Coq projects for type theory 2020

Herman Geuvers, James McKinna, Freek Wiedijk,
Robbert Krebbers, Dan Frumin

February 3, 2020

Here are five projects for the type theory course to choose from. Each student has to choose one of these projects. More than one student can choose the same project, but they should not collaborate on it.

At the end of the course each student needs to submit both a Coq formalization, plus a short report (up to ten pages) that describes the formalization and discusses the choices made while formalizing.

For each project we first give a high level description of what should be done, followed by details of one specific way to do that. However, students are free not to follow these specifics.

1 Type check the simply typed lambda calculus

Formalize the typing rules of the simply typed lambda calculus, then formalize a type checker *for this system, and finally prove that the type checker will produce a correct type judgment if it succeeds.*

This project is an instance of *reflection.* Although the type theory of Coq contains simply typed lambda calculus as a subsystem, the terms and types that the formalization talks about will not be *those* Coq terms and types, but syntactic objects *modelled* in Coq.

You do not need to prove the *completeness* of your type checker.

Here are specifics of one possible solution, which takes 97 lines of Coq:

- Define inductive types `type` and `term` for the types and terms of the simply typed lambda calculus. For example the definition of `type` might look like

  ```
  Inductive type : Set :=
  | var_type : string -> type
  | fun_type : type -> type -> type
  ```

  With this definition the type $(A \to B) \to C$ would be represented by

  ```
  fun_type (fun_type (var_type "A") (var_type "B"))
           (var_type "C")
  ```

To get string notation in Coq, put

```
Require Import String.
Open Local Scope string_scope.
```

at the start of your file.

- Define an inductive predicate has_type, such that the Coq formula

  ```
  has_type Gamma M A
  ```

  corresponds to the derivability of the judgment

  $$\Gamma \vdash M : A$$

- Write a recursive function type_check that (in a given context) returns the type of an element of term. A possible Coq type for this function might be

  ```
  type_check
       : list (string * type) -> term -> option type
  ```

  If the input term (the second argument) is not type correct, the function will have to return None. For this reason the output type is not type but option type.

- You will need to look up variables in the context. For this define an inductive predicate assoc (to be used in the variable case of has_type) and a recursive function lookup (to be used in the variable case of type_check):

  ```
  assoc
       : forall A B : Set, list (A * B) -> A -> B -> Prop
  ```

  ```
  lookup
       : forall A B : Set,
         (forall x y : A, {x = y} + {x <> y}) ->
         list (A * B) -> A -> option B
  ```

  The third argument of lookup is a decision procedure for equality on A. If the keys are strings this argument should be string_dec.

  The functions assoc and lookup correspond to each other in exactly the same way that has_type and type_check do.

- The type checker needs to be able to decide equality of types. (If you apply a function to an argument, the type of the argument needs to match the type of the domain of the function.) For this prove the lemma

```
type_dec
     : forall A B : type, {A = B} + {A <> B}
```

A convenient tactic for proving this is `decide equality.`

- Next we will need to prove our type checker correct. A nice way to do this is by changing the types of `lookup` and `type_check` to have them also return 'proof objects' for the properties of the objects they return:

```
lookup
     : forall A B : Set,
       (forall x y : A, {x = y} + {x <> y}) ->
       forall (l : list (A * B)) (a : A),
       option {b : B | assoc l a b}
type_check
     : forall (Gamma : list (string * type)) (M : term),
       option {A : type | has_type Gamma M A}
```

To find out about the meaning of the set notation `{ ... | ... }` do `Check exist` or `Print sig`.

The function `exist` has implicit arguments. If you want to give those arguments explicitly because Coq cannot figure them out, write `@exist`: then all four arguments can and should be given.

- Often Coq will complain if you just use a simple `match ... with`. In that case using `match ... in ... return ... with` can improve things. For example, the `match` the we used in our definition of `lookup` looks like

```
match l return option {b : B | assoc l a b} with ...
```

- When writing dependently-typed function (which also return proofs), it might be helpful to use the `Program Definition` or `Program Fixpoint` vernacular instead of `Definition` and `Fixpoint`. Using those you will be able to leave some "holes" in your programs using the underscore character `_`, and fill those holes in later using Coq's tactic language. Please consult the Coq reference manual [1] for details.

- An alternative solution would be to use a more "decoupled approach": instead of writing dependently-typed functions that return their proofs, you can write simply-typed functions and prove their properties separately.

