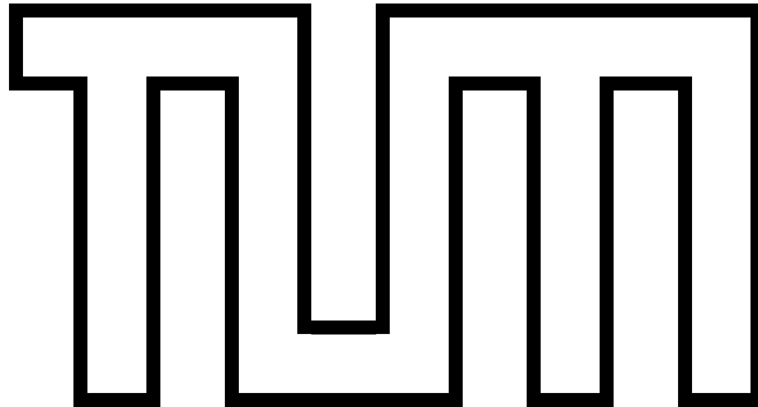


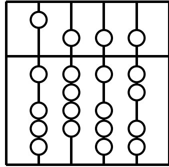
TECHNISCHE UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK

Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL

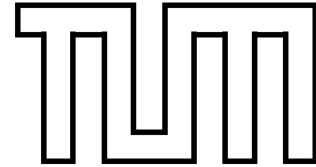
Diplomarbeit

Stefan Berghofer



INSTITUT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT
MÜNCHEN



Definitorische Konstruktion induktiver Datentypen in Isabelle/HOL

Diplomarbeit

Stefan Berghofer

Aufgabensteller	:	Prof. Tobias Nipkow, Ph.D.
Betreuer	:	Markus Wenzel
		David von Oheimb
Abgabedatum	:	24. September 1998

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 24. September 1998

(*Stefan Berghofer*)

Zusammenfassung

Induktive Datentypen stellen ein wichtiges Hilfsmittel zur Spezifikation funktionaler Programme dar. Wir beschreiben die Implementierung eines Datentyppakets für Isabelle/HOL, das neue Datentypen definitorisch einführt. Hierfür wird zunächst die repräsentierende Menge des Datentyps induktiv definiert und dann die Isomorphie zwischen dieser Menge und dem neuen Typ durch eine HOL-Typdefinition ausgedrückt. Aus dieser Definition werden dann wichtige charakteristische Eigenschaften des Datentyps wie Verschiedenheit und Injektivität der Konstruktoren sowie eine strukturelle Induktionsregel abgeleitet. Ebenfalls durch eine induktive Definition wird ein Kombinator für primitive Rekursion auf dem Datentyp konstruiert.

Durch die definitorische Vorgehensweise wird sichergestellt, daß durch das Hinzufügen von Datentypen zu Theorien die Existenz von Modellen erhalten bleibt.

Das beschriebene Datentyppaket erlaubt die Definition von verschränkt und verschachtelt rekursiven Datentypen, so daß die meisten für die praktische Anwendung wichtigen Klassen von Datentypen abgedeckt sind. Des weiteren kann das Paket leicht auf Datentypen mit beliebigem Verzweigungsgrad erweitert werden.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation und Einordnung 1

1.2 Aufgabenstellung 1

1.3 Aufbau der Arbeit 2

1.4 Einführendes Beispiel 2

2 Grundlagen und Werkzeuge **6**

2.1 Isabelle 6

2.2 Isabelle/HOL 7

2.2.1 Typen 8

2.2.2 Logische Verknüpfungen und Inferenzregeln 9

2.2.3 Definitorische Theorieerweiterung 11

2.3 Theoretische Grundlagen 13

2.3.1 Fixpunkte 13

2.3.2 Induktive Definitionen 15

3 Arten und Eigenschaften von Datentypen **24**

3.1 Grundlagen 24

3.2 Verschränkt rekursive Datentypen 26

3.3 Verschachtelt rekursive Datentypen 30

4 Konstruktion von Datentypen in HOL **36**

4.1 Auffalten verschachtelt rekursiver Definitionen 37

4.2 Konstruktion der repräsentierenden Mengen 38

4.2.1 Ein Universum für rekursive Typen 39

4.2.2 Induktive Definition der repräsentierenden Mengen 41

4.3 Beweis von Isomorphie-Eigenschaften 43

4.3.1 Definition von Repräsentationsfunktionen 45
4.3.2 Surjektivität der Repräsentationsfunktionen 47
4.3.3 Injektivität der Repräsentationsfunktionen 49
4.3.4 Wertebereich der Repräsentationsfunktionen 52
4.3.5 Umkehrung der Repräsentationsfunktionen 53
4.4 Definition der Konstruktoren 54
4.5 Repräsentationsabhängige Beweise 54
4.5.1 Verschiedenheit von Konstruktoren 54
4.5.2 Injektivität von Konstruktoren 55
4.5.3 Strukturelle Induktion 56
4.6 Repräsentationsunabhängige Beweise 60
4.6.1 Fallunterscheidung 61
4.6.2 Primitive Rekursion 61
4.6.2.1 Induktive Definition der Funktionsgraphen 61
4.6.2.2 Totalität und Eindeutigkeit 63
4.6.2.3 Definition der Kombinatoren 65
4.6.2.4 Charakteristische Gleichungen 66
4.6.3 Case-Kombinator 67

5 Erweiterungsmöglichkeiten 68

5.1 Verschachtelt rekursive Datentypen mit Funktionstypen 68
5.2 Ein Universum für Datentypen mit beliebigem Verzweigungsgrad 70
5.3 Konstruktion eines Datentyps mit unendlichem Verzweigungsgrad 72
5.3.1 Repräsentierende Menge 72
5.3.2 Konstruktoren 72
5.3.3 Induktionsregel 73
5.3.4 Primitive Rekursion 74

6 Vergleich mit anderen Arbeiten 77

6.1 Andere auf HOL basierende Datentyppakete 77
6.2 Alternative Formulierung charakteristischer Theoreme 78
6.3 Alternative Konstruktionsverfahren 82

7 Schlußbemerkungen 85

7.1 Ausblick auf coinduktive Datentypen 85
7.2 Weitere Arbeitspunkte 86

A Häufig verwendete HOL-Regeln **87**

B Beweislaufzeit einiger Datentypen **88**

C Bewiesene Theoreme **89**

„The method of 'postulating' what we want has many advantages;
they are the same as the advantages of theft over honest toil.
Let us leave them to others and proceed with our honest toil."

Bertrand Russell, Introduction to Mathematical Philosophy

Kapitel 1

Einleitung

1.1 Motivation und Einordnung

Seit einigen Jahren werden Computerprogramme in immer größerem Ausmaß zur Steuerung von Abläufen eingesetzt, die unser alltägliches Leben betreffen. So enthält z.B. schon jedes handelsübliche Mobiltelefon Software im Umfang von mehreren tausend Lines of Code. Da Software zunehmend auch in sicherheitskritischen Bereichen eingesetzt wird, wo Fehlfunktionen oder Ausfälle teilweise immense Kosten verursachen oder gar Menschenleben gefährden können, kommt der Frage der Korrektheit von Software eine wachsende Bedeutung zu. In besonders spektakulärer Weise wurde diese Problematik durch den Absturz der europäischen Trägerrakete Ariane 5 deutlich gemacht, bei dem ein Schaden von über einer Milliarde DM entstand. Auch wenn viele Praktiker behaupten, durch besonders ausführliches Testen alle Programmierfehler aufdecken zu können, kann der Nachweis der tatsächlichen Korrektheit von Software nur durch einen formalen Beweis erbracht werden.

Aufgrund der hohen Komplexität von Software-Systemen ist es jedoch sehr schwer und auch unübersichtlich, einen derartigen Korrektheitsbeweis nur auf dem Papier zu führen. In diesem Zusammenhang sind daher maschinelle Verifikationswerkzeuge von großem Nutzen. Ein solches Werkzeug ist der generische Theorembeweiser Isabelle [Paulson, 1993a]. Hierbei bedeutet *generisch*, daß Isabelle nicht auf eine einzige, fest einprogrammierte Logik beschränkt ist, sondern vielmehr eine Implementierungsplattform für eine große Klasse verschiedener Logiken darstellt. Eine solche implementierte Logik ist Isabelle/HOL, die wegen ihrer Ausdrucksstärke und ihres einfachen Typsystems eine gute Ausgangsbasis für die Verifikation von Programmen bietet.

Eine wichtiges Instrument zur Formalisierung von Programmen und deren Eigenschaften sind induktive Datentypen, also Typen, für die ein Induktionsprinzip ähnlich dem aus der Mathematik bekannten Prinzip der *vollständigen Induktion* für natürliche Zahlen gilt.

1.2 Aufgabenstellung

Für Isabelle/HOL existierte bereits ein Paket zur Beschreibung induktiver Datentypen, das die Eigenschaften des Typs wie z.B. Induktionsregeln als Axiome einführte. Diese Vorgehensweise

birgt jedoch die Gefahr in sich, daß durch die Einführung fehlerhafter Axiome Inkonsistenzen entstehen können. Sicherer und methodisch angemessener ist es, Datentypen *definitorisch* durch Angabe einer geeigneten Repräsentation einzuführen, aus der dann die gewünschten Eigenschaften des Datentyps als Theoreme abgeleitet werden können.

Die im Rahmen der Diplomarbeit zu bearbeitende Aufgabenstellung bestand nun im wesentlichen aus zwei Teilen: Zum einen sollte theoretisch untersucht werden, ob und wie bestimmte Klassen von Datentypen in Isabelle/HOL repräsentiert werden können. Außerdem sollte, aufbauend auf den angestellten theoretischen Überlegungen, die Implementierung eines definitorischen Datentyppakets für Isabelle/HOL vorgenommen werden, das auch die in der Praxis häufig vorkommenden Fälle von induktiven Datentypen mit verschränkter und verschachtelter Rekursion behandeln kann.

1.3 Aufbau der Arbeit

Kapitel 2 führt zunächst die notwendigen Grundlagen ein. Dies umfaßt zum einen das verwendete Beweissystem Isabelle sowie die Logik Isabelle/HOL, wobei insbesondere auf das Prinzip der definitorischen Theorieerweiterung eingegangen wird. Außerdem werden einige wichtige theoretische Konzepte wie Fixpunkte und induktive Definitionen behandelt.

Kapitel 3 beschreibt auf relativ abstraktem Niveau die Klassen der verschränkt und verschachtelt rekursiven Datentypen, die mit dem im Rahmen der Diplomarbeit implementierten Paket definiert werden können. Es wird hierbei ein Überblick über die durch die Definition eingeführten Konstanten und Funktionen sowie die bewiesenen charakteristischen Theoreme gegeben.

Kapitel 4 behandelt anschließend die Repräsentation der in Kapitel 3 beschriebenen Datentypen in Isabelle/HOL sowie die zum Einsatz kommenden Beweistechniken, die auch in Form von Isabelle-Beweisprozeduren implementiert wurden.

Kapitel 5 diskutiert eine mögliche Erweiterung des Datentyppakets auf eine eingeschränkte Klasse verschachtelt rekursiver Datentypen mit Funktionstypen. Das beschriebene Konstruktionsverfahren gehört zwar nicht zum Funktionsumfang des implementierten Pakets, jedoch wurde die Konstruktion für einen speziellen Typ exemplarisch in Isabelle/HOL durchgeführt.

Kapitel 6 enthält einen Vergleich mit anderen auf HOL basierenden Datentyppaketen in Hinblick auf die Formulierung der charakteristischen Theoreme und die zum Einsatz kommenden Konstruktionsverfahren.

1.4 Einführendes Beispiel

Als einführendes Beispiel wollen wir die Formalisierung einer einfachen funktionalen Programmiersprache betrachten. Diese Programmiersprache soll dabei sowohl arithmetische Ausdrücke, dargestellt durch den Typ α `aexp`, als auch boolesche Ausdrücke, dargestellt durch

den Typ α bexp enthalten. Wir beschreiben nun die Typen α aexp und α bexp durch folgende Datentypdefinition

$$
\begin{array}{llcl}
\mathsf{datatype} & \alpha\ \mathsf{aexp} & = & \mathsf{If_then_else}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & | & \mathsf{Sum}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & | & \mathsf{Diff}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & | & \mathsf{Var}\ \alpha \\
 & & | & \mathsf{Num\ nat} \\
\mathsf{and} & \alpha\ \mathsf{bexp} & = & \mathsf{Less}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & | & \mathsf{And}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{bexp}) \\
 & & | & \mathsf{Or}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{bexp})
\end{array}
$$

Durch diese Definition werden die injektiven Funktionen $\mathsf{Sum} :: \alpha\ \mathsf{aexp} \to \alpha\ \mathsf{aexp} \to \alpha\ \mathsf{aexp}$, $\mathsf{And} :: \alpha\ \mathsf{bexp} \to \alpha\ \mathsf{bexp} \to \alpha\ \mathsf{bexp}$, etc. eingeführt, die auch als *Konstruktoren* des Datentyps bezeichnet werden. Alle zulässigen Ausdrücke der Programmiersprache können mit Hilfe dieser Konstruktoren aufgebaut werden. Ein zulässiger Ausdruck ist beispielsweise

$$
\begin{array}{l}
\mathsf{If_then_else}\ (\mathsf{And}\ (\mathsf{Less}\ (\mathsf{Var}\ x)\ (\mathsf{Num}\ 2))\ (\mathsf{Less}\ (\mathsf{Num}\ 1)\ (\mathsf{Var}\ y))) \\
\quad (\mathsf{Sum}\ (\mathsf{Sum}\ (\mathsf{Num}\ 1)\ (\mathsf{Var}\ x))\ (\mathsf{Var}\ y)) \\
\quad (\mathsf{Diff}\ (\mathsf{Var}\ y)\ (\mathsf{Num}\ 2))
\end{array}
$$

wobei x und y „Variablennamen“ vom Typ α sind. Damit ist die Syntax der Programmiersprache formalisiert. Auf syntaktischer Ebene kann nun eine Substitution von Variablen durch Ausdrücke vom Typ α aexp mit Hilfe der Funktionen

$$
\begin{array}{lcl}
\mathsf{subst_aexp} & :: & (\alpha \to \alpha\ \mathsf{aexp}) \to \alpha\ \mathsf{aexp} \to \alpha\ \mathsf{aexp} \\
\mathsf{subst_bexp} & :: & (\alpha \to \alpha\ \mathsf{aexp}) \to \alpha\ \mathsf{bexp} \to \alpha\ \mathsf{bexp}
\end{array}
$$

durchgeführt werden. Diese Substitutionsfunktionen können mittels *primitiver Rekursion* durch die Gleichungen

$$
\begin{array}{lcl}
\mathsf{subst_aexp}\ f\ (\mathsf{If_then_else}\ b\ a_1\ a_2) & = & \mathsf{If_then_else}\ (\mathsf{subst_bexp}\ f\ b) \\
 & & \quad (\mathsf{subst_aexp}\ f\ a_1)\ (\mathsf{subst_aexp}\ f\ a_2) \\
\mathsf{subst_aexp}\ f\ (\mathsf{Sum}\ a_1\ a_2) & = & \mathsf{Sum}\ (\mathsf{subst_aexp}\ f\ a_1)\ (\mathsf{subst_aexp}\ f\ a_2) \\
\mathsf{subst_aexp}\ f\ (\mathsf{Diff}\ a_1\ a_2) & = & \mathsf{Diff}\ (\mathsf{subst_aexp}\ f\ a_1)\ (\mathsf{subst_aexp}\ f\ a_2) \\
\mathsf{subst_aexp}\ f\ (\mathsf{Var}\ v) & = & f\ v \\
\mathsf{subst_aexp}\ f\ (\mathsf{Num}\ n) & = & \mathsf{Num}\ n \\
 & & \\
\mathsf{subst_bexp}\ f\ (\mathsf{Less}\ a_1\ a_2) & = & \mathsf{Less}\ (\mathsf{subst_aexp}\ f\ a_1)\ (\mathsf{subst_aexp}\ f\ a_2) \\
\mathsf{subst_bexp}\ f\ (\mathsf{And}\ b_1\ b_2) & = & \mathsf{And}\ (\mathsf{subst_bexp}\ f\ b_1)\ (\mathsf{subst_bexp}\ f\ b_2) \\
\mathsf{subst_bexp}\ f\ (\mathsf{Or}\ b_1\ b_2) & = & \mathsf{Or}\ (\mathsf{subst_bexp}\ f\ b_1)\ (\mathsf{subst_bexp}\ f\ b_2)
\end{array}
$$

spezifiziert werden. Auffallend hierbei ist, daß die Funktionen mittels Fallunterscheidung über die verschiedenen Konstruktoren definiert werden, wobei als Argumente für die rekursiven Aufrufe von subst_aexp und subst_bexp immer nur die auf der linken Seite der jeweiligen Gleichung angegebenen Argumente des Konstruktors verwendet werden. Dies stellt sicher, daß die rekursiven Argumente mit jedem Aufruf strukturell kleiner werden und die Funktionen daher stets terminieren.

Die Semantik einer solchen Programmiersprache beschreibt man typischerweise durch Auswertungsfunktionen

$$\begin{array}{lll} \mathsf{eval_aexp} & :: & (\alpha \to \mathsf{nat}) \to \alpha\ \mathsf{aexp} \to \mathsf{nat} \\ \mathsf{eval_bexp} & :: & (\alpha \to \mathsf{nat}) \to \alpha\ \mathsf{bexp} \to \mathsf{bool} \end{array}$$

die den Wert eines Ausdrucks bzgl. einer bestimmten Belegung *env* der Variablen ermitteln. Ebenfalls mittels primitiver Rekursion können diese Auswertungsfunktionen nun durch die Gleichungen

$$\begin{array}{lll} \mathsf{eval_aexp}\ env\ (\mathsf{If_then_else}\ b\ a_1\ a_2) & = & \mathsf{if}\ \mathsf{eval_bexp}\ env\ b\ \mathsf{then}\ \mathsf{eval_aexp}\ env\ a_1 \\ & & \quad \mathsf{else}\ \mathsf{eval_aexp}\ env\ a_2 \\ \mathsf{eval_aexp}\ env\ (\mathsf{Sum}\ a_1\ a_2) & = & \mathsf{eval_aexp}\ env\ a_1 + \mathsf{eval_aexp}\ env\ a_2 \\ \mathsf{eval_aexp}\ env\ (\mathsf{Diff}\ a_1\ a_2) & = & \mathsf{eval_aexp}\ env\ a_1 - \mathsf{eval_aexp}\ env\ a_2 \\ \mathsf{eval_aexp}\ env\ (\mathsf{Var}\ v) & = & env\ v \\ \mathsf{eval_aexp}\ env\ (\mathsf{Num}\ n) & = & n \\ & & \\ \mathsf{eval_bexp}\ env\ (\mathsf{Less}\ a_1\ a_2) & = & \mathsf{eval_aexp}\ env\ a_1 < \mathsf{eval_aexp}\ env\ a_2 \\ \mathsf{eval_bexp}\ env\ (\mathsf{And}\ b_1\ b_2) & = & \mathsf{eval_bexp}\ env\ b_1 \wedge \mathsf{eval_bexp}\ env\ b_2 \\ \mathsf{eval_bexp}\ env\ (\mathsf{Or}\ b_1\ b_2) & = & \mathsf{eval_bexp}\ env\ b_1 \vee \mathsf{eval_bexp}\ env\ b_2 \end{array}$$

charakterisiert werden. Die Grundidee besteht hierbei darin, die Operatoren der Programmiersprache wie z.B. Sum und And auf die entsprechenden HOL-Operatoren $+$ und $\wedge$ abzubilden.

In vielen Lehrbüchern über Semantik findet man in diesem Zusammenhang oft Theoreme, die Substitutions- und Auswertungsfunktionen miteinander in Beziehung setzten. Für die hier betrachtete Programmiersprache läßt sich dieser Zusammenhang durch

$$\begin{array}{l} \mathsf{eval_aexp}\ env\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ a) = \\ \quad \mathsf{eval_aexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ a\ \wedge \\ \mathsf{eval_bexp}\ env\ (\mathsf{subst_bexp}\ (\mathsf{Var}(v := a'))\ b) = \\ \quad \mathsf{eval_bexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ b \end{array}$$

ausdrücken, wobei

$$f(x := y)\ z \equiv \mathsf{if}\ x = z\ \mathsf{then}\ y\ \mathsf{else}\ f\ z$$

Die Funktion $\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))$ ersetzt genau die Variable v in einem Ausdruck durch a' und läßt die restlichen Variablen unverändert. Das obige Theorem besagt also, daß die Auswertung jeweils das gleiche Ergebnis liefert, wenn man vor der Auswertung alle Vorkommen von v durch a' ersetzt, oder wenn man die Belegung *env* an der Stelle v geeignet verändert. Das in dieser Situation üblicherweise zur Anwendung kommende Beweisprinzip wird meist als *strukturelle Induktion* bezeichnet, da es sich am Aufbau der Ausdrücke orientiert. Für die

Datentypen α aexp und α bexp wird dies durch die Regel

$$\frac{\begin{array}{lcl} \forall b, a_1, a_2.\ P_2\ b \wedge P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_1\ (\mathsf{If_then_else}\ b\ a_1\ a_2) \\ \forall a_1, a_2.\ P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_1\ (\mathsf{Sum}\ a_1\ a_2) \\ \forall a_1, a_2.\ P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_1\ (\mathsf{Diff}\ a_1\ a_2) \\ \forall v. & & P_1\ (\mathsf{Var}\ v) \\ \forall n. & & P_1\ (\mathsf{Num}\ n) \\ \forall a_1, a_2.\ P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_2\ (\mathsf{Less}\ a_1\ a_2) \\ \forall b_1, b_2.\ P_2\ b_1 \wedge P_2\ b_2 & \Longrightarrow & P_2\ (\mathsf{And}\ b_1\ b_2) \\ \forall b_1, b_2.\ P_2\ b_1 \wedge P_2\ b_2 & \Longrightarrow & P_2\ (\mathsf{Or}\ b_1\ b_2) \end{array}}{P_1\ a \wedge P_2\ b}$$

ausgedrückt. Da die Datentypen α aexp und α bexp verschränkt rekursiv sind, müssen die Eigenschaften P_1 und P_2 simultan bewiesen werden. Für den Konstruktor If_then_else müßte beim Beweis des obigen Substitutionstheorems gemäß der Induktionsregel die Implikation

$$\begin{array}{l} \forall b, a_1, a_2. \\ \quad \mathsf{eval_bexp}\ env\ (\mathsf{subst_bexp}\ (\mathsf{Var}(v := a'))\ b) = \\ \quad \mathsf{eval_bexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ b\ \wedge \\ \quad \mathsf{eval_aexp}\ env\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ a_1) = \\ \quad \mathsf{eval_aexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ a_1\ \wedge \\ \quad \mathsf{eval_aexp}\ env\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ a_2) = \\ \quad \mathsf{eval_aexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ a_2\ \Longrightarrow \\ \qquad \mathsf{eval_aexp}\ env\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ (\mathsf{If_then_else}\ b\ a_1\ a_2)) = \\ \qquad \mathsf{eval_aexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ (\mathsf{If_then_else}\ b\ a_1\ a_2) \end{array}$$

gezeigt werden, d.h. unter der Induktionsvoraussetzung, daß die zu beweisenden Eigenschaften bereits für b, a_1 und a_2 gelten, muß die Gültigkeit der Eigenschaft für If_then_else b a_1 a_2 nachgewiesen werden. Wir können dies leicht durch folgende Umformung zeigen:

$$\begin{array}{ll} & \mathsf{eval_aexp}\ env\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ (\mathsf{If_then_else}\ b\ a_1\ a_2)) \\ = & \quad \{\text{charakteristische Gleichungen für } \mathsf{subst_aexp}\} \\ & \mathsf{eval_aexp}\ env\ (\mathsf{If_then_else}\ (\mathsf{subst_bexp}\ (\mathsf{Var}(v := a'))\ b) \\ & \quad (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ a_1)\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ a_2)) \\ = & \quad \{\text{charakteristische Gleichungen für } \mathsf{eval_aexp}\} \\ & \mathsf{if}\ (\mathsf{eval_bexp}\ env\ (\mathsf{subst_bexp}\ (\mathsf{Var}(v := a'))\ b)) \\ & \mathsf{then}\ (\mathsf{eval_aexp}\ env\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ a_1)) \\ & \mathsf{else}\ (\mathsf{eval_aexp}\ env\ (\mathsf{subst_aexp}\ (\mathsf{Var}(v := a'))\ a_2)) \\ = & \quad \{\text{Induktionsvoraussetzung}\} \\ & \mathsf{if}\ (\mathsf{eval_bexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ b) \\ & \mathsf{then}\ (\mathsf{eval_aexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ a_1) \\ & \mathsf{else}\ (\mathsf{eval_aexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ a_2) \\ = & \quad \{\text{charakteristische Gleichungen für } \mathsf{eval_aexp}\} \\ & \mathsf{eval_aexp}\ (env(v := \mathsf{eval_aexp}\ env\ a'))\ (\mathsf{If_then_else}\ b\ a_1\ a_2) \end{array}$$

Kapitel 2

Grundlagen und Werkzeuge

2.1 Isabelle

Isabelle ist ein generischer Theorembeweiser, der an der Universität Cambridge in Zusammenarbeit mit der Technischen Universität München unter der Leitung von Dr. Lawrence C. Paulson und Prof. Tobias Nipkow entwickelt wurde. Isabelle wurde in der funktionalen Programmiersprache ML implementiert und ist auf einer Vielzahl verschiedener Plattformen verfügbar.

Im Gegensatz zu anderen Beweisern wie PVS oder dem HOL-System von Gordon und Melham [Gordon, Melham, 1993] ist Isabelle nicht auf eine einzige, fest einprogrammierte Logik beschränkt. Charakteristisch für die Architektur von Isabelle ist die Aufteilung in verschiedene Schichten:

- Die Basis von Isabelle bildet die sogenannte *Metalogik*, Isabelle/Pure. Dies ist im wesentlichen eine intuitionistische Logik höherer Stufe, die auf dem einfach getypten Lambda-Kalkül aufbaut. Als logische Verknüpfungen stehen auf dieser Ebene Allquantifizierung $\bigwedge x.\, \phi$, Implikation $\phi \Longrightarrow \psi$ und Gleichheit $a \equiv b$ zur Verfügung. Mit Hilfe des Lambda-Kalküls kann unter anderem das für viele Logiken wichtige Konzept der Substitution formal präzise ausgedrückt werden. Das Typsystem von Isabelle/Pure unterstützt Polymorphie sowie das aus der Programmiersprache Haskell bekannte Konzept der *Typklassen*.

- Innerhalb dieser Metalogik von Isabelle können eine Vielzahl von *Objektlogiken* wie z.B. Higher Order Logic (HOL) oder Zermelo-Fraenkel Mengenlehre (ZF) formalisiert werden.

Objektlogik-Schlußregeln der Form

$$\frac{\phi_1 \quad \ldots \quad \phi_n}{\psi}$$

können mit Hilfe der Meta-Implikation durch

$$\phi_1 \Longrightarrow (\ldots \Longrightarrow (\phi_n \Longrightarrow \psi) \ldots) \qquad \text{bzw. abkürzend} \qquad [\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \psi$$

ausgedrückt werden.

Die Schlußregeln der Metalogik von Isabelle sind als ML-Funktionen realisiert, die Theoreme auf Theoreme abbilden. So wird z.B. die Elimination der Meta-Implikation

$$\frac{\phi \Longrightarrow \psi \quad \phi}{\psi}$$

durch die ML-Funktion `implies_elim` des Typs `thm` → `thm` → `thm` realisiert. Bei `thm` handelt es sich um einen abstrakten Datentyp, der nur von bestimmten Funktionen verarbeitet werden kann. So ist es insbesondere nicht möglich, völlig unkontrolliert beliebige Theoreme zu erzeugen. Die wohl wichtigste Meta-Inferenzregel stellt die sogenannte *Resolutionsregel*

$$\frac{[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi \qquad [\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi}{([\![\phi_1; \ldots; \phi_{i-1}; \psi_1; \ldots; \psi_m; \phi_{i+1}; \ldots; \phi_n]\!] \Longrightarrow \phi)\sigma} \qquad \text{wobei} \qquad \psi\sigma = \phi_i\sigma$$

dar: Sind zwei Theoreme $[\![\psi_1; \ldots; \psi_m]\!] \Longrightarrow \psi$ und $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ gegeben und besitzen die Prämisse ϕ_i und die Konklusion ψ einen Unifikator σ, so erhält man ein neues Theorem, indem man ϕ_i durch die Prämissen $\psi_1, \ldots, \psi_m$ ersetzt und den Unifikator σ anwendet. Diese Regel ist für die Konstruktion von Beweisen in Isabelle von entscheidender Bedeutung. Schwerpunktmäßig werden von Isabelle sogenannte *Rückwärtsbeweise* unterstützt. Beweiszustände werden hierbei durch Theoreme der Art

$$[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$$

repräsentiert, wobei ϕ das eigentliche Beweisziel ist und $\phi_1, \ldots, \phi_n$ die noch zu beweisenden Teilziele darstellen. Zum Beweis des Theorems ϕ wird ausgehend vom (trivialen) Theorem

$$\phi \Longrightarrow \phi$$

der Beweiszustand schrittweise mit Hilfe der Resolutionsregel immer weiter verfeinert, bis der Beweiszustand keine Prämissen mehr hat. Zur Durchführung dieser Beweisschritte dienen sogenannte *Taktiken*. Hierbei handelt es sich um Funktionen, die aus einem Beweiszustand eine Menge möglicher neuer Beweiszustände erzeugen. So wird z.B. die obige Resolutionsregel durch `resolve_tac` *rules i* realisiert, wobei versucht wird, die Konklusion einer der Regeln aus *rules* mit dem Teilziel ϕ_i zu unifizieren. Des weiteren ist es möglich, mittels sogenannter *Tacticals* aus einzelnen Taktiken neue Taktiken zu konstruieren. So liefert z.B. die Taktik `ALLGOALS` *tactic* neue Beweiszustände, die sich nach Anwendung von *tactic* auf alle Teilziele des aktuellen Beweiszustandes ergeben.

Eine detaillierte Beschreibung aller Isabelle zugrundeliegender Konzepte, der bereitgestellten Funktionen und deren Verwendung, sowie einiger implementierter Objektlogiken ist in [Paulson, 1993a], [Paulson, 1993b] und [Paulson, 1993c] zu finden. Für eine genauere theoretische Betrachtung der Metalogik von Isabelle sowie der maßgeblichen Design-Prinzipien sei auf [Paulson, 1989] verwiesen.

2.2 Isabelle/HOL

Eine der in Isabelle implementierten Objektlogiken ist Isabelle/HOL. HOL steht für *Higher Order Logic* und geht auf Alonzo Churchs *Simple Theory of Types* [Church, 1940] zurück.

Im Gegensatz zur herkömmlichen Prädikatenlogik erster Stufe, in der sich bekanntlich viele wichtige mathematische Begriffe wie z.B. Endlichkeit nicht charakterisieren lassen, ist HOL relativ ausdrucksstark, was diese Logik insbesondere auch für praktische Anwendungen wie beispielsweise Software- und Hardwareverifikation interessant macht. Die Implementierung von HOL in Isabelle erfolgte in Anlehnung an das HOL-Sytem von Gordon und Melham [Gordon, Melham, 1993]. Im folgenden wollen wir einige Aspekte von HOL genauer beleuchten, die im weiteren Verlauf der Arbeit von Bedeutung sein werden.

2.2.1 Typen

Typen spielen in vielen Bereichen der Informatik eine wichtige Rolle. So dienen sie in Programmiersprachen wie z.B. Java oder ML dazu, festzulegen, welche Arten von Operationen auf bestimmten Elementen zugelassen sind, weshalb viele Fehler bereits zur Übersetzungszeit erkannt werden können. So sollten z.B. Ausdrücke wie `sqrt("Hello world" + 5)` von Compilern abgewiesen werden. Auch in der Logik sind Typen zur Vermeidung von Inkonsistenzen von Bedeutung. Die Notwendigkeit von Typen wurde hierbei erstmals von *Bertrand Russell* erkannt, auf den das folgende Paradoxon zurückgeht: Gegeben sei die Menge

$$M = \{x \mid x \notin x\}$$

d.h. die Menge aller Mengen, die sich selbst nicht enthalten. Nimmt man nun an, daß $M \in M$ gilt, so folgt aufgrund der Definition von M, daß $M \notin M$ gelten muß. Nimmt man $M \notin M$ an, so folgt $M \in M$.

Isabelle/HOL übernimmt das Typsystem der Metalogik Isabelle/Pure. Es werden folgende Arten von Typen unterschieden:

Typvariablen α bzw. $\alpha :: S$
: Hierbei bezeichnet S eine *Sorte* der Gestalt $\{C_1, \ldots, C_n\}$, wobei C_i eine *Typklasse* darstellt.

Zusammengesetzte Typen $(\tau_1, \ldots, \tau_n)t$
: Hierbei ist t ein sogenannter *Typkonstruktor*. Insbesondere ist auch der Fall $n = 0$ eingeschlossen.

Funktionstypen $\tau \to \tau'$

Typen, in denen Typvariablen vorkommen, werden auch als *polymorph* bezeichnet. Sorten bzw. Typklassen dienen dazu, den Grad der Polymorphie genauer zu dosieren. Hierbei steht $\alpha :: \{C_1, \ldots, C_n\}$ für einen beliebigen Typ, der den Typklassen $C_1, \ldots, C_n$ angehört. Typklassen können dabei als eine Ansammlung von Typen mit bestimmten Eigenschaften verstanden werden. So steht z.B. $\alpha :: \{\mathsf{ord}\}$ für einen beliebigen Typ, auf dem $\leq$ eine Ordnung darstellt.

In Isabelle/HOL werden zahlreiche Typkonstruktoren definiert, unter anderem die nullstelligen Typkonstruktoren bool und nat, die einstelligen Typkonstruktoren set und list, sowie die zweistelligen Typkonstruktoren $+$ (disjunkte Summe) und $\times$ (kartesisches Produkt). Typkonstruktoren besitzen sogenannte *Aritäten*, die die Typklassen-Zugehörigkeit eines Typs $(\tau_1, \ldots, \tau_n)t$ in Abhängigkeit von der Typklassen-Zugehörigkeit der Typargumente $\tau_1, \ldots, \tau_n$ ausdrücken. So kann durch die Arität $\times :: (\mathsf{ord}, \mathsf{ord})\mathsf{ord}$ ausgedrückt werden, daß $\leq$ auf dem

Typ $\tau \times \tau'$ eine (z.B. lexikographische) Ordnung darstellt, falls $\leq$ auf τ und τ' eine Ordnung darstellt.

Auf semantischer Ebene stellt die Interpretation $[\![\tau]\!]$ eines Typs τ eine Menge aus dem sogenannten *Universum* $\mathcal{U}$ dar. Bei $\mathcal{U}$ handelt es sich um eine Menge von Mengen, die bestimmte theoretisch interessante Eigenschaften wie Abgeschlossenheit unter Bildung von Teilmengen und Funktionenräumen besitzt. Die Interpretation eines nullstelligen Typkonstruktors ist eine Menge aus $\mathcal{U}$, während die Interpretation eines mehrstelligen Typkonstruktors t eine Funktion $\mathcal{U} \times \ldots \times \mathcal{U} \to \mathcal{U}$ ist. Bei polymorphen Typen ist die Interpretation zusätzlich von der Belegung der Typvariablen abhängig. Für die Korrektheit der im folgenden Abschnitt beschriebenen Inferenzregeln von HOL ist von entscheidender Bedeutung, daß die durch $[\![\tau]\!]$ bezeichnete Menge stets nichtleer ist. Eine genauere formale Betrachtung der Semantik von Typen und Termen in HOL ist in [Gordon, Melham, 1993] zu finden.

Die folgende Abbildung zeigt einige in Isabelle/HOL definierte Konstanten und deren Typ.

$$
\begin{array}{rcl}
0 & :: & \mathsf{nat} \\
\mathsf{Suc} & :: & \mathsf{nat} \to \mathsf{nat} \\
\mathsf{length} & :: & \alpha\ \mathsf{list} \to \mathsf{nat} \\
\mathsf{hd} & :: & \alpha\ \mathsf{list} \to \alpha \\
\mathsf{tl} & :: & \alpha\ \mathsf{list} \to \alpha\ \mathsf{list} \\
\leq & :: & \alpha \to \alpha \to \mathsf{bool} \\
\in & :: & \alpha \to \alpha\ \mathsf{set} \to \mathsf{bool}
\end{array}
$$

Abbildung 2.1: Konstanten aus Isabelle/HOL

Betrachtet man das anfangs erwähnte Paradoxon, so kann festgestellt werden, daß $x \in x$ (und damit auch $x \notin x$, was $\neg(x \in x)$ entspricht) nicht wohlgetypt ist, da x sonst sowohl vom Typ α als auch vom Typ α set sein müßte. Das Problem liegt – präziser ausgedrückt – darin, daß die Typen α und α set aufgrund des Vorkommens von α in α set nicht *unifizierbar* sind, was oft mit dem Schlagwort *occurs check* ausgedrückt wird.

2.2.2 Logische Verknüpfungen und Inferenzregeln

In der Logik Isabelle/HOL stehen alle grundlegenden logischen Operatoren wie

$$
\begin{array}{lcl}
\neg & :: & \mathsf{bool} \to \mathsf{bool} \\
\wedge & :: & \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \\
\vee & :: & \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool} \\
\longrightarrow & :: & \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}
\end{array}
\qquad
\begin{array}{lcl}
\forall & :: & (\alpha \to \mathsf{bool}) \to \mathsf{bool} \\
\exists & :: & (\alpha \to \mathsf{bool}) \to \mathsf{bool} \\
\exists_1 & :: & (\alpha \to \mathsf{bool}) \to \mathsf{bool}
\end{array}
$$

zur Verfügung. Hierbei bezeichnet bool den Typ der Wahrheitswerte in HOL. Beweise in HOL werden mit Hilfe des *Kalküls des natürlichen Schließens* geführt, das auf Gerhard Gentzen zurückgeht. Für jeden der oben angegebenen logischen Operatoren existieren hierbei *Einführungs-* und *Eliminationsregeln*. Wie bereits in Abschnitt 2.1 erwähnt, werden Inferenz-

regeln von Objektlogiken mit Hilfe der Metalogik von Isabelle formuliert. Eine Inferenzregel wie

$$\frac{P \quad Q}{P \wedge Q}$$

wird in Isabelle beispielsweise durch

$$[\![P; Q]\!] \Longrightarrow P \wedge Q \qquad \text{(conjI)}$$

ausgedrückt. Die Regel zur Einführung der Implikation lautet

$$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q \qquad \text{(impI)}$$

Die Objekt-Implikation $\longrightarrow$ wird hierbei auf die Meta-Implikation $\Longrightarrow$ zurückgeführt, die zur Darstellung der lokalen Annahmen eines Beweisziels dient. Die Regel

$$\left(\bigwedge x.\ P\ x\right) \Longrightarrow \forall x.\ P\ x \qquad \text{(allI)}$$

für die Einführung des Allquantors führt den Objekt-Allquantor $\forall$ auf den Meta-Allquantor $\bigwedge$ zurück. Spezielle Meta-Inferenzregeln für $\Longrightarrow$ und $\bigwedge$ wie *Lifting über Annahmen* und *Lifting über Parameter*, die in [Paulson, 1989, Paulson, 1993a] genauer beschrieben werden, stellen sicher, daß die lokalen Annahmen eines Beweisziels in alle daraus entstehenden Teilziele übernommen und Nebenbedingungen von Quantorenregeln wie „x nicht frei in ...“ eingehalten werden.

Für die Korrektheit der Quantorenregeln

$$\forall x.\ P\ x \Longrightarrow P\ (y :: \tau) \qquad \text{(spec)}$$

$$P\ (y :: \tau) \Longrightarrow \exists x.\ P\ x \qquad \text{(exI)}$$

ist bedeutsam, daß die Interpretation $[\![\tau]\!]$ des Typs τ stets eine nichtleere Menge ist. So wäre z.B. die mittels (spec) und (exI) ableitbare Formel $\forall x.\ P(x) \Longrightarrow \exists x.\ P(x)$ falsch, falls x einen leeren Typ besitzen würde. Erläuterungen zu dieser Problematik finden sich in [Paulson, 1990].

