Methoden und Werkzeuge zur Verifikation von Software

Coq

Markus Stöhr
stoehr@informatik.uni-frankfurt.de

Frankfurt am Main
Juni 2000

Dieser Vortrag wurde im Rahmen des Seminars Methoden und Werkzeuge zur Verifikation von Software im Sommersemester 2000 an der Johann Wolfgang Goethe – Universität Frankfurt am Main von mir angefertigt und gehalten.

Inhaltsverzeichnis

Vorwort

Bei der Planung und Erstellung dieses Vortrags stieß Ich auf ein kleines Problem . Es gibt noch sehr wenige gute Dokumente für die Einführung in Coq. Das ist um so trauriger, da Coq ein wirklich mächtiges System darstellt, das vielen Menschen, aus Gründen des mangelnden Literaturangebots, hoffentlich nicht mehr lange vorbehalten bleibt (diese Bemerkung sollte als Aufruf an mögliche Autoren verstanden werden).
Da dieser Vortrag auf der Basis zweier heterogener Dokumente beruht, konnte eine klare Linie nicht immer eingehalten werden. Beim ersten Dokument handelt es sich um ein sehr Praxis orientiertes Einführungs-Tutorial. Als zweites Dokument verwendete Ich das Referenz-Manual, daß keine Einführung sondern lediglich eine Beschreibung von Coq darstellt. Sollte es beim Lesen meines Vortrags Unklarheiten geben, verweise Ich hiermit auf die genannten Dokumente. Weitere Informationen finden Sie unter Quellen, am Ende dieses Vortrags.

1 Was ist Coq ?

Coq ist ein System, das die Beschreibung logischer Konstrukte erlaubt und Werkzeuge beinhaltet, mit deren Hilfe man Beweise führen kann. Es ist ein Beweis-Assistent für Logiken höherer Ordnung. Das System ist sehr generisch und flexibel aufgebaut. Das zeigt sich zum Beispiel an dem einheitlichen Umgang mit Termen und Typen und dem wohldefinierten Kalkül. Coq ist wie eine Programmiersprache aufgebaut, bei der die Kapselung von bereits bewiesenen Theorien und für den ständigen Gebrauch benötigte mathematische Definitionen in Form von Modulen in Bibliotheken organisiert ist. Module werden dann beim Start des Systems teilweise automatisch der *Umgebung* hinzugeladen. Module können auch explizit mit einem Befehl geladen werden. Des weiteren ist es natürlich auch möglich eigene Module zu generieren.
Ähnlich verhält es sich beim Beweisführungssystem, das ein wichtiger Bestandteil von Coq ist. Es verfügt über eine Menge von Befehlen, den sogenannten *Tactics,* mit deren Hilfe man einen Beweis Schritt für Schritt führen kann. Es gibt aber auch Tactics, die mehrere Beweisschritte zusammenfassen oder kleinere Beweise automatisch durchführen. Zusätzlich gibt es auch die Möglichkeit eigene Beweisskripte, sogenannte *Tacticals,* durch Kombination von Tactics zu erstellen.
Da die Struktur des Systems auf einem typisierten Lambda-Kalkül beruht, sind natürlich fast alle denkbaren Formen von Aussagen, Datenstrukturen, Funktionen, Theorien und Programmen in Coq darstellbar, aber wie gesagt nur fast alle, da nicht alle Lambda-Terme einen einfachen Typ haben. Die Sprache in der Objekte in Coq spezifiziert werden heißt *Gallina*. Das dieser Sprache zugrundeliegende Kalkül nennt sich CIC (Calculus of Inductive Constructions). Die Befehlssprache zur interaktiven Kommunikation mit Coq nennt sich *The Vernacular*.

2 Geschichtliches

Das Vorläufer-System von Coq benutzte einen einfacheren Kalkül, das sich CoC (Calculus of Construction) nannte . Es wurde 1985 von T.Coquand und G.Huet präsentiert. Es wurde damals in einer funktionalen Programmiersprache Namens CAML bei INRIA in Rocquencout (Frankreich) implementiert. Dieses starke logische System erlaubte mächtige Axiomatisierungen, doch direkte induktive Definitionen waren noch nicht möglich. Diese mußten noch umständlich indirekt durch funktionale Schreibweise definiert werden. Deshalb wurde der Formalismus 1989 von T.Coquand und C.Paulin zum heutigen CIC erweitert.
CIC kann verstanden werden als Mixtur von Prädikaten-Kalkül, induktiver Definitionen wie in Prolog und rekursiver Funktions-Defnitionen ähnlich wie beim funktionalen Programmieren.

3 Einführung in Gallina

Gallina dient der Entwicklung und Formulierung mathematischer Theorien und Programm-Spezifikationen. Theorien werden aus folgenden Komponenten erstellt: Axiome, Hypothesen, Parameter, Lemmas, Theoreme und Definitionen von Konstanten, Funktionen, Prädikaten und Mengen.
In Coq haben alle Objekte einen Typ, der zu logischen Formeln korrespondiert. Dafür gibt es den Kalkül, der festlegt, welche Konstruktionen von Formeln erlaubt sind bzgl. ihrem Typs.
Im Kapitel CIC wird genau auf die festgelegte Syntax und Semantik eingegangen. Vorab sollen hier einige einfache Beispiele von Formulierungen mathematischer Aussagen gegeben werden.
Für Coq gibt es einen Shell-Modus, indem wie in einer Shell üblich, Befehle nacheinander eingegeben werden.
Objekte können in Gallina, wie in anderen Programmiersprachen auch, deklariert und definiert werden:

Coq < Variable n:nat.
n is assumed

Hier wurde eine Variable n vom Typ nat deklariert. Wie bereits weiter oben angedeutet haben alle Objekte in Coq einen Typ, also auch nat:

Coq < Check nat.
nat
: Set

nat hat also den Typ Set. Und Set ?

