1 Introduction

MetaCoq [1] is a project providing tools for manipulating Coq terms and developing certified plugins (i.e. translations, compilers or tactics) in Coq.

In MetaCoq one can write meta-programs, such as a program which derives the induction principle of any inductive type, one can then prove that the result of the program is well-typed or has other semantic properties.

Instead of proving these properties after the fact, it would be interesting to immediately provide such guarantees at the time of implementing the meta-programs, e.g. via a type of well-typed terms or well-scoped Coq syntax.

This project aims at using several approaches to define such types with guarantees for the user, compare and contrast them, and implement case studies using them.

Which can be used as basis for meta-programming of tactics and commands, to prove meta-theoretic properties of the type theory of Coq such as subject reduction, and to verify programs crucial in the implementation of Coq such as type checking or extraction.

1.1 Template-Coq

At the center of MetaCoq is the Template-Coq quoting library for Coq. It takes Coq terms and constructs a representation of their syntax tree as an inductive data type. The representation is based on the kernel's term representation.

For example, the MetaCoq quotes the function (`fun (x:nat) ⇒ x`) to

```
tLambda {| binder_name := nNamed "x"; binder_relevance := Relevant |} (*name of variable*)
        (tInd
           {|
             inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
             inductive_ind := 0
           |} [])   (*type of variable*)
        (tRel 0) (*body*)
     : term
```

The MetaCoq also provides tool to unquote the term above back to (`fun (x:nat) ⇒ x`).

In the example above, the body of the lambda abstraction is represented by `tRel 0`, which is exactly the de Bruijn index[2]. The quotation of MetaCoq uses pure de Bruijn indices as the binder. The de Bruijn index is used in the kernel of Coq, it is one reason why MetaCoq chooses this binder.

1.2 de Bruijn index

De Bruijn index is a tool invented for representing terms of lambda calculus without naming the bound variables. Each de Bruijn index is a natural number that represents an occurrence of a variable in a λ-term, and denotes the number of binders that are in scope between that occurrence and its corresponding binder. For example, the lambda term $\lambda x \lambda y.y$ is represented by $\lambda\lambda 0$, where the 0 represents the closest binder.

Let us look at a more complicated example, for the inductive type:

```
Inductive vec (A:Type) : nat → Type :=
 | nil : vec A (Nat.0)
 | cons (a:A) : forall (n:nat), vec  A  n → vec A (S n).
```

Its quotation in MetaCoq looks like (&_ means the de Bruijn index):

```
Inductive vec (_:Type) : nat → Type :=
 | nil : &1 &0 (Nat.O)
 | cons : &0 → forall (_:nat), &3 &2 &0 → &4 &3 (S &1).
```

For the underlined "vec" above, there are 4 binders that can be referenced, i.e. n, a, A, vec, their de Bruijn indices are 0, 1, 2, 3, since here we need to represent vec, it should be represented by &3. For the same reason, the "A" noted above is represented by &2.

Using MetaCoq to write meta-programs, the user does not only need to know how the induction principle should be formed, but also must have a good command of de Bruijn index. In order to write meta-programs by MetaCoq, the calculation of de Bruijn indices can be complicated and error-prone.

For example, we can write a program to derive the induction principle of any inductive type. The induction principle of the vector defined above is:

```
forall (A:Type), forall (P: forall (n:nat), vec A n → Prop),
  P 0 (nil A) → (forall (a:A) (n:nat) (v:vec A n), P n v → P (S n) (cons A a n v)) →
    forall (n:nat) (x:vec A n), P n x.
```

Whose quotation looks like:

```
forall (_:Type), forall (_: forall (_:nat), vec &1 &0 → Prop),
  &0 (Nat.O) (nil &1) →
  (forall (_:&2) (_:nat) (_:vec &4 &0), &4 &1 &0 → &5 (S &2) (cons &6 &3 &2 &1)) →
    forall (_:nat) (_:vec &4 &0), &4 &1 &0.
```

If we compute each de Bruijn index directly, for the last "P" appears in the induction principle, its de Bruijn index should be equal to (number of type constructors) + (number of indices of type) + 1, i.e. $2 + 1 + 1 = 4$. Similar calculation is inevitable for each de Bruijn index. If we just compute each index in this nutural way, it will be error-prone and laborious.

1.3 First approach

Due to the cumber of calculation of de Bruijn index, one aspect of this project is to propose an approach to avoid directly using the de Bruijn index during programming MetaCoq and so that reduces the difficulty of meta-programming through MetaCoq.