Bemerkung 2.1
Zur Vereinfachung der Notation werden wir bei der Darstellung von Beweisen in den folgenden Kapiteln nicht mehr zwischen Meta- und Objekt-Implikation bzw. Meta- und Objekt-Allquantor unterscheiden. Wir verwenden $\forall$ für Allquantifikation und $\Longrightarrow$ für Implikation, um Verwechslungen mit Funktionstypen zu vermeiden. Um die Darstellung der Beweise kompakt zu halten, werden außerdem elementare Beweisschritte wie z.B. Implikations- und Allquantor-Einführung meist weggelassen.

In der Form

$$\frac{\begin{array}{c}[P\ x_1\ \ldots\ x_n]_{x_1,\ldots,x_n}\\ \vdots\\ Q\ x_1\ \ldots\ x_n\end{array} \qquad \ldots}{R}$$

notierte Inferenzregeln sind gleichbedeutend mit

$$((\forall x_1, \ldots, x_n.\ P\ x_1\ \ldots\ x_n \Longrightarrow Q\ x_1\ \ldots\ x_n) \wedge \ldots) \Longrightarrow R$$

Hilberts ε - Operator Eine Besonderheit von HOL ist der sogenannte *Hilbertsche Auswahloperator*

$$\varepsilon \ :: \ (\alpha \to \mathsf{bool}) \to \alpha$$

Für ein Prädikat P auf dem Typ α liefert $\varepsilon x.\ P\ x$ ein nicht genauer festgelegtes Element vom Typ α mit der Eigenschaft P, falls ein solches existiert. Hierbei ist zu beachten, daß $\varepsilon x.\ P\ x$ immer ein Ergebnis liefert – auch dann, wenn kein Element mit der Eigenschaft P existiert. Dieses Verhalten von ε kann durch die Gleichung

$$P\ (\varepsilon x.\ P\ x) = \exists x.\ P\ x \qquad \text{(select_eq_Ex)}$$

beschrieben werden. Falls genau ein Element mit der Eigenschaft P existiert, kann die Regel

$$\frac{\exists_1 x.\ P\ x \quad P\ a}{(\varepsilon x.\ P\ x) = a}\ \text{(select1_equality)}$$

verwendet werden, wobei

$$\exists_1 x.\ P\ x \equiv \exists x.\ P\ x \wedge (\forall y. P\ y \Longrightarrow y = x)$$

Die Konstante arbitrary In manchen Fällen kann es hilfreich sein, ein universell verwendbares, nicht näher spezifiziertes „don't care“ - Element zur Verfügung zu haben. Hierzu dient die polymorphe Konstante arbitrary :: α.

2.2.3 Definitorische Theorieerweiterung

Das Ergebnis der Formalisierung eines Sachverhalts in Isabelle/HOL sind sogenannte *Theorien*. Eine Theorie besteht aus einer Menge von Typkonstruktoren und Konstanten, oft auch *Signatur* genannt, sowie aus einer Menge von Axiomen. Bei der Entwicklung neuer Theorien geht man in der Regel von bestehenden Theorien aus, die man um zusätzliche Typkonstruktoren, Konstanten und Axiome erweitert. Eine wichtige Grundphilosophie von HOL ist es, hierbei nicht das Hinzufügen beliebiger Axiome zu erlauben, sondern nur sogenannte *definitorische* Erweiterungsmechanismen zuzulassen. Die Verwendung derartiger Erweiterungsmechanismen garantiert, daß bei der Erweiterung die Existenz von *Modellen* erhalten bleibt, d.h. die entstehende Theorie ein Modell besitzt, falls die ursprüngliche Theorie ein Modell hatte. Unter einem Modell M versteht man hierbei eine Interpretation von Typkonstruktoren und Konstanten, so daß die Interpretation jedes Axioms der Theorie bzgl. M für jede Belegung der freien Variablen stets „wahr“ ist. Für eine formale Definition des Modellbegriffs sei wieder auf [Gordon, Melham, 1993] verwiesen.

Die Existenz eines Modells impliziert die sogenannte *syntaktische Konsistenz* einer Theorie. Eine Theorie heißt hierbei syntaktisch konsistent, wenn in ihr nicht jede Formel – insbesondere nicht False – ableitbar ist. Man beachte, daß die syntaktische Konsistenz einer Theorie jedoch nicht unbedingt die Existenz eines Modells impliziert.

Im wesentlichen werden von HOL folgende Arten der definitorischen Theorieerweiterung unterstützt:

Konstantendefinition Die Konstante c kann zur Signatur hinzugefügt und ein Axiom der Gestalt $c\ v_1\ \ldots\ v_n \equiv t$ eingeführt werden, falls $\mathrm{TVars}(t) \subseteq \mathrm{TVars}(c)$ sowie $\mathrm{Vars}(t) \subseteq \{v_1, \ldots, v_n\}$ gilt, c neu ist und nicht in t vorkommt.

Typdefinition Sei $t_\mathsf{rep_set}$ ein Term vom Typ τ set, der eine nichtleere Menge beschreibt, d.h. es gilt $u \in t_\mathsf{rep_set}$ für geeignetes u. Weiterhin sei $\mathrm{TVars}(t_\mathsf{rep_set}) \subseteq \{\alpha_1, \ldots, \alpha_n\}$. Dann kann der Typ $(\alpha_1, \ldots, \alpha_n)t$ und die Konstanten

$$\begin{aligned} \mathsf{Abs_}t &:: \tau \to (\alpha_1, \ldots, \alpha_n)t \\ \mathsf{Rep_}t &:: (\alpha_1, \ldots, \alpha_n)t \to \tau \end{aligned}$$

zur Signatur hinzugefügt sowie die Axiome

$$\begin{array}{ll} \mathsf{Abs_}t(\mathsf{Rep_}t(x)) = x & (\mathrm{Rep_}t\mathrm{_inverse}) \\ y \in t_\mathsf{rep_set} \Longrightarrow \mathsf{Rep_}t(\mathsf{Abs_}t(y)) = y & (\mathrm{Abs_}t\mathrm{_inverse}) \\ \mathsf{Rep_}t(x) \in t_\mathsf{rep_set} & (\mathrm{Rep_t}) \end{array}$$

eingeführt werden.

Die Grundidee der Typdefinition besteht hierbei darin, neue Typen durch Teilmengen bereits existierender Typen zu repräsentieren. Die Axiome (Rep_t_inverse), (Abs_t_inverse) und (Rep_t) drücken hierbei die Existenz einer Bijektion zwischen der Menge $\tau_\mathsf{rep_set}$ und den Elementen des neuen Typs $(\alpha_1, \ldots, \alpha_n)t$ aus, was oft auch als *Isomorphie* bezeichnet wird. Dieser Sachverhalt ist in Abbildung 2.2 graphisch dargestellt. Die Beweisbarkeit von $u \in t_\mathsf{rep_set}$ ist notwendig, da Typen in HOL, wie bereits in Abschnitt 2.2.1 erwähnt, nur durch nichtleere Mengen repräsentiert werden dürfen.

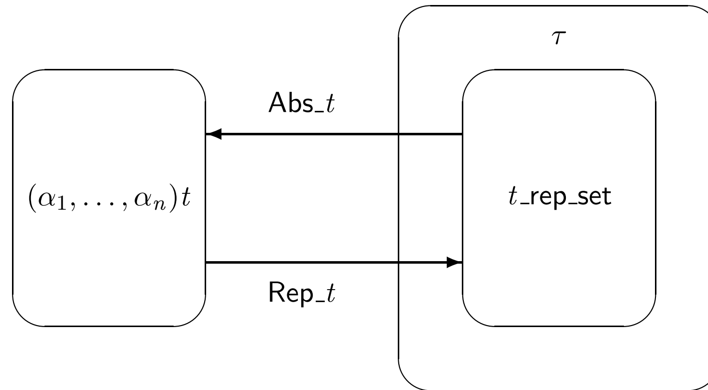


Abbildung 2.2: Definition eines neuen Typs

Da $\mathsf{Rep_}t$ die Umkehrfunktion $\mathsf{Abs_}t$ und $\mathsf{Abs_}t$ auf der Menge t_rep_set die Umkehrfunktion $\mathsf{Rep_}t$ besitzt, folgt sofort die Injektivität von $\mathsf{Rep_}t$ sowie die Injektivität von $\mathsf{Abs_}t$ auf der Menge t_rep_set, d.h.

$$\begin{aligned} \mathsf{Rep_}t(x_1) = \mathsf{Rep_}t(x_2) &\Longrightarrow x_1 = x_2 \\ y_1 \in t_\mathsf{rep_set} \wedge y_2 \in t_\mathsf{rep_set} \wedge \mathsf{Abs_}t(y_1) = \mathsf{Abs_}t(y_2) &\Longrightarrow y_1 = y_2 \end{aligned}$$

Die repräsentierende Menge muß hierbei so geschickt gewählt werden, daß sich die für den neuen Typ gewünschten Eigenschaften mit Hilfe der Isomorphie aus den entsprechenden Eigenschaften der repräsentierenden Menge herleiten lassen. Elementare Funktionen auf dem neuen Typ werden hierbei mittels $\mathsf{Abs_}t$ und $\mathsf{Rep_}t$ ausgedrückt. Sind die charakteristischen Eigenschaften einmal bewiesen, kann von der konkreten Repräsentation des Typs völlig abstrahiert werden. Im Gegensatz zur axiomatischen Vorgehensweise, die immer die Gefahr von Inkonsistenzen in sich birgt, können mit diesem Verfahren abstrakte Datentypen sicher eingeführt werden.

Ein formaler Beweis der Tatsache, daß die Existenz von Modellen bei Verwendung der obigen Definitionsmechanismen tatsächlich erhalten bleibt, ist wiederum in [Gordon, Melham, 1993] zu finden.

Mit Hilfe des beschriebenen Verfahrens kann so beispielsweise der Typ der *disjunkten Summe* $\alpha + \beta$ beschrieben werden. Als repräsentierende Menge wird hierbei

$$\begin{aligned}
\mathsf{sum} &:: (\alpha \to \beta \to \mathsf{bool} \to \mathsf{bool})\mathsf{set} \\
\mathsf{sum} &\equiv \{f \mid (\exists a.\ f = \lambda\ x, y, p.\ x = a \wedge p) \vee (\exists b.\ f = \lambda\ x, y, p.\ y = b \wedge \neg p)\}
\end{aligned}$$

verwendet. Die beiden Injektionen $\mathsf{Inl} :: \alpha \to \alpha + \beta$ und $\mathsf{Inr} :: \beta \to \alpha + \beta$ sind definiert durch

$$\begin{aligned}
\mathsf{Inl}\ a &\equiv \mathsf{Abs_sum}(\lambda\ x, y, p.\ x = a \wedge p) \\
\mathsf{Inr}\ b &\equiv \mathsf{Abs_sum}(\lambda\ x, y, p.\ y = b \wedge \neg p)
\end{aligned}$$

An diesem Beispiel wird deutlich, daß selbst die Konstruktion einfacher Typen relativ umständlich und mitunter auch undurchsichtig sein kann. Es ist daher erforderlich, für bestimmte Klassen häufig vorkommender Typen Verfahren für deren systematische Konstruktion zu entwickeln.

2.3 Theoretische Grundlagen

Der folgende Abschnitt behandelt einige theoretische Grundlagen wie Fixpunkte und induktive Definitionen, die für die Konstruktion von Datentypen wichtig sind. Die beschriebenen abstrakten Konzepte sind in Isabelle/HOL in Form von Theorien bzw. generischen Beweisprozeduren implementiert. Die folgenden Beweise werden daher in Anlehnung an Isabelle/HOL oft in einem rückwärtsgerichteten Stil und unter Verwendung von für HOL typischer Notation präsentiert, wobei von konkreten Implementierungsdetails jedoch weitgehend abstrahiert wird.

2.3.1 Fixpunkte

Im folgenden wollen wir kurz auf einige wichtige Ergebnisse aus der Fixpunkttheorie eingehen, die für die präzise mathematische Beschreibung von Datentypen und rekursiven Funktionen sowie induktiven Definitionen von zentraler Bedeutung sind.

Definition 2.2 (Untere Schranke)
Sei $\sqsubseteq \subseteq S \times S$ eine Ordnungsrelation auf S und $T \subseteq S$. Dann heißt $b \in S$ *untere Schranke* von T, falls $b \sqsubseteq x$ für alle $x \in T$ gilt.

g heißt *größte untere Schranke* (Infimum) von T, falls $b \sqsubseteq g$ für alle unteren Schranken b von T gilt. Falls eine solche stets eindeutige größte untere Schranke von T existiert, wird diese auch mit $\sqcap T$ bezeichnet. □

Definition 2.3 (Vollständiger Verband)
Eine Menge S zusammen mit einer Ordnungsrelation $\sqsubseteq \subseteq S \times S$ heißt *vollständiger Verband* $(S, \sqsubseteq)$, falls jede Menge $T \subseteq S$ eine größte untere Schranke bzgl. $\sqsubseteq$ besitzt. □

Definition 2.4 (Monotonie von Funktionen)
Eine Funktion $F : S \to S$ heißt *monoton*, falls für alle x, y gilt:

$$x \sqsubseteq y \Longrightarrow F(x) \sqsubseteq F(y)$$

□

Theorem 2.5 (Knaster-Tarski)
Sei $(S, \sqsubseteq)$ ein vollständiger Verband und $F : S \to S$ eine monotone Funktion. Dann existiert ein durch

$$\mathsf{lfp}(F) = \bigsqcap \{x \mid F(x) \sqsubseteq x\}$$

eindeutig bestimmter *kleinster Fixpunkt* von F, d.h. es gilt

$$F(\mathsf{lfp}(F)) = \mathsf{lfp}(F) \qquad \text{und} \qquad F(x) \sqsubseteq x \Longrightarrow \mathsf{lfp}(F) \sqsubseteq x$$

Beweis Bezeichne

$$P = \{x \mid F(x) \sqsubseteq x\}$$

die Menge der sogenannten *Präfixpunkte* von F.

Für jedes $x \in P$ gilt $\mathsf{lfp}(F) \sqsubseteq x$, da $\mathsf{lfp}(F)$ eine untere Schranke der Menge P ist. Aufgrund der Monotonie von F gilt daher auch $F(\mathsf{lfp}(F)) \sqsubseteq F(x)$. Wegen $F(x) \sqsubseteq x$ folgt hieraus $F(\mathsf{lfp}(F)) \sqsubseteq x$, d.h. $F(\mathsf{lfp}(F))$ ist ebenfalls eine untere Schranke von P. Da $\mathsf{lfp}(F)$ die größte untere Schranke von P ist, gilt daher $F(\mathsf{lfp}(F)) \sqsubseteq \mathsf{lfp}(F)$, d.h. $\mathsf{lfp}(F)$ ist ein Präfixpunkt von F. Aufgrund seiner Definition ist $\mathsf{lfp}(F)$ offensichtlich auch der kleinste Präfixpunkt von F.

Aus $F(\mathsf{lfp}(F)) \sqsubseteq \mathsf{lfp}(F)$ folgt $F(F(\mathsf{lfp}(F))) \sqsubseteq F(\mathsf{lfp}(F))$ aufgrund der Monotonie von F. Es gilt also auch $F(\mathsf{lfp}(F)) \in P$ und damit $\mathsf{lfp}(F) \sqsubseteq F(\mathsf{lfp}(F))$.

Damit gilt $F(\mathsf{lfp}(F)) = \mathsf{lfp}(F)$. □

In den folgenden Abschnitten werden hauptsächlich vollständige Verbände der Form $(\mathcal{P}(S), \subseteq)$ von Bedeutung sein. Hierbei bezeichnet $\mathcal{P}(S) = \{x \mid x \subseteq S\}$ die *Potenzmenge* von S, weshalb

$(\mathcal{P}(S), \subseteq)$ auch *Potenzmengenverband* genannt wird. Die größte untere Schranke einer Menge $T \subseteq \mathcal{P}(S)$ ist hierbei durch $\bigcap T$ bestimmt.

Eine Menge $M \subseteq S$ mit der Eigenschaft $F(M) \subseteq M$ wird auch als *unter F abgeschlossen* bezeichnet. $\mathsf{lfp}(F)$ stellt also die kleinste unter F abgeschlossene Menge dar.

Die Begriffe *kleinste obere Schranke* und *größter Fixpunkt* werden dual dazu definiert. So gilt

$$\mathsf{gfp}(F) = \bigsqcup \{x \mid x \sqsubseteq F(x)\}$$

Man beachte, daß in einem vollständigen Verband jede beliebige Menge auch eine eindeutige kleinste obere Schranke besitzt, nämlich

$$\bigsqcup T = \bigsqcap \{x \mid \forall y \in T.\ y \sqsubseteq x\}$$

In vielen Arbeiten über Datentypen wie z.B. [Hensel, Jacobs, 1997], die nicht Mengenlehre, sondern Kategorientheorie als Grundlage verwenden, findet man oft die Begriffe *initiale Algebren* bzw. *terminale Coalgebren*. Diese stellen das kategorientheoretische Analogon zum Konzept des kleinsten bzw. größten Fixpunkts dar.

2.3.2 Induktive Definitionen

Unter einer *induktiv definierten Menge* versteht man die kleinste Menge, die unter bestimmten Regeln abgeschlossen ist. So kann z.B. die Menge *even* der positiven geraden Zahlen durch folgende zwei Regeln charakterisiert werden:

$$\frac{}{0 \in \mathit{even}} \qquad \frac{n \in \mathit{even}}{n+2 \in \mathit{even}}$$

Derartige Regeln bezeichnet man auch als *Einführungsregeln*. Die erste Regel besagt, daß 0 eine gerade Zahl ist. Die zweite Regel besagt, daß $n + 2$ ebenfalls eine gerade Zahl ist, falls n eine gerade Zahl ist. Man beachte, daß die Menge der positiven geraden Zahlen nicht die einzige unter diesen Regeln abgeschlossene Menge ist. So ist z.B. auch die Menge $\mathbb{N}$ der natürlichen Zahlen und die Menge $\mathbb{R}$ der reellen Zahlen unter diesen Regeln abgeschlossen, denn 0 ist auch eine reelle Zahl und $n + 2$ ist auch eine reelle Zahl, falls n eine reelle Zahl ist. Jedoch stellt die Menge der positiven geraden Zahlen die kleinste unter diesen Regeln abgeschlossene Menge dar.

Wie in Abschnitt 2.3.1 angedeutet, ist die kleinste unter einer monotonen Funktion F abgeschlossene Menge genau der kleinste Fixpunkt von F. Die Menge der positiven geraden Zahlen läßt sich daher wie folgt als Fixpunkt einer geeigneten Funktion F, die sich direkt aus den Einführungsregeln ergibt, definieren:

$$\begin{array}{rcl} F(M) & = & \{x \mid x = 0 \lor \exists n.\ x = n+2 \land n \in M\} \\ \mathit{even} & = & \mathsf{lfp}(F) \\ & = & \bigcap \{x \mid F(x) \subseteq x\} \end{array}$$

Aus dieser Definition lassen sich nun mit Hilfe der Fixpunkteigenschaft die obigen Einführungsregeln ableiten, denn es gilt offensichtlich

$$0 \in \{x \mid x = 0 \lor \exists n.\ x = n+2 \land n \in \mathsf{lfp}(F)\} = F(\mathsf{lfp}(F)) = \mathsf{lfp}(F) = \mathit{even}$$

Ist $n \in \mathit{even} = \mathsf{lfp}(F)$ so gilt

$$n + 2 \in \{x \mid x = 0 \vee \exists n.\ x = n + 2 \wedge n \in \mathsf{lfp}(F)\} = \mathit{even}$$

Aus der Tatsache, daß $\mathsf{lfp}(F)$ die kleinste unter F abgeschlossene Menge ist, läßt sich unmittelbar die Induktionsregel

$$\frac{\mathsf{mono}(F) \quad F(\{x \mid P(x)\}) \subseteq \{x \mid P(x)\}}{\mathsf{lfp}(F) \subseteq \{x \mid P(x)\}} \ \text{(weakInd)}$$

ableiten. Diese Regel läßt sich noch zu

$$\frac{\mathsf{mono}(F) \quad F(\mathsf{lfp}(F) \cap \{x \mid P(x)\}) \subseteq \{x \mid P(x)\}}{\mathsf{lfp}(F) \subseteq \{x \mid P(x)\}} \ \text{(strongInd)}$$

verstärken.

Beweis Sei F monoton und es gelte $F(\mathsf{lfp}(F) \cap \{x \mid P(x)\}) \subseteq \{x \mid P(x)\}$. Offensichtlich ist

$$\mathsf{lfp}(F) \cap \{x \mid P(x)\} \subseteq \mathsf{lfp}(F)$$

woraus aufgrund der Monotonie von F sowie der Fixpunkteigenschaft

$$F(\mathsf{lfp}(F) \cap \{x \mid P(x)\}) \subseteq F(\mathsf{lfp}(F)) = \mathsf{lfp}(F)$$

folgt. Damit gilt

$$F(\mathsf{lfp}(F) \cap \{x \mid P(x)\}) \subseteq \mathsf{lfp}(F) \cap \{x \mid P(x)\}$$

Mit Hilfe von (weakInd) folgt hieraus

$$\mathsf{lfp}(F) \subseteq \mathsf{lfp}(F) \cap \{x \mid P(x)\} \subseteq \{x \mid P(x)\}$$

□

Aus dieser abstrakten Induktionsregel kann nun für jede induktiv definierte Menge eine spezielle Induktionsregel abgeleitet werden. So erhält man z.B. für die Menge *even* die Induktionsregel

$$\frac{x \in \mathit{even} \quad P(0) \quad \forall n.\ P(n) \wedge n \in \mathit{even} \Longrightarrow P(n+2)}{P(x)}$$

Beweis Im folgenden sei wieder

$$\begin{aligned} F(M) &= \{x \mid x = 0 \vee \exists n.\ x = n + 2 \wedge n \in M\} \\ \mathit{even} &= \mathsf{lfp}(F) \end{aligned}$$

und es gelte $P(0)$, $\forall n.\ P(n) \wedge n \in \mathit{even} \Longrightarrow P(n+2)$ sowie $x \in \mathit{even}$. Wir zeigen nun

$$\underbrace{\{x \mid x = 0 \vee \exists n.\ x = n+2 \wedge n \in \mathit{even} \cap \{x \mid P(x)\}\}}_{F(\mathit{even} \cap \{x \mid P(x)\})} \subseteq \{x \mid P(x)\}$$

Sei $z \in \{x \mid x = 0 \vee \exists n.\ x = n+2 \wedge n \in \mathit{even} \cap \{x \mid P(x)\}\}$. Es sind nun zwei Fälle zu unterscheiden:

1. $z = 0$
   Gemäß Voraussetzung gilt $P(0)$ und damit $z \in \{x \mid P(x)\}$.

2. $z = n+2 \wedge n \in \mathit{even} \cap \{x \mid P(x)\}$
   Es gilt $P(n) \wedge n \in \mathit{even}$, woraus nach Voraussetzung $P(n+2)$ und damit $z \in \{x \mid P(x)\}$ folgt.

Mittels (strongInd) folgt also $\mathit{even} \subseteq \{x \mid P(x)\}$. Nach Voraussetzung ist $x \in \mathit{even}$, weshalb $P(x)$ gilt.

□

Aus der Tatsache, daß *even* ein Fixpunkt ist, läßt sich außerdem noch die folgende *Eliminationsregel*

$$\frac{x \in \mathrm{even} \quad x = 0 \Longrightarrow P \quad \forall n.\ x = n+2 \wedge n \in \mathit{even} \Longrightarrow P}{P}$$

ableiten, die eine Fallunterscheidung über alle Entstehungsmöglichkeiten von Elementen aus *even* darstellt.

Beweis Angenommen, es gelte $x \in \mathrm{even}$, $x = 0 \Longrightarrow P$ und $\forall n.\ x = n+2 \wedge n \in \mathit{even} \Longrightarrow P$. Aus $x \in \mathrm{even}$ erhält man aufgrund der Fixpunkteigenschaft von *even*

$$x \in \{x \mid x = 0 \vee \exists n.\ x = n+2 \wedge n \in \mathit{even}\}$$

Analog zum Beweis der Induktionsregel sind die zwei Fälle

1. $x = 0$ und

2. $x = n+2 \wedge n \in \mathit{even}$

zu unterscheiden, aus denen gemäß Voraussetzung sofort P folgt. □

Die soeben am Beispiel der Menge *even* erarbeiteten Konzepte sollen nun im folgenden verallgemeinert werden. Hierbei wird insbesondere auch die simultane induktive Definition mehrerer Mengen erlaubt.

Definition 2.6 (Injektionen, Fallunterscheidung)
Seien die Typen $\tau_1, \ldots, \tau_n$ gegeben. Dann bezeichne

$$\mathsf{in}_i \ :: \ \tau_i \to \tau_1 + \cdots + \tau_n$$

eine Funktion mit den Eigenschaften

$$i \neq j \Longrightarrow \mathsf{in}_i(x) \neq \mathsf{in}_j(y)$$
$$x \neq y \Longrightarrow \mathsf{in}_i(x) \neq \mathsf{in}_i(y)$$

Weiterhin sei

$$\mathsf{sum_case} \ :: \ (\tau_1 \to \tau) \to \cdots \to (\tau_n \to \tau) \to \tau_1 + \cdots + \tau_n \to \tau$$

eine Funktion mit der Eigenschaft

$$\mathsf{sum_case}\, f_1 \ \ldots \ f_n \ (\mathsf{in}_i(x)) = f_i(x)$$

zur Formulierung von Fallunterscheidungen auf dem Typ $\tau_1 + \cdots + \tau_n$. □

Definition 2.7 (Inverses Bild)
Das *inverse Bild* vimage f S der Menge S bzgl. einer Funktion f sei definiert durch

$$\mathsf{vimage}\ f\ S = \{x \mid f(x) \in S\}$$

□

Definition 2.8 (Induktiv definierte Mengen)
Seien S_1, …, S_n Bezeichner für Mengen über den Typen τ_1, …, τ_n. Ferner sei R eine Menge von Einführungsregeln der Gestalt

$$\frac{\ldots \quad Q_j \quad \ldots \quad t_k \in F_l(S_{i'}) \quad \ldots}{t \in S_i}$$

wobei Q_j eine beliebige logische Formel ist, die keine der Bezeichner S_i enthält. t und t_k seien beliebige Terme und F_l eine monotone Funktion.

Dann sind die Mengen S_1, …, S_n durch R *induktiv definiert*, falls gilt

$$\begin{aligned}
F(U) &= \{x \mid \ldots \vee \\
&\qquad \exists v_1, \ldots, v_m.\ x = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in F_l(\mathsf{vimage}\ \mathsf{in}_{i'}\ U) \wedge \ldots \vee \\
&\qquad \ldots\} \\
T &= \mathsf{lfp}(F) \\
S_1 &= \mathsf{vimage}\ \mathsf{in}_1\ T \\
&\vdots \\
S_n &= \mathsf{vimage}\ \mathsf{in}_n\ T
\end{aligned}$$

□

Hierbei entspricht jedes Glied der Disjunktion innerhalb von $\{x \mid \ldots\}$ einer Einführungsregel. Die $v_1, \ldots, v_m$ sind die in der entsprechenden Einführungsregel frei vorkommenden Variablen. Die Menge T enthält sämtliche Elemente aller Mengen $S_1, \ldots, S_n$, „verpackt“ in die entsprechenden Injektionen $\mathsf{in}_1, \ldots, \mathsf{in}_n$. Die den einzelnen Mengen zugehörigen Elemente werden dann mittels $\mathsf{vimage}\ \mathsf{in}_1\ T, \ldots, \mathsf{vimage}\ \mathsf{in}_n\ T$ wieder aus T „ausgepackt“. Um die Monotonie von F sicherzustellen, müssen Einführungsregeln stets die oben beschriebene Gestalt besitzen.

Theorem 2.9 (Charakteristische Eigenschaften induktiver Mengen)
Für induktiv definierte Mengen $S_1, \ldots, S_n$ lassen sich folgenden Regeln ableiten:

- Einführungsregeln der Form

$$\frac{\ldots \quad Q_j \quad \ldots \quad t_k \in F_l(S_{i'}) \quad \ldots}{t \in S_i}$$

- Eliminationsregeln für jede der Mengen $S_1, \ldots, S_n$ der Form

$$\frac{x \in S_i \quad \ldots \quad \begin{array}{c}[x = t \quad \ldots \quad Q_i \quad \ldots \quad t_k \in F_l(S_{i'}) \quad \ldots]_{v_1,\ldots,v_m} \\ \vdots \\ P \end{array} \quad \ldots}{P}$$

- Eine Induktionsregel für alle Mengen $S_1, \ldots, S_n$ der Form

$$\frac{\ldots \quad \begin{array}{c}[\ldots \quad Q_i \quad \ldots \quad t_k \in F_l(S_{i'} \cap \{x \mid P_{i'}(x)\}) \quad \ldots]_{v_1,\ldots,v_m} \\ \vdots \\ P_i(t) \end{array} \quad \ldots}{x_1 \in S_1 \Longrightarrow P_1(x_1) \wedge \ldots \wedge x_n \in S_n \Longrightarrow P_n(x_n)}$$

  bzw.

$$\frac{\ldots \quad \begin{array}{c}[\ldots \quad Q_i \quad \ldots \quad t_k \in S_{i'} \quad P_{i'}(t_k) \quad \ldots]_{v_1,\ldots,v_m} \\ \vdots \\ P_i(t) \end{array} \quad \ldots}{x_1 \in S_1 \Longrightarrow P_1(x_1) \wedge \ldots \wedge x_n \in S_n \Longrightarrow P_n(x_n)}$$

  falls $F_l = \mathsf{Id}$

Beweis

Monotonie Unter der Voraussetzung $S \subseteq T$ zeigen wir

$$\forall x.\ x \in F(S) \Longrightarrow x \in F(T)$$

Für alle beteiligten Operatoren werden hierbei Monotonieregeln wie z.B.

$$\frac{P \Longrightarrow Q \quad P' \Longrightarrow Q'}{P \wedge P' \Longrightarrow Q \wedge Q'} \qquad \frac{P \Longrightarrow Q \quad P' \Longrightarrow Q'}{P \vee P' \Longrightarrow Q \vee Q'}$$

$$\frac{\forall x.\ P(x) \Longrightarrow Q(x)}{(\exists x.\ P(x)) \Longrightarrow (\exists x.\ Q(x))} \qquad \frac{S \subseteq T}{x \in \mathsf{vimage}\ f\ S \Longrightarrow x \in \mathsf{vimage}\ f\ T}$$

verwendet, durch deren sukzessive Anwendung das obige Beweisziel auf die Beweisziele $S \subseteq S$, $T \subseteq T$ und $S \subseteq T$ reduziert werden kann.

Einführungsregeln Unter den Voraussetzungen Q_j, $t_k \in F_l(S_{i'})$, ... muß $t \in S_i$ gezeigt werden. Es gilt

$$
\begin{array}{ll}
 & t \in S_i \\
\iff & \quad \{\text{Definition von } S_i\} \\
 & t \in \mathsf{vimage}\ \mathsf{in}_i\ T \\
\iff & \quad \{\text{Definition von } \mathsf{vimage}\} \\
 & \mathsf{in}_i(t) \in T \\
\iff & \quad \{\text{Fixpunkteigenschaft von } T\text{, Monotonie von } F\} \\
 & \mathsf{in}_i(t) \in \{x \mid \exists \ldots .\ x = \mathsf{in}_1(\ldots) \wedge \ldots \vee \\
 & \qquad \ldots \vee \\
 & \qquad \exists v_1, \ldots, v_m.\ x = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in F_l(\mathsf{vimage}\ \mathsf{in}_{i'}\ T) \wedge \ldots \vee \\
 & \qquad \ldots \vee \\
 & \qquad \exists \ldots .\ x = \mathsf{in}_n(\ldots) \wedge \ldots\}
\end{array}
$$

Man erhält somit das neue Beweisziel

$$
\begin{array}{l}
\exists \ldots .\ \mathsf{in}_i(t) = \mathsf{in}_1(\ldots) \wedge \ldots \vee \\
\ldots \vee \\
\exists v_1, \ldots, v_m.\ \mathsf{in}_i(t) = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in F_l(\mathsf{vimage}\ \mathsf{in}_{i'}\ T) \wedge \ldots \vee \\
\ldots \vee \\
\exists \ldots .\ \mathsf{in}_i(t) = \mathsf{in}_n(\ldots) \wedge \ldots
\end{array}
$$

Durch Auswahl des geeigneten Zweigs der Disjunktion erhält man hieraus das Beweisziel

$$\exists v_1, \ldots, v_m.\ \mathsf{in}_i(t) = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in F_l(\mathsf{vimage}\ \mathsf{in}_{i'}\ T) \wedge \ldots$$

was sich mit Hilfe von Reflexivität und der Voraussetzungen lösen läßt.

Eliminationsregeln Unter den Voraussetzungen

$$
\begin{array}{l}
x \in S_i \\
\forall v_1, \ldots, v_m.\ x = t \wedge Q_i \wedge t_k \in F_l(S_{i'}) \wedge \ldots \Longrightarrow P \\
\qquad \vdots
\end{array}
$$

muß nun P gezeigt werden. Die Vorgehensweise ist ähnlich wie beim Beweis der Einführungsregeln. Aus der Voraussetzung $x \in S_i$ erhält man nach Definition $x \in \mathsf{vimage}\ \mathsf{in}_i\ T$, was gleichbedeutend ist mit $\mathsf{in}_i(x) \in T$. Unter Ausnutzung der Fixpunkteigenschaft von T ergibt sich hieraus

$$
\begin{array}{l}
\mathsf{in}_i(x) \in \{x \mid \exists \ldots .\ x = \mathsf{in}_1(\ldots) \wedge \ldots \vee \\
\qquad \ldots \vee \\
\qquad \exists v_1, \ldots, v_m.\ x = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in F_l(\mathsf{vimage}\ \mathsf{in}_{i'}\ T) \wedge \ldots \vee \\
\qquad \ldots \vee \\
\qquad \exists \ldots .\ x = \mathsf{in}_n(\ldots) \wedge \ldots\}
\end{array}
$$

Nach Aufspaltung der Disjunktion und Berücksichtigung von $\mathsf{vimage}\ \mathsf{in}_{i'}\ T = S_{i'}$ erhält man somit die Beweisziele

$$\begin{array}{ll} & \forall \ldots.\ \mathsf{in}_i(x) = \mathsf{in}_1(\ldots) \wedge \ldots \Longrightarrow P \\ & \quad\vdots \\ (\star) & \forall v_1, \ldots, v_m.\ \mathsf{in}_i(x) = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in F_l(S_{i'}) \wedge \ldots \Longrightarrow P \\ & \quad\vdots \\ & \forall \ldots.\ \mathsf{in}_i(x) = \mathsf{in}_n(\ldots) \wedge \ldots \Longrightarrow P \end{array}$$

Die Beweisziele mit Prämissen der Gestalt $\mathsf{in}_i(x) = \mathsf{in}_{i''}(\ldots)$, wobei $i \neq i''$, sind wegen $\mathsf{in}_i(x) \neq \mathsf{in}_{i''}(\ldots)$ trivial. Die verbleibenden Beweisziele mit Prämissen der Gestalt $\mathsf{in}_i(x) = \mathsf{in}_i(\ldots)$ lassen sich unter Verwendung der Injektivität von in_i umformen. Aus $(\star)$ erhält man beispielsweise

$$\forall v_1, \ldots, v_m.\ x = t \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in F_l(S_{i'}) \wedge \ldots \Longrightarrow P$$

was sich unter Verwendung der Voraussetzungen lösen läßt.

Induktionsregel Zum Beweis der Regel zeigen wir zunächst das Lemma

$$\frac{\forall x.\ x \in T \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x}{x_1 \in S_1 \Longrightarrow P_1(x_1) \wedge \ldots \wedge x_n \in S_n \Longrightarrow P_n(x_n)}$$

Aus $x_i \in S_i$ folgt nach Definition $x_i \in \mathsf{vimage}\ \mathsf{in}_i\ T$. Aufgrund der Eigenschaften von vimage folgt hieraus $\mathsf{in}_i(x_i) \in T$. Nach Voraussetzung gilt daher $\mathsf{sum_case}\ P_1\ \ldots\ P_n\ (\mathsf{in}_i(x_i))$, was gleichbedeutend mit $P_i(x_i)$ ist.

Dieses Lemma ist notwendig, da die zu beweisende Induktionsregel n verschiedene Prädikate enthält, während die im folgenden Beweis verwendete abstrakte Induktionsregel (strongInd) nur für ein einzelnes Prädikat formuliert ist. Wir zeigen nun die Regel

$$\frac{\ldots \qquad \forall v_1, \ldots, v_m.\ \ldots \wedge Q_i \wedge \ldots \wedge t_k \in S_{i'} \wedge P_{i'}(t_k) \wedge \ldots \Longrightarrow P_i(t) \qquad \ldots}{\forall x.\ x \in T \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x}$$

indem wir unter den Voraussetzungen

$$\begin{array}{l} \forall v_1, \ldots, v_m.\ \ldots \wedge Q_i \wedge \ldots \wedge t_k \in S_{i'} \wedge P_{i'}(t_k) \wedge \ldots \Longrightarrow P_i(t) \\ \quad\vdots \end{array}$$

das Beweisziel

$$\forall x.\ x \in T \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x$$

zeigen. Mit Hilfe von (strongInd) und der Monotonie von F erhält man hieraus das neue Beweisziel

$$\begin{aligned}
&\forall x.\ x \in \{x \mid \exists \ldots .\ x = \mathsf{in}_1(\ldots) \wedge \ldots \vee \\
&\quad \ldots \vee \\
&\quad \exists v_1, \ldots, v_m.\ x = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge \\
&\qquad t_k \in (\mathsf{vimage\ in}_{i'}\ (T \cap \{x \mid \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x\})) \wedge \ldots \vee \\
&\quad \ldots \vee \\
&\quad \exists \ldots .\ x = \mathsf{in}_n(\ldots) \wedge \ldots\} \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x
\end{aligned}$$

Wegen

$$\begin{aligned}
& \mathsf{vimage\ in}_{i'}\ (T \cap \{x \mid \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x\}) \\
= \quad & \{\text{Distributivität von } \mathsf{vimage} \text{ über } \cap\} \\
& (\mathsf{vimage\ in}_{i'}\ T) \cap (\mathsf{vimage\ in}_{i'}\ \{x \mid \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x\}) \\
= \quad & \{\text{Vereinfachungsregel für } \mathsf{vimage} \text{ und Mengenkomprehension}\} \\
& (\mathsf{vimage\ in}_{i'}\ T) \cap \{x \mid P_{i'}(x)\} \\
= \quad & \{\text{Definition von } S_{i'}\} \\
& S_{i'} \cap \{x \mid P_{i'}(x)\}
\end{aligned}$$

erhält man hieraus das Beweisziel

$$\begin{aligned}
&\forall x.\ x \in \{x \mid \exists \ldots .\ x = \mathsf{in}_1(\ldots) \wedge \ldots \vee \\
&\quad \ldots \vee \\
&\quad \exists v_1, \ldots, v_m.\ x = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge t_k \in (S_{i'} \cap \{x \mid P_{i'}(x)\}) \wedge \ldots \vee \\
&\quad \ldots \vee \\
&\quad \exists \ldots .\ x = \mathsf{in}_n(\ldots) \wedge \ldots\} \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x
\end{aligned}$$

Nach Aufspaltung der Disjunktion erhält man $|R|$ verschiedene Beweisziele:

$$\begin{aligned}
& \forall x, \ldots .\ x = \mathsf{in}_1(\ldots) \wedge \ldots \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x \\
& \qquad \vdots \\
(\star\star) \quad & \forall x, v_1, \ldots, v_m.\ x = \mathsf{in}_i(t) \wedge \ldots \wedge Q_j \wedge \ldots \wedge \\
& \qquad t_k \in S_{i'} \wedge P_{i'}(t_k) \wedge \ldots \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x \\
& \qquad \vdots \\
& \forall x, \ldots .\ x = \mathsf{in}_n(\ldots) \wedge \ldots \Longrightarrow \mathsf{sum_case}\ P_1\ \ldots\ P_n\ x
\end{aligned}$$

Aus $(\star\star)$ ergibt sich unter Verwendung der Prämisse $x = \mathsf{in}_i(t)$ das neue Beweisziel

$$\forall x, v_1, \ldots, v_m.\ Q_j \wedge \ldots \wedge t_k \in S_{i'} \wedge P_{i'}(t_k) \wedge \ldots \Longrightarrow P_i(t)$$

das sich mit Hilfe der Voraussetzungen lösen läßt. Kombiniert man die soeben bewiesene Regel mit dem obigen Lemma, so erhält man die gewünschte Induktionsregel. □

Viele der hier vorgestellten Beweisprinzipien sowie weitere Hinweise zur Implementierung eines Pakets für induktive Definitionen in Isabelle sind in [Paulson, 1994] zu finden. Dieses

Papier weicht allerdings in manchen Punkten, insbesondere bei der Behandlung verschränkt rekursiver Mengendefinitionen, etwas von der hier gegebenen Darstellung ab. Da dies in Isabelle/HOL bisher nur unvollständig implementiert war, wurde das existierende Paket für induktive Definitionen im Rahmen dieser Arbeit dahingehend verbessert.

