Technische Hochschule Rosenheim

Programmcodevalidierung mit Coq

Lukas Kiederle
Fakultät für Informatik

WS 2019/20

Kurzfassung

Aus Zeit- und Kostengründen beim Entwickeln und Testen von komplexen Systemen werden Tools zur Programmcodevalidierung immer relevanter. Diese Tools ermöglichen das Schreiben von Programmen, welche mathematisch und maschinell geprüft sind. Dadurch ist sichergestellt, dass das beschriebene Programm sich auch wie gewünscht, verhält.
Ziel dieser Arbeit ist es, einen sowohl theoretischen als auch technischen Einblick in die Programmcodevalidierung mit dem Proof Assistent Tool Coq darzustellen. Als Einstieg werden die grundlegende Begriffe geklärt und ein kurzer Überblick über Tools in diesem Fachbereich dargestellt. Dabei wird insbesondere auf Coq eingegangen.
Um ein Verständnis zu bekommen, wie ein Proof Assistent Tool die Qualität von Programmcode sicherstellt, müssen zunächst die Grundlagen dieser Sprache anhand von Beispielen erklärt werden. Anschließend wird näher auf das Zusammenspielen zwischen Programmcode und Proof Assistent eingegangen.
Es gibt bereits einige sehr erfolgreiche Forschungsprojekte, die Coq im Einsatz haben. Diese werden abschließend vorgestellt. Schlussendlich wird ein Fazit inklusive Ausblick in Hinsicht auf die Verwendbarkeit von Proof Assistent Tools gezogen.

Schlagworte:

- Proof Assistant
- Coq
- Programcodevalidierung

Leseanleitung

Hinweise auf referenzierte Literatur und die daraus entnommenen Zitate, welche in eckigen Klammern angegebenen sind, werden im Literaturverzeichnis am Ende der Arbeit aufgeführt. Soll ein Begriff oder eine Formulierung besonders hervorgehoben werden, ist diese *kursiv* geschrieben. Abkürzungen werden bei erstmaligem Auftreten einmal in runden Klammern, anschließend an das Wort ausgeschrieben. Um den Lesefluss nicht zu stören, werden alle darauf folgenden Wiederholungen der Abkürzungen nicht immer explizit ausgeschrieben.
Möglicherweise unbekannte Begriffe und Fachbegriffe werden bei ihrer ersten Nennung **fett** gedruckt. Diese sind im Glossar in alphabetischer Reihenfolge aufgelistet und werden näher erklärt. Einzige Ausnahme hierbei sind Überschriften von Tabellen. Um ein zusammenhängendes Lesen der Arbeit zu erleichtern, werden bei Bedarf Erklärungen bereits im Text gegeben. Dabei wird davon ausgegangen, dass der Leser bereits mit grundlegenden Begriffen der Informatik vertraut ist. Ausgehend vom Wissensstand eines entsprechend vorgebildeten Lesers, werden demzufolge nur fachlich speziellere Begriffe erklärt.
Um Unklarheiten zu vermeiden, werden Fachbegriffe in und zur Beschreibung von Bildern und Prozessen in ihrer originalen Sprache Englisch verwendet und nicht immer übersetzt.
An den Stellen, an denen es der Ausführung des Textes dient, sind kurze Codebeispiele im Text eingebunden. Außerdem werden Abbildungen zur Veranschaulichung verwendet, um komplexe Prozesse einfacher und verständlicher zu machen. Größere Abbildungen befinden sich im Anhang und werden im passenden Textabschnitt referenziert. Damit soll der Lesefluss nicht gestört werden.

Inhaltsverzeichnis

1 Motivation

Heutige Software wird üblicherweise so gebaut, dass sie funktioniert. Das bedeutet, dass eine Person, welche mit der Software interagiert, diese möglichst problemfrei nutzen kann. Dabei spielt oftmals vor alle in sicherheitskritischen Bereichen die Qualität eine große Rolle. Um diese zu gewährleisten, werden verschiedenste Methoden wie zum Beispiel Unit-Tests, Integrationstests und manuelles Testen eingesetzt.
Ein Problem bei dieser Art Software zu testen ist allerdings, wenn Fälle außerhalb der bereits bekannten Tests aufkommen. Dies könnte für die jeweilige Software bedeuten, dass es zu keinem Problem bis hin zu großen finanziellem oder auch menschlichem Schaden kommen kann.
Mathematisch und maschinell geprüfte Programme sollen diese Lücke in Zukunft schließen.