2 Approach

Currently, this approach is limited on generating function/type from the inductive type definition. We will take the generation of (type of) induction principle of inductive type as an example to illustrate this approach. With this approach, the user needs to know very little knowledge of de Bruijn index.

2.1 Basic idea

The idea to avoid direct calculation of De Bruijn index is to carry a local information during the generation, which includes the correct De Bruijn indices of some binders and other important information, the local information will be updated implicitly or explicitly during the generation.

In the subsection below, some important type structures and several necessary functions will be explained.

2.2 Documentation

The type `term` below is the `term` defined in the MetaCoq, i.e. `MetaCoq.Template.Ast.term`.

2.2.1 base

```
Inductive information : Type :=
  | information_list (l : list (BasicAst.context_decl term))
  | information_nat (n : nat).

Record infolocal : Type := mkinfo {
  renaming : list (BasicAst.context_decl nat);
  info : list (string * information) ;
  info_source : list (string * information) ;
  kn : kername; }.
```

The local information. For simplicity of explication, we will regard `renaming`, `information_list` as a list of natural numbers in the rest.

```
Inductive saveinfo:=
  | Savelist (s:string)
  | Saveitem (s:string)
  | NoSave.
```

Indicator used in some function, indicates whether new information will be saved into the `info` field of local information and in which format the new information will be saved.

```
Definition geti_info (na:string) (e:infolocal) (i:nat) : term
```

Get the `information_list` named `na` of `e.(info)`, and get its ith element `k`, return `tRel k`.

```
Definition rels_of (na:string) (e:infolocal): list term
```

Get the `information_list` named `na` of `e.(info)`, reverse and transform it to a list of `tRel _` term.

```
Definition rel_of (na:string) (e:infolocal): term
```

Get the `information_nat` named `na` of `e.(info)`, and get its value `k`, return `tRel k`.

2.2.2 inductive type

Some functions for the case when the source is the inductive type definition.

```
Definition make_initial_info (kn:kername) (ty:mutual_inductive_body):infolocal
```

Initialize the local information.

```
Definition is_recursive_call_gen (e:infolocal) (i:nat) : option nat
```

For a local information `e`, by checking `e.(info_source)`, see whether (`tRel i`) refers to the type name of one inductive body of the source. In the example above of $\texttt{vec}_{1.2}$, the underlined "vec" refers exactly to the type name.
If (`tRel i`) refers to the type name of the kth inductive body, return `Some k`; Otherwise, return `None`.

```
Definition type_rename_transformer (e:infolocal) (t:term) : term
```

Renaming transformation, transform the term `t` according to the renaming map. i.e. for each subterm (`tRel i`) of `t`, use the former function to check if it refers to the type name, if it is the case, maps it to the term of the type (a tInd term); otherwise, maps it to (`geti_rename e i`).

2.2.3 term generation

```
Definition mktProd (saveinfo:saveinfo) (na:aname) (e:infolocal)
    (t1:term) (t2:infolocal → term): term
```

Produce a Prod term, namely (`forall (na:t1), _`).

- `saveinfo`: whether save the information of new variable into `e`
- `na`: the aname of the new variable
- `e`: the local information
- `t1`: the type of variable
- `t2`: the body (need to be fed with a local information)

```
Definition kptProd (saveinfo:saveinfo) (na:aname) (e:infolocal)
    (t1:term) (t2:infolocal → term): term
```

Similar to the former one, but for different usage.

```
Definition it_kptProd (saveinfo:option string) (ctx:context) (e:infolocal)
    (t: infolocal → term): term
```

Iterative version of `kptProd`. When `ctx` ≈ [`vk; ... v2; v1`], the result will be like: (`forall (_:v1) (_:v2) ... (_:vk), _`). (need to transform these types by renaming transformation)

- `saveinfo`: whether save the information of the new variables into `e`
- `ctx`: the list of the new variables
- `e`: the local information
- `t`: the body (need to be fed with a local information)

```
Definition it_mktProd (saveinfo:option string) (ctx:context) (e:infolocal)
    (t: infolocal → term) :term
```

iterative version of `mktProd`.

▶ Remark 1. How to use `mktProd`, `kptProd` ?

Simply speaking, when creating a Prod term,

- use (`kptProd saveinfo na e t1 t2`) if `na` refers to a binder, or say a term that could be referenced(by `tRel _`) in the source
- otherwise, or say when the new variable does not occur in the source, use `mktProd`

For example, for a general inductive type:

`Inductive T (P_1:Param_1) ... (P_k:Param_k) : Ind_1 → ... → Ind_m :=...`

The type of its induction principle is:

```
forall (P_1:Param_1) ... (P_k:Param_k),
  forall (P: forall (i_1:Ind_1) ... (i_m:Ind_m): T P_1 ... P_k i_1 ... i_m → Prop), ......
```

When handle the parameters, since the parameters can be referenced in the source, use `kptProd`, but for `i_1`, ... `i_m` which do not occur in the source, use `mktProd`. See more concrete code in the section of case study.

