FernUniversität in Hagen
Fakultät für Mathematik und Informatik

AIG
Artificial Intelligence Group

Über Formalisierung und semi-maschinelles Beweisen des Standard-Repräsentationstheorems für kumulatives Schließen in Propositionaler Logik mit Coq

Bachelorarbeit

zur Erlangung des Grades einer Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Jonathan Heinrich Walther

Erstgutachter:	Prof. Dr. Kai Sauerwald Artificial Intelligence Group
Betreuer:	Prof. Dr. Kai Sauerwald Artificial Intelligence Group

Erklärung

Ich erkläre, dass ich die Bachelorarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Bachelorarbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Bachelorarbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Der Veröffentlichung dieser Arbeit auf der Webseite des Lehrgebiets Künstliche Intelligenz und damit dem freien Zugang zu dieser Arbeit stimme ich ausdrücklich zu.

Für diese Arbeit erstellte Software wurde quelloffen verfügbar gemacht, ein entsprechender Link zu den Quellen ist in dieser Arbeit enthalten. Gleiches gilt für angefallene Forschungsdaten.

(Ort, Datum) (Unterschrift)

Zusammenfassung

Diese Arbeit widmet sich der Formalsierung und dem Beweisen des Repräsentationstheorems für kumulatives Schließen mit dem Beweisassistenten Coq. Das Theorem stellt eines der zentralen Theoreme der nichtmonotonen Logik dar. Nichmonotones Schließen ermöglicht die Revision von Schlüssen bei neuen Informationen und berücksichtigt Ausnahmen, was dem menschlichen Denken näher kommt als die klassische monotone Logik. Ein typisches Beispiel ist „Vögel fliegen“, eine Regel, welche normalerweise gilt, aber durch spezifischere Informationen wie „Pinguine fliegen nicht“ außer Kraft gesetzt werden kann. Das kumulative Schließen geht auf die Arbeit von Kraus, Lehmann und Magidor aus dem Jahr 1990 zurück, die das System C mit seinen fünf Regeln entwickelten. Eine Besonderheit der maschinellen Formalisierung mit Coq ist, dass sie absolute Präzision und Vollständigkeit garantiert, während handgeschriebene Beweise Fehler enthalten können, oder implizite Annahmen verwenden. Wir beginnen mit der syntaktischen Formalisierung der System C Regeln und gehen dann zur semantischen Modellierung kumulativer Modelle über. Dabei beschränken wir uns auf propositionale Logik, um die Komplexität überschaubar zu halten und uns auf die wesentlichen Aspekte des KLM-Theorems zu konzentrieren. Die Formalisierung und die Beweise bieten eine wiederverwendbare Basis für nichtmonotones Schließen und schaffen eine Grundlage für Erweiterungen in der Wissensrepräsentation.

Abstract

This thesis is dedicated to the formalization and proving of the representation theorem for cumulative reasoning using the proof assistant Coq. The theorem represents one of the central theorems of nonmonotonic logic. Nonmonotonic reasoning enables the revision of conclusions when new information becomes available and considers exceptions, which is closer to human thinking than classical monotonic logic. A typical example is "birds fly", a rule that normally holds but can be overridden by more specific information such as "penguins do not fly". Cumulative reasoning traces back to the work of Kraus, Lehmann and Magidor from 1990, who developed System C with its five rules. A distinctive feature of machine-verified formalization with Coq is that it guarantees absolute precision and completeness, while hand-written proofs may contain errors or use implicit assumptions. We begin with the syntactic formalization of the System C rules and then proceed to the semantic modeling of cumulative models. We restrict ourselves to propositional logic in order to keep the complexity manageable and to focus on the essential aspects of the KLM-Theorem. The formalization and proofs provide a reusable foundation for non-monotonic reasoning and create a basis for extensions in knowledge representation.

Inhaltsverzeichnis

1 Einleitung

Eine zentrale Form des logischen Schließens stellt das nichtmonotone Schließen dar. Dabei wird durch die Nichtmonotonie die Revision von Schlüssen ermöglicht, was grundlegend auch widerspiegelt, wie wir Menschen denken. Es besagt, dass wir eine einmal als wahr angenommene Schlussfolgerung beim Erhalt neuer Informationen wieder anpassen können. Zum Beispiel wissen wir, dass Vögel im Allgemeinen fliegen können. Gleichzeitig wissen wir aber auch, dass Pinguine Vögel sind, die nicht fliegen können. Dieser scheinbare Widerspruch lässt sich durch nichtmonotones Schließen auflösen: Die Regel „Vögel fliegen“ gilt, kann aber durch spezifischere Informationen, wie im Fall von den Pinguinen, außer Kraft gesetzt werden. Wie an diesem Beispiel gut zu sehen ist, können wir so Ausnahmen berücksichtigen. Anders als bei der klassischen Monotonie, wo das Hinzunehmen neuer Prämissen nicht die Schlussfolgerung invalidieren kann. So würde die Annahme „Vögel fliegen“ beim Hinzunehmen von „Pinguine sind Vögel“ auch zu „Pinguine fliegen“ führen, was offensichtlich nicht der Fall ist.

Nichtmonotones Schließen bietet die formale Grundlage, um diese Arten von Wissen, bei denen bestimmte Annahmen üblicherweise, aber nicht ausnahmslos gelten, in einem logischen System darzustellen. Das hierfür 1990 von Kraus, Lehmann und Magidor [11] entwickelte logische System nennt sich System C. Die darin enthaltenen Regeln stellen den Grundstein für nichtmonotones Schließen dar und ermöglichen es, logische Schlüsse unter Berücksichtigung möglicher Ausnahmen zu ziehen.

Um diesen Regeln eine semantische Bedeutung zu geben, wurden kumulative Modelle entwickelt. Diese Modelle stellen eine formale Struktur dar, welche aus Zuständen, einer Labeling-Funktion und einer Präferenzrelation besteht. Dabei repräsentieren die Zustände mögliche Interpretationen, die Labeling-Funktion verbindet Zustände mit Welten, und die Präferenzrelation ermöglicht es, „normalere“ oder „typischere“ Zustände zu finden. Diese semantische Grundlage ermöglicht uns damit, die syntaktischen Regeln des System C in einem modelltheoretischen Rahmen zu interpretieren.

Eines der grundlegenden Theoreme im Bereich der nichtmonotonen Logik ist das Repräsentationstheorem für kumulatives Schließen (KLM-Theorem), welches ebenfalls 1990 von Kraus, Lehmann und Magidor vorgestellt wurde. Das Theorem stellt eine Verknüpfung zwischen der syntaktischen Ebene der Regeln und Merkmalen von kumulativen Konsequenzrelationen und der semantischen Ebene dar, die durch kumulative Modelle beschrieben wird.

Das KLM-Theorem besagt, dass eine Konsequenzrelation genau dann kumulativ ist, wenn sie durch ein kumulatives Modell definiert werden kann. Diese Äquivalenz bildet die theoretische Basis für kumulative Logik und gewährleistet einerseits die korrekte Darstellung kumulativer Konsequenzrelationen sowohl durch syntaktische Beschreibungen, wie des System C, als auch durch eine konsistente semantische Interpretation. Andererseits stellt das Theorem dadurch sicher, dass jede Re-

lation, die durch ein kumulatives Modell definiert ist, den Regeln des System C entspricht.

Diese Arbeit widmet sich der Formalisierung des Theorems 3.25 und des semimaschinellen Beweisens des Repräsentationstheorems für kumulatives Schließen mit dem Beweisassistenten Coq [19]. Ein Beweisassistent wie Coq ermöglicht die vollständige formale Verifikation mathematischer Beweise durch eine Kombination aus menschlicher Führung und maschineller Überprüfung. Dabei beschränken wir uns vorrangig auf den Fall von kumulativem Schließen basierend auf propositionaler Logik, um die sprachliche Komplexität überschaubar zu halten und uns auf die wesentlichen Aspekte des Repräsentationstheorems zu konzentrieren.

Während handgeschriebene Beweise Fehler enthalten können, garantiert dann die maschinelle Überprüfung die Vollständigkeit und Korrektheit aller Beweisschritte. Dabei werden unter anderem implizite Annahmen, welche auch in informellen Beweisen vorkommen könnten, vermieden, was wiederum das Vertrauen in die Gültigkeit des Theorems erhöht. Neben der vollständigen Präzision und Explizitheit der Annahmen führt die Formalisierung auch zu der Wiederverwendbarkeit von formalisierten Definitionen und Lemmas zur weiteren Forschung. Dabei schafft dies auch eine solide Grundlage für Erweiterungen und verwandte Systeme.

Die Herausforderungen der Arbeit belaufen sich auf einige wesentliche Punkte. In erster Linie gilt es, die Smoothness-Bedinung korrekt und vollständig in Coq darzustellen. Dabei ist zu beachten, dass in der propositionalen Logik die Beweisstruktur endlich ist. Dies führt dazu, dass es bei n atomaren Formeln maximal $2^{(2^n)}$ semantisch unterschiedliche Formeln gibt und jede absteigende Kette muss irgendwann ein minimales Element haben. Mit diesem Wissen lässt sich dann die Präferenzrelation so strukturieren, dass sie automatisch die Smothness-Bedingung erfüllt, was den Beweis des Repräsentationstheorems vereinfachen sollte. Das wäre bei einer prädikatenlogischen Erweiterung komplexer, da wir hier unendlich viele semantisch unterschiedliche Formeln handhaben müssten. Eine weitere Herausforderung stellen die genaue Definition der kumulativen Modelle mit ihren Eigenschaften und die exakte Formalisierung der System C Regeln in Coq dar. Die mathematischen Definitionen müssen in die strenge Syntax von Coq übersetzt werden, wobei es oft mehrere Möglichkeiten gibt, diese mathematischen Konzepte zu kodieren. Dabei ist zu beachten, dass alle impliziten Annahmen aus einer mathematischen Definition in Coq explizit gemacht werden müssen. Zudem müssen die Definitionen so gestaltet sein, dass sie später im Beweis effektiv verwendet werden können. Das bedeutet konkret, dass wir schon früh darüber eine Entscheidung treffen müssen, wie Konsequenzrelationen, Zustände, Labeling-Funktion und Präferenzrelation repräsentiert werden. Letztlich gilt es damit dann die komplexen Beweisschritte im Repräsentationstheorem zu bewältigen.

Damit beläuft sich das Hauptziel dieser Arbeit auf die Frage: „Wie kann das KLM-Theorem präzise in Coq formalisiert werden?“ Mitunter soll dann auch gezeigt werden, wie kumulative Modelle formal dargestellt werden können, wie sich das System C in Coq kodieren lässt und wie die Äquivalenz zwischen beiden nachgewie-

sen werden kann. Dafür werden konkret die Formalisierung der Grundbegriffe, die Konstruktion des Beweises des KLM-Theorems und die Entwicklung wiederverwendbarer Definitionen für nichtmonotones Schließen behandelt.

Zunächst werden wir uns auf die theoretischen Grundlagen von propositionaler Logik und nichtmonotonem Schließen konzentrieren. Danach wird Coq als Beweisassistent eingeführt und beschrieben. Anschließend werden wir den geplanten Formalisierungsansatz benennen und konkrete Implementierungsdetails nennen. Danach widmen wir uns dem Beweis des Repräsentationstheorems. Schlussendlich werden wir die Ergebnisse des Formalisierungsversuches besprechen und bewerten, bevor wir mit Schlussfolgerungen und einem Ausblick auf mögliche Erweiterungen der Formalisierung die Arbeit abschließen.

2 Theoretische Grundlagen

Im Folgenden werden wir zunächst die theoretischen Grundlagen dieser Arbeit benennen und verdeutlichen. Dabei werden wir zunächst auf propositionale Logik und deren Bedeutung in dem Kontext eingehen, bevor wir uns den Eigenschaften und der Bedeutung der nichtmonotonen Logik widmen.

2.1 Propositionale Logik

In der propositionalen Logik beschäftigen wir uns mit Aussagen, die entweder wahr $\top$ oder falsch $\bot$ sein können, nicht beides und nicht keines von beiden. In der Sprache $\mathcal{L}$ definiert die Menge der Aussagenvariablen das Alphabet Σ, welches je nach formaler Definition endlich oder abzählbar unendlich ist. Die Aussagenvariablen sind dabei atomare Aussagen die, nicht weiter zerlegt werden können. Beispiele für atomare Aussagen sind „Es regnet“ oder „Die Sonne scheint“. Diese werden durch Variablen wie p, q, r repräsentiert. Formeln der propositionalen Logik werden durch die Nutzung logischer Konnektive, wie Konjunktion ($\wedge$), Disjunktion ($\vee$), Negation ($\neg$), Implikation ($\rightarrow$) und die Äquivalenz ($\leftrightarrow$), rekursiv aufgebaut.

Die Syntax der propositionalen Logik beruht auf der induktiven Definition der Menge der wohlgeformten Formeln (*wff*) [1].

Diese besagt,

- dass jede atomare Aussage p eine Formel ist,
- wenn p eine Formel ist, dann ist auch deren Negation $\neg p$ eine Formel,
- wenn p und q Formeln sind dann sind auch deren Konjunktion $p \wedge q$, Disjunktion $p \vee q$, Implikation $p \rightarrow q$ und Äquivalenz $p \leftrightarrow q$ eine Formel,
- und nur Ausdrücke, die durch das endliche Anwenden dieser Regeln gebildet werden, sind Formeln.

Um Mehrdeutigkeiten bei komplexeren Formeln zu vermeiden, können außerdem Klammern eingesetzt werden. Um jedoch eine Klammerung zu reduzieren, gibt es die Präzedenzregeln, die die Stärke der Bindung von den jeweiligen Konnektiven festlegt, das heißt, sie geben an, welches Konnektiv in einer Formel zuerst ausgewertet wird. So bindet $\neg$ stärker als $\vee$ und $\wedge$ und diese wiederum stärker als $\rightarrow$ und $\leftrightarrow$.

Die Semantik von Formeln in der propositionalen Logik gibt eine Interpretation an, die jeder atomaren Aussage einen Wahrheitswert zuordnet.

Für eine Interpretation v in der Sprache $\mathcal{L}$ gilt daher:

$$v : \Sigma \rightarrow \{\top, \bot\}$$

Damit die Interpretation v auch Wahrheitswerte für komplexe Formeln liefern kann, erweitern wir sie induktiv wie folgt:

$$\begin{aligned}
v(\neg p) &= \top \text{ genau dann, wenn } v(p) = \bot \\
v(p \wedge q) &= \top \text{ genau dann, wenn } v(p) = \top \text{ und } v(q) = \top \\
v(p \vee q) &= \top \text{ genau dann, wenn } v(p) = \top \text{ oder } v(q) = \top \\
v(p \rightarrow q) &= \top \text{ genau dann, wenn } v(p) = \bot \text{ oder } v(q) = \top \\
v(p \leftrightarrow q) &= \top \text{ genau dann, wenn } v(p) = v(q)
\end{aligned}$$

Dabei ist die Interpretation v ein Modell für eine Formel p, geschrieben $v \models p$ wenn $v(p) = \top$ gilt.

Dies bedeutet ebenfalls, dass eine Formel erfüllbar ist, wenn diese mindestens ein Modell hat. Zudem können wir noch Aussagen zu der Tautologie und Kontradiktion treffen. Eine Formel ist logisch gültig (Tautologie), wenn jede Interpretation ein Modell ist und eine Formel ist eine Kontradiktion, wenn es kein Modell gibt.

Folgend werden wir noch zwei weitere wichtige Eigenschaften der propositionalen Logik, die logische Äquivalenz und logische Folgerung, ergänzen

Die logische Äquivalenz besagt, dass zwei Formeln p und q logisch äquivalent sind $p \equiv q$, wenn sie für jede mögliche Interpretation der Variablen den gleichen Wahrheitswert haben, also v, $v(p) = v(q)$ für jede Interpretation v gilt. Weitere logische Äquivalenzen der propositionalen Logik sind dabei:

Kommutativität:	$p \wedge q \equiv q \wedge p,$	$p \vee q \equiv q \vee p$
Assoziativität:	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r),$	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributivität:	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r),$	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
De Morgan:	$\neg(p \wedge q) \equiv \neg p \vee \neg q,$	$\neg(p \vee q) \equiv \neg p \wedge \neg q$
Implikation:	$p \rightarrow q \equiv \neg p \vee q$	
Kontraposition:	$p \rightarrow q \equiv \neg q \rightarrow \neg p$	

Für die logische Folgerung gilt, dass eine Formel q logisch aus einer anderen Formel p, geschrieben $p \models q$, folgt, wenn für jede Interpretation v die Wahrheit der Formel p auch die Wahrheit der Formel q garantiert: $v(p) = \top$, dann auch $v(q) = \top$.

Die logische Folgerung können wir ebenfalls noch auf Formelmengen erweitern. Dabei folgt eine Formel p aus einer Menge von Formeln Γ, geschrieben $\Gamma \models p$, wenn für jede Interpretation v die Formeln in Γ die Wahrheit von p gewährleisten: $v(\gamma) = \top$ für alle $\gamma \in \Gamma$, dann auch $v(p) = \top$.

2.2 Nichtmonotones Schließen

Wie schon anfangs erwähnt handelt es sich bei nichtmonotonem Schließen um eine spezielle Form des logischen Schließens, welche es ermöglicht bei dem Hinzunehmen neuer Informationen, schon bereits getroffene Schlussfolgerungen zu revidieren.

Im Folgenden werden wir, für diese speziellen Folgerungsbeziehungen, auch Konsequenzrelation genannt, das Symbol $\mid\!\sim$ nutzen. Damit grenzen wir die nichtmonotone Konsequenzrelation von der klassischen logischen Folgerung $\models$ ab.

Formal bedeutet dies also, dass eine Konsequenzrelation $\mid\!\sim$ nichtmonoton ist, wenn es Formeln a, b, c gibt, für die $a \mid\!\sim c$ aber $a \wedge b \not\mid\!\sim c$. Dies steht in dem Gegensatz zur klassischen Logik, wo durch das Hinzufügen weiterer Prämissen nie weniger folgen kann. Für Beispiel können wir uns wieder an dem bereits kurz vorgestellten Vogel-Pinguin Beispiel orientieren.

Nehmen wir an, dass wir aus „Tweety ist ein Vogel“ folgern, „Tweety kann fliegen“. Wenn wir dann lernen „Tweety ist ein Pinguin“, müssen wir die vorher getroffene Schlussfolgerung, nämlich dass Tweety fliegen kann, zurückziehen. Dies spiegelt unsere menschliche Art zu denken wider, da wir ständig unter neuem Wissen Schlüsse ziehen und anpassen müssen. Gleichzeitig ermöglicht die Nichtmonotonie auch, dass wir mit Ausnahmen und Standards, die „normalerweise“ oder „üblicherweise“ gelten, umgehen können, im Vergleich zu der klassischen Monotonie, welche nur mit strikten Regeln arbeiten kann. Darum wird $a \mid\!\sim b$ auch als b folgt *üblicherweise* aus a gelesen. Wir behandeln bei dem nichtmonotonen Schließen also den Umgang mit unvollständigem Wissen, was es uns ermöglicht, vorläufige Schlüsse zu ziehen, die dann später revidiert werden können.

Es kann aber folglich auch zu Konflikten durch solche Ausnahmen kommen. Um dies genauer darzustellen, möchten wir hier auf das Beispiel der *Nixon-Raute* [17] eingehen. Präsident Nixon ist sowohl Quäker als auch Republikaner. Quäker sind normalerweise pazifistisch, wohingegen Republikaner normalerweise keine Pazifisten sind.

Beispiel 1. *Die Nixon-Raute*

Fakten:

$Quäker(nixon)$

$Republikaner(nixon)$

Nichtmonotone Regeln:

$Quäker(x) \mid\sim Pazifist(x)$	(Quäker sind normalerweise Pazifisten)
$Republikaner(x) \mid\sim \neg Pazifist(x)$	(Republikaner sind normalerweise keine Pazifisten)

Wie zu sehen ist, können wir nun zwei widersprüchliche, nichtmonotone Schlussfolgerungen ableiten. Nämlich einerseits, dass aus *Nixon ist ein Quäker* normalerweise folgt *Nixon ist ein Pazifist* und andererseits aus *Nixon ist ein Republikaner* normalerweise folgt *Nixon ist kein Pazifist*:

$$Quäker(nixon) \mid\sim Pazifist(nixon)$$
$$Republikaner(nixon) \mid\sim \neg Pazifist(nixon)$$

Dieses Beispiel veranschaulicht gut, dass für solch einen Konflikt noch zusätzliche Mechanismen benötigt werden, um besser mit diesem Umgehen zu können. Dafür gibt es einige Konfliktlösungsstrategien, die wir an dieser Stelle kurz vorstellen wollen. Es könnte gegebenenfalls mit einer Art Spezifitätsprinzip gearbeitet werden, wo spezifischere Informationen dann den Vorrang haben. Dies wäre aber bei dem obigen Beispiel nur schwer anwendbar, da wir hier nur zwei gleichwertige Informationen haben. Bei unserem Vogel-Pinguin Beispiel könnte man aber annehmen, dass „Pinguine fliegen nicht“ eine spezifischere Information ist als „Vögel fliegen“ wodurch diese nicht übertrumpft werden würde. Es gäbe aber auch die Möglichkeit, nur nicht-kontroverse Schlussfolgerungen zu akzeptieren oder generell Prioritätsordnungen vorzunehmen.

Bevor solche Lösungsansätze für Konflikte entwickelt werden können, müssen erst einmal die grundlegenden Eigenschaften des nichtmonotonen Schließen festgelegt werden. Dafür stellen Kraus, Lehmann und Magidor *nichtmonotone Schlussfolgerungssysteme* [11] vor. Diese bestimmen, wie sich nichtmonotone Schlussfolgerungen verhalten sollen und definieren dafür formale Bedingungen für „vernünftiges nichtmonotones Schließen“ [11]. Dies stellt sicher, dass die Schlussfolgerungen unter konsistenten neuen Informationen stabil bleiben. Im Folgenden werden wir das kumulative Schlussfolgerungssystem *System C* vorstellen.

2.2.1 System C

Das System C[1] stellt nach Kraus, Lehmann und Magidor [11] die minimalen Anforderungen an rationales nichtmonotones Schließen dar. Es beinhaltet die folgenden grundlegenden Regeln:

Definition 1 (System C)**.**

$$\textit{(Ref)} \quad a \mid\sim a \qquad \text{(Reflexivity)}$$

$$\textit{(LLE)} \quad \frac{\models a \leftrightarrow b \quad a \mid\sim c}{b \mid\sim c} \qquad \text{(Left Logical Equivalence)}$$

$$\textit{(RW)} \quad \frac{\models a \rightarrow b, \quad c \mid\sim a}{c \mid\sim b} \qquad \text{(Right Weakening)}$$

$$\textit{(Cut)} \quad \frac{a \wedge b \mid\sim c, \quad a \mid\sim b}{a \mid\sim c} \qquad \text{(Transitivity)}$$

$$\textit{(CM)} \quad \frac{a \mid\sim b \quad a \mid\sim c}{a \wedge b \mid\sim c} \qquad \text{(Cautious Monotonicity)}$$

Jede dieser Regeln erfüllt einen spezifischen Zweck, um menschliche Denkprozesse intuitiv nachzuvollziehen. Um die Regeln zu veranschaulichen, werden wir dafür wieder auf das Vogel-Pinguin Beispiel zurückgreifen und es etwas ergänzen. Reflexivity etabliert dabei die Grundeigenschaft, dass jede Formel sich selber beinhaltet. Das heißt, für unser Beispiel gilt „Vögel sind normalerweise Vögel“, was offensichtlich der Fall sein sollte. Left Logical Equivalence hingegen stellt sicher, dass logisch äquivalente Formeln in den Prämissen ausgetauscht werden können. Wenn wir also „Flugunfähige Pinguine sind normalerweise Schwimmer“ und „Pinguine, die nicht fliegen können, sind das gleiche wie Flugunfähige Pinguine“ annehmen, können wir daraus schlussfolgern, dass „Pinguine, die nicht fliegen können, sind normalerweise Schwimmer“ gilt. Hieran ist gut zu erkennen, dass nur der Inhalt der Prämisse eine Rolle spielt und nicht ihre syntaktische Form. Dies ermöglicht also auch Umformulierungen oder andere Beschreibungen, ohne dabei die Bedeutung einer Prämisse zu ändern. Right Weakening erlaubt es uns, schwächere Schlussfolgerungen von stärkeren Schlussfolgerungen ableiten zu können. Das beutet, wenn wir zum Beispiel „Vögel haben normalerweise Flügel“ und „Flügel haben Federn“ annehmen, dann folgt daraus auch „Vögel haben normalerweise Federn“. Dadurch wird ermöglicht, Schlussfolgerungen aus allgemeineren Eigenschaften zu schließen. Die Cut Regel stellt eine Art von Transitivität dar, was wiederum relevant ist, um zwei separate Schlussfolgerungen zu einer einzigen zu verknüpfen. So schließen wir aus dem Beispiel „Vögel können normalerweise fliegen“ und „Vögel die fliegen, können

[1] Das „C“ steht hier für *Cumulative*

landen“ auch dass „Vögel können normalerweise landen“ gilt. Damit ist es möglich, Zwischenschritte in Schlussfolgerungen zu überspringen, um so Wiederholungen zu vermeiden. Und letztlich stellt Cautious Monotonicity eine eingeschränkte Form der Monotonie dar. Dabei werden die Einschränkungen der klassischen Monotonie, dass zusätzliche Prämissen alle bisherigen Schlussfolgerungen enthalten, vermieden. Dennoch wird gewährleistet, dass die Schlussfolgerungen konsistent bleiben. Eine eingeschränkte Form der Monotonie bedeutet dabei, dass nur dann eine neue Prämisse hinzugefügt werden kann, wenn wir für beide Schlussfolgerungen bereits unabhängig festgestellt haben, dass diese gültig sind. Beispielhaft können wir also annehmen, dass „Vögel normalerweise Federn haben“ und „Vögel normalerweise fliegen können“ und damit dann „Vögel, die Federn haben, können normalerweise fliegen“ folgern. Hierbei bleibt also die typische Eigenschaft, dass Vögel fliegen können, erhalten, auch wenn die weitere typische Eigenschaft von Vögeln, dass diese Federn haben, hinzugenommen wird.

Wenn es nun eine Konsequenzrelation $\mid\!\sim$ gibt, die alle Regeln des System C erfüllt, dann bezeichnen wir diese als eine *kumulative Konsequenzrelation*. Dabei ist eine zentrale Eigenschaft kumulativer Konsequenzrelationen, dass diese eine zugrundeliegende Logik erfordern, die festlegt, welche Formeln a, b, c in den Regeln verwendet werden dürfen. In dieser Arbeit wurde die propositionale Logik als zugrundeliegende Logik gewählt, da sie einfach zu formalisieren ist und eine gut etablierte Grundlage für die Anwendung kumulativer Regeln bietet.

Es bleiben jedoch die Grenzen der Ausdrucksstärke. Wenn wir „Vögel $\mid\!\sim$ fliegen“ und „Pinguine $\rightarrow$ Vögel“ annehmen, erlaubt System C nicht automatisch, „Pinguine $\mid\!\sim$ nicht fliegen“ abzuleiten. Das bedeutet, System C alleine kann nicht direkt mit Ausnahmen umgehen.

2.3 Kumulative Modelle

Folgend zeigen wir eine Methode, um das Problem der Ausnahmen für „normalere“ oder „typischere“ Zustände zu minimieren. Dafür stellen wir zunächst *kumulative Modelle* vor und definieren diese formal, um dann das Repräsentationstheorem für kumulative Konsequenzrelationen (*KLM-Theorem*) zu präsentieren.

Definition 2 (Kumulatives Modell)**.** *Ein kumulatives Modell wird formal durch ein Tripel $\mathcal{W} = \langle S, l, \prec \rangle$, beschrieben, wobei gilt:*

- *S ist eine Menge, deren Elemente als Zustände bezeichnet werden.*
- *$l : S \rightarrow 2^W$ ist die Labeling-Funktion, die jedem Zustand eine nicht-leere Menge von Welten zuordnet.*
- *$\prec$ ist die binäre Präferenzrelation auf S, die die Smoothness Condition erfüllt:*
  - *Für alle Formeln a ist die Menge $\hat{a} = \{s \mid s \in S, s \models a\}$ smooth.*

Die Zustände S in einem kumulativen Modell bilden dabei die Grundlage der semantischen Struktur. Jeder Zustand repräsentiert eine mögliche Weltbeschreibung oder Wissenskontexte, wobei dabei zwischen typischen und außergewöhnlichen Fällen unterschieden wird. Zum Beispiel kann ein Zustand die Information „Vogel“, und ein anderer Zustand „Pinguin“ repräsentieren.

Die Labeling-Funktion l ordnet jedem Zustand S eine nicht-leere Menge von Welten zu. Eine Welt ist hier eine Interpretation, welche jeder atomaren Formel einen Wahrheitswert zuweist. Das bedeutet, diese definiert, welche Propositionen in welchem Zustand wahr sind. So würde ein Zustand „Taube“ mit „kann fliegen“ verbunden werden, während „Pinguin“ mit „kann nicht fliegen“ verbunden wird.

Die Präferenzrelation $\prec$ erfasst dabei die Intuition darüber, welche Zustände als „normal“ oder „typisch“ gelten. Das bedeutet, wenn $s \prec t$ gilt, wird der Zustand s als normaler oder typischer angesehen als der Zustand t. Diese Relation ist irreflexiv, das heißt, kein Zustand steht in Relation mit sich selbst. Konkret würde „ein Vogel, der fliegen kann“ normalerweise höher eingestuft als „ein Vogel, der ein Pinguin ist“, da Letzteres eher eine Ausnahme darstellt. Dadurch wird sichergestellt, dass Standard-Schlussfolgerungen bevorzugt werden, sofern diese nicht durch anderweitige explizite Informationen überdeckt werden.

Eine wichtige Eigenschaft der kumulativen Modelle ist die *Minimalität* der Präferenzrelation. Dabei bezeichnen wir die minimalen Elemente einer Menge von Zuständen, für die es keinen noch typischeren Zustand in der Menge gibt:

Definition 3 (Minimale Elemente)**.** *Sei $A \subseteq S$ eine Menge von Zuständen und $\prec$ eine Präferenzrelation auf S. Die Menge der minimalen Elemente von A bezüglich $\prec$ ist definiert als:*

$$min_{\prec}(A) = \{s \in A \mid \nexists t \in A : t \prec s\}$$

Für eine Formel a definieren wir die Menge $\hat{a}$ als die Menge aller Zustände, in denen a gilt:

$$\hat{a} = \{s \mid s \in S, s \models\!\!\!\models a\}$$

wobei $s \models\!\!\!\models a$ bedeutet, dass die Formel a im Zustand s erfüllt ist. Formal ist $s \models\!\!\!\models a$ genau dann, wenn für alle Welten $w \in l(s)$ gilt: $w \models a$.

2.3.1 Smoothness-Bedingung

Die Smoothness-Bedingung stellt eine weitere wichtige Eigenschaft der Präferenzrelation in den kumulativen Modellen dar. Diese garantiert, dass für jede Formelmenge, in welcher es mindestens einen Zustand gibt, auch minimale Elemente bezüglich der Präferenzrelation existieren. Das bedeutet, dass wir für jede Menge von Zuständen immer die Typischsten finden können. Dies entspricht auch wieder der menschlichen Tendenz, zuerst die typischsten Fälle zu betrachten. Wir geben zunächst an, wann eine beliebige Menge P als smooth bezüglich einer Relation $\prec$ betrachtet wird.

Definition 4 (Smoothness-Eigenschaft)**.** *Sei $P \subseteq U$ eine Menge und $\prec$ eine binäre Relation auf U. Die Menge P heißt* smooth *bezüglich $\prec$ genau dann, wenn für alle $t \in P$ entweder $t \in min_{\prec}(P)$ oder es existiert ein $s \in min_{\prec}(P)$ mit $s \prec t$ gilt.*

Die intuitive Bedeutung der Smoothness-Eigenschaft ist, dass jedes Element entweder selber minimal ist oder einem minimalen Element untergeordnet ist. Dies verhindert in der Präferenzrelation problematische Strukturen, wie unendlich absteigende Ketten. In kumulativen Modellen betrachten wir speziell die Mengen der Form $\hat{a}$, also die Menge aller Zustände, in denen eine Formel a gilt. Dafür definieren wir die Smoothness-Bedingung. Ohne diese Bedingung könnte der Fall eintreten, dass es keine minimalen Modelle für eine Formel geben könnte, was dann wiederum die semantische Interpretation der Präferenzrelation unmöglich machen würde.

