Technische Hochschule Rosenheim

Programmcodeverifikation 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 Programmcodeverifikation 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 Programmcodeverifikation 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 Assistent
- Coq
- Programcodeverifikation

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.

Inhaltsverzeichnis

1 Motivation

Wenn heutzutage die Spezifikation eines Projektes in Englisch formuliert wird, ist diese Spezifikation von Anfang an mehrdeutig, behauptet Jeannette Wing, Coperate Vice President von Microsoft Research. Des Weiteren sagt sie, dass jede natürliche Sprache mehrdeutig ist. Hingegen in formalen Spezifikationen, wird basierend auf Mathematik präzise erklärt, was genau ein Programm machen soll.[Har]

Diese Seminararbeit soll einen tieferen Einblick in den Themenbereich der Programmcodeverifikation schaffen. Dabei wird sowohl auf die Grundlagen als auch technisch-detaillierte Beispiele eingegangen. Als technisches Mittel wird hierfür der Proof-Assistent Coq verwendet.

2 Warum ist Formale Verifikation interessant?

Es existieren unzählige Beispiele für die Relevanz von Formaler Verifikation. Zum Beispiel jeder Computer besteht aus vielen elektronischen Hardwareteilen wie Prozessoren, Grafikkarte et cetera. Um diesen nutzen zu können, wird eine Hardware Description Language (HDL) eingesetzt. Diese beschreibt sowohl die Struktur als auch das Verhalte des elektronischen Bauteils.
Eine Firma X verkauft nun einen Computer mit einer fehlerhaften HDL-Software. Je nach Ausmaß des Bugs, kann dies gravierende Folgen nach sich ziehen.

Heutzutage gehen viele Computer-Nutzer davon aus, dass ein Prozessor, ein Compiler oder ein Programm 100% korrekt funktionieren. Doch wie ist sichergestellt, dass das Speichern in einem Editor oder das Compilieren von C Code auch das gewünschte Ergebnis liefert? Es ist eine Annahme, dass dies so funktioniert. Sehr wahrscheinlich wurden etliche Tests vor der Veröffentlichung der jeweiligen Software geschrieben. Trotzdem ist dies nur eine Annahme und kein Beweis, dass die Software auch wirklich so funktioniert, wie diese funktionieren soll.
Wenn beispielsweise diese HDL-Software jedoch formal verifiziert wäre, wäre bewiesen, dass, solange die Spezifikation korrekt ist, kein Fehler auftreten kann.
Somit lässt sich schlussfolgern, dass formale Verifikation vor allem in Software, die 100% korrekt sein muss benötigt wird. Hauptsächlich betrifft dies sicherheitskritische und intensiv genutzte Systeme.

3 Grundlagen

3.1 Theorem Provers

Ein Theorem Prover ist ein Programm. In diesem definiert ein Mensch Aussagen, welche das Tool versucht zu beweisen, wenn es möglich ist.

3.2 Proof Assistent

Ein Proof Assistent, welcher auch interaktiver Theorem Proofer genannt wird, ist ein Softwaretool, dass hilft formale Beweise durchzuführen. Dabei wird meistens ein interaktiver Editor verwendet, mit dem ein Mensch am Computer Schritt für Schritt Beweise schreiben kann. Der Unterschied zu einem Theorem Prover ist, dass die Software mit dem Bediener interagiert.

3.3 Übersicht

Zum Zeitpunkt dieser Arbeit existieren circa 17 verschiedene Tools für formale Verifikation. Dabei sind **ACL2**, **Isabelle** und **Coq** die bekanntesten.[Wie] In dieser Arbeit wird, wie bereits erwähnt, ausschließlich auf Coq eingegangen, da das Vergleichen den Umfang dieser Arbeit stark ausdehnen würde. Hintergrundwissen diesbezüglich ist ersichtlich im Paper von Freek Wiedijk und Dana Scott. Es ist unter `http://www.cs.ru.nl/~freek/comparison/comparison.pdf` veröffentlicht.

4 Coq

Der Proof Assistent Coq wurde erstmals im Mai 1989 veröffentlicht. Das National Institute for Research in Computer Science and Automation (INRIA) hat dessen Entwicklung bereits seit 1984 unterstützt.
Bestandteile in Coq-Projekt sind die funktionale Programmiersprache Coq selbst und eine Entwicklungsumgebung namens Coqide. Beides ist zum heutigen Zeitpunkt plattform-unabhängig und open-source erhältlich unter `https://github.com/coq/coq`. Die dort aktuell veröffentlichte Version ist 8.10.2.[COQ]
Coq ist zum Großteil in Ocaml geschrieben. Prüfen auf formale Richtigkeit von Beweisen ist das grundlegende Feature von der Programmiersprache. Dabei wird der Entwickler durch die eingebaute Entwicklungsumgebung interaktiv während des Schreiben eines Beweises unterstützt. Eine weitere interessante Funktionalität ist die Code Extraktion. Coq bietet diese für Ocaml, Haskel oder mit Hilfe von externen Bibliotheken auch für C an. Dies wird im Kapitel 7 detaillierter anhand eines Beispiels aufgegriffen. Dabei werden Funktionen formal verifiziert und anschließend in ausführbaren Ocaml Code extrahiert.

