Technische Hochschule Rosenheim

Programmcodeverifikation mit Coq

Lukas Kiederle
Fakultät für Informatik

WS 2019/20

Kurzfassung

Aus Zeit- und Kostengründen beim Entwickeln und Testen von immer komplexer werdenden 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 grundlegenden 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 Ausblicks in Hinsicht auf die Verwendbarkeit von Proof Assistent Tools gezogen.

Schlagworte:

- Proof Assistent
- Coq
- Programcodeverifikation
- Formale Verifikation

Inhaltsverzeichnis

1 Motivation

Wenn heutzutage die Spezifikation eines Projektes in englischer Sprache 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 mathematisch präzise erklärt, was genau ein Programm machen soll.[Harb]

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 Programmiersprache wird hierfür der Proof-Assistent Coq verwendet.

2 Relevanz formaler Verifikation

Es existieren unzählige Beispiele für die Relevanz von formaler Verifikation. Jeder Computer besteht beispielsweise aus vielen elektronischen Hardwareteilen wie Prozessoren, Grafikkarte et cetera. Zur Benutzung wird eine Hardware Description Language (HDL) eingesetzt, welche sowohl die Struktur als auch das Verhalten der elektronischen Bauteile beschreibt. Falls eine Firma nun einen Computer mit einer fehlerhaften HDL-Software verkauft, kann dies gravierende Folgen nach sich ziehen. Daher ist es wichtig, dass sowohl die Bauteile als auch das Zusammenspiel von Hardware und Software ausgiebig getestet werden.

Heutzutage gehen die meisten Computer-Nutzer davon aus, dass ein Betriebssystem, ein Compiler oder andere Software zu 100% korrekt funktionieren. Doch wie ist sichergestellt, dass das Speichern in einem Editor oder das Kompilieren von C Code auch in jedem Fall das gewünschte Ergebnis liefert?
Sehr wahrscheinlich wurden viele verschiedene Tests bereits vor der Veröffentlichung der jeweiligen Software durchgeführt. Die bekanntesten sind Unit-, Integration- und manuelle Tests. Dabei versucht man sowohl Standard- als auch Grenz- und Sonderfälle abzudecken, aber natürlich nicht jeden Einzelfall. Würde man einen kompletten Test durchführen, müsste beispielsweise für eine simple Funktion, die zwei Integer addiert, jeder beliebige Wert, den dieser annehmen könnte, getestet werden. Dies ist nicht in annehmbarer Laufzeit mit den oben genannten Testformen durchführbar.
Nichts desto trotz schränken viel Tests ein Fehlverhalten der jeweiligen Software ein. Dies ist für vielerlei Anwendungen bereits ausreichend. Allerdings vor allem in sicherheitskritischen und intensiv genutzten Systemen sollte diese zu 100% korrekt funktionieren. Um alle Fälle abzudecken, muss eine andere Technik verwendet werden. Hier kommt die formale Verifikation zum Einsatz. Die dabei verwendete Sprache ist mathematisch aufgebaut und erlaubt es somit Konstrukte, wie beispielsweise **für alle natürlichen Zahlen gilt** niederzuschreiben.
Generell ist es möglich fast jeden Programmcode formal zu verifizieren. Der Aufwand, der dafür aufzubringen ist, unterscheidet sich von Fall zu Fall. 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 keine 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.
Zum Zeitpunkt dieser Arbeit existieren circa 17 verschiedene Tools für formale Verifikation. Dabei sind **Coq**, **Isabelle** und **ACL2** die bekanntesten.[Wie] In dieser Arbeit wird ausschließlich auf Coq eingegangen.

3 Einführung in Coq

3.1 Begriffe

Ein **Theorem Prover** ist ein Programm. In diesem werden Aussagen definiert, die das Tool zu beweisen versucht, falls es möglich ist.

Ein **Proof Assistent**, welcher auch interaktiver Theorem Prover genannt wird, ist ein Softwaretool, das hilft formale Beweise durchzuführen. Dabei wird ein interaktiver Editor verwendet, mit dem es möglich ist, programmatisch Schritt für Schritt maschinell Beweise zu prüfen. Die Software interagiert dabei mit dem Bediener.

3.2 Das Projekt 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.[COQb] Der Name leitet sich vom französischen Wort Coq (zu Deutsch Hahn) ab. Traditionell werden Entwicklungswerkzeuge in Frankreich nach Tieren benannt. Außerdem erinnert der Name an den französischen Mathematiker und Informatiker Thierry Coquand, der das Konstruktionskalkül mitentwickelt hat, was die in Coq verwendete Typentheorie ist.[PM] Bestandteile des Coq-Projekts sind die funktionale Programmiersprache Coq selbst und eine Entwicklungsumgebung namens CoqIde. Beides ist zum heutigen Zeitpunkt plattformunabhängig und open-source erhältlich unter `https://github.com/coq/coq`. Die dort aktuell veröffentlichte Version ist 8.10.2.[COQa]
Coq ist größtenteils in Ocaml geschrieben. Das grundlegende Feature der Programmiersprache ist das Prüfen auf formale Richtigkeit von Beweisen. Dabei wird der Entwickler durch die eingebaute Entwicklungsumgebung CoqIde interaktiv während des Schreibens eines Beweises unterstützt. Eine weitere 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 5 detailliert anhand eines Beispiels aufgegriffen. Dabei werden Funktionen formal verifiziert und anschließend in ausführbaren Ocaml Code extrahiert.
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 6 genauer erläutert.

3.3 Bewertung von Coq mittels Open Hub

Der Online-Dienst **Open Hub**, ehemals **Ohloh** genannt, katalogisiert open-source Softwareprojekte. Für jedes Projekt werden Daten wie Name, 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 fließen auch Daten weiterer open-source Projekte ein, um aussagekräftige Statistiken und Aussagen treffen zu

können.

In der Auswertung steht beispielsweise, dass Coq aus über 30000 Beiträgen von 246 Entwicklern entstanden ist. Weiterhin wird der Codestand mit qualitativ hochwertig bewertet. Trotz der großen Menge an Contributern, scheint die Anzahl an Beiträgen in 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]

4 Programmatische Coq-Grundlagen

In diesem Kapitel werden die Grundlagen der Programmiersprache Coq erläutert. Es wird zuerst auf das spezielle Type-System eingegangen. Das dabei verwendete Konzept stellt die Basis für programmatische formale Verifikation dar. Weiterhin werden die darauf aufbauenden Coq-Features und das Zusammenspiel von Coq und der interaktiven CoqIde erklärt.

