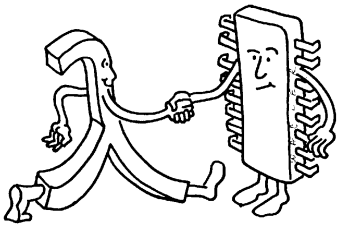
Formale Methoden des Systementwurfs

Wintersemester 2006/2007

3. Februar 2021



Peter Padawitz

Fachbereich Informatik
TU Dortmund

Inhaltsverzeichnis

↬ Bitte schicken Sie Korrekturen, Verbesserungs- und Ergänzungsvorschläge an *peter.padawitz@udo.edu*!

1 Wozu formale Methoden?

Reicht die Praxis nicht aus? Genügt es nicht, sich in existierende Softwareentwürfe einzuarbeiten, vielleicht die Fähigkeit zu entwickeln, dies möglichst schnell zu tun und das dabei Erlernte ebenso schnell zur Lösung neuer Probleme einzusetzen? Die Frage läßt sich allgemein nicht beantworten, heute weniger denn je. Allen Behauptungen zum Trotz, dass formale Methoden zur Erstellung zuverlässiger Software unumgänglich seien, hat sich in den letzten dreißig Jahren vieles im *software engineering* getan, ohne dass der Einfluß formaler Methoden dabei besonders sichtbar wäre. Tatsächlich ist das Verhältnis von Mathematik und Informatik wohl weitaus komplexer als z.B. das von Mathematik und Physik. Zumindest ist historisch nachvollziehbar, dass Informatik in der Mathematik, genauer gesagt der formalen Logik, ihren Ausgangspunkt hat, sich dann im Rahmen ihrer ingenieurwissenschaftlichen Entwicklung davon entfernt hat, aber gerade die sich ausweitenden ingenieurmäßigen Anforderungen nach formalen Ansätze verlangt haben, was den Theoretikern wiederum sehr lieb war, sie aber zwang, die neuen Anforderungen erst einmal auf ein für sie handhabbares Maß zurechtzustutzen, womit dann die Informatikingenieure wieder nicht so recht zufrieden waren. Das ging und geht immer noch so, nicht weil Wissenschaftler engstirnig sind, sondern weil eine solche Entwicklung vielfältigen sozialen, politischen und wirtschaftlichen Zwängen unterworfen ist.

Theorie-orientierte Informatik hat daher äußerst verschiedene Zielsetzungen. Sie reichen vom Ringen um grundlegende Begriffe von Berechenbarkeit und Komplexität sowie semantische Modelle von Sprachen und Maschinen bis hin zu solchen Entwurfs-, Auswertungs-, Lösungs- und Beweismethoden, die selbst rechnerunterstützt, einzelne Probleme von Software- oder Hardwareentwicklern oder in bestimmten Anwendungsbereichen arbeitenden Benutzern lösen helfen sollen. Dementsprechend werden wir auch im engeren Bereich der Programmierung mit theoretischen Ansätzen zu tun haben, von denen einige grundlegende Fragen angehen wie:

☞ Mit welchen mathematischen Modellen läßt sich die Bedeutung bestimmter Programmkonstrukte präzisieren?

während andere auf Sprachmittel und mehr oder weniger algorithmische Methoden zur Entwicklung korrekter Programme abzielen. Auch bei dieser Klasse theoretischer Ansätze sind aber eher die fundamentalen Ideen, Begriffe und Definitionen entscheidend als die "Kochrezepte", die sich dauernd ändern und die man daher nur verstehen und ggf. selbst entwickeln kann, wenn man die grundlegenden Konzepte kennt und erstaunt feststellt, dass sie - unter wechselnden Namen - immer wieder neu erfunden werden.

Grundlegende Konzepte kann man auf zweierlei Weise einführen. Entweder man formalisiert die eierlegende Wollmilchsau, die jedoch, da sie noch keiner gesehen hat, verdammt schwer zu modellieren ist. Oder man geht exemplarisch vor und hofft auf ein *intuitives* Verständnisses des Konzeptes hinter den Beispielen. Ich bevorzuge den zweiten Weg, weise aber schon mal darauf hin, dass "Beispiel" eine komplette Logik sein kann und das Konzept hinter dem Beispiel die allgemeine Struktur eines logischen Systems, d.h. Formelsyntax, Gültigkeitsbegriff und Regelmenge zur Ableitung gültiger Formeln.[1]

2 Programmiersprachen und -konzepte

An sich ist jede Programmiersprache selbst ein Formalismus, ja sogar ein logisches System im o.g. Sinne, wenn man Programme als Formeln, Gültigkeit als syntaktische und/oder semantische Korrektheit und Compiler oder Interpreter als Regelmengen zur Ableitung und Transformation korrekter Programme ansieht. Diese Sichtweise zu präzisieren ist das wesentliche Ziel bei der folgenden Auswahl theoretischer Ansätze. Jeder Abstraktion sollte aber eine Klassifizierung des Untersuchungsgegenstandes, hier der Programmiersprachen, vorangehen, hauptsächlich um später zu sehen, wieviel davon die einzelnen Ansätze erfassen (können). Grob lassen sich drei Klassen unterscheiden:

❶ *imperative, prozedurale, objekt-* und *zustandsorientierte* Sprachen wie Beta, C, C++, Eiffel, Java und Modula,

❷ *logische, deklarative* und *relationale* Sprachen wie Prolog und diverse Datenbanksprachen,

❸ *funktionale* und *applikative* Sprachen wie Haskell, Lisp oder ML.

Inzwischen werden auch Sprachen entwickelt, die Konzepte verschiedener Klassen in einer Sprache vereinen, insbesondere solche aus ❶ und ❸ oder aus ❷ und ❸. Die Schwierigkeit dabei ist natürlich der Bau von Compilern, die unterschiedliche Konzepte mit gleicher Effizienz verarbeiten können. Das kommt daher, dass den drei Klassen verschiedene *Ausführungsmodelle* zugrundeliegen, die wiederum auf die unterschiedlichen Klassen von Problemen zurückgehen, zu deren Lösung Programme geschrieben werden. So werden zustandsorientierte Sprachen vorwiegend benutzt, um Systeme der "realen Welt" zu steuern, Objekte zu erzeugen, deren Zustände zu verändern oder sie miteinander kommunizieren zu lassen. Demgegenüber werden logische Sprachen in Informationssystemen verwendet, um Relationen herzustellen, Anfragen zu beantworten oder Lösungen zu suchen. Funktionale Sprachen dienen der Auswertung von Ausdrücken, die i.a. statische Objekte darstellen.

Deklarativ bedeutet, dass - im Idealfall - ein Programm nur noch eine Aufgabe *beschreibt* und die Wahl des geeigneten Lösungsverfahrens ganz dem Compiler/Interpreter überläßt, während funktionale Sprachen die explizite Programmierung i.a. deterministischer Algorithmen erfordern. Diese sind dort allerdings - im Gegensatz zu ❶ - als Operationen auf *abstrakten* Daten, d.h. auf symbolischen Ausdrücken zu formulieren, was den meisten rein rechnerischen Problemen sehr angemessen ist, für die oft darüberliegende Aufgabe, *reaktive Systeme*

[1] Dieses - in der Theorie der Programmierung äußerst wichtige - Konzept (vgl. 4.2.4) hat übrigens inzwischen mehrere eierlegende Wollmilchsäue hervorgebracht, d.h. Logiken von Logiken, in denen fast alles in alles übersetzt werden kann.

dynamischer, kommunizierender Objekte zu steuern, jedoch vom Ansatz her weniger geeignet ist. *Applikativ* bedeutet, dass Funktionen nicht als Objekte, sondern durch ihre Applikationen, d.h. Anwendungen auf Argumente, definiert werden.

Welches sind nun die für ❶, ❷ bzw. ❸ charakteristischen Sprachkonstrukte? Was sind die wesentlichen Bestandteile von Programmen der jeweiligen Klasse? Wir werden diese Fragen hier nur sehr grob beantworten und einzelne Sprachkonstrukte erst später und dann ausgehend von ihrer *abstrakten Syntax* (vgl. Kap. 3) behandeln. [102] und [112] liefern sehr empfehlenswerte Einführungen und Gegenüberstellungen aller aktuellen Konzepte, ausgehend von konkreten Sprachen, das sind C, C++, Modula-2, Smalltalk, Lisp, ML, Prolog sowie Ada in [102] und Pascal, Ada, Smalltalk, Eiffel, ML, Lazy ML, Haskell und Prolog in [112].

2.1 Der Zustandsbegriff

Ein imperatives Programm ist eine strukturierte Folge von Anweisungen, die Werte von *dynamischen* oder *Programm-* oder *Zustands-Variablen* verändern. Früher genügte als Modellvorstellung für eine dynamische Variable die Speicherzelle, die mit einfachen Werten, meist Zahlen gefüllt wird. In einer objektorientierten Sprache steht die dynamische Variable für ein *Objekt*. Dies hat keinen *Wert*, sondern einen (veränderbaren) *Zustand*. Dieser Begriff stammt aus der Theorie dynamischer Systeme und Automaten, die älter ist als die objektorientierten Sprachen, zur Modellierung "real existierender" physikalischer Systeme entwickelt wurde und über den Hardware-Entwurf in die Informatik gekommen ist. Ein Zustand ist sozusagen nur ein halber Wert. Man kann die Objekte physikalischer Systeme zwar benennen, ihre Struktur aber oft nur unvollständig erkennen. Also *beobachtet* man ihr *Verhalten*, z.B. mithilfe von Messungen, um zumindest die Werte wichtiger *Attribute* von Objekten festzustellen und daraus genügend Information über die Objekte des Systems zu gewinnen, um es erfolgreich in einer bestimmten Richtung steuern zu können. Den Objekten werden *Nachrichten* geschickt, die sie veranlassen, in Abhängigkeit von ihrem jeweiligen Zustand ihnen inhärente *Prozeduren, Methoden* auszuführen und damit ihre Zustände zu ändern. Der Zustand eines Objektes besteht also aus Werten der für den geschilderten Prozeß entscheidenden Attribute.

Objektorientierte Sprachen übertragen diese physikalische Begriffswelt auf den Software-Entwurf und damit auf künstliche Systeme, die selbst der *Abstraktion* realer Systeme dienen. Dadurch kommt manches hinzu, wie z.B. die *Strukturierung* objektorientierter Programme, wofür man in der System- oder Automatentheorie keine formale Begründung finden wird. Es ist auch fraglich, ob der Begriff eines Zustands als Menge von Attributwerten (in imperativen Sprachen üblicherweise als *record* implementiert) ausreicht, um alle interessanten Objekttypen zu erfassen. Aus seinem Zustand soll ja in gewisser Weise die Identität eines Objektes erschlossen werden. Reichen dazu immer augenblickliche Attributwerte aus? Wann bestimmt eher die *Geschichte* des Objektes, d.h. die Folge der Zustände, die es bisher durchlaufen, seine Identität? Kann die Geschichte immer in eine endliche Zustandsstruktur hineincodiert werden? Die klassische Automatentheorie ist inzwischen zu mehreren *Theorien kommunizierender Systeme* erweitert worden, wo man gar nicht mehr von Objekten spricht, sondern nur noch *Prozesse*, also Objektgeschichten, untersucht und als - manchmal unendliche - Strukturen darstellt.

2.2 Die klassische CPO-Theorie

Einige Konzepte objektorientierter Sprachen, insbesondere die älteren imperativen Sprachen, werden klassischerweise in der *Theorie vollständiger Halbordnungen (CPOs = complete partial orders)* modelliert. Ein CPO A ist eine Menge, auf der eine Halbordnung $\leq$ definiert ist. A definiert alle Ausdrücke eines Typs der Sprache, z.B. alle Prozeduren. $\leq$ beschreibt die *Approximationsrelation* zwischen den Ausdrücken. $a \leq b$ bedeutet: a *approximiert* b. Wird z.B. Prozedur a im Zustand s aufgerufen und hält dann im Zustand s', dann tut dies auch Prozedur b, aber nicht notwendigerweise umgekehrt: wenn a nicht terminiert, kann b, im selben Anfangszustand aufgerufen, tun was es will, also auch in einem definierten Endzustand anhalten. Zum CPO gehört weiterhin die Eigenschaft,

dass jede *Kette*, d.h. jede abzählbare Folge immer "besserer" Approximationen, ein *Supremum* besitzt. Suprema modellieren auch "virtuelle Werte", die Prozeduren liefern, wenn sie nicht terminieren. Tatsächlich lassen sich in dieser Theorie nicht nur die einfachen Kontrollstrukturen imperativer Sprachen interpretieren, sondern auch *rekursive Prozedurdefinitionen*, ja sogar *rekursive* und *polymorphe Typdefinitionen*. Die Interpretation von Typdefinitionen heißt auch *Bereichstheorie*. Hier sind die Elemente der CPOs selbst Mengen, im Fall funktionaler Typen wieder CPOs. Diese typmodellierenden Mengen-CPOs müssen noch zusätzliche Komponenten haben, die sie zu ω-*Kategorien* machen (s. Def. 11.2.6).

Leider ist die klassische Theorie imperativer Sprachen nicht besonders handlich. Das liegt daran, dass sie dynamische Variablen, Prozeduren und Programmeigenschaften wie Termination in die Sprache der mathematischen Logik (und Algebra) übersetzt, also auf deren Grundkonzepte, nämlich Ausdrücke, Funktionen und Relationen, zurückführen muss. Demgegenüber ist die Modellierung funktional-logischer Sprachen einfacher, weil sie selbst (Teil-)Sprachen der mathematischen Logik sind, und zwar syntaktisch, semantisch und *operationell*. Letzteres bedeutet, wir können Compiler und Interpreter für solche Sprachen direkt aus Regelsystemen ihrer jeweiligen Logik herleiten. Deshalb werden wir mit diesen Sprachen beginnen und später imperative Konzepte hinzufügen. Tatsächlich liefert die Theorie funktional-logischer Sprachen Begriffe, mit denen auch imperative und objektorientierte Konzepte modelliert werden können und in die Ideen der klassischen (CPO-)Theorie eingehen, ohne dass diese selbst darin vorkommt. Man darf auch imperative *Konzepte* nicht mit imperativen *Sprachen* verwechseln. Was letztlich in einer imperativen Sprache implementiert wird, kann (und sollte) auf einer höheren, abstrakteren Ebene des Entwurfs eher funktional-logisch formuliert werden, zumindest was die - meist sehr großen - Anteile der Problemstellung anbelangt, die einer solchen Formulierung angemessen sind (s.o.). Programme lassen sich auf dieser Ebene weitaus leichter entwickeln und verifizieren als wenn die zugrundeliegenden Algorithmen sofort mit allerlei "Schmutz" der späteren Implementierungssprache daherkommen. Schließlich steht bereits eine Reihe von Compilern zur Verfügung, die funktional-logische Entwürfe in effiziente imperative Realisierungen, z.B. in C, übersetzen können.

➽ In Kapitel 2 bis 9 werden aufeinander aufbauende Begriffe entwickelt und Ergebnisse vorgestellt, die insbesondere auf formale Methoden zur Verifikation und schrittweisen Synthese von Programmen abzielen. Sie sind unabhängig von den letzten beiden Kapiteln 10 und 11, wo die Grundlagen der klassischen CPO- und Bereichssemantik von Programmen bzw. Datentypen präsentiert werden.

3 Konkrete und abstrakte Syntax

Die *konkrete* Syntax einer Programmiersprache ist i.a. durch eine kontextfreie Grammatik gegeben:

CF-Grammatik

Definition 3.1 Eine **kontextfreie** oder **CF-Grammatik** ist ein Quadrupel $G = (N, T, P, S)$, bestehend aus einer Menge N von **Nichtterminalen**, einer Menge T von **Terminalen**, einer Menge P von **Produktionen** oder **Regeln** der Form

$$A \to w \quad \text{mit } A \in N \text{ und } w \in (N \cup T)^*$$

und einem **Startsymbol** $S \in N$. Anstelle von n Produktionen mit derselben linken Seite:

$$A \to w_1 \quad , \ldots , \quad A \to w_n$$

schreiben wir eine:

$$A \to w_1 \mid \ldots \mid w_n .$$

Die Menge

$$L(G) = \{w \in T^* \mid S \xrightarrow{*}_G w\}$$

ist die **von G erzeugte Sprache**.[2]

▲ ▲

Sei $\mathbb{Z}$ die Menge der ganzen Zahlen.

Beispiel 3.2 (konkrete Syntax einer funktionalen Sprache) Eine funktionale Sprache mit verketteten (geschachtelten) Listen als grundlegender Datenstruktur (wie in Lisp) und nicht-applikativen Funktionsausdrücken (wie in FP; vgl. [11]), wird durch folgende CF-Grammatik definiert:

★ FPF[3]$= (N, T, P, S)$

★ $N = \{const, constL, fun\}$

★ $T = \mathbb{Z} \cup \{[], <, >, id, head, tail, num, +, *, =, \equiv, \circ, [,], if, then, else, ,, \alpha, /\}$

★ P besteht aus den Produktionen

- ✦ $const \rightarrow i$ für alle $i \in \mathbb{Z}$[4]
- ✦ $const \rightarrow []$
- ✦ $const \rightarrow < constL >$
- ✦ $constL \rightarrow const, constL$
- ✦ $constL \rightarrow const$
- ✦ $fun \rightarrow id \mid head \mid tail \mid num \mid + \mid * \mid = \mid \equiv const \mid fun \circ fun \mid [fun, \ldots, fun]$
- ✦ $fun \rightarrow if\ fun\ then\ fun\ else\ fun \mid \alpha fun \mid /fun$

★ $S = fun$

Also ist $L(FPF)$ die Menge der syntaktisch korrekten FPF-Programme. Semantisch beschreibt jedes FPF-Programm eine Funktion. Dateneingaben erfolgen über die Funktion $\equiv$. So beschreibt z.B. das Wort

$$head \circ [\equiv 3, \equiv 5, \equiv 2]$$

die konstante Funktion, die den Kopf der Liste $[3, 5, 2]$ liefert. ✌

Beispiel 3.3 (konkrete Syntax einer imperativen Sprache) Eine imperative Sprache mit den üblichen nicht-rekursiven Kontrollstrukturen ist durch folgende kontextfreie Grammatik gegeben: Sei V der (Denotationen) von Zustandsvariablen (s. 2.1).

★ IPF[5]$= (N, T, P, S)$

★ $N = \{const, var, exp, boolexp, com\}$

★ $T = \mathbb{Z} \cup V \cup \{+, =, and, :=, ;, if, then, else, while, do, repeat, until, skip, true, false, >\}$

★ P besteht aus den Produktionen

- ✦ $const \rightarrow i$ für alle $i \in \mathbb{Z}$

[2] Zur Definition von $\overset{*}{\rightarrow}_G$ vgl. Literatur über *Formale Sprachen*.
[3] "functional programming fragment"; vgl. [66], 1.3
[4] Diese Zeile steht für unendlich viele Regeln, was der übliche Begriff einer Grammatik eigentlich verbietet. Da sich jedoch die (z.B. Dezimal-) *Darstellungen* aller Elemente von $\mathbb{Z}$ aus endlich vielen Regeln ableiten lassen, betrachten wir die Zeile als Kurzform für eine solche endliche Regelmenge.
[5] "imperative programming fragment"

- ✦ $var \rightarrow x$ für alle $x \in V$
- ✦ $exp \rightarrow const \mid var \mid exp + exp$
- ✦ $boolexp \rightarrow true \mid false \mid exp = exp \mid exp > exp \mid boolexp\ and\ boolexp \mid \neg\ boolexp$
- ✦ $com \rightarrow var := exp \mid com; com \mid if\ boolexp\ then\ com\ else\ com \mid while\ boolexp\ do\ com$
- ✦ $com \rightarrow repeat\ com\ until\ boolexp \mid skip$

★ $S = com$

Also ist $L(IPF)$ die Menge der syntaktisch korrekten IPF-Programme. Semantisch beschreibt jedes IPF-Programm eine Relation zwischen *Variablenbelegungen* (s. Def. 4.1.6). Dateneingaben erfolgen über Variablenbelegungen. So beschreibt z.B. das Wort

$$y := 1;\ while\ x > 0\ do\ (y := y * x;\ x := x - 1)$$

die Relation $\{((x, y), (0, x!)) \mid x, y \in \mathbb{N}\}$. ✌

Die abstrakte Syntax erhält man aus der konkreten, d.h. der CF-Grammatik, indem man alle Terminale aus den Produktionen entfernt und für jede Produktion ein Funktionssymbol einführt, das den *Konstruktor* eines Datentyps (wie in Hakell) liefert.[6] Abstrakte Programme sind dann aus Konstruktoren zusammengesetzte *Terme* (= funktionale Ausdrücke). Jedem Nichtterminal der Grammatik entspricht ein Datentyp, zunächst syntaktisch: eine *Sorte*:

sortierte Menge

Definition 3.4 Sei S eine Menge von **Sorten**. Eine Mengenfamilie $A = \{A_s\}_{s \in S}$ heißt **S-sortierte Menge**.[7] Ist $w = s_1 \dots s_n \in S^+$, dann schreiben wir A_w anstelle von $A_{s_1} \times \dots \times A_{s_n}$. Sind A und B S-sortierte Mengen, dann heißt eine Familie $R = \{R_s \subseteq A_s \times B_s\}_{s \in S}$ von Relationen **S-sortierte (binäre) Relation**.

Sind A und B S-sortierte Mengen, dann heißt eine Funktion $f : A \rightarrow B$ **S-sortiert**, wenn es für alle $s \in S$ eine Funktion $f_s : A_s \rightarrow B_s$ existiert mit $f_s(a) = f(a)$ für alle $a \in A_s$. Ist $s_1, \dots, s_n \in S$ und $w = s_1 \dots s_n$, dann wird die Funktion $f_w : A_w \rightarrow B_w$ definiert durch $f_w(a) = (f_{s_1}(a_1), \dots, f_{s_n}(a_n))$ für alle $a = (a_1, \dots, a_n) \in A_w$.

Die Sortierung von Mengen und Funktionen nennt man auch *Typisierung*. Sorten sind allerdings zunächst nur unstrukturierte Namen und können daher – im Gegensatz zu den Elementen vieler sortierter Mengen – nicht zu Ausdrücken zusammengesetzt werden.

Signatur

Definition 3.5 Eine **(mehrsortige) Signatur** $\Sigma = (S, F, R)$ besteht aus einer Menge S von **Sorten** und zwei S^+-sortierten Mengen F von **Funktionssymbolen** und R von **Prädikaten** oder **Relationssymbolen**[8]. Die Elemente von F_s, $s \in S$, heißen **Konstanten**. Anstelle von

$$f \in F_s \quad \text{bzw.} \quad g \in F_{s_1 \dots s_n s} \quad \text{bzw.} \quad r \in R_{s_1 \dots s_n}$$

schreiben wir

$$f :\rightarrow s \in \Sigma \quad \text{bzw.} \quad g : s_1 \dots s_n \rightarrow s \in \Sigma \quad \text{bzw.} \quad r : s_1 \dots s_n \in \Sigma \quad \text{oder}$$
$$g : s_1 \times \dots \times s_n \rightarrow s \in \Sigma \quad \text{bzw.} \quad r : s_1 \times \dots \times s_n \in \Sigma.$$

[6] Durch | getrennte Alternativen einer Produktion stehen für mehrere Regeln, liefern also auch mehrere Funktionssymbole
[7] Eine Mengenfamilie ist eine Abbildung, die jedem Element einer Indexmenge (hier S) eine Menge zuordnet.
[8] In der Künstlichen Intelligenz spricht man auch von **Konzepten**.

Ein Symbol $\equiv: ss$ heißt **Gleichheitsprädikat**. Andere Prädikate heißen **logische Prädikate**. $r : w \in R$ **hat ein Komplement**, wenn das Symbol $\overline{r} : w$ zu R gehört. Bei $r = \equiv$ schreiben wir $\not\equiv$ anstelle von $\overline{r}$.

Dies ist alles noch pure Syntax. S, F und R werden erst später als Datenmengen, Funktionen bzw. Relationen (auf den Datenmengen) interpretiert (s. Def. 4.1.2). Die abstrakte Syntax einer Programmiersprache wird als Signatur definiert, die man wie folgt aus der CF-Grammatik der Sprache konstruiert:

abstrakte Syntax

Definition 3.6 Sei $G = (N, T, P, S)$ eine CF-Grammatik. Dann heißt die folgende Signatur **abstrakte Syntax** von G:

- N ist die Sortenmenge.
- Für jede Produktion $p = (A \to w_1 A_1 w_2 \dots w_n A_n w_{n+1})$ mit $w_i \in T^*$ und $A_i \in N$ enthält Σ ein Funktionssymbol $f_p : A_1 \dots A_n \to A$.
- Σ enthält keine weiteren Symbole.

Beipiel 3.7 Die abstrakte Syntax von FPF (vgl. 3.2) lautet wie folgt:

$$
\begin{aligned}
&\text{Sortenmenge} = \{const, constL, fun\}\\
&\text{Menge der Funktionssymbole} = \{\ i :\to const \quad \text{für alle } i \in \mathbb{Z},\\
&\qquad [] :\to const,\\
&\qquad mkConst : constL \to const,\\
&\qquad single : const \to constL,\\
&\qquad app : const \times constL \to constL,\\
&\qquad id, head, tail, num, add, mul, eq :\to fun,\\
&\qquad C : const \to fun,\\
&\qquad comp : fun \times fun \to fun,\\
&\qquad prod : fun \times \dots \times fun \to fun,\\
&\qquad cond : fun \times fun \times fun \to fun,\\
&\qquad map, fold : fun \to fun\ \}
\end{aligned}
$$

Beispiel 3.8 Die abstrakte Syntax von IPF (vgl. 3.3) lautet wie folgt:

$$
\begin{aligned}
&\text{Sortenmenge} = \{const, var, exp, boolexp, com\}\\
&\text{Menge der Funktionssymbole} = \{\ i :\to const \quad \text{für alle } i \in \mathbb{Z},\\
&\qquad x :\to var \quad \text{für alle } x \in V,\\
&\qquad C : const \to exp,\\
&\qquad V : var \to exp,\\
&\qquad add, mul : exp \times exp \to exp,\\
&\qquad True, False :\to boolexp,\\
&\qquad eq, greater : exp \times exp \to boolexp,\\
&\qquad and : boolexp \times boolexp \to boolexp,\\
&\qquad not : boolexp \to boolexp,\\
&\qquad assign : var \times exp \to com,
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{seq} : \mathit{com} \times \mathit{com} \to \mathit{com},\\
&\mathit{cond} : \mathit{boolexp} \times \mathit{com} \times \mathit{com} \to \mathit{com},\\
&\mathit{loop} : \mathit{boolexp} \times \mathit{com} \to \mathit{com},\\
&\mathit{repeat} : \mathit{com} \times \mathit{boolexp} \to \mathit{com},\\
&\mathit{skip} :\to \mathit{com}\ \}\ \text{✌}
\end{aligned}
$$

Die Namen der Funktionssymbole einer abstrakten Syntax können auch aus mehreren Teilen bestehen (*mixfix*-Syntax), so dass häufig schon die Terminale auf der rechten Seite einer Produktion den Namen des entsprechenden Funktionssymbols bilden: z.B. $\mathit{while\_do\_} : \mathit{boolexp} \times \mathit{com} \to \mathit{com}$ statt $\mathit{loop} : \mathit{boolexp} \times \mathit{com} \to \mathit{com}$.

Term, Atom

Definition 3.9 Sei $\Sigma = (S, F, R)$ eine Signatur und $\mathbf{X}$ eine S-sortierte Menge von *Individuen*-**Variablen**. Die S-sortierte Menge $T_\Sigma(X)$ der **Σ-Terme** und die Menge $At_\Sigma(X)$ der **Σ-Atome** sind *induktiv* definiert:

- Für alle $s \in S$ ist $X_s \subseteq T_\Sigma(X)_s$.
- Für alle $w \in S^*,\ s \in S,\ f : w \to s \in \Sigma$ und $t \in T_\Sigma(X)_w$[9] ist $f(t) \in T_\Sigma(X)_s$.
- Für alle $w \in S^+,\ r : w \in \Sigma$ und $t \in T_\Sigma(X)_w$ ist $r(t) \in At_\Sigma(X)$.

Ein variablenfreier Term heißt **Grundterm**.[10] Die Menge der Σ-Grundterme wird mit T_Σ bezeichnet. Ist Σ die abstrakte Syntax einer CF-Grammatik G, dann heißen die Σ-Grundterme auch **Syntaxbäume von** G. Ein variablenfreies Atom heißt **Grundatom** oder **Fakt**.

Abstrakte Syntax dient der Darstellung und Verwendung von Programmen als mathematische Objekte. Die Übersetzung von Quellprogrammen in Syntaxbäume gehört zum *front-end* eines jeden Compilers. Abstrakte Syntax macht Programme zu baumartig-rekursiv aufgebauten Objekten, die von Übersetzungsalgorithmen top-down oder bottom-up *attributiert* und dann in Zielcode transformiert werden.

4 Mehrsortige Prädikatenlogik

4.1 Syntax und Semantik

Signaturen, Terme und Atome sind Grundbegriffe mehrsortiger Logik. Das vorhergehende Kapitel zeigt, wie Programmiersprachen direkt in abstrakte Syntax übersetzt werden. Es geht aber nicht nur um die Programme, sondern auch um ihre Eigenschaften. Um die zu formulieren, gehen wir in der Syntax einen Schritt weiter und bilden prädikatenlogische Formeln:

Formel

Definition 4.1.1 Sei $\Sigma = (S, F, R)$ eine Signatur und X eine S-sortierte Variablenmenge. Die Menge der **(prädikatenlogischen) Σ-Formeln (über X)** ist induktiv definiert:

- *True* und *False* sind Σ-Formeln.
- Jedes Σ-Atom ist eine Σ-Formel.

[9] vgl. 3.4

[10] Das Präfix "Grund" wird auch anderen Formelmengen vorangestellt und bedeutet immer "variablenfrei".

❀ Für alle Σ-Formeln φ, ψ und $x \in X$ sind auch $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$, $\forall x\varphi$ und $\exists x\varphi$ Σ-Formeln.

Der nächste Schritt bei der Konstruktion einer Logik ist die Festlegung auf semantische Bereiche, in denen Terme und Formeln interpretiert werden sollen.

Struktur, Kongruenz, Homomorphismus

Definition 4.1.2 Sei $\Sigma = (S, F, R)$ eine Signatur. Eine **Σ-Struktur** ist ein Tripel $(A, \mathcal{F}, \mathcal{R})$, kurz: A, bestehend aus einer S-sortierten Menge A, einer Menge $\mathcal{F}$ von Funktionen und einer Menge $\mathcal{R}$ von Relationen derart, dass

❀ A_s für alle $s \in S$ nichtleer ist,[11]

❀ $\mathcal{F}$ für alle $f : w \to s \in F$ eine Funktion $f^A : A_w \to A_s$ enthält,

❀ $\mathcal{R}$ für alle $r : w \in R$ eine Relation $r^A \subseteq A_w$ enthält,

❀ die Interpretation $\equiv^A$ der Gleichheitsprädikate von Σ (siehe Def. 3.5) eine prädikatenverträgliche Σ-Kongruenz ist (s.u.).

Die Menge A_s, $s \in S$, heißt **Trägermenge** oder **Datenbereich** von s. Enthält Σ keine Prädikate, dann heißt A auch **Σ-Algebra**.

Sei A eine Σ-Struktur. A heißt **Σ-Struktur mit Gleichheit**, wenn $\equiv^A$ die **Gleichheit** in A oder **Diagonale** von $A \times A$ ist, d.h. $\equiv^A = \{(a, a) \mid a \in A\}$.

A heißt **Σ-Struktur mit Komplementen**, wenn für alle Prädikate $r : w \in \Sigma$, die ein Komplement haben (siehe Def. 3.5), $\overline{r}^A$ das relationales Komplement von r^A ist, also $\overline{r}^A = A_w \setminus r^A$ gilt.

Sei $\sim \subseteq A \times A$ eine S-sortierte Relation (s. Def. 3.4). $\sim$ ist **funktionsverträglich**, wenn für alle und Funktionssymbole $f : w \to s$ von Σ und alle $a = (a_1, \ldots, a_n),\ b = (b_1, \ldots, b_n) \in A_w$ gilt:

$$(\forall\ 1 \leq i \leq n : a_i \sim b_i) \quad \Rightarrow \quad f^A(a) \sim f^A(b).$$

Eine funktionsverträgliche Äquivalenzrelation heißt **Σ-Kongruenz(relation)**. $\sim$ ist **prädikatenverträglich**, wenn für alle Prädikate $r : w$ von Σ und $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n) \in A_w$ gilt:

$$a \in r^A \wedge (\forall\ 1 \leq i \leq n : a_i \sim b_i) \quad \Rightarrow \quad b \in r^A.$$

Sei $\sim \subseteq A \times A$ eine Σ-Kongruenz. Der **Quotient** oder die **Faktorisierung** $A/\sim$ **nach** $\sim$ die folgendermaßen definierte Σ-Struktur:

❀ Für alle $s \in S$ ist $A_s/\sim_s$ die Menge der Äquivalenzklassen $[a]_\sim$ von Elementen von A_s.

❀ Für alle $f : w \to s \in \Sigma$ und $(a_1, \ldots, a_n) \in A_w$,

$$f^{A/\sim}([a_1]_\sim, \ldots, [a_n]_\sim) \quad =_{def} \quad [f^A(a_1, \ldots, a_n)]_\sim.$$

❀ Für alle $r : w \in \Sigma$ und $(a_1, \ldots, a_n) \in A_w$,

$$(c_1, \ldots, c_n) \in r^{A/\sim} \iff_{def} \exists\ a_1 \in c_1 : \ldots \exists\ a_n \in c_n : (a_1, \ldots, a_n) \in r^A.$$

[11] Entsprechend dieser Voraussetzung sei im Folgenden auch für alle $s \in S$ die Termmenge $T_{\Sigma,s}$ stets nichtleer. Man sagt: Σ **hat keine leeren Sorten.** Das erlaubt es, jede prädikatenlogische Formel so mit einer Grundsubstitution zu instanziieren, dass eine Grundformel entsteht. Ohne diese Voraussetzung können ungültige Formeln abgeleitet werden, die nur deshalb ungültig sind, weil sie Variablen mit leeren Sorten enthalten.

Seien A und B Σ-Strukturen. Eine S-sortierte Abbildung $h : A \to B$ heißt **Σ-Homomorphismus**, falls für alle Funktionssymbole $f : w \to s \in \Sigma$ $h_s \circ f^A = f^B \circ h_w$ gilt. h ist **monoton**, wenn für alle Prädikate $r : w$ von Σ gilt: $h(r^A) \subseteq r^B$.

Sei h ein Σ-Homomorphismus. Gibt es einen Σ-Homomorphismus $g : B \to A$ mit $g \circ h = id^A$ und $h \circ g = id^B$, dann ist h ein **Σ-Isomorphismus** und A und B sind **isomorph**, geschrieben: $A \cong B$.

Eine Σ-Struktur C heißt **Σ-Unterstruktur von** A, falls für alle Sorten $s \in \Sigma$ C_s eine Teilmenge von A_s ist, für alle Funktionssymbole $f : w \to s \in \Sigma$ und $c \in C_w$ $f^C(c) = f^A(c)$ und für alle Prädikate $r : w \in \Sigma$ $r^C = r^A \cap C_w$ gilt. Die kleinste Σ-Unterstruktur von A bezeichnen wir mit A_Σ.

Die S-sortierte Menge $T_\Sigma(X)$ der Σ-Terme über X wird wie folgt zur **Σ-Termalgebra** erweitert:

- ❀ Für alle $f : w \to s \in \Sigma$ und $t \in T_\Sigma(X)_w$ ist $f^{T_\Sigma(X)}(t) = f(t)$.

Durch eine Interpretation jedes Prädikates $r : w \in \Sigma$ als Teilmenge von $T_\Sigma(X)_w$ wird $T_\Sigma(X)$ zu einer Σ-Struktur, die **Herbrandstruktur** genannt wird. Ist X die leere Variablenmenge, dann schreiben wir T_Σ anstelle von $T_\Sigma(X)$.

☞ Zeigen Sie, dass die Zeile

$$(c_1, \ldots, c_n) \in r^{A/\sim} \iff_{def} \exists\, a_1 \in c_1 : \ldots \exists\, a_n \in c_n : (a_1, \ldots, a_n) \in r^A$$

in der Definition von $A/\sim$ durch

$$([a_1], \ldots, [a_n]) \in r^{A/\sim} \iff_{def} (a_1, \ldots, a_n) \in r^A$$

ersetzt werden kann, wenn $\sim$ mit $r : w$ verträglich ist!

☞ Zeigen Sie, dass eine Äquivalenzrelation $\sim \subseteq A \times A$ genau dann mit $r : w$ verträglich ist, wenn $\sim$ mit dem Funktionssymbol $f_r : w \to \mathit{bool}$ verträglich ist, wobei bool und f_r wie folgt interpretiert und $\sim$ wie folgt auf $\{0,1\}^2$ erweitert wird:

$$\begin{array}{lll} A_{bool} & =_{def} & \{0,1\}, \\ f_r^A(a) & =_{def} & \begin{cases} 1 & \text{falls } a \in r^A \\ 0 & \text{sonst,} \end{cases} \\ b \sim b' & \iff_{def} & b = b'. \end{array}$$

☞ Zeigen Sie, dass sich zu jedem Σ-Homomorphismus $h : A \to B$ die Bildmenge $h(A) =_{def} \{h(a) \mid a \in A\}$ zu einer Σ-Struktur fortsetzen läßt, d.h. es gibt eine Σ-Struktur C mit $C_s = h(A_s)$ für alle Sorten von Σ!

☞ Zeigen Sie, dass eine Σ-Struktur A genau dann Σ-Unterstruktur einer Σ-Struktur B ist, wenn für alle Sorten $s \in \Sigma$ A_s eine Teilmenge von B_s ist und die Inklusionen $A_s \subseteq B_s$ einen Σ-Homomorphismus bilden!

☞ Zeigen Sie, dass ein Σ-Homomorphismus genau dann ein Isomorphismus ist, wenn er bijektiv ist!

Homomorphismen liefern Kongruenzen und umgekehrt:

Äquivalenzkern und -abschluss

Definition 4.1.3 Sei $\sim$ eine Σ-Kongruenzrelation auf A, dann ist die **natürliche Abbildung** $nat : A \to A/\sim$ mit $nat(a) =_{def} [a]_\sim$ für alle $a \in A$ ein Σ-Homomorphismus. Ist umgekehrt $h : A \to B$ ein Σ-Homomorphismus, dann ist die durch

$$a \sim_h b \iff_{def} h(a) = h(b)$$

definierte Äquivalenzrelation $\sim_h$, der **Äquivalenzkern von** h, eine (in der Regel nicht prädikatenverträgliche!) Σ-Kongruenz. Umgekehrt spricht man vom **Äquivalenzabschluss** R^{eq} einer binären Relation $R \subseteq A \times A$ als der kleinsten Äquivalenzrelation, die R umfasst.

☞ Wie wird R^{eq} konstruiert?

☞ Zeigen Sie durch Induktion über die Konstruktion von R^{eq} aus R, dass R^{eq} eine Kongruenzrelation ist, wenn R funktionsverträglich ist!

Dieses Ergebnis impliziert, dass die Funktionsverträglichkeit von $\sim \subseteq A \times A$ genügt, um einen Quotienten von A zu bilden: Ist $\sim$ keine Äquivalenz, dann faktorisiert man eben nach $\sim^{eq}$ und nicht nach $\sim$.

Homomorphismen liefern Unterstrukturen und umgekehrt: Ist C eine Σ-Unterstruktur von A, dann ist die **Einbettung** $inc : C \to A$ mit $inc(c) =_{def} c$ für alle $c \in C$ ein Σ-Homomorphismus. Ist umgekehrt $h : B \to A$ ein Σ-Homomorphismus, dann ist $h(B)$ eine Σ-Unterstruktur von A (Übung; siehe oben).

Diese Dualität führt zu den Homomorphiesätzen:

Satz 4.1.4 (Homomorphiesätze)

(1) Zu jedem Σ-Homomorphismus $h : A \to B$ gibt es eindeutig einen Σ-Homomorphismus $h^* : A/\sim_h \to B$ mit $h^* \circ nat = h$. h^* ist injektiv. Ist h surjektiv, dann ist h^* bijektiv.

(2) Zu jedem Σ-Homomorphismus $h : B \to A$ gibt es eindeutig einen Σ-Homomorphismus $h^* : B \to h(B)$ mit $inc \circ h^* = h$. h^* ist surjektiv. Ist h injektiv, dann ist h^* bijektiv.

Beweis. ☞ Übung. ❑

Prinzipiell kann man die Zugehörigkeit von Daten zu verschiedenen Datenbereichen von A auch durch einstellige Prädikate ausdrücken, die ja auch als Teilmengen von A zu interpretieren wären. S könnte dann einelementig sein. Die meisten Funktionssymbole müßten dann aber als *partielle* Funktionen interpretiert werden. Die Auswertbarkeit von Termen (s.u.) wäre nicht mehr syntaktisch entscheidbar. Sortierung und Typisierung wurden in Programmiersprachen gerade deshalb eingeführt, weil dadurch ein großer Teil der Korrektheit syntaktisch, zur *Übersetzungszeit*, abprüfbar wird.

Weit verbreitet hat sich inzwischen auch das Konzept der **Untersorten**, das auf der Ebene abstrakter Syntax dem objektorientierten Konzept der **Vererbung** entspricht. Man schreibt z.B. $nat < int$ und legt damit fest, dass jede Σ-Struktur A nat als Teilmenge von A_{int} interpretiert. Der Beziehung $nat < int$ entspricht eine **Einbettungsfunktion** $in : nat \to int$, die in A als Inklusion interpretiert wird: $in^A(n) =_{def} n$. Partielle Funktionen lassen sich genausogut mithilfe von Obersorten spezifizieren (s. §§5.1 und 6.4).

Beispiel 4.1.5 (Semantik von FPF) Sei Σ die abstrakte Syntax der funktionalen Sprache FPF (Bsp. 3.7). Die intendierte Semantik von FPF läßt sich als eine Σ-Algebra A formulieren:

★ A_{const} = Menge der geschachtelten, in eckige Klammern eingeschlossenen Listen ganzer Zahlen

★ $A_{constL} = A_{const} \setminus \{[]\}$

★ A_{fun} = Menge der partiellen Funktionen von A_{const} nach A_{const}

★ $i^A = [i]$ für alle $i \in \mathbb{Z}$

★ $[]^A = []$

★ $mkConst^A(L) = L$ für alle $L \in A_{constL}$

★ $single^A(L) = [L]$ für alle $L \in A_{const}$

★ $app^A(L, [L_1, \ldots, L_n]) = [L, L_1, \ldots, L_n]$ für alle $n \geq 1$ und $L_1, \ldots, L_n \in A_{const}$

★ $id^A(L) = L$ für alle $L \in A_{const}$

★ $head^A([L_1,\ldots,L_n]) = L_1$ und $tail^A([L_1,\ldots,L_n]) = [L_2,\ldots,L_n]$ für alle $n \geq 1$ und $L_1,\ldots,L_n \in A_{const}$
$head^A([])$ und $tail^A([])$ sind undefiniert

★ $num^A([i]) = [[]]$ für alle $i \in \mathbb{Z}$[12]
$num^A(L) = []$ für alle anderen $L \in A_{const}$

★ $add^A([i,j]) = [i+j]$ für alle $i,j \in \mathbb{Z}$
$add^A(L)$ ist undefiniert für alle anderen $L \in A_{const}$

★ $mul^A([i,j]) = [i*j]$ für alle $i,j \in \mathbb{Z}$
$mul^A(L)$ ist undefiniert für alle anderen $L \in A_{const}$

★ $eq^A([L,L']) = \left\{ \begin{array}{ll} [[]] & \text{falls } L = L' \\ [] & \text{sonst} \end{array} \right\}$ für alle $L, L' \in A_{const}$
$eq^A(L)$ ist undefiniert für alle anderen $L \in A_{const}$

★ $C^A(L)(L') = L$ für alle $L \in A_{const}$

★ $comp^A(f,g) = f \circ g$ für alle $f,g \in A_{fun}$

★ $prod^A(f_1,\ldots,f_n)(L) = \left\{ \begin{array}{ll} [f_1(L),\ldots,f_n(L)] & \text{falls } f_1(L),\ldots,f_n(L) \text{ definiert sind} \\ \text{undefiniert} & \text{sonst} \end{array} \right\}$
für alle $f_1,\ldots,f_n \in A_{fun}$, $n \geq 1$, und $L \in A_{const}$

★ $cond^A(f,g,h)(L) = \left\{ \begin{array}{ll} g(L) & \text{falls } f(L) = [[]] \\ h(L) & \text{falls } f(L) = [] \\ \text{undefiniert} & \text{sonst} \end{array} \right\}$ für alle $f,g,h \in A_{fun}$ und $L \in A_{const}$

★ $map^A(f)([]) = []$ für alle $f \in A_{fun}$
$map^A(f)([L_1,\ldots,L_n]) = \left\{ \begin{array}{ll} [f(L_1),\ldots,f(L_n)] & \text{falls } f(L_1),\ldots,f(L_n) \text{ definiert sind} \\ \text{undefiniert} & \text{sonst} \end{array} \right\}$
für alle $f \in A_{fun}$ und $L_1,\ldots,L_n \in A_{const}$, $n \geq 1$

★ $fold^A(f)([]) = []$ für alle $f \in A_{fun}$
$fold^A(f)([L]) = [L]$ für alle $f \in A_{fun}$ und $L \in A_{const}$
$fold^A(f)([L_1,\ldots,L_n]) = \left\{ \begin{array}{ll} f([L_1, f([L_2,\ldots,f([L_{n-1},L_n])\ldots])]) & \text{falls } f \text{ auf allen} \\ & \text{zweielementigen Listen definiert ist} \\ \text{undefiniert} & \text{sonst} \end{array} \right\}$
für alle $f \in A_{fun}$ und $L_1,\ldots,L_n \in A_{const}$, $n \geq 2$ 🐰

Auf ähnliche Weise könnte auch die Semantik von Teilen der imperativen Beispielsprache IPF definiert werden. Eine entsprechende Σ-Algebra A ist aber erheblich komplizierter als die Semantik von FPF (s. Beispiel 10.1.10). Die entscheidenden Datenbereiche von A sind bereits Funktionenräume zweiter Ordnung. Die Interpretation von Funktionssymbolen von Σ liefert demzufolge Funktionen dritter Ordnung. Der Umgang mit solchen Strukturen erfordert spezielle Rechenregeln, die weit entfernt sind von dem auf abstrakter Syntax und mehrsortiger Prädikatenlogik basierenden Ansatz, den wir hier verfolgen. Die Grundidee dieses Ansatzes besteht in der Beschränkung aller Semantikdefinitionen auf *Herbrand*strukturen und deren Quotienten (s. Def. 4.1.2). Wie wir sehen werden, lassen sich geeignete Herbrandstrukturen direkt aus der Syntax gegebener Spezifikationen ableiten und erfordern deshalb keine spezielle Modellsprache oder -logik. Modelle liefern die semantische Grundlage für Rechenregeln. Die Regeln selbst sind jedoch rein syntaktische Gebilde, deren Korrektheit und Mächtigkeit umso einfacher zu begreifen ist, je enger die Sprache der Modelle der Sprache der Regeln angenähert ist.