5 Coq Bewertung

Der Online-Dienst **Open Hub**, ehemals **Ohloh** genannt, katalogisiert open-source Softwareprojekte. Dabei werden Daten wie Projektname, Beschreibung und Sourcecode erfasst. Basierend auf diesen Daten erstellt Open Hub eine Statistik, die es ermöglicht, Codeanalyse, Projektmitarbeiter, Aktivitäten und eine Übersicht zu erhalten. Dabei werden auch viele weitere open source Projekte miteinander verglichen um aussagekräftige Statistiken und Aussagen treffen zu können.

In der Auswertung über OpenHAB steht beispielsweise, dass Coq aus über 30000 Beiträgen von 246 Entwicklern besteht. Weiterhin wird der Codestand mit qualitativ hochwertig beschrieben. Trotz der hohen Anzahl an Contributern, scheint die Anzahl an Beiträgen von diesem Jahr im Vergleich zum Vorjahr abzunehmen. Dies könnte einerseits bedeuten, dass das Interesse schwindet, andererseits ist es auch möglich, dass der Code weniger Bugfixes und Änderungen benötigt.[OH0]
Im Großen und Ganzen ist es eine aktuell sehr weit verbreite Sprache für formale Verifikation. Dabei wird sowohl Programmcode verifiziert, als auch mathematische Theoreme bewiesen, welche auch teilweise unabhängig zum Fachbereich Informatik sind. Die Anwendungsbereiche sind im Kapitel 8 genauer erläutert.

6 Programmatische Coq-Grundlagen

In diesem Kapitel werden die Grundlagen der Programmiersprache Coq erläutert. Dabei wird zuerst das spezielle Type-System eingegangen. Das dabei verwendete Konzept stellt die Basis für programmatische formale Verifikation dar. Weiterhin werden die darauf aufbauenden Coq-Features erklärt.
Das Kapitel der Grundlagen wird mit dem Zusammenspiel von Coq und der interaktiven CoqIde abgeschlossen. Um dieses und die fortlaufende Kapitel besser zu verstehen, lohnt es sich das umfangreiche Tutorial von Benjamin C. Pierce auf `https://softwarefoundations.cis.upenn.edu/lf-current/Basics.html#lab18` anzulesen. Der gezeigte Programmcode stammt teilweise aus dieser Lektüre.[dACCMGMGCHVSBY19]

6.1 Dependent Type Language

Java, C# und PHP verwendete Objekte als universellen Datentyp. C und Go nutzen hingegen Strukturen. Allerdings das Typ-System von Coq basiert weder auf Objekten, noch auf Strukturen - es ist eine zu Deutsch typ-abhängige Sprache.
Angenommen es gäbe eine Funktion, die irgendetwas mit einem User-Objekt macht. In den häufig verwendeten Sprachen, wie beispielsweise Java, sollten die ersten Zeilen einen Check, ob das User-Objekt null ist beinhalten (zu sehen in Codeblock 1).

```
1 public void doSomething(User user) {
2         if(user == null) {
3                 throw new Exception("Recieved empty user!");
4         }
5 ...
6 }
```

Codebeispiel 1: Java Funktion für den initialen Check auf null des User Objektes

Dadurch ist gewährleistet, dass wenn es zu einem Fehler während der Laufzeit kommen sollte, dieser kontrolliert abgefangen wird. Allerdings bedeutet dies auch, dass die Funktion den Status eines User-Objekts, welches null ist, als gültiges Objekt entgegennimmt.
Wären Funktionen, welche bereits zum Compilierungs-Zeitpunkt prüfen, ob das übergebene Objekt korrekt ist, nicht viel praktischer? Genau dies ist mit Hilfe von typ-abhängigen Sprachen umsetzbar. Dabei muss zuerst definiert werden, was ein **korrektes User-Objekt** charakterisiert. Beispielsweise könnte es bedeuten, dass das Objekt ungleich null oder eine bestimmte Rolle gesetzt ist. Dadurch könnte Beispielsweise die folgende Funktion in Form von Pseudocode geschrieben werden.

```
1 setRole: (user: User, role: String) -> userWithRole: User,
2                         where userWithRole.role == role;
```

Codebeispiel 2: Pseudocode Check auf null des User Objektes

Sodass diese Funktion bereits beim Compilieren auf Korrektheit geprüft werden kann, muss noch eine weitere grundlegende Voraussetzung erfüllt sein. Unter der Annahme, dass es insgesamt nur drei verschiedene Rollen gibt, können einzelne Typen für jede Rolle erstellt werden. Beispielsweise: UserWithAdminRole, UserWithSupportRole, UserWithUserRole. Jetzt ist es möglich, dass bereits der Compiler sicherstellen kann, dass die Funktion aus 2 korrekt abläuft. Zur Erklärung hilft folgendes Codebeispiel:

```
result: UserWithAdminRole = setRole (user, adminRole);
```

