



# Proost 0.3.0

## User manual

ENS Paris-Saclay

Arthur ADJEDJ  
Vincent LAFEYCHINE

Augustin ALBERT  
Lucas TABARY-MAUJEAN

30th January 2023

## 1 Introduction

This program provides the user a set of tools to work with  $\lambda$ -terms as described in the Calculus of Construction (CoC). Through the Curry-Howard correspondence users can associate a property or theorem they are willing to prove to a type which can be expressed in CoC. Proving it then corresponds to constructing a term of that type.

**Release 0.1.0** of the project includes a toplevel interface also called **proost**, which is the main way users can interact with this piece of software. For a use directly through the crate APIs, please refer to their respective documentations.

**Release 0.2.0** includes the ability to manipulate universe-polymorphic declarations, a framework for hardcoding axioms in the code base of the kernel, a better handling of error locations in the interface and a proof-of-concept implementation of the LSP protocol.

**Release 0.3.0** provides a variety of minor upgrades, including a better printing of terms, some optimisations in the kernel, many fixes in the parser and the toplevel, the beginning of a standard library as well as new types, most notably natural numbers and equality. We would like to thank all new contributors for their efforts!

## 2 Toplevel session

In the toplevel, users are greeted with a prompt. There, they may enter the following commands:

- `import file1 file2` typechecks and loads the files in the current environment;
- `search v` looks for the definition of variable `v`;
- `def a := t` defines an alias `a` that can be used in any following command;
- `def a: ty := t` defines an alias `a` that is checked to be of type `ty`;
- `check u: t` verifies `u` has type `t`;
- `check u` provides the type of `u`;
- `eval u` provides the normal form of `u`.

Optionally, defined terms can be of the form `a.{i, j}`, meaning they are universe-polymorphic in `i` and `j`. In that case, they are called *declarations*. Later, these declarations can be used for creating new terms, by calling them like `a.{n, m}`, where `n` and `m` are well-defined universe levels.

If the command succeeds, the toplevel returns a green check mark, with an associated result if there is any. Otherwise, a red cross indicates an error occurred, next to some details about it. The command is discarded and the user may enter another command.

The toplevel provides to a certain extent history browsing, either *via* the up and down arrow keys or some auto-completion from previous commands. The toplevel also provides partial syntax highlighting, multi-line editing, which integrates with a detection of the currently-opened parentheses, if any. An example session is shown in figure 2.

```

» import std/nat.mdln
✓
» add Zero Zero
X ^-^
X expected def var := term, [...] eval term, import path_to_file, or search var
» eval add Zero Zero
✓ Zero
» eval add (add Zero (fun p: Prop -> Prop, x: Prop => p (p x))) Zero
X ^-----^
X function (λ Nat => NatRec (λ b: Nat => Nat) 1 (λ Nat => λ Nat => Succ 1)) Zero
X expects a term of type Nat, received λ (b: Prop) -> Prop => λ Prop => 2 (2 1):
X (a: (b: Prop) -> Prop) -> (b: Prop) -> Prop

```

Figure 1: Example of an interactive toplevel session

### 3 Language

Language syntax is as such:

- Functions ( $\lambda$ -abstractions) are defined with the keyword **fun**: **fun**  $x: A \Rightarrow u$ , **fun**  $x\ y: A \Rightarrow u$  (both  $x$  and  $y$  are of type  $A$ ), **fun**  $x: A, y: B \Rightarrow u$  (multiple arguments, where  $B$  may depend on  $x$ );
- Dependent function types ( $\Pi$ -types) are defined with a pair of parentheses before an arrow, as in:  $(x: A) \rightarrow B$ ,  $(x\ y: A) \rightarrow B$  or  $(x: A, y: B) \rightarrow C$  (where the distinctions are similar to the previous item. Additionally, there is some usual syntactic sugar when the output type does not mention the input argument, which corresponds to usual function types:  $A \rightarrow B$ ,  $A \rightarrow B \rightarrow C$  (right-associativity);
- Function application of two terms  $u$  and  $v$  is simply written  $u\ v$ , and is left-associative when there are multiple arguments;
- Variables are regular strings, which may only correspond to bound variables or previously defined terms (or declarations as explained in the previous section);
- The type of propositions and higher-order types are written **Prop** and **Type**  $i$ , as usual. One may also refer to the universes in hierarchy through the **Sort** keyword as follows: **Sort**  $0 = \text{Prop}$  and **Sort**  $n + 1 = \text{Type } n$ .

The syntax of universe levels is the following:

- numeric constants and level variables defined in the definition of a declaration are valid universe level;
- if  $u$  is a valid level,  $u + n$  is valid, where  $n$  is a numeric constant;
- if  $u$  and  $v$  are valid levels, **max**  $u\ v$  and **imax**  $u\ v$  (impredicative maximum) are valid levels.

### 4 Axioms

There is no simple way to browse the set of axioms or their recursors for the moment. This should be made easier in future releases. The general current syntax is, for an inductive type **Foo**, to have a declaration **Foo\_rec** which corresponds to its recursor, and other appropriately-named functions as its constructors.

Please refer to the code presented in **example/** and **std/** for concrete examples.