4.1 Dependent Type Sprache

Java, C# und PHP verwenden Objekte als universellen Datentyp. C und Go nutzen hingegen Strukturen. Das Typ-System von Coq basiert weder auf Objekten, noch auf Strukturen - es ist eine dependent Type Sprache (zu Deutsch Typen die von etwas abhängen).
Angenommen es gäbe eine Funktion, die irgendetwas mit einem User-Objekt/einer User-Struktur macht. In den häufig verwendeten Sprachen, wie beispielsweise Java, sollten die ersten Zeilen einen Check beinhalten, ob das User-Objekt null ist. (siehe 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.
Praktischer wären Funktionen, welche bereits zum Compilierungs-Zeitpunkt prüfen, ob das übergebene Objekt korrekt ist. Dies ist mit Hilfe von dependent type Sprachen umsetzbar. In Coq heißt diese **Gallina**. Dabei muss zuerst definiert werden, was einen **korrekten User** charakterisiert. Beispielsweise könnte es bedeuten, dass er ungleich null oder einer bestimmte Rolle zugeordnet ist. Dadurch könnte die folgende Funktion in Form von Pseudocode geschrieben werden.

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

Codebeispiel 2: Pseudocode Check auf null des User Objektes

Die Funktion **setRole** aus dem Beispiel 2 zeigt, dass ein User mit einer alten Rolle **oldRole** eine neue Rolle **newRole** bekommt. Dies ist durch die Where-Klausel aus Zeile 2 immer sicher gestellt. Dabei ist es wichtig zu verstehen, dass **nicht** die Werte verglichen werden, sondern Typen, welche den Wert, der noch nicht zugewiesen wurde repräsentieren. Im Beispiel wird geprüft, ob nach dem ausführen der Funktion der neue Typ userWithRole einen Typ **role** hat, welcher ungleich **null** ist. Zur Verdeutlichung könnte die Where-Klausel auch **where typeof(userWithRole.role) == Role** lauten.
oldRole wäre ein impliziter Parameter, da er sich aus dem Typ von User ergibt. Es ist notwendig den Role-Parameter der Funktion zu übergeben, sodass der User immer einen Role-Parameter hat.

Es muss noch eine weitere grundlegende Voraussetzung erfüllt sein, damit diese Funktion bereits beim Kompilieren auf Korrektheit geprüft werden kann. Ein Typ repräsentiert normalerweise unterschiedliche Ausprägungen. Allerdings wird für dependent type Sprachen für jede Ausprägung ein eigener Typ benötigt.
Unter der Annahme, dass es beispielsweise insgesamt drei verschiedene Rollen (adminRole, supportRole, userRole) gibt, können einzelne Typen für jede Rolle erstellt werden. Jetzt ist es möglich, dass bereits der Compiler sicherstellen kann, dass die Funktion aus Codeblock 2 korrekt abläuft. Zur Erklärung hilft folgendes Beispiel:

```
1 result: userWithRole = setRole (userRole, user, adminRole);
```

Codebeispiel 3: Pseudocode Check auf null des User Objektes

Jeder der oben genannten Typen stellt genau einen User mit einer bestimmten Rolle mit einem bestimmten Wert dar.
Zusammenfassend gesagt, ist es durch dependent type 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 durchlaufen werden, sind dadurch bereits zur Compilezeit des Programms geprüft.[Ben]

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 dependent type Sprachen, falls die korrekten Beweise durchgeführt wurden.
Generell ist es möglich fast alles mit dependent Types darzustellen. Beispielsweise eine Login-Funktion, die keine Leerstrings erlaubt, oder eine Funktion, die natürliche Zahlen dividiert ohne eine Null zu erlauben, ist dadurch problemlos umsetzbar.

4.2 Verwendung in Coq von Dependent Type Sprache

Coq verwendet dependent type Sprache um logische Aussagen auf Typen abzubilden. Eine Aussage ist ein Typ und diese ist wahr, wenn eine Instanz des Typs hingeschrieben werden kann. Dafür existieren in Coq die zwei induktiven Typen **True** und **False**.

```
1  Inductive True : Type := I.
2
3  Inductive False : Type := .
4
5  Inductive and (A B : Type) : Type :=
6  conj : A -> B -> and A B.
7
8  Inductive or (A B : Type) : Type :=
9  or_introl : A -> or A B
10 | or_intror : B -> or A B.
```

Codebeispiel 4: Coq Basis Typen

Im Codeblock 4 sind diese in den Zeilen 1 und 3 beschrieben. Ein Type True kann durch den Konstruktor **I** instantiiert werden. Ein Typ False hingegen besitzt keine Konstruktoren.
Weiterhin nutzen logische Aussagen Verbindungen, wie beispielsweise **Und** und **Oder**. Die in Coq verwendeten Definition sind die induktiven Typen **and** und **or**. Um ein **and**-Typen instantiieren zu können, muss ein Instanz beider Typen A und B angegeben werden. Bei Oder hingegen reicht eine der beiden Aussagen. In Coq ist es hierbei nicht möglich falschen Aussagen anzugeben.

Ein Theorem ist eine Funktion, die aus den forall-Werten (zum Beispiel: forall n: nat) und Voraussetzungen eine Instanz der Theoremausgabe kreiert. Ein Beispiel hierfür ist das Theorem **nat_rect**, das die Induktion über natürliche Zahlen darstellt (siehe Codeblock 5).

```
1 nat_rect : forall P : nat -> Type,
2 P 0 ->
3 (forall n : nat, P n -> P (S n)) ->
4 forall n : nat, P n
```

Codebeispiel 5: Coq Induktion der natürlichen Zahlen

Es bedeutet, dass für alle Aussagen P über natürlich Zahlen folgendes gilt:
„Wenn die Aussage für 0 gilt und wenn für alle n aus der Aussage für n die Aussage für (S n) folgt, so gilt die Aussage für alle n.“

Zusammenfassend bedeutet dies, dass in Coq die Induktion nur über einfache rekursive Funktionen abgebildet werden. Diese wird von Coq selbst für jeden Induktiven Type erstellt. Bei einer Anwendung der Taktik **induction**, wird zu einem bestimmten Zeitpunkt diese Funktion aufgerufen.
Außerdem ist die Induktion nur mittels dependent Type Sprache abzubilden. Denn nur so ist gewährleistet, dass zum Beispiel eine **and**-Aussage von den beiden Teilaussagen A und B abhängig ist.

4.3 Basisbegriffe

4.3.1 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 6: Coq Typedefinition

Die Beispiele aus dem Codeblock 6 stellen drei Typedefinitionen in Coq dar. Ersteres ist ein klassischer Bool, der true oder false annehmen kann. Der zweite Typ day repräsentiert alle Wochentage.
Die letzte Definition wird verwendet, um alle natürlichen Zahlen darzustellen. **S (n : nat)** stellt den Successor, zu Deutsch die Nachfolgefunktion, dar, welche eine Konstruktorfunktion ist. Dadurch kann 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**.
Durch das Schlüsselwort **Inductive** ist es möglich Komposition abzubilden.