Codebeispiel 3: Pseudocode Check auf null des User Objektes

Jeder der oben genannten Typen stellt genau einen User mit einer bestimmten Rolle/ einem bestimmten Wert dar. Dies bedeutet typ-abhängig. Und nur dadurch ist es dem Compiler möglich den Code auf Korrektheit zu untersuchen.[Ben] Zusammenfassend gesagt, ist es durch typ-abhängige Sprachen möglich, zu prüfen, ob etwas wahr ist, bevor ein konkretes Objekt beziehungsweise eine Instanz mit Werten erstellt wurde. Die durchgeführten Checks, die normalerweise bei der Laufzeit durch laufen werden, sind dadurch bereits zur Compilezeit des Programms geprüft.

Diese Art von Typ-System benötigt einerseits für jeden Wert einen eigenen Typ, andererseits entstehen dadurch viele neue Optionen. Die viele Arbeit zur Erstellung eines Typs pro Wert, übernimmt der Compiler.
Der größte Vorteil ist die Vermeidung von Bugs. Zugriffe auf nicht existente Array Indizes, Nullpointer-Exceptions oder nicht endlicher Code sind faktisch keine Probleme in typ-abhängigen Sprachen - insofern die korrekten Checks durchgeführt wurden.

Des Weiteren ist es möglich fast alles auch mit dependent Types darzustellen. Beispielsweise eine Login Funktion, die keine Leerstrings erlaubt oder eine Funktion, die natürliche Zahlen dividiert, ohne eine Null zu erlauben, sind dadurch problemlos umsetzbar.

6.2 Basisbegriffe

6.2.1 Typdefinition

```
Inductive bool : Type :=
        | true
        | false.

Inductive day : Type :=
        | monday
        | tuesday
        | wednesday
        | thursday
        | friday
        | saturday
        | sunday.

Inductive nat : Type :=
        | O
        | S (n : nat).
```

Codebeispiel 4: Coq Typedefinition

Die Beispiele aus dem Codeblock 4 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.

6.2.2 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 5 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 5: 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.

```
1  (* Run function plus with 3 and 2. Result => 5 *)
2  Compute (plus 3 2).
3
4  (*  plus (S (S (S O))) (S (S O))
5          ==> S (plus (S (S O)) (S (S O)))
6  by the second clause of the match
7          ==> S (S (plus (S O) (S (S O))))
8  by the second clause of the match
```

```
9           ==> S (S (S (plus O (S (S O)))))
10 by the second clause of the match
11          ==> S (S (S (S O)))
12 by the first clause of the match
13 *)
```

Codebeispiel 6: Coq rekursive Funktion

Um ein tieferes Verständnis für die Rekursion in Coq zu bekommen, sind im Codeblock 6 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**.

6.3 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.

```
1  Theorem plus_1_l : forall n:nat, 1 + n = S n.
2  Proof.
3          intros n.
4          reflexivity.
5  Qed.
6
7  Theorem plus_n_O : forall n:nat, n = n + 0.
8  Proof.
9          intros n.
10         induction n as [| n' IHn'].
11                 - (* n = 0 *) reflexivity.
12                 - (* n = S n' *) simpl.
13                   rewrite <- IHn'.
14         reflexivity.
15 Qed.
```

Codebeispiel 7: Coq Beispielbeweis

Im Codeblock 7 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. Das Schlüsselwort **rewrite** wird in folgendem Unterkapitel erläutert.

6.4 Wie wird Coq verwendet?

Diese Kapitel beinhaltet einen Beispielbeweis und geht somit auf den praktischen Einsatz von Coq ein. Die zur Programmiersprache Coq parallel entwickelte Coqide wird hierfür verwendet. Wie bereits erwähnt, ist diese Entwicklungsumgebung interaktiv. Das bedeutet, dass der Nutzer Informationen vom Programm erhält. Diese können sowohl Hinweise, als auch Fehlermeldungen sein. Um sich die Coqide genauer vorstellen zu können, wird folgende Illustration 1 verwendet.