2 The pigeon hole principle

Use Coq to formally prove the pigeon hole principle *which says that if you put n pigeons in m holes, with $m < n$, then at least one hole with have more than one pigeon in it.*

Here are specifics of one possible solution, which takes 61 lines of Coq (but with many tactics per line!):

- A possible way to write the statement is:

  ```
  Lemma pigeon_hole :
    forall m n, m < n ->
      forall f, (forall i, i < n -> f i < m) ->
        exists i, i < n /\
          exists j, j < n /\ i <> j /\ f i = f j.
  ```

  Here `f` is the function that maps the number of a pigeon in $\{0 \ldots n-1\}$ to the number of its hole in $\{0 \ldots m-1\}$. The fact that `f` also will map numbers $\geq n$ to something will not hurt.

- A useful tactic to automatically prove equalities and inequalities between natural numbers is

  ```
  omega.
  ```

  To make it available put

  ```
  Require Import Omega.
  ```

  at the start of your file.

- If for natural numbers x and y in a term you want to make a distinction between whether $x \leq y$ or $y < x$, you can write:

  ```
  if le_lt_dec x y then ... else ...
  ```

  Then to do a case split between those two cases in the proof, one can use:

  ```
  elim (le_lt_dec x y).
  ```

3 Proving an expression compiler correct

Formalize both an interpreter and a compiler for a simple language of arithmetical expressions, and show that both give the same results. Compile the expressions to code for a simple stack machine. Use dependent types to make Coq aware of the fact that the compiled code will never lead to a run time error.

Here are specifics of one possible solution, which takes 78 lines of Coq:

- Consider the following expression language:

  $$\begin{array}{rcl} \langle\text{exp}\rangle & ::= & \langle\text{literal}\rangle \mid \langle\text{exp}\rangle + \langle\text{exp}\rangle \mid \ldots \\ \langle\text{literal}\rangle & ::= & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$

  Give an `Inductive` definition of a datatype `Exp` of (the abstract syntax for) ⟨exp⟩s.

- Define a function

  ```
  eval: Exp -> nat
  ```

  giving a semantics for ⟨exp⟩s.

- Give an `Inductive` definition of a datatype `RPN` of Reverse Polish Notation for ⟨exp⟩s.

- Write a compiler

  ```
  rpn : Exp -> RPN
  ```

- Write an evaluator `rpn_eval` for `RPN`, returning an `option nat`.

- Prove that

  ```
  forall e:Exp, Some (eval e) = rpn_eval (rpn e)
  ```

- Generalize the above to `Exp`ressions containing variables, and evaluation with respect to an environment of bindings of variables to `nat`s.

- Discuss how you might avoid explicit consideration of `None` terms in the definition of `rpn_eval`, and explain how you need to modify your formalization in Coq.

4 The unary versus the binary natural numbers

Define both the unary and the binary natural numbers, define addition on both types, define mappings in both directions, show that those mappings are inverse to each other, and finally show that the two addition functions correspond to each other under these mappings.

This exercise is more difficult than it looks, but it is a nice challenge. There are various approaches possible:

- Define the binary numbers with the possibility of leading zeroes. This amounts to 'lists of bits'.

  In the case the mappings between the two types are not simply inverse to each other, as different binary representations might represent the same number. One can define a predicate of two representations being ‘equal’ (= representing the same number), a predicate of a representation being in normal form (= having no leading zeroes), and a function to normalize a representation (= remove the leading zeroes).

Our solution using this approach (with many tactics per line!) takes 347 lines and has 34 lemmas.

- Alternatively one can use a binary representation that is unique. There are at least two possibilities for this:

  – Define a type of *positive* binary natural numbers first, and then use that to define a type of *all* binary natural number. This is similar to the way the type of binary integers `Z` is defined in the Coq standard library.

    Our solution using this approach takes 155 lines and has 11 lemmas.

  – Define *mutual* types of positive and non-negative binary natural numbers:

    ```
    Inductive bnat : Set :=
    | b0 : bnat
    | i : pnat -> bnat

    with pnat : Set :=
    | bit0 : pnat -> pnat
    | bit1 : bnat -> pnat.
    ```

    Here the functions `bit0` and `bit1` add a 0 or 1 at the right end of the number, so they amount to doubling the number respectively doubling it and adding one. Note that the type `bnat` has unique representations for all natural numbers.

    Our solution using this approach takes 143 lines and has 10 lemmas.

  Both of these approaches are quite hairy with most definitions and lemmas appearing twice for the two different types for binary numbers.

5 Integers à la Margaris

Margaris [2] gives a “direct” formalization of the integers, so not as pairs of naturals, or as two copies of the naturals, but by directly defining a language with 0, s (successor) and p (predecessor) with suitable axioms, including an induction scheme. The assignment is to formalize this in Coq, and to prove certain properties about it. The formalization will be slightly different from [2].