Kapitel 3

Arten und Eigenschaften von Datentypen

Wir beschreiben im folgenden die Klassen von Datentypen, die mit Hilfe des im Rahmen dieser Dipomarbeit implementierten Datentyppakets definiert werden können. Im einzelnen sind dies *verschränkt* und *verschachtelt rekursive* Datentypen, für die wir jeweils charakteristische Theoreme angeben.

3.1 Grundlagen

Viele funktionale Programmiersprachen wie z.B. Haskell oder ML bieten die Möglichkeit, rekursive Datentypen zu definieren. Ein Beispiel hierfür ist der Datentyp der Listen

$$\mathsf{datatype}\ \alpha\ \mathsf{list}\ =\ \mathsf{Nil}\ \mid\ \mathsf{Cons}\ \alpha\ (\alpha\ \mathsf{list})$$

Durch diese Definition werden die Konstanten $\mathsf{Nil} :: \alpha\ \mathsf{list}$ und $\mathsf{Cons} :: \alpha \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$ eingeführt, die als *Konstruktoren* des Datentyps bezeichnet werden. Vom Standpunkt der algebraischen Spezifikation funktionaler Programme aus betrachtet sind folgende charakteristische Eigenschaften derartiger Datentypen von Bedeutung:

- Die Konstruktoren sind verschieden, d.h. $\mathsf{Nil} \neq \mathsf{Cons}\ x\ xs$
- Die Konstruktoren sind injektiv, d.h. $\mathsf{Cons}\ x\ xs = \mathsf{Cons}\ y\ ys \Longleftrightarrow x = y \wedge xs = ys$
- Es gilt eine strukturelle Induktionsregel

$$\frac{P\ \mathsf{Nil} \quad \forall x, xs.\ P\ xs \Longrightarrow P\ (\mathsf{Cons}\ x\ xs)}{P\ xs}$$

Datentypen, deren Konstruktoren verschieden und injektiv sind, werden oft auch als *freie Datentypen* bezeichnet. Die Konstruktordarstellung von Elementen des Datentyps ist hierbei stets eindeutig. So gilt

$$\mathsf{Cons}\ x_1\ (\mathsf{Cons}\ x_2\ (\ldots\ (\mathsf{Cons}\ x_m\ \mathsf{Nil})\ldots)) = \mathsf{Cons}\ y_1\ (\mathsf{Cons}\ y_2\ (\ldots\ (\mathsf{Cons}\ y_n\ \mathsf{Nil})\ldots))$$

genau dann, wenn

$$m = n \wedge \forall i.\ 1 \leq i \leq m \Longrightarrow x_i = y_i$$

Datentypen wie α list, für die eine strukturelle Induktionsregel gilt, bezeichnet man als *induktive Datentypen*. Charakteristisch für induktive Datentypen ist, daß jedes Element dieses Typs eine Darstellung als Konstruktorterm besitzt, weshalb derartige Datentypen gelegentlich auch *termerzeugt* genannt werden. Insbesondere sind alle Elemente dieses Typs von endlicher „Tiefe“. Im Fall des Typs α list bedeutet dies, daß die durch

$$xs \prec xs' \equiv \exists x.\ xs' = \mathsf{Cons}\ x\ xs$$

definierte Relation $\prec$ *fundiert* ist. Mit Hilfe der Induktionsregel läßt sich beispielsweise auch zeigen, daß für alle Listen xs stets $xs \neq \mathsf{Cons}\ x\ xs$ gilt.

Funktionen auf derartigen Datentypen werden meist durch Angabe mehrerer charakteristischer Gleichungen unter Verwendung von Pattern Matching spezifiziert. So kann z.B. die Funktion $\mathsf{foldl} :: (\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \mathsf{list} \to \alpha$ durch die Gleichungen

$$\begin{array}{lcl} \mathsf{foldl}\ f\ a\ \mathsf{Nil} & = & a \\ \mathsf{foldl}\ f\ a\ (\mathsf{Cons}\ x\ xs) & = & \mathsf{foldl}\ f\ (f\ a\ x)\ xs \end{array}$$

beschrieben werden. Bei der Umsetzung derartiger Funktionsdefinitionen in HOL ist zu beachten, daß Funktionen in HOL stets total sind. Dies macht Beweise bestimmter Eigenschaften von Funktionen zwar einerseits relativ einfach, erfordert jedoch andererseits bei deren Definition eine gewisse Vorsicht: So läßt sich die durch die Gleichung

$$\mathsf{undefined}(x) = \mathsf{undefined}(x) + 1$$

charakterisierte, für jedes Argument nichtterminierende Funktion in HOL nicht unmittelbar beschreiben. Obige Gleichung würde eine Inkonsistenz zur Folge haben, da sich hieraus leicht $0 = 1$ folgern ließe. Zur Beschreibung partieller Funktionen besser geeignet sind Logiken wie HOLCF, wo jedoch viele Beweise schwierigere Konzepte wie *Fixpunktinduktion* und *Zulässigkeit* erfordern.

Um eine rekursive Funktion in HOL definieren zu können, muß also stets ein Nachweis für deren Terminierung erbracht werden. Bei sogenannten *primitiv rekursiven* Funktionen ist die Terminierung besonders offensichtlich. Im Falle des Datentyps α list versteht man hierunter jede Funktion, die sich mit Hilfe eines Kombinators

$$\mathsf{list_rec} :: \beta \to (\alpha \to \alpha\ \mathsf{list} \to \beta \to \beta) \to \alpha\ \mathsf{list} \to \beta$$

darstellen läßt, der durch die Gleichungen

$$\begin{array}{lcll} \mathsf{list_rec}\ f_1\ f_2\ \mathsf{Nil} & = & f_1 & \text{(list_rec_Nil)} \\ \mathsf{list_rec}\ f_1\ f_2\ (\mathsf{Cons}\ x\ xs) & = & f_2\ x\ xs\ (\mathsf{list_rec}\ f_1\ f_2\ xs) & \text{(list_rec_Cons)} \end{array}$$

charakterisiert wird. So kann die obige Funktion foldl durch

$$\mathsf{foldl} = \lambda f', a', xs'.\ \mathsf{list_rec}\ (\lambda f, a.\ a)\ (\lambda x, xs, r.\ (\lambda f, a.\ r\ f\ (f\ a\ x)))\ xs'\ f'\ a'$$

definiert werden. Wie man leicht nachprüft, gilt

$$\begin{array}{ll} & \mathsf{foldl}\ f\ \ a\ (\mathsf{Cons}\ x\ xs) \\ = & \quad \{\text{Definition von } \mathsf{foldl}\} \\ & \mathsf{list_rec}\ (\lambda f, a.\ a)\ (\lambda x, xs, r.\ (\lambda f, a.\ r\ f\ (f\ \ a\ x)))\ (\mathsf{Cons}\ x\ xs)\ f\ \ a \\ = & \quad \{\text{list_rec_Cons}\} \\ & \mathsf{list_rec}\ (\lambda f, a.\ a)\ (\lambda x, xs, r.\ (\lambda f, a.\ r\ f\ (f\ \ a\ x)))\ xs\ f\ (f\ \ a\ x) \\ = & \quad \{\text{Definition von } \mathsf{foldl}\} \\ & \mathsf{foldl}\ f\ (f\ \ a\ x)\ xs \end{array}$$

Die Existenz eines derartigen Kombinators list_rec kann, wie in Kapitel 4 gezeigt wird, aus den charakteristischen Eigenschaften des Datentyps, d.h. aus der Verschiedenheit und Injektivität der Konstruktoren sowie der Induktionsregel abgeleitet werden. Man beachte, daß das hier beschriebene, in HOL verwendete Konzept der primitiven Rekursion mächtiger ist als das gleichnamige, aus der Berechenbarkeitstheorie bekannte Konzept. So muß nur ein ausgezeichnetes Argument – im Falle der Funktion foldl das dritte Argument – bei jedem rekursiven Aufruf strukturell kleiner werden. Obwohl die anderen Argumente jeden beliebigen Wert annehmen dürfen, ist die Terminierung der Funktion damit sichergestellt.

Ein allgemeinerer Mechanismus zur Definition rekursiver Funktionen, bei dem der Nachweis der Terminierung durch die Angabe einer *fundierten Ordnung* auf dem Typ des rekursiven Arguments erbracht wird, ist in [Slind, 1996] beschrieben.

3.2 Verschränkt rekursive Datentypen

Verschränkt rekursive Datentypen haben die allgemeine Gestalt

$$\begin{array}{llcl} \mathsf{datatype} & (\alpha_1, \ldots, \alpha_h) t_1 & = & C_1^1\ \tau_{1,1}^1\ \ldots\ \tau_{1,m_1^1}^1\ \mid\ \ldots\ \mid\ C_{k_1}^1\ \tau_{k_1,1}^1\ \ldots\ \tau_{k_1,m_{k_1}^1}^1 \\ & & \vdots & \\ \mathsf{and} & (\alpha_1, \ldots, \alpha_h) t_n & = & C_1^n\ \tau_{1,1}^n\ \ldots\ \tau_{1,m_1^n}^n\ \mid\ \ldots\ \mid\ C_{k_n}^n\ \tau_{k_n,1}^n\ \ldots\ \tau_{k_n,m_{k_n}^n}^n \end{array}$$

wobei $\text{TVars}\left(\tau_{i,i'}^j\right) \subseteq \{\alpha_1, \ldots, \alpha_h\}$ gilt und $\tau_{i,i'}^j$ ein *zulässiger* Typausdruck sein muß. Die *Zulässigkeit* eines Typausdrucks ist hierbei wie folgt definiert:

Definition 3.1 (Zulässigkeit von Typausdrücken)
Ein Typausdruck τ in einer verschränkt rekursiven Datentypdefinition heißt *zulässig*, falls gilt:

- τ ist nicht rekursiv, d.h. τ enthält keinen der neu definierten Typkonstruktoren $t_1, \ldots, t_n$, oder
- $\tau = (\alpha_1, \ldots, \alpha_h) t_{j'}$ für $1 \leq j' \leq n$

□

Um die Konstruierbarkeit der Datentypen sicherzustellen, dürfen auf der rechten Seite der „Typgleichungen“ keine spezielleren Instanzen von Typen auf der linken Seite der Gleichungen vorkommen. Bei echt verschränkt rekursiven Datentypdefinitionen kann diese Bedingung nur dann erfüllt sein, wenn alle neu definierten Typkonstruktoren von gleicher Stelligkeit sind, weshalb alle Typkonstruktoren t_1, ..., t_n in der obigen Definitionen die Typparameter α_1, ..., α_h besitzen.

Wie in Abschnitt 2.2.1 angedeutet wurde, dürfen Typen in HOL nicht leer sein. Jeder neue Datentyp $(\alpha_1, \ldots, \alpha_h)t_j$ mit $1 \leq j \leq n$ ist hierbei genau dann nicht leer, falls er einen Konstruktor C_i^j besitzt, für den gilt: Für alle Argumenttypen $\tau_{i,i'}^j$ der Form $(\alpha_1, \ldots, \alpha_h)t_{j'}$ ist der Datentyp $(\alpha_1, \ldots, \alpha_h)t_{j'}$ nicht leer. Eine notwendige Bedingung ist offensichtlich, daß mindestens ein Datentyp $(\alpha_1, \ldots, \alpha_h)t_j$ einen Konstruktor C_i^j ohne rekursive Argumente hat.

Konstruktoren Für die Typen der durch diese Datentypdefinition eingeführten Konstruktorfunktionen gilt

$$C_i^j \ :: \ \tau_{i,1}^j \to \ldots \to \tau_{i,m_i^j}^j \to (\alpha_1, \ldots, \alpha_h)t_j$$

Die Konstruktoren sind verschieden, d.h.

$$i' \neq i'' \Longrightarrow C_{i'}^j \ x_1 \ \ldots \ x_{m_{i'}^j} \neq C_{i''}^j \ y_1 \ \ldots \ y_{m_{i''}^j}$$

und injektiv, d.h.

$$C_i^j \ x_1 \ \ldots \ x_{m_i^j} = C_i^j \ y_1 \ \ldots \ y_{m_i^j} \Longleftrightarrow x_1 = y_1 \wedge \ldots \wedge x_{m_i^j} = y_{m_i^j}$$

Induktion Für die Typen $(\alpha_1, \ldots, \alpha_h)t_1, \ldots, (\alpha_1, \ldots, \alpha_h)t_n$ gilt eine simultane Induktionsregel der Form

$$\frac{\begin{array}{lcl}
\forall x_1, \ldots, x_{m_1^1}.\ P_{s_{1,1}^1}\left(x_{r_{1,1}^1}\right) \wedge \ldots \wedge P_{s_{1,l_1^1}^1}\left(x_{r_{1,l_1^1}^1}\right) & \Longrightarrow & P_1\left(C_1^1 \ x_1 \ \ldots \ x_{m_1^1}\right) \\
 & \vdots & \\
\forall x_1, \ldots, x_{m_{k_1}^1}.\ P_{s_{k_1,1}^1}\left(x_{r_{k_1,1}^1}\right) \wedge \ldots \wedge P_{s_{k_1,l_{k_1}^1}^1}\left(x_{r_{k_1,l_{k_1}^1}^1}\right) & \Longrightarrow & P_1\left(C_{k_1}^1 \ x_1 \ \ldots \ x_{m_{k_1}^1}\right) \\
 & \vdots & \\
\forall x_1, \ldots, x_{m_1^n}.\ P_{s_{1,1}^n}\left(x_{r_{1,1}^n}\right) \wedge \ldots \wedge P_{s_{1,l_1^n}^n}\left(x_{r_{1,l_1^n}^n}\right) & \Longrightarrow & P_n\left(C_1^n \ x_1 \ \ldots \ x_{m_1^n}\right) \\
 & \vdots & \\
\forall x_1, \ldots, x_{m_{k_n}^n}.\ P_{s_{k_n,1}^n}\left(x_{r_{k_n,1}^n}\right) \wedge \ldots \wedge P_{s_{k_n,l_{k_n}^n}^n}\left(x_{r_{k_n,l_{k_n}^n}^n}\right) & \Longrightarrow & P_n\left(C_{k_n}^n \ x_1 \ \ldots \ x_{m_{k_n}^n}\right)
\end{array}}{P_1\left(x_1\right) \wedge \ldots \wedge P_n\left(x_n\right)}$$

wobei

$$\begin{array}{l}
\left\{\left(r_{i,1}^j, s_{i,1}^j\right), \ldots, \left(r_{i,l_i^j}^j, s_{i,l_i^j}^j\right)\right\} = \\
\left\{(i', i'') \ \middle| \ 1 \leq i' \leq m_i^j \wedge 1 \leq i'' \leq n \wedge \tau_{i,i'}^j = (\alpha_1, \ldots, \alpha_h)t_{i''}\right\}
\end{array}$$

Hierbei bezeichnen $r^j_{i,1}, \ldots, r^j_{i,l^j_i}$ die Positionen der rekursiven Argumenttypen für den Konstruktor C^j_i. Die Indizes der in den entsprechenden Positionen vorkommenden rekursiven Typen werden durch $s^j_{i,1}, \ldots, s^j_{i,l^j_i}$ angegeben.

Primitive Rekursion Für jeden der n Typen existiert ein Kombinator für primitive Rekursion

$$\begin{aligned} t_j_\mathsf{rec} \;::\; & (\tau^1_{1,1} \to \cdots \to \tau^1_{1,m^1_1} \to \beta_{s^1_{1,1}} \to \cdots \to \beta_{s^1_{1,l^1_1}} \to \beta_1) \to \\ & \cdots \to \\ & (\tau^1_{k_1,1} \to \cdots \to \tau^1_{k_1,m^1_{k_1}} \to \beta_{s^1_{k_1,1}} \to \cdots \to \beta_{s^1_{k_1,l^1_{k_1}}} \to \beta_1) \to \\ & \cdots \to \\ & (\tau^n_{1,1} \to \cdots \to \tau^n_{1,m^n_1} \to \beta_{s^n_{1,1}} \to \cdots \to \beta_{s^n_{1,l^n_1}} \to \beta_n) \to \\ & \cdots \to \\ & (\tau^n_{k_n,1} \to \cdots \to \tau^1_{k_n,m^n_{k_n}} \to \beta_{s^n_{k_n,1}} \to \cdots \to \beta_{s^n_{k_n,l^n_{k_n}}} \to \beta_n) \to \\ & (\alpha_1, \ldots, \alpha_h)t_j \to \beta_j \end{aligned}$$

mit den charakteristischen Gleichungen

$$\begin{aligned} & t_j_\mathsf{rec}\; f^1_1 \;\ldots\; f^1_{k_1} \;\ldots\; f^n_1 \;\ldots\; f^n_{k_n} \; \left(C^j_i \; x_1 \;\ldots\; x_{m^j_i}\right) = \\ & f^j_i \; x_1 \;\ldots\; x_{m^j_i} \; \left(t_{s^j_{i,1}}_\mathsf{rec}\; f^1_1 \;\ldots\; f^n_{k_n} \; x_{r^j_{i,1}}\right) \;\ldots\; \left(t_{s^j_{i,l^j_i}}_\mathsf{rec}\; f^1_1 \;\ldots\; f^n_{k_n} \; x_{r^j_{i,l^j_i}}\right) \end{aligned}$$

Für manche Beweise ist anstatt von Induktion bereits eine Fallunterscheidung über alle Konstruktoren des Datentyps t_j ausreichend, wofür die Regel

$$\frac{\begin{array}{lcl} \forall x_1, \ldots, x_{m^j_1}.\; u = C^j_1 \; x_1 \;\ldots\; x_{m^j_1} & \Longrightarrow & P \\ & \vdots & \\ \forall x_1, \ldots, x_{m^j_{k_j}}.\; u = C^j_{k_j} \; x_1 \;\ldots\; x_{m^j_{k_j}} & \Longrightarrow & P \end{array}}{P}$$

verwendet wird, die sich leicht aus der obigen Induktionsregel ableiten läßt. Anstelle des Kombinators t_j_rec ist zur Definition von Funktionen auf $(\alpha_1, \ldots, \alpha_h)t_j$ manchmal auch der einfachere Kombinator

$$\begin{aligned} t_j_\mathsf{case} \;::\; & (\tau^j_{1,1} \to \cdots \to \tau^j_{1,m^j_1} \to \beta) \to \cdots \to (\tau^j_{k_j,1} \to \cdots \to \tau^j_{k_j,m^j_{k_j}} \to \beta) \to \\ & (\alpha_1, \ldots, \alpha_h)t_j \to \beta \end{aligned}$$

mit den charakteristischen Gleichungen

$$t_j_\mathsf{case}\; f_1 \;\ldots\; f_{k_j} \; \left(C^j_i \; x_1 \;\ldots\; x_{m^j_i}\right) = f_i \; x_1 \;\ldots\; x_{m^j_i}$$

ausreichend, der sich leicht auf den entsprechenden Kombinator t_j_rec zurückführen läßt.

Ein Beispiel für verschränkt rekursive Datentypen sind die bereits aus Abschnitt 1.4 bekannten arithmetischen und booleschen Ausdrücke, die sich wie folgt definieren lassen:

$$\begin{array}{llcl}
\mathsf{datatype} & \alpha\ \mathsf{aexp} & = & \mathsf{If_then_else}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & \mid & \mathsf{Sum}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & \mid & \mathsf{Diff}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & \mid & \mathsf{Var}\ \alpha \\
 & & \mid & \mathsf{Num\ nat} \\
\mathsf{and} & \alpha\ \mathsf{bexp} & = & \mathsf{Less}\ (\alpha\ \mathsf{aexp})\ (\alpha\ \mathsf{aexp}) \\
 & & \mid & \mathsf{And}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{bexp}) \\
 & & \mid & \mathsf{Or}\ (\alpha\ \mathsf{bexp})\ (\alpha\ \mathsf{bexp})
\end{array}$$

für die Typen α aexp und α bexp erhält man die Induktionsregel

$$\frac{\begin{array}{lcl}
\forall b, a_1, a_2.\ P_2\ b \wedge P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_1\ (\mathsf{If_then_else}\ b\ a_1\ a_2) \\
\forall a_1, a_2.\ P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_1\ (\mathsf{Sum}\ a_1\ a_2) \\
\forall a_1, a_2.\ P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_1\ (\mathsf{Diff}\ a_1\ a_2) \\
\forall v. & & P_1\ (\mathsf{Var}\ v) \\
\forall n. & & P_1\ (\mathsf{Num}\ n) \\
\forall a_1, a_2.\ P_1\ a_1 \wedge P_1\ a_2 & \Longrightarrow & P_2\ (\mathsf{Less}\ a_1\ a_2) \\
\forall b_1, b_2.\ P_2\ b_1 \wedge P_2\ b_2 & \Longrightarrow & P_2\ (\mathsf{And}\ b_1\ b_2) \\
\forall b_1, b_2.\ P_2\ b_1 \wedge P_2\ b_2 & \Longrightarrow & P_2\ (\mathsf{Or}\ b_1\ b_2)
\end{array}}{P_1\ a \wedge P_2\ b}$$

Die Kombinatoren $\mathsf{aexp_rec}$ und $\mathsf{bexp_rec}$ für primitive Rekursion besitzen die charakteristischen Gleichungen

$$\begin{array}{l}
\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{If_then_else}\ b\ a_1\ a_2) = \\
\quad f_1\ b\ a_1\ a_2\ (\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ b)\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_1)\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_2) \\
\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{Sum}\ a_1\ a_2) = \\
\quad f_2\ a_1\ a_2\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_1)\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_2) \\
\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{Diff}\ a_1\ a_2) = \\
\quad f_3\ a_1\ a_2\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_1)\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_2) \\
\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{Var}\ v) = f_4\ v \\
\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{Num}\ n) = f_5\ n
\end{array}$$

$$\begin{array}{l}
\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{Less}\ a_1\ a_2) = \\
\quad f_6\ a_1\ a_2\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_1)\ (\mathsf{aexp_rec}\ f_1\ \ldots\ f_8\ a_2) \\
\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{And}\ b_1\ b_2) = \\
\quad f_7\ b_1\ b_2\ (\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ b_1)\ (\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ b_2) \\
\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ (\mathsf{Or}\ b_1\ b_2) = \\
\quad f_8\ b_1\ b_2\ (\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ b_1)\ (\mathsf{bexp_rec}\ f_1\ \ldots\ f_8\ b_2)
\end{array}$$

3.3 Verschachtelt rekursive Datentypen

Wir betrachten nun Datentypdefinitionen der Form

$$
\begin{array}{lll}
\textsf{datatype} & (\alpha_1,\dots,\alpha_h)t_1 & = & C_1^1\ \tau_{1,1}^1\ \dots\ \tau_{1,m_1^1}^1\ \mid\ \dots\ \mid\ C_{k_1}^1\ \tau_{k_1,1}^1\ \dots\ \tau_{k_1,m_{k_1}^1}^1 \\
 & & \vdots & \\
\textsf{and} & (\alpha_1,\dots,\alpha_h)t_n & = & C_1^n\ \tau_{1,1}^n\ \dots\ \tau_{1,m_1^n}^n\ \mid\ \dots\ \mid\ C_{k_n}^n\ \tau_{k_n,1}^n\ \dots\ \tau_{k_n,m_{k_n}^n}^n
\end{array}
$$

wobei für die Typen $\tau_{i,i'}^j$ nun eine gegenüber Abschnitt 3.2 leicht geänderte Zulässigkeitsbedingung gilt.

Definition 3.2 (Zulässigkeit von Typausdrücken)
Ein Typausdruck τ in einer verschachtelt rekursiven Datentypdefinition heißt *zulässig*, falls gilt:

- τ ist nicht rekursiv, d.h. τ enthält keinen der neu definierten Typkonstruktoren $t_1,\dots,t_n$, oder
- $\tau = (\alpha_1,\dots,\alpha_h)t_{j'}$ für $1 \le j' \le n$, oder
- $\tau = (\tau_1',\dots,\tau_{\tilde{h}}')\tilde{t}_{j''}$, wobei $\tilde{t}_{j''}$ der Typkonstruktor eines bereits durch eine verschränkt rekursive Datentypdefinition

$$
\begin{array}{llll}
\textsf{datatype} & (\beta_1,\dots,\beta_{\tilde{h}})\tilde{t}_1 & = & D_1^1\ \tilde{\tau}_{1,1}^1\ \dots\ \tilde{\tau}_{1,\tilde{m}_1^1}^1\ \mid\ \dots\ \mid\ D_{\tilde{k}_1}^1\ \tilde{\tau}_{\tilde{k}_1,1}^1\ \dots\ \tilde{\tau}_{\tilde{k}_1,\tilde{m}_{\tilde{k}_1}^1}^1 \\
 & & \vdots & \\
\textsf{and} & (\beta_1,\dots,\beta_{\tilde{h}})\tilde{t}_{\tilde{n}} & = & D_1^{\tilde{n}}\ \tilde{\tau}_{1,1}^{\tilde{n}}\ \dots\ \tilde{\tau}_{1,\tilde{m}_1^{\tilde{n}}}^{\tilde{n}}\ \mid\ \dots\ \mid\ D_{\tilde{k}_{\tilde{n}}}^{\tilde{n}}\ \tilde{\tau}_{\tilde{k}_{\tilde{n}},1}^{\tilde{n}}\ \dots\ \tilde{\tau}_{\tilde{k}_{\tilde{n}},\tilde{m}_{\tilde{k}_{\tilde{n}}}^{\tilde{n}}}^{\tilde{n}}
\end{array}
$$

  im Sinne von Abschnitt 3.2 gegebenen Datentyps ist und für alle Typausdrücke $\tilde{\tau}_{\tilde{\imath},\tilde{\imath}'}^{\tilde{\jmath}}$, die nicht von der Form $(\beta_1,\dots,\beta_{\tilde{h}})\tilde{t}_{j'''}$ sind, der Typausdruck $\tilde{\tau}_{\tilde{\imath},\tilde{\imath}'}^{\tilde{\jmath}}\left[\tau_1'/\beta_1,\dots,\tau_{\tilde{h}}'/\beta_{\tilde{h}}\right]$ zulässig ist.

□

Bemerkung 3.3
Wie später gezeigt wird, können verschachtelt rekursive Datentypen auf verschränkt rekursive Datentypen ohne Verschachtelungen zurückgeführt werden. Wir dürfen daher o.B.d.A. davon ausgehen, daß es sich bei den in der oben beschriebenen Datentypdefinition vorkommenden, bereits früher definierten Datentypen um verschränkt rekursive Datentypen im Sinne von Abschnitt 3.2 handelt.

Beispiel 3.4
Sei die korrekte Datentypdefinition

$$
\begin{array}{llll}
\textsf{datatype} & (\alpha,\beta)\textsf{example1} & = & \textsf{C1}\ (\alpha \to \textsf{bool})\ (\beta\ \textsf{list}) \\
 & & \mid & \textsf{C2}\ ((\alpha,\beta)\textsf{example1})
\end{array}
$$

gegeben. Dann ist

$$\begin{array}{lll} \mathsf{datatype} \quad \gamma\ \mathsf{example2} & = & \mathsf{C3}\ ((\gamma\ \mathsf{example2}, \gamma)\mathsf{example1}) \\ & | & \mathsf{C4}\ \gamma \end{array}$$

keine korrekte Datentypdefinition, denn γ example2 $\rightarrow$ bool ist kein korrekter Typausdruck im Sinne der obigen Definition. Die Definition

$$\begin{array}{lll} \mathsf{datatype} \quad \gamma\ \mathsf{example3} & = & \mathsf{C5}\ ((\gamma, \gamma\ \mathsf{example3})\mathsf{example1}) \\ & | & \mathsf{C6}\ \gamma \end{array}$$

ist jedoch korrekt, denn (γ example3)list und $\gamma \rightarrow$ bool sind korrekte Typausdrücke. □

Ein weiteres Beispiel für einen Datentyp mit verschachtelter Rekursion ist der Datentyp der Terme, der durch

$$\begin{array}{lll} \mathsf{datatype} \quad (\alpha, \beta)\mathsf{term} & = & \mathsf{Var}\ \alpha \\ & | & \mathsf{App}\ \beta\ (((\alpha, \beta)\mathsf{term})\mathsf{list}) \end{array}$$

definiert ist. Zur Handhabung derartiger Datentypen sind in der einschlägigen Literatur im wesentlichen zwei verschiedene Verfahren zu finden. Mit einem dieser Verfahren, das auch für das im Rahmen dieser Arbeit implementierte Datentyppaket für Isabelle/HOL gewählt wurde, beschäftigt sich der folgende Abschnitt. Ein etwas anderer Ansatz ist in Abschnitt 6.2 beschrieben.

Formulierung mittels Auffalten In [Gunter, 1992, Gunter, 1993] wird vorgeschlagen, Datentypen mit verschachtelter Rekursion mittels „Auffalten“ auf verschränkt rekursive Datentypen zurückzuführen: Sei eine Datentypdefinition der Form

$$\begin{array}{llll} \mathsf{datatype} & (\alpha_1, \ldots, \alpha_h)t_1 & = & C_1^1\ \tau_{1,1}^1\ \ldots\ \tau_{1,m_1^1}^1 \ \mid\ \ldots\ \mid\ C_{k_1}^1\ \tau_{k_1,1}^1\ \ldots\ \tau_{k_1,m_{k_1}^1}^1 \\ & & \vdots & \\ \mathsf{and} & (\alpha_1, \ldots, \alpha_h)t_j & = & \ldots \\ & & | & C_i^j\ \tau_{i,1}^j\ \ldots\ \tau_{i,i'-1}^j\ (\tau_1', \ldots, \tau_{\tilde{h}}')\tilde{t}_{j'}\ \tau_{i,i'+1}^j\ \ldots\ \tau_{1,m_i^j}^1 \\ & & | & \ldots \\ & & \vdots & \\ \mathsf{and} & (\alpha_1, \ldots, \alpha_h)t_n & = & C_1^n\ \tau_{1,1}^n\ \ldots\ \tau_{1,m_1^n}^n \ \mid\ \ldots\ \mid\ C_{k_n}^n\ \tau_{k_n,1}^n\ \ldots\ \tau_{k_n,m_{k_n}^n}^n \end{array}$$

gegeben, wobei $\tau_1', \ldots, \tau_{\tilde{h}}'$ die neu definierten Typkonstruktoren $t_1, \ldots, t_n$ enthalten und der Datentyp $\tilde{t}_{j'}$ durch eine verschränkt rekursive Datentypdefinition der Form

$$\begin{array}{llll} \mathsf{datatype} & (\beta_1, \ldots, \beta_{\tilde{h}})\tilde{t}_1 & = & D_1^1\ \tilde{\tau}_{1,1}^1\ \ldots\ \tilde{\tau}_{1,\tilde{m}_1^1}^1 \ \mid\ \ldots\ \mid\ D_{\tilde{k}_1}^1\ \tilde{\tau}_{\tilde{k}_1,1}^1\ \ldots\ \tilde{\tau}_{\tilde{k}_1,\tilde{m}_{\tilde{k}_1}^1}^1 \\ & & \vdots & \\ \mathsf{and} & (\beta_1, \ldots, \beta_{\tilde{h}})\tilde{t}_{\tilde{n}} & = & D_1^{\tilde{n}}\ \tilde{\tau}_{1,1}^{\tilde{n}}\ \ldots\ \tilde{\tau}_{1,\tilde{m}_1^{\tilde{n}}}^{\tilde{n}} \ \mid\ \ldots\ \mid\ D_{\tilde{k}_{\tilde{n}}}^{\tilde{n}}\ \tilde{\tau}_{\tilde{k}_{\tilde{n}},1}^{\tilde{n}}\ \ldots\ \tilde{\tau}_{\tilde{k}_{\tilde{n}},\tilde{m}_{\tilde{k}_{\tilde{n}}}^{\tilde{n}}}^{\tilde{n}} \end{array}$$

beschrieben ist. Wir definieren nun eine Transformationsfunktion trans auf Typausdrücken τ durch

$$\mathsf{trans}(\tau) = \begin{cases} (\alpha_1, \ldots, \alpha_h)\widehat{t}_{j''} & \text{falls} \quad \tau = (\beta_1, \ldots, \beta_{\tilde{h}})\tilde{t}_{j''} \\ \tau\left[\tau'_1/\beta_1, \ldots, \tau'_{\tilde{h}}/\beta_{\tilde{h}}\right] & \text{sonst} \end{cases}$$

wobei $\widehat{t}_1, \ldots, \widehat{t}_{\tilde{n}}$ bisher nicht verwendete Typkonstruktoren sind. Dann kann die obige Typdefinition zu

$$\begin{array}{lllll}
\mathsf{datatype} & (\alpha_1, \ldots, \alpha_h)t_1 & = & C^1_1\ \tau^1_{1,1}\ \ldots\ \tau^1_{1,m^1_1}\ \mid\ \ldots\ \mid\ C^1_{k_1}\ \tau^1_{k_1,1}\ \ldots\ \tau^1_{k_1,m^1_{k_1}} \\
 & & \vdots & \\
\mathsf{and} & (\alpha_1, \ldots, \alpha_h)t_j & = & \ldots \\
 & & \mid & C^j_i\ \tau^j_{i,1}\ \ldots\ \tau^j_{i,i'-1}\ (\alpha_1, \ldots, \alpha_h)\widehat{t}_{j'}\ \tau^j_{i,i'+1}\ \ldots\ \tau^1_{1,m^j_i} \\
 & & \mid & \ldots \\
 & & \vdots & \\
\mathsf{and} & (\alpha_1, \ldots, \alpha_h)t_n & = & C^n_1\ \tau^n_{1,1}\ \ldots\ \tau^n_{1,m^n_1}\ \mid\ \ldots\ \mid\ C^n_{k_n}\ \tau^n_{k_n,1}\ \ldots\ \tau^n_{k_n,m^n_{k_n}} \\
\mathsf{and} & (\alpha_1, \ldots, \alpha_h)\widehat{t}_1 & = & D^1_1\ \left(\mathsf{trans}\ \tilde{\tau}^1_{1,1}\right)\ \ldots\ \left(\mathsf{trans}\ \tilde{\tau}^1_{1,\tilde{m}^1_1}\right) \\
 & & \mid & \ldots \\
 & & \mid & D^1_{\tilde{k}_1}\ \left(\mathsf{trans}\ \tilde{\tau}^1_{\tilde{k}_1,1}\right)\ \ldots\ \left(\mathsf{trans}\ \tilde{\tau}^1_{\tilde{k}_1,\tilde{m}^1_{\tilde{k}_1}}\right) \\
 & & \vdots & \\
\mathsf{and} & (\alpha_1, \ldots, \alpha_h)\widehat{t}_{\tilde{n}} & = & D^{\tilde{n}}_1\ \left(\mathsf{trans}\ \tilde{\tau}^{\tilde{n}}_{1,1}\right)\ \ldots\ \left(\mathsf{trans}\ \tilde{\tau}^{\tilde{n}}_{1,\tilde{m}^{\tilde{n}}_1}\right) \\
 & & \mid & \ldots \\
 & & \mid & D^{\tilde{n}}_{\tilde{k}_{\tilde{n}}}\ \left(\mathsf{trans}\ \tilde{\tau}^{\tilde{n}}_{\tilde{k}_{\tilde{n}},1}\right)\ \ldots\ \left(\mathsf{trans}\ \tilde{\tau}^{\tilde{n}}_{\tilde{k}_{\tilde{n}},\tilde{m}^{\tilde{n}}_{\tilde{k}_{\tilde{n}}}}\right)
\end{array}$$

umgeformt werden. Durch wiederholtes Anwenden dieser Transformation erhält man schließlich eine Datentypdefinition im Sinne von Abschnitt 3.2, in der die neu definierten Typen nicht mehr verschachtelt vorkommen.