Wir fahren zunächst mit den allgemeinen semantischen Begriffen fort.

Belegung, Auswertung

[12] Die Listen [[]] und [] dienen hier der Darstellung der Booleschen Werte *true* bzw. *false*.

Definition 4.1.6 Sei $\Sigma = (S, F, R)$ eine Signatur, A eine Σ-Struktur und $b : X \to A$ eine S-sortierte Funktion (vgl. 3.4), genannt **Belegung von X in A**. Die S-sortierte **Auswertungsfunktion** $b^* : T_\Sigma(X) \to A$ ist induktiv definiert über dem Aufbau von $T_\Sigma(X)$ (vgl. 3.9):

- Für alle $x \in X$ ist $b^*(x) = b(x)$.
- Für alle $w \in S^*,\ s \in S,\ f \in F_{ws}$ und $t \in T_\Sigma(X)_w$ ist $b^*(f(t)) = f^A(b^*(t))$.

Für $x \in X$ und $a \in A$ ist die Belegung $b[a/x]$ ("b bis auf a für x") definiert durch:

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

Sei $s \in S$, $t \in T_\Sigma(X)_s$, $\{x_1, \ldots, x_n\}$ die Menge der Variablen von t. Ist $x_i \in X_{s_i}$ für alle $1 \le i \le n$, dann definiert t eine aus den Funktionssymbolen von t **abgeleitete Funktion** $t^A : A_{s_1 \ldots s_n} \to A_s$:

$$t^A(a_1, \ldots, a_n) \ =_{def} \ b^*(t),$$

wobei $b(x_i) =_{def} a_i$ für alle $1 \le i \le n$. Ist t ein Grundterm, dann ist t^A nullstellig, also $t^A \in A_s$. A heißt **erreichbar** oder **termerzeugt**, wenn für alle $a \in A$ ein $t \in T_\Sigma$ mit $t^A = a$ existiert.

☞ Zeigen Sie, dass für die kleinste Σ-Unterstruktur A_Σ einer Σ-Struktur A (vgl. Def. 4.1.2) und alle $s \in S$

$$A_{\Sigma,s} \ = \ \{t^A \mid t \in T_\Sigma\}$$

gilt. Folgern Sie daraus, dass A_Σ erreichbar ist und genau dann mit A übereinstimmt, wenn A erreichbar ist!

☞ Zeigen Sie, dass jede Herbrandstruktur erreichbar ist!

☞ Zeigen Sie, dass es zwischen je zwei erreichbaren Σ-Strukturen höchstens einen Σ-Homomorphismus gibt!

Substitution

Definition 4.1.7 Eine S-sortierte Funktion $\sigma : X \to T_\Sigma(X)$ heißt **Substitution**. Der **Domain von σ**, $dom(\sigma)$, ist die Menge aller Variablen x mit $\sigma(x) \neq x$. σ ist eine **Grundsubstitution**, wenn $\sigma(x) \in T_\Sigma$ für alle $x \in dom(\sigma)$ gilt. Für alle Mengen T von Termen bezeichnet $\sigma : X \to T$ eine Substitution, die alle Variablen ihres Domains durch Terme von T ersetzt. Enthält $t \in T_\Sigma(X)$ genau n Variablen $x_1, \ldots, x_n$, dann heißt der Term $t\sigma =_{def} t^{T_\Sigma(X)}(\sigma(x_1), \ldots, \sigma(x_n))$ **σ-Instanz von t**. $t\sigma$ ist eine **Grundinstanz von t**, wenn $t\sigma$ ein Grundterm ist.

Ist $dom(\sigma)$ endlich, also z.B. $dom(\sigma) = \{x_1, \ldots, x_n\}$ und $\sigma(x_1) = t_1, \ldots, \sigma(x_n) = t_n$, dann schreibt man auch $[t_1/x_1, \ldots, t_n/x_n]$ oder $[t_i/x_i \mid 1 \le i \le n]$ ("t_i für x_i") anstelle von σ. Diese Schreibweise wird nicht nur für zu ersetzende Variablen, sondern ganze Terme oder Teilformeln verwendet (wie z.B. unten in der Regel der Fixpunktinduktion).

Hier ist eine Haskell-Implementierung von Termen und Substitutionen:

```
data Term = V String | F String [Term]          -- Datentyp fuer Terme

instance Eq Term where

  (==) :: Term -> Term -> Bool                  -- Gleichheit auf Termen als
                                                -- Instanz der Typklasse Eq
```

```
  V x == V y       = x == y
  V _ == F _ _     = False
  F _ _ == V _     = False
  F x ts == F y us = x == y && ts == us

isin :: String -> Term -> Bool                  -- Kommt x in t vor?

x `isin` V y    = x == y
x `isin` F _ ts = any (x `isin`) ts

any :: (a -> Bool) -> [a] -> Bool               -- Test, ob ein Praedikat fuer
                                                -- (Boolesche Funktion) fuer
any f []    = False                             -- mindestens ein Listenelement
any f (a:s) = f a || any f s                    -- gilt

domain :: (String -> Term) -> [String] -> [String]

domain f xs = [x | x <- xs, f x /= V x]         -- Liste der Variablen von xs,
                                                -- die im Domain der Substitution
                                                -- f vorkommen

for :: Term -> String -> String -> Term         -- Erzeugung einer Substitution
                                                -- mit einelementigem Domain
t `for` x = upd V x t

except :: (String -> Term) -> String -> String -> Term
                                                -- Reduktion des Domains einer
                                                -- Substitution um eine Variable
f `except` x = upd f x (V x)

upd :: (a -> b) -> a -> b -> a -> b             -- Update einer Funktion an
                                                -- einer Stelle
upd f a b a' | a == a' = b
             | True    = f a'

(>>>) :: Term -> (String -> Term) -> Term       -- Anwendung einer Substitution
                                                -- auf einen Term
V x >>> f    = f x
F x ts >>> f = F x (map (>>> f) ts)

map :: (a -> b) -> [a] -> [b]                   -- Anwendung einer Funktion auf
                                                -- alle Elemente einer Liste
map f []    = []
map f (a:s) = f a:map f s

andThen :: (String -> Term) -> (String -> Term) -> String -> Term
                                                -- sequentielle Komposition von
(f `andThen` g) x = f x >>> g                   -- Substitutionen
```

Der Wert eines Terms t hängt also ab von der Interpretation der Funktionssymbole von t und einer Belegung der Variablen von t. Substitutionen sind Belegungen in Termalgebren. Die aus $\sigma : X \to T_\Sigma(X)$ gebildete Auswertungsfunktion $\sigma^* : T_\Sigma(X) \to T_\Sigma(X)$ liefert gerade die σ-Instanzen: $\sigma^*(t) = t\sigma$. Komponiert man σ mit einer Auswertungsfunktion $b^* : T_\Sigma(X) \to A$, so entsteht die Belegung $b^* \circ \sigma : X \to A$, die das **Substitutionslemma**

$$(b^* \circ \sigma)^* = b^* \circ \sigma^*$$

erfüllt (s. Fig. 1).

☞ Zeigen Sie das Substitutionslemma!

Gültigkeit, logische Äquivalenz

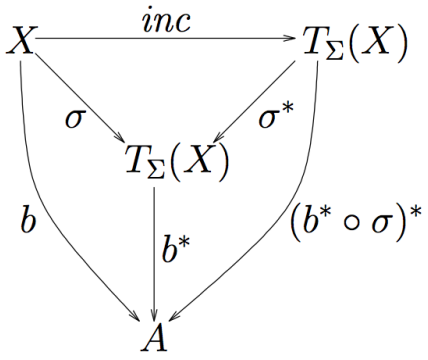


Figure 1. *Auswertung und Substitution*

Definition 4.1.8 Sei φ eine Σ-Formel über X, A eine Σ-Struktur und b eine Belegung von X in A. Die Eigenschaft b **löst** φ in A oder b **erfüllt** φ in A, geschrieben: $A \models_b \varphi$, ist induktiv definiert über dem syntaktischen Aufbau von φ:

- $A \models_b \mathit{True}$ gilt immer.
- $A \models_b \mathit{False}$ gilt niemals.
- $A \models_b r(t) \iff_{def} b^*(t) \in r^A$ für alle Σ-Atome $r(t)$.
- $A \models_b \neg\varphi \iff_{def} A \models_b \varphi$ gilt nicht.
- $A \models_b \varphi \wedge \psi \iff_{def} A \models_b \varphi$ und $A \models_b \psi$.
- $A \models_b \varphi \vee \psi \iff_{def} A \models_b \varphi$ oder $A \models_b \psi$.
- $A \models_b \varphi \Rightarrow \psi \iff_{def} A \models_b \varphi$ impliziert $A \models_b \psi$.
- $A \models_b \varphi \Leftrightarrow \psi \iff_{def} A \models_b (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$.
- $A \models_b \forall x\varphi \iff_{def} A \models_{b[a/x]} \varphi$ für alle $a \in A$.
- $A \models_b \exists x\varphi \iff_{def} A \models_{b[a/x]} \varphi$ für ein $a \in A$.

A **erfüllt** eine Σ-Formel φ oder A ist ein **Modell** von φ oder φ **gilt** in A, geschrieben: $A \models \varphi$, wenn alle Belegungen von X in A φ lösen. φ gilt in einer Klasse $\mathcal{K}$ von Σ-Strukturen, wenn φ in allen ihren Strukturen gilt.

φ ist **allgemeingültig**, wenn φ in allen Σ-Strukturen mit Komplementen gilt. ψ **folgt aus** bzw. ist **logisch äquivalent** zu φ, wenn $\varphi \Rightarrow \psi$ bzw. $\varphi \Leftrightarrow \psi$ allgemeingültig ist.

Eine Σ-Struktur B ist **monoton** bzgl. einer Σ-Struktur A, wenn B alle in A gültigen Grundatome erfüllt.

☞ Sei $b : X \to A$ eine Belegung und $h : A \to B$ ein monotoner Σ-Homomorphismus. Zeigen Sie, dass für alle Σ-Formeln φ gilt:

$$A \models_b \varphi \quad \Longrightarrow \quad B \models_{h \circ b} \varphi. \tag{1}$$

Aus (1) folgt die **elementare Äquivalenz** isomorpher Strukturen A und B, d.h. für alle Σ-Formeln φ gilt:

$$A \models \varphi \quad \Longleftrightarrow \quad B \models \varphi. \tag{2}$$

Ist nämlich $h : A \to B$ ein monotoner Isomorphismus, dann gibt es einen monotonen Homomorphismus $g : B \to A$ mit $g \circ h = id^A$ und $h \circ g = id^B$. Daraus folgt zunächst, dass es zu jeder Belegung $c : X \to B$ eine Belegung $b : X \to A$ mit $h \circ b = c$ gibt, nämlich $b = g \circ c$, und umgekehrt zu jeder Belegung $b : X \to A$ eine Belegung $c : X \to B$ mit $g \circ c = b$ gibt, nämlich $c = h \circ b$. Ausserdem folgt

$$B \models_c \varphi \quad \Longrightarrow \quad A \models_{g \circ c} \varphi \tag{3}$$

aus (1), weil g homomorph und monoton ist. (1) und (3) liefern (2):

$$\begin{array}{l} A \models \varphi \iff \forall\, b : X \to A : A \models_b \varphi \overset{(1)}{\Longrightarrow} \forall\, b : X \to A : B \models_{h \circ b} \varphi \\ \Longrightarrow \forall\, c : X \to B : B \models_c \varphi \iff B \models \varphi \iff \forall\, c : X \to B : B \models_c \varphi \\ \overset{(3)}{\Longrightarrow} \forall\, c : X \to B : A \models_{g \circ c} \varphi \implies \forall\, b : X \to A : A \models_b \varphi \iff A \models \varphi. \end{array}$$

☞ Gilt auch die Umkehrung von (2), d.h. sind alle elementar äquivalenten Strukturen isomorph?

Lemma 4.1.9 (Monotonie und Homomorphie) Sei A eine erreichbare Σ-Struktur und B eine Σ-Struktur mit Gleichheit (siehe Def. 4.1.2). B ist genau dann monoton bzgl. A, wenn es einen monotonen Σ-Homomorphismus $h : A \to B$ gibt.

Beweis. "⇒": Sei B monoton bzgl. A. Da A erreichbar ist, gibt es für alle $a \in A$ ein $t \in T_\Sigma$ mit $t^A = a$. Wir definieren h durch $h(t^A) = t^B$. h ist wohldefiniert: Sei $t^A = u^A$. Da $\equiv^A$ reflexiv ist, folgt $t^A \equiv^A u^A$, also $A \models t \equiv u$. Da B monoton bzgl. A ist, gilt auch $B \models t \equiv u$, also $t^B \equiv^B u^B$ und daher $h(t^A) = t^B = u^B = h(u^A)$, weil B eine Struktur mit Gleichheit ist. Sei $f : w \to s$ ein Funktionssymbol von Σ und $t^A \in A_w$. Dann gilt

$$h(f^A(t^A)) = h(f(t)^A) = f(t)^B = f^B(t^B) = f^B(h(t^A)).$$

Sei r ein Prädikat von Σ und $t^A \in r^A$. Dann gilt $A \models r(t)$, also auch $B \models r(t)$ wegen der Monotonie von B bzgl. A. Daraus folgt $h(t^A) = t^B \in r^B$. Damit haben wir $h(r^A) \subseteq r^B$ gezeigt.

"⇐": Sei $h : A \to B$ ein monotoner Σ-Homomorphismus und $t \in T_\Sigma$. Durch Induktion über die Größe von t zeigt man $h(t^A) = t^B$. Sei $r(t)$ ein Grundatom mit $A \models r(t)$. Dann ist $t^A \in r^A$, also $t^B = h(t^A) \in h(r^A) \subseteq r^B$. Daraus folgt $B \models r(t)$. ❑

Seien φ und ψ Formeln oder Terme. **var(φ)** bezeichnet die Menge der in φ vorkommenden Variablen. Jedes Auftreten von $x \in var(\varphi)$ in einer Teilformel $\forall x\psi$ oder $\exists x\psi$ von φ nennt man ein **gebundenes Vorkommen von x in φ**. Andere Vorkommen von x in φ heißen **frei**. x ist eine **freie Variable von** φ, wenn x in φ frei vorkommt. φ ist **geschlossen**, wenn φ keine freien Variablen enthält.

Substititutionen werden entlang der Syntax von Σ-Formeln auf diese fortgesetzt:

$$\begin{array}{rcll} r(t)\sigma & =_{def} & r(t\sigma) & \\ (\neg\varphi)\sigma & =_{def} & \neg(\varphi\sigma) & \\ (\varphi \wedge \psi)\sigma & =_{def} & \varphi\sigma \wedge \psi\sigma & \\ (\varphi \vee \psi)\sigma & =_{def} & \varphi\sigma \vee \psi\sigma & \\ (\varphi \Rightarrow \psi)\sigma & =_{def} & \varphi\sigma \Rightarrow \psi\sigma & \\ (\varphi \Leftrightarrow \psi)\sigma & =_{def} & \varphi\sigma \Leftrightarrow \psi\sigma & \\ (\forall x\varphi)\sigma & =_{def} & \forall x\varphi\sigma[x/x] & \text{falls für alle } y \in var(\varphi) \setminus \{x\}\ x \notin var(\sigma(y)) \text{ gilt} \\ (\exists x\varphi)\sigma & =_{def} & \exists x\varphi\sigma[x/x] & \text{falls für alle } y \in var(\varphi) \setminus \{x\}\ x \notin var(\sigma(y)) \text{ gilt} \end{array}$$

M.a.W. wird die **σ-Instanz** $\varphi\sigma$ einer Formel φ induktiv über dem Formelaufbau definiert, so wie die σ-Instanz $t\sigma$ eines Terms t induktiv über dem Termaufbau definiert ist. Die Nebenbedingung bei der Instanziierung quantifizierter Formeln stellt sicher, dass σ keine zusätzlichen Vorkommen der gebundenen Variable x im Scope des Quantors erzeugt. Sie lässt sich herstellen, indem man diese Variable vor der Anwendung von σ ggf. so umbenennt, dass sie für kein $y \in var(\varphi) \setminus \{x\}$ in $var(\sigma(y))$ vorkommt.

Ist z.B. $\varphi = \forall x : r(x, y)$ und σ eine Substitution mit $\sigma(y) = f(x)$, dann ist die Nebenbedingung nicht erfüllt und wir erhalten ohne Umbennung von x die—inkorrekte—σ-Instanz $\forall x : r(f(x), x)$, mit Umbenennung—von x in z—jedoch $\forall z : r(f(x), z)$. Semantisch garantiert die Nebenbedingung bei der Instanziierung quantifizierter Formeln, dass für *alle* Σ-Formeln φ die Äquivalenz

$$A \models_b \varphi\sigma \quad \Longleftrightarrow \quad A \models_{b^* \circ \sigma} \varphi \tag{4}$$

gilt, die das Substitutionslemma (s.o.) von Termen auf Formeln fortsetzt.

☞ Zeigen Sie (4)!

Die Schreibweise $\varphi(x_1, \ldots, x_n)$ für eine Formel φ deutet an, dass die Variablen $x_1, \ldots, x_n$ in φ vorkommen können. $\varphi(t_1, \ldots, t_n)$ bezeichnet dann die Instanz $\varphi[t_i/x_i \mid 1 \leq i \leq n]$ von φ.

Nach Definition von $\models$ gilt φ in A genau dann, wenn A die geschlossene Formel $\forall Y : \varphi$ erfüllt, wobei Y die Menge der freien Variablen von φ ist.

So wie ein Term mit n Variablen eine n-stellige Funktion definiert, so liefert eine Σ-Formel φ mit freien Variablen $x_1, \ldots, x_n$ eine n-stellige **abgeleitete Relation** $\varphi^A \subseteq A^n$: Sei $b : X \to A$.

$$(b(x_1), \ldots, b(x_n)) \in \varphi^A \qquad \Longleftrightarrow_{def} \qquad A \models_b \varphi.$$

Die Analogie zwischen der Auswertung von Termen und der Gültigkeit von Formeln wird auch deutlich, wenn man sich klarmacht, dass beide dem gleichen universellen Schema zur induktiven Definition von **Urteilen**

$$state \vdash expression : value$$

("der Ausdruck *expression* hat im Zustand *state* den Wert *value*") folgen, dessen definierende Regeln die Form

$$\frac{state \vdash f(exp_1, \ldots, exp_n) : h(val_1, \ldots, val_n)}{g_1(state) \vdash exp_1 : val_1, \ldots, g_n(state) \vdash exp_n : val_n} \Uparrow$$

haben.[13] Während f der Operator ist, den die Regel definiert, sind $g_1, \ldots, g_n$ und h von der jeweiligen Anwendung dieses Regelschemas bestimmte Funktionen, die aus einem gegebenen Zustand bzw. Wert einen neuen berechnen.

Die Definitionen der Termauswertung und der Formelgültigkeit lassen sich 1:1 in solche Regeln überführen, wenn man sie wie folgt als Urteile definiert:

$$A, b \vdash t : a \qquad \Longleftrightarrow_{def} \qquad b^*(t) = a$$

bzw.

$$\begin{array}{lll} A, b \vdash \varphi : True & \Longleftrightarrow_{def} & A \models_b \varphi \\ A, b \vdash \varphi : False & \Longleftrightarrow_{def} & A \not\models_b \varphi \end{array}$$

Was als Wertemenge angesehen wird (hier sind es Elemente von A bzw. $\{True, False\}$), hängt vom jeweiligen Kontext ab und kann deshalb auch bei derselben Zustands- oder Ausdrucksmenge variieren. So könnte man z.B. auch die *Sorten* von Termen mit diesem Schema definieren:[14]

$$X \vdash t : s \qquad \Longleftrightarrow_{def} \qquad t \in T_\Sigma(X)_s$$

Das entsprechende – auf strukturierte Sorten fortgesetzte – Regelsystem bildet die Grundlage von in Compilern eingesetzten *Typinferenzalgorithmen* (siehe [78], §6.1).

[13] Der Aufwärtspfeil $\Uparrow$ deutet an, dass die Konjunktion der unter dem waagrechten Strich stehenden Aussagen die darüberstehende Aussage impliziert.

[14] Dieses Beispiel begründet die Verwendung des Doppelpunkts in $state \vdash expression : value$.

4.2 Logische Programme, Kalküle und Herbrandmodelle

Wir schauen uns jetzt an, aus welchen Signaturen und prädikatenlogischen Formeln logische Programme zusammengesetzt sind und in welcher daraus abgeleiteten Struktur sie interpretiert werden.

Goal, Hornformel

Definition 4.2.1 Eine Konjunktion von Σ-Atomen heißt **Σ-Goal** (*Anfrage, Zielformel*). Die Formel *True* ist ebenfalls ein Goal. Sei $r(t)$ ein Atom und G ein Goal. Die Formel

$$\varphi \quad = \quad (r(t) \Leftarrow G)$$

heißt **Σ-Hornformel**[15]. $r(t)$ heißt **Kopf** oder **Konklusion**, G **Rumpf** oder **Prämisse von** φ. Ist $G = True$, dann schreiben wir $r(t)$ anstelle von $r(t) \Leftarrow G$.

Eine Menge von Hornformeln mit demselben Kopfprädikat r nennen wir **logisches Programm für** r oder Menge der **Axiome für** r.

In der Syntax von Prolog schreibt man $r(t) \ :- \ r_1(t_1), \ldots, r_n(t_n)$ für die Hornformel

$$r(t) \ \Leftarrow \ r_1(t_1) \wedge \cdots \wedge r_n(t_n).$$

Dieser Programmbegriff weicht ab von dem in Kapitel 3 verwendeten, der (abstrakte) Programme als Terme und nicht als Formeln definiert. Dort ging es auch eher um die Einbettung konkreter Programmiersprachen in die Sprache der Prädikatenlogik. Das kann man auch mit logischen Programmiersprachen wie Prolog machen, wenn es darum geht, die Konstrukte der Sprache — auch die imperativen — in abstrakter Syntax und darauf aufbauender Semantik formal zu erfassen. Dann ist auch ein logisches Programm ein Term, der in einer geeigneten Struktur als funktionales oder relationales Objekt interpretiert wird. Wir wollen hier jedoch keinen Datentyp "Programme" behandeln, sondern die (abstrakten) Objekte, die von Programmen erzeugt, benutzt und verändert werden, sowie die Ein/Ausgabe-Relationen, die die Programme herstellen. Die Prädikatenlogik *ist* unsere (abstrakte) Programmiersprache. Sie dient — im Gegensatz zu den Beispielen aus Kapitel 3 und §4.1 — im folgenden vorwiegend nicht dazu, *über* Programmiersprachen zu reden.

Beispiel 4.2.2 Sei Σ eine Signatur mit drei Sorten: *entry*, *list* und *list*(*list*), vier Funktionssymbolen:

$[] :\rightarrow list$,
$[] :\rightarrow list(list)$,
$\_ : \_ : entry \times list \rightarrow list$,
$\_ : \_ : list \times list(list) \rightarrow list(list)$,

und folgenden Prädikaten:

$\leq, >: entry \times entry$,
$append : list \times list \times list$,
$sorted : list$,
$quicksort, perm : list \times list$,
$part : list \times list(list)$,
$filter : entry \times list \times list \times list$,

[15] In der Terminologie des Automatischen Beweisens werden disjunktive Normalformen als *Klauseln* bezeichnet und solche, die zu Hornformeln im obigen Sinne *oder* zu negierten Goals äquivalent sind, als *Hornklauseln*.

$insert : entry \times list \times list$,

$glue : entry \times list(list) \times list(list)$.

Eine Σ-Struktur A sei wie folgt definiert:

A_{entry} ist eine beliebige Menge,

$A_{list} = \{[a_1, \ldots, a_n] \mid a_1, \ldots, a_n \in A_{entry}\}$,

$A_{list(list)} = \{[L_1, \ldots, L_n] \mid L_1, \ldots, L_n \in A_{list}\}$.

$[]^A =$ leere Liste,

$a :^A [a_1, \ldots, a_n] = [a, a_1, \ldots, a_n]$.

$\leq^A$ ist eine beliebige totale Ordnung auf A_{entry},

$>^A$ ist das Komplement von $\leq^A$,

$([a_1, \ldots, a_n], [b_1, \ldots, b_k], L) \in append^A \iff_{def} [a_1, \ldots, a_n, b_1, \ldots, b_k] = L$,

$L \in sorted^A \iff_{def} L$ ist eine bzgl. $\leq^A$ aufsteigend sortierte Liste,

$(L, L') \in quicksort^A \iff_{def} L'$ ist eine bzgl. $\leq^A$ aufsteigend sortierte Permutation von L,

$(L, L') \in perm^A \iff_{def} L'$ ist eine Permutation von L,

$(L, P) \in part^A \iff_{def} P$ ist eine Partition von L,

$(a, L, low, high) \in filter^A \iff_{def} low = \{x \in L \mid x \leq a\} \wedge high = \{x \in L \mid x > a\}$,

$(a, [], L) \in insert^A \iff_{def} [a] = L$,

$(a, [a_1, \ldots, a_n], L) \in insert^A \iff_{def} \exists 1 \leq i \leq n : [a_1, \ldots, a_i, a, a_{i+1}, \ldots, a_n] = L$,

$(a, [], P) \in glue^A \iff_{def} [[a]] = P$,

$(a, [L_1, \ldots, L_n], P) \in glue^A \iff_{def} \exists 1 \leq i \leq n : [L_1, \ldots, L_{i-1}, a :^A L_i, \ldots, L_{i+1}, \ldots, L_n] = P$.

Seien $X, Y, Z, L, L1, L2, L3, Low, High, P, Q$ sind Variablen. Dann bilden die folgenden Hornformeln logische Programme für die Prädikate von Σ:

```
append([],L,L)
append(X:L,L1,X:L2) <== append(L,L1,L2)

sorted([])
sorted(X:[])
sorted(X:Y:L) <== X <= Y /\ sorted(Y:L)

quicksort([],[])
quicksort(Z:L,L3) <== filter(Z,L,Low,High) /\ quicksort(Low,L1) /\
         quicksort(High,L2) /\ append(L1,Z:L2,L3)

filter(Z,[],[],[]).
filter(Z,X:L,X:Low,High) <== X <= Z /\ filter(Z,L,Low,High)
filter(Z,X:L,Low,X:High) <== X > Z /\ filter(Z,L,Low,High)

perm([],[])
perm(X:L,P) <== perm(L,Q) /\ insert(X,Q,P)

insert(X,Q,X:Q)
insert(X,Y:Q,Y:P) <== insert(X,Q,P)

part([],[])
part(X:L,[X]:Q) <== part(L,Q)
part(X:L,P) <== part(L,Q) /\ glue(X,Q,P)
```

```
glue(X,[],[X]:[])
glue(X,L:Q,(X:L):Q)
glue(X,L:Q,L:P) <== glue(X,Q,P)
```

☞ Zeigen Sie, dass A die angegebenen Hornformeln erfüllt. ✌

Ähnlich der oben erwähnten Semantik von IPF liefert die Struktur A ein spezielles Modell, bzgl. dessen Anforderungen an die logischen Programme verifiziert werden könnten. Das Beispiel läßt aber vermuten, dass ein zu A isomorphes Modell direkt aus den logischen Programmen und ihrer Signatur ableitbar ist. Dieses wird **Herbrandmodell** genannt (siehe Def. 4.2.6). Als Herbrandstruktur erfordert es im Gegensatz zu A zu seiner Definition keine über die oben eingeführten syntaktischen Begriffe hinausgehenden Notationen.

Bevor wir auf die aus logischen Programmen gebildeten Modelle eingehen, wollen wir jedoch ein Beispiel einer Berechnung mit logischen Programmen geben. Solche Berechnungen setzen sich aus Anwendungen der **Resolutionsregel** (siehe Def. 4.2.12) zusammen.[16]. Korrekt sind die Berechnungen bezüglich des eben erwähnten Herbrandmodells. All das wird im Rest des Kapitels genau definiert.

Beispiel 4.2.3 Ein logisches Programm berechnet Substitutionen der Variablen eines Goals (siehe Def. 4.2.1), die es lösen. Wir wählen das Goal

$$\exists\, q : (perm([1,2,3], p) \wedge append(q, [2], p)). \tag{1}$$

Hier wird nach Permutationen der Liste [1,2,3] gesucht, die mit einer 2 enden. Der eingefügte Existenzquantor bewirkt, dass nur die Substitution der Variablen p in die endgültige Lösung des Goals aufgenommen wird. Dazu muss zwar auch eine Lösung von q berechnet werden. Die wird aber am Schluss wieder vergessen. Die entscheidende Regel, mit der solche Berechnungen durchgeführt wird, ist die Resolution (Def. 4.2.12). ✌

Berechnungen logischer Programme sind nichts anderes als logische **Ableitungen**. Diesen Begriff gilt es zunächst genauer zu definieren.

Kalkül, Ableitung

Definition 4.2.4 Ein **Kalkül** (**Ableitungs-, Deduktions-** oder **Inferenzsystem**) ist eine endliche Menge $\mathcal{K}$ von **Beweis-, Deduktions-, Ableitungs-** oder **Inferenzregeln** der Form

$$\frac{\varphi_1, \ldots, \varphi_m}{\psi_1, \ldots, \psi_n},$$

die festlegen, aus welchen **Antezedenten** $\varphi_1, \ldots, \varphi_n$ welche **Sukzedenten** $\psi_1, \ldots, \psi_n$ geschlossen werden kann. Ein leerer Antezedent steht für die Formel *True*, ein leerer Sukzedent für *False*.

Eine Folge $\vartheta_1, \ldots, \vartheta_k$ von Formeln heißt **$\mathcal{K}$-Beweis** oder **-Ableitung** von ϑ_k aus ϑ_1, falls es für alle $1 \leq i \leq k$ $\mathcal{K}$ eine Regel

$$\frac{\varphi_1, \ldots, \varphi_m}{\psi_1, \ldots, \psi_n}$$

gibt mit $\varphi_1, \ldots, \varphi_m \in \{\vartheta_1, \ldots, \vartheta_{i-1}\}$ und $\vartheta_i \in \{\psi_1, \ldots, \psi_n\}$.

Die **Ableitungs-** oder **Inferenzrelation** $\vdash_{\mathcal{K}}$ von $\mathcal{K}$ ist definiert durch:

$$\varphi \vdash_{\mathcal{K}} \psi \quad \Longleftrightarrow_{def} \quad \text{es gibt einen } \mathcal{K}\text{-Beweis von } \psi \text{ aus } \varphi.$$

I.a. bestehen die Antezedenten und Sukzedenten von Inferenzregeln aus Formel*schemata* mit Formelvariablen, die erst bei der Anwendung einer Regel durch komplexe Formeln ersetzt werden, analog der Substitution von

[16] Im folgenden Beispiel ist von **Narrowing** und nicht von Resolution die Rede, weil Resolution eine Regel des Narrowing-Kalküls, der neben dieser zwei weitere Regeln zur Anwendung von Hornformeln enthält (siehe Def. 5.1.22)

Variablen durch Terme (s. 4.1.7). Beweisen bestehen demzufolge aus *Instanzen* von Ante- oder Sukzedenten angewendeter Regeln.

Kalküle werden benutzt, um Rechenregeln zu formulieren, Übersetzungsschritte zu präzisieren oder die operationelle Semantik einer Programmiersprache zu beschreiben. Im letzten Fall repräsentiert jede abgeleitete Formel einen Zustand (auch *Konfiguration* genannt) eines Interpreters der Sprache. Konfigurationen enthalten neben Programmstücken i.a. weitere *Attribute*, die der Interpreter zur Ausführung der Programme benötigt. Ein **Kalkül ist korrekt** bzgl. einer gegebenen Semantik (= Struktur oder Menge von Strukturen) A, wenn die Antezendenten seiner Regeln zu den jeweiligen Sukzedenten in einer bestimmten semantischen Beziehung stehen. Generell gibt es drei Möglichkeiten:

$$\frac{\varphi_1, \ldots, \varphi_m}{\psi_1, \ldots, \psi_n} \Downarrow \quad (1)$$

$$\frac{\varphi_1, \ldots, \varphi_m}{\psi_1, \ldots, \psi_n} \Uparrow \quad (2)$$

$$\frac{\varphi_1, \ldots, \varphi_m}{\psi_1, \ldots, \psi_n} \Updownarrow \quad (3)$$

Seien Y und Z die Mengen der freien Variablen von $\varphi_1, \ldots, \varphi_m$ bzw. $\psi_1, \ldots, \psi_n$. (1), (2) bzw. (3) ist **korrekt bzgl.** A, falls in A

- im Fall (1) $\exists Z(\psi_1 \vee \cdots \vee \psi_n)$ aus $\forall Y(\varphi_1 \wedge \cdots \wedge \varphi_m)$ folgt,
- im Fall (2) $\forall Y(\varphi_1 \wedge \cdots \wedge \varphi_m)$ aus $\exists Z(\psi_1 \vee \cdots \vee \psi_n)$ folgt,
- im Fall (3) diese beiden Formeln äquivalent sind.

Spezifikation, Schnittkalkül

Definition 4.2.5 Sei Σ eine Signatur und AX eine Menge von Σ-Hornformeln. Das Paar $SP = (\Sigma, AX)$ nennen wir **Spezifikation**. Der **Schnittkalkül für** SP besteht aus folgenden Regeln zur Ableitung von Σ-Formeln:

Axiome	$\frac{True}{\varphi} \Downarrow$	für alle $\varphi \in AX$
Instanziierung	$\frac{\varphi}{\varphi[t/x]} \Downarrow$	für alle $t \in T_\Sigma(X)_{sort(x)}$
Modus Ponens	$\frac{\varphi, \quad \varphi \Rightarrow \psi}{\psi} \Downarrow$	
$\wedge$-Einführung	$\frac{\varphi, \quad \psi}{\varphi \wedge \psi} \Downarrow$	

$\vdash_{cut}$ bezeichnet die Inferenzrelation des Schnittkalküls (für SP).

☞ Zeigen Sie, dass der Schnittkalkül für SP bzgl. jeder Σ-Struktur, die AX erfüllt (s. Def. 4.1.8), korrekt ist!

Herbrandmodell

Definition 4.2.6 Sei $\Sigma = (S, F, R)$ eine Signatur und $SP = (\Sigma, AX)$ eine Spezifikation. Durch folgende Interpretation von R wird die Termalgebra T_Σ zu einer Herbrandstruktur (s. Def. 4.1.2), dem **Herbrandmodell Her(SP)** von SP:

❀ Für alle $r \in R$, $r^{Her(SP)} =_{def} \{t \in T_\Sigma \mid True \vdash_{cut} r(t)\}$.

Satz 4.2.7 Für alle Grundgoals G gilt:

$$Her(SP) \models G \quad \Longleftrightarrow \quad True \vdash_{cut} G. \tag{1}$$

Für alle prädikatenlogischen Formeln φ gilt:

$$Her(SP) \models \varphi \quad \Longleftrightarrow \quad \forall \sigma : X \to T_\Sigma : Her(SP) \models \varphi\sigma. \tag{2}$$

Beweis. ☞ Übung. ❑

Wegen des induktiven Aufbaus von T_Σ läßt sich $\forall \sigma : \varphi\sigma$ i.a. durch Induktion zeigen. Wegen (2) nennt man eine im Herbrandmodell von SP gültige Formel deshalb auch **induktives Theorem** von SP.

$Her(SP)$ interpretiert die Prädikate von Σ als *kleinste* Relationen, die alle Σ-Formeln von AX erfüllen:

Satz 4.2.8 Sei AX eine Menge von Σ-Hornformeln.

❶ $Her(SP)$ ist ein Modell von AX.

❷ $Her(SP)$ ist das **kleinste Modell** von AX, d.h. für alle Σ-Herbrandstrukturen A, die AX erfüllen, und alle Prädikate $r \in \Sigma$ gilt $r^{Her(SP)} \subseteq r^A$.

❸ Für alle Modelle A von AX existiert eindeutig ein monotoner Σ-Homomorphismus $h : Her(SP) \to A$.

Beweis. ❶ Sei $\varphi = (r(t) \Leftarrow H) \in AX$, $H = r_1(t_1) \wedge \cdots \wedge r_n(t_n)$ und $b : X \to T_\Sigma$ eine Belegung mit $Her(SP) \models_b H$. Dann gilt $b^*(t_i) \in r_i^{Her(SP)}$ für alle $1 \leq i \leq n$. Da $Her(SP)$ Funktionssymbole genauso wie T_Σ interpretiert, gibt es eine Substitution σ mit $u\sigma = b^*(u)$ für alle $u \in T_\Sigma(X)$. Also gilt $True \vdash_{cut} r_i(t_i\sigma)$ nach Definition von $r_i^{Her(SP)}$ für alle $1 \leq i \leq n$. Eine Anwendung der Instanziierungsregel des Schnittkalküls auf φ liefert $True \vdash_{cut} \varphi\sigma$. n-1 Anwendungen der $\wedge$-Einführung liefern $True \vdash_{cut} r_1(t_1\sigma) \wedge \cdots \wedge r_n(t_n\sigma)$. Wendet man den Modus Ponens auf $\varphi\sigma$ und $r_1(t_1\sigma) \wedge \cdots \wedge r_n(t_n\sigma)$ an, erhält man $True \vdash_{cut} r(t\sigma)$, also $b^*(t) = t\sigma \in r^{Her(SP)}$ und damit $Her(SP) \models_b r(t)$. Deshalb ist φ in $Her(SP)$ gültig.

❷ Nach Definition von $r^{Her(SP)}$ ist ❷ äquivalent zu folgender Bedingung. Für alle Σ-Herbrandstrukturen A und alle Grundatome $r(t)$ gilt:

$$A \models AX \ \wedge \ True \vdash_{cut} r(t) \quad \Rightarrow \quad A \models r(t).$$

Das folgt aber aus der Korrektheit des Schnittkalküls (s.o.).

❸ Sei HA die folgendermaßen definierte Σ-Herbrandstruktur. Für alle Prädikate $r \in \Sigma$ besteht r^{HA} aus allen $t \in T_\Sigma$ mit $t^A \in r^A$. Mit A ist auch HA ein Modell von AX. Da $Her(SP)$ nach ❷ das kleinste Modell von AX ist, folgt, dass A monoton bzgl. $Her(SP)$ ist (s. 4.1.2):

$$Her(SP) \models r(t) \ \Rightarrow \ t \in r^{Her(SP)} \ \Rightarrow \ t \in r^{HA} \ \Leftrightarrow \ t^A \in r^A \ \Rightarrow \ A \models r(t).$$

Demnach liefert Lemma 4.1.9 die Behauptung. Aus dem Beweis des Lemmas folgt auch die Eindeutigkeit von h. ❑

Beispiel 4.2.9 Seien $SP = (\Sigma, AX)$ und A wie in Bsp. 4.2.2. Es ist leicht zu sehen, dass es eine Bijektion h von $Her(SP)$ nach A gibt. Der Liste $[a_1, \ldots, a_n] \in A_{list}$ entspricht z.B. der Σ-Term $t_1 : (t_2 : \cdots : (t_n : [])\ldots)$, wobei $t_1, \ldots, t_n$ geeignete Grundtermdarstellungen der Listenelemente $a_1, \ldots, a_n \in A_{entry}$ sind. Da A ein Modell von AX ist, folgt aus ❸ , dass $h(r^{Her(SP)}) \subseteq r^A$ für alle sieben Prädikate $r \in \Sigma$ gilt. Tatsächlich gilt auch die Umkehrung $r^A \subseteq h(r^{Her(SP)})$, was gleichbedeutend ist mit:

$$\forall\, a \in r^A \ \exists\, t \in T_\Sigma : h(t) = a \wedge True \vdash_{cut} r(t). \tag{1}$$

Zeigen Sie (1) durch Induktion über die Länge von Listen. ✌

Die Regeln des Schnittkalküls sind zwar sehr einfach, aber zur praktischen Verifikation wenig geeignet. Die Charakterisierung von $Her(SP)$ als kleinstes Modell von AX hingegen liefert Inferenzregeln, mit denen Anforderungen an AX zum Teil sehr schnell bewiesen werden können. So impliziert Satz 4.2.8, dass die folgende Beweisregel bzgl. $Her(SP)$ korrekt ist. Sei r ein Prädikat von Σ und $AX_r = \{\varphi \in AX \mid r \text{ kommt in } \varphi \text{ vor}\}$

Fixpunktinduktion über r $\quad \dfrac{r(x) \;\Rightarrow\; \psi(x)}{\bigwedge_{\varphi \in AX_r} \varphi[\psi(u)/r(u) \mid r(u) \text{ kommt in } \varphi \text{ vor}]} \Uparrow$

☞ Begründen Sie diese Behauptung!

In der Literatur über logische Programmierung ([59],[4]) wird $Her(SP)$ als **deklarative Semantik** bezeichnet und nicht über ein Inferenzsystem, sondern als Supremum approximierender Modelle konstruiert. Die Modelleigenschaft ergibt sich aus dem Fixpunktsatz von Kleene[17] (4.2.11), der eine Spezialisierung des Fixpunktsatzes von Knaster-Tarski ist:[18]

Satz 4.2.10 (Fixpunktsatz von Knaster-Tarski) Sei A ein **vollständiger Verband**, d.i. eine Menge mit Halbordnung $\leq$, bezüglich der A ein kleinstes Element $\bot$, ein größtes Element $\top$ sowie ein Supremum $\sqcup B$ und ein Infimum $\sqcap B$ für jede Teilmenge B von A enthält. Weiterhin sei $\Phi : A \to A$ eine bezüglich $\leq$ monotone Funktion. $a \in A$ ist ein **Fixpunkt** von Φ, wenn $\Phi(a) = a$ gilt.

$$lfp(\Phi) \;=_{def}\; \sqcap\{a \in A \mid \Phi(a) \leq a\}$$

und

$$gfp(\Phi) \;=_{def}\; \sqcup\{a \in A \mid a \leq \Phi(a)\}$$

sind der kleinste bzw. größte Fixpunkt von Φ. ❑

Satz 4.2.11 (Fixpunktsatz von Kleene) Sei A wieder ein vollständiger Verband. $\Phi : A \to A$ ist **aufwärtsstetig**, wenn für alle **aufsteigenden Ketten** $a_1 \leq a_2 \leq a_3 \leq \ldots$ von Elementen aus A

$$\Phi(\sqcup_{i \in \mathbb{N}} a_i) \;\leq\; \sqcup_{i \in \mathbb{N}} \Phi(a_i)$$

gilt. Φ ist **abwärtsstetig**, wenn für alle **absteigenden Ketten** $a_1 \geq a_2 \geq a_3 \geq \ldots$ von Elementen aus A

$$\sqcap_{i \in \mathbb{N}} \Phi(a_i) \;\leq\; \Phi(\sqcap_{i \in \mathbb{N}} a_i)$$

gilt. Ist Φ monoton und aufwärtsstetig, dann ist der **obere Kleene-Abschluss**

$$\Phi^{\infty} \;=_{def}\; \sqcup_{i \in \mathbb{N}} \Phi^i(\bot)$$

der kleinste Fixpunkt von Φ. Ist Φ monoton und abwärtsstetig, dann ist der **untere Kleene-Abschluss**

$$\Phi_{\infty} \;=_{def}\; \sqcap_{i \in \mathbb{N}} \Phi^i(\top)$$

der größte Fixpunkt von Φ. ❑

☞ Zeigen Sie, dass Φ genau dann monoton ist, wenn $\sqcup_{i \in \mathbb{N}} \Phi(a_i) \leq \Phi(\sqcup_{i \in \mathbb{N}} a_i)$ gilt.

☞ Zeigen Sie, dass Φ genau dann monoton ist, wenn $\Phi(\sqcap_{i \in \mathbb{N}} a_i) \leq \sqcap_{i \in \mathbb{N}} \Phi(a_i)$ gilt.

☞ Beweisen Sie die Sätze 4.2.10 und 4.2.11!

☞ Sei Φ monoton. Zeigen Sie $\Phi^{\infty} \leq lfp(\Phi)$ und $gfp(\Phi) \leq \Phi_{\infty}$!

[17] sprich: Klini
[18] Daher kommt auch die Bezeichnung "Fixpunktinduktion".

☞ Sei Φ monoton. Folgern Sie $lfp(\Phi) = \Phi^\infty$ aus $\Phi(\Phi^\infty) \leq \Phi^\infty$ sowie $gfp(\Phi) = \Phi_\infty$ aus $\Phi(\Phi_\infty) \leq \Phi_\infty$![19]

☞ Sei Φ monoton. Zeigen Sie, dass $lfp(\Phi)$ das bzgl. $\leq$ größte Element $a \in A$ mit $\Phi(a) \leq a$ ist! Zeigen Sie, dass $gfp(\Phi)$ das bzgl. $\leq$ kleinste Element $a \in A$ mit $a \leq \Phi(a)$ ist!