Basic definitions

Create a section with the following parameters and hypotheses

```
Parameter ZZ : Set.
Parameter oZ : ZZ.
Parameter pZ sZ : ZZ -> ZZ.

Hypothesis Zps : forall x : ZZ, pZ (sZ x) = x.
Hypothesis Zsp : forall x : ZZ, sZ (pZ x) = x.

Hypothesis ZZ_ind_margaris : forall Q : ZZ-> Prop,
  Q oZ -> (forall y, Q y <-> Q (sZ y)) -> forall x, Q x.
```

So we have a zero and a successor and predecessor that are each others inverses. The final hypothesis is the induction principle for the integers, that Margaris assumes.

Define the predicate N ($N(x)$ says that "x is a natural number") on $\mathbb{Z}$ by

$$N(x) := \forall Q, Q(0) \to (\forall y, Q(y) \to Q(s(y))) \to Q(x)$$

and add as an axiom (using Coq's `Hypothesis` command) the assumption

$$\neg N(p(0))$$

Define the predicate M on $\mathbb{Z}$ by

$$M(x) := \forall Q, Q(0) \to (\forall y, Q(y) \to Q(p(y))) \to Q(x)$$

Prove the following lemmas for $\mathbb{Z}$:

- s and p are injective.
- $N(0)$ and $M(0)$, $\forall x, N(x) \to N(s(x))$ and $\forall x, M(x) \to M(p(x))$.
- $\forall x, N(p(x)) \to N(x)$ and $\forall x, M(s(x)) \to M(x)$.
- $\forall x, p(x) \neq x$ and $\forall x, s(x) \neq x$.
- $\forall x, N(x) \to s(x) \neq 0$ and $\forall x, M(x) \to p(x) \neq 0$.
- $\forall x, M(x) \to \neg N(p(x))$ and $\neg M(s(0))$ and $\forall x, N(x) \to \neg M(s(x))$.

Define the following and prove the given properties

- Define "positive" and "negative" as predicates pos and neg on $\mathbb{Z}$.
- Prove that, for $x : \mathbb{Z}$, *either* $\text{pos}(x)$ *or* $x = 0$ *or* $\text{neg}(x)$.

Possible definitions are $\text{pos}(x) := N(p(x))$ and $\text{neg}(x) := M(s(x))$ and then prove $\forall x : \mathbb{Z}, N(x) \to x = 0 \vee N(p(x))$ and $\forall x : \mathbb{Z}, M(x) \to x = 0 \vee M(s(x))$ to prove $\forall x : \mathbb{Z}, \text{pos}(x) \vee x = 0 \vee \text{neg}(x)$ and $\forall x : \mathbb{Z}, \text{pos}(x) \to x \neq 0 \wedge \neg\text{neg}(x)$ etc.

A full formalization of this assignment can be done in 121 lines of Coq.

6 Finite maps with canonical representation

Define a datatype of finite maps that supports extensional equality. By that we mean the following. If $m_1, m_2 : \mathbb{N} \xrightarrow{fin} A$ are finite maps such that $\forall n, m_1(n) = m_2(n)$, then $m_1 = m_2$.

Specifically define a type `map A` of finite maps from the natural numbers to `A`. Define an operation `lookup :  map A -> nat -> Some A` and prove `forall m1 m2, (forall x, lookup m1 x = lookup m2 x) -> m1 = m2`. Define terms `empty A : map A` and `singleton :  nat -> A -> map A` satisfying

```
forall x, lookup (@empty A) x = None
forall x, x <> y -> lookup (singleton a y) x = None
forall x, lookup (singleton a x) x = Some a
```

As a bonus exercise, define an insertion function and prove its specification in terms of `lookup`.

One possible solution (that can be implemented in about 170 lines of Coq code) is the following.

- A possible representation of finite maps of integers are lists over an option type `list (option A)`. The value `m(k)` is then exactly the element at position `k`. If either `k >= length m` or `m !!  k = None`, then the value $m(k)$ is undefined. For example, a map $–1 \mapsto a, 3 \mapsto b''$ can be represented as `[None; Some a; None; Some b]`.

  ```
  Require Import List Nat.
  (** For lists notations like [] etc *)
  Export ListNotations.
  Definition premap A := list (option A).
  ```

- Maps defined in such a way are not uniquely represented. For instance, the map above can also be represented as `[None; Some a; None; Some b; None; None]`. We will use dependent types to exclude such representations with trailing `None`s.