Der eingangs erwähnte Typ term ließe sich demnach zu

$$\begin{array}{llll}
\mathsf{datatype} & (\alpha, \beta)\mathsf{term} & = & \mathsf{Var}\ \alpha \\
 & & \mid & \mathsf{App}\ \beta\ ((\alpha, \beta)\mathsf{term_list}) \\
\mathsf{and} & (\alpha, \beta)\mathsf{term_list} & = & \mathsf{Nil'} \\
 & & \mid & \mathsf{Cons'}\ ((\alpha, \beta)\mathsf{term})\ ((\alpha, \beta)\mathsf{term_list})
\end{array}$$

auffalten. Man beachte, daß der Typ term_list sowie die Konstruktoren Nil' und Cons' nicht wirklich eingeführt werden, da dies zur Folge hätte, daß für den Datentyp list bewiesene Theoreme in Beweisen von Theoremen über term nicht mehr verwendbar wären. Vielmehr muß, wie in Abschnitt 4.3 dargestellt wird, die Isomorphie zwischen dem Typ $((\alpha, \beta)\mathsf{term})\mathsf{list}$ und der repräsentierenden Menge des „aufgefalteten“ (nicht wirklich eingeführten) Typs $(\alpha, \beta)\mathsf{term_list}$

nachgewiesen werden. Die Induktionsregel für term lautet daher

$$\frac{\begin{array}{lcl} \forall a. & & P_1\ (\mathsf{Var}\ a) \\ \forall b, ts.\ P_2\ ts & \Longrightarrow & P_1\ (\mathsf{App}\ b\ ts) \\ & & P_2\ \mathsf{Nil} \\ \forall t, ts.\ P_1\ t \wedge P_2\ ts & \Longrightarrow & P_2\ (\mathsf{Cons}\ t\ ts) \end{array}}{P_1\ t \wedge P_2\ ts}$$

Aufgrund der Auffaltung sind für primitive Rekursion nun die zwei Kombinatoren

$$\begin{array}{lcl} \mathsf{term_rec} & : & (\alpha \to \gamma) \to (\beta \to (((\alpha,\beta)\mathsf{term})\mathsf{list}) \to \delta \to \gamma) \to \\ & & \delta \to ((\alpha,\beta)\mathsf{term} \to (((\alpha,\beta)\mathsf{term})\mathsf{list}) \to \gamma \to \delta \to \delta) \to \\ & & (\alpha,\beta)\mathsf{term} \to \gamma \\ \mathsf{term_list_rec} & : & (\alpha \to \gamma) \to (\beta \to (((\alpha,\beta)\mathsf{term})\mathsf{list}) \to \delta \to \gamma) \to \\ & & \delta \to ((\alpha,\beta)\mathsf{term} \to (((\alpha,\beta)\mathsf{term})\mathsf{list}) \to \gamma \to \delta \to \delta) \to \\ & & (((\alpha,\beta)\mathsf{term})\mathsf{list}) \to \delta \end{array}$$

erforderlich, die die charakteristischen Gleichungen

$$\mathsf{term_rec}\ f_1\ \ldots\ f_4\ (\mathsf{Var}\ a) = f_1\ a$$

$$\mathsf{term_rec}\ f_1\ \ldots\ f_4\ (\mathsf{App}\ b\ ts) = f_2\ b\ ts\ (\mathsf{term_list_rec}\ f_1\ \ldots\ f_4\ ts)$$

$$\mathsf{term_list_rec}\ f_1\ \ldots\ f_4\ \mathsf{Nil} = f_3$$

$$\begin{array}{l} \mathsf{term_list_rec}\ f_1\ \ldots\ f_4\ (\mathsf{Cons}\ t\ ts) = \\ \quad f_4\ t\ ts\ (\mathsf{term_rec}\ f_1\ \ldots\ f_4\ t)\ (\mathsf{term_list_rec}\ f_1\ \ldots\ f_4\ ts) \end{array}$$

besitzen. Bei tiefer verschachtelten Typen wie

$$\begin{array}{llcl} \mathsf{datatype} & (\alpha,\beta)\mathsf{term2} & = & \mathsf{Var}\ \alpha \\ & & \mid & \mathsf{App}\ \beta\ ((((\alpha,\beta)\mathsf{term2})\mathsf{list})\mathsf{list}) \end{array}$$

muß mehrmals aufgefaltet werden. So erhält man für term2 die Auffaltung

$$\begin{array}{llcl} \mathsf{datatype} & (\alpha,\beta)\mathsf{term2} & = & \mathsf{Var}\ \alpha \\ & & \mid & \mathsf{App}\ \beta\ ((\alpha,\beta)\mathsf{term2_list_list}) \\ \mathsf{and} & (\alpha,\beta)\mathsf{term2_list_list} & = & \mathsf{Nil'} \\ & & \mid & \mathsf{Cons'}\ ((\alpha,\beta)\mathsf{term2_list})\ ((\alpha,\beta)\mathsf{term2_list_list}) \\ \mathsf{and} & (\alpha,\beta)\mathsf{term2_list} & = & \mathsf{Nil''} \\ & & \mid & \mathsf{Cons''}\ ((\alpha,\beta)\mathsf{term2})\ ((\alpha,\beta)\mathsf{term2_list}) \end{array}$$

Beispiel 3.5
Um die Anwendung der soeben dargestellten Theoreme für term zu demonstrieren, wollen wir im folgenden die Substitution von Variablen in Termen durch Terme formalisieren. Zu diesem Zweck werden die beiden Funktionen

$$\mathsf{subst_term} :: (\alpha \to (\alpha,\beta)\mathsf{term}) \to (\alpha,\beta)\mathsf{term} \to (\alpha,\beta)\mathsf{term}$$

$$\mathsf{subst_term_list} :: (\alpha \to (\alpha,\beta)\mathsf{term}) \to ((\alpha,\beta)\mathsf{term})\mathsf{list} \to ((\alpha,\beta)\mathsf{term})\mathsf{list}$$

benötigt. Die Idee ist, eine gegebene Funktion f vom Typ $\alpha \to (\alpha, \beta)\mathsf{term}$ auf Terme bzw. Listen von Termen zu erweitern. Das Verhalten der primitiv rekursiven Funktionen subst_term und subst_term_list kann durch die Gleichungen

$$\begin{array}{lcl} \mathsf{subst_term}\ f\ (\mathsf{Var}\ a) & = & f\ a \\ \mathsf{subst_term}\ f\ (\mathsf{App}\ b\ ts) & = & \mathsf{App}\ b\ (\mathsf{subst_term_list}\ f\ ts) \\[1ex] \mathsf{subst_term_list}\ f\ \mathsf{Nil} & = & \mathsf{Nil} \\ \mathsf{subst_term_list}\ f\ (\mathsf{Cons}\ t\ ts) & = & \mathsf{Cons}\ (\mathsf{subst_term}\ f\ t)\ (\mathsf{subst_term_list}\ f\ ts) \end{array}$$

charakterisiert werden. Mit Hilfe der Kombinatoren term_rec und term_list_rec definieren wir nun

$$\begin{array}{lcl} \mathsf{subst_term}\ f & \equiv & \mathsf{term_rec}\ f\ (\lambda b, xs, ys.\ \mathsf{App}\ b\ ys)\ \mathsf{Nil}\ (\lambda x, xs, y, ys.\ \mathsf{Cons}\ y\ ys) \\ \mathsf{subst_term_list}\ f & \equiv & \mathsf{term_list_rec}\ f\ (\lambda b, xs, ys.\ \mathsf{App}\ b\ ys)\ \mathsf{Nil}\ (\lambda x, xs, y, ys.\ \mathsf{Cons}\ y\ ys) \end{array}$$

woraus sich die obigen charakteristischen Gleichungen sofort ableiten lassen. Für die Komposition von Substitutionsfunktionen auf $(\alpha, \beta)\mathsf{term}$ gilt offenbar

$$\mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ t = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ t)$$

Um dies mit der oben angegebenen Induktionsregel beweisen zu können, muß allerdings auch noch eine entsprechende Eigenschaft für Substitutionsfunktionen auf $((\alpha, \beta)\mathsf{term})\mathsf{list}$ formuliert werden. Wir wählen also das Beweisziel

$$\begin{array}{l} \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ t = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ t) \wedge \\ \mathsf{subst_term_list}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ ts = \mathsf{subst_term_list}\ f_1\ (\mathsf{subst_term_list}\ f_2\ ts) \end{array}$$

Mittels simultaner Induktion über t und ts erhält man hieraus die neuen Beweisziele

(1) $\forall a.\ \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ (\mathsf{Var}\ a) = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ (\mathsf{Var}\ a))$

(2) $$\begin{array}{l} \forall b, xs.\ \mathsf{subst_term_list}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ xs = \\ \quad \mathsf{subst_term_list}\ f_1\ (\mathsf{subst_term_list}\ f_2\ xs) \Longrightarrow \\ \qquad \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ (\mathsf{App}\ b\ xs) = \\ \qquad \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ (\mathsf{App}\ b\ xs)) \end{array}$$

(3) $\mathsf{subst_term_list}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ \mathsf{Nil} = \mathsf{subst_term_list}\ f_1\ (\mathsf{subst_term_list}\ f_2\ \mathsf{Nil})$

(4) $$\begin{array}{l} \forall x, xs.\ \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ x = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ x) \wedge \\ \quad \mathsf{subst_term_list}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ xs = \\ \quad \mathsf{subst_term_list}\ f_1\ (\mathsf{subst_term_list}\ f_2\ xs) \Longrightarrow \\ \qquad \mathsf{subst_term_list}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ (\mathsf{Cons}\ x\ xs) = \\ \qquad \mathsf{subst_term_list}\ f_1\ (\mathsf{subst_term_list}\ f_2\ (\mathsf{Cons}\ x\ xs)) \end{array}$$

Beweisziel (4) kann nun beispielsweise durch die folgende Umformung gezeigt werden:

$$
\begin{aligned}
& \mathsf{subst_term_list}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ (\mathsf{Cons}\ x\ xs) \\
= \quad & \{\text{charakteristische Gleichungen für } \mathsf{subst_term_list}\} \\
& \mathsf{Cons}\ (\mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ x)\ (\mathsf{subst_term_list}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ xs) \\
= \quad & \{\text{Induktionsvoraussetzung}\} \\
& \mathsf{Cons}\ (\mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ x))\ (\mathsf{subst_term_list}\ f_1\ (\mathsf{subst_term_list}\ f_2\ xs)) \\
= \quad & \{\text{charakteristische Gleichungen für } \mathsf{subst_term_list}\} \\
& \mathsf{subst_term_list}\ f_1\ (\mathsf{Cons}\ (\mathsf{subst_term}\ f_2\ x)\ (\mathsf{subst_term_list}\ f_2\ xs)) \\
= \quad & \{\text{charakteristische Gleichungen für } \mathsf{subst_term_list}\} \\
& \mathsf{subst_term_list}\ f_1\ (\mathsf{subst_term_list}\ f_2\ (\mathsf{Cons}\ x\ xs))
\end{aligned}
$$

□

Kapitel 4

Konstruktion von Datentypen in HOL

Im folgenden Kapitel wird aufgezeigt, wie die Datentypen, deren Eigenschaften in Kapitel 3 abstrakt beschrieben wurden, konkret in Isabelle/HOL konstruiert werden können. Im einzelnen umfaßt diese Konstruktion folgende Teilschritte:

1. Auffalten verschachtelt rekursiver Definitionen
2. Konstruktion der repräsentierenden Mengen
3. Definition der neuen Typen
4. Beweis von Isomorphie-Eigenschaften
5. Definition der Konstruktoren
6. Repräsentationsabhängige Beweise charakteristischer Theoreme
    - Verschiedenheit von Konstruktoren
    - Injektivität von Konstruktoren
    - Strukturelle Induktion
7. Repräsentationsunabhängige Beweise charakteristischer Theoreme
    - Fallunterscheidung
    - Primitive Rekursion
    - case-Kombinator
    - ...

Die hier beschriebenen Verfahren sind im Rahmen dieser Arbeit in Form eines Datentyppakets für Isabelle/HOL implementiert worden. Durch die definitorische Vorgehensweise ist sichergestellt, daß die durch das Paket abgeleiteten Regeln auch korrekt sind. Eine weitere Frage ist, ob die Verfahren auch vollständig sind, d.h. für alle in Kapitel 3 beschriebenen Datentypen funktionieren. Dies soll in den folgenden Abschnitten geklärt werden.

4.1 Auffalten verschachtelt rekursiver Definitionen

In Abschnitt 3.3 wurde bereits erwähnt, daß verschachtelt rekursive Definitionen durch Auffalten realisiert werden sollen. Ergebnis dieser Auffaltung ist eine Typdefinition der Form

$$
\begin{array}{lll}
\textsf{datatype} & (\alpha_1,\ldots,\alpha_h)t_1 & = C_1^1 \ \ldots \mid \ldots \mid C_{k_1}^1 \ \ldots \\
 & & \vdots \\
\textsf{and} & (\alpha_1,\ldots,\alpha_h)t_{\nu_1-1} & = C_1^{\nu_1-1} \ \ldots \mid \ldots \mid C_{k_{\nu_1-1}}^{\nu_1-1} \ \ldots \\
\textsf{and} & (\alpha_1,\ldots,\alpha_h)t_{\nu_1} & = C_1^{\nu_1} \ \ldots \mid \ldots \mid C_{k_{\nu_1}}^{\nu_1} \ \ldots \\
 & & \vdots \\
\textsf{and} & (\alpha_1,\ldots,\alpha_h)t_{\nu_2-1} & = C_1^{\nu_2-1} \ \ldots \mid \ldots \mid C_{k_{\nu_2-1}}^{\nu_2-1} \ \ldots \\
 & & \vdots \\
\textsf{and} & (\alpha_1,\ldots,\alpha_h)t_{\nu_{n'}} & = C_1^{\nu_{n'}} \ \ldots \mid \ldots \mid C_{k_{\nu_{n'}}}^{\nu_{n'}} \ \ldots \\
 & & \vdots \\
\textsf{and} & (\alpha_1,\ldots,\alpha_h)t_n & = C_1^n \ \ldots \mid \ldots \mid C_{k_n}^n \ \ldots
\end{array}
$$

die im folgenden als Ausgangspunkt für die abstrakte Beschreibung der Konstruktionen und Beweise dienen soll. Hierbei sind nur die Typen $(\alpha_1,\ldots,\alpha_h)t_1$, ..., $(\alpha_1,\ldots,\alpha_h)t_{\nu_1-1}$ tatsächlich neu, während die n' Gruppen von Typen

$$
\begin{array}{lll}
(\alpha_1,\ldots,\alpha_h)t_{\nu_1} & \ldots & (\alpha_1,\ldots,\alpha_h)t_{\nu_2-1} \\
 & \vdots & \\
(\alpha_1,\ldots,\alpha_h)t_{\nu_{n'}} & \ldots & (\alpha_1,\ldots,\alpha_h)t_n
\end{array}
$$

jeweils in einem Auffaltungsschritt entstandene Kopien bereits existierender, verschränkt rekursiver Typen

$$
\begin{array}{lll}
\left(\hat{\tau}_1^{\nu_1},\ldots,\hat{\tau}_{\tilde{h}_1}^{\nu_1}\right)\tilde{t}_{\nu_1} & \ldots & \left(\hat{\tau}_1^{\nu_2-1},\ldots,\hat{\tau}_{\tilde{h}_1}^{\nu_2-1}\right)\tilde{t}_{\nu_2-1} \\
 & \vdots & \\
\left(\hat{\tau}_1^{\nu_{n'}},\ldots,\hat{\tau}_{\tilde{h}_{n'}}^{\nu_{n'}}\right)\tilde{t}_{\nu_{n'}} & \ldots & \left(\hat{\tau}_1^{n},\ldots,\hat{\tau}_{\tilde{h}_{n'}}^{n}\right)\tilde{t}_n
\end{array}
$$

sind, wobei die Reihenfolge der Aufschreibung der Reihenfolge der Auffaltung entspricht. Bei der Konstruktion der neuen Typen kann daher auf bereits bewiesene charakteristische Theoreme der schon existierenden Typen wie z.B. simultane Induktionsregeln für $[\tilde{t}_{\nu_1},\ldots,\tilde{t}_{\nu_2-1}]$, $\ldots$, $[\tilde{t}_{\nu_{n'}},\ldots,\tilde{t}_n]$ sowie auf die Kombinatoren $\tilde{t}_{\nu_1}$_rec, ..., $\tilde{t}_n$_rec für primitive Rekursion zurückgegriffen werden.

In der aufgefalteten Definition von term taucht beispielsweise der Typ (α,β)term_list auf, der eine Kopie des Typs $((\alpha,\beta)$term$)$list ist. Faltet man die Definition des Typs term2 auf, enthält die entstehende Definition zusätzlich den Typ (α,β)term2_list_list, der eine Kopie des Typs $(((\alpha,\beta)$term2$)$list$)$list ist, sowie den Typ (α,β)term2_list, bei dem es sich um eine Kopie des Typs $((\alpha,\beta)$term2$)$list handelt.

4.2 Konstruktion der repräsentierenden Mengen

In Abschnitt 2.2.3 wurde erläutert, wie neue Typen durch Angabe einer repräsentierenden Menge mit geeigneten Eigenschaften definitorisch eingeführt werden können. Als repräsentierende Menge dient hierbei eine Teilmenge eines bereits existierenden Typs. Es stellt sich daher die Frage, welcher Typ für diesen Zweck passend ist und wie die entsprechende Teilmenge dieses Typs am besten charakterisiert werden kann. Vergleicht man die gewünschten Eigenschaften der repräsentierenden Menge eines Datentyps mit den in Abschnitt 2.3.2 beschriebenen Eigenschaften induktiv definierter Mengen, so fallen eine Reihe von Gemeinsamkeiten auf. So sollte auch die repräsentierende Menge eines Datentyps unter gewissen Regeln abgeschlossen sein. Ist z.B. xs eine Liste, so sollte auch $\mathsf{Cons}\ x\ xs$ eine Liste sein. Außerdem sollte auf dem Datentyp eine Induktionsregel gelten. Als Methode zur Charakterisierung der repräsentierenden Menge erscheinen daher induktive Definitionen bzw. kleinste Fixpunkte als angemessen. Die „Gleichung“

$$\mathsf{datatype}\quad \alpha\ \mathsf{list}\ =\ \mathsf{Nil}\ \mid\ \mathsf{Cons}\ \alpha\ (\alpha\ \mathsf{list})$$

auf Typebene wird also zur Fixpunktgleichung

$$\mathsf{list_rep_set} = F(\mathsf{list_rep_set})$$

auf Mengenebene, wobei die monotone Funktion F die rechte Seite der obigen „Typgleichung“ verkörpert.

Soll ein rekursiver Datentyp durch eine Menge vom Typ $\tau\ \mathsf{set}$ repräsentiert werden, so müssen auf τ bestimmte Funktionen definiert sein:

- injektive Funktionen $f_i\ :\ \tau_i \to \tau$ zur Einbettung nichtrekursiver Elemente vom Typ τ_i
- injektive Funktionen $g\ :\ \tau \to \tau$ und $g'\ :\ \tau \to \tau$ mit disjunktem Wertebereich („disjunkte Summe“) zur Modellierung mehrerer verschiedener Konstruktoren
- eine injektive Funktion $p\ :\ \tau \to \tau \to \tau$ zur Darstellung von Konstruktoren mit mehreren Argumenten („Produkt“)

Auf den ersten Blick scheint es naheliegend zu sein, zur Repräsentation die in HOL standardmäßig vorhandenen Typen $\alpha + \beta$ für disjunkte Summen bzw. $\alpha \times \beta$ für das kartesische Produkt mit den injektiven Funktionen

$$\begin{array}{lll} \mathsf{Inl} & : & \alpha \to \alpha + \beta \\ \mathsf{Inr} & : & \beta \to \alpha + \beta \\ (_,_) & : & \alpha \to \beta \to \alpha \times \beta \end{array}$$

zu verwenden, wobei $\mathsf{Inl}\ x \neq \mathsf{Inr}\ y$ gilt. Das Problem liegt jedoch darin, daß der Ergebnistyp der Funktionen Inl, Inr und $(_,_)$ stets strukturell größer als der Typ der Argumente ist. Dies bedeutet im Falle der obigen Fixpunktgleichung, daß auch der Typ von $F(\mathsf{list_rep_set})$ strukturell größer als der Typ von $\mathsf{list_rep_set}$ ist, weshalb die Fixpunktgleichung bei Verwendung der Funktionen Inl, Inr und $(_,_)$ in F nicht wohlgetypt sein kann. Um die Wohlgetyptheit

der Fixpunktgleichung zu gewährleisten, ist daher zur Repräsentation rekursiver Typen ein Typ notwendig, der unter Summen und Produkten abgeschlossen ist. Ein Typ mit derartigen Eigenschaften – im folgenden *Universum* genannt – wurde in [Paulson, 1997] vorgestellt. Die wichtigsten Eigenschaften dieses Typs werden nun im nächsten Abschnitt kurz beschrieben.

4.2.1 Ein Universum für rekursive Typen

Mit dem in [Paulson, 1997] präsentierten und in Form der Isabelle/HOL-Theorie Univ implementierten Typ können Bäume wie z.B. in Abbildung 4.1 in HOL dargestellt werden. Da sich die Elemente rekursiver Datentypen als Bäume auffassen lassen, ist dieser Typ gut zur Repräsentation von Datentypen geeignet.

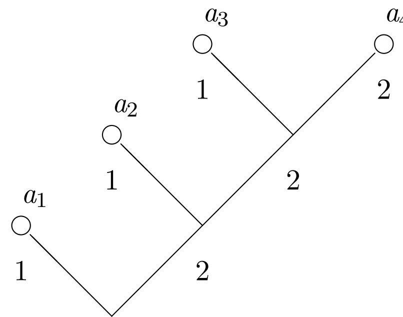


Abbildung 4.1: Endlicher Baum

Derartige Bäume sind im wesentlichen Mengen von Blättern, die über einen bestimmten Pfad erreichbar sind. Ein Baum wird daher durch eine Menge von Paaren dargestellt, wobei jedes Paar ein Blatt des Baums repräsentiert. Die erste Komponente des Paars enthält dabei den zum Blatt führenden Pfad, kodiert als Funktion vom Typ $\mathtt{nat} \to \mathtt{nat}$. Die zweite Komponente enthält den eigentlichen Wert des Blatts. Für den Baum aus Abbildung 4.1 ergibt sich somit die Repräsentation[1]

$$T = \{(f_1, a_1), (f_2, a_2), (f_3, a_3), (f_4, a_4)\}$$

wobei

$$\begin{aligned} f_1 &= \langle 1, 0, 0, \ldots \rangle \\ f_2 &= \langle 2, 1, 0, 0, \ldots \rangle \\ f_3 &= \langle 2, 2, 1, 0, 0, \ldots \rangle \\ f_4 &= \langle 2, 2, 2, 0, 0, \ldots \rangle \end{aligned}$$

Der Wert 0 markiert hierbei jeweils das Ende eines Pfads. Nach diesem Verfahren lassen sich prinzipiell auch Bäume unendlicher Tiefe bzw. mit unendlichem Verzweigungsgrad beschreiben. Da als Labels für Verzweigungen natürliche Zahlen gewählt wurden, kann der Verzweigungsgrad dabei höchstens abzählbar unendlich sein. In Kapitel 5 werden für diese Labels

[1] $f = \langle y_0, y_1, \ldots \rangle$ bedeutet hierbei $f(0) = y_0$, $f(1) = y_1$, ...

beliebige Typen zugelassen. Formal haben Bäume den Typ α item, wobei

$$\begin{array}{rcl} \alpha \text{ node} & = & (\text{nat} \to \text{nat}) \times (\alpha + \text{nat}) \\ \alpha \text{ item} & = & (\alpha \text{ node})\text{set} \end{array}$$

Hierbei ist zu beachten, daß nicht alle Elemente des Typs α item tatsächlich Bäume bzw. Elemente von Datentypen repräsentieren müssen. Dies ist z.B. dann der Fall, wenn α item Blätter mit unendlich langen Zugriffspfaden enthält. Bei der induktiven Definition der repräsentierenden Menge eines Datentyps werden diese störenden Elemente – gelegentlich auch als „Junk“ bezeichnet – jedoch ausgeschlossen.

Zur Erzeugung von Bäumen sind die folgenden Funktionen definiert[2]:

$$\begin{array}{lcl} \text{push} & :: & \text{nat} \to \alpha \text{ node} \to \alpha \text{ node} \\ \text{push } c\ n & \equiv & (\text{nat_case } c\ (\text{fst } n), \text{snd } n) \\ \text{Scons} & :: & \alpha \text{ item} \to \alpha \text{ item} \to \alpha \text{ item} \\ \text{Scons } i_1\ i_2 & \equiv & (\text{push } 1 \text{ `` } i_1) \cup (\text{push } 2 \text{ `` } i_2) \\ \text{Leaf} & :: & \alpha \to \alpha \text{ item} \\ \text{Leaf } a & \equiv & \{(\lambda x.\ 0, \text{Inl } a)\} \\ \text{Numb} & :: & \text{nat} \to \alpha \text{ item} \\ \text{Numb } k & \equiv & \{(\lambda x.\ 0, \text{Inr } k)\} \end{array}$$

Die Funktion push fügt ein Label als Präfix an den Pfad eines Blatts an, d.h.

$$\text{push } c\ (\langle y_0, y_1, \ldots \rangle, a) = (\langle c, y_0, y_1, \ldots \rangle, a)$$

Mit Scons können zwei Bäume zu einem neuen Baum verknüpft werden. Hierbei wird an alle Pfade der Blätter aus der Menge i_1 das Präfix 1, an alle Pfade der Blätter aus der Menge i_2 das Präfix 2 hinzugefügt und die sich so ergebenden Blättermengen vereinigt. Dies ist in Abbildung 4.2 graphisch dargestellt.

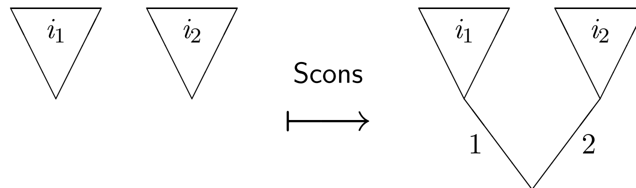


Abbildung 4.2: Verknüpfen zweier Bäume durch Scons

[2] Es gilt

$$f \text{ `` } S = \{y \mid \exists x.\ x \in S \wedge y = f(x)\}$$

$$\begin{array}{lcl} \text{nat_case} & : & \alpha \to (\text{nat} \to \alpha) \to \text{nat} \to \alpha \\ \text{nat_case } f\ g\ 0 & = & f \\ \text{nat_case } f\ g\ (\text{Suc } x) & = & g\ x \end{array}$$

Die Funktionen Leaf und Numb erzeugen jeweils einen Baum der Tiefe 0 mit genau einem Blatt. Mit Hilfe elementarer Eigenschaften von Mengen und Funktionen kann die Injektivität der Funktionen Scons, Leaf und Numb leicht nachgewiesen werden. Darüberhinaus werden noch die Funktionen

$$\begin{array}{lcl} \mathsf{In0} & :: & \alpha\ \mathsf{item} \to \alpha\ \mathsf{item} \\ \mathsf{In0}\ i & \equiv & \mathsf{Scons}\ (\mathsf{Numb}\ 0)\ i \\ \mathsf{In1} & :: & \alpha\ \mathsf{item} \to \alpha\ \mathsf{item} \\ \mathsf{In1}\ i & \equiv & \mathsf{Scons}\ (\mathsf{Numb}\ 1)\ i \end{array}$$

definiert. Diese sind ebenfalls injektiv und es gilt $\mathsf{In0}\ x \neq \mathsf{In1}\ y$, was aus der Injektivität von Scons und Numb sowie $0 \neq 1$ folgt. Mit den Funktionen Leaf, In0, In1 und Scons stehen somit die eingangs geforderten Konstruktionselemente für rekursive Typen zur Verfügung.

4.2.2 Induktive Definition der repräsentierenden Mengen

Mit Hilfe der soeben beschriebenen Funktionen kann somit die den Datentyp α list repräsentierende Menge

$$\mathsf{list_rep_set}\ ::\ (\alpha\ \mathsf{item})\mathsf{set}$$

induktiv durch die beiden Regeln

$$\frac{}{\mathsf{In0\ arbitrary} \in \mathsf{list_rep_set}} \qquad \frac{xs \in \mathsf{list_rep_set}}{\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ x)\ xs) \in \mathsf{list_rep_set}}$$

bzw. als kleinster Fixpunkt durch

$$\begin{array}{rcl} F(L) & = & \{x \mid x = \mathsf{In0\ arbitrary} \vee \exists y :: \alpha, ys.\ x = \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ y)\ ys) \wedge ys \in L\} \\ \mathsf{list_rep_set} & = & \mathsf{lfp}(F) \end{array}$$

definiert werden, wobei F vom Typ $(\alpha\ \mathsf{item})\mathsf{set} \to (\alpha\ \mathsf{item})\mathsf{set}$ ist.

Für den Typ (α, β)term sind die beiden Mengen

$$\begin{array}{lcl} \mathsf{term_rep_set} & :: & ((\alpha + \beta)\mathsf{item})\mathsf{set} \\ \mathsf{term_list_rep_set} & :: & ((\alpha + \beta)\mathsf{item})\mathsf{set} \end{array}$$

induktiv durch die Regeln

$$\frac{}{\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a)) \in \mathsf{term_rep_set}} \qquad \frac{xs \in \mathsf{term_list_rep_set}}{\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ xs) \in \mathsf{term_rep_set}}$$

$$\frac{}{\mathsf{In0\ arbitrary} \in \mathsf{term_list_rep_set}} \qquad \frac{x \in \mathsf{term_rep_set} \quad xs \in \mathsf{term_list_rep_set}}{\mathsf{In1}\ (\mathsf{Scons}\ x\ xs) \in \mathsf{term_list_rep_set}}$$

zu definieren. Aus diesen Einführungsregeln ergibt sich gemäß Abschnitt 2.3.2 die Induktionsregel

$$\frac{\begin{array}{l}\forall a.\ Q_1\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a)))\\ \forall b, ys.\ Q_2\ ys \wedge ys \in \mathsf{term_list_rep_set} \Longrightarrow Q_1\ (\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ ys))\\ Q_2\ (\mathsf{In0}\ \mathsf{arbitrary})\\ \forall y, ys.\ Q_1\ y \wedge y \in \mathsf{term_rep_set} \wedge Q_2\ ys \wedge ys \in \mathsf{term_list_rep_set} \Longrightarrow\\ \quad Q_2\ (\mathsf{In1}\ (\mathsf{Scons}\ y\ ys))\end{array}}{y \in \mathsf{term_rep_set} \Longrightarrow Q_1\ y \wedge ys \in \mathsf{term_rep_set} \Longrightarrow Q_2\ ys}$$

Allgemeines Verfahren Seien $\tau_1, \ldots, \tau_\xi$ alle nichtrekursiven Typen, die in der aufgefalteten Datentypdefinition aus Abschnitt 4.1 vorkommen. Dann sind die repräsentierenden Mengen t_1_rep_set, ..., t_n_rep_set der Datentypen $(\alpha_1, \ldots, \alpha_h)t_1$, ..., $(\alpha_1, \ldots, \alpha_h)t_n$ vom Typ $((\tau_1 + \cdots + \tau_\xi)\mathsf{item})\mathsf{set}$, wobei $\mathrm{TVars}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_h\}$ gilt. Enthält eine Datentypdefinition keine nichtrekursiven Typen, so haben die repräsentierenden Mengen den Typ (unit item)set.

Konstruktion von Injektionen Für die Definition der repräsentierenden Mengen werden die Funktionen

$$\begin{aligned} \mathsf{in}_i &:: \tau_i \to \tau_1 + \cdots + \tau_\xi \\ \underline{\mathsf{in}}_i &:: \alpha\ \mathsf{item} \to \alpha\ \mathsf{item} \end{aligned}$$

mit den Eigenschaften

$$\begin{aligned} i \neq j &\Longrightarrow \mathsf{in}_i\ x \neq \mathsf{in}_j\ y \\ x \neq y &\Longrightarrow \mathsf{in}_i\ x \neq \mathsf{in}_i\ y \end{aligned}$$

$$\begin{aligned} i \neq j &\Longrightarrow \underline{\mathsf{in}}_i\ x \neq \underline{\mathsf{in}}_j\ y \\ x \neq y &\Longrightarrow \underline{\mathsf{in}}_i\ x \neq \underline{\mathsf{in}}_i\ y \end{aligned}$$

verwendet. Die Injektionen in_i werden mit Hilfe der bereits erwähnten Funktionen Inl und Inr konstruiert. Hierbei wird eine relativ effiziente, in [Paulson, 1994] vorgeschlagene Kodierung verwendet. So werden die Injektionen beispielsweise für $\xi = 4$ nicht durch

$$\begin{aligned} \mathsf{in}_1\ x &\equiv \mathsf{Inl}\ x \\ \mathsf{in}_2\ x &\equiv \mathsf{Inr}\ (\mathsf{Inl}\ x) \\ \mathsf{in}_3\ x &\equiv \mathsf{Inr}\ (\mathsf{Inr}\ (\mathsf{Inl}\ x)) \\ \mathsf{in}_4\ x &\equiv \mathsf{Inr}\ (\mathsf{Inr}\ (\mathsf{Inr}\ x)) \end{aligned}$$

sondern durch

$$\begin{aligned} \mathsf{in}_1\ x &\equiv \mathsf{Inl}\ (\mathsf{Inl}\ x) \\ \mathsf{in}_2\ x &\equiv \mathsf{Inl}\ (\mathsf{Inr}\ x) \\ \mathsf{in}_3\ x &\equiv \mathsf{Inr}\ (\mathsf{Inl}\ x) \\ \mathsf{in}_4\ x &\equiv \mathsf{Inr}\ (\mathsf{Inr}\ x) \end{aligned}$$

d.h. durch „balancierte“ Kombination von Inl und Inr dargestellt. Bei ξ Typen beträgt die Schachtelungstiefe der Ausdrücke dabei maximal $\lceil \log_2(\xi) \rceil$ im Gegensatz zu $\xi - 1$ bei „unbalancierter“ Kombination. Dies wirkt sich positiv auf den Platzbedarf und die Laufzeit der Beweise aus. Im obigen Beispiel sind die Funktionen inj_i vom Typ

$$\tau_i \to (\tau_1 + \tau_2) + (\tau_3 + \tau_4)$$

wobei die Klammerung zu beachten ist. Für $\underline{\mathsf{in}}_i$ ist die Konstruktion analog, wobei die Funktionen $\mathsf{In0}$ und $\mathsf{In1}$ anstelle von Inl und Inr verwendet werden.

Einführungsregeln Für die verschränkte induktive Definition der Mengen $t_1\mathsf{_rep_set}, \ldots, t_n\mathsf{_rep_set}$ wird für jeden Konstruktor der beteiligten Datentypen eine Einführungsregel angegeben. Für einen Konstruktor der Form

$$C_i^j \; \ldots \; ((\alpha_1, \ldots, \alpha_h)t_{j'}) \; \ldots \; \tau_{i'} \; \ldots$$

gibt man hierbei eine Einführungsregel der Form

$$\frac{x \in t_{j'}\mathsf{_rep_set} \quad \ldots}{\underline{\mathsf{in}}_i \; (\cdots \bullet x \bullet \cdots \bullet \mathsf{Leaf} \; (\mathsf{in}_{i'} \; x') \bullet \cdots) \in t_j\mathsf{_rep_set}}$$

an, wobei $x_1 \bullet x_2$ die Infix-Schreibweise von $\mathsf{Scons} \; x_1 \; x_2$ ist. Für einen Konstruktor $C_{\tilde{i}}^{\tilde{j}}$ ohne Argumente wird die Einführungsregel

$$\frac{}{\underline{\mathsf{in}}_{\tilde{i}} \; \mathsf{arbitrary} \in t_{\tilde{j}}\mathsf{_rep_set}}$$

angegeben.

Induktionsregel Für die Mengen $t_1\mathsf{_rep_set}, \ldots, t_n\mathsf{_rep_set}$ kann nun gemäß Abschnitt 2.3.2 die simultane Induktionsregel

$$\frac{\begin{array}{c} [P_{j'}(x) \quad x \in t_{j'}\mathsf{_rep_set} \quad \ldots]_{x,x',\ldots} \\ \vdots \\ P_j(\underline{\mathsf{in}}_i \; (\cdots \bullet x \bullet \cdots \bullet \mathsf{Leaf} \; (\mathsf{in}_{i'} \; x') \bullet \cdots)) \quad \ldots \end{array}}{y_1 \in t_1\mathsf{_rep_set} \Longrightarrow P_1(y_1) \wedge \ldots \wedge y_n \in t_n\mathsf{_rep_set} \Longrightarrow P_n(y_n)}$$

bewiesen werden.

4.3 Beweis von Isomorphie-Eigenschaften

Sind die repräsentierenden Mengen konstruiert, so werden die neuen Typen durch den in Abschnitt 2.2.3 beschriebenen Typdefinitionsmechanismus eingeführt. Für den Typ $(\alpha, \beta)\mathsf{term}$ werden hierbei die Abstraktions- und Repräsentationsfunktionen

$$\begin{array}{lcl} \mathsf{Abs_term} & :: & (\alpha + \beta)\mathsf{item} \to (\alpha, \beta)\mathsf{term} \\ \mathsf{Rep_term} & :: & (\alpha, \beta)\mathsf{term} \to (\alpha + \beta)\mathsf{item} \end{array}$$

deklariert und die Axiome

$$\begin{array}{ll} \mathsf{Abs_term}\ (\mathsf{Rep_term}\ x) = x & (\text{Rep_term_inverse}) \\ y \in \mathsf{term_rep_set} \Longrightarrow \mathsf{Rep_term}\ (\mathsf{Abs_term}\ y) = y & (\text{Abs_term_inverse}) \\ \mathsf{Rep_term}\ x \in \mathsf{term_rep_set} & (\text{Rep_term}) \end{array}$$

eingeführt. Damit ist die Isomorphie zwischen der Menge term_rep_set und dem neuen Typ (α, β)term ausgedrückt. Auch zwischen term_list_rep_set und $((\alpha, \beta)$term)list besteht eine derartige Isomorphie. Da list kein neu eingeführter Typkonstruktor ist, ergibt sich diese Isomorphie allerdings nicht direkt aus entsprechenden Axiomen, sondern muß explizit nachgewiesen werden. Unser Ziel ist daher die Definition einer geeigneten Repräsentationsfunktion

$$\mathsf{rep_term_list} \ :: \ ((\alpha, \beta)\mathsf{term})\mathsf{list} \to (\alpha + \beta)\mathsf{item}$$

mit den Eigenschaften

$$\begin{array}{l} \mathsf{inv\ rep_term_list}\ (\mathsf{rep_term_list}\ x) = x \\ y \in \mathsf{term_list_rep_set} \Longrightarrow \mathsf{rep_term_list}\ (\mathsf{inv\ rep_term_list}\ y) = y \\ \mathsf{rep_term_list}\ x \in \mathsf{term_list_rep_set} \end{array}$$

die die gleiche Struktur wie die oben beschriebenen Axiome für term aufweisen. Die zugrundeliegende Idee ist in Abbildung 4.3 veranschaulicht.

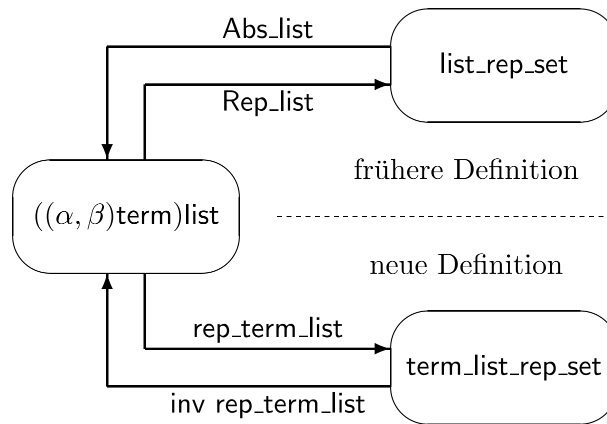


Abbildung 4.3: Isomorphie zwischen term_list_rep_set und $((\alpha, \beta)$term)list

Die ersten beiden Eigenschaften erhält man hierbei mit Hilfe von Standardtheoremen über die Umkehrbarkeit von Funktionen

$$\begin{array}{llll} \mathsf{inj}\ f & \Longrightarrow & \mathsf{inv}\ f\ (f\ \ x) = x & (\text{inv_f_f}) \\ y \in \mathsf{range}\ f & \Longrightarrow & f\ (\mathsf{inv}\ f\ \ y) = y & (\text{f_inv_f}) \end{array}$$

wofür die Injektivität von rep_term_list sowie

$$y \in \mathsf{term_list_rep_set} \Longrightarrow y \in \mathsf{range\ rep_term_list}$$

zu zeigen ist. Letzteres besagt, daß für jedes Element y aus der repräsentierenden Menge term_list_rep_set eine Liste x mit $y =$ rep_term_list x existieren muß, d.h. rep_term_list ist surjektiv. Die Definition von Repräsentationsfunktionen sowie die Ableitung der entsprechenden Eigenschaften wird nun in den folgenden Abschnitten sowohl am Beispiel der Funktion rep_term_list als auch allgemein beschrieben.

4.3.1 Definition von Repräsentationsfunktionen

Bei der Definition von rep_term_list kann man sich die Tatsache zunutze machen, daß für list bereits ein Kombinator list_rec für primitive Rekursion definiert wurde. Die primitiv rekursive Funktion rep_term_list läßt sich durch die Gleichungen

$$\begin{array}{lcl} \mathsf{rep_term_list\ Nil} & = & \mathsf{In0\ arbitrary} \\ \mathsf{rep_term_list\ (Cons}\ x\ xs) & = & \mathsf{In1\ (Scons\ (Rep_term}\ x)\ (\mathsf{rep_term_list}\ xs)) \end{array}$$

charakterisieren. Im Fall des Konstruktors Cons erhält man hierbei die Repräsentation des Arguments x vom Typ (α, β)term durch die bei der Typdefinition eingeführte Funktion Rep_term, während die Repräsentation des Arguments xs vom Typ $((\alpha, \beta)$term)list durch einen rekursiven Aufruf von rep_term_list ermittelt wird. Wir definieren nun rep_term_list mit Hilfe von list_rec durch

$$\mathsf{rep_term_list} \equiv \mathsf{list_rec\ (In0\ arbitrary)}\ (\lambda x, xs, ys.\ \mathsf{In1\ (Scons\ (Rep_term}\ x)\ ys))$$

woraus sich die obigen Gleichungen mit Hilfe der charakteristischen Gleichungen für list_rec leicht ableiten lassen.

Bei größerer Verschachtelungstiefe sind mehrere Repräsentationsfunktionen erforderlich. So werden für den Typ term2 die Funktionen

$$\begin{array}{lcl} \mathsf{rep_term2_list} & :: & ((\alpha, \beta)\mathsf{term2})\mathsf{list} \rightarrow (\alpha + \beta)\mathsf{item} \\ \mathsf{rep_term2_list_list} & :: & (((\alpha, \beta)\mathsf{term2})\mathsf{list})\mathsf{list} \rightarrow (\alpha + \beta)\mathsf{item} \end{array}$$

mit den charakteristischen Gleichungen

$$\begin{array}{lcl} \mathsf{rep_term2_list\ Nil} & = & \mathsf{In0\ arbitrary} \\ \mathsf{rep_term2_list\ (Cons}\ x\ xs) & = & \mathsf{In1\ (Scons\ (Rep_term2}\ x)\ (\mathsf{rep_term2_list}\ xs)) \\ \mathsf{rep_term2_list_list\ Nil} & = & \mathsf{In0\ arbitrary} \\ \mathsf{rep_term2_list_list\ (Cons}\ x\ xs) & = & \mathsf{In1\ (Scons\ (rep_term2_list}\ x)\ (\mathsf{rep_term2_list_list}\ xs)) \end{array}$$

benötigt, die sich durch

$$\begin{array}{lcll} \mathsf{rep_term2_list} & \equiv & \mathsf{list_rec} & (\mathsf{In0\ arbitrary}) \\ & & & (\lambda x, xs, ys.\ \mathsf{In1\ (Scons\ (Rep_term2}\ x)\ ys) \\ \mathsf{rep_term2_list_list} & \equiv & \mathsf{list_rec} & (\mathsf{In0\ arbitrary}) \\ & & & (\lambda x, xs, ys.\ \mathsf{In1\ (Scons\ (rep_term2_list}\ x)\ ys) \end{array}$$

definieren lassen. Man beachte, daß hierbei in der Definition der Funktion $\mathsf{rep_term2_list}$ nur auf $\mathsf{Rep_term2}$ und in der Definition von $\mathsf{rep_term2_list_list}$ nur auf $\mathsf{rep_term2_list}$ zurückgegriffen werden muß.