Mithilfe von Satz 4.2.10 läßt sich das Herbrandmodell einer Spezifikation SP als kleinster Fixpunkt konstruieren. Dazu machen wir die Menge $\mathcal{H}$ aller Σ-Herbrandstrukturen zum vollständigen Verband: Seien $B, C \in \mathcal{H}$.

$$B \leq C \quad \Longleftrightarrow_{def} \quad \text{für alle Prädikate } r : w \in \Sigma,\ r^B \subseteq r^C.$$

Sei $r : w \in \Sigma$ ein Prädikat.

$$r^{\sqcup\mathcal{B}} =_{def} \bigcup_{B\in\mathcal{B}} r^B, \qquad r^\bot =_{def} \emptyset, \qquad r^{\sqcap\mathcal{B}} =_{def} \bigcap_{B\in\mathcal{B}} r^B, \qquad r^\top =_{def} T_{\Sigma,w}.$$

Da Sorten und Funktionssymbole in jeder Σ-Herbrandstruktur gleich interpretiert werden, unterscheiden sich Σ-Herbrandstrukturen nur in der Interpretation der Prädikate von Σ. Dementsprechend ist eine Funktion $\Phi : \mathcal{H} \to \mathcal{H}$ vollständig definiert, wenn man für alle $B \in \mathcal{H}$ die Prädikate von Σ in der Bildstruktur $\Phi(B)$ interpretiert.

Sei $r : w$ ein Prädikat von Σ, $r(t_1) \Leftarrow \varphi_1, \ldots, r(t_n) \Leftarrow \varphi_n$ die Hornformeln von AX mit Kopfprädikat r und X_i, $1 \leq i \leq n$, die Menge der Variablen von $r(t_i) \Leftarrow \varphi_i$. Wir nennen die Σ-Formel

$$\varphi_{r,AX}(x) \quad =_{def} \quad \bigvee_{i=1}^{n} \exists X_i : (x \equiv t_i \wedge \varphi_i),$$

die **AX-Definition von r**.

☞ Zeigen Sie, dass die Hornformel $r(x) \Leftarrow \varphi_{r,AX}(x)$ zur Konjunktion der Hornformeln $r(t_i) \Leftarrow \varphi_i$, $1 \leq i \leq n$, logisch äquivalent ist!

Die für alle $B \in \mathcal{H}$ durch

$$r^{\Phi(B)} \quad =_{def} \quad \varphi^B_{r,AX} \tag{1}$$

definierte Funktion $\Phi : \mathcal{H} \to \mathcal{H}$ heißt **von AX induzierte Schrittfunktion**.[20]

Nach Definition von Φ und $\leq$ ist Φ monoton bzgl. $\leq$, wenn für alle $B, C \in \mathcal{H}$, Σ-Goals G und $b : X \to T_\Sigma$ gilt:

$$B \leq C \wedge B \models_b G \quad \Rightarrow \quad C \models_b G. \tag{2}$$

☞ Zeigen Sie (2) (durch Induktion über den Aufbau von G)!

☞ Zeigen Sie, dass $lfp(\Phi) = \Phi^\infty$ für die von AX induzierte Schrittfunktion Φ gilt, wenn AX **terminiert**, d.h. wenn es $n \in \mathbb{N}$ mit $\Phi^n(\bot) = \Phi^{n+1}(\bot)$ gibt.

Nach Definition von Φ ist eine Herbrandstruktur B genau dann ein Fixpunkt von Φ, wenn für alle Prädikate $r : w \in \Sigma$

$$r^B \quad = \quad \varphi^B_{r,AX}, \tag{3}$$

m.a.W. B erfüllt $r(x) \Longleftrightarrow \varphi_{r,AX}(x)$. Da $r(x) \Leftrightarrow \varphi_{r,AX}(x)$ logisch äquivalent zur Konjunktion der Axiome für r ist, bedeutet $r^B \subseteq \varphi^B_{r,AX}$, dass B ein Modell von AX ist. Also ist jeder Fixpunkt von Φ ein Modell von AX. Da Φ monoton ist, hat Φ nach Satz 4.2.10 den kleinsten Fixpunkt $lfp(\Phi)$. Dieser ist also ein Modell von AX. Da $Her(SP)$ nach Satz 4.2.8❷ das kleinste Modell von AX ist, folgt $r^{Her(SP)} \subseteq r^{lfp(\Phi)}$. Die Umkehrung

[19] Demnach ist es für die Fixpunkteigenschaft eines Kleene-Abschlusses hinreichend, wenn Φ monoton ist und auf- bzw. abwärtsstetig bzgl. der Kette, aus dem er gebildet ist! Interessante Beispiele, die nur diese eingeschränkte Stetigkeit haben, sind aber nicht bekannt.

[20] $\varphi^B_{r,AX}$ ist die aus $\varphi_{r,AX}$ abgeleitete Relation auf B (siehe §4.1).

$r^{lfp(\Phi)} \subseteq r^{Her(SP)}$ gilt, wenn $Her(SP)$ selbst ein Fixpunkt von Φ ist, also (3) für $B = Her(SP)$ gilt, was im Folgenden gezeigt wird.

Sei $t \in r^{Her(SP)}$. Dann gibt es einen Schnittkalkül-Beweis von $r(t)$ aus *True*, also ein Axiom $r(u) \Leftarrow G$ und eine Substitution $\sigma : X \to T_\Sigma$ mit $u\sigma = t$ und $True \vdash_{cut} G\sigma$. Daraus folgt $Her(SP) \models G\sigma$ und damit $Her(SP) \models \varphi_{r,AX}[t/x]$ nach Definition von $\varphi_{r,AX}$. Also ist $t \in \varphi_{r,AX}^{Her(SP)}$. Die Umkehrung $\varphi_{r,AX}^{Her(SP)} \subseteq r^{Her(SP)}$ gilt, weil $Her(SP)$ ein Modell von AX und damit von $r(x) \Leftarrow \varphi_{r,AX}(x)$ ist.

Also ist der kleinste Fixpunkt von φ tatsächlich das Herbrandmodell von SP.

Da Φ aufwärtsstetig ist[21], wendet man hier oft auch Kleene's Fixpunktsatz an. Die Eigenschaften von Φ werden natürlich von der Syntax der Formeln von AX bestimmt. Nicht-Hornformeln können die Stetigkeit verletzen. So bemüht man in der Modallogik eher den allgemeineren Satz von Knaster-Tarski, um Fixpunkte einzuführen. Kann man Kleene's Satz anwenden, dann erhält man jedoch eine iterative *Konstruktion* der Fixpunkte, aus der sich ein induktives Beweisprinzip ergibt. Diese Induktion stimmt im Fall der Schrittfunktion Φ mit der Induktion über die Länge von Schnittkalkülableitungen überein.

☞ Um die letzte Behauptung zu präzisieren, zeigen Sie, dass für alle Prädikate $r \in \Sigma$ und $i \in \mathbb{N}$ eine Grundterm(tupel) t genau dann zur Interpretation von r in $\Phi^i(\emptyset)$ gehört, wenn es eine Schnittkalkülableitung von $r(t)$ aus *True* mit i Anwendungen des Modus Ponens gibt.

Im nächsten Kapitel werden wir sehen, wie logische Programme zu *funktional*-logischen Programmen erweitert werden können und zwar so, dass immer noch ein Herbrandmodell mit den Eigenschaften von Satz 4.2.8 existiert.

Sowohl für die Definition des Herbrandmodells als auch für Satz 4.2.8 ist die Beschränkung von AX auf Hornformeln entscheidend. Wären Negationen in AX zugelassen, dann könnte AX Widersprüche enthalten, z.B. $r(x) \wedge \neg r(x)$, woraus folgen würde, dass AX überhaupt kein Modell hat. Wären Disjunktionen möglich, z.B. $r(x) \vee \neg r(x)$, dann kann es mehrere minimale Modelle, aber kein kleinstes geben. "Kleinstes Modell" bedeutet auch, dass die Interpretation der Prädikate *ausschließlich* durch ihre Programme bestimmt ist. Das nennen Logik-Programmierer die **closed world assumption**. Sie erlaubt uns, Aussagen über SP zu widerlegen: Ist ein Fakt $r(t)$ nicht aus AX ableitbar, dann gilt $\neg r(t)$ im Herbrandmodell von SP (**negation as failure**). Erlaubt man in den Prämissen von Hornformeln neben Atomen auch *Literale* der Form $\neg r(t)$, dann kommt man zu **default-Logiken** [15]. Eine Alternative bilden **Coprädikate**, mit deren Hilfe sich Komplementprädikate axiomatisieren lassen, so dass explizite Negation überflüssig wird (siehe §6.1).

Wir wollen jetzt den Zusammenhang zwischen der oben definierten *deklarativen* Semantik logischer Programme und ihrer *operationellen Semantik* herstellen. Wie wird AX ausgeführt und in welchem Sinne ist die Ausführung korrekt bzgl. $Her(SP)$?

Ein Σ-Goal G (s. 4.2.1) stellt einen Aufruf der in G vorkommenden Prädikate dar und ist gleichzeitig eine Eingabe für AX. Die zugehörige Ausgabe ist eine Lösung von G in $Her(SP)$, d.h. eine Substitution τ mit $True \vdash_{cut} G\tau$ (s. 4.1.8). Berechnet wird τ mithilfe der folgenden Regel, die sog. Constraints transformiert:

Constraint, Resolution

Definition 4.2.12 Sei $SP = (\Sigma, AX)$ eine Spezifikation. Ein **Constraint** ist ein Paar $\langle G, \sigma \rangle$, bestehend aus einem Goal G und einer Substitution σ derart, dass keine Variable des Domains von σ in G vorkommt (s. 4.1.7). Sei $\sigma : X \to T_\Sigma(X)$.

Resolution $\dfrac{\langle G \wedge r(t), \tau \rangle}{\langle G\sigma \wedge H\sigma, \tau\sigma \rangle}$[22] $\Uparrow$[23] falls $t\sigma = u\sigma$, $r(u) \Leftarrow H \in AX$ und der Sukzedent wieder ein Constraint ist

[21] Das ist der Fall, weil die Prämisse einer Hornformel keine Allquantoren enthält.

$\vdash_{SP}$ bezeichnet die Inferenzrelation der Resolution mit AX.

Die folgende Regel optimiert die Resolutionsregel insofern, als $r(t)$ nur dann ersetzt wird, wenn ein Teil von H, der sog. **Wächter** des angewendeten Axioms, lösbar ist:

bewachte Resolution $\dfrac{\langle G \wedge r(t), \tau\rangle}{\langle G\sigma \wedge H_1\sigma, \tau\sigma\rangle} \Uparrow$ falls $t\sigma = u\sigma$, $r(u) \Leftarrow H_0 \wedge H_1 \in AX$, $Her(SP) \models H_0\sigma$ und der Sukzedent wieder ein Constraint ist

Um deutlich zu machen, welcher Teil der Prämisse einer Hornformel als Wächter vor deren Anwendung gelöst werden soll und welcher die Konklusion ersetzt, schreiben wir auch

$$H_0 \Rightarrow (r(u) \Leftarrow H_1) \qquad \text{anstelle von} \qquad r(u) \Leftarrow H_0 \wedge H_1.$$

Die logische Programmiersprache PARLOG [40] trennt H_0 durch einen Doppelpunkt von H_1.

Grob gesagt ist Resolution ein umgekehrter Modus Ponens: Aus $G\sigma$ und (einem Axiom) $G \Leftarrow H$ leitet man $H\sigma$ ab. Im Gegensatz zu Schnittkalkülableitungen gibt es keine *beliebigen* Instanziierungsschritte.

Das Constraint $\langle G, \sigma\rangle$ entspricht semantisch dem Goal

$$\exists\, Z : G \wedge \bigwedge_{x \in dom(\sigma)} x \equiv x\sigma,$$

wobei Z die Menge aller in der Formel vorkommenden, aber nicht zum Domain von σ gehörenden Variablen und $\equiv$ das Gleichheitsprädikat ist (vgl. §5.1). Beim Beweis von Eigenschaften der Resolution (und des Narrowing; siehe §5.1) ist es jedoch vorteilhaft, die kompaktere Constraint-Notation zu verwenden.

Beispiel 4.2.13 Wir geben die Berechnung der Lösungen des Goals aus Beispiel 4.2.3 so wieder, wie sie mit dem Beweiseditor Expander2 [81] erstellt wurde. Hier steht `Any` für den Existenzquantor, `&` für die konjunktive und `|` für die disjunktive Verknüpfung. Letztere verbindet die Zwischengoals verschiedener Berechnungszweige, die zu verschiedenen Lösungen führen können. Eine Aufspaltung der Berechnung in mehrere Fälle erfolgt immer dann, wenn mehrere Hornformeln auf dasselbe Atom des jeweiligen Zwischengoals anwendbar sind. Alle möglichen Anwendungen werden dann in einem Schritt parallel ausgeführt. Jedem Resolutionsschritt (hier Narrowing genannt) folgen **Simplifikationsschritte**, die das jeweilige Redukt soweit wie möglich vereinfachen. Ohne Simplifikationen würde sich das das zu lösende Goal gewaltig aufblähen und, da tautologische oder widersprüchliche Teilformeln viel zu spät erkannt würden, wäre nicht nur der Platzverbrauch erheblich, sondern auch der Zeitaufwand inakzeptabel.

Simplifikationen bestehen nicht nur in der Eliminierung logischer Operatoren, sondern auch in der **partiellen Auswertung** von Termen und Atomen, also der Anwendung von Gleichungen bzw. logischen Äquivalenzen, die im Herbrandmodell immer wiederkehrender Basistypen gelten.[24]. In Beispiel 4.2.2 bietet es sich z.B. an, Atome mit dem Prädikat *append* nicht durch Resolution mit seinen beiden Axiomen

```
append([],L,L)
append(X:L,L1,X:L2) <== append(L,L1,L2),
```

sondern durch Simplifikation der zum Atom $append(t, u, v)$ äquivalenten Gleichung $t +\!\!+ u = v$ auszuwerten. Damit entsprechende Simplifikationsschritt automatisch auf die Zwischenformeln des Beweises von $\exists\, q$:

[22] Die Komposition von Substitutionen ist von links nach rechts zu lesen: $\tau\sigma$ bedeutet $\sigma^* \circ \tau$.
[23] Die Bedeutung der Implikation zwischen Constraints ergibt sich aus Satz 4.2.14❶ .
[24] Mehr über Simplifikationen findet sich in §7.1

$(perm([1,2,3],p) \wedge append(q,[2],p))$ angewendet werden, lösen wir nicht diese Goal, sondern das logisch äquivalente $\exists\, q : (perm([1,2,3],p) \wedge q +\!\!+ [2] = p)$. Es bleiben die Axiome für das Prädikat *perm* und sein Hilfsprädikat *insert*:

```
perm([],[])
perm(X:L,P) <== perm(L,Q) & insert(X,Q,P)

insert(X,Q,X:Q)
insert(X,Y:Q,Y:P) <== insert(X,Q,P)
```

Die Resolutionsschritte in der folgenden Ableitung wenden also ausschließlich diese vier Axiome an.

```
Any q:(perm([1,2,3],p) & q++[2] = p)

Narrowing the preceding formula leads to

Any q:(Any q0:(perm([2,3],q0) & insert(1,q0,p)) & q++[2] = p)

Simplifying the preceding formula (2 steps) leads to

Any q0:(perm([2,3],q0) & insert(1,q0,p)) & Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any q0:(Any q1:(perm([3],q1) & insert(2,q1,q0)) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any q0:(Any q1:(Any q2:(perm([],q2) & insert(3,q2,q1)) & insert(2,q1,q0)) &
        insert(1,q0,p)) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any q0:(Any q1:(Any q2:(q2 = [] & insert(3,q2,q1)) & insert(2,q1,q0)) &
        insert(1,q0,p)) &
Any q:(q++[2] = p)

Simplifying the preceding formula (2 steps) leads to

Any q0:(Any q1:(insert(3,[],q1) & insert(2,q1,q0)) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any q0:(Any q1:(q1 = 3:[] & insert(2,q1,q0)) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Simplifying the preceding formula (3 steps) leads to

Any q0:(insert(2,[3],q0) & insert(1,q0,p)) & Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any q0:((q0 = 2:[3] | Any p5:(insert(2,[],p5) & q0 = 3:p5)) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Simplifying the preceding formula (6 steps) leads to
```

```
insert(1,[2,3],p) & Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

(p = 1:[2,3] | Any p6:(insert(1,[3],p6) & p = 2:p6)) & Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Simplifying the preceding formula (9 steps) leads to

Any p6:(insert(1,[3],p6) & p = 2:p6) & Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any p6:((p6 = 1:[3] | Any p7:(insert(1,[],p7) & p6 = 3:p7)) & p = 2:p6) &
Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Simplifying the preceding formula (14 steps) leads to

Any p6:(Any p7:(insert(1,[],p7) & p6 = 3:p7) & p = 2:p6) & Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any p6:(Any p7:(p7 = 1:[] & p6 = 3:p7) & p = 2:p6) & Any q:(q++[2] = p) |
Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Simplifying the preceding formula (17 steps) leads to

Any q0:(Any p5:(insert(2,[],p5) & q0 = 3:p5) & insert(1,q0,p)) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

Any q0:(Any p5:(p5 = 2:[] & q0 = 3:p5) & insert(1,q0,p)) & Any q:(q++[2] = p)

Simplifying the preceding formula (9 steps) leads to

insert(1,[3,2],p) & Any q:(q++[2] = p)

Narrowing the preceding formula leads to

(p = 1:[3,2] | Any p10:(insert(1,[2],p10) & p = 3:p10)) & Any q:(q++[2] = p)

Simplifying the preceding formula (7 steps) leads to

p = [1,3,2] | Any p10:(insert(1,[2],p10) & p = 3:p10) & Any q:(q++[2] = p)

Narrowing the preceding formula leads to

p = [1,3,2] |
Any p10:((p10 = 1:[2] | Any p11:(insert(1,[],p11) & p10 = 2:p11)) & p = 3:p10) &
Any q:(q++[2] = p)
```

```
Simplifying the preceding formula (13 steps) leads to

p = [1,3,2] | p = [3,1,2] |
Any p10:(Any p11:(insert(1,[],p11) & p10 = 2:p11) & p = 3:p10) &
Any q:(q++[2] = p)

Narrowing the preceding formula leads to

p = [1,3,2] | p = [3,1,2] |
Any p10:(Any p11:(p11 = 1:[] & p10 = 2:p11) & p = 3:p10) & Any q:(q++[2] = p)

Simplifying the preceding formula (17 steps) leads to

p = [1,3,2] | p = [3,1,2]

Number of proof steps: 24
```

$p = [1,3,2]$ und $p = [3,1,2]$ sind tatsächlich die beiden einzigen Lösungen des Goals aus Beispiel 4.2.3. 🐰

Resolution ist korrekt in folgendem Sinne:

Satz 4.2.14 Sei $SP = (\Sigma, AX)$ eine Spezifikation. Die Resolution mit AX ist **lösungskorrekt bzgl.** $Her(SP)$, d.h. für alle Σ-Goals G und die **identische Substitution** id ($dom(id) = \emptyset$) gilt:

$$\langle G, id\rangle \vdash_{SP} \langle True, \tau\rangle \quad \text{impliziert} \quad Her(SP) \models G\tau.$$

Beweis. Die Behauptung läßt sich in zwei Bedingungen zerlegen:

❶ Für jeden einzelnen Resolutionsschritt $\langle G, \tau\rangle \vdash_{SP} \langle G', \tau'\rangle$ gilt $Her(SP) \models G' \Rightarrow G\tau'$ und τ' ist eine *Spezialisierung* von τ, d.h. es gibt eine Substitution ρ mit $\tau\rho = \tau'$.

❷ Transitiver Abschluß: Sei $\langle G, \tau\rangle \vdash_P \langle G', \tau'\rangle \vdash_{SP} \langle G'', \tau''\rangle$, $Her(SP) \models G' \Rightarrow G\tau'$, τ' eine Spezialisierung von τ, $Her(SP) \models G'' \Rightarrow G'\tau''$ und τ'' eine Spezialisierung von τ'. Dann gilt $Her(SP) \models G'' \Rightarrow G\tau''$ und τ'' ist eine Spezialisierung von τ.

❶ Sei $\langle G\sigma \wedge H\sigma, \tau\sigma\rangle$ mit einem Resolutionsschritt aus $\langle G \wedge r(t), \tau\rangle$ ableitbar und b erfülle $G\sigma \wedge H\sigma$. Dann erfüllt b auch $r(t)\sigma = r(u)\sigma$, weil die angewendete Hornformel $r(u) \Leftarrow H$ in $Her(SP)$ gültig ist. Wegen $var(G) \cap dom(\tau) = \emptyset$ stimmt $G\sigma$ mit $G\tau\sigma$ überein. Also gilt $Her(SP) \models_b (G \wedge r(t))\tau\sigma$.

❷ Nach Vor. gibt es Substitutionen ρ, ρ' mit $\tau\rho = \tau'$ und $\tau'\rho' = \tau''$. Sei $Her(SP) \models_b G$. Nach Vor. folgt $Her(SP) \models_b G'\tau'' = G'\tau'\rho'$. Wegen $var(G') \cap dom(\tau') = \emptyset$ erhält man $Her(SP) \models_b G'\rho'$, also $Her(SP) \models_{b^* \circ \rho'} G'$. Nach Vor. folgt $Her(SP) \models_{b^* \circ \rho'} G\tau'$, also $Her(SP) \models_b G\tau'\rho' = G\tau''$. Damit ist $Her(SP) \models G \Rightarrow G\tau''$ gezeigt. Wegen $\tau\rho\rho' = \tau'\rho' = \tau''$ ist τ'' eine Spezialisierung von τ. ❑

Resolution ist auch **lösungsvollständig**: Für alle Grundsubstitutionen σ mit $Her(SP) \models G\sigma$ gibt es eine Substitution τ mit $\langle G, id\rangle \vdash_{SP} \langle True, \tau\rangle$ und τ läßt sich zu σ fortsetzen, d.h. es gibt ρ mit $\tau\rho = \sigma$.

Um die Resolutionsregel anzuwenden, muss man in AX nach einer Hornformel suchen, deren Kopf $r(u)$ mit einem Atom $r(t)$ des zu lösenden Goals **unifizierbar**, d.h. es gibt eine Substitution σ mit $t\sigma = u\sigma$. Voraussetzung dabei ist, dass t und u keine gemeinsamen Variablen haben. Das läßt sich immer sicherstellen, weil t und u aus verschiedenen, durch keine gemeinsamen Quantoren verknüpften Formeln stammen: t gehört zum Goal, u zu AX. Nach einem Resolutionsschritt gelangen evtl. Variablen von AX in das Goal. Diese müssen dann in Variablen umbenannt werden, die in weder in AX noch $dom(\tau\sigma)$ vorkommen, damit auch die Constraint-Bedingung wiederhergestellt ist (s. 4.2.12).

Der durch das folgende Haskell-Programm definierte **Unifikationsalgorithmus von Robinson** berechnet für zwei beliebige Terme t und u mit disjunkten Variablenmengen gleich den **allgemeinsten Unifikator** σ,

d.h. für jeden Unifikator τ gibt es eine Substitution ρ mit $\sigma\rho = \tau$.

```
unify :: Term -> Term -> Maybe (String -> Term)

unify (V x) (V y)       = if x == y then Just V else Just (V y `for` x)
unify (V x) t           = if x `isin` t then Nothing else Just (t `for` x)
unify t (V x)           = unify (V x) t
unify (F x ts) (F y us) = if x == y then unifyall ts us else Nothing

unifyall :: [Term] -> [Term] -> Maybe (String -> Term)

unifyall [] []         = Just V
unifyall (t:ts) (u:us) = do f <- unify t u
                            g <- unifyall (map (>>> f) ts) (map (>>> f) us)
                            Just (f `andThen` g)
unifyall _ _           = Nothing
```

`x `isin` t` stellt fest, ob x in t vorkommt. Wenn ja, sind x und t nicht unifizierbar. Auf diesen sog. **occurs check** kann man nicht verzichten, auch wenn `unify` am Anfang immer auf Terme mit disjunkten Variablenmengen angewendet wird. Selbst dann kann es nämlich vorkommen, dass spätere rekursive Aufrufe von `unify` auf Paare von Termen mit nicht-disjunkten Variablenmengen angewendet werden. Sollen z.B. $f(x, x)$ und $f(y, g(y))$ unifiziert werden, dann wird zunächst x durch y ersetzt, so dass ein rekursiver Aufruf von `unify` y und $g(y)$ zu unifizieren versucht, was beim *occurs check* erkannt und verhindert werden muss.

☞ Rekapitulieren Sie die Ableitung in Beispiel 4.2.3 als Folge von Anwendungen der Resolutionsregel und bekannter logischer Äquivalenzen!

5 Funktional-logische Spezifikation und Termersetzung

5.1 Konstruktorbasierte Spezifikationen

Die in logischen Programmen auftretenden Funktionssymbole werden in der Termalgebra T_Σ interpretiert. Je zwei syntaktisch verschiedene Σ-Terme sind also auch semantisch verschieden: $3 + 5$ ist nicht dasselbe wie 8. Funktionsaufrufe stellen nur sich selbst dar. Funktionen werden nicht ausgewertet. Einigen Funktionen, nämlich den **Konstruktoren**, aus denen Daten gebildet werden, ist diese Semantik durchaus angemessen. Z.B. beschreibt der Term $3 : 5 : 9 : []$ mit den Konstruktoren $:$ ("append"), $[]$, 3, 5 und 9 die Liste $[3, 5, 9]$. Jeder andere aus diesen Konstruktoren zusammengesetzte Term bezeichnet eine andere Liste. Ein impliziter Konstruktor ist z.B. die **n-Tupel-Bildung**

$$(\_, \ldots, \_) : s_1 \ldots s_n \to s_1 \times \cdots \times s_n.$$

$s_1 \times \cdots \times s_n$ bezeichnet hier eine Sorte, die standardmäßig als kartesisches Produkt der $s_1, \ldots, s_n$ zugeordneten Datenmengen interpretiert wird (s. auch §6.5).

Auf der anderen Seite nennt man z.B. $+$ eine **definierte Funktion**, weil sie durch eine Menge von Axiomen, ein funktionales Programm o.ä. definiert wird. Deshalb wird von jetzt an jedes Funktionssymbol entweder als Konstruktor oder als definierte Funktion deklariert. Konstruktoren werden weiterhin wie in der Termalgebra *frei* interpretiert. Die Bedeutung definierter Funktionen hingegen ergibt sich aus *Gleichungen* zwischen Termen, die die jeweiligen Funktionen enthalten.

Eine Hornformel der Form $t_1 \equiv t_2 \Leftarrow G$ heißt **(bedingte) Gleichung** mit **linker Seite** t_1 und **rechter Seite** t_2. Atome, die keine Gleichungen sind, nennt man **logische Atome**.[25]

[25] Wenn keine Verwechslung mit der Identität von Termen oder Formeln möglich ist, schreiben wir auch $=$ anstelle von $\equiv$.

Hier stellt sich die Frage, warum man logische Prädikate überhaupt noch zuläßt und diese nicht auf Boolesche Funktionen reduziert, so dass auch keine logischen Atome, sondern nur Gleichungen vorkommen. Wir tun das nicht, weil Prädikate ein allgemeineres Konzept als Boolesche Funktionen darstellen. Intuitiv gesprochen sind Prädikate *partiell* definierte Boolesche Funktionen: Man definiert ihren Gültigkeitsbereich (die "true"-Fälle), aber nicht dessen Komplement (die "false"-Fälle). Logik-Programmierung wird hauptsächlich im Datenbankbereich angewendet, wo ein Prädikat r bestimmte Beziehungen zwischen Daten beschreibt. Wer die Datenbank füttert, muss sagen, für welche Daten r gilt, würde sich aber i.a. schwertun, alle Daten aufzuzählen, für die r *nicht* gilt. Selbst wenn das gelänge, müßte diese *negative Information* bei jeder Änderung der Datenmenge überprüft und ggf. modifiziert werden, um die *Konsistenz* des Systems zu erhalten. Es wäre nicht *monoton*, d.h. neue Information kann die Gültigkeit alter Information beeinflussen.

Natürlich muss es möglich sein, negative *Anfragen* zu stellen und beantwortet zu bekommen. Dazu macht man die *closed world assumption*, was nichts anderes bedeutet als dass die Semantik des Systems als Herbrandmodell der logischen Programme definiert wird: Eine prädikatenlogische Formel ist genau dann gültig, wenn sie im Herbrandmodell gilt. Folglich ist ein negiertes Goal $\neg G$ genau dann gültig, wenn *keine* Instanz von G mit dem Schnittkalkül ableitbar ist, oder, wegen der Lösungsvollständigkeit der Resolution, keine Substitution τ mit $\langle G, id\rangle \vdash_{SP} \langle True, \tau\rangle$ existiert (vgl. §4.2).

Für die Verknüpfung von funktionalen mit logischen Konzepten und deren jeweiligen Anwendungsbereichen ist es demnach notwendig, das Konzept des Prädikats beizubehalten und nicht auf Gleichheiten einzuschränken. Ein Prädikat ist keine Boolesche Funktion. Diese werden als Funktionssymbole in die Sorte *bool* spezifiziert, von der es genau zwei Normalformen geben sollte, nämlich $true$ und $false$.

Partialität kann sich bei der Auswertung eines Funktionsaufrufs $f(a)$ auf zweierlei Weise zeigen:

❶ Das Programm für f kann an der Stelle a gar nicht ausgeführt werden, so dass die Auswertung von $f(a)$ sofort abbricht.

❷ Die Auswertung von $f(a)$ bricht überhaupt nicht ab, weil sie unendlich viele rekursive Aufrufe von f (oder einer von f verwendeten Funktion) nach sich zieht, was nicht passieren kann, wenn die Argumente der entstehenden Aufruffolge — bzgl. einer *wohlfundierten* Ordnung — von Aufruf zu Aufruf kleiner werden:

Definition 5.1.1 Eine binäre Relation R auf einer Menge A heißt **wohlfundiert**, wenn jede nichtleere Teilmenge von A ein bzgl. R minimales Element hat. ❑

Eine wohlfundierte Relation ist weder reflexiv noch symmetrisch.

❶ kann durchaus beabsichtigt sein: Division durch 0 wird i.a. dadurch verhindert, dass man 0 als Argument explizit ausschließt. Allgemein heißt das, man *kennt* die Argumente, für die f nicht definiert ist. Oftmals ist diese Kenntnis auch schon auf einer höheren Ebene des Programmentwurfs erforderlich, wo man vielleicht noch nicht festlegen will, *wie* fehlerhafte Funktionsaufrufe (*Ausnahmen*) behandelt werden sollen, aber zumindest sagen muss, *wo* sie auftreten können.

Es gibt mehrere Möglichkeiten, Partialität auf der Ebene abstrakter Syntax formal zu repräsentieren. In der CPO-Theorie werden Ausnahmewerte durch zusätzliche (minimale) Elemente eines CPO dargestellt. Dieser Ansatz ließe sich auch auf die hier zugrundegelegte Mengensemantik übertragen, hat aber den Nachteil, dass die Wirkung der zusätzlichen Elemente auf jede Funktion und jedes Prädikat der jeweiligen Signatur spezifiziert werden muss, was zu erheblichem Overhead bei der Formalisierung führt. Besser ist die Verwendung von Untersorten (s. §4.1). Anstelle einer partiellen Funktion $f : s'' \to s'$ wird eine totale Funktion $f : s'' \to s$ spezifiziert, wobei s eine "Obersorte" von s' ist, d.h. es gibt eine implizite Einbettungsfunktion $in : s' \to s$. Die Ausnahmewerte von f werden der Sorte s zugeordnet, womit sie von den "definierten", s' zugeordneten Daten getrennt bleiben.

Von nun an enthalte die Axiomenmenge einer Spezifikation implizit die Menge EQ_Σ der **Kongruenzaxiome für** Σ:

$x \equiv x$
$y \equiv x \quad \Leftarrow \quad x \equiv y$
$f(x_1,\ldots,x_n) \equiv f(y_1,\ldots,y_n) \quad \Leftarrow \quad x_1 \equiv y_1 \wedge \cdots \wedge x_n \equiv y_n$ für alle Funktionssymbole $f : s_1 \ldots s_n \to s$ von Σ
$r(x_1,\ldots,x_n) \quad \Leftarrow \quad r(y_1,\ldots,y_n) \wedge x_1 \equiv y_1 \wedge \cdots \wedge x_n \equiv y_n$ für alle Prädikate $r : s_1 \ldots s_n$ von Σ.

In Def. 4.1.2 haben wir gefordert, dass jede Σ-Struktur $A \equiv$ als prädikatenverträgliche Σ-Kongruenz interpretiert. Diese Forderung ist gleichbedeutend der Gültigkeit der Kongruenzaxiome für Σ.

☞ Zeigen Sie, dass $\equiv^A$ transitiv ist, wenn A die Kongruenzaxiome für Σ erfüllt!

Aus all dem ergibt sich folgendes Schema einer abstrakten Syntax für funktional-logische Programme und die von ihnen benutzten Datenstrukturen:

konstruktorbasierte Spezifikation

Definition 5.1.2 Sei $\Sigma = (S, F, R)$ eine Signatur derart, dass $F = CO_\Sigma \uplus DF_\Sigma$[26] aus einer Menge CO_Σ von **Konstruktoren** und einer Menge DF_Σ von **definierte Funktionen** besteht und R für alle $s \in S$ das Gleichheitsprädikat $\equiv: ss$ enthält.

Σ-Terme, die nur aus Konstruktoren und Variablen bestehen, heißen **Σ-Normalformen**. $NF_\Sigma(X)$ bzw. NF_Σ bezeichnet die Menge aller bzw. aller variablenfreien Σ-Normalformen.

Sei AX eine Menge von Σ-Hornformeln

$$f(t_1,\ldots,t_n) \equiv u \Leftarrow G \quad \text{oder} \quad r(t_1,\ldots,t_n) \Leftarrow G$$

mit folgenden Eigenschaften:

(1) $var(u) \subseteq var(t_1,\ldots,t_n,G)$,
(2) $f \in DF_\Sigma$, $t_1,\ldots,t_n \in NF_\Sigma(X)$.

Wir nennen die Hornformel ein **Axiom für** f bzw. r und $SP = (\Sigma, AX)$ eine **(konstruktorbasierte) Spezifikation**. Eine Σ-Struktur, die AX erfüllt, heißt **SP-Modell**.

Beispiel 5.1.3

NAT

`sorts`	nat
`constructs`	$0 :\to nat$
	$suc : nat \to nat$
`defuncts`	$1, 2, 3, \ldots :\to nat$
	$+, -, *, min, max : nat \times nat \to nat$
`preds`	$\leq, \geq, <, >: nat \times nat$
`vars`	$m, n : nat$
`axioms`	$1 \equiv suc(0)$
	$2 \equiv suc(1)$
	$3 \equiv suc(2)$
	$\ldots$
	$n + 0 \equiv n$

[26] Die **disjunkte Vereinigung** $\uplus$ setzt implizit voraus, dass die vereinigten Mengen keine gemeinsamen Elemente haben.

$m + suc(n) \equiv suc(m + n)$
$n - 0 \equiv n$
$0 - n \equiv 0$
$suc(m) - suc(n) \equiv m - n$
$n * 0 \equiv 0$
$m * suc(n) \equiv (m * n) + m$
$min(m, n) \equiv m \Leftarrow m \leq n$
$min(m, n) \equiv n \Leftarrow n \leq m$
$max(m, n) \equiv n \Leftarrow m \leq n$
$max(m, n) \equiv m \Leftarrow n \leq m$
$0 \leq n$
$suc(m) \leq suc(n) \Leftarrow m \leq n$
$m \geq n \Leftarrow n \leq m$
$0 < suc(n)$
$suc(m) < suc(n) \Leftarrow m < n$
$m > n \Leftarrow n < m$ ✌

Beispiel 5.1.4

STACK = NAT `and`

`sorts`	$stack$
`constructs`	$() :\rightarrow 1 + nat$
	$just : nat \rightarrow 1 + nat$
	$empty :\rightarrow stack$
	$push : nat \times stack \rightarrow stack$
`defuncts`	$top : stack \rightarrow 1 + nat$
	$pop : stack \rightarrow stack$
`vars`	$x : nat \quad s : stack$
`Horn axioms`	$top(empty) \equiv ()$
	$pop(empty) \equiv empty$
	$top(push(x, s)) \equiv just(x)$
	$pop(push(x, s)) \equiv s$ ✌

Beispiel 5.1.5

DIVREP = NAT `then`

`sorts`	$tree$
`constructs`	$() :\rightarrow 1 + (nat \times nat)$
	$(\_, \_) : nat \times nat \rightarrow 1 + (nat \times nat)$
	$leaf : nat \rightarrow tree$
	$\_\#\_ : tree \times tree \rightarrow tree$
`defuncts`	$div : nat \times nat \rightarrow 1 + (nat \times nat)$
	$rep\&min : tree \times nat \rightarrow tree \times nat$
	$repByMin : tree \rightarrow tree$
`vars`	$m, n, n', q, r : nat \quad T, T', U, U' : tree$
`Horn axioms`	$div(m, n) \equiv (0, m) \Leftarrow m < n$
	$div(m, n) \equiv (suc(q), r) \Leftarrow 0 < n \wedge n \leq m \wedge div(m - n, n) \equiv (q, r)$
	$div(m, 0) \equiv ()$
	$rep\&min(leaf(n), m) \equiv (leaf(m), n)$

$$\begin{aligned} rep\&min(T\#T', m) &\equiv (U\#U', min(n, n')) \\ &\Leftarrow\ rep\&min(T, m) \equiv (U, n) \wedge rep\&min(T', m) \equiv (U', n') \\ repByMin(T) \equiv T' &\ \Leftarrow\ rep\&min(T, m) \equiv (T', m) \end{aligned}$$

$div(m, n)$ berechnet den ganzzahligen Quotienten von m und n und gleichzeitig den verbleibenden Rest.

$rep\&min(T, m)$ liefert den minimalen Eintrag von T sowie einen neuen Baum, der aus T entsteht, wenn man alle Einträge durch m ersetzt. $repByMin(T)$ liefert den Baum, der aus T entsteht, wenn man alle Einträge durch den minimalen Eintrag von T ersetzt. ✌

Tupelkonstruktoren wie der in DIVREP verwendete Paarkonstruktor $(\_, \_) : tree \times nat \to tree \times nat$ werden nicht explizit deklariert.

Da $\vdash_{cut}$ von den Axiomen von SP abhängt, schreiben wir in Zukunft

$$SP \vdash_{cut} G \qquad \text{anstelle von} \qquad True \vdash_{cut} G.$$

Da alle Formeln von AX und EQ_Σ (s.o.) Hornformeln sind, ist das Herbrandmodell von $(\Sigma, AX \cup EQ_\Sigma)$ (siehe 4.2.6), kurz: **Her(SP)**, nach Satz 4.2.8 ein SP-Modell. Die Kongruenzaxiome für Σ implizieren, dass *jedes* SP-Modell Gleichheitsprädikate als Kongruenzrelationen interpretiert. Die Interpretation im Herbrandmodell nennen wir **SP-Äquivalenz** und bezeichnen sie mit $\equiv_{SP}$. Nach Definition von $Her(SP)$ gilt für alle $t, t' \in T_\Sigma(X)$,

$$t \equiv_{SP} t' \iff SP \vdash_{cut} t \equiv t'.$$

Enthält SP keine Axiome für definierte Funktionen, dann sind alle SP-äquivalenten Terme gleich. Das lässt sich einfach durch Induktion über die Länge einer kürzesten Schnittkalkül-Ableitung von $t \equiv t'$ zeigen. O.B.d.A. kann vorausgesetzt werden, dass die Instanziierungsregel nur auf Axiome angewendet wird. Dann kann $t \equiv t'$ nur durch Anwendung des Modus ponens auf ein Kongruenzaxiom für Σ entstanden sein, d.h. es liegt einer der folgenden vier Fälle vor:

(1) Anwendung des Reflexivitätsaxioms. Dann ist $t = t'$ und wir sind fertig.

(2) Anwendung des Symmetrieaxioms. Dann ist $t' \equiv t$ ein Vorgänger in der Ableitung von $t \equiv t'$. Also gilt $t' = t$ nach Induktionsvoraussetzung.

(3) Anwendung des Transitivitätsaxioms. Dann gibt es $u \in T_\Sigma$ so, dass $t \equiv u$ und $u \equiv t'$ ein Vorgänger in der Ableitung von $t \equiv t'$. Also gelten $t = u$ und $u = t'$ nach Induktionsvoraussetzung und damit auch $t = t'$.

(4) Anwendung eines Kompatibilitätsaxioms. Dann gibt es ein Funktionssymbol f und Terme $t_1, \ldots, t_n, t'_1, \ldots, t'_n$ mit $t = f(t_1, \ldots, t_n)$ und $t' = f(t'_1, \ldots, t'_n)$ so, dass für alle $1 \leq i \leq n$ $t_i \equiv t'_i$ ein Vorgänger in der Ableitung von $t \equiv t'$ ist. Also gilt $t_i = t'_i$ nach Induktionsvoraussetzung und damit auch $t = f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n) = t'$.

Die Daten von $Her(SP)$ sind Terme. Die Gleichheit in $Her(SP)$ kann also nicht mit der SP-Äquivalenz übereinstimmen. Das gilt erst für den Quotienten von $Her(SP)$ nach $\equiv_{SP}$ (siehe Def. 4.1.2):

initiales Modell

Definition 5.1.6 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation. Die Interpretation der Gleichheiten für Σ im Herbrandmodell von SP heißt **SP-Äquivalenz** und wird mit $\equiv_{SP}$ bezeichnet. Der Quotient

$$Ini(SP) \quad =_{def} \quad Her(SP)/\equiv_{SP}$$

heißt **initiales Modell von** SP.

Lemma 5.1.7 Für alle prädikatenlogischen Formeln φ gilt:

$$Ini(SP) \models \varphi \quad \Longleftrightarrow \quad Her(SP) \models \varphi.$$

Beweis. ☞ Übung. ❑

Nach Satz 4.2.8❶ ist $Her(SP)$ ein SP-Modell. Damit folgt aus Lemma 5.1.7, dass auch $Ini(SP)$ ein SP-Modell ist. Darüberhinaus ist $Ini(SP)$ eine Struktur mit Gleichheit (s. 4.1.2), da für alle $t, t' \in T_\Sigma$

$$[t] \equiv^{Ini(SP)} [t'] \quad \Longleftrightarrow_{def} \quad t \equiv^{Her(SP)} t' \quad \Longleftrightarrow_{def} \quad SP \vdash_{cut} t \equiv t' \quad \Longleftrightarrow \quad t \equiv_{SP} t' \quad \Longleftrightarrow \quad [t] = [t']$$

gilt.

☞ Zeigen Sie, dass Satz 4.2.8❸ auch gilt, wenn man dort $Her(SP)$ durch $Ini(SP)$ und die Klasse der SP-Modelle durch die Klasse der SP-Modelle mit Gleichheit ersetzt!

Die Unterscheidung zwischen Konstruktoren und definierten Funktionen einer konstruktorbasierten Spezifikation $SP = (\Sigma, AX)$ legt die Forderung nahe, dass jeder Σ-Grundterm genau eine SP-äquivalente Normalform hat. Diese Eigenschaft wird in zwei Bedingungen zerlegt:

Vollständigkeit, Konsistenz

Definition 5.1.8 Eine konstruktorbasierte Spezifikation $SP = (\Sigma, AX)$ ist **vollständig**, wenn für alle $t \in T_\Sigma$ ein $u \in NF_\Sigma$ mit $t \equiv_{SP} u$ existiert. SP ist **konsistent**, wenn je zwei SP-äquivalente Grundnormalformen von SP identisch sind. SP ist **funktional**, wenn SP vollständig und konsistent ist. Ist SP funktional, dann bezeichnen wir die eindeutige Normalform von $t \in T_\Sigma$ mit **nf(t)**.

Eine konstruktorbasierte Spezifikation ohne definierte Funktionen ist immer funktional.

☞ Zeigen Sie, dass die Menge NF_Σ der Grundnormalformen zu einer zu $Ini(SP)$ isomorphen Σ-Struktur erweitert werden kann, sofern SP funktional ist! Diese Darstellung von $Ini(SP)$ ist anschaulicher als die oben definierte, weil sie wie T_Σ eine Struktur ist, deren Datenbereiche aus Termen und nicht aus Äquivalenzklassen bestehen.

Beweisen Sie: Ist SP vollständig, dann gilt für alle prädikatenlogischen Formeln φ:

$$Her(SP) \models \varphi \quad \Longleftrightarrow \quad \forall \sigma : X \to NF_\Sigma : Her(SP) \models \varphi\sigma.$$

Vollständigkeit zu zeigen ist i.a. eine leichte Aufgabe. Es genügt, für jeden **innermost** Σ-Grundterm $f(t)$ (d.h. f ist definierte Funktion und t Normalform) eine äquivalente Normalform anzugeben. Man braucht eine wohlfundierte Ordnung auf den Argumenten von f und zeigt $f(t) \equiv_{SP} u$ durch Induktion über t entlang dieser Ordnung. Ist DIVREP aus Bsp. 5.1.5 vollständig?

Das Problem der Konsistenz ist schwieriger zu lösen. An dieser Stelle kommt man nur mit zusätzlichen Begriffen aus der **Theorie der Termersetzung (rewriting theory)** weiter. Die Frage ist, ob jede SP-Äquivalenz auf eine Sequenz *orientierter* Ersetzung von Teiltermen zurückgeführt werden kann: Gilt $t \equiv_{SP} t'$ *genau dann*, wenn t und t' durch links-rechts-Anwendungen von Gleichungsaxiomen in denselben Term überführbar sind? Nehmen wir das an, dann folgt sofort, dass SP konsistent ist: Derart ineinander überführbare Normalformen sind identisch, weil es keine Axiome gibt, die man auf sie anwenden kann (s. Def. 5.1.2).

