TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Datentypspezifikationen in Higher-Order LCF

Diplomarbeit

David von Oheimb

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN
UNIVERSITÄT MÜNCHEN

Forschungs- und Lehreinheit IV
Programmiermethodik, Verteilte Systeme

Datentypspezifikationen in Higher-Order LCF

Diplomarbeit

David von Oheimb[1]

Aufgabensteller	:	Prof. Tobias Nipkow
Betreuer	:	Dr. Franz Regensburger
Abgabedatum	:	15. November 1995

[1]e-mail: oheimb@informatik.tu-muenchen.de

Hiermit erkläre ich, daß ich diese Diplomarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe.

München, den 15. November 1995

(*David von Oheimb*)

Zusammenfassung

Für die Logik HOLCF (Regensburger) wird das Beweissystem Isabelle (Paulson, Nipkow) um ein Datentyp-Paket erweitert, das die Eigenschaften von durch rekursive Gleichungen beschriebenen Datentypen bestimmt und beweist bzw. aus Datentypgenerierungsregeln ein Induktionsaxiom erzeugt. Die Datentypen dürfen verschränkt rekursiv sein und nicht-strikte Konstruktoren enthalten.

Für mittels Bereichsgleichungen definierte Typen entsteht eine konservative Theorieerweiterung, was eine bereits gegebene externe kategorientheoretische Argumentation bestätigt, die hier weiter motiviert und auf simultane Gleichungen angepaßt worden ist. Als Axiome werden lediglich aus den Gleichungen resultierende Isomorphismen und eine Fixpunkteigenschaft des sog. 'Kopierfunktionals' benötigt.

Es werden eine Reihe von anwendungsrelevanten Theoremen bezüglich der Konstruktoren, Diskriminatoren und Selektoren der Datentypen sowie Induktions- und Coinduktionsprinzipien bewiesen. Weil sie den Umgang mit Datentypen stark vereinfacht, ist diese Arbeit ein wichtiger Schritt zur Verwendbarkeit von Isabelle/HOLCF als Spezifikations- und Verifikationswerkzeug.

Inhaltsverzeichnis

You can only find *truth* with logic
if you have already found truth without it.
G.K. Chesterton, *The Man who was Orthodox*

Jesus spricht [...]: Ich bin der Weg, *die Wahrheit* und das Leben;
niemand kommt zum Vater außer durch mich.

Wenn ihr an meinem Wort bleibt, so seid ihr wahrhaftig meine Jünger
und werdet *die Wahrheit* erkennen, und die Wahrheit wird euch frei machen.
Die Bibel, *Johannes 14,6* und *8,31b-32*

Kapitel 1

Einführung

1.1 Einleitung

Bei der Entwicklung von informationsverarbeitenden Systemen, sei es Computersoftware oder hochstrukturierte Hardware, spielt die Frage nach ihrer Korrektheit eine entscheidende Rolle. Die Systeme werden immer umfangreicher, auch unübersichtlicher und damit fehlerträchtiger. Weil sie oft in großem Rahmen und zum Teil an kritischen Stellen eingesetzt werden, bergen sie ein hohes finanzielles und zeitliches Risiko oder können sogar sicherheitsgefährdend sein. Es ist also sehr wichtig, daß Fehler so weit wie nur irgend möglich vermieden oder zumindest rechtzeitig erkannt werden.

Derzeit bemühen sich Informatiker weltweit, das Problem der Korrektheitssicherung in den Griff zu bekommen, indem sie Verfahren zur Spezifikation und Verifikation der Systeme entwickeln. In Deutschland gibt es zu diesem Zweck zum Beispiel das vom (ehem.) Bundesministerium für Forschung und Technik geförderte Projekt KORSO [Hußmann Loeckx Reif 93]. Man versucht, auf möglichst formale, exakte Weise zum einen zu beschreiben, was ein System überhaupt leisten soll, und zum anderen zu beweisen, daß eine gegebene Implementierung diesen Anforderungen auch genügt. Für beide Gebiete ist es notwendig, die Bedeutung aller Komponenten eines Systems, speziell der verwendeten programmiersprachlichen bzw. schaltungstechnischen Konzepte, genau zu kennen und damit umgehen zu können. Hier leistet die Semantik als Teilgebiet der Theoretischen Informatik ihren Beitrag. Ein wesentlicher Aspekt der Semantik ist die Behandlung von Datentypen, um die es auch in dieser Arbeit geht.

Franz Regensburger hat in seiner Dissertation [Regensburger 94] eine Variante höherer Stufe für die bekannte Logik LCF von Dana Scott mit dem Namen *HOLCF* (= Higher-Order Logic of Computable Functions) entwickelt, und zwar als konservative Theorieerweiterung der Prädikatenlogik höherer Stufe HOL, die für den generischen Theorembeweiser Isabelle [Paulson 94 I] zur Verfügung steht. Als wichtige Anwendung der Logik HOLCF behandelt er darin Datentypen, die mit Hilfe rekursiver Bereichsgleichungen beschrieben werden können, und zwar sowohl den theoretischen Hintergrund als auch die (exemplarische) Implementierung in Isabelle.

Das Ziel dieser Arbeit war es nun, ein Datentyp-Paket zu entwickeln, das *allgemein* aus Datentypspezifikationen die Eigenschaften der beschriebenen Datentypen bestimmt und diese dem Benutzer in Form von Axiomen und Theoremen zur Verfügung stellt. Damit können in Isabelle/HOLCF auch komplexe Datentypen bequem und einheitlich verwendet werden. Es ist also ein Werkzeug entstanden, das den Informatiker bei den oben erwähnten Spezifikations- und Verifikationsaufgaben unterstützt.

1.2 Aufgabenstellung und Arbeitsumfang

Die ursprüngliche Aufgabenstellung der Diplomarbeit umfaßt folgende Punkte:

- Erweiterung des Isabelle-Theorieparsers für die HOLCF-Logik um eine `domain`-Sektion, die die Spezifikation von freien Datentypen im Stil funktionaler Programmiersprachen erlaubt.

- Aus der syntaktischen Information der `domain`-Sektion soll in einem ersten Schritt eine Menge von Axiomen (einschließlich Ableitungsregeln) erzeugt werden, die den freien Typ eindeutig charakterisiert und dem Benutzer einen *hinreichend komfortablen* Satz an logischen Eigenschaften der Theorie zur Verfügung stellt.

- In einem zweiten Schritt soll die Axiomatisierung des Typs in der Hinsicht optimiert werden, daß eine konservative Theorieerweiterung mit wenigen Kern-Axiomen entsteht und die restlichen für die Benutzbarkeit notwendigen Eigenschaften und Ableitungsregeln automatisch vom Isabelle-System als Theoreme bewiesen werden.

- Es soll die Spezifikation sowohl von freien Datentypen als auch von nichtfreien Typen unterstützt werden. Dazu soll eine `generated`-Sektion eingebaut werden, die die Funktion des aus SPECTRUM bekannten Konstrukts 'generated by' übernimmt. Hier wird im wesentlichen eine Induktionsregel als Axiom erzeugt; die Theorieerweiterung ist nicht konservativ.

Aus praktischen Gründen sind der zweite und der dritte Punkt hiervon gleich in einem gemeinsamen Schritt bearbeitet worden. Außerdem ist eine minimale und eine erhebliche Erweiterung der Arbeit hinzugekommen:

- Auf eine spätere Anregung des Aufgabenstellers und des Betreuers hin wird eine Syntax-Transformationsregel erzeugt, die die Formulierung von Fallunterscheidungen über einem Datentyp erleichtert.

- Aus Interesse des Bearbeiters und der Anwender werden nicht nur einfache, sondern auch simultane Datentypgleichungen (für verschränkt rekursive Datentypen) behandelt. Dies macht eine Erweiterung der theoretischen Rechtfertigung des verwendeten Verfahrens erforderlich.

1.3 Überblick

Hier sei zunächst der Aufbau der Arbeit skizziert.
Nach dieser Einführung folgt in Kapitel 2 eine Kurzvorstellung der Programmierumgebung und der Logiken, die hier verwendet werden.
Kapitel 3 geht auf die theoretischen Grundlagen der Arbeit ein. Es beinhaltet eine Motivation für den gewählten theoretischen Ansatz, einen Abriß der Ergebnisse dieses Ansatzes sowie dessen Erweiterung für simultane Gleichungen.
Die allgemeine Struktur der Datentypspezifikationen, ihre Syntax und Kontextbedingungen werden in Kapitel 4 behandelt.
Kapitel 5 definiert die Bedeutung der Datentypspezifikationen, also die Isabelle-Syntax, Axiome und Theoreme, die das Datentyp-Paket aus diesen erzeugt, und veranschaulicht sie durch ein ausführliches Beispiel.
Die Beweise der Theoreme werden in Kapitel 6 anhand des Beispiels aus dem vorigen Kapitel erläutert.
Kapitel 7 beschreibt die Implementierung des Datentyp-Pakets und insbesondere die allgemeine Programmierung eines Theore-Extenders in Isabelle.
Schließlich gibt Kapitel 8 ein komplettes Anwendungsbeispiel und Hinweise zu Einsatz- und Erweiterungsmöglichkeiten des Datentyp-Pakets.
Nach dem Literaturverzeichnis sind im Anhang Installationshinweise u.ä. aufgeführt und die Quelltexte des Datentyp-Pakets abgedruckt.

Kapitel 2

Hilfsmittel

In diesem Kapitel werden die Programmierumgebung und die in der Arbeit verwendeten Logiken vorgestellt, soweit dies hier relevant bzw. in knapper Form möglich ist.

2.1 Isabelle

Die Programmierumgebung des Datentyp-Pakets ist das Beweissystem Isabelle von Lawrence C. Paulson, das am Lehrstuhl IV der Fakultät für Informatik der Technischen Universität München unter Leitung von Prof. Tobias Nipkow eingesetzt und erweitert wird. Es ist in [Paulson 94 I] und [Paulson 94 II] ausführlich beschrieben, weshalb hier nur ein kurzer Abriß über die Eigenschaften des Systems folgt.

Isabelle ist generisch sowohl bezüglich der verwendbaren Objektlogiken als auch bezüglich der zum Einsatz kommenden Beweistechniken. Seine universelle Metalogik, genannt Pure, ist eine intuitionistische Logik höherer Stufe mit einem polymorphen Typsystem mit Typklassen, ähnlich wie bei der Programmiersprache Haskell. Als Termsprache wird der einfach getypte λ-Kalkül verwendet, mit den syntaktischen Elementen (definitorische) Gleichheit $\equiv$, Meta-Implikation $\Longrightarrow$, λ-Abstraktion und Meta-Allquantifikation $\bigwedge$.

Die Metalogik erlaubt Beweisschritte wie beim Kalkül des natürlichen Schließens, insbesondere Resolutionsschritte vorwärts von den Prämissen oder rückwärts von der Konklusion eines Teilziels, sowie Gleichungslogik mit dem sog. Simplifier. Für viele Objektlogiken kann der sog. Classical Reasoner definiert werden, der per Tiefen- oder Breitensuche viele Standardaufgaben automatisch löst. Beweise lassen sich interaktiv oder als Programm führen. Zur Formulierung einzelner Schritte dienen Taktiken, die Übergänge zwischen den Beweiszuständen bewirken, und Takticals als Kombinatoren für Taktiken, womit sehr komplexe anwendungsspezifische Beweisverfahren möglich sind.

Die Definition von Objektlogiken geschieht durch Theorien, wobei der Benutzer bei der Syntax (mit graphischen Symbolen, Mixfix-Operatoren und Transformationsregeln) und ebenso bei der Formulierung von Axiomen und Definitionen durch die Ausdrucksmächtigkeit der Metalogik große Freiheiten hat.

2.2 ML

Für Isabelle wird sowohl bei der Implementierung als auch der Anwendung die Sprache ML = Meta Language eingesetzt, genauer Standard ML. Sie ist als allgemeine *funktionale* Sprache relativ bekannt, aber wie der Name bereits ausdrückt, war sie ursprünglich als Metasprache für Logik-Anwendungen gedacht, und diese Aufgabe hat sie hier tatsächlich. Die Sprache und ihre Anwendung in der Symbolverarbeitung wird in dem hervorragenden Buch [Paulson 92] ausführlich behandelt, so daß hier auf nähere Ausführungen verzichtet werden kann.

Das Datentyp-Paket macht viel Gebrauch von den komplexen Möglichkeiten zur Listenverarbeitung und den angenehmen Infix-Operatoren, während es die prozeduralen Elemente der Sprache lediglich am Rande für die Ein- und Ausgabe nutzt. Besonders bei der Isabelle-Implementierung selbst, aber auch hier, spielen das polymorphe Hindley/Milner-Typsystem, Funktionen höherer Stufe und die Modularisierungsmöglichkeiten von SML eine entscheidende Rolle. Für die bloße Anwendung des Isabelle-Systems und des hier beschriebenen Datentyp-Pakets werden allerdings nur geringe ML-Kenntnisse benötigt.

2.3 HOL mit Ergänzungen

HOL = Higher-Order Logic ist eine Prädikatenlogik höherer Stufe nach Church, die zuerst von Gordon (siehe [Gordon Melham 93]) implementiert worden ist und auch für Isabelle als Objektlogik zur Verfügung steht, siehe [Paulson 94 III, Chapter 4]. Sie dient hier als Basis zur Entwicklung von HOLCF, wie in den beiden folgenden Abschnitten dargestellt.

Neben den üblichen aussagenlogischen Konnektiven $\wedge$, $\vee$ und $\neg$ sowie den (Objekt-)Quantoren $\forall$ und $\exists$ gibt es in HOL die (Objekt-)Gleichheit `=` und den Hilbert-Operator ε, außerdem Unterstützung für den Umgang mit Funktionen. Es gilt das Deduktionstheorem, das tertium non datur und die Extensionalität von Funktionen.

Aus der Sicht des Anwenders fällt sehr angenehm auf, daß HOL das relativ mächtige Typsystem von Pure übernimmt, weshalb die Typinferenz auch automatisch geschieht. Für Beweise stehen die üblichen Einführungs- und Eliminationsregeln des natürlichen Schließens zur Verfügung, wie z.B.
`conjI = "⟦ P; Q ⟧ ⟹ P ∧ Q"` oder
`allE  = "⟦ ∀x. P x; P x ⟹ R ⟧ ⟹ R"`.
Zudem sind der Simplifier und der Classical Reasoner definiert, die mittels der Taktiken `simp_tac HOL_ss` bzw. `fast_tac HOL_cs` aufgerufen werden können.

Durch Hinzunahme der Datentypen Menge, Summe, Produkt und natürliche Zahlen, sowie der Konzepte kleinster Fixpunkt, transitive Hülle und Noethersche Induktion wird HOL weiter (zur Theorie Nat) ergänzt.

2.4 LCF

Die von Scott entwickelte Logik LCF = Logic of Computable Functions, die u.a. in [Paulson 87, Chapter 3] beschrieben ist, dient der formalen Argumentation über die Berechnung von Werten, mit besonderer Betrachtung der Definiertheit und Approximierbarkeit. Sie ist - neben HOL - das zweite Standbein von HOLCF.

LCF ist ein typisierter λ-Kalkül und eine Prädikatenlogik *erster* Stufe (mit einem Kalkül des natürlichen Schließens), die um die folgenden Konzepte[1] zur Behandlung der Berechenbarkeit erweitert wird.

Zunächst einmal gibt es primitive (Objekt-)Typen, wie den der Wahrheitswerte `tr`, und zusammengesetzte Typen wie das kartesische Produkt oder den der stetigen, also meist *berechenbaren* Funktionen. Jeder Typ bezeichnet einen Bereich, eine Trägermenge mit einer partiellen Ordnung $\sqsubseteq$, die als "schwächer definiert oder gleich" interpetiert wird. Damit ist die Gleichheit = auf den Typen durch $\texttt{x = y} \equiv \texttt{x} \sqsubseteq \texttt{y} \wedge \texttt{y} \sqsubseteq \texttt{x}$ definiert. Die partielle Ordnung besitzt ein kleinstes Element $\bot$, das Nichtterminierung einer Funktion darstellt. Die Regeln $[\![\ \texttt{x} \sqsubseteq \texttt{y};\ \texttt{f} \sqsubseteq \texttt{g}\ ]\!] \Longrightarrow \texttt{f`x} \sqsubseteq \texttt{g`y}$ und $(\bigwedge \texttt{x. f`x} \sqsubseteq \texttt{g`x}) \Longrightarrow \texttt{f} \sqsubseteq \texttt{g}$ legen fest, daß die Funktionen monoton und extensional sind. Schließlich bezeichnet `fix` den Operator zur Bestimmung des kleinsten (d.h. am wenigsten definierten) Fixpunkts einer Funktion, für den die Regel zur Fixpunktinduktion $[\![\ \texttt{adm P}^2;\ \texttt{P} \bot;\ \bigwedge \texttt{x. P x} \Longrightarrow \texttt{P (F`x)}\ ]\!] \Longrightarrow \texttt{P (fix`F)}$ gefordert wird.
Hinter den gerade genannten Konzepten auf Objektebene stecken einige in der Objektlogik nicht formulierbare Meta-Argumentationen: Für die Approximation von Werten spielt der Begriff der ω-Kette, also einer (unendlichen) Folge aufsteigend geordneter Elemente, eine wichtige Rolle. Jeder Bereich ist eine PCPO = Pointed Complete Partial Order, besitzt also für alle ω-Ketten D eine kleinste obere Schranke $\bigsqcup_{n\in\omega} D$, und die Funktionen sind als stetig vorausgesetzt, d.h. es gilt $f(\bigsqcup_{n\in\omega} D) = \bigsqcup_{n\in\omega}(f(D))$ für alle f und D. Damit läßt sich die Fixpunktinduktionsregel rechtfertigen, wobei die Zulässigkeit (admissibility) des betreffenden Prädikats, eine Aussage zweiter Stufe, wesentlich ist.

Es entsteht das Problem, daß man für alle Beweise, die Approximationen potentiell unendlicher Elemene betreffen, im Kalkül von LCF die Fixpunktinduktion benutzen muß, welche einen unbequemen Zulässigkeitstest erfordert, für den es zudem im Kalkül lediglich unvollständige *syntaktische* Kriterien gibt. Eine weitere Einschränkung von LCF ergibt sich daraus, daß keine nicht-stetigen Funktionen und allgemeinen Prädikate behandelt werden können.

2.5 HOLCF

Wie der Name bereits suggeriert, stellt HOLCF = Higher-Order LCF eine Erweiterung von HOL um LCF dar, sie kann aber auch umgekehrt als Variante höherer Stufe von LCF angesehen werden. Diese Logik wurde von Franz Regensburger in seiner Dissertation [Regensburger 94] entwickelt. Auf die dort gegebene ausführliche Motivation wird hier zum Teil eingegangen.

[1] Hier wird bereits HOLCF-Notation benutzt, z.B. der Funktionsapplikations-Operator `

[2] Zulässigkeit von P, eigentlich eine (syntaktische) Anwendbarkeitsbedingung der Regel

Neben der einfachen Summation aller positiven Eigenschaften von HOL und LCF entsteht durch die Kombination eine große Ausdrucksmächtigkeit mit entsprechenden Vorteilen gegenüber LCF. So werden mittels der Typklassen von HOL die Konzepte der Bereichstheorie, wie zum Beispiel die Klasse der PCPOs, durch die Methode der konservativen Theorieerweiterung eingeführt. Für eine neue Klasse definiert man dabei zunächst die charakteristischen Eigenschaften und beweist dann, daß die betreffenden Objekte diese Eigenschaften tatsächlich erfüllen. In diesem Zusammenhang ist die Möglichkeit sehr wichtig, Konzepte höherer Stufe wie die Stetigkeit von Funktionen, kleinste Fixpunkte und die Zulässigkeit von Prädikaten *im Kalkül* nicht nur zu axiomatisieren, sondern sogar zu *definieren* und deren Eigenschaften zu beweisen, was ebenfalls zur gesicherten Konsistenz der Theorieerweiterung beiträgt.

Unter anderem kann man somit viele Aussagen über kleinste Fixpunkte ohne die Fixpunktinduktionsregel *direkt* zeigen, und falls diese Regel doch verwendet werden soll, die Zulässigkeit des betreffenden Prädikats auf *semantischer* Ebene nachweisen. Ein interessanter Anwendungsfall hierzu sind die Regeln zur Strukturellen Induktion über Datentypen, die das Datentyp-Paket wie in Unterabschnitt 6.4.4 dargestellt beweist, wobei für endliche Datentypen die Zulässigkeitsvoraussetzung über das Induktionsprädikat entfällt!

Die Theorie HOLCF stellt, aufbauend auf der Theorie `Nat`, unter anderem folgende Konzepte bereit: Zunächst die Typklasse `pcpo` der kettenvollständigen partiellen Ordnungen mit kleinstem Element, die so einfache Vertreter enthält wie den leeren Datentyp `void`, aber auch so komplexe wie den der stetigen Funktionen (mit der Form `'a → 'b`)[3] zwischen `pcpo`-Typen. Daß sie tatsächlich dieser Klasse angehören, ist in HOLCF bewiesen worden, wobei z.B. der Begriff der ω-Kette durch das Prädikat `is_chain F ≡ ∀i. F i ⊑ F (Suc i)` definiert ist. Für stetige Funktionen, die hier `Operationen` genannt werden, gibt es eine eigene λ-Abstraktion `Λ` und den Infix-Applikationsoperator ` ‘ ` auf `pcpo`-Typen. Die *Bildung* von Termen für Operationen ist nicht eingeschränkt, während die stetige *Anwendung* von Funktionen einem Stetigkeitstest unterliegt:
`cont c1 ⟹ (Λx. c1 x)‘u = c1 u`, den aber die Taktik `cont_tac`, die auch in `HOLCF_ss` eingebaut ist, meist automatisch erledigt. Eigenschaften der Operationen wie `contlub_cfun_fun = "is_chain FY ⟹ lub (range FY)‘x = (⨆i. FY i‘x)"` stehen in Form von Theoremen ebenfalls zur Verfügung.

Weitere Mitglieder der Klasse `pcpo` sind das kartesische Produkt `'a × 'b`, das strikte Produkt `'a ⊗ 'b`, die strikte Summe `'a ⊕ 'b`, das Lifting `'a u`, der einelementige Typ `one` und der Boolesche Typ `tr`. Ferner wird die Operation `fix` mit Typ `('a → 'a) → 'a` zur Bestimmung des kleinsten Fixpunkts einer Operation definiert, dessen wesentliche Eigenschaften ebenfalls bewiesen sind, insbesondere die Fixpunktinduktionsregel. Dabei ist das Zulässigkeitsprädikat als `adm P ≡ ∀Y. is_chain Y ⟶ (∀i. P (Y i)) ⟶ P (lub (range Y))` definert, und die Theoremliste `adm_thms` enthält viele Regeln zum Beweis der Zulässigkeit eines Prädikats.

Schließlich gibt es zur Komposition von Operationen den Infixoperator `oo` mit der Identität `ID`, was die Klasse `pcpo` syntaktisch zu einer Kategorie ergänzt.

---

[3] `'a` und `'b` sind Typvariablen von Isabelle.

2.6 Kategorientheorie

Die **Kategorientheorie** befaßt sich mit den abstrakten Eigenschaften, die allen Arten von komponierbaren Abbildungen gemeinsam sind. Nachdem das folgende Kapitel zum theoretischen Hintergrund dieser Arbeit nur wenige grundlegende Begriffe der Kategorientheorie verwendet, werden diese hier kurz aufgeführt, und ansonsten sei auf die einschlägige Literatur wie [Mac Lane 72] verwiesen.

Eine **Kategorie** $\mathcal{C}$ besteht aus

- einer Kollektion $Obj(\mathcal{C})$ von **Objekten**,
- einer Kollektion $Mor(\mathcal{C})$ von **Morphismen**,
- zwei Operationen dom und cod, die jedem Morphismus ein Objekt zuordnen, d.h. ein Element aus $Obj(\mathcal{C})$, genannt **Quelle** bzw. **Ziel**,
- einer Operation id, die **Identität**, die jedem Objekt A einen Morphismus $id_A \in Mor(\mathcal{C})$ mit $dom(id_A) = cod(id_A) = A$ zuordnet,
- einer Operation $\circ$, die **Komposition**, die je zwei Morphismen f und g mit $dom(f) = cod(g)$ einen Morphismus $f \circ g$ mit $dom(f \circ g) = dom(g)$ und $cod(f \circ g) = cod(f)$ zuordnet, wobei (falls h ein weiterer Morphismus mit $cod(h) = dom(g)$ ist) gilt:
  - $id_{cod(f)} \circ f = f$,
  - $g \circ id_{dom(g)} = g$, und
  - $(f \circ g) \circ h = f \circ (g \circ h)$.

Die Kollektion aller Morphismen mit Quelle A und Ziel B wird mit $\mathcal{C}(A, B)$ bezeichnet, und daß f ein solcher Morphismus ist, kurz durch $f\colon A \to B$ notiert.

Ein Objekt 0 einer Kategorie $\mathcal{C}$ heißt **initial**, wenn es für jedes Objekt B genau einen Morphismus $f\colon 0 \to B$ gibt, und **terminal**, wenn es für jedes Objekt A genau einen Morphismus $f\colon A \to 0$ gibt.

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien, dann ist ein **(covarianter) Funktor** $F\colon \mathcal{C} \to \mathcal{D}$ definiert als ein Paar von Abbildungen $F\colon Obj(\mathcal{C}) \to Obj(\mathcal{D})$ und $F\colon Mor(\mathcal{C}) \to Mor(\mathcal{D})$, so daß für alle Morphismen f und g und Objekte A gilt:

- $F(f)\colon F(dom(f)) \to F(cod(f))$,
- $F(g \circ f) = F(g) \circ F(f)$, und
- $F(id_A) = id_{F(A)}$.

Ein Funktor $F\colon \mathcal{C} \to \mathcal{C}$ heißt **Endofunktor** auf $\mathcal{C}$.
Bei den in Kapitel 3 vorkommenden Kategorien liegt auf den Morphismen meist eine PCPO-Struktur vor, und die Funktoren sind dann strukturerhaltende Abbildungen zwischen den Kategorien.

Für zwei Kategorien $\mathcal{C}$ und $\mathcal{D}$ besteht die **Produktkategorie** $\mathcal{C} \times \mathcal{D}$ aus Paaren von Objekten bzw. Morphismen aus $\mathcal{C}$ und $\mathcal{D}$, wobei $(f', g') \circ (f, g) = (f' \circ f, g' \circ g)$ definiert wird. Auf solchen zusammengesetzten Elementen sind die Funktoren P und Q als Projektionen auf die erste bzw. zweite Komponente erklärt.

Kapitel 3

Theoretischer Hintergrund

Dieses Kapitel gibt einen Abriß über die dem Datentyp-Paket zugrundeliegende Theorie, die in [Regensburger 94, Kapitel 5] dargestellt ist, allerdings erweitert um eine ausführlichere Motivation der kategorientheoretischen Abstraktion und eine Anpassung für simultane Gleichungen.

3.1 Bereichsgleichungen und ihre Lösung

Unter der Lösung einer rekursiven Bereichsgleichung $X \cong FX$ versteht man einen Bereich A, der isomorph zu FA ist, und der durch eine geeignete Minimalitätsbedingung eindeutig ist. Diese Bedingung ist die Initialität einer sog. F-Algebra, wie unten ausgeführt. Sie ist so gewählt, daß sie auch die Ableitbarkeit von Induktionsprinzipien für den Bereich sichert.

Zu diesem sehr abstrakten kategorientheoretischen Lösungsansatz wird hier eine *Analogie* gegeben, nämlich die Lösung einer rekursiven Gleichung in einer PCPO. Dazu muß allerdings die Isomorphie von Bereichen als Gleichheit aufgefaßt werden. So kann man die Lösung der Bereichsgleichung analog dem Satz von Kleene als kleinsten Fixpunkt des zugehörigen stetigen Bereichsfunktionals F auffassen. Um zu zeigen, daß dies eine sinnvolle Interpretation ist, sollte zunächst geklärt werden, inwiefern das Universum pcpo von Bereichen mit PCPO-Struktur seinerseits eine PCPO-Struktur besitzt, bezüglich der man überhaupt von kleinsten Fixpunkten u.ä. sprechen kann, und die auch sicherstellt, daß es genau einen solchen Fixpunkt gibt.

Als (quasi-) partielle Ordnungsrelation zwischen Bereichen wird die Existenz eines speziellen Funktionenpaars zwischen den Bereichen definiert:

Definition 3.1 *EP-Paare,* $\lhd$

> Seien A und B Bereiche. Ein Paar (ϵ, π) von stetigen Funktionen $\epsilon\colon A \to B$ und $\pi\colon B \to A$ heißt EP-Paar (Einbettungs-/Projektionspaar), wenn gilt:
>
> $$\pi \circ \epsilon = id_A \quad \text{und} \quad \epsilon \circ \pi \sqsubseteq id_B$$
>
> Daß (ϵ, π) ein EP-Paar zwischen A und B ist, kann durch $\epsilon : A \lhd B$ dargestellt werden, weil dann π durch ϵ bereits eindeutig bestimmt wird und A ist insofern 'kleiner' als B ist, als ϵ injektiv von A nach B abbildet.

Die Relation $\lhd$ ist tatsächlich eine (quasi-) partielle Ordnung, denn sie ist reflexiv und transitiv (leichter Beweis), und immerhin quasi-antisymmetrisch, d.h. antisymmetrisch bis auf Isomorphie:
Für alle Bereiche A und B mit $\epsilon : A \lhd B$ und $\epsilon' : B \lhd A$ gilt, daß ϵ und ϵ' Injektionen zwischen diesen sind, und nach dem Satz von Schröder-Bernstein gibt es auch eine Bijektion zwischen A und B, sie sind also isomorph.

Desweiteren ist $\lhd$ sogar eine vollständige (quasi-) partielle Ordnung mit kleinstem Element `void`, wie mit Hilfe des folgenden Theorems deutlich wird:

Theorem 3.1 *Kleinstes Element, Vollständigkeit von* $\lhd$

Für jeden Bereich D und jede Folge von Bereichen $D_0,\ D_1,\ \ldots\ D_n,\ \ldots$ mit
$\epsilon_0 : D_0 \lhd D_1,\ \epsilon_1 : D_1 \lhd D_2,\ \ldots\ ,\epsilon_n : D_n \lhd D_{n+1},\ \ldots$ gilt:

1. $(\lambda x.\bot_D) : \texttt{void} \lhd D$
2. Es gibt einen Bereich D_∞, der obere Schranke aller D_n ist, d.h. für alle n gibt es $\rho_n : D_n \lhd D_\infty$.
3. D_∞ ist die kleinste obere Schranke (bis auf Isomorphie) der D_n, d.h. für jede weitere obere Schranke E gibt es $\theta : D_\infty \lhd E$.

Beweis: Siehe [Paulson 87, S. 99-104].

In obigem Sinne entspricht also die Menge der Bereiche mit $\lhd$ einer PCPO. Die gewünschte Lösung der Bereichsgleichung $X \cong FX$ mit dem stetigen Funktional F ist demnach quasi der kleinste Fixpunkt von F. Nachdem allgemein für PCPOs der kleinste Fixpunkt eines Funktionals identisch mit dem kleinsten Präfixpunkt des Funktionals ist, werden im folgenden Entsprechungen zu *Präfixpunkten* betrachtet.

3.2 Kategorientheoretische Lösung

Der im vorigen Abschnitt motivierte Begriff der Lösung einer Bereichsgleichung wird nun auf kategorientheoretische Konzepte abstrahiert, was im Gegensatz zur Deutung als kleinstem Fixpunkt ein gesichertes Vorgehen darstellt. Das Universum der Bereiche wird zu einer CPO-Kategorie $\mathcal{C}$ (mit speziellen Eigenschaften, siehe unten), das stetige Bereichsfunktional F zu einem lokal stetigen Endofunktor auf $\mathcal{C}$, Präfixpunkte bzgl. F werden zu F-Algebren (mit einer Zusatzbedingung) bzgl. des Funktors F, und der kleinste Präfixpunkt wird zur initialen F-Algebra in der Kategorie der F-Algebren. (Eine alternative kategorientheoretische Konstruktion der Lösung der Bereichsgleichung, die sogar den stetigen Funktionenraum behandeln kann, ist in [Paulson 87, Section 4.7] beschrieben. Diese hat jedoch den Nachteil, daß ihre Axiomatisierung in LCF oder einem anderen logischen System wie z.B. HOLCF nicht motiviert, geschweige denn deren Korrektheit bewiesen wird.)

3.2.1 Konstruktion der Lösung

Hier seien zunächst der Übersichtlichkeit halber die (Standard-) Definitionen für F-Algebren etc. wiederholt, wie sie auch in [Regensburger 94, S. 231-237] zu finden sind.

Definition 3.2 *F-Algebra, F-Homomorphismus, F-Alg*

Sei $F: \mathcal{C} \to \mathcal{C}$ ein Endofunktor auf einer Kategorie $\mathcal{C}$ und A und B Objekte aus $Obj(\mathcal{C})$. Eine F-**Algebra** (bzgl. dem Funktor F) ist ein Paar (A, θ) mit einem Morphismus $\theta: FA \to A$. Ein F-**Homomorphismus** zwischen F-Algebren (A, θ) und (B, φ) ist ein Morphismus $h: A \to B$, so daß $h \circ \theta = \varphi \circ Fh$. Graphisch bedeutet dies, daß folgendes Diagramm kommutiert:

$$\begin{tikzcd} FA \arrow[r, "\theta"] \arrow[d, "Fh"'] & A \arrow[d, "h"] \\ FB \arrow[r, "\varphi"'] & B \end{tikzcd} \qquad h \circ \theta = \varphi \circ Fh$$

Abbildung 3.1: F-Algebren, F-Homomorphismus

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.

Definition 3.3 *Strikte O-Kategorien*

Eine Kategorie $\mathcal{C}$ heißt **strikte O-Kategorie**, wenn folgende Bedingungen erfüllt sind:

so-cat1: Für alle Objekte $A, B \in Obj(\mathcal{C})$ trägt die Kollektion $\mathcal{C}(A, B)$ 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[1] ist, und zweitens existiert in jeder Kollektion $\mathcal{C}(A, B)$ ein kleinstes Element $\perp_{A,B}$ bezüglich $\sqsubseteq$.

so-cat2: Die Komosition $\circ$ von Morphismen ist stetig in beiden Argumenten bezüglich ω-Ketten.

so-cat3: Die Komposition $\circ$ von Morphismen ist strikt in beiden Argumenten, d.h.

$$\perp_{B,C} \circ f = \perp_{A,C} \quad \text{und} \quad g \circ \perp_{A,B} = \perp_{A,C}$$

für Morphismen $f: A \to B$ und $g: B \to C$.

[1]ω-Ketten seien auch hier so definiert, daß sie stets nicht-leer sind.

Definition 3.4 *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

$$(\epsilon_n : D_n \rightarrow D_{n+1}, \pi_n : D_{n+1} \rightarrow D_n)_{n\in\omega}$$

gibt es ein Objekt $D \in \mathit{Obj}(\mathcal{C})$ und eine Folge von EP-Paaren

$$(\iota_n : D_n \rightarrow D, \rho_n : D \rightarrow D_n)_{n\in\omega}$$

so daß gilt:

1. $\iota_{n+1} \circ \epsilon_n = \iota_n$ und $\pi_n \circ \rho_{n+1} = \rho_n$
2. $\bigsqcup_{n\in\omega} \iota_n \circ \rho_n = id_D$

Definition 3.5 *Lokal monotone, lokal stetige Funktoren*

Seien $\mathcal{C}$ und $\mathcal{D}$ strikte O-Kategorien. Ein Funktor $F : \mathcal{C} \rightarrow \mathcal{D}$ heißt **lokal monoton**, wenn für alle Morphismen $f, g : A \rightarrow B$ gilt:

$$f \sqsubseteq g \Longrightarrow Ff \sqsubseteq Fg$$

Ein lokal monotoner Funktor $F : \mathcal{C} \rightarrow \mathcal{D}$ heißt **lokal stetig**, wenn für alle ω-Ketten $(f_n)_{n\in\omega}$ von Morphismen gilt:

$$F(\bigsqcup_{n\in\omega} f_n) = \bigsqcup_{n\in\omega} Ff_n$$

Nun kann gezeigt werden, wie man die initiale F-Algebra als Lösung der Bereichsgleichung gewinnt, nämlich ihre Existenz und eindeutige Charakterisierung.

Mit Hilfe der gerade erwähnten Begriffe läßt sich die globale Charakterisierung der Lösung (per initialer F-Algebra) auf eine lokale zurückführen. Dies ist für die Axiomatisierbarkeit in HOLCF wesentlich, weil dort die Menge der F-Algebren syntaktisch nicht ausgedrückt werden kann, wohl aber die konkrete Anwendung des Funktors F auf Objekte und Morphismen.

