

Christian-Albrechts-Universität zu Kiel

Bachelor Thesis

TODO:
Weitere Formalitäten...

Title: TBD

Niels Bunkenburg

SS 2016

Contents

1	Preliminaries	1
1.1	Coq	1
1.1.1	Data types and functions	1
1.1.2	Propositions and proofs	2

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. Coq¹ 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 don't 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 deduces 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, \dots, x_{n-1} with $p_0 \rightarrow e_0 \mid \dots \mid p_{m-1} \rightarrow e_{m-1}$` 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 *)

```

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, in the third example, about every natural number that is greater than zero. The implication allows us to require specific properties that limit the universal quantification.

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.`

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.

TODO:
Highlighting
für Proof
und Qed

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` isn't 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 can't 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 doesn't 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