Dive deeper into Coq...

软件科学基础习题课2

黄柘铳 TA

Well-founded recursion

Coq's termination of fixpoints

Coq's keyword `Fixpoint` is used to define recursive functions, and we know that the recursive functions are defined to ensure that the recursion terminates. In fact, we will learn that this is guaranteed by the well–founded recursion principle.

- Well–founded relation in math means that there is no infinite descending chain of elements.
- Eg. `<` is a well–founded relation, but `=` is not. An inductive data and its components are also well–founded.
- In Coq, the parameters of the recursive function are required to be well–founded so that the recursive function invocation terminates.

```
well_founded = fun (A : Type) (R : A -> A -> Prop) => forall a : A, Acc R a
     : forall [A : Type], (A -> A -> Prop) -> Prop

Inductive Acc (A0 : Type) (R : A0 -> A0 -> Prop) (x : A0) : Prop :=
    Acc_intro : (forall y : A0, R y x -> Acc R y) -> Acc R x.
```

When fixpoint definition fails

Normally when we define a `Fixpoint` function, coq will check if its parameters are well–founded, but most of the time, this relation can be infered from the parameters automatically.

The default well–founded relation is a **structural decrease** over one of the parameters. For the mergesort function below, however, the default well–founded relation inference failed.

```
Variable A : Type.
Variable le : A -> A -> bool.
Fixpoint insert (x : A) (ls : list A) : list A := ...
Fixpoint merge (ls1 ls2 : list A) : list A := ...
Fixpoint split (ls : list A) : list A * list A := ...
Fail Fixpoint mergeSort (ls : list A) : list A :=
  match ls with
  | nil => nil
  | h :: nil => h :: nil
  | _ => let lss := split ls in
         merge (mergeSort (fst lss)) (mergeSort (snd lss))
  end.
```

How to "truly" define a fixpoint function?

When we use `Fixpoint`, we actually called the combinator `Fix`, which is defined as:

```
Check Fix.
```

```
Fix
     : ∀ (A : Type) (R : A → A → Prop),
       well_founded R →
       ∀ P : A → Type,
       (∀ x : A, (∀ y : A, R y x → P y) → P x) →
       ∀ x : A, P x
```

If we want to define the mergesort above, we have to manually provide all the parameters to the `Fix` combinator, which is a bit tedious.

Definition and proof for the well-founded relation

```
Definition lengthOrder (ls1 ls2 : list A) :=
length ls1 < length ls2.

Lemma lengthOrder_wf' : forall len, forall ls, length ls <= len -> Acc lengthOrder ls.
Proof.
unfold lengthOrder; induction len.
- intros. destruct ls.
    + constructor. intros. inversion H0.
    + inversion H.
- intros. constructor. intros. apply IHlen. lia.
Qed.

Theorem lengthOrder_wf : well_founded lengthOrder.
Proof.
red. intro. eapply lengthOrder_wf'. eauto.
Qed.
```

Define the mergesort function

```
Lemma split_wf1 : forall ls, 2 <= length ls -> lengthOrder (fst (split ls)) ls.
Admitted.
Lemma split_wf2 : forall ls, 2 <= length ls -> lengthOrder (snd (split ls)) ls.
Admitted.

Definition mergeSort : list A -> list A.
refine (Fix lengthOrder_wf (fun _ => list A)
(fun (ls : list A)(mergeSort : forall (ls':list A), lengthOrder ls' ls -> list A) =>
    if le_lt_dec 2 (length ls)
        then let lss := split ls in
            merge (mergeSort (fst lss) _) (mergeSort (snd lss) _)
        else ls)); subst lss.
- apply split_wf1. assumption.
- apply split_wf2. assumption.
Defined.
```

Conclusion

To manually define a fixpoint function, we need to:

- Define a relation **less-than**
- Prove that the relation **less-than** is well-founded
- Define the function using the `Fix` combinator
- Prove that all the parameters in the recursive function calls are **less-than** the original parameters.

In fact, in reality, people don't prefer the manual definition method. They think up a few tricks to make one of the parameters structurally decrease, so that coq can infer the well-founded relation automatically.