Theorem 3.2 *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 gilt:

Die F-Algebra (A, θ) ist initial in der Kategorie F-Alg, wenn der Morphismus θ ein Isomorphismus ist, und die Identität id_A der kleinste F-Endomorphismus zwischen (A, θ) und sich selbst ist.

In Zeichen bedeutet dies:[2]

$$\begin{aligned} \theta \circ \theta^{-1} &= id_A \\ \theta^{-1} \circ \theta &= id_{FA} \\ id_A &= \mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1}) \end{aligned}$$

Beweis: Siehe [Regensburger 94, S. 234-237].

Daß eine solche F-Algebra existiert, folgt abschließend aus

Theorem 3.3 *Existenz von initialen F-Algebren*

Sei F ein lokal stetiger Endofunktor auf einer CPO-Kategorie $\mathcal{C}$. Dann gibt es eine F-Algebra (D, θ), so daß θ ein Isomorphismus und die Identität id_D der kleinste F-Endomorphismus zwischen (D, θ) und sich selbst ist.

Beweis: Siehe [Regensburger 94, S. 239-242].

3.2.2 Motivation der Lösung

Die soeben gewonnene kategorientheoretische Lösung der Gleichung $X \cong FX$ kann man als kleinsten Präfixpunkt von F (im oben genannten Sinn) interpretieren, wie folgt:

Eine F-Algebra mit einer Zusatzbedingung läßt sich als Präfixpunkt verstehen:

Definition 3.6 *Präfix-F-Algebra*

Eine **Präfix-F-Algebra** (A, θ) ist eine F-Algebra (A, θ), für die $\theta : FA \lhd A$ gilt. (Die oben gegebene Definition der EP-Paare von stetigen Funktionen gelte analog, auf Morphismen übertragen.)

Damit ist insbesondere die initiale F-Algebra (A, θ) eine Präfix-F-Algebra, denn es gilt folgendes Theorem:

Theorem 3.4 *Analogie zum Präfixpunkt*

Wenn die F-Algebra (A, θ) initial in F-Alg ist, dann ist θ ein Isomorphismus, d.h. es gibt ein $\bar{\theta}\colon A \to FA$ mit $\theta \circ \bar{\theta} = id_A$ und $\bar{\theta} \circ \theta = id_{FA}$. Insbesondere gilt $\theta : FA \lhd A$.

Der Beweis ist in [Regensburger 94, S. 231-232] dargestellt.

Bei einer auf obigem Weg gewonnen initialen F-Algebra (A, θ) kann der Bereich A sogar als *kleinster* Präfixpunkt von F aufgefaßt werden, denn:

Theorem 3.5 *Analogie zum kleinsten Präfixpunkt*

Sei F ein lokal stetiger Endofunktor auf einer strikten O-Kategorie $\mathcal{C}$. Sei weiterhin (A, ϵ_A) eine F-Algebra, wobei ϵ_A ein Isomorphismus ist (mit Umkehrung π_A), und $id_A = \mathit{fix}(\lambda h.\ \epsilon_A \circ Fh \circ \pi_A)$. Dann gilt:

(A, ϵ_A) ist eine initiale F-Algebra und eine Präfix-F-Algebra,
und zu jeder anderen Präfix-F-Algebra (B, ϵ_B) gibt es ein $h : A \lhd B$.

[2] $\mathit{fix}(\lambda f.\ Ff)$ ist eine Kurzschreibweise für $\bigsqcup_{n\in\omega} f_n$, wobei $f_0 := \perp$ und $f_{n+1} := Ff_n$

Beweis: Nach Theorem 3.2 ist (A, ϵ_A) initial in der Kategorie F-Alg, und weiter nach Theorem 3.4 eine Präfix-F-Algebra.
Sei nun (B, ϵ_B) eine beliebige weitere Präfix-F-Algebra. Hierfür wird ein EP-Paar (h,g) mit $h : A \lhd B$ konstruiert, nach folgender Abbildung:

$$\begin{tikzcd}
FA \arrow[r, shift left, "\epsilon_A"] \arrow[d, shift left, "Fh"] & A \arrow[l, shift left, "\pi_A"] \arrow[d, shift left, "h"] \\
FB \arrow[r, shift left, "\epsilon_B"] \arrow[u, shift left, "Fg"] & B \arrow[l, shift left, "\pi_B"] \arrow[u, shift left, "g"]
\end{tikzcd}$$

Abbildung 3.2: Konstruktion eines EP-Paars

Sei dazu $h_0 := \bot$; $h_{n+1} := \epsilon_B \circ Fh_n \circ \pi_A$; $h := \bigsqcup_{n\in\omega} h_n$. h ist wohldefiniert, weil die h_n eine ω-Kette bilden (leichte Induktion).
Analog sei $g_0 := \bot$; $g_{n+1} := \epsilon_A \circ Fg_n \circ \pi_B$; $g := \bigsqcup_{n\in\omega} g_n$.

Damit gilt: $g \circ h =$

$$\begin{aligned}
&= (\bigsqcup_{n\in\omega} g_n) \circ (\bigsqcup_{n\in\omega} h_n) = (\bigsqcup_{n\in\omega} g_{n+1}) \circ (\bigsqcup_{n\in\omega} h_{n+1}) = \bigsqcup_{n\in\omega} (g_{n+1} \circ h_{n+1}) \\
&= \bigsqcup_{n\in\omega} ((\epsilon_A \circ Fg_n \circ \pi_B) \circ (\epsilon_B \circ Fh_n \circ \pi_A)) \\
&= \bigsqcup_{n\in\omega} (\epsilon_A \circ Fg_n \circ (\pi_B \circ \epsilon_B) \circ Fh_n \circ \pi_A) \\
&= \bigsqcup_{n\in\omega} (\epsilon_A \circ Fg_n \circ Fh_n \circ \pi_A) = \bigsqcup_{n\in\omega} (\epsilon_A \circ F(g_n \circ h_n) \circ \pi_A) \\
&= \bigsqcup_{n\in\omega} (\epsilon_A \circ Ff_n \circ \pi_A) = \bigsqcup_{n\in\omega} f_{n+1} = \bigsqcup_{n\in\omega} f_n \\
&= \mathit{fix}(\lambda f.\ \epsilon_A \circ Ff \circ \pi_A) = id_A\ ,
\end{aligned}$$

wobei $f_n := g_n \circ h_n$, also

$$\begin{aligned}
f_0 = \bot \text{ und } f_{n+1} &= (\epsilon_A \circ Fg_n \circ \pi_B) \circ (\epsilon_B \circ Fh_n \circ \pi_A) = \\
&= (\epsilon_A \circ Fg_n \circ Fh_n \circ \pi_A) = (\epsilon_A \circ Ff_n \circ \pi_A)\ .
\end{aligned}$$

Andererseits gilt: $h \circ g =$

$$\begin{aligned}
&= (\bigsqcup_{n\in\omega} h_n) \circ (\bigsqcup_{n\in\omega} g_n) = \ldots \quad \text{(analog)} \ = \bigsqcup_{n\in\omega} (\epsilon_B \circ F(h_n \circ g_n) \circ \pi_B) \\
&= \bigsqcup_{n\in\omega} (\epsilon_B \circ Ff_n \circ \pi_B) = \bigsqcup_{n\in\omega} f_{n+1} = \bigsqcup_{n\in\omega} f_n \sqsubseteq id_B\ ,
\end{aligned}$$

wobei $f_n := h_n \circ g_n$, und somit $\bigsqcup_{n\in\omega} f_n \sqsubseteq id_B$, weil

(per Induktion) $\forall n.\ f_n \sqsubseteq id_B$ gilt:

$f_0 = \bot \sqsubseteq id_B$, und

$$\begin{aligned}
f_n \sqsubseteq id_B \Longrightarrow f_{n+1} &= \epsilon_B \circ Ff_n \circ \pi_B \sqsubseteq \epsilon_B \circ F(id_B) \circ \pi_B = \\
&= \epsilon_B \circ id_{FB} \circ \pi_B \sqsubseteq id_B\ .
\end{aligned}$$

Zusammenfassend gilt also: Wenn F ein lokal stetiger Endofunktor $F : \mathcal{C} \to \mathcal{C}$ auf einer CPO-Kategorie $\mathcal{C}$ ist, dann existiert die (initiale, dem kleinsten Fixpunkt entsprechende) Lösung A der Gleichung $X \cong FX$, und diese ist eindeutig durch

$$\begin{aligned}
&\theta \circ \theta^{-1} = id_A \\
&\theta^{-1} \circ \theta = id_{FA} \\
&id_A = \mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1})
\end{aligned}$$

charakterisiert.

3.3 Anwendung auf simultane Gleichungen

Wie in [Regensburger 94, Abschnitt 5.2] sei im folgenden $Th = (\Sigma, Ax, KAx)$ eine HOLCF-Theorie, welche die Theorie `ccc1` als Teiltheorie enthält und in der durch eine Gleichung $X \cong FX$ ein Bereich definiert werden soll, und sei $\mathcal{M} = (\mathcal{TM}, \mathcal{C})$ mit $\mathcal{TM} = (\mathcal{K}, \mathcal{TC})$ ein beliebiges, durch konservative Modellerweiterung entstandenes, Modell für Th. Aus $\mathcal{M}$ wird nach [Regensburger 94, Definitionen 5.7 - 5.9 und Theoreme 5.10 - 5.11] eine CPO-Kategorie $\mathcal{PCPO}$ induziert. Falls die Gleichung $X \cong FX$ eine einfache Gleichung ist, läßt sich die Theorie aus dem obigen Abschnitt direkt anwenden und in HOLCF-Axiome übertragen, denn dann ist erstens $\mathcal{PCPO}$ eine geeignete Kategorie, und zweitens beschreibt der sich aus der Bereichsgleichung ergebende Funktorterm FT nach [Regensburger 94, Definitionen 5.10 - 5.11 und Theoreme 5.12 - 5.14] einen lokal stetigen Endofunktor F auf $\mathcal{PCPO}$.

Dieser Abschnitt behandelt daher lediglich die Besonderheiten, die bei simultanen Gleichungen (GL) :

$$\begin{aligned}
&X_1 \cong F_1(X_1, (\ldots, X_n)\ldots), \\
&\vdots, \\
&X_n \cong F_n(X_1, (\ldots, X_n)\ldots)
\end{aligned}$$

auftreten.

3.3.1 Theorie

Der eine Aspekt der Besonderheiten ist die Anwendung der obigen Ergebnisse auf theoretischer Seite: Die gerade genannte simultane Gleichung kann in eine (äquivalente) *einfache* Gleichung in einer geeigneten Produktkategorie umgewandelt werden, und zwar durch iterative Anwendung der folgenden Prozedur für $i = n, \ldots, 2$:

Die letzten beiden Gleichungen $X_{i-1} \cong F_{i-1}(X_1, (\ldots, X_n)\ldots)$ und $X_i \cong F_i(X_1, (\ldots, X_n)\ldots)$ in den Kategorien $\mathcal{C}_{i-1}$ bzw. $\mathcal{C}_i$ werden zu einer Gleichung $(X_{i-1}, X_i) \cong (F_{i-1}, F_i)(X_1, (\ldots, X_n)\ldots)$ in der Produktkategorie $\mathcal{C}_{i-1} \times \mathcal{C}_i$ (mit den Funktoren P und Q als Projektionen auf die erste bzw. zweite Komponente) verschmolzen. Wenn $\mathcal{C}_{i-1}$ und $\mathcal{C}_i$ CPO-Kategorien sind, dann auch $\mathcal{C}_{i-1} \times \mathcal{C}_i$, und wenn F_{i-1} und F_i lokal stetige Funktoren sind, dann auch (F_{i-1}, F_i)[3], wie

[3] Das Funktortupel (F, G) sei durch $(F, G)x := (Fx, Gx)$ definiert

man durch Nachrechnen mit der partiellen Ordnung auf Morphismenpaaren $(f_1, f_2) \sqsubseteq (g_1, g_2) :\Longleftrightarrow f_1 \sqsubseteq g_1 \wedge f_2 \sqsubseteq g_2$ bestätigen kann. Desweiteren ändert sich die Lösungsmenge der Gleichung nicht, was die komponentenweise Betrachtung zeigt.

Somit gilt:

Theorem 3.6 *Produkt-Gleichung*

Seien $\mathcal{C}_1, \ldots, \mathcal{C}_n$ CPO-Kategorien und für $i = 1, \ldots, n$: F_i ein lokal stetiger Funktor $\mathcal{C}_1 \times (\ldots \times \mathcal{C}_n) \rightarrow \mathcal{C}_i$. Sei ferner (GL) eine simultane Bereichsgleichung wie oben. Dann hat die einzelne Gleichung $X \cong FX$ mit $X = (X_1, (\ldots, X_n)\ldots)$ und $F = (F_1, (\ldots, F_n)\ldots)$ in der CPO-Produktkategorie $\mathcal{C}_1 \times (\ldots \times \mathcal{C}_n)$ und mit dem lokal-stetigen Endofunktor F dieselben Lösungen wie (GL).

Zur Darstellung der Funktoren F_i ist noch die Menge der Funktorterme und ihre Interpretation aus [Regensburger 94, Definitionen 5.10 - 5.11] um Projektionen und (allgemeine) Komposition als Funktorterme zu erweitern:

Definition 3.7 *Funktorterme*

Seien $\tau, \tau_1, \tau_2, \ldots$ Typterme und $\tau\tau_n$ ein n-Tupel (für $n \geq 1$) von Typtermen aus $T_{\Omega,pcpo}$, deren Typvariablen ausschließlich Variablen der Menge Ξ_{pcpo} sind, und seien $t_1 \in \mathbf{T}_{\Sigma,\tau_1 \rightarrow \tau_2}$ und $t_2 \in \mathbf{T}_{\Sigma,\tau_3 \rightarrow \tau_4}$ Terme und $tt_n \in \mathbf{T}_{\Sigma,\tau_5 \rightarrow \tau_6} \times \ldots \times \mathbf{T}_{\Sigma,\tau_{2n+3} \rightarrow \tau_{2n+4}}$ ein n-Tupel von Termen für Operationen mit $TV(t_1) \cup TV(t_2) \cup TV(tt_n) \subseteq \Xi_{pcpo}$.

Bezeichne weiter ν eine Typvariablenbelegung, und $(\llbracket FT^O \rrbracket_\nu, \llbracket FT^M \rrbracket_\nu)$ den Objekt- und Morphismusanteil der Interpretation eines Funktorterms FT bezüglich ν im Modell $\mathcal{M}$.

Dann ist die Menge der Funktorterme über der Theorie Th durch folgende induktive Definition gegeben:

$\vdots$ *(wie bisher in [Regensburger 94, Definition 5.10 und 5.11])*

Projektion auf die erste Komponente: Der Term $P1$ ist ein $(n+1)$-stelliger Funktorterm. Die Menge der in $P1$ vorkommenden Typvariablen ist definiert als:

$$TV(P1) = \emptyset$$

Die Anwendung auf ein $(n+1)$-Tupel von Typtermen ist definiert als:

$$P1(\tau_1, \tau\tau_n) = \tau_1$$

Die Anwendung auf ein $(n+1)$-Tupel von Operationen ist definiert als:

$$P1(t_1, tt_n) = t_1$$

Die Semantik von $P1$ ist definiert als:

$$\begin{aligned} \llbracket P1^O \rrbracket_\nu &= P \\ \llbracket P1^M \rrbracket_\nu &= P \end{aligned}$$

Projektion auf zweite Komponente: Der Term $P2$ ist ein $(n+1)$-stelliger Funktorterm, und seine Anwendung liefert ein n-Tupel. Die Menge der in $P2$ vorkommenden Typvariablen ist definiert als:

$$TV(P2) = \emptyset$$

Die Anwendung auf ein $(n+1)$-Tupel von Typtermen ist definiert als:

$$P2(\tau_1, \tau\tau_n) = \tau\tau_n$$

Die Anwendung auf ein $(n+1)$-Tupel von Operationen ist definiert als:

$$P2(t_1, tt_n) = tt_n$$

Die Semantik von $P2$ ist definiert als:

$$\begin{array}{rcl} [\![P2^O]\!]_\nu & = & Q \\ [\![P2^M]\!]_\nu & = & Q \end{array}$$

Komposition: Ist FT_2 ein n-stelliger Funktorterm, dessen Anwendung ein m-Tupel liefert, und ist FT_1 ein m-stelliger Funktorterm, dann ist $FT1 \circ FT2$ ein n-stelliger Funktorterm. Die Menge der in $FT1 \circ FT2$ vorkommenden Typvariablen ist definiert als:

$$TV(FT1 \circ FT2) = TV(FT1) \cup TV(FT_2)$$

Die Anwendung auf ein n-Tupel von Typtermen ist definiert als:

$$(FT1 \circ FT2)\ \tau\tau_n = FT2(FT1\ \tau\tau_n)$$

Die Anwendung auf ein n-Tupel von Operationen ist definiert als:

$$(FT1 \circ FT2)\ tt_n = FT2(FT1\ tt_n)$$

Die Semantik von $FT1 \circ FT2$ ist definiert als:

$$\begin{array}{rcl} [\![(FT1 \circ FT2)^O]\!]_\nu & : & X \mapsto ([\![FT_1^O]\!]_\nu)(([\![FT_2^O]\!]_\nu)(X)) \\ [\![(FT1 \circ FT2)^M]\!]_\nu & : & m \mapsto ([\![FT_1^M]\!]_\nu)(([\![FT_2^M]\!]_\nu)(m)) \end{array}$$

Die Semantik der Projektionen und der Komposition ist wohldefiniert, weil sie gerade die Projektionsfunktoren P und Q bzw. die Funktorkomposition darstellt, und offensichtlich sind jeweils die syntaktische und die semantische Definitionsvariante der neu eingeführten Funktorterme miteinander verträglich. Desweiteren beschreiben diese Funktorterme lokal stetige Funktoren bzgl. strikten O-Kategorien (mit der oben erwähnten partiellen Ordnung auf Morphismenpaaren), wie leicht nachzurechnen ist.

3.3.2 Axiomatisierung in HOLCF

Der andere Aspekt, nämlich die Axiomatisierung in HOLCF, bedarf eines Tricks, weil in HOLCF zwar die Kategorie $\mathcal{PCPO}$ (mit dem Universum $\texttt{pcpo}^{\mathcal{TM}}$ als Objektanteil) formulierbar ist, nicht jedoch Produktkategorien. Aber man kann die Produkte von Kategorien $\mathcal{PCPO}$ in den Axiomen teilweise vermeiden und ansonsten in geeignete kartesische Produkte *innerhalb* der Kategorie $\mathcal{PCPO}$ überführen, wie im folgenden gezeigt.

In der Axiomatisierung des Bereichsisomorphismus lassen sich die Produktkategorien schrittweise entfernen, denn:

Lemma 3.7 *Alternative Isomorphie-Eigenschaft*

Seien $\mathcal{C}$ und $\mathcal{D}$ Kategorien, F ein Bifunktor $\mathcal{C} \times \mathcal{D} \to \mathcal{C} \times \mathcal{D}$ und $((A, B), (\theta_1, \theta_2))$ eine F-Algebra, wobei (θ_1, θ_2) ein Isomorphismus ist und $F(A, B) = (A', B')$. Dann sind die beiden folgenden Aussagen gleichwertig:

1. $(\theta_1, \theta_2) \circ (\theta_1, \theta_2)^{-1} = id_{(A,B)} \wedge$
   $(\theta_1, \theta_2)^{-1} \circ (\theta_1, \theta_2) = id_{(A',B')}\ (= id_{F(A,B)})$
2. $\theta_1 \circ \theta_1^{-1} = id_A \ \wedge \ \theta_2 \circ \theta_2^{-1} = id_B \ \wedge$
   $\theta_1^{-1} \circ \theta_1 = id_{A'} \ \wedge \ \theta_2^{-1} \circ \theta_2 = id_{B'}$

Beweis: $(\theta_1 \circ \theta_1^{-1}, \theta_2 \circ \theta_2^{-1}) = (\theta_1, \theta_2) \circ (\theta_1, \theta_2)^{-1} = id_{(A,B)} = (id_A, id_B)$
$\iff \theta_1 \circ \theta_1^{-1} = id_A \ \wedge \ \theta_2 \circ \theta_2^{-1} = id_B$,
wobei $\theta_1^{-1} := P((\theta_1, \theta_2)^{-1})$ und $\theta_2^{-1} := Q((\theta_1, \theta_2)^{-1})$.
Der zweite Teil folgt analog.

Die Axiomatisierung des kleinsten F-Endomorphismus kann nicht so weit aufgebrochen werden, denn hier ist die Verschränkung der Bereichsanteile wesentlich. Aber immerhin ist eine Transformation in eine Formulierung mithilfe des kartesischen Produkts $\times$ (hier von stetigen Funktionen) in HOLCF möglich. Dazu bedarf es des folgenden Isomorphismus:

Definition 3.8 *Produkt-Isomorphismus*

Seien $\mathcal{C}_1, \ldots, \mathcal{C}_n$ strikte O-Kategorien und η eine Termvariablenbelegung, wobei $[\![t]\!]_\eta$ die Semantik des Terms t bezüglich η im Modell $\mathcal{M}$ darstellt. (Auf eine vollständige Typisierung der Terme wird hier aus Einfachheitsgründen verzichtet.) Dann wird für alle $A_1, A_1' \in \mathit{Obj}(\mathcal{C}_1)$, $\ldots$, $A_n, A_n' \in \mathit{Obj}(\mathcal{C}_n)$ definiert:

$\lfloor\_\rfloor$: $\mathcal{C}_1 \times (\ldots \times \mathcal{C}_n)((A_1, (\ldots, A_n)\ldots), (A_1', (\ldots, A_n')\ldots)) \to$
$(A_1 \to A_1') \times \ldots \times (A_n \to A_n')$,
$\lfloor (f_1, (\ldots, f_n)\ldots) \rfloor := [\![\texttt{<f1, ..., fn>}]\!]_{\eta[f_1/\texttt{f1},\ldots,f_n/\texttt{fn}]}$;

$\lceil\_\rceil$: $(A_1 \to A_1') \times \ldots \times (A_n \to A_n') \to$
$\mathcal{C}_1 \times (\ldots \times \mathcal{C}_n)((A_1, (\ldots, A_n)\ldots), (A_1', (\ldots, A_n')\ldots))$,
$\lceil f \rceil := ([\![\texttt{cfst`f}]\!]_{\eta[f/\texttt{f}]}, (\ldots, [\![\texttt{csnd`(...(csnd`f)...)}]\!]_{\eta[f/\texttt{f}]})\ldots)$, also
$\lceil [\![\texttt{<f1, ..., fn>}]\!]_{\eta[f_1/\texttt{f1},\ldots,f_n/\texttt{fn}]} \rceil = (f_1, (\ldots, f_n)\ldots)$;

$proj_i$: `((`A_1 `->` A'_1`) * ... * (`A_n `->` A'_n`)) -> (`A_i `->` A'_i`)`
die i-te Projektion eines n-fachen kartesischen Produkts $\times$ in HOLCF (mittels geeigneter Komposition von $[\![\texttt{cfst}]\!]_\eta$ und $[\![\texttt{csnd}]\!]_\eta$);

P_i: $\mathcal{C}_1 \times (\ldots \times \mathcal{C}_n)((A_1, (\ldots, A_n)\ldots), (A'_1, (\ldots, A'_n)\ldots)) \to \mathcal{C}_i(A_i, A'_i)$
die i-te Projektion der n-fachen Produktkategorie $\mathcal{C}_1 \times (\ldots \times \mathcal{C}_n)$ (mittels geeigneter Komposition von P und Q).

Trivialerweise ist $\lfloor\_\rfloor$ eine Bijektion mit Umkehrung $\lceil\_\rceil$,
und $\lfloor\_\rfloor$ ist strikt und stetig.

Lemma 3.8 *Alternative Fixpunkteigenschaft*

Seien $\mathcal{C}_1, \ldots, \mathcal{C}_n$ strikte O-Kategorien, $F = (F_1, (\ldots, F_n)\ldots)$ ein lokal stetiger Endofunktor auf $\mathcal{C}_1 \times (\ldots \times \mathcal{C}_n)$ und (A, θ) eine F-Algebra, wobei $\theta = (\theta_1, (\ldots, \theta_n)\ldots)$ ein Isomorphismus ist und $A = (A_1, (\ldots, A_n)\ldots)$. Dann sind folgende beide Aussagen gleichwertig:

1. $id_A = \mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1})$
2. Für alle $i = 1, \ldots, n$: $id_{A_i} =$
   $proj_i(\texttt{fix(}\lambda\texttt{f. <}\theta_1\ \texttt{oo}\ F_1\lceil\texttt{f}\rceil\ \texttt{oo}\ \theta_1^{-1}\texttt{,...,}\theta_n\ \texttt{oo}\ F_n\lceil\texttt{f}\rceil\ \texttt{oo}\ \theta_n^{-1}\texttt{>))}$

(Hier und weiter unten tritt eine Mischung aus abstrakter semantischer Notation und Isabelle-Syntax auf. Genaugenommen müßte man wie oben auf die Isabelle-Teilterme jeweils einen Interpretationsoperator (mit geeigneter Termsubstitution) anwenden, doch würde dies die Notation sehr unübersichlich machen.)

Beweis: $(id_{A_1}, (\ldots, id_{A_n})) = id_A = \mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1})$
$= \ (P_1(\mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1})), (\ldots, P_n(\mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1}))))$
$\iff$ Für alle $i = 1, \ldots, n$: $id_{A_i} = P_i(\mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1}))$
$\iff$ Für alle $i = 1, \ldots, n$: $id_{A_i} = P_i(\lceil\lfloor \mathit{fix}(\lambda h.\ \theta \circ Fh \circ \theta^{-1})\rfloor\rceil) \ =$
$P_i(\lceil\lfloor \mathit{fix}(\lambda h.\ (\theta_1, (\ldots, \theta_n)\ldots) \circ (P_1(Fh), (\ldots, P_n(Fh))\ldots) \circ$
$(\theta_1^{-1}, (\ldots, \theta_n^{-1})\ldots))\rfloor\rceil) \ =$
$P_i(\lceil\lfloor \mathit{fix}(\lambda h.\ (\theta_1 \circ P_1(Fh) \circ \theta_1^{-1}, (\ldots, \theta_n \circ P_n(Fh) \circ \theta_n^{-1})\ldots))\rfloor\rceil) \ =$
$proj_i(\lfloor \mathit{fix}(\lambda h.\ (\theta_1 \circ F_1 h \circ \theta_1^{-1}, (\ldots, \theta_n \circ F_n h \circ \theta_n^{-1})\ldots))\rfloor) \overset{(*)}{=}$
$proj_i(\texttt{fix(}\lambda\texttt{f.}\ \lfloor(\theta_1 \circ F_1\lceil\texttt{f}\rceil \circ \theta_1^{-1}, (\ldots, \theta_n \circ F_n\lceil\texttt{f}\rceil \circ \theta_n^{-1})\ldots)\rfloor\texttt{))} =$
$proj_i(\texttt{fix(}\lambda\texttt{f. <}\theta_1\ \texttt{oo}\ F_1\lceil\texttt{f}\rceil\ \texttt{oo}\ \theta_1^{-1}\texttt{,...,}\theta_n\ \texttt{oo}\ F_n\lceil\texttt{f}\rceil\ \texttt{oo}\ \theta_n^{-1}\texttt{>))}$.

$(*)$ $\lfloor \mathit{fix}(\lambda h.\ Gh)\rfloor = \lfloor \bigsqcup_{n\in\omega} h_n\rfloor = \bigsqcup\texttt{n.}\ \lfloor h_n\rfloor = \bigsqcup\texttt{n.}\ f_n =$
$\texttt{fix(}\lambda\texttt{f.}\ \lfloor G\lceil\texttt{f}\rceil\rfloor\texttt{)}$,
wobei $h_0 := (\bot_1,\ (\ldots, \bot_n)\ldots)$; $h_{n+1} := Gh_n$;
$f_n := \lfloor h_n\rfloor$,
somit $f_0 \ = \lfloor h_0\rfloor = \bot$
und $f_{n+1} = \lfloor Gh_n\rfloor = \lfloor G\lceil\lfloor h_n\rfloor\rceil\rfloor = \lfloor G\lceil f_n\rceil\rfloor$.

Insgesamt ergibt sich also:

Theorem 3.9 *Lösung simultaner Bereichsgleichungen*

Sei Th eine HOLCF-Theorie, welche die Theorie `ccc1` als Teiltheorie enthält, und

$$\begin{array}{l} X_1 \cong F_1(X_1,(\ldots,X_n)\ldots), \\ \vdots, \\ X_n \cong F_n(X_1,(\ldots,X_n)\ldots) \end{array}$$

eine simultane Bereichsgleichung bezüglich Th, wobei die F_i Funktorterme gemäß Definition 3.7 sind. Mit $X := (X_1,(\ldots,X_n)\ldots)$ und $F := (F_1,(\ldots,F_n)\ldots)$ existiert die (initiale) Lösung der Gleichung $X \cong FX$, und sie kann durch die Axiome

$$\begin{array}{l} \theta_1 \circ \theta_1^{-1} = id_{X_1} \;\wedge\; \theta_1^{-1} \circ \theta_1 = id_{P_1(FX)} \;\wedge \\ \vdots \;\wedge \\ \theta_n \circ \theta_n^{-1} = id_{X_n} \;\wedge\; \theta_n^{-1} \circ \theta_n = id_{P_n(FX)} \;\wedge \\ \\ id_{A_1} = proj_1(\mathtt{fix(}\lambda\mathtt{f.\ <}\theta_1\ \mathtt{oo}\ F_1\lceil\mathtt{f}\rceil\ \mathtt{oo}\ \theta_1^{-1}\mathtt{,\ \ldots,} \\ \qquad\qquad\qquad\qquad\qquad \theta_n\ \mathtt{oo}\ F_n\lceil\mathtt{f}\rceil\ \mathtt{oo}\ \theta_n^{-1}\mathtt{>))} \;\wedge \\ \vdots \;\wedge \\ id_{A_n} = proj_n(\mathtt{fix(}\lambda\mathtt{f.\ <}\theta_1\ \mathtt{oo}\ F_1\lceil\mathtt{f}\rceil\ \mathtt{oo}\ \theta_1^{-1}\mathtt{,\ \ldots,} \\ \qquad\qquad\qquad\qquad\qquad \theta_n\ \mathtt{oo}\ F_n\lceil\mathtt{f}\rceil\ \mathtt{oo}\ \theta_n^{-1}\mathtt{>))} \end{array}$$

in HOLCF eindeutig charakterisiert werden.

Beweis: Statt die n Teilgleichungen simultan jeweils in der CPO-Kategorie $\mathcal{PCPO}$ zu lösen, wird die Gesamtgleichung (äquivalenterweise, nach Theorem 3.6) im Produkt dieser Kategorien gelöst. Die Theoreme 3.3 und 3.2 sichern nun die Existenz und liefern eine eindeutige Charakterisierung der Lösung. Die gewünschte für HOLCF geeignete Darstellung entsteht dann durch $(n-1)$-malige Anwendung von Lemma 3.7 und mit dem Lemma 3.8.

Kapitel 4

Struktur der Datentypspezifikationen

Dieses Kapitel beschreibt die beiden vom Datentyp-Paket unterstützten Formen von **Datentypspezifikationen**, nämlich zum einen rekursive Gleichungen ähnlich dem ML-Konstrukt `datatype`, siehe [Paulson 92], zum anderen Datentypgenerierungsregeln, wie sie in SPECTRUM [Broy et al 93b] verwendet werden.

4.1 Datentypgleichungen

Die **Datentypgleichungen**, also Bereichsgleichungen für einen Datentyp (im folgenden kurz: **Gleichungen**) dienen dazu, freie Datentypen über ihre Konstruktorstruktur zu definieren. Es ist an die `datatype`-Sektion von Isabelle/HOL [Paulson 94 III, Section 4.6] angelehnt; im Gegensatz zu dieser ermöglicht es aber nichtstrikte Konstruktoren und die Definition von Selektoren. Zudem sind *verschränkt* rekursive Datentypen erlaubt, und die entstehende Theorieerweiterung ist konservativ. Die Gleichungen haben folgende allgemeine Struktur:

$$
\begin{array}{llllll}
\texttt{domain} & typ_1 & \texttt{=} & con_{11} & arg_{111} \; \dots \; arg_{11k_{11}} & syn_{11} \\
 & & \texttt{|} & \vdots & & \\
 & & \texttt{|} & con_{1m_1} & arg_{1m_11} \; \dots \; arg_{1m_1k_{1m_1}} & syn_{1m_1}\texttt{,} \\
 & \vdots\texttt{,} & & & & \\
 & typ_n & \texttt{=} & con_{n1} & arg_{n11} \; \dots \; arg_{n1k_{n1}} & syn_{n1} \\
 & & \texttt{|} & \vdots & & \\
 & & \texttt{|} & con_{nm_n} & arg_{nm_n1} \; \dots \; arg_{nm_nk_{nm_n}} & syn_{nm_n} \\
\textsf{where} & typ_i & = & (tv_{i1}\texttt{,} \; \dots\texttt{,} \; tv_{il_i}\texttt{)} \; nam_i & & \\
 & arg_{ijk} & = & lazy_{ijk} \; sel_{ijk} \; \texttt{::} \; atyp_{ijk} & &
\end{array}
$$

Das (optionale) Attribut $lazy_{ijk}$ bestimmt, ob der Konstruktor con_{ij} im k-ten Argument nicht strikt ist.

Um die Konstruierbarkeit der Bereiche sicherzustellen, müssen die Gleichungen einer Reihe von (Kontext-) Bedingungen genügen:[1]

[1] Ein leerer Bereich (bei dem in einem typ_i alle seine Konstruktoren nicht-strikt rekursiv sind) ist kein Fehler. In einem solchen Fall erhält der Benutzer eine Warnung.

- Alle Typnamen nam_i sind neu und voneinander verschieden.
- Für alle i sind die Typvariablen $tv_{i1}, \ldots, tv_{il_i}$ voneinander verschieden.
- Für alle i in allen Typen $atyp_{ijk}$ sind alle auftretenden Typvariablen aus $tv_{i1}, \ldots, tv_{il_i}$.
- Falls ein Typausdruck $atyp_{ijk}$ nicht nur aus einer Typvariable besteht, ist er mit einem der Typausdrücke $typ_1, \ldots, typ_n$ identisch (Rekursion), oder sein Typkonstruktor ist bereits deklariert.
- Alle Konstruktornamen con_{ij} sind neu und voneinander verschieden.
- Alle Selektornamen sel_{ijk} sind neu und voneinander verschieden. (Ist ein Selektorname nicht gegeben, wird der generische Name nam_i`_sel_`j`_`k verwendet.)
- Alle Diskriminatornamen, welche durch Voranstellen von `is_` aus den Selektornamen sel_{ijk} gewonnen werden, sind neu.

Falls einer der Typen $atyp_{ijk}$ ein komplexer Typausdruck sein soll, muß für diesen zunächst mittels einer anderen Typspezifikation ein Synonym erklärt und dieses in $atyp_{ijk}$ verwendet werden. Damit ist indirekte Rekursion zwar nicht erlaubt, kann aber mittels verschränkter Rekursion simuliert werden, z.B. ist (sei `domain 'a dlist = dnil | dcons 'a ('a dlist)` bereits deklariert)

```
domain t = C (t dlist)  (* illegal ! *)
```

durch

```
domain t  = C t',
       t' = tnil | tcons t t'
```

zu ersetzen.

Die gegebene Konstruktor-Struktur erzwingt implizit eine spezielle Form von Gleichungen über den Bereichen, und zwar

$$\begin{aligned} typ_i = {} & ty_{i11} \otimes \ldots \otimes ty_{i1k_{i1}} \oplus \\ & \vdots \oplus \\ & ty_{im_i1} \otimes \ldots \otimes ty_{im_ik_{im_i}} \\ & \textsf{where } at_{ijk} \textsf{ is atomic} \\ & \qquad ty_{ijk} = at_{ijk} \textsf{ or } (at_{ijk})\mathtt{u} \end{aligned}$$

, d.h. stets sind die Summen außen und Liftings innen. Zwar lassen sich somit keine Bereichsgleichungen formulieren, bei denen etwa Produkte außen und Summen innen sind, aber die Distribution von Produkten über Summen liefert isomorphe Bereiche ($x \otimes (y \oplus z) \simeq (x \otimes y) \oplus (x \otimes z)$), so daß man wenigstens äquivalente Bereiche spezifizieren kann. Dagegen ergibt die Distribution von Liftings über Produkte und Summen ($(x \oplus y)\mathtt{u} \rightsquigarrow (x)\mathtt{u} \oplus (y)\mathtt{u}$ und $(x \otimes y)\mathtt{u} \rightsquigarrow (x)\mathtt{u} \otimes (y)\mathtt{u}$) nur annähernd isomorphe Bereiche mit etwas mehr Information in dem Fall, daß undefinierte Elemente aus den Komponenten x bzw. y in der Summe bzw. dem Produkt kombiniert werden.

