Nutzen von Dependent Types am Beispiel der Programmiersprache Gallina

Simon Schneemelcher
s.schneemelcher@tu-braunschweig.de
Technische Universität Braunschweig
Institut für Softwaretechnik und Fahrzeuginformatik
Braunschweig, Niedersachsen

ABSTRACT

Diese Arbeit beschäftigt sich mit Dependent Types und der Programmiersprache Gallina des Beweistools Coq. Dependent Types sind eine Möglichkeit das Typsystem einer Programmiersprache so zu erweitern, dass Eigenschaften der darin geschriebenen Programme formal beweisbar werden. Diese Eigenschaft ist zum einen wichtig um mathematische Beweise mit Software zu prüfen, aber sie ist zum anderen auch von großer Bedeutung für die Softwareentwicklung, da formal bewiesene Programme eine höhere Sicherheit gegen Fehler liefern, die vom Programmierer gemacht werden können. Im besten Fall können diese damit komplett ausgeschlossen werden. In dieser Arbeit wird zuerst kurz auf die Geschichte hinter Dependent Types eingegangen, danach werden die Einsatzmöglichkeiten und eine mögliche Implementierung in Form der Programmiersprache Gallina vorgestellt. Zum Schluss gibt es noch eine Diskussion über Vor- und Nachteile und ein Ausblick auf die zukünftige Entwicklung.

KEYWORDS

Dependent Types, Type Theory, Gallina, Coq, Calculus of Constructions

1 EINFÜHRUNG

Computer wurden erfunden, um mathematische Berechnungen durchzuführen, für welche ein Mensch von Hand erheblich länger brauchen würde oder welche sogar ohne ihn praktisch unmöglich wären. Computer sind sehr gut darin, die Befehle, welche ein Programmierer ihnen gibt, schnell und beliebig oft hintereinander auszuführen. Jedoch haben die Maschinen kein Verständnis dafür, was sie eigentlich tun. Man könnte daher denken, dass sie mathematische Beweise nicht verstehen und durchführen können. Durch logische Ansätze allerdings, ist es möglich, Computern dies beizubringen.

Wenn Mathematiker Zusammenhänge beweisen wollen, müssen diese oft seitenlange formale Beweise schreiben. Diese werden dann von ihren Kollegen begutachtet und falls niemand einen Fehler entdeckt, der Allgemeinheit zur Verfügung gestellt. Da Mathematiker jedoch auch nur Menschen sind und Fehler machen, kann es vorkommen, dass sich in Beweise Fehler einschleichen und diese ungültig machen. Computergestützte Beweistools wollen sich diesem Problem annehmen und die Wahrscheinlichkeit Fehler zu übersehen minimieren. Dabei arbeiten sie mit verschiedenen Methoden. Mit einer davon, Dependent Types, wird sich diese Arbeit näher beschäftigen.

Aber nicht nur in der Mathematik besteht Interesse daran, Programme beweisbar zu machen, auch Softwareentwickler können Dependent Types nutzen, um ihre Programme sicherer zu machen und die Möglichkeit Fehler zu machen auf ein Minimum zu beschränken. In der heutigen Welt ist dies wichtiger denn je, da Softwaresysteme die Grundlage der digitalen Kommunikation und das Geschäftsmodell vieler Firmen bilden, täglich von Milliarden Menschen benutzt werden und ihr Ausfall erhebliche wirtschaftliche Konsequenzen hat[1]. Außerdem wird ein großer Anteil der Produktionskosten und Zeit für das Testen einer neuen Software verwendet [1] und selbst dann ist nicht gewährleistet, dass nicht noch Fehler vorhanden sind, die in Situationen auftreten, an welche man beim Testen noch gar nicht gedacht hatte.

Das Konzept der in dieser Arbeit vorgestellten Dependent Types, nutzt die Parallelen zwischen Computerprogrammen und mathematischen Beweisen, beziehungsweise logischen Aussagen, um Eigenschaften des geschriebenen Programmes schon zur Compilezeit zu verifizieren. Sie wurden seit ihrer Entstehung in vielen Programmiersprachen implementiert, dazu gehören zum Beispiel Agda[2], Idris[3], Gallina[4], aber auch in der vergleichsweise bekannten[5] funktionalen Programmiersprache Haskell[6] und in der 2018 am meisten verwendeten Sprache auf Github[7] JavaScript [32], werden momentan Erweiterungen für Dependent Types entwickelt[8].

In dieser Arbeit wird zunächst kurz auf die Geschichte hinter den Dependent Types eingegangen, danach folgt eine Erklärung des mathematisch-logischen Gedankens dahinter, dann wird anhand der Programmiersprache Gallina ein praktisches Beispiel für eine mögliche Implementierung in Software vorgestellt und schließlich folgt eine kritische Bewertung des Nutzens und der weiteren zukünftigen Entwicklung der Dependent Types.