Coq < Check Set.
set
: Type

Alle Objekte in Coq sind letzlich vom Typ *Type*. Wie das genau funktioniert sehen Sie im anschließenden Kapitel.

Wie bereits erwähnt, werden bei der Initialisierung von Coq bereits einige Module geladen, darunter auch eins das bereits einige arithmetische Definitionen enthält. So sind zum Beispiel folgende Objekte verfügbar:

Coq < Check O.
O (O repräsentiert nie Null)
: nat

Coq < Check S.
S (S ist die Nachfolger-Funktion)
: nat -> nat

Coq < Check plus.
plus (plus addiert zwei Werte)
: nat -> nat -> nat

(nat → nat → nat) steht dabei für (nat → (nat → nat)). Hier deutet „→“ auf eine Funktion hin.
Nun zu einer einfachen Definition:

Coq < Definition one : nat := (S O).
one is defined

Hier wurde soeben die Konstante one vom Typ nat als Nachfolger von O definiert.
Funktionen definiert man mittels der Abstraktions-Klammern [ ]:

Coq < Definition double := [m:nat] (plus m m).
double is defined

Coq < Check double.
double
: nat -> nat

Die Semantik der Abstraktion entspricht genau der, des Lambda-Kalküls. Wird eine Funktion auf ein Objekt angewendet – der Typ dieses Objekts muß gleich dem Typ des Abstraktions-Objekt sein – wird das Abstraktions-Objekt durch dieses im Term ersetzt:

Coq < Definition two:= double one.
two is defined

Coq < Check two.
two
: nat

Wir stellen noch eine weitere Funktion vor:

Coq < Check gt.
gt („ist größer als“-Relation)
: (_,_:nat)Prop

Diese Funktion kann dargestellt werden als: (nat → (nat → Prop)). Das heißt sie erwartet zwei Zahlen und ergibt eine Aussage Prop.
Weiter oben wurde eine Variable n als nat deklariert. Möchte man die Variable weiter spezifizieren, könnte man der Umgebung eine Hypothese hinzufügen:

Coq < Hypothese n_>_two : (gt n two).
n_>_two is assumed

Diese Hypothese besagt also, daß n größer ist als two. Wird sie surch deklaration in den Kontext aufgenommen, wird also gezwungen, daß n größer ist als two, damit die Hypothese erfüllt ist.

Wir haben oben die ganze Zeit den induktiv definierten Typ „nat“ benutzt. Wir zeigen jetzt, wie ein solcher Typ definert wird:

Coq < Inductive nat : Set := O : nat | S : nat -> nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined

In der Definition werden zwei Konstruktoren angegeben: O und S. O hat den Typ nat und S hat den Typ nat->nat. Diese Konstruktoren werden wie auch nat ebenfalls in die Umgebung als neu definierte Konstanten aufgenommen.
Zusätzlich werden noch eine Reihe von Destruktoren definiert. Sie stellen Eliminations-Regeln für den Umgang mit Objekten induktiven Typs dar. Um dies besser zu verstehen, schauen wir uns mal den Aufbau von nat_ind an:

Coq < Check nat_ind.
nat_ind
: (P : (nat->Prop)) (P O) -> ((n:nat)(P n) -> (P(S n))) -> (n:nat)(P n)

Diese Regel stellt Peano’s Induktions-Prinzip dar und kann bei einer Beweisführung als Eliminationsregel genutzt werden. Beispiele dafür, werden am Schluß dieses Vortrags präsentiert.

Wir stellen jetzt noch kurz da, wie eine rekursive Funktion auf induktiven Typen wie “nat” in Gallina definiert wird:

Coq < Fixpoint plus [n:nat] : nat -> nat := [m:nat] Cases n of O => m
| (S p) => (S (plus p m))
end.
plus is recursivley defined

Durch den Cases-Operator wird also der Konstruktor des Typs des Wertes n erkannt und es wird entsprechend rekursiv fortgefahren.
Wir wollen uns nun die Addition von one und two ausgeben lassen:

Coq < Eval Compute in (plus one two).
= (S (S (S O)))
: nat

Der Befehl führt auf den Term (plus one two) eine Reduktion aus. Diese wird unter anderem im nächsten Kapitel behandelt.

Bevor wir nun Beweise exemplarisch vorführen, werden wir im Folgenden erst einmal das Kalkül von Coq darstellen.