Definition 5 (Smoothness-Bedingung)**.** *Sei L die Menge aller Formeln in der Sprache der propositionalen Logik. Ein Triple $(S, l, \prec)$ erfüllt die Smoothness-Bedingung, wenn für jede Formel $a \in L$ die Menge $\hat{a}$ smooth ist.*

Im Kontext der propositionalen Logik ist hervorzuheben, dass bei einer endlichen Anzahl von atomaren Formeln die Smoothness-Bedingung leichter zu handhaben ist als in allgemeineren logischen Systemen, da es bei n atomaren Formeln maximal $2^{(2^n)}$ semantisch unterschiedliche Formeln gibt. Dadurch ist auch jede absteigende Kette in der Präferenzrelation endlich, was wiederum automatisch die Existenz minimaler Elemente garantiert und damit einen wichtigen Punkt für die spätere Formalisierung in Coq darstellt. Dafür berufen wir uns auf die Arbeit von Lehmann und Magidor (1992) [12] in welcher sie klarstellen:

> *The smoothness condition is only a technical condition. It is satisfied in any well-founded preferential model, and, in particular, in any finite model. When the language $\mathcal{L}$ is logically finite, we could have limited ourselves to finite models and forgotten the smoothness condition.*

2.3.2 Konsequenzrelation in kumulativen Modellen

Basierend auf den bisherigen Definitionen können wir nun die Konsequenzrelation definieren, die durch ein kumulatives Modell bestimmt wird.

Definition 6 (Modellbasierte Konsequenzrelation)**.** *Sei $\mathcal{W} = \langle S, l, \prec \rangle$ ein kumulatives Modell. Die durch $\mathcal{W}$ induzierte Konsequenzrelation $\mid\!\sim_{\mathcal{W}}$ ist definiert als:*

$$a \mid\!\sim_{\mathcal{W}} b \quad \textit{genau dann, wenn} \quad \forall s \in min_{\prec}(\hat{a}) : s \models b$$

Diese Definition besagt, dass b normalerweise aus a folgt, wenn b in allen minimalen Zuständen von a gilt. Wir beschränken uns hier auf die minimalen Zustände von a, da diese die typischsten Fälle repräsentieren wo a gilt. In unserem Vogel-Pinguin Beispiel würde die Aussage „Vögel können normalerweise fliegen“ semantisch dadurch repräsentiert, dass die Eigenschaft „können fliegen“ in den minimalsten Zuständen gilt, welche Vögel beschreiben. Im Gegensatz dazu würde die

Ausnahme von den Pinguinen nicht in den minimalen Zuständen enthalten sein, wodurch die Gültigkeit der allgemeinen Regel nicht beeinflusst wird. Diese modellbasierte Definition der Konsequenzrelation entspricht der semantischen Interpretation der syntaktischen Regeln des System C und das Repräsentationstheorem von Kraus, Lehmann und Magidor stellt genau diese Verbindung formal her.

Um zu illustrieren, wie ein kumulatives Modell konstruiert werden kann, greifen wir dafür auf das vorherige Beispiel 1 zurück und stellen dafür die benötigten Zustände, die Labeling-Funktionen und die Präferenzrelationen dar.

Beispiel 2. *Kumulatives Modell für die Nixon-Raute*
Wir konstruieren ein kumulatives Modell $\mathcal{W} = \langle S, l, \prec \rangle$ für die Nixon-Raute wie folgt:

1. ***Zustände (S):*** $S = \{s_1, s_2, s_3, s_4, s_5\}$
2. ***Labeling-Funktion (*l*)**, die jedem Zustand eine Menge von Welten zuordnet:*
   - $l(s_1) = \{w_1\}$,
     wobei in w_1 *gilt:* $Quäker(nixon)$, $Republikaner(nixon)$,$Pazifist(nixon)$
   - $l(s_2) = \{w_2\}$,
     wobei in w_2 *gilt:* $Quäker(nixon)$, $Republikaner(nixon)$, $\neg Pazifist(nixon)$
   - $l(s_3) = \{w_3\}$,
     wobei in w_3 *gilt:* $Quäker(nixon)$, $\neg Republikaner(nixon)$, $Pazifist(nixon)$
   - $l(s_4) = \{w_4\}$,
     wobei in w_4 *gilt:* $\neg Quäker(nixon)$, $Republikaner(nixon)$, $\neg Pazifist(nixon)$
   - $l(s_5) = \{w_5\}$,
     wobei in w_5 *gilt:* $\neg Quäker(nixon)$, $\neg Republikaner(nixon)$, $Pazifist(nixon)$
3. ***Präferenzrelation (*$\prec$*)**, die typischere Zustände kennzeichnet:*
   - $s_3 \prec s_1$ *(Ein Quäker, der kein Republikaner ist, ist typischer als ein Quäker, der Republikaner ist)*
   - $s_4 \prec s_2$ *(Ein Republikaner, der kein Quäker ist, ist typischer als ein Republikaner, der Quäker ist)*

Betrachten wir nun einige Mengen von Zuständen:

- *Für die Formel* $a = Quäker(nixon)$ *gilt:*
  - $\hat{a} = \{s_1, s_2, s_3\}$ *(alle Zustände, in denen Nixon ein Quäker ist)*
  - $min_{\prec}(\hat{a}) = \{s_3\}$ *(der typischste Zustand, in dem Nixon ein Quäker ist)*
- *Für die Formel* $b = Republikaner(nixon)$ *gilt:*
  - $\hat{b} = \{s_1, s_2, s_4\}$ *(alle Zustände, in denen Nixon ein Republikaner ist)*
  - $min_{\prec}(\hat{b}) = \{s_4\}$ *(der typischste Zustand, in dem Nixon ein Republikaner ist)*

Schlussfolgerungen gemäß der Definition der modellbasierten Konsequenzrelation:

1. $Quäker(nixon) \mid\!\sim_{\mathcal{W}} Pazifist(nixon)$, *denn:*
   $s_3 \in min_{\prec}(\widehat{Quäker(nixon)})$ *und* $s_3 \models Pazifist(nixon)$

2. $Republikaner(nixon) \mid\!\sim_{\mathcal{W}} \neg Pazifist(nixon)$, *denn:*
   $s_4 \in min_{\prec}(\widehat{Republikaner(nixon)})$ *und* $s_4 \models \neg Pazifist(nixon)$

3. $Quäker(nixon) \wedge Republikaner(nixon)$ *hat keine eindeutige Schlussfolgerung bezüglich* $Pazifist(nixon)$, *denn:*
   - $Quäker(nixon) \wedge \widehat{Republikaner(nixon)} = \{s_1, s_2\}$
   - $min_{\prec}(Quäker(nixon) \wedge \widehat{Republikaner(nixon)}) = \{s_1, s_2\}$
   - *Da* $s_1 \models Pazifist(nixon)$ *aber* $s_2 \models \neg Pazifist(nixon)$, *gibt es keine einheitliche Schlussfolgerung.*

Beispiel 2 veranschaulicht, wie das kumulative Modell die Nixon-Raute formalisiert. Dabei gilt, dass typische Quäker Pazifisten (s_3), und typische Republikaner keine Pazifisten (s_4) sind. Ein Konflikt entsteht, wenn Nixon sowohl Quäker als auch Republikaner ist, da wir dann zwei minimale Zustände haben (s_1 und s_2), die zu unterschiedlichen Schlussfolgerungen führen.

2.4 KLM-Theorem zum Kumulativen Schließen

Nachdem wir die syntaktische Charakterisierung kumulativer Konsequenzrelationen durch System C und die semantische Charakterisierung durch kumulative Modelle eingeführt haben, kommen wir nun zum Repräsentationstheorem selbst.

Im Jahr 1990 wurde das Repräsentationstheorem für kumulative Konsequenzrelationen (KLM-Theorem) von Kraus, Lehmann und Magidor vorgestellt und bewiesen [11]. Es stellt eine Verbindung zwischen der syntaktischen und semantischen Ebene her.

Theorem 1 (KLM-Repräsentationstheorem für kumulatives Schließen)**.** *Eine Konsequenzrelation* $\mid\!\sim$ *ist genau dann eine kumulative Konsequenzrelation, wenn es ein kumulatives Modell* $\mathcal{W} = \langle S, l, \prec \rangle$ *gibt, sodass für alle Formeln* a *und* b

$$a \mid\!\sim b \quad \textit{genau dann, wenn} \quad a \mid\!\sim_{\mathcal{W}} b$$

gilt.

Die Kernaussage des KLM-Theorems bezieht sich dabei auf die Äquivalenz zwischen kumulativer Konsequenzrelationen und deren Repräsentierbarkeit durch kumulative Modelle. Dabei können wir das Theorem in zwei Richtungen aufteilen:

1. **Korrektheit (Soundness)**: Wenn $\mid\!\sim$ durch ein kumulatives Modell definiert ist, dann erfüllt $\mid\!\sim$ alle Regeln von System C. Das heißt, jede modellbasierte Konsequenzrelation ist kumulativ.

2. **Vollständigkeit (Completeness)**: Wenn $\mid\sim$ eine kumulative Konsequenzrelation ist, dann existiert ein kumulatives Modell $\mathcal{W}$, sodass $\mid\sim$ genau der durch $\mathcal{W}$ induzierten Konsequenzrelation entspricht.

Damit ist das Theorem zentral für nichtmonotones Schließen, da es zeigt, dass die Konsequenzrelation in einem kumulativen Modell nichtmonotones Schließen präzise darstellen kann. Dabei kann rationales nichtmonotones Schließen als Präferenz für typischere Situationen verstanden werden, und das Theorem zeigt, dass die System C Regeln genau die Regeln sind, die mathematisch aus der Präferenzrelation und der dadurch entstandenen Ordnung der Zustände folgen. Außerdem ergibt sich durch das KLM-Theorem noch eine praktische Anwendung, da es ermöglicht, zwischen syntaktischer und semantischer Ebene zu wechseln. Das bedeutet, wir können Konsequenzrelationen syntaktisch durch die Regeln von System C definieren und überprüfen, andererseits können wir sie aber auch semantisch durch kumulative Modelle interpretieren. Dies ist gerade für Beweise von Vorteil, da manche Eigenschaften auf der semantischen Ebene einfacher zu beweisen sind, wie zum Beispiel, wenn wir zeigen wollen, dass eine bestimmte Schlussfolgerung nicht gültig ist. Zudem lässt sich durch das Theorem die Konsistenz einer Wissensbasis mit nichtmonotonen Schlussregeln sehr gut nachweisen. Wenn wir zeigen können, dass ein kumulatives Modell existiert, welches diese Wissensbasis repräsentiert, folgt daraus direkt, dass die darin enthaltenen Schlussregeln mit den System-C-Regeln konsistent sind.

3 Coq als Beweisassistent

Im Folgenden werden wir den Beweisassistenten Coq vorstellen. Dabei gehen wir zunächst auf die interaktive Art der Beweise in Coq ein und stellen wesentliche Unterschiede zu händisch formulierten Beweisen und vollautomatisierten Beweisassistenten dar. Nach dieser Einführung werden wir uns den *Calculus of Inductive Constructions*, welcher die formale Sprache von Coq bildet, genauer ansehen. Zudem stellen wir danach die Spezifikationssprache von Coq *Gallina* vor, bevor wir konkret auf die Beweisführung anhand von *Taktiken* eingehen werden.

3.1 Interaktives Beweisen

Das Grundprinzip des interaktiven Beweisens besteht im Wesentlichen aus einem Zusammenspiel zwischen menschlicher Intuition und maschineller Präzision. Dabei kann der Beweisprozess als eine Art Dialog zwischen Mensch und Maschine gesehen werden. Mit Coq werden die Beweise dabei schrittweise konstruiert und nicht automatisch erzeugt. Das bedeutet, dass der Mensch die Beweisidee vorgibt und die Maschine dann die formale Korrektheit jedes Schrittes überprüft. Anders als bei vollautomatischen Beweisassistenten, wie *Z3* [7] von Microsoft und *Vampire* [10], wo der Beweisprozess vollständig automatisiert ist, und dadurch kein direkter Benutzereingriff vorgesehen ist. Es ist aber zu beachten, dass vollautomatische

Beweisassistenten meist auf einer *First Order Logic* basieren. Dadurch sind diese weniger ausdrucksstark, da keine Quantifikation über Funktionen oder Relationen erlaubt, und keine Mengen oder Typen direkt unterstützt sind. Coq, im Vergleich, arbeitet auf einer höheren *Typentheorien* Logik. Hervorzuheben ist, dass V3 zwar ebenfalls First Order Logic als Grundlage nutzt, jedoch mit zusätzlichen Theorien, wie Arithmetik und Mengen, arbeitet. Dennoch bleibt die Ausdrucksstärke schwächer als bei Typentheorien. Ein weiteres Manko, welches mit First oder Logic einhergeht, ist dessen Semi-Entscheidbarkeit. Wie wir wissen, gibt es keinen Algorithmus, welcher entscheidet, ob eine beliebige Formel der First Order Logic beweisbar ist. Dadurch arbeiten diese Theorembeweiser für eine automatisierte Theorembewertung oft mit Heuristiken, wie zum Beispiel bei Instanziierung von Variablen. Für einen Ausdruck „Für alle x" ($\forall x$) wählt der Beweisassistent dann zuerst einige „sinnvolle" Werte aus, welche zu einem schnellen Widerspruch führen könnten. Das bedeutet auch, dass keine vollständige Suche möglich ist und bei dem Anwenden von Regeln dann „geraten" wird, welche am besten anzuwenden sind. In Coq müssen Allquantoren explizit behandelt werden, wobei dann wiederum der Benutzer über die Instanziierung entscheidet. Zudem sind vollautomatische Beweisassistenten oft wenig konstruktiv in ihrer Ausgabe. Sie geben meistens nur „Ja, ist beweisbar" oder „Nein, ist nicht beweisbar" aus und brechen den Beweis nach keinem weiteren Fortschritt ab, wodurch der eigentliche Beweis nicht vollständig nachvollziehbar und schwer überprüfbar ist, also eine Art Black-Box Beweis. Vergleichsweise erzwingt ein interaktiver Beweisassistent wie Coq eine vollständige Beweisüberprüfung und eine explizite Angabe der Beweisschritte. Dies macht interaktive Beweisassistenten auch deutlich modularer im Vergleich zu den vollautomatisierten Beweisassistenten, da hier der Beweis in Blöcken aufgebaut werden kann und zum Beispiel verschiedene Lemmas definiert werden, welche später wiederverwendet werden können. Bei vollautomatischen Beweisassistenten ist der Benutzer ebenfalls oft auf bestimmte Domänen beschränkt. So eignet sich zum Beispiel Z3 besonders für *Satisfiability Modulo Theories*, um zu entscheiden, ob eine logische Formel erfüllbar ist, und Vampire wiederum für klassische First Order Logic Theoreme. Dabei sind diese sehr spezifisch auf eine Domäne optimiert und können nicht auf anderen Domänen arbeiten. Hingegen bleibt Coq damit sehr flexibel, da die Domäne von dem Benutzer formuliert werden kann, was aber wiederum je nach Domäne auch sehr komplex und aufwendig ist.

Insgesamt lässt sich erkennen, dass vollautomatische Beweisassistenten leichter zu nutzen sind. Beide Beweisassistenten erfordern eine gewisse Einarbeitung des Benutzers, die bei den vollautomatischen Beweisassistenten weniger Aufwand erfordert, da der Benutzer nur verstehen muss, wie eine Formel korrekt formuliert werden kann. Danach erfolgt der Beweisprozess vollautomatisch. In einem interaktiven Beweisassistenten wie Coq muss jeder Beweisschritt explizit angegeben werden, was insgesamt viel mehr Verständnis von Coq erfordert, um korrekt damit arbeiten zu können. Es ist folglich viel mehr Handarbeit, aber dafür ist ein Beweis vollständig nachvollziehbar. Vollautomatische Beweisassistenten sind zudem oft unvor-

hersehbar. Manche Formeln können in Sekunden gelöst werden, ähnliche Formeln können in Stunden nicht gelöst werden und Beweise sind nicht immer reproduzierbar. Sollte ein Beweis dann auch noch tatsächlich fehlschlagen, ist es eventuell nicht möglich zu verstehen, warum genau der Beweis fehlschlägt. Da dadurch vollautomatische Beweisassistenten ebenfalls oft nicht mit komplexeren Beweisen umgehen können oder diese gar lösen können, eignen sich interaktive Beweisassistenten wie Coq deutlich mehr für hochkomplexe und aufwendige Beweise.

3.2 Calculus of Inductive Constructions

Nun möchten wir die formale Sprache von Coq vorstellen. Der Calculus of Inductive Constructions (CIC) definiert neben der grundlegenden Syntax (Typen, Terme und Propositionen) ebenfalls Typisierungsregeln, Reduktionsregeln und Regeln für induktive Definitionen. Coq stellt dem Benutzer eine benutzerfreundliche Oberfläche und Syntax bereit, um dort Beweise formulieren zu können. Diese Eingabe wird letztlich in CIC-Ausdrücke übersetzt, welche danach, gemäß den formalen Regeln des CIC, geprüft werden, ob diese wohlgeformt sind.

Der Ursprung von CIC findet sich in dem einfach typisierten *λ-Kalkül* aus 1940 von Church [4]. Es führt eine einfache Typenhierarchie ein. Aufbauend darauf wurde das polymorphe λ-Kalkül [24] von Girard und Reynolds vorgestellt, welches das einfach typisierte λ-Kalkül um einen Polymorphismus erweitert. Dieser ermöglicht, Typen zu quantifizieren und parametrische Funktionen definieren zu können. Eine weitere Erweiterung dieser stellt der *Calculus of Constructions* von Coquand und Huet (1986) [5] dar. Hierbei wird das polymorphe λ-Kalkül um abhängige Typen erweitert. Dies ermöglicht folglich das Formulieren von komplexeren logischen Aussagen. Schlussendlich wird der Calculus of Inductive Constructions von Coquand und Paulin-Mohring (1990er) [15] eingeführt, welcher es zusätzlich zu dem Calculus of Constructions ermöglicht, induktive Datentypen direkt zu definieren. Dabei behält CIC aber nach wie vor die Kernkonzepte des einfach typisierten λ-Kalküls bei. Um den CIC formal definieren zu können, wird das Konzept der *Pure Type Systems* (PTS) [18] verwendet. Dieses bietet einen einheitlichen Rahmen zur Definition von verschiedenen Typensystemen und definiert dafür ein typisiertes λ-Kalkül mit einer gemeinsamen Syntax für Terme und Typen. Dabei erfolgt die Spezifikation des PTS über drei wesentliche Komponenten:

- Eine Menge von Sorts[2], zum Beispiel `Prop`, `Set`, $\texttt{Type}_1$ und $\texttt{Type}_2$. Sorts stellen die höchste Ebene der Typenhierarchie dar und werden als Klassifizierer für Typen und nicht als Objekte betrachtet.

- Axiome, welche wiederum die Typen von Sorts festlegen.
  So wäre `Prop` : $\texttt{Type}_1$ und `Set` : $\texttt{Type}_2$. Dabei werden die Sorts selber wiederum durch andere Sorts typisiert, um eine Hierarchie bilden zu können.

[2] englisch für „Sorten“

Durch diese Hierarchie wird sichergestellt, dass Axiome wie `Type : Type`, welche nicht zulässig wären, vermieden werden.

- Regeln, um zu bestimmen, welche Produkte gebildet werden können. Diese Regeln werden durch ein Tripel von Sorts ausgedrückt. Als Beispiel betrachten wir die Regel `(Prop, Prop, Prop)`. Diese besagt, dass wenn `A : Prop` und `B : Prop`, dann ist auch `A`→`B` von der Sort `Prop`.

Der Einfachheit halber werden wir mit Hinblick auf die Coq-Terminologie auch von „Typ Prop“ sprechen, obwohl es sich technisch um eine Sort handelt und betrachten damit `Prop` als einen Typ von Typen.

Durch das PTS kann bewiesen werden, dass im CIC keine Widersprüche abgeleitet werden können. So ist es nicht möglich, sowohl einen Satz als auch dessen Negation zu beweisen. Wie Paulin-Mohring erläutert *„Consistency can be derived as a consequence because they cannot be a proof without hypothesis of $\bot$ which has no constructor.“ [15]* wodurch sich die Konsistenz des Systems ergibt.

Außerdem kann durch die Struktur des PTS gezeigt werden, dass jeder wohltypisierte Term zu einem Normalform-Term, in welchem keine Reduktionen mehr möglich wären, reduziert werden kann.

Diese Eigenschaften sind nicht nur entscheidend für eine Feststellung in endlicher Zeit, ob ein Term korrekt typisiert ist, sondern auch für die Zuverlässigkeit von Beweisassistenten wie Coq. Es erklärt, weshalb Coq korrekt funktioniert und warum man den Beweisen, die in Coq formuliert werden, vertrauen kann.

3.2.1 Produkttypen und Funktionstypen

Nachdem wir die Grundkonzepte des Calculus of Inductive Constructions (CIC) eingeführt haben, wenden wir uns nun den Typenkonstruktionen zu. Diese bilden das Fundament für komplexere Strukturen und sind damit entscheidend für die Ausdruckskraft des CIC, wie es später bei der Formalisierung des KLM-Theorems verwendet wird [20]. Wir betrachten zunächst *Produkttypen* und *Funktionstypen* und werden dann die Typenhierarchie des CIC behandeln und in dieser die Typenkonstruktionen einordnen.

Die Produkt- und Funktionstypen stellen die Grundbausteine für die Typentheorie dar und ermöglichen komplexe Typen für nichtmonotones Schließen, wie es in unserer Formalisierung von System C und kumulativen Modellen genutzt wird [11]. Damit bilden diese die formale Grundlage für logische Strukturen. Zudem charakterisieren wir in der Typentheorie Datentypen durch zwei Arten von Regeln. Auf der einen Seite gibt es *Einführungsregeln*, welche beschreiben, wie Werte von einem Typ konstruiert werden können, und auf der anderen Seite haben wir *Eliminationsregeln* durch welche festgelegt wird, wie auf solche Werte zugegriffen wird und wie diese verwendet werden [15].

Die Grundidee bei Produkttypen ist die Kombination zweier Typen zu einem neuen Typ, welcher dann geordnete Paare aus den beiden Ausgangstypen enthält. Formal bedeutet dies $A \times B$ ist definiert als die Menge aller Paare (a, b) mit $a \in A$ und

$b \in B$, also:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}.$$

Produkttypen stellen die typentheoretische Umsetzung des kartesischen Produkts dar und werden in Coq mit `A * B` notiert.[3] Wie auch in der Mengenlehre gibt es eine Projektion π, welche es uns ermöglicht, aus einem Objektpaar eines Produkttyps `A × B` einen bestimmten Wert zu extrahieren. Für ein Objekt vom Typ `A × B`, also dem Paar (a, b), extrahiert die Projektion π_1 den Term a und die Projektion π_2 den Term b. Dies notieren wir durch $\pi_1(a, b) = a$ und $\pi_2(a, b) = b$.

Sei Γ der Typisierungskontext, welcher eine Menge von Variablen und ihre zugewiesenen Typen darstellt. Die Notation $\Gamma \vdash a : A$ bedeutet dabei, dass im Kontext Γ der Term a den Typ A hat.

Die Einführungsregel für Produkttypen [15] lautet damit:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

und die Eliminationsregeln ergeben sich wie folgt:

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_1(p) : A} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \pi_2(p) : B}$$

Eine Verallgemeinerung des Produkttypen stellt dabei der *abhängige Summentyp* (Σ-Typ) dar. Der abhängige Summentyp enthält Paare (a, b), wobei a vom Typ A und b vom Typ $B(a)$ ist. Das bedeutet, dass die Abhängigkeit hier durch $B(a)$ erreicht wird, da B von a abhängt. Formal notieren wir abhängige Summentypen als

$$\Sigma x : A.B(x) = \{(a, b) \mid a \in A, b \in B(a)\}$$

zusammen mit der Einführungsregel

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B(a)}{\Gamma \vdash (a, b) : \Sigma x : A.B(x)}$$

und den Eliminationsregeln

$$\frac{\Gamma \vdash p : \Sigma x : A.B(x)}{\Gamma \vdash \pi_1(p) : A} \qquad \frac{\Gamma \vdash p : \Sigma x : A.B(x)}{\Gamma \vdash \pi_2(p) : B(\pi_1(p))}$$

Wie wir erkennen, zeigt die zweite Eliminationsregel einen wichtigen Unterschied zu gewöhnlichen Produkttypen, da der Typ der zweiten Komponente vom Wert der ersten abhängt. Wenn wir also mit $\pi_1(p)$ die erste Komponente extrahieren, muss deren Wert in den Typ B eingesetzt werden, damit wir den korrekten Typ der zweiten Komponente erhalten.

[3] Die eigentliche Definition von Produkttypen ist dabei `Inductive prod (A B : Type)` und wird aber mit einer Notation syntaktisch durch $*$ vereinfacht [20].

In Coq wird dieser als `sig` oder `{x:A | B x}` notiert. Ein Beispiel für einen abhängigen Summentyp wäre der Typ aller Paare (n, v), wo n eine natürliche Zahl und a ein Vektor der Länge n ist. Die abhängigen Summentypen ermöglichen damit die Definition von Typen mit integrierten Invarianten. Zum Beispiel könnten wir in Coq `{n:N | even n}` definieren, was im Falle des Typs `N` der natürlichen Zahlen einen Typen definieren, welcher nur alle geraden Zahlen (`even`) beschreiben würde. Das Nutzen von diesem neuen Typ würde dann garantieren, dass wir nur mit geraden Zahlen arbeiten, da diese Eigenschaft in dem Typen selber kodiert ist. Wir verbinden damit also Daten und Metadaten, wie Beweise. Außerdem wird der abhängige Summentyp dafür verwendet, um die existenzielle Quantifikation in Coq umzusetzen. Dies ist genau dann der Fall, wenn `B` eine Proposition ist, was in Coq dem Typen **`Prop`** entspricht.

Durch die Funktionstypen ist es möglich, Abbildungen zwischen Typen zu definieren. $A \to B$ ist definiert als die Menge aller Abbildungen, die jedem Element aus A genau ein Element aus B zuordnen, das ist formal:

$$A \to B = \{f \mid f : A \to B\}.$$

Es handelt sich um eine Abbildung, die jedem Element $a \in A$ genau ein Element aus B zuordnet. In Coq notieren wir Funktionstypen mit `A -> B`. Zu beachten ist, dass in Coq `->` auch als logische Implikation genutzt wird, wenn `A` und `B` Propositionen sind. Bei Funktionstypen verwenden wir die λ-Abstraktion, um Funktionen zu definieren. Der Ausdruck $\lambda x : A.t$ bezeichnet dabei eine Funktion, welche ein Argument x vom Typ A nimmt und den Term t zurückgibt.

Somit lautet die Einführungsregel für Funktionstypen

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash (\lambda x : A.b) : A \to B}$$

und die Eliminationsregel lautet:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B}$$

Diese Regeln zeigen, wie Funktionen f durch die Abstraktion erstellt werden und wie sie dann durch die Anwendung verwendet werden. Die Anwendung der Funktion vom Typ $A \to B$ auf einem Argument vom Typ A ergibt dabei ein Ergebnis vom Typ B.

Auch hier gibt es eine verallgemeinerte Form des Funktionstypen, dargestellt durch den *abhängigen Produkttyp* (Π-Typ). Dabei ist erlaubt, dass der Rückgabetyp $B(x)$ selber vom Wert des Arguments x anhängt. Formal geschrieben ist

$$\Pi x : A.B(x) = \{f \mid \forall a \in A.f(a) \in B(a)\}$$

mit der Einführungsregel

$$\frac{\Gamma, x : A \vdash b : B(x)}{\Gamma \vdash \lambda x : A.b : \Pi x : A.B(x)}$$

und der Eliminationsregel:

$$\frac{\Gamma \vdash f : \Pi x : A.B(x) \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B(a)}$$

Wir notieren dies in Coq als `forall x: A,B x`. In Coq muss der Typ von `x` ebenfalls explizit angegeben werden, in diesem Fall ist es der Typ `A`. Zu beachten ist hier, dass durch das Schlüsselwort `forall` nahelegt, dass es sich dabei ausschließlich um die universelle Quantifikation handelt, jedoch beschreibt `forall` wiederum nur die universelle Quantifikation (∀), wenn es sich bei `B` um eine Proposition (Typ **`Prop`**) handelt. Ansonsten nutzt Coq das Schlüsselwort allgemein für abhängige Funktionstypen. Dies ist der Fall da Coq aufgrund des Curry-Howard-Isomorphismus, welchen wir in der Sektion 3.2.5 behandeln werden, die Allquantifikation und abhängige Funktionstypen als dasselbe Konzept ansieht, jedoch mit den unterschiedlichen Interpretationen je nachdem ob wir im Bereich der Logik mit **`Prop`** oder mit der Berechnung (**`Type`**) arbeiten.

Die größere Bedeutung der abhängigen Typen liegt darin, dass diese es ermöglichen, präzise Eigenschaften direkt im Typ auszudrücken. Dies führt einerseits zu einer stärkeren Typsicherheit und andererseits ermöglicht es mehr dieser Eigenschaften bereits zu Kompilierzeit zu prüfen, anstatt diese zur Laufzeit verifizieren zu müssen. Dies wird eine wichtige Rolle bei der Implementierung der syntaktischen Regeln des System C, der Spezifikation der Eigenschaften kumulativer Konsequenzrelationen und der Konstruktion des kumulativen Modells spielen.

Produkttypen und Funktionstypen erhalten grundsätzlich die höchste Sort-Ebene ihrer Komponenten, wobei Typenbildungsregeln genau definieren, in welcher Sort der resultierende Typ liegt.

Für Produkttypen gelten die folgenden Typenbildungsregeln:

$$\begin{aligned}
A : Type_i, \quad B : Type_j \quad &\Rightarrow \quad A \times B : Type_{\max(i,j)} \\
A : Prop, \quad B : Prop \quad &\Rightarrow \quad A \times B : Prop \\
A : Type_i. \quad B : Prop \quad &\Rightarrow \quad A \times B : Type_i
\end{aligned}$$

wobei $max(i, j)$ den Typen der höheren Sort-Ebene bestimmt, wodurch die höhere Sort dominant wird. Dies spiegelt die Kumulativität der Typenhierarchie wider.

Für Funktionstypen gelten die folgenden Typenbildungsregeln:

$$\begin{aligned}
A : Type_i, \quad B : Type_j \quad &\Rightarrow \quad A \to B : Type_{\max(i,j)} \\
A : Prop, \quad B : Prop \quad &\Rightarrow \quad A \to B : Prop \\
A : Type_i, \quad B : Prop \quad &\Rightarrow \quad A \to B : Prop
\end{aligned}$$

Die letzte Regel zeigt die Besonderheit von `Prop`. Wenn der Zieltyp B eine Proposition ist, dann ist der gesamte Funktionstyp ebenfalls eine Proposition, unabhängig davon, welcher Sort der Argumenttyp A angehört. Diese Regel ist eine direkte Konsequenz der Tatsache, dass `Prop` imprädikativ ist, was bedeutet, dass in `Prop` über

alle Propositionen qunatifiziert werden darf und diese Quantifikation dann selbst wieder eine Proposition ist. Damit unterscheidet sich diese Regel von der entsprechenden Regel für Produkttypen und ermöglicht auch im CIC, dass Quantifikationen über alle Propositionen selber wieder Propositionen sind. Produkt- und Funktionstypen bilden damit die Grundlage für das Typsystem des CIC, und ermöglichen die Kombination und Transformation von Typen. Dies ist jedoch nicht ausreichend für komplexere Datenstrukturen wie rekursive Definitionen, da sie in der Ausdruckskraft für selbstreferenzielle oder induktiv definierte Konzepte begrenzt sind. Sie können nicht direkt auf sich selbst verweisen und bieten keine Möglichkeit für eine Fallunterscheidung, was für das Definieren induktiver Strukturen benötigt ist. Dafür werden wir folgend die Erweiterung des Typsystems, die *induktiven Definitionen*, vorstellen, welche es möglich machen, rekursive Strukturen zu erzeugen und Definitionen von Datentypen durch deren Konstruktionsweise anzugeben. Induktive Definitionen werden ein wichtiges Werkzeug zur Formalisierung des KLM-Theorems sein, da diese uns erst ermöglichen, die Konsequenzrelationen und Modelle präzise zu definieren.

