Bertrand Russell shows that naive set theory (or type theory) is inconsistent:

\{x \mid x \not\in x\} \in \{x \mid x \not\in x\}

Hilbert: formalism, leads eventually to ZFC set theory

Russell: logicism, leads eventually to an early version of type theory

Brouwer, Heyting, Bishop: intuitionism, rejects excluded middle
Brouwer–Heyting–Kolmogorov interpretation

\[ \bot \] does not exist

\[ A \rightarrow B \] maps proofs of \( A \) to proofs of \( B \)

\[ A \land B \] proof of \( A \) and proof of \( B \)

\[ A \lor B \] proof of \( A \) or a proof of \( B \)

\[ \forall x. P(x) \] maps \( x \) to a proof of \( P(x) \)

\[ \exists x. P(x) \] object \( a \) with proof of \( P(a) \)

proof of existence corresponds to constructing an example

program extraction rough idea

an intuitionistic (constructive) proof

corresponds to an executable algorithm

constructive functional programming

program specification: example

• program specification

• constructive proof of existence

• automatically generated functional program

the correctness proof of the specification

\[ \forall l : \text{natlist}. \exists l' : \text{natlist}. \text{permutation}(l, l') \land \text{sorted}(l') \]

yields a program (function) from natlist to natlist
∀x : A. P(x) → ∃y : B. Q(x, y)

- A  input type
- B  output type
- P(x) precondition
- Q(x, y) input/output behaviour

The correctness proof yields a program from A to B.

Program extraction in Coq

Coq proof in type theory gives a functional program in OCaml or Haskell or Scheme.

Program extraction in Coq

Is "almost" the identity function but

- other typing system
- information from Prop is erased

Existential quantification in Prop

Inductive type:

\[
\text{Inductive } \text{ex} (A : \text{Type}) (P : A \to \text{Prop}) : \text{Prop} := \\
\text{ex_intro : forall } x : A, P x \to \text{ex } P
\]

Syntax:

\[
\exists x : A, P x.
\]
existential quantification in Set

inductive type:

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=
  exist : forall x : A, P x -> sig P
```

syntax:

```
{x:A | P x}
```

successor: existence proof and extracted program

specification:

```
Theorem successor :
  forall n:nat, {m:nat | m = S n}.
```

extracted program:

```
let successor n =
  S n
```

predecessor: existence proof and extracted program

specification:

```
Theorem predecessor :
  forall n:nat, ~(n = 0) -> {m:nat | S m = n}.
```

extracted program:

```
let rec predecessor = function
  | O -> assert false (* absurd case *)
  | S n0 -> n0
```
Theorem Sort :
  \forall l : \text{natlist},
  \{ l' : \text{natlist} \mid \text{permutation} \ l \ l' \land \text{sorted} \ l' \}.

Inductive permutation : \text{natlist} -> \text{natlist} -> \text{Prop} :=
| permutation_nil : permutation \ \text{nil} \ \text{nil}
| permutation_cons :
  \forall (n : \text{nat}) (l \ l' \ l'' : \text{natlist}),
  \text{permutation} l l' ->
  \text{inserted} n l' l'' ->
  \text{permutation} (\text{cons} n l) l''.

Inductive inserted (n : \text{nat}) :
  \text{natlist} -> \text{natlist} -> \text{Prop} :=
| inserted_front :
  \forall l : \text{natlist}, \text{inserted} n l (\text{cons} n l)
| inserted_cons :
  \forall (m : \text{natt}) (l \ l' : \text{nattlist}),
  \text{inserted} n l l' ->
  \text{inserted} n (\text{cons} m l) (\text{cons} m l').

Inductive le (n:\text{natt}) : \text{nat} -> \text{Prop} :=
| le_n : le n n
| le_S : \forall m : \text{natt}, le n m \rightarrow le n (S m).

le_ind :
  \forall (n : \text{natt}) (P : \text{nat} -> \text{Prop}),
  P \ n ->
  (\forall m : \text{natt}, le n m \rightarrow P m \rightarrow P (S m)) ->
  \forall n0 : \text{natt}, le n n0 \rightarrow P n0.
**Leibniz equality**

Two terms are equal if they have the same properties