2 GESCHICHTE

Anfang der 20er Jahre des 20. Jahrhunderts veröffentlichte der deutsche Mathematiker David Hilbert, inspiriert von der „Principia Mathematica“ von Russel und Whitehead, das sogenannte Hilbertprogramm [2]. Dies war eine Axiomatisierung der formalen Logik,

[1] https://www.tricentis.com/resources/software-fail-watch-5th-edition/ letzter Zugriff 19.01.20
[2] https://github.com/agda/agda letzter Zugriff 21.01.20
[3] https://www.idris-lang.org/ letzter Zugriff 21.01.20
[4] https://coq.inria.fr/ letzter Zugriff 21.01.20
[5] https://www.tiobe.com/tiobe-index/ letzter Zugriff 17.01.20
[6] https://www.haskell.org/ letzter Zugriff 21.01.20
[7] https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/ letzter Zugriff 22.01.20
[8] https://gitlab.haskell.org/ghc/ghc/wikis/dependent-haskell letzter Zugriff 21.01.20

welche eine Lösung der Grundlagenkrise der Mathematik [3] darstellen sollte. Das Programm hatte mehrere Ziele. Zum einen sollte es zeigen, dass jede Aussage entscheidbar, also entweder wahr oder falsch sei. Dies wurde jedoch vom Österreichischen Logiker Kurt Gödel widerlegt, welcher zeigte, dass es Aussagen gibt die weder wahr noch falsch seien können [4]. Eine Definition für Berechenbarkeit wurde zum einen von Alonzo Church mit dem Lambda Kalkül [5], zum anderen von Gödel mit seinen rekursiven Funktionen [6] und von Alan Turing mit der Idee der Turingmaschinen [7] gefunden.

Im Lambda Kalkül gibt es nichts als Funktionsdefinitionen, Variablen und deren Anwendung. Er beschreibt also eine funktionale Definition der Berechenbarkeit[9]. Rekursive Funktionen sind ebenfalls eine funktionale Definition der Berechenbarkeit[10]. Turing Maschinen beschreiben eine Zustandsbasierte Definition der Berechenbarkeit indem es in ihnen immer jeweils einen Zustand und eine Eingabe in Form eines beschriebenen Bandes gibt[11].

Diese drei Ideen stellten sich jedoch als gleich mächtig heraus [8]. Church erkannte, dass sich mit seinem Lambda Kalkül keine logischen Formeln beschreiben ließen, da er inkonsistent war und wandelte von ihm noch einen anderen Kalkül, den sogenannten einfach getypten Lambda Kalkül ab, mit dem man logische Aussagen formulieren konnte. Dessen Aussagen hatten aber immer eine Normalform, was einem Halten der entsprechenden Turingmaschine entsprechen würde [8].

Das Hilbertprogramm sollte die Mathematik außerdem konsistent machen, es sollte also keine Widersprüche enthalten. Daran arbeitete Gerhard Gentzen, der zwei Methoden veröffentlichte, um die Konsistenz formaler Logiken zu zeigen. Außerdem ergänzte er zu dem bereits bekannten Existenzquantor ($\exists$) noch den Allquantor ($\forall$). Der Existenzquantor $\exists x$ sagt aus, dass es mindestens ein x gibt für welches eine Aussage $A(x)$ gilt. Der Allquantor $\forall x$ sagt aus, dass eine Aussage $A(x)$ für jedes beliebige x gilt. Auf Grundlage dieser Entwicklungen machte Haskell Curry die Beobachtung, dass Typen von Funktionen den Aussagen in der Logik entsprechen und die Beweise der Aussagen, Programmen mit dem Typ der Aussage entsprechen [9][10][8]. Der Logiker William Howard zeigte darauf aufbauend noch weitere Korrespondenzen und führte dabei auch Dependent Types als Typen ein, die den $\exists$ und $\forall$ Quantoren der Logik entsprechen [8]. Diese Korrespondenz zwischen Beweisen und Programmen nennt man „Curry-Howard Isomorphismus“ oder auch „Propositions as Types“.

Auf dieser Idee aufbauend, veröffentlichte Per Martin-Löf, ein schwedischer Mathematiker und Philosoph seine „Intuitionistic Type Theory“ [11], welche Dependent Types verwendet. Der Logiker Jean-Yves Girard zeigte jedoch, dass Martin-Löfs System inkonsistent war, sich also alle beliebigen Aussagen darin beweisen lassen, da sie Typen enthält, deren Typ sich selbst referenziert [12][13][12]. 1986 veröffentlichten Thierry Coquand und Gérard Huet ihren „Calculus of Constructions“ [14]. Auf einer Variante diesem Kalküls, dem Calculus of inductive Constructions, welcher zusätzlich induktive Typen einführt, basiert die Sprache Gallina. Mit dem Kalkül und der Sprache wird sich in späteren Kapiteln noch genauer beschäftigt.

3 EINFÜHRUNG IN DAS KONZEPT DER DEPENDENT TYPES

In diesem Kapitel werden kurz die mathematisch-logischen Hintergründe der Dependent Types erläutert und worum es sich dabei handelt. Dafür wird erst allgemein auf Typisierung eingegangen und dann die Dependent Types erklärt, welche einen Teil des Typsystems darstellen.

3.1 Typisierung

Typisierung von Programmiersprachen dient dazu, dass unerlaubte Operationen schon von vornherein beim compilieren erkannt und verhindert werden können. Dabei wird der Wertebereich eines Objekts eines bestimmten Typs eingeschränkt. Damit können dann die zulässigen Argumente von Funktionen eingeschränkt werden. Ein Beispiel hierfür wäre die Addition zwischen einem String und einem Integer, falls eine solche Addition nicht in der Sprache definiert ist. Der Compiler kann in dem Fall erkennen, dass die Operation unzulässig ist und wird den Programmcode nicht compilieren. Dazu verwendet der Compiler einer Programmiersprache Typprüfung, welche jedoch oft nicht alle Fehler erkennen kann, wenn das Typsystem der Sprache nicht Ausdrucksstark genug ist [15]. Die meisten Programmiersprachen nutzen direkt in die Sprache integrierte Typen wie Zeichenketten (Strings), Ganzzahlwerte (Integer) und Wahrheitswerte (Boolean). Es gibt allerdings auch ein Paar Programmiersprachen ohne Typen oder mit sehr eingeschränkten Typsysteme wie Brainfuck[13] oder B[14]. Datentypen existieren jedoch nur als Abstraktion in der Programmiersprache. Die meisten Prozessorarchitekturen kennen die Typen der Daten mit welchen sie arbeiten nicht explizit, da sie nur die einzelnen Bytes sehen und ihre Operationen ausführen. Der Compiler muss dann die passende Prozessoroperation für die Datentypen auswählen [16]. Eine Alternative dazu ist die sogenannte „Tagged Architecture“, bei welcher zusätzlich zu den reinen Daten noch ein Datentyp als „Tag“ direkt im Speicher hinterlegt ist [16]. Diese Architektur setzte sich jedoch nicht durch.