4.2 Datentypgenerierungsregeln

Die Datentypgenerierungsregeln (im folgenen kurz: Gen-by-Regeln) dienen zur Vereinfachung der Definition von allgemeinen, nicht-freien Datentypen, und erzeugen lediglich ein Induktionsaxiom. Sie dürfen wie die Datentypgleichungen verschränkt rekursiv sein, haben aber eine einfachere Struktur, da sie sich auf bereits deklarierte Typen und Konstruktoren beziehen:

`generated` *finite*
    nam_1 `by` con_{11} | ... | con_{1m_1},
    $\vdots$,
    nam_n `by` con_{n1} | ... | con_{nm_n}

Das (optionale) Attribut *finite* bestimmt, ob der Datentyp nur endliche Elemente enthält, so daß in der Induktionsregel auf die Voraussetzung der Zulässigkeit (Admissibility) der betreffenden Prädikate verzichtet werden kann.

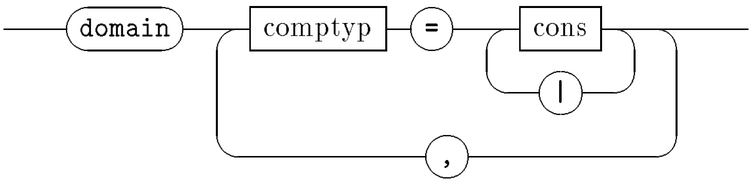
Damit ein Induktionsaxiom erzeugt werden kann, muß die Regel folgende Kontextbedingungen erfüllen:

- Alle Typnamen nam_i sind bereits deklariert und voneinander verschieden, und die entsprechenden Typen gehören der Klasse `pcpo` an.
- Alle Konstruktornamen con_{ij} sind bereits deklariert und voneinander verschieden.
- Für alle i sind die Resultattypen aller Konstruktoren con_{ij} identisch und tragen den Typnamen nam_i.
- In allen Argumenttypen aller Konstruktoren con_{ij} treten die Typnamen nam_1, ..., nam_n höchstens als äußerster Typkonstruktor auf (keine indirekte Rekursion).

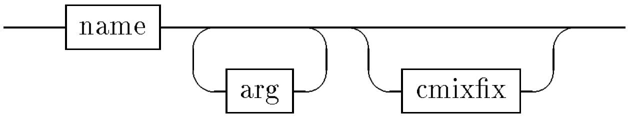
4.3 Konkrete Syntax

Die Syntax der Datentypspezifikationen ist in Abbildung 4.1 dargestellt, wobei *id*, *tid*, *nat*, *string*, *name*, *mixfix* und *infix* die gleichen lexikalischen Klassen wie in [Paulson 94 II, Appendix A] bezeichnen. Sie ist eine Erweiterung des Syntax-Vorschlags aus [Regensburger 95].

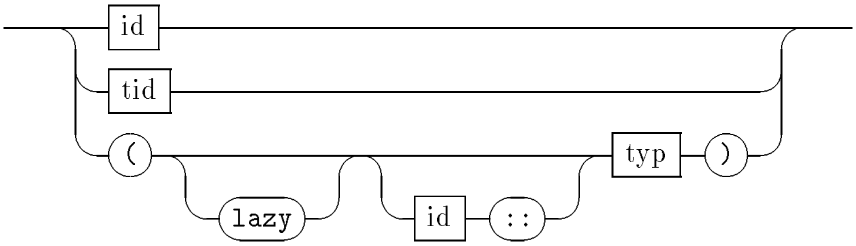
DomainDef



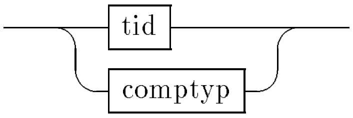
cons



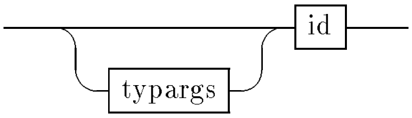
arg



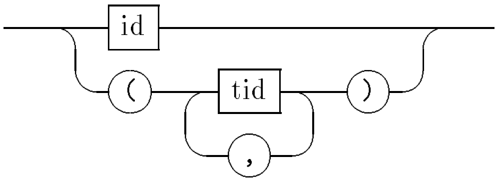
typ



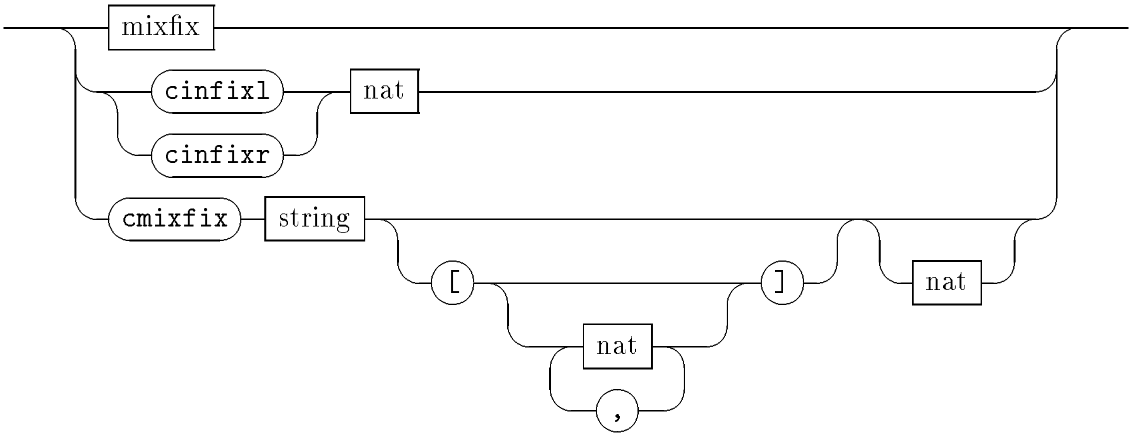
comptyp



typargs



cmixfix



GenByDef

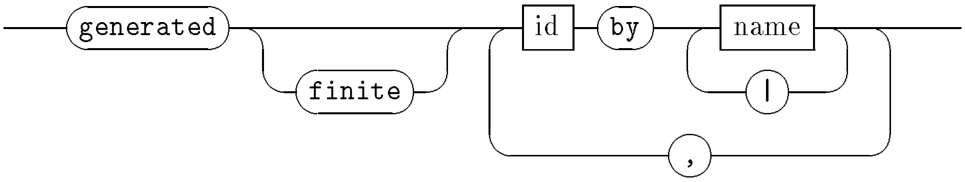


Abbildung 4.1: Syntax der Datentypspezifikationen

Besonders zu erwähnen ist die Möglichkeit, für Konstruktoren eine Infix- bzw. Mixfix-Syntax zu vereinbaren. Da es sich bei diesen um stetige Funktionen (mit dem Typkonstruktor $\rightarrow$) handelt, sollten hierfür die `cinfixl`-, `cinfixr`- und `cmixfix`- Varianten verwendet werden. Die ersteren beiden sind in [Mayr 95, Die Spezifikation von `ops`] beschrieben, während letztere im Zuge dieser Arbeit ergänzt worden ist. Auch in Gen-By-Regeln können Infix-Operatoren *nfx* als Konstruktoren eingesetzt werden; diese sind dann in der Form "`op` *nfx*" anzugeben.

Kapitel 5

Bedeutung der Datentypspezifikationen

Eine Gleichung oder Gen-by-Regel mit einer Struktur, wie sie im vorigen Kapitel gegeben ist, übersetzt das Datentyp-Paket in eine Menge von Typ-, Syntax- und Axiomdeklarationen innerhalb der aktuellen Isabelle-Theorie. Bei Gleichungen erzeugt es zusätzlich Theoreme und beweist diese. Dieses Kapitel stellt die erzeugten Strukturen allgemein und anhand eines aussagekräftigen Beispiels dar.

Zwar ist die Implementierung des Datentyp-Pakets selbst eine formale Spezifikation dieser Strukturen, aber als abstrakte Termbäume und mit unübersichtlichen technischen Details. Die hier dargestellte Spezifikation entspricht dieser logisch, aber verwendet eine möglichst konkrete, intuitive Syntax. Metasprache hierzu ist eine primitive funktionale Sprache, die in $\mathsf{SansSerif}$ notiert ist, mit *kursiv* dargestellten Metavariblen. Der Übersichtlichkeit halber ist in vielen Aufzählungen und Termen mit Schachtelungen flexibler Tiefe auf die explizite Darstellung der Fälle mit insbesondere nur einem oder gar keinem Element verzichtet; solche Terme sind folgendermaßen zu verstehen:

$$
\begin{aligned}
\Lambda v_1 \ \ldots \ v_2 .\ t \ &= \Lambda v_1 v_2 .\ t \\
\Lambda v_1 \ \ldots \ v_1 .\ t \ &= \Lambda v_1 .\ t \\
\Lambda v_1 \ \ldots \ v_0 .\ t \ &= t
\end{aligned}
$$

$$
\begin{aligned}
f(a_1, \ \ldots \ f(a_2, a_3) \ \ldots) \ &= \ f(a_1, f(a_2, a_3)) \\
f(a_1, \ \ldots \ f(a_1, a_2) \ \ldots) \ &= \ f(a_1, a_2) \\
f(a_1, \ \ldots \ f(a_0, a_1) \ \ldots) \ &= \ a_1
\end{aligned}
$$

$$
\begin{aligned}
f(g_1(\ \ldots \ f(g_2(g_3)) \ \ldots \ ) \ &= f(g_1(f(g_2(g_3)))) \\
f(g_1(\ \ldots \ f(g_1(g_2)) \ \ldots \ ) \ &= f(g_1(g_2)) \\
f(g_1(\ \ldots \ f(g_0(g_1)) \ \ldots \ ) \ &= g_1
\end{aligned}
$$

5.1 Theorieerweiterung

Die Abbildung 5.1 zeigt, welche Theorieerweiterungen das Datentyp-Paket aus den Datentypspezifikationen erzeugt.

```
types
```

$nam_1 \;\; l_1$
$\vdots$
$nam_n \;\; l_n$

```
arities
```

nam_1 `:: (pcpo`$_1$`, ..., pcpo`$_{l_1}$`) pcpo`
$\vdots$
nam_n `:: (pcpo`$_1$`, ..., pcpo`$_{l_n}$`) pcpo`

```
ops curried
```

ops_1
$\vdots$
ops_n
ops_{comp}

where $nam_{comp} = nam_1\_\ldots\_nam_n$

v_{ijk} = a variable name that is distinct from $v_{ij1}, \ldots, v_{ijk-1}$ with attributes is_lazy : bool, is_rec : bool, rec_of : int

arg_typ_{ijk} = if is_lazy (v_{ijk}) then $(atyp_{ijk})$`u` else $atyp_{ijk}$

$FT_i(t) = arg\_typ_{i11} \otimes \ldots \otimes arg\_typ_{i1k_{i1}} \oplus \ldots \oplus arg\_typ_{im_i1} \otimes \ldots \otimes arg\_typ_{im_ik_{im_i}}$

$con\_typ_{ij}(t)$ = if $k_{ij} = 0$ then t else $atyp_{ij1} \to \ldots \to atyp_{ijk_{ij}} \to t$

$ops_i =$

nam_i`_abs`	`:: "`$FT_i(typ_1, (\ \ldots\ , typ_i)\ldots) \to typ_i$`"`	
nam_i`_rep`	`:: "`$typ_i \to FT_i(typ_1, (\ \ldots\ , typ_i)\ldots)$`"`	
nam_i`_when`	`:: "`con_typ_{i1}`('t)` $\to \ldots \to con\_typ_{im_i}$`('t)` $\to typ_i \to$ `'t"`	
con_{i1}	`:: "`$con\_typ_{i1}(typ_i)$`"`	syn_{i1}
$\vdots$		
con_{im_i}	`:: "`$con\_typ_{im_i}(typ_i)$`"`	syn_{im_i}
`is_`con_{i1}	`:: "`$typ_i \to$ `tr"`	`("is'_`con_{i1}`")`
$\vdots$		
`is_`con_{im_i}	`:: "`$typ_i \to$ `tr"`	`("is'_`con_{im_i}`)`
sel_{i11}	`:: "`$typ_i \to atyp_{i11}$`"`	
$\ddots$		
$sel_{im_ik_{im_i}}$	`:: "`$typ_i \to atyp_{im_ik_{im_i}}$`"`	
nam_i`_copy`	`:: "`$(typ_1 \to typ_1) \times \ldots \times (typ_n \to typ_n) \to typ_i$`"`	
nam_i`_take`	`:: "nat` $\Rightarrow typ_i \to typ_i$`"`	
nam_i`_finite`	`:: "`$typ_i \Rightarrow$ `bool"`	

$ops_{comp} =$

nam_{comp}`_copy`	`:: "`$(typ_1 \to typ_1) \times \ldots \times (typ_n \to typ_n) \to$
	`::` $(typ_1 \to typ_1) \times \ldots \times (typ_n \to typ_n)$`"`
nam_{comp}`_bisim`	`:: "`$(typ_1 \Rightarrow typ_1 \Rightarrow$ `bool)` $\times \ldots \times (typ_n \Rightarrow typ_n \Rightarrow$ `bool)` $\Rightarrow$ `bool"`

`translations`

$trans_1$
...
$trans_n$

where $case\_arg_{ij}$ = con_{ij}`'aj_1'...'aj_`k_{ij} `=> ej`
$when\_arg_{ij}$ = `(Λaj_1 ... aj_`k_{ij}`. ej)`
$trans_i$ = `"case x of` $case\_arg_{i1}$ `| ... |` $case\_arg_{im_i}$`" ==`
`"`nam_i`_when'`$when\_arg_{i1}$`'...'`$when\_arg_{im_i}$`'x"`

`rules`

$rules_1$
$\vdots$
$rules_n$
$rules_{comp}$

where
$lv_{ijk}(s)$ = if is_lazy (v_{ijk}) then s' v_{ijk} else v_{ijk}
$darg_{ijk}(t_1, t_2)$ = Λv_{ij1} ... $v_{ijk_{ij}}$.if $k = j$ then t_1 else t_2
$bin\_branchr(f_1, f_2, t, i, j)$ = if $i = 1$ then t else if $j = 1$ then f_1't else if $i = 2$ then f_2't
else $bin\_branchr(f_1, f_2, f_2`t, i{-}1, j{-}1)$
$cproj_i(t)$ = $bin\_branchr($`cfst,csnd`$, t, n, i)$
$proj_i(t)$ = $bin\_branchr($ `fst, snd`$, t, n, i)$

$rules_i$ =
nam_i`_abs_iso` `"`nam_i`_rep'(`nam_i`_abs'x) = x"`
nam_i`_rep_iso` `"`nam_i`_abs'(`nam_i`_rep'x) = x"`
nam_i`_when_def` `"`nam_i`_when ≡ Λf1 ... f`m_i `x.`
`sswhen'`$split_{i1}$`'( ... sswhen'`$split_{im_i-1}$`'`$split_{im_i}$ `... )'(`nam_i`_rep'x)"`
where $fapp_{ij}$ = `f`j`'`lv_{ij1}`(lift'ID)'...'`$lv_{ijk_{ij}}$`(lift'ID)`
$split_{ij}$ = if $k_{ij} = 0$ then `Λdummy. f`j else if $k_{ij} = 1$ then `f`j else
`ssplit'(Λ`v_{ij1}`. ... ssplit'(Λ`$v_{ijk_{ij}-1}$ `Λ`$v_{ijk_{ij}}$`.` $fapp_{ij}$`) ... )`
con_{i1}`_def` `"`con_{i1} ≡ Λv_{ij1} ... $v_{ijk_{ij}}$. nam_i`_abs'`rep_con_{i1}`"`
$\vdots$
con_{im_i}`_def` `"`con_{im_i} ≡ Λv_{ij1} ... $v_{ijk_{ij}}$. nam_i`_abs'`$rep\_con_{im_i}$`"`
where $inj_{ij}(t)$ = if $i = 1$ then t else if $j = 1$ then `sinl'`t else
if $i = 2$ then `sinr'`t else `sinr'`$inj_{i-1j-1}(t)$
rep_con_{ij} = if $k_{ij} = 0$ then `one` else inj_{m_ij}`((|`lv_{ij1}`(up), ...,` $lv_{ijk_{ij}}$`(up)|))"`
`is_`con_{i1}`_def` `"is_`con_{i1} ≡ nam_i`_when'`$darg_{i11}$ `(TT ,FF)'...'`$darg_{i1k_{i1}}$ `(TT ,FF)"`
$\vdots$
`is_`con_{im_i}`_def` `"is_`con_{im_i} ≡ nam_i`_when'`$darg_{im_i1}$`(TT ,FF)'...'`$darg_{im_ik_{im_i}}$`(TT ,FF)"`
sel_{i11}`_def` `"`sel_{i11} ≡ nam_i`_when'`$darg_{i11}$ $(v_{i11}$ $,\bot)$`'...'`$darg_{i1k_{1i}}$ $(v_{i11}$ $,\bot)$`"`
$\ddots$
$sel_{im_ik_{im_i}}$`_def` `"`$sel_{im_ik_{im_i}}$ ≡ nam_i`_when'`$darg_{im_i1}(v_{im_ik_{im_i}},\bot)$`'...'`$darg_{im_ik_{im_i}}(v_{im_ik_{im_i}},\bot)$`"`

nam_i_copy_def "nam_i_copy $\equiv$ Λf. nam_i_abs oo FT_i(f) oo nam_i_rep"
where $farg_{ijk}$ = if is_rec (v_{ijk}) then $cproj_{\mathsf{rec\_of}(v_{ijk})}$(f) else ID
$carg_{ijk}$ = if is_lazy (v_{ijk}) then lift‘(up oo $farg_{ijk}$) else $farg_{ijk}$
$split_{ij}$ = if k_{ij} = 0 then ID else ssplit‘(Λx y.
⦇$carg_{ij1}$‘x, ... ssplit‘(Λx y. ⦇$carg_{ijk_{ij}-1}$‘x, $carg_{ijk_{ij}}$‘y⦈) ... ⦈)
$FT_i(f)$ = sswhen‘(sinl oo $split_{i1}$)‘
(sinr oo ... sswhen‘(sinl oo $split_{im_i-1}$)‘(sinr oo $split_{im_i}$) ...)

$rules_{comp}$ =
nam_{comp}_copy_def "nam_{comp}_copy $\equiv$ Λf. <nam_1_copy‘f, ..., nam_n_copy‘f>"
nam_1_reach "$cproj_1$(fix‘nam_{comp}_copy)‘x1 = x1"
⋮
nam_n_reach "$cproj_n$(fix‘nam_{comp}_copy)‘xn = xn"
nam_1_take_def "nam_1_take $\equiv$ λn. $cproj_1$(iterate n nam_{comp}_copy ⊥)"
⋮
nam_n_take_def "nam_n_take $\equiv$ λn. $cproj_n$(iterate n nam_{comp}_copy ⊥)"
nam_1_finite_def "nam_1_finite $\equiv$ λx1. ∃n. nam_1_take n‘x1 = x1"
⋮
nam_n_finite_def "nam_n_finite $\equiv$ λxn. ∃n. nam_n_take n‘xn = xn"
nam_{comp}_bisim_def "nam_{comp}_bisim $\equiv$ λR. $bisim_1$ ∧ ... ∧ $bisim_n$"
where $pair_{ijk}$ = (if is_lazy (v_{ijk}) then True else v_{ijk} ≠ ⊥ ∧ v_{ijk}’ ≠ ⊥) ∧
(if is_rec (v_{ijk}) then $proj_{\mathsf{rec\_of}(v_{ijk})}$(R) v_{ijk} v_{ijk}’ else v_{ijk} = v_{ijk}’)
sim_{ij} = ∃v_{ij1} v_{ij1}’ ... $v_{ijk_{ij}}$ $v_{ijk_{ij}}$’. $pair_{ij1}$ ∧ ... ∧ $pair_{ijk_{ij}}$ ∧
xi = con_{ij}‘v_{ij1}‘...‘$v_{ijk_{ij}}$ ∧ xi’ = con_{ij}‘v_{ij1}’‘...‘$v_{ijk_{ij}}$’
$bisim_i$ = ∀xi xi’. $proj_i$(R) xi xi’ ⟶ xi = ⊥ ∧ xi’ = ⊥ ∨ sim_{i1} ∨ ... ∨ sim_{im_i}

(* for generated-by-rule *)

nam_1_..._nam_n_induct
"⟦ (if finite then True else adm P1; ...; adm Pn);
P1 ⊥; ...; Pn ⊥;
$step_{11}$; ...; $step_{1m_1}$;
⋮
$step_{n1}$; ...; $step_{nm_n}$ ⟧ ⟹
P1 x1 ∧ ... ∧ Pn xn"
where k_{ij} = arity of con_{ij}
v_{ijk} = a variable name that is distinct from $v_{ij1}, \ldots, v_{ijk-1}$
with attributes is_rec : bool, rec_of : int
con_app_{ij} = con_{ij}‘v_{ij1}‘...‘$v_{ijk_{ij}}$
rec_{ijk} = if is_rec (v_{ijk}) then P(rec_of(v_{ijk})) v_{ijk} else True
$step_{ij}$ = ⋀v_{ij1} ... $v_{ijk_{ij}}$. ⟦ rec_{ij1}; ...; $rec_{ijk_{ij}}$; con_app_{ij} ≠ ⊥ ⟧ ⟹ Pi (con_app_{ij})

Abbildung 5.1: erzeugte Theorie-Elemente

Für Gleichungen werden zunächst die Typdeklarationen und die zugehörigen Aritätsvereinbarungen erzeugt. Man beachte, daß alle Typen der Klasse `pcpo` angehören. Danach werden die für alle Axiome und Definitionen notwendigen logischen Konstanten mit ihrer jeweiligen Typisierung deklariert, ebenso eine syntaktische Übersetzungsregel zur intuitiven Darstellung von Fallunterscheidungen über den Datentypen mit dem von funktionalen Sprachen bekannten `case`-Konstrukt.

Es folgen die Axiome der Bereichsisomorphismen `abs_iso` und `rep_iso` sowie – weiter unten – die `reach`-Axiome mit den `copy`-Funktionalen, welche die Bereiche eindeutig und konsistent charakterisieren, wie in Kapitel 3 ausgeführt. Das `copy`-Funktional ist der Morphismusanteil des dort genannten mit der Bereichsgleichung verbundenen Funktors, mit vorgeschalteter Repräsentationsfunktion und nachgeschalteter Abstraktion. Der restliche Teil sind reine Definitionen, nämlich die `when`-Funktionale (zur Behandlung von Fallunterscheidungen über dem Datentyp), die Konstruktoren (zum Aufbau von Datenelementen), Diskriminatoren (zum Testen, welche Varianten des Datentyps in einem Element vorliegt) und Selektoren (zum Auslesen von Werten aus einem Datenelement) sowie für die Induktionsprinzipien die `take`-Funktionale (zur Darstellung der schrittweisen Approximation der Datentypen, im wesentlichen die iterative Anwendung der `copy`-Funktionale), schließlich die Prädikate zur Endlichkeit der Datentypen (zur Vereinfachung der Induktion) und die Definition der Bisimulation auf dem Datentyp (für die Coinduktion).

Für Gen-By-Regeln wird lediglich die Induktionsregel als Axiom erzeugt; dies stellt *keine* konservative Theorieerweiterung dar.

5.2 Theoremerzeugung

Aus den gerade genannten Axiomen und Definitionen bestimmt und beweist das Datentyp-Paket für jede Gleichung eine Menge von Theoremen, die die Eigenschaften des erzeugten Datentyps in für Anwendungen geeigneter Form beschreiben. Dies sind pro Teil-Datentyp die Striktheit des Bereichsisomorphismus, eine Regel zur Ausschöpfung und Fallunterscheidung über die Konstruktoren, die Striktheit und Anwendungsfälle des `when`-Funktionals, Striktheit und Definiertheit der Konstruktoren, Striktheit, Definiertheit und Anwendungsfälle der Selektoren und Diskriminatoren, Nicht-Abschwächung und Ungleichheit der verschiedenen Konstruktoren, Invers-Monotonie und Injektivität der Konstruktoren, sowie Striktheit und Anwendungsfälle des `copy`-Funktionals. Für den Gesamt-Datentyp sind dies die Striktheit und Anwendungsfälle der `take`- Funktionale, Gleichheitslemmata für die Approximation, Endlichkeit der Datentypen (ansonsten eine triviale Aussage), sowie die Regel zur strukturellen Induktion für endliche Approximationen, zur allgemeinen strukturellen Induktion und zur Coinduktion.

Die Theoreme werden in je einer ML-Struktur pro Gleichungsteil und einer für den gemeinsamen Anteil (bei verschränkt rekursiven Datentypen) dem Benutzer zur Verfügung gestellt. Abbildung 5.2 zeigt, welche Strukturen und Theoreme erzeugt werden.

(* for the general case $n \neq 1$ *)

$struct_1$
$\vdots$
$struct_n$
$struct_{comp}$

where $struct_i$ = structure nam_i :

```
      sig
         val abs_iso   : thm
         val rep_iso   : thm
         val when_def  : thm
         val con_defs  : thm list
         val dis_defs  : thm list
         val sel_defs  : thm list
         val copy_def  : thm
         val iso_rews  : thm list
         val exhaust   : thm
         val cases     : thm
         val when_rews : thm list
         val con_rews  : thm list
         val sel_rews  : thm list
         val dis_rews  : thm list
         val dists_le  : thm list
         val dists_eq  : thm list
         val inverts   : thm list
         val injects   : thm list
         val copy_rews : thm list
         val rews      : thm list
      end
```

$struct_{comp}$ = structure nam_{comp} :

```
      sig
         val copy_def    : thm
         val reachs      : thm list
         val take_defs   : thm
         val finite_defs : thm
         val bisim_def   : thm
         val take_rews   : thm list
         val take_lemmas : thm list
         val finites     : thm list
         val finite_ind  : thm
         val ind         : thm
         val coind       : thm
         val rews        : thm list
      end
```

```
(* for the special case n = 1 *)

   structure nam_1 :
      sig
         val abs_iso   : thm
         val rep_iso   : thm
         val when_def  : thm
         val con_defs  : thm list
         val dis_defs  : thm list
         val sel_defs  : thm list
         val copy_def  : thm
         val iso_rews  : thm list
         val exhaust   : thm
         val cases     : thm
         val when_rews : thm list
         val con_rews  : thm list
         val sel_rews  : thm list
         val dis_rews  : thm list
         val dists_le  : thm list
         val dists_eq  : thm list
         val inverts   : thm list
         val injects   : thm list
         val copy_rews : thm list
         val reach       : thm
         val take_def    : thm
         val finite_def  : thm
         val bisim_def   : thm
         val take_rews   : thm list
         val take_lemma  : thm
         val finite      : thm
         val finite_ind  : thm
         val ind         : thm
         val coind       : thm
         val rews        : thm list
      end
```

```
(* for all cases *)
```

$$
\begin{array}{lcl}
is_{finite}(t) & = & \text{there is non} - \text{strict type recursion in } t \\
defd_{ijk} & = & \text{if is\_lazy } (v_{ijk}) \text{ then } \mathtt{True} \text{ else } v_{ijk} \neq \perp \\
defd'_{ijk} & = & \text{if is\_lazy } (v'_{ijk}) \text{ then } \mathtt{True} \text{ else } v'_{ijk} \neq \perp \\
defd\_nlas_{ij} & = & [\![\; defd_{ij1} ; \; \ldots ; \; defd_{ijk_{ij}} \;]\!] \\
con\_app_{ij} & = & con_{ij} \text{`} v_{ij1} \text{`} \ldots \text{`} v_{ijk_{ij}}
\end{array}
$$

```
(* open nam_i; *)

val abs_iso   = (* ... *)
⋮
val copy_def  = (* ... *)
val iso_rews  = ["nam_i_rep`(nam_i_abs`x) = x", "nam_i_abs`(nam_i_rep`x) = x",
                 "nam_i_abs`⊥  = ⊥", "nam_i_rep`⊥  = ⊥"] : thm list
val exhaust   =                  "x = ⊥                                              ∨
                (∃v_{i11} ... v_{i1k_{i1}}        . x = con_app_{i1}   ∧ defd_{i11}    ∧ ... ∧ defd_{i1k_{i1}}        )  ∨
                 ⋮                                                                                      ∨
                (∃v_{im_i1} ... v_{im_ik_{im_i}}. x = con_app_{im_i} ∧ defd_{im_i1} ∧ ... ∧ defd_{im_ik_{im_i}})" : thm
val cases     = "⟦                         x = ⊥                                          ⟹ P;
                 ⋀v_{i11} ... v_{i1k_{i1}}        . ⟦ x = con_app_{i1}   ; defd_{i11}; ...; defd_{i1k_{i1}}         ⟧ ⟹ P;
                 ⋮                                                                                        ;
                 ⋀v_{im_i1} ... v_{im_ik_{im_i}}. ⟦ x = con_app_{im_i}; defd_{im_i1}; ...; defd_{im_ik_{im_i}} ⟧ ⟹ P ⟧
                 ⟹ P" : thm
val when_rews = ["(if m_i = 1 ∧ k_{i1} = 1 then f`⊥ = ⊥ else True) ⟹ when_i_app(⊥) = ⊥",
                 when_{i1}, ..., when_{im_i}] : thm list
          where when_i_app(t) = nam_i_when`f1`...`fm_i`t
                when_{ij}        = "defd_nlas_{ij} ⟹ when_i_app(con_{ij}`v_{ij1}`...`v_{ijk_{ij}}) = fj`v_{ij1}`...`v_{ijk_{ij}}"
val con_rews  = [con_strict_{i11},
                 ⋱,
                 con_strict_{im_ik_{im_i}},
                 "defd_nlas_{i1} ⟹ con_app_{i1} ≠ ⊥", ..., "defd_nlas_{im_i} ⟹ con_app_{im_i} ≠ ⊥"]
                : thm list
    where con_strict_{ijk} = if is_lazy (v_{ijk}) then "True" else "con_{ij}`v_{ij1}`...`v_{ijk-1}`⊥`v_{ijk+1}`...`v_{ijk_{ij}} = ⊥"
val sel_rews  = ["sel_{i11}   `⊥ = ⊥",
                 ⋱,
                 "sel_{im_ik_{im_i}}`⊥ = ⊥"] @
                 (if m_i = 1 then ["x ≠ ⊥ ⟹ sel_{i11}`x ≠ ⊥", ..., "x ≠ ⊥ ⟹ sel_{i1k_{i1}}`x ≠ ⊥"]
                            else []) @
                 sel_apps_{i11} @
                 ⋱ @
                 sel_apps_{im_ik_{im_i}}  : thm list
          where sel_app_{ijkl} = "defd_nlas_{il} ⟹ sel_{ijk}`(con_{il}`v_{il1}`...`v_{ilk_{il}}) = (if j = l then v_{ijk} else ⊥)"
                sel_apps_{ijk} = [sel_app_{ijk1}, ..., sel_app_{ijkm_i}]
val dis_rews  = (if m_i = 1 ∧ k_{i1} = 1 then ["is_con_{i1}`⊥ ≠ ⊥"]
                                          else ["is_con_{i1}`⊥ = ⊥", ..., "is_con_{im_i}`⊥ = ⊥"]) @
                ["x ≠ ⊥ ⟹ is_con_{i1}`x ≠ ⊥", ..., "x ≠ ⊥ ⟹ is_con_{im_i}`x ≠ ⊥",
                 dis_app_{i11},
                 ⋱,
                 dis_app_{im_im_i}] : thm list
          where dis_app_{ijk} = "defd_nlas_{ij} ⟹ is_con_{ij}`con_app_{ik} = (if j = k then TT else FF)"
```

```
val dists_le  = d_le_{i12} @ ... @ d_le_{i1m_i} @
                ⋮ @
                d_le_{im_i-1m_i} : thm list
          where d_le_{ijk} = ["defd_nlas_{ij} ⟹ ¬ con_app_{ij} ⊑ con_app_{ik}",
                              "defd_nlas_{ik} ⟹ ¬ con_app_{ik} ⊑ con_app_{ij}"]
val dists_eq  = d_eq_{i12} @ ... @ d_eq_{i1m_i} @
                ⋮ @
                d_eq_{im_i-1m_i} : thm list
          where neq_{ijk}  = "defd_nlas_{ij} ⟹ con_app_{ij} ≠ con_app_{ik}"
                d_eq_{ijk} = if (∀l. is_lazy (v_{ijl})) ∨ (∀l. is_lazy (v_{ilk}))
                             then ["con_app_{ij} ≠ con_app_{ik}", "con_app_{ik} ≠ con_app_{ij}"]
                             else [neq_{ijk}, neq_{ikj}]
val inverts   = invj_i(⊑)
val injects   = invj_i(=)
          where in_{ij}(Ξ)  = "⟦ con_{ij}‘v_{ij1}‘...‘v_{ijk_{ij}} Ξ con_{ij}‘v'_{ij1}‘...‘v'_{ijk_{ij}}; defd_{ij1}; ...; defd_{ijk_{ij}};
                                 defd'_{ij1}; ...; defd'_{ijk_{ij}} ⟧ ⟹ v_{ij1} Ξ v'_{ij1} ∧ ... ∧ v_{ijk_{ij}} Ξ v'_{ijk_{ij}}"
                invj_i(Ξ) = [in_{i1}(Ξ), ..., in_{im_i}(Ξ)] : thm list
val copy_rews = ["(if m_i = 1 ∧ k_{i1} = 1 then f‘⊥ = ⊥ else True) ⟹ nam_i_copy‘f‘⊥ = ⊥",
                 copy_a_{i1}, ..., copy_a_{im_i},
                 copy_s_{i1}, ..., copy_s_{im_i}] : thm list
          where defrec_{ijk} = if is_lazy (v_{ijk}) ∨ ¬ is_rec (v_{ijk}) then True else v_{ijk} ≠ ⊥
                arg_{ijk}    = if is_rec (v_{ijk}) then (cproj_{rec_of(v_{ijk})}(f)‘v_{ijk}) else v_{ijk}
                copy_a_{ij}  = "⟦ defrec_{ij1}; ...; defrec_{ijk_{ij}} ⟧ ⟹
                                 nam_i_copy‘f‘(con_{ij}‘v_{ij1}‘...‘v_{ijk_{ij}}) = con_{ij}‘arg_{ij1}‘...‘arg_{ijk_{ij}}"
                copy_s_{ij}  = if ∀l. (is_lazy (v_{ijk}) ∨ ¬ is_rec (v_{ijk})) then "True"
                               else "nam_i_copy‘⊥‘(con_app_{ij}) = ⊥"
val rews      = iso_rews @ when_rews @
                con_rews @ sel_rews @ dis_rews @ dists_le @ dists_eq @
                copy_rews

(* open nam_{comp}; *)
val copy_def  = (* ... *)
⋮
val bisim_def = (* ... *)
val take_rews = ["nam_1_take n‘⊥  = ⊥", ..., "nam_n_take n‘⊥  = ⊥",
                 "nam_1_take 0‘x1 = ⊥", ..., "nam_n_take 0‘xn = ⊥",
                 take_{11},
                 ⋱
                 take_{nm_n}] : thm list
          where arg_{ijk} = if is_rec (v_{ijk}) then (nam_{rec_of(v_{ijk})}_take n‘v_{ijk}) else v_{ijk}
                take_{ij} = "nam_i_take (Suc n)‘(con_{ij}‘v_{ij1}‘...‘v_{ijk_{ij}}) = con_{ij}‘arg_{ij1}‘...‘arg_{ijk_{ij}}"
val take_lemmas=["(⋀n. nam_1_take n‘x1 = nam_1_take n‘x1') ⟹ x1 = x1'",
                  ⋮
                 "(⋀n. nam_n_take n‘xn = nam_n_take n‘xn') ⟹ xn = xn'"] : thm list
```

```
val finites    = if is_finite (nam_comp)
                 then ["nam_1_finite x1", ..., "nam_n_finite xn"] : thm list
                 else  ["¬ nam_1_finite x1 ∨ nam_1_finite x1", ...,
                        "¬ nam_n_finite xn ∨ nam_n_finite xn"] : thm list
val finite_ind= "⟦ prems ⟧ ⟹ P1 (nam_1_take n`x1) ∧ ... ∧ Pn (nam_n_take n`xn)" : thm
val ind        = "⟦ (if is_finite (nam_comp) then True else adm P1; ...; adm Pn); prems ⟧ ⟹
                  P1 x1 ∧ ... ∧ Pn xn" : thm
            where prems = P1 ⊥; step_11; ...; step_1m_1;
                           ⋮
                           Pn ⊥; step_n1; ...; step_nm_n
                  rec_ijk  = if is_rec (v_ijk) then P(rec_of(v_ijk)) v_ijk else True
                  step_ij  = ⋀v_ij1 ... v_ijk_ij. ⟦ defd_ij1; ...; defd_ijk_ij; rec_ij1; ...; rec_ijk_ij ⟧ ⟹
                                             Pi (con_ij`v_ij1`...`v_ijk_ij)
val coind      = "⟦ nam_comp_bisim R; proj_1(R) x1 x1'; ...; proj_n(R) xn xn' ⟧ ⟹
                  x1 = x1' ∧ ... ∧ xn = xn'" : thm
val rews       = struct_1.rews @ ... @ struct_n.rews @ take_rews


(* transformation of the structures for the special case n = 1 *)

(* open nam_1; *)

val abs_iso    = struct_1.abs_iso   : thm
⋮
val copy_def   = struct_1.copy_def  : thm
val iso_rews   = struct_1.iso_rews  : thm list
val exhaust    = struct_1.exhaust   : thm
val cases      = struct_1.cases     : thm
val when_rews = struct_1.when_rews : thm list
⋮
val copy_rews = struct_1.copy_rews : thm list
val reach         = hd(struct_comp.reachs)       : thm
val take_def      = hd(struct_comp.take_defs)    : thm
val finite_def   = hd(struct_comp.finite_defs) : thm
val bisim_def    =     struct_comp.bisim_def     : thm
val take_rews    =     struct_comp.take_rews     : thm list
val take_lemma   = hd(struct_comp.take_lemmas) : thm
val finite        = hd(struct_comp.finites)      : thm
val finite_ind   =     struct_comp.finite_ind    : thm
val ind           =     struct_comp.ind           : thm
val coind         =     struct_comp.coind         : thm
val rews          = struct_1.rews @ struct_comp.take_rews : thm list
```

Abbildung 5.2: erzeugte Theoreme

5.3 Beispiele für Datentypspezifikationen

Dieser Abschnitt bringt signifikante Beispiele für Datentypspezifikationen und die daraus gewonnenen Datentyp-Eigenschaften. Weil hier Datentypen ganz unterschiedlicher Komplexität (Zahl der Gleichungen, Konstruktoren und Konstruktorargumente u.ä.) beteiligt sind, kann an ihnen die Wirkung des Datentyp-Pakets eingehend studiert werden.