- Similar functions for `Lambda` instead of `Prod` as follow:

```
Definition mktLambda (saveinfo:saveinfo) (na:aname) (e:infolocal)
    (t1:term) (t2:infolocal → term): term
Definition kptLambda (saveinfo:saveinfo) (na:aname) (e:infolocal)
    (t1:term) (t2:infolocal → term): term
Definition it_kptLambda (saveinfo:option string) (ctx:context) (e:infolocal)
    (t: infolocal → term): term
Definition it_mktLambda (saveinfo:option string) (ctx:context) (e:infolocal)
    (t: infolocal → term): term
```

2.3 Implementation

In this section, we will explain how to implement several important functions and type structures, and the idea behind that. Without notice, we will always consider the generation of induction principle of inductive type, we will use the word "source" to indicate the definition of inductive type, and the "target" to mean what we are going to generate.

2.3.1 local information

Remind the definition of local information:

```
Record infolocal : Type := mkinfo {
  renaming : list (BasicAst.context_decl term);
  info : list (string * information) ;
  info_source : list (string * information);
  kn : kername;
}.

Inductive information : Type :=
  | information_list (l : list (BasicAst.context_decl nat))
  | information_nat (n : nat).
```

The field `info` is designed to save the de Bruijn indices (in the target environment). For example, we can save a binder as (`name`, `information_nat` 0) into the `info` once, then if we update the local information correctly during the term generation, later at the time when we want to refer to this binder, we can just use (`rel_of name e`) to get it. The typical `information_list` includes parameters, indices, arguments...

The field `info_source` is used to save the de Bruijn indices of binders in the source environment. For example, save the de Bruijn index of the type, in our example of vec $_{1.2}$, we need to save the de Bruijn index of the type name "vec".

When using this approach to generate term from an inductive type, the generation process can be seen as a consecutive process of reading the source and writing the target, usually we do both at the same time. The `info` should be updated every time we produce a binder in the target, the `info_source` should be updated every time we read a binder in the source.

The `renaming` can be seen as a map (of de Bruijn index) from source to target, for simplicity, in the explication below, we regard `renaming`, `information_list` simply as a list of natural numbers.

The ith (begin with 0th) element of **renaming** is **t** means: the (**tRel i**) in the source environment should be expressed by **t** in the target environment. See the explication below:

Let us look at the example to illustrate the meaning of **renaming** and how should it change during the generation. For `vec` $_{1.2}$, its induction principle is:

```
forall (A:Type), forall (P: forall (n:nat), vec A n → Prop),
  P 0 (nil A) → (forall (a:A) (n:nat) (v:vec A n), P n v → P (S n) (cons A a n v)) →
    forall (n:nat) (x:vec A n), P n x.
```

When generating the induction principle, the generation can be visualized. For each step, • in the source is just after the term that we read at this step, • in the target is after the term generated at this step, the **renaming**, **info**, **info_source** show the value of these fields at the end of this step:

```
Source:
    Inductive vec • (A:Type)
Target:
renaming: [vec] (*tInd _ _ *)
info: [ ]
info_source: [ ("type", information_nat 0)] (*i.e. "vec" *)
```

```
Source:
    Inductive vec (A:Type) •
Target:
    forall (_:Type), •
renaming: [ tRel 0; vec ]
info: [( "parameters", information_list [0])]
info_source: [("type", information_nat 1)]
```

```
Source:
    Inductive vec (A:Type): nat → Type • :=
Target:
    forall (_:Type), (forall _: forall (_:nat), •),
renaming: [ tRel 1; vec ]
info: [( "parameters", information_list [1])]
info_source: [("type", information_nat 1)]
```

```
Source:
    Inductive vec (A:Type): nat → Type • :=
Target:
    forall (_:Type), (forall _: forall (_:nat), vec &1 • → ...)
renaming: [ tRel 1; vec ]
info: [( "parameters", information_list [1])]
info_source: [("type", information_nat 1)]
```

At the first step, we just start the generation, the **renaming** is empty. At the second step, we read the parameter '`(A:Type)`', produce a Prod term (`forall (_:Type), _`), after that, the **renaming** is updated to [0] . At the third step, we read the indice **nat**, produce a Prod term (`forall (n:nat), ...`) , and update **renaming** to [1] , which means that the &0 of source (i.e. `A`) corresponds exactly to the &1 of target.