- Normale Software wird so gebaut und evaluiert, sodass sie funktioniert
- "If you start with an English-language specification, you're inherently starting with an ambiguous specification," said Jeannette Wing, corporate vice president at Microsoft Research. "Any natural language is inherently ambiguous. In a formal specification you're writing down a precise specification based on mathematics to explain what it is you want the program to do."
- `https://www.quantamagazine.org/formal-verification-creates-hacker-proof-cod`
- Fehler 40
- Was wenn ein Compiler auf unterschiedlichen System aus dem selben Sourcecode unterschiedliche Runnables macht?
- `https://www.mikrocontroller.net/articles/Compilerfehler`
- `https://blog.regehr.org/archives/26`
- Unit tests schreiben, um möglichst viele Fehler zu finden -> Proof Assistent um für alle korrekt zu funktionieren
- Proof Assistent benutzen um formal richtigen Code zu schreiben/generieren
- **Fragestellung: Wie gewährleiste ich sichereren/gut getesteten Code?**

2 Grundlagen

2.1 Was ist ein Proof Assisstant

`https://www.youtube.com/watch?v=95VlaZTaWgc&t=2646s`

2.1.1 Proof Verifier

2.1.2 Theorem Provers

2.2 Übersicht

`https://en.wikipedia.org/wiki/Proof_assistant`

- ACL2
- Isabelle
- Coq

3 Coq

https://www.amazon.de/Certified-Programming-Dependent-Types-Introduction/dp/0262026651/ref=sr_1_fkmr0_1?__mk_de_DE=ÅMÅŽ~O~N&keywords=coq+proof+assistent&qid=1572974699&sr=8-1-fkmr0

- Warum Coq
- funktionaler Programmierstil
- Frenchmade
- Warum ist coq so erfolgreich?

4 Programmatische Coq-Grundlagen

- Dependent type Sprache: So we established that we can prove things are true before we have a concrete value. To do this in an actual programming language, we need a way to encode these statements into the type system itself, which means our type system needs an upgrade. https://medium.com/background-thread/the-future-of-programming-is-dependent-types-programming-word-of-the-day-f We're turning the above run-time assertion into compile-time type.
- https://softwarefoundations.cis.upenn.edu/lf-current/Basics.html#lab18 [dACCMGMGCHVSBY19]
- Coq und Coqide - Zusammenspiel mit Screenshot => wird in Kapitel mit Beispielbeweis genauer drauf eingegangen

4.1 Basisbegriffe

4.1.1 Typdefinition

```
1  Inductive bool : Type :=
2          | true
3          | false.
4
5  Inductive day : Type :=
6          | monday
7          | tuesday
8          | wednesday
9          | thursday
10         | friday
11         | saturday
```

```
12          | sunday.
13
14  Inductive nat : Type :=
15          | O
16          | S (n : nat).
```

Codebeispiel 1: Coq Typedefinition

Die Beispiele aus dem Codeblock 1 stellen drei Typedefinitionen in Coq dar. Ersteres ist ein klassischer Bool. Sowie dieser true oder false annehmen kann, repräsentiert der zweite Type day alle Wochentage.
Die letzte Definition wird verwendet um alle natürlichen Zahlen darzustellen. **S (n : nat)** stellt den Successor z.d. die Nachfolgefunktion dar. Somit kann durch diesen Typ jeder Zahlenwert der natürlichen Zahlen dargestellt werden. Eine 4 würde beispielsweise durch die vierte Nachfolgefunktion von 0 wie folgt dargestellt werden. **(S (S (S (S O)))) => 0 + 1 + 1 + 1 + 1 => 4**.
Des Weiteren ist es auch möglich Komposition durch das Schlüsselwort **Inductive** abzubilden.

4.1.2 Funktionen

In Coq gibt es mehrere Arten von Funktionstypen. Mit dem Keyword **Definition** können einfach Funktionen dargestellt werden. Oftmals wird allerdings Rekursion benötigt. Diese ist nur möglich, wenn die Deklaration mit **Fixpoint** oder ähnlichen Wörtern beschrieben ist. Anstelle von **Theorem** könnten Beispielsweise auch **Example, Lemma, Fact oder Remark** stehen. Diese Schlüsselwörter ermöglichen es in Coq mittels des Allquantors die Korrektheit einer Funktion für alle Elemente einer Menge zu beweisen. In Codeblock 2 ist für die unterschiedlichen Funktionstypen jeweils ein Beispiel dargestellt.

```
1   Definition minustwo (n : nat) : nat :=
2   match n with
3           | O => O
4           | S O => O
5           | S (S n') => n'
6   end.
7
8   Theorem plus_O_n' : forall n : nat,
9   0 + n = n.
10
11  Fixpoint plus (n : nat) (m : nat) : nat :=
12  match n with
13          | O => m
14          | S n' => S (plus n' m)
15  end.
```

Codebeispiel 2: Coq Funktionen

Die erste Funktion **minustwo** zieht von einer eingegebenen natürlichen Zahl zwei ab. Allerdings ergibt **0 - 2, 1 - 2 => 0**. Dies ist durch die ersten zwei Fälle des **match**-Begriffs dargestellt.
Das **Theorem plus_O_n** liest sich wie folgt: "Für alle natürlichen Zahlen n gilt 0 + n = 0".