3.2.2 Induktive Definitionen

Die Erweiterung, welche den Calculus of Constructions zu dem Calculus of Induktive Constructions erweitert, beschreibt die induktiven Definitionen. In Coq werden diese wiederum durch das Schlüsselwort **`Inductive`** definiert. Induktive Definitionen werden dabei verwendet, um Datentypen oder Relationen durch Konstruktoren zu definieren. Die Konstruktoren geben dabei an, wie Objekte eines bestimmten Typs gebildet werden können und können als die Bildungsregeln gesehen werden. Das bedeutet, dass bei der Definition eines induktiven Typs die Konstruktoren mit angegeben werden müssen. Diese beschreiben, wie die Basiselemente (zum Beispiel das Startelement) eines Typs aussehen und wie komplexere Elemente aus einfacheren gebildet werden können. Dabei sind Konstruktoren injektiv, da verschiedene Eingaben verschiedene Ausgaben erzeugen. Außerdem sind ihre Bilder disjunkt, da Ausgaben verschiedener Konstruktoren unterschiedlich sind. Jedes Element des Typs wird dabei durch genau eine Folge von Konstruktoranwendungen erzeugt, was die Beweisführung durch Induktion ermöglicht, da dann alle Fälle durch die Konstruktoren abgedeckt werden. In Coq werden die Konstruktoren durch „|“ (*pipe*) getrennt. Um dies zu verdeutlichen, betrachten wir ein Beispiel von Paulin-Mohring aus *Introduction to the Calculus of Inductive Constructions* [15].

Beispiel 3. *Induktive Definition*

```
Inductive N : Type :=
| z : N
| S : N -> N.
```

Hier wird der Typ von `N`, welcher Eigenschaften von natürlichen Zahlen darstellen soll, durch zwei Konstruktoren definiert. Der erste Konstruktor `z` wird für das Startelement, welches den Basisfall darstellt, definiert. Dieses Element kann ohne die Verwendung anderer Elemente des Typs konstruiert werden. Wir interpretieren also `z` als 0, da wir hier `N` als Repräsentation der natürlichen Zahlen verstehen wollen. Die vollständige Deklaration `z : N`, gibt an, dass `z` ein Element vom Typ `N` erzeugt. Der zweite Konstruktor `S` erzeugt aus einem Element von `N` ein neues Nachfolge-Element von `N`. Wir bezeichnen `S` auch als einen Konstruktor höherer Ordnung, da dieser als Argument einen Wert von `N` hat, welchen er gerade selber mitdefiniert. Das bedeutet, dass der Konstruktor `S` rekursiv auf sich selber verweist und damit unendlich viele Elemente ermöglicht. Die Rekursion liegt bei einer induktiven Definition daher in der Struktur des Typs. Dadurch, dass `S` die Nachfolgeoperation abbildet, wird ermöglicht, alle natürlichen Zahlen (außer 0) iterativ zu konstruieren. Der Konstruktor `S` ist injektiv, das bedeutet, wenn es ein `n` und `m` vom Typ `N` gibt und `S n = S m` gilt, dann gilt auch `n = m`. Umgekehrt, wenn `S n` $\neq$ `S m` gilt, gilt auch `n` $\neq$ `m`. Es ist also nicht möglich, dass zwei verschiedene Elemente `n` und `m` durch Anwendung des Konstruktors `S` zum selben Element werden. Zudem ist das Bild des Konstruktors `S` disjunkt von dem Bild von `z`, denn für alle `n` vom Typ `N` gilt `S n` $\neq$ `z`, da ein Term, der mit `S` beginnt, niemals gleich `z` sein kann.

Nach der Einführung induktiver Definition stellt sich vielleicht die Frage, wie Eigenschaften für alle Elemente eines induktiven Typs bewiesen werden können. Dafür generiert Coq für jede induktive Definition automatisch das *Induktionsprinzip*. Es ist ein grundlegendes Prinzip für die Beweisführung in Coq und spiegelt die Struktur der induktiven Definition wider.

Eine wichtige Einschränkung für induktive Definitionen stellt die *Positivitätsbedingung* [15] dar, welche sicherstellt, dass keine paradoxen Typen definiert werden können.

Definition 7 (Positivitätsbedingung)**.** *Eine induktive Definition ist positiv, wenn der zu definierende Typ T in allen Konstruktortypen nur in positiven Positionen vorkommt.*

Sei T ein zu definierender induktiver Typ mit Konstruktoren $c_1, \ldots, c_n$ vom Typ $C_1 \ldots, C_n$. T erscheint in positiver Position in:

- *Einem Typ A, wenn T nicht in A vorkommt*
- *$A \to B$, wenn T nicht in negativer Position in A und positiver Position in B vorkommt*
- *$\Pi x : A.B(x)$, wenn für alle $a \in A$, T in positiver Position in $B(a)$ vorkommt.*

Durch diese Bedingung wird die Konsistenz des Typsystems sichergestellt. Ohne diese könnten wir paradoxe Definitionen angeben, was wir mit dem folgenden Beispiel demonstrieren wollen.

Beispiel 4. *Paradoxer induktiver Typ*

```
Inductive P : Prop :=
| paradox : (P -> False) -> P.
```

Wie in Beispiel 4 zu erkennen ist, liegt `P` in negativer Position, also links vom Pfeil, als Argument des Konstruktors vor, was direkt gegen die Positivitätsbedingung verstößt. Diese Definition würde zu einem logischen Widerspruch führen, da wir so einen Beweis von `False` ohne Annahmen konstruieren könnten. Durch die Positivitätsbedingung wird also ebenfalls sichergestellt, dass wir einen wohldefinierten Induktionsschritt haben, was uns nun zum Induktionsprinzip führt.

3.2.3 Das Induktionsprinzip

Jede induktive Definition führt automatisch zu einem korrespondierenden Induktionsprinzip [20]. Konkret formalisiert das Prinzip das Muster von Induktionsbasis und Induktionsschritt.

Das Induktionsprinzip für einen induktiven Typ T mit Konstruktoren $c_1, \ldots, c_n$ wird automatisch als eine Funktion vom Typ:

$$T\_ind : \forall P : T \to \text{Prop}, \quad P\_c_1 \to \cdots \to P\_c_n \to \forall x : T, P(x)$$

generiert, wobei P_c_i die Induktionshypothese für den Konstruktor c_i ist. Für jeden rekursiven Aufruf eines Konstruktors gilt:

wenn $c_i : \cdots \to T \to \cdots \to T$ lautet,

dann ist $P\_c_i : \cdots \to \forall t : T, P(t) \to \cdots \to P(c_i(\ldots, t, \ldots))$.

Als Beispiel betrachten wir das Induktionsprinzip des induktiven Typen `N`. Wir kennzeichnen das Induktionsprinzip hier mit $\texttt{N}_{\texttt{ind}}$.

Beispiel 5. *Induktionsprinzip des induktiven Typen `N`*

```
N_ind : forall P : N -> Prop,
P z -> (forall n : N, P n -> P (S n)) ->
forall n : N, P n.
```

Wie man an der Signatur des Induktionsprinzips erkennen kann, handelt es sich hierbei um einen abhängigen Produkttyp. Zuerst wird hier eine Eigenschaft `P` von natürlichen Zahlen `N` definiert. Eine Eigenschaft ist dabei eine Funktion, welche jeder natürlichen Zahl einen Wahrheitswert `Prop` zuordnet `N -> Prop`. Zum Beispiel könnte `P` die Eigenschaft „ist gerade“ oder „ist größer als 10“ darstellen. `P z` gibt dann an, dass eine Eigenschaft `P` für z, also nach unserer Definition für 0, gilt. Dies stellt den Basisfall der Induktion dar.

Für den Induktionsschritt betrachten wir `forall n : N, P n -> P (S n)`. Dies sagt aus, dass wenn die Eigenschaft `P` für eine beliebige Zahl `n` gilt, dann gilt

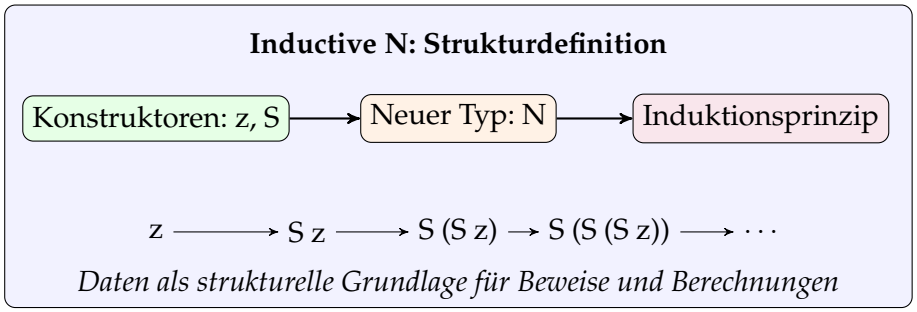

Abbildung 1: Konzeptuelle Darstellung einer induktiven Definition.

diese auch für den Nachfolger von `n`, als für `S n`. Das heißt, wenn zum Beispiel `n + m = x` gilt, dann gilt auch `(S n) + m = S x`. Zuletzt wird noch die Schlussfolgerung des Prinzips angegeben, welche aussagt, dass eine Eigenschaft `P` nach dem Zutreffen der vorherigen Bedingungen, für alle natürlichen Zahlen gilt. Zusammenfassen besagt das ganze Prinzip also „Wenn eine Eigenschaft für 0 gilt und wenn aus der Gültigkeit für eine Zahl `n` auch die Gültigkeit für `n + 1` folgt, dann gilt die Eigenschaft für alle natürlichen Zahlen".

Das Induktionsprinzip spiegelt direkt die Konstruktoren von dem Typ `N` wider, da `N` durch z (0) und die S-Operation (Nachfolgefunktion) aufgebaut wird. Dabei führt jeder Konstruktor zu einer Prämisse im Induktionsprinzip und die Form des Induktionsschritts wird durch die Struktur der rekursiven Konstruktoren bestimmt.

Abbildung 1 verdeutlicht den konzeptuellen Zusammenhang der induktiven Definition `N`. Aus den definierten Konstruktoren `z` und `S` entsteht der neue Typ `N`, wofür Coq automatisch das Induktionsprinzip generiert. Ebenfalls stellen wir Instanzen des Typs dar, welche durch wiederholte Anwendung der Konstruktoren entstehen. Diese strukturelle Definition bildet die Grundlage für Beweise durch Induktion als auch für Berechnungen mit rekursiven Funktionen auf diesem Typ.

Folgend möchten wir noch auf zwei weitere Beispiele von Paulin-Mohring [15] eingehen, die auf dem obigen Beispiel aufbauen, um die Relevanz der induktiven Definition weiter hervorzuheben und weitere wichtige Eigenschaften darzustellen.

Beispiel 6. *Induktive Definition als Relation*

```
Inductive le : N -> N -> Prop :=
| lez:∀x, le z x
| leS:∀x y, le x y -> le (S x) (S y).
```

Diese Definition deklariert eine binäre Relation `le` auf natürlichen Zahlen. Dabei gibt der Typ `N -> N -> Prop` an, dass zwei natürliche Zahlen genommen werden und eine Proposition zurückgegeben werden soll. Wieder definieren die zwei Konstruktoren, wann diese Relation gilt. Der Konstruktor `lez` gibt dabei an, dass `z` $\leq$ `x` für alle natürlichen Zahlen `x`, also 0 ist kleiner oder gleich jeder Zahl, gilt.

Der zweite Konstruktor `leS` definiert, dass wenn `x ≤ y` gilt, dann gilt auch `S z ≤ S x`. Das bedeutet, auch wenn beide Zahlen erhöht werden, bleibt die Relation erhalten.

Die ≤-Relation wird durch ihre grundlegenden Eigenschaften durch die Definition `le` charakterisiert, und die kleinste Relation, die diese Regeln erfüllt, ist genau die mathematische ≤-Relation.

Beispiel 7. *Induktive Definition mit Parametern*

```
Inductive RT A (R : A -> A -> Prop) : A -> A -> Prop :=
| RTrefl:∀ x, RT A R x x
| RTR:∀ x y, R x y -> RT A R x y
| RTtran:∀ x y z, RT A R x z -> RT A R z y -> RT A R x y.
```

Bei dieser Definition wird die reflexiv-transitive Hülle einer beliebigen Relation `R` auf einem Typ `A` formalisiert. Erneut gibt hier der Typ `A -> A -> Prop` an, dass die neue Relation `RT A R` ebenfalls eine binäre Relation auf `A` ist. Dabei ist der Parameter `A` der Basistyp, auf welchem die Relation definiert ist und der Parameter `R` die Ausgangsrelation, von der die Hülle gebildet wird. Der erste Konstruktor `RTrefl` besagt, dass jedes Element in Relation zu sich selbst steht, dies bedeutet, dass egal welches Element `x` von dem Typ `A` betrachtet wird, immer `RT A R x x` gilt. Dies beschreibt die Eigenschaft, dass die reflexiv-transitive Hülle reflexiv ist. Der Konstruktor `RTR` definiert, wenn zwei Elemente `x` und `y` in der Relation `R` stehen, dann sind diese auch in der reflexiv-transitiven Hülle `RT A R` enthalten. Es stellt dabei also sicher, dass die Relation `R` vollständig in der reflexiv-transitiven Hülle enthalten ist. Letztlich implementiert der Konstruktor `RTtran` die Transitivität. Das bedeutet, wenn ein Element `x` in Relation zu `z` steht und `z` ebenfalls in Relation zu dem Element `y` steht, dann steht auch `x` in Relation zu `y`.

Diese Definition demonstriert, wie in CIC höherstufige Definitionen mit Relationen als Parametern erstellt werden können, welche im weiteren Verlauf der Arbeit noch eine wichtige Rolle spielen werden.

3.2.4 Rekursive Funktionen auf induktiven Typen

Nachdem wir gesehen haben, wie induktive Definitionen Datentypen und Relationen definieren können, wollen wir nun zeigen, wie Funktionen über diesen Typen definiert werden können. In Coq werden Funktionen über induktive Typen mit dem Schlüsselwort **`Fixpoint`** definiert, welches ebenfalls rekursive Definitionen ermöglicht. Um dies zu veranschaulichen, zeigen wir zwei Beispiele, in denen die Funktionen Operationen über den zuvor gezeigten induktiven Typen `N` definieren.

Beispiel 8. *Rekursion Addition auf natürlichen Zahlen*

```
Fixpoint add (n m : N) {struct n} : N :=
  match n with
  | z => m
  | S p => S (add p m)
  end.
```

Dabei gibt `(n m : N)` an, dass zwei Eingabeparameter `n` und `m` vom Typ `N` erwartet werden und `: N` am Ende der Definition, dass der Ausgabeparameter ebenfalls vom Typ `N` ist. Wie zu erkennen ist, führen wir mit diesem Beispiel ebenfalls den **`struct`**-Ausdruck und den **`match`**-Ausdruck ein. Der **`struct`**-Ausdruck gibt an, über welchen Parameter, in diesem Fall `n`, die Rekursion erfolgen soll. Dies gibt für die Prüfung der Rekursion an, dass bei jedem rekursiven Aufruf von `add` der Parameter `n` verringert werden soll, wir bezeichnen dies auch als den *Terminierungsnachweis*. Der **`match`**-Ausdruck wird verwendet, um *Pattern-Matching* durchführen zu können. Bei dem Pattern-Matching wird zuerst angegeben, auf welchem Parameter das Pattern-Matching stattfinden soll, in unserem Fall ist dies der Parameter `n`. Danach werden die verschiedenen Fälle angegeben, im Beispiel sind es zwei Fälle, die auch wieder mit `|` getrennt werden.

1. Im ersten Fall `n = z` ist das Ergebnis direkt `m`, der zweite Parameter.
2. Im zweiten Fall `n = S p` ist das Ergebnis `S (add p m)`, wobei `p` der Subterm von `n` ist, da `p` strukturell keiner ist als `S p`.

Die Eigenschaft „strukturell kleiner“ bedeutet dabei, dass ein Term durch das Entfernen von mindestens einem Konstruktor aus einem anderen Term folgt. Der Terminierungsnachweis garantiert hierbei, dass die Rekursion nicht unendlich ist, da rekursive Aufrufe nur auf strukturell kleineren Argumenten erlaubt sind. Für das Beispiel bedeutet dies, dass `p` strukturell kleiner ist als `n = S p`. Durch das Reduzieren des Konstruktors `S` von `n` entsteht `p`. Dabei wird der Ausdruck `n = S p` durch Pattern-Matching im zweiten Fall hergeleitet. Der rekursive Aufruf erfolgt demnach auf einem strukturell kleineren Term im zweiten Fall. Dies ist eine essenzielle Bedingung für rekursive Aufrufe im CIC, denn sie garantiert, dass rekursive Funktionen terminieren. Durch rekursive Aufrufe auf strukturell kleineren Termen werden immer weitere Konstruktoren des Terms reduziert. Dies führt zwangsläufig irgendwann immer dazu, bis zum Basiskonstruktor reduziert zu haben, wonach keine Reduktion mehr möglich ist.

Zur Veranschaulichung der Fixpoint-Definition stellen wir Abbildung 2 vor. Diese visualisiert die Hauptkomponenten einer Fixpoint-Definition. Als Eingabe haben wir den induktiven Typ `N`, auf welchem die Funktion operiert. Zentral stellen wir als Verhaltensregel die Fixpoint-Funktion `add` und deren zwei Hauptfälle, welche über den **`match`**-Ausdruck ausgewählt werden, dar. Der Basisfall `add z m = m` liefert direkt ein Ergebnis ohne Rekursion und der rekursive Fall ist der Aufruf auf dem strukturell kleineren Term `(add (S p)m = S (add pm))`. In Abhängigkeit

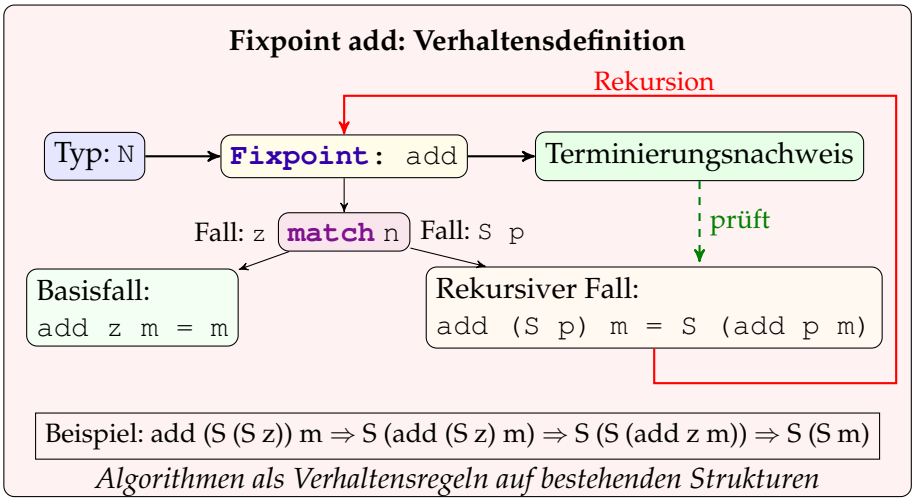

Abbildung 2: Konzeptuelle Darstellung einer Fixpoint-Definition mit Rekursionsvisualisierung

zu der Hauptfunktion `add` steht der Terminierungsnachweis, welcher durch die Angabe des strukturell abnehmenden Arguments, in dem Fall von `add` ist dies **`struct n`**, angegeben wird. Er prüft, ob bei jedem rekursiven Aufruf tatsächlich ein strukturell kleiner Term verwendet wird. Durch diese strenge Anforderung wird die logische Konsistenz des Beweissystems sichergestellt. Die Fixpoint-Definition implementiert demnach das Prinzip der strukturellen Rekursion in Coq und setzt das Grundprinzip aus dem CIC, dass alle Berechnungen terminieren müssen, praktisch um.

Nun kommen wir zu einem Beispiel, bei dem nicht unmittelbar offensichtlich ist, dass die Rekursion terminiert.

Beispiel 9. *Rekursion Division mit Rest*[4]

```
Fixpoint div (a b : N) {struct a} : N * N :=
  match b <=? a with
  | false => (z, a)
  | true  => let (q, r) := div (a sub b) b in (S q, r)
  end.
```

Wie zu erkennen ist, erfolgt der rekursive Aufruf auf dem Term `(a sub b)`. Hier ist aber syntaktisch nicht erkennbar, dass es sich bei dem Term um einen strukturell kleineren Term als `a` handelt, da der Term das Ergebnis einer Berechnung

[4]Wir setzten hier voraus, dass eine Entscheidungsfunktion `<=?` existiert, die bestimmt, ob `(le b a)` zutrifft und ebenfalls, dass es eine Definition für `sub` gibt, um zwei Argumente vom Typ `N` voneinander subtrahieren zu können. Diese wurden einfachheitshalber nicht explizit angegeben [25].

darstellt. Intuitiv lässt sich erkennen, dass so lange `(b > 0)` gilt, `(a - b)` in jedem rekursiven Schritt kleiner wird. Das heißt, irgendwann wird auch der Basisfall `a < b`, also dass nicht `(b ≤ a)` gilt, durch die Rekursion erreicht. Diese intuitive Erkenntnis ist für uns offensichtlich, und wir könnten davon ausgehen, dass die Funktion terminiert. Für die Anforderungen des CIC reicht dies aber nicht aus, da die Regeln des CIC auf einer syntaktischen Prüfung der rekursiven Elemente basieren. Auch Coq kann, trotz der expliziten `{struct a}` Anweisung, nicht automatisch prüfen, ob die Rekursion tatsächlich terminiert. Daher ist der Benutzer gefragt, um eine formale Garantie, also einen Beweis zu liefern, dass der Term `(a sub b)` des rekursiven Aufrufs tatsächlich strukturell kleiner ist als `a`. Für unseren zweiten Fall, der eintrifft, wenn `(b ≤ a)` gilt, führen wir zusätzlich noch eine lokale Bindungskonstruktion `let(q,r) ... in ...` ein. Damit werden lokale Variablen für Zwischenergebnisse erzeugt, wobei der Term nach `in` dann auf diese Variablen zugreifen kann. Dies entspricht mathematisch: Sei `(q,r)` das Paar (`q` der Quotient und `r` der Rest), dass sich aus dem rekursiven Aufruf der Division von $\frac{(a - b)}{b}$ ergibt, dann ist das Ergebnis der ursprünglichen Division das Paar `(q + 1 ,r)`, also der um 1 erhöhte Quotient `q` mit dem gleichen Rest `r`. Das Ergebnis ist ein Paar von dem Typ `N`, was durch den Rückgabeparameter `N * N` angegeben wird.

Im Folgenden möchten wir zusammenfassend die Unterschiede und Gemeinsamkeiten von **`Inductive`** und **`Fixpoint`** zusammentragen und verdeutlichen. Während **`Inductive`** genutzt wird, um Datentypen oder Relationen durch dessen Konstruktoren zu definieren, definiert **`Fixpoint`** wiederum rekursive Funktionen auf eben diesen induktiven Typen. Damit beschreibt **`Inductive`** die Struktur von Typen, also„was etwas ist“ und **`Fixpoint`** das Verhalten von solchen Typen, also „was etwas tut“. Das bedeutet, dass durch **`Inductive`** neue Typen und deren dazugehörige Induktionsprinzipien erzeugt werden, und diese nun bestehenden Typen und deren Struktur dann bei **`Fixpoint`** genutzt werden. Dieses Zusammenspiel haben wir in Abbildung 3 zur Veranschaulichung dargestellt. Dabei enthalten beide Definitionen rekursive Elemente. Bei **`Inductive`** liegt die Rekursion in der Struktur des Typen selbst, während bei **`Fixpoint`** die Rekursion in der Funktionsdefinition auftritt. Beide unterliegen in der Rekursion bestimmten Terminierungsbedingungen. **`Inductive`** muss dabei Positivitätsbedingungen erfüllen und **`Fixpoint`** muss strukturelle Verkleinerung nachweisen.

Der Anwendungsbereich beider unterscheidet sich ebenfalls. **`Inductive`** wird für Beweisstrukturen und Datentypen verwendet, wobei hingegen **`Fixpoint`** für Algorithmen und Berechnungen verwendet wird.

Wie wir sehen, ermöglichen induktive Definitionen und deren rekursive Funktionen die strukturierte Konstruktion mathematischer Objekte sowie die Formulierung und den Beweis von Eigenschaften über diese mittels struktureller Induktion. Die induktiven Definitionen ermöglichen präzise Strukturierung von Daten und Propositionen. Fixpoint-Funktionen erlauben dann Berechnungen auf diesen Strukturen. Dieser Zusammengang ist in Abbildung 3 visualisiert.

Damit bilden diese zusammen das Fundament für formale Definitionen und Be-

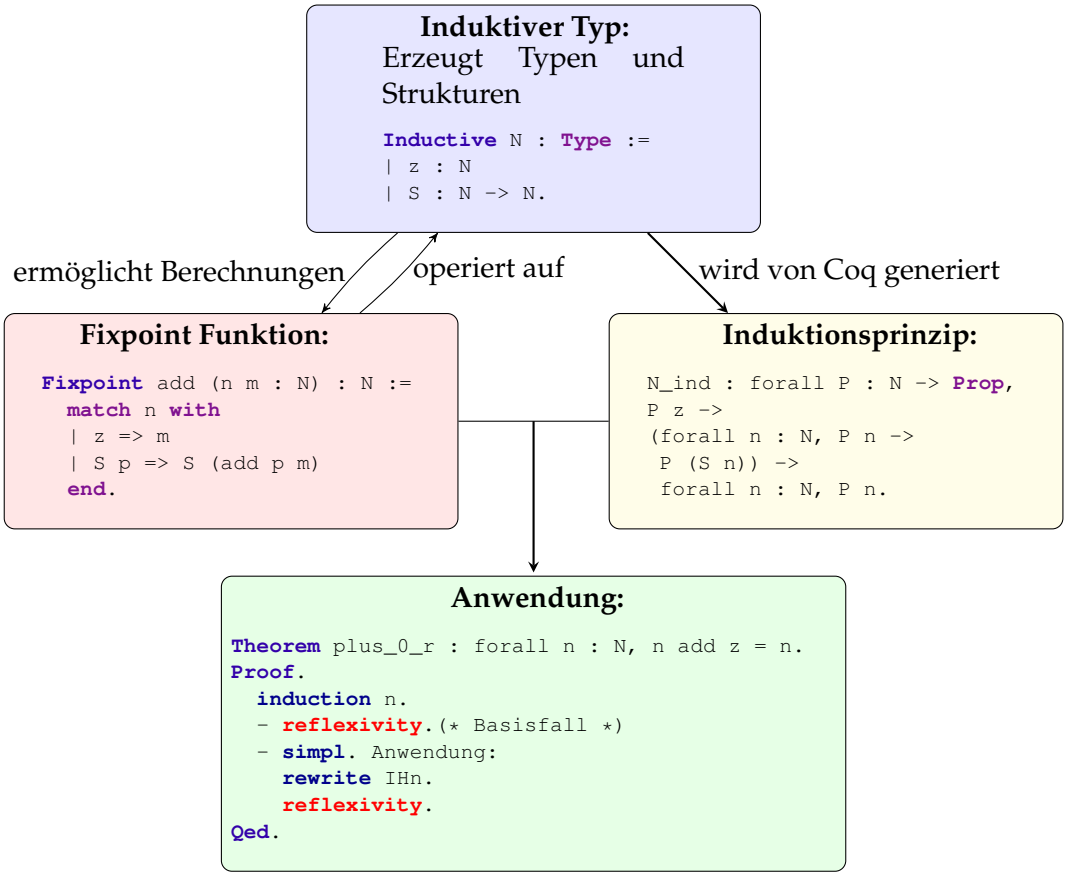

Abbildung 3: Beziehung zwischen induktivem Typ, Induktionsprinzip und Fixpoint

weise und bilden zentrale Konzepte des CIC ab.

Nachdem wir nun die grundlegenden Konzepte des CIC, die Produkttypen, Funktionstypen, induktiven Definitionen und rekursive Funktionen vorgestellt haben, kommen wir nun zum Curry-Howard-Isomorphismus, wo sich die tiefere Bedeutung dieser grundlegenden Konzepte erschließt. Dieser Isomorphismus stellt eine Beziehung zwischen Typentheorie und Logik her und erklärt, weshalb wir Coq sowohl als Programmiersprache als auch als Beweissystem bezeichnen können.

3.2.5 Curry-Howard-Isomorphismus im CIC

Der Curry-Howard-Isomorphismus stellt in dem CIC eine fundamentale Verbindung zwischen Logik und Berechnungen dar und findet den Ursprung in den Arbeiten von Curry (1934) und Howard (1969). Haskell Curry erkannte die Ähnlichkeit zwischen der Struktur von Theoremen in der *Combinatory Logic* [6] und Typen in der functional Analysis, und William Howard formalisierte später diese Bezie-

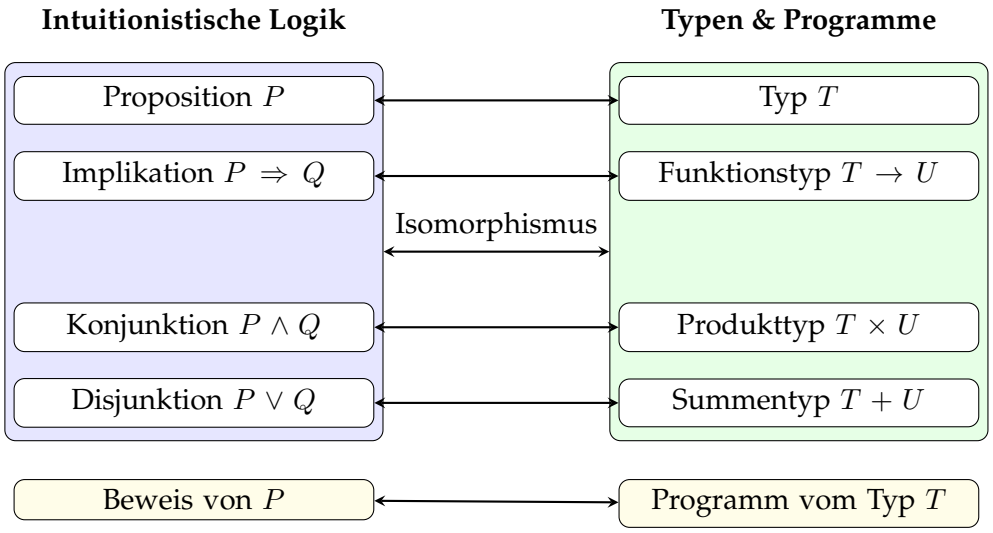

Abbildung 4: **Curry-Howard-Isomorphismus**. Formale Korrespondenz zwischen Beweisen in der intuitionistischen Logik und Programmen im typisierten λ-Kalkül

.

hung als einen Isomorphismus zwischen *Intuitionistic Logic* [9] und dem einfachen typisierten λ-Kalkül. Später wurden diese Arbeiten dann als der Curry-Howard-Isomorphismus bekannt.

Die Grundidee des Curry-Howard-Isomorphismus ist dabei, dass jeder Typ einer Proposition, und jedes Programm (Term) dieses Typs einem Beweis dieser Proposition entspricht. Das bedeutet, dass ein Term vom Typ `T` ein konstruktiver Beweis ist, dass `T` wahr ist und es gilt:

- die logische Konjunktion ($A \wedge B$) entspricht Produkttypen ($A \times B$)
- die logische Disjunktion ($A \vee B$) entspricht Summentypen ($A + B$)
- die logische Implikation ($A \Rightarrow B$) entspricht Funktionstypen ($A \to B$)
- die universelle Quantifizierung ($\forall x.P(x)$) entspricht abhängigen Produkttypen ($\Pi x : A.P(x)$)
- die existenzielle Quantifizierung ($\exists x.P(x)$) entspricht abhängigen Summentypen ($\Sigma x : A.P(x)$)

Diese Korrespondenzen ermöglichen die Interpretation von konstruktiven Beweisen als ausführbare Programme.

Abbildung 4 visualisiert die grundlegenden Korrespondenzen des Curry-Howard-Isomorphismus. Die erweiterten Korrespondenzen mit abhängigen Typen, die für

die volle Ausdruckskraft des CIC wesentlich sind, umfassen zusätzlich die universelle Quantifizierung ($\forall x.P(x)$) als abhängigen Produkttyp ($\Pi x : A.P(x)$) und die existenzielle Quantifizierung ($\exists x.P(x)$) als abhängigen Summentyp ($\Sigma x : A.P(x)$).

Schon der Calculus of Constructions von Coquand und Huet wurde als Erweiterung des Curry-Howard-Isomorphismus entwickelt, um sich diese Eigenschaften zunutze zu machen. Dabei war die Kernidee, ein System zu schaffen, welches sowohl als Programmiersprache und auch als Beweissystem dienen kann.

Durch die Erweiterung der grundlegenden Korrespondenz auf induktive Typen wird dann die Darstellung von Datentypen und induktiven Beweisen ermöglicht:

- ein induktiver Typ `Inductive I : Prop := c1 : T1 | ...| cn : Tn` entspricht einer induktiven Proposition
- jeder Konstruktor `ci` entspricht einer Bildungsregel für die Proposition
- das von Coq automatisch erzeugte Induktionsprinzip `I_ind`, welches zur Durchführung struktureller Induktionen verwendet wird, entspricht der Eliminationsregel für die Proposition
- und die Reduktionsregeln für das Pattern-Matching entsprechen den Berechnungsregeln für Beweise.