5.3.1 Leerer Datentyp

Ein Extremfall ist der leere Typ, der isomorph zu `void` ist. Dieser wird hier `empty` genannt und mittels Rekursion auf einem strikten Argument des einzigen Konstruktors `silly` definiert. Das Datentyp-Paket gibt für ihn eine Warnung aus, wie aus Abbildung 5.5 ersichtlich. Zudem möge er als Vertreter der Klasse von Datentypspezifikationen mit nur einer Gleichung und einem Konstruktor mit nur einem Argument dazu dienen, die Besonderheiten dieser Klasse und ansonsten die einfachste Form der erzeugten Strukturen zu repräsentieren.

5.3.2 Transfiniter Baum

Das zweite Beispiel ist die etwas "exotische" Konstruktion eines Baumes mit unendlicher Verzweigung, der über eine dreifache simultane Gleichung definiert ist (wobei die mittlere redundant ist) und über 0- bis 3-stellige Konstruktoren verfügt. Als syntaktische Besonderheiten bietet er mehrstellige Typ-Parametrisierung, Konstruktoren mit Sonderzeichen sowie einen Infix- und einen Mixfix-Konstruktor.

5.3.3 Beliebig verzweiger Baum

Als typisches Beispiel für eine Gen-By-Regel mit einem verschränkt rekursiven Datentyp seien hier die bekannten `Tree`s and `Forest`s erwähnt, die die Gültigkeit der Induktionsregel über einen beliebig verzweigten Baum fordert. Daß dies der Fall ist und die `finite`- Option sinnvoll ist, bedarf eines externen semantischen Arguments, in das die (durch hier allerdings fehlende Definitionen bestimmten) Eigenschaften der Konstruktoren eingehen.

5.3.4 Fehlerhafte Induktion

Das letzte Beispiel zeigt eine unsachgemäße Gen-By-Regel. Sie behauptet die Konstruierbarkeit des Typs `('a,'b) branch` (aus dem oben erwähnten Datentyp eines transfiniten Baumes) ohne Einbeziehung der Typen `('a,'b) tree_1` und `('b,'a) tree_2`. Dies liefert eine falsche Induktionsregel.

Auf den folgenden Seiten ist die Isabelle-Theorie mit der Definition der Beispiel-Datentypspezifikationenen sowie die daraus erzeugte Theorie abgedruckt.

```
Example = HOLCF +

(* Empty domain *)
(* FT = I *)

domain empty = silly empty

(* Exotic transfinite tree domain *)
(* FT_1 = <++, K_one, <++, P2 o P2, <**, K_'b, K_tr> > >
   FT_2 = P1
   FT_3 = <**, K_'a, <**, P1 o P2, <U, P2 o P2> > > *)

domain
('a,'b) tree_1 = leaf | node (branch :: ('a,'b) branch) | in'_f 'b tr (cinfixr 5),
('b,'a) tree_2 = "#" (('a,'b) tree_1)                                  ("#"),
('a,'b) branch = con (val::'a) (bhd::('b,'a) tree_2) (lazy btl::('a,'b) branch)
                                                                       (cmixfix "<_:_|>>_")
(* Induction rule for Trees and Forests *)

types

  Tree   1
  Forest 1

arities

  Tree   :: (pcpo) pcpo
  Forest :: (pcpo) pcpo

consts

  eTree   ::                          "'a Tree"
  mTree   :: "'a      → 'a Forest → 'a Tree"

  eForest ::                          "'a Forest"
  mForest :: "'a Tree → 'a Forest → 'a Forest"

generated finite Tree   by eTree   | mTree,
                 Forest by eForest | mForest

(* Wrong induction rule for branch *)

generated branch by con

end
```

Abbildung 5.3: Beispiel: gegebene Theorie

```
Example' = HOLCF +

types
  empty  0

  tree_1 2
  tree_2 2
  branch 2

  Tree   1
  Forest 1

arities
  empty  ::             pcpo

  tree_1 :: (pcpo,pcpo) pcpo
  tree_2 :: (pcpo,pcpo) pcpo
  branch :: (pcpo,pcpo) pcpo

  Tree   :: (pcpo)      pcpo
  Forest :: (pcpo)      pcpo

ops curried
  empty_abs      ::          "empty → empty"
  empty_rep      :: "empty → empty"
  empty_when     :: "(empty → 't) → empty → 't"
  silly          :: "empty → empty"
  is_silly       ::          "empty → tr"                            ("is'_silly")
  empty_sel_1_1  ::          "empty → empty"
  empty_copy     :: "(empty → empty) → empty → empty"
  empty_take     ::              "nat ⇒ empty → empty"
  empty_finite   ::                              "empty ⇒ bool"
  empty_bisim    :: "(empty ⇒ empty ⇒ bool) ⇒ bool"

  tree_1_abs     ::                  "one ⊕ ('a,'b) branch ⊕ 'b ⊗ tr → ('a,'b) tree_1"
  tree_1_rep     :: "('a,'b) tree_1 → one ⊕ ('a,'b) branch ⊕ 'b ⊗ tr"
  tree_1_when    :: "'t → (('a,'b) branch→'t) → ('b→tr→'t) → ('a,'b) tree_1 → 't"
  leaf           ::                  "('a,'b) tree_1"
  node           :: "('a,'b) branch → ('a,'b) tree_1"
  "in'_f"        ::      "'b → tr → ('a,'b) tree_1"                  (cinfixr 5)
  is_leaf        ::                  "('a,'b) tree_1 → tr"           ("is'_leaf")
  is_node        ::                  "('a,'b) tree_1 → tr"           ("is'_node")
  is_in_f        ::                  "('a,'b) tree_1 → tr"           ("is'_in'_f")
  branch         ::            "('a,'b) tree_1 → ('a,'b) branch"
  tree_1_sel_3_1 ::            "('a,'b) tree_1 → 'b"
  tree_1_sel_3_2 ::            "('a,'b) tree_1 → tr"
```

```
tree_1_copy    :: "(('a,'b) tree_1 → ('a,'b) tree_1) ×
                   (('b,'a) tree_2 → ('b,'a) tree_2) ×
                   (('a,'b) branch → ('a,'b) branch) → ('a,'b) tree_1 → ('a,'b) tree_1"
tree_1_take    :: "nat ⇒ ('a,'b) tree_1 → ('a,'b) tree_1"
tree_1_finite  ::                          "('a,'b) tree_1 ⇒ bool"

tree_2_abs     ::                   "('a,'b) tree_1 → ('b,'a) tree_2"
tree_2_rep     :: "('b,'a) tree_2 → ('a,'b) tree_1"
tree_2_when    :: "(('a,'b) tree_1 → 't) → ('b,'a) tree_2 → 't"
"#"            ::  "('a,'b) tree_1 → ('b,'a) tree_2"
"is_#"         ::                    "('b,'a) tree_2 → tr"                  ("is'_#")
tree_2_sel_1_1 ::                    "('b,'a) tree_2 → ('a,'b) tree_1"
tree_2_copy    :: "(('a,'b) tree_1 → ('a,'b) tree_1) ×
                   (('b,'a) tree_2 → ('b,'a) tree_2) ×
                   (('a,'b) branch → ('a,'b) branch) → ('b,'a) tree_2 → ('b,'a) tree_2"
tree_2_take    :: "nat ⇒ ('b,'a) tree_2 → ('b,'a) tree_2"
tree_2_finite  ::                          "('b,'a) tree_2 ⇒ bool"

branch_abs     :: "'a ⊗ ('b,'a) tree_2 ⊗ ('a,'b) branch u → ('a,'b) branch"
branch_rep     :: "('a,'b) branch → 'a ⊗ ('b,'a) tree_2 ⊗ ('a,'b) branch u"
branch_when    :: "('a → ('b,'a) tree_2 → ('a,'b) branch → 't) → ('a,'b) branch →'t"
con            :: "'a→('b,'a) tree_2→('a,'b) branch→('a,'b) branch" (cmixfix "<_:_|>>_")
is_con         ::                                    "('a,'b) branch → tr" ("is'_con")
val            ::                                    "('a,'b) branch → 'a"
bhd            ::                                    "('a,'b) branch → ('b,'a) tree_2"
btl            ::                                    "('a,'b) branch → ('a,'b) branch"
branch_copy    :: "(('a,'b) tree_1 → ('a,'b) tree_1) ×
                   (('b,'a) tree_2 → ('b,'a) tree_2) ×
                   (('a,'b) branch → ('a,'b) branch) → ('a,'b) branch → ('a,'b) branch"
branch_take    :: "nat ⇒ ('a,'b) branch → ('a,'b) branch"
branch_finite  ::                          "('a,'b) branch ⇒ bool"

tree_1_tree_2_branch_copy  ::
  "(('a,'b) tree_1 → ('a,'b) tree_1) × (('b,'a) tree_2 → ('b,'a) tree_2) ×
   (('a,'b) branch → ('a,'b) branch) →
   (('a,'b) tree_1 → ('a,'b) tree_1) × (('b,'a) tree_2 → ('b,'a) tree_2) ×
   (('a,'b) branch → ('a,'b) branch)"
tree_1_tree_2_branch_bisim  ::
  "(('a,'b) tree_1 ⇒ ('a,'b) tree_1 ⇒ bool) ×
   (('b,'a) tree_2 ⇒ ('b,'a) tree_2 ⇒ bool) ×
   (('a,'b) branch ⇒ ('a,'b) branch ⇒ bool) ⇒ bool"

eTree   ::                         "'a Tree"
mTree   :: "'a      → 'a Forest → 'a Tree"
eForest ::                         "'a Forest"
mForest :: "'a Tree → 'a Forest → 'a Forest"
```

```
translations
  "case x of silly`a1_1 => e1"         == "empty_when`(Λa1_1. e1)`x"
  "case x of leaf => e1 | node`a2_1 => e2 | a3_1 in_f a3_2 => e3" ==
        "tree_1_when`e1`(Λa2_1. e2)`(Λa3_1 a3_2. e3)`x"
  "case x of #`a1_1 => e1"             == "tree_2_when`(Λa1_1. e1)`x"
  "case x of con`a1_1`a1_2`a1_3 => e1" == "branch_when`(Λa1_1 a1_2 a1_3. e1)`x"

rules
  empty_abs_iso     "empty_rep`(empty_abs`x) = x"
  empty_rep_iso     "empty_abs`(empty_rep`x) = x"
  empty_when_def    "empty_when ≡ Λf x. f`(empty_rep`x)"
  silly_def         "silly       ≡ Λe. empty_abs`e
  is_silly_def      "is_silly     ≡ empty_when`(Λe. TT)"
  empty_sel_1_1_def "empty_sel_1_1 ≡ empty_when`(Λe. e)"
  empty_copy_def    "empty_copy ≡ Λf  . empty_abs oo f oo empty_rep"
  empty_take_def    "empty_take   ≡ λn. iterate n empty_copy ⊥"
  empty_reach       "fix`empty_copy`x = x"
  empty_finite_def  "empty_finite ≡ λx. ∃n. empty_take n`x = x"
  empty_bisim_def   "empty_bisim ≡ λR. ∀x x'. R x x' ⟶
                     x = ⊥ ∧ x' = ⊥ ∨
                     (∃e e'. e ≠ ⊥ ∧ e' ≠ ⊥ ∧ R e e' ∧ x = silly`e ∧ x' = silly`e')"

  tree_1_abs_iso     "tree_1_rep`(tree_1_abs`x) = x"
  tree_1_rep_iso     "tree_1_abs`(tree_1_rep`x) = x"
  tree_1_when_def    "tree_1_when ≡ Λf1 f2 f3 x.
                      sswhen`(Λdummy. f1)`(sswhen`f2`(ssplit`f3))`(tree_1_rep`x)"
  tree_1_copy_def    "tree_1_copy ≡ Λf. tree_1_abs oo
                      sswhen`sinl`(sinr oo sswhen`(sinl oo csnd`(csnd`f))`
                      (sinr oo ssplit`(Λx y. (|x, y|)))) oo tree_1_rep"
  leaf_def           "leaf    ≡        tree_1_abs`(sinl`      one   )"
  node_def           "node    ≡ Λb.    tree_1_abs`(sinr`(sinl` b    ))"
  in_f_def           "op in_f ≡ Λb t.  tree_1_abs`(sinr`(sinr`(|b, t|)))"
  is_leaf_def        "is_leaf       ≡ tree_1_when`TT`(Λb. FF)`(Λb t. FF)"
  is_node_def        "is_node       ≡ tree_1_when`FF`(Λb. TT)`(Λb t. FF)"
  is_in_f_def        "is_in_f       ≡ tree_1_when`FF`(Λb. FF)`(Λb t. TT)"
  branch_def         "branch        ≡ tree_1_when`⊥ `(Λb. b) `⊥"
  tree_1_sel_3_1_def "tree_1_sel_3_1 ≡ tree_1_when`⊥ `⊥       `(Λb t. b)"
  tree_1_sel_3_2_def "tree_1_sel_3_2 ≡ tree_1_when`⊥ `⊥       `(Λb t. t)"

  tree_2_abs_iso     "tree_2_rep`(tree_2_abs`x) = x"
  tree_2_rep_iso     "tree_2_abs`(tree_2_rep`x) = x"
  tree_2_when_def    "tree_2_when ≡ Λf x. f`(tree_2_rep`x)"
  tree_2_copy_def    "tree_2_copy ≡ Λf  . tree_2_abs oo cfst`f oo tree_2_rep"
  #_def              "#             ≡ Λt. tree_2_abs`t"
  is_#_def           "is_#          ≡     tree_2_when`(Λt. TT)"
  tree_2_sel_1_1_def "tree_2_sel_1_1 ≡     tree_2_when`(Λt. t )"
```

```
branch_abs_iso     "branch_rep‘(branch_abs‘x) = x"
branch_rep_iso     "branch_abs‘(branch_rep‘x) = x"
branch_when_def    "branch_when ≡ Λf x.
                    ssplit‘(Λa. ssplit‘(Λt b. f‘a‘t‘(lift‘ID‘b)))‘(branch_rep‘x)"
branch_copy_def    "branch_copy ≡ Λf. branch_abs oo ssplit‘(Λx y. (|x, ssplit‘(Λx y.
                    (|cfst‘(csnd‘f)‘x, lift‘(up oo csnd‘(csnd‘f))‘y|))‘y|)) oo branch_rep"
con_def            "con    ≡ Λa t b. branch_abs‘(|a, t, up‘b|)"
is_con_def         "is_con ≡         branch_when‘(Λa t b. TT)"
val_def            "val    ≡         branch_when‘(Λa t b. a)"
bhd_def            "bhd    ≡         branch_when‘(Λa t b. t)"
btl_def            "btl    ≡         branch_when‘(Λa t b. b)"

tree_1_tree_2_branch_copy_def  "tree_1_tree_2_branch_copy ≡
                   Λf. <tree_1_copy‘f, tree_2_copy‘f, branch_copy‘f>"
tree_1_take_def "tree_1_take ≡ λn.       cfst‘(iterate n tree_1_tree_2_branch_copy ⊥) "
tree_2_take_def "tree_2_take ≡ λn. cfst‘(csnd‘(iterate n tree_1_tree_2_branch_copy ⊥))"
branch_take_def "branch_take ≡ λn. csnd‘(csnd‘(iterate n tree_1_tree_2_branch_copy ⊥))"
tree_1_reach        "cfst‘(fix‘tree_1_tree_2_branch_copy) ‘x1 = x1"
tree_2_reach  "cfst‘(csnd‘(fix‘tree_1_tree_2_branch_copy))‘x2 = x2"
branch_reach  "csnd‘(csnd‘(fix‘tree_1_tree_2_branch_copy))‘x3 = x3"
tree_1_finite_def  "tree_1_finite ≡ λx1. ∃n. tree_1_take n‘x1 = x1"
tree_2_finite_def  "tree_2_finite ≡ λx2. ∃n. tree_2_take n‘x2 = x2"
branch_finite_def  "branch_finite ≡ λx3. ∃n. branch_take n‘x3 = x3"
tree_1_tree_2_branch_bisim_def  "tree_1_tree_2_branch_bisim ≡ λR.
     (∀x1 x1’. fst R x1 x1’ −→             x1 = ⊥          ∧ x1’ = ⊥        ∨
                                           x1 = leaf       ∧ x1’ = leaf     ∨
         (∃b b’       . b ≠ ⊥ ∧ b’ ≠ ⊥ ∧
                        snd (snd R) b b’  ∧ x1 = node‘b     ∧ x1’ = node‘b’) ∨
         (∃b t        . b ≠ ⊥ ∧ t  ≠ ⊥  ∧ x1 = (b in_f t) ∧ x1’ = (b in_f t))) ∧
     (∀x2 x2’. fst (snd R) x2 x2’ −→       x2 = ⊥          ∧ x2’ = ⊥        ∨
         (∃t t’       . t ≠ ⊥ ∧ t’ ≠ ⊥ ∧
                        fst R t t’       ∧ x2 = #‘t        ∧ x2’ = #‘t’))       ∧
     (∀x3 x3’. snd (snd R) x3 x3’ −→       x3 = ⊥          ∧ x3’ = ⊥        ∨
         (∃a t b t’ b’. a ≠ ⊥ ∧ t ≠ ⊥ ∧ t’ ≠ ⊥ ∧ fst (snd R) t t’ ∧ snd (snd R) b b’
                                        ∧ x3 = (<a:t|>>b) ∧ x3’ = <a:t’|>>b’))"
Tree_Forest_induct
 "[[ P1 ⊥; P2 ⊥;          eTree       ≠ ⊥   =⇒ P1  eTree;
    ⋀a F. [[ P2 F;        mTree‘a‘F   ≠ ⊥ ]] =⇒ P1 (mTree‘a‘F);
                          eForest     ≠ ⊥   =⇒ P2  eForest;
    ⋀T F. [[ P1 T; P2 F; mForest‘T‘F ≠ ⊥ ]] =⇒ P2 (mForest‘T‘F) ]] =⇒ P1 x1 ∧ P2 x2"

branch_induct
 "[[ adm P; P ⊥; ⋀a t b. [[ P b; <a:t|>>b ≠ ⊥ ]] =⇒ P <a:t|>>b ]] =⇒ P x"

end
```

Abbildung 5.4: Beispiel: erzeugte Theorie

Die auf dieser Seite beginnende Abbildung 5.5 zeigt zum einen die Ausgabe des Datentyp-Pakets bei der Verarbeitung der Datentypspezifikationen. Nachdem die Beweise für die Datentyp-Eigenschaften einige Sekunden lang dauern, wird für jeden Teiltyp eine entsprechende Meldung angezeigt. Man beachte auch die Warnung für den leeren Datentyp.

Zum anderen zeigt die Abbildung die ML-Ausgabe der erzeugten Strukturen, etwas umgestellt, gekürzt und optisch aufbereitet. Daraus wird deutlich, daß die ML-Struktur `Example` den Inhalt der gesamten Datei `Example` repräsentiert: Ihre Komponenten sind die Isabelle-Theorie `thy`, die Hilfsvariablen für das Datentyp-Paket `empty_equations` und `tree_1_tree_2_branch_equations`, die Substrukturen `empty`, `tree_1`, `tree_2`, `branch` und `tree_1_tree_2_branch` sowie die Induktionsaxiome `Tree_Forest_induct` und `branch_induct`. Die fünf Substrukturen enthalten die in den beiden vorangehenden Abschnitten genannten Axiome, Definitionen und Theoreme der entsprechenden Datentypen, jeweils an ML-Namen zur Verwendung in Beweisen gebunden.

```
> time_use_thy "Example";

**** Starting Theory Example ****
Reading "Example.thy"
Loading theory "Example"
val thy = {ProtoPure, CPure, HOL, Ord, Set, Prod, Lfp, Relation, Trancl, WF,
  Nat, Holcfb, Void, Porder0, Porder, Pcpo, Fun1, Fun2, Fun3, Cont, Cfun1,
  Cfun2, Cfun3, Cprod1, Cprod2, Cprod3, Sprod0, Sprod1, Sprod2, Sprod3,
  Ssum0, Ssum1, Ssum2, Ssum3, Sum, Lift1, Lift2, Lift3, Fix, ccc1, One, Tr1,
  Tr2} : theory

Building new grammar...
Building new grammar...
Building new grammar...
Proving isomorphism properties of domain empty...
Proving induction properties of domain empty...
WARNING: domain empty is empty!
Proving isomorphism properties of domain tree_1...
Proving isomorphism properties of domain tree_2...
Proving isomorphism properties of domain branch...
Proving induction properties of domain tree_1_tree_2_branch...
```

```
structure Example : sig
   val thy : theory
   val empty_equations : eq list
   structure empty :
      sig
         val abs_iso : thm
         val rep_iso : thm
         val when_def : thm
         val con_defs : thm list
         val dis_defs : thm list
         val sel_defs : thm list
         val copy_def : thm
         val iso_rews : thm list
         val exhaust : thm
         val cases : thm
         val when_rews : thm list
         val con_rews : thm list
         val sel_rews : thm list
         val dis_rews : thm list
         val dists_le : thm list
         val dists_eq : thm list
         val inverts : thm list
         val injects : thm list
         val copy_rews : thm list
         val reach : thm
         val take_def : thm
         val finite_def : thm
         val bisim_def : thm
         val take_rews : thm list
         val take_lemma : thm
         val finite : thm
         val finite_ind : thm
         val ind : thm
         val coind : thm
         val rews : thm list
      end
   val tree_1_tree_2_branch_equations : eq list
   structure tree_1 :
      sig
         val abs_iso : thm
         val rep_iso : thm
         val when_def : thm
         val con_defs : thm list
         val dis_defs : thm list
         val sel_defs : thm list
         val copy_def : thm
```

```
         val iso_rews : thm list
         val exhaust : thm
         val cases : thm
         val when_rews : thm list
         val con_rews : thm list
         val sel_rews : thm list
         val dis_rews : thm list
         val dists_le : thm list
         val dists_eq : thm list
         val inverts : thm list
         val injects : thm list
         val copy_rews : thm list
         val rews : thm list
      end
   structure tree_2 : (* ... *)
   structure branch : (* ... *)
   structure tree_1_tree_2_branch :
      sig
         val copy_def : thm
         val reachs : thm list
         val take_defs : thm
         val finite_defs : thm
         val bisim_def : thm
         val take_rews : thm list
         val take_lemmas : thm list
         val finites : thm list
         val finite_ind : thm
         val ind : thm
         val coind : thm
         val rews : thm list
      end
   val Tree_Forest_induct : thm
   val branch_induct : thm
end

*** Finished Theory Example ****
Process time (incl GC): 229.3 secs
val it = () : unit
```

```
> open Example;
val thy =
   {ProtoPure, CPure, HOL, Ord, Set, Prod, Lfp, Relation, Trancl, WF, Nat,
     Holcfb, Void, Porder0, Porder, Pcpo, Fun1, Fun2, Fun3, Cont, Cfun1,
     Cfun2, Cfun3, Cprod1, Cprod2, Cprod3, Sprod0, Sprod1, Sprod2, Sprod3,
     Ssum0, Ssum1, Ssum2, Ssum3, Sum, Lift1, Lift2, Lift3, Fix, ccc1, One,
     Tr1, Tr2, Example} : theory
val empty_equations =
   [(("empty", []),
      [("silly"  , [((false,  0), "empty_sel_1_1" , "e")])])] : eq list
structure empty : (* ... *)
val tree_1_tree_2_branch_equations =
   [(("tree_1", ["'a", "'b"]),
      [("leaf"   , []),
       ("node"   , [((false,  2), "branch"        , "b")]),
       ("op in_f", [((false, ~1), "tree_1_sel_3_1", "b"),
                    ((false, ~1), "tree_1_sel_3_2", "t")])]),
    (("tree_2", ["'b", "'a"]),
      [("#"      , [((false,  0), "tree_2_sel_1_1", "t")])]),
    (("branch", ["'a", "'b"]),
      [("con"    , [((false, ~1), "val"           , "a"),
                    ((false , 1), "bhd"           , "t"),
                    ((true ,  2), "btl"           , "b")])])] : eq list
structure tree_1 : (* ... *)
structure tree_2 : (* ... *)
structure branch : (* ... *)
structure tree_1_tree_2_branch : (* ... *)
val Tree_Forest_induct =
   "⟦ P1 ⊥; P2 ⊥;
                          eTree       ≠ ⊥   ⟹ P1  eTree;
       ⋀a F. ⟦ P2 F;       mTree`a`F   ≠ ⊥ ⟧ ⟹ P1 (mTree`a`F);
                          eForest     ≠ ⊥   ⟹ P2  eForest;
       ⋀T F. ⟦ P1 T; P2 F; mForest`T`F ≠ ⊥ ⟧ ⟹ P2 (mForest`T`F) ⟧ ⟹
    P1 x1 ∧ P2 x2" : thm
val branch_induct =
   "⟦ adm P; P ⊥; ⋀a t b. ⟦ P b; <a:t|>>b ≠ ⊥ ⟧ ⟹ P <a:t|>>b ⟧ ⟹ P x"
: thm
```

```
> open empty;
val abs_iso   = (* ... *)
⋮
val copy_def  = (* ... *)
val iso_rews  = ["empty_rep‘(empty_abs‘x) = x", "empty_abs‘(empty_rep‘x) = x",
                 "empty_abs‘⊥ = ⊥", "empty_rep‘⊥ = ⊥"] : thm list
val exhaust   =  "x = ⊥      ∨ (∃e.   x = silly‘e ∧ e ≠ ⊥)" : thm
val cases     = "⟦ x = ⊥ ⟹ P;  ⋀e. ⟦ x = silly‘e ; e ≠ ⊥ ⟧ ⟹ P ⟧ ⟹ P" : thm
val when_rews = ["f‘⊥ = ⊥  ⟹ empty_when‘f‘⊥ = ⊥",
                 "e ≠ ⊥   ⟹ empty_when‘f‘(silly‘e) = f‘e"] : thm list
val con_rews  = ["silly‘⊥ = ⊥", "e ≠ ⊥ ⟹ silly‘e ≠ ⊥"] : thm list
val sel_rews  = ["empty_sel_1_1‘⊥ = ⊥",
                 "x ≠ ⊥ ⟹ empty_sel_1_1‘x ≠ ⊥",
                           "empty_sel_1_1‘(silly‘e) = e"] : thm list
val dis_rews  = ["is_silly‘⊥ ≠ ⊥",
                 "x ≠ ⊥ ⟹ is_silly‘x ≠ ⊥",
                 "⌈is_silly‘(silly‘e)⌉"] : thm list
val dists_le  = [] : thm list
val dists_eq  = [] : thm list
val inverts   = ["⟦ silly‘e ⊑ silly‘e’; e ≠ ⊥; e’ ≠ ⊥ ⟧ ⟹ e ⊑ e’"] : thm list
val injects   = ["⟦ silly‘e = silly‘e’; e ≠ ⊥; e’ ≠ ⊥ ⟧ ⟹ e = e’"] : thm list
val copy_rews = ["f‘⊥ = ⊥ ⟹ empty_copy‘f‘⊥ = ⊥",
                 "e ≠ ⊥  ⟹ empty_copy‘f‘(silly‘e) = silly‘(f‘e)",
                            "empty_copy‘⊥‘(silly‘e) = ⊥"] : thm list


val reach       = (* ... *)
⋮
val bisim_def   = (* ... *)
val take_rews   = ["empty_take n‘⊥ = ⊥",
                   "empty_take 0‘x = ⊥",
                   "empty_take (Suc n)‘(silly‘x) = silly‘(empty_take n‘x)"] : thm list
val take_lemma  = "(⋀n. empty_take n‘x = empty_take n‘x’) ⟹ x = x’" : thm
val finite      = "empty_finite x" : thm
val finite_ind  = "⟦ P ⊥; ⋀e. ⟦ e ≠ ⊥; P e ⟧ ⟹ P (silly‘e) ⟧ ⟹
                   P (empty_take n‘x)" : thm
val ind         = "⟦ P ⊥;  ⋀e. ⟦ e ≠ ⊥; P e ⟧ ⟹ P (silly‘e) ⟧ ⟹ P x" : thm
val coind       = "⟦ empty_bisim R; R x x’ ⟧ ⟹ x = x’" : thm
val rews        = (* ... *)
```

```
> open tree_1;
val abs_iso   = (* ... *)
⋮
val copy_def  = (* ... *)
val iso_rews  = ["tree_1_rep‘(tree_1_abs‘x) = x", "tree_1_abs‘(tree_1_rep‘x) = x",
                 "tree_1_abs‘⊥ = ⊥", "tree_1_rep‘⊥ = ⊥"] : thm list
val exhaust   =          "x = ⊥                              ∨
                          x = leaf                           ∨
                 (∃b   .  x = node‘b      ∧ b ≠ ⊥)           ∨
                 (∃b t.   x = (b in_f t) ∧ b ≠ ⊥ ∧ t ≠ ⊥)" : thm
val cases     = "⟦        x = ⊥                              ⟹ P;
                          x = leaf                           ⟹ P;
                  ⋀b.   ⟦ x = node‘b      ; b ≠ ⊥         ⟧ ⟹ P;
                  ⋀b t. ⟦ x = (b in_f t) ; b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P ⟧ ⟹ P" : thm
val when_rews = ["tree_1_when‘f1‘f2‘f3‘⊥ = ⊥",
                                  "tree_1_when‘f1‘f2‘f3‘ leaf      = f1",
                 "  b ≠ ⊥          ⟹ tree_1_when‘f1‘f2‘f3‘(node‘b  ) = f2‘b",
                 "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ tree_1_when‘f1‘f2‘f3‘(b in_f t) = f3‘b‘t"]
                : thm list
val con_rews  =  ["node‘⊥     = ⊥",
                  "(⊥ in_f t) = ⊥",
                  "(b in_f ⊥) = ⊥",
                                   "leaf      ≠ ⊥",
                   "b ≠ ⊥          ⟹  node‘b    ≠ ⊥",
                 "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ (b in_f t) ≠ ⊥"] : thm list
val sel_rews  = ["branch        ‘⊥ = ⊥",
                 "tree_1_sel_3_1‘⊥ = ⊥",
                 "tree_1_sel_3_2‘⊥ = ⊥",
                           "branch        ‘ leaf      = ⊥",
                           "branch        ‘(node‘b  ) = b",
                           "branch        ‘(b in_f t) = ⊥",
                           "tree_1_sel_3_1‘ leaf      = ⊥",
                           "tree_1_sel_3_1‘(node‘b)   = ⊥",
                 "t ≠ ⊥ ⟹ tree_1_sel_3_1‘(b in_f t) = b",
                           "tree_1_sel_3_2‘ leaf      = ⊥",
                           "tree_1_sel_3_2‘(node‘b)   = ⊥",
                 "b ≠ ⊥ ⟹ tree_1_sel_3_2‘(b in_f t) = t"] : thm list
val dis_rews  = ["is_leaf‘⊥ = ⊥",
                 "is_node‘⊥ = ⊥",
                 "is_in_f‘⊥ = ⊥",
                 "x ≠ ⊥ ⟹ is_leaf‘x ≠ ⊥",
                 "x ≠ ⊥ ⟹ is_node‘x ≠ ⊥",
                 "x ≠ ⊥ ⟹ is_in_f‘x ≠ ⊥",
                                 "⌈is_leaf‘ leaf     ⌉",
                   "b ≠ ⊥          ⟹ ⌊is_leaf‘(node‘b)  ⌋",
                 "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ ⌊is_leaf‘(b in_f t)⌋",
```

```
                                           "⌊is_node‘ leaf     ⌋",
                  "b ≠ ⊥           ⟹ ⌈is_node‘(node‘b)  ⌉",
                "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ ⌊is_node‘(b in_f t)⌋",
                                           "⌊is_in_f‘ leaf     ⌋",
                  "b ≠ ⊥           ⟹ ⌊is_in_f‘(node‘b)  ⌋",
                "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ ⌈is_in_f‘(b in_f t)⌉"] : thm list
val dists_le  = [                          "¬ leaf   ⊑ node‘b",
                  "b ≠ ⊥           ⟹ ¬ node‘b ⊑ leaf",
                                           "¬ leaf       ⊑ (b in_f t)",
                "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ ¬ (b in_f t) ⊑ leaf",
                  "b ≠ ⊥           ⟹ ¬ node‘b     ⊑ (b in_f t)",
                "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ ¬ (b in_f t) ⊑ node‘b"] : thm list
val dists_eq  = [                           "leaf    ≠ node‘b1",
                                            "node‘b2 ≠ leaf",
                                            "leaf        ≠ (b in_f t1)",
                                            "(b in_f t2) ≠ leaf",
                  "b1 ≠ ⊥           ⟹ node‘b1     ≠ (b in_f t1)",
                "⟦ b ≠ ⊥; t1 ≠ ⊥ ⟧ ⟹ (b in_f t1) ≠ node‘b1"] : thm list
val inverts   =
   ["⟦ node‘b ⊑ node‘b’; b ≠ ⊥; b’ ≠ ⊥ ⟧                          ⟹ b ⊑ b’",
    "⟦ (b in_f t) ⊑ (b’ in_f t’); b ≠ ⊥; t ≠ ⊥; b’ ≠ ⊥; t’ ≠ ⊥ ⟧ ⟹ b ⊑ b’ ∧ t ⊑ t’"]
   : thm list
val injects   =
   ["⟦ node‘b = node‘b’; b ≠ ⊥; b’ ≠ ⊥ ⟧                          ⟹ b = b’",
    "⟦ (b in_f t) = (b’ in_f t’); b ≠ ⊥; t ≠ ⊥; b’ ≠ ⊥; t’ ≠ ⊥ ⟧ ⟹ b = b’ ∧ t = t’"]
   : thm list
val copy_rews = ["tree_1_copy‘f‘⊥ = ⊥",
                          "tree_1_copy‘f‘ leaf      = leaf",
                "b ≠ ⊥ ⟹  tree_1_copy‘f‘(node‘b  ) = node‘(csnd‘(csnd‘f)‘b)",
                          "tree_1_copy‘f‘(b in_f t) = (b in_f t)",
                          "tree_1_copy‘⊥‘(node‘b  ) = ⊥"] : thm list
val rews      = (* ... *)


> open tree_2;
val abs_iso   = (* ... *)
⋮
val copy_def  = (* ... *)
val iso_rews  = ["tree_2_rep‘(tree_2_abs‘x) = x", "tree_2_abs‘(tree_2_rep‘x) = x",
                 "tree_2_abs‘⊥ = ⊥", "tree_2_rep‘⊥ = ⊥"] : thm list
val exhaust   =   "x = ⊥      ∨ (∃t.   x = #‘t ∧ t ≠ ⊥)" : thm
val cases     = "⟦ x = ⊥ ⟹ P;  ⋀t. ⟦ x = #‘t ; t ≠ ⊥ ⟧ ⟹ P ⟧ ⟹ P" : thm
val when_rews = ["f‘⊥ = ⊥ ⟹ tree_2_when‘f‘⊥ = ⊥",
                  "t ≠ ⊥ ⟹ tree_2_when‘f‘(#‘t) = f‘t"] : thm list
```

```
val con_rews  = ["#‘⊥ = ⊥", "t ≠ ⊥ ⟹ #‘t ≠ ⊥"] : thm list
val sel_rews  = ["tree_2_sel_1_1‘⊥ = ⊥", "x ≠ ⊥ ⟹ tree_2_sel_1_1‘x ≠ ⊥",
                 "tree_2_sel_1_1‘(#‘t) = t"] : thm list
val dis_rews  = ["is_#‘⊥ ≠ ⊥", "x ≠ ⊥ ⟹ is_#‘x ≠ ⊥", "t ≠ ⊥ ⟹ ⌈is_#‘(#‘t)⌉"]
                : thm list
val dists_le  = [] : thm list
val dists_eq  = [] : thm list
val inverts   = ["⟦ #‘t ⊑ #‘t’; t ≠ ⊥; t’ ≠ ⊥ ⟧ ⟹ t ⊑ t’"] : thm list
val injects   = ["⟦ #‘t = #‘t’; t ≠ ⊥; t’ ≠ ⊥ ⟧ ⟹ t = t’"] : thm list
val copy_rews = ["cfst‘f‘⊥ = ⊥ ⟹ tree_2_copy‘f‘⊥ = ⊥",
                        "t ≠ ⊥ ⟹ tree_2_copy‘f‘(#‘t) = #‘(cfst‘f‘t)",
                                  "tree_2_copy‘⊥‘(#‘t) = ⊥"] : thm list
val rews = (* ... *)


> open branch;
val abs_iso   = (* ... *)
⋮
val copy_def  = (* ... *)
val iso_rews  = ["branch_rep‘(branch_abs‘x) = x", "branch_abs‘(branch_rep‘x) = x",
                 "branch_abs‘⊥ = ⊥", "branch_rep‘⊥ = ⊥"]
                : thm list
val exhaust   =  "x = ⊥     ∨ (∃a t b.   x = <a:t|>>b ∧ a ≠ ⊥ ∧ t ≠ ⊥)" : thm
val cases     = "⟦ x = ⊥ ⟹ P;  ⋀a t b. ⟦ x = <a:t|>>b ; a ≠ ⊥ ; t ≠ ⊥ ⟧ ⟹ P ⟧ ⟹ P"
                : thm
val when_rews = ["branch_when‘f‘⊥ = ⊥",
                 "⟦ a ≠ ⊥; t ≠ ⊥ ⟧ ⟹ branch_when‘f‘<a:t|>>b = f‘a‘t‘b"] : thm list
val con_rews  = ["<⊥:t|>>b = ⊥", "<a:⊥|>>b = ⊥",
                 "⟦ a ≠ ⊥; t ≠ ⊥ ⟧ ⟹ <a:t|>>b ≠ ⊥"] : thm list
val sel_rews  = ["val‘⊥ = ⊥", "bhd‘⊥ = ⊥", "btl‘⊥ = ⊥",
                 "x ≠ ⊥ ⟹ val‘x ≠ ⊥", "x ≠ ⊥ ⟹ bhd‘x ≠ ⊥",
                   "t ≠ ⊥          ⟹ val‘<a:t|>>b = a",
                   "a ≠ ⊥          ⟹ bhd‘<a:t|>>b = t",
                 "⟦ a ≠ ⊥; t ≠ ⊥ ⟧ ⟹ btl‘<a:t|>>b = b"] : thm list
val dis_rews  = ["is_con‘⊥ = ⊥", "x ≠ ⊥ ⟹ is_con‘x ≠ ⊥",
                 "⟦ a ≠ ⊥; t ≠ ⊥ ⟧ ⟹ ⌈is_con‘<a:t|>>b⌉"] : thm list
val dists_le  = [] : thm list
val dists_eq  = [] : thm list
val inverts   = ["⟦ <a:t|>>b ⊑ <a’:t’|>>b’; a ≠ ⊥; t ≠ ⊥; a’ ≠ ⊥; t’ ≠ ⊥ ⟧ ⟹
                  a ⊑ a’ ∧ t ⊑ t’ ∧ b ⊑ b’"] : thm list
val injects   = ["⟦ <a:t|>>b = <a’:t’|>>b’; a ≠ ⊥; t ≠ ⊥; a’ ≠ ⊥; t’ ≠ ⊥ ⟧ ⟹
                  a = a’ ∧ t = t’ ∧ b = b’"] : thm list
val copy_rews = ["branch_copy‘f‘⊥ = ⊥", "t ≠ ⊥ ⟹
                  branch_copy‘f‘<a:t|>>b = <a:cfst‘(csnd‘f)‘t|>>csnd‘(csnd‘f)‘b",
                 "branch_copy‘⊥‘<a:t|>>b = ⊥"] : thm list
val rews      = (* ... *)
```

```
> open tree_1_tree_2_branch;
val copy_def    = (* ... *)
⋮
val bisim_def   = (* ... *)
val take_rews   =
   ["tree_1_take n‘⊥  = ⊥", "tree_2_take n‘⊥  = ⊥", "branch_take n‘⊥  = ⊥",
    "tree_1_take 0‘x1 = ⊥", "tree_2_take 0‘x2 = ⊥", "branch_take 0‘x3 = ⊥",
    "tree_1_take (Suc n)‘ leaf      = leaf",
    "tree_1_take (Suc n)‘(node‘b)   = node‘(branch_take n‘b)",
    "tree_1_take (Suc n)‘(b in_f t) = (b in_f t)",
    "tree_2_take (Suc n)‘(#‘t)      = #‘(tree_1_take n‘t)",
    "branch_take (Suc n)‘<a:t|>>b   = <a:tree_2_take n‘t|>>branch_take n‘b"]
   : thm list
val take_lemmas =
   ["(⋀n. tree_1_take n‘x1 = tree_1_take n‘x1’) ⟹ x1 = x1’",
    "(⋀n. tree_2_take n‘x2 = tree_2_take n‘x2’) ⟹ x2 = x2’",
    "(⋀n. branch_take n‘x3 = branch_take n‘x3’) ⟹ x3 = x3’"] : thm list
val finites     =
   ["¬ tree_1_finite x1 ∨ tree_1_finite x1",
    "¬ tree_2_finite x2 ∨ tree_2_finite x2",
    "¬ branch_finite x3 ∨ branch_finite x3"] : thm list
val finite_ind  =
   "⟦ P1 ⊥;                                         P1 leaf;
          ⋀b.     ⟦ b ≠ ⊥;        P3 b       ⟧ ⟹ P1 (node‘b);
          ⋀b t.   ⟦ b ≠ ⊥; t ≠ ⊥             ⟧ ⟹ P1 (b in_f t);
      P2 ⊥; ⋀t.   ⟦ t ≠ ⊥;        P1 t       ⟧ ⟹ P2 (#‘t);
      P3 ⊥; ⋀a t b. ⟦ a ≠ ⊥; t ≠ ⊥; P2 t; P3 b ⟧ ⟹ P3 <a:t|>>b ⟧ ⟹
    P1 (tree_1_take n‘x1) ∧ P2 (tree_2_take n‘x2) ∧ P3 (branch_take n‘x3)" : thm
val ind         =
   "⟦ adm P1; adm P2; adm P3;
      P1 ⊥;                                         P1 leaf;
          ⋀b.     ⟦ b ≠ ⊥;        P3 b       ⟧ ⟹ P1 (node‘b);
          ⋀b t.   ⟦ b ≠ ⊥; t ≠ ⊥             ⟧ ⟹ P1 (b in_f t);
      P2 ⊥; ⋀t.   ⟦ t ≠ ⊥;        P1 t       ⟧ ⟹ P2 (#‘t);
      P3 ⊥; ⋀a t b. ⟦ a ≠ ⊥; t ≠ ⊥; P2 t; P3 b ⟧ ⟹ P3 <a:t|>>b ⟧ ⟹
    P1 x1 ∧ P2 x2 ∧ P3 x3" : thm
val coind       =
   "⟦ tree_1_tree_2_branch_bisim R;
      fst R x1 x1’; fst (snd R) x2 x2’; snd (snd R) x3 x3’ ⟧ ⟹
    x1 = x1’ ∧ x2 = x2’ ∧ x3 = x3’" : thm
val rews        = (* ... *)
```