Im folgenden Kapitel wird gezeigt, wie eine solche Funktion mathematisch bewiesen werden kann.

```
(* Run function plus with 3 and 2. Result => 5 *)
Compute (plus 3 2).

(*  plus (S (S (S O))) (S (S O))
        ==> S (plus (S (S O)) (S (S O)))
by the second clause of the match
        ==> S (S (plus (S O) (S (S O))))
by the second clause of the match
        ==> S (S (S (plus O (S (S O)))))
by the second clause of the match
        ==> S (S (S (S (S O))))
by the first clause of the match
*)
```

Codebeispiel 3: Coq rekursive Funktion

Um ein tieferes Verständnis für die Rekursion in Coq zu bekommen, sind im Codeblock 3 die einzelnen Schritte in einem Kommentar-block (gekennzeichnet durch (* *)) abgebildet. Im 1. Schritt stellt Coq, wie bereits bei den Typdefinitionen der natürlichen Zahlen gezeigt, die Dezimalzahlen zwei und drei mittels der Successor-funktion dar. Anschließend beginnt die Rekursion. Solange $\mathbf{n > 0}$, wird 1 mehr zum Endergebnis gezählt. Wenn $\mathbf{n = 0}$, dann wird, wie in den letzten zwei Zeilen im Codeblock dargestellt, das plus durch **m** ersetzt. Somit ergibt **plus 3 2 => 5**.

4.2 Beweise und Taktiken

Úm zu prüfen, dass die definierten Funktionen mathematisch korrekt sind, stellt der Proof Assistent verschiedene Taktiken zur Verfügung. Diese werden zwischen den **Proof.** und **Qed.** Schlüsselworten angegeben.
Eine grundlegende Beweismethode ist die Induktion, welche nur für die natürlichen Zahlen verwendet werden kann. Dabei wird zuerst geprüft, ob beim Einsetzen in die zu beweisende Funktion der kleinste Wert gültig ist. Anschließend soll die Aussage für $\mathbf{n + 1}$ bewiesen werden. Wenn beides zu einem gültigen Ergebnis führt, ist die Funktion mathematisch valide.

```
Theorem plus_1_l : forall n:nat, 1 + n = S n.
Proof.
        intros n.
        reflexivity.
Qed.

Theorem plus_n_O : forall n:nat, n = n + 0.
Proof.
        intros n.
        induction n as [| n' IHn'].
                - (* n = 0 *) reflexivity.
                - (* n = S n' *) simpl.
                  rewrite <- IHn'.
```

```
14          reflexivity.
15 Qed.
```

Codebeispiel 4: Coq Beispielbeweis

Im Codeblock 4 sind zwei Theoreme bewiesen. Ersteres kann durch zwei Taktiken geprüft werden. **Intros** in Verbindung mit allen verwendeten Variablen des Theorems, setzt diese in den Kontext. Dies ist vergleichbar mit: "Gegeben sei n, eine natürliche Zahl".
Ein anschließendes Anwenden von **reflexivity** sorgt dafür, dass das Programm überprüft, ob die linke und rechte Seite identisch sind. Dabei führt **reflexivity** auch noch ein **simpl** zur Vereinfachung (z.B.: 0 + n => 0) aus. **Reflexivity** muss somit immer am Ende eines Beweises stehen, sodass er abgeschlossen ist.
Die zweite Funktion wird mit Hilfe der Taktik **induction** gelöst. Diese teilt die Aussage in zwei Subgoals (z.d. Teilziele) auf. Anschließend gilt es, jedes einzelne Ziel zu prüfen. Diese werden in verschiedenen Ebenen mithilfe von **-, +, *** gekennzeichnet. Ein **-** wird bei der ersten Subgoal-Ebene verwendet. Für das Adressieren weiterer Subgoals von Subgoals werden + und * genutzt.

