MiniProver — A Coq-like Proof Assistant

Zhenwen Li, Sirui Lu, Kewen Wu

June 17, 2018

Contents

1 Introduction

In modern mathematical and computer science world, providing a proof is hard, but confirming its correctness is sometimes not a easy work as well. In November 2015, Babai announced a quasipolynomial time algorithm for graph isomorphism problem, where Harald Helfgott discovered a flaw. Though Babai re-announced a correction in 2017, the new proof has not been fully peer-reviewed yet.

On the other hand, some famous problems have been solved with the help of computer programs, for example proving *Four Color Theorem* using Coq.

The key idea is that although searching for proofs is an undeciable and unpractical task with respect to the limited computation power and ability of computer, checking the validity of a certain proof can be done in computer with the help of type system.

In our implementation, using *Curry-Howard isomorphism*, programs, properties, and proofs are formalized in the same language – *Calculus of Inductive Constructions*, which is a λ-calculus with an enriched type system.

2 Syntax

```
firstletter   ::=  a..z | A..Z
letter        ::=  a..z | A..Z | _
ident         ::=  firstletter {letter}
term          ::=  forall binder {binder}, term
                 |  fun {binder} => term
                 |  fix ident binder {binder} : term := term
                 |  let ident {binder} :term := term in term
                 |  term -> term
                 |  term arg {arg}
                 |  match term as ident in ident {ident} return term with
                       { | equation}
                    end
                 |  Type
                 |  (term)
arg           ::=  term
binder        ::=  (ident : term)
equation      ::=  pattern => term
pattern       ::=  ident {ident}


sentence      ::=  axiom
                 |  definition
                 |  inductive
                 |  fixpoint
                 |  assertion proof
axiom         ::=  Axiom ident : term .
definition    ::=  Definition ident {binder} : term := term .
inductive     ::=  Inductive ident {binder} : term :=
                       { | ident : term} .
fixpoint      ::=  Fixpoint ident {binder} : term := term .
assertion     ::=  Theorem ident {binder} : term .
proof         ::=  Proof . {tactic .} Qed .
helper        ::=  printing
                 |  proof_handling
printing      ::=  Print ident .
                 |  Check term .
proof_handling ::= Undo .
                 |  Restart .
                 |  Admitted .
                 |  Abort .
```

tactic	::=	*applying*
	|	*context_managing*
	|	*case_analyzing*
	|	*rewriting*
	|	*computing*
	|	*equality*
applying	::=	`exact` *term*
	|	`apply` *term* [`in` *ident*]
	|	`left`
	|	`right`
	|	`split`
	|	`exists`
context_managing	::=	`intro` {*ident*}
	|	`intros`
case_analyzing	::=	`destruct` *term*
	|	`induction` *term*
rewriting	::=	`rewrite` [`<-` | `->`] *term* [`in` *term*]
computing	::=	`simpl` [`in` *ident*]
	::=	`unfold` [`in` *ident*]
equality	::=	`reflexivity`
	|	`symmetry`

3 Lexer and Parser

The parser is implemented as a simple $LL(\infty)$ parser using parser combinators so it may be quite slow on large inputs, but it's sufficient for our purpose.

Our AST for terms is defined as follows, for brevity, AST for other objects are not described here.

```
type Name = String
type Index = Int

data Term =
    TmRel
      Name        -- name of the variable, used for pretty printing
      Index       -- 0 based DeBruijn index
  | TmVar
      Name        -- name of the variable
  | TmAppl
      [Term]      -- the first is the abstraction and the rest are the arguments
  | TmProd
      Name        -- name of the abstracted variable, used for pretty printing
      Term        -- type of the abstracted variable
      Term        -- body of the abstraction
  | TmLambda
      Name        -- name of the abstracted variable, used for pretty printing
      Term        -- type of the abstracted variable
      Term        -- body of the abstraction
  | TmFix
      Int         -- the index of the decreasing variable
      Term        -- body of the fix definition
  | TmLetIn
      Name        -- name of the local binding variable
      Term        -- type of the local binding
      Term        -- body of the local binding
      Term        -- let body, the binding will be added here
  | TmIndType
      Name        -- name of the inductive type constructor
      [Term]      -- argument list
  | TmConstr
      Name        -- name of the term constructor
      [Term]      -- argument list
  | TmType
  | TmTypeHigher
  | TmMatch
      Int         -- how many parameters does the inductive type need
      Term        -- the term pattern matching on
      Name        -- the name of the local binding of the term in the return type
      [Name]      -- the matching name list for the return type
      Term        -- return type
      [Equation] -- equations, described below
  deriving (Eq, Show)

data Equation =
    Equation
      [Name]      -- matching name list for the term constructor
      Term        -- body of the equation
    deriving (Eq, Show)
```

4 Typing

4.1 Term

1. $\texttt{Type}$ is a term.
2. Variables $\texttt{x}, \texttt{y}$, etc., are terms.
3. Constants $\texttt{c}, \texttt{d}$, etc., are terms.
4. If $\texttt{x}$ is a variable and $\texttt{T}, \texttt{U}$ are terms, then $\forall \texttt{x} : \texttt{T}, \texttt{U}$ is a term.
5. If $\texttt{x}$ is a variable and $\texttt{T}, \texttt{u}$ are terms, then $\lambda \texttt{x} : \texttt{T}.\ \texttt{u}$ is a term.
6. If $\texttt{x}$ and $\texttt{u}$ are terms, then $(\texttt{t}\ \texttt{u})$ is a term.
7. If $\texttt{x}$ is a variable and $\texttt{t}, \texttt{T}, \texttt{u}$ are terms, then $\texttt{let}\ \ \texttt{x} := \texttt{t} : \texttt{T}\ \ \texttt{in}\ \ \texttt{u}$ is a term.

4.2 Typing Rule

4.2.1 Notation

- Γ : local context.
- $\texttt{u}\{\texttt{x}/\texttt{t}\}$: substitute free occurrence of variable $\texttt{x}$ to term $\texttt{t}$ in term $\texttt{u}$.
- $\mathcal{WF}(\Gamma)$: Γ is well-formed.