[9] https://www.youtube.com/watch?v=eis11j_iGMs&t letzter Zugriff 21.01.20
[10] https://plato.stanford.edu/entries/recursive-functions/ letzter Zugriff 21.01.20
[11] https://www.youtube.com/watch?v=dNRDvLACg5Q letzter Zugriff 21.01.20
[12] Dies nennt man seitdem Girard's Paradoxon. Es ist verwandt mit Russell's Paradoxon (https://math.berkeley.edu/~kpmann/Russell.pdf letzter Zugriff 21.01.20) in der Mengenlehre
[13] https://www.muppetlabs.com/~breadbox/bf/ letzter Zugriff 21.01.20
[14] https://www.bell-labs.com/usr/dmr/www/bintro.html letzter Zugriff 21.01.20

3.2 Möglichkeiten der Abhängigkeiten zwischen Termen und Typen

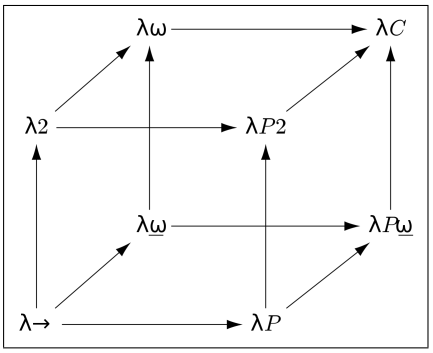

Abbildung 1: Lambda Würfel[15]

In **Abbildung 1** ist der Lambda Würfel zu sehen, ein Konzept des Niederländischen Logikers Henk Barendregt [17]. Jede Dimension des Würfels beschreibt dabei eine Abhängigkeit zwischen Termen und Typen und an jeder Ecke steht ein Kalkül, welcher auf dem einfach getypten Lambda Kalkül basiert, diesem jedoch abhängig von seiner Position im Würfel Abhängigkeiten hinzufügt.

An der vorderen, linken, unteren Ecke befindet sich der einfach getypte Lambda Kalkül $\lambda\!\rightarrow$. Bei diesem können Terme nur von Termen abhängen [17]. Dies entspricht den im Lambda Kalkül üblichen Funktionen.

Geht man im Würfel auf der y-Achse nach oben, fügt man die Abhängigkeit von Termen auf Typen hinzu [17]. Dies wird auch Polymorphismus genannt. Ein Beispiel hierfür wäre die Definition eines Operators wie „+“ für verschiedene Typen mit jeweils verschiedener Funktion. Beispielsweise in der Sprache Java wird der Operator zwei Integer addieren, zwei Strings jedoch konkatenieren.

Geht man den Würfel auf der z-Achse nach hinten, fügt man die Abhängigkeit von Typen auf Typen hinzu [17]. Dies sind sogenannte Typ Operatoren, welche einen Typ nehmen und daraus einen neuen Typ machen. Ein Beispiel dafür wäre der Konstruktor einer Liste eines bestimmten Typs.

Zuletzt bleibt noch die x-Achse. Geht man diese nach rechts, fügt man dem System die Abhängigkeit von Typen auf Terme hinzu [17]. Dies sind die Dependent Types. Da sich in dieser Arbeit mit den anderen Kalkülen nicht beschäftigen wird, werden diese hier nicht benannt, an der hinteren, rechten, oberen Ecke jedoch steht ein Kalkül, welcher jede dieser Eigenschaften miteinander vereint. Mit λC ist der Calculus of Constructions gemeint, welcher bereits erwähnt wurde [14].

3.3 Calculus of Constructions

Der Calculus of Construction wurde 1986 von Coquand und Huet eingeführt [14]. Er basiert auf dem Lambda Kalkül und ist ein Formalismus höherer Ordnung, das bedeutet, dass in ihm Funktionen als Argumente und Rückgabewerte anderer Funktionen verwendet werden können [14]. Mit ihm lassen sich konstruktive mathematische Beweise in der Form des natürlichen Schließens erstellen, daher der Name „Konstruktionskalkül“ [14]. Der Kalkül beinhaltet alle Abhängigkeiten zwischen Typen und Termen und kann als funktionale Programmiersprache angesehen werden. Außerdem besitzt der Kalkül die Eigenschaft stark normalisierend zu sein, das bedeutet, dass jede Berechnung eine Normalform hat und nicht endlos werden kann [14]. Damit ist er nicht Turing-vollständig.

3.4 Dependent Types

Einfach gesagt, sind Dependent Types Typen, die von konkreten Werten abhängen. Während bei herkömmlichen Typsystemen der Bereich der Werte und der Typen klar getrennt ist, verschwimmen diese hier. Dadurch wird es zum Beispiel möglich Funktionen zu schreiben, deren Rückgabetyp vom Wert der Eingabe abhängt. Das gibt dem Compiler ein größeres Wissen darüber, wie das Programm funktionieren soll.