- https://github.com/WouterSchols/Coq_Quiksort
- https://softwarefoundations.cis.upenn.edu/vfa-current/Merge.html

Why a prop can be eliminated only to build another prop

Starting from a question...

forall (n:nat), ev n -> Prop这句话是什么意思呢

你有一个函数，其有一个输入是(ev n的)证明，然后返回值是一个命题，这就违反了Coq“从证明出发，只能构造证明”这个原则

我想表达的是，Coq 的这个原则本身是不是只是一种人为添加的规定，假如 Coq 允许写这种函数是不是也没有问题

嗯这是一个好问题，我暂时不能给你明确的答复，只能说我猜想：
在实际应用之中不会存在这个需求，直接ban掉它使得很多东西都更简单了，所以这么设计
上述猜想是否合理需要你自己衡量！

Go back to the derivation of type theory!

The Russell Paradox tells us that mathematical objects should be constructed carefully.

$$S = \{x \mid x \notin x\} \implies S \notin ? S$$

Scientists constructed the set theory and the type theory to **avoid all kinds of mathematical paradoxes**.

- Set theory:
  - ZF, ZFC
  - NBG
  - ...
- Type theory:
  - Hindley Milner type theory,
  - Martin–Löf type theory,
  - Calculus of constructions, Calculus of Inductive Constructions
  - ...

Calculus of Inductive Constructions(CIC)

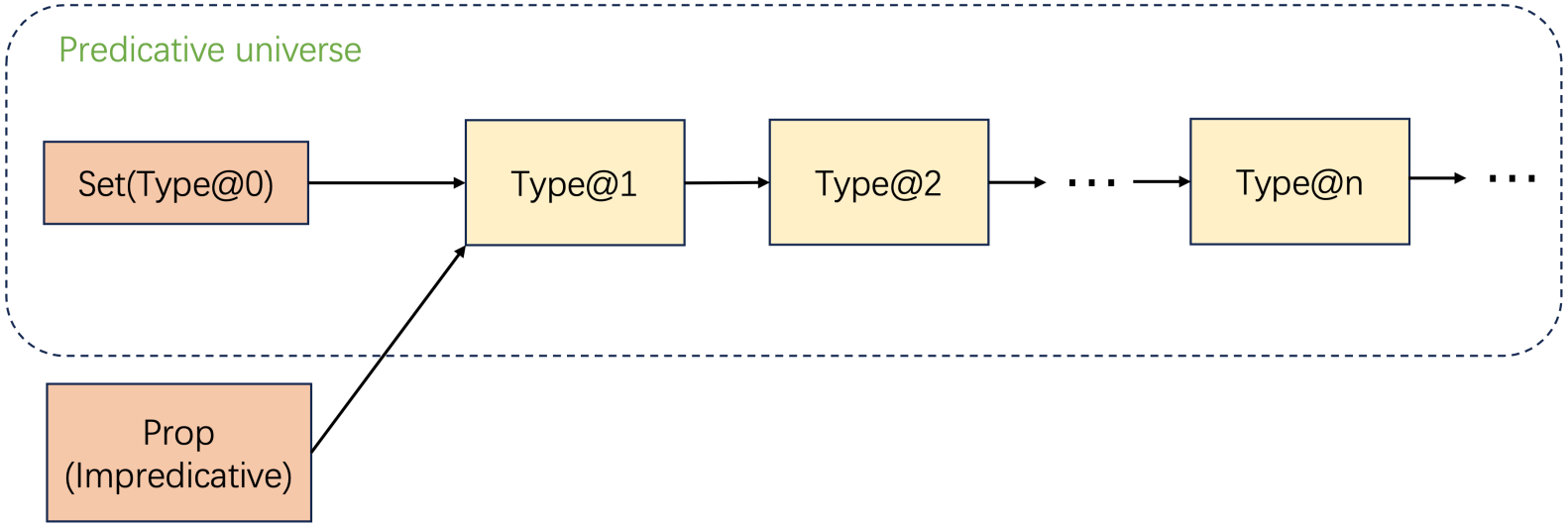
Type theory is a formal system in which we reason about if term has a type: $t : T$

- In coq, we struggle to present a `proof` has a type `p`/proves some prop `p`
- In STLC which we will learn in the following leasons, we will design typing rules for various terms and types.(In fact, stlc is very close to the well-known Hindley-Milner type system, plus a let-polymorphism)
- CIC is a well-known type theory, which has several key differences from naive type theory:
  - It has dependent types, which are types that depend on values
  - It has inductive types, which are types that are defined by their constructors
  - It has a universe hierarchy, which is a collection of types
  - **It has an impredicative universe** `Prop`

Coq universe hierarchy

In modern type theories, we view everything as a term(has a type), including types themselves. Type hierarchy is proposed to avoid paradoxes in self-references like $Type : Type$.