4.2.2 Typing Rules

$$\mathcal{WF}([])[] \tag{T-Empty}$$

$$\frac{\Gamma \vdash \texttt{T} : \texttt{Type} \qquad \texttt{x} \notin \Gamma}{\mathcal{WF}(\Gamma :: (\texttt{x} : \texttt{T}))} \tag{T-Ax}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T} \qquad \texttt{c} \notin \Gamma}{\mathcal{WF}(\Gamma : \texttt{c} := \texttt{t} : \texttt{T})} \tag{T-Def}$$

$$\frac{\mathcal{WF}(\Gamma) \qquad (\texttt{x} : \texttt{T}) \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}} \tag{T-Var1}$$

$$\frac{\mathcal{WF}(\Gamma) \qquad (\texttt{x} := \texttt{t} : \texttt{T}) \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}} \tag{T-Var2}$$

$$\frac{\mathcal{WF}(\Gamma) \qquad (\texttt{c} : \texttt{T}) \in \Gamma}{\Gamma \vdash \texttt{c} : \texttt{T}} \tag{T-Const1}$$

$$\frac{\mathcal{WF}(\Gamma) \qquad (\texttt{c} := \texttt{t} : \texttt{T}) \in \Gamma}{\Gamma \vdash \texttt{c} : \texttt{T}} \tag{T-Const2}$$

$$\frac{\Gamma \vdash \texttt{T} : \texttt{Type} \qquad \Gamma :: (\texttt{x} : \texttt{T}) \vdash \texttt{U} : \texttt{Type}}{\Gamma \vdash \forall \texttt{x} : \texttt{T}, \texttt{U} : \texttt{Type}} \tag{T-Prod}$$

$$\frac{\Gamma \vdash \forall \texttt{x} : \texttt{T}, \texttt{U} : \texttt{Type} \qquad \Gamma :: (\texttt{x} : \texttt{T}) \vdash \texttt{t} : \texttt{U}}{\Gamma \vdash \lambda \texttt{x} : \texttt{T}.\ \texttt{t} : \forall \texttt{x} : \texttt{T}, \texttt{U}} \tag{T-Abs}$$

$$\frac{\Gamma \vdash \forall \texttt{x} : \texttt{U}, \texttt{T} \qquad \Gamma \vdash \texttt{u} : \texttt{U}}{\Gamma \vdash (\texttt{t}\ \texttt{u}) : \texttt{T}\{\texttt{x}/\texttt{u}\}} \tag{T-App}$$

$$\frac{\Gamma \vdash \texttt{t} : \texttt{T} \qquad \Gamma :: (\texttt{x} := \texttt{t} : \texttt{T}) \vdash \texttt{u} : \texttt{U}}{\Gamma \vdash \texttt{let}\ \ \texttt{x} := \texttt{t} : \texttt{T}\ \ \texttt{in}\ \ \texttt{u} : \texttt{U}\{\texttt{x}/\texttt{t}\}} \tag{T-Let}$$

$$\frac{(\Gamma \vdash \texttt{A}_\texttt{i} : \texttt{s}_\texttt{i})_{\texttt{i}=1..\texttt{n}} \qquad (\Gamma, \texttt{f}_1 : \texttt{A}_1, \cdots, \texttt{f}_\texttt{n} : \texttt{A}_\texttt{n} \vdash \texttt{t}_\texttt{i} : \texttt{A}_\texttt{i})_{\texttt{i}=1..\texttt{n}}}{\Gamma \vdash \texttt{Fix}\ \texttt{f}_\texttt{i}\{\texttt{f}_1 : \texttt{A}_1 := \texttt{t}_1 \cdots \texttt{f}_\texttt{n} : \texttt{A}_\texttt{n} := \texttt{t}_\texttt{n}\} : \texttt{A}_\texttt{i}} \tag{T-Fix}$$

4.3 Inductive Definition

4.3.1 Notation

- $\texttt{Ind}[p](\Gamma_I := \Gamma_C)$: inductive definition.
- Γ_I : names and types of inductive type.
- Γ_C : names and types of constructors of inductive type.
- p : the number of parameters of inductive type.
- Γ_P : the context of parameters.

Example

Here is an example:

```
Inductive list (T : Type) : Type :=
| nil : list T
| cons : T -> list T -> list T.
```

$$\texttt{Ind}\ [1]\left([\texttt{list} : \texttt{Type} \to \texttt{Type}] := \begin{bmatrix} \texttt{nil} : \forall \texttt{T} : \texttt{Type},\ \texttt{list T} \\ \texttt{cons} : \forall \texttt{T} : \texttt{Type},\ \texttt{T} \to \texttt{list T} \to \texttt{list T} \end{bmatrix}\right)$$

Sadly, due to our limited time and energy, our system only supports inductive definition with $|\Gamma_I| = 1$, which means inhibiting mutual inductive type like the following case:

```
Inductive tree : Type :=
    node : forest -> tree
with forest : Type :=
    | emptyf : forest
    | consf : tree -> forest -> forest.
```

$$\texttt{Ind}\ []\left(\begin{bmatrix} \texttt{tree} : \texttt{Type} \\ \texttt{forest} : \texttt{Type} \end{bmatrix} := \begin{bmatrix} \texttt{node} : \texttt{forest} \to \texttt{tree} \\ \texttt{emptyf} : \texttt{forest} \\ \texttt{consf} : \texttt{tree} \to \texttt{forest} \to \texttt{forest} \end{bmatrix}\right)$$

4.3.2 Typing Rule

$$\frac{\mathcal{WF}(\Gamma) \qquad \texttt{Ind}[p](\Gamma_I := \Gamma_C) \in \Gamma \qquad (\texttt{a} : \texttt{A}) \in \Gamma_i}{\Gamma \vdash \texttt{a} : \texttt{A}} \tag{T-Ind}$$

$$\frac{\mathcal{WF}(\Gamma) \qquad \texttt{Ind}[p](\Gamma_I := \Gamma_C) \in \Gamma \qquad (\texttt{c} : \texttt{C}) \in \Gamma_C}{\Gamma \vdash \texttt{c} : \texttt{C}} \tag{T-Constr}$$

$$\frac{(\Gamma_P \vdash \texttt{A}_\texttt{j} : \texttt{s}'_\texttt{j})_{j=1..k} \qquad (\Gamma_i; \Gamma_P \vdash \texttt{C}_\texttt{i} : \texttt{s}_{\texttt{q}_\texttt{i}})_{i=1..n}}{\mathcal{WF}(\Gamma; \texttt{Ind}[p](\Gamma_I := \Gamma_C))} \tag{T-Wf-Ind}$$

4.4 Match

4.4.1 Notation

The basic idea of this operator is that we have an object $\mathtt{m}$ in an inductive type $\mathtt{I}$ and we want to prove a property which possibly depends on $\mathtt{m}$. For this, it is enough to prove the property for $\mathtt{m} = (\mathtt{c_i}\ \mathtt{u_1}\ \cdots\ \mathtt{u_{p_i}})$ for each constructor of $\mathtt{I}$. The term for this proof will be written:

$$\mathtt{match\ m\ in\ I\ \_\ a\ return\ P\ with}\ (\mathtt{c_1}\ \mathtt{x_{11}}\ \cdots\ \mathtt{x_{1p_1}}) \Rightarrow \mathtt{f_1} \mid \cdots \mid (\mathtt{c_n}\ \mathtt{x_{n1}}\ \cdots\ \mathtt{x_{np_n}}) \Rightarrow \mathtt{f_n}\ \mathtt{end}$$