- This is achieved by endowing `premap`s with a well-formedness predicate:

  ```
  Definition is_Some {A} (x : option A) : bool :=
    match x with
    | Some _ => true
    | None => false
    end.

  Fixpoint premap_wf {A} (may_be_nil : bool) (m : premap A) : bool :=
    match m with
    | [] => may_be_nil
  ```

```
  | x :: m => premap_wf (is_Some x) m
  end.

Definition is_True (b : bool) :=
  if b then True else False.

Definition map_wf {A : Type} (m : premap A) : Prop := is_True (premap_wf true m).

Record map (A : Type) := make_map {
  map_car : premap A;
  map_prf : map_wf map_car
}.
(** Make the parameter [A] implicit *)
Arguments make_map {_} _ _.
Arguments map_car {_} _.
Arguments map_prf {_} _.
```

- To formulate extensional equality we need to define the lookup operation on maps. First we define it on `premaps` and then we lift it to the level of well-formed maps:

```
Fixpoint prelookup {A : Type} (m : premap A) (n : nat) : option A :=
  match m with
  | [] => None
  | x::m’ =>
    match n with
    | O => x
    | S n’ => prelookup m’ n’
    end
  end.

Definition lookup {A : Type} (m : map A) (n : nat) : option A :=
  prelookup (map_car m) n.
```

- Similarly using lifting we can define the empty map and a singleton map.

```
Definition preempty {A : Type} : premap A := [].

Fixpoint presingleton {A : Type} (k : nat) (v : A) : premap A :=
  match k with
  | O => [Some v]
  | S k’ => None :: presingleton k’ v
  end.
```

Exercise: lift those definitions to the type of well-formed maps. The following lemmas might help:

```
Lemma preempty_wf (A : Type) : map_wf (@preempty A).
Lemma presingleton_wf {A : Type} k (v : A) b :
  is_True (premap_wf b (presingleton k v)).

Definition empty {A : Type} : map A.
Definition singleton {A : Type} (k : nat) (v : A) : map A.
```

- **Exercise:** prove the following specification for the operations:

```
Lemma preempty_prelookup {A : Type} (i : nat) :
  prelookup (@preempty A) i = None.

Lemma presingleton_prelookup {A : Type} (k : nat) (v : A) :
  prelookup (presingleton k v) k = Some v.

Lemma presingleton_prelookup_ne {A : Type} (k i : nat) (v : A) :
  k <> i ->
  prelookup (presingleton k v) i = None.

Lemma empty_lookup {A : Type} (k : nat) :
  lookup (@empty A) k = None.

Lemma singleton_lookup {A : Type} (k : nat) (v : A) :
  lookup (singleton k v) k = Some v.

Lemma singleton_lookup_ne {A : Type} (k i : nat) (v : A) :
  k <> i -> lookup (singleton k v) i = None.
```

- **Exercise:** prove that the extensional equality on maps implies intensional equality. The following helper lemmas might be of use:

```
Section canonicity.
  Variable A : Type.

  Lemma map_prf_irrel (m : premap A) (p1 p2 : map_wf m) :
    p1 = p2.
  Proof.
    unfold map_wf in *.
    destruct (premap_wf true m); simpl in *.
    - destruct p1, p2; auto.
    - exfalso; auto.
  Qed.

  Lemma map_car_eq (m1 m2 : map A) :
    map_car m1 = map_car m2 -> m1 = m2.
  Proof.
```

```
    destruct m1 as [m1 Hm1]. destruct m2 as [m2 Hm2]. simpl.
    intros Hm. subst.
    rewrite (map_prf_irrel _ Hm1 Hm2). reflexivity.
  Qed.

  Lemma map_wf_nil (b : bool) (m : premap A) :
    is_True (premap_wf b m) ->
    (forall i, prelookup m i = None) ->
    m = [].
  Proof. ...

  Lemma extensionality (m1 m2 : map A) :
    (forall k, lookup m1 k = lookup m2 k) -> m1 = m2.
  Proof. ...

End canonicity.
```

Hint: for `extensionality` you might need to do induction on `m1` and case analysis on `m2`. Note that this part of the exercise is independent from the part on singleton maps.

- **Exercise (bonus):** define the insert function on maps and prove its specification.

```
Definition insert {A : Type} (m : map A) (k : nat) (v : A) : map A.
Lemma insert_lookup {A : Type} (k : nat) (v : A) m :
  lookup (insert m k v) k = Some v.
Lemma insert_lookup_ne {A : Type} (k i : nat) (v : A) m :
  k <> i ->
  lookup (insert m k v) i = lookup m i.
```

7 Well-founded relations, constructively

In classical mathematics there are many equivalent ways of stating that a relation is well-founded. Typically you would either say that either there are no infinitely descending chains, or that every non-empty subset has a minimal element. In constructive mathematics those definitions are not very useful. The former condition is too weak and the later is too strong.