4 CIC (Calculus of Inductive Construction)

In CIC haben – wie bereits angedeutet – alle Objekte einen Typ. Sowohl Daten, Funktionen und Programme, wie auch Beweise und Typen selbst haben einen Typ. Aus diesem Grunde ist zum Beispiel folgende Aussage „für alle x, P“ nicht erlaubt, da x kein Typ zugeordnet wurde. Richtig muß es heißen: „für alle x vom Typ T, P“.
CIC beinhaltet auch ein Reduktions-Mechanismus. Es beinhaltet Konversions-Regeln, durch die eine Gleichheit von Objekten festgelegt wird.

4.1 Terme

Die meisten Typ-Theorien machen einen syntaktischen Unterschied zwischen Typen und Terme. Das heißt es gibt getrennte Regeln für die Schreibweise von Typen und Termen. Coq dagegen benutzt für Definitionen von Typen und Termen die gleiche syntaktische Struktur. Zum Beispiel stellt nat den Typ für natürliche Zahlen, nat → nat den Typ von Funktionen von nat auf nat dar. Nat → Prop stellt den Typ eines unären Prädikats über den natürlichen Zahlen dar. Zum Beispiel ist [x : nat] (x = x) ein solches Prädikat. (P x) stellt dann eine Aussage dar und (x : nat)(P x) den Typ von Funktionen, die zu jeder natürlichen Zahl x ein Objekt vom Typ (P x) zuordnet und sie formuliert somit die Aussage „$\forall x.P(x)$“.

Sorten
Der Typ eines Typs ist immer eine Konstante der Sprache, genannt Sorte. Die zwei basis Sorten in der Sprache von CIC sind Set und Prop. Der Typ Prop stellt den Typ einer logischen Aussage dar. Set ist der Typ für Spezifikationen von Programmen und üblichen Mengen wie die Menge der natürlichen Zahlen, die boolsche Menge, Listen etc. .
Sorten werden ebenfalls wie Terme behandelt, was bedeutet, daß Sorten ebenfalls wieder einen Typ besitzen müssen. Würde man einfach annehmen, daß Set wieder vom Typ Set wäre, würde das zu einer inkonsistenten Theorie führen. Deshalb gibt es in CIC eine Hierarchie von Typen, wie folgt definiert:

$S \equiv \{\text{Prop},\text{Set},\text{Type}(i) \mid i \in \text{N}\}$

Prop und Set sind vom Typ Type(0)

und Type(i) ist vom Typ Type(i+1)

Der Index bleibt dabei für den Benutzer verborgen, so daß gilt: Type : Type.

Konstanten
Neben den Sorten gibt es auch andere Konstanten in der Umgebung. Es handelt sich dabei um vorherige Definitionen von Objekten aber auch um Objekte bzgl. induktiver Definitionen.
In Coq werden induktive Objekte also als in der Umgebung deklarierte Konstanten behandelt. Eine andere Implementierung von induktiven Objekten als Terme wäre jedoch auch denkbar.

Sprache
Hier folgt nun eine genau Definition der Regeln, die die Sprache des CIC beschreibt:

1. Die Sorten Set, Prop, Type sind Terme.

2. Konstanten der Umgebung sind Terme.

3. Variable sind Terme.

4. Ist x eine Variable und sind T, U Terme, dann ist auch (x : T)U ein Term. Ist x in U vorhanden, so nennt man (x : T)U ein abhängiges Produkt, auch bekannt als „für alle x vom Typ T, U“. Ist x nicht in U vorhanden, so nennt man (x : T)U ein unabhängiges Produkt und schreibt auch: $T \rightarrow U$.

5. Ist x eine Variable und sind T, U Terme, dann ist auch [x : T]U ein Term. Dies entspricht der Notation der λ-Abstraktion des λ-Kalküls. Der Term [x : T]U ist eine Funktion, die Elemente von T auf U abbildet.

6. Sind T und U Terme, so ist auch (T U) ein Term. Der Term (T U) bedeutet „T angewendet auf U“.

Die Substitution von einer Variablen x durch einen Term T in einem Term U wird geschrieben als U{x / T}.

4.2 Typisierte Terme

Als Objekte einer Typen-Theorie, müssen alle Terme einer Typen-Diziplin folgen. Die wohl geformte Typisierung eines Terms hängt von der Menge der deklarierten Variablen, genannt der *Kontext*, ab. Ein Kontext Γ hat also die Form [x1 : T1; ... ;xn : Tn]. Falls Γ x : T enthält, schreiben wir $(x : T) \in \Gamma$. Die Notation $\Gamma :: (y : T)$ bezeichnet den Kontext [x1 : T1; ... ;xn : Tn;y : T].

Umgebung
Da wir mit Konstanten arbeiten, müssen wir auch eine Umgebung *E* berücksichtigen. $(c : T) \in E$ bedeutet, daß die Konstante c mit Typ T in *E* definiert ist.
Im folgenden nehmen wir an, daß E eine gültige Umgebung ist. Wir definieren nun zwei Ausdrücke. Der erste $E[\Gamma] \vdash t : T$ besagt, daß der Term t in der Umgebung E und dem Kontext Γ wohl geformt und vom Typ T ist. Der zweite $\mathit{WF}(E)[\Gamma]$ besagt, daß die Umgebung E wohl geformt und der Kontext Γ ein gültiger Kontext in dieser Umgebung ist.
Ein Term t ist nun dann wohl typisiert in einer Umgebung E, wenn es einen Kontext Γ und einen Term T gibt, so daß die Ableitung $E[\Gamma] \vdash t : T$ sich aus den folgenden Regeln ergibt:

W-E

$$\mathbf{WF}\ ([])[[]]$$

Dies ist die Startsituation. Eine leere Umgebung mit leerem Kontext ist wohl geformt.