Note that the arguments of $\mathtt{I}$ corresponding to parameters must be _, because the result type is not generalized to all possible values of the parameters. The other arguments of $\mathtt{I}$ (sometimes called *indices* in the literature) have to be variables ($\mathtt{a}$ above) and these variables can occur in $\mathtt{P}$. The expression after `in` must be seen as an inductive type pattern. Notice that expansion of implicit arguments and notations apply to this pattern. For the purpose of presenting the inference rules, we use a more compact notation:

$$\mathtt{case}(\mathtt{m}, (\forall \mathtt{a}x.\ \mathtt{P}), \lambda \mathtt{x_{11}} \cdots \mathtt{x_{1p_1}}.\ \mathtt{f_1} \mid \cdots \mid \lambda \mathtt{x_{n1}} \cdots \mathtt{x_{np_n}}.\ \mathtt{f_n})$$

4.4.2 Type of Branch

Before coming to the typing rule of `match`, we have to deal with the type of every branch case of `match`.

Let $\mathtt{c}$ be a term of type $\mathtt{C}$, assume $\mathtt{C}$ is a type of constructor for an inductive type $\mathtt{I}$. Let $\mathtt{P}$ be a term that represents the property to be proved. Assume $\mathtt{r}$ is the number of parameters and p is the number of arguments.

Define a new type $\{\mathtt{c} : \mathtt{C}\}^{\mathtt{P}}$ which represents the type of the branch corresponding to the $\mathtt{c} : \mathtt{C}$ constructor.

$$\begin{array}{lcl} \{\mathtt{c} : (\mathtt{I}\ \mathtt{p_1}\ \cdots\ \mathtt{p_r}\ \mathtt{t_1}\ \cdots\ \mathtt{t_p})\}^{\mathtt{P}} & \equiv & (\mathtt{P}\ \mathtt{t_1}\ \cdots\ \mathtt{t_p}\ \mathtt{c}) \\ \{\mathtt{c} : \forall \mathtt{x} : \mathtt{T},\ \mathtt{C}\}^{\mathtt{P}} & \equiv & \forall \mathtt{x} : \mathtt{T}, \{(\mathtt{c}\ \mathtt{x}) : \mathtt{C}\}^{\mathtt{P}} \end{array}$$

We write $\{\mathtt{c}\}^{\mathtt{P}}$ for $\{\mathtt{c} : \mathtt{C}\}^{\mathtt{P}}$ with $\mathtt{C}$, the type of $\mathtt{c}$.

4.4.3 Typing Rule

$$\frac{\Gamma \vdash \mathtt{c} : (\mathtt{I}\ \mathtt{q_1}\ \cdots\ \mathtt{q_r}\ \mathtt{t_1}\ \cdots\ \mathtt{t_s}) \qquad \Gamma \vdash [\mathtt{P} \mid \mathtt{I}\ \mathtt{q_1}\ \cdots\ \mathtt{q_r}] \qquad (\Gamma \vdash \mathtt{f_i} : \{(\mathtt{c_{p_i}}\ \mathtt{q_1}\ \cdots\ \mathtt{q_r})\}^{\mathtt{P}})_{i=1..l}}{\Gamma \vdash \mathtt{case}(\mathtt{c}, \mathtt{P}, \mathtt{f_1} \mid \cdots \mid \mathtt{f_l}) : (\mathtt{P}\ \mathtt{t_1}\ \cdots\ \mathtt{t_s}\ \mathtt{c})} \quad \text{(T-Match)}$$

$[\mathtt{A} \mid \mathtt{B}]$ means the inner-most term of $\mathtt{A}$ is $\mathtt{B}$.

5 Reduction

5.1 Notation

- $\Gamma \vdash \mathtt{t} \rhd \mathtt{u} : \mathtt{t}$ reduces to $\mathtt{u}$ in Γ with one of the $\beta, \iota, \delta, \zeta$ reductions.
- $\Gamma \vdash \mathtt{t} \overset{*}{\rhd} \mathtt{u} : \Gamma \vdash \mathtt{t} \rhd \cdots \rhd \mathtt{u}$.
- $\mathtt{u} \equiv \mathtt{v} : \mathtt{u}$ and $\mathtt{v}$ are identical.

5.2 Conversion Rules

$$\Gamma \vdash ((\lambda \mathtt{x} : \mathtt{T}.\ \mathtt{t})\ \mathtt{u}) \ \rhd_\beta\ \mathtt{t}\{\mathtt{x}/\mathtt{u}\} \tag{β-Conv}$$

$$\Gamma \vdash \mathtt{x} \ \rhd_\delta\ \mathtt{t} \qquad \textit{if}\ (\mathtt{x} := \mathtt{t} : \mathtt{T}) \in \Gamma \tag{δ-Conv}$$

$$\Gamma \vdash \mathtt{let}\ \ \mathtt{x} := \mathtt{u}\ \ \mathtt{in}\ \ \mathtt{t} \ \rhd_\zeta\ \mathtt{t}\{\mathtt{x}/\mathtt{u}\} \tag{ζ-Conv}$$

$$\mathtt{case}((\mathtt{c}_{\mathtt{p_i}}\ \mathtt{q}_1\ \cdots\ \mathtt{q}_\mathtt{r}\ \mathtt{a}_1\ \cdots\ \mathtt{a}_\mathtt{m}), \mathtt{P}, \mathtt{f}_1 \mid \cdots \mid \mathtt{f}_\mathtt{n}) \ \rhd_\iota\ (\mathtt{f}_\mathtt{i}\ \mathtt{a}_1\ \cdots\ \mathtt{a}_\mathtt{m}) \tag{ι-Conv}$$

$$\frac{\Gamma \vdash \mathtt{t} : \forall \mathtt{x} : \mathtt{T}, \mathtt{U} \qquad \mathtt{x}\ \textit{fresh in}\ \mathtt{t}}{\mathtt{t}\ \rhd_\eta\ \lambda \mathtt{x} : \mathtt{T}.\ (\mathtt{t}\ \mathtt{x})} \tag{η-Exp}$$

Here's an example for ι-reduction:

```
match (S m) as s0 in nat return nat with
| S n => n
| O => O
end

(* ι-reduction *)
(fun (n:nat) => n) m.
```

Definition 1 (Convertibility)**.** $\mathtt{t}_1$ *and* $\mathtt{t}_2$ *are convertible iff there exists* $\mathtt{u}_1$ *and* $\mathtt{u}_2$ *such that* $\Gamma \vdash \mathtt{t}_1 \overset{*}{\rhd} \mathtt{u}_1$ *and* $\Gamma \vdash \mathtt{t}_2 \overset{*}{\rhd} \mathtt{u}_2$ *and either* $\mathtt{u}_1 \equiv \mathtt{u}_2$ *or they are convertible up to* η*-expansion.*

6 Safety Check

Naturally, we prefer error alerts right after inputting rather than let the system run till crash. Parser helps filter bad inputs, type check verifies the transaction between term and term, while neither of them guarantees termination.

