

Proost: specifications

A small proof assistant written in Rust

ENS Paris-Saclay

Arthur ADJEDJ
Vincent LAFEYCHINE

Augustin ALBERT
Lucas TABARY-MAUJEAN

10th November 2022

This project is under development, and its specifications themselves are subject to changes, should time be an issue or a general consensus be reached to change the purposes of the tool. An example of that is the syntax of the language, which is still largely unstable.

1 General purpose and functions

This project aims at providing a small tool for typechecking expressions written in the language of the Calculus of Construction (CoC). This tool shall be both terminal and editor based through the **proost** program that provides both compiler and toplevel-like capacities and options and a LSP called **tilleul**. The file extension used by both programs is **.mdl_n**, which is short for *madelaine*, the name of the language manipulated by users in these files.

2 Project structure

Each category of the project is assigned some or all members of the group, meaning the designated members will *mainly* make progress in the associated categories and review the corresponding advancements. Any member may regardless contribute to any part of the development of the tool.

Some specific categories and items will be added a star (★) or two (★★) to indicate whether they are respectively late requirements (for the last release due in December) or extra requirements that will be considered only if there is enough time.

2.1 Language design

all members

The **proost** tool is a simple proof assistant and does not provide any tactics. As new features arrive from extensions of the kernel type theory, the *madelaine* language must provide convenient shorthands and notations. The syntax of commands is the following:

- (★) **import** *relative_path_to_file* typecheck and load the the file in the current environment;
- **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 definition of *u*.

Below is an overview of the syntax of the terms, both present and future. The syntax is strongly inspired by that of OCaml. Comments are defined using the keyword **//**.

elementary type theory:

```
1 // Construction of natural numbers
2 def Nat :=
3   (N: Type) -> (N -> N) -> N -> N
4
5 def z := fun N:Type =>
6   fun f:(N -> N), x:N => x
7 check z : Nat
8
9 def succ := fun n: Nat, N: Type =>
10  fun f: (N -> N), x: N => f (n N f
11    x)
11 check succ : Nat -> Nat
```

(★) universe polymorphism:

```
1 def foo.{i,j} : Type (max i j) + 1
2 := Type i -> Type j
```

(★★) unification:

```
1 def comm := \ / x,y, x + y = y + x
```

(★★) existential types:

```
1 def t := E n, n * n - n + 4 = 0
```

2.2 Toplevel

AuA

The `proost` command, when provided with no argument, is expected to behave like a toplevel, akin to `ocaml` or `coqtop`. There, user is greeted with a prompt and may enter commands. When provided with existing file paths, `proost` intends to typecheck them in order, that is, reading them as successive inputs in the toplevel. Further features for this *might* include a more extended notion of “modules” where files may provides scopes.

```
Welcome to proost 0.1.0
» def f := fun x:Prop =>
X
X expected variable, abstraction, dependent product,
  application, product, Prop, or Type
» check fun x:Prop => x -> x
✓ Π Prop → Prop
» import
```

2.3 (★) LSP

VL

`tilleul` shall provide an implementation of the Language Server Protocol in order to provide linting and feedback during an editing session.

2.4 Parsing

AuA

The parsing approach is straightforward and relies on external libraries. The parser is expected to keep adapting to changes made in the term definitions and unification capability. The parser is thoroughly tested to guarantee full coverage.

2.5 Kernel

all members

The kernel manipulates λ -terms in the Calculus of Construction and is expected to store and manage them with a relative level of efficiency. The type theory used to build the terms will be successively extended with:

- abstractions, Π -types, predicative universes with `Prop`;
- (★) universe polymorphism;
- (★) Σ -types, equality types, natural numbers;
- (★★) extraction;
- (★★) lists, records, accessibility predicate.

2.6 (★) Optimisation

ArA LTM

Extra care must be put into designing an efficient memory management model for the kernel, along with satisfactory typing and reduction algorithms.

In particular, the first iteration of the program manipulates directly terms on the heap, with no particular optimisation: every algorithm is applied soundly but naively.

A first refactor of the memory model includes using a common memory location for terms, ensuring invariant like unicity of a term in memory, providing laziness and storing results of the most expensive functions (*memoizing*). This model also provides stronger isolation properties, preventing several memory pools (*arena* is the technical term used in the project) from interacting with one another.

This can be further extended with other invariants like ensuring every term in memory is in weak head normal form.

2.7 (★★) Unification

Early versions of the tool may require the user to explicit every type at play. Successive versions may gradually include unification tools (meta-variables) of better quality to assist the user and alleviate some of their typing-annotation burden.

2.8 Developpement tools

VL LTM

Tests are mandatory in every part of the project. A tool originally developed by the Mozilla team was modified to allow for a more precise branch coverage of the project. The Nix framework is used to automatically build and package the application as well as generating a docker image and providing developers tools of the same version.