Da SP i.a. nicht nur Funktionen, sondern auch Relationen enthält und die Axiome von SP bedingt sein können, verallgemeinern wir Termersetzung zur Goalersetzung, die über folgenden Kalkül definiert wird: Für

Reduktionskalkül

Definition 5.1.9 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation. Der **Reduktionskalkül für** SP besteht aus drei Regeln zur Ableitung von Σ-Goals. Sei $\sigma : X \to T_\Sigma(X)$ und $(var(H) \setminus var(t))\sigma \subseteq NF_\Sigma(X)$.

Resolution $\dfrac{G \wedge r(t\sigma)}{G \wedge H\sigma} \Uparrow$ falls $r(t) \Leftarrow H \in AX$ und r kein Gleichheitsprädikat ist

Rewriting $\dfrac{G(t\sigma)}{G(u\sigma) \wedge H\sigma} \Uparrow$ falls $t \equiv u \Leftarrow H \in AX$

Reflexion $\dfrac{G \wedge t \equiv t}{G} \Uparrow$

$\vdash_{SP}$ bezeichnet die Inferenzrelation des Reduktionskalküls für SP. Eine Ableitung $G_1, \ldots, G_n$ mit dem Reduktionskalkül für SP heißt SP**-Reduktion von** G_1. Ist $G_n = True$, dann ist die Reduktion **erfolgreich** (*successful reduction*). Ist auf $G_n \neq True$ keine der drei Regeln anwendbar, dann **scheitert** die Reduktion (*failing reduction*). Gibt es eine erfolgreiche SP-Reduktion von G, gilt also $G \vdash_{SP} True$, dann nennen wir G **erfolgreich** SP**-reduzierbar**.

☞ Zeigen Sie, dass der Reduktionskalkül bzgl. jedes Modells von SP korrekt ist!

Die Resolutionsregel von 4.2.14 wird zur Resolution des Reduktionskalküls, wenn σ keine echten Terme substituiert, sondern Variablen nur umbenennt. Die Regel beschreibt die Anwendung eines Relationsaxioms. Analog beschreibt die Rewriting-Regel die (links-rechts-)Anwendung eines Funktionsaxioms $t \equiv u \Leftarrow H$ als Ersetzung der im Goal $G(t\sigma)$ vorkommenden Instanz $t\sigma$ der linken Seite des Axioms durch die entsprechende Instanz $u\sigma$ der rechten Seite des Axioms. Das Ziel dieses Kalküls ist es, Gleichungen zu eliminieren, was, wie die Reflexionsregel zeigt, genau dann gelingt, wenn beide Seiten der Gleichungen in denselben Term transformierbar sind.

Die Variablen von H, die nicht in t vorkommen, heißen **frische Variablen** von $r(t) \Leftarrow H \in AX$ bzw. $t \equiv u \Leftarrow H \in AX$, weil ihre Instanzen bei der Anwendung der Resolutions- bzw. Rewriting-Regel erst erzeugt werden. Die Bedingung $(var(H) \setminus var(t))\sigma \subseteq NF_\Sigma(X)$ besagt, dass sie nur durch Normalformen ersetzt werden dürfen. An den Regeln erkennt man sofort, dass ein Goal genau dann erfolgreich SP-reduzierbar ist, wenn alle seine Atome erfolgreich SP-reduzierbar sind. Der **Reduktionskalkül ist korrekt bzgl. des Schnittkalküls**, d.h. für alle Grundgoals G gilt:

$$G \vdash_{SP} True \quad \Longrightarrow \quad SP \vdash_{cut} G.$$

Bei diesem Korrektheitsbegriff wird ein Kalkül nicht mit einer Semantik, sondern mit einem anderen Kalkül in Beziehung gesetzt. Im vorliegenden Fall läßt sich allerdings das eine auf das andere zurückführen. Wie?

Aus dem Reduktions*kalkül* ergibt sich die Reduktions*relation*:

Reduktionsrelation, Konfluenz

Definition 5.1.10 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation. Die SP**-Reduktionsrelation** $\longrightarrow_{SP}$ ist eine binäre Relation auf **Termen und Goals** und folgendermaßen definiert. Sei $G(x)$ ein Σ-Term oder -Goal und $\sigma : X \to T_\Sigma(X)$.

$$G(t\sigma) \longrightarrow_{SP} G(u\sigma) \iff_{def} \exists\, t \equiv u \Leftarrow H \in AX : H\sigma \vdash_{SP} True \wedge (var(H) \setminus var(t))\sigma \subseteq NF_\Sigma(X).$$

t' ist ein SP**-Redukt** von t, wenn $t \longrightarrow^*_{SP} t'$ gilt. Ist t das einzige Redukt von t, dann heißt t SP**-reduziert**. Eine Substitution σ' ist ein Redukt einer Substitution σ, wenn $x\sigma \longrightarrow^*_{SP} x\sigma'$ für alle $x \in X$ gilt. Wir schreiben dann: $\sigma \longrightarrow^*_{SP} \sigma'$.

SP ist **terminierend**, wenn $\longrightarrow^{-1}_{SP}$ wohlfundiert ist (siehe Def. 5.1.1) und alle SP-reduzierten Terme Normalformen sind.

SP ist **konfluent**, wenn für alle $t \in T_\Sigma$ je zwei SP-Redukte von t ein gemeinsames SP-Redukt haben.

Ein Goal G ist SP**-konvergent**, wenn alle SP-Redukte von G erfolgreich SP-reduzierbar sind.

☞ Zeigen Sie, dass $\longrightarrow_{SP}$ bzgl. des Schnittkalküls korrekt ist, d.h. für alle Grundgoals G gilt:

$$t \xrightarrow{*}_{SP} t' \quad \Rightarrow \quad SP \vdash_{cut} t \equiv t'.$$

☞ Zeigen Sie, dass jede terminierende Spezifikation vollständig ist! (*)

Konfluenz bedeutet intuitiv: Gibt es überhaupt eine erfolgreiche Reduktion eines Grundgoals G, dann läßt sich *jede Reduktion* von G zu einer solchen fortsetzen. M.a.W.: Alle erfolgreichen Reduktionen "fließen zusammen". Auf Terme bezogen heißt das gerade, dass zwei Redukte desselben Terms selbst wieder ein gemeinsames Redukt haben.

☞ Zeigen Sie folgende Behauptungen:

(1) Für alle Grundgoals G, G' gilt:

$$\begin{array}{lll} G \longrightarrow_{SP} G' \wedge G' \vdash_{SP} \mathit{True} & \Rightarrow & G \vdash_{SP} \mathit{True}, \\ SP \vdash_{cut} G \wedge G \longrightarrow_{SP} G' & \Rightarrow & SP \vdash_{cut} G'. \end{array}$$

(2) Zwei Σ-Terme t und t' haben genau dann ein gemeinsames SP-Redukt, wenn die Gleichung $t \equiv t'$ erfolgreich SP-reduzierbar ist.

(3) Alle Normalformen einer konstruktorbasierten Spezifikation SP sind SP-reduziert.

Satz 5.1.11 SP ist genau dann konfluent, wenn alle erfolgreich SP-reduzierbaren Grundgoals SP-konvergent sind.

Beweis. Die Behauptung folgt aus 5.1.2(2) sowie (2), (3) und folgendem Lemma:

Lemma 5.1.12 SP ist genau dann konfluent, wenn für alle Σ-Normalformen t, Grundsubstitutionen σ und Σ-Terme u gilt:

$$t\sigma \xrightarrow{*}_{SP} u \quad \Rightarrow \quad \exists\, \tau : X \to T_\Sigma : (u \xrightarrow{*}_{SP} t\tau \wedge \sigma \xrightarrow{*}_{SP} \tau).$$

Konfluenz bedeutet auch, dass alle mit dem Schnittkalkül aus SP ableitbaren Grundgoals erfolgreich SP-reduzierbar sind. M.a.w.: Der Reduktionskalkül ist **vollständig** bzgl. des Schnittkalküls:

Satz 5.1.13 (Church-Rosser-Theorem) Eine vollständige Spezifikation SP ist genau dann konfluent, wenn für alle Grundgoals G gilt:

$$SP \vdash_{cut} G \quad \Longrightarrow \quad G \vdash_{SP} \mathit{True}.$$

Diese Implikation besagt, dass jeder *Vorwärtsbeweis* (eines *Grund*goals) mit dem Schnittkalkül einem *Rückwärtsbeweis* mit dem Reduktionskalkül entspricht. Die im Schnittbeweis angewendeten Kongruenzaxiome gehen im Reduktionsbeweis in Rewriting- und Reflexionsschritte ein. Satz 5.1.13 verwendet wesentlich das folgende Lemma:

Lemma 5.1.14 Sei SP vollständig und konfluent. Für alle Kongruenzaxiome und Axiome $C = (p \Leftarrow H)$ von SP und alle Grundsubstitutionen σ gilt:

$$H\sigma \vdash_{SP} \mathit{True} \quad \Longrightarrow \quad p\sigma \vdash_{SP} \mathit{True}.$$

Beweis. ☞ Übung. ❑

Beweis von Satz 5.1.13. "⇐": Wegen Satz 5.1.11 genügt es zu zeigen, dass alle erfolgreich SP-reduzierbaren Grundgoals SP-konvergent sind. Sei $G \vdash_{SP} \mathit{True}$ und $G \xrightarrow{*}_{SP} G'$. Die Korrektheit des Reduktionskalküls bzgl. des Schnittkalküls liefert $SP \vdash_{cut} G$, also wegen (1) auch $SP \vdash_{cut} G'$. Nach Voraussetzung folgt $G' \vdash_{SP} \mathit{True}$.

Der ⇒-Teil läßt sich unter Verwendung von Lemma 5.1.14 leicht durch Induktion über die Länge einer kürzesten Schnittkalkülableitung von G zeigen. ❑

Die beiden Voraussetzungen von Lemma 5.1.14 liefern auch Konsistenz:

Korollar 5.1.15 Eine vollständige und konfluente Spezifikation ist konsistent.

Beweis. Die Behauptung folgt aus Satz 5.1.13 sowie (2) und (3). ❑

Terminierende Spezifikationen erfüllen auch die Umkehrung:

Satz 5.1.16 Eine terminierende Spezifikation ist genau dann konfluent, wenn sie konsistent ist.

Beweis. "⇒": Korollar 5.1.15.

"⇐": Sei SP terminierend und konsistent und seien u und u' zwei SP-Redukte eines Grundterms t. Dann gibt es Normalformen v und v' von u bzw. u' mit $u \xrightarrow{*}_{SP} v$ und $u' \xrightarrow{*}_{SP} v'$. Aus der Korrektheit der Reduktionsrelation $\longrightarrow_{SP}$ folgt, dass t, u, v sowie t, u', v' jeweils untereinander SP-äquivalent sind. Also sind auch v und v' SP-äquivalent sind. Da SP konsistent ist, stimmen beide Normalformen überein, bilden also ein *gemeinsames* Redukt von u und u'. ❑

Konfluenz und Termination sind hinreichend für die Entscheidbarkeit der Gültigkeit von Grundgoals im Herbrandmodell:

Satz 5.1.17 Sei $SP = (\Sigma, AX)$ eine terminierende und konfluente Spezifikation und G ein Σ-Grundgoal. Dann gibt es einen Algorithmus, der entscheidet, ob G in $Her(SP)$ (oder $Ini(SP)$) gilt.

Beweis. Wir zeigen die Behauptung hier nur für Gleichungen. Dass sie auch für beliebige Grundgoals gilt, folgt dann aus Satz 5.1.11. Sei $t \equiv t'$ eine Gleichung ohne Variablen. Da SP terminierend ist, lassen sich in endlicher Zeit zwei Normalformen u und u' mit $t \xrightarrow{*}_{SP} u$ und $t' \xrightarrow{*}_{SP} u'$ konstruieren. Die Entscheidung, ob $t \equiv t'$ in $Her(SP)$ gilt, reduziert sich auf die Frage, ob u und u' gleich sind. Mit anderen Worten, es bleibt die Äquivalenz

$$Her(SP) \models t \equiv t' \quad \Longleftrightarrow \quad u = u'$$

zu zeigen.

$t \equiv t'$ gelte in $Her(SP)$. Nach Definition von $\equiv^{Her(SP)}$ ist das gleichbedeutend mit $SP \vdash_{cut} t \equiv t'$. Da SP terminierend ist, ist SP vollständig, so dass nach Satz 5.1.13 $t \equiv t'$ erfolgreich SP-reduzierbar ist. Wegen (2) haben t und t' ein gemeinsames SP-Redukt v. Da SP konfluent ist und u und u' Normalformen, also nach (3) SP-reduziert sind, sind u und u' zwei SP-Redukte von v. Also haben u und u' wegen der Konfluenz von SP ein gemeinsames SP-Redukt, das mit u und u' übereinstimmt, weil u und u' SP-reduziert sind.

Sei $u = u'$. Dann folgen $SP \vdash_{cut} t \equiv u$ und $SP \vdash_{cut} t' \equiv u'$, also auch $SP \vdash_{cut} t \equiv t'$ aus der Korrektheit von $\longrightarrow_{SP}$ bzgl. des Schnittkalküls. Somit ist $t \equiv t'$ nach Definition von $\equiv^{Her(SP)}$ in $Her(SP)$ gültig. ❑

(*), das Church-Rosser-Theorem und Satz 5.1.16 liefern schließlich – zumindest für terminierende Spezifikationen SP – eine deduktive Charakterisierung der Funktionalität von SP:

Korollar 5.1.18 Eine terminierende Spezifikation SP ist genau dann funktional, wenn für alle Grundgoals G gilt:

$$SP \vdash_{cut} G \quad \Longrightarrow \quad G \vdash_{SP} \mathit{True}.$$

Mithilfe von Korollar 5.1.18 kann man u.a. die Charakterisierung von Herbrandmodellen als kleinste Modelle logischer Programme (Satz 4.2.8) auf initiale Modelle konstruktorbasierter Spezifikationen übertragen. Dazu

werden definierte Funktionen durch ihre Graphen ersetzt:

flattening

Definition 5.1.19 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation. SP ist **relational**, wenn SP keine definierten Funktionen enthält. SP ist **flach**, wenn in allen Axiomen φ von SP definierte Funktionen nur als führende Symbole von Gleichungen auftreten, d.h. kommt $f \in DF_\Sigma$ in φ vor, dann nur innerhalb von Gleichungen der Form $f(t) \equiv u$ mit $t, u \in NF_\Sigma(X)$. Wir bezeichnen eine solche Gleichung als **flach**. Weitere flache Atome sind von der Form $r(t)$, wobei t eine Normalform ist. Eine Formel ist **flach**, wenn sie nur flache Atome enthält.

Jede Hornformel φ kann mit folgenden Regeln (und ihren symmetrischen Varianten) in eine flache Hornformel **flat(φ)** transformiert werden: Sei x eine Variable, die im jeweiligen Regel-Antezedenten nicht vorkommt.

- ★ $\dfrac{p[t/x] \Leftarrow \varphi}{p \Leftarrow t \equiv x \wedge \varphi}$ ⇳
- ★ $\dfrac{p \Leftarrow \varphi[t/x] \wedge \psi}{p \Leftarrow \varphi \wedge t \equiv x \wedge \psi}$ ⇳

Sei $f : w \to s \in DF_\Sigma$. Das Prädikat $r_f : ws$ heißt **Graph** oder **Ein/Ausgabe-Relation** von f. Sei $\mathbf{rel}(\Sigma) = \Sigma \setminus DF_\Sigma \cup \{r_f \mid f \in DF_\Sigma\}$.

Eine flache Σ-Hornformel φ wird zur $rel(\Sigma)$-Hornformel **rel(φ)**, indem man für alle $f \in DF_\Sigma$ jede (flache) Gleichung $f(t) \equiv u$ oder $u \equiv f(t)$ von φ durch das Atom $r_f(t, u)$ ersetzt.

Sei $flat(AX) = \{flat(\varphi) \mid \varphi \in AX\}$ und $rel(AX) = \{rel(\varphi) \mid \varphi \in AX\}$. Die Spezifikationen **flat(SP)** $= (\Sigma, flat(AX))$ und **rel(SP)** $= (rel(\Sigma), rel(flat(AX)))$ heißen **flache** bzw. **relationale Version von** SP.

Bzgl. welcher Σ-Strukturen sind die ersten beiden Regeln korrekt?

Der Übergang von φ nach $flat(\varphi)$ wird *flattening* genannt, weil dabei jeder zusammengesetzte Aufruf von n definierten Funktionen in n Gleichungen zerlegt ("flachgeklopft") wird. Aus $f(g(h(x))) \equiv y$ wird z.B. $f(x_1) \equiv y \Leftarrow h(x) \equiv x_2 \wedge g(x_2) \equiv x_1$ mit den beiden o.g. Regeln und dann $r_f(x_1, y) \Leftarrow r_h(x, x_2) \wedge r_g(x_2, x_1)$. Auf diese Weise lassen sich funktionale Programme in rein logische Programme übersetzen. Die obigen Regeln sind ein Beispiel für einen Kalkül zur Transformation von Formeln, bei der es nicht um deren Beweis, sondern ihre Übersetzung in eine andere Syntax geht, wobei die Bedeutung erhalten bleibt. So gilt für alle Hornformeln φ:[27]

$$Her(SP) \quad \models \quad \varphi \Leftrightarrow flat(\varphi) \tag{1}$$

und

$$Her(SP) \models \varphi \quad \Longleftrightarrow \quad Her(flat(SP)) \models \varphi. \tag{2}$$

(2) besagt, dass SP und $flat(SP)$ induktiv, d.h. bzgl. ihrer Herbrandmodelle, äquivalent sind (s. Def. 6.1.3). Ist eine von zwei induktiv äquivalenten Spezifikationen funktional, dann ist es auch die andere. $rel(SP)$ ist immer funktional. Warum?

Man beachte, dass beim Übergang von SP zu $rel(SP)$ die Signatur verändert wird. Aus Σ wird $rel(\Sigma)$. Damit verschwinden alle Gleichungen aus SP und wir müssen voraussetzen, dass SP terminierend und konsistent ist, um mithilfe von Korollar 5.1.18 die induktive Äquivalenz von SP und $rel(SP)$ — *modulo der Signaturänderung* — ableiten zu können:

[27] Andere prädikatenlogische Formeln lassen sich auch "flachklopfen". ☞ Geben Sie Transformationsregeln an, die das tun!

Satz 5.1.20 (Äquivalenz einer funktionalen Spezifikation und ihrer relationalen Version) Sei SP terminierend und konsistent. Dann gilt für alle flachen Σ-Atome p:

$$Her(SP) \models p \quad \Longleftrightarrow \quad Her(rel(SP)) \models rel(p), \tag{3}$$

also für alle Σ-Formeln φ:

$$Her(SP) \models \varphi \quad \Longleftrightarrow \quad Her(rel(SP)) \models rel(flat(\varphi)).$$

Beweis. Da SP terminierend und konsistent ist, gilt das auch für $flat(SP)$, so dass (3) nach Korollar 5.1.18 äquivalent ist zur Konjunktion dreier Bedingungen: Für alle $f \in DF_\Sigma$, Prädikate $r \in \Sigma$ und $t, u \in NF_\Sigma$ gilt:

$$f(t) \equiv u \vdash_{flat(SP)} True \quad \Longleftrightarrow \quad True \vdash_{cut} r_f(t,u), \tag{4}$$
$$t \equiv u \vdash_{flat(SP)} True \quad \Longleftrightarrow \quad True \vdash_{cut} t \equiv u, \tag{5}$$
$$r(t) \vdash_{flat(SP)} True \quad \Longleftrightarrow \quad True \vdash_{cut} r(t), \tag{6}$$

oder, zusammengefaßt: Für alle flachen Σ-Atome p gilt:

$$p \vdash_{flat(SP)} True \quad \Longleftrightarrow \quad True \vdash_{cut} rel(p). \tag{7}$$

(7) zeigt man durch Induktion über die Länge kürzester SP- bzw. $rel(SP)$-Reduktionen. Wir tun das hier exemplarisch für die $\Rightarrow$-Richtungen von (4) und (5).

(4): Sei $f(t) \equiv u \vdash_{flat(SP)} True$. Wegen $f \in DF_\Sigma$ und $t, u \in NF_\Sigma$ sind die Terme $f(t)$ und u verschieden. Deshalb kann der erste Schritt $f(t) \equiv u \vdash G$ einer kürzesten Reduktion von $f(t) \equiv u$ nur eine Anwendung der Rewriting-Regel sein. Demnach gibt es ein flaches Axiom $\varphi = (f(l) \equiv r \Leftarrow H)$ und eine Substitution $\sigma : X \to NF_\Sigma$ mit $l\sigma = t$ und $(r\sigma \equiv u \wedge H\sigma) = G$. Insbesondere ist r eine Normalform und damit $r\sigma \equiv u$ eine Gleichung zwischen Normalformen. Also sind auch H und G flach. Nach Induktionsvoraussetzung sind $rel(G) = (r\sigma \equiv u \wedge rel(H\sigma))$, also insbesondere $rel(H\sigma) = rel(H)\sigma$ mit dem Schnittkalkül für $rel(SP)$ aus $True$ ableitbar. Mit der (auf das Axiom $rel(\varphi) = (r_f(l,r) \Leftarrow rel(H))$ angewendeten) Instanziierungsregel und dem Modus ponens lässt sich also auch $r_f(l,r)\sigma = r_f(t,u)$ aus $True$ herleiten.

(5): Sei $t \equiv u \vdash_{flat(SP)} True$. Wegen $t, u \in NF_\Sigma$ kann der erste Schritt $t \equiv u \vdash G$ einer kürzesten Reduktion von $t \equiv u$ nur eine Anwendung der Reflexionsregel sein, d.h. t und u sind identisch. Daraus folgt sofort $True \vdash_{cut} t \equiv u$. ❑

Aus 4.2.8❷ folgt, dass Fixpunktinduktion (siehe §4.2) auch auf definierte Funktionen angewendet werden kann. Sei $f \in DF_\Sigma$ und $AX_f = \{\varphi \in AX \mid f \text{ kommt in } \varphi \text{ vor}\}$.

Fixpunktinduktion über f

$$\frac{f(x) \equiv y \;\Rightarrow\; \psi(x,y)}{\bigwedge_{\varphi \in flat(AX_f)} \varphi[\psi(u,v)/f(u) \equiv v \mid f(u) \equiv v \text{ kommt in } \varphi \text{ vor}]} \Uparrow$$

☞ Zeigen Sie, dass die Regel bzgl. $Her(SP)$ korrekt ist!

Die Antezedent der Regel besagt, dass zwischen der "Eingabe" x und der "Ausgabe" y der Funktion f die Relation ψ besteht. Die Korrektheit der Fixpunktregel besagt, dass dies tatsächlich so ist, wenn ψ die Axiome für f löst, d.h. die Axiome für f gelten in $Her(SP)$, auch wenn man alle Vorkommen einer Gleichung $f(t) \equiv u$ (bzw. des entsprechenden Atoms $r_f(t,u)$ der logischen Version von SP) durch $\psi(t,u)$ ersetzt.

Funktionalität muss auch vorausgesetzt werden, damit die Resolutionsregel von Def. 4.2.12 zu einem lösungsvollständigen Kalkül für konstruktorbasierte Spezifikationen erweitert werden kann:

Narrowing-Kalkül

Definition 5.1.21 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation. Der **Narrowing-Kalkül für** SP besteht aus drei Regeln zur Ableitung von Constraints (s. 4.2.12). Sei $\sigma : X \to T_\Sigma(X)$ und $(var(H) \setminus var(t))\sigma \subseteq NF_\Sigma(X)$.

Resolution $\dfrac{\langle G \wedge r(t), \tau\rangle}{\langle G\sigma \wedge H\sigma, \tau\sigma\rangle} \Uparrow$ falls $t\sigma = u\sigma$, $r(u) \Leftarrow H \in AX$, r kein Gleichheitsprädikat ist und der Sukzedent wieder ein Constraint ist

Narrowing $\dfrac{\langle G(t), \tau\rangle}{\langle G(v)\sigma \wedge H\sigma, \tau\sigma\rangle} \Uparrow$ falls $t \notin X$, $t\sigma = u\sigma$, $u \equiv v \Leftarrow H \in AX$ und der Sukzedent wieder ein Constraint ist

Unifikation $\dfrac{\langle G \wedge t \equiv u, \tau\rangle}{\langle G\sigma, \tau\sigma\rangle} \Uparrow$ falls $t\sigma = u\sigma$ und der Sukzedent wieder ein Constraint ist

$\vdash_{SP}$ bezeichnet die Inferenzrelation des Narrowing-Kalküls für SP.

Wir verwenden dieselbe Bezeichnung $\vdash_{SP}$ für die Inferenzrelationen des Narrowing- und des Reduktionskalküls (s. 5.1.9), weil letzterer ein Spezialfall des Narrowing-Kalküls ist: Beschränkt man in allen drei Regeln σ und τ auf die identische Substitution, erhält man exakt die entsprechenden Regeln des Reduktionskalküls und es gilt:

$$G \vdash_{SP} True \iff \langle G, id\rangle \vdash_{SP} \langle True, id\rangle.$$

Im Gegensatz zu Schnitt- und Reduktionskalkül, die nur zur Semantikdefinition eingeführt wurden, bildet der Narrowing-Kalkül die Grundlage für alle Interpreter funktional-logischer Sprachen. Sie unterscheiden sich i.w. in der *Strategie*, nach der Resolution, Narrowing und Unifikation angewendet werden und in den *Simplifikationen*, die zwischen den Regelanwendungen Constraints vereinfachen, um das Auffinden von Lösungen zu beschleunigen. Die folgende Verallgemeinerung von Satz 4.2.14 zeigt, dass mit dem Narrowing-Kalkül tatsächlich alle Goallösungen im initialen Modell von SP herleitbar sind:

Satz 5.1.22 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation. Der Narrowing-Kalkül für SP ist lösungskorrekt bzgl. $Her(SP)$: Für alle Σ-Goals G gilt:

$$\langle G, id\rangle \vdash_{SP} \langle True, \tau\rangle \quad \text{impliziert} \quad Her(SP) \models G\tau.$$

Ist SP terminierend und konsistent, dann ist der Narrowing-Kalkül für SP auch lösungsvollständig: Für alle Σ-Goals G und Substitutionen $\tau : X \to NF_\Sigma$ gilt:

$$Her(SP) \models G\tau \quad \text{impliziert} \quad \langle G, id\rangle \vdash_{SP} \langle True, \tau\rangle.$$

Beweis. Der Nachweis der Lösungskorrektheit folgt dem Beweis von Satz 4.2.14. Dort wurde bereits die Korrektheit von Resolutionsschritten gezeigt. Analog zu 4.2.14 ❶ bleibt zu zeigen:

❶ Für jeden Narrowingschritt und jeden Unifikationsschritt $\langle G, \tau\rangle \vdash_P \langle G', \tau'\rangle$ gilt: $Her(SP) \models G' \Rightarrow G\tau'$.

☞ Beweis! Um die Lösungsvollständigkeit nachzuweisen, schließt man zunächst mithilfe von Korollar 5.1.18:

$$Her(SP) \models G\tau \quad \Rightarrow \quad SP \vdash_{cut} G\tau \quad \Rightarrow \quad G\tau \vdash_{SP} True,$$

zerlegt $G\tau \vdash_{SP} True$ in einen ersten Schritt $G\tau \vdash_{SP} G'$ und den Rest $G' \vdash_{SP} True$, zeigt dann, dass für alle drei Regeln des Reduktionskalküls (5.1.9), die $G\tau \vdash_{SP} G'$ erzeugt haben können, dieser Reduktionsschritt in

den Narrowing-Schritt $\langle G, id\rangle \vdash_{SP} \langle G', \tau\rangle$ entlang einer entsprechenden Regel des Narrowing-Kalküls überführt werden kann, und erhält schließlich

$$\langle G, id\rangle \vdash_{SP} \langle G', \tau\rangle \vdash_{SP} \langle \mathit{True}, \tau\rangle.$$

Der Übergang von $G\tau \vdash_{SP} G'$ nach $\langle G, id\rangle \vdash_{SP} \langle G', \tau\rangle$ ist insbesondere deshalb möglich, weil τ nur Normalformen substituiert. Da aber jedes Axiom von SP mit einer definierten Funktion oder einem Prädikat beginnt, muss jede Reduktion von $G\tau$ in G "hineinreichen". ❑

Analog zur bewachten Resolution (vgl. §4.2) optimiert die folgende Regel die Narrowing-Regel:

bewachtes Narrowing

$$\frac{\langle G(t), \tau\rangle}{\langle G(v)\sigma \wedge H_1\sigma, \tau\sigma\rangle} \Uparrow$$

falls $t \notin X$, $t\sigma = u\sigma$, $H_0 \Rightarrow (u \equiv v \Leftarrow H_1) \in AX$, $Her(SP) \models H_0\sigma$ und der Sukzedent wieder ein Constraint ist

Um als halbwegs effizientes Lösungsverfahren zu taugen, müssen die Anwendungen der Regeln des Narrowing-Kalküls einer deterministischen Strategie folgen, die insbesondere festlegt, in welcher Reihenfolge die Narrowing-Regel auf die Teilterme eines Goals angewendet wird. Eine *outermost*-Strategie transformiert immer die jeweils größten transformierbaren Teilterme, während eine *innermost*-Strategie mit den kleinsten Teiltermen beginnt. Innermost-Strategien sind lösungsvollständig, aber ineffizient (warum?). Outermost-Strategien sind effizienter, aber leider nicht lösungsvollständig, es sei denn, man erweitert den Kalkül um die folgende Regel zum **Needed-Narrowing-Kalkül** [83, 3].

pre-Narrowing

$$\frac{\langle G(t), \tau\rangle}{\langle G(t)\sigma, \tau\sigma\rangle} \Uparrow$$

falls $t \notin X$, σ ein *partieller Unifikator* von t und u ist, $u \equiv v \Leftarrow H \in AX$ und der Sukzedent wieder ein Constraint ist

Ein **partieller Unifikator** σ von t und u substituiert t und u so, dass $t\sigma$ und $u\sigma$ zumindest an allen Konstruktorpositionen übereinstimmen. Beispielsweise erhielte man die Lösung $X \equiv 0 \wedge Y \equiv 0 \wedge Z \equiv 0$ des Goals $f(f(X, Y), Z) \equiv 0$ nicht ohne pre-Narrowing, wie die folgenden mit Expander2 [81] berechneten Formeln einer Narrowing-Ableitung zeigt.[28] Hier ist f eine definierte Funktion mit den Axiomen:

$$f(0, 0) \equiv 0, f(\mathit{suc}(x), 0) \equiv 1, f(x, \mathit{suc}(y)) \equiv 2.$$

$0, 1, 2, \mathit{suc}$ seien Konstruktoren.

```
f(f(x,y),z) = 0

Narrowing the preceding formula leads to

f(f(x,y),0) = 0 & z = 0 | Any y0:(2 = 0 & z = suc(y0))

Narrowing the preceding formula leads to

(f(0,0) = 0 & x = 0 & y = 0 | Any x1:(f(1,0) = 0 & x = suc(x1) & y = 0) |
 Any y1:(f(2,0) = 0 & y = suc(y1))) &
z = 0 |
Any y0:(2 = 0 & z = suc(y0))

Narrowing the preceding formula leads to

(0 = 0 & x = 0 & y = 0 | Any x1:(f(1,0) = 0 & x = suc(x1) & y = 0) |
```

[28] Im Gegensatz zu Beispiel 4.2.13 werden zunächst nur Narrowing- bzw. pre-Narrowing-Schritte durchgeführt. Erst der letzte Schritt besteht aus Simplifikationen, die das Goal auf die einzige Lösung des Startgoals reduzieren.

```
 Any y1:(f(2,0) = 0 & y = suc(y1))) &
z = 0 |
Any y0:(2 = 0 & z = suc(y0))

Narrowing the preceding formula leads to

(0 = 0 & x = 0 & y = 0 | Any x1:(1 = 0 & x = suc(x1) & y = 0) |
 Any y1:(f(2,0) = 0 & y = suc(y1))) &
z = 0 |
Any y0:(2 = 0 & z = suc(y0))

Narrowing the preceding formula leads to

(0 = 0 & x = 0 & y = 0 | Any x1:(1 = 0 & x = suc(x1) & y = 0) |
 Any y1:(1 = 0 & y = suc(y1))) &
z = 0 |
Any y0:(2 = 0 & z = suc(y0))

Simplifying the preceding formula (21 steps) leads to

x = 0 & y = 0 & z = 0

Number of proof steps: 6

Solutions:

x = 0 & y = 0 & z = 0
```

Beispiel 5.1.23 Analog zum Beispiel 4.2.3 listen wir die Formeln einer Narrowing-Ableitung auf, die mit den Lösungen des Goals

$$repByMin(leaf(3)\#(leaf(2)\#leaf(6))) = V \tag{1}$$

(in der Variablen V) endet. Die Ableitung wurde wieder mit Expander2 [81] erstellt.

```
repByMin(leaf(3)#(leaf(2)#leaf(6))) = T

Narrowing the preceding formula leads to

Any T' m:(T' = T & repAndMin(leaf(3)#(leaf(2)#leaf(6)),m) = (T',m))

Narrowing the preceding formula leads to

Any T' m:(T' = T &
          Any U U' n n':((U#U',min(n,n')) = (T',m) &
                         repAndMin(leaf(3),m) = (U,n) &
                         repAndMin(leaf(2)#leaf(6),m) = (U',n')))

Narrowing the preceding formula leads to

Any T' m:(T' = T &
          Any U U' n n':((U#U',min(n,n')) = (T',m) & (leaf(m),3) = (U,n) &
                         repAndMin(leaf(2)#leaf(6),m) = (U',n')))

Narrowing the preceding formula leads to

Any T' m:(T' = T &
          Any U U' n n':((U#U',min(n,n')) = (T',m) & (leaf(m),3) = (U,n) &
                         Any U1 U'1 n1 n'1:((U1#U'1,min(n1,n'1)) = (U',n') &
                                            repAndMin(leaf(2),m) = (U1,n1) &
                                            repAndMin(leaf(6),m) = (U'1,n'1))))
```

```
Narrowing the preceding formula leads to

Any T' m:(T' = T &
          Any U U' n n':((U#U',min(n,n')) = (T',m) & (leaf(m),3) = (U,n) &
                         Any U1 U'1 n1 n'1:((U1#U'1,min(n1,n'1)) = (U',n') &
                                            (leaf(m),2) = (U1,n1) &
                                            repAndMin(leaf(6),m) = (U'1,n'1))))

Narrowing the preceding formula leads to

Any T' m:(T' = T &
          Any U U' n n':((U#U',min(n,n')) = (T',m) & (leaf(m),3) = (U,n) &
                         Any U1 U'1 n1 n'1:((U1#U'1,min(n1,n'1)) = (U',n') &
                                            (leaf(m),2) = (U1,n1) &
                                            (leaf(m),6) = (U'1,n'1))))

Simplifying the preceding formula (38 steps) leads to

leaf(2)#(leaf(2)#leaf(2)) = T
```

Das ist die erwartete Lösung: Die Blätter des ursprünglichen Baum $leaf(3)\#(leaf(2)\#leaf(6))$ wurden mit dessen minimalem Blatteintrag markiert. ✌

5.2 Terminations- und Konfluenzkriterien

Vollständigkeit wird auch **schwache Termination** genannt, weil sie nur die Existenz äquivalenter Normalformen fordert, nicht aber verlangt, dass jede Reduktion terminiert, d.h. $\longrightarrow_{SP}^{-1}$ wohlfundiert ist. Diese Eigenschaft ist, falls alle reduzierten Grundterme Normalformen sind, hinreichend, aber nicht notwendig für die Vollständigkeit von SP. Wie oben bemerkt, läßt sich Vollständigkeit i.a. durch Induktion über Terme zeigen. Die Wohlfundiertheit von $\longrightarrow_{SP}^{-1}$ ist jedoch ein wichtiges Kriterium für Konfluenz, weil dann Konfluenz aus der Gültigkeit endlich vieler **kritischer Klauseln** folgt. Diesen Schluß kann man nämlich durch **Noethersche Induktion** (s. §7.2) entlang einer **Reduktionsordnung** ziehen, die mit den Axiomen verträglich ist und deshalb die Wohlfundiertheit von $\longrightarrow_{SP}^{-1}$ erzwingt.

Reduktionsordnung

Definition 5.2.1 Eine **Reduktionsordnung** $>$ **für** SP ist eine transitive Relation auf Σ-Grundtermen und -goals mit folgenden Eigenschaften:

Wohlfundiertheit	$<=_{def}>^{-1}$ ist wohlfundiert.
AX**-Verträglichkeit**	$r(t)\sigma > H\sigma$ bzw. $c(t\sigma) > (c(u\sigma) \wedge H\sigma)$ für alle Axiome $r(t) \Leftarrow H$ bzw. $t \equiv u \Leftarrow H$ von SP, Grundsubstitutionen σ und Terme $c(x)$ mit $H\sigma \vdash_{SP} True$ und $(var(H) \setminus var(t))\sigma \subseteq NF_\Sigma$.
Teiltermverträglichkeit	$t > u$ für alle $t \in T_\Sigma$ und alle echten Teilterme u von t.[29]

Eine SP-Reduktion $G_1, \ldots, G_n$ ist $>$**-absteigend**, wenn für alle $1 \leq i < n$ $G_i > G_{i+1}$ gilt.

Eine Spezifikation SP ist **stark terminierend**, wenn es eine Reduktionsordnung für SP gibt und alle SP-reduzierten Terme Normalformen sind (siehe Def. 5.1.10).

☞ (A) Zeigen Sie, dass jede stark terminierende Spezifikation terminierend ist (siehe Def. 5.1.10)!

[29] M.a.W.: Die Teiltermordnung ist eine Teilmenge von $>$.

☞ (B) Zeigen Sie, dass jede erfolgreiche SP-Reduktion $>$-absteigend ist, falls $>$ eine Reduktionsordnung für SP ist!

Eine Reduktionsordnung für (Σ, AX) läßt sich oft aus einer transitiven Ordnung $>_\Sigma$ der (endlich vielen) Symbole von Σ und einer wohlfundierten Ordnung $\ll$ der Normalformen zusammensetzen. Für jedes Axiom $f(t) \equiv u \Leftarrow H$ wird dann $>_\Sigma$ so gewählt, dass für alle Symbole g, die in u oder H vorkommen, $f >_\Sigma g$ gilt. Daraus folgt insbesondere, dass Konstruktoren minimal bzgl. $>_\Sigma$ sind. $>_\Sigma$ ist nicht notwendig antisymmetrisch. Sind z.B. zwei Funktionen f und g wechselseitig-rekursiv definiert, d.h. es gibt zwei Axiome $f(t) \equiv u \Leftarrow H$ und $g(t') \equiv u' \Leftarrow H'$ so, dass g in u oder H und f in u' oder H' vorkommen, dann gilt sowohl $f >_\Sigma g$ als auch $g >_\Sigma f$. In diesem Fall muss die Normalformordnung $\ll$ garantieren, dass die jeweiligen Argumente von f bzw. g kleiner werden.

Satz 5.2.2 (Terminationskriterium) Sei $\Sigma = (S, F, R)$ und $SP = (\Sigma, AX)$ eine vollständige Spezifikation derart, dass alle SP-reduzierten Terme Normalformen sind und es eine wohlfundierte Ordnung $\ll$ auf NF_Σ gibt.

Eine binäre Relation $>_\Sigma$ auf $F \cup R$ ist wie folgt definiert:

$$f >_\Sigma g \quad \Longleftrightarrow_{def} \quad \text{es gibt } f(t)\{\equiv u\} \Leftarrow H \text{ in } AX \text{ und } g \text{ kommt in } u \text{ oder } H \text{ vor.}^{30}$$

SP ist stark terminierend, wenn für alle $f(t)\{\equiv u\} \Leftarrow H \in AX$, Atome und Teilterme $g(t')$ von u oder H, Grundsubstitutionen σ, Normalformen v von $t\sigma$ und v' von $t'\sigma$ folgendes gilt:

$$g >_\Sigma^+ f \wedge SP \vdash_{cut} H\sigma \wedge (var(H) \setminus var(t))\sigma \subseteq NF_\Sigma \quad \Rightarrow \quad v' \ll v. \quad \square$$

☞ Zeigen Sie mithilfe dieses Kriteriums, dass DIVREP (Bsp. 5.1.5) stark terminierend ist!

Der Beweis von Satz 5.2.2 läuft über mehrere Stufen. Ausgehend von $>_\Sigma$ wird induktiv eine Reduktionsordnung $>$ konstruiert, von der man zunächst die Transitivität und dann, durch Kontraposition, die Wohldefiniertheit zeigt. Die AX-Verträglichkeit von $>$ folgt aus der o.g. Bedingung an $>_\Sigma$. Die Teiltermverträglichkeit von $>$ ist Teil der induktiven Definition von $>$. Details findet man in [80], §6.2. Terminationskriterien für Termersetzungssysteme bilden ein eigenes Forschungsgebiet.

Satz 5.2.2 greift auch bei geschachtelter Rekursion. So erfüllen z.B. die Axiome der Ackermann-Funktion die Bedingung von 5.2.2, wenn man $\ll$ als lexikographische Erweiterung der Teiltermordnung wählt (s. Bsp. 5.1.5):

```
ACK = NAT then
        defuncts  ack:nat*nat -> nat
        axioms    ack(0,n) = suc(n)
                  ack(suc(m),0) = ack(m,1)
                  ack(suc(m),suc(n)) = ack(m,ack(suc(m),n))
```

Nach Definition von $>_\Sigma$ (s. 5.2.2) gilt: $ack >_\Sigma ack >_\Sigma \{0, 1, suc\}$. Außerdem ist $\ll$ wohlfundiert und man erhält für alle ACK-Grundterme t, u:[31]

$$(suc(t), 0) \gg (t, 1), \quad (suc(t), suc(u)) \gg (t, ack(suc(t), u)), \quad (suc(t), suc(u)) \gg (suc(t), u).$$

Damit sind alle Bedingungen von 5.2.2 erfüllt.

Ist SP terminierend, dann läßt sich die Konfluenz von SP auf lokale Bedingungen an Paare einzelner Reduktionsschritte zurückführen. Die Idee dieser Zurückführung mithilfe der Termination wollen wir an einem

[30] Die Optionalschreibweise $t\{\equiv u\}$ umfaßt den Fall einer Gleichung $t \equiv u$ und den Fall eines logischen Atoms t.
[31] $\gg$ die zu $\ll$ inverse Relation.

einfachen Konfluenzkriterium zeigen. Auch wenn dieses Kriterium ist noch weit entfernt von einer entscheidbaren Bedingung ist, so bildet es doch die wesentliche Grundlage für die praktisch relevanteren 5.2.6, 5.2.7 und 5.2.9.

Theorem 5.2.3 Sei $SP = (\Sigma, AX)$ terminierend und **lokal konfluent**, d.h. für alle Σ-Grundterme t, t_1, t_2 mit $t \longrightarrow_{SP} t_1$ und $t \longrightarrow_{SP} t_2$ gibt es einen Term u mit $t_1 \xrightarrow{*}_{SP} u$ und $t_2 \xrightarrow{*}_{SP} u$. Dann ist SP konfluent.

Beweis. Da $\longrightarrow_{SP}^{-1}$ wohlfundiert ist, können wir *Noethersche Induktion* entlang $\longrightarrow_{SP}$ verwenden (siehe §7.2). Seien u_1 und u_2 zwei SP-Redukte von t. Ist $t = u_1$ oder $t = u_2$, dann ist u_2 bzw. u_1 ein gemeinsames SP-Redukt von u_1 und u_2. Andernfalls gibt es Terme t_1, t_2 mit $t \longrightarrow_{SP} t_1 \xrightarrow{*}_{SP} u_1$ und $t \longrightarrow_{SP} t_2 \xrightarrow{*}_{SP} u_2$. Da SP lokal konfluent ist, haben t_1 und t_2 ein gemeinsames SP-Redukt u. Wegen $t \longrightarrow_{SP} t_1$ und $t \longrightarrow_{SP} t_2$, können wir die Induktionsvoraussetzung auf t_1 und t_2 anwenden: Aus $u \xleftarrow{*}_{SP} t_1 \xrightarrow{*}_{SP} u_1$ folgt $u \xrightarrow{*}_{SP} v_1 \xleftarrow{*}_{SP} u_1$ für einen Term v_1 und aus $u \xleftarrow{*}_{SP} t_2 \xrightarrow{*}_{SP} u_2$ folgt $u \xrightarrow{*}_{SP} v_2 \xleftarrow{*}_{SP} u_2$ für einen Term v_2. Wegen $t \longrightarrow_{SP} t_1 \xrightarrow{*}_{SP} u$, können wir die Induktionsvoraussetzung auch auf u anwenden: Aus $v_1 \xleftarrow{*}_{SP} u \xrightarrow{*}_{SP} v_2$ folgt $v_1 \xrightarrow{*}_{SP} v \xleftarrow{*}_{SP} v_2$ für einen Term v. Wegen $u_1 \xrightarrow{*}_{SP} v_1 \xrightarrow{*}_{SP} v$ und $u_2 \xrightarrow{*}_{SP} v_2 \xrightarrow{*}_{SP} v$ ist v ein gemeinsames SP-Redukt von u_1 und u_2. ❑

Die meisten Paare von Reduktionsschritten mit gleichem Ausgangsterm t entstehen dadurch, dass zwei Gleichungsaxiome auf t angewendet werden und sich dabei die linken Seiten der beiden Axiome in t nicht überlappen. Für solche Paare erhält man lokale Konfluenz automatisch. Es bleiben also nur die kritischen, d.h. überlappenden Fälle zu prüfen. Handelt es sich um *bedingte* Gleichungsaxiome, dann muss man die in Theorem 5.2.3 verwendete Induktion entlang der Reduktionsordnung allerdings noch ein zweites Mal bemühen und zwar, um eine geeignete Induktionshypothese über die Prämissen der Axiome zu bekommen. Damit erhält man schließlich ein Konfluenzkriterium, das (fast) nur noch syntaktische Bedingungen an Paare von Axiomen und die sich daraus ergebenden *kritischen Klauseln* stellt:

kritische Klausel

Definition 5.2.4 Seien $SP = (\Sigma, AX)$ eine Spezifikation, $v(x)$ ein Σ-Term oder -Atom mit genau einem Vorkommen der Variablen x,

$$\varphi \quad = \quad t\{\equiv u\} \Leftarrow G \quad \text{und} \quad \psi \quad = \quad t' \equiv u' \Leftarrow H$$

Axiome von SP und σ, τ *minimale* Substitutionen[32] derart, dass $t\sigma = v(t'\tau)$ gilt und die Position von x in $v(x)$ mit der Position eines Funktionssymbols in t übereinstimmt. Dann heißt die Hornformel

$$KK \quad = \quad v(u'\tau)\{\equiv u\sigma\} \Leftarrow G\sigma \wedge H\tau$$

von φ und ψ bei $t\sigma$ erzeugte **kritische Klausel** von SP. KK heißt SP**-Overlay**, falls $v(x) = x$.[33]

SP ist **orthogonal**, wenn für jede kritische Klausel $t\{\equiv u\} \Leftarrow G$ von SP $t = u$ oder G in $Her(SP)$ unlösbar ist.