In this part, we will demonstrate how to prevent non-halting definition from getting accepted into the system.

6.1 Fixpoint

The introduction of *Fixpoint* gives a wide range of flexibility of the program, without which recursive function can not formulate. On the other hand, the risk of non-terminating appears.

To avoid this, we put a strong constraint on recursive function, which is that it must descend on at least one argument.

Proposition 1 (Termination of Fixpoint)**.** *If a recursive function descends on at least one argument, it will always terminate in reduction.*

Definition 2 (Descending)**.** *Recursive function $f(x_1, x_2, \cdots, x_n)$ is descending on x_k if any occurrence of f in function definition has the form $f(y_1, y_2, \cdots, y_n)$, where y_k is inferior to x_k.*

Definition 3 (Inferior)**.** *In a function definition, x is inferior to y if $y \to_m y_1 \to_m \cdots \to_m y_s \equiv x$. $a \to_m b$ means a pattern match directly from a to b.*

Here is an example.

```
(* accepted *)
Fixpoint plus (n:nat)(m:nat) : nat :=
    match n as n0 in nat return (nat) with
    | O => m
    | S p => S (plus p m)
    end.
```

```
(* rejected *)
Fixpoint plus (n:nat)(m:nat) : nat :=
    match n as n0 in nat return (nat) with
    | O => m
    | S p => S (plus m p)
    end.
```

Implementation

Check every argument of the recursive function. For a specific argument, maintain a list of terms inferior to it during the process and verify every occurrence of the recursive function.

6.2 Inductive Type

The inductive type is actually a recursive definition, which is so powerful that we can create non-terminating program without actually defining a recursive function.

```
Inductive ill : Type :=
| malf : (ill -> ill) -> ill.

Definition extract (t:ill) : ill :=
    match t as t0 in ill return (ill) with
    | malf f => f t
    end.

extract (malf extract)  (* not terminating *)
```

Proposition 2 (Termination of Inductive Type)**.** *If the type of constructor of the inductive definition satisfies the positivity condition for the inductive type, it will always terminate in reduction.*

Definition 4 (Positivity)**.** *The type of constructor* $\mathtt{T}$ *satisfies the positivity condition for a constant* $\mathtt{X}$ *if*

- $\mathtt{T} \equiv (\mathtt{X}\ \mathtt{t_1}\ \cdots\ \mathtt{t_n})$ and $\mathtt{X}$ does not occur free in $\mathtt{t_i}$
- $\mathtt{T} \equiv \forall \mathtt{x} : \mathtt{U}, \mathtt{V}$ and $\mathtt{X}$ occurs only *strictly positively* in $\mathtt{U}$ and $\mathtt{V}$ satisfies *the positivity condition* for $\mathtt{X}$

Definition 5 (Strictly Positivity)**.** *The constant* $\mathtt{X}$ *occurs strictly positively in* $\mathtt{T}$ *if*

- $\mathtt{X}$ does not occur in $\mathtt{T}$
- $\mathtt{T} \rhd^* (\mathtt{X}\ \mathtt{t_1}\ \cdots\ \mathtt{t_n})$ and $\mathtt{X}$ does not occur in $\mathtt{t_i}$
- $\mathtt{T} \rhd^* \forall \mathtt{x} : \mathtt{U}, \mathtt{V}$ and $\mathtt{X}$ does not occur in $\mathtt{U}$ but occurs *strictly positively* in $\mathtt{V}$
- $\mathtt{T} \rhd^* (\mathtt{I}\ \mathtt{a_1}\ \cdots\ \mathtt{a_m}\ \mathtt{t_1}\ \cdots\ \mathtt{t_p})$, where $\mathtt{Ind}[m](\mathtt{I} : \mathtt{A} := \mathtt{c_1} : \forall \mathtt{p_1} : \mathtt{P_1}, \ldots \forall \mathtt{p_m} : \mathtt{P_m}, \mathtt{C_1}; \cdots ; \mathtt{c_n} : \forall \mathtt{p_1} : \mathtt{P_1}, \ldots, \forall \mathtt{p_m} : \mathtt{P_m}, \mathtt{C_n})$, and $\mathtt{X}$ does not occur in t_i, and the types of constructor $\mathtt{C_i}\{\mathtt{p_j}/\mathtt{a_j}\}_{\mathtt{j=1..m}}$ satisfies *the nested positivity condition* for $\mathtt{X}$

Definition 6 (Nested Positivity)**.** *The type of constructor* $\mathtt{T}$ *satisfies the nested positivity condition for a constant* $\mathtt{X}$ *if*

- $\mathtt{T} \equiv (\mathtt{I}\ \mathtt{b_1}\ \cdots\ \mathtt{b_m}\ \mathtt{u_1}\ \cdots\ \mathtt{u_p})$, where $\mathtt{I}$ is an inductive definition with m parameters and $\mathtt{X}$ does not occur in u_i
- $\mathtt{T} \equiv \forall \mathtt{x} : \mathtt{U}, \mathtt{V}$ and $\mathtt{X}$ occurs *strictly positively* in $\mathtt{U}$ and $\mathtt{V}$ satisfies *the nested positivity condition* for $\mathtt{X}$

7 Top Level

In top level, the system maintains the interaction with user and adds the user input, namely, *Axiom*, *Definition*, *Fixpoint*, and *Inductive Definition*, into the context.

It is also in this part, the inductive rule forms.

7.1 Acceptance of Command

7.1.1 Axiom

After parsing and checking, the command will be like

$$\texttt{Ax}\ \ \mathit{name}\ \mathit{term},$$

where *term* denotes the type of this axiom. Since it is an axiom, we do not have to (sometimes can not) build the corresponding term.

Just build the corresponding term as *Nothing* then put it into the context.

7.1.2 Definition

After parsing and checking, the command will be like

$$\texttt{Def}\ \ \mathit{name}\ \mathit{term2}\ \mathit{term1},$$

where *term2* is the type of *term1*.

Simply bind the name, term, type together and put it into the context.