Ein einfaches Beispiel dafür wäre zum Beispiel der Typ der Vektoren, beziehungsweise Listen der Länge n, A^{n} [18]. Dieser hängt bei einer konkreten Liste von ihrer Länge n ab, also einem konkreten Wert. Ein Vektor der Länge 2 hätte also beispielsweise den Typ A^2. Alle Instanzen die nun diesen Typ erhalten, haben nachweislich genau zwei Werte. Damit ist für den Compiler ersichtlich, was er bei Instanzen dieses Typs zu erwarten hat. Damit sind dann Fehler, die durch das zugreifen auf einen Index der nicht vorhanden ist ausgelöst werden, zur Laufzeit nicht möglich da der Compiler diese schon zur Compilezeit erkennen kann. Würde man in unserem Beispiel im Programmcode auf den dritten Eintrag eines Vektors v vom Typ A^2 zugreifen, wüsste bereits der Compiler, dass es diesen Eintrag nicht geben kann und bricht die Compilation ab.

In der Mathematik verwendet man heutzutage die Mengenlehre als Grundlage. Alles ist in der Mathematik eine Menge und kann über bestimmte Axiome definiert werden. Es gibt in der Mengenlehre einen Operator, über den alle Anderen Operationen definiert werden können, nämlich $\in$ „ist Element von“ [19]. Allerdings ist die Mengenlehre nicht die einzig mögliche Grundlagentheorie der Mathematik. Eine weitere mögliche mathematische Grundlage wären die Typentheorien. In diesen hat jeder Ausdruck einen Typ. Auf diesen Typen sind dann Operationen definiert. Typentheorien, wie der Calculus of Inductive Constructions eignen sich für formale Beweissysteme besser als die Mengenlehre, da Programmiersprachen üblicherweise auch ein Typsystem besitzen und dies typentheoretische Eigenschaften hat.

Bei einer typentheoretischen Auffassung der Mathematik können Instanzen der Typen als Behauptungen betrachtet werden und die Typen selbst als Beweise für die Aussagen. Der Typ A^{n} enthält also ein Programm, welches beweist, dass ein gegebener Vektor eine Länge von genau n hat. Dadurch, dass jeder Typ auch ein Beweis ist, erhält man die Sicherheit, dass die Instanzen des Typen korrekt sind.

Es gibt zwei grundlegende Arten von dependent Types, zum einen Dependent Function Types und zum Anderen Dependent Pair Types. Dependent Function Types, auch Π Types genannt, sind Funktionen, dessen Rückgabewert vom Argumentwert abhängt. $\Pi_{\mathrm{x:A}}B(x)$ ist dann beispielsweise die Funktion, die jedem x vom Typ A ein Element vom Typ $B(x)$ zuweist [20]. Diese Typen entsprechen den All-Quantoren der Prädikatenlogik. Dependent Pair Types, auch Σ Types genannt, sind Tupel, dessen zweiter Wert vom ersten Wert abhängt. Beispielsweise wäre der Typ $\Sigma_{\mathrm{x:A}}B(x)$ die Menge der

[15] By Tellofou - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=76344034 letzter Zugriff 22.01.20

Tupel (a, b) mit a vom Typ A und b vom Typ $B(a)$ [20]. Diese Typen entsprechen den Existenz-Quantoren der Prädikatenlogik.

4 GALLINA

In diesem Kapitel wird gezeigt, wie eine mögliche Implementierung von Dependent Types in einer Programmiersprache aussehen kann. Es werden ein paar allgemeine Informationen über die Sprache Gallina und anhand von ein paar Beispielen ein kurzer Einblick in die Syntax gegeben. Des Weiteren wird gezeigt, wie man einen mathematischen Beweis in dieser Sprache führt, da dies ihre Hauptanwendung ist.

4.1 Allgemein

Gallina ist die Sprache des Beweistools Coq. Coq wurde vom Projekt TypiCal des Inria[16] , dem „Institut national de recherche en informatique et en automatique“ in Frankreich entwickelt und wird noch aktiv weiterentwickelt. Gallina basiert auf dem „Calculus of Inductive Constructions“, welches auf dem Calculus of Constructions von Coquand und Huet [14] basiert.

Dieser Kalkül hat einige besondere Eigenschaften, welche ihn interessant für die Verwendung in einem Beweistool machen. Zum einem ist er sehr ausdrucksstark, da er alle Möglichen Abhängigkeiten von Termen und Typen ermöglicht, so also auch Dependent Types. Durch „Propositions as Types“ ist bekannt, dass Dependent Types den Quantoren der Logik entsprechen. Daher kann man diese dann in Beweisen als solche verwenden. Zum anderen ermöglicht er auch Inductive Types, also Typen, welche man induktiv aus anderen Typen aufbauen kann. Damit lassen sich Datenstrukturen wie Listen oder auch die natürlichen Zahlen definieren [20].

Des Weiteren ist in dem Kalkül Typechecking entscheidbar obwohl er Dependent Types unterstützt. Normalerweise verursachen Dependent Types die Unentscheidbarkeit des Typecheckings [21]. Der Calculus of Inductive Constructions verhindert dies, indem alle Aussagen in ihm eine Normalform haben. Übertragen auf Gallina bedeutet das, dass alle Programme halten. Diese Eigenschaft nennt man stark normalisierend. Damit ist die Sprache nicht Turingvollständig, es lassen sich also nicht alle theoretisch möglichen Berechnungen in ihr durchführen. Das hat aber den Vorteil, dass sich die Programme tatsächlich beweisen lassen, denn der Typechecker kann damit die Typen, welche logischen Aussagen entsprechen, auf ihre Korrektheit überprüfen. Daher ist Coq gut für computergestützte mathematische Beweise geeignet. Ein weiterer Verwendungszweck von Coq ist das extrahieren von Programmen aus Gallina Code oder Beweisen in andere Sprachen wie Haskell, OCaml und Scheme[17]. Die Spezifikationen solcher Programme sind dann formal bewiesen, was eine hohe Sicherheit der Korrektheit der Programme mit sich bringt. Jedoch können bei der Extraktion Probleme entstehen, sodass eine fehlerfreie Ausführung nicht immer gewährleistet werden kann. Aber auch dafür gibt es Lösungsansätze [22][23].

