Coq projects for CS4135: Software Verification

Robbert Krebbers   Arjen Rouvoet

June 15, 2018

1 Rules and Guidelines

Dates and deliverables. In order to complete the Coq project, you have to make the following deliverables:

1. A set of Coq `.v` files containing the solution to your project.
2. A corresponding written report.

Both deliverables are due

Sunday July 1 at 23:59.

This deadline is strict, i.e. failure to submit both deliverables results in failure of the course. Apart from that, you are advised to acquire feedback during the labs.

Project. This document contains several project suggestions, out of which you should pick one. Each project suggestion consists of a **basis part** and several **extensions**, which are marked with their level of difficulty: easy, moderate, difficult, or very difficult. You have to pick **at least one** extension.

Some remarks:

1. You are advised to first complete the whole exercise before starting to work on the extension.
2. If you want to pursue a difficult or very difficult extension, you are advised to discuss it with the teachers.
3. The level of difficulty of the chosen extension impacts your project grade, as described below.

Report. The written report has to be 4 to 10 pages, should be in academic style (preferably typeset using LaTeX), written in English, and should include:

1. Name and student number at the first page.
2. A small introduction with at least a description of the project you have chosen. You should not just copy the description from this document.
3. A description of the problems you have encountered during the project and how you have solved these. If you had multiple solutions in mind, describe these, and explain why you have picked the one that you have used.
4. A conclusion with at least your experience using Coq: what you did and did not like, what features you think are missing in Coq, etc.

You should not include unnecessarily long excerpts of Coq code in the report, but use small fragments to illustrate a point instead. Informal proofs should be formulated in the way taught during the course. Do *not* paraphrase Coq proofs in natural language by writing: "and now we apply tactic `X` so our goals becomes `Y`".

Grading. The grade of the Coq project is based on the following items:

1. *Correctness:* whether the Coq definitions and lemmas correctly model the problem at hand.
2. *Completeness:* whether all lemmas are proven (e.g. no lemmas omitted and proofs finished with `admit` or `Admitted`).
3. *Style:* whether the Coq code follows sensible style guidelines (e.g. consistent indentation, proper use of parentheses, sensible variable names, proper use of implicit arguments, documentation where needed, etc.).
4. *Effectiveness:* whether the definitions and proofs are carried out effectively (e.g. no useless proof steps and the `induction` tactic not used blindly).
5. *Report:* style, presentation, language and correctness of the report.

The level of difficulty of the extension impacts your grade as follows:

1. *Easy:* your maximum project grade will be an **8**.
2. *Moderate:* your maximum project grade will be an **8.5**.
3. *Difficult:* your maximum project grade will be a **9**.
4. *One very difficult or two difficult extensions:* your maximum project grade will be a **10**.

Notice that picking a (very) difficult extension does not necessarily result in a high grade. Your project should also be graded high w.r.t. the criteria above.

Collaboration. The project is **individual work**. You are allowed to collaborate with at most one fellow student, provided:

1. You acknowledge your collaborator in both your report and Coq files.
2. You carry out a **different extension** than your collaborator.
3. You write your report by yourself.

Library file. As part of your Coq project you are allowed to make use of the file `coq-lab/Maps.v` from Software Foundations, and all transitive dependencies on the Coq standard library.

2 The simply typed λ-calculus

The goal of this project is to formalize the simply typed λ-calculus in Coq and to implement a correct type checker for it.

The simply typed λ-calculus is a subsystem of Coq that only has function types. The types and terms are given by the following grammar:

$$A, B ::= X \mid A \to B$$
$$M, N ::= x \mid M\,N \mid \lambda x : A \,.\, M$$

Here, X ranges over a set of type variables and x over a set of term variables. Examples of types are $X \to Y \to X$ and $(X \to X) \to (X \to X)$, and examples of terms are $\lambda x : X \,.\, \lambda y : Y \,.\, x$ and $\lambda f : X \to X \,.\, \lambda x : X \,.\, f\,(f\,x)$.