So repräsentiert zum Beispiel der induktive Typ `bool := true | false` die Proposition „wahr oder falsch“, der induktive Typ `N := z | S N` der Induktion über natürliche Zahlen und ein induktiv definiertes Prädikat `le : N -> N -> Prop` einer binären Relation auf natürlichen Zahlen.

Das Induktionsprinzip der induktiven Definitionen wird in dem Curry-Howard-Isomorphismus als eine spezielle Form der Eliminationsregel für induktive Typen angesehen, welche wir bereits für Produkt- und Funktionstypen eingeführt haben. Diese ermöglicht es, einen durch Konstruktoren gebildeten Term systematisch in seine strukturellen Teilterme zu zerlegen, also die Werte eines Typs zu verwenden. Dabei werden komplexere Terme entsprechend der induktiven Definition von ihrem Typ rekursiv in einfachere Terme aufgelöst und für jeden Konstruktor wird definiert, wie aus den Ergebnissen der Teilterme ein Gesamtergebnis zu konstruieren ist.

Um diesen Prozess zu verdeutlichen, betrachten wir ein konkretes Beispiel der Addition zweier natürlicher Zahlen mittels der Eliminationsregel:

Beispiel 10. *Anwendung der Eliminationsregel zur Addition*

Betrachten wir die Addition $n + m$ mit $n = 2$ (repräsentiert als `S (S z)`) und $m = 3$ (repräsentiert als `S (S (S z))`). Die Eliminationsregel `N_ind` wird auf n, mit dem Basiswert m und der Nachfolgefunktion `S` für den rekursiven Fall, angewendet:

$$\begin{aligned}&\texttt{add}\,(\texttt{S}\ \ (\texttt{S}\ \ \texttt{z}))\,(\texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ \texttt{z})))\\&= \texttt{N\_ind}\,(\texttt{S}\ \ (\texttt{S}\ \ \texttt{z}))\,(\texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ \texttt{z})))\ \texttt{S}\end{aligned}$$

Die Elimination erfolgt durch strukturelle Zerlegung von n:

$$\begin{aligned}&= \text{Fall } \texttt{S}\ \ (\texttt{S}\ \ \texttt{z})\ \text{: rekursiver Aufruf auf } \texttt{S}\ \ \texttt{z}\\&= \texttt{S}\,(\texttt{N\_ind}\,(\texttt{S}\ \ \texttt{z})\ (\texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ \texttt{z}))))\\&= \text{Fall } \texttt{S}\ \ \texttt{z}\ \text{: rekursiver Aufruf auf } \texttt{z}\\&= \texttt{S}\,(\texttt{S}\,(\texttt{N\_ind}\,\texttt{z}\ (\texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ \texttt{z})))))\\&= \text{Fall } \texttt{z}\ \text{: Rückgabe des Basiswerts } \texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ \texttt{z}))\\&= \texttt{S}\,(\texttt{S}\,(\texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ \texttt{z}))))\\&= \texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ (\texttt{S}\ \ \texttt{z}))))\\&= 5\end{aligned}$$

Die Eliminationsregel zerlegt den Term n in seine strukturellen Teilterme, wobei für jeden `S`-Konstruktor in n ein rekursiver Aufruf erfolgt. Im Basisfall wird der Wert m zurückgegeben, und beim Zurückverfolgen der Rekursion wird für jeden `S`-Konstruktor in n einmal die Funktion `S` auf das Zwischenergebnis angewendet. Dies entspricht der mathematischen Berechnung $2 + 3 = 5$.

Letztendlich ermöglicht der Curry-Howard-Isomorphismus und dessen Erweiterung durch den CIC, dass sowohl Programmierung als auch formale Beweisführungen in Coq möglich sind. Dabei werden Programme und Beweise in derselben Sprache ausgedrückt und die Beweise wiederum können als Korrektheitszertifikate für Programme dienen. Außerdem können Programme aus konstruktiven Beweisen extrahiert werden, da konstruktive Beweise implizit Algorithmen enthalten. Durch dessen Extraktion können also die berechnenden Bestandteile des Beweises isoliert werden, wobei die nicht berechnenden Bestandteile, wie zum Beispiel Annahmen, des Beweises entfernt werden. Solch ein Korrektheitsbeweis ist zum Beispiel in der Entwicklungsphase von sicherheitskritischer Software relevant, ermöglicht aber auch die Zertifizierung von Programmen. Durch Coq wurde zum Beispiel der C-Compiler *CompCert* [13] zertifiziert, wodurch der Complier selbst und dessen Korrektheit in Coq formal bewiesen wurden.

Wir haben uns in der letzten Sektion bereits durch die Beispiele von **`Induktive`** und **`Fixpoint`** angesehen, wie eine Implementierung in Coq möglich ist, ohne uns dabei die formale Sprache anzusehen. Diese Implementierung wird mit der Spezifikationssprache *Gallina* vorgenommen, welche wir folgend genauer vorstellen werden.

3.3 Gallina als deklarative Programmiersprache von Coq

Der Name „Gallina“ kommt aus dem Italienischen und steht für „Huhn“, eine kleine Anspielung auf Coq was, aus dem Französischen stammt und „Hahn“ bedeutet. Es entstand zusammen mit Coq in den frühen 1990er Jahren durch Coquand, Huet und Paulin-Mohring [20] und wurde als formale Spezifikationssprache für mathematische Definitionen und Beweise in Coq konzipiert. Dabei liegt der Fokus stärker auf der Ausdruckskraft als auf der Effizienz, da das Hauptziel ist, mathematische Präzision und Beweisbarkeit zu erreichen anstelle von Laufzeitperformanz. Es muss jedoch auch garantiert sein, dass alle Funktionen nachweisbar terminieren. Dies wirkt auf den ersten Blick so, als würde es die Ausdruckskraft einschränken, da nicht alle intuitiv berechenbaren Funktionen direkt ausdrückbar sind, wie wir bei dem Beispiel Rekursion Division mit Rest gesehen haben. Die Terminierungsanforderung dient jedoch primär der logischen Konsistenz, nicht der Effizienz, da diese paradoxe Definitionen verhindert und garantiert, dass Berechnungen einen wohldefinierten Wert liefern. Es handelt sich also um strenge Terminierungsanforderungen, die durch den CIC die Eigenschaft mit sich bringen, dass alle rekursiven Funktionen strukturell rekursiv sein müssen.

Die Ausdruckskraft in Gallina bezieht sich auf die Mächtigkeit des Typensystems, wie abhängige Typen oder Induktion, und die Fähigkeit, mathematische Konzepte präzise zu formalisieren. Wir behandeln also eher eine „mathematische Ausdruckskraft“ als eine „algorithmische Ausdruckskraft“. Gallina setzt dabei auf starke Typisierung, um Inkonsistenzen zu vermeiden. Das Typsystem verhindert logische Widersprüche, da jeder Ausdruck wohl-typisiert sein muss, wobei die Typen wiederum garantieren, dass nur sinnvolle Operationen möglich sind. So ist es zum Beispiel nicht möglich, ein `N` mit einem `bool` zu multiplizieren. Durch die Typenhierarchie wird verhindert, dass keine Selbstreferenzen entstehen können und es wird zudem zwischen `Prop`, den logischen Formeln, und `Type`, den Datentypen unterschieden. Die Typprüfung findet statisch, das heißt zur Compile-Zeit nicht zur Laufzeit statt, um so Fehler vor der Ausführung aufdecken zu können. Dies bietet eine höhere Sicherheit für beweiskritische Anwendungen. Diese Typprüfung entspricht also der Beweisverifikation.

Gallina stellt zusammen mit Coq also die Benutzerschnittstelle zum CIC dar. Der CIC ist dabei die formale Grundlage und Gallina deren konkrete Syntax. Das bedeutet, dass jeder Gallina-Ausdruck intern in CIC-Terme übersetzt wird. Gallina-Typen entsprechen direkt den CIC-Typen, Polymorphie und abhängige Typen kommen auch direkt aus dem CIC und die Typprüfung in Gallina ist die des CIC. Es gibt aber auch Erweiterungen in Gallina, wie die **`Record`**-Typen, welche dann auf induktive Typen im CIC abgebildet werden. Demnach ist die Bedeutung jedes Gallina-Ausdrucks durch dessen CIC-Interpretation definiert und die logische Konsistenz und Grenzen von Gallina basieren auf der Konsistenz und den Grenzen des CIC.

Kategorie	Beispiele
Konstanten	`z, O, true, nil`
Variablen	`x, n, f, A`
Let-Bindungen	`let x := e1 in e2`
Lambda-Abstraktionen	`fun x => x + 1,`
	`fun (n : N) => S n`
Funktionsanwendungen	`plus n m`
Pattern-Matching	`match n with`
	`| z => m`
	`| S p => S (add p m)`
	`end`

Tabelle 1: Übersicht über Terme

3.3.1 Syntax und Hauptsprachelemente

Da wir in vorherigen Beispielen schon einige Syntaxelemente genutzt haben, möchten wir diese nun noch einmal kategorisieren und genauer benennen und werden dazu weiter relevante Elemente vorstellen. Gallina unterscheidet dabei zwischen verschiedenen Arten von Ausdrücken, welche wir in zwei Hauptkategorien einteilen können. Terme, Ausdrücke, welche berechnet werden, und Typen, die Klassifikation dieser Terme. Die Tabelle 1 für Terme und die Tabelle 2 für Typen geben einen Überblick über die wichtigsten Konstrukte, welche ebenfalls eine hohe Relevanz für die spätere Formalisierung des KLM-Theorems haben.

Die Terme repräsentieren ein konkretes Verhalten oder Berechnungen. Konstanten sind dabei unveränderliche vordefinierte Werte, welche direkt ohne Parameter verwendet werden können, wie unsere Konstante `z`, welche die natürliche Zahl Null darstellt. Hingegen können Variablen mit beliebigen Werten eines bestimmten Typs während des Beweisens belegt werden. So kann `n` eine beliebige natürliche Zahl sein, `f` eine Variable für eine Funktion, oder `A` eine Variable für einen Typen sein. Wir bezeichnen `n` auch als Wertevariable, `f` als Funktionsvariable und `A` als Typenvariable, was uns demnach die Formulierung von Aussagen und parametrischer Definitionen ermöglicht. Variablen werden innerhalb eines Bereichs (*Scope*) durch Quantoren, lokale Definitionen und Abstraktionen gebunden, was bedeutet, dass Variablen nicht global existieren, sondern immer nur in einem bestimmten Kontext eingeführt und dort verwendet werden können. Die Let-Bindung ermöglicht uns dann lokale Definitionen. Dabei wird in dem Beispiel der Wert von `e1` an `x` für die Auswertung von `e2` gebunden. Lambda-Abstraktionen, mit dem Bezug auf das λ-Kalkül, definieren anonyme Funktionen ohne explizite Benennung, was die Definition von Funktionen direkt dort möglich macht, wo sie benötigt werden. Es werden also Funktionen in einem lokalen Scope definiert, ähnlich wie bei der let-

Kategorie	Beispiele
Einfache Typen	`N, bool`
Parametrisierte Typen	`list A, prod A B`
Produkttypen	`prod A B`
Summentypen	`option A, sum A B`
Funktionstypen	`N -> bool, N -> N -> N`
Abhängige Typen	`(forall n : N, vector A n)`
Propositionale Typen	`Prop, le n m`

Tabelle 2: Übersicht über Typen

Bindung für Werte. In den Beispielen kann `fun x => x + 1` verwendet werden, um eine Nachfolgefunktion darzustellen, hier wird einfach ein Wert erhöht. Dies ist ähnlich wie bei `fun (n: N) => S n` nur wird hier explizit der Typ von `n`, nämlich `N` angegeben, um klarzustellen, dass `n` eine natürliche Zahl ist, was es ermöglicht, den Konstruktor `S` von dem Typ `N` anzuwenden. Das Pattern-Matching hatten wir bereits vorgestellt. Es ermöglicht Fallunterscheidungen basierend auf der Struktur eines Werts. Dabei ist zu beachten, dass alle möglichen Konstruktor-Varianten behandelt werden müssen. In unserem Beispiel haben wir für die natürlichen Zahlen zwischen dem Fall `z` und `S n` unterschieden.

Auch Funktionsanwendungen stellen eine wichtige Kategorie von Termen dar. Dabei wird eine Funktion auf konkrete Argumente zu der Berechnung eines Ergebnisses angewendet. Für das Beispiel `plus n m` bedeutet dies, dass die Addition auf die Werte `n` und `m` angewendet wird ($n + m$). Hierbei ist die Syntax in Coq sehr einfach gehalten. Funktionsnamen werden gefolgt von Argumenten, welche mit Leerzeichen getrennt sind, dargestellt.

Typen klassifizieren Terme und garantieren so deren konsistente Verwendung. Einfache Typen repräsentieren dabei grundlegende Datenstrukturen ohne Parameter oder Abhängigkeiten. Dies betrifft unseren Typ `N` und den Typ `bool` für Wahrheitswerte. Wir bezeichnen diese als grundlegend, da diese die Basis für komplexere Typenkonstruktionen bilden. Parametrisierte Typen sind wiederum Typen, welche durch einen oder mehrere Typparameter angepasst werden können. In dem Beispiel stellt `list A` eine Liste von Elementen des Typs `A` dar. Dies ist ähnlich zu den Funktionsanwendungen der Terme, jedoch werden als Parameter anstatt Werten Typen übergeben. Zu dieser Kategorie gehören auch die Produkt- und Summentypen. Produkttypen werden in Coq mit dem Schlüsselwort `prod` definiert und Summentypen können entweder mit `option` oder `sum` angegeben werden. Summentypen drücken die Wahl zwischen zwei möglichen Werten aus. Dabei steht `option A` für einen Wert von Typ `A` oder `None` und `sum A B` für einen Wert, entweder vom Typ `A` oder vom Typ `B`. Funktionstypen beschreiben Abbildungen von einem Typen auf einen anderen. So ist `N -> bool` eine Abbildung von natürlichen Zahlen auf Wahr-

heitswerte. Es ist ebenfalls möglich, mehrere Argumente anzugeben, wie `N -> N -> N` zeigt. Dabei entspricht dies (N -> (N -> N)), also einer Funktion, die einen Wert von Typ `N` nimmt und eine weitere Funktion `N -> N` zurückgibt, welche dann wiederum einen weiteren Wert vom Typ `N` nimmt und dann das Ergebnis vom Typ `N` zurückgibt. Abhängige Typen sind Typen, von denen die Definition von Werten abhängt und nicht nur von anderen Typen. Das Beispiel `forall n : N, vector A n` beschreibt einen Vektor vom Typ `A` mit der Länge `n`, wobei `n` vom Typ `N` ist. Das Ergebnis von `vector A n` ist also vom Wert `n` abhängig. Propositionale Typen werden in Coq verwendet, um logische Aussagen und Beweise darzustellen. Der Typ `Prop` ist der Typ aller beweisbaren Aussagen. Der Typ `le n m` ist der Typ einer Aussage über die Eigenschaft „kleiner gleich“ von `n` und `m`, wobei der Wert des Beweises für diese Aussage zeigt, dass `n` kleiner oder gleich `m` ist. Wenn wir also zeigen wollen, dass `n` kleiner gleich `m` ist, würden wir einen Wert vom Typ `le n m` erzeugen.

3.3.2 Module und Strukturierung

Bei der Modularisierung des Formalierungsprojekts wird die Trennung von Konzepten in logisch zusammenhängende Einheiten ermöglicht und gleichzeitig erhöht dies die Wiederverwendbarkeit des Codes in unterschiedlichen Beweisen. Wir reduzieren daher die Komplexität der Formalisierung durch Kapselung und Abstraktion und erleichtern damit auch die Wartung und das Fortführen unserer Formalisierungen.

Wenn wir von Modularisierung sprechen, drücken wir damit aus, dass der Code in Module zur Codeorganisation aufgeteilt wird. Module sind dann dabei eine Art Container für Definitionen, Theoreme und auch Beweise, welche ebenfalls wiederum geschachtelt auftreten können und ihren eigenen Namespace besitzen. Coq stellt dabei die Syntax **`Module`** `module_name.` ...**`End`** `module_name.` zur Verfügung [2, 20]. Um die Inhalte dann in einem Formalisierungsprojekt nutzen zu können, werden die Module über **`Import`** `module_name` importiert oder direkt über das Modul durch `module_name.inhalt` angesprochen, wodurch kein expliziter Import benötigt ist.

Um Schnittstellen (*Interfaces*) für die Module zu definineren bietet Coq die Syntax **`Module Type`** `interface_name`....**`End`** `interface_name.` Dabei wird durch `Modultypen` festgelegt, welche Bestandteile, wie Typen, Konstanten oder Beweise, ein Modul bereitstellen muss, ohne die konkrete Implementierung vorzugeben. Modultypen werden in Coq in dem jeweiligen Modul durch **`Module`** `module_name : interface_name.` ...**`End`** `module_name.` angegeben [20], und ermöglicht damit verschiedene Implementierungen derselben Schnittstelle.

Eine weitere nützliche Eigenschaft in Coq stellt der **`Record`** dar, womit wir zusammengehörige Konzepte und deren Daten und Eigenschaften bündeln können. Ein Record kann in Coq mit dem Recordnamen, den Feldern und deren Typen durch **`Record`** `record_name := field`$_1$ `: Typ`$_1$`;` ...`field`$_n$ `: Type`$_n$`.` definiert

werden [20]. Um auf die Felder eines Records zuzugreifen, wird die Punktnotation auf einer Instanz des Record verwendet. Um demnach auf das erste Feld im Record zuzugreifen, kann zum Beispiel `record_instance.(field`$_1$`)` genutzt werden.

Für die Trennung des Formalisierung des KLM-Theorems in logische Einheiten werden diese Eigenschaften von Coq genutzt. Wir werden im späteren Verlauf Basisklassen für System C und die kumulativen Konsequenzrelationen, sowie für kumulative Modelle und deren Eigenschaften und Module für den Beweis des Repräsentationstheorems vorstellen. Diese Strukturierung bietet Vorteile für die Beweiswartung und orientiert sich an der mathematischen Strukturierung von Kraus, Lehmann und Magidor [11]. Außerdem erleichtert die Aufteilung in Module die schrittweise Verifikation und ermöglicht die unabhängige Entwicklung und Validierung einzelner Komponenten [3, 22, 20]. Auch eine potenzielle Erweiterung, beispielsweise auf System P, wird dadurch erleichtert.

Um dem Benutzer die Beweisführung zu vereinfachen, bietet Coq zusätzlich selber bestimmte Automatisierungstechniken, die sogenannten *Tactics*, an. Aufgrund dieser Eigenschaft werden Beweise in Coq auch als *semi-automatische* Beweise bezeichnet.

3.4 Beweisführung mit Taktiken

Die Taktiken dienen dabei als Anweisungen für die schrittweise Transformation der Beweisziele. Das bedeutet konkrekt, dass wir über die Taktiken eine Beweisstrategie vorgeben, wobei Coq dann die Korrektheit verifiziert. Durch das Einsetzten von Taktiken wird der Beweis dann Stück für Stück abgearbeitet, wobei die Taktiken den Beweiszustand verändern. Dies geschieht so lange, bis alle Beweisziele gelöst sind.

Der Beweiszustand gibt dabei den aktuellen Fortschritt des Beweises an und besteht aus dem Kontext und den Beweiszielen. Der Kontext enthält dabei alle verfügbaren Hypothesen, Definitionen und Variablen des Beweises, und jedes Beweisziel besteht aus einem Typ, welcher bewiesen werden soll. Dabei gilt, dass wir in einem Beweiszustand auch mehrere offene Beweisziele haben können, welche dann ebenfalls nach einander gelöst werden sollen. Je nach der genutzten Entwicklungsumgebung (IDE) wird der Beweiszustand etwas anders dargestellt, aber meist wird der Kontext und die Beweisziele explizit dargestellt, um die Übersicht über den Beweis behalten zu können.

Der initiale Beweiszustand enthält dabei nur das zu beweisende Theorem als Beweisziel und keine lokalen Hypothesen. Danach liegt es an uns, die korrekten Taktiken für die Beweisführung anzuwenden und die dadurch neu erzeugten Beweisziele schrittweise zu beweisen. Dies stellt genau die interaktive Beweisführung dar, welche wir zu Beginn angesprochen hatten. Um den Beweiszustand zu verändern, implementieren die Taktiken eine bestimmte Transformation. Das können zum Beispiel Taktiken sein, die ein Beweisziel in Teilziele aufteilen, das Hinzufügen neuer Hypothesen ermöglichen und auch das Lösen eines Ziels unter bestimmten Bedin-

gungen automatisch durchführen. Dabei ist auch zu beachten, dass Taktiken ebenfalls fehlschlagen können, wenn eine Transformation nicht anwendbar ist.

In Coq wird ein Beweis mit dem Schlüsselwort `Proof.` begonnen und mit `Qed.` abgeschlossen. In einer IDE, welche die Darstellung des Beweiszustands unterstützt, kann der Beweis schrittweise durchlaufen werden, wie bei einer Art von Debugging. Dies ermöglicht es, Veränderungen des Beweiszustands zu überwachen und nachzuvollziehen. Die Darstellung des Beweiszustands ist bei der interaktiven Beweisführung ein kritischer Punkt, denn dies gibt Hinweise darauf, welche Taktiken im nächsten Schritt sinnvoll anzuwenden sein könnten.

3.4.1 Taktiksprache und deren Anwendung

In Coq gibt es eine Vielzahl von verfügbaren Taktiken [21] von denen wir folgend einige relevante vorstellen werden. Um einen Überblick zu verschaffen, haben wir die Taktiken nach ihrem Funktionsbereich kategorisiert. Tabelle 3 zeigt eine Zusammenstellung der Taktiken, die bei der Beweisführung in Coq und bei der Formalisierung des KLM-Theorems häufig zum Einsatz kommen.

Die Gruppe der logischen Operationen umfasst Taktiken, welche für die Transformation logischer Formeln, wie Prämissen, Implikationen und Quantoren, eingesetzt werden. Der erste Schritt in einem Beweis sind üblicherweise die `intro` und `intros` Taktiken, da diese Hypothesen in den Kontext einführen. Die Taktik `apply` wird dabei genutzt, um ein Theorem oder eine Hypothese auf ein aktuelles Ziel anzuwenden, wohingegen `exact` und `assumption` direkt eine Lösung liefern, wenn eine passende Hypothese oder ein passender Term aus dem Kontext verfügbar ist.

Taktiken, die genutzt werden können, um Äquivalenzen in Beweisen zu zeigen, haben wir in die Gruppe der Äquivalenzbeweise einsortiert, da diese verschiedene Aspekte des Umgangs mit Äquivalenzen unterstützen. Die Taktik `reflexivity` löst dabei einfache Gleichungen der Form $n = n$ und um einen Term mithilfe von Gleichungen umzuformulieren und zu substituieren, wird die Taktik `rewrite` angeboten. Die Taktik `congruence` automatisiert einfache Äquivalenzbeweise durch die Kombination mehrerer Äquivalenzregeln.

Die Gruppe der strukturellen Manipulation umfasst Taktiken, die die Struktur von Beweiszielen oder Hypothesen anpassen. Dabei zerlegt die Taktik `split` konjunktive Ziele in Teilziele, die bei der Behandlung von Konjunktionen in Regeln wie zum Beispiel Cautious Monotonicity relevant sein werden. Taktiken wie `left` und `right` wählen dann eine Seite der Disjunktion aus, zum Beispiel bei der Formalisierung von disjunktiven Prämissen. Die Taktik `destruct` führt Fallunterscheidungen durch, zum Beispiel für Hypothesen in Beweisen von System C Regeln, und `induction` wendet strukturelle Induktion an, um Beweise über induktiv definierte Strukturen, zum Beispiel auf die Struktur von kumulativen Konsequenzrelationen, zu führen.

Die Taktiken für die Kontextmanipulation helfen uns, den Beweiskontext übersichtlich zu halten. So führt `assert` neue Hypothesen oder Zwischenziele ein, um

zum Beispiel Eigenschaften von kumulativen Modellen zu beweisen. Die Taktik **clear** entfernt irrelevante Hypothesen, **rename** benennt Variablen und Hypothesen um, und **generalize** sowie **specialize** verallgemeinern oder spezialisieren Hypothesen, indem spezifische Terme durch Variablen ersetzt werden, um diese in anderen Kontexten nutzen zu können oder um eine allgemeine Hypothese auf einen konkreten Fall zu beschränken. Um Hypothesen in dem Kontext zu ordnen, können wir die Taktik **move** anwenden, was bei komplexen Beweisen hilfreich sein kann.

Die Gruppe der Automatisierung umfasst Taktiken, die Beweise teilweise automatisieren. Dabei lösen **auto** und **eauto** einfache Ziele mit vordefinierten Hinweisen, zum Beispiel in Beweisen für die Reflexivität. Die Taktik **tauto** beweist logische Tautologien, und **intuition** wird genutzt um komplexe logische Ausdrücke automatisch in deren Bestandteile zu zerlegen, also zum Beispiel um Disjunktionen, Konkunktionen oder Implikationen aufzuspalten. Dabei werden Einführungs- und Eliminierungs-Regeln angewendet um Teilziele zu vereinfachen.

Taktik-Kombinatoren (LTCF-Kombinatoren) ermöglichen eine flexible Beweisführung. Das Semikolon `;` führt Taktiken nacheinander aus, `| |` kombiniert Alternativen, und `try`, versucht Taktiken optional auszuführen. Der Kombinator `repeat` wiederholt Taktiken, bis diese Fehlschlagen oder zu keiner Änderung mehr führen und `solve` versucht, Ziele vollständig zu lösen, in dem es Taktiken anwendet. Sollten Ziele offen bleiben, schlägt `solve` fehl. Der Kombinator `first` wählt die erste erfolgreiche Taktik aus einer Liste und versucht dann die Taktik **auto** anzuwenden.

3.5 Semi-automatisches Beweisen

Bei dem semi-automatischen Beweisen ist gefordert, eine gute Balance zwischen manueller Beweisführung durch den Benutzer und der Automatisierung mit Taktiken zu finden. Dies erfordert wiederkehrende Beweismuster, welche sich durch Taktiken automatisieren lassen, zu erkennen, sowie Unterbeweise technischer Natur, wie zum Beispiel Umformungsschritte, von konzeptionellen Kernbeweisen zu isolieren. Dabei unterscheiden wir zwischen trivialen Teilzielen, die automatisch gelöst werden, und den komplexen Zielen, welche manuell geführt werden. Für das KLM-Theorem würden so zum Beispiel die Eigenschaften kumulativer Konsequenzrelationen automatisch geprüft werden können, während die Hauptschritte des Beweises manuell geführt werden.

Es ist ratsam, komplexere Beweise strategisch zu zerlegen. So sollte der Gesamtbeweis in logische Einheiten mit klaren Abhängigkeiten aufgeteilt werden, was wiederum zu Zwischenzielen mit überschaubarer Komplexität führt. Dabei ist es relevant, eine sinnvolle Reihenfolge für diese Teilbeweise festzulegen.

In Coq sind Automatisierungstechniken domänenspezifisch. Das bedeutet konkret, dass Taktiken für bestimmte mathematische Bereiche spezialisiert sind, zum Beispiel die Taktik **ring** für Algebra und **lia** für lineare Arithmetik.

Es gibt aber auch die Möglichkeit, eigene Taktiken mithilfe der Taktiksprache *Ltac* zu definieren. In solch einer Definition können dann wiederum weitere Tak-

Taktik	Beschreibung
Logische Operationen	
`intro`, `intros`	Hypothesen für Implikationen/Quantoren einführen
`apply`	Theorem auf Beweisziel anwenden
`exact`	Term direkt als Beweisziel angeben
`assumption`	Passende Hypothese aus Kontext nutzen
Äquivalenzsbeweise	
`reflexivity`	Äquivalenz durch Reflexivität beweisen
`rewrite`	Term mit Gleichung umschreiben
`congruence`	Einfache Äquivalenzziele automatisch lösen
Strukturelle Manipulation	
`split`	Konjunktion in Teilziele aufteilen
`left`, `right`	Seite einer Disjunktion auswählen
`destruct`	Fallunterscheidung nach Termstruktur
`induction`	Induktion auf induktive Typen anwenden
Kontextmanipulation	
`assert`	Neue Hypothese oder Zwischenziel einführen
`clear`	Unnötige Hypothesen entfernen
`rename`	Variablen/Hypothesen umbenennen
`generalize`	Term verallgemeinern
`specialize`	Hypothese spezialisieren
`move`	Hypothesen im Kontext umordnen
Automatisierung	
`auto`	Beweis automatisch mit Hints suchen
`eauto`	Erweiterte Suche mit mehr Tiefe
`tauto`	Tautologien der Aussagenlogik beweisen
`intuition`	`tauto` mit Zerlegung kombinieren
Taktik-Kombinatoren (*LCF combinators*)	
`;`	Taktiken nacheinander ausführen
`||`	Alternative Taktiken ausprobieren
`try`	Taktik optional ausführen
`repeat`	Taktik wiederholen bis Fehlschlag
`solve`	Ziel vollständig lösen versuchen
`first`	Erste erfolgreiche Taktik aus Liste wählen

Tabelle 3: Zusammenstellung zentraler Taktiken und Kombinatoren für die interaktive Beweisführung in Coq

tiken verwendet werden. Dabei ist es durch Ltac möglich, `match`-Konstrukte zu nutzen, um Taktiken abhängig von Beweiszielen auszuführen. Außerdem werden Konstrukte wie `repeat` zur Verfügung gestellt, um Schleifen zu erzeugen, womit eine Taktik so lange zu wiederholt werden kann, bis diese nicht mehr anwendbar ist. In Ltac kann sogar der Beweisfluss mit den Konstrukten `try` und `fail` gesteuert werden. Durch diese Taktiksprache wird nochmals die Wiederverwendbarkeit verstärkt und es ermöglicht oft hintereinander genutzte Taktiken in einer einzigen Taktik zu definieren, was wiederum die Komplexität reduziert.

Auch bei der Verwendung von Taktiken gelten jedoch die bereits zuvor angesprochenen Grenzen der vollautomatischen Beweise, welche jedoch stückweit durch menschliche Führung überwunden werden können. Dies wiederum setzt voraus, dass für die zu führenden Beweise ein tiefes semantisches Verständnis existiert.

4 Formalisierungsansatz

Für die Formalisierung des KLM-Theorems haben wir uns für einen modularen Aufbau entschieden, dabei gibt es verschiedene Coq-Module für die jeweiligen unterschiedlichen Aspekte der Formalisierung. Diese Entscheidung basiert auf der Umsetzung der mathematischen Definitionen aus der Arbeit von Kraus, Lehmann und Magidor [11]. Wie bereits zahlreich besprochen, beschränken wir uns in dieser Arbeit auf die Formalisierung des KLM-Theorems mit propositionaler Logik, um die Komplexität, im Vergleich zu einer prädikatenlogischen Formulierung, möglichst gering zu halten. Dies spiegelt sich ebenfalls in der Strategie zur Handhabung der Smoothness-Bedingung bei endlichen Mengen semantisch unterschiedlicher Formeln aus Kapitel 2.3.1 wider. Um die propositionale Logik in Coq zu implementieren, haben wir uns außerdem für eine bereits existierende Bibliothek entschieden, damit die Grundlagen nicht erneut implementiert werden müssen und der Fokus auf der Formalisierung des KLM-Theorems bleibt.

4.0.1 Aufbau des Beweises

Um die Formalisierung des KLM-Theorems in Coq zu veranschaulichen, zeigen wir in Abbildung 5 eine Übersicht der wichtigsten Bestandteile. Das KLM-Theorem, wie es von Kraus, Lehmann und Magidor in Theorem 3.25 [11] definiert wurde, sagt aus, dass eine kumulative Konsequenzrelation ($\Gamma : p \mid\sim q$) genau dann gilt, wenn sie in einem Modell semantisch erfüllt ist ($\Gamma \models p \mid\sim_W q$). Unsere Formalisierung in Coq beweist diese Äquivalenz. Die Grafik zeigt, wie wir die mathematischen Konzepte von Kraus, Lehmann und Magidor in Coq umgesetzt haben. Wir übernehmen aus deren Arbeit die kumulative Konsequenzrelation und die Lemmata 3.15 bis 3.24, welche die Eigenschaften des KLM-Theorems definieren. Die Grundlagen für den Beweis stellt die induktive Definition der kumulativen Konsequenzrelation `CumulCons` aus `KLM_Cumulative.v`, zusammen mit den fünf Regeln für Reflexivity, Left Logical Equivalence, Right Weakening, Cut und Cautious Monotonicity, welche als Konstruktoren definiert sind, dar. Die semantische Seite, basierend auf den kumulativen Modellen wird in `KLM_Semantics.v` definiert. Der Beweis des KLM-Theorems ist in zwei Module nach Kapitel 2.4 aufgeteilt. Ein Modul aus `KLM_Soundness.v` implementiert den Soundness Beweis, also kumulative Modelle erfüllen System C, und ein Modul übernimmt den Completeness Beweis aus `KLM_Completeness.v`, wo gezeigt wird, dass für jede kumulative Konsequenzrelation ein kumulatives Modell existiert.

