HOLCF: Eine konservative Erweiterung von HOL um LCF

Franz Regensburger

Fakultät für Informatik
der Technischen Universität München

HOLCF: Eine konservative Erweiterung von HOL um LCF

Franz Regensburger

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Christoph Zenger
Prüfer der Dissertation:	
1.	Univ.-Prof. Tobias Nipkow, Ph.D.
2.	Univ.-Prof. Dr. Manfred Broy

Die Dissertation wurde am 20. Oktober 1994 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 30. November 1994 angenommen.

Kurzfassung

Auf der Basis von HOLC (Higher-Order Logic with Classes) wird die bekannte Logik LCF (Scott: Logic of Computable Functions) durch konservative Theorieerweiterungen entwickelt. Dadurch entsteht die LCF-Variante höherer Stufe HOLCF (Higher-Order Logic of Computable Functions).

Die als Basis benutzte Logik HOLC ist eine Version der klassischen Logik höherer Stufe HOL (Church, Gordon), die neben einem Hindley/Milner Polymorphismus zusätzlich das Konzept von Typklassen ähnlich dem der Programmiersprache HASKELL zur Verfügung stellt. Typklassen bieten im Gegensatz zum einfachen Hindley/Milner Polymorphismus die Möglichkeit, den Grad der Polymorphie einer Funktion oder eines Typkonstruktors genauer zu dosieren.

Für diese Version von HOL lag bereits eine Instanz im generischen Theorembeweiser Isabelle vor (Paulson, Nipkow). Um allerdings die Methode der konservativen Erweiterung von HOL an die Variante HOLC anpassen zu können, wird in der vorliegenden Arbeit als erstes eine Semantik für HOLC entwickelt, die dann als Grundlage für die Definition konservativer Erweiterungsmechanismen im Zusammenhang mit Typklassen dient. Eine Theorie Th_2 wird hierbei als konservative Erweiterung der Theorie Th_1 bezeichnet, wenn jedes Modell der Theorie Th_1 streng persistent zu einem Modell der Theorie Th_2 erweitert werden kann.

Anschließend wird die Logik HOLC durch konservative Theorieerweiterung schrittweise um Begriffe für Bereichstheorie erweitert. Das Endergebnis dieser Entwicklung ist HOLCF, eine in HOLC formalisierte Theorie für Bereiche, die die Logik LCF vollständig beinhaltet. In HOLCF sind Begriffe wie ω-Ketten, (kleinste) obere Schranken, Stetigkeit von Funktionen oder zulässige Prädikate explizit definiert.

Die Erweiterung von HOLC um die Logik von LCF erfordert aber die saubere Trennung zwischen Typen, die nur als Mengen ohne zusätzliche Struktur interpretiert werden (Mengentypen) und Typen, die zum Beispiel als Mengen mit ω-cpo Struktur interpretiert werden (Bereichstypen). Dieses technische Problem wird mit Hilfe der Typklassen gelöst. Insbesondere können mittels Typklassen der Typkonstruktor für den vollen Funktionenraum über Mengentypen und der Typkonstruktor für den Raum der stetigen Funktionen über Bereichstypen syntaktisch unterschieden werden. Für beide Funktionstypen gibt es in HOLCF eine separate λ-Abstraktion und Applikation.

Die Erweiterung von HOLC um LCF in kleinen, konservativen Teilschritten war die technische Herausforderung der Arbeit. Um nicht nur die theoretische Machbarkeit sondern auch die praktische Durchführbarkeit und Benutzbarkeit der Erweiterung zu demonstrieren, wurde diese mit dem generischen Theorembeweiser Isabelle durchgeführt. Somit steht eine sehr leistungsfähige Implementierung der Logik HOLCF zur Verfügung.

Danksagung

Für Kommentare zu Vorversionen dieser Arbeit bedanke ich mich bei Burkhart Wolff, Oskar Slotosch, Max Fuchs und Cornelia Pusch. Ebenso bedanke ich mich bei Thomas Streicher und Bernhard Reus für die hilfreichen Gespräche über rekursive Bereiche und deren Formalisierung in der Kategorientheorie.

Weiterhin möchte ich mich bei Tobias Nipkow und Manfred Broy dafür bedanken, daß sie mich zur Erstellung der Arbeit ermutigt haben und durch ihre Unterstützung die notwendigen Rahmenbedingungen geschaffen haben. Insbesondere gilt mein Dank Tobias Nipkow für seinen fachlichen Rat und die viele Zeit, die er mir geopfert hat.

Nicht zuletzt danke ich allen meinen Bekannten, meiner Familie und vor allem Cornelia für die Geduld und Nachsicht, die sie während der Erstellung dieser Arbeit mit mir hatten.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Einleitung

Alle technischen Disziplinen, die in irgendeiner Weise in die industrielle Produktentwicklung involviert sind, haben über kurz oder lang sowohl Formalismen zur Quantifizierung von Qualität als auch spezifische Formen der Kommunikation entwickelt. Ein exzellentes Beispiel hierfür bietet der Maschinenbau mit seiner immensen Anzahl von Normen und seiner streng reglementierten Form der Kommunikation, den Konstruktionszeichnungen.

Ein formaler Begriff von Qualität sowie eine technikspezifische Form der Kommunikation sind die Grundvoraussetzungen für die ingenieurhafte Anwendung einer technischen Disziplin, und diese wiederum ist Voraussetzung für eine erfolgreiche Anwendung in der industriellen Praxis.

Darüber hinaus ist es für den Einsatz in der Praxis wichtig, daß der Beschreibungsrahmen, die Kommunikationsform, möglichst stufenlos ineinander übergreifende Abstraktionsebenen zuläßt, die zudem noch nahtlos in die semiformale bzw. informelle Beschreibungsebene übergehen sollten. Das eben angeführte Beispiel des Maschinenbaus zeigt, daß dies durchaus möglich ist.

Die Herstellung von Software gleicht zu einem sehr großen Prozentsatz der Herstellung beliebiger anderer Produkte, und in der industriellen Praxis bewährte Methoden der Produktentwicklung und der Qualitätssicherung finden bis zu einem gewissen Grad ihre Anwendung. Diese angepaßten Methoden stellen den Hauptanteil dessen, was heute unter dem Schlagwort *Software–Engineering* bekannt ist. Ein prominentes Beispiel hierfür ist die in Großbritannien entwickelte SSADM-Methode [DCC92], die in der Industrie auch eingesetzt wird.

Der Nachteil der gerade erwähnten Methoden besteht darin, daß sie eben nur angepaßte Methoden sind und der spezifischen Problematik der Softwareentwicklung nur bis zu einem gewissen Grad gerecht werden. Dies wird besonders deutlich, wenn man bedenkt, daß das Endprodukt jeder Softwareentwicklung ein Programm ist, ein vollständig formales Objekt. Die bekannten Software-Engineering-Methoden arbeiten jedoch bestenfalls mit semiformalen Techniken.

Um aber die Qualität eines Softwareprodukts exakt quantifizieren zu können, bedarf es eines Formalismus, der in der Lage ist, die Anforderungen an das Produkt, den Entwicklungsprozeß und auch sein Endprodukt, das Programm, in einem einheitlichen Rahmen darzustellen.

Ein Rückblick in die noch sehr kurze Geschichte der Informatik zeigt, daß ein Großteil der wissenschaftlichen Bemühungen auf dem Gebiet der Entwicklung von Programmiersprachen, Spezifikationssprachen und Methoden der Programmentwicklung und Verifikation darauf gerichtet war (und noch immer ist), einen solchen Rahmen zu schaffen.

Ein Beispiel für diese Art der Forschung ist das derzeit im Abschluß befindliche BMFT[1] Projekt KORSO[2] [HLR93, BW93]. In diesem Verbundprojekt wurde versucht, bereits etablierte Techniken der Softwarespezifikation, der Softwareentwicklung und der Verifikation von Software in einen einheitlichen Methoden– und Technikrahmen zu integrieren. Als Teil dieses Projekts wurde an der Technischen Universität München unter der Leitung von Manfred Broy die Spezifikationssprache SPECTRUM entwickelt. Eine Beschreibung der Sprache SPECTRUM findet sich in [BFG+93a, BFG+93b]. Die technischen Hintergründe der Semantik und Logik, für die Radu Grosu und ich verantwortlich waren, sind in [GR94] beschrieben.

Die hier vorliegende Arbeit ist ein Beitrag zur Fundierung von Sprachen bzw. Logiken wie etwa SPECTRUM. Auf der Basis von HOLC (Higher-Order Logic with Classes) wird die bekannte Logik LCF (Scott: Logic of Computable Functions) [GMW79] durch konservative Theorieerweiterungen entwickelt[3]. Dadurch entsteht die LCF-Variante höherer Stufe HOLCF (Higher-Order Logic of Computable Functions).

Die als Basis benutzte Logik HOLC ist eine Version der klassischen Logik höherer Stufe HOL [And86, Gor85, GM93], die neben einem Hindley/Milner Polymorphismus [Mil78, DM82] zusätzlich das Konzept von Typklassen [NP93] ähnlich dem der Programmiersprache HASKELL [HJW92] zur Verfügung stellt. Typklassen bieten im Gegensatz zum einfachen Hindley/Milner Polymorphismus die Möglichkeit, den Grad der Polymorphie einer Funktion oder eines Typkonstruktors genauer zu dosieren. Für diese Version von HOL lag bereits eine Instanz im generischen Theorembeweiser Isabelle [Pau94] vor, um allerdings die Methode der konservativen Erweiterung von HOL an die Variante HOLC anpassen zu können, mußte in der vorliegenden Arbeit erst noch eine Semantik für HOLC entwickelt werden.

Die Motivation für die Arbeit war der Umstand, daß in der Logik LCF viele der semantischen Eigenschaften der Logik syntaktisch nicht ausgedrückt werden können bzw. die Handhabung der Logik in einigen Fällen unbequem ist. Obwohl sie zentrale Begriffe der Semantik von LCF sind, ist es in LCF zum Beispiel nicht möglich, explizit über ω-Ketten, (kleinste) obere Schranken, Stetigkeit von Funktionen oder zulässige Prädikate zu reden. In HOLC dagegen können diese semantischen Konzepte explizit gemacht werden. Durch konservative Theorieerweiterung wird die Logik HOLC schrittweise um Begriffe für Bereichstheorie erweitert. Das Endergebnis dieser Entwicklung ist HOLCF, eine in HOLC formalisierte Theorie für Bereiche, die die Logik LCF vollständig beinhaltet.

Die oben angesprochene Erweiterung von HOLC um die Logik von LCF erfordert aber die saubere Trennung zwischen Typen, die nur als Mengen ohne zusätzliche Struktur interpretiert werden (Mengentypen) und Typen, die zum Beispiel als Mengen mit ω-cpo Struktur interpretiert werden (Bereichstypen). Dieses technische Problem wird mit Hilfe der Typklassen gelöst. Insbesondere können mittels Typklassen der Typkonstruktor für den vollen Funktionenraum über Mengentypen und der Typkonstruktor für den Raum der stetigen Funktionen

[1] Bundesministerium für Forschung und Technik der Bundesrepublik Deutschland.

[2] Korrekte Software.

[3] eine Theorie Th_2 wird hierbei als konservative Erweiterung der Theorie Th_1 bezeichnet, wenn jedes Modell der Theorie Th_1 streng persistent [EGL89] zu einem Modell der Theorie Th_2 erweitert werden kann.

über Bereichstypen syntaktisch unterschieden werden. Für beide Funktionstypen gibt es in HOLCF eine separate λ-Abstraktion und Applikation.

Die Erweiterung von HOLC um LCF in kleinen, konservativen Teilschritten war die technische Herausforderung der Arbeit. Um nicht nur die theoretische Machbarkeit sondern auch die praktische Durchführbarkeit und Benutzbarkeit der Erweiterung zu demonstrieren, wurde diese mit dem generischen Theorembeweiser Isabelle durchgeführt. Somit steht eine sehr leistungsfähige Implementierung der Logik HOLCF zur Verfügung.

In den folgenden Abschnitten werde ich die Motivation und die Zielsetzung für meine Arbeit genauer beschreiben. Ich möchte dabei historisch vorgehen und die Entwicklung der Motivation und Zielsetzung aufzeigen. Zuerst werde ich in Abschnitt 1.2 einige Aspekte der Spezifikationssprache SPECTRUM behandeln. Damit lassen sich die ursprünglichen Ziele für meine Arbeit erklären, die in Abschnitt 1.3 dargelegt sind. Eine konsequente Weiterführung der Gedankengänge führt dann zu einer Erweiterung der Zielsetzung, was in Abschnitt 1.4 dargestellt ist. Die letztendlichen Ziele und die Thesen der Arbeit sind dann in Abschnitt 1.5 prägnant zusammengefaßt. Auf verwandte Ansätze gehe ich kurz in Abschnitt 1.6 ein. Der Abschnitt 1.7 über den Aufbau meiner Arbeit beschließt dann dieses einführende Kapitel.

1.2 Bemerkungen zur Spezifikationssprache SPECTRUM

Das wesentliche Entwicklungsziel für SPECTRUM war die Konzeption einer reichhaltigen Spezifikationssprache, die es erlaubt, Software und, auf einem relativ abstrakten Niveau, auch Hardware zu beschreiben. Ein besonderes Merkmal von SPECTRUM ist die Kombination von Techniken aus der Schule der algebraischen Spezifikationssprachen ASL [SW83], PLUSS [Gau86], LARCH [GHW85] und bekannten Techniken aus λ-Kalkül-basierten Programmiersprachen HASKELL [HJW92], SML [HMM86, Pau92] bzw. Logiken wie LCF [GMW79].

Der algebraische Einfluß beschränkt sich im wesentlichen auf die Konstrukte des *Spezifizierens im Großen*, d.h. der Kombination von Spezifikationseinheiten. Auf der Ebene des *Spezifizierens im Kleinen* dominieren hingegen Konzepte der λ-Kalkül-basierten Programmiersprachen, den sogenannten funktionalen Programmiersprachen. Hierzu zählen speziell Funktionen höherer Stufe (Funktionale) und ein polymorphes Typsystem mit Typklassen.

Für die Konstrukte des Spezifizierens im Großen wurde eine transformationelle Semantik angegeben, die es erlaubt, komplexe hierarchische Spezifikationen in eine einzige flache Spezifikation umzuformen, die nur noch Konstrukte des Spezifizierens im Kleinen enthält. Diese flachen Spezifikationen können dann durch Expansion von Spezifikationsmakros[4] und einfache syntaktische Transliteration in eine Logik übersetzt werden, die im wesentlichen der Logik LCF entspricht [Pus94a]. In dieser Logik können dann Eigenschaften der Spezifikationen, wie zum Beispiel die Erfüllung von Verifikationsbedingungen, hergeleitet werden.

Die Logik von SPECTRUM erlaubt die uneingeschränkte Formulierung von Axiomen der mehrsortigen Prädikatenlogik erster Stufe, wobei der Typ der stetigen Funktionen ebenfalls ein Typ von Individuen ist, über die quantifiziert werden darf. Zudem stehen in eingeschränkter Form auch Konstrukte der zweiten Stufe zur Verfügung, die die Formulierung von Induktionsprinzipien für Datentypen erlauben. Somit können in SPECTRUM nichtkonstruktive Spezifikationen

[4] dazu zählen etwa das *data*-Konstrukt, sowie die Annotationen *strict* und *total*.

formuliert werden, was sich im Hinblick auf den Abstraktionsgrad von Anforderungsspezifikationen als vorteilhaft erwiesen hat [SHNR93]. Natürlich kann man bei entsprechender Beschränkung auch konstruktive Spezifikationen schreiben, die direkt als Programme, speziell funktionale Programme, interpretiert werden können. Die schrittweise Überführung von nichtkonstruktiven Anforderungsspezifikationen über eine Designspezifikation in eine ausführbare Spezifikation wurde zum Beispiel in [SHNR93] vorgestellt und in [Pus94b] verifiziert. Diese Methode stellt die im Umfeld der Sprache SPECTRUM favorisierte Entwicklungsmethode dar.

Wie oben bereits erwähnt, ist die Logik von SPECTRUM der Logik LCF sehr ähnlich. Sie unterscheidet sich jedoch auch in einigen Punkten von dieser.

Ein bei der Anwendung der Logik durchaus spürbarer, wenn auch nicht allzu großer Unterschied besteht darin, daß SPECTRUM im Gegensatz zur zweiwertigen Logik von LCF eine dreiwertige Logik benutzt. Die Verwendung der dreiwertigen Logik hat ihren Ursprung in dem Wunsch, Formeln als Spezialfall von Termen des dreielementigen Datentyps *bool* zu behandeln. Somit muß zumindest vom Typsystem her kein Unterschied mehr zwischen Formeln und Termen des Typs *bool* gemacht werden. Über die methodischen und auch didaktischen Vorteile einer solchen Vereinheitlichung, die zum Beispiel durch die Verwendung von dreiwertiger Logik erzielt werden kann, läßt sich streiten. Was allerdings die mit der dreiwertigen Logik verbundenen Auswirkungen auf den Kalkül von SPECTRUM und die daraus resultierende umständliche Handhabung des Kalküls angeht [Pus94b], tendiere ich aufgrund meiner mittlerweile erworbenen Erfahrung mit dem dreiwertigen Kalkül von SPECTRUM und der in dieser Arbeit vorgestellten zweiwertigen Logik HOLCF eindeutig wieder zur zweiwertigen Logik.

Der wesentliche Unterschied zwischen der Logik von SPECTRUM und LCF besteht darin, daß in SPECTRUM neben dem Typ der stetigen Funktionen $\alpha \to \beta$ über dem Argumentbereich[5] α und dem Bildbereich β auch der Typ aller totalen Funktionen $\alpha \Rightarrow \beta$ von α nach β zur Verfügung steht[6]. Diese Erweiterung gegenüber LCF ermöglicht die Kodierung von Prädikaten über dem Bereich α als volle Funktionen vom Typ $\alpha \Rightarrow$ *bool*, wie sie aus imprädikativen Logiken der höheren Stufe, etwa HOL [And86, Gor85, GM93], bekannt ist. Die Verwendung von beliebigen Prädikatensymbolen ist bereits ein deutlicher Schritt über LCF hinaus. Daneben können aber auch volle Funktionen mit beliebigem Bildbereich spezifiziert werden, etwa die nichtmonotone Konkatenation von Strömen.

Um die Komplexität der Semantik etwas im Zaum zu halten, wurde die Verwendung von vollen Funktionen drastisch eingeschränkt. Zum einen gibt es keine Variablen vom Typ $\alpha \Rightarrow \beta$, zum anderen darf der Typkonstruktor $\Rightarrow$ nur an äußerster Stelle im Typterm auftreten. Typterme wie $(\alpha \Rightarrow \beta) \times \gamma$ oder $(\alpha \Rightarrow \beta) \Rightarrow \gamma$ sind somit nicht erlaubt. Weiterhin wurde die Verwendung der λ-Abstraktion dahingehend eingeschränkt, daß durch sie nur stetige Funktionen gebildet werden können. Ist etwa die Konstante f vom Typ $\alpha \Rightarrow \beta$, so wird die Bildung des Terms $\lambda x.fx$, der per Extensionalität die gleiche Semantik hätte wie der Term f, durch die Prüfung einer Kontextbedingung, dem sogenannten †-Test[7], untersagt.

[5] unter Bereichen verstehe ich in diesem Zusammenhang partielle Ordnungen, die bzgl. ω-Ketten vollständig sind und ein kleinstes Element enthalten.

[6] diese nicht durch Stetigkeitsannahmen eingeschränkten vollen Funktionen werden in SPECTRUM *mappings* genannt. In SPECTRUM wird der Konstruktor für den vollen Funktionenraum mit *to* bezeichnet. In der hier vorgestellten Logic HOLCF wähle ich statt dessen die Bezeichnung $\Rightarrow$.

[7] sprich 'dagger'-Test. Dieser Test wird ausführlich in [GR94] behandelt.

Durch den obigen Katalog von Einschränkungen schützt man sich vor Inkonsistenzen der Logik, die durch die unvorsichtige Mischung von stetigen und nichtstetigen Funktionen entstehen könnten. Speziell verhindert man damit aber auch die Bildung einer Logik höherer Stufe.

Wenn man nun auf die letzten Absätze zurückblickt oder den technischen Bericht [GR94] liest, so entsteht zu Recht der Eindruck, daß in SPECTRUM neben anderen gelungenen Erweiterungen auch das durchaus interessante Konzept der nichtstetigen Funktionen eingearbeitet wurde, daß diese Einarbeitung aber in einer so restriktiven Form erfolgte, daß dadurch eher technische Probleme erzeugt denn gelöst wurden.

Die diesbezügliche Unzufriedenheit mit meiner eigenen Arbeit war der ausschlaggebende Grund für die Entwicklung von HOLCF, der *Higher-Order Logic of Computable Functions*. Die ursprüngliche Motivation hinter meiner Arbeit, die anfänglichen Ziele und was sich während der Arbeit an dieser Dissertation daraus entwickelte, möchte ich nun in den nächsten Abschnitten genauer darstellen.

1.3 Ursprüngliche Motivation

Wie ich im letzten Abschnitt dargelegt habe, ist die simultane Bereitstellung des Bereichs der stetigen Funktionen und des vollen Funktionsraums über Bereichen in SPECTRUM nicht sehr gut geglückt. Diesem Mangel wollte ich Abhilfe schaffen und eine Logik konzipieren, in der auch der volle Funktionenraum mit dem zugehörigen Typkonstruktor $\Rightarrow$ uneingeschränkt zur Verfügung steht. Das heißt, der Typkonstruktor $\Rightarrow$ sollte beliebig mit anderen Typkonstruktoren kombiniert werden können, und es sollte auch eine λ-Abstraktion für volle Funktionen geben.

Mehrere Gründe sind dafür verantwortlich, daß ich zum einen von der dreiwertigen SPECTRUM-Logik abgewichen und zur zweiwertigen LCF-Logik zurückgekehrt bin und zum anderen die Logik höherer Stufe HOL als Ausgangspunkt für meine Entwicklung gewählt habe. Die folgenden Abschnitte werden zeigen, daß die Paarung von LCF und HOL naheliegend war. Auf die Historie der Logik HOL werde ich im Kapitel 2 noch gesondert eingehen. In Bezug auf LCF möchte ich auf Larry Paulson verweisen, der in [Pau87] die Historie von LCF ausführlich beschreibt.

Das Konzept der SPECTRUM-Logik, Prädikate über einem Bereich α als volle Funktionen vom Typ $\alpha \Rightarrow \mathit{bool}$ zu kodieren, empfand ich als sehr nützlich und wollte einen Typ der Wahrheitswerte in meiner Logik beibehalten. Zudem sollte der Konstruktor $\Rightarrow$ für den vollen Funktionenraum uneingeschränkt zur Verfügung stehen. Damit war aber die Entscheidung für eine volle Stufenlogik bereits gefallen. Die Wahl einer zweiwertigen Logik höherer Stufe lag ebenfalls nahe, denn eine solche gab es ja bereits, nämlich HOL. Die Logik HOL ist eine bekannte und reichhaltig dokumentierte Logik, für die es neben einem speziellen System von Gordon [Gor85, Cam89, GM93] auch eine Implementierung im generischen Theorembeweiser Isabelle [Pau94] gibt. Da ich die SPECTRUM-Logik ebenfalls in Isabelle kodiert hatte und Isabelle dabei schätzen gelernt hatte, war es naheliegend, die Isabelle-Version von HOL zu benutzen.

Mit der Entscheidung für die zweiwertige Logik HOL war aber gleichzeitig auch die Entscheidung für die zweiwertige Logik LCF gefallen, denn wenn man die dreiwertige Logik von

SPECTRUM durch eine zweiwertige ersetzt, besteht der Unterschied zu LCF im wesentlichen nur noch in dem zusätzlichen Konzept der nichtstetigen Funktionen. Dieses Konzept wollte ich aber gerade geschickter einbauen. Ein weiterer Vorteil von LCF gegenüber der Logik von SPECTRUM ist die Fülle von wissenschaftlichen Untersuchungen, die im Umfeld von LCF gemacht wurden. Die Semantik von LCF wird vollständig von den Standardergebnissen der Bereichstheorie abgedeckt, die ich unverändert übernehmen konnte. Somit konnte ich mich voll auf die Aufgabe konzentrieren, den vollen Funktionenbereich in LCF einzubauen, bzw. richtiger gesagt, die Logik HOL mit dem von Haus aus eingebauten vollen Funktionenraum um Bereichstheorie für stetige Funktionen und damit LCF zu erweitern.

1.4 Erweiterte Zielsetzung der Arbeit

Da HOL die Basis für meine Erweiterung sein sollte, war es naheliegend, eine elegantere Lösung für zwei weitere Probleme ins Auge zu fassen. Das erste Problem betrifft die stetige λ-Abstraktion. Mit stetiger λ-Abstraktion ist der Bindungsmechanismus gemeint, der einen Term von Typ $\alpha \to \beta$ erzeugt. In HOLCF wird dieser Binder mit Λ bezeichnet, um ihn von der λ-Abstraktion für volle Funktionen zu unterscheiden. Von nun an werde ich die HOLCF-Notation verwenden und von Λ-Abstraktion anstatt von stetiger λ-Abstraktion reden.

Wie bereits erwähnt, wird die Bildung von Λ-Termen in SPECTRUM durch eine Kontextbedingung, den sogenannten †-Test, eingeschränkt. Dieser Test prüft *hinreichende* Kriterien für die Stetigkeit eines Terms e vom Typ β in der freien Variablen x vom Typ α, so daß die anschließende Typzuordnung von $\alpha \to \beta$ für den Term $\Lambda x.e$ nicht zu Inkonsistenzen führt. Hinter dem †-Test stecken natürlich die aus der Bereichstheorie bekannten hinreichenden Kriterien, daß die Identität und konstante Funktionen stetig sind, und daß die Applikation stetiger Funktionsterme auf stetige Argumentterme sowie die Λ-Abstraktion über stetigen Termen die Stetigkeit erhalten.

Der entscheidende Punkt beim †-Test ist die Tatsache, daß die Bildung des Terms $\Lambda x.e$ von vornherein nicht erlaubt ist, wenn der Test fehlschlägt. Diese Vorgehensweise mag für rein theoretische Untersuchungen angebracht erscheinen. Will man jedoch die Logik in einem Logical-Framework wie Isabelle formalisieren, so birgt die Verwendung einer solch komplexen Kontextbedingung für die Termbildung einen gravierenden Nachteil: sie läßt sich einfach nicht als Kontextbedingung formalisieren. Kontextbedingungen müssen in Isabelle entweder durch das Typsystem der Metalogik abgedeckt werden oder aber, wenn das Typsystem dafür zu schwach ist, zumindest in der Metalogik von Isabelle als Prädikate formalisiert werden[8]. Wird die Kontextbedingung durch ein Prädikat formalisiert, dann ist die *Bildung* des Terms nicht mehr eingeschränkt, vielmehr wird seine *Verwendung* durch das Prädikat derart beschränkt, daß keine Inkonsistenzen auftreten können. Im Beispiel würde das bedeuten, daß der Term $\Lambda x.e$ immer gebildet werden darf, daß aber etwa die Anwendung der β-Gleichheit $(\Lambda x.e)t = e[t/x]$ nur dann erlaubt ist, wenn die Stetigkeit des Terms e in der Variablen x gezeigt werden kann. Das wiederum heißt, daß man das Prädikat e *ist stetig in* x im Kalkül beweisen muß.

Natürlich können die obigen hinreichenden Kriterien zur Propagierung der Stetigkeit in der Metalogik von Isabelle formalisiert werden. Da in meinem Fall aber HOL die Objektlo-

[8] diese Einschränkung gilt übrigens für alle mir bekannten Logical-Frameworks.

gik ist, bietet es sich an, die Stetigkeit bereits in die Objektlogik zu internalisieren. Eine Axiomatisierung für die Propagierung der Stetigkeit könnte etwa so aussehen.

$contX(\lambda x.c)$	Konstante
$contX(\lambda x.x)$	Identität
$contX(\lambda x.ft(x)) \wedge contX(\lambda x.tt(x)) \rightarrow$ $contX(\lambda x.(ft(x))[tt(x)])$	_[_] Applikation
$\forall x.contX(\lambda y.h(x)(y)) \wedge \forall y.contX(\lambda x.h(x)(y)) \rightarrow$ $contX(\lambda x.\Lambda y.h(x)(y))$	Λ-Abstraktion

Hierbei wird die λ-Abstraktion für volle Funktionen mit einem kleinen λ bezeichnet, und die Applikation voller Funktionen wird mit runden Klammern _(_) geschrieben. Runde Klammern dienen auch dazu, den Vorrang in Termen auszudrücken. Die Abstraktion zur Bildung von Termen des Typs $\alpha \rightarrow \beta$ wird durch ein großes Λ notiert. Im Gegensatz zu SPECTRUM werde ich in HOLCF die Elemente des Typs $\alpha \rightarrow \beta$ als *Operationen* bezeichen. *Funktionen* hingegen sind Elemente des Typs $\alpha \Rightarrow \beta$, und stetige Funktionen bilden eine Teilmenge dieses Typs. Operationen und stetige Funktionen stehen in einem sehr engen Zusammenhang, der in der Theorie der Operationen in den Abschnitten 4.7.1 bis 4.7.6 exakt formalisiert wird. Hinter den eckigen Klammern _[_] verbirgt sich die Applikationsfunktion für Elemente des Typs $\alpha \rightarrow \beta$, den Operationen. Die Applikationsfunktion selbst hat den Typ $(\alpha \rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$ und ist stetig im *ersten* und *zweiten* Argument.

Die Funktion *contX* vom Typ $(\alpha \Rightarrow \beta) \Rightarrow bool$ ist ein Prädikat für volle Funktionen, der Typ *bool* ist in HOL der *zweielementige* Typ der Propositionen[9]. Das Interessante an der obigen Axiomatisierung ist der Umstand, daß statt des Prädikats *Term e ist stetig in der freien Variablen x* das Prädikat *die Funktion $\lambda x.e$ ist stetig* formalisiert wird. Die λ-Abstraktion für Funktionen ermöglicht also das Reden über freie Vorkommen von Variablen in Termen. Dieser Effekt wird in allen Logical-Frameworks ausgenützt.

Die Applikation für Funktionen ersetzt umgekehrt die Notation für Substitution. Dies wird deutlich in der Axiomatisierung der oben angesprochenen Beschränkung der β-Gleichheit für die Λ-Abstraktion.

$$contX(f) \rightarrow (\Lambda x.f(x))[y] = f(y)$$

Die eben axiomatisierten hinreichenden Kriterien sind allerdings sehr schwach[10], da durch sie der Typ der Operationen $\alpha \rightarrow \beta$ relativ isoliert vom vollen Funktionsraum $\alpha \Rightarrow \beta$ behandelt wird. Dem inhärenten Zusammenhang zwischen Operationen und vollem Funktionsraum über Bereichen, speziell dem Zusammenhang zwischen Λ-Abstraktion für Operationen und der λ-Abstraktion für volle Funktionen, werden diese hinreichenden Kriterien in keiner Weise gerecht.

[9] hier wäre die Verwendung der Bezeichnung *prop* schöner. Im HOL-System wie auch in der Isabelle-Version von HOL wird aber *bool* verwendet. Somit muß für den dreielementigen Typ der *programmiersprachlichen Wahrheitswerte* eine andere Bezeichnung verwendet werden. Diese lautet in HOLCF nach der LCF-Konvention *tr* (truth values).

[10] schwach in dem Sinn, daß viele stetige Funktionen diese Kriterien nicht erfüllen.

Bei Verwendung von HOL läßt sich hier Abhilfe schaffen, denn in HOL kann die Stetigkeit nicht nur axiomatisiert werden, sie kann sogar *definiert* werden. Die Ausführungen über die Theorie der stetigen Funktionen in Abschnitt 4.6 und die Theorie der Operationen in den Abschnitten 4.7.1 bis 4.7.6 werden zeigen, daß dadurch der Zusammenhang zwischen Operationen, stetigen Funktionen und dem vollen Funktionsraum befriedigend hergestellt wird. Die oben angeführten Axiome für die Propagierung der Stetigkeit sowie die Beschränkung der β-Gleichheit können dann als Theoreme abgeleitet werden.

Der Grund, wieso ich auf die Behandlung der Stetigkeit so ausführlich eingehe, liegt darin, daß die Stetigkeit von Termen eine subtile Rolle in der Nebenbedingung einer prominenten Inferenzregel der LCF-Logik spielt. Die Rede ist von der Fixpunktinduktionsregel und der Prüfung der Zulässigkeit (admissibility) des Induktionsprädikats als Anwendbarkeitsbedingung. Das bringt uns zum oben angekündigten zweiten Problem, das sich bei Verwendung von HOL eleganter lösen läßt.

Die Regel für die Fixpunktinduktion in LCF läßt sich unter Verwendung der HOLCF-Notation etwa so formulieren:

$$\frac{P(\bot) \quad \forall x.P(x) \to P(f[x])}{P(\mathit{fix}[f])} \qquad \text{provided } P(y) \text{ is admissible in } y$$

In LCF wird die Prüfung der Zulässigkeit als Anwendbarkeitsbedingung der Inferenzregel gehandhabt. Dies bedeutet, daß die Prüfung automatisch vom System durchgeführt wird, indem ein Standardsatz von *hinreichenden* syntaktischen Bedingungen für die Zulässigkeit herangezogen wird. Genügt die Formel P diesen Kriterien nicht, so läßt das System die Anwendung der Fixpunktregel nicht zu. In diesem Fall muß man versuchen, eine äquivalente Induktionsformel zu finden, die die Kriterien erfüllt und dann mit dieser die Induktion durchführen.

Diese Situation ist schon vom rein pragmatischen Standpunkt her unbefriedigend, da zum einen der Satz der Kriterien in der Implementierung fixiert ist, und eine Erweiterung somit sehr aufwendig wird, zum anderen der Induktionsbeweis mit einer äquivalenten Formel bisweilen unnatürlicher sein kann als der Beweis mit der ursprünglichen Formel.

Diese Einschränkung des LCF-Systems wird von Larry Paulson in seinem Buch über Cambridge LCF [Pau87, Seite 70] wie folgt kritisiert:

> Fixed point induction is difficult to formalize. An inference rule operates on the syntactic structure of its premises, but chain-completeness refers to semantics. No simple test is known for determining that a formula is chain-complete. $PP\lambda$'s complicated test rejects many chain-complete formulae[11].

Im Artikel [Pau84] über Induktion in LCF gibt Paulson ein Beispiel (Seiten 207–209) für verschränkt rekursive Typen (Ausdrücke und Listen von Ausdrücken). Zur Axiomatisierung dieser Typen verwendet er Formeln mit lokalen Deklarationen unter Zuhilfenahme der Implikation, um die Formeln lesbarer zu machen. Diese Formeln sind zwar zulässig, werden aber

[11] $PP\lambda$ (*Polymorphic Predicate λ-calculus*) ist die ursprüngliche Bezeichnung von Dana Scott für LCF. Näheres dazu siehe [Pau87].

vom Zulässigkeitstest aufgrund der auftretenden Implikationen abgewiesen. Die äquivalente Formulierung über Selektoren, die den Test besteht, ist laut Paulson wesentlich schlechter lesbar und in ihrer Handhabung unnatürlich.

Formalisiert man LCF in einem Logical-Framework wie Isabelle[12], so lassen sich zumindest die Nachteile einer festen Implementierung der Zulässigkeitsprüfung als Anwendbarkeitsbedingung beheben. Hier wird nämlich die Zulässigkeit analog zur Prüfung der Stetigkeit als Prädikat axiomatisiert. Diese Axiomatisierung kann viel einfacher erweitert werden. Ich gebe hier gleich ausschnittsweise ein Beispiel für die Axiomatisierung der Zulässigkeit, wenn die LCF-Logik um den vollen Funktionenraum erweitert wird und somit das Problem der nichtstetigen Terme im Zusammenhang mit der Zulässigkeit von Formeln entsteht. Hier taucht das Prädikat *contX* an einer im ersten Moment unerwarteten Stelle wieder auf[13].

$$contX(u) \wedge contX(v) \rightarrow adm(\lambda x.u(x) \sqsubseteq v(x))$$
$$contX(t) \rightarrow adm(\lambda x.\neg(t(x) \sqsubseteq u))$$
$$contX(t) \wedge adm(P) \rightarrow adm(\lambda x.P(t(x)))$$

$$adm(P) \wedge adm(Q) \rightarrow adm(\lambda x.P(x) \wedge Q(x))$$
$$adm(P) \wedge adm(Q) \rightarrow adm(\lambda x.P(x) \vee Q(x))$$

$$(\forall x.P(x) \leftrightarrow Q(x)) \rightarrow (adm(P) \leftrightarrow adm(Q))$$

In der Fixpunktinduktionsregel wird aus der Anwendbarkeitsbedingung dann eine Prämisse, und der Nachweis der Zulässigkeit $adm(P)$ muß bzw. kann jetzt im Kalkül geführt werden.

$$\frac{adm(P) \quad P(\bot) \quad \forall x.P(x) \rightarrow P(f[x])}{P(\mathit{fix}[f])}$$

Ein weiterer Vorteil dieser Internalisierung des Zulässigkeitstests wird durch eines der obigen Axiome deutlich:

$$(\forall x.P(x) \leftrightarrow Q(x)) \rightarrow (adm(P) \leftrightarrow adm(Q))$$

Kann man die Äquivalenz der Formeln P und Q zeigen, so darf man die Induktion mit P durchführen, wenn zumindest Q mit dem Satz von hinreichenden Kriterien als zulässig nachgewiesen werden kann. Das setzt natürlich voraus, daß man erstens ein solches Q findet und zweitens den Äquivalenzbeweis im Kalkül führen kann. Muß die Äquivalenz etwa durch eine neuerliche Induktion gezeigt werden, so steht das Gelingen von vornherein erheblich in Frage, da hier dann die Zulässigkeit einer Äquivalenzformel gezeigt werden muß. Äquivalenzen, d.h. Biimplikationen, sträuben sich aber bekannterweise sehr gegen den Zulässigkeitstest.

Ich werde im Kapitel 6 über Datentypen Beispiele für in HOLCF als zulässig nachweisbare Formeln P geben, für die es fraglich erscheint, ob man äquivalente, mit den üblichen LCF-Kriterien beweisbar zulässige, Formeln Q findet, so daß zudem der Äquivalenzbeweis in LCF gelingt.

[12] eine Formalisierung von LCF in Isabelle liegt bereits vor. Sie wird aber leider nicht in [Pau94] beschrieben.
[13] man beachte die Rolle der λ-Abstraktion.

Die obigen Ausführungen legen es nahe, sowohl die Stetigkeit von Funktionen als auch die Zulässigkeit von Formeln zu *definieren*, anstatt sie nur zu axiomatisieren. Die Definition eines Konzepts hat gegenüber einer Axiomatisierung immer den Vorteil, daß sie das Konzept sowohl notwendig als auch hinreichend beschreibt. Da die Zulässigkeit eines Prädikats eine Eigenschaft eines Prädikats ist, also ein Prädikat über Prädikaten, ist klar, daß zur Definition der Zulässigkeit die Logik erster Stufe nicht mehr ausreicht. Auch in diesem Fall konnte die Logik höherer Stufe HOL nutzbringend eingesetzt werden. Die Ausführungen über die Fixpunkttheorie im Abschnitt 4.12 werden dies deutlich machen. Neben der Zulässigkeit wird in dieser Theorie auch der Fixpunktoperator *fix* definiert. Unter anderem werden in dieser Theorie die Fixpunktinduktion und alle oben als Axiome angeführten Eigenschaften der Zulässigkeit als Theoreme abgeleitet.

An dieser Stelle will ich nun aufhören, die Entwicklung der Ziele bis ins Detail zu verfolgen. Eine konsequente Weiterführung der gerade angedeuteten Gedankengänge führte schlußendlich zu einer Zielsetzung, die die Entwicklung der kompletten, für LCF notwendigen, Bereichstheorie in HOL vorsah, eben HOLCF, die *Higher-Order Logic of Computable Functions*. HOLCF ist dabei nur eine komplexe Ansammlung von HOL-Theorien.

Der Vorteil der *Definition* von Konzepten wurde ja schon kurz angedeutet. Ein weiterer Grund, der für diese Vorgehensweise spricht, liegt darin, daß definitorische Theorieerweiterungen die Existenz von Modellen garantieren, wenn die ursprüngliche Theorie ein Modell hatte. Eine Methode, die die Verwendung von definitorischen Erweiterungen als zentrales Vehikel zur Theoriebildung vorsieht, ist unter der Bezeichnung *Sichere Erweiterung von Theorien (safe extension of theories)* bekannt. Sie ist die für die Benutzung des HOL-Systems favorisierte Methode und wird zum Beispiel in [Cam89, GM93] beschrieben. Bei dieser Methode sind nur solche Theorieerweiterungen zugelassen, für die man eine Modellkonstruktion angeben kann, die jedes beliebige Modell der alten Theorie zu einem Modell für die neue Theorie erweitert, so daß die Restriktion des erweiterten Modells bzgl. der Signatur der alten Theorie gerade wieder das ursprüngliche Modell ergibt. Eine Theorieerweiterung, die diesen Eigenschaften genügt, werde ich in der hier vorliegenden Arbeit als *konservative Theorieerweiterung* [Gal86] bezeichnen und die damit einhergehende Modellerweiterung als *konservative Modellerweiterung*. In der Literatur [EGL89] findet sich statt *konservative Theorieerweiterung* auch der Begriff *streng persistente Erweiterung*. Im Falle einer konservativen Theorieerweiterung läßt sich ableiten, daß eine Formel über der Signatur der alten Theorie genau dann in der neuen Theorie *gültig* ist, wenn sie auch in der alten Theorie *gültig* ist.

Der Begriff der konservativen Erweiterung muß mit einiger Vorsicht behandelt werden. Manche Autoren [And86] definieren die Konservativität über ein *syntaktisches* Kriterium. Bei ihnen ist eine Erweiterung dann und nur dann konservativ, wenn eine Formel über der alten Signatur in der neuen Theorie *herleitbar* ist, wenn sie auch schon in der alten Theorie *herleitbar* ist. Dieses Kriterium gewährleistet 'nur' die Erhaltung der Konsistenz (nicht jede Formel ist herleitbar). In einer Logik erster Stufe reicht die Erhaltung der Konsistenz aus, denn jede konsistente Menge ist in der Logik erster Stufe auch erfüllbar. Diese Tatsache ist gerade das zentrale Argument für den Vollständigkeitsbeweis der Logik erster Stufe [Her72]. In einer Logik höherer Stufe kann aus der Konsistenz einer Menge leider nicht mehr auf deren Erfüllbarkeit geschlossen werden, denn sonst wäre die Logik ja vollständig. Ein Erweiterungsmechanismus, der 'nur' die Erhaltung der Konsistenz gewährleistet, ist daher für die Logik höherer Stufe ungenügend, da er nicht mehr die Existenz von Modellen für die erweiterte

Theorie garantiert.

Aus diesem Grund wird bei der Rechtfertigung der Mechanismen zur Theorieerweiterung in HOL [GM93] eine semantische Argumentation verwendet, und aus dem selben Grund heißt in dieser Arbeit eine Erweiterung der Theorie Th_1 zur Theorie Th_2 konservativ, wenn jedes Modell von Th_1 streng persistent zu einem Modell von Th_2 erweitert werden kann. Die formale Definition der konservativen Erweiterung findet sich in Abschnitt 2.6.

Um die HOL-Methode zu wahren und damit gleichzeitig auch die Existenz von Modellen für die Logik HOLCF zu sichern, habe ich HOLCF ausschließlich unter Benutzung solcher konservativer Theorieerweiterungen entwickelt. Dabei wollte ich aber auf keinen Fall die Erfüllung einer anderen wichtigen Zielsetzung gefährden. Die entstehende Logik HOLCF sollte auch benutzbar sein, d.h. nicht nur die prinzipielle Machbarkeit der Formalisierung von Bereichstheorie in HOL in Form einer unzugänglichen Kodierung aufzeigen. Wenn möglich sollte HOLCF die übliche Formalisierung von LCF als Teillogik enthalten. *Look and feel* von LCF sollten sozusagen erhalten bleiben.

Dieser Wunsch wird verständlich im Hinblick auf ein weiteres von mir angestrebtes Ziel. Die Entwicklung von HOLCF sollte vollständig im Logical-Framework Isabelle durchgeführt werden. Der Vorteil dieser Vorgehensweise liegt auf der Hand. Mit Fertigstellung der Entwicklung steht dann nämlich sofort ein Theorembeweiser mit durchaus ansprechenden Automatisierungsmöglicheiten zur Verfügung. Dieser wiederum kann dann als Basis für eine Spezifikationssprache dienen, etwa die Fortentwicklung von SPECTRUM, die HOLCF als logischen Hintergrund hat. Dieser Gedanke wird derzeit in der SPECTRUM-Gruppe in Erwägung gezogen.

Bereits die ersten Versuche, Bereichstheorie in HOL zu formalisieren und dabei ein vernünftiges Maß an Benutzbarkeit zu gewährleisten, zeigten, daß die Wahl von Isabelle als Logical-Framework sehr glücklich war. Im Gegensatz zur Logik höherer Stufe im HOL-System [GM93] ist die HOL-Version in Isabelle mit einem stärkeren Typsystem ausgestattet. Die Metalogik von Isabelle stellt neben dem üblichen Hindley/Milner Polymorphismus [Mil78, DM82] auch das Konzept der Typklassen zur Verfügung [NP93], das vor allem durch die Programmiersprache HASKELL [HJW92] bekannt geworden ist. Dieses stärkere Typsystem schlägt ebenfalls auf die in Isabelle formalisierten Objektlogiken durch. Mit Hilfe der Typklassen war es mir möglich, umständliche Kodierungen zu vermeiden und gewisse Anteile der Logik von Bereichen vom Typsystem abdecken zu lassen. Der filigrane Zusammenhang zwischen internalisierten Konzepten und dem Typsystem spielt in allen Typtheorien eine wichtige Rolle. Die Entscheidung, was explizit in der Logik formalisiert wird und was über das Typsystem abgehandelt wird, bestimmt zu einem wesentlichen Anteil die Eleganz der entstehenden Theorie. Den von mir eingeschlagenen Weg werde ich in aller Ausführlichkeit im Kapitel 4 beschreiben.

Durch die Verwendung von Isabelle konnte ich also eine Version der Logik höherer Stufe benutzen, die Typklassen zur Verfügung stellt. Leider stellte sich dabei heraus, daß HOL mit Typklassen zwar in Isabelle implementiert war und auch schon vielfach verwendet wurde, daß aber keinerlei semantische Untersuchungen zur Logik höherer Stufe mit Typklassen existierten. Damit ergab sich leider für mich ein neuerliches Ziel, nämlich die Erweiterung der Semantik von HOL um das Konzept der Typklassen. Die syntaktischen Schwierigkeiten waren glücklicherweise durch [Nip91, NP93] bereits gelöst, und in Form des Isabelle-Systems lag sogar eine Implementierung vor. Daher konnte ich mich im wesentlichen darauf konzentrieren, die Semantik von HOL und die Methode der konservativen Erweiterung in Bezug auf

Typklassen zu erweitern. Ich habe mich dieser Aufgabe im Kapitel 2 so knapp wie möglich entledigt, denn die damit verbundenen Fragestellungen lagen nicht im Zentrum der von mir geplanten Untersuchungen.

Wenn man die von mir gerade entwickelten Zielsetzungen verfolgt und die damit verbundene Arbeit bedenkt, so stellt sich zu recht die Frage, ob sich denn der ganze Aufwand lohnt. Diese Frage kann ich mit gutem Gewissen mit 'Ja' beantworten.

Eine erste Begründung für dieses 'Ja' ist zum Beispiel der Hinweis auf die Verwandtschaft mit dem AUTOMATH Projekt [dB73]. Meine persönliche Erfahrung mit Lehrbüchern über Bereichstheorie, die ich während der Entwicklung von HOLCF konsultiert habe, hat gezeigt, daß die Beweise einiger Theoreme zwar nicht unbedingt falsch waren, jedoch zumindest unverständlich dargestellt waren. Zudem waren es nicht selten die üblicherweise als Übungsaufgaben gestellten *Hilfssätze*, die die wirklich harten Nüsse bei der Entwicklung von HOLCF darstellten.

Weiterhin deckt HOLCF den Stoff jeder Grundvorlesung über Bereichstheorie ab. Das HOLCF-System läßt sich daher vorlesungsbegleitend zur Lösung von Übungsaufgaben einsetzen oder kann für Praktika verwendet werden.

HOLCF kann, wie bereits erwähnt, im Zusammenhang mit einer Spezifikationssprache eingesetzt werden und zu Verifikationszwecken herangezogen werden. Speziell könnte zum Beispiel für die im FOCUS-Projekt [BDD+93] eingesetzten Methoden und Techniken eine maschinell unterstützte formale Basis geschaffen werden.

Die Ergebnisse im Kapitel 6 zeigen aber auch, daß sich durch die Entwicklung von HOLCF zusätzlich neue Impulse für bereits bekannte Theorien ergeben. So kann aus der bekannten LCF-Axiomatisierung für baumartige rekursive Bereiche [Mon85, Pau87], beispielsweise strikte Listen, Ströme oder nichtstrikte Listen, aufgrund der größeren Ausdrucksstärke von HOLCF wesentlich mehr abgeleitet werden.

Für all diese baumartigen Typen läßt sich durch einfache Argumentation über kleinste obere Schranken und durch Expansion der Definition des Fixpunktoperators ein sogenanntes *Take-Lemma* ableiten[14]. Mit Hilfe dieses datentypspezifischen Theorems läßt sich aus der Gleichheit aller endlichen Approximationen zweier Elemente auf die Gleichheit der Elemente selbst, der Limiten der Approximationen schließen. Dieses Theorem müßte in LCF per Induktion abgeleitet werden, was aufgrund des eingeschränkten Zulässigkeitstests nicht möglich ist. Das Take-Lemma für einen Datentyp ist aber die wesentliche Grundlage für die Ableitung eines Co-Induktionsprinzips [Pit92] innerhalb der Logik.

Zudem ermöglicht das Take-Lemma für Datentypen mit ausschließlich strikten Konstruktoren, etwa strikte Listen, die Ableitung einer bzgl. Zulässigkeit *uneingeschränkten* Regel zur strukturellen Induktion. Wer die LCF-Literatur kennt, weiß, welche außerordentlichen Anstrengungen darauf verwendet werden, die Kettenendlichkeit (chain-finiteness) eines rekursiven Typs bzw. deren Erhaltung zu zeigen. So sind zum Beispiel strikte Listen über dem Typ α nur dann kettenendlich, wenn der Typ α selbst ebenfalls kettenendlich ist. Nur in diesem Fall entfällt dann aufgrund eines bzgl. LCF *externen* Arguments die Anwendbarkeitsbedingung der Zulässigkeit bei der strukturellen Induktion. Im Kapitel 6 werde ich

[14] mehr dazu in Kapitel 6

demonstrieren, wie sich mit Hilfe von HOLCF die strukturelle Induktion für Typen mit strikten Konstruktoren ohne die lästige Zulässigkeitsbedingung ableiten läßt. Zentral beim Beweis ist die Idee, die strukturelle Induktion nicht, wie in LCF üblich, aus der Fixpunktinduktion abzuleiten und damit die Zulässigkeitsbedingung in Kauf nehmen zu müssen, sondern das Take-Lemma als entscheidendes Argument zu verwenden. Dabei ist die Ordnungsstruktur der Argumenttypen unerheblich. Die Vorteile einer uneingeschränkten Induktionsregel und damit auch von HOLCF liegen somit auf der Hand!

1.5 Zusammenfassung der Ziele und Thesen der Arbeit

In diesem Abschnitt möchte ich noch einmal konzentriert die eben entwickelten Ziele zusammenfassen und die Thesen dieser Dissertationsschrift formulieren.

Ziele:

1. Auf der Basis von HOL soll die komplette, für LCF notwendige Bereichstheorie in Form der Logik HOLCF entwickelt werden.
2. Die Entwicklung von HOLCF hat nach der Methode der konservativen Theorieerweiterung zu erfolgen.
3. Die Entwicklung soll unter Verwendung des Logical-Frameworks Isabelle durchgeführt werden. Das von Isabelle zur Verfügung gestellte Konzept der Typklassen soll dabei gewinnbringend eingesetzt werden. HOLCF soll die übliche Formalisierung von LCF als Teillogik enthalten.
4. Die entstehende Logik HOLCF soll praktisch nutzbar sein und nicht nur die prinzipielle Machbarkeit der Formalisierung von Bereichstheorie in HOL in Form einer unzugänglichen Kodierung aufzeigen.
5. Es soll eine hinreichende Formalisierung der Semantik von HOL mit Typklassen erarbeitet werden. Die Methode der konservativen Theorieerweiterung soll so erweitert werden, daß die Typklassen darin eingesetzt werden können.

Thesen:

1. Der volle Funktionenraum kann mit dem Raum der Operationen über Bereichen gleichberechtigt in einer Logik kombiniert werden. Der inhärente Zusammenhang zwischen diesen beiden Funktionsräumen kann in HOLCF explizit gemacht werden und kann auch gewinnbringend eingesetzt werden.
2. Die Stetigkeit von Funktionen und die Zulässigkeit von Prädikaten kann in HOLCF notwendig und hinreichend charakterisiert werden. Die diesbezüglichen Nachteile von LCF sind damit beseitigt.
3. Mit Hilfe der Typklassen ist es möglich, umständliche Kodierungen zu vermeiden und gewisse Anteile der Logik von Bereichen vom Typsystem abdecken zu lassen. Das *look and feel* von LCF bleibt erhalten.

4. Durch die Entwicklung von HOLCF in Isabelle steht ein Theorembeweiser mit durchaus ansprechenden Automatisierungsmöglicheiten zur Verfügung. Dieser kann für Verifikationsaufgaben im Zusammenhang mit einer auf HOLCF basierten Spezifikationssprache genutzt werden.

5. Für Sprachen und Methoden, die auf bereichstheoretischen Grundlagen aufbauen, läßt sich mit Hilfe von HOLCF eine maschinell unterstützte formale Basis schaffen.

6. Mit HOLCF steht eine Plattform für weitere Experimente im Rahmen der Bereichstheorie zur Verfügung.

7. Das HOLCF-System kann begleitend zu Vorlesungen über Bereichstheorie oder in Praktika eingesetzt werden.

8. Durch die Entwicklung von HOLCF ergeben sich neue Impulse für bereits bekannte Theorien. Für baumartige Datentypen lassen sich in HOLCF Induktionsprinzipien sowie Co-Induktionsprinzipien entwickeln, die bisher nur durch rein semantische Argumentationen etabliert werden konnten und nicht aus der üblichen Axiomatisierung ableitbar waren.

1.6 Verwandte Ansätze

In dem von mir bearbeiteten Themengebiet der Formalisierung von Bereichstheorie in einer Logik höherer Stufe sind mir drei vergleichbare Ansätze bekannt. Von diesen drei Arbeiten ist aber bisher nur eine abgeschlossen.

Die abgeschlossene Arbeit ist von Kim Dam Petersen [Pet93] und behandelt die Kodierung des $P\omega$-Modells von Scott [Sco76, Sto77, SG90] im HOL-System von Gordon. Sein Zugang zur Bereichstheorie erfolgt über die Konstruktion eines universellen Bereichs (universal domain) [SG90], nämlich des sogenannten *$P\omega$-Modells* oder *Graph-Modells*. Der Bereich $P\omega$ ist so reichhaltig, daß er auch rekursive Bereiche als Teilbereiche enthält, die als reflexive Bereiche (reflexive domains) über die Benutzung von Retraktionen (retracts) gebildet werden.

Ausgehend von der grundlegenden Kodierung einer partiellen Ordnung (D, R) als Paar aus Trägermenge D und Relation $R \subseteq D \times D$ über dieser Trägermenge, wobei die partiellen Ordnungseigenschaften für (D, R) gelten müssen, führt Petersen Definitionen für obere Schranken, kleinste obere Schranken und gerichtet vollständige Mengen ein. Bereiche sind bei Petersen also Paare (D, R) , die die bekannten Eigenschaften für gerichtet vollständige Mengen erfüllen. Aufbauend auf dieser Theorie der Bereiche entwickelt Petersen dann Theorien für Topologien, Scott-Topologie, algebraische CPO's, endliche Mengen, Potenzmengen und letztendlich eine Formalisierung des $P\omega$-Modells. Speziell sind bei Petersen Bereichskonstruktoren als Funktionen kodiert, die Bereiche, Paare (D, R), in Bereiche abbilden. Dies erlaubt die explizite Manipulation der Trägermenge D und der Ordnung R der Bereiche und gibt Zugang zu allen Konzepten der Bereichstheorie.

Die Behandlung von Mengen in HOL als Teilmengen eines repräsentierenden Typs (eigentlich Prädikate über diesem Typ) erzeugt unangenehmen notationellen Aufwand bei der Bildung neuer Bereiche. Speziell ergeben sich technische Schwierigkeiten bei der Bildung von Funktionsbereichen über Bereichen. Wenn $D1 : \tau_1 \Rightarrow bool$ bzw. $D2 : \tau_2 \Rightarrow bool$ Teilmengen der

Typen τ_1 bzw. τ_2 darstellen, so muß die Menge aller Funktionen mit Argumentbereich $D1$ und Bildbereich $D2$ als Teilmenge aller Funktionen vom Typ $\tau_1 \Rightarrow \tau_2$ kodiert werden. Dabei müssen aber alle Funktionen, die sich auf der Teilmenge $D1$ gleich verhalten, identifiziert werden bzw. ein eindeutiger Vertreter muß ausgewählt werden, um die Extensionalität zu gewährleisten. Durch die Verwendung des Auswahloperators können diese partiellen Funktionen (Funktion auf Teilmengen) zwar durch totale Funktionen auf dem ganzen Typ simuliert werden, allerdings müssen die konkreten Argumentbereiche $D1$ und $D2$ immer explizit mitbehandelt werden.

Der Vorteil von Petersens Methode liegt darin, daß die Konstruktion des $P\omega$-Modells vollständig in HOL unter Einhaltung der Methode der konservativen Theorieerweiterung durchgeführt werden kann. Insbesondere ist damit auch die Konstruktion von Lösungen für rekursive Bereichsgleichungen innerhalb von HOL möglich! Der gravierende Nachteil ist allerdings, daß Bereiche und Bereichskonstruktoren nicht als Typen bzw. Typkonstruktoren in HOL behandelt werden. Somit ist das Typsystem von HOL stark vom Konzept der Bereiche getrennt, was die Handhabung der Theorie meiner Meinung nach umständlich macht. Die Kodierung von Bereichen als Paare (D, R) und von Bereichskonstruktoren als Funktionen, die solche Paare wieder in Paare abbilden, hat zur Folge, daß neben der eingebauten Typdisziplin von HOL eine separate 'Typdisziplin' für Bereiche entsteht. Zum Beispiel muß die Eigenschaft, daß (D, R) eine gerichtet vollständige partielle Ordnung darstellt und bei Anwendung eines Bereichskonstruktors F auf (D, R) wieder eine solche entsteht, immer explizit in Form von Prädikaten ausgedrückt und herumgereicht werden. Es wäre wünschenswerter, dies einmal explizit zu beweisen und dann diese Tatsache in Form einer Typinformation zu verstecken, wie es etwa bei Verwendung von Typklassen möglich ist. Je mehr logische Information in Form eines Typs kodiert werden kann, desto mehr kann auch vom Typinferenzalgorithmus automatisch propagiert werden.

Fairerweise muß hier gesagt werden, daß bei Benutzung von Typklassen in HOLCF ein Teil der Rechtfertigung für die Zuordnung von Typinformation außerhalb des syntaktischen Rahmens der Logik stattfindet und sich auf semantische Argumentation stützt. Wie in Petersens Ansatz werden in HOLCF explizit die Ordnungsrelationen für die Bereiche auf der Termebene definiert und die Ordnungseigenschaften im Kalkül bewiesen. In einem nächsten Schritt wird diese Information aber auf die Typebene hochgehoben und in Form einer Klassenzuordnung für den Typ bzw. den Typkonstruktor ausgedrückt. Die Rechtfertigung für diese Reflektion erfolgt durch eine externe semantische Argumentation, die die konservative Erweiterbarkeit der Modelle zeigt.

Zudem habe ich durch meine Art der Kodierung von Bereichen die Definierbarkeit von Lösungen rekursiver Bereichsgleichungen ausgeschlossen. Der Preis dafür, daß Bereiche auf der Ebene von Typen behandelt werden und somit die Handhabung des Systems verbessert wird, ist die Tatsache, daß die Konstruktion rekursiver Typen ebenfalls auf der Typebene zu erfolgen hätte, etwa durch Verwendung der Inversen-Limes-Konstruktion [KK92, Sch86, Pau87]. Dies erfordert dann aber den Einsatz von abhängigen Typen (dependent types), damit die bei der Colimes-Konstruktion auftretenden Ketten von Bereichen nebst zugehörigen Einbettungen und Projektionen getypt werden können. Dies liegt aber jenseits der Ausdrucksmächtigkeit des Hindley/Milner Polymorphismus mit Typklassen. In HOLCF werden daher rekursive Typen wie in LCF axiomatisiert! Die Rechtfertigung für die diesbezüglich eingesetzten Axiome erfolgt wieder durch modell-theoretische Argumentationen (siehe Kapitel 5), die die Konser-

vativität der Erweiterung zeigen. Bei der Entwicklung von HOLCF habe ich mich bewußt zugunsten einer besseren Benutzbarkeit gegen die Möglichkeit einer vollständigen Definierbarkeit von Lösungen rekursiver Bereichsgleichungen entschieden.

Ein zu Petersens Arbeit relativ ähnlicher Ansatz wird von Sten Agerholm verfolgt. Diese Arbeit entstand parallel zu meiner Entwicklung, und wir hatten anfangs losen Kontakt per Electronic-Mail, um die Zielsetzungen abzugleichen. Teile seiner Arbeit sind in [Age93] beschrieben, der volle Umfang seiner Entwicklung wird in der gerade entstehenden These [Age94] dargestellt werden. Wie Petersen arbeitet Agerholm mit dem originalen HOL-System von Gordon. Auch seine Formalisierung von Bereichen erfolgt über Paare (D, R) aus Trägermenge und Relation und entspricht somit im wesentlichen der von Petersen. Dies bedeutet, daß auch bei Agerholm Bereiche bzw. Bereichskonstruktoren nicht durch Typen bzw. Typkonstruktoren ausgedrückt werden und daß die an Petersens Arbeit geübte Kritik hier ebenfalls greift.

Agerholm arbeitet wie ich, im Gegensatz zu Petersen, mit ω-kettenvollständigen partiellen Ordnungen. Desweiteren formalisiert er auch keine Theorien für Topologie, Scott Topologie, algebraische CPO's und das $P\omega$-Modell wie Petersen. Er versucht nicht, einen universellen Bereich wie $P\omega$ zu konstruieren, in dem dann rekursive Bereichsgleichungen gelöst werden können, sondern reichert Typen, die durch die herkömmliche HOL-Methode definiert werden, nachträglich mit einer partiellen Ordnung an. So wird zum Beispiel der HOL-Typ der natürlichen Zahlen zuerst mit einer diskreten Ordnung versehen und in einem zweiten Schritt per Lifting in einen Bereich mit kleinstem Element verwandelt. Dieser flach geordnete Bereich ist dann natürlich durch die durchgeführten Konstruktionen nicht mehr als Typ repräsentiert, sondern wie bei Petersen als Paar (D, R), wodurch die unbequeme Handhabung entsteht.

Für baumartige rekursive Typen, wie etwa Listen, gibt es in HOL eine Methode der Kodierung, die sich auf endliche, beblätterte Bäume als repräsentierenden Typ abstützt. Diese Art der Kodierung wird zum Beispiel von Paulson [Pau94] in Isabelle-HOL verwendet. Auch für diese Typen kann Agerholm eine Anreicherung zum Bereich durchführen. Wie er mir mitteilte, arbeitet er an einer Kodierungstechnik, die es sogar erlaubt, Typen mit unendlichen Elementen wie Streams oder Lazy-Listen in HOL zu formalisieren und diese Typen dann in Bereiche zu verwandeln. Hier ist aber ein erheblicher Kodierungsaufwand erforderlich. Inwiefern seine Technik zur Formalisierung von Typen mit unendlichen Elementen der von Paulson [Pau94] gleicht, kann ich nicht beurteilen, da er diese Ergebnisse erst in seiner These veröffentlichen wird.

Angesichts der Tatsache, daß die Lösung rekursiver Bereichsgleichungen semantisch wohl verstanden ist, tendiere ich eher dazu, rekursive Bereiche wie in LCF durch wenige, semantisch gerechtfertigte Axiome zu axiomatisieren. Die nach der LCF-Methode benötigten Axiome zur Axiomatisierung eines rekursiven Typs sind meiner Meinung nach übersichtlicher und notationell einfacher zu handhaben als die puristischere Kodierung in HOL, wie sie etwa von Paulson verwendet wird. Zudem kann man zeigen (siehe Kapitel 5), daß auch bei Verwendung der LCF-Methode die Kriterien für eine konservative Theorieerweiterung erfüllt sind, und somit bleibt man im Rahmen der Methode der konservativen Theorieerweiterung von HOL.

Die dritte Arbeit in diesem Themenkreis wird derzeit von Bernhard Reus an der Ludwig-Maximilian Universität München durchgeführt. Die theoretischen Grundlagen seiner Arbeit sind in [RS93] dargestellt, mit der praktischen Implementierung hat Bernhard Reus jedoch erst

vor kurzem begonnen. Er benutzt im Gegensatz zu Petersen, Agerholm und mir nicht HOL als Plattform, sondern eine konstruktive Typtheorie. Der von ihm verfolgte Ansatz fällt in den Bereich der *synthetischen Bereichstheorie (synthetic domain theory)* [Pho90, Hyl91, Tay91], die nur in intuitionistischer Logik konsistent ist. Er verwendet das LEGO-System [LPT89] für seine Kodierung der synthetischen Bereichstheorie. Die reichhaltige Typsprache von Luos ECC (*extended calculus of constructions*) [Luo89, Luo91], welcher die Grundlage des LEGO-Systems ist, erlaubt die Konstruktion von Bereichen als Typen, die zusätzlich zur Trägermenge des Bereichs auch die Ordnungsrelation mitsamt den geforderten Ordnungseigenschaften festlegen. Das Besondere an der synthetischen Bereichstheorie ist, daß Bereiche nicht als Paare von Träger und Ordnungsrelation kodiert werden, sondern die Ordnung schon durch die Konstruktion des Trägers (hier des Typs!) festgelegt ist. Die Ordnung ist dabei über ein Beobachtungsprinzip ähnlich dem der Leibnitz-Gleichheit gegeben. Zudem kann im Gegensatz zu HOL die Inverse-Limes-Konstruktion zur Lösung rekursiver Bereichsgleichungen in LEGO vollständig auf der Typebene formalisiert werden. In diesem Punkt bietet die Verwendung der konstruktiven Typtheorie gegenüber HOL also deutliche Vorteile.

Verglichen mit meinem Ansatz ist die Formalisierung von Bernhard Reus wesentlich konstruktiver, denn er kommt bis auf einige grundlegende Axiome und die Vorgabe eines speziellen imprädikativen Typuniversums, das die Grundlage für die Entwicklung seiner synthetischen Bereichstheorie darstellt, ohne externe semantische Argumentation aus. Der Preis für die Verwendung der konstruktiven Typtheorie liegt allerdings darin, daß die für die Benutzbarkeit des Systems so wertvolle Typinferenz für den ECC nicht mehr möglich ist. Terme müssen in LEGO immer voll getypt werden, und die für die konstruktive Typtheorie charakteristischen Beweise für Typurteile (typing judgements) müssen vom Benutzer jeweils explizit erbracht werden. Desweiteren bietet LEGO im Gegensatz zu HOL bzw. Isabelle keine Unterstützung durch einen Theorembeweiser, und somit müssen alle Beweise bis ins kleinste Detail manuell durchgeführt werden.

1.7 Aufbau der Arbeit

In diesem Abschnitt möchte ich den Aufbau der Arbeit skizzieren. Das Kapitel 2 beschreibt HOLC, eine Variante der Logik höherer Stufe HOL, die das zusätzliche Konzept der Typklassen anbietet. In Abschnitt 2.1 werde ich kurz auf die Historie von HOL eingehen. Abschnitt 2.2 motiviert die Erweiterung von HOL um Typklassen und enthält eine informelle Semantik der Typklassen. Danach folgen die Abschnitte 2.3, 2.4 und 2.5, in denen Syntax, Semantik und das Deduktionssystem von HOLC formalisiert werden. Der letzte Abschnitt 2.6 dieses Kapitels handelt von der Anpassung der Methode der konservativen Erweiterung an die Logik HOLC.

Kapitel 3 beschäftigt sich mit der Formalisierung dieser Version von HOL in Isabelle, wie sie von Tobias Nipkow durchgeführt wurde [Pau94]. Das Kapitel dient im wesentlichen dazu, die Isabelle-Notation einzuführen, da ich diese dann in den späteren Kapiteln verwenden werde.

Kapitel 4 stellt den Hauptteil meiner Arbeit dar. Hier werde ich die Entwicklung der Logik HOLCF mit allen abgeleiteten Theoremen darstellen und die interessanten Beweise im einzelnen durchgehen. Natürlich kann und will ich nicht alle Beweise präsentieren, denn die Darstellung der über 500, in Isabelle maschinell bewiesenen, Theoreme würde bei weitem

den Rahmen dieser Arbeit sprengen und erscheint mir auch nicht sonderlich sinnvoll. Zudem ist die HOLCF-Logik bereits in der aktuellen Isabelle-Distribution enthalten, wodurch die Möglichkeit besteht, die Beweise aller Theoreme mit dem Isabelle-System nachzuvollziehen. Es folgt nun eine Beschreibung der einzelnen Theorien, wie sie in Kapitel 4 dargestellt sind.

Im Abschnitt 4.1 werden zunächst ein Minimierungsoperator für die natürlichen Zahlen eingeführt sowie einige nützliche prädikatenlogische Theoreme bewiesen. Danach entwickle ich in Abschnitt 4.2 eine Theorie für den trivialen Typ *void*, der nur ein Element in der Trägermenge enthält. Auf diesem Typ definiere ich dann die offensichtliche Ordnung und weise die Eigenschaften einer partiellen und später einer kettenvollständigen partiellen Ordnung mit kleinstem Element nach. Der Typ *void* dient als Rechtfertigung für die Typklassen der partiellen Ordnungen *po* und der ω-kettenvollständigen partiellen Ordnungen *pcpo* mit kleinstem Element, die in den darauffolgenden Abschnitten 4.3 und 4.4 eingeführt werden. Im Abschnitt 4.5 wird für Funktionen auf Bereichstypen (Typen in Klasse *pcpo*) eine Ordnungsrelation definiert, und die Ordnungseigenschaften werden nachgewiesen. Dies rechtfertigt dann auch die zusätzliche Zuordnung von Klasseninformation für den Typkonstruktor $\Rightarrow$, die besagt, daß $\Rightarrow$ Bereichstypen wieder in Bereichstypen abbildet. Der Abschnitt 4.6 bringt Definitionen für monotone und stetige Funktionen auf Bereichstypen. Im Abschnitt 4.7 werden diese Ergebnisse über stetige Funktionen ausgenutzt, um den Typkonstruktor $\rightarrow$ der Operationen sowie eine Ordnung auf Operationen zu definieren und die Ordnungseigenschaften dafür abzuleiten. Zudem werden wieder zusätzliche Zuordnungen von Klasseninformation für den Typkonstruktor $\rightarrow$ vorgenommen, die ausdrücken, daß der Typ der Operation über Bereichstypen ebenfalls in der Klasse *pcpo* der Bereiche liegt. Weiterhin werden hier die Theoreme zur Propagierung der Stetigkeit von Termen abgeleitet. Das gleiche Programm wird im Abschnitt 4.8 für das strikte Produkt, in 4.9 für das kartesische Produkt, in 4.10 für die strikte Summe und in 4.11 für den gelifteten Bereich durchgeführt. In Abschnitt 4.12 wird dann die Fixpunkttheorie für LCF formalisiert. Speziell werden hier der Fixpunktoperator auf Operationen *fix* und das Zulässigkeitsprädikat *adm* definiert und deren wesentlichen Eigenschaften abgeleitet. Dazu zählen die Fixpunktinduktion und die Propagierung der Zulässigkeit. In Abschnitt 4.13 werden dann noch eine Komposition für Operationen und eine Identitätsoperation eingeführt, was die spätere Axiomatisierung rekursiver Bereichstypen notationell vereinfacht.

Kapitel 5 beschäftigt sich mit Ergebnissen aus der Kategorientheorie [KK92, Fre90] über die Axiomatisierung initialer Lösungen für rekursive Bereichsgleichungen. Mit Hilfe dieser Ergebnisse wird die Verwendung gewisser Axiome gerechtfertigt, die zur Axiomatisierung rekursiver Bereiche in HOLCF benötigt werden. Hierbei handelt es sich um eine der oben erwähnten externen modelltheoretischen Argumentationen, um die Konservativität einer Theorieerweiterung zu garantieren. Die verwendeten Axiome entsprechen zwar denjenigen, die auch in LCF verwendet werden [Mon85, Pau84, Pau87], die wesentlich komplexere Logik von HOLCF macht jedoch eine neuerliche Rechtfertigung notwendig.

Kapitel 6 beschreibt dann die Anwendung der Ergebnisse aus Kapitel 5 in HOLCF. Zuerst werden in Abschnitt 6.1 der Typ *one* mit nur einem definierten Element und dann in Abschnitt 6.2 der Typ *tr* der programmiersprachlichen Wahrheitswerte mit den zugehörigen Operationen eingeführt. Dann werde ich die Axiomatisierung der Datentypen für natürliche Zahlen 6.4, Ströme 6.5 und strikte Listen 6.6 beschreiben, sowie die Ableitung der bereits erwähnten Induktions– und Co-Induktionsprinzipien vorstellen.

In Kapitel 7 werde ich meine Arbeit mit einem Ausblick auf zukünftige Arbeiten beenden.

Kapitel 2

HOLC: Eine Logik höherer Stufe mit Typklassen

Dieses Kapitel beschreibt HOLC, eine Variante der Logik höherer Stufe HOL, die das zusätzliche Konzept der Typklassen anbietet. In Abschnitt 2.1 werde ich kurz auf die Historie von HOL eingehen. Abschnitt 2.2 motiviert die Erweiterung von HOL um Typklassen und enthält eine informelle Semantik der Typklassen. Danach folgen die Abschnitte 2.3, 2.4 und 2.5, in denen Syntax, Semantik und das Deduktionssystem von HOLC formalisiert werden. Der letzte Abschnitt dieses Kapitels 2.6 handelt von der Anpassung der Methode der konservativen Theorieerweiterung an die Logik HOLC.

2.1 Zur Geschichte von HOL

Prädikatenlogiken werden im allgemeinen dadurch klassifiziert, daß man ihnen eine *Stufe (order)* zuordnet. Am bekanntesten ist die Prädikatenlogik *erster Stufe (first-order logic)*, die nur die Quantifizierung über Individuenvariablen zuläßt. Führt man zusätzlich zu den Individuenvariablen auch Variablen für Prädikate über Individuen ein und erlaubt die Quantifizierung über diese Prädikatsvariablen, so erhält man Prädikatenlogik *zweiter Stufe (second-order logic)*. Die Einführung von Variablen für Prädikate über Prädikaten führt zur Logik *dritter Stufe*. Durch endliche Iteration dieses Prozesses lassen sich Logiken der n-ten Stufe erzeugen. Einer Logik, die genau alle Prädikatenlogiken n-ter Stufe für endliches n als Teilsysteme enthält, ordnet man die Stufe ω zu *(ω-order logic)*. Eine Iteration über ω hinaus führt zu Logiken mit transfiniter Stufe. Stellt die Logik neben Variablen auch Konstanten bereit, so wird die obige Einteilung etwas komplizierter. In einer Logik n-ter Stufe sind Prädikatenkonstanten bis zur Stufe n und Prädikatenvariablen bis zur Stufe $n-1$ nebst Quantifizierung über letztere zugelassen. Eine formale Definition dieser Begriffe findet sich zum Beispiel in [And86] oder [VEB83].

In der Literatur, etwa [Tak75] oder [And86], wird statt des Begriffs *Prädikatenlogik der n-ten Stufe* auch oft der Begriff *Theorie vom Typus n* verwendet. Allgemein spricht man dann auch von *Typentheorie (type theory)* was allerdings nicht mit *konstruktiver Typtheorie (constructive type theory)* verwechselt werden darf. In der englischen Literatur findet man statt

der Bezeichnung *ω-order logic* auch die Synonyme *finite type theory, simple type theory* oder *higher-order logic*. In der deutschen Literatur [VEB83] finden sich *endliche Typentheory, volle Stufenlogik* und *Logik höherer Stufe*.

Eine Version der endlichen Typentheorie, die auf der λ-Abstraktion als einzigem Bindungsoperator aufbaut, wurde zuerst von Alonzo Church unter dem Namen *simple theory of types* in [Chu40] eingeführt, deren Vollständigkeit wurde von Henkin [Hen50] gezeigt. In dieser Logik wird die Gleichheit über Quantoren und primitive Junktoren definiert. Eine andere Variante der Logik höherer Stufe erhält man, wenn man die Gleichheit als grundlegendes Konzept verwendet und die Quantoren und Junktoren dann über die Gleichheit definiert. Diese Variante wurde von Henkin [Hen63] eingeführt und von Andrews [And63] vereinfacht. Eine ausführliche Beschreibung dieser Logik sowie eine umfassende Darstellung der Historie der endlichen Typentheorie, aus der ich auch die obige Zusammenfassung extrahiert habe, findet sich in [And86].

Aufbauend auf diesen theoretischen Untersuchungen wurden in den 80-er Jahren die ersten Implementierungen für die Logik höherer Stufe entwickelt. Neben dem TPS-Theorembeweiser [AML+84] möchte ich hier vor allem das HOL-System von Gordon [Gor85] erwähnen.

Die HOL-Logik unterscheidet sich von Churchs und Andrews Logik dadurch, daß sie zusätzlich Variablen für Typen zuläßt (Hindley/Milner Polymorphismus) und statt des ι-Deskriptors den Hilbertschen ε-Deskriptor verwendet, um das Auswahlaxiom (axiom of choice) zu formalisieren. Wie in [And86] ist auch in Gordons Logik die Gleichheit ein primitives Konzept. Das HOL-System wird ausführlich in den Manualen [Cam89, GM93] beschrieben. Die HOL-Logik wird neben [Gor85] ebenfalls in [Cam89, GM93] dargestellt.

Auf der Grundlage von Gordons HOL wurde 1991 im generischen Theorembeweiser Isabelle von Tobias Nipkow [Pau94] eine Variante von HOL entwickelt, die neben dem Hindley/Milner Polymorphismus das zusätzliche Konzept der Typklassen zur Verfügung stellt. Eine theoretische Untersuchung zu dieser Version von HOL gibt es leider bisher nicht. Diesem Mangel versuche ich im folgenden Abhilfe zu schaffen.

2.2 Eine informelle Semantik für Typklassen in HOLC

In diesem Abschnitt möchte ich eine informelle Erklärung für das Konzept des Hindley/Milner Polymorphismus mit Typklassen in HOLC angeben. Zum einen soll damit die sehr technische Präsentation von Syntax, Semantik, Deduktionssystem und konservativer Theorieerweiterung in den Abschnitten 2.3 bis 2.6 vorbereitet werden, zum anderen möchte ich auch die Möglichkeit bieten, beim ersten Lesen der Arbeit diese technischen Abschnitte auszulassen und gleich mit Kapitel 3 weiterzumachen.

2.2.1 Hindley/Milner Polymorphismus

Gordons HOL-Logik ist, wie bereits erwähnt, eine Logik mit Hindley/Milner Polymorphismus, d.h. es gibt in der Syntax Variablen für Typen (*Typvariablen*) und polymorphe Konstanten, die evtl. einen funktionalen Typ haben. Typvariablen werden üblicherweise mit kleinen griechischen Buchstaben bezeichnet und wurden von mir im Text schon mehrfach

verwendet, etwa in $\alpha \Rightarrow \beta$. Ein Beispiel für eine polymorphe Konstante ist die Konstante $=$ für die Identitätsfunktion vom Typ $\alpha \Rightarrow \alpha \Rightarrow bool$. Das besondere Merkmal des Hindley/Milner Polymorphismus ist das Fehlen eines Bindungsmechanismus für Typvariablen in Typtermen. Somit gibt es in Typtermen nur freie Vorkommen von Typvariablen. Hinsichtlich der syntaktischen Behandlung polymorpher Konstanten gibt es in den Systemen mit Hindley/Milner Polymorphismus kleine Unterschiede. In HOL [GM93] beispielsweise werden die Typen polymorpher Konstanten, etwa bei $=$ der Ausdruck $\alpha \Rightarrow \alpha \Rightarrow bool$, nur als *Typschemata* aufgefaßt, in denen die Typvariablen beliebig durch Typterme[1] ersetzt werden können, um einen echten Typ, eine sogenannte *Instanz* des Typschemas, zu erhalten. Hier gibt es auch auf der Termebene keine Möglichkeit, Typvariablen zu binden. In der formalen Syntax für ML-artige Sprachen [Mit90] hingegen gibt es einen Bindungsmechanismus zur Bindung von Typvariablen in den Typen polymorpher Konstanten und bei der Verwendung des *let* Konstrukts. Der Identität würde etwa der Typ $\forall\alpha.\alpha \Rightarrow \alpha \Rightarrow bool$ zugeordnet. Diese syntaktisch deutlichere Auszeichnung des Schemacharakters erlaubt die Behandlung des sogenannten *ML-Polymorphismus (auch let-Polymorphismus)*, der eine etwas stärkere Art des Polymorphismus darstellt [Mit90]. Für HOL jedoch genügt die einfachere Behandlung der Typen polymorpher Konstanten als Typschemata, da es hier kein *let* Konstrukt gibt.

Die Typen von HOL werden als Mengen in einem Universum U interpretiert, einem sogenannten *type frame* [And86]. Vom Mengenuniversum U werden spezielle Abschlußeigenschaften gefordert, wie etwa die Abgeschlossenheit gegenüber nichtleeren Teilmengen, kartesischem Produkt und Potenzmengenbildung. Damit die Gültigkeit des Auswahlaxioms gesichert ist, darf U nur nichtleere Mengen enthalten und es gibt zusätzlich eine Auswahlfunktion auf U. Die Forderung nach mindestens einer unendlichen Menge in U ist die Basis für das Axiom der Unendlichkeit (axiom of infinity), welches die Grundlage für die spätere Formalisierung der natürlichen Zahlen in HOL ist. Weiterhin werden an die Mengen in U keinerlei strukturelle Anforderungen gestellt, d.h. die Elemente in U sind einfach nur Mengen. Die gerade skizzierten Abschlußeigenschaften werden in [Cam89, Kapitel 10] und [GM93, Kapitel 15] genauer formuliert. Sie entsprechen im wesentlichen den von mir im Abschnitt 2.4 geforderten Eigenschaften.

Die Semantik von Typtermen und von polymorphen Konstanten ergibt sich nun wie folgt: Typkonstruktoren, wie etwa der Konstruktor $\Rightarrow$ für den vollen Funktionenraum, werden als *totale* Abbildungen auf U interpretiert. Für das Beispiel des Konstruktors $\Rightarrow$ ist die Interpretation eine Abbildung von $U \times U$ nach U. Interpretiert man $\Rightarrow$ als den Konstruktor für *alle* totalen Funktionen, so spricht man in diesem Zusammenhang auch von *full type frames* oder *Henkin Standardmodellen* [And86, Mit90]. Typkonstanten, wie etwa *bool*, werden als nullstellige Typkonstruktoren aufgefaßt. Typterme, in denen ja Typvariablen vorkommen dürfen, können interpretiert werden, indem man eine Belegung der Typvariablen mit Mengen aus dem Universum U benutzt. Auf diese Weise setzt man den Begriff des *Environment Modells* [Mit90] konsequent auf Systeme mit polymorphen Termen fort. Man kann aber auch wie in [GM93] Typterme als Funktionen in ihren freien Typvariablen auf U interpretieren und erhält somit eine notationelle Variante.

Die Semantik einer polymorphen Konstanten, der also als Typ ein Typschema zugeordnet wird, etwa $=$ mit Typschema $\alpha \Rightarrow \alpha \Rightarrow bool$, ist durch eine ganze Familie von Interpretationen gegeben. Wird das Schema zum Beispiel mit dem Typterm τ instantiiert, so erhält

[1] die Typterme dürfen ihrerseits wieder Typvariablen enthalten.

man den konkreten wohlgeformten Term $= : \tau \Rightarrow \tau \Rightarrow \mathit{bool}$. Wenn für eine beliebige Belegung von Typvariablen die Semantik des Typterms τ die Menge $M_1 \in U$ ist und die Semantik von $\tau \Rightarrow \tau \Rightarrow \mathit{bool}$ unter derselben Belegung $M_2 \in U$, dann ist die Semantik des Terms $= : \tau \Rightarrow \tau \Rightarrow \mathit{bool}$ unter dieser Belegung gerade dasjenige Element der Familie, welches *mit der Menge* M_1 *indiziert ist* und welches *in der Menge* M_2 *liegt*. Mathematisch läßt sich dieser komplexe Zusammenhang über die Verwendung eines allgemeinen mehrstelligen[2] kartesischen Produkts (*generalized cartesian product,* Π-*product*) ausdrücken. Barendregt [Bar91] spricht hierbei von *Type dependent Objects*, da zur Indizierung Typen verwendet werden und Objekte in Typen das Ergebnis sind. Die Einschränkung auf Type dependent Objects, wobei Typschemata selbst nicht als Typen aufgefaßt werden und somit auch nicht zur Indizierung zugelassen sind (predicative polymorphism), ist ein charakteristisches Merkmal des Hindley/Milner Polymorphismus. In Systemen mit stärkerem Polymorphismus werden auch andere Index- bzw. Zielmengen zugelassen. Dies wird dann natürlich auch syntaktisch durch entsprechende Bindungsmechanismen reflektiert. Eine Einführung in diese Modellvorstellung für den Hindley/Milner Polymorphismus findet sich in [Sok89]. Dort wird auch der Begriff *shallow polymorphism* als Synonym für Hindley/Milner Polymorphismus verwendet.

Ein wichtiger Umstand, der sich auch auf die noch zu beschreibende Semantik von Typklassen auswirkt, ist die Tatsache, daß es im Mengenuniversum U wesentlich mehr Mengen gibt, als über die Interpretation von Typtermen erreicht werden können. U ist somit nicht *termerzeugt* bzgl. Typtermen! Dies sieht man sofort ein, wenn man die geforderten Abschlußeigenschaften für das Universum U betrachtet. Da es zum Beispiel kein syntaktisches Konstrukt zur Bildung von Subtypen gibt, der Abschluß von U aber bezüglich dieser semantischen Konstruktion gefordert wird, kann U nicht termerzeugt sein. Der Grund für diese starken Abschlußforderungen liegt zum Teil in der Erklärung für die Methode der konservativen Erweiterung von Modellen. Hierbei wird aus einem beliebigen Modell für die alte Theorie ein Modell für die neue Theorie konstruiert, so daß die Restriktion des neuen Modells bzgl. der Signatur der alten Theorie gerade das alte Modell ergibt. Bei der Einführung eines neuen Typs bzw. eines Typkonstruktors muß ein Modell konstruiert werden, das eine zusätzliche Interpretation für den neuen Typ(konstruktor) aufbauend auf einem Modell für die alte Theorie enthält[3]. Damit die Abgeschlossenheit bzgl. der bereits im alten Modell interpretierten Typkonstruktoren erhalten bleibt, ist dies im allgemeinen aber nur möglich, wenn schon das Modell für die alte Theorie reichhaltig genug ist, um den neuen Konstruktor zu interpretieren. Bei der konservativen Theorieerweiterung um neue Typen werden stets nichtleere Teilmengen eines schon interpretierten Typs herangezogen, und daher muß das Modell der alten Theorie schon alle potentiellen Teilmengen enthalten. Daraus erklärt sich die Forderung nach der Abgeschlossenheit bzgl. nichtleerer Teilmengen. Ähnliche Überlegungen in Bezug auf konservative Theorieerweiterung führen dazu, daß es bei der Definition für die Semantik polymorpher Funktionen vernünftiger ist, nicht mit einer syntaktischen Indizierung durch Typterme zu arbeiten, sondern die Mengen in U als Indizes zu verwenden.

Nach dieser Skizzierung des Hindley/Milner Polymorphismus werde ich jetzt schrittweise die Idee für eine Erweiterung dieses Konzepts um Typklassen präsentieren. Hierbei werde ich mein konkretes Ziel der Integration von LCF in HOL als Leitfaden benutzen und vorwiegend semantische Argumente für die Verwendung von Typklassen ins Feld führen. Es sei

[2] für jede frei vorkommende Typvariable eine Stelle.

[3] siehe [GM93, Kapitel 16] und Abschnitt 2.6.

jedoch darauf hingewiesen, daß das Konzept der Typklassen unabhängig von diesen semantischen Überlegungen aus rein syntaktischen Gründen von Tobias Nipkow in Isabelle eingebaut wurde [Nip91], lange bevor ich mit der hier vorliegenden Arbeit begonnen habe. Diese hier dargestellte informelle Beschreibung und auch die formale Behandlung in den Abschnitten 2.3 bis 2.6 erfolgte nachträglich und stellt meine persönliche Auffassung vom Konzept der Polymorphie mit Typklassen dar.

Nachdem die Semantik für Typen, Typkonstruktoren, Typvariablen und polymorphe Konstanten in HOL skizziert ist, können wir uns nun den entsprechenden Konzepten in der Logik LCF zuwenden. Die Erklärung der Polymorphie erfolgt hier genauso wie in HOL, nur daß sich die LCF-Semantik von Typen von der entsprechenden HOL-Semantik wesentlich unterscheidet. In LCF werden Typen nicht nur als Mengen interpretiert sondern als Bereiche, d.h. Mengen mit ω-cpo Struktur und einem bzgl. der Ordnung kleinsten Element. Hierfür benötigt man ein Universum BU von Bereichen, welches bzgl. diverser Bereichskonstruktionen abgeschlossen sein muß, etwa der Bildung des strikten Produkts oder des Bereichs der stetigen Funktionen über Bereichen. Die Belegung von Typvariablen erfolgt mit Bereichen aus diesem Universum. Neben anderen polymorphen Konstanten gibt es in LCF die speziellen Konstanten $\bot$ und $\sqsubseteq$. Wenn der Typ τ unter einer bestimmten Belegung den Bereich $B_1 \in BU$ als Interpretation hat, so wird die Instanz von $\bot$ für diesen Typ τ als das kleinste Element in B_1 interpretiert. Entsprechend ist die Interpretation für die τ-Instanz von $\sqsubseteq$ gerade die Ordnungsrelation auf dem Bereich B_1.

2.2.2 Hindley/Milner Polymorphismus mit Typklassen

Will man nun HOL und LCF zusammenbringen und den Polymorphismus beider Logiken beibehalten, d.h. sowohl Typkonstruktoren und Typvariablen für HOL-Mengentypen als auch Typkonstruktoren und Typvariablen für LCF-Bereichstypen anbieten, so muß man syntaktisch und semantisch eine Unterscheidung dieser beiden Welten vorsehen.

Eine syntaktische Trennung erreicht man, indem man zwei Arten (Klassen) von Typen einführt. Die Klasse für die HOL-Mengentypen nenne ich *top*[4] und die Klasse für die LCF-Bereichstypen nenne ich *pcpo*. Typvariablen für Mengentypen (Typen in Klasse *top*) bekommen den Index *top*, etwa α_{top}, und Typvariablen für Bereichstypen bekommen den Index *pcpo*, etwa α_{pcpo}. Bei Typkonstruktoren kennzeichnet man, auf welcher Klasse sie operieren, d.h. man ordnet ihnen eine sogenannte Arität (arity) zu. Die Arität besagt, aus welcher Klasse die Argumente des Konstruktors kommen und in welcher Klasse das Ergebnis der Konstruktion liegt. Zum Beispiel erhält der Konstruktor $\Rightarrow$ für den vollen Funktionsraum die Arität $\Rightarrow :(top, top)top$, der Konstruktor $\rightarrow$ für den Typ der stetigen Funktionen aber dagegen die Arität[5] $\rightarrow :(pcpo, pcpo)pcpo$. Typkonstanten sind der Spezialfall eines Typkonstruktors. So bekommt zum Beispiel der zweielementige[6] HOL-Typ *bool* die Arität $bool{:}top$, wogegen der LCF-Typ der programmiersprachlichen Wahrheitswerte *tr* die Arität $tr{:}pcpo$ erhält. Eine Semantik für die beiden Klassen *top* und *pcpo* erhält man, indem man

[4] die Bezeichnung *top* ist an dieser Stelle noch unmotiviert, wird sich aber später klären.

[5] eigentlich ist $\Rightarrow :(top, top)top$ die *Aritätsvereinbarung*, die dem Typkonstruktor $\Rightarrow$ die Arität $(top, top)top$ zuordnet. Ich werde im folgenden jedoch nicht so genau zwischen Arität und Aritätsvereinbarung unterscheiden.

[6] die Interpretation des Typs ist eine zweielementige Menge.

die Klasse *top* durch das HOL-Mengenuniversum U interpretiert und die Klasse *pcpo* durch das LCF-Universum von Bereichen BU.

Diese gerade skizzierten syntaktischen und semantischen Erweiterungen genügen jedoch noch nicht, um HOL und LCF innerhalb eines logischen Rahmens zu behandeln. Versuchen wir etwa, der polymorphen Konstanten $\sqsubseteq$ ein Typschema zuzuordnen, so kommt dafür nur das Schema $\alpha_{pcpo} \Rightarrow \alpha_{pcpo} \Rightarrow bool$ in Frage. Auf der einen Seite soll die Konstante $\sqsubseteq$ gemäß LCF-Semantik für jede Instanz als die Ordnungsrelation auf dem entsprechenden Bereich interpretiert werden, daher die Typvariable α_{pcpo}. Auf der anderen Seite bekommt eine binäre Relation in HOL den Typ $\alpha_{top} \Rightarrow \alpha_{top} \Rightarrow bool$, denn nur durch Verwendung des Konstruktors $\Rightarrow$ für den vollen Funktionsraum und der Typkonstanten *bool* kann ausgedrückt werden, daß es sich um ein zweistelliges Prädikat handeln soll. Das Typschema $\alpha_{pcpo} \Rightarrow \alpha_{pcpo} \Rightarrow bool$ läßt sich aber bei Beachtung der Arität $\Rightarrow :(top, top)top$ nicht bilden. Vielmehr würde für die Bildung des Teilterms $\alpha_{pcpo} \Rightarrow bool$ und darauf aufbauend in einem zweiten Schritt für $\alpha_{pcpo} \Rightarrow \alpha_{pcpo} \Rightarrow bool$ die Arität $\Rightarrow :(pcpo, top)top$ benötigt.

Als Lösung für dieses technische Problem kann man die Zuordnung mehrerer Aritäten für einen Typkonstruktor erlauben, hier $\Rightarrow :(top, top)top$ und $\Rightarrow :(pcpo, top)top$, oder man geht gleich darüber hinaus und führt eine Ordnungsbeziehung (Subklassenbeziehung) zwischen Typklassen ein, ähnlich dem Subtypkonzept aus der ordnungssortierten Algebra [Gog78, Gog76, Qia90, Han91]. Dort dürfen auf einem bestimmten Typ definierte Funktionen auch auf Argumente eines jeden Subtyps angewendet werden. Wenn wir diese Idee übertragen, so bietet es sich an, die Klasse *pcpo* als Teilklasse von *top* zu vereinbaren ($pcpo < top$). Dann ist die Bildung des Typterms $\alpha_{pcpo} \Rightarrow \alpha_{pcpo} \Rightarrow bool$ mit der Arität $\Rightarrow :(top, top)top$ aufgrund der ordnungssortierten Interpretation sehr wohl möglich. Die zusätzliche Arität $\Rightarrow :(pcpo, top)top$ erübrigt sich dann.

Dieser Weg wurde von Tobias Nipkow bei der Erweiterung des Isabelle-Typsystems eingeschlagen. Ursprünglich wurde dieses stärkere Typkonzept von ihm eingebaut, um ein feineres Instrument zur Formalisierung von Objektlogiken bereitzustellen. Tobias Nipkow gibt hierzu in [Nip91] einige Beispiele. Da aber das Typsystem der Isabelle-Metalogik auch in den Objektlogiken zur Verfügung steht, ist mit Nipkows Formalisierung von HOL in Isabelle eine Version von HOL mit Typklassen entstanden. Trotz dieses stärkeren Typkonzepts, das die bereits erwähnte Anwendung von Typkonstruktoren auf Typen in Teilklassen und zusätzlich die Überladung von Typkonstruktoren mit mehreren Aritäten erlaubt, bleibt die wichtige Eigenschaft der automatischen Typinferenz erhalten [Nip91, NP93].

Nach Beseitigung der syntaktischen Schwierigkeiten durch Einführung der Polymorphie mit ordnungssortierten Typklassen ist jetzt noch eine Erklärung für die Semantik der Teilklassenbeziehung notwendig. Diese möchte ich anhand des Beispiels der beiden Klassen *top* und *pcpo* und ihrer Teilklassenbeziehung $pcpo < top$ geben. Betrachten wir dazu die Abbildung 2.1.

Jede Klasse wird durch ein eigenes Universum interpretiert. Welche mathematischen Strukturen die Elemente der einzelnen Universen sind, hängt von der jeweiligen Klasse ab. Im Fall der Klasse *top* ist die Interpretation das oben vorgestellte Universum U von Mengen. In Abbildung 2.1 wird es durch die rechte Blase dargestellt. Die Klasse *pcpo* hat als Interpretation ein Universum von Bereichen, was durch die linke Blase in Abbildung 2.1 repräsentiert wird. Die beiden Universen sind aber nicht völlig unabhängig voneinander, sondern sind über eine

Abbildung *fgt* miteinander gekoppelt, so daß eine Reihe von Bedingungen erfüllt ist. Einige dieser Bedingungen seien im folgenden informell beschrieben.

charakteristische Konstanten der Klasse *pcpo*:

$$\bot : \alpha_{pcpo}$$

$$\sqsubseteq : \alpha_{pcpo} \Rightarrow \alpha_{pcpo} \Rightarrow bool$$

Interpretationen der Instanzen für Typ tr:

$$\bot_{tr} \in fgt(tr)$$

$$\sqsubseteq\ tr \in fgt(tr) \Rightarrow fgt(tr) \Rightarrow bool$$

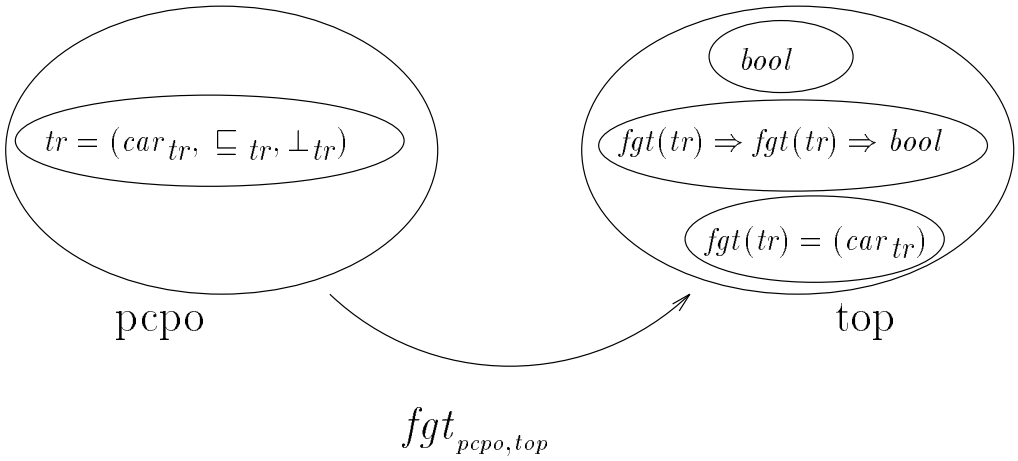


Abbildung 2.1: Teilklassenbeziehung zwischen *pcpo* und *top*

Im linken Universum für die Klasse *pcpo* dürfen nur Bereiche enthalten sein, also Mengen mit ω-cpo Struktur und kleinstem Element. Mathematisch lassen sich solche Bereiche zum Beispiel durch Tripel $X = (car_X, \sqsubseteq_X, \bot_X)$ modellieren, wobei car_X die Trägermenge darstellt, $\sqsubseteq_X$ die Ordnungsrelation und $\bot_X$ das kleinste Element in car_X bzgl. $\sqsubseteq_X$.

Charakteristisch für Elemente im Universum für die Typklasse *pcpo* ist das zusätzliche Vorhandensein der Interpretationen $\sqsubseteq_X$ und $\bot_X$ für die syntaktischen, in α_{pcpo} polymorphen, Konstanten $\sqsubseteq$ und $\bot$. Diese Konstanten werden die *charakteristischen Konstanten der Klasse pcpo* genannt. Über das pure Vorhandensein hinaus müssen diese Interpretationen natürlich auch die intendierten Eigenschaften erfüllen. $\sqsubseteq_X$ muß eine ω-kettenvollständige Ordnungsrelation auf car_X sein, und $\bot_X$ muß wirklich das kleinste Element bzgl. dieser Ordnung sein. Diese Eigenschaften können durch Axiome über den polymorphen Konstanten $\sqsubseteq$ und $\bot$ formuliert werden. Diese Axiome werden die *charakteristischen Axiome der Klasse pcpo* genannt[7]. In Abschnitt 2.3 werden wir sehen, wie charakteristische Konstanten und Axiome

[7] hier im Beispiel erwähne ich nur die beiden Klassen *pcpo* und *top*. Im Kapitel 4 werde ich eine reichhaltigere Klassenhierarchie verwenden und eine zusätzliche Klasse *po* der partiellen Ordnungen einführen.

für eine Klasse als solche in HOLC–Theorien gekennzeichnet werden. Über die spezielle Interpretation dieser charakteristischen Konstanten werde ich gleich noch mehr sagen.

Weiterhin kann man aus jedem Element im linken Universum ein Element im rechten Universum gewinnen, indem man einfach die zusätzliche Bereichsstruktur, d.h. $\sqsubseteq_X$ und $\bot_X$, *vergißt*. Dieses Vergessen der Struktur für jeden beliebigen Bereich in *pcpo* wird mathematisch über die totale Abbildung *fgt* (forget) von *pcpo* nach *top* bewerkstelligt. Die Teilklassenbeziehung $pcpo < top$ wird also durch das Vorhandensein der Vergiß-Abbildung $fgt_{pcpo,top}$ mit den gerade erwähnten Eigenschaften interpretiert.

Allgemein wird die Teilklassenbeziehung $k_2 < k_1$ zwischen zwei Typklassen interpretiert, indem man eine Vergiß-Abbildung fgt_{k_2,k_1} zwischen den beiden Universen für k_1 und k_2 definiert, die gerade die bzgl. k_1 zusätzliche charakteristische Struktur der Elemente in k_2 vergißt und so ein Element im Universum für k_1 erzeugt. Die Abbildung fgt_{k_2,k_1} muß immer total sein, ist im allgemeinen aber weder injektiv noch surjektiv! Im Beispiel aus Bild 2.1 wird $fgt_{pcpo,top}$ wohl nicht injektiv sein, da es viele Möglichkeiten gibt, eine Menge mit einer Bereichsstruktur zu versehen. Vergiß-Abbildungen fgt_{k_2,k_1}, die nicht surjektiv sind, treten immer dann auf, wenn in k_1 Elemente enthalten sind, die nicht geeignet zu Elementen in k_2 erweitert werden können, so daß die charakteristischen Eigenschaften der Klasse k_2 gelten. Man betrachte etwa die Klasse *fin* der *endlichen Typen* mit $fin < top$. Hier gibt es zwar keine charakteristischen Konstanten, aber die charakteristische Eigenschaft, eine endliche Trägermenge zu haben, schließt alle Typen mit unendlicher Trägermenge aus *top* als potentielle Elemente in *fin* aus.

Im Bild 2.1 wird die Abbildung $fgt_{pcpo,top}$ am Beispiel des Elements *tr* gezeigt[8]. Durch Anwendung von *fgt* auf *tr* entsteht nach Vergessen der zusätzlichen Bereichsstruktur ein Element $fgt(tr)$ in der Klasse *top*, welches nur aus der Trägermenge von *tr* besteht, die ich im Bild mit car_{tr} bezeichnet habe. Aufgrund der speziellen Semantik der Klasse *top* bestehen alle Elemente dieses Universums nur aus Trägern. Im allgemeinen, wenn die Zielklasse nicht *top* ist, liefert die Anwendung der *fgt*-Funktion aber Elemente, die nicht nur aus Trägern bestehen, sondern zusätzlich Interpretationen für charakteristische Konstanten enthalten. In diesem Zusammenhang ist bei Verwendung des $\in$ Zeichens stets der Zugriff auf die Trägermenge des Elements gemeint. Als ich vorhin erklärt habe, daß die Semantik der Klasse *top* das HOL-Universum U von Mengen ist, war das nicht ganz korrekt. Eigentlich sind im Universum *top* Elemente enthalten, die nur aus einer Trägermenge bestehen, welche ihrerseits eine Menge im HOL-Universum U ist.

Mit Hilfe der Abbildung *fgt* können wir jetzt auch die Interpretation des Terms $tr \Rightarrow tr \Rightarrow bool$ erklären, wenn $\Rightarrow$ die Arität $(top, top)top$ hat. Zuerst interpretieren wir *tr* im Universum für *pcpo* und erhalten das Tripel in der linken Blase. Dann wenden wir *fgt* an, was zu einem Element $fgt(tr)$ in der Klasse *top* führt. Auf dieses Element dürfen wir die Interpretation von $\Rightarrow$ anwenden und bekommen somit die Interpretation $fgt(tr) \Rightarrow fgt(tr) \Rightarrow bool$ für den Typterm $tr \Rightarrow tr \Rightarrow bool$. Analog erklärt man die Interpretation für den Typterm $\alpha_{pcpo} \Rightarrow \alpha_{pcpo} \Rightarrow bool$, der uns vorhin schon bei der Motivation für die Einführung von Typklassen als Typschema für die polymorphe Konstante $\sqsubseteq$ begegnet ist.

Die Erklärung für polymorphe Konstanten habe ich oben bereits für den normalen Hindley/Milner Polymorphismus gegeben. Für die Polymorphie mit Typklassen läßt sich die

[8] um die Notation übersichtlich zu halten, habe ich im Bild nicht zwischen dem Typ *tr* und seiner Interpretation als Bereich unterschieden. Dasselbe gilt für den Typ *bool* und den Typkonstruktor $\Rightarrow$.

übliche Sichtweise einfach anpassen. Die Idee mit der Indizierung einer Familie durch Elemente eines Typuniversums wird übernommen. Betrachten wir die polymorphe Konstante c mit Typschema $\tau(\alpha_{pcpo}, \beta_{top})$. Dabei steht $\tau(\alpha_{pcpo}, \beta_{top})$ für ein Typschema τ, in dem die Typvariablen α_{pcpo} und β_{top} vorkommen. Die Interpretation von c ist wieder eine ganze Familie, die diesmal durch einen zweistelligen Index indiziert wird. An erster Stelle wird mit Elementen aus dem Universum für *pcpo* indiziert und an zweiter Stelle mit Elementen aus dem Universum *top*. Dies wird durch die im Schema vorkommenden Typvariablen der verschiedenen Klassen festgelegt[9]. Sei nun τ_1 ein Typterm in der Klasse *pcpo* und τ_2 ein Typterm in der Klasse *top*[10], und sei fernerhin unter irgendeiner Belegung die Interpretation für τ_1 der Bereich $B_1 \in pcpo$, die Interpretation für τ_2 das Element $M_2 \in top$ und E die Interpretation der τ_1, τ_2-Instanz $\tau[\tau_1/\alpha_{pcpo}, \tau_2/\beta_{top}]$. E ist dabei ein Element im Universum für die Klasse, die dem Typterm $\tau[\tau_1/\alpha_{pcpo}, \tau_2/\beta_{top}]$ zugeordnet wird. Dann ist die Interpretation des Terms $c{:}\tau[\tau_1/\alpha_{pcpo}, \tau_2/\beta_{top}]$ dasjenige Element in der Familie, welches mit (B_1, M_2) indiziert wird und welches im Träger der Struktur E liegt. Die formale Definition dieser Begriffe in Abschnitt 2.4 wird zeigen, daß hier nur an den richtigen Stellen die entsprechenden Vergiß-Abbildungen eingesetzt werden müssen, damit die Idee für den Hindley/Milner Polymorphismus auf Polymorphie mit Typklassen hochgezogen werden kann.

Das eben gesagte trifft auf alle polymorphen Konstanten zu. Für die Interpretation der charakteristischen Konstanten einer Klasse, die immer polymorph in genau einer Typvariablen dieser Klasse sind, gelten darüber hinaus noch weitere Eigenschaften. Betrachten wir dazu die charakteristische Konstante $\sqsubseteq$ der Klasse *pcpo* aus unserem Beispiel in Abbildung 2.1 und überprüfen dabei zunächst die Modellvorstellung aus dem letzten Absatz. Die Interpretation von $\sqsubseteq$ ist eine Familie, die wegen des Typschemas $\alpha_{pcpo} \Rightarrow \alpha_{pcpo} \Rightarrow bool$ mit einem Element aus dem Universum für die Klasse *pcpo* indiziert wird. Dieses Universum, die linke Blase, ist im Bild der Einfachheit halber ebenfalls mit *pcpo* bezeichnet. Die Interpretation des Typterms tr ist das Tripel $(car_{tr}, \sqsubseteq_{tr}, \bot_{tr})$, welches im Bild wieder einfach mit tr abgekürzt wird. Gemäß dem letzten Abschnitt ist die Interpretation der tr-Instanz $\sqsubseteq : tr \Rightarrow tr \Rightarrow bool$, im Bild mit $\sqsubseteq_{tr}$ bezeichnet, dasjenige Element der Familie, welches mit dem Tripel tr indiziert ist, und welches ein Element im Träger der Interpretation des Typterms $tr \Rightarrow tr \Rightarrow bool$ ist. Die Interpretation dieses Typterms ist aber wie bereits beschrieben das Element $fgt(tr) \Rightarrow fgt(tr) \Rightarrow bool$ im Universum *top*. Soweit gibt es keinen Unterschied zu anderen polymorphen Konstanten.

Der besondere Punkt bei der Interpretation charakteristischer Konstanten ist, daß das indizierte Mitglied der Familie, hier $\sqsubseteq_{tr}$, bereits im Index $(car_{tr}, \sqsubseteq_{tr}, \bot_{tr})$ enthalten ist. In dieser Eigenheit finden wir die LCF-Idee wieder, daß die Interpretation des polymorphen Ordnungssymbols $\sqsubseteq$ immer gerade die Ordnung auf dem Bereich ist, für den wir die polymorphe Funktion spezialisieren (instantiieren). Im allgemeinen gilt für alle Elemente im Universum einer Klasse, daß sie die Interpretationen all der für die Klasse charakteristischen Konstanten mit sich herumtragen[11].

Nun können wir die Syntax und Semantik von Typen in der Klasse *pcpo* unter einem neuen Blickwinkel betrachten. Der Typterm tr steht für den ganzen Bereich. Ausdrücke wie $TT{:}tr$

[9] welche Stelle im Index dabei von welcher Typvariablen gekennzeichnet wird, entscheidet man über eine geeignete totale Ordnung auf Typvariablen.

[10] bzw. ein Typterm in einer entsprechenden Teilklasse.

[11] in Abschnitt 2.4 werden wir sehen, daß auch die Interpretationen für die char. Konstanten aller Oberklassen in geeigneter Weise in den Elementen der Klassen kodiert sind.

bedeuten, daß die Interpretation der Konstanten TT ein Element im Träger des Typs tr ist. Auf die partielle Ordnung des Bereichs greifen wir über die Instantiierung der charakteristischen Konstanten $\sqsubseteq$ zu, und das kleinste Element ist über die charakteristische Konstante $\bot$ erreichbar. Die charakteristischen Konstanten einer Klasse fungieren sozusagen als Selektoren für die Komponenten der Elemente in den Klassenuniversen. Auf den Träger wird dabei syntaktisch nur implizit Bezug genommen.

Wenn wir diese Behandlung von Bereichen als Typen mit der Formalisierung von Bereichen in den Arbeiten von Petersen und Agerholm vergleichen, so werden die Vorteile der Polymorphie mit Typklassen sofort deutlich. Statt immer explizit in der Syntax mit Paaren (D, R) zu arbeiten und die Eigenschaften der Bereiche eigenhändig in Form von Prämissen über D und R herumzureichen, wird die Ordnung über die charakteristische polymorphe Klassenkonstante $\sqsubseteq$ erreichbar gemacht, und die Eigenschaften werden durch die charakteristischen Klassenaxiome ausgedrückt. Die logische Aussage, daß eine Konstruktion auf Bereichspaaren (D, R) wieder einen Bereich erzeugt, wird hier über die Zuordnung von Aritäten an Typkonstruktoren bewerkstelligt. Der Formalismus der Aritäten kann als einfache Hornklausellogik aufgefaßt werden, und die Typinferenz ist ein automatischer Beweiser für Mengen solcher Hornklauseln, der auf der Grundlage der Aritäten berechnet, ob ein Typterm in einer bestimmten Klasse liegt. Hier finden sich also in einer schwachen Form die Ideen des *propositions as types* Paradigmas aus der konstruktiven Typtheorie wieder. Teile der Logik, die bei Petersen und Agerholm explizit auf der Termebene abgehandelt werden müssen, werden hier automatisch von Typsystem erledigt.

2.2.3 Einführung einer neuen Klasse

Wenn man eine Theorie durch Signatur oder Axiome erweitert, dann stellt sich immer die Frage, ob es für die neue Theorie Modelle gibt. Diese Frage läßt sich nicht allgemein beantworten, sondern hängt immer von der speziellen Erweiterung ab. Man kann aber für gewisse Erweiterungsschemata die Existenz von Modellen garantieren, wenn die Theorie, auf der die Erweiterung basiert, ein Modell hat. Für HOL sind solche Schemata bekannt, die sogenannten konservativen Theorieerweiterungen. In den folgenden Abschnitten werde ich informell Schemata einführen, die die konservative Erweiterung von Theorien um neue Klassen und Aritäten für Typkonstruktoren sicherstellen. Die hier vorgestellten Schemata werden formal in Abschnitt 2.6 behandelt und stellen den Satz von Schemata dar, der neben den von HOL bekannten Erweiterungsmechanismen notwendig war, um HOLCF konservativ zu entwickeln.