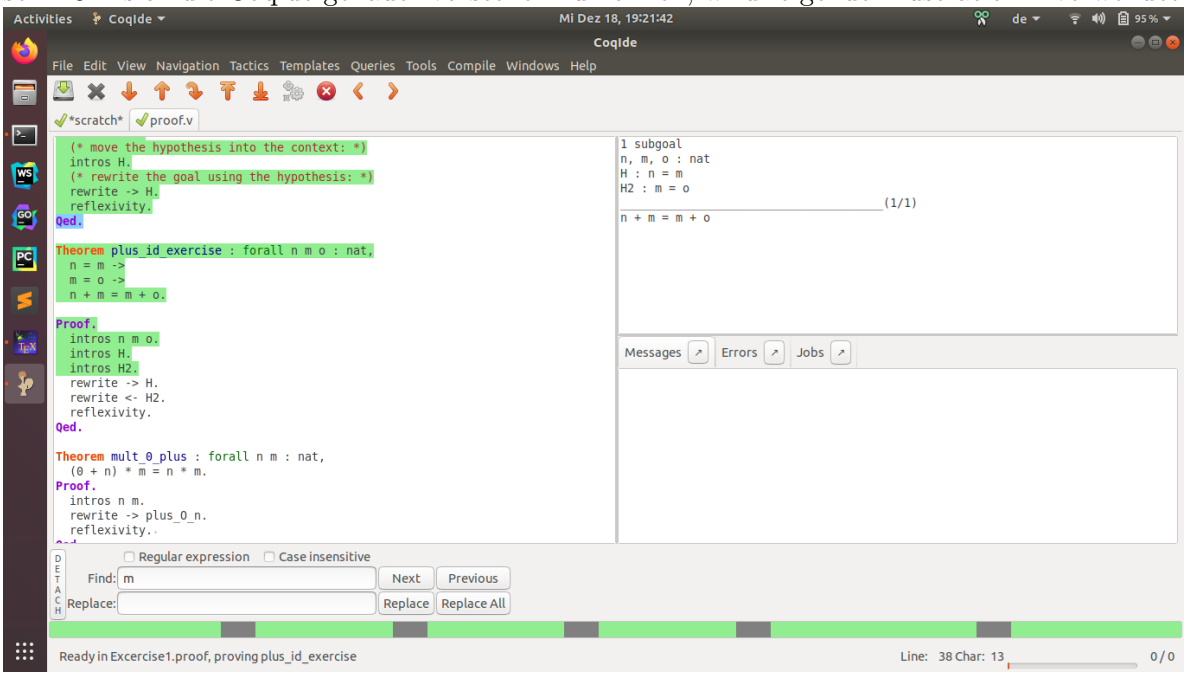

Abbildung 1: Coqide

Die Coqide bietet viele spezielle Features für formale Verifikation. Beispielsweise ist es möglich mit den Pfeilen in der Navigationsleiste die einzelnen Kommandos aus der linken Textbox auszuführen. Je nach Pfeil springt man einen Schritt vorwärts, bis zum Zeiger vorwärts oder auch rückwärts.

Die Entwicklungsumgebung stellt grundsätzlich drei Fenster dar. Dabei wird eines für den Programmcode genutzt. Die anderen beiden Fenster auf der rechten Seite dienen ausschließlich der Informationsausgabe.
Im Screenshot ist ein Beweis zu sehen. Dabei wird überprüft, wenn die natürlichen Zahlen **n, m, o** und die Beziehungen $\mathbf{n = m}$ und $\mathbf{m = o}$ gegeben sind, dass $\mathbf{n + m = m + o}$ gilt.
Zunächst fällt auf, dass der Coq-Code teilweise grün markiert ist. Dies symbolisiert den bereits erfolgreich ausgeführten Teil. Das Ausgabefenster rechts oben zeigt, hierfür die passende Ausgabe. Dabei werden über dem Trennstrich die gegebenen Zahlen und Hypothesen angezeigt. Zusätzlich ist noch die Anzahl an Zielen abgebildet. Den darunter stehende Ausdruck gilt es

zu beweisen.
Das dritte Fenster rechts unten dient zur Meldung von Hinweisen, Fehlern und Konsolenausgaben, wie beispielsweise von einer Suche.