W-s

$$\frac{\boldsymbol{E}[\Gamma] \mid\!- \mathbf{T : s} \quad \mathbf{s} \in \boldsymbol{S} \quad \mathbf{x} \notin \Gamma \cup \boldsymbol{E}}{\mathbf{WF}\ (\mathbf{E})[\ \Gamma :: (\mathbf{x : T})]}$$

Diese Regel besagt, daß die Hinzunahme einer Deklaration einer Variablen x, die noch nicht vergeben wurde, in den Kontext dann zu einer wohl geformten Umgebung mit gültigem Kontext führt, wenn der Typ dieser Variablen selber den Typ Prop, Set oder Type hat. Zum Beispiel beim Deklarieren von x mit Typ nat, wenn nat den Typ Set hat und x noch nicht vergeben wurde

Ax

$$\frac{\mathbf{WF}\ (\mathbf{E})[\Gamma]}{\boldsymbol{E}[\Gamma] \mid\!- \texttt{Prop : Type(p)}} \qquad \frac{\mathbf{WF}\ (\mathbf{E})[\Gamma]}{\boldsymbol{E}[\Gamma] \mid\!- \texttt{Set : Type(q)}}$$

$$\frac{\mathbf{WF}\ (\mathbf{E})[\Gamma] \quad \mathbf{i < j}}{\boldsymbol{E}[\Gamma] \mid\!- \texttt{Type(i) : Type(j)}}$$

Hier wird definiert, daß der Typ von Prop und Set immer vom Typ Type ist und außerdem, daß der Typ von Type immer Type ist, jedoch mit einem höheren Index.

Var

$$\frac{\mathbf{WF}\ (\mathbf{E})[\Gamma] \quad (\mathbf{x : T}) \in \Gamma}{\boldsymbol{E}[\Gamma] \mid\!- \mathbf{x : T}}$$

Hier ist einfach angegeben, daß eine Variable x vom Typ T sich dann aus ***E*[Γ]** ableiten läßt, wenn x :T im Kontext enthalten ist.

Const

$$\frac{\mathbf{WF}\ (\mathbf{E})[\Gamma] \quad (\mathbf{c : T}) \in \mathbf{E}}{\boldsymbol{E}[\Gamma] \mid\!- \mathbf{c : T}}$$

Das gleich gilt auch für Konstanten. Jedoch sind diese natürlich nicht im Kontext sondern in der Umgebung definiert.

Prod

$$\frac{\boldsymbol{E}[\Gamma] \mid\!- \mathbf{T} : s_1 \quad \boldsymbol{E}[\Gamma :: (\mathbf{x : T})] \mid\!- \mathbf{U} : s_2 \quad s_1 \in \{\mathbf{Prop, Set}\} \ \ \mathbf{or} \ \ s_2 \in \{\mathbf{Prop, Set}\}}{\boldsymbol{E}[\Gamma] \mid\!- (\mathbf{x : T})\mathbf{U} : s_2}$$

$$\frac{\boldsymbol{E}[\Gamma] \mid\!- \mathbf{T : Type(i)} \quad \boldsymbol{E}[\Gamma :: (\mathbf{x : T})] \mid\!- \mathbf{U : Type(j)} \quad \mathbf{i} \le \mathbf{k} \quad \mathbf{j} \le \mathbf{k}}{\boldsymbol{E}[\Gamma] \mid\!- (\mathbf{x : T})\mathbf{U : Type(k)}}$$

Diese zwei Regeln beschreiben die Voraussetzungen und das Resultat bei der Einführung eines Produkts. Wenn ein Produkt (x : T) U erwünscht ist, so muß also erst einmal der Typ T vom Typ Type(i) vorhanden sein und es muß sich bei der Hinzunahme von (x : T) in den

Kontext der Typ U vom Typ Type(j) ableiten lassen. Außerdem hat (x :T) U dann den Typ Type(k) bei der Annahme, daß $i \le k$ und $j \le k$.

Lam

$$\frac{\boldsymbol{E}[\Gamma] \vdash (x : T)U : s \quad \boldsymbol{E}[\Gamma :: (x : T)] \vdash t : U}{\boldsymbol{E}[\Gamma] \vdash [x : T]t : (x : T)U}$$

Die Lambda-Regel beschreibt die Erstellung einer Funktion. Es muß erst einmal ein Produkt der Form (x: T) U vom Typ s ableitbar sein. Unter der Annahme, daß t vom Typ U aus einem Kontext, das (x : T) enthält, ableitbar ist, so folgt, daß auch der typisierte Labda-Ausdruck [x : T]t vom Typ (x : T)U ableitbar ist.
Dies gilt für alle Produkte und insbesondere auch für unabhängige Produkte. Da kann man dann die Schreibweise vereinfacht darstellen:

Lam_simple

$$\frac{\boldsymbol{E}[\Gamma] \vdash T \to U : s \quad \boldsymbol{E}[\Gamma :: (x : T)] \vdash t : U}{\boldsymbol{E}[\Gamma] \vdash [x : T]\, t : T \to U}$$

Diese vereinfachte Form der Lambda-Regel werden wir beim Beweisen sehr oft benutzen. Sie bezieht sich auf Funktionen, deren Typ fest ist und nicht von ihrem Übergabewert abhängt.