Allgemeines Verfahren Für die in Abschnitt 4.1 angegebene Datentypdefinition werden Funktionen benötigt, mit denen Elemente der bereits existierenden, verschachtelt rekursiv vorkommenden Typen

$$\left(\hat{\tau}_1^j, \ldots, \hat{\tau}_{\tilde{h}_1}^j\right) \tilde{t}_j \qquad \text{mit } \nu_1 \leq j \leq n$$

in Elemente der Menge $t_j\mathsf{_rep_set}$ und umgekehrt konvertiert werden können. Mittels primitiver Rekursion definieren wir daher für $1 \leq \mu \leq n'$ jeweils simultan für die Typen

$$\left(\hat{\tau}_1^{\nu_\mu}, \ldots, \hat{\tau}_{\tilde{h}_\mu}^{\nu_\mu}\right) \tilde{t}_{\nu_\mu} \quad \ldots \quad \left(\hat{\tau}_1^{\nu_{\mu+1}-1}, \ldots, \hat{\tau}_{\tilde{h}_\mu}^{\nu_{\mu+1}-1}\right) \tilde{t}_{\nu_{\mu+1}-1}$$

die verschränkt rekursiven Funktionen

$$\begin{array}{rcl}
\mathsf{rep}_{\nu_\mu} & :: & \left(\hat{\tau}_1^{\nu_\mu}, \ldots, \hat{\tau}_{\tilde{h}_\mu}^{\nu_\mu}\right) \tilde{t}_{\nu_\mu} \rightarrow (\tau_1 + \cdots + \tau_\xi)\mathsf{item} \\
& \vdots & \\
\mathsf{rep}_{\nu_{\mu+1}-1} & :: & \left(\hat{\tau}_1^{\nu_{\mu+1}-1}, \ldots, \hat{\tau}_{\tilde{h}_\mu}^{\nu_{\mu+1}-1}\right) \tilde{t}_{\nu_{\mu+1}-1} \rightarrow (\tau_1 + \cdots + \tau_\xi)\mathsf{item}
\end{array}$$

Mit Hilfe der bereits definierten Kombinatoren $\tilde{t}_{\nu_\mu}\mathsf{_rec}, \ldots, \tilde{t}_{\nu_{\mu+1}-1}\mathsf{_rec}$ für primitive Rekursion lassen sich diese Funktionen durch

$$\begin{array}{rcl}
\mathsf{rep}_{\nu_\mu} & \equiv & \tilde{t}_{\nu_\mu}\mathsf{_rec}\ f_1^{\nu_\mu} \ \ldots\ f_{k_{\nu_\mu}}^{\nu_\mu} \ \ldots\ f_1^{\nu_{\mu+1}-1} \ \ldots\ f_{k_{\nu_{\mu+1}-1}}^{\nu_{\mu+1}-1} \\
& \vdots & \\
\mathsf{rep}_{\nu_{\mu+1}-1} & \equiv & \tilde{t}_{\nu_{\mu+1}-1}\mathsf{_rec}\ f_1^{\nu_\mu} \ \ldots\ f_{k_{\nu_\mu}}^{\nu_\mu} \ \ldots\ f_1^{\nu_{\mu+1}-1} \ \ldots\ f_{k_{\nu_{\mu+1}-1}}^{\nu_{\mu+1}-1}
\end{array}$$

ausdrücken, wobei jede Funktion f_i^j mit einem Konstruktor C_i^j korrespondiert. Für einen Konstruktor der Form

$$C_i^j \ \ldots\ (\alpha_1, \ldots, \alpha_h)t_{j'} \ \ldots\ (\alpha_1, \ldots, \alpha_h)t_{j''} \ \ldots\ (\alpha_1, \ldots, \alpha_h)t_{j'''} \ \ldots\ \tau_{i'} \ \ldots$$

mit $\nu_\mu \leq j' < \nu_{\mu+1}$, $\nu_{\mu+1} \leq j''$ und $1 \leq j''' < \nu_1$ hat f_i^j hierbei die Gestalt

$$\begin{array}{l}
\lambda x_1, \ldots, x', \ldots, x'', \ldots, x''', \ldots, x_{m_i^j}, y_1, \ldots, y, \ldots, y_{l_i^j}. \\
\quad \underline{\mathsf{in}}_i\ (\cdots \bullet y \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \cdots \bullet \mathsf{Rep_}t_{j'''}\ x'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots)
\end{array}$$

Aus der Definition von rep_j läßt sich für den Konstruktor C_i^j dann die charakteristische Gleichung

$$\begin{array}{l}
\mathsf{rep}_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots) = \\
\quad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{rep}_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \cdots \bullet \mathsf{Rep_}t_{j'''}\ x'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots)
\end{array}$$

ableiten. Man beachte, daß bei der Definition von rep_{ν_μ}, ..., $\mathsf{rep}_{\nu_{\mu+1}-1}$ stets nur auf rep_j mit $\nu_{\mu+1} \leq j$ oder $\mathsf{Rep_}t_j$ mit $1 \leq j < \nu_1$ zurückgegriffen wird. Bei der folgenden allgemeinen Erläuterung der Beweise von Eigenschaften dieser Repräsentationsfunktionen wird jeweils exemplarisch die Behandlung des Konstruktors C_i^j beschrieben.

4.3.2 Surjektivität der Repräsentationsfunktionen

Wie eingangs erwähnt, muß für rep_term_list die Eigenschaft

$$y \in \mathsf{term_list_rep_set} \Longrightarrow y \in \mathsf{range\ rep_term_list}$$

nachgewiesen werden. Dies geschieht mittels simultaner Induktion unter Verwendung der Induktionsregel für die Mengen term_rep_set und term_list_rep_set. Wir gehen hierbei vom Beweisziel

$$y' \in \mathsf{term_rep_set} \Longrightarrow \mathsf{True} \wedge y \in \mathsf{term_list_rep_set} \Longrightarrow y \in \mathsf{range\ rep_term_list}$$

aus. Da für term keine Repräsentationsfunktion definiert werden muß und daher für die Menge term_rep_set nichts zu zeigen ist, hat das erste Glied der Konjunktion die Form $\ldots \Longrightarrow \mathsf{True}$. Durch Anwendung der Induktionsregel erhalten wir die Beweisziele

(1) $\ldots \Longrightarrow \mathsf{True}$

(2) $\ldots \Longrightarrow \mathsf{True}$

(3) $\mathsf{In0\ arbitrary} \in \mathsf{range\ rep_term_list}$

(4) $\forall y, ys.\ y \in \mathsf{term_rep_set} \wedge ys \in \mathsf{term_list_rep_set} \wedge ys \in \mathsf{range\ rep_term_list} \Longrightarrow$
$\mathsf{In1}\ (\mathsf{Scons}\ y\ ys) \in \mathsf{range\ rep_term_list}$

Die Beweisziele (1) und (2) sind trivial. Aus Beweisziel (3) erhält man mittels (range_eqI) das neue Beweisziel

$$\mathsf{rep_term_list\ Nil} = \mathsf{In0\ arbitrary}$$

was sich sofort mit Hilfe der charakteristischen Gleichungen für rep_term_list lösen läßt. Durch Anwendung von (rangeE) erhält man aus Beweisziel (4) das neue Beweisziel

$$\forall y, ys, xs.\ y \in \mathsf{term_rep_set} \wedge ys \in \mathsf{term_list_rep_set} \wedge ys = \mathsf{rep_term_list}\ xs \Longrightarrow$$
$$\mathsf{In1}\ (\mathsf{Scons}\ y\ ys) \in \mathsf{range\ rep_term_list}$$

Unter Ausnutzung der Prämisse $ys = \mathsf{rep_term_list}\ xs$ erhält man hieraus das Beweisziel

$$\forall y, ys, xs.\ y \in \mathsf{term_rep_set} \wedge ys \in \mathsf{term_list_rep_set} \Longrightarrow$$
$$\mathsf{In1}\ (\mathsf{Scons}\ y\ (\mathsf{rep_term_list}\ xs)) \in \mathsf{range\ rep_term_list}$$

Mit Hilfe der Prämisse $y \in \mathsf{term_rep_set}$ und dem Axiom (Abs_term_inverse) ergibt sich

$$\forall y, ys, xs.\ ys \in \mathsf{term_list_rep_set} \Longrightarrow$$
$$\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ (\mathsf{Abs_term}\ y))\ (\mathsf{rep_term_list}\ xs)) \in \mathsf{range\ rep_term_list}$$

woraus man mittels (range_eqI) das Beweisziel

$$\begin{array}{l}\forall y, ys, xs.\ ys \in \mathsf{term_list_rep_set} \Longrightarrow \\ \quad \mathsf{rep_term_list}\ (\mathsf{Cons}\ (\mathsf{Abs_term}\ y)\ xs) = \\ \quad \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ (\mathsf{Abs_term}\ y))\ (\mathsf{rep_term_list}\ xs))\end{array}$$

erhält. Dies folgt ebenso sofort aus den charakteristischen Gleichungen für rep_term_list.

Allgemeines Verfahren Wir zeigen

$$y_j \in t_j\mathsf{_rep_set} \Longrightarrow y_j \in \mathsf{range\ rep}_j$$

für $\nu_1 \leq j \leq n$. Der Beweis erfolgt unter Verwendung der simultanen Induktionsregel für die Mengen t_1_rep_set, ..., t_n_rep_set. Wir gehen hierbei vom Beweisziel

$$\begin{array}{l}y_1 \in t_1\mathsf{_rep_set} \Longrightarrow \mathsf{True} \wedge \ldots \wedge y_{\nu_1 - 1} \in t_{\nu_1 - 1}\mathsf{_rep_set} \Longrightarrow \mathsf{True} \wedge \\ y_{\nu_1} \in t_{\nu_1}\mathsf{_rep_set} \Longrightarrow y_{\nu_1} \in \mathsf{range\ rep}_{\nu_1} \wedge \ldots \wedge y_n \in t_n\mathsf{_rep_set} \Longrightarrow y_n \in \mathsf{range\ rep}_n\end{array}$$

aus. Für die Mengen t_1_rep_set, ..., $t_{\nu_1 - 1}$_rep_set ist nichts zu zeigen, weshalb die ersten Glieder der Konjunktion die Gestalt $\ldots \Longrightarrow$ True haben. Die Anwendung der Induktionsregel ergibt nun die neuen Beweisziele

$$\begin{array}{ll} & \ldots \Longrightarrow \mathsf{True} \\ & \quad \vdots \\ (\star) & \forall x, x', x'', x''', \ldots .\ x \in \mathsf{range\ rep}_{j'} \wedge x' \in \mathsf{range\ rep}_{j''} \wedge x'' \in t_{j'''}\mathsf{_rep_set} \wedge \ldots \Longrightarrow \\ & \quad \underline{\mathsf{in}}_i\ (\cdots \bullet x \bullet \cdots \bullet x' \bullet \cdots \bullet x'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots) \in \mathsf{range\ rep}_j \\ & \quad \vdots \end{array}$$

Die Beweisziele der Form $\ldots \Longrightarrow$ True sind trivial. Aus dem Beweisziel ($\star$) erhält man durch mehrmalige Anwendung von (rangeE) das Beweisziel

$$\begin{array}{l}\forall x, x', x'', x''', z, z', \ldots .\ x = \mathsf{rep}_{j'}\ z \wedge x' = \mathsf{rep}_{j''}\ z' \wedge x'' \in t_{j'''}\mathsf{_rep_set} \wedge \ldots \Longrightarrow \\ \quad \underline{\mathsf{in}}_i\ (\cdots \bullet x \bullet \cdots \bullet x' \bullet \cdots \bullet x'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots) \in \mathsf{range\ rep}_j\end{array}$$

woraus man unter Ausnutzung der Prämissen $x = \mathsf{rep}_{j'}\ z$ und $x' = \mathsf{rep}_{j''}\ z'$ das Beweisziel

$$\begin{array}{l}\forall x, x', x'', x''', z, z', \ldots .\ x'' \in t_{j'''}\mathsf{_rep_set} \wedge \ldots \Longrightarrow \\ \quad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{rep}_{j'}\ z \bullet \cdots \bullet \mathsf{rep}_{j''}\ z' \bullet \cdots \bullet x'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots) \in \mathsf{range\ rep}_j\end{array}$$

erhält. Unter Verwendung der Prämisse $x'' \in t_{j'''}$_rep_set und des bei der Typdefinition eingeführten Axioms

$$x'' \in t_{j'''}\mathsf{_rep_set} \Longrightarrow \mathsf{Rep_}t_{j'''}\ (\mathsf{Abs_}t_{j'''}\ x'') = x'' \qquad (\mathrm{Abs_}t_{j'''}\mathrm{_inverse})$$

ergibt sich hieraus das Beweisziel

$$\begin{array}{l}\forall x, x', x'', x''', z, z', \ldots .\ \ldots \Longrightarrow \\ \quad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{rep}_{j'}\ z \bullet \cdots \bullet \mathsf{rep}_{j''}\ z' \bullet \\ \qquad \cdots \bullet \mathsf{Rep_}t_{j'''}\ (\mathsf{Abs_}t_{j'''}\ x'') \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots) \in \mathsf{range\ rep}_j\end{array}$$

mittels (range_eqI) erhält man schließlich

$$\begin{array}{l}\forall x, x', x'', x''', z, z', \ldots .\ \ldots \Longrightarrow \\ \quad \mathsf{rep}_j\ (C_i^j\ \ldots\ z\ \ldots\ z'\ \ldots\ (\mathsf{Abs_}t_{j'''}\ x'')\ \ldots\ x'''\ \ldots) = \\ \qquad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{rep}_{j'}\ z \bullet \cdots \bullet \mathsf{rep}_{j''}\ z' \bullet \\ \qquad\quad \cdots \bullet \mathsf{Rep_}t_{j'''}\ (\mathsf{Abs_}t_{j'''}\ x'') \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots)\end{array}$$

was sich mit Hilfe der charakteristischen Gleichungen für rep_j lösen läßt.

4.3.3 Injektivität der Repräsentationsfunktionen

Der Nachweis der Injektivität von rep_term_list erfolgt mit Hilfe der Induktionsregel für list. Hierbei gehen wir vom Beweisziel

$$\forall ys'.\ \mathsf{rep_term_list}\ ys = \mathsf{rep_term_list}\ ys' \Longrightarrow ys = ys'$$

aus. Induktion über ys liefert die Beweisziele

$$\begin{array}{ll}(1) & \forall ys'.\ \mathsf{rep_term_list}\ \mathsf{Nil} = \mathsf{rep_term_list}\ ys' \Longrightarrow \mathsf{Nil} = ys' \\ (2) & \forall x, xs.\ (\forall ys''.\ \mathsf{rep_term_list}\ xs = \mathsf{rep_term_list}\ ys'' \Longrightarrow xs = ys'') \Longrightarrow \\ & \quad \forall ys'.\ \mathsf{rep_term_list}\ (\mathsf{Cons}\ x\ xs) = \mathsf{rep_term_list}\ ys' \Longrightarrow \mathsf{Cons}\ x\ xs = ys'\end{array}$$

Mittels Fallunterscheidung über ys' erhält man aus (1) die neuen Beweisziele

$$\begin{array}{ll}(3) & \mathsf{rep_term_list}\ \mathsf{Nil} = \mathsf{rep_term_list}\ \mathsf{Nil} \Longrightarrow \mathsf{Nil} = \mathsf{Nil} \\ (4) & \forall z, zs.\ \mathsf{rep_term_list}\ \mathsf{Nil} = \mathsf{rep_term_list}\ (\mathsf{Cons}\ z\ zs) \Longrightarrow \mathsf{Nil} = \mathsf{Cons}\ z\ zs\end{array}$$

Hierbei läßt sich (3) unmittelbar durch Reflexivität lösen. Aus (4) erhält man mit Hilfe der charakteristischen Gleichungen für rep_term_list das Beweisziel

$$\forall z, zs.\ \mathsf{In0}\ \mathsf{arbitrary} = \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ z)\ (\mathsf{rep_term_list}\ zs)) \Longrightarrow \mathsf{Nil} = \mathsf{Cons}\ z\ zs$$

was wegen $\mathsf{In0}\ \ldots \neq \mathsf{In1}\ \ldots$ trivial lösbar ist. Ebenfalls mittels Fallunterscheidung über ys' erhält man aus (2) die Beweisziele

$$\begin{array}{l}\forall x, xs.\ (\forall ys''.\ \mathsf{rep_term_list}\ xs = \mathsf{rep_term_list}\ ys'' \Longrightarrow xs = ys'') \wedge \\ \quad \mathsf{rep_term_list}\ (\mathsf{Cons}\ x\ xs) = \mathsf{rep_term_list}\ \mathsf{Nil} \Longrightarrow \\ \qquad \mathsf{Cons}\ x\ xs = \mathsf{Nil}\end{array}$$

$$\begin{array}{l}\forall x, xs, z, zs.\ (\forall ys''.\ \mathsf{rep_term_list}\ xs = \mathsf{rep_term_list}\ ys'' \Longrightarrow xs = ys'') \wedge \\ \quad \mathsf{rep_term_list}\ (\mathsf{Cons}\ x\ xs) = \mathsf{rep_term_list}\ (\mathsf{Cons}\ z\ zs) \Longrightarrow \\ \qquad \mathsf{Cons}\ x\ xs = \mathsf{Cons}\ z\ zs\end{array}$$

Mit Hilfe der charakteristischen Gleichungen für rep_term_list erhält man hieraus die Beweisziele

$$
\begin{array}{ll}
(5) & \forall x, xs.\ (\forall ys''.\ \mathsf{rep_term_list}\ xs = \mathsf{rep_term_list}\ ys'' \Longrightarrow xs = ys'') \wedge \\
& \quad \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ x)\ (\mathsf{rep_term_list}\ xs)) = \mathsf{In0}\ \mathsf{arbitrary} \Longrightarrow \\
& \qquad \mathsf{Cons}\ x\ xs = \mathsf{Nil} \\
(6) & \forall x, xs, z, zs.\ (\forall ys''.\ \mathsf{rep_term_list}\ xs = \mathsf{rep_term_list}\ ys'' \Longrightarrow xs = ys'') \wedge \\
& \quad \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ x)\ (\mathsf{rep_term_list}\ xs)) = \\
& \quad \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ z)\ (\mathsf{rep_term_list}\ zs)) \Longrightarrow \\
& \qquad \mathsf{Cons}\ x\ xs = \mathsf{Cons}\ z\ zs
\end{array}
$$

Beweisziel (5) ist wegen $\mathsf{In1}\ \ldots \neq \mathsf{In0}\ \ldots$ wieder trivial lösbar. Aus der Prämisse

$$\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ x)\ (\mathsf{rep_term_list}\ xs)) = \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ z)\ (\mathsf{rep_term_list}\ zs))$$

von Beweisziel (6) erhält man aufgrund der Injektivität von In1 und Scons, sowie der aus dem Axiom (Rep_term_inverse) folgerbaren Injektivität von Rep_term die Gleichungen $x = z$ sowie $\mathsf{rep_term_list}\ xs = \mathsf{rep_term_list}\ zs$, woraus sich mit Hilfe der Prämisse

$$\forall ys''.\ \mathsf{rep_term_list}\ xs = \mathsf{rep_term_list}\ ys'' \Longrightarrow xs = ys''$$

schließlich noch $xs = zs$ folgern läßt. Damit ist auch Beweisziel (6) gelöst.

Bei verschachtelt rekursiven Datentypdefinitionen mit größerer Verschachtelungstiefe müssen die Eigenschaften der Repräsentationsfunktionen für die zuletzt aufgefalteten, also in der Verschachtelung am weitesten innen liegenden Typen als erstes gezeigt werden. So muß z.B. zuerst die Injektivität von rep_term2_list und dann die Injektivität von rep_term2_list_list bewiesen werden, da für letzteres die Injektivität von rep_term2_list benötigt wird.

Allgemeines Verfahren Wir zeigen die Injektivität der Funktionen rep_j für $\nu_1 \leq j \leq n$. Für alle $1 \leq \mu \leq n'$ wird dies jeweils mit simultaner Induktion für die Funktionen rep_{ν_μ}, ..., $\mathsf{rep}_{\nu_{\mu+1}-1}$ gezeigt. Da hierbei außer der Injektivität von $\mathsf{Rep_}t_j$ für $1 \leq j < \nu_1$ auch die Injektivität von rep_j für $\nu_{\mu+1} \leq j \leq n$ benötigt wird, muß mit $\mu = n'$ begonnen werden.

Wir gehen vom Beweisziel

$$
\begin{array}{l}
\forall y'.\ \mathsf{rep}_{\nu_\mu}\ y_{\nu_\mu} = \mathsf{rep}_{\nu_\mu}\ y' \Longrightarrow y_{\nu_\mu} = y' \wedge \ldots \wedge \\
\forall y'.\ \mathsf{rep}_j\ y_j = \mathsf{rep}_j\ y' \Longrightarrow y_j = y' \wedge \ldots \wedge \\
\forall y'.\ \mathsf{rep}_{\nu_{\mu+1}-1}\ y_{\nu_{\mu+1}-1} = \mathsf{rep}_{\nu_{\mu+1}-1}\ y' \Longrightarrow y_{\nu_{\mu+1}-1} = y'
\end{array}
$$

aus. Simultane Induktion über y_{ν_μ}, ..., $y_{\nu_{\mu+1}-1}$ liefert die Beweisziele

$$\begin{array}{ll}
& \ldots \Longrightarrow \forall y'.\ \mathsf{rep}_{\nu_\mu}\ (C_1^{\nu_\mu}\ \ldots) = \mathsf{rep}_{\nu_\mu}\ y' \Longrightarrow C_1^{\nu_\mu}\ \ldots = y' \\
& \vdots \\
& \ldots \Longrightarrow \forall y'.\ \mathsf{rep}_{\nu_\mu}\ \left(C_{k_{\nu_\mu}}^{\nu_\mu}\ \ldots\right) = \mathsf{rep}_{\nu_\mu}\ y' \Longrightarrow C_{k_{\nu_\mu}}^{\nu_\mu}\ \ldots = y' \\
& \vdots \\
(\star) & \forall x, x', x'', x''', \ldots.\ (\forall y''.\ \mathsf{rep}_{j'}\ x = \mathsf{rep}_{j'}\ y'' \Longrightarrow x = y'') \wedge \ldots \Longrightarrow \\
& \quad \forall y'.\ \mathsf{rep}_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots) = \mathsf{rep}_j\ y' \Longrightarrow \\
& \qquad C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots = y' \\
& \vdots \\
& \ldots \Longrightarrow \forall y'.\ \mathsf{rep}_{\nu_{\mu+1}-1}\ \left(C_1^{\nu_{\mu+1}-1}\ \ldots\right) = \mathsf{rep}_{\nu_{\mu+1}-1}\ y' \Longrightarrow C_1^{\nu_{\mu+1}-1}\ \ldots = y' \\
& \vdots \\
& \ldots \Longrightarrow \forall y'.\ \mathsf{rep}_{\nu_{\mu+1}-1}\ \left(C_{k_{\nu_{\mu+1}-1}}^{\nu_{\mu+1}-1}\ \ldots\right) = \mathsf{rep}_{\nu_{\mu+1}-1}\ y' \Longrightarrow C_{k_{\nu_{\mu+1}-1}}^{\nu_{\mu+1}-1}\ \ldots = y'
\end{array}$$

Mittels Fallunterscheidung über y' erhält man aus dem Beweisziel $(\star)$ die neuen Beweisziele

$$\begin{array}{l}
\forall \ldots.\ \mathsf{rep}_j\ (C_i^j\ \ldots) = \mathsf{rep}_j\ (C_1^j\ \ldots) \Longrightarrow \ldots \\
\vdots \\
\forall x, x', x'', x''', \hat{x}, \hat{x}', \hat{x}'', \hat{x}''', \ldots.\ (\forall y''.\ \mathsf{rep}_{j'}\ x = \mathsf{rep}_{j'}\ y'' \Longrightarrow x = y'') \wedge \\
\quad \mathsf{rep}_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots) = \\
\quad \mathsf{rep}_j\ (C_i^j\ \ldots\ \hat{x}\ \ldots\ \hat{x}'\ \ldots\ \hat{x}''\ \ldots\ \hat{x}'''\ \ldots) \wedge \ldots \Longrightarrow \\
\qquad C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots = C_i^j\ \ldots\ \hat{x}\ \ldots\ \hat{x}'\ \ldots\ \hat{x}''\ \ldots\ \hat{x}'''\ \ldots \\
\vdots \\
\forall \ldots.\ \mathsf{rep}_j\ (C_i^j\ \ldots) = \mathsf{rep}_j\ (C_{k_j}^j\ \ldots) \Longrightarrow \ldots
\end{array}$$

woraus sich nach Anwendung der charakteristischen Gleichungen für rep_j die Beweisziele

$$\begin{array}{ll}
& \forall \ldots.\ \underline{\mathsf{in}}_i\ \ldots = \underline{\mathsf{in}}_1\ \ldots \Longrightarrow \ldots \\
& \vdots \\
(\star\star) & \forall x, x', x'', x''', \hat{x}, \hat{x}', \hat{x}'', \hat{x}''', \ldots.\ (\forall y''.\ \mathsf{rep}_{j'}\ x = \mathsf{rep}_{j'}\ y'' \Longrightarrow x = y'') \wedge \\
& \quad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{rep}_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \\
& \qquad \cdots \bullet \mathsf{Rep_}t_{j'''}\ x'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots) = \\
& \quad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{rep}_{j'}\ \hat{x} \bullet \cdots \bullet \mathsf{rep}_{j''}\ \hat{x}' \bullet \\
& \qquad \cdots \bullet \mathsf{Rep_}t_{j'''}\ \hat{x}'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ \hat{x}''') \bullet \cdots) \wedge \ldots \Longrightarrow \\
& \qquad\quad C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots = \\
& \qquad\quad C_i^j\ \ldots\ \hat{x}\ \ldots\ \hat{x}'\ \ldots\ \hat{x}''\ \ldots\ \hat{x}'''\ \ldots \\
& \vdots \\
& \forall \ldots.\ \underline{\mathsf{in}}_i\ \ldots = \underline{\mathsf{in}}_{k_j}\ \ldots \Longrightarrow \ldots
\end{array}$$

ergeben. Alle Beweisziele bis auf $(\star\star)$ sind wegen der Verschiedenheit der Injektionen trivial lösbar. Aufgrund der Injektivität von $\underline{\mathsf{in}}_i$, $\mathsf{in}_{i'}$, $_ \bullet _$ bzw. Scons und Leaf, der Prämisse bzw.

Induktionsvoraussetzung $\forall y''.\ \mathsf{rep}_{j'}\ x = \mathsf{rep}_{j'}\ y'' \Longrightarrow x = y''$, der wegen $\nu_{\mu+1} \leq j''$ bereits bewiesenen Injektivität von $\mathsf{rep}_{j''}$ sowie der aus dem Axiom (Rep_$t_{j'''}$_inverse) folgerbaren Injektivität von $\mathsf{Rep_}t_{j'''}$ erhält man aus ($\star\star$) schließlich das Beweisziel

$$\begin{array}{l} \forall x, x', x'', x''', \hat{x}, \hat{x}', \hat{x}'', \hat{x}''', \ldots .\ x = \hat{x} \wedge x' = \hat{x}' \wedge x'' = \hat{x}'' \wedge x''' = \hat{x}''' \wedge \ldots \Longrightarrow \\ \quad C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots = \\ \quad C_i^j\ \ldots\ \hat{x}\ \ldots\ \hat{x}'\ \ldots\ \hat{x}''\ \ldots\ \hat{x}'''\ \ldots \end{array}$$

was sich nach Verwendung der Prämissen $x = \hat{x}, \ldots$ mittels Reflexivität lösen läßt.

4.3.4 Wertebereich der Repräsentationsfunktionen

Es muß nun noch nachgewiesen werden, daß die Funktion rep_term_list für alle Argumente stets ein Element aus term_list_rep_set liefert. Der Beweis erfolgt ebenfalls wieder mit Hilfe der Induktionsregel für list, wobei wir vom Beweisziel

$$\mathsf{rep_term_list}\ y \in \mathsf{term_list_rep_set}$$

ausgehen. Induktion über y liefert die Beweisziele

$$\mathsf{rep_term_list\ Nil} \in \mathsf{term_list_rep_set}$$

$$\begin{array}{l} \forall x, xs.\ \mathsf{rep_term_list}\ xs \in \mathsf{term_list_rep_set} \Longrightarrow \\ \quad \mathsf{rep_term_list}\ (\mathsf{Cons}\ x\ xs) \in \mathsf{term_list_rep_set} \end{array}$$

Durch Anwendung der charakteristischen Gleichungen für die Funktion rep_term_list erhält man

(1) $\mathsf{In0\ arbitrary} \in \mathsf{term_list_rep_set}$

(2) $\forall x, xs.\ \mathsf{rep_term_list}\ xs \in \mathsf{term_list_rep_set} \Longrightarrow$
$\quad \mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Rep_term}\ x)\ (\mathsf{rep_term_list}\ xs)) \in \mathsf{term_list_rep_set}$

Beweisziel (1) folgt hierbei sofort aus der entsprechenden Einführungsregel für die Menge term_list_rep_set. Ebenso kann Beweisziel (2) unter Verwendung der Prämisse rep_term_list $xs \in$ term_list_rep_set sowie dem Axiom (Rep_term) mit Hilfe einer geeigneten Einführungsregel gelöst werden.

Allgemeines Verfahren Wir zeigen $\mathsf{rep}_j\ y_j \in t_j\mathsf{_rep_set}$ für $\nu_1 \leq j \leq n$, wobei analog zum Beweis der Injektivität simultane Induktion verwendet wird. Wir gehen hierbei vom Beweisziel

$$\mathsf{rep}_{\nu_\mu}\ y_{\nu_\mu} \in t_{\nu_\mu}\mathsf{_rep_set} \wedge \ldots \wedge \mathsf{rep}_{\nu_{\mu+1}-1}\ y_{\nu_{\mu+1}-1} \in t_{\nu_{\mu+1}-1}\mathsf{_rep_set}$$

aus. Simultane Induktion über $y_{\nu_\mu}, \ldots, y_{\nu_{\mu+1}-1}$ liefert die Beweisziele

$$
\begin{array}{ll}
& \ldots \Longrightarrow \mathsf{rep}_{\nu_\mu}\ (C_1^{\nu_\mu}\ \ldots) \in t_{\nu_\mu}_\mathsf{rep_set} \\
& \vdots \\
& \ldots \Longrightarrow \mathsf{rep}_{\nu_\mu}\ \left(C_{k_{\nu_\mu}}^{\nu_\mu}\ \ldots\right) \in t_{\nu_\mu}_\mathsf{rep_set} \\
& \vdots \\
(\star) & \forall x, x', x'', x''', \ldots .\ \mathsf{rep}_{j'}\ x \in t_{j'}_\mathsf{rep_set} \wedge \ldots \Longrightarrow \\
& \quad \mathsf{rep}_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots\ x'''\ \ldots) \in t_j_\mathsf{rep_set} \\
& \vdots \\
& \ldots \Longrightarrow \mathsf{rep}_{\nu_{\mu+1}-1}\ \left(C_1^{\nu_{\mu+1}-1}\ \ldots\right) \in t_{\nu_{\mu+1}-1}_\mathsf{rep_set} \\
& \vdots \\
& \ldots \Longrightarrow \mathsf{rep}_{\nu_{\mu+1}-1}\ \left(C_{k_{\nu_{\mu+1}-1}}^{\nu_{\mu+1}-1}\ \ldots\right) \in t_{\nu_{\mu+1}-1}_\mathsf{rep_set}
\end{array}
$$

Aus $(\star)$ erhält man durch Anwendung der charakteristischen Gleichungen für rep_j das Beweisziel

$$
\begin{array}{l}
\forall x, x', x'', x''', \ldots .\ \mathsf{rep}_{j'}\ x \in t_{j'}_\mathsf{rep_set} \wedge \ldots \Longrightarrow \\
\quad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{rep}_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \\
\qquad \cdots \bullet \mathsf{Rep_}t_{j'''}\ x'' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ x''') \bullet \cdots) \in t_j_\mathsf{rep_set}
\end{array}
$$

Unter Verwendung der entsprechenden Einführungsregel für die Menge $t_j_\mathsf{rep_set}$ erhält man hieraus die Beweisziele

$$
\begin{array}{l}
\forall x, x', x'', x''', \ldots .\ \mathsf{rep}_{j'}\ x \in t_{j'}_\mathsf{rep_set} \wedge \ldots \Longrightarrow \mathsf{rep}_{j'}\ x \in t_{j'}_\mathsf{rep_set} \\
\ldots \Longrightarrow \mathsf{rep}_{j''}\ x' \in t_{j''}_\mathsf{rep_set} \\
\ldots \Longrightarrow \mathsf{Rep_}t_{j'''}\ x'' \in t_{j'''}_\mathsf{rep_set} \\
\quad \vdots
\end{array}
$$

die sich mit Hilfe der Prämisse bzw. Induktionsvoraussetzung $\mathsf{rep}_{j'}\ x \in t_{j'}_\mathsf{rep_set}$, des wegen $\nu_{\mu+1} \leq j''$ bereits bewiesenen Theorems $\mathsf{rep}_{j''}\ x' \in t_{j''}_\mathsf{rep_set}$ und des Axioms (Rep_$t_{j'''}$) lösen lassen.

4.3.5 Umkehrung der Repräsentationsfunktionen

Mit Hilfe der Theoreme

$$
\begin{array}{lcll}
\mathsf{inj}\ f & \Longrightarrow & \mathsf{inv}\ f\ (f\ x) = x & \text{(inv_f_f)} \\
y \in \mathsf{range}\ f & \Longrightarrow & f\ (\mathsf{inv}\ f\ y) = y & \text{(f_inv_f)}
\end{array}
$$

erhält man aus den soeben bewiesenen Eigenschaften von rep_j die Theoreme

$$
\begin{array}{l}
\mathsf{inv}\ \mathsf{rep}_j\ (\mathsf{rep}_j\ x) = x \\
y \in t_j_\mathsf{rep_set} \Longrightarrow \mathsf{rep}_j\ (\mathsf{inv}\ \mathsf{rep}_j\ y) = y
\end{array}
$$

für $\nu_1 \leq j \leq n$. Zusammen mit

$$\mathsf{rep}_j\ x \in t_j\mathsf{_rep_set}$$

steht also auch für die zu bereits existierenden Typen isomorphen Mengen $t_{\nu_1}\mathsf{_rep_set}$, ..., $t_n\mathsf{_rep_set}$ der gleiche Satz von Theoremen wie für die repräsentierenden Mengen der neu eingeführten Typen zur Verfügung.

4.4 Definition der Konstruktoren

Unter Verwendung der Abstraktions- und Repräsentationsfunktionen können nun für den Datentyp $(\alpha, \beta)\mathsf{term}$ die Konstruktoren Var und App durch

$$\begin{array}{rcl} \mathsf{Var} & :: & \alpha \to (\alpha,\beta)\mathsf{term} \\ \mathsf{Var}\ a & \equiv & \mathsf{Abs_term}\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a))) \\ \mathsf{App} & :: & \beta \to ((\alpha,\beta)\mathsf{term})\mathsf{list} \to (\alpha,\beta)\mathsf{term} \\ \mathsf{App}\ b\ ts & \equiv & \mathsf{Abs_term}\ (\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{rep_term_list}\ ts))) \end{array}$$

definiert werden

Allgemeines Verfahren Für die neu eingeführten Typen definieren wir die Konstruktoren C_1^1, ..., $C_{k_1}^1$, ..., $C_1^{\nu_1-1}$, ..., $C_{k_{\nu_1-1}}^{\nu_1-1}$. Ein Konstruktor der Gestalt

$$C_i^j\ \ldots\ (\alpha_1,\ldots,\alpha_h)t_{j'}\ \ldots\ (\alpha_1,\ldots,\alpha_h)t_{j''}\ \ldots\ \tau_{i'}\ \ldots$$

mit $1 \leq j' < \nu_1$ und $\nu_1 \leq j''$ wird hierbei durch

$$\begin{array}{l} C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots \equiv \\ \quad \mathsf{Abs_}t_j\ (\underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{Rep_}t_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i''}\ x'') \bullet \cdots)) \end{array}$$

definiert. Für einen Konstruktor $C_{\tilde{i}}^{\tilde{j}}$ ohne Argumente lautet die Definition

$$C_{\tilde{i}}^{\tilde{j}} \equiv \underline{\mathsf{in}}_{\tilde{i}}\ \mathsf{arbitrary}$$

4.5 Repräsentationsabhängige Beweise

4.5.1 Verschiedenheit von Konstruktoren

Da die Anzahl von Ungleichungen der Form

$$C_i^j\ \ldots \neq C_{i'}^j\ \ldots \qquad \text{für } i \neq i'$$

quadratisch mit der Anzahl der Konstruktoren steigt, werden stattdessen Gleichungen der Form

$$\mathsf{Rep_}t_j\ (C_i^j\ \ldots) = \underline{\mathsf{in}}_i\ \ldots$$

bewiesen. Mit Hilfe des Theorems

$$\mathsf{Rep_}t_j\ (C_i^j\ \ldots) \neq \mathsf{Rep_}t_j\ (C_{i'}^j\ \ldots) \Longrightarrow C_i^j\ \ldots \neq C_{i'}^j\ \ldots$$

das man durch Kontraposition aus der Kongruenzregel für Funktionen erhält, läßt sich somit die Verschiedenheit von C_i^j und $C_{i'}^j$ auf die Verschiedenheit von $\underline{\mathsf{in}}_i$ und $\underline{\mathsf{in}}_{i'}$ zurückführen, die sich aufgrund der in Abschnitt 4.2 erläuterten Konstruktion der Injektionen in maximal $\lceil \log_2(k_j) \rceil$ Schritten zeigen läßt, wobei k_j die Anzahl der Konstruktoren des Datentyps ist.

Im Fall des in Abschnitt 4.4 angegebenen Konstruktors C_i^j gehen wir vom Beweisziel

$$\begin{aligned}&\mathsf{Rep_}t_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots) = \\ &\quad \underline{\mathsf{in}}_i(\cdots \bullet \mathsf{Rep_}t_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i''}\ x'') \bullet \cdots)\end{aligned}$$

aus. Mit Hilfe der Regel

$$\frac{\mathsf{inj_on}\ f\ S \quad f\ a = f\ b \quad a \in S \quad b \in S}{a = b}\ (\text{inj_onD})$$

und der Injektivität von $\mathsf{Abs_}t_j$ auf $t_j\mathsf{_rep_set}$ erhält man hieraus die neuen Beweisziele

$$\begin{aligned}(1)\quad &\mathsf{Abs_}t_j\ (\mathsf{Rep_}t_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots)) = \\ &\quad \mathsf{Abs_}t_j\ (\underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{Rep_}t_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i''}\ x'') \bullet \cdots))\end{aligned}$$

$$(2)\quad \mathsf{Rep_}t_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots) \in t_j\mathsf{_rep_set}$$

$$(3)\quad \underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{Rep_}t_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i''}\ x'') \bullet \cdots) \in t_j\mathsf{_rep_set}$$

Beweisziel (1) kann mit Hilfe des Axioms (Rep_t_j_inverse), der Definition des Konstruktors C_i^j und Reflexivität gelöst werden, Beweisziel (2) mit Hilfe des Axioms (Rep_t_j). Beweisziel (3) wird schließlich durch Anwendung der entsprechenden Einführungsregel für die Menge $t_j\mathsf{_rep_set}$ mit anschließender Anwendung des Axioms (Rep_$t_{j'}$) und des in Abschnitt 4.3 bewiesenen Theorems $\mathsf{rep}_{j''}\ x' \in t_{j''}\mathsf{_rep_set}$ gezeigt.