- `Set`: the type of small types
- `Prop`: the type of propositions
- `Type`: the type of types
- `Type@i`: the type of types at level `i`, $Type@i : Type@(i+1)$



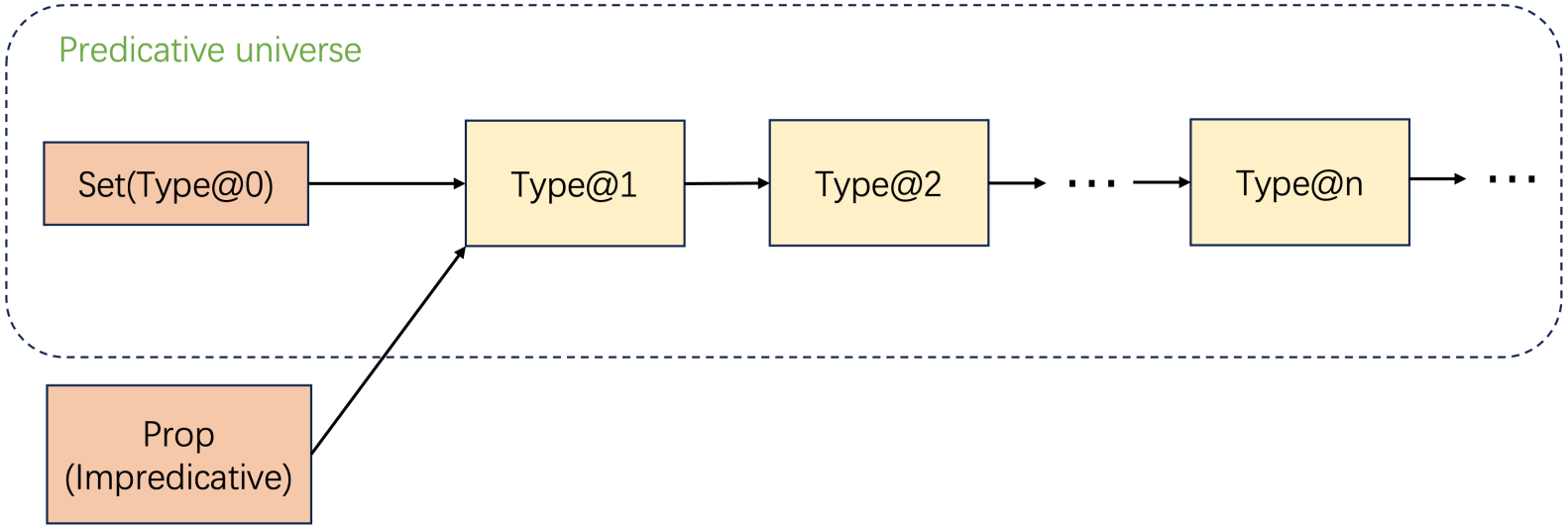
Predicative vs Impredicative

- Predicative: a type can only be defined with quantilization over smaller types.

$$Type@i \to Type@i : Type@(i+1)$$

- Impredicative: a type can be defined with quantilization over all types including itself.

$$Type@i \to prop : Prop$$



Inconsistency triangle

There is an inconsistency triangle in type theory:

> impredicative + large elimination + excluded middle → inconsistency

- We need to reserve the space for `excluded middle` for mathematical reasoning.
- `Set` chooses to preserve large elimination and live in the predicative world.
- `Prop` chooses impredicativity and give up large elimination.

Prop: small elimination

- Large elimination means we can use datas to construct results in a larger universe, like $Array : nat \to Set$
- Small elimination means we can only use datas to construct results in the same universe, like $p : n = 2 \to even(n)$
- `Prop` is a small universe, which means we can only use a prop to construct results in `Prop`, like $f : even(n) \to nat$ is forbidden.
- Small elimination prevents us from leaking abstract/logical information to the concrete/computational world, thus avoiding paradoxes.
- **Conclusion**: a prop can only be eliminated to build another prop → start from a proof, we can only build another proof.

References

- Book: Certified Programming with Dependent Types
- https://rocq-prover.org/doc/v8.20/refman/language/core/sorts.html
- https://github.com/rocq-prover/rocq/wiki/The-Logic-of-Coq
- https://github.com/FStarLang/FStar/issues/360
- https://proofassistants.stackexchange.com/questions/1551/why-not-have-prop-set-in-coq
- https://cstheory.stackexchange.com/questions/21836/why-does-coq-have-prop/21878#21878