4.3.2 Funktionen

In Coq gibt es mehrere Arten von Funktionstypen. Mit dem Keyword **Definition** können einfache Funktionen dargestellt werden. Rekursionen hingegen sind mit **Fixpoint** oder ähnlichen Wörtern beschrieben. Anstelle dessen, können auch **Example, Lemma, Fact oder Remark** stehen. Auch **Theoreme** sind Funktionen in Coq - sie bilden Werte auf Aussagen ab. Dabei wird mithilfe des Allquantors die Korrektheit einer Funktion für alle Elemente einer Menge bewiesen. Im Codeblock 7 ist für die unterschiedlichen Funktionstypen jeweils ein Beispiel dargestellt.

```
1   Definition pred (n : nat) : nat :=
2   match n with
3   | O => O
4   | S n' => n'
5   end.
6
7   Definition minustwo (n : nat) : nat :=
8   match n with
9           | O => O
10          | S O => O
11          | S (S n') => n'
12  end.
13
```

```
14 Theorem plus_O_n' (n : nat) :
15 0 + n = n.
16
17 Fixpoint plus (n : nat) (m : nat) : nat :=
18 match n with
19         | O => m
20         | S n' => S (plus n' m)
21 end.
```

Codebeispiel 7: Coq Funktionen

Die erste Funktion **pred** gibt den Vorgänger (Predecessor) einer natürlichen Zahl zurück. Die zweite Funktion **minustwo** zieht von einer eingegebenen natürlichen Zahl zwei ab. Allerdings ergibt **0 - 2** und **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 = n“. Im folgenden Kapitel wird gezeigt, wie eine solche Funktion mathematisch bewiesen werden kann.
Fixpoint plus ist eine Addition zweier natürlicher Zahlen. Hierfür wird Rekursion verwendet.

```
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 8: Coq rekursive Funktion

Zum besseren Verständnis für den Ablauf der Rekursion in Coq, sind im Codeblock 8 die einzelnen Schritte in einem Kommentar-block (gekennzeichnet durch (* *)) abgebildet. In Zeile stellt Coq, wie bereits bei den Typdefinitionen der natürlichen Zahlen gezeigt, die Zahlen zwei und drei mittels der Nachfolger-Funktion dar. Anschließend beginnt die Rekursion. Solange **n > 0**, wird 1 mehr zum Endergebnis gezählt. Wenn **n = 0**, dann wird, wie in den letzten zwei Zeilen im Codeblock dargestellt, das plus durch **m** ersetzt. Somit ergibt **plus 3 2 => 5**.

4.4 Beweise und Taktiken

Um zu prüfen, dass die definierten Funktionen mathematisch korrekt sind, stellt der Proof Assistent verschiedene Taktiken zur Verfügung. Diese werden zwischen den **Proof.** und **Qed.**. 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 **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 9: Coq Beispielbeweis

Im Codeblock 9 sind zwei Theoreme bewiesen. **Theorem plus_1_l** prüft ob **1 + n** der Nachfolger-Funktion von **n** entspricht. Für den Beweise werden zwei Taktiken genutzt. **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 ein **simpl** zur Vereinfachung (z.B.: 0 + n => n) aus. **Reflexivity** steht somit meistens am Ende eines Beweises mit dem Format **links = rechts**, sodass er abgeschlossen werden kann. Dafür gibt es auch anderweitige Möglichkeiten.
Die zweite Funktion **plus_n_O** validiert das **n = n + 0** gilt und wird mit Hilfe der Taktik **induction** gelöst. Diese teilt die Aussage in zwei Subgoals (zu Deutsch Teilziele) auf, passend zu Induktion. Dabei wird geprüft, dass **n = 0** (Zeile 11) gilt und das aus der Aussage für **n n + 1** (Zeile 12-13) folgt. Anschließend muss jedes einzelne Ziel überprüft werden. Dies verschiedenen Ebenen von Subgoals sind mit Hilfe von **-, +, *** gekennzeichnet. Der ***-Operator** ist dabei nicht mit der vorher beschriebenen Kommentar-Notation **(* *)** zu verwechseln. Ein **-** wird bei der ersten Teilziel-Ebene verwendet. Für das Adressieren weiterer Subgoals von Subgoals werden **+** und ***** genutzt. Das Schlüsselwort **rewrite** wird in folgendem Unterkapitel erläutert.

4.5 Anwendung von Coq

Dieses Kapitel beinhaltet einen Beispielbeweis und geht somit auf den praktischen Einsatz von Coq ein. Dabei wird die interaktive Entwicklungsumgebung CoqIde verwendet. Das bedeutet, dass der Nutzer während des Beweisvorgangs Informationen vom Programm erhält. Dies können sowohl Hinweise als auch Fehlermeldungen sein. Um sich die CoqIde genauer vorstellen zu können, wird folgende Illustration 1 verwendet.

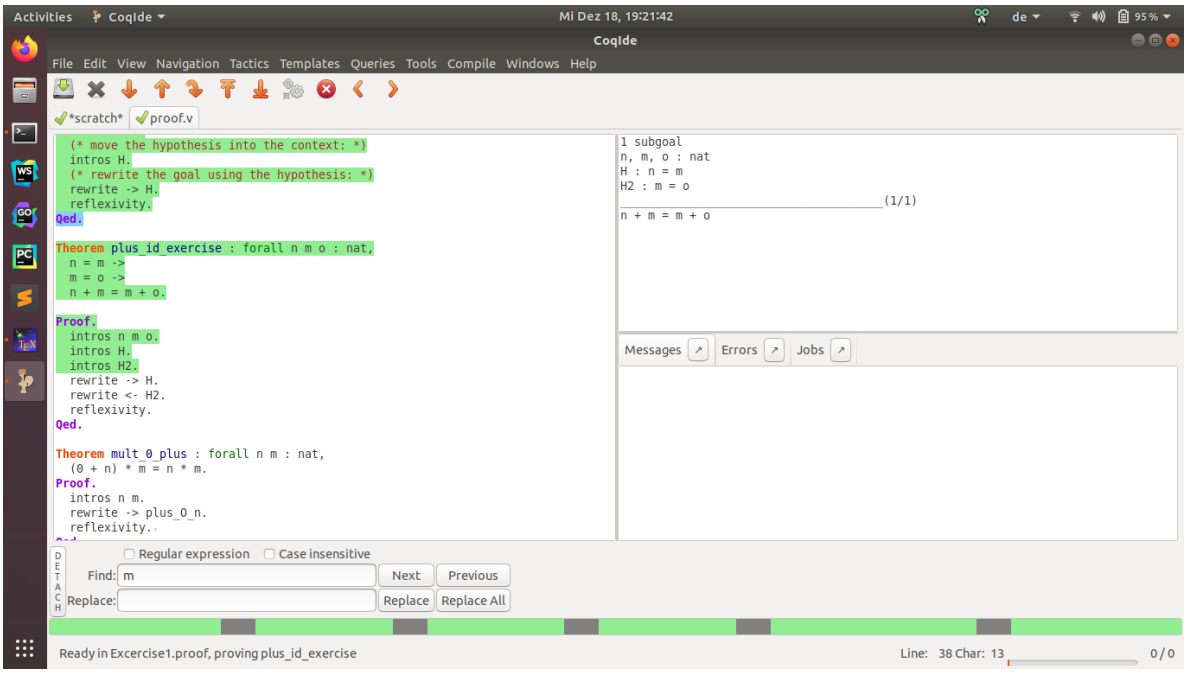

Abbildung 1: Coqide

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 Mauszeiger vorwärts oder auch rückwärts.

Die Entwicklungsumgebung besteht grundsätzlich aus drei Fenster. 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 Variablen und Hypothesen angezeigt. Zusätzlich ist noch die Anzahl an Zielen abgebildet. Den darunter stehenden 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 Codeblock 10 folgt die detaillierte Ausgabe der CoqIde nach jedem Schritt des in Abbildung 1 gezeigten Beispiels. Diese ist durch die Kommentarblöcke gekennzeichnet, welche durch **(* result:** eingeleitet und durch ***)** beendet werden. Wie bereits zuvor erwähnt, gilt es $\mathbf{n + m = m + o}$ zu beweisen. Die gegebenen Hypothesen sind $\mathbf{n = m}$ und $\mathbf{m = o}$.
Nachdem das Theorem durch die CoqIde eingelesen wurde (grün markiert in Abbildung 1), ergibt sich die Ausgabe von Zeile 7 - 11. Darin ist das fast unveränderte Theorem abgebildet. Wie bereits im vorherigen Kapitel erklärt, gilt es alles unter dem Trennstrich zu beweisen.
Als nächstes wird der Beweis durch das Schlüsselwort **Proof** in Zeile 13 gestartet. Mit Hilfe von **intros** in Zeile 15 sind die drei Variablen n m o in den Kontext gesetzt. Das Resultat

dieser Operation ist in Zeile 19 zu erkennen. Die Variablen werden nun als gegeben angesehen. Dasselbe Kommando ermöglicht es die zwei Hypothesen **H** und **H2** aufzustellen. Somit ergibt sich die Ausgabe in den Zeilen 27 bis 33. Die gegebenen Variablen und Hypothesen sind oberhalb der Linie aufgelistet und darunter ist das einzige zu beweisende Ziel **n + m = m + o** zu sehen. Die in den zwei Zeilen 36 und 43 verwendete Taktik **rewrite** nutzt **H** und **H2**. **Rewrite** bedeutet vereinfacht ausgedrückt, dass je nach Richtung des Pfeils die eine oder andere Seite eingesetzt wird. In Zeile 41 ist das Ergebnis nach dem Einsetzen von der Hypothese **H** dargestellt, wobei alle **n**'s durch **m**'s ersetzt wurden. Selbiges Resultat entsteht nach der Verwendung von **H2**. Allerdings werden dabei alle **o**'s durch **m**'s ersetzt. In beiden Ausgaben sind nur ...zu sehen, da der gegebene Teil unverändert bleibt. Abschließend kann das Ziel mit **reflexivity** bewiesen werden, da **m + m** dasselbe ist, wie **m + m**.

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