Die folgende Regel führt die Anwendung eines Terms, dessen Typ ein Produkt ist, auf einen Term ein:

App

$$\frac{\boldsymbol{E}[\Gamma] \vdash t : (x : U)T \quad \boldsymbol{E}[\Gamma] \vdash u : U}{\boldsymbol{E}[\Gamma] \vdash (t\ u) : T\{x / u\}}$$

Wir sehe, daß die Anwendung eine Veränderung des Typs durch die Substitution bewirkt. Das wäre nicht der Fall wenn wir wie oben von einem unabhängigen Typ ausgehen:

App_simple

$$\frac{\boldsymbol{E}[\Gamma] \vdash t : U \to T \quad \boldsymbol{E}[\Gamma] \vdash u : U}{\boldsymbol{E}[\Gamma] \vdash (t\ u) : T}$$

In dieser Regel ist t also ein Term dessen Typ ein unabhängiges Produkt ist. Der Typ sagt uns, daß ein Term vom Typ U erwartet und ein Term vom Typ T zurückgegeben wird. Auf diese Regel werden wir während des Beweisens auch andauernd stoßen.

4.3 Konversions Regeln

Die **β-Reduktion** (oder auch Konversion) definiert den Term, der nach einer Anwendung entsteht. Geschrieben lautet sie:

$$([x : T]t\ u) \blacktriangleright_\beta t\{x / u\}$$

Die **ι-Reduktion** steht in Verbindung mit den induktiv definierten Objekten in der Umgebung. Sie stellt den „case“-Operator dar, der induktiv definierte Objekte „auseinander-nimmt“.

Die δ-**Reduktion** dient der Ersetzung einer Konstante durch ihren Term. Sie kann auch als Definitions-Expansion bezeichnet werden.

Die Schreibweise t ► u sagt nun, daß t durch die Reduktionen β, ι oder δ zu u reduziert. Zwei Terme t1 und t2 sind nun genau dann *konvertibel*, wenn es einen Term u gibt, so daß t1 ► ... ► u und t2 ► ... ► u gilt. Wir schreiben t1 $=_{\beta\iota\delta}$ t2.

Die Konversions-Relation erlaubt das Einführen einer neuen Regel bzgl. Typen.
Vorher definieren wir eine neue Relation $\leq_{\beta\iota\delta}$:

1. ist M $=_{\beta\iota\delta}$ N, dann auch M $\leq_{\beta\iota\delta}$ N,

2. ist i ≤ j, dann Type(i) $\leq_{\beta\iota\delta}$ Type(j),

3. ist T $=_{\beta\iota\delta}$ U und ist M $\leq_{\beta\iota\delta}$ N, dann (x : T)M $\leq_{\beta\iota\delta}$ (x : U)N.

Die Konversionsregel lautet nun:

Konv

$$\frac{E[\Gamma] \vdash U : S \quad E[\Gamma] \vdash t : T \quad U \leq_{\beta\iota\delta} T}{E[\Gamma] \vdash t : U}$$

Ein Term ist in der **Normal-Form,** wenn dieser Term nicht weiter reduziert werden kann. Beim Reduktions-Mechanismus liegt eine Priorität natürlich in der Terminierung. Es dürfen also keine Funktionen und induktive Objekte definiert werden, die möglicherweise eine Reduzierung bewirken, die niemals terminiert. Aus diesem Grund gibt es für die Definition von induktiven Typen und rekursiven Funktionen Regeln, die eine Korrektheit und Terminierung garantieren. Dies wird bewirkt, indem implizit nur Funktionen akzeptiert werden, die primitiv rekursiv sind. Es wird jedoch innerhalb dieses Vortrags nicht näher darauf eingegangen. Hiermit sei auf das Referenz-Handbuch verwiesen.

4.4 Definitionen in der Umgebung

Wir geben nun die Regeln an, die uns den Umgang mit Konstanten in der Umgebung ermöglichen. Da eine Konstante von vorher definierten Konstanten abhängen kann, stellt die Umgebung eine geordnete Liste von Deklarationen dar.
Es können zwei Sorten von Objekten in die Umgebung eingefügt werden, einfache Objekte und induktive Objekte.

4.4.1 Einfache Definitionen

Definitionen in der Umgebung werden durch Def(Γ)(c := t : T), mit der Bedeutung, daß c mit dem Wert t und dem Typ T eine gültige Konstante in dem Kontext Γ ist, repräsentiert.
Wenn nun Def(Γ)(c := t : T) in der Umgebung E ist, so wird automatisch auch die δ-Reduktion c $\blacktriangleright_{\delta}$ t eingefügt.
Die Regel zur Hinzunahme einer Definition lautet:

Def

$$\frac{E[\Gamma] \vdash t : T \quad c \notin E \cup \Gamma}{\mathcal{WF}\ (E; \mathrm{Def}(\Gamma)(c := t : T))[\Gamma]}$$

Nun lassen sich aus den genannten Regeln weitere ableiten. Sie werden hier nur kurz erläutert:

Substitutions-Eigenschaft:

$$\frac{\mathcal{WF}\ (E; \mathrm{Def}(\Gamma)(c := t : T);F)[\Gamma]}{\mathcal{WF}\ (E;F\{c/t\})\ [\Delta\{c/t\}]}$$

Diese Regel beschreibt die Substitution einer Konstanten durch seinen Term in der Umgebung.