Zuerst möchte ich erklären, wie man zu einer Theorie eine neue Klasse hinzufügt und welche Arbeiten im Vorfeld nötig sind, um die konservative Erweiterbarkeit für Modelle der alten Theorie zu gewährleisten. Betrachten wird dazu eine Theorie *Base*, die nur die Typklasse *top* enthält. Dies könnte zum Beispiel die pure HOL-Theorie von Gordon sein, da die Typklasse *top* in der Logik HOLC gerade die Rolle des kompletten Typuniversums der Logik HOL übernimmt. Weiterhin sei in der Theorie *Base* der Typ *void* zusammen mit einer binären Relation *less_void* auf dem Typ *void* bekannt. Die Rolle des Typs *void* und der Relation *less_void* wird im Laufe des folgenden Beispiels klar werden.

Zu dieser Theorie werden wir jetzt die Typklasse *po* der partiellen Ordnungen mit der charakteristischen Konstanten $\sqsubseteq$ und den charakteristischen Axiomen für eine partielle Ordnung

hinzufügen. Ich wähle das Beispiel der Klasse *po*, da hier nur eine charakteristische Konstante notwendig ist und die charakteristischen Axiome einfach zu formulieren sind. Zudem entspricht die Erweiterung um *po* auch genau der Vorgehensweise, die ich bei der Entwicklung von HOLCF in Kapitel 4 angewendet habe. Der Grund, wieso ich in den vorangegangenen Abschnitten gleich mit der Klasse *pcpo* von ω-kettenvollständigen partiellen Ordnungen mit kleinstem Element gearbeitet habe, liegt darin, daß ich die Einführung des Klassenmechanismus über den Versuch der Integration von HOL und LCF motiviert habe. Hier war die Verwendung der Klasse *pcpo* am naheliegendsten.

Zur Notierung des Beispiels verwende ich gleich die Syntax des Isabelle-Systems[12], die in Kapitel 3 noch genauer beschrieben wird. Die Erweiterung der Theorie *Base* durch die Klasse *po* ist in Bild 2.2 abgebildet.

$Porder = Base +$

classes $po < top$
arities $void : po$

consts
  $\sqsubseteq : \alpha_{po} \Rightarrow \alpha_{po} \Rightarrow bool$ (infixl 55)

rules
  refl_less $x \sqsubseteq x$
  antisym_less $x \sqsubseteq y \wedge y \sqsubseteq x \rightarrow x = y$
  trans_less $x \sqsubseteq y \wedge y \sqsubseteq z \rightarrow x \sqsubseteq z$

  inst_void_po $(\sqsubseteq : void \Rightarrow void \Rightarrow bool) = less\_void$
end

Abbildung 2.2: Neue Typklasse *po*

Die neue Theorie *Porder* baut auf der Theorie *Base* auf, was mit dem Zeichen + angezeigt wird. Zuerst wird mit dem Schlüsselwort 'classes' die neue Klasse *po* eingeführt und deren Einordnung in die Klassenhierarchie mittels $po < top$ angezeigt. Die nächste Zeile mit Schlüsselwort 'arities' vereinbart für den Typkonstruktor *void* die Arität $void : po$, d.h. der Typ *void* soll in der Klasse *po* liegen. Über den Zweck dieser Vereinbarung werde ich weiter unten noch reden, vorerst können wir diese Zeile jedoch ignorieren. Danach wird mit dem Schlüsselwort 'consts' der Teil der Spezifikation eingeleitet, in dem die neuen Konstanten vereinbart werden. Hier ist es nur die polymorphe Konstante $\sqsubseteq$ mit Typschema $\alpha_{po} \Rightarrow \alpha_{po} \Rightarrow bool$, die die charakteristische Konstante für die Klasse *po* sein soll. Für die Konstante wird auch gleich eine Infix-Schreibweise nebst Bindungsstärke vereinbart. Im letzten Abschnitt, der mit dem Schlüsselwort 'rules' beginnt, stehen die zusätzlichen Axiome der Theorie *Porder*. Die ersten drei Axiome beschreiben die charakteristischen Eigenschaften der Klasse *po*. Bemerkenswert

[12] die Syntax ist sehr ähnlich zur Isabelle-Syntax. Ich verwende jedoch graphische Sonderzeichen und unterdrücke syntaktische Feinheiten, die für das Beispiel irrelevant sind.

dabei ist, daß die charakteristische Konstante $\sqsubseteq$ zur Formulierung der Ordnungseigenschaften benutzt wird, und daß die Axiome genau in der Variablen α_{po} polymorph sind[13]. Das letzte etwas abgesetzte Axiom beschreibt die Instanz der Ordnungsfunktion $\sqsubseteq$ für den Typ *void*. Wie die Vereinbarung der Arität *void*:*po* wollen wir dieses Axiom zunächst ignorieren.

In Abschnitt 2.3 über die formale Syntax von HOLC werden wir sehen, wie gewisse Konstanten der Signatur und gewisse Axiome als charakteristisch für eine Klasse ausgezeichnet werden können. Beim Betrachten des Beispiels stellen wir jedoch fest, daß in der verwendeten Isabelle-Syntax keine Möglichkeit besteht, charakteristische Konstanten und Axiome auszuzeichnen. Hier zeigt es sich, daß der Klassenmechanismus ursprünglich nicht dazu gedacht war, um in Objektlogiken eingesetzt zu werden. Über dieses kleine notationelle Problem helfe ich mir durch folgende Konvention hinweg. Wenn in einer Spezifikation das Schlüsselwort 'classes' vorkommt, dann darf nur genau eine neue Klasse eingeführt werden. Damit wird diese Spezifikation als Klassenspezifikation gekennzeichnet. Alle Konstanten, die eingeführt werden, sind charakteristische Konstanten dieser Klasse. Alle Axiome außer Instanzaxiomen, die an ihrer syntaktischen Form leicht zu erkennen sind, sind charakteristische Axiome der neuen Klasse.

Syntaktisch haben wir die Klasse *po* jetzt eingeführt, und es bleibt noch zu zeigen, wie ein beliebiges Modell M_1 für die Theorie *Base* zu einem Modell M_2 für die Theorie *Porder* erweitert werden kann, so daß die Restriktion von M_2 bzgl. der Signatur der Theorie *Base* wieder das ursprüngliche Modell M_1 ergibt. Sei also jetzt ein Modell M_1 für die Theorie *Base* gegeben. Aus der vorher skizzierten Modellvorstellung für die Polymorphie mit Typklassen ersehen wir, daß wir ein neues Universum konstruieren müssen, das als Interpretation für die Klasse *po* dienen kann. Als Ausgangspunkt für die Konstruktion wählen wir das Universum für die Typklasse *top*, denn *top* wurde in der Theorie *Porder* mittels $po < top$ als unmittelbare Oberklasse von *po* vereinbart. Der Einfachheit halber bezeichnen wir die Universen, welche als Interpretationen für die Typklassen dienen, wieder mit den Namen der Typklassen. Die Interpretation für die neue Klasse *po* ist das Universum *po*, das durch folgende (semiformale) Definition gegeben ist.

$$po = \{X = (car_X,\ \sqsubseteq_X) \mid \left(\begin{array}{l} \text{es gibt ein } \widehat{X} = (car_{\widehat{X}}) \in top, \text{ so daß:} \\ car_X = car_{\widehat{X}} \wedge \\ \sqsubseteq_X \in \widehat{X} \Rightarrow \widehat{X} \Rightarrow bool \wedge \\ \text{char. Axiome von } po \text{ gelten unter } \alpha_{po} \mapsto (car_X,\ \sqsubseteq_X) \end{array}\right)\}$$

In *po* sind also Paare[14] $X = (car_X,\ \sqsubseteq_X)$ enthalten, so daß $\widehat{X} = (car_X)$ ein Element im Universum *top* ist und $\sqsubseteq_X \in \widehat{X} \Rightarrow \widehat{X} \Rightarrow bool$ eine typkorrekte Interpretation für die Spezialisierung der polymorphen Konstanten $\sqsubseteq$ auf X ist, wie es unserer Vorstellung von der Instanz einer charakteristischen Funktion entspricht. Die dritte Bedingung sagt, daß die charakteristischen Axiome der Klasse *po* gelten müssen, wenn wir sie in einer Belegung interpretieren, die der Typvariablen α_{po} das Paar $(car_X,\ \sqsubseteq_X)$ zuweist. Mit anderen Worten besteht das Universum *po* aus allen partiellen Ordnungen $(car_X,\ \sqsubseteq_X)$, die einen Träger $car_X = car_{\widehat{X}}$ eines Elements

[13] diese Tatsache sieht man natürlich nur, wenn man versucht die Axiome durch die allgemeinst mögliche Typinformation zu ergänzen. Diese ergibt für die Variablen x, y, z jeweils den Typ α_{po}.

[14] in Abschnitt 2.4 werden wir sehen, daß die Kodierung geringfügig komplizierter ist. Für das Beispiel hier genügt aber die Benutzung von Paaren.

$\widehat{X}$ aus dem Universum *top* verwenden und die mit irgendeiner Ordnungsfunktion $\sqsubseteq_X$ auf car_X angereichert sind.

Nachdem das Universum für die neue Typklasse konstruiert ist, müssen wir die Funktion *fgt* definieren, die die beiden Klassen *po* und *top* gemäß der Teilklassenbeziehung $po < top$ koppelt. Die Aufgabe der Funktion *fgt* ist das Vergessen der zusätzlichen Struktur und somit folgt deren Definition kanonisch aus der Konstruktion der Paare in der Klasse *po*. Wir definieren:

$$fgt_{po,top}((car_X, \sqsubseteq_X)) := (car_X)$$

Die Interpretation der charakteristischen Konstanten $\sqsubseteq$ ist ebenso kanonisch vorgegeben, denn bei Indizierung der Familie durch das Element $(car_X, \sqsubseteq_X)$ wird nach Modellvorstellung immer $\sqsubseteq_X$ ausgewählt. Ein Modell M_2 für die Theorie *Porder* erhalten wir, indem wir das Modell M_1 für die Theorie *Base* um die Interpretation der Klasse *po*, die Funktion $fgt_{po,top}$ und die Interpretation der polymorphen Konstanten $\sqsubseteq$ erweitern. Die Modellerweiterung ist konservativ über *Base*, da die Restriktion des Modells M_2 bzgl. der Signatur der Theorie *Base* gerade wieder M_1 ergibt.

Bei der obigen Konstruktion habe ich vorausgesetzt, daß sie überhaupt durchführbar ist. Der einzige Punkt, an dem die Konstruktion scheitern könnte, ist die Bildung des neuen Universums. Wir bilden hier eine Menge von Paaren $(car_X, \sqsubseteq_X)$ und müssen sicherstellen, daß die Menge nicht leer ist[15]. Daß sich Paare $X = (car_X, \sqsubseteq_X)$ mit

$$(car_X) \in top \wedge \sqsubseteq_X \in \widehat{X} \Rightarrow \widehat{X} \Rightarrow bool$$

bilden lassen, steht außer Frage. Es könnte jedoch sein, daß es kein Paar $(car_X, \sqsubseteq_X)$ gibt, für das die Klassenaxiome gelten[16].

Um diesem Fall bei der Einführung einer neuen Klasse vorzubeugen, sucht man im Vorfeld einen Typ, für den die charakteristischen Axiome der Klasse sicher gelten. Dieser Typ wird der *Prototyp* der Klasse genannt und ist in unserem Beispiel der Typ *void*, der genau ein Element in der Trägermenge hat. Die Axiomatisierung des Typs *void* unterschlage ich in diesem Beispiel und zähle nur die Annahmen auf, die für die Einführung der Klasse *po* wichtig sind.

- der Typ *void* ist schon in der Theorie *Base* bekannt und hat in dieser die Arität $void : top$. Diese Arität wurde ihm automatisch verliehen, als er gemäß der HOL-Methode als Typkonstruktor eingeführt wurde.
- weiterhin gibt es eine Funktion *less_void* vom Typ $void \Rightarrow void \Rightarrow bool$ für die folgende Theoreme in der Theorie *Base* bewiesen wurden:

$$\begin{aligned} &less\_void(x)(x) \\ &less\_void(x)(y) \wedge less\_void(y)(x) \rightarrow x = y \\ &less\_void(x)(y) \wedge less\_void(y)(z) \rightarrow less\_void(x)(z) \end{aligned}$$

[15] analog zu leeren Typen versucht man auch leere Klassen zu vermeiden, da sonst die Semantik und auch die Formalisierung des Kalküls umständlicher wird. Läßt man leere Klassen zu, dann muß man im Kalkül explizite Kontexte für die im Beweis benutzten Klassen vorsehen.

[16] im Beispiel der Klasse *po* ergeben sich natürlich keine Schwierigkeiten, im allgemeinen Fall kann es aber sein, daß die Axiome einer Klasse in keinem Typ gültig sind. Die Einführung solcher Klassen muß verhindert werden.

Die Arität $void{:}top$ bedeutet, daß der Typ *void* in jedem Modell von *Base* durch ein Element $\widehat{X} = (car_{\widehat{X}}) \in top$ mit nichtleerer Trägermenge $car_{\widehat{X}}$ interpretiert wird. Die Existenz der Funktion *less_void* bedeutet, daß in jedem Modell von *Base*, das den Typ *void* durch das Element $\widehat{X}$ interpretiert, die Interpretation der Funktion *less_void* ein Element im Träger von $\widehat{X} \Rightarrow \widehat{X} \Rightarrow bool$ ist. Wegen der bewiesenen Theoreme wissen wir aber auch, daß es zumindest ein Paar $X = (car_X, \sqsubseteq_X)$ gibt, so daß die Axiome der Klasse *po* gelten. Man nehme für car_X einfach die Trägermenge $car_{\widehat{X}}$ der Interpretation $\widehat{X}$ des Typs *void* und für $\sqsubseteq_X$ die Interpretation der Funktion *less_void*. Somit ist die Konstruierbarkeit für das Universum *po* und damit auch die konservative Erweiterung von M_1 zu M_2 sichergestellt.

Der Typ *void* und die Eigenschaften der Funktion *less_void* sind also Zeugen dafür, daß jedes Modell der Theorie *Base* zu einem Modell M_2 der Theorie *Porder* erweitert werden kann. Um dieses Zeugnis in der Klassenspezifikation von *po* zu fixieren, wird dort an *void* die zusätzliche Arität $void{:}po$ vergeben und in den Axiomen die Instanz von $\sqsubseteq$ für den Typ *void* auf die Funktion *less_void* festgelegt. Das erklärt die Teile der Klassenspezifikation von *po*, die ich bisher nicht kommentiert habe.

2.2.4 Hinzufügen neuer Aritäten

In diesem Abschnitt möchte ich erklären, wie man die Vereinbarung einer neuen Arität $(\bar{k}_1, \ldots, \bar{k}_n)\bar{k}$ für einen Typkonstruktor *tc* rechtfertigt. Dabei wird vorausgesetzt[17], daß der Typkonstruktor *tc* schon mit einer Arität $(k_1, \ldots, k_n)k$ ausgestattet ist, so daß $\bar{k} < k \wedge \forall i.\bar{k}_i < k_i$. Insbesondere muß k ein unmittelbarer Nachfolger von $\bar{k}$ bzgl. der Klassenordnung sein. Die Teilklassenbeziehung $\bar{k}_i < k_i$ bedeutet, daß die Typen in $\bar{k}_i$ mehr Struktur aufweisen als die Typen in k_i. Daher kann man die neue Arität so deuten, daß der Typkonstruktor *tc* die zusätzliche Struktur der Typen in $\bar{k}_i$ dazu benutzt, einen Ergebnistyp in der Klasse $\bar{k}$ zu konstruieren, der wegen $\bar{k} < k$ eine bzgl. der Klasse k zusätzliche Struktur aufweist. Wenn k ein unmittelbarer Nachfolger von $\bar{k}$ ist, dann wird die zusätzliche Struktur der Typen in $\bar{k}$ genau durch die Klassenspezifikation von $\bar{k}$ gegeben. Das wiederum erleichtert die Formulierung der Konstruktion und aller damit zusammenhängenden Bedingungen.

Die Voraussetzung, daß für den Konstruktor *tc* schon die Arität $(k_1, \ldots, k_n)k$ vereinbart ist, bedeutet, daß *tc* schon in der Basistheorie bekannt sein muß. Mit dem Schema, das ich gleich beschreiben werde, kann man also keine neuen Typkonstruktoren einführen. Eine Methode, um einen neuen Konstruktor einzuführen, ist die Methode der Erweiterung durch Typdefinition, wie sie für die Logik HOL bekannt ist. Diese ist in [GM93] ausführlich beschrieben und wird auch von mir in Abschnitt 2.6 nochmals vorgestellt. Die Typkonstruktoren der Logik HOL entsprechen Konstruktoren in HOLC, die eine Arität $(k_1, \ldots, k_n)top$ haben. Daher kann man die HOL-Methode verwenden, um einen Konstruktor mit Arität $(k_1, \ldots, k_n)top$ einzuführen und aufbauend auf dieser Arität dann mit dem hier beschriebenen Mechanismus weitere Aritäten für den Konstruktor einführen.

Das Schema zur Einführung einer neuen Arität möchte ich hier wieder an einem Beispiel erklären. Betrachten wir dazu den Typkonstruktor $\Rightarrow$ mit der Arität $(top, top)top$. Bekanntlich kann man Funktionen partiell ordnen, wenn für die Bildmenge eine partielle Ordnung

[17] in Abschnitt 2.6 werden noch mehr Annahmen gemacht, die für die Erhaltung der automatischen Typinferenz wichtig sind. Diese sind jedoch für diese informelle Erklärung unwichtig.

bekannt ist, und man die Funktionen gemäß dieser Ordnung punktweise ordnet. Dies legt die neue Arität $(top, po)po$ für den Typkonstruktor $\Rightarrow$ nahe. Die Einführung dieser Arität erfolgt in drei Schritten. Zuerst definieren wir eine polymorphe Funktion *less_fun*[18] mit Typschema $(\alpha_{top} \Rightarrow \beta_{po}) \Rightarrow (\alpha_{top} \Rightarrow \beta_{po}) \Rightarrow bool$. Die entsprechende Theorie ist in Bild 2.3 dargestellt.

$Fun1 = Porder +$

consts
$less\_fun$: $(\alpha_{top} \Rightarrow \beta_{po}) \Rightarrow (\alpha_{top} \Rightarrow \beta_{po}) \Rightarrow bool$

rules
less_fun_def $\quad less\_fun = (\lambda f_1 f_2 . \forall x . f_1(x) \sqsubseteq f_2(x))$

end

Abbildung 2.3: Definition der punktweisen Ordnung

Man beachte, wie auf der rechten Seite der Definition in Bild 2.3 das polymorphe Ordnungssymbol $\sqsubseteq$ verwendet wird. Dies ist syntaktisch korrekt, da bei Ergänzung der vollen Typinformation die Terme $f_1(x)$ und $f_2(x)$ den Typ β_{po} haben.

Im zweiten Schritt weisen wir nach, daß die Funktion *less_fun* den Raum der Funktionen $\alpha_{top} \Rightarrow \beta_{po}$ partiell ordnet. Genauer gesagt beweisen wir, daß die charakteristischen Axiome der Klasse *po* gelten, wenn wir in den Axiomen die charakteristische Konstante $\sqsubseteq$ durch die Funktion *less_fun* ersetzen. In der Theorie *Fun1* sind also folgende Theoreme zu zeigen:

$$
\begin{aligned}
&less\_fun(x)(x)\\
&less\_fun(x)(y) \wedge less\_fun(y)(x) \rightarrow x = y\\
&less\_fun(x)(y) \wedge less\_fun(y)(z) \rightarrow less\_fun(x)(z)
\end{aligned}
$$

Jetzt sind alle Voraussetzungen für die Einführung der neuen Arität $\Rightarrow : (top, po)po$ gegeben. Betrachten wir dazu die Abbildung 2.4. Sei ein Modell für *Fun1* gegeben, und sei die Interpretation des Typterms τ_1 ein Element $M_1 \in top$ und die Interpretation des Typterms τ_2 die partielle Ordnung $P_2 \in po$. Dann wird der Typterm $\tau_1 \Rightarrow \tau_2$ gemäß der Arität $\Rightarrow : (top, top)top$ als Element $M_2 \in top$ mit $M_2 = (M_1 \Rightarrow fgt_{po,top}(P_2))$ interpretiert.

Weiterhin ist die Interpretation der τ_1, τ_2-Instanz der polymorphen Funktion *less_fun* ein Element $less\_fun_{M_1,P_2} \in M_2 \Rightarrow M_2 \Rightarrow bool$. Ziehen wir noch die bewiesenen Theoreme für *less_fun* in Betracht, dann sehen wir, daß das Paar $(car_{M_2}, less\_fun_{M_1,P_2})$ sicher eines der Paare ist, die bei der konservativen Konstruktion der Klasse *po* als Elemente des Universums *po* aufgenommen wurden. Die Interpretation des Typkonstruktors $\Rightarrow$ mit Arität $(top, po)po$ kann somit als diejenige Abbildung festgelegt werden, die Argumente M_1 und P_2 auf das Paar $(car_{M_2}, less\_fun_{M_1,P_2})$ abbildet, wobei $M_2 = (M_1 \Rightarrow fgt_{po,top}(P_2))$ ist. Hier zeigt sich,

[18] der Name der Funktion soll nicht bedeuten, daß mir die Formalisierung dieser Theorie keinen Spaß gemacht hat.

daß bei der Einführung einer neuen Klasse alle potentiell möglichen Elemente des Universums hinzugenommen werden müssen, damit bei einer späteren Erweiterung durch eine neue Arität das konstruierte Paar sicher im Universum enthalten ist.

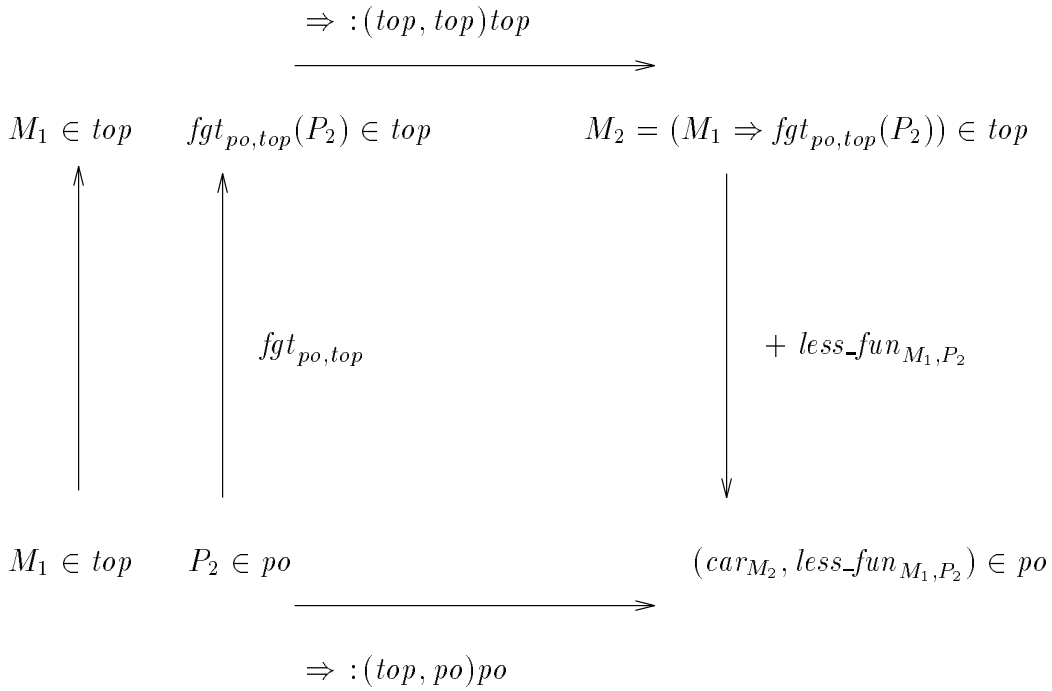


Abbildung 2.4: Beziehung der Überladungen von $\Rightarrow$

Diese Überlegungen zeigen, daß für die neue Arität $\Rightarrow :(top, po)po$ eine Interpretation gefunden werden kann, und daß das Modell von *Fun1* konservativ erweitert werden kann, indem man die eben skizzierte Interpretation für die neue Arität hinzufügt. Im dritten und letzten Schritt, der in Bild 2.5 gezeigt ist, führen wir die Theorie *Fun2* mit der neuen Arität nebst Instanzaxiom für die Ordnung ein.

Fun2 = *Fun1* +

arities $\Rightarrow :(top, po)po$

rules
  inst_fun_po $(\sqsubseteq :(\alpha_{top} \Rightarrow \beta_{po}) \Rightarrow (\alpha_{top} \Rightarrow \beta_{po}) \Rightarrow bool) = less\_fun$
end

Abbildung 2.5: Einführung der neuen Arität und Instanz

Nach dieser informellen Erläuterung für die Polymorphie mit Typklassen in HOLC folgt in den verbleibenden Abschnitten dieses Kapitels die formale Behandlung der eben vorgestellten Konzepte.

2.3 Syntax von HOLC

In diesem Abschnitt der Arbeit werde ich die formale Syntax der Logik HOLC (Higher-Order Logic with Classes) einführen. Ich werde mich dabei, wie auch in den folgenden Abschnitten über die Semantik 2.4, das Deduktionssystem 2.5 und die Methode der konservativen Theorieerweiterung 2.6, so kurz wie möglich fassen, da die Formalisierung von HOLC nicht das zentrale Thema meiner Arbeit darstellt.

2.3.1 Typsignaturen und Typterme

Zuerst führe ich einige Mengen für Bezeichner ein. Es seien folgende unendlichen und jeweils paarweise disjunkten Mengen gegeben:

KID: die Menge KID enthält Namen für Klassen. Auf KID sei eine totale, strikte Ordnung $\prec_{\mathrm{KID}}$ gegeben.

CID: die Menge CID enthält Namen für Konstanten.

TID: die Menge TID enthält Namen für Typkonstruktoren.

TVID: die Menge TVID enthält Präfixe für Typvariablen. Auf TVID sei eine totale, strikte Ordnung $\prec_{\mathrm{TVID}}$ gegeben. Ich werde stets kleine griechische Buchstaben mit evtl. zusätzlichen Ziffern für die Präfixe verwenden.

Ξ: die Menge Ξ ist die eigentliche Bezeichnermenge für Typvariablen. Sie wird aus den Mengen KID und TVID gebildet und ist folgendermaßen definiert:

$$\Xi = \{tv_k \mid tv \in \mathrm{TVID} \ \wedge k \in \mathrm{KID}\}$$

Die totalen Ordnungen der Mengen KID und TVID werden auf Ξ wie folgt fortgesetzt:

$$tv1_{k1} \prec tv2_{k2} \iff \begin{array}{l}(tv1 \prec_{\mathrm{TVID}} tv2) \vee \\ (tv1 = tv2 \wedge k1 \prec_{\mathrm{KID}} k2)\end{array}$$

Die Menge aller Typvariablen Ξ läßt sich partitionieren in die Mengen der Typvariablen Ξ_k der Klassen $k \in \mathrm{KID}$, indem man definiert:

$$\Xi_k = \{tv_k \mid tv_k \in \Xi \wedge k \in \mathrm{KID}\}$$

Ψ: die Menge Ψ ist die Menge der Termvariablen.

Der Zweck der oben geforderten totalen Ordnungen wird im Abschnitt 2.4 deutlich werden. Dort muß für polymorphe Konstanten eine Semantik definiert werden, die im allgemeinen ein Element aus einem mehrstelligen, verallgemeinerten kartesischen Produkt sein wird. Die Stelligkeit ergibt sich dabei aus der Anzahl der im Typschema der polymorphen Konstanten vorkommenden Typvariablen. Mittels der totalen Ordnung können diese Typvariablen kanonisch geordnet werden, und dies ermöglicht dann letztendlich die Definition einer eindeutigen

Semantik. Auf diese Weise kann man die Typen polymorpher Konstanten als Schemata behandeln, die beliebig instantiiert werden dürfen. Dieser Trick wird auch in [GM93] verwendet, wohingegen in der älteren Version [Cam89] noch ein diesbezüglicher Fehler enthalten war. Alternativ könnte man auch einen expliziten Bindungsmechanismus für Typvariablen einführen, der die Reihenfolge der Typvariablen festlegen würde. Dies verkompliziert aber unnötig die Syntax und Semantik.

Im folgenden werde ich öfters endliche Abbildungen verwenden und verschiedene Notationen für sie benützen. Es ist jedoch immer das durch folgende Definition gegebene Konzept gemeint.

Definition 2.1 *Endliche Abbildung*

Eine endliche Abbildung f von der Menge A in die Menge B ist eine rechtseindeutige, endliche Teilmenge $f \subseteq A \times B$ des Kreuzprodukts $A \times B$ und wird als Menge von Paaren $f = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ notiert. Die Eigenschaft *rechtseindeutig* bedeutet dabei wie üblich:

$$\forall (x_1, y_1), (x_2, y_2) \in f.\ x_1 = x_2 \Longrightarrow y_1 = y_2$$

Die Anwendung einer endlichen Abbildung ist definiert durch:

$$f(x) = \begin{cases} y & \text{wenn } (x, y) \in f \\ \text{Fehler} & \text{sonst} \end{cases}$$

Statt der oben gerade eingeführten Notation $f = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ werde ich bisweilen auch alternative Notationen wie $f = \{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ oder $f = \{x_1 : y_1, \ldots, x_n : y_n\}$ verwenden.

Als nächstes wird der Begriff der Typsignatur eingeführt.

Definition 2.2 *Typsignatur*

Eine Typsignatur Ω ist ein Tupel $\Omega = (K, \leq, TC)$, so daß folgende Eigenschaften erfüllt sind:

E1: $K \subseteq$ KID ist eine endliche Teilmenge von Klassenbezeichnern. Insbesondere enthält K den Klassenidentifikator *top*.

E2: $\leq$ ist eine partielle Ordnung auf K, die durch eine endliche Menge von Paaren $(k_1, k_2) \in K \times K$ notiert wird. Die Ordnung $\leq$ steht für die Teilklassenbeziehung zwischen den Typklassen und hat nichts mit der totalen Ordnung $\prec_{\text{KID}}$ zu tun. Insbesondere muß *top* die größte Klasse bezüglich $\leq$ sein.

E3: TC ist eine endliche Menge von Tripeln $(tc, [k_1, \ldots, k_n], k) \in \text{TID} \times K\,list \times K$, durch die alle Aritätsvereinbarungen für Typkonstruktoren kodiert werden. Statt der Tripelnotation verwende ich die Notation $tc{:}(k_1, \ldots, k_n)k$ bzw. $tc{:}k$, falls die Liste leer ist. Wenn die einzelnen Komponenten der Liste $[k_1, \ldots, k_n]$ nicht relevant sind, schreibe ich auch kurz $tc{:}(w)k$.

E4: alle Typkonstruktoren haben eine einheitliche Stelligkeit, wobei mit Stelligkeit die Länge $\mid w \mid$ der Liste w gemeint ist. Es gilt also für alle $tc_1 : (w_1)k_1 \in TC$ und $tc_2 : (w_2)k_2 \in TC$:

$$tc_1 = tc_2 \Longrightarrow \mid w_1 \mid = \mid w_2 \mid$$

E5: die Teilklassenordnung $\leq$ ist nach unten vollständig (downward complete), d.h. für je zwei Klassen $k_1, k_2 \in K$ ist die Menge

$$k_1 \sqcap k_2 = \{k \mid k \leq k_1 \wedge k \leq k_2\}$$

entweder leer oder besitzt ein größtes Element.

E6: die Typsignatur Ω ist regulär (regular), d.h. für alle tc und w ist die Menge

$$R(tc, w) = \{k \mid \exists w'.\ w \leq w' \wedge tc : (w')k \in TC\}$$

entweder leer oder besitzt ein kleinstes Element. Die Ordnung $\leq$ auf den Klassen wird dabei komponentenweise auf Listen w über K fortgesetzt.

E7: die Typsignatur Ω ist coregulär (coregular), d.h. für alle tc und k ist die Menge

$$D(tc, k) = \{w \mid \exists k'.\ k' \leq k \wedge tc : (w)k' \in TC\}$$

entweder leer oder besitzt ein größtes Element.

E8: die Typsignatur Ω ist comonoton, d.h. für alle $tc : (w_1)k_1 \in TC$ und $tc : (w_2)k_2 \in TC$ gilt stets:

$$k_1 \leq k_2 \Longrightarrow w_1 \leq w_2$$

E9: die Typsignatur Ω enthält die minimale Typsignatur

$$\Omega_{Min} = (K_{Min}, \leq_{Min}, TC_{Min})$$

als Teilsignatur (i.Z. $\Omega_{Min} \subseteq \Omega$). Die Teilmengenbeziehung ist dabei komponentenweise gemeint. Die einzelnen Komponenten der Minimalsignatur sind wie folgt definiert:

$$\begin{array}{rcl} K_{Min} & = & \{top\} \\ \leq_{Min} & = & \{(top, top)\} \\ TC_{Min} & = & \{bool : top, ind : top,\ \Rightarrow : (top, top)top\} \end{array}$$

Die Eigenschaften *downward complete, regular und coregular* sind aus [Nip91] entnommen. Meine formale Behandlung der Typklassen orientiert sich stark an dieser Arbeit. Diese Eigenschaften sind wichtig für Beweise im Zusammenhang mit der automatischen Typinferenz. Die hier zusätzlich geforderte Eigenschaft der *Comonotonie* spielt eine wichtige Rolle in den Beweisen im Abschnitt 2.4 über die Semantik. Sie erleichtert allerdings auch einige Beweise über die Syntax von HOLC.

Nun können wir zuerst rohe Typterme und dann wohlgeformte Typterme über der Typsignatur Ω einführen.

Definition 2.3 *Rohe Typterme*

Die Menge der rohen Typterme RTT_Ω über der Typsignatur Ω ist definiert wie folgt:

$$\begin{array}{rcll} RTT & = & \alpha_k & \text{falls } k \in K \\ & | & (RTT_1 \ldots RTT_n tc) & \text{falls } tc:(k_1,\ldots,k_n)k \in TC \\ & & & \text{für beliebige } k_1,\ldots,k_n,k \in K \end{array}$$

Definition 2.4 *Wohlgeformte Typterme, Typklassenherleitung*

Die Menge der wohlgeformten Typterme T_Ω über der Typsignatur Ω ist durch das untenstehende Ableitungssystem definiert. Dabei bedeutet das Typterm-Urteil $\tau :: k$, daß τ ein wohlgeformter Typterm der Klasse k ist.

$$\text{(tvar)}\ \frac{}{\alpha_k :: k}\Big\{\ k \in K$$

$$\text{(tc)}\ \frac{\tau_1 :: k_1 \ldots \tau_n :: k_n}{(\tau_1 \ldots \tau_n tc) :: k}\Big\{\ tc:(k_1,\ldots,k_n)k \in TC$$

$$\text{(coerce)}\ \frac{\tau :: k}{\tau :: k'}\Big\{\ k \leq k'$$

Die Menge der wohlgeformten Terme $T_{\Omega,k}$ der Klasse k ist definiert als:

$$T_{\Omega,k} = \{\tau \mid \tau :: k\}$$

Die Menge aller wohlgeformten Terme T_Ω ist die Vereinigung:

$$T_\Omega = \bigcup_{k \in K} T_{\Omega,k}$$

Für Typterme, die den Standardkonstruktor $\Rightarrow$ enthalten, verwende ich eine bequemere Schreibweise. Statt $(\tau_1 \tau_2 \Rightarrow)$ schreibe ich $\tau_1 \Rightarrow \tau_2$, und Typterme wie $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3$ sollen wie üblich nach rechts, hier im Beispiel als $\tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$, geklammert sein.

Weiterhin soll im folgenden $TV(\tau)$ die Menge aller im Typterm τ vorkommenden Typvariablen bezeichnen. Die induktive Definition von $TV(\tau)$ ist offensichtlich.

Nachdem die wohlgeformten Typterme mittels des obigen Ableitungssystems definiert sind, gebe ich nun einen Algorithmus an, der zu einem gegebenen rohen Typterm eine Herleitung für dessen Wohlgeformtheit findet, falls dies überhaut möglich ist. Weiterhin hat die Herleitung die Eigenschaft, daß sie ein kleinstes Typterm-Urteil berechnet.

Definition 2.5 *Klasseninferenz $TTINF_\Omega$ für Typterme*

Der Algorithmus wird in ML-artiger Notation angegeben.

$$TTINF_\Omega(\alpha_k) = \alpha_k :: k$$

$$TTINF_\Omega((tc)) =$$
$$\begin{array}{ll} let & k' = mincodom(tc, []) \\ in & (\text{tc})\ \dfrac{}{(tc) :: k'} \end{array}$$

$$TTINF_\Omega((rt_1 \ldots rt_n tc)) =$$
$$\begin{array}{ll} let & \dfrac{D_1}{rt_1 :: k_1} = TTINF_\Omega(rt_1) \\ & \vdots \\ & \dfrac{D_n}{rt_n :: k_n} = TTINF_\Omega(rt_n) \\ & \\ & k' = mincodom(tc, [k_1, \ldots, k_n]) \\ & [k'_1, \ldots, k'_n] = maxdom(tc, k') \\ & \\ & \forall i.\ D'_i = (\text{coerce})\dfrac{\dfrac{D_i}{rt_i :: k_i}}{rt_i :: k'_i} \\ & \\ in & (\text{tc})\ \dfrac{D'_1 \ldots D'_n}{(rt_1 \ldots rt_n tc) :: k'} \end{array}$$

Die Hilfsfunktionen *mincodom* und *maxdom* sind definiert wie folgt:

$$mincodom(tc, w) = \begin{cases} k & \text{wenn } min(R(tc, w)) = \{k\} \\ fail & \text{sonst} \end{cases}$$

$$maxdom(tc, k) = \begin{cases} w & \text{wenn } max(D(tc, k)) = \{w\} \\ fail & \text{sonst} \end{cases}$$

Für den Algorithmus $TTINF_\Omega$ gilt folgender Satz:

Theorem 2.1 *Korrektheit von $TTINF_\Omega$*

Der Algorithmus $TTINF_\Omega$ ist korrekt bzgl. des Herleitungssystems für wohlgeformte Typterme.

Beweis: per Induktion über den Aufbau der Typterme unter Ausnutzung der Regularität und Comonotonie der Typsignatur Ω.

Weiterhin gilt folgender Satz über die Vollständigkeit von $TTINF_\Omega$.

Theorem 2.2 *Vollständigkeit von $TTINF_\Omega$*

Der Algorithmus $TTINF_\Omega$ ist vollständig bzgl. des Herleitungssystems für wohlgeformte Typterme. Insbesondere ist die von $TTINF_\Omega$ berechnete Klasse die kleinste Klasse, die über das Herleitungssystem für wohlgeformte Typterme berechnet werden kann.

$$\tau \in T_{\Omega,k'} \Longrightarrow TTINF_\Omega(\tau) = \frac{D}{\tau :: k} \wedge k \leq k'$$

Beweis: per Induktion über den Aufbau der Typterme unter Ausnutzung der Regularität und Comonotonie der Typsignatur Ω.

Definition 2.6 *Kleinste Klasse*

Die kleinste Klasse $lst_\Omega(\tau)$ des wohlgeformten Typterms $\tau \in T_\Omega$ ist definiert als

$$lst_\Omega(\tau) = k \quad \text{wenn } TTINF_\Omega(\tau) = \frac{D}{\tau :: k}$$

Wenn es aus dem Zusammenhang klar ist, lasse ich den Index Ω bei lst_Ω weg.

Als nächstes führe ich den Begriff der Typsubstitution ein.

Definition 2.7 *Typsubstitution*

Sei Ω eine Typsignatur. Eine Typsubstitution $\sigma_\Omega : \Xi \to T_\Omega$ ist eine totale, und daher unendliche Abbildung von Typvariablen aus Ξ in Typterme $\tau \in T_\Omega$. Typsubstitutionen sind zwar unendliche Abbildungen, unterscheiden sich aber nur an endlich vielen Stellen von der Identität. Daher werden sie wie endliche Abbildungen kodiert, und lediglich die Anwendung wird unterschiedlich definiert. Für die endliche Kodierung einer Typsubstitution sollen stets folgende Bedingungen gelten:

E1: $\sigma_\Omega \subseteq \Xi \times T_\Omega$ ist endlich.

E2: $(\alpha_k, \tau) \in \sigma_\Omega \Longrightarrow k \in K \wedge \tau \in T_{\Omega,k}$

E3: $(\alpha_k, \tau_1) \wedge (\alpha_k, \tau_2) \Longrightarrow \tau_1 = \tau_2$

Die Anwendung ist definiert als:

$$\sigma_\Omega(\alpha_k) = \begin{cases} \tau & \text{wenn } (\alpha_k, \tau) \in \sigma_\Omega \\ \alpha_k & \text{sonst} \end{cases}$$

Wenn es aus dem Zusammenhang klar ist, lasse ich den Index Ω bei σ_Ω weg.

Eine Substitution $\sigma_\Omega : \Xi \to T_\Omega$ kann eindeutig homomorph zu einer Substitution $\sigma^\star_\Omega : T_\Omega \to T_\Omega$ auf Typtermen erweitert werden. Die Fortsetzung wird offensichtlich wie folgt definiert:

Definition 2.8 *Typsubstitution auf Typtermen*

Wenn $\sigma_\Omega : \Xi \to T_\Omega$ eine Typsubstitution, dann ist die Fortsetzung auf Typterme $\sigma^\star_\Omega : T_\Omega \to T_\Omega$ definiert als:

$$\sigma^\star_\Omega(\tau) = \begin{cases} \sigma_\Omega(\alpha_k) & \text{falls } \tau = \alpha_k \\ (\sigma^\star_\Omega(\tau_1) \ldots \sigma^\star_\Omega(\tau_n)\, tc) & \text{falls } \tau = (\tau_1 \ldots \tau_n tc) \end{cases}$$

Statt $\sigma^\star_\Omega$ schreibe ich auch für die Fortsetzung kurz σ, wenn die Bedeutung aus dem Zusammenhang klar ist.

Für die Fortsetzung von Typsubstitutionen auf Typterme gilt folgendes Theorem:

Theorem 2.3

Wenn $\sigma_\Omega : \Xi \to T_\Omega$ eine Typsubstitution und $\tau \in T_{\Omega,k}$ ein Typterm der Klasse k, dann gilt stets:

$$\sigma^\star_\Omega(\tau) \in T_{\Omega,k} \quad \text{und} \quad lst_\Omega(\sigma^\star_\Omega(\tau)) \leq lst_\Omega(\tau)$$

Beweis: Induktion über den Aufbau der Typklassenherleitung $TTINF_\Omega(\tau)$. Der Beweis verwendet insbesondere die Regularität und Comonotonie der Signatur Ω.

Ein ähnliches Theorem gilt auch, wenn für einen Term über der Typsignatur Ω_1 die Klasseninferenz bzgl. einer erweiterten Typsignatur Ω_2 durchgeführt wird. Dieses Theorem ist wichtig für Beweise in Bezug auf Theorieerweiterungen.

Theorem 2.4

Wenn $\tau \in T_{\Omega_1,k}$ ein Typterm der Klasse k und $\Omega_1 \subseteq \Omega_2$, dann gilt stets:

$$\tau \in T_{\Omega_2,k} \text{ und } lst_{\Omega_2}(\tau) \leq lst_{\Omega_1}(\tau)$$

Beweis: Induktion über den Aufbau der Typklassenherleitung $TTINF_{\Omega_1}(\tau)$. Der Beweis verwendet wieder die Regularität und Comonotonie der Signatur Ω.

Für die Definition von Signaturen benötige ich noch das Konzept der *eingeschränkten Typsignatur* $\Omega \setminus k$. Diese entsteht aus der Typsignatur Ω, indem man alle Typkonstruktoren entfernt, die in ihren Aritäten eine Klasse k' verwenden, die von k abhängt, d.h. für die gilt $k' \leq k$.

Definition 2.9 *Eingeschränkte Typsignatur*

Sei $\Omega = (K, \leq, TC)$ eine Typsignatur und $k \in K$. Dann ist die eingeschränkte Typsignatur $\Omega \setminus k$ definiert als:

$$\Omega \setminus k = (K, \leq, TC \setminus k)$$

wobei

$$TC \setminus k = TC \setminus \{(tc, [c_1, \ldots, c_n], c) \mid \exists k' \in \{c_1, \ldots, c_n, c\}.\, k' \leq k\}$$

Man beachte, daß obige Definition nicht das Auftreten von Typvariablen einer Klasse $k' \leq k$ in Typtermen $\tau \in T_{\Omega\setminus k}$ verbietet. Diese Sonderfälle werden an den entsprechenden Stellen durch zusätzliche Bedingungen ausgeschlossen.

2.3.2 Signaturen und Terme

Aufbauend auf den Begriffen für Typsignaturen und Typterme können wir jetzt den Begriff der Signatur Σ einführen. Die Signaturen von HOLC unterscheiden sich von den Signaturen der normalen HOL-Logik insofern, als daß sie einen dritten Parameter haben, der es gestattet, gewisse Konstanten als *charakteristisch* für eine Klasse auszuzeichnen.

Definition 2.10 *Signatur*

Eine Signatur $\Sigma = (\Omega, C, KS)$ ist ein Tripel, das folgende Eigenschaften erfüllt:

E1: $\Omega = (K, \leq, TC)$ ist eine Typsignatur

E2: $C \subseteq \text{CID} \times T_\Omega$ ist eine endliche Abbildung von Konstantennamen in Typterme. Sie kodiert die Menge der getypten Konstanten der Signatur. Da C eine endliche Abbildung ist, hat jede Konstante einen eindeutigen Typ. Statt der Paarnotation (c, τ) verwende ich für die Konstanten auch die Schreibweise $c{:}\tau$.

E3: $KS = (KS_k)_{k \in K}$ ist eine K-indizierte Familie von charakteristischen Konstanten mit $\bigcup_{k \in K} KS_k \subseteq C$. Für jedes $k \in K$ heißt KS_k die Menge der charakteristischen Konstanten der Klasse k.

Um die in Abschnitt 2.2 vorgestellte informelle Semantik der Typklassen zu reflektieren, werden an die Typen der charakteristischen Konstanten in der Menge KS_k weitere Anforderungen gestellt:

E3.1: Für jede Klasse $k \in K$ gibt es eine Typvariable α_k, die sogenannte Klassenvariable der Klasse k, so daß für alle Paare $(c, \tau) \in KS_k$ gilt:

$$TV(\tau) = \{\alpha_k\}$$

In den Typen der charakteristischen Konstanten der Klasse k kommt also genau eine Typvariable vor, und zwar immer die Klassenvariable α_k.

E3.2: Weiterhin gilt für die Typen τ in den Paaren $(c, \tau) \in KS_k$, daß sie bis auf die Klassenvariable α_k unabhängig von der Klasse k sind, d.h.:

$$\tau \in T_{\Omega \setminus k}$$

Diese Bedingung ist wichtig für die schichtweise Definition der Semantik von Typklassen.

E4: Die Signatur Σ enthält die Minimalsignatur $\Sigma_{Min} = (\Omega_{Min}, C_{Min}, KS_{Min})$ als Teilsignatur (i.Z. $\Sigma_{Min} \subseteq \Sigma$). Die Teilmengenbeziehung ist wieder komponentenweise gemeint. Die einzelnen Komponenten von Σ_{Min} sind wie folgt definiert:

$$\begin{aligned}
\Omega_{Min} &= \text{siehe Definition von Typsignaturen} \\
C_{Min} &= \{ \\
&\quad \rightarrow :bool \Rightarrow bool \Rightarrow bool, \\
&\quad = :\alpha_{top} \Rightarrow \alpha_{top} \Rightarrow bool, \\
&\quad \varepsilon :(\alpha_{top} \Rightarrow bool) \Rightarrow bool \\
&\} \\
KS_{Min} &= \emptyset
\end{aligned}$$

Die Konstante $\rightarrow :bool \Rightarrow bool \Rightarrow bool$ steht für die logische Implikation, die polymorphe Konstante $= :\alpha_{top} \Rightarrow \alpha_{top} \Rightarrow bool$ für die Identität und

$\varepsilon : (\alpha_{top} \Rightarrow bool) \Rightarrow bool$ ist die polymorphe Konstante für den Hilbert-schen Auswahloperator.

Die Menge KS_{Min} ist leer, was bedeutet, daß es für die Klasse *top*, die in der minimalen Typsignatur Ω_{Min} als einzige Standardklasse enthalten ist, keine charakteristischen Konstanten gibt.

Die Bedingungen E3.1 und E3.2 schränken das Klassenkonzept stärker ein, als es zum Beispiel in der funktionalen Programmiersprache HASKELL [HJW92] der Fall ist. Der Grund für die Einschränkung liegt in der von mir verwendeten semantischen Modellierung von Typklassen. In meiner Modellierung werden die Instanzen einer charakteristischen Konstanten für einen speziellen Typ stets durch Elemente einer Menge des zugrundeliegenden Mengenuniversums interpretiert, sie dürfen also nicht von weiteren Typen abhängen. Dies vereinfacht die Konstruktion der Semantik von Klassen erheblich, weil damit polymorphe Funktionen nicht als sogenannte 'first-class citizens' behandelt werden müssen. Zudem besteht kein Grund, das Klassenkonzept komplizierter zu machen, als es unbedingt notwendig ist. Das von mir verwendete Konzept der Typklassen hat sich für die Entwicklung der Logik HOLCF als hinreichend mächtig erwiesen. Eine ausführliche Beschreibung der Modellierung von Klassen findet sich in Abschnitt 2.4.

Aufbauend auf der Definition für Signaturen werde ich jetzt *rohe Terme (raw terms, preterms)* und dann *wohlgeformte Terme* über einer Signatur Σ einführen. Ich werde dabei nicht nur explizit getypte Terme, sondern auch teilweise getypte Terme zulassen, wie es der Praxis entspricht. Allerdings wird jedem teilweise getypten Term t durch einen Typinferenzalgorithmus in eindeutiger Weise ein vollgetypter Term t' zugeordnet, die sogenannte *dekorierte Variante* des teilweise getypten Terms t. Für die Definitionen der Semantik und des Deduktionssystems werden nur vollgetypte, d.h. dekorierte Terme, verwendet. Die Möglichkeit, Typinformation in Termen wegzulassen, ist eine reine Schreiberleichterung für den Benutzer, und die fehlende Typinformation wird immer vom Typinferenzalgorithmus ergänzt. Man kann also genau die Typinformation weglassen, von der man weiß, daß sie vom Typinferenzalgorithmus ergänzt wird. Explizit getypt wird nur dann, wenn es die Lesbarkeit der Terme erhöht oder wenn die errechnete Typinformation allgemeiner wäre, als die erwünschte.

Definition 2.11 *Rohe Terme RT_Σ über Signatur Σ*

Sei eine Signatur $\Sigma = (\Omega, C, KS)$ und die am Anfang des Abschnitts eingeführte Menge Ψ von Termvariablen gegeben. Die Menge der rohen Typterme RT_Σ über der Signatur Σ ist dann durch folgende induktive Definition gegeben:

RT	::=	c	falls $c : e \in C$ für beliebiges $e \in T_\Omega$
		$c : \tau$	falls $c : e \in C$ für beliebiges $e \in T_\Omega$ und $\tau \in T_\Omega$
		x	falls $x \in \Psi$
		$x : \tau$	falls $x \in \Psi$ und $\tau \in T_\Omega$
		$(RT_1\, RT_2)$	
		$(\lambda x.\, RT_1)$	falls $x \in \Psi$
		$(\lambda x : \tau.\, RT_1)$	falls $x \in \Psi$ und $\tau \in T_\Omega$

Die Menge der Typvariablen in einem (rohen) Term t wird wie schon bei Typtermen mit $TV(t)$ bezeichnet und kann auf offensichtliche Weise aus den in t vorkommenden Typtermen

τ berechnet werden. Aus dem Kontext ist stets klar, ob bei Verwendung von $TV(t)$ ein Term oder ein Typterm gemeint ist.

Die Menge der frei vorkommenden Termvariablen in einem (rohen) Term t wird mit $FV(t)$ bezeichnet, die Menge der gebundenen Variablen mit $BV(t)$. Die Definitionen für $FV(t)$ bzw. $BV(t)$ erfolgt auf die für λ-Terme übliche Weise und wird hier nicht wiederholt.

Für die induktive Definition der wohlgeformten Terme verwende ich wie üblich [Mit90, Mit93, Nip91] einen Kalkül (term formation calculus). Da Termvariablen durch die λ-Abstraktion gebunden und dadurch verschattet werden können, benötigt man einen Kontext für die aktuelle Typisierung der frei vorkommenden Termvariablen, der dynamisch wachsen und schrumpfen kann.

Definition 2.12 *Typkontext Γ über Signatur Σ*

Ein Typkontext Γ über der Signatur $\Sigma = (\Omega, C, KS)$ und den Termvariablen aus Ψ ist eine endliche Abbildung $\Gamma : \Psi \to T_\Omega$, die die aktuellen Typannahmen für die freien Variablen kodiert. Für die einzelnen Paare der endlichen Abbildung schreibe ich $x:\tau$.

Definition 2.13 *Wohlgeformte Terme T_Σ über Signatur Σ*

Die Menge der wohlgeformten Terme T_Σ über der Signatur $\Sigma = (\Omega, C, KS)$ ist eine Teilmenge von RT_Σ und wird mit Hilfe einer induktiv definierten Relation

$$\rhd \subseteq (\Psi \to T_\Omega) \times RT_\Sigma \times T_\Omega$$

charakterisiert.

Die Aussage $(\Gamma, t, \tau) \in \rhd$ heißt dabei:

> Wenn man für die freien Termvariablen im Term t annimmt, daß sie gemäß dem Typkontext $\Gamma = \{x_1:\tau_1, \ldots, x_n:\tau_n\}$ getypt sind, dann hat der Term t den Typ τ

Die induktive Definition der Relation $\rhd$ ist durch das folgende Ableitungssystem (term formation calculus) gegeben. Statt $(\Gamma, t, \tau) \in \rhd$ schreibe ich, der üblichen Konvention folgend, $\Gamma \rhd t :: \tau$.

$$(\text{const})\frac{}{\emptyset \rhd c :: \tau}\left\{\begin{array}{l}\text{falls } c:e \in C \text{ mit } TV(e) = \{\alpha_1, \ldots, \alpha_n\}, n \geq 0\\ \text{und es gibt Substitution } \sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}\\ \text{so daß } \sigma(e) = \tau\end{array}\right.$$

$$(\text{typed\_const})\ \frac{}{\emptyset \rhd c:\tau :: \tau}\left\{\text{Bedingung wie bei (const)}\right.$$

Bemerkung: Hier wird der Schemacharakter von $c:e \in C$ deutlich. Jede Instanz $\sigma(e)$ des Typschemas $e \in T_\Omega$ mittels einer Typsubstitution σ führt zu einem wohlgeformten Term. Wenn speziell $TV(e) = \emptyset$, dann ist $c:e$ keine polymorphe Konstante. Bei der Formulierung der Substitution σ habe ich statt der ausführlichen Schreibweise α_{ik_i}

kurz α_i geschrieben, da in diesem Zusammenhang die Klasse k_i, zu der α_{ik_i} gehört, uninteressant ist.

$$\text{(var)}\ \frac{}{\{x:\tau\} \rhd\ x::\tau}$$

$$\text{(typed\_var)}\ \frac{}{\{x:\tau\} \rhd\ x:\tau::\tau}$$

$$\text{(weak)}\ \frac{\Gamma_1 \rhd\ t::\tau}{\Gamma_1 \cup \Gamma_2 \rhd\ t::\tau}$$

Bemerkung: Der Kontext Γ_2 muß natürlich so geartet sein, daß $\Gamma_1 \cup \Gamma_2$ wieder ein Kontext ist (rechtseindeutig). Das gilt auch für alle anderen Regeln, bei denen Kontexte vereinigt werden.

$$\text{(app)}\ \frac{\Gamma_1 \rhd\ t_1::\tau_1 \Rightarrow \tau_2 \quad \Gamma_2 \rhd\ t_2::\tau_1}{\Gamma_1 \cup \Gamma_2 \rhd\ (t_1 t_2)::\tau_2}$$

$$\text{(abs)}\ \frac{\Gamma \cup \{x:\tau_1\} \rhd\ t::\tau_2}{\Gamma \rhd\ (\lambda x.\ t)::\tau_1 \Rightarrow \tau_2}\Bigg\{ \text{falls } x:\tau_1 \notin \Gamma$$

$$\text{(typed\_abs)}\ \frac{\Gamma \cup \{x:\tau_1\} \rhd\ t::\tau_2}{\Gamma \rhd\ (\lambda x:\tau_1.\ t)::\tau_1 \Rightarrow \tau_2}\Bigg\{ \text{falls } x:\tau_1 \notin \Gamma$$

Aufbauend auf diesem Kalkül wird die Menge $T_{\Sigma,\tau}$ der wohlgeformten Terme vom Typ τ jetzt definiert als:

$$T_{\Sigma,\tau} = \{t \mid \text{es gibt } \Gamma \text{ und } \tau \text{ so daß } \Gamma \rhd\ t::\tau\}$$

Die Menge aller wohlgeformten Terme T_Σ ist die Vereinigung über alle $\tau \in T_\Omega$:

$$T_\Sigma = \bigcup_{\tau \in T_\Omega} T_{\Sigma,\tau}$$

Im folgenden muß ich öfters Typsubstitutionen $\sigma:\Xi \to T_\Omega$ auf die Typtermanteile in Kontexten, Termen bzw. ganzen Herleitungen anwenden. Die diesbezüglichen Erweiterungen sind in offensichtlicher Weise über den Aufbau der jeweiligen Struktur definiert und seien hier nicht besonders aufgeführt. Auch die Erweiterung der Substitution σ bezeichne ich stets wieder mit σ.

Der Unifikationsalgorithmus $UNIFY_\Omega$ und der Typinferenzalgorithmus $TINF_\Sigma$, die ich beide gleich beschreiben werde, benötigen während der Arbeit immer wieder frische Typvariablen, die noch niemals benutzt wurden. Will man die Bereitstellung von garantiert neuen Typvariablen funktional in die Algorithmen einbauen, so ergibt sich daraus ein erheblicher und nicht trivialer Kodierungsaufwand. Eine solche rein funktionale Lösung wird in [NP93] vorgestellt. Für meine knappe Behandlung der Unifikation und der Typinferenz in diesem Abschnitt werde

ich mich jedoch, wie auch viele andere Autoren, auf die Annahme beschränken, daß zu jedem Zeitpunkt während der Abarbeitung der Algorithmen für jede Klasse $k \in K$ eine frische Typvariable existiert. Diese werde ich stets mit new_k bezeichnen. Eine solche Annahme ist durchaus realistisch, da in konkreten Implementierungen neue Variablen durch eine Funktion $New(k)$ generiert werden können, die an einen, für den Benutzer verbotenen, Standardvariablennamen den Wert eines lokalen Zählers anhängt. Dieser Zähler wird bei jedem Aufruf inkrementiert. Die Funktion *New* arbeitet dabei natürlich mit einem Seiteneffekt, was zu einer nicht mehr ganz funktionalen Implementierung führt.

Der Unifikationsalgorithmus $UNIFY_\Omega$, den ich hier präsentiere, ist bis auf eine minimale Änderung identisch mit [Nip91]. Der Unterschied besteht lediglich darin, daß mein Unifikationsalgorithmus im Aufruf neben den Unifikationspaaren noch eine zusätzliche Menge von schreibgeschützten Typvariablen als Argument erhält. Die Substitution, die $UNIFY_\Omega$ als allgemeinsten Unifikator berechnet, bildet diese Typvariablen stets identisch in sich selbst ab. Über diesen Mechanismus wird die vom Benutzer explizit angegebene Typinformation in den Termen unverändert übernommen und somit respektiert.

Die Aufgabe des Unifikationsalgorithmus $UNIFY_\Omega$ läßt sich wie folgt beschreiben: Sei R eine endliche Menge von schreibgeschützten Typvariablen und $E = [(\tau_1, \tau_1'), \ldots, (\tau_n, \tau_n')]$ eine Liste von Unifikationspaaren. Der Algorithmus $UNIFY_\Omega(R, E)$ soll, falls dies überhaupt möglich ist, eine (allgemeinste) Typsubstitution σ berechnen, so daß für alle Paare (τ_i, τ_i') in der Liste E gilt: $\sigma(\tau_i) = \sigma(\tau_i')$. Weiterhin sollen alle Typvariablen $\alpha_k \in R$ identisch abgebildet werden, d.h. $\sigma(\alpha_k) = \alpha_k$.

Definition 2.14 *Ordnungssortierte Unifikation $UNIFY_\Omega(R, E)$ nach [Nip91]*

Sei $\Omega = (K, \leq, TC)$ eine Typsignatur, $R \subseteq \Xi$ eine endliche Menge von schreibgeschützten Typvariablen (read only variables) und $E = [(\tau_1, \tau_1'), \ldots, (\tau_n, \tau_n')]$ eine Liste von Unifikationspaaren. Der Algorithmus $UNIFY_\Omega(R, E)$ wird über ein Termersetzungssystem $\Rrightarrow_{\Omega,R}$ definiert. In der nachfolgenden Beschreibung steht $e :: l$ für das Anfügen des Elements e an die Liste l, und $l_1 @ l_2$ steht für die Konkatenation der Listen l_1 und l_2:

$$UNIFY_\Omega(R, E) = \begin{cases} \sigma & \text{falls } \langle E, \emptyset \rangle \Rrightarrow_{\Omega,R} \langle [], \sigma \rangle \\ \mathit{fail} & \text{sonst} \end{cases}$$

Das Termersetzungssystem $\langle E_1, \sigma_1 \rangle \Rrightarrow_{\Omega,R} \langle E_2, \sigma_2 \rangle$ ist definiert wie folgt:

(vv) $\langle (\alpha_k = \alpha_k) :: E, \sigma \rangle \Rrightarrow_{\Omega,R} \langle E, \sigma \rangle$

(tv) $\langle (\tau = \alpha_k) :: E, \sigma \rangle \Rrightarrow_{\Omega,R} \langle (\alpha_k = \tau) :: E, \sigma \rangle$
falls $\tau \notin \Xi \setminus R$ und $\alpha_k \in \Xi \setminus R$

(vt) $\langle (\alpha_k = \tau) :: E, \sigma \rangle \Rrightarrow_{\Omega,R} \langle \bar{\sigma}(E), \bar{\sigma} \circ \sigma \rangle$
falls $\alpha_k \in \Xi \setminus R$ und $\alpha_k \notin TV(\tau)$
wobei $\hat{\sigma} = weak_{\Omega,R}(\tau, k)$
und $\bar{\sigma} = \hat{\sigma} \cup \{\alpha_k \mapsto \hat{\sigma}(\tau)\}$

(tt) $\langle (\tau_1 \ldots \tau_n tc = \tau_1' \ldots \tau_n' tc) :: E, \sigma \rangle \Rrightarrow_{\Omega,R} \langle [\tau_1 = \tau_1', \ldots, \tau_n = \tau_n'] @ E, \sigma \rangle$

Im Fall (vv) sind die Variablen gleich und es ist nichts zu tun, im Fall (tv) werden die Positionen getauscht, falls τ keine schreibbare Typvariable ist und α_k schreibbar ist. Der Fall (tt) verlagert das Unifikationsproblem auf Teilterme. Der Fall (vt) ist der interessanteste. Hier wird eine Substitution für α_k erzeugt, falls α_k schreibbar ist und nicht in τ vorkommt (occurs check). Vorher müssen jedoch die schreibbaren Typvariablen in τ so durch kleinere neue Typvariablen ersetzt werden, damit ein Term der Klasse k entsteht. Diese Abschwächung wird durch die Substitution $weak_{\Omega,R}(\tau,k)$ vorgenommen. Die Funktion $weak_{\Omega,R}$ berechnet, falls möglich, eine Substitution für schreibbare Typvariablen in τ, so daß $weak_{\Omega,R}(\tau,k) \in T_{\Omega,k}$. Die Funktion $weak_{\Omega,R}$ wird ihrerseits über eine Hilfsfunktion $w_{\Omega,R}$ berechnet:

$$weak_{\Omega,R}(\tau,k) = w_{\Omega,R}([(\tau,k)],\emptyset)$$

Die Hilfsfunktion $w_{\Omega,R}$ ist definiert wie folgt:

$$w_{\Omega,R}([],\theta) = \theta$$

$$w_{\Omega,R}((\beta_k,k') :: ps,\theta) = \begin{cases} w_{\Omega,R}(ps,\theta) & \text{falls } k \leq k' \\ w_{\Omega,R}(\theta'(ps),\theta' \circ \theta) & \text{falls } k \not\leq k' \text{ und } \beta_k \in \Xi \setminus R \\ & \text{und } glb(k,k') \text{ definiert} \\ & \text{Dabei ist } \theta' = \beta_k \mapsto new_{glb(k,k')} \\ \text{fail} & \text{sonst} \end{cases}$$

$$w_{\Omega,R}((\tau_1 \ldots \tau_n tc,k) :: ps,\theta) = \begin{cases} w_{\Omega,R}([(\tau_1,k_1),\ldots,(\tau_n,k_n)]@ps,\theta) & \\ \qquad \text{falls } maxdom(tc,k) \text{ definiert} & \\ \qquad \text{und } maxdom(tc,k) = [k_1,\ldots,k_n] & \\ \text{fail} \qquad \text{sonst} & \end{cases}$$

Die Hilfsfunktion glb ist dabei definiert als:

$$glb(k_1,k_2) = \begin{cases} k & \text{falls } max(k_1 \sqcap k_2) = \{k\} \\ \text{fail} & \text{sonst} \end{cases}$$

Theorem 2.5 *Eigenschaften von $UNIFY_\Omega$*

Der Algorithmus $UNIFY_\Omega$ hat folgende Eigenschaften

1. $UNIFY_\Omega$ ist korrekt, d.h. wenn $UNIFY_\Omega(R,E) = \sigma$, dann gilt $\sigma(\tau_i) = \sigma(\tau_i')$ für alle Paare (τ_i,τ_i') in E.
2. $UNIFY_\Omega$ berechnet den allgemeinsten Unifikator, d.h. wenn es einen Unifikator $\bar{\sigma}$ für E gibt, dann gibt es stets eine Substitution σ', so daß $\bar{\sigma} = \sigma' \circ UNIFY_\Omega(R,E)$.
3. $UNIFY_\Omega$ respektiert die schreibgeschützten Typvariablen, d.h. wenn $UNIFY_\Omega(R,E) = \sigma$, dann gilt $\sigma(\alpha) = \alpha$ für alle $\alpha \in R$.

Beweis: Für die ersten beiden Behauptungen siehe [Nip91] und weiterführende Literatur. Die letzte Behauptung folgt unmittelbar aus der Definition für $UNIFY_\Omega$.

Mit Hilfe des Unifkationsalgorithmus $UNIFY_\Omega$ kann jetzt leicht ein Typinferenzalgorithmus $TINF_\Sigma$ für wohlgeformte Terme formuliert werden. Im Gegensatz zu den meisten Typinferenzalgorithmen berechnet der hier vorgestellte Algorithmus nicht nur den allgemeinsten Typ eines Terms, sondern liefert eine komplette Typherleitung für die Wohlgeformtheit des Terms. Diese Typherleitung wird dann benutzt, um die dekorierte (vollgetypte) Variante des wohlgeformten Terms zu berechnen, die ihrerseits zur Definition der Semantik herangezogen wird.

Der im folgenden präsentierte Algorithmus $TINF_\Sigma$ berechnet, falls dies überhaupt möglich ist, eine Typherleitung für die Wohlgeformtheit des Terms t. Insbesondere wird dabei jegliche explizite Typinformation im Term t respektiert, d.h. der Algorithmus hält sich an die vom Benutzer vorgegebenen Typen und vesucht nicht, eine allgemeinere Typinformation zu berechnen. Die Definition des Algorithmus ist grob an [Mit93] angelehnt.

Definition 2.15 *Typinferenzalgorithmus* $TINF_\Sigma$

Der Algorithmus $TINF_\Sigma$ benutzt eine Hilfsfunktion $WF_{\Sigma,R}$, die als zusätzliches Argument die Menge der in t vorkommenden Typvariablen erhält. Diese Typvariablen werden als schreibgeschüzte Variablen behandelt, und somit wird die explizit in t vorkommende Typinformation respektiert.

$$TINF_\Sigma(t) = WF_{\Sigma,TV(t)}(t)$$

Die Hilfsfunktion $WF_{\Sigma,R}$ wird induktiv über den Aufbau des rohen Terms t definiert.

(var)
$$WF_{\Sigma,R}(x) = \{x:\mu\} \rhd x::\mu \quad \text{where } \mu = new_{top}$$

(typed_var)
$$WF_{\Sigma,R}(x:\tau) = \{x:\tau\} \rhd x:\tau::\tau$$

(const)
$$WF_{\Sigma,R}(c) =$$
$$\begin{array}{ll} \mathit{if} & c:e \in C \text{ mit } TV(e) = \{\alpha 1_{k1}, \ldots, \alpha n_{kn}\}, n \geq 0 \\ \mathit{then} & \\ \quad \mathit{let} & \sigma = \{\alpha 1_{k1} \mapsto new_{k1}, \ldots, \alpha n_{kn} \mapsto new_{kn}\} \\ \quad \mathit{in} & \emptyset \rhd c::\sigma(e) \\ \mathit{else} & \mathit{fail} \end{array}$$

(typed_const)
$$WF_{\Sigma,R}(c:\tau) =$$
$$\begin{array}{ll} \mathit{if} & c:e \in C \text{ mit } TV(e) = \{\alpha 1_{k1}, \ldots, \alpha n_{kn}\}, n \geq 0 \\ \mathit{then} & \\ \quad \mathit{let} & \bar{\sigma} = \{\alpha 1_{k1} \mapsto new_{k1}, \ldots, \alpha n_{kn} \mapsto new_{kn}\} \end{array}$$

$$\sigma' = UNIFY_{\Omega}(R, [(\bar{\sigma}(e), \tau)])$$

in $\emptyset \rhd c : \tau :: \tau$

else *fail*

(abs)

$WF_{\Sigma,R}(\lambda x.t) =$

let $\dfrac{D}{\Gamma \rhd t :: \rho} = WF_{\Sigma,R}(t)$

in

if $(x, \tau) \in \Gamma$ for some τ

then $\dfrac{WF_{\Sigma,R}(t)}{\Gamma \setminus \{(x,\tau)\} \rhd (\lambda x.t) :: \tau \Rightarrow \rho}$

else

let $\mu = new_{top}$

in $\dfrac{\dfrac{WF_{\Sigma,R}(t)}{\Gamma \cup \{(x,\mu)\} \rhd t :: \rho}}{\Gamma \rhd (\lambda x.t) :: \mu \Rightarrow \rho}$

(typed_abs)

$WF_{\Sigma,R}(\lambda x : \tau.t) =$

let $\dfrac{D}{\Gamma \rhd t :: \rho} = WF_{\Sigma,R}(t)$

in

if $(x, \tau_1) \in \Gamma$ for some τ_1

then

let $\sigma = UNIFY_{\Omega}(R, [(\tau_1, \tau)])$

in $\sigma\left(\dfrac{WF_{\Sigma,R}(t)}{\Gamma \setminus \{(x,\tau_1)\} \rhd (\lambda x : \tau_1.t) :: \tau_1 \Rightarrow \rho}\right)$

else $\dfrac{\dfrac{WF_{\Sigma,R}(t)}{\Gamma \cup \{(x,\tau)\} \rhd t :: \rho}}{\Gamma \rhd (\lambda x : \tau.t) :: \tau \Rightarrow \rho}$

(app)

$WF_{\Sigma,R}(t_1 t_2) =$

let $\dfrac{D_1}{\Gamma_1 \rhd t_1 :: \tau_1} = WF_{\Sigma,R}(t_1)$

$\dfrac{D_2}{\Gamma_2 \rhd t_2 :: \tau_2} = WF_{\Sigma,R}(t_2)$

$\mu = new_{top}$

$$E = [(\rho, \rho') \mid \exists x.(x, \rho) \in \Gamma_1 \wedge (x, \rho') \in \Gamma_2]$$
$$\sigma = UNIFY_\Omega(R, (\tau_1, \tau_2 \Rightarrow \mu) :: E)$$

$$in \quad \frac{\sigma(\,WF_{\Sigma,R}(t_1)) \qquad \sigma(\,WF_{\Sigma,R}(t_2))}{\sigma(\Gamma_1) \cup \sigma(\Gamma_2) \rhd \;\; \sigma(t_1 t_2) :: \sigma(\mu)}$$

Bemerkungen zum Algorithmus $TINF_\Sigma$:

(const): Hier wird der Schemacharakter des Typs e der Konstanten c deutlich. Es wird jedesmal eine neue Instanz gebildet, wobei die Typvariablen in e durch frische Variablen der selben Klasse ersetzt werden. Die Substitution σ ist die in der Definition 2.13 geforderte Instanz.

(typed_const): Wie auch bei der Regel (const) werden hier zuerst die Schemavariablen mittels $\bar{\sigma}$ umbenannt. Dann wird geprüft, ob der explizite Typ τ eine korrekte Instanz des Typschemas e ist. Falls die Unifikation eine Substitution σ' berechnen kann, so gilt offensichtlich $\sigma'(\bar{\sigma}(e)) = \sigma'(\tau) = \tau$, da $TV(\tau) \subseteq R$. Die geforderte Substitution ist also $\sigma' \circ \bar{\sigma}$.

(typed_abs): Die im *if*-Fall angewendete Substitution σ erzeugt sicher die gewünschte Herleitung für $\lambda x\!:\!\tau.t$, da $\sigma(\tau_1) = \sigma(\tau) = \tau$ wegen $TV(\tau) \subseteq R$.

(app): Der Kontext $\sigma(\Gamma_1) \cup \sigma(\Gamma_2)$ ist sicher wohlgeformt, da evtl. Kollisionen durch die per Unifikation berechnete Substitution σ eliminiert werden. Weiterhin gilt $\sigma(\tau_1) = \sigma(\tau_2 \Rightarrow \mu) = (\sigma(\tau 2) \Rightarrow \sigma(\mu))$.

Der eben vorgestellte Typinferenzalgorithmus $TINF_\Sigma$ ist korrekt und vollständig. Es lassen sich folgende Theoreme formulieren.

Theorem 2.6 *Korrektheit von $TINF_\Sigma$*

Wenn der Algorithmus $TINF_\Sigma$, angewendet auf den (rohen) Term t, nicht mit Fehler abbricht, dann erzeugt er eine Typherleitung D für den Term t gemäß der Definition 2.13, die mit $\Gamma \rhd t :: \tau$ endet. Insbesondere gilt dann:

$$t \in T_{\Sigma,\tau}$$
$$FV(t) = FV(\Gamma)$$
$$TV(t) \subseteq TV(\Gamma) \cup TV(\tau)$$

Beweis: Induktion über den Aufbau des Terms t und Verwendung der Eigenschaften des Unifikationsalgorithmus $UNIFY_\Omega$.

Theorem 2.7 *Vollständigkeit von $TINF_\Sigma$*

Wenn es für einen (rohen) Term t eine Typherleitung D' gibt, die mit $\Gamma' \rhd t :: \tau'$ endet, dann findet auch der Algorithmus $TINF_\Sigma$ eine Herleitung D, die mit $\Gamma \rhd t :: \tau$ endet. Insbesondere gibt es auch eine Typsubstitution σ, so daß $\sigma(\tau) = \tau'$ und $\sigma(\Gamma) \subseteq \Gamma'$. Der Algorithmus $TINF_\Sigma$ berechnet also eine allgemeinste Herleitung und einen allgemeinsten Typ (principal type).

Beweis: Ohne Beschränkung der Allgemeinheit kann man annehmen, daß in der Herleitung D' die Regel (weak) nur angewendet wird, falls dies unbedingt notwendig ist, das heißt so spät wie möglich. Eine diesbezügliche Normalisierung der Herleitungen ist immer möglich. Induktion über den Aufbau dieser normalisierten Herleitung D' liefert dann das gewünschte Resultat. Dabei werden natürlich wieder die Eigenschaften des Unifikationsalgorithmus $UNIFY_\Omega$ eingesetzt.

Die obigen Theoreme geben Anlaß zu folgender Definition.

Definition 2.16 *Generischer Typ, generischer Kontext, generische Herleitung*

Wenn der Algorithmus $TINF_\Sigma$, angewendet auf den (rohen) Term t, nicht mit Fehler abbricht, sondern eine Typherleitung D erzeugt, die mit $\Gamma \rhd t::\tau$ endet, so heißt D die generische Herleitung von t (i.Z. $GD_\Sigma(t)$). Der Kontext Γ heißt generischer Kontext von t (i.Z. $GC_\Sigma(t)$), und der Typ τ heißt generischer Typ von t (i.Z. $GT_\Sigma(t)$). Statt 'generisch' verwende ich bisweilen auch 'allgemeinst'(principal). Wenn es aus dem Zusammenhang klar ist, dann lasse ich den Index Σ weg.

Der Algorithmus $TINF_\Sigma$ berechnet, falls möglich, für einen Term t eine generische Herleitung D. Während der Typinferenz wird die vollständige Typinformation für alle Teilterme von t berechnet. Diese Information, zu der insbesondere alle Instanzen von Konstanten und Typen von freien und gebundenen Variablen zählen, wird allerdings nicht im Term abgespeichert, sondern bis auf die Typen der freien Variablen, die durch den generischen Kontext Γ erhältlich sind, wieder vergessen. In der Praxis, etwa im Isabelle-System, wird dies natürlich nicht so gehandhabt, sondern dort wird während der Typinferenz schrittweise eine vollgetypte Variante, die sogenannte dekorierte Variante, des Terms t erzeugt, in der alle Vorkommen von Konstanten und freien bzw. gebundenen Variablen explizit getypt sind.

Der Einfachheit halber habe ich die Dekoration der Terme nicht in den Typinferenzalgorithmus eingearbeitet, sondern schalte die Dekorationsphase als zweite Phase nach. Meine Vorgehensweise unterscheidet sich von der in der Literatur [Mit93, Jon91] üblichen dadurch, daß mein Typinferenzalgorithmus schon teilweise getypte Terme als Eingabe akzeptiert und diese partielle Typinformation auch respektiert. Bei der herkömmlichen Formulierung des Typinferenzproblems [Mit93] geht man davon aus, daß die Eingabe t keinerlei Typinformation enthält und erzeugt dann während der Typinferenz eine Herleitung für eine vollgetypte Variante t', so daß nach Entfernen (erase function) sämtlicher expliziter Typinformation in t' gerade wieder t entsteht. Diese Charakterisierung der Typinferenzaufgabe ist bei Behandlung von teilweise getypten Termen natürlich nicht mehr möglich.

Bei den Definitionen für Theorien, der Semantik und des Deduktionssystems werde ich mich ausschließlich auf vollgetypte Terme beschränken, da in diesem Zusammenhang die Verwendung von nur teilweise getypten Termen nicht sinnvoll erscheint. Es folgen jetzt die Definitionen für die Dekoration von Termen und die Teilmenge der vollgetypten Terme.

Definition 2.17 *Dekoration von Termen*

Sei $t \in T_\Sigma$ ein wohlgeformter Term. Seine explizit getypte Variante (dekorierte Variante) $dec(t)$ ist der Term, der am Ende der dekorierten Herleitung $DEC(TINF_\Sigma(t))$ steht. Der Algorithmus DEC für die Erzeugung der dekorierten Herleitung ist über den Aufbau der Herleitung $D = TINF_\Sigma(t)$ definiert.

Fall (const) mit $D = \emptyset \rhd c :: \tau$
$DEC(D) = \emptyset \rhd c : \tau :: \tau$

Fall (typed_const) mit $D = \emptyset \rhd c : \tau :: \tau$
$DEC(D) = D$

Fall (var) mit $D = \{(x, \tau)\} \rhd x :: \tau$
$DEC(D) = \{(x, \tau)\} \rhd x : \tau :: \tau$

Fall (typed_var) mit $D = \{(x, \tau)\} \rhd x : \tau :: \tau$
$DEC(D) = D$

Fall (weak) mit $D = \dfrac{D_1}{\Gamma \rhd t :: \tau}$

$DEC(D) =$

$$\textit{let} \quad \frac{D_{10}}{\Gamma_1 \rhd t' :: \tau} = DEC(D_1)$$

$$\textit{in} \quad \frac{DEC(D_1)}{\Gamma \rhd t' :: \tau}$$

Fall (app) mit $D = \dfrac{D_1 \qquad D_2}{\Gamma \rhd (t_1 t_2) :: \tau_2}$

$DEC(D) =$

$$\textit{let} \quad \frac{D_{10}}{\Gamma_1 \rhd t_1' :: \tau_1 \Rightarrow \tau_2} = DEC(D_1)$$

$$\frac{D_{20}}{\Gamma_2 \rhd t_2' :: \tau_1} = DEC(D_2)$$

$$\textit{in} \quad \frac{DEC(D_1) \qquad DEC(D_1)}{\Gamma \rhd (t_1' t_2') :: \tau_2}$$

Fall (abs) mit $D = \dfrac{D_1}{\Gamma \rhd (\lambda x . t) :: \tau_1 \Rightarrow \tau_2}$

$DEC(D) =$

$$let \quad \frac{D_{10}}{\Gamma \cup \{x:\tau_1\} \rhd \ t'::\tau_2} = DEC(D_1)$$

$$in \quad \frac{DEC(D_1)}{\Gamma \rhd \ (\lambda x:\tau_1.t')::\tau_1 \Rightarrow \tau_2}$$

Fall (typed_abs) mit $\quad D = \dfrac{D_1}{\Gamma \rhd \ (\lambda x:\tau_1.t)::\tau_1 \Rightarrow \tau_2}$

$DEC(D) =$

$$let \quad \frac{D_{10}}{\Gamma \cup \{x:\tau_1\} \rhd \ t'::\tau_2} = DEC(D_1)$$

$$in \quad \frac{DEC(D_1)}{\Gamma \rhd \ (\lambda x:\tau_1.t')::\tau_1 \Rightarrow \tau_2}$$

Die dekorierte Variante $dec(t)$ des teilweise getypten Terms t wird definiert als:

$dec(t) =$

$$let \quad \frac{D}{\Gamma \rhd \ t'::\tau} = DEC(TINF_\Sigma(t))$$

$$in \quad t'$$

Nun kann die Teilmenge DT_Σ der dekorierten Terme definiert werden.

Definition 2.18 *Dekorierte Terme DT_Σ*

Die Teilmenge der dekorierten Terme $DT_{\Sigma,\tau}$ vom Typ τ ist definiert als:

$$DT_{\Sigma,\tau} = \{t \mid t \in T_{\Sigma,\tau} \wedge dec(t) = t\}$$

Die Menge aller dekorierten Terme ergibt sich durch die Vereinigung über alle τ:

$$DT_\Sigma = \bigcup_{\tau \in T_\Omega} DT_{\Sigma,\tau}$$

Als nächstes führe ich den Begriff der Formel ein. Wie in HOL üblich, wird die Menge der Formeln durch alle Terme vom Typ *bool* gebildet. Da in Theorien aber nur vollgetypte Terme eine Rolle spielen, ist die Definition der Formeln auf vollgetypte Terme eingeschränkt.

Definition 2.19 *Formeln*

Die Menge der Formeln $FORM_\Sigma$ über der Signatur Σ ist definiert als:

$$FORM_\Sigma = DT_{\Sigma,bool}$$

Um in den folgenden Abschnitten die Terme lesbarer zu machen, führe ich nun noch kurz eine bequemere Schreibweise für die speziellen Konstanten der Standardsignatur Σ_{Min} ein. Sie entspricht der Schreibweise im Isabelle-System, wo für Konstanten beliebige Mixfix-Schreibweisen vereinbart werden können, die dann vom System automatisch in die interne und offizielle Form überführt werden. In der rechten Spalte sind die Prioritäten für die Auflösung der Mixfix-Schreibweise angegeben. Durch die Zuordnung von Prioritäten lassen sich viele Klammern sparen. Die technischen Details zu dieser Art von Prioritätsgrammatiken können in [Pau94] nachgelesen werden. In der nachstehenden Tabelle lasse ich wieder die explizite Typinformation weg, die Terme seien aber trotzdem mittels *dec* vollgetypt.

	Extern	Intern	Mixfix-Priorität
Implikation	$t_1 \rightarrow t_2$	$((\rightarrow t_1)t_2)$	Rechts 25
Identität	$t_1 = t_2$	$((= t_1)t_2)$	Links 50
Hilbert-Deskriptor	$\varepsilon x.\ P$	$(\varepsilon(\lambda x.\ P))$	Binder 10

Jetzt ist fast alles eingeführt, was zur Definition des Begriffs der Theorie in HOLC benötigt wird. Eine letzte vorbereitende Definition ist noch notwendig, um die Signatur einzuschränken, über der die charakteristischen Axiome einer Klasse gebildet werden. Die charakteristischen Axiome der Klasse $k \in K$ sollen bis auf die charakteristischen Konstanten der Klasse k unabhängig von der Klasse k und all ihren Teilklassen sein, damit eine schichtweise Einführung von Klassen möglich ist.

Definition 2.20 *Eingeschränkte Signatur* $\Sigma \setminus k$

Sei $\Sigma = (\Omega, C, KS)$ eine Signatur und $k \neq top$. Die Komponenten der bzgl. der Klasse $k \in K$ eingeschränkten Signatur $\Sigma \setminus k = (\Omega \setminus k, C \setminus k, KS \setminus k)$ sind definiert wie folgt:

$\Omega \setminus k$ ist bekannt aus Definition 2.9

$$C \setminus k = \{c{:}\tau \in C \mid \tau \in T_{\Omega\setminus k} \wedge ((TV(\tau) \cap \bigcup_{k' \leq k} \Xi_{k'} \neq \emptyset) \Rightarrow c{:}\tau \in KS_k)\}$$

$$KS \setminus k = (KS_{k'})_{k' \in K \wedge k' \not\leq k}$$

Bemerkung:
Mit obiger Definition gilt insbesondere, daß $KS_k \subseteq C \setminus k$ und daß KS_k ein Element der Familie $KS \setminus k$ ist. Die etwas umständliche Bedingung bei der Mengenkomprehension ist notwendig, weil $\tau \in T_{\Omega\setminus k}$ nicht ausschließt, daß τ Typvariablen einer Klasse k' mit $k' \leq k$ enthält.

2.3.3 Theorien

In diesem Abschnitt wird der Begriff der Theorie eingeführt.

Definition 2.21 *Theorie*

Eine Σ-Theorie $Th = (\Sigma, Ax, KAx)$ ist ein Tripel mit folgenden Eigenschaften:

E1: $\Sigma = (\Omega, C, KS)$ ist eine Signatur.

E2: $Ax \subseteq FORM_\Sigma$ ist eine endliche Teilmenge von Formeln, den sogenannten Axiomen der Theorie.

Bemerkung: Die Axiome dürfen freie Variablen enthalten. Diesem Umstand wird bei der Definition von Modellen und im Deduktionssystem Rechnung getragen.

E3: $KAx = (KAx_k)_{k \in K}$ ist eine K-indizierte Familie von Axiomen. Dabei gilt:

E3.1: $\bigcup_{k \in K} KAx_k \subseteq Ax$. Die Mengen KAx_k dienen also nur dazu, um gewisse Axiome aus Ax als charakteristisch für eine Klasse auszuzeichnen. Für jedes $k \in K$ heißt KAx_k die Menge der charakteristischen Axiome der Klasse k. Diese Axiome müssen in jedem Typ der Klasse k gelten (siehe Semantik).

E3.2: Die Signatur der charakteristischen Axiome KAx_k wird noch speziell eingeschränkt. Für alle Klassen $k \in K$ gibt es, wie schon in Definition 2.10, jeweils eine ausgezeichnete Typvariable $\alpha_k \in \Xi_k$, die sogenannte Klassenvariable, so daß gilt:

$$\forall ax \in KAx_k.\, TV(ax) = \{\alpha_k\} \wedge ax \in FORM_{\Sigma \setminus k}$$

Die Bedingung E3 garantiert, daß die charakteristischen Axiome der Klasse k bis auf die charakteristischen Konstanten der Klasse unabhängig von der Klasse k sind, und daß genau eine Typvariable der Klasse k, nämlich die Klassenvariable α_k, in den Typen vorkommt.

Damit sind jetzt alle Begriffe für die Syntax von HOLC eingeführt. Im nächsten Abschnitt werde ich eine darauf aufbauende Semantik für HOLC angeben.

2.4 Semantik von HOLC

In diesem Abschnitt werde ich die Semantik von HOLC definieren. Die Semantik von HOLC basiert auf der Semantik von HOL, wie sie in [GM93] definiert wurde. Um die syntaktischen Konzepte bzgl. der Typklassen zu reflektieren, waren jedoch einige Erweiterungen notwendig. Die grundlegenden Ideen der Semantik für HOLC habe ich ja bereits in Abschnitt 2.2 dargelegt. Hier erfolgt nun die formale Definition.

2.4.1 Modelle für Typsignaturen

Zur Interpretation für Typterme und Terme benötigt man ein Universum von Mengen. Dieses Universum, das die Grundlage für alle Interpretationen ist, entspricht bis auf die zusätzliche Abschlußeigenschaft (E4) genau dem in [GM93] eingeführten. Es wird durch folgende Definition charakterisiert.

Definition 2.22 *Präuniversum PU*

Ein Präuniversum PU ist eine Menge von Mengen[19], den sogenannten Trägermengen, die folgende Abschlußeigenschaften erfüllt.

[19] das Universum PU muß wieder die Eigenschaften einer Menge und nicht nur die einer Klasse im mengentheoretischen Sinn haben.

E1: Jedes Element in PU ist eine nichtleere Menge.

E2: Wenn $X \in PU$ und $Y \subseteq X$ mit $Y \neq \emptyset$, dann ist auch Y in PU.

E3: Wenn $X \in PU$ und $Y \in PU$, dann ist auch das kartesische Produkt beider Mengen $X \times Y$ in PU.

E4: Wenn $(X_i)_{i \in \omega}$ eine Familie nichtleerer Mengen mit $X_i \in PU$ für alle $i \in \omega$, dann ist auch das abzählbar unendliche Produkt $(\Pi_{i \in \omega} X_i)$ in PU.

E5: Wenn $X \in PU$, dann ist auch die Potenzmenge $\wp(X) = \{Y \mid Y \subseteq X\}$ in PU.

E6: Das Präuniversum PU enthält eine unendliche Menge Inf.

E7: Es gibt eine Auswahlfunktion $ch \in \Pi_{X \in PU} X$, die jeder Menge $X \in PU$ ein Element $ch(X) \in X$ zuordnet.

Mit Hilfe einer axiomatischen Mengentheorie, etwa der Zermelo-Fraenkel Theorie mit Auswahlaxiom (ZFC), kann die Existenz solcher Präuniversen gezeigt werden. In ZFC lassen sich dann auch noch die untenstehenden zusätzlichen Eigenschaften von Präuniversen als Theoreme aus E1–E7 ableiten.

E8: Wenn $X \in PU$ und $Y \in PU$, dann ist auch die Menge aller totalen Funktionen Y^X in PU. Y^X bezeichnet dabei wie üblich den vollen Funktionenraum über X und Y.

E9: Es gibt eine zweielementige Menge $Two \in PU$ mit $Two = \{\mathbb{T}, \mathbb{F}\}$.

Bis auf die Bedingung E4 entsprechen die Abschlußeigenschaften genau denjenigen, die auch in [GM93] gefordert werden. Die Forderung E4 wurde von mir aufgenommen, um bei der Entwicklung von LCF in HOLC auch semantische Bereiche für die Lösung von rekursiven Bereichsgleichungen zur Verfügung zu haben. Hier wird ja bekannterweise bei der Konstruktion des Inversen-Limes eine Teilmenge des unendlichen Kreuzprodukts mit der entsprechenden punktweisen Ordnung als Teil des Co-Limes benötigt. Die Eigenschaft E4 garantiert, daß der Träger für den Bereich des unendlichen Kreuzprodukts zur Verfügung steht. Mehr dazu später im Kapitel 5.

In der informellen Erklärung der Semantik von Typklassen in Abschnitt 2.2 habe ich vereinfachend angenommen, daß die Interpretation für einen Typ τ in der Klasse k eine mathematische Struktur X ist, die neben einem Träger car_X noch die Interpretationen für die charakteristischen Konstanten der Klasse k enthält. Für die formale Definition der Semantik ist es jedoch notwendig, die Interpretationen für die charakteristischen Konstanten der Klasse k und all ihren Oberklassen gemeinsam in geeigneter Weise in der Struktur X zu kodieren. Ein entscheidender Punkt dabei ist, daß die Definition der *fgt*-Funktion auch möglich sein muß, wenn die Klasse k mehrere unmittelbare Oberklassen k_1 bis k_n hat. Es muß also in irgendeiner Weise in der Struktur X kodiert werden, welche Information vergessen werden darf, wenn man aus der Struktur X in der Klasse k durch Anwendung von *fgt* eine Struktur $\widehat{X}$ in einer der Klassen k_i machen will.

Diese Idee der Strukturierung der Interpretationen für charakteristische Konstanten wird in den folgenden Definitionen formalisiert. Zuerst werden die unmittelbaren Oberklassen, d.h. die direkten Nachfolger bzgl. der Klassenordnung charakterisiert.

Definition 2.23 *Direkter Nachfolger*

Sei $\Omega = (K, \leq, TC)$ eine Typsignatur und $k_1 \in K$. Eine Klasse $k_2 \in K$ heißt direkter Nachfolger von k_1 (i.Z. $k_1 \rightsquigarrow k_2$), wenn

$$k_1 < k_2 \wedge \neg(\exists k \in K.\ k_1 < k \wedge k < k_2)$$

Dabei ist die Ordnung $<$ der irreflexive Anteil der Klassenordnung $\leq$ mit der offensichtlichen Vereinbarung:

$$k_1 < k_2 \Longleftrightarrow k_1 \leq k_2 \wedge k_1 \neq k_2$$

Da K eine endliche Menge von Klassenidentifikatoren ist und $\leq$ per Definition eine partielle Ordnung auf K ist, hat die gerade definierte Relation $\rightsquigarrow$ die Eigenschaften einer fundierten Ordnung. Dies wird gleich durch folgende rekursive Definition ausgenützt, die die Kodierung für die Interpretationen von charakteristischen Konstanten festlegt.

Definition 2.24 *Konstantenstruktur für die Klasse k*

Sei $\Omega = (K, \leq, TC)$ eine Typsignatur und $k \in K$. Eine Konstantenstruktur *stru* für die Klasse k ist ein Paar, dessen Eigenschaften durch folgende rekursive Definition gegeben sind.

Basisfall $k = top$**:**
In diesem Fall ist *stru* das Paar $(const, strufam)$ mit $const = \emptyset$ und $strufam = \emptyset$.

Induktionsschritt $k \neq top$**:**
In diesem Fall ist *stru* ein Paar $(const, strufam)$, so daß folgende Eigenschaften gelten:

B1: *const* ist eine endliche Abbildung von Konstantenidentifikatoren aus CID in die Menge $\bigcup_{X \in PU} X$.

B2: *strufam* ist eine endliche Abbildung der Menge $\{k' \mid k' \in K \wedge k \rightsquigarrow k'\}$ in Konstantenstrukturen, so daß das Bild $strufam(k')$ jeweils eine Konstantenstruktur der Klasse k' ist.

Eine Konstantenstruktur für die Klasse *top* ist also stets trivial, da es keine charakteristischen Konstanten zu interpretieren gibt und auch keine Nachfolger da sind.

Wenn die Klasse k nicht *top* ist, dann sind über die Funktion *const* die Interpretationen der charakteristischen Konstanten der Klasse k erreichbar. In der obigen Definition wird nur verlangt, daß *const* irgendwelche Konstanten in irgendwelche Elemente eines beliebigen Trägers abbildet. Sobald wir eine Signatur Σ haben, wird diese Bedingung natürlich dahingehend verschärft, daß genau die charakteristischen Konstanten $c{:}\tau \in KS_k$ abgebildet werden und daß die Bilder typkorrekt bzgl. der Typen τ sind.

Die Bedingung B1 der obigen Definition zeigt deutlich die Modellierung der Instanzen von charakteristischen Konstanten. Die Instanz muß ein Element einer Menge $X \in PU$ sein. Diese Forderung motiviert nachträglich die syntaktischen Bedingungen E3.1 und E3.2 aus der Definition 2.10 für Signaturen.

Die Abbildung *strufam* kodiert eine mit den direkten Nachfolgern der Klasse k indizierte Familie von Konstantenstrukturen, so daß unter dem Index k' gerade immer eine Konstantenstruktur der Klasse k' erreicht werden kann. Über *strufam* sind also die Konstantenstrukturen aller direkten Nachfolger zugreifbar. Wenn $k \rightsquigarrow k'$, dann sind in der Konstantenstruktur $strufam(k')$ unter der dortigen Komponente *const* die Interpretation der charakteristischen Konstanten der Klasse k' kodiert, und unter der Komponente *strufam* wieder die Konstantenstrukturen der direkten Nachfolger von k'. Dies geht so weiter, bis man bei Konstantenstrukturen der Klasse *top* anlangt, die nicht weiter strukturiert sind.

Wenn man eine Konstantenstruktur der Klasse k_1 gegeben hat, so kann man immer eine Konstantenstruktur der Klasse k_2 daraus erzeugen, wenn $k_1 \leq k_2$. Man muß dazu nur irgendeinen Pfad $[k_1, \ldots, k_2]$ von k_1 nach k_2 berechnen und entlang dieses Pfades über sukzessive Anwendung der Funktionen *strufam* die irrelevanten Anteile vergessen. Hier steckt im Prinzip schon die Idee der *fgt*-Funktion verborgen. Damit man aber *fgt* als *Funktion* definieren kann, muß es egal sein, welchen Pfad von k_1 nach k_2 man dabei auswählt. Hieraus ergibt sich eine Art Konfluenzbedingung, die durch die folgenden zwei Definitionen charakterisiert ist.

Definition 2.25 *Pfad*

Seien $c_1, c_2 \in K$ mit $c_1 \leq c_2$. Eine nichtleere Liste von Klassen $P = [k_1, \ldots, k_n]$ heißt Pfad von c_1 nach c_2, wenn:

1. $P = [k] \Longrightarrow c_1 = k = c_2$
2. $P = (k_1 :: k_2 :: ps) \Longrightarrow c_1 = k_1 \wedge k_1 \rightsquigarrow k_2 \wedge (k_2 :: ps)$ ist Pfad von k_2 nach c_2

Definition 2.26 *Konsistente Konstantenstruktur*

Eine Konstantenstruktur *stru* für eine Klasse $c_1 \in K$ heißt konsistent, wenn für eine beliebige Klasse $c_2 \in K$ mit $c_1 \leq c_2$ und beliebige Pfade p_1, p_2 von c_1 nach c_2 gilt:

$$fgt\_path(p_1)(stru) \quad = \quad fgt\_path(p_2)(stru)$$

wobei

$$\begin{aligned} fgt\_path([k])(stru) &= stru \\ fgt\_path(k_1 :: k_2 :: ps)(const, strufam) &= fgt\_path(k_2 :: ps)(strufam(k_2)) \end{aligned}$$

Für konsistente Konstantenstrukturen der Klasse $c_1 \in K$ ist es also gleichgültig, welchen Pfad man wählt, um per *fgt_path* eine Konstantenstruktur der Klasse $c_2 \in K$ mit $c_1 \leq c_2$ zu erzeugen. Aus der Definition für konsistente Konstantenstrukturen folgt unmittelbar, daß das Paar $(\emptyset, \emptyset)$ eine konsistente Konstantenstrutkur der Klasse *top* ist. Im allgemeinen gilt das folgende Theorem, dessen Gültigkeit direkt aus der Definition für konsistente Konstantenstrukturen folgt:

Theorem 2.8

Wenn *stru* eine konsistente Konstantenstruktur der Klasse $c_1 \in K$ und p ein Pfad von c_1 nach $c_2 \in K$, dann ist $fgt\_path(p)(stru)$ eine konsistente Konstantenstruktur der Klasse c_2.

Für konsistente Konstantenstrukturen kann man jetzt eine einfachere Vergißfunktion *fgt* einführen.

Definition 2.27 *fgt für konsistente Konstantenstrukturen*

Seien $c_1, c_2 \in K$ mit $c_1 \leq c_2$ und *stru* eine konsistente Konstantenstruktur der Klasse c_1. Die Funktion fgt_{c_1,c_2}, die aus *stru* eine konsistente Konstantentstruktur der Klasse c_2 erzeugt, ist dann definiert wie folgt:

$$fgt_{c_1,c_2}(stru) \quad = \quad fgt\_path(p)(stru)$$

wobei p ein beliebiger Pfad von c_1 nach c_2 ist.

Für je zwei Klasse $c_1, c_2 \in K$ mit $c_1 \leq c_2$ ist hiermit also eine Vergißfunktion fgt_{c_1,c_2} definiert. Für diese Funktionen gilt außerdem folgende Konfluenzeigenschaft:

Theorem 2.9 *Konfluenz von fgt*

Seien $c_1, c_2, c_3 \in K$ mit $c_1 \leq c_2 \leq c_3$ und *stru* eine konsistente Konstantenstruktur der Klasse c_1. Dann gilt stets:

$$fgt_{c_1,c_3}(stru) \quad = \quad fgt_{c_2,c_3}(fgt_{c_1,c_2}(stru))$$

Beweis: Die Eigenschaft folgt unmittelbar aus der Konsistenz der Konstantenstruktur *stru*.

Mit den obigen Definitionen haben wir die Kodierung für die Interpretationen der charakteristischen Konstanten festgelegt. Jetzt können wir die Elemente beschreiben, die die Interpretationen für Typen sein werden. Aus der informellen Einführung in Abschnitt 2.2 wissen wir, daß Typen der Klasse k als Elemente in einem speziellen Universum für die Klasse k interpretiert werden. Diese Elemente, die ich als *Strukturen der Klasse k* bezeichnen werde, bestehen aus einer Trägermenge des Präuniversums PU und einer konsistenten Konstantenstruktur der Klasse k. Bisweilen werde ich die Strukturen der Klasse k auch als Algebren bezeichnen, obwohl diese Bezeichnung nicht ganz zutreffend ist, da bei Algebren die Operationen nicht so hierarchisch strukturiert sind, wie es hier bei den Konstantenstrukturen der Fall ist.

Definition 2.28 *Strukturen der Klasse k*

Sei $k \in K$. Eine Struktur (Algebra) der Klasse k ist ein Paar $X = (car, stru)$, so daß $car \in PU$ eine Trägermenge und $stru = (const, strufam)$ eine konsistente Konstantenstruktur der Klasse k ist.

Zur Schreiberleichterung werden folgende Selektoren auf Strukturen vereinbart, die der Einfachheit halber den selben Namen wie die selektierte Komponente haben sollen. Sei $X = (car, stru)$ mit $stru = (const, strufam)$, dann wird definiert:

$$\begin{aligned} car(X) &= car \\ stru(X) &= stru \\ const(X) &= const \\ strufam(X) &= strufam \end{aligned}$$

Mit der obigen Definition folgt sofort, daß für alle $car \in PU$ das Tupel $(car, (\emptyset, \emptyset))$ eine Struktur der Klasse *top* ist.

Die Vergißfunktion für konsistente Konstantenstrukturen läßt sich jetzt in offensichtlicher Weise auf Strukturen fortsetzten.

Definition 2.29 *fgt für Strukturen*

Seien $c_1, c_2 \in K$ Klassen mit $c_1 \leq c_2$ und sei X eine Struktur der Klasse c_1. Dann ist die Vergißfunktion fgt_{c_1,c_2} definiert wie folgt:

$$fgt_{c_1,c_2}(X) \quad = \quad (car(X), fgt_{c_1,c_2}(stru(X)))$$

Aus dieser Definition folgt unmittelbar das nachstehende Theorem, dessen Beweise wieder offensichtlich ist.

Theorem 2.10 *Carrier-Lemma*

Wenn X eine Struktur der Klasse c_1 und $c_1 \leq c_2$, dann ist $fgt_{c_1,c_2}(X)$ eine Struktur der Klasse c_2. Insbesondere gilt:

$$car(X) \quad = \quad car(fgt_{c_1,c_2}(X))$$

Alle Strukturen, die über *fgt* 'in Beziehung stehen', haben also die gleichen Trägermengen. Diese Eigenschaft wird später bei der Interpretation von überladenen Typkonstruktoren wichtig sein. Aufgrund der Fortsetzung der Funktion *fgt* auf Strukturen gilt jetzt natürlich auch folgender Satz:

Theorem 2.11 *Konfluenz von fgt für Strukturen*

Seien $c_1, c_2, c_3 \in K$ mit $c_1 \leq c_2 \leq c_3$ und X eine Struktur der Klasse c_1. Dann gilt stets:

$$fgt_{c_1,c_3}(X) \quad = \quad fgt_{c_2,c_3}(fgt_{c_1,c_2}(X))$$

Beweis: folgt unmittelbar aus der Definition

Jetzt sind alle Begriffe eingeführt, die zur Definition von Modellen für Typsignaturen Ω benötigt werden.

Definition 2.30 *Modelle für Typsignaturen* Ω

Ein Modell $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$ für eine Typsignatur $\Omega = (K, \leq, TC)$ ist ein Paar mit folgenden Eigenschaften:

E1: $\mathcal{K} = (k^{\mathcal{TM}})_{k \in K}$ ist eine K-indizierte Familie nichtleerer Mengen $k^{\mathcal{TM}}$, so daß jede dieser Mengen $k^{\mathcal{TM}}$ nur Strukturen der Klasse k enthält. Die Mengen $k^{\mathcal{TM}}$ werde ich im folgenden als Universen der Klasse k bezeichnen.

E2: Das Universum $top^{\mathcal{TM}}$ enthält für jeden Träger $car \in PU$ eine Struktur $X = (car, (\emptyset, \emptyset))$, d.h.:

$$top^{\mathcal{TM}} = \{(car, (\emptyset, \emptyset)) \mid car \in PU\}$$

E3: $\mathcal{K}$ ist abgeschlossen bzgl. der Abbildung fgt, d.h. für alle Klassen $k_1, k_2 \in K$ mit $k_1 \leq k_2$ gilt:

$$\forall X \in k_1^{\mathcal{TM}}.\, fgt_{k_1,k_2}(X) \in k_2^{\mathcal{TM}}$$

E4: $\mathcal{TC}$ ist ein Menge von Interpretationen für Typkonstruktoren, so daß es für jeden Typkonstruktor

$$tc{:}(k_1, \ldots, k_n)k \in TC$$

eine totale Abbildung

$$tc^{\mathcal{TM}}_{(k_1,\ldots,k_n)k} \in \mathcal{TC}$$

gibt mit

$$tc^{\mathcal{TM}}_{(k_1,\ldots,k_n)k} {:} k_1^{\mathcal{TM}} \times \ldots \times k_n^{\mathcal{TM}} \rightarrow k^{\mathcal{TM}}$$

Wenn $tc{:}k \in TC$ ein nullstelliger Typkonstruktor ist, dann soll $tc_k^{\mathcal{TM}} \in \mathcal{TC}$ wie üblich ein Element in $k^{\mathcal{TM}}$ sein. Da die Abbildungen in $\mathcal{TM}$ total sein sollen, ergeben sich daraus entsprechende Forderungen bzgl. der Abgeschlossenheit der Universen $k^{\mathcal{TM}}$.

E5: Für die Interpretationen von überladenen Typkonstruktoren, d.h. Typkonstruktoren mit mehreren Aritäten, wird noch eine weitere Eigenschaft verlangt. Wenn $tc{:}(c_1, \ldots, c_n)c \in TC$ und $tc{:}(k_1, \ldots, k_n)k \in TC$ sowie $c \leq k$, dann gilt wegen der Comonotonie der Typsignatur Ω auch $[c_1, \ldots, c_n] \leq [k_1, \ldots, k_n]$. In diesem Fall muß für die Interpretationen der beiden Typkonstruktoren gelten:

$$\forall X_1 \in c_1^{\mathcal{TM}} \ldots X_n \in c_n^{\mathcal{TM}}.$$

$$fgt_{c,k}(tc^{\mathcal{TM}}_{(c_1,\ldots,c_n)c}(X_1, \ldots, X_n)) \quad = \quad tc^{\mathcal{TM}}_{(k_1,\ldots,k_n)k}(fgt_{c_1,k_1}(X_1), \ldots, fgt_{c_n,k_n}(X_n))$$

E6: Die Typkonstruktoren aus der Minimalsignatur Ω_{Min} müssen standardmäßig interpretiert werden. Die Eigenschaften aus der Definition 2.22 für das Präuniversum PU garantieren die Wohldefiniertheit der folgenden Vereinbarungen:

$$bool_{top}^{\mathcal{TM}} = (Two, (\emptyset, \emptyset)) \qquad ind_{top}^{\mathcal{TM}} = (Inf, (\emptyset, \emptyset))$$

Die Mengen Two und Inf sind aus der Definition 2.22 bekannt. Der Konstruktor $\Rightarrow^{\mathcal{TM}}_{(top,top)top}$ ist die Abbildung

$$(X, Y) \mapsto (car(Y)^{car(X)}, (\emptyset, \emptyset))$$

Die Strukturen X und Y werden also abgebildet auf eine Struktur, die als Träger die Menge $car(Y)^{car(X)}$ hat, die wie schon in Definition 2.22 den vollen Funktionsraum über Urbildmenge $car(X)$ und Bildmenge $car(Y)$ bezeichnet. In diesem Zusammenhang verwende ich im folgenden die Infix-Schreibweise

$$X \Rightarrow^{\mathcal{TM}}_{(top,top)top} Y \quad \text{bzw. kurz} \quad X \Rightarrow Y$$

wenn aus dem Zusammenhang klar ist, daß die Interpretation des Konstruktors $\Rightarrow$ und nicht die Syntax gemeint ist. Entsprechende Schreiberleichterungen werde ich mir auch bei andern Typkonstruktoren gönnen.

Im folgenden möchte ich die obige Definition kurz kommentieren. Eigenschaft E1 besagt, daß es für jede syntaktische Klasse $k \in K$ ein eigenes Universum bestehend aus lauter Strukturen der Klasse k gibt. Diese Universen sind aber nicht völlig unabhängig voneinander, sondern durch die jeweiligen Vergißfunktionen *fgt* miteinander gekoppelt. Die Eigenschaft E3 garantiert, daß jede Struktur der Klasse k durch schrittweise (und bzgl. *fgt* konsistente) Erweiterung der Konstantenstrukturen aus einer Struktur der Klasse *top* entsteht.

Aus der Eigenschaft E2 ist ersichtlich, daß das Universum $top^{\mathcal{TM}}$ maximal groß ist, d.h. jede nur erdenkliche Struktur der Klasse *top* ist per Definition in $top^{\mathcal{TM}}$ enthalten. Diese Maximalitätsforderung wird später in geeigneter Weise auch auf die anderen Universen ausgedehnt, wobei jedoch die Maximalität nur bzgl. solcher Strukturen gefordert wird, die die charakteristischen Konstanten typkorrekt interpretieren und in denen zudem die Klassenaxiome gelten. Bis auf die zusätzliche triviale Konstantenstruktur $(\emptyset, \emptyset)$, die aus technischen Gründen angebracht wird, stimmen die Strukturen der Klasse *top* mit den Mengen im Präuniversum PU überein. Das Universum PU ist bekannterweise die Basis für die Semantik von Typen in der Logik HOL [GM93]. Hier sieht man die 'konservative' Erweiterung der Semantik von HOL zu einer Semantik von HOLC.

Mit der Eigenschaft E4 wird ausgedrückt, daß Typkonstruktoren gemäß ihrer Arität als Abbildungen zwischen den Universen interpretiert werden. Die Eigenschaft E5 ist zentral für die Semantik von überladenen Typkonstruktoren. Hinter dieser Eigenschaft verbirgt sich die Idee, daß ein Typkonstruktor zwar immer die gleiche Trägermenge konstruiert, daß aber abhängig von seiner Arität die zusätzliche Information in den Konstantenstrukturen unterschiedlich gut ausgenützt wird. Je mehr Information in den Argumenten steckt, desto mehr Strukturinformation steckt auch im Ergebnis. Die Eigenschaft E5 garantiert eine gewisse Monotonie der Verwertung der Strukturinformation in den Konstantenstrukturen.

Nachdem nun Modelle für Typsignaturen definiert sind, können wir die Interpretation von Typtermen $\tau \in T_\Omega$ definieren. Da in Typtermen Typvariablen vorkommen können, benötigen wir zunächst eine Belegung für Typvariablen.

Definition 2.31 *Belegung für Typvariablen*

Sei $\Omega = (K, \leq, TC)$ eine Typsignatur und $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$ ein Modell für Ω.

Eine Belegung für Typvariablen (Typvariablenbelegung) ist eine K-indizierte Familie $\nu = (\nu_k)_{k \in K}$ von Abbildung $\nu_k : \Xi_k \to k^{\mathcal{TM}}$.

Bei der Anwendung der einzelnen Abbildungen schreibe ich vereinfacht immer ν statt ν_k, da aus dem Kontext immer klar ist, welches ν_k gemeint ist.

Mit Hilfe dieser Typvariablenbelegungen und den von $TTINF_\Omega$ erzeugten Typherleitungen kann jetzt einfach die Interpretation von Typtermen definiert werden.

Definition 2.32 *Interpretation von Typtermen*

Sei $\Omega = (K, \leq, TC)$ eine Typsignatur, $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$ ein Modell für Ω und ν eine Typvariablenbelegung. Die Interpretation $\mathcal{TM}[\![\tau]\!]^\Omega_\nu$ des Typterms $\tau \in T_\Omega$ im Modell $\mathcal{TM}$ unter Typvariablenbelegung ν wird induktiv über den Aufbau des Typterms τ

unter Verwendung der Typherleitung $TTINF_\Omega(\tau)$ definiert.

Fall $\tau = \alpha_k$

mit $\quad TTINF_\Omega(\tau) = \alpha_k :: k$

dann $\quad \mathcal{TM}[\![\tau]\!]^\Omega_\nu = \nu(\alpha_k)$

Fall $\tau = (tc)$

mit $\quad TTINF_\Omega(\tau) = \dfrac{}{(tc) :: k'}$

dann $\quad \mathcal{TM}[\![\tau]\!]^\Omega_\nu = tc^{\mathcal{TM}}_{k'}$

Fall $\tau = (\tau_1 \ldots \tau_n tc)$

mit $\quad TTINF_\Omega(\tau) = \dfrac{(\text{coerce})\dfrac{TTINF_\Omega(\tau_1)}{\tau_1 :: k'_1} \ldots (\text{coerce})\dfrac{TTINF_\Omega(\tau_n)}{\tau_n :: k'_n}}{(\tau_1 \ldots \tau_n tc) :: k'}$

wobei $\quad \forall i.\ TTINF_\Omega(\tau_i) = \dfrac{D_i}{\tau_i :: k_i} \quad \text{und } k_i \leq k'_i$

dann $\quad \mathcal{TM}[\![\tau]\!]^\Omega_\nu = tc^{\mathcal{TM}}_{(k'_1,\ldots,k'_n)k'}($
$fgt_{k_1,k'_1}(\mathcal{TM}[\![\tau_1]\!]^\Omega_\nu), \ldots,$
$fgt_{k_n,k'_n}(\mathcal{TM}[\![\tau_n]\!]^\Omega_\nu))$

Für die oben definierte Interpretation von Typtermen gilt folgende Eigenschaft.