4.5.2 Injektivität von Konstruktoren

Beim Beweis der Injektivität des Konstruktors C_i^j gehen wir vom Beweisziel

$$\begin{aligned}&C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots = C_i^j\ \ldots\ y\ \ldots\ y'\ \ldots\ y''\ \ldots \Longrightarrow \\ &\quad x = y \wedge x' = y' \wedge x'' = y'' \wedge \ldots\end{aligned}$$

aus. Durch Anwendung der Kongruenzregel für Funktionen erhalten wir

$$\begin{aligned}&\mathsf{Rep_}t_j\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots) = \mathsf{Rep_}t_j\ (C_i^j\ \ldots\ y\ \ldots\ y'\ \ldots\ y''\ \ldots) \Longrightarrow \\ &\quad x = y \wedge x' = y' \wedge x'' = y'' \wedge \ldots\end{aligned}$$

woraus sich mit Hilfe der in Abschnitt 4.5.1 bewiesenen Regeln

$$\begin{array}{l}\underline{\mathsf{in}}_i(\cdots \bullet \mathsf{Rep_}t_{j'}\ x \bullet \cdots \bullet \mathsf{rep}_{j''}\ x' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i''}\ x'') \bullet \cdots) = \\ \quad \underline{\mathsf{in}}_i(\cdots \bullet \mathsf{Rep_}t_{j'}\ y \bullet \cdots \bullet \mathsf{rep}_{j''}\ y' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i''}\ y'') \bullet \cdots) \Longrightarrow \\ \quad\quad x = y \wedge x' = y' \wedge x'' = y'' \wedge \ldots\end{array}$$

ergibt. Dieses Beweisziel läßt sich mit Hilfe der Injektivität von $\underline{\mathsf{in}}_i$, $\mathsf{in}_{i''}$ und Leaf, der aus dem Axiom (Rep_$t_{j'}$_inverse) folgerbaren Injektivität von $\mathsf{Rep_}t_{j'}$ sowie der in Abschnitt 4.3 bewiesenen Injektivität von $\mathsf{rep}_{j''}$ lösen. Die andere Richtung

$$\begin{array}{l}x = y \wedge x' = y' \wedge x'' = y'' \wedge \ldots \Longrightarrow \\ \quad C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots = C_i^j\ \ldots\ y\ \ldots\ y'\ \ldots\ y''\ \ldots\end{array}$$

ist trivial.

4.5.3 Strukturelle Induktion

Mit Hilfe der in Abschnitt 4.3 bewiesenen Isomorphie-Eigenschaften können wir aus der konkreten Induktionsregel

$$\frac{\begin{array}{l}\forall a.\ Q_1\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a))) \\ \forall b, ys.\ Q_2\ ys \wedge ys \in \mathsf{term_list_rep_set} \Longrightarrow Q_1\ (\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ ys)) \\ Q_2\ (\mathsf{In0}\ \mathsf{arbitrary}) \\ \forall y, ys.\ Q_1\ y \wedge y \in \mathsf{term_rep_set} \wedge Q_2\ ys \wedge ys \in \mathsf{term_list_rep_set} \Longrightarrow \\ \quad Q_2\ (\mathsf{In1}\ (\mathsf{Scons}\ y\ ys))\end{array}}{y \in \mathsf{term_rep_set} \Longrightarrow Q_1\ y \wedge ys \in \mathsf{term_rep_set} \Longrightarrow Q_2\ ys}$$

für die Mengen term_rep_set und term_list_rep_set die abstrakte Induktionsregel

$$\frac{\begin{array}{lcl}\forall a. & & P_1\ (\mathsf{Var}\ a) \\ \forall b, ts.\ P_2\ ts & \Longrightarrow & P_1\ (\mathsf{App}\ b\ ts) \\ & & P_2\ \mathsf{Nil} \\ \forall t, ts.\ P_1\ t \wedge P_2\ ts & \Longrightarrow & P_2\ (\mathsf{Cons}\ t\ ts)\end{array}}{P_1\ t \wedge P_2\ ts}$$

für die Typen $(\alpha, \beta)\mathsf{term}$ und $((\alpha, \beta)\mathsf{term})\mathsf{list}$ herleiten. Hierfür zeigen wir unter den Voraussetzungen

(i) $\forall a.\ P_1\ (\mathsf{Var}\ a)$

(ii) $\forall b, ts.\ P_2\ ts \Longrightarrow P_1\ (\mathsf{App}\ b\ ts)$

(iii) $P_2\ \mathsf{Nil}$

(iv) $\forall t, ts.\ P_1\ t \wedge P_2\ ts \Longrightarrow P_2\ (\mathsf{Cons}\ t\ ts)$

das Beweisziel

$$P_1\ t \wedge P_2\ ts$$

Mit Hilfe der Axiome (Rep_term_inverse) und (Rep_term) sowie der in Abschnitt 4.3 bewiesenen Theoreme

$$\textsf{inv rep_term_list}\ (\textsf{rep_term_list}\ ts) = ts \qquad \text{(iso1)}$$

und $\textsf{rep_term_list}\ ts \in \textsf{term_list_rep_set}$ erhalten wir hieraus das Beweisziel

$$\begin{array}{l}\textsf{Rep_term}\ t \in \textsf{term_rep_set} \Longrightarrow P_1\ (\textsf{Abs_term}\ (\textsf{Rep_term}\ t)) \wedge \\ \textsf{rep_term_list}\ ts \in \textsf{term_list_rep_set} \Longrightarrow P_2\ (\textsf{inv rep_term_list}\ (\textsf{rep_term_list}\ ts))\end{array}$$

Nun kann die Induktionsregel für die Mengen $\textsf{term_rep_set}$ und $\textsf{term_list_rep_set}$ angewendet werden, wobei $Q_1 = P_1 \circ \textsf{Abs_term}$, $Q_2 = P_2 \circ \textsf{inv rep_term_list}$ und $y = \textsf{Rep_term}\ t$, $ys = \textsf{rep_term_list}\ ts$. Man erhält somit die Beweisziele

$$\begin{array}{ll}(1) & \forall a.\ P_1\ (\textsf{Abs_term}\ (\textsf{In0}\ (\textsf{Leaf}\ (\textsf{Inl}\ a)))) \\ (2) & \forall b, ys.\ P_2\ (\textsf{inv rep_term_list}\ ys) \wedge ys \in \textsf{term_list_rep_set} \Longrightarrow \\ & \qquad P_1\ (\textsf{Abs_term}\ (\textsf{In1}\ (\textsf{Scons}\ (\textsf{Leaf}\ (\textsf{Inr}\ b))\ ys))) \\ (3) & P_2\ (\textsf{inv rep_term_list}\ (\textsf{In0 arbitrary})) \\ (4) & \forall y, ys.\ P_1\ (\textsf{Abs_term}\ y) \wedge y \in \textsf{term_rep_set} \wedge \\ & \qquad P_2\ (\textsf{inv rep_term_list}\ ys) \wedge ys \in \textsf{term_list_rep_set} \Longrightarrow \\ & \qquad\quad P_2\ (\textsf{inv rep_term_list}\ (\textsf{In1}\ (\textsf{Scons}\ y\ ys)))\end{array}$$

Mit Hilfe der Definition des Konstruktors Var erhält man aus (1) das Beweisziel $\forall a.\ P_1\ (\textsf{Var}\ a)$ was sofort aus Voraussetzung (i) folgt. Aus (2) erhält man aufgrund der Prämisse $ys \in \textsf{term_list_rep_set}$ und des Theorems

$$ys \in \textsf{term_list_rep_set} \Longrightarrow \textsf{rep_term_list}\ (\textsf{inv rep_term_list}\ ys) = ys \qquad \text{(iso2)}$$

aus Abschnitt 4.3 das neue Beweisziel

$$\begin{array}{l}\forall b, ys.\ P_2\ (\textsf{inv rep_term_list}\ ys) \Longrightarrow \\ \qquad P_1\ (\textsf{Abs_term}\ (\textsf{In1}\ (\textsf{Scons}\ (\textsf{Leaf}\ (\textsf{Inr}\ b))\ (\textsf{rep_term_list}\ (\textsf{inv rep_term_list}\ ys)))))\end{array}$$

Aufgrund der Definition des Konstruktors App ergibt sich hieraus das Beweisziel

$$\forall b, ys.\ P_2\ (\textsf{inv rep_term_list}\ ys) \Longrightarrow P_1\ (\textsf{App}\ b\ (\textsf{inv rep_term_list}\ ys))$$

was sich mit Hilfe von Voraussetzung (ii) lösen läßt. Aus (3) ergibt sich unter Verwendung der charakteristischen Gleichungen für rep_term_list das Beweisziel

$$P_2\ (\textsf{inv rep_term_list}\ (\textsf{rep_term_list Nil}))$$

Wegen (iso1) ist dies gleichbedeutend mit P_2 Nil, was wieder direkt aus Voraussetzung (iii) folgt. Aus (4) erhält man unter Verwendung der Prämisse $y \in \textsf{term_rep_set}$ in Verbindung mit dem Axiom (Abs_term_inverse) sowie der Prämisse $ys \in \textsf{term_list_rep_set}$ in Verbindung mit dem Theorem (iso2) das Beweisziel

$$\begin{array}{l}\forall y, ys.\ P_1\ (\textsf{Abs_term}\ y) \wedge P_2\ (\textsf{inv rep_term_list}\ ys) \Longrightarrow \\ \qquad P_2\ (\textsf{inv rep_term_list}\ (\textsf{In1}\ (\textsf{Scons}\ (\textsf{Rep_term}\ (\textsf{Abs_term}\ y)) \\ \qquad\quad (\textsf{rep_term_list}\ (\textsf{inv rep_term_list}\ ys)))))\end{array}$$

Unter Verwendung der charakteristischen Gleichungen für rep_term_list ergibt sich das Beweisziel

$$\forall y, ys.\ P_1\ (\mathsf{Abs_term}\ y) \wedge P_2\ (\mathsf{inv\ rep_term_list}\ ys) \Longrightarrow \\ P_2\ (\mathsf{inv\ rep_term_list}\ (\mathsf{rep_term_list}\ (\mathsf{Cons}\ (\mathsf{Abs_term}\ y)\ (\mathsf{inv\ rep_term_list}\ ys))))$$

Wegen (iso1) ist dies gleichbedeutend mit

$$\forall y, ys.\ P_1\ (\mathsf{Abs_term}\ y) \wedge P_2\ (\mathsf{inv\ rep_term_list}\ ys) \Longrightarrow \\ P_2\ (\mathsf{Cons}\ (\mathsf{Abs_term}\ y)\ (\mathsf{inv\ rep_term_list}\ ys))$$

was sich mit Hilfe von Voraussetzung (iv) lösen läßt.

Allgemeines Verfahren Für die abstrakte Erläuterung des Beweises der strukturellen Induktionsregel wollen wir annehmen, daß die aufgefaltete Datentypdefinition aus Abschnitt 4.1 die zwei Konstruktoren C_i^j mit $1 \leq j < \nu_1$ und $1 \leq i \leq k_j$ sowie $C_{\tilde{\imath}}^{\tilde{\jmath}}$ mit $\nu_1 \leq \tilde{\jmath} \leq n$ und $1 \leq \tilde{\imath} \leq k_{\tilde{\jmath}}$ enthält. C_i^j ist also ein Konstruktor eines neuen Datentyps, während $C_{\tilde{\imath}}^{\tilde{\jmath}}$ ein Konstruktor eines bereits existierenden Datentyps ist. Diese Konstruktoren seien von der Gestalt

$$C_i^j\ \ldots\ (\alpha_1, \ldots, \alpha_h)t_{j'}\ \ldots\ (\alpha_1, \ldots, \alpha_h)t_{j''}\ \ldots\ \tau_{i'}\ \ldots \\ C_{\tilde{\imath}}^{\tilde{\jmath}}\ \ldots\ (\alpha_1, \ldots, \alpha_h)t_{\tilde{\jmath}'}\ \ldots\ (\alpha_1, \ldots, \alpha_h)t_{\tilde{\jmath}''}\ \ldots\ \tau_{\tilde{\imath}'}\ \ldots$$

wobei $1 \leq j',\ \tilde{\jmath}' < \nu_1$ und $\nu_1 \leq j'',\ \tilde{\jmath}'' \leq n$. Die konkrete Induktionsregel für die repräsentierenden Mengen hat die Gestalt

$$\frac{\begin{array}{l} \ldots \Longrightarrow Q_1\ (\underline{\mathsf{in}}_1\ \ldots) \\ \quad\quad \vdots \\ \ldots \Longrightarrow Q_1\ (\underline{\mathsf{in}}_{k_1}\ \ldots) \\ \quad\quad \vdots \\ \forall y, y', y'', \ldots.\ Q_{j'}\ y \wedge y \in t_{j'}\mathsf{_rep_set} \wedge Q_{j''}\ y' \wedge y' \in t_{j''}\mathsf{_rep_set} \wedge \ldots \Longrightarrow \\ \quad Q_j\ (\underline{\mathsf{in}}_i\ (\cdots \bullet y \bullet \cdots \bullet y' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ y'') \bullet \cdots)) \\ \quad\quad \vdots \\ \forall y, y', y'', \ldots.\ Q_{\tilde{\jmath}'}\ y \wedge y \in t_{\tilde{\jmath}'}\mathsf{_rep_set} \wedge Q_{\tilde{\jmath}''}\ y' \wedge y' \in t_{\tilde{\jmath}''}\mathsf{_rep_set} \wedge \ldots \Longrightarrow \\ \quad Q_{\tilde{\jmath}}\ (\underline{\mathsf{in}}_{\tilde{\imath}}\ (\cdots \bullet y \bullet \cdots \bullet y' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{\tilde{\imath}'}\ y'') \bullet \cdots)) \\ \quad\quad \vdots \\ \ldots \Longrightarrow Q_n\ (\underline{\mathsf{in}}_1\ \ldots) \\ \quad\quad \vdots \\ \ldots \Longrightarrow Q_n\ (\underline{\mathsf{in}}_{k_n}\ \ldots) \end{array}}{y_1 \in t_1\mathsf{_rep_set} \Longrightarrow Q_1\ y_1 \wedge \ldots \wedge y_n \in t_n\mathsf{_rep_set} \Longrightarrow Q_n\ y_n}$$

Die zu beweisende, abstrakte Induktionsregel hat die Gestalt

$$\frac{\begin{array}{ccc} & \vdots & \\ \forall x, x', x'', \ldots .\ P_{j'}\ x \wedge P_{j''}\ x' \wedge \ldots & \Longrightarrow & P_j \left(C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots \right) \\ & \vdots & \\ \forall x, x', x'', \ldots .\ P_{\tilde{j}'}\ x \wedge P_{\tilde{j}''}\ x' \wedge \ldots & \Longrightarrow & P_{\tilde{j}} \left(C_{\tilde{i}}^{\tilde{j}}\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots \right) \\ & \vdots & \end{array}}{P_1\ x_1 \wedge \ldots \wedge P_{\nu_1 - 1}\ x_{\nu_1 - 1} \wedge P_{\nu_1}\ x_{\nu_1} \wedge \ldots \wedge P_n\ x_n}$$

Unter den Voraussetzungen

$$\begin{array}{l} \forall x, x', x'', \ldots .\ P_{j'}\ x \wedge P_{j''}\ x' \wedge \ldots \Longrightarrow P_j \left(C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots \right) \\ \forall x, x', x'', \ldots .\ P_{\tilde{j}'}\ x \wedge P_{\tilde{j}''}\ x' \wedge \ldots \Longrightarrow P_{\tilde{j}} \left(C_{\tilde{i}}^{\tilde{j}}\ \ldots\ x\ \ldots\ x'\ \ldots\ x''\ \ldots \right) \\ \qquad\qquad \vdots \end{array}$$

zeigen wir nun

$$P_1\ x_1 \wedge \ldots \wedge P_{\nu_1 - 1}\ x_{\nu_1 - 1} \wedge P_{\nu_1}\ x_{\nu_1} \wedge \ldots \wedge P_n\ x_n$$

Aus diesem Beweisziel erhält man mit Hilfe der Axiome (Rep_t_μ_inverse) und (Rep_t_μ) sowie der in Abschnitt 4.3 bewiesenen Theoreme $\mathsf{inv\ rep}_{\mu'}\ (\mathsf{rep}_{\mu'}\ x) = x$ und $\mathsf{rep}_{\mu'}\ x \in t_{\mu'}\mathsf{_rep_set}$ das neue Beweisziel

$$\begin{array}{l} \mathsf{Rep_}t_1\ x_1 \in t_1\mathsf{_rep_set} \Longrightarrow P_1\ (\mathsf{Abs_}t_1\ (\mathsf{Rep_}t_1\ x_1)) \wedge \ldots \wedge \\ \mathsf{Rep_}t_{\nu_1 - 1}\ x_{\nu_1 - 1} \in t_{\nu_1 - 1}\mathsf{_rep_set} \Longrightarrow P_{\nu_1 - 1}\ (\mathsf{Abs_}t_{\nu_1 - 1}\ (\mathsf{Rep_}t_{\nu_1 - 1}\ x_{\nu_1 - 1})) \wedge \\ \mathsf{rep}_{\nu_1}\ x_{\nu_1} \in t_{\nu_1}\mathsf{_rep_set} \Longrightarrow P_{\nu_1}\ (\mathsf{inv\ rep}_{\nu_1}\ (\mathsf{rep}_{\nu_1}\ x_{\nu_1})) \wedge \ldots \wedge \\ \mathsf{rep}_n\ x_n \in t_n\mathsf{_rep_set} \Longrightarrow P_n\ (\mathsf{inv\ rep}_n\ (\mathsf{rep}_n\ x_n)) \end{array}$$

Nun kann die konkrete Induktionsregel für die repräsentierenden Mengen angewendet werden, wobei $Q_1 = P_1 \circ \mathsf{Abs_}t_1, \ldots, Q_n = P_n \circ \mathsf{inv\ rep}_n$ und $y_1 = \mathsf{Rep_}t_1\ x_1, \ldots, y_n = \mathsf{rep}_n\ x_1$. Dies liefert die Beweisziele

$$\begin{array}{ll} & \vdots \\ (\star) & \forall y, y', y'', \ldots .\ P_{j'}\ (\mathsf{Abs_}t_{j'}\ y) \wedge y \in t_{j'}\mathsf{_rep_set} \wedge \\ & \quad P_{j''}\ (\mathsf{inv\ rep}_{j''}\ y') \wedge y' \in t_{j''}\mathsf{_rep_set} \wedge \ldots \Longrightarrow \\ & \qquad P_j\ (\mathsf{Abs_}t_j\ (\underline{\mathsf{in}}_i\ (\cdots \bullet y \bullet \cdots \bullet y' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ y'') \bullet \cdots))) \\ & \vdots \\ (\star\star) & \forall y, y', y'', \ldots .\ P_{\tilde{j}'}\ (\mathsf{Abs_}t_{\tilde{j}'}\ y) \wedge y \in t_{\tilde{j}'}\mathsf{_rep_set} \wedge \\ & \quad P_{\tilde{j}''}\ (\mathsf{inv\ rep}_{\tilde{j}''}\ y') \wedge y' \in t_{\tilde{j}''}\mathsf{_rep_set} \wedge \ldots \Longrightarrow \\ & \qquad P_{\tilde{j}}\ (\mathsf{inv\ rep}_{\tilde{j}}\ (\underline{\mathsf{in}}_{\tilde{i}}\ (\cdots \bullet y \bullet \cdots \bullet y' \bullet \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{\tilde{i}'}\ y'') \bullet \cdots))) \\ & \vdots \end{array}$$

Aus ($\star$) erhält man mittels (Abs_$t_{j'}$_inverse) in Verbindung mit der Prämisse $y \in t_{j'}\mathsf{_rep_set}$, sowie dem Theorem

$$y' \in t_{j''}\mathsf{_rep_set} \Longrightarrow \mathsf{rep}_{j''}\ (\mathsf{inv\ rep}_{j''}\ y') = y'$$

aus Abschnitt 4.3 in Verbindung mit der Prämisse $y' \in t_{j''}_\mathsf{rep_set}$ das Beweisziel

$$\begin{array}{l}\forall y, y', y'', \ldots .\ P_{j'}\ (\mathsf{Abs_}t_{j'}\ y) \wedge P_{j''}\ (\mathsf{inv\ rep}_{j''}\ y') \wedge \ldots \Longrightarrow \\ \quad P_j\ (\mathsf{Abs_}t_j\ (\underline{\mathsf{in}}_i\ (\cdots \bullet \mathsf{Rep_}t_{j'}\ (\mathsf{Abs_}t_{j'}\ y) \bullet \cdots \bullet \mathsf{rep}_{j''}\ (\mathsf{inv\ rep}_{j''}\ y') \bullet \\ \qquad \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{i'}\ y'') \bullet \cdots)))\end{array}$$

Die Anwendung der Definition von C_i^j ergibt nun

$$\begin{array}{l}\forall y, y', y'', \ldots .\ P_{j'}\ (\mathsf{Abs_}t_{j'}\ y) \wedge P_{j''}\ (\mathsf{inv\ rep}_{j''}\ y') \wedge \ldots \Longrightarrow \\ \quad P_j\ (C_i^j\ \ \ldots\ \ (\mathsf{Abs_}t_{j'}\ y)\ \ \ldots\ \ (\mathsf{inv\ rep}_{j''}\ y')\ \ \ldots\ \ y''\ \ \ldots)\end{array}$$

was sich mit Hilfe der Voraussetzungen lösen läßt. In ähnlicher Weise erhält man aus $(\star\star)$ das Beweisziel

$$\begin{array}{l}\forall y, y', y'', \ldots .\ P_{\tilde{\jmath}'}\ (\mathsf{Abs_}t_{\tilde{\jmath}'}\ y) \wedge P_{\tilde{\jmath}''}\ (\mathsf{inv\ rep}_{\tilde{\jmath}''}\ y') \wedge \ldots \Longrightarrow \\ \quad P_{\tilde{\jmath}}\ (\mathsf{inv\ rep}_{\tilde{\jmath}}\ (\underline{\mathsf{in}}_{\tilde{\imath}}\ (\cdots \bullet \mathsf{Rep_}t_{\tilde{\jmath}'}\ (\mathsf{Abs_}t_{\tilde{\jmath}'}\ y) \bullet \cdots \bullet \mathsf{rep}_{\tilde{\jmath}''}\ (\mathsf{inv\ rep}_{\tilde{\jmath}''}\ y') \bullet \\ \qquad \cdots \bullet \mathsf{Leaf}\ (\mathsf{in}_{\tilde{\imath}'}\ y'') \bullet \cdots)))\end{array}$$

Mit Hilfe der charakteristischen Gleichungen für $\mathsf{rep}_{\tilde{\jmath}}$ erhält man

$$\begin{array}{l}\forall y, y', y'', \ldots .\ P_{\tilde{\jmath}'}\ (\mathsf{Abs_}t_{\tilde{\jmath}'}\ y) \wedge P_{\tilde{\jmath}''}\ (\mathsf{inv\ rep}_{\tilde{\jmath}''}\ y') \wedge \ldots \Longrightarrow \\ \quad P_{\tilde{\jmath}}\ (\mathsf{inv\ rep}_{\tilde{\jmath}}\ (\mathsf{rep}_{\tilde{\jmath}}\ (C_{\tilde{\imath}}^{\tilde{\jmath}}\ \ \ldots\ \ (\mathsf{Abs_}t_{\tilde{\jmath}'}\ y)\ \ \ldots\ \ (\mathsf{inv\ rep}_{\tilde{\jmath}''}\ y')\ \ \ldots\ \ y''\ \ \ldots)))\end{array}$$

Wegen $\mathsf{inv\ rep}_{\tilde{\jmath}}\ (\mathsf{rep}_{\tilde{\jmath}}\ x) = x$ ergibt sich hieraus nun das Beweisziel

$$\begin{array}{l}\forall y, y', y'', \ldots .\ P_{\tilde{\jmath}'}\ (\mathsf{Abs_}t_{\tilde{\jmath}'}\ y) \wedge P_{\tilde{\jmath}''}\ (\mathsf{inv\ rep}_{\tilde{\jmath}''}\ y') \wedge \ldots \Longrightarrow \\ \quad P_{\tilde{\jmath}}\ (C_{\tilde{\imath}}^{\tilde{\jmath}}\ \ \ldots\ \ (\mathsf{Abs_}t_{\tilde{\jmath}'}\ y)\ \ \ldots\ \ (\mathsf{inv\ rep}_{\tilde{\jmath}''}\ y')\ \ \ldots\ \ y''\ \ \ldots)\end{array}$$

das ebenfalls mit Hilfe der Voraussetzungen gelöst werden kann.

4.6 Repräsentationsunabhängige Beweise

Sind die elementaren Eigenschaften eines Datentyps wie Verschiedenheit und Injektivität von Konstruktoren sowie strukturelle Induktion einmal bewiesen, so können, wie im folgenden beschrieben wird, alle anderen Eigenschaften wie z.B. die Existenz eines Kombinators für primitive Rekursion daraus abgeleitet werden. Dieser Sachverhalt ermöglicht es auch, in Isabelle/HOL manuell konstruierte Typen wie $\alpha \times \beta$ oder $\alpha + \beta$, die alle elementaren Eigenschaften eines Datentyps besitzen, nachträglich als Datentyp zu deklarieren. Derartige Typen können somit insbesondere auch in verschachtelt rekursiven Datentypen wie

$$\begin{array}{lcl}\mathsf{datatype}\quad (\alpha, \beta)\mathsf{test} & = & C\ \alpha \\ & \mid & D\ (((\alpha, \beta)\mathsf{test} + \beta)\mathsf{list})\end{array}$$

verwendet werden. Die Repräsentationsfunktion für den Typ $(\alpha, \beta)\mathsf{test} + \beta$ wird hierbei, wie in Abschnitt 4.3 beschrieben, mit Hilfe eines Kombinators für primitive Rekursion konstruiert,

dessen Existenz sich aus der Injektivität und Verschiedenheit der Konstruktoren Inl und Inr sowie der – etwas degenerierten – Induktionsregel

$$\frac{\forall x.\ P\ (\mathsf{Inl}\ x) \quad \forall x.\ P\ (\mathsf{Inr}\ x)}{P\ s}$$

ergibt.

4.6.1 Fallunterscheidung

Für $1 \leq j < \nu_1$ erhält man aus der strukturellen Induktionsregel

$$\frac{\begin{array}{ccc} & \vdots & \\ \forall x_1, \ldots, x_{m_1^j}.\ \ldots & \Longrightarrow & P_j \left(C_1^j\ x_1\ \ldots\ x_{m_1^j} \right) \\ & \vdots & \\ \forall x_1, \ldots, x_{m_{k_j}^j}.\ \ldots & \Longrightarrow & P_j \left(C_{k_j}^j\ x_1\ \ldots\ x_{m_{k_j}^j} \right) \\ & \vdots & \end{array}}{P_1\ x_1 \wedge \ldots \wedge P_{\nu_1 - 1}\ x_{\nu_1 - 1} \wedge P_{\nu_1}\ x_{\nu_1} \wedge \ldots \wedge P_n\ x_n}$$

leicht die Fallunterscheidungsregel

$$\frac{\begin{array}{ccc} \forall x_1, \ldots, x_{m_1^j}.\ u = C_1^j\ x_1\ \ldots\ x_{m_1^j} & \Longrightarrow & P \\ & & \vdots \\ \forall x_1, \ldots, x_{m_{k_j}^j}.\ u = C_{k_j}^j\ x_1\ \ldots\ x_{m_{k_j}^j} & \Longrightarrow & P \end{array}}{P}$$

indem $P_j = \lambda z.\ u = z \Longrightarrow P$ und $P_{j'} = \lambda z.\ \mathsf{True}$ für alle $j' \neq j$ gesetzt wird.

4.6.2 Primitive Rekursion

Die Kombinatoren für primitive Rekursion erhält man relativ leicht, indem man deren Graph, d.h. die Relation zwischen Funktionsargumenten und Ergebnissen induktiv definiert. Dieses Verfahren wird auch in [Harrison, 1995] vorgeschlagen. Unter Verwendung der Induktionsregel für die beteiligten Datentypen kann dann gezeigt werden, daß die induktiv definierten Relationen tatsächlich totale Funktionen repräsentieren, d.h. *linkstotal* und *rechtseindeutig* sind. Mit Hilfe des Hilbertschen Auswahloperators können aus diesen Relationen dann Funktionen gewonnen und deren charakteristische Gleichungen bewiesen werden.

4.6.2.1 Induktive Definition der Funktionsgraphen

Im Falle der Typen $(\alpha, \beta)\mathsf{term}$ und $((\alpha, \beta)\mathsf{term})\mathsf{list}$ werden die Mengen bzw. Relationen

$$\begin{array}{lcl} \mathsf{term_rec_set}\ f_1\ \ldots\ f_4 & :: & ((\alpha, \beta)\mathsf{term} \times \gamma)\mathsf{set} \\ \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 & :: & (((\alpha, \beta)\mathsf{term})\mathsf{list} \times \delta)\mathsf{set} \end{array}$$

induktiv durch die Einführungsregeln

$$\frac{}{(\mathsf{Var}\ a,\ f_1\ a) \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4}$$

$$\frac{(xs,\ y) \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4}{(\mathsf{App}\ b\ xs,\ f_2\ b\ xs\ y) \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4}$$

$$\frac{}{(\mathsf{Nil},\ f_3) \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4}$$

$$\frac{(x,\ y) \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4 \quad (xs,\ y') \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4}{(\mathsf{Cons}\ x\ xs,\ f_4\ x\ xs\ y\ y') \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4}$$

definiert. Gemäß Abschnitt 2.3.2 erhalten wir für die so definierten Mengen die Eliminationsregeln

$$\frac{\begin{array}{l} r \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4 \\ \forall a.\ r = (\mathsf{Var}\ a,\ f_1\ a) \Longrightarrow P \\ \forall b, xs, y.\ r = (\mathsf{App}\ b\ xs,\ f_2\ b\ xs\ y) \wedge (xs,\ y) \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4 \end{array}}{P}$$

$$\frac{\begin{array}{l} s \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4 \\ s = (\mathsf{Nil},\ f_3) \Longrightarrow Q \\ \forall x, xs, y, y'.\ s = (\mathsf{Cons}\ x\ xs,\ f_4\ x\ xs\ y\ y') \wedge (x,\ y) \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4\ \wedge \\ \qquad (xs,\ y') \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4 \Longrightarrow Q \end{array}}{Q}$$

Allgemeines Verfahren Wir definieren induktiv die Relationen

$$\begin{array}{lcl} t_1\mathsf{_rec_set}\ f_1^1\ \dots\ f_{k_1}^1\ \dots\ f_1^n\ \dots\ f_{k_n}^n & :: & ((\alpha_1, \dots, \alpha_h)t_1 \times \beta_1)\mathsf{set} \\ & \vdots & \\ t_n\mathsf{_rec_set}\ f_1^1\ \dots\ f_{k_1}^1\ \dots\ f_1^n\ \dots\ f_{k_n}^n & :: & ((\alpha_1, \dots, \alpha_h)t_n \times \beta_n)\mathsf{set} \end{array}$$

wobei wir, wie auch schon bei der Definition der repräsentierenden Mengen, für jeden der Konstruktoren $C_1^1, \dots, C_{k_1}^1, \dots, C_1^n, \dots, C_{k_n}^n$ der beteiligten Datentypen eine Einführungsregel angeben. Für einen Konstruktor der Form

$$C_i^j\ \dots\ (\alpha_1, \dots, \alpha_h)t_{j'}\ \dots\ \tau_{i'}\ \dots$$

gibt man hierbei die Regel

$$\frac{(x,\ y) \in t_{j'}\mathsf{_rec_set}\ f_1^1\ \dots\ f_{k_n}^n \quad \dots}{\begin{array}{l}(C_i^j\ x_1 \dots\ x\ \dots\ x'\ \dots\ x_{m_i^j}, \\ \qquad f_i^j\ x_1 \dots\ x\ \dots\ x'\ \dots\ x_{m_i^j}\ y_1\ \dots\ y\ \dots\ y_{l_i^j}) \in t_j\mathsf{_rec_set}\ f_1^1\ \dots\ f_{k_n}^n\end{array}}$$

an. Wir erhalten hierbei Eliminationsregeln der Form

$$\frac{\begin{array}{l} r \in t_j_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \\ \quad\vdots \\ \forall x, x', y, \ldots.\ r = (C_i^j\ x_1 \ldots\ x\ \ldots\ x'\ \ldots\ x_{m_i^j}, \\ \quad f_i^j\ x_1 \ldots\ x\ \ldots\ x'\ \ldots\ x_{m_i^j}\ y_1\ \ldots\ y\ \ldots\ y_{l_i^j}) \wedge \\ \quad\quad (x,\ y) \in t_{j'}_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \wedge \ldots \Longrightarrow P \\ \quad\vdots \end{array}}{P}$$

4.6.2.2 Totalität und Eindeutigkeit

Es muß nun gezeigt werden, daß die soeben definierten Relationen tatsächlich totale Funktionen repräsentieren. Der Beweis erfolgt mit Hilfe der in Abschnitt 4.5.3 bewiesenen strukturellen Induktionsregel. Im Falle der Typen $(\alpha, \beta)\mathsf{term}$ und $((\alpha, \beta)\mathsf{term})\mathsf{list}$ gehen wir hierbei vom Beweisziel

$$\exists_1 y.\ (x,\ y) \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4 \wedge \exists_1 y.\ (xs,\ y) \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4$$

aus. Durch Induktion über x und xs erhält man die Beweisziele

(1) $\forall a.\ \exists_1 y.\ (\mathsf{Var}\ a,\ y) \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4$

(2) $\forall b, xs.\ (\exists_1 y'.\ (xs,\ y') \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4) \Longrightarrow$
$\quad\exists_1 y.\ (\mathsf{App}\ b\ xs,\ y) \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4$

(3) $\exists_1 y.\ (\mathsf{Nil},\ y) \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4$

(4) $\forall x, xs.\ (\exists_1 y'.\ (x,\ y') \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4) \wedge$
$\quad(\exists_1 y''.\ (xs,\ y'') \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4) \Longrightarrow$
$\quad\quad\exists_1 y.\ (\mathsf{Cons}\ x\ xs,\ y) \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4$

Da die Vorgehensweise für alle Beweisziele ähnlich ist, wird diese im folgenden exemplarisch am Beweisziel (2) dargestellt. Durch Elimination von $\exists_1$ in der Prämisse erhält man aus (2) das Beweisziel

$$\begin{array}{l} \forall b, xs, z.\ (xs,\ z) \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 \wedge \\ \quad (\forall z'.\ (xs,\ z') \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow z' = z) \Longrightarrow \\ \quad\quad \exists_1 y.\ (\mathsf{App}\ b\ xs,\ y) \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4 \end{array}$$

Mit Hilfe der Einführungsregel für $\exists_1$ ergeben sich daraus die Beweisziele

(5) $\forall b, xs, z.\ (xs,\ z) \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow$
$\quad(\mathsf{App}\ b\ xs,\ f_2\ b\ xs\ z) \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4$

(6) $\forall b, xs, z, z''.\ (\forall z'.\ (xs,\ z') \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow z' = z) \wedge$
$\quad(\mathsf{App}\ b\ xs,\ z'') \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow z'' = f_2\ b\ xs\ z$

Hierbei ergibt sich (5) aus der entsprechenden Einführungsregel für term_rec_set. Durch Anwendung der Eliminationsregel für term_rec_set erhält man aus (6) die neuen Beweisziele

$$(7)\quad \forall b, xs, z, z'', a.\ (\forall z'.\ (xs,\ z') \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow z' = z) \wedge (\mathsf{App}\ b\ xs,\ z'') = (\mathsf{Var}\ a,\ f_1\ a) \Longrightarrow z'' = f_2\ b\ xs\ z$$

$$(8)\quad \forall b, xs, z, z'', b', xs', z'''.\ (\forall z'.\ (xs,\ z') \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow z' = z) \wedge (\mathsf{App}\ b\ xs,\ z'') = (\mathsf{App}\ b'\ xs',\ f_2\ b'\ xs'\ z''') \wedge (xs',\ z''') \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow z'' = f_2\ b\ xs\ z$$

Beweisziel (7) ist wegen App $\ldots \neq$ Var $\ldots$ trivial lösbar. Aus der Prämisse

$$(\mathsf{App}\ b\ xs,\ z'') = (\mathsf{App}\ b'\ xs',\ f_2\ b'\ xs'\ z''')$$

von (8) erhält man aufgrund der Injektivität von $(_,_)$ und App die Gleichungen $b = b'$, $xs = xs'$ sowie $z'' = f_2\ b'\ xs'\ z'''$. Zusammen mit den Prämissen

$$(xs',\ z''') \in \mathsf{term_rec_set}\ f_1\ \ldots\ f_4$$

und

$$\forall z'.\ (xs,\ z') \in \mathsf{term_list_rec_set}\ f_1\ \ldots\ f_4 \Longrightarrow z' = z$$

erhält man schließlich noch $z''' = z$, womit auch Beweisziel (8) gelöst ist.

Allgemeines Verfahren Wir zeigen

$$\exists_1 y.\ (x_1,\ y) \in t_1\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \wedge \ldots \wedge \exists_1 y.\ (x_n,\ y) \in t_n\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$

mit Hilfe der in Abschnitt 4.5.3 bewiesenen strukturellen Induktionsregel. Induktion über x_1, $\ldots$, x_n ergibt die Beweisziele

$$\ldots \Longrightarrow \exists_1 y.\ (C_1^1\ \ldots,\ y) \in t_1\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$
$$\vdots$$
$$\ldots \Longrightarrow \exists_1 y.\ (C_{k_1}^1\ \ldots,\ y) \in t_1\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$
$$\vdots$$
$$(\star)\quad \forall x, x', \ldots.\ (\exists_1 y'.\ (x,\ y') \in t_{j'}\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n) \wedge \ldots \Longrightarrow \exists_1 y.\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots, y) \in t_j\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$
$$\vdots$$
$$\ldots \Longrightarrow \exists_1 y.\ (C_1^n\ \ldots,\ y) \in t_n\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$
$$\vdots$$
$$\ldots \Longrightarrow \exists_1 y.\ (C_{k_n}^n\ \ldots,\ y) \in t_n\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$

Durch Elimination von $\exists_1$ in der Prämisse erhält man aus $(\star)$ das Beweisziel

$$\forall x, x', z, \ldots.\ (x,\ z) \in t_{j'}\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \wedge (\forall z'.\ (x,\ z') \in t_{j'}\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \Longrightarrow z' = z) \wedge \ldots \Longrightarrow \exists_1 y.\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ y) \in t_j\mathsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$

Verwendung der Einführungsregel für $\exists_1$ ergibt die Beweisziele

$$\begin{array}{ll}(1) & \forall x, x', z, \ldots.\ (x,\ z) \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \wedge \ldots \Longrightarrow \\ & \quad (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ f_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ z\ \ldots) \in t_j\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \\ (2) & \forall x, x', z, z'', \ldots.\ (\forall z'.\ (x,\ z') \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \Longrightarrow z' = z) \wedge \\ & \quad (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ z'') \in t_j\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \wedge \ldots \Longrightarrow \\ & \qquad z'' = f_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ z\ \ldots\end{array}$$

Beweisziel (1) kann mit Hilfe der Einführungsregeln für $t_j\textsf{_rec_set}$ in Verbindung mit den Prämissen gelöst werden. Durch Anwendung der Eliminationsregel für $t_j\textsf{_rec_set}$ erhält man aus (2) die neuen Beweisziele

$$\begin{array}{ll} & \forall \ldots.\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ z'') = (C_1^j\ \ldots,\ f_1^j\ \ldots) \wedge \ldots \Longrightarrow \ldots \\ & \quad \vdots \\ (\star\star) & \forall x, x', z, z'', \hat{x}, \hat{x}', z''', \ldots.\ (\forall z'.\ (x,\ z') \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \Longrightarrow z' = z) \wedge \\ & \quad (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ z'') = \\ & \quad (C_i^j\ \ldots\ \hat{x}\ \ldots\ \hat{x}'\ \ldots,\ f_i^j\ \ldots\ \hat{x}\ \ldots\ \hat{x}'\ \ldots\ z'''\ \ldots) \wedge \\ & \qquad (\hat{x},\ z''') \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \wedge \ldots \Longrightarrow \\ & \qquad\quad z'' = f_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ z\ \ldots \\ & \quad \vdots \\ & \forall \ldots.\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ z'') = (C_{k_j}^j\ \ldots,\ f_{k_j}^j\ \ldots) \wedge \ldots \Longrightarrow \ldots\end{array}$$

Alle Beweisziele bis auf $(\star\star)$ sind aufgrund der Verschiedenheit der Konstruktoren trivial lösbar. Aufgrund der Injektivität von $(_,_)$ und C_i^j erhält man aus $(\star\star)$ das Beweisziel

$$\begin{array}{l}\forall x, x', z, z'', \hat{x}, \hat{x}', z''', \ldots.\ (\forall z'.\ (x,\ z') \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \Longrightarrow z' = z) \wedge \\ \quad x = \hat{x} \wedge x' = \hat{x}' \wedge z'' = f_i^j\ \ldots\ \hat{x}\ \ldots\ \hat{x}'\ \ldots\ z'''\ \ldots\ \wedge \\ \qquad (\hat{x},\ z''') \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \wedge \ldots \Longrightarrow z'' = f_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ z\ \ldots\end{array}$$

Aus den Prämissen $x = \hat{x}$ und $(\hat{x},\ z''') \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$ sowie

$$\forall z'.\ (x,\ z') \in t_{j'}\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \Longrightarrow z' = z$$

ergibt sich $z''' = z$, womit sich das Beweisziel lösen läßt.