Weitere Regeln:

Abstraktions-Eigenschaft:

$$\frac{\mathcal{WF}\ (E; \mathrm{Def}(\Gamma::(x : U))(c := t : T);F)[\ \Delta] \quad \mathcal{WF}\ (E)\ [\Gamma]}{\mathcal{WF}\ (E; \mathrm{Def}(\Gamma)(c := [x : U]t : (x : U)T);F\{c/(c\ x)\})\ [\Delta\{c/(c\ x)\}]}$$

Reduzierungs-Eigenschaft:

$$\frac{\mathcal{WF}\ (E; \mathrm{Def}(\Delta)(c := t : T))[\Gamma]}{\mathcal{WF}\ (E; \mathrm{Def}(\Delta)(c := t : T))[\Delta]}$$

4.4.2 Induktive Definitionen

Wir haben in der Einführung bereits ein Beispiel für eine Induktion präsentiert. In Coq werden Induktions-Typen wie „nat“ als Konstanten in der Umgebung behandelt. Wie wir in dem Beispiel sahen, definiert man eine Induktion mittels Konstruktoren. Diese werden ebenfalls in die Umgebung eingefügt. Wir zeigen nun wie eine induktive Definition im Kalkül gehandhabt wird. Es folgt die Deklaration des Typs für natürliche Zahlen:

Ind()(nat : Set := O : nat, S : nat → nat)

Eine Deklaration kann auch vorher deklarierte Variablen einbeziehen:

Ind(A : Set)(List : Set := nil : List, cons : A → List → List)

Hier haben wir den Typ einer Liste definiert, die Elemente aus A enthält.

Generell hat eine induktive Definition die Form Ind(Γ)($\Gamma_I := \Gamma_C$), wobei Γ der Kontext ist, in der Die Definition gültig ist und Γ_I den Definitions-Kontext und Γ_C den Konstruktoren-Kontext darstellt.
Für $\Gamma_I = [I_1 : A_1; \dots ;I_k : A_k]$ und $\Gamma_C = [c_1 : C_1; \dots ; c_n : C_n]$ gilt:

$$\frac{\mathrm{Ind}(\Gamma)(\ \Gamma_I := \Gamma_C\ ) \in E \quad j = 1 \dots k}{(I_j : A_j) \in E}$$

$$\frac{\mathbf{Ind}(\Gamma)(\Gamma_I := \Gamma_C) \in \mathbf{E} \quad i = 1 \ldots n}{(c_i : C_i\{I_j / I_j\}j=1\ldots k) \in \mathbf{E}}$$

Dies sind Regel, die die Definitionen des induktiven Typs sowie ihre Konstruktoren in die Umgebung einfügen.

Es gibt in Coq noch Varianten induktiver Definitionen, auf die hier aber nicht weiter eingegangen werden soll.

Wir geben hier noch zur Vervollständigung die Regel zur Einfügung einer induktiven Definition in die Umgebung. Sie stellt Bedingungen auf, die für eine erlaubte Einfügung erfüllt sein müssen. Sie soll hier nicht näher erläutert werden:

W-Ind

$$\frac{(E[\Gamma] \vdash A_j : s'_j)_{j=1\ldots k} \quad (E[\Gamma;\Gamma_I] \vdash C_i : s_{pi})_{i=1\ldots k}}{\mathit{WF}\,(E;\, \mathbf{Ind}(\Gamma)(\Gamma_I := \Gamma_C))\,[\Gamma]}$$

Bestimmte Nebenbedingungen müssen dabei erfüllt sein, die hier auch unterschlagen werden. Bei Interesse sei der Leser auf das Referenz-Manual verwiesen.

Destruktoren
Im einführenden Beispiel war auch die Rede von Destruktoren. Sie werden bei der induktiven Definition automatisch erzeugt und in die Umgebung hinzugefügt. Sie dienen in Beweisen als Rückführung von einem induktiven Objekt auf Objekte, die mittels der Konstruktoren gebildet wurden. Destruktoren stellen Eliminations-Regeln für induktive Objekte dar.
Als Beispiel schauen wir uns einen Destruktor an, wie er bei der Definition von „nat“ erzeugt wurde:

Coq < Check nat_ind.
nat_ind
: (P : (nat->Prop)) (P O) -> ((n:nat)(P n) -> (P(S n))) -> (n:nat)(P n)

Der Destruktor besagt also, daß für alle Prädikate des Typs nat->Prop folgendes gilt:
Gilt (P O) und gilt für alle n:nat ((P n)->(P(S n))), so gilt für alle n:nat (P n).
Hat man nun eine Aussage (n:nat)(P n) kann man also mittels des Destruktors auf die Konstruktoren rückschließen. Dazu schaue man sich die Beispiele im letzten Kapitel an.
Mittels der Destruktoren lassen sich wie wir sehen werden sogar die primitiv rekursiven Funktionen definieren.
Da der Umgang mit Destruktoren aber sehr verwirrend ist, wurde der „case“-Operator eingeführt mit dessen Hilfe man Funktionen einfacher definieren kann. Auch darauf soll hier nicht weiter eingegangen werden. Es soll hier lediglich erwähnt werden, daß die ι-Reduktion genau die Arbeitsweise des „case“-Operators beschreibt.