Die Soundness des KLM-Theorems stellt dabei den weniger komplexen Teil des Beweises dar. Hier werden wir die Beweisstruktur entlang der fünf Regeln des System C aufbauen und die Soundness als Induktion über die Ableitungsregeln in `CumulCons` beweisen. Dafür werden wir weitere separate Hilfssätze für jede der fünf Regeln des System C entwickeln. Diese Hilfssätze sind in dabei in den Lemmata `soundness_reflexivity`, `soundness_LLE`, `soundness_RW`, `soundness_Cut` und `soundness_CM` definiert, wie es durch Lemma 3.24 [11] gefordert ist. Für den

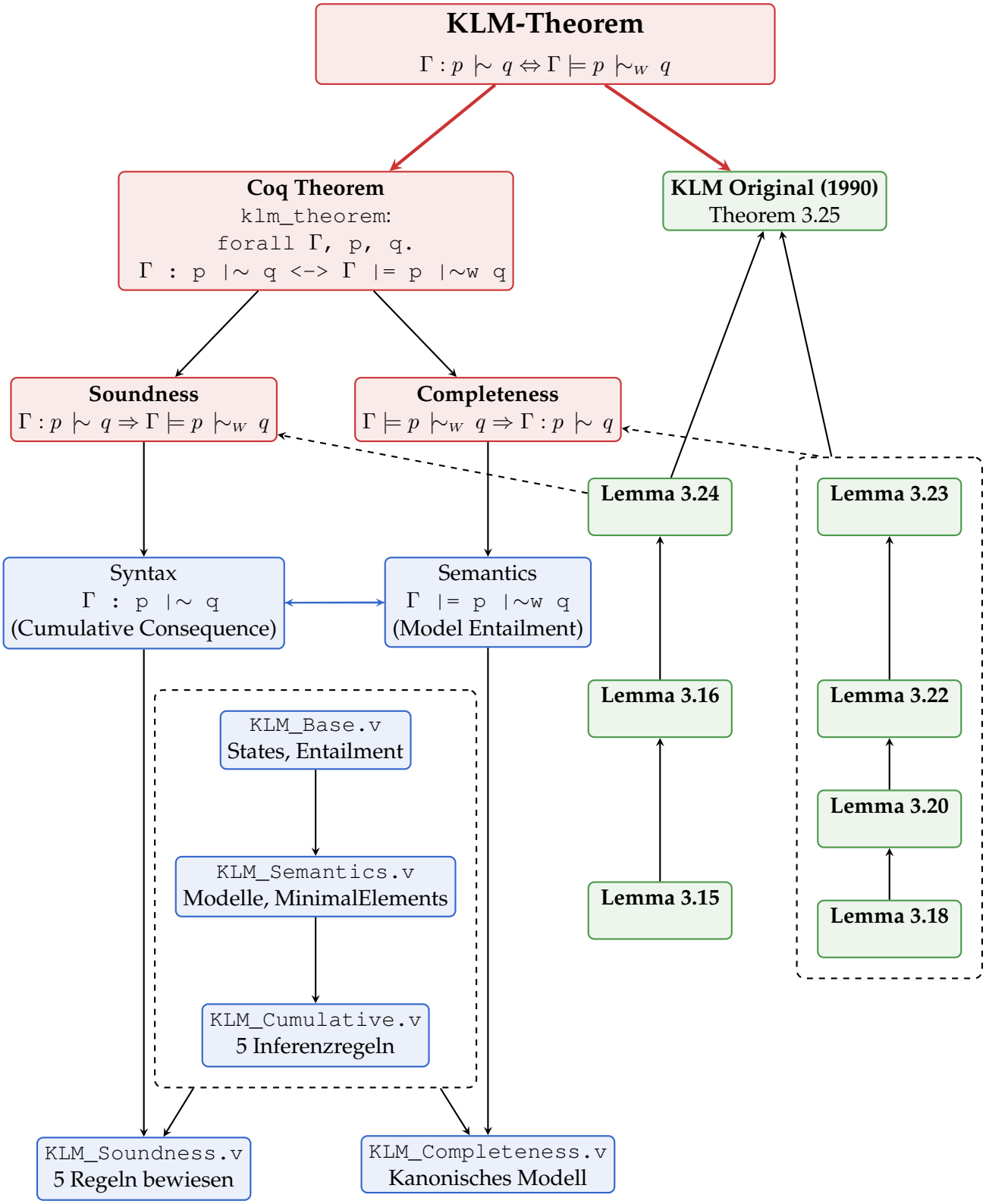

Abbildung 5: Übersicht der Formalisierung des KLM-Theorems in Coq

eigentlichen Beweis nutzen wir die strukturelle Induktion über den Aufbau von `CumulCons`. Für jeden Konstruktor, also jede der fünf Regeln des System C, wird individuell nachgewiesen, dass dessen Eigenschaft im kumulativen Modell gilt. Dabei wird die Induktionshypothese jeweils für die Teilableitungen angewendet.

Für den Completeness-Beweis werden wir ein kanonisches Modell konstruieren, welches genau die gegebene kumulative Konsequenzrelation repräsentiert, und orientieren uns hierbei an den Lemmata 3.17 bis 3.23 [11]. Dabei basiert die Konstruktion des Modells auf maximal konsistenten Mengen, welche als Zustände in dem Modell dienen. Außerdem zeigen wir in einem Beweisschritt, dass die Smoothness-Bedingung von dieser Konstruktion erfüllt wird und konstruieren formal die Präferenzrelation des Modells. Auch hier werden wir mehrere Hilfssätze einführen, welche die Eigenschaften maximal konsistenter Mengen und deren Verhalten in Bezug auf die kumulative Konsequenzrelation widerspiegelt.

Die Formalisierung dieses Teils ist komplexer als der Soundness Beweis und erfordert daher verschiedene Zwischenschritte.

4.0.2 Einbinden der Library für Propositionale Logik

Für die Implementierung der Propositionalen Logik haben wir uns dazu entschieden die Library von Dakai Guo und Wensheng Yu aus 2023 [8] einzubinden, welche uns die benötigten Grundlagen der Propositionalen Logik bereitstellt. Die für diese Arbeit relevanten Details befinden sich in den `base_pc.v`, `semantic.v` sowie `syntax.v` und `complete.v` Dateien. Durch die Nutzung der externen Library können wir uns vorrangig auf die eigentliche Formalisierung des KLM-Theorems fokussieren. Die wichtigsten Komponenten der Library ermöglichen dann die Definition von Formeln und Wahrheitswerten, das Formulieren von Ableitungsregeln und eine semantische Interpretation und Folgerungsbeziehung.

4.0.3 Überblick über die Formalisierungsschritte

Die Formalisierung wird in aufeinander aufbauenden, logisch voneinander abhängigen Phasen eingeteilt und in Modulen für eine bessere Übersicht definiert:

1. Definitionen der grundlegenden syntaktischen Elemente, wie die Konsequenzrelation.
2. Formalisierung des System C und die damit verbundenen fünf Regeln
3. Implementation der Smoothness-Bedingung als Axiom, unter der Berücksichtigung der Besonderheiten im propositionalen Fall.
4. Beweis der Soundness des KLM-Theorems.
5. Beweis der Completeness des KLM-Theorems.
6. Zusammenführung aller Module zum vollständigen Repräsentationstheorem.

4.1 Darstellung der Syntax

4.1.1 Kodierung propositionaler Formeln

Für die Formalisierung werden wir den vordefinierten Datentyp `Formula` aus der externen Library verwenden. `Formula` basiert auf drei Konstruktoren:

- `Var n`: Dies stellt eine atomare Aussagenvariable dar. Dabei ist `n` eine natürliche Zahl, welche als Bezeichner benutzt wird. `Var 0` wäre dann zum Beispiel die Aussagenvariable p_0.
- `Not f`: Dieser Konstruktor wird für die logische Negation einer Formel `f` genutzt. Dieses `Not` wird später noch durch eine Notation auf das Symbol $\neg$ abgebildet, um Negation als $\neg$`f` darstellen zu können.
- `Contain f1 f2`: Hier repräsentiert der Konstruktor die logische Implikation von Formel `f1` zu `f2`. Auch hier wird später die Notation $\rightarrow$ eingeführt, um die Implikation als `f1` $\rightarrow$ `f2` schreiben zu können.

Aus diesen drei Konstruktoren können dann, durch weitere entsprechende Notationen, auch andere logische Verknüpfungen wie die Konjunktion ($\wedge$), Disjunktion ($\vee$) und Äquivalenz ($\leftrightarrow$) definiert werden. Dies ist ausreichend für die Formalisierung des KLM-Theorems, da in dieser Arbeit das System C auf propositionaler Logik aufbaut, die kumulativen Konsequenzrelationen damit definierbar sind und die Syntax der Arbeit von Kraus, Lehmann und Magidor damit abbildbar ist.

Konkret bedeutet dies, dass wir `Formula` direkt in unsere Formalisierung integrieren können und damit auch direkt die vorgegebene Typstruktur verwenden können. Damit decken diese propositionalen Formeln die Grundlagen für die Definition der kumulativen Konsequenzrelation und der kumulativen Modelle und somit für die Formalisierung der Beweisschritte im KLM-Theorem ab. Zudem ist es ebenfalls nicht notwendig, die bestehende Definition strukturell zu erweitern, da diese bereits ausreichende Ausdruckskraft besitzt. Auch die Semantik der propositionalen Logik ist bereits in `semantic.v` formuliert und kann von uns durch `Formula` mitgenutzt werden.

4.1.2 Induktive Definition der Syntax

Auch die induktiven Definitionen, auf deren definierte induktive Struktur wir direkt Zugriff haben, werden von besonderer Bedeutung für die Beweisführung sein, da sie strukturelle Induktion über Formeln möglich machen, was wiederum die Grundlage für Beweise über alle Formeln einer bestimmten Form ist.

Jede Formel ist dabei eindeutig durch ihre Konstruktion über ihre Konstruktoren bestimmt. Dabei garantiert die Injektivität der Konstruktoren die Eindeutigkeit, denn es gilt `Var` $\neq$ `Not` $\neq$ `Contain`. Dies ist für exakte Fallunterscheidungen in Beweisen, wie zum Beispiel bei dem Pattern-Matching aus Kapitel 3.2.4,

entscheidend. Jede Formel fällt dabei genau in eine Kategorie der Fallunterscheidung, und dies dient damit als Grundlage für den Beweis, dass Eigenschaften für alle Formeln gelten, indem alle möglichen Konstruktionswege abgedeckt werden. Das heißt, durch das Pattern-Matching kann garantiert werden, dass alle möglichen Fälle betrachtet werden, was für die Vollständigkeit von Beweisen notwendig ist.

Es gibt noch weitere Definitionen für abgeleitete Operatoren. Darunter die Konjunktion `p ∧ q`, die als `¬(p → ¬q)` definiert wird, die Disjunktion `p ∨ q`, welche wiederum als `¬p → q` definiert ist und die Äquivalenz `p ↔ q`, definiert als bidirektionale Implikation durch `(p → q)∧(q → p)`.

4.1.3 Darstellung von Wahrheitswerten

Die Library stellt ebenfalls eine Implementierung von Wahrheitswerten und ihrer Semantik zur Verfügung. Hierbei wird der boolesche Typ `bool` mit den Werten `true` und `false` genutzt. Durch implementierte boolesche Funktionen wird die Handhabung der Wahrheitswerte vereinfacht. Wie zum Beispiel die Bewertungsfunktion `value`. Diese dient der semantischen Interpretation für die Zuordnung von Wahrheitswerten zu Formeln und ist eine rekursive Definition auf der Formelstruktur. Die semantischen Eigenschaften von `value` sind in `keep_not` und `keep_contain` definiert. Diese spezifizieren die korrekte Verhaltensweise der Bewertungsfunktion, indem die Negation und Implikation korrekt interpretiert werden. Für `keep_not v` bedeutet das, dass der Wahrheitswert einer negierten Formel die Negation des Wahrheitswerts der Formel ist. Während `keep_contain` genutzt wird, damit der Wahrheitswert den Regeln der klassischen Logik folgt. Nur Funktionen, welche beide dieser Eigenschaften erfüllen, sind semantisch korrekt. Dies gibt die Grundlage für die Definition von Tautologien und semantischem Entailment.

Die Tautologie `Tautology p := forallv, valuev → vp = true` formalisiert Formeln, die unter jeder Bewertung wahr sind. Und das semantische Entailment, gegeben als $\Gamma \models p$, definiert für Formeln, das `p` semantisch aus Γ folgt und bildet damit eine Verbindung zwischen Semantik und Syntax. Die Notation Γ wird dabei eingeführt, um eine Menge von Formeln zu repräsentieren, die als vorausgesetzt oder bekannt sind. Wir nennen Γ auch eine Wissensbasis oder den Kontext. Formalisiert ist Γ als `Ensemble Formula`, welches auch leer sein darf. Es definiert, dass bei jeder Bewertung `v`, welche alle Formeln in Γ zu `true` auswertet, auch `p` wahr ist. Dies ist essenziell für die semantische Interpretation kumulativer Konsequenzrelationen und erlaubt die Formalisierung „`p` folgt aus Γ unter Berücksichtigung der minimalen Modelle“.

4.2 Formalisierung von System C

Die Formalisierung des Systems C, wie es von Kraus, Lehmann und Magidor [11] definiert wurde, bildet eine der Grundlagen der Formalisierung des KLM-Theorems. System C stellt die minimalen Anforderungen an nichtmonotones Schließen dar

und wird in Coq durch den induktiven Typ `CumulCons` umgesetzt, der die kumulative Konsequenzrelation repräsentiert. Mit einer Wissensbasis vom Typ `Ensemble Formula` und den fünf Grundregeln (`Ref`, `LLE`, `RW`, `Cut`, `CM`) können wir die nichtmonotone Logik präzise in Coq abbilden. Die folgenden Abschnitte beschreiben die Implementierung der Regeln, eine intuitive Notation, eine benutzerdefinierte Taktik zur Vereinfachung von Beweisen sowie Beispiele für Ableitungen im System C.

4.2.1 Formalisierung der fünf Grundregeln

Wir implementieren das System C zusammen mit den fünf Grundregeln als den induktiven Typen `CumulCons`, welcher direkt die kumulative Konsequenzrelation darstellt. Dies ermöglicht es, die Inferenzregeln direkt als die Konstruktoren des induktiven Typen abzubilden. Außerdem unterstützen wir damit die strukturelle Induktion für Beweise, und Coq generiert dafür wiederum die Induktionsprinzipien. Zudem haben wir das `Ensemble Formula`, statt einer Liste genutzt, um ebenfalls Mengeneigenschaften direkt nutzen zu können und somit auch Duplikate oder eine Reihenfolgeproblematik zu verhindern. Der Typ der Relation `CumulCons` nimmt drei Parameter. Zuerst Γ vom Typ `Ensemble Formula`, welcher unsere Wissensbasis darstellt, welche die gemeinsamen Annahmen enthält und danach zwei Formeln `p` und `q` vom Typ `Formula`, welche die Prämisse und die Konklusion der nichtmonotonen Folgerung darstellen. Der Rückgabetyp **`Prop`** wird dabei verwendet, um Propositionen, die bewiesen werden können, und die Relation als logische Eigenschaft, statt einer berechenbaren Funktion, darzustellen. Entgegen der mathematischen Definition von Kraus, Lehmann und Magidor führen wir die Wissensbasis ein, um den Kontext, welcher sonst nur implizit angenommen wird, in Coq explizit darstellen zu können, da Coq präzise Definitionen erfordert. Damit können wir in Coq Formeln voraussetzen, welche als Annahmen oder Hintergrundwissen dienen.

Für die Konstruktornamen haben wir uns ebenfalls an der mathematischen Notation orientiert.

Definition 8. *Kumulative Konsequenzrelation und System C in Coq*

```
1  Inductive CumulCons :
2    Ensemble Formula -> Formula -> Formula -> Prop :=
3    | Ref : forall Γ p,
4        CumulCons Γ p p
5    | LLE : forall Γ p q r,
6        In Formula Γ (p ↔ q) ->
7        CumulCons Γ p r ->
8        CumulCons Γ q r
9    | RW : forall Γ p q r,
10       In Formula Γ (p → q) ->
11       CumulCons Γ r p ->
12       CumulCons Γ r q
```

```
13    | Cut : forall Γ p q r,
14        CumulCons Γ (p ∧ q) r ->
15        CumulCons Γ p q ->
16        CumulCons Γ p r
17    | CM : forall Γ p q r,
18        CumulCons Γ p q ->
19        CumulCons Γ p r ->
20        CumulCons Γ (p ∧ q) r.
```

Der `Ref`-Konstruktor ist durch universelle Quantifizierung über alle Formeln implementiert. Dabei wird der Parameter `p` zweimal genutzt, um die Reflexivität auszudrücken. Zu beachten ist, dass wir hier keine Vorbedingung formulieren, da die Reflexivität immer gelten soll.

Für den `LLE`-Konstruktor nutzen wir drei Formeln (`p`, `q`, `r`) für die Äquivalenz und Folgerung. Dabei sagt `In Formula Γ (p ↔ q)` aus, dass die Äquivalenz ein Element der Wissensbasis sein muss und der Typ der Wissensbasis `Formula` sein soll. Dies erfüllt die Bedingung der LLE-Regel, dass die Äquivalenz als Voraussetzung bekannt sein muss, bzw. wir wissen, dass `p` und `q` äquivalent sind. Als zweite Prämisse nutzen wir nach der LLE-Regel und der induktiven Struktur den rekursiven Aufruf von `CumulCons`, um darzustellen, dass die neue Folgerung `CumulCons Γ q r` von einer bereits bestehenden Folgerung `Γ p r` abhängt.

Die Definition des `RW`-Konstruktors hat eine ähnliche Struktur wie schon der `LLE`-Konstruktor, nur mit einer Implikation statt der Äquivalenz. Wir verwenden `In Formula Γ (p → q)` wieder für die Implikation aus der Wissensbasis. Danach erfolgt ebenfalls der rekursive Aufruf von `CumulCons Γ r p` als weitere Prämisse für die Folgerung `CumulCons Γ r q`.

Der `Cut`-Konstruktor definiert zwei rekursive Aufrufe von `CumulCons` als Prämissen. Dabei verwenden wir bei `CumulCons Γ (p ∧ q) r` die Konjunktion aus der externen Library. Mit den beiden Prämissen und der daraus ableitbaren Folgerung beschreiben wir die Transitivität der Cut-Regel aus.

Letztlich definieren wir noch den `CM`-Konstruktor, welcher die Eigenschaften der Cautious Monotonicity repräsentiert. Dabei führen wir wieder wie bei dem `Cut`-Konstruktor zwei Prämissen ein, die zu der Folgerung `CumulCons Γ (p ∧ q) r` führen. Auch dies entspricht wieder wie bei allen anderen Konstruktoren der mathematischen Definition der Regeln aus Kapitel 2.2.1. Hierbei lässt sich gut erkennen, dass die Cut-Regel erlaubt, eine Konjunktion aus den Prämissen zu entfernen, während Cautious Monotonicity es ermöglicht, eine Konjunktion in die Prämissen einzuführen.

4.2.2 Definition kumulativer Konsequenzrelationen

Für die kumulative Konsequenzrelation führen wir ebenfalls eine intuitive Notation zur besseren Lesbarkeit ein.

Notation 1. *Die kumulative Konsequenzrelation*

```
Notation "Γ : p |∼ q" := (CumulCons Γ p q) (at level 80).
```

Mit dieser Notation sagen wir aus, dass unter der Wissensbasis Γ, `q` normalerweise aus `p` folgt. Der Parameter `at level N` bestimmt in Coq die Bindungsstärke des Operators, um festzulegen, wie die Notation in Ausdrücken geparst wird, also wie Coq Ausdrücke ohne Klammern interpretiert. Dabei stellen niedrigere Zahlen eine stärkere Bindung dar. Wir wollen hier die kumulative Konsequenzrelation als top-level Operator, welcher eine Beziehung zwischen einer Menge Γ und zwei Formeln, `p` und `q`, darstellen. Die schwache Bindung (`level 80`) stellt dabei sicher, dass andere Operatoren wie für die Implikation oder Äquivalenz innerhalb von `p` und `q` zuerst geparst werden, ohne dass weitere Klammern nötig werden. Für die Bindungsstärke orientieren wir uns an der Library von Guo und Yu [8].

4.2.3 Hilfssätze zu den Regeln

Wir stellen außerdem eine eigene Taktik, die mit Ltac formuliert ist, zur Verfügung. Diese dient der Vereinfachung von der Beweisführung mit kumulativen Konsequenzrelationen.

```
Ltac solve_cumul :=
  match goal with
  | |- CumulCons _ _ _ => constructor; solve_cumul
  | _ => try assumption
  end.
```

Diese Taktik wird verwendet, um einen passenden Konstruktor von `CumulCons` basierend auf dem Beweisziel anzuwenden. Unterziele werden dabei rekursiv, durch erneutes Anwenden der Taktik, gelöst und `assumption` wird verwendet, um verbleibende Ziele mit passenden Hypothesen zu beweisen. Sollte es jedoch keine passende Hypothese geben, würde ein Fehler auftreten, welchen wir mit `try` abfangen. Durch die Taktik `solve_cumul` können sich wiederholende Beweisschritte reduziert werden, was ebenfalls die Lesbarkeit komplexerer Beweise erheblich verbessert und können uns so auf die relevanten Aspekte des Beweises, statt auf technische Details, fokussieren.

Kommen wir nun zu Anwendungsbeispielen für zwei der bereits definierten Regeln. Zunächst stellen wir ein einfaches Beispiel für die Anwendung der Reflexivitätsregel vor.

Beispiel 11. *Einfache Anwendung von `solve_cumul` für Reflexivity*

```
Example simple_reflexivity :
  forall Γ,  Γ : (Var 0) |∼ (Var 0).
Proof.
  intros.
  solve_cumul.
Qed.
```

Wir nutzen hier `solve_cumul`, um direkt einen der Konstruktoren von `CumulCons` anzuwenden. Der korrekt zu nutzende Konstruktor wäre hier der Konstruktor `REF` und `solve_cumul` wählt dann diesen als passenden aus. Wenn wir den Beweis jedoch manuell führen wollten, würden wir `apply REF.` aufrufen. Der Vorteil hierbei ist, dass wir erstmal nicht direkt wissen müssen, welcher der passend anzuwendende Konstruktor ist.

Ein weiteres Beispiel zeigt die Anwendung der Regel für Cautious Monotonicity (`CM`), die es erlaubt, eine Konjunktion in die Prämisse einzuführen.

Beispiel 12. *Anwendung von `solve_cumul` für Cautious Monotonicity*

```
Example simple_CM :
  forall Γ p q r,
    Γ : p |~ q ->
    Γ : p |~ r ->
    Γ : (p ∧ q) |~ r.
Proof.
  intros.
  solve_cumul.
Qed.
```

In diesem Beispiel verwenden wir `solve_cumul`, um den `CM`-Konstruktor des induktiven Typen `CumulCons` anzuwenden. Die Taktik erkennt, dass die Voraussetzungen `Γ : p |~ q` und `Γ : p |~ r` gegeben sind, und wählt `CM`, um die Folgerung `Γ : (p ∧ q) |~ r` zu beweisen. Ein manueller Beweis würde `apply CM.` verwenden. Dieses Beispiel illustriert, wie `solve_cumul` die Beweisführung vereinfacht, was besonders in komplexeren Beweisen, wie der Korrektheit von `CM` in unserer Formalisierung, nützlich ist [11].

Ein komplexeres Beispiel kombiniert die Regeln für Cautious Monotonicity (`CM`) und Right Weakening (`RW`), um die Interaktion von Wissensbasis und nichtmonotonem Schließen zu zeigen.

Beispiel 13. *Kombination von `CM` und `RW`*

```
Example CM_RW_example :
  forall Γ p q r s,
    In Formula Γ (r → s) ->
    Γ : p |~ q ->
    Γ : p |~ r ->
    Γ : (p ∧ q) |~ s.
Proof.
  intros Γ p q r s H_impl H_p_q H_p_r.
  apply CM.
  - exact H_p_q.
  - apply RW with r.
```

```
12      + exact H_impl.
13      + exact H_p_r.
14  Qed.
```

In diesem Beispiel zeigen wir, wie `RW` und `CM` zusammenwirken, um eine komplexe Folgerung abzuleiten. Die Wissensbasis Γ enthält die Implikation `r` $\rightarrow$ `s`, und wir nehmen an, dass Γ `: p |~ q` und Γ `: p |~ r` gelten. Ziel ist es, Γ `: (p` $\wedge$ `q) |~ s` zu beweisen. Wir wenden `CM` an, um die Prämissen Γ `: p |~ q` und Γ `: p |~ s` zu nutzen. Für Γ `: p |~ s` verwenden wir `RW`, da `r` $\rightarrow$ `s` in Γ liegt und Γ `: p |~ r` gegeben ist. Alternativ könnte `solve_cumul` den `CM`-Konstruktor anwenden, aber `RW` erfordert explizite Angabe der Implikation. Ein praktisches Beispiel verdeutlicht dies: Sei `p` = „Es ist ein Vogel“, `q` = „Es hat Federn“, `r` = „Es hat Flügel“, `s` = „Es kann sich fortbewegen“ und `r` $\rightarrow$ `s` = „Alles mit Flügeln kann sich fortbewegen“ in Γ. Dann folgt, dass wenn Vögel üblicherweise Federn und Flügel haben, kann ein befiederter Vogel sich üblicherweise fortbewegen. Dieses Beispiel zeigt die Stärke der KLM-Regeln, da `RW` Wissen mit nichtmonotonem Schließen kombiniert und `CM` die Prämisse durch eine Konjunktion verstärkt.

4.3 Modellierung kumulativer Modelle

Für die Formalisierung des kumulativen Modells haben wir uns für einen Record-Typ entschieden. Dies gibt uns Vorteile gegenüber separaten Definitionen oder induktiven Typen. Die zusammengehörigen Komponenten des Modells sind in einem Record als benannte Felder gebündelt, auf welche durch automatische Projektion zugegriffen werden kann.

Definition 9. *Kumulative Modelle*

```
1  Record CumulModel : Type := {
2    States : Type;
3    Labeling : States -> State;
4    PreferenceRel : States -> States -> Prop;
5  }.
```

Für die Zustände `States` haben wir uns entschieden, diese als abstrakten `Type` zu definieren. Dies gibt uns später die Flexibilität für verschiedene Modellierungsansätze und bietet uns die Möglichkeit konkrete Instanzen nach Bedarf zu erstellen.

Die Funktion `Labeling` stellt eine direkte Abbildung der Labeling-Funktion aus der KLM-Definition dar. Dabei ist `State` eine Funktion, die jeder Formel einen Wahrheitswert zuordnet.

Definition 10. *Zustand als Wahrheitsbelegung für Formeln*

```
1  Definition State := Formula -> bool
```

Damit stellt `Labeling` eine funktionale Implementierung dar, da sie jedem Zustand direkt eine Interpretationsfunktion gibt. Die implementierte Labeling-Funktion ist zudem ebenfalls auf propositionale Logik angepasst und vereinfacht, da jede Welt genau einer Wahrheitswertzuweisung entspricht und für jede Menge von Welten eine kanonische Welt gewählt werden kann, welche dieselben Formeln erfüllt. Das bedeutet, wir reduzieren eine Menge von Welten zu einer repräsentativen Welt. Für das Repräsentationstheorem ist dies ausreichend, da nur die Information benötigt wird, welche Formeln in welchem Zustand gelten. Damit reduzieren wir auch die Komplexität der Formalisierung, ohne den Verlust wichtiger Eigenschaften.

Die Präferenzrelation `PreferenceRel` wird als binäre Relation vom Typ **`Prop`** implementiert. Dabei gibt es in dem Record-Typen erst einmal keine Einschränkung in der Definition, denn dieser enthält eben nur diesen Typen. Die Eigenschaften wie das Erfüllen der Smoothness-Bedingung formulieren wir später als separate Axiome. Diese Eigenschaften werden dann nicht bei dem Record-Typen gefordert, sondern erst bei dessen Verwendung. Damit erhalten wir eine Trennung zwischen Strukturen und Eigenschaften und bleiben modular im Aufbau des Beweises. Würden wir die Eigenschaften in dem Record-Typen direkt implementieren, müssten wir bei jeder Modellkonstruktion direkt alle diese Eigenschaften nachweisen, was die Beweise sehr viel komplexer machen würde.

Die Grundlage für die Auswertung von Formeln haben wir mit der Definition von `entails` geschaffen. Dabei handelt es sich um eine Fixpoint-Definition für die rekursive Auswertung der Formelstruktur.

Definition 11. *Entailment in Coq*

```
1 Fixpoint entails (state : State) (formula : Formula) :
2 Prop :=
3   match formula with
4   | Var n => state formula = true
5   | Not p => ~(entails state p)
6   | Contain p1 p2 => entails state p1 -> entails state p2
7   end.
```

Im Basisfall für eine Variable `Var n`, wird der Wahrheitswert direkt aus dem Zustand ausgelesen und damit geprüft, ob der Zustand der Variablen den Wert `true` zuweist. Hier benötigen wir keine Rekursion, da die atomaren Variablen die Basisbausteine darstellen. In dem rekursiven Fall für die Negation `Not p` wird erst p rekursiv ausgewertet und dann wird das logische Komplement daraus gebildet. Den letzten rekursiven Fall geben wir für die Implikation an. Dabei wird eine Implikation durch Rekursion auf beiden Teilformeln ausgewertet. Durch die Fixpoint-Definition von `entails` erhalten wir ein direktes Ergebnis, also einen Wahrheitswert, statt den Beweis einer Gültigkeit, was uns die direkte Verwendung des Ergebnisses in weiteren Berechnungen ermöglicht und später in der Definition minimaler Elemente von Vorteil sein wird, da wir dann ermitteln müssen, welche spezifische

Zustände eine Formel erfüllen. Mit `entails` stellen wir die Verbindung zwischen der syntaktischen und der semantischen Ebene her und ermöglichen die Identifikation von minimalen Zustände für jede Formel, was dann wiederum die Grundlage für die semantische Interpretation der Konsequenzrelation bildet.

Für die Formalisierung der minimalen Elemente führen wir zusätzlich die Definition von `MinimalElements` ein.

Definition 12. *Minimale Elemente*

```
1 Definition MinimalElements (model : CumulModel)
2 (formula : Formula) : Ensemble (States model) :=
3   fun state =>
4     entails (Labeling model state) formula /\
5     ~ exists state', entails (Labeling model state')
6     formula /\ PreferenceRel model state' state.
```

Die Funktion `MinimalElements` wird als Ensemble bildende Funktion implementiert. Die Parameter sind dabei `state`, das kumulative Modell, und `formula`, die zu überprüfende Formel. Als Rückgabe erhalten wir dann ein Ensemble `States model`, also eine Menge von Zuständen. Wir unterscheiden zwei Bedingungen in der Konjunktion. Wir prüfen zuallererst, dass die Formel in dem Zustand gilt und danach, dass kein präferierter Zustand existiert, in welchem die ebenfalls Formel gilt. Das bedeutet wir werten die Formel mit `entails` zunächst im Zustand `state` aus, und dann prüfen wir, durch die Negation der Existenzaussage, dass es keinen Zustand `state'` gibt, der die Formel erfüllt und präferierter als `state` ist. Dies entspricht direkt der Definition 3 aus Kapitel 2.3. Dabei ist zu beachten, dass $\hat{a}$, die Menge aller Zustände `state`, mit dem ersten Konjunkt implizit angegeben ist.

4.3.1 Definition der modellbasierten Konsequenzrelation

Wir verwenden `MinimalElements` für `SemanticEntails`, wobei geprüft wird, ob eine Konklusion `conclusion` in allen minimalen Zuständen aus der Prämisse `premise` gilt.
Formal gilt also:

$$\forall \text{state} \in \text{MinimalElements}(\text{model}, \text{premise}) \colon \text{entails}(\text{state}, \text{conclusion})$$

Durch `SemanticEntails` wollen wir auf `entails` aufbauen und es um das Konzept der minimalen Zustände. Wir definieren eine Folgerungsrelation, die aussagt, dass eine Formel aus einer anderen Formel in einem Modell folgt. Damit verbinden wir die syntaktische Konsequenzrelation mit den semantischen Modellen und erfassen die Intuition das etwas „typischerweise“ gilt, indem wir uns dabei auf die minimalen Zustände fokussieren. Dies bildet direkt die Definition der nichtmonotonen Folgerung von Kraus, Lehmann und Magidor ab.