```
37
38 (* result:
39 ...
40 ____________________________________(1/1)
41 m + m = m + o
42 *)
43         rewrite <- H2.
44
45 (* result:
46 ...
47 ____________________________________(1/1)
48 m + m = m + m
49 *)
50         reflexivity.
51 Qed.
```

Codebeispiel 10: Coq Beispielbeweis

5 Coq und Programmcode

Beweise in Coq können in zwei verschiedenen Richtungen in Verbindung mit Programmcode verwendet werden. Einerseits ist es möglich, Theoreme zuerst in Coq zu beweisen und anschließend in Programmiersprachen zu extrahieren. Andererseits kann auch erst ein Programm entwickelt werden, um es anschließend 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 in Coq zu Programm anhand eines genauen Codebeispiels erläutert.

5.1 Proof -> Programm

5.1.1 Theorie

Wie bereits erwähnt, ist der erste Schritt die Erstellung einer Spezifikation, welche die Software beschreibt. Anschließend muss diese in mathematischer Form in Coq geschrieben und bewiesen werden. Fehlerhafte Beweise lässt der Proof-Assistent nicht zu. Dadurch sollten alle Fälle, welche in der realen Welt auftreten, mit diesem beschriebenen Modell abdeckt sein.
Schlussendlich müssen die bewiesenen Anforderungen der Spezifikation von Coq-Code in Programmcode konvertiert werden. Dieser Prozess wird in folgendem Kapitel genauer erklärt.[Hel]

5.1.2 Praxis

In diesem Kapitel wird erklärt, wie Funktionen formal mit Coq bewiesen werden und anschließend in Ocaml extrahiert und ausgeführt werden. Ocaml („Objective Categorical Abstract Machine + ML“) ist eine sowohl funktionale als auch objektorientierte Programmiersprache. Früher bedeutete ML Meta-Language, heutzutage Maschine Learning. In dieser Arbeit dient Ocaml lediglich als Beispiel für die Code-Extraktion aus Coq. Allerdings ist der Coq-Code zu

Ocaml Extraktor nicht formal verifiziert. Die Korrektheit wird trotzdem angenommen, da Coq größtenteils in Ocaml geschrieben ist.
Das verwendete Beispiel 11 stellt einen Datentyp Paar von natürlichen Zahlen dar. Hierfür sollen verschiedene Funktionalitäten implementiert werden, die die Ausgabe des ersten und zweiten Wertes und das Vertauschen ermöglicht. Zunächst muss eine **.v**-Datei erstellt werden, in welche der Coq-Code geschrieben wird. Anschließend sind sowohl die Funktionen als auch die dazugehörigen Beweise, die die Eigenschaften der einzelnen Funktionen prüfen, dargestellt.

