}.
A type with one element

\[\text{Datatype } \text{unit} := \text{tt}.\]

\[\text{unit} \overset{\text{def}}{=} \{ \text{tt} \}\]
A type with two elements

**Datatype** `bool := true | false`.
A type with two elements

Datatype bool := true | false.

bool def { true, false }
A type with no elements

Datatype empty := .
A type with no elements

\textbf{Datatype} \texttt{empty} := .

\texttt{empty} \overset{\text{def}}{=} \{ \}
A type defined recursively

\textbf{Datatype} \texttt{nat} := 0 \mid .+1 \textbf{of} \texttt{nat}.
A type defined recursively

\[
\begin{align*}
\text{Datatyp}e \ \text{nat} & := \ 0 \mid +1 \ \text{of} \ \text{nat}. \\
\end{align*}
\]

\[
\begin{align*}
n \in \text{nat} \Rightarrow n.+1 \in \text{nat}
\end{align*}
\]
A type defined recursively

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

```
n ∈ nat ⇒ n.+1 ∈ nat
```

```
nat = \{ 0, (0.+1), (0.+1.+1), ... \}
```
A type defined recursively

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

\[ n \in \text{nat} \Rightarrow n.+1 \in \text{nat} \]

\[ \text{Datatype} \quad \text{nat} := 0 \mid .+1 \text{ of nat.} \]

\[ \text{nat} \overset{\text{def}}{=} \{ 0, (0.+1), (0.+1.+1), \ldots \} \]

\[ \begin{array}{c}
1 \\
2
\end{array} \]
A parametrized type

Datatype prod A B := pair of A & B
A parametrized type

\textbf{Datatype} \ prod \ A \ B \ := \ \text{pair of} \ A \ \& \ B

\[ A \times B \overset{\text{def}}{=} \{ (a, b) \mid a \in A, b \in B \} \]
Another parametrized type

**Datatype** sum A B := inl of A | inr of B
Another parametrized type

**Datatype** \( \text{sum} \ A \ B := \text{inl} \ \text{of} \ A \mid \text{inr} \ \text{of} \ B \)

\[
A + B \overset{\text{def}}{=} \left\{ (a, 1) \mid a \in A \right\} \cup \left\{ (b, 2) \mid b \in B \right\}
\]
Parametrized recursive type

Datatype list A := Nil | Cons of A & (list A).
Parametrized recursive type

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

list A ndef = Nil ∪ { Cons(a, l) | l ∈ list A }
A simple function

Function negate : bool -> bool :=

fun b => match b with
  | true  => false
  | false => true
end.
A simple function

\[\text{negate} \in \text{bool} \rightarrow \text{bool}\]

Function negate : bool -> bool :=
\[
\begin{array}{c}
\text{fun } b \Rightarrow \text{match } b \text{ with} \\
| \text{true } \Rightarrow \text{false} \\
| \text{false } \Rightarrow \text{true}
\end{array}
\]
end.
A recursive function

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

\[
\text{even} \in \text{nat} \rightarrow \text{bool}
\]

\[
\begin{align*}
\text{Function} & \text{ even : nat -> bool :=} \\
& \quad \text{fun n => match n with} \\
& \quad \mid n'.+1 \Rightarrow \text{negate (even n')} \\
& \quad \mid 0 \Rightarrow \text{true} \\
& \text{end.}
\end{align*}
\]
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 \textit{value} has an appropriate \textit{type}, i.e., that it belongs to the \textit{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
bool
bool

true
bool

false
nat
nat

0
nat

1
nat

2014
empty
empty

???
A → A
\[ A \rightarrow A \]

\[
\text{fun} \ (a: A) \Rightarrow a
\]
fun (a: A) => a
A -> (A -> B) -> B
A → (A → B) → B

fun (a: A) (f : A → B) => f a
\[ A \rightarrow (A \rightarrow B) \rightarrow B \]

**fun** (a: A)

(f : A \rightarrow B) => f a

\[ \vdash A \]

\[ \vdash A \Rightarrow B \]

\[ \begin{array}{c}
\vdash A \\
\hline
\vdash A \Rightarrow B \\
\hline
\vdash B
\end{array} \]
\[(A \times B) \rightarrow A\]
\[(A \times B) \rightarrow A\]

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

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

⊢ A ∧ B

⊢ A

⊢ A
(A \times B) \rightarrow B
\[(A \times B) \rightarrow B\]

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

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

⊢ A ∧ B

⊢ B
A \rightarrow B \rightarrow (A \times B)
A \rightarrow B \rightarrow (A \times B)

\textbf{fun} \ (a: A)(b: B) \Rightarrow \text{pair} \ a \ b
A \rightarrow B \rightarrow (A \times B)

\textbf{fun} \ (a: A)(b: B) \Rightarrow \text{pair} \ a \ b
A → (A × B)
A \rightarrow (A \times B)