4.3 Beispielbeweise

```
1  (* Initiating the theorem to proof. *)
2  Theorem plus_id_exercise : forall n m o : nat,
3          n = m ->
4          m = o ->
5          n + m = m + o.
6
7  (* result:
8  1 subgoal
9  ____________________________________(1/1)
10 forall n m o : nat,
11 n = m -> m = o -> n + m = m + o*)
12
13 Proof.
14 (* move quantifiers into the context: *)
15         intros n m o.
16
17 (* result:
18 1 subgoal
19 n, m, o : nat
20 ____________________________________(1/1)
21 n = m -> m = o -> n + m = m + o*)
22
23 (* move hypothesises into the context: *)
24         intros H.
25         intros H2.
26
27 (* result:
28 1 subgoal
29 n, m, o : nat
```

```
H : n = m
H2 : m = o
______________________________________(1/1)
n + m = m + o*)

(* rewrite the goal using the hypothesises: *)
        rewrite -> H.

(* result:
...
______________________________________(1/1)
m + m = m + o
*)
        rewrite <- H2.

(* result:
...
______________________________________(1/1)
m + m = m + m
*)
        reflexivity.
Qed.
```

Codebeispiel 5: Coq Beispielbeweis

5 Zusammenspiel Coq und Programmcode

- Wir haben eine Liste von Dingen, die die Software tun soll, und verwenden Logik, um zu beweisen, dass die Software diese Dinge tut.
- `https://www.youtube.com/watch?v=Ue8QG8pf0wU`
- Codegenerierung, Proof -> Programm (ich gehe vorallem hierauf ein)
- Programm -> Proof

5.1 Proof -> Programm

5.1.1 theoretisch

- `https://medium.com/@ahelwer/formal-verification-casually-explained-3fb4fef2`
- Erstelle eine Spezifikation, die ein Programm beschreibt
- Schreibe die Spezifikation mathematisch in ein Proof Tool
- Wenn es Fehler enthält, sagt es dir der Verifier
- Code to Ocaml extractor ist nicht formal verifiziert. Wird aber viel verwendet, weil es einfach ist

5.1.2 praktisch

Funktionalitäten schreiben und mathematisch beweisen

- natprod beschreibt ein paar von 2 natürlichen Zahlen
- Check und Compute wird als Konsolen-Ausgabe genutzt
- fst gibt x zurück
- snd gibt y zurück
- Notation ist ein Alias um bestimmte Ausdrücke anders als normal zu schreiben
- swap_pair vertauscht x und y
- surjective_pairing und surjective_pairing_stuck beweist, dass das Erstellen von einem neuem Paar demselben entspricht, wenn man fst und snd von diesem Paar nimmt und ein neues Paar bildet.
- snd_fst_is_swap und fst_swap_is_snd prüft die swap_pair Funktion auf Korrektheit.
- destruct bedeutet: The tactic that tells Coq to consider, separately, the cases where n = O and where n = S n' is called destruct. The destruct generates two subgoals, which we must then prove, separately, in order to get Coq to accept the theorem. The same as we used in induction. The difference of destruct and induction is, that induction is capable of checking n = 0, n+1 and than say, if this is correct -> the whole proof is correct. In this example we use destruct just to split the pair p into n and m again. Otherwise Coq can't simplify the statement.

```
From LF Require Export Induction.

Inductive natprod : Type :=
        | pair (n1 n2 : nat).

Check (pair 3 5).

Definition fst (p : natprod) : nat :=
match p with
        | pair x y => x
end.

Definition snd (p : natprod) : nat :=
match p with
        | pair x y => y
end.

Compute (fst (pair 3 5)).
Compute (snd (pair 5 7)).

Notation "( x , y )" := (pair x y).
```

```
22
23 Compute (fst (3,5)).
24 Definition fst' (p : natprod) : nat :=
25 match p with
26         | (x,y) => x
27 end.
28 Definition snd' (p : natprod) : nat :=
29 match p with
30         | (x,y) => y
31 end.
32 Definition swap_pair (p : natprod) : natprod :=
33 match p with
34         | (x,y) => (y,x)
35 end.
36
37 Theorem surjective_pairing' : forall (n m : nat),
38 (n,m) = (fst (n,m), snd (n,m)).
39 Proof.
40         simpl.
41         reflexivity.
42 Qed.
43
44 Theorem surjective_pairing_stuck : forall (p : natprod),
45 p = (fst p, snd p).
46 Proof.
47         intros p.
48         destruct p as [n m].
49         simpl.
50         reflexivity.
51 Qed.
52
53 Theorem snd_fst_is_swap : forall(p : natprod),
54 (snd p, fst p) = swap_pair p.
55 Proof.
56         intros p.
57         destruct p as [n m].
58         simpl.
59         reflexivity.
60 Qed.
61
62 Theorem fst_swap_is_snd : forall(p : natprod),
63 fst (swap_pair p) = snd p.
64 Proof.
65         intros p.
66         destruct p as [n m].
67         simpl.
68         reflexivity.
69 Qed.
```