```
1  From LF Require Export Induction.
2
3  Inductive natprod : Type :=
4          | pair (n1 n2 : nat).
5
6  Definition fst (p : natprod) : nat :=
7  match p with
8          | pair x y => x
9  end.
10
11 Definition snd (p : natprod) : nat :=
12 match p with
13         | pair x y => y
14 end.
15
16 Compute (fst (pair 3 5)).
17 Compute (snd (pair 5 7)).
18
19 Notation "( x , y )" := (pair x y).
20
21 Compute (fst (3,5)).
22
23 Definition swap_pair (p : natprod) : natprod :=
24 match p with
25         | (x,y) => (y,x)
26 end.
```

Codebeispiel 11: Coq Funktionen für Paare aus natürlichen Zahlen (PaperPair.v)

Zu Beginn wird der **induktive Typ natprod** definiert, welcher ein Paar natürlicher Zahlen repräsentiert. Die Funktionen **fst** und **snd** geben jeweils x oder y eines Paares zurück. Im darauffolgenden Teil des Codebeispiels wird mehrmals das Schlüsselwort **Compute** verwendet. Mit Hilfe dieser Funktion können stichprobenartig einzelne Werte in die Definitionen eingesetzt werden. Dies dient zu Test- und Debugging-Zwecken. Beispielsweise Compute aus der Zeile 16 zeigt den Wert 3 in der CoqIde Ausgabe.
Weiterhin ist eine Notation für die Definition von Paaren in Zeile 19 dargestellt. Diese dient ausschließlich dafür, dass anstelle von **(pair x y)** auch **(x , y)** verwendet werden darf, wie in Zeile 21 zu sehen ist. Ohne eine solche Notation müsste eine komplett neue Funktion **fst'** für die Syntax von **(x , y)** geschrieben werden. Notationen sind prinzipiell Aliase und tragen zur Kürzung des Programmieraufwands bei.
Zuletzt wird die **swap_pair** Funktion definiert, die den x- und y-Wert eines Paares vertauscht

zurückgibt.

Anschließend werden die Eigenschaften in Codebeispiel 12 bewiesen, welche diese Funktionen erfüllen sollen. Zuerst wird mit Hilfe der Theoreme **surjective_pairing** und **surjective_pairing_stuck** bewiesen, dass das Erstellen von einem neuen Paar demselben entspricht, wenn man fst und snd von diesem Paar nimmt und daraus ein neues Paar bildet. Die zwei Funktionen unterscheiden sich lediglich in der Verwendung der Syntax von Paaren. Beim Beweis von **surjective_pairing_stuck** wird außerdem die **destruct** Taktik benötigt.
Diese teilt normalerweise das Hauptziel des Beweises in die einzelnen Subgoals **n = 0** und **n = S n'** auf. Sobald beide Ziele bewiesen wurden, akzeptiert Coq dieses Theorem. Hierbei ist der Unterschied der Induktion, dass dabei **n = 0** und **n + 1** gecheckt wird und anschließend die Schlussfolgerung auf Korrektheit möglich ist.
Im Beispiel wird **destruct** jedoch nur zur Aufteilung von dem Paar **p** in die natürlichen Zahlen n und m verwendet. Ab dieser Zeile entspricht der 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 12: Coq Beweise für Paar Funktionen (PaperPair.v)

Die letzten zwei Theoreme prüfen bestimmte Eigenschaften der **swap_pair** Funktion. Zuerst wird mit **snd_fst_is_swap** bewiesen, dass das zweite Element eines Paares in Verbindung mit dem ersten das Ergebnis der Definition **swap_pair** ist. Der letzte Beweis **fst_swap_is_snd** 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.
In der Praxis ergeben sich meist die Theorem aus einem Modul, welche zu beweisen sind, automatisch indem geprüft wird, wie ein anderes Modul dieses eine benötigt.

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. **PaperPair.v** repräsentiert die Datei mit den Inhalten von den zuvor gezeigten Codeblöcken 11 und 12.
Für den nächsten Schritt in der Extraktion muss eine weitere Coq-Datei mit dem Code aus aus Codeblock 13 erstellt werden, welche definiert, wie und was in welcher Sprache extrahiert werden soll.

```
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 13: Coq Code extrahieren (PaperPairExtraction.v)

Darin ist zu sehen, dass verschiedene Dateien mit **Require** und **Require Import** importiert werden. **Extraction** und **ExtraOcamlBasic** sind beispielsweise Standard-Features von Coq, um Funktionen von Coq-Code in Ocaml-Code umzuwandeln.
Weiterhin wird in der Zeile 7 und 8 **Extract Inductive nat => ...** ein Ausdruck verwendet, der die natürlichen Zahlen auf integers in Ocaml mappet. Dieser Vorgang ist nicht zu 100% sicher, da Integer in Ocaml beschränkt sind. Damit schlussendlich eine ausführbare Datei entsteht, muss definiert werden, welche Funktionen in welche Datei extrahiert werden sollen, was mit Zeile 10 geschieht.
Anschließend muss diese Datei ebenfalls über die Kommandozeile, wie zuvor beschrieben, compiliert werden.
Der folgende Ocaml-Code 14 und 15 ist das Resultat des zuvor gezeigten Coq-Codes. Dieser

wurde mittels CoqIde ausgeführt und in den Dateien **paperimpl.ml** und **paperimpl.mli** gespeichert. **.ml** enthält dabei die Programmlogik und **.mli** ist eine Header-Datei, welche diese Logik beschreibt.

```
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 14: Ocaml Code Resultat (paperimpl.ml)

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

val fst : natprod -> int

val snd : natprod -> int

val swap_pair : natprod -> natprod
```

Codebeispiel 15: Ocaml Code Resultat (paperimpl.mli)

Dadurch dass der Coq-To-Ocaml-Extraktor nicht komplett formal verifiziert ist, kann es sein, dass der Code nicht zu 100% korrekt ist. Um die Korrektheit sicherzustellen, wurde im Nachhinein ein paar Tests in Ocaml geschrieben, welche im Codebeispiel 16 dargestellt werden. Diese Tests beschreiben einfache Funktionsaufrufe, wie zum Beispiel das Erhalten des ersten und zweiten Wertes eines Paares in Zeile 2 und 3. Anschließend werden dieselben Funktionen noch einmal aufgerufen - allerdings auf ein neues Paar (Zeile 8-10), dass durch 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;;
```

```
7
8  let pair2 = swap_pair pair;;
9  let resultfst2 = fst pair2;;
10 let resultsnd2 = snd pair2;;
11
12 Printf.printf "Result fst: %d \n%!" resultfst2;;
13 Printf.printf "Result snd: %d \n%!" resultsnd2;;
```

Codebeispiel 16: Ocaml Code testen (paperimpl.ml)

Ocaml-Code muss genauso wie Coq- und C-Code compiliert werden. Folgender Befehl in Codebeispiel 17 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.

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

Codebeispiel 17: Ocaml Code compilieren

In Codeblock 18 werden die Ausgaben der Tests dargestellt. Die ersten zwei Zeilen zeigen den x- und y- Wert eines Paares, dass durch **x: 3** und **y: 4** initialisiert wurde. Die Ausgaben in Zeile 4 und 5 stellen **fst** und **snd** des invertierten Paars dar.