Im Anschluss folgt das bereits gezeigte Codebeispiel 8 mit der ausführlichen Ausgabe der Coqide nach jedem Schritt. Diese ist durch die Kommentarblöcke, welche durch **(*** eingeleitet und durch ***)** beendet werden, gekennzeichnet. Wie bereits zuvor erklärt, gilt es, unter den gegebenen Umständen, **n + m = m + o** zu beweisen. Hierfür wird die **rewrite** Taktik eingesetzt. Nicht fachlich ausgedrückt bedeutet es, dass je nach Richtung des Pfeils die eine oder die andere Seite eingesetzt wird. Somit verändert sich die zu beweisende Aussage dynamisch im Coqide-Ausgabefeld. Nach dem ersten **Rewrite**, wird **n** mit **m** ersetzt. Das Einsetzen der zweiten Hypothese **H2** führt schlussendlich zu dem Ergebnis: **m + m = m + m**. Folglich kann das Ziel mit **reflexivity** bewiesen werden.

```
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
30 H : n = m
31 H2 : m = o
32 ____________________________________(1/1)
33 n + m = m + o*)
34
```

```
(* 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 8: Coq Beispielbeweis

7 Coq und Programmcode

Beim Zusammenspiel von Beweisen in Coq und Programmcode gibt es zwei verschiedene Richtungen. Einerseits können Theoreme bewiesen und dann in Programmiersprachen extrahiert werden. Andererseits ist es auch möglich, erst ein Programm zu entwickeln und anschließend dieses in Coq zu verifizieren.
Das dabei verwendete Prinzip lautet vereinfacht gesagt: **Es gibt eine Liste von Dingen, die eine Software tun soll. Hierfür wird Logik verwendet, um zu beweisen, dass diese Software auch genau diese Dinge tut.**
In den folgenden Unterkapiteln sind beide Wege beschrieben. Dabei wird die Richtung Proof zu Programm anhand eines genauen Codebeispiels erläutert.

7.1 Proof -> Programm

7.1.1 theoretisch

Wie bereits zuvor erwähnt, ist der erste Schritt die Erstellung einer Spezifikation, welche die Software beschreibt. Anschließend muss diese in mathematischer Form in ein Proof Tool geschrieben und bewiesen werden. Auf fehlerhafte Beweise weißt dieses Tool hin. Sobald jetzt ein Fehler in der realen Welt auftritt, sollte dieser mit diesem beschrieben Model abgefangen werden.
Schlussendlich müssen die bewiesenen Anforderungen der Spezifikation in Programmcode konvertiert werden. Dieser Prozess wird in folgendem Kapitel genauer erklärt.[Hel]

7.1.2 praktisch

In diesem Kapitel wird erklärt, wie Funktionen formal mit Coq bewiesen werden und anschließend in Ocaml extrahiert und ausgeführt werden. Ocaml („Categorical Abstract Machine +

ML“) ist eine sowohl funktionale, als auch objektorientierte Programmiersprache. ML bedeutet Meta-Language. Dies ist allerdings für diese Arbeit weniger relevant, da Ocaml lediglich als Beispiel für Code-Extraktion genutzt wird. Der Fokus dieser Arbeit liegt auf dem Prinzip und nicht auf einer speziellen Programmiersprache.
Bei dem Ansatz von Proof zum Programmcode musst zu Beginn eine **.v**-Datei erstellt werden. Darin wird der Coq-Code geschrieben. Das verwendete Beispiel 9 stellt ein Datentyp Paar von natürlichen Zahlen dar. Hierfür sollen verschiedenste Funktionalitäten implementiert werden. Dabei sind Funktionen wie **fst**, **snd** und **swap_pair**. Anschließend sind Beweise, die verschiedenste Eigenschaften der einzelnen Funktionen prüfen, dargestellt.

```
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).

Compute (fst (3,5)).

Definition swap_pair (p : natprod) : natprod :=
match p with
        | (x,y) => (y,x)
end.
```

Codebeispiel 9: Coq Funktionen für Paare aus natürlichen Zahlen

Zu Beginn wird der **induktive Typ natprod** definiert, welcher ein Paar repräsentiert. Im darauf folgenden Teil des Codebeispiels werden immer wieder die Schlüsselwörter **Check** und **Compute** verwendet. Mit Hilfe dieser Funktionen können stichprobenartig einzelne Werte in die Definitionen oder Typen eingesetzt werden. Anschließend wird das daraus resultierende Ergebnis ausgegeben.
Die Funktionen **fst** und **snd** geben jeweils x oder y eines Paares zurück. Weiterhin ist eine Notation für die Definition von Paaren dargestellt. Diese dient ausschließlich dafür, dass anstelle von **(pair x y)** auch **(x , y)** verwendet werden darf. Wenn eine solche Notation

nicht vorhanden wäre, müsse eine komplett neue Funktion **fst'** für den Syntax von **(x , y)** geschrieben werden. Notationen sind prinzipiell Aliase und tragen zur Kürzung des Aufwands bei.
Zuletzt wird die **swap_pair** Funktion, die den X- und Y-Wert eines Paares vertauscht zurückgibt, definiert.

Anschließend müssen Eigenschaften die diese Funktionen erfüllen sollen, bewiesen werden. Zuerst wird mit Hilfe der Theoreme surjective_pairing und surjective_pairing_stuck bewiesen, dass das Erstellen von einem neuem Paar demselben entspricht, wenn man fst und snd von diesem Paar nimmt und somit ein neues Paar bildet. Die zwei Funktionen unterscheiden sich lediglich in der Verwendung des Syntax von Paaren. Bei surjective_pairing_stuck wird außerdem die **destruct** Taktik benötigt.
Diese teilt normalerweise das Hauptziel des Beweises in die einzelnen Subgoals **n = O** und **n = S n'** auf. Sobald beide Ziele bewiesen wurden, akzeptiert Coq dieses Theorem. Hierbei ist der Unterschied zur Induktion, dass dabei **n = 0, n + 1** gecheckt wird und anschließend die Schlussfolgerung auf Korrektheit möglich ist.
Im Beispiel wird **destruct** jedoch nur zur Aufteilung von **p** in die natürlichen Zahlen n und m verwendet. Ab dieser Zeile entspricht dieser Beweis exakt dem Ersten, welcher von Anfang an natürliche Zahlen verwendet.