Gallina implementiert ein funktionales Paradigma, da der unterliegende Kalkül auf dem ebenfalls funktionalen Lambda Kalkül basiert. Sie besteht zum einen aus der Sprache der Terme[18] und zum anderen aus einer Sprache von Befehlen die „Vernacular“[19] genannt wird. Diese Befehle beinhalten unter anderem sogenannte Taktiken, die dabei helfen, Beweisziele zu erreichen.

Für Gallina ist es entscheidend, dass die Programme terminieren, da man sonst aus einem Programm, welches einem Beweis entspricht, alles folgern könnte. Da aber nicht entscheidbar ist, ob ein Programm hält oder nicht, kann der Gallina Compiler dies nicht immer erkennen. Wenn Zweifel darüber bestehen, ob ein Programm hält, geht der Compiler standardmäßig davon aus, dass das Programm nicht hält. In solchen Fällen muss man darauf zurückgreifen, einen Beweis für das Halten des Programms zu schreiben, welcher dann vom Compiler überprüft werden kann. Die Tatsache, dass Gallina keine Endlosschleifen zulässt, bedeutet, dass sie nicht Turing-vollständig ist. Dementsprechend ist es nicht möglich, einen Selbstinterpreter für Gallina zu schreiben, also einen Interpreter für Gallina, geschrieben in Gallina selbst.

4.2 Typsystem

Gallina besitzt ein statisches Typsystem, das bedeutet, der Compiler weiß welche Variablen welchen Typ haben. Gallinas Typen sind Induktiv, das heißt, man kann neue Typen aus anderen Typen aufbauen. Der unterste Typ in Gallinas Typenhierarchie ist der Typ `Type`[20]. Um Girard's Paradoxon zu umgehen, welches auftreten würde wenn der Typ `Type` sich selbst als Typ hätte, hat `Type` einen Index und `Type(n)` hat den Typ `Type(n+1)`. Gallina hat also eine unendliche Typenhierarchie. Alle weiteren Typen bauen auf `Type` auf und können vom Programmierer selbst definiert werden, sofern man nicht die `Standard Bibliothek`[21] von Gallina für gebräuchliche Typen nutzen möchte.

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

Code-Beispiel 1: Beispiel für die Definition der natürlichen Zahlen nach Peano

Code-Beispiel 1 zeigt die mögliche Implementierung der natürlichen Zahlen nach den Peano Axiomen als Inductive Type. Dabei steht O für die 0, welche vom Typ `nat` ist. S steht für den Nachfolger einer natürlichen Zahl und hat den Typ `nat`, angewandt auf ein anderes Objekt vom Typ `nat`. So wäre zum Beispiel die eins S O, die zwei S (S O), die drei S (S (S O)) und so weiter. Jede natürliche Zahl n ist also der n-te Nachfolger von 0 oder 0 selbst. Auf ähnliche Weise lassen sich auch Operationen auf Typen definieren, wie beispielsweise die Addition der natürlichen Zahlen.

```
Fixpoint plus (n m:nat) : nat :=
  match n with
  | O => m
  | S p => S (p + m)
  end
```

Code-Beispiel 2: Definition der Addition von natürlichen Zahlen in Coq aus der Standard Library[22]

[16] https://www.inria.fr/en letzter Zugriff 28.01.20
[17] https://coq.inria.fr/refman/addendum/extraction.html letzter Zugriff 22.01.20
[18] https://coq.inria.fr/refman/language/gallina-specification-language.html#terms letzter Zugriff 22.01.20
[19] https://coq.inria.fr/refman/language/gallina-specification-language.html#the-vernacular letzter Zugriff 22.01.20
[20] http://adam.chlipala.net/cpdt/html/Universes.html letzter Zugriff 22.01.20
[21] https://coq.inria.fr/refman/language/coq-library.html letzter Zugriff 22.01.20

Hier muss der Befehl „Fixpoint ... match“ benutzt werden, welchen man verwendet, um rekursive Funktionen zu definieren. Rekursionen in Gallina müssen immer wohlfundiert sein, das bedeutet, dass die Funktionen sich nicht endlos oft selbst aufrufen dürfen [24]. Man kann im Beispiel sehen, dass die Rekursion immer halten muss, da man das Argument n in der vierten Zeile immer echt kleiner macht und wenn es bei 0 ankommt, die dritte Zeile ausgeführt wird. Bei einer Berechnung von n + m ruft sich die Funktion so lang selbst auf indem sie n mit S p => S (p+m) matcht und macht dabei aus

„plus $\overbrace{S(S..(S\ \mathrm{O})}^{\mathbf{n}mal}$ $\overbrace{S(S..(S\ \mathrm{O})}^{\mathbf{m}mal}$“, „S(plus $\overbrace{S(S..(S\ \mathrm{O})}^{\mathbf{n\text{-}1}mal}$ $\overbrace{S(S..(S\ \mathrm{O})}^{\mathbf{m}mal}$“, bis sie schliesslich bei „$\overbrace{S(S..(}^{\mathbf{n}mal}$ plus O) $\overbrace{S(S..(S\ \mathrm{O})}^{\mathbf{m}mal}$“ ankommt. Dann matcht sie n mit O => m und macht daraus „ $\overbrace{S(S..(S\ \mathrm{O})}^{\mathbf{n+m}mal}$“. Dies ist nun wieder eine natürliche Zahl, nämlich der (n+m)-te Nachfolger der 0, also n+m. Damit tut die Funktion genau das, was gefordert war.