5 Einführung in die Beweisführung

Eine Deduktions-Regel ist eine Verbindung einer Formel, die Konklusion, mit einer Menge von Formeln, die Pämisse. Diese Regel kann nun auf zwei unterschiedliche Weise gelesen werden. Aus der Prämisse kann auf die Konklusion geschlußfolgert werden. Zum Beispiel haben wir einen Beweis von A und einen Beweis von B, uns somit können wir auf $A \wedge B$ schließen. Dies nennt man vorwärts gerichtetes Schließen, von der Prämisse zur Konklusion. Man kann aber auch folgende Aussage treffen: Um $A \wedge B$ zu beweisen, muß Ich A und B beweisen. Dies nennt man rückwärts gerichtetes Schließen, von der Konklusion zurück auf die Prämisse. Genau dies ist die Aufgabe der Tactics. Wir nennen dabei die Konklusion das Ziel, das bewiesen werden soll. Die Prämissen sind dann die Unterziele, die ebenfalls bewiesen werden müssen, um das Ziel zu beweisen. Wird nun eine Tactic an ein Ziel angewendet, so wird dieses durch die Unterziele, die durch die Anwendung der Tactic generiert wurden ersetzt. Dabei kann ein oder mehrere neue Unterziele entstehen.

Im folgenden werden wir an einer Reihe von Beispielen die verschiedenen Tactics und deren Benutzung erläutern. Dabei soll dem Leser ein Überblick über die Funktionsweise der Beweisführung auf Grundlage des Kalküls gegeben werden.

5.1 Implikation

Wir deklarieren im folgenden drei Aussagenvariablen und werden eine einfache Tautologie beweisen:

```
Coq < Variables A,B,C : Prop.
A is assumed
B is assumed
C is assumed

Coq < Goal (A -> (B -> C)) -> ((A -> B) -> (A -> C))
1 subgoal

        ===================================
          (A -> (B -> C)) -> ((A -> B) -> (A -> C))
```

Durch den Befehl Goal sind wir in nun im Beweis-Modus. Der angegebene Term hinter dem Befehl Goal ist nun der Typ unseres generierten Ziels.
Durch die Tactic Intro werden Produkte aufgespalten. Dies wir durch die Lambda-Regel ermöglicht. Da Tactics ja Regeln immer rückwärts anwenden folgt also nach der Lambda-Regel, daß wir aus $T \rightarrow U$, das neue Ziel U bekommen, wobei T in den Kontext eingefügt wird. Der linke Teil wird also zu einer neuen Hypothese in unserem Kontext und der rechte Teil bildet das neue Ziel.

```
Coq < Intro H.
1 subgoal

        H : A->(B->C)
        =============
        (A->B)->(A->C)
```

Die neue Hypothese bekommt den Namen H wie angegeben. Durch den Befehl Intros werden automatisch alle möglichen Produkte gespalten und in den Kontext eingefügt:

```
Coq < Intros.
1 subgoal

        H : A->(B->C)
        H0 : A->B
        H1 : A
        ==============
        C
```

Mit der Tactic Apply können wir nun eine Hypothese an das aktuelle Ziel anwenden. Diese Tactic benutzt die App-Regel, natürlich wieder rückwärts. Wir nehmen an, daß unser Ziel der Typ der Anwendung (t u) ist und erhalten dann u:U als neues Ziel. Diese Prozedur wird bei der Apply-Tactic mehrmals in Folge ausgeführt, so daß wir wie in diesem Beispiel mehrere Ziele bekommen:

```
Coq < Apply H.
2 subgoals

        H : A->(B->C)
        H0 : A->B
        H1 : A
        =============
        A
subgoal 2 is:
        B
```

Wir können jetzt das erste Ziel der beiden lösen da wir ja eine Hypothese im Kontext haben, die dem Ziel enspricht. Dies gelingt mittels der Tactic Assumption, die die Var-Regel implementiert:

```
Coq < Assumption.
1 subgoal

        H : A->(B->C)
        H0 : A->B
        H1 : A
        =============
        B
```

Nun wenden wir die Hypothese H0 an:

```
Coq < Apply H0.
1 subgoal
        H : A->(B->C)
        H0 : A->B
        H1 : A
        =============
        A
```

Nun können wir wieder mit der Tactic Assumption das aktuelle Ziel erreichen:

```
Coq < Assumption.
Subtree proved !
```

Nun ist der Beweis fertig und wir können ihn auch speichern, um ihn wiederum in weiteren Beweisen zu benutzen, dann wieder mit der Apply-Tactic.