???
A \rightarrow A + B
\[
A \rightarrow A + B
\]

fun (a: A) => inl a
A \rightarrow A + B

\textbf{fun} \ (a: A) \Rightarrow \text{inl} \ a
\[(A + B) \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C\]
\[(A + B) \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C\]

```ocaml
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

\[
\begin{align*}
\Gamma & : A \lor B \\
\Delta & : A \Rightarrow C \\
\Theta & : B \Rightarrow C
\end{align*}
\]
\[
\therefore C
\]
empty -> A
empty  ->  A

fun (x: empty) => match x with end
empty \rightarrow A

\text{fun } (x: \text{ empty}) \Rightarrow \text{match } x \text{ with end}

\vdash \text{False}

\hline
\vdash \text{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

<p>| True   | unit   |</p>
<table>
<thead>
<tr>
<th>True</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>empty</td>
</tr>
</tbody>
</table>
Curry-Howard correspondence

<table>
<thead>
<tr>
<th>True</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>empty</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>$A \times B$</td>
</tr>
</tbody>
</table>
Curry-Howard correspondence

<table>
<thead>
<tr>
<th>True</th>
<th>unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>empty</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>$A \times B$</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>$A + B$</td>
</tr>
<tr>
<td>Curry-Howard correspondence</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>unit</td>
</tr>
<tr>
<td>False</td>
<td>empty</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>$A \times B$</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>$A + B$</td>
</tr>
<tr>
<td>$A \Rightarrow B$</td>
<td>$A \rightarrow B$</td>
</tr>
</tbody>
</table>
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
a hypothesis
introduction rule

function application
function argument
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 $A \to B$, 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 $A \rightarrow B$, 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: $A \lor \neg A$
- Neither a proof of $A$, nor of $\neg A$ is required;
Non-constructive Axioms

• Excluded middle: \( A \lor \neg A \)
  • Neither a proof of \( A \), nor of \( \neg A \) is required;

• Double negation: ((\( A \Rightarrow \text{False} \)) \( \Rightarrow \text{False} \)) \( \Rightarrow A \)
  • Again, the proof of \( A \) is not required.
Non-constructive Axioms

- Excluded middle: $A \lor \neg A$
  - Neither a proof of $A$, nor of $\neg A$ is required;
- Double negation: $((A \Rightarrow \text{False}) \Rightarrow \text{False}) \Rightarrow A$
  - 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

\[ A \rightarrow B \]

\[ \text{bool} \rightarrow \text{nat} \]
Function Type

\[ A \rightarrow B \]

\[ \text{bool} \rightarrow \text{nat} \]

\[
\text{fun } b \Rightarrow \\
\text{match } b \text{ with} \\
| \text{true} \Rightarrow 0 \\
| \text{false} \Rightarrow 1 \\
\text{end}
\]
Dependent Function Type

\( \Pi(x : A) \cdot B(x) \)
Dependent Function Type

\[ \Pi(x : A). \ B(x) \]

\[ \Pi(b : \text{bool}). \ \text{if } b \ \text{then} \ \text{nat} \ \text{else} \ \text{unit} \]
Dependent Function Type

\[
\Pi(x : A). \ B(x)
\]

\[
\Pi(b : bool). \ if \ b \ then \ nat \ else \ unit
\]

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

\[ \Sigma(x : A). \ P(x) \]
Dependent Pair Type

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

**Datatype** $\text{sigma} \ A \ (P: A \rightarrow B):= \text{ex} \ (x: A) \ \text{of} \ P \ x$
Dependent Pair Type

\[ \Sigma(x: A). \, P(x) \]

**Datatype** \( \sigma A \) (\( P: A \rightarrow B \)) :=
\[ \text{ex } (x: A) \, \text{of} \, P \, x \]

Every value of type \( \sigma A \) \( P \) contains a value \( x \) of type \( A \) (\emph{witness}) and a value of type \( P(x) \).
Curry-Howard correspondence
Curry-Howard correspondence

\[ \forall x \in A, P(x) \quad \text{and} \quad \Pi(x:A). P(x) \]
Curry-Howard correspondence

\[ \forall x \in A, P(x) \quad \Pi(x:A). P(x) \]

\[ \exists x \in A, P(x) \quad \Sigma(x:A). P(x) \]
An example of dependent type

\[ \Pi (P : \text{nat} \rightarrow \text{Prop}). \]
\[ P(0) \rightarrow \]
\[ (\Pi (n : \text{nat}). P(n) \rightarrow P(n.+1)) \rightarrow \]
\[ \Pi (n : \text{nat}). P\ n \]
An example of dependent type

\[ \forall (P \in \text{nat} \Rightarrow \text{Prop}). \]

\[ P(0) \Rightarrow \]

\[ (\forall (n \in \text{nat}). \ P(n) \Rightarrow P(n + 1)) \Rightarrow \]

\[ \forall (n \in \text{nat}). \ P \ n \]
An example of dependent type

\( \forall (P \in \text{nat} \Rightarrow \text{Prop}). \)
\( P(0) \Rightarrow \)
\( (\forall (n \in \text{nat}). P(n) \Rightarrow P(n + 1)) \Rightarrow \)
\( \forall (n \in \text{nat}). P n \)

**Function** `nat_ind` (\( P : \text{nat} \rightarrow \text{Prop} \))
\( (f0 : P 0) \)
\( (fn : \prod (n : \text{nat}), P n \rightarrow P (n.+1)) :\)
\[ \text{fun } (n : \text{nat}) \Rightarrow \]
\[ \text{match } n \text{ with} \]
\[ | 0 \Rightarrow f0 \]
\[ | n'.+1 \Rightarrow fn n' (\text{nat}_{\text{ind}} P f0 fn n0) \]
\[ \text{end}. \]
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 : \exists x : A, ¬ P x) =>
    (fun F : \forall(x : A) (p : (fun x0 : A => ¬ P x0) x),
      (fun _ : (\exists x0 : A, ¬ P x0) => ¬ (\forall x0 : A, P x0))
        (ex_intro (fun x0 : A => ¬ P x0) x p) =>
          match hyp as e
            return ((fun _ : (\exists x : A, ¬ P x) => ¬ (\forall x : A, P x)) e)
          with
            | ex x x0 => F x x0
          end) (fun (x : A) (H1 : ¬ P x) (H2 : \forall 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