4.6.2.3 Definition der Kombinatoren

Die Kombinatoren für primitive Rekursion können nun mit Hilfe der Mengen $t_1\textsf{_rec_set}, \ldots,$ $t_n\textsf{_rec_set}$ unter Verwendung des Hilbertschen Auswahloperators durch

$$\begin{array}{rcl}t_1\textsf{_rec}\ f_1^1\ \ldots\ f_{k_n}^n\ x & \equiv & \varepsilon y.\ (x, y) \in t_1\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \\ & \vdots & \\ t_n\textsf{_rec}\ f_1^1\ \ldots\ f_{k_n}^n\ x & \equiv & \varepsilon y.\ (x, y) \in t_n\textsf{_rec_set}\ f_1^1\ \ldots\ f_{k_n}^n\end{array}$$

definiert werden.

4.6.2.4 Charakteristische Gleichungen

Um die definierten Kombinatoren komfortabel verwenden zu können, werden nun noch charakteristische Gleichungen bewiesen, die deren Wirkung auf bestimmte Konstruktoren beschreiben. Für die Kombinatoren term_rec und term_list_rec sind dies die bereits aus Abschnitt 3.3 bekannten Gleichungen

$$\mathsf{term_rec}\ f_1\ \dots\ f_4\ (\mathsf{Var}\ a) = f_1\ a$$
$$\mathsf{term_rec}\ f_1\ \dots\ f_4\ (\mathsf{App}\ b\ ts) = f_2\ b\ ts\ (\mathsf{term_list_rec}\ f_1\ \dots\ f_4\ ts)$$

$$\mathsf{term_list_rec}\ f_1\ \dots\ f_4\ \mathsf{Nil} = f_3$$
$$\mathsf{term_list_rec}\ f_1\ \dots\ f_4\ (\mathsf{Cons}\ t\ ts) = \\ f_4\ t\ ts\ (\mathsf{term_rec}\ f_1\ \dots\ f_4\ t)\ (\mathsf{term_list_rec}\ f_1\ \dots\ f_4\ ts)$$

Wir erläutern hier exemplarisch den Beweis von

$$\mathsf{term_rec}\ f_1\ \dots\ f_4\ (\mathsf{App}\ b\ ts) = f_2\ b\ ts\ (\mathsf{term_list_rec}\ f_1\ \dots\ f_4\ ts)$$

Durch Anwendung der Definition von term_rec und term_list_rec erhält man das Beweisziel

$$\varepsilon y.\ (\mathsf{App}\ b\ ts,\ y) \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4 = \\ f_2\ b\ ts\ (\varepsilon y'.\ (ts,\ y') \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4)$$

Mit Hilfe der Regel

$$\frac{\exists_1 y.\ P\ y \quad P\ a}{(\varepsilon y.\ P\ y) = a}\ (\text{select1_equality})$$

und des soeben bewiesenen Theorems

$$\exists_1 y.\ (\mathsf{App}\ b\ ts,\ y) \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4$$

erhält man das neue Beweisziel

$$(\mathsf{App}\ b\ ts,\ f_2\ b\ ts\ (\varepsilon y'.\ (ts,\ y') \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4)) \in \mathsf{term_rec_set}\ f_1\ \dots\ f_4$$

Es ist also noch zu zeigen, daß der auf der rechten Seite der zu beweisenden charakteristischen Gleichung angegebene Wert tatsächlich in Relation mit $\mathsf{App}\ b\ ts$ steht. Wir wenden nun die entsprechende Einführungsregel für die Menge term_rec_set an und erhalten so das Beweisziel

$$(ts,\ \varepsilon y'.\ (ts,\ y') \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4) \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4$$

das mit Hilfe des Theorems

$$P\ (\varepsilon y.\ P\ y) = \exists y.\ P\ y \qquad (\text{select_eq_Ex})$$

und $\exists y.\ (ts,\ y) \in \mathsf{term_list_rec_set}\ f_1\ \dots\ f_4$ gelöst werden kann.

Allgemeines Verfahren Wir beweisen

$$\begin{array}{l} t_j_\mathsf{rec}\ f_1^1\ \ldots\ f_{k_n}^n\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots) = \\ \quad f_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ (t_{j'}_\mathsf{rec}\ f_1^1\ \ldots\ f_{k_n}^n\ x)\ \ldots \end{array}$$

Die Anwendung der Definition von t_j_rec und $t_{j'}_\mathsf{rec}$ ergibt das Beweisziel

$$\begin{array}{l} (\varepsilon y.\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ y) \in t_j_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n) = \\ \quad f_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ (\varepsilon y'.\ (x,\ y') \in t_{j'}_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n)\ \ldots \end{array}$$

Aufgrund der Regel (select1_equality) und des soeben bewiesenen Theorems

$$\exists_1 y.\ (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots,\ y) \in t_j_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$$

erhält man das neue Beweisziel

$$\begin{array}{l} (C_i^j\ \ldots\ x\ \ldots\ x'\ \ldots, \\ \quad f_i^j\ \ldots\ x\ \ldots\ x'\ \ldots\ (\varepsilon y'.\ (x,\ y')\ \in t_{j'}_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n)\ \ldots) \in t_j_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \end{array}$$

Unter Verwendung der entsprechenden Einführungsregel für die Menge $t_j_\mathsf{rec_set}$ ergeben sich die Beweisziele

$$\begin{array}{ll} (\star) & (x,\ \varepsilon y'.\ (x,\ y') \in t_{j'}_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n) \in t_{j'}_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n \\ & \quad \vdots \end{array}$$

wobei $(\star)$ mittels (select_eq_Ex) und $\exists y.\ (x,\ y) \in t_{j'}_\mathsf{rec_set}\ f_1^1\ \ldots\ f_{k_n}^n$ gelöst werden kann.

4.6.3 Case-Kombinator

Für $1 \leq j < \nu_1$ läßt sich der Kombinator t_j_case mit den charakteristischen Gleichungen der Form

$$t_j_\mathsf{case}_j\ g_1\ \ldots\ g_{k_j}\ \left(C_i^j\ x_1\ \ldots\ x_{m_i^j}\right) = g_i\ x_1\ \ldots\ x_{m_i^j}$$

leicht durch

$$t_j_\mathsf{rec}\ f_1^1\ \ldots\ f_{k_1}^1\ \ldots\ f_1^n\ \ldots\ f_{k_n}^n$$

ausdrücken, indem

$$f_i^j = \lambda x_1, \ldots, x_{m_i^j}, y_1, \ldots, y_{l_i^j}.\ g_i\ x_1\ \ldots\ x_{m_i^j}$$

für alle $1 \leq i \leq k_j$ und $f_{i'}^{j'} = \mathsf{arbitrary}$ für alle $j' \neq j$ gesetzt wird. Die Ergebnisse $y_1, \ldots, y_{l_i^j}$ der rekursiven Aufrufe werden hierbei also wegprojiziert. Da deshalb auch keine Rekursion über andere Datentypen $(\alpha_1, \ldots, \alpha_h)t_{j'}$ erfolgt, können die für Konstruktoren dieser Datentypen zuständigen Funktionen $f_{i'}^{j'}$ mit beliebigen Werten belegt werden.

Kapitel 5

Erweiterungsmöglichkeiten

In Kapitel 3 wurden einige Klassen von Datentypen vorgestellt, die sich mit den in Kapitel 4 anschließend dargestellten Verfahren in HOL repräsentieren lassen. Das folgende Kapitel beschäftigt sich nun mit der Frage, welche Datentypen darüberhinaus noch in HOL repräsentierbar sind, wobei insbesondere verschachtelt rekursive Datentypen mit Funktionstypen genauer betrachtet werden sollen. Hierfür werden zunächst grundsätzliche theoretische Einschränkungen der Repräsentierbarkeit von rekursiven Datentypen diskutiert. Sodann wird eine Erweiterung des in Abschnitt 4.2.1 vorgestellten Universums vorgeschlagen. Aufbauend auf diesem Universum wird schließlich exemplarisch die Konstruktion eines solchen Datentyps demonstriert. Die hier beschriebenen Konstruktionsverfahren gehören zwar momentan noch nicht zum Funktionsumfang des im Rahmen dieser Arbeit implementierten Datentyppakets, wurden jedoch in Isabelle/HOL nachvollzogen.

5.1 Verschachtelt rekursive Datentypen mit Funktionstypen

In Abschnitt 3.3 wurde für Datentypdefinitionen als Zulässigkeitsbedingung festgelegt, daß in Argumenttypen von Konstruktoren, die die Form $(\ldots,(\alpha_1,\ldots,\alpha_h)t_j,\ldots)t'$ haben, t' der Konstruktor eines bereits definierten Datentyps sein muß. Dies ist keinesfalls eine willkürliche Einschränkung, sondern stellt sicher, daß der Datentyp tatsächlich konstruierbar ist. Um die Notwendigkeit dieser Bedingung einzusehen, betrachten wir die Typdefinition

$$\begin{array}{llll} \mathsf{datatype} & \alpha\ \mathsf{dt} & = & \mathsf{C}\ \alpha \\ & & \mid & \mathsf{D}\ (\alpha\ \mathsf{dt} \to \mathsf{bool}) \end{array}$$

Wie in Kapitel 4 beschrieben wurde, werden neue Datentypen durch die Angabe ihrer repräsentierenden Mengen definiert. Um also den Typ α dt definieren zu können, muß zunächst eine repräsentierende Menge dt_rep_set gefunden werden. Da der Datentyp α dt die (injektive) Konstruktorfunktion

$$\mathsf{D}\ ::\ (\alpha\ \mathsf{dt} \to \mathsf{bool}) \to \alpha\ \mathsf{dt}$$

besitzen soll, muß eine Injektion von der repräsentierenden Menge des Typs α dt $\to$ bool in die Menge dt_rep_set existieren. Da jede Funktion vom Typ α dt $\to$ bool eine Teilmenge

von α dt charakterisiert, ist die repräsentierenden Menge des Typs α dt $\rightarrow$ bool isomorph zu $\mathcal{P}$(dt_rep_set), es muß also auch eine Injektion von $\mathcal{P}$(dt_rep_set) in die Menge dt_rep_set existieren. Dies steht jedoch im Widerspruch zu folgendem Theorem von *Cantor*:

Theorem 5.1 (Cantor)
Für keine Menge S existiert eine Surjektion $f : S \rightarrow \mathcal{P}(S)$ oder eine Injektion $g : \mathcal{P}(S) \rightarrow S$.

Beweis

1. Angenommen, es gibt eine Surjektion $f : S \rightarrow \mathcal{P}(S)$. Wir definieren nun

$$R = \{x \in S \mid x \notin f(x)\}$$

   Da $R \in \mathcal{P}(S)$ und f surjektiv ist, gibt es ein $r \in S$ mit $R = f(r)$. Gilt nun $r \in R$, so folgt $r \notin f(r) = R$. Gilt $r \notin R = f(r)$, so folgt $r \in R$. Es kann also keine derartige Surjektion existieren.

2. Angenommen, es gibt eine Injektion $g : \mathcal{P}(S) \rightarrow S$. Dann existiert eine Surjektion $f : S \rightarrow \mathcal{P}(S)$, nämlich $f = \mathsf{inv}\ g$, wobei $\mathsf{inv}\ g = (\lambda y.\ \varepsilon x.\ g(x) = y)$. Da keine Surjektion $f : S \rightarrow \mathcal{P}(S)$ existiert, kann es also auch keine Injektion g geben.

□

Trotz dieser Erkenntnis ist es möglich, bestimmte verschachtelt rekursive Datentypen mit Funktionstypen zu konstruieren. Es muß hierbei sichergestellt sein, daß die betreffende Datentypdefinition nur Typausdrücke enthält, in denen die gerade definierten rekursiven Typen $(\alpha_1, \ldots, \alpha_h)t_j$ ausschließlich *strikt positiv* vorkommen, was wie folgt definiert ist:

Definition 5.2 (Strikt positives Vorkommmen von Typen)
Ein Typ τ' kommt in einem Typausdruck τ *strikt positiv* vor, falls eine der folgenden Bedingungen erfüllt ist:

- $\tau = \tau'$
- $\tau = \tau_1 \rightarrow \tau_2$ und τ' kommt in τ_2 strikt positiv vor
- $\tau = (\tau_1, \ldots, \tau_{h'})t'$, wobei t' der Konstruktor eines bereits definierten Datentyps mit den Typparametern $\beta_1, \ldots, \beta_{h'}$ ist, τ' strikt positiv in τ_i vorkommt und der Typparameter β_i in allen Typausdrücken in der Definition von t' nur strikt positiv vorkommt.

□

Demnach ist auch

$$\begin{array}{lcl} \mathsf{datatype}\ \ \alpha\ \mathsf{dt} & = & \mathsf{C}\ \alpha \\ & \mid & \mathsf{D}\ ((\alpha\ \mathsf{dt} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}) \end{array}$$

unzulässig, denn gäbe es eine Injektion

$$\mathsf{D} :: ((\alpha\ \mathsf{dt} \rightarrow \mathsf{bool}) \rightarrow \mathsf{bool}) \rightarrow \alpha\ \mathsf{dt}$$

könnte man mit Hilfe der Injektion

$$(\lambda x.\ (\lambda y.\ x = y))\ ::\ (\alpha\ \mathsf{dt} \to \mathsf{bool}) \to ((\alpha\ \mathsf{dt} \to \mathsf{bool}) \to \mathsf{bool})$$

die Injektion

$$\mathsf{D} \circ (\lambda x.\ (\lambda y.\ x = y))\ ::\ (\alpha\ \mathsf{dt} \to \mathsf{bool}) \to \alpha\ \mathsf{dt}$$

konstruieren, was ebenfalls einen Widerspruch zu Cantors Theorem darstellen würde.

Ein Beispiel für einen gemäß dieser Definition zulässigen Datentyp ist der Typ $(\alpha, \beta, \gamma)\mathsf{tree}$ der Bäume mit potentiell unendlichem Verzweigungsgrad $|\gamma|$

$$\begin{array}{lll} \mathsf{datatype}\quad (\alpha,\beta,\gamma)\mathsf{tree} & = & \mathsf{Atom}\ \alpha \\ & \mid & \mathsf{Branch}\ \beta\ (\gamma \to (\alpha,\beta,\gamma)\mathsf{tree}) \end{array}$$

Die Induktionsregel für diesen Typ hat die Gestalt

$$\frac{\forall a.\ P\ (\mathsf{Atom}\ a) \quad \forall b, f.\ (\forall x.\ P\ (f\ x)) \Longrightarrow P\ (\mathsf{Branch}\ b\ f)}{P\ t}$$

Der Kombinator

$$\begin{array}{lll} \mathsf{tree_rec} & :: & (\alpha \to \delta) \to \\ & & (\beta \to (\gamma \to (\alpha,\beta,\gamma)\mathsf{tree}) \to (\gamma \to \delta) \to \delta) \to \\ & & (\alpha,\beta,\gamma)\mathsf{tree} \to \delta \end{array}$$

für primitive Rekursion besitzt hierbei die charakteristischen Gleichungen

$$\begin{array}{lll} \mathsf{tree_rec}\ f_1\ f_2\ (\mathsf{Atom}\ a) & = & f_1\ a \\ \mathsf{tree_rec}\ f_1\ f_2\ (\mathsf{Branch}\ b\ f) & = & f_2\ b\ f\ ((\mathsf{tree_rec}\ f_1\ f_2) \circ f) \end{array}$$

Die genaueren Details der Konstruktion dieses Typs werden im folgenden Abschnitt beschrieben. Auch in [Gunter, 1993] wird auf die Handhabung derartiger Typen in HOL eingegangen.

Bemerkung 5.3
Die durch das Theorem von Cantor gegebenen Einschränkungen gelten im Kontext von klassischer Mengenlehre, wobei $\to$ den *vollen Funktionenraum* bezeichnet. Verwendet man anstelle des vollen Funktionenraums den *stetigen Funktionenraum*, so gelten diese Einschränkungen nicht mehr. Derartige Konstruktionen werden in der Literatur oft mit dem Begriff *Scott Domains* bezeichnet.

5.2 Ein Universum für Datentypen mit beliebigem Verzweigungsgrad

Der in Abschnitt 4.2.1 eingeführte Typ zur Darstellung von Bäumen wird nun wie folgt verändert:

$$\begin{array}{lll} (\alpha,\beta)\mathsf{node} & = & (\mathsf{nat} \to (\beta + \mathsf{nat})) \times (\alpha + \mathsf{nat}) \\ (\alpha,\beta)\mathsf{item} & = & ((\alpha,\beta)\mathsf{node})\mathsf{set} \end{array}$$

Als Labels für Verzweigungen werden hier nun neben Elementen vom Typ nat auch Elemente des Typs β zugelassen. Bei der Repräsentation von Datentypen, in deren Definition Typausdrücke der Form $\tau_j \to (\alpha_1, \ldots, \alpha_h)t_i$ mit $1 \le j \le \xi$ vorkommen, wird für β die Summe $\tau_1 + \cdots + \tau_\xi$ aller Verzweigungstypen $\tau_1, \ldots, \tau_\xi$ gewählt. Bei einem komplexeren Typausdruck der Form $\tau'_1 \to \cdots \to \tau'_\mu \to (\alpha_1, \ldots, \alpha_h)t_i$ wird $\tau'_1 \times \cdots \times \tau'_\mu$ als Verzweigungstyp verwendet.

Die Definition der in Abschnitt 4.2.1 angegebenen Funktionen auf item lautet nun

$$\begin{array}{lcl}
\mathsf{push} & :: & (\beta + \mathsf{nat}) \to (\alpha, \beta)\mathsf{node} \to (\alpha, \beta)\mathsf{node} \\
\mathsf{push}\ c\ n & \equiv & (\mathsf{nat_case}\ c\ (\mathsf{fst}\ n), \mathsf{snd}\ n) \\
\mathsf{Scons} & :: & (\alpha, \beta)\mathsf{item} \to (\alpha, \beta)\mathsf{item} \to (\alpha, \beta)\mathsf{item} \\
\mathsf{Scons}\ i_1\ i_2 & \equiv & (\mathsf{push}\ (\mathsf{Inr}\ 1)\ \text{``}\ i_1) \cup (\mathsf{push}\ (\mathsf{Inr}\ 2)\ \text{``}\ i_2) \\
\mathsf{Leaf} & :: & (\alpha, \beta) \to (\alpha, \beta)\mathsf{item} \\
\mathsf{Leaf}\ a & \equiv & \{(\lambda x.\ \mathsf{Inr}\ 0, \mathsf{Inl}\ a)\} \\
\mathsf{Numb} & :: & \mathsf{nat} \to (\alpha, \beta)\mathsf{item} \\
\mathsf{Numb}\ k & \equiv & \{(\lambda x.\ \mathsf{Inr}\ 0, \mathsf{Inr}\ k)\}
\end{array}$$

Zur Repräsentation von Funktionen des Typs $\beta \to (\alpha, \beta)$item durch ein Element vom Typ (α, β)item definieren wir

$$\begin{array}{lcl}
\mathsf{Lim} & :: & (\beta \to (\alpha, \beta)\mathsf{item}) \to (\alpha, \beta)\mathsf{item} \\
\mathsf{Lim}\ f & \equiv & \bigcup\{z \mid \exists x.\ z = \mathsf{push}\ (\mathsf{Inl}\ x)\ \text{``}\ (f\ x)\}
\end{array}$$

Durch Lim f wird die Menge von Knoten bezeichnet, die sich ergibt, wenn für alle x jeweils an alle Pfade der Knoten in der Menge $f\ x$ das Präfix x angefügt wird und die so entstehenden Knotenmengen vereinigt werden. Dies ist in Abbildung 5.1 graphisch dargestellt.

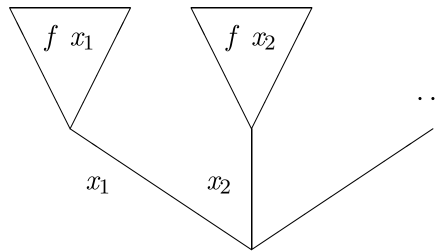


Abbildung 5.1: Repräsentation von Funktionen durch Bäume

Injektivität von Lim Mittels elementarer Eigenschaften von Funktionen und Mengen läßt sich nun für Lim die Eigenschaft

$$\mathsf{Lim}\ f = \mathsf{Lim}\ g \Longrightarrow f = g$$

nachweisen.

Beweis Der Beweis erfolgt durch Widerspruch: Angenommen, es gilt Lim f = Lim g und $f \neq g$, d.h. es existiert ein x mit $f\ x \neq g\ x$. Da es sich bei $f\ x$ und $g\ x$ um Mengen handelt, bedeutet dies nun, daß ein Element y existiert, für das

- $y \in f\ x$ und $y \notin g\ x$ oder
- $y \in g\ x$ und $y \notin f\ x$

gilt. Wir betrachten hier exemplarisch den ersten dieser beiden Fälle: Gilt $y \in f\ x$, so folgt $\mathsf{push}\ (\mathsf{Inl}\ x)\ y \in \mathsf{Lim}\ f$ aufgrund der Definition von Lim und damit auch $\mathsf{push}\ (\mathsf{Inl}\ x)\ y \in \mathsf{Lim}\ g$. Dies bedeutet wiederum wegen der Definition von Lim, daß $y' \in g\ x'$ und $\mathsf{push}\ (\mathsf{Inl}\ x)\ y = \mathsf{push}\ (\mathsf{Inl}\ x')\ y'$ für geeignete x' und y' gelten muß. Aufgrund der Injektivität von push und Inl erhält man hieraus $x = x'$ und $y = y'$, was zusammen mit $y' \in g\ x'$ und $y \notin g\ x$ zu einem Widerspruch führt. □

5.3 Konstruktion eines Datentyps mit unendlichem Verzweigungsgrad

Mit Hilfe der soeben eingeführten Konstruktionselemente kann nun der Typ

$$\begin{array}{lll} \mathsf{datatype}\quad (\alpha,\beta,\gamma)\mathsf{tree} & = & \mathsf{Atom}\ \alpha \\ & \mid & \mathsf{Branch}\ \beta\ (\gamma \to (\alpha,\beta,\gamma)\mathsf{tree}) \end{array}$$

konstruiert werden.

5.3.1 Repräsentierende Menge

Die repräsentierende Menge $\mathsf{tree_rep_set}$ ist hierbei vom Typ $((\alpha+\beta,\ \gamma)\mathsf{item})\mathsf{set}$ und wird induktiv durch die Regeln

$$\frac{}{\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a)) \in \mathsf{tree_rep_set}} \qquad \frac{g \in \mathsf{Funs}\ \mathsf{tree_rep_set}}{\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ g)) \in \mathsf{tree_rep_set}}$$

definiert, wobei die in der zweiten Regel verwendete monotone Funktion Funs die Definition

$$\begin{array}{lll} \mathsf{Funs} & :: & \beta\ \mathsf{set} \to (\alpha \to \beta)\mathsf{set} \\ \mathsf{Funs}\ S & \equiv & \{g \mid \mathsf{range}\ g \subseteq S\} \end{array}$$

hat. Mit der Prämisse $g \in \mathsf{Funs}\ \mathsf{tree_rep_set}$ der zweiten Einführungsregel wird also ausgedrückt, daß alle Bildelemente der Funktion g Repräsentationen von Bäumen sind.

5.3.2 Konstruktoren

Die Konstruktoren

$$\begin{array}{lll} \mathsf{Atom} & :: & \alpha \to (\alpha,\beta,\gamma)\mathsf{tree} \\ \mathsf{Branch} & :: & \beta \to (\gamma \to (\alpha,\beta,\gamma)\mathsf{tree}) \to (\alpha,\beta,\gamma)\mathsf{tree} \end{array}$$

werden durch

$$\begin{array}{lll} \mathsf{Atom}\ a & \equiv & \mathsf{Abs_tree}\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a))) \\ \mathsf{Branch}\ b\ f & \equiv & \mathsf{Abs_tree}\ (\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ (\mathsf{Rep_tree} \circ f)))) \end{array}$$

definiert. Im Falle des Konstruktors Branch wird hierbei die Repräsentation des Arguments $f :: \gamma \to (\alpha, \beta, \gamma)\mathsf{tree}$ berechnet, indem zuerst durch die Komposition von Rep_tree mit f die Repräsentationen der Bildelemente von f berechnet werden. Die Repräsentation der sich so ergebenden Funktion vom Typ $\gamma \to (\alpha + \beta, \gamma)\mathsf{item}$ wird dann mittels Lim berechnet. Beim Beweis der Injektivität des Konstruktors Branch wird zusätzlich die Injektivität von Lim benötigt.

5.3.3 Induktionsregel

Die in 5.1 angegebene Induktionsregel kann nun, ähnlich wie in Abschnitt 4.5.3 beschrieben, mit Hilfe der Induktionsregel

$$\frac{\begin{array}{l}\forall a.\ Q\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a)))\\ \forall b, g.\ \mathsf{range}\ g \subseteq \mathsf{tree_rep_set} \cap \{x \mid Q\ x\} \Longrightarrow Q\ (\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ g)))\end{array}}{u \in \mathsf{tree_rep_set} \Longrightarrow Q\ u}$$

für die Menge tree_rep_set bewiesen werden. Hierbei muß unter den Annahmen

(i) $\forall a.\ \mathsf{Atom}\ a$

(ii) $\forall b, f.\ (\forall x.\ P\ (f\ x)) \Longrightarrow P\ (\mathsf{Branch}\ b\ f)$

das Beweisziel $P\ t$ gezeigt werden. Mittels (Rep_tree_inverse) und (Rep_tree) erhält man hieraus das Beweisziel

$$\mathsf{Rep_tree}\ t \in \mathsf{tree_rep_set} \Longrightarrow P\ (\mathsf{Abs_tree}\ (\mathsf{Rep_tree}\ t))$$

Die Anwendung der obigen Induktionsregel mit $Q = P \circ \mathsf{Abs_tree}$ und $u = \mathsf{Rep_tree}\ t$ liefert nun die Beweisziele

(1) $\forall a.\ P\ (\mathsf{Abs_tree}\ (\mathsf{In0}\ (\mathsf{Leaf}\ (\mathsf{Inl}\ a))))$

(2) $\forall b, g.\ \mathsf{range}\ g \subseteq \mathsf{tree_rep_set} \cap \{x \mid P\ (\mathsf{Abs_tree}\ x)\} \Longrightarrow$
$P\ (\mathsf{Abs_tree}\ (\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ g))))$

Wir erläutern hier exemplarisch die Behandlung von (2). Aufgrund des Theorems

$$\mathsf{range}\ g \subseteq \mathsf{tree_rep_set} \Longrightarrow \mathsf{Rep_tree} \circ (\mathsf{Abs_tree} \circ g) = g$$

das man leicht mittels Extensionalität aus (Abs_tree_inverse) erhält, ergibt sich aus (2) das neue Beweisziel

$$\begin{array}{l}\forall b, g.\ \mathsf{range}\ g \subseteq \mathsf{tree_rep_set} \cap \{x \mid P\ (\mathsf{Abs_tree}\ x)\} \Longrightarrow\\ \quad P\ (\mathsf{Abs_tree}\ (\mathsf{In1}\ (\mathsf{Scons}\ (\mathsf{Leaf}\ (\mathsf{Inr}\ b))\ (\mathsf{Lim}\ (\mathsf{Rep_tree} \circ (\mathsf{Abs_tree} \circ g))))))\end{array}$$

Aufgrund der Definition des Konstruktors Branch läßt sich dies zu

$$\forall b, g.\ \mathsf{range}\ g \subseteq \mathsf{tree_rep_set} \cap \{x \mid P\ \mathsf{Abs_tree}\ x)\} \Longrightarrow P\ (\mathsf{Branch}\ b\ (\mathsf{Abs_tree} \circ g))$$

umformen. Unter Verwendung von Voraussetzung (ii) erhält man das Beweisziel

$$\forall b, g, x'.\ \mathsf{range}\ g \subseteq \mathsf{tree_rep_set} \cap \{x \mid P\ (\mathsf{Abs_tree}\ x)\} \Longrightarrow P\ (\mathsf{Abs_tree}\ (g\ x'))$$

das sich leicht mit Hilfe elementarer Eigenschaften von range zeigen läßt.

5.3.4 Primitive Rekursion

Der Graph des Kombinators

$$\begin{array}{lll} \mathsf{tree_rec} & :: & (\alpha \to \delta) \to \\ & & (\beta \to (\gamma \to (\alpha,\beta,\gamma)\mathsf{tree}) \to (\gamma \to \delta) \to \delta) \to \\ & & (\alpha,\beta,\gamma)\mathsf{tree} \to \delta \end{array}$$

ist durch die Menge

$$\mathsf{tree_rec_set}\ f_1\ f_2 \ ::\ ((\alpha,\beta,\gamma)\mathsf{tree} \times \delta)\mathsf{set}$$

gegeben, die induktiv durch die Regeln

$$\frac{}{(\mathsf{Atom}\ a,\ f_1\ a) \in \mathsf{tree_rec_set}\ f_1\ f_2} \qquad \frac{f' \in \mathsf{compose}\ f\ (\mathsf{tree_rec_set}\ f_1\ f_2)}{(\mathsf{Branch}\ b\ f,\ f_2\ b\ f\ f') \in \mathsf{tree_rec_set}\ f_1\ f_2}$$

definiert ist. Die monotone Funktion compose ist hierbei definiert durch

$$\begin{array}{lll} \mathsf{compose} & :: & (\alpha \to \beta) \to (\beta \times \gamma)\mathsf{set} \to (\alpha \to \gamma)\mathsf{set} \\ \mathsf{compose}\ f\ R & \equiv & \{f' \mid \forall x.\ (f\ x,\ f'\ x) \in R\} \end{array}$$

Die zweite Einführungsregel drückt aus, daß die durch die Relation tree_rec_set repräsentierte Funktion rekursiv auf alle Bildelemente der Funktion f, also auf alle Unterbäume des Baums Branch $b\ f$ angewendet werden muß. Ergebnis dieser rekursiven Anwendung ist die Funktion f', die man durch die Komposition der Funktion f mit der Relation tree_rec_set erhält. Dieser Sachverhalt wird mit Hilfe von compose ausgedrückt. Die etwas umständlich anmutende Formulierung ist notwendig, da die Einführungsregeln induktiver Mengen eine bestimmte, in Abschnitt 2.3.2 festgelegte Struktur aufweisen müssen, um die Monotonie der sich aus den Regeln ergebenden Funktion zu gewährleisten.

Totalität und Eindeutigkeit Mit Hilfe der in Abschnitt 5.3.3 bewiesenen Induktionsregel für (α,β,γ)tree kann nun, ähnlich wie in Abschnitt 4.6.2 gezeigt werden, daß die soeben definierte Relation linkstotal und rechtseindeutig ist, d.h.

$$\exists_1 y.\ (x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2$$

Induktion über x liefert die Beweisziele

(1) $\forall a.\ \exists_1 y.\ (\mathsf{Atom}\ a,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2$

(2) $\forall b, f.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow$
$\exists_1 y.\ (\mathsf{Branch}\ b\ f,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2)$

wobei wir wieder nur die Behandlung von (2) betrachten. Wir wenden auf (2) zunächst die Einführungsregel für $\exists_1$ an, wobei wir als „Zeugen" für die Existenzaussage den Term

$$f_2\ b\ f\ ((\lambda x.\ \varepsilon y.\ (x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \circ f)$$

angeben. Dies liefert die neuen Beweisziele

$$\begin{array}{ll}(3) & \forall b, f.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow \\ & \quad (\mathsf{Branch}\ b\ f,\ f_2\ b\ f\ ((\lambda x.\ \varepsilon y.\ (x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \circ f)) \in \mathsf{tree_rec_set}\ f_1\ f_2 \end{array}$$

$$\begin{array}{ll}(4) & \forall b, f, z.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \wedge \\ & \quad (\mathsf{Branch}\ b\ f,\ z) \in \mathsf{tree_rec_set}\ f_1\ f_2 \Longrightarrow \\ & \qquad z = f_2\ b\ f\ ((\lambda x.\ \varepsilon y.\ (x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \circ f) \end{array}$$

Aus (3) erhält man nach Anwendung der entsprechenden Einführungsregel für tree_rec_set und der Definition von compose schließlich

$$\begin{array}{l}\forall b, f.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow \\ \quad \forall x.\ (f\ x,\ \varepsilon y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \in \mathsf{tree_rec_set}\ f_1\ f_2 \end{array}$$

was sich leicht mittels (select_eq_Ex) lösen läßt. Aus (4) erhält man mit Hilfe der Eliminationsregel

$$\frac{\begin{array}{l} r \in \mathsf{tree_rec_set}\ f_1\ f_2 \\ \forall a.\ r = (\mathsf{Atom}\ a,\ f_1\ a) \Longrightarrow P \\ \forall b, f', g.\ r = (\mathsf{Branch}\ b\ f',\ f_2\ b\ f'\ g) \wedge (\forall x.\ (f'\ x,\ g\ x) \in \mathsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow P \end{array}}{P}$$

für tree_rec_set die Beweisziele

$$\begin{array}{ll}(5) & \forall b, f, z, a.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \wedge \\ & \quad (\mathsf{Branch}\ b\ f,\ z) = (\mathsf{Atom}\ a,\ f_1\ a) \Longrightarrow \\ & \qquad z = f_2\ b\ f\ ((\lambda x.\ \varepsilon y.\ (x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \circ f) \end{array}$$

$$\begin{array}{ll}(6) & \forall b, f, z, b', f', g.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \wedge \\ & \quad (\mathsf{Branch}\ b\ f,\ z) = (\mathsf{Branch}\ b'\ f',\ f_2\ b'\ f'\ g) \wedge \\ & \qquad (\forall x.\ (f'\ x,\ g\ x) \in \mathsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow \\ & \qquad\quad z = f_2\ b\ f\ ((\lambda x.\ \varepsilon y.\ (x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \circ f) \end{array}$$

wobei (5) wegen Branch $\ldots \neq$ Atom $\ldots$ trivial ist. Unter Ausnutzung der Injektivität von $(_, _)$ und Branch ergibt sich aus (6) das Beweisziel

$$\begin{array}{l}\forall b, f, z, b', f', g.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \wedge \\ \quad (\forall x.\ (f'\ x,\ g\ x) \in \mathsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow \\ \qquad f_2\ b'\ f'\ g = f_2\ b'\ f'\ ((\lambda x.\ \varepsilon y.\ (x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \circ f') \end{array}$$

woraus man mit Hilfe von Kongruenz und Extensionalität das Beweisziel

$$\begin{array}{l}\forall f', g, x'.\ (\forall x.\ \exists_1 y.\ (f\ x,\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2) \wedge \\ \quad (\forall x.\ (f'\ x,\ g\ x) \in \mathsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow \\ \qquad g\ x' = \varepsilon y.\ (f'\ x',\ y) \in \mathsf{tree_rec_set}\ f_1\ f_2 \end{array}$$

erhält. Durch Anwendung von (select1_equality) erhält man schließlich die leicht lösbaren Beweisziele

(7) $\forall f', g, x'.\ (\forall x.\ \exists_1 y.\ (f\ \ x,\ \ y) \in \textsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow$
$\exists_1 y.\ (f'\ \ x',\ \ y) \in \textsf{tree_rec_set}\ f_1\ f_2$

(8) $\forall f', g, x'.\ (\forall x.\ (f'\ \ x,\ \ g\ \ x) \in \textsf{tree_rec_set}\ f_1\ f_2) \Longrightarrow$
$(f'\ \ x',\ \ g\ \ x') \in \textsf{tree_rec_set}\ f_1\ f_2$

Die Definition des Kombinators tree_rec mittels tree_rec_set sowie die Ableitung der charakteristischen Gleichungen für tree_rec erfolgt nahezu genauso wie in Abschnitt 4.5.3.

Kapitel 6

Vergleich mit anderen Arbeiten

Die in Kapitel 3 und 4 beschriebenen Ansätze zur Realisierung von Datentypen sollen nun mit anderen Arbeiten verglichen werden. Hierfür geben wir zunächst einen kurzen Überblick über bisher vorhandene Datentyppakete. Eines dieser Pakete wird dann in Hinblick auf die Formulierung der charakteristischen Theoreme und die zum Einsatz kommenden Konstruktionsverfahren mit dem im Rahmen dieser Diplomarbeit implementierten Paket verglichen.

6.1 Andere auf HOL basierende Datentyppakete

Das erste definitorische Datentyppaket für das HOL-System wurde 1989 von Tom Melham [Melham, 1989] implementiert. Auch hier wird zunächst ein allgemeiner Typ von Bäumen wie in Abschnitt 4.2.1 definiert, mit dessen Hilfe alle Datentypen repräsentiert werden. Dieses Paket wurde von Elsa Gunter [Gunter, 1992] um verschränkt und verschachtelt rekursive Datentypen erweitert. In [Gunter, 1993] wird beschrieben, wie dieses Paket auf unendlich verzweigende Datentypen ähnlich wie in Kapitel 5 erweitert werden kann. Von Lawrence C. Paulson wurde in [Paulson, 1994] vorgeschlagen, die repräsentierenden Mengen von Datentypen durch induktive Definitionen zu konstruieren, die mittels einer allgemeinen Theorie von Fixpunkten und monotonen Funktionen realisiert werden. Dieser Ansatz wurde auch von John Harrison [Harrison, 1995] verfolgt, der für seinen Theorembeweiser HOL Light ebenfalls ein Datentyppaket implementierte. Sowohl Gunter als auch Harrison realisieren verschachtelt rekursive Datentypen durch Auffalten. Anhang B enthält einen Vergleich von John Harrisons Paket mit dem im Rahmen dieser Diplomarbeit implementierten Paket in Hinblick auf die Beweislaufzeiten für einige ausgewählte Datentypen.

Von Norbert Völker [Völker, 1995, Völker, 1997] wurde für Isabelle/HOL ein Datentyppaket implementiert, mit dem auch verschachtelt rekursive, jedoch keine verschränkt rekursiven Datentypen definiert werden können. Anstatt diese Datentypen durch Auffaltung wie in Kapitel 3 und 4 zu realisieren, wird hier ein etwas anderer Ansatz gewählt, der in den folgenden Abschnitten kurz beschrieben werden soll.