Definition 13. *Semantisches Entailment*

```
Definition SemanticEntails (model : CumulModel)
(premise conclusion : Formula) : Prop :=
  forall state, In (States model)
                (MinimalElements model premise) state ->
                entails (Labeling model state) conclusion.
```

Mit `forall state` sichern wir ab, dass die Folgerungsbeziehung in allen minimalen Zustände gelten muss und mit der Implikation stellen wir sicher, dass nur die minimalen Zustände berücksichtigt werden, um die Idee, dass wir uns auf die typischsten Situationen konzentrieren, zu formalisieren. Wir sagen demnach aus, dass wenn ein Zustand minimal bezüglich `premise` ist, dann muss `conclusion` in diesem Zustand gelten.

Wir definieren noch eine Notation, um die Ähnlichkeit zur mathematischen Definition 6 der Konsequenzrelation in kumulativen Modellen aus Kapitel 2.3.2 von Kraus, Lehmann und Magidor zu verdeutlichen.

Notation 2. *Coq SemanticEntails Notation*

```
Notation "model : premise |~w conclusion" :=
(SemanticEntails model premise conclusion) (at level 80).
```

Dies ermöglicht ebenfalls eine intuitive Formulierung von Beweiszielen im Repräsentationstheorem, um für die Soundness zu zeigen, dass modellbasierte Relationen die System C Regeln erfüllen und umgekehrt, für die Completeness, dass für jede syntaktische Relation ein Modell existiert und kann ebenfalls für unsere Erweiterung zur Konsequenzrelation in kumulativen Modellen, in welcher wir mehrere Modelle behandeln, genutzt werden.

Definition 14. *Erweiterung der Konsequenzrelation in kumulativen Modellen*

```
Definition SatisfiesKnowledgeBase (model : CumulModel)
  (Γ : Ensemble Formula) : Prop :=
  forall formula, In Formula Γ formula ->
                  forall state,
                  entails (Labeling model state) formula.

Definition CumulativeModelEntails (Γ : Ensemble Formula)
  (premise conclusion : Formula) : Prop :=
  forall model, SatisfiesKnowledgeBase model Γ ->
  model : premise |~w conclusion.

Notation "Γ |= premise |~w conclusion" :=
(CumulativeModelEntails Γ premise conclusion) (at level 80).
```

Die Definition von `CumulativeModelEntails` baut dabei auf der Definition von `SemanticEntails` auf, um die semantische Konsequenzrelation von Kraus, Lehmann und Magidor, die wir in Kapitel 2.3 vorgestellt hatten, für eine Wissensbasis

Γ zu formalisieren. Während `SemanticEntails` die Relation für ein einzelnes kumulatives Modell beschreibt, indem wir prüfen, ob die Konklusion in allen minimalen Zuständen der Prämisse gilt, erweitern wir mit `CumulativeModelEntails` diese Idee auf eine Menge von Formeln, die genau der Wissensbasis Γ entspricht. Formal definieren wir `CumulativeModelEntails` so, dass die Konsequenzrelation in allen kumulativen Modellen gilt, die die Wissensbasis Γ erfüllen. Das bedeutet, dass jedes Modell, welches alle Formeln in Γ respektiert, was wir durch `SatisfiesKnowledgeBase` zeigen, die Bedingung von `SemanticEntails` für die Prämisse und Konklusion erfüllen muss.

Durch diese Verallgemeinerung erfassen wie die Intuition nichtmonotoner Logik, dass Konsequenzen nicht nur in einem spezifischen Modell, sondern in allen möglichen Modellen gelten müssen, die die Wissensbasis konsistent repräsentieren und spiegelt damit die semantische Definition wider, bei der eine Konsequenzrelation genau dann gilt, wenn sie in allen kumulativen Modellen, welche die gegebenen Annahmen erfüllen, ebenfalls gültig ist.

4.4 Die Smoothness Bedingung formalisiert in Coq

Um die Smoothness Bedingung aus Kapitel 2.3.1 in Coq zu formalisiern, haben wir uns dazu entschieden, diese als Axiom, statt eines bewiesenen Theorems einzuführen. Wir berufen uns hierbei erneut auf die Aussage von Lehmann und Magidor über die Endlichkeit einer Logik und den damit verbundenen Zusammenhang zur Smoothness Bedingung. Dies ist in unserem Fall eine Vereinfachung gegenüber einer konstruktiven Definition für die propositionale Logik, auf welcher Formalisierungsansatz basiert und bietet uns praktische Vorteile für die Beweisführung. Das Axiom sagt aus, dass es für jeden Zustand, in dem eine Formel gilt, einen minimalen Zustand gibt. Dieser minimale Zustand ist entweder ein präferierter Zustand oder identisch mit dem ursprünglichen Zustand, aber es gibt keinen noch präferierteren Zustand, in dem die Formel ebenfalls gilt.

Axiom 1. *Die Smoothness Bedingung*

```
1 Axiom smoothness : forall model formula state,
2   entails (Labeling model state) formula ->
3   exists minimal_state,
4     entails (Labeling model minimal_state) formula /\
5     (PreferenceRel model minimal_state state \/
6     minimal_state = state) /\
7     In (States model)
8     (MinimalElements model formula) minimal_state.
```

Als Prämisse setzten wir mit `entails` voraus, dass die Formel in dem momentanen Zustand gilt. Dann deklarieren wir die Existenz eines minimalen Zustands über den Existenzquantor. Dabei gibt es keine explizite Konstruktion, wie dieser Zustand gefunden werden kann.

Dann konstruieren wir die Eigenschaften des minimalen Zustands. Wir sichern über `entails` erneut ab, dass die Formel auch in dem minimalen Zustand gilt, und die nächste Eigenschaft der Disjunktion von `PreferenceRel` und `minimal_state = state` gibt an, dass der minimale Zustand präferiert oder identisch mit dem Ausgangszustand ist.

Schließlich sagt das Axiom aus, dass dieser minimale Zustand ein Element der Menge `MinimalElements` für die gegebene Formel ist. Gemäß unserer Definition von `MinimalElements` bedeutet dies, dass es keinen anderen noch präferierteren Zustand, in dem die Formel gilt, gibt. Damit stellen wir die eigentliche Minimalität dar.

Die axiomatische Formulierung entspricht ebenfalls wieder direkt der mathematischen Definition und vermeidet eine komplexere konstruktive Definition im propositionalen Fall, da es sich, wie wir geklärt hatten, bei der Smoothness in endlichen Sprachen als eine „technische Bedingung“ [12] handelt. Dabei fokussieren wir uns für die Vereinfachung der Beweisführung auf die wesentlichen Eigenschaften der Smoothness Bedingung. Dabei werden ebenfalls die Eigenschaften direkt aus `MinimalElements` anwendbar gemacht und wir erhalten eine unmittelbare Ableitung, dass der gefundene Zustand wirklich minimal ist. Um auf die Eigenschaften des Axioms zugreifen zu können und diese für einen Beweis verfügbar zu machen nutzen wir die Taktik **`destruct`**.

```
1 destruct (smoothness model formula state H)
2 as [minimal_state [H1 [H2 H3]]]
```

Die Hypothese `H1` würde dann aussagen, dass die Formel `formula` im minimalen Zustand gilt. `H2` gibt die Beziehung zum ursprünglichen Zustand an, ob dieser präferiert oder identisch ist. Letztlich stellt `H3` die Minimalitätsbedingung dar, also dass der Zustand ein Element von `MinimalElements` ist.

5 Coq-Beweis des Repräsentationstheorems

Der Beweis des KLM-Theorems ist in zwei Hauptmodule, `KLM_Soundness_M` und `KLM_Completeness_M`, aufgeteilt. Dabei wird jeweils der Beweis schrittweise zerlegt und die einzelnen Beweisschritte in separaten Lemmas bewiesen.

Wie bereits bei Abschnitt 2.4 in Theorem 1 dargestellt, zeigt das Repräsentationstheorem die bidirektionale Äquivalenz zwischen syntaktischen und semantischen Charakterisierungen kumulativer Konsequenzrelationen und erfordert daher den separaten Nachweis beider Richtungen.

Der **Korrektheitsbeweis** zeigt, dass jede syntaktisch durch System C ableitbare Konsequenzrelation auch semantisch in kumulativen Modellen gültig ist. Für den Beweis nutzen wir strukturelle Induktion über den induktiven Typ `CumulCons` 8. Damit weisen wir für jede der fünf Regeln des System C nach, dass diese in einem kumulativen Modell gilt. Für jede Regel formalisieren wir ein separates Lemma, wodurch wir eine modulare Beweisführung ermöglichen.

Der **Vollständigkeitsbeweis** weist dann die umgekehrte Richtung nach. Jede semantische Konsequenzrelation, die in kumulativen Modellen gilt, ist auch syntaktisch durch System C ableitbar. Hierbei nutzen wir die Konstruktion eines kanonischen Modells aus maximalen konsistenten Mengen und beweisen die Vollständigkeit durch einen Widerspruch.

Mit dieser Struktur bilden wir die mathematische Beweisidee von Kraus, Lehmann und Magidor ab und orienteiren uns für die Beweise unserer Lemmata an den Lemmata 3.15, für den Korrektheitsbeweis, und 3.17 - 3.23 für den Vollständigkeitsbeweis [11].

5.1 Korrektheitsbeweis (Soundness)

Der Soundness-Beweis zeigt, dass alle durch die syntaktischen Regeln des System C ableitbaren Konsequenzrelationen auch semantisch in kumulativen Modellen gültig sind. Die entwickelte Beweisstruktur folgt dabei dem Theorem `soundness_klm`, welches durch strukturelle Induktion über `CumulCons` alle fünf Regeln des System C abdeckt. Wir formalisieren für jede Regel ein eigenes Lemma, das dann die entsprechende semantische Eigenschaft nachweist. Die Reflexivitätsregel ist unkompliziert und direkt zu beweisen, während Right Weakening und Left Logical Equivalence zusätzliche Transformationsschritte zwischen syntaktischer und semantischer Ebene benötigen. Cut und Cautious Monotonicity nutzen die Induktionshypothesen zwar direkt, sind aber in ihrer semantischen Argumentation weitaus komplexer, wobei Cautious Monotonicity zusätzlich die Smoothness Bedingung benötigt.

5.1.1 Reflexivity Regel

Der Beweis für die Reflexivität ist in Lemma `soundness_reflexivity` festgehalten und formalisiert, dass jede Formel in den minimalen Zuständen gilt, in welchen diese selber auch gilt. Konkret wollen wir also zeigen, dass `model : formula |~w formula` gilt.

```
1 Lemma soundness_reflexivity :
2   forall (model : CumulModel) (formula : Formula),
3     model : formula |~w formula.
4 Proof.
5   unfold SemanticEntails, MinimalElements.
6   intros model formula state [H_satisfies _].
7   exact H_satisfies.
8 Qed.
```

Wir nutzen hier die Eigenschaft, dass die minimalen Zustände von `formula` direkt durch `In (States model) (MinimalElements model formula)` charakterisiert werden. Dadurch erhalten wir eine direkte, formale Definition, wann ein Zustand minimal ist, ohne vorher die minimalen Zustände konstruieren zu müssen. Da wir hier mit der Konsequenzrelation `|~w` aus dem Modell arbeiten, müssen

wir zuerst diese Definition entfalten, um überhaupt damit in dem Beweis arbeiten zu können und die tatsächliche Definition aufzudecken. Dies geschieht durch den Taktik-Aufruf `unfold SemanticEntails`. Damit erhalten wir die qualifizierte Struktur mit dem Universalquantor von `SemanticEntails` und die Implikationsstruktur der Definition. Dieser Schritt macht uns dann klar, dass wir für *alle* minimalen Zustände zeigen müssen, dass `formula` gilt, und ermöglicht uns die Minimalitätsbedingung direkt als Prämisse über `MinimalElements` einzuführen, welche wir dann auch direkt anwenden können. Durch das Auflösen der Definition mit `unfold MinimalElements`, sehen wir, dass die erste Komponente bereits erfordert, dass `formula` in dem Zustand gilt, da das erste Konjunkt der Konjunktion in `MinimalElements` uns genau das gibt, was wir beweisen wollen, nämlich gerade `entails (Labeling model state) formula`. Damit haben wir nach dem Auflösen beider Definitionen bereits das als Annahme, was wir als Ziel beweisen wollen, und können den Beweis abschließen.

5.1.2 Left Logical Equivalence Regel

Für den Beweis von Left Logical Equivalence in Lemma `soundness_LLE`, gehen wir zunächst ähnlich vor wie bei dem Beweis zuvor. Wir wollen zeigen, dass wenn `model : p |~w r` und `p` und `q` äquivalent sind, dann gilt `model : q |~w r`, also dass die minimalen Zustände für äquivalente Formeln vergleichbare Eigenschaften haben. Die Strategie hier ist also, dass wir für jeden minimalen Zustand von `q` nachweisen, dass er auch minimal für `p` sein müsste und dass ein präferierter Zustand für `q`, nach der Annahme der Äquivalenz, auch ein präferierter Zustand für `p` sein müsste.

```
1  Lemma soundness_LLE :
2    forall (model : CumulModel) (p q r : Formula),
3      (forall state, entails (Labeling model state) p <->
4                     entails (Labeling model state) q) ->
5      model : p |~w r ->
6      model : q |~w r.
7  Proof.
8    unfold SemanticEntails, MinimalElements.
9    intros model p q r H_equiv H_entails state [H_q H_minimal].
10   assert (H_p : entails (Labeling model state) p).
11   apply H_equiv; assumption.
12   apply H_entails.
13   split.
14   - assumption.
15   - intro H_exists.
16     destruct H_exists as [state' [H_p' H_pref]].
17     exfalso.
18     apply H_minimal.
```

```
19      exists state'.
20      split.
21      + apply H_equiv; assumption.
22      + assumption.
23  Qed.
```

Wie zuvor lösen wir zunächst wieder die Definitionen von `SemanticEntails` und `MinimalElements` auf, um wieder mit diesen grundlegenden Konzepten arbeiten zu können. Wir führen dann alle benötigten Variablen und Hypothesen ein. Dabei stellt `H_equiv` die Äquivalenz zwischen `p` und `q` in allen Zuständen dar, `H_entails` ist die Annahme, dass `r` aus `p` folgt und `[H_q H_minimal]` zerlegt die Annahme, dass `state` ein minimales Element für `q` ist. Genauer gibt `H_q` an, dass `q` in `state` erfüllt ist und `H_minimal` sagt aus, dass es keinen präferierteren Zustand gibt, der `q` ebenfalls erfüllt. Wir behaupten zunächst, dass `p` in `state` gilt und beweisen diese Behauptung durch die Anwendung der Äquivalenz `H_equiv`. Damit zeigen wir, dass ein minimaler Zustand `q` auch als ein Zustand `p` betrachtet werden kann. Als Nächstes wollen wir zeigen, dass `r` aus `p` folgt und nutzen dafür die Hypothese `H_entails`, was wiederum den Nachweis erfordert, dass der Zustand `state` ein minimales Element für `p` ist. Dafür teilen wir das Ziel an dieser Stelle in zwei Unterziele auf. Wir zeigen dann zuerst über `H_p`, dass auch `p` in `state` gilt und müssen dann für das zweite Teilziel noch die Minimalitätsbedingung zeigen. Diesen Beweis führen wir durch einen Widerspruch und nehmen zuerst an, dass ein präferierter Zustand für `p` existiert. Danach zerlegen wir diese Annahme in einen neuen Zustand `state'`, zusammen mit den Hypothesen `H_p'`, `p` gilt auch in `state'`, und `H_pref`, also `state'` ist präferierter gegenüber `state`. Wir haben in `H_minimal` bereits angenommen, dass `state` minimal für `q` ist, also dass kein präferierter Zustand existiert, in dem `q` gilt. Für `state'` haben wir allerdings angenommen, dass dieser Zustand präferierter gegenüber `state` ist und `p` darin gilt. Jetzt können wir wiederum durch die Äquivalenz `H_equiv` schließen, dass auch `q` in `state'` gilt. Somit haben wir nun einen Zustand `state'`, der präferierter ist und in dem `q` gilt, was im direkten Widerspruch zu `H_minimal` steht. Durch diesen Widerspruch zeigen wir demnach, dass die Annahme eines präferierten Zustands, in dem `p` ebenfalls gilt, falsch sein muss. Somit ist `state` auch minimal für `p` weshalb durch `model : p |~w r` auch `r` in `state` gilt.

5.1.3 Right Weakening Regel

In Lemma `soundness_RW` für Right Weakening formalisieren wir, dass stärkere Schlussfolgerungen auch schwächere implizieren. Um den Beweis zu führen, nutzen wir die Implikationseigenschaft `H_imp`, um direkt ableiten zu können, dass die Konklusion auch in minimalen Zuständen gilt. Unser Beweisziel ist daher zu zeigen, dass wenn `model : r |~w p` und `p -> q`, also `entails state p -> entails state q`, gilt, dann gilt auch `model : r |~w q` und damit, dass im minimalen Zustand für `r` eine Implikation erhalten bleibt.

```
Lemma soundness_RW :
  forall (model : CumulModel) (p q r : Formula),
    (forall state, entails (Labeling model state) p ->
                   entails (Labeling model state) q) ->
    model : r |~w p ->
    model : r |~w q.
Proof.
  unfold SemanticEntails, MinimalElements.
  intros model p q r H_imp H_entails state H_minimal.
  apply H_imp.
  apply H_entails; assumption.
Qed.
```

Wie auch bereits bei den vorherigen Beweisen müssen wir zunächst die Definition von `SemanticEntails` und damit auch `MinimalElements` auflösen. Danach führen wir alle benötigten Variablen und Hypothesen ein. Die Variable `model` stellt unser kumulatives Modell dar, und die Variablen `p`, `q` und `r` unsere Formeln. Die Hypothese `H_imp` sagt aus, dass `p` wiederum `q` impliziert, also

```
forall state, entails (Labeling model state) p
-> entails (Labeling model state) q.
```

`H_entails` nutzen wir wieder als Hypothese, dass `p` üblicherweise aus `r` folgt, genauer `model : r |~w p`. Für einen beliebigen Zustand nutzen wir `state` und führen schlussendlich noch die Hypothese `H_minimal` ein, dass `state` minimal für `r` ist. Unser Beweisziel ist zunächst `entails (Labeling model state) q`. Wir haben keinen direkten Weg, um zu zeigen, dass `q` in `state` gilt, aber wissen, dass wenn `p` in `state` gilt, dann gilt auch `q` in `state`, was der Hypothese `H_imp` entspricht. Außerdem wissen wir, dass `p` wiederum in allen minimalen Zuständen, wo `r` gilt, ebenfalls gilt und `state` ein minimaler Zustand ist, indem schon `r` gilt. Also wenden wir zunächst die Hypothese `H_imp` an, um unser Beweisziel zu `entails (Labeling model state) p` zu ändern. Jetzt müssen wir zeigen, dass `p` in `state` gilt, was wir durch die Anwendung der Hypothese `H_entails` auf `H_minimal` erreichen. Die Hypothese `H_entails` bewirkt, dass das Beweisziel durch die Annahme ersetzt wird, dass `state` minimal für `r` ist. Da wir aber bereits `H_minimal` haben, das genau dieser Bedingung entspricht, können wir mit **`assumption`** das Beweisziel direkt lösen und den Beweis abschließen.

5.1.4 Cut Regel

Der Beweis für Cut ist in Lemma `soundness_Cut` formalisiert. Die Herausforderung hierbei liegt darin zu zeigen, dass $p \mathrel{|\!\sim_w} r$ gilt, wenn sowohl $p \mathrel{|\!\sim_w} q$ als auch $p \wedge q \mathrel{|\!\sim_w} r$ gelten. Wir wollen zunächst zeigen, dass in den minimalen Zuständen, in welchen p gilt, auch q gilt, um dann nachzuweisen, dass in diesen Zuständen dann auch r gelten muss. Hierfür verwenden wir unser Hilfslemma `entails_conjunction`, um die Konjunktion zu handhaben.

```
1  Lemma soundness_Cut :
2    forall (model : CumulModel) (p q r : Formula),
3      model : (p ∧ q) |∼w r ->
4      model : p |∼w q ->
5      model : p |∼w r.
6  Proof.
7    unfold SemanticEntails, MinimalElements.
8    intros model p q r H_conj_entails H_p_entails_q
9           state [H_p H_minimal].
10   assert (H_q : entails (Labeling model state) q).
11   apply H_p_entails_q; split; [exact H_p | exact H_minimal].
12   apply H_conj_entails.
13   split.
14   - rewrite entails_conjunction.
15     split; assumption.
16   - intro H_exists.
17     destruct H_exists as [state' [H_conj' H_pref]].
18     rewrite entails_conjunction in H_conj'.
19     destruct H_conj' as [H_p' _].
20     exfalso.
21     apply H_minimal.
22     exists state'.
23     split; assumption.
24 Qed.
```

Erneut wie zuvor lösen wir die Definitionen von `SemanticEntails` und `Minimal Elements` auf und führen danach die Variablen und Hypothesen ein. Wir haben wieder `model` für unser Modell, `p`, `q` und `r` für unsere Formeln und `state` für den Zustand. Die Hypothese `H_conj_entails` ist die Annahme, dass `r` aus `p ∧ q` folgt und `H_p_entails_q` nimmt an, dass `q` aus `p` folgt. Und `[H_p H_minimal]` zerlegt die Annahme, dass `state` minimal für `p` ist. Wir behaupten vorerst, dass `q` in `state` gilt und beweisen diese Behauptung, durch das Anwenden der Hypothese `H_p_entails_q`, dass `q` aus `p` folgt. Danach wenden wir die Hypothesen `H_p` und `H_minimal` an, um zu zeigen, dass `state` minimal für `p` ist. Wir zeigen also, dass in minimalen Zuständen, in welchen `p` gilt, auch `q` gilt. Zu diesem Beweisschritt können wir nun die Konjunktionsannahme `H_conj_entails` anwenden, und geben damit an, dass `r` aus `(p ∧ q)` folgt. Dies wiederum erfordert dann aber auch den Nachweis, dass `state` minimal für `(p ∧ q)` ist. Da wir nach dem Anwenden der Hypothese eine Propositions-Konjunktion (`/\`) erhalten, können wir die Taktik **`split`** anwenden, um den Beweis in zwei Unterziele an dieser Stelle aufzuspalten. Somit erhalten wir, das erste Ziel, wo wir zeigen müssen, dass `(p ∧ q)` in state gilt und als zweites Ziel, dass keine präferierten Zustände existieren in denen `(p ∧ q)` ebenfalls gilt. Um die Konjunktion von unseren Formeln `p` und `q` zu lösen, nutzen wir das Lemma `entails_conjunktion` mit der Taktik **`rewrite`**. Das Lemma

sagt aus, dass eine Konjunktion genau dann gilt, wenn beide Konjunkte gelten. Wir übersetzten hier die Bedeutung von `entials (Labeling model state)(p ∧ q)` zu:

```
1 entails (Labeling model state) p /\
2 entails (Labeling model state) q
```

Ein weiteres `split` teilt dann auch diese Konjunktion in zwei weitere Ziele auf. Zuerst, dass die Formel `p` im Zustand `state` gilt, und als nächstes, dass ebenfalls die Formel `q` in Zustand `state` gilt. Da wir genau diese Ziele bereits als Hypothesen `H_p` und `H_q` eingeführt haben, können diese direkt ausgewählt und die Ziele damit gelöst werden.

Da nun dieses erste Konjunkt aus der ursprünglichen Konjunktion vollständig gelöst wurde, fokussieren wir nun das zweite Ziel, indem wir noch die Minimalität für `(p ∧ q)` nachweisen müssen, also dass es keinen präferierten Zustand `state'` gibt, in dem `(p ∧ q)` gilt. Wir zeigen dies wieder mit einem Gegenbeispiel und nehmen an, dass eben solch ein Zustand `state'` existiert. Aus der Tatsache, dass `(p ∧ q)` in `state'` gilt, folgt dann, dass auch `p` in `state'` gilt. Dies steht aber im Widerspruch zur Annahme, dass `state` bereits minimal für `p` ist. Durch diesen Widerspruch haben wir folglich gezeigt, dass `state` auch minimal für `(p ∧ q)` sein muss und `state` erfüllt damit alle Bedingungen, um ein minimales Element für `(p ∧ q)` zu sein. Da die Hypothese `H_conj_entails` aussagt, dass `r` in allen minimalen Zuständen, in denen `(p ∧ q)` gilt, auch gilt, folgt unmittelbar, dass `r` auch in `state` gelten muss. Damit haben wir eine Transitivitätseigenschaft der kumulativen Konsequenzrelation gezeigt. Wenn `p` zu `q` führt und `(p ∧ q)` zu `r`, dann führt auch `p` zu `r`, da minimale Zustände, in denen `p` gilt, auch minimal für `(p ∧ q)` sind.

5.1.5 Cautious Monotonicity Regel

Lemma `soundness_CM` beweist die eingeschränkte Form der Monotonie. Die Cautious Monotonicity Regel ist die komplexeste der fünf Regeln und erfordert das Anwenden der Smoothness Bedingung. Wir zeigen, dass wenn in minimalen Zuständen, in denen `p` gilt, sowohl `q` als auch `r` gelten, dann gilt in minimalen Zuständen, in denen `(p ∧ q)` gilt, auch `r`. Demnach lautet unser Beweisziel:

> Wenn `model : p |∼w q` und `model : p |∼w r`
> dann `model : (p ∧ q) |∼w r`

Wir beweisen dafür, dass `r` in allen minimalen Zuständen, wo auch `(p ∧ q)` gilt, ebenfalls gilt.

```
1 Lemma soundness_CM :
2   forall (model : CumulModel) (p q r : Formula),
3     model : p |∼w q ->
4     model : p |∼w r ->
```

```
5       model : (p ∧  q) |~w r.
6   Proof.
7     unfold SemanticEntails, MinimalElements.
8     intros model p q r H_p_q H_p_r state [H_conj H_minimal].

10    rewrite entails_conjunction in H_conj.
11    destruct H_conj as [H_p H_q].

13    assert (H_p_in_state : entails (Labeling model state) p).
14    { exact H_p. }

16    destruct (smoothness model p state H_p) as
17      [min_state [H_min_p [H_pref_or_eq H_min_element]]].

19    unfold MinimalElements in H_min_element.
20    destruct H_min_element as [H_min_p_satisfies H_min_minimal].

22    assert (H_min_q : entails (Labeling model min_state) q).
23    apply H_p_q; split; [assumption | exact H_min_minimal].

25    assert (H_min_conj : entails (Labeling model min_state)
26                                 (p ∧ q)).
27    apply entails_conjunction; split; assumption.

29    destruct H_pref_or_eq as [H_pref | H_eq].

31    - (* Fall 1: min_state < state *)
32      exfalso.
33      apply H_minimal.
34      exists min_state.
35      split; [assumption | assumption].

37    - (* Fall 2: min_state = state *)
38      subst min_state.
39      apply H_p_r.
40      split; [assumption | exact H_min_minimal].
41  Qed.
```

Wie auch schon bei den vorherigen Regeln beginnen wir mit dem Auflösen der Definitionen von `SemanticEntails` und `MinimalElements`. Danach führen wir die Variablen `model`, `p`, `q`, `r` und `state` ein und benennen unsere Hypothesen. Dabei steht `H_p_q` für `q` folgt üblicherweise aus `p` und `H_p_r` für `r` folgt üblicherweise aus `p`. Um auszudrücken, dass in `state` die Konjunktion `(p ∧ q)` gilt, nutzen wir `H_conj` und für die Minimalität von `state` zu `(p ∧ q)` die Hypothese

H_minimal.

Wir wollen wieder zuerst die einzelnen Konjunkte aus der Konjunktion extrahieren. Dafür greifen wir erneut auf das Lemma entails_conjunction und die Taktik **rewrite** zurück. Danach zerlegen wir die Konjunktion in zwei separate Hypothesen, wobei H_p aussagt, dass p in state gilt und H_q, dass q in state gilt. Dies ermöglicht es uns später zu zeigen, dass state in einem bestimmten Verhältnis zu minimalen Zuständen steht, in denen p gilt.

Da state die Formel p erfüllt, können wir nun die Smoothness-Bedingung nutzen, da diese garantiert, dass für jeden Zustand, in dem p gilt, ein minimaler Zustand existiert, welcher präferiert oder mit diesem Zustand identisch ist. Dadurch erhalten wir min_state, den minimalen Zustand, wo p gilt, und H_min_p, die Annahme, dass p in min_state gilt. Außerdem sagen wir in H_pref_or_eq aus, dass min_state präferiert oder identisch zu state ist und geben mit H_min_element an, dass min_state minimal für p ist.

Folglich können wir die Minimalitätsbedingung für H_min_minimal aus der Annahme H_min_element ableiten, denn die Annahme H_min_element besagt, dass es keinen Zustand gibt, welcher präferierter gegenüber min_state ist und in dem p gilt. Da min_state minimal in MinimalElements model p ist und H_p_q gilt, folgt, dass min_state auch q erfüllt. Dies zeigen wir durch Anwendung von H_p_q auf min_state, da genau min_state die Formel p erfüllt und auch minimal ist. Es folgt, dass (p ∧ q) in min_state erfüllt ist, denn wir haben bereits für p und sowohl auch für q gezeigt, dass diese in min_state gelten, und damit ist dies auch für die Konjunktion erfüllt.

Wir spalten als Nächstes H_pref_or_eq in zwei Fälle auf, welche wir gesondert untersuchen werden. Den ersten Fall stellt H_pref dar und sagt aus, dass min_state präferierter gegenüber state ist. Der zweite Fall H_eq gibt an, dass min_state identisch mit state ist.

Für den ersten Fall, wenn min_state präferiert gegenüber state ist, erhalten wir aber einen Widerspruch aufgrund der Minimalität von state, da in min_state sowohl die Konjunktion (p ∧ q) erfüllt ist (H_min_conj) und min_state ebenfalls präferierter gegenüber state ist. Wir zeigen diesen Widerspruch über die Minimalitätsbedingung H_minimal für state und geben min_state als ein Gegenbeispiel an. Auch hier können wir nun automatisch das Beweisziel des Widerspruchs aufgrund der vorher eingeführten Hypothesen lösen. Denn wie wir gezeigt haben, erfüllt min_state genau diese Bedingungen, da (p ∧ q) in min_state gilt (H_min_conj) und min_state präferierter gegenüber state ist.

Im zweiten Fall betrachten wir, dass min_state identisch mit state ist. Die Taktik **subst** min_state weist dabei Coq an, nach Gleichungen zu suchen, welche min_state definieren, und diese Variable dann zu ersetzen. In diesem Fall haben wir

```
H_eq : min_state = state
```

aus unserer Fallunterscheidung. Demnach ersetzt Coq nun für den gesamten Be-

weis alle Vorkommen von `min_state` durch `state`. Dieser Schritt ist eine bewusste Entscheidung für die Vereinfachung des Beweises, denn wir betrachten hier den Fall, in welchem `min_state` und `state` identisch sind, und es daher unnötig kompliziert wäre, beide Variablennamen beizubehalten. Außerdem arbeiten wir durch diese Substitution mit einem einzigen Zustand `state`. Da jetzt `state` minimal in `MinimalElements model p` ist, weil wir `min_state = state` substituieren, können wir direkt die Hypothese `H_p_r` anwenden, welche besagt, dass alle minimalen Zustände, in denen `p` gilt, auch `r` erfüllen. Wir müssen demnach zeigen, dass `entails (Labeling model state) r` gilt. Dies folgt aus `H_p_r`, da `state p` erfüllt (`H_p`) und minimal in `MinimalElements model p` ist (`H_min_minimal`). Auch hier können wir das Beweisziel automatisch durch die Taktik **`apply`** `H_p_r; split; [assumption | exact H_min_minimal]` lösen.