In order to define the set of well-typed terms, we define a typing judgment $\Gamma \vdash M : A$ that states that *a term* M *has type* A *in context* Γ. The context Γ associates types to term variables. The typing judgment $\Gamma \vdash M : A$ is defined by the following inference rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A \,.\, M : A \to B}$$

For example, the term $\lambda x : X \,.\, \lambda y : Y \,.\, x$ has type $X \to Y \to X$, which is shown by the following derivation:

$$\cfrac{\cfrac{x : X, y : Y \vdash x : X}{x : X \vdash \lambda y : Y \,.\, x : Y \to X}}{\vdash \lambda x : X \,.\, \lambda y : Y \,.\, x : X \to Y \to X}$$

We formalize the types and terms of the simply typed λ-calculus using the following inductive definitions in Coq:

```
Inductive type :=
  | tvar : string -> type
  | tarr : type -> type -> type.

Inductive term :=
  | var : string -> term
  | app : term -> term -> term
  | lam : string -> type -> term -> term.
```

Exercise 2.1 *Give Coq definitions of type* `type` *corresponding to:*

$$X \to Y \to X \quad \text{and} \quad (X \to X) \to (X \to X).$$

Exercise 2.2 *Give Coq definitions of type* `term` *corresponding to:*

$$\lambda x : X \,.\, \lambda y : Y \,.\, x \quad \text{and} \quad \lambda f : X \to X \,.\, \lambda x : X \,.\, f\,(f\,x).$$

We will represent contexts Γ as partial functions:

```
Definition ctx := partial_map type.
```

Exercise 2.3 *Represent the typing judgment* $\Gamma \vdash M : A$ *as an inductively defined proposition. Do so by completing the following definition:*

```
Inductive typed : ctx -> term -> type -> Prop :=
  | var_typed : forall E x A,
     E x = Some A ->
```

```
    typed E (var x) A
  (* fill out *).
```

Exercise 2.4 *Prove* $\vdash \lambda x : X . \lambda y : Y . x \ : \ X \to Y \to X$*. You have to create a lemma of the following shape:*

```
Lemma test : typed (*your context*) (*term*) (*type*).
```

Exercise 2.5 *Implement a type checker by filling out the following definition:*

```
Fixpoint typecheck (E : ctx) (t : term) : option type :=
  (* fill out *).
```

Exercise 2.6 *Explain the difference between the typing judgment* `typed` *and the type checker* `typecheck`*.*

Exercise 2.7 *Write 2 positive tests and 2 negative tests of the type checker and prove these using the* `reflexivity` *tactic.*

Exercise 2.8 *Prove completeness and soundness of your type checker with respect to the typing rules:*

```
Lemma typecheck_complete : forall E t A,
  typed E t A ->
  typecheck E t = Some A.
Lemma typecheck_sound : forall E t A,
  typecheck E t = Some A ->
  typed E t A.
```

To prove completeness, you may want to use `induction` *on the hypothesis* `typed E t A`*. To prove soundness, you may want to vary the induction hypothesis.*

Exercise 2.9 *Extend the language with a feature of your choice. You have to extend the types, terms, typing rules, type checker and proofs. For example:*

easy	*Products/conjunctions.*
moderate	*Sums/disjunctions.*
moderate	*Natural numbers and recursion.*
difficult	*Polymorphism (à la System F/λ2).* *For this extension, it is recommend to represent type variables using De Bruijn indices instead of strings, because you otherwise have to deal with capture avoiding substitution.*
very difficult	*β-reduction and a proof of type soundness by progress and type preservation.* *For this extension, it is recommend to represent variables using De Bruijn indices instead of strings.*
very difficult	*Curry-style λ-bindings (i.e. without type annotations).* *For this exercise, you will have to implement the type inference algorithm and unification algorithm as we discussed during the lectures.*

Alternatively you may choose to:

difficult *Implement a correct-by-construction type-checker for STLC in Agda:*

```
typecheck :
  ∀ (t : Term) → Dec (∃ λ (a : Type) → Typed [] a t)
```

Where `Dec` *is the Agda standard library type of proof-carrying decisions. For this exercise you may make use of the entire Agda standard library and it is recommended to represent variables using De Bruijn indices.*

3 SAT solver

The goal of this project is to implement a Boolean Satisfiability (SAT) solver in Coq and to prove soundness of your implementation.

Boolean formulas are given by the following grammar:

$$p, q ::= x \mid \mathsf{true} \mid \mathsf{false} \mid p \wedge q \mid p \vee q \mid p \to q \mid \neg p.$$

Here, x ranges over an infinite set of propositional variables. Examples of formulas are $(x \vee \neg y) \wedge (\neg x \vee y)$, $y \to (x \vee y)$ and $x \wedge \neg x \wedge \mathsf{true}$.