```
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 18: Ocaml Code ausführen

5.2 Programm -> Proof

Die folgenden Unterkapitel erläutern den theoretischen und praktischen Weg, um Programmcode im Nachhinein mit Coq zu verifizieren. Dies wird anhand von der Programmiersprache C erläutert. Dabei wird die **Verified Software Toolchain (VST)**, auch Princeton Tool Chain genannt, verwendet, welche hauptsächlich von der Princeton University entwickelt wurde.

5.2.1 Beispiel Programmiersprache C und VST

Ein C-Programm kann nicht einfach in Coq kopiert und anschließend bewiesen werden. Dafür wird das Hilfsmittel **VST** benötigt. Diese ermöglicht es, ein C-Programm über **Verifiable C** in verifizierten Maschinencode zu übersetzen. In folgender Abbildung 2 sind die einzelnen Komponenten von der VST zu sehen.

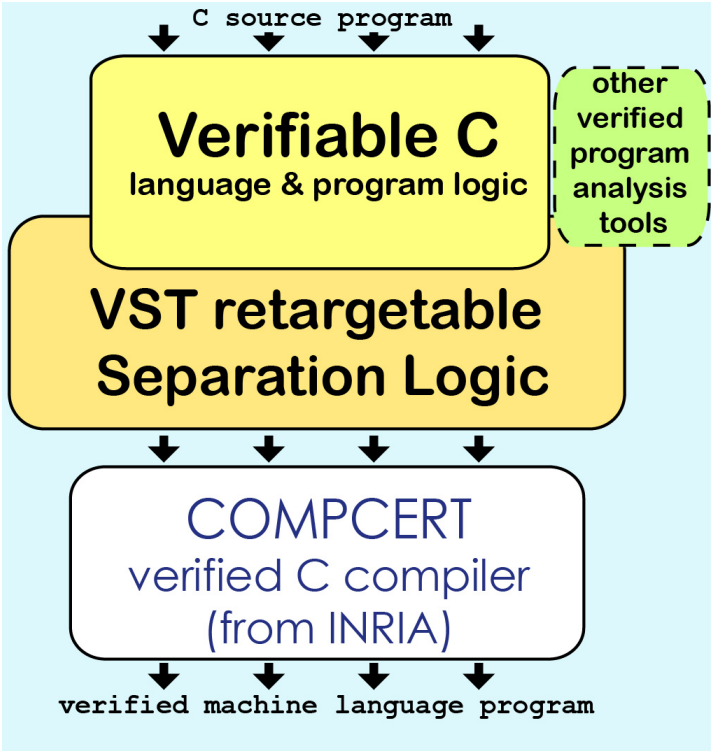

Abbildung 2: Verified Software Toolchain[PRI]

Zuerst wird dabei das C-Source-Programm durch die **Clight**-Technologie von **CompCert** 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.“(übersetzt aus [AWAwLB19a])
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 Komponente **VST retargetable Separation Logic** ergänzt die **program logic**. Außerdem können dadurch zusätzliche **verified program analysis tools** integriert werden.

Der nächste Schritt in der Princeton Tool Chain ist der **verified Compiler CompCert**. Dieser wurde von INRIA entwickelt und ist wie die VST open-source auf GitHub erhältlich unter `https://github.com/PrincetonUniversity/VST` und `https://github.com/PrincetonUniversity/VST/tree/master/compcert`. Um CompCert allerdings kommerziell nutzen zu können, wird ein kostenpflichtige Lizenz benötigt. CompCert wandelt schlussendlich den Verifiable C Code in Maschinensprache um. Der Compiler CompCert ist bereits komplett formal durch Coq verifiziert. Dies bedeutet, dass jeder in Verifiable C geschriebene Programmcode korrekt in den entsprechenden verifizierten Assemblercode übersetzt wird.

Es stellt sich die Frage, wie und wo die selbstgeschriebenen Funktionen auf mathematische Korrektheit geprüft werden.

5.2.2 Benutzung VST

Das Vorgehen zur Überprüfung der mathematischen Korrektheit von selbst geschriebenen C-Code ist im Paper „**Verifiable C** Applying the Verified Software Toolchain to C programs“ beschrieben.[AWAwLB19b]
Zuerst muss ein C-Programm beispielsweise in einer Datei **F.c** vorhanden sein. Anschließend muss der normale C-Code mit Hilfe von **Clightgen** von CompCert in Verifiable C übersetzt werden. Dies ist mit Hilfe des Command Line Interface (CLI) Kommandos **clightgen -normalize F.c** möglich. Dadurch entsteht eine Coq-Datei **F.v**. Um die enthaltenen Funktionen zu beweisen, muss eine Datei mit beispielsweise dem Namen **verif_F.v** erstellt werden. Wichtig dabei ist, dass in der Datei sowohl **F.v** als auch das VST Floyd Programm-Verifikationssystem **VST.floyd.proofauto** importiert werden (siehe Codebeispiel 13 für Import). Wenn ein Beweisen in der Datei **verif_F.v** aller Funktionen gelingt, ist das C-Programm korrekt geschrieben, kann kompiliert werden und ist somit maschinell formal verifiziert.

6 Aktuelle Anwendung

In diesem Kapitel werden einige Anwendungsfälle und Projekte kurz vorgestellt.

6.1 Proofed Stack

Mit Hilfe von VST inklusive CompCert wird versucht eine Art Stack aufzubauen, der von Maschinensprache über das Betriebssystem bis hin zu C-Programmcode formal verifiziert ist. Hierfür wird zusätzlich ein spezielles Coq-Framework namens **Kami** verwendet. Damit ist es möglich Hardware-Design formal zu beweisen. Diese Domain Specific Language (DSL) ist hauptsächlich inspiriert von **Bluespec**. Mit Hilfe von Kami ist es zum Beispiel möglich einen formal verifizierten Prozessor zu entwickeln. Die Firma SiFive arbeitet zur Zeit an einem Formal verifizierten RiscV Prozessor. **RISC-V** ist eine Befehlssatzarchitektur.[KAM] Vereinfacht gesagt, beschreibt dies die Verhaltensweise eines Prozessors in Form von einer formalen Spezifikation.
Die zur Architektur passenden Chips werden beispielsweise durch **SiFive**, welche auch **Kami** entwickelt haben, hergestellt.
Schlussendlich fehlt noch ein Betriebssystem, welches zum ein oder anderen Zeitpunkt mit Prozessoren und Speichern interagiert. Hierfür wurde **CertiKOS**, eines der ersten formal verifizierte Betriebssystem, entwickelt. Es unterstützt **x86** inklusive Multiprocessing. Außerdem ist es möglich, einen Hypervisor zu hosten und dadurch mehrere Betriebssysteme gleichzeitig auf der selben Maschine zu verwenden.[Wei][Hara]

Der ganze Proofed Stack könnte ungefähr in den Ebenen wie in der folgenden Illustration 3 dargestellt werden. Dabei sind Hardware-Komponenten in rot markiert. Der Softwareteil wird farblich unterschiedlich abgebildet mit dem Ziel, dass das Bild übersichtlicher ist.

