Proof Assistants

Type theory, formal logic and proof assistants

Prabhakar Ragde
University of Waterloo

We can’t completely test our programs or our mathematical conjectures.

Proof offers a finite (hopefully shorter) way to have confidence in these.

Informal proofs can be wrong.

Completely formal proofs are often too long or tedious to write.
(And can still be wrong.)

But completely formal proofs can be machine-checked.

Informal proofs require human ingenuity to construct.

Fully formal proofs can be found by a machine.

But automatic theorem proving is usually difficult.

Idea:
A proof assistant to help a human construct a correct formal proof.

Some systems using this idea:
ACL2, Isabelle/HOL, Coq, Agda, Idris, F*, Lean.

Let's start with a simpler task.

Expression in program: `f(g(x))`.

Do we have to run the program to see if `f` and `g` are being used correctly?

Expression: `f(g(x))`

If we know the **types** of `f`, `g`, `x`,
we can check if it is possible
to apply `g` to `x`.

Expression: `f(g(x))`

`x` is a value.
It has some type A (e.g `Integer`).

We say $x : A$.

Then `g` must have type $A \to B$
for some type B.

Expression: `f(g(x))`

$x : A$

$g : A \to B$

Similarly, `f` must have type $B \to C$ for some type C.

This is called **typechecking**.

Let's generalize this reasoning.

Function names are unimportant.
We use **lambda notation**
to describe functions.

If $f(x) = x^2$, we replace
f with $\lambda x \,.\, x^2$.

Our functions now look like $\lambda x.e$ for some expression e that uses x.

How do we **verify** that $\lambda x.e : T \to U$?

To verify that $\lambda x.e : T \to U$,
we need to verify that $e : U$.

But e uses the parameter x, and
we need to remember that $x : T$
while we verify $e : U$.

We remember this information
in a **context** Γ.

Our verification task is now
$\Gamma \vdash \lambda x.e : T \to U$.

To verify this, we must verify
$\Gamma, x : T \vdash e : U$
where $\Gamma, x : T$ is $\Gamma \cup \{x : T\}$.

We write this in a concise format called an **inference rule**.

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x.e : T \to U}\ (\textsc{Abs})$$

This is the inference rule for functions (abstractions).

We must handle
function application.

We write $f\ x$ instead of $f(x)$.

To verify $\Gamma \vdash f\ x : U$, we need to verify $\Gamma \vdash f : T \to U$ for some T, and we need to verify $\Gamma \vdash x : T$.

This is the inference rule for function applications.

$$\frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash x : T}{\Gamma \vdash f\ x : U}\ (\textsc{App})$$

One rule remains.

To verify $\Gamma \vdash x : T$,
we can check $x : T \in \Gamma$.

We can write this rule as

$$\frac{}{\Gamma', x : T \vdash x : T}\ (\textsc{Var})$$

Typechecking rules (summary):

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x.e : T \to U}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash x : T}{\Gamma \vdash f\ x : U}\ (\textsc{App})$$

$$\frac{}{\Gamma, x : T \vdash x : T}\ (\textsc{Var})$$

$\lambda x.x$ is the identity function.

Here is a verification of
$\lambda x.x : A \to A$.

$$\dfrac{\dfrac{}{x : A \vdash x : A}\,(\textsc{Var})}{\vdash \lambda x.x : A \to A}\,(\textsc{Abs})$$

Here is a verification of
$\lambda x.\lambda y.x : A \to (B \to A)$.

$$\dfrac{\dfrac{\dfrac{}{x : A, y : B \vdash x : A}\,(\textsc{Var})}{x : A \vdash \lambda y.x : B \to A}\,(\textsc{Abs})}{\vdash \lambda x.\lambda y.x : A \to (B \to A)}\,(\textsc{Abs})$$

Here is a verification of
$\lambda x.\lambda y.(y\ x) : A \to ((A \to B) \to B)$.

$$\dfrac{\dfrac{\dfrac{}{x : A, y : A \to B \vdash y : A \to B} \quad \dfrac{}{x : A, y : A \to B \vdash x : A}\ (\textsc{Var})}{x : A, y : A \to B \vdash y\ x : B}\ (\textsc{App})}{x : A \vdash \lambda y\ .\ (y\ x) : (A \to B) \to B}\ (\textsc{Abs})}{\vdash \lambda x\ .\ \lambda y\ .\ (y\ x) : A \to ((A \to B) \to B)}\ (\textsc{Abs})$$

We have used $A \to B$ to mean the type of functions from A to B.