A Boolean formula p is said to be *satisfiable* in case its truth table contains at least one row whose outcome is 1. For example:

x	y	$(x \vee \neg y) \wedge (\neg x \vee y)$
0	0	1
0	1	0
1	0	0
1	1	1

satisfiable

x	y	$\neg y \to (x \vee y)$
0	0	0
0	1	1
1	0	1
1	1	1

satisfiable

x	$x \wedge \neg x \wedge \mathsf{true}$
0	0
1	0

unsatisfiable

The first part of this project is to represent Boolean formulas in Coq and to formally define the notion of satisfiability.

Exercise 3.1 *Define an inductive type* `form` *that represents Boolean formulas. Start by completing the following definition:*

```
Inductive form :=
  | var : string -> form
  (* fill out *).
```

Exercise 3.2 *Give Coq definitions of type* `form` *corresponding to:*

$$(x \vee \neg y) \wedge (\neg x \vee y) \qquad \neg y \to (x \vee y) \qquad \text{and} \qquad x \wedge \neg x \wedge \mathsf{true}.$$

In order to define satisfiability we introduce the notion of a *valuation*, which is a function that assigns true or false to each propositional variable.

```
Definition valuation := total_map bool.
```

Exercise 3.3 *Define the interpretation of a formula (using the 'truth table semantics') by filling out the following definition:*

```
Fixpoint interp (V : valuation) (p : form) : bool :=
  (* fill out *).
```

A formula is said to be *satisfiable* if there exists a valuation that makes the formula true. This corresponds to the following definition in Coq:

```
Definition satisfiable (p : form) : Prop :=
  exists V : valuation, interp V p = true.
```

You may want to read the section "Existential Quantification" of the chapter "MoreLogic" of Software Foundations.

Exercise 3.4 *Prove in Coq that $(x \vee \neg y) \wedge (\neg x \vee y)$ and $\neg y \to (x \vee y)$ are satisfiable. You should create two lemmas of the shape below:*

```
Lemma test1 : satisfiable (*formula 1*).
Lemma test2 : satisfiable (*formula 2*).
```

Exercise 3.5 *Define a function that given a formula computes a valuation in which the formula is true. You should implement this function by enumerating all possible valuations (that is generating its truth table).*

```
Definition find_valuation (p : form) : option valuation :=
  (* fill out *).
```

The function `find_valuation` *should yield* `None` *in case no such valuation exists (i.e. the formula is unsatisfiable).*

We can now define our SAT solver as follows:

```
Definition solver (p : form) : bool :=
  match find_valuation p with
  | Some _ => true
  | None => false
  end.
```

Exercise 3.6 *Explain the difference between* `satisfiable` *and* `solver`*.*

Exercise 3.7 *Write 2 positive and 2 negative tests of the solver and prove these tests using the* `reflexivity` *tactic.*

Exercise 3.8 *Prove that* `solver` *is sound. That means:*

```
Lemma solver_sound : forall p,
  solver p = true -> satisfiable p.
```

Exercise 3.9 *Extend your SAT solver in one of the following ways:*

moderate *Extend your development to* three-valued logic*. Three-valued logic has three truth values:* true, false *and some indeterminate third value. See* ***https://en.wikipedia.org/wiki/Three-valued_logic****.*

moderate *Write an optimizer that simplifies a Boolean formula using the laws below, and prove correctness of your optimizer.*

$x \wedge \mathsf{true} = x$	$\mathsf{true} \wedge x = x$	$x \wedge \mathsf{false} = \mathsf{false}$	$\mathsf{false} \wedge x = \mathsf{false}$
$x \vee \mathsf{true} = \mathsf{true}$	$\mathsf{true} \vee x = \mathsf{true}$	$x \vee \mathsf{false} = x$	$\mathsf{false} \vee x = x$

You should incorporate this optimizer in `solver` *and its correctness proof.*

difficult *Write a converter that transforms Boolean formulas into negation normal form and prove correctness of your converter. A Boolean formula is in negation normal form if the negation operator $\neg$ is only applied to variables. You should incorporate the optimizer into* `solver` *and its correctness proof.*

very difficult *Write a converter that transforms Boolean formulas into conjunctive nor-*

mal form and prove correctness of your converter. See also ***https://en.wikipedia.org/wiki/Conjunctive_normal_form****. You should incorporate the optimizer into* `solver` *and its correctness proof.*

very difficult *Prove completeness of your solver. That means:*

```
Lemma solver_complete : forall p,
  satisfiable p -> solver p = true.
```