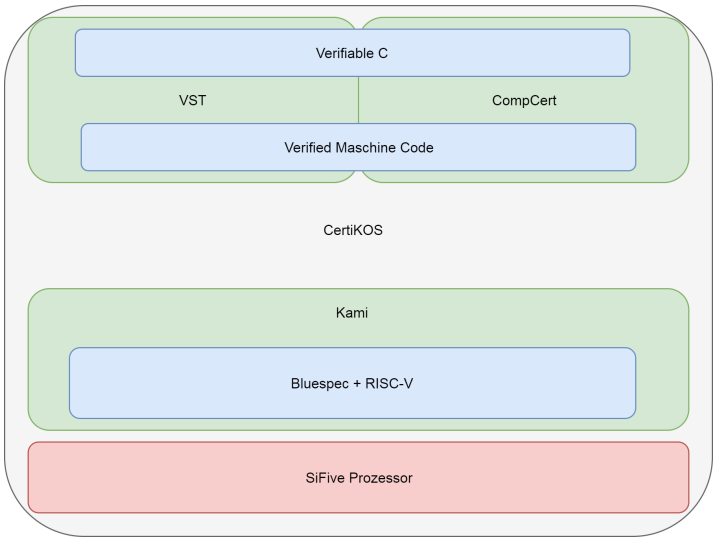

Abbildung 3: Proofed Stack

6.2 JSCert

JSCert ist eine maschinell in Coq geprüfte Spezifikation von JavaScript. Es wurde mit der Coq Version 8.4.6 geschrieben. Dabei wurde versucht, sich so nah wie möglich am englischen Standard ECMAScript 5 zu halten. Des Weiteren ist ein in Ocaml geschriebener Interpreter namens **JSRef** damit als korrekt bewiesen worden. Dies deckt 262 Testfälle ab.[JSCa][JSCb]

6.3 CertiCoq

CertiCoq ist ein in Coq verifizierter Compiler für Coq. Genau genommen beweist CertiCoq, dass die Kompilierung von dependent type Sprachen, wie Gallina korrekt verläuft.[AA][CER]

6.4 Beweise für Probleme der Mathematik oder Informatik

Mit Coq kann nicht nur Programmcode verifiziert werden. Die Grundlage dafür sind schließlich mathematische Beweise inklusive Taktiken und der interaktiven Entwicklungsumgebung. Somit wird Coq auch von Theoretikern, welche vor allem aus den Fachbereichen der Mathematik und Informatik kommen, genutzt. Die folgenden zwei bekannten Probleme wurden mittels Coq bewiesen.

6.4.1 Satz von Feit-Thompson

Das Odd-Order-Theorem, zu Deutsch Satz von Feit-Thompson sagt aus, dass jede endliche Gruppe von ungeraden Ordnungen auflösbar ist. Dies wurde bereits durch Walter Feit und John Griggs Thompson 1963 bewiesen.[GG13]
Nach einer sechsjährigen Arbeit gelang Georges Gonthier von INRIA 2012 die Verifikation in Coq. Dies wurde als Forschungs-Projekt zur Weiterentwicklung von Coq durchgeführt. Dabei sind über 150.000 Zeilen an Beweis-Code entstanden, welche circa 4.000 Definitionen und 13.000 Theoreme enthalten.[GG13]

6.4.2 Vier-Farben-Theorem

Das Vier-Farben Theorem sagt aus, dass es immer möglich ist, mit Hilfe von vier Farben eine beliebige Landkarte in der euklidischen Ebene einzufärben. Dabei dürfen angrenzende Länder niemals gleich gefärbt sein. Weiterhin werden isolierte gemeinsame Punkte nicht als Grenze gewertet. Eine weitere Einschränkung ist, dass keine Exklaven vorhanden sein dürfen. Dies bedeutet, dass die Karte aus einer zusammenhängenden Fläche bestehen muss (siehe Abbildung 4).[ACO]
Im Jahr 2005 wurde dieses Theorem erstmals durch die zwei Forscher Georges Gonthier und Benjamin Werner vollständig mit Coq bewiesen. [Gon]

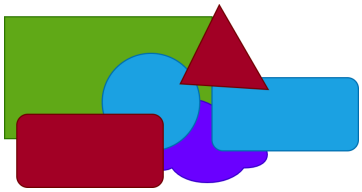

Abbildung 4: Beispiel für Vier-Farben-Problem

7 Aufwand in der Praxis

Der Aufwand für formale Verifikation ist schwer abzuschätzen, da diese Art des Programmierens nicht für jedes Teilgebiet ausreichend erforscht ist oder überhaupt benötigt wird. Grundsätzlich nutzen vor allem große Firmen im Bereich der low-level Software diese Form der Verifizierung. Damit kann beispielsweise sichergestellt werden, dass ein Prozessor immer korrekt angesteuert wird et cetera. Dabei werden meist funktionale Sprachen, wie C, oder maschinennahe Sprachen verwendet. Im Vergleich zu anderen Programmiersprachen wie Java oder C#, ist in diesen Bereichen die Forschung bereits sehr weit vorgedrungen.

In einem Vortrag über das Projekt Ironclad von Microsoft Research sprach Bryan Parno 2014 über den Einsatz von Formaler Verifikation bei einer Ende-zu-Ende Sicherheits-Anwendung. Dabei benannte er den Aufwand für das Programmieren mit etwa fünf Zeilen extra Beweis-Code pro Zeile Programmcode. Außerdem sei es eine komplett neue Art Software zu entwickeln. Abschließend fügte er noch hinzu, dass der meiste neue Code „out-of-the-box“ funktionierte und faktisch kaum Bugs enthielt.[CH]

Generell ist heutzutage vor allem in sicherheitskritischen Bereichen das Testen um einiges teurer als die eigentliche Entwicklung der Software. Durch formale Verifikation entstehen erstens weniger Bugs und zweiten senkt sich dadurch der Testaufwand enorm. Zusätzlich ist der Programmcode qualitativ hochwertiger als bei herkömmlichen Techniken, denn der Code wurde bereits zur Compilezeit und nicht erst zu Laufzeit auf Korrektheit geprüft.