Abbildung 5.5: Beispiel: erzeugte Theoreme

Kapitel 6

Beweis der Theoreme

Dieses Kapitel gibt Einblick in die Techniken, die das Datentyp-Paket beim Beweis der Theoreme für einen durch Bereichsgleichungen beschriebenen Datentyp verwendet. Die *allgemeinen* Beweistaktiken können der Datei `theorems.ML` (im Anhang B abgedruckt) entnommen werden, während hier eine schrittweise Darstellung ausgewählter Beweise erfolgt, und zwar anhand des Beispiels aus dem vorigen Kapitel, weil eine allgemeine, generische Formulierung recht technisch und nicht besonders instruktiv wäre. Hier kommt es in erster Linie auf die grundsätzliche Wahl der Beweisverfahren an, wie Formel- bzw. Termvereinfachung (`simp_tac` usw.), Anwendung (z.T. schrittweise) des Classical Reasoners (`fast_tac`, `safe_tac` u.ä.), Induktion, Fallunterscheidung oder einzelne Resolutionsschritte.

6.1 Hilfsfunktionen

Zunächst ein paar Funktionen, die die Formulierung des Beweisanfangs (mit Einsetzung der Prämissen) und der interaktiven Taktik-Anwendung (ohne Klammerung der Taktik-Terme) vereinfachen.

```
> fun goalprems thy t = by(cut_facts_tac (goal thy t) 1);

> fun pg thy defs t tacsf = prove_goalw thy defs t
                              (fn []    => tacsf
                               |  prems => (cut_facts_tac prems 1)::tacsf);

> infixr 0 y;         (* zur Verwendung in b y tac ...  statt by(tac ...) *)
> fun _ y t = by t;   (* rechtes Argument hat niedrigstmögliche Priorität *)
> val b = 0;          (* beliebiger Wert für linkes Argument *)
```

Hilfstaktik für Sonderbehandlung des undefinierten Falles

```
> fun case_UU_tac rews i v = res_inst_tac [("Q",v^" = UU")] classical2 i THEN
                             asm_simp_tac (HOLCF_ss addsimps rews) i;
```

Hilfstaktik zum Beweis der Ketteneigenschaft

```
> val chain_tac = REPEAT_DETERM o resolve_tac [is_chain_iterate, ch2ch_fappR,
                                                                  ch2ch_fappL];
```

Hilfstaktik zur Einführung von Allquantoren für Induktionstheoreme

```
> fun quant_tac i = EVERY(mapn (fn n => fn _ => res_inst_tac [("x",x_name n)] spec i)
                              1 dnames);
```

6.2 Lemmata

Dieser Abschnitt enthält diverse Lemmata, die sich mittels Prädikatenlogik und/oder einfachen Eigenschaften von LCF-Operatoren ergeben.

```
> val quant_ss = HOL_ss addsimps (map
                        (fn s => prove_goal HOL.thy s (fn _ => [fast_tac HOL_cs 1]))
                 ["(∀x. P x ∧ Q) = ((∀x. P x) ∧ Q)",
                  "(∀x. P ∧ Q x) = (P ∧ (∀x. Q x))"]);

> val swap3 = prove_goal HOL.thy "⟦ Q ⟹ P; ¬ P ⟧ ⟹ ¬ Q" (fn prems => [
                                cut_facts_tac prems 1,
                                etac swap 1,
                                dtac notnotD 1,
                                etac (hd prems) 1]);

> val prems = goal Porder.thy "¬ x ⊑ y ⟹ x ≠ y";
Level 0
x ≠ y
 1. x ≠ y

> b y rtac swap3 1; (* Widerspruchsbeweis durch Annahme des Gegenteils *)
Level 1
x ≠ y
 1. x = y ⟹ ?P
 2. ¬ ?P

> b y etac (antisym_less_inverse RS conjunct1) 1; (* "?x1 = ?y1 ⟹ ?x1 ⊑ ?y1" : thm *)
Level 2
x ≠ y
 1. ¬ x ⊑ y

> b y resolve_tac prems 1;
Level 3
x ≠ y
No subgoals!

> val dist_eqI = result();

> val cfst_strict  = prove_goal Cprod3.thy "cfst‘⊥ = ⊥" (fn _ => [
                                (simp_tac (HOLCF_ss addsimps [inst_cprod_pcpo2]) 1)]);
```

Es folgt je ein exemplarischer Beweis für alle auftretenden Arten von Theoremen, in Form einer an den entscheidenden Stellen kommentierten Interaktion mit dem Beweissystem Isabelle.
Ab hier wird bei den Beweisen die interaktive Ausgabe von Isabelle verkürzt dargestellt, ohne Ausgabe der Beweisschritt-Nummer und der Zielformel in jedem Schritt, und ohne das Binden des Theorems an einen ML-Namen. Der Name des entsprechenden Theorems wird stattdessen beim Beginn des Beweises rechts außen als Kommentar erwähnt.

6.3 Isomorphismusanteil

6.3.1 Isomorphismus

Isomorphie-Axiome und Definitionen aus der Theorie, hier für den Typ `tree_1`

```
val ax_abs_iso = "tree_1_rep‘(tree_1_abs‘?x) = ?x" : thm
val ax_rep_iso = "tree_1_abs‘(tree_1_rep‘?x) = ?x" : thm
val ax_when_def = "tree_1_when ≡ Λf1 f2 f3 x.
                   sswhen‘(Λdummy. f1)‘(sswhen‘f2‘(ssplit‘f3))‘(tree_1_rep‘x)" : thm
val axs_con_def = ["leaf    ≡       tree_1_abs‘(sinl‘ one        )",
                   "node    ≡ Λb.   tree_1_abs‘(sinr‘(sinl‘ b    ))",
                   "op in_f ≡ Λb t. tree_1_abs‘(sinr‘(sinr‘(|b, t|)))"] : thm list
val axs_dis_def = ["is_leaf ≡ tree_1_when‘TT‘(Λb. FF)‘(Λb t. FF)",
                   "is_node ≡ tree_1_when‘FF‘(Λb. TT)‘(Λb t. FF)",
                   "is_in_f ≡ tree_1_when‘FF‘(Λb. FF)‘(Λb t. TT)"] : thm list
val axs_sel_def = ["branch ≡ tree_1_when‘⊥‘(Λb. b)‘⊥",
                   "tree_1_sel_3_1 ≡ tree_1_when‘⊥‘⊥‘(Λb t. b)",
                   "tree_1_sel_3_2 ≡ tree_1_when‘⊥‘⊥‘(Λb t. t)"] : thm list
val ax_copy_def = "tree_1_copy ≡  Λf.tree_1_abs oo
                   sswhen‘sinl‘(sinr oo sswhen‘(sinl oo csnd‘(csnd‘f))‘(sinr oo
                   split‘(Λx y. (|x, y|))) oo tree_1_rep" : thm

> val (rep_strict, abs_strict) = let
             val r = ax_rep_iso RS (ax_abs_iso RS (allI  RSN(2,allI RS iso_strict)))
             in (r RS conjunct1, r RS conjunct2) end;
val rep_strict = "tree_1_rep‘⊥ = ⊥" : thm
val abs_strict = "tree_1_abs‘⊥ = ⊥" : thm


> goalprems thy "tree_1_abs‘x = ⊥ ⟹ x = ⊥";                        (* abs_defin’ *)
 1. tree_1_abs‘x = ⊥ ⟹ x = ⊥

> b y res_inst_tac [("t","x")] (ax_abs_iso RS subst) 1;
 1. tree_1_abs‘x = ⊥ ⟹ tree_1_rep‘(tree_1_abs‘x) = ⊥

> b y asm_simp_tac (HOL_ss addsimps [rep_strict]) 1;
No subgoals!


> goalprems thy "tree_1_rep‘x = y ⟹ x = tree_1_abs‘y";                (* iso_swap *)
 1. tree_1_rep‘x = y ⟹ x = tree_1_abs‘y

> b y dres_inst_tac [("f","tree_1_abs")] cfun_arg_cong 1;
 1. tree_1_abs‘(tree_1_rep‘x) = tree_1_abs‘y ⟹ x = tree_1_abs‘y

> b y etac (ax_rep_iso RS subst) 1;
No subgoals!
```

6.3.2 Fallunterscheidung und Ausschöpfung

```
> val prems = goal thy "⟦ x = ⊥ ⟹ P; x = leaf ⟹ P;                        (* cases *)
  ⋀b. ⟦ x = node‘b; b ≠ ⊥ ⟧ ⟹ P; ⋀b t. ⟦ x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P ⟧ ⟹ P";
  b y cut_facts_tac [excluded_middle] 1 THEN etac disjE 1;
 1. ¬ ?P1 ⟹ P
 2. ?P1 ⟹ P
```

Behandlung des undefinierten Falls

```
> b y rtac (hd prems) 2;
 1. ¬ ?P1 ⟹ P
 2. ?P1 ⟹ x = ⊥

> b y etac rep_defin’ 2;
 1. tree_1_rep‘x ≠ ⊥ ⟹ P
```

Weiter nach der Struktur des Funktorterms `<++, K_one, <++, P2 o P2, <**, K_’b, K_tr> > >`
Fallunterscheidung für die Summe

```
> b y rtac ssumE 1;
 1. ⟦ tree_1_rep‘x ≠ ⊥; ?p4 = ⊥ ⟧ ⟹ P
 2. ⋀xa. ⟦ tree_1_rep‘x ≠ ⊥; ?p4 = sinl‘xa; xa ≠ ⊥ ⟧ ⟹ P
 3. ⋀y.  ⟦ tree_1_rep‘x ≠ ⊥; ?p4 = sinr‘y; y ≠ ⊥ ⟧ ⟹ P

> b y contr_tac 1;                          (* Undefinierter Fall bereits erledigt *)
  b y kill_neq_tac 1 THEN kill_neq_tac 2; (* Vernichten störender Prämissen *)
 1. ⋀xa. ⟦ tree_1_rep‘x = sinl‘xa; xa ≠ ⊥; True ⟧ ⟹ P
 2. ⋀y.  ⟦ tree_1_rep‘x = sinr‘y; y ≠ ⊥; True ⟧ ⟹ P
```

Erster Summand ist ein einelementiger Bereich

```
> b y rtac oneE 1 THEN contr_tac 1;
 1. ⋀xa. ⟦ tree_1_rep‘x = sinl‘xa; xa ≠ ⊥; True; xa = one ⟧ ⟹ P
 2. ⋀y.  ⟦ tree_1_rep‘x = sinr‘y; y ≠ ⊥; True ⟧ ⟹ P
```

Rest des ersten Summanden

```
> b y rtac (hd(tl prems)) 1; (* Verwenden der Prämisse "x = leaf ⟹ P" *)
 1. ⋀xa. ⟦ tree_1_rep‘x = sinl‘xa; xa ≠ ⊥; True; xa = one ⟧ ⟹ x = leaf
 2. ⋀y.  ⟦ tree_1_rep‘x = sinr‘y; y ≠ ⊥; True ⟧ ⟹ P

> b y rewrite_goals_tac axs_con_def; (* Auffalten der Konstruktor-Definition *)
 1. ⋀xa. ⟦ tree_1_rep‘x = sinl‘xa; xa ≠ ⊥; True; xa = one ⟧ ⟹
        x = tree_1_abs‘(sinl‘one)
 2. ⋀y.  ⟦ tree_1_rep‘x = sinr‘y; y ≠ ⊥; True ⟧ ⟹ P

> b y dtac iso_swap 1; (* Lemma "tree_1_rep‘?x = ?y ⟹ ?x = tree_1_abs‘?y" *)
 1. ⋀xa. ⟦ xa ≠ ⊥; True; xa = one; x = tree_1_abs‘(sinl‘xa) ⟧ ⟹
        x = tree_1_abs‘(sinl‘one)
 2. ⋀y.  ⟦ tree_1_rep‘x = sinr‘y; y ≠ ⊥; True ⟧ ⟹ P
```

```
> b y simp_tac HOLCF_ss 1; (* Beta-Reduktion, hier unnötig *)
  b y fast_tac HOL_cs 1;    (* Gleichungslogik, unter Verwendung der Annahmen *)
 1. ⋀y. ⟦ tree_1_rep‘x = sinr‘y; y ≠ ⊥; True ⟧ ⟹ P
```

Zweiter Summand ist wieder eine Summe

```
> b y rtac ssumE 1 THEN contr_tac 1; b y kill_neq_tac 1 THEN kill_neq_tac 2;
 1. ⋀y xa. ⟦ tree_1_rep‘x = sinr‘y; True; y = sinl‘xa; xa ≠ ⊥; True ⟧ ⟹ P
 2. ⋀y ya. ⟦ tree_1_rep‘x = sinr‘y; True; y = sinr‘ya; ya ≠ ⊥; True ⟧ ⟹ P
```

Rest des zweiten Summanden, analog

```
> b y rtac (hd(tl(tl prems))) 1; (* Prämisse "⋀b. ⟦ x = node‘b; b ≠ ⊥ ⟧ ⟹ P" *)
  b y rewrite_goals_tac axs_con_def; b y dtac iso_swap 1;
  b y simp_tac HOLCF_ss 1;
  b y fast_tac HOL_cs   1 THEN fast_tac HOL_cs 1;
 1. ⋀y ya. ⟦ tree_1_rep‘x = sinr‘y; True; y = sinr‘ya; ya ≠ ⊥; True ⟧ ⟹ P
```

Dritter Summand ist ein Produkt

```
> b y rtac sprodE 1 THEN contr_tac 1;
 1. ⋀y ya xa yb. ⟦ tree_1_rep‘x = sinr‘y; True; y = sinr‘ya; ya ≠ ⊥; True;
                  ya = (|xa, yb|); xa ≠ ⊥; yb ≠ ⊥ ⟧ ⟹ P
```

Rest des dritten Summanden, analog

```
> b y rtac (hd(tl(tl(tl prems)))) 1; (* "⋀b t. ⟦ x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P" *)
  b y rewrite_goals_tac axs_con_def; b y dtac iso_swap 1;
  b y simp_tac HOLCF_ss 1;
  b y fast_tac HOL_cs   1 THEN fast_tac HOL_cs 1 THEN fast_tac HOL_cs 1;
No subgoals!
```

```
> goalw thy [] "x = ⊥ ∨ x = leaf ∨ (∃b. x = node‘b ∧ b ≠ ⊥) ∨          (* exhaust *)
                (∃b t. x = (b in_f t) ∧ b ≠ ⊥ ∧ t ≠ ⊥)";
 1. x = ⊥ ∨ x = leaf ∨ (∃b. x = node‘b ∧ b ≠ ⊥) ∨ (∃b t. x = (b in_f t) ∧ b ≠ ⊥ ∧ t ≠ ⊥)

> b y rtac cases 1; (* Rückführung auf die Fallunterscheidung *)
 1. ?x1 = ⊥ ⟹ x = ⊥ ∨ x = leaf ∨ (∃b. x = node‘b ∧ b ≠ ⊥) ∨
               (∃b t. x = (b in_f t) ∧ b ≠ ⊥ ∧ t ≠ ⊥)
 2. ?x1 = leaf ⟹ x = ⊥ ∨ x = leaf ∨ (∃b. x = node‘b ∧ b ≠ ⊥) ∨
                  (∃b t. x = (b in_f t) ∧ b ≠ ⊥ ∧ t ≠ ⊥)
 3. ⋀b. ⟦ ?x1 = node‘b; b ≠ ⊥ ⟧ ⟹ x = ⊥ ∨ x = leaf ∨ (∃b. x = node‘b ∧ b ≠ ⊥) ∨
                                   (∃b t. x = (b in_f t) ∧ b ≠ ⊥ ∧ t ≠ ⊥)
 4. ⋀b t. ⟦ ?x1 = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ x = ⊥ ∨ x = leaf ∨
          (∃b. x = node‘b ∧ b ≠ ⊥) ∨ (∃b t. x = (b in_f t) ∧ b ≠ ⊥ ∧ t ≠ ⊥)

> b y REPEAT(fast_tac HOL_cs 1); (* Reine Prädikatenlogik *)
No subgoals!
```

6.3.3 `when`-Funktional

Triviales Lemma zur Applikation des `when`-Funktionals aus Effizienzgründen, weil es öfters gebraucht wird und die impliziten Stetigkeitsbeweise für die Beta-Reduktion sehr aufwendig sind

```
> val when_appl = pg thy [ax_when_def]
  "tree_1_when‘f1‘f2‘f3‘x = sswhen‘(Λdummy. f1)‘(sswhen‘f2‘(ssplit‘f3))‘(tree_1_rep‘x)"
  [simp_tac HOLCF_ss 1];
```

```
> pg thy [] "tree_1_when‘f1‘f2‘f3‘⊥ = ⊥"                                  (* when_strict *)
  [simp_tac(HOLCF_ss addsimps [when_appl,rep_strict]) 1];
```

```
> pg thy axs_con_def "b ≠ ⊥ ⟹ tree_1_when‘f1‘f2‘f3‘(node‘b) = f2‘b"(** when_apps **)
  [asm_simp_tac (HOLCF_ss addsimps [when_appl,ax_abs_iso]) 1];
```

6.3.4 Diskriminatoren, Konstruktoren und Selektoren

```
> pg thy axs_dis_def "is_node‘⊥ = ⊥"                                    (** dis_stricts **)
  [simp_tac (HOLCF_ss addsimps when_rews) 1];
```

```
> pg thy axs_dis_def "⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ is_node‘(b in_f t) = FF"   (** dis_apps **)
  [asm_simp_tac (HOLCF_ss addsimps when_rews) 1];
```

```
> goalprems thy "x ≠ ⊥ ⟹ is_node‘x ≠ ⊥";                               (** dis_defins **)
 1. x ≠ ⊥ ⟹ is_node‘x ≠ ⊥

> b y rtac cases 1; (* Fallunterscheidung über den Datentyp *)
 1. ⟦ x ≠ ⊥; ?x1 = ⊥ ⟧ ⟹ is_node‘x ≠ ⊥
 2. ⟦ x ≠ ⊥; ?x1 = leaf ⟧ ⟹ is_node‘x ≠ ⊥
 3. ⋀b. ⟦ x ≠ ⊥; ?x1 = node‘b; b ≠ ⊥ ⟧ ⟹ is_node‘x ≠ ⊥
 4. ⋀b t. ⟦ x ≠ ⊥; ?x1 = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ is_node‘x ≠ ⊥

> b y contr_tac 1; b y REPEAT(CHANGED(asm_simp_tac (HOLCF_ss addsimps dis_apps) 1));
No subgoals!
```

```
> pg thy axs_con_def "node‘⊥ = ⊥"                                       (** con_stricts **)
[asm_simp_tac (HOLCF_ss addsimps [abs_strict]) 1];
```

```
> goalprems thy "b ≠ ⊥ ⟹ node‘b ≠ ⊥";                                   (** con_defins **)
 1. b ≠ ⊥ ⟹ node‘b ≠ ⊥
```

```
> b y rtac swap3 1; (* Widerspruchsbeweis durch Annahme des Gegenteils *)
 1. ⟦ b ≠ ⊥; node`b = ⊥ ⟧ ⟹ ?P1
 2. b ≠ ⊥ ⟹ ¬ ?P1

> b y eres_inst_tac [("f","is_node")] cfun_arg_cong 1;
                    (* Leibniz-Ersetzbarkeit "?x = ?y ⟹ ?f`?x = ?f`?y" *)
 1. b ≠ ⊥ ⟹ is_node`(node`b) ≠ is_node`⊥

> b y asm_simp_tac (HOLCF_ss addsimps dis_rews) 1;
No subgoals!


> pg thy axs_sel_def "branch`⊥ = ⊥"                          (** sel_stricts **)
  [simp_tac (HOLCF_ss addsimps when_rews) 1];


> goalw thy axs_sel_def "tree_1_sel_3_1`(node`b) = ⊥";          (** sel_apps **)
 1. tree_1_when`⊥`⊥`(Λb t. b)`(node`b) = ⊥

> b y case_UU_tac (when_rews @ con_stricts) 1 "b"; (* Undefinierter Fall vorweg *)
 1. b ≠ ⊥ ⟹ tree_1_when`⊥`⊥`(Λb t. b)`(node`b) = ⊥

> b y asm_simp_tac (HOLCF_ss addsimps when_rews) 1;
No subgoals!


> goalprems thy "x ≠ ⊥ ⟹ empty_sel_1_1`x ≠ ⊥";                (** sel_defins **)
 1. x ≠ ⊥ ⟹ empty_sel_1_1`x ≠ ⊥

> b y rtac cases 1; (* Fallunterscheidung über den Datentyp *)
 1. ⟦ x ≠ ⊥; ?x1 = ⊥ ⟧ ⟹ empty_sel_1_1`x ≠ ⊥
 2. ⋀e. ⟦ x ≠ ⊥; ?x1 = silly`e; e ≠ ⊥ ⟧ ⟹ empty_sel_1_1`x ≠ ⊥

> b y contr_tac 1; (* Widerspruch *)
 1. ⋀e. ⟦ x ≠ ⊥; x = silly`e; e ≠ ⊥ ⟧ ⟹ empty_sel_1_1`x ≠ ⊥

> b y asm_simp_tac (HOLCF_ss addsimps sel_apps) 1; (* Gleichungslogik *)
No subgoals!
```

6.3.5 Nicht-Abschwächung und Ungleichheit

```
> goalprems thy "b ≠ ⊥ ⟹ ¬ node‘b ⊑ (ba in_f t)";                  (** dists_le **)
 1. b ≠ ⊥ ⟹ ¬ node‘b ⊑ (ba in_f t)

> b y rtac swap3 1; (* Widerspruchsbeweis *)
 1. ⟦ b ≠ ⊥; node‘b ⊑ (ba in_f t) ⟧ ⟹ ?P1
 2. b ≠ ⊥ ⟹ ¬ ?P1

> b y eres_inst_tac [("fo5","is_node")] monofun_cfun_arg 1; (* Monotonie von is_node *)
 1. b ≠ ⊥ ⟹ ¬ is_node‘(node‘b) ⊑ is_node‘(ba in_f t)

> b y case_UU_tac (con_stricts @ dis_rews) 1 "ba"; (* Fallunterscheidung bzgl. ba *)
  b y case_UU_tac (con_stricts @ dis_rews) 1 "t" ; (* Fallunterscheidung bzgl. t *)
 1. ⟦ b ≠ ⊥; ba ≠ ⊥; t ≠ ⊥ ⟧ ⟹ ¬ is_node‘(node‘b) ⊑ is_node‘(ba in_f t)

> b y asm_simp_tac (HOLCF_ss addsimps dis_rews) 1;
No subgoals!


> dist_le RS dist_eqI;                                                 (** dists_eq **)
val it = "?b1 ≠ ⊥ ⟹ node‘?b1 ≠ (?ba1 in_f ?t1)" : thm
```

6.3.6 Invers-Monotonie und Injektivität

```
> goalprems thy "⟦ node‘b ⊑ node‘b’; b ≠ ⊥; b’ ≠ ⊥ ⟧ ⟹ b ⊑ b’";     (** inverts **)
 1. ⟦ node‘b ⊑ node‘b’; b ≠ ⊥; b’ ≠ ⊥ ⟧ ⟹ b ⊑ b’

> b y dres_inst_tac [("fo5","branch")] monofun_cfun_arg 1;
 1. ⟦ b ≠ ⊥; b’ ≠ ⊥; branch‘(node‘b) ⊑ branch‘(node‘b’) ⟧ ⟹ b ⊑ b’

> b y asm_full_simp_tac (HOLCF_ss addsimps sel_apps) 1;
No subgoals!

> goalprems thy "⟦ node‘b = node‘b’; b ≠ ⊥; b’ ≠ ⊥ ⟧ ⟹ b = b’";     (** injects **)
 1. ⟦ node‘b = node‘b’; b ≠ ⊥; b’ ≠ ⊥ ⟧ ⟹ b = b’
```

Zurückführung auf Invers-Monotonie

```
> b y etac (antisym_less_inverse RS conjE) 1; (* "?x = ?y ⟹ ?x ⊑ ?y ∧ ?y ⊑ ?x" *)
 1. ⟦ b ≠ ⊥; b’ ≠ ⊥; node‘b ⊑ node‘b’; node‘b’ ⊑ node‘b ⟧ ⟹ b = b’

> b y dtac invert 1 THEN REPEAT(atac 1); b y dtac invert 1 THEN REPEAT(atac 1);
 1. ⟦ b ≠ ⊥; b’ ≠ ⊥; b ⊑ b’; b’ ⊑ b ⟧ ⟹ b = b’

> b y REPEAT(rtac antisym_less 1 ORELSE atac 1); (* "⟦ ?x ⊑ ?y; ?y ⊑ ?x ⟧ ⟹ ?x = ?y" *)
No subgoals!
```

6.3.7 copy-Funktional

```
> pg thy [ax_copy_def] "tree_1_copy‘f‘⊥ = ⊥"                      (* copy_strict *)
  [asm_simp_tac(HOLCF_ss addsimps [abs_strict,rep_strict,cfst_strict,csnd_strict]) 1];


> goalprems thy "b ≠ ⊥ ⟹                                          (** copy_apps **)
                 tree_1_copy‘f‘(node‘b) = node‘(csnd‘(csnd‘f)‘b)";
  b y rewrite_goals_tac (ax_copy_def::axs_con_def);
 1. b ≠ ⊥ ⟹
    (Λf. tree_1_abs oo sswhen‘sinl‘(sinr oo sswhen‘(sinl oo csnd‘(csnd‘f))‘(sinr oo
     ssplit‘(Λx y. (|x, y|)))) oo tree_1_rep)‘f‘((Λb. tree_1_abs‘(sinr‘(sinl‘b)))‘b) =
    (Λb. tree_1_abs‘(sinr‘(sinl‘b)))‘(csnd‘(csnd‘f)‘b)

> b y simp_tac (HOLCF_ss addsimps [ax_abs_iso]) 1;
No subgoals!
```

Nur für Konstruktoren mit strikten, rekursiven Argumenten

```
> goal thy "tree_1_copy‘⊥‘(node‘b) = ⊥";                         (** copy_stricts **)
 1. tree_1_copy‘⊥‘(node‘b) = ⊥

> b y case_UU_tac (cfst_strict::csnd_strict::copy_strict::copy_apps@con_rews) 1 "b";
  (* Undefinierter Fall vorweg für alle strikten Argumente, hier nur eines *)
 1. b ≠ ⊥ ⟹ tree_1_copy‘⊥‘(node‘b) = ⊥

> b y asm_simp_tac (HOLCF_ss addsimps
                  (cfst_strict::csnd_strict::copy_strict::copy_apps@con_rews)) 1;
No subgoals!
```

6.4 Fixpunktanteil

Fixpunkt-Axiom und Definitionen, hier für dem Gesamt-Typ `tree_1_tree_2_branch`

```
val axs_reach = ["cfst`(fix`tree_1_tree_2_branch_copy) `?x1.0 = ?x1.0",
           "cfst`(csnd`(fix`tree_1_tree_2_branch_copy))`?x2.0 = ?x2.0"
           "csnd`(csnd`(fix`tree_1_tree_2_branch_copy))`?x3.0 = ?x3.0"] : thm list
val axs_take_def = ["tree_1_take ≡ λn. cfst`(iterate n tree_1_tree_2_branch_copy ⊥) ",
              "tree_2_take ≡ λn. cfst`(csnd`(iterate n tree_1_tree_2_branch_copy ⊥))",
              "branch_take ≡ λn. csnd`(csnd`(iterate n tree_1_tree_2_branch_copy ⊥))"]
val axs_finite_def = ["tree_1_finite ≡ λx1. ∃n. tree_1_take n`x1 = x1",
                      "tree_2_finite ≡ λx2. ∃n. tree_2_take n`x2 = x2",
                      "branch_finite ≡ λx3. ∃n. branch_take n`x3 = x3"] : thm list
val ax_copy2_def = "tree_1_tree_2_branch_copy ≡
                   Λf. <tree_1_copy`f, tree_2_copy`f, branch_copy`f>" : thm
val ax_bisim_def = "tree_1_tree_2_branch_bisim ≡  λR. (* ... *)"

val casess    = [tree_1.cases     , tree_2.cases     , branch.cases]   ;
val con_rews  =  tree_1.con_rews  @ tree_2.con_rews  @ branch.con_rews ;
val copy_rews =  tree_1.copy_rews @ tree_2.copy_rews @ branch.copy_rews;
val iterate_Cprod_ss = simpset_of "Fix"
                       addsimps [cfst_strict, csnd_strict] addsimps Cprod_rews;
```

6.4.1 take-Funktionale

Der Term `(Λf.<tree_1_copy`f, tree_2_copy`f, branch_copy`f>)` wird in diesem Unterabschnitt aus Gründen der Übersichtlichkeit durch `fun` abgekürzt.

```
> goalw thy (ax_copy2_def::axs_take_def)                          (* take_stricts *)
  "tree_1_take n`⊥ = (⊥::('a,'b)tree_1) ∧ tree_2_take n`⊥ = (⊥::('b,'a)tree_2) ∧
   branch_take n`⊥ = (⊥::('a,'b)branch)";
 1. cfst`(iterate n fun ⊥) `⊥ = ⊥ ∧ cfst`(csnd`(iterate n fun ⊥))`⊥ = ⊥ ∧
                                    csnd`(csnd`(iterate n fun ⊥))`⊥ = ⊥
```