Alternatively you may choose to:

moderate *Implement Boolean formulas, the solver and its soundness proof in Agda*

difficult *Implement a solver in Agda that is* sound by construction *of type:*

```
solver : ∀ (f : Formula) → Maybe (Satisfiable f)
```

Where `Maybe` *is the Agda standard library version of* `option`*. For this exercise you may make use of the entire Agda standard library.*

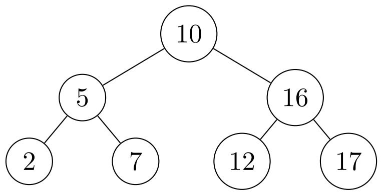
4 Binary search trees

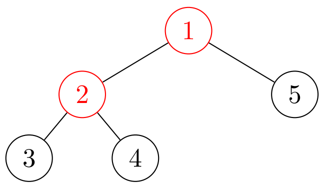
The goal of this project is to implement some operations on binary search trees in Coq and to prove essential properties of the implementation.

A binary search tree is a data structure to efficiently store a finite set of natural numbers. Binary trees are inductively defined in Coq as follows:

```
Inductive tree :=
  | leaf : tree
  | node : tree -> nat -> tree -> tree.
```

A binary tree is said to be a binary search tree if it satisfies the *binary search tree property*, which states that the value in each node is greater (but not equal to) than all values in the left sub-tree, and smaller (but not equal to) than all values in the right sub-tree. For example, the tree on the left satisfies the property, but the one on the right does not (leafs are omitted):





The tree on the right does not satisfy the binary search tree property because there is a node numbered 2 as a left sub-tree of a node numbered 1.

Exercise 4.1 *Give Coq definitions of type* `tree` *corresponding to the above binary trees.*

Exercise 4.2 *Define an inductively defined proposition that describes whether a binary tree satisfies the binary search tree property. Do so by completing the following definition:*

```
Inductive sorted : tree -> Prop :=
  | leaf_sorted : sorted leaf
  (* fill out *).
```

In order to define the above function you may want to define helpers `greater : nat -> tree -> Prop` *and* `smaller : nat -> tree -> Prop` *or* `all : (nat -> Prop) -> tree -> Prop` *as inductively defined propositions.*

Exercise 4.3 *Prove that the first tree you have defined in Exercise 4.1 satisfies the binary search tree property, and that the second one does not. You may want to make use of composition* `;` *of tactics and the* `repeat` *tactical.*

Exercise 4.4 *Define a function that describes whether an element is contained in a binary search tree or not:*

```
Fixpoint elem_of (x : nat) (t : tree) : bool :=
  (* fill out *).
```

To define an efficient implementation of the `elem_of` *function, you should not traverse the whole tree, but make use of the binary search tree property.*

Exercise 4.5 *Write 2 positive tests and 2 negative tests of the* `elem_of` *function and prove these using the* `reflexivity` *tactic.*

The last part of this project is to define a function that inserts a natural number into a binary search tree. You have to prove that the resulting tree is indeed a binary search tree, and you have to prove that it is correct.

Exercise 4.6 *Define a function that inserts a natural number into a binary search tree:*

```
Fixpoint insert (x : nat) (t : tree) : tree :=
  (* fill out *).
```

If the number `x` *is already in the tree, it should return the original tree.*

Exercise 4.7 *Write 2 tests of the* `insert` *function and prove these using the* `reflexivity` *tactic.*

Exercise 4.8 *Prove that the* `insert` *function preserves the binary search tree property. That means:*

```
Lemma insert_sorted : forall t x,
  sorted t -> sorted (insert x t).
```

Exercise 4.9 *Prove that the* `insert` *function is correct. That means:*

```
Lemma insert_correct : forall t x y,
  sorted t ->
  elem_of y (insert x t) = orb (elem_of y t) (beq_nat x y).
```

Exercise 4.10 *Extend your development on binary search trees with a feature of your choice. For example:*

easy *A converter from lists to binary search trees and vice versa. Prove a useful property about the interaction between these functions.*

moderate *A delete function. Prove that delete preserves the binary search tree property and prove a property about the interaction with* `elem_of`*.*

very difficult *Extend your development to self-balancing search trees, such as red-black trees or AVL trees.*

For this extension, you have to extend the `sorted` *proposition to not only account for the binary search tree property, but also for the AVL or red-black property.*

Alternatively you may choose to:

difficult *Implement binary trees in Agda, and use dependent types to make sure that your trees enjoy the binary search tree property. Subsequently, implement the insert function, and prove its correctness using Agda.*

5 Arithmetic expression decompiler

The goal of this project is to formalize a compiler and decompiler for an arithmetical expression language. You have to prove that the compiler is correct and that the decompiler is an inverse of the compiler.