5.1.6 Induktionsbeweis der Soundness

Das Theorem `soundness_KLM` formalisiert die Korrektheit des KLM-Theorems. Wenn eine Konsequenzrelation syntaktisch durch die Regeln des System C ableitbar ist (`CumulCons`), dann ist sie auch semantisch durch kumulative Modelle repräsentierbar `(|= p |~w q)`. Da wir nun für jede der fünf Regeln des System C ein Lemma eingeführt und bewiesen haben, können wir den Beweis für die Soundness des KLM-Theorems konstruieren. Der Beweis nutzt dabei strukturelle Induktion über den induktiven Typ `CumulCons`. Für jeden Konstruktor müssen wir also zeigen, dass wenn die Prämissen semantisch gelten, auch die Konklusion semantisch gilt.

```
1  Theorem soundness_klm :
2    forall (Γ : Ensemble Formula) (p q : Formula),
3      CumulCons Γ p q -> Γ |= p |~w q.
4  Proof.
5    intros Γ p q H_cons.
6    unfold CumulativeModelEntails.
7    intros model H_respects_kb.

9    induction H_cons.

11   - apply soundness_reflexivity.

13   - apply soundness_LLE with p.
14     + intros state.
15       assert (H_equiv : In (Formula) Γ (p ↔ q)).
16       assumption.
17       apply H_respects_kb in H_equiv.
18       assert (H_state_equiv : entails (Labeling model state)
19                                       (p ↔ q)).
20       { apply H_equiv. }
```

```
21        apply entails_equivalence in H_state_equiv.
22        assumption.
23      + apply IHH_cons.
24        exact H_respects_kb.

26    - apply soundness_RW with p.
27      + intros state H_p.
28        assert (H_impl : In (Formula) Γ (p → q)).
29        assumption.
30        apply H_respects_kb in H_impl.
31        simpl in H_impl.
32        apply H_impl.
33        assumption.
34      + apply IHH_cons.
35        exact H_respects_kb.

37    - apply soundness_Cut with q.
38      + apply IHH_cons1.
39        exact H_respects_kb.
40      + apply IHH_cons2.
41        exact H_respects_kb.

43    - apply soundness_CM.
44      + apply IHH_cons1.
45        exact H_respects_kb.
46      + apply IHH_cons2.
47        exact H_respects_kb.
48  Qed.
```

Wir beginnen den Beweis damit, die Variablen und Hypothesen einzuführen. Die Variable Γ ist von Typ `Ensemble Formula` und repräsentiert unsere Wissensbasis, welche alle Formeln enthält, die wir als Grundwissen vorausgesetzt werden. Die Variable `p` ist vom Typ `Formula` und stellt die Prämisse der Konsequenzrelation dar. Analog dazu führen wir die Variable `q` vom Typ `Formula` ein, um die Konklusion der Konsequenzrelation darzustellen. Die Hypothese `H_cons` vom Typ `CumulCons Γ p q`, besagt, dass `q` syntaktisch aus `p` unter der Wissensbasis Γ nach den Regeln des System C folgt. Unser Beweisziel ist es dann zu zeigen, dass `q` auch semantisch aus `p` unter der Wissensbasis Γ folgt. Zudem lösen wir die Definition von `CumulativeModelEntails` auf, um die konkrete Semantik offenzulegen. Damit ändert sich nun unser Beweisziel zu:

```
forall model,
SatisfiesKnowledgeBase model Γ -> model : p |∼w q
```

Es sagt aus, dass aus jedem kumulativen Modell `model`, welches die Wissensbasis Γ respektiert, `q` nichtmonoton aus `p` folgt. Als Nächstes führen wir noch die Variable `model` vom Typ `CumulModel` als ein beliebiges kumulatives Modell ein und geben mit der Hypothese `H_respects_kb` an, dass das Modell `model` die Wissensbasis Γ respektiert, das heißt, dass alle Formeln in Γ in allen Zuständen des Modells gelten. Nach diesem Beweisschritt reduziert sich das Beweisziel auf `mode : p |∼w q`, was bedeutet, dass wir nun zeigen müssen, dass in diesem spezifischen Modell `model` die semantische Konsequenzrelation gilt.

Mit der Taktik **`induction`** `H_cons` beginnen wir die strukturelle Induktion über die Hypothese `H_cons`, was dazu führt, dass durch die fünf Konstruktoren von `CumulCons` fünf separate Beweisziele erzeugt werden.

- Reflexivitätsfall: Wir müssen zeigen, dass `model : p |∼w p` gilt.
- LLE-Fall: Wir müssen zeigen, dass wenn `model : p |∼w r` gilt und `p ↔ q in Γ ist, dann gilt auch model : q |∼w r`
- RW-Fall: Wir müssen zeigen, dass wenn `model : r |∼w p` gilt und `p → q` in Γ ist, dann gilt auch `model : r |∼w q`.
- Cut-Fall: Wir müssen zeigen, dass wenn `model : (p ∧ q) |∼w r` und `model : p |∼w q` gelten, dann gilt auch `model : p |∼w r`.
- Und wir müssen zeigen, dass wenn `model : p |∼w q` und `model : p |∼w r` gelten, dann gilt auch `model : (p ∧ q) |∼w r`.

Für jedes Beweisziel erhalten wir zudem entsprechende Induktionshypothesen, welche besagen, dass die Prämissen jeder Regel bereits semantisch gültig sind und wir müssen zeigen, dass auch die Konklusion semantisch gültig ist.

Reflexivität Für den Reflexivitätsfall ist der Beweis direkt lösbar, denn wir müssen einfach nur das bereits bewiesene Lemma `soundness_reflexivity` anwenden, da dieses Lemma zeigt, dass die Reflexivitätseigenschaft in jedem kumulativen Modell gilt.

Left Logical Equivalence Der LLE-Fall ist komplexer, da wir die Verbindung zwischen der syntaktischen Äquivalenz in der Wissensbasis und der semantischen Äquivalenz im Modell herstellen müssen. Das bedeutet konkret, dass wir aus `(p ↔ q)` dann `entails (Labeling model state) p <-> entails (Labeling model state) q` ableiten müssen. Wir geben zunächst an, dass `(p ↔ q)` in Γ enthalten ist und nutzen dann `H_respects_kb`, um zu zeigen, dass das Modell diese Äquivalenz respektiert. Außerdem behaupten wir, dass diese Äquivalenz in jedem Zustand gilt. Nun haben wir die Hypothese `H_state_equiv`, die besagt,

dass die Formel (p ↔ q) im Zustand state des Modells gilt und können diese Äquivalenz nun mit **apply** entails_equivalence in die semantische Äquivalenz umwandeln und erhalten damit genau die Form, welche wir für das Lemma soundness_LLE benötigen. Schließlich können wir die Induktionshypothese IHH_cons anwenden, um den Beweis vervollständigen zu können und müssen als Letztes noch zeigen, dass SatisfiesKnowledgeBase model Γ gilt, was aber trivial ist, da wir genau das bereits durch H_respects_kb gegeben haben. Damit ist der Teilbeweis abgeschlossen und wir haben gezeigt, dass wenn p und q semantisch äquivalent sind und r semantisch aus p folgt, dann folgt auch r semantisch aus q, also genau wie es die LLE-Regel erfordert.

Right Weakening Ähnlich wie schon bei dem vorhergehenden Beweis müssen wir auch hier erst noch die syntaktische Implikation (p → q in der Wissensbasis in eine semantische Implikation im Modell übersetzen. Wieder bestätigen wir, dass (p → q) in der Wissensbasis Γ enthalten ist. Wir nutzen dann H_respects_kb, um wieder zu zeigen, dass das Modell diese Implikation respektiert. Durch die Taktik **simpl** vereinfachen wir die Definition von entails für eine Implikationsformel und erhalten dadurch die Form:

```
H_impl : forall state,
  entails (Labeling model state) p ->
  entails (Labeling model state) q
```

Für jeden Zustand des Modells gilt, dass wenn p in dem Zustand gilt, dann gilt auch q in diesem Zustand. Jetzt können wir die Hypothese H_impl anwenden und über **assumption** die verfügbare Hypothese H_p nutzen, welche aussagt dass p tatsächlich im Zustand gilt. Daraus können wir nun folgern, dass auch q in dem Zustand gelten muss, so wie es für die RW-Regel notwendig ist. Nachdem wir gezeigt haben, dass wenn p gilt, gilt auch q können wir wieder die Induktionshypothese IHH_cons anwenden, um das zweite Unterziel abzuschließen, welches besagt, dass p nichtmonoton aus r folgt und wir können unseren Teilbeweis wie auch zuvor abschließen.

Cut Der Cut-Fall erscheint einfacher, obwohl das Lemma komplexer ist. Dies liegt in der Struktur den Induktionsbeweises, denn wir haben bei Cut zwei Induktionshypothesen. IHH_cons1 zeigt, dass (p ∧ q) |∼w r semantisch gilt und IHH_cons2 sagt aus, dass (p |∼w q) ebenfalls semantisch gilt. Beide dieser Hypothesen stellen bereits die Verbindung zwischen Syntax und Semantik her und wir müssen keine zusätzlichen Schritte zur Umformung zwischen diesen Ebenen durchführen und auch keine Formeln aus der Wissensbasis Γ extrahieren oder denen semantische Gültigkeit gesondert nachweisen. Wir können hier direkt das Lemma soundness_Cut mit den beiden Induktionshypothesen anwenden. Diese direkte Anwendung ist möglich, weil die Cut-Regel ausschließlich auf kumulativen

Konsequenzrelationen basiert und keine expliziten Formeln aus der Wissensbasis verwendet.

Cautious Monotonicity Der CM-Fall ist ebenfalls wie der Cut-Fall wieder einfacher zu handhaben. Wir haben erneut zwei Induktionshypothesen. Die erste Hypothese `IHH_cons1` sagt aus, dass `p |∼w q` semantisch gilt und `IHH_con2` zeigt, dass auch `p |∼w r` semantisch gilt. Demnach können wir wieder direkt das Lemma `soundness_CM` anwenden, da durch die Induktionshypothesen bereits alle benötigten semantischen Eigenschaften liefern. Das Lemma kombiniert diese dann zu unserer geplanten Schlussfolgerung. Das macht diesen Anwendungsfall im Induktionsbeweis vergleichsweise einfacher, denn die Komplexität liegt schon im Lemma selber, wo intern die Smoothness-Bedingung genutzt wird. Dies zeigt ebenfalls einen weiteren guten Grund, warum die angesetzte Modularität für die Formalisierung von Vorteil ist. Sobald ein komplexeres Lemma erst einmal bewiesen ist, kann dies als modularer Baustein in weiteren Beweisen verwendet werden, ohne jedes Mal die interne Komplexität neu adressieren zu müssen.

Diese Modularität findet sich auch in der Struktur des System C wieder, wo Cautious Monotonicity eine eigenständige Regel darstellt und zusammen mit den anderen Regeln eben jene Grundlage für nichtmonotones Schließen bildet. Der Induktionsbeweis zeigt außerdem, dass jede dieser Regeln unabhängig voneinander semantisch fundiert ist, was auch wiederum die Korrektheit des gesamten Systems garantiert. Mit dem Beweis des Theorems `soundness_KLM` haben wir den ersten Teil des KLM-Theorems formalisiert und werden uns im Folgenden dem Vollständigkeitsbeweis zuwenden.

5.2 Vollständigkeitsbeweis (Completeness)

Für die Formalisierung der Vollständigkeit des KLM-Theorems wollen wir zeigen, dass jede kumulative Konsequenzrelation, die durch die Regeln des Systems (Reflexivität, LLE, RW, Cut, CM) definiert ist, durch ein kumulatives Modell repräsentierbar ist. Das bedeutet, dass jede semantisch gültige Schlussfolgerung `Γ |= p |∼w q` auch syntaktisch ableitbar ist `Γ : p |∼w q`. Dafür werden wir zunächst ein kanonisches Modell konstruieren, welches genau die gegebene Konsequenzrelation repräsentiert. Danach werden wir zeigen, dass das Modell kumulativ ist und dass jeder Zustand im Modell die Wissensbasis Γ respektiert. In dem Hauptbeweis `completeness_klm` werden wir dann die Vollständigkeit durch einen Widerspruch beweisen, indem wir annehmen, dass eine Konklusion semantisch gültig ist (`Γ |= p |∼w q`), aber nicht syntaktisch ableitbar (`∼ CumulCons Γ p q`).

Kraus, Lehmann und Magidor definieren in ihrer Arbeit ein Modell basierend auf Äquivalenzklassen von Formeln und normalen Welten, welches einen typischen Fall, in dem eine Formel gilt, darstellt. Für die Formalisierung in Coq werden wir hier etwas von der Arbeit von Kraus, Lehmann und Magidor [11] abweichen und statt Äquivalenzklassen auf maximal konsistente Mengen als Zustände zurückgrei-

fen. Dabei entspricht eine Äquivalenzklasse $[\alpha]$ einer Formel α direkt einer maximalen konsistenten Menge, welche α enthält. Das bedeutet, dass beide Ansätze zum gleichen logischen Ergebnis führen und technisch nur unterschiedlich dargestellt sind.

Die Grundidee dabei ist, dass wenn eine Formel `q` nicht aus `p` unter einer Wissensbasis Γ ableitbar ist, dann gibt es eine maximale konsistente Menge, welche Γ und `p` enthält, aber nicht `q`. Das bedeutet konkret, dass die syntaktische Aussage „`q` ist nicht aus `p` ableitbar“ in eine semantische Aussage über die Existenz einer maximalen konsistenten Menge übersetzt. Damit erhalten wir eine Brücke, um zwischen syntaktischer Nicht-Ableitbarkeit und semantischer Nicht-Folgerung wechseln zu können. Außerdem erhalten wir dadurch eine konkrete Eigenschaft, welche wir im Beweis verwenden können, um auszudrücken, wann eine Formel nicht aus einer anderen ableitbar ist und damit auch nicht-ableitbare Formeln zu bestimmen. Der Hauptgrund warum wir uns dazu entschieden haben auf die maximalen konsistenten Mengen zurückzugreifen liegt jedoch in der Praktikabilität in Coq, denn wir können direkt mit `Ensemble Formula` weiterarbeiten, ohne dabei komplexere Strukturen implementieren zu müssen, welche die Komplexität der Formalisierung erheblich erhöhen würde. Zudem bietet die von uns genutzte Library bereits eine Implementierung der maximalen konsistenten Mengen, welche wir verwenden können. Wir behalten damit zudem den Fokus auf dem Repräsentationstheorem.

5.2.1 Kanonisches Modell

Wir beginnen die Formalisierung damit, ein kanonisches Modell zu konstruieren.

Definition 15. *Das kanonische Modell in Coq*

```
1  Definition CanonicalStates := Ensemble Formula.

3  Definition CanonicalPreferenceRel
4    (w1 w2 : CanonicalStates) : Prop :=
5    exists p, w1 ⊢ p /\ ~ (w2 ⊢ p).

7  Definition CanonicalModel : CumulModel :=
8  {|
9    States := CanonicalStates;
10   Labeling := fun w p => valuemaxf w p;
11   PreferenceRel := CanonicalPreferenceRel
12 |}.
```

Dabei entspricht `CanonicalStates` den Äquivalenzklassen aus der Arbeit von Kraus, Lehmann und Magidor, jedoch als maximale konsistente Mengen. Eine maximale konsistente Menge ist eine Menge von Formeln, die konsistent ist, also keine Widersprüche enthält, und maximal ist, da keine weiteren Formeln hinzugefügt

werden können, ohne eine Inkonsistenz zu erzeugen. Diese Zustände repräsentieren mögliche Welten, in denen bestimmte Formeln wahr sind.

Die Definition `CanonicalPreferenceRel` implementiert die Präferenzrelation aus Definition 3.21 [11]. Wir sagen mit `CanonicalPreferenceRel` aus, dass `w1` präferiert gegenüber `w2` ist, wenn es eine Formel `p` gibt, die in `w1` ableitbar (`w1 ⊢ p`) ist, aber nicht in `w2` ((`∼(w2 ⊢ p)`). Diese Relation ist demnach entscheidend für die Definition minimaler Elemente.

Für die Labeling-Funktion `Labeling` verwenden wir `valuemaxf` aus der Library [8]. Dies dient der Bewertung von Formeln in Zuständen. Konkret wird durch `valuemaxf w p` geprüft, ob die Formel `p` in der maximal konsistenten Menge `w` enthalten ist. Wenn `p ∈ w` dann ist die Formel `p` in `w` wahr, andernfalls ist sie es nicht.

5.2.2 Existenz und Eigenschaften maximal konsistenter Mengen

An dieser Stelle führen wir ein Axiom ein, welches dem Lemma 3.18 [11] entsprechen soll. Lemma 3.18, zeigt, dass wenn $\alpha \not\sim \beta$ gilt, dann gibt es eine normale Welt für α, die nicht β erfüllt.

Axiom 2. *Existenzsatz für maximal konsistente Mengen*

```
1 Axiom exists_maximal_consistent : forall Γ p q,
2   ∼ (CumulCons Γ p q) ->
3   exists w,
4   maximal_consistent_set w /\ Γ ⊆ w /\
5   p ∈ w /\ ∼ q ∈ w.
```

Wir sagen mit dem Axiom dann aus, dass wenn `q` nicht aus `p` unter Γ ableitbar ist, dann gibt es eine maximale konsistente Menge `w` und diese Menge enthält Γ und `p` aber nicht `q`. Genau hier definieren wir also die Brücke zwischen syntaktischer Nicht-Ableitbarkeit und semantischer Repräsentation und ermöglicht es uns im Beweis `completeness_klm` den Widerspruch zu formulieren, indem wir mit dem Axiom ein Gegenbeispiel erzeugen werden.

Für die maximalen konsistenten Mengen benötigen wir außerdem noch eine Formalisierung der Eigenschaften dieser Mengen, um mit diesen im kanonischen Modell korrekt arbeiten zu können.

Lemma 1. *Deduktive Äquivalenz in maximalen konsistenten Mengen*

```
1 Lemma max_consistent_deduction :
2   forall (w : Ensemble Formula) (p : Formula),
3     maximal_consistent_set w -> (p ∈ w <-> w ⊢ p).
```

Das Lemma `max_consistent_deduction` zeigt, dass eine Formel `p` genau dann in einer maximalen konsistenten Menge `w` ist, wenn diese aus `w` abgeleitet werden kann. Es dient also dazu, dass wir zeigen können, dass eine Formel `p` in einem

Zustand `state` enthalten ist, wenn `state ⊢ q` gilt. Wir formalisieren damit die Eigenschaft von maximalen konsistenten Mengen, dass diese alle konsistenten Formeln erhalten und keine Widersprüche erzeugen. Dadurch wird sichergestellt, dass auch die Zustände konsistent sind und alle ableitbaren Formeln enthalten, was für die Korrektheit des Modells notwendig ist.

Damit wir garantieren können, dass die Zustände in dem kanonischen Modell vollständig sind, also dass eine maximale konsistente Menge für jede Formel `p` entweder `p` oder `¬p` enthält und wir eine klare Bewertung für jede Formel treffen können, führen wir noch Lemma `max_consistent_complete` ein.

Lemma 2. *Konsistenz maximaler konsistenter Mengen*

```
1 Lemma max_consistent_complete :
2   forall (w : Ensemble Formula) (p : Formula),
3     maximal_consistent_set w -> p ∈ w \/ ¬p ∈ w.
```

Durch dieses Lemma können wir unvollständige Zustände verhindern und sichern damit die Grundlagen für die Präferenzrelation `CanonicalPreferenceRel`, die darauf basiert, welche Formeln in einem Zustand wahr sind. Ohne den Nachweis der Vollständigkeit könnten die minimalen Elemente nicht korrekt formalisiert werden, denn auch `MinimalElements` 3 erfordert, dass Zustände `p` erfüllen und keine weiteren Zustände existieren, welche ebenfalls `p` erfüllen.

5.2.3 Semantische Interpretation im kanonischen Modell

In der Arbeit von Kraus, Lehmann und Magidor wird in Lemma 3.24 [11] eine Beziehung zwischen der syntaktischen Konsequenzrelation und der semantischen Modellrelation hergestellt.

Lemma 3.24. *Das Äquivalenzlemma der Konsequenzrelationen*

$$a \mid\!\sim b \textit{ iff } a \mid\!\sim_W b$$

Es sagt aus, dass eine syntaktische Konsequenzrelation zwischen Formeln der semantischen Relation im kanonischen Modell entspricht. Um diese Äquivalenz zwischen syntaktischer und semantischer Konsequenzrelation zu etablieren, führen wir ein Axiom ein.

Axiom 3. *Charakterisierung der Erfüllbarkeit im kanonischen Modell*

```
1 Axiom canonical_entails :
2   forall (w : CanonicalStates) (p : Formula),
3     maximal_consistent_set w ->
4     entails (Labeling CanonicalModel w) p <-> p ∈ w.
```

Das Axiom zeigt, dass für einen spezifischen Zustand `w` eine Formel `p` genau dann in diesem Zustand gilt, wenn `p` ein Element von `w` ist. Wir stellen also hier nicht direkt die Beziehung zwischen der syntaktischen und semantischen Konsequenzrelation her, sondern zwischen der semantischen Wahrheit einer Formel in einem Zustand und der Mengenzugehörigkeit der Formel in diesem Zustand. Es stellt daher eine technische Grundlage dar, welche es ermöglicht, das Lemma 3.24 später zu etablieren. Das Axiom zeigt, wie Formeln in individuellen Zuständen interpretiert werden und die Gesamtheit aller maximalen konsistenten Mengen, mit ihren Eigenschaften, führt dann mit diesem Axiom zur Äquivalenz aus dem Lemma 3.24.

5.2.4 Minimalität und Smoothness im kanonischen Modell

Nachdem wir nun ein kanonisches Modell konstruieren können und die Verbindung zwischen syntaktischer und semantischer Ebene hergestellt haben müssen wir noch die Eigenschaften für die Minimalität und Smoothness des Modells sicherstellen, damit wir im späteren Beweis ein gültiges kumulatives Modell erhalten können und orientieren uns hierfür an dem Lemma 3.23 [11].

Wir müssen für die Minimalität zeigen, dass für jede Formel `p` die entsprechenden Zustände, welche `p` erfüllen, minimal bezüglich der Präferenzrelation sind, da unser Modell sonst die semantische Konsequenzrelation `SemanticEntails` 13 nicht korrekt repräsentieren würde.

Für die Smoothness müssen wir garantieren, dass das kanonische Modell die Smoothness Bedingung erfüllt, welche spezifisch für die Korrektheit der Cautious Monotonicity Regel erforderlich ist.

Beide dieser Eigenschaften sind nicht automatisch durch die Konstruktion des Modells gegeben und müssen daher explizit nachgewiesen werden.

Um die Minimalität nachzuweisen, müssen wir zeigen, dass ein Zustand `w`, welcher eine Formel `p` enthält, auch minimal für `p` ist. Dabei muss diese Formalisierung die Definition von `MinimalElements` 3 erfüllen und müssen demnach sowohl zeigen, dass `p` in `w` gilt, und dass es keinen präferierten Zustand gibt, der ebenfalls `p` erfüllt.

Lemma 3. *Minimale Elemente im kanonischen Modell*

```
1  Axiom canonical_states_maximal :
2    forall w : CanonicalStates, maximal_consistent_set w.

4  Axiom canonical_minimality :
5    forall (p : Formula) (w : CanonicalStates),
6      p ∈ w ->
7      ∼ exists state',
8      p ∈ state' /\ CanonicalPreferenceRel state' w.

10 Lemma minimal_elements_canonical :
```

```
11    forall (p : Formula) (w : CanonicalStates),
12      maximal_consistent_set w ->
13      p ∈ w ->
14      In CanonicalStates (MinimalElements CanonicalModel p) w.
15  Proof.
16    intros p w H_max H_p_in_w.
17    unfold MinimalElements.
18    split.
19    - apply canonical_entails; auto.
20    - intros [state' [H_entails_state' H_pref]].

22      assert (H_max_state' : maximal_consistent_set state').
23      apply canonical_states_maximal.

25      assert (H_p_in_state' : p ∈ state').
26      apply canonical_entails; auto.

28      assert (H_no_preferred : ~ exists state',
29        p ∈  state' /\ CanonicalPreferenceRel state' w).
30      apply canonical_minimality; auto.

32      apply H_no_preferred.
33      exists state'.
34      split; auto.
35  Qed.
```

Das Lemma zeigt, dass jede maximale konsistente Menge `w`, die eine Formel `p` enthält, auch direkt ein minimales Element für `p` im kanonischen Modell ist. Dabei können wir in dem Beweis durch `canonical_entails` 3 zeigen, dass `p` in `w` semantisch wahr ist und zeigen dann über einen Widerspruch, dass es keinen präferierten Zustand gibt, der ebenfalls `p` erfüllt. Um den Beweis der Minimalität zu vereinfachen, führen wir zwei Axiome ein. Das Axiom `canonical_states_maximal` sagt aus, dass jeder Zustand im kanonischen Modell maximal konsistent ist, was uns die Anwendung von `canonical_entails` ermöglicht, und das zweite Axiom `canonical_minimality` garantiert dabei, dass für einen Zustand `w` mit `p ∈ w` kein präferierter Zustand existiert, der ebenfalls `p` enthält. Für den Widerspruch wird angenommen, dass es einen präferierten Zustand `state'` gibt, welcher `p` erfüllt. Mit den Axiomen `canonical_states_maximal` und `canonical_entails` folgt `p ∈ state'`, was aber durch `canonical_minimality` zu einem Widerspruch führt, da kein solcher Zustand `state'` existieren darf.

Für die Formalisierung der Smoothness-Bedingung für das kanonische Modell führen wir das Lemma `smoothness_canonical` ein.

Lemma 4. *Die Smoothness Eigenschaft des kanonischen Modells*

```
Lemma smoothness_canonical :
  forall (p : Formula) (w : CanonicalStates),
    entails (Labeling CanonicalModel w) p ->
    exists min_w,
      entails (Labeling CanonicalModel min_w) p /\
      (CanonicalPreferenceRel min_w w \/ min_w = w) /\
      In CanonicalStates
        (MinimalElements CanonicalModel p) min_w.
```

Das Lemma sagt aus, dass wenn eine Formel `p` in einem Zustand `w` des kanonischen Modells gilt, dann existiert ein minimaler Zustand `min_w` für `p`, welcher entweder identisch mit `w` ist oder gegenüber `w` präferiert ist und in `MinimalElements` liegt. Wir nutzen hier unser allgemeines `smoothness` Axiom1, um zu zeigen, dass auch die Smoothness Bedingung spezifisch im kanonischen Modell gilt.

5.2.5 Hauptbeweisschritte der Completeness

Da wir nun die Grundlagen für den Vollständigkeitsbeweis geschaffen haben, können wir den abschließenden Beweis in dem Theorem `completeness_klm` führen. Wir zeigen, dass jede semantische Konsequenzrelation auch syntaktisch ableitbar ist, und werden dies ebenfalls durch einen Widerspruch zeigen. Wir nehmen dafür an, dass eine Formel `q` semantisch aus `p` folgt, das heißt, dass alle Modelle, in denen `p` wahr ist, ebenfalls auch `q` erfüllen, aber `q` nicht syntaktisch aus `p` ableitbar ist, also $p \not\sim q$. Dafür konstruieren wir zunächst ein kanonisches Modell, was wir als ein Gegenbeispiel verwenden werden, da wir dieses Modell so konstruieren, dass es die Wissensbasis respektiert und dennoch die zuvor angenommene Nicht-Ableitbarkeit widerspiegelt. Durch die Anwendung der Eigenschaften maximaler konsistenter Mengen und der semantischen Interpretation des kanonischen Modells können wir dann zeigen, dass dieses Gegenbeispiel zu einem logischen Widerspruch führt.

```
Theorem completeness_klm :
  forall (Γ : Ensemble Formula) (p q : Formula),
    Γ |= p |∼w q -> Γ : p |∼ q.
Proof.
  intros Γ p q H_sem.
  destruct (classic (CumulCons Γ p q))
  as [H_syn | H_not_syn].
    exact H_syn.
  - assert (H_sem_check : Γ |= p |∼w q).
    { exact H_sem. }

    destruct (exists_maximal_consistent Γ p q H_not_syn)
    as [w [H_max [H_sub [H_p_in_w H_not_q_in_w]]]].

    assert (H_satisfies :
```

```
16      SatisfiesKnowledgeBase CanonicalModel Γ).
17      apply canonical_satisfies_kb with (w := w); auto.

19      assert (H_minimal : In CanonicalStates
20              (MinimalElements CanonicalModel p) w).
21      apply minimal_elements_canonical; auto.

23      assert (H_entails_q :
24      entails (Labeling CanonicalModel w) q).
25      apply H_sem; auto.

27      assert (H_q_in_w : q ∈ w).
28      apply canonical_entails; auto.

30      contradiction.
31  Qed.
```

Wir führen zunächst unsere Variablen und Annahmen ein. Wieder stellt Γ unsere Wissensbasis und `p` und `q` unsere Formeln dar. Die Annahme `H_sem` besagt, dass Γ `|= p |~w q` semantisch gilt, das heißt, dass in allen kumulativen Modellen, welche die Wissensbasis Γ respektieren, `q` aus `p` folgt. Nachfolgend untersuchen wir, ob `CumulCons Γ p q` gilt oder nicht. Der erste Fall, dass `CumulCons Γ p q` gilt, ist direkt trivial zu beweisen, da wir bereits `H_syn : CumulCons Γ p q` haben. Für den zweiten Fall, dass `CumulCons Γ p q` nicht gilt, wollen wir den Widerspruch herleiten. Dafür konstruieren wir eine maximale konsistente Menge `w`, welche folgende Eigenschaften hat:

1. `H_max` : `w` ist eine maximale konsistente Menge.
2. `H_sub` : Die Wissensbasis Γ ist in `w` enthalten ($\Gamma \subseteq w$).
3. `H_p_in_w` : Die Prämisse `p` ist in `w` enthalten.
4. `H_not_q_in_w` : Und die Konklusion `q` ist nicht in `w` enthalten.

Dabei können wir mithilfe des Axioms `exists_maximal_consistent` zeigen, dass wenn `q` nicht syntaktisch aus `p` unter Γ ableitbar ist, dann gibt es eine konsistente Erweiterung von der Wissensbasis $\Gamma \cup \{p\}$, die nicht `q enthält`.

Nachdem wir die maximale konsistente Menge konstruiert haben, zeigen wir durch das Axiom `canonical_satisfies_kb`, dass das kanonische Modell die Wissensbasis Γ respektiert. Das ist notwendig, damit wir die semantische Annahme `H_sem` auf das kanonische Modell anwenden können, denn Γ `|= p |~w q` gilt nur für Modelle, die Γ respektieren. Darauf folgend müssen wir ebenfalls die von der Konsequenzrelation geforderte Minimalität des Zustands `w` für die Formel `p` im kanonischen Modell nachweisen, um zu zeigen, dass `w` zu den typischsten Zuständen gehört, in denen `p` gilt. Lemma `minimal_elements_canonical` sagt genau dies aus, und wir können es demnach hier anwenden.

Da wir nun gezeigt haben, dass das kanonische Modell die Wissensbasis Γ respektiert und w ein minimaler Zustand für p ist, können wir die Annahme H_sem auf den Zustand w anwenden und Schlussfolgern, dass q ebenfalls in w gelten muss, denn die Definition von Γ |= p |∼w q sagt aus, dass in allen Modellen, die Γ respektieren, q auch in allen minimalen Zuständen gilt, in denen auch p gilt. Genau dies führt uns zu dem Widerspruch. Wir haben eine semantische Aussage über w abgeleitet, welche im Widerspruch zu den syntaktischen Eigenschaften von w stehen wird. Über das Axiom canonical_entails können wir die semantische Aussage „q gilt in w“ in die syntaktische Aussage „q ist ein Element von w“ transformieren. Wir erhalten die Äquivalenz:

```
entails (Labeling CanonicalModel w) q <-> q ∈ w
```

Damit existieren zwei widersprüchliche Aussagen, denn H_q_in_w gibt an, dass q in w enthalten ist, aber H_not_q_in_w sagt genau das Gegenteil aus, dass q nicht in w enthalten ist. Somit erhalten wir einen logischen Widerspruch, der automatisch von Coq erkannt wird, und wir können den Beweis mit **contradiction** abschließen. Der Widerspruch zeigt, dass unsere vorher getroffene Annahme ∼ CumulCons Γ p q falsch war. Wir hatten angenommen, dass q semantisch aus p folgt (Γ |= p |∼w q), aber nicht syntaktisch ableitbar ist ((Γ : p |∼ q)), und konnten einen Widerspruch zeigen. Demnach muss die syntaktische Ableitbarkeit Γ : p |∼ q auch gelten.

Damit haben wir gezeigt, dass für jede semantische Konsequenzrelation eine syntaktische Ableitung existiert, was genau der Completeness-Richtung des Repräsentationstheorems von Kraus, Lehmann und Magidor entspricht.

```
1 Theorem klm_theorem :
2   forall (Γ : Ensemble Formula) (p q : Formula),
3     Γ : p |∼ q <-> Γ |= p |∼w q.
4 Proof.
5   intros Γ p q.
6   split.
7   - apply KLM_Soundness_M.soundness_klm.
8   - apply KLM_Completeness_M.completeness_klm.
9 Qed.
```

Wenn eine Konsequenzrelation durch ein kumulatives Modell definiert werden kann Γ |= p |∼w q, dann ist sie auch eine kumulative Konsequenzrelation des System C Γ : p |∼ q. Zusammen mit dem Soundness-Beweis stellt dies die vollständige Äquivalenz zwischen der syntaktischen Ebene kumulativer Konsequenzrelationen durch System C und ihrer semantischen Repräsentation durch kumulative Modelle dar.