φ und ψ erzeugen immer dann eine kritische Klausel, wenn die linke Seite von ψ diejenige von φ **überlappt**. Im Fall $v(x) = x$ sind φ *und* ψ bedingte Gleichungen und die linken Seiten von φ bzw. ψ sind unifizierbar. Kritisch für die Konfluenz (s. Def. 5.1.10) ist eine Überlappung immer dann, wenn es ein Grundgoal G_1 gibt, das einerseits durch Anwendung von φ lösbar wird ($G_1 \vdash_{SP} True$) und andererseits durch Anwendung von ψ reduziert werden kann ($G_1 \longrightarrow_{SP} G_2$). Das ist aber nur möglich, wenn die Rümpfe $G\sigma$ und $H\tau$ der Überlappungsinstanzen von φ bzw. ψ *gleichzeitig* (in $Her(SP)$) lösbar sind.

[32] minimal bzgl. der **Subsumptionsordnung** $\leq$, die definiert ist durch: $\sigma \leq \sigma' \Leftrightarrow \exists \tau : \sigma\tau = \sigma'$.
[33] In diesem Fall ist φ eine bedingte Gleichung.

Die *min*- und *div*-Axiome von DIVREP (Bsp. 5.1.5) erzeugen z.B. die kritischen Klauseln

$$\begin{aligned}
m \equiv n &\quad \Leftarrow \quad m \leq n \wedge n \leq m \\
(0, m) \equiv (suc(q), r) &\quad \Leftarrow \quad m < n \wedge 0 < n \leq m \wedge div(m - n, n) \equiv (q, r) \\
(0, m) \equiv () &\quad \Leftarrow \quad m < 0 \\
(suc(q), r) \equiv () &\quad \Leftarrow \quad m < 0 \wedge 0 < 0 \leq m \wedge div(m - 0, 0) \equiv (q, r).
\end{aligned}$$

Außerdem erzeugt jede bedingte Gleichung mit lösbarem Rumpf durch Überlappung mit sich selbst eine kritische Klausel. Z.B. liefert die Überlappung des zweiten *div*-Axioms von DIVREP die kritische Klausel

$$(suc(q), r) \equiv (suc(q'), r') \quad \Leftarrow \quad 0 < n \leq m \wedge div(m - n, n) \equiv (q, r) \wedge div(m - n, n) \equiv (q', r').$$

☞ Zeigen Sie, dass DIVREP nicht orthogonal ist!

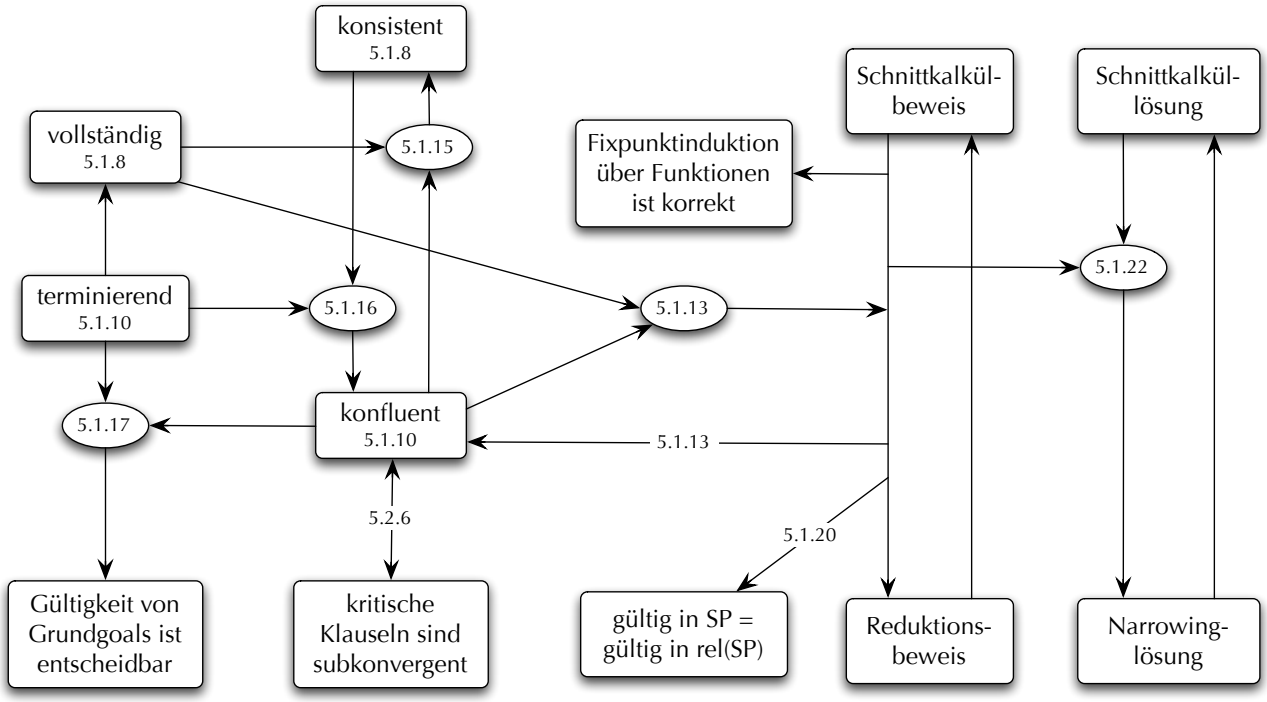


Figure 2. *Termination, Konfluenz und Reduktionsbeweise (Pfeile bezeichnen Implikationen).*

Eine terminierende Spezifikation ist konfluent, wenn ihre kritischen Klauseln in folgendem Sinne gültig sind:

Subkonvergenz

Definition 5.2.5 Sei $SP = (\Sigma, AX)$ eine terminierende Spezifikation mit Reduktionsordnung $>$ und t ein Σ-Term oder -Atom. Eine Σ-Hornformel $p \Leftarrow H$ ist **sub-t-konvergent**, wenn für alle Grundsubstitutionen σ derart, dass alle erfolgreich SP-reduzierbaren Grundgoals $G < t\sigma$ SP-konvergent sind, folgendes gilt:

$$H\sigma \vdash_{SP} True \quad \Rightarrow \quad p\sigma \vdash_{SP} True.$$

Eine bei t erzeugte kritische Klausel ist **subkonvergent** oder **subjoinable**, wenn sie sub-t-konvergent ist.

Subkonvergenz berücksichtigt also nur solche Grundsubstitutionen, die, grob gesagt, unterhalb von t konfluent sind. Subkonvergenz ist i.a. weder stärker noch schwächer als induktive Gültigkeit. Kritische Klauseln von SP sind immer induktive Theoreme von SP (☞ Beweis!), aber nicht immer subkonvergent. Das ist gerade ein Kriterium für Konfluenz:

Theorem 5.2.6 (Superpositionstheorem) Eine stark terminierende Spezifikation SP ist genau dann konfluent, wenn alle kritischen Klauseln von SP subkonvergent sind.

Beweis. "⇒": Seien $SP = (\Sigma, AX)$ konfluent, $>$ eine Reduktionsordnung für SP, φ, ψ und KK wie in Def. 5.2.4 und ρ eine Grundsubstitution derart, dass $G\sigma\rho \wedge H\tau\rho$ erfolgreich SP-reduzierbar ist und alle erfolgreich SP-reduzierbaren Grundgoals $G < t\sigma\rho$ SP-konvergent sind. Da $<$ AX-verträglich ist, gilt $t\sigma\rho > G\sigma\rho \wedge H\tau\rho$. Also sind $G\sigma\rho$ und $H\tau\rho$ SP-konvergent. Da $<$ wohlfundiert und AX-verträglich ist und alle SP-reduzierten Terme Normalformen sind, gibt es ein Normalformredukt ρ' von ρ. Daraus folgt $G\sigma\rho \xrightarrow{*}_{SP} G\sigma\rho'$ und $H\tau\rho \xrightarrow{*}_{SP} H\tau\rho'$. Da $G\sigma\rho$ und $H\tau\rho$ stark SP-konvergent sind, sind $G\sigma\rho'$ and $H\tau\rho'$ erfolgreich SP-reduzierbar. Da φ und ψ Axiome von SP sind, folgt $t\sigma\rho'\{\equiv u\sigma\rho'\} \vdash_{SP} \mathit{True}$ und $t\sigma\rho' = v[t'\tau/x]\rho' \xrightarrow{*}_{SP} v[u'\tau/x]\rho'$. Da SP konfluent ist, folgt wegen Satz 5.1.11 schließlich $v[u'\tau/x]\rho'\{\equiv u\sigma\rho'\} \vdash_{SP} \mathit{True}$, also auch $v[u'\tau/x]\rho\{\equiv u\sigma\rho\} \vdash_{SP} \mathit{True}$.

"⇐": Für den Fall unbedingter Gleichungen als Axiome findet man einen Beweis z.B. in [49], §6.3. Die allgemeine Behauptung entspricht Theorem 6.10 von [80], in dessen Beweis die Reduktionsordnung $>$ als Induktionsordnung benutzt wird. Man zeigt durch Induktion über p entlang $>$, dass alle erfolgreich SP-reduzierbaren Grundatome p konvergent sind, also wegen Satz 5.1.11 SP konfluent ist. Dabei wird u.a. die ähnlich Lemma 5.1.14 beweisbare Tatsache verwendet, dass alle Axiome $t\{\equiv u\} \Leftarrow H$ sub-t-konvergent sind. ❑

Wir geben uns mit diesem Satz noch nicht zufrieden, weil er den Konfluenzbeweis zwar auf die Prüfung endlich vieler Formeln reduziert, die zu prüfende Eigenschaft selbst (Subkonvergenz) aber keine syntaktische Eigenschaft ist. Orthogonalität (s. Def. 5.2.4) ist zwar hinreichend, schließt aber überlappende Axiome mit lösbaren Prämissen aus:

Korollar 5.2.7 (Konfluenzkriterium) Jede orthogonale und stark terminierende Spezifikation ist konfluent.

Beweis. ☞ Übung. ❑

Die folgende Bedingung liefert ein syntaktisches Konfluenzkriterium (5.2.9), das überlappende Axiome zuläßt:

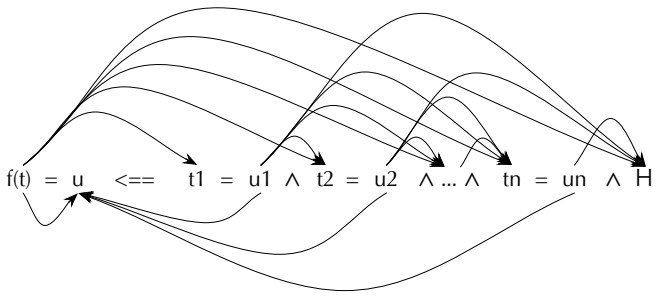


Figure 3. *"Datenfluss" einer deterministischen Hornformel: Jede Variable eines Terms $v \in \{u, t_1, \ldots, t_n\}$ kommt im Quellterm einer Kante vor, die auf v zeigt.*

deterministische Hornformel

Definition 5.2.8 Sei $SP = (\Sigma, AX)$ eine Spezifikation. Eine Σ-Hornformel φ der Form

$$f(t)\{\equiv u\} \quad \Leftarrow \quad t_1 \equiv u_1 \wedge \cdots \wedge t_n \equiv u_n \wedge H \tag{1}$$

heißt **deterministisch**, falls die Terme $u_1, \ldots, u_n$ Normalformen sind, $var(u) \subseteq V_n$ und für alle $1 \leq i \leq n$ $var(t_i) \subseteq V_{i-1}$ gilt, wobei $V_0 =_{def} var(t)$ und $V_i =_{def} V_{i-1} \uplus var(u_i)$.

▲ ▲

Deterministisch ist hier der Datenfluß durch die Formel φ bei ihrer Anwendung (in einem Resolutions- bzw. Rewritingschritt): Falls die Normalform u_i eine frische Variable "erzeugt", ist deren "Verwendung" auf die Terme $t_{i+1}, \ldots, t_n$ und u beschränkt.

Deterministische Axiome geben die übliche Struktur funktionaler Programms wieder. So läßt sich z.B. die Axiome

$$f(t_1) \equiv u_1 \;\Leftarrow\; t_{11} \equiv u_{11} \wedge \cdots \wedge t_{1n_1} \equiv u_{1n_1}$$
$$\ldots$$
$$f(t_k) \equiv u_k \;\Leftarrow\; t_{k1} \equiv u_{k1} \wedge \cdots \wedge t_{kn_k} \equiv u_{kn_k}$$

direkt in folgendes Haskell-Programm mit lokalen Definitionen übersetzen:

$$\texttt{f(}t_1\texttt{) = let } u_{11} \texttt{ = } t_{11} \texttt{ ... } u_{1n_1} \texttt{ = } t_{1n_1} \texttt{ in } u_1$$
$$\ldots$$
$$\texttt{f(}t_k\texttt{) = let } u_{k1} \texttt{ = } t_{k1} \texttt{ ... } u_{kn_k} \texttt{ = } t_{kn_k} \texttt{ in } u_k$$

Außer

$$repByMin(T) \equiv T' \;\Leftarrow\; rep\&min(T, m) \equiv (T', m) \qquad (\varphi)$$

sind alle Axiome von DIVREP (Bsp. 5.1.5) deterministisch. φ ist nicht deterministisch, weil die Variable m auf beiden Seiten der Rumpfgleichung vorkommt. M.a.W.: Eine Belegung der frischen Variablen m und T' kann *nicht* durch sequentielle Auswertung von Termen berechnet werden. Dennoch hat der Rumpf von φ offensichtlich eine eindeutige Lösung in m und T'. Die von φ und φ bei $repByMin$ erzeugte kritische Klausel

$$T' \equiv U \;\Leftarrow\; rep\&min(T, m) \equiv (T', m) \wedge rep\&min(T, n) \equiv (U, n) \qquad (KK)$$

ist nämlich ein induktives Theorem von DIVREP$\backslash\{repByMin, \varphi\}$. Tatsächlich ist auch das hinreichend für die Subkonvergenz von KK und damit für die Konfluenz von DIVREP:

Satz 5.2.9 (Konfluenzkriterium) Sei $SP = (\Sigma, AX)$ eine stark terminierende Spezifikation und $SP' = (\Sigma', AX')$ eine konfluente Teilspezifikation von SP. SP ist konfluent, falls

(a) die Zielsorte jedes Konstruktors von $\Sigma \setminus \Sigma'$ zu $\Sigma \setminus \Sigma'$ gehört.

(b) für alle Axiome $f(t)\{\equiv u\} \Leftarrow H$ von $SP \setminus SP'$ f zu $\Sigma \setminus \Sigma'$ gehört,

(c) für alle bedingten Gleichungen φ, ψ von $AX \setminus AX'$ gilt:

(c.1) $\varphi = \psi$ und φ ist deterministisch oder

(c.2) jedes von φ und ψ erzeugte Overlay ist ein induktives Theorem von SP'.

Beweis. Nach Satz 5.2.6 ist zu zeigen, dass alle kritischen Klauseln von SP subkonvergent sind. Wegen (b) und da SP' konfluent ist, folgt umgekehrt aus Satz 5.2.6, dass alle von zwei AX'-Formeln erzeugten kritischen Klauseln bereits subkonvergent sind. Jede andere kritische Klausel von SP ist demnach ein von zwei Formeln $\varphi, \psi \in AX \setminus AX'$ erzeugtes Overlay KK.

Fall 1: $\varphi = \psi$ und φ ist deterministisch. Dann hat φ die Form (1) und erfüllt die Bedingungen von Def. 5.2.8. Demnach gibt es eine Variablenumbenennung ρ derart, dass $t = t\rho$ gilt und KK die Form

$$u \equiv u\rho \;\Leftarrow\; t_1 \equiv u_1 \wedge \cdots \wedge t_n \equiv u_n \wedge H \wedge (t_1 \equiv u_1 \wedge \cdots \wedge t_n \equiv u_n \wedge H)\rho \qquad (2)$$

hat. Seien G der Rumpf von KK und σ eine Grundsubstitution so, dass $G\sigma$ erfolgreich SP-reduzierbar ist und alle erfolgreich SP-reduzierbaren Grundgoals $G' < f(t\sigma)$ SP-konvergent sind. Da SP stark terminierend ist, gibt es $\tau : X \to NF_\Sigma$ mit $G\sigma \xrightarrow{*}_{SP} G\tau$. Wegen $f(t\sigma) > G\sigma$ sind $G\sigma$ und damit auch $G\tau$ SP-konvergent. Sei $V_0 = var(t)$ und $V_i = V_{i-1} \cup var(u_i)$ für alle $1 \leq i \leq n$.

Wir zeigen durch Induktion über i, dass für alle $0 \leq i \leq n$ und $x \in V_i$ $x\tau = x\rho\tau$ gilt. Wegen $t = t\rho$ gilt $x\tau = x\rho\tau$ für alle $x \in V_0$. Es gelte $x\tau = x\rho\tau$ für alle $x \in V_{i-1}$. Wegen $var(t_i) \subseteq V_{i-1}$ folgt $t_i\tau = t_i\rho\tau$. Da $t_i\tau \equiv u_i\tau$ erfolgreich SP-reduzierbar ist, gibt es u mit

$$u_i\tau \xrightarrow{*}_{SP} u \xleftarrow{*}_{SP} t_i\tau = t_i\rho\tau.$$

Da $u_i\tau$ eine Normalform ist, folgt $u_i\rho\tau = u$, also

$$u_i\tau \xleftarrow{*}_{SP} t_i\tau = t_i\rho\tau.$$

Die erfolgreiche Reduzierbarkeit von $(t_i\rho\tau \equiv u_i\rho\tau) \in G\tau$ impliziert $u_i\tau \equiv u_i\rho\tau \vdash_{SP} \mathit{True}$. Wir erhalten v mit

$$u_i\tau \xrightarrow{*}_{SP} v \xleftarrow{*}_{SP} u_i\rho\tau,$$

also $u_i\tau = u_i\rho\tau$, weil $u_i\tau$ und $u_i\rho\tau$ Normalformen sind. Daraus folgt $x\tau = x\rho\tau$ für alle $x \in var(u_i)$, also $x\tau = x\rho\tau$ für alle $x \in V_{i-1} \cup var(u_i) = V_i$.

Insbesondere ist $x\tau = x\rho\tau$ für alle $x \in V_n$, so dass $var(u) \subseteq V_n$ $u\tau = u\rho\tau$ impliziert. Demnach ist $u\sigma \equiv u\rho\sigma$ erfolgreich SP-reduzierbar, womit in Fall 1 die Subkonvergenz von KK bewiesen ist.

Fall 2: KK ist ein induktives Theorem von SP'. Dann ist G ein Σ'-Goal. Insbesondere gehören die Sorten aller Variablen von G zu Σ'. Sei $KK = (t \equiv u \Leftarrow G)$ und σ eine Grundsubstitution derart, dass alle erfolgreich SP-reduzierbaren Grundgoals $G' < t\sigma$ SP-konvergent sind. Sei $G\sigma$ erfolgreich SP-reduzierbar. Da SP stark terminierend ist und SP und SP' wegen (a) für alle Sorten von Σ' dieselben Normalformen haben, gibt es $\tau : X \to NF_\Sigma$ mit $G\sigma \xrightarrow{*}_{SP} G\tau$. Wegen $t\sigma > G\sigma$ sind $G\sigma$ und damit auch $G\tau$ SP-konvergent. $G\tau$ ist ein Σ'-Goal, weil G ein solches ist. Daher ist wegen (b) kein Axiom von $AX \setminus AX'$ auf $G\tau$ anwendbar. Aus der erfolgreichen SP-Reduzierbarkeit von $G\tau$ folgt deshalb die erfolgreiche SP'-Reduzierbarkeit und damit $SP' \vdash_{cut} G\tau$. Da KK ein induktives Theorem von SP' ist, gilt also auch $SP' \vdash_{cut} t\tau \equiv u\tau$. Da SP' konfluent ist, folgt die erfolgreiche SP'-Reduzierbarkeit und damit auch die erfolgreiche SP-Reduzierbarkeit von $t\tau \equiv u\tau$ aus Satz 5.1.13. Also ist auch $t\sigma \equiv u\sigma$ erfolgreich SP-reduzierbar. Damit ist auch in diesem Fall die Subkonvergenz von KK bewiesen. ❑

☞ Zeigen Sie mithilfe von Satz 5.2.9, dass DIVREP (Bsp. 5.1.5) konfluent ist!

6 Strukturierungskonzepte

6.1 Relative Konsistenz

So wie man große Programme in Module aufteilt, um sie später über Schnittstellen miteinander zu verbinden, soll das auch mit großen Spezifikationen geschehen. Hier wie dort unterstützen Strukturierungskonzepte den modularen Entwurf. Darüberhinaus ist bei der Wahl syntaktischer Konstrukte zur Modularisierung von Spezifikationen zu berücksichtigen, dass jene mit einer formalen Semantik ausgestattet werden können, die sich in sinnvoller Weise aus der Semantik einzelner Module zusammensetzt. Dazu müssen neue Begriffe eingeführt werden, die syntaktische bzw. semantische Anforderungen an die *Beziehung* zwischen mehreren Spezifikation(smodul)en beschreiben. Der erste dieser Begriffe ist der Signaturmorphismus, der die Komponenten zweier Signaturen in Beziehung setzt:

Signaturmorphismus, σ-Redukt

Definition 6.1.1 Seien $\Sigma = (S, F, R)$ und $\Sigma' = (S', F', R')$ Signaturen (s. Def. 3.5). Ein **Signaturmorphismus** $\sigma : \Sigma \to \Sigma'$ besteht aus Abbildungen von S, F und R nach S', F' bzw. R' mit folgenden Eigenschaften:

- Für alle $s_1, \ldots, s_n \in S$ ist $\sigma(s_1 \ldots s_n) =_{def} \sigma(s_1) \ldots \sigma(s_n)$.
- Für alle $w \in S^+$ und $f \in F_w$ ist $\sigma(f) \in F'_{\sigma(w)}$.
- Für alle $w \in S^+$ und $r \in R_w$ ist $\sigma(r) \in R'_{\sigma(w)}$.

σ wird wie folgt auf Terme, Formeln und Spezifikationen fortgesetzt. Ist φ ein Σ-Term oder eine Σ-Formel, dann entsteht $\sigma(\varphi)$ aus φ, indem jedes Symbol $f \in \Sigma$ durch $\sigma(f)$ ersetzt wird. Außerdem wird jede Variable einer Sorte $s \in \Sigma$ zur Variable der Sorte $\sigma(s)$.

Analog zu Substitutionen schreiben wir $[g_1/f_1, \ldots, g_n/f_n]$ für einen Signaturmorphismus, der für alle $1 \leq i \leq n$ f_i auf g_i abbildet, im übrigen aber keine Symbole verändert.

Sei A eine Σ'-Struktur (s. Def. 4.1.2). Das **σ-Redukt von** A, geschrieben: $A|_\sigma$, ist die wie folgt definierte Σ-Struktur:

- Für alle $s \in S$ ist $(A|_\sigma)_s = A_{\sigma(s)}$.
- Für alle $f \in F \cup R$ ist $f^{A|_\sigma} = \sigma(f)^A$.

Die kleinste Σ-Unterstruktur von $A|_\sigma$ (s. Def. 4.1.2) nennen wir **σ-Restriktion von** A und bezeichnen sie mit A_σ. Nach Def. 4.1.6 ist also $A_\sigma = (A|_\sigma)_\Sigma$.

Ist σ eine Inklusion und ist AX in AX' enthalten, dann nennen wir SP' eine **Extension von** SP.

☞ Wann hat das σ-Redukt von A weniger Trägermengen als A?

Signaturmorphismen ersetzen Symbole durch Symbole. Substitutionen (die i.a. auch mit kleinen griechischen Buchstaben bezeichnet werden) ersetzen Variablen durch Terme (s. Def. 4.1.7).

Während das σ-Redukt $A|_\sigma$ dieselben Datenbereiche wie A hat, enthält die σ-Restriktion A_σ nur noch Interpretationen von $\sigma(\Sigma)$-Grundtermen.

☞ Zeigen Sie, dass ein Σ'-Grundterm genau dann zu $Her(SP)_\sigma$ gehört, wenn er das σ-Bild eines Σ-Grundterms ist!

Ist σ eine Inklusion, also Σ eine Teilsignatur von Σ', dann schreiben wir Σ anstelle von σ und erhalten damit $(A_\Sigma)_s \subseteq (A|_\Sigma)_s = A_s$ für alle Sorten $s \in \Sigma$.

Lemma 6.1.2 (Gültigkeit in Redukten und Restriktionen) Sei $\sigma : \Sigma \to \Sigma'$ ein Signaturmorphismus, A eine Σ'-Struktur und φ eine Σ-Formel. Es gilt:

$$A|_\sigma \models \varphi \iff A \models \sigma(\varphi), \tag{1}$$

$$A_\sigma \models \varphi \iff \forall \tau : X \to T_\Sigma : A \models \sigma(\varphi\tau). \tag{2}$$

Beweis. ☞ Übung. ❑

induktive Äquivalenz

Definition 6.1.3 Seien $SP_1 = (\Sigma, AX_1)$ und $SP_2 = (\Sigma, AX_2)$ Spezifikationen mit derselben Signatur Σ. SP_1 und SP_2 sind **induktiv äquivalent**, wenn für alle Σ-Grundatome p gilt:

$$Her(SP_1) \models p \iff Her(SP_2) \models p.$$

Zeigen Sie folgende Behauptungen:

(1) Sind SP_1 und SP_2 induktiv äquivalent, dann gilt für alle prädikatenlogischen Formeln φ,

$$Her(SP_1) \models \varphi \quad \Longleftrightarrow \quad Her(SP_2) \models \varphi.$$

(2) Sind SP_1 und SP_2 induktiv äquivalent und ist SP_1 funktional, dann ist auch SP_2 funktional.

(3) SP_1 und SP_2 sind genau dann induktiv äquivalent, wenn SP_1 konsistent bzgl. (SP_2, Σ) und SP_2 konsistent bzgl. (SP_1, Σ) ist.

(4) SP_1 und SP_2 sind genau dann induktiv äquivalent, wenn $Her(SP_1)$ die Axiome von SP_2 und $Her(SP_2)$ die Axiome von SP_1 erfüllt.

Die Begriffe Vollständigkeit und Konsistenz (s. Def. 5.1.8) beschreiben Beziehungen zwischen dem Herbrandmodell einer Spezifikation SP und dem Herbrandmodell der größten Teilspezifikation von SP, die keine definierten Funktionen enthält. Beide Eigenschaften lassen sich zu Bedingungen verallgemeinern, die ein Paar von Strukturen zweier Signaturen betreffen, die selbst durch einen Signaturmorphismus miteinander verbunden sind:

relative Vollständigkeit, Monotonie, Konsistenz

Definition 6.1.4 Seien $SP = (\Sigma, AX)$ und $SP' = (\Sigma', AX')$ Spezifikationen und $\sigma : \Sigma \to \Sigma'$ ein Signaturmorphismus.

SP' ist **(relativ) vollständig** bzgl. σ, wenn für alle Sorten $s \in \Sigma$ und $t' \in T_{\Sigma', \sigma(s)}$ ein Σ-Grundterm t mit $t' \equiv_{SP'} \sigma(t)$ existiert.

SP' ist **monoton bzgl.** (SP, σ), wenn $Her(SP')_\sigma$ monoton bzgl. $Her(SP)$ ist, wenn also alle in $Her(SP)$ gültigen Grundatome auch in $Her(SP')_\sigma$ gelten (s. 4.1.8).

SP' ist **(relativ) konsistent** bzgl. (SP, σ), wenn $Her(SP)$ monoton bzgl. $Her(SP')_\sigma$ ist.

Ist σ eine Inklusion, d.h. für alle $f \in \Sigma$ gilt $\sigma(f) = f$, dann schreiben wir SP anstelle von (SP, σ).

☞ Zeigen Sie unter den Bedingungen von Def. 6.1.4 folgende Behauptungen:

(A) SP' ist genau dann vollständig bzgl. σ, wenn das σ-Redukt von $Ini(SP')$ mit der σ-Restriktion von $Ini(SP')$ übereinstimmt.

(B) SP' ist monoton bzgl. (SP, σ), wenn $Her(SP')_\sigma$ ein SP-Modell ist.

(C) $Her(SP')_\Sigma$ ist ein SP-Modell, wenn SP' eine Extension von SP ist.

Aus (B) und (C) folgt, dass jede Extension von SP monoton bzgl. SP ist.

Wie die Vollständigkeit einer Spezifikation, so lässt sich auch ihre *relative* Vollständigkeit i.d.R. einfach durch Induktion über Grundterme zeigen. Zum Nachweis relativer Konsistenz von Extensionen kann man folgende Kriterien verwenden. Das erste (6.1.5(1)) nützt kaum, weil es die Hinzunahme neuer Symbole bei der Extension verbietet. Es wird auch eher umgekehrt verwendet, nämlich zum Nachweis der Gültigkeit von Hornformeln *durch* Reduktion auf einen Konsistenzbeweis. Das zweite Kriterium (6.1.5(2)) lässt sich einfach zeigen, wenn man sich klarmacht, wie unter den dort genannten Voraussetzungen die Beweis induktive Theoreme aussehen. Das dritte Kriterium (6.1.5(3)) schwächt die Voraussetzungen von 6.1.5(2) ab, verlangt aber starke Vollständigkeit und Konsistenz von SP'.

Satz 6.1.5 (Konsistenzkriterien) Sei $SP' = (\Sigma', AX')$ eine Extension von $SP = (\Sigma, AX)$.

(1) Sei $\Sigma = \Sigma'$. SP' ist genau dann konsistent bzgl. SP, wenn alle Axiome von $SP' \setminus SP$ induktive Theoreme von SP sind.

SP' ist konsistent bzgl. SP, falls

(2) die Zielsorte jedes Funktionssymbols von $\Sigma' \setminus \Sigma$ und das Kopfprädikat jedes Axioms von $SP' \setminus SP$ zu $\Sigma' \setminus \Sigma$ gehören oder

(3) SP' terminierend und konsistent ist, die Zielsorte jedes Konstruktors von $\Sigma' \setminus \Sigma$ zu $\Sigma' \setminus \Sigma$ gehört und für alle Axiome $f(t)\{\equiv u\} \Leftarrow H$ von $SP' \setminus SP$, f zu $\Sigma' \setminus \Sigma$ gehört.

Beweis. (1) folgt aus folgender Äquivalenz. Für alle Σ-Hornformeln φ gilt:

$$Her(SP) \models \varphi \iff SP \cup \{\varphi\} \text{ ist konsistent bzgl. } SP.$$

Zunächst stellen wir fest, dass die Konsistenz von $SP \cup \{\varphi\}$ bzgl. SP äquivalent ist zu:

$$\forall\ \Sigma\text{-Grundatome } p: \quad SP \cup \{\varphi\} \vdash_{cut} p \ \Rightarrow\ SP \vdash_{cut} p, \tag{3}$$

während die Gültigkeit von einer Hornformel $q \Leftarrow G$ in $Her(SP)$ äquivalent ist zu:

$$\forall\ \sigma : X \to T_\Sigma: \quad SP \vdash_{cut} G\sigma \ \Rightarrow\ SP \vdash_{cut} q\sigma. \tag{4}$$

"$\Rightarrow$": Sei φ und p ein Σ-Grundatom mit $SP \cup \{\varphi\} \vdash_{cut} p$. Dann gibt es eine Ableitung minimaler Länge mit Axiomen und Regeln des Schnittkalküls für $SP \cup \{\varphi\}$, die mit p endet.

Fall 1. Es gibt $q \in SP \cup \{\varphi\}$ und $\sigma : X \to T_\Sigma$ mit $p = q\sigma$. $Her(SP) \models AX \cup \{\varphi\}$ und (5) implizieren $Her(SP) \models q$, also auch $Her(SP) \models p$. Wegen (5) folgt $SP \vdash_{cut} p$.

Fall 2. Es gibt $(q \Leftarrow q_1 \wedge \cdots \wedge q_n) \in SP \cup \{\varphi\}$ und $\sigma : X \to T_\Sigma$ mit $p = q\sigma$ derart, dass die o.g. Ableitung von p für alle $1 \le i \le n$ eine Ableitung von $q_i\sigma$ enthält. Diese Ableitungen sind kürzer als die Ableitung von p. Nach Induktionsvoraussetzung folgt $SP \vdash_{cut} q_i\sigma$, also auch $SP \vdash_{cut} q_1\sigma \wedge \cdots \wedge q_n\sigma$. $Her(SP) \models AX \cup \{\varphi\}$ und (5) implizieren $SP \vdash_{cut} q\sigma$, also $SP \vdash_{cut} p$.

Wegen (4) folgt in beiden Fällen die Konsistenz von $SP \cup \{\varphi\}$ bzgl. SP.

"$\Leftarrow$": Sei $\varphi = (q \Leftarrow q_1 \wedge \cdots \wedge q_n)$ und $\sigma : X \to T_\Sigma$ mit $Her(SP) \models q_1\sigma \wedge \cdots \wedge q_n\sigma$. Daraus folgt $Her(SP) \cup \{\varphi\} \models q_1\sigma \wedge \cdots \wedge q_n\sigma$, also $SP \cup \{\varphi\} \vdash_{cut} q_1\sigma \wedge \cdots \wedge q_n\sigma$ wegen (5). Die Ableitung von $q_1\sigma \wedge \cdots \wedge q_n\sigma$ lässt sich zu einer Ableitung von $q\sigma$ fortsetzen. Also gilt $SP \cup \{\varphi\} \vdash_{cut} q\sigma$. Da $SP \cup \{\varphi\}$ konsistent bzgl. SP ist, folgt $SP \vdash_{cut} q\sigma$ wegen (4).

Wegen (5) folgt die Gültigkeit von φ in $Her(SP)$.

(2): ☞ Übung.

(3): Sei p ein Σ-Grundatom mit $Her(SP')_\Sigma \models p$. Dann gilt $Her(SP') \models p$ nach Lemma 6.1.2(2), also $SP' \vdash_{cut} p$ nach Satz 4.2.7. Daraus folgt $p \vdash_{SP'} \mathit{True}$ nach Korollar 5.1.18. Die syntaktischen Bedingungen von (3) implizieren — durch Induktion über die Länge einer kürzesten erfolgreichen SP'-Reduktion $\mathcal{R}$ von p —, dass $\mathcal{R}$ eine SP-Reduktion ist. Damit gilt $SP \vdash_{cut} p$, also $Her(SP) \models p$. ❑

☞ Zeigen Sie, dass (2) und (3) auch implizieren, dass $Her(SP')$ vollständig bzgl. Σ ist!

Wir verbinden 6.1.5(3) mit Korollar 5.1.15 und Satz 5.2.9 und setzen dabei die in letzterem *verlangte* (konfluente) Teilspezifikation mit der jetzt *vorgegebenen* Teilspezifikation SP gleich:[34]

Korollar 6.1.6 (Konsistenzkriterium) Sei $SP' = (\Sigma', AX')$ eine Extension von $SP = (\Sigma, AX)$. SP' ist konsistent bzgl. SP, wenn SP konfluent ist, SP' stark terminierend ist,

[34] Bitte nicht davon verwirren lassen, dass in Satz 5.2.9 umgekehrt die Teilspezifikation mit SP und die gesamte Spezifikation mit SP' bezeichnet werden!

(1) die Zielsorte jedes Konstruktors von $\Sigma' \setminus \Sigma$ zu $\Sigma' \setminus \Sigma$ gehört,

(2) für alle Axiome $\varphi = (f(t)\{\equiv u\} \Leftarrow H)$ von $SP' \setminus SP$ f zu $SP' \setminus SP$ gehört,

(3) für alle bedingten Gleichungen φ, ψ von $AX' \setminus AX$ gilt:

(3.1) $\varphi = \psi$ und φ ist deterministisch oder

(3.2) jedes von φ und ψ erzeugte Overlay ist ein induktives Theorem von SP.

Baut man eine Spezifikation als Folge von Extensionen auf, dann ist Konsistenz bzgl. der jeweils darunterliegenden Ebene eine übliche Anforderung. Sie stellt insbesondere sicher, dass die Identität aller Daten eines initialen Modells $Ini(SP)$ bei der Extension von SP zu SP' erhalten bleibt: Seien t und u Σ-Grundterme.

$$t^{Ini(SP')} = u^{Ini(SP')} \quad \Leftrightarrow \quad Her(SP') \models t \equiv u \quad \overset{Konsistenz}{\Longrightarrow} \quad Her(SP) \models t \equiv u \quad \Leftrightarrow \quad t^{Ini(SP)} = u^{Ini(SP)}.$$

Vollständigkeit hingegen ist nicht immer erwünscht. Sie schließt nämlich aus, dass beim Übergang von SP nach SP' ein Datenbereich um neue Elemente erweitert wird. Im Gegensatz zu 6.1.5(2) und (3) können die Bedingungen des folgenden Satzes erfüllt sein, ohne dass SP' vollständig bzgl. Σ ist.

☞ Zeigen Sie, dass sowohl die Kriterien von 6.1.5(2) und als auch die von 6.1.5(3) implizieren, dass SP' vollständig bzgl. SP ist!

Satz 6.1.7 (Konsistenzkriterium) Sei $SP' = (\Sigma', AX')$ eine Extension von $SP = (\Sigma, AX)$. SP' ist konsistent bzgl. SP, wenn SP vollständig ist, SP' terminierend und konsistent ist,

(1) für alle Axiome $\varphi = (f(t)\{\equiv u\} \Leftarrow H)$ von $SP' \setminus SP$ f zu $SP' \setminus SP$ gehört,

(2) alle Axiome φ von SP (!) deterministisch sind und dabei alle in H vorkommenden Variablen zu V_n oder $\bigcup_{s\in\Sigma}\{X_s \mid NF_{\Sigma,s} = NF_{\Sigma',s}\}$ gehören, wobei H und V_n wie in Def. 5.2.8 definiert sind.

Beweis. Sei p ein Σ-Grundatom mit $Her(SP') \models p$. Nach Voraussetzung folgt $p \vdash_{SP'} True$ nach Korollar 5.1.18. Das folgende Lemma impliziert, dass jede erfolgreiche SP'-Reduktion von p bereits eine SP-Reduktion ist. Demnach gilt $p \vdash_{SP} True$, also $Her(SP) \models p$. ❑

Lemma 6.1.8 Es gelten die Voraussetzungen von Satz 6.1.7. Sei $\varphi = (f(t)\{\equiv u\} \Leftarrow G)$ ein Axiom von SP und σ eine Grundsubstitution mit $t\sigma \in T_\Sigma$, $G\sigma \vdash_{SP'} True$ und $(var(G) \setminus var(t))\sigma \subseteq NF_{\Sigma'}$. Dann ist $\varphi\sigma$ eine Σ-Formel.

Beweis. G hat die Form $t_1 \equiv u_1 \wedge \cdots \wedge t_n \equiv u_n \wedge H$, wobei die Bedingungen von Def. 5.2.8 und 6.1.7(2) gelten. Sei $1 \leq i \leq n$. Wegen $var(u_i) \subseteq var(G) \setminus var(t)$ und $(var(G) \setminus var(t))\sigma \subseteq NF_{\Sigma'}$ ist $u_i\sigma$ eine Σ'-Normalform. Daher folgt $t_i\sigma \xrightarrow{*}_{SP'} u_i\sigma$ aus $G\sigma \vdash_{SP'} True$. Durch Induktion über i zeigen wir, dass für alle $0 \leq i \leq n$, $V_i\sigma$ aus Σ-Termen besteht (s. Def. 5.2.8). $V_0\sigma \subseteq T_\Sigma$ folgt aus der Voraussetzung $t\sigma \in T_\Sigma$. Sei $i > 0$. Wegen $t_i \in T_\Sigma(X)$ und $var(t_i) \subseteq V_{i-1}$ liefert die Induktionsvoraussetzung $t_i\sigma \in T_\Sigma$. Da SP vollständig ist, gibt es eine Σ-Normalform v mit $t_i\sigma \equiv_{SP} v$. Deshalb und wegen $t_i\sigma \xrightarrow{*}_{SP'} u_i\sigma$ sind v und $u_i\sigma$ SP'-äquivalent. Da beide Terme Σ'-Normalformen sind und SP' konsistent ist, folgt $v = u_i\sigma$. Also ist $u_i\sigma$ ein Σ-Term. Damit besteht $V_i\sigma = V_{i-1}\sigma \cup var(u_i)$ aus Σ-Termen. Da alle frischen Variablen von φ zu $V_n \cup H$ und alle Variablen von H zu V_n oder $\bigcup_{s\in\Sigma}\{X_s \mid NF_{\Sigma,s} = NF_{\Sigma',s}\}$ gehören, folgt aus $V_n\sigma \subseteq T_\Sigma$ und $t\sigma \in T_\Sigma$, dass $\varphi\sigma$ eine Σ-Formel ist. ❑

Im Folgenden beschäftigen wir uns mit der Frage, wann $Her(SP)$ eine Σ-Struktur mit Komplementen ist (siehe Def. 4.1.2). Ein weiteres Konsistenzkriterium ist nämlich die Komplementabgeschlossenheit:

Komplementabgeschlossenheit

Definition 6.1.9 Eine Spezifikation $SP = (\Sigma, AX)$ ist **komplementabgeschlossen**, wenn die Komplementprädikate aller Prädikate von Σ zu Σ gehören und das Herbrandmodell von SP eine Σ-Struktur mit Komplementen ist.

Zeigen Sie, dass $Her(SP)$ genau dann eine Σ-Struktur mit Komplementen ist, wenn für alle Prädikate $r : w \in \Sigma$ mit Komplement und alle $t \in T_{\Sigma,w}$ gilt:

$$SP \vdash_{cut} \overline{r}(t) \iff SP \not\vdash_{cut} r(t). \tag{1}$$

Wie spezifiziert man Komplementprädikate? Sei $SP = (\Sigma, AX)$ eine Spezifikation, $r : w \in \Sigma$ ein Prädikat, $r(t_1) \Leftarrow \varphi_1, \ldots, r(t_n) \Leftarrow \varphi_n$ die Menge der Axiome für r und X_i die Menge der Variablen von $r(t_i) \Leftarrow \varphi_i$. Wegen der Fixpunkteigenschaft von $Her(SP)$ (siehe §4.2) ist die Σ-Formel

$$r(x) \iff \varphi_{r,AX}(x) \tag{2}$$

ein induktives, also in $Her(SP)$ gültiges, Theorem von SP. Nach Definition von $\varphi_{r,AX}$ (siehe §4.2) ist dann auch die Negation von (1):

$$\neg r(x) \iff \bigwedge_{i=1}^{n} \forall X_i : (\neg x \equiv t_i \vee \neg\varphi_i) \tag{3}$$

ein induktives Theorem von SP. So ergeben sich z.B. aus den Axiomen für $sorted : list$ die induktiven SORTED-Theoreme (siehe Bsp. 6.3.2):

$$sorted(L) \iff L \equiv [] \vee \exists x : L \equiv x : [] \vee \exists x, y, L' : (L \equiv x : y : L' \wedge x \leq y \wedge sorted(y : L')) \tag{4}$$

$$\neg sorted(L) \iff \neg L \equiv [] \wedge \forall x : \neg L \equiv x : [] \wedge \forall x, y, L' : (\neg L \equiv x : y : L' \vee \neg x \leq y \vee \neg sorted(y : L')) \tag{5}$$

Die rechte Seite von (5) ist äquivalent zu folgender Disjunktion:

$$\exists x, y, L' : (L \equiv x : y : L' \wedge x > y) \vee \exists x, y, L' : (L \equiv x : y : L' \wedge \neg sorted(y : L')). \tag{6}$$

Daraus lassen sich Hornaxiome für *unsorted*, das Komplement von *sorted*, bilden:

$$\begin{aligned} unsorted(x : y : L) &\Leftarrow x > y \\ unsorted(x : y : L) &\Leftarrow unsorted(y : L) \end{aligned}$$

Tatsächlich lassen sich auf diese Weise immer aus den Axiomen für ein logisches Prädikat r Axiome für sein Komplement $\overline{r}$ gewinnen, vorausgesetzt, die Negation von $\varphi_{r,AX}$ ist äquivalent zu einer Disjunktion existenzquantifizierter Goals:

Satz 6.1.10 (Komplementkriterium I) Zu jedem logischen Prädikat $r : w \in \Sigma$ gebe es in Σ ein logisches Prädikat $\overline{r} : w$ derart, daß $\varphi_{\overline{r},AX}$ logisch äquivalent zu $\neg\varphi_{r,AX}$ ist. Dann ist $\overline{r}^{Her(SP)}$ das relationale Komplement von $r^{Her(SP)}$.