Theorem 2.12 *Klassenkorrektheit (class soundness)*

Für alle Belegungen ν gilt:

$$\mathcal{TM}[\![\tau]\!]^\Omega_\nu \in (lst_\Omega(\tau))^{\mathcal{TM}}$$

Beweis: Einfache Induktion über den Aufbau von τ.

Wie in Environment-Modellen so üblich, benötigen wir auch hier den Begriff der Belegungsänderung. Auf der Ebene der Typen kann man diese so definieren.

Definition 2.33 *Belegungsänderung für Typvariablen*

Sei ν eine Typvariablenbelegung bzgl. Typsignatur Ω und Typmodell $\mathcal{TM}$. Die Änderung der Belegung ν an der Stelle $\alpha \in \Xi_k$ durch die Struktur X der Klasse $k \in K$ ist definiert wie folgt:

$$\nu[X/\alpha](\beta) = \begin{cases} X & \text{falls } \alpha = \beta \\ \nu(\beta) & \text{sonst} \end{cases}$$

In der Definition für die Abänderung von Belegungen habe ich nicht verlangt, daß die Struktur X, mit der die Belegung an der Stelle α_k abgeändert wird, ein Element aus dem Universum $k^{\mathcal{TM}}$ ist. Der Grund für diese allgemeinere Definition wird später in Definition 2.35 für typkorrekte Strukturen der Klasse k deutlich werden. Wenn aber $X \notin k^{\mathcal{TM}}$, dann ist $\nu[X/\alpha_k]$ keine echte Typvariablenbelegung mehr, da für diese verlangt wird, daß $\nu_k : \Xi_k \to k^{\mathcal{TM}}$. Versucht man, einen Term $\tau \in T_\Omega$ unter dieser Pseudo-Belegung zu interpretieren, dann kann es passieren, daß die Interpretation nicht wohldefiniert ist, wenn ein Typkonstruktor auf α_k angewendet wird, der auch ein Argument aus der Klasse k erwartet. Die Typkonstruktoren müssen ja nur auf den Strukturen in den Universen $k^{\mathcal{TM}}$ definiert sein.

Wenn allerdings der Typterm τ ein Term über der eingeschränkten Typsignatur $T_{\Omega\setminus k}$ ist, und für X noch zusätzliche Bedingungen gelten, dann kann man τ auch unter der Pseudo-Belegung $\nu[X/\alpha_k]$ interpretieren. Diese Aussage wird durch den folgenden Satz konkretisiert.

Theorem 2.13 *Interpretation unter Pseudo-Belegung $\nu[X/\alpha_k]$*

Sei ν eine Typvariablenbelegung und X eine Struktur der Klasse k, die nicht notwendigerweise Element des Universums $k^{\mathcal{TM}}$ ist. Sei weiterhin $\tau \in T_{\Omega\setminus k}$ und gelte für alle direkten Nachfolger $k' \in \{\bar{k} \mid k \rightsquigarrow \bar{k}\}$ von k daß:

$$fgt_{k,k'}(X) \in k'^{\mathcal{TM}}$$

Dann ist die Interpretation des Typterms τ auch unter der Pseudo-Belegung $\nu[X/\alpha_k]$ wohldefiniert und es gilt:

$$\begin{array}{lcl} \tau = \alpha_k & \Longrightarrow & \mathcal{TM}[\![\tau]\!]^{\Omega\setminus k}_{\nu[X/\alpha_k]} = X \\ \tau \neq \alpha_k & \Longrightarrow & \mathcal{TM}[\![\tau]\!]^{\Omega\setminus k}_{\nu[X/\alpha_k]} \in (lst_{\Omega\setminus k}(\tau))^{\mathcal{TM}} \end{array}$$

Beweis: Der Beweis der ersten Aussage ist trivial. Für die zweite Aussage benutzt man, daß τ entweder eine andere Typvariable ist, was wieder einen Trivialfall ergibt oder aber, daß τ zusammengesetzt ist. Hier gilt dann nach Voraussetzung, daß $\tau \in T_{\Omega\setminus k}$ und somit bei allen Vorkommen von α_k in τ in der Typherleitung eine Anwendung der (coerce) Regel mit $k < \widehat{k}$ (echt kleiner) auftritt. Mit der Voraussetzung $fgt_{k,k'}(X) \in k'^{\mathcal{TM}}$ für alle $k' \in \{\bar{k} \mid k \rightsquigarrow \bar{k}\}$ wird garantiert, daß bei $fgt_{k,\widehat{k}}(X)$ eine Struktur im Universum $\widehat{k}^{\mathcal{TM}}$ entsteht, auf der dann die Interpretation des unmittelbar auf α_k angewendeten Tykonstruktors definiert ist. Damit folgt sofort die Behauptung. Von der Struktur X wird also nur der Anteil genutzt, der in einem der Universen liegt.

Die Interpretation von Typtermen unter Pseudo-Belegungen werden wir später noch gezielt einsetzen. Ich werde aber immer explizit im Text darauf hinwiesen, wenn dies der Fall ist. Wenden wir uns jetzt wieder den echten Belegungen zu.

Für die Interpretation eines Typterms in einer abgeänderten Belegung $\nu[X/\alpha]$ mit $X \in k^{\mathcal{TM}}$ gilt folgendes Koinzidenztheorem.

Theorem 2.14 *Koinzidenztheorem für Typterme*

Sei ν eine Typvariablenbelegung und $X \in k^{\mathcal{TM}}$ eine Struktur der Klasse k. Wenn $\alpha \notin TV(\tau)$, dann gilt:

$$\mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu[X/\alpha]} = \mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu}$$

Beweis: Leichte Induktion über den Aufbau des Terms τ.

Das Koinzidenztheorem rechtfertigt, für Terme $\tau \in T_{\Omega}$ mit $TV(\tau) = \emptyset$ statt $\mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu}$ einfach $\tau^{\mathcal{TM}}$ zu schreiben.

Für die formale Behandlung der Logik HOLC ist es bisweilen notwendig, die Interpretation eines Typterms τ mit der Interpretation von $\sigma(\tau)$ zu vergleichen, wobei σ eine Typsubstitution ist. Zu diesem Zweck wird jetzt eine zweite Art der Belegungsänderung eingeführt.

Definition 2.34 *Belegungsänderung $\nu[\sigma/V]$ bzgl. Substitution σ und Variablenmenge V*

Sei ν eine Belegung und σ eine Typsubstitution. Die Änderung $\nu[\sigma/V]$ der Belegung ν bzgl. der Substitution σ und der Menge V von Typvariablen ist dann definiert wie folgt:

$$\nu[\sigma/V](\alpha_k) = \begin{cases} fgt_{k',k}(\mathcal{TM}[\![\sigma(\alpha_k)]\!]^{\Omega}_{\nu}) & \text{falls } \alpha_k \in V \\ & \text{wobei } k' = lst_{\Omega}(\sigma(\alpha_k)) \\ \nu(\alpha_k) & \text{sonst} \end{cases}$$

Die Belegung ν wird also für alle Variablen α_k in V so abgeändert, daß der neue Wert gerade die Interpretation des Terms $\sigma(\alpha_k)$ ist. Im allgemeinen kann $\sigma(\alpha_k)$ jedoch ein Typterm der Klasse k' mit $k' \leq k$ sein, und somit muß die Interpretation noch auf die Klasse k reduziert werden, damit wieder eine korrekte Typvariablenbelegung entsteht.

Theorem 2.15

Für $\nu[\sigma/V]$ gelten offensichtlich folgende Aussagen, die in den Beweisen für noch folgende Theoreme benötigt werden.

$$\mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu[\sigma/V]} = \mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu[\sigma/TV(\tau)\cap V]}$$

$$TV(\tau) \subseteq V \implies \mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu[\sigma/V]} = \mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu[\sigma/TV(\tau)]}$$

$$TV(\tau) \cap V = \emptyset \implies \mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu[\sigma/V]} = \mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu}$$

Beweis: folgt unmittelbar aus Definition von $\nu[\sigma/V]$ und dem Koinzidenztheorem für Typterme.

Das nächste Theorem ist unter anderem zentral für die Korrektheit der Instantiierungsregel im Deduktionssystem von HOLC, die es ermöglicht, polymorphe Aussagen zu spezialisieren. Weiterhin wird es für den Beweis eines Substitutionslemmas auf Termebene benötigt.

Theorem 2.16 *Substitutionslemma für Typsubstitutionen in Typtermen*

Sei $\tau \in T_\Omega$, σ eine Substitution, $\mathcal{TM}$ ein Ω-Modell und ν eine Typvariablenbelegung. Dann gilt:

$$fgt_{c,k}(\mathcal{TM}[\![\sigma(\tau)]\!]^\Omega_\nu) = \mathcal{TM}[\![\tau]\!]^\Omega_{\nu[\sigma/TV(\tau)]}$$

wobei

$$c = lst_\Omega(\sigma(\tau)) \quad \text{und} \quad k = lst_\Omega(\tau)$$

Beweis: Per Induktion über den Aufbau von τ. Dabei wird insbesondere das Theorem 2.3 benutzt, sowie eine zum Beweis von Theorem 2.3 analoge Argumentation.

Ein zum Substitutionslemma 2.16 sehr ähnliches Theorem macht eine Aussage über den Zusammenhang der Interpretationen des Typterms $\tau \in T_{\Omega_1}$ bzgl. der beiden Signaturen Ω_1 und Ω_2 wobei $\Omega_1 \subseteq \Omega_2$. Dieses Theorem wird bei Argumentationen über die konservative Erweiterung benötigt, um zu zeigen, daß sich die Interpretation von Termen über der alten Signatur nicht ändert.

Theorem 2.17 *Zusammenhang von* $\mathcal{TM}[\![\tau]\!]^{\Omega_1}_\nu$ *und* $\mathcal{TM}[\![\tau]\!]^{\Omega_2}_\nu$

Seien Ω_1 und Ω_2 Signaturen mit $\Omega_1 \subseteq \Omega_2$ und sei $\mathcal{TM}$ ein Ω_2-Modell. Sei weiterhin $\tau \in T_{\Omega_1}$ und ν eine Typvariablenbelegung. Dann gilt:

$$fgt_{c,k}(\mathcal{TM}[\![\tau]\!]^{\Omega_2}_\nu) = \mathcal{TM}[\![\tau]\!]^{\Omega_1}_\nu$$

wobei

$$c = lst_{\Omega_2}(\tau) \quad \text{und} \quad k = lst_{\Omega_1}(\tau)$$

Beweis: Per Induktion über den Aufbau von τ. Dabei wird insbesondere das Theorem 2.4 benutzt, sowie eine zum Beweis von Theorem 2.4 analoge Argumentation.

Als nächstes möchte ich Modelle für Signaturen Σ einführen. Um diese entsprechend charakterisieren zu können, benötige ich aber noch den Begriff der typkorrekten Struktur. Eine Struktur der Klasse k soll typkorrekt heißen, wenn die unter $const(X)(c)$ erreichbare Interpretation für eine charakteristische Konstante $c{:}\tau$ ein Element im Träger der Interpretation des Typs τ ist. Um diese Eigenschaft zu testen, werde ich Pseudo-Typvariablenbelegungen verwenden. Dies ist zwar noch nicht notwendig für die Charakterisierung von Σ-Modellen, wird aber dann bei Beweisen für konservative Erweiterungen in Abschnitt 2.6 eingesetzt.

Definition 2.35 *Typkorrekte Struktur*

Sei $\Sigma = (\Omega, C, KS)$ eine Signatur mit $\Omega = (K, \leq, TC)$ und $\mathcal{TM}$ ein Typmodell für Ω. Sei weiterhin X eine Struktur der Klasse $k \in K$, die nicht notwendigerweise Element des Universums $k^{\mathcal{TM}}$ ist.

Die Struktur X heißt typkorrekt bzgl. der Klasse k und dem Typmodell $\mathcal{TM}$ (i.Z. $X\ tcor\ (\mathcal{TM}, KS_k)$), wenn folgende Eigenschaften erfüllt sind.

1. Für alle direkten Nachfolger $k' \in \{\bar{k} \mid k \rightsquigarrow \bar{k}\}$ von k gilt, daß:

   $$fgt_{k,k'}(X) \in k'^{\mathcal{TM}}$$

   Die Struktur X ist also aus Teilstrukturen zusammengesetzt, die Elemente des Typmodells $\mathcal{TM}$ sind.

2. Die Abbildung $const(X)$ ist genau für die charakteristischen Konstanten $c{:}\tau \in KS_k$ definiert, d.h.

   $$const(X)(c) \text{ definiert } \Leftrightarrow \exists \tau.\ c{:}\tau \in KS_k$$

3. Nach Bedingung E3 der Definition 2.10 für Signaturen haben die charakteristischen Konstanten $c{:}\tau \in KS_k$ die Eigenschaft, daß ihre Typen τ aus $T_{\Omega\setminus k}$ sind und genau eine gemeinsame Klassenvariable α_k enthalten.

   Für alle charakteristischen Konstanten $c{:}\tau \in KS_k$ und Belegungen ν soll gelten:

   $$const(X)(c) \in car(\mathcal{TM}[\![\tau]\!]^{\Omega\setminus k}_{\nu[X/\alpha_k]})$$

   Da in τ nur die eine Typvariable α_k vorkommt und obendrein ν an dieser Stelle abgeändert wird, ist die Wahl von ν unerheblich. Nach Theorem 2.13 ist die Interpretation in der Pseudobelegung $\nu[X/\alpha_k]$ wohldefiniert.

2.4.2 Modelle für Signaturen

In diesem Abschnitt werden Modelle für Signaturen eingeführt.

Definition 2.36 *Modelle für Signaturen* Σ

Sei $\Sigma = (\Omega, C, KS)$ eine Signatur mit $\Omega = (K, \leq, TC)$. Ein Modell $\mathcal{M} = (\mathcal{TM}, \mathcal{C})$ für Σ ist ein Paar, das folgende Bedingungen erfüllt.

E1: $\mathcal{TM}$ ist ein Ω-Modell $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$

E2: Für alle Klassen $k \in K$ soll gelten:

$k^{\mathcal{TM}}$ enthält nur typkorrekte Strukturen X der Klasse k.

Gemäß der Definition für typkorrekte Strukturen bedeutet dies also:

$$\forall c{:}\tau \in KS_k.\ \forall X \in k^{\mathcal{TM}}.\ \forall \nu$$
$$const(X)(c) \in car(\mathcal{TM}[\![\tau]\!]^{\Omega\setminus k}_{\nu[X/\alpha_k]})$$

Bemerkung: Da $X \in k^{\mathcal{TM}}$ ist, handelt es sich bei $\nu[X/\alpha_k]$ um eine echte Belegung.

E3: $\mathcal{C}$ ist eine Menge von Interpretation für die Konstanten in C, so daß es für jede Konstante $c{:}\tau \in C$ mit $TV(\tau) = \{\alpha_1, \ldots, \alpha_n\}$ und $\alpha_1 \prec \ldots \prec \alpha_n$ eine Funktion $c^{\mathcal{M}}$ aus einem n-stelligen verallgemeinerten Produkt gibt mit:

$$c^{\mathcal{M}} \in \{f \mid \forall \nu.\ f(\nu(\alpha_1), \ldots, \nu(\alpha_n)) \in car(\mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu})\}$$

Die Reihenfolge der Argumente von $c^{\mathcal{M}}$ ist dabei eindeutig durch die Ordnung $\prec$ auf den Typvariablen in $TV(\tau)$ gegeben. Wenn speziell $TV(\tau) = \emptyset$, dann soll die Interpretation $c^{\mathcal{M}}$ der Konstanten $c{:}\tau$ wie üblich ein Element im Träger $car(\tau^{\mathcal{TM}})$ sein.

E4: Für alle Klassen $k \in K$ und alle charakteristischen Konstanten $c{:}\tau \in KS_k$ soll gelten:

$$\forall X \in k^{\mathcal{TM}}.\ c^{\mathcal{M}}(X) = const(X)(c)$$

Durch diese Bedingung wird also erzwungen, daß die Instanz der polymorphen Interpretation $c^{\mathcal{M}}$ der charakteristischen Konstanten $c{:}\tau \in KS_k \subseteq C$ für die Struktur X gerade dasjenige Element ist, das durch $const(X)(c)$ selektiert wird. Hiermit kommt zum Ausdruck, daß die Strukturen X der Klasse k alle Interpretationen für die charakteristischen Konstanten mit sich herumtragen.

E5: Die Konstanten der Minimalsignatur Σ_{Min} werden nach folgendem Standard interpretiert:

- Die Interpretation $\rightarrow^{\mathcal{M}}$ für die Konstante $\rightarrow : bool \Rightarrow bool \Rightarrow bool$ ist die logische Implikationsfunktion, die für beliebige Elemente $b_1, b_2 \in Two$ wie folgt definiert ist:
$$\rightarrow^{\mathcal{M}}(b_1)(b_2) = \begin{cases} \mathbb{F} & \text{falls } b_1 = \mathbb{T} \text{ und } b_2 = \mathbb{F} \\ \mathbb{T} & \text{sonst} \end{cases}$$
- Die Interpretation $=^{\mathcal{M}}$ für die polymorphe Konstante $= : \alpha_{top} \Rightarrow \alpha_{top} \Rightarrow bool$ ist die Funktion, die jeder Struktur $X \in top^{\mathcal{TM}}$ den Identitätstest auf der Trägermenge von X zuordnet. Es gilt also für beliebige Elemente $a, b \in car(X)$:
$$=^{\mathcal{M}}(X)(a)(b) = \begin{cases} \mathbb{T} & \text{falls } a \text{ identisch mit } b \\ \mathbb{F} & \text{sonst} \end{cases}$$
- Die Interpretation $\varepsilon^{\mathcal{M}}$ für die polymorphe Konstante $\varepsilon{:}(\alpha_{top} \Rightarrow bool) \Rightarrow bool$ ist die Funktion, die jeder Struktur $X \in top^{\mathcal{TM}}$ die Auswahlfunktion $\varepsilon^{\mathcal{M}}(X) \in car((X \Rightarrow Two) \Rightarrow Two)$ zuordnet. Es gilt also für beliebiges $f \in car(X \Rightarrow Two)$:
$$\varepsilon^{\mathcal{M}}(X)(f) = \begin{cases} ch(f^{-1}(\mathbb{T})) & \text{falls } f^{-1}(\mathbb{T}) \neq \emptyset \\ ch(car(X)) & \text{sonst} \end{cases}$$
Dabei ist ch die in Eigenschaft E7 aus Definition 2.22 für Präuniversen geforderte Auswahlfunktion auf nichtleeren Mengen. Die Menge $f^{-1}(\mathbb{T})$ ist definiert als:
$$f^{-1}(\mathbb{T}) = \{y \in car(X) \mid f(y) = \mathbb{T}\}$$
und wegen Eigenschaften E2 und E7 von Präuniversen ist $ch(f^{-1}(\mathbb{T}))$ definiert, wenn die Menge $f^{-1}(\mathbb{T})$ nicht leer ist.

Die Eigenschaften E3 und E4 implizieren bereits die Eigenschaft E2. Da beim Prozeß der konservativen Erweiterung jedoch unter anderem gerade die Eigenschaft E2 nachgewiesen werden muß, habe ich sie hier eigenständig aufgeführt.

Als nächstes wird die Interpretation von Termen in Σ-Modellen definiert. Wie bereits erwähnt, verwende ich zur Definition der Semantik und auch später im Deduktionssystem nur vollgetypte Terme $t \in DT_\Sigma$. In den Termen t können nebst Typvariablen $\alpha \in \Xi$ auch Termvariablen $x \in \Psi$ vorkommen. Daher benötigt man zur Interpretation neben einer Typvariablenbelegung ν auch eine Termvariablenbelegung η.

Definition 2.37 *Termvariablenbelegung η*

Sei Σ eine Signatur und $\mathcal{M}$ eine Modell für Σ. Eine Belegung η für die Termvariablenmenge Ψ ist eine Abbidung

$$\eta : \Psi \to \bigcup_{X \in PU} X$$

Nicht alle Termvariablenbelegungen η 'passen' zu den Typannahmen im generischen Kontext $GC(t)$ des Terms t. Wie in [Mit90] wird daher eine Charakterisierung der Belegungen η eingeführt, die für einen Term t relevant sind. Relevant heißt dabei, daß η die Variable x mit einem Element belegt, das in der Trägermenge des Typs τ liegt, wenn $x:\tau \in GC(t)$.

Definition 2.38 *Belegung η erfüllt Kontext Γ unter Typvariablenbelegung ν*

Sei $\mathcal{M}$ eine Modell für Σ, ν eine Belegung für Typvariablen und Γ ein Variablenkontext. Eine Belegung η erfüllt Kontext Γ unter Typvariablenbelegung ν (i.Z. $\eta \; sat \; (\nu, \Gamma)$) gdw.

$$\forall x \in \Psi.\, (\exists \tau.\, (x, \tau) \in \Gamma) \Longrightarrow \eta(x) \in car(\mathcal{T}\mathcal{M}[\![\tau]\!]^{\Omega}_{\nu})$$

Die Abänderung einer Variablenbelegung η an der Stelle x wird wie üblich definiert.

Definition 2.39 *Belegungsänderung für Termvariablen*

Sei η eine Termvariablenbelegung. Die Änderung der Belegung η an der Stelle x ist definiert wie folgt:

$$\eta[a/x](y) = \begin{cases} a & \text{falls } x = y \\ \eta(y) & \text{sonst} \end{cases}$$

Mit Hilfe der obigen Definitionen kann jetzt die Interpretation von vollgetypten Termen $t \in DT_{\Sigma}$ unter den Belegungen ν und η in einem Σ-Modell $\mathcal{M}$ definiert werden.

Definition 2.40 *Interpretation von Termen*

Sei Σ eine Signatur, $\mathcal{M}$ ein Σ-Modell und $t \in DT_{\Sigma}$ ein vollgetypter Term mit generischem Kontext $\Gamma = GC(t)$. Sei weiterhin ν eine Typvariablenbelegung und η eine Termvariablenbelegung mit $\eta \; sat \; (\nu, \Gamma)$.

Die Interpretation $\mathcal{M}[\![t]\!]^{\Sigma}_{\nu,\eta}$ des Terms t ist induktiv über den Aufbau des Terms t definiert.

(typed_const)
wenn $t = c:\tau$
mit $c:e \in C$,
$TV(e) = \{\alpha_1, \ldots, \alpha_n\},\ n \geq 0$,
$\alpha_1 \prec \ldots \prec \alpha_n$,
$\sigma(e) = \tau$

dann $\mathcal{M}[\![c:\tau]\!]^{\Sigma}_{\nu,\eta} = c^{\mathcal{M}}(fgt_{c_1,k_1}(\mathcal{TM}[\![\sigma(\alpha_1)]\!]^{\Omega}_{\nu}), \ldots, fgt_{c_n,k_n}(\mathcal{TM}[\![\sigma(\alpha_n)]\!]^{\Omega}_{\nu}))$
wobei $c_i = lst_{\Omega}(\sigma(\alpha_i))$ und $k_i = lst_{\Omega}(\alpha_i)$

(typed_var)
wenn $t = x:\tau$
dann $\mathcal{M}[\![x:\tau]\!]^{\Sigma}_{\nu,\eta} = \eta(x)$

(app)
wenn $t = (t_1 t_2)$
dann $\mathcal{M}[\![(t_1 t_2)]\!]^{\Sigma}_{\nu,\eta} = (\mathcal{M}[\![t_1]\!]^{\Sigma}_{\nu,\eta})(\mathcal{M}[\![t_2]\!]^{\Sigma}_{\nu,\eta})$

(typed_abs)
wenn $t = (\lambda x:\tau.\ t)$
dann $\mathcal{M}[\![(\lambda x:\tau.\ t)]\!]^{\Sigma}_{\nu,\eta} = f$
wobei f ist die per Extensionalität eindeutig
bestimmte Funktion
$f : a \in car(\mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu}) \mapsto \mathcal{M}[\![t]\!]^{\Sigma}_{\nu,\eta[a/x]}$

Die soeben definierte Interpretation von Termen respektiert die Typen der Terme.

Theorem 2.18 *Typkorrektheit (type soundness)*

Sei $t \in DT_{\Sigma}$ mit generischem Kontext $\Gamma = GC(t)$ und generischem Typ $\tau = GT(t)$. Dann gilt für alle Σ-Modelle $\mathcal{M}$ und Belegungen ν, η mit $\eta\ sat\ (\nu, \Gamma)$:

$$\mathcal{M}[\![t]\!]^{\Sigma}_{\nu,\eta} \in car(\mathcal{TM}[\![\tau]\!]^{\Omega}_{\nu})$$

Beweis: Induktion über den Aufbau des Terms t.

Für die Interpretation eines Terms ist nur die Belegung der freien Variablen des Terms interessant. Dies wird durch das folgende Koinzidenztheorem für Terme ausgedrückt.

Theorem 2.19 *Koinzidenztheorem für Terme*

Sei $t \in DT_{\Sigma}$ mit generischem Kontext $\Gamma = GC(t)$ und sei $y \notin FV(t)$. Dann gilt für alle Modelle $\mathcal{M}$ und Belegungen ν, η mit $\eta\ sat\ (\nu, \Gamma)$:

$$\forall a.\ \mathcal{M}[\![t]\!]^{\Sigma}_{\nu,\eta[a/y]} = \mathcal{M}[\![t]\!]^{\Sigma}_{\nu,\eta}$$

Beweis: Induktion über den Aufbau des Terms t und übliche Argumentation für den Fall der λ-Abstraktion.

Wie schon bei Typtermen ist es interessant, den Effekt einer Typsubstitution auf die Interpretation eines Terms zu beobachten.

Theorem 2.20 *Typsubstitution in Termen*

Sei $t \in DT_\Sigma$ mit generischem Kontext $\Gamma = GC(t)$ und σ eine Typsubstitution. Dann gilt für alle Modelle $\mathcal{M}$ und Belegungen ν, η mit $\eta \; sat \; (\nu, \Gamma)$:

$$\mathcal{M}[\![\sigma(t)]\!]^\Sigma_{\nu,\eta} = \mathcal{M}[\![t]\!]^\Sigma_{\nu[\sigma/TV(t)],\eta}$$

Beweis: Induktion über den Aufbau des Terms t. Wesentliches Argument für den Fall (typed_const) ist das Theorem 2.16 für die Substitution in Typtermen. Für die Fälle (typed_var) und (abs) verwendet man das Carrier-Lemma 2.10.

Bei der Erweiterung von Theorien möchte man natürlich sicherstellen, daß die Interpretation von Termen über der alten Signatur sich nicht ändert. Eine Eigenschaft, die später mithilft diese Tatsache zu beweisen, wird durch folgendes Theorem charakterisiert.

Theorem 2.21 *Invarianz der Interpretation bei Signaturerweiterung*

Sei $t \in DT_{\Sigma_1}$ mit generischem Kontext $\Gamma = GC_{\Sigma_1}(t)$ bzgl. Typinferenz in Σ_1 und $\Sigma_1 \subseteq \Sigma_2$. Dann gilt für alle Σ_2-Modelle $\mathcal{M}$ und Belegungen ν, η mit $\eta \; sat \; (\nu, \Gamma)$:

$$\mathcal{M}[\![t]\!]^{\Sigma_2}_{\nu,\eta} = \mathcal{M}[\![t]\!]^{\Sigma_1}_{\nu,\eta}$$

Beweis: Induktion über den Aufbau des Terms t. Wesentliches Argument für den Fall (typed_const) ist das Theorem 2.17 für die Interpretation bei Erweiterung der Typsignatur. Für die Fälle (typed_var) und (abs) verwendet man das Carrier-Lemma 2.10.

Als nächstes führe ich das Konzept der Termsubsitution $t[t'/x]$ ein, wobei in einem vollgetypten Term t für jedes freie Vorkommen der Termvariablen x der Term t' eingesetzt wird, ohne daß es dabei zu unerwünschten Variablenbindungen kommt. Da der Substitutionsmechanismus in allen Abhandlungen über den λ-Kalkül behandelt wird, möchte ich in der folgenden Definition nur eine informelle Charakterisierung der Substitution angeben. In Bezug auf die Eigenschaften der Substitution und deren Beweise möchte ich auf die Standardliteratur verweisen.

Definition 2.41 *Substitution von Termen in Termen*

Für einen vollgetypten Term $t \in DT_\Sigma$ mit generischem Kontext $\Gamma = GC(t)$ bezeichnet $t[t'/x]$ den Term, der durch das Ersetzen aller freien Vorkommen der Termvariablen $x \in \Psi$ durch den ebenfalls vollgetypten Term $t' \in DT_\Sigma$ entsteht. Es müssen dabei folgende Bedingungen erfüllt sein:

E1: Wenn $(x, \tau') \in \Gamma$ für ein beliebiges τ', dann ist der generische Typ $GT(t')$ von t' ebenfalls τ'. Diese Eigenschaft garantiert eine typkorrekte Ersetzung.

E2: Gebundene Variablen werden während der Substitution gegebenenfalls umbenannt, damit nicht aus Versehen freie Variablen in t' gebunden werden.

Für die Interpretationen der Terme t und $t[t'/x]$ ergibt sich folgender Zusammenhang.

Theorem 2.22 *Substitution in Termen*

Sei $t \in DT_\Sigma$ mit generischem Kontext $\Gamma = GC(t)$ und sei $t[t'/x]$ das Ergebnis der Substitution von $t' \in DT_\Sigma$ für die Variable $x \in \Psi$. Dann gilt für alle Modelle $\mathcal{M}$ und Belegungen ν, η mit $\eta \; sat \; (\nu, \Gamma)$:

$$\mathcal{M}[\![t[t'/x]]\!]^\Sigma_{\nu,\eta} = \mathcal{M}[\![t]\!]^\Sigma_{\nu,\eta[a/x]} \text{ wobei } a = \mathcal{M}[\![t']\!]^\Sigma_{\nu,\eta}$$

Beweis: Induktion über den Aufbau des Terms t.

Im Abschnitt 2.6 werde ich unter anderem auch die konservative Erweiterung durch eine neue Klasse k behandeln. Dabei ist es wichtig, daß das Universum $k^{\mathcal{TM}}$ für die neue Klasse maximal groß definiert wird. Maximal bedeutet hierbei, daß alle gemäß Definition 2.35 typkorrekten Strukturen X der Klasse k aufgenommen werden, für die die charakteristischen Axiome der Klasse k gelten. Zu diesem Zweck führe ich nun den Begriff der *klassengültigen* Struktur ein. Wie schon bei Definition 2.35 werden auch hier wieder Pseudobelegungen $\nu[X/\alpha_k]$ benötigt.

Definition 2.42 *Klassengültige Strukturen*

Sei $Th = (\Sigma, Ax, KAx)$ eine Σ-Theorie gemäß Definition 2.21 mit $\Sigma = (\Omega, C, KS)$ und $\Omega = (K, \leq, TC)$, und sei $\mathcal{M} = (\mathcal{TM}, \mathcal{C})$ ein Σ-Modell. Nach Bedingung E3 aus Definition 2.10 und Bedingung E3 aus Definition 2.21 gibt es für alle Klassen $k \in K$ eine Klassenvariable $\alpha_k \in \Xi_k$, so daß gilt:

$$\forall c{:}\tau \in KS_k.\, TV(\tau) = \{\alpha_k\} \wedge \tau \in T_{\Omega\backslash k}$$
$$\forall ax \in KAx_k.\, TV(ax) = \{\alpha_k\} \wedge ax \in FORM_{\Sigma\backslash k}$$

Eine typkorrekte Struktur X der Klasse $k \in K$, die nicht notwendigerweise Element des Universums $k^{\mathcal{TM}}$ sein muß, heißt klassengültig im Modell $\mathcal{M}$ (i.Z. $X \; cval \; (\mathcal{M}, KAx_k)$), wenn alle charakteristischen Axiome der Klasse k unter der (evtl. Pseudo-) Typvariablenbelegung $\nu[X/\alpha_k]$ gelten. Formal heißt das:

Für alle $ax \in KAx_k$ mit generischem Kontext $\Gamma = GC(ax)$ und für alle Belegungen ν, η mit $\eta \; sat \; (\nu[X/\alpha_k], \Gamma)$ muß gelten:

$$\mathcal{M}[\![ax]\!]^{\Sigma\backslash k}_{\nu[X/\alpha_k],\eta} = \mathbb{T}$$

Dabei sei die Interpretation für die charakteristischen Konstanten $c{:}\tau \in KS_k$ kurzzeitig so erweitert, daß für die Struktur X gilt:

$$c^{\mathcal{M}}(X) = const(X)(c)$$

Bemerkung: Wenn $X \in k^{\mathcal{TM}}$ ist, dann ist $\nu[X/\alpha_k]$ eine echte Belegung. Dann ist aber die obige Definition für $c^{\mathcal{M}}(X)$ gar keine Erweiterung, sondern entspricht der ohnehin in Bedingung E4 aus Definition 2.36 geforderten Interpretation.

2.4.3 Modelle für Theorien

In diesem Abschnitt führe ich die Begriffe der Gültigkeit von Formeln und der Modelle für Theorien ein.

Definition 2.43 *Gültigkeit von Formeln*

Sei Σ eine Signatur und $\mathcal{M}$ ein Σ-Modell. Eine Formel $p \in FORM_\Sigma$ mit generischem Kontext $\Gamma = GC(p)$ heißt gültig in $\mathcal{M}$ (i.Z. $\mathcal{M} \models p$), wenn für alle Belegungen ν, η mit $\eta \; sat \; (\nu, \Gamma)$ gilt:

$$\mathcal{M}[\![p]\!]^\Sigma_{\nu,\eta} = \mathbb{T}$$

Definition 2.44 *Modelle für Theorien*

Sei $Th = (\Sigma, Ax, KAx)$ eine Σ-Theorie gemäß Definition 2.21 und sei $\mathcal{M}$ ein Σ-Modell.

Das Σ-Modell $\mathcal{M}$ ist ein Modell für die Theorie Th (i.Z. $\mathcal{M} \models Th$), wenn folgende Bedingungen erfüllt sind:

E1: Alle Axiome der Theorie Th in der Menge Ax sind gültig in $\mathcal{M}$, d.h.:

$$\forall ax \in Ax . \; \mathcal{M} \models ax$$

Bemerkung: Damit gelten insbesondere alle charakteristischen Axiome, da $KAx \subseteq Ax$.

E2: Die Universen $k^{\mathcal{T}\mathcal{M}}$ für die Klassen $k \in K$ sind maximal groß, d.h. für alle Klassen $k \in K$ und für alle typkorrekten Strukturen X der Klasse k gilt:

$$X \; cval\,(\mathcal{M}, KAx_k) \Longrightarrow X \in k^{\mathcal{T}\mathcal{M}}$$

Bemerkung: Diese Bedingung wird später bei der konservativen Erweiterung durch eine neue Arität ausgenützt werden. Vergleiche dazu auch die informelle Einführung aus Abschnitt 2.2.

Definition 2.45 *Gültigkeit von Formeln in einer Theorie*

Eine Formel $p \in FORM_\Sigma$ heißt gültig in der Theorie Th (i.Z. $Th \models p$), wenn die Formel p in allen Th-Modellen $\mathcal{M}$ gilt, d.h:

$$Th \models p \Longleftrightarrow \forall \mathcal{M} . \; \mathcal{M} \models Th \Longrightarrow \mathcal{M} \models p$$

2.5 Deduktionssystem von HOLC

In diesem Abschnitt werde ich den Kalkül von HOLC einführen. Zu diesem Zweck benötige ich die Begriffe für Sequenzen und Deduktionssysteme. Die Definitionen sind eng an [GM93] angelehnt, und auch der Kalkül von HOLC unterscheidet sich nur minimal vom Kalkül der Logik HOL.

Der Kalkül von HOLC ist ein Kalkül des natürlichen Schließens (Annahmenkalkül) [Gen35]. Um die Nebenbedingungen (side conditions) für Inferenzregeln leichter formulieren zu können,

verwende ich zur Notation der Regeln jedoch explizite Kontexte für die 'lebenden Annahmen' (living hypotheses). Ein weiterer Grund für die explizite Verwaltung der Annahmen liegt darin, daß so die Korrektheit der Inferenzregeln leichter formuliert werden kann. Bei der originalen Notation für Annahmenkalküle sind die lebenden Annahmen über den ganzen Beweisbaum verteilt und dies erschwert die Formulierung der Korrektheit für einzelne Regeln.

Obwohl die Kontexte endliche *Mengen* von Formeln sind, werden Paare $(\{h_1, \ldots, h_n\}, p)$, bestehend aus der Menge der lebenden Annahmen $\{h_1, \ldots, h_n\}$ und der Formel p, in der Literatur [Gor85, GM93, Pau87] als *Sequenzen (sequents)* bezeichnet und in der Form $h_1, \ldots, h_n \blacktriangleright p$ notiert[20]. Diese Sequenzen dürfen jedoch nicht mit den Sequenzen in einem richtigen Sequenzenkalkül, etwa dem Kalkül LK von Gerhard Gentzen [Gen35, Sza69, Tak75], verwechselt werden. In der englischen Literatur findet sich statt *sequent* auch die Bezeichnung *assertion*. Da mir die deutsche Übersetzung *Behauptung* jedoch nicht prägnant genug erscheint, bleibe ich bei der etwas unglücklichen Bezeichnung *Sequenz*.

In einem echten Sequenzenkalkül bestehen Sequenzen $\Gamma \to \Delta$ aus einer *Liste* von Annahmen Γ, dem Antezedens (antecedent), und einer Liste von Folgerungen Δ, dem Sukzedens (succedent) [Gen35, Sza69]. In intuitionistischen Sequenzenkalkülen, etwa LJ [Gen35, Sza69, Tak75], verkümmert der Sukzedens wieder zu einer einzigen Formel. In Sequenzenkalkülen ist die Multiplizität und die Anordnung von Formeln in einer der Listen von entscheidender Bedeutung, und es gibt spezielle strukturelle Regeln, die eine diesbezügliche Manipulation erlauben.

Definition 2.46 *Sequenzen*

Eine Sequenz über der Signatur Σ ist ein Paar (H, p) mit folgenden Eigenschaften:

E1: $H = \{h_1, \ldots, h_n\}$ ist eine endliche Menge von Σ-Formeln und heißt die Menge der Annahmen (Annahmenmenge) oder der Antezedens der Sequenz.

E2: p ist eine Σ-Formel und heißt die Konsequenz oder Sukzedens der Sequenz.

E3: $h_1 \to \ldots h_n \to p$ ist ebenfalls eine Σ-Formel.

Statt der Paarnotation verwende ich auch, der Konvention entsprechend, die etwas schlampige Notation

$$h_1, \ldots, h_n \blacktriangleright p$$

bei der die Menge der Annahmen als Liste notiert wird. Es ist jedoch immer eine Menge gemeint!

Die Bedingung E3 mag etwas sonderbar anmuten. Ihr Zweck ist lediglich, eine einheitliche Typisierung für alle freien Variablen in den Formeln $h_1, \ldots, h_n, p$ zu garantieren. Ohne diese Bedingung wären auch Sequenzen wie $x \blacktriangleright (xy) \to z$ erlaubt. Hier müßte das x im Antezedens als Term vom Typ *bool* getypt werden, wohingegen das x im Sukzedens einen funktionalen Typ haben müßte. In diesem Fall wäre dann die gleich folgende Definition der Semantik von Sequenzen, die an [GM93] angelehnt ist, nicht brauchbar.

[20] zur Trennung der Annahmen $\{h_1, \ldots, h_n\}$ und der Formel p werden vielerlei Zeichen, manchmal sogar $\vdash$, benutzt. Ich habe mich für das Zeichen $\blacktriangleright$ entschieden.

Definition 2.47 *Semantik von Sequenzen*

Sei $h_1, \ldots, h_n \blacktriangleright p$ eine Sequenz über der Signatur Σ, und sei $\mathcal{M}$ ein Σ-Modell. Die Sequenz $h_1, \ldots, h_n \blacktriangleright p$ heißt gültig im Modell $\mathcal{M}$ (i.Z. $\mathcal{M} \models h_1, \ldots, h_n \blacktriangleright p$) gdw.:

$$\mathcal{M} \models h_1 \rightarrow \ \ldots \rightarrow h_n \rightarrow p$$

Eine Sequenz $h_1, \ldots, h_n \blacktriangleright p$ heißt gültig in der Σ-Theorie Th (i.Z. $Th \models h_1, \ldots, h_n \blacktriangleright p$), wenn:

$$\forall \mathcal{M}.\, \mathcal{M} \models Th \Longrightarrow \mathcal{M} \models h_1, \ldots, h_n \blacktriangleright p$$

Da in der Minimalsignatur Σ_{Min} noch kein Zeichen für die logische Konjunktion $\wedge$ enthalten ist, habe ich die Annahmen durch eine Implikation verbunden. Würde man in der obigen Definition die Annahmen durch eine Konjunktion verbinden, also $h_1 \wedge \ \ldots \ \wedge h_n \rightarrow p$, ergäbe sich die gleiche Semantik. Wichtig ist, daß der Sequenzenpfeil $\blacktriangleright$ die Semantik einer Implikation $\rightarrow$ erhält. Eine Sequenz $h_1, \ldots, h_n \blacktriangleright p$ ist also dann gültig, wenn unter *jeder Belegung* der Variablen aus der Gültigkeit der Formeln $h_1, \ldots, h_n$ die Gültigkeit der Formel p folgt.

Die obige Definition der Semantik wird unter anderem auch durch die Übersetzbarkeit von Beweisen im Kalkül des natürlichen Schließens NK in Beweise des Sequenzenkalküls LK motiviert. Diese Übersetzung wird zum Beispiel im Originalpapier [Gen35] und auch in der englischen Übersetzung [Sza69] angegeben.

Es gibt verschiedene Möglichkeiten, die Axiome einer Theorie in Herleitungen zu verwenden. In der hier vorgestellten Variante werden Axiome nicht als Annahmen behandelt, sondern bekommen einen speziellen Status im Deduktionssystem. Auf diese Weise kann die Herleitbarkeit bezüglich einer Theorie ausgedrückt werden. Zu diesem Zweck werden triviale Sequenzen, die sogenannten Theoriesequenzen einer Theorie, eingeführt. Theoriesequenzen haben eine leere Annahmenmenge und die Konsequenz besteht aus einem Axiom der Theorie.

Definition 2.48 *Theoriesequenzen*

Sei $Th = (\Sigma, Ax, KAx)$ eine Theorie. Die Menge Δ_{Th} der zur Theorie Th assoziierten Theoriesequenzen ist die endliche Menge:

$$\Delta_{Th} = \{\emptyset \blacktriangleright ax \mid ax \in Ax\}$$

Als nächstes wird der Begriff eines Deduktionssystems eingeführt. Die Definition folgt dabei wieder [GM93].

Definition 2.49 *Deduktionssystem*

Ein Deduktionssystem D über einer Signatur Σ ist eine im allgemeinen unendliche Menge von Paaren (PR, C), wobei PR eine evtl. leere Liste von Σ-Sequenzen und C eine einzelne Σ-Sequenz ist.

Die Paare $(PR, C) \in D$ werden als Regeln des Deduktionssystems D bezeichnet. Die Komponente PR ist die Liste der Prämissen der Regel (PR, C), und die Sequenz C heißt die Konklusion der Regel.

Die unendliche Menge D der Regeln wird üblicherweise durch eine endliche Menge von Regelschemata beschrieben. In diesen Schemata dürfen Schemavariablen für Terme, Formeln und Formelmengen vorkommen. Jede mögliche Instanz der Schemavariablen liefert eine Regel des Deduktionssystems D. Der Begriff der Instanz hat hierbei nichts mit den Instanzen polymorpher Konstanten zu tun. Um die Schemavariablen in den Regelschemata zu formalisieren, werden in Systemen wie Isabelle Unifikationsvariablen benutzt, und die Instantiierung von Regeln wird per Unifikation behandelt.

Für die Notation eines Regelschemas für Regeln $(PR, C) \in D$ verwendet man statt der Paarnotation

$$([H_1 \blacktriangleright p_1, \ldots, H_n \blacktriangleright p_n], H \blacktriangleright p)]$$

die bekannte Schreibweise

$$\frac{H_1 \blacktriangleright p_1 \quad \ldots \quad H_n \blacktriangleright p_n}{H \blacktriangleright p} \Big\{ \text{Nebenbedingungen}$$

In den Formelmengen H_i, H und den Formeln p_i, p dürfen, wie bereits erwähnt, Schemavariablen vorkommen. Jede mögliche Instanz dieser Schemavariablen, die die rechts von der Regel notierten Nebenbedingungen erfüllt, ist eine Regel des Deduktionssystems D. Natürlich dürfen durch die Instantiierung nur wohlgeformte Terme entstehen. Die Verwendung der expliziten Annahmenmengen H_i, H ermöglicht diese 'lokale' Charakterisierung von Regeln. Nebenbedingungen, wie etwa 'x kommt nicht frei vor in der Annahmenmenge H', werden also nur für die Prüfung der Zulässigkeit einer Regelinstanz benötigt. Im Deduktionssystem D selbst werden keine Nebenbedingungen mehr benötigt!

Will man neben dem logischen Kalkül auch noch den Mechanismus der Ableitung neuer Regeln formalisieren, so wird der formale Apparat schnell unhandlich. Eine befriedigende Lösung bietet hier die Verwendung von Meta-Logiken und Unifikationsmechanismen, mit Hilfe derer dann die Begriffe der Regelableitung und der Instanz von Regeln sauber definiert werden können. Dieser Ansatz wird auch im Isabelle-System verwendet.

Mit Hilfe eines Deduktionssystems D und der Theoriesequenzen Δ_{Th} über der Theorie Th kann jetzt der Begriff der Herleitbarkeit von Sequenzen relativ zu Th und D erklärt werden.

Definition 2.50 *Herleitbarkeit*

Sei Th eine Σ-Theorie, Δ_{Th} die assoziierte Menge der Theoriesequenzen und D ein Deduktionssystem über Σ. Eine Σ-Sequenz $H \blacktriangleright p$ ist in der Theorie Th unter Verwendung des Deduktionssystem D herleitbar (i.Z. $(Th, D) \vdash H \blacktriangleright p$), dann und nur dann, wenn es eine endliche Folge

$$(H_1 \blacktriangleright p_1, \ldots, H_n \blacktriangleright p_n), n > 0$$

gibt, einen sogenannten Beweis für $H \blacktriangleright p$, so daß folgende Bedingungen erfüllt sind:

1. Der Beweis endet mit $H \blacktriangleright p$, d.h:

   $$H \blacktriangleright p = H_n \blacktriangleright p_n$$

2. Jedes Glied der Folge ist entweder eine Theoriesequenz, und damit ein Axiom der Theorie, oder entsteht durch die Anwendung einer Regel des Deduktionssystems D unter Verwendung von Prämissen, die weiter vorne in der Folge stehen. Formal heißt das:

   Für alle $i \in \{1, \ldots, n\}$ gilt eine der beiden folgenden Bedingungen:

   (a) $H_i \blacktriangleright p_i \in \Delta_{Th}$

   (b) es gibt eine Liste von Prämissen PR_i
   mit
   $$PR_i = [B_{i1} \blacktriangleright p_{i1}, \ldots, B_{im} \blacktriangleright p_{im}], m \geq 0$$
   und
   $$\forall j \in \{1, \ldots, m\}.\ B_{ij} \blacktriangleright p_{ij} \in \{H_1 \blacktriangleright p_1, \ldots, H_{i-1} \blacktriangleright p_{i-1}\}$$
   so daß
   $$(PR_i, H_i \blacktriangleright p_i) \in D$$

Wenn die Menge der Annahmen leer ist, dann schreibe ich statt $(Th, D) \vdash \emptyset \blacktriangleright p$ auch kurz $(Th, D) \vdash p$.

Der Herleitungsbegriff ist monoton bezüglich der Erweiterung von Theorien.

Theorem 2.23 *Monotonie der Herleitbarkeit*

Wenn die Sequenz $H \blacktriangleright p$ in einer Theorie Th_1 herleitbar ist und wenn $Th_1 \subseteq Th_2$, dann ist die Sequenz auch in Th_2 herleitbar. Es gilt also:

$$\forall D.\ (Th_1, D) \vdash H \blacktriangleright p \text{ und } Th_1 \subseteq Th_2 \Longrightarrow (Th_2, D) \vdash H \blacktriangleright p$$

Beweis: Folgt offensichtlich aus der Definition der Herleitbarkeit

Ein Deduktionssystem wird dazu benutzt, um Theoreme einer Theorie abzuleiten. Aus diesem Grund sind nur solche Deduktionssysteme sinnvoll, die garantieren, daß alle abgeleiteten Sequenzen gültig sind bezüglich der Theorie. In diesem Fall spricht man von einem korrekten Deduktionssystem.

Definition 2.51 *Korrektheit von Deduktionssystemen*

Ein Deduktionssystem D über der Signatur Σ heißt korrekt bzgl. einer Σ-Theorie Th (i.Z. $(D \; soundfor \; Th)$) genau dann, wenn für alle Th-Modelle $\mathcal{M}$ und Regeln $([Pr_1, \ldots, Pr_n], C) \in D$ gilt:

$$\mathcal{M} \models Pr_1 \wedge \ \ldots \ \wedge \mathcal{M} \models Pr_n \Longrightarrow \mathcal{M} \models C$$

Ein Deduktionssystem D ist also dann korrekt bezüglich einer Theorie, wenn in jedem Modell der Theorie durch die Regeln gültige Sequenzen wieder in gültige Sequenzen überführt werden.

Wenn ein Deduktionssystem korrekt bezüglich einer Theorie ist, dann ist es auch korrekt bezüglich jeder größeren Theorie. Dies drückt folgender Satz aus.

Theorem 2.24 *Monotonie der Korrektheit von Deduktionssystemen*

$(D \; soundfor \; Th_1) \wedge Th_1 \subseteq Th_2 \Longrightarrow (D \; soundfor \; Th_2)$

Beweis: folgt unmittelbar aus der Definition der Korrektheit und Theorem 2.21 über die Invarianz der Interpretation bei Theorieerweiterung.

Aus der Korrektheit eines Deduktionssystems bezüglich einer Theorie folgt unmittelbar, daß jede Sequenz in der Theorie gültig ist, die mit Hilfe dieses Deduktionssystems abgeleitet wird.

Theorem 2.25 *Korrektheit des Herleitungsbegriffs*

Wenn ein Deduktionssystem D korrekt ist bzgl. der Theorie Th, dann ist jede in der Theorie Th unter Verwendung von D herleitbare Sequenz auch gültig in der Theorie.

$$(D \; soundfor \; Th) \; \wedge \; (Th, D) \vdash H \blacktriangleright p \; \Longrightarrow \; Th \models H \blacktriangleright p$$

Beweis: Leichte Induktion über die Länge der Herleitungen und Verwendung der Korrektheit von D.

Für den speziellen Fall, daß die Annahmenmenge der Sequenz leer ist, kann obiges Theorem mit der bereits eingeführten Kurzschreibweise für diesen Fall wie folgt notiert werden:

$$(D \; soundfor \; Th) \; \wedge \; (Th, D) \vdash p \; \Longrightarrow \; Th \models p$$

Als nächstes möchte ich die Minimaltheorie *Min* und das Deduktionssystem D_{Min} einführen. Die Theorie *Min* ist die Basis für alle Theorien der Logik HOLC und das Deduktionssystem D_{Min} ist das Deduktionssystem der Logik HOLC.

Definition 2.52 *Minimaltheorie Min*

Die Minimaltheorie *Min* besteht aus der Minimalsignatur Σ_{Min} (siehe Definition 2.10) und einer leeren Menge von Axiomen.

$$Min = (\Sigma_{Min}, \emptyset, \emptyset)$$

Aus dieser Definition folgt unmittelbar, daß jedes Modell für die Minimalsignatur Σ_{Min} auch ein Modell für die Minimaltheorie *Min* ist.

Das hier vorgestellte Deduktionssystem D_{Min} entspricht im wesentlichen dem Deduktionssystem von HOL [GM93]. Der Unterschied der beiden Systeme besteht nur darin, daß bei der Regel [type_inst] statt einer normalen Typsubstitution eine ordnungssortierte Substitution im Sinne der Definition 2.7 verwendet wird. Dies ist nicht weiter verwunderlich, da HOLC ja gerade die Logik HOL um das Konzept der Typklassen erweitert. Obwohl der technische Apparat für die Syntax und die Semantik von HOLC um einiges komplexer ist als die entsprechenden Gegenstücke in HOL, ist davon an der 'Oberfläche' nicht allzuviel zu sehen.

Definition 2.53 *Deduktionssystem von HOLC*

Das Deduktionssystem D_{Min} der Logik HOLC besteht aus 8 Regelschemata, die in der bereits erwähnten Notation präsentiert werden. Dabei stehen die Schemavariablen p und q für Formeln, die Schemavariablen H, H_i für Annahmenmengen und die Schemavariablen t, t_1, t_2 für beliebige Terme.

Jede mögliche Instanz der Regelschemata, für die die evtl. Nebenbedingungen erfüllt sind, soll im Deduktionssystem D_{Min} enthalten sein.

Einführung von Annahmen: [hyp]

$$\frac{}{p \blacktriangleright p}$$

Reflexivität der Gleichheit: [refl]

$$\frac{}{\emptyset \blacktriangleright t = t}$$

β-Gleichheit: [beta]

$$\frac{}{\emptyset \blacktriangleright (\lambda x.t_1)t_2 = t_1[t_2/x]}$$

Substitutionsregel: [subst]

$$\frac{H_1 \blacktriangleright t_1 = t_2 \quad H_2 \blacktriangleright p[t_1/x]}{H_1 \cup H_2 \blacktriangleright p[t_2/x]}$$

Abstraktionsregel: [abstract]

$$\frac{H \blacktriangleright t_1 = t_2}{H \blacktriangleright (\lambda x.t_1) = (\lambda x.t_2)} \left\{ x \notin FV(H) \right.$$

Typinstantiierung: [type_inst]

$$\frac{H \blacktriangleright p}{H \blacktriangleright \sigma(p)} \left\{ \begin{array}{l} \sigma \text{ ist eine Typsubstitution und} \\ \alpha \in TV(H) \Longrightarrow \sigma(\alpha) = \alpha \end{array} \right.$$

Annahmenentlastung: [disch]

$$\frac{H \blacktriangleright q}{H \setminus \{p\} \blacktriangleright p \rightarrow q}$$

Modus Ponens: [mp]

$$\frac{H_1 \blacktriangleright p \rightarrow q \quad H_2 \blacktriangleright p}{H_1 \cup H_2 \blacktriangleright q}$$

Im Anschluß an die Definition des Deduktionssystems möchte ich einige der Regeln noch kurz kommentieren.

disch: Bei dieser Regel sollte beachtet werden, daß die Formel p, die als Prämisse der Implikation eingeführt wird, nicht unbedingt in der Annahmenmenge H vorkommen muß. Die Regel

$$\frac{\emptyset \blacktriangleright q}{\emptyset \blacktriangleright p \rightarrow q}$$

ist also eine korrekte Instanz des Schemas [disch]. Durch eine derartige Verwendung des Schemas [disch] kann sehr leicht die Abschwächungsregel [weak]

$$\frac{H \blacktriangleright q}{H \cup \{p\} \blacktriangleright q}$$

hergeleitet werden [Gor85].

beta,subst: In diesen beiden Regeln wird der Mechanismus der Termsubstitution $t_1[t_2/x]$ verwendet, der aus Definition 2.41 bekannt ist. Dabei werden alle freien Vorkommen der Variablen x im Term t_1 durch den Term t_2 ersetzt, wobei vorher evtl. gebundene Variablen in t_1 umbenannt werden, so daß keine Variablen in t_2 versehentlich gebunden werden. Speziell in der Regel [subst] bedeutet die Schreibweise $p[t_1/x]$, daß es einen Term p gibt, in dem evtl. die Variable x frei vorkommt, und daß man nach Substitution von t_1 für x gerade die Formel $p[t_1/x]$ erhält, die im Sukzedens der Prämisse der Regel steht.

type_inst: Diese Regel mag auf den ersten Blick sehr eingeschränkt erscheinen, da nur Typsubstitutionen σ verwendet werden dürfen, die die Formeln im Kontext H nicht verändern. Durch mehrfache Anwendung der Regel [disch] kann der Kontext H aber für jede beliebige Typsubstitution σ in eine Form gebracht werden, in der die Anwendung von [type_inst] erlaubt ist. Nach Anwendung von [type_inst] können die vorher entlasteten Annahmen mittels der Regel [undisch]

$$\frac{H \blacktriangleright p \rightarrow q}{H \cup \{p\} \blacktriangleright q}$$

die ihrerseits leicht per [hyp] und [mp] ableitbar ist [Gor85], in den Kontext zurückexpidiert werden. Diese Bemerkungen legen es nahe, gleich die bequemere Regel

$$\frac{H \blacktriangleright p}{\sigma(H) \blacktriangleright \sigma(p)} \Big\{ \sigma \text{ ist eine Typsubstitution}$$

in den Kalkül aufzunehmen. Da diese Regel aber eine Aktivität im Kontext beschreibt, die über das pure Hinzufügen und Entlasten von Annahmen hinausgeht, ist die Nähe zum eigentlichen Kalkül des natürlichen Schließens nicht mehr unmittelbar gegeben. Die Verwendung der expliziten Kontexte sollte ja nur eine notationelle Variante für die übliche Aufschreibung von Annahmenkalkülen sein. Selbsverständlich handelt es sich hier um eine reine Geschmacksfrage.

Als letztes Theorem dieses Abschnittes möchte ich nun die Korrektheit des Deduktionssystems D_{Min} formulieren.

Theorem 2.26 *Korrektheit von D_{Min}*

Das Deduktionssystem D_{Min} ist korrekt bezüglich jeder Theorie, d.h. für alle Theorien gilt:

$$(D_{Min}\ soundfor\ Th)$$

und damit wegen Theorem 2.25 auch:

$$(Th, D_{Min}) \vdash H \blacktriangleright p \implies Th \models H \blacktriangleright p$$

Beweis: Da die Theorie *Min* keine Axiome hat und die Signatur Σ_{Min} enthält, gilt aufgrund der Definition des Theoriebegriffs, daß $Min \subseteq Th$ für jede Theorie Th. Wegen Theorem 2.24 über die Monotonie der Korrektheit von Deduktionssystemen folgt damit aus der Korrektheit von D_{Min} bezüglich *Min* die Korrektheit bezüglich jeder Theorie. Es reicht also zu zeigen, daß:

$$(D_{Min}\ soundfor\ Min)$$

Da *Min* keine Axiome enthält, ist jedes Modell der Minimalsignatur auch ein Modell der Minimaltheorie *Min*. Es reicht damit zu zeigen, daß in jedem Modell der Minimalsignatur Σ_{Min} die Regeln des Deduktionssystems gültige Sequenzen in gültige Sequenzen überführen. Um dies zu zeigen, reichen bereits die Theoreme aus Abschnitt 2.4 aus.

Ein wesentliches Argument für die Korrektheit des Deduktionssystems D_{Min} ist die Tatsache, daß in HOLC sowohl die Universen für Typklassen als auch die Trägermengen der Interpretationen für die Typen stets nicht leer sind! Würde man leere Klassen oder Typen zulassen, so müßte man explizite Kontexte für alle im Beweis auftretenden Typvariablen und Termvariablen halten. Die Probleme, die bei Logiken mit leeren Typen auftreten, sind hinreichend bekannt. Analoge Probleme ergeben sich auch bei leeren Klassen. Speziell wären die Regeln [subst] und [type_inst] nicht korrekt, wenn leere Typen oder Klassen zugelassen wären.

Ich möchte nun die wesentlichen Argumente für die Korrektheit der einzelnen Regeln aufzählen.

Regeln hyp, refl, disch, mp: Die Korrektheit dieser Regeln folgt unmittelbar aus der Definition für Modelle, speziell aus der Interpretation für die Minimalsignatur Σ_{Min}.

Regel subst: Hier folgt die Korrekheit aus dem Substitutionslemma für Terme 2.22 und der Tatsache, daß die Träger von Typen nicht leer sind.

Regel type_inst: Die Korrektheit folgt aus dem Substitutionslemma für Typen 2.20 und aus der Tatsache, daß Klassen nicht leer sind.

Regeln beta und abstract: Ein Argument ist natürlich wieder, daß Typen und Klassen nicht leer sind. Zusätzlich wird hier aber noch die Interpretation der Gleichheit, sowie die Definition der Semantik der λ-Abstraktion und die Extensionalität von Funktionen eingesetzt.

2.6 Konservative Theorieerweiterung in HOLC

In diesem Abschnitt werde ich den Begriff der *konservativen Theorieerweiterung* formal einführen. Diese Art der Theorieerweiterung garantiert, daß die erweiterte Theorie ein Modell hat, wann immer die ursprüngliche Theorie ein Modell hat. Der Ausgangspunkt für alle Theorieerweiterungen ist die Minimaltheorie *Min*, von der sehr einfach gezeigt werden kann, daß sie ein Modell hat.

Theorem 2.27

Die Minimaltheorie *Min* hat ein Modell.

Beweis: Die Minimaltheorie $Min = (\Sigma_{Min}, \emptyset, \emptyset)$ hat per Definition keine Axiome. Daher genügt es zu zeigen, daß es Modelle für die Minimalsignatur Σ_{Min} gibt. Ein Modell für die Minimalsignatur besteht neben der Interpretation für die Konstanten der Minimalsignatur nur aus einem Modell für die minimale Typsignatur. Für all diese Komponenten sind die Interpretationen aber bereits in den Definitionen des jeweiligen Modellbegriffs festgelegt, und die Wohldefiniertheit der Interpretationen läßt sich aus der Existenz von Präuniversen ableiten.

Um den Begriff der konservativen Theorieerweiterung formulieren zu können, benötige ich den Begriff der *Modellrestriktion*. Wenn zwei Signaturen Σ_1 und Σ_2 mit $\Sigma_1 \subseteq \Sigma_2$ gegeben sind, dann ist die Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ eines Modells für Σ_2 im allgemeinen als dasjenige Modell für Σ_1 definiert, das aus $\mathcal{M}2$ entsteht, indem man nur die Interpretationen der Signatur Σ_1 beibehält und den Rest vergißt. Die Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ stützt sich dabei natürlich auf eine entsprechende Restriktion $\mathcal{T}\mathcal{M}2\,|_{\Omega_1}$ des Typmodells ab. Die Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ eines Modells für Σ_2 auf die Teilsignatur Σ_1 ist immer der Spezialfall eines Redukts [EGL89], wobei der Signaturmorphismus von Σ_1 nach Σ_2 eine Inklusion ist.

Leider ist in HOLC die Bildung der Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ nicht immer möglich. Dies liegt an den Interpretationen für Typklassen. Eine Typklasse $k \in K$ wird als Universum interpretiert, das nur Strukturen $X = (car(X), stru(X))$ der Klasse k enthält. Im Aufbau der Konstantenstruktur $stru(X)$ jedoch ist die vollständige Hierarchie aller Oberklassen von k kodiert. Wenn jetzt zum Beispiel k, k_1, k_2, k_3 mit $k \leq k_1 \leq k_2 \leq k_3$ Klassen der Signatur Σ_2 sind, und wenn in der Signatur Σ_1 etwa die Klasse k_2 fehlt, dann reicht es nicht aus, wenn für die Bildung der Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ einfach die Interpretation $k_2^{\mathcal{T}\mathcal{M}2}$ vergessen wird. Es müssten zudem alle Konstantenstrukturen aller Teilklassen von k_2, also insbesondere die Strukturen der Klassen k und k_1, umgebaut werden, damit sie ordnungsgemäße Strukturen bezüglich der Signatur Σ_1 sind. Ein solcher Umbau ist aber im allgemeinen nicht möglich, da dann auch die Interpretationen von Typkonstruktoren mitgeändert werden müssten. Aus diesem Grund ist eine allgemeine Definition der Restriktion von Modellen nicht möglich.

Bei allen Erweiterungen, die ich im folgenden betrachten werde, treten diese Schwierigkeiten aber nicht auf, und die Restriktion der Modelle kann in jedem Fall sehr einfach definiert werden, indem man die Interpretation der Σ_1 Signaturanteile von der Interpretation im Modell $\mathcal{M}2$ übernimmt und den Rest vergißt. Wenn die Restriktion möglich ist, dann ist sie in dieser uniformen Weise definiert. Sie muß nur eben für jede Form der Theorieerweiterung separat definiert werden.

In all den von mir vorgestellten Fällen der Theorieerweiterung kann die Restriktion $\mathcal{M}2\mid_{\Sigma_1}$ für Σ_2-Modelle definiert werden, und es läßt sich folgendes Restriktionslemma formulieren und durch Induktion über den Aufbau des Terms t beweisen.

Muster für das Restriktionslemma:

Wenn Σ_1, Σ_2 Signaturen sind mit $\Sigma_1 \subseteq \Sigma_2$, dann gilt für alle Terme $t \in DT_{\Sigma_1}$ und alle Belegungen ν, η mit η *sat* $(\nu, GC(t))$:

$$\mathcal{M}2[\![t]\!]^{\Sigma_2}_{\nu,\eta} = \mathcal{M}2\mid_{\Sigma_1}[\![t]\!]^{\Sigma_1}_{\nu,\eta}$$

Für alle Erweiterungen außer den Erweiterungen um eine neue Klasse (Abschnitt 2.6.3) bzw. eine neue Arität (Abschnitt 2.6.4) ist diese Aussage trivial und folgt direkt aus der Definition der Restriktion und der Tatsache, daß die zusätzlichen Signaturanteile der Theorie Th_2 in den Termen über der Signatur Σ_1 gar nicht vorkommen. Bei den Erweiterungen um eine neue Klasse bzw. eine neue Arität hingegen wird für einen bereits in der Signatur Σ_1 vorkommenden Typkonstruktor eine neue Überladung hinzugefügt. Um hier das Restriktionslemma zu zeigen, muß man die speziellen Eigenschaften von überladenen Typkonstruktoren ausnützen.

Die Begriffe der konservativen Theorieerweiterung und der konservativen Modellerweiterung werden nun definiert wie folgt:

Definition 2.54 *Konservative Theorieerweiterung, konservative Modellerweiterung*

Seien $Th_1 = (\Sigma_1, Ax_1, KAx_1)$ und $Th_2 = (\Sigma_2, Ax_2, KAx_2)$ Theorien mit $Th_1 \subseteq Th_2$ und sei die Restriktion $\mathcal{M}2\mid_{\Sigma_1}$ für Σ_2-Modelle definiert.

Die Theorie Th_2 ist genau dann eine konservative Erweiterung der Theorie Th_1, wenn es für *jedes* Modell $\mathcal{M}1$ der Theorie Th_1 ein Modell $\mathcal{M}2$ der Theorie Th_2 gibt, so daß:

$$\mathcal{M}2\mid_{\Sigma_1} = \mathcal{M}1$$

Wenn das Modell $\mathcal{M}2$ die obige Bedingung erfüllt, dann wird die Erweiterung des Modells $\mathcal{M}1$ zu $\mathcal{M}2$ als konservative Modellerweiterung bezeichnet.

Der Nachweis der Existenz des Modells $\mathcal{M}2$ wird im allgemeinen dadurch erbracht, daß man für ein beliebiges Modell $\mathcal{M}1$ konkrete Interpretationen für die neuen Signaturanteile aus Komponenten von $\mathcal{M}1$ konstruiert, so daß ein Modell für Th_2 entsteht. Der gerade eingeführte Begriff der konservativen Theorieerweiterung entspricht dem Begriff der *streng persistenten Erweiterung* von [EGL89], wo verlangt wird, daß das bzgl. dem Signaturmorphismus σ gebildete Redukt $\bar{\sigma}(\mathcal{M}2)$ *identisch* mit dem zugrundeliegenden Modell $\mathcal{M}1$ vor der Erweiterung ist, und nicht nur ein Isomorphismus zwischen den beiden Σ_1-Modellen existiert. In der Literatur über die Logik HOL [GM93] spricht man statt von einer konservativen Erweiterung einfach von *safe extension*.

Wenn Th_2 eine konservative Erweiterung der Theorie Th_1 ist, dann gilt folgende Eigenschaft, die der wesentliche Grund für die Bezeichnung 'konservativ' ist.

Theorem 2.28

Wenn Th_2 eine konservative Erweiterung der Theorie Th_1 ist, dann gilt für alle Formeln $p \in FORM_{\Sigma_1}$:

$$Th_1 \models p \Longleftrightarrow Th_2 \models p$$

Beweis: Folgt unmittelbar aus der Definition der konservativen Erweiterung und dem jeweiligen Restriktionslemma.

Wenn man im obigen Theorem das Zeichen $\models$ für die Gültigkeit durch das Zeichen $\vdash$ für die Herleitbarkeit einer Formel in einer Theorie ersetzt, dann erhält man für Logik erster Stufe eine äquivalente syntaktische Charakterisierung der konservativen Theorieerweiterung [And86]. Das syntaktische Kriterium garantiert, daß die Theorie Th_2 genau dann konsistent (nicht jede Formel ist ableitbar) ist, wenn die Theorie Th_1 konsistent ist. In der Logik erster Stufe ist jede konsistente Menge erfüllbar, und damit ist auch die Existenz von Modellen für die erweiterte Theorie garantiert. In der Logik höherer Stufe HOL gilt dieser Erweiterungssatz leider nicht, und daher muß die Konservativität über das oben angegebene Kriterium der streng persistenten Erweiterbarkeit definiert werden. Schließlich ist man primär an der Existenz von Modellen für die neue Theorie interessiert und nicht nur an deren Konsistenz. Die Existenz eines Modells garantiert umgekehrt natürlich die Konsistenz der Theorie.

2.6.1 Erweiterung durch Konstantendefinition

Die Theorieerweiterung durch eine Konstantendefinition ist die einfachste Form der Erweiterung. Diese Art der Erweiterung ist bereits für die Logik HOL bekannt, und der hier eingeführte Mechanismus entspricht genau dem in [GM93] beschriebenen.

Damit ich die Theorieanteile, die bei einer Erweiterung neu hinzukommen, besser beschreiben kann, werde ich für jede Form der Erweiterung eine spezielle Syntax einführen. Diese soll aber nicht als formaler Anteil der Logik verstanden werden, sondern dient einzig und allein dazu, die Erweiterung klarer ausdrücken zu können. Im Fall der Erweiterung durch eine Konstantendefinition werde ich zum Beispiel die Syntax

$$Th_2 = Th_1 +_{const} \langle c \equiv t \rangle$$

verwenden. Diese Notation ist an [GM93] angelehnt.

Die Theorieerweiterung durch eine Konstantendefinition wird durch die nun folgenden Definitionen charakterisiert.

Definition 2.55 *Erweiterung durch Konstantendefinition*

Sei $Th_1 = (\Sigma_1, Ax_1, KAx_1)$ eine Theorie mit $\Sigma_1 = (\Omega_1, C_1, KS_1)$. Die Erweiterung der Theorie Th_1 durch eine Konstantendefinition wird dann folgendermaßen notiert:

$$Th_2 = Th_1 +_{const} \langle c \equiv t \rangle$$

Dabei müssen die folgenden syntaktischen Bedingungen erfüllt sein:

B1: c ist ein neues Konstantensymbol bzgl. der Konstantenmenge C_1.

B2: $t \in DT_{\Sigma_1}$ ist ein vollgetypter Term über der Signatur Σ_1 mit $\tau = GT_{\Sigma_1}(t)$ und es gilt:

$$GC_{\Sigma_1}(t) = \emptyset \wedge TV(t) = TV(\tau)$$

t ist also ein geschlossener Term, und alle in t vorkommenden Typvariablen kommen auch in seinem generischen Typ vor. In [GM93] wird ausführlich auf die Wichtigkeit dieser Bedingung hingewiesen. Sie ist die Grundvoraussetzung für die Wohldefiniertheit der weiter unten angegebenen Modellerweiterung.

Die Komponenten der neuen Theorie Th_2 sind dann definiert wie folgt:

$$\begin{aligned} C_2 &= C_1 \cup \{(c, \tau)\} \\ \Sigma_2 &= (\Omega_1, C_2, KS_1) \\ Ax_2 &= Ax_1 \cup \{c{:}\tau = t\} \\ Th_2 &= (\Sigma_2, Ax_2, KAx_1) \end{aligned}$$

Offensichtlich ist Th_2 eine Theorie im Sinne der Definition 2.21.

Wie in der Einleitung zu diesem Abschnitt bereits erwähnt, werde ich für jede Erweiterung die zugehörige Restriktion der Modelle definieren. Im Fall der Erweiterung um eine Konstantendefinition ist dies sehr einfach.

Definition 2.56 *Restriktion bei* $+_{const}$

Sei $Th_2 = Th_1 +_{const} \langle c \equiv t \rangle$ und sei $\mathcal{M}2$ ein Modell für die Signatur Σ_2. Dann ist die Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ dasjenige Modell für Σ_1, das aus $\mathcal{M}2$ entsteht, wenn man die Interpretation $c^{\mathcal{M}2}$ wegläßt.

Die Tatsache, daß $\mathcal{M}2\,|_{\Sigma_1}$ ein Modell für Σ_1 ist, ist dabei offensichtlich.

Aus der Definition für die Restriktion folgt unmittelbar das zugehörige Restriktionslemma.

Theorem 2.29 *Restriktionslemma bei* $+_{const}$

Sei $Th_2 = Th_1 +_{const} \langle c \equiv t \rangle$ und sei $\mathcal{M}2$ ein Modell für die Signatur Σ_2. Dann gilt für alle Terme $t \in DT_{\Sigma_1}$ und alle Belegungen ν, η mit $\eta\ sat\ (\nu, GC(t))$:

$$\mathcal{M}2[\![t]\!]^{\Sigma_2}_{\nu,\eta} = \mathcal{M}2\,|_{\Sigma_1}[\![t]\!]^{\Sigma_1}_{\nu,\eta}$$

Beweis: leichte Induktion über den Aufbau von t.

Die Modellerweiterung bei der Erweiterung durch eine Konstantendefinition ist ebenso einfach.

Definition 2.57 *Modellerweiterung bei* $+_{const}$

Sei $Th_2 = Th_1 +_{const} \langle c \equiv t \rangle$ und sei $\mathcal{M}1 = (\mathcal{TM}, \mathcal{C}_1)$ ein Modell für die Theorie Th_1. Ein Modell $\mathcal{M}2$ für Σ_2 benötigt eine zusätzliche Interpretation für die Konstante c.

Sei ohne Beschränkung der Allgemeinheit

$$TV(\tau) = \{\alpha_1, \ldots, \alpha_m\}, \quad m \geq 0, \quad \alpha_1 \prec \ldots \prec \alpha_m$$

Die Interpretation $c^{\mathcal{M}2}$ für die neue Konstante c im Modell $\mathcal{M}2$ wird dann eindeutig durch folgende Definition festgelegt:

Für alle Belegungen ν, η mit $\eta \; sat \, (\nu, GC(t))$ soll gelten[21]:

$$c^{\mathcal{M}2}(\nu(\alpha_1), \ldots, \nu(\alpha_m)) = \mathcal{M}1[\![t]\!]^{\Sigma_1}_{\nu,\eta}$$

Das Modell $\mathcal{M}2$ entsteht aus $\mathcal{M}1$ durch Hinzufügen der Interpretation $c^{\mathcal{M}2}$:

$$\mathcal{M}2 = (\mathcal{TM}, \mathcal{C}_1 \cup \{c^{\mathcal{M}2}\})$$

Offensichtlich ist $\mathcal{M}2$ ein Modell für Σ_2 und es gilt $\mathcal{M}2\,|_{\Sigma_1} = \mathcal{M}1$. Durch Verwendung der Interpretation von c sieht man ebenso leicht, daß $\mathcal{M}2$ ein Modell der Theorie Th_2 ist. Zusätzlich zu den Axiomen der Theorie Th_1 muß ja nur die Gleichung $c{:}\tau = t$ gültig sein.

2.6.2 Die Axiome der Logik HOLC

Mit Hilfe der Erweiterung durch Konstantendefinition kann nun die minimale Theorie *Min* zur Theorie *Log* erweitert werden. In *Log* werden die üblichen Junktoren und Quantoren der Prädikatenlogik eingeführt, sowie zusätliche Abkürzungen, die die Formulierung der Axiome der Logik höherer Stufe in der auf *Log* aufbauenden Theorie *Init* vereinfachen. Alle folgenden Definitionen sind bis auf kleine notationelle Unterschiede [GM93] entnommen, da die Logik HOLC sich in diesen Punkten nicht von HOL unterscheidet. Die Terme in den folgenden Definitionen seien wie üblich mittels *dec* vollgetypt.

Definition 2.58 *Theorie Log*

Die Theorie *Log* entsteht aus der Theorie *Min* durch mehrfache Erweiterung per Konstantendefinition.

$$\begin{aligned}
Log = \; & Min \\
+_{const} \; & \langle True \equiv (\lambda x{:}bool.\, x) = (\lambda x{:}bool.\, x) \rangle \\
+_{const} \; & \langle \forall \equiv \lambda P.\, (P = (\lambda x.\, True)) \rangle \\
+_{const} \; & \langle \exists \equiv \lambda P.\, P(\varepsilon x.\, P) \rangle \\
+_{const} \; & \langle False \equiv \forall P.\, P \rangle \\
+_{const} \; & \langle \neg \equiv \lambda P.\, P \rightarrow False \rangle \\
+_{const} \; & \langle \wedge \equiv \lambda PQ.\forall R.\, (P \rightarrow Q \rightarrow R) \rightarrow R \rangle \\
+_{const} \; & \langle \vee \equiv \lambda PQ.\forall R.\, (P \rightarrow R) \rightarrow (Q \rightarrow R) \rightarrow R \rangle \\
+_{const} \; & \langle inj \equiv \lambda f.\forall xy.\, fx = fy \rightarrow x = y \rangle \\
+_{const} \; & \langle surj \equiv \lambda f.\forall y.\exists x.\, y = fx \rangle
\end{aligned}$$

[21] η wird eigentlich gar nicht benötigt, da t ein geschlossener Term ist.

In der obigen Definition wurden wieder einige Schreiberleichterungen für die neu eingeführten Konstanten benutzt. Sie sind in der folgenden Tabelle mit den zugehörigen Prioritäten aufgeführt.

	Extern	Intern	Mixfix-Priorität
Allquantor	$\forall x.\, P$	$(\forall(\lambda x.\, P))$	Binder 10
Existenzquantor	$\exists x.\, P$	$(\exists(\lambda x.\, P))$	Binder 10
Konjunktion	$t_1 \wedge t_2$	$((\wedge\, t_1)t_2)$	Rechts 35
Disjunktion	$t_1 \vee t_2$	$((\vee\, t_1)t_2)$	Rechts 30

Aufbauend auf der Theorie *Log* kann nun die Theorie *Init* eingeführt werden, mit der dann die Formalisierung der Logik höherer Stufe HOLC abgeschlossen ist. Die Logik höherer Stufe HOLC setzt sich also aus dem Deduktionssystem D_{Min} und der Theorie *Init* zusammen. Die Definitionen für Präuniversen, Typmodelle und Modelle sind so abgefaßt, daß die nachfolgenden Axiome bereits in allen Modellen der Theorie *Log* gelten. Die Axiome dienen nur dazu, die Eigenschaften der Logik höherer Stufe syntaktisch verfügbar zu machen. Diese zusätzlichen Axiome ändern allerdings nichts an der Tatsache, daß die Logik HOLC unvollständig ist! Eine ausführliche Behandlung der Vollständigkeitsproblematik findet sich [And86]. Die dort untersuchte Logik Q_0^∞ entspricht, abgesehen vom polymorphen Typsystem, der Logik HOL von Gordon [Gor85, GM93] und der hier vorgestellten Erweiterung HOLC.

Definition 2.59 *Theorie Init, Standardtheorien, Standardmodelle*

Die Theorie *Init* entsteht aus der Theorie *Log* durch Hinzufügen der folgenden 5 Axiome. Die Axiome sind wieder [GM93] entnommen.

$\forall P.\, P = \mathit{True} \vee P = \mathit{False}$	True_or_False
$\forall PQ.\, (P \to Q) \to (Q \to P) \to P = Q$	iff
$\forall f.\, (\lambda x.\, fx) = f$	eta
$\forall Px.\, Px \to P(\varepsilon x.\, Px)$	choice
$\exists f : \mathit{ind} \Rightarrow \mathit{ind}.\, (\mathit{inj}\, f) \wedge (\neg(\mathit{surj}\, f))$	infinity

Jede Theorie, die eine Theorieerweiterung von *Init* ist, wird als Standardtheorie bezeichnet. Modelle von Standardtheorien werden als Standardmodelle bezeichnet. Im folgenden werden nur noch Standardtheorien betrachtet.

Die Definitionen für Präuniversen, Typmodelle und Modelle sind so abgefaßt, daß folgendes Theorem gilt:

Theorem 2.30

Jedes Modell der Theorie *Log* erfüllt bereits die Axiome der Theorie *Init*. Die Theorie *Init* ist damit trivialerweise eine konservative Erweiterung der Theorie *Log*.

Beweis: folgt unmittelbar aus den Eigenschaften von Modellen und den Konstantendefinitionen der Theorie *Log*. Ein ausführlicher Beweis findet sich in [GM93].

Per Definition sind Standardtheorien Erweiterungen der Theorie *Init* und somit auch Erweiterungen von *Log*. Die Definition der Konstante *False* führt jedoch dazu, daß *False* in jedem Standardmodell durch das Element $\mathbb{F} \in \mathit{Two}$ interpretiert wird. Daraus ergibt sich sehr leicht der Beweis für folgendes Theorem.

Theorem 2.31 *Konsistenz von Standardtheorien*

Wenn eine Standardtheorie ein Modell hat, dann ist sie auch konsistent.

Eine Theorie heißt dabei konsistent, wenn in ihr nicht jede Formel ableitbar ist. In Standardtheorien ist dies äquivalent dazu, daß die Sequenz $\emptyset \blacktriangleright \mathit{False}$ nicht herleitbar ist.

Beweis: Wegen der Korrektheit des Deduktionssystems D_{Min} würde aus der Herleitbarkeit der Sequenz $\emptyset \blacktriangleright \mathit{False}$ bereits deren Gültigkeit folgen. Die Sequenz $\emptyset \blacktriangleright \mathit{False}$ ist jedoch in keinem Standardmodell gültig, da sonst das Element $\mathbb{F}$ identisch mit dem Element $\mathbb{T}$ sein müßte!

Der Satz kann natürlich auch für Theorien formuliert werden, in denen die Konstante *False* noch nicht definiert ist, etwa für die Theorie *Min*. Man verwendet im oben angedeuteten Widerspruchsbeweis statt der Herleitbarkeit der Formel *False* einfach den Term, durch den die Konstante *False* in der Theorie *Log* definiert wird. Der Widerspruch ergibt sich im Endeffekt immer dadurch, daß in allen Modellen die Elemente $\mathbb{F}$ und $\mathbb{T}$ unterschiedlich sein müssen.

2.6.3 Erweiterung durch eine Klasse

In diesem Abschnitt werde ich die konservative Theorieerweiterung durch eine neue Klasse beschreiben. Die Idee, die hinter dieser Erweiterung steckt, wurde bereits in Abschnitt 2.2 beschrieben. Hier folgt nun die formale Behandlung der Erweiterung. Mit diesem Erweiterungsmechanismus kann nur eine einzelne Klasse hinzugefügt werden, und es darf nur eine unmittelbare Oberklasse angegeben werden. Auf diese Weise können nur baumartige Klassenhierarchien aufgebaut werden. Eine allgemeinere Art der Erweiterung mit mehreren unmittelbaren Oberklassen ist durchaus denkbar, allerding wird dadurch die Modellerweiterung erheblich kompliziert. Da ich für die Entwicklung von HOLCF nur eine lineare Klassenhierarchie benötigt habe, genügt aber der hier vorgestellte Erweiterungsmechanismus vollständig.

Die Theorieerweiterung durch eine neue Klasse wird durch folgende Definition charakterisiert.

Definition 2.60 *Erweiterung durch eine neue Klasse*

Sei $Th_1 = (\Sigma_1, Ax_1, KAx_1)$ eine Theorie mit Signatur $\Sigma_1 = (\Omega_1, C_1, KS_1)$ und Typsignatur $\Omega_1 = (K_1, \leq_1, TC_1)$. Die Erweiterung der Theorie Th_1 durch eine neue Klasse wird dann folgendermaßen notiert:

$$Th_2 = Th_1 +_{class}$$
$$\text{class } k2 \leq k1$$
$$\text{classvar } \alpha_{k2}$$

$$
\begin{array}{l}
\text{classconsts } KS_{k2} = \{c_1 : \tau_1, \ldots, c_n : \tau_n\} \\
\text{classaxioms } KAx_{k2} = \{ax_1, \ldots, ax_m\} \\
\text{witness } tc : k1 \\
\text{instmap } I : \begin{array}{lcl} \alpha_{k2} & \mapsto & tc \\ c_1 & \mapsto & \widetilde{c_1} \\ & \vdots & \\ c_n & \mapsto & \widetilde{c_n} \end{array}
\end{array}
$$

Damit die Theorieerweiterung durchgeführt werden kann, müssen einige syntaktische Bedingungen erfüllt sein. Um diese formulieren zu können, werden gleich die Signaturen für die neue Theorie Th_2 mit ihren jeweiligen Komponenten eingeführt:

$$
\begin{array}{rcl}
K_2 & = & K_1 \cup \{k2\} \\
\leq_2 & = & (\leq_1 \cup \{(k2, k1)\})^\star \\
TC_2 & = & TC_1 \cup \{tc : k2\}
\end{array}
$$

$$
\begin{array}{rcl}
\Omega_2 & = & (K_2, \leq_2, TC_2) \\
C_2 & = & C_1 \cup KS_{k2} \\
KS_2 & = & (KS_k)_{k \in K2}
\end{array}
$$

$$
\Sigma_2 = (\Omega_2, C_2, KS_2)
$$

Dabei bedeutet die Anwendung des Operators $(\_)^\star$ die Bildung der transitiven und reflexiven Hülle einer Relation. Dies ist hier notwendig, da die Klassenordnung $\leq_2$ wieder eine partielle Ordnung sein muß. Die syntaktischen Bedingungen sind nun wie folgt:

B1: $k2 \notin K_1$ und $k1 \in K_1$

B2: $\forall i \in \{1, \ldots, n\}.\ \tau_i \in T_{\Omega_2 \setminus k2} \wedge TV(\tau_i) = \{\alpha_{k2}\} \wedge (\neg \exists \rho.\ (c_i, \rho) \in C_1)$

B3: $\forall ax \in KAx_{k2}.\ ax \in FORM_{\Sigma_2 \setminus k2} \wedge TV(ax) = \{\alpha_{k2}\}$

B4: $tc : k1 \in TC_1 \wedge k1 = mincodom(tc, [])$

B5: $\forall i \in \{1, \ldots, n\}.\ I(\tau_i) \in T_{\Omega_1} \wedge (\widetilde{c_i}, I(\tau_i)) \in C_1$, wobei die Instanzabbildung I in offensichtlicher Weise auf Typterme fortgesetzt wird.

B6: $\forall ax \in KAx_{k2}.\ (Th_1, D_{Min}) \vdash I(ax)$, wobei die Instanzabbildung I in offensichtlicher Weise auf Terme fortgesetzt wird.

Die neue Theorie Th_2 wird dann definiert wie folgt:

$$
\begin{array}{rcl}
Ax_2 & = & Ax_1 \cup KAx_{k2} \cup \\
 & & \{c_1 : I(\tau_1) = \widetilde{c_1} : I(\tau_1), \ldots, c_n : I(\tau_n) = \widetilde{c_n} : I(\tau_n)\} \\
KAx_2 & = & (KAx_k)_{k \in K2}
\end{array}
$$

$$
Th_2 = (\Sigma_2, Ax_2, KAx_2)
$$

Die Bedingung B1 besagt, daß $k2$ ein neuer Klassenbezeichner ist und die Einordnung der neuen Klasse $k2$ in die Klassenhierarchie der Theorie Th_1 möglich ist. Bedingungen B2 und B3 garantieren die in den Definitionen für Signaturen und Theorien geforderten Eigenschaften für charakteristische Konstanten und charakteristische Axiome. Weiterhin wird durch B2 gefordert, daß die charakteristischen Konstanten c_i neue Konstanten sind. Bedingung B4 sichert die Regularität der neuen Typsignatur Ω_2. Aus Bedingung B5 folgt die Wohlgeformtheit der Instanzaxiome in Ax_2. Die Bedingung B6 ist von zentraler Bedeutung für die Erweiterbarkeit von Modellen der Theorie Th_1. Nach dieser Bedingung sind die charakteristischen Axiome der neuen Klasse $k2$ in allen Modellen der Theorie Th_1 gültig, wenn sie für den Zeugen $tc{:}k1$ spezialisiert werden. Die Interpretation der neuen Arität $tc{:}k2$ für den Typkonstruktor tc wird in der Modellerweiterung gerade so definiert, daß sich aus $(Th_1, D_{Min}) \vdash I(ax)$ die Gültigkeit $Th_2 \models ax$ der charakteristischen Axiome in Th_2 folgern läßt. Die Bedingungen garantieren also zum einen, daß Ω_2 und Σ_2 wohlgeformte Signaturen sind bzw. Th_2 eine wohlgeformte Theorie ist, zum anderen ermöglichen sie aber auch die konservative Modellerweiterung.

Als Beispiel für die Einführung einer neuen Klasse möchte ich hier nochmals die Einführung der Klasse *po* aus der informellen Einführung in Abschnitt 2.2 bringen. In der eben definierten Notation für die Klassenerweiterung sieht das Beispiel folgendermaßen aus:

$$
\begin{array}{l}
Porder = Base{+}_{class} \\
\quad \text{class } po \leq top \\
\quad \text{classvar } \alpha_{po} \\
\quad \text{classconsts } KS_{po} = \{\, \sqsubseteq : \alpha_{po} \Rightarrow \alpha_{po} \Rightarrow bool \,\} \\
\quad \text{classaxioms } KAx_{po} = \ \{ \\
\qquad\qquad\qquad\qquad x \sqsubseteq x, \\
\qquad\qquad\qquad\qquad x \sqsubseteq y \wedge y \sqsubseteq x \rightarrow x = y, \\
\qquad\qquad\qquad\qquad x \sqsubseteq y \wedge y \sqsubseteq z \rightarrow x \sqsubseteq z \\
\qquad\qquad\qquad\qquad \} \\
\quad \text{witness } void{:}top \\
\quad \text{instmap } I : \ \alpha_{po} \ \mapsto \ void \\
\qquad\qquad\qquad\quad \sqsubseteq \ \mapsto \ less\_void
\end{array}
$$

Die Restriktion von Σ_2-Modellen auf die Signatur Σ_1 wird folgendermaßen definiert.

Definition 2.61 *Restriktion bei* $+_{class}$

Sei die Theorie Th_2 durch eine Erweiterung um eine neue Klasse mittels $+_{class}$ entstanden, und sei $\mathcal{M}2$ ein Modell für die Signatur Σ_2. Dann ist die Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ dasjenige Modell für Σ_1, das aus $\mathcal{M}2$ entsteht, wenn man im Typmodell $\mathcal{T}\mathcal{M}2$ die Interpretation $k2^{\mathcal{T}\mathcal{M}2}$ für das Universum der Klasse $k2$ und die Interpretation $tc_{k2}^{\mathcal{T}\mathcal{M}2}$ für die neue Überladung $tc{:}k2$ entfernt, sowie im Modell $\mathcal{M}2$ die Interpretationen für die charakteristischen Konstanten $c_1^{\mathcal{M}2}, \ldots, c_n^{\mathcal{M}2}$ wegläßt.

Die Tatsache, daß $\mathcal{M}2\,|_{\Sigma_1}$ ein Modell für Σ_1 ist, ist wieder offensichtlich.

Aus der Definition für die Restriktion folgt durch leichte Induktion das folgende Restriktionslemma, welches diesmal auch für Typterme formuliert werden muß, da die Typsignatur bei $+_{class}$ ebenfalls verändert wird.

Theorem 2.32 *Restriktionslemma bei* $+_{class}$

Sei die Theorie Th_2 durch eine Erweiterung um eine neue Klasse mittels $+_{class}$ entstanden, und sei $\mathcal{M}2 = (\mathcal{TM}2, \mathcal{C}2)$ ein Modell für die Signatur Σ_2.

Dann gilt für alle Typterme $\tau \in T_{\Omega_1}$ und Belegungen ν:

$$fgt_{c,k}(\mathcal{TM}2[\![\tau]\!]_\nu^{\Omega_2}) = \mathcal{TM}2\mid_{\Omega_1}[\![\tau]\!]_\nu^{\Omega_1}$$

wobei

$$c = lst_{\Omega_2}(\tau) \quad \text{und} \quad k = lst_{\Omega_1}(\tau)$$

Weiterhin gilt für alle Terme $t \in DT_{\Sigma_1}$ und alle Belegungen ν, η mit $\eta\ sat\,(\nu, GC(t))$:

$$\mathcal{M}2[\![t]\!]_{\nu,\eta}^{\Sigma_2} = \mathcal{M}2\mid_{\Sigma_1}[\![t]\!]_{\nu,\eta}^{\Sigma_1}$$

Beweis: leichte Induktion über den Aufbau der Terme τ bzw. t.

Nun wollen wir uns der schrittweisen Konstruktion eines Modells für die Theorie Th_2 zuwenden, wenn Th_2 aus Th_1 durch $+_{class}$ entsteht und ein Modell $\mathcal{M}1$ für Th_1 gegeben ist. Sei dabei $\mathcal{M}1 = (\mathcal{TM}1, \mathcal{C}1)$ mit $\mathcal{TM}1 = (\mathcal{K}1, \mathcal{TC}1)$.

Schritt 1

Zuerst definieren wir ein Universum $k2^{\mathcal{TM}2_1}$ für die neue Klasse $k2$. Dieses Universum besteht nur aus Strukturen der Klasse $k2$ (vgl. Definition 2.28). Der Index 1 bedeutet, daß es sich dabei um den ersten Schritt der Konstruktion handelt. In weiteren Schritten wird dieses Universum dann verkleinert, so daß am Schluß nur noch klassengültige Strukturen der Klasse $k2$ enthalten sind. Das Universum $k2^{\mathcal{TM}2_1}$ ist definiert wie folgt:

$$k2^{\mathcal{TM}2_1} = \{(car(X), (const, \{k1 \mapsto stru(X)\})) \mid \text{ Bedingung } \}$$

wobei

$$\text{Bedingung } = \begin{cases} X \in k1^{\mathcal{TM}1} \text{ und} \\ const \subseteq \text{ CID } \times \bigcup_{Y \in PU} Y \\ \quad \text{ist eine endliche Abbildung mit} \\ \quad const(c) \text{ definiert } \Leftrightarrow \exists \tau.\ c{:}\tau \in KS_{k2} \end{cases}$$

Das Universum $k2^{\mathcal{TM}2_1}$ ist per Konstruktion eine nichtleere Kollektion von Strukturen der Klasse $k2$. Die Konsistenz der Konstantenstrukturen folgt dabei aus der Tatsache, daß $k1$ der einzige unmittelbare Nachfolger von $k2$ ist.

Weiterhin definieren wir die Interpretation $tc_{k2}^{\mathcal{TM}2_1}$ für die neue Überladung $tc{:}k2$. Für beliebiges[22] ν definieren wir:

$$tc_{k2}^{\mathcal{TM}2_1} = Z$$

[22] ν ist hier unerheblich, da alle weiter unten auftretenden Typterme variablenfrei sind.

wobei:

$$\begin{array}{lcl} Z & = & (car(\widehat{Z}), (const, \{k1 \mapsto stru(\widehat{Z})\})) \\ \widehat{Z} & = & tc_{k1}^{\mathcal{TM}1} \\ const & = & c_1 \mapsto \mathcal{M}1[\![I(c_1{:}\tau_1)]\!]_\nu^{\Sigma_1} \\ & & \vdots \\ & & c_n \mapsto \mathcal{M}1[\![I(c_n{:}\tau_n)]\!]_\nu^{\Sigma_1} \end{array}$$

Per Konstruktion ist $tc_{k2}^{\mathcal{TM}2_1}$ eine Struktur der Klasse $k2$ und nach Definition von $k2^{\mathcal{TM}2_1}$ ebenfalls ein Element dieses Universums.

Nun können wir ein erstes Modell $\mathcal{TM}2_1$ für die Typsignatur Ω_2 definieren[23].

$$\begin{array}{lcl} \mathcal{TC}2_1 & = & \mathcal{TC}1 + tc_{k2}^{\mathcal{TM}2_1} \\ \mathcal{TM}2_1 & = & (\mathcal{K}1 + k2^{\mathcal{TM}2_1}, \mathcal{TC}2_1) \end{array}$$

Per Konstruktion erfüllt $\mathcal{TM}2_1$ alle Eigenschaften für ein Ω_2-Modell.

Schritt 2

Das Universum $k2^{\mathcal{TM}2_1}$ enthält sicher zuviele Strukturen, denn es wurden bisher keinerlei Einschränkungen bezüglich der typkorrekten Interpretation der charakteristischen Konstanten der Klasse $k2$ gemacht. In einem weiteren Schritt schränken wir $k2^{\mathcal{TM}2_1}$ derart ein, daß nur noch typkorrekte Strukturen der Klasse $k2$ verbleiben. Wir verkleinern das Universum $k2^{\mathcal{TM}2_1}$ und definieren:

$$k2^{\mathcal{TM}2_2} = \{X \in k2^{\mathcal{TM}2_1} \mid X \; tcor\,(\mathcal{TM}2_1, KS_{k2})\}$$

wobei $X \; tcor\,(\mathcal{TM}2_1, KS_{k2})$ die Bedingungen für typkorrekte Strukturen gemäß Definition 2.35 beschreibt.

Das verkleinerte Universum $k2^{\mathcal{TM}2_2}$ ist nicht leer! Diese Aussage beweisen wir, indem wir zeigen, daß die Interpretation $tc_{k2}^{\mathcal{TM}2_1}$ ein Element dieses Universums ist.

Theorem 2.33 $tc_{k2}^{\mathcal{TM}2_1} \in k2^{\mathcal{TM}2_2}$

Um diese Aussage zu beweisen, müssen wir zeigen, daß die Bedingungen für typkorrekte Strukturen gemäß Definition 2.35 von $tc_{k2}^{\mathcal{TM}2_1}$ erfüllt werden. Neben Bedingungen, die bereits trivialerweise per Konstruktion erfüllt sind, bleibt zu zeigen:

Für alle charakteristischen Konstanten $c{:}\tau \in KS_{k2}$ und Belegungen ν soll gelten:

$$const(Z)(c) \in car(\mathcal{TM}2_1[\![\tau]\!]_{\nu[Z/\alpha]}^{\Omega_2 \setminus k2})$$

wobei Z für $tc_{k2}^{\mathcal{TM}2_1}$ steht und α die Klassenvariable der Klasse $k2$ ist.

[23] Ich schreibe in den folgenden Definitionen kurz + für die jeweils an dieser Stelle angebrachte Erweiterung der einzelnen Strukturen. Dies macht die Notation erheblich lesbarer.

Aus der Definition für $Z = tc_{k2}^{\mathcal{TM}2_1}$ wissen wir:

$$const(Z)(c) = \mathcal{M}1[\![I(c:\tau)]\!]_\nu^{\Sigma_1}$$

und mit Theorem 2.18 über die Typkorrektheit der Interpretationsfunktion folgt dann

$$const(Z)(c) \in car(\mathcal{TM}1[\![I(\tau)]\!]_\nu^{\Omega_1})$$

Es reicht also zu zeigen:

$$car(\mathcal{TM}2_1[\![\tau]\!]_{\nu[Z/\alpha]}^{\Omega_2 \setminus k2}) = car(\mathcal{TM}1[\![I(\tau)]\!]_\nu^{\Omega_1})$$

Nun wissen wir, daß $\mathcal{TM}2_1$ ein Typmodell für Ω_2 ist und daß $Z = tc_{k2}^{\mathcal{TM}2_1}$ sicher in $k2^{\mathcal{TM}2_1}$. Weiterhin ist der Teil der Instanzabbildung I, der die Klassenvariable α auf den Zeugen tc abbildet, bezüglich der Signatur Ω_2 eine zulässige Typsubstitution, da $tc:k2 \in TC_2$. Wir dürfen also das Substitutionslemma 2.16 anwenden. Es gilt folgende Gleichungskette, die den Beweis abschließt:

$$\begin{aligned}
&car(\mathcal{TM}2_1[\![\tau]\!]_{\nu[Z/\alpha]}^{\Omega_2 \setminus k2}) \\
&\qquad = \quad \text{(Theorem 2.17 und Carrier-Lemma 2.10)} \\
&car(\mathcal{TM}2_1[\![\tau]\!]_{\nu[Z/\alpha]}^{\Omega_2}) \\
&\qquad = \quad \text{(Substitutionslemma 2.16 und Carrier-Lemma 2.10)} \\
&car(\mathcal{TM}2_1[\![I(\tau)]\!]_\nu^{\Omega_2}) \\
&\qquad = \quad \text{(Restriktionslemma 2.32 und Carrier-Lemma 2.10)} \\
&car(\mathcal{TM}1[\![I(\tau)]\!]_\nu^{\Omega_1})
\end{aligned}$$

Nun können wir das Typmodell $\mathcal{TM}2_1$ verfeinern, indem wir das Universum $k2^{\mathcal{TM}2_1}$ durch das angemessenere Universum $k2^{\mathcal{TM}2_2}$ ersetzen. Der gerade geführte Beweis garantiert uns, daß dies wieder zu einem Modell für Ω_2 führt.

$$\mathcal{TM}2_2 \quad = \quad (\mathcal{K}1 + k2^{\mathcal{TM}2_2}, \mathcal{TC}2_1)$$

Das neue Typmodell $\mathcal{TM}2_2$ ist die Basis für ein erstes Σ_2-Modell. Um ein solches zu erhalten müssen wir die Interpretationen für die charakteristischen Konstanten $c:\tau \in KS_{k2}$ festlegen.

Für alle Strukturen $X \in k2^{\mathcal{TM}2_2}$ und alle charakteristischen Konstanten $c:\tau \in KS_{k2}$ legen wir fest:

$$c^{\mathcal{M}2_2}(X) = const(X)(c)$$

Ein Modell für die Signatur Σ_2 ergibt sich dann durch folgende Definition:

$$\mathcal{M}2_2 = (\mathcal{TM}2_2, \mathcal{C}2_2)$$

wobei $\mathcal{C}2_2 = \mathcal{C}1 + \{c^{\mathcal{M}2_2} \mid \exists \tau.\, c:\tau \in KS_{k2}\}$

Per Konstruktion erfüllt $\mathcal{M}2_2$ die Eigenschaften aus der Definition 2.36 für Signaturmodelle.

Schritt 3

In einem letzten Schritt schränken wir das Universum $k2^{\mathcal{TM}2_2}$ noch weiter ein, so daß nur noch klassengültige Strukturen im Universum verbleiben, d.h. nur solche Strukturen, für die die charakteristischen Axiome der Klasse $k2$ gelten.

Wir verkleinern das Universum $k2^{\mathcal{TM}2_2}$ noch weiter und definieren:

$$k2^{\mathcal{TM}2} = \{X \in k2^{\mathcal{TM}2_2} \mid X \; cval\,(\mathcal{M}2_2, KAx_{k2})\}$$

wobei $X \; cval\,(\mathcal{TM}2_2, KAx_{k2})$ die Bedingungen für klassengültige Strukturen gemäß Definition 2.42 beschreibt.

Ein weiterer Index 3 ist in der obigen Definition nicht mehr nötig, da es sich um den letzten Schritt der Erweiterung handelt. Per Definition enthält das Universum $k2^{\mathcal{TM}2}$ nur noch Strukuren X, für die die charakteristischen Axiome der Klasse $k2$ gelten. Es stellt sich die Frage, ob das Universum $k2^{\mathcal{TM}2}$ nicht leer ist. Diese Frage können wir aber positiv beantworten, indem wir wieder zeigen, daß die Interpretation $tc_{k2}^{\mathcal{TM}2_1}$ ein Element dieses Universums ist.

Theorem 2.34 $\quad tc_{k2}^{\mathcal{TM}2_1} \in k2^{\mathcal{TM}2}$

Um diese Aussage zu beweisen, müssen wir zeigen, daß die Bedingungen für klassengültige Strukturen gemäß Definition 2.42 von $tc_{k2}^{\mathcal{TM}2_1}$ erfüllt werden. Wenn wir wieder die Abkürzung $Z = tc_{k2}^{\mathcal{TM}2_1}$ wählen und wenn α die Klassenvariable der Klasse $k2$ ist, müssen wir zeigen:

Für alle $ax \in KAx_{k2}$ mit generischem Kontext $\Gamma = GC(ax)$ und für alle Belegungen ν, η mit $\eta \; sat\,(\nu[Z/\alpha], \Gamma)$ muß gelten:

$$\mathcal{M}2_2[\![ax]\!]^{\Sigma_2 \setminus k2}_{\nu[Z/\alpha],\eta} = \mathbb{T}$$

Wegen Bedingung B6 aus Definition 2.60 und der Korrektheit des Herleitungsbegriffs reicht es aber zu zeigen:

$$\mathcal{M}2_2[\![ax]\!]^{\Sigma_2 \setminus k2}_{\nu[Z/\alpha],\eta} = \mathcal{M}1[\![I(ax)]\!]^{\Sigma_1}_{\nu,\eta}$$

Wir bilden wieder eine Gleichungskette:

$$\begin{aligned}
&\mathcal{M}2_2[\![ax]\!]^{\Sigma_2 \setminus k2}_{\nu[Z/\alpha],\eta} \\
&\qquad = \quad \text{(Theorem 2.21)} \\
&\mathcal{M}2_2[\![ax]\!]^{\Sigma_2}_{\nu[Z/\alpha],\eta} \\
&\qquad = \quad \text{(Hilfslemma 2.35 siehe unten)} \\
&\mathcal{M}2_2[\![I(ax)]\!]^{\Sigma_2}_{\nu,\eta} \\
&\qquad = \quad \text{(Restriktionslemma 2.32)} \\
&\mathcal{M}1[\![I(ax)]\!]^{\Sigma_1}_{\nu,\eta}
\end{aligned}$$

Im obigen Beweis haben wir die aus Schritt 2 bekannte Eigenschaft ausgenützt, daß $\mathcal{M}2_2$ ein Modell für die Signatur Σ_2 ist. Weiterhin wurde das folgende technische Hilfslemma benutzt, das fast wie das Substitutionslemma 2.22 aussieht. Es unterscheidet sich jedoch dadurch, daß die Instanzabbildung I auf Termebene Konstanten ersetzt und nicht Variable wie bei einer normalen Substitution.

Theorem 2.35

Im Kontext des obigen Beweises gilt:

$$\mathcal{M}2_2[\![ax]\!]^{\Sigma_2}_{\nu[Z/\alpha],\eta} = \mathcal{M}2_2[\![I(ax)]\!]^{\Sigma_2}_{\nu,\eta}$$

Beweis: leichte Induktion über den Aufbau des Axioms ax.

Wie wissen also jetzt, daß auch das Universum $k2^{\mathcal{TM}2}$ nicht leer ist. Weiterhin ist dieses Universum maximal groß in dem Sinn, daß jede klassengültige Struktur der Klasse $k2$, die sich auf der Basis des Typmodells $\mathcal{TM}1$ bilden läßt, im Universum enthalten ist. Folgende Definitionen beschließen nun die Modellkonstruktion für den Fall der Erweiterung um eine neue Klasse:

Definition 2.62 *Modell für Th_2*

Das Typmodell für die Typsignatur Ω_2 wird festgelegt als:

$$\mathcal{TM}2 \quad = \quad (\mathcal{K}1 + k2^{\mathcal{TM}2}, \mathcal{TC}2_1)$$

Für alle Strukturen $X \in k2^{\mathcal{TM}2}$ und alle charakteristischen Konstanten $c{:}\tau \in KS_{k2}$ legen wir fest:

$$c^{\mathcal{M}2}(X) = const(X)(c)$$

Ein Modell für die Signatur Σ_2 ergibt sich dann durch folgende Definition:

$$\mathcal{M}2 = (\mathcal{TM}2, \mathcal{C}2)$$

wobei $\mathcal{C}2 = \mathcal{C}1 + \{c^{\mathcal{M}2} \mid \exists\tau.\, c{:}\tau \in KS_{k2}\}$

Wie schon das Σ_2-Modell $\mathcal{M}2_2$ aus Schritt 2 erfüllt $\mathcal{M}2$ wieder die Eigenschaften aus der Definition 2.36 für Signaturmodelle. Per Konstruktion des Universums $k2^{\mathcal{TM}2}$ sind in $\mathcal{M}2$ aber zusätzlich alle charakteristischen Axiome der Klasse $k2$ gültig. Weiterhin sind auch die Instanzaxiome für den Prototyp tc per Konstruktion gültig. Zusammenfassend gilt folgendes Theorem:

Theorem 2.36

Das in Definition 2.62 definierte Modell $\mathcal{M}2$ ist ein Modell für die Theorie Th_2 und es gilt:

$$\mathcal{M}2\,|_{\Sigma_1} = \mathcal{M}1$$

Die Theorie Th_2 ist also durch konservative Erweiterung der Theorie Th_1 entstanden.

Beweis: folgt direkt aus der Konstruktion des Modells $\mathcal{M}2$.

2.6.4 Erweiterung durch eine neue Arität

In diesem Abschnitt werde ich die konservative Theorieerweiterung durch eine neue Arität für einen Typkonstruktor beschreiben. Auch für diesen Erweiterungsmechanismus habe ich die Idee bereits in Abschnitt 2.2 beschrieben. Mit Hilfe dieses Mechanismus können nicht beliebige Aritäten hinzugefügt werden. Zum einen gibt es syntaktische Beschränkungen, die sich im wesentlichen durch die Regularitätsforderung für Typsignaturen ergeben. Zum anderen soll die Erweiterung konservativ sein, und somit muß eine relativ starke Beziehung zwischen der neuen Arität für den Typkonstruktor und schon bereits in der Theorie enthaltenen Aritäten bestehen.

Die Theorieerweiterung durch eine neue Arität wird durch folgende Definition charakterisiert.

Definition 2.63 *Erweiterung durch eine neue Arität*

Sei $Th_1 = (\Sigma_1, Ax_1, KAx_1)$ eine Theorie mit Signatur $\Sigma_1 = (\Omega_1, C_1, KS_1)$ und Typsignatur $\Omega_1 = (K_1, \leq_1, TC_1)$. Die Erweiterung der Theorie Th_1 durch eine neue Arität für den Typkonstruktor tc wird dann folgendermaßen notiert:

$$
\begin{array}{l}
Th_2 = Th_1 +_{arity} \\
\quad \begin{array}{lll}
\text{arities} & tc{:}(h_1, \ldots, h_m)h & \\
\text{witness} & tc{:}(k_1, \ldots, k_m)k & \\
\text{instmap} & I: \alpha \mapsto (\beta_1 \ldots \beta_m tc) & \\
 & \quad c_1 \mapsto \widetilde{c_1} & \\
 & \quad \vdots & \\
 & \quad c_n \mapsto \widetilde{c_n} &
\end{array}
\end{array}
$$

Damit die Theorieerweiterung durchgeführt werden kann, müssen einige syntaktische Bedingungen erfüllt sein.

B1: Anforderungen an die Klassen h, h_j, k, k_j:

$$
\begin{array}{l}
\{h_1, \ldots, h_m, h, k_1, \ldots, k_m, k\} \subseteq K_1 \\
h \neq top \wedge \{k' \mid h \rightsquigarrow k'\} = \{k\} \\
\forall j \in \{1, \ldots, m\}.\, h_j \leq k_j
\end{array}
$$

Die zweite Eigenschaft besagt, daß k der einzige Nachfolger von h ist und daß h nicht die Klasse *top* ist. Um einen Typkonstruktor mit Arität $tc{:}(h_1, \ldots, h_m)top$ einzuführen, muß der Mechanismus zur Einführung eines neuen Typkonstruktors aus Abschnitt 2.6.5 eingesetzt werden. Die dritte Eigenschaft sichert unter anderem die Comonotonie der neuen Signatur.

B2: Anforderungen bzgl. charakteristischer Konstanten:

$$
\begin{array}{l}
KS_h = \{c_1{:}\tau_1, \ldots, c_n{:}\tau_n\} \\
\forall i \in \{1, \ldots, n\}.\, TV(\tau_i) = \{\alpha\}
\end{array}
$$

In der Instanzabbildung I werden also alle charakteristischen Konstanten der Klasse h berücksichtigt. Weiterhin ist α die Klassenvariable der Klasse h.

B3: Anforderungen an den Zeugen:

$$tc{:}(k_1,\ldots,k_m)k \in TC_1$$
$$\forall w.\ w \leq [h_1,\ldots,h_n] \Longrightarrow k = mincodom(tc,w)$$

Diese Eigenschaften werden später die Regularität der neuen Signatur sicherstellen.

B4: Anforderungen an die Instanzabbildung I:

$$\forall j \in \{1,\ldots,m\}.\ \beta_j \in \Xi_{h_j}$$
$$\forall i \in \{1,\ldots,n\}.\ (\widetilde{c_i}, I(\tau_i)) \in C_1$$

wobei die Instanzabbildung I in offensichtlicher Weise auf Typterme fortgesetzt wird. Diese Eigenschaften garantieren, daß die spätere Definition für die Interpretation der neuen Überladung klassenkorrekt ist und daß die Interpretationen für die Instanzen der c_i typkorrekt sind.

B5: Die Instanzen der Klassenaxiome gelten für den Zeugen:

$$\forall ax \in KAx_h.(Th_1, D_{Min}) \vdash I(ax)$$

wobei die Instanzabbildung I in offensichtlicher Weise auf Terme fortgesetzt wird. Diese Bedingung ist zentral für die nachfolgende konservative Modellerweiterung.

Die Komponenten der neuen Theorie Th_2 werden dann definiert wie folgt:

$$\begin{aligned}
K_2 &= K_1\\
\leq_2 &= \leq_1\\
TC_2 &= TC_1 \cup \{tc{:}(h_1,\ldots,h_m)h\}
\end{aligned}$$

$$\begin{aligned}
\Omega_2 &= (K_2, \leq_2, TC_2)\\
C_2 &= C_1\\
KS_2 &= KS_1
\end{aligned}$$

$$\begin{aligned}
\Sigma_2 &= (\Omega_2, C_2, KS_2)\\
Ax_2 &= Ax_1 \cup \{c_1{:}I(\tau_1) = \widetilde{c_1}{:}I(\tau_1), \ldots, c_n{:}I(\tau_n) = \widetilde{c_n}{:}I(\tau_n)\}\\
KAx_2 &= KAx_1
\end{aligned}$$

$$Th_2 = (\Sigma_2, Ax_2, KAx_2)$$

Aus der obigen Definition für die Komponenten der Theorie Th_2 ist ersichtlich, daß in der Typsignatur Ω_2 lediglich die neue Überladung $tc{:}(h_1,\ldots,h_m)h$ hinzugefügt wird. Ansonsten bleibt die Signatur unverändert. Die Bedingungen B1 und B3 garantieren die Wohlgeformtheit der neuen Signaturen Ω_2 und Σ_2. Die Menge der Axiome AX_2 unterscheidet sich von Ax_1 nur durch die zusätzlichen Instanzaxiome für die charakteristischen Konstanten. Die Bedingungen B1 – B4 garantieren die Wohlgeformtheit der Theorie Th_2.