8 Fazit

8.1 Zusammenfassung

Diese Arbeit stellt die Grundlagen der formalen Verifikation vor, welche auf Programmcode angewendet werden kann. Dabei wird zuerst auf die Relevanz eingegangen. Weitere Kapitel führen den Leser in maschinelles Beweisen mit Hilfe von Coq und der CoqIde ein. Das

Herzstück dieser Arbeit ist die Programmcodeverifikation, welche auf zwei Wegen durchführbar ist. Einerseits kann zuerst ein Beweis in einem Proof-Assistent für ein Theorem geschrieben und anschließend extrahiert werden. Andererseits ist es auch möglich, geschriebenen Programmcode im Nachhinein formal zu verifizieren. Beide Vorgehen werden in dieser Arbeit sowohl theoretisch als auch praktisch erläutert. Die Extraktion von bewiesenen Funktionen in Coq zu Ocaml-Code wird dabei anhand eines konkreten Codebeispiels detaillierter beschrieben. Abschließend werden konkrete Anwendungsfälle von formaler Verifikation mit Coq und eine Aufwandsschätzung für dessen praktische Anwendung präsentiert.

8.2 Schlussfolgerung

Programmcodeverifikation ist zum aktuellen Zeitpunkt ein weniger bekanntes Thema in der Informatik. Da es damit möglich ist 100% korrekte Software zu entwickeln, setzen momentan vor allem große Firmen, die sicherheitskritische oder viel verwendete Systeme bauen, auf diese Technologie. Der Aufwand für formal verifizierte Software ist dabei vergleichsweise deutlich höher. Entwickler von Microsoft Research schätzen den Aufwand auf circa 5 Zeilen Beweis-Code für jede Zeile Programmcode basierend auf ihrem durchgeführten Forschungsprojekt Ironclad. Außerdem nannten sie es eine neue Art für die Entwicklung der Software. Der Programmcode funktionierte meistens bereits beim ersten Mal und enthielt nur wenige Fehler.[CH] Dies kann durch das spezielle Typ-System von Coq, den dependent Types, erzielt werden. Dadurch sind Checks auf korrekten Code, welche normalerweise erst während der Laufzeit passieren, bereits bei der Kompilierung durchführbar.
Obwohl ein zu 100% funktionierender Programmcode in jedem Fachbereich ideal wäre, stehen Aufwand und Kosten nicht immer im passenden Verhältnis zum gewonnenen Nutzen. Des Weiteren ist zu erwähnen, dass nur ein Code nur so gut sein kann, wie sein Spezifikation. Mit Hilfe der formalen Verifikation kann zwar bewiesen werden, dass der Programmcode der Spezifikation entspricht, allerdings nicht, ob die Spezifikation selbst richtig ist.
Ein weiterer interessanter Punkt ist, dass alle inneren Spezifikationen heraus fallen. Solange beide Seiten einer Schnittstelle mit der Spezifikation leben können (das heißt gegen die Spezifikation verifiziert werden), ist es egal, was die Spezifikation ist. Das trifft zum Beispiel auf die Spezifikation von C in der Kombination VST + CompCert zu. Beide verwenden die selbe Spezifikation von C. Daher ist es egal, ob diese dem ANSI Standard entspricht. Die Verbindung abstrakte Logik zu Maschinencode ist trotzdem richtig.

Dadurch, dass eine Software immer auf Hardware läuft, kann zusätzlich keine wirkliche komplette Korrektheit eines Systems mit verifizierter Software garantiert werden. Hardware ist immer ab einem gewissen Zeitpunkt fehlbar und muss komplett ausgetauscht oder repariert werden.
Zusammenfassend wird Coq und formale Verifikation von Programmcode als sehr interessantes Thema mit viel Potential durch den Autor dieser Arbeit bewertet. Weiterführende Artikel, Vorträge und anderweitige Quellen sind im Kapitel A des Anhangs zu finden.

A Weiterführende Inhalte für Formale Verifikation

- DeepSpec: The science of deep specification https://deepspec.org/main
- 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/
- COMPCERT http://compcert.inria.fr
- CertiCoq: A verified compiler for Coq https://www.cs.princeton.edu/~appel/papers/certicoq-coqpl.pdf
- Software Foundations: Sehr ausführliches Coq-Tutorial https://softwarefoundations.cis.upenn.edu/current/index.html

Literatur

[AA] G. M. Z. P. R. P. O. S. B. M. S. M. W. Abhishek Anand, Andrew W. Appel. CertiCoq: A verified compiler for Coq. https://www.cs.princeton.edu/ appel/papers/certicoq-coqpl.pdf. Last visit: 26 Dez 2019.

[ACO] F. R. Amin Coja-Oghlan, Samuel Hetterich. Der Vier-Farben-Satz. `https://www.math.uni-frankfurt.de/~acoghlan/night_of_science_2013_kurz.pdf`. Last visit: 27 Dez 2019.

[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.

[CER] PrincetonUniversity/certicoq. https://github.com/PrincetonUniversity/certicoq. Last visit: 26 Dez 2019.

[CH] J. H. Chris Hawblitzel und B. P. D. Z. B. Z. Jacob R. Lorch, Arjun Narayan. Ironclad Apps: End-to-End Security via Automated Full-System Verification. https://www.usenix.org/node/186162. Last visit: 27 Dez 2019.

[COQa] How to get it? https://coq.inria.fr/. Last visit: 15 Dez 2019.

[COQb] Coq received ACM SIGPLAN Programming Languages Software 2013 award. https://coq.inria.fr/news/coq-received-acm-sigplan-programming-languages-software-2013-award.html. Last visit: 27 Dez 2019.

[GG13] J. A. Y. B. C. C. F. G. S. L. R. A. M. R. O. S. O. B. e. a. Georges Gonthier, Andrea Asperti. A Machine-Checked Proof of the Odd Order Theorem. S. 15, 2013.

[Gon] G. Gonthier. Formal Proof?The FourColor Theorem. http://www.ams.org/notices/200811/tx081101382p.pdf. Last visit: 27 Dez 2019.

[Hara] R. Harris. Unhackable OS? CertiKOS enables creation of secure system kernels. https://www.zdnet.com/article/certikos-a-hacker-proof-os/. Last visit: 26 Dez 2019.

[Harb] 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.

[JSCa] jscert/jscert. https://github.com/jscert/jscert. Last visit: 26 Dez 2019.

[JSCb] JSCert: Certified JavaScript. http://www.jscert.org. Last visit: 26 Dez 2019.

[KAM] Kami. http://plv.csail.mit.edu/kami/. Last visit: 26 Dez 2019.

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

[PM] C. PaulinMohring. Introduction to the Calculus of Inductive Constructions.

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

[Wei] W. Weir. CertiKOS: A breakthrough toward hacker-resistant operating systems. https://news.yale.edu/2016/11/14/certikos-breakthrough-toward-hacker-resistant-operating-systems. 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.