Beweis. Sei Φ die von AX induzierte Schrittfunktion (siehe §4.2). Da $Her(SP)$ ein Fixpunkt von Φ ist, erhält man nach Definition von Φ und Voraussetzung:

$$\begin{aligned} \overline{r}^{Her(SP)} &= \overline{r}^{\Phi(Her(SP))} = \varphi^{Her(SP)}_{\overline{r},AX} = (\neg\varphi_{r,AX})^{Her(SP)} \\ T_{\Sigma,w} \setminus \varphi^{Her(SP)}_{r,AX} &= T_{\Sigma,w} \setminus r^{\Phi(Her(SP))} = T_{\Sigma,w} \setminus r^{Her(SP)}. \quad \square \end{aligned}$$

6.2 Coprädikate

Schön wäre es, wenn (eine normalisierte Form von) $\neg\varphi_{r,AX}$ direkt Axiome für $\overline{r}$ liefern würde. Dann entfiele die Voraussetzung von Satz 6.1.10 und Axiome für Komplementprädikate könnten automatisch aus den Axiomen für die ursprünglichen Prädikate konstruiert werden. (2) zeigt aber, dass die Negation von $\varphi_{r,AX}$ i.a. statt einer Disjunktion existenzquantifizierter Goals (= Konjunktionen von Atomen) eine Konjunktion allquantifizierter

Disjunktionen von (negierten) Atomen liefert. Das legt nahe, für die Axiomatisierung von Komplementprädikaten anstelle von Hornformeln einen "dualen" Formeltyp zu verwenden. Tatsächlich brauchen wir auf syntaktischer Ebene nur den Implikationspfeil einer Hornformel umdrehen und erhalten einen neuen Typ von Axiomen, der (nicht nur!) zur Spezifikation von Komplementprädikaten geeignet ist:

co-Hornformel, Spezifikation mit Coprädikaten

Definition 6.2.1 Seien Σ eine Signatur, $r(t)$ ein Σ-Atom und φ eine beliebige Σ-Formel. Dann nennen wir

$$r(t) \;\Rightarrow\; \varphi$$

eine **Σ-co-Hornformel für** r und r ein **Coprädikat**.

$\Sigma = (S, F, R)$ enthalte für alle $s \in S$ nicht nur das Gleichheitsprädikat $\equiv: ss$, sondern auch sein Komplement $\not\equiv: ss$. Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation (siehe Def. 5.1.2) und $coAX$ eine Menge von Σ-co-Hornformeln $r(t) \Rightarrow \varphi$ mit folgenden Eigenschaften:

(1) r kommt in AX nicht vor.

(2) t eine Normalform.

(3) Es gibt eine zu φ logisch äquivalente negations- und implikationsfreie Σ-Formel.

Dann ist $SP = (\Sigma, AX \cup coAX)$ eine **Spezifikation mit Coprädikaten**. Die Menge EQ_Σ der Kongruenzaxiome für Σ (s. §5.1) wird für jedes Coprädikat r um die co-Hornformel

$$r(x_1, \ldots, x_n) \;\Rightarrow\; ((x_1 \equiv y_1 \wedge \cdots \wedge x_n \equiv y_n) \;\Rightarrow\; r(y_1, \ldots, y_n))$$

erweitert. Eine Σ-Struktur, die AX und $coAX$ erfüllt, heisst **SP-Modell**.

Da wegen 6.2.1(1) kein Prädikat von Σ sowohl Hornaxiome als auch co-Hornaxiome in SP haben kann, nennen wir fortan nur noch diejenigen Relationssymbole von Σ Prädikate, deren Axiome Hornformeln sind.

Das Herbrandmodell einer Spezifikation $SP = (\Sigma, AX)$ mit Coprädikaten interpretiert diese nicht als kleinste, sondern als größte Lösung ihrer Axiome. Bedingung 6.2.1(3) garantiert die Monotonie der zugrundeliegenden Schrittfunktion Ψ auf Σ-Herbrandstrukturen (s.u.). Sei B eine Σ-Herbrandstruktur. Analog zu Prädikaten ist auch für Coprädikate r Ψ durch die **AX-Definition von** r, $\varphi_{r,AX}$, definiert (vgl. §4.2):

$$r^{\Phi(B)} \;=_{def}\; \varphi^B_{r,AX}.$$

Hier ist $\varphi_{r,AX}$ allerdings eine *Kon*junktion *all*quantifizierter Formeln: Seien $r(t_1) \Rightarrow \varphi_1, \ldots, r(t_n) \Rightarrow \varphi_n$ die co-Hornaxiome für r und X_i, $1 \le i \le n$, die Menge der Variablen von $r(t_i) \Rightarrow \varphi_i$, $1 \le i \le n$. Dann ist

$$\varphi_{r,AX} \;=_{def}\; \bigwedge_{i=1}^{n} \forall X_i : (x \equiv t_i \Rightarrow \varphi_i).$$

Damit ist sowohl für Prädikate als auch für Coprädikate r die Konjunktion der Axiome für r zur Implikation

$$r(x) \Leftarrow \varphi_{r,AX}(x) \quad \text{bzw.} \quad r(x) \Rightarrow \varphi_{r,AX}(x)$$

logisch äquivalent.

Grob gesagt interpretiert das für Spezifikationen mit Coprädikaten erweiterte Herbrandmodell Coprädikate nicht im kleinsten, sondern im größten Fixpunkt von Φ. Da man aber nicht Teile einer Signatur in einer Struktur und den Rest in einer anderen interpretieren kann, übernehmen wir zur Interpretation der Prädikate

das Herbrandmodell von Def. 4.2.6 und bilden zur Interpretation der Coprädikate eine zweite Schrittfunktion $\Psi : \mathcal{K} \to \mathcal{K}$ auf der Menge derjenigen Σ-Herbrandstrukturen, deren Redukt auf die Prädikate mit jenem Herbrandmodell übereinstimmt:

Herbrandmodell mit Coprädikaten

Definition 6.2.2 Sei $SP = (\Sigma, AX)$ eine Spezifikation mit Coprädikaten, coP die Menge der Coprädikate von Σ, $coAX$ die Menge der co-Hornformeln von AX und $\mathcal{K}$ die Menge der Σ-Herbrandstrukturen B mit

$$B|_{\Sigma \setminus coP} \;=\; Her(\Sigma \setminus coP, AX \setminus coAX).$$

Sei $\Psi : \mathcal{K} \to \mathcal{K}$ die **von** $coAX$ **induzierte Schrittfunktion**, d.h. für alle $B \in \mathcal{K}$ und $r \in coP$ ist

$$r^{\Psi(B)} \;=_{def}\; \varphi^{B}_{r,AX},$$

wobei $r(t_1) \Rightarrow \varphi_1, \ldots, r(t_n) \Rightarrow \varphi_n$ die co-Hornaxiome für r sind und X_i, $1 \leq i \leq n$, die Menge der Variablen von $r(t_i) \Rightarrow \varphi_i$, $1 \leq i \leq n$, ist.

Das **Herbrandmodell** von SP, **Her**(SP), ist der größte Fixpunkt von Ψ.

Demnach ist eine konstruktorbasierte Spezifikation im Sinne von Def. 5.1.2 syntaktisch wie semantisch eine Spezifikation mit Coprädikaten, wobei die Menge der Coprädikate leer ist. Dementsprechend verwenden wir im Folgenden den Begriff "konstruktorbasierte Spezifikation" auch für Spezifikationen mit Coprädikaten.

☞ Zeigen Sie, dass Bedingung 6.2.1(3) die Monotonie von Ψ impliziert und deshalb nach Satz 4.2.10 einen größten Fixpunkt hat!

In Def. 4.2.6 wurde das Herbrandmodell über den Schnittkalkül eingeführt: Für Prädikate r gilt ein Grundatom $r(t)$ genau dann in $Her(SP)$, wenn $r(t)$ aus den Axiomen von SP ableitbar ist. Gibt es eine entsprechende Charakterisierung von in $Her(SP)$ gültigen Grundatomen $r(t)$, wenn r ein Coprädikat ist? Ja, wenn der Schnittkalkül so erweitert wird, dass auch (die fast beliebigen) Konklusionen von co-Hornformeln abgeleitet werden können (vgl. Def. 4.2.5):

erweiterter Schnittkalkül

Definition 6.2.3 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation. Der **(erweiterte) Schnittkalkül für** SP besteht aus den folgenden Regeln zur Ableitung von Σ-Formeln:

Axiome	$\dfrac{True}{\varphi} \Downarrow$ für alle $\varphi \in AX \cup EQ_\Sigma$
Instanziierung	$\dfrac{\varphi}{\varphi[t/x]} \Downarrow$ für alle $t \in T_\Sigma(X)_{sort(x)}$
Modus Ponens	$\dfrac{\varphi, \quad \varphi \Rightarrow \psi}{\psi} \Downarrow$
$\wedge$-Einführung	$\dfrac{\varphi, \quad \psi}{\varphi \wedge \psi} \Downarrow$
$\wedge$-Eliminierung	$\dfrac{\varphi \wedge \psi}{\varphi} \Downarrow \qquad \dfrac{\varphi \wedge \psi}{\psi} \Downarrow$
$\forall$-Eliminierung	$\dfrac{\forall x : \varphi}{\varphi} \Downarrow$
$\exists$-Eliminierung	$\dfrac{\exists x : \varphi}{\bigvee\{\varphi[t/x] \mid t \in T_{\Sigma,sort(x)}\}} \Downarrow$

$\vdash_{cut}$ bezeichnet wieder die Inferenzrelation des Schnittkalküls.

Satz 6.2.4 Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation, P die Menge der Prädikate und coP die Menge der Coprädikate von Σ und A die folgendermaßen definierte Σ-Herbrandstruktur:

- Für alle Prädikate $r \in \Sigma$, $r^A =_{def} \{t \in T_\Sigma \mid \mathit{True} \vdash_{cut} r(t)\}$.
- Für alle Coprädikate $r \in \Sigma$, $r^A =_{def} \{t \in T_\Sigma \mid r(t) \nvdash_{cut} \mathit{False}\}$.

A stimmt mit $Her(SP)$ überein.

Beweis. Sei A ein SP-Modell.

Sei r ein Prädikat von Σ. $r^A = r^{Her(SP)}$ ergibt sich aus der Definition des Herbrandmodells in §4.2. Sei r ein Coprädikat von Σ. r^A ist in $r^{Her(SP)}$ enthalten, weil der (erweiterte) Schnittkalkül für SP bzgl. jedes SP-Modells, also auch bzgl. A, korrekt ist, d.h. $r(t) \vdash_{cut} \mathit{False}$ impliziert $A \models r(t) \Rightarrow \mathit{False}$, also $A \models \neg r(t)$. Umgekehrt ist $r^{Her(SP)}$ in r^A enthalten, weil $Her(SP)$ das bzgl. der Größe der Interpretation von r größte SP-Modell ist.

Es bleibt also zu zeigen, dass A die Horn- und co-Hornaxiome von SP erfüllt. Sei $r(t) \Leftarrow \varphi \in AX$ und es gelte $A \models_\sigma \varphi$ für ein $\sigma : X \to T_\Sigma$. Nach Lemma 6.2.5(1) (s.u.) folgt $\mathit{True} \vdash_{cut} \varphi\sigma$, also unter Verwendung von Instanziierung und Modus ponens auch $\mathit{True} \vdash_{cut} r(t)\sigma$, d.h. $A \models_\sigma r(t)$ nach Definition von r^A. Also gilt $r(t) \Leftarrow \varphi$ in A. Sei $r(t) \Rightarrow \varphi \in AX$ und es gelte $A \models r(t)\sigma$ für ein $\sigma : X \to T_\Sigma$, also $r(t)\sigma \nvdash_{cut} \mathit{False}$ nach Definition von r^A. Unter Verwendung von Instanziierung und Modus ponens folgt $\varphi\sigma \nvdash_{cut} \mathit{False}$, d.h. $A \models \varphi\sigma$ nach Lemma 6.2.5(2) (s.u.). Also gilt $r(t) \Rightarrow \varphi$ in A. ❑

Lemma 6.2.5 Sei A wie in Satz 6.2.4 definiert. Dann gilt für alle Grundsubstitutionen $\sigma : X \to T_\Sigma$ und alle möglichen Prämissen φ von Hornaxiomen bzw. Konklusionen ψ von co-Hornaxiomen,

(1) $A \models \varphi\sigma$ impliziert $\mathit{True} \vdash_{cut} \varphi\sigma$,
(2) $A \not\models \psi\sigma$ impliziert $\psi\sigma \vdash_{cut} \mathit{False}$. ❑

☞ Zeigen Sie Lemma 6.2.5 durch Induktion über den Aufbau von φ bzw. ψ!

☞ Die Regeln des erweiterten Schnittkalküls wurden gerade so gewählt, dass Lemma 6.2.5 gilt. Welche weiteren Regeln bräuchte man, damit Lemma 6.2.5 auch für verallgemeinerte Hornaxiome $r(t) \Leftarrow \varphi$ gilt, bei denen φ eine beliebige negations- und implikationsfreie Σ-Formel ist?

☞ Ein Hornaxiom $r(t) \Leftarrow \varphi$ ist **nicht-rekursiv**, wenn φ weder r noch ein von r abhängiges Prädikat enthält. Angenommen, alle Axiome für r sind nichtrekursiv. Wie lauten dann co-Hornaxiome für r, so dass die Interpretationen von r als Prädikat bzw. Coprädikat im Herbrandmodell übereinstimmen?

Zurück zu den Komplementprädikaten. Die Negation der AX-Definition eines Prädikates r liefert co-Hornaxiome für $\overline{r}$. So ergeben sich aus (5) die folgenden Axiome für *unsorted*:

$$\begin{aligned} &\mathit{unsorted}([]) \;\Rightarrow\; \mathit{False} \\ &\mathit{unsorted}([x]) \;\Rightarrow\; \mathit{False} \\ &\mathit{unsorted}(x : y : L) \;\Rightarrow\; (x \leq y \;\Rightarrow\; \mathit{unsorted}(y : L)). \end{aligned}$$

Die letzte Implikation erfüllt nur deshalb die Bedingung 6.2.1(3) an eine co-Hornformel, weil (bei geeigneter Wahl der zugrundeliegenden Spezifikation) ihre Konklusion zur negations- und implikationsfreien Formel $x > y \vee \mathit{unsorted}(y : L)$ äquivalent ist.

Satz 6.2.6 (Komplementkriterium II) Sei $SP = (\Sigma, AX)$ eine konstruktorbasierte Spezifikation, $r : w$ ein Prädikat und $\overline{r} : w$ ein Coprädikat $\overline{r} : w$.

$$\begin{array}{lcll} r(t_1) & \Leftarrow & \varphi_1 \wedge q_1(u_1) & \qquad r(v_1) \\ & \ldots & & \qquad \ldots \\ r(t_n) & \Leftarrow & \varphi_n \wedge q_n(u_n) & \qquad r(v_k) \end{array}$$

seien die Axiome für r.

$$\begin{array}{lcll} \overline{r}(t_1) & \Rightarrow & (\varphi_1 \Rightarrow \overline{q_1}(u_1)) & \qquad \overline{r}(v_1) \Rightarrow \mathit{False} \\ & \ldots & & \qquad \ldots \\ \overline{r}(t_n) & \Rightarrow & (\varphi_n \Rightarrow \overline{q_n}(u_n)) & \qquad \overline{r}(v_k) \Rightarrow \mathit{False} \end{array}$$

seien die Axiome für $\overline{r}$. Dann ist $\overline{r}^{Her(SP)}$ das relationale Komplement von $r^{Her(SP)}$.

Beweis. Seien AX und $coAX$ die Mengen der Hornaxiome bzw. co-Hornaxiome von SP und Φ und Ψ die von AX bzw. $coAX$ induzierten Schrittfunktionen (siehe §4.2 bzw. Def. 6.2.2). Da $\varphi_{\overline{r},coAX}$ logisch äquivalent zu $\neg\varphi_{r,AX}$ und $Her(SP)$ ein Fixpunkt von Φ und Ψ ist, erhält man nach Definition von Φ bzw. Ψ:

$$\begin{array}{l} \overline{r}^{Her(SP)} = \overline{r}^{\Psi(Her(SP))} = \varphi^{Her(SP)}_{\overline{r},coAX} = (\neg\varphi_{r,AX})^{Her(SP)} \\ T_{\Sigma,w} \setminus \varphi^{Her(SP)}_{r,AX} = T_{\Sigma,w} \setminus r^{\Phi(Her(SP))} = T_{\Sigma,w} \setminus r^{Her(SP)}. \quad \square \end{array}$$

☞ Komplementieren Sie die Prädikate *exists* und *forall* aus Beispiel 6.3.2 gemäß Satz 6.2.6!

Satz 6.1.10 muss noch ergänzt werden um Bedingungen, unter denen auch Gleichheitsprädikate Komplemente haben. Wie zu erwarten, sind mal wieder Konsistenz und Vollständigkeit (im Sinne von Def. 5.1.8) die geeigneten Bedingungen:

Satz 6.2.7 (Komplementkriterium III) Sei $SP = (\Sigma, AX)$ eine funktionale Spezifikation. Für alle Sorten $s \in \Sigma$ enthalte Σ das Prädikat $\not\equiv$: ss. Sind die folgenden Hornformeln die einzigen Axiome von SP, in denen $\not\equiv$ vorkommt, dann ist $\not\equiv^{Her(SP)}$ das relationale Komplement von $\equiv^{Her(SP)}$.

$$c(x_1, \ldots, x_n) \not\equiv c(y_1, \ldots, y_n) \Leftarrow x_i \not\equiv y_i \qquad \text{für alle } n\text{-stelligen Konstruktoren } c \in \Sigma \text{ und } 1 \leq i \leq n \quad (1)$$

$$c(x) \not\equiv d(y) \qquad \text{für je zwei verschiedene Konstruktoren } c, d \in \Sigma \quad (2)$$

Beweis. Zu zeigen ist: Für alle $t, t' \in T_\Sigma$ gilt:

$$SP \vdash_{cut} t \not\equiv t' \iff SP \not\vdash_{cut} t \equiv t'. \quad (3)$$

"⇒": Wir wenden Satz 4.2.8 an und benutzen dazu die Σ-Herbrandstruktur A, die bis auf die folgende Interpretation von $\not\equiv$ mit $Her(SP)$ übereinstimmt:

$$t \not\equiv^A t' \iff_{def} SP \not\vdash_{cut} t \equiv t'.$$

Ist A ein Modell von AX, dann gilt die ⇒-Richtung von (3) nach Satz 4.2.8❷ . Da (1) und (2) die einzigen Axiome von SP sind, in denen $\not\equiv$ vorkommt, genügt es zu zeigen, dass A genau diese Axiome erfüllt.

Sei $c : s_1 \ldots s_n \to s$ ein Konstruktor von Σ, $t_k, t'_k \in T_{\Sigma,s_k}$ für alle $1 \leq k \leq n$ derart, dass für ein $1 \leq i \leq n$ $A \models t_i \not\equiv t'_i$, also $SP \not\vdash_{cut} t_i \equiv t'_i$ gilt. Da SP funktional ist, sind die eindeutigen Normalformen von t_i bzw. t'_i verschieden. Wären die Terme $c(t_1, \ldots, t_n)$ und $c(t'_1, \ldots, t'_n)$ SP-äquivalent, dann wären auch $c(nf(t_1), \ldots, nf(t_n))$ und $c(nf(t'_1), \ldots, nf(t'_n))$ SP-äquivalent, also identisch, weil diese beiden Terme Normalformen sind. Das widerspricht aber der Ungleichheit von $nf(t_i)$ und $nf(t'_i)$. Damit ist gezeigt, dass A alle Axiome der Form (1) erfüllt. Zeigen Sie analog, dass A alle Formeln (2) erfüllt.

"⇐": Sei $SP \not\vdash_{cut} t \equiv t'$. Dann sind die eindeutigen Normalformen $nf(t)$ und $nf(t')$ von t bzw. t' verschieden und lassen sich wie folgt zerlegen. Es gibt Normalformen u, v, v', zwei verschiedene Konstruktoren c und d sowie $x \in var(u)$ mit $nf(t) = u[c(v)/x]$ und $nf(t') = u[d(v')/x]$. Man sieht sofort (oder beweist durch Induktion über die Größe von u), dass sich die Ungleichung $u[c(v)/x] \not\equiv u[d(v')/x]$ mit dem Schnittkalkül aus den Axiomen (1) und (2) ableiten läßt. Da Terme und ihre jeweiligen Normalformen SP-äquivalent sind, folgt sofort $SP \vdash_{cut} t \not\equiv t'$. ❑

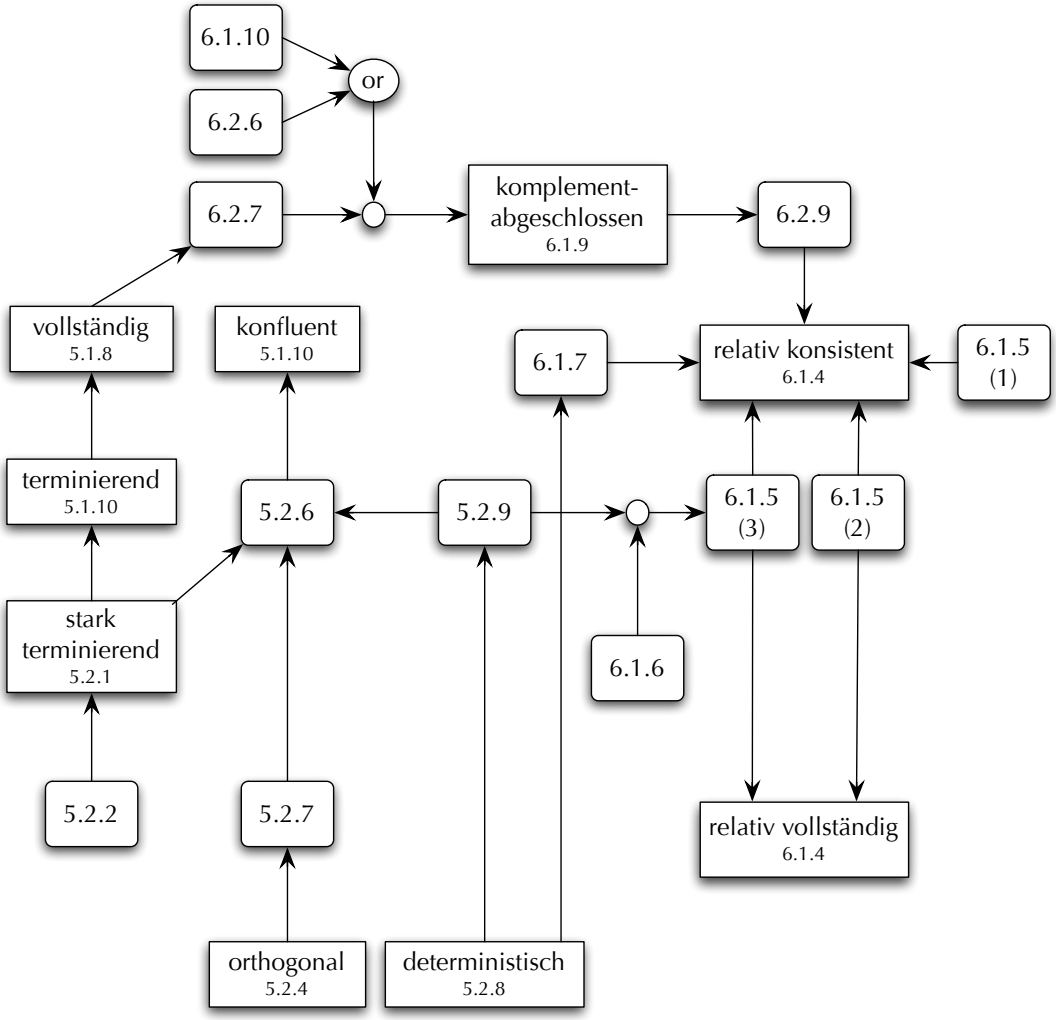


Figure 4. *Vollständigkeits-, Konfluenz- und relative Konsistenzkriterien*

Auch der Monotoniebegriff (siehe Def. 6.1.4) muss an Spezifikationen mit Coprädikaten angepasst werden:

Definition 6.2.8 Seien $SP = (\Sigma, AX)$ und $SP' = (\Sigma', AX')$ konstruktorbasierte Spezifikationen und $\sigma : \Sigma \to \Sigma'$ ein Signaturmorphismus. SP' ist **monoton bzgl.** (SP, σ), wenn

- für alle Prädikate r von Σ alle in $Her(SP)$ gültigen Grundatome auch in $Her(SP')_\sigma$ gelten,
- für alle Coprädikate r von Σ alle in $Her(SP)$ ungültigen Grundatome auch nicht in $Her(SP')_\sigma$ gelten.

Der Konsistenzbegriff von Def. 6.1.4 muss nicht geändert werden und die auf Def. 6.1.4 folgenden Behauptungen (A), (B) und (C) bleiben gültig.

☞ Zeigen Sie, dass (B) gültig bleibt!

Satz 6.2.9 (Konsistenzkriterium) Seien $SP = (\Sigma, AX)$ und $SP' = (\Sigma', AX')$ Spezifikationen und $\sigma : \Sigma \to \Sigma'$ ein Signaturmorphismus derart, dass SP und SP' komplementabgeschlossen sind und σ Komplemente erhält, d.h. für alle Prädikate $r \in \Sigma$ gilt: $\sigma(\overline{r}) = \overline{\sigma(r)}$.

SP' ist konsistent bzgl. (SP, σ), wenn $Her(SP')_\sigma$ ein SP-Modell ist.

Beweis. Sei $r(t)$ ein Σ-Grundatom. Wir müssen zeigen:

(1) $Her(SP')_\sigma \models r(t) \quad \Rightarrow \quad Her(SP) \models r(t)$, falls r ein Prädikat von Σ ist,
(2) $Her(SP) \models r(t) \quad \Rightarrow \quad Her(SP')_\sigma \models r(t)$, falls r ein Coprädikat von Σ ist.

(1) Sei r ein Prädikat und $Her(SP) \not\models r(t)$. Dann gilt $Her(SP) \models \overline{r}(t)$. Da $Her(SP')_\sigma$ nach Vor. eine Lösung von $EQ_\Sigma \cup AX$ ist, $Her(SP)$ aber die kleinste dieser Lösungen, folgt $Her(SP')_\sigma \models \overline{r}(t)$, also

$$Her(SP') \models \sigma(\overline{r}(t)) = \sigma(\overline{r})(\sigma(t)) = \overline{\sigma(r)}(\sigma(t)).$$

Daraus folgt $Her(SP') \not\models \sigma(r)(\sigma(t)) = \sigma(r(t))$, d.h. $Her(SP')_\sigma \not\models r(t)$.

(2) Sei r ein Coprädikat und $Her(SP')_\sigma \not\models r(t)$, also $Her(SP') \not\models \sigma(r(t)) = \sigma(r)(\sigma(t))$. Dann gilt

$$Her(SP') \models \overline{\sigma(r)}(\sigma(t)) = \sigma(\overline{r})(\sigma(t)) = \sigma(\overline{r}(t)),$$

also $Her(SP')_\sigma \models \overline{r}(t)$. Da $Her(SP')_\sigma$ nach Vor. eine Lösung von $EQ_\Sigma \cup AX$ ist, $Her(SP)$ aber die größte dieser Lösungen, folgt $Her(SP) \models \overline{r}(t)$, also $Her(SP) \not\models r(t)$. ❏

6.3 Parametrisierung

Extensionen erlauben den schrittweisen hierarchischen Aufbau von Spezifikationen. Bestimmte Sorten, Funktionen oder Prädikate können in mehreren Modulen durchaus unter verschiedenen Namen verwendet werden, da sich mithilfe von Signaturmorphismen bereits auf syntaktischer Ebene präzisieren läßt, welche Komponenten einer Signatur denen einer anderen entsprechen sollen. Jeder als Spezifikation beschriebene Modul hat seine eindeutige Semantik (initiales oder Herbrandmodell). Bildet ein Signaturmorphismus die Signatur Σ auf die Signatur Σ' ab, dann werden Σ'-Strukturen durch σ-Redukte und σ-Restriktionen als Σ-Strukturen interpretiert (s. Def. 6.1.1).

Signaturmorphismen sind auch das formale Mittel, um *Parameterspezifikationen* zu *aktualisieren* und damit solche programmiersprachlichen Konzepte wie **Polymorphie**, **Parametrisierung** und **Generizität** im Spezifikations-Kontext zu realisieren. Im Gegensatz zu den bisher betrachteten Spezifikationen beschreibt eine Parameterspezifikation keinen *Entwurf*, sondern nur eine Reihe von *Anforderungen* an mögliche Entwürfe, die den möglichen Aktualisierungen des Parameters entsprechen. Dementsprechend sind wir bei den syntaktischen Bedingungen an eine Parameterspezifikation sehr liberal. Der Signaturbegriffbleibt derselbe. Die Axiome einer Parameterspezifikation können aber beliebige prädikatenlogische Formeln über der Parametersignatur sein.

parametrisierte Spezifikation

Definition 6.3.1 Eine **(formale) Parameterspezifikation** $PAR = (P\Sigma, PAX)[SP_1, \ldots, SP_k]$ besteht aus einer Menge $P\Sigma$ von Signaturelementen, einer Menge PAX prädikatenlogischer Formeln und $k \geq 0$ konstruktorbasierten Spezifikationen $SP_i = (\Sigma_i, AX_i)$, $1 \leq i \leq k$, den **konstanten Teilspezifikationen** von PAR derart, dass

$$\Sigma(PAR) =_{def} P\Sigma \cup \bigcup_{i=1}^{k} \Sigma_i$$

eine Signatur und

$$AX(PAR) \ =_{def} \ PAX \cup \bigcup_{i=1}^{k} AX_i$$

eine Menge von $\Sigma(PAR)$-Formeln ist.

Eine $\Sigma(PAR)$-Struktur A mit Gleichheit ist ein **Parametermodell** von PAR, wenn sie $AX(PAR)$ erfüllt und für alle $1 \leq i \leq k$, das Σ_i-Redukt von A isomorph zum initialen Modell von SP_i ist (siehe Def. 5.1.6).

Eine **parametrisierte Spezifikation** $PSP = (\Sigma, AX)[PAR_1, \ldots, PAR_n]$ besteht aus einer Menge Σ von Signaturelementen, einer Menge AX von Hornformeln und $n \geq 0$ Parameterspezifikationen $PAR_1, \ldots, PAR_n$ derart, dass

$$\Sigma(PSP) \ =_{def} \ \Sigma \cup \bigcup_{i=1}^{n} \Sigma(PAR_i)$$

die Signatur und AX die Axiomenmenge einer konstruktorbasierten Spezifikation ist.

Für alle $1 \leq i \leq n$ sei A_i ein Parametermodell von PAR_i. Die **semantische Aktualisierung von** PSP **durch** $A_1, \ldots, A_n$ ist die konstruktorbasierte Spezifikation $(\Sigma, AX)[A_1, \ldots, A_n]$, deren Signatur aus Σ und allen Elementen von $A_1 \cup \cdots \cup A_n$, aufgefasst als Konstanten (nullstellige Konstruktoren), und deren Axiomenmenge aus AX und allen Gleichungen $f(a) \equiv f^{A_i}(a)$ und logischen Atomen $r(b)$ mit $f, r \in \Sigma(PAR_i)$, $a \in A_i$ und $b \in r^{A_i}$ für ein $1 \leq i \leq n$ besteht.

Ein **induktives Theorem von** PSP ist ein induktives Theorem aller semantischen Aktualisierungen von PSP (siehe §4.2).

PSP ist **(relativ) konsistent** bzgl. einer konstruktorbasierten Spezifikation SP, wenn alle Aktualisierungen von PSP konsistent bzgl. SP sind (siehe Def. 6.1.4).

Insbesondere ist eine konstruktorbasierte Spezifikation $SP = (\Sigma, AX)$ eine parametrisierte Spezifikation ohne Parameterspezifikationen. In diesem Fall ist $\Sigma(PSP) = \Sigma$ und $AX(PSP) = AX$.

In Def. 6.3.1 können $(P\Sigma, PAX)$ oder (Σ, AX) Symbole enthalten, die in $P\Sigma$ bzw. Σ nicht deklariert sind, so dass dies keine kompletten Signaturen im Sinne von Def. 3.5 sind. Das werden sie erst durch Hinzunahme der Signaturen der konstanten Teilspezifikationen von PAR bzw. der Parameterspezifikationen von PSP. So entstehen gerade die Signaturen $\Sigma(PAR)$ bzw. $\Sigma(PSP)$.

Bei der *syntaktischen* Aktualisierung von PSP mit einem aktuellen Parameter werden die formalen Parameter von PSP durch parametrisierte Spezifikationen ersetzt (siehe Def. 6.3.4).

Beispiel 6.3.2 Die folgende konstruktorbasierte Spezifikation stellt Boolesche Werte und aussagenlogischer Verknüpfungen zur Verfügung:

BOOL

`sorts`	$bool$
`constructs`	$true, false :\rightarrow bool$
`defuncts`	$not : bool \rightarrow bool$
	$and, or : bool \times bool \rightarrow bool$
`vars`	$b : bool$
`axioms`	$not(false) \equiv true$
	$not(true) \equiv false$
	$false \ and \ b \equiv false$

$$true\ and\ b \equiv b$$
$$false\ or\ b \equiv b$$
$$true\ or\ b \equiv true$$

Die einfachste Parameterspezifikation besteht aus einer Sorte mit Gleichheitsprädikat und deren Komplement:

NEQ(s)

`sorts`	s
`preds`	$\equiv, \not\equiv: s \times s$
`axioms`	$x \not\equiv y \iff \neg(x \equiv y)$

NEQ(s) wird zu einer Parameterspezifikation mit konstanter Teilspezifikation erweitert:[35]

TRIV(s)[BOOL] `where` TRIV(s) = NEQ(s) `and`

`defuncts`	$eq, neq : s \times s \to bool$
`Horn axioms`	$eq(x,y) \equiv true \Leftarrow x \equiv y$
	$eq(x,y) \equiv false \Leftarrow x \not\equiv y$
	$neq(x,y) \equiv true \Leftarrow x \not\equiv y$
	$neq(x,y) \equiv false \Leftarrow x \equiv y$

Laut Def. 6.3.1 ist jedes Modell A von TRIV(s)[BOOL] mit Gleichheit und $A|_{\Sigma(\mathrm{BOOL})} \cong Ini(\mathrm{BOOL})$ ein Parametermodell von TRIV(s)[BOOL].

TRIV(s) wird zu einer Parameterspezifikation mit Ordnungsrelationen auf s erweitert:

ORD(s)[BOOL] `where` ORD(s) = TRIV(s) `and`

`functs`	$min, max : s \times s \to s$
`preds`	$\_ < \_, \_ > \_, \_ \leq \_, \_ \geq \_ : s \times s$
`vars`	$x, y : s$
`axioms`	$min(x,y) \equiv x \Leftarrow x \leq y$
	$min(x,y) \equiv y \Leftarrow y \leq x$
	$max(x,y) \equiv x \Leftarrow y \leq x$
	$max(x,y) \equiv y \Leftarrow x \leq y$
	$x < y \iff y > x$
	$x \leq y \iff \neg(x > y)$
	$x \geq y \iff \neg(x < y)$

Die folgende parametrisierte Spezifikation verwendet TRIV(s)[BOOL] als Parameterspezifikation und *list* als neuen Sortenkonstruktor:[36]

LIST[TRIV(s)[BOOL]] `where` LIST =

`sorts`	$list(s)$
`constructs`	$[] :\to list(s)$
	$\_ : \_ : s \times list(s) \to list(s)$
	$\lambda y.not(eq(x,y)) : s \to (s \to bool)$

[35] Der Operator **and** bezeichnet die komponentenweise Vereinigung von (partiellen) Spezifikationen. Er ist der Spezifikationssprache CASL [17] entlehnt. **where** kapselt wie in Haskell lokale Definitionen, hier von Spezifikationen.

[36] Einige Sortenkonstruktoren haben eine feste Bedeutung (siehe §6.6).

`defuncts` $[\_] : s \to list(s)$
$\_ +\!\!+ \_ : list(s) \times list(s) \to list(s)$
$reverse : list(s) \to list(s)$
$exists, forall : (s \to bool) \times list(s) \to bool$
$eq : list(s) \times list(s) \to bool$
$filter : (s \to bool) \times list(s) \to list(s)$
$remove : s \times list(s) \to list(s)$
$apply : ((s \to bool) \times s) \to bool$

`preds` $\_ \in \_ : s \times list(s)$
$\_ \notin \_ : s \times list(s)$

`vars` $x, y : s \quad L, L' : list(s) \quad f : s \to bool$

`Horn axioms`
$[x] \equiv x : []$
$[] +\!\!+ L \equiv L$
$(x : L) +\!\!+ L' \equiv x : (L +\!\!+ L')$
$reverse([]) \equiv []$
$reverse(x : L) \equiv reverse(L) +\!\!+ [x]$
$exists(f, []) \equiv false$
$exists(f, x : L) \equiv f(x)\ or\ exists(f, L)$
$forall(f, []) \equiv true$
$forall(f, x : L) \equiv f(x)\ and\ forall(f, L)$
$eq([], []) \equiv true$
$eq([], x : L) \equiv false$
$eq(x : L, []) \equiv false$
$eq(x : L, y : L') \ \equiv\ eq(x, y)\ and\ eq(L, L')$
$filter(f, []) \equiv []$
$filter(f, x : L) \equiv x : filter(f, L) \ \Leftarrow\ f(x) \equiv true$
$filter(f, x : L) \equiv filter(f, L) \ \Leftarrow\ f(x) \equiv false$
$remove(x, L) \equiv filter(\lambda y.not(eq(x, y)), L)$ (*)
$apply(\lambda y.not(eq(x, y)), y) \equiv not(eq(x, y))$ (*)
$x \in y : L \ \Leftarrow\ x \equiv y \vee x \in L$
$x \notin []$
$x \notin y : L \ \Leftarrow\ x \not\equiv y \wedge x \notin L$

Die folgende parametrisierte Spezifikation erweitert LIST[TRIV(s)[BOOL]] zu einer Spezifikation geordneter Listen:

SORTED[ORD(s)[BOOL]] `where` SORTED = LIST `and`

`preds` $sorted : list(s)$

`Horn axioms`
$sorted([])$
$sorted(x : [])$
$sorted(x : y : L) \ \Leftarrow\ x \leq y \wedge sorted(y : L)$

Die folgende Erweiterung von LIST[TRIV(s)[BOOL]] führt Funktionen zwischen verschiedenen Listentypen ein:

MAPLIST[TRIV(s)[BOOL],TRIV(s')[BOOL]] `where` MAPLIST = LIST `and`

`defuncts` $map : (s \to s') \times list(s) \to list(s')$
$concatMap : (s \to list(s')) \times list(s) \to list(s')$

`vars`	$x : s \quad L : list(s) \quad f : s \to s' \quad g : s \to list(s')$
`Horn axioms`	$map(f, []) \equiv []$
	$map(f, x : L) \equiv f(x) : map(f, L)$
	$concatMap(g, []) \equiv []$
	$concatMap(g, x : L) \equiv g(x) +\!\!+ concatMap(g, L)$

Eine weitere parametrisierte Spezifikation mit mehreren Kopien derselben Parameterspezifikation:

FUNCOMP[NEQ(s),NEQ(s'),NEQ(s'')] `where` FUNCOMP =

`constructs`	$\_ \circ \_ : (s' \to s'') \times (s \to s') \to (s \to s'')$
`defuncts`	$apply : ((s \to s'') \times s) \to s''$
`vars`	$x : s \quad f : s \to s' \quad g : s' \to s''$
`Horn axioms`	$apply(g \circ f, x) \equiv g(f(x))$ 🐰 (*)

Bis auf *sorted* entsprechen alle Funktionen und Prädikate von Beispiel 6.3.2 Standardfunktionen von Haskell, wobei die Prädikate als Boolesche Funktionen implementiert sind.

☞ Programmieren Sie die definierten Funktionen und Prädikate von SORTED[ORD(s)[BOOL]] in Haskell!

Besondere Beachtung verdienen die gesternten Zeilen, wo Konstruktoren funktionaler Sorten benutzt werden. Den hier als Konstruktorkonstante deklarierten Ausdruck $\lambda y.not(eq(x, y))$ bezeichnet man als **λ-Abstraktion**. λ-Abstraktionen sind Konstruktoren zur Bildung funktionaler Objekte. Zur Auswertung von λ-Abstraktionen, die Funktionen des Typs $s \to s'$ darstellen, benötigt man die Funktion $apply : (s \to s') \times s \to s'$, die solche Funktionen auf Argumente der Sorte s anwendet und die jeweiligen Funktionswerte der Sorte s' liefert. Anstelle eines Terms $apply(t, u)$ schreibt man in der Regel $t(u)$. Ist t eine λ-Abstraktion, dann nennt man $t(u)$ eine **λ-Applikation**. Mehr zu λ-Termen in §6.5.

Sei $PSP = SP[PAR_1, \ldots, PAR_n]$ eine parametrisierte Spezifikation. Nach den Definitionen 4.2.6 und 6.3.1 ist eine $\Sigma(PSP)$-Formel φ ein induktives Theorem von PSP, wenn für alle $1 \leq i \leq n$ und alle Parametermodelle A_i von PAR_i

$$Her(SP[A_1, \ldots, A_n]) \models \varphi$$

gilt. Als Verallgemeinerung von Satz 4.2.7 ergibt sich daraus das folgende Kriterium zum Beweis induktiver Theoreme von PSP:

Lemma 6.3.3 Sei $SP = (\Sigma, AX)$, $PSP = SP[PAR_1, \ldots, PAR_n]$ eine parametrisierte Spezifikation, CAX die Menge der Axiome konstanter Teilspezifikationen von $PAR_1 \cup \cdots \cup PAR_n$ und Σ' die Signatur einer beliebigen semantischen Aktualisierung von PSP. Eine $\Sigma(PSP)$-Formel φ ein induktives Theorem von PSP, wenn für alle $\sigma : X \to T_{\Sigma'}$ $\varphi\sigma$ ein induktives Theorem der konstruktorbasierten Spezifikation $SP' =_{def} (\Sigma', CAX \cup AX)$ ist. ❑

Σ' enthält also Elemente von Parametermodellen. Deren Eigenschaften können im Beweis von φ aber nicht verwendet werden, weil die entprechenden Axiome $f(a) \equiv f^{A_i}(a)$ bzw. $r(a)$ (siehe Def. 6.3.1) in SP' fehlen. Nur die Axiome konstanter Teilspezifikationen formaler Parameter (CAX) sind verfügbar. Obwohl der Beweis von φ in *einer* semantischen Aktualisierung von PSP geführt wird, können nur Eigenschaften benutzt werden, die in *jeder* Aktualisierung gelten. Prinzipiell gilt das zwar für alle Axiome von PSP, aber nur die von AX und CAX sind in jedem Fall Hornformeln und lassen sich daher im Rahmen einer Schnittkalkül- oder Narrowing-Ableitung anwenden.