6.2 Alternative Formulierung charakteristischer Theoreme

Für verschachtelt rekursive Datentypen wie

$$\begin{array}{lcl} \mathsf{datatype}\quad (\alpha,\beta)\mathsf{term} & = & \mathsf{Var}\ \alpha \\ & \mid & \mathsf{App}\ \beta\ (((\alpha,\beta)\mathsf{term})\mathsf{list}) \end{array}$$

wird in [Völker, 1995, Völker, 1997] für Induktionsregeln sowie für die charakteristischen Gleichungen der Rekursionskombinatoren eine etwas kompaktere Formulierung präsentiert. Es werden hierbei zahlreiche vordefinierte Standardfunktionen für Datentypen genutzt. Für den Typ list sind dies die Funktionen

$$\begin{array}{lcl} \mathsf{map} & :: & (\alpha \to \beta) \to \alpha\ \mathsf{list} \to \beta\ \mathsf{list} \\ \mathsf{list_all} & :: & (\alpha \to \mathsf{bool}) \to \alpha\ \mathsf{list} \to \mathsf{bool} \\ \mathsf{set_of_list} & :: & \alpha\ \mathsf{list} \to \alpha\ \mathsf{set} \end{array}$$

mit den charakteristischen Gleichungen

$$\begin{array}{lcl} \mathsf{map}\ f\ \mathsf{Nil} & = & \mathsf{Nil} \\ \mathsf{map}\ f\ (\mathsf{Cons}\ x\ xs) & = & \mathsf{Cons}\ (f\ x)\ (\mathsf{map}\ f\ xs) \\ \mathsf{list_all}\ Q\ \mathsf{Nil} & = & \mathsf{True} \\ \mathsf{list_all}\ Q\ (\mathsf{Cons}\ x\ xs) & = & Q\ x \wedge (\mathsf{list_all}\ Q\ xs) \\ \mathsf{set_of_list}\ \mathsf{Nil} & = & \{\} \\ \mathsf{set_of_list}\ (\mathsf{Cons}\ x\ xs) & = & \{x\} \cup (\mathsf{set_of_list}\ xs) \end{array}$$

Unter Verwendung der Funktion list_all kann die Induktion über term nun durch die Regel

$$\frac{\forall a.\ P\ (\mathsf{Var}\ a) \quad \forall b, ts.\ \mathsf{list_all}\ P\ ts \Longrightarrow P\ (\mathsf{App}\ b\ ts)}{P\ t}$$

bzw. mit set_of_list durch

$$\frac{\forall a.\ P\ (\mathsf{Var}\ a) \quad \forall b, ts.\ \mathsf{set_of_list}\ ts \subseteq \{x \mid P\ x\} \Longrightarrow P\ (\mathsf{App}\ b\ ts)}{P\ t}$$

ausgedrückt werden. Der Kombinator

$$\mathsf{term_rec}\ ::\ (\alpha \to \gamma) \to (\beta \to (((\alpha,\beta)\mathsf{term})\mathsf{list}) \to \gamma\ \mathsf{list} \to \gamma) \to (\alpha,\beta)\mathsf{term} \to \gamma$$

für primitive Rekursion läßt sich hierbei mit Hilfe von map durch die Gleichungen

$$\begin{array}{lcl} \mathsf{term_rec}\ f_1\ f_2\ (\mathsf{Var}\ a) & = & f_1\ a \\ \mathsf{term_rec}\ f_1\ f_2\ (\mathsf{App}\ b\ ts) & = & f_2\ b\ ts\ (\mathsf{map}\ (\mathsf{term_rec}\ f_1\ f_2)\ ts) \end{array}$$

charakterisieren. Auch wenn die obige Induktionsregel im Vergleich zur „aufgefalteten“ Induktionsregel aus Abschnitt 3.3 etwas kompakter wirkt, kann deren Anwendung in Beweisen

gelegentlich noch eine zusätzliche Induktion über den Typ list erfordern. Die kompakte Induktionsregel für term läßt sich leicht mit Hilfe der „aufgefalteten" Induktionsregel herleiten, indem P_1 mit P und P_2 mit list_all P instantiiert wird.

Beweis Angenommen, es gilt $\forall a.\ P\ (\mathsf{Var}\ a)$ und $\forall b, ts.\ \mathsf{list_all}\ P\ ts \Longrightarrow P\ (\mathsf{App}\ b\ ts)$. Ausgehend vom Beweisziel

$$P\ t \wedge \mathsf{list_all}\ P\ ts$$

liefert die Anwendung der simultanen Induktionsregel aus Abschnitt 3.3 die neuen Beweisziele

(1) $\forall a.\ P\ (\mathsf{Var}\ a)$

(2) $\forall b, ts.\ \mathsf{list_all}\ P\ ts \Longrightarrow P\ (\mathsf{App}\ b\ ts)$

(3) $\mathsf{list_all}\ P\ \mathsf{Nil}$

(4) $\forall t, ts.\ P\ t \wedge \mathsf{list_all}\ P\ ts \Longrightarrow \mathsf{list_all}\ P\ (\mathsf{Cons}\ t\ ts)$

Die Beweisziele (1) und (2) lassen sich mit Hilfe der Voraussetzungen lösen. Durch Anwendung der charakteristischen Gleichungen von list_all erhält man aus (3) und (4) die trivial beweisbaren Beweisziele True und

$$\forall t, ts.\ P\ t \wedge \mathsf{list_all}\ P\ ts \Longrightarrow P\ t \wedge \mathsf{list_all}\ P\ ts$$

□

Umgekehrt läßt sich aus der kompakten Induktionsregel mit Hilfe von Induktion über list die „aufgefaltete" Regel herleiten. Auch bei den Kombinatoren für primitive Rekursion kann man die kompakte Variante term_rec leicht aus der „aufgefalteten" Variante (im folgenden zur Vermeidung von Verwechslungen mit term_rec' und term_list_rec' bezeichnet) erhalten. Es gilt

$$\mathsf{term_rec}\ f_1\ f_2\ t = \mathsf{term_rec'}\ f_1\ f_2\ \mathsf{Nil}\ (\lambda x, xs.\ \mathsf{Cons})\ t$$

$$\mathsf{map}\ (\mathsf{term_rec}\ f_1\ f_2)\ ts = \mathsf{term_list_rec'}\ f_1\ f_2\ \mathsf{Nil}\ (\lambda x, xs.\ \mathsf{Cons})\ ts$$

wie man durch simultane Induktion über t und ts unter Verwendung der Induktionsregel aus Abschnitt 3.3 nachweisen kann. Umgekehrt gilt

$$\begin{array}{l}\mathsf{term_rec'}\ f_1\ f_2\ f_3\ f_4\ t = \\ \quad \mathsf{term_rec}\ f_1 \\ \quad\quad (\lambda b, ts, xs.\ f_2\ b\ ts\ (\mathsf{list_rec}\ f_3 \\ \quad\quad\quad (\lambda y, ys, z.\ f_4\ (\mathsf{fst}\ y)\ (\mathsf{map}\ \mathsf{fst}\ ys)\ (\mathsf{snd}\ y)\ z)\ (\mathsf{zip}\ ts\ xs)))\ t\end{array}$$

$$\begin{array}{l}\mathsf{term_list_rec'}\ f_1\ f_2\ f_3\ f_4\ ts = \\ \quad \mathsf{list_rec}\ f_3 \\ \quad\quad (\lambda y, ys, z.\ f_4\ (\mathsf{fst}\ y)\ (\mathsf{map}\ \mathsf{fst}\ ys)\ (\mathsf{snd}\ y)\ z)\ (\mathsf{zip}\ ts\ (\mathsf{map} \\ \quad\quad\quad (\mathsf{term_rec}\ f_1 \\ \quad\quad\quad\quad (\lambda b, ts', xs.\ f_2\ b\ ts'\ (\mathsf{list_rec}\ f_3 \\ \quad\quad\quad\quad\quad (\lambda y, ys, z.\ f_4\ (\mathsf{fst}\ y)\ (\mathsf{map}\ \mathsf{fst}\ ys)\ (\mathsf{snd}\ y)\ z)\ (\mathsf{zip}\ ts'\ xs))))\ ts))\end{array}$$

was sich analog beweisen läßt. Für die Funktion

$$\mathsf{zip} :: \alpha\ \mathsf{list} \to \beta\ \mathsf{list} \to (\alpha \times \beta)\mathsf{list}$$

gilt hierbei

$$\begin{array}{lcl} \mathsf{zip\ Nil\ Nil} & = & \mathsf{Nil} \\ \mathsf{zip}\ (\mathsf{Cons}\ x\ xs)\ (\mathsf{Cons}\ y\ ys) & = & \mathsf{Cons}\ (x, y)\ (\mathsf{zip}\ xs\ ys) \end{array}$$

Kommt der Typ term selbst wieder in einer verschachtelt rekursiven Datentypdefinition vor, so benötigt man zur kompakten Formulierung der charakteristischen Theoreme des resultierenden Datentyps auch für term entsprechende Varianten der eingangs erwähnten Standardfunktionen

$$\begin{array}{lcl} \mathsf{map_term} & :: & (\alpha \to \gamma) \to (\beta \to \delta) \to (\alpha, \beta)\mathsf{term} \to (\gamma, \delta)\mathsf{term} \\ \mathsf{term_all} & :: & (\alpha \to \mathsf{bool}) \to (\beta \to \mathsf{bool}) \to (\alpha, \beta)\mathsf{term} \to \mathsf{bool} \\ \mathsf{set_of_term} & :: & (\alpha, \beta)\mathsf{term} \to (\alpha + \beta)\mathsf{set} \end{array}$$

die in diesem Fall durch die Gleichungen

$$\begin{array}{lcl} \mathsf{map_term}\ f_1\ f_2\ (\mathsf{Var}\ a) & = & \mathsf{Var}\ (f_1\ a) \\ \mathsf{map_term}\ f_1\ f_2\ (\mathsf{App}\ b\ ts) & = & \mathsf{App}\ (f_2\ b)\ (\mathsf{map}\ (\mathsf{map_term}\ f_1\ f_2)\ ts) \\ & & \\ \mathsf{term_all}\ Q_1\ Q_2\ (\mathsf{Var}\ a) & = & Q_1\ a \\ \mathsf{term_all}\ Q_1\ Q_2\ (\mathsf{App}\ b\ ts) & = & Q_2\ b \wedge (\mathsf{list_all\ Id}\ (\mathsf{map}\ (\mathsf{term_all}\ Q_1\ Q_2)\ ts)) \\ & & \\ \mathsf{set_of_term}\ (\mathsf{Var}\ a) & = & \{\mathsf{Inl}\ a\} \\ \mathsf{set_of_term}\ (\mathsf{App}\ b\ ts) & = & \{\mathsf{Inr}\ b\} \cup \bigcup(\mathsf{set_of_list}\ (\mathsf{map\ set_of_term}\ ts)) \end{array}$$

charakterisiert werden. Derartige Funktionen lassen sich auf beliebige Datentypen verallgemeinern. Für einen Datentyp $(\alpha_1, \ldots, \alpha_n)t$ haben die entsprechenden Funktionen die Typen

$$\begin{array}{lcl} \mathsf{map_}t & :: & (\alpha_1 \to \beta_1) \to \cdots \to (\alpha_n \to \beta_n) \to (\alpha_1, \ldots, \alpha_n)t \to (\beta_1, \ldots, \beta_n)t \\ t\mathsf{_all} & :: & (\alpha_1 \to \mathsf{bool}) \to \cdots \to (\alpha_n \to \mathsf{bool}) \to (\alpha_1, \ldots, \alpha_n)t \to \mathsf{bool} \\ \mathsf{set_of_}t & :: & (\alpha_1, \ldots, \alpha_n)t \to (\alpha_1 + \cdots + \alpha_n)\mathsf{set} \end{array}$$

Dies gilt allerdings nur für sogenannte *reine Datentypen*, in deren Definition nur Datentypen vorkommen. Andernfalls weisen die Typen dieser Funktionen nicht mehr diese einfache Struktur auf. So hätte z.B. die Funktion map_example für den Datentyp

$$\begin{array}{llcl} \mathsf{datatype} & (\alpha, \beta)\mathsf{example} & = & \mathsf{C}\ (\alpha \to \mathsf{bool})\ \alpha\ (\beta\ \mathsf{list}) \\ & & \mid & \mathsf{D}\ ((\alpha, \beta)\mathsf{example}) \end{array}$$

nicht den Typ

$$(\alpha \to \gamma) \to (\beta \to \gamma') \to (\alpha, \beta)\mathsf{example} \to (\gamma, \gamma')\mathsf{example}$$

sondern

$$((\alpha \to \mathsf{bool}) \to (\delta \to \mathsf{bool})) \to (\alpha \to \delta) \to (\beta \to \delta') \to (\alpha, \beta)\mathsf{example} \to (\delta, \delta')\mathsf{example}$$

Da derartige Datentypen mit dem oben angegebenen allgemeinen Typschema nicht so gut vereinbar sind und daher deren Handhabung technisch aufwendiger wäre, beschränkt man sich in [Völker, 1995, Völker, 1997] auf reine Datentypen.

Beispiel 6.1
Wir wollen nun Beispiel 3.5 aus Abschnitt 3.3 erneut betrachten, wobei wir diesmal die kompakte Induktionsregel und die mit Hilfe von map formulierten charakteristischen Gleichungen für den Rekursionskombinator verwenden werden. Wir benötigen nun nur noch eine Substitutionsfunktion

$$\mathsf{subst_term} :: (\alpha \to (\alpha, \beta)\mathsf{term}) \to (\alpha, \beta)\mathsf{term} \to (\alpha, \beta)\mathsf{term}$$

die sich durch die Gleichungen

$$\begin{aligned} \mathsf{subst_term}\ f\ (\mathsf{Var}\ a) &= f\ a \\ \mathsf{subst_term}\ f\ (\mathsf{App}\ b\ ts) &= \mathsf{App}\ b\ (\mathsf{map}\ (\mathsf{subst_term}\ f)\ ts) \end{aligned}$$

charakterisieren läßt. Wir definieren also

$$\mathsf{subst_term}\ f \equiv \mathsf{term_rec}\ f\ (\lambda b, xs, ys.\ \mathsf{App}\ b\ ys)$$

woraus sich die obigen Gleichungen sofort ableiten lassen. Als Beweisziel wählen wir nun

$$\mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ t = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ t)$$

Mit Hilfe der alternative Induktionsregel erhalten wir durch Induktion über t die Beweisziele

(1) $\forall a.\ \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ (\mathsf{Var}\ a) = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ (\mathsf{Var}\ a))$

(2) $\forall b, ts.\ \mathsf{list_all}\ (\lambda u.\ \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ u = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ u))\ ts \Longrightarrow \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ (\mathsf{App}\ b\ ts) = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ (\mathsf{App}\ b\ ts))$

Beweisziel (1) läßt sich leicht mit Hilfe der charakteristischen Gleichungen für subst_term lösen. Zum Beweis von (2) werden die beiden Lemmata

$$\mathsf{map}\ (f \circ g)\ xs = \mathsf{map}\ f\ (\mathsf{map}\ g\ xs) \qquad \text{(map_compose)}$$

$$\mathsf{list_all}\ (\lambda x.\ f\ x = g\ x)\ xs \Longrightarrow \mathsf{map}\ f\ xs = \mathsf{map}\ g\ xs \qquad \text{(list_all_map_eq)}$$

verwendet, die sich leicht mittels Induktion über xs zeigen lassen. Aus der Prämisse

$$\mathsf{list_all}\ (\lambda u.\ \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ u = \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ u))\ ts$$

von Beweisziel (2) erhalten wir wegen (list_all_map_eq) somit

$$\mathsf{map}\ (\mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2))\ xs = \mathsf{map}\ (\lambda u.\ \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ u))\ xs$$

was aufgrund der Definition von $\circ$ gleichbedeutend ist mit

$$\mathsf{map}\ (\mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2))\ xs = \mathsf{map}\ (\mathsf{subst_term}\ f_1 \circ \mathsf{subst_term}\ f_2)\ xs \qquad (\star)$$

Wir erhalten schließlich

$$\begin{array}{ll} & \mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2)\ (\mathsf{App}\ b\ ts) \\ = & \quad \{\text{charakteristische Gleichungen für } \mathsf{subst_term}\} \\ & \mathsf{App}\ b\ (\mathsf{map}\ (\mathsf{subst_term}\ (\mathsf{subst_term}\ f_1 \circ f_2))\ ts) \\ = & \quad \{(\star)\} \\ & \mathsf{App}\ b\ (\mathsf{map}\ (\mathsf{subst_term}\ f_1 \circ \mathsf{subst_term}\ f_2)\ ts) \\ = & \quad \{(\text{map_compose})\} \\ & \mathsf{App}\ b\ (\mathsf{map}\ (\mathsf{subst_term}\ f_1)\ (\mathsf{map}\ (\mathsf{subst_term}\ f_2)\ ts)) \\ = & \quad \{\text{charakteristische Gleichungen für } \mathsf{subst_term}\} \\ & \mathsf{subst_term}\ f_1\ (\mathsf{subst_term}\ f_2\ (\mathsf{App}\ b\ ts)) \end{array}$$

womit Beweisziel (2) gelöst ist. □

Offensichtlich sind für den effektiven Einsatz der kompakten Induktionsregel entsprechende Hilfstheoreme für map, list_all etc. nützlich, da sie in einigen Beweisen eine zusätzliche Induktion über Listen ersparen. Beim Beweis von komplexeren Eigenschaften kann jedoch u.U. eine zusätzliche Induktion über Listen notwendig sein.

6.3 Alternative Konstruktionsverfahren

Auch bei der Konstruktion der repräsentierenden Mengen von verschachtelt rekursiven Datentypen wird in [Völker, 1995, Völker, 1997] ein etwas anderer Weg beschritten. Zur Repräsentation von Datentypen wird zunächst, aufbauend auf dem in 4.2.1 dargestellten Universum, manuell der Datentyp

$$\begin{array}{lcl} \mathsf{datatype}\quad \alpha\ \mathsf{univ} & = & \mathsf{ONE} \\ & \mid & \mathsf{LEAF}\ \alpha \\ & \mid & \mathsf{SCONS}\ (\alpha\ \mathsf{univ})\ (\alpha\ \mathsf{univ}) \\ & \mid & \mathsf{IN0}\ (\alpha\ \mathsf{univ}) \\ & \mid & \mathsf{IN1}\ (\alpha\ \mathsf{univ}) \\ & \mid & \mathsf{WRAP}\ \mathsf{nat}\ (\alpha\ \mathsf{univ}) \end{array}$$

konstruiert. Die Konstruktoren LEAF, SCONS, IN0 und IN1 haben den gleichen Zweck wie die entsprechenden Funktionen aus Abschnitt 4.2.1. Der Konstruktor ONE wird zur Darstellung nullstelliger Konstruktoren verwendet. Auf den Konstruktor WRAP wird später genauer eingegangen. Die Tatsache, daß α univ selbst ein Datentyp ist, hat gegenüber dem relativ allgemeinen Universum aus 4.2.1 den Vorteil, daß auf α univ Funktionen mittels primitiver Rekursion definiert werden können, die dann bei der Konstruktion von repräsentierenden Mengen zum Einsatz kommen.

Berechnung von Repräsentationen Wir wollen zunächst voraussetzen, daß der Typ $(\alpha, \beta)\mathsf{term}$ bereits durch die Angabe einer repräsentierenden Menge $\mathsf{term_rep_set}$ vom Typ $((\alpha + \beta)\mathsf{univ})\mathsf{set}$ definiert wurde und die Funktionen

$$
\begin{array}{lll}
\mathsf{Rep_term} & :: & (\alpha, \beta)\mathsf{term} \to (\alpha + \beta)\mathsf{univ} \\
\mathsf{Abs_term} & :: & (\alpha + \beta)\mathsf{univ} \to (\alpha, \beta)\mathsf{term}
\end{array}
$$

sowie

$$\mathsf{Rep_list} \ :: \ \gamma \ \mathsf{list} \to \gamma \ \mathsf{univ}$$

zur Verfügung stehen. Für die Definition des Konstruktors App mit den Argumenten $b :: \beta$ und $ts :: ((\alpha, \beta)\mathsf{term})\mathsf{list}$ muß nun die Repräsentation von ts berechnet werden. Dies geschieht mit Hilfe der Funktionen $\mathsf{Rep_list}$, $\mathsf{Rep_term}$ und map. Die Grundidee hierbei ist, zunächst die Repräsentation aller Listenelemente und dann die Repräsentation der Gesamtliste zu berechnen:

$$\underbrace{\mathsf{Rep_list} \ (\overbrace{\mathsf{map} \ \mathsf{Rep_term} \ ts}^{((\alpha+\beta)\mathsf{univ})\mathsf{list}})}_{((\alpha+\beta)\mathsf{univ})\mathsf{univ}}$$

Hierdurch erhält man nun allerdings ein Element des Typs $((\alpha + \beta)\mathsf{univ})\mathsf{univ}$. Die Elemente von $\mathsf{term_rep_set}$ müssen jedoch den Typ $(\alpha+\beta)\mathsf{univ}$ besitzen. Um dieses Problem zu beheben, wird in [Völker, 1997] die injektive Funktion

$$\mathsf{flat_univ} \ :: \ (\delta \ \mathsf{univ})\mathsf{univ} \to \delta \ \mathsf{univ}$$

eingeführt. Der Name wurde dabei in Anlehnung an die bekannte Funktion flat für Listen gewählt. Die Funktion $\mathsf{flat_univ}$ wird nun mittels primitiver Rekursion auf univ durch die Gleichungen

$$
\begin{array}{lll}
\mathsf{flat_univ} \ \mathsf{ONE} & = & \mathsf{ONE} \\
\mathsf{flat_univ} \ (\mathsf{LEAF} \ u) & = & \mathsf{WRAP} \ 0 \ u \\
\mathsf{flat_univ} \ (\mathsf{SCONS} \ v_1 \ v_2) & = & \mathsf{SCONS} \ (\mathsf{flat_univ} \ v_1) \ (\mathsf{flat_univ} \ v_2) \\
\mathsf{flat_univ} \ (\mathsf{IN0} \ v) & = & \mathsf{IN0} \ (\mathsf{flat_univ} \ v) \\
\mathsf{flat_univ} \ (\mathsf{IN1} \ v) & = & \mathsf{IN1} \ (\mathsf{flat_univ} \ v) \\
\mathsf{flat_univ} \ (\mathsf{WRAP} \ n \ v) & = & \mathsf{WRAP} \ (\mathsf{Suc} \ n) \ (\mathsf{flat_univ} \ v)
\end{array}
$$

definiert. Als Argumente des Konstruktors LEAF auftretende Elemente u vom Typ $\delta \ \mathsf{univ}$ werden hierbei – mit dem Index 0 versehen – mit Hilfe des Konstruktors WRAP „eingepackt“. Um Kollisionen mit bereits früher „eingepackten“ Elementen zu vermeiden, werden deren Indizes inkrementiert. Die Injektivität von $\mathsf{flat_univ}$ wird hierbei durch die Angabe einer entsprechenden Umkehrfunktion

$$\mathsf{expand_univ} \ :: \ \delta \ \mathsf{univ} \to (\delta \ \mathsf{univ})\mathsf{univ}$$

nachgewiesen, die die Eigenschaft

$$\mathsf{expand_univ} \ (\mathsf{flat_univ} \ u) = u$$

besitzt, was sich mittels Induktion über u zeigen läßt. Die Funktion expand_univ wird ebenfalls mittels primitiver Rekursion durch die Gleichungen

$$
\begin{array}{lcl}
\textsf{expand_univ ONE} & = & \textsf{ONE} \\
\textsf{expand_univ (LEAF } a) & = & \textsf{arbitrary} \\
\textsf{expand_univ (SCONS } u_1\ u_2) & = & \textsf{SCONS (expand_univ } v_1)\ \textsf{(expand_univ } v_2) \\
\textsf{expand_univ (IN0 } u) & = & \textsf{IN0 (expand_univ } u) \\
\textsf{expand_univ (IN1 } u) & = & \textsf{IN1 (expand_univ } u) \\
\textsf{expand_univ (WRAP } n\ u) & = & \textsf{if } n = 0 \textsf{ then LEAF } u \textsf{ else} \\
 & & \quad \textsf{WRAP } (n-1)\ \textsf{(expand_univ } u)
\end{array}
$$

definiert. Mit Hilfe von flat_univ erhalten wir nun die Repräsentation der Liste ts vom Typ $((\alpha, \beta)\textsf{term})\textsf{list}$ durch

$$
\underbrace{\textsf{flat_univ}\ (\underbrace{\textsf{Rep_list}\ (\overbrace{\textsf{map Rep_term } ts}^{((\alpha+\beta)\textsf{univ})\textsf{list}})}_{((\alpha+\beta)\textsf{univ})\textsf{univ}})}_{(\alpha+\beta)\textsf{univ}}
$$

Die Repräsentationsfunktion wird hierbei also ausschließlich aus bereits vorhandenen Funktionen zusammengesetzt. Dies setzt natürlich voraus, daß für alle beteiligten Typen t eine Funktion Rep_t existieren muß. Um Typen wie $\alpha \times \beta$ in Datentypdefinitionen verwenden zu können, muß daher zunächst manuell eine geeignete Repräsentationsfunktion Rep_prod definiert werden.

Repräsentierende Menge Die repräsentierende Menge term_rep_set wird nun induktiv durch die Regeln

$$
\frac{}{\textsf{IN0 (LEAF (Inl } a)) \in \textsf{term_rep_set}}
$$

$$
\frac{xs \in \textsf{list_set term_rep_set}}{\textsf{IN1 (SCONS (LEAF (Inr } b))\ \textsf{(flat_univ (Rep_list } xs)))}
$$

definiert, wobei die monotone Funktion list_set durch

$$
\begin{array}{lcl}
\textsf{list_set} & :: & \gamma\ \textsf{set} \to (\gamma\ \textsf{list})\textsf{set} \\
\textsf{list_set } S & \equiv & \{xs \mid \textsf{set_of_list } xs \subseteq S\}
\end{array}
$$

definiert ist. Mit $xs \in$ list_set term_rep_set wird also ausgedrückt, daß $xs :: ((\alpha + \beta)\textsf{univ})\textsf{list}$ eine Liste ist, deren Elemente aus der Menge term_rep_set stammen.

Kapitel 7

Schlußbemerkungen

Induktive Datentypen sind für viele Spezifikationsaufgaben von Bedeutung. So können z.B. Ausdrücke einer Programmiersprache oder Datenstrukturen für Algorithmen durch induktive Datentypen modelliert werden. Mit dem im Rahmen dieser Diplomarbeit implementierten Datentyppaket sind die wichtigsten in der Praxis vorkommenden Fälle wie verschränkt und verschachtelt rekursive Datentypen abgedeckt. Durch die definitorische Vorgehensweise ist die Korrektheit der abgeleiteten Regeln sichergestellt.

7.1 Ausblick auf coinduktive Datentypen

Gelegentlich kann es hilfreich sein, auch Datentypen mit Elementen unendlicher Tiefe zur Verfügung zu haben. Bei der Spezifikation verteilter, reaktiver Systeme treten beispielsweise sehr oft unendliche Abläufe auf. Die Komponenten eines solchen Systems lassen sich als sogenannte *stromverarbeitende Funktionen* auffassen. Eine Möglichkeit ist es, Datenströme durch unendliche Listen α llist mit den Konstruktoren

$$\begin{aligned} \mathsf{LNil} &:: \alpha\ \mathsf{llist} \\ \mathsf{LCons} &:: \alpha \rightarrow \alpha\ \mathsf{llist} \rightarrow \alpha\ \mathsf{llist} \end{aligned}$$

darzustellen [Paulson, 1997]. Die in Kapitel 3 beschriebenen Methoden der *strukturellen Induktion* und der *primitiven Rekursion* sind typischerweise nur für induktive Datentypen anwendbar, da diese nur Elemente endlicher Tiefe besitzen. Im Gegensatz dazu gibt es Listen $xs :: \alpha\ \mathsf{llist}$ mit der Eigenschaft

$$xs = \mathsf{LCons}\ x\ xs$$

Wie in Abschnitt 3.1 erwähnt wurde, ist dieser Fall für endliche Listen α list durch das Prinzip der strukturellen Induktion ausgeschlossen.

Für Datentypen wie α llist, die in der Literatur oft auch als *coinduktive Datentypen* bezeichnet werden, existieren andere Beweisprinzipien und Methoden zur Definition rekursiver Funktionen. Die Gleichheit zweier Listen xs und ys des Typs α llist kann durch Angabe einer geeigneten *Bisimulation*, d.h. einer speziellen Relation $r :: (\alpha\ \mathsf{llist} \times \alpha\ \mathsf{llist})\mathsf{set}$ nachgewiesen

werden, die das Paar (xs, ys) enthält. Diese Methode wird auch im Kontext von Prozeßalgebren wie CCS [Milner, 1989] zum Nachweis der Äquivalenz zweier Prozesse verwendet. Zur Definition rekursiver Funktionen existiert das Prinzip der *Corekursion*. Die Definition primitiv rekursiver Funktionen auf endlichen Listen α list ist *eingabeorientiert*, d.h. abhängig von der Gestalt der als Funktionsargument angegebenen Liste erfolgt eine bestimmte Ausgabe. Im Gegensatz dazu werden corekursive Funktionen *ausgabeorientiert* definiert: Das Funktionsergebnis ist stets eine potentiell unendliche Liste vom Typ α llist, wobei eine Fallunterscheidung danach erfolgt, ob eine leere Liste LNil oder eine Liste der Gestalt LCons x xs als Ergebnis geliefert wird. Dieser Umstand führt gelegentlich zu etwas unnatürlichen Definitionen. Besonders schwierig zu definieren sind Funktionen wie filter: Um zu entscheiden, ob das Ergebnis nun die Gestalt LNil oder LCons x xs hat, muß um eine unbekannte Anzahl von Elementen „vorausgeschaut" werden.

Verschiedene Möglichkeiten der Definition von filter mittels Corekursion werden in der Dissertation von Olaf Müller [Müller, 1998] beschrieben. Diese Arbeit enthält auch einen Vergleich von mehreren, teilweise auf coinduktiven Datentypen basierenden Formalisierungen unendlicher Listen, wobei deren Eignung für die Verifikation verteilter Systeme genauer untersucht wird. In [Paulson, 1997] wird beschrieben, wie coinduktive Datentypen, ebenfalls aufbauend auf dem in 4.2.1 beschriebenen Universum für rekursive Datentypen, in HOL definitorisch konstruiert werden können. Auch diese dort beschriebenen Konstruktionsverfahren könnten in Form eines Datentyppakets implementiert werden.

In diesem Zusammenhang ist insbesondere interessant, inwieweit sich die in Kapitel 4 beschriebenen Verfahren zur Konstruktion verschränkt und verschachtelt rekursiver Typen auch auf coinduktive Datentypen anwenden lassen. Bislang unklar ist, wie induktive und coinduktive Datentypen sinnvoll kombiniert werden können und welche Beweisprinzipien hierfür geeignet sind.

7.2 Weitere Arbeitspunkte

Im Zusammenhang mit dem implementierten Datentyppaket sind für die Zukunft noch folgende weitere Arbeitspunkte denkbar:

- Standardfunktionen für Datentypen wie z.B. t_map und t_all sowie geeignete Hilfstheoreme könnten vom Datentyppaket bereitgestellt werden.
- Für verschachtelt rekursive Datentypen könnte eine kompakte Induktionsregel, wie in Abschnitt 6.2 beschrieben, abgeleitet werden.
- Von Konrad Slind [Slind, 1996] wurde für HOL ein Paket namens TFL entwickelt, das die Definition von Funktionen mittels fundierter Rekursion erlaubt. Da hiermit momentan keine Funktionen auf verschränkt oder verschachtelt rekursiven Datentypen definiert werden können, wäre es wünschenswert, dieses Paket dahingehend zu erweitern. Hierbei ist von Interesse, ob die Funktionsdefinitionen dabei wie bisher auf den allgemeinen Kombinator wfrec für fundierte Rekursion abgestützt werden sollten, oder ob eine induktive Definition des Funktionsgraphen, wie dies in Abschnitt 4.6.2 für primitive Rekursion dargestellt wurde, evtl. einfacher ist.

Anhang A

Häufig verwendete HOL-Regeln

Regeln für $\exists_1$

$$\frac{P\ a \qquad \begin{matrix}[P\ x]_x \\ \vdots \\ x = a\end{matrix}}{\exists_1 x.\ P\ x}\ (\text{ex1I}) \qquad \frac{\exists_1 x.\ P\ x \qquad \begin{matrix}[P\ x \quad \forall y.\ P\ y \Longrightarrow y = x]_x \\ \vdots \\ Q\end{matrix}}{Q}\ (\text{ex1E})$$

Regeln für range

$$\frac{b = f\ x}{b \in \mathsf{range}\ f}\ (\text{range_eqI}) \qquad \frac{b \in \mathsf{range}\ f \qquad \begin{matrix}[b = f\ x]_x \\ \vdots \\ P\end{matrix}}{P}\ (\text{rangeE})$$

Regeln für ε

$$P\ (\varepsilon x.\ P\ x) = \exists x.\ P\ x \qquad (\text{select_eq_Ex})$$

$$\frac{\exists_1 x.\ P\ x \qquad P\ a}{(\varepsilon x.\ P\ x) = a}\ (\text{select1_equality})$$

Regeln für inv

$$\mathsf{inj}\ f \Longrightarrow \mathsf{inv}\ f\ (f\ x) = x \qquad (\text{inv_f_f})$$

$$y \in \mathsf{range}\ f \Longrightarrow f\ (\mathsf{inv}\ f\ y) = y \qquad (\text{f_inv_f})$$

Anhang B

Beweislaufzeit einiger Datentypen

Die folgende Tabelle zeigt die Beweislaufzeit für einige ausgewählte Datentypen. Es sind hierbei Laufzeiten sowohl für das im Rahmen dieser Arbeit implementierte Datentyppaket für Isabelle/HOL als auch für das von John Harrison implementierte Datentyppaket für HOL Light angegeben. Isabelle/HOL wurde hierbei mit dem New Jersey SML Compiler (Version 110.7) übersetzt. Die Werte wurden auf einer Sun Ultra-2 Workstation mit 512 MB Hauptspeicher ermittelt. Die Laufzeiten für HOL Light wurden von John Harrison mit CAML Light auf einem Pentium II mit einer Taktfrequenz von ca. 200-300 MHz ermittelt.

	Konstruktoren	**Verschränkungen**	**Verschachtelungen**	**Gesamtlaufzeit [s]**	
Beschreibung				**Isabelle**	**HOL Light**
Datentypen für kryptographisches Protokoll	34	-	-	133.37	76.13
	24	3	-		
68000 Instruction Set	3	-	-	1281.42	498.53
	8	-	-		
	8	-	-		
	2	-	-		
	14	-	-		
	10	-	-		
	98	-	-		
SML Grammatik	62	13	10	2759.88	3243.61
Verilog Grammatik	124	15	11	17516.06	15267.35

Bei der Definition der Verilog Grammatik mit Isabelle wurde eine maximale Prozeßgröße von ca. 300 MB ermittelt. Sind für ein Testbeispiel mehrere Zeilen angegeben, so bedeutet dies, daß das Beispiel aus mehreren Datentypen (d.h. einem großen und einigen wesentlich kleineren) bestand, für die eine gemeinsame Zeit ermittelt wurde.

Nach der Integration des neuen Datentyppakets war keine signifikante Verlängerung der Übersetzungszeit von Isabelle/HOL festzustellen, da hierbei nur relativ kleine Datentypen vorkommen. Größere Datentypen benötigen mit dem neuen Paket natürlich signifikant länger als mit dem alten axiomatischen Paket. Für den interaktiven Gebrauch ist jedoch nützlich, daß die charakteristischen Eigenschaften von Datentypen durch Setzen des Flags `quick_and_dirty` als Axiome eingeführt werden können, wobei die obigen Beispiele wesentlich weniger Zeit benötigen (7 s, 42 s, 34 s und 110 s).

Anhang C

Bewiesene Theoreme

Für die verschränkt rekursiven Datentypen $t_1, \ldots, t_n$ werden die folgenden Theoreme bewiesen und in der Theorie gespeichert:

- t_1_..._t_n`.induct`
  Simultane Induktionsregel für die Datentypen $t_1, \ldots, t_n$
- t_1_..._t_n`.recs`
  Charakteristische Gleichungen für Rekursionskombinatoren
- t_1_..._t_n`.size`
  Charakteristische Gleichungen für Maßfunktion size für fundierte Rekursion (siehe [Slind, 1996])

Für jeden Datentyp t_j werden die folgenden Theoreme bereitgestellt:

- t_j`.exhaust`
  Fallunterscheidungsregel für den Datentyp t_j
- t_j`.cases`
  Charakteristische Gleichungen für case-Kombinator des Datentyps t_j
- t_j`.distinct`
  Verschiedenheits-Theoreme für Konstruktoren des Datentyps t_j
- t_j`.inject`
  Injektivitäts-Theoreme für Konstruktoren des Datentyps t_j
- t_j`.case_cong`
  case-Kongruenz-Theorem (siehe [Slind, 1996])
- t_j`.nchotomy`
  Alternatives Fallunterscheidungstheorem (siehe [Slind, 1996])

Literaturverzeichnis

[Church, 1940] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5, pp. 56–68, 1940.

[Gordon, Melham, 1993] M. J. C. Gordon and T. F. Melham (eds.). *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.

[Gunter, 1992] Elsa L. Gunter. Why We Can't have SML Style datatype Declarations in HOL, in *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 International Workshop, Leuven, September 1992*, edited by L. J. M. Claesen and M. J. C. Gordon, IFIP Transactions A-20 (North-Holland, 1993), pp. 561-568.

[Gunter, 1993] Elsa L. Gunter. A Broader Class of Trees for Recursive Type Definitions for HOL, in *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93, Vancouver, B.C., August 11-13 1993: Proceedings*, edited by J. J. Joyce and C.-J. H. Seger, Lecture Notes in Computer Science, vol. 780, Springer-Verlag, 1994, pp. 141-154.

[Harrison, 1995] John Harrison. Inductive Definitions: automation and applications, in *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop, Aspen Grove, Utah, September 1995: Proceedings*, edited by E. T. Schubert, P. J. Windley, and J. Alves-Foss, Lecture Notes in Computer Science, Volume 971 (Springer-Verlag, 1995), pp. 200-213.

[Hensel, Jacobs, 1997] Ulrich Hensel and Bart Jacobs, Proof Principles for Datatypes with Iterated Recursion. In E. Moggi and G. Rosolini (eds), *Category Theory and Computer Science*, LNCS 1290, p. 220-241, Springer-Verlag, 1997.

[Melham, 1989] T. F. Melham, Automating Recursive Type Definitions in Higher Order Logic, in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P. A. Subrahmanyam (Springer-Verlag, 1989), pp. 341-386.

[Milner, 1989] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Müller, 1998] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* Dissertation, Technische Universität München, 1998.

[Paulson, 1989] Lawrence C. Paulson. The foundation of a generic Theorem Prover. *Journal of Automated Reasoning*, 5(3), pp. 363–397, 1989.

[Paulson, 1990] Lawrence C. Paulson. A formulation of the simple theory of types (for Isabelle). In P. Martin-Löf and G. Mints, editors, *COLOG-88: International Conference on Computer Logic*, Tallin, Estonian Academy of Sciences. LNCS 417, 1990.

[Paulson, 1993a] Lawrence C. Paulson. *Introduction to Isabelle*. Technical Report 280, University of Cambridge, Computer Laboratory, 1993. Aktuelle Version erhältlich unter `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/`

[Paulson, 1993b] Lawrence C. Paulson. *The Isabelle Reference Manual*. Technical Report 283, University of Cambridge, Computer Laboratory, 1993. Aktuelle Version erhältlich unter `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/`

[Paulson, 1993c] Lawrence C. Paulson. *Isabelle's Object-Logics*. Technical Report 286, University of Cambridge, Computer Laboratory, 1993. Aktuelle Version erhältlich unter `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/`

[Paulson, 1994] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 148-161, Nancy, France, June 1994. Springer-Verlag LNAI 814.

[Paulson, 1997] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175-204, March 1997.

[Slind, 1996] Konrad Slind. Function Definition in Higher Order Logic, in *Theorem Proving in Higher Order Logics: 9th International Conference, Turku, Finland, August 1996: Proceedings*, edited by J. von Wright, J. Grundy, and J. Harrison, Lecture Notes in Computer Science, Volume 1125 (Springer-Verlag, 1996), pp. 381-397.

[Völker, 1995] Norbert Völker. On the Representation of Datatypes in Isabelle/HOL. In Lawrence C. Paulson (Ed.), *Proceedings of the First Isabelle Users Workshop*, University of Cambridge Computer Laboratory, Technical Report 379, 1995.

[Völker, 1997] Norbert Völker. *Über Datentypen und die Verifikation reaktiver Systeme in HOL*. Dissertation, Fernuniversität Hagen, 1997.