Simultane Induktion für alle drei Teiltypen

```
> b y nat_ind_tac "n" 1;
 1. cfst`(iterate 0 fun ⊥) `⊥ = ⊥ ∧ cfst`(csnd`(iterate 0 fun ⊥))`⊥ = ⊥ ∧
                                    csnd`(csnd`(iterate 0 fun ⊥))`⊥ = ⊥
 2. ⋀n1. cfst`(iterate n1 fun ⊥) `⊥ = ⊥ ∧ cfst`(csnd`(iterate n1 fun ⊥))`⊥ = ⊥ ∧
                                          csnd`(csnd`(iterate n1 fun ⊥))`⊥ = ⊥
 ⟹ cfst`(iterate (Suc n1) fun ⊥) `⊥ = ⊥ ∧ cfst`(csnd`(iterate (Suc n1) fun ⊥))`⊥ = ⊥ ∧
                                            csnd`(csnd`(iterate (Suc n1) fun ⊥))`⊥ = ⊥

> b y     simp_tac  iterate_Cprod_ss                       1; (* Lösen des Basisfalls *)
  b y asm_simp_tac (iterate_Cprod_ss addsimps copy_rews) 1; (* Lösen des Induktions-
                                    schritts unter Zuhilfenahme der Induktionsannahmen *)
No subgoals!

val take_stricts'= rewrite_rule copy_take_defs take_stricts;
```

```
> pg thy axs_take_def "tree_1_take 0‘x1= ⊥"                          (** take_0s **)
  [simp_tac iterate_Cprod_ss 1];


> goalw thy (ax_copy2_def::axs_take_def)                             (* take_apps *)
        "tree_1_take (Suc n)‘leaf       = leaf ∧
(∀b    . tree_1_take (Suc n)‘(node‘b)   = node‘(branch_take n‘b)) ∧
(∀b t  . tree_1_take (Suc n)‘(b in_f t) = (b in_f t)) ∧
(∀t    . tree_2_take (Suc n)‘(#‘t)      = #‘(tree_1_take n‘t)) ∧
(∀a t b. branch_take (Suc n)‘(<a:t|>>b) = <a:tree_2_take n‘t|>>(branch_take n‘b))";
 1.                cfst‘(iterate (Suc n) fun ⊥)‘leaf = leaf ∧
    (∀b.           cfst‘(iterate (Suc n) fun ⊥)‘(node‘b) =
                                 node‘(csnd‘(csnd‘(iterate n fun ⊥))‘b)) ∧
    (∀b t.         cfst‘(iterate (Suc n) fun ⊥)‘(b in_f t) = (b in_f t)) ∧
    (∀t.     cfst‘(csnd‘(iterate (Suc n) fun ⊥))‘(#‘t) = #‘(cfst‘(iterate n fun ⊥)‘t)) ∧
    (∀a t b. csnd‘(csnd‘(iterate (Suc n) fun ⊥))‘<a:t|>>b =
             <a:cfst‘(csnd‘(iterate n fun ⊥))‘t|>>csnd‘(csnd‘(iterate n fun ⊥))‘b)
```

Simultane Induktion für alle drei Teiltypen

```
> b y simp_tac iterate_Cprod_ss 1; (* Vereinfachung *)
  b y nat_ind_tac "n" 1;
 1.          tree_1_copy‘(iterate 0 fun ⊥)‘leaf = leaf ∧
    (∀b    . tree_1_copy‘(iterate 0 fun ⊥)‘(node‘b) =
                                 node‘(csnd‘(csnd‘(iterate 0 fun ⊥))‘b)) ∧
    (∀b t  . tree_1_copy‘(iterate 0 fun ⊥)‘(b in_f t) = (b in_f t)) ∧
    (∀t    . tree_2_copy‘(iterate 0 fun ⊥)‘(#‘t) =
                                 #‘(cfst‘(iterate 0 fun ⊥)‘t)) ∧
    (∀a t b. branch_copy‘(iterate 0 fun ⊥)‘<a:t|>>b =
                <a:cfst‘(csnd‘(iterate 0 fun ⊥))‘t|>>csnd‘(csnd‘(iterate 0 fun ⊥))‘b)
 2. ⋀n1.     tree_1_copy‘(iterate n1 fun ⊥)‘leaf = leaf ∧
    (∀b    . tree_1_copy‘(iterate n1 fun ⊥)‘(node‘b) =
                                 node‘(csnd‘(csnd‘(iterate n1 fun ⊥))‘b)) ∧
    (∀b t  . tree_1_copy‘(iterate n1 fun ⊥)‘(b in_f t) = (b in_f t)) ∧
    (∀t    . tree_2_copy‘(iterate n1 fun ⊥)‘(#‘t) =
                                 #‘(cfst‘(iterate n1 fun ⊥)‘t)) ∧
    (∀a t b. branch_copy‘(iterate n1 fun ⊥)‘<a:t|>>b =
            <a:cfst‘(csnd‘(iterate n1 fun ⊥))‘t|>>csnd‘(csnd‘(iterate n1 fun ⊥))‘b)
 ⟹           tree_1_copy‘(iterate (Suc n1) fun ⊥)‘leaf = leaf ∧
    (∀b    . tree_1_copy‘(iterate (Suc n1) fun ⊥)‘(node‘b) =
                                 node‘(csnd‘(csnd‘(iterate (Suc n1) fun ⊥))‘b)) ∧
    (∀b t  . tree_1_copy‘(iterate (Suc n1) fun ⊥)‘(b in_f t) = (b in_f t)) ∧
    (∀t    . tree_2_copy‘(iterate (Suc n1) fun ⊥)‘(#‘t) =
                                 #‘(cfst‘(iterate (Suc n1) fun ⊥)‘t)) ∧
    (∀a t b. branch_copy‘(iterate (Suc n1) fun ⊥)‘<a:t|>>b =
    <a:cfst‘(csnd‘(iterate (Suc n1) fun ⊥))‘t|>>csnd‘(csnd‘(iterate (Suc n1) fun ⊥))‘b)
```

Lösen des Basisfalls mit dem Simplifier

```
> b y simp_tac (iterate_Cprod_ss addsimps copy_con_rews) 1;
```

Induktionsschritt für die einfacheren Fälle

```
  b y asm_full_simp_tac (HOLCF_ss addsimps (filter (has_fewer_prems 1) copy_rews)) 1;
```

Aufspalten der Annahmen und der Konklusion

```
  b y TRY(safe_tac HOL_cs);
 1. ⋀n1 b. ⟦ ∀b. tree_1_copy‘(iterate n1 fun ⊥)‘(node‘b) =
            node‘(csnd‘(csnd‘(iterate n1 fun ⊥))‘b);
             ∀t. tree_2_copy‘(iterate n1 fun ⊥)‘(#‘t) = #‘(cfst‘(iterate n1 fun ⊥)‘t);
             ∀a t b. branch_copy‘(iterate n1 fun ⊥)‘<a:t|>>b =
            <a:cfst‘(csnd‘(iterate n1 fun ⊥))‘t|>>csnd‘(csnd‘(iterate n1 fun ⊥))‘b ⟧ ⟹
    tree_1_copy‘<tree_1_copy‘(iterate n1 fun ⊥), tree_2_copy‘(iterate n1 fun ⊥),
                 branch_copy‘(iterate n1 fun ⊥)>‘(node‘b) =
    node‘(branch_copy‘(iterate n1 fun ⊥)‘b)
 2. (* ... *)
 3. (* ... *)
```

Induktionsschritt für die Konstruktoren mit strikten, rekursiven Argumenten

Erster solcher Konstruktor

```
> b y case_UU_tac (take_stricts'::copy_con_rews) 1 "b"; (* Undefinierter Fall *)
 1. ⋀n1 b. ⟦ (* ... *); b ≠ ⊥ ⟧ ⟹
    tree_1_copy‘<tree_1_copy‘(iterate n1 fun ⊥), tree_2_copy‘(iterate n1 fun ⊥),
                 branch_copy‘(iterate n1 fun ⊥)>‘(node‘b) =
    node‘(branch_copy‘(iterate n1 fun ⊥)‘b)
 2. (* ... *)
 3. (* ... *)

> b y asm_full_simp_tac (HOLCF_ss addsimps copy_rews) 1; (* Definierter Fall *)
 1. ⋀n1 t. ⟦ (* ... *) ⟧ ⟹
    tree_2_copy‘<tree_1_copy‘(iterate n1 fun ⊥), tree_2_copy‘(iterate n1 fun ⊥),
                 branch_copy‘(iterate n1 fun ⊥)>‘(#‘t) =
    #‘(tree_1_copy‘(iterate n1 fun ⊥)‘t)
 2. (* ... *)
```

Zweiter solcher Konstruktor, analog

```
> b y case_UU_tac (take_stricts'::copy_con_rews) 1 "t";
  b y asm_full_simp_tac (HOLCF_ss addsimps copy_rews) 1;
 1. ⋀n1 a t b. ⟦ (* ... *) ⟧ ⟹
    branch_copy‘<tree_1_copy‘(iterate n1 fun ⊥), tree_2_copy‘(iterate n1 fun ⊥),
                 branch_copy‘(iterate n1 fun ⊥)>‘<a:t|>>b =
       <a:tree_2_copy‘(iterate n1 fun ⊥)‘t|>>branch_copy‘(iterate n1 fun ⊥)‘b
```

Dritterr solcher Konstruktor, analog

```
> b y case_UU_tac(take_stricts'::copy_con_rews)1 "t";
  b y asm_full_simp_tac (HOLCF_ss addsimps copy_rews) 1;
No subgoals!
```

6.4.2 Endliche Induktion

```
> val prems = goal thy "⟦ P1 ⊥; P1 leaf;                                    (* finite_ind *)
              ⋀b     . ⟦ b ≠ ⊥; P3 b                  ⟧ ⟹ P1 (node‘b);
              ⋀b t   . ⟦ b ≠ ⊥; t ≠ ⊥                 ⟧ ⟹ P1 (b in_f t);
        P2 ⊥; ⋀t     . ⟦ t ≠ ⊥; P1 t                  ⟧ ⟹ P2 (#‘t);
        P3 ⊥; ⋀a t b. ⟦ a ≠ ⊥; t ≠ ⊥; P2 t; P3 b ⟧ ⟹ P3 <a:t|>>b ⟧ ⟹
   P1 (tree_1_take n‘x1) ∧ P2 (tree_2_take n‘x2) ∧ P3 (branch_take n‘x3)";
  b y quant_tac 1 THEN simp_tac quant_ss 1;
 1. (∀x. P1 (tree_1_take n‘x)) ∧ (∀x. P2 (tree_2_take n‘x)) ∧ (∀x. P3 (branch_take n‘x))
```

Simultane Induktion für alle drei Teiltypen, Lösen des Basisfalls mit dem Simplifier

```
> b y nat_ind_tac "n" 1;
  b y simp_tac (HOL_ss addsimps take_rews addsimps prems) 1;
 1. ⋀n1. (∀x. P1 (tree_1_take n1‘x)) ∧
        (∀x. P2 (tree_2_take n1‘x)) ∧ (∀x. P3 (branch_take n1‘x)) ⟹
        (∀x. P1 (tree_1_take (Suc n1)‘x)) ∧
        (∀x. P2 (tree_2_take (Suc n1)‘x)) ∧ (∀x. P3 (branch_take (Suc n1)‘x))
```

Aufspalten der Annahmen und der Konklusion

```
> b y TRY(safe_tac HOL_cs);
 1. ⋀n1 x. ⟦ ∀x. P1 (tree_1_take n1‘x);   ∀x. P2 (tree_2_take n1‘x);
            ∀x. P3 (branch_take n1‘x) ⟧ ⟹   P1 (tree_1_take (Suc n1)‘x)
 2. ⋀n1 x. ⟦ ∀x. P1 (tree_1_take n1‘x);   ∀x. P2 (tree_2_take n1‘x);
            ∀x. P3 (branch_take n1‘x) ⟧ ⟹   P2 (tree_2_take (Suc n1)‘x)
 3. ⋀n1 x. ⟦ ∀x. P1 (tree_1_take n1‘x);   ∀x. P2 (tree_2_take n1‘x);
            ∀x. P3 (branch_take n1‘x) ⟧ ⟹   P3 (branch_take (Suc n1)‘x)
```

Fallunterscheidung für erstes Teilziel (erster Teiltyp)

```
> b y res_inst_tac [("x","x")] (hd casess) 1;
 1. ⋀n1 x. ⟦ ∀x. P1 (tree_1_take n1‘x); ∀x. P2 (tree_2_take n1‘x);
            ∀x. P3 (branch_take n1‘x); x = ⊥  ⟧ ⟹ P1 (tree_1_take (Suc n1)‘x)
 2. ⋀n1 x.     ⟦ (* ... *); x = leaf            ⟧ ⟹ P1 (tree_1_take (Suc n1)‘x)
 3. ⋀n1 x b.   ⟦ (* ... *); x = node‘b; b ≠ ⊥ ⟧ ⟹ P1 (tree_1_take (Suc n1)‘x)
 4. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹
                                                  P1 (tree_1_take (Suc n1)‘x)
 5. ⋀n1 x. ⟦ (* ... *) ⟧ ⟹ P2 (tree_2_take (Suc n1)‘x)
 6. ⋀n1 x. ⟦ (* ... *) ⟧ ⟹ P3 (branch_take (Suc n1)‘x)
```

Lösen des undefinierten Falls

```
> b y asm_simp_tac (HOL_ss addsimps take_rews addsimps prems) 1; (* mit P1 ⊥ *)
 1. ⋀n1 x.     ⟦ (* ... *); x = leaf            ⟧ ⟹ P1 (tree_1_take (Suc n1)‘x)
 2. ⋀n1 x b.   ⟦ (* ... *); x = node‘b; b ≠ ⊥ ⟧ ⟹ P1 (tree_1_take (Suc n1)‘x)
 3. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹
                                                  P1 (tree_1_take (Suc n1)‘x)
 4. (* ... *)
 5. (* ... *)
```

Erster Konstruktor

```
>   b y asm_simp_tac (HOL_ss addsimps take_rews) 1; (* Einsetzen von x = leaf,
                                                       Vereinfachung damit *)
 1. ⋀n1 x    . ⟦ (* ... *); x = leaf            ⟧ ⟹ P1 leaf
 2. ⋀n1 x b  . ⟦ (* ... *); x = node‘b; b ≠ ⊥ ⟧ ⟹ P1 (tree_1_take (Suc n1)‘x)
 3. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹
                                                     P1 (tree_1_take (Suc n1)‘x)
 4. (* ... *)
 5. (* ... *)

>   b y resolve_tac prems 1; (* Fertig nach Voraussetzung P1 leaf *)
 1. ⋀n1 x b.   ⟦ (* ... *); x = node‘b; b ≠ ⊥ ⟧ ⟹ P1 (tree_1_take (Suc n1)‘x)
 2. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹
                                                     P1 (tree_1_take (Suc n1)‘x)
 3. (* ... *)
 4. (* ... *)
```

Zweiter Konstruktor

```
>   b y asm_simp_tac (HOL_ss addsimps take_rews) 1; (* Einsetzen von x = node‘b, analog *)
 1. ⋀n1 x b. ⟦ (* ... *); x = node‘b; b ≠ ⊥ ⟧ ⟹ P1 (node‘(branch_take n1‘b))
⋮
 4. (* ... *)
```

Fallunterscheidung für jedes strikte und rekursive Argument (hier eines)

```
>     b y case_UU_tac (prems@con_rews) 1 "branch_take n1‘b";
 1. ⋀n1 x b. ⟦ (* ... *); x = node‘b; b ≠ ⊥; branch_take n1‘b ≠ ⊥ ⟧ ⟹
                                                     P1 (node‘(branch_take n1‘b))
 2. (* ... *)
 3. (* ... *)
 4. (* ... *)

>     b y resolve_tac prems 1; (* Voraussetzung ⋀b. ⟦ b ≠ ⊥; P3 b ⟧ ⟹ P1 (node‘b) *)
 1. ⋀n1 x b. ⟦ (* ... *); (* ... *); branch_take n1‘b ≠ ⊥ ⟧ ⟹ branch_take n1‘b ≠ ⊥
 2. ⋀n1 x b. ⟦ ∀x. P1 (tree_1_take n1‘x); ∀x. P2 (tree_2_take n1‘x);
              ∀x. P3 (branch_take n1‘x); (* ... *)     ⟧ ⟹ P3 (branch_take n1‘b)
 3. (* ... *)
 4. (* ... *)
 5. (* ... *)

>     b y atac      1; (* Lösen nach Annahme, für jedes strikte Argument (hier eines) *)
      b y etac spec 1; (* Lösen nach Ind.vor. für jedes rekursive Argument (hier eines) *)
 1. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹
                                                     P1 (tree_1_take (Suc n1)‘x)
 2. (* ... *)
 3. (* ... *)
```

Dritter Konstruktor

```
>   b y asm_simp_tac (HOL_ss addsimps take_rews) 1;
 1. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P1 (b in_f t)
 2. (* ... *)
 3. (* ... *)
```

Hier keine strikten, rekursiven Argumente

```
>   b y resolve_tac prems 1; (* Voraussetzung ⋀b t. ⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P1 (b in_f t) *)
 1. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ b ≠ ⊥
 2. ⋀n1 x b t. ⟦ (* ... *); x = (b in_f t); b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ t ≠ ⊥
 3. (* ... *)
 4. (* ... *)
```

Hier zwei strikte Argumente, und kein rekursives Argument

```
>     b y atac 1 THEN atac 1;
 1. ⋀n1 x. ⟦ ∀x. P1 (tree_1_take n1‘x); ∀x. P2 (tree_2_take n1‘x);
            ∀x. P3 (branch_take n1‘x) ⟧ ⟹  P2 (tree_2_take (Suc n1)‘x)
 2. ⋀n1 x. ⟦ ∀x. P1 (tree_1_take n1‘x); ∀x. P2 (tree_2_take n1‘x);
            ∀x. P3 (branch_take n1‘x) ⟧ ⟹  P3 (branch_take (Suc n1)‘x)
```

Zweiter Teiltyp, analog

```
> b y res_inst_tac [("x","x")] (hd (tl casess)) 1;
  b y asm_simp_tac (HOL_ss addsimps take_rews addsimps prems) 1;
    b y asm_simp_tac (HOL_ss addsimps take_rews) 1;
      b y case_UU_tac (prems@con_rews) 1 "tree_1_take n1‘t";
      b y resolve_tac prems 1;
      b y atac      1;
      b y etac spec 1;
 1. ⋀n1 x. ⟦ ∀x. P1 (tree_1_take n1‘x); ∀x. P2 (tree_2_take n1‘x);
            ∀x. P3 (branch_take n1‘x) ⟧ ⟹  P3 (branch_take (Suc n1)‘x)
```

Dritter Teiltyp, analog

```
> b y res_inst_tac [("x","x")] (hd (tl (tl casess))) 1;
  b y asm_simp_tac (HOL_ss addsimps take_rews addsimps prems) 1;
    b y asm_simp_tac (HOL_ss addsimps take_rews) 1;
      b y case_UU_tac (prems@con_rews) 1 "tree_2_take n1‘t";
      b y resolve_tac prems 1;
      b y atac      1 THEN atac      1;
      b y etac spec 1 THEN etac spec 1;
No subgoals!
```

6.4.3 Induktionslemmata

```
> val prems = goalw thy axs_take_def                        (** take_lemmas **)
        "(⋀n. tree_1_take n`x1= tree_1_take n`x1') ⟹ x1 = x1'";
 1. x1 = x1'
```

Zweimaliges Auffalten mit dem `reach`-Axiom

```
> b y res_inst_tac [("t","x1" )] (hd axs_reach RS subst) 1 THEN
      res_inst_tac [("t","x1'")] (hd axs_reach RS subst) 1;
 1. cfst`(fix`tree_1_tree_2_branch_copy)`x1 = cfst`(fix`tree_1_tree_2_branch_copy)`x1'
```

Auffalten mit der alternativen Fixpunkt-Definition

```
> b y rtac (fix_def2 RS ssubst) 1; (*  "fix`?F = (⨆i. iterate i ?F ⊥)" *)
 1. cfst`(⨆i. iterate i tree_1_tree_2_branch_copy ⊥)`x1 =
    cfst`(⨆i. iterate i tree_1_tree_2_branch_copy ⊥)`x1'

> b y rtac (contlub_cfun_arg RS ssubst) 1; (* Supremum herausziehen *)
 1. is_chain (λi. iterate i tree_1_tree_2_branch_copy ⊥)
 2. (⨆i. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))`x1 =
    (⨆i. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))`x1'

> b y chain_tac 1; (* Nachweis der Ketteneigenschaften *)
 1. (⨆i. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))`x1 =
    (⨆i. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))`x1'
```

Zweimal Stetigkeit der Funktionsanwendung bzgl. der Funktion ausnutzen

```
> b y rtac (contlub_cfun_fun RS ssubst) 1 THEN rtac (contlub_cfun_fun RS ssubst) 2;
 1. is_chain (λi. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))
 2. is_chain (λi. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))
 3. (⨆i. cfst`(iterate i tree_1_tree_2_branch_copy ⊥)`x1) =
    (⨆i. cfst`(iterate i tree_1_tree_2_branch_copy ⊥)`x1')

> b y rtac lub_equal 3; (* Zulässigkeit der Gleichheit *)
 1. is_chain (λi. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))
 2. is_chain (λi. cfst`(iterate i tree_1_tree_2_branch_copy ⊥))
 3. is_chain (λi. cfst`(iterate i tree_1_tree_2_branch_copy ⊥)`x1)
 4. is_chain (λi. cfst`(iterate i tree_1_tree_2_branch_copy ⊥)`x1')
 5. ∀k. cfst`(iterate k tree_1_tree_2_branch_copy ⊥)`x1 =
        cfst`(iterate k tree_1_tree_2_branch_copy ⊥)`x1'
```

Nachweis der nötigen Ketteneigenschaften, Allquantor auflösen

```
> b y chain_tac 1; b y rtac allI 1;
 1. ⋀k. cfst`(iterate k tree_1_tree_2_branch_copy ⊥)`x1 =
        cfst`(iterate k tree_1_tree_2_branch_copy ⊥)`x1'

> b y resolve_tac prems 1; (* Die (aufgefaltete) Voraussetzung paßt *)
No subgoals!
```

Für endliche Datentypen, hier `empty`

```
> goalprems thy "⟦ x ≠ ⊥; (∀n. empty_take n‘x = ⊥) ∨          (** finite_lemmas1a **)
                   (∃n. empty_take n‘x = x) ⟧ ⟹ ∃n. empty_take n‘x = x";
 1. ⟦ x ≠ ⊥; (∀n. empty_take n‘x = ⊥) ∨ (∃n. empty_take n‘x = x) ⟧ ⟹
    ∃n. empty_take n‘x = x

> b y etac disjE 1; (* Aufspalten der Disjunktion *)
 1. ⟦ x ≠ ⊥; ∀n. empty_take n‘x = ⊥ ⟧ ⟹ ∃n. empty_take n‘x = x
 2. ⟦ x ≠ ⊥; ∃n. empty_take n‘x = x ⟧ ⟹ ∃n. empty_take n‘x = x

> b y atac 2;        (* Zweites Teilziel trivial nach Annahme *)
 1. ⟦ x ≠ ⊥; ∀n. empty_take n‘x = ⊥ ⟧ ⟹ ∃n. empty_take n‘x = x
```

Herleitung des Widerspruchs `x ≠ ⊥ ∧ x = ⊥`

```
> b y etac notE 1;  (* "⟦ ¬ ?P; ?P ⟧ ⟹ ?R" *)
 1. ∀n. empty_take n‘x = ⊥ ⟹ x = ⊥

> b y resolve_tac take_lemmas 1; (* Verwendung der take-Lemmata *)
 1. ⋀n. ∀n. empty_take n‘x = ⊥ ⟹ empty_take n‘x = empty_take n‘⊥

> b y asm_simp_tac take_ss 1; (* Fertig nach spezialisierter Annahme
                              (* und Striktheit des take-Funktionals *)
No subgoals!


> goal thy "∀n x::empty. empty_take n‘x = ⊥ ∨ empty_take n‘x = x"; (* finite_lemma1b *)
  b y rtac allI 1;
 1. ⋀n. ∀x. empty_take n‘x = ⊥ ∨ empty_take n‘x = x
```

Simultane Induktion für alle Teiltypen (hier nur einer)

```
> b y nat_ind_tac "n" 1;
 1. ⋀n. ∀x. empty_take 0‘x = ⊥ ∨ empty_take 0‘x = x
 2. ⋀n n1. ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x ⟹
          ∀x. empty_take (Suc n1)‘x = ⊥ ∨ empty_take (Suc n1)‘x = x
```

Lösen des Basisfalls, Aufspalten der Annahmen und Konklusion im Induktionsschritt

```
> b y simp_tac take_ss 1:
  b y TRY(safe_tac(empty_cs addSEs[conjE] addSIs[conjI]));
 1. ⋀n n1. ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x ⟹
          ∀x. empty_take (Suc n1)‘x = ⊥ ∨ empty_take (Suc n1)‘x = x
```

Für alle Teiltypen (hier nur einer)

```
> b y simp_tac take_ss 1; (* Vereinfachung, hier unnötig *)
Warning: same as previous level
 1. ⋀n n1. ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x ⟹
          ∀x. empty_take (Suc n1)‘x = ⊥ ∨ empty_take (Suc n1)‘x = x
```

Fallunterscheidung über den Teiltyp

```
> b y rtac allI 1;
  b y res_inst_tac [("x","x")] (hd casess) 1;
 1. ⋀n n1 x.
       ⟦ ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x; x = ⊥ ⟧ ⟹
       empty_take (Suc n1)‘x = ⊥ ∨ empty_take (Suc n1)‘x = x
 2. ⋀n n1 x e.
       ⟦ ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x; x = silly‘e; e ≠ ⊥ ⟧ ⟹
       empty_take (Suc n1)‘x = ⊥ ∨ empty_take (Suc n1)‘x = x
```

Undefinierter Fall

```
>   b y asm_simp_tac take_ss 1; (* Mit Striktheit von take *)
 1. ⋀n n1 x e.
       ⟦ ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x; x = silly‘e; e ≠ ⊥ ⟧ ⟹
       empty_take (Suc n1)‘x = ⊥ ∨ empty_take (Suc n1)‘x = x
```

Für alle Konstruktoren (hier nur einer)

```
>   b y asm_simp_tac take_ss 1;
 1. ⋀n n1 x e.
       ⟦ ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x; x = silly‘e; e ≠ ⊥ ⟧ ⟹
       silly‘(empty_take n1‘e) = ⊥ ∨ silly‘(empty_take n1‘e) = silly‘e
```

Für alle strikten, rekursiven Argumente (hier eines)

```
>     b y eres_inst_tac [("x","e")] all_dupE 1; (* Spezialisierung der Annahme *)
      b y etac disjE 1;                         (* Aufspaltung der Disjunktion *)
 1. ⋀n n1 x e.
       ⟦ x = silly‘e; e ≠ ⊥; ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x;
          empty_take n1‘e = ⊥ ⟧ ⟹
       silly‘(empty_take n1‘e) = ⊥ ∨ silly‘(empty_take n1‘e) = silly‘e
 2. ⋀n n1 x e.
       ⟦ x = silly‘e; e ≠ ⊥; ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x;
          empty_take n1‘e = e ⟧ ⟹
       silly‘(empty_take n1‘e) = ⊥ ∨ silly‘(empty_take n1‘e) = silly‘e
```

Erstes Disjunktionsglied im ersten Teilziel lösbar mit der Annahme `empty_take n1‘e = ⊥`

```
> b y asm_simp_tac (HOL_ss addsimps con_rews) 1;
 1. ⋀n n1 x e.
       ⟦ x = silly‘e; e ≠ ⊥; ∀x. empty_take n1‘x = ⊥ ∨ empty_take n1‘x = x;
          empty_take n1‘e = e ⟧ ⟹
       silly‘(empty_take n1‘e) = ⊥ ∨ silly‘(empty_take n1‘e) = silly‘e
```

Das zweite Disjunktionsglied ist mit der Annahme `empty_take n1‘e = e` lösbar

```
> b y asm_simp_tac take_ss 1;
No subgoals!
```

6.4.4 (Un-)Endlichkeit, Induktion

Für endliche Datentypen, hier `empty`

```
> goalw thy axs_finite_def "empty_finite x";                          (** finites **)
  b y case_UU_tac take_rews 1 "x"; (* Undefinierter Fall, per Striktheit *)
 1. x ≠ ⊥ ⟹ ∃n. empty_take n‘x = x
```

Der Rest folgt mittels Prädikatenlogik aus den beiden Lemmata

```
> b y eresolve_tac finite_lemmas1a 1;
 1. (∀n. empty_take n‘x = ⊥) ∨ (∃n. empty_take n‘x = x)

> b y step_tac HOL_cs 1 THEN step_tac HOL_cs 1;
  b y cut_facts_tac [hd atomize finite_lemma1b] 1;
 1. ⋀n. ⟦ empty_take (?n8 n)‘(?x8 n) = ⊥ ∨ empty_take (?n8 n)‘(?x8 n) = ?x8 n;
         empty_take n‘x ≠ ⊥ ⟧ ⟹ empty_take (?n6 n)‘x = x
> b y fast_tac HOL_cs 1;
No subgoals!
```

Für unendliche Datentypen, hier `tree_1`

```
read_instantiate_sg (sign_of thy) [("P", "tree_1_finite x1")] excluded_middle;
val it = "¬ tree_1_finite x1 ∨ tree_1_finite x1" : thm                (** finites **)
```

Für endliche Datentypen, hier `empty`

```
> val prems = goal thy "⟦ P ⊥; ⋀e. ⟦ e ≠ ⊥; P e ⟧ ⟹ P (silly‘e) ⟧ ⟹ P x" (* ind *)
  b y TRY(safe_tac HOL_cs); (* Aufspalten des Ziels nach Teiltypen (hier nur einer) *)
 1. P x
```

Erster Teiltyp `(*` Einsetzen der Endlichkeit `"∃n. empty_take n‘?x = ?x" *)`

```
> b y rtac (rewrite_rule axs_finite_def (hd finites) RS exE) 1;
  b y etac subst 1; (* Statt x wird dessen endliche Approximation verwendet *)
 1. ⋀xa. P (empty_take xa‘x)

> b y rtac (hd (atomize finite_ind)) 1; (* Rückführung auf endliche Induktion *)
 1. ⋀x. P ⊥
 2. ⋀x e. ⟦ e ≠ ⊥; P e ⟧ ⟹ P (silly‘e)
```

Erster Konstruktor (hier nur einer)

```
>   b y resolve_tac prems 1; (* Undefinierter Fall nach Voraussetzung P ⊥ *)
 1. ⋀x e. ⟦ e ≠ ⊥; P e ⟧ ⟹ P (silly‘e)
>   b y resolve_tac prems 1; (* Voraussetzung ⋀e. ⟦ e ≠ ⊥; P e ⟧ ⟹ P (silly‘e) *)
 1. ⋀x e. ⟦ e ≠ ⊥; P e ⟧ ⟹ e ≠ ⊥
 2. ⋀x e. ⟦ e ≠ ⊥; P e ⟧ ⟹ P e
```

Für jedes strikte und jedes rekursive Argument (hier je eines)

```
>     b y atac 1; b y atac 1; (* Lösen nach Annahme *)
No subgoals!
```

Für unendliche Datentypen, hier `tree_1`

```
> val prems = goal thy "⟦ adm P1; adm P2; adm P3; P1 ⊥;                  (* ind *)
P1 leaf; ⋀b. ⟦ b ≠ ⊥; P3 b ⟧ ⟹ P1 (node‘b); ⋀b t. ⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P1 (b in_f t);
P2 ⊥; ⋀t. ⟦ t ≠ ⊥; P1 t ⟧ ⟹ P2 (#‘t);
P3 ⊥; ⋀a t b. ⟦ a ≠ ⊥; t ≠ ⊥; P2 t; P3 b ⟧ ⟹ P3 <a:t|>>b ⟧ ⟹ P1 x1 ∧ P2 x2 ∧ P3 x3";
 1. P1 x1 ∧ P2 x2 ∧ P3 x3

> b y rtac (hd axs_reach RS subst) 1; b y rtac (hd (tl axs_reach) RS subst) 1;
  b y rtac (hd (tl (tl axs_reach)) RS subst) 1; (* Auffalten mit dem reach-Axiom *)
  b y quant_tac 1; (* Einführung eines Allquantors, für eine starke Induktionsvoraussetzung *)
 1. ∀x xa xb. P1 (cfst‘(fix‘tree_1_tree_2_branch_copy) ‘xb) ∧
       P2 (cfst‘(csnd‘(fix‘tree_1_tree_2_branch_copy))‘xa) ∧
       P3 (csnd‘(csnd‘(fix‘tree_1_tree_2_branch_copy))‘x )
```

Fixpunktinduktion mit `"⟦ adm ?P; ∀n. ?P (iterate n ?F ⊥) ⟧ ⟹ ?P (fix‘?F)"`

```
> b y rtac (adm_impl_admw RS wfix_ind) 1;
 1. adm (λu. ∀x xa xb. P1 (cfst‘u‘xb) ∧ P2 (cfst‘(csnd‘u)‘xa) ∧ P3 (csnd‘(csnd‘u)‘x))
 2. ∀n x xa xb. P1 (cfst‘(iterate n tree_1_tree_2_branch_copy ⊥) ‘xb) ∧
         P2 (cfst‘(csnd‘(iterate n tree_1_tree_2_branch_copy ⊥))‘xa) ∧
         P3 (csnd‘(csnd‘(iterate n tree_1_tree_2_branch_copy ⊥))‘x )
```

Beweis der Zulässigkeit

```
> b y REPEAT_DETERM(rtac adm_all2 1); (* "(⋀y. adm (?P y)) ⟹ adm (λx. ∀y. ?P y x)" *)
 1. ⋀x xa xb. adm (λxc. P1 (cfst‘xc‘xb) ∧ P2 (cfst‘(csnd‘xc)‘xa) ∧ P3 (csnd‘(csnd‘xc)‘x))
 2. (* ... *)

> b y REPEAT_DETERM(
       TRY(rtac adm_conj 1) THEN (* "⟦ adm ?P ; adm ?Q ⟧ ⟹ adm (λx. ?P x ∧ ?Q x)" *)
       rtac adm_subst 1     THEN (* "⟦ cont ?t; adm ?P ⟧ ⟹ adm (λx. ?P (?t x))" *)
       cont_tacR 1          THEN (* Beweis mit Stetigkeit von cfst und csnd *)
       resolve_tac prems 1);     (* und den Zulässigkeitsvoraussetzungen *)
  b y strip_tac 1;
 1. ⋀n x xa xb. P1 (cfst‘(iterate n tree_1_tree_2_branch_copy ⊥) ‘xb) ∧
         P2 (cfst‘(csnd‘(iterate n tree_1_tree_2_branch_copy ⊥))‘xa) ∧
         P3 (csnd‘(csnd‘(iterate n tree_1_tree_2_branch_copy ⊥))‘x )
```

Rückführung auf endliche Induktion

```
> b y rtac (rewrite_rule axs_take_def finite_ind) 1;
 1. ⋀n x xa xb.                          P1 ⊥
 2. ⋀n x xa xb.                          P1 leaf
 3. ⋀n x xa xb b.   ⟦ b ≠ ⊥; P3     ⟧ ⟹ P1 (node‘b)
 4. ⋀n x xa xb b t. ⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P1 (b in_f t)
 5. ⋀n x xa xb.                      P2 ⊥
 6. ⋀n x xa xb t. ⟦ t ≠ ⊥; P1 t ⟧ ⟹ P2 (#‘t)
 7. ⋀n x xa xb.                                     P3 ⊥
 8. ⋀n x xa xb a t b. ⟦ a ≠ ⊥; t ≠ ⊥; P2 t; P3 b ⟧ ⟹ P3 <a:t|>>b
```

Erster Teiltyp

```
> b y resolve_tac prems 1;              (* Voraussetzung P1 ⊥ *)
 1. ⋀n x xa xb. P1 leaf
⋮
 7. (* ... *)
```

Erster Konstruktor, keine Argumente

```
>   b y resolve_tac prems 1;            (* Voraussetzung P1 leaf *)
 1. ⋀n x xa xb b. ⟦ b ≠ ⊥; P3 b ⟧ ⟹ P1 (node‘b)
⋮
 6. (* ... *)
```

Zweiter Konstruktor

```
>   b y resolve_tac prems 1; (* Voraussetzung ⋀b. ⟦ b ≠ ⊥; P3 b ⟧ ⟹ P1 (node‘b); *)
 1. ⋀n x xa xb b. ⟦ b ≠ ⊥; P3 b ⟧ ⟹ b ≠ ⊥
 2. ⋀n x xa xb b. ⟦ b ≠ ⊥; P3 b ⟧ ⟹ P3 b
⋮
 7. (* ... *)
```