4.3 Beispiel der Dependent Types in Gallina

Dieses Kapitel beschäftigt sich mit der konkreten Implementierung von Dependent Types in Gallina. Zur Veranschaulichung dient eine beispielhafte Implementierung eines Listentyps welcher von der Länge der Liste abhängt.

```
Section ilist.
  Variable A : Set.

  Inductive ilist : nat -> Set :=
  | Nil : ilist O
  | Cons : forall n, A -> ilist n -> ilist (S n).
```

Code-Beispiel 3: Definition eines Dependent List Type in Gallina, Beispiel von Adam Chlipala[23]

In **Code-Beispiel 3** ist ein Beispiel für einen induktiven Dependent Type zu sehen. Es wird eine Liste mit dem Datentyp `ilist n`, was für eine Liste der Länge n steht, definiert. Dabei ist `ilist` vom Typ `nat -> Set`, also ein Dependent Type. Im Konstruktor wird mit dem `forall n` auch ein Dependent Type verwendet. Der Unterschied einer solchen Liste zu beispielsweise einer mit fester Größe in Java besteht darin, dass die Größe nicht zur Compilezeit bekannt sein muss, sondern während der Laufzeit festgelegt werden kann[24]. Außerdem können damit nun keine Fehler wie der Zugriff auf ein Element der Liste mit einem index welcher nicht vorhanden ist während der Laufzeit entstehen, da der Typ in Funktionen verwendet und damit eingeschränkt werden kann, was für Listen mit welcher Länge erwartet werden.

4.4 Ablauf von Beweisverfahren

In diesem Kapitel wird veranschaulicht, wie ein Beweis mit Coq in Gallina geführt wird. Dies ist deshalb interessant, da es die Hauptaufgabe von Coq ist und Dependent Types dabei zum Einsatz kommen.

[22] https://coq.inria.fr/stdlib/Coq.Init.Nat.html letzter Zugriff 22.01.20
[23] http://adam.chlipala.net/cpdt/html/MoreDep.html letzter Zugriff 22.01.20
[24] http://adam.chlipala.net/cpdt/html/MoreDep.html letzter Zugriff 22.01.20

```
Theorem my_first_proof : (forall A : Prop, A -> A).

Proof.
  intros A.
  intros proof_of_A.
  exact proof_of_A.
Qed.
```

Code-Beispiel 4: Einfacher Coq Beweis aus dem Coq Tutorial[25]

In **Code-Beispiel 4** ist ein kurzer Beweis für die reflexivität der Implikation zu sehen. In der ersten Zeile wird die Behauptung, die bewiesen werden soll, mittels dem Vernacular Befehl „Theorem“ aufgestellt. Die aufgestellte Behauptung besagt, dass für alle *A* vom Typ `Prop`, welcher der Typ der Propositionen, also der Aussagen ist, gilt, dass sie aus sich selbst folgen, also reflexiv sind. Der Befehl „Proof“ hat keine Bedeutung für den Compiler sondern dient nur dem Programmierer zur Übersichtlichkeit. Man verwendet ihn, wenn man einen Beweis beginnt.

Die deklarierte Behauptung ist ein Beweisziel, welches in Gallina „Goal“ genannt wird. Um dieses zu erreichen, müssen alle Zwischenziele erreicht werden, welche sich „Subgoals“ nennen [25]. Wird der interaktive Modus von Coq[26] verwendet, bekommt man folgende Ausgabe.

```
1 subgoal

  ============================
  forall A : Prop, A -> A
```

Code-Beispiel 5: Ausgabe von Coqtop nach Ausführung der ersten Zeile

Nun werden die eingangs erwähnten Taktiken verwendet, um das Beweisziel zu erreichen. Im Term „intros A.“ wird die Taktik „intros“ verwendet. Diese wird genutzt, um Annahmen einzuführen, also in diesem Fall um die Annahme zu machen, es gäbe ein *A* vom Typ `Prop`. So wird das „forall A“ aus der Behauptung beseitigt, da dies auch den Typ `Prop` hat. Damit wird das nächste Subgoal erreicht und es bleibt noch `A -> A` zu beweisen.

```
1 subgoal

  A : Prop
  ============================
  A -> A
```

Code-Beispiel 6: Ausgabe von Coqtop nach intros A

Das „intros proof_of_A.“ führt nun eine neue Variable mit dem Typ *A* ein. Somit bleibt nur noch „A“ in der Behauptung stehen. Da die Annahmen aber die Variable „proof_of_A“ vom Typ `A` enthalten, kann der Befehl „exact“ verwendet werden um *A* zu beweisen.

```
1 subgoal

  A : Prop
  proof_of_A : A
  ============================
  A
```

Code-Beispiel 7: Ausgabe von Coqtop nach intros proof_of_A

[25] https://coq.inria.fr/tutorial-nahas letzter Zugriff 22.01.20
[26] https://www.mankier.com/1/coqtop letzter Zugriff 22.01.20

Da nun alles bewiesen wurde, gibt Coq die Mitteilung „No more subgoals.“ aus. Nun kann mit dem Befehl „Qed.“ der Beweis beendet werden. Nun wurde also ein Beweis für die Reflexivität definiert, welcher selbst ein Programm ist und theoretisch in einem größeren Programm verwendet werden könnte.

5 DISKUSSION

In diesem Kapitel wird ein Überblick über die Vorteile von Dependent Types gegeben, sowie über mögliche Nachteile die entstehen können, wenn man Dependent Types in einer Programmiersprache nutzt. Außerdem werden zwei Anwendungen vorgestellt, welche Coq verwenden und bei denen Dependent Types eine Rolle spielen oder gespielt haben.