We consider an expression language of arithmetical expressions that is defined by the following inductive types in Coq:

```
Inductive unop :=
  | OSucc : unop
  | OPred : unop.

Inductive binop :=
  | OPlus : binop
  | OMult : binop.

Inductive exp :=
  | ENat : nat -> exp
  | EUnOp : unop -> exp -> exp
  | EBinOp : binop -> exp -> exp -> exp.
```

The expression `ENat` denotes a constant. The unary operator `OSucc` computes the successor and `OPred` computes the predecessor (which is defined to be 0 in case the argument is 0). The operators `OPlus` and `OMult` perform addition and multiplication as usual.

Exercise 5.1 *Define a function* `eval` *that gives a semantics to expressions. Fill out the following template:*

```
Fixpoint eval (e : exp) : nat :=
  (* fill out *).
```

Exercise 5.2 *Write 2 tests of the* `eval` *function and prove these tests using the* `reflexivity` *tactic.*

The first part of this project is to define a compiler from arithmetical expressions to a stack machine. The instructions of the stack machine are in reverse Polish notation and are given by the following Coq definitions:

```
Inductive instruction :=
  | IPush : nat -> instruction
  | IUnOp : unop -> instruction
  | IBinOp : binop -> instruction.
```

Exercise 5.3 *Implement a compiler:*

```
Fixpoint compile (e : exp) : list instruction :=
  (* fill out *).
```

Exercise 5.4 *Implement a virtual machine for stack instructions.*

```
Definition vm : list instruction -> option nat :=
  (* fill out *).
```

The virtual machine should yield `None` *in case of stack underflow. For example,* `vm [IPush 10;`

`IBinOp OPlus] = None`. *In order to define the virtual machine, you may define a helper function.*

Exercise 5.5 *Write 2 positive tests and 2 negative tests of the virtual machine and prove these using the* `reflexivity` *tactic.*

Exercise 5.6 *Prove correctness of your compiler:*

```
Lemma vm_correct : forall e,
   vm (compile e) = Some (eval e).
```

The next part of this project is to define a decompiler. The decompiler translates a list of machine instructions into a corresponding arithmetical expression. You have to show that the decompiler is an inverse of the compiler.

Exercise 5.7 *Define a decompiler:*

```
Definition decompile : list instruction -> option exp :=
  (* fill out *).
```

The decompiler should yield `None` *in case the list of instructions is ill-formed. For example,* `decompile [EBinOp OPlus; IPush 10] = None`.

The implementation of the decompiler is very similar to the implementation of the virtual machine. However, instead of evaluating a list of instructions to a value, you have to evaluate the list to an expression.

Exercise 5.8 *Prove that the decompiler is a left-inverse of the compiler:*

```
Lemma decompile_correct : forall e,
  decompile (compile e) = Some e.
```

Exercise 5.9 *Extend your project in one of the following ways:*

easy *Extend the expression language with variables, and the stack language with a load instruction.*

moderate *Extend the language with an operator that may fail (for example, division may fail in case of division by zero). You have to extend the syntax and semantics of the expression and machine language, as well as your proofs.*

For this extension, you have to describe the semantics of both the expression language and the stack machine as an inductive relation.

difficult *Prove that the decompiler is a right inverse of the compiler:*

```
Lemma compile_decompile : forall ins e,
  decompile ins = Some e -> compile e = ins.
```

difficult *As you may have noticed, the decompiler and the virtual machine, as well as its correctness proofs, are very similar. Try to figure out an abstraction so you can factor out similarities.*

very difficult *Extend the language with a non-deterministic random number generator. You have to extend the syntax and semantics of the expression and machine language, as well as your proofs.*

Alternatively you may choose to:

difficult *Implement your compiler and decompiler in Agda, and use dependent types to make sure that stack underflow in the virtual machine cannot happen. Subsequently, use Agda to prove correctness of the compiler and decompiler.*