In constructive mathematics we use a different definition [3]:

Definition. *Let X be a set and $R \subset X \times X$ be a relation. A subset $S \subseteq X$ is R-inductive if*

$$\forall x.(\forall y.R(y,x) \implies y \in S) \implies x \in S.$$

In other words, if we want to show that some $x \in S$, then it suffices to show that $y \in S$ for all y that are "smaller" than x.

Definition. *A relation R is* well-founded *if the only R-inductive subset of X is X itself. Equivalently, if the smallest R-inductive subset of X is X itself.*

In Coq, a relation on a type `X` is represented by a binary predicate `R : X -> X -> Prop`, and a "subset" is given by an unary predicate `S : X -> Prop`. Using Coq's inductive propositions mechanism we can define the smallest `R`-inductive subset of `X` as follows:

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

We then define `R` to be well-founded if every element of `X` is in the smallest R-inductive subset:

```
Definition well_founded {X : Type} (R : X -> X -> Prop) : Prop :=
   forall (x : X), Acc R x.
```

The point of well-foundedness is that if we have a type `X` with a well-founded relation `R`, then we can prove properties `forall (x:X), P x` for `P : X -> Prop` by well-founded induction.

Well-founded induction in this case is just the application of the induction principle for `Acc` (consider `Acc_ind`). Indeed, if we want to prove `forall x, P x` and we know that `R` is well-founded, it suffices to prove `forall x, Acc R x -> P x`. For this we apply the induction principle for `Acc`, and it remains to prove:

```
    forall x, (forall y, R y x -> Acc R y /\ P y) -> P x.
```

But we know that the first conjunct in that formula always holds (by definition of well-foundedness). Then it only suffices to prove `forall x, (forall y, R y x -> P y) -> P x`.

- **Exercise:** Prove that the standard ordering on natural numbers is well-founded.

  ```
  Require Import Nat Omega.
  Theorem nat_wf : well_founded lt.
  ```

- The notion of classical well-foundedness in terms of non-empty subsets is way to strong of a notion for constructive mathematics.

  Exercise: there is an obvious "less than" relation on the type `bool` of Booleans. Show that this relation is constructively well-founded. Show that if this relation is classically well-founded (every non-empty subset has a minimal element), then the law of excluded middle holds.

  ```
  Definition bool_lt (b1 b2 : bool) : Prop.
  Theorem bool_lt_wf : well_founded bool_lt.
  ```

```
Definition classical_well_founded {X : Type} (R : X -> X -> Prop) :=
  forall (P : X -> Prop), (exists x, P x) ->
       exists t, P t /\ (forall y, P y -> ~ (R y t)).

Theorem classical_lem :
  classical_well_founded bool_lt -> (forall Q : Prop, Q \/ ~ Q).
```

- **Exercise.** There is another classical definition of well-foundedness stating that a relation is well-founded if there are no infinite decreasing chains. Prove that this definition is weaker than constructive well-foundedness.

```
Theorem wf_no_chains {X : Type} (R: X -> X -> Prop) :
  well_founded R ->
  ~ (exists s : nat -> X, forall (i : nat), R (s (S i)) (s i)).
```

- **Exercise:** prove that well-founded relations are closed under transitive closure.

```
Inductive TC {X : Type} (R : X -> X -> Prop) : X -> X -> Prop :=
| tc_one x y : R x y -> TC R x y
| tc_step x y z : R x y -> TC R y z -> TC R x z.

Theorem wf_tc {X : Type} (R : X -> X -> Prop) :
  well_founded R -> well_founded (TC R).
```

- One possible solution takes about 80 lines of Coq code.

- **Note:** most of those definitions and theorems are already present in Coq's standard library. Please do not copy-paste the proofs from the standard library in your solution.

References

[1] Matthieu Sozeau, *Program.* In the *Coq Reference Manual for Coq 8.9.0.* URL: `https://coq.inria.fr/refman/addendum/program.html`. Last retrieved: 10.06.2019.

[2] Angelo Margaris, *Successor Axioms for the Integers.* The American Mathematical Monthly, Vol. 68, No. 5 (May, 1961), pp. 441-444 Published by Mathematical Association of America. URL: `http://www.jstor.org/stable/2311096`.

[3] nLab, *Well-founded relation.* URL: `https://ncatlab.org/nlab/show/well-founded+relation`. Last retrieved: 28.01.2018.