In der eben definierten Notation für die Erweiterung durch eine neue Arität sieht das Beispiel aus der informellen Einführung in Abschnitt 2.2, Abbildung 2.5 folgendermaßen aus:

$Fun2 = Fun1 +_{arity}$

arities	$\Rightarrow$	$:(top, po)po$		
witness	$\Rightarrow$	$:(top, top)top$		
instmap	$I:$	α_{po}	$\mapsto$	$(\beta_{top} \Rightarrow \beta_{po})$
		$\sqsubseteq$	$\mapsto$	$less\_fun$

Die Restriktion von Σ_2-Modellen auf die Signatur Σ_1 wird für die Erweiterung mittels $+_{arity}$ folgendermaßen definiert.

Definition 2.64 *Restriktion bei* $+_{arity}$

Sei die Theorie Th_2 durch eine Erweiterung um eine neue Arität mittels $+_{arity}$ entstanden, und sei $\mathcal{M}2$ ein Modell für die Signatur Σ_2. Dann ist die Restriktion $\mathcal{M}2\mid_{\Sigma_1}$ dasjenige Modell für Σ_1, das aus $\mathcal{M}2$ entsteht, wenn man im Typmodell $\mathcal{TM}2$ die Interpretation $tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}$ für die neue Überladung $tc{:}(h_1,\ldots,h_m)h$ entfernt.

Die Tatsache, daß $\mathcal{TM}2\mid_{\Omega_1}$ ein Modell für Ω_1 ist und daß $\mathcal{M}2\mid_{\Sigma_1}$ ein Modell für Σ_1 ist, ist wieder offensichtlich.

Aus der Definition für die Restriktion folgt durch leichte Induktion das folgende Restriktionslemma.

Theorem 2.37 *Restriktionslemma bei* $+_{arity}$

Sei die Theorie Th_2 durch eine Erweiterung um eine neue Arität mittels $+_{arity}$ entstanden, und sei $\mathcal{M}2 = (\mathcal{TM}2, \mathcal{C}2)$ ein Modell für die Signatur Σ_2.

Dann gilt für alle Typterme $\tau \in T_{\Omega_1}$ und Belegungen ν:

$$fgt_{c,k}(\mathcal{TM}2[\![\tau]\!]^{\Omega_2}_{\nu}) = \mathcal{TM}2\mid_{\Omega_1}[\![\tau]\!]^{\Omega_1}_{\nu}$$

wobei

$$c = lst_{\Omega_2}(\tau) \quad \text{und} \quad k = lst_{\Omega_1}(\tau)$$

Weiterhin gilt für alle Terme $t \in DT_{\Sigma_1}$ und alle Belegungen ν, η mit $\eta\ sat\,(\nu, GC(t))$:

$$\mathcal{M}2[\![t]\!]^{\Sigma_2}_{\nu,\eta} = \mathcal{M}2\mid_{\Sigma_1}[\![t]\!]^{\Sigma_1}_{\nu,\eta}$$

Beweis: leichte Induktion über den Aufbau der Terme τ bzw. t.

Nachdem die Restriktion definiert ist, können wir uns nun der konservativen Modellerweiterung im Fall $+_{arity}$ zuwenden. Sei also Th_2 aus Th_1 durch $+_{arity}$ entstanden und sei ein Modell $\mathcal{M}1$ für Th_1 gegeben mit $\mathcal{M}1 = (\mathcal{TM}1, \mathcal{C}1)$ und $\mathcal{TM}1 = (\mathcal{K}1, \mathcal{TC}1)$.

Um ein Modell für die neue Theorie Th_2 zu konstruieren, müssen wir im wesentlichen eine Interpretation $tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}$ für die neue Überladung $tc{:}(h_1,\ldots,h_m)h$ festlegen. Dabei werden

wir uns natürlich auf die bereits in $\mathcal{M}1$ vorliegenden Interpretationen $tc^{\mathcal{TM}1}_{(k_1,\ldots,k_m)k}$ und $\widetilde{c}_i^{\mathcal{M}1}$ abstützen. Für jede Belegung ν definieren wir den Wert $tc^{\mathcal{TM}1}_{(h_1,\ldots,h_m)h}(\nu(\beta_1),\ldots,\nu(\beta_m))$ derart, daß jedesmal eine Struktur der Klasse h entsteht. Die Bedingungen B1–B5 garantieren dabei, daß diese Strukturen typkorrekt und klassengültig sind. Weil die Universen für Modelle per Definition maximal groß sein müssen, können wir daraus dann ableiten, daß die Strukturen bereits Elemente des Universums $h^{\mathcal{TM}1}$ waren, was das zentrale Argument für den Nachweis der Konservativität ist.

Für beliebiges ν definieren wir die Interpretation $tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}$ für die neue Überladung $tc{:}(h_1,\ldots,h_m)h$ wie folgt:

$$tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}(\nu(\beta_1),\ldots,\nu(\beta_m)) = Z_\nu$$

wobei:

$$\begin{array}{lcll} Z_\nu & = & (car(\widehat{Z_\nu}),(const_\nu,\{k \mapsto stru(\widehat{Z_\nu})\})) & \\ \widehat{Z_\nu} & = & \mathcal{TM}1[\![(\beta_1 \ldots \beta_m tc)]\!]^{\Omega_1}_\nu & \\ const_\nu & = & c_1 & \mapsto \mathcal{M}1[\![I(c_1{:}\tau_1)]\!]^{\Sigma_1}_\nu \\ & & \vdots & \\ & & c_n & \mapsto \mathcal{M}1[\![I(c_n{:}\tau_n)]\!]^{\Sigma_1}_\nu \end{array}$$

Wegen Bedingungen B3 und B4 gilt mit obiger Definition dann

$$\widehat{Z_\nu} = tc^{\mathcal{TM}1}_{(k_1,\ldots,k_m)k}(fgt_{h_1,k_1}(\nu(\beta_1)),\ldots,fgt_{h_m,k_m}(\nu(\beta_m)))$$

und deswegen auch

$$\begin{array}{l} fgt_{h,k}(tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}(\nu(\beta_1),\ldots,\nu(\beta_m))) \\ = \\ tc^{\mathcal{TM}1}_{(k_1,\ldots,k_m)k}(fgt_{h_1,k_1}(\nu(\beta_1)),\ldots,fgt_{h_m,k_m}(\nu(\beta_m))) \end{array}$$

Dies ist gerade die Eigenschaft E5 für überladene Typkonstruktoren aus der Definition 2.30 für Typmodelle.

Für die gerade definierte Interpretation

$$tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}(\nu(\beta_1),\ldots,\nu(\beta_m)) = Z_\nu$$

beweisen wir jetzt eine Reihe von Theoremen, mit deren Hilfe wird dann die Konservativität der Modellerweiterung zeigen. Das Ziel, das wir dabei verfolgen, ist der Nachweis, daß Z_ν bereits schon im Typmodell $\mathcal{TM}1$ ein Element des Universums $h^{\mathcal{TM}1}$ ist. Per Konstruktion ist Z_ν eine Struktur der Klasse h. Um zu zeigen, daß Z_ν auch typkorrekt und klassengültig ist, werden wir die vorerst als Pseudo-Belegung zu betrachtende Belegung $\nu[Z_\nu/\alpha]$ verwenden.

Als erstes zeigen wir zwei technische Hilftheoreme, die in späteren Beweisen benötigt werden.

Theorem 2.38

Für alle ν und alle Typen $\tau \in T_{\Omega_1 \backslash h}$ von charakteristischen Konstanten $c{:}\tau \in KS_h$ mit $TV(\tau) = \{\alpha\}$ gilt:

$$fgt_{\widetilde{k_1},\widetilde{k_2}}(\mathcal{TM}1[\![\tau]\!]^{\Omega_1 \backslash h}_{\nu[Z_\nu/\alpha]}) = \mathcal{TM}1[\![I(\tau)]\!]^{\Omega_1 \backslash h}_{\nu}$$

wobei

$$\widetilde{k_1} = lst_{\Omega_1 \backslash h}(\tau) \wedge \widetilde{k_2} = lst_{\Omega_1 \backslash h}(I(\tau))$$

Beweis: Per Induktion über den Aufbau von τ und einer zum Beweis von Theorem 2.16 analogen Argumentation. Gemäß Theorem 2.13 sind alle hierbei verwendeten Interpretationen von Termen wohldefinert.

Theorem 2.39

Für alle $ax \in KAx_h$ mit $\Gamma = GC(ax)$, $TV(ax) = \{\alpha\}$ und alle Belegungen ν, η mit $\eta\ sat\ (\nu[Z_\nu/\alpha], \Gamma)$ gilt:

$$\mathcal{M}1[\![ax]\!]^{\Sigma_1 \backslash h}_{\nu[Z_\nu/\alpha],\eta} = \mathcal{M}1[\![I(ax)]\!]^{\Sigma_1 \backslash h}_{\nu,\eta}$$

Beweis: Per Induktion über den Aufbau des Terms t und Verwendung von Theorem 2.38.

Nun beweisen wir, daß für jede Belegung ν die soeben definierte Struktur Z_ν ein Element des Universums $h^{\mathcal{TM}1}$ ist. Dazu benötigen wir vier Schritte.

Theorem 2.40 *Schritt 1*

Z_ν ist eine Struktur der Klasse h.

Beweis: folgt trivial aus der Konstruktion von Z_ν.

Theorem 2.41 *Schritt 2*

Für alle Belegungen ν ist Z_ν eine typkorrekte Struktur bzgl. Typmodell $\mathcal{TM}1$ und Klasse h (i.Z. $Z_\nu\ tcor\ (\mathcal{TM}1, KS_h)$).

Beweis: Um diese Aussage zu beweisen, müssen wir zeigen, daß die Bedingungen für typkorrekte Strukturen gemäß Definition 2.35 von Z_ν erfüllt werden. Neben Bedingungen, die bereits trivialerweise per Konstruktion erfüllt sind, bleibt zu zeigen:

Für alle charakteristischen Konstanten $c{:}\tau \in KS_h$ und Belegungen ν soll gelten:

$$const(Z_\nu)(c) \in car(\mathcal{TM}1[\![\tau]\!]^{\Omega_1 \backslash h}_{\nu[Z_\nu/\alpha]})$$

wobei α wieder die Klassenvariable der Klasse h ist.

Aus der Definition für Z_ν wissen wir:

$$const(Z_\nu)(c) = \mathcal{M}1[\![I(c{:}\tau)]\!]^{\Sigma_1}_{\nu}$$

und mit Theorem 2.18 über die Typkorrektheit der Interpretationsfunktion folgt dann

$$const(Z_\nu)(c) \in car(\mathcal{T}\mathcal{M}1[\![I(\tau)]\!]_\nu^{\Omega_1})$$

Es reicht also zu zeigen:

$$car(\mathcal{T}\mathcal{M}1[\![\tau]\!]_{\nu[Z_\nu/\alpha]}^{\Omega_1\setminus h}) = car(\mathcal{T}\mathcal{M}1[\![I(\tau)]\!]_\nu^{\Omega_1})$$

Es gilt folgende Gleichungskette, die den Beweis abschließt:

$$\begin{aligned} & car(\mathcal{T}\mathcal{M}1[\![\tau]\!]_{\nu[Z_\nu/\alpha]}^{\Omega_1\setminus h}) \\ & \qquad = \quad \text{(Theorem 2.38 und Carrier-Lemma 2.10)} \\ & car(\mathcal{T}\mathcal{M}1[\![I(\tau)]\!]_\nu^{\Omega_1\setminus h}) \\ & \qquad = \quad \text{(Theorem 2.17 und Carrier-Lemma 2.10)} \\ & car(\mathcal{T}\mathcal{M}1[\![I(\tau)]\!]_\nu^{\Omega_1}) \end{aligned}$$

Theorem 2.42 *Schritt 3*

Für alle Belegungen ν ist Z_ν eine klassengültige Struktur bzgl. Σ_1-Modell $\mathcal{M}1$ (i.Z. Z_ν $cval\,(\mathcal{M}1, KAx_h)$).

Beweis: Um diese Aussage zu beweisen, müssen wir zeigen, daß die Bedingungen für klassengültige Strukturen gemäß Definition 2.42 von Z_ν erfüllt werden. Wenn α die Klassenvariable der Klasse h ist, müssen wir zeigen:

Für alle $ax \in KAx_h$ mit generischem Kontext $\Gamma = GC(ax)$ und für alle Belegungen ν, η mit η $sat\,(\nu[Z_\nu/\alpha], \Gamma)$ muß gelten:

$$\mathcal{M}1[\![ax]\!]_{\nu[Z_\nu/\alpha],\eta}^{\Sigma_1\setminus h} = \mathbb{T}$$

Wegen Bedingung B5 aus Definition 2.63 und der Korrektheit des Herleitungsbegriffs reicht es aber zu zeigen:

$$\mathcal{M}1[\![ax]\!]_{\nu[Z_\nu/\alpha],\eta}^{\Sigma_1\setminus h} = \mathcal{M}1[\![I(ax)]\!]_{\nu,\eta}^{\Sigma_1}$$

Wir bilden wieder eine Gleichungskette:

$$\begin{aligned} & \mathcal{M}1[\![ax]\!]_{\nu[Z_\nu/\alpha],\eta}^{\Sigma_1\setminus h} \\ & \qquad = \quad \text{(Theorem 2.39)} \\ & \mathcal{M}1[\![I(ax)]\!]_{\nu,\eta}^{\Sigma_1\setminus h} \\ & \qquad = \quad \text{(Theorem 2.21)} \\ & \mathcal{M}1[\![I(ax)]\!]_{\nu,\eta}^{\Sigma_1} \end{aligned}$$

Theorem 2.43 *Schritt 4*

Für alle Belegungen ν ist Z_ν ein Element im Universum $h^{\mathcal{T}\mathcal{M}1}$.

Beweis: folgt unmittelbar aus der Maximalität der Universen im Modell $\mathcal{M}1$ und Theorem 2.42 aus Schritt 3.

Wir wissen jetzt, daß für alle Belegungen ν die Struktur

$$tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}(\nu(\beta_1),\ldots,\nu(\beta_m)) = Z_\nu$$

ein Element des Universums $h^{\mathcal{TM}1}$ ist. Aus diesem Grund reicht es, das Modell $\mathcal{M}1$ um die Interpretation $tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}$ für die neue Überladung $tc{:}(h_1,\ldots,h_m)h$ zu erweitern. Die Interpretationen $c^{\mathcal{TM}1}(Z_\nu)$ für die charakteristischen Konstanten $c{:}\tau \in KS_h$ sind wegen $Z_\nu \in h^{\mathcal{TM}1}$ schon bereits in $\mathcal{M}1$ definiert.

Folgende Definitionen beschließen nun die Modellkonstruktion für den Fall der Erweiterung um eine neue Arität:

Definition 2.65 *Modell für Th_2*

Das Typmodell für die Typsignatur Ω_2 wird festgelegt als:

$$\mathcal{TM}2 \quad = \quad (\mathcal{K}1, \mathcal{TC}1 + tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h})$$

Ein Modell für die Signatur Σ_2 ergibt sich durch folgende Definition:

$$\mathcal{M}2 = (\mathcal{TM}2, \mathcal{C}1)$$

Theorem 2.44

Das in Definition 2.65 definierte Modell $\mathcal{M}2$ ist ein Modell für die Theorie Th_2 und es gilt:

$$\mathcal{M}2\,|_{\Sigma_1} = \mathcal{M}1$$

Die Theorie Th_2 ist also durch konservative Erweiterung der Theorie Th_1 entstanden.

Beweis: Aus Definition 2.65 und den für die neue Interpretation $tc^{\mathcal{TM}2}_{(h_1,\ldots,h_m)h}$ gezeigten Theoremen folgt, daß $\mathcal{M}2$ ein Modell für die Signatur Σ_2 ist. Die Gültigkeit der Instanzaxiome folgt für alle ν leicht aus der Konstruktion von Z_ν und Anwendung des Restriktionslemmas 2.37 für $+_{arity}$. Die Gültigkeit aller übrigen Axiome folgt direkt aus dem Restriktionslemma.

2.6.5 Erweiterung durch Typdefinition

In diesem Abschnitt werde ich die konservative Theorieerweiterung durch die Einführung eines neuen Typkonstruktors beschreiben. Diese Form der Erweiterung ist, wie auch die Erweiterung um eine neue Konstante, bereits für die Logik HOL bekannt und wird ausführlich in [GM93, Kapitel 16] (*extension by type definition*) beschrieben. Die hier vorgestellte Variante wurde lediglich an die Logik mit Typklassen angepaßt. Mit diesem Erweiterungsmechanismus wird ein neuer Typkonstruktor tc mit Arität $(k_1,\ldots,k_m)top$ eingeführt.

Die Theorieerweiterung durch einen neuen Typkonstruktor wird durch folgende Definition charakterisiert.

Definition 2.66 *Erweiterung durch einen neuen Typkonstruktor*

Sei $Th_1 = (\Sigma_1, Ax_1, KAx_1)$ eine Theorie mit Signatur $\Sigma_1 = (\Omega_1, C_1, KS_1)$ und Typsignatur $\Omega_1 = (K_1, \leq_1, TC_1)$. Die Erweiterung der Theorie Th_1 durch einen neuen Typkonstruktor tc wird dann folgendermaßen notiert:

$Th_2 = Th_1 +_{type}$

type	$tc : (k_1, \ldots, k_m)top$
rep_type	τ
rep_pred	p
rep_fun	$rep : (\alpha_1 \ldots \alpha_m tc) \Rightarrow \tau$
abs_fun	$abs : \tau \Rightarrow (\alpha_1 \ldots \alpha_m tc)$

Damit die Theorieerweiterung durchgeführt werden kann, müssen wieder einige syntaktische Bedingungen erfüllt sein.

B1: Anforderungen an die Klassen k_j:

$$\{k_1, \ldots, k_m\} \subseteq K_1$$

Die Klassen in der Arität des neuen Konstruktors tc müssen bereits in K_1 bekannt sein.

B2: Anforderungen an den neuen Konstruktor:

$$\neg \exists wk.\ tc : (w)k \in TC_1$$

Der Typkonstruktor tc ist neu.

B3: Anforderungen an den repräsentierenden Typ τ und die Typvariablen $\alpha_1, \ldots, \alpha_m$:

$$\tau \in T_{\Omega_1}$$
$$TV(\tau) = \{\alpha_1, \ldots, \alpha_m\}$$
$$\forall j \in \{1, \ldots, m\}.\ \alpha_j \in \Xi_{k_j}$$

Im Typterm τ über der Signatur Ω_1 kommen also genau die Typvariablen $\alpha_1, \ldots, \alpha_m$ vor, und die α_j passen jeweils zu den Klassen k_j.

B4: Anforderungen an das Repräsentationsprädikat p:

$$p \in DT_{\Sigma_1} \wedge GT_{\Sigma_1}(p) = (\tau \Rightarrow bool) \wedge GC_{\Sigma_1}(p) = \emptyset$$

p ist also ein wohlgeformtes und geschlossenes Prädikat über dem repräsentierenden Typ τ.

B5: Anforderungen an die Abstraktions– und Repräsentationsfunktionen abs, rep:

$$\neg \exists \rho.\ (rep, \rho) \in C_1 \vee (abs, \rho) \in C_1$$

Die Konstanten abs, rep sind neu.

B6: Das Repräsentationsprädikat p beschreibt eine nichtleere Teilmenge des Repräsentationstyps τ:

$$(Th_1, D_{Min}) \vdash \exists x : \tau.\ px$$

Die Komponenten der neuen Theorie Th_2 werden dann definiert wie folgt:

$$
\begin{array}{rcl}
K_2 & = & K_1 \\
\leq_2 & = & \leq_1 \\
TC_2 & = & TC_1 \cup \{tc{:}(k_1, \ldots, k_m)top\}
\end{array}
$$

$$
\begin{array}{rcl}
\Omega_2 & = & (K_2, \leq_2, TC_2) \\
C_2 & = & C_1 \cup \{rep{:}(\alpha_1 \ldots \alpha_m tc) \Rightarrow \tau, abs{:}\tau \Rightarrow (\alpha_1 \ldots \alpha_m tc)\} \\
KS_2 & = & KS_1
\end{array}
$$

$$
\begin{array}{rcl}
\Sigma_2 & = & (\Omega_2, C_2, KS_2) \\
Ax_2 & = & Ax_1 \cup \ \{ \\
 & & \quad p(rep\ x{:}(\alpha_1 \ldots \alpha_m tc)), \\
 & & \quad abs(rep\ x) = x{:}(\alpha_1 \ldots \alpha_m tc), \\
 & & \quad p\ y{:}\tau \rightarrow rep(abs\ y) = y \\
 & & \quad \} \\
KAx_2 & = & KAx_1
\end{array}
$$

$$
Th_2 = (\Sigma_2, Ax_2, KAx_2)
$$

Aus der obigen Definition für die Komponenten der Theorie Th_2 ist ersichtlich, daß zur Typsignatur Ω_1 lediglich der neue Typkonstruktor *tc* hinzugefügt wird, und die Signatur Σ_1 um die neuen Konstanten *rep* und *abs* erweitert wird. Ansonsten bleibt die Signatur unverändert. Die Bedingungen B1 bis B5 garantieren die Wohlgeformtheit der neuen Signaturen Ω_2 und Σ_2, sowie der Theorie Th_2.

In der eben definierten Notation für die Erweiterung durch einen neuen Typkonstruktor sieht die Einführung des Typkonstruktors $\rightarrow$ für Operationen (vgl. Abschnitt 1.4 und Abschnitt 4.7.1) wie folgt aus:

$Cfun1 = Cont{+}_{type}$

type	$\rightarrow {:}(pcpo, pcpo)top$
rep_type	$\alpha_{pcpo} \Rightarrow \beta_{pcpo}$
rep_pred	$\lambda f{:}\alpha_{pcpo} \Rightarrow \beta_{pcpo}.\ contX\ f$
rep_fun	$fapp{:}(\alpha_{pcpo} \rightarrow \beta_{pcpo}) \Rightarrow (\alpha_{pcpo} \Rightarrow \beta_{pcpo})$
abs_fun	$fabs{:}(\alpha_{pcpo} \Rightarrow \beta_{pcpo}) \Rightarrow (\alpha_{pcpo} \rightarrow \beta_{pcpo})$

Die Restriktion von Σ_2-Modellen auf die Signatur Σ_1 wird für die Erweiterung mittels $+_{type}$ folgendermaßen definiert.

Definition 2.67 *Restriktion bei* $+_{type}$

Sei die Theorie Th_2 durch eine Erweiterung um einen neuen Typkonstruktor *tc* mittels $+_{type}$ entstanden, und sei $\mathcal{M}2$ ein Modell für die Signatur Σ_2. Dann ist die Restriktion $\mathcal{M}2\,|_{\Sigma_1}$ dasjenige Modell für Σ_1, das aus $\mathcal{M}2$ entsteht, wenn man im Typmodell $\mathcal{T}\mathcal{M}2$

die Interpretation $tc^{\mathcal{TM}2}_{(k_1,\ldots,k_m)top}$ für den neuen Konstruktor $tc{:}(k_1,\ldots,k_m)top$ entfernt und im Modell $\mathcal{M}2$ die Interpretationen für die Konstanten *rep* und *abs* vergißt.

Die Tatsache, daß $\mathcal{TM}2\mid_{\Omega_1}$ ein Modell für Ω_1 ist und daß $\mathcal{M}2\mid_{\Sigma_1}$ ein Modell für Σ_1 ist, ist wieder offensichtlich.

Wie auch im Fall der Erweiterung durch eine Konstante ist der Beweis für das Restriktionslemma bei $+_{type}$ trivial.

Theorem 2.45 *Restriktionslemma bei* $+_{type}$

Sei die Theorie Th_2 durch eine Erweiterung um einen neuen Typkonstruktor tc mittels $+_{type}$ entstanden, und sei $\mathcal{M}2 = (\mathcal{TM}2, \mathcal{C}2)$ ein Modell für die Signatur Σ_2.

Dann gilt für alle Typterme $\tau \in T_{\Omega_1}$ und Belegungen ν:

$$fgt_{c,k}(\mathcal{TM}2[\![\tau]\!]^{\Omega_2}_{\nu}) = \mathcal{TM}2\mid_{\Omega_1}[\![\tau]\!]^{\Omega_1}_{\nu}$$

wobei

$$c = lst_{\Omega_2}(\tau) \quad \text{und} \quad k = lst_{\Omega_1}(\tau)$$

Weiterhin gilt für alle Terme $t \in DT_{\Sigma_1}$ und alle Belegungen ν, η mit $\eta\ sat\,(\nu, GC(t))$:

$$\mathcal{M}2[\![t]\!]^{\Sigma_2}_{\nu,\eta} = \mathcal{M}2\mid_{\Sigma_1}[\![t]\!]^{\Sigma_1}_{\nu,\eta}$$

Beweis: leichte Induktion über den Aufbau der Terme τ bzw. t und Verwendung der Tatsache, daß der Typkonstruktor tc und die Konstanten *rep* und *abs* in der alten Signatur Σ_1 überhaupt nicht vorkommen.

Nachdem die Restriktion definiert ist, können wir uns nun der konservativen Modellerweiterung im Fall $+_{type}$ zuwenden. Sei also Th_2 aus Th_1 durch $+_{type}$ entstanden, und sei ein Modell $\mathcal{M}1$ für Th_1 gegeben mit $\mathcal{M}1 = (\mathcal{TM}1, \mathcal{C}1)$ und $\mathcal{TM}1 = (\mathcal{K}1, \mathcal{TC}1)$.

Um ein Modell für die neue Theorie Th_2 zu konstruieren, müssen wir das Typmodell $\mathcal{TM}1$ um eine Interpretation $tc^{\mathcal{TM}2}_{(k_1,\ldots,k_m)top}$ für den neuen Konstruktor $tc{:}(k_1,\ldots,k_m)top$ erweitern, und wir müssen das Signaturmodell $\mathcal{M}1$ um eine Interpretation für die beiden Konstanten *rep* und *abs* erweitern. Wenden wir uns zunächst der Erweiterung des Typmodells zu.

Für beliebiges ν definieren wir die Anwendung der Interpretation $tc^{\mathcal{TM}2}_{(k_1,\ldots,k_m)top}$ für den neuen Typkonstruktor $tc{:}(k_1,\ldots,k_m)top$ wie folgt:

$$tc^{\mathcal{TM}2}_{(k_1,\ldots,k_m)top}(\nu(\alpha_1),\ldots,\nu(\alpha_m)) = Z_\nu$$

wobei:

$$\begin{array}{lcl} Z_\nu & = & (\widehat{Z_\nu}, (\emptyset, \emptyset)) \\ \widehat{Z_\nu} & = & \{y \mid y \in car(\mathcal{TM}1[\![\tau]\!]^{\Omega_1}_{\nu}) \wedge (\mathcal{M}1[\![p]\!]^{\Omega_1}_{\nu})(y) = \mathbb{T}\} \end{array}$$

Es gilt folgendes Theorem:

Theorem 2.46

Für alle Belegungen ν ist das soeben definierte Z_ν eine Struktur der Klasse *top*.

Beweis: Es reicht zu zeigen, daß $\widehat{Z_\nu}$ ein Element im Präuniversum PU ist, da dann per Definition $Z_\nu = (\widehat{Z_\nu}, (\emptyset, \emptyset))$ ein Element im Universum *top* ist. Aus der Definition von $\widehat{Z_\nu}$ folgt, daß $\widehat{Z_\nu}$ eine Teilmenge der Trägermenge $car(\mathcal{TM}1[\![\tau]\!]_\nu^{\Omega_1})$ ist, welche ihrerseits sicher ein Element in PU ist. Ferner wissen wir aus Bedingung E2 der Definition 2.22, daß PU bzgl. nichtleerer Teilmengen abgeschlossen ist. Der Beweis reduziert sich also auf den Nachweis, daß es $y \in car(\mathcal{TM}1[\![\tau]\!]_\nu^{\Omega_1})$ gibt, so daß gilt[24]:

$$(\mathcal{M}1[\![p]\!]_\nu^{\Omega_1})(y) = \mathbb{T}$$

Dies folgt aber offensichtlich aus $(Th_1, D_{Min}) \vdash \exists x : \tau.\ px$ in Bedingung B6 der Definition 2.66 und der Korrektheit des Beweissystems, denn es gilt:

$$(\text{es gibt } y \text{ mit: } (\mathcal{M}1[\![p]\!]_\nu^{\Omega_1})(y) = \mathbb{T}) \Longleftrightarrow \mathcal{M}1[\![\exists x : \tau.\ px]\!]_\nu^{\Omega_1} = \mathbb{T}$$

Im obigen Beweis haben wir gesehen, daß $\widehat{Z_\nu}$ eine nichtleere Teilmenge der Trägermenge $car(\mathcal{TM}1[\![\tau]\!]_\nu^{\Omega_1})$ ist und somit ebenfalls eine Menge im Präuniversum PU ist. Aus diesem Grund ist klar, wie wir die Interpretationen für die Konstanten *rep* und *abs* definieren müssen, damit die neuen Axiome in Ax_2 gültig sind.

Sei für alle Belegungen ν die Struktur $\widehat{Z_\nu}$ definiert wie vorher, und sei $Y = car(\mathcal{TM}1[\![\tau]\!]_\nu^{\Omega_1})$. Damit gilt offensichtlich $\widehat{Z_\nu} \subseteq Y$. Sei weiterhin

$$\{\beta_1, \ldots, \beta_m\} = \{\alpha_1, \ldots, \alpha_m\} \text{ mit } \beta_1 \prec \ldots \prec \beta_m$$

d.h. die Typvariablen β_j sind die α_j aufgereiht gemäß der kanonischen Ordnung $\prec$ auf Typvariablen.

Wir definieren (die Instanzen von) $rep^{\mathcal{M}2}$ als Inklusion:

$$rep^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m)) : \widehat{Z_\nu} \to Y \text{ mit } x \in \widehat{Z_\nu} \mapsto x$$

und (die Instanzen von) $abs^{\mathcal{M}2}$ als Projektion:

$$abs^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m)) : Y \to \widehat{Z_\nu} \text{ mit } y \in Y \mapsto \begin{cases} y & \text{falls } y \in \widehat{Z_\nu} \\ ch(\widehat{Z_\nu}) & \text{sonst} \end{cases}$$

wobei *ch* die Auswahlfunktion aus Definition 2.22 ist.

Mit obigen Definitionen gilt offensichtlich:

$$\begin{array}{l} rep^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m))(x) \in \widehat{Z_\nu} \wedge \\ abs^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m))(rep^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m))(x)) = x \wedge \\ y \in \widehat{Z_\nu} \Longrightarrow rep^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m))(abs^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m))(y)) = y \end{array}$$

Folgende Definitionen beschließen nun die Modellkonstruktion für den Fall der Erweiterung um einen neuen Typkonstruktor:

[24] da das Restriktionsprädikat p geschlossen ist, ist die Variablenbelegung η uninteressant.

Definition 2.68 *Modell für Th_2*

Das Typmodell für die Typsignatur Ω_2 wird festgelegt als:

$$\mathcal{TM}2 \quad = \quad (\mathcal{K}1, \mathcal{TC}1 + tc^{\mathcal{TM}2}_{(k_1,\ldots,k_m)top})$$

Ein Modell für die Signatur Σ_2 ergibt sich durch folgende Definition:

$$\mathcal{M}2 = (\mathcal{TM}2, \mathcal{C}1 + \{rep^{\mathcal{M}2}, abs^{\mathcal{M}2}\})$$

Theorem 2.47

Das in Definition 2.68 definierte Modell $\mathcal{M}2$ ist ein Modell für die Theorie Th_2 und es gilt:

$$\mathcal{M}2\,|_{\Sigma_1} = \mathcal{M}1$$

Die Theorie Th_2 ist also durch konservative Erweiterung der Theorie Th_1 entstanden.

Beweis: Folgt offensichtlich aus den obigen Definitionen für die Interpretationen $tc^{\mathcal{TM}2}_{(k_1,\ldots,k_m)}top$ und $rep^{\mathcal{M}2}$ bzw. $abs^{\mathcal{M}2}$.

Es sind jetzt alle konservativen Erweiterungsmechanismen für Theorien dargestellt, die zur Entwicklung der Logik HOLCF (siehe Kapitel 4) benutzt wurden. Hiermit endet dann auch die rein technische Präsentation von HOLC, der Logik höherer Stufe mit Typklassen.

In den Kapiteln 3 und 4 folgt nun die eher praktische Anwendung der Logik HOLC im Isabelle-System.

Kapitel 3

Formalisierung von HOLC in Isabelle

In diesem Kapitel werde ich die Implementierung der Logik HOLC im generischen Theorembeweiser Isabelle [Pau94] beschreiben. Wie bereits im Abschnitt 2.1 erwähnt, stammt diese Variante der Logik HOL von Tobias Nipkow. Da Isabelle das Konzept der Typklassen bereitstellt, ist dieses automatisch in allen Objektlogiken verfügbar, die in der Metalogik von Isabelle formalisiert werden. Somit handelt es sich bei der Isabelle-Variante von HOL um eine Logik höherer Stufe mit Typklassen.

Im vorangegangenen Kapitel 2 habe ich die Logik HOLC unabhängig von der Isabelle-Implementierung eingeführt und die Methode der konservativen Erweiterung für den Umgang mit Klassen erweitert. Dieses Vorgehen war insbesondere notwendig, um eine formale Semantik für das Klassenkonzept im Zusammenhang mit der Logik HOLC zu schaffen und die spezifische Verwendung von Typklassen bei der Entwicklung von HOLCF im Kapitel 4 zu rechtfertigen. Weiterhin konnte ich dadurch die Logik HOLC an sich untersuchen und ihre Kodierung als Objektlogik im Isabelle-System ausklammern, was sonst die Präsentation der Semantik von HOLC nur undurchsichtig gemacht hätte.

Im folgenden sei mit HOLC immer die Logik gemeint, die ich in Kapitel 2 eingeführt habe. Mit HOL meine ich die Logik und das System von Gordon [GM93]. Wenn die Logik HOLC von ihrer Implementierung im Isabelle-System unterschieden werden soll, dann werde ich die Implementierung als Isabelle-HOL bzw. als Isabelle-Variante von HOL bezeichnen.

Im ersten Abschnitt dieses Kapitels werde ich das Isabelle-System kurz vorstellen. Eine ausführliche Behandlung von Isabelle, welche eine Einführung, ein Referenzmanual und eine Beschreibung einiger Objektlogiken beinhaltet, findet sich in [Pau94]. Der zweite Abschnitt befaßt sich mit Tobias Nipkows Formalisierung von HOLC in Isabelle, welche ebenfalls in [Pau94] beschrieben ist.

3.1 Der generische Theorembeweiser Isabelle

Isabelle ist ein generischer Theorembeweiser, das heißt er kann für verschiedene Objektlogiken instantiiert werden. Systeme, die die Formalisierung von Objektlogiken ermöglichen, werden

bisweilen auch Logical-Frameworks genannt. In Isabelle erfolgt die Formalisierung einer Objektlogik dadurch, daß die Axiome und Schlußregeln der Objektlogik in einer speziellen Logik, der sogenannten Metalogik von Isabelle, ausgedrückt werden. Die Metalogik von Isabelle ist eine spezielle Form der intuitionistischen Logik höherer Stufe. Eine ausführliche Beschreibung der technischen Hintergründe findet sich in [Pau89].

Mit Hilfe der Metalogik wird die Bildung von Beweisen in der Objektlogik kontrolliert und die Ableitung neuer Inferenzregeln auf eine formale Basis gestellt. Neben dieser Funktion als reiner Beweisprüfer können in Isabelle aber auch Beweise programmiert werden. Dabei werden primitive Beweisschritte, sogenannte Taktiken (tactics), durch spezielle Ablaufstrukturen, den sogenannten Tacticals, verknüpft, was eine komplexe Taktik ergibt, die bei Anwendung auf ein Beweisziel dieses unter Umständen bereits vollständig löst. Aus diesem Grund wird Isabelle als taktischer Theorembeweiser bezeichnet, der, je nach Beweisziel und Konstruktion einer geeigneten Taktik, eine schrittweise, eine halbautomatische bzw. im Grenzfall sogar eine vollautomatische Lösung des Beweiszieles erlaubt. In der aktuellen Version stehen ein leistungsfähiger klassischer Beweiser und ein umfangreiches Paket zur Termsimplifikation [Nip89] zur Verfügung, die Isabelle zu einem ansprechenden Automatisierungsgrad verhelfen.

Das Isabelle-System selbst ist in der funktionalen Programmiersprache ML [HMM86] programmiert, und die Interaktion mit dem Benutzer erfolgt über die normale ML-Oberfläche. Isabelle wurde erstmals 1986 in Umlauf gebracht. Die Version von 1987 benutzte dann bereits die oben angesprochene Metalogik zur Formalisierung von Objektlogiken. In der Version von 1988 wurde ein polymorphes Typsystem eingebaut, und es wurde eine Unterstützung für natürliches Schließen integriert. In der Version von 1991 wurde dann von Tobias Nipkow das Konzept der Polymorphie mit ordnungssortierten Typklassen eingebaut [Nip91], was eine wesentlich filigranere Formulierung von Objektlogiken erlaubte. Dieses Klassenkonzept entspricht genau demjenigen, das ich in Kapitel 2 beschrieben habe. In der aktuellen Version von 1994 wurde dieses Konzept nochmals erweitert, indem statt mit ordnungssortierten Klassen mit sogenannten Durchschnittsklassen gearbeitet wird [NP93]. Für die hier beschriebene Entwicklung von HOLCF sind die damit einhergehenden technischen Unterschiede jedoch unerheblich[1], und somit habe ich mich in Kapitel 2 auf das Konzept der ordnungssortierten Typklassen beschränkt. Darüberhinaus kann für die Polymorphie mit ordnungssortierten Klassen eine erheblich einfachere Semantik angegeben werden[2].

In den folgenden drei Teilabschnitten werde ich kurz beschreiben, wie Objektlogiken in Isabelle formalisiert werden. Dadurch wird auch gleich die Syntax von Isabelle[3] eingeführt, die ich in Kapitel 4 benutzen werde. Die folgende Darstellung ist nur eine Zusammenfassung aus dem Einführungskapitel in [Pau94] und ist nicht dazu gedacht, dieses zu ersetzen.

3.1.1 Formalisierung logischer Syntax in Isabelle

Die Metalogik von Isabelle ist eine intuitionistische Logik höherer Stufe mit Typklassenpolymorphie, die bis auf ihre intuitionistische Beschränkung sehr verwandt mit der Logik HOLC

[1] die Entwicklung von HOLCF nützt die zusätzliche Ausdrucksstärke der Durchschnittsklassen nicht aus.

[2] bei Verwendung von Durchschnittsklassen ist der Begriff des Typmodells erheblich komplizierter, da die Familie der Universen bzgl. Durchschnitten abgeschlossen sein muß. Insbesondere muß man auch leere Klassen erlauben, was die Formulierung des Kalküls umständlicher macht.

[3] Ich verwende eine graphisch geschönte Variante der Isabelle-Syntax.

ist. Die Terme und Formeln der Logik sind also auch hier getypte λ-Terme. Die Termsprache läßt sich wie folgt zusammenfassen:

`x`	Variable
`c`	Konstante
`t::`τ	Typeinschränkung für Term oder gebundene Variable
λ`x.t`	Abstraktion
λ`x1 ... xn.t`	Mehrfach gecurriete Abstraktion λ`x1`...λ`xn.t`
`t(u)`	Applikation
`t(u1,...,un)`	Mehrfach gecurriete Applikation `t(u1)...(un)`

Im Gegensatz zu HOLC dürfen Typbeschränkungen für beliebige Terme geschrieben werden, und die Applikation wird als `t(u)` geschrieben und nicht als `(t u)`. Die Syntax für Typen entspricht ebenfalls im wesentlichen der von HOLC. Bei Typvariablen darf das Subskript für die Klasse weggelassen werden. In diesem Fall wird automatisch die Defaultklasse angehängt, die jeweils eingestellt werden kann.

$\alpha_{\texttt{c}}$	Typvariable der Klasse `c`
α	Typvariable der Defaultklasse
τ `tc`	Postfix Anwendung des Typkonstruktors `tc`
τ`1` $\Rightarrow$ τ`2`	Infix-Schreibweise für Funktionsraumkonstruktor $\Rightarrow$ der Metalogik
[τ`1,...,`τ`n]` $\Rightarrow \tau$	Kurzschreibweise für τ`1` $\Rightarrow$ `(...`τ`n` $\Rightarrow \tau$`)`

Die Typklassen werden in Isabelle primär dazu eingesetzt, um syntaktische Kategorien einzuführen und diese sauber zu unterscheiden, beziehungsweise abzustufen. In [Nip91] findet sich eine ausführliche Beschreibung und Motivation für diesen syntaktischen Gebrauch von Typklassen. In der aktuellen Version von Isabelle wird zwar mit Durchschnittsklassen gearbeitet, doch will ich in dieser kurzen Einführung noch die ordnungssortierte Version von 1991 verwenden. Die größte syntaktische Kategorie sei mit `any` bezeichnet, die alle Typen umfaßt. Eine Teilklasse davon ist die syntaktische Kategorie `logic` (also `logic < any`), die für die Typisierung der Konstanten der Metalogik verwendet wird. Auf diese werde ich später noch zurückkommen. Um die Syntax der Objektlogiken von der Metalogik zu unterscheiden werden für die jeweilige Objektlogik weitere syntaktische Teilkategorien eingeführt. Im Fall von HOLC ist dies die Kategorie `term` der Terme der Objektlogik HOLC. Diese Klasse entspricht der Klasse *top* aus dem Kapitel 2. Es gilt also `term < logic`. Da in der Logik HOLC kein Unterschied zwischen Formeln und Termen gemacht wird, reicht diese eine syntaktische Kategorie aus.

Die von mir in Kapitel 2 eingeführte Syntax von HOLC läßt sich vollständig in Isabelle abbilden. Speziell die Klassenhierarchie von HOLC findet sich als Teilhierarchie der Isabelle-Klassen wieder, indem die Isabelle-Klasse `term` als die HOLC-Klasse *top* interpretiert wird. In Bezug auf die Semantik von HOLC möchte ich an dieser Stelle betonen, daß in Isabelle lediglich die Syntax der Objektlogik abgebildet wird, und der Ableitungsbegriff der Objektlogik mit Hilfe der Metalogik von Isabelle formalisiert wird. Die Semantik von Isabelle ist aber jeweils unabhängig von den Semantiken der Objektlogiken. Entscheidend für die Formalisierung der Objektlogik HOLC ist lediglich, daß die wohlgeformten Typen und Terme, sowie der Ableitungsbegriff von HOLC adäquat in Isabelle formalisiert werden. Adäquat bedeutet hierbei, daß es für jeden wohlgeformten Term der Logik HOLC einen entsprechenden wohlgeformten Term in der Isabelle-Formalisierung gibt (u.u.), und daß jeder Ableitung im Kalkül

von HOLC eine Ableitung im Isabelle-System entspricht (u.u.). Eine ausführliche Diskussion dieser Problematik findet sich in [Pau89][4].

Wenden wir uns nun der weiteren Formalisierung der Syntax von HOLC in Isabelle zu. Die Typkonstruktoren der minimalen Typsignatur von HOLC (Definition 2.2) werden in Isabelle durch folgende Aritätsvereinbarungen formalisiert:

`bool::term`	Wahrheitswerte *bool* der Objektlogik HOLC
`ind ::term`	Typ der Individuen *ind*
$\Rightarrow$ `::(term,term)term`	Konstruktor für den Funktionsraum

Im Fall des Konstruktors $\Rightarrow$ sieht man deutlich, wie der Konstruktor $\Rightarrow$ der Metalogik auch für die Objektlogik zugänglich gemacht wird. Dies hat zur Folge, daß die λ-Abstraktion der Metalogik mit all ihren Eigenschaften ebenfalls als λ-Abstraktion der Objektlogik verwendet werden kann. Diese Vorgehensweise bietet sich für alle Objektlogiken mit λ-Abstraktion an, da dann die Eigenschaften der λ-Abstraktion aus der Metalogik vererbt werden und nicht eigens neu formalisiert werden müssen. Dies setzt natürlich voraus, daß die beiden Abstraktionen wirklich die gleichen Eigenschaften haben sollen, was im Fall der Objektlogik HOLC der Fall ist.

Als Beispiel für die Isabelle-Formalisierung von Konstanten seien hier die Konstanten der Minimalsignatur aus Definition 2.10 angegeben:

$\rightarrow$ `::[bool,bool]` $\Rightarrow$ `bool`	Implikation
`=` `::[`$\alpha_{\texttt{term}}$`,`α`]` $\Rightarrow$ `bool`	Identität
ε `::(`α $\Rightarrow$ `bool)` $\Rightarrow$ $\alpha_{\texttt{term}}$	Hilbert Auswahloperator

Man beachte, daß bei den Konstanten `=` und ε die Polymorphie auf die Klasse `term` beschränkt wird, was eine saubere Trennung von der Metalogik gewährleistet. Auf die eben skizzierte Art kann die komplette Syntax der Logik HOLC in Isabelle formalisiert werden. Wenden wir uns nun der Formalisierung des Kalküls zu.

3.1.2 Formalisierung von Inferenzregeln in Isabelle

Bisher haben wir nur die Termsprache der Metalogik von Isabelle um neue Typen und Konstanten erweitert. Um Inferenzregeln einer Objektlogik zu formalisieren, müssen wir zur Metalogik von Isabelle neue Axiome hinzunehmen. Wie bereits erwähnt, handelt es sich bei der Metalogik von Isabelle um intuitionistische Logik höherer Stufe. Diese wird ausführlich in [Pau89] beschrieben. Zu Anfang sind in der Metalogik nur die beiden Typklassen `any` und `logic` mit `logic < any` bekannt. Neben Typvariablen gibt es als einzige Typkonstruktoren die Typkonstante `prop` für den Typ der Wahrheitswerte der Metalogik und den Konstruktor $\Rightarrow$ für den Funktionenraum. Für diese beiden Konstruktoren sind anfangs folgende Aritäten vereinbart:

```
prop  :: logic
  ⇒   :: (logic,logic)logic
```

Die logischen Konstanten der Metalogik sind die für intuitionistische Logik höherer Stufe üblichen:

[4] Paulson betrachtet in [Pau89] die beiden Richtungen der Biimplikation einzeln und bezeichnet sie mit *adequate* und *faithful*.

$\Longrightarrow$	:: [prop,prop] $\Rightarrow$ prop	Implikation der Metalogik
$\bigwedge$	:: $(\alpha_{\texttt{logic}} \Rightarrow \texttt{prop}) \Rightarrow \texttt{prop}$	Allquantor der Metalogik
$\equiv$	:: $[\alpha_{\texttt{any}}, \alpha] \Rightarrow \texttt{prop}$	Identität der Metalogik

Der entscheidende Trick, um die Inferenzregeln und Axiome der Objektlogik in der Metalogik auszudrücken, besteht darin, die beiden Typen prop::logic und bool::term durch eine Injektionsfunktion Trueprop zu verbinden.

```
Trueprop :: bool ⇒ prop
```

Aufgrund des Typs von Trueprop handelt es sich hierbei um ein Prädikat der Metalogik auf dem Typ bool::term der Formeln in der Objektlogik. Intuitiv bedeutet Trueprop(p), daß die Formel p in der Objektlogik herleitbar sein soll. Die Implikation $\Longrightarrow$ der Metalogik wird benutzt, um Regeln auszudrücken, und der Allquantor $\bigwedge$ dient zur Formalisierung von Eigenvariablenbedingungen (Variable kommt nicht frei vor in Formelmenge).

Im Abschnitt 2.5 habe ich den Kalkül des natürlichen Schließens von HOLC mit expliziter Verwaltung der lebenden Annahmen notiert. Dies hat dort die Formulierung der Nebenbedingungen und der Korrektheit des Kalküls vereinfacht. Betrachten wir als Beispiel dazu die beiden Regeln [disch] und [mp]:

$$[\text{disch}]\ \frac{H \blacktriangleright q}{H \setminus \{p\} \blacktriangleright p \rightarrow q} \qquad [\text{mp}]\ \frac{H_1 \blacktriangleright p \rightarrow q \quad H_2 \blacktriangleright p}{H_1 \cup H_2 \blacktriangleright q}$$

Um die Formalisierung von HOLC in Isabelle zu verstehen, ist es aber einfacher, die Regeln in der ursprünglichen Notation für das natürliche Schließen anzugeben.

$$[\text{disch}]\ \begin{array}{c}[p]\\ \vdots\\ q\end{array}\over{p \rightarrow q} \qquad [\text{mp}]\ \frac{p \rightarrow q \quad p}{q}$$

Die beiden Regeln werden in Isabelle formalisiert wie folgt:

$\bigwedge$ p q.(Trueprop(p)$\Longrightarrow$Trueprop(q))$\Longrightarrow$Trueprop(p $\rightarrow$ q)	Regel [disch] in Isabelle
$\bigwedge$ p q.Trueprop(p $\rightarrow$ q)$\Longrightarrow$Trueprop(p)$\Longrightarrow$Trueprop(q)	Regel [mp] in Isabelle

Am Beispiel der Regel [disch] wird die Behandlung von lebenden Annahmen deutlich. Die Verwaltung der lebenden Annahmen in der Objektlogik wird über die Verwendung der Metaimplikation auf die Annahmenverwaltung der Metalogik abgeschoben. Damit die Regeln besser lesbar werden, werden das Prädikat Trueprop sowie pränexe Quantoren unterdrückt. Ebenso werden geschachtelte Implikationen der Form

$$\Phi_1 \Longrightarrow (\ldots \Phi_n \Longrightarrow \Psi \ldots)$$

verkürzt als

$$[\![\ \Phi_1; \ldots; \Phi_n\ ]\!] \Longrightarrow \Psi$$

geschrieben. Somit ergibt sich für die beiden Regeln [disch] und [mp] die Schreibweise:

`(p⟹q)⟹p → q`	Regel [disch] in Kurznotation
`⟦ p → q ; p ⟧⟹q`	Regel [mp] in Kurznotation

Um Eigenvariablenbedingungen auszudrücken, wird der Allquantor $\bigwedge$ der Metalogik verwendet. Ähnlich wie bei der Annahmenverwaltung wird hierdurch die Beachtung von Eigenvariablenbedingungen in der Objektlogik durch die entsprechende Behandlung in der Metalogik erzwungen. Die Regel [abstract] der Logik HOLC wird in der originalen Notation für das natürliche Schließen folgendermaßen formuliert.

$$[\text{abstract}]\ \frac{t_1 = t_2}{(\lambda x.t_1) = (\lambda x.t_2)} \left\{ x \text{ nicht frei in den lebenden Annahmen} \right.$$

In Isabelle wird daraus:

```
⋀t1 t2. (⋀x::α_term. Trueprop(t1(x)::β_term = t2(x)))
                ⟹ Trueprop(λx.t1(x) = λx.t2(x))
```

bzw. in Kurzschreibweise

```
(⋀x::α_term. t1(x)::β_term = t2(x))⟹λx.t1(x) = λx.t2(x)
```

Um auszudrücken, daß in den Termen `t1` und `t2` evtl. freie Variablen vorkommen, werden in Isabelle Variablen vom Funktionstyp verwendet und diese im Anwendungskontext `t1(x)` bzw. `t2(x)` geschrieben. Die expliziten Typisierungen `x::`$\alpha_{\texttt{term}}$ und `t1(x)::`$\beta_{\texttt{term}}$ sind notwendig, um den Grad der Polymorphie zu beschränken. Eine andere Regel, zu deren Formalisierung Funktionsvariablen und Applikation benutzt werden, ist die Substitutionsregel von HOLC. Sie lautet in der Originalnotation des natürlichen Schließens:

$$[\text{subst}]\ \frac{t_1 = t_2 \quad p[t_1/x]}{p[t_2/x]}$$

In Isabelle wird daraus (gleich in Kurzschreibweise):

```
⟦ t1 = t2 ; p(t1)⟧⟹p(t2)
```

Die β-Konversion der Metalogik und die Verwendung von Funktionsvariablen ersetzen hierbei den Substitutionsmechanismus der Objektlogik HOLC. Die Regeln [hyp] und [type_inst] der Objektlogik HOLC müssen in Isabelle nicht eigens formalisiert werden, denn sie werden bereits durch spezielle Regeln der Metalogik (Annahmentaktik und Resolutionstaktik) abgedeckt. Mehr dazu jedoch im nächsten Abschnitt.

3.1.3 Ableitung von Formeln und Regeln in Isabelle

Die vorhergehenden Abschnitte haben gezeigt, wie man die Syntax und die Inferenzregeln der Objektlogik in Isabelles Metalogik formalisiert. In diesem Abschnitt werde ich kurz beschreiben, wie diese Formalisierung benutzt wird, um Herleitungen im Objektkalkül in der Metalogik zu simulieren, und wie neue Inferenzregeln der Objektlogik abgeleitet werden können. Dazu ist es notwendig, ein paar Bemerkungen über den Kalkül der Metalogik zu machen.

Es wurde bereits mehrfach erwähnt, daß Isabelles Metalogik eine intuitionistische Logik höherer Stufe ist. Die Inferenzregeln dieser Logik sind im Isabelle-System als Funktionen programmiert, die Theoreme in Theoreme abbilden. Auf der Ebene der Metalogik findet sich also der Gedanke der LCF-Systeme [GMW79, Pau87] wieder. Im Gegensatz zu LCF werden diese primitiven Inferenzregeln in der Beweisarbeit mit der jeweiligen Objektlogik aber selten gebraucht, denn hier wird mit einem speziellen Satz von abgeleiteten Inferenzregeln gearbeitet. Zu diesen abgeleiteten Regeln gehört vor allem die Resolutionsregel, mit deren Hilfe nahezu alle Beweisschritte der Objektlogik abgedeckt werden.

Zunächst jedoch möchte ich den Satz der primitiven Inferenzregeln der Metalogik präsentieren. Die nachfolgende Aufstellung ist dem Kapitel 5, Abschnitt 2 (Primitive meta-level inference rules) im Referenz-Teil von [Pau94] entnommen.

Die Metalogik ist ebenfalls ein Kalkül des natürlichen Schließens (Annahmenkalkül), der formal mit Behauptungen (assertions[5]) arbeitet. Eine Behauptung ϕ ist eine Formel, die von einer Menge von (lebenden) Annahmen $\{\phi_1, \ldots, \phi_n\}$ abhängt. In der Literatur über Isabelle [Pau89, Pau94] wird eine Behauptung folgendermaßen notiert:

$$\phi \quad [\phi_1, \ldots, \phi_n]$$

Die informelle Lesart ist dabei:

> Die Formel ϕ kann unter Annahme der Formeln ϕ_i hergeleitet werden.

bzw. die Semantik davon:

> Unter der Annahme, daß die Formeln ϕ_i gültig sind, ist die Formel ϕ gültig.

Der Kalkül des natürlichen Schließens ist gerade so konzipiert, daß bei vollständiger *Entlastung (discharge)* der Formel ϕ von allen Annahmen ϕ_i durch die Regel $(\Longrightarrow I)$ die entstehende Formel

$$\phi_1 \Longrightarrow \ldots \Longrightarrow \phi_n \Longrightarrow \phi$$

eine Tautologie ist. In diesem Zusammenhang möchte ich speziell auf die Dissertation von Gerhard Gentzen [Gen35, Sza69] hinweisen, in der eine hervorragende Diskussion der Motivation für Kalküle des natürlichen Schließens enthalten ist. In dieser Arbeit findet sich auch eine kompakte Darstellung des Zusammenhangs zwischen Sequenzenkalkülen (L-Systeme), Annahmenkalkülen (N-Systeme) und den Hilbert-Frege Systemen.

Die obigen Behauptungen $\phi \quad [\phi_1, \ldots, \phi_n]$ entsprechen natürlich exakt den in Abschnitt 2.5 eingeführten Objekten $\phi_1, \ldots, \phi_n \blacktriangleright \phi$, die in der HOL-Literatur [GM93] (und daher auch bei mir) unglücklicherweise als Sequenzen bezeichnet werden.

Die Regeln von Annahmenkalkülen werden üblicherweise so notiert, daß die lebenden Annahmen, bzw. deren Veränderung bei Anwendung einer Schlußregel, nur implizit erwähnt werden.

[5] diese Terminologie wird von Paulson [Pau94] verwendet.

Diese Kurznotation verhilft den Kalkülen des natürlichen Schließens im wesentlichen zu ihrer Eleganz. Die explizite Verwaltung der Annahmenmenge, wie sie in Abschnitt 2.5 und in [GM93] verwendet wurde, hat ihre Begründung dagegen zum einen in der Tatsache, daß dann die Eigenvariablenbedingungen klarer fomuliert werden können. Der eigentliche Grund für die explizite Notation liegt aber darin, daß nur so eine direkte Formulierung der Korrektheit der Regeln möglich ist. Wird aber der Annahmenkalkül auf einen anderen Kalkül zurückgeführt, in [Gen35] ist dies ein Hilbert-Frege Kalkül, so entfällt diese zweite Begründung. Die Zurückführung auf einen anderen Kalkül lohnt sich allerdings nur, wenn die Korrektheit dieses Kalküls bereits bewiesen wurde. Im Fall von HOLC hätte sich hier eine angepaßte Version[6] von Andrews Logik höherer Stufe [And86] angeboten, aber die Anpassung erschien mir (und wohl auch den Verfassern der HOL-Semantik) unangenehmer zu sein, als die direkte Verwaltung der Annahmen und der direkte Korrektheitsbeweis.

Die primitiven Inferenzregeln des Annahmenkalküls der Metalogik sind durch folgende Menge von Regeln gegeben[7]:

Logische Implikation[8]:

$$\frac{\begin{array}{c}[\phi]\\ \vdots\\ \psi\end{array}}{\phi \Longrightarrow \psi}(\Longrightarrow I) \qquad \frac{\phi \Longrightarrow \psi \quad \phi}{\psi}(\Longrightarrow E)$$

Logische Äquivalenz:

$$\frac{\begin{array}{cc}[\phi] & [\psi]\\ \vdots & \vdots\\ \psi & \phi\end{array}}{\phi \equiv \psi}(\equiv I) \qquad \frac{\phi \equiv \psi \quad \phi}{\psi}(\equiv E)$$

$$a \equiv a\ (\text{refl}) \qquad \frac{a \equiv b}{b \equiv a}(\text{sym}) \qquad \frac{a \equiv b \quad b \equiv c}{a \equiv c}(\text{trans})$$

λ-Konversionen[9]:

$$(\lambda x.\, a) \equiv (\lambda y.\, a[y/x]) \qquad ((\lambda x.\, a)(b)) \equiv a[b/x] \qquad \frac{f(x) \equiv g(x)}{f \equiv g}(\text{ext})$$

Abstraktion und Kombination[10]:

$$\frac{a \equiv b}{(\lambda x.\, a) \equiv (\lambda x.\, b)}(\text{abs}) \qquad \frac{f \equiv g \quad a \equiv b}{f(a) \equiv g(b)}(\text{comb})$$

[6] in Andrews Logik höherer Stufe gibt es keine Polymorphie, geschweige denn Typklassen.

[7] die Regel (inst) wurde von mir hinzugefügt, da sie meiner Meinung nach in [Pau94] fehlt. In [Pau89] hingegen wird noch eine Metalogik ohne Polymorphie betrachtet.

[8] in $(\Longrightarrow I)$ werden alle lebenden Vorkommen der Annahme ϕ, von denen ψ abhängt, entlastet.

[9] α-Konversion gilt, falls y nicht frei in a, (ext) gilt, falls x nicht frei in den lebenden Annahmen, f oder g.

[10] (abs) gilt, falls x nicht frei in den lebenden Annahmen.

Universelle Quantifizierung[11]:

$$\frac{\phi}{\bigwedge x.\ \phi}(\bigwedge I) \qquad \frac{\bigwedge x.\ \phi}{\phi[b/x]}(\bigwedge E)$$

Typinstanz (Polymorphie)[12]:

$$\frac{\phi}{\sigma(\phi)}(\text{inst})$$

In [Pau94] werden diverse Regeln der Metalogik vorgestellt, die sich aus den obigen primitiven Regeln ableiten lassen. Die beiden wichtigsten abgeleiteten Regeln, mit denen der Großteil der Beweisarbeit mit der Objektlogik abgedeckt wird, sind die Annahmentaktik[13] (ass) und die Resolutionstaktik (res). Zuerst stelle ich die beiden Taktiken kurz dar, ohne auf ihre Korrektheit näher einzugehen. Ihr Sinn wird jedoch erst in der nachfolgenden Präsentation der Beweistechnik in Isabelle deutlich.

Annahmentaktik:

$$\frac{[\![\ \phi_1;\ldots;\phi_n\ ]\!]\Longrightarrow\phi}{([\![\ \phi_1;\ldots;\phi_{i-1};\phi_{i+1};\ldots;\phi_n\ ]\!]\Longrightarrow\phi)s}(\text{ass})$$

Dabei wird angenommen, daß ϕ_i die Form

$$\bigwedge x_1\ldots x_l.\ [\![\ \theta_1;\ldots;\theta_k\ ]\!]\Longrightarrow\theta$$

hat und daß es einen Unifikator[14] s und einen Index j mit $1 \leq j \leq k$ gibt, so daß:

$$(\lambda x_1\ldots x_l.\ \theta_j)s = (\lambda x_1\ldots x_l.\ \theta)s$$

Mit Hilfe dieser Regel werden Beweise per Annahme in der Objektlogik simuliert. Die Korrektheit der Regel ist intuitiv klar, denn das Entfernen einer Tautologie[15] aus der Prämisse einer Implikation ändert nichts an der logischen Aussage.

Resolutionstaktik:

$$\frac{[\![\ \psi_1;\ldots;\psi_m\ ]\!]\Longrightarrow\psi \quad [\![\ \phi_1;\ldots;\phi_n\ ]\!]\Longrightarrow\phi}{([\![\ \phi_1;\ldots;\phi_{i-1};\psi_1;\ldots;\psi_m;\phi_{i+1};\ldots;\phi_n\ ]\!]\Longrightarrow\phi)s}(\text{res})$$

Dabei wird angenommen, daß es einen Unifikator[16] s gibt, so daß:

$$(\psi)s \equiv (\phi_i)s$$

[11] $(\bigwedge I)$ gilt, falls x nicht frei in den lebenden Annahmen.

[12] (inst) gilt, falls für alle lebenden Annahmen ψ, von denen ϕ abhängt, gilt: $\alpha \in FV(\psi) \Longrightarrow \sigma(\alpha) = \alpha$. Dabei ist σ eine ordnungssortierte Typsubstitution im Sinne der Definition 2.7.

[13] nicht zu verwechseln mit der Einführung von Annahmen in der Metalogik.

[14] hier ist Unifikation höherer Stufe für Typ- und Termvariablen gemeint.

[15] falls es den Unifikator s gibt, dann handelt es sich bei $(\phi_i)s$ um eine Tautologie.

[16] hier ist ebenfalls Unifikation höherer Stufe für Typ- und Termvariablen gemeint.

Um die Resolutionstaktik anwenden zu können, muß die linke Prämisse der Regel (res) evtl. vorher noch durch sogenannte *Lifting-Regeln* vorbereitet werden. Dies ist immer dann notwendig, wenn ϕ_i nicht von der atomaren Form `Trueprop(p)` ist, was zum Beispiel bei den Regeln [disch] und [abstract] der Objektlogik HOLC der Fall ist. Durch die Lifting-Regeln werden alle Bestandteile der linken Prämisse von (res) durch tautologische Umformungen so verändert, daß die dadurch entstehende neue Konklusion ψ' dann mit ϕ_i unifiziert werden kann. Die Darstellung dieser Regeln würde den Rahmen dieser Kurzfassung sprengen, und ich möchte diesbezüglich auf [Pau89] und [Pau94] verweisen.

Ich habe diese beiden wesentlichen Regeln auch darum hier vorgestellt, da ihr Vorhandensein in der Metalogik der Grund dafür ist, daß die Einführung von Annahmen und die Instantiierungsregel für polymorphe Formeln in Objektlogiken, in Abschnitt 2.5 waren dies die Regeln [hyp] und [type_inst], nicht eigens in Isabelle formalisiert werden müssen. Sie werden eben gerade durch die abgeleiteten Regeln (ass) und (res) der Metalogik abgedeckt.

Ich werde nun kurz den Mechanismus erklären, wie in Isabelle neue Inferenzregeln für die Objektlogik abgeleitet werden. Relativ zur Metalogik handelt es sich hierbei einfach um die Ableitung eines Theorems. Die Herleitung eines Theorems der Objektlogik ist dabei nur ein Spezialfall der Herleitung einer Regel, die keine Prämissen bzgl. der Metaimplikation $\Longrightarrow$ hat.

Angenommen, es soll folgende Regel der Objektlogik abgeleitet werden:

$$\frac{\theta_1 \ldots \theta_k}{\phi}$$

Bezogen auf die Metalogik gilt es also folgendes Theorem herzuleiten[17]:

$$[\![\ \theta_1; \ldots; \theta_k\ ]\!] \Longrightarrow \phi$$

Aufgrund der Regel $(\Longrightarrow I)$ ist dies sehr einfach möglich, wenn unter Annahme der Formeln $\theta_1, \ldots, \theta_k$ die Formel ϕ hergeleitet werden kann. Hierzu leitet man zuerst trivial per Annahme und $(\Longrightarrow I)$ die Formel $\phi \Longrightarrow \phi$ her. Damit gilt trivialerweise auch die Behauptung

$$\phi \Longrightarrow \phi \qquad [\theta_1, \ldots, \theta_k]$$

Diese Behauptung (Formel $\phi \Longrightarrow \phi$ abhängig von lebenden Annahmen $\theta_1, \ldots, \theta_k$) interpretiert man jetzt als initialen Beweiszustand. Durch schrittweise Weiterentwicklung des Beweiszustandes, bei der vor allem die Resolutionstaktik (res) eingesetzt wird, indem man den aktuellen Beweiszustand mit bereits abgeleiteten Theoremen[18] der Metalogik resolviert,

$$\frac{\text{Theorem} \qquad \text{aktueller Beweiszustand}}{\text{neuer Beweiszustand}}$$

versucht man einen Beweiszustand zu erreichen, der die Form

$$\phi \qquad [\theta_1, \ldots, \theta_k]$$

[17] im Beispiel sind keine Eigenvariablenbedingungen vorhanden. Eine diesbezügliche Verallgemeinerung des Beispiels ist aber trivial.

[18] dies können Theoreme der puren Metalogik sein, aber vor allem auch bereits andere abgeleitete Regeln der Objektlogik.

hat. Eine abschließende Entlastung der Hypothesen $\theta_1, \ldots, \theta_k$ liefert dann die erwünschte abgeleitete Regel.

Auch wenn es aufgrund dieser äußerst kurzen Darstellung nicht ganz offensichtlich geworden sein sollte, so ist doch durch die Verwendung des Metaquantors $\bigwedge$ zur Formulierung von Eigenvariablenbedingungen stets die korrekte Mitführung derselbigen im Metakalkül gewährleistet. Dies ist besonders wichtig für die Ableitung neuer Regeln in der Objektlogik, wie sie eben gerade skizziert wurde. Um dies klarer zu machen, müßten speziell auch die oben kurz angesprochenen Liftingregeln und der Prozeß der Unifikation höherer Stufe eingehender vorgestellt werden. Dies würde aber den Rahmen dieser Darstellung bei weitem sprengen.

An dieser Stelle möchte ich die Vorstellung von Isabelle beenden und hoffe, daß sie ausreichend war, um die nachfolgenden Ausführungen über die Formalisierung der Logik HOLC in Isabelle und daran anschließend die Entwicklung von HOLCF verständlich zu machen.

3.2 Ausschnitte aus der Isabelle-Formalisierung von HOLC

In diesem Abschnitt präsentiere ich Ausschnitte aus der Formalisierung von HOLC im Isabelle-System. Ich werde dabei nur soviel von den einzelnen Theorien zeigen, wie zum Verständnis der Entwicklung von HOLCF in Kapitel 4 notwendig ist. Über Logik höherer Stufe und speziell über HOLC habe ich in Kapitel 2 bereits genügend geschrieben, und so werde ich die Isabelle-Formalisierung, die wie bereits mehrfach erwähnt von Tobias Nipkow stammt, nur in bezug auf die Eigenheiten von Isabelle kommentieren.

Die Formalisierung von HOLC in Isabelle beginnt mit der Theorie `HOL`, die direkt auf der Metalogik von Isabelle aufsetzt. Darauf bauen mehrere andere Theorien auf, die ich zusammen mit der Theorie `HOL` nun im einzelnen kurz vorstellen werde. Die Abbildung 3.1 zeigt die einzelnen Theorien und ihre Abhängigkeit, soweit sie für HOLCF relevant sind.

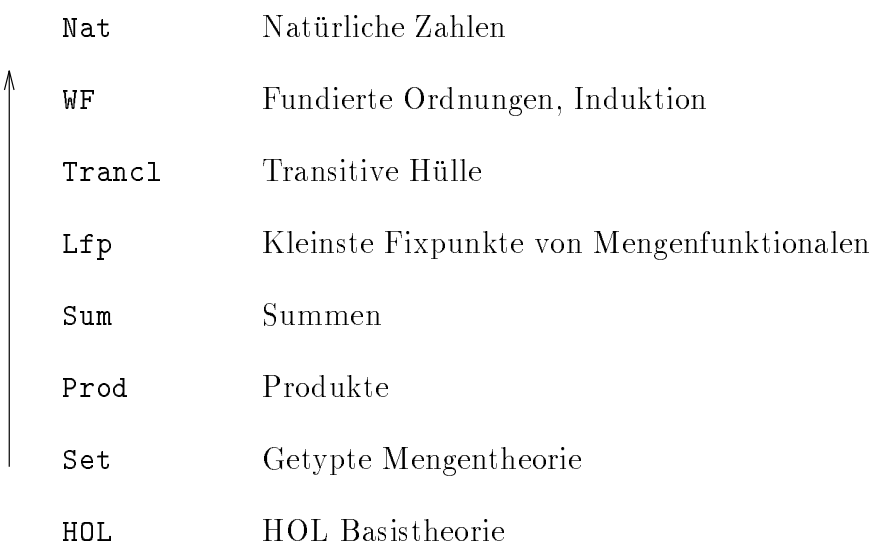


Abbildung 3.1: Hierarchie der HOL-Theorien in Isabelle

Neben den in Abbildung 3.1 gezeigten Theorien gibt es noch diverse andere HOL-Theorien im Isabelle-System, wie zum Beispiel Arithmetik für natürliche Zahlen, Listen und Lazy-Listen. Diesbezüglich möchte ich auf [Pau94] und die weiterführende Literatur verweisen.

Die Formalisierung der Syntax, der Axiome und der Regeln von HOLC erfolgt in Isabelle nach dem Muster, wie es gerade im letzten Abschnitt beschrieben wurde. Die Isabelle-Syntax für Theorien sollte zusammen mit den nachfolgenden Kommentaren genügend erklärt sein. Für genauere Details möchte ich wieder auf [Pau94] verweisen.

In Kapitel 2 wurden die Signatur und die Axiome der Logik HOLC schrittweise in den einzelnen Theorien *Min*, *Log* und *Init* eingeführt. Separat dazu wurde das Deduktionssystem D_{Min} von HOLC eingeführt. In der Isabelle-Formalisierung geschieht dies in einem Aufwasch. Abgesehen vom Typ der Individuen *ind* und dem Axiom der Unendlichkeit, die in der Isabelle-Theorie `Nat` nachgeliefert werden, wird die komplette Logik HOLC in der Isabelle-Theorie `HOL` formalisiert. Es ist offensichtlich, daß auch in Isabelle eine schrittweise Einführung der Konzepte entsprechend Kapitel 2 möglich wäre.

3.2.1 Theorie `HOL`

Der wesentliche Teil der Basistheorie `HOL`, in dem ich einige Details weggelassen habe, sieht folgendermaßen aus:

```
HOL = Pure +

classes
  term < logic

default
  term

types
  bool

arities
  fun :: (term, term) term
  bool :: term

consts

  (* Constants *)
  Trueprop      ::  bool ⇒ prop                    ( (_)  5)
  not           ::  bool ⇒ bool                    ( ¬_  [40] 40)
  True, False   ::  bool
  if            ::  [bool, α, α] ⇒ α
  Inv           ::  (α ⇒ β) ⇒ (β ⇒ α)

  (* Binders *)
```

```
  Eps           ::  (α ⇒ bool) ⇒ α                (binder  ε  10)
  All           ::  (α ⇒ bool) ⇒ bool             (binder  ∀  10)
  Ex            ::  (α ⇒ bool) ⇒ bool             (binder  ∃  10)

  (* Infixes *)
   o            ::  [β ⇒ γ, α ⇒ β, α] ⇒ γ          (infixr 50)
   =            ::  [α, α] ⇒ bool                 (infixl 50)
   ¬=           ::  [α, α] ⇒ bool                 ( (_ ¬=/ _)  [50, 51] 50)
   ∧            ::  [bool, bool] ⇒ bool           (infixr 35)
   ∨            ::  [bool, bool] ⇒ bool           (infixr 30)
   →            ::  [bool, bool] ⇒ bool           (infixr 25)

translations
   x ¬= y  ≡  ¬(x = y)

rules
  eq_reflection  (x=y)⟹(x ≡ y)

  (* Basic Rules *)
  refl           t = t::α
  subst          ⟦ s = t; P(s)⟧⟹P(t::α)
  impI           (P⟹Q)⟹P → Q
  mp             ⟦ P → Q;  P⟧⟹Q

  ext            (⋀x::α. f(x)::β = g(x))⟹(λx.f(x)) = (λx.g(x))
  selectI        P(x::α)⟹P(εx.P(x))

  (* Definitions *)
  True_def       True ≡ ((λx::bool.x)=(λx.x))
  All_def        All(P) ≡ (P = (λx.True))
  Ex_def         Ex(P) ≡ P(εx.P(x))
  False_def      False ≡ (∀P.P)
  not_def        ¬P ≡ P → False
  and_def        P ∧ Q ≡ ∀R. (P → Q → R) → R
  or_def         P ∨ Q ≡ ∀R. (P → R) → (Q → R) → R

  (* Axioms *)
  True_or_False  (P=True) ∨ (P=False)
  iff            (P → Q) → (Q → P) → (P=Q)

  (* Misc Definitions *)
  Inv_def        Inv(f::α ⇒ β) ≡ (λy. εx. f(x)=y)
  o_def          (f::β ⇒ γ) o g ≡ (λ(x::α). f(g(x)))
  if_def         if(P,x,y) ≡ εz::α. (P=True → z=x) ∧ (P=False → z=y)
end
```

Abbildung 3.2: Theorie HOL

In der Sektion `classes` wird die neue Klasse `term` eingeführt, die der Klasse *top* in HOLC entspricht. In der Sektion `default` wird eingestellt, daß jede Typvariable ohne expliziten Klassenindex automatisch von der Klasse `term` sein soll. Dies macht die Terme besser lesbar. In der Sektion `types` wird die neue Typkonstante `bool` eingeführt. In der Sektion `arities` wird zum einen vereinbart, daß die Typkonstante `bool` in der Klasse `term` liegen soll, zum anderen wird für den Typkonstruktor `fun` der Metalogik die neue Arität `(term,term)term` vereinbart. Dadurch wird der Typkonstruktor `fun`, der in Typtermen stets mit der bekannten Infix-Schreibweise $\Rightarrow$ notiert wird, auch für die Objektlogik `HOL` zugänglich.

In der Sektion `consts` werden die Konstanten nebst evtl. Notationsvereinfachungen wie Infix-, Mixfix- oder Binder-Syntax eingeführt. Als erstes wird die Konstante `Trueprop` für die Einbindung in die Metalogik eingeführt. Die spezielle Notationsvereinfachung `( (_) 5)` führt dazu, daß die Konstante `Trueprop` in konkreten Termen unsichtbar bleibt und an den richtigen Stellen eingefügt wird. Die einzelnen Konstanten sind bereits weitestgehend aus Kapitel 2 bekannt. Die Prädikate `inj` und `surj` für Injektivität und Surjektivität werden erst in der Theorie `Set` eingeführt. Dagegen werden hier in `HOL` die zusätzlichen Abkürzungen `If`, `Inv`, $\neg$`=` und `o` für Fallunterscheidung, Inversenbildung, Ungleichheit und Funktionskomposition vereinbart.

Die nächste Sektion `translations` dient im allgemeinen dazu, um die automatische Hin– und Rückübersetzung von Notationsvereinfachungen auszudrücken. Im Beispiel wird vereinbart, daß Terme der Form `x` $\neg$`= y` intern behandelt werden wie der Term $\neg$`(x = y)`.

In der Sektion `rules` werden gleichzeitig Axiome und Inferenzregeln der Objektlogik HOLC vereinbart. Jedes Axiom bzw. jede Regel hat in Isabelle einen Namen, über die das jeweilige Axiom bzw. die Regel im Theorembeweiser ansprechbar ist. Der Name steht immer in der linken Spalte.

Als Axiome der Objektlogik können dabei aber eigentlich nur noch solche Formeln bezeichnet werden, die keine freien Variablen enthalten und in denen die Symbole der Metalogik $\bigwedge$, $\Longrightarrow$ und $\equiv$ nicht vorkommen. Enthält ein Axiom der Objektlogik eine freie Variable, so wird diese von Isabelle automatisch in eine Unifikationsvariable bzw. in eine Schemavariable im Sinn von Abschnitt 2.5 verwandelt. Aus dem Axiom wird somit eine Inferenzregel ohne Prämissen.

Die Objektlogik HOLC ist im Prinzip so ausdrucksstark, daß eigentlich fast nur Axiome verwendet werden müßten. In [And86] wird eine Version der Logik höherer Stufe beschrieben, die mit einer einzigen Regel (Regel R) auskommt. In Kapitel 2 werden (nach Gordon [GM93]) ebenfalls relativ wenig Regeln verwendet. In der vorliegenden Isabelle-Theorie `HOL` hingegen werden möglichst viele Axiome als Regeln dargestellt, weil damit eine effizientere Handhabung des Isabelle-Systems möglich ist. So werden zum Beispiel Konstantendefinitionen zumeist über die Metagleichheit $\equiv$ abgewickelt, und statt Axiomen der Form `p` $\wedge$ `q` $\rightarrow$ `r` verwendet man lieber die Regel $[\![$ `p ; q` $]\!] \Longrightarrow$`r`. Bei Konstantendefinitionen für Funktionssymbole definiert man aus Bequemlichkeit die Funktion im vollen Anwendungskontext `f(x)` $\equiv$ `E` mit freier Variable (und damit Schemavariable) `x`, anstatt die eigentliche Definitionsschreibweise `f =` λ`x.E` einzusetzen. Dies ist aber aufgrund der Extensionalität von Funktionen in `HOL` und der entsprechenden Einbettung in die Metalogik äquivalent. Diese Bevorzugung der Metalogik führt zum Beispiel dazu, daß in der vorliegenden Theorie `HOL` kein einziges echtes Axiom der Objektlogik HOLC mehr vorkommt. Dies ist aber nicht weiter tragisch, da es sich durchwegs um äquivalente Formulierungen handelt, deren Zusammenhang mit der Schreibweise

in der Objektlogik offensichtlich ist. Im folgenden werde ich aber auch bei solchen Pseudo-Axiomen bzw. Pseudo-Definitionen trotzdem von Axiomen bzw. Definitionen der Objektlogik reden und nur bei Termen, die das Symbol $\Longrightarrow$ enthalten von Regeln sprechen, es sei denn es handelt sich schon in der Objektlogik um eine Regel ohne Prämissen (wie z.B. bei `refl`).

Nun zu den einzelnen Axiomen und Regeln. Als erstes wird noch das eher technisch relevante Metaaxiom `eq_reflection` eingeführt, mit dessen Hilfe Gleichheiten in der Objektlogik in Gleichheiten der Metalogik überführt werden können. Dies ist notwendig für den Einsatz des Simplifikationspakets, das ausschließlich auf Metagleichheiten arbeitet [Pau94, Reference Manual, Kapitel 10]. Die Regeln `refl`, `subst`, `ImpI` und `mp` sind bereits aus Abschnitt 2.5 bekannt. Statt der Axiome (eta) und (choice) der Theorie *Init* (Abschnitt 2.6) werden hier die Regeln `ext` und `selectI` eingeführt, was zu einer äquivalenten Formalisierung führt. Die restlichen Regeln [hyp], [beta], [abstract] und [type_inst] des Deduktionssystems D_{Min} werden bereits durch die Metalogik von Isabelle abgedeckt. Danach folgen, bis auf die Definitionen der Konstanten `inj` und `surj`, die in Theorie `Set` nachgeholt werden, exakt die Definitionen für die logischen Konstanten der Theorie *Log* aus Abschnitt 2.6. Von den Axiomen der Theorie *Init* sind hier nur die beiden Axiome (iff) und (True_or_False) übrig geblieben. Statt der Axiome (eta) und (choice) sind wie gesagt die Regeln `ext` und `selectI` verwendet worden, und das Axiom (infinity) folgt in leicht variierter Form in der noch folgenden Theorie `Nat`. Zuletzt werden noch die Konstanten `If`, `Inv` und `o` definiert, wobei die Definitionen von `If` und `Inv` sehr eindrucksvoll den Einsatz des Hilbert-Deskriptors ε zeigen.

Mit der Isabelle-Theorie `HOL` steht nun eine Logik zur Verfügung, die in etwa der Logik Q_0 von Andrews [And86] entspricht, da das Unendlichkeitsaxiom (infinity) der Logik HOLC noch nicht hinzugenommen wurde. Für `HOL` werden im Isabelle-System diverse Theoreme abgeleitet, wobei es sich vor allem um abgeleitete Regeln für die Verwendung der logischen Symbole handelt. Eine Auflistung dieser Theoreme findet sich zum Beispiel in [Pau94].

Ich werde nun kurz die weiteren Theorien aus Abbildung 3.1 beschreiben. Im Gegensatz zur Theorie `HOL`, die ich noch relativ ausführlich dargestellt habe, werde ich bei den nun folgenden Theorien nur mehr die für das Kapitel 4 relevanten Konstanten zusammen mit ihren wichtigsten (abgeleiteten) Eigenschaften darstellen.

3.2.2 Theorie `Set`

Mit der Theorie `Set` wird getypte Mengentheorie eingeführt. Die Abbildung 3.3 zeigt nur einen geringen Bruchteil davon.

```
Set = HOL +
types
  α set
arities
  set :: (term) term
consts
  (* Constants *)
  Collect       ::  (α ⇒ bool) ⇒ α set            (*comprehension*)
   ∈            ::  [α, α set] ⇒ bool              (infixl 50)
```

```
  range          ::  (α ⇒ β) ⇒ β set              (*of function*)
  inj, surj      ::  (α ⇒ β) ⇒ bool               (*inj/surjective*)

  (** Binding Constants **)
   @Coll         ::  [idt, bool] ⇒ α set            ( {_./ _} )

translations
   {x. P}  ≡  Collect(λx. P)

rules
  mem_Collect_eq     (a ∈ {x.P(x)}) = P(a)
  Collect_mem_eq     {x.x∈A} = A

  (* Definitions *)
  range_def       range(f) ≡ {y. ∃x. y=f(x)}
  inj_def         inj(f) ≡ ∀x y. f(x)=f(y) → x=y
  surj_def        surj(f) ≡ ∀y. ∃x. y=f(x)
end
```

Abbildung 3.3: Theorie `Set`

Der Typ α `set` steht für die Potenzmenge des Typs α, die isomorph zum Typ $\alpha \Rightarrow$ `bool` aller Prädikate über α ist. Die Konstanten für den Isomorphismus sind `Collect`::$(\alpha \Rightarrow$ `bool`$) \Rightarrow \alpha$ `set` und $\in$::`[`α`,` α `set]` $\Rightarrow$ `bool`. Statt der unhandlichen Schreibweise `Collect(`λ`x. P)` wird jedoch in der Sektion `translations` die Mixfix-Syntax `{x. P}` eingeführt. In dieser Schreibweise sind dann auch die Isomorphie-Axiome `mem_Collect_eq` und `Collect_mem_eq` formuliert. Die Rolle der Konstanten $\in$ beim Isomorphismus zwischen α `set` und $\alpha \Rightarrow$ `bool` wird deutlicher, wenn man im Typ `[`α`,` α `set]` $\Rightarrow$ `bool` von $\in$ die Position der Argumente vertauscht. Man erhält dann (α `set`) $\Rightarrow$ ($\alpha \Rightarrow$ `bool`).

Die Funktion `range` bildet eine Funktion `f` in ihre Bildmenge ab. Dies wird durch die Definition `range_def` ausgedrückt. Die Prädikate `inj` und `surj` stehen für Injektivität und Surjektivität von Funktionen. Man vergleiche die Definitionen `inj_def` und `surj_def` mit den entsprechenden Definitionen der Theorie *Log* aus Abschnitt 2.6.

Die für uns interessanten abgeleiteten Regeln der Theorie `Set` sind in Abbildung 3.4 dargestellt.

```
CollectI    ⟦ P(a)⟧⟹a ∈ {x.P(x)}
CollectD    ⟦ a ∈ {x.P(x)}⟧⟹P(a)
CollectE    ⟦ a ∈ {x.P(x)}; P(a)⟹W⟧⟹W
rangeI      f(x) ∈ range(f)
rangeE      ⟦ b ∈ range(λx.f(x));  ⋀x. b=f(x)⟹P⟧⟹P
injI        ⟦ ⋀x y. f(x) = f(y)⟹x=y⟧⟹inj(f)
injD        ⟦ inj(f); f(x) = f(y)⟧⟹x=y
```

Abbildung 3.4: Theoreme der Theorie `Set`

3.2.3 Theorie Prod

Aus der Theorie Prod für das kartesische Produkt möchte ich nur die Signatur und einige abgeleitete Regeln präsentieren.

```
Prod = Set +
types
  (α,β)  *           (infixr 20)
arities
  *  :: (term,term)term
consts
  fst      ::  α * β ⇒ α
  snd      ::  α * β ⇒ β
  split    ::  [α * β, [α,β] ⇒ γ] ⇒ γ
  Pair     ::  [α,β] ⇒ α * β
end
```

Abbildung 3.5: Signatur der Theorie Prod

Statt der Notation Pair(a,b) darf die Notation <a,b> verwendet werden. Die entsprechende Einführung der Mixfix-Notation <_>, wie auch die Axiome der Theorie Prod, habe ich hier unterschlagen. Für die obigen Konstanten gelten die erwarteten Eigenschaften, die in Abbildung 3.6 zusammengestellt sind.

```
Pair_inject    ⟦ <a,b> = <a',b'>;  ⟦ a=a';  b=b' ⟧⟹R ⟧⟹R
Pair_eq        (<a,b> = <a',b'>) = (a=a' ∧ b=b')
PairE          ⟦ ⋀x y. p = <x,y>⟹Q ⟧⟹Q

fst_conv       fst(<a,b>) = a
snd_conv       snd(<a,b>) = b
split          split(<a,b>, c) = c(a,b)
```

Abbildung 3.6: Theoreme der Theorie Prod

3.2.4 Theorie Sum

Die Summe α + β der beiden Typen α und β ist in gewissem Sinne dual zum Produkt definiert.

```
Sum = Prod +
types
  (α,β)  +           (infixl 10)
arities
  +  :: (term,term)term
consts
```

```
  Inl      ::  α ⇒ α+β
  Inr      ::  β ⇒ α+β
  sum_case ::  [α+β, α ⇒ γ, β ⇒ γ] ⇒ γ
end
```

Abbildung 3.7: Signatur der Theorie `Sum`

Die wichtigsten abgeleiteten Eigenschaften sind in Abbildung 3.8 dargestellt.

```
Inl_not_Inr   Inl(a) ¬= Inr(b)
inj_Inl       inj(Inl)
inj_Inr       inj(Inr)

sum_case_Inl  sum_case(Inl(x), f, g) = f(x)
sum_case_Inr  sum_case(Inr(x), f, g) = g(x)

sumE          ⟦ ⋀x::α. s=Inl(x)⟹P; ⋀y::β. s=Inr(y)⟹P⟧⟹P
```

Abbildung 3.8: Theoreme der Theorie `Sum`

3.2.5 Theorien `Lfp`, `Trancl` und `WF`

Die beiden Theorien `Lfp` und `Trancl` möchte ich an dieser Stelle nur mit Worten beschreiben, da sowohl ihre Signatur als auch ihre Eigenschaften im folgenden nicht explizit auftreten. Ich möchte sie aber trotzdem anführen, da sie die Grundlage für die Einführung der natürlichen Zahlen in der Theorie `Nat` darstellen.

Die Theorie `Lfp` führt einen Fixpunktoperator für monotone Mengenfunktionale ein, und zu den Theoremen der Theorie gehört der Fixpunktsatz von Knaster-Tarski. In der Theorie `Trancl` werden Operatoren zur Bildung der transitiven bzw. transitiven und reflexiven Hülle von Mengenrelationen eingeführt.

In der Theorie `WF` wird neben anderen Dingen auch das Konzept der fundierten Ordnung (well-founded relation) eingeführt, welches Noethersche Induktion erlaubt. Da diese Form der Induktion auch für HOLCF interessant sein kann, werde ich hier wieder kurz die relevante Signatur nebst Eigenschaften darstellen.

```
WF = Trancl +
consts
   wf            ::  (α * α)set ⇒ bool
end

wf_induct     ⟦ wf(r);
                ⋀x.⟦ ∀y. <y,x>∈ r → P(y)⟧⟹P(x)⟧
                ⟹P(a)
```

Abbildung 3.9: Signatur und Eigenschaften der Theorie `WF`

3.2.6 Theorie `Nat`

Die Theorie `Nat` ist die letzte Theorie, die ich in diesem Kapitel vorstellen möchte. Auf ihr bauen dann die Theorien aus Kapitel 4 auf. Die Theorie `Nat` ist in Abbildung 3.10 dargestellt.

```
Nat = WF +
types
  ind
  nat

arities
  ind,nat :: term

consts
  Zero_Rep      ::  ind
  Suc_Rep       ::  ind ⇒ ind

   0            ::  nat
  Suc           ::  nat ⇒ nat

  nat_case      ::  [nat, α, nat ⇒ α] ⇒ α
  nat_rec       ::  [nat, α, [nat, α] ⇒ α] ⇒ α

   <,<=         ::  [nat, nat] ⇒ bool              (infixl 50)

rules
      (*the axiom of infinity in 2 parts*)

  inj_Suc_Rep            inj(Suc_Rep)
  Suc_Rep_not_Zero_Rep   ¬(Suc_Rep(x) = Zero_Rep)

end
```

Abbildung 3.10: Theorie `Nat`

In den Sektionen `types` und `arities` werden die beiden Typkonstanten `ind` und `nat` eingeführt. Der Typ `ind` (individuals) entspricht dem Typ *ind* der Logik HOLC. Zum einen wird mit seiner Hilfe das Axiom der Unendlichkeit ausgedrückt, zum anderen dient er als Repräsentant für den Typ `nat` der natürlichen Zahlen.

In der Sektion `consts` werden wieder diverse Konstanten eingeführt. `Zero_Rep` und `Suc_Rep` sind die Repräsentanten für die abstrakten Konstanten `0` und `Suc`. Die Konstante `nat_case` ist ein Diskriminatorfunktional für die natürlichen Zahlen, und `nat_rec` ist das Funktional für primitive Rekursion. Die Konstanten `<` bzw. `<=` stehen für die 'kleiner' bzw. 'kleiner gleich' Relation auf den natürlichen Zahlen[19].

In der Sektion `rules` habe ich nur die beiden Axiome aufgeführt, die in Isabelle-HOL die Rolle des Axioms der Unendlichkeit übernehmen. Man vergleiche hierzu das Axiom (infinity) aus der Theorie *Init* im Abschnitt 2.6. Die beiden Axiome `inj_Suc_Rep` und `Suc_Rep_not_Zero_Rep` besagen, daß `Suc_Rep` zwar eine injektive, aber keine surjektive Funktion ist. Hier wird also statt der existentiellen Aussage des Axioms (infinity) der konkreten Einführung von Konstanten der Vorzug gegeben. Dies rührt daher, daß in der Isabelle-Formalisierung die natürlichen Zahlen als kleinster Fixpunkt eines monotonen Mengenfunktionals definiert werden[20] [Pau93a]. Dieses Funktional läßt sich einfacher mittels der Konstanten `Zero_Rep` und `Suc_Rep` aufschreiben. Eine alternative Formulierung für die natürlichen Zahlen, die besser zum Axiom (infinity) paßt, findet sich in [Gor85].

Die wichtigsten Eigenschaften der natürlichen Zahlen sind in Abbildung 3.11 dargestellt.

```
nat_induct          ⟦ P(0); ⋀k. P(k)⟹P(Suc(k))⟧⟹P(n)
natE                ⟦ n=0⟹P; ⋀x. n = Suc(x)⟹P⟧⟹P

Suc_not_Zero        Suc(m) ¬= 0
Suc_Suc_eq          (Suc(m)=Suc(n)) = (m=n)
n_not_Suc_n         n ¬= Suc(n)

nat_case_0          nat_case(0, a, f) = a
nat_case_Suc        nat_case(Suc(k), a, f) = f(k)

nat_rec_0           nat_rec(0,c,h) = c
nat_rec_Suc         nat_rec(Suc(n), c, h) = h(n, nat_rec(n,c,h))

less_trans          ⟦ i<j;  j<k⟧⟹i<k
lessI               n < Suc(n)
less_SucI           i<j⟹i<Suc(j)
zero_less_Suc       0 < Suc(n)

less_not_sym        n<m⟹¬m<n
less_not_refl       ¬n<n
```

[19] eigentlich werden `<` und `<=` als *überladene* Ordnungssymbole in der Theorie `Ord` eingeführt. Um aber unnötige Details zu vermeiden, habe ich in dieser Arbeit diese Symbole als nichtpolymorphe Konstanten des Typs `nat` behandelt.

[20] siehe dazu auch die Isabelle-Theorie Nat.thy.

```
not_less0          ¬n<0
Suc_less_eq        (Suc(m) < Suc(n)) = (m<n)
not_Suc_n_less_n   ¬(Suc(n) < n)

less_linear        m<n ∨ m=n ∨ n<m

leD                m<=n⟹¬(n<m::nat)
leI                ¬(n<m)⟹m<=n::nat

less_induct        [| ⋀n. [| ∀m. m<n → P(m)|]⟹P(n)|]⟹P(n)
```

Abbildung 3.11: Theoreme der Theorie `Nat`

Hiermit sind nun alle Theorien eingeführt, die für die nachfolgende Entwicklung von HOLCF in Kapitel 4 relevant sind. Für eine umfangreichere Darstellung möchte ich nochmals auf [Pau94, Pau93b] verweisen. Erschöpfende Information liefern aber letztendlich nur die Quelltexte der Isabelle-Theorien, in denen eine Vielzahl von Theoremen enthalten sind, die für den Umgang mit den obigen Theorien nützlich sein können.

Kapitel 4

HOLCF: Entwicklung der einzelnen Theorien

Dieses Kapitel stellt den Hauptteil meiner Arbeit dar. Hier stelle ich die Entwicklung der Logik HOLCF und den Großteil der abgeleiteten Theoreme vor. Die Logik HOLCF entsteht durch schrittweise Erweiterung von Isabelle-HOL um Definitionen für bereichstheoretische Konzepte. Jede Theorie wird in einem eigenen Abschnitt dargestellt. Diese Abschnitte sind gegebenenfalls in weitere Unterabschnitte aufgeteilt, wenn die Theorie in mehreren kleinen Schritten eingeführt wird. Diese Unterteilung in mehrere Teilschritte wird dabei hauptsächlich von der Methode der konservativen Erweiterung diktiert.

Die Beschreibung einer einzelnen (Teil-)Theorie untergliedert sich in zwei bzw. manchmal sogar in drei Teile. Im ersten Teil wird immer die Isabelle-Theorie dargestellt, und im zweiten Teil werden die in Isabelle bewiesenen Theoreme der Theorie aufgelistet. Diese beiden Teile werden jeweils durch erklärende Kommentare begleitet, die auf eventuelle Designentscheidungen hinweisen oder einzelne Theoreme näher erläutern. Auf diese Weise kann man sich einen schnellen Überblick über die einzelnen Theorien verschaffen.

Bei einigen Theorien folgt dann noch ein dritter Teil, der explizite Beweise für ausgesuchte Theoreme der Theorie enthält. Die Auswahl der explizit dargestellten Beweise erfolgte nach rein subjektiven Kriterien. So habe ich zum Beispiel Theoreme ausgewählt, deren Beweise meiner Meinung nach das Verständnis für die jeweilige Theorie erhöhen. Bisweilen wurden Theoreme aufgenommen, weil mir ihr Beweis besonders schwierig erschien. Aus Platzgründen mußte ich mich in meiner Auswahl jedoch drastisch beschränken und so fehlen für viele Theoreme die expliziten Beweise, obwohl sie auf die ein oder andere Weise durchaus interessant gewesen wären. Dieser Umstand ist aber nicht sonderlich tragisch, da die Logik HOLCF vollständig in der Isabelle-Distribution enthalten ist. Daher können die in dieser Arbeit fehlenden Beweise bei Bedarf ausführlich am Isabelle-System nachgespielt werden.

Es folgt nun ein kurzer Überblick über die in diesem Kapitel dargestellten Theorien. Ein graphischer Überblick findet sich in Abbildung 4.1. Im Standard-Umfang der Logik LCF sind normalerweise auch die Theorien für den Typ `one`, der nur ein definiertes Element aufweist, und die Theorie `tr` der programmiersprachlichen Wahrheitswerte enthalten. Da diese Typen aber durch einen Definitionsmechanismus eingeführt werden, dessen Rechtfertigung erst durch

die in Kapitel 5 dargestellte Theorie möglich ist, wird ihre Einführung auf das Kapitel 6 verschoben.

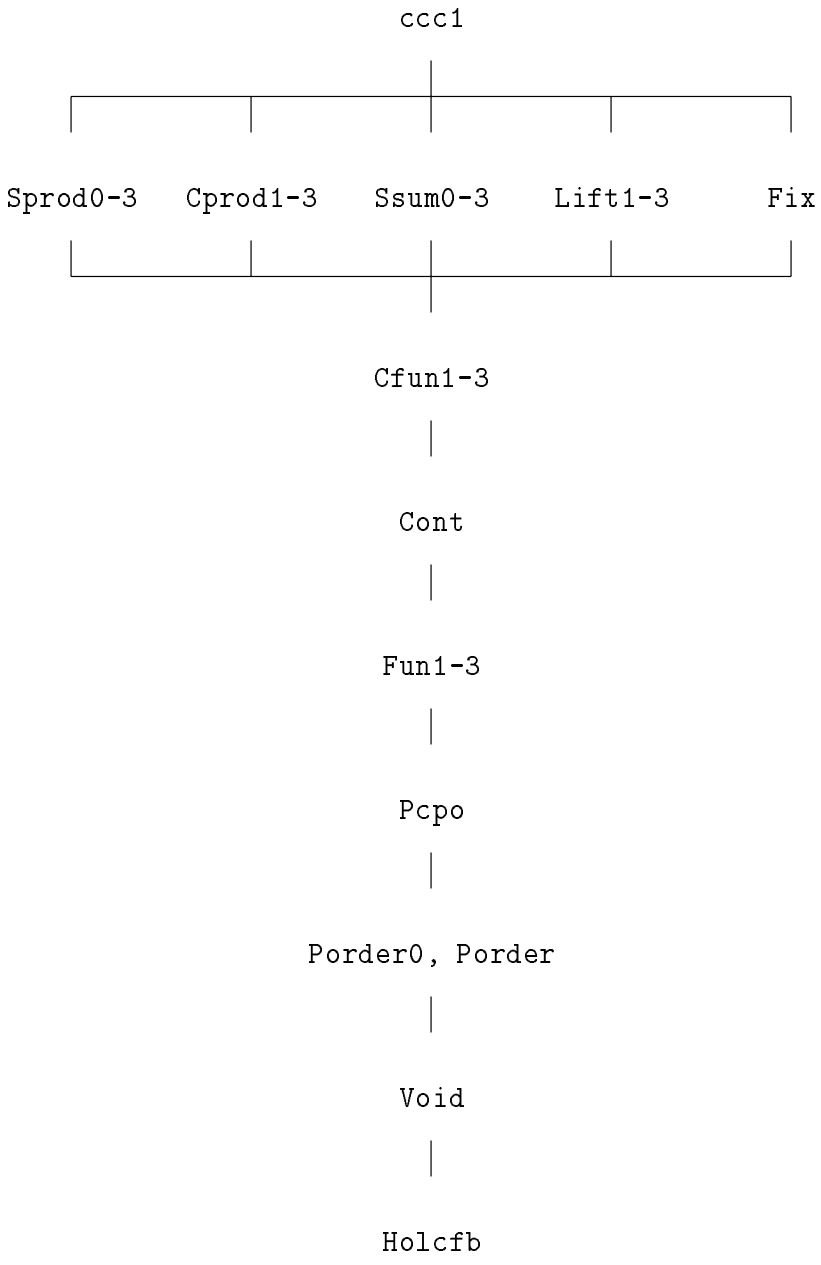


Abbildung 4.1: Hierarchie der HOLCF-Theorien in Isabelle

Im Abschnitt 4.1 (`Holcfb`) werden zunächst ein Minimierungsoperator für die natürlichen Zahlen eingeführt sowie einige nützliche prädikatenlogische Theoreme bewiesen. Danach entwickle ich in Abschnitt 4.2 (`Void`) eine Theorie für den trivialen Typ *void*, der nur ein Element in der Trägermenge enthält. Auf diesem Typ definiere ich dann die offensichtliche Ordnung und weise die Eigenschaften einer partiellen und später einer kettenvollständigen

partiellen Ordnung mit kleinstem Element nach. Der Typ *void* dient als Rechtfertigung für die Typklassen der partiellen Ordnungen *po* und der ω-kettenvollständigen partiellen Ordnungen *pcpo* mit kleinstem Element, die in den darauffolgenden Abschnitten 4.3 (`Porder0, Porder`) und 4.4 (`Pcpo`) eingeführt werden. Im Abschnitt 4.5 (`Fun1-3`) wird für Funktionen auf Bereichstypen (Typen in Klasse *pcpo*) eine Ordnungsrelation definiert, und die Ordnungseigenschaften werden nachgewiesen. Dies rechtfertigt dann auch die zusätzliche Zuordnung von Klasseninformation für den Typkonstruktor $\Rightarrow$, die besagt, daß $\Rightarrow$ Bereichstypen wieder in Bereichstypen abbildet. Der Abschnitt 4.6 (`Cont`) bringt Definitionen für monotone und stetige Funktionen auf Bereichstypen. Im Abschnitt 4.7 (`Cfun1-3`) werden diese Ergebnisse über stetige Funktionen ausgenutzt, um den Typkonstruktor $\rightarrow$ der Operationen sowie eine Ordnung auf Operationen zu definieren und die Ordnungseigenschaften dafür abzuleiten. Zudem werden wieder zusätzliche Zuordnungen von Klasseninformation für den Typkonstruktor $\rightarrow$ vorgenommen, die ausdrücken, daß der Typ der Operationen über Bereichstypen ebenfalls in der Klasse *pcpo* der Bereiche liegt. Weiterhin werden hier die Theoreme zur Propagierung der Stetigkeit von Termen abgeleitet. Das gleiche Programm wird im Abschnitt 4.8 (`Sprod0-3`) für das strikte Produkt, in 4.9 (`Cprod1-3`) für das kartesische Produkt, in 4.10 (`Ssum0-3`) für die strikte Summe und in 4.11 (`Lift1-3`) für den gelifteten Bereich durchgeführt. In Abschnitt 4.12 (`Fix`) wird dann die Fixpunkttheorie für Operationen formalisiert. Speziell werden hier der Fixpunktoperator auf Operationen *fix* und das Zulässigkeitsprädikat *adm* definiert und deren wesentlichen Eigenschaften abgeleitet. Dazu zählen die Fixpunktinduktion und die Propagierung der Zulässigkeit. In Abschnitt 4.13 (`ccc1`) werden dann noch eine Komposition für Operationen und eine Identitätsoperation eingeführt, was die spätere Axiomatisierung rekursiver Bereichstypen notationell vereinfacht.

4.1 Basistheorie für HOLCF

Die Theorie `Holcfb` enthält die Definition für einen Minimierungsoperator `theleast` auf den natürlichen Zahlen. Dieser Operator wird in der Theorie `Fix` dazu benutzt, um die Propagierung der Zulässigkeit für die Disjunktion zu beweisen.

4.1.1 Die Theorie `Holcfb`

Die Theorie `Holcfb` ist in Abbildung 4.2 dargestellt.

```
Holcfb = Nat +

consts   theleast :: (nat ⇒ bool) ⇒ nat

rules
theleast_def     theleast(P) ≡ (εz.(P(z) ∧ (∀n.P(n) → z<=n)))
end
```

Abbildung 4.2: Theorie `Holcfb`

Der Operator `theleast` wird mit Hilfe des Auswahloperators ε definiert. Da die Ordnung `<=` eine Wohlordnung auf den natürlichen Zahlen ist, ist das kleinste Element jeder nichtleeren Menge eindeutig bestimmt.

4.1.2 Theoreme der Theorie `Holcfb`

Die Theoreme der Theorie `Holcfb` sind in Abbildung 4.3 dargestellt.

```
well_ordering_nat
              ∀P. P(x::nat) → (∃y. P(y) ∧ (∀x. P(x) → y <= x))

theleast1     P1(x1)⟹P1(theleast(P1))
theleast2     P(x)⟹theleast(P) <= x

de_morgan1    (¬a ∧ ¬b)=(¬(a ∨ b))
de_morgan2    (¬a ∨ ¬b)=(¬(a ∧ b))
notall2ex     (¬(∀x.P(x))) = (∃x.¬P(x))
notex2all     (¬(∃x.P(x))) = (∀x.¬P(x))

selectI2      (∃x. P(x))⟹P(εx.P(x))
selectE       P(εx.P(x))⟹(∃x. P(x))
select_eq_Ex  (P(εx.P(x))) =  (∃x. P(x))

notnotI       P⟹¬¬P
classical2    ⟦ Q⟹R; ¬Q⟹R⟧⟹R
classical3    ⟦ P⟹¬P1; P⟹P1⟧⟹¬P

nat_less_cases
              ⟦ m::nat < n⟹P(n,m);
                 m=n⟹P(n,m);
                 n < m⟹P(n,m)⟧⟹P(n,m)
```

Abbildung 4.3: Theoreme der Theorie `Holcfb`

Das Theorem `well_ordering_nat` drückt aus, daß `<=` eine Wohlordnung auf dem Typ `nat` ist. Dies kann durch vollständige Induktion bewiesen werden. Die Theoreme `theleast1` und `theleast2` beschreiben die Eigenschaften des Operators `theleast`. Die restlichen Theoreme wurden abgeleitet, da sie sich bei der Entwicklung von HOLCF als nützlich erwiesen haben.

4.2 Der Typ `void`

Der Typ `void` ist ein Typ, der nur ein Element in der Trägermenge enthält. Dieser Typ wird eingeführt, weil sich auf ihm in trivialer Weise eine partielle Ordnung definieren läßt, die kettenvollständig ist und ein kleinstes Element enthält. Der Typ `void` dient als Prototyp für die Klassen `po` und `pcpo`.

4.2.1 Die Theorie Void

Die Theorie Void ist in Abbildung 4.4 dargestellt. Der Typ void wird nach der Methode der konservativen Erweiterung eingeführt, die in Abschnitt 2.6 beschrieben wurde. Im praktischen Umgang mit Isabelle-HOL wird dieser Erweiterungsmechanismus in leicht abgeänderter Form eingesetzt, wie ich gleich im folgenden näher erläutern werde.

```
Void = Holcfb +

types   void 0
arities void :: term

consts
  Void          ::  bool set
  ⊥_void_Rep    ::  bool
  Rep_Void      ::  void ⇒ bool
  Abs_Void      ::  bool ⇒ void
  ⊥_void        ::  void
  less_void     ::  [void,void] ⇒ bool

rules

  (* The unique element in Void is False::bool *)

  ⊥_void_Rep_def         ⊥_void_Rep ≡ False
  Void_def               Void ≡ {x. x = ⊥_void_Rep}

  (* faking a type definition... *)
  (* void is isomorphic to Void  *)

  Rep_Void               Rep_Void(x)∈Void
  Rep_Void_inverse       Abs_Void(Rep_Void(x)) = x
  Abs_Void_inverse       y∈Void⟹Rep_Void(Abs_Void(y)) = y

  (* defining the abstract constants *)

  ⊥_void_def     ⊥_void ≡ Abs_Void(⊥_void_Rep)
  less_void_def  less_void(x,y) ≡ (Rep_Void(x) = Rep_Void(y))
end
```

Abbildung 4.4: Theorie Void

Die in Abschnitt 2.6 vorgestellte Methode der konservativen Erweiterung durch Typdefinition sieht vor, daß ein Repräsentationsprädikat P::τ ⇒ bool für den repräsentierenden Typ τ eingeführt wird, und daß das Theorem ∃x.P(x) gezeigt wird. In Isabelle-HOL wird dagegen per Konvention eine Teilmenge RepSet::τ set eingeführt und ein Theorem t∈RepSet für einen

beliebigen Term t::τ gezeigt. Dies ist aufgrund der Semantik des Typkonstruktors `set` und der Elementrelation $\in$ äquivalent zur Methode, die in Abschnitt 2.6 vorgestellt wurde.

Im Beispiel der Theorie `Void` wird die dem Repräsentationsprädikat entsprechende Teilmenge durch die Konstante `Void::bool set` verkörpert. Die Definitionen `⊥_void_Rep_def` und `Void_def` besagen, daß die Menge `Void` nur aus dem Element `False` besteht. Die Verwendung der redundanten Konstanten `⊥_void_Rep` ist reine Geschmackssache. Um nachzuweisen, daß die Teilmenge `Void` nicht leer ist, zeigt man das triviale Theorem `⊥_void_Rep∈Void`. Strenggenommen müßten die Teilmenge `Void` und das Theorem `⊥_void_Rep∈Void` vorher in einer separaten Theorie eingeführt werden, und erst in einem nächsten Schritt dürfte der Typkonstruktor `void` mit den Typdefinitionsaxiomen eingeführt werden. Wenn die Herleitung des Theorems aber wie hier trivial aus der Definition der Teilmenge folgt, erlaubt man sich die simultane Einführung der repräsentierenden Teilmenge und der anschließenden Typdefinition. Puristen dürfen selbverständlich an der korrekten schrittweisen Einführung festhalten. Die Axiome `Rep_Void`, `Rep_Void_inverse` und `Abs_Void_inverse` sind die Typdefinitionsaxiome für den Typ `void`. Die Verwendung der Teilmenge `Void` und der Elementrelation $\in$ liefert eine zu Abschnitt 2.6 äquivalente Form der Typdefinition.

Zusätzlich zum Typ `void` werden noch die beiden Konstanten `⊥_void` und `less_void` definiert. Die Konstante `⊥_void` steht für das kleinste und einzige Element im Typ `void`, und `less_void` steht für die triviale Ordnung auf dem Typ `void`. Auch hier wurde darauf verzichtet, die Konstantendefinitionen in einer separaten Theorie vorzunehmen.

4.2.2 Theoreme der Theorie `Void`

Die Theoreme der Theorie `Void` sind in Abbildung 4.5 dargestellt. Das Theorem `VoidI` rechtfertigt die Einführung des Typs `void`. Die Theoreme `refl_less_void`, `antisym_less_void` und `trans_less_void` zeigen, daß `less_void` den Typ `void` partiell ordnet. Diese drei Theoreme erlauben die nachfolgende Einführung der Klasse `po`, wobei der Typ `void` als Prototyp angegeben wird.

```
VoidI      ⊥_void_Rep ∈ Void

refl_less_void       less_void(x,x)
antisym_less_void    ⟦ less_void(x,y); less_void(y,x)⟧⟹x = y
trans_less_void      ⟦ less_void(x,y); less_void(y,z)⟧⟹less_void(x,z)
```

Abbildung 4.5: Theoreme der Theorie `Void`

4.3 Die Klasse `po`

Dieser Abschnitt beschreibt die Theorien `Porder0` und `Porder`. In der Theorie `Porder0` wird die Klasse `po` der partiell geordneten Typen eingeführt. Die Theorie `Porder` baut auf `Porder0` auf und enthält Definitionen für obere Schranken, kleinste obere Schranken und Ketten.

4.3.1 Die Theorie Porder0

Die Theorie Porder0 ist in Abbildung 4.6 dargestellt. In ihr wird die Klasse po durch konservative Erweiterung eingeführt. Die Einführung der Klasse po wird durch die Theoreme refl_less_void, antisym_less_void und trans_less_void der Theorie Void gerechtfertigt. Der Prototyp für die Klasse po ist der Typ void. Das Beispiel der Klasse po wurde in dieser Arbeit schon mehrfach erwähnt und bedarf daher keiner weiteren Erklärung.

```
Porder0 = Void +

classes po < term
arities void :: po

consts    ⊑      ::       [α,α_po] ⇒ bool    (infixl 55)

rules

refl_less        x ⊑ x
antisym_less     ⟦ x ⊑ y ; y ⊑ x⟧⟹x = y
trans_less       ⟦ x ⊑ y ; y ⊑ z⟧⟹x ⊑ z

inst_void_po     ( ⊑ ::[void,void] ⇒ bool) = less_void
end
```

Abbildung 4.6: Theorie Porder0

Per Konvention werden alle in der Theorie eingeführten Konstanten, in diesem Fall also $\sqsubseteq ::[\alpha, \alpha_{\mathtt{po}}] \Rightarrow \mathtt{bool}$, als charakteristische Konstanten der Klasse aufgefaßt. Weiterhin werden alle Axiome außer den Instanzaxiomen für den Prototyp, die an ihrer Form leicht zu erkennen sind, als charakteristische Axiome der Klasse interpretiert. Im Fall der Klasse po sind dies also die Axiome refl_less, antisym_less und trans_less. Diese Konvention zeichnet in eindeutiger Weise die charakteristischen Konstanten und Axiome einer Klasse aus. Sie ist notwendig, da in Isabelle keine spezielle syntaktische Kennzeichnung von charakteristischen Konstanten und Axiomen möglich ist, wie sie die Logik HOLC aus Kapitel 2 vorsieht.

