Christian-Albrechts-Universität zu Kiel

Bachelor Thesis

Title: TBD

Niels Bunkenburg

SS 2016

TODO: Weitere Formalitäten...

Contents

1 Preliminaries

1.1 Coq

The formalization of Curry programs requires a language that allows us to express the code itself and the propositions we intend to prove. Coq[1] is an interactive proof management system that meets these requirements, thus it will be the main tool used in the following chapters.

TODO: Kommasetzung?

1.1.1 Data types and functions

Coq's predefined definitions, contrary to e.g. Haskell's Prelude, are very limited. However, being a functional language, there is a powerful mechanism for defining new data types. A definition of polymorphic lists could look like this:

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

We defined a type named `list` with two members: the constant `nil`, which represents an empty list, and a binary constructor `cons` that takes an element and a list of the same type as arguments. In fact, `nil` and `cons` have one additional argument, a type `X`. This is required, because we want polymorphic lists – but we do not want to explicitly state the type. Fortunately, Coq allows us to declare type arguments as implicit by enclosing them in curly brackets:

```
Check (cons nat 8 (nil nat)). (*cons nat 8 (nil nat) : list nat*)
Arguments nil {X}.
Arguments cons {X} _ _.
```

Coq's type inference system infers the type of a list automatically now.

```
Check (cons 2 (cons 4 nil)). (* cons 2 (cons 4 nil) : list nat *)
Check (cons 2 (cons nil nil)).
(* Error: The term "cons nil nil" has type "list (list ?X0)"
while it is expected to have type "list nat". *)
```

Based on this we can write a function that determines if a list is empty:

[1] https://coq.inria.fr/

```
Definition isEmpty {X : Type} (l : list X) : bool :=
match l with
| nil      => true
| cons _ _ => false
end.
```

Function definitions begin with the keyword `Definition`. `isEmpty` takes an (implicit) type and a list and returns a boolean value. To distinguish empty from non-empty lists, pattern matching can be used on n arguments by writing `match` $x_0, ..., x_{n-1}$ `with` $| \ p_0 \rightarrow e_0 \ | \ ... \ | \ p_{m-1} \rightarrow e_{m-1}$ for m pattern p, consisting of a sub-pattern for every x_i and expressions e_i.

The definition of recursive functions requires that the function is called with a smaller structure than before in each iteration, which ensures that the function eventually terminates. A recursive function is indicated by using `Fixpoint` instead of `Definition`.

```
Fixpoint app {X : Type} (l1 l2 : list X) : (list X) :=
match l1 with
| nil => l2
| cons h t => cons h (app t l2)
end.
```

In this case l_1 gets shorter with every iteration, thus the function terminates after a finite amount of recursions.

Coq allows us to define notations for functions and constructors by using the keyword `Notation`, followed by the desired syntax and the expression.

```
Notation "x :: y" := (cons x y) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y) (at level 60, right associativity).
```

1.1.2 Propositions and proofs

Every claim that we state or prove has the type `Prop`. Propositions can be any statement, regardless of its truth. A few examples:

```
Check 1 + 1 = 2. (* : Prop *)
Check forall (X : Type) (l : list X), l ++ [] = l. (* : Prop *)
Check forall (n : nat), n > 0 -> n * n > 0. (* : Prop *)
Check (fun n => n <> 2). (* : nat -> Prop*)
```

The first proposition is a simple equation, while the second one contains an universal quantifier. This allows us to state propositions about every type of list, or, as shown in the third example, about every natural number greater than zero. Combined with implications we can premise specific properties that limit the set of elements the proposition applies to. The last example contains an anonymous function, which is used by stating the functions' arguments and an expression.

Now how do we prove these propositions? Proving an equation requires to show that both sides are equal, usually by simplifying one side until it looks exactly like the other. Coq allows us to do this by using tactics, which can perform a multitude of different operations.

```
Example e1 : 1+1=2.
Proof. simpl. reflexivity. Qed.
```

TODO: Highlighting für Proof und Qed

After naming the proposition as an example, theorem or lemma it appears in the interactive subgoal list that Coq provides. The `simpl` tactic performs basic simplification like adding two numbers in this case. The updated subgoal is now `2=2`, which is obviously true. By using the `reflexivity` tactic we tell Coq to check both sides for equality, which succeeds and clears the subgoal list, followed by `Qed` to complete the proof.

```
Example e2 : forall (X : Type) (l : list X), [] ++ l = l.
Proof. intros X l. reflexivity. Qed.
```

Universal quantifiers allow us to introduce variables, the corresponding tactic is called `intros`. The new context contains a type `X` and a list `l`, with the remaining subgoal `[ ] ++ l = l`. Because we defined `app` to return the second argument if the first one is an empty list, `reflexivity` directly proves our goal. `reflexivity` is not useful for obvious equations only, it also simplifies and unfolds definitions until the flat terms match each other if possible.

To prove that the proposition `l ++ [] = l` holds, we need more advanced tactics, because we cannot just apply the definition. `app` works by iterating through the first list, but we need to prove the proposition for every list, regardless of its length. One possibility to solve this problem is by using structural induction.

```
Example e3 : forall (X : Type) (l : list X), l ++ [] = l.
Proof. intros X. induction l as [|l ls IH].
reflexivity.
simpl. rewrite IH. reflexivity.
Qed.
```

The proof begins by introducing type `X`, followed by the `induction` tactic applied to `l`. Coq names newly introduced variables by itself, which can be done manually by adding `as [c1|...|cn]` to the tactic. Each c_i represents a sequence of variable names, which will be used when introducing variables in the corresponding case. Cases are ordered as listed in the `Definition`.