Spezifikationen sollen unabhängig voneinander entworfen werden können und nur über ihre Schnittstellen Daten, Funktionen und Prädikate austauschen. Folglich weichen die Signaturen aktueller Parameter häufig von der Signatur des formalen Parameters ab. Der Zuordnung aktueller zu formalen Parametern dienen Signaturmorphismen (siehe Def. 6.1.1). Sie beschreiben die Aktualisierung syntaktisch:

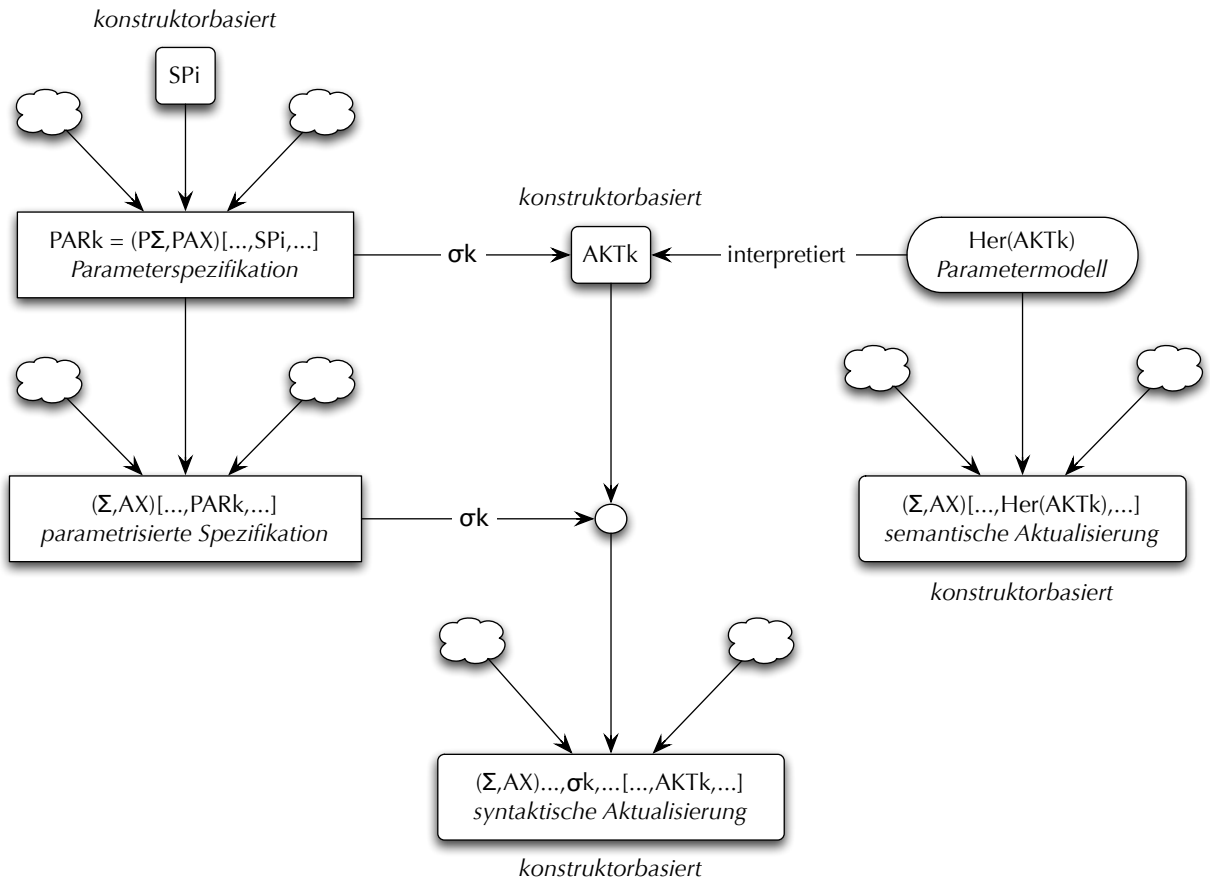


Figure 5. *Parametrisierung und Aktualisierung durch konstruktorbasierte Parameter*

syntaktische Aktualisierung

Definition 6.3.4 Seien $SP = (\Sigma, AX)$ und

$$PSP = SP[PAR_1, \ldots, PAR_k, PAR_{k+1}, \ldots, PAR_n],\ PSP_1,\ \ldots,\ PSP_k$$

parametrisierte Spezifikationen. Für alle $1 \leq i \leq k$ sei $P\Sigma_i$ die Menge der Signatursymbole von PAR_i, die nicht zu einer konstanten Teilspezifikation von PAR_i gehören, und $\sigma_i : P\Sigma_i \to \Sigma(PSP_i)$ ein Signaturmorphismus. Die parametrisierte Spezifikation

$$SP_{\sigma_1,\ldots,\sigma_k}[PSP_1, \ldots, PSP_k][PAR_{k+1}, \ldots, PAR_n] \ =_{def}\ (\Sigma', AX')[PAR_{k+1}, \ldots, PAR_n]$$

mit

$$\Sigma' \ =_{def}\ \bigcup_{i=1}^{k}(\Sigma(PSP_i) \cup \sigma_i(\Sigma)) \ \text{ und } \ AX' \ =_{def}\ \bigcup_{i=1}^{k}(AX(PSP_i) \cup \sigma_i(AX))$$

heißt **Verklebung** (*amalgamation*) von PSP mit $PSP_1 \ldots, PSP_k$ oder **syntaktische Aktualisierung** von PSP durch $PSP_1, \ldots, PSP_k$ entlang $\sigma_1 \ldots, \sigma_k$, wobei σ_i, $1 \leq i \leq k$, folgendermaßen auf Σ fortgesetzt wird:

- $\sigma_i(s) = s$ für alle unstrukturierten Sorten $s \in \Sigma$,
- $\sigma_i(s(s_1, \ldots, s_n)) = s(\sigma_i(s_1), \ldots, \sigma_i(s_n))$ für alle strukturierten Sorten $s \in \Sigma$,
- $\sigma_i(f : w \to s) = f : \sigma_i(w) \to \sigma_i(s)$ für alle Funktionssymbole $f \in \Sigma$,

- $\sigma_i(r : w) = r : \sigma_i(w)$ für alle Prädikate $r \in \Sigma$. ❑

Beschränkt man die Modelle aktueller Parameterspezifikationen auf Herbrandmodelle, stellt sich die Frage, ob das Herbrandmodell einer syntaktischen Aktualisierung mit semantischen Aktualisierung durch die Herbrandmodelle der aktueller Parameterspezifikationen übereinstimmt. Wir betrachten den Fall, dass $k = n$ ist und $PSP_1, \ldots, PSP_n$ und damit auch die Verklebung von PSP mit $PSP_1, \ldots, PSP_n$ unparametrisierte, also konstruktorbasierte, Spezifikationen sind:

Satz 6.3.5 (Äquivalenz syntaktischer und semantischer Aktualisierung) Seien

$$PSP \;=\; SP[PAR_1, \ldots, PAR_n]$$

eine parametrisierte Spezifikation, $SP_1, \ldots, SP_n$ konstruktorbasierte Spezifikationen und $\sigma_1, \ldots, \sigma_n$ Signaturmorphismen wie in Def. 6.3.4 derart, dass für alle $1 \leq i \leq n$ $Her(SP_i)|_{\sigma_i}$ ein Parametermodell von PAR_i ist. Dann gilt:

$$Her(SP[Her(SP_1)|_{\sigma_1}, \ldots, Her(SP_n)|_{\sigma_n}]) \;\cong\; Her(SP_{\sigma_1,\ldots,\sigma_n}[SP_1, \ldots, SP_n]). \quad ❑$$

6.4 Refinement

Zurück zu den in §6.1 eingeführten Begriffen. In objektorientierter Terminologie sind Extensionen einer Spezifikation SP **Erben** von SP. Neue Axiome in SP' für eine Funktion $f \in SP$ entsprechen einer **Redefinition** von f. Sie können dazu führen, dass SP' nicht konsistent bzgl. SP ist (s. Def. 6.1.4).

Ähnlich der Vererbung dienen Extensionen und Signaturmorphismen auch dazu, einen Softwareentwurf **vertikal zu strukturieren**. Man spricht dann von einer **Verfeinerung (Refinement)** oder **abstrakten Implementierung**. Eine "abstrakte" Spezifikation $SP = (\Sigma, AX)$ wird durch eine "konkrete" $SP' = (\Sigma', AX')$ verfeinert, wenn es einen Signaturmorphismus $rep : \Sigma \to \Sigma'$ gibt, der zwei Bedingungen erfüllt:

Verfeinerung

Definition 6.4.1 Seien $SP = (\Sigma, AX)$ und $SP' = (\Sigma', AX')$ Spezifikationen und $rep : \Sigma \to \Sigma'$ ein Signaturmorphismus, genannt **Repräsentationsmorphismus**. SP' ist eine **Verfeinerung von** SP **entlang** rep, wenn

(1) das **Refinement-Modell** $Her(SP')_{rep}$ ein SP-Modell ist,

(2) SP' konsistent bzgl. (SP, rep) ist.

Die zweite Bedingung verhindert insbesondere, dass in SP unterschiedene Daten im Refinement-Modell gleichgesetzt werden. Sie gilt genau dann, wenn es einen monotonen Σ-Homomorphismus

$$Her(SP')_{rep} \quad \xrightarrow{abs} \quad Ini(SP) = Her(SP)/\equiv_{SP}$$

gibt. *abs* heißt **Abstraktionshomomorphismus** der Verfeinerung. Damit folgt aus dem Homomorphiesatz 4.1.4(1), dass (2) genau dann gilt, wenn der Quotient des Refinement-Modells $Her(SP')_{rep}$ nach dem Äquivalenzkern von *abs* isomorph zum initialen Modell von SP ist, kurz:

$$Her(SP')_{rep}/\sim_{abs} \;\cong\; Ini(SP)$$

(siehe Def. 4.1.2). $\sim_{abs}$ wird **Gleichheitsinterpretation** der Verfeinerung genannt, weil die Isomorphie zwischen $Her(SP')_{rep}/\sim$ und $Ini(SP)$ bedeutet, dass die Gleichheit im abstrakten Modell $Ini(SP)$ der Äquivalenz $\sim_{abs}$ im Refinement-Modell entspricht.

☞ Folgern Sie die logische Äquivalenz zwischen Bedingung 6.4.1(2) und der Existenz von *abs* aus Lemma 4.1.9!

☞ Zeigen Sie, dass *abs* surjektiv ist!

Zusammengefaßt stimmt also das Herbrandmodell von SP' **modulo** der Symbolzuordnung *rep*, **modulo** der Restriktion auf Σ-erzeugbare Daten und **modulo** dem Äquivalenzkern von *abs* mit dem initialen Modell von SP überein.

Das Konsistenzkriterium 6.2.9 liefert sofort ein Kriterium für Verfeinerungen:

Korollar 6.4.2 (Refinementkriterium) Seien SP, SP' und *rep* wie in Def. 6.4.1. Sind SP und SP' komplementabgeschlossen, gilt $rep(\overline{r}) = \overline{rep(r)}$ für alle Prädikate $r \in \Sigma$ und ist $Her(SP')_{rep}$ ein SP-Modell, dann ist SP' eine Verfeinerung von SP entlang *rep*. ❑

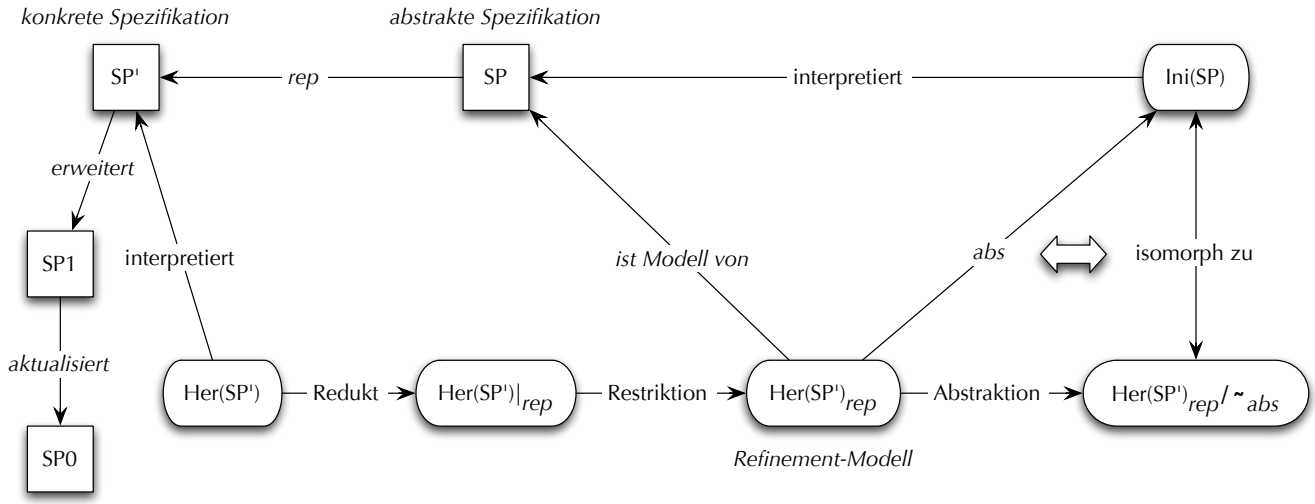


Figure 6. *Syntax und Semantik von Verfeinerungen*

Beispiel 6.4.3 (MAP implementiert STACK) Die folgende Verfeinerung im Sinne von Def. 6.4.1 dient oft als Benchmark für formale Refinement-Ansätze verwendet. Keller werden implementiert als Paare, bestehend aus einem Feld (= Funktion mit endlichem Definitionsbereich) und einem Zeiger auf das zuletzt eingetragene Element. Die Kelleroperationen werden entsprechend verfeinert. Wir beginnen mit einer parametrisierten Version der Spezifikation STACK aus Beispiel 5.1.4, die das Komplement der *stack*-Gleichheit enthält (siehe Beispiel 6.3.2):

SP = STACK[NEQ(*entry*)] `where` STACK =

`sorts`	$stack(entry)$
`constructs`	$() :\to 1 + entry$
	$just : entry \to 1 + entry$
	$empty :\to stack(entry)$
	$push : entry \times stack(entry) \to stack(entry)$
`defuncts`	$top : stack(entry) \to 1 + entry$
	$pop : stack(entry) \to stack(entry)$
`preds`	$\not\equiv : stack(entry) \times stack(entry)$
`vars`	$x : entry \quad s : stack(entry)$
`Horn axioms`	$top(empty) \equiv ()$
	$pop(empty) \equiv empty$

$top(push(x,s)) \equiv just(x)$
$pop(push(x,s)) \equiv s$
Axiome für $\not\equiv$ gemäß Satz 6.2.7

MAP[NEQ($index$),NEQ($entry$)] `where` MAP $=$

`sorts`	$map(index, entry)$	
`constructs`	$index : index \to index + entry$	
	$entry : entry \to index + entry$	
	$new :\to map(index, entry)$	
	$upd : index \times entry \times map(index, entry) \to map(index, entry)$	*"update"*
`defuncts`	$get : map(index, entry) \times index \to index + entry$	
`preds`	$\not\equiv : map(index, entry) \times map(index, entry)$	
	$\not\equiv : (index + entry) \times (index + entry)$	
`vars`	$i, j : index \quad x : entry \quad f : map(index, entry)$	
`axioms`	$get(new, i) \equiv index(i)$	
	$get(upd(i, x, f), i) \equiv entry(x)$	
	$get(upd(i, x, f), j) \equiv get(f, j) \quad \Leftarrow \quad i \not\equiv j$	
	Axiome für $\not\equiv$ gemäß Satz 6.2.7	

Die Herbrandmodelle der Aktualisierungen von MAP[NEQ($index$),NEQ($entry$)] liefern leider keine *eindeutigen* Repräsentationen von Feldern. Das ist für die hier behandelte Verwendung von MAP als Verfeinerung aber auch irrelevant. Zu einer adäquateren Modellierung von Feldern vgl. §6.4.

Der Repräsentationsmorphismus rep bildet alle SP-Symbole bis auf die folgenden auf sich selbst ab:

$$\begin{aligned} rep(1) &= nat, \\ rep(stack(entry)) &= map(nat, entry) \times nat, \\ rep(\equiv : stack(entry) \times stack(entry)) &= \sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat), \\ rep(\not\equiv : stack(entry \times stack(entry)) &= \not\sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat). \end{aligned}$$

Es folgt die Verfeinerung als Extension einer Aktualisierung des ersten Parameters von MAP[NEQ($index$),NEQ($entry$)] durch NAT (siehe Def. 6.3.4):

$SP' =$ MAP$_{nat/index}$[NAT][NEQ($entry$)] `and`

`defuncts`	$() :\to nat + entry$	
	$just : entry \to nat + entry$	
	$empty :\to map(nat, entry) \times nat$	
	$push : entry \times (map(nat, entry) \times nat) \to map(nat, entry) \times nat$	
	$top : map(nat, entry) \times nat \to nat + entry$	
	$pop : map(nat, entry) \times nat \to map(nat, entry) \times nat$	
`preds`	$\not\sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat)$	
`copreds`	$\sim : (map(nat, entry) \times nat) \times (map(nat, entry) \times nat)$	
`vars`	$i, j : nat \quad x : entry \quad f : map(nat, entry) \quad s, s' : map(nat, entry) \times nat$	
`axioms`	$() \equiv index(0)$	
	$just(x) \equiv entry(x)$	
	$empty \equiv (new, 0)$	
	$push(x, s) \equiv (upd(suc(i), x, f), suc(i)) \quad \Leftarrow \quad s \equiv (f, i)$	
	$top(f, i) \equiv get(f, i)$	
	$pop(f, i) \equiv (f, i - 1)$	
	$s \sim s' \quad \Rightarrow \quad top(s) \equiv top(s') \wedge pop(s) \sim pop(s')$	

$$s \not\sim s' \quad \Leftarrow \quad top(s) \not\equiv top(s')$$
$$s \not\sim s' \quad \Leftarrow \quad pop(s) \not\sim pop(s')$$

SP und SP' sind orthogonal und terminierend, also nach Kor. 5.2.7 konfluent. Da beide Spezifikationen vollständig sind, sind sie nach Kor. 5.1.15 auch konsistent, also funktional. Demzufolge sind nach Satz 6.2.7 $\not\equiv^{Her(SP)}$ und $\not\equiv^{Her(SP')}$ die Komplemente von $\equiv^{Her(SP)}$ bzw. $\equiv^{Her(SP')}$. Die duale Version des Satzes 6.2.6, mit deren Hilfe man Komplemente von Coprädikaten als Prädikate spezifizieren kann, liefert die Komplementeigenschaft von $\not\sim^{Her(SP')}$ bzgl. $\sim^{Her(SP')}$.

Nach Lemma 6.1.2(2) und Korollar 6.4.2 ist SP' eine Verfeinerung von SP entlang rep, wenn die folgenden rep-Bilder der Axiome von SP induktive Theoreme von SP' sind:

$$top(empty) \equiv () \tag{1}$$
$$top(push(x,s)) \equiv just(x) \tag{2}$$
$$pop(empty) \sim empty \tag{3}$$
$$pop(push(x,s)) \sim s \tag{4}$$
$$empty \not\sim push(x,s) \tag{5}$$
$$push(x,s) \not\sim empty \tag{6}$$
$$push(x,s) \not\sim push(y,s') \quad \Leftarrow \quad x \not\equiv y \tag{7}$$
$$push(x,s) \not\sim push(y,s') \quad \Leftarrow \quad s \not\sim s' \tag{8}$$
$$s \sim s \tag{9}$$
$$s \sim s' \quad \Leftarrow \quad s' \sim s \tag{10}$$
$$s \sim s'' \quad \Leftarrow \quad s \sim s' \wedge s' \sim s'' \tag{11}$$
$$push(x,s) \sim push(x,s') \quad \Leftarrow \quad s \sim s' \tag{12}$$
$$top(s) \equiv top(s') \quad \Leftarrow \quad s \sim s' \tag{13}$$
$$pop(s) \sim pop(s') \quad \Leftarrow \quad s \sim s' \tag{14}$$

Da die Relation $\sim$ im Herbrandmodell von SP' als größte Lösung ihres Axioms interpretiert wird und der Äquivalenzabschluss von $\sim^{Her(SP')}$ dieses Axiom ebenfalls erfüllt, gelten die Formeln (9)-(11) in $Her(SP')$.

☞ Führen Sie dieses Argument näher aus!

Die Konjunktion von (13) und (14) ist zum Axiom für $\sim$ äquivalent. Es bleiben also (1)-(8) und (12) zu zeigen. Die folgenden Narrowing-Ableitungen von (1)-(4) und (12) wurden wieder mit *Expander2* [81] erstellt. Die Ableitung von (4) verwendet das Lemma

$$upd(i,x,f),j) \sim (f,j) \quad \Leftarrow \quad i > j, \tag{15}$$

dessen Beweis wir in Beispiel 7.3.1 führen werden.

```
top(empty) = nothing                              (1)

Narrowing the preceding formula leads to

top((new,0)) = nothing

Narrowing the preceding formula leads to

top((new,0)) = index(0)

Simplifying the preceding formula leads to

top(new,0) = index(0)
```

```
Narrowing the preceding formula leads to

get(new,0) = index(0)

Narrowing the preceding formula leads to

index(0) = index(0)

Simplifying the preceding formula leads to

True
--------------------------------------------------

top(push(x,s)) = just(x)                          (2)

Narrowing the preceding formula leads to

Any f i:(top((upd(suc(i),x,f),suc(i))) = just(x) & s = (f,i))

Narrowing the preceding formula leads to

Any f i:(top((upd(suc(i),x,f),suc(i))) = entry(x) & s = (f,i))

Simplifying the preceding formula leads to

Any f i:(top(upd(suc(i),x,f),suc(i)) = entry(x) & s = (f,i))

Narrowing the preceding formula leads to

Any f i:(get(upd(suc(i),x,f),suc(i)) = entry(x) & s = (f,i))

Narrowing the preceding formula leads to

Any f i:((entry(x) = entry(x) | get(f,suc(i)) = entry(x) & suc(i) =/= suc(i)) &
         s = (f,i))

Simplifying the preceding formula (7 steps) leads to

Any f i:(s = (f,i))

Applying the theorem

True ===> Any f i:(s = (f,i))

at position [] of the preceding formula leads to

All s0:(Any f2 i2:(s0 = (f2,i2))) ==> Any f i:(s = (f,i))

Simplifying the preceding formula leads to

True
--------------------------------------------------

pop(empty) ~ empty                                (3)

Applying the axiom resp. theorem

empty = (new,0)

at positions [1],[0,0] of the preceding formula leads to

pop((new,0)) ~ (new,0)
```

```
Simplifying the preceding formula leads to

pop(new,0) ~ (new,0)

Applying the axiom resp. theorem

pop(f,i) = (f,pred(i))

at position [0] of the preceding formula leads to

(new,pred(0)) ~ (new,0)

Applying the axiom resp. theorem

pred(0) = 0

at position [0,1] of the preceding formula leads to

(new,0) ~ (new,0)

Simplifying the preceding formula leads to
                                -- Anwendung von (9)
True
--------------------------------------------------

pop(push(x,s)) ~ s                              (4)

Applying the axiom resp. theorem

push(x,(f,i)) = (upd(suc(i),x,f),suc(i))

at position [0,0] of the preceding formula leads to

Any f i:(pop((upd(suc(i),x,f),suc(i))) ~ (f,i) & s = (f,i))

Simplifying the preceding formula leads to

Any f i:(pop(upd(suc(i),x,f),suc(i)) ~ (f,i) & s = (f,i))

Applying the axiom resp. theorem

pop(f,i) = (f,pred(i))

at position [0,0,0] of the preceding formula leads to

Any f i:((upd(suc(i),x,f),pred(suc(i))) ~ (f,i) & s = (f,i))

Applying the axiom resp. theorem

pred(suc(i)) = i

at position [0,0,0,1] of the preceding formula leads to

Any f i:((upd(suc(i),x,f),i) ~ (f,i) & s = (f,i))

Applying the theorem

(upd(i,x,f),j) ~ (f,j) <=== i > j

at position [0,0] of the preceding formula leads to
```

```
Any f i:(suc(i) > i & s = (f,i))

Applying the axiom resp. theorem

suc(i) > i

at position [0,0] of the preceding formula leads to

Any f i:(True & s = (f,i))

Simplifying the preceding formula leads to

Any f i:(s = (f,i))

Applying the theorem

True ===> Any f i:(s = (f,i))

at position [] of the preceding formula leads to

All s0:(Any f2 i4:(s0 = (f2,i4))) ==> Any f i:(s = (f,i))

Simplifying the preceding formula leads to

True
----------------------------------------------------

s ~ s' ==> push(x,s) ~ push(x,s')               (12)

Applying the axiom resp. theorem

s ~ s' ===> top(s) = top(s') & pop(s) ~ pop(s')

at position [1] of the preceding formula leads to

(s ~ s' ==> top(push(x,s)) = top(push(x,s'))) &
(s ~ s' ==> pop(push(x,s)) ~ pop(push(x,s')))

The reducts have been simplified.

Applying the axiom resp. theorem

top(push(x,s)) = entry(x)                       (2)

at position [0,1,0] of the preceding formula leads to

(s ~ s' ==> entry(x) = top(push(x,s'))) &
(s ~ s' ==> pop(push(x,s)) ~ pop(push(x,s')))

The reducts have been simplified.

Applying the axiom resp. theorem

top(push(x,s)) = entry(x)

at position [0,1,1] of the preceding formula leads to

s ~ s' ==> pop(push(x,s)) ~ pop(push(x,s'))

The reducts have been simplified.

A transitivity axiom at position [1] of the preceding formula leads to
```

```
                                                    -- Anwendung von (11)
s ~ s' ==> Any z0:(pop(push(x,s)) ~ z0 & z0 ~ pop(push(x,s')))

Applying the theorem

pop(push(x,s)) ~ s                                  (4)

at position [1,0,0] of the preceding formula leads to

s ~ s' ==> s ~ pop(push(x,s'))

The reducts have been simplified.

A transitivity axiom at position [1] of the preceding formula leads to
                                                    -- Anwendung von (11)
s ~ s' ==> Any z1:(s ~ z1 & z1 ~ pop(push(x,s')))

Applying the theorem

pop(push(x,s)) ~ s                                  (4)

at position [1,0,1] of the preceding formula leads to

True
```

☞ Zeigen Sie (5)-(11)!

☞ Wie ist der Abstraktionshomomorphismus dieser Verfeinerung definiert (siehe Def. 6.4.1)? ✌

Beispiel 6.4.4 (STACK implementiert SYMTAB) Auch das folgende ist ein wiederholt zur Illustration von Verfeinerungsmethoden verwendetes Beispiel. Hier geht es um die Implementierung von Symboltabellen, oder allgemeiner: blockstruktierten Abbildungen, durch Keller, deren Elemente Felder im Sinne von MAP (s.o.) sind. Im Unterschied zu Bsp. 6.4.3 besteht hier die Abstraktion von der implementierenden hin zur implementierten Spezifikation nicht in der Gleichsetzung von Daten, sondern in einer echten Restriktion des implementierenden Datenbereichs. Zumindest eins seiner Elemente, nämlich der leere Keller, implementiert nichts. Formal ausgedrückt, $Her(SP')_{rep}$ ist hier eine echte Unterstruktur von $Her(SP')|_{rep}$ (siehe Def. 6.4.1, 6.1.1 und 4.1.2). Zunächst die (parametrisierte) Symboltabellenspezifikation:

SP = SYMTAB[NEQ($index$),NEQ($entry$)] **where** SYMTAB =

sorts	$symtab(index, entry)$	
constructs	$() :\rightarrow 1 + entry$	
	$just : entry \rightarrow 1 + entry$	
	$init :\rightarrow symtab(index, entry)$	
	$enter : symtab(index, entry) \rightarrow symtab(index, entry)$	
	$add : index \times entry \times symtab(index, entry) \rightarrow symtab(index, entry)$	
defuncts	$leave : symtab(index, entry) \rightarrow symtab(index, entry)$	
	$retrieve : symtab(index, entry) \times index \rightarrow 1 + entry$	
preds	$\not\equiv : symtab(index, entry) \times symtab(index, entry)$	
vars	$i, j : index \quad x : entry \quad s : symtab(index, entry)$	
Horn axioms	$leave(init) \equiv init$	(A)
	$leave(enter(s)) \equiv s$	(B)
	$leave(add(i, x, s)) \equiv leave(s)$	
	$retrieve(init, i) \equiv ()$	
	$retrieve(enter(s), i) \equiv retrieve(s, i)$	
	$retrieve(add(i, x, s), i) \equiv just(x)$	

$$retrieve(add(i,x,s),j) \equiv retrieve(s,j) \quad \Leftarrow \quad i \not\equiv j$$
Axiome für $\not\equiv$ gemäß Satz 6.2.7

Es folgt die Verfeinerung als Extension der Aktualisierung (des Parameters NEQ($entry$)) von STACK[NEQ($entry$)] durch MAP(NEQ($index$),NEQ($entry$)) (siehe Beispiel 6.4.3):

SP' = STACK$_{map(index,entry)/entry}$[MAP(NEQ($index$),NEQ($entry$))] `and`

`defuncts`	$init :\rightarrow stack(map(index,entry))$
	$enter : stack(map(index,entry)) \rightarrow stack(map(index,entry))$
	$add : index \times entry \times stack \rightarrow stack(map(index,entry))$
	$leave : stack(map(index,entry)) \rightarrow stack(map(index,entry))$
	$retrieve : stack(map(index,entry)) \times index \rightarrow 1 + entry$
`preds`	$\not\equiv: stack(map(index,entry)) \times stack(map(index,entry))$
`vars`	$i,j : index \quad x : entry \quad f : map(index,entry) \quad s : stack(map(index,entry))$
`axioms`	$init \equiv push(new,empty)$
	$enter(s) \equiv push(new,s)$
	$add(i,x,empty) \equiv empty$
	$add(i,x,push(f,s)) \equiv push(upd(i,x,f),s)$
	$leave(empty) \equiv empty$
	$leave(push(f,empty)) \equiv push(f,empty)$ (C)
	$leave(push(f,push(g,s))) \equiv push(g,s)$
	$retrieve(empty,i) \equiv ()$
	$retrieve(push(new,s),i) \equiv retrieve(s,i)$
	$retrieve(push(upd(i,x,f),s),i) \equiv just(x)$
	$retrieve(push(upd(i,x,f),s),j) \equiv retrieve(push(f,s),j) \quad \Leftarrow \quad i \not\equiv j$
	Axiome für $\not\equiv$ gemäß Satz 6.2.7

Wie im Beispiel 6.4.3 sind SP und SP' sind orthogonal und terminierend, also nach Kor. 5.2.7 konfluent. Da beide Spezifikationen vollständig sind, sind sie nach Kor. 5.1.15 auch konsistent, also funktional. Demzufolge sind nach Satz 6.2.7 $\not\equiv^{Her(SP)}$ und $\not\equiv^{Her(SP')}$ die Komplemente von $\equiv^{Her(SP)}$ bzw. $\equiv^{Her(SP')}$.

Der Repräsentationsmorphismus rep bildet $symtab$ auf $stack$ und alle anderen SP-Symbole auf sich selbst ab. Im Gegensatz zu Beispiel 6.4.3 ist hier das Refinement-Modell $Her(SP')_{rep}$ eine echte Restriktion des rep-Reduktes von $Her(SP')$: Jeder nur aus Funktionen von SP zusammengesetzte $stack$-Term ist SP'-äquivalent zu einer Normalform der Gestalt $push(f,s)$, repräsentiert also einen nichtleeren Keller. Damit ist nach Lemma 6.1.2(2) und Korollar 6.4.2 SP' eine Verfeinerung von SP entlang rep, wenn alle mit rep übersetzten Instanzen von SYMTAB-Axiomen, die $stack$-Variablen durch Terme der Form $push(f,s)$ substituieren, induktive Theoreme von SP' sind.

☞ Zeigen Sie, dass $Her(SP')$ Axiom (A) erfüllt! Zeigen Sie, dass $Her(SP')$ Axiom (A) nicht erfüllt, wenn man Axiom (C) durch

$$leave(push(f,empty)) \equiv empty$$

ersetzt!

☞ Welche Terme von $Her(SP')$ enthält $Her(SP')_{rep}$ nicht?

☞ Zeigen Sie mithilfe von Lemma 6.1.2(2), dass $Her(SP')_{rep}$, aber nicht $Her(SP')$, Axiom (B) erfüllt!

☞ Zeigen Sie mithilfe von Lemma 6.1.2(2), dass $Her(SP')_{rep}$ alle Axiome von SP erfüllt! 🐰

Beispiel 6.4.5 (MAP implementiert HEAP) Ein Heap ist ein binärer Baum, der sich so als Feld darstellen läßt, dass alle Knoteneinträge auf benachbarte Feldelemente abgebildet werden. Die folgende Heaps-

pezifikation stellt neben den Binärbaumkonstruktoren die Funktion *heapify* zur Verfügung, die einen Heap in einen strukturgleichen, aber partiell geordneten Heap umwandelt (vgl. Beispiel 7.4.4):

SP = HEAP[ORD($entry$)] `where` HEAP =

`sorts`	$bintree(entry)$
`constructs`	$mt :\rightarrow bintree(entry)$
	$\_\#\_\#\_ : bintree(entry) \times entry \times bintree(entry) \rightarrow bintree(entry)$
`defuncts`	$heapify, sift : bintree(entry) \rightarrow bintree(entry)$
`vars`	$x, y, z : entry \quad T1, T2, T3, T4 : bintree(entry)$
`Horn axioms`	$heapify(mt) \equiv mt$
	$heapify(T1\#x\#T2) \equiv sift(heapify(T1)\#x\#heapify(T2))$
	$sift(mt) \equiv mt$
	$sift(mt\#x\#mt) \equiv mt\#x\#mt$
	$sift((mt\#y\#mt)\#x\#mt) \equiv (mt\#y\#mt)\#x\#mt \quad \Leftarrow \quad x \leq y$
	$sift((m\#y\#mt)\#x\#mt) \equiv (mt\#x\#mt)\#y\#mt \quad \Leftarrow \quad x > y$
	$sift((T1\#y\#T2)\#x\#(T3\#z\#T4)) \equiv (T1\#y\#T2)\#x\#(T3\#z\#T4)$
	$\Leftarrow \quad x \leq y \wedge x \leq z$
	$sift((T1\#y\#T2)\#x\#(T3\#z\#T4)) \equiv sift(T1\#x\#T2)\#y\#(T3\#z\#T4)$
	$\Leftarrow \quad x > y \wedge y \leq z$
	$sift((T1\#y\#T2)\#x\#(T3\#z\#T4)) \equiv (T1\#y\#T2)\#z\#sift(T3\#x\#T4)$
	$\Leftarrow \quad x > z \wedge y > z$
	In den restlichen Fällen ist das Argument von *sift* kein Heap:
	$sift(mt\#x\#(T1\#y\#T2)) \equiv mt$
	$sift(((T1\#x\#T2)\#y\#T3)\#z\#mt) \equiv mt$
	$sift((T1\#x\#(T2\#y\#T3))\#z\#mt) \equiv mt$

sift läßt den Eintrag x der Wurzel eines Heaps so lange in diesen "einsinken", bis er auf einen Knoten trifft, dessen direkter Nachfolger größer als x sind. *heapify*(T) wendet *sift* auf alle Teilbäume von T an und macht T so zum partiell geordneten Baum.

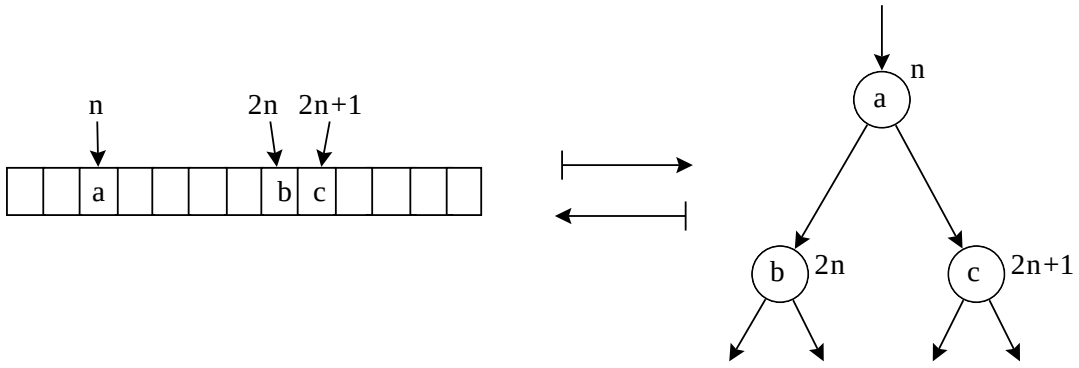


Figure 7. *Vom Feld zum Heap und zurück.*

Die Refinementspezifikation implementiert zunächst die Heapkonstruktoren durch Feldfunktionen. Die definierten Funktionen *heapify* und *sift* werden dann – modulo dieser Datenstrukturtransformation – durch die gleichen Algorithmen implementiert, die schon in HEAP zur Spezifikation von *heapify* bzw. *sift* verwendet wurden. Dazu benötigen wir noch die Rückübersetzung *heap2map* von Heaps in Felder.

SP' = $\text{MAP}_{nat/index}$[NAT][ORD($entry$)] `and`

defuncts $mt : (\to nat, entry)map(nat, entry)$
$\_\#\_\#\_ : map(nat, entry) \times entry \times map(nat, entry) \to map(nat, entry)$
$heapify, sift : map(nat, entry) \to map(nat, entry)$
$shift : (nat \to nat) \times map(nat, entry) \to map(nat, entry)$
$\_ + \_ : map(nat, entry) \times map(nat, entry) \to map(nat, entry)$
$g_1, g_2, h_1, h_2 : nat \to nat$

vars $i, j, k : nat \quad x, y, z : entry \quad f, f_1, f_2, g : map(nat, entry) \quad h : nat \to nat$

axioms
$mt \equiv new$
$f_1\#x\#f_2 \equiv upd(1, x, shift(g_1, f_1) + shift(g_2, f_2))$
$shift(h, new) \equiv new$
$shift(h, upd(i, x, f)) \equiv upd(h(i), x, shift(h, f))$
$new + f \equiv f$
$upd(i, x, f) + g \equiv upd(i, x, f + g)$
$g_1(k) \equiv k + 2^i \quad \Leftarrow \quad k \equiv 2^i + j \ \wedge \ \forall r, s : (k \equiv 2^r + s \Rightarrow r \leq i)$[37]
$g_2(k) \equiv k + 2^{i+1} \quad \Leftarrow \quad k \equiv 2^i + j \ \wedge \ \forall r, s : (k \equiv 2^r + s \Rightarrow r \leq i)$
$heapify(new) \equiv new$
$heapify(upd(i, x, f)) \equiv sift(heapify(shift(h_1, f))\#x\#heapify(shift(h_2, f)))$
$h_1(k) \equiv k - 2^i \quad \Leftarrow \quad k \equiv 2^{i+1} + j \ \wedge \ \forall r, s : (k \equiv 2^{r+1} + s \Rightarrow r \leq i)$
$h_2(k) \equiv k - 2^{i+1} \quad \Leftarrow \quad k \equiv 2^{i+1} + 2^i + j \ \wedge \ \forall r, s : (k \equiv 2^{r+1} + 2^r + s \Rightarrow r \leq i)$
$sift(new) \equiv new$
$sift(upd(i, x, new)) \equiv upd(i, x, new)$
$sift(upd(i, x, f)) \equiv upd(i, x, f) \quad \Leftarrow \quad f \equiv upd(2 * i, y, new) \wedge x \leq y$
$sift(upd(i, x, f)) \equiv upd(i, y, upd(2 * i, x, new)) \quad \Leftarrow \quad f \equiv upd(2 * i, y, new) \wedge x > y$
$sift(upd(i, x, f)) \equiv upd(i, x, f) \quad \Leftarrow \quad f \equiv upd(2 * i, y, upd(2 * i + 1, z, g)) \wedge x \leq y \wedge x \leq z$
$sift(upd(i, x, f)) \equiv upd(i, y, upd(2 * i + 1, z, sift(upd(2 * i, x, f))))$
$\Leftarrow \quad f \equiv upd(2 * i, y, upd(2 * i + 1, z, g)) \wedge x > y \wedge y \leq z$
$sift(upd(i, x, f)) \equiv upd(i, z, upd(2 * i, y, sift(upd(2 * i + 1, x, f))))$
$\Leftarrow \quad f \equiv upd(2 * i, y, upd(2 * i + 1, z, g)) \wedge x > z \wedge y > z$

Übersetzung der 3 Fälle, in denen das Argument von *sift* kein Heap ist:
$sift(upd(i, x, f)) \equiv new \quad \Leftarrow \quad get(f, 2 * i) \equiv index(j) \wedge f \equiv upd(2 * i + 1, y, g)$
$sift(upd(i, z, f)) \equiv new \quad \Leftarrow \quad get(f, 2 * i + 1) \equiv index(j) \wedge f \equiv upd(2 * i, x, upd(4 * i, y, g))$
$sift(upd(i, z, f)) \equiv new \quad \Leftarrow \quad get(f, 2 * i + 1) \equiv index(j) \wedge f \equiv upd(2 * i, x, upd(4 * i + 1, y, g))$

Der Repräsentationsmorphismus *rep* bildet *bintree* auf *map* und alle anderen SP-Symbole auf sich selbst ab. Die Implementierungsaxiome wurden ähnlich den in Kapitel 8 abgeleiteten Programmen aus dem Beweis von Anforderungen an sie hergeleitet. Die Anforderungen sind hier die Axiome von HEAP. Sie müssen ja im Refinementmodell $Her(SP')_{rep}$ gelten, damit SP' eine Verfeinerung von HEAP ist.

Wie in Beispiel 6.4.4 gehören auch hier nicht alle Elemente des SP'-Herbrandmodells zum Refinementmodell $Her(SP')_{rep}$: Jeder nur aus definierten Funktionen von SP' zusammengesetzte Grundterm der Sorte *map* ist zu einer Normalform der Gestalt

$$upd(1, x_1, upd(2, x_2, \ldots, upd(n, x_n, new) \ldots))$$

SP'-äquivalent. Damit ist nach Lemma 6.1.2(2) und Korollar 6.4.2 SP' eine Verfeinerung von SP entlang *rep*, wenn alle mit *rep* übersetzten Instanzen von HEAP-Axiomen, die *map*-Variablen solche Normalformen zuordnen, induktive Theoreme von SP' sind.

Der **Heapsort**-Algorithmus überführt zunächst ein Feld mithilfe von *heapify* in die Felddarstellung eines

[37] Keine Hornformel, ließe sich aber unter Verwendung einer Hilfssuchfunktion in eine solche umwandeln.

partiell geordneten Baums T. Da dessen Wurzel das kleinste Element unter allen Knoteneinträgen von T enthält, wird dieser Knoten aus T entfernt und zum ersten Element des sortierten Feldes. Das – bzgl. der Feldposition – größte Blatt von T wird zur neuen Wurzel von T und mithilfe von *sift* so lange "nach unten geschoben", bis sein Eintrag von Nachfolgereinträgen nicht mehr überschritten wird. Der Baum T hat jetzt einen Knoten weniger und seine Wurzel enthält das zweitkleinste Element des Ausgangsfeldes. Hat T n Knoten, dann terminiert der Algorithmus nach maximal n Iterationen des eben beschriebenen Schrittes mit einer sortierten Permutation des Ausgangsfeldes. Als Erweiterung von SP' lautet der Heapsort-Algorithmus wie folgt:

HEAPSORT $=$ SP' `and`

`constructs` $() :\rightarrow 1 + nat$

$just : nat \rightarrow 1 + nat$

`defuncts` $heapsort, loop : map(nat, entry) \rightarrow map(nat, entry)$

$remove : map(nat, entry) \times nat \rightarrow map(nat, entry)$

$maxindex : map(nat, entry) \rightarrow 1 + nat$

`axioms` $heapsort(f) \equiv loop(heapify(f))$

$loop(new) \equiv new$

$loop(upd(i, x, new)) \equiv upd(i, x, new)$

$loop(upd(i, x, f)) \equiv upd(i, x, loop(sift(upd(suc(i), y, remove(f, k)))))$

$\Leftarrow \quad maxindex(f) \equiv just(k) \wedge get(f, k) \equiv entry(y)$

$remove(new, i) \equiv new$

$remove(upd(i, x, f), i) \equiv f$

$remove(upd(i, x, f), j) \equiv upd(i, x, remove(f, j)) \quad \Leftarrow \quad i \not\equiv j$

$maxindex(new) \equiv ()$

$maxindex(upd(i, x, f)) \equiv just(i) \quad \Leftarrow \quad maxindex(f) \equiv ()$

$maxindex(upd(i, x, f)) \equiv just(max(i, k)) \quad \Leftarrow \quad maxindex(f) \equiv just(k)$

Obwohl *heapsort* hier vollständig spezifiziert ist, genügt es, die Korrektheit des Algorithmus', d.h. die Gültigkeit der Formel

$$sorted(heapsort(f)) \ \wedge \ heapsort(f) \text{ `ist eine Permutation von' } f$$

nur für die o.g. im Refinementmodell vorkommenden Normalforminstanzen von f zu zeigen. Spezifikationen des Sortiertheitsprädikates und der Permutationsrelation (auf Listen!) findet man in den Beispielen 6.3.2 bzw. 7.2.6. ✌

6.5 Ausnahmen und Nichtdeterminismus

Die Spezifikation MAP von Feldern (siehe Beispiel 6.4.3) liefert ein Herbrand-Modell, in dem mehrere inäquivalente Grundnormalformen dasselbe Feld repräsentieren. So sind $upd(i, x, f)$ und $upd(i, x, upd(i, y, f))$ nicht MAP-äquivalent, obwohl beide Terme für dasselbe Feld stehen, weil die letzte Änderung von f an der Stelle i alle vorangegangenen Updates dieser Stelle überschreibt. In §6.1 haben wir Coprädikate eingeführt, zunächst um Komplementprädikate zu spezifizieren. Eine Axiomatisierung der MAP-Äquivalenz als Coprädikat löst auch das eben beschriebene Problem. Wir erweitern MAP um das Coprädikat $\sim$: $map \times map$ und das co-Hornaxiom

$$f \sim g \quad \Rightarrow \quad get(f, i) \equiv get(g, i).$$

Die Interpretation von $\sim$ im Herbrandmodell als größte Relation, die dieses Axiom erfüllt (siehe Def. 6.2.2), führt dazu, dass nicht nur die beiden o.g. Terme äquivalent sind, sondern auch alle anderen Grundnormalformen, die dasselbe Feld repräsentieren. Genauer gesagt, $Her(MAP)$ erfüllt folgende Äquivalenzen:

$$\begin{aligned} upd(i, x, upd(i, y, f)) &\sim upd(i, x, f) \\ upd(i, x, upd(j, y, f)) &\sim upd(j, y, upd(i, x, f)) \quad \Leftarrow \quad i \not\equiv j \end{aligned}$$

☞ Zeigen Sie analog zum Beweis der Äquivalenzeigenschaft von $\sim^{Her(SP')}$ in Beispiel 6.4.3, dass $\sim^{Her(MAP)}$ eine Äquivalenzrelation ist!

Ein anderes Beispiel für eine Klasse von Äquivalenzrelationen, die sich einfach als Coprädikate spezifizieren läßt, sind *starke Äquivalenzen*, die die Identität der Elemente eines Summanden einer zweistelligen Summensorte erhalten, jedoch alle Elemente des anderen Summanden gleichsetzen soll. Die $index + entry$-Komponente des Abstraktionshomomorphismus in Beispiel 6.4.3 hat z.B. eine starke Äquivalenz als Äquivalenzkern: Er bildet nämlich alle Grundnormalformen der Gestalt $index(t)$ auf $index(0)$ ab, diejenigen der Gestalt $entry(t)$ bleiben jedoch erhalten.

starke Äquivalenz

Definition 6.5.1 Sei $SP = (\Sigma, AX)$ eine Spezifikation und $s + s'$ eine Summensorte von Σ mit zwei Konstruktoren $def : s \to s + s'$ für "definierte" Werte und $undef : s' \to s + s'$ für "undefinierte" Werte oder **Ausnahmen** (*exceptions*).[38] Wir erweitern SP um das Coprädikat $\approx$: $(s + s') \times (s + s')$ und die co-Hornaxiome

$$x \approx y \;\Rightarrow\; (x \equiv def(x') \;\Rightarrow\; x \equiv y),$$
$$x \approx y \;\Rightarrow\; (y \equiv def(y') \;\Rightarrow\; x \equiv y).$$

Die Interpretation von $\approx$ im Herbrandmodell (der Erweiterung) von SP heißt **starke SP-Äquivalenz**.