6 Evaluation und Diskussion

In den vorangegangenen Kapiteln haben wir eine Formalisierung des Repräsentationstheorems für kumulative Konsequenzrelationen in Coq formalisiert. Diese Formalisierung umfasst sowohl die syntaktische Ebene durch System C als auch die semantische Repräsentation durch kumulative Modelle, einschließlich des Soundness und Completeness Beweises.

Zum einen abstrahieren wir komplexe mathematische Konstruktionen durch Axiome, zum anderen werden theoretisch begründbare Eigenschaften axiomatisch angenommen, deren expliziter Beweis den Rahmen einer verständlichen Formalisierung sprengen würde und damit die Komplexität deutlich erhöht. Die Smoothness-Bedingung, wie schon in Kapitel 2.3.1 besprochen, ist beispielsweise in der endlichen propositionalen Logik automatisch erfüllt und wird daher, wie auch schon von Kraus, Lehmann und Magidor, als „technische Bedingung“ axiomatisch behandelt.

Diese Designentscheidungen ermöglichen uns, den Fokus auf die wesentlichen Konzepte des nichtmonotonen Schließens zu legen, ohne sich in den technischen Details der Implementierung oder in theoretisch bereits geklärten Nebenaspekten zu verlieren. Dabei unterscheiden wir für die Formalisierung bewusst zwischen theoretisch begründeten Axiomen und denen, die die komplexen Konstruktionen des Completeness Beweises abstrahieren.

In diesem Kapitel werden wir die gegebene Formalisierung unter verschiedenen Gesichtspunkten kritisch evaluieren und dabei sowohl die Stärken als auch die Grenzen des gewählten Ansatzes diskutieren, die aufgetretenen Implementierungsherausforderungen analysieren, welche sich aus dem nichtmonotonen Schließen und den Besonderheiten der Coq-Formalisierung ergeben, und die getroffenen Designentscheidungen reflektieren.

6.1 Vollständigkeit und Korrektheit

Für den Soundness Beweis haben wir alle fünf Regeln des System C (Reflexivity, LLE, RW, Cut, CM) vollständig formalisiert und bewiesen. Dabei haben wir durch strukturelle Induktion über `CumulCons` jeden einzelnen der fünf Konstruktoren von `CumulCons` bewiesen. Jede Regel wurde dabei durch ein separates Lemma abgesichert, was die Korrektheit und Nachvollziehbarkeit gewährleistet. Der Beweis der Cautious Monotonicity Regel nutzt die Smoothness Bedingung, welche sicherstellt, dass jede Formel minimale Zustände hat, was eine wichtige Eigenschaft kumulativer Modelle ist. Die Beweise der Lemmata für die Soundness wurden von Coq akzeptiert und typgeprüft und sind damit ebenfalls von Coq verifiziert und enthalten keine unbewiesenen Annahmen. Der Beweis stellt zudem eine Interpretation zu Lemma 3.16 und Lemma 3.24 [11] von Kraus, Lehmann und Magidor dar, woran wir uns auch für die Struktur des Beweises orientiert haben.

Der Completeness Beweis folgt einer üblichen Beweisstruktur für die Korrektheit durch einen Widerspruchsbeweis, wie sie in der modalen Logik für die Konstruktion kanonischer Modelle etabliert ist [26, 16]. Dabei verwenden wir die Axiome

konsistent und zweckgemäß und erzeugen eine gültige Schlussfolgerungskette von der Annahme und Modellkonstruktion bis zum Widerspruch. Insbesondere orientiert sich unsere Konstruktion des kanonischen Modells an der Idee maximal konsistenter Mengen, wie sie in [26] und [16] für modale Logiken beschrieben wird, angepasst an die spezifischen Anforderungen unserer bisherigen Formalisierung. Während in [26] eine *Accessibility Relation* definiert wird, verwenden wir eine Präferenzrelation `CanonicalPreferenceRel`, welche auf der Minimalitätsbedingung von KLM-Modellen basiert [11]. Diese Anpassung war notwendig, um die semantischen Eigenschaften des nichtmonotonen Schließens zu berücksichtigen. Die Axiome wie `canonical_entails` und `exists_maximal_consistent` wurden eingeführt, um die Komplexität der unendelichen Modelle abdecken zu können und spiegeln die Aussage von *Lindenbaum's Lemma* [26, 16] wider, welches die Existenz maximaler konsistenter Mengen sichert.

Auch der Completeness Beweis wird von Coq als korrekt verifiziert, was jedoch durch die axiomatischen Annahmen unterstützt wird. Wir haben mit der Formalisierung dennoch alle Komponenten des KLM-Theorems, wie das System C und kumulative Modelle, abgedeckt und haben beide Richtungen (Soundness und Completeness) behandelt. Wir haben dabei sichergestellt, dass alle fünf Regeln des System C korrekt und vollständig formalisiert sind und auch, dass die Modelle alle erforderlichen Komponenten (Zustände, Labeling-Funktion und Präferenzrelation) beinhaltet.

6.2 Komplexität der Formalisierung und Lösungsansätze

Die axiomatischen Abstraktionen im Completeness Beweis sind ein ausschlaggebender Punkt für die Komplexität unserer Formalisierung. Wir haben insgesamt fünf Axiome eingeführt, welche teils aufeinander aufbauen und damit komplexere Konstruktionen ersetzen. Es ist dabei hervorzuheben, dass die fünf Axiome

1. `exists_maximal_consistent`,
2. `canonical_entails`, `canonical_satisfies_kb`,
3. `canonical_states_maximal`
4. und `canonical_minimality`

nicht konstruktiv bewiesen sind, während wir das Axiom `smoothness` als gegebene Vereinfachung durch die Eigenschaften propositionaler Logik sehen.

Während der Implementierung haben wir erkannt, dass die unendliche Größe des kanonischen Modells konstruktive Beweise sehr komplex macht. Die unendliche Größe ist schon selber durch `Ensemble Formula` gegeben, da es sich hier um eine Menge von Formeln handelt. Wir hatten bereits geklärt, dass es in der propositionalen Logik nur eine endliche Anzahl semantisch unterschiedlicher Formeln geben kann. Dies ist jedoch nicht der Fall für syntaktische verschiedene Formeln,

mit denen wir ebenfalls für die Formalisierung arbeiten. Dadurch, dass es also unendlich viele Formeln gibt, aus denen die maximalen konsistenten Mengen gebildet werden, entsteht die Unendlichkeit der Modelle. Die erhöhte Komplexität spiegelt sich gerade in den definierten Axiomen wider.

Die Konstruktion von `exists_maximal_consistent` erfordert dabei eine systematische Behandlung abzählbar unendlich vieler Formeln, denn wir müssen für jede einzelne Formel entscheiden, ob diese in der Menge enthalten ist oder nicht. Dies führt dazu, dass wir darüber unendlich viele Entscheidungen treffen müssten. Die verwendete Library [8] stellt für die Enumeration aller Formeln die Datei und das Lemma `bijection_nat_formula`, also einer bijektion zwischen $\mathbb{N}$ und `Formula` und eine Funktion `maxmapf`, welche für jede Formel entscheiden kann, ob diese in der Menge hinzugefügt werden kann, zur Verfügung. Aber dennoch bleibt es ein unendlicher Prozess, diese Entscheidung zu treffen. Für jede Formel $f_0, f_1, f_2, ...$ müss entschieden werden ob diese der Menge hinzugefügt werden kann:

Starte mit $\Gamma \cup \{p\}$
Formel f_0: Kann f_0 ohne Inkonsistenz hinzugefügt werden?
  $\top \rightarrow$ füge f_0 hinzu
  $\bot \rightarrow$ füge $\neg f_0$ hinzu
Formel f_1: Kann f_1 ohne Inkonsistenz hinzugefügt werden?
  $\top \rightarrow$ füge f_1 hinzu
  $\bot \rightarrow$ füge $\neg f_1$ hinzu
Formel f_2: ...

Wir müssten außerdem zeigen, dass die Konsistenz erhalten bleibt und dass wir auch tatsächlich eine maximale Menge erzeugen. Wir umgehen dieses Problem mit dem Axiom `exists_maximal_consistent`, indem wir postulieren, dass eine solche Menge existiert und diese deshalb nicht konstruieren müssen.

Auch der Beweis von `canonical_entails` ist aufgrund der rekursiven Struktur und aufgetretenen Typisierungsproblemen in Coq zu kompliziert. Auf der semantischen Ebene arbeiten wir für das Entailment mit der syntaktischen Mengenzugehörigkeit `valuemaxf` aus der Library [8], der strukturellen Eigenschaften maximaler konsistenter Mengen, und der semantischen Auswertung durch `entails`. Jede dieser Ebenen fordert eigene Coq-spezifische Typisierungsanforderungen. Frühere Beweisversuche scheiterten an der rekursiven Induktion über die Formelstruktur, da die Verwendung von `CanonicalStates` als abhängiger Typ die Extraktion des `Ensemble Formula` mit dem Verwenden der Funktion `proj1_sig` erforderte und dies zu komplexen Typisierungsfehlern führte.

Das Axiom `canonical_satisfies_kb` abstrahiert die komplexe Konstruktion, welche zeigen würde, dass wenn eine maximale konsistente Menge die Wissensbasis enthält, das entsprechende kanonische Modell diese Wissensbasis respektiert.

Für das Axiom benötigen wir daher einen Nachweis komplexer Modellstruktur-Eigenschaften, wie zum Beispiel, dass alle Zustände des Modells alle Formeln aus der Wissensbasis Γ erfüllen, wo die Unendlichkeit der Zustände wieder problematisch ist. Außerdem müssten wir die Verbindung zwischen der Mengenzugehörigkeit einer Formel φ in Γ ($\varphi \in \Gamma$) und der semantischen Gültigkeit nachweisen. Dafür würden wir selber wieder `canonical_enails` für alle Zustände verwenden, was wiederum selber axiomatisch ist.

Die Axiomatisierung diente daher vorrangig als strategische Entscheidung. Wir haben die Axiome als Lösung für bestimmte Komplexitätsprobleme eingeführt und vermeiden damit technische Details und behalten den Fokus auf den Kernkonzepten des KLM-Theorems. Für die Axiome haben wir uns zudem an den entsprechenden Lemmata aus der Arbeit von Kraus, Lehmann und Magidor orientiert und haben diese nicht einfach willkürlich eingeführt, da jedes dieser Axiome eine klare theoretische Rechtfertigung besitzt.

6.3 Äquivalenzklassen als mögliche Alternative

Es könnte argumentiert werden, dass das Problem der Unendlichkeit der Modelle gelöst werden könnte, indem keine maximalen konsistenten Mengen genutzt werden, sondern Äquivalenzklassen. Jedoch hat sich bei vorhergehenden Ansätzen gezeigt, dass diese deutlich komplexer in Coq zu definieren sind. Es müsste ebenfalls eine Äquivalenzrelation und die Klassenkonstruktion formalisiert werden und dies hätte ebenfalls vollständige Quotientenstrukturen in Coq erfordert.

Dies hätte jedoch ebenfalls die Komplexität erhöht und ebenfalls zu mehreren axiomatischen Annahmen geführt. Letztendlich haben wir uns für die Repräsentation mit maximalen konsistenten Mengen entschieden, auch weil uns diese schon zur Verfügung gestellt wurden. Dabei haben wir darauf geachtet, maximalen konsistenten Mengen so zu konstruieren, dass diese trotz der technischen Unterschiede logisch äquivalent zu den von Kraus, Lehmann und Magidor vorgestellten Äquivalenzklassen sind, da beide die gleichen semantischen Eigenschaften, wie Smothness und Minimalität, repräsentieren.

7 Fazit

Das Repräsentationstheorem für kumulatives Schließen nach Kraus, Lehmann und Magidor [11] bildet die theoretische Grundlage dieser Arbeit. Es beschreibt die Äquivalenz zwischen den syntaktischen Regeln des Systems C und der Semantik kumulativer Modelle und ist zentral für das nichtmonotone Schließen, das Wissen mit Ausnahmen wie „Vögel fliegen, aber Pinguine nicht“ modelliert. Diese Äquivalenz ist essenziell für Anwendungen in der künstlichen Intelligenz, etwa in Expertensystemen. Die Formalisierung in Coq verifiziert diese theoretische Verbindung und schafft eine Grundlage für weitere Forschungen.

Ziel dieser Arbeit war es, das KLM-Theorem in propositionaler Logik mit dem Beweisassistenten Coq zu formalisieren. Dabei wurden die Regeln des Systems C, kumulative Modelle sowie die Beweise für Soundness und Completeness in Coq kodiert, um die Äquivalenz zwischen Syntax und Semantik zu verifizieren. Die Formalisierung schafft eine verifizierte, wiederverwendbare Grundlage für nichtmonotones Schließen und zeigt Potenzial für Erweiterungen, etwa auf präferenzielle Logiken wie System P, sowie für Anwendungen in Expertensystemen. Im Folgenden fassen wir unsere Hauptergebnisse zusammen. Dabei reflektieren wir die erreichten Ziele, zeigen Verbesserungsmöglichkeiten auf und skizzieren danach kurz zukünftige Arbeiten.

7.1 Zusammenfassung der Beiträge

Die Formalisierung umfasst alle Komponenten des KLM-Theorems. Auf der syntaktischen Ebene wurde das System C mit dessen fünf Regeln vollständig formalisiert. Auf der semantischen Ebene wurden kumulative Modelle mit Zuständen, einer Labeling-Funktion und einer Präferenzrelation definiert, wobei maximale konsistente Mengen als Zustände verwendet wurden. Der Soundness-Beweis in Theorem `soundness_klm` zeigt, dass jede vom System C abgeleitete Konsequenz semantisch gültig ist, was durch strukturelle Induktion über `CumulCons` mit separaten Lemmata für jede Regel abgesichert wurde. Der Completeness-Beweis in Theorem `completeness_klm` zeigt, dass jede semantisch gültige Konsequenz syntaktisch ableitbar ist, und nutzt einen Widerspruchsansatz mit einem kanonischen Modell. Beide Beweise wurden von Coq typgeprüft, was ihre Korrektheit bestätigt.

Wir haben damit unser Hauptziel, eine mögliche Formalisierung für das KLM-Theorem mit Coq zu erstellen, erreicht und haben gezeigt, wie System C, kumulative Modelle und die Äquivalenz zwischen diesen mit Coq kodiert werden können. Wir haben uns dabei auf die propositionale Logik beschränkt und Axiome eingeführt, um die Komplexität so gering wie möglich zu halten und den Fokus auf das eigentliche Repräsentationstheorem von Kraus, Lehmann und Magidor zu halten. Dabei haben wir uns für einen modularen Aufbau des Beweises entschieden, um zukünftige Arbeiten zu erleichtern und die Übersichtlichkeit zu erhöhen.

Die Formalisierung stützt sich auf Fähigkeiten von Coq, wie in Kapitel 3 beschrieben. Interaktives Beweisen mit Taktiken wie `induction`, `simpl`, `apply` und

`rewrite`, unterstützt durch die deklarative Programmierung mit Gallina und den Calculus of Inductive Constructions, ermöglichte eine präzise Formalisierung der Beweise. Die Library [8] erleichterte die Arbeit mit propositionaler Logik und implemtierte einige Eigenschaften der propositionalen Logik, welche wir für unsere Formalisierung einsetzten konnten. Herausforderungen wie Typisierungsprobleme und unklare Dokumentation erforderten jedoch eine intensive Einarbeitung, unterstreichen aber die Stärke von Coq für formale Verifikationen als mächtiges Wekzeug.

7.2 Erkenntnisse

Die Formalisierung liefert uns mehrere Erkenntnisse, welche die Herausforderungen aus Kapitel 1 widerspiegeln. Die unendliche Größe des kanonischen Modells stellte dabei eine zentrale Herausforderung dar und führte zu der Einführung unserer Axiome aus Kapitel 5.2.1, welche diese Komplexität umgingen. Aufgetretene Typisierungsfehler und gelegentliche Bulletpoint-Fehler stellten eine unerwartete Herausforderung dar. Es hat sich herausgestellt, dass obgleich der Beweiskontext durch Coq präsentiert werden kann, nicht immer direkt klar oder ersichtlich ist, welchen Typen gerade eine Hypothese besitzt. Oft haben sich bei für uns logischen und nachvollziehbaren Beweisschritten Typisierungsfehler ergeben, die nicht nachvollziehbar erschienen, aber von Coq aufgrund des Type-Checking bemängelt wurden. Außerdem lag ein weiteres Hindernis selber in der Arbeit mit dem Beweisassistenten Coq, denn es zeigte sich, dass die Handhabung und Beweisführung mit Coq nicht intuitiv ist. Die Dokumentation ist teilweise auch nicht klar genug, um nachvollziehen zu können, wie bestimmte, erweiterte oder fortgeschrittenere Taktiken hätten eingesetzt werden können. Dies verzögerte die Beweisführung.

Das Einarbeiten in die genutzte Library [8] stellte ebenfalls eine Herausforderung aufgrund deren Komplexität dar. Dennoch konnte die Library an einigen Stellen sinnvoll eingesetzt werden. Diese Library implementiert einige Konzepte propositionaler Logik und bietet ebenfalls zahlreiche auto-solver Funktionen. Das macht diese aber auch etwas komplexer als ursprünglich angenommen. Jedoch zeigt diese auch bestimmte Beweistechniken und Formatierungen, an denen wir uns orientieren konnten.

7.2.1 Bewährte Praktiken

Im Laufe der Formalisierung haben sich verschiedene Praktiken bewährt. Die Verwendung von Axiomen ist eine effektive Strategie, um die Herausforderungen der unendlichen Größe und komplexe Beweise zu lösen. So konnten wir die Konstruktion des kanonischen Modells vereinfachen und die Formalisierung zugänglicher gestalten. Außerdem war die Orientierung an der Arbeit von Kraus, Lehmann und Magidor [11] entscheidend, um die theoretische Fundierung sicherzustellen, insbesondere bei der Struktur des Soundness und Completeness Beweises. Die Library [8] erleichterte die Arbeit mit maximalen konsistenten Mengen, was die Implementierung effizienter machte. Wir konnten so erfolgreich das System C und die ku-

mulativen Modelle formalisieren und bieten explizite Definitionen für Zustände, Labeling-Funktionen und Präferenzrelationen an, die in Beweisen effektiv genutzt werden können.

7.2.2 Potenzielle Verbesserungen und Alternativen

Trotz der Erfolge gibt es Verbesserungsmöglichkeiten. Die Axiome aus Kapitel 5.2.1 könnten durch explizite Beweise ersetzt werden, um die Abhängigkeit von Annahmen zu reduzieren, auch wenn dies die Komplexität erhöhen würde. Dafür wäre eine detaillierte Analyse der Typisierungsprobleme ebenfalls hilfreich, auch um ähnliche Herausforderungen in zukünftigen Projekten zu vermeiden.

In der Einleitung haben wir die Möglichkeit, die Präferenzrelation so zu strukturieren, dass die Smoothness-Bedingung automatisch erfüllt ist. Dies wurde durch die axiomatische Annahme von `smoothness` erreicht, aber eine explizite Konstruktion wäre auch eine Alternative gewesen. Eine weitere Alternative ist, wie wir schon diskutiert hatten, die Verwendung von Äquivalenzrelationen und Quotientenstrukturen in Coq, die ähnliche Axiome nötig gemacht hätten. Eine dritte Alternative, die Konstruktion einer äquivalenten Formel für maximal konsistente Menge, wurde verworfen, da sich dies als unpraktikabel im Umgang mit propositionaler Logik erwies, weshalb für uns der gewählte Ansatz mit maximal konsistenten Mengen die beste Wahl darstellte.

Die Formalisierung ist dennoch als Erfolg zu werten, da sie das KLM-Theorem vollständig abdeckt. Die Kodierung des System C, die Definition kumulativer Modelle und die Beweise für Soundness und Completeness wurden durch Coq verifiziert, was die Korrektheit und Präzision garantiert. Der Ansatz ist sehr gut nachvollziehbar, da Axiome komplexe Konstruktionen vereinfachen und die Library [8] die Implementierung ebenfalls unterstützt. Schwächen sind jedoch ebenfalls durch die Abhängigkeit dieser Axiome gegeben und beschränken die Eigenständigkeit der Formalisierung. Dennoch bietet unsere Arbeit eine solide Grundlage für das Verständnis nichtmonotonen Schließens und gibt eine formale Verifikation in Coq, die für zukünftige Projekte zugänglich ist.

8 Zukünftige Arbeiten

Die Formalisierung des Repräsentationstheorems für kumulative Konsequenzrelationen (System C) in Coq, wie in dieser Arbeit vorgestellt, bietet eine solide Grundlage für weitere Forschungen und Erweiterungen im Bereich des nichtmonotonen Schließens. Neben der Vervollständigung unvollständiger Beweise und der Erweiterung auf andere Logiken gibt die Arbeit die Möglichkeit, das System P (Preferential Logics), eine Erweiterung von System C, zu formalisieren. Dieser Abschnitt skizziert, wie System P eingeführt werden kann, wie die vorliegende Formalisierung als Basis genutzt werden kann, welche Änderungen im Coq-Code erforderlich sind, und berücksichtigt die Transitivität der Präferenzrelation, die für System P relevant ist.

8.1 System P

Als Erweiterung des Systems C wurde 1990 von Kraus, Lehmann und Magidor [11] das *System P* als präferenzielles Schlussfolgerungssystem vorgestellt. System P fügt eine zusätzliche Regel hinzu, die nichtmonotone Schlüsse mit einer präferenziellen Semantik formalisiert. Diese Regel lautet:

$$\text{(OR)} \quad \frac{a \mathrel{|\!\sim} c \quad b \mathrel{|\!\sim} c}{a \vee b \mathrel{|\!\sim} c} \qquad \text{(Disjunktion)}$$

Die Regel besagt, dass, wenn c üblicherweise aus a folgt und c üblicherweise aus b folgt, dann c auch aus $a \vee b$ folgt. So gilt zum Beispiel „Wenn es regnet, wird der Rasen üblicherweise nass“ ($regen \mathrel{|\!\sim} nass$) und „Wenn der Rasensprenger läuft, wird der Rasen üblicherweise nass“ ($sprenger \mathrel{|\!\sim} nass$). Da beide Prämissen zur gleichen Schlussfolgerung führen, folgt aus „Es regnet oder der Rasensprenger läuft“ ($regen \vee sprenger \mathrel{|\!\sim} nass$), dass der Rasen üblicherweise nass wird. Diese Regel ermöglicht es, disjunktive Prämissen in nichtmonotonen Schlüssen zu behandeln, was die Flexibilität des Systems erhöht.

Semantisch basiert System P auf präferenziellen Modellen, die im Vergleich zu kumulativen Modellen eine striktere Präferenzrelation erfordern. Während die Präferenzrelation in System C nur irreflexiv ist, ist sie in System P sowohl irreflexiv als auch *transitiv*. Die Transitivität stellt sicher, dass, wenn Zustand s_1 präferierter als s_2 ist und s_2 präferierter als s_3, dann s_1 auch präferierter als s_3 ist. Dies spiegelt die Intuition wider, dass „typischere“ Zustände konsistent bevorzugt werden.

8.1.1 Erweiterung auf System P

Die vorliegende Formalisierung von System C bietet eine robuste Basis für die Erweiterung auf System P, da viele Definitionen, Axiome und Beweisstrukturen wiederverwendet oder angepasst werden können. Die Arbeit definiert bereits kumulative Modelle (`CumulModel`), das syntaktische System C (`CumulCons`), maximale konsistente Mengen (`CanonicalStates`), sowie die Beweise für die Soundness

(soundness_klm) und Completeness (completeness_klm). Im Folgenden skizzieren wir, wie diese Komponenten für System P angepasst werden können, und zeigen grob, welche Änderungen im Coq-Code nötig sind.

Syntaktische Anpassungen Um System P zu formalisieren, muss die Regel OR in die Definition von CumulCons in KLM_Cumulative.v integriert werden. Dies erfordert einen neuen Konstruktor:

```
| OR : forall a b c : Formula,
  CumulCons Γ a c ->
  CumulCons Γ b c ->
  CumulCons Γ (a ∨ b) c
```

Dieser Konstruktor repräsentiert die Disjunktionsregel und ermöglicht es, dass $a \vee b \mid\sim c$ abgeleitet wird, wenn $a \mid\sim c$ und $b \mid\sim c$ gelten. Für den Soundness-Beweis (KLM_Soundness.v) müsste ein neues Lemma soundness_OR hinzugefügt werden, das zeigt, dass die Regel semantisch gültig ist. Ein Ansatz wäre:

```
Lemma soundness_OR :
  forall (model : CumulModel) (a b c : Formula),
    model : a |∼w c -> model : b |∼w c ->
    model : (a ∨ b) |∼w c.
Proof.
 [...]
Qed.
```

Dieser Beweis könnte die Tatsache nutzen, dass minimale Zustände, die $a \vee b$ erfüllen, entweder a oder b erfüllen, und wendet dann die entsprechende Prämisse an.

Semantische Anpassungen Semantisch erfordert System P die Definition von *PreferentialModel*, die eine transitive Präferenzrelation haben. In KLM_Semantics.v müsste die Definition von CumulModel angepasst werden, um die Transitivität zu garantieren:

```
Record CumulModel : Type := {
  States : Type;
  Labeling : States -> Formula -> bool;
  PreferenceRel : States -> States -> Prop;
  PreferenceIrreflexive : forall s, ∼ PreferenceRel s s;
  PreferenceTransitive : forall s1 s2 s3,
  PreferenceRel s1 s2 ->
  PreferenceRel s2 s3 -> PreferenceRel s1 s3
}.
```

Die bestehende Smoothness Bedingung (`smoothness`) bleibt erhalten, da sie auch für System P gilt. Im kanonischen Modell (`CanonicalModel`) muss aber die Präferenzrelation entsprechend angepasst werden:

```
1 Definition CanonicalPreferenceRel
2 (w1 w2 : CanonicalStates) : Prop :=
3   exists p, w1 ⊢ p /\ ∼ (w2 ⊢  p) /\
4   forall w3, (forall q, w2 ⊢  q ->
5          w3 ⊢ q) ->
6          w1 ⊢ p  ->
7          w3 ⊢ p.
```

Dies stellt sicher, dass die Präferenzrelation transitiv ist, indem diese die Implikationskette über Zustände respektiert. Der Beweis von `smoothness_canonical` würde durch die Transitivität aber auch komplexer, da minimale Zustände in einer transitiven Ordnung strenger definiert sind. Die unendliche Größe des Modells bleibt eine Herausforderung, wird aber durch die bestehenden Axiome wie `exists_maximal_consistent` weiterhin gehandhabt.

Unsere Formalisierung von System C bietet eine direkte Basis für System P. Die Definitionen von `Formula`, `Ensemble Formula`, und `CanonicalStates` können unverändert übernommen werden. Die Library [8] unterstützt mit Funktionen wie `bijection_nat_formula` und `valuemaxf` weiterhin die Konstruktion maximal konsistenter Mengen. Axiome wie `canonical_satisfies_kb` und `canonical_minimality` sind für System P kompatibel, da sie allgemeine Eigenschaften des kanonischen Modells abdecken. Auch Soundness und Completeness Beweise folgen einer ähnlichen Struktur, wobei die neue Regel `OR` und die Transitivität die Hauptunterschiede darstellen.

8.2 Mögliche Anwendungsbereiche

Ein vielversprechender Anwendungsbereich der Formalisierung ist der Einsatz in Expertensystemen. Dadurch, dass die Verifikation von Coq die Korrektheit des Programms garantiert, könnte unsere Formalisierung für Expertensysteme eingesetzt werden, beispielsweise in medizinischen Diagnosesystemen oder Entscheidungsunterstützungssystemen. Nichtmonotones Schließen, wie es durch System C und System P ermöglicht wird, ist ideal für solche Anwendungen, da es Ausnahmen und typische Schlüsse handhaben kann, wie im Beispiel „Vögel fliegen, aber Pinguine nicht“ aus der Einleitung illustriert. Die durch Coq gewährleistete Korrektheit erhöht das Vertrauen in solche Systeme, insbesondere in kritischen Bereichen, wo Fehler schwerwiegende Konsequenzen haben könnten. Unsere Formalisierung könnte als Modul in größere KI-Systeme integriert werden, etwa durch Extraktion des Coq-Codes in funktionale Programme, wie Haskell [14] oder OCaml [23], um Schlussfolgerungsregeln direkt anzuwenden. Eine Erweiterung auf System P würde die Anwendbarkeit weiter verbessern, da die präferenzielle Semantik „typischere“ Zustände bevorzugt, was für realistische Entscheidungsfindung in Expertensys-

temen nützlich ist. Zudem erleichtert die Wiederverwendbarkeit der Definitionen, wie in der Einleitung betont, die Integration in solche Systeme und unterstützt die Entwicklung vertrauenswürdiger KI-Anwendungen.

Das Extrahieren des Coq-Codes in OCaml oder Haskell ermöglicht es, die Formalisierung ausführbar zu machen und in praktischen Anwendungen zu nutzen. OCaml ist besonders geeignet, da Coq selbst in OCaml geschrieben ist und die Extraktion nativ unterstützt, was zu effizientem Code für Definitionen wie `CumulCons` oder `SemanticEntails` führt. Haskell eignet sich durch sein reines funktionales Paradigma und starkes Typsystem, das gut zu Coq's mathematischen Konstrukten passt, insbesondere für Beweise wie `soundness_klm`. Außerdem erhöht dies die Wiederverwendbarkeit der Formalisierung, wie in der Einleitung betont, und ermöglicht die Integration in KI-Frameworks für Expertensysteme. Dabei bleibt die Korrektheit durch Coq's Verifikation erhalten, was wiederum das Vertrauen in kritische Anwendungen stärkt. Der Extraktionsprozess erfolgt dabei durch Markieren extrahierbarer Definitionen in Coq (zum Beispiel mit `Extract Inductive`), Entfernen oder Ersetzen von Axiomen wie `exists_maximal_consistent` durch konkrete Implementierungen und Verwenden des Befehls `Extraction` nach OCaml oder Haskell, wie in [20] beschrieben. Der extrahierte Code kann dann mit einem OCaml oder Haskell Compiler kompiliert werden, um die Schlussfolgerungsregeln direkt anzuwenden.

Damit sind eine Erweiterung auf System P und die Anwendung in Expertensystemen vielversprechende nächste Schritte, da sie die vorliegende Formalisierung direkt nutzen und die versehentliche Transitivität der Präferenzrelation integrieren. Die Arbeit bietet damit eine solide Grundlage für die formale Verifikation präferenzieller Logiken und trägt zur Weiterentwicklung des nichtmonotonen Schließens in Coq sowie dessen praktischer Anwendung bei.

Literatur

[1] Adel Mohammed Al-Odhari. Features of propositional logic. *Pure Mathematical Sciences*, 10(1):35–44, 2021.

[2] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

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

[4] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[5] Thierry Coquand and Gérard Huet. *The calculus of constructions*. PhD thesis, INRIA, 1986.

[6] Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[8] Dakai Guo and Wensheng Yu. A comprehensive formalization of propositional logic in coq: Deduction systems, meta-theorems, and automation tactics. *Mathematics*, 11(11), 2023.

[9] William A Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[10] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[11] Sarit Kraus, Daniel Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models, and cumulative logics. *Artificial Intelligence*, 44(1):167–207, 1990.

[12] Daniel Lehmann and Menachem Magidor. What does a conditional knowledge base entail? *Artificial intelligence*, 55(1):1–60, 1992.

[13] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[14] Simon Marlow et al. *Haskell 2010 Language Report*. Haskell.org, 2010. Accessed May 12, 2025.

[15] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical Logic and Foundations)*. College Publications, 2015.

[16] Open Logic Project. Completeness and canonical models. Chapter, Open Logic Project, December 2024. Revision: 6891b66, licensed under CC-BY.

[17] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. pearson, 2016.

[18] Morten Heine Sørensen and Paweł Urzyczyn. Chapter 14 - pure type systems and the λ-cube. In Morten Heine Sørensen and Paweł Urzyczyn, editors, *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*, pages 343–359. Elsevier, 2006.

[19] Coq Team. Introduction the coq proof assistant https://coq.inria.fr/doc/v8.7.2/refman/introduction.html.

[20] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Inria, 2025. Version 8.19, accessed 2025-04.

[21] The Coq Development Team. *Tactics - The Coq Reference Manual*. Inria, 2025. Accessed: 2025-04.

[22] The Mathematical Components Team. The mathematical components library, 2025. Accessed May 25, 2025.

[23] The OCaml Team. *The OCaml System: Documentation and User's Manual*. Inria, 2025. Version 5.2, accessed May 12, 2025.

[24] Philip Wadler. The girard–reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1):201–226, 2007.

[25] Jonathan Heinrich Walther. Einführende Beispiele für das Beweisen mit Coq https://github.com/jonawa-q9677453/KLMCoq, 2025. Siehe Datei coq_examples_introduction.v.

[26] Edward N Zalta. Basic concepts in modal logic. *Center for the Study of Language and Information*, 1995.