$A \to B$ also has a meaning in logic.

It is an **implication**. It means "If A, then B", or "A implies B".

Suppose we erase the expressions and pretend the types are logical formulas (logical statements).

$x : A \vdash \lambda y.x : B \to A$
becomes $A \vdash B \to A$.

Does this (and do our rules) have any meaning in logic?

In $\Gamma \vdash \phi$:
ϕ is a logical formula.
Γ is a set of formulas.

$\Gamma \vdash \phi$ means:
“Assuming proofs for formulas in Γ,
we can prove ϕ”.

Typechecking rules (review):

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x.e : T \to U}\ (\textsc{Abs})$$

$$\frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash x : T}{\Gamma \vdash f\ x : U}\ (\textsc{App})$$

$$\frac{}{\Gamma, x : T \vdash x : T}\ (\textsc{Var})$$

Logical rules:

$$\frac{\Gamma, T \vdash U}{\Gamma \vdash T \to U} \ (\to \text{INTRODUCTION})$$

$$\frac{\Gamma \vdash T \to U \qquad \Gamma \vdash T}{\Gamma \vdash U} \ (\to \text{ELIMINATION})$$

$$\frac{}{\Gamma, T \vdash T} \ (\text{ASSUMPTION})$$

We erased the expressions.

Could we keep them?

Do they have a logical meaning?

Yes! Let’s explore this.

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x.e : T \to U}\ (\textsc{Abs})$$

Informally, to prove $T \to U$,
we assume T and
use the assumption to prove U.

Suppose we have proved $A \longrightarrow B$.

In that proof of B, there may be many places where the assumption rule is used with A.

Now suppose we have a proof of A.

We could take the proof of B using the assumption A and replace every use of the assumption rule with A with the proof of A.

That would give us a proof of B without the assumption.

In other words, a proof of $A \to B$ can be viewed as a function that consumes a proof of A and produces a proof of B.

$$\frac{\Gamma, x : T \vdash e : U}{\Gamma \vdash \lambda x.e : T \to U} \text{ (ABS)}$$

$$\frac{\Gamma \vdash f : T \to U \qquad \Gamma \vdash x : T}{\Gamma \vdash f\ x : U}\ (\textsc{App})$$

Informally, how do we use a proof of $A \to B$?

To use the theorem $A \to B$, we find a proof of A, then apply the theorem. This gives us a proof of B.

Using a proof of $A \to B$
is function application.

The Curry-Howard correspondence

Logic	Programming
formula	type
$\rightarrow_{\texttt{intro}}$ rule	Abs rule
$\rightarrow_{\texttt{elim}}$ rule	App rule
Assumption rule	Var rule
proof of α	expr of type α
proof-checking	type-checking

$$\lambda f \,.\, \lambda g \,.\, \lambda x \,.\, f\,(g\,x) :$$
$$(B \to C) \to ((A \to B) \to (A \to C))$$

The proof of
the transitivity of implication
is the function composition operator.

We can extend this to the other logical connectives
$\wedge$ (and), $\vee$ (or), $\neg$ (not)
to get a proof system
for intuitionistic propositional logic.

This is a constructive logic.
It does not admit LEM: $A \vee (\neg A)$.

For propositional logic,
we add the logical quantifiers
$\forall$ (for all), $\exists$ (exists).

This is best done with
dependent types.

Informally, how do we use
a for-all statement?

To use $\forall(x : A).\phi$, we substitute a specific value $v : A$ for x in ϕ.

Using a for-all is function application.
So the proof of a for-all is a function.

The proof of $\forall(x : A).\phi$ is a function whose result type $\phi(x)$ **depends** on the *value* v of its argument x.

We also need to handle equality.

Finally, we need concrete data types, both non-recursive (e.g. Booleans) and recursive (e.g. natural numbers, lists).

Agda (Göteborg/Chalmers, Sweden)

Early work 1990
First version 1999
Second version 2005

Interface using Emacs (text editor)
Explicit proof expressions

Agda notation

`T → U` is a regular function type.
`(x : T) → U` is a
dependent function type.

Alternatives:
`forall (x : T) → U`
`∀ (x : T) → U`

Interacting with Agda in Emacs

`?` represents a hole or goal.

`C-c C-l` to **load** a file.
This typechecks the code
(holes always typecheck).

`C-c C-,` gives hole information.
`C-c C-r` **refines** a hole.

Coq (INRIA, France)

Early work 1984
Major revisions 2001, 2006

Application; also Emacs interface
Proof script using tactics
(hides proof expressions)
Program extraction