4.3.2 Die Theorie Porder

Die Theorie Porder ist in Abbildung 4.7 dargestellt. In ihr werden mehrere Konstanten eingeführt, die relativ zur charakteristischen Konstanten $\sqsubseteq$ der Klasse po definiert sind. Der Term $\mathtt{S} \lhd \mathtt{x}$ wird gelesen als '$\mathtt{x}$ ist obere Schranke der Menge S', und $\mathtt{S} \lll \mathtt{x}$ heißt '$\mathtt{x}$ ist kleinste obere Schranke der Menge S'. Da kleinste obere Schranken eindeutig sind, wenn sie existieren, kann mit Hilfe des Hilbert Operators ε die Funktion lub definiert werden, so daß lub(S) die kleinste obere Schranke von S bezeichnet, falls diese existiert. Man beachte die Anwendung der Logik höherer Stufe und der Typklassen, die in ihrer Kombination eine elegante Notation ermöglichen.

```
Porder = Porder0 +

consts
         <|      ::      [α set,α_po] ⇒ bool        (infixl 55)
         <<|     ::      [α set,α_po] ⇒ bool        (infixl 55)
        lub      ::      α set ⇒ α_po
        is_chain ::      (nat ⇒ α_po) ⇒ bool

        max_in_chain ::  [nat,nat ⇒ α_po] ⇒ bool
        finite_chain ::  (nat ⇒ α_po) ⇒ bool

rules

is_ub            S <| x ≡ ∀y.y∈S → y ⊑ x
is_lub           S <<| x ≡ S <| x ∧ (∀u. S <| u → x ⊑ u)
lub              lub(S) = (εx. S <<| x)

is_chain         is_chain(F) ≡ (∀i.F(i) ⊑ F(Suc(i)))

max_in_chain_def  max_in_chain(i,C) ≡ ∀j. i <= j → C(i) = C(j)
finite_chain_def  finite_chain(C) ≡ is_chain(C) ∧ (∃i. max_in_chain(i,C))

end
```

Abbildung 4.7: Theorie `Porder`

Durch das Prädikat `is_chain` wird der Begriff der aufsteigenden und nicht leeren ω-Kette formalisiert. Es mag im ersten Moment überraschen, daß hier die abzählende Funktion `F` selbst als Kette bezeichnet wird und nicht ihr Bild. Die hier verwendete Definition des Kettenbegriffs und die Totalität der HOL-Funktionen haben zur Folge, daß die leere Menge nie das Bild einer Kette sein kann. Dieser Umstand ist wichtig für die späteren Definitionen der vollständigen Ordnungen[1] und der stetigen Funktionen[2].

Zuletzt werden noch die beiden Prädikate `max_in_chain` und `finite_chain` eingeführt, die die Argumentation über endliche Ketten erleichtern. Eine Kette heißt dabei endlich, wenn ihr Bild eine endliche Menge ist. Dieser Umstand wird aus technischen Gründen wieder indirekt beschrieben, indem man verlangt, daß die Abzählung ab einem gewissen Index konstant ist.

4.3.3 Theoreme der Theorien `Porder0` und `Porder`

Die Theoreme der beiden Theorien `Porder0` und `Porder` sind in Abbildung 4.8 dargestellt. Die meisten Theoreme bedürfen keiner Erklärung, da sie aufgrund der obigen Definitionen zu erwarten waren. Einige Beweise sind in Abschnitt 4.3.4 genauer ausgeführt. Die Definition des Kettenbegriffs hat zur Folge, daß in Axiomen und Theoremen häufig die Funktion `range`:: $(\alpha \Rightarrow \beta) \Rightarrow \beta$ `set` auftaucht, die die Bildmenge `range(S)` einer Funktion `S` liefert.

[1] siehe Theorie `Pcpo` in Abschnitt 4.4
[2] siehe Theorie `Cont` in Abschnitt 4.6

Besonders erwähnen möchte ich noch die beiden Theoreme `minimal_void` und `cpo_void`, die besagen, daß der Typ `void` ein kleinstes Element hat, und daß es für jede Kette `S::nat ⇒ void` im Typ `void` eine kleinste obere Schranke gibt. Aufgrund der trivialen Struktur des Typs `void` ist dies aber nicht weiter verwunderlich! Diese beiden Theoreme werden die Einführung der Klasse `pcpo` der kettenvollständigen partiellen Ordnungen mit kleinstem Element rechtfertigen, die in der Theorie `Pcpo` vorgenommen wird.

```
antisym_less_inverse  x=y⟹x ⊑ y ∧ y ⊑ x
box_less        ⟦ a ⊑ b; c ⊑ a; b ⊑ d⟧⟹c ⊑ d
unique_lub      ⟦ S ≪| x ; S ≪| y⟧⟹x=y

chain_mono      is_chain(F)⟹x<y → F(x) ⊑ F(y)
chain_mono3     ⟦ is_chain(F); x <= y⟧⟹F(x) ⊑ F(y)

lubI            (∃x. M ≪| x)⟹M ≪| lub(M)
lubE            M ≪| lub(M)⟹∃x. M ≪| x
lub_eq          (∃x. M ≪| x)  = M ≪| lub(M)
thelubI         M ≪| l⟹lub(M) = l
is_lubE         S ≪| x⟹S <| x ∧ (∀u. S <| u → x ⊑ u)
is_lubI         S <| x ∧ (∀u. S <| u → x ⊑ u)⟹S ≪| x
is_chainE       is_chain(F)⟹∀i. F(i) ⊑ F(Suc(i))
is_chainI       ∀i. F(i) ⊑ F(Suc(i))⟹is_chain(F)
ub_rangeE       range(S) <| x⟹∀i. S(i) ⊑ x
ub_rangeI       ∀i. S(i) ⊑ x⟹range(S) <| x
is_ub_lub       range(S1) ≪| x1⟹S1(x) ⊑ x1
is_lub_lub      ⟦ S3 ≪| x3; S3 <| x1⟧⟹x3 ⊑ x1

minimal_void    ⊥_void ⊑ x
cpo_void        is_chain(S::nat ⇒ void)⟹∃x. range(S) ≪| x

lub_finch1      ⟦ is_chain(C) ; max_in_chain(i,C)⟧⟹range(C) ≪| C(i)
lub_finch2      finite_chain(C)⟹range(C) ≪| C(εi. max_in_chain(i,C))
bin_chain       x ⊑ y⟹is_chain(λi. if(i=0,x,y))
bin_chainmax    x ⊑ y⟹max_in_chain(Suc(0),λi. if(i=0,x,y))
lub_bin_chain   x ⊑ y⟹range(λi. if(i = 0,x,y)) ≪| y

lub_chain_maxelem
                ⟦ is_chain(Y);∃i.Y(i)=c;∀i.Y(i) ⊑ c⟧⟹lub(range(Y)) = c
lub_const       range(λx.c) ≪| c
```

Abbildung 4.8: Theoreme der Theorien `Porder0` und `Porder`

4.3.4 Beweise für ausgesuchte Theoreme

In diesem Abschnitt werde ich die Isabelle-Beweise für die Theoreme `lub_chain_maxelem` und `lub_const` vorstellen. Ich stelle die Beweise in der Form dar, die sich durch die Interaktion

mit dem Isabelle-System ergibt. Wie bereits in Kapitel 3 dargestellt, werden Beweise in Isabelle zielorientiert geführt. Beginnend mit dem eigentlichen Beweisziel (goal) führt man dieses schrittweise auf kleinere bzw. einfachere Teilziele (subgoals) zurück, bis man nur noch triviale Beweisaufgaben zu lösen hat. Die Reduktion eines Ziels auf Teilziele erfolgt durch Resolution[3] des aktuellen Beweiszustandes mit Regeln des Beweissystems.

4.3.4.1 Beweis für das Theorem `lub_chain_maxelem`

Wir beginnen, indem wir uns das gewünschte Beweisziel vorgeben[4]:

```
val prems = goal Porder.thy
 "[| is_chain(Y);∃i.Y(i)=c;∀i.Y(i) ⊑ c|]==>lub(range(Y)) = c";
- by (cut_facts_tac prems 1);
```

Das System präsentiert uns das vorerst einzige Beweisziel:

```
 1. [| is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c|]==>lub(range(Y)) = c
```

Resolution mit der Regel[5]

```
 ?M ≪| ?l==>lub(?M) = ?l
- by (rtac thelubI 1);
```

bringt das Prädikat $\ll|$ ins Spiel und liefert das neue Ziel:

```
 1. [| is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c|]==>range(Y) ≪| c
```

Auffaltung der Definition für $\ll|$ mittels der Regel

```
 ?S <| ?x ∧ (∀u. ?S <| u → ?x ⊑ u)==>?S ≪| ?x
- by (rtac is_lubI 1);
```

liefert das Ziel:

```
 1. [| is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c|]==>
    range(Y) <| c ∧ (∀u. range(Y) <| u → c ⊑ u)
```

Eine Konjunktion beweist man im Kalkül des natürlichen Schließens, indem man für jedes Konjugat einen Beweis liefert. Mittels der Regel

```
 [| ?P; ?Q|]==>?P ∧ ?Q
- by (rtac conjI 1);
```

spalten wir unser Ziel in zwei kleinere Teilziele auf:

```
 1. [| is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c|]==>range(Y) <| c
 2. [| is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c|]==>
    ∀u. range(Y) <| u → c ⊑ u
```

Das erste Ziel löst sich leicht durch Anwendung der Regel

[3] siehe Kapitel 3.

[4] eine eingehende Beschreibung der Interaktion mit dem Isabelle-System findet sich in [Pau94].

[5] die ? vor den Variablen kennzeichnen diese als Unifikationsvariablen. Nur solche Variablen können bei Anwendung der Resolutionsregel des Metakalküls instantiiert werden.

```
 ∀i. ?S(i) ⊑ ?x⟹range(?S) <| ?x
- by (etac ub_rangeI 1);
```

mit anschließendem Beweis per Annahme (etac). Es bleibt nun noch das Ziel

```
 1. ⟦ is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c⟧⟹
    ∀u. range(Y) <| u → c ⊑ u
```

welches jetzt die neue Teilzielnummer 1 trägt. Die Methodik des natürlichen Schließens sieht hier vor, das Ziel für ein beliebiges, aber festes u zu beweisen und die Prämisse der Implikation in den Annahmenkontext zu übernehmen. Dieser schrittweise Aufbau des Annahmenkontextes könnte durch einzelne Anwendung der Regeln

```
allI  (⋀x. ?P(x))⟹∀x. ?P(x)
impI  (?P⟹?Q)⟹?P → ?Q
```

erreicht werden. Isabelle bietet hier aber bereits die Taktik strip_tac an, die diese Regeln je nach Bedarf mehrmals anwendet.

```
- by (strip_tac 1);

 1. ⋀u. ⟦ is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c; range(Y) <| u⟧⟹c ⊑ u
```

An dieser Stelle müssen wir jetzt die existenzielle Aussage im Annahmenkontext ausnützen. Dies geschieht durch Anwendung der Existenzelimination

```
 ⟦ ∃x. ?P(x); ⋀x. ?P(x)⟹?Q⟧⟹?Q
- by (res_inst_tac [("P","λi.Y(i)=c")] exE 1);
```

wobei wir die Instantiierung fest vorgegeben haben. Wir erhalten:

```
 1. ⋀u. ⟦ is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c; range(Y) <| u⟧⟹
        ∃x. Y(x) = c
 2. ⋀u x. ⟦ is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c; range(Y) <| u;
             Y(x) = c⟧⟹c ⊑ u
```

Die explizite Instantiierung mittels res_inst_tac wäre hier nicht nötig gewesen, wenn wir die Isabelle-Taktik (etac) verwendet hätten. So müssen wir das erste Teilziel explizit per Annahme beweisen. Das verbleibende Teilziel erhält die Nummer 1:

```
- by (atac 1);

 1. ⋀u x. ⟦ is_chain(Y); ∃i. Y(i) = c; ∀i. Y(i) ⊑ c; range(Y) <| u;
             Y(x) = c⟧⟹c ⊑ u
```

Laut Annahme ist Y(x) = c für einen beliebigen aber festen Index x. Per Gleichungslogik dürfen wir demnach überall die Konstante c durch den Term Y(x) ersetzen. Dies geschieht durch Anwendung einer speziellen Substitutionstaktik:

```
- by (hyp_subst_tac 1);

 1. ⋀u x. ⟦ range(Y) <| u; ∀i. Y(i) ⊑ Y(x); ∃i. Y(i) = Y(x);
             is_chain(Y)⟧⟹Y(x) ⊑ u
```

Wir kombinieren nun die beiden Regeln
```
ub_rangeE   range(?S) <| ?x==>∀i. ?S(i) ⊑ ?x
spec        ∀x. ?P(x)==>?P(?x)
```

durch Vorwärtsresolution `RS` und erhalten die neue Regel:
```
(ub_rangeE RS spec)   range(?S1) <| ?x1==>?S1(?x) ⊑ ?x1
```

Eine Anwendung dieser Regel auf unser Beweisziel mit anschließendem Beweis per Annahme löst dann das Beweisziel vollständig.
```
- by (etac (ub_rangeE RS spec) 1);
No subgoals
```

4.3.4.2 Beweis für das Theorem `lub_const`

Wir beginnen, indem wir uns das Hauptbeweisziel vorgeben:
```
- val prems = goal Porder.thy  "range(λx.c) <<| c";
```

Das aktuelle Beweisziel ist nun:
```
 1. range(λx. c) <<| c
```

Wir entfalten die Definition von $\ll|$ mit der Regel
```
 ?S <| ?x ∧ (∀u. ?S <| u → ?x ⊑ u)==>?S <<| ?x
- by (rtac is_lubI 1);
```

und erhalten:
```
 1. range(λx. c) <| c ∧ (∀u. range(λx. c) <| u → c ⊑ u)
```

Der nächste Schritt ist kanonisch vorgegeben und spaltet das Ziel in zwei kleinere auf:
```
- by (rtac conjI 1);

 1. range(λx. c) <| c
 2. ∀u. range(λx. c) <| u → c ⊑ u
```

Wir lösen zuerst das erste Teilziel. Da das zweite Ziel sich hierbei nicht ändert, werde ich es im nächsten Schritt nicht mehr wiederholen. Mittels der Regel
```
 ∀i. ?S(i) ⊑ ?x==>range(?S) <| ?x
- by (rtac ub_rangeI 1);
```

reduzieren wir das erste Ziel auf:
```
 1. ∀i. c ⊑ c
```

Dies läßt sich nach Beseitigung des Quantors trivial per Reflexivität lösen:
```
- by (strip_tac 1);
- by (rtac refl_less 1);
```

Das verbleibende Teilziel erhält nun die Nummer `1` und lautet:

```
 1. ∀u. range(λx. c) <| u → c ⊑ u
```

Durch Anwendung der Taktik `strip_tac` erzeugen wir uns den nötigen Annahmenkontext. Das Ziel reduziert sich zu:

```
- by (strip_tac 1);

 1. ⋀u. range(λx. c) <| u⟹c ⊑ u
```

Durch Kombination der beiden Regeln

```
ub_rangeE  range(?S) <| ?x⟹∀i. ?S(i) ⊑ ?x
spec       ∀x. ?P(x)⟹?P(?x)
```

erhalten wir aber:

```
(ub_rangeE RS spec)  range(?S1) <| ?x1⟹?S1(?x) ⊑ ?x1
```

womit sich das aktuelle Teilziel einfach lösen läßt:

```
- by (etac (ub_rangeE RS spec) 1);
No subgoals
```

4.4 Die Klasse pcpo

Dieser Abschnitt beschreibt die Theorie `Pcpo`, in der die Klasse `pcpo` (*pointed complete partial order*) der ω-kettenvollständigen Ordnungen mit kleinstem Element eingeführt wird. Der Prototyp dieser Klasse ist wieder der Typ `void`, für den in der Theorie `Porder` durch die Theoreme `minimal_void` und `cpo_void` gezeigt wurde, daß er die charakteristischen Eigenschaften der Klasse `pcpo` erfüllt. Somit handelt es sich um eine konservative Theorieerweiterung.

4.4.1 Die Theorie Pcpo

Die Theorie `Pcpo` ist in Abbildung 4.9 dargestellt. Die einzige charakteristische Konstante der Klasse `pcpo` ist das Symbol $\bot::\alpha_{\texttt{pcpo}}$. Das charakteristische Axiom `minimal` besagt, daß $\bot$ in jedem `pcpo`-Typ das kleinste Element ist. Das zweite charakteristische Axiom `cpo` der Klasse `pcpo` fordert, daß jede Kette in einem `pcpo`-Typ eine kleinste obere Schranke hat. Die explizite Typisierung $\texttt{x}::\alpha_{\texttt{pcpo}}$ ist hierbei unbedingt erforderlich, da die automatische Typinferenz sonst die Typisierung $\texttt{x}::\alpha_{\texttt{po}}$ berechnen würde, was zu allgemein wäre. Die Kettenvollständigkeit soll nur für `pcpo`-Typen gefordert werden und nicht für `po`-Typen!

Die Existenz eines kleinsten Elements muß hier explizit gefordert werden, da es keine Kette gibt, deren Bild die leere Menge ist. Somit kann aus der Kettenvollständigkeit nicht die Existenz eines kleinsten Elements gefolgert werden.

```
Pcpo = Porder +

classes pcpo < po

arities void::pcpo

consts
        ⊥:: α_pcpo

rules

minimal          ⊥ ⊑ x
cpo              is_chain(S)⟹∃x. range(S) ≪| x::α_pcpo

inst_void_pcpo   (⊥::void) = ⊥_void

end
```

Abbildung 4.9: Theorie Pcpo

Das Instanzaxiom **inst_void_pcpo** legt fest, daß die **void**-Instanz der charakteristischen Konstante ⊥ durch die Konstante **⊥_void** gegeben ist. Dieses Axiom wird, wie auch die Aritätsvereinbarung **void::pcpo**, durch die Theoreme **minimal_void** und **cpo_void** gerechtfertigt.

4.4.2 Theoreme der Theorie Pcpo

Die Theoreme der Theorie Pcpo sind in Abbildung 4.10 dargestellt.

```
thelubE              ⟦ is_chain(S); lub(range(S)) = l::α_pcpo⟧
                       ⟹range(S) ≪| l
is_ub_thelub         is_chain(S1)⟹S1(x) ⊑ lub(range(S1))
is_lub_thelub        ⟦ is_chain(S5); range(S5) <| x1⟧
                       ⟹lub(range(S5)) ⊑ x1

lub_mono             ⟦ is_chain(C1::nat ⇒ α_pcpo);is_chain(C2);
                       ∀k. C1(k) ⊑ C2(k)⟧
                       ⟹lub(range(C1)) ⊑ lub(range(C2))

lub_equal            ⟦ is_chain(C1::nat ⇒ α_pcpo);is_chain(C2);
                       ∀k.C1(k)=C2(k)⟧⟹lub(range(C1))=lub(range(C2))

lub_mono2            ⟦ ∃j.∀i. j<i → X(i::nat)=Y(i);
                       is_chain(X::nat ⇒ α_pcpo);is_chain(Y)⟧
                       ⟹lub(range(X)) ⊑ lub(range(Y))
```

```
lub_equal2          [| ∃j.∀i. j<i → X(i)=Y(i);
                       is_chain(X::nat ⇒ α_pcpo);is_chain(Y)|]
                       ⟹lub(range(X))=lub(range(Y))

lub_mono3           [| is_chain(Y::nat ⇒ α_pcpo);is_chain(X);
                       ∀i. ∃j. Y(i) ⊑ X(j)|]
                       ⟹lub(range(Y)) ⊑ lub(range(X))

eq_⊥_iff            (x=⊥)=(x ⊑ ⊥)
⊥_I                 x ⊑ ⊥⟹x = ⊥
not⊥_I              [| x ⊑ y; ¬x=⊥|]⟹¬y=⊥
not_less2not_eq     ¬x ⊑ y⟹¬x=y

chain_⊥_I           [| is_chain(Y);lub(range(Y))=⊥|]⟹∀i.Y(i)=⊥
chain_⊥_I_inverse   ∀i.Y(i::nat)=⊥⟹lub(range(Y::nat ⇒ α_pcpo))=⊥
chain_⊥_I_inverse2  ¬lub(range(Y::nat ⇒ α_pcpo))=⊥⟹∃i.¬Y(i)=⊥

chain_mono2         [| ∃j.¬Y(j)=⊥;is_chain(Y::nat ⇒ α_pcpo)|]
                       ⟹∃j.∀i.j<i → ¬Y(i)=⊥

unique_void2        x::void=⊥
```

Abbildung 4.10: Theoreme der Theorie Pcpo

Gleich das erste Theorem thelubE demonstriert, wie durch den Einsatz von Typklassen die Notation knapp und übersichtlich gehalten werden kann. Das Theorem gilt nur für pcpo-Typen, da im Beweis das charakteristische Axiom cpo der Klasse pcpo benutzt werden muß. Weil jede Kette in einem pcpo-Typ gemäß Axiom cpo eine kleinste obere Schranke hat, kann man aufgrund der Eindeutigkeit folgern, daß diese durch den Term lub(range(S)) bezeichnet wird. Diese Schlußweise wäre mit der Typisierung l::$\alpha_{\mathtt{po}}$ nicht möglich gewesen.

Die Verwendung der funktionalen Schreibweise mittels lub entspricht der gängigen Notation in der Literatur. Durch Verwendung der Theoreme thelubE und thelubI kann leicht zwischen der funktionalen und der prädikativen Schreibweise mittels $\ll|$ hin und her konvertiert werden. Die wesentliche Rechtfertigung für die zusätzliche Einführung der funktionalen Schreibweise mittels lub ist durch die Eigenschaften der Relationen = und $\sqsubseteq$ gegeben. Beide Relationen sind transitiv und besitzen Kongruenz- und Extensionalitätseigenschaften bzgl. geeigneter Abstraktions- bzw. Applikationsmechanismen. Diese Eigenschaften erlauben es, Gleichungsketten bzw. Ungleichungsketten in Beweisen zu bilden, d.h. sie erlauben den Einsatz von allgemeinen Termrewriting-Techniken. In diesem Zusammenhang ist die funktionale Schreibweise mittels lub vorteilhafter.

4.4.3 Beweise für ausgesuchte Theoreme

In diesem Abschnitt möchte ich exemplarisch für die Theorie `Pcpo` die Beweise der Theoreme `lub_mono`, `lub_mono3` und `chain_⊥_I` vorführen.

4.4.3.1 Beweis für das Theorem `lub_mono`

Wir erzeugen das initiale Beweisziel mittels

```
val prems = goal Pcpo.thy
    "⟦ is_chain(C1::nat ⇒ α_pcpo);is_chain(C2); ∀k. C1(k) ⊑ C2(k)⟧
            ⟹lub(range(C1)) ⊑ lub(range(C2))";
- by (cut_facts_tac prems 1);
```

und erhalten:

```
 1. ⟦ is_chain(C1); is_chain(C2); ∀k. C1(k) ⊑ C2(k)⟧⟹
    lub(range(C1)) ⊑ lub(range(C2))
```

Die explizite Typisierung `(C1::nat ⇒ α_pcpo)` ist hier wieder notwendig, da das Theorem nur mit dieser Beschränkung der Polymorphie gültig ist. Sie wird im folgenden zwar nicht mehr angezeigt, ist aber intern weiter vorhanden. Zuerst nützen wir aus, daß `lub(range(C1))` die kleinste obere Schranke der Kette `C1` ist. Dies geschieht durch die Regel

```
 ⟦ is_chain(?S6); range(?S6) <| ?x1⟧⟹lub(range(?S6)) ⊑ ?x1
- by (etac is_lub_thelub 1);
```

die wir mit einem anschließenden Beweis per Annahme (`etac`) kombinieren. So steuern wir gezielt die Unifikation des Systems. Als neues Beweisziel erhalten wir:

```
 1. ⟦ is_chain(C2); ∀k. C1(k) ⊑ C2(k)⟧⟹range(C1) <| lub(range(C2))
```

Um nachzuweisen, daß `lub(range(C2))` eine obere Schranke der Kette `C1` ist, genügt es zu zeigen, daß `lub(range(C2))` stärker als jedes Kettenglied ist. Hierzu verwenden wir die Regel

```
 ∀i. ?S(i) ⊑ ?x⟹range(?S) <| ?x
- by (rtac ub_rangeI 1);
```

und erhalten als neues Ziel:

```
 1. ⟦ is_chain(C2); ∀k. C1(k) ⊑ C2(k)⟧⟹∀i. C1(i) ⊑ lub(range(C2))
```

Der nächste Schritt ist kanonisch durch die Beweismethodik des natürlichen Schließens vorgegeben. Es reicht, die Aussage für ein beliebiges, aber festes `i` zu zeigen. Wir verwenden

```
 (⋀x. ?P(x))⟹∀x. ?P(x)
- by (rtac allI 1);
```

und erhalten:

```
 1. ⋀i. ⟦ is_chain(C2); ∀k. C1(k) ⊑ C2(k)⟧⟹C1(i) ⊑ lub(range(C2))
```

Nun setzen wir die restlichen Annahmen im Kontext ein. Dies wird durch die Anwendung der Transitivitätsregel `trans_less` vorbereitet.

```
 [| ?x ⊑ ?y; ?y ⊑ ?z|]==>?x ⊑ ?z
- by (rtac trans_less 1);

 1. ⋀i. [| is_chain(C2); ∀k. C1(k) ⊑ C2(k)|]==>C1(i) ⊑ ?y12(i)
 2. ⋀i. [| is_chain(C2); ∀k. C1(k) ⊑ C2(k)|]==>?y12(i) ⊑ lub(range(C2))
```

Wir erhalten zwei Teilziele, in denen der noch genauer zu bestimmende Term `?y12(i)` vorkommt, der von der Variablen `i` abhängen darf. Die Annahme `∀k. C1(k) ⊑ C2(k)` legt nahe, hierfür den Term `C2(i)` zu wählen. Wir lösen zuerst das erste Teilziel. Durch Verwendung der Eliminationsresolution `(etac)`, d.h. Resolution mit anschließendem Beweis per Annahme, im Zusammenhang mit der Regel

```
 ∀x. ?P(x)==>?P(?x)
- by (etac spec 1);
```

steuern wir die Unifikation in die gewünschte Richtung. Die Anwendung von `(etac)` führt nur dann zum Erfolg, wenn das erste neu entstehende Teilziel per Annahme lösbar ist. Hierdurch scheiden viele Unifikatoren aus, die man bei alleiniger Verwendung der einfachen Resolution `(rtac)` erhalten würde. Durch Anwendung von `(etac spec 1)` wird das erste Teilziel vollständig gelöst, und es verbleibt das Ziel:

```
 1. ⋀i. [| is_chain(C2); ∀k. C1(k) ⊑ C2(k)|]==>C2(i) ⊑ lub(range(C2))
```

Dieses läßt sich aber leicht unter Verwendung der Regel

```
 is_chain(?S1)==>?S1(?x) ⊑ lub(range(?S1))
- by (etac is_ub_thelub 1);
```

mittels Eliminationsresolution lösen.

4.4.3.2 Beweis für das Theorem `lub_mono3`

Dieses Theorem unterscheidet sich von `lub_mono` dadurch, daß die Majoranteneigenschaft, die die beiden Ketten in Beziehung setzt, schwächer ist. Wir erzeugen das initiale Beweisziel mittels

```
val prems = goal Pcpo.thy
"[| is_chain(Y::nat ⇒ α_pcpo);is_chain(X); ∀i. ∃j. Y(i) ⊑ X(j)|]
 ==>lub(range(Y)) ⊑ lub(range(X))";

- by (cut_facts_tac prems 1);
```

und erhalten:

```
 1. [| is_chain(Y); is_chain(X); ∀i. ∃j. Y(i) ⊑ X(j)|]==>
    lub(range(Y)) ⊑ lub(range(X))
```

Die kleinste obere Schranke ist kleiner als jede andere obere Schranke. Anwendung der Regel

```
 [| is_chain(?S6); range(?S6) <| ?x1|]==>lub(range(?S6)) ⊑ ?x1
- by (etac is_lub_thelub 1);
```

mittels Eliminationsresolution liefert:

```
 1. [| is_chain(Y); is_chain(X); ∀i. ∃j. Y(i) ⊑ X(j)|]==>
    range(Y) <| lub(range(X))
```

Der nächste Schritt entfaltet die Definition für obere Schranken. Mittels

```
 ∀i. ?S(i) ⊑ ?x==>range(?S) <| ?x
- by (rtac ub_rangeI 1);
```

erhalten wir:

```
 1. [| is_chain(Y); is_chain(X); ∀i. ∃j. Y(i) ⊑ X(j)|]==>
    ∀i. Y(i) ⊑ lub(range(X))
```

Der nächste Schritt ist durch die Methodik des natürlichen Schließen vorgegeben. Es reicht, die Behauptung für ein beliebiges, aber festes `i` zu zeigen.

```
- by (strip_tac 1);

 1. ⋀i. [| is_chain(Y); is_chain(X); ∀i. ∃j. Y(i) ⊑ X(j)|]==>
        Y(i) ⊑ lub(range(X))
```

Nun nützen wir die Majoranteneigenschaft im Kontext aus. Zuerst erfolgt die Spezialisierung der Prämisse mittels

```
 [| ∀x. ?P(x); ?P(?x)==>?R|]==>?R
- by (etac allE 1);
```

Die vorher universell quantifizierte Variable `i` wird hiermit durch einen noch genauer festzulegenden Term `?i12(i)` ersetzt. Die Eliminationsresolution löst das erste neue Teilziel und wir erhalten:

```
 1. ⋀i. [| is_chain(Y); is_chain(X); ∃j. Y(?i12(i)) ⊑ X(j)|]==>
        Y(i) ⊑ lub(range(X))
```

Als nächstes benutzen wir die Existenzaussage in der Prämisse mit Hilfe der Existenzeliminationsregel

```
 [| ∃x. ?P(x); ⋀x. ?P(x)==>?Q|]==>?Q
- by (etac exE 1);
```

Wir erhalten das neue Teilziel:

```
 1. ⋀i j. [| is_chain(Y); is_chain(X); Y(?i12(i)) ⊑ X(j)|]==>
          Y(i) ⊑ lub(range(X))
```

indem die Behauptung nun für ein beliebiges, aber festes `j` zu zeigen ist. Diese Eigenvariablenbedingung resultiert aus der Elimation des Existenzquantors. Die kleinste obere Schranke `lub(range(X))` ist sicher stärker als alle Glieder `X(j)` der Kette. Als Vorbereitung für diese Argumentation verwenden wir die Transitivitätsregel.

```
 [| ?x ⊑ ?y; ?y ⊑ ?z|]==>?x ⊑ ?z
- by (rtac trans_less 1);
```

Dadurch wird ein noch genauer zu bestimmender Term `?y14(i, j)` eingeführt, der von den

Variablen i und j abhängen darf. Wir erhalten zwei neue Teilziele:

```
 1. ⋀i j. ⟦ is_chain(Y); is_chain(X); Y(?i12(i)) ⊑ X(j)⟧⟹
          Y(i) ⊑ ?y14(i, j)
 2. ⋀i j. ⟦ is_chain(Y); is_chain(X); Y(?i12(i)) ⊑ X(j)⟧⟹
          ?y14(i, j) ⊑ lub(range(X))
```

Für das zweite Teilziel verwenden wir nun die Eigenschaft der kleinsten oberen Schranke

```
 is_chain(?S1)⟹?S1(?x) ⊑ lub(range(?S1))
- by (etac is_ub_thelub 2);
```

wobei wir das daraus entstehende Teilziel is_chain(X) gleich per Annahme lösen können. Es verbleibt das Teilziel:

```
 1. ⋀i j. ⟦ is_chain(Y); is_chain(X); Y(?i12(i)) ⊑ X(j)⟧⟹
          Y(i) ⊑ X(?x15(i, j))
```

mit dem noch unbestimmten Term ?x15(i, j). Da die beiden Terme Y(?i12(i)) ⊑ X(j) und Y(i) ⊑ X(?x15(i, j)) unifiziert werden können, wird dieses Ziel durch einfachen Beweis per Annahme gelöst.

```
- by (atac 1);
No subgoals
```

4.4.3.3 Beweis für das Theorem chain_⊥_I

Wir erzeugen das initiale Beweisziel mittels:

```
- val prems = goal Pcpo.thy
      "⟦ is_chain(Y);lub(range(Y))=⊥⟧⟹∀i.Y(i)=⊥";
- by (cut_facts_tac prems 1);
```

und erhalten:

```
 1. ⟦ is_chain(Y); lub(range(Y)) = ⊥⟧⟹∀i. Y(i) = ⊥
```

Es reicht, die Aussage für ein beliebiges, aber festes i zu zeigen.

```
- by (rtac allI 1);

 1. ⋀i. ⟦ is_chain(Y); lub(range(Y)) = ⊥⟧⟹Y(i) = ⊥
```

Durch Verwendung der Antisymmetrie

```
 ⟦ ?x ⊑ ?y; ?y ⊑ ?x⟧⟹?x = ?y
- by (rtac antisym_less 1);
```

erhalten wir zwei neue Teilziele,

```
 1. ⋀i. ⟦ is_chain(Y); lub(range(Y)) = ⊥⟧⟹Y(i) ⊑ ⊥
 2. ⋀i. ⟦ is_chain(Y); lub(range(Y)) = ⊥⟧⟹⊥ ⊑ Y(i)
```

von denen das zweite trivial per

```
 ⊥ ⊑ ?x
- by (rtac minimal 2);
```

gelöst werden kann. Es verbleibt:

```
 1. ⋀i. ⟦ is_chain(Y); lub(range(Y)) = ⊥⟧⟹Y(i) ⊑ ⊥
```

Nun ersetzen wir die Konstante $\bot$ durch den Term `lub(range(Y))`, was aufgrund der Annahme möglich ist.

```
- by (res_inst_tac [("t","⊥")] subst 1);
- by (atac 1);
```

Als neues Teilziel erhalten wir

```
 1. ⋀i. ⟦ is_chain(Y); lub(range(Y)) = ⊥⟧⟹Y(i) ⊑ lub(range(Y))
```

welches aber trivial mittels

```
 is_chain(?S1)⟹?S1(?x) ⊑ lub(range(?S1))
- by (etac is_ub_thelub 1);
```

gelöst werden kann.

4.5 Theorien für den vollen Funktionenraum

Durch die Theorien `Porder0` und `Pcpo` wurde die Logik höherer Stufe HOLC um die Klassen `po` und `pcpo` erweitert. Außer dem Prototypen `void` ist bisher kein Typ bekannt, der in einer der beiden Klassen liegt. Im folgenden werden für verschiedene Typen Ordnungen definiert und entsprechende Eigenschaften dieser Ordnungen abgeleitet, die es erlauben, die Typen mit den zugehörigen Instantiierungen den Klassen zuzuordnen. Aufgrund der konservativen Erweiterungsmethodik geschieht dies für jeden Typkonstruktor in mehreren Schritten.

Dieser Abschnitt befaßt sich mit dem Typkonstruktor $\Rightarrow$ für den vollen Funktionenraum.

4.5.1 Die Theorie `Fun1`

In der Theorie `Fun1` wird eine Ordnung `less_fun` für den vollen Funktionenraum über partiell geordneten Typen (Typen in Klasse `po`) definiert. Die Theorie ist in Abbildung 4.11 dargestellt. Es handelt sich hierbei um die übliche punktweise Ordnung.

Um die Ordnung definieren zu können, muß nur der Bildbereich partiell geordnet sein. Man beachte in diesem Zusammenhang den Typ der Konstanten `less_fun`. Die Typvariable $\beta_{\texttt{po}}$ wird explizit getypt, während die Typvariable α durch den `default`-Mechanismus die Typisierung $\alpha_{\texttt{term}}$ erhält. Die `default` Klasse ist nach wie vor die Klasse `term`.

```
Fun1 = Pcpo +
consts  less_fun :: [α ⇒ β_po, α ⇒ β] ⇒ bool

rules

less_fun_def  less_fun(f1,f2) ≡ ∀x. f1(x) ⊑ f2(x)

end
```

Abbildung 4.11: Theorie `Fun1`

In der Definition `less_fun_def` kann aufgrund der Typisierung $\beta_{\mathtt{po}}$ das polymorphe Ordnungssymbol $\sqsubseteq$ für die Ordnung auf dem Bildbereich verwendet werden. Hier zeigen sich wieder die notationellen Vorteile der Logik mit Typklassen.

4.5.2 Theoreme der Theorie `Fun1`

Die Funktion `less_fun` hat die Eigenschaften einer partiellen Ordnungsrelation. Die diesbezüglichen Theoreme, deren Beweise offensichtlich sind, sind in Abbildung 4.12 dargestellt.

```
refl_less_fun     less_fun(f,f)
antisym_less_fun  [| less_fun(f1,f2); less_fun(f2,f1)|]==>f1 = f2
trans_less_fun    [| less_fun(f1,f2); less_fun(f2,f3)|]==>less_fun(f1,f3)
```

Abbildung 4.12: Theoreme der Theorie `Fun1`

4.5.3 Die Theorie `Fun2`

Die Theoreme `refl_less_fun`, `antisym_less_fun` und `trans_less_fun` rechtfertigen die Aritätsvereinbarung `fun::(term,po)po`. Für den Infix-Typkonstruktor $\Rightarrow$ ist bekanntlich auch die Schreibweise `fun` erlaubt. Die Theorie `Fun2` ist in Abbildung 4.13 dargestellt.

```
Fun2 = Fun1 +
arities fun :: (term,po)po

consts  ⊥_fun  ::  α_term ⇒ β_pcpo

rules
inst_fun_po      ( ⊑ ::[α ⇒ β_po, α ⇒ β_po ] ⇒ bool) = less_fun

⊥_fun_def        ⊥_fun ≡ (λx.⊥)
end
```

Abbildung 4.13: Theorie `Fun2`

Neben der konservativen Erweiterung durch die Aritätsvereinbarung `fun::(term,po)po` wird auch gleich eine Konstante für die bzgl. der Ordnung `less_fun` kleinste Funktion im Funktionenraum eingeführt. Daß es sich bei `⊥_fun` tatsächlich um die kleinste Funktion handelt, muß natürlich erst noch bewiesen werden.

4.5.4 Theoreme der Theorie `Fun2`

Die Theoreme der Theorie `Fun2` sind in Abbildung 4.14 dargestellt. Die wichtigsten Theoreme sind `minimal_fun` und `cpo_fun`, die zeigen, daß es eine kleinste Funktion gibt, und daß die Ordnung `less_fun` kettenvollständig ist. Das Theorem `less_fun` erlaubt eine schönere Schreibweise für die Ordnung auf den Funktionen. Die Theoreme `ch2ch_fun` und `ub2ub_fun` zeigen, wie sich Ketten und obere Schranken von Funktionen fortpflanzen. Das zentrale Hilfstheorem für den Beweis von `cpo_fun` ist das Theorem `lub_fun`, dessen Beweis im Abschnitt 4.5.5 vorgeführt wird.

`minimal_fun`	`⊥_fun ⊑ f`
`less_fun`	`(f1 ⊑ f2) = (∀x. f1(x) ⊑ f2(x))`
`ch2ch_fun`	`is_chain(S::nat ⇒ (`$\alpha_{\texttt{term}}$` ⇒ `$\beta_{\texttt{po}}$`))` `⟹is_chain(λi.S(i)(x))`
`ub2ub_fun`	`range(S::nat ⇒ (`$\alpha_{\texttt{term}}$` ⇒ `$\beta_{\texttt{po}}$`)) <| u` `⟹range(λi. S(i,x)) <| u(x)`
`lub_fun`	`is_chain(S::nat ⇒ (`$\alpha_{\texttt{term}}$` ⇒ `$\beta_{\texttt{pcpo}}$`))` `⟹range(S) <<| (λx.lub(range(λi.S(i)(x))))`
`thelub_fun`	`is_chain(S1)` `⟹lub(range(S1)) = (λx. lub(range(λi. S1(i,x))))`
`cpo_fun`	`is_chain(S::nat ⇒ (`$\alpha_{\texttt{term}}$` ⇒ `$\beta_{\texttt{pcpo}}$`))` `⟹∃x. range(S) <<| x`

Abbildung 4.14: Theoreme der Theorie `Fun2`

4.5.5 Beweise für ausgesuchte Theoreme

In diesem Abschnitt wird nur der Beweis für das Theorem `lub_fun` dargestellt. Die Beweisidee ist [All86] entnommen. Es ist wichtig, daß das Theorem `lub_fun` vor dem Theorem `thelub_fun` bewiesen wird, da dieses nur gezeigt werden kann, wenn die Existenz einer kleinsten oberen Schranke bereits gesichert ist. Das Theorem `thelub_fun` ist eine notationelle Variante des Theorems `lub_fun`, die besser für Gleichungsbeweise geeignet ist. Aus dem Theorem `lub_fun` folgt dann trivial das Theorem `cpo_fun`, da ein konkreter Zeuge für die Existenzaussage zur Verfügung steht.

4.5.5.1 Beweis für das Theorem lub_fun

Wir erzeugen das initiale Beweisziel mittels

```
val prems = goal  Fun2.thy
        "is_chain(S::nat ⇒ (α_term ⇒ β_pcpo))⟹
         range(S) ≪| (λx.lub(range(λi.S(i)(x))))";

- by (cut_facts_tac prems 1);
```

und erhalten:

```
 1. is_chain(S)⟹range(S) ≪| (λx. lub(range(λi. S(i, x))))
```

Die explizite Typisierung beim Erzeugen des Beweisziels ist wichtig, um den Grad der Polymorphie geeignet zu beschränken. Wir entfalten die Definition für kleinste obere Schranken mit der Regel

```
 ?S <| ?x ∧ (∀u. ?S <| u → ?x ⊑ u)⟹?S ≪| ?x
- by (rtac is_lubI 1);

 1. is_chain(S)⟹
    range(S) <| (λx. lub(range(λi. S(i, x)))) ∧
    (∀u. range(S) <| u → (λx. lub(range(λi. S(i, x)))) ⊑ u)
```

Als nächstes brechen wir das Ziel in zwei kleinere Beweisziele auf. Wir verwenden

```
 ⟦ ?P; ?Q⟧⟹?P ∧ ?Q
- by (rtac conjI 1);
```

und erhalten die beiden Teilziele:

```
 1. is_chain(S)⟹range(S) <| (λx. lub(range(λi. S(i, x))))
 2. is_chain(S)⟹∀u. range(S) <| u → (λx. lub(range(λi. S(i, x)))) ⊑ u
```

Zunächst lösen wir das erste Teilziel. Das zweite Teilziel wird im folgenden erst wieder dargestellt, wenn das erste Teilziel vollständig abgearbeitet ist. Wie entfalten die Definition für obere Schranken mittels

```
 ∀i. ?S(i) ⊑ ?x⟹range(?S) <| ?x
- by (rtac ub_rangeI 1);

 1. is_chain(S)⟹∀i. S(i) ⊑ (λx. lub(range(λi. S(i, x))))
```

Es genügt, die Behauptung für ein beliebiges, aber festes i zu zeigen. Wie verwenden

```
 (⋀x. ?P(x))⟹∀x. ?P(x)
- by (rtac allI 1);

 1. ⋀i. is_chain(S)⟹S(i) ⊑ (λx. lub(range(λi. S(i, x))))
```

Nun setzen wir die Ordnung für Funktionen geeignet ein. Zwei Funktionen befinden sich in Ordnungsrelation, wenn ihre Anwendungen punktweise in Relation stehen. Eine Kombination der Regeln

```
less_fun     ?f1.0 ⊑ ?f2.0 = (∀x. ?f1.0(x) ⊑ ?f2.0(x))
ssubst       ⟦ ?t = ?s; ?P(?s)⟧⟹?P(?t)
```

durch Resolution (RS) ergibt die Regel

```
 ?P(∀x. ?f1.1(x) ⊑ ?f2.1(x))⟹?P(?f1.1 ⊑ ?f2.1)
- by (rtac (less_fun RS ssubst) 1);
```

Die Anwendung dieser kombinierten Regel führt zum neuen Beweisziel:

```
 1. ⋀i. is_chain(S)⟹∀x. S(i, x) ⊑ lub(range(λi. S(i, x)))
```

Der nächste Schritt ist kanonisch vorgegeben. Es reicht, die Behauptung für ein beliebiges, aber festes x zu zeigen.

```
- by (rtac allI 1);

 1. ⋀i x. is_chain(S)⟹S(i, x) ⊑ lub(range(λi. S(i, x)))
```

Wir benutzen die Tatsache, daß kleinste obere Schranken insbesondere auch obere Schranken sind. Durch Anwendung der Regel

```
 is_chain(?S1)⟹?S1(?x) ⊑ lub(range(?S1))
- by (rtac is_ub_thelub 1);
```

reduziert sich unser Beweisziel zu:

```
 1. ⋀i x. is_chain(S)⟹is_chain(λi. S(i, x))
```

Dieses lösen wir vollständig durch Eliminationsresolution mit dem Theorem

```
 is_chain(?S)⟹is_chain(λi. ?S(i, ?x))
- by (etac ch2ch_fun 1);
```

Nun können wir uns dem zweiten Teilziel zuwenden, das wir vorher zurückgestellt haben. Es lautet:

```
 1. is_chain(S)⟹∀u. range(S) <| u → (λx. lub(range(λi. S(i, x)))) ⊑ u
```

Es reicht, die Aussage für ein beliebiges, aber festes u zu zeigen. Außerdem nehmen wir die Prämisse der zu zeigenden Implikationsformel in den Annahmenkontext auf. Dies geschieht durch Verwendung der Taktik

```
- by (strip_tac 1);
```

Wir erhalten das neue Ziel:

```
 1. ⋀u. ⟦ is_chain(S); range(S) <| u⟧⟹(λx. lub(range(λi. S(i, x)))) ⊑ u
```

Wie vorhin verwenden wir die Ordnung auf dem Funktionenraum mittels der kombinierten Regel

```
 ?P(∀x. ?f1.1(x) ⊑ ?f2.1(x))⟹?P(?f1.1 ⊑ ?f2.1)
- by (rtac (less_fun RS ssubst) 1);
```

Dies ergibt als neues Teilziel:

```
 1. ⋀u. [| is_chain(S); range(S) <| u|]==>
        ∀x. lub(range(λi. S(i, x))) ⊑ u(x)
```

Es genügt, die Aussage für ein beliebiges, aber festes `x` zu zeigen. Anwendung der Regel
`(⋀x. ?P(x))==>∀x. ?P(x)`

```
- by (rtac allI 1);
```

liefert:

```
 1. ⋀u x. [| is_chain(S); range(S) <| u|]==>lub(range(λi. S(i, x))) ⊑ u(x)
```

Im Unterschied zum vorhergehenden Fall nützen wir diesmal aus, daß
`lub(range(λi. S(i, x)))` die kleinste obere Schranke ist.

```
 [| is_chain(?S6); range(?S6) <| ?x1|]==>lub(range(?S6)) ⊑ ?x1
- by (rtac is_lub_thelub 1);
```

Wir erhalten zwei neue Teilziele:

```
 1. ⋀u x. [| is_chain(S); range(S) <| u|]==>is_chain(λi. S(i, x))
 2. ⋀u x. [| is_chain(S); range(S) <| u|]==>range(λi. S(i, x)) <| u(x)
```

Beide Ziele lassen sich trivial durch Anwendung der Theoreme

```
ch2ch_fun      is_chain(?S)==>is_chain(λi. ?S(i, ?x))
ub2ub_fun      range(?S) <| ?u==>range(λi. ?S(i, ?x)) <| ?u(?x)
```

lösen.

```
- by (etac ch2ch_fun 1);
- by (etac ub2ub_fun 1);
No subgoals
```

4.5.6 Die Theorie `Fun3`

Die beiden Theoreme `minimal_fun` und `cpo_fun` der Theorie `Fun2` erlauben die konservative Erweiterung durch die Aritätsvereinbarung `fun::(term,pcpo)pcpo`. Diese wird in der Theorie `Fun3`, die in Abbildung 4.15 dargestellt ist, vorgenommen. Sie besagt, daß der Raum der vollen Funktionen eine `pcpo`-Struktur hat, wenn der Bildbereich eine `pcpo`-Struktur aufweist. Für die Theorie `Fun3` wurden keine Theoreme bewiesen.

```
Fun3 = Fun2 +
arities fun  :: (term,pcpo)pcpo

rules

inst_fun_pcpo     ⊥::α ⇒ β_pcpo = ⊥_fun

end
```

Abbildung 4.15: Theorie `Fun3`

4.6 Theorie der stetigen Funktionen Cont

In diesem Abschnitt werden die Mengen der monotonen und stetigen Funktionen durch Prädikate über dem vollen Funktionenraum ausgezeichnet.

4.6.1 Die Theorie Cont

Die Theorie Cont führt die Begriffe der monotonen und stetigen Funktionen ein. Die Theorie ist in Abbildung 4.16 abgebildet. Um in den folgenden Theorien unnötige Schreibarbeit zu sparen, wird als default-Klasse die Klasse pcpo vereinbart. Wenn also eine Typvariable ohne explizite Angabe der Klasse verwendet wird, dann ist automatisch die Klasse pcpo gemeint. Diese default-Einstellung bleibt für den Rest der Entwicklung von HOLCF bestehen.

```
Cont = Fun3 +

default pcpo

consts
        monofun :: (α_po ⇒ β_po) ⇒ bool
        contlub :: (α ⇒ β) ⇒ bool
        contX   :: (α ⇒ β) ⇒ bool

rules

monofun          monofun(f) ≡ ∀x y. x ⊑ y → f(x) ⊑ f(y)

contlub          contlub(f) ≡ ∀Y. is_chain(Y) →
                                  f(lub(range(Y))) = lub(range(λi.f(Y(i))))

contX            contX(f) ≡ ∀Y. is_chain(Y) →
                                  range(λi.f(Y(i))) ≪| f(lub(range(Y)))

end
```

Abbildung 4.16: Theorie Cont

Für Funktionen auf Typen der Klasse po wird das Prädikat monofun für monotone Funktionen definiert. Für Funktionen auf Typen der Klasse pcpo werden die beiden Prädikate contlub und contX eingeführt. Das Prädikat contlub formuliert, unter Verwendung der funktionalen Schreibweise mittels lub, die bekannte Vertauschung der Funktionsanwendung und der Bildung der kleinsten oberen Schranke. Auch hier führt die Verwendung von Typklassen zu einer eleganteren Notation, denn ohne Typklassen müssten alle Definitionen mit Prämissen versehen werden, die die Ordnungsrelation auf den polymorphen Typen beschreiben. Dadurch würden aber die Lesbarkeit und die Benutzbarkeit der Formalisierung leiden.

Eine Funktion f wird üblicherweise als (ketten-)stetig bezeichnet, wenn sie sowohl das Prädikat monofun, als auch das Prädikat contlub erfüllt. Interessanterweise führt die alleinige Verwendung des Prädikats contX zu einer äquivalenten Formulierung der Stetigkeit. Dieser Ansatz wird zum Beispiel in [Sto77] verfolgt. In der Definition von contX wird anstatt der funktionalen Schreibweise mittels lub das Prädikat $\ll|$ verwendet. Hierdurch wird bereits gefordert, daß f(lub(range(Y))) die kleinste obere Schranke von range(λi.f(Y(i))) ist[6], während in der Definition von contlub die Vertauschbarkeit zwar gegeben ist, aber nicht garantiert wird, daß f(lub(range(Y))) wirklich die kleinste obere Schranke von range(λi.f(Y(i))) ist. Um diese Eigenschaft nachzuweisen, muß zusätzlich die Monotonie der Funktion f gefordert werden.

An diesem Beispiel wird der subtile Unterschied zwischen der funktionalen Schreibweise mittels lub und der prädikativen Schreibweise mittels $\ll|$ deutlich. Der Unterschied resultiert im Endeffekt aus der folgenden Eigenschaft des Kennzeichnungsoperators

select_equality $\quad [\![P(a); \bigwedge x.\ P(x) \Longrightarrow x=a]\!] \Longrightarrow (\varepsilon x.\ P(x)) = a$

deren Umkehrung leider falsch ist. Aus (εx. P(x))= a kann nicht P(a) gefolgert werden.

An dieser Stelle möchte ich noch einmal auf die Definition des Kettenbegriffs aus Theorie Porder zurückkommen. Würde man Ketten so definieren, daß auch die leere Menge eine Kette ist, dann müßte bei der Definition des Stetigkeitsbegriffs die leere Kette ausgenommen werden. Andernfalls könnte aus der Stetigkeit einer Funktion sofort auf deren Striktheit geschlossen werden, da die kleinste obere Schranke der leeren Menge das jeweils kleinste Element im Bereich ist[7]. Bei der hier verwendeten Definition für Ketten mittels der abzählenden Funktion kann dies aber nicht vorkommen, da das Bild einer totalen Funktion von den natürlichen Zahlen in den jeweiligen Bereich stets nicht leer ist!

4.6.2 Theoreme der Theorie Cont

Die Monotonie und Stetigkeit von Funktionen sind von zentraler Bedeutung für die Entwicklung von HOLCF. Aus diesem Grund wurden in der Theorie Cont eine Vielzahl von Theoremen abgeleitet. Zur besseren Übersicht habe ich die Theoreme in den Abbildungen 4.17 bis 4.20 getrennt dargestellt.

Die meisten Theoreme haben den Charakter von Hilfstheoremen, die die Beweise in späteren Theorien erleichtern. Nichtsdestoweniger waren gerade die Beweise für diese Hilfstheoreme bisweilen schwieriger zu führen, als die Beweise für die eigentlichen Theoreme. Aus diesem Grund habe ich sie auch alle aufgelistet.

[6] das X in contX steht für die zusätzliche Existenzaussage.

[7] der Hinweis auf diese Tatsache stammt von Bernhard Möller.

```
contlubI  ∀Y. is_chain(Y) → f(lub(range(Y))) = lub(range(λi. f(Y(i))))
           ⟹contlub(f)

contlubE  contlub(f)⟹
           ∀Y. is_chain(Y) → f(lub(range(Y))) = lub(range(λi. f(Y(i))))

contXI    ∀Y. is_chain(Y) → range(λi.f(Y(i))) ≪| f(lub(range(Y)))
           ⟹contX(f)

contXE    contX(f)⟹
           ∀Y. is_chain(Y) → range(λi.f(Y(i))) ≪| f(lub(range(Y)))

monofunI  ∀x y. x ⊑ y → f(x) ⊑ f(y)⟹monofun(f)
monofunE  monofun(f)⟹∀x y. x ⊑ y → f(x) ⊑ f(y)
```

Abbildung 4.17: Theoreme der Theorie `Cont` - Teil 1

In Abbildung 4.17 sind Theoreme von eher trivialer Natur dargestellt. Sie erlauben lediglich den besseren Zugang zu den Definitionen der Theorie `Cont` und bedürfen keiner näheren Erläuterung.

```
ch2ch_monofun      [| monofun(f); is_chain(Y)|]⟹is_chain(λi. f(Y(i)))
ub2ub_monofun      [| monofun(f); range(Y) <| u|]
                     ⟹range(λi.f(Y(i))) <| f(u)

monocontlub2contX  [| monofun(f);contlub(f)|]⟹contX(f)
binchain_contX     [| contX(f); x ⊑ y|]
                     ⟹range(λi. f(if(i = 0,x,y))) ≪| f(y)
contX2mono         contX(f)⟹monofun(f)
contX2contlub      contX(f)⟹contlub(f)

ch2ch_MF2L     [| monofun(MF2::(α_po ⇒ β_po ⇒ γ_po)); is_chain(F)|]
                 ⟹is_chain(λi. MF2(F(i),x))

ch2ch_MF2R     [| monofun(MF2(f)::(β_po ⇒ γ_po)); is_chain(Y)|]
                 ⟹is_chain(λi. MF2(f,Y(i)))

ch2ch_MF2LR    [| monofun(MF2::(α_po ⇒ β_po ⇒ γ_pcpo));
                  ∀f.monofun(MF2(f)::(β_po ⇒ γ_pcpo));
                  is_chain(F); is_chain(Y)|]
                 ⟹is_chain(λi. MF2(F(i))(Y(i)))

ch2ch_lubMF2R  [| monofun(MF2::(α_po ⇒ β_po ⇒ γ_pcpo));
                  ∀f.monofun(MF2(f)::(β_po ⇒ γ_pcpo));
                  is_chain(F);is_chain(Y)|]
```

```
                ⟹is_chain(λj. lub(range(λi. MF2(F(j),Y(i)))))

ch2ch_lubMF2L   ⟦ monofun(MF2::(α_po ⇒ β_po ⇒ γ_pcpo));
                  ∀f.monofun(MF2(f)::(β_po ⇒ γ_pcpo));
                  is_chain(F);is_chain(Y)⟧
                  ⟹is_chain(λi. lub(range(λj. MF2(F(j),Y(i)))))
```

Abbildung 4.18: Theoreme der Theorie Cont - Teil 2

In Abbildung 4.18 sind vorwiegend Theoreme dargestellt, die die Fortpflanzung von Ketten beschreiben, wenn monotone bzw. stetige Funktionen angewendet werden. Sie werden in Beweisen für Eigenschaften monotoner und stetiger Funktionen ständig benötigt und kombiniert, da der Nachweis von Ketteneigenschaften einen wesentlichen Anteil an der Beweisarbeit ausmacht. In der Abbildung sind aber auch die Theoreme monocontlub2contX, contX2mono und contX2contlub aufgelistet, die den Zusammenhang zwischen den Prädikaten monofun, contlub und contX zeigen.

```
lub_MF2_mono    ⟦ monofun(MF2::(α_po ⇒ β_po ⇒ γ_pcpo));
                  ∀f.monofun(MF2(f)::(β_po ⇒ γ_pcpo));
                  is_chain(F)⟧
                  ⟹monofun(λx.lub(range(λj.MF2(F(j),x))))

ex_lubMF2       ⟦ monofun(MF2::(α_po ⇒ β_po ⇒ γ_pcpo));
                  ∀f.monofun(MF2(f)::(β_po ⇒ γ_pcpo));
                  is_chain(F); is_chain(Y)⟧
                  ⟹
                  lub(range(λj. lub(range(λi. MF2(F(j),Y(i)))))) =
                  lub(range(λi. lub(range(λj. MF2(F(j),Y(i))))))

diag_lemma1     ⟦ contX(CF2);∀f.contX(CF2(f));is_chain(FY);is_chain(TY)⟧
                  ⟹is_chain(λi. lub(range(λj. CF2(FY(j), TY(i)))))

diag_lemma2     ⟦ contX(CF2);is_chain(FY); is_chain(TY)⟧
                  ⟹is_chain(λm. CF2(FY(m), TY(n::nat)))

diag_lemma3     ⟦ ∀f.contX(CF2(f));is_chain(FY); is_chain(TY)⟧
                  ⟹is_chain(λm. CF2(FY(n), TY(m)))

diag_lemma4     ⟦ contX(CF2);∀f.contX(CF2(f));is_chain(FY); is_chain(TY)⟧
                  ⟹is_chain(λm. CF2(FY(m), TY(m)))

diag_lubCF2_1   ⟦ contX(CF2);∀f.contX(CF2(f));is_chain(FY);is_chain(TY)⟧
                  ⟹
                  lub(range(λi. lub(range(λj. CF2(FY(j))(TY(i)))))) =
                  lub(range(λi. CF2(FY(i))(TY(i))))
```

```
diag_lubCF2_2  [| contX(CF2);∀f.contX(CF2(f));is_chain(FY);is_chain(TY)|]
                 ==>
                 lub(range(λj. lub(range(λi. CF2(FY(j))(TY(i))))))) =
                 lub(range(λi. CF2(FY(i))(TY(i))))

contlub_CF2    [| contX(CF2);∀f.contX(CF2(f));is_chain(FY);is_chain(TY)|]
                 ==>
                 CF2(lub(range(FY)))(lub(range(TY))) =
                 lub(range(λi.CF2(FY(i))(TY(i))))
```

Abbildung 4.19: Theoreme der Theorie `Cont` - Teil 3

Die Abbildung 4.19 zeigt Theoreme über eine 'zweistellige' Funktion[8], die monoton bzw. stetig in 'beiden' Argumenten ist. Die Theoreme zeigen die Wirkung der Funktion auf kleinste obere Schranken. Besonders erwähnen möchte ich die Theoreme `ex_lubMF2`, `diag_lubCF2_1`, `diag_lubCF2_2` und `contlub_CF2`.

```
monofun_fun_fun  f1 ⊑ f2==>f1(x) ⊑ f2(x)
monofun_fun_arg  [| monofun(f); x1 ⊑ x2|]==>f(x1) ⊑ f(x2)
monofun_fun      [| monofun(f1); monofun(f2); f1 ⊑ f2; x1 ⊑ x2|]
                   ==>f1(x1) ⊑ f2(x2)

mono2mono_MF1L         [| monofun(c1)|]==>monofun(λx. c1(x,y))
contX2contX_CF1L       [| contX(c1)|]==>contX(λx. c1(x,y))
mono2mono_MF1L_rev     ∀y.monofun(λx.c1(x,y))==>monofun(c1)
contX2contX_CF1L_rev   ∀y.contX(λx.c1(x,y))==>contX(c1)

mono2mono_app          [| monofun(ft);∀x.monofun(ft(x));monofun(tt)|]
                         ==>monofun(λx.(ft(x))(tt(x)))

contX2contlub_app      [| contX(ft);∀x.contX(ft(x));contX(tt)|]
                         ==>contlub(λx.(ft(x))(tt(x)))

contX2contX_app        [| contX(ft);∀x.contX(ft(x));contX(tt)|]
                         ==>contX(λx.(ft(x))(tt(x)))

contX_id               contX(λx.x)
contX_const            contX(λx.c)
```

Abbildung 4.20: Theoreme der Theorie `Cont` - Teil 4

Die Abbildung 4.20 zeigt neben den 'Kongruenz'-Eigenschaften von monotonen Funktionen

[8] die Funktion wird hier in Curry-Schreibweise verwendet.

insbesondere die Fortpflanzung der Monotonie und Stetigkeit unter λ-Abstraktion und Applikation. Diese Theoreme finden in der Theorie der Operationen in Abschnitt 4.7 mehrfache Anwendung. Die Theoreme `contX_id` und `contX_const` zeigen die Stetigkeit der Identitätsfunktion und von konstanten Funktionen.

4.6.3 Beweise für ausgesuchte Theoreme

In diesem Abschnitt präsentiere ich die Beweise für die Theoreme `diag_lubCF2_1`, `contlub_CF2` und `contX2contlub_app`. Diese Theoreme wurden ausgewählt, weil die Argumentation in ihren Beweisen typisch für die Theorie `Cont` der stetigen Funktionen ist. Gleichzeitig möchte ich durch die Beweise auch dokumentieren, daß die maschinenunterstützte Beweisführung mit dem Isabelle-System im Gedankengang mit der Beweisführung mit Papier und Bleistift übereinstimmt.

4.6.3.1 Beweis für das Theorem `diag_lubCF2_1`

Der Beweis dieses Theorems demonstriert den Umgang mit kleinsten oberen Schranken und stetigen Funktionen. Das Theorem zeigt, daß die Bildung der kleinsten oberen Schranken bezüglich zweier Laufvariablen identisch ist mit der diagonalisierten Bildung einer oberen Schranke.

Wir beginnen, indem wir uns das gewünschte Beweisziel vorgeben:

```
val prems = goal Cont.thy
  "[| contX(CF2);∀f.contX(CF2(f));is_chain(FY);is_chain(TY)|]==>
              lub(range(λi. lub(range(λj. CF2(FY(j))(TY(i))))))) =
              lub(range(λi. CF2(FY(i))(TY(i))))";

- by (cut_facts_tac prems 1);
```

Unser Ausgangspunkt ist nun:

```
 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    lub(range(λi. lub(range(λj. CF2(FY(j), TY(i))))))) =
    lub(range(λi. CF2(FY(i), TY(i))))
```

Diese Gleichung zeigen wir, indem wir die Antisymmetrie der Gleichheit ausnützen und die 'Inklusion' in beiden Richtungen zeigen.

```
- by (rtac antisym_less 1);

 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    lub(range(λi. lub(range(λj. CF2(FY(j), TY(i))))))) ⊑
    lub(range(λi. CF2(FY(i), TY(i))))
 2. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    lub(range(λi. CF2(FY(i), TY(i)))) ⊑
    lub(range(λi. lub(range(λj. CF2(FY(j), TY(i)))))))
```

Wir lösen zuerst das erste Teilziel. Das zweite Teilziel werde ich bis zur vollständigen Abarbeitung des ersten Teilziels nicht mehr auflisten. Wir verwenden die Tatsache, daß die linke kleinste obere Schranke kleiner ist als jede obere Schranke.

```
 [| is_chain(?S6); range(?S6) <| ?x1|]==>lub(range(?S6)) << ?x1
- by (rtac is_lub_thelub 1);

 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    is_chain(λi. lub(range(λj. CF2(FY(j), TY(i)))))
 2. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    range(λi. lub(range(λj. CF2(FY(j), TY(i))))) <|
    lub(range(λi. CF2(FY(i), TY(i))))
```

Wir erhalten zwei neue Teilziele, wobei wir im wesentlichen am zweiten interessiert sind. Das erste Teilziel besteht im Nachweis einer Ketteneigenschaft. Diese weisen wir nach, indem wir den Annahmenkontext verwenden und die darin vorgegebenen Ketteneigenschaften geeignet propagieren. Der entsprechende Beweis wurde schon vorweggenommen und ist im Hilfstheorem

```
diag_lemma1
   [| contX(?CF2.0); ∀f. contX(?CF2.0(f)); is_chain(?FY); is_chain(?TY)|]
     ==>is_chain(λi. lub(range(λj. ?CF2.0(?FY(j), ?TY(i)))))
```

kondensiert. Die Anwendung des Hilfstheorems mit anschließendem mehrfachen Beweis per Annahme löst das erste Teilziel vollständig.

```
- by (etac diag_lemma1 1);
- by (REPEAT (atac 1));
```

Im verbleibenden Teilziel:

```
 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    range(λi. lub(range(λj. CF2(FY(j), TY(i))))) <|
    lub(range(λi. CF2(FY(i), TY(i))))
```

entfalten wir die Definition von oberen Schranken mittels

```
 ∀i. ?S(i) << ?x==>range(?S) <| ?x
- by (rtac ub_rangeI 1);
```

Wir erhalten:

```
 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    ∀i. lub(range(λj. CF2(FY(j), TY(i)))) << lub(range(λi. CF2(FY(i), TY(i))))
```

Es reicht, die Behauptung für ein beliebiges, aber festes `i` zu zeigen.

```
- by (strip_tac 1 );

 1. ⋀i. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
        lub(range(λj. CF2(FY(j), TY(i)))) << lub(range(λi. CF2(FY(i), TY(i))))
```

Die wesentliche Beweisidee besteht nun darin, zu zeigen, daß die rechte Kette eine Majorante der linken Kette ist. Hierfür haben wir bereits ein passendes Theorem abgeleitet.

```
   [| is_chain(?Y); is_chain(?X); ∀i. ∃j. ?Y(i) ⊑ ?X(j)|]==>
   lub(range(?Y)) ⊑ lub(range(?X))
- by (rtac lub_mono3 1);

 1. ⋀i. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
        is_chain(λj. CF2(FY(j), TY(i)))
 2. ⋀i. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
        is_chain(λi. CF2(FY(i), TY(i)))
 3. ⋀i. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
        ∀ia. ∃j. CF2(FY(ia), TY(i)) ⊑ CF2(FY(j), TY(j))
```

Die ersten beiden Teilziele bestehen im Nachweis von Ketteneigenschaften. Sie werden über die vorher eigens bewiesenen Hilfstheoreme

```
diag_lemma2
   [| contX(?CF2.0); is_chain(?FY); is_chain(?TY)|]==>
   is_chain(λm. ?CF2.0(?FY(m), ?TY(?n)))

diag_lemma4
   [| contX(?CF2.0); ∀f. contX(?CF2.0(f)); is_chain(?FY); is_chain(?TY)|]
   ==>is_chain(λm. ?CF2.0(?FY(m), ?TY(m)))
```

gelöst. Im dritten Teilziel reicht es, die Behauptung für ein beliebiges, aber festes `ia` zu zeigen.

```
- by (etac diag_lemma2 1);
- by (REPEAT (atac 1));
- by (etac diag_lemma4 1);
- by (REPEAT (atac 1));
- by (rtac allI 1);
```

Wir erhalten:

```
 1. ⋀i ia.
       [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
       ∃j. CF2(FY(ia), TY(i)) ⊑ CF2(FY(j), TY(j))
```

Wir müssen nun zeigen, daß es für beliebige Indizes `i` und `ia` stets einen Index `j` gibt, sodaß die linke Seite schwächer ist, als die rechte Seite. Wir machen eine Fallunterscheidung über die Ordnungsbeziehung der Indizes `i` und `ia`. Der gesuchte Index `j` wird dann jeweils das Maximum der beiden Indizes `i` und `ia` sein. In dieser Wahl steckt die Diagonalisierungsidee des ganzen Beweises. Zur Fallunterscheidung verwenden wir die Regel

```
   [| ?m < ?n==>?P(?n, ?m);
      ?m = ?n==>?P(?n, ?m);
      ?n < ?m==>?P(?n, ?m)|]==>?P(?n, ?m)

- by (res_inst_tac [("m","i"),("n","ia")] nat_less_cases 1);
```

Dies beschert uns die drei Teilziele:

```
1. ⋀i ia.
      ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
         i < ia⟧⟹
      ∃j. CF2(FY(ia), TY(i)) ⊑ CF2(FY(j), TY(j))
2. ⋀i ia.
      ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
         i = ia⟧⟹
      ∃j. CF2(FY(ia), TY(i)) ⊑ CF2(FY(j), TY(j))
3. ⋀i ia.
      ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
         ia < i⟧⟹
      ∃j. CF2(FY(ia), TY(i)) ⊑ CF2(FY(j), TY(j))
```

Konzentrieren wir uns auf den ersten Fall `i < ia`. Die Auflistung der beiden anderen Fälle werde ich solange wieder unterdrücken. Der Index `ia` ist größer als `i`, und somit wählen wir zum Beweis der Existenzaussage den Index `ia`. Wir verwenden die Regel

```
 ?P(?x)⟹∃x. ?P(x)
- by (res_inst_tac [("x","ia")] exI 1);
```

mit konkreter Instantiierung und erhalten:

```
1. ⋀i ia.
      ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
         i < ia⟧⟹
      CF2(FY(ia), TY(i)) ⊑ CF2(FY(ia), TY(ia))
```

Durch Anwendung der Monotonieregel

```
 ⟦ is_chain(?F1); ?x1 < ?y1⟧⟹?F1(?x1) ⊑ ?F1(?y1)
- by (rtac (chain_mono RS mp) 1);
```

für Ketten reduzieren wir auf die Teilziele:

```
1. ⋀i ia.
      ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
         i < ia⟧⟹
      is_chain(λu. CF2(FY(ia), TY(u)))
2. ⋀i ia.
      ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
         i < ia⟧⟹
      i < ia
```

Das zweite lösen wir trivial per Annahme, und das erste lösen wir durch Verwendung des Hilfstheorems

```
diag_lemma3
   ⟦ ∀f. contX(?CF2.0(f)); is_chain(?FY); is_chain(?TY)⟧⟹
   is_chain(λm. ?CF2.0(?FY(?n), ?TY(m)))

- by (etac diag_lemma3 1);
- by (REPEAT (atac 1));
```

Wenden wir uns nun dem zweiten Fall `i = ia` zu, der besonders einfach ist.

```
 1. ⋀i ia.
       ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
          i = ia⟧⟹
       ∃j. CF2(FY(ia), TY(i)) ⊑ CF2(FY(j), TY(j))
```

Durch Anwendung von Gleichungslogik ersetzen wir `i` durch `ia` und erhalten:

```
- by (hyp_subst_tac 1);

 1. ⋀i ia.
       ⟦ is_chain(TY); is_chain(FY); ∀f. contX(CF2(f)); contX(CF2)⟧⟹
       ∃j. CF2(FY(ia), TY(ia)) ⊑ CF2(FY(j), TY(j))
```

Offensichtlich wählen wir in diesem Fall `ia` als Zeugen für die Existenzaussage. Dadurch löst sich das Teilziel durch triviale Anwendung der Reflexivitätsregel für die Ordnung $\sqsubseteq$.

```
- by (res_inst_tac [("x","ia")] exI 1);
- by (rtac refl_less 1);
```

Im dritten Fall `ia < i` argumentieren wir analog zum Fall `i < ia`. Durch Anwendung von

```
- by (res_inst_tac [("x","i")] exI 1);
- by (rtac (chain_mono RS mp) 1);
- by (etac diag_lemma2 1);
- by (REPEAT (atac 1));
```

wird das Teilziel

```
 1. ⋀i ia.
       ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY);
          ia < i⟧⟹
       ∃j. CF2(FY(ia), TY(i)) ⊑ CF2(FY(j), TY(j))
```

vollständig gelöst, und die eine Richtung der Inklusionsbeziehung ist hiermit gezeigt.

Es bleibt noch die umgekehrte Richtung der Inklusion zu zeigen. Das bislang zurückgestellte Beweisziel lautet:

```
 1. ⟦ contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)⟧⟹
    lub(range(λi. CF2(FY(i), TY(i)))) ⊑
    lub(range(λi. lub(range(λj. CF2(FY(j), TY(i))))))
```

Auch in diesem Fall argumentieren wir über eine Majoranteneigenschaft, welche aber im Gegensatz zum vorhergehenden Fall wesentlich einfacher ist. Diesmal reicht es, die Inklusionsbeziehung der Ketten punktweise zu zeigen. Wir verwenden die Regel

```
   ⟦ is_chain(?C1.0); is_chain(?C2.0); ∀k. ?C1.0(k) ⊑ ?C2.0(k)⟧⟹
   lub(range(?C1.0)) ⊑ lub(range(?C2.0))
- by (rtac lub_mono 1);
```

und erhalten:

```
1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
   is_chain(λi. CF2(FY(i), TY(i)))
2. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
   is_chain(λi. lub(range(λj. CF2(FY(j), TY(i)))))
3. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
   ∀k. CF2(FY(k), TY(k)) ⊑ lub(range(λj. CF2(FY(j), TY(k))))
```

Die ersten beiden Teilziele werden über die schon vorher verwendeten Hilfstheoreme `diag_lemma4` und `diag_lemma1` gezeigt. Im dritten Teilziel reicht es, die Behauptung für ein beliebiges, aber festes `k` zu zeigen.

```
- by (etac diag_lemma4 1);
- by (REPEAT (atac 1));
- by (etac diag_lemma1 1);
- by (REPEAT (atac 1));
- by (strip_tac 1 );
```

Wir erhalten als verbleibendes Ziel:

```
1. ⋀k. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]
          ==>CF2(FY(k), TY(k)) ⊑ lub(range(λj. CF2(FY(j), TY(k))))
```

Dieses lösen wir durch die Tatsache, daß kleinste obere Schranken einer Kette insbesondere größer als alle Kettenelemente sind. Wir verwenden die Regel

```
is_chain(?S1)==>?S1(?x) ⊑ lub(range(?S1))
- by (rtac is_ub_thelub 1);
```

Dies führt zu einem neuen Teilziel

```
1. ⋀k. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]
          ==>is_chain(λj. CF2(FY(j), TY(k)))
```

welches wir aber in bewährter Weise durch Verwendung des Hilfstheorems `diag_lemma2` lösen.

```
- by (etac diag_lemma2 1);
- by (REPEAT (atac 1));
No subgoals
```

4.6.3.2 Beweis für das Theorem `contlub_CF2`

Der Beweis für das Theorem `contlub_CF2` macht im wesentlichen Gebrauch vom Theorem `diag_lubCF2_2`, welches seinerseits trivial aus `diag_lubCF2_1` durch Anwendung des Theorems `ex_lubMF2` folgt. Der folgende Beweis zeigt die Anwendung der Verkettung von Regeln mittels Resolution `RS`, wodurch Rückwärts- und Vorwärtsbeweise im Isabelle-System kombiniert werden können.

Wir verschaffen uns das initiale Beweisziel mittels

```
- val prems = goal Cont.thy
"[| contX(CF2);∀f.contX(CF2(f));is_chain(FY);is_chain(TY)|]==>
  CF2(lub(range(FY)))(lub(range(TY))) = lub(range(λi.CF2(FY(i))(TY(i))))";
```

```
- by (cut_facts_tac prems 1);
```

und erhalten:

```
 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
      CF2(lub(range(FY)), lub(range(TY))) = lub(range(λi. CF2(FY(i), TY(i))))
```

Die Idee ist nun, die Stetigkeit von `CF2` im ersten Argument zu verwenden. Wir ersetzen
`CF2(lub(range(FY)), lub(range(TY)))`
durch den Term
`lub(range(λi.CF2(FY(i))),lub(range(TY)))`

Man beachte dabei, daß Isabelle die Anwendung einer Funktion höherer Stufe `f(x)(y)` als `f(x,y)` druckt! Dies ist eine bisweilen verwirrende Eigenheit des Systems. Da hier massiv Funktionen höherer Stufe beteiligt sind, müssen wir die Unifikation des Systems geeignet steuern. Wir verwenden hierzu die Verkettung von Regeln mittels Resolution `RS`. Über `(hd prems)` sprechen wir die erste Prämisse im Kontext an. Sie ist zusammen mit den restlichen an der kombinierten Regel beteiligten Theoremen im folgenden aufgelistet:

```
(hd prems)      contX(CF2)
contX2contlub   contX(?f)==>contlub(?f)
contlubE        contlub(?f)==>∀Y. is_chain(Y) →
                                  ?f(lub(range(Y))) = lub(range(λi. ?f(Y(i))))
spec            ∀x. ?P(x)==>?P(?x)
mp              [| ?P → ?Q; ?P|]==>?Q
ssubst          [| ?t = ?s; ?P(?s)|]==>?P(?t)
```

Die Kombination dieser Regeln
`((hd prems) RS contX2contlub RS contlubE RS spec RS mp RS ssubst)`

mittels `RS` liefert die Regel

```
   [| is_chain(?x2); ?P(lub(range(λi. CF2(?x2(i)))))|]
       ==>?P(CF2(lub(range(?x2))))
```

Anwendung dieser Regel ergibt die gewünschte Reduktion des Beweisziels:

```
- by (rtac ((hd prems) RS contX2contlub RS contlubE RS
             spec RS mp RS ssubst) 1);

 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    is_chain(FY)
 2. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    lub(range(λi. CF2(FY(i))), lub(range(TY))) =
    lub(range(λi. CF2(FY(i), TY(i))))
```

Das erste Beweisziel lösen wir trivial per Annahme.

```
- by (atac 1);
```

Im verbleibenden Teilziel formen wir die linke Seite der Gleichung noch weiter um, indem wir die konkrete Bauart der kleinsten oberen Schranke einer Kette von Funktionen einsetzen.

Wir verwenden hierzu die kombinierte Regel

```
   [| is_chain(?S2); ?P(λx. lub(range(λi. ?S2(i, x)))) |]
      ==>?P(lub(range(?S2)))
- by (rtac (thelub_fun RS ssubst) 1);
```

und erhalten:

```
 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY) |]==>
    is_chain(λi. CF2(FY(i)))
 2. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY) |]==>
    lub(range(λi. CF2(FY(i), lub(range(TY))))) =
    lub(range(λi. CF2(FY(i), TY(i))))
```

Das erste Teilziel erfordert den Beweis einer Ketteneigenschaft, welche wir aber durch Verwendung der Stetigkeit der Funktion `CF2` im ersten Argument leicht nachweisen.

```
- by (rtac ((hd prems) RS contX2mono RS ch2ch_monofun) 1);
- by (atac 1);
```

Es verbleibt das Teilziel:

```
 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY) |]==>
    lub(range(λi. CF2(FY(i), lub(range(TY))))) =
    lub(range(λi. CF2(FY(i), TY(i))))
```

Indem wir die Stetigkeit der Funktion `CF2` im zweiten Argument verwenden und geeignet mit Kongruenz- und Extensionalitätseigenschaften von Funktionen kombinieren, ersetzen wir den Term

```
lub(range(λi. CF2(FY(i), lub(range(TY)))))
```

durch den Term

```
lub(range(λi. lub(range(λia. CF2(FY(i), TY(ia))))))
```

Dies wird durch folgendes Beweisskript erreicht:

```
- by (rtac trans 1);
- by (rtac (((hd (tl prems)) RS spec RS contX2contlub) RS contlubE RS
               spec RS mp RS ext RS arg_cong RS arg_cong) 1);
- by (atac 1);
```

Wir erhalten als neues Teilziel:

```
 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY) |]==>
    lub(range(λi. lub(range(λia. CF2(FY(i), TY(ia)))))) =
    lub(range(λi. CF2(FY(i), TY(i))))
```

Nun haben wir einen Beweiszustand erreicht, in dem wir das Theorem

```
diag_lubCF2_2
   [| contX(?CF2.0); ∀f. contX(?CF2.0(f)); is_chain(?FY); is_chain(?TY) |]==>
      lub(range(λj. lub(range(λi. ?CF2.0(?FY(j), ?TY(i)))))) =
      lub(range(λi. ?CF2.0(?FY(i), ?TY(i))))
```

einsetzen können. Wir erhalten:

```
- by (rtac diag_lubCF2_2 1);

 1. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    contX(CF2)
 2. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    ∀f. contX(CF2(f))
 3. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    is_chain(FY)
 4. [| contX(CF2); ∀f. contX(CF2(f)); is_chain(FY); is_chain(TY)|]==>
    is_chain(TY)
```

Alle drei Ziele sind trivial per Annahme lösbar.

```
- by (REPEAT (atac 1));
No subgoals
```

4.6.3.3 Beweis für das Theorem contX2contlub_app

Dieses Theorem ist rein technischer Natur und ist die Grundlage für das Theorem contX2contX_app (siehe Abbildung 4.20) und damit auch für den Beweis des Theorems contX2contX_fapp in Theorie Cfun3 (siehe Abschnitt 4.7.8).

Wir verschaffen uns das initiale Beweisziel mittels

```
- val prems = goal Cont.thy
"[| contX(ft);∀x.contX(ft(x));contX(tt)|]==>contlub(λx.(ft(x))(tt(x)))";

- by (cut_facts_tac prems 1);
```

Dies liefert:

```
 1. [| contX(ft); ∀x. contX(ft(x)); contX(tt)|]==>contlub(λx. ft(x, tt(x)))
```

Im ersten Schritt entfalten wir die Definition des Prädikats contlub

```
   ∀Y. is_chain(Y) → ?f(lub(range(Y))) =lub(range(λi. ?f(Y(i))))
     ==>contlub(?f)

- by (rtac contlubI 1);
```

und erhalten:

```
 1. [| contX(ft); ∀x. contX(ft(x)); contX(tt)|]==>
    ∀Y. is_chain(Y) →
         ft(lub(range(Y)), tt(lub(range(Y)))) =
         lub(range(λi. ft(Y(i), tt(Y(i)))))
```

Gemäß der Methodik des natürlichen Schließens schaffen wir uns einen passenden Annahmenkontext.

```
- by (strip_tac 1);

 1. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         ft(lub(range(Y)), tt(lub(range(Y)))) =
         lub(range(λi. ft(Y(i), tt(Y(i)))))
```

Wir verwenden die Stetigkeit der Funktion `tt` und ersetzen durch Gleichungslogik den Teilterm `tt(lub(range(Y)))` durch den Term `lub(range(λi. tt(Y(i))))`. Dies erfolgt durch den Befehl:

```
- by (res_inst_tac [("f3","tt")] (contlubE RS spec RS mp RS ssubst) 1);
```

Wir erhalten:

```
 1. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         contlub(tt)
 2. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         is_chain(Y)
 3. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         ft(lub(range(Y)), lub(range(λi. tt(Y(i))))) =
         lub(range(λi. ft(Y(i), tt(Y(i)))))
```

Die beiden ersten Teilziele lassen sich leicht durch Verwendung des Annahmenkontextes lösen.

```
- by (etac contX2contlub 1);
- by (atac 1);
```

Es verbleibt das Ziel:

```
 1. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         ft(lub(range(Y)), lub(range(λi. tt(Y(i))))) =
         lub(range(λi. ft(Y(i), tt(Y(i)))))
```

In dieser Situation können wir aber das Theorem `contlub_CF2`

```
   [| contX(?CF2.0); ∀f. contX(?CF2.0(f)); is_chain(?FY); is_chain(?TY)|]==>
   ?CF2.0(lub(range(?FY)), lub(range(?TY))) =
   lub(range(λi. ?CF2.0(?FY(i), ?TY(i))))

- by (rtac contlub_CF2 1);
```

einsetzen, was zu den folgenden trivialen Teilzielen führt:

```
 1. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         contX(ft)
 2. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         ∀f. contX(ft(f))
 3. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         is_chain(Y)
 4. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
         is_chain(λi. tt(Y(i)))
```

Die ersten drei lösen wir direkt per Annahme.

```
- by (REPEAT (atac 1));
```

Das verbleibende Teilziel:

```
 1. ⋀Y. [| contX(ft); ∀x. contX(ft(x)); contX(tt); is_chain(Y)|]==>
          is_chain(λi. tt(Y(i)))
```

lösen wir, indem wir die kombinierte Regel

```
 [| contX(?f); is_chain(?Y)|]==>is_chain(λi. ?f(?Y(i)))
- by (etac (contX2mono RS ch2ch_monofun) 1);
```

mit anschließendem Beweis per Annahme verwenden.

```
- by (atac 1);
No subgoals
```

4.7 Theorien für Operationen

Stetige Funktionen spielen in der Bereichstheorie für LCF eine wesentliche Rolle. Sie bilden eine Teilmenge des vollen Funktionenraums $\alpha_{\texttt{pcpo}} \Rightarrow \beta_{\texttt{pcpo}}$ und werden durch das Prädikat `contX` aus der Theorie `Cont` in Abschnitt 4.6 ausgezeichnet. Im Prinzip könnte der noch fehlende Teil der Bereichstheorie für LCF auf der syntaktischen Basis der Theorie `Cont` entwickelt werden. Dies würde aber zu einer schwerfälligen Notation führen, da nahezu jedes Theorem in irgendeiner Weise das Prädikat `contX` enthalten würde. Somit wäre auch das erklärte Ziel dieser Arbeit nicht erreicht, die Logik LCF ohne größere syntaktische Änderungen in der Logik HOLC zu repräsentieren. Stattdessen wird ein neuer Typkonstruktor eingeführt, der in geeigneter Weise mit den stetigen Funktionen gekoppelt ist. Die konservative Einführung des Konstruktors gliedert sich in drei Teilschritte.

4.7.1 Die Theorie `Cfun1`

Die zentrale Rolle der stetigen Funktionen rechtfertigt die Einführung eines speziellen Typkonstruktors $\rightarrow$, so daß die Elemente des Typs $\alpha_{\texttt{pcpo}} \rightarrow \beta_{\texttt{pcpo}}$ in geeigneter Weise mit den stetigen Funktionen über den Bereichen α und β zusammenhängen. Um Verwirrung zu vermeiden, muß mit Einführung des Konstruktors $\rightarrow$ aber auch eine neue Terminologie geprägt werden. Als *Funktionen* werden Elemente des Typs $\alpha \Rightarrow \beta$ bezeichnet. *Stetige Funktionen* sind Elemente des Typs $\alpha \Rightarrow \beta$, die zusätzlich noch das Prädikat `contX` erfüllen. Im Gegensatz dazu werden Elemente des Typs $\alpha \rightarrow \beta$ als *Operationen* bezeichnet. Die Theorie `Cfun1`, die den Typkonstruktor $\rightarrow$ einführt, ist in Abbildung 4.21 dargestellt. In ihr wird der Zusammenhang zwischen Operationen und stetigen Funktionen deutlich gemacht.

Gemäß der Methode für die konservative Einführung eines neuen Typkonstruktors aus Abschnitt 2.6 wird der Typkonstruktor $\rightarrow$ mit der Arität `(pcpo,pcpo)term` eingeführt. Die Klasse `pcpo` auf Argumentposition ist notwendig, da sonst das Repräsentationsprädikat `contX` nicht eingesetzt werden kann. Der Isabelle-Technik folgend, führen wir eine Bezeichnung `Cfun`

für die Menge der Repräsentanten ein. Die Definition `Cfun_def` legt fest, daß es sich hierbei um die Menge aller stetigen Funktionen handelt. Die Repräsentationsfunktion ist `fapp`, und die Abstraktionsfunktion ist `fabs`. Die Typdefinition wird durch die Axiome `Rep_Cfun`, `Rep_Cfun_inverse` und `Abs_Cfun_inverse` in bekannter Weise bewerkstelligt.

```
Cfun1 = Cont +

types   →   2        (infixr 5)

arities   →   :: (pcpo,pcpo)term

consts
        Cfun     ::  (α ⇒ β)set

        fapp     ::  (α → β) ⇒ (α ⇒ β)       ( (_[_])  [1000,0] 1000)
        fabs     ::  (α ⇒ β) ⇒ (α → β)       (binder  Λ   10)

        less_cfun ::  [(α → β),(α → β)] ⇒ bool

rules

  Cfun_def               Cfun ≡ {f. contX(f)}

  Rep_Cfun               fapp(fo)∈Cfun
  Rep_Cfun_inverse       fabs(fapp(fo)) = fo
  Abs_Cfun_inverse       f∈Cfun⟹fapp(fabs(f))=f

  less_cfun_def          less_cfun(fo1,fo2) ≡ ( fapp(fo1) ⊑ fapp(fo2) )

end
```

Abbildung 4.21: Theorie `Cfun1`

Als letztes wird noch die partielle Ordnung der Funktionen mittels der Repräsentationsfunktion `fapp` auf die Operationen übertragen. Ein Teil der noch folgenden Beweisarbeit wird sich mit dem Nachweis beschäftigen, daß die Funktion `less_cfun` den Typ der Operationen mit einer kettenvollständigen partiellen Ordnung versieht, um so die Aritäten $\rightarrow$ ::`(pcpo,pcpo)po` und $\rightarrow$ ::`(pcpo,pcpo)pcpo` in den Theorien `Cfun2` und `Cfun3` zu rechtfertigen.

Die Repräsentationsfunktion `fapp` dient gleichzeitig auch als Applikationsfunktion für Operationen. Wenn `f` eine Operation vom Typ $\alpha \rightarrow \beta$ ist, so ist `fapp(f)` eine Funktion vom Typ $\alpha \Rightarrow \beta$. Diese kann ihrerseits auf ein `x` vom Typ α angewendet werden. Der Term `fapp(f)(x)` kann also als Applikation der Operation `f` auf `x` gedeutet werden. Um diese Auffassung zu unterstreichen, wurde für die Konstante `fapp` die Mixfix-Schreibweise `_[_]` vereinbart. Statt `fapp(f)(x)` darf man `f[x]` schreiben. Entsprechend dazu wurde für die Abstraktionsfunktion `fabs` ein neuer Bindungsmechanismus eingeführt. Statt `fabs(`λ`x.t)` darf man Λ`x.t`

schreiben. Somit erhält man einen Abstraktionsmechanismus für Operationen, der sich auf die λ-Abstraktion der Logik HOLC abstützt.

Die gerade beschriebenen notationellen Vereinbarungen, die durch das Isabelle-System ermöglicht werden, sind zusammen mit der Verwendung von Typklassen der wesentliche Grund dafür, daß die hier vorgestellte Entwicklung von HOLCF praktisch benutzbar ist. Während der noch folgenden Entwicklung von HOLCF muß man selbverständlich bisweilen die unterliegende Notation mittels `fapp` und `fabs` verwenden, aber nach Abschluß der Entwicklung kann man vollständig davon abstrahieren und die Notation `f[x]` und Λ`x.t` als primitiv ansehen. Das Studium der noch folgenden Theorien mit ihren Theoremen wird schrittweise zu einem besseren Verständnis der Operationen führen.

4.7.2 Theoreme der Theorie `Cfun1`

Die Theoreme der Theorie `Cfun1` sind in Abbildung 4.22 dargestellt.

```
CfunI                (λx.x)∈Cfun

refl_less_cfun       less_cfun(f,f)
antisym_less_cfun    ⟦ less_cfun(f1,f2); less_cfun(f2,f1)⟧⟹f1 = f2
trans_less_cfun      ⟦ less_cfun(f1,f2); less_cfun(f2,f3)⟧
                       ⟹less_cfun(f1,f3)

cfun_cong            ⟦ f=g; x=y⟧⟹f[x] = g[y]
cfun_fun_cong        f=g⟹f[x] = g[x]
cfun_arg_cong        x=y⟹f[x] = f[y]

Abs_Cfun_inverse2    contX(f)⟹fapp(fabs(f)) = f
Cfunapp2             contX(f)⟹(fabs(f))[x] = f(x)

beta_cfun            contX(c1)⟹(Λx.c1(x))[u] = c1(u)
```

Abbildung 4.22: Theoreme der Theorie `Cfun1`

Das erste Theorem ist eine Rechtfertigung für die Einführung des Typkonstruktors $\rightarrow$. Das Theorem wurde nur aus formalen Gründen bewiesen, da mit dem Theorem `contX_id` aus Theorie `Cont` in Abbildung 4.20 ohnehin gezeigt wurde, daß die Repräsentationsmenge `Cfun` nicht leer sein kann.

Die Theoreme `refl_less_cfun`, `antisym_less_cfun` und `trans_less_cfun` zeigen, daß der Typ $\alpha \rightarrow \beta$ durch die Funktion `less_cfun` partiell geordnet wird. Dieser Umstand wird in der folgenden Theorie `Cfun2` ausgenützt.

Die Theoreme `cfun_cong`, `cfun_fun_cong` und `cfun_arg_cong` beschreiben Kongruenzeigenschaften von Operationen. Die Theoreme `Abs_Cfun_inverse2` und `Cfunapp2` sind rein technischer Natur und erlauben nur einen leichteren Umgang mit den Axiomen der Typdefinition.

Im Gegensatz dazu ist das Theorem `beta_cfun` von praktischem Interesse im Umgang mit Operationen. Es beschreibt die eingeschränkte β-Reduktion bzgl. der Λ-Abstraktion für Operationen. Der Beweis des Theorems ist trivial und folgt direkt aus den Typdefinitionsaxiomen. Das Theorem verdeutlicht den Zusammenhang zwischen Operationen und Funktionen.

Betrachten wir als Beispiel dazu die Funktion `λx.x=x`. In diesem Beispiel sei für den Typ `bool` die Arität `bool::pcpo` vereinbart, und die `pcpo`-Ordnung sei eine der beiden für `bool` möglichen. Die Funktion `λx.x=x` ist extensional gleich zur Identitätsfunktion `=`, die bzgl. keiner der möglichen Ordnungen monoton sein kann. Da die Abstraktionsfunktion `fabs` der Operationen total ist, wie alle Funktionen der Logik HOLC, beschreibt `fabs(λx.x=x)` trotzdem eine Operation. Wenn wir nun die Repräsentationsfunktion `fapp` anwenden, so beschreibt der Term `fapp(fabs(λx.x=x))` wegen Axiom `Rep_Cfun` ein Element in der Menge `Cfun`, und somit auch eine stetige Funktion. Um welche Funktion es sich hierbei handelt, wissen wir aber nicht. Es kann nicht `λx.x=x` sein, da diese Funktion nicht monoton ist. In diesem Fall läßt sich also die Anwendung der Abstraktionsfunktion `fabs` nicht durch die Anwendung der Repräsentationsfunktion `fapp` rückgängig machen.

Wenn wir die Mixfix-Syntax verwenden, erkennen wir die praktische Bedeutung des Beispiels. Die Mixfix-Schreibweise für die Operation `fabs(λx.x=x)` ist `Λx.x=x`. Die Anwendung der Operation mittels `fapp` auf ein `y` ergibt den Term `fapp(fabs(λx.x=x))(y)`, der in Mixfix-Schreibweise `(Λx.x=x)[y]` lautet. Diesen Term können wir aber nicht zu `(λx.x=x)(y)` und weiter zu `y=y` reduzieren, da hierzu wegen Typdefinitionsaxiom `Abs_Cfun_inverse` ein Nachweis der Stetigkeit von `λx.x=x` benötigt wird. Da die Identitätsfunktion `=` nicht einmal monoton ist, werden alle Bemühungen in dieser Richtung vergebens sein.

Um auch ein positives Beispiel anzugeben, betrachten wir den Term `Λx.x`, der für `fabs(λx.x)` steht. Die Applikation dieser Operation auf ein `y` lautet `fapp(fabs(λx.x))(y)` bzw. in Mixfix-Syntax `(Λx.x)[y]`. Diesen Term können wir wegen `contX(λx.x)` und Typdefinitionsaxiom `Abs_Cfun_inverse` zu `(λx.x)(y)` und weiter zu `y` reduzieren.

Der eben skizzierte Sachverhalt verdeutlicht das Zusammenspiel von evtl. nicht-stetigen Funktionen und Operationen. Terme, in denen Operationen und Funktionen gemischt vorkommen, dürfen im Gegensatz zur Sprache SPECTRUM uneingeschränkt gebildet werden, ihre weitere Verwendung ist jedoch durch Stetigkeitsforderungen beschränkt, wie aus dem Beispiel `(Λx.x=x)[y]` ersichtlich ist.

4.7.3 Die Theorie `Cfun2`

In der Theorie `Cfun2` wird die Arität `→ ::(pcpo,pcpo)po` mit dem entsprechenden Instanzaxiom eingeführt. Die Rechtfertigung für diese Aritätsvereinbarung wird durch die Theoreme `refl_less_cfun`, `antisym_less_cfun` und `trans_less_cfun` geliefert. Die Theorie ist in Abbildung 4.23 dargestellt.

Darüberhinaus wird die Konstante `⊥_cfun` durch Konstantendefinition eingeführt. Es wird sich herausstellen, daß `⊥_cfun` die kleinste Operation ist. In der Definition für die Konstante `⊥_cfun` habe ich statt der Mixfix-Notation `Λx.⊥` die explizite Schreibweise `fabs(λx.⊥)` gewählt, um den Zusammenhang mit dem kleinsten Element `λx.⊥` im vollen Funktionenraum zu verdeutlichen.

```
Cfun2 = Cfun1 +

arities   →   :: (pcpo,pcpo)po

consts
         ⊥_cfun:: α → β
rules

inst_cfun_po      ( ⊑ ::[α → β,α → β] ⇒ bool) = less_cfun

⊥_cfun_def        ⊥_cfun ≡ fabs(λx.⊥)

end
```

Abbildung 4.23: Theorie Cfun2

4.7.4 Theoreme der Theorie Cfun2

Die Theoreme der Theorie Cfun2 sind in Abbildung 4.24 dargestellt.

```
less_cfun =        ( f1 ⊑ f2 ) = (fapp(f1) ⊑ fapp(f2))

minimal_cfun       ⊥_cfun ⊑ f

contX_fapp2        contX(fapp(fo))
monofun_fapp2      monofun(fapp(fo1))
contlub_fapp2      contlub(fapp(fo1))

contX_cfun_arg     is_chain(x1)⟹
                    range(λi.fo[x1(i)]) ≪| fo[lub(range(x1))]
contlub_cfun_arg   is_chain(x1)⟹
                    fo[lub(range(x1))] = lub(range(λi.fo[x1(i)]))

monofun_fapp1      monofun(fapp)
monofun_cfun_fun   f1 ⊑ f2⟹f1[x] ⊑ f2[x]
monofun_cfun_arg   x2 ⊑ x1⟹fo[x2] ⊑ fo[x1]
monofun_cfun       ⟦ f1 ⊑ f2;x1 ⊑ x2⟧⟹f1[x1] ⊑ f2[x2]

ch2ch_fappR        is_chain(Y)⟹is_chain(λi.f[Y(i)])
ch2ch_fappL        is_chain(F)⟹is_chain(λi.F(i)[x])

lub_cfun_mono      is_chain(F)⟹monofun(λx.lub(range(λj.F(j)[x])))
ex_lubcfun         ⟦ is_chain(F); is_chain(Y)⟧
                     ⟹ lub(range(λj.lub(range(λi.F(j)[Y(i)])))) =
                        lub(range(λi.lub(range(λj.F(j)[Y(i)]))))
```

```
contX_lubcfun      is_chain(F)⟹contX(λx.lub(range(λj.F(j)[x])))
lub_cfun           is_chain(CCF)⟹
                    range(CCF) ≪| fabs(λx.lub(range(λi.CCF(i)[x])))
thelub_cfun        is_chain(CCF)⟹
                    lub(range(CCF)) = fabs(λx.lub(range(λi.CCF(i)[x])))
cpo_cfun           is_chain(CCF::nat ⇒ (α_pcpo → β_pcpo))
                    ⟹∃x. range(CCF) ≪| x

ext_cfun           (⋀x. f[x] = g[x])⟹f = g
semi_monofun_fabs  ⟦ contX(f);contX(g);f ⊑ g⟧⟹fabs(f) ⊑ fabs(g)
less_cfun2         (⋀x. f[x] ⊑ g[x])⟹f ⊑ g
```

Abbildung 4.24: Theoreme der Theorie `Cfun2`

Das erste Theorem nützt die neue Arität aus und formuliert die Ordnung auf den Operationen mittels der charakteristischen Funktion $\sqsubseteq$. Das Theorem `minimal_cfun` zeigt, daß `⊥_cfun` die kleinste Operation ist.

Die Theoreme `contX_fapp2` bis `contlub_cfun_arg` drücken die Stetigkeit der Funktion `fapp` im zweiten Argument aus, sowie einige Folgerungen daraus. Das Theorem `contlub_cfun_arg` hat den gleichen logischen Gehalt wie das Theorem `contlub_fapp2`, mit dem Unterschied, daß die Mixfix-Schreibweise verwendet wird.

Die Theoreme `monofun_fapp1` bis `monofun_cfun` beschreiben die Monotonie der Funktion `fapp` im ersten und zweiten Argument in verschiedenen Variationen. Man beachte, daß es sich hierbei exakt um die Monotonieregeln der Logik LCF handelt. Die Theoreme `ch2ch_fappR` und `ch2ch_fappL` zeigen die Propagierung von Ketten.

Die Theoreme `lub_cfun_mono` und `contX_lubcfun` zeigen, daß die Abstraktion über die kleinste obere Schranke der Applikationen einer Kette von Operationen selbst wieder eine monotone und stetige Funktion ist. Das Theorem `lub_cfun` zeigt, daß sich daraus durch Anwendung der Abstraktionsfunktion `fabs` die kleinste obere Schranke der verwendeten Kette von Operationen gewinnen läßt. Diese Tatsache nützt das Theorem `cpo_cfun` aus, das die Kettenvollständigkeit des Typs der Operationen beschreibt. In Abschnitt 4.7.5 werde ich die Beweise für die Theoreme `contX_lubcfun` und `lub_cfun` explizit vorführen.

Die Theoreme `ext_cfun` und `less_cfun2` beschreiben Extensionalitätseigenschaften von Operationen bzgl. der Relationen `=` und $\sqsubseteq$. Damit sind zwei weitere Inferenzregeln der Logik LCF abgeleitet.

4.7.5 Beweise für ausgesuchte Theoreme

In diesem Abschnitt werden die Beweise für die Theoreme `contX_lubcfun` und `lub_cfun` vorgeführt.

4.7.5.1 Beweis für das Theorem contX_lubcfun

Wie beginnen, indem wir uns das Beweisziel mittels
```
val prems = goal Cfun2.thy
        "is_chain(F)⟹contX(λx.lub(range(λj.F(j)[x])))";
- by (cut_facts_tac prems 1);
```
vorgeben und erhalten:
```
 1. is_chain(F)⟹contX(λx. lub(range(λj. F(j)[x])))
```

Die Regel
```
 ⟦ monofun(?f); contlub(?f)⟧⟹contX(?f)
- by (rtac monocontlub2contX 1);
```
erlaubt uns, den Nachweis der Stetigkeit getrennt über den Nachweis der Monotonie und der Vertauschbarkeit der Grenzwertbildung mit der Applikation zu zeigen. Wir erhalten die neuen Teilziele:
```
 1. is_chain(F)⟹monofun(λx. lub(range(λj. F(j)[x])))
 2. is_chain(F)⟹contlub(λx. lub(range(λj. F(j)[x])))
```

Das erste lösen wir vollständig durch Anwendung des Theorems
```
 is_chain(?F)⟹monofun(λx. lub(range(λj. ?F(j)[x])))
- by (etac lub_cfun_mono 1);
```

Im zweiten Ziel falten wird die Definition des Prädikats `contlub` auf, und erhalten:
```
- by (rtac contlubI 1);

 1. is_chain(F)⟹
    ∀Y. is_chain(Y) →
         lub(range(λj. F(j)[lub(range(Y))])) =
         lub(range(λi. lub(range(λj. F(j)[Y(i)]))))
```

Im nächsten Schritt erzeugen wir uns standardmäßig einen geeigneten Annahmenkontext für den weiteren Beweis:
```
- by (strip_tac 1);

 1. ⋀Y. ⟦ is_chain(F); is_chain(Y)⟧⟹
         lub(range(λj. F(j)[lub(range(Y))])) =
         lub(range(λi. lub(range(λj. F(j)[Y(i)]))))
```

Sodann vertauschen wir die Applikation der Operation `F(j)` mit der Grenzwertbildung. Dies erlaubt uns die Regel `contlub_cfun_arg`
```
 is_chain(?x1)⟹?fo4[lub(range(?x1))] = lub(range(λi. ?fo4[?x1(i)]))
```

die wir geeignet mit der Extensionalitätsregel `ext` und der Substitutionsregel `ssubst` kombinieren. Das reduziert unser Beweisziel zu:
```
- by (rtac (contlub_cfun_arg RS ext RS ssubst) 1);
```

```
 1. ⋀Y x. [| is_chain(F); is_chain(Y)|]==>is_chain(Y)
 2. ⋀Y. [| is_chain(F); is_chain(Y)|]==>
        lub(range(λj. lub(range(λi. F(j)[Y(i)])))) =
        lub(range(λi. lub(range(λj. F(j)[Y(i)]))))
```

Das erste Teilziel lösen wir leicht per Annahme. Das zweite Teilziel wird vollständig durch Anwendung des Theorems `ex_lubcfun`

```
   [| is_chain(?F); is_chain(?Y)|]==>
   lub(range(λj. lub(range(λi. ?F(j)[?Y(i)])))) =
   lub(range(λi. lub(range(λj. ?F(j)[?Y(i)]))))
```

gelöst.

```
- by (atac 1);
- by (etac ex_lubcfun 1);
- by (atac 1);
No subgoals
```

4.7.5.2 Beweis für das Theorem `lub_cfun`

Mit dem Theorem `lub_cfun` beweisen wir, daß die kleinste obere Schranke einer Kette von Operationen `CCF` die Operation `Λx.lub(range(λi. CCF(i)[x]))` ist. Im Beweis müssen wir zwischen Operationen und Funktionen konvertieren und daher verwenden wir die explizite Schreibweise `fabs(λx.t)`, um die Gedankengänge klarer zu machen. Das Isabelle-System macht diese Bemühung jedoch teilweise zunichte, da es seinerseits automatisch die Mixfix-Schreibweise verwendet. Wir beginnen, indem wir uns das Beweisziel vorgeben.

```
- val prems = goal Cfun2.thy
   "is_chain(CCF)==>range(CCF) <<| fabs(λx. lub(range(λi.CCF(i)[x])))";
- by (cut_facts_tac prems 1);
```

und erhalten:

```
 1. is_chain(CCF)==>range(CCF) <<| (Λx. lub(range(λi. CCF(i)[x])))
```

Wir entfalten die Definition der kleinsten oberen Schranke mittels

```
 ?S <<| ?x ∧ (∀u. ?S <<| u → ?x ⊑ u)==>?S <<| ?x
- by (rtac is_lubI 1);
```

was uns folgendes Teilziel beschert:

```
 1. is_chain(CCF)==>
    range(CCF) <<| (Λx.lub(range(λi. CCF(i)[x]))) ∧
    (∀u. range(CCF) <<| u → (Λx. lub(range(λi. CCF(i)[x]))) ⊑ u)
```

Der nächste Schritt ist kanonisch vorgegeben. Wir beweisen die Teile der Konjunktion in zwei Teilbeweisen.

```
- by (rtac conjI 1);
```

```
 1. is_chain(CCF)⟹range(CCF) <| (Λx. lub(range(λi. CCF(i)[x])))
 2. is_chain(CCF)⟹
    ∀u. range(CCF) <| u → (Λx. lub(range(λi. CCF(i)[x]))) ⊑ u
```

Im folgenden lösen wir zuerst das erste Teilziel. Erst wenn dieses abgearbeitet ist, werde ich das zweite Teilziel wieder auflisten. Im ersten Ziel entfalten wir die Definition für obere Schranken mittels

```
- by (rtac ub_rangeI 1);
 ∀i. ?S(i) ⊑ ?x⟹range(?S) <| ?x
```

Dies führt zum Teilziel:

```
 1. is_chain(CCF)⟹∀i. CCF(i) ⊑ (Λx. lub(range(λi. CCF(i)[x])))
```

Es reicht, die Behauptung für ein beliebiges, aber festes `i` zu zeigen.

```
- by (rtac allI 1);

 1. ⋀i. is_chain(CCF)⟹CCF(i) ⊑ (Λx. lub(range(λi. CCF(i)[x])))
```

Wir müssen zeigen, daß zwei Operationen in Ordnungsrelation $\sqsubseteq$ stehen. Dazu verwenden wir die Regel

```
 ?P(fapp(?f1.1) ⊑ fapp(?f2.1))⟹?P(?f1.1 ⊑ ?f2.1)
- by (rtac (less_cfun RS ssubst) 1);

 1. ⋀i. is_chain(CCF)⟹
        fapp(CCF(i)) ⊑ fapp(Λx. lub(range(λi. CCF(i)[x])))
```

Erinnern wir uns, daß `Λx.t(x)` nur die Mixfix-Schreibweise für `fabs(λx.t(x))` ist. Die Regel `Abs_Cfun_inverse2` erlaubt uns, auf der rechten Seite den Präfix `fapp(fabs(_))` zu eliminieren. Wir verwenden das Theorem `Abs_Cfun_inverse2` in Kombination mit Gleichungslogik

```
 ⟦ contX(?s); ?P(?s)⟧⟹?P(fapp(fabs(?s)))
- by (rtac (Abs_Cfun_inverse2 RS ssubst) 1);
```

und erhalten zwei neue Teilziele, von denen das erste der Preis für die Anwendung des Theorems `Abs_Cfun_inverse2` ist.

```
 1. ⋀i. is_chain(CCF)⟹contX(λu. lub(range(λi. CCF(i)[u])))
 2. ⋀i. is_chain(CCF)⟹fapp(CCF(i)) ⊑ (λu. lub(range(λi. CCF(i)[u])))
```

Das erste Teilziel wird vollständig durch Anwendung des Theorems `contX_lubcfun` gelöst, und wir erhalten:

```
- by (etac contX_lubcfun 1);

 1. ⋀i. is_chain(CCF)⟹fapp(CCF(i)) ⊑ (λu. lub(range(λi. CCF(i)[u])))
```

Links vom Ordnungssymbol $\sqsubseteq$ steht ein Element einer Kette von Funktionen. Erinnern wir uns an das Theorem `lub_fun`, das die kleinste obere Schranke einer Kette von Funktionen charakterisiert:

```
 is_chain(?S)==>range(?S) <<| (λx. lub(range(λi. ?S(i, x))))
```

Kleinste obere Schranken sind aber insbesondere obere Schranken, und eine obere Schranke ist stärker als jedes Glied der Kette. Eine Kombination der Theoreme:

```
lub_fun     is_chain(?S)==>range(?S) <<| (λx. lub(range(λi. ?S(i, x))))
is_lubE     ?S <<| ?x==>?S <| ?x ∧ (∀u. ?S <| u → ?x ⊑ u)
conjunct1   ?P ∧ ?Q==>?P
ub_rangeE   range(?S) <| ?x==>∀i. ?S(i) ⊑ ?x
spec        ∀x. ?P(x)==>?P(?x)
```

liefert das zusammengesetzte Theorem:

```
 is_chain(?S1)==>?S1(?x) ⊑ (λx. lub(range(λi. ?S1(i, x))))
- by (rtac (lub_fun RS is_lubE RS conjunct1 RS ub_rangeE RS spec) 1);
```

Durch Anwendung dieses Theorems erhalten wir das neue Teilziel:

```
 1. ⋀i. is_chain(CCF)==>is_chain(λu. fapp(CCF(u)))
```

Dieses ist aber leicht mit Hilfe der Monotonie der Applikationsfunktion zu zeigen.

```
 is_chain(?Y)==>is_chain(λi. fapp(?Y(i)))
- by (etac (monofun_fapp1 RS ch2ch_monofun) 1);
```

Wir haben zu Beginn des Beweises das folgende Teilziel zurückgestellt:

```
 1. is_chain(CCF)==>
    ∀u. range(CCF) <| u → (Λx. lub(range(λi. CCF(i)[x]))) ⊑ u
```

Der nächste Schritt zur Lösung dieses Teilziels ist kanonisch durch die Methodik des natürlichen Schließens vorgegeben. Wir verschaffen uns den nötigen Annahmenkontext und haben zu zeigen:

```
- by (strip_tac 1);

 1. ⋀u. [| is_chain(CCF); range(CCF) <| u|]==>
         (Λx. lub(range(λi. CCF(i)[x]))) ⊑ u
```

Wir müssen nachweisen, daß zwei Operationen in Ordnungsrelation stehen. Dazu verwenden wir, wie im ersten Teil des Beweises, die Regel

```
 ?P(fapp(?f1.1) ⊑ fapp(?f2.1))==>?P(?f1.1 ⊑ ?f2.1)
- by (rtac (less_cfun RS ssubst) 1);

 1. ⋀u. [| is_chain(CCF); range(CCF) <| u|]==>
         fapp(Λx. lub(range(λi. CCF(i)[x]))) ⊑ fapp(u)
```

und anschließend

```
 [| contX(?s); ?P(?s)|]==>?P(fapp(fabs(?s)))
- by (rtac (Abs_Cfun_inverse2 RS ssubst) 1);
```

um den Präfix `fapp(fabs(_))` zu eliminieren. Dies führt zu den neuen Teilzielen:

```
 1. ⋀u. [| is_chain(CCF); range(CCF) <| u|]==>
        contX(λu. lub(range(λi. CCF(i)[u])))
 2. ⋀u. [| is_chain(CCF); range(CCF) <| u|]==>
        (λu. lub(range(λi. CCF(i)[u]))) ⊑ fapp(u)
```

Das erste wird wieder vollständig durch das Theorem

```
 is_chain(?F)==>contX(λx. lub(range(λj. ?F(j)[x])))
- by (etac contX_lubcfun 1);
```

gelöst. Es verbleibt das Teilziel:

```
 1. ⋀u. [| is_chain(CCF); range(CCF) <| u|]==>
        (λu. lub(range(λi. CCF(i)[u]))) ⊑ fapp(u)
```

Diesmal benützen wir die Tatsache, daß kleinste obere Schranken schwächer sind als jede andere obere Schranke. Durch Kombination der Regeln

```
lub_fun     is_chain(?S)==>range(?S) <<| (λx. lub(range(λi. ?S(i, x))))
is_lubE     ?S <<| ?x==>?S <| ?x ∧ (∀u. ?S <| u → ?x ⊑ u)
conjunct2   ?P ∧ ?Q==>?Q
spec        ∀x. ?P(x)==>?P(?x)
mp          [| ?P → ?Q; ?P|]==>?Q
```

erhalten wir die zusammengesetzte Regel

```
   [| is_chain(?S4); range(?S4) <| ?x1|]==>
   (λx. lub(range(λi. ?S4(i, x)))) ⊑ ?x1
- by (rtac (lub_fun RS is_lubE RS conjunct2 RS spec RS mp) 1);
```

deren Anwendung die beiden folgenden Teilziele ergibt:

```
 1. ⋀u. [| is_chain(CCF); range(CCF) <| u|]==>is_chain(λi. fapp(CCF(i)))
 2. ⋀u. [| is_chain(CCF); range(CCF) <| u|]==>
        range(λi. fapp(CCF(i))) <| fapp(u)
```

Beide Ziele werden im wesentlichen über die Monotonie der Applikationsfunktion `fapp` gelöst. Das erste lösen wir mittels

```
 is_chain(?Y)==>is_chain(λi. fapp(?Y(i)))
- by (etac (monofun_fapp1 RS ch2ch_monofun) 1);
```

das zweite mittels

```
 range(?Y) <| ?u==>range(λi. fapp(?Y(i))) <| fapp(?u)
- by (etac (monofun_fapp1 RS ub2ub_monofun) 1);
No subgoals
```

4.7.6 Die Theorie `Cfun3`

Die Theorie `Cfun3` ist in Abbildung 4.25 dargestellt. Sie führt die Arität → ::(pcpo,pcpo)pcpo ein, was durch die Theoreme `minimal_cfun` und `cpo_cfun` gerechtfertigt wird.