Lösen nach Annahme, für jedes strikte Argument (hier eines), und
für jedes rekursive Argument (hier eines)

```
>     b y atac 1 THEN atac 1;
 1. ⋀n x xa xb b t. ⟦ b ≠ ⊥; t ≠ ⊥ ⟧ ⟹ P1 (b in_f t)
⋮
 5. (* ... *)
```

Dritter Konstruktor, analog

```
>   b y resolve_tac prems 1; b y atac 1 THEN atac 1;
 1. ⋀n x xa xb.                     P2 ⊥
 2. ⋀n x xa xb t. ⟦ t ≠ ⊥; P1 t ⟧ ⟹ P2 (#‘t)
 3. (* ... *)
 4. (* ... *)
```

Zweiter Teiltyp, analog

```
> b y resolve_tac prems 1;
    b y resolve_tac prems 1; b y atac 1 THEN atac 1;
 1. ⋀n x xa xb.                                         P3 ⊥
 2. ⋀n x xa xb a t b. ⟦ a ≠ ⊥; t ≠ ⊥; P2 t; P3 b ⟧ ⟹ P3 <a:t|>>b
```

Dritter Teiltyp, analog

```
> b y resolve_tac prems ;
    b y resolve_tac prems 1; b y atac 1 THEN atac 1 THEN atac 1 THEN atac 1;
No subgoals!
```

6.4.5 Bisimulation

```
> goalw thy [ax_bisim_def] "tree_1_tree_2_branch_bisim R ⟶           (* coind_lemma *)
       (∀x1 x1' x2 x2' x3 x3'. fst R x1 x1'⟶ fst (snd R) x2 x2'⟶ snd (snd R) x3 x3'⟶
       tree_1_take n‘x1 = tree_1_take n‘x1' ∧ tree_2_take n‘x2 = tree_2_take n‘x2' ∧
       branch_take n‘x3 = branch_take n‘x3')";
  b y rtac impI 1;
 1. (∀x1 x1'.       fst R x1 x1' ⟶ x1 = ⊥ ∧ x1' = ⊥ ∨ x1 = leaf ∧ x1' = leaf ∨
        (∃b b'. b ≠ ⊥ ∧ b' ≠ ⊥ ∧ snd (snd R) b b' ∧ x1 =  node‘b ∧ x1' = node‘b') ∨
        (∃b t . b ≠ ⊥ ∧ t ≠ ⊥                   ∧ x1 = (b in_f t) ∧ x1' = (b in_f t))) ∧
    (∀x2 x2'. fst (snd R) x2 x2' ⟶ x2 = ⊥ ∧ x2' = ⊥ ∨
        (∃t t'. t ≠ ⊥ ∧ t' ≠ ⊥ ∧ fst R t t'          ∧ x2 = #‘t ∧ x2' = #‘t')) ∧
    (∀x3 x3'. snd (snd R) x3 x3' ⟶ x3 = ⊥ ∧ x3' = ⊥ ∨
        (∃a t b t' b'. a ≠ ⊥ ∧ t ≠ ⊥ ∧ t' ≠ ⊥ ∧ fst (snd R) t t' ∧ snd (snd R) b b' ∧
                                                x3 = <a:t|>>b ∧ x3' = <a:t'|>>b')) ⟹
    ∀x1 x1' x2 x2' x3 x3'. fst R x1 x1' ⟶ fst (snd R) x2 x2' ⟶ snd (snd R) x3 x3' ⟶
       tree_1_take n‘x1 = tree_1_take n‘x1' ∧ tree_2_take n‘x2 = tree_2_take n‘x2' ∧
       branch_take n‘x3 = branch_take n‘x3'

> b y nat_ind_tac "n" 1;                  (* Induktion über die Approximationstiefe *)
  b y simp_tac (HOL_ss addsimps take_rews) 1;                   (* Induktionsanfang *)
  b y safe_tac HOL_cs;          (* Aufspalten des Induktionsschritts nach Teiltypen *)
 1. ⋀n1 x1 x1' x2 x2' x3 x3'. ⟦ ∀x1 x1' x2 x2' x3 x3'. fst R x1 x1' ⟶
        fst(snd R) x2 x2'⟶ snd(snd R) x3 x3'⟶ tree_1_take n1‘x1 = tree_1_take n1‘x1' ∧
        tree_2_take n1‘x2 = tree_2_take n1‘x2' ∧  branch_take n1‘x3 = branch_take n1‘x3';
       ∀x1 x1'. fst R x1 x1'       ⟶ x1 = ⊥ ∧ x1' = ⊥ ∨ (* ... *)
       ∀x2 x2'. fst (snd R) x2 x2' ⟶ x2 = ⊥ ∧ x2' = ⊥ ∨ (* ... *)
       ∀x3 x3'. snd (snd R) x3 x3' ⟶ x3 = ⊥ ∧ x3' = ⊥ ∨ (* ... *)
       fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧
             ⟹ tree_1_take (Suc n1)‘x1 = tree_1_take (Suc n1)‘x1'
 2. (* ... *) ⟹ tree_2_take (Suc n1)‘x2 = tree_2_take (Suc n1)‘x2'
 3. (* ... *) ⟹ branch_take (Suc n1)‘x3 = branch_take (Suc n1)‘x3'
```

Erster Teiltyp

```
> b y etac allE 1; back();                    (* Ausnutzen der Bisimulationseigenschaft *)
  b y etac allE 1; back(); back(); back();    (* ∀x1 x1'. fst R x1 x1' ⟶ ... *)
  b y eres_inst_tac [("P1", "fst(R) x1 x1'")] (mp RS disjE) 1;
                (* "⟦ ?P1 ⟶ ?P ∨ ?Q; ?P1; ?P ⟹ ?R; ?Q ⟹ ?R ⟧ ⟹ ?R *)
 1. ⋀n1 x1 x1' x2 x2' x3 x3'. ⟦ (* ... *); fst R x1 x1'; (* ... *) ⟧ ⟹ fst R x1 x1'
 2. ⋀n1 x1 x1' x2 x2' x3 x3'. ⟦ (* ... *); x1 = ⊥ ∧ x1' = ⊥ ⟧ ⟹
       tree_1_take (Suc n1)‘x1 = tree_1_take (Suc n1)‘x1'
 3. ⋀n1 x1 x1' x2 x2' x3 x3'. ⟦ (* ... *); x1 = leaf ∧ x1' = leaf ∨
          (∃b b'. b ≠ ⊥ ∧ b' ≠ ⊥ ∧ snd (snd R) b b' ∧ x1 = node‘b ∧ x1' = node‘b') ∨
          (∃b t. b ≠ ⊥ ∧ t ≠ ⊥ ∧ x1 = (b in_f t) ∧ x1' = (b in_f t)) ⟧ ⟹
       tree_1_take (Suc n1)‘x1 = tree_1_take (Suc n1)‘x1'
 4. (* ... *)
 5. (* ... *)
```

Lösen der einfachen Fälle mit Gleichungslogik, unter Verwendung der Annahmen

```
> b y TRY(safe_tac HOL_cs);
 1. ⋀n1 x1 x1' x2 x2' x3 x3'. ⟦ ∀x1 x1' x2 x2' x3 x3'. fst R x1 x1' ⟶
       fst(snd R) x2 x2'⟶ snd(snd R) x3 x3'⟶ tree_1_take n1`x1 = tree_1_take n1`x1' ∧
       tree_2_take n1`x2 = tree_2_take n1`x2' ∧  branch_take n1`x3 = branch_take n1`x3';
        (* ... *); (* ... *);
         fst R (node`b) (node`b'); fst (snd R) x2 x2'; snd (snd R) x3 x3';
         b ≠ ⊥; b' ≠ ⊥; snd (snd R) b b' ⟧ ⟹
      tree_1_take (Suc n1)`(node`b) = tree_1_take (Suc n1)`(node`b')
 2. (* ... *)
 3. (* ... *)
```

Lösen der rekursiven Fälle unter Verwendung der Induktionsvoraussetzung `∀x1 x1' x2 x2' x3 x3'.`
`fst R x1 x1' ⟶ fst (snd R) x2 x2'⟶ snd (snd R) x3 x3'⟶ ...`

```
> b y REPEAT(CHANGED(asm_simp_tac (HOL_ss addsimps take_rews) 1));
 1. (* ... *) ⟹ tree_2_take (Suc n1)`x2 = tree_2_take (Suc n1)`x2'
 2. (* ... *) ⟹ branch_take (Suc n1)`x3 = branch_take (Suc n1)`x3'
```

Zweiter Teiltyp, analog

```
> b y etac allE 1; back(); back();
  b y etac allE 1; back(); back(); back();
  b y eres_inst_tac [("P1", "fst(snd(R)) x2 x2'")](mp RS disjE) 1;
  b y TRY(safe_tac HOL_cs);
  b y REPEAT(CHANGED(asm_simp_tac (HOL_ss addsimps take_rews) 1));
 1. (* ... *) ⟹ branch_take (Suc n1)`x3 = branch_take (Suc n1)`x3'
```

Dritter Teiltyp, analog

```
> b y etac allE 1; back(); back(); back();
  b y etac allE 1; back(); back(); back();
  b y eres_inst_tac [("P1", "snd(snd(R)) x3 x3'")](mp RS disjE) 1;
  b y TRY(safe_tac HOL_cs);
  b y REPEAT(CHANGED(asm_simp_tac (HOL_ss addsimps take_rews) 1));
No subgoals!
```

```
> goalprems thy "⟦ tree_1_tree_2_branch_bisim R; fst R x1 x1';             (* coind *)
       fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧ ⟹ x1 = x1' ∧ x2 = x2' ∧ x3 = x3'";
  b y TRY(safe_tac HOL_cs); (* Aufspalten des Ziels nach Teiltypen *)
 1. ⟦ tree_1_tree_2_branch_bisim R;
     fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧ ⟹ x1 = x1'
 2. ⟦ tree_1_tree_2_branch_bisim R;
     fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧ ⟹ x2 = x2'
 3. ⟦ tree_1_tree_2_branch_bisim R;
     fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧ ⟹ x3 = x3'
```

Erster Teiltyp

```
> b y rtac (hd take_lemmas) 1;
  b y cut_facts_tac [coind_lemma] 1;
 1. ⋀n. ⟦ tree_1_tree_2_branch_bisim R;
         fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3';
         tree_1_tree_2_branch_bisim (?R8 n) ⟶ (∀x1 x1' x2 x2' x3 x3'.
         fst (?R8 n) x1 x1'⟶ fst (snd (?R8 n)) x2 x2'⟶ snd (snd (?R8 n)) x3 x3'⟶
              tree_1_take (?n8 n)`x1 = tree_1_take (?n8 n)`x1' ∧
              tree_2_take (?n8 n)`x2 = tree_2_take (?n8 n)`x2' ∧
              branch_take (?n8 n)`x3 = branch_take (?n8 n)`x3') ⟧ ⟹
       tree_1_take n`x1 = tree_1_take n`x1'
 2. (* ... *)
 3. (* ... *)

> b y fast_tac HOL_cs 1; (* Reine Prädikatenlogik *)
 1. ⟦ tree_1_tree_2_branch_bisim R;
      fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧ ⟹ x2 = x2'
 2. ⟦ tree_1_tree_2_branch_bisim R;
      fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧ ⟹ x3 = x3'
```

Zweiter Teiltyp, analog

```
> b y rtac (hd (tl take_lemmas)) 1;
  b y cut_facts_tac [coind_lemma] 1;
  b y fast_tac HOL_cs 1;
 1. ⟦ tree_1_tree_2_branch_bisim R;
      fst R x1 x1'; fst (snd R) x2 x2'; snd (snd R) x3 x3' ⟧ ⟹ x3 = x3'
```

Dritter Teiltyp, analog

```
> b y rtac (hd (tl (tl take_lemmas))) 1;
  b y cut_facts_tac [coind_lemma] 1;
  b y fast_tac HOL_cs 1;
No subgoals!
```

Die umseitige Abbildung 6.1 zeigt die logischen Abhängigkeiten der Theoreme in Form eines Hasse-Diagramms (d.h. durch die Transitivität der Abhängigkeitsrelation erschließbare Kanten sind weggelassen). Die Knoten des Graphen sind die Namen von Axiomen, Theoremen und Theoremgruppen, wie sie in den oben aufgeführten Beweisen verwendet werden, also wie sie in der Datei `theorems.ML` auftreten. Diese Namen sind den vom Datentyp-Paket erzeugten Namen in den Strukturen für den Anwender sehr ähnlich, nur ist die Struktur der Theoreme hier detaillierter.

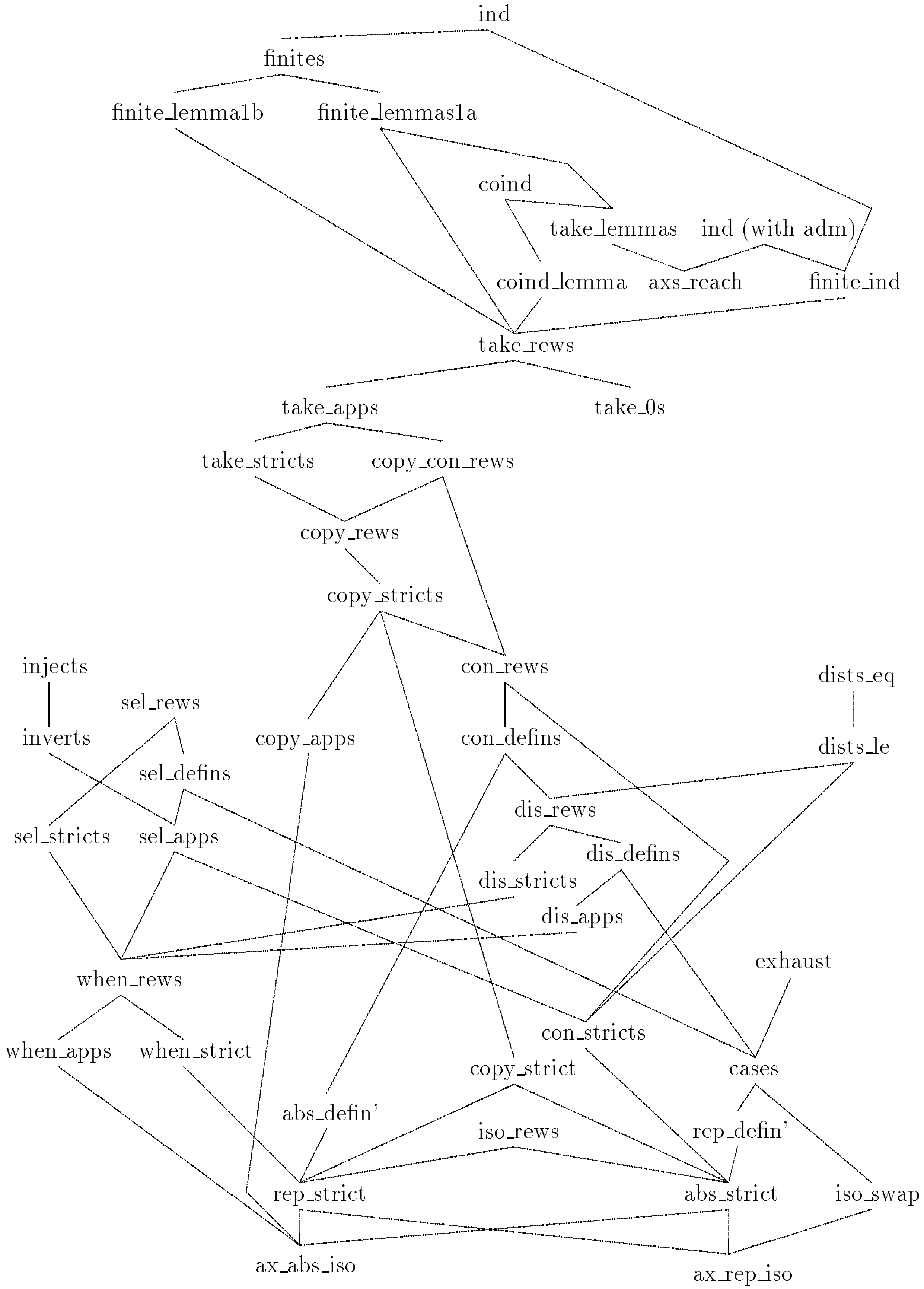


Abbildung 6.1: Abhängigkeiten der Theoreme

Kapitel 7

Implementierung

7.1 Theorie-Extender (allgemein)

Dieser Abschnitt beschreibt das Vorgehen bei der Programmierung eines Theorie-Extenders für Isabelle, nachdem es hierfür noch keine allgemein verfügbare Dokumentation gibt.

Ein Theorie-Extender ist ein ML-Programm, das den Theorie-Lesemechanismus von Isabelle so erweitert, daß in einer Theoriebeschreibung (also in einer `.thy`-Datei) neue Sektionen mit den gewünschten Deklarationsmöglichkeiten neben den bisherigen wie `types`, `consts`, usw. auftreten können. Für jede neue Sektion ist ein Parser zur Verfügung zu stellen, der eine syntaktische Analyse ihres Inhalts vornimmt und die Erweiterung der zu erzeugenden Theorie um die deklarierten Elemente veranlaßt. Außerdem kann der Extender die vollständig erzeugte Theorie (nach dem Einlesen aller Sektionen) verwenden, um ML-Befehle damit ausführen zu lassen, also zum Beispiel Axiome auszulesen, Theoreme zu beweisen und ähnliches.

7.1.1 Theorielesevorgang

Die Verarbeitung einer Isabelle-Theoriedatei *Name*`.thy` erfolgt indirekt, indem der Gesamtparser eine Zwischendatei `.`*Name*`.thy.ML` mit ML-Befehlen zur Definition einer Struktur *Name* mit der Theorie und allen sonstigen dazugehörigen Werten aufbaut. Anschließend wird diese Datei dem ML-Übersetzer zur Verarbeitung übergeben, wie beim ML-Befehl `use`, und wieder gelöscht, sofern `delete_tmpfiles` gesetzt ist.

Der Anfang der Zwischendatei hat folgende Form:

```
val thy = mk_base [Thy "Basistheorie"] "Name" (* ... *);

structure Name =
struct

(* ... *)

val thy = thy
```

Die Zeile `val thy = thy` ist die Basis zum auch auf syntaktischer Ebene inkrementellen Aufbau der Theorie durch die einzelnen Sektionsparser, aus denen sich der Gesamtparser zusammensetzt. Jeder solche Parser liefert eine Zeichenkette mit vorangestelltem Operator `|>`, die eine ML-Funktion mit dem Typ `theory -> theory` zur Theorieerweiterung darstellt und hinter die genannte Zeile in die Datei aufgenommen wird. Diese Funktion wird also per *Rechts*applikation

```
infix  |>;
fun (x |> f) = f x;
```

auf den bisherigen Theorie-Ausdruck angewendet, so daß ein um die in der entsprechenden Sektion deklarierten Elemente ergänzter Theorie-Ausdruck entsteht, der seinerseits durch die Parser der noch folgenden Sektionen erweitert werden kann. Dann folgt als Abschluß der Theoriedefinition der Text

```
|> add_thyname "Name";

val () = store_theory (thy, "Name");
```

Dahinter kommen der Reihe nach ebenfalls von den Sektionsparsern erzeugte Zeichenketten (mit ML-Befehlen), gefolgt vom Ende der Datei

```
(* ... *)
end;
```

Die Abbildung 7.1 zeigt Ausschnitte aus der Datei, die für die Beispiel-Theorie aus Abschnitt 5.3 erzeugt wird.

7.1.2 Parsing

Der Parser (mit ML-Typ `token list -> (string * string)*token list`) erhält als Eingabe eine Tokenliste und liefert ein Paar von Zeichenketten zur Theorieerweiterung und ML-Befehlsausführung und die restliche Tokenliste zurück. Er arbeitet auf einer sehr niedrigen Ebene, insbesondere sind ihm (außer der reinen Tokenliste) noch keine Informationen über die zu erzeugende Theorie verfügbar.

Der Typ `token` ist in der Datei `src/Pure/Thy/thy_parse.ML` der Isabelle-Distribution definiert, ebenso die Konstruktoren zum Aufbau eines Parsers, wie sie auch in [Paulson 92] beschrieben sind, z.B.

- `$$ : string -> token list -> string*token list`
  als Teilparser für lexikalische Einheiten,
- `op -- : ('a -> 'b*'c) * ('c -> 'd*'e) -> 'a -> ('b * 'd)*'e`
  als (Infix-)Kombinator für sequentielle Komposition, oder
- `op >> : ('a -> 'b * 'c) * ('b -> 'd) -> 'a -> 'd * 'c`
  zur Abbildung des von einem Teilparser zurückgelieferten Werts.

```
val thy = mk_base [Thy "HOLCF"] "Example" true;

structure Example =
struct

local
 val parse_ast_translation = [];
 val parse_translation     = [];
 val print_translation     = [];
 val print_ast_translation = [];
in

val thy = thy

|> add_trfuns
(parse_ast_translation, parse_translation, print_translation, print_ast_translation)

val (thy, empty_equations) = thy |> Domain_Extender.add_domain ("empty",
        [(("empty", []),
                [("silly", ThyOps.Mixfix NoSyn,
                        [(false, "empty_sel_1_1", Type("empty", []))])])]);
val thy = thy

val (thy, tree_1_tree_2_branch_equations) = thy |> Domain_Extender.add_domain
                                                    ("tree_1_tree_2_branch", (* ... *))
val thy = thy

|> add_types   [("Tree", 1, NoSyn)            , ("Forest", 1, NoSyn)            ]
|> add_tyabbrs []

|> add_arities [("Tree", [["pcpo"]], ["pcpo"]), ("Forest", [["pcpo"]], ["pcpo"])]

|> add_consts
[("eTree"   , "'a Tree"                          , NoSyn),
 ("mkTree"  , "'a      -> 'a Forest -> 'a Tree"  , NoSyn),
 ("eForest" , "'a Forest"                        , NoSyn),
 ("mkForest", "'a Tree -> 'a Forest -> 'a Forest", NoSyn)]

|> Domain_Extender.add_gen_by (("Tree_Forest", true ),
                 (["Tree", "Forest"], [["eTree", "mkTree"], ["eForest", "mkForest"]]))

|> Domain_Extender.add_gen_by (("branch", false), (["branch"], [["con"]]))

|> add_thyname "Example";

val () = store_theory (thy, "Example");
```

```
structure empty  = struct
val abs_iso      =  get_axiom thy "empty_abs_iso";
(* ... *)
val dis_defs     = [get_axiom thy "is_silly_def"];
val (iso_rews, exhaust, cases, when_rews, copy_rews,con_rews, sel_rews, dis_rews,
     dists_le, dists_eq, inverts, injects) =
    Domain_Theorems.theorems thy (hd(funpow 0 tl empty_equations), empty_equations);
val [take_def  ] = [get_axiom thy "empty_take_def"];
val [finite_def] = [get_axiom thy "empty_finite_def"];
val [reach     ] = [get_axiom thy "empty_reach"];
val  bisim_def   =  get_axiom thy "empty_bisim_def";
val (take_rews, [take_lemma], [finite], finite_ind, ind, coind) =
Domain_Theorems.comp_theorems thy ("empty", empty_equations, [cases], copy_rews, con_rews);
val rews = iso_rews @ when_rews @ copy_rews @
           con_rews @ sel_rews @ dis_rews @ dists_le @ dists_eq;
end; (* struct *)

structure tree_1 = (* ... *)

structure tree_2 = (* ... *)

structure branch = (* ... *)

structure tree_1_tree_2_branch = struct
val copy_def     =  get_axiom thy "tree_1_tree_2_branch_copy_def";
val take_defs    = [get_axiom thy "tree_1_take_def", get_axiom thy "tree_2_take_def",
                    get_axiom thy "branch_take_def"];
val finite_defs  = [get_axiom thy "tree_1_finite_def",
(* ... *)
val bisim_def    =  get_axiom thy "tree_1_tree_2_branch_bisim_def";
val (take_rews, take_lemmas, finites, finite_ind, ind, coind) =
Domain_Theorems.comp_theorems thy ("tree_1_tree_2_branch", tree_1_tree_2_branch_equations,
[tree_1.cases,tree_2.cases,branch.cases],tree_1.copy_rews@tree_2.copy_rews@branch.copy_rews,
tree_1.con_rews @tree_2.con_rews @branch.con_rews);
val rews = tree_1.rews@tree_2.rews@branch.rews@ take_rews;
end; (* struct *)

val Tree_Forest_induct = get_axiom thy "Tree_Forest_induct";

val branch_induct      = get_axiom thy "branch_induct";

end;
end;
```

Abbildung 7.1: Beispiel: Zwischendatei

7.1.3 Theorieerweiterung

Zur inkrementellen Erweiterung einer Theorie dienen ML-Funktionen mit dem allgemeinen Typ `... -> theory -> theory`. Isabelle stellt dazu für alle Arten von Theoreelementen eine Erweiterungsfunktion zur Verfügung, die in der Datei `src/Pure/thm.ML` aufgeführt ist, nämlich

```
val add_classes  : (class * class list                    ) list -> theory -> theory
val add_classrel : (class * class                         ) list -> theory -> theory
val add_defsort  :  sort                                         -> theory -> theory
val add_types    : (string * int * mixfix                 ) list -> theory -> theory
val add_tyabbrs  : (string * string list * string * mixfix) list -> theory -> theory
val add_tyabbrs_i: (string * string list * typ    * mixfix) list -> theory -> theory
val add_arities  : (string * sort list * sort             ) list -> theory -> theory
val add_consts   : (string * string * mixfix              ) list -> theory -> theory
val add_consts_i : (string * typ    * mixfix              ) list -> theory -> theory
val add_syntax   : (string * string * mixfix              ) list -> theory -> theory
val add_syntax_i : (string * typ    * mixfix              ) list -> theory -> theory
val add_trfuns   : (string * (ast list -> ast))   list *
                   (string * (term list -> term)) list *
                   (string * (term list -> term)) list *
                   (string * (ast list -> ast))   list           -> theory -> theory
val add_trrules  : (string * string) trrule                list -> theory -> theory
val add_trrules_i:  ast              trrule                list -> theory -> theory
val add_axioms   : (string * string                       ) list -> theory -> theory
val add_axioms_i : (string * term                         ) list -> theory -> theory
```

Die Namen dieser Funktionen sollten selbsterklärend sein; solche mit Endung `_i` sind "interne" Varianten, bei denen gewisse Argumente in abstrakter Form (d.h. nicht mehr als String) erwartet werden. Durch Anwendung der Funktionen entsteht eine Zwischentheorie mit dem Pseudo-Namen `#`; erst durch die Funktion `add_thyname : string -> theory -> theory` wird die Theorie anwendbar.

Der Parser kann im ersten Teil des erzeugten Zeichenkettenpaars Aufrufe direkt für diese Funktionen generieren, aber auch für eigene Funktionen, die spezielle Aufgaben mit der bisherigen Zwischentheorie ausführen, bei Bedarf intern die Erweiterungsfunktionen anwenden und die ergänzte Theorie zurückliefern.

7.1.4 ML-Befehlsausführung

Im zweiten Teil des erzeugten Zeichenkettenpaars kann der Parser beliebige ML-Befehle erzeugen, die die Möglichkeit haben, auf die fertige Theorie über den Namen `thy` zuzugreifen. Typischerweise werden dabei Axiome aus der Theorie gelesen (mittels `get_axiom : theory -> string -> thm`) und durch eine `val`-Deklaration an ML-Namen zur späteren Verwendung gebunden. Auch können komplexe Funktionen aufgerufen werden, die z.B. die Signatur der Theorie analysieren, Theoreme bewiesenn und dergleichen.

Etwas umständlich ist die Erzeugung von ML-Ausdrücken und -Befehlen in Form von Zeichenketten. Deshalb gibt es hierfür Hilfsfunktionen, die in der Datei `src/Pure/Thy/thy_parse.ML` als Elemente der Struktur `ThyParse` deklariert sind, wie zum Beispiel `mk_list : string list -> string`.

7.1.5 Einbindung eines Extenders

Isabelle verfügt über eine Struktur namens `ThySynData` mit der in der Datei `src/Pure/Thy/thy_syn.ML` definierten Signatur

```
signature THY_SYN_DATA =
sig
  val user_keywords: string list
  val user_sections: (string * (token list ->
                       (string * string) * token list)) list
end;
```

Dort sind alle bereits vorhandenen Theorie-Extender eingetragen. Um einen Extender zur Behandlung einer neuen Sektion dem Isabelle-System bekannt zu machen, ist die Liste `user_sections` um das Paar aus dem Sektionsnamen und dem Parser zu erweitern, ebenso die Liste `user_keywords` um die neu hinzukommenden lexikalischen Einheiten. Die so erzeugte neue Version von `ThySynData` ist dann dem Funktor `ThySynFun` zu übergeben, der daraus die Struktur `ThySyn : THY_SYN` erzeugt. Durch den Befehl `init_thy_reader ()` wird der Parser für `use_thy`-Aufrufe verfügbar.

7.2 Das Datentyp-Paket

In diesem Abschnitt wird die spezielle Implementierung des Datentyp-Pakets als Theorie-Extender anhand von Auszügen aus den Quelltexten erläutert.

Das Datentyp-Paket ist nach den verschiedenen internen Aufgabenbereichen (Parsing, Tests, Theorieerweiterung durch Syntax und Axiome, Theoremgenerierung und -beweis) in mehrere Module untergliedert. Jedes dieser Module wird durch eine ML-Struktur repräsentiert, die in einer eigenen Datei abgelegt ist. Zudem gibt es eine Hauptdatei, `ROOT.ML`, die die im folgenden beschriebenen Dateien in der gegebenen Reihenfolge lädt und die beiden durch das Datentyp-Paket behandelten neuen Sektionen `domain` und `generated` in den Theorie-Lesemechanismus von Isabelle einbettet. Die folgenden Abschnitte beschreiben die Aufgabe und den Inhalt der einzelnen Module und zeigen die Grobstruktur der entsprechenden Quelldateien.

7.2.1 Datei `library.ML`

Diese Datei enthält Hilfsfunktionen und -werte, auf die von mehreren anderen Teilen des Datentyp-Pakets zugegriffen wird. Dazu gehören zunächst allgemeine Funktionen zur Behandlung von Listen (elementweise Abbildung und die oft benötigte rechtsassoziative Faltung), Tripeln und binären Verzweigungen, sowie zur Zerlegung von Theoremen in deren Konjunktionsglieder.

Alles weitere ist spezifischer und daher in der Struktur `Domain_Library` abgelegt. Dies sind eine spezielle Ausnahmebehandlung, die Bearbeitung von Namen, insbesondere die Generierung paarweise verschiedener Argumentnamen für Konstruktoren, Zugriffsfunktionen auf die Elemente der wichtigsten Datenstruktur des Datentyp-Pakets (nämlich die Konstruktorlisten, die alle Informationen zu der verarbeiteten Datentypgleichung beinhalten), und Hilfen zur

Erzeugung abstrakter Typ- und Termbäume. Für fast alle Arten von Verzweigungen und Endknoten gibt es (zum Teil Infix-)Konstruktoren, die den Aufbau der Bäume möglichst knapp, übersichtlich und nahe der konkreten Syntax ermöglichen. Desweiteren wird eine η-Konversion für Terme stetiger Funktionen und der Rumpf des `when`-Funktionals zur Verfügung gestellt.

```
(* ----- general support ---------------------------------------------------------- *)
fun mapn (* ... *)
(* ... *)
fun foldr' f l = (* ...*)
(* ... *)
fun first  (x,_,_) = x; fun second (_,x,_) = x; fun third  (_,_,x) = x;
(* ... *)
fun bin_branchr f1 f2 y is j = (* ... *)
fun atomize thm = (* ... *)

(* ----- specific support for domain ----------------------------------------------- *)
structure Domain_Library = struct

  exception Impossible of string;
  fun Imposs msg = raise Impossible ("Domain:"^msg);

  (* ----- name handling ----- *)
  val strip_esc = (* ... *)
  (* ... *)
  fun dis_name_ con = "is_"^ (strip_esc   con);
  fun mk_var_names types : string list = (* ... *)
  (* ... *)

  (* ----- constructor list handling ----- *)
  type cons = (string *                (* operator name of constr *)
              ((bool*int)*             (*  (lazy,recursive element or ~1) *)
                string*                (*   selector name    *)
                string)                (*   argument name    *)
              list);                   (* argument list      *)
  type eq = (string *          (* name      of abstracted type *)
             typ list) *       (* arguments of abstracted type *)
            cons list;         (* represented type, as a constructor list *)

  val rec_of    = snd o first;
  (* ... *)
  fun is_one_con_one_arg p cons = length cons = 1 andalso
        let val args = snd(hd cons) in length args = 1 andalso p (hd args) end;

  (* ----- support for type and mixfix expressions ----- *)
  fun mk_tvar s = TFree("'"^s,["pcpo"]);
  fun mk_typ t (S,T) = Type(t,[S,T]);
  infixr 5 -->;
  infixr 6 ~>; val op ~> = mk_typ "->";
  val NoSyn' = ThyOps.Mixfix NoSyn;
```

```
  (* ----- support for term expressions ----- *)
  fun %% s = Const(s,dummyT);
  (* ... *)
  fun mk_All  (x,P) = %%"all" $ mk_lam(x,P); (* meta universal quantification *)
  infixr 0 ===>;fun S ===> T = %%"==>" $ S $ T;
  (* ... *)
  fun lift tfn = foldr (fn (x,t)=> (mk_trp(tfn x) ===> t));
  (* ... *)
  val UU = %%"UU";
  fun strict f = f`UU === UU;
  (* ... *)
  fun cont_eta_contract (* ... *)
  fun when_body cons funarg = (* ... *)

end; (* struct *)
```

7.2.2 Datei syntax.ML

Die Funktion `add_syntax` als alleiniges Element der Struktur `Domain_Syntax` fügt zur Syntax der übergebenen Theorie die in Abschnitt 5.1 beschriebenen Typdeklarationen (inklusive der Aritäten), Funktionalitätsvereinbarungen (für alle Operationen, Prädikate usw.) und Syntax-Tranformationsregeln für eine Datentypgleichung hinzu. Dazu bestimmt die lokale Funktion `calc_syntax` den zu je einer Teilgleichung gehörenden Anteil.

```
structure Domain_Syntax = struct

local
  open Domain_Library;

  fun calc_syntax dtypeprod ((dname, typevars), (cons':
                          (string * ThyOps.cmixfix * (bool*string*typ) list) list)) =
  let
  (* ----- constants concerning the isomorphism ---------------------------------- *)
  (* ... *)
  val const_rep  = (dname^"_rep" ,             dtype  ~> dtype2, NoSyn');
  val const_abs  = (dname^"_abs" ,             dtype2 ~> dtype , NoSyn');
  val const_when = (* ... *)
  val const_copy = (dname^"_copy", dtypeprod ~> dtype  ~> dtype , NoSyn');

  (* ----- constants concerning the constructors, discriminators and selectors ---- *)
  (* ... *)
  val consts_con = map con cons';
  val consts_dis = map dis cons';
  val consts_sel = flat(map sel cons');

  (* ----- constants concerning induction ----------------------------------------- *)
  val const_take   = (dname^"_take"  , Type("nat",[]) --> dtype ~> dtype, NoSyn');
  val const_finite = (dname^"_finite", dtype-->HOLogic.boolT           , NoSyn');

  (* ----- case translation ------------------------------------------------------- *)
  local open Syntax in val case_trans = (* ... *) end;
```

```
  in ([const_rep, const_abs, const_when, const_copy] @
       consts_con @ consts_dis @ consts_sel @
      [const_take, const_finite],
     [case_trans])
  end; (* let *)

(* ----- putting all the syntax stuff together ------------------------------------ *)

in (* local *)

  fun add_syntax (comp_dname,eqs': ((string * typ list) *
                (string * ThyOps.cmixfix * (bool*string*typ) list) list) list) thy'' =
  let
    (* ... *)
    val thy_types   = map (thy_type  o fst) eqs';
    val thy_arities = map (thy_arity o fst) eqs';
    (* ... *)
    val const_copy   = (comp_dname^"_copy"  ,funprod ~> funprod       , NoSyn');
    val const_bisim  = (comp_dname^"_bisim" ,relprod --> HOLogic.boolT, NoSyn');
    val ctt           = map (calc_syntax funprod) eqs';
    val add_cur_ops_i = ThyOps.add_ops_i {curried=true, strict=false, total=false};
  in thy'' |> add_types      thy_types
           |> add_arities    thy_arities
           |> add_cur_ops_i (flat(map fst ctt))
           |> add_cur_ops_i ((if length eqs'>1 then [const_copy] else[])@[const_bisim])
           |> add_trrules_i (flat(map snd ctt))
  end; (* let *)

end; (* local *)
end; (* struct *)
```

7.2.3 Datei `axioms.ML`

Die Struktur `Domain_Axioms` enthält zum einen die Funktion `add_axioms`, die mittels `calc_axioms` analog zu `add_syntax` die Axiome zu einer Gleichung einfügt, zum anderen die Funktion `add_induct`, die die Induktionsregel zu einer Gen-By-Regel erzeugt, wie ebenfalls in Abschnitt 5.1 beschrieben.

```
structure Domain_Axioms = struct

local
  open Domain_Library;
  (* ... *)

  fun calc_axioms comp_dname (eqs : eq list) n (((dname,_),cons) : eq)=
  let

  (* ----- axioms and definitions concerning the isomorphism ---------------------- *)
  (* ... *)
  val ax_abs_iso  = (* ... *)
  val ax_rep_iso  = (* ... *)
  val ax_when_def = (* ... *)
  val ax_copy_def = (* ... *)
```

```
  (* ----- definitions concerning the constructors, discriminators and selectors -- *)
  val axs_con_def = (* ... *)
  val axs_dis_def = (* ... *)
  val axs_sel_def = (* ... *)

  (* ----- axiom and definitions concerning induction ---------------------------- *)
  (* ... *)
  val ax_reach    = (* ... *)
  val ax_take_def = (* ... *)
  val ax_finite_def=(* ... *)

  in [ax_abs_iso, ax_rep_iso, ax_when_def, ax_copy_def] @
      axs_con_def @ axs_dis_def @ axs_sel_def @
     [ax_reach, ax_take_def, ax_finite_def]
  end;

in (* local *)

  fun add_axioms (comp_dname, eqs : eq list) thy' = let
    (* ... *)
    val ax_copy_def  = (* ... *)
    val ax_bisim_def = (* ... *)
    val thy_axs = flat (mapn (calc_axioms comp_dname eqs) 0 eqs) @
                  (if length eqs>1 then [ax_copy_def] else []) @ [ax_bisim_def];
  in thy' |> add_axioms_i (infer_types thy' thy_axs) end;

  fun add_induct ((tname,finite),(typs,cnstrs)) thy' = let
    (* ... *)
    val induct = (* ... *)
  in thy' |> add_axioms_i (infer_types thy' [induct]) end;

end; (* local *)
end; (* struct *)
```

7.2.4 Datei `theorems.ML`

Der komplexeste Teil des Datentyp-Pakets sind die Funktionen `theorems` und `comp_theorems`, die die beiden Elemente der Struktur `Domain_Theorems` bilden. Sie bestimmen für die gegebene Gleichung zu je einer Teilgleichung bzw. zur Gesamtgleichung die entsprechenden Eigenschaften des Datentyps und beweisen diese. Die Funktion `pg : thm list -> term -> tactic list -> thm` führt dabei in der aktuellen Theorie für eine Formel, die als trivial getypter Term gegeben wird, eine vollständige Typinferenz durch und beweist die Formel nach dem Einsetzen der gegebenen Definitionen und Prämissen mittels der ebenfalls gegebenen Taktikliste.