```
1  Theorem surjective_pairing' : forall (n m : nat),
2  (n,m) = (fst (n,m), snd (n,m)).
3  Proof.
4          simpl.
5          reflexivity.
6  Qed.
7
8  Theorem surjective_pairing_stuck : forall (p : natprod),
9  p = (fst p, snd p).
10 Proof.
11         intros p.
12         destruct p as [n m].
13         simpl.
14         reflexivity.
15 Qed.
16
17 Theorem snd_fst_is_swap : forall(p : natprod),
18 (snd p, fst p) = swap_pair p.
19 Proof.
20         intros p.
21         destruct p as [n m].
22         simpl.
23         reflexivity.
24 Qed.
25
26 Theorem fst_swap_is_snd : forall(p : natprod),
27 fst (swap_pair p) = snd p.
28 Proof.
```

```
29          intros p.
30          destruct p as [n m].
31          simpl.
32          reflexivity.
33  Qed.
```

Codebeispiel 10: Coq Beweise für Paar Funktionen

Die nächsten zwei Theorem prüfen bestimmte Eigenschaften der **swap_pair** Funktion. Zuerst wird Bewiesen, dass das zweite Element eines Paares in Verbindung mit dem ersten, das Ergebnis der Definition **swap_pair** ist. Letzterer Beweis stellt sicher, dass für jeden **natprod** das erste Element eines Paares nach einem Aufruf von **swap_pair** dem ursprünglich zweiten Wert des Paares entspricht.
Beide Theoreme werden identisch zu surjective_pairing_stuck bewiesen. Dabei wird der **natprod**-Typ ebenfalls mit Hilfe von **destruct** in zwei natürliche Zahlen aufgeteilt.

Um anschließend die formal bewiesenen Funktionen in Programmen nutzen zu können, muss die Datei, in der die Beweise geschrieben wurden, in Coq compiliert werden. Dafür muss folgender Befehl in die Kommandozeile eingeben werden: **coqc -Q . LF PaperPair.v**.
Coqc ist hierbei der Aufruf des Coq-Compilers. **-Q . LF** sorgt dafür, dass alle .v-Dateien aus dem Paket LF in andere Coq-Dateien importiert werden können. Ein Paket in Coq ist ähnlich zu anderen Programmiersprachen wie beispielsweise Java.
Für den nächsten Schritt in der Extraktion wird eine Coq-Datei benötigt, welche definiert, wie und was in welcher Sprache extrahiert werden soll. Diese muss ebenfalls über die Kommandozeile, wie zuvor beschrieben, compiliert werden.

```
1  Require Extraction.
2  Extraction Language OCaml.
3  Require Import ExtrOcamlBasic.
4  Require Import ExtrOcamlString.
5  Require Import Arith Even Div2 EqNat Euclid.
6
7  Extract Inductive nat => int [ "0" "Pervasives.succ" ]
8  "(fun fO fS n -> if n=0 then fO () else fS (n-1))".
9
10 Extraction "paperimpl.ml" fst snd swap_pair.
```

Codebeispiel 11: Coq Code extrahieren

Im Codeblock 11 ist zu sehen, dass verschiedene Dateien mit **Require** und **Require Import** importiert werden. **Extraction** und **ExtraOcamlBasic** sind beispielweise Standard-Features von Coq um Funktionen von Coq-Code in Ocaml-Code umzuwandeln.
Weiterhin wird in der Zeile **Extract Inductive nat => ...** ein Ausdruck verwendet, sodass OCaml den Typ der natürlichen Zahlen aus Coq verwenden kann. Allerdings ist der Coq-Code zu Ocaml Extraktor nicht formal verifiziert. Die Korrektheit wird trotzdem angenommen, da Coq großteils Ocaml geschrieben ist. Damit schlussendlich eine ausführbare Datei entsteht, muss definiert werden, welche Funktionen in welche Datei extrahiert werden sollen.

Der folgende Code 12 ist das Resultat, des zuvor gezeigten Coq-Codes.

```
1  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 12: Ocaml Code anpassen