2.3.2 update the local information

As we know, **infolocal** is the local information that should be carried during the whole process of generation. It is crucial that the local information is updated correctly at any point of the generation.

The definition of `kptProd` is:

```
Definition kptProd (saveinfo:saveinfo) (na:aname) (e:infolocal)
  (t1:term) (t2:infolocal → term): term :=
  let e' := update_kp na e saveinfo in
  tProd na t1 (t2 e').
```

Where an update function is used.

```
Definition update_kp (na:aname) (e:infolocal) (saveinfo:saveinfo): infolocal.
```

As mentioned before$_1$, `kptProd` is used for creating a prod term when the variable "na" refers to a binder in the source. As we create the prod term, we add a variable, we must update the `infolocal` for the body of the abstraction.

`update_kp` does following things:

- update the `renaming`: add each element of `renaming` by one, and add an value 0 to the head of the list.
- update the `info`, `info_source`: for each `information_list`, add each element by 1; for each `info_nat`, add its value by one.
- According to the `saveinfo`
  - `NoSave`, do nothing.
  - `Savelist str`, add a new item of value 0 to the head of `information_list` named `str` of `info`.
  - `Saveitem str`, add (`str`, `information_nat` 0) to the `info`.

The only difference between `mktProd` and `kptProd` is that it uses `update_mk` instead of `update_kp`.

`update_mk` updates the local information a little differently:

- update the `renaming`: add each element of `renaming` by one.
- update the `info`: for each `information_list`, add each element by 1; for each `info_nat`, add its value by one.
- remain the `info_source` unchanged.
- according to `Saveinfo`, does the same thing as defined in `update_kp`.

The reason of the difference between these two update function is that: `kptProd` is used for variable that could be referenced in the source, but `mktProd` is used for creating a variable(which does not appear in the source), see Remark 1. So when we use `mktProd`, the "reading context", the `info_source` should be unchanged, the number of items in `renaming` should remain the same, but each element of `renaming` will be added by one since we produce a new variable there.

3 Case study

In this section, we will show how this approach works by an example: generating the type of induction principle of an inductive type. In the code below, $\overrightarrow{p : Param}$ is the list of parameters, $\overrightarrow{Ind}$ is the list of type of indices, $\overrightarrow{arg}$ is the list of arguments.

```
Inductive T p→:Param→: Ind→ → Type :=
  | Cstr arg→ : T p→ (?) ... (?)
  ...
```

generation $\Longrightarrow$

```
forall p⃗ : Param⃗,
    forall (P: forall i⃗ : Ind⃗, T p⃗ i⃗ → Prop)
        (...)   (depends on the constructors)
        forall i⃗ : Ind⃗ (x:T p⃗ i⃗), P i⃗ x.
```

For simplicity of explication, we do not consider mutual inductive type.

The main function takes the kernel name and the type definition (its quotation) as arguments and returns a term:

```
Definition GenerateIndp (na : kername) (ty : mutual_inductive_body) : term
```

To begin the generation, build the initial local information first:

```
Definition GenerateIndp (na : kername) (ty : mutual_inductive_body) : term :=
    let initial_info := make_initial_info na ty in
    (*suppose single inductive body.*)
    let the_inductive := {| inductive_mind := na; inductive_ind := 0 |} in
    let params := ty.(ind_params) in
    let body := getfirstbody ty.(ind_bodies) in
    let indices := body.(ind_indices) in
    ...
```

Then it is the time to build the `Prod` term, the following code produces the part `forall` $\overrightarrow{p : Param}$, `todo.` where `todo` will be developed later.

```
Definition GenerateIndp (na : kername) (ty : mutual_inductive_body) : term :=
    ...
    e ← it_kptProd (Some "params") params initial_info;; todo.
```

▶ Remark 2. The notation "← ;;" is defined as

`(e ← c1;; c2)  ⇔ (c1 (fun e ⇒ c2))`

Then, for the part `forall (P: forall` $\overrightarrow{i : Ind}$, `T` $\vec{p}\ \vec{i} \rightarrow$ `Prop`), ...