5.2 Aussagenlogik

Schlußfolgerungs-Regeln können immer in zwei Kategorien aufgeteilt werden, in Einfügeregeln(mittels Konstruktoren) und Eliminationsregeln(mittels Destruktoren). Aber nicht vergessen, daß die Tactics diese Regeln rückwärts benutzen um aus einem Ziel neue Unterziele zu generieren.
Wie wir aus dem Vortrag über konstruktive Logik bereits wissen, wird für jeden Operator immer mindestens eine Einfügeregel und mindestens eine Eliminationsregel eingeführt. Coq besitzt nun eine Tactic mit Namen „Constructor“ und eine Tactic mit Namen „Elim“, die automatisch die richtige Einfügeregel bzw. Eliminationsregel für einen bestimmten Operator findet und anwendet.
Als erstes wollen wir die Kommutativität der Konjunktion beweisen:

```
Coq < Goal and_comm : A /\ B -> B /\ A.
1 subgoal
          =============
           A /\ B -> B /\ A

Coq < Intro H.
1 subgoal

          H : A /\ B
          ============
          B /\ A

Coq < Elim H.
1 subgoal

          H : A /\ B
          ============
          A -> B -> B /\ A

Coq < Intros.
1 subgoal

          H : A /\ B
          H0 : A
          H1 : B
          ============
          B /\ A
```

Und nun wenden wir die Einfüge-Regel mittels der Construct-Tactic an und erhalten 2 neue Unterziele:

```
Coq < Constructor.
2 subgoals

        H : A /\ B
        H0 : A
        H1 : B
        ============
        B
Subgoal 2 is:
A
```

Der Rest des Beweises ist nun trivial.

Wir wollen nun die Kommutativität der Disjunktion beweisen:

```
Coq < Goal or_comm : A \/ B -> B \/ A.
1 subgoal
        =============
         A \/ B -> B \/ A

Coq < Intro H.
1 subgoal

        H : A \/ B
        ============
        B \/ A
```

Jetzt wenden wir die Eliminationsregel an unter Angabe einer Hypothese H:

```
Coq < Elim H.
2 subgoals

        H : A \/ B
        ============
        A -> B \/ A
subgoal 2 is:
B -> B \/ A

Coq < Intro HA.
2 subgoals

        H: A \/ B
        HA : A
        ============
        B \/ A
subgoal 2 is:
B -> B \/ A
```

Nun können wir mittels einer der beiden Einfüge-Regeln für die Disjunktion ein neues Ziel erzeugen:

```
Coq < Constructor 1.
2 subgoals

        H: A \/ B
        HA : A
        ============
        A
subgoal 2 is:
B -> B \/ A
```

Nun hat das Ziel also den Typ A und damit identisch mit Hypothese HA. Genauso verfahren wir mit dem zweiten Ziel. Damit ist der Beweis vollständig.

5.3 Induktion

Boolsche Werte
Wir definieren im Folgenden den Typ für boolsche Werte:

```
Coq < Inductive bool : Set := true : bool | false : bool.
bool_ind is defined
bool_rec is defined
bool_rect is defined
bool is defined

Coq < Check bool_ind.
bool_ind
        : (P: (bool -> Prop)) (P true) -> (P false) -> (b : bool) (P b)
```

Nun werden wir mittels des Destrukors beweisen, daß jeder Variable boolschen Typs entweder true oder false ist:

```
Coq < Goal (b:bool) (b=true \/ b=false).
1 subgoal

        ======================
        (b:bool) (b=true \/ b=false)

Coq < Intro b.
1 subgoal

        b : bool
        ======================
         b=true \/ b=false
```

```
Coq < Elim b.
2 subgoals

        b : bool
        ======================
         true=true \/ true=false
subgoal 2 is:
        false=true \/ false=false
```

Nun läßt sich wieder die Einfüge-Regel für die Disjunktion anwenden und der Beweis ist dann klar.

Natürliche Zahlen
Zum Schluß wollen wir darstellen, wie sich Funktionen auf den natürlichen Zahlen mittels des Destruktors nat_ind definieren lassen:

```
Coq < Definition prim_rec := (nat_rec [i : nat] nat).
prim_rec is defined
```

Wir schauen uns jetzt den Typ von prim_rec an:

```
Coq < Check prim-rec.
prim_rec
        :  ([_:nat]nat O)
        ->((n:nat)([_:nat]nat n) -> ([_:nat]nat (S n)))
        ->(n:nat)([_:nat]nat n)
```

Eine Funktion wie zum Beispiel die Addition wird nun auf folgender Weise definiert:

```
Coq < Definition addition := [n,m:nat] (prim_rec m [p:nat] [rec:nat] (S rec) n).
```

Wir testen nun addition:

```
Coq < Eval Compute in (addition (S (S O)) (S (S (S O)))).
        = (S (S (S (S (S O)))))
        : nat
```

Wie Sie leicht merken werde, ist diese Art der Definition einer Funktion ziemlich unverständlich. Das ist auch der Grund wieso der case-Operator zusammen mit der Fixpoint-Definition für Funktionen eingeführt wurde.

Mit diesen leider sehr kurzgefassten Worten endet dieser Vortrag. Da dieser Vortrag beim Leser sicherlich sehr viele Fragen unbeantwortet ließ, verweise Ich abermals auf die folgenden Quellen:

6 Quellen

Tutorial und Referenz-Manual finden Sie unter:

http://pauillac.inria.fr/coq/pres-eng.html