7.1.3 Fixpoint

After parsing and checking, the command will be like

$$\texttt{Fix}\ \mathit{name}\ \ (\lambda f : \mathit{term1},\ \mathit{term2}),$$

where *term1* is the type of *term2* and it is a recursive function of *f*.

Since this recursive function has passed all the type check and safety check, we can safely use it without worrying termination problem. On the other hand, whether it is a *Fixpoint* definition will not influence any reduction, because the reduction always finds the term in context according to its index.

So simply remove the *Fixpoint* mark and put it into the context.

7.1.4 Inductive Definition

After parsing and checking, the command will be like

$$\texttt{Ind}\ \ \mathit{name}\ p\ \mathit{term2}\ \mathit{term1}\ \mathit{constructors},$$

where p is the number of parameters of the inductive type, *term1* is the type of the inductive definition, and *term2* is the corresponding term.

Apart from the ordinary operation, we also need to add inductive rule, which is actually a type theory view of mathematical induction. Since the proof of a claim becomes a term of certain type, the induction rule is a term offering inductive scheme.

For example,

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.

(* build inductive rule*)

fun (P:nat -> Type)(f:P O)(f0:forall (n:nat), P n -> P (S n))
    fix F (n:nat) : P n :=
        match n as n0 in nat return (P n0) with
        | O => f
        | S n0 => f0 n0 (F n0)
        end
:
forall (P:nat -> Type), P O ->
    (forall (n:nat), P n -> P (S n)) ->
        forall (n:nat), P n
```

The intuition here is that for a proposition `P`,

- it is true on `O`;
- if it is true on `n`, then it is true on `S n`.

Then it is true on all term of `nat` type, which is reasonable according mathematical induction.

Basically, to build such term, we should build weakened assumptions for all constructors first, like the `f`, `f0` above. After that, the final proposition which applies to all the terms of such inductive type shall come out, like the `F` above.

Every occurrence of the inductive type on the constructors demands a verification of the proposition, which explains why for constructor `S : nat -> nat`, which depends on a `nat` term, the weakened assumption is `f0 : forall (n:nat), P n -> P (S n)`.

The reason why the inductive rule of `nat` requires *Fixpoint* is that some constructors of it rely on the term of type `nat`. Here is a case which do not need recursive function.

```
Inductive eq (T : Type) (x : T) : T -> Type :=
| eq_refl : eq T x x.

(* build inductive rule*)

fun (T:Type) (x:T) (P:forall (t:T) (_:eq T x t), Type)
    (f:P x (eq_refl T x)) (t:T) (e:eq T x t) =>
        match e as e0 in eq _ _ a0 return (P a0 e0) with
        | eq_refl _ _ => f
        end
:
forall (T:Type) (x:T) (P:forall (t:T) (_:eq T x t), Type)
    (f:P x (eq_refl T x)) (t:T) (e:eq T x t),
        P t e
```

Sadly, we have to admit that because of the lack of references, time, and energy, the induction rule in our system is not complete. The inductive definition acceptable to our system must satisfy:

Assume the inductive type is $\mathtt{A}_1 \to \mathtt{A}_2 \to \cdots \mathtt{A}_n$, then $\mathtt{A}_k$ must be any one of the following

- A ordinary term, like `Type`, `T`.

- An application, like `P n`.
- An inductive type, like `nat`, `eq T x y`.

Others like product type `U -> V` is not supported.

7.2 Requests to Environment

7.2.1 `Print` *ident*

This command displays on the screen information about the declared or defined object referred by *ident*.

7.2.2 `Check` *term*

This command displays the type of *term*.

7.3 Top Loop

The main work flow of our top level loop:

Reading raw input

The MiniProver will read the user's input until a dot ('.'), and any further input in the same line will be abandoned.

Parsing

The raw input will be parsed without nameless representation.

Here is an example, the raw input

```
(* raw input *)
Fixpoint plus (n:nat) (m:nat) : nat :=
match n as n0 in nat return nat with
| O => m
| S n0 => S (plus n0 m)
end.
```

will be parsed as the AST

```
Fix "plus"
  ( TmFix (-1)
    ( TmLambda "plus"
      ( TmProd "n" ( TmVar "nat" )
        ( TmProd "m" ( TmVar "nat" ) ( TmVar "nat" )))
      ( TmLambda "n" ( TmVar "nat" )
        ( TmLambda "m" ( TmVar "nat" )
          ( TmMatch (-1) ( TmVar "n" ) "n0" [ "nat" ]
            ( TmVar "nat" )
            [ Equation [ "O" ] ( TmVar "m" )
            , Equation [ "S", "n0" ]
              ( TmAppl
                [ TmLambda "n" ( TmVar "nat" )
                  ( TmAppl [ TmVar "S", TmVar "n" ])
                , TmAppl [ TmVar "plus", TmVar "n0", TmVar "m" ]])])))))
```

Duplicate global name checking

After parsing, we can get the name of the input command, and the name should not be the same with any defined or declared object in the environment.

Name checking

Before building the nameless representation, there should be no unbounded name in the AST.

Nameless representation building

If all names are bounded, we can build the nameless representation. The variable pointed to a type constructor or a term constructor will be unfolded to its functional representation.

Here is an example, the nameless AST will be built for the previous AST:

```
Fix "plus"
( TmFix (-1)
  ( TmLambda "plus"
    ( TmProd "n" ( TmIndType "nat" [])
      ( TmProd "m" ( TmIndType "nat" []) ( TmIndType "nat" [])))
    ( TmLambda "n" ( TmIndType "nat" [])
      ( TmLambda "m" ( TmIndType "nat" [])
        ( TmMatch 0 ( TmRel "n" 1 ) "n0" [ "nat" ] ( TmIndType "nat" [])
          [ Equation [ "O" ] ( TmRel "m" 0 )
          , Equation [ "S", "n0" ]
            ( TmAppl
              [ TmLambda "n" ( TmIndType "nat" [])
                ( TmConstr "S" [ TmRel "n" 0 ])
              , TmAppl
                [ TmRel "plus" 3
                , TmRel "n0" 0
                , TmRel "m" 1 ]])])))))
```

Positivity Checking (Inductive Definition Only)

For an inductive definition, after building it's nameless representation, the positivity could be checked.

Termination Checking

All subterms with fixpoint definitions will be checked if they are terminating. After checking, annotations for the indices of decreasing variables will be added to the AST.

Type checking

Before actually dealing with the command, the top level will check if it's a well-typed command.

Processing the command

The definitions and declarations will be processed as described before. And an assertion will lead to the proof editing mode.

8 Proof Handling

In the proof editing mode, the user can use the tactics to deal with logical reasoning, and can also use some other specialized commands to deal with the proof environment.

In our implementation, the proof procedure is organized as a tree. Initially, the tree consists only the theorem itself as the tree root and the only leaf. Every time a tactic is applied, either the current leaf is expanded to an internal node or the proof procedure of the subtree is completed, then the next leaf of the tree will be focused. The proof procedure is completed when all of the tree leaves are proven, then the whole proof object will be built recursively.

To each subgoal is associated a number of hypotheses called the *local context* of the goal. Initially the local context contains nothing, it is enriched by the use of certain tactics.

When a proof is completed, the message `No more subgoals` is displayed. One can then register this proof as a defined constant in the environment. Because there exists a correspondence between proofs and terms of λ-calculus, known as the *Curry-Howard isomorphism*, the MiniProver stores proofs as terms of *CIC*. Those terms are called *proof terms*.

8.1 `Theorem`

The proof editing mode is entered by asserting a theorem.

8.2 `Proof`

This command is a noop which is useful to delimit the sequence of tactic commands which start a proof, after a `Theorem` command.

`Theorem` *ident {binder} : term.*

8.3 `Qed`

This command is available in interactive editing proof mode when the proof is completed. Then `Qed` extracts a proof term from the proof script, switches back to the top level and attaches the extracted proof term to the declared name of the original goal.

8.4 `Admitted`

This command is available in interactive editing proof mode to give up the current proof and declare the initial goal as an axiom.

8.5 `Abort`

This command cancels the current proof development, switching back to the top level.

8.6 `Undo`

This command cancels the effect of the last command. Thus, it backtracks one step.

8.7 `Restart`

This command restores the proof editing process to the original goal.

9 Tactic

Although proofs can be automatically checked given the system above, it requires user to input the whole term of the specific type, which is tedious, exhaustive, mostly anti-intuitive.

So in this part, we introduced some tactics into the system to assist users in better writing the proof. With the help of tactics, the proof becomes more human-readable and intuitive.

9.1 *apply*

This tactic applies to any goal. The purpose of this tactic is to extract premises of current goal.

Usage

- *apply* `t.` : `t` is a term of a type whose conclusion is the goal.
- *apply* `t` *in* `H.` : `t` is a term of a type whose condition is `H`.

Implementation

The tactic *apply* tries to match the current goal against the conclusion of the type of term. If it succeeds, it returns as many subgoals as the number of non-dependent premises of the type of term.

Here is an example:

```
n : nat
m : nat
o : nat
e : eq nat n m -> eq nat m o -> eq nat n o
(* Current Goal *)
eq nat n o

(* ⇓ apply n ⇓ *)

n : nat                                          n : nat
m : nat                                          m : nat
o : nat                                          o : nat
e : eq nat n m -> eq nat m o -> eq nat n o       e : eq nat n m -> eq nat m o -> eq nat n o
(* SubGoal 1 *)                                  (* SubGoal 2 *)
eq nat m o                                       e : eq nat n m
```

Because `e` has a type of `eq nat n m -> eq nat m o -> eq nat n o` and the goal is `eq nat n o`, «««< HEAD when applied with `e`, the goal equals to the conclusion (inner-most term) of the type of the term. ======= when applied with `n`, the goal is convertible to the inner-most part of the type of `e`. »»»> 8f8ff3f60ce84e8c747ee454d6ccc529ab657bd9 And all of the non-dependent premises of the type of term become subgoals.

The tactic *apply* `tm` *in* `H` tries to match the hypothesis `H` against the condition of the type of `tm`. If it succeeds, the hypothesis `H` will be replaced by the conclusion of `tm`

```
n : nat
m : nat
o : nat
H : eq nat n m
e : eq nat n m -> eq nat m o -> eq nat n o
(* Current Goal *)
```

```
eq nat m o -> eq nat n o

(* ⇓ apply e in H ⇓ *)
n : nat
m : nat
o : nat
H : eq nat m o -> eq nat n o
e : eq nat n m -> eq nat m o -> eq nat n o
(* SubGoal 1*)
eq nat m o -> eq nat n o
```

Because `e` has a type of `eq nat n m -> eq nat m o -> eq nat n o` and `H` is `eq nat n m`, when applied with `e`, the `H` equals to the condition of the type of the term. And `H` was replaced by the conclusion of `e`.

9.2 *destruct*

When *destruct* is applied on a term, this term must be inductive type. Then *destruct* will divide current goal into several classified subgoals based on the constructors of the inductive type. During this process, no induction hypothesis is generated by *destruct*; the *destruct* uses `match` to handle classified discussion.

Usage

- *destruct* `t.` : `t` is a term of inductive type.

Implementation

This tactic not only divides current goal based on the constructors of the inductive type, but requires the system to handle several equivalent relations proposed by the constructors as well.

Here is an example:

```
n, m : nat
e : eq nat n m                                  n : nat
f : eq nat m m        (* destruct e => *)        f : eq nat n n
(* Current Goal *)                              (* Current Goal *)
eq nat m n                                      eq nat n n
```

From the constructor `eq_refl : eq T x x` and the type of `e : eq nat n m`, we have `m = n`, so any occurrence of `m` should be replaced by `n`.

If some term depends on the destructed term, we should rewrite it.

```
n, m : nat                                  n : nat                 n : nat
e : eq nat n m                              e0 : eq nat n O         e0 : eq nat n (S m0)
f : eq nat m m        (* destruct m => *)   f0 : eq nat O O         f0 : eq nat (S m0) (S m0)
(* Current Goal *)                          (* SubGoal 1 *)          (* SubGoal 2 *)
eq nat m n                                  eq nat O n              eq nat (S m0) n
```

The two cases above are viewed from user's perspective. On the other hand, in the system we also need to construct the term for back substitution.

```
n, m : nat
e : eq nat (S n) (S m)
(* Current Goal*)
eq nat (S m) (S n)

(* ⇓ destruct (S n) ⇓ *)

(* The Proof Object *)
(fun (n:nat)(m:nat)(e:eq nat (S n) (S m)) =>
    match (S n) as n0 in nat return (eq nat n0 (S m) -> eq nat (S m) n0) with
    | O => fun e0 : eq nat O (S m) => (* SubGoal 1*)
    | S n0 => fun e0 : eq nat (S n0) (S m) => (* SubGoal 2*)
    end e)
```

After building the *return type*, we can build the branches by applying *type of branch*.

9.3 *exact*

This tactic checks if the input term is actually of the goal's type. If so, the proof of the current goal is finished and the constructed proof object is exactly the input term.

Usage

- *exact* `t.` : `t` is a term of the goal's type.

Implementation

It's easy to implement. Here is an example:

```
m : nat
(* Current Goal *)  (* exact (eq_refl nat m) => *)  (* No More Goals *)
eq nat m m
```

9.4 *exists*

This tactic handles the exists qualifier. In our system, $\exists(\mathtt{x} : \mathtt{T}), \mathtt{P}\ \mathtt{x}$ is represented by `ex T P`.

Here is the definition of `ex`.