Codebeispiel 6: Coq Beispielbeweis

Proof extrahieren coqc -Q . LF PaperPairExtraction.v

```
Require Extraction.
Extraction Language OCaml.
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlString.
Require Import Arith Even Div2 EqNat Euclid.

Extract Inductive nat => int [ "0" "Pervasives.succ" ]
"(fun fO fS n -> if n=0 then fO () else fS (n-1))".

Extraction "paperimpl.ml" fst snd swap_pair.
```

Codebeispiel 7: Coq Code extrahieren

Ocaml Code anpassen (Tests hinzufügen Sachen ausbessern)

```
type natprod =
| Pair of int * int

(** val fst : natprod -> int **)

let fst = function
| Pair (x, _) -> x

(** val snd : natprod -> int **)

let snd = function
| Pair (_, y) -> y

(** val swap_pair : natprod -> natprod **)

let swap_pair = function
| Pair (x, y) -> Pair (y, x)
```

Codebeispiel 8: Ocaml Code anpassen

```
let pair = Pair(3, 4);;
let resultfst = fst pair;;
let resultsnd = snd pair;;

Printf.printf "Result fst: %d \n%!" resultfst;;
Printf.printf "Result snd: %d \n%!" resultsnd;;

let pair2 = swap_pair pair;;
let resultfst2 = fst pair2;;
let resultsnd2 = snd pair2;;
```

```
11
12 Printf.printf "Result fst: %d \n%!" resultfst2;;
13 Printf.printf "Result snd: %d \n%!" resultsnd2;;
```

Codebeispiel 9: Ocaml Code anpassen

Ocaml code compilieren

```
1 ocamlc -w -20 -w -26 -o paperimp paperimpl.mli paperimpl.ml
```

Codebeispiel 10: Ocaml Code compilieren

Ocaml code ausführen

```
1 lukas@luk-ubuntu@~/Documents/coq-test/lf: ./paperimp
2 Result fst: 3
3 Result snd: 4
4 Result fst: 4
5 Result snd: 3
```

Codebeispiel 11: Ocaml code ausführen

5.2 Programm -> Proof

- nutze Verified Software Toolchain (VST) der PrincetonUniversity um C Code mathematisch zu beweisen
- Schreibe ein C Program F.c
- Führe clightgen -normalize F.c aus. Dadurch entsteht eine Datei Coq-File F.v
- Schreibe eine formale Verifikation in einer Datei (z.B.: verif_F.v). In dieser File müssen sowohl F.v als auch das VST Floya Programm-Verifikationssystem VST.floyd.proofauto importieren.[AWAwLB19]

6 Aktuelle Anwendung

6.1 Proofed Stack

- CompCert (C compiler)
- Princeton VST
- Certikos (verified Operating System with hypervisor and multi instances)
- `http://plv.csail.mit.edu/kami/`
- `https://www.zdnet.com/article/certikos-a-hacker-proof-os/`
- `https://github.com/PrincetonUniversity/VST`
- `https://vst.cs.princeton.edu`
- `https://news.yale.edu/2016/11/14/certikos-breakthrough-toward-hacker-resist`

6.2 CompCert for C

`http://compcert.inria.fr`

6.3 JSCert for ECMA 5

`https://github.com/jscert/jscert`

6.4 4-Farben Rätsel ist löstbar!

6.5 CertiCoq

`https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf`

7 Anwendbarkeit in der Praxis

- Objektorientierung eher schwer, da direkte Umwandlung von funktionaler Sprache in eine funktionale Sprache einfacher ist. (Außerdem programmieren die Menschen, die das entwickeln eigentlich nicht wirklich objektorientiert)
- Sicherheitskritische Systeme
- Compilerbau

8 Fazit

8.1 Aussicht

9 Glossar

A Erster Abschnitt des Anhangs

In diesem Anhang wird ...

Literatur

[AWAwLB19] J. D. Andrew W. Appel with Lennart Beringer, Qinxiang Cao. Verifiable C, Applying the Verified Software Toolchain to C programs. S. 8, 2019.

[dACCMGMGCHVSBY19] B. C. P. A. A. de Amorim Chris Casinghino Marco Gaboardi Michael Greenberg Caetaelin Hritcu Vilhelm Sjoeberg Brent Yorgey. Logical Foundations. 1, 2019.