Dadurch das der Coq-To-Ocaml-Extraktor nicht komplett formal verifiziert ist, kann es sein, dass der Code nicht 100% korrekt ist. Um sicherzustellen, dass dies der Fall ist, wurden im Nachhinein ein paar Tests geschrieben, welche im Codebeispiel 13 dargestellt werden. Diese Tests beschreiben einfache Funktionsaufrufe wie zum Beispiel das Erhalten des ersten und zweiten Wertes eines Paares. Anschließend werden die selben Funktionen noch einmal aufgerufen - allerdings auf ein neues Paar, dass du die swap_ pair Funktion entstanden ist. Zur Nachvollziehbarkeit werden dabei die jeweiligen Ergebnisse auf der Kommandozeile ausgegeben.

```
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;;

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

Codebeispiel 13: Ocaml Code anpassen

Ocaml-Code muss genauso wie C-Code compiliert werden. Folgender Befehl ermöglicht es aus der **paperimpl.ml** und der **paperimpl.mli** Datei funktionierenden compilierten Code zu erhalten. Dieser wird unter dem Namen **paperimp** im selben Verzeichnis abgelegt.

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

Codebeispiel 14: Ocaml Code compilieren

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

Codebeispiel 15: Ocaml code ausführen

In Codeblock 15 werden die Ausgaben der Tests dargestellt. Die ersten zwei Ergebnisse sind die Werte, welches mit den Werten **fst: 3** und **snd: 4** initiiert wurde. Die zweiten zwei Ausgaben stellen **fst** und **snd** des invertierten Paars dar.

7.2 Programm -> Proof

Generell ist es möglich fast jeden Programmcode formal zu verifizieren. Der Aufwand, der dafür aufzubringen ist, unterscheidet sich gewaltig. Um den Code einer Sprache vollständig formal beweisen zu können, muss theoretisch auch der Compiler der Sprache formal bewiesen sein. Ansonsten wäre zwar der Programmcode formal verifiziert, allerdings kann nicht garantiert werden, dass der Compiler dennoch Fehler beim übersetzen macht. Des Weiteren ist noch zu erwähnen, dass der Compiler nicht die niedrigste Software-Ebene repräsentiert. In den Schichten darunter sind beispielsweise noch Hardwarebeschreibungssprachen. Sobald dort ein gravierender Bug enthalten ist, kann der high-level Programmcode formal bewiesen sein und trotzdem fehlerhaft laufen.

Der Code aus der Programmiersprache C wird zum Beispiel in Maschinencode umgewandelt. Um diese Konvertierung durchführen zu können, haben Forscher von INRIA und der Princeton University die Verified Software Toolchain (VST), oder auch Princeton Tool Chain genannt, entwickelt. Auf folgender Illustration 2 sind die einzelnen Komponenten von der VST abgebildet. Zuerst wird dabei das C-Source-Programm in Verifiable C übersetzt.
„Verifiable C ist grundsätzlich korrekt. Das heißt, es ist nachgewiesen (mit einem maschinell geprüften Beweis im Coq-Proof-Assistenten), dass:
Egal welche beobachtbare Eigenschaft eines C-Programms Sie auch immer beweisen wollen. Unter Verwendung der Verifiable C-Programmlogik, wird diese Eigenschaft tatsächlich im Assemblerprogramm enthalten sein, welches der C-Compiler generiert.“
Mit **program logic** ist eine Art Hoare Logik gemeint, die ein leichteres Beweisen von Programmcode mit Pointern, Funktions-Pointern, Datenabstraktion und Datenstrukturen ermöglicht.[AWAwLB19a] Die Hoare Logik wurde durch den britischen Informatiker C. A. R. Hoare entwickelt und dient generell als Hilfsmittel für das Beweisen von Programmcode. Dabei werden logische Regeln aufgestellt, die es erlauben, Aussage in mathematischer Form über Computer-Programme zu treffen.
Die in orange dargestellte Komponenten bezüglich der **VST retargetable Seperation Logic** ergänzt die **program logic**. Zusätzlich können in diesem Schritt weitere **verified program analysis tools** integriert werden.

Der nächste Teil in der Princeton Tool Chain ist der **verified Compiler CompCert**, welcher von INRIA entwickelt wurde. Sowohl CompCert, als auch die ganze VST ist open-source erhältlich auf GitHub unter `https://github.com/PrincetonUniversity/VST` und

https://github.com/PrincetonUniversity/VST/tree/master/compcert.

- Übergang, was CompCert macht in einem Satz