Die Hilfstaktiken und Lemmata sind bereits in den Abschnitten 6.1 und 6.2 aufgeführt, während die Beweisverfahren in den Abschnitten 6.3 und 6.4 beschrieben sind. Wie flexibel sich die Isabelle-Taktiken kombinieren lassen, wird zum Beispiel bei den Beweisen von `cases` und `finite_ind` deutlich, wo die Folge der Beweisschritte stark vom Inhalt der Konstruktorlisten abhängt (siehe Quelltexte im Anhang B).

```
structure Domain_Theorems = struct

local
  open Domain_Library;
  (* ... *)

  (* ----- general proof facilities -------------------------------------------- *)
  fun inferT sg pre_tm = #2(Sign.infer_types sg (K None) (K None) [] true
                         ([pre_tm],propT));

  fun pg'' thy defs t = let val sg = sign_of thy;
                            val ct = Thm.cterm_of sg (inferT sg t);
                        in prove_goalw_cterm defs ct end;
  fun pg'  thy defs t tacsf = pg'' thy defs t (fn []   => tacsf
                                      | prems=> (cut_facts_tac prems 1)::tacsf);
  (* ... *)

  (* ----- general proofs ------------------------------------------------------ *)
  (* ... *)
in

  fun theorems thy (((dname,_),cons) : eq, eqs :eq list) =
  let

    val dummy = writeln ("Proving isomorphism properties of domain "^dname^"...");
    val pg = pg' thy;

    (* ----- getting the axioms and definitions -------------------------------- *)
    (* ... *)
    (* ----- theorems concerning the isomorphism ------------------------------- *)
    (* ... *)
    val iso_rews  = [ax_abs_iso,ax_rep_iso,abs_strict,rep_strict];
    (* ... *)
    val cases     = (* ... *)
    val exhaust   = (* ... *)
    (* ... *)
    val when_rews = when_strict::when_apps;

    (* ----- theorems concerning the constructors, discriminators and selectors --- *)
    (* ... *)
    val dis_rews  = dis_stricts @ dis_defins @ dis_apps;
    (* ... *)
    val con_rews  = con_stricts @ con_defins;
    (* ... *)
    val sel_rews  = sel_stricts @ sel_defins @ sel_apps;
    val dists_le  = (* ... *)
    val dists_eq  = (* ... *)
    val inverts   = (* ... *)
    val injects   = (* ... *)

    (* ----- theorems concerning one induction step ---------------------------- *)
    (* ... *)
    val copy_rews = copy_strict::copy_apps @ copy_stricts;
```

```
    in      (iso_rews, exhaust, cases, when_rews,
             con_rews, sel_rews, dis_rews, dists_le, dists_eq, inverts, injects,
             copy_rews)
    end; (* let *)

  fun comp_theorems thy (comp_dname, eqs: eq list, casess, con_rews, copy_rews) =
  let

    val dummy = writeln ("Proving induction properties of domain "^comp_dname^"...");
    val pg = pg' thy;
    (* ... *)

    (* ----- getting the composite axiom and definitions ------------------------- *)
    (* ... *)
    (* ----- theorems concerning finite approximation and finite induction -------- *)
    (* ... *)
    take_rews = atomize take_stricts @ take_0s @ atomize take_apps;
    local
      (* ... *)
      val is_emptys   = (* list of flags indicating equation parts *)
      val is_finite   = (* whether whole type is finite *)
    in (* local *)
      val finite_ind  = (* ... *)
      val take_lemmas = (* ... *)

    (* ----- theorems concerning finiteness and induction ------------------------ *)

      val (finites,ind) = if is_finite then
        let
          (* ... *)
          val finite_lemmas1a = (* ... *)
          val finite_lemma1b  = (* ... *)
          val finites         = (* ... *)
        in
          (finites, (* ... *))
        end else
        ((* ... *), (* ... *))
    end; (* local *)

    (* ----- theorem concerning coinduction --------------------------------------- *)

    local
      (* ... *)
      val coind_lemma = pg [ax_bisim_def] (* ... *)
    in
      val coind = pg [] (* ... *)
    end; (* local *)

in (take_rews, take_lemmas, finites, finite_ind, ind, coind)

end; (* let *)
end; (* local *)
end; (* struct *)
```

7.2.5 Datei `extender.ML`

Diese Datei ist das Kernstück des Theorie-Extenders, denn hier laufen die Aufrufe zur Erweiterung der Theorie um den Syntax- und Axiomanteil zusammen, und zwar in den zwei Elementen `add_domain` und `add_gen_by` der Struktur `Domain_Extender`. Außerdem prüfen diese beiden Funktionen zunächst mittels `check_domain` bzw. `check_gen_by`, ob die Kontexbedingungen aus Abschnitt 4.1 bzw. 4.2 erfüllt sind, und bauen die Konstruktorlisten von dem vom Parser gelieferten Format in die für die Axiom- und Theoremerzeugung notwendige Form um.

```
structure Domain_Extender =
struct

local

  open Domain_Library;

  (* ----- general testing and preprocessing of constructor list ------------------ *)

  fun check_domain(eqs':((string * typ list) *
                (string * ThyOps.cmixfix * (bool*string*typ) list) list) list) = let
    val cons' = flat (map snd (map fst eqs'));
    val test_dupl_typs = (case duplicates (map fst dtnvs) of
        [] => false | dups => error ("Duplicate types: " ^ commas_quote dups));
    (* ... *)
    in true end; (* let *)

  fun check_gen_by thy' (typs': string list,cnstrss': string list list) = let
    (* ... *)
    val cnstrss = let (* ... *)
    in map (map (fn cn => let val (args,res) = args_result_type (type_of cn) in
                      (cn,mk_var_names args,(args,res)) end)) cnstrss'
        : (string *                     (* operator name of constr *)
           string list *                (* argument name list *)
           (typ list *                  (* argument types *)
            typ))                       (* result type *)
          list list end;
    fun test_equal_type tn (cn,_,(_,rt)) = fst (type_name_vars rt) = tn orelse
                               error("Inappropriate result type for constructor "^cn);
    val typs = map (fn (tn, cnstrs) => (map (test_equal_type tn) cnstrs;
                                        snd(third(hd(cnstrs))))) (typs'~~cnstrss);
    (* ... *)
  in (typs, flat cnstrss) end;

(* ----- calls for building new thy and thms ------------------------------------- *)

in

  fun add_domain (comp_dname,eqs') thy'' = let
    val ok   = check_domain eqs';
    val thy' = thy'' |> Domain_Syntax.add_syntax (comp_dname,eqs');
    (* ... *)
```

```
    val eqs = map (fn (dtnvs,cons') => (dtnvs,cons cons')) eqs' : eq list;
    val thy         = thy' |> Domain_Axioms.add_axioms (comp_dname,eqs);
  in (thy,eqs) end;

  fun add_gen_by ((tname,finite),(typs',cnstrss')) thy' = let
   val (typs,cnstrs) = check_gen_by thy' (typs',cnstrss');
  in
   Domain_Axioms.add_induct ((tname,finite),(typs,cnstrs)) thy' end;

end (* local *)
end (* struct *)
```

7.2.6 Datei `interface.ML`

In dieser Datei werden die Parser für die beiden neuen Sektionen `domain` und `generated` definiert und in die Struktur `ThySynData` und `ThySyn` eingetragen, wie im obigen Abschnitt 7.1 erklärt. Einen breiten Raum nimmt hier die Erzeugnung der Zeichenketten für die Theorieerweiterung (Aufruf von `add_domain` bzw. `add_gen_by`) und die ML-Befehlsausführung (Binden von Axiom- und Theoremnamen, sowie Aufruf von `theorems` und `comp_theorems`) ein.

```
structure ThySynData: THY_SYN_DATA = (* overwrites old version of ThySynData !!!!!! *)

struct
  local
    open ThyParse;
    open Domain_Library;

  (* ----- generation of bindings for axioms and theorems in trailer of .thy.ML - *)

    fun gt_ax         name   = "get_axiom thy " ^ quote name;
    fun gen_val dname name   = "val "^name^" = "^ gt_ax (dname^"_"^name)^";";
    (* ... *)

    fun gen_domain eqname num ((dname,_), cons') = let
      (* ... *)
      in
        "structure "^dname^" = struct\n"^
        cat_lines(map (gen_val dname) axioms1)^"\n" ^
        (* ... *) ^
        "val ("^ theorems ^") =\n\
        \Domain_Theorems.theorems thy "^eqname^";\n"^(* ... *)
      end;

    fun mk_domain (eqs'') = let
      (* ... *)

      (* ----- string for calling add_domain -------------------------------------- *)
      val thy_ext_string = let (* ... *)
      in "val (thy, "^comp_dname^"_equations) = thy |> Domain_Extender.add_domain \n"
         ^ mk_pair(quote comp_dname,
                   mk_list(map (fn (t,cs)=> mk_pair (mk_tnv t,mk_conslist cs)) eqs'))
```

```
        ^ ";\nval thy = thy"
      end;

      (* ----- string for calling (comp_)theorems and producing the structures ---- *)
      val val_gen_string =  let (* ... *)
      in
        implode (separate "end; (* struct *)\n\n"
                   (mapn (fn n => gen_domain (eqname n) num) 0 eqs')) ^
        (* ... *) ^
        "val ("^ comp_theorems ^") =\n\
        \Domain_Theorems.comp_theorems thy " ^
        (* ... *) ^
        "end; (* struct *)"
      end;
    in (thy_ext_string, val_gen_string) end;

    fun mk_gen_by (finite,eqs) = let (* ... *) in
        ("|> Domain_Extender.add_gen_by "
         ^ mk_pair(mk_pair(quote tname,makestring (finite:bool)), (* ... *)) ,
         "val "^tname^"_induct = " ^gt_ax (tname^"_induct")^";") end;

  (* ----- parser for domain declaration equation -------------------------------- *)
  (* ... *)
  in
    val domain_decl = (enum1 "," (con_typ --$$ "="  -- !!
                                  (enum1 "|" domain_cons)))            >> mk_domain;
    val gen_by_decl = (optional ($$ "finite" >> K true) false) --
                      (enum1 "," (ident   --$$ "by" -- !!
                                  (enum1 "|" (name >> strip_quotes)))) >> mk_gen_by;
  end; (* local *)

  val user_keywords = "lazy"::"by"::"finite"::
                  (**)filter_out (fn s=>s="lazy" orelse s="by" orelse s="finite")(**)
                      ThySynData.user_keywords;
  val user_sections = ("domain", domain_decl)::("generated", gen_by_decl)::
                  (**)filter_out (fn (s,_)=>s="domain" orelse s="generated")(**)
                      ThySynData.user_sections;
end; (* struct *)

structure ThySyn = ThySynFun(ThySynData); (* overwrites old version of ThySyn!!!!!! *)
```

Kapitel 8

Anwendung des Datentyp-Pakets

Dieses Kapitel zeigt Anwendungs- und Erweiterungsmöglichkeiten des Datentyp-Pakets auf, zunächst an einem einfachen Beispiel, dann im Rahmen größerer aktueller Projekte.

8.1 Anwendungsbeispiel

Das typische Beispiel schlechthin für rekursive Datenstrukturen ist die endliche Liste. Daran wird hier das Vorgehen bei der Definition eines Datentyps mit seinen Operationen und beim Beweisen bzw. Verwenden seiner Eigenschaften erläutert.

8.1.1 Isabelle-Theorie

Als Kontext für die Datentypspezifikationen ist eine Isabelle-Theorie zu verwenden, die von der Theorie `HOLCF` (und damit von `ccc1`) abgeleitet ist, hier `Usage` genannt. Sie ist in Abbildung 8.1 dargestellt.

```
Usage = HOLCF +

domain 'a list = "[]"                              ("[]")
               | "#" (hd :: 'a) (tl :: 'a list) (cinfixr 65)

ops curried
  "@"       ::  "'a list → 'a list → 'a list" (cinfixr 65)

defs
  append_def    "op @ ≡ fix‘(Λh xs ys.
                             case xs of []     => ys
                                      | x # xs => x # h‘xs‘ys)"
end
```

Abbildung 8.1: Datei `Usage.thy`

Die Liste mit dem parametrisierten Typnamen `'a list` verfügt über den Konstruktor `[]` für die polymorphe leere Liste und den Infix-Konstruktor `#` zum Anhängen eines Elements des Parametertyps `'a` vorn an eine Liste des Typs `'a list`, mit Selektoren `hd` und `tl`. Auf dem damit definierten Datentyp wird die Verkettung zweier Listen, der Infix-Operator `@`, deklariert und per primitiver Rekursion mittels des Fixpunktoperators `fix` definiert. Dazu ist die `case`-Translation sehr angenehm, weil sie das etwas unübersichtliche `when`-Funktional versteckt.

8.1.2 Beweisdatei

In der zur Theorie `Usage` gehörenden ML-Datei (siehe Abbildung 8.2) werden wie üblich die Beweise der mit der Theorie assoziierten Theoreme, also der charakteristischen Eigenschaften der darin definierten Konstrukte, abgelegt. Den Beweis der Eigenschaften des Datentyps `list` an sich übernimmt allerdings das Datentyp-Paket! Zu Demonstrationszwecken sind hier folgende Eigenschaften des Verkettungsoperators bewiesen: Seine Anwendung induktiv auf die leere Liste und eine Liste mit mindestens einem Element, seine Striktheit in beiden Argumenten und seine Definiertheitsbedingungen.

8.1.3 Benutzung des Datentyp-Pakets

Als eine einfache Anwendung soll die Assoziativität der Listenverkettung bewiesen werden, weil sie von fast allen bisher beschriebenen Konstrukten Gebrauch macht, wie in Abbildung 8.3 ausgeführt. Diese demonstriert den kompletten Umgang mit dem Datentyp-Paket, also das Einlesen der Isabelle-Theorie mit der Beweisdatei und den interaktiv geführten, hier mit Kommentaren versehenen Beweis.

Man beachte hierbei besonders den Umgang mit der Struktur `Usage` und seiner Substruktur `list`, sowie die Verwendung der Datentyp-Eigenschaften `ind` bzw. `con_rews` und der Eigenschaften der Listenverkettung aus der Beweisdatei.

8.2 Einsatzmöglichkeiten

Wie in der Einleitung bereits erwähnt, dient das Datentyp-Paket zur Unterstützung des Umgangs mit Datentypen im Gebiet der Spezifikation und Verifikation.

Für Spezifikationsaufgaben ist am Lehrstuhl Broy der Technischen Universität München die Sprache SPECTRUM entwickelt worden, welche aber nur eine indirekte Anbindung an ein Verifikationswerkzeug wie Isabelle hat. Daher hat man den Entschluß gefaßt, die Konzepte von SPECTRUM auf Isabelle/HOLCF zu übertragen, unter Berücksichtigung von Verbesserungen des Sprachdesigns, die sich aufgrund der Erfahrungen im Einsatz von SPECTRUM als wünschenswert erwiesen haben. Aus HOLCF entsteht so eine Spezifikationssprache mit integriertem Beweissystem, die im Gegensatz zu SPECTRUM über eine zweiwertige (statt dreiwertige) Logik höherer Stufe (statt erster Stufe) verfügt.

```
local open Usage in

  val append_bot = prove_goalw thy [append_def] "⊥ @ ys = ⊥" (fn _ => [
                                rtac (fix_eq RS ssubst) 1,
                                simp_tac (HOLCF_ss addsimps list.when_rews) 1]);
  val append_nil = prove_goalw thy [append_def] "[] @ ys = ys" (fn _ => [
                                rtac (fix_eq RS ssubst) 1,
                                simp_tac (HOLCF_ss addsimps list.when_rews) 1]);
  val append_con = prove_goalw thy [append_def]
                        "⟦ x ≠ ⊥; xs ≠ ⊥ ⟧ ⟹ (x#xs) @ ys = x#(xs@ys)" (fn prems => [
                                cut_facts_tac prems 1,
                                rtac (fix_eq RS ssubst) 1,
                                simp_tac HOLCF_ss 1,
                                eresolve_tac (list.when_rews RL [ssubst]) 1,
                                atac 1,
                                simp_tac HOLCF_ss 1,
                                rtac (fix_eq RS ssubst) 1,
                                simp_tac HOLCF_ss 1,
                                IF_UNSOLVED no_tac]);
  val append_bot2 = prove_goal thy "xs @ ⊥ = ⊥" (fn _ => [
                                res_inst_tac [("x","xs")] list.ind 1,
                                rtac append_bot 1,
                                rtac append_nil 1,
                                asm_simp_tac (HOL_ss addsimps [append_con]) 1,
                                resolve_tac list.con_rews 1]);
  val append_defin = prove_goal thy "⟦ xs ≠ ⊥; ys ≠ ⊥ ⟧ ⟹ xs @ ys ≠ ⊥" (fn prems => [
                                cut_facts_tac prems 1,
                                rtac impE 1, atac 2, atac 2,
                                res_inst_tac [("x","xs")] list.ind 1,
                                simp_tac HOL_ss 1,
                                asm_simp_tac (HOL_ss addsimps [append_nil]) 1,
                                asm_simp_tac (HOL_ss addsimps [append_con]) 1,
                                mp_tac 1,
                                rtac impI 1,
                                eresolve_tac list.con_rews 1,
                                atac 1]);
  val append_rews = [append_bot, append_nil, append_con, append_bot2, append_defin];

end
```

Abbildung 8.2: Datei `Usage.ML`

```
> use_thy "Usage"; (* Einlesen der Isabelle-Theorie *)
Reading "Usage.thy"
Loading theory "Usage"
val thy = {ProtoPure, CPure, HOL, Ord, Set, Prod, Lfp, Relation, Trancl, WF,
  Nat, Holcfb, Void, Porder0, Porder, Pcpo, Fun1, Fun2, Fun3, Cont, Cfun1,
  Cfun2, Cfun3, Cprod1, Cprod2, Cprod3, Sprod0, Sprod1, Sprod2, Sprod3,
  Ssum0, Ssum1, Ssum2, Ssum3, Sum, Lift1, Lift2, Lift3, Fix, ccc1, One, Tr1,
  Tr2} : theory
Building new grammar...
Building new grammar...
Building new grammar...
Proving isomorphism properties of domain list...
Proving induction properties of domain list...
structure Usage : sig
   val thy : theory
   val list_equations : eq list
   structure list : (* ... *)
   val total_axms : 'a list
   val strict_axms : 'a list
   val append_def : thm
end
Reading "Usage.ML"
val append_bot =                              "⊥      @ ys = ⊥"           : thm
val append_nil =                              "[]     @ ys = ys"          : thm
val append_con = "⟦ x ≠ ⊥; xs ≠ ⊥ ⟧ ⟹ (x # xs) @ ys = x # xs @ ys" : thm
val append_bot2  =                             "xs @ ⊥  = ⊥" : thm
val append_defin = "⟦ xs ≠ ⊥; ys ≠ ⊥ ⟧ ⟹ xs @ ys ≠ ⊥" : thm
val append_rews = ["⊥ @ ys = ⊥", "[] @ ys = ys",
    "⟦ x ≠ ⊥; xs ≠ ⊥ ⟧ ⟹ (x # xs) @ ys = x # xs @ ys",
    "xs @ ⊥ = ⊥", "⟦ xs ≠ ⊥; ys ≠ ⊥ ⟧ ⟹ xs @ ys ≠ ⊥"] : thm list
val it = () : unit

> open Usage; (* Die Elemente thy und list direkt zugänglich machen *)
val thy =
   {ProtoPure, CPure, HOL, Ord, Set, Prod, Lfp, Relation, Trancl, WF, Nat,
     Holcfb, Void, Porder0, Porder, Pcpo, Fun1, Fun2, Fun3, Cont, Cfun1,
     Cfun2, Cfun3, Cprod1, Cprod2, Cprod3, Sprod0, Sprod1, Sprod2, Sprod3,
     Ssum0, Ssum1, Ssum2, Ssum3, Sum, Lift1, Lift2, Lift3, Fix, ccc1, One,
     Tr1, Tr2, Usage} : theory
val list_equations = [(("list", ["'a"]), [("[]"  , []),
      ("op #", [((false, ~1), "hd", "a"), ((false, 0), "tl", "l")])])]: eq list
structure list : (* ... *)
val strict_axms = [] : 'a list
val total_axms = [] : 'a list
val append_def = "op @ ≡
                  fix`(Λh xs ys. case xs of [] => ys | x # xs => x # h`xs`ys)" : thm
```

```
> val prems = goal thy "(xs @ ys) @ zs = xs @ (ys @ zs)"; (* Beginn des Beweises *)
Level 0
(xs @ ys) @ zs = xs @ ys @ zs
 1. (xs @ ys) @ zs = xs @ ys @ zs
val prems = [] : thm list

> by(res_inst_tac [("x","xs")] list.ind 1); (* Induktion nach xs *)
Level 1
(xs @ ys) @ zs = xs @ ys @ zs
 1. (⊥ @ ys) @ zs = ⊥ @ ys @ zs
 2. ([] @ ys) @ zs = [] @ ys @ zs
 3. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs ⟧ ⟹
         ((a # l) @ ys) @ zs = (a # l) @ ys @ zs

> by(simp_tac (HOL_ss addsimps [append_bot]) 1); (* Undefinierter Fall trivial:
mehrfache Anwendung von "⊥ @ ?ys = ⊥", sowie abschließend "?t = ?t" *)
Level 2
(xs @ ys) @ zs = xs @ ys @ zs
 1. ([] @ ys) @ zs = [] @ ys @ zs
 2. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs ⟧ ⟹
         ((a # l) @ ys) @ zs = (a # l) @ ys @ zs

> by(simp_tac (HOL_ss addsimps [append_nil]) 1); (* Eigentlicher Induktionsanfang *)
Level 3
(xs @ ys) @ zs = xs @ ys @ zs
 1. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs ⟧ ⟹
         ((a # l) @ ys) @ zs = (a # l) @ ys @ zs

> by(asm_simp_tac (HOL_ss addsimps [append_con]) 1); (* Vereinfachung des Ind.Schritts durch
zweimalige Anwendung von  "⟦ ?x ≠ ⊥; ?xs ≠ ⊥ ⟧ ⟹ (?x # ?xs) @ ?ys = ?x # ?xs @ ?ys" *)
Level 4
(xs @ ys) @ zs = xs @ ys @ zs
 1. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs ⟧ ⟹
         (a # l @ ys) @ zs = a # l @ ys @ zs

> by(res_inst_tac [("Q","ys = ⊥")] classical2 1); (* Fallunterscheidung bzgl.
Definiertheit von ys notwendig, um append_con weiter anwenden zu können *)
Level 5
(xs @ ys) @ zs = xs @ ys @ zs
 1. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs; ys = ⊥ ⟧ ⟹
         (a # l @ ys) @ zs = a # l @ ys @ zs
 2. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs; ys ≠ ⊥ ⟧ ⟹
         (a # l @ ys) @ zs = a # l @ ys @ zs
```

```
> by(asm_simp_tac (HOL_ss addsimps (append_bot::append_bot2::list.con_rews)) 1);
(* Undefinierter Fall mittels Striktheit der Verkettung "⊥ @ ?ys = ⊥", "?xs @ ⊥ = ⊥"
   und des Konstruktors "?a # ⊥ = ⊥" abgehandelt *)
Level 6
(xs @ ys) @ zs = xs @ ys @ zs
 1. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs; ys ≠ ⊥ ⟧ ⟹
          (a # l @ ys) @ zs = a # l @ ys @ zs

> by(etac (append_con RS ssubst) 1); (* Jetzt ist append_con anwendbar *)
Level 7
(xs @ ys) @ zs = xs @ ys @ zs
 1. ⋀a l. ⟦ l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs; ys ≠ ⊥ ⟧ ⟹ l @ ys ≠ ⊥
 2. ⋀a l. ⟦ l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs; ys ≠ ⊥ ⟧ ⟹
          a # (l @ ys) @ zs = a # l @ ys @ zs

> by(etac append_defin 1 THEN atac 1); (* Definiertheitsvoraussetzung von
append_con ist jetzt beweisbar mittels "⟦ ?xs ≠ ⊥; ?ys ≠ ⊥ ⟧ ⟹ ?xs @ ?ys ≠ ⊥" *)
Level 8
(xs @ ys) @ zs = xs @ ys @ zs
 1. ⋀a l. ⟦ l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs; ys ≠ ⊥ ⟧ ⟹
          a # (l @ ys) @ zs = a # l @ ys @ zs

> by(asm_simp_tac HOL_ss 1); (* Jetzt greift die Induktionsvoraussetzung *)
Level 9
(xs @ ys) @ zs = xs @ ys @ zs
No subgoals!
```

Die letzten drei Schritte können auch durch einem komplexen Schritt ersetzt werden:

```
> choplev 6;
Level 6
(xs @ ys) @ zs = xs @ ys @ zs
 1. ⋀a l. ⟦ a ≠ ⊥; l ≠ ⊥; (l @ ys) @ zs = l @ ys @ zs; ys ≠ ⊥ ⟧ ⟹
          (a # l @ ys) @ zs = a # l @ ys @ zs

> by(asm_simp_tac (HOL_ss addsimps append_rews) 1);
Level 7
(xs @ ys) @ zs = xs @ ys @ zs
No subgoals!

> val append_assoc = result(); (* Namenszuweisung an das bewiesene Theorem *)
val append_assoc = "(?xs @ ?ys) @ ?zs = ?xs @ ?ys @ ?zs" : thm
```

Abbildung 8.3: Benutzung des Datentyp-Pakets

Für die praktische Einsetzbarkeit dieses Systems ist es wesentlich, eine automatische Behandlung der Semantik für die üblichen (auch benutzerdefinierten) Datentypen bereitzustellen. Die vorliegende Arbeit verwirklicht dies, so daß nun Isabelle/HOLCF als Spezifikations- und Verifikationswerkzeug (bis auf angemessenere Strukturierungsmöglichkeiten von Theorien, für die Spezifikation im Großen) komplett ist.

Damit können jetzt beispielsweise die bisherigen Arbeiten aus dem FOCUS-Projekt zur Beschreibung verteilter Systeme [Broy et al 93a] maschinell nachvollzogen und weitere Arbeiten rechnergestützt durchgeführt werden. Ganz allgemein vereinfacht das Datentyp-Paket die Handhabung von Bereichskonstruktionen für Datentypen erheblich, unter Nutzung aller Vorteile, die die Kombination von HOL mit LCF bietet.

8.3 Erweiterungsmöglichkeiten

Durch das vorliegende Datentyp-Paket sind bereits die in der Praxis wichtigsten Fälle der Verwendung von Datentypen abgedeckt, insbesondere weil auch verschränkte Rekursion möglich ist.

Ohne die darunterliegende Theorie wesentlich modifizieren zu müssen, ist noch denkbar, Exponenten zu den bisherigen Bereichskonstruktionen Summe, Produkt und Lifting hinzuzunehmen, mit deren Hilfe beliebigstellige Tupel darstellbar sind. Desweitern könnte indirekte Rekursion erlaubt werden, zumindest auf syntaktischem Weg.

Eine wesentliche Erweiterung, auch von theoretischer Seite, wäre die Hinzunahme des stetigen Funktionenraums als Bereichskonstruktion. Dies erforderte in der kategoriellen Behandlung allerdings gemischt co- und contravariante Funktoren und würde sowohl bei der Theorie als auch für die Implementierung erheblichen technischen Mehraufwand bedeuten, während die praktische Bedeutung gering sein dürfte.

Durchaus praktischem Nutzen hätte dagegen ein Zusatz, der die Definition von Funktionen auf den Datentypen per primitiver Rekursion unterstützt. Bisher sind solche Definitionen nur entweder über Anwendungsregeln als Axiome (mit der Gefahr von Inkonsistenzen) oder (etwas umständlich) über explizite Verwendung des Fixpunktoperators möglich. Ein solcher Zusatz kann in Anlehnung an die `primrec`-Sektion von HOL entwickelt werden.

Literaturverzeichnis

[Broy et al 93a]
Broy M., Dederichs Fr., Dendorfer Cl., Fuchs M., Gritzner Th., Weber R.: The Design of Distributed Systems: An Introduction to FOCUS. Revised Version. Technical Report TUM-I9202-2, Institut für Informatik, Technische Universität München, 1993.

[Broy et al 93b]
Broy M., Facchi Chr., Grosu R., Hettler R., Hussmann H., Nazareth D., Regensburger Fr., Slotosch O., Stølen K.: The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I and II. Technical Report TUM-I9311 and TUM-I9312, Institut für Informatik, Technische Universität München, 1993.

[Gordon Melham 93]
Gordon M.J.C., Melham T.F.: Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, 1993.

[Huber Oheimb Regensburger Rumpe 95]
Huber Fr., von Oheimb D., Regensburger Fr., Rumpe B.: "isa2latex" - converts a file written using the Isabelle font into a LaTeX source. Manual page for an internal tool program, Institut für Informatik, Technische Universität München, 1995.

[Hußmann Loeckx Reif 93]
Hußmann H., Loeckx J., Reif W.: Korrekte Software - Das Projekt KORSO. Tagungsband GI-Jahrestagung 1993.

[Mac Lane 72]
Mac Lane S.: Kategorien. Begriffssprache und mathematische Theorie. Springer, 1972.

[Mayr 95]
Mayr, T.: Erweiterung des Isabelle–Theorieparsers um die Sektionen "axioms" und "ops". Praktische Semesterarbeit, Institut für Informatik, Technische Universität München, 1995.

[Paulson 87]
Paulson, L.C.: Logic and Computation, Interactive Proof with Cambridge LCF. Cambridge Tracts in Theoretical Computer Science. Volume 2. Cambridge University Press, 1987.

[Paulson 92]
Paulson L.C.: ML for the Working Programmer. Cambride University Press, 1992.

[Paulson 94 I]
Paulson L.C.: Isabelle - A Generic Theorem Prover. Part I: Introduction to Isabelle. Springer LNCS 829, 1994.

[Paulson 94 II]
Paulson L.C.: Isabelle - A Generic Theorem Prover. Part II: The Isabelle Reference Manual. Springer LNCS 829, 1994.

[Paulson 94 III]
Paulson L.C.: Isabelle - A Generic Theorem Prover. Part III: Isabelle's Object Logics. Springer LNCS 829, 1994.

[Regensburger 94]
Regensburger Fr.: HOLCF: Eine konservative Erweiterung von HOL um LCF. Dissertation, Institut für Informatik, Technische Universität München, 1994.

[Regensburger 95]
Regensburger, Fr.: Syntax-Vorschlag für HOLCF. Internes Papier, Institut für Informatik, Technische Universität München, 1995.

Abbildungsverzeichnis

Anhang A

Technische Hinweise

Die Installation des Datentyp-Pakets geschieht wie folgt. Zunächst ruft man wie gewohnt den ML-Übersetzer, z.B. `PolyML`[1], mit der HOLCF-Datenbank auf. Mit dem ML-Befehl `cd` wechselt man in das Verzeichnis, in dem sich die Quelltexte des Datentyp-Pakets befinden, und führt dann den Befehl `use "ROOT.ML"` aus. Damit sind die Sektionen `domain` und `generated` für alle folgenden Aufrufe von `use_thy` bereits bekannt. Übrigens ist es für das Datentyp-Paket unerheblich, ob der spezielle graphische Zeichensatz von Isabelle verwendet wird oder nicht.

Für den Beweis der Eigenschaften komplexerer, insbesondere verschränkter, Datentypen benötigt das Datentyp-Paket etliche Sekunden. Es kann dabei auch zu Engpässen im verfügbaren Speicher und damit zu Fehlermeldungen wie

```
Run out of store - interrupting console processes
Exception- Interrupt raised
Exception- failure raised
Exception failure raised
```

kommen. In solchen Fällen ist der ML-Übersetzer anzuweisen, mehr Speicherplatz zu reservieren, wie im folgenden Beispiel-Aufruf von Isabelle/HOLCF mit PolyML auf einer Sun-Workstation unter Solaris gezeigt.

```
1 ksh-sunbroy30:/usr/stud/oheimb > poly -r ~regensbu/Save95bin/HOLCF -h 15000

Poly/ML runtime system version Solaris-2.07MX (17:08 Tue 01 Feb 1994)
Running with heap parameters (h=15000K,ib=2000K,ip=90%,mb=2000K,mp=50%)
Opening /usr/wiss/regensbu/Save95bin/HOLCF
Opening /usr/wiss/regensbu/Save95bin/HOL
Opening /usr/wiss/regensbu/Save95bin/Pure
Poly/ML Solaris-2.03X - Definition of SML Version 4 (18:29 Thu 03 Feb 1994)
Poly/ML for X - Motif Edition for Solaris 2.3
Ref: PML0001TUM (site licence)
Insitut[2] fuer Informatik, T. U. Muenchen
```

[1] Kommerzielle Produkte werden hier ohne Kennzeichung der eventuell geschützten Handelsnamen erwähnt.

[2] sic!

```
The use of this computer program is governed by the Licence from
Abstract Hardware Ltd to the Licensee for use on the Licensee's
authorised computer(s).
All copyright and intellectual property rights are reserved to
Abstract Hardware Ltd. and/or the owner of the software.

(c) Abstract Hardware Ltd. 1994

> use "ROOT"; (* Einlesen des Datentyp-Pakets *)
Higher-Order Logic of Computable Functions; with domain equations
val it = () : unit
val HOLCF2_build_completed = () : unit
val it = () : unit
```

Für das möglichst intuitive Arbeiten mit HOLCF ist der graphische 8-bit-Zeichensatz sehr angenehm, der in dieser Arbeit verwendet wird. Die Dokumentation dafür ist in der Isabelle-Distribution im Verzeichnis `8bit` zu finden, und in [Huber Oheimb Regensburger Rumpe 95] ist speziell das hier ebenfalls eingesetzte Werkzeugprogramm `isa2latex` erklärt, das zur Umsetzung von Texten mit diesem Zeichensatz in LaTeX-Format dient.

Anhang B

Quelltexte

Auf den folgenden Seiten ist der gesamte Quelltext des Datentyp-Pakets abgedruckt. Dies sind die Dateien

- `ROOT.ML`
- `library.ML`
- `syntax.ML`
- `axioms.ML`
- `theorems.ML`
- `extender.ML`
- `interface.ML`