5.1 Vorteile und praktische Anwendungen von Dependent Types

Der größte Vorteil von Dependent Types liegt darin, dass man beweisen kann, dass das Programm das tut worauf es spezifiziert ist [26]. Zwar besteht die Möglichkeit, dass Programmfehler im Compiler dafür sorgen, dass Fehler beim compilieren gemacht werden die dann Fehlverhalten des Programmes zufolge haben, jedoch ist dies im Verhältnis zu den Fehlern, die Programmierer beim Schreiben neuer Software machen, meist zu vernachlässigen.

In Bereichen, wo eine Fehlfunktion oder sogar ein Ausfall eines Programms gravierende Folgen nach sich ziehen kann ist dies anders. Dort versucht man natürlich wirklich jeden möglichen Fehler zu vermeiden, da diese unter Umständen Menschenleben kosten können [27].

Ein weiterer Vorteil von Dependent Types ist, dass man mit ihnen formal verifizierte Programme schreiben kann ohne einen echten formalen Beweis zu formulieren, was man mit Beweistools, die nicht auf Dependent Types basieren, nicht unbedingt kann [28].

5.1.1 CompCert C. Ein Beispiel bei dem formale Beweissysteme in einer allgemeinen Anwendung verwendet werden ist der CompCert C Compiler[27] der an der Inria entwickelt wird. Dieser Compiler hat das Ziel, das Fehlverhalten, welches vermutlich in jedem Compiler vorhanden ist, sogenannte „Miscompilations“, auf ein Minimum zu reduzieren. Dieser Compiler ist dann besonders dort wichtig, wo es nicht in erster Linie auf die Schnelligkeit des Codes ankommt, denn diese ist im Vergleich zu herkömmlichen Compilern meist schlechter, sondern auf die absolute Vermeidung von Fehlern [27].

5.1.2 Beweis des Vier-Farben-Satzes. 1852 stellte Francis Guthrie die Vermutung auf, dass die Regionen einer planaren Karte so gefärbt werden können, dass je zwei benachbarte Regionen eine unterschiedliche Farbe haben. Diese Vermutung wird Vier-Farben-Satz[28] gennant. Sie konnte erstmals 1976 mithilfe eines Computers gelöst werden. Dabei wurde das Problem auf eine Anzahl an Instanzen verringert, welche einzeln mittels Computer auf ihre vierfärbbarkeit überprüft werden mussten. Da es jedoch möglich war, dass das Programm, welches die Instanzen überprüft, Fehler enthält, wollten Mathematiker und Informatiker sich damit nicht zufrieden geben und suchten nach einem besseren Beweis mithilfe formaler Beweistools. 2005 gelang es Georges Gonthier und Benjamin Werner den Satz mithilfe von Coq zu formalisieren und zu überprüfen [29].

5.2 Nachteile von Dependent Types

Die Nachteile die Programmierung mit Dependent Types in den meisten Fällen mit sich bringt sind zum einen eine kompliziertere Syntax, welche ein höheres Verständnis über die zugrundeliegende Mathematik voraussetzt als andere Sprachen ohne dieses Feature.

Zum anderen gibt es in Sprachen mit Dependent Types meist viele Wege um das selbe auszudrücken, was eigentlich ein Vorteil ist, jedoch die Verständlichkeit des Codes erschweren kann, wenn mehrere Personen an einem Projekt arbeiten und einen unterschiedlichen Programmierstil verwenden.

5.3 Abwägung

Es ist zu vermuten, dass Dependent Types in absehbarer Zukunft nicht zum Standard in mainstream Programmiersprachen werden. Die Softwareentwicklung mit ihnen kann sehr mathematisch werden, wodurch es für Entwickler die wenig explizites mathematisches Hintergrundwissen haben schwieriger werden würde, die Programme zu schreiben. Für Firmen, welche die Entwickler beschäftigen, würde dies vermutlich auch Mehrkosten verursachen, da die Entwicklung unter Umständen länger dauert und die Entwickler eventuell besser ausgebildet sein müssen um Dependent Types effizient und sinnvoll nutzen zu können. Da auch herkömmliche Programmiersprachen schon Checks verwenden und viele Fehler im Voraus erkennen können, wird es die weitere Verbesserung der Sicherheit im Allgemeinen wahrscheinlich den meisten Firmen und Entwicklern den zusätzlichen Aufwand nicht wert sein, vor allem da aus einer besseren Softwaresicherheit nicht direkt Kapital geschlagen werden kann.

In spezifischen Feldern wie vor allem der Mathematik und bestimmten kritischen Anwendungen, bei denen möglichst keine Fehler gemacht werden dürfen, könnte es jedoch zu einer steigenden Wichtigkeit von Dependent Types in den nächsten Jahren kommen. Zu kritischen Anwendungen könnte man zum Beispiel die Verwendung von Software in der Fahrzeuginformatik zählen, welche immer größere Wichtigkeit erlangt und in welcher das richtige und verlässliche Funktionieren der Software oft auch über Leben und Tod entscheiden kann, beispielsweise im Fall von Brake-by-wire [30] oder dem automatisierten Fahren. Auch in der Luftfahrt wird zunehmend Software zur Steuerung des Flugzeugs eingesetzt, bei welcher die Fehlfunktion im schlimmsten Fall zu Abstürzen führen kann und auch schon geführt hat [29]. Außerdem könnten sie auch vermehrt in der Websicherheit eingesetzt werden, um beispielsweise Hackerangriffe zu erschweren oder die Anwendungen resistenter gegen falsche Bedienung zu machen [31].