```
Inductive ex (A:Type) (P:A -> Type) : Type :=
| ex_intro : forall (x:A), P x -> ex A P.
```

The proof of $\exists(\mathtt{x} : \mathtt{T}), \mathtt{P}\ \mathtt{x}$ needs the user to provide a value `x` and prove that it satisfies the predicate `P`.

Usage

- *exists* `t.` : If the goal is `ex` $(\mathtt{x} : \mathtt{T}), \mathtt{P}\ \mathtt{t}$, and `t` is a term of type `T`.

Implementation

The tactic automatically generates the subgoal `P x`.

```
(* Current Goal *)
ex nat (fun (n:nat) => eq nat (S (S O)) (plus n n))

(* ⇓ exists S O ⇓ *)

(* SubGoal *)
eq nat (S (S O)) (plus (S O) (S O))
```

The proof object built is

```
ex_intro nat (fun (n:nat) =>
                eq nat (S (S O)) (plus n n)) (S O) (* SubGoal *)
```

9.5 *induction*

This tactic applies to any goal. The argument term must be of inductive type and the tactic *induction* generates subgoals, one for each possible form of term, i.e., one for each constructor of the inductive type.

Usage

- *induction* `t.` : `t` is a term of some inductive type.

Implementation

The tactic generates the predicate by the goal and automatically generates the subgoals for every constructor of the inductive type. The equivalence relations introduced by the constructors are also handled.

If the term is a hypothesis, *induction* will find the dependencies, and try to move them and replace the subterms to solve the dependencies. And then erase the original hypothesis.

```
n : nat
(* Current Goal *)
eq nat (plus n O) n

(* ⇓ induction n ⇓ *)

(* SubGoal 1 *)
eq nat (plus O O) O

n0 : nat
e : eq nat (plus n0 O) n0
(* SubGoal 2 *)
eq nat (plus (S n0) O) (S n0)
```

After *induction*, all we need to prove is the property holds for `O` and for any `n`, if `P n` holds, then the two subgoals are generated and `n` is replaced by the constructors.

The case above is viewed from user's perspective. On the other hand, in the system we also need to construct the term for back substitution.

```
n : nat
(* Current Goal*)
eq nat (plus n O) n

(* ⇓ induction n ⇓ *)

(* The Proof Object *)
(fun (n:nat) =>
    nat_rect (fun (n0:nat) => eq nat (plus n0 O) n0)
    (* SubGoal 1 *)
    (fun (n0:nat) (e:eq nat (plus n0 O) n0) => (* SubGoal 2 *))
    n)
```

9.6 *intro*

If the current goal is a dependent product ∀x : T, U, *intro* puts x : T in the context, and the new subgoal is U.

If the current goal is a non-dependent product ∀_ : T, U, *intro* renames _ into s not in T, U and context. Then put s : T in the context, and the new subgoal becomes U.

Usage

- *intro* . : introduce the outermost argument of current goal and rename them automatically.
- *intro* a_1 a_2 $\cdots$ a_n. : introduce n outermost arguments of current goal and try to rename them into $a_1, a_2, \cdots, a_n$.

Implementation

Since the context automatically preserves the index, we can safely move the arguments out into the context without worrying about the index.

If the argument is dependent in either the conclusion or some hypotheses of the goal, the argument is replaced by the appropriate constructor form in each of the resulting subgoals and induction hypotheses are added to the local context.

Here is an example:

```
                                                    P : Type
(* Current Goal *)                (* intro P => *)    (* SubGoal *)
forall (x:Type) (f:False), x                        forall (f:False), P
```

9.7 *intros*

If the current goal is a dependent (or non-dependent) product with m arguments, namely, $\forall x_1 : T_1$, $\forall x_2 : T_2$, $\cdots$ $\forall x_m : T_m$, U where U is not a product type. Then *intros* will put these m arguments into the context and rename automatically if necessary.

Usage

- *intros*. : introduce all the argument of current goal and rename them automatically.

Implementation

Keep calling *intro* until all the goal has no arguments.

9.8 *left right*

These tactics apply to a goal that is of the disjunction type. Then we can proof one of the two sides to solve the goal.

Usage

- *left*.
- *right*.

Implementation

The disjunction type in our system is defined as follows:

```
Inductive or (A:Type) (B:Type) : Type :=
| or_introl : A -> or A B
| or_intror : B -> or A B.
```

The tactic *left* builds the proof object with the `or_introl` constructor. The tactic *right* builds the proof object with the `or_intror` constructor. Here is an example:

```
(* Current Goal *)
or (eq nat O O) (eq nat (S O) (S O))

(* ⇓ left ⇓ *)

(* The Proof Object *)
or_introl (eq nat O O) (eq nat (S O) (S O)) (* SubGoal 1 *)
```

9.9 *split*

This tactic applies to a goal that is of the conjunction type. Then we need to proof the both sides to solve the goal.

Usage

- *split*.

Implementation

The conjunction type in our system is defined as follows:

```
Inductive and (A:Type) (B:Type) : Type :=
| conj : A -> B -> and A B.
```

The proof object is built with the `conj` constructor. Here is an example:

```
(* Current Goal *)
and (eq nat O O) (eq nat (S O) (S O))

(* ⇓ split ⇓ *)

(* The Proof Object *)
conj (eq nat O O) (eq nat (S O) (S O)) (* SubGoal 1 *) (* SubGoal 2 *)
```

9.10 *reflexivity*

This tactic applies to a goal that has the form `eq T t u`, it check that `t` and `u` are convertible and then solves the goal.

Usage

- *reflexivity*.

Implementation

This tactic simply checks if the terms are convertible and build the proof object with `eq_refl` constructor.

```
n : nat
(* Current Goal *)
eq nat n (plus O n)

(* reflexivity *)

(* The Proof Object *)
fun (n:nat) => eq_refl nat n
```

9.11 *rewrite*

This tactic applies to any goal. The type of term must have the form `eq T t u`. Then the subterms `t` in the goal will be rewritten by `u`.

Usage

- *rewrite* `H.` : `H` should have the form `eq T t u`, all the subterms `t` in the goal will be replaced by `u`.
- *rewrite* `<- H.` : `H` should have the form `eq T t u`, all the subterms `u` in the goal will be replaced by `t`.
- *rewrite* [`<-`/`->`] `H` *in* `H1.` : `H` should have the form `eq T t u`, for the `->` case, all the subterms `t` in `H1` will be replaced by `u`.

Implementation

This tactic builds the proof object with `eq_rect` or `eq_rect_r`.