Darüberhinaus werden noch die beiden Konstanten `Istrictify` und `strictify` definiert. Die

Definitionen sind so ausgeführt, daß für jede Operation `f` die strikte Variante der Operation durch `strictify[f]` bezeichnet wird.

```
Cfun3 = Cfun2 +

arities   →       :: (pcpo,pcpo)pcpo

consts
        Istrictify    ::  (α → β) ⇒ α ⇒ β
        strictify     ::  (α → β) → α → β

rules

inst_cfun_pcpo    ⊥::α → β = ⊥_cfun

Istrictify_def    Istrictify(f,x) ≡ (εz.
                              ( x=⊥ → z = ⊥)
                            ∧ (¬x=⊥ → z = f[x]))

strictify_def     strictify ≡ (Λf x.Istrictify(f,x))

end
```

Abbildung 4.25: Theorie `Cfun3`

Am Beispiel der Funktion `strictify` zeigt sich ein Muster, das uns noch mehrmals bei der konservativen Einführung von Operationen begegnen wird. Im ersten Schritt wird mit den Mitteln der Logik höherer Stufe die Funktion `Istrictify` definiert[9]. Im zweiten Schritt leitet man die wesentlichen Eigenschaften der Funktion ab. Speziell versucht man zu zeigen, daß sie stetig ist, was ein nichtriviales Unterfangen ist. Erst wenn der Stetigkeitsbeweis gelingt, können die Eigenschaften der Funktion `Istrictify` mittels β-Reduktion auf die Operation `strictify` übertragen werden. Gelingt der Beweis der Stetigkeit aber nicht, so ist die Definition der Operation `strictify` wertlos, da die Eigenschaften der Funktion `Istrictify` nicht ausgenützt werden können. Die Einführung der internen Konstanten, hier im Beispiel `Istrictify`, ist zur Definition einer Operation nicht unbedingt erforderlich, ist aber als notationelle Abkürzung empfehlenswert.

4.7.7 Theoreme der Theorie `Cfun3`

Die Theoreme der Theorie `Cfun3` sind in den Abbildungen 4.26 und 4.29 abgebildet.

Die Theoreme `contlub_fapp1` bis `contX_cfun_fun` betreffen die Stetigkeit der Applikationsfunktion `fapp` im ersten Argument. Dabei ist zum Beispiel das Theorem `contlub_fapp1` logisch gleichwertig zum Theorem `contlub_cfun_fun`, mit dem Unterschied, daß im letzteren die Mixfix-Schreibweise benutzt wird.

[9] der Präfix `I` steht für Intern.

Die Theoreme `contlub_cfun` und `contX_cfun` beschreiben die Stetigkeit der Applikationsfunktion `fapp` in beiden Argumenten gleichzeitig. Für das Rechnen in Gleichungsketten sind vor allem die Theoreme `contlub_cfun_fun` und `contlub_cfun` geeignet. Das dazu passende Theorem `contlub_cfun_arg` wurde bereits in der Theorie `Cfun2` bewiesen.

```
contlub_fapp1      contlub(fapp)
contX_fapp1        contX(fapp)

contlub_cfun_fun   is_chain(FY)
                     ⟹lub(range(FY))[x] = lub(range(λi.FY(i)[x]))
contX_cfun_fun     is_chain(FY)
                     ⟹range(λi.FY(i)[x]) ≪| lub(range(FY))[x]

contlub_cfun       [| is_chain(FY);is_chain(TY)|]
                     ⟹lub(range(FY))[lub(range(TY))] =
                        lub(range(λi.FY(i)[TY(i)]))

contX_cfun         [| is_chain(FY);is_chain(TY)|]
                     ⟹range(λi.FY(i)[TY(i)]) ≪|
                        lub(range(FY))[lub(range(TY))]


contX2contX_fapp   [| contX(λx.ft(x));contX(λx.tt(x))|]
                     ⟹contX(λx.(ft(x))[tt(x)])

contX2mono_Λ       [| ∀x.contX(c(x)); ∀y.monofun(λx.c(x,y))|]
                     ⟹monofun(λx.Λy.c(x,y))

contX2contX_Λ      [| ∀x.contX(c(x)); ∀y.contX(λx.c(x,y))|]
                     ⟹contX(λx.Λy.c(x,y))


contX2contX_Λ2     [| ⋀x.contX(c(x)); ⋀y.contX(λx.c(x,y))|]
                     ⟹contX(λx.Λy.c(x,y))
```

Abbildung 4.26: Theoreme der Theorie `Cfun3` - Teil 1

Die Theoreme `contX2contX_fapp` und `contX2contX_Λ2` sind von zentraler Bedeutung für das Arbeiten mit Operationen. Sie erlauben es, aus der Stetigkeit von Teiltermen auf die Stetigkeit von Termen zu schließen, die mittels Applikation und Abstraktion für Operationen gebildet sind. Hier handelt es sich um die bekannten Theoreme für stetige Funktionen. Im Unterschied zur üblichen Darstellung werden in der Formalisierung durch Logik höherer Stufe die freien Variablen im Term durch die λ-Abstraktion für Funktionen gebunden. Statt 'Term `t` ist stetig in der freien Variablen `x`' verwendet man 'die Funktion λ`x.t(x)` ist stetig'. Bei der Anwendung dieser Theoreme werden die Funktionsvariablen `ft`, `tt` und `c` durch Unifikation höherer Stufe

mit den konkreten Termen unifiziert. An dieser Stelle sollte noch erwähnt werden, daß der Term `contX(c(x))` η-äquivalent zu `contX(`λ`y.c(x,y))` ist. Isabelle überführt alle Terme automatisch in η-Normalform, und somit wird der Term `contX(`λ`y.c(x,y))` automatisch als `contX(c(x))` dargestellt. In der nicht-normalisierten Form wäre das Theorem intuitiv klarer.

Einen Term, der ausschließlich durch Anwendung der Applikation und Abstraktion für Operationen aus Konstanten und Variablen entsteht, werde ich im folgenden als *LCF-Term* bezeichnen, da Terme dieser Bauart auch in der Logik LCF gebildet werden können. Die Stetigkeit eines LCF-Terms in einer freien Variablen kann allein durch die Anwendung der Theoreme `contX2contX_fapp`, `contX2contX_`Λ`2` und `contX_const`, `contX_id` gezeigt werden. Die Theoreme `contX_const` und `contX_id` wurden bereits in Theorie `Cont` bewiesen. Als Beispiel sei der Term Λ`z.(`Λ`x.f[x])[z[y]]` mit der freien Variablen `y` genannt.

Um den Beweis der Stetigkeit für LCF-Terme in Isabelle zu automatisieren, genügen die Vereinbarungen, die in Abbildung 4.27 dargestellt sind. Es handelt sich dabei um die konkrete Eingabe für das Isabelle-System.

```
val contX_lemmas = [contX_const, contX_id, contX_fapp2,
                    contX2contX_fapp,contX2contX_Λ2];
val contX_tac    = (fn i ⇒ (resolve_tac contX_lemmas i));
val contX_tacR   = (fn i ⇒ (REPEAT (contX_tac i)));
```

Abbildung 4.27: Beweistaktik für die Stetigkeit von LCF-Termen

Die oben genannten Theoreme werden in `contX_lemmas` zu einer Liste von Theoremen zusammengefaßt. Das Theorem `contX_fapp2` wurde nur aus Effizienzgründen hinzugefügt. Die nächsten beiden Vereinbarungen programmieren zwei einfache Taktiken. Durch Anwendung von `contX_tac` wird versucht, im Beweisziel eines der Stetigkeitstheoreme anzuwenden. Bei Verwendung der Taktik `contX_tacR` werden die Stetigkeitstheoreme auf das Beweisziel und alle neu entstehenden Teilziele wiederholt (`REPEAT`) angewendet, bis die Anwendung fehlschlägt, oder keine Ziele mehr zu beweisen sind. Wenn der Term `t(x)` ein LCF-Term ist, so wird das Beweisziel `contX(`λ`x.t(x))` durch die Taktik `contX_tacR` stets automatisch gelöst.

Das Beweisskript, das die Stetigkeit des vorherigen Beispielterms in der freien Variablen `y` beweist, ist in Abbildung 4.28 dargestellt.

```
goal Cfun3.thy  "contX(λy.Λz.(Λx.f[x])[z[y]])";
by (contX_tacR 1);
No subgoals
```

Abbildung 4.28: Stetigkeitsbeweis fur einen LCF-Term

Die Abbildung 4.29 zeigt die restlichen Theoreme der Theorie `Cfun3`. Das Theorem `strict_fapp1` drückt die Striktheit der Applikationsfunktion `fapp` im ersten Argument in Mixfix-Schreibweise aus.

Die restlichen Theoreme zeigen am Beispiel der Operation `strictify`, wie die Reduktionseigenschaften einer konservativ eingeführten Operation schrittweise abgeleitet werden. Zuerst zeigt man die Reduktionseigenschaften der zugrundeliegenden Funktion. Dann leitet man die Monotonie und Stetigkeit der Funktion in allen Argumenten ab. Dies erlaubt in einem letzten Schritt, die Reduktionseigenschaften der Funktion auf die Operation zu übertragen.

`strict_fapp1`	`⊥[x] = ⊥`
`Istrictify1`	`Istrictify(f)(⊥)=⊥`
`Istrictify2`	`¬x=⊥⟹Istrictify(f)(x)=f[x]`
`monofun_Istrictify1`	`monofun(Istrictify)`
`monofun_Istrictify2`	`monofun(Istrictify(f))`
`contlub_Istrictify1`	`contlub(Istrictify)`
`contlub_Istrictify2`	`contlub(Istrictify(f))`
`contX_Istrictify1`	`contX(Istrictify)`
`contX_Istrictify2`	`contX(Istrictify(f))`
`strictify1`	`strictify[f][⊥]=⊥`
`strictify2`	`¬x=⊥⟹strictify[f][x]=f[x]`

Abbildung 4.29: Theoreme der Theorie `Cfun3` - Teil 2

Mit der Ableitung der Theoreme für die Propagierung der Stetigkeit und der Programmierung der entsprechenden Taktiken `contX_tac` und `contX_tacR` ergibt sich nun auch die Möglichkeit, das Isabelle-Werkzeug zur Termsimplifikation verstärkt einzusetzen. Abbildung 4.30 zeigt den ersten Schritt, mit dem der Simplifikator an die entstehende Logik HOLCF angepaßt wird. Die Anpassung des Simplifikators wird hier nur dargestellt, um einen Eindruck vom Umgang mit dem Isabelle-System zu vermitteln. Eine genaue Beschreibung des Simplifikators findet sich in [Pau94].

Die Liste `Cfun_rews` enthält eine Ansammlung sinnvoller Termersetzungsregeln, zu denen vor allem die β-Reduktion für Operationen gehört. Der Simplifikator arbeitet mit Termersetzungsregeln bzgl. der Metagleicheit $\equiv$. Theoreme der Form `t1(x)=t2(x)` werden in `t1(x)` $\equiv$ `t2(x)` umgebaut, und Theoreme der Form `p` werden in `p` $\equiv$ `True` umgebaut. Daneben gibt es noch weitere raffinierte Mechanismen, mit denen Theoreme automatisch in Simplifikationsregeln umgeformt werden [Pau94]. Die Termersetzung ersetzt die Terme auf der linken Seite durch die Terme auf der rechten Seite des $\equiv$ Zeichens. Da die Anwendung der β-Reduktion stets einen Stetigkeitsbeweis als Teilziel erzeugt, wird `contX_tac` als automatische Taktik zur Lösung solcher Ziele mittels `setsolver` eingebaut.

Im Gegensatz zur Programmierung der einfachen Taktiken `contX_tac` und `contX_tacR` gehört die Anpassung des Simplifikators bereits zu den delikateren Eingriffen in das Isabelle-System. Neben einem tieferen Einblick in die Arbeitsweise des Simplifikators ist auch ein gutes Verständnis des gesamten Isabelle-Systems erforderlich, damit der umgebaute Simplifikator die gewünschte Leistung erbringt.

```
val Cfun_rews  = [minimal,refl_less,beta_cfun,strict_fapp1,strictify1,
                  strictify2];

val Cfun_ss = HOL_ss
        addsimps  Cfun_rews
        setsolver
        (fn thms => (resolve_tac (TrueI::refl::thms)) ORELSE' atac ORELSE'
                    (fn i => DEPTH_SOLVE_1 (contX_tac i))
        );
```

Abbildung 4.30: Erste Anpassung des Simplifikators

4.7.8 Beweise für ausgesuchte Theoreme

In diesem Abschnitt präsentiere ich die Beweise für die drei Theoreme `contlub_fapp1`, `contX2contX_fapp` und `contX2contX_Λ`. Diese Theoreme sind wohlbekannt, werden in der Literatur aber, wenn überhaupt, nur sehr oberflächlich gezeigt. Dies liegt wohl vor allem daran, daß in der Literatur die Stetigkeit von Termen in freien Variablen nicht exakt genug formalisiert ist, und deswegen Beweise nur skizzenhaft geführt werden können.

4.7.8.1 Beweis für das Theorem `contlub_fapp1`

Wir beginnen, indem wir uns das Beweisziel vorgeben.

```
val prems = goal Cfun3.thy  "contlub(fapp)";

 1. contlub(fapp)
```

Im ersten Schritt expandieren wir die Definition des Prädikats `contlub`, und erhalten so:

```
- by (rtac contlubI 1);

 1. ∀Y. is_chain(Y) → fapp(lub(range(Y))) = lub(range(λi. fapp(Y(i))))
```

Gemäß der Methodik des natürlichen Schließens beweisen wir die Behauptung für ein beliebiges, aber festes `Y` und fügen die Prämisse der Implikation zu unseren Annahmen hinzu. Dies führt zu:

```
- by (strip_tac 1);

 1. ⋀Y. is_chain(Y)⟹fapp(lub(range(Y))) = lub(range(λi. fapp(Y(i))))
```

Wir müssen die Gleichheit zweier Funktionen zeigen. Argumentation über die extensionale Gleichheit von Funktionen mittels

```
 ∀x. ?f1(x) = ?g1(x)⟹?f1 = ?g1
- by (rtac (expand_fun_eq RS iffD2) 1);
```

liefert:

```
 1. ⋀Y. is_chain(Y)⟹∀x. lub(range(Y))[x] = lub(range(λi. fapp(Y(i))), x)
```

Es reicht, die Behauptung für ein beliebiges, aber festes `x` zu zeigen. Wir erhalten:

```
- by (strip_tac 1);

 1. ⋀Y x. is_chain(Y)⟹lub(range(Y))[x] = lub(range(λi. fapp(Y(i))), x)
```

Auf der linken Seite wird der Grenzwert einer Kette von Operationen auf `x` angewendet. Durch Anwendung des Theorems `thelub_cfun`

```
 is_chain(?CCF1)⟹
 lub(range(?CCF1)) = (Λx. lub(range(λi. ?CCF1(i)[x])))
```

und Gleichungslogik erhalten wir:

```
- by (rtac (thelub_cfun RS ssubst) 1);

 1. ⋀Y x. is_chain(Y)⟹is_chain(Y)
 2. ⋀Y x. is_chain(Y)⟹
          (Λx. lub(range(λi. Y(i)[x])))[x] = lub(range(λi. fapp(Y(i))), x)
```

Das erste Ziel lösen wir trivial per Annahme. Im zweiten Ziel wenden wir auf der linken Seite β-Reduktion für Operationen an. Mittels

```
- by (atac 1);
- by (rtac (beta_cfun RS ssubst) 1);
```

erhalten wir:

```
 1. ⋀Y x. is_chain(Y)⟹contX(λx. lub(range(λi. Y(i)[x])))
 2. ⋀Y x. is_chain(Y)⟹
          lub(range(λi. Y(i)[x])) = lub(range(λi. fapp(Y(i))), x)
```

Das erste Ziel wird vollständig durch das Theorem

```
 is_chain(?F)⟹contX(λx. lub(range(λj. ?F(j)[x])))
- by (etac contX_lubcfun 1);
```

gelöst. Es verbleibt das Teilziel:

```
 1. ⋀Y x. is_chain(Y)⟹
          lub(range(λi. Y(i)[x])) = lub(range(λi. fapp(Y(i))), x)
```

Auf der rechten Seite wird der Grenzwert einer Kette von Funktionen auf `x` angewendet. Verwendung des Theorems `thelub_fun`

```
 is_chain(?S1)⟹lub(range(?S1)) = (λx. lub(range(λi. ?S1(i, x))))
- by (rtac (thelub_fun RS ssubst) 1);
```

und Gleichungslogik liefert:

```
 1. ⋀Y x. is_chain(Y)⟹is_chain(λi. fapp(Y(i)))
 2. ⋀Y x. is_chain(Y)⟹lub(range(λi. Y(i)[x])) = lub(range(λi. Y(i)[x]))
```

Das erste Teilziel lösen wir über die Monotonie der Applikationsfunktion für Operationen,

das zweite wird trivial per Reflexivität gelöst.

```
- by (etac (monofun_fapp1 RS ch2ch_monofun) 1);
- by (rtac refl 1);
No subgoals
```

4.7.8.2 Beweis für das Theorem contX2contX_fapp

Wir erzeugen das initiale Beweisziel mittels

```
- val prems = goal Cfun3.thy
        "[| contX(λx.ft(x));contX(λx.tt(x))|]==>contX(λx.(ft(x))[tt(x)])";
- by (cut_facts_tac prems 1);
```

und erhalten:

```
 1. [| contX(ft); contX(tt)|]==>contX(λx. ft(x)[tt(x)])
```

In der Theorie Cont haben wir schon das Theorem contX2contX_app2

```
 [| contX(?ft);⋀x.contX(?ft(x));contX(?tt)|]==>contX(λx. ?ft(x, ?tt(x)))
```

bewiesen. Wir wenden das Theorem an, wobei erst der zweite vom System vorgeschlagene Unifikator das erwünschte Ergebnis liefert. Wir erhalten:

```
- by (rtac contX2contX_app2 1);
- back();

 1. [| contX(ft); contX(tt)|]==>contX(λu. fapp(ft(u)))
 2. ⋀x. [| contX(ft); contX(tt)|]==>contX(fapp(ft(x)))
 3. [| contX(ft); contX(tt)|]==>contX(tt)
```

Das erste Teilziel verrät uns, daß die Unifikationsvariable ?ft des Theorems mit dem Term λu. fapp(ft(u)) unifiziert wurde. Wir wenden erneut das Theorem contX2contX_app2 auf das erste Teilziel an, und erhalten diesmal mit Unifikation von ?ft und λu. fapp folgenden Beweiszustand:

```
- by (rtac contX2contX_app2 1);

 1. [| contX(ft); contX(tt)|]==>contX(λu. fapp)
 2. ⋀u. [| contX(ft); contX(tt)|]==>contX(fapp)
 3. [| contX(ft); contX(tt)|]==>contX(ft)
 4. ⋀x. [| contX(ft); contX(tt)|]==>contX(fapp(ft(x)))
 5. [| contX(ft); contX(tt)|]==>contX(tt)
```

Alle fünf Teilziele sind einfach zu lösen. Ziel 1 lösen wir durch die Stetigkeit von Konstanten. Die Ziele 3 und 5 werden trivial per Annahme gelöst. Die Ziele 2 und 4 werden durch die Stetigkeit der Applikationsfunktion im ersten und im zweiten Argument gelöst.

```
- by (rtac contX_const 1);
- by (rtac contX_fapp1 1);
- by (atac 1);
```

```
- by (rtac contX_fapp2 1);
- by (atac 1);
No subgoals
```

4.7.8.3 Beweis für das Theorem contX2contX_Λ

Wir beginnen, indem wir das initiale Beweisziel erzeugen.

```
- val prems = goal Cfun3.thy
   "[| ∀x.contX(c1(x)); ∀y.contX(λx.c1(x,y))|]==>contX(λx. Λy. c1(x,y))";
- by (cut_facts_tac prems 1);

 1. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y))|]==>
    contX(λx. Λy. c1(x, y))
```

Zuerst führen wir den Nachweis der Stetigkeit auf die Beweise für Monotonie und Vertauschbarkeit des Grenzwerts zurück.

```
 [| monofun(?f); contlub(?f)|]==>contX(?f)
- by (rtac monocontlub2contX 1);
```

Wir erhalten die beiden Teilziele:

```
 1. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y))|]==>
    monofun(λx. Λy. c1(x, y))
 2. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y))|]==>
    contlub(λx. Λy. c1(x, y))
```

Das erste Teilziel lösen wir durch das Hilfstheorem contX2mono_Λ

```
 [| ∀x. contX(?c1(x)); ∀y. monofun(λx. ?c1(x, y))|]==>
   monofun(λx. Λy. ?c1(x, y))
```

und Verwendung der Stetigkeitsannahmen im Kontext. Anwendung des Skripts

```
- by (etac contX2mono_Λ 1);
- by (rtac (contX2mono RS allI) 1);
- by (etac spec 1);
```

löst das erste Teilziel vollständig. Es verbleibt:

```
 1. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y))|]==>
    contlub(λx. Λy. c1(x, y))
```

Wir expandieren die Definition des Prädikats contlub und erhalten:

```
- by (rtac contlubI 1);

 1. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y))|]==>
    ∀Y. is_chain(Y) →
         (Λy. c1(lub(range(Y)), y)) = lub(range(λi. Λy. c1(Y(i), y)))
```

Wir beweisen die Behauptung für ein beliebiges, aber festes `Y` und übernehmen die Prämisse der Implikation in den Annahmenkontext.

```
- by (strip_tac 1);

 1. ⋀Y. [[ ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)]]⟹
         (Λy. c1(lub(range(Y)), y)) = lub(range(λi. Λy. c1(Y(i), y)))
```

Auf der rechten Seite der Gleichung steht der Grenzwert einer Kette von Operationen. Anwendung des Theorems `thelub_cfun`

```
 is_chain(?CCF1)⟹
 lub(range(?CCF1)) = (Λx. lub(range(λi. ?CCF1(i)[x])))
```

und Gleichungslogik liefert:

```
- by (rtac (thelub_cfun RS ssubst) 1);

 1. ⋀Y. [[ ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)]]⟹
         is_chain(λi. Λy. c1(Y(i), y))
 2. ⋀Y. [[ ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)]]⟹
         (Λy. c1(lub(range(Y)), y)) =
         (Λx. lub(range(λi. (Λy. c1(Y(i), y))[x])))
```

Das erste Ziel lösen wir durch erneute Anwendung des Hilfstheorems `contX2mono_Λ` und Standardargumentation über die Propagierung von Ketten unter Verwendung der Annahmen im Kontext. Durch Anwendung des folgenden Skripts wird das erste Teilziel vollständig gelöst.

```
- by (rtac (contX2mono_Λ RS ch2ch_monofun) 1);
- by (atac 1);
- by (rtac (contX2mono RS allI) 1);
- by (etac spec 1);
- by (atac 1);
```

Es verbleibt das Teilziel:

```
 1. ⋀Y. [[ ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)]]⟹
         (Λy. c1(lub(range(Y)), y)) =
         (Λx. lub(range(λi. (Λy. c1(Y(i), y))[x])))
```

Wir müssen nun die Gleichheit zweier Abstraktionen für Operationen zeigen. Erinnern wir uns, daß `Λx.t(x)` eine Abkürzung für `fabs(λx.t(x))` ist. Auf beiden Seiten der Gleichung wird also die Funktion `fabs` angewendet. Mit Hilfe des Kongruenztheorems

```
 ?x = ?y⟹?f(?x) = ?f(?y)
- by (res_inst_tac [("f","fabs")] arg_cong 1);
```

reduzieren wir zu:

```
 1. ⋀Y. [[ ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)]]⟹
         c1(lub(range(Y))) = (λx. lub(range(λi. (Λy. c1(Y(i), y))[x])))
```

Eine weitere Argumentation über die extensionale Gleichheit von Funktionen liefert dann:

```
- by (rtac ext 1);
```

```
 1. ⋀Y y. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)|]⟹
           c1(lub(range(Y)), y) = lub(range(λi. (Λy. c1(Y(i), y))[y]))
```

Als nächstes führen wir in einem Teilterm der rechten Seite β-Reduktion für Operationen durch.

```
- by (rtac (beta_cfun RS ext RS ssubst) 1);

 1. ⋀Y y x.
       [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)|]⟹
       contX(c1(x))
 2. ⋀Y y. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)|]⟹
           c1(lub(range(Y)), y) = lub(range(λi. c1(Y(i), y)))
```

Das erste Teilziel lösen wir trivial durch die Annahme `contX(c1(x))`. Es verbleibt:

```
- by (etac spec 1);

 1. ⋀Y y. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)|]⟹
           c1(lub(range(Y)), y) = lub(range(λi. c1(Y(i), y)))
```

Nun bringen wir die zweite Annahme `∀y.contX(λx. c1(x, y))` ins Spiel. Mittels

```
[| contX(?f2); is_chain(?x1)|]⟹
   ?f2(lub(range(?x1))) = lub(range(λi. ?f2(?x1(i))))

- by (rtac (contX2contlub RS contlubE RS spec RS mp ) 1);
```

reduzieren wir zu:

```
 1. ⋀Y y. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)|]⟹
           contX(λu. c1(u, y))
 2. ⋀Y y. [| ∀x. contX(c1(x)); ∀y. contX(λx. c1(x, y)); is_chain(Y)|]⟹
           is_chain(Y)
```

Diese beiden Teilziele lösen wir trivial mit Hilfe des Annahmenkontextes.

```
- by (etac spec 1);
- by (atac 1);
No subgoals
```

4.8 Theorien für das strikte Produkt

In diesem Abschnitt führe ich den Konstruktor ** für das strikte Produkt zweier Bereichstypen ein. Ähnlich wie bei der konservativen Einführung des Konstruktors $\rightarrow$ für Operationen werden hierfür mehrere Teilschritte benötigt.

Am Beispiel des strikten Produkts werde ich die konservative Einführung eines Typkonstruk-

tors nochmals ausführlich behandeln. Die Einführung der anderen Typkonstruktoren, kartesisches Produkt * in Abschnitt 4.9, strikte Summe ++ in Abschnitt 4.10 und Lifting (_)u in Abschnitt 4.11, werde ich dagegen knapper halten, da sie durch ein jeweils analoges Vorgehen bewerkstelligt wurde.

4.8.1 Die Theorie Sprod0

Zunächst wird der Typkonstruktor ** gemäß der Methode für konservative Erweiterung mit der Arität (pcpo,pcpo)term eingeführt. Die Klasse pcpo auf Argumentposition bringt zum Ausdruck, daß als Argumente des Typkonstruktors nur Bereichstypen zugelassen sind. Die Beschränkung auf pcpo-Typen ist notwendig, da sonst die Striktheit des Produkts in der Kodierung der Repräsentanten nicht ausgedrückt werden könnte. Die Ordnungsinformation der Argumente wird später auch dazu verwendet, um eine pcpo-Struktur für das strikte Produkt zu definieren. Man beachte, daß ohne die Verwendung von Typklassen die Definition des Konstruktors ** für das strikte Produkt gar nicht möglich wäre.

Der Typkonstruktor * für das kartesische Produkt mit Arität (term,term)term, siehe Abbildungen 3.5 und 3.6 in Kapitel 3, konnte für die Kodierung der Repräsentation nicht gewinnbringend verwendet werden. Die direkte Kodierung der Repräsentanten war einfacher. Die Theorie ist in Abbildung 4.31 dargestellt.

```
Sprod0 = Cfun3 +

types    **  2  (infixr 20)
arities  **  :: (pcpo,pcpo)term

consts
  Sprod          ::  (α ⇒ β ⇒ bool)set
  Spair_Rep      ::  [α,β] ⇒ [α,β] ⇒ bool
  Rep_Sprod      ::  (α ** β) ⇒ (α ⇒ β ⇒ bool)
  Abs_Sprod      ::  (α ⇒ β ⇒ bool) ⇒ (α ** β)
  Ispair         ::  [α,β] ⇒ (α ** β)
  Isfst          ::  (α ** β) ⇒ α
  Issnd          ::  (α ** β) ⇒ β

rules

  Spair_Rep_def          Spair_Rep ≡ (λa b. λx y.
                                 (¬a=⊥ ∧ ¬b=⊥ → x=a ∧ y=b ))

  Sprod_def              Sprod ≡ {f. ∃a b. f = Spair_Rep(a,b)}

  Rep_Sprod              Rep_Sprod(p)∈Sprod
  Rep_Sprod_inverse      Abs_Sprod(Rep_Sprod(p)) = p
  Abs_Sprod_inverse      f∈Sprod⟹Rep_Sprod(Abs_Sprod(f)) = f
```

```
  Ispair_def     Ispair(a,b) ≡ Abs_Sprod(Spair_Rep(a,b))
  Isfst_def      Isfst(p) ≡ εz.
                                        (p=Ispair(⊥,⊥) → z=⊥)
                 ∧ (∀a b. ¬a=⊥ ∧ ¬b=⊥ ∧ p=Ispair(a,b) → z=a)

  Issnd_def      Issnd(p) ≡ εz.
                                        (p=Ispair(⊥,⊥) → z=⊥)
                 ∧ (∀a b. ¬a=⊥ ∧ ¬b=⊥ ∧ p=Ispair(a,b) → z=b)

end
```

Abbildung 4.31: Theorie `Sprod0`

Die Menge `Sprod` ist die Menge der Repräsentanten für Paare. Als Repräsentanten werden zweistellige Prädikate über den Argumenttypen $\alpha_{\texttt{pcpo}}$ und $\beta_{\texttt{pcpo}}$ verwendet. Die explizite Angabe der Klasse `pcpo` kann wegen der `default`-Einstellung entfallen, die in der Theorie `Cont` vorgenommen wurde. Die Definition der Repräsentantenmenge wird durch Axiom `Sprod_def` vorgenommen. Dazu wird die Hilfsfunktion `Spair_Rep` verwendet, die in Axiom `Spair_Rep_def` definiert wird. Die Kodierung der Paare orientiert sich an der üblichen Kodierung [GM93, Pau94] für das kartesische Produkt, die für das strikte Produkt angepaßt wurde. Die Theoreme zur Theorie `Sprod0` werden zeigen, daß die vorliegende Kodierung wirklich das strikte Produkt repräsentiert.

Es ist leicht einzusehen, daß die Repräsentantenmenge nicht leer ist, da `Spair_Rep(a,b)` für beliebiges `a` und `b` in `Sprod` liegt. Somit ist die Konservativität der Typdefinition sichergestellt, die durch die drei Axiome `Rep_Sprod`, `Rep_Sprod_inverse` und `Abs_Sprod_inverse` nach dem üblichen Schema erfolgt.

Zuletzt werden noch interne Konstanten definiert. `Ispair` ist die Konstruktorfunktion für strikte Paare, und `Isfst` und `Issnd` sind die Destruktoren[10]. Die entsprechenden Operationen `spair`, `sfst` und `ssnd` werden erst in Theorie `Sprod3` eingeführt, wenn genügend Eigenschaften für die internen Funktionen gezeigt wurden.

4.8.2 Theoreme der Theorie `Sprod0`

Die Theoreme der Theorie `Sprod0` sind in Abbildung 4.32 dargestellt. Sie beschreiben die wichtigen Eigenschaften des Typs der strikten Produkte.

Als allererstes erfolgt mittels `SprodI` wieder die Rechtfertigung für die Typdefinition. Die Theoreme `inj_onto_Abs_Sprod` bis `inject_Spair_Rep` beschreiben Eigenschaften der Repräsentations- und Abstraktionsfunktion. Im Beweis für dieses Theorem muß noch mit der kryptischen Kodierung für die Paare hantiert werden, danach kann man davon abstrahieren.

Die Theoreme `inject_Ispair` bis `defined_Ispair` beschreiben Eigenschaften der Konstruktorfunktion `Ispair`. Danach folgen die Ausschöpfungseigenschaft für strikte Paare `Exh_Sprod` und das entsprechende Eliminationstheorem `IsprodE`.

[10] `Ispair` steht für internes striktes Paaren, `Isfst` steht für interne strikte Selektion der ersten Komponente, `Issnd` steht für interne strikte Selektion der zweiten Komponente.

```
SprodI                  Spair_Rep(a,b)∈Sprod

inj_onto_Abs_Sprod      inj_onto(Abs_Sprod,Sprod)

strict_Spair_Rep        (a=⊥ ∨ b=⊥)
                         ==>(Spair_Rep(a,b) = Spair_Rep(⊥,⊥))
defined_Spair_Rep_rev   (Spair_Rep(a,b) = Spair_Rep(⊥,⊥))
                         ==>(a=⊥ ∨ b=⊥)
inject_Spair_Rep        [| ¬aa=⊥ ; ¬ba=⊥ ;
                           Spair_Rep(a,b)=Spair_Rep(aa,ba)|]
                           ==>a=aa ∧ b=ba

inject_Ispair           [| ¬aa=⊥ ; ¬ba=⊥ ; Ispair(a,b)=Ispair(aa,ba)|]
                           ==>a=aa ∧ b=ba
strict_Ispair           (a=⊥ ∨ b=⊥)==>Ispair(a,b)=Ispair(⊥,⊥)
strict_Ispair1          Ispair(⊥,b) = Ispair(⊥,⊥)
strict_Ispair2          Ispair(a,⊥) = Ispair(⊥,⊥)
strict_Ispair_rev       ¬Ispair(x,y)=Ispair(⊥,⊥)==>¬x=⊥ ∧ ¬y=⊥
defined_Ispair_rev      Ispair(a,b) = Ispair(⊥,⊥)==>(a = ⊥ ∨ b = ⊥)
defined_Ispair          [| ¬a=⊥; ¬b=⊥|]==>¬(Ispair(a,b) = Ispair(⊥,⊥))

Exh_Sprod               z=Ispair(⊥,⊥) ∨
                        (∃a b. z=Ispair(a,b) ∧ ¬a=⊥ ∧ ¬b=⊥)
IsprodE                 [| p=Ispair(⊥,⊥)==>Q ;
                           ⋀x y. [| p=Ispair(x,y);¬x=⊥ ; ¬y=⊥|]==>Q|]==>Q

strict_Isfst            p=Ispair(⊥,⊥)==>Isfst(p)=⊥
strict_Isfst1           Isfst(Ispair(⊥,y)) = ⊥
strict_Isfst2           Isfst(Ispair(x,⊥)) = ⊥
strict_Issnd            p=Ispair(⊥,⊥)==>Issnd(p)=⊥
strict_Issnd1           Issnd(Ispair(⊥,y)) = ⊥
strict_Issnd2           Issnd(Ispair(x,⊥)) = ⊥

Isfst                   [| ¬x=⊥ ;¬y=⊥|]==>Isfst(Ispair(x,y)) = x
Issnd                   [| ¬x=⊥ ;¬y=⊥|]==>Issnd(Ispair(x,y)) = y
Isfst2                  ¬y=⊥==>Isfst(Ispair(x,y))=x
Issnd2                  ¬x=⊥==>Issnd(Ispair(x,y))=y

defined_IsfstIssnd      ¬p=Ispair(⊥,⊥)==>¬Isfst(p)=⊥ ∧ ¬Issnd(p)=⊥
surjective_pairing_Sprod  z = Ispair(Isfst(z))(Issnd(z))
```

Abbildung 4.32: Theoreme der Theorie `Sprod0`

Die Theoreme `strict_Isfst` bis `defined_IsfstIssnd` beschreiben Definiertheits- und Reduktionseigenschaften der Selektoren `Isfst` und `Issnd`. Das letzte Theorem ist eine alternative Formulierung der Ausschöpfungseigenschaft.

4.8.3 Die Theorie Sprod1

In der Theorie Sprod1 wird eine Ordnungsrelation für das strikte Produkt definiert. Sie stützt sich auf die Ordnung der Argumenttypen ab. Die Theorie ist in Abbildung 4.33 dargestellt.

```
Sprod1 = Sprod0 +

consts   less_sprod:: [(α ** β),(α ** β)] ⇒ bool
rules
  less_sprod_def  less_sprod(p1,p2) ≡ εz.
              ( p1=Ispair(⊥,⊥) → z = True)
            ∧ (¬p1=Ispair(⊥,⊥) → z = ( Isfst(p1) ⊑ Isfst(p2) ∧
                                       Issnd(p1) ⊑ Issnd(p2)))
end
```

Abbildung 4.33: Theorie Sprod1

4.8.4 Theoreme der Theorie Sprod1

Die Theoreme der Theorie Sprod1 sind in Abbildung 4.34 abgebildet.

```
less_sprod1a    p1=Ispair(⊥,⊥)⟹less_sprod(p1,p2)
less_sprod1b   ¬p1=Ispair(⊥,⊥)⟹
                less_sprod(p1,p2) =
                ( Isfst(p1) ⊑ Isfst(p2) ∧ Issnd(p1) ⊑ Issnd(p2))

less_sprod2a   less_sprod(Ispair(x,y),Ispair(⊥,⊥))⟹x = ⊥ ∨ y = ⊥
less_sprod2b   less_sprod(p,Ispair(⊥,⊥))⟹p = Ispair(⊥,⊥)
less_sprod2c   [| less_sprod(Ispair(xa,ya),Ispair(x,y));
                  ¬xa = ⊥ ; ¬ya = ⊥; ¬x = ⊥ ; ¬y = ⊥|]
                  ⟹xa ⊑ x ∧ ya ⊑ y

refl_less_sprod     less_sprod(p,p)
antisym_less_sprod  [| less_sprod(p1,p2);less_sprod(p2,p1)|]⟹p1=p2
trans_less_sprod    [| less_sprod(p1,p2);less_sprod(p2,p3)|]
                      ⟹less_sprod(p1,p3)
```

Abbildung 4.34: Theoreme der Theorie Sprod1

Die Theoreme less_sprod1a bis less_sprod2c sind Hilfstheoreme, die es schlußendlich gestatten, die Eigenschaften einer partiellen Ordnung refl_less_sprod bis trans_less_sprod für die Funktion less_sprod nachzuweisen.

4.8.5 Die Theorie Sprod2

In der Theorie Sprod2 wird die neue Aritätsvereinbarung **::(pcpo,pcpo)po nebst Instantiierung des polymorphen Ordnungssymbols vorgenommen, was durch die Theoreme refl_less_sprod bis trans_less_sprod aus der Theorie Sprod1 gerechtfertigt wird. Die Theorie ist in Abbildung 4.35 dargestellt.

```
Sprod2 = Sprod1 +

arities  **::(pcpo,pcpo)po

rules
inst_sprod_po    ( ⊑ ::[α ** β,α ** β] ⇒ bool) = less_sprod
end
```

Abbildung 4.35: Theorie Sprod2

4.8.6 Theoreme der Theorie Sprod2

Die Theoreme der Theorie Sprod2 sind in Abbildung 4.36 dargestellt. Die Theoreme less_sprod3a bis less_sprod4c entsprechen den Theoremen less_sprod1a bis less_sprod2c aus Abbildung 4.34. Sie machen die gleiche logische Aussage, mit dem Unterschied, daß jetzt das polymorphe Ordnungssymbol $\sqsubseteq$ verwendet wird. Dies erleichtert zum einen die Notation, ermöglicht aber vor allem, die polymorphen Theoreme bezüglich $\sqsubseteq$ auf die Ordnungsrelation less_sprod anzuwenden.

Die beiden Theoreme minimal_sprod und cpo_sprod weisen nach, daß das strikte Produkt durch die Ordnung less_sprod eine pcpo-Struktur erhält. Die Theoreme monofun_Ispair1 bis monofun_Issnd weisen die Monotonie des Konstruktors Ispair und der Selektoren Isfst und Issnd nach.

```
less_sprod3a      p1=Ispair(⊥,⊥)⟹p1 ⊑ p2
less_sprod3b      ¬p1=Ispair(⊥,⊥)⟹
                    (p1 ⊑ p2) = (Isfst(p1) ⊑ Isfst(p2) ∧ Issnd(p1) ⊑ Issnd(p2))
less_sprod4b      p ⊑ Ispair(⊥,⊥)⟹p = Ispair(⊥,⊥)
less_sprod4a      Ispair(a,b) ⊑ Ispair(⊥,⊥)⟹a = ⊥ ∨ b = ⊥
less_sprod4c      [| Ispair(xa,ya) ⊑ Ispair(x,y);¬xa=⊥;¬ya=⊥;¬x=⊥;¬y=⊥ |]
                    ⟹xa ⊑ x ∧ ya ⊑ y

minimal_sprod     Ispair(⊥,⊥) ⊑ p

monofun_Ispair1   monofun(Ispair)
monofun_Ispair2   monofun(Ispair(x))
monofun_Ispair    [| x1 ⊑ x2; y1 ⊑ y2 |]⟹Ispair(x1,y1) ⊑ Ispair(x2,y2)
```

```
monofun_Isfst    monofun(Isfst)
monofun_Issnd    monofun(Issnd)

lub_sprod        [| is_chain(S)|]
                   ==>range(S) <<| Ispair(lub(range(λi.Isfst(S(i)))),
                                          lub(range(λi.Issnd(S(i)))))

thelub_sprod     is_chain(S1)==>lub(range(S1)) =
                   Ispair(lub(range(λi. Isfst(S1(i)))),
                          lub(range(λi. Issnd(S1(i)))))

cpo_sprod        is_chain(S::nat ⇒ α**β)==>∃x.range(S) <<| x
```

Abbildung 4.36: Theoreme der Theorie `Sprod2`

4.8.7 Die Theorie `Sprod3`

Die Theorie `Sprod3` stellt den letzten Schritt in der Einführung des strikten Produkts dar. Die Theorie ist in Abbildung 4.37 abgebildet.

```
Sprod3 = Sprod2 +

arities  **  :: (pcpo,pcpo)pcpo

consts
         @spair      ::  α ⇒ β ⇒ (α**β)   ( _##_  [101,100] 100)
         cop @spair  ::  α → β → (α**β)   ( spair )

        sfst         ::  (α**β) → α
        ssnd         ::  (α**β) → β
        ssplit       ::  (α → β → γ) → (α**β) → γ

translations  x##y  ⇒  spair[x][y]

rules

inst_sprod_pcpo  ⊥::α**β = Ispair(⊥,⊥)
spair_def        spair ≡ (Λx y.Ispair(x,y))
sfst_def         sfst ≡ (Λp.Isfst(p))
ssnd_def         ssnd ≡ (Λp.Issnd(p))
ssplit_def       ssplit ≡ (Λf. strictify[Λp.f[sfst[p]][ssnd[p]]])
end
```

Abbildung 4.37: Theorie `Sprod3`

Eine Aufgabe der Theorie Sprod3 ist die Einführung der neuen Arität **::(pcpo,pcpo)pcpo und die Instantiierung für das kleinste Element bzgl. der Ordnungsrelation in Axiom inst_sprod_pcpo. Dies wird durch die Theoreme minimal_sprod und cpo_sprod der Theorie Sprod2 gerechtfertigt.

Darüberhinaus werden die den internen Funktionen Ispair, Isfst und Issnd entsprechenden Operationen spair, sfst und ssnd eingeführt. Für die Paarbildung spair wird eine Infix-Schreibweise vereinbart, die es erlaubt, statt spair[x][y] kurz x##y zu schreiben. Der Infix-Konstruktor ## wurde rechtsassoziierend eingeführt, was die bequeme Schachtelung von Paaren erlaubt. Damit können n-stellige Produkte simuliert werden. Statt spair[x][spair[y][z]] schreibt man kurz x##y##z.

Die Einführung einer Infix-Operation mit zugehöriger Bindungspriorität ist derzeit noch etwas umständlich in Isabelle. Um die Infix-Schreibweise _##_ für den Konstruktor spair zu vereinbaren, waren die beiden für Isabelle internen Konstanten '@spair' und 'cop @spair' nötig. Die Übersetzung der Mixfix-Schreibweise x##y in die Isabelle-interne Präfixschreibweise mittels spair[x][y] (parse translation) wird durch die Vereinbarung hinter dem Schlüsselwort translations erreicht. Die Rückübersetzung in die Mixfix-Schreibweise bei der Ausgabe wird durch ein spezielles ML-Programm (print translation) für die Applikationsfunktion fapp geleistet, das sich am Standard-Präfix 'cop @' für Infix-Operationen orientiert. Der Übersetzungsapparat mittels dem Schlüsselwort translations war für die Rückübersetzung nicht mächtig genug. Der Mechanismus zur Einführung von Mixfix-Operationen in Isabelle wird im folgenden noch mehrfach verwendet werden und wird sich dadurch selbst erklären. Die Interna sind im Rahmen dieser Arbeit uninteressant.

Als letztes wird in der Theorie Sprod3 das Diskriminatorfunktional ssplit als Operation eingeführt. Die Reduktionseigenschaften von ssplit können den Theoremen zur Theorie Sprod3 in Abbildung 4.39 entnommen werden. Bei der Definition von ssplit ist die Verwendung der Operation strictify aus Theorie Cfun3 von entscheidender Bedeutung, um ein striktes Verhalten der Operation ssplit im Paarargument zu erreichen.

4.8.8 Theoreme der Theorie Sprod3

Die Theoreme der Theorie Sprod3 sind in den Abbildungen 4.38 und 4.39 dargestellt. Im ersten Teil in Abbildung 4.38 sind Theoreme enthalten, die die Stetigkeit der Funktionen Ispair, Isfst und Issnd zeigen. Die wesentliche Beweisarbeit steckt dabei in den Hilfstheoremen sprod3_lemma1 bis sprod3_lemma6.

```
sprod3_lemma1    [| is_chain(Y);  x¬= ⊥;  lub(range(Y))¬= ⊥|]==>
                    Ispair(lub(range(Y)),x) =
                    Ispair(lub(range(λi. Isfst(Ispair(Y(i),x)))),
                           lub(range(λi. Issnd(Ispair(Y(i),x)))))

sprod3_lemma2    [| is_chain(Y); ¬x = ⊥; lub(range(Y)) = ⊥|]==>
                    Ispair(lub(range(Y)),x) =
                    Ispair(lub(range(λi. Isfst(Ispair(Y(i),x)))),
                           lub(range(λi. Issnd(Ispair(Y(i),x)))))
```

```
sprod3_lemma3    [| is_chain(Y); x = ⊥|]==>
                    Ispair(lub(range(Y)),x) =
                    Ispair(lub(range(λi. Isfst(Ispair(Y(i),x)))),
                           lub(range(λi. Issnd(Ispair(Y(i),x)))))

sprod3_lemma4    [| is_chain(Y); ¬x = ⊥; ¬lub(range(Y)) = ⊥|]==>
                    Ispair(x,lub(range(Y))) =
                    Ispair(lub(range(λi. Isfst(Ispair(x,Y(i))))),
                           lub(range(λi. Issnd(Ispair(x,Y(i))))))

sprod3_lemma5    [| is_chain(Y); ¬x = ⊥; lub(range(Y)) = ⊥|]==>
                    Ispair(x,lub(range(Y))) =
                    Ispair(lub(range(λi. Isfst(Ispair(x,Y(i))))),
                           lub(range(λi. Issnd(Ispair(x,Y(i))))))

sprod3_lemma6    [| is_chain(Y); x = ⊥|]==>
                    Ispair(x,lub(range(Y))) =
                    Ispair(lub(range(λi. Isfst(Ispair(x,Y(i))))),
                           lub(range(λi. Issnd(Ispair(x,Y(i))))))

contX_Ispair1    contX(Ispair)
contX_Ispair2    contX(Ispair(x))
contX_Isfst      contX(Isfst)
contX_Issnd      contX(Issnd)
```

Abbildung 4.38: Theoreme der Theorie `Sprod3` - Teil 1

In Abbildung 4.39 sind unter anderem die Reduktionseigenschaften der in Theorie `Sprod3` eingeführten Operationen formuliert. Daneben werden die bereits bekannten und für den späteren Gebrauch wichtigen Eigenschaften des strikten Produkts unter Verwendung der Operationen ausgedrückt. Die Beweise für die Theoreme in Abbildung 4.39 sind alle von den Stetigkeitsresultaten aus Abbildung 4.38 abhängig.

Als letztes ist noch die Vereinbarung der Liste von Theoremen `Sprod_rews` gezeigt, die die Grundlage für eine Menge von Termersetzungsregeln bildet, mit denen der Simplifikator bei Beweisen über strikte Produkte gefüttert werden kann.

```
spair_eq           [| x1=x2;y1=y2|]==>x1##y1 = x2##y2
inject_spair       [| ¬aa=⊥ ; ¬ba=⊥ ; (a##b)=(aa##ba)|]==>a=aa ∧ b=ba
inst_sprod_pcpo2   ⊥ = (⊥##⊥)

strict_spair       (a=⊥ ∨ b=⊥)==>(a##b)=⊥
strict_spair1      (⊥##b) = ⊥
strict_spair2      (a##⊥) = ⊥
strict_spair_rev   ¬(x##y)=⊥==>¬x=⊥ ∧ ¬y=⊥
```

```
defined_spair_rev  (a##b) = ⊥==>(a = ⊥ ∨ b = ⊥)
defined_spair      [| ¬a=⊥; ¬b=⊥|]==>¬(a##b) = ⊥

Exh_Sprod2         z=⊥ ∨ (∃a b. z=(a##b) ∧ ¬a=⊥ ∧ ¬b=⊥)
sprodE             [| p=⊥==>Q;
                      ⋀x y. [| p=(x##y);¬x=⊥ ; ¬y=⊥|]==>Q|]==>Q

strict_sfst        p=⊥==>sfst[p]=⊥
strict_sfst1       sfst[⊥##y] = ⊥
strict_sfst2       sfst[x##⊥] = ⊥
strict_ssnd        p=⊥==>ssnd[p]=⊥
strict_ssnd1       ssnd[⊥##y] = ⊥
strict_ssnd2       ssnd[x##⊥] = ⊥

sfst2              ¬y=⊥==>sfst[x##y]=x
ssnd2              ¬x=⊥==>ssnd[x##y]=y
defined_sfstssnd   ¬p=⊥==>¬sfst[p]=⊥ ∧ ¬ssnd[p]=⊥

surjective_pairing_Sprod2  (sfst[p] ## ssnd[p]) = p

less_sprod5b       ¬p1=⊥==>(p1 ⊑ p2) =
                              (sfst[p1] ⊑ sfst[p2] ∧ ssnd[p1] ⊑ ssnd[p2])
less_sprod5c       [| xa##ya ⊑ x##y;¬xa=⊥;¬ya=⊥;¬x=⊥;¬y=⊥|]
                     ==>xa ⊑ x ∧ ya ⊑ y

lub_sprod2         [| is_chain(S)|]==>range(S) ≪|
                     (lub(range(λi.sfst[S(i)]))##lub(range(λi.ssnd[S(i)])))

thelub_sprod2      is_chain(S1)
                   ==>lub(range(S1)) = (lub(range(λi. sfst[S1(i)]))##
                                        lub(range(λi. ssnd[S1(i)])))

ssplit1            ssplit[f][⊥]=⊥
ssplit2            [| ¬x=⊥;¬y=⊥|]==>ssplit[f][x##y]=f[x][y]
ssplit3            ssplit[spair][z]=z


val Sprod_rews = [strict_spair1,strict_spair2,strict_sfst1,strict_sfst2,
                  strict_ssnd1,strict_ssnd2,sfst2,ssnd2,
                  ssplit1,ssplit2];
```

Abbildung 4.39: Theoreme der Theorie Sprod3 - Teil 2

4.9 Theorien für das kartesische Produkt

In diesem Abschnitt stelle ich die Theorien für das kartesische Produkt vor. Die konservative Einführung des Typkonstruktors gliedert sich auch hier wieder in mehrere Schritte. Da diese aber analog zur Einführung des strikten Produkts vollzogen wird, fasse ich mich diesmal sehr kurz und stelle nur die einzelnen Theorien mit wenigen Kommentaren dar. In Abschnitt 4.9.4 werden dann nur diejenigen Theoreme aufgelistet, die für das spätere Arbeiten mit dem kartesischen Produkt relevant sind.

4.9.1 Die Theorie `Cprod1`

Im Gegensatz zum strikten Produkt ist die Formalisierung des kartesischen Produkts schon teilweise in Isabelle-HOL vorweggenommen worden. Auszüge aus dieser Formalisierung sind in den Abbildungen 3.5 und 3.6 aus Kapitel 3 dargestellt. Daher können wir schon auf dem Typkonstruktor `*` mit Arität `(term,term)term` aufsetzen und mit einer Definition für die partielle Ordnung beginnen. Bei der Definition gehen wir davon aus, daß der Konstruktor `*` auf Bereichstypen angewendet wird. Die diesbezügliche Theorie ist in Abbildung 4.41 dargestellt.

```
Cprod1 = Cfun3 +

consts
  less_cprod    ::  [(α_pcpo * β_pcpo),(α * β)] ⇒ bool

rules

  less_cprod_def  less_cprod(p1,p2) ≡ ( fst(p1) ⊑ fst(p2) ∧
                                        snd(p1) ⊑ snd(p2))
end
```

Abbildung 4.40: Theorie `Cprod1`

4.9.2 Die Theorie `Cprod2`

In der Theorie `Cprod2` wird die Aritätsvereinbarung `*::(pcpo,pcpo)po` zusammen mit der Instantiierung für das polymorphe Ordnungssymbol $\sqsubseteq$ durchgeführt. Die entsprechenden Theoreme zur Rechtfertigung dieser Vorgehensweise wurden in der Theorie `Cprod1` bewiesen. Die Theorie `Cprod2` ist in Abbildung 4.41 dargestellt.

```
Cprod2 = Cprod1 +

arities  *  :: (pcpo,pcpo)po

rules

inst_cprod_po    ( ⊑ ::[α * β,α * β] ⇒ bool) = less_cprod

end
```

Abbildung 4.41: Theorie `Cprod2`

4.9.3 Die Theorie `Cprod3`

In der Theorie `Cprod3` wird die Aritätsvereinbarung `*::(pcpo,pcpo)pcpo` mit zugehöriger Instanz für das kleinste Element $\bot$ im Bereich eingeführt. Die entsprechenden Theoreme zur Rechtfertigung wurden in der Theorie `Cprod2` bewiesen. Die Theorie ist in Abbildung 4.42 dargestellt.

Weiterhin werden die den Funktionen `Pair`, `fst` und `snd` entsprechenden Operationen `cpair`, `cfst` und `csnd` eingeführt. Die Operation `csplit` stützt sich nicht auf die Funktion `split` ab, sondern wird analog zum strikten Produkt über die Destruktoren definiert.

Für die Konstruktor-Operation `cpair` wird analog zum strikten Produkt eine nach rechts assoziierende Infix-Schreibweise eingeführt. Statt `cpair[x][cpair[y][z]]` darf man kurz `x#y#z` schreiben.

```
Cprod3 = Cprod2 +

arities  *  :: (pcpo,pcpo)pcpo

consts
         @cpair      ::  α ⇒ β ⇒ (α*β)   ( _#_  [101,100] 100)
         cop @cpair  ::  α → β → (α*β)   ( cpair )

        cfst         ::  (α*β) → α
        csnd         ::  (α*β) → β
        csplit       ::  (α → β → γ) → (α*β) → γ

translations  x#y  ⇒  cpair[x][y]

rules

inst_cprod_pcpo  ⊥::α*β = <⊥,⊥>

cpair_def        cpair ≡ (Λx y.<x,y>)
```

```
cfst_def        cfst ≡ (Λp.fst(p))
csnd_def        csnd ≡ (Λp.snd(p))
csplit_def      csplit ≡ (Λf p.f[cfst[p]][csnd[p]])
end
```

Abbildung 4.42: Theorie Cprod3

4.9.4 Theoreme der Theorie Cprod3

In diesem Abschnitt werden die anwendungsrelevanten Eigenschaften für das kartesische Produkt dargestellt. Die Theoreme sind in Abbildung 4.43 aufgelistet. Als letztes ist die Vereinbarung der Liste Cprod_rews gezeigt, die die Basis für Termsimplifiktion bezüglich kartesischer Produkte ist.

```
inject_cpair       (a#b)=(aa#ba)⟹a=aa ∧ b=ba
inst_cprod_pcpo2   ⊥ = (⊥#⊥)
defined_cpair_rev  (a#b) = ⊥⟹a = ⊥ ∧ b = ⊥

Exh_Cprod2         ∃a b. z=(a#b)
cprodE             ⟦ ⋀x y. ⟦ p=(x#y)⟧⟹Q⟧⟹Q

cfst2              cfst[x#y]=x
csnd2              csnd[x#y]=y
surjective_pairing_Cprod2   (cfst[p] # csnd[p]) = p

less_cprod5b   (p1 ⊑ p2) = (cfst[p1] ⊑ cfst[p2] ∧ csnd[p1] ⊑ csnd[p2])
less_cprod5c   xa#ya ⊑ x#y⟹xa ⊑ x ∧ ya ⊑ y

lub_cprod2     ⟦ is_chain(S)⟧⟹range(S) ≪
                 (lub(range(λi.cfst[S(i)])) # lub(range(λi.csnd[S(i)])))
thelub_cprod2  is_chain(S1)
               ⟹lub(range(S1)) = lub(range(λi. cfst[S1(i)]))#
                                 lub(range(λi. csnd[S1(i)]))

csplit2        csplit[f][x#y]=f[x][y]
csplit3        csplit[cpair][z]=z

val Cprod_rews = [cfst2,csnd2,csplit2];
```

Abbildung 4.43: Theoreme der Theorie Cprod3

4.10 Theorien für die strikte Summe

In diesem Abschnitt beschreibe ich die Theorien für die strikte Summe. Die konservative Einführung des Typkonstruktors ++ gliedert sich in mehrere Schritte, die der Einführung des strikten Produkts entsprechen. Daher fasse ich mich kurz und stelle nur die einzelnen Theorien mit wenigen Kommentaren dar. In Abschnitt 4.10.5 werden dann nur diejenigen Theoreme aufgelistet, die für das spätere Arbeiten mit dem Typ der strikten Summe relevant sind. Die Probleme beim Beweisen der Theoreme für die strikte Summe waren, wie nicht anders zu erwarten, dual zu denen des strikten Produkts gelagert.

4.10.1 Die Theorie Ssum0

Analog zum strikten Produkt muß in einem ersten Schritt der Typkonstruktor ++ für die strikte Summe mit der Arität (pcpo,pcpo)term eingeführt werden. Der Typkonstruktor + für die Summe, der schon in Isabelle-HOL eingeführt wurde, siehe Abbildungen 3.7 und 3.8 in Kapitel 3, war für die Kodierung der Repräsentation nicht geeignet. Die Kodierung für die strikte Summe ++ orientiert sich aber an der üblichen Kodierung [GM93, Pau94] für die Summe +. Die Theorie Ssum0, die den Typkonstruktor einführt, ist in Abbildung 4.44 abgebildet.

```
Ssum0 = Cfun3 +

types    ++  2  (infixr 10)
arities  ++  :: (pcpo,pcpo)term

consts
  Ssum          ::  ([α,β,bool] ⇒ bool)set
  Sinl_Rep      ::  [α,α,β,bool] ⇒ bool
  Sinr_Rep      ::  [β,α,β,bool] ⇒ bool
  Rep_Ssum      ::  (α ++ β) ⇒ ([α,β,bool] ⇒ bool)
  Abs_Ssum      ::  ([α,β,bool] ⇒ bool) ⇒ (α ++ β)
  Isinl         ::  α ⇒ (α ++ β)
  Isinr         ::  β ⇒ (α ++ β)
  Iwhen         ::  (α → γ) ⇒ (β → γ) ⇒ (α ++ β) ⇒ γ

rules

  Sinl_Rep_def          Sinl_Rep ≡ (λa.λx y p.(¬a=⊥ → x=a ∧ p))

  Sinr_Rep_def          Sinr_Rep ≡ (λb.λx y p.(¬b=⊥ → y=b ∧ ¬p))

  Ssum_def              Ssum ≡ {f.(∃a.f=Sinl_Rep(a)) ∨ (∃b.f=Sinr_Rep(b))}

  Rep_Ssum              Rep_Ssum(p)∈Ssum
  Rep_Ssum_inverse      Abs_Ssum(Rep_Ssum(p)) = p
```

```
  Abs_Ssum_inverse       f∈Ssum⟹Rep_Ssum(Abs_Ssum(f)) = f

  Isinl_def      Isinl(a) ≡ Abs_Ssum(Sinl_Rep(a))
  Isinr_def      Isinr(b) ≡ Abs_Ssum(Sinr_Rep(b))

  Iwhen_def      Iwhen(f)(g)(s) ≡ εz.
                                    (s=Isinl(⊥) → z=⊥)
                           ∧ (∀a. ¬a=⊥ ∧ s=Isinl(a) → z=f[a])
                           ∧ (∀b. ¬b=⊥ ∧ s=Isinr(b) → z=g[b])
end
```

Abbildung 4.44: Theorie `Ssum0`

Die Typdefinition durch die Axiome `Rep_Ssum`, `Rep_Ssum_inverse` und `Abs_Ssum_inverse` ist konservativ, da `Sinl_Rep(a)` und `Sinr_Rep(b)` für jedes `a` und `b` Elemente der Menge `Ssum` sind. Die Kodierung der Repräsentanten ist wie beim strikten Produkt sehr kryptisch.

Die Funktionen `Isinl` und `Isinr` sind die Konstruktoren für Summenelemente, die Funktion `Iwhen` ist ein Diskriminatorfunktional für den Typ der strikten Summe. Für diese Funktionen werden in der noch folgenden Theorie `Ssum3` wieder entsprechende Operationen eingeführt.

4.10.2 Die Theorie `Ssum1`

In der Theorie `Ssum1` wird die Ordnungsrelation `less_ssum` eingeführt. Die Definition `less_ssum_def` zeigt, wie man in Logik höherer Stufe Funktionen über Patternmatching unter Zuhilfenahme des Hilbertoperators definiert. Für die Relation `less_ssum` wird in den Theoremen zu den Theorien `Ssum1` und `Ssum2` gezeigt, daß sie die strikte Summe mit einer pcpo-Struktur versieht. Die Theorie `Ssum1` ist in Abbilding 4.45 dargestellt.

```
Ssum1 = Ssum0 +

consts

  less_ssum     ::  [(α ++ β),(α ++ β)] ⇒ bool

rules

  less_ssum_def  less_ssum(s1,s2) ≡ (εz.
           (∀u x.s1=Isinl(u) ∧ s2=Isinl(x) → z = (u ⊑ x))
         ∧ (∀v y.s1=Isinr(v) ∧ s2=Isinr(y) → z = (v ⊑ y))
         ∧ (∀u y.s1=Isinl(u) ∧ s2=Isinr(y) → z = (u = ⊥))
         ∧ (∀v x.s1=Isinr(v) ∧ s2=Isinl(x) → z = (v = ⊥)))
end
```

Abbildung 4.45: Theorie `Ssum1`

4.10.3 Die Theorie Ssum2

In der Theorie Ssum2 werden die Aritätsvereinbarung ++::(pcpo,pcpo)po nebst der Instantiierung für die charakteristische Konstante $\sqsubseteq$ vorgenommen. Die Rechtfertigung hierfür erfolgt durch den Beweis entsprechender Theoreme in der Theorie Ssum1. Die Theorie Ssum2 ist in Abbildung 4.46 dargestellt.

```
Ssum2 = Ssum1 +

arities  ++  :: (pcpo,pcpo)po

rules

inst_ssum_po     ( ⊑ ::[α ++ β,α ++ β] ⇒ bool) = less_ssum

end
```

Abbildung 4.46: Theorie Ssum2

4.10.4 Die Theorie Ssum3

In der Theorie Ssum3 wird die Aritätsvereinbarung ++::(pcpo,pcpo)pcpo mit zugehöriger Instanz für die charakteristische Konstante $\bot$ eingeführt. Die entsprechenden Theoreme zur Rechtfertigung wurden in der Theorie Ssum2 bewiesen. Die Theorie ist in Abbildung 4.47 dargestellt. Weiterhin werden die den Funktionen Isinl, Isinr und Iwhen entsprechenden Operationen sinl, sinr und when eingeführt.

```
Ssum3 = Ssum2 +

arities  ++  :: (pcpo,pcpo)pcpo

consts
        sinl :: α → (α++β)
        sinr :: β → (α++β)
        when :: (α → γ) → (β → γ) → (α ++ β) → γ
rules

inst_ssum_pcpo   ⊥::α++β = Isinl(⊥)

sinl_def         sinl ≡ (Λx.Isinl(x))
sinr_def         sinr ≡ (Λx.Isinr(x))
when_def         when ≡ (Λf g s.Iwhen(f)(g)(s))

end
```

Abbildung 4.47: Theorie Ssum3

4.10.5 Theoreme der Theorie Ssum3

In diesem Abschnitt werden die anwendungsrelevanten Eigenschaften für die strikte Summe dargestellt. Die Theoreme sind in Abbildung 4.48 aufgelistet. Als letztes ist die Vereinbarung der Liste Ssum_rews gezeigt, die die Basis für Termsimplifiktion bezüglich des Typs der strikten Summe ist.

```
strict_sinl     sinl[⊥]=⊥
strict_sinr     sinr[⊥]=⊥

noteq_sinlsinr  sinl[a]=sinr[b]⟹a=⊥ ∧ b=⊥
inject_sinl     sinl[a1]=sinl[a2]⟹a1=a2
inject_sinr     sinr[a1]=sinr[a2]⟹a1=a2
defined_sinl    ¬x=⊥⟹¬sinl[x]=⊥
defined_sinr    ¬x=⊥⟹¬sinr[x]=⊥

Exh_Ssum1       z=⊥ ∨ (∃a. z=sinl[a] ∧ ¬a=⊥) ∨ (∃b. z=sinr[b] ∧ ¬b=⊥)
ssumE           ⟦ p=⊥⟹Q ;
                  ⋀x.⟦ p=sinl[x]; ¬x=⊥⟧⟹Q;
                  ⋀y.⟦ p=sinr[y]; ¬y=⊥⟧⟹Q⟧⟹Q

ssumE2          ⟦ ⋀x.⟦ p=sinl[x]⟧⟹Q;
                  ⋀y.⟦ p=sinr[y]⟧⟹Q⟧⟹Q

when1           when[f][g][⊥] = ⊥
when2           ¬x=⊥⟹when[f][g][sinl[x]] = f[x]
when3           ¬x=⊥⟹when[f][g][sinr[x]] = g[x]
when4           when[sinl][sinr][z]=z

less_ssum4a     (sinl[x] ⊑ sinl[y]) = (x ⊑ y)
less_ssum4b     (sinr[x] ⊑ sinr[y]) = (x ⊑ y)
less_ssum4c     (sinl[x] ⊑ sinr[y]) = (x = ⊥)
less_ssum4d     (sinr[x] ⊑ sinl[y]) = (x = ⊥)

ssum_chainE     is_chain(Y)⟹(∀i.∃x.Y(i)=sinl[x]) ∨ (∀i.∃y.Y(i)=sinr[y])


thelub_ssum2a   ⟦ is_chain(Y); ∀i.∃x. Y(i) = sinl[x]⟧
                  ⟹lub(range(Y)) =
                     sinl[lub(range(λi. when[Λx.x][Λy.⊥][Y(i)]))]

thelub_ssum2b   ⟦ is_chain(Y); ∀i.∃x. Y(i) = sinr[x]⟧
                  ⟹lub(range(Y)) =
                     sinr[lub(range(λi. when[Λy.⊥][Λx.x][Y(i)]))]

thelub_ssum2a_rev  ⟦ is_chain(Y); lub(range(Y)) = sinl[x]⟧
                     ⟹∀i.∃x.Y(i)=sinl[x]
```

```
thelub_ssum2b_rev  [| is_chain(Y); lub(range(Y)) = sinr[x] |]
                     ==>∀i.∃x.Y(i)=sinr[x]

thelub_ssum3       is_chain(Y)
                   ==>lub(range(Y)) =
                        sinl[lub(range(λi. when[Λx.x][Λy.⊥][Y(i)]))]
                    ∨ lub(range(Y)) =
                        sinr[lub(range(λi. when[Λy.⊥][Λx.x][Y(i)]))]

val Ssum_rews = [strict_sinl,strict_sinr,when1,when2,when3];
```

Abbildung 4.48: Theoreme der Theorie `Ssum3`

4.11 Theorien für den gelifteten Bereich

Als letzten Typkonstruktor für Bereichstypen führe ich in diesem Abschnitt den Konstruktor für das Lifting von Typen ein. Die Darstellung ist ähnlich knapp gehalten wie die für die Konstruktoren `*` und `++`. Analog zur konservativen Einführung des Konstruktors `**` für das strikte Produkt werden wieder mehrere Schritte benötigt. Die für die Anwendung relevanten Eigenschaften des gelifteten Typs sind in Abschnitt 4.11.4 dargestellt.

4.11.1 Die Theorie `Lift1`

In einem ersten Schritt wird der einstellige Typkonstruktor `u` mit der Arität `(pcpo)term` eingeführt. Die Theorie dazu ist in Abbildung 4.49 dargestellt.

Die Anwendung des Typkonstruktors wird in Postfixnotation geschrieben. Für die Kodierung der Repräsentanten werden die bereits bekannten Typkonstruktoren `+` für die Summe, siehe Abbildungen 3.7 und 3.8 in Kapitel 3, und der triviale Typ `void`, siehe Abbildung 4.4 in Abschnitt 4.2, verwendet. Repräsentanten für den neuen Typ sind alle Elemente aus dem Typ (`void +` $\alpha_{\texttt{pcpo}}$), und somit ist die Definition einer Teilmenge von Repräsentanten nicht nötig. Die Typdefinition wird durch die Axiome `Rep_Lift_inverse` und `Abs_Lift_inverse` vorgenommen. Die Rechtfertigung für diese Typdefinition ist diesmal trivial, da kein Repräsentationsprädikat verwendet wird.

Zusätzlich werden noch mehrere Konstanten definiert. Die Funktion `Iup` ist der einzige Konstruktor des neuen Typs. Die Funktion `Ilift` ist ein Diskriminatorfunktional. Die Funktion `less_lift` steht für die Ordnungsrelation, und `⊥_lift` übernimmt die Rolle des kleinsten Elements. Alle Funktionen werden direkt über den Repräsentationstyp definiert. Im Gegensatz zu allen bisher eingeführten Typkonstruktoren für Bereichstypen ist das kleinste Element im gelifteten Typ (`α`)`u` unabhängig vom kleinsten Element des Parametertyps α.

```
Lift1 = Cfun3 +

types    u  1
arities  u  :: (pcpo)term

consts

  Rep_Lift      ::  (α)u ⇒ (void + α)
  Abs_Lift      ::  (void + α) ⇒ (α)u

  Iup           ::  α ⇒ (α)u
  ⊥_lift        ::  (α)u
  Ilift         ::  (α → β) ⇒ (α)u ⇒ β
  less_lift     ::  (α)u ⇒ (α)u ⇒ bool

rules

  Rep_Lift_inverse       Abs_Lift(Rep_Lift(p)) = p
  Abs_Lift_inverse       Rep_Lift(Abs_Lift(p)) = p

  ⊥_lift_def    ⊥_lift ≡ Abs_Lift(Inl(⊥))
  Iup_def        Iup(x) ≡ Abs_Lift(Inr(x))

  Ilift_def      Ilift(f)(x) ≡
                 sum_case  (Rep_Lift(x)) (λy.⊥) (λz.f[z])

  less_lift_def  less_lift(x1)(x2) ≡
          (sum_case (Rep_Lift(x1))
                    (λy1.True)
                    (λy2.sum_case  (Rep_Lift(x2))
                                   (λz1.False)
                                   (λz2.y2 ⊑ z2)))
end
```

Abbildung 4.49: Theorie `Lift1`

4.11.2 Die Theorie `Lift2`

In der Theorie `Ssum2` werden die Aritätsvereinbarung `u::(pcpo)po` nebst der Instantiierung für die charakteristische Konstante $\sqsubseteq$ vorgenommen. Die Rechtfertigung hierfür erfolgt durch den Beweis entsprechender Theoreme in der Theorie `Lift1`. Die Theorie `Lift2` ist in Abbildung 4.50 dargestellt.

```
Lift2 = Lift1 +

arities  u  :: (pcpo)po

rules

inst_lift_po     ( ⊑ ::[(α)u,(α)u] ⇒ bool) = less_lift

end
```

Abbildung 4.50: Theorie `Lift2`

4.11.3 Die Theorie `Lift3`

In der Theorie `Lift3` wird die Aritätsvereinbarung `u::(pcpo)pcpo` mit zugehöriger Instanz für die charakteristische Konstante $\bot$ eingeführt. Die entsprechenden Theoreme zur Rechtfertigung wurden in der Theorie `Lift2` bewiesen. Die Theorie ist in Abbildung 4.51 dargestellt.

Weiterhin werden die den Funktionen `Iup` und `Ilift` entsprechenden Operationen `up` und `lift` eingeführt.

```
Lift3 = Lift2 +

arities  u  :: (pcpo)pcpo

consts
          up :: α → (α)u
        lift :: (α → γ) → (α)u → γ

rules

inst_lift_pcpo   ⊥::(α)u = ⊥_lift

up_def           up ≡ (Λx.Iup(x))
lift_def         lift ≡ (Λf p.Ilift(f)(p))

end
```

Abbildung 4.51: Theorie `Lift3`

4.11.4 Theoreme der Theorie `Lift3`

In diesem Abschnitt werden die anwendungsrelevanten Eigenschaften für den gelifteten Typ dargestellt. Die Theoreme sind in Abbildung 4.52 aufgelistet. Als letztes ist die Vereinbarung

der Liste `lift_rews` gezeigt, die die Basis für Termsimplifiktion bezüglich des gelifteten Typs ist.

```
Exh_Lift1          z = ⊥ ∨ (∃x. z = up[x])

inject_up          up[x]=up[y]⟹x=y
defined_up         ¬up[x]=⊥

liftE1             [| p=⊥⟹Q; ⋀x. p=up[x]⟹Q|]⟹Q

lift1              lift[f][⊥]=⊥
lift2              lift[f][up[x]]=f[x]
lift3              lift[up][x]=x

less_lift4b        ¬up[x] ⊑ ⊥
less_lift4c        (up[x] ⊑ up[y]) = (x ⊑ y)

thelub_lift2a      [| is_chain(Y); ∃i x. Y(i) = up[x]|]
                     ⟹lub(range(Y)) =
                        up[lub(range(λi.lift[Λx.x][Y(i)]))]
thelub_lift2b      [| is_chain(Y); ∀i x. ¬Y(i) = up[x]|]
                      ⟹lub(range(Y)) = ⊥

lift_lemma2        (∃x.z = up[x]) = (¬z=⊥)

thelub_lift2a_rev  [| is_chain(Y); lub(range(Y)) = up[x]|]
                     ⟹∃i x. Y(i) = up[x]

thelub_lift2b_rev  [| is_chain(Y); lub(range(Y)) = ⊥|]
                     ⟹∀i x. ¬Y(i) = up[x]

thelub_lift3       is_chain(Y)⟹
                    lub(range(Y))= ⊥ ∨
                    lub(range(Y))= up[lub(range(λi.lift[Λx.x][Y(i)]))]

val lift_rews = [lift1,lift2,defined_up];
```

Abbildung 4.52: Theoreme der Theorie `Lift3`

4.12 Fixpunkttheorie für Operationen

In diesem Abschnitt stelle ich eine der interessantesten Theorien der Entwicklung von HOLCF dar. In der Theorie `Fix` werden unter anderem der Fixpunktoperator `fix` für Operationen und das Prädikat `adm` für zulässige Prädikate eingeführt.

4.12.1 Die Theorie `Fix`

Die Theorie `Fix` ist in Abbildung 4.53 dargestellt.

```
Fix = Cfun3 +

consts

iterate      :: nat ⇒ (α → α) ⇒ α ⇒ α
Ifix         :: (α → α) ⇒ α
fix          :: (α → α) → α
adm          :: (α ⇒ bool) ⇒ bool
admw         :: (α ⇒ bool) ⇒ bool
chain_finite :: α ⇒ bool
flat         :: α ⇒ bool

rules

iterate_def        iterate(n,F,c) ≡ nat_rec(n,c,λn x.F[x])
Ifix_def           Ifix(F) ≡ lub(range(λi.iterate(i,F,⊥)))
fix_def            fix ≡ (Λf. Ifix(f))

adm_def            adm(P) ≡ ∀Y. is_chain(Y) →
                                (∀i.P(Y(i))) → P(lub(range(Y)))

admw_def           admw(P) ≡ ∀F. (∀n.P(iterate(n,F,⊥))) →
                                P(lub(range(λi.iterate(i,F,⊥))))

chain_finite_def   chain_finite(x::α) ≡
                    ∀Y. is_chain(Y::nat ⇒ α) → (∃n.max_in_chain(n,Y))

flat_def           flat(x::α) ≡
                    ∀x y. x::α ⊑ y → (x = ⊥) ∨ (x=y)
end
```

Abbildung 4.53: Theorie `Fix`

Zuerst wird das Funktional `iterate` im Axiom `iterate_def` durch primitive Rekursion so definiert, daß `iterate(n,F,c)` die `n`-fache Iteration der Operation `F` beginnend mit dem

Anfangswert `c` bezeichnet. Mit Hilfe von `iterate` können somit die einzelnen Elemente der Kleene-Kette erzeugt werden, wenn für `c` das kleinste Element $\bot$ gewählt wird.

Die Konstante `Ifix` steht für die Funktion, die den Fixpunkt einer Operation bildet. Die Definition `Ifix_def` entspricht exakt der üblichen Definition des Fixpunktoperators. Die Konstante `fix` steht für die der Funktion `Ifix` entsprechende Operation. Sie wird in bereits bekannter Manier in Axiom `fix_def` über `Ifix` unter Benutzung der Λ-Abstraktion definiert.

Das Axiom `adm_def` führt das Prädikat `adm` (admissible) für zulässige Prädikate ein. Die anschließende Definition `admw_def` führt eine schwächere Form der Zulässigkeit ein, die sich enger an der Kleene-Kette orientiert. Beide Definitionen der Zulässigkeit erlauben jeweils zusammen mit der Definition des Fixpunktoperators die Ableitung einer Regel zur Fixpunktinduktion.

Die beiden letzten Definitionen charakterisieren kettenendliche (chain-finite) und noch speziellere flache (flat) `pcpo`-Typen. Es kann gezeigt werden, daß alle Prädikate über kettenendlichen Typen zulässig sind, was die Fixpunktinduktion über solchen Typen erheblich vereinfacht.

Allen Definition der Theorie `Fix` ist gemeinsam, daß sie nur in der Logik höherer Stufe mit Typklassen in dieser intuitiven und eleganten Weise formuliert werden können. In LCF wären die Definitionen überhaupt nicht möglich, und in herkömmlicher Logik höherer Stufe müßten die Definitionen mit lästigen Prämissen versehen werden, die die nötigen Ordnungseigenschaften ausdrücken.

4.12.2 Theoreme der Theorie `Fix`

Für die Theorie `Fix` wurde eine ganze Anzahl von Theoremen abgeleitet. Um die Übersichtlichkeit der Darstellung zu gewährleisten, habe ich die Theoreme in vier Gruppen aufgeteilt, die in den Abbildungen 4.54 bis 4.57 dargestellt sind.

```
iterate_0             iterate(0,F,x) = x
iterate_Suc           iterate(Suc(n),F,x) = F[iterate(n,F,x)]

val iterate_ss = Cfun_ss addsimps [iterate_0,iterate_Suc];

iterate_Suc2          iterate(Suc(n),F,x) = iterate(n,F,F[x])

is_chain_iterate2     x ⊑ F[x] ⟹ is_chain(λi.iterate(i,F,x))
is_chain_iterate      is_chain(λi.iterate(i,F,⊥))

Ifix_eq               Ifix(F)=F[Ifix(F)]
Ifix_least            F[x]=x ⟹ Ifix(F) ⊑ x

monofun_iterate       monofun(iterate(i))
contlub_iterate       contlub(iterate(i))
contX_iterate         contX(iterate(i))
monofun_iterate2      monofun(iterate(n,F))
contlub_iterate2      contlub(iterate(n,F))
```

```
contX_iterate2          contX(iterate(n,F))

monofun_Ifix            monofun(Ifix)

is_chain_iterate_lub    is_chain(Y)⟹
                        is_chain(λi. lub(range(λia. iterate(ia,Y(i),⊥))))

contlub_Ifix_lemma1     is_chain(Y)
                        ⟹iterate(n,lub(range(Y)),y) =
                            lub(range(λi. iterate(n,Y(i),y)))

ex_lub_iterate          is_chain(Y)⟹
                        lub(range(λi. lub(range(λia. iterate(i,Y(ia),⊥)))))=
                        lub(range(λi. lub(range(λia. iterate(ia,Y(i),⊥)))))

contlub_Ifix            contlub(Ifix)
contX_Ifix              contX(Ifix)
```

Abbildung 4.54: Theoreme der Theorie `Fix` - Teil 1

Die Theoreme `iterate_0` und `iterate_Suc` in Abbildung 4.54 beschreiben die Reduktionseigenschaften des Funktionals `iterate`, die sich in offensichtlicher Weise aus der primitiv rekursiven Definition ergeben. Beide Theoreme werden mittels der Vereinbarung von `iterate_ss` in den Simplifikator integriert. Der so konfigurierte Simplifikator wurde in den Beweisen für die nachstehenden Theoreme wiederholt eingesetzt. Das Theorem `iterate_Suc2` beschreibt eine weitere Reduktionseigenschaft der Funktion `iterate`, die eine alternative Form der induktiven Argumentation über den Iterator erlaubt.

Das Theorem `is_chain_iterate2` besagt, daß die Iteration einer Operation `F` mit Startwert `x` dann zu einer Kette führt, wenn sich durch die erstmalige Anwendung der Operation auf den Startwert der Anfang einer Kette ergibt. Das Theorem `is_chain_iterate` ist die Spezialisierung des Theorems `is_chain_iterate2`, die durch die Initialität des Elements $\bot$ ermöglicht wird. Es besagt, daß die Kleene-Iteration einer Operation zu einer Kette, der sogenannten Kleene-Kette, führt.

Die Theoreme `Ifix_eq` und `Ifix_least` beschreiben die bekannten Eigenschaften des Fixpunktoperators. Sie sind hier noch über die interne Funktion `Ifix` formuliert.

In den Theoremen `monofun_iterate` bis `contX_iterate2` werden die Monotonie und die Stetigkeit des Funktionals `iterate` im 'zweiten' und 'dritten' Argument ausgedrückt. Man beachte, daß schon die Formulierung der entsprechenden Eigenschaften für das erste Argument technisch falsch wäre, da das erste Argument ein Element aus dem Typ `nat` ist. Der Typ `nat` ist nämlich weder ein Typ der Klasse `po` noch ein Typ der Klasse `pcpo`, sondern lediglich ein Typ in der Klasse `term` der Mengentypen. Für die Abzählung der Ketten ist keinerlei `pcpo`-Struktur notwendig, sondern lediglich die übliche totale Ordnung der natürlichen Zahlen.

Dieser Umstand ist verantwortlich dafür, daß zum Beweis der Stetigkeit der Funktion `Ifix` die Hilfstheoreme `is_chain_iterate_lub`, `contlub_Ifix_lemma1` und `ex_lub_iterate` durch

eine Spezialargumentation hergeleitet werden müssen. Die entsprechenden Theoreme über stetige Funktionen aus der Theorie `Cont` konnten hierfür nicht verwendet werden, da sie nur im Zusammenhang mit `pcpo`-Typen anwendbar sind. Die Beweise für die Hilfstheoreme sind zwar nicht identisch zu den Beweisen in der Theorie `Cont`, verlaufen jedoch analog.

Die Monotonie und Stetigkeit des Fixpunktoperators `Ifix` werden durch die Theoreme `monofun_Ifix` und `contX_Ifix` ausgedrückt.

```
fix_eq                fix[F]=F[fix[F]]
fix_least             F[x]=x⟹fix[F] ⊑ x

fix_eq2               f ≡ fix[F]⟹f = F[f]
fix_eq3               f ≡ fix[F]⟹f[x] = F[f][x]
fix_eq4               f = fix[F]⟹f = F[f]
fix_eq5               f = fix[F]⟹f[x] = F[f][x]

fix_def2              fix[F] = lub(range(λi. iterate(i,F,⊥)))
```

Abbildung 4.55: Theoreme der Theorie `Fix` - Teil 2

In Abbildung 4.55 sind Eigenschaften des Fixpunktoperators unter Benutzung der Operation `fix` dargestellt. Sie folgen alle leicht aus den entsprechenden Eigenschaften der internen Funktion `Ifix` und deren Stetigkeit. Die Theoreme `fix_eq` und `fix_least` entsprechen der Formalisierung des Fixpunktoperators in der Logik LCF. Die Theoreme `fix_eq2` bis `fix_eq5` erleichtern die Arbeit mit Fixpunktdefinitionen für Operationen. Das Theorem `fix_def2` geht wieder über die Ausdruckskraft der Logik LCF hinaus und beschreibt die Definition des Fixpunktoperators direkt über die Kleene-Kette.

Das Theorem `fix_def2` spielt später in Kapitel 6 bei der Herleitung von Induktionsprinzipien für rekursive Datentypen eine zentrale Rolle. Gerade der Umstand, daß `fix_def2` in der schwächeren Logik LCF nicht formuliert werden kann, ist der Grund dafür, daß sich in HOLCF stärkere Induktionsprinzipien ableiten lassen als in LCF.

```
adm_def2              adm(P) = (∀Y. is_chain(Y) →
                                   (∀i.P(Y(i))) → P(lub(range(Y))))
admw_def2             admw(P) = (∀F.((∀n.P(iterate(n,F,⊥))) →
                                   P(lub(range(λi.iterate(i,F,⊥))))))
adm_impl_admw         adm(P)⟹admw(P)

fix_ind               [| adm(P);P(⊥);⋀x. P(x)⟹P(F[x])|]⟹P(fix[F])
wfix_ind              [| admw(P); ∀n. P(iterate(n,F,⊥))|]⟹P(fix[F])

adm_max_in_chain      ∀Y. is_chain(Y::nat ⇒ α) →
                         (∃n.max_in_chain(n,Y))⟹adm(P::α ⇒ bool)
adm_chain_finite      chain_finite(x::α)⟹adm(P::α ⇒ bool)
flat_imp_chain_finite flat(x::α)⟹chain_finite(x::α)
```

```
adm_flat            flat(x::α)⟹adm(P::α ⇒ bool)
flat_void           flat(⊥::void)

iso_strict          ⋀f g.⟦ ∀y.f[g[y]]=(y::β) ; ∀x.g[f[x]]=(x::α)⟧
                          ⟹f[⊥]=⊥ ∧ g[⊥]=⊥
isorep_defined      ⟦ ∀x.rep[abs[x]]=x;∀y.abs[rep[y]]=y;z¬=⊥⟧
                      ⟹rep[z]¬=⊥
isoabs_defined      ⟦ ∀x.rep[abs[x]]=x;∀y.abs[rep[y]]=y;z¬=⊥⟧
                      ⟹abs[z]¬=⊥

chfin2chfin         ⋀f g.⟦ chain_finite(x::α); ∀y.f[g[y]]=(y::β)
                          ∀x.g[f[x]]=(x::α)⟧⟹chain_finite(y::β)
flat2flat           ⋀f g.⟦ flat(x::α); ∀y.f[g[y]]=(y::β)
                          ∀x.g[f[x]]=(x::α)⟧⟹flat(y::β)

flat_codom          ⟦ flat(y::β);f[x::α]=(c::β)⟧
                      ⟹f[⊥::α]=⊥::β ∨ (∀z.f[z::α]=c)
```

Abbildung 4.56: Theoreme der Theorie `Fix` - Teil 3

Die Theoreme `adm_def2` und `admw_def2` in Abbildung 4.56 erleichtern den Zugang zu den Definitionen der Prädikate `adm` und `admw`. Das Theorem `adm_impl_admw` drückt aus, daß das Prädikat `adm` stärker als `admw` ist. Abgesehen von diesem Theorem wurde der Zusammenhang zwischen den Prädikaten `adm` und `admw` von mir nicht genauer untersucht.

Das Theoreme `fix_ind` beschreibt eine Form der Fixpunktinduktion, die allgemein als Scott-Induktion bekannt ist. Das Theorem `wfix_ind` beschreibt Fixpunktinduktion, für die die schwache Form der Zulässikeit `admw` genügt.

Die Theoreme `adm_max_in_chain` bis `adm_flat` zeigen, daß alle Prädikate über kettenendlichen bzw. flachen Typen zulässig sind. Das Theorem `flat_void` beschreibt die offensichtliche Tatsache, daß der Typ `void` flach geordnet ist.

Die Theoreme `iso_strict` bis `isoabs_defined` beschreiben Eigenschaften von zwei Operationen `abs` und `rep`, die Ordnungs-Isomorphismen sind. Solche Paare von Ordnungs-Isomorphismen spielen später in Kapitel 6 eine wichtige Rolle bei der Axiomatisierung von rekursiven Datentypen. Die Theoreme `chfin2chfin` und `flat2flat` zeigen, daß kettenendliche bzw. flache Typen unter Anwendung von Ordnungs-Isomorphismen wieder in solche übergehen.

Das letzte Theorem `flat_codom` in Abbildung 4.56 formuliert eine nützliche Eigenschaft für Operationen. Es besagt, daß nicht-strikte Operationen mit flachem Bildbereich konstant sind.

```
adm_less            ⟦ contX(u);contX(v)⟧⟹adm(λx.u(x) ⊑ v(x))
adm_conj            ⟦ adm(P); adm(Q)⟧⟹adm(λx.P(x) ∧ Q(x))
adm_cong            (∀x. P(x) = Q(x))⟹adm(P)=adm(Q)
adm_not_free        adm(λx.t)
```

```
adm_not_less        contX(t)⟹adm(λx.¬t(x) ⊑ u)
adm_all             ∀y.adm(P(y))⟹adm(λx.∀y.P(y,x))
adm_all2            ⋀y. adm(P(y)))⟹adm(λx.∀y.P(y, x)
adm_subst           ⟦ contX(t); adm(P)⟧⟹adm(λx.P(t(x)))
adm_⊥_not_less      adm(λx.¬⊥ ⊑ t(x))
adm_not_⊥           contX(t)⟹adm(λx.¬t(x) = ⊥)
adm_eq              ⟦ contX(u);contX(v)⟧⟹adm(λx.u(x)= v(x))

adm_disj_lemma11    ⟦ adm(P); is_chain(Y); ∀i. ∃j. i < j ∧ P(Y(j))⟧
                      ⟹P(lub(range(Y)))
adm_disj_lemma12    ⟦ adm(P); is_chain(Y);∃i. ∀j. i < j → P(Y(j))⟧
                      ⟹P(lub(range(Y)))

adm_disj            ⟦ adm(P); adm(Q)⟧⟹adm(λx.P(x) ∨ Q(x))
adm_impl            ⟦ adm(λx.¬P(x)); adm(Q)⟧⟹adm(λx.P(x) → Q(x))

val adm_thms = [adm_impl,adm_disj,adm_eq,adm_not_⊥,adm_⊥_not_less,
                adm_all2,adm_not_less,adm_not_free,adm_conj,adm_less];
```

Abbildung 4.57: Theoreme der Theorie `Fix` - Teil 4

In Abbildung 4.57 sind ausschließlich Theoreme zur Propagierung der Zulässigkeit aufgelistet. Ähnlich wie bei den Theoremen zur Propagierung der Stetigkeit in LCF-Termen aus Theorie `Cfun3` wird die λ-Abstraktion für Funktionen dazu eingesetzt, um die Zulässigkeit von Prädikaten in freien Variablen zu formalisieren. Die Theoreme zeigen deutlich, daß die Zulässigkeit von Prädikaten eng mit der Stetigkeit der beteiligten Teilterme verknüpft ist.

Bis auf den Beweis des Theorems `adm_disj` waren alle diesbezüglichen Beweise einfach zu führen. Der Beweis für `adm_disj` erforderte jedoch wesentlich mehr Aufwand. Der Grund hierfür liegt darin, daß bei der Disjunktion zulässiger Prädikate mehrere Fälle unterschieden werden müssen, in denen zum Teil die Konstruktion von neuen Teil-Ketten erforderlich ist. In Hilfstheoremen mußte jeweils mühsam argumentiert werden, daß in den einzelnen Fällen die kleinsten oberen Schranken der Teil-Ketten mit denen der gesamten Kette übereinstimmen.

Wie bei der Propagierung der Stetigkeit werden die einzelnen Zulässigkeitstheoreme zu einer Liste `adm_thms` zusammengefaßt, deren iterierte Anwendung mittels der Wiederholungsanweisung `REPEAT` bei einer Vielzahl von Prädikaten automatisch die Zulässigkeit des Prädikats beweist.

4.12.3 Beweise für ausgesuchte Theoreme

In diesem Abschnitt werden ausgesuchte Beweise für Theoreme der Theorie `Fix` vorgestellt. Die Wahl fiel auf die Theoreme `Ifix_eq`, `Ifix_least`, `contlub_iterate`, `contlub_Ifix` und `fix_ind`, die die typische Argumentationsweise im Zusammenhang mit dem Fixpunktoperator `Ifix` verdeutlichen.

4.12.3.1 Beweis für das Theorem `Ifix_eq`

Das Theorem `Ifix_eq` beweist, daß die Funktion `Ifix` ein Fixpunktoperator ist. Wir beginnen, indem wir uns das Hauptbeweisziel vorgeben. Dabei expandieren wir schon im Aufruf die Definition des Funktionals `Ifix` mittels `goalw`:

```
val prems = goalw Fix.thy  [Ifix_def] "Ifix(F)=F[Ifix(F)]";
```

Wir erhalten dadurch folgendes initiale Beweisziel:

```
 1. lub(range(λi. iterate(i, F, ⊥))) = F[lub(range(λi. iterate(i, F, ⊥)))]
```

Auf der rechten Seite der Gleichung läßt sich das Theorem `contlub_cfun_arg`

```
 is_chain(?x1)⟹?fo4[lub(range(?x1))] = lub(range(λi. ?fo4[?x1(i)]))
- by (rtac (contlub_cfun_arg RS ssubst) 1);
```

verbunden mit Gleichungslogik anwenden. Dies führt zu:

```
 1. is_chain(λi. iterate(i, F, ⊥))
 2. lub(range(λi. iterate(i, F, ⊥))) = lub(range(λi. F[iterate(i, F, ⊥)]))
```

Das erste Teilziel lösen wir direkt durch das Theorem

```
 is_chain(λi. iterate(i, ?F, ⊥))
- by (rtac is_chain_iterate 1);
```

Das zweite Teilziel spalten wir per Antisymmetrie in zwei kleinere Ziele auf. Dies führt zu:

```
- by (rtac antisym_less 1);

 1. lub(range(λi. iterate(i, F, ⊥))) ⊑ lub(range(λi. F[iterate(i, F, ⊥)]))
 2. lub(range(λi. F[iterate(i, F, ⊥)])) ⊑ lub(range(λi. iterate(i, F, ⊥)))
```

Wir lösen zunächst Teilziel 1 und unterdrücken solange die Darstellung des zweiten Teilziels. Im ersten Teilziel wenden wir die Majorantenregel

```
 ⟦ is_chain(?C1.0); is_chain(?C2.0); ∀k. ?C1.0(k) ⊑ ?C2.0(k)⟧⟹
    lub(range(?C1.0)) ⊑ lub(range(?C2.0))
- by (rtac lub_mono 1);
```

an. Dies führt zu drei neuen Teilzielen:

```
 1. is_chain(λi. iterate(i, F, ⊥))
 2. is_chain(λi. F[iterate(i, F, ⊥)])
 3. ∀k. iterate(k, F, ⊥) ⊑ F[iterate(k, F, ⊥)]
```

Die ersten beiden betreffen Ketteneigenschaften. Sie werden im wesentlichen über das Theorem `is_chain_iterate`

```
 is_chain(λi. iterate(i, ?F, ⊥))
```

gelöst. Im dritten Teilziel reicht es, die Behauptung für ein beliebiges, aber festes `k` zu zeigen.

```
- by (rtac is_chain_iterate 1);
- by (rtac ch2ch_fappR 1);
- by (rtac is_chain_iterate 1);
```

```
- by (rtac allI 1);
```

Wir erhalten nach Anwendung des obigen Skripts den Beweiszustand:

```
 1. ⋀k. iterate(k, F, ⊥) ⊑ F[iterate(k, F, ⊥)]
```

Auf der rechten Seite können wir die Eigenschaft

```
 iterate(Suc(?n), ?F, ?x) = ?F[iterate(?n, ?F, ?x)]
- by (rtac (iterate_Suc RS subst) 1);
```

der Funktion `iterate` rückwärts anwenden. Wir erhalten damit:

```
 1. ⋀k. iterate(k, F, ⊥) ⊑ iterate(Suc(k), F, ⊥)
```

Dieses Teilziel wird aber wieder im wesentlichen über das Theorem `is_chain_iterate`

```
 is_chain(λi. iterate(i, ?F, ⊥))
- by (rtac (is_chain_iterate RS is_chainE RS spec) 1);
```

gelöst. Wir können uns nun dem zweiten Ziel zuwenden, das wir bisher zurückgestellt haben. Es lautet:

```
 1. lub(range(λi. F[iterate(i, F, ⊥)])) ⊑ lub(range(λi. iterate(i, F, ⊥)))
```

Wir nützen aus, daß kleinste obere Schranken kleiner als alle anderen oberen Schranken sind. Anwendung des Theorems

```
 ⟦ is_chain(?S6); range(?S6) <| ?x1⟧⟹lub(range(?S6)) ⊑ ?x1
- by (rtac is_lub_thelub 1);
```

liefert die Teilziele:

```
 1. is_chain(λi. F[iterate(i, F, ⊥)])
 2. range(λi. F[iterate(i, F, ⊥)]) <| lub(range(λi. iterate(i, F, ⊥)))
```

Das erste lösen wir über Verwendung des Theorems `is_chain_iterate`

```
 is_chain(λi. iterate(i, ?F, ⊥))
- by (rtac ch2ch_fappR 1);
- by (rtac is_chain_iterate 1);
```

Im zweiten Teilziel expandieren wir die Definition für obere Schranken:

```
 ∀i. ?S(i) ⊑ ?x⟹range(?S) <| ?x
- by (rtac ub_rangeI 1);
- by (rtac allI 1);

 1. ⋀i. F[iterate(i, F, ⊥)] ⊑ lub(range(λi. iterate(i, F, ⊥)))
```

Jetzt können wir auf der linken Seite wieder eine der Reduktionseigenschaften des Funktionals `iterate` rückwärts anwenden. Anwendung der Regel

```
 ?P(iterate(Suc(?n1), ?F1, ?x1))⟹?P(?F1[iterate(?n1, ?F1, ?x1)])
- by (rtac (iterate_Suc RS subst) 1);
```

liefert den Beweiszustand:

```
 1. ⋀i. iterate(Suc(i), F, ⊥) ⊑ lub(range(λi. iterate(i, F, ⊥)))
```

Diese Behauptung ist aber leicht zu zeigen, denn die kleinste obere Schranke auf der rechten Seite ist insbesondere auch ein obere Schranke. Mittels

```
 is_chain(?S1)⟹?S1(?x) ⊑ lub(range(?S1))
- by (rtac is_ub_thelub 1);
```

und

```
 is_chain(λi. iterate(i, ?F, ⊥))
- by (rtac is_chain_iterate 1);
```

lösen wir das Beweisziel vollständig.

4.12.3.2 Beweis für das Theorem `Ifix_least`

Das Theorem `Ifix_least` besagt, daß `Ifix` den kleinsten Fixpunkt erzeugt. Zu Beginn geben wir uns das initiale Beweisziel vor. Wir expandieren wieder die Definition der Konstanten `Ifix` schon im Aufruf:

```
- val prems = goalw Fix.thy [Ifix_def]  "F[x]=x⟹Ifix(F) ⊑ x";
- by (cut_facts_tac prems 1);
```

Dies liefert das Beweisziel:

```
 1. F[x] = x⟹lub(range(λi. iterate(i, F, ⊥))) ⊑ x
```

Zuerst nützen wir aus, daß kleinste obere Schranken kleiner als alle oberen Schranken sind. Wir verwenden

```
 ⟦ is_chain(?S6); range(?S6) <| ?x1⟧⟹lub(range(?S6)) ⊑ ?x1
- by (rtac is_lub_thelub 1);
```

und erhalten zwei neue Teilziele:

```
 1. F[x] = x⟹is_chain(λi. iterate(i, F, ⊥))
 2. F[x] = x⟹range(λi. iterate(i, F, ⊥)) <| x
```

Das erste lösen wir direkt über das Theorem

```
 is_chain(λi. iterate(i, ?F, ⊥))
- by (rtac is_chain_iterate 1);
```

Im zweiten expandieren wir die Definition für obere Schranken. Anwendung von

```
 ∀i. ?S(i) ⊑ ?x⟹range(?S) <| ?x
- by (rtac ub_rangeI 1);
```

auf das zweite Ziel liefert:

```
 1. F[x] = x⟹∀i. iterate(i, F, ⊥) ⊑ x
```

Diese Behauptung beweisen wir durch strukturelle Induktion über dem Typ der natürlichen Zahlen `nat`:

```
- by (strip_tac 1);
- by (nat_ind_tac "i" 1);
```

Wir erhalten die beiden neuen Teilziele:

```
 1. ⋀i. F[x] = x⟹iterate(0, F, ⊥) ⊑ x
 2. ⋀i i1.
       ⟦ F[x] = x; iterate(i1, F, ⊥) ⊑ x⟧⟹iterate(Suc(i1), F, ⊥) ⊑ x
```

Das erste Teilziel lösen wir vollständig durch Termsimplifikation bezüglich der Reduktionseigenschaften der Funktion `iterate`. Das zweite können wir ebenfalls durch Anwendung des Simplifikators vereinfachen. Nach Anwendung der Simplifikation verbleibt:

```
- by (asm_simp_tac iterate_ss 1);
- by (asm_simp_tac iterate_ss 1);

 1. ⋀i i1.
       ⟦ F[x] = x; iterate(i1, F, ⊥) ⊑ x⟧⟹F[iterate(i1, F, ⊥)] ⊑ x
```

Jetzt verwenden wir die Annahme, daß `x` ein Fixpunkt ist. Anwendung von Gleichungslogik liefert:

```
- by (eres_inst_tac [("t","x")] subst 1);

 1. ⋀i i1. iterate(i1, F, ⊥) ⊑ x⟹F[iterate(i1, F, ⊥)] ⊑ F[x]
```

Dieses letzte Teilziel lösen wir leicht durch Anwendung der Monotonieregel für Operationen

```
 ?x2 ⊑ ?x1⟹?fo5[?x2] ⊑ ?fo5[?x1]
- by (etac monofun_cfun_arg 1);

No subgoals
```

4.12.3.3 Beweis für das Theorem `contlub_iterate`

Dieses Theorem zeigt in Verbindung mit dem Theorem `monofun_iterate`, daß der Iterator `iterate` im 'zweiten' Argument stetig ist. Mit seiner Hilfe kann dann später leicht die Stetigkeit des Fixpunktoperators `Ifix` gezeigt werden. Wir beginnen den Beweis, indem wir den initialen Beweiszustand erzeugen. Dabei expandieren wir gleich die Definition des Prädikats `contlub`.

```
- val prems = goalw Fix.thy  [contlub] "contlub(iterate(i))";

 1. ∀Y. is_chain(Y) →
         iterate(i, lub(range(Y))) = lub(range(λia. iterate(i, Y(ia))))
```

Gemäß der Methodik des natürlichen Schließens beweisen wir die Behauptung für ein beliebiges, aber festes `Y` und übernehmen die Prämisse der Implikation in den Annahmenkontext. Dies führt zu:

```
- by (strip_tac 1);

 1. ⋀Y. is_chain(Y)⟹
        iterate(i, lub(range(Y))) = lub(range(λia. iterate(i, Y(ia))))
```

Diese Behauptung beweisen wir durch strukturelle Induktion über `i`, was uns zwei neue Teilziele beschert:

```
- by (nat_ind_tac "i" 1);

 1. ⋀Y. is_chain(Y)⟹
        iterate(0, lub(range(Y))) = lub(range(λi. iterate(0, Y(i))))
 2. ⋀Y i1.
    [| is_chain(Y);
    iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))|]⟹
    iterate(Suc(i1), lub(range(Y))) = lub(range(λi. iterate(Suc(i1), Y(i))))
```

Die Induktionsbasis beweisen wir leicht durch Simplifikation bezüglich der Reduktionseigenschaften von `iterate` und durch das Theorem `lub_const`

```
 range(λx. ?c) ≪| ?c
- by (asm_simp_tac iterate_ss 1);
- by (rtac (lub_const RS thelubI RS sym) 1);
```

über kleinste obere Schranken von konstanten Ketten. Es verbleibt der Induktionsschritt:

```
 1. ⋀Y i1.
    [| is_chain(Y);
    iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))|]⟹
    iterate(Suc(i1), lub(range(Y))) = lub(range(λi. iterate(Suc(i1), Y(i))))
```

Hier wenden wir wieder den Simplifikator an, der zuerst die Reduktionseigenschaften von `iterate` benutzt und dann im Ergebnis die Induktionshypothese anwendet. Dabei werden automatisch Kongruenzeigenschaften mitverwendet. Dies führt zu:

```
- by (asm_simp_tac iterate_ss 1);

 1. ⋀Y i1.
       [| is_chain(Y);
          iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))|]⟹
       (λu. lub(range(Y))[lub(range(λi. iterate(i1, Y(i))))(u)]) =
       lub(range(λi u. Y(i)[iterate(i1, Y(i), u)]))
```

Wir müssen zeigen, daß zwei Funktionen gleich sind. Daher verwenden wir im nächsten Schritt die Extensionalitätsregel. Dies liefert:

```
- by (rtac ext 1);
 1. ⋀Y i1 u.
       [| is_chain(Y);
          iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))|]⟹
       lub(range(Y))[lub(range(λi. iterate(i1, Y(i))), u)] =
       lub(range(λi u. Y(i)[iterate(i1, Y(i), u)]), u)
```

Auf der rechten Seite der Gleichung wird die kleinste obere Schranke einer Kette von Funktionen angewendet. Durch Verwendung der Regel

```
 is_chain(?S1)⟹lub(range(?S1)) = (λx. lub(range(λi. ?S1(i, x))))
- by (rtac (thelub_fun RS ssubst) 1);
```

und Gleichungslogik erhalten wir die beiden neuen Ziele:

```
 1. ⋀Y i1 u.
       ⟦ is_chain(Y);
          iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))⟧⟹
       is_chain(λi u. Y(i)[iterate(i1, Y(i), u)])
 2. ⋀Y i1 u.
       ⟦ is_chain(Y);
          iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))⟧⟹
       lub(range(Y))[lub(range(λi. iterate(i1, Y(i))), u)] =
       lub(range(λi. Y(i)[iterate(i1, Y(i), u)]))
```

Im ersten Ziel muß eine Ketteneigenschaft gezeigt werden, die sich im wesentlichen durch die Monotonie der Applikationsfunktion `fapp` und der Iteratorfunktion `iterate` beweisen läßt. Eine, wenn auch etwas längere, Standardargumentation löst dieses Beweisziel vollständig. Wir wenden das entsprechende Skript

```
- by (rtac is_chainI 1);
- by (rtac allI 1);
- by (rtac (less_fun RS iffD2) 1);
- by (rtac allI 1);
- by (rtac (is_chainE RS spec) 1);
- by (rtac (monofun_fapp1 RS ch2ch_MF2LR) 1);
- by (rtac allI 1);
- by (rtac monofun_fapp2 1);
- by (atac 1);
- by (rtac ch2ch_fun 1);
- by (rtac (monofun_iterate RS ch2ch_monofun) 1);
- by (atac 1);
```

an und können uns dann dem verbleibenden Beweisziel zuwenden:

```
 1. ⋀Y i1 u.
       ⟦ is_chain(Y);
          iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))⟧⟹
       lub(range(Y))[lub(range(λi. iterate(i1, Y(i))), u)] =
       lub(range(λi. Y(i)[iterate(i1, Y(i), u)]))
```

Auf der linken Seite der Gleichung wird in einem Teilterm die kleinste obere Schranke einer Kette von Funktionen angewendet. Daher wenden wir erneut die Regel

```
 is_chain(?S1)⟹lub(range(?S1)) = (λx. lub(range(λi. ?S1(i, x))))
- by (rtac (thelub_fun RS ssubst) 1);
```

in Verbindung mit Gleichungslogik an. Dies führt zu:

```
1. ⋀Y i1 u.
      ⟦ is_chain(Y);
         iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))⟧⟹
      is_chain(λi. iterate(i1, Y(i)))
2. ⋀Y i1 u.
      ⟦ is_chain(Y);
         iterate(i1, lub(range(Y))) = lub(range(λi. iterate(i1, Y(i))))⟧⟹
      lub(range(Y))[lub(range(λi. iterate(i1, Y(i), u)))] =
      lub(range(λi. Y(i)[iterate(i1, Y(i), u)]))
```

Das erste Ziel lösen wir leicht durch Verwendung der Monotonie des Iterators `iterate`.

```
 is_chain(?Y)⟹is_chain(λi. iterate(?i1, ?Y(i)))
- by (etac (monofun_iterate RS ch2ch_monofun) 1);
```

Im verbleibenden Ziel wenden wir die Regel

```
 ⟦ is_chain(?FY); is_chain(?TY)⟧⟹
   lub(range(?FY))[lub(range(?TY))] = lub(range(λi. ?FY(i)[?TY(i)]))
- by (etac contlub_cfun  1);
```

an. Das erste neu entstehende Teilziel lösen wir direkt per Annahme, das zweite Ziel löst sich wieder über die Anwendung der Monotonie von `iterate`.

```
- by (etac (monofun_iterate RS ch2ch_monofun RS ch2ch_fun) 1);
No subgoals
```

4.12.3.4 Beweis für das Theorem `contlub_Ifix`

Das Theorem `contlub_Ifix` zeigt zusammen mit dem Theorem `monofun_Ifix`, daß der Fixpunktoperator `Ifix` eine stetige Funktion ist. Wir erzeugen uns den initialen Beweiszustand mittels

```
- val prems = goalw Fix.thy [contlub,Ifix_def] "contlub(Ifix)";
```

wobei wir schon im Aufruf die Definitionen für das Prädikat `contlub` und den Fixpunktoperator `Ifix` entfalten. Dies liefert:

```
1. ∀Y. is_chain(Y) →
        lub(range(λi. iterate(i, lub(range(Y)), ⊥))) =
        lub(range(λi. lub(range(λia. iterate(ia, Y(i), ⊥)))))
```

Diese Behauptung zeigen wir, indem wir den Beweis für ein beliebiges, aber festes `Y` führen und die Prämisse der Implikation in den Annahmenkontext übernehmen. Wir erhalten:

```
- by (strip_tac 1);

1. ⋀Y. is_chain(Y)⟹
        lub(range(λi. iterate(i, lub(range(Y)), ⊥))) =
        lub(range(λi. lub(range(λia. iterate(ia, Y(i), ⊥)))))
```

Jetzt setzen wir das Hilfstheorem `contlub_Ifix_lemma1`
```
 is_chain(?Y)==>
 iterate(?n, lub(range(?Y)), ?y) = lub(range(λi. iterate(?n, ?Y(i), ?y)))
```

ein, welches unmittelbar aus dem Theorem `contlub_iterate` folgt. In Verbindung mit Gleichungslogik erhalten wir somit:
```
- by (etac (contlub_Ifix_lemma1 RS ext RS ssubst) 1);

 1. ⋀Y. is_chain(Y)==>
         lub(range(λi. lub(range(λia. iterate(i, Y(ia), ⊥))))) =
         lub(range(λi. lub(range(λia. iterate(ia, Y(i), ⊥)))))
```

Diese Behauptung lösen wir vollständig, indem wir das Hilfstheorem
```
 is_chain(?Y)==>
 lub(range(λi. lub(range(λia. iterate(i, ?Y(ia), ⊥))))) =
 lub(range(λi. lub(range(λia. iterate(ia, ?Y(i), ⊥)))))
- by (etac ex_lub_iterate 1);
```

einsetzen. Dieses Hilfstheorem zum Verstauschen der `lub`-Anwendungen wurde analog zum Vertauschungstheorem `exlub_MF2` aus Theorie `Cont` bewiesen.