We introduce the variable "P" first:

```
    ...
    e ← it_kptProd (Some "params") params initial_info;;
    e ← mktProd (Saveitem "P") prop_name e (todo) (*type of P*);;
    (todo) (*body of the Prod*)
```

$\Longrightarrow$

```
    ...
    e ← it_kptProd (Some "params") params initial_info;;
    e ← mktProd (Saveitem "P") prop_name e
        (e ← it_mktProd (Some "indices") indices e;; todo);;
    todo (*body of the Prod*)
```

$\Longrightarrow$

```
    ...
    e ← it_kptProd (Some "params") params initial_info;;
    e ← mktProd (Saveitem "P") prop_name e
        (e ← it_mktProd (Some "indices") indices e;;
         tProd the_name
          (tApp (tInd the_inductive []) (*T*)
```

```
              (rels_of "params" e ++ rels_of "indices" e)) (*p i*)
            (tSort sProp) (*Prop*)
          );;
    todo (*body of the Prod*)
```

We use `tProd` directly in the inner part since the type on the right side of the arrow is just `Prop`, a constant value, so no need to update the local information, we could still use `mktProd` here of course. It is important to understand each time why we use one of (`it`)`mktProd`, (`it`)`kptProd` but not the other, see explication in the remark[1.]

Now let us produce the part

`forall` $\overrightarrow{i : Ind}$, `forall` (x:T $\vec{p}$ $\vec{i}$), P $\vec{i}$ x.

which should be at the end of the induction principle.

```
    ...
    e ← it_kptProd (Some "params") params initial_info;;
    e ← mktProd (Saveitem "P") prop_name e (fun e ⇒ ... ) (*type of P*) ;;
    ...
     e ← it_mktProd (Some "indices") indices e;;
     e ← mktProd (Saveitem "x") the_name e
      (*type of x: T A1 ... Ak i1 ... im*)
      (tApp (tInd the_inductive [])
          (rels_of "params" e ++ rels_of "indices" e));;
     (*P i1 ... im x*)
     tApp (rel_of "P" e) (rels_of "indices" e ++ [rel_of "x" e])
```

Now it remains to generate the part of the type constructors. For clarity, let us take the vector for example, the induction principle of vector[1.2:]

```
forall (A:Type), forall (P: forall (n:nat), vec A n → Prop),
  P 0 (nil A) (*generated from nil*) →
  (forall (a:A) (n:nat) (v:vec A n),  P n v → P (S n) (cons A a n v)) (*from cons*) →
    forall (n:nat) (x:vec A n),  P n x.
```

Notice that each constructor generates a term independently, we just need to link them with arrows. For specific constructor `cons`, the generation can be divided into the generation of each argument(`A`, (`n:nat`), `vec A n`), and the generation of the return type (`vec A (S n)`).

The generation of each argument works as follow:

- Save the argument into the local information. (will be used elsewhere)
- Check the type of the argument, check if its type is exactly the one we are defining (here i.e. `vec`); if not, just do a renaming transformation; otherwise, transform its type, and build a prop type (here i.e. `P n v`) and link them.
- link to the following term

A segment of the code is:

```
(*this function generate the term from the arg, and link to the term t at last*)
let auxarg arg (t:infolocal → term): infolocal → term :=
  let t1 := arg.(decl_type) in
  let na := arg.(decl_name) in
  fun e ⇒
  match t1 with (*take arg (vec A n) for example*)
  | tApp (tRel i) tl (*tApp 'vec' '[A n]'*) ⇒
    match is_recursive_call_gen e i with
    | Some _ ⇒
```

```
      (*save the argument v into information list "args"*)
      e ⟵ mktProd (Savelist "args") na e
        (*type of v: vec A n*)
        (tApp (tInd the_inductive []) (map (type_rename_transformer e) tl));;
      (* P n v → t*)
      kptProd NoSave the_name e
        (tApp
          (rel_of "P" e) (*P*)
          (let tl := n_tl tl (length params) in
            (*do x times List.tl, i.e. remove the params*)
            (map (type_rename_transformer e) tl) (*n*)
             ++ [geti_info "args" e 0] (*v*))
        ) t
    | None ⇒
      kptProd (Savelist "args") na e
        (type_rename_transformer e t1)
        t
    end
  |...
```

See more details in code.

4 TODO

A limitation of the current approach:

If one variable in the source will be "used" in the generation, then all variables that occur before it in the source should also be "used" before in the target.

For example, for this type definition: `Inductive T (A:Type) (x:nat) (y:A) ....`

We can generate the type of its induction principle as

`forall (A:Type) (x:nat) (y:A) ...`

But Someone may want to switch the order:

`forall (x:nat) (A:Type) (y:A) ...`

However, currently our approach does not allow the later one.

References

1 Abhishek Anand et al. Metacoq. https://github.com/MetaCoq/metacoq, 2024.

2 Wikipedia contributors. De bruijn index — Wikipedia, the free encyclopedia, 2024. [Online; accessed 6-June-2024]. URL: https://en.wikipedia.org/w/index.php?title=De_Bruijn_index&oldid=1208839602.