Now we need to prove two cases: the empty list and a cons construct. The first case does not require any new variable names, thus the first section in the squared brackets is empty. It is easily solved by applying `reflexivity`, because of the definition of `app`. The second case requires variables for the list's head and tail, which we call `l` and `ls` respectively. The variable name `IH` identifies the induction hypothesis `ls ++ [ ] = ls`, which Coq generates automatically. The goal changes as following:

```
(l :: ls) ++ [ ] = l :: ls
 l :: ls  ++ [ ] = l :: ls (* simpl *)
 l :: ls         = l :: ls (* rewrite with IH *)
```

The tactic `rewrite` changes the current goal by replacing every occurrence of the left side of the provided equation with the right side. Both sides are equal now, hence `reflexivity` proves the last case.

Example `e4` is different from the other examples, in the sense that one cannot prove a function by itself and that only supplying an argument returns a verifiable inequality.

```
Example e4 : (fun n => n <> 2) 1.
Proof.
  simpl.        (* 1 <> 2 *)
  unfold not.   (* 1 = 2 -> False *)
  intros H.     (* H : 1 = 2, False *)
  inversion H.  (* No more subgoals. *)
Qed.
```

This proof is not as straight forward as the other ones, mainly because of the inequality, which is a notation[2] for `not (x = y)`. Because `not` is the outermost term, we need to eliminate it first by applying `unfold`. This replaces `not` with its definition `fun A : Prop => A -> False`, where `False` is the unprovable proposition. Why does this work? Assuming that a proposition `P` is true, `not P` means that `P` implies `False`, which is false, because something true cannot imply something false. On the other hand, if `P` is false, then `False -> False` is true because anything follows from falsehood, as stated by the principle of explosion.

TODO: verweis?

The current goal `4 = 8 -> False` is further simplified by introducing `4 = 8` as an hypothesis `H`, leaving `False` as the remaining goal. Intuitively we know that `H` is false, but Coq needs a justification for this claim. Conveniently the tactic `inversion` solves this problem easily by applying two core principles of inductively defined data types:

- Injectivity: `C n = C m` implies that n and m are equal for a constructor `C`.
- Disjoint constructors: Values created by different constructors cannot be equal.

By applying `inversion` to the hypothesis `2 = 1` we tell Coq to add all inferable equations as additional hypotheses. In this case we start with `2 = 1` or the Peano number representation `S(S(O)) = S(0)`. Injectivity implies that if the previous equation was true, `S(O) = O` must also be true. This is obviously false, since it would allow two different representations of nil. Hence, the application of `inversion` to `2 = 1` infers the current goal `False`, which concludes the proof.

Besides directly supplying arguments to functions that return propositions, there are other interesting applications for them, that we will discuss in the next section.

[2] It is often useful to be able to look up notations, `Locate "<>"` returns the term associated with `<>`.

1.1.3 Higher-order constructs

Functions can be passed as arguments to other functions or returned as a result, they are first-class citizens in Coq. This allows us create higher-order functions, such as map or fold.

TODO: minted bug?

```
Fixpoint map {X Y : Type} (f : X -> Y) (l : list X) : (list Y) :=
  match l with
  | []      => []
  | h :: t => (f h) :: (map f t)
  end.
```

Function types are represented by combining two or more type variables with an arrow. Coq does not only allow higher-order functions, but also higher-order propositions. A predefined example is `Forall`, which features a `A -> Prop` construct from the last section.

```
Forall : forall A : Type, (A -> Prop) -> list A -> Prop
```

`Forall` takes a *property* of `A`, which returns a `Prop` for any given `A`, plus a list of `A` and returns a proposition. It works by applying the property to every element of the given list and can be proven by showing that all elements satisfy the property.

```
Example e5 : Forall (fun n => n <> 8) [2;4].
Proof.
apply Forall_cons. intros H. inversion H.
(* Forall (fun n : nat => n <> 8) [4] *)
apply Forall_cons. intros H. inversion H.
(* Forall (fun n : nat => n <> 8) [ ] *)
apply Forall_nil.
Qed.
```

`Forall` is an inductively defined proposition, which requires rules to be applied in order to prove a certain goal. This will be further explained in the next section, for now it sufficient to know that `Forall` can be proven by applying the rules `Forall_cons` and `Forall_nil`, depending on the remaining list. Because we begin with a non-empty list, we have to apply `Forall_cons`. The goal changes to `2 <> 8`, the head of the list applied to the property. We have already proven these types of inequality before, `inversion` is actually able to do most of the work we did manually by itself. Next the same procedure needs to be done for the list's tail `[4]`, which works exactly the same. To conclude the proof, we need to show that the property is satisfied by the empty list. `Forall_nil` covers this case, which is trivially fulfilled.

1.1.4 Logic and inductively defined propositions

```
Fixpoint InB (x : nat) (l : list nat) : bool :=
match l with
```

```
| [] => false
| x' :: l' => if (beq_nat x x') then true else InB x l'
end.

Fixpoint In (x : nat) (l : list nat) : Prop :=
match l with
| [] => False
| x' :: l' => x' = x \/ In x l'
end.

Inductive InInd : nat -> list nat -> Prop :=
| Head : forall n l, n = hd n l -> InInd n l
| Tail : forall n l, InInd n (tl l) -> InInd n l.

Example e5 : InB 42 [1;2;42] = true.
Proof. simpl. reflexivity. Qed.
Example e6 : In 42 [1;2;42].
Proof. simpl. right. right. left. reflexivity. Qed.
Example e7 : InInd 42 [1;2;42].
Proof. apply Tail. simpl. apply Tail. simpl. apply Head. simpl. reflexivity. Qed.
```

2 Wat?

3 CuMin

3.1 Typing

4 Curry

4.1 FlatCurry

4.2 Typing

4.2.1 Differences to CuMin

4.2.2 Conversion of FlatCurry to Coq

5 Conclusion