Foundations of Computation

Ana Bove

Programming Logic Group

February 9th 2017

Warming-up Exercise

$$2 < 4?$$

Now, can you formally prove it?

What would you need to do so?

How to Give a Formal Proof of $2 < 4$?

We need to understand the objects we manipulate ...

Natural numbers: $\mathbb{N}$ is a set (inductively) defined as

$$\frac{}{0 : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{n + 1 : \mathbb{N}}$$

... and also how to relation $<$ is defined!

$\_ < \_ : \mathbb{N} \to \mathbb{N} \to \text{Prop}$

$$\frac{n : \mathbb{N}}{0 < n + 1} \qquad \frac{n < m}{n + 1 < m + 1}$$

Now we can formally prove that $2 < 4$!
Can you see how?

What about more Complex Proofs?

Conjunction:

$$\frac{P \qquad Q}{P \wedge Q}$$

Disjunction:

$$\frac{P}{P \vee Q} \qquad \frac{Q}{P \vee Q}$$

Implication:

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \Rightarrow Q}$$

Propositions as Types, Proofs as Programs

Conjunction:

$$\frac{P \qquad Q}{P \wedge Q}$$

Cartesian product:

$$\frac{a : A \qquad b : B}{< a, b >: A \times B}$$

Disjunction:

$$\frac{P}{P \vee Q} \qquad \frac{Q}{P \vee Q}$$

Disjoint sum:

$$\frac{a : A}{\mathsf{inl}\ a : A + B} \qquad \frac{b : B}{\mathsf{inr}\ b : A + B}$$

Implication:

$$\begin{array}{c} [P] \\ \vdots \\ Q \\ \hline P \Rightarrow Q \end{array}$$

Functions:

$$\begin{array}{c} [a : A] \\ \vdots \\ b : B \\ \hline \lambda a.b : A \rightarrow B \end{array}$$

Are We Missing Something?

Quantifiers!!!

$$\forall x.P(x) \qquad\qquad \exists x.P(x)$$

What do they correspond to in the word of types?

Dependent Types!!

Dependent Types

A *dependent type* is a type that depends on a *value*.

Example: List of a given length.

```
data Vec (A : Set) : ℕ → Set where
  [ ] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

Could also be used to state properties of certain objects!

Example: Property of being a sorted vector.