4.12.3.5 Beweis für das Theorem `fix_ind`

Das Theorem `fix_ind` beschreibt das Prinizip der Scott-Induktion. Es folgt leicht aus den Eigenschaften des Fixpunktoperators `fix` und des Prädikats `adm`. Wir beginnen, indem wir den anfänglichen Beweiszustand erzeugen.
```
- val prems = goal  Fix.thy
     "⟦ adm(P);P(⊥);⋀x. P(x)==>P(F[x])⟧==>P(fix[F])";
- by (cut_facts_tac prems 1);
```

Dies liefert uns den Beweiszustand:
```
 1. ⟦ adm(P); P(⊥)⟧==>P(fix[F])
```

Die dritte Annahme
`⋀x. P(x)==>P(F[x])`

kann durch Anwendung des Kommandos `by (cut_facts_tac prems 1)` nicht sichtbar gemacht werden, ist aber trotzdem über die Liste der initialen Annahmen `prems` erreichbar. Wir werden später davon Gebrauch machen. Wir verwenden das Theorem
```
 fix[?F] = lub(range(λi. iterate(i, ?F, ⊥)))
- by (rtac (fix_def2 RS ssubst) 1);
```

um die Definition des Fixpunktoperators `fix` zu entfalten. Dies führt zu:
```
 1. ⟦ adm(P); P(⊥)⟧==>P(lub(range(λi. iterate(i, F, ⊥))))
```

Im nächsten Schritt benutzen wir die Definition der Zulässigkeit. Anwendung des Theorems

```
adm_def2
 adm(?P) = (∀Y. is_chain(Y) → (∀i. ?P(Y(i))) → ?P(lub(range(Y))))
- by (rtac (adm_def2 RS iffD1 RS spec RS mp RS mp) 1);
```

in Verbindung mit Gleichungslogik liefert die neuen Teilziele:

```
 1. ⟦ adm(P); P(⊥)⟧⟹adm(P)
 2. ⟦ adm(P); P(⊥)⟧⟹is_chain(λi. iterate(i, F, ⊥))
 3. ⟦ adm(P); P(⊥)⟧⟹∀i. P(iterate(i, F, ⊥))
```

Das erste lösen wir direkt per Annahme, für das zweite verwenden wir das Theorem `is_chain_iterate`

```
 is_chain(λi. iterate(i, ?F, ⊥))
- by (atac 1);
- by (rtac is_chain_iterate 1);
```

Somit verbleibt nur das letzte Teilziel:

```
 1. ⟦ adm(P); P(⊥)⟧⟹∀i. P(iterate(i, F, ⊥))
```

Dieses beweisen wir durch Induktion über `i`.

```
- by (rtac allI 1);
- by (nat_ind_tac "i" 1);
```

Es entstehen die beiden neuen Teilziele:

```
 1. ⋀i. ⟦ adm(P); P(⊥)⟧⟹P(iterate(0, F, ⊥))
 2. ⋀i i1.
       ⟦ adm(P); P(⊥); P(iterate(i1, F, ⊥))⟧⟹P(iterate(Suc(i1), F, ⊥))
```

Das erste lösen wir durch Termsimplifikation und Beweis per Annahme. Im zweiten vereinfachen wir ebenfalls mittels der Reduktionseigenschaften des Iterators.

```
- by (asm_simp_tac iterate_ss 1);
- by (asm_simp_tac iterate_ss 1);
```

Dies führt zu:

```
 1. ⋀i i1.
       ⟦ adm(P); P(⊥); P(iterate(i1, F, ⊥))⟧⟹P(F[iterate(i1, F, ⊥)])
```

Nun können wir aber die ursprüngliche Annahme

```
⋀x. P(x)⟹P(F[x])
- by (resolve_tac prems 1);
```

einsetzten, was folgendes, aufgrund der Induktionshypothese triviale Ziel

```
 1. ⋀i i1. ⟦ adm(P); P(⊥); P(iterate(i1, F, ⊥))⟧⟹P(iterate(i1, F, ⊥))
```

ergibt.

```
- by (atac 1);
No subgoals
```

4.13 Operationen für Identität und Komposition

Im letzten Schritt der Entwicklung von HOLCF werden die Theorien aus den Abschnitten 4.8 bis 4.12 vereinigt. Zusätzlich werden noch Operationen für die Identität und die Komposition von Operationen eingeführt.

4.13.1 Die Theorie ccc1

Die Theorie ccc1 ist in Abbildung 4.58 dargestellt. Ursprünglich war geplant, eine Reihe von Theorien zu entwickeln, die zeigen, daß die Menge der Typen in der Klasse pcpo als Objekte zusammen mit der Menge der Operationen auf pcpo-Typen als Morphismen eine kartesisch abgeschlossene Kategorie (cartesian closed categorie) bilden. Die Theorie ccc1 ist die erste Theorie in dieser Reihe, und daher resultiert auch ihr Name. Sie weist nach, daß das obige Paar eine Kategorie bildet. Aus Zeitgründen habe ich bisher jedoch auf die Entwicklung der anderen Theorien zum Nachweis der kartesischen Abgeschlossenheit verzichtet.

```
ccc1 = Cprod3 + Sprod3 + Ssum3 + Lift3 + Fix +

consts
        ID         ::  α → α
         @oo       ::  (β → γ) ⇒ (α → β) ⇒ α → γ   ( _ oo _   [101,100]100)
         cop @oo   ::  (β → γ) → (α → β) → α → γ   ( cfcomp )

translations  f1 oo f2  ⇒  cfcomp[f1][f2]

rules

  ID_def          ID ≡ (Λx.x)
  oo_def          cfcomp ≡ (Λf g x.f[g[x]])

end
```

Abbildung 4.58: Theorie ccc1

Die Operation ID steht für die Identitäts-Operation, die Operation oo steht für die Komposition von Operationen. Die Infix-Operation oo wird mit dem bereits bekannten Isabelle-Mechanismus als nach rechts bindender Infix-Operator eingeführt. Die Definitionen sind jeweils offensichtlich.

4.13.2 Theoreme der Theorie ccc1

Die Theoreme der Theorie ccc1 sind in Abbildung 4.59 dargestellt. Die Theoreme zeigen, daß ID und oo als Identität und Komposition für Morphismen in der oben angesprochenen

Kategorie geeignet sind. Insbesondere sind durch die Theoreme `ID2` und `ID3` die Identitätsgesetze gezeigt, und `assoc_oo` weist das Assoziativitätsgesetz der Komposition nach. Natürlich sind die Theoreme auch ohne den kategoriellen Hintergrund interessant.

Als letztes wird mit der Vereinbarung von `ccc1_ss` der Simplifikator des Isabelle-Systems konfiguriert.

```
ID1       ID[x]=x
cfcomp1   (f oo g)=(Λx.f[g[x]])
cfcomp2   (f oo g)[x]=f[g[x]]

ID2       f oo ID = f
ID3       ID oo f = f

assoc_oo  f oo (g oo h) = (f oo g) oo h

val ccc1_ss = Cfun_ss
              addsimps Cprod_rews
              addsimps Sprod_rews
              addsimps Ssum_rews
              addsimps lift_rews
              addsimps [ID1,ID2,ID3,cfcomp2];
```

Abbildung 4.59: Theoreme der Theorie `ccc1`

In Kapitel 5 werde ich in einem leicht veränderten Kontext von den eben angesprochenen kategoriellen Eigenschaften Gebrauch machen. Dort wird die gleiche Menge der Objekte verwendet aber statt der Menge aller Operationen nur die Menge der strikten Operationen als Menge der Morphismen gewählt. Auch in dieser Konstellation können `ID` als Identität und `oo` als Komposition verwendet werden, da sie die Striktheit von Operationen respektieren.

Kapitel 5

Formalisierung von Bereichsgleichungen in HOLCF

In diesem Kapitel werde ich die Formalisierung rekursiver Typen der Klasse `pcpo` als initiale Lösungen von Bereichsgleichungen beschreiben. Die Formalisierung erfolgt dabei, wie in der Logik LCF, durch einen speziellen Satz von Axiomen, deren Form sich aus der jeweiligen Bereichsgleichung ableiten läßt. Wie bei den in Abschnitt 2.6 vorgestellten Erweiterungsmechanismen stellt sich auch hier die Frage nach der Konservativität der Theorieerweiterung.

Die Lösung von Bereichsgleichungen wird ausführlich in der Literatur zur Bereichstheorie diskutiert. Neben den frühen Arbeiten im Umfeld von Dana Scott [SS71, Sco72, Sco73, Sco76] und einigen nicht-kategoriellen Präsentationen [Sto77, Sch86] wird das Thema vorwiegend in der Kategorientheorie behandelt [KK92, AL91, Gun92]. All diesen Arbeiten ist gemeinsam, daß sie, wenn überhaupt, nur wenig Hinweise geben, ob und wie die initialen Lösungen von Bereichsgleichungen in einer Logik wie LCF oder HOLCF axiomatisiert werden können. Die dort verwendete Notation geht weit über die Ausdrucksmächtigkeit von LCF hinaus und ist auch zum größten Teil für die Logik HOLCF unbrauchbar, da die bei allen Behandlungen verwendete Konstruktion des Inversen-Limes (Co-Limes) den Einsatz von Dependent-Types erfordert. Lediglich in [KK92] erfolgt am Rande ein Hinweis auf die Axiomatisierung von rekursiven Bereichen, die durch covariante Funktoren charakterisiert werden. Für den allgemeinen Fall von Funktoren mit contravarianten Anteilen fehlt aber auch hier eine Unterstützung.

Auf der anderen Seite gibt es spezifische Literatur zur Logik LCF [Pau84, Mon85, Pau87], in der bestimmte Axiome zur Formalisierung von rekursiven Typen einfach verwendet werden, ohne diese Axiome explizit zu rechtfertigen. Gerade in [Pau87] erfolgt die Axiomatisierung gänzlich unmotiviert und nicht formal fundiert. Dies soll nicht etwa heißen, daß diese Arbeiten irgendwelche Fehler enthielten, ich möchte an dieser Stelle lediglich die fehlende Motivation für die verwendeten Axiome bemängeln.

Ziel dieses Kapitels ist die Darstellung einer kategoriellen Theorie für die Lösung von Bereichsgleichungen, die die unmittelbare Anwendung ihrer Ergebnisse auf die Logiken HOLCF bzw. LCF erlaubt. Speziell soll in dieser Darstellung auch die Form der zur Axiomatisierung der initialen Lösungen eingesetzten Axiome motiviert werden. Dabei werden keine neuen technischen Ergebnisse präsentiert, vielmehr werden bereits bekannte kategorielle Techniken aufbereitet und deren praktischer Einsatz in der Logik HOLCF demonstriert.

Die hier vorgestellte kategorielle Behandlung von Bereichsgleichungen stützt sich auf Arbeiten von Peter Freyd [Fre90, Fre91, Fre92] ab. Diese Arbeiten erlangen ihre praktische Relevanz durch den Umstand, daß die initialen Lösungen von Bereichsgleichungen nicht wie üblich durch globale Kriterien mittels der Co-Limes Konstruktion charakterisiert sind, sondern vielmehr lokale Kriterien angegeben werden, die unmittelbar als Axiomatisierung in einer Logik wie HOLCF verwendet werden können. Diese Erkenntnis wird in den Arbeiten von Freyd jedoch nicht ausdrücklich betont und bleibt dem kategoriell unbedarften Leser mit Sicherheit verborgen, da die Darstellung der Inhalte auf einem sehr abstrakten kategoriellen Niveau erfolgt.

Mein Zugang zum Inhalt von Freyds Arbeiten erfolgte ausschließlich durch eine Vorlesung über Bereichstheorie, die Thomas Streicher im Wintersemester 1993/94 an der Ludwig-Maximilians Universität München hielt. In den Vorlesungsnotizen, die Thomas Streicher mir freundlicherweise zur Verfügung stellte, wird in verständlicher und eingängiger Form die kategorielle Theorie zur Formalisierung und Lösung von Bereichsgleichungen dargestellt. Im ersten Teil der Notizen werden rekursive Bereiche behandelt, die durch covariante Funktoren charakterisiert werden können. Im zweiten Teil wird dann der allgemeine Fall von Funktoren mit covarianten und contravarianten Anteilen (mixed-variant functors) abgehandelt.

In Abschnitt 5.1 werde ich mich auf den einfachen Fall der ausschließlich covarianten Funktoren beschränken, da in Kapitel 6 nur solche rekursiven Typen behandeln werden, die sich durch covariante Funktoren beschreiben lassen. Durch diese Beschränkung kommen die Ergebnisse von Freyds Arbeit zwar nicht voll zum Tragen, die Zweckmäßigkeit seines Ansatzes wird aber auch im einfachen Fall der covarianten Funktoren deutlich. Die Darstellung der Theorie in Abschnitt 5.1 orientiert sich eng an den Notizen von Thomas Streicher. In Abschnitt 5.2 werde ich zeigen, wie die Ergebnisse der abstrakten kategoriellen Behandlung auf die Logik HOLCF angewendet werden können. In Abschnitt 5.3 wird dann die konservative Theorieerweiterung durch rekursive Typen im Stil von Abschnitt 2.6 formuliert.

5.1 Kategorielle Behandlung von Bereichsgleichungen

Zum Verständnis dieses Abschnitts sind nur grundlegende Kenntnisse der Kategorientheorie erforderlich. Vorausgesetzt wird die Kenntnis der Begriffe Kategorie, Objekte der Kategorie, Morphismen der Kategorie, initiales und terminales Objekt und Funktor auf einer Kategorie. Eine Definition dieser Begriffe findet sich auf den ersten Seiten jeder Einführung in die Kategorientheorie [AL91, LS86].Die restlichen Begriffe werden eingeführt.

Rekursive Gleichungen für stetige Funktionen löst man dadurch, daß man den kleinsten Fixpunkt des zugehörigen Funktionals konstruiert. Der kleinste Fixpunkt eines Funktionals ist zugleich auch der kleinste Präfixpunkt des Funktionals. Die Charakterisierung der gesuchten Lösung für die rekursive Gleichung als kleinster Präfixpunkt des zugehörigen Funktionals liefert die Motivation für die in diesem Abschnitt dargestellte kategorielle Verallgemeinerung. Aus dem Bereich mit cpo-Struktur wird eine CPO-Kategorie, die stetige Endofunktion wird zum (lokal) stetigen Endofunktor auf der Kategorie. Präfixpunkte bzgl. der stetigen Funktion werden zu sogenannten F-Algebren bzgl. des Funktors F abstrahiert, und der kleinste Präfixpunkt wird zur initialen F-Algebra in der Kategorie der F-Algebren.

Definition 5.1 *F-Algebra, F-Homomorphismus, F-Alg*

Sei $\mathcal{C}$ eine Kategorie und $F : \mathcal{C} \to \mathcal{C}$ ein Endofunktor auf $\mathcal{C}$. Eine F-Algebra bzgl. Funktor F ist ein Paar (A, θ) mit $\theta \in \mathcal{C}(FA, A)$[1]. Ein F-Homomorphismus zwischen F-Algebren (A, θ) und (B, φ) ist ein Morphismus $h \in \mathcal{C}(A, B)$, so daß $h \circ \theta = \varphi \circ Fh$. Graphisch bedeutet dies, daß folgendes Diagramm kommutiert:

$$\begin{array}{ccc} FA & \xrightarrow{\theta} & A \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle h} \\ FB & \xrightarrow[\varphi]{} & B \end{array} \qquad h \circ \theta = \varphi \circ Fh$$

F-Algebren und F-Homomorphismen bzgl. eines Endofunktors F bilden eine Kategorie F-Alg, deren Objekte die F-Algebren sind, und deren Morphismen die F-Homomorphismen sind.

Für die initiale F-Algebra in der Kategorie F-Alg gilt folgendes Theorem.

Theorem 5.1

Wenn die F-Algebra (A, θ) initial in F-Alg ist, dann ist θ ein Isomorphismus, d.h. es gibt ein $\bar{\theta} : A \to FA$ mit $\theta \circ \bar{\theta} = id_A$ und $\bar{\theta} \circ \theta = id_{FA}$. Da in diesem Fall $\bar{\theta}$ eindeutig ist, wird es i.a. mit θ^{-1} bezeichnet.

Der Beweis für dieses Theorem findet sich u.a. in [Gun92, S. 317 ff.]. Er sei hier kurz dargestellt. Betrachten wir dazu das untenstehende Diagramm:

Aus der Initialität von (A, θ) folgt, daß h existiert und eindeutig bestimmt ist. Daraus läßt sich ableiten, daß alle Teildiagramme des Diagramms kommutieren:

oberes Diagramm:

$$F\theta \circ Fh = h \circ \theta$$

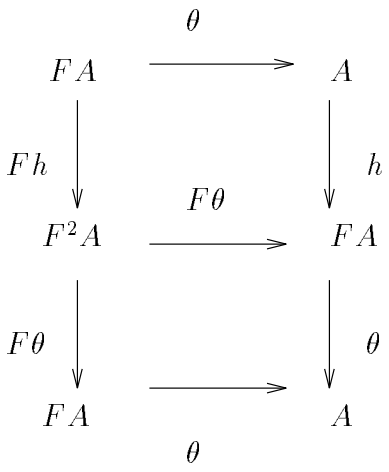
unteres Diagramm:

$$\theta \circ F\theta = \theta \circ F\theta$$

äußeres Diagramm:

$$\begin{array}{lcl} \theta \circ F\theta \circ Fh & = & \theta \circ h \circ \theta \\ & \Longleftrightarrow & \\ \theta \circ F(\theta \circ h) & = & (\theta \circ h) \circ \theta \end{array}$$

[1] statt $f \in \mathcal{C}(A, B)$ schreibe ich bisweilen für Morphismen f auch kurz $f : A \to B$.



Aus der letzten Gleichung folgt, daß $\theta \circ h : A \to A$ ein F-Homomorphismus ist. Der Identitätsmorphismus id_A ist aber ebenfalls ein F-Homomorphismus. Damit folgt wegen der Initialität von (A, θ), daß:

$$\theta \circ h = id_A$$

Aus der Kommutativität des oberen Diagramms folgt weiter:

$$\theta \circ h = F\theta \circ Fh = F(\theta \circ h) = F(id_A) = id_{FA}$$

Damit ist gezeigt, daß θ eine Isomorphismus ist mit $\theta^{-1} = h$.

Wir betrachten nun Kategorien $\mathcal{C}$, in denen jeder Endofunktor $F : \mathcal{C} \to \mathcal{C}$ mit geeigneten zusätzlichen Eigenschaften eine initiale F-Algebra besitzt. Damit dies möglich ist, müssen an die Kategorie $\mathcal{C}$ und an den Funktor F weitere Anforderungen gestellt werden.

Folgende Definition charakterisiert eine Variante von Wand's O-Kategorien (order-enriched categories) [Wan79].

Definition 5.2 *Strikte O-Kategorien*

Eine Kategorie $\mathcal{C}$ ist genau dann eine strikte O-Kategorie, wenn folgende Bedingungen erfüllt sind:

so-cat1: Für alle Objekte $A, B \in Obj(\mathcal{C})$ trägt das Homset $\mathcal{C}(A, B)$ eine ω-pcpo Struktur, d.h. es gibt erstens eine partielle Ordnung $\sqsubseteq$ auf den Morphismen in $\mathcal{C}(A, B)$, die vollständig bezüglich ω-Ketten[2] ist, und zweitens gibt es in jedem Homset $\mathcal{C}(A, B)$ ein kleinstes Element $\bot_{A,B}$ bezüglich $\sqsubseteq$.

so-cat2: Die Komposition $\circ$ von Morphismen ist stetig in beiden Argumenten bezüglich ω-Ketten.

[2] ω-Ketten seien auch hier so definiert, daß sie stets nicht-leer sind.

so-cat3: Die Komposition $\circ$ von Morphismen ist strikt in beiden Argumenten, d.h.

$$\bot_{B,C} \circ f = \bot_{A,C} \quad \text{und} \quad g \circ \bot_{A,B} = \bot_{A,C}$$

für Morphismen $f \in \mathcal{C}(A,B)$ und $g \in \mathcal{C}(B,C)$.

In [AL91] ist die Striktheit der Komposition im ersten Argument bereits in der Definition von O-Kategorien enthalten, während in [Gun92] nur die Eigenschaften so-cat1 und so-cat2 für O-Kategorien gefordert werden. Da ich in Bedingung so-cat3 die Striktheit in beiden Argumenten fordere, habe ich die Bezeichnung *strikte O-Kategorie* gewählt. Streicher verwendet in seiner Vorlesung statt *strikte O-Kategorie* die Bezeichnung *strict cpo-enriched categorie*. In [Fre90] werden strikte O-Kategorien gleich mit zusätzlichen Eigenschaften eingeführt und werden dort als *CPO-Kategorien* bezeichnet. Eine Variante der Freyd'schen Definition werde ich später ebenfalls unter der Bezeichnung *CPO-Kategorie* einführen.

Definition 5.3 *Lokal monotone, lokal stetige Funktoren [Gun92]*

Seien $\mathcal{C}$ und $\mathcal{D}$ strikte O-Kategorien. Ein Funktor $F:\mathcal{C} \to \mathcal{D}$ heißt lokal monoton, wenn für alle Morphismen $f, g \in \mathcal{C}(A,B)$ gilt:

$$f \sqsubseteq g \Longrightarrow Ff \sqsubseteq Fg$$

Ein lokal monotoner Funktor $F:\mathcal{C} \to \mathcal{D}$ heißt lokal stetig, wenn für alle ω-Ketten $(f_n)_{n\in\omega}$ von Morphismen gilt:

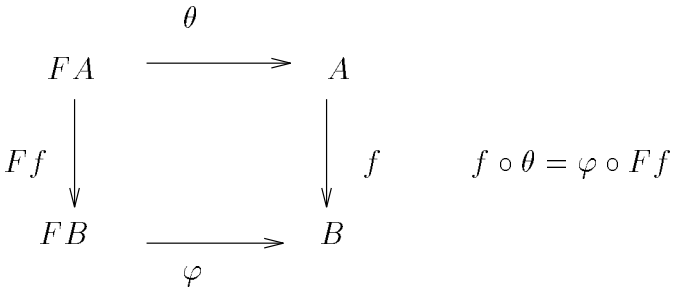
$$F(\bigsqcup_{n\in\omega} f_n) = \bigsqcup_{n\in\omega} Ff_n$$

Das folgende Theorem für F-Algebren in strikten O-Kategorien wird sich in den späteren Ausführungen als nützlich erweisen.

Theorem 5.2

Sei $\mathcal{C}$ eine strikte O-Kategorie, und sei F ein lokal stetiger Endofunktor auf $\mathcal{C}$. Seien weiterhin (A,θ) eine F-Algebra, θ ein Isomorphismus und (B,φ) eine andere F-Algebra. Dann gibt es stets einen kleinsten F-Homomorphismus $f:A \to B$.

Beweis: Wir suchen einen Morphismus f, so daß erstens das folgende Diagramm kommutiert:



$$f \circ \theta = \varphi \circ Ff$$

und zweitens f kleiner ist, als jeder andere Morphismus g, für den das Diagramm ebenfalls kommutiert. Da θ ein Isomorphismus ist, reicht es einen Morphismus f zu finden, der der kleinste Fixpunkt für folgende Gleichung ist:

$$f = \varphi \circ Ff \circ \theta^{-1}$$

Da $\mathcal{C}$ eine strikte O-Kategorie ist, und der Funktor F lokal stetig ist, können wir den kleinsten Fixpunkt der Gleichung durch die Standard-Konstruktion für kleinste Fixpunkte gewinnen. Wir definieren:

$$f = \bigsqcup_{n \in \omega} f_n \quad \text{wobei } f_0 = \bot_{A,B} \text{ und } f_{n+1} = \varphi \circ Ff_n \circ \theta^{-1}$$

Per Konstruktion ist der so definierte Morphismus f der kleinste F-Homomorphismus zwischen (A, θ) und (B, φ).

Die im Beweis benutzte Konstruktion für den kleinsten F-Homomorphismus wird uns noch öfter begegnen. Daher werde ich im folgenden statt 'f ist der kleinste F-Homomorphismus zwischen (A, θ) und (B, φ)' etwas salopper schreiben '$f = \mathit{fix}(\lambda f.\varphi \circ Ff \circ \theta^{-1})$', um an die Konstruktion zu erinnern.

Jetzt kommen wir zur zentralen Aussage dieses Abschnitts. Hier wird die kategorielle Rechtfertigung für die Axiome gegeben, die in HOLCF und LCF zur Axiomatisierung von rekursiven Bereichstypen verwendet werden. Die hier gemachte Aussage für den einfachen Fall des covarianten Funktors findet sich zwar auch schon in Vorlesungsnotizen von Plotkin aus dem Jahr 1983 [KK92], die besondere Betonung der lokalen Charakterisierbarkeit der initialen F-Algebra findet man aber erst in [Fre90].

Theorem 5.3 *Lokales Kriterium für initiale F-Algebren*

Sei $\mathcal{C}$ eine strikte O-Kategorie und F ein lokal stetiger Endofunktor auf $\mathcal{C}$. Sei weiterhin (A, θ) eine F-Algebra. Dann sind folgende beiden Aussagen[3] äquivalent:

1. Der Morphismus θ ist ein Isomorphismus, und die Identität id_A ist der kleinste F-Endomorphismus zwischen (A, θ) und sich selbst. In Zeichen, unter Verwendung der saloppen Notation, bedeutet dies:
   $$\theta \circ \theta^{-1} = id_A$$
   $$\theta^{-1} \circ \theta = id_{FA}$$
   $$id_A = \mathit{fix}(\lambda h.\theta \circ Fh \circ \theta^{-1})$$
2. Die F-Algebra (A, θ) ist initial in der Kategorie F-Alg.

Bemerkungen zum Theorem:

Die erste Bedingung ist ein lokales Kriterium zur Charakterisierung der initialen F-Algebra, die ausschließlich Aussagen über die eine F-Algebra (A, θ) macht. Dagegen spricht die zweite Bedingung über die Menge aller F-Algebren. Der Vorteil

[3] in [Fre90] wird noch eine dritte äquivalente Aussage über terminale F-CoAlgebren gemacht.

der ersten Bedingung liegt darin, daß für konkrete Funktoren F die Anwendung FA des Funktors auf Objekte bzw. die Anwendung Fh des Funktors auf Morphismen in einer Logik wie HOLCF oder LCF syntaktisch ausgedrückt werden kann. Somit besteht in der Logik die Möglichkeit zur Axiomatisierung der initialen F-Algebra, obwohl der Funktor F an sich nicht geschlossen in der Sprache der Logik dargestellt werden kann. So ist in der Logik HOLCF für einen konkreten Funktor F die Anwendung FA ein Typterm für `pcpo`-Typen, und die Anwendung Fh ist ein Term für eine Operation.

Beweis des Theorems 5.3:

$2 \Longrightarrow 1$: Die Isomorphie-Eigenschaft folgt unmittelbar aus Theorem 5.1. Da der Morphismus id_A stets existiert und wegen der Initialität von (A, θ) damit der einzige ist, ist er trivialerweise auch der kleinste.

$1 \Longrightarrow 2$: Sei (B, φ) eine beliebige F-Algebra. Aus Theorem 5.2 wissen wir, daß der Morphismus

$$f = \mathit{fix}(\lambda f.\varphi \circ Ff \circ \theta^{-1})$$

der kleinste F-Homomorphismus zwischen (A, θ) und (B, φ) ist. Es gilt also insbesondere:

$$\begin{array}{ccc} FA & \xrightarrow{\theta} & A \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle f} \\ FB & \xrightarrow[\varphi]{} & B \end{array} \qquad f \circ \theta = \varphi \circ Ff$$

Die Existenz eines F-Homomorphismus ist somit trivial sichergestellt. Um die Initialität von (A, θ) zu zeigen, müssen wir nur noch nachweisen, daß f der einzige F-Homomorphismus ist!

Nach Voraussetzung wissen wir, daß:

$$id_A = \mathit{fix}(\lambda h.\theta \circ Fh \circ \theta^{-1})$$

d.h.:

$$id_A = \bigsqcup_{n \in \omega} h_n \quad \text{wobei } h_0 = \bot_{A,A} \text{ und } h_{n+1} = \theta \circ Fh_n \circ \theta^{-1}$$

Aus Theorem 5.2 wissen wir, daß:

$$f = \mathit{fix}(\lambda f.\varphi \circ Ff \circ \theta^{-1})$$

d.h.:

$$f = \bigsqcup_{n\in\omega} f_n \quad \text{wobei } f_0 = \bot_{A,B} \text{ und } f_{n+1} = \varphi \circ F f_n \circ \theta^{-1}$$

Sei nun $g : A \to B$ ein beliebiger anderer F-Homomorphismus, d.h.:

$$\varphi \circ Fg = g \circ \theta$$

Wenn f der einzige F-Homomorphismus sein soll, dann muß $f = g$ gelten. Dazu benutzen wir das Hilfstheorem $\forall n.\ g \circ h_n = f_n$, das wir im Anschluß an den Hauptbeweis zeigen werden. Mit dem Hilfstheorem läßt sich folgende Gleichungskette bilden:

$$\begin{array}{ll}
 & g \\
= & \\
 & g \circ id_A \\
= & \{ \text{ Voraussetzung } id_A = \bigsqcup_{n\in\omega} h_n \ \} \\
 & g \circ \bigsqcup_{n\in\omega} h_n \\
= & \{ \circ \text{ ist stetig } \} \\
 & \bigsqcup_{n\in\omega} g \circ h_n \\
= & \{ \text{ Hilfstheorem } \forall n.\ g \circ h_n = f_n \ \} \\
 & \bigsqcup_{n\in\omega} f_n \\
= & \\
 & f
\end{array}$$

Der Beweis für das Hilfstheorem $\forall n.\ g \circ h_n = f_n$ erfolgt durch Induktion über n:

Induktionsanfang: $n = 0$

$$\begin{array}{ll}
 & g \circ h_0 \\
= & \\
 & g \circ \bot_{A,A} \\
= & \{ \circ \text{ ist strikt } \} \\
 & \bot_{A,B} \\
= & \\
 & f_0
\end{array}$$

Induktionsschritt: $n \rightsquigarrow n+1$

$$
\begin{aligned}
& g \circ h_{n+1} \\
= & \\
& g \circ (\theta \circ F h_n \circ \theta^{-1}) \\
= & \quad \{ g \text{ ist ein } F\text{-Homomorphismus, d.h. } \varphi \circ Fg = g \circ \theta \ \} \\
& \varphi \circ Fg \circ F h_n \circ \theta^{-1} \\
= & \\
& \varphi \circ F(g \circ h_n) \circ \theta^{-1} \\
= & \quad \{ \text{ Induktionsvoraussetzung } \} \\
& \varphi \circ F(f_n) \circ \theta^{-1} \\
= & \\
& f_{n+1}
\end{aligned}
$$

Damit ist Theorem 5.3 bewiesen.

Theorem 5.3 zeigt, wie man initiale F-Algebren lokal charakterisieren kann, es macht jedoch keine Aussagen über die Existenz von initialen F-Algebren. Um diese zu gewährleisten, müssen weitere Anforderungen an die Kategorie $\mathcal{C}$ gestellt werden. Zunächst eine vorbereitende Definition.

Definition 5.4 *EP-Paare*

Sei $\mathcal{C}$ eine strikte O-Kategorie. Ein Paar (ϵ, π) von Morphismen $\epsilon : A \to B$ und $\pi : B \to A$ heißt EP-Paar (embedding-projection pair), wenn:

$$\pi \circ \epsilon = id_A \quad \text{und} \quad \epsilon \circ \pi \sqsubseteq id_B$$

Es folgt die Definition für eine Variante von Freyd's CPO-Kategorien, die sich enger an die Notizen von Thomas Streicher anlehnt.

Definition 5.5 *CPO-Kategorien*

Eine strikte O-Kategorie $\mathcal{C}$ heißt CPO-Kategorie, wenn folgende zusätzlichen Bedingungen erfüllt sind:

cpo-cat1: Es gibt ein Nullobjekt 0 (Bi-Terminator) in $\mathcal{C}$, d.h. ein Objekt, das initial und terminal zugleich ist.

cpo-cat2: Für jede Folge von EP-Paaren

$$(\epsilon_n : D_n \to D_{n+1}, \pi_n : D_{n+1} \to D_n)_{n \in \omega}$$

gibt es ein Objekt $D \in \mathit{Obj}(\mathcal{C})$ und eine Folge von EP-Paaren

$$(\iota_n : D_n \to D, \rho_n : D \to D_n)_{n \in \omega}$$

so daß gilt:

1. $\iota_{n+1} \circ \epsilon_n = \iota_n$ und $\pi_n \circ \rho_{n+1} = \rho_n$
2. $\bigsqcup_{n\in\omega} \iota_n \circ \rho_n = id_D$

Bemerkungen zur Bedingung cpo-cat2:

Die $\iota_n \circ \rho_n$ bilden eine Kette, was man leicht durch folgende Überlegung einsieht:

$$\begin{aligned} \iota_n \circ \rho_n &= \iota_{n+1} \circ \epsilon_n \circ \pi_n \circ \rho_{n+1} \\ &\sqsubseteq \iota_{n+1} \circ id_{D_{n+1}} \circ \rho_{n+1} \\ &= \iota_{n+1} \circ \rho_{n+1} \end{aligned}$$

Weil $\mathcal{C}$ eine strikte O-Kategorie ist, existiert daher das Supremum in Bedingung cpo-cat2.2!

In kategorieller Sprechweise garantieren die Bedingungen cpo-cat1 und cpo-cat2, daß es zu jedem Diagramm $(\epsilon_n : D_n \to D_{n+1})_{n\in\omega}$ einen Limes $(D, \iota_n)_{n\in\omega}$ gibt, so daß $(D, \rho_n)_{n\in\omega}$ zugleich ein Co-Limes für das Diagramm $(\pi_n : D_{n+1} \to D_n)_{n\in\omega}$ ist.

Für die Komposition von EP-Paaren einer Folge $(\epsilon_n, \pi_n)_{n\in\omega}$ bietet sich folgende Abkürzung an.

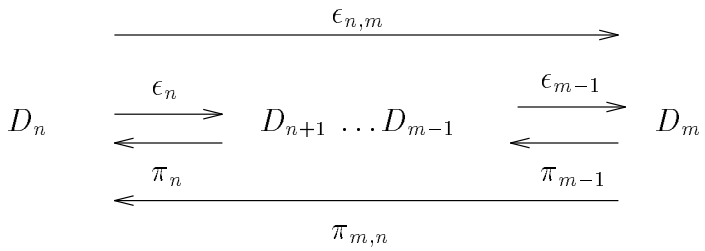
Definition 5.6

Sei $\mathcal{C}$ eine CPO-Kategorie und $(\epsilon_n : D_n \to D_{n+1}, \pi_n : D_{n+1} \to D_n)_{n\in\omega}$ eine Folge von EP-Paaren. Dann sind die Morphismen $\epsilon_{n,m} : D_n \to D_m$ und $\pi_{m,n} : D_m \to D_n$ für alle $n \leq m$ definiert wie folgt:

$$\epsilon_{n,m} = \begin{cases} \epsilon_{m-1} \circ \ldots \circ \epsilon_n & n < m \\ id_{D_n} & n = m \end{cases}$$

$$\pi_{m,n} = \begin{cases} \pi_n \circ \ldots \circ \pi_{m-1} & n < m \\ id_{D_m} & n = m \end{cases}$$

Graphisch veranschaulicht heißt dies:



Die eben definierten Morphismen $\epsilon_{n,m} : D_n \to D_m$ und $\pi_{m,n} : D_m \to D_n$ haben folgende Eigenschaften.

Theorem 5.4

Für alle $n \leq m$ ist $(\epsilon_{n,m}, \pi_{m,n})$ ein EP-Paar.

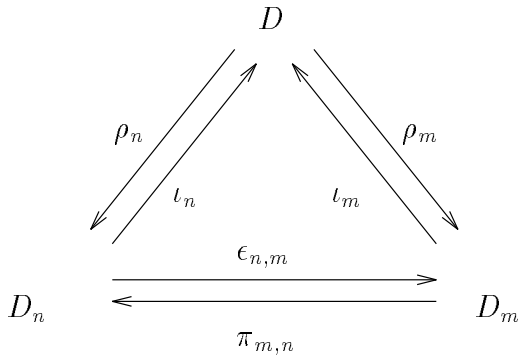
Beweis: leichte Induktion

Theorem 5.5

Für alle $n \leq m$ gilt:

$$\iota_m \circ \epsilon_{n,m} = \iota_n \quad \text{und} \quad \pi_{m,n} \circ \rho_m = \rho_n$$

Graphisch bedeutet dies, daß in folgendem Bild jeweils das innere bzw. das äußere Diagramm kommutiert:



Beweis: Induktion über m und Verwendung von Bedingung cpo-cat2.1.

Theorem 5.6

Für alle $n \leq m$ gilt:

$$\epsilon_{n,m} = \rho_m \circ \iota_n \quad \text{und} \quad \pi_{m,n} = \rho_n \circ \iota_m$$

Beweis: folgt direkt aus Theorem 5.5 durch Linkskomposition mit ρ_m bzw. Rechtskomposition mit ι_m und Eigenschaft $\rho_m \circ \iota_m = id$ der EP-Paare (ι_m, ρ_m).

Das nächste Theorem garantiert die Existenz von initialen F-Algebren in CPO-Kategorien.

Theorem 5.7 *Existenz von initialen F-Algebren*

Sei F ein lokal stetiger Endofunktor $F : \mathcal{C} \to \mathcal{C}$ auf der CPO-Kategorie $\mathcal{C}$. Dann gibt es eine F-Algebra (D, θ), so daß θ ein Isomorphismus ist und $id_D = \mathit{fix}(\lambda h.\theta \circ Fh \circ \theta^{-1})$.

Bemerkung: Mit Theorem 5.3 folgt dann sofort, daß (D, θ) initial in der Kategorie F-Alg ist!

Beweis des Theorems 5.7:
Zuerst definieren wir eine Folge von EP-Paaren

$$(\epsilon_n : D_n \to D_{n+1}, \pi_n : D_{n+1} \to D_n)_{n \in \omega}$$

Wir nützen dabei aus, daß $\mathcal{C}$ eine CPO-Kategorie ist, und demnach ein Nullobjekt 0 besitzt.

$$\begin{array}{rclcrcl} D_0 & = & 0 & & D_{n+1} & = & FD_n \\ \epsilon_0 & = & \bot_{0,D_1} & & \epsilon_{n+1} & = & F\epsilon_n \\ \pi_0 & = & \bot_{D_1,0} & & \pi_{n+1} & = & F\pi_n \end{array}$$

Da F lokal monoton ist, ist $(\epsilon_n, \pi_n)_{n \in \omega}$ eine Folge von EP-Paaren. Nach Bedingung cpo-cat2 in Definition 5.5 gibt es dann aber ein Objekt $D \in Obj(\mathcal{C})$ und eine Folge von EP-Paaren

$$(\iota_n : D_n \to D, \rho_n : D \to D_n)_{n \in \omega}$$

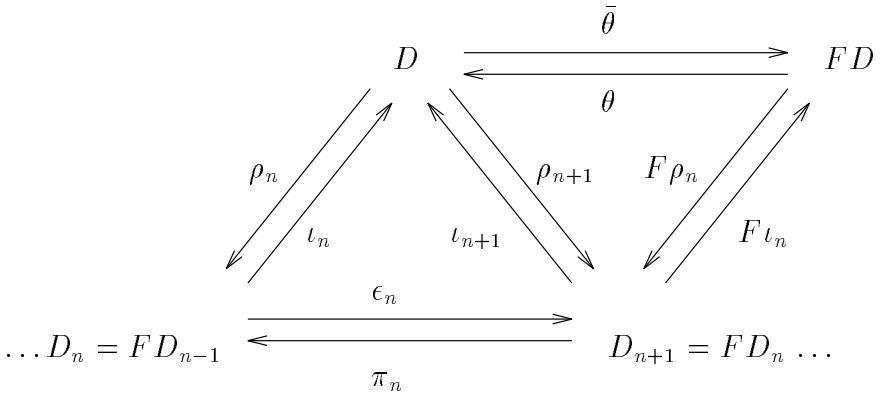
so daß gilt:

1. $\iota_{n+1} \circ \epsilon_n = \iota_n$ und $\pi_n \circ \rho_{n+1} = \rho_n$
2. $\bigsqcup_{n \in \omega} \iota_n \circ \rho_n = id_D$

Dies nützen wir aus und definieren die beiden Morphismen $\theta : FD \to D$ und $\bar{\theta} : D \to FD$ wie folgt:

$$\theta = \bigsqcup_{n \in \omega} \iota_{n+1} \circ F\rho_n \quad \text{und} \quad \bar{\theta} = \bigsqcup_{n \in \omega} F\iota_n \circ \rho_{n+1}$$

Mit Hilfe der Eigenschaft cpo-cat2.1 und der Definition für die Folge $(\epsilon_n, \pi_n)_{n \in \omega}$ weist man leicht die Existenz der obigen Suprema nach. Folgende Graphik verdeutlicht die bisherige Konstruktion:



Per Konstruktion ist (D, θ) eine F-Algebra. Als nächstes zeigen wir, daß θ ein Isomorphismus ist mit $\bar{\theta} = \theta^{-1}$.

$$\begin{aligned}
\theta \circ \bar{\theta} &= (\bigsqcup_{n\in\omega} \iota_{n+1} \circ F\rho_n) \circ (\bigsqcup_{n\in\omega} F\iota_n \circ \rho_{n+1}) \\
&\overset{\circ \text{ stetig}}{=} \bigsqcup_{n\in\omega} \iota_{n+1} \circ F\rho_n \circ F\iota_n \circ \rho_{n+1} \\
&= \bigsqcup_{n\in\omega} \iota_{n+1} \circ F(\rho_n \circ \iota_n) \circ \rho_{n+1} \\
&= \bigsqcup_{n\in\omega} \iota_{n+1} \circ F(id_{D_n}) \circ \rho_{n+1} \\
&= \bigsqcup_{n\in\omega} \iota_{n+1} \circ \rho_{n+1} \\
&\overset{\text{cpo-cat2.2}}{=} id_D
\end{aligned}$$

Umgekehrt zeigen wir:

$$\begin{aligned}
\bar{\theta} \circ \theta &= (\bigsqcup_{n\in\omega} F\iota_n \circ \rho_{n+1}) \circ (\bigsqcup_{n\in\omega} \iota_{n+1} \circ F\rho_n) \\
&\overset{\circ \text{ stetig}}{=} \bigsqcup_{n\in\omega} F\iota_n \circ \rho_{n+1} \circ \iota_{n+1} \circ F\rho_n \\
&\overset{\text{EP}}{=} \bigsqcup_{n\in\omega} F\iota_n \circ F\rho_n \\
&= \bigsqcup_{n\in\omega} F(\iota_n \circ \rho_n) \\
&\overset{F \text{ lokal stetig}}{=} F(\bigsqcup_{n\in\omega} \iota_n \circ \rho_n) \\
&\overset{\text{cpo-cat2.2}}{=} F(id_D) \\
&= id_{FD}
\end{aligned}$$

Damit ist gezeigt, daß θ ein Isomorphismus ist mit $\bar{\theta} = \theta^{-1}$. Als letztes zeigen wir nun noch, daß die Identität id_D gleich dem Morphismus $h = \mathit{fix}(\lambda h.\theta \circ Fh \circ \theta^{-1})$ ist, der gemäß Theorem 5.2 existiert und der kleinste F-Endomorphismus ist. Die Schreibweise

$$h = \mathit{fix}(\lambda h.\theta \circ Fh \circ \theta^{-1})$$

steht bekanntlich für die Konstruktion

$$h = \bigsqcup_{n\in\omega} h_n \quad \text{wobei } h_0 = \bot \text{ und } h_{n+1} = \theta \circ Fh_n \circ \theta^{-1}$$

Mit dem Hilfstheorem $\forall n.\, h_n = \iota_n \circ \rho_n$, das wir im Anschluß an den Hauptbeweis gleich zeigen werden, können wir den Beweis für das Theorem 5.7 abschließen.

$$\begin{aligned}
& \forall n.\, h_n = \iota_n \circ \rho_n \\
\Longrightarrow \quad & \bigsqcup_{n\in\omega} h_n = \bigsqcup_{n\in\omega} \iota_n \circ \rho_n \\
\overset{\text{cpo-cat2.2}}{\Longleftrightarrow} \quad & \mathit{fix}(\lambda h.\theta \circ Fh \circ \theta^{-1}) = id_D
\end{aligned}$$

Wir beweisen nun das Hilfstheorem $\forall n.\ h_n = \iota_n \circ \rho_n$ per Induktion.

Induktionsanfang: $n = 0$

$$h_0 = \bot_{D,D} = \bot_{0,D} \circ \bot_{D,0} = \iota_0 \circ \rho_0$$

Induktionsschritt: $n \rightsquigarrow n+1$

$$\begin{array}{ll}
 & h_{n+1} \\
= & \\
 & \theta \circ F h_n \circ \theta^{-1} \\
= & \{\ \text{Induktionsvoraussetzung}\ \} \\
 & \theta \circ F(\iota_n \circ \rho_n) \circ \theta^{-1} \\
= & \\
 & \theta \circ F\iota_n \circ F\rho_n \circ \theta^{-1} \\
= & \{\ \text{Definitionen für } \theta \text{ und } \bar{\theta} = \theta^{-1}\ \} \\
 & (\bigsqcup_{n\in\omega} \iota_{n+1} \circ F\rho_n) \circ F\iota_n \circ F\rho_n \circ (\bigsqcup_{n\in\omega} F\iota_n \circ \rho_{n+1}) \\
= & \{\ \text{Standardargumentation über Majoranten, Laufindex } m \text{ mit } n \text{ konstant!}\ \} \\
 & (\bigsqcup_{n\le m} \iota_{m+1} \circ F\rho_m) \circ F\iota_n \circ F\rho_n \circ (\bigsqcup_{n\le m} F\iota_m \circ \rho_{m+1}) \\
= & \{\ \circ \text{ stetig, } F \text{ lokal stetiger Funktor}\ \} \\
 & (\bigsqcup_{n\le m} \iota_{m+1} \circ F(\rho_m \circ \iota_n)) \circ (\bigsqcup_{n\le m} F(\rho_n \circ \iota_m) \circ \rho_{m+1}) \\
= & \{\ \text{Theorem 5.6: } \epsilon_{n,m} = \rho_m \circ \iota_n \text{ und } \pi_{m,n} = \rho_n \circ \iota_m\ \} \\
 & (\bigsqcup_{n\le m} \iota_{m+1} \circ F(\epsilon_{n,m})) \circ (\bigsqcup_{n\le m} F(\pi_{m,n}) \circ \rho_{m+1}) \\
= & \{\ \text{Definition der } \epsilon_{n,m} \text{ und } \pi_{m,n}\ ,\ F \text{ Funktor}\ \} \\
 & (\bigsqcup_{n\le m} \iota_{m+1} \circ \epsilon_{n+1,m+1}) \circ (\bigsqcup_{n\le m} \pi_{m+1,n+1} \circ \rho_{m+1}) \\
= & \{\ \text{Theorem 5.5: } \iota_m \circ \epsilon_{n,m} = \iota_n \text{ und } \pi_{m,n} \circ \rho_m = \rho_n\ \} \\
 & (\bigsqcup_{n\le m} \iota_{n+1}) \circ (\bigsqcup_{n\le m} \rho_{n+1}) \\
= & \{\ \text{Standardargumentation über Suprema: } n \text{ ist konstant!}\ \} \\
 & \iota_{n+1} \circ \rho_{n+1}
\end{array}$$

Somit ist das Hilfstheorem bewiesen und damit auch der Beweis für das Theorem 5.7 beendet.

Zum Abschluß dieses Abschnitts möchte ich noch kurz die wesentliche Aussage von Freyd's Arbeit [Fre90] für den allgemeineren Fall eines lokal stetigen Funktors $F:\mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ mit contravarianten und covarianten Anteilen (mixed-variant functor) darstellen. Analog zum Begriff der F-Algebra definiert man hier den Begriff der F-Dialgebra als Tripel $(A, \theta, \bar{\theta})$ mit Objekt A und Morphismen $\theta:FAA \to A$ und $\bar{\theta}:A \to FAA$. Es gilt dann folgender, an die Notation dieses Abschnitts angepaßte Satz, der eine Verallgemeinerung der Theoreme 5.3 und 5.7 ist:

Theorem 5.8

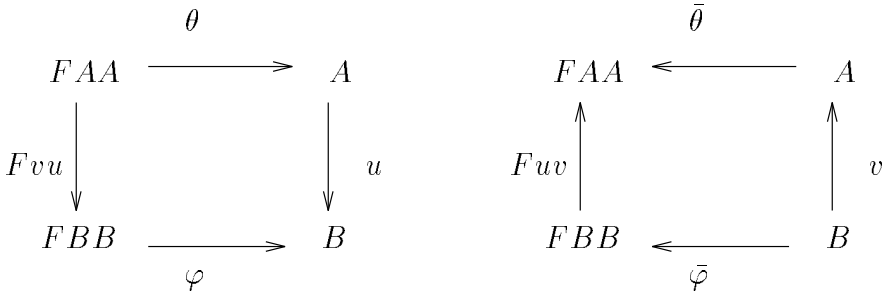
Sei $F:\mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$ ein lokal stetiger Funktor auf der CPO-Kategorie $\mathcal{C}$ und sei $(A, \theta, \bar{\theta})$ eine F-Dialgebra. Dann sind folgende beiden Aussagen äquivalent:

1. (lokale Charakterisierung) Der Morphismus θ ist ein Isomorphismus mit $\theta^{-1} = \bar{\theta}$, und die Identität id_A ist der kleinste F-Endomorphismus zwischen $(A, \theta, \bar{\theta})$ und sich selbst, d.h. $id_A \sqsubseteq h$ für jedes h mit $\theta \circ Fhh = h \circ \theta$.

   Unter Verwendung der saloppen Notation bedeutet dies:

   $$\theta \circ \theta^{-1} = id_A$$
   $$\theta^{-1} \circ \theta = id_{FAA}$$
   $$id_A = \mathit{fix}(\lambda h.\theta \circ Fhh \circ \theta^{-1})$$

2. (globale Charakterisierung, Eindeutigkeit) Für jede andere F-Dialgebra $(B, \varphi, \bar{\varphi})$ gibt es eindeutig bestimmte Morphismen $u:A \to B$ und $v:B \to A$, so daß folgende Diagramme kommutieren:



Weiterhin gibt es immer eine F-Dialgebra $(A, \theta, \bar{\theta})$, für die die erste Aussage erfüllt ist (Existenzaussage).

Beweis: siehe [Fre90]

5.2 Anwendung der kategoriellen Theorie auf HOLCF

Im letzten Abschnitt wurde gezeigt, wie initiale Lösungen von Bereichsgleichungen in strikten O-Kategorien charakterisiert werden und wie sie in CPO-Kategorien konstruiert werden. In diesem Abschnitt wird gezeigt, wie die kategorielle Theorie auf Modelle von HOLCF-Theorien angewendet werden kann.

Im folgenden werden HOLCF-Theorien Th betrachtet, die die Theorie `ccc1` aus Abschnitt 4.13 als Teiltheorie enthalten. In Abschnitt 5.2.1 wird eine CPO-Kategorie $\mathcal{PCPO}$ ausgezeichnet, die sich aus dem jeweiligen Modell $\mathcal{M}$ für die Theorie Th ableiten läßt. In Abschnitt 5.2.2 werden dann sogenannte Funktorterme eingeführt, mit deren Hilfe sich lokal stetige Endofunktoren $F:\mathcal{PCPO} \to \mathcal{PCPO}$ beschreiben lassen. Gemäß der Theorie aus Abschnitt 5.1 existieren für diese Funktoren stets initiale F-Algebren, die sich durch die lokalen Kriterien aus Satz 5.3 auch in einer Logik wie HOLCF oder LCF syntaktisch charakterisieren lassen.

5.2.1 Die CPO-Kategorie $\mathcal{PCPO}$

Sei im folgenden $Th = (\Sigma, Ax, KAx)$ eine HOLCF-Theorie, die die Theorie `ccc1` aus Abschnitt 4.13 als Teiltheorie enthält, und sei $\mathcal{M} = (\mathcal{TM}, \mathcal{C})$ mit $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$ ein beliebiges, durch konservative Modellerweiterung entstandenes, Modell für Th. Dann können wir folgende Kategorie $\mathcal{PCPO}$ aus $\mathcal{M}$ ableiten.

Definition 5.7 *Die Kategorie $\mathcal{PCPO}$*

Als Menge der Objekte $Obj(\mathcal{PCPO})$ wählen wir das Universum $\mathtt{pcpo}^{\mathcal{TM}}$, das die Interpretation der Klasse `pcpo` in $\mathcal{TM}$ ist.

$$Obj(\mathcal{PCPO}) = \mathtt{pcpo}^{\mathcal{TM}}$$

Die Morphismen $Mor(\mathcal{PCPO})$ leiten wir aus der Menge der *strikten* Operationen über der Klasse `pcpo` ab. Die Morphismen werden explizit als Tripel $(f, dom_f, codom_f)$ definiert, um der kategoriellen Maschinerie Genüge zu tun. Diese verlangt, daß es zwei Abbildungen *dom* und *codom* gibt, die jedem Morphismus seinen Domain und Codomain zuweisen.

$$Mor(\mathcal{PCPO}) = \{(f, X, Y) \mid \exists XY \in \mathtt{pcpo}^{\mathcal{TM}}.\ f \in X \to^{\mathcal{TM}} Y \wedge \mathtt{fapp}_{X,Y}(f)(\bot_X) = \bot_Y\}$$

Dabei stehen $\mathtt{fapp}_{X,Y}$, $\bot_X$ und $\bot_Y$ für die Interpretationen der polymorphen Konstanten `fapp` und $\bot$ spezialisiert auf die jeweiligen X und Y.

Die Abbildungen *dom* und *codom* sind definiert wie folgt:

$$(f, X, Y) \in Mor(\mathcal{PCPO}) \Rightarrow dom(f, X, Y) = X \wedge codom(f, X, Y) = Y$$

Die Abbildung *id*, die jedem Objekt X den Identitätsmorphismus id_X zuweist, ist definiert als:

$$id(X) = id_X = (\mathtt{ID}_X, X, X)$$

Dabei ist $\mathtt{ID}_X$ die Interpretation der polymorphen Konstanten `ID=`Λ`y.y` aus Theorie `ccc1` spezialisiert auf X.

Die Komposition $\circ$ von Morphismen (f, Y, Z) und (g, X, Y) ist definiert als:

$$(f, Y, Z) \circ (g, X, Y) = (f\ \mathtt{oo}_{X,Y,Z}\ g, X, Z)$$

Dabei ist $\mathtt{oo}_{X,Y,Z}$ die Interpretation der polymorphen Konstanten `oo` aus Theorie `ccc1` spezialisiert auf X,Y und Z.

Es gilt offensichtlich der folgende Satz.

Theorem 5.9

Das in Definition 5.7 eingeführte mathematische Objekt $\mathcal{PCPO}$ ist eine Kategorie.

Die Aussage folgt direkt aus der Definition von $\mathcal{PCPO}$. Die Erfüllung der Identitäts- und Assoziativitätsgesetze für die Komposition von Morphismen folgt aus den Theoremen der Theorie `ccc1` in Abschnitt 4.13 und der Tatsache, daß die Komposition strikter Operationen mittels `oo` wieder strikte Operationen ergibt.

Als nächstes definieren wir eine Ordnung auf den Homsets $\mathcal{PCPO}(A, B)$ und machen damit $\mathcal{PCPO}$ zu einer strikten O-Kategorie.

Definition 5.8 *Die strikte O-Kategorie $\mathcal{PCPO}$*

Die Ordnung $\sqsubseteq$ auf den Homsets $\mathcal{PCPO}(A, B)$ leiten wir aus der Ordnung $\sqsubseteq_{A \to B}$ ab, welche die Interpretation des polymorphen Ordnungssymbols $\sqsubseteq$ spezialisiert auf den Raum der Operationen $A \to^{\mathcal{TM}} B$ ist. Wir definieren:

$$(f, A, B) \sqsubseteq (g, A, B) \Longleftrightarrow f \sqsubseteq_{A \to B} g$$

Das kleinste Element $\bot_{A,B}$ ist damit:

$$\bot_{A,B} = (\bot_{A \to B}, A, B)$$

Es gilt auch hier offensichtlich der folgende Satz.

Theorem 5.10

Mit der in Definition 5.8 eingeführten Ordnung wird $\mathcal{PCPO}$ zu einer strikten O-Kategorie.

Per Definition erhält jeder Homset eine ω-pcpo Struktur mit kleinstem Element. Die Striktheit und Stetigkeit der Komposition $\circ$ von Morphismen folgt direkt aus den entsprechenden Eigenschaften der Komposition `oo` von Operationen, die sich (nur) für strikte Argument-Operationen selbst strikt verhält.

Als nächstes zeigen wir, daß die O-Kategorie $\mathcal{PCPO}$ sogar eine CPO-Kategorie ist. Die dazu benötigten zusätzlichen Objekte mit ihren Eigenschaften sind durch die Wahl der Ordnung auf den Morphismen und die interne Struktur der Kategorie $\mathcal{PCPO}$ bereits festgelegt, da Nullobjekte, Limiten und Co-Limiten bis auf Isomorphie eindeutig sind, falls sie existieren. Die nachfolgende Konstruktion für das Objekt D und die Folge der EP-Paare (ι_n, ρ_n) entspricht natürlich der Standardkonstruktion aus der Literatur, die hier nur für die speziellen Belange der Kategorie $\mathcal{PCPO}$ angepaßt wurde.

Definition 5.9 *Die CPO-Kategorie $\mathcal{PCPO}$*

Das Nullobjekt in $\mathcal{PCPO}$ ist durch $\mathtt{void}^{\mathcal{TM}}_{\mathtt{pcpo}}$ gegeben, also die Interpretation der Typkonstanten `void` mit Arität `pcpo`.

$$0 = \mathtt{void}^{\mathcal{TM}}_{\mathtt{pcpo}}$$

Wenn $A \in Obj(\mathcal{PCPO})$, dann sind die eindeutigen initialen und terminalen Morphismen für beliebige Typbelegung ν gegeben durch:

$$\begin{aligned} i_A &= (\llbracket \Lambda \mathtt{x}::\mathtt{void}\,.\,\bot::\alpha \rrbracket_{\nu[A/\alpha]}, 0, A) \\ t_A &= (\llbracket \Lambda \mathtt{x}::\alpha\,.\,\bot::\mathtt{void} \rrbracket_{\nu[A/\alpha]}, A, 0) \end{aligned}$$

Es ist leicht einzusehen, daß die verwendeten Operationen $\Lambda \mathtt{x}::\mathtt{void}\,.\,\bot::\alpha$ und $\Lambda \mathtt{x}::\alpha\,.\,\bot::\mathtt{void}$ die einzigen strikten Operationen von und in den Typ `void` sind.

Als nächstes weisen wir die Existenz des Objekts D und der Folge $(\iota_n, \rho_n)_{n\in\omega}$ nach. Sei dazu gemäß Bedingung cpo-cat2 aus Definition 5.5 eine Folge von EP-Paaren

$$(\epsilon_n : D_n \to D_{n+1}, \pi_n : D_{n+1} \to D_n)_{n\in\omega}$$

gegeben.

Als Objekt D wählen wir eine geeignete Teilmenge des abzählbaren kartesischen Produkts $\Pi_{n\in\omega} D_n$. Die Objekte der Kategorie $\mathcal{PCPO}$ sind nach Definition 5.7 Elemente X aus dem Universum $\mathtt{pcpo}^{\mathcal{TM}}$, d.h. mathematische Strukturen, die aus einer Trägermenge, einer Interpretation für die partielle Ordnung $\sqsubseteq$ und einer Interpretation für das kleinste Element $\bot$ bestehen[4].

Als Trägermenge car_D für das Objekt D wählen wir:

$$car_D = \{x \in \Pi_{n\in\omega} car(D_n) \mid \forall n \in \omega.\ \pi_n[x_{n+1}] = x_n\}$$

Dabei ist $\pi_n[x_{n+1}]$ die Kurzschreibweise für $\mathtt{fapp}_{D_{n+1},D_n}(f)(x_{n+1})$, wenn der Morphismus π_n das Tripel (f, D_{n+1}, D_n) ist, und x_n steht jeweils für die n-te Komponente des abzählbar unendlichen Tupels x.

Die partielle Ordnung $\sqsubseteq_D$ auf dem Träger des Objekts D gewinnen wir durch punktweise Fortsetzung der partiellen Ordnungen auf den D_n, welches die kanonische Ordnung auf Produkten ergibt. Das kleinste Element $\bot_D$ ist damit automatisch das unendliche Tupel $(\bot_{D_0}, \bot_{D_1}, \bot_{D_2}, \ldots)$. Somit erhalten wir ein Element im Universum $\mathtt{pcpo}^{\mathcal{TM}}$ und damit ein Objekt der Kategorie $\mathcal{PCPO}$. Zur Wohldefiniertheit werde ich im Anschluß an die Definition noch einige Bemerkungen machen.

Nun definieren wir die Folge der Morphismen

$$(\iota_n : D_n \to D, \rho_n : D \to D_n)_{n\in\omega}$$

so daß sie eine Folge von EP-Paaren bildet. Der Morphismus $\iota_n : D_n \to D$ wird für alle n definiert als:

$$\iota_n = (\mathtt{fabs}_{D_n,D}(f_n), D_n, D)$$

[4] auf die exakte Formulierung mittels der Selektoren *car,stru,strufam* und *const* verzichte ich zugunsten der besseren Lesbarkeit. Diesbezügliche Details finden sich in Abschnitt 2.4.

wobei die Hilfsfunktion f_n für alle Komponenten m des unendlichen Ergebnistupels in D definiert wird wie folgt:

$$(f_n(x))_m = \begin{cases} (\epsilon_{m-1} \circ \ldots \circ \epsilon_n)[x] & ; \quad n < m \\ x & ; \quad n = m \\ (\pi_m \circ \ldots \circ \pi_{n-1})[x] & ; \quad m < n \end{cases}$$

Dabei wurde wieder die vereinfachte Notation $m[x]$ für $\mathtt{fapp}_{X,Y}(fo)(x)$ verwendet, wenn $m = (fo, X, Y)$.

Der Morphismus $\rho_n : D \to D_n$ wird für alle n definiert als:

$$\rho_n = (\mathtt{fabs}_{D,D_n}(f_n), D, D_n)$$

wobei

$$f_n(x_1, \ldots, x_n, \ldots) = x_n$$

Der Morphismus ρ_n ist also einfach die n-te Projektion für das unendliche Tupel.

Bemerkungen zur Wohldefiniertheit des Objekts D und der Morphismen (ι_n, ρ_n):

- Die Trägermenge car_D des Objekts D ist wegen der Abschlußeigenschaften von Präuniversen aus Definition 2.22, Abschnitt 2.4 wieder ein Element des Präuniversums PU, welches dem Modell $\mathcal{M}$ zugrunde liegt. Für die diesbezügliche Argumentation können insbesondere die Eigenschaften E2 (Abschluß bezüglich Teilmengenbidlung) und E4 (Abschluß bezüglich Bildung des abzählbaren Produkts) herangezogen werden. Damit ist aber auch sichergestellt, daß das Objekt D ein Element des Universums $\mathtt{pcpo}^{T\mathcal{M}}$ ist, da dieses Universum durch konservative Modellerweiterung entstanden ist und somit maximal groß ist, d.h. jeden Bereich mit ω-pcpo Struktur enthält, der auf der Grundlage des Präuniversums PU gebildet werden kann. Per Konstruktion ist D ein Bereich mit ω-pcpo Struktur.
- Aus der Konstruktion der Hilfsfunktionen f_n, die bei der Definition der Morphismen ι_n und ρ_n eingesetzt werden, ist leicht ersichtlich, daß die Abbildungen f_n strikte und stetige Funktionen sind. Daher entstehen durch Anwendung der Abstraktionsfunktion `fabs` für Operationen sicher strikte Operationen, und damit ist die Wohldefiniertheit der Morphismen ι_n und ρ_n garantiert.

Theorem 5.11

Die Kategorie $\mathcal{PCPO}$ ist eine CPO-Kategorie. Insbesondere erfüllen das Objekt D und die Folge $(\iota_n, \rho_n)_{n \in \omega}$, die für die Kategorie $\mathcal{PCPO}$ festgelegt wurden, die Bedingung cpo-cat2 aus der Definition 5.5.

Beweis:
Per Konstruktion erfüllt das Objekt 0 mit den zugehörigen Morphismen i_A und t_A die

Eigenschaften eines Nullobjekts. Damit ist Bedingung cpo-cat1 für CPO-Kategorien erfüllt.

Die komplette Konstruktion des Objekts D und der Folge $(\iota_n, \rho_n)_{n\in\omega}$ findet sich in teils anderer, aber analoger Notation, in [Sch86, Seiten 260-262] und [Pau87, Seiten 101-103]. Insbesondere findet man dort Beweise für:

1. $(\iota_n, \rho_n)_{n\in\omega}$ ist eine Folge von EP-Paaren.
2. $\iota_{n+1} \circ \epsilon_n = \iota_n$ und $\pi_n \circ \rho_{n+1} = \rho_n$
3. $\bigsqcup_{n\in\omega} \iota_n \circ \rho_n = id_D$

Somit ist gezeigt, wie in jedem Modell $\mathcal{M}$ einer Theorie Th mit $\texttt{ccc1} \subseteq Th$ eine CPO-Kategorie $\mathcal{PCPO}$ indentifiziert werden kann. Wenden wir uns nun der Frage zu, wie lokal stetige Endofunktoren $F : \mathcal{PCPO} \to \mathcal{PCPO}$ in der Syntax der Logik HOLCF beschrieben werden können.

5.2.2 Funktorterme

Die Audrucksstärke der Logik HOLCF reicht nicht aus, um einen Funktor F als ganzes in Form eines Terms der Sprache darzustellen. Wenn wir jedoch die lokale Charakterisierung von initialen F-Algebren in Theorem 5.3 betrachten, so fällt auf, daß der Funktor F nur im Anwendungskontext vorkommt. Folglich reicht es, wenn wir die Anwendung FA des Funktors F auf Objekte A und die Anwendung Fh des Funktors F auf Morphismen h in der Logik syntaktisch ausdrücken können.

Aus der speziellen Struktur der Kategorie $\mathcal{PCPO}$ folgt, daß der Objektanteil F^O eines Funktors F eine Abbildung von pcpo-Typen in pcpo-Typen ist, und daß der Morphismusanteil F^M des Funktors eine Abbildung von strikten Operationen in strikte Operationen sein muß. Daraus wiederum folgt, daß im Anwendungskontext $F^O A$ einen pcpo-Typ darstellt, und $F^M h$ für eine strikte Operation steht.

Im folgenden werde ich eine neue Sorte von Termen einführen, die sogenannten Funktorterme, die es erlauben, primitive Funktoren und deren Kompositionen zu beschreiben. Aufgrund der in HOLCF vorkommenden Polymorphie enthalten diese Funktorterme Typvariablen. Funktorterme stehen also für ganze Familien von Funktoren, man kann sie auch als polymorphe Funktoren auffassen. Gleichzeitig definiere ich deren Anwendung auf pcpo-Typterme und Operationen dergestalt, daß die Anwendung des Funktorterms FT auf einen pcpo-Typterm τ durch einen pcpo-Typterm τ' dargestellt werden kann und analog dazu die Anwendung des Funktorterms FT auf einen Operationsterm t durch einen Operationsterm t' ausgedrückt werden kann.

Funktorterme werden nur als technisches Hilfsmittel eingeführt, um die kategorielle Theorie aus Abschnitt 5.1 auf die Logik HOLCF anwenden zu können. In den resultierenden HOLCF-Axiomen zur Formalisierung der initialen Lösung von Bereichsgleichungen treten die Funktorterme nicht mehr auf.

Im Anschluß daran werde ich eine Semantik für Funktorterme definieren, so daß für jede Belegung ν von Typvariablen die Interpretation des Funktorterms unter dieser Belegung einen

lokal stetigen Funktor auf der Kategorie $\mathcal{PCPO}$ beschreibt. Für die so definierte Semantik wird dann noch gezeigt, daß sie mit der vorher definierten Elimination der Funktorterme im Anwendungskontext verträglich ist, was letztendlich die getrennte Darstellung des Objektanteils und des Morphismusanteils von Funktortermen in der Syntax von HOLCF rechtfertigt.

Die folgende Definition führt das Konzept der ein- und zweistelligen Funktorterme ein. Zweistellig sind dabei nur die primitiven Funktorterme $**$ und $++$. Zusammengesetzte Funktorterme sind immer einstellig, was der Intuition entspricht, daß sie Endofunktoren auf der Kategorie $\mathcal{PCPO}$ beschreiben. Simultan mit der Definition der Funktorterme wird auch ihre Anwendung auf Typterme und Operationen definiert, sowie die Menge der in Funktortermen FT vorkommenden Typvariablen $TV(FT)$.

Definition 5.10 *Funktorterme*

Sei $Th = (\Sigma, Ax, KAx)$ eine HOLCF-Theorie, die die Theorie `ccc1` aus Abschnitt 4.13 als Teiltheorie enthält. Seien weiterhin τ, τ_1, τ_2, τ_3 und τ_4 Typterme aus $T_{\Omega,pcpo}$ deren Typvariablen ausschließlich Variablen der Menge Ξ_{pcpo} sind, und seien $t_1 \in T_{\Sigma,\tau_1 \to \tau_2}$ und $t_2 \in T_{\Sigma,\tau_3 \to \tau_4}$ Terme für Operationen mit $(TV(t_1) \cup TV(t_2)) \subseteq \Xi_{pcpo}$.

Dann ist die Menge der Funktorterme über der Theorie Th durch folgende induktive Definition gegeben:

Identität: Der Term I ist ein einstelliger Funktorterm. Die Menge der in I vorkommenden Typvariablen ist definiert als:

$$TV(I) = \emptyset$$

Die Anwendung auf Typterme τ_1 ist definiert als:

$$I\tau_1 = \tau_1$$

Die Anwendung auf Operationen t_1 ist definiert als:

$$It_1 = t_1$$

Konstante: Der Term K_τ ist ein einstelliger Funktorterm. Die Menge der in K_τ vorkommenden Typvariablen ist definiert als:

$$TV(K_\tau) = TV(\tau)$$

Der Term K_τ ist der einzige Funktorterm, der Typvariablen und damit Polymorphie einführt. Alle anderen primitiven Funktorterme enthalten keine Typvariablen. Die weiter unten definierten Kompositionen propagieren lediglich die durch K_τ Terme eingeführten Typvariablen! Die Anwendung auf Typterme τ_1 ist definiert als:

$$K_\tau\tau_1 = \tau$$

Die Anwendung auf Operationen t_1 ist definiert als:

$$K_\tau t_1 = \texttt{ID} :: \tau$$

Lifting: Der Term U ist ein einstelliger Funktorterm. Die Menge der in U vorkommenden Typvariablen ist definiert als:

$$TV(U) = \emptyset$$

Die Anwendung auf Typterme τ_1 ist definiert als:

$$U\tau_1 = (\tau_1)\texttt{u}$$

Die Anwendung auf Operationen t_1 ist definiert als:

$$Ut_1 = \texttt{lift[up oo } (t_1)\texttt{]}$$

striktes Produkt: Der Term $**$ ist ein zweistelliger Funktorterm. Die Menge der in $**$ vorkommenden Typvariablen ist definiert als:

$$TV(**) = \emptyset$$

Die Anwendung auf ein Paar von Typtermen τ_1 und τ_2 ist definiert als:

$$**(\tau_1, \tau_2) = (\tau_1)\ \texttt{**}\ (\tau_2)$$

Die Anwendung auf ein Paar von Operationen t_1 und t_2 ist definiert als:

$$**(t_1, t_2) = \ \texttt{ssplit[}\Lambda\texttt{xy.}(t_1)\texttt{[x]\#\#}(t_2)\texttt{[y]]}$$

strikte Summe: Der Term $++$ ist ein zweistelliger Funktorterm. Die Menge der in $++$ vorkommenden Typvariablen ist definiert als:

$$TV(++) = \emptyset$$

Die Anwendung auf ein Paar von Typtermen τ_1 und τ_2 ist definiert als:

$$++(\tau_1, \tau_2) = (\tau_1)\ \texttt{++}\ (\tau_2)$$

Die Anwendung auf ein Paar von Operationen t_1 und t_2 ist definiert als:

$$++(t_1, t_2) = \texttt{when[sinl oo } (t_1)\texttt{][sinr oo } (t_2)\texttt{]}$$

zweistellige Komposition: Wenn FT_1 und FT_2 einstellige Funktorterme sind, dann ist ihre Komposition $\langle FT_1, FT_2\rangle$ ein einstelliger Funktorterm. Die Menge der in $\langle FT_1, FT_2\rangle$ vorkommenden Typvariablen ist definiert als:

$$TV(\langle FT_1, FT_2\rangle) = TV(FT_1) \cup TV(FT_2)$$

Die Anwendung auf Typterme τ_1 ist definiert als:

$$\langle FT_1, FT_2\rangle\tau_1 = FT_1(FT_2\tau_1)$$

Die Anwendung auf Operationen t_1 ist definiert als:

$$\langle FT_1, FT_2\rangle t_1 = FT_1(FT_2 t_1)$$

dreistellige Komposition: Wenn FT_1 und FT_2 einstellige Funktorterme sind, und GT ein zweistelliger Funktorterm ist, dann ist ihre Komposition $\langle GT, FT_1, FT_2\rangle$ ein einstelliger Funktorterm. Die Menge der in $\langle GT, FT_1, FT_2\rangle$ vorkommenden Typvariablen ist definiert als:

$$TV(\langle GT, FT_1, FT_2\rangle) = TV(FT_1) \cup TV(FT_2)$$

Da alle zweistelligen Funktorterme keine Typvariablen enthalten, ist $TV(GT) = \emptyset$ und muß somit nicht berücksichtigt werden. Die Anwendung auf Typterme τ_1 ist definiert als:

$$\langle GT, FT_1, FT_2\rangle\tau_1 = GT(FT_1\tau_1, FT_2\tau_1)$$

Die Anwendung auf Operationen t_1 ist definiert als:

$$\langle GT, FT_1, FT_2\rangle t_1 = GT(FT_1 t_1, FT_2 t_1)$$

Das folgende Beispiel soll die obige Definition für Funktorterme erläutern.

Beispiel 5.1 *Beispiel für einen Funktorterm*

Als Beispiel betrachten wir den Funktorterm $FT = \langle **, K_\alpha, U\rangle$, der zur Formalisierung von polymorphen Strömen über Typ α benutzt werden kann. Dabei sei $\alpha \in \Xi_{pcpo}$.

Per Definition ist die Menge der in FT vorkommenden Typvariablen gegeben durch:

$$TV(FT) = TV(\langle **, K_\alpha, U\rangle) = \{\alpha\}$$

Die Anwendung von FT auf einen Typterm τ_1 ist gegeben durch:

$$\begin{array}{lcl} FT\tau_1 & = & \langle **, K_\alpha, U\rangle\tau_1 \\ & \stackrel{\langle,,\rangle}{=} & ** (K_\alpha\tau_1, U\tau_1) \\ & \stackrel{**}{=} & (K_\alpha\tau_1) \texttt{**} (U\tau_1) \\ & \stackrel{K,U}{=} & \alpha \texttt{ ** } (\tau_1)\texttt{u} \end{array}$$

Die Anwendung von FT auf einen Term t_1, der eine Operation beschreibt, ist gegeben durch:

$$\begin{array}{lcl} FTt_1 & = & \langle **, K_\alpha, U\rangle t_1 \\ & \stackrel{\langle,,\rangle}{=} & ** (K_\alpha t_1, Ut_1) \\ & \stackrel{**}{=} & \texttt{ssplit[}\Lambda\texttt{xy.(}\ K_\alpha t_1\texttt{)[x]\#\#(}Ut_1\texttt{)[y]]} \\ & \stackrel{K,U}{=} & \texttt{ssplit[}\Lambda\texttt{xy.(ID::}\alpha\texttt{)[x]\#\#(lift[up oo }t_1\texttt{])[y]]} \\ & \stackrel{HOLCF}{=} & \texttt{ssplit[}\Lambda\texttt{xy.x::}\alpha\texttt{ \#\# lift[up oo }t_1\texttt{][y]]} \end{array}$$

In der folgenden Definition wird die Semantik von Funktortermen festgelegt. Die Definition erfolgt dabei dergestalt, daß die Interpretation für jede Variablenbelegung ν einen lokal stetigen Funktor auf der Kategorie $\mathcal{PCPO}$ darstellt, und daß die Semantik verträglich ist mit den in Definition 5.10 festgelegten Anwendungen des Funktorterms auf Typen und Operationen.

Definition 5.11 *Semantik von Funktortermen*

Sei FT ein Funktorterm über der Theorie Th, $\mathcal{M} = (\mathcal{TM}, \mathcal{C})$ mit $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$ ein beliebiges Modell für Th und $\mathcal{PCPO}$ die dadurch induzierte Kategorie.

Dann ist für jede Typvariablenbelegung ν die Interpretation $\mathcal{M}[\![FT]\!]_\nu$ des Funktorterms ein Paar $(\mathcal{M}[\![FT^O]\!]_\nu, \mathcal{M}[\![FT^M]\!]_\nu)$, so daß für einen einstelligen Funktorterm:

$$\begin{array}{lcl} \mathcal{M}[\![FT^O]\!]_\nu & : & Obj(\mathcal{PCPO}) \to Obj(\mathcal{PCPO}) \\ \mathcal{M}[\![FT^M]\!]_\nu & : & Mor(\mathcal{PCPO}) \to Mor(\mathcal{PCPO}) \end{array}$$

bzw. im Fall eines zweistelligen Funktorterms:

$$\begin{array}{lcl} \mathcal{M}[\![FT^O]\!]_\nu & : & Obj(\mathcal{PCPO}) \times Obj(\mathcal{PCPO}) \to Obj(\mathcal{PCPO}) \\ \mathcal{M}[\![FT^M]\!]_\nu & : & Mor(\mathcal{PCPO}) \times Mor(\mathcal{PCPO}) \to Mor(\mathcal{PCPO}) \end{array}$$

Die Definition der Interpretation $\mathcal{M}[\![FT]\!]_\nu$ erfolgt durch Induktion über den Aufbau des Funktorterms FT. Im folgenden wird die Annotation des Modells $\mathcal{M}$ weggelassen, da es sich immer um dasselbe Modell handelt.

Identität:

$$\begin{aligned}[\![I^O]\!]_\nu &: X \mapsto X\\ [\![I^M]\!]_\nu &: m \mapsto m\end{aligned}$$

Konstante:

$$\begin{aligned}[\![K_\tau^O]\!]_\nu &: X \mapsto [\![\tau]\!]_\nu\\ [\![K_\tau^M]\!]_\nu &: m \mapsto ([\![\texttt{ID::}\tau]\!]_\nu, [\![\tau]\!]_\nu, [\![\tau]\!]_\nu)\end{aligned}$$

Lifting:

$$\begin{aligned}[\![U^O]\!]_\nu &: X \mapsto u^{\mathcal{TM}}_{(\texttt{pcpo})\texttt{pcpo}}(X)\\ [\![U^M]\!]_\nu &: (f,X,Y) \mapsto \\ &\quad (([\![\lambda\texttt{fo::}\alpha \to \beta\texttt{.lift[up oo fo]}]\!]_{\nu[X/\alpha,Y/\beta]})(f),\\ &\quad\ [\![U^O]\!]_\nu(X), [\![U^O]\!]_\nu(Y))\end{aligned}$$

striktes Produkt:

$$\begin{aligned}[\![{**}^O]\!]_\nu &: (X,Y) \mapsto {**}^{\mathcal{TM}}_{(\texttt{pcpo,pcpo})\texttt{pcpo}}(X,Y)\\ [\![{**}^M]\!]_\nu &: ((f,X_1,Y_1),(g,X_2,Y_2)) \mapsto \\ &\quad (([\![t]\!]_{\nu[X_1/\alpha_1,X_2/\alpha_2,Y_1/\beta_1,Y_2/\beta_2]})(f)(g),\\ &\quad\ [\![{**}^O]\!]_\nu(X_1,X_2), [\![{**}^O]\!]_\nu(Y_1,Y_2))\end{aligned}$$

wobei

$$t = \lambda\texttt{fo::}\alpha_1 \to \beta_1.\lambda\texttt{go::}\alpha_2 \to \beta_2\texttt{.ssplit[}\Lambda\texttt{xy.fo[x]\#\#go[y]]}$$

strikte Summe:

$$\begin{aligned}[\![{++}^O]\!]_\nu &: (X,Y) \mapsto {++}^{\mathcal{TM}}_{(\texttt{pcpo,pcpo})\texttt{pcpo}}(X,Y)\\ [\![{++}^M]\!]_\nu &: ((f,X_1,Y_1),(g,X_2,Y_2)) \mapsto \\ &\quad (([\![t]\!]_{\nu[X_1/\alpha_1,X_2/\alpha_2,Y_1/\beta_1,Y_2/\beta_2]})(f)(g),\\ &\quad\ [\![{++}^O]\!]_\nu(X_1,X_2), [\![{++}^O]\!]_\nu(Y_1,Y_2))\end{aligned}$$

wobei

$$t = \lambda\texttt{fo::}\alpha_1 \to \beta_1.\lambda\texttt{go::}\alpha_2 \to \beta_2\texttt{.when[sinl oo fo][sinr oo go]}$$

zweistellige Komposition:

$$\begin{aligned}[\![\langle FT_1, FT_2\rangle^O]\!]_\nu &: X \mapsto ([\![FT_1^O]\!]_\nu)(([\![FT_2^O]\!]_\nu)(X))\\ [\![\langle FT_1, FT_2\rangle^M]\!]_\nu &: m \mapsto ([\![FT_1^M]\!]_\nu)(([\![FT_2^M]\!]_\nu)(m))\end{aligned}$$

dreistellige Komposition:

$$\begin{aligned}[\![\langle GT, FT_1, FT_2\rangle^O]\!]_\nu &: X \mapsto ([\![GT_1^O]\!]_\nu)(([\![FT_1^O]\!]_\nu)(X),([\![FT_2^O]\!]_\nu)(X))\\ [\![\langle GT, FT_1, FT_2\rangle^M]\!]_\nu &: m \mapsto ([\![GT_1^M]\!]_\nu)(([\![FT_1^M]\!]_\nu)(m),([\![FT_2^M]\!]_\nu)(m))\end{aligned}$$

Bemerkungen zur Wohldefiniertheit der Semantik von Funktortermen:

Die Definition für die Objektanteile $[\![FT^O]\!]_\nu$ ist unproblematisch und beschreibt sicher eine Abbildung $Obj(\mathcal{PCPO}) \to Obj(\mathcal{PCPO})$ bzw. im Fall der beiden zweistelligen Funktorterme $**$ und $++$ eine Abbildung $Obj(\mathcal{PCPO}) \times Obj(\mathcal{PCPO}) \to Obj(\mathcal{PCPO})$.

Für die Fälle I und K_τ sind $[\![I^M]\!]_\nu$ und $[\![K_\tau^M]\!]_\nu$ offensichtlich Abbildungen $Mor(\mathcal{PCPO}) \to Mor(\mathcal{PCPO})$. Ebenso unproblematisch sind die Kompositionen, da sie lediglich die Abbildung ihrer Teilargumente komponieren. In den anderen Fällen ist eine eingehendere Analyse notwendig, um die Wohldefiniertheit der Morphismusanteile einzusehen. Die Operationen f und g in den Argumenten für die Funktorterme U, $**$ und $++$ sind strikt nach Definition für die Morphismen der Kategorie $\mathcal{PCPO}$. Die Eigenschaften der beteiligten `lift`, `ssplit` und

when-Funktionale garantieren, daß die ersten Komponenten der Ergebnistripel stets eine strikte Operation darstellen. Die explizite Typinformation in den ersten Komponenten der Ergebnistripel sorgt weiterhin dafür, daß die in den zweiten und dritten Komponenten der Tripel berechneten Argument- bzw. Zielbereiche der Morphismen konsistent mit der ersten Komponente sind. Durch die definierten Abbildungen entstehen somit jeweils wieder Morphismen der Kategorie $\mathcal{PCPO}$.

Das folgende Theorem erlaubt die syntaktische Elimination von Funktortermen im Anwendungskontext, d.h. wenn FT ein Funktorterm ist, und τ ein Typterm bzw. t ein Term für eine strikte Operation, dann liefert die Anwendung der Interpretation $\mathcal{M}[\![FT]\!]_\nu$ auf die Interpretation der Terme dasselbe Ergebnis wie die Interpretation der gemäß Definition 5.10 vereinbarten Anwendung $FT\tau$ bzw. FTt.

Theorem 5.12 *Funktorterme im Anwendungskontext*

Sei Th ein Theorie, $\mathcal{M} = (\mathcal{TM}, \mathcal{C})$ mit $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$ ein beliebiges Modell für Th und $\mathcal{PCPO}$ die dadurch induzierte Kategorie.

Seien weiterhin FT ein einstelliger Funktorterm, GT ein zweistelliger Funktorterm über der Theorie Th, τ, τ_1, τ_2, τ_3 und τ_4 Typterme aus $T_{\Omega,pcpo}$ deren Typvariablen ausschließlich Variablen der Menge Ξ_{pcpo} sind, und $t, t_1 \in DT_{\Sigma,\tau_1 \to \tau_2}$ und $t_2 \in DT_{\Sigma,\tau_3 \to \tau_4}$ vollgetypte Terme für strikte Operationen mit $(TV(t_1) \cup TV(t_2)) \subseteq \Xi_{pcpo}$. Dann gilt

- für alle ν, η mit $\eta \; sat \; (\nu, GC(t))$:

$$(\![FT^O]\!]_\nu)([\![\tau]\!]_\nu) = [\![FT\tau]\!]_\nu$$
$$(\![FT^M]\!]_\nu)([\![t]\!]_{\nu,\eta}, [\![\tau_1]\!]_\nu, [\![\tau_2]\!]_\nu) = ([\![FTt]\!]_{\nu,\eta}, [\![FT\tau_1]\!]_\nu, [\![FT\tau_2]\!]_\nu)$$

- für alle ν, η mit $\eta \; sat \; (\nu, GC(t_1))$ und $\eta \; sat \; (\nu, GC(t_2))$:

$$(\![GT^O]\!]_\nu)([\![\tau_1]\!]_\nu, [\![\tau_2]\!]_\nu) = [\![GT(\tau_1, \tau_2)]\!]_\nu$$
$$(\![GT^M]\!]_\nu)(([\![t_1]\!]_{\nu,\eta}, [\![\tau_1]\!]_\nu, [\![\tau_2]\!]_\nu), ([\![t_2]\!]_{\nu,\eta}, [\![\tau_3]\!]_\nu, [\![\tau_4]\!]_\nu)) =$$
$$([\![GT(t_1, t_2)]\!]_{\nu,\eta}, [\![GT(\tau_1, \tau_3)]\!]_\nu, [\![GT(\tau_2, \tau_4)]\!]_\nu)$$

Beweis:
leichte Induktion über den Aufbau der Funktorterme und Benutzung der Definitionen 5.10 und 5.11, sowie der Semantik von Termen.

Folgendes Beispiel soll die Aussage des Theorems 5.12 illustrieren.

Beispiel 5.2

Für das Beispiel $FT = \langle **, K_\alpha, U \rangle$ bedeutet das Theorem, daß für alle ν, η mit $\eta \; sat \; (\nu, GC(t))$:

$$(\![\langle **, K_\alpha, U\rangle^O]\!]_\nu)([\![\tau]\!]_\nu) = [\![\alpha \; ** \; (\tau)\mathtt{u}]\!]_\nu$$
$$(\![\langle **, K_\alpha, U\rangle^M]\!]_\nu)([\![t]\!]_{\nu,\eta}, [\![\tau_1]\!]_\nu, [\![\tau_2]\!]_\nu) =$$
$$([\![\mathtt{ssplit[}\Lambda \mathtt{xy.x::}\alpha \mathtt{\#\# \; lift[up \; oo \;} t\mathtt{][y]]}]\!]_{\nu,\eta}, [\![\alpha \; ** \; (\tau_1)\mathtt{u}]\!]_\nu, [\![\alpha \; ** \; (\tau_2)\mathtt{u}]\!]_\nu)$$

Theorem 5.13 *Funktorterme beschreiben Funktoren*

Sei Th ein Theorie, $\mathcal{M}$ ein beliebiges Modell für Th und $\mathcal{PCPO}$ die dadurch induzierte Kategorie.

Dann beschreibt ein einstelliger Funktorterm FT für jede Typbelegung ν einen Endofunktor $[\![FT]\!]_\nu$ auf der Kategorie $\mathcal{PCPO}$, bzw. ein zweistelliger Funktorterm GT beschreibt für jede Typbelegung ν einen Funktor $[\![GT]\!]_\nu : \mathcal{PCPO} \times \mathcal{PCPO} \to \mathcal{PCPO}$.

Beweis:
Per Induktion über den Aufbau der Funktorterme. Für einen einstelligen Funktorterm FT ist zu zeigen:

1. $[\![FT]\!]_\nu$ ist ein Paar von Abbildungen:
$$\begin{array}{rcl} [\![FT^O]\!]_\nu & : & Obj(\mathcal{PCPO}) \to Obj(\mathcal{PCPO}) \\ [\![FT^M]\!]_\nu & : & Mor(\mathcal{PCPO}) \to Mor(\mathcal{PCPO}) \end{array}$$
2. Für Morphismen $f \in \mathcal{PCPO}(X,Y)$ und $g \in \mathcal{PCPO}(Y,Z)$ gilt:
   (a) $([\![FT^M]\!]_\nu)(f) \in \mathcal{PCPO}(([\![FT^O]\!]_\nu)(X), ([\![FT^O]\!]_\nu)(Y))$
   (b) $([\![FT^M]\!]_\nu)(g \circ f) = ([\![FT^M]\!]_\nu)(g) \circ ([\![FT^M]\!]_\nu)(f)$
   (c) $([\![FT^M]\!]_\nu)(id_X) = id_{([\![FT^O]\!]_\nu)(X)}$

Für einen zweistelligen Funktorterm GT ist entsprechend der Definition der Produktkategorie $\mathcal{PCPO} \times \mathcal{PCPO}$ zu zeigen:

1. $[\![GT]\!]_\nu$ ist ein Paar von Abbildungen:
$$\begin{array}{rcl} [\![GT^O]\!]_\nu & : & Obj(\mathcal{PCPO}) \times Obj(\mathcal{PCPO}) \to Obj(\mathcal{PCPO}) \\ [\![GT^M]\!]_\nu & : & Mor(\mathcal{PCPO}) \times Mor(\mathcal{PCPO}) \to Mor(\mathcal{PCPO}) \end{array}$$
2. Für Morphismen $f \in \mathcal{PCPO}(X_1,Y_1)$, $g \in \mathcal{PCPO}(Y_1,Z_1)$ und $f' \in \mathcal{PCPO}(X_2,Y_2)$, $g' \in \mathcal{PCPO}(Y_2,Z_2)$ gilt:
   (a) $([\![GT^M]\!]_\nu)(f,f') \in \mathcal{PCPO}(([\![GT^O]\!]_\nu)(X_1,X_2), ([\![GT^O]\!]_\nu)(Y_1,Y_2))$
   (b) $([\![GT^M]\!]_\nu)(g \circ f, g' \circ f') = ([\![GT^M]\!]_\nu)(g,g') \circ ([\![GT^M]\!]_\nu)(f,f')$
   (c) $([\![GT^M]\!]_\nu)(id_X, id_Y) = id_{([\![GT^O]\!]_\nu)(X,Y)}$

Die Eigenschaften 1 und 2a folgen in allen Fällen direkt aus der Definition 5.11 für die Semantik von Funktortermen. Die Eigenschaften 2b und 2c ergeben sich durch einfaches Nachrechnen, wobei in den Fällen U, $**$ und $++$ ausgenützt wird, daß nur strikte Operationen zugelassen sind.

Das folgende Beispiel erläutert die Aussage des Theorems 5.13.

Beispiel 5.3

Die Eigenschaften 2b und 2c kann man für konkrete Funktorterme auch mit Hilfe des Theorems 5.12 nachweisen. Für Terme $t_1 \in DT_{\Sigma,\tau_1 \to \tau_2}$ und $t_2 \in DT_{\Sigma,\tau_2 \to \tau_3}$ mit t_1`[⊥]=`⊥ und t_2`[⊥]=`⊥, d.h. t_1 und t_2 sind Terme für strikte Operationen, rechnet man in der HOLCF-Theorie `ccc1` für $\langle **, K_\alpha, U\rangle$ leicht nach:

```
   ssplit[Λxy.x::α ## lift[up oo (t2 oo t1)][y]]
=  ssplit[Λxy.x::α ## lift[up oo t2][y]] oo
   ssplit[Λxy.x::α ## lift[up oo t1][y]]
```

Ebenso zeigt man:

```
ssplit[Λxy.x::α ## lift[up oo ID][y]] = ID
```

Theorem 5.14 *Funktorterme beschreiben lokal stetige Funktoren*

Sei Th ein Theorie, $\mathcal{M}$ ein beliebiges Modell für Th und $\mathcal{PCPO}$ die dadurch induzierte Kategorie.

Dann beschreibt ein einstelliger Funktorterm FT für jede Typbelegung ν einen lokal stetigen Endofunktor $[\![FT]\!]_\nu$ auf der Kategorie $\mathcal{PCPO}$, bzw. ein zweistelliger Funktorterm GT beschreibt für jede Typbelegung ν einen lokal stetigen Funktor $[\![GT]\!]_\nu : \mathcal{PCPO} \times \mathcal{PCPO} \to \mathcal{PCPO}$.

Beweis:
Per Induktion über den Aufbau der Funktorterme. Für einen einstelligen Funktorterm FT ist für alle Belegungen ν zu zeigen:

1. $f_1 \sqsubseteq f_2 \Longrightarrow ([\![FT^M]\!]_\nu)(f_1) \sqsubseteq ([\![FT^M]\!]_\nu)(f_2)$
2. $([\![FT^M]\!]_\nu)(\bigsqcup_{n\in\omega} f_n) = \bigsqcup_{n\in\omega}([\![FT^M]\!]_\nu)(f_n)$

Für einen zweistelligen Funktorterm GT ist für alle Belegungen ν zu zeigen:

1. $f_1 \sqsubseteq f_2 \wedge g_1 \sqsubseteq g_2 \Longrightarrow ([\![GT^M]\!]_\nu)(f_1, g_1) \sqsubseteq ([\![GT^M]\!]_\nu)(f_2, g_2)$
2. $([\![GT^M]\!]_\nu)(\bigsqcup_{n\in\omega} f_n, \bigsqcup_{n\in\omega} g_n) = \bigsqcup_{n\in\omega}([\![GT^M]\!]_\nu)(f_n, g_n)$

Die Eigenschaften zeigt man in den einzelnen Fällen wieder durch leichtes Nachrechnen. Die Monotonie- und Stetigkeitseigenschaften ergeben sich dabei aus den entsprechenden Eigenschaften der zugrundeliegenden Abbildungen für Operationen.

Auch in diesem Fall kann für konkrete Funktorterme ein Teil des Beweises über Verwendung von Theorem 5.12 erbracht werden. Betrachten wir dazu folgendes Beispiel.

Beispiel 5.4

Die lokale Monotonie und Stetigkeit des Funktors $[\![FT]\!]_\nu$ für den Beispielterm $FT = \langle **, K_\alpha, U\rangle$ ergeben sich unter Benutzung von Theorem 5.12 im wesentlichen aus der Monotonie und Stetigkeit des HOLCF-Terms

```
λf.ssplit[Λxy.x::α ## lift[up oo f][y]]
```

Man zeigt in der Theorie `ccc1` durch Verwendung der Taktik `contX_tacR` automatisch das Prädikat `contX(t)` und damit `monofun(t)` und `contlub(t)`, wenn `t` für den obigen Term steht.

5.3 Konservative Theorieerweiterung durch pcpo-Typen

Abschnitt 5.2 zeigt, daß sich durch Funktorterme FT Familien von lokal stetigen Funktoren auf der CPO-Kategorie $\mathcal{PCPO}$ darstellen lassen. Die Indizierung der Familie erfolgt über die in den Funktortermen vorkommenden Typvariablen bzw. deren Interpretationen unter Variablenbelegungen ν. Für jede konkrete Belegung ν erhalten wir einen speziellen Funktor. Aus Abschnitt 5.1 wissen wir, daß es zu jedem der Funktoren $[\![FT]\!]_\nu$ eine initiale $[\![FT]\!]_\nu$-Algebra gibt. Dies nützen wir aus, um einen polymorphen Typkonstruktor zu definieren, der für jede Belegung ν die Argumentbereiche in das Objekt der initialen $[\![FT]\!]_\nu$-Algebra abbildet. Die Argumentbereiche sind dabei gerade die Interpretationen der in FT vorkommenden Typvariablen unter der jeweiligen Belegung ν. Die Axiomatisierung des neuen Typkonstruktors erfolgt gemäß der lokalen Kriterien für initiale F-Algebren, die wir mit Hilfe der Ergebnisse aus Abschnitt 5.2, speziell Theorem 5.12, in der Logik HOLCF ausdrücken können.

Das gerade skizzierte Verfahren zur Definition eines neuen Typkonstruktors für pcpo-Typen wird nun im Stil von Abschnitt 2.6 zu einem Mechanismus für konservative Theorieerweiterung zusammengefaßt.

Definition 5.12 *Erweiterung durch einen* pcpo*-Typkonstruktor*

Sei $Th_1 = (\Sigma_1, Ax_1, KAx_1)$ eine Theorie mit Signatur $\Sigma_1 = (\Omega_1, C_1, KS_1)$ und Typsignatur $\Omega_1 = (K_1, \leq_1, TC_1)$. Sei fernerhin $\texttt{ccc1} \subseteq Th_1$.

Die Erweiterung der Theorie Th_1 durch einen neuen, evtl. rekursiven, pcpo-Typkonstruktor tc wird dann folgendermaßen notiert:

$Th_2 = Th_1 +_{rec}$

	$\overbrace{\qquad\qquad}^{m\text{-mal}}$
rectype	$tc\texttt{:}(\overbrace{\texttt{pcpo},\ldots,\texttt{pcpo}}\texttt{)pcpo}$
functor_term	FT
rep_op	$\texttt{rep}\texttt{:}(\alpha_1 \ldots \alpha_m tc) \to FT(\alpha_1 \ldots \alpha_m tc)$
abs_op	$\texttt{abs}\texttt{:}FT(\alpha_1 \ldots \alpha_m tc) \to (\alpha_1 \ldots \alpha_m tc)$

Damit die Theorieerweiterung durchgeführt werden kann, müssen folgende syntaktische Bedingungen erfüllt sein:

B1: Anforderungen an den Typkonstruktor.

$$\neg\exists wk.\ tc\texttt{:}(w)k \in TC_1$$

Der Typkonstruktor tc ist neu.

B2: Anforderungen an den Funktorterm FT. Der Funktorterm FT ist ein einstelliger Funktorterm über der Theorie Th_1 gemäß Definition 5.10. Insbesondere gilt:

$$TV(FT) = \{\alpha_1, \ldots, \alpha_m\} \subseteq \Xi_{\texttt{pcpo}}$$

B3: Anforderungen an die Operationen abs, rep für Abstraktion und Repräsentation:

$$\neg\exists \rho.\ (\texttt{rep}, \rho) \in C_1 \vee (\texttt{abs}, \rho) \in C_1$$

Die Konstanten abs, rep sind neu. Die im Typ der Operationen vorkommende Anwendung $FT(\alpha_1 \ldots \alpha_m tc)$ des Funktorterms FT ist der expandierte HOLCF-Typterm gemäß der Definition 5.10 für die Anwendungen von Funktortermen auf Typterme.

Die Komponenten der neuen Theorie Th_2 werden dann definiert wie folgt:

$$
\begin{array}{rcl}
K_2 & = & K_1 \\
\leq_2 & = & \leq_1 \\
TC_2 & = & TC_1 \cup \{tc\texttt{:}(\underbrace{\texttt{pcpo},\ldots,\texttt{pcpo}}_{m\text{-mal}})\texttt{pcpo}\} \\
\Omega_2 & = & (K_2, \leq_2, TC_2) \\
C_2 & = & C_1 \cup \{\texttt{rep:}(\alpha_1 \ldots \alpha_m tc) \rightarrow FT(\alpha_1 \ldots \alpha_m tc), \\
 & & \quad \texttt{abs:}FT(\alpha_1 \ldots \alpha_m tc) \rightarrow (\alpha_1 \ldots \alpha_m tc)\} \\
KS_2 & = & KS_1 \\
 & & \\
\Sigma_2 & = & (\Omega_2, C_2, KS_2) \\
Ax_2 & = & Ax_1 \cup \{ \\
 & & \quad \texttt{rep oo abs} \;=\; \texttt{ID}, \\
 & & \quad \texttt{abs oo rep} \;=\; \texttt{ID}, \\
 & & \quad \texttt{fix}[\Lambda\texttt{f.abs oo}\ FT\texttt{f oo rep}] \;=\; \texttt{ID} \\
 & & \quad \} \\
KAx_2 & = & KAx_1 \\
 & & \\
Th_2 & = & (\Sigma_2, Ax_2, KAx_2)
\end{array}
$$

Die im dritten neuen Axiom vorkommende Anwendung $FT\texttt{f}$ des Funktorterms FT auf die Operation `f` versteht sich als die expandierte Form der Anwendung eines Funktorterms auf Operationsterme gemäß Definition 5.10.

Aus der obigen Definition für die Komponenten der Theorie Th_2 ist ersichtlich, daß zur Typsignatur Ω_1 lediglich der neue Typkonstruktor tc hinzugefügt wird, und die Signatur Σ_1 um die neuen Konstanten `rep` und `abs` erweitert wird. Ansonsten bleibt die Signatur unverändert. Die Bedingungen B1 bis B3 garantieren die Wohlgeformtheit der neuen Signaturen Ω_2 und Σ_2, sowie der Theorie Th_2.

Beispiel 5.5 *Polymorphe Ströme*

In der eben definierten Notation für die Erweiterung durch einen pcpo-Typkonstruktor sieht die Einführung des rekursiven Typs der polymorphen Ströme über Parametertyp $\alpha_{\texttt{pcpo}}$ wie folgt aus:

`Stream` = `ccc1`$+_{rec}$

rectype	`stream:(pcpo)pcpo`
functor_term	$\langle **, K_\alpha, U\rangle$
rep_op	`stream_rep:`α `stream` $\rightarrow$ (α `**` (α `stream`)`u`)
abs_op	`stream_abs:`(α `**` (α `stream`)`u`) $\rightarrow$ α `stream`

Das Beispiel erfüllt alle Anforderungen der Definition. Folgende Signatur und Axiome werden im Beispiel zur Theorie `ccc1` neu hinzugefügt.

```
stream : (pcpo)pcpo
stream_rep : α stream → (α ** (α stream)u)
stream_abs : (α ** (α stream)u) → α stream

stream_rep oo stream_abs = ID
stream_abs oo stream_rep = ID
fix[Λf. stream_abs oo
        (ssplit[Λxy. x ## lift[up oo f][y]])
        oo stream_rep] = ID
```

Die Restriktion von Σ_2-Modellen auf die Signatur Σ_1 wird für die Erweiterung mittels $+_{rec}$ folgendermaßen definiert.

Definition 5.13 *Restriktion bei* $+_{rec}$

Sei die Theorie Th_2 durch eine Erweiterung um einen neuen Typkonstruktor tc mittels $+_{rec}$ entstanden, und sei $\mathcal{M}2$ ein Modell für die Signatur Σ_2. Dann ist die Restriktion $\mathcal{M}2\mid_{\Sigma_1}$ dasjenige Modell für Σ_1, das aus $\mathcal{M}2$ entsteht, wenn man im Typmodell $\mathcal{T}\mathcal{M}2$ die Interpretation $tc^{\mathcal{T}\mathcal{M}2}_{(\mathtt{pcpo},\ldots,\mathtt{pcpo})\mathtt{pcpo}}$ für den neuen Konstruktor tc entfernt und im Modell $\mathcal{M}2$ die Interpretationen für die Konstanten `rep` und `abs` vergißt.

Die Tatsache, daß $\mathcal{T}\mathcal{M}2\mid_{\Omega_1}$ ein Modell für Ω_1 ist und daß $\mathcal{M}2\mid_{\Sigma_1}$ ein Modell für Σ_1 ist, ist offensichtlich.

Der Beweis für das folgende Restriktionslemma bei $+_{rec}$ ist trivial.

Theorem 5.15 *Restriktionslemma bei* $+_{rec}$

Sei die Theorie Th_2 durch eine Erweiterung um einen neuen Typkonstruktor tc mittels $+_{rec}$ entstanden, und sei $\mathcal{M}2 = (\mathcal{T}\mathcal{M}2, \mathcal{C}2)$ ein Modell für die Signatur Σ_2.

Dann gilt für alle Typterme $\tau \in T_{\Omega_1}$ und Belegungen ν:

$$fgt_{c,k}(\mathcal{T}\mathcal{M}2[\![\tau]\!]^{\Omega_2}_{\nu}) = \mathcal{T}\mathcal{M}2\mid_{\Omega_1}[\![\tau]\!]^{\Omega_1}_{\nu}$$

wobei

$$c = lst_{\Omega_2}(\tau) \quad \text{und} \quad k = lst_{\Omega_1}(\tau)$$

Weiterhin gilt für alle Terme $t \in DT_{\Sigma_1}$ und alle Belegungen ν, η mit $\eta\ sat\,(\nu, GC(t))$:

$$\mathcal{M}2[\![t]\!]^{\Sigma_2}_{\nu,\eta} = \mathcal{M}2\mid_{\Sigma_1}[\![t]\!]^{\Sigma_1}_{\nu,\eta}$$

Beweis: leichte Induktion über den Aufbau der Terme τ bzw. t und Verwendung der Tatsache, daß der Typkonstruktor tc und die Konstanten `rep` und `abs` in der alten Signatur Σ_1 überhaupt nicht vorkommen.

Nachdem die Restriktion definiert ist, können wir uns nun der konservativen Modellerweiterung im Fall $+_{rec}$ zuwenden. Sei also Th_2 aus Th_1 durch $+_{rec}$ entstanden, und sei ein Modell $\mathcal{M}1$ für Th_1 gegeben mit $\mathcal{M}1 = (\mathcal{TM}1, \mathcal{C}1)$ und $\mathcal{TM}1 = (\mathcal{K}1, \mathcal{TC}1)$.

Um ein Modell für die neue Theorie Th_2 zu konstruieren, müssen wir das Typmodell $\mathcal{TM}1$ um eine Interpretation $tc^{\mathcal{TM}2}_{(\texttt{pcpo},\ldots,\texttt{pcpo})\texttt{pcpo}}$ für den neuen Konstruktor tc:(pcpo, ..., pcpo)pcpo erweitern, und wir müssen das Signaturmodell $\mathcal{M}1$ um eine Interpretation für die beiden Konstanten rep und abs erweitern.

Gemäß Theorem 5.14 ist für jedes ν die Interpretation $[\![FT]\!]_\nu$ des einstelligen Funktorterms FT ein lokal stetiger Funktor auf der durch Modell $\mathcal{M}$ induzierten CPO-Kategorie $\mathcal{PCPO}$. Deswegen gibt es nach Theorem 5.7 für jede Belegung ν eine initiale $[\![FT]\!]_\nu$-Algebra (D_ν, θ_ν). Die Interpretation $[\![FT]\!]_\nu$ und damit auch (D_ν, θ_ν) ist wegen $TV(FT) = \{\alpha_1, \ldots, \alpha_m\}$ nur von der Belegung für die Typvariablen $\alpha_1, \ldots, \alpha_m$ abhängig. Dies erlaubt uns, folgende Interpretation für den Typkonstruktor tc zu definieren:

Für alle Belegungen ν sei:

$$tc^{\mathcal{TM}2}_{(\texttt{pcpo},\ldots,\texttt{pcpo})\texttt{pcpo}}(\nu(\alpha_1), \ldots, \nu(\alpha_m)) = D_\nu$$

Damit ist die Interpretation der Typkonstanten tc festgelegt. Die Reihenfolge der Argumente α_i für die Interpretation des Typkonstruktors tc ergibt sich dabei durch die Reihenfolge ihrer Aufschreibung im Definitionsformular $+_{rec}$ bei den Typen der Operationen abs und rep.

Sei jetzt

$$\{\beta_1, \ldots, \beta_m\} = \{\alpha_1, \ldots, \alpha_m\} \text{ mit } \beta_1 \prec \ldots \prec \beta_m$$

d.h. die Typvariablen β_j sind die α_j aufgereiht gemäß der kanonischen Ordnung $\prec$ auf Typvariablen. Nach Theorem 5.7 ist der Morphismus $\theta_\nu : ([\![FT]\!]_\nu)D_\nu \to D_\nu$ ein Isomorphismus mit Inverser $\theta_\nu^{-1} : D_\nu \to ([\![FT]\!]_\nu)D_\nu$. Die Morphismen der Kategorie $\mathcal{PCPO}$ sind per Definition Tripel. Seien also die Isomorphismen θ_ν und θ_ν^{-1} durch folgende Tripel gegeben:

$$\begin{aligned} \theta_\nu &= (op_\nu, ([\![FT]\!]_\nu)D_\nu, D_\nu) \\ \theta_\nu^{-1} &= (op_\nu^{-1}, D_\nu, ([\![FT]\!]_\nu)D_\nu) \end{aligned}$$

Wir definieren für alle Belegungen ν:

$$\begin{aligned} \texttt{rep}^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m)) &= op_\nu^{-1} \\ \texttt{abs}^{\mathcal{M}2}(\nu(\beta_1), \ldots, \nu(\beta_m)) &= op_\nu \end{aligned}$$

Damit sind alle Anwendungen der polymorphen Konstanten rep und abs definiert, und somit sind die Interpretationen $\texttt{rep}^{\mathcal{M}2}$ und $\texttt{abs}^{\mathcal{M}2}$ per Extensionalität eindeutig definiert.

Folgende Definitionen beschließen nun die Modellkonstruktion für den Fall der Erweiterung um einen neuen pcpo-Typkonstruktor:

Definition 5.14 *Modellerweiterung bei* $+_{rec}$

Das Typmodell für die Typsignatur Ω_2 wird festgelegt als:

$$\mathcal{TM}2 \quad = \quad (\mathcal{K}1, \mathcal{TC}1 + tc^{\mathcal{TM}2}_{(\mathtt{pcpo},\ldots,\mathtt{pcpo})\mathtt{pcpo}})$$

Ein Modell für die Signatur Σ_2 ergibt sich durch folgende Definition:

$$\mathcal{M}2 = (\mathcal{TM}2, \mathcal{C}1 + \{\mathtt{rep}^{\mathcal{M}2}, \mathtt{abs}^{\mathcal{M}2}\})$$

Die Wohldefiniertheit der einzelnen Interpretationen folgt im wesentlichen aus Theorem 5.7, das die Existenz der initialen $[\![FT]\!]_\nu$-Algebra mit ihren Anteilen (D_ν, θ_ν) garantiert.

Theorem 5.16

Das in Definition 5.14 definierte Modell $\mathcal{M}2$ ist ein Modell für die Theorie Th_2 und es gilt:

$$\mathcal{M}2\,|_{\Sigma_1} = \mathcal{M}1$$

Die Theorie Th_2 ist also durch konservative Erweiterung der Theorie Th_1 entstanden.

Beweis:
Die Tatsache, daß $\mathcal{M}2$ ein Modell für Th_2 ist, folgt im wesentlichen aus Theorem 5.7, das die Existenz initialer F-Algebren garantiert und gleichzeitig auch die Gültigkeit der neuen Axiome in Theorie Th_2 sicherstellt.

Die Restriktionseigenschaften werden trivial per Definition des erweiterten Modells erfüllt.

Damit ist gezeigt, daß die Axiomatisierung rekursiver `pcpo`-Typen (bzgl. covarianter Funktoren) im Stil der Logik LCF ebenfalls zu einer konservativen Theorieerweiterung führt, die Existenz von Modellen und damit auch die Konsistenz ist also sichergestellt. Die in diesem Kapitel präsentierte Theorie motiviert und rechtfertigt die dabei verwendeten Axiome, was eine Verbesserung in bezug auf die Darstellungen derselben Problematik in [Pau87] und [Mon85] bedeutet.

Kapitel 6

Beispiele für Datentypen in HOLCF

In Kapitel 4 endete die Entwicklung der Logik HOLCF mit der Einführung der Theorie `ccc1`. In diesem Kapitel wird die Logik HOLCF unter Verwendung der Ergebnisse aus Kapitel 5 um Theorien für Standarddatentypen erweitert. Abbildung 6.1 zeigt den Aufbau der Theorien, die im folgenden beschrieben werden.

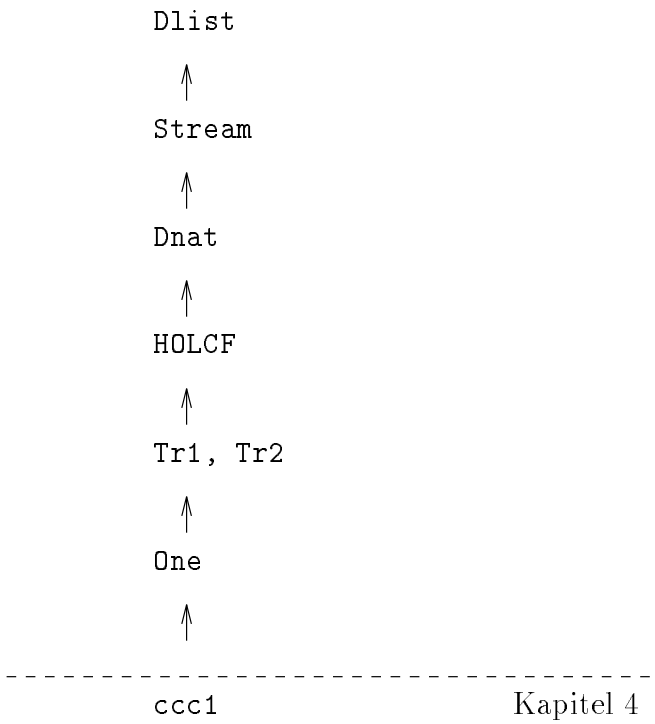


Abbildung 6.1: Standarddatentypen in HOLCF

Den Datentypen, die in den Theorien `One` bis `Dlist` eingeführt werden, ist gemeinsam, daß sie als initiale F-Algebren bezüglich eines Funktors F axiomatisiert sind, der durch einen

Funktorterm *FT* beschrieben wird. Diese Vorgehensweise wird durch die Ergebnisse aus Kapitel 5 gerechtfertigt.

Zuerst wird die Theorie `One` mit dem Typ `one` eingeführt, der neben dem kleinsten Element nur ein definiertes Element im Träger enthält. Danach wird in der Theorie `Tr1` der Typ `tr` der programmiersprachlichen Wahrheitswerte eingeführt. In der darauffolgenden Theorie `Tr2` werden für den Typ `tr` die sequentielle Konjunktion `andalso`, die sequentielle Disjunktion `orelese` und die Negation `neg` als Operationen definiert. Mit der Theorie `Tr2` liegt dann eine logische Theorie vor, die den gesamten Standardumfang der Logik LCF, wie etwa in [Pau87] beschrieben, in einer Version höherer Stufe umfaßt. Um dies zu verdeutlichen, wurde die Theorie `HOLCF` eingeführt, die lediglich einen neuen Theorienamen `HOLCF` einführt und alle bis dato eingeführten Mengen von Simplifikationsregeln zu einer einzigen zusammenfaßt.

Danach werden die rekursiven Datentypen der natürlichen Zahlen, Ströme und Listen axiomatisiert, und für jeden dieser Typen wird eine Vielzahl von Theoremen bewiesen, die sich bei der praktischen Beweisarbeit als sinnvoll erwiesen haben. Besonders interessant sind dabei die Herleitungen von Induktions- und Co-Induktionsprinzipien für die jeweiligen Typen. Diese Herleitungen nützen die Ausdrucksstärke der Logik HOLCF aus und können auch in vergleichbarer Weise in der Logik LCF nicht durchgeführt werden.