```coq
Inductive eq (A : Type) (x : A) : A -> Prop :=
  refl_equal : x = x

eq_ind
  : forall (A : Type) (x : A) (P : A -> Prop),
  P x -> forall y : A, x = y -> P y
```

**verified programs: two approaches**

- correctness proofs from program to proof
- program extraction from proof to program

---

**insertion sort: predicate sorted**

```coq
Inductive sorted : natlist -> Prop :=
  | sorted0 : sorted nil
  | sorted1 : forall n:nat , sorted (cons n nil)
  | sorted2 : forall n h:nat , forall t:natlist,
             le n h ->
             sorted (cons h t) ->
             sorted (cons n (cons h t)).
```
correctness proofs: Hoare logic

imperative program
⇝
annotated imperative program
⇝
proof obligations

mirror: correctness proof

define a function mirror
and prove its correctness:

Theorem Mirrored_mirror :
  forall t : bintree,
  Mirrored t (mirror t).

mirror: program extraction

prove the specification correct
and extract a program from it

Theorem Mirror :
 forall t : bintree,
  {t' : bintree | Mirrored t t'}.

summarizing the two approaches

• specification
  Inductive Mirrored

• approach 1: implementation
  Fixpoint mirror

• approach 1: correctness
  Theorem Mirrored_mirror

• approach 2: program extracted from existence proof
  Theorem Mirror
logics and type theory

1st-order minimal propositional logic $\leftrightarrow$
simple type theory

1st-order minimal predicate logic $\leftrightarrow$
dependent type theory

2nd-order minimal propositional logic $\leftrightarrow$
polymorphic type theory

formulas of prop1 (already seen)

$\begin{align*}
a & b & c & p & q \\
A & \rightarrow & B \\
\bot & \\
T & \\
A & \land & B \\
A & \lor & B
\end{align*}$

formulas of pred1 (already seen)

(Using terms)

$\begin{align*}
a(...) & b(...) & c(...) & p(...) & q(...) \\
A & \rightarrow & B \\
\forall x. A \\
\bot & \\
T & \\
A & \land & B \\
A & \lor & B \\
\exists x. A
\end{align*}$

formulas of prop2 (new)

$\begin{align*}
a & b & c & p & q \\
A & \rightarrow & B \\
\bot & \\
T & \\
A & \land & B \\
A & \lor & B \\
\exists a. A
\end{align*}$
examples

in prop1:
\[ a \rightarrow a \]

in pred1:
\[ \forall x. a(x) \rightarrow a(x) \]

in prop2:
\[ \forall a. a \rightarrow a \]

for every proposition, that proposition implies itself

higher-order

first order:
object

second order:
set of first-order objects
predicate on objects
function from objects to objects

third order:
set of second-order objects
predicate on predicates on objects
functions from second order objects

higher-order logic

first-order:
quantification over variables of order 1
\[ a \rightarrow a \]
\[ \forall x. a(x) \rightarrow a(x) \]

second-order:
quantification over variables of order 2
\[ \forall a. a \rightarrow a \]
\[ \forall a. \forall x. a(x) \rightarrow a(x) \]
\[ \forall f. \forall x. a(f(x)) \rightarrow a(f(x)) \]

third-order:
quantification over variables of order 3
\[ \forall b. \forall f. b(f) \rightarrow \forall x. a(f(x)) \]

quantify over predicates gives pred2
same without terms gives prop2

second-order predicate logic: example

induction principle for natural numbers
\[ \forall a. a(0) \rightarrow (\forall m. a(m) \rightarrow a(S(m))) \rightarrow \forall n. a(n) \]