```
data SortedV : ∀ {n} → Vec ℕ n → Set where
  sorted[ ] : SortedV [ ]
  sorted[-] : ∀ {x} → SortedV [ x ]
  sorted:: : ∀ {n x y} (xs : Vec ℕ n) → x ≤ y →
             SortedV (y :: xs) → SortedV (x :: y :: xs)
```

Programming with Dependent Types: Sorting

How can we write a function that sorts a sequence of numbers?
What type will it have?

$\mathsf{sort} : \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N}$

Result should have the <u>same number of elements</u>:

$\mathsf{sort} : \forall\ \{\mathsf{n}\} \to \mathsf{Vec}\ \mathbb{N}\ \mathsf{n} \to \mathsf{Vec}\ \mathbb{N}\ \mathsf{n}$

Result should <u>be sorted</u>:

$\mathsf{sort} : \forall\ \{\mathsf{n}\} \to \mathsf{Vec}\ \mathbb{N}\ \mathsf{n} \to \exists\ (\lambda\ \mathsf{ys} \to \mathsf{SortedV}\ \{\mathsf{n}\}\ \mathsf{ys})$

Result should have the <u>same elements</u>:

$$\begin{aligned}\mathsf{sort} : &\ \forall\ \{\mathsf{n}\}\ (\mathsf{xs} : \mathsf{Vec}\ \mathbb{N}\ \mathsf{n}) \to \\ &\ \exists\ (\lambda\ \mathsf{ys} \to \mathsf{SortedV}\ \{\mathsf{n}\}\ \mathsf{ys} \times \mathsf{PermV}\ \mathsf{xs}\ \mathsf{ys})\end{aligned}$$

What about the Law of Excluding Middle (LEM)?

We have learnt that the LEM

$$P \vee \neg P$$

is always true (tautology).

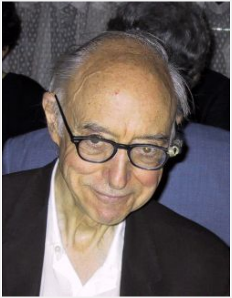
But here we can only construct a proof of it if we know that
P is true or $\neg P$ is true!!

$$\frac{P}{P \vee \neg P} \qquad \frac{\neg P}{P \vee \neg P}$$

We work here with *intuitionistic/constructive* logic!
(as oposite to *classical* logic)

Curry-Howard Isomorphism

In 1934, Haskell Curry observed the correspondance between (a theory of) functions and (a theory of) implications.



In 1969, William Howard extended the correspondance to other logic connectives.
He also proposes new concepts for types (now known as *dependent types*) that would correspond to the quantifiers $\forall$ and $\exists$.

Propositions as Types

Mathematicians and computer scientists proposed numerous systems based on this concept:

- de Bruijn's Automath
- Martin-Löf's type theory, developed into the Agda proof assistant (here at D&IT, Chalmers-GU)
- Bates and Constable's nuPRL
- Coquand and Huet's Calculus of Constructions, developed into the Coq proof assistant

Programming Logic Research Group

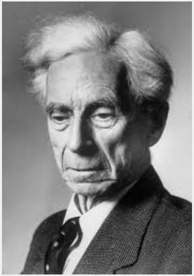
It is our thesis that formal elegance is a prerequisite to efficient implementation.

Gérard Huet

Senior members:

- Thierry Coquand
- Peter Dybjer
- Andreas Abel
- Ana Bove
- Nils Anders Danielsson
- Ulf Norell
- Simon Huber

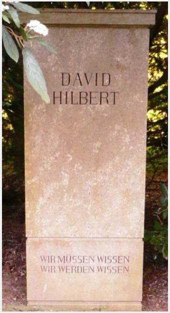
Once Upon a Time ...



In early 1900's, Bertrand Russell showed that formal logic can express large parts of mathematics.

In 1928, David Hilbert posed a challenge known as the Entscheidungsproblem (decision problem). This problem asked for an *effectively calculable* procedure to determine whether a given statement is provable from the axioms using the rules of logic.

To Prove or Not To Prove: That Is the Question!



The decision problem presupposed completness: any statement or its negation can be proved.

"Wir müssen wissen, wir werden wissen"
("We must know, we will know")

In 1931, Kurt Gödel published the *incompleteness theorems*.

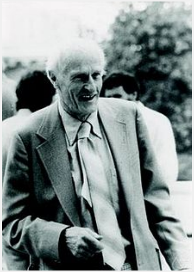
The first theorem shows that any consistent system capable of expressing arithmetic cannot be complete: there is a true statement that cannot be proved with the rules of the system.

The second theorem shows that such a system could not prove its own consistency.

λ-Calculus as a Language for Logic

In the '30s, Alonzo Church (and his students Stephen Kleene and John Barkley Rosser) introduced the λ-calculus as a way to define notations for logical formulas:

$$x \mid \lambda x.M \mid M\ N$$



In 1935, Kleene and Rosser proved the system inconsistent (due to self application).

λ-Calculus as a Language for Computations

Church discovered how to encode numbers in the λ-calculus.

For example, 3 is encoded as $\lambda f.\lambda x.f(f(f(x)))$.

Encoding for addition, multiplication and (later) predecesor were defined.

Thereafter Church and his students became convinced any *effectively calculable* function of numbers could be represented by a term in the λ-calculus.

Church's Thesis

Church proposed λ-definability as the definition of effectively calculable (known today as *Church's Thesis*).

He also demonstrated that the problem of whether a given λ-term has a normal form was not λ-definable (equivalent to the *Halting problem*).

A year later, he demonstrated there was no λ-definable solution to the Entscheidungsproblem.

General Recursive Functions

1933: Gödel was not convinced by Church's assertion that every effectively calculable function was λ-definable.

Church offered that Gödel would propose a different definition which he then would prove it was included in λ-definability.

1934: Gödel proposed the *general recursive functions* as his candidate for effective calculability (system which Kleene after developed and published).

Church and his students then proved that the two definitions were equivalent.

Now Gödel doubt his own definition was correct!

Turing Machines

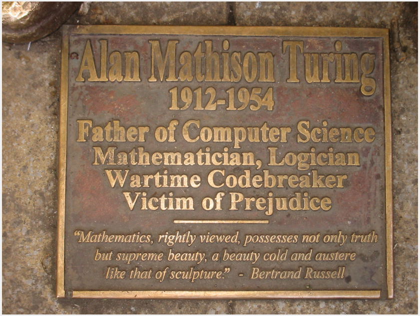
Simultaneously, Alan Mathison Turing formulated his notion of effectively calculable in terms of a *Turing machine*.

He used the Turing machines to show the Entscheidungsproblem undecidable.

Turing also proved the equivalence of the λ-calculus and his machines. (*Church-Turing Thesis*)

Gödel is now finally convinced! :-)

Computer Science Was Born!



Turing's approach took into account the capabilities of a *(human) computer*: a human performing a computation assisted by paper and pencil.

Turing Award

Since 1966, annual prize from the Association for Computing Machinery (ACM) for lasting technical contributions to the computing community.

Seen as the *Nobel Prize of computing*.

Which Courses Can You Take?

- TMV027/DIT321 *Finite Automata Theory and Formal Languages*.
  Bachelor course given in LP4.
- DAT060/DIT201 *Logic in Computing Sciences*.
  Master course given in LP1.
- DAT140(DAT350)/DIT232 *Types for Programs and Proofs*.
  Master course given in LP1.
- TDA184/DIT310 *Models of Computation*.
  Master course given in LP2.