Aus der Axiomatisierung des jeweiligen Typs wird ein sogenanntes *Take-Lemma* abgeleitet, wobei der Beweis von den höherstufigen Ausdrucksmitteln Gebrauch macht. Aus dem Take-Lemma kann dann das Co-Induktionsprinzip für den Typ abgeleitet werden. Im Fall von strikten Datentypen, wie etwa den Listen, kann aus dem Take-Lemma weiterhin ein strukturelles Induktionsprinzip abgeleitet werden, das keine Beschränkung bezüglich der Zulässigkeit des Induktionsprädikats enthält. Dies stellt einen deutlichen Vorteil gegenüber der Logik LCF dar, wo die strukturelle Induktion durch die Verwendung der Fixpunktinduktion abgeleitet werden muß und somit stets eine Beschränkung der Induktion auf zulässige Prädikate resultiert. Für Datentypen mit nicht-strikten Konstruktoren wird in HOLCF die strukturelle Induktion ebenfalls über die Fixpunktinduktion hergeleitet. Aufgrund der Logik höherer Stufe gibt es zwar auch in diesem Fall alternative Wege, aber wegen der nicht-strikten Konstruktoren läßt sich hier die Beschränkung auf zulässige Prädikate nicht umgehen.

Wie auch das Kapitel 4 ist dieses Kapitel in Abschnitte untergliedert, von denen jeder die Axiomatisierung eines Datentyps sowie die dazu abgeleiteten Theoreme beschreibt. Bei den Theoremen werde ich nur auf die Herleitung der Induktions- und Co-Induktionsprinzipien ausführlicher eingehen, da hier der wesentliche Vorteil der Logik HOLCF gegenüber LCF deutlich wird. Die Ableitung der restlichen Theoreme für jeden Typ erfolgte gemäß dem LCF-Standard und kann zum Beispiel in [Pau84, Pau87] nachgelesen werden.

6.1 Der Datentyp `one`

6.1.1 Die Theorie `One`

Der Datentyp `one` enthält nur ein definiertes Element im Träger. Er wird als die initiale Lösung der trivialen Bereichsgleichung

```
one = (void)u
```

axiomatisiert. Der zugehörige Funktorterm FT ist durch

$$FT = \langle U, K_{\texttt{void}} \rangle$$

gegeben. Da die Bereichsgleichung nicht rekursiv ist, verkümmert das üblicherweise dritte Axiom der Wohlfundiertheit

$$\texttt{ID} \;=\; \texttt{fix}[\Lambda \texttt{f}.\ \texttt{abs\_one oo}\ FT\ \texttt{f oo rep\_one}]$$

zu einer trivialen Aussage und kann daher entfallen. Die Theorie ist in Abbildung 6.2 dargestellt.

```
(* Introduce atomic type: one                                    *)
(* domain equation: one = (void)u                                *)
(* functor term   :  FT = <U,K_{void}>                           *)

One = ccc1+

types one 0
arities one :: pcpo

consts
        abs_one         ::  (void)u → one
        rep_one         ::  one → (void)u
        one             ::  one
        one_when        ::  γ → one → γ

rules

  abs_one_iso   abs_one[rep_one[u]] = u
  rep_one_iso   rep_one[abs_one[x]] = x

  one_def       one ≡ abs_one[up[⊥]]
  one_when_def  one_when ≡ (Λc u.lift[Λx.c][rep_one[u]])

end
```

Abbildung 6.2: Theorie `One`

Zusätzlich zur Axiomatisierung des Datentyps werden noch die beiden Konstanten `one` und `one_when` eingeführt, die für das einzige definierte Element im Träger und ein Diskriminatorfunktional stehen.

6.1.2 Theoreme der Theorie `One`

Die abgeleiteten Theoreme sind in Abbildung 6.3 dargestellt. Die Theoreme `Exh_one` und `oneE` ermöglichen Beweise per Fallunterscheidung. `dist_less_one` und `dist_eq_one` sind Listen von Theoremen. Dies wird durch die Verwendung der ML-Schreibweise `[... , ...]` für Listen verdeutlicht. Die beiden Listen enthalten Theoreme über die Gleichheit und Ungleichheit von Elementen des Typs `one`. Die Listenschreibweise wird im folgenden öfters eingesetzt, um Namen für Theoreme zu sparen. Das Theorem `flat_one` zeigt die offensichtliche Tatsache, daß der Typ `one` flach geordnet ist. Die Theoreme der Liste `one_when` zeigen das Verhalten des Diskriminatorfunktionals `one_when`.

```
Exh_one        z = ⊥ ∨ z = one
oneE           ⟦ p=⊥⟹Q; p = one⟹Q⟧⟹Q

dist_less_one  [ ¬one ⊑ ⊥ ]
dist_eq_one    [ one ¬= ⊥ ,
                 ⊥ ¬= one ]

flat_one       flat(one)

one_when       [ one_when[x][⊥] = ⊥ ,
                 one_when[x][one] = x ]
```

Abbildung 6.3: Theoreme der Theorie `One`

6.2 Der Datentyp der Wahrheitswerte `tr`

6.2.1 Die Theorie `Tr1`

Der Datentyp `tr` der programmiersprachlichen Wahrheitswerte wird als die initiale Lösung der trivialen Bereichsgleichung

```
tr = one ++ one
```

axiomatisiert. Der zugehörige Funktorterm FT ist durch

$$FT = \langle ++, K_{\texttt{one}}, K_{\texttt{one}} \rangle$$

gegeben. Da die Bereichsgleichung nicht rekursiv ist, verkümmert auch hier das üblicherweise dritte Axiom der Wohlfundiertheit

```
ID = fix[Λf. abs_tr oo FT f oo rep_tr]
```

zu einer trivialen Aussage und kann entfallen. Die Theorie ist in Abbildung 6.4 dargestellt. Neben der Axiomatisierung des Datentyps werden noch die Konstanten TT und FF für die booleschen Wahrheitswerte eingeführt. Die Konstante tr_when steht für das Diskriminatorfunktional auf dem Typ tr.

```
(* Introduce atomic type tr:                                    *)
(* domain equation: tr = one ++ one                             *)
(* functor term   : FT = <++,K_{one},K_{one}>                    *)

Tr1 = One +

types   tr 0
arities tr :: pcpo

consts
        abs_tr          ::  one ++ one → tr
        rep_tr          ::  tr → one ++ one
        TT              ::  tr
        FF              ::  tr
        tr_when         ::  γ → γ → tr → γ

rules

  abs_tr_iso   abs_tr[rep_tr[u]] = u
  rep_tr_iso   rep_tr[abs_tr[x]] = x

  TT_def       TT ≡ abs_tr[sinl[one]]
  FF_def       FF ≡ abs_tr[sinr[one]]

  tr_when_def  tr_when ≡ (Λe1 e2 t.when[Λx.e1][Λy.e2][rep_tr[t]])

end
```

Abbildung 6.4: Theorie Tr1

6.2.2 Theoreme der Theorie Tr1

Die Theoreme der Theorie Tr1 sind in Abbildung 6.5 dargestellt. Die Listen dist_less_tr und dist_eq_tr enthalten Theoreme über Gleichungen und Ungleichungen zwischen Elementen des Typs tr, die Theoreme Exh_tr und trE ermöglichen Fallunterscheidung für den Typ. Das Theorem flat_tr zeigt, daß tr flach geordnet ist, und die Liste tr_when beschreibt das Verhalten des Diskriminatorfunktionals.

```
dist_less_tr  [ ¬TT ⊑ ⊥ ,  ¬FF ⊑ ⊥ ,  ¬TT ⊑ FF ,  ¬FF ⊑ TT ]
dist_eq_tr    [ TT ¬= ⊥ ,  FF ¬= ⊥ ,  TT ¬= FF ,  ⊥ ¬= TT ,  ⊥ ¬= FF ,
                FF ¬= TT ]

Exh_tr        t = ⊥ ∨ t = TT ∨ t = FF
trE           ⟦ p = ⊥⟹Q; p = TT⟹Q; p = FF⟹Q⟧⟹Q

flat_tr       flat(TT)

tr_when       [ tr_when[x][y][⊥] = ⊥ ,
                tr_when[x][y][TT] = x ,
                tr_when[x][y][FF] = y ]
```

Abbildung 6.5: Theoreme der Theorie `Tr1`

6.2.3 Die Theorie `Tr2`

In der Theorie `Tr2` werden die üblichen Operationen auf den Wahrheitswerten eingeführt. Die Theorie ist in Abbildung 6.6 dargestellt.

```
Tr2 = Tr1 +

consts
        @cifte       :: tr ⇒ γ ⇒ γ ⇒ γ   ( (3If _/ (then _/ else _) fi)  60)
        Icifte       :: tr → γ → γ → γ
        @andalso     :: tr ⇒ tr ⇒ tr   ( _ andalso _   [36,35] 35)
        cop @andalso :: tr → tr → tr   ( trand )
        @orelse      :: tr ⇒ tr ⇒ tr   ( _ orelse _    [31,30] 30)
        cop @orelse  :: tr → tr → tr   ( tror )
        neg          :: tr → tr

translations  x andalso y  ⇒  trand[x][y]
              x orelse y   ⇒  tror[x][y]
              If b then e1 else e2 fi  ⇒  Icifte[b][e1][e2]

rules

  ifte_def    Icifte ≡ (Λt e1 e2.tr_when[e1][e2][t])
  andalso_def  trand ≡ (Λt1 t2.tr_when[t2][FF][t1])
  orelse_def    tror ≡ (Λt1 t2.tr_when[TT][t2][t1])
  neg_def        neg ≡ (Λt. tr_when[FF][TT][t])

end
```

Abbildung 6.6: Theorie `Tr2`

Das Mixfix-Konstrukt `If_then_else_fi` dient als Schreiberleichterung für die Operation `Icifte` zur Fallunterscheidung. Für die Operationen der sequentiellen Konjunktion `trand` und Disjunktion `tror` werden Infix-Konstrukte `andalso` und `orelse` vereinbart. Die Negation `neg` wird in der üblichen Präfixnotation für Operationen geschrieben. Der etwas mühsame Mechanismus zur Einführung von Mixfix- und Infix-Konstrukten ist aus Abschnitt 4.8.7 bekannt. Alle Operationen werden unter Verwendung des Diskriminatorfunktionals `tr_when` definiert, welches in der praktischen Anwendung wohl eher eine untergeordnete Rolle spielt.

6.2.4 Theoreme der Theorie Tr2

Die Theoreme der Theorie `Tr2` sind in Abbildung 6.7 dargestellt. Sie beschreiben das Verhalten der neu eingeführten Operationen und bedürfen keiner weiteren Erläuterung.

```
andalso_thms  [ (TT andalso y) = y ,
                (FF andalso y) = FF ,
                (⊥ andalso y) = ⊥ ,
                (x andalso TT) = x ]

orelse_thms   [ (TT orelse y) = TT ,
                (FF orelse y) = y ,
                (⊥ orelse y) = ⊥ ,
                (x orelse FF) = x ]

neg_thms      [ neg[TT] = FF ,
                neg[FF] = TT ,
                neg[⊥] = ⊥ ]

ifte_thms     [ If ⊥ then e1 else e2 fi = ⊥ ,
                If FF then e1 else e2 fi = e2 ,
                If TT then e1 else e2 fi = e1 ]
```

Abbildung 6.7: Theoreme der Theorie `Tr2`

6.3 Die Theorie HOLCF

Die Theorie `HOLCF` führt lediglich den Theorienamen `HOLCF` ein. Die Theorie `HOLCF` wird vereinbart, weil mit der Theorie `Tr2` eine logische Theorie vorliegt, die den Standardumfang der Logik LCF vollständig abdeckt. Die triviale Theorie `HOLCF` ist in Abbildung 6.8 dargestellt. Die Abbildung zeigt auch die Vereinbarung für die Simplifikatorstruktur `HOLCF_ss`, welche die Simplifikatorstruktur `ccc1_ss` aus Abschnitt 4.13, Abbildung 4.59 um Simplifikationsregeln bezüglich der neuen Datentypen `one` und `tr` anreichert.

```
HOLCF = Tr2
```

Konfiguration des Simplifikators:

```
val HOLCF_ss = ccc1_ss
                addsimps one_when
                addsimps dist_less_one
                addsimps dist_eq_one
                addsimps dist_less_tr
                addsimps dist_eq_tr
                addsimps tr_when
                addsimps andalso_thms
                addsimps orelse_thms
                addsimps neg_thms
                addsimps ifte_thms;
```

Abbildung 6.8: Theorie HOLCF

6.4 Der Datentyp der natürlichen Zahlen dnat

6.4.1 Die Theorie Dnat

Die Theorie Dnat führt den Datentyp der natürlichen Zahlen dnat als initiale Lösung der Bereichsgleichung

```
dnat = one ++ dnat
```

ein. Der zugehörige Funktorterm FT ist durch

$$FT = \langle ++, K_{\texttt{one}}, I \rangle$$

gegeben. Die Bezeichnung dnat (d für Datentyp) rührt daher, daß in der bisher angesammelten Signatur bereits eine Theorie Nat mit Typkonstruktor nat vorkommt, es mußten also neue Bezeichner verwendet werden. Die Theorie Nat ist aus Kapitel 3, Abbildung 3.10 hinreichend bekannt und beschreibt den Mengen-Typ der natürlichen Zahlen. Dieser Typ wurde unter anderem in Kapitel 4 zur Formalisierung von Ketten eingesetzt. Die beiden Typen nat und dnat werden in HOLCF vollkommen unabhängig voneinander verwendet. Die Theorie Dnat ist in Abbildung 6.9 dargestellt.

Die Konstanten dnat_rep und dnat_abs sind die bereits bekannten Konstanten für die Isomorphismen auf der initialen Lösung der Bereichsgleichung. Zur Vereinfachung wurde die Konstante dnat_copy als Abkürzung für den Term

```
Λf. dnat_abs oo (when[sinl][sinr oo f]) oo dnat_rep
```

eingeführt, der die expandierte Form von

Λf. dnat_abs oo FT f oo dnat_rep

darstellt. Die Namensgebung dnat_copy kommt nicht von ungefähr, denn das Funktional dnat_copy entspricht exakt dem *Kopier-Funktional (copy functional)* von [Pau87]. Bei Paulson wird das Kopier-Funktional jedoch nicht definiert, sondern seine Eigenschaften werden durch implizite Gleichungen axiomatisiert.

```
(* Introduce recursive type dnat:                              *)
(* domain equation: dnat = one ++ dnat                         *)
(* functor term   :   FT = <++,K_{one},I>                      *)

Dnat = HOLCF +

types   dnat 0
arities dnat :: pcpo

consts

dnat_rep        ::  dnat → (one ++ dnat)
dnat_abs        ::  (one ++ dnat) → dnat
dnat_copy       ::  (dnat → dnat) → dnat → dnat

dzero           ::  dnat
dsucc           ::  dnat → dnat
dnat_when       ::  β → (dnat → β) → dnat → β
is_dzero        ::  dnat → tr
is_dsucc        ::  dnat → tr
dpred           ::  dnat → dnat
dnat_take       ::  nat ⇒ dnat → dnat
dnat_bisim      ::  (dnat ⇒ dnat ⇒ bool) ⇒ bool

rules

dnat_abs_iso     dnat_rep[dnat_abs[x]] = x
dnat_rep_iso     dnat_abs[dnat_rep[x]] = x
dnat_copy_def    dnat_copy ≡ (Λf. dnat_abs oo
                                  (when[sinl][sinr oo f]) oo dnat_rep )
dnat_reach       (fix[dnat_copy])[x]=x

dzero_def        dzero ≡ dnat_abs[sinl[one]]
dsucc_def        dsucc ≡ (Λn. dnat_abs[sinr[n]])

dnat_when_def    dnat_when ≡ (Λf1 f2 n.when[Λx.f1][f2][dnat_rep[n]])
```

```
is_dzero_def     is_dzero ≡ dnat_when[TT][Λx.FF]
is_dsucc_def     is_dsucc ≡ dnat_when[FF][Λx.TT]
dpred_def        dpred ≡ dnat_when[⊥][Λx.x]

dnat_take_def    dnat_take ≡ (λn.iterate(n,dnat_copy,⊥))
dnat_bisim_def   dnat_bisim ≡
                 (λR.∀s1 s2.
                   R(s1,s2) →
                    ((s1=⊥ ∧ s2=⊥) ∨ (s1=dzero ∧ s2=dzero) ∨
                     (∃s11 s21. s11¬=⊥ ∧ s21¬=⊥ ∧ s1=dsucc[s11] ∧
                                 s2 = dsucc[s21] ∧ R(s11,s21))))

end
```

Abbildung 6.9: Theorie `Dnat`

Die Axiome `dnat_abs_iso`, `dnat_rep_iso` und `dnat_reach` charakterisieren den Typ `dnat` in der aus Kapitel 5 bekannten Weise als die initiale Lösung der Bereichsgleichung. Durch die Verwendung der Hilfskonstanten `dnat_copy` wird das Axiom der Wohlfundiertheit `dnat_reach` leichter lesbar.

Im Prinzip ist damit der Datentyp `dnat` hinreichend axiomatisiert. Damit der Typ aber vernünftig verwendet werden kann, müssen unbedingt weitere Konstanten eingeführt werden. Dies geschieht durch eine Reihe von konservativen Konstantendefinitionen.

Die Konstanten `dzero` und `dsucc` stehen für die Konstruktoren des Datentyps `dnat`, und die Konstante `dnat_when` steht für ein Diskriminatorfunktional auf dem Typ `dnat`. Die Definitionen für diese drei Konstanten stützen sich noch auf die Isomorphismen `dnat_abs` und `dnat_rep`. Die Diskriminatoren `is_dzero` und `is_dsucc` sowie der Destruktor `dpred` werden dagegen bereits über das Funktional `dnat_when` definiert.

Die beiden letzten Konstanten `dnat_take` und `dnat_bisim` dienen zur Herleitung von Induktions- und Co-Induktionsprinzipien für den Datentyp `dnat`. Die Definition des sogenannten *Take-Funktionals* `dnat_take` erfolgt für alle rekursiven Datentypen uniform. Das Take-Funktional entspricht immer einer Funktion, die das Kopier-Funktional des jeweiligen Typs `n`-mal beginnend mit $\bot$ iteriert und somit für jedes `n` eine Endo-Operation, einen sogenannten `n`-*Taker*, auf dem jeweiligen Typ liefert. Man beachte speziell den Typ der Funktion `dnat_take`. Das Verhalten der Funktion `dnat_take` läßt sich am besten aus den weiter unten in Abbildung 6.10 dargestellten Theoremen entnehmen. Das Take-Funktional `dnat_take` erzeugt für größer werdende Argumente `n` immer bessere bzw. gleichbleibend gute Approximationen für die Identitäts-Operation `ID` auf dem Typ, und das Supremum über alle diese Approximationen entspricht der Identität `ID`. Diese Tatsache kann aufgrund der Ausdrucksmächtigkeit der Logik HOLCF aus der Definition des Take-Funktionals und dem Fundierungsaxiom, hier `dnat_reach`, abgeleitet werden. Wie sich daraus ein Prinzip zur strukturellen Induktion ableiten läßt, werde ich ausführlich für den Datentyp der Listen in Abschnitt 6.6 vorführen.

Die Konstante dnat_bisim steht für ein Prädikat auf binären Relationen über dem Typ dnat und zeichnet gewisse Relationen als *Bisimulationen* aus. Bisimulationen auf einem Typ sind immer Ausschnitte der Identitäts-Relation = auf dem Typ. Sie sind symmetrisch und transitiv, jedoch im allgemeinen nicht reflexiv. Die spezielle Definition für Bisimulationen hängt von der jeweiligen Bereichsgleichung ab und kann aus dem Funktorterm abgeleitet werden. Mit Hilfe von Bisimulationen kann das Prinzip der Co-Induktion formuliert werden. Man beachte hierzu das Theorem dnat_coind in Abbildung 6.11. Eine ausführliche Diskussion der Co-Induktion für rekursive Bereiche findet sich in [Pit92]. Die Herleitung eines Co-Induktionsprinzips werde ich ausführlich für den Datentyp der Ströme in Abschnitt 6.5 vorführen.

6.4.2 Theoreme der Theorie Dnat

Die Theoreme der Theorie Dnat sind in den folgenden Abbildungen 6.10 und 6.11 dargestellt. Die meisten Theoreme der Abbildung 6.10 beschreiben Eigenschaften der neu definierten Konstanten, die sich unmittelbar aus den Definitionen ergeben. Daneben sind auch die Theoreme Exh_dnat und dnatE zur Fallunterscheidung, sowie diverse Theoreme zur Gleichheit bzw. Ungleichheit von Elementen aufgeführt. Sie bedürfen alle keiner besonderen Erläuterung.

```
dnat_iso_strict   dnat_rep[⊥] = ⊥ ∧ dnat_abs[⊥] = ⊥

dnat_copy         [ dnat_copy[f][⊥]=⊥ ,
                    dnat_copy[f][dzero]= dzero ,
                    n¬=⊥⟹dnat_copy[f][dsucc[n]] = dsucc[f[n]]  ]

Exh_dnat          n = ⊥ ∨ n = dzero ∨ (∃x . x¬=⊥ ∧ n = dsucc[x])

dnatE             ⟦  n=⊥⟹Q;
                     n=dzero⟹Q;
                     ⋀x.⟦ n=dsucc[x];x¬=⊥⟧⟹Q⟧⟹Q

dnat_when         [ dnat_when[c][f][⊥]=⊥ ,
                    dnat_when[c][f][dzero]=c ,
                    n¬=⊥⟹dnat_when[c][f][dsucc[n]]=f[n] ]

dnat_discsel      [ is_dzero[dzero] = TT ,
                    n ¬= ⊥⟹is_dzero[dsucc[n]] = FF ,
                    is_dsucc[dzero] = FF ,
                    n ¬= ⊥⟹is_dsucc[dsucc[n]] = TT ,
                    dpred[dzero] = ⊥ ,
                    n ¬= ⊥⟹dpred[dsucc[n]] = n ,
                    is_dzero[⊥] = ⊥ ,
                    is_dsucc[⊥] = ⊥ ,
                    dpred[⊥] = ⊥ ]


dnat_constrdef    [ dsucc[⊥] = ⊥ ,
```

```
                  dzero ¬= ⊥ ,
                  n ¬= ⊥⟹dsucc[n] ¬= ⊥ ]

dnat_dist_less    [ n ¬= ⊥⟹¬dsucc[n] ⊑ dzero ,
                  ¬dzero ⊑ dsucc[n] ]
dnat_dist_eq      [ dzero ¬= dsucc[n] ,
                  dsucc[n1] ¬= dzero ]
dnat_invert       [ ⟦ x1 ¬= ⊥; y1 ¬= ⊥; dsucc[x1] ⊑ dsucc[y1]⟧
                     ⟹x1 ⊑ y1 ]
dnat_inject       [ ⟦ x1 ¬= ⊥; y1 ¬= ⊥; dsucc[x1] = dsucc[y1]⟧
                     ⟹x1 = y1 ]
dnat_discsel_def  [ n ¬= ⊥⟹is_dzero[n] ¬= ⊥ ,
                  n ¬= ⊥⟹is_dsucc[n] ¬= ⊥ ]

dnat_take         [ dnat_take(Suc(n))[dsucc[xs]] = dsucc[dnat_take(n)[xs]] ,
                  dnat_take(Suc(n))[dzero] = dzero ,
                  dnat_take(0)[xs] = ⊥ ,
                  dnat_take(n)[⊥] = ⊥ ]
```

Abbildung 6.10: Theoreme der Theorie `Dnat` - Teil 1

Die Theoreme der Abbildung 6.11 behandeln bis auf das Theorem `dnat_flat` ausschließlich Eigenschaften, die mit Induktionsprinzipien auf dem Typ `dnat` zusammenhängen. Darunter seien vor allem die Theoreme `dnat_take_lemma`, `dnat_coind` und `dnat_ind` erwähnt, die verschiedene Induktionsprinzipien auf dem Typ `dnat` zum Ausdruck bringen. In den Abschnitten 6.5 und 6.6 werde ich genauer auf die Zusammenhänge der verschiedenen Prinzipien eingehen und daher an dieser Stelle auf weitere Kommentare verzichten.

```
dnat_take_lemma    (⋀n.dnat_take(n)[s1]=dnat_take(n)[s2])⟹s1=s2

dnat_coind_lemma   dnat_bisim(R)⟹
                     ∀p q.R(p,q) → dnat_take(n)[p]=dnat_take(n)[q]

dnat_coind         ⟦ dnat_bisim(R);R(p,q)⟧⟹p = q

dnat_finite_ind    ⟦ P(⊥);P(dzero);
                     ⋀s1.⟦ s1¬=⊥;P(s1)⟧⟹P(dsucc[s1])⟧
                     ⟹∀s.P(dnat_take(n)[s])

dnat_all_finite_lemma1  ∀s.dnat_take(n)[s]=⊥ ∨ dnat_take(n)[s]=s

dnat_all_finite_lemma2  ∃n.dnat_take(n)[s]=s
```

```
dnat_ind          [[ P(⊥);P(dzero);
                     ⋀s1.[[ s1¬=⊥;P(s1)]]⟹P(dsucc[s1])]]
                     ⟹P(s)

dnat_flat  flat(dzero)
```

Abbildung 6.11: Theoreme der Theorie Dnat - Teil 2

6.5 Der Datentyp der polymorphen Ströme stream

6.5.1 Die Theorie Stream

Die Theorie Stream führt den Datentyp der polymorphen Ströme 'α stream' als initiale Lösung der Bereichsgleichung

α stream = α ** (α stream)u

ein. Der zugehörige Funktorterm FT ist durch

$$FT = \langle **, K_\alpha, U \rangle$$

gegeben.

Die Theorie Stream ist in Abbildung 6.12 dargestellt. Analog zur Theorie Dnat bezeichnen die Konstanten stream_rep und stream_abs Konstanten für die Isomorphismen auf der initialen Lösung der Bereichsgleichung. Zur Vereinfachung wurde wieder die Konstante stream_copy als Abkürzung für den Term

Λf. stream_abs oo (ssplit[Λx y. x ## lift[up oo f][y]]) oo stream_rep

eingeführt, der die expandierte Form von

Λf. stream_abs oo FT f oo stream_rep

darstellt. Man vergleiche hierzu auch das Beispiel 5.5 aus Abschnitt 5.3. Die Axiomatisierung der initialen Lösung für die Bereichsgleichung erfolgt gemäß Abschnitt 5.3 durch die drei Axiome stream_abs_iso, stream_rep_iso und stream_reach.

```
(* Introduce recursive type: α stream                          *)
(* domain equation: α stream = α**(α stream)u                  *)
(* functor term   : FT = <**,K_{α},U>                          *)

Stream = Dnat +

types   stream 1
arities stream::(pcpo)pcpo

consts

stream_rep      ::  (α stream) → (α ** (α stream)u)
stream_abs      ::  (α ** (α stream)u) → (α stream)
stream_copy     ::  (α stream → α stream) → α stream → α stream

scons           ::  α → α stream → α stream
stream_when     ::  (α → α stream → β) → α stream → β
is_scons        ::  α stream → tr
shd             ::  α stream → α
stl             ::  α stream → α stream
stream_take     ::  nat ⇒ α stream → α stream
stream_finite   ::  α stream ⇒ bool
stream_bisim    ::  (α stream ⇒ α stream ⇒ bool) ⇒ bool

rules

stream_abs_iso   stream_rep[stream_abs[x]] = x
stream_rep_iso   stream_abs[stream_rep[x]] = x
stream_copy_def  stream_copy ≡ (Λf. stream_abs oo
                                (ssplit[Λx y. x ## lift[up oo f][y]])
                                 oo stream_rep)
stream_reach     (fix[stream_copy])[x]=x

scons_def        scons ≡ (Λx l. stream_abs[x##up[l]])
stream_when_def  stream_when ≡
                 (Λf l.ssplit[Λx l.f[x][lift[ID][l]]][stream_rep[l]])

is_scons_def     is_scons ≡ stream_when[Λx l.TT]
shd_def          shd ≡ stream_when[Λx l.x]
stl_def          stl ≡ stream_when[Λx l.l]

stream_take_def    stream_take ≡ (λn.iterate(n,stream_copy,⊥))
stream_finite_def  stream_finite ≡ (λs.∃n.stream_take(n)[s]=s)
```

```
stream_bisim_def   stream_bisim ≡ (λR.∀s1 s2.
                      R(s1,s2) →
                       ((s1=⊥ ∧ s2=⊥) ∨
                        (∃x s11 s21.
                           x¬=⊥ ∧ s1=scons[x][s11] ∧
                           s2 = scons[x][s21] ∧ R(s11,s21))))
end
```

Abbildung 6.12: Theorie Stream

Bei den restlichen Axiomen handelt es sich um Definitionen für diverse zusätzliche Konstanten. Die Theorie Stream ist diesbezüglich analog zur Theorie Dnat strukturiert, und so muß die Aufgabe der einzelnen Komponenten der Theorie nicht nochmals näher erläutert werden.

6.5.2 Theoreme der Theorie Stream

Die Theoreme der Theorie Stream sind in den folgenden Abbildungen 6.13 und 6.15 dargestellt. Die Theoreme der Abbildung 6.13 beschreiben im wesentlichen Eigenschaften der neu definierten Konstanten, die sich unmittelbar aus den Definitionen ergeben. Daneben sind auch die Theoreme Exh_stream und streamE zur Fallunterscheidung, sowie diverse Theoreme zur Gleichheit bzw. Ungleichheit von Elementen aufgeführt.

```
stream_iso_strict   stream_rep[⊥] = ⊥ ∧ stream_abs[⊥] = ⊥

stream_copy         [ stream_copy[f][⊥]=⊥ ,
                      x¬=⊥⟹stream_copy[f][scons[x][xs]]= scons[x][f[xs]]


Exh_stream          s = ⊥ ∨ (∃x xs. x¬=⊥ ∧ s = scons[x][xs])
streamE             ⟦ s=⊥⟹Q; ⋀x xs.⟦ s=scons[x][xs];x¬=⊥⟧⟹Q⟧⟹Q

stream_when         [ stream_when[f][⊥]=⊥ ,
                      x¬=⊥⟹stream_when[f][scons[x][xs]]= f[x][xs] ]


stream_discsel      [ x ¬= ⊥⟹is_scons[scons[x][xs]] = TT ,
                      x ¬= ⊥⟹shd[scons[x][xs]] = x ,
                      x ¬= ⊥⟹stl[scons[x][xs]] = xs ,
                      is_scons[⊥] = ⊥ ,
                      shd[⊥] = ⊥ ,
                      stl[⊥] = ⊥ ]

stream_constrdef    [ scons[⊥][xs] = ⊥ ,
                      x ¬= ⊥⟹scons[x][xs] ¬= ⊥ ]
```

```
stream_invert       [ ⟦ x1 ¬= ⊥; y1 ¬= ⊥;
                        scons[x1][x2] ⊑ scons[y1][y2]⟧
                        ==>x1 ⊑ y1 ∧ x2 ⊑ y2 ]

stream_inject       [ ⟦ x1 ¬= ⊥; y1 ¬= ⊥;
                         scons[x1][x2] = scons[y1][y2]⟧
                          ==>x1 = y1 ∧ x2 = y2 ]

stream_discsel_def  [ s ¬= ⊥==>is_scons[s] ¬= ⊥ ,
                      s ¬= ⊥==>shd[s] ¬= ⊥ ]

stream_take         [ x ¬= ⊥==>stream_take(Suc(n))[scons[x][xs]] =
                                  scons[x][stream_take(n)[xs]] ,
                      stream_take(n)[⊥] = ⊥ ,
                      stream_take(0)[xs] = ⊥ ]

stream_copy2        stream_copy[f][scons[x][xs]]= scons[x][f[xs]]
shd2                shd[scons[x][xs]]=x
stream_take2        stream_take(Suc(n))[scons[x][xs]]=
                    scons[x][stream_take(n)[xs]]
```

Abbildung 6.13: Theoreme der Theorie `Stream` - Teil 1

Aus den in Abbildung 6.13 dargestellten Theoremen wird eine Liste von Simplifikationsregeln `stream_rews` zusammengestellt. Bei Beweisen über den Datentyp der Ströme kann die aus Abschnitt 6.3 bekannte Simplifikatorstruktur `HOLCF_ss` um diese Regeln angereichert und mit dem Isabelle-Simplifikator (`simp_tac`) verwendet werden. Von dieser Möglichkeit wurde bei den Beweisen für Induktionsprinzipien in Abschnitt 6.5.3 häufig Gebrauch gemacht. Die Liste `stream_rews` der Strom-Simplifikationsregeln ist in Abbildung 6.14 dargestellt.

```
- val stream_rews =

  [ stream_rep[⊥] = ⊥ ,
    stream_abs[⊥] = ⊥ ,
    stream_copy[f][⊥] = ⊥ ,
    stream_copy[f][scons[x][xs]] = scons[x][f[xs]] ,
    stream_when[f][⊥] = ⊥ ,
    x ¬= ⊥==>stream_when[f][scons[x][xs]] = f[x][xs] ,
    x ¬= ⊥==>is_scons[scons[x][xs]] = TT ,
    shd[scons[x][xs]] = x ,
    x ¬= ⊥==>stl[scons[x][xs]] = xs ,
    is_scons[⊥] = ⊥ ,
    shd[⊥] = ⊥ ,
    stl[⊥] = ⊥ ,
    scons[⊥][xs] = ⊥ ,
```

```
x ¬= ⊥⟹scons[x][xs] ¬= ⊥ ,
s ¬= ⊥⟹is_scons[s] ¬= ⊥ ,
s ¬= ⊥⟹shd[s] ¬= ⊥ ,
stream_take(Suc(n))[scons[x][xs]] = scons[x][stream_take(n)[xs]] ,
stream_take(n)[⊥] = ⊥ ,
stream_take(0)[xs] = ⊥ ]
```

Abbildung 6.14: Simplifikationsregeln für Ströme

Die Theoreme der Abbildung 6.15 beschreiben ausschließlich Eigenschaften, die mit Induktionsprinzipien auf dem Typ `stream` zusammenhängen. Der Beweis für die Ableitung des Take-Lemmas `stream_take_lemma`, sowie die Herleitungen des Theorems `stream_coind_lemma` und des Co-Induktionsprinzips `stream_coind` werden in Abschnitt 6.5.3 ausführlich dargestellt.

Die Ableitung des Induktionsprinzips `stream_finite_ind` für endliche Ströme erfolgt analog zur Ableitung des Induktionsprinzips für endliche Listen `dlist_finite_ind`, welche ausführlich in Abschnitt 6.6.3 behandelt wird. Die Ableitung des Induktionsprinzips für beliebige Ströme aus dem Axiom der Wohlfundiertheit `stream_reach` ist Routine und entspricht der Vorgehensweise, die in [Pau84, Pau87] beschrieben wird.

```
stream_take_lemma    (⋀n.stream_take(n)[s1]=stream_take(n)[s2])⟹s1=s2
stream_reach2        lub(range(λi.stream_take(i)[s]))=s

stream_coind_lemma   stream_bisim(R)⟹
                      ∀p q.R(p,q) → stream_take(n)[p]=stream_take(n)[q]

stream_coind         ⟦ stream_bisim(R);R(p,q)⟧⟹p = q

stream_finite_ind    ⟦ P(⊥);
                       ⋀x s1.⟦ x¬=⊥;P(s1)⟧⟹P(scons[x][s1])⟧
                       ⟹∀s.P(stream_take(n)[s])

stream_finite_ind2   (⋀n.P(stream_take(n)[s]))
                       ⟹stream_finite(s) → P(s)

stream_finite_ind3   ⟦ P(⊥);
                       ⋀x s1.⟦ x¬=⊥;P(s1)⟧⟹P(scons[x][s1])⟧
                       ⟹stream_finite(s) → P(s)

stream_ind           ⟦ adm(P); P(⊥);
                       ⋀x s1.⟦ x¬=⊥;P(s1)⟧⟹P(scons[x][s1])⟧
                       ⟹P(s)
```

Abbildung 6.15: Theoreme der Theorie `Stream` - Teil 2

Neben den dargestellten Theoremen wurde noch eine Vielzahl anderer Theoreme für Ströme im Isabelle-System abgeleitet. Aus Platzgründen können diese jedoch in der vorliegenden Arbeit nicht dargestellt werden. Ich möchte diesbezüglich auf die Isabelle-Distribution verweisen, die neben der Theorie `Stream` noch eine weitere Theorie `Stream2` sowie die Theorie `Coind` mit einem Beispiel zur Co-Induktion enthält.

6.5.3 Ableitung von Induktionsprinzipien für Ströme

In diesem Abschnitt werde ich ausführlich die Beweise für die Theoreme `stream_take_lemma`, `stream_coind_lemma` und `stream_coind` darstellen. Die Beweise sind insofern interessant, als daß sie logische Argumentationen verwenden, die in entsprechender Form in der Logik LCF nicht möglich sind. Bevor ich jedoch die Beweise im einzelnen präsentiere, möchte ich einen kleinen Überblick über die Zusammenhänge geben. Abbildung 6.16 zeigt die Abhängigkeiten der einzelnen Theoreme.

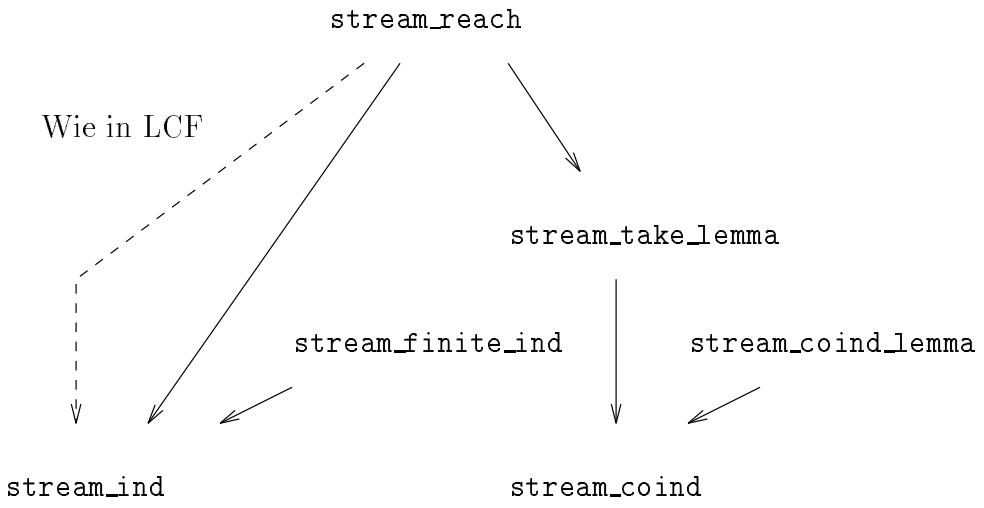


Abbildung 6.16: Zusammenhang der Induktionsprinzipien für Ströme

Zuerst wird aus dem Axiom der Wohlfundiertheit `stream_reach` das sogenannte Take-Lemma `stream_take_lemma` für Ströme abgeleitet. Der Beweis, der ausführlich in Abschnitt 6.5.3.1 behandelt wird, erfolgt nicht per Induktion, sondern verwendet die Definition des Fixpunktoperators `fix` und Argumentationen über kleinste obere Schranken. Diese Vorgehensweise ist in der Logik LCF nicht möglich und ist ein Beispiel für die Vorteile der Logik HOLCF gegenüber den herkömmlichen LCF-Systemen.

In LCF müßte der Beweis des Take-Lemmas per struktureller Strom-Induktion erfolgen, wozu aber vorher die Zulässigkeit der Induktionsformel gezeigt werden müßte. Aufgrund der Formelstruktur des Take-Lemmas

$$(\forall n.\texttt{stream\_take(n)[s1]=stream\_take(n)[s2]}) \Longrightarrow \texttt{s1=s2}$$

scheitern jedoch alle Versuche, die Zulässigkeit der Formel in den Stromargumenten `s1` oder `s2` zu zeigen, da diese auch 'negativ' vorkommen. Ein Nachweis der Zulässigkeit mit den bekannten syntaktischen Kriterien ist daher nicht möglich. Da das Take-Lemma in HOLCF anderweitig abgeleitet werden kann, ist die obige Formel für das Take-Lemma aus trivialen Gründen in allen Stromargumenten zulässig. Die Kenntnis über die Zulässigkeit der Formel ist jedoch, nachdem die universelle Gültigkeit bereits gezeigt werden konnte, nicht mehr von sonderlichem Interesse!

Aus dem Take-Lemma läßt sich unter anderem auch das Co-Induktionsprinzip für Ströme herleiten. Die wesentliche Beweisarbeit steckt dabei in der Herleitung des Hilfstheorems `stream_coind_lemma`, das per struktureller Induktion über den Mengen-Typ `nat` der natürlichen Zahlen bewiesen wird. Die abschließende Herleitung des Theorems `stream_coind` ist dann leicht auszuführen. Da in LCF ein zum Take-Lemma vergleichbares Theorem nicht abgeleitet werden kann, ist auch die Herleitung des Co-Induktionsprinzips in LCF nicht möglich. Eine eingehende Diskussion von Co-Induktionsprinzipien für rekursive Bereiche findet sich in [Pit92]. Dort wird die Gültigkeit von Co-Induktionsprinzipien für beliebige rekursive Bereiche mit kategoriellen Mitteln bewiesen. In HOLCF ist eine derartige allgemeine Herleitung nicht möglich, vielmehr müssen für jeden Datentyp die Induktions- bzw. Co-Induktionsprinzipien jeweils speziell abgeleitet werden. Die Ausdrucksmächtigkeit der Logik HOLCF ist zwar erheblich größer als die der Logik LCF, aufgrund des Fehlens von abhängigen Typen sind allerdings auch hier klare Grenzen gesetzt.

Weiterhin kann aus dem Axiom der Wohlfundiertheit `stream_reach` unter Verwendung der Fixpunktinduktion ein Prinzip zur strukturellen Induktion `stream_ind` abgeleitet werden [Mon85, Pau84, Pau87]. Im Bild ist dies durch die gestrichelte Linie angedeutet. Da der Konstruktor `scons` für Ströme eine nicht-strikte Funktion ist, wird in HOLCF im Fall der Ströme ebenfalls dieser Weg eingeschlagen. Aufgrund der höherstufigen Sprachmittel von HOLCF kann die strukturelle Induktion alternativ auch aus der Induktion für endliche Ströme `stream_finite_ind` und dem Fundierungsaxiom `stream_reach` abgeleitet werden. Hierzu muß dann die Definition der Zulässigkeit `adm_def` expandiert werden. Der Beweisaufwand bei dieser Alternative ist allerdings größer, als bei der LCF-Methode über die Fixpunktinduktion, und somit lohnt sich dieser Weg nicht. Für Datentypen mit ausschließlich strikten Konstruktoren, wie etwa beim Typ `dlist` der strikten Listen, wird die strukturelle Induktion dagegen nicht per Fixpunktinduktion hergeleitet. Für diesen speziellen Fall kann in HOLCF eine strukturelle Induktionsregel abgeleitet werden, die keine Einschränkung auf zulässige Prädikate enthält. Mehr zu diesem Thema findet sich in Abschnitt 6.6, wo die Ableitung der strukturellen Induktion für Listen aus dem Take-Lemma für Listen ausführlich dargestellt ist.

Das Theorem `stream_finite_ind` ist wie das Theorem `stream_coind_lemma` unabhängig vom Axiom der Wohlfundiertheit `stream_reach` und wird per struktureller Induktion über den Mengen-Typ `nat` der natürlichen Zahlen bewiesen. Für den Datentyp der Ströme spielt das Theorem `stream_finite_ind` eine untergeordnete Rolle, wohingegen im Fall des Datentyps der strikten Listen das entsprechende Theorem `dlist_finite_ind` ein wesentliches Argument zur Herleitung der bzgl. Zulässigkeit unbeschränkten strukturellen Induktion `dlist_ind` ist.

6.5.3.1 Beweis für das Theorem stream_take_lemma

Das Theorem stream_take_lemma zeigt, daß sich aus der Gleichheit aller n-Takes zweier Ströme s1 und s2 bereits die Gleichheit der potentiell unendlichen Ströme s1 und s2 ableiten läßt.

Für den Datentyp der Ströme sind n-Takes immer Präfixe der Länge n, und im speziellen Fall von Strömen über einem flachen Parameterbereich entspricht die Approximationsordnung $\sqsubseteq$ auf Strömen der sogenannten Präfixordnung auf Strömen [BDD+93]. Im allgemeinen Fall jedoch unterscheiden sich die Approximationsordnung und die Präfixordnung, und viele Theoreme für Ströme über flachen Bereichen verlieren ihre Gültigkeit für nicht-flache Parameterbereiche. Als Beispiel sei hier die Eigenschaft genannt, daß jeder Strom s1, der echt schwächer[1] als der Strom s2 ist, ein endlicher Präfix des Stroms s2 sein muß.

Wir beginnen den Beweis des Take-Lemmas für Ströme, indem wir uns das zu beweisende Theorem als Ziel vorgeben.

```
- val prems = goalw Stream.thy [stream_take_def]
"(⋀n.stream_take(n)[s1]=stream_take(n)[s2])⟹s1=s2";
```

Die Annahme verschwindet im impliziten Kontext des Beweises, und als erstes Teilziel erhalten wir:

```
 1. s1 = s2
```

Zunächst wenden wir die Eigenschaft der Wohlfundiertheit stream_reach

```
 fix[stream_copy][?x] = ?x
```

auf die beiden Ströme s1 und s2 an. In Verbindung mit Gleichungslogik erhalten wir:

```
- by (res_inst_tac [("t","s1")] (stream_reach RS subst) 1);
- by (res_inst_tac [("t","s2")] (stream_reach RS subst) 1);

 1. fix[stream_copy][s1] = fix[stream_copy][s2]
```

In dieser Situation versuchen wir nun nicht, per Fixpunktinduktion weiterzumachen, sondern expandieren die Definition des Fixpunktoperators fix, was in einer herkömmlichen LCF-Logik nicht möglich wäre.

```
- by (rtac (fix_def2 RS ssubst) 1);

 1. lub(range(λi. iterate(i, stream_copy, ⊥)))[s1] =
    lub(range(λi. iterate(i, stream_copy, ⊥)))[s2]
```

Als nächstes vertauschen wir auf beiden Seiten die Bildung des Supremums und die Operationsanwendung auf die Argumente s1 und s2. Dazu kombinieren wir die Regel contlub_cfun_fun

```
 is_chain(?FY)⟹lub(range(?FY))[?x] = lub(range(λi. ?FY(i)[?x]))
```

mit Gleichungslogik. Die dabei entstehenden Teilziele zum Nachweis von Ketteneigenschaften

[1] s1 ist genau dann 'echt schwächer' als s2, wenn s1 $\sqsubseteq$ s2 und s1$\neg$=s2.

lösen wir beide Male mittels Theorem is_chain_iterate
```
 is_chain(λi. iterate(i, ?F, ⊥))
```

Anwendung der nachstehenden Befehlsfolge liefert:
```
- by (rtac (contlub_cfun_fun RS ssubst) 1);
- by (rtac is_chain_iterate 1);
- by (rtac (contlub_cfun_fun RS ssubst) 1);
- by (rtac is_chain_iterate 1);

 1. lub(range(λi. iterate(i, stream_copy, ⊥)[s1])) =
    lub(range(λi. iterate(i, stream_copy, ⊥)[s2]))
```

Die Gleichheit der beiden Suprema zeigen wir, indem wir die Regel lub_equal
```
  ⟦ is_chain(?C1); is_chain(?C2); ∀k. ?C1(k) = ?C2(k)⟧
      ⟹lub(range(?C1)) = lub(range(?C2))

- by (rtac lub_equal 1);
```

anwenden. Dies führt zu drei neuen Teilzielen:
```
 1. is_chain(λi. iterate(i, stream_copy, ⊥)[s1])
 2. is_chain(λi. iterate(i, stream_copy, ⊥)[s2])
 3. ∀k. iterate(k, stream_copy, ⊥)[s1] = iterate(k, stream_copy, ⊥)[s2]
```

Die beiden ersten Teilziele erfordern den Nachweis von Ketteneigenschaften. Diesen erbringen wir jeweils durch Anwendung der Regel
```
  is_chain(λi. iterate(i, ?F2, ⊥)[?x])

- by (rtac (is_chain_iterate RS ch2ch_fappL) 1);
- by (rtac (is_chain_iterate RS ch2ch_fappL) 1);
```

Es verbleibt nun noch das Teilziel:
```
 1. ∀k. iterate(k, stream_copy, ⊥)[s1] = iterate(k, stream_copy, ⊥)[s2]
```

Wenn wir im Teilziel die Definition des Take-Funktionals stream_take falten, so entspricht die zu beweisende Aussage bis auf den expliziten Quantor gerade der Annahme im impliziten Kontext des Beweises. Diese Faltung ist aber nicht nötig, da wir beim Aufsetzen des ursprünglichen Beweisziels mittels
```
- val prems = goalw Stream.thy [stream_take_def]
```

bereits die Entfaltung der Definition des Take-Funktionals stream_take veranlaßt haben. Somit liegt die globale Annahme
```
⋀n.stream_take(n)[s1]=stream_take(n)[s2]
```

bereits in der aufgefalteten Form
```
⋀n.iterate(n, stream_copy, ⊥)[s1] = iterate(n, stream_copy, ⊥)[s2]
```

vor. Folgende Befehlssequenz ermöglicht daher den Abschluß des Beweises:

```
- by (rtac allI 1);
- by (resolve_tac prems 1);
No subgoals
```

Man sieht sofort ein, daß sich ein entsprechendes Theorem ableiten läßt, wenn statt der Identitätsrelation = die Approximationsordnung $\sqsubseteq$ verwendet wird.

Take-Lemmata lassen sich für alle Datentypen, die als initiale Lösungen von Bereichsgleichungen im Stil von Kapitel 5 axiomatisiert sind, analog zum obigen Beweis zeigen. Im Fall von Datentypen mit ausschließlich strikten Konstruktoren, wie etwa natürliche Zahlen `dnat` oder Listen `dlist`, muß die Aussage des jeweiligen Take-Lemmas allerdings mit Vorsicht interpretiert werden. Wegen der Striktheit der Konstruktoren kollabiert hier ein `n`-Take entweder zu $\bot$ oder entspricht bereits dem ganzen Element. Im allgemeinen ist ein `n`-Take also keine Reproduktion eines Elements 'bis zur Tiefe `n`'.

6.5.3.2 Beweis für das Theorem `stream_coind_lemma`

Wir beginnen den Beweis, indem wir uns das Theorem als initiales Beweisziel vorgeben und dabei gleich die Definition für Bisimulationen entfalten.

```
- val prems = goalw Stream.thy [stream_bisim_def]
"stream_bisim(R)⟹∀p q.R(p,q) → stream_take(n)[p]=stream_take(n)[q]";
- by (cut_facts_tac prems 1);
```

Dies führt zum globalen Beweisziel:

```
 1. ∀s1 s2.
       R(s1, s2) →
       s1 = ⊥ ∧ s2 = ⊥ ∨
       (∃x s11 s21.
           x ¬= ⊥ ∧ s1 = scons[x][s11] ∧ s2 = scons[x][s21] ∧ R(s11, s21))⟹
    ∀p q. R(p, q) → stream_take(n)[p] = stream_take(n)[q]
```

Um die Aussage zu beweisen, verwenden wir Induktion über die Variable `n` vom Typ `nat`.

```
- by (nat_ind_tac "n" 1);
```

```
 1. ∀s1 s2.
       R(s1, s2) →
       s1 = ⊥ ∧ s2 = ⊥ ∨
       (∃x s11 s21.
           x ¬= ⊥ ∧ s1 = scons[x][s11] ∧ s2 = scons[x][s21] ∧ R(s11, s21))⟹
    ∀p q. R(p, q) → stream_take(0)[p] = stream_take(0)[q]
```

```
2. ⋀n1. ⟦ ∀s1 s2.
              R(s1, s2) →
              s1 = ⊥ ∧ s2 = ⊥ ∨
              (∃x s11 s21.
                  x ¬= ⊥ ∧
                  s1 = scons[x][s11] ∧ s2 = scons[x][s21] ∧ R(s11, s21));
          ∀p q. R(p, q) → stream_take(n1)[p] = stream_take(n1)[q]⟧⟹
       ∀p q. R(p, q) → stream_take(Suc(n1))[p] = stream_take(Suc(n1))[q]
```

Das erste Teilziel lösen wir automatisch durch Anwendung des Simplifikators, den wir um die Simplifikationsregeln für Ströme anreichern. Im zweiten Teilziel gehen wir gemäß der Methodik des natürlichen Schließens vor, d.h. wir entfernen die Quantoren und übernehmen die Prämisse `R(p,q)` in den Annahmenkontext.

```
- by (simp_tac (HOLCF_ss addsimps stream_rews) 1);
- by (strip_tac 1);
```

```
1. ⋀n1 p q.
      ⟦ ∀s1 s2.
            R(s1, s2) →
            s1 = ⊥ ∧ s2 = ⊥ ∨
            (∃x s11 s21.
                x ¬= ⊥ ∧
                s1 = scons[x][s11] ∧ s2 = scons[x][s21] ∧ R(s11, s21));
         ∀p q. R(p, q) → stream_take(n1)[p] = stream_take(n1)[q];
         R(p, q)⟧⟹
      stream_take(Suc(n1))[p] = stream_take(Suc(n1))[q]
```

Als nächstes spezialisieren wir die Bisimulationseigenschaft von `R` für `p` und `q` und nützen die Annahme `R(p,q)` mittels Modus-Ponens aus. Um den Prozeß der Regel-Unifikation des Systems zu steuern, verketten wir dazu die einzelnen Beweisschritte durch das 'Tactical' `THEN` und führen gleich noch eine Elimination der Disjunktion in der nun spezialisierten Prämisse durch.

```
- by ((etac allE 1) THEN (etac allE 1) THEN (etac (mp RS disjE) 1));
- by (atac 1);
```

Wir erhalten zwei Fälle:

```
1. ⋀n1 p q.
      ⟦ ∀p q. R(p, q) → stream_take(n1)[p] = stream_take(n1)[q]; R(p, q);
         p = ⊥ ∧ q = ⊥⟧⟹
      stream_take(Suc(n1))[p] = stream_take(Suc(n1))[q]
2. ⋀n1 p q.
      ⟦ ∀p q. R(p, q) → stream_take(n1)[p] = stream_take(n1)[q]; R(p, q);
         ∃x s11 s21.
            x ¬= ⊥ ∧
            p = scons[x][s11] ∧ q = scons[x][s21] ∧ R(s11, s21)⟧⟹
      stream_take(Suc(n1))[p] = stream_take(Suc(n1))[q]
```

Das erste Teilziel lösen wir automatisch durch Simplifikation mittels der zusätzlichen Regeln in der Theorem-Liste `stream_rews` und Verwendung des Annahmenkontextes (`asm_simp_tac` statt `simp_tac`).

```
- by (asm_simp_tac (HOLCF_ss addsimps stream_rews) 1);
```

Im zweiten Ziel eliminieren wir zuerst die Existenzquantoren im Annahmenkontext.

```
- by (etac exE 1);
- by (etac exE 1);
- by (etac exE 1);
```

Dies führt zu folgendem Teilziel:

```
 1. ⋀n1 p q x s11 s21.
       ⟦ ∀p q. R(p, q) → stream_take(n1)[p] = stream_take(n1)[q]; R(p, q);
          x ¬= ⊥ ∧ p = scons[x][s11] ∧ q = scons[x][s21] ∧ R(s11, s21)⟧⟹
       stream_take(Suc(n1))[p] = stream_take(Suc(n1))[q]
```

Unter Verwendung des Annahmenkontextes kann durch Simplifikation mittels der Eigenschaften von `stream_take` folgende Vereinfachung automatisch erzielt werden:

```
- by (asm_simp_tac (HOLCF_ss addsimps stream_rews) 1);

 1. ⋀n1 p q x s11 s21.
       ⟦ ∀p q. R(p, q) → stream_take(n1)[p] = stream_take(n1)[q]; R(p, q);
          x ¬= ⊥ ∧ p = scons[x][s11] ∧ q = scons[x][s21] ∧ R(s11, s21)⟧⟹
       scons[x][stream_take(n1)[s11]] = scons[x][stream_take(n1)[s21]]
```

Der nächste Schritt im Beweis verwendet die Kongruenzregel

```
 ?x = ?y⟹?f[?x] = ?f[?y]

- by (rtac cfun_arg_cong 1);
```

und wir erhalten:

```
 1. ⋀n1 p q x s11 s21.
       ⟦ ∀p q. R(p, q) → stream_take(n1)[p] = stream_take(n1)[q]; R(p, q);
          x ¬= ⊥ ∧ p = scons[x][s11] ∧ q = scons[x][s21] ∧ R(s11, s21)⟧⟹
       stream_take(n1)[s11] = stream_take(n1)[s21]
```

Eine Spezialisierung der Induktionsannahme für `s11` und `s21` unter Verwendung der Annahme `R(s11, s21)` erlaubt dann den Abschluß des Beweises. Obwohl es sich hierbei um einen durchaus komplexen Beweisschritt handelt, wird er automatisch vom klassischen Beweiser des Isabelle-Systems (`fast_tac`) durchgeführt.

```
- by (fast_tac HOL_cs 1);
No subgoals
```

6.5.3.3 Beweis für das Theorem stream_coind

Die Herleitung des Co-Induktionsprinzips ist nahezu trivial. Die eigentliche Beweisarbeit steckt in den Herleitungen für die Theoreme stream_take_lemma und stream_coind_lemma.

Wir beginnen den Beweis wie üblich mit:

```
- val prems = goal Stream.thy "⟦ stream_bisim(R);R(p,q)⟧⟹p = q";
```

und erhalten als erstes Teilziel:

```
1. p = q
```

Die Annahmen stream_bisim(R) und R(p,q) verschwinden im impliziten Kontext des Beweises, den wir im Gegensatz zu den meisten anderen Beweisen diesmal nicht durch Anwendung der Taktik cut_facts_tac sichtbar machen.

Als nächstes wenden wir das Take-Lemma stream_take_lemma

```
 (⋀n. stream_take(n)[?s1] = stream_take(n)[?s2])⟹?s1 = ?s2

- by (rtac stream_take_lemma 1);
```

an und erhalten:

```
 1. ⋀n. stream_take(n)[p] = stream_take(n)[q]
```

Spezialisierung und Verwendung des Modus-Ponens erlaubt die Anwendung des Theorems stream_coind_lemma

```
 stream_bisim(?R)⟹
   ∀p q. ?R(p, q) → stream_take(?n)[p] = stream_take(?n)[q]
```

womit unter Ausnutzung des impliziten Annahmenkontextes der Beweis trivial abgeschlossen wird.

```
- by (rtac (stream_coind_lemma RS spec RS spec RS mp) 1);
- by (resolve_tac prems 1);
- by (resolve_tac prems 1);
No subgoals
```

6.6 Der Datentyp der polymorphen Listen dlist

In diesem Abschnitt wird die Theorie der polymorphen strikten Listen eingeführt, die als Beispiel für einen polymorphen rekursiven Datentyp mit ausschließlich strikten Konstruktoren dient. Für derartige Datentypen kann in der Logik HOLCF ein strukturelles Induktionsprinzip abgeleitet werden, das keine Beschränkungen bzgl. der Zulässigkeit des Induktionsprädikats enthält. In dieser Hinsicht bietet die Logik HOLCF also deutliche Vorteile gegenüber der herkömmlichen LCF-Logik, in der die uneingeschränkte Induktion nur für flache Parametertypen hergeleitet werden kann. Diese Tatsache ist der wesentliche Grund für die Darstellung der Listentheorie in dieser Arbeit.

6.6.1 Die Theorie Dlist

Die Theorie Dlist enthält die Formalisierung des Datentyps dlist der strikten Listen. Ähnlich wie bei der Theorie Dnat rührt das Voranstellen des Buchstabens d daher, daß in der zugrundeliegenden Logik HOL bereits ein (Mengen-)Typ der Listen list in der Theorie List formalisiert wurde. Um Namenskonflikte zu verhindern, mußten also auch hier Ersatznamen verwendet werden.

Der Datentyp dlist wird als initiale Lösung der Bereichsgleichung

$$\alpha \text{ dlist} = \text{one} \mathbin{++} (\alpha \mathbin{**} \alpha \text{ dlist})$$

formalisiert. Der zugehörige Funktorterm FT ist durch

$$FT = \langle ++, K_{\text{one}}, \langle **, K_\alpha, I\rangle\rangle$$

gegeben. Die Theorie Dlist ist in Abbildung 6.17 dargestellt und weist einen zu den Theorien Dnat und Stream analogen Aufbau auf.

```
(* Introduce recursive type α dlist                          *)
(* domain equation:  α dlist = one ++ (α ** α dlist)         *)
(* functor term   :  FT = <++,K_{one},<**,K_{α},I>>           *)

Dlist = Stream +

types   dlist 1
arities dlist::(pcpo)pcpo

consts

dlist_rep       ::  (α dlist) → (one ++ α ** α dlist)
dlist_abs       ::  (one ++ α ** α dlist) → (α dlist)
dlist_copy      ::  (α dlist → α dlist) → α dlist → α dlist

dnil            ::  α dlist
dcons           ::  α → α dlist → α dlist
dlist_when      ::  β → (α → α dlist → β) → α dlist → β
is_dnil         ::  α dlist → tr
is_dcons        ::  α dlist → tr
dhd             ::  α dlist → α
dtl             ::  α dlist → α dlist
dlist_take      ::  nat ⇒ α dlist → α dlist
dlist_finite    ::  α dlist ⇒ bool
dlist_bisim     ::  (α dlist ⇒ α dlist ⇒ bool) ⇒ bool

rules
```

```
dlist_abs_iso     dlist_rep[dlist_abs[x]] = x
dlist_rep_iso     dlist_abs[dlist_rep[x]] = x
dlist_copy_def    dlist_copy ≡ (Λf. dlist_abs oo
                          (when[sinl][sinr oo (ssplit[Λx y. x ## f[y]])])
                                    oo dlist_rep)
dlist_reach       (fix[dlist_copy])[x]=x

dnil_def          dnil ≡ dlist_abs[sinl[one]]
dcons_def         dcons ≡ (Λx l. dlist_abs[sinr[x##l]])

dlist_when_def    dlist_when ≡ (Λf1 f2 l.
                     when[Λx.f1][ssplit[Λx l.f2[x][l]]][dlist_rep[l]])

is_dnil_def       is_dnil ≡ dlist_when[TT][Λx l.FF]
is_dcons_def      is_dcons ≡ dlist_when[FF][Λx l.TT]
dhd_def           dhd ≡ dlist_when[⊥][Λx l.x]
dtl_def           dtl ≡ dlist_when[⊥][Λx l.l]

dlist_take_def    dlist_take ≡ (λn.iterate(n,dlist_copy,⊥))

dlist_finite_def  dlist_finite ≡ (λs.∃n.dlist_take(n)[s]=s)

dlist_bisim_def   dlist_bisim ≡ (λR.∀l1 l2.
                   R(l1,l2) →
                      ((l1=⊥ ∧ l2=⊥) ∨
                       (l1=dnil ∧ l2=dnil) ∨
                       (∃x l11 l21. x¬=⊥ ∧ l11¬=⊥ ∧ l21¬=⊥ ∧
                                    l1=dcons[x][l11] ∧
                                    l2 = dcons[x][l21] ∧ R(l11,l21))))
end
```

Abbildung 6.17: Theorie Dlist

Die Konstanten dlist_rep und dlist_abs sind die Konstanten für den Isomorphismus auf der initialen Lösung der Bereichsgleichung, und die Konstante dlist_copy wird als Abkürzung für den Term

```
Λf. dlist_abs oo
      (when[sinl][sinr oo (ssplit[Λx y. x ## f[y]])]) oo dlist_rep
```

eingeführt, der die expandierte Form von

Λf. dlist_abs oo *FT* f oo dlist_rep

darstellt. Die Axiomatisierung der initialen Lösung für die Bereichsgleichung erfolgt gemäß Abschnitt 5.3 durch die drei Axiome dlist_abs_iso, dlist_rep_iso und dlist_reach. Wie

schon bei den vorangegangenen Theorien werden neben dieser minimalen Formalisierung des Typs noch eine Reihe von nützlichen Konstanten durch konservative Definitionen eingeführt.

6.6.2 Theoreme der Theorie `Dlist`

Die Theoreme der Theorie `Dlist` sind zum besseren Überblick getrennt in den Abbildungen 6.18 und 6.20 dargestellt.

In Abbildung 6.18 sind Theoreme dargestellt, die sich zumeist unmittelbar aus der Definition der Konstanten ergeben. Daneben sind aber auch die Theoreme `Exh_dlist` und `dlistE` zur Fallunterscheidung enthalten, sowie diverse Theoreme zur Gleichheit bzw. Ungleichheit von Elementen aufgeführt.

```
dlist_iso_strict  dlist_rep[⊥] = ⊥ ∧ dlist_abs[⊥] = ⊥

dlist_copy
                  [ xl¬= ⊥⟹dlist_copy[f][dcons[x][xl]] = dcons[x][f[xl]] ,
                    dlist_copy[f][dnil] = dnil ,
                    dlist_copy[f][⊥] = ⊥ ]

Exh_dlist         l = ⊥ ∨ l = dnil ∨ (∃x xl. x¬=⊥ ∧ xl¬=⊥ ∧ l=dcons[x][xl])
dlistE            ⟦ l=⊥⟹Q; l=dnil⟹Q;
                    ⋀x xl.⟦ l=dcons[x][xl];x¬=⊥;xl¬=⊥⟧⟹Q⟧⟹Q

dlist_when        [ ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹
                       dlist_when[f1][f2][dcons[x][xl]] = f2[x][xl] ,
                    dlist_when[f1][f2][dnil] = f1 ,
                    dlist_when[f1][f2][⊥] = ⊥ ]

dlist_discsel     [ is_dnil[dnil] = TT ,
                    ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹is_dnil[dcons[x][xl]] = FF ,
                    is_dcons[dnil] = FF ,
                    ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹is_dcons[dcons[x][xl]] = TT ,
                    dhd[dnil] = ⊥ ,
                    ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹dhd[dcons[x][xl]] = x ,
                    dtl[dnil] = ⊥ ,
                    ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹dtl[dcons[x][xl]] = xl ,
                    is_dnil[⊥] = ⊥ ,
                    is_dcons[⊥] = ⊥ ,
                    dhd[⊥] = ⊥ ,
                    dtl[⊥] = ⊥ ]

dlist_constrdef   [ dcons[⊥][xl] = ⊥ ,
                    dcons[x][⊥] = ⊥ ,
                    dnil ¬= ⊥ ,
                    ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹dcons[x][xl] ¬= ⊥ ]
```

```
dlist_dist_less   [ ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹¬dcons[x][xl] ⊑ dnil ,
                    ¬dnil ⊑ dcons[x][xl] ]

dlist_dist_eq     [ dnil ¬= dcons[x][xl] ,
                    dcons[x1][xl1] ¬= dnil ]

dlist_invert      [ ⟦ x1 ¬= ⊥; y1 ¬= ⊥; x2 ¬= ⊥; y2 ¬= ⊥;
                      dcons[x1][x2] ⊑ dcons[y1][y2]⟧
                      ⟹x1 ⊑ y1 ∧ x2 ⊑ y2 ]

dlist_inject      [ ⟦ x1 ¬= ⊥; y1 ¬= ⊥; x2 ¬= ⊥; y2 ¬= ⊥;
                      dcons[x1][x2] = dcons[y1][y2]⟧
                      ⟹x1 = y1 ∧ x2 = y2 ]

dlist_discsel_def [ l ¬= ⊥⟹is_dnil[l] ¬= ⊥ ,
                    l ¬= ⊥⟹is_dcons[l] ¬= ⊥ ]

dhd2            xl¬=⊥⟹dhd[dcons[x][xl]]=x

dtl2            x¬=⊥⟹dtl[dcons[x][xl]]=xl

dlist_take      [ dlist_take(Suc(n))[dcons[x][xl]]=dcons[x][dlist_take(n)[xl]],
                  dlist_take(Suc(n))[dnil] = dnil ,
                  dlist_take(0)[xs] = ⊥ ,
                  dlist_take(n)[⊥] = ⊥ ]
```

Abbildung 6.18: Theoreme der Theorie Dlist - Teil 1

In Abbildung 6.19 ist eine Liste von Simplifikationsregeln dargestellt, um die der Termsimplifikator von Isabelle bei Bedarf angereichert werden kann. Die abgebildete Liste von Simplifikationsregeln hat sich in den von mir geführten Beweisen als sinnvoll erwiesen.

```
val dlist_rews =
  [ dlist_take(Suc(n))[dcons[x][xl]] = dcons[x][dlist_take(n)[xl]] ,
    dlist_take(Suc(n))[dnil] = dnil ,
    dlist_take(0)[xs] = ⊥ ,
    dlist_take(n)[⊥] = ⊥ ,
    xl ¬= ⊥⟹dhd[dcons[x][xl]] = x ,
    x ¬= ⊥⟹dtl[dcons[x][xl]] = xl ,
    l ¬= ⊥⟹is_dnil[l] ¬= ⊥ ,
    l ¬= ⊥⟹is_dcons[l] ¬= ⊥ ,
    ⟦ x ¬= ⊥; xl ¬= ⊥⟧⟹¬dcons[x][xl] ⊑ dnil ,
    ¬dnil ⊑ dcons[x][xl] ,
    dnil ¬= dcons[x][xl] ,
```

```
dcons[x1][xl1] ¬= dnil ,
dcons[⊥][xl] = ⊥ ,
dcons[x][⊥] = ⊥ ,
dnil ¬= ⊥ ,
[| x ¬= ⊥; xl ¬= ⊥|]==>dcons[x][xl] ¬= ⊥ ,
is_dnil[dnil] = TT ,
[| x ¬= ⊥; xl ¬= ⊥|]==>is_dnil[dcons[x][xl]] = FF ,
is_dcons[dnil] = FF ,
[| x ¬= ⊥; xl ¬= ⊥|]==>is_dcons[dcons[x][xl]] = TT ,
dhd[dnil] = ⊥ ,
[| x ¬= ⊥; xl ¬= ⊥|]==>dhd[dcons[x][xl]] = x ,
dtl[dnil] = ⊥ ,
[| x ¬= ⊥; xl ¬= ⊥|]==>dtl[dcons[x][xl]] = xl ,
is_dnil[⊥] = ⊥ ,
is_dcons[⊥] = ⊥ ,
dhd[⊥] = ⊥ ,
dtl[⊥] = ⊥ ,
[| x¬=⊥;xl¬=⊥|]==>dlist_when[f1][f2][dcons[x][xl]] = f2[x][xl] ,
dlist_when[f1][f2][dnil] = f1 ,
dlist_when[f1][f2][⊥] = ⊥ ,
xl ¬= ⊥==>dlist_copy[f][dcons[x][xl]] = dcons[x][f[xl]] ,
dlist_copy[f][dnil] = dnil ,
dlist_copy[f][⊥] = ⊥ ,
dlist_rep[⊥] = ⊥ ,
dlist_abs[⊥] = ⊥ ]
```

Abbildung 6.19: Simplifikationsregeln für Listen

Die Theoreme der Abbildung 6.20 beschreiben ausschließlich Eigenschaften, die mit Induktionsprinzipien auf dem Typ `dlist` zusammenhängen. Der Beweis für die Ableitung des strukturellen Induktionsprinzips `dlist_ind` sowie die Beweise für die dazu benötigten Hilfstheoreme `dlist_finite_ind`, `dlist_all_finite_lemma1` und `dlist_all_finite_lemma2` werden in Abschnitt 6.6.3 ausführlich dargestellt.

Die Ableitung des Take-Lemmas `dlist_take_lemma` sowie die Herleitungen des Theorems `dlist_coind_lemma` und des Co-Induktionsprinzips `dlist_coind` erfolgen analog zu den Herleitungen der entsprechenden Theoreme in der Theorie `Stream`. Diese wurden ausführlich in Abschnitt 6.5.3 dargestellt.

```
dlist_take_lemma    (⋀n.dlist_take(n)[l1]=dlist_take(n)[l2])⟹l1=l2

dlist_coind_lemma   dlist_bisim(R)⟹
                     ∀p q.R(p,q) → dlist_take(n)[p]=dlist_take(n)[q]

dlist_coind         ⟦ dlist_bisim(R);R(p,q)⟧⟹p = q

dlist_finite_ind    ⟦ P(⊥);P(dnil);
                      ⋀x l1.⟦ x¬=⊥;l1¬=⊥;P(l1)⟧⟹P(dcons[x][l1])⟧
                      ⟹∀l.P(dlist_take(n)[l])

dlist_all_finite_lemma1  ∀l.dlist_take(n)[l]=⊥ ∨ dlist_take(n)[l]=l
dlist_all_finite_lemma2  ∃n.dlist_take(n)[l]=l
dlist_all_finite          dlist_finite(l)

dlist_ind           ⟦ P(⊥);P(dnil);
                      ⋀x l1.⟦ x¬=⊥;l1¬=⊥;P(l1)⟧⟹P(dcons[x][l1])⟧
                      ⟹P(l)
```

Abbildung 6.20: Theoreme der Theorie Dlist - Teil 2

6.6.3 Ableitung von Induktionsprinzipien für Listen

In diesem Abschnitt werde ich ausführlich die Beweise für die Theoreme dlist_finite_ind, dlist_all_finite_lemma1, dlist_all_finite_lemma2 und dlist_ind darstellen. Das Theorem dlist_ind beschreibt das Prinzip der strukturellen Induktion für den Datentyp der Listen, und die anderen drei Theoreme sind Hilfstheoreme, die zur Ableitung von dlist_ind benötigt werden. Die Herleitung der Theoreme wird ausführlich dargestellt, weil sich hier erneut der Vorteil der Logik HOLCF gegenüber herkömmlichen LCF-artigen Logiken zeigt. Die Abhängigkeiten der einzelnen Theoreme sind in der Abbildung 6.21 dargestellt.

Wie beim Datentyp der Ströme werden aus dem Axiom der Wohlfundiertheit dlist_reach das Take-Lemma dlist_take_lemma und darauf aufbauend das Co-Induktionsprinzip dlist_coind abgeleitet. Die Beweise erfolgen analog zu den entsprechenden Beweisen für den Datentyp der Ströme und werden daher nicht nochmals vorgeführt.

Im Fall der Listen sind alle Konstruktoren strikt, und daher kann das Theorem dlist_all_finite_lemma1 hergeleitet werden, welches zeigt, daß für beliebiges n die n-Takes einer Liste entweder undefiniert sind oder bereits die gesamte Liste enthalten. Eine derartige Aussage kann nur für Datentypen hergeleitet werden, die ausschließlich strikte Konstruktoren haben.

Mit Hilfe des Take-Lemmas wird aus dem Theorem dlist_all_finite_lemma1 das Theorem dlist_all_finite_lemma2 abgeleitet, welches zeigt, daß es für jede Liste einen n-Take gibt, der die Liste bereits vollständig enthält, alle Listen sind also endlich. Das Theorem dlist_finite_ind erlaubt aber Induktion für alle endlichen Listen, und damit folgt das Induktionsprinzip dlist_ind für beliebige Listen trivial aus dlist_finite_ind und

dlist_all_finite_lemma2. Da das Induktionsprinzip dlist_finite_ind durch strukturelle Induktion über den Mengen-Typ nat hergeleitet wird und daher keine Einschränkung bzgl. der Zulässigkeit des Induktionsprädikats anfällt, ist auch die Induktionsregel dlist_ind nicht durch Zulässigkeitsforderungen beschränkt.

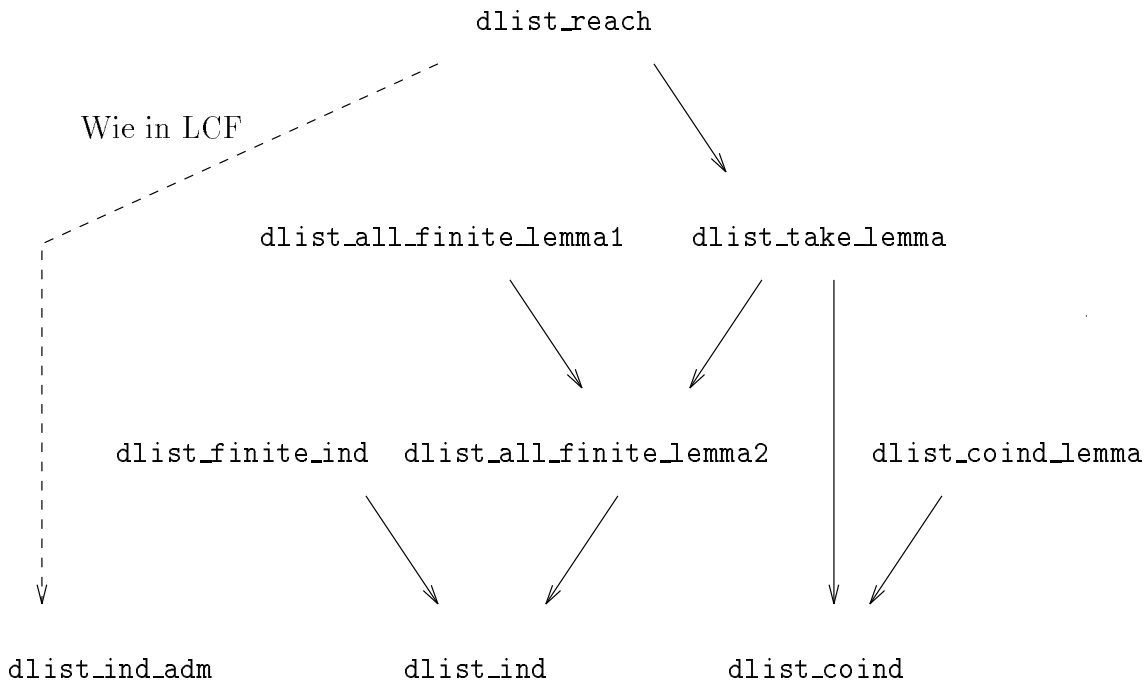


Abbildung 6.21: Zusammenhang der Induktionsprinzipien für Listen

Ähnlich wie beim Datentyp der Ströme kann aus dem Axiom der Wohlfundiertheit dlist_reach mit Hilfe der Fixpunktinduktion direkt eine Regel zur strukturellen Induktion dlist_ind_adm abgeleitet werden. Diese Möglichkeit ist durch die gestrichelte Linie gekennzeichnet. Die Regel dlist_ind_adm sieht wie folgt aus:

```
⟦ adm(P);
  P(⊥);P(dnil);
  ⋀x l1.⟦ x¬=⊥;l1¬=⊥;P(l1)⟧⟹P(dcons[x][l1])⟧
  ⟹P(l)
```

Wegen der Verwendung der Fixpunktinduktionsregel im Beweis enthält die resultierende Regel im Gegensatz zur Regel dlist_ind die zusätzliche Prämisse adm(P), welche den Nachweis der Zulässigkeit des Induktionsprädikats P erfordert. Nur wenn der Parametertyp, über dem die Listen gebildet werden, kettenendlich bzw. sogar flach ist, kann die Prämisse adm(P) eliminiert werden. Im allgemeinen Fall eines nicht kettenendlichen Parametertyps gibt es aber trotz der Striktheit der Listen echt unendliche Ketten von Listen und somit auch Prädikate, die nicht zulässig sind. Die Prämisse adm(P) kann im allgemeinen also nicht eliminiert werden, und daher kann man das Theorem dlist_ind auch nicht aus dlist_ind_adm herleiten.

Das Beispiel zeigt, daß im Fall von Datentypen mit ausschließlich strikten Konstruktoren die übliche Ableitung eines strukturellen Induktionsprinzips nach der LCF-Methode zu einer unnötig schwachen Induktionsregel führt. Die stärkere Induktionsregel, die keine Beschränkung durch Zulässigkeitsannahmen enthält, kann allerdings nur in der Logik HOLCF abgeleitet werden, da die Herleitung auf dem Take-Lemma basiert.

6.6.3.1 Beweis für das Theorem dlist_finite_ind

Der Beweis des Theorems erfolgt im wesentlichen per Induktion über den Mengen-Typ nat der natürlichen Zahlen. Wir beginnen den Beweis, indem wir uns das Theorem als initiales Beweisziel vorgeben.

```
- val prems = goal Dlist.thy
  "⟦ P(⊥);P(dnil);
     ⋀x l1.⟦ x¬=⊥;l1¬=⊥;P(l1)⟧⟹P(dcons[x][l1])⟧
     ⟹∀l.P(dlist_take(n)[l])";
```

Die Prämissen der Regel verschwinden im impliziten Kontext prems des Beweises und wir erhalten als erstes Beweisziel:

```
 1. ∀l. P(dlist_take(n)[l])
```

Induktion über die Variable n ergibt die beiden neuen Teilziele:

```
- by (nat_ind_tac "n" 1);

 1. ∀l. P(dlist_take(0)[l])
 2. ⋀n1. ∀l. P(dlist_take(n1)[l])⟹∀l. P(dlist_take(Suc(n1))[l])
```

Das erste Ziel läßt sich leicht lösen, indem man dlist_take(0)[l] zu ⊥ vereinfacht und dann die Annahme P(⊥) im Kontext prems verwendet. Folgende Befehlssequenz führt diese Schritte durch:

```
- by (simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (resolve_tac prems 1);
```

Im zweiten Teilziel zeigen wir die Annahme für eine beliebige, aber feste Liste l und führen Listenfallunterscheidung gemäß der Regel dlistE

```
  ⟦ ?l = ⊥⟹?Q; ?l = dnil⟹?Q;
    ⋀x xl. ⟦ ?l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥⟧⟹?Q⟧
    ⟹?Q

- by (rtac allI 1);
- by (res_inst_tac [("l","l")] dlistE 1);
```

durch. Dies führt zu den neuen Teilzielen:

```
 1. ⋀n1 l.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = ⊥⟧⟹P(dlist_take(Suc(n1))[l])
 2. ⋀n1 l.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dnil⟧⟹P(dlist_take(Suc(n1))[l])
```

```
 3. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥⟧⟹
       P(dlist_take(Suc(n1))[l])
```

Die beiden ersten Teilziele lösen wir, indem wir zuerst mit dem Simplifikator vereinfachen und dann die Annahmen `P(⊥)` und `P(dnil)` im Annahmenkontext `prems` verwenden. Im dritten Teilziel vereinfachen wir ebenfalls mit dem Simplifikator. Die eben angesprochenen Aktivitäten werden durch folgende Befehlssequenz durchgeführt:

```
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (resolve_tac prems 1);
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (resolve_tac prems 1);
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
```

Dies führt zu folgendem Beweisziel:

```
 1. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥⟧⟹
       P(dcons[x][dlist_take(n1)[xl]])
```

In dieser Situation verwenden wir eine klassische Argumentation; entweder ist `dlist_take(n1)[xl]` definiert oder nicht. Wir setzen die Regel `classical2`

```
  ⟦ ?Q⟹?R; ¬?Q⟹?R⟧⟹?R

- by (res_inst_tac [("Q","dlist_take(n1)[xl]=⊥")] classical2 1);
```

ein und erhalten die beiden Fälle:

```
 1. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥;
          dlist_take(n1)[xl] = ⊥⟧⟹
       P(dcons[x][dlist_take(n1)[xl]])
 2. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥;
          dlist_take(n1)[xl] ¬= ⊥⟧⟹
       P(dcons[x][dlist_take(n1)[xl]])
```

Im ersten Fall vereinfachen wir wieder zu `P(⊥)` und verwenden die Annahme `P(⊥)` im Kontext `prems`.

```
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (resolve_tac prems 1);
```

Es verbleibt das Teilziel:

```
 1. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥;
          dlist_take(n1)[xl] ¬= ⊥⟧⟹
       P(dcons[x][dlist_take(n1)[xl]])
```

Diesmal verwenden wir die Annahme

```
⋀x l1.⟦ x¬=⊥;l1¬=⊥;P(l1)⟧⟹P(dcons[x][l1])
```

aus dem Kontext prems und erhalten:

```
- by (resolve_tac prems 1);

 1. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥;
          dlist_take(n1)[xl] ¬= ⊥⟧⟹
       x ¬= ⊥
 2. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥;
          dlist_take(n1)[xl] ¬= ⊥⟧⟹
       dlist_take(n1)[xl] ¬= ⊥
 3. ⋀n1 l x xl.
       ⟦ ∀l. P(dlist_take(n1)[l]); l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥;
          dlist_take(n1)[xl] ¬= ⊥⟧⟹
       P(dlist_take(n1)[xl])
```

Die ersten beiden Ziele können sofort durch Beweis per Annahme gelöst werden. Das dritte Ziel lösen wir, indem wir die Induktionsannahme ∀l. P(dlist_take(n1)[l]) für xl spezialisieren.

```
- by (atac 1);
- by (atac 1);
- by (etac spec 1);
No subgoals
```

6.6.3.2 Beweis für das Theorem dlist_all_finite_lemma1

Der Beweis beginnt wie üblich damit, daß wir uns das Theorem als initiales Beweisziel vorgeben.

```
- val prems = goal Dlist.thy
"∀l.dlist_take(n)[l]=⊥ ∨ dlist_take(n)[l]=l";
```

Wir erhalten als erstes Teilziel:

```
 1. ∀l. dlist_take(n)[l] = ⊥ ∨ dlist_take(n)[l] = l
```

welches wir durch Induktion über die Variable n in zwei neue Teilziele aufbrechen:

```
- by (nat_ind_tac "n" 1);

 1. ∀l. dlist_take(0)[l] = ⊥ ∨ dlist_take(0)[l] = l
 2. ⋀n1. ∀l. dlist_take(n1)[l] = ⊥ ∨ dlist_take(n1)[l] = l⟹
          ∀l. dlist_take(Suc(n1))[l] = ⊥ ∨ dlist_take(Suc(n1))[l] = l
```

Das erste Teilziel läßt sich automatisch durch Vereinfachung mit dem Simplifikator lösen. Im zweiten Teilziel genügt es, die Aussage für eine beliebige, aber feste Liste l zu zeigen.

```
- by (simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (rtac allI 1);
```

Dies führt zu folgendem neuen Teilziel:

```
 1. ⋀n1 l.
       ∀l. dlist_take(n1)[l] = ⊥ ∨ dlist_take(n1)[l] = l⟹
       dlist_take(Suc(n1))[l] = ⊥ ∨ dlist_take(Suc(n1))[l] = l
```

In dieser Situation führen wir eine Fallunterscheidung über den Aufbau der Liste `l` durch. Dies ergibt:

```
- by (res_inst_tac [("l","l")] dlistE 1);
```

```
 1. ⋀n1 l.
       ⟦ ∀l. dlist_take(n1)[l] = ⊥ ∨ dlist_take(n1)[l] = l; l = ⊥⟧⟹
       dlist_take(Suc(n1))[l] = ⊥ ∨ dlist_take(Suc(n1))[l] = l
 2. ⋀n1 l.
       ⟦ ∀l. dlist_take(n1)[l] = ⊥ ∨ dlist_take(n1)[l] = l; l = dnil⟧⟹
       dlist_take(Suc(n1))[l] = ⊥ ∨ dlist_take(Suc(n1))[l] = l
 3. ⋀n1 l x xl.
       ⟦ ∀l. dlist_take(n1)[l] = ⊥ ∨ dlist_take(n1)[l] = l; l = dcons[x][xl];
          x ¬= ⊥; xl ¬= ⊥⟧⟹
       dlist_take(Suc(n1))[l] = ⊥ ∨ dlist_take(Suc(n1))[l] = l
```

Die beiden ersten Ziele werden automatisch vom Simplifikator gelöst, der die Annahmen im lokalen Kontext selbständig zur Vereinfachung mitbenutzt. Im dritten Teilziel setzen wir ebenfalls den Simplifikator zur Vereinfachung ein.

```
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
```

Es verbleibt das Teilziel:

```
 1. ⋀n1 l x xl.
       ⟦ ∀l. dlist_take(n1)[l] = ⊥ ∨ dlist_take(n1)[l] = l; l = dcons[x][xl];
          x ¬= ⊥; xl ¬= ⊥⟧⟹
       dcons[x][dlist_take(n1)[xl]] = ⊥ ∨
       dcons[x][dlist_take(n1)[xl]] = dcons[x][xl]
```

In dieser Situation nützen wir die Induktionsannahme aus. Zuerst spezialisieren wir die Induktionsannahme für `xl`. Dann eliminieren wir die jetzt spezialisierte Disjunktion im lokalen Kontext mittels der Regel `disjE`

```
⟦ ?P ∨ ?Q; ?P⟹?R; ?Q⟹?R⟧⟹?R
```

Dies führt zu:

```
- by (eres_inst_tac [("x","xl")] allE 1);
- by (etac disjE 1);
```

```
1. ⋀n1 l x xl.
      ⟦ l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥; dlist_take(n1)[xl] = ⊥⟧⟹
      dcons[x][dlist_take(n1)[xl]] = ⊥ ∨
      dcons[x][dlist_take(n1)[xl]] = dcons[x][xl]
2. ⋀n1 l x xl.
      ⟦ l = dcons[x][xl]; x ¬= ⊥; xl ¬= ⊥; dlist_take(n1)[xl] = xl⟧⟹
      dcons[x][dlist_take(n1)[xl]] = ⊥ ∨
      dcons[x][dlist_take(n1)[xl]] = dcons[x][xl]
```

Beide Fälle werden automatisch durch den Simplifikator gelöst. Insbesondere im ersten Fall spielt dabei die Striktheit des Konstruktors `dcons` eine entscheidende Rolle!

```
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
No subgoals
```

6.6.3.3 Beweis für das Theorem dlist_all_finite_lemma2

Wir geben uns das Theorem als initiales Beweisziel vor. Dies ergibt:

```
- val prems = goal Dlist.thy "∃n.dlist_take(n)[l]=l";

1. ∃n. dlist_take(n)[l] = l
```

Wir verwenden eine klassische Argumentation und unterscheiden die beiden Fälle; entweder ist `l` definiert oder nicht:

```
- by (res_inst_tac [("Q","l=⊥")] classical2 1);

1. l = ⊥⟹∃n. dlist_take(n)[l] = l
2. l ¬= ⊥⟹∃n. dlist_take(n)[l] = l
```

Der erste Fall wird automatisch vom Simplifikator gelöst!

```
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
```

Im zweiten Fall verwenden wir das Hilfslemma

```
(∀n.dlist_take(n)[l]=⊥) ∨ (∃n.dlist_take(n)[l]=l)
```

welches in die Liste der noch zu beweisenden Teilziele aufgenommen wird! Die Einführung von Hilfslemmata erfolgt über die Taktik `subgoal_tac`. Das Hilfslemma erscheint weiter unten als zweites Teilziel:

```
- by (subgoal_tac  "(∀n.dlist_take(n)[l]=⊥) ∨ (∃n.dlist_take(n)[l]=l)" 1);

1. ⟦ l ¬= ⊥;
      (∀n. dlist_take(n)[l] = ⊥) ∨ (∃n. dlist_take(n)[l] = l)⟧⟹
   ∃n. dlist_take(n)[l] = l
2. l ¬= ⊥⟹(∀n. dlist_take(n)[l] = ⊥) ∨ (∃n. dlist_take(n)[l] = l)
```

Wir verwenden das Hilfslemma zur Fallunterscheidung im ersten Teilziel. Dies führt zu:
```
- by (etac disjE 1);

 1. [| l ¬= ⊥; ∀n. dlist_take(n)[l] = ⊥|]==>∃n. dlist_take(n)[l] = l
 2. [| l ¬= ⊥; ∃n. dlist_take(n)[l] = l|]==>∃n. dlist_take(n)[l] = l
 3. l ¬= ⊥==>(∀n. dlist_take(n)[l] = ⊥) ∨ (∃n. dlist_take(n)[l] = l)
```

Konzentrieren wir uns zunächst auf das erste Teilziel. Die beiden anderen Teilziele werden erst wieder aufgelistet, wenn das erste Teilziel gelöst ist.

Aus den Annahmen des ersten Teilziels kann sowohl `l=⊥` als auch `l¬=⊥` abgeleitet werden. Diesen Widerspruch nützen wir durch die Regel `notE`
```
 [| ¬?P; ?P|]==>?R
```

aus, die wir gleich mit einem Beweis per Annahme kombinieren (`eres_inst_tac`), da die eine Hälfte des Widerspruchs bereits im Kontext vorliegt. Nach Anwendung der Regel verbleibt:
```
- by (eres_inst_tac [("P","l=⊥")] notE 1);

 1. ∀n. dlist_take(n)[l] = ⊥==>l = ⊥
```

Eine Anwendung des Take-Lemmas für Listen liefert das Ziel
```
- by (rtac dlist_take_lemma 1);

 1. ⋀n. ∀n. dlist_take(n)[l] = ⊥==>dlist_take(n)[l] = dlist_take(n)[⊥]
```

welches wieder automatisch vom Simplifikator gelöst wird.
```
- by (asm_simp_tac (HOLCF_ss addsimps dlist_rews) 1);
```

Wir müssen noch die beiden Teilziele beweisen, die wir vorher zurückgestellt haben. Diese sind:
```
 1. [| l ¬= ⊥; ∃n. dlist_take(n)[l] = l|]==>∃n. dlist_take(n)[l] = l
 2. l ¬= ⊥==>(∀n. dlist_take(n)[l] = ⊥) ∨ (∃n. dlist_take(n)[l] = l)
```

Das erste Teilziel lösen wir trivial durch Beweis per Annahme. Für den Beweis des zweiten Ziels führen wir ein Hilfslemma ein, das gerade dem Theorem `dlist_all_finite_lemma1` entspricht. Durch diesen technischen Kniff läßt sich der klassische Beweiser (`fast_tac`) im Anschluß daran leichter einsetzen. Das eingeführte Hilfstheorem wird dem Kontext des zweiten Teilziels hinzugefügt und erscheint selbst als weiteres zu beweisendes Teilziel. Nach Anwendung von
```
- by (atac 1);
- by (subgoal_tac "∀n.∀l.dlist_take(n)[l]=⊥ ∨ dlist_take(n)[l]=l" 1);
```

erhalten wir:
```
 1. [| l ¬= ⊥; ∀n l. dlist_take(n)[l] = ⊥ ∨ dlist_take(n)[l] = l|]==>
    (∀n. dlist_take(n)[l] = ⊥) ∨ (∃n. dlist_take(n)[l] = l)
 2. l ¬= ⊥==>∀n l. dlist_take(n)[l] = ⊥ ∨ dlist_take(n)[l] = l
```

Das erste Teilziel wird nun automatisch von klassischen Beweiser gelöst! Das zweite Teilziel, also der Beweis für das Hilfstheorem, wird trivial durch die Verwendung des Theorems dlist_all_finite_lemma1 gelöst.

```
- by (fast_tac HOL_cs 1);
- by (rtac allI 1);
- by (rtac dlist_all_finite_lemma1 1);
No subgoals
```

6.6.3.4 Beweis für das Theorem dlist_ind

Der Beweis für das Theorem dlist_ind ist nahezu trivial, da eigentlich nur die beiden Theoreme dlist_all_finite_lemma2 und dlist_finite_ind geschickt kombiniert werden müssen.

Wir beginnen den Beweis, indem wir uns das gewünschte Theorem als initiales Beweisziel vorgeben.

```
- val prems = goal Dlist.thy
"⟦ P(⊥);P(dnil);
   ⋀x l1.⟦ x¬=⊥;l1¬=⊥;P(l1)⟧⟹P(dcons[x][l1])⟧⟹P(l)";
```

Die Annahmen der Regel verschwinden im globalen Kontext des Beweises prems und wir erhalten:

```
 1. P(l)
```

Das Theorem dlist_all_finite_lemma2

```
 ∃n. dlist_take(n)[?l] = ?l
```

ist eine Existenzaussage. Diese kombinieren wir mit der Regel zur Elimination des Existenzquantors exE

```
  ⟦ ∃x. ?P(x); ⋀x. ?P(x)⟹?Q⟧⟹?Q
```

```
- by (rtac (dlist_all_finite_lemma2 RS exE) 1);
```

und erhalten dadurch:

```
 1. ⋀x. dlist_take(x)[?l1] = ?l1⟹P(l)
```

Wir spezialisieren die Unifikationsvariable ?l1 im Kontext zu l und erhalten durch Gleichungslogik das Teilziel:

```
- by (etac subst 1);
```

```
 1. ⋀x. P(dlist_take(x)[l])
```

In dieser Situation verwenden wir das Theorem dlist_finite_ind für die Induktion auf

endlichen Listen und erhalten die drei neuen Teilziele:

```
- by (rtac (dlist_finite_ind RS spec) 1);

 1. ⋀x. P(⊥)
 2. ⋀x. P(dnil)
 3. ⋀x xa l1. ⟦ xa ¬= ⊥; l1 ¬= ⊥; P(l1)⟧⟹P(dcons[xa][l1])
```

Für alle drei Ziele sind passende Annahmen im globalen Kontext `prems` des Beweises enthalten, und somit lassen sich alle verbleibenden Ziele trivial lösen.

```
- by (REPEAT (resolve_tac prems 1));
- by (REPEAT (atac 1));
No subgoals
```

Kapitel 7

Ausblick

Im letzten Kapitel dieser Arbeit werde ich kurz auf zukünftige Arbeiten eingehen, die sich aufbauend auf der hier beschriebenen Entwicklung der Logik HOLCF anbieten.

Das Kapitel 5 hat gezeigt, wie initiale Lösungen für Bereichsgleichungen in der Logik HOLCF formalisiert werden können, und mit den in Kapitel 6 vorgestellten Datentypen liegen einige konkrete Beispiele vor. Gerade die Beispiele zeigen, daß sowohl die reine Axiomatisierung eines Datentyps, als auch die Herleitung einer hinreichenden Menge von Theoremen über den Datentyp einen nicht unerheblichen Arbeitsaufwand bedeuten.

Damit das HOLCF-System vernünftig für konkrete Fallstudien eingesetzt werden kann, muß unbedingt ein Datentyp-Definitionspaket zur Verfügung gestellt werden, das ausgehend von einer Beschreibung des gewünschten Datentyps, die der Benutzer liefert, automatisch die Isabelle-Theorie zur Axiomatisierung des Typs erzeugt und anschließend automatisch eine Reihe von Theoremen über den Datentyp beweist, die bei der praktischen Beweisarbeit notwendig sind.

Anregungen für ein solches Datentyp-Definitionspaket sind reichlich in der Literatur vorhanden. Für Edinburgh-LCF hat Robin Milner ein Paket zur Axiomatisierung von Datentypen entwickelt [CM82], das später von Brian Monahan [Mon85] verbessert wurde. Für Cambridge-LCF wurde von Larry Paulson [Pau84, Pau87] ein Datentyp-Definitionspaket entwickelt, das sogar die Axiomatisierung verschränkt rekursiver Datentypen erlaubt. Für den generischen Theorembeweiser Isabelle [Pau94] liegen ebenfalls Definitionspakete für Datentypen vor. Ein sehr umfangreiches Paket existiert für die Isabelle-Instanz der Zermelo-Fraenkel Mengentheorie. Ein anderes erlaubt die Definition von (Mengen-)Typen in der Isabelle-Instanz der Logik HOL, die Theorie für dieses Paket wird in [Pau93a] beschrieben.

Die obigen Pakete für die LCF-Logiken sind auf Bereichsgleichungen beschränkt, die baumartige Datentypen beschreiben. Genauer gesagt setzen sich die Funktoren, die der Bereichsgleichung zugrunde liegen, aus den primitiven Funktoren zusammen, die in Kapitel 5 beschrieben wurden. Zum Teil wird neben dem strikten Produkt auch noch die Verwendung des kartesischen Produkts zugelassen. Beim Entwurf eines Datentyp-Definitionspakets für die Logik HOLCF ist zu überlegen, ob nicht von vornherein die Verwendung von Funktoren mit gemischten Varianzen (mixed-variance functors) erlaubt werden soll. Die diesbezügliche technische Fundierung findet sich in [Fre90] und wurde am Ende des Abschnitts 5.1 kurz angedeutet.

Auf der anderen Seite stellt sich allerdings die Frage, ob die damit zur Verfügung gestellten zusätzlichen Modellierungsmöglichkeiten für den praktischen Einsatz relevant sind.

Ein weiterer Ansatzpunkt für zukünftige Arbeiten liegt in der Modellierung von Datentypen, die durch eine Quotientenbildung über einem bereits vorliegenden Typ entstehen. Beispiele hierfür sind endliche polymorphe Mengen, endliche polymorphe Multimengen oder rationale Zahlen. Im Beispiel der polymorphen Mengen müssen an den Parametertyp zusätzliche Anforderungen gestellt werden, damit die Quotientenbildung durchgeführt werden kann und implizite Stetigkeitseigenschaften der beteiligten Zugriffsfunktionen des Quotiententyps gewährleistet sind. Auch hier muß ein entsprechender Mechanismus zur Unterstützung des Benutzers bei der Datentyp-Definition vorgesehen werden.

Die Logik HOLCF bietet sich als Basis für Spezifikationssprachen bzw. Beschreibungstechniken an, die inhärent auf Konzepte der Bereichstheorie aufbauen. Ein Beispiel hierfür ist der Beschreibungsrahmen für verteilte Systeme FOCUS [BDD+93]. Der Ausschnitt von FOCUS, der sich mit der Spurspezifikation verteilter Systeme bzw. der Spezifikation stromverarbeitender Funktionen beschäftigt, kann ohne besondere Probleme auf HOLCF aufgesetzt werden, da die technisch anspruchsvollen Anteile der Beschreibungssprache bereits alle in HOLCF formalisiert sind. Dies hätte den weiteren Vorteil, daß damit für die in HOLCF formalisierten Anteile von FOCUS eine Unterstützung durch den Theorembeweiser Isabelle zur Verfügung stehen würde. Auch hier muß die reine Formalisierung von FOCUS in HOLCF von der Bereitstellung einer benutzerfreundlichen Eingabesyntax begleitet werden, die von der technischen Kodierung in Form von stromverarbeitenden Funktionen und deren Kombination abstrahiert.

Als letzter Ansatzpunkt für zukünftige Arbeiten sei noch die Entwicklung einer operationellen Semantik für einen Ausschnitt der Logik HOLCF genannt. Für Datentypen, die als initiale Lösungen von Bereichsgleichungen axiomatisiert sind, sowie für Operationen, die darauf aufbauend durch explizite Fixpunktdefinitionen eingeführt werden, sollte sich relativ unproblematisch eine operationelle Semantik angeben lassen, die zu der in dieser Arbeit entwickelten denotationellen Semantik paßt. Diese kann als Ansatz für die Auszeichnung einer Teilsprache von HOLCF verwendet werden, die die Einführung von Operationen durch bequemere 'Pattern-Matching'-Definitionen im Stil einer funktionalen Programmiersprache erlaubt. Da in HOLCF die Begriffe von Funktionen, stetigen Funktionen und Operationen klar umrissen sind, steht die Behandlung der technischen Probleme bei der Auszeichnung der operationellen Teilsprache auf einem sicheren Fundament.

Literaturverzeichnis

[Age93] Sten Agerholm. Domain theory in HOL. In *Proceedings of the 1993 International Meeting on Higher Order Logic Theorem Proving and its Applications*, LNCS Series. Springer-Verlag, August 1993.

[Age94] Sten Agerholm. *A HOL Basis for Reasoning about Functional Programs.* PhD thesis, Aarhus University, Computer Science Departement, 1994. (in preparation).

[AL91] A. Asperti and G. Longo. *Categories, Types, and Structures.* Foundations of Computing. The MIT Press, 1991.

[All86] L. Allison. *A practical introduction to denotational Semantics.* Cambridge University Press, 1986.

[AML+84] P. Andrews, D. Miller, E. Longini, Cohen, and F. Pfenning. Automating Higher Order Logic. *Contemporary Mathematics 29*, 1984.

[And63] Peter Andrews. A reduction of the axioms for the theory of propositional types. *Fund. Math. 52*, 1963.

[And86] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[Bar91] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

[BDD+93] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems: An Introduction to FOCUS. Technical Report TUM-I9202-2, Institut für Informatik, Technische Universität München, 1993.

[BFG+93a] Manfred Broy, Christian Facchi, Radu Grosu, Rudi Hettler, Heinrich Hussmann, Dieter Nazareth, Franz Regensburger, Oscar Slotosch, and Ketil Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part i. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, Fakultät für Informatik, TUM, 80290 München, Germany, May 1993.

[BFG+93b] Manfred Broy, Christian Facchi, Radu Grosu, Rudi Hettler, Heinrich Hussmann, Dieter Nazareth, Franz Regensburger, Oscar Slotosch, and Ketil Stølen. The

Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part ii. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, Fakultät für Informatik, TUM, 80290 München, Germany, May 1993.

[BW93] Manfred Broy and Martin Wirsing. Korrekte Software - Vom Experiment zur Anwendung. In *Tagungsband GI-Jahrestagung 1993*, 1993.

[Cam89] Juanito Camilleri. The HOL System Description, Version 1 for HOL 88.1.10. Technical report, Cambridge Research Center, 1989.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic 5*, 1940.

[CM82] A. Cohn and R. Milner. On using Edinburgh-LCF to prove the Correctness of a Parsing Algorithm. Technical Report CSR-113-82, EUCSD Report, 1982.

[dB73] Nicolaas G. de Bruijn. AUTOMATH, a language for mathematics. Technical report, Les Presses de l'Université de Montréal, 1973.

[DCC92] E. Downs, P. Clare, and I. Coe. *Structured systems analysis and design method (2nd ed).* Prentice-Hall, 1992.

[DM82] L. Damas and R. Milner. Principle Type-Schemes for Functional Programs. In *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[EGL89] H.D. Ehrich, M. Gogolla, and U.W. Lippeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, 1989.

[Fre90] Peter Freyd. Recursive types reduce to inductive types. In *Proceedings of the fifth annual IEEE symposium on Logic in Computer Science*. 1990.

[Fre91] Peter Freyd. Algebraically Complete Categories. In A Carboni et al., editors, *Proceedings of the 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104. Springer-Verlag, Berlin, 1991.

[Fre92] Peter Freyd. Remarks on algebraically compact categories. In *Applications of Categories in Computer Science*, number 177 in Notes of the London Mathematical Society. 1992.

[Gal86] Jean Gallier. *Logig for Computer Science.* Harper and Row, 1986.

[Gau86] M.-C. Gaudel. Towards Structured Algebraic Specifications. *ESPRIT '85', Status Report of Continuing Work (North-Holland)*, pages 493–510, 1986.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical report, Digital, Systems Research Center, Paolo Alto, California, 1985.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[Gog76] M. Gogolla. Partially Ordered Sorts in Algebraic Specifications. In B. Courcelle, editor, *Proc. 9th CAAP 1984, Bordeaux*. Cambridge University Press, 1976.

[Gog78] J.A. Goguen. Order Sorted Algebras: Exception and Error Sorts, Coercions and Overloaded Operators. *Semantics and Theory of Computation*, 14, 1978. University of California, Los Angeles.

[Gor85] M.J.C. Gordon. HOL: a machine oriented formulation of Higher Order Logic. Technical Report 68, Computer Laboratory, Univerity of Cambridge, 1985.

[GR94] Radu Grosu and Franz Regensburger. The Logical Framework of SPECTRUM. Technical Report TUM-I9402, Institut für Informatik, Technische Universität München, 1994.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, 1992.

[Han91] M. Hanus. Parametric Order-Sorted Types in Logic Programming. Technical Report 377, Universität Dortmund Fachbereich Informatik, January 1991.

[Hen50] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic 15*, 1950.

[Hen63] Leon Henkin. A theory of propositional types. *Fund. Math. 52*, 1963.

[Her72] H. Hermes. *Einführung in die mathematische Logik*. Teubner, 1972.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, May 1992.

[HLR93] Heinrich Hußmann, Jacques Loeckx, and Wolfgang Reif. Korrekte Software - Das Projekt KORSO. Tagungsband GI-Jahrestagung 1993, 1993.

[HMM86] R.W. Harper, D.B. MacQueen, and R.G. Milner. Standard ML. *Report ECS-LFCS-86-2, Univ. Edinburgh*, 1986.

[Hyl91] J.M.E. Hyland. First steps towards synthetic domain theory. In *Conference on Category Theory '90*, volume 1488 of *Lecture Notes in Mathematics*. Springer Verlag, Berlin, 1991.

[Jon91] Mark P. Jones. Type Inference For Qualified Types. Technical Report PRG-TR-10-91, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, 1991.

[KK92] Y. Kashiwagi and H. Kondoh. Domains. TEX version of course notes by Gordon Plotkin of a lecture in 1983 at the University of Edinburgh, 1992.

[LPT89] Z. Luo, R. Pollack, and P. Taylor. How to Use LEGO. *Departement of Computer Science, University of Edinburgh*, 1989.

[LS86] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, 1986.

[Luo89] Z. Luo. ECC, an Extended Calculus of Constructions. In *Proc. of the Fourth Ann. Symp. on Logic in Computer Science (LICS)*, pages 385 – 395, 1989.

[Luo91] Z. Luo. A unifying theory of dependent types I. Technical Report LFCS Report 91-154, Edinburgh, 1991.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mit90] J. C. Mitchell. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. Elsevier Science Publisher, 1990.

[Mit93] J. C. Mitchell. *Introduction to Programming Language Theory*. The MIT Press, 1993.

[Mon85] B. Monahan. *Data Type Proofs Using Edinburgh LCF*. PhD thesis, University of Edinburgh, 1985. Departement of Computer Science, CST-34-85.

[Nip89] Tobias Nipkow. Term Rewriting and Beyond — Theorem Proving in Isabelle. *FAC*, 1:320–338, 1989.

[Nip91] Tobias Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418, 1993.

[Pau84] L.C. Paulson. *Deriving Structural Induction in LCF*, volume 173 of *LNCS*, pages 197–214. Springer, 1984.

[Pau87] L.C. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[Pau89] L.C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[Pau92] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[Pau93a] L.C. Paulson. Co-induction and co-recursion in higher-order logic. Technical report, University of Cambridge, Computer Laboratory, 1993.

[Pau93b] L.C. Paulson. Isabelle's Object Logics. Technical Report 286, University of Cambridge, Computer Laboratory, 1993.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[Pet93] Kim Dam Petersen. Graph model of lambda : Constructed in higher order logic. Technical Report TFL RR 1993-4, Tele Danmark Research, 1993.

[Pho90] W. K.-S. Phoa. *Domain Theory in Realizability Toposes*. PhD thesis, Trinity College, Cambridge, 1990.

[Pit92] Andrew Pitts. A co-induction principle for recursively defined domains. Technical Report 252, University of Cambridge, Computer Laboratory, 1992.

[Pus94a] Cornelia Pusch. Übersetzung von SPECTRUM Spezifikationen nach Isabelle, 1994. Fortgeschrittenenpraktikum, Technische Universität München.

[Pus94b] Cornelia Pusch. Verifikation einer Entwicklung von AVL-Bäumen in Isabelle, 1994. Diplomarbeit, Technische Universität München.

[Qia90] Zhenyu Qian. Higher-Order Order-Sorted Algebras. In H. Kirchner and W. Wechsler, editors, *Algebraic and Logic Programming*, pages 86–100. Springer LNCS 463, October 1990.

[RS93] Bernhard Reus and Thomas Streicher. Naive synthetic domain theory – a logical approach. technical report, Universität München, 1993.

[Sch86] D.A. Schmidt. *Denotational Semantics*. Allan and Bacon, 1986.

[Sco72] D. Scott. Continuous lattices. In *Toposes, Algebraic Geometry and Logic*, pages 97–136. Springer, Berlin, 1972.

[Sco73] D. Scott. Models for various type–free calculi. In *Logic, Methodology and Philosophy of Science IV*, pages 157–187. North Holland, Amsterdam, 1973.

[Sco76] D. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3), September 1976.

[SG90] D. Scott and C. Gunter. Semantic Domains and Denotational Semantics. In *Handbook of Theoretical Computer Science*, chapter 12, pages 633–674. Elsevier Science Publisher, 1990.

[SHNR93] C. Sudergat, R. Hettler, D. Nazareth, and F. Regensburger. AVL-Trees: A case study for software development in SPECTRUM. Interner Bericht der TU München, 1993.

[Sok89] S. Sokolowski. Applicative High-Order Programming or Standard ML in the Battlefield. Technical Report MIP 8908, Universität Passau, Lehrstuhl für Informatik, February 1989.

[SS71] D. Scott and C. Strachey. Towards a mathematical semantics of computer languages. In *Proc. Symp. Computers and Automata, Brooklyn*, pages 19–46, 1971.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[SW83] D.T. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. Technical Report CSR-131-83, University of Edinburgh, Edinburgh EH9 3JZ, September 1983.

[Sza69] M.E. Szabo. *The collected papers of Gerhard Gentzen*. North Holland, 1969.

[Tak75] Gaisi Takeuti. *Proof Theory*. North Holland, 1975.

[Tay91] P. Taylor. The fixed point property in synthetic domain theory. In *Proc. of Logic in Computer Science (LICS)*, pages 152 – 160, 1991.

[VEB83] *Wörterbuch der Logik*. VEB Bibliographisches Institut Leipzig, 1983.

[Wan79] M. Wand. Fixed-Point Constructions in Order-Enriched Categories. *Theor. Comp. Sci.*, (8):13–30, 1979.

Abbildungsverzeichnis

Anhang A

Errata

Seite 174, Beispiel:

Im Beispiel müssen folgende Ersetzungen vorgenommen werden:

λ`x.`λ`z.x=z`	statt	λ`x.x=x`
Λ`x.`λ`z.x=z`	statt	Λ`x.x=x`
λ`z.y=z`	statt	`y=y`

Seite 232, Mitte:

Aus der Kommutativität des oberen Diagramms folgt weiter:

$$h \circ \theta = F\theta \circ Fh = F(\theta \circ h) = F(id_A) = id_{FA}$$

Seite 238, Mitte:

In kategorieller Sprechweise garantieren die Bedingungen cpo-cat1 und cpo-cat2, daß es zu jedem Diagramm $(\epsilon_n : D_n \to D_{n+1})_{n\in\omega}$ einen Co-Limes $(D, \iota_n)_{n\in\omega}$ gibt, so daß $(D, \rho_n)_{n\in\omega}$ zugleich ein Limes für das Diagramm $(\pi_n : D_{n+1} \to D_n)_{n\in\omega}$ ist.

Seite 249, Unten:

Die Anwendung auf Operationen t_1 ist definiert als:

$$K_\tau t_1 = \texttt{ID} :: \tau \to \tau$$

Seite 251, Mitte:

$$\texttt{ssplit[}\Lambda\texttt{xy.(ID::}\alpha \to \alpha\texttt{)[x]\#\#(lift[up oo }t_1\texttt{])[y]]}$$

Seite 252, Oben:

$$\begin{array}{lcl} [\![K^O_\tau]\!]_\nu & : & X \mapsto [\![\tau]\!]_\nu \\ [\![K^M_\tau]\!]_\nu & : & m \mapsto ([\![\texttt{ID}::\tau \to \tau]\!]_\nu, [\![\tau]\!]_\nu, [\![\tau]\!]_\nu) \end{array}$$