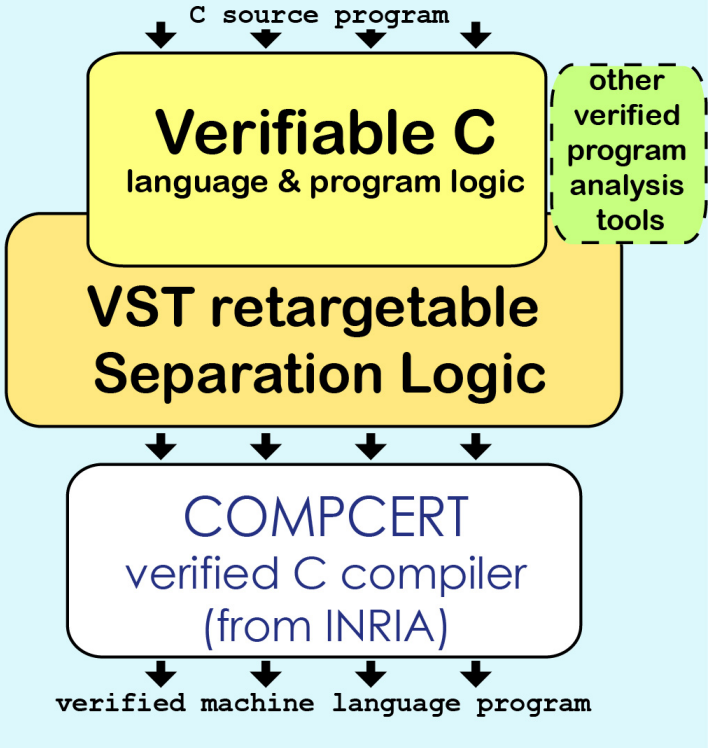

Abbildung 2: Verified Software Toolchain[PRI]

- Wie benutzt man nun praktisch die VST?
- 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.[AWAwLB19b]

8 Aktuelle Anwendung

8.1 Proofed Stack

- CompCert (C compiler) http://compcert.inria.fr
- Princeton VST https://github.com/PrincetonUniversity/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://vst.cs.princeton.edu
- https://news.yale.edu/2016/11/14/certikos-breakthrough-toward-hacker-resist

8.2 JSCert for ECMA 5

https://github.com/jscert/jscert

8.3 4-Farben Rätsel ist lösbar!

http://www.ams.org/notices/200811/tx081101382p.pdf

8.4 Satz von Feit-Thompson

Der Satz von Feit-Thompson sagt aus, dass jede endliche Gruppe ungerader Ordnung auflösbar ist. Er wurde 1963 von Walter Feit und John Griggs Thompson bewiesen.

Georges Gonthier gelang mit Kollegen nach sechsjähriger Arbeit 2012 die Verifikation des Beweises mit Coq (nicht das erste mal Bewiesen)

- Number of lines 170 000
- Number of definitions 15 000
- Number of theorems 4 200

https://web.archive.org/web/20161119094854/http://www.msr-inria.fr/news/feit-thomson-proved-in-coq/

8.5 CertiCoq

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

9 Aufwand in der Praxis

- Für jede Zeile Code wurden >= 5 extra Zeilen Proofs geschrieben.
- Code der geschrieben wurde, hat meistens sofort funktioniert. (Es ist eine neue Art zu programmieren)
- Projekt Ironclad von Microsoft Research. Talk von Bryan Parno 2014 https://www.usenix.org/node/186162
- 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)

10 Fazit

- https://medium.com/background-thread/the-future-of-programming-is-dependen
- Programmcodeverifikation nimmt vor allem in sicherheitskritischen Bereichen zu
- Programmcodeverifikation ist deutlich zeitintensiver (5mal)
- Mit dieser Verifikation kann 100%tige Garantie für funktionierende Software gewährleistet werden. Weil Assertions bereits zu Compilezeit durchgeführt werden und nicht mehr zur Laufzeit.

- Es hat großes Potential
- Wird sehr stark durch die Community weiterentwickelt
- Große Firmen nutzen es immer aktiver
- nur Software kann sicher gemacht werden. Hardware ist immer noch fehlbar!

11 Glossar

A Weiterführende Inhalte für Formale Verifikation

- Generating Correct Code with Coq by Rob Dickerson https://www.youtube.com/watch?v=95VlaZTaWgc
- Ironclad Apps: End-to-End Security via Automated Full-System Verification https://www.usenix.org/node/186162
- The Seventeen Provers of the World Compiled by Freek Wiedijk (and with a Foreword by Dana Scott) http://www.cs.ru.nl/~freek/comparison/comparison.pdf
- Coq in a Hurry by Yves Bertot https://cel.archives-ouvertes.fr/file/index/docid/459139/filename/coq-hurry.pdf
- VST - Verifiable C https://vst.cs.princeton.edu/veric/

Literatur

[AWAwLB19a] J. D. Andrew W. Appel with Lennart Beringer, Qinxiang Cao. Verifiable C, Applying the Verified Software Toolchain to C programs. S. 5, 2019.

[AWAwLB19b] J. D. Andrew W. Appel with Lennart Beringer, Qinxiang Cao. Verifiable C, Applying the Verified Software Toolchain to C programs. S. 8, 2019.

[Ben] M. Benčević. The Future of Programming is Dependent Types ? Programming Word of the Day. https://medium.com/background-thread/the-future-of-programming-is-dependent-types-programming-word-of-the-day-fcd5f2634878. Last visit: 25 Dez 2019.

[COQ] How to get it? https://coq.inria.fr/. Last visit: 15 Dez 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.

[Har] K. Hartnett. Hacker-Proof Code Confirmed . https://www.quantamagazine.org/formal-verification-creates-hacker-proof-code-20160920/. Last visit: 15 Dez 2019.

[Hel] A. Helwer. Formal Verification, Casually Explained. https://medium.com/@ahelwer/formal-verification-casually-explained-3fb4fef2e69a. Last visit: 16 Dez 2019.

[OH0] Coq proof assistant. https://www.openhub.net/p/coq. Last visit: 25 Dez 2019.

[PRI] Verified Software Toolchain. https://vst.cs.princeton.edu/VST-diagram.jpg. Last visit: 26 Dez 2019.

[Wie] F. Wiedijk. The Seventeen Provers of the World . http://www.cs.ru.nl/ freek/comparison/comparison.pdf. Last visit: 15 Dez 2019.