```
a : nat                                          a : nat
b : nat                                          b : nat
e : eq nat a b                                   e : eq nat a b
(* Current Goal *)      (* rewrite e => *)      (* SubGoal *)
eq nat b a                                       eq nat b b

(* Proof Object *)
fun (a:nat) (b:nat) (e:eq nat a b) =>
  (fun (e0:eq nat b b) =>
    eq_rect_r nat b (fun (n:nat) (e1:eq nat b n) =>
      eq nat b n) e0 a e) (* SubGoal *)
```

9.12 *simpl*

This tactic applies to any goal, it tries to reduce a term to something still readable instead of fully normalizing it. It performs a sort of strong normalization with a key difference:

> it unfold a constant if and only if it leads to a ι-reduction, i.e., reducing a `match` or unfolding a `Fix`.

Usage

- *simpl*.
- *simpl in* `H`. : `H` is a hypothesis.

Implementation

The tactic first applies $\beta\iota$-reduction, then it expands constants and tries to reduce further using $\beta\iota$-reduction, but when no ι rule is applied after unfolding, ι-reductions are not applied. For instance, trying to use *simpl* on `eq nat (plus n O) n` changes nothing.

```
n : nat                                          e : nat
(* Current Goal *)          (* simpl => *)      (* Current Goal *)
eq nat (plus n O) (plus O n)                     eq nat (plus n O) n
```

The tactic can also be applied to the hypotheses, and the same reduction rules applies.

```
n : nat                                          n : nat
e : eq nat (plus n O) (plus O n)                 e : eq nat (plus n O) n
(* Current Goal *)              (* simpl in e => *)   (* Current Goal *)
eq nat (plus n O) n                              eq nat (plus n O) n
```

The tactic would never change the current built proof object and won't modify the proof objects built after it. It is transparent in the proof tree.

9.13 *exact*

This tactic checks if the input term is actually of the goal's type. If so, the proof of the current goal is finished and the constructed proof object is exactly the input term.

Usage

- *exact* `t`. : `t` is a term of the goal's type.

Implementation

It's easy to implement. Here is an example:

```
m : nat
(* Current Goal *)  (* exact eq_refl nat m => *)  (* No more goals *)
eq nat m m
```

9.14 *symmetry*

This tactic applies to a goal that has the form `eq T t u` and changes it into `eq T u t`.

Usage

- *symmetry*.

Implementation

This tactic builds the proof object use the pre-defined theorem `eq_sym`, here is an example for the proof object building.

```
m : nat
n : nat
e : eq nat m n
(* Current Goal*)
eq nat n m

(* ⇓ symmetry ⇓ *)

(* The proof object *)
fun (m:nat) (n:nat) (e:eq nat m n) =>
    eq_sym nat m n (* SubGoal *)
```

9.15 *unfold*

This tactic applies to any goal, the argument should denote a defined constant (i.e., not a hypothesis or a axiom). The tactic applies the δ rule to each occurrence of the constant in the current goal and then replaces it with its $\beta\iota$-normal form.

Usage

- *unfold* `i.` : `i` is an identifier.
- *unfold* `i` *in* `H.` : `i` is an identifier and `H` is a hypothesis.

Implementation

It finds the occurrences of the constant and applies the δ rule, the nameless index is automatically preserved.

10 Usage

10.1 System Requirements

- Linux or macOS, currently not tested on Windows.
- The Haskell Tool Stack

10.2 Compiling

Thanks to the haskell tool stack, the project is easy to build in one command.

```
stack build
```

10.3 Execution

The project can be executed without installing to the system environment:

```
stack exec mini-prover-exe
```

It will read the `./libs/Init/Prelude.v` for pre-defined objects.

For verbose output:

```
stack exec mini-prover-exe -- -v[0-3]
```

10.4 Demos

Some demo code tested in our system can be found at `./demo/demo.v`

10.5 Unit Test

To run the unit tests, just run

```
stack test
```

11 Conclusion

In conclusion, we have built a proof assistant, that can automatically check the validity of the user-inputted proof. This project demonstrates the power of type system and type theory, which is indeed fascinating. We believe such proof assistant will ultimately be developed and employed in all kinds of proof checking tasks.

Despite of these, we have to admit that due to the limited time and energy, this is the best we can offer. It is likely that some bug remains unspotted in the code.

12 Division of Labour

This table roughly summarizes everyone's part in this project. But the division is not that restrict.

Work	Zhenwen Li	Sirui Lu	Kewen Wu
Syntax		√	
Parser		√	
Typing	√		
Reduction		√	
Safety Check			√
Top Level		√	√
Proof Handling		√	
Tactic	√	√	√
Unit Test	√	√	√
Documentation	√	√	√

References

[1] Adam Chlipala. Certified programming with dependent types, 2011.

[2] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 8.7.2 edition, February 2018.

[3] Rob P.. Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[4] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.

Appendix

In this part, several interesting constructions are listed, while others are left in `./libs/Init/Prelude.v`.

Boolean

```
Inductive True : Type :=
| I : True.

Inductive False : Type :=.

Inductive and (A : Type) (B : Type) : Type :=
| conj : A -> B -> and A B.

Inductive or (A : Type) (B : Type) : Type :=
| or_introl : A -> or A B
| or_intror : B -> or A B.
```

Logic

```
Definition If_then_else (P : Type) (Q : Type) (R : Type) : Type :=
    or (and P Q) (and (not P) Q).

Definition not (A : Type) : Type :=
    A -> False.

Definition iff (A : Type) (B : Type) : Type :=
    and (A -> B) (B -> A).

Inductive ex (A : Type) (P : A -> Type) : Type :=
| ex_intro : forall (x : A), P x -> ex A P.

Inductive eq (A : Type) (x : A) : A -> Type :=
| eq_refl : eq A x x.
```

Natural Number

```
Inductive nat : Type :=
| O : nat
| S : nat -> nat.

Fixpoint plus (n : nat) (m : nat) : nat :=
    match n as n0 in nat return nat with
    | O => m
    | S n1 => S (plus n1 m)
    end.

Fixpoint multiply (n : nat) (m : nat) : nat :=
    match n as n0 in nat return nat with
    | O => O
    | S n1 => plus m (multiply n1 m)
    end.
```

```
Fixpoint power (n : nat) (m : nat) : nat :=
    match m as m0 in nat return nat with
    | O => S O
    | S m1 => multiply n (power n m1)
    end.
```

List

```
Inductive list (T : Type) : Type :=
| nil : list T
| cons : T -> list T -> list T.

Inductive iliist (T : Type) : nat -> Type :=
| inil : ilist T O
| icons : forall (n : nat), T -> ilist T n -> ilist T (S n).
```