6 FAZIT UND ZUKÜNFTIGE ENTWICKLUNG

In dieser Arbeit wurde ein kurzer Einblick in die Welt der Dependent Types, sowie ein grober Überblick über eine mögliche Implementierung und verschiedene Use Cases gegeben. Bei der Recherche

[27] http://compcert.inria.fr/ letzter Zugriff 22.01.20

[28] http://blog.fsmath.uni-oldenburg.de/wp-content/uploads/2009/07/vier_farben_problem.pdf letzter Zugriff 22.01.20

[29] https://www.wired.com/story/boeings-737-max-cars-software/ letzter Zugriff 22.01.20

für diese Arbeit hat sich gezeigt, dass es einige Softwareentwickler gibt, die große Hoffnung auf Dependent Types setzen. Diese werben damit, dass die meisten Bugs der Vergangenheit angehören könnten, wenn alle Programme formal verifiziert wären und kein unerwartetes Verhalten mehr zeigen könnten[30].

Diese Entwicklung zeigt sich auch an Projekten wie dem Eingangs erwähnten Dependent Haskell und Implementierungen von Dependent Types in anderen mainstream Programmiersprachen wie Javascript.

Abschließend lässt sich festhalten, dass Dependent Types ein interessanter Ansatz sind um sichere Software zu schreiben, jedoch trotz der Bemühungen einiger Entwickler, welche sie in Programmiersprachen integrieren, noch nicht in der allgemeinen Softwareentwicklung angekommen sind und möglicherweise auch nie komplett dort ankommen werden. Sie könnten jedoch, sollten sie sich durchsetzen, eine massive Veränderung in der Softwareproduktion mit sich bringen, da das Testen der Software stark an Bedeutung verlieren würde.

Zur zukünftigen Entwicklung lässt sich, wie bereits in Kapitel 5 beschrieben, sagen, dass es eher unwahrscheinlich ist, dass Dependent Types sich in den mainstream Sprachen noch durchsetzen werden. Allerdings gibt es ähnliche Konzepte, die möglicherweise bessere Chancen haben, da sie nicht die selbe hohe Komplexität wie Dependent Types, jedoch trotzdem ein paar ihrer Vorteile haben. Eine solche Idee sind beispielsweise die sogenannten Refinement Types [33]. Diese können, durch festlegen von Vor- und Nachbedingungen für Funktionen, stärker einschränken welche Werte zu erwarten sind als herkömmliche Typen. Sie können jedoch nicht so viel ausdrücken wie Dependent Types. Ihr Vorteil liegt in einer einfacheren Typinferenz. Sie vereinen daher also in gewisser Weise Vorteile beider Welten.

[30] https://medium.com/background%2Dthread/the%2Dfuture%2Dof%2Dprogramming%2Dis%2Ddependent%2Dtypes%2Dprogramming%2Dword%2Dof%2Dthe%2Dday%2Dfcd5f2634878letzter Zugriff 22.01.20

LITERATUR

[1] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[2] Richard Zach. Hilbert's program. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2019 edition, 2019.

[3] José Ferreirós. The crisis in the foundations of mathematics.

[4] Panu Raatikainen. Gödel's incompleteness theorems. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2018 edition, 2018.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[6] Martin Davis. Why gödel didn't have church's thesis. *Information and control*, 54(1-2):3–24, 1982.

[7] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[8] Philip Wadler. Propositions As Types. *Communications of the ACM*, 58(12):75–84, November 2015.

[9] H. B. Curry. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584,590, 1934.

[10] Haskell Brooks Curry and Robert Feys. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

[11] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[12] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Éditeur inconnu, 1972.

[13] Douglas J Howe. The computational behaviour of girard's paradox. Technical report, Cornell University, 1987.

[14] Gérard Huet. T. Coquand. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[15] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[16] Edward F. Gehringer and J. Leslie Keedy. Tagged architecture: How compelling are its advantages? *SIGARCH Comput. Archit. News*, 13(3):162–170, June 1985.

[17] Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991.

[18] Ana Bove and Peter Dybjer. Dependent types at work. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*, pages 57–99. Springer, 2008.

[19] Thomas Jech. *Set theory*. Springer Science & Business Media, 2013.

[20] Jeremy Avigad. Type inference in mathematics. *arXiv preprint arXiv:1111.5885*, 2011.

[21] Limin Jia, Jianzhou Zhao, Vilhelm Sjöberg, and Stephanie Weirich. Dependent types and program equivalence. In *ACM Sigplan Notices*, volume 45, pages 275–286. ACM, 2010.

[22] Eric Tanter and Nicolas Tabareau. Lost in extraction, recovered.

[23] Yannick Forster and Fabian Kunze. Verified extraction from coq to a lambda-calculus. In *Coq Workshop*, volume 2016, 2016.

[24] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In *International Conference on Theorem Proving in Higher Order Logics*, pages 1–16. Springer, 2000.

[25] Georges Gonthier and Roux Stéphane Le. An ssreflect tutorial. 2009.

[26] Amy P. Felty Andrew W. Appel. Dependent types ensure partial correctnessof theorem provers. *Journal of Functional Programming*,, 14(1):3,19, January 2004.

[27] Xavier Leroy et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 53, 2012.

[28] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2013.

[29] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[30] W. Xiang, P. C. Richardson, C. Zhao, and S. Mohammad. Automobile brake-by-wire control system design and analysis. *IEEE Transactions on Vehicular Technology*, 57(1):138–145, Jan 2008.

[31] Simon Fowler and Edwin Brady. Dependent types for safe and secure web programming. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, page 49–60, New York, NY, USA, 2013. Association for Computing Machinery.

[32] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. In *ACM SIGPLAN Notices*, volume 47, pages 587–606. ACM, 2012.

[33] Tim Freeman. Refinement types ml. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994.