m 1st order variable
n 1st order variable
0 1st order constant
a 2nd order variable
S 2nd order constant (or 1st order function)
there exists a sorting function

\[ \exists f : \text{natlist} \rightarrow \text{natlist}. \forall l : \text{natlist}. \text{sorted}(f(l)) \land \text{permutation}(l, f(l)) \]

\( f \) 2nd order variable
\( l \) 1st order variable
\text{sorted} 2nd order constant (or 1st order function)
\text{permutation} 2nd order constant (or 1st order function)

prop2\begin{align*}
\forall a. a & \rightarrow a \\
\forall p. \forall x. p(x) & \rightarrow p(x)
\end{align*}
prop1\begin{align*}
\forall a. a & \rightarrow a \\
\forall x. p(x) & \rightarrow p(x)
\end{align*}

proof rules for prop2

<table>
<thead>
<tr>
<th>introduction rules</th>
<th>elimination rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( I \top )</td>
<td>( E \bot )</td>
</tr>
<tr>
<td>( I [x] \rightarrow )</td>
<td>( E \rightarrow )</td>
</tr>
<tr>
<td>( I \land )</td>
<td>( E I \land, Er \land )</td>
</tr>
<tr>
<td>( I \lor, Ir \lor )</td>
<td>( E \lor )</td>
</tr>
<tr>
<td>( I \forall )</td>
<td>( E \forall )</td>
</tr>
<tr>
<td>( I \exists )</td>
<td>( E \exists )</td>
</tr>
</tbody>
</table>

universal quantification for prop2

\( \exists \) introduction:

\[ \frac{}{\forall a. \quad A} I\forall \]

variable condition: \( a \) not free in any open assumption
check: variable does not occur in any of the available assumptions

\( \exists \) elimination:

\[ \frac{\forall a. A}{A[a := B]} E\exists \]
existential quantification for prop2

\[ \exists \text{ introduction:} \]
\[ A[a := B] \]
\[ \exists a. A \quad I \exists \]

\[ \exists \text{ elimination:} \]
\[ \exists a. A \quad \forall a. A \rightarrow B \quad E\exists \]
variable condition: \( a \) not free in \( B \)
check: variable does not occur in the conclusion

examples of tautologies

\[ \bullet (\forall b. b) \rightarrow a \]
\[ \bullet a \rightarrow \forall b. (b \rightarrow a) \]
\[ \bullet a \rightarrow \forall b. ((a \rightarrow b) \rightarrow b) \]
\[ \bullet (\exists b. a) \rightarrow a \]
\[ \bullet \exists b. ((a \rightarrow b) \lor (b \rightarrow a)) \]

examples of non-tautological formulas

\[ \bullet a \rightarrow (\forall a. a) \]
\[ \bullet p(x) \rightarrow (\forall x. p(x)) \]
\[ \bullet (\exists a. a) \rightarrow a \]
\[ \bullet \forall a. \forall b. (a \rightarrow b) \lor (b \rightarrow a) \]
(classical logic needed)

minimal prop2: detour

introduction rule for a connective
immediately followed by an
elimination rule for the same connective
elimination of an implication detour (as in prop1)

\[
\begin{array}{c}
\therefore B \\
\frac{A \rightarrow B}{A} \rightarrow B\ I[x] \rightarrow \therefore A \\
\end{array}
\]

is replaced by

\[
\begin{array}{c}
\therefore B \\
\therefore B \\
\end{array}
\]

where every occurrence of the assumption \( A^x \) is replaced by the proof

\[
\begin{array}{c}
\therefore \overline{A} \\
\therefore \overline{A} \\
\end{array}
\]

elimination of an universal quantification detour

(similar to pred1)

\[
\begin{array}{c}
\frac{B}{\forall a. B} \ I\forall \\
\frac{B[a := A]}{\exists a. B} \ E\forall \\
\end{array}
\]

everywhere \( a \) is replaced by \( A \)