☞ Zeigen Sie analog zum Beweis der Äquivalenzeigenschaft von $\sim^{Her(SP')}$ in Beispiel 6.4.3, dass $\approx^{Her(SP)}$ eine Äquivalenzrelation ist!

☞ Zeigen Sie, dass im Herbrandmodell von SP folgende Äquivalenz gilt:

$$x \approx y \quad\Longleftrightarrow\quad (x \equiv y \;\vee\; (\exists\, z : x \equiv undef(z) \;\wedge\; \exists\, z : y \equiv undef(z))).$$

Gleichheitsprädikate werden verwendet, um Funktionen zu axiomatisieren. Umgekehrt werden in den co-Hornformeln, die eine Äquivalenzrelation axiomatisieren, Funktionen verwendet: *top* und *pop* in Beispiel 6.4.3, *get* in MAP und *def* in den Axiomen für $\approx$. Dies ist neben der Komplementspezifizierbarkeit eine weitere Dualität zwischen Prädikaten und Coprädikaten.

Striktheit, Regularität

Definition 6.5.2 Sei $SP = (\Sigma, AX)$ eine Spezifikation wie in Def. 6.5.1. Ein Funktionssymbol $f : s_1 \times \cdots \times s_n \to s + s' \in \Sigma$ ist **strikt**, wenn für alle $1 \le i \le n$

$$f(x_1, \ldots, x_n) \equiv def(y) \;\Rightarrow\; \exists\, y : x_i \equiv def(y)$$

in $Her(SP)$ gilt. f ist **regulär**, wenn für alle $1 \le i \le n$

$$f(x_1, \ldots, x_n) \equiv def(y) \wedge \exists\, z : x_i \equiv undef(z) \;\Rightarrow\; \forall\, x : f(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n) \equiv def(y)$$

in $Her(SP)$ gilt. Ein Prädikat $r : w \in \Sigma$ ist **regulär**, wenn für alle $1 \le i \le n$

$$r(x_1, \ldots, x_n) \wedge \exists\, z : x_i \equiv undef(z) \;\Rightarrow\; \forall\, x : r(x_1, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_n)$$

gilt. SP ist **regulär**, wenn alle Funktionssymbole und alle logischen Prädikate von Σ regulär sind.

[38] Letztere werden durch die starke Äquivalenz gleichgesetzt.

Liefert eine Funktion f einen "definierten" Wert, obwohl sie auf ein "undefiniertes" Argument angewendet wurde, dann bedeutet Regularität, dass f an dieser Stelle immer denselben Wert liefert. Nichtstrikte reguläre Funktionen projizieren ihre Argumente auf einzelne Komponenten, oft in Abhängigkeit von den Werten anderer Komponenten. Im folgenden Beispiel ist der Konstruktor für Ausnahmewerte nullstellig. Deshalb bezeichnen wir ihn wie üblich mit ().

REGULÄR = TRIV(s) `and`

`constructs`	$() :\to 1+s$
	$def : s \to 1+s$
	$() :\to 1+bool$
	$def : bool \to 1+bool$
`defuncts`	$\pi_1, \pi_2 : (1+s)\times(1+s) \to 1+s$
	$\_and\_, \_or\_ : (1+bool)\times(1+bool) \to 1+bool$
	$if\_then\_else : (1+bool)\times(1+s)\times(1+s) \to 1+s$
`vars`	$x, y : 1+s \quad b : 1+bool$
`axioms`	$\pi_1(x,y) \equiv x$
	$\pi_2(x,y) \equiv y$
	$() \ and \ b \equiv ()$
	$def(true) \ and \ b \equiv b$
	$def(false) \ and \ b \equiv def(false)$
	$() \ or \ b \equiv ()$
	$def(true) \ or \ b \equiv def(true)$
	$def(false) \ or \ b \equiv b$
	$if \ () \ then \ x \ else \ y \equiv ()$
	$if \ def(true) \ then \ x \ else \ y \equiv x$
	$if \ def(false) \ then \ x \ else \ y \equiv y$

Im Gegensatz zu strikten Funktionen erlauben reguläre Funktionen das *recovering* von einem undefinierten zu einem definierten Wert. Allerdings unterscheidet eine reguläre Funktion nicht zwischen verschiedenen Ausnahmen, sondern weist allen denselben Funktionswert zu.

Die Unterscheidung zwischen definierten und undefinierten Werten induziert die folgende **flache Halbordnung** im Herbrandmodell von SP:

$$t \ \leq \ u \quad \Longleftrightarrow_{def} \quad t \equiv_{SP} u \vee (\exists\, t' : t \equiv_{SP} undef(t') \wedge \exists\, u' : u \equiv_{SP} def(u')).$$

☞ Zeigen Sie folgende Behauptungen:

(1) Bezüglich der flachen Halbordnung ist eine Funktion genau dann regulär, wenn sie monoton ist (oder auch stetig, weil hier Monotonie mit Stetigkeit zusammenfällt).

(2) Jede aus regulären Funktionen zusammengesetzte Funktion ist regulär.

(3) Eine starke Äquivalenz $\approx$: ss ist mit einer Funktion $f : s \to s' \in \Sigma$ oder einem Prädikat $r : s \in \Sigma$ verträglich, wenn f bzw. r regulär ist.

(4) $\approx$ ist genau dann mit $\equiv$ verträglich, wenn (die Interpretationen von) $\approx$ und $\equiv$ (in $Her(SP)$) miteinander übereinstimmen.

Wegen (3) und der Äquivalenzeigenschaft von $\approx$ erlaubt Regularität die Faktorisierung von $Her(SP)$ nach $\approx^{Her(SP)}$. Um diese Relation zu einer S-sortierten zu machen, setzen wir $\approx$ auf allen Sorten von Σ außer $s + s'$

mit $\equiv$ gleich. Der Quotient

$$Strong(SP) \quad =_{def} \quad Her(SP)/{\approx^{Her(SP)}}$$

unterscheidet sich vom initialen Modell $Ini(SP)$ (siehe Def. 5.1.6) dadurch, dass zwei Terme gleichgesetzt werden, wenn sie SP-äquivalent sind *oder* zu einer Ausnahmesorte gehören.

Leider gilt Lemma 5.1.7 nicht $Strong(SP)$ anstelle von $Ini(SP)$. Zumindest die Axiome von SP sollten aber auch von $Strong(SP)$ erfüllt werden. Das wird durch die Regularität der Funktionen und Prädikate von SP garantiert.

☞ Zeigen Sie, dass $Strong(SP)$ ein SP-Modell ist, wenn alle Funktionen und logischen Prädikate von Σ regulär sind!

(4) legt nahe, dass starke Äquivalenzen i.a. nicht mit Gleichheitsprädikaten verträglich sind. Immerhin sind sie *Bisimulationen* bzgl. Gleichheitsprädikaten:

Bisimulation

Definition 6.5.3 Sei $\Sigma = (S, F, R)$ eine Signatur, $\rightarrow: ss' \in R$ und A eine Σ-Struktur. Eine S-sortierte Relation $\sim \subseteq A \times A$ ist **zickzack-verträglich** mit $\rightarrow$ oder eine **Bisimulation** bzgl. $\rightarrow$, wenn für alle $a, b \in A_s$ und $a', b' \in A_{s'}$ gilt:

$$\begin{aligned} a \sim b \wedge a \rightarrow^A a' \quad &\Rightarrow \quad \exists\, b' : b \rightarrow^A b' \wedge a' \sim b', \\ a \sim b \wedge b \rightarrow^A b' \quad &\Rightarrow \quad \exists\, a' : a \rightarrow^A a' \wedge a' \sim b'. \end{aligned}$$

In der Regel bezeichnen s und s' Zustandsmengen und $\rightarrow: ss'$ die Übergangsrelation eines Transitionssystems. Handelt es sich um ein mit Ereignissen, Eingaben, Ausgaben, etc. markiertes Transitionssystem, dann ist s eine Produktsorte, z.B. $s = state \times lab$, wobei $state$ als Zustandsmenge und lab als Markierungsmengen interpretiert wird (siehe §9.1).

☞ (A) Zeigen Sie, dass jede mit $\rightarrow$ verträgliche reflexive Relation mit $\rightarrow$ zickzack-verträglich ist!

☞ (B) Zeigen Sie, dass jede $\equiv_{SP}$ umfassende Äquivalenzrelation $\sim \subseteq T_\Sigma$ mit $\equiv$ zickzack-verträglich ist! (Damit ist insbesondere die Interpretation einer starken Äquivalenz $\approx: ss$ im Herbrandmodell mit $\equiv: ss$ zickzack-verträglich.)

☞ (C) Zeigen Sie, dass die Interpretation von $\rightarrow: ss'$ im Quotienten $A/\sim$:

$$c_1 \rightarrow^{A/\sim} c_2 \iff_{def} \exists\, a_1 \in c_1 : \exists\, a_2 \in c_2 : a_1 \rightarrow^A a_2$$

(siehe Def. 4.1.2) durch

$$[a_1] \rightarrow^{A/\sim} c_2 \iff_{def} \exists\, a_2 \in c_2 : a_1 \rightarrow^A a_2$$

ersetzt werden kann, wenn $\sim$ mit $r : ss'$ zickzack-verträglich ist!

☞ (D) Sei A eine Σ-Struktur. Zeigen Sie, dass eine Relation $\sim \subseteq A \times A$ genau dann mit $\rightarrow: ss'$ zickzack-verträglich ist, wenn $\sim$ mit dem Funktionssymbol $f_\rightarrow : s \rightarrow set(s')$ verträglich ist, wobei $set(s')$ und $f_\rightarrow$ wie folgt interpretiert und $\sim$ wie folgt auf Potenzmengen fortgesetzt wird: Sei $a \in A_s$ und $M, N \subseteq A_s$.

$$\begin{aligned} A_{set(s')} \quad &=_{def} \quad \wp(A_{s'}), \\ f^A_\rightarrow(a) \quad &=_{def} \quad \{a' \mid a \rightarrow^A a'\}, \\ M \sim N \quad &\iff_{def} \quad (\forall a \in M\ \exists b \in N : a \sim b) \wedge (\forall b \in N\ \exists a \in M : a \sim b). \end{aligned}$$

Funktionen eines Typs $s \rightarrow set(s')$ bezeichnet man auch als **nichtdeterministische Funktionen** des Typs $s \rightarrow s'$.

☞ (E) Folgern Sie aus (D), dass $\sim$ genau dann mit einer Funktion $f : s \to s'$ verträglich ist, wenn $\sim$ mit dem Graphen von f (siehe Def. 5.1.19) zickzack-verträglich ist!

☞ (F) Wie lautet die Interpretation von $f_{\to}$ im Quotienten $A/\sim$?

☞ (G) Zeigen Sie, dass jede Vereinigung von Bisimulationen wieder eine Bisimulation ist und folgern Sie daraus, dass es eine größte Bisimulation (bzgl. einer festen Relation $\to$) gibt!

☞ (H) Zeigen Sie, dass die größte Bisimulation eine Äquivalenzrelation ist!

Im Zusammenhang mit Zustandsäquivalenzen werden wir uns in §9.2 noch einmal mit Bisimulationen befassen.

6.6 Partiell-rekursive Funktionen

Partiell-rekursive Funktionen sind partielle Funktionen, deren Definitionsbereich nicht entscheidbar sein muss. Hierzu gehören z.B. die Interpreter iterativer oder rekursiver Programme, die bekanntlich vor ihrer Ausführung nicht auf ihre Termination hin geprüft werden können (Halteproblem). Hat eine partielle Funktion jedoch einen entscheidbaren Definitionsbereich, dann kann man sie auf einfache Weise *totalisieren*, indem man ihr an den undefinierten Stellen besondere Werte zuweist. Zur Darstellung der erweiterten Wertebereiche haben wir Summensorten verwendet. Mithilfe der CPO-Theorie (s. §2.2) lassen sich auch partiell-rekursive Funktionen mit unentscheidbarem Definitionsbereich totalisieren. Das basiert auf einer Variante des Fixpunktsatzes von Kleene (4.2.11), die anstelle eines vollständigen Verbandes nur einen CPO verlangt. Die Schrittfunktion Φ transformiert hier nicht Relationen, sondern Funktionen, hin zur Fixpunktsemantik keines logischen, sondern eines funktionalen Programms. Dazu muss Φ, wie in Satz 4.2.11 verlangt, (aufwärts-)stetig sein. Axiomatisiert man diese Konstruktion mit Hornformeln, dann stellt sich heraus, dass die Stetigkeit der oben definierten Regularität entspricht.

Ausgangspunkt ist die Spezifikation SP einer partiellen Funktion $f : w \to s$:

$$\begin{array}{rcl} f(t_1) & \equiv & u_1 \Leftarrow H_1 \\ & \dots & \\ f(t_k) & \equiv & u_k \Leftarrow H_k. \end{array} \tag{1}$$

Die Partialität zeigt sich darin, dass SP nicht (stark) terminierend ist: Es gibt keine Reduktionsordnung $>$ mit der Eigenschaft, dass alle SP-Reduktionen $>$-absteigend sind (siehe Defs. 5.1.10 und 5.2.1).

Die Axiomatisierung der Fixpunktsemantik von f mit Hornformeln könnte wie folgt aussehen. Zunächst setzen wir voraus, dass die Bedingungen $H_1, \dots, H_k$ keine rekursiven Aufrufe von f enthalten. Dann übersetzen wir (1) in Axiome für eine **Approximationsfunktion** $f' : nat \times w \to 1 + s$, die der von einem logischen Programm induzierten Schrittfunktion entspricht (siehe §4.2):

$$\begin{array}{rcl} f'(0, x) & \equiv & () \\ f'(suc(n), t_1) & \equiv & v_1 \Leftarrow H_1 \\ & \dots & \\ f'(suc(n), t_k) & \equiv & v_k \Leftarrow H_k. \end{array}$$

Hierbei erhält man v_i aus u_i, indem man alle Teilterme $h(f(t))$ von u_i mit $h \neq f$ durch λ-Applikationen $(\lambda(x).h(x))(f'(n,t))$ ersetzt (siehe §6.6)[39] Hat u_i die Form $f(t)$, dann wird zunächst u_i durch $f(n,t)$ ersetzt.

[39] Ist $h : s \to s'$, dann ist die Funktion $(\lambda(x).h(x)) : 1 + s \to s'$ definiert durch: $(\lambda\underline{(}x\underline{)}.h(x))(\underline{(}y\underline{)}) = h(y)$. Die unterstrichenen Klammern bezeichnen die Einbettung $(\_) : s \to 1 + s$.

Die Axiome für f verwenden die Approximationsfunktion f':

$$\begin{aligned} f(x) &\equiv y \Leftarrow f'(n,x) \equiv (y) \\ f(x) &\equiv () \Leftarrow \forall n : f'(n,x) \equiv (). \end{aligned}$$

AX_f soll f als Funktion auf $Her(SP)$ spezifizieren. Dazu müßte $ESP = (\Sigma \cup \{f\}, AX \cup AX_f)$ konsistent bzgl. SP sein. Nach Korollar 6.1.6 ist das der Fall, wenn

(1) SP konfluent ist,

(2) ESP stark terminierend ist,

(3) für alle $\varphi \in AX_f$ gilt, dass φ deterministisch ist oder das von φ und φ erzeugte Overlay ein induktives Theorem von SP ist,

(4) alle von je zwei verschiedenen Axiomen von AX_f erzeugten Overlays induktive Theoreme von SP sind.

Eine mithilfe von Satz 5.2.2 konstruierte Reduktionsordnung für SP ist auch eine Reduktionsordnung für ESP, weil f weder auf der rechten Seite noch in der Prämisse eines Axioms von AX_f vorkommt. Damit wäre (2) erfüllt. (4) folgt aus der Unlösbarkeit der Konjunktion von $\exists n, y : f'(n,x) \equiv (y)$ und $\forall n : f'(n,x) \equiv ()$ im Herbrandmodell von SP. Das vom zweiten Axiom von AX_f mit sich selbst erzeugte Overlay hat die Konklusion $() \equiv ()$, erfüllt also (3). Das vom ersten Axiom mit sich selbst erzeugte Overlay lautet:

$$(y) \equiv (y') \quad \Leftarrow \quad f'(m,x) \equiv (y) \wedge f'(n,x) \equiv (y'). \tag{5}$$

Demnach bleibt zu zeigen, dass (5) ein induktives Theorem von SP ist.

Sei $\sigma : X \to T_\Sigma$ eine Lösung der Prämisse von (5) in $Her(SP)$ und seien $m\sigma$ und $n\sigma$ o.B.d.A. Normalformen, also natürliche Zahlen. Im Fall $m\sigma = n\sigma$ gilt $f'(m\sigma, x\sigma) \equiv_{SP} f'(n\sigma, x\sigma)$, also auch $(y\sigma) \equiv_{SP} (y'\sigma)$. O.B.d.A. sei $m\sigma < n\sigma$. Dann gibt es einen Term t ohne Vorkommen von $()$ sowie zwei Reduktionen

$$f'(m\sigma, x\sigma) \xrightarrow{*}_{SP} t[()/z] \quad \text{und} \quad f'(n\sigma, x\sigma) \xrightarrow{*}_{SP} t[u/z].$$

Kommt z in t nicht vor, dann ist $t[()/z] = t[u/z]$, also auch

$$(y\sigma) \equiv_{SP} f'(m\sigma, x\sigma) \equiv_{SP} t[()/z] = t[u/z] \equiv_{SP} f'(n\sigma, x\sigma) \equiv_{SP} (y'\sigma). \tag{6}$$

Andernfalls nehmen wir an, dass t aus regulären Funktionen zusammengesetzt ist (siehe Def. 6.5.2). Wegen $f'(m\sigma, x\sigma) \equiv (y\sigma)$ gilt dann $Her(SP) \models t[()/z] \equiv t[u/z]$ für *alle* $u \in T_{\Sigma, sort(z)}$. Analog zu (6) erhält man $(y\sigma) \equiv_{SP} (y'\sigma)$. Also ist (5) in $Her(SP)$ gültig, und für die relative Konsistenz von ESP bzgl. SP bliebe zu zeigen, dass SP stark terminierend und konfluent ist.

6.7 Strukturierte Sorten und λ-Terme

So wie implizit in jeder Spezifikation Gleichheitsrelationen vorausgesetzt werden, macht man in der Regel auch implizite Annahmen über die Interpretation der Sortenkonstruktoren $+$, $\times$ und $\to$:

Sortenkonstruktoren, λ-Terme

Definition 6.7.1 Seien $SP = (\Sigma, AX)$ eine Spezifikation und $s_1, s_2, \ldots, s_n$ Sorten von Σ. Eine Sorte der Form $s_1 \times \cdots \times s_n$ heißt **Produktsorte**. Eine Sorte der Form $s_1 + \cdots + s_n$ heißt **Summensorte**. Eine Sorte der Form $s_1 \to s_2$ heißt **funktionale Sorte**.

Für jede Produktsorte $s_1 \times \cdots \times s_n \in \Sigma$, $n \geq 2$, enthält Σ implizit den **Tuplung** genannten einzigen (!) Konstruktor

$$(\_, \ldots, \_) : s_1 \ldots s_n \to s_1 \times \cdots \times s_n$$

und die **Projektionen** genannten definierten Funktionen $\pi_i : s_1 \times \cdots \times s_n \to s_i$, $1 \leq i \leq n$, mit den Axiomen

$$\begin{aligned} \pi_1(x_1, \ldots, x_n) &\equiv x_1, \\ &\ldots \\ \pi_n(x_1, \ldots, x_n) &\equiv x_n. \end{aligned}$$

Für jede Summensorte $s_1 + \cdots + s_n \in \Sigma$, $n \geq 2$, enthält Σ implizit die **Injektionen** genannten einzigen (!) Konstruktoren $\kappa_i : s_i \to s_1 + \cdots + s_n$, $1 \leq i \leq n$. Die Verwendung von Summensorten induziert eine Sortenordnung:

$$s < s' \quad \Longleftrightarrow_{def} \quad \text{es gibt Injektionen } \kappa_1 : s \to s_1, \kappa_2 : s_1 \to s_2, \ldots, \kappa_n : s_{n-1} \to s'.$$

Sei $s < s'$ Dann nennt man s eine **Untersorte** von s' (siehe §4.1). Die Sortenordnung wird wie folgt auf Σ-Strukturen A fortgesetzt: Sei $a \in A_s$ und $b \in A_{s'}$.

$$a <^A b \quad \Longleftrightarrow_{def} \quad \text{es gibt Injektionen } \kappa_1, \ldots, \kappa_n \text{ mit } \kappa_n(\ldots(\kappa_1(a))\ldots) = b.$$

Für jede funktionale Sorte $s_1 \to s_2 \in \Sigma$ enthält Σ implizit die (definierte) **Applikationsfunktion**

$$apply : (s_1 \to s_2) \times s_1 \to s_2.$$

Jedes Funktionssymbol $f : s_1 \to s_2 \in \Sigma$ liefert einen nullstelligen Konstruktor $f :\to (s_1 \to s_2)$ und das folgende Axiom für *apply*:

$$apply(f, x) \equiv f(x).$$

Ein Ausdruck e der Form

$$(\lambda p_1.t_1) \mathbf{I} \ldots \mathbf{I} (\lambda p_k.t_k)$$

heißt **λ-Abstraktion**, falls $p_1, \ldots, p_k \in NF_\Sigma(X)_{s_1}$ und $t_1, \ldots, t_k \in T_\Sigma(X)_{s_2}$ gilt. Sei $[x_1, \ldots, x_n]$ die Liste der **freien Variablen** (= Variablen, die nicht in $p_1, \ldots, p_k$ vorkommen) von e in der Reihenfolge ihres Auftretens. Als Konstruktor der Sorte $s_1 \to s_2$ hat e den Typ

$$e : sort(x_1) \times \ldots \times sort(x_n) \to (s_1 \to s_2)$$

und implizit die folgenden Axiome:

$$\begin{aligned} apply(e(x_1, \ldots, x_n), p_1) &\equiv t_1, \\ &\ldots \\ apply(e(x_1, \ldots, x_n), p_k) &\equiv t_k. \end{aligned} \tag{1}$$

Die Normalformen $p_1, \ldots, p_k$ nennen wir **Argumentmuster** von e. Terme mit λ-Abstraktionen heißen **λ-Terme**. Ein Term der Form $apply(\lambda p.t, u)$ heißt **λ-Applikation**. Beliebige Terme einer funktionaler Sorte heißen **Terme höherer Ordnung**.

Z.B. liefert die λ-Abstraktion $\lambda y.not(eq(x, y))$ aus der Listenspezifikation (siehe 6.3.2) einen Konstruktor des Typs $entry \to (entry \to bool)$.

In der Regel schreibt man $t(u)$ anstelle von $apply(t, u)$.

☞ Zeigen Sie, dass bzgl. der oben definierten Sortenordnung für alle Σ-Strukturen A Folgendes gilt:

$$s < s' \quad \wedge \quad b \in A_{s'} \qquad \Rightarrow \qquad \text{es gibt genau ein } a \in A_s \text{ mit } a < b.$$

b hat also eine eindeutige Repräsentation in der Untersorte s.

☞ Zeigen Sie, dass im initialen Modell einer funktionalen Spezifikation SP (siehe Def. 5.1.6) das kartesische Produkt der Interpretationen von n Sorten $s_1, \ldots, s_n$ bijektiv zur Interpretation der Produktsorte $s_1 \times \cdots \times s_n$ ist!

Für funktionalen Sorte $s \to s'$ gibt es leider keine entsprechende Bijektion zwischen $[Ini(SP)_s \to Ini(SP)_{s'}]$ und $Ini(SP)_{s \to s'}$. Weder hat jede Funktion $f : Ini(SP)_s \to Ini(SP)_{s'}$ einen λ(-Term)-Repräsentanten, noch ist eine solche Repräsentation eindeutig. Da in der Theoretischen Informatik λ-Definierbarkeit als eine Charakterisierung von Berechenbarkeit gilt, wären alle Funktionen berechenbar, wenn jede Funktion einen λ-Repräsentanten hätte. Und gäbe es nur eindeutige λ-Repräsentanten, dann wäre nach Satz 5.1.17 die Äquivalenz λ-definierbarer Funktionen entscheidbar.

Ist ein Gleichheitsprädikat für eine funktionale Sorte $s \to s'$ überhaupt spezifizierbar? Ja, im Prinzip so, wie wir in §6.4 die MAP-Gleichheit spezifiziert haben, nämlich als Coprädikat unter Zuhilfenahme einer Zugriffsfunktion auf die jeweiligen Funktionswerte. Bei Funktionen beliebigen Typs ist das gerade die *apply*-Funktion:

extensionale Äquivalenz

Definition 6.7.2 Sei $SP = (\Sigma, AX)$ eine Spezifikation und $s \to s'$ eine funktionale Sorte von Σ. Wir erweitern SP um das Coprädikat $\asymp$: $(s \to s') \times (s \to s')$ und das co-Hornaxiom

$$x \asymp y \;\Rightarrow\; apply(x, z) \asymp apply(y, z).$$

Die Interpretation von $\asymp$ im Herbrandmodell (der Erweiterung) von SP heißt **extensionale SP-Äquivalenz**.

Wie bei starken Äquivalenzen wird $\asymp^{Her(SP)}$ zu einer S-sortierten Relation, indem wir $\asymp$ auf allen Sorten von Σ außer $s \to s'$ mit $\equiv$ gleichsetzen.

☞ Zeigen Sie analog zum Beweis der Äquivalenzeigenschaft von $\sim^{Her(SP')}$ in Beispiel 6.4.3, dass $\asymp^{Her(SP)}$ eine Äquivalenzrelation ist! Folgern Sie daraus, dass der Quotient

$$Ext(SP) \quad =_{def} \quad Her(SP)/\asymp^{Her(SP)}$$

wohldefiniert ist, falls $\asymp_{SP}$ mit den Funktionen von Σ verträglich ist!

Wann ist eine extensionale Äquivalenz mit einer Funktion verträglich? Kurz gesagt, immer dann, wenn deren Axiome nicht auf spezielle Repräsentationen (z.B. bestimmte λ-Abstraktionen) ihrer funktionalen Argumenten Bezug nehmen. So sind z.B. die extensionalen LIST-Äquivalenzen auf den Sorten $s \to bool$ bzw. $s \to s'$ mit den Funktionen $filter$ bzw. map verträglich, weil in deren Axiomen die Variable f das einzige funktionale Argument ist (siehe Beispiel 6.3.2).

Außer bei starken Äquivalenzen beschreiben die Axiome als Coprädikate definierter Äquivalenzrelationen offenbar die Verträglichkeit mit bestimmten Funktionen: *pop* und *top* bei $\sim^{Her(SP')}$ (siehe 6.4.3), *get* bei $\sim^{Her(MAP)}$ (siehe §6.4), $f_{\to} : s \to set(s')$ bei Bisimulationen und jetzt *apply* bei extensionaler Äquivalenz. Solche Funktionen werden auch **Destruktoren** genannt (siehe §9.2).

Sei t ein λ-Term mit einer λ-Abstraktion $\lambda p.u$. Jedes Auftreten einer Variablen x von p in u nennt man — analog zur Bindung quantifizierter Variablen in Formeln — ein **gebundenes Vorkommen** von x in t. Andere Vorkommen von x in t heißen **frei**. x ist eine **freie Variable** von t, wenn x in t frei vorkommt. t ist **geschlossen**, wenn t keine freien Variablen enthält.

Rewriting-Schritte mit *apply*-Axiomen (siehe 6.7.1(1)), bei denen die freien Variablen $x_1, \ldots, x_n$ der applizierten λ-Terme nicht instanziiert werden, entsprechen Reduktionen im Sinne von Anwendungen der **β-Regel**

des **λ-Kalküls**.[40] So erhält man z.B. — über einer Spezifikation natürlicher Zahlen — die Reduktion

$$(\lambda x.(\lambda y.(x+y))(3))(4) \longrightarrow (\lambda x.(x+3))(4) \longrightarrow 4+3.$$

Eine andere Reduktion desselben λ-Terms führt zum selben Ergebnis:

$$(\lambda x.(\lambda y.(x+y))(3))(4) \longrightarrow (\lambda y.(4+y))(3) \longrightarrow 4+3.$$

Um die Konfluenz von Spezifikationen mit der β-Regel allgemein sicherzustellen, müssen jedoch — wie bei Formeln mit Quantoren — vor der Instanziierung eines λ-Terms t durch eine Substititution $\sigma : X \to T_\Sigma(X)$ alle sowohl in $range(\sigma) =_{def} dom(\sigma)\sigma$ als auch in t vorkommenden Variablen in Variablen von $X \setminus range(\sigma)$ umbenannt werden. Ohne Umbenennung erhielte man z.B. die folgenden Reduktionen desselben λ-Terms, deren Redukte auf kein gemeinsames Redukt reduzierbar sind:

$$(\lambda x.(\lambda y.(x+y))(3))(1+y) \longrightarrow (\lambda x.(x+3))(1+y) \longrightarrow (1+y)+3,$$
$$(\lambda x.(\lambda y.(x+y))(3))(1+y) \longrightarrow (\lambda y.(1+y)+y)(3) \longrightarrow (1+3)+3.$$

Nach Umbennung der gebundenen Variable in der λ-Abstraktion $(\lambda y.x+y)(3)$ vor deren Anwendung auf $1+y$ führt die zweite Reduktion zum selben Redukt wie die erste:

$$(\lambda x.(\lambda y.(x+y))(3))(1+y) \overset{y'/y}{\rightsquigarrow} (\lambda x.(\lambda y'.(x+y'))(3))(1+y) \longrightarrow (\lambda y'.(1+y)+y')(3) \longrightarrow (1+y)+3.$$

☞ Übersetzen Sie diese λ-Terme in Terme einer gemäß Def. 6.7.1 gebildeten Signaturerweiterung und übertragen Sie die β-Reduktionen in Anwendungen der *apply*-Axiome 6.7.1(1). Ist die Variablenumbenennung dann noch erforderlich?

Durch ❙ miteinander verknüpfte λ-Abstraktionen mit komplexen Argumentmustern kann man mithilfe folgender Axiome in eine einzige äquivalente Abstraktion mit einer Variablen als Argumentmuster transformieren.[41] Seien f, g, h, x Variablen.

$$(\lambda p_1.t_1)❙\ldots❙(\lambda p_k.t_k) \equiv \lambda x.match(p_1, x, t_1, match(p_2, x, t_2, \ldots, match(p_k, x, t_k, fail)\ldots))$$
wobei x in $p_1, \ldots, p_k, t_1, \ldots, t_k$ nicht vorkommt

$$match(c(p_1, \ldots, p_n), u, t, rest) \equiv$$
$$ite(eq_c(u), match(p_1, arg_1(u), match(p_2, arg_2(u), \ldots, match(p_n, arg_n(u), t, rest)\ldots, t, rest), t, rest), rest)$$
für alle n-stelligen Konstruktoren c, $n \geq 1$

$$match(c, u, t, rest) \equiv ite(eq_c(u), t, rest)$$ für alle Konstruktorkonstanten c

$$match(x, u, t, rest) \equiv t$$ für alle Variablen x.

$match(p, u, t, rest)$ liefert, intuitiv gesprochen, t bzw. $rest$, je nachdem, ob die Normalform von u eine Instanz von p ist oder nicht.

Durch Anwendung dieser Gleichungen lässt sich jede λ-Abstraktion in eine der Form $\lambda x.t$ transformieren, die folgende Hilfsfunktionen enthält:

$ite : (s \to bool) \times (s \to s') \times (s \to s') \to (s \to s')$		für alle funktionalen Sorten $s \to s'$
$eq_c : s \to bool$		für alle Konstruktoren $c : w \to s$
$arg_i : s \to s_i$		für alle Konstruktoren $c : s_1 \ldots s_n \to s$ und $1 \leq i \leq n$
$fail :\to (s \to s')$		für alle funktionalen Sorten $s \to s'$.

[40] Mehr zum λ-Kalkül in §11.1
[41] Wir folgen hier [109], §§5.7 u. 5.8.

Hier ist $fail$ ein Konstruktor, während ite ("if-then-else"), eq_c und $arg_1, \ldots, arg_k$ durch die folgenden Axiome definiert werden:

$$
\begin{array}{lll}
ite(h,f,g)(x) \equiv f(x) \Leftarrow h(x) \equiv true & & \\
ite(h,f,g)(x) \equiv g(x) \Leftarrow h(x) \equiv false & & \\
eq_c(c(x_1,\ldots,x_n)) \equiv true & \text{für alle Konstruktoren } c & \\
eq_c(d(x_1,\ldots,x_n)) \equiv false & \text{für alle Konstruktoren } c \neq d & \\
arg_i(c(x_1,\ldots,x_n)) \equiv x_i & \text{für alle Konstruktoren } c \text{ und } 1 \leq i \leq n. &
\end{array}
$$

Beispiel 6.7.3

REPAL = NAT and LIST[TRIV(s)] and (s. 5.1.5 u. 6.3.2)

sorts	$tree$
constructs	$leaf : nat \to tree$
	$\_\#\_ : tree \times tree \to tree$
defuncts	$rev : list(s) \to list(s)$
	$pal : list(s) \to bool$
	$rep : tree \times nat \to tree$
	$min : tree \to nat$
	$repByMin : tree \to tree$
vars	$x, y : entry \quad L, L' : list(s) \quad m, n : nat \quad T, T' : tree$
axioms	$rev([]) \equiv []$
	$rev(x : L) \equiv rev(L) +\!\!+ [x]$
(A)	$pal(L) \equiv eq(L, rev(L))$
	$min(leaf(n)) \equiv n$
	$min(T\#T') \equiv min(min(T), min(T'))$
	$rep(leaf(m), n) \equiv leaf(n)$
	$rep(T\#T', n) \equiv rep(T, n)\#rep(T', n)$
(B)	$repByMin(T) \equiv rep(T, min(T))$

$rev(L)$ revertiert die Liste L. Das Herbrandmodell von REPAL erfüllt $pal(L) \equiv true$ genau dann, wenn L ein **Palindrom** ist. $rep(T, n)$ ersetzt alle Blatteinträge von T durch n. $min(T)$ berechnet den minimalen Blatteintrag des Baums T. $repByMin(T)$ ersetzt alle Blatteinträge von T durch das Minimum der Blatteinträge von T. Die Axiome für pal bzw. $repByMin$ liefern recht ineffiziente Implementierungen der beiden Funktionen, weil jede Liste bzw. jeder Baum zweimal durchlaufen werden müssen, um das jeweilige Ergebnis zu berechnen. Mithilfe von λ-Abstraktionen lassen sie sich effizienter implementieren:

REPALA = NAT and LIST[TRIV(s)] and

sorts	$tree$
constructs	$leaf : nat \to tree$
	$\_\#\_ : tree \times tree \to tree$
defuncts	$eq\&rev : list(s) \to ((list(s) \to bool) \times list(s))$
	$pal : list \to bool$
	$rep\&min : tree \to ((nat \to tree) \times nat)$
	$repByMin : tree \to tree$
vars	$x, y : s \quad L, L', L_1 : list(s) \quad f : list(s) \to bool \quad k, m, n : nat \quad T, T' : tree \quad g, h : nat \to tree$
axioms	$eq\&rev([]) \equiv (\lambda[].true \blacksquare \lambda x : L.false, [])$
	$eq\&rev(x : L) \equiv (\lambda[].false \blacksquare \lambda y : L'.(eq(x, y)\ and\ f(L')), L_1 +\!\!+ [x])$
	$\Leftarrow \quad eq\&rev(L) \equiv (f, L_1)$

(A') $pal(L) \equiv f(L') \Leftarrow eq\&rev(L) \equiv (f, L')$
$rep\&min(leaf(n)) \equiv (leaf, n)$
$rep\&min(T\#T') \equiv (\lambda k.(g(k)\#h(k)), min(m, n))$
$\Leftarrow rep\&min(T) \equiv (g, m) \wedge rep\&min(T') \equiv (h, n)$
(B') $repByMin(T) \equiv g(n) \Leftarrow rep\&min(T) \equiv (g, n)$

☞ Zeigen Sie durch Induktion über (die Größe von) *list*- bzw. *tree*-Normalformen, dass das Herbrandmodell von REPAL `and` REPALA die Gleichungen

$$eq\&rev(L) \equiv (\lambda L'.eq(L, L'),\ rev(L)), \tag{1}$$
$$rep\&min(T) \equiv (\lambda n.rep(T, n),\ min(T)) \tag{2}$$

erfüllt! (A') und (B') kann man auch systematisch aus (1) und (A) bzw. (2) und (B) herleiten (s. §8.4). ✌

Eine aus Konstruktoren bestehende λ-Abstraktion wie $\lambda x.(leaf(x)\#leaf(x))$ kann man als Abstraktion einer **verkettete Datenstruktur** ansehen oder sie als solche implementieren. Gebundene Variablen entsprechen Zeigern, die bei β-Reduktionen gesetzt werden. So wäre z.B. die

$$(\lambda x.(leaf(x)\#leaf(x)))(5) \longrightarrow leaf(5)\#leaf(5)$$

nicht anderes als die Setzung des Zeigers x auf 5.

In §5.2 haben wir deterministische Hornaxiome der Form

$$\begin{aligned} f(u_1) \equiv t_1 &\Leftarrow t_{11} \equiv u_{11} \wedge \cdots \wedge t_{1n_1} \equiv u_{1n_1} \\ &\cdots \\ f(u_k) \equiv t_k &\Leftarrow t_{k1} \equiv u_{k1} \wedge \cdots \wedge t_{kn_k} \equiv u_{kn_k} \end{aligned} \tag{3}$$

behandelt, die sich direkt in ein funktionales Programm übersetzen lassen. Wichtig ist hier, dass alle $u_1, \ldots, u_n$ Normalformen sind. Aus (3) wird das Haskell-Programm

```
f(u1) = let u11 = t11 ... u1n1 = t1n1 in t1
...
f(uk) = let uk1 = tk1 ... uknk = tknk in tk
```

Compiler funktionaler Sprachen übersetzen solche Programme in λ-Abstraktionen. Dabei werden die lokalen Definitionen `let val u = t` (bzw. Hornformelprämissen von (3)) durch λ-Applikationen ersetzt. So wird aus den Axiomen

$rep\&min(leaf(n), m) \equiv (leaf(m), n)$
$rep\&min(T\#T', m) \equiv (U\#U', min(n, n')) \Leftarrow rep\&min(T, m) \equiv (U, n) \wedge rep\&min(T', m) \equiv (U', n')$

von Beispiel 5.1.5 die λ-Abstraktion

$\lambda(leaf(n), m).(leaf(m), n)$ ❙
$\lambda(T\#T', m).(\lambda(U, n).(\lambda(U', n').(U\#U', min(n, n')))(rep\&min(T', m)))(rep\&min(T, m)).$

Allgemein wird aus (3) die λ-Abstraktion

$$e_f = \lambda u_1.(\lambda u_{11}.\ldots.(\lambda u_{1n_1}.t_1)(t_{1n_1})\ldots)(t_{11}) \text{ ❙ } \ldots \text{ ❙ } \lambda u_k.(\lambda u_{k1}.\ldots.(\lambda u_{kn_k}.t_k)(t_{kn_k})\ldots)(t_{k1}).$$

War das Programm rekursiv, dann kommt f in e_f vor. In jedem Fall ist die Bedeutung von f eine **Lösung** der Gleichung $f \equiv e_f$ in f. Das drückt man im λ-Kalkül mit dem μ-Operator aus: $\mu f.e_f$ bezeichnet die – im Fall

einer vollständigen Spezifikation von f eindeutige – Lösung von $f \equiv e_f$ in f. Sie entspricht der Interpretation von f im initialen Modell von (3). Zum Beispiel entsprechen die Axiome für die Subtraktion:

$$\begin{aligned} &n - 0 \equiv n, \\ &0 - n \equiv 0, \\ &suc(m) - suc(n) \equiv m - n \end{aligned}$$

aus Beispiel 5.1.3) der **μ-Abstraktion**

$$\mu - .(\lambda(n, 0).n \blacksquare \lambda(0, n).0 \blacksquare \lambda(suc(m), suc(n)).m - n).$$

Der syntaktische Aufbau von μ-Abstraktionen ist ähnlich dem von λ-Abstraktionen. Es gibt freie und gebundene Variablen und man kann sie wie oben applizieren. Allerdings ist f in $\mu f.e_f$ immer eine Variable funktionaler Sorte. Man könnte auch μ-Abstraktionen als Konstruktoren auffassen und alle durch Axiome der Form (3) definierten Funktionen im rekursiven Fall als μ-Abstraktionen und im nicht-rekursiven Fall als λ-Abstraktionen darstellen.

Die Übersetzung der Hornformeln (3) in den Term $\mu f.e_f$ verläuft gegenüber dem flattening von (3) (siehe Def. 5.1.19) in entgegengesetzter Richtung. Flattening transformiert ein funktional-logisches Programm in ein logisches, μ-Abstraktion überführt es in ein streng funktionales ohne logische Prämissen.

Der μ-Operator läßt sich auch als Anwendung des **Fixpunktoperators** $Y : (s \to s) \to s$ auf eine λ-Abstraktion definieren:

$$\mu f.t \quad =_{def} \quad Y(\lambda f.t).$$

Umgekehrt kann der Fixpunktoperator als Lösung der Gleichung $f(Y(f)) \equiv Y(f)$ in Y definiert werden. Die λ-Applikation

$$Y(f) \quad =_{def} \quad (\lambda x.f(x(x)))(\lambda x.f(x(x)))$$

liefert eine Lösung. Allerdings verhindert die Selbstanwendung von x in diesem Ausdruck dessen Verwendung im typisierten Kontext, wie wir ihn hier (aus guten Gründen) voraussetzen. Schlimmer noch: Mithilfe prädikatenlogischer Interpretationen von λ-Abstraktion, λ-Applikation und f kann man aus diesem Ausdruck den berühmtesten logischen Widerspruch, die **Russellsche Antinomie**, herleiten (siehe §11.4).

6.8 Monaden

Normalerweise verlangt die Komposition zweier Funktionen f und g, dass die Zielsorte von f mit der Quellsorte von g übereinstimmt. Genau das ist aber oft nicht der Fall, wenn man zustandsbasiert programmiert, d.h. wenn f neben der Berechnung (sichtbaren) Werte Zustandsänderungen bewirkt ("Seiteneffekte" hat). Zustände sind dann nämlich weitere (implizite) Werte von f, die aber im Definitionsbereich von g nicht vorkommen. Damit g dennoch auf die Werte von f anwendbar ist, muss g in eindeutiger Weise auf den Wertebereich von f fortsetzbar sein, so dass man weiter g schreiben kann, auch wenn die Fortsetzung gemeint ist. Das gilt z.B. für eine Substitution σ, die ja eigentlich nur auf einer Menge X von Variablen definiert ist, dann aber auch auf andere Terme angewendet wird. Man weiss eben, wie sich die Fortsetzung aus der auf X definierten Funktion ergibt.

Ein Sortenkonstruktor M, der jeden Wertebereich A so zu einer Menge $M(A)$ erweitert, dass sich Funktionen von A nach $M(B)$ eindeutig zu Funktionen von $M(A)$ nach $M(B)$ fortsetzen lassen, heißt **Monade**. M gehört also zu der gleichen syntaktischen Kategorie wie $\times$, $+$ und $\to$. Die am häufigsten verwendeten sind **Zustandsmonaden**. Hier beschreibt jeder Term der Sorte $M(s)$ eine Zustandstransformation:

STATEMON[TRIV($state$),NEQ(s),NEQ(s')] `where` STATEMON =

`sorts`	$M(s) \ =_{def} \ state \to (s \times state)$
`defuncts`	$return : s \to M(s)$
	$\_^* : (s \to M(s')) \to (M(s) \to M(s'))$
	$\_ \ggg= \_ : M(s) \times (s \to M(s')) \to M(s')$
	$\_ \gg \_ : M(s) \times M(s') \to M(s')$
`vars`	$x : s \quad st, st' : state \quad m : M(s) \quad m' : M(s') \quad f : s \to M(s')$
`Horn axioms`	$return(x)(st) \equiv (x, st)$
	$f^*(m)(st) \equiv f(x)(st') \quad \Leftarrow \quad m(st) \equiv (x, st')$
	$m \ggg= f \equiv f^*(m)$
	$m \gg m' \equiv m \ggg= \lambda x.m'$

Der Sortenkonstruktor M, die Einbettung $return$ und der Fortsetzungsoperator $*$ sind hier speziell für Zustandsmonaden definiert. $\ggg=$, $\gg$ und $\circ$ hingegen werden in jeder Monade wie oben aus $return$ und $*$ abgeleitet. $m \ggg= \lambda x.m'$ repräsentiert die Zuweisung der Ausgabe der Zustandstransformation m an die Variable x und die anschließende Verwendung des Wertes von x als Eingabe der nachfolgenden Zustandstransformation m'. Deshalb kürzt man einen Term der Form

$$m_1 \ggg= \lambda x_1.(m_2 \ggg= \lambda x_2.(\ldots(m_n \ggg= \lambda x_n.m)\ldots))$$

in Haskell wie folgt ab:

$$do \ \{x_1 \leftarrow m_1; \ x_2 \leftarrow m_2; \ \ldots; \ x_n \leftarrow m_n; \ m\}.$$

So kommt man in Haskell zu imperativen Programmen. Entscheidend ist hierbei, dass Werte der Sorte *state* bei der Ausführung solcher Terme zwar verändert werden, aber in der Syntax nicht explizit auftauchen. Genau das gilt eben auch für imperative Programme. Belegt man *state* mit einer konstruktorbasierten Sorte wie *map* (s. Bsp. 6.4.3), dann lassen sich die zugehörigen Konstruktoren und definierten Funktionen in Zustandstransformationen übersetzen und wie Methoden im objektorientierten Sinn verwenden:

MAPMON[NEQ(s)] = STATEMON$_{map(index,entry)/state}$[MAP[NEQ($index$),NEQ($entry$)]]
[NEQ($index + entry$),NEQ(s)] `and`

`defuncts`	$New : M(index + entry)$
	$Upd : index \times entry \to M(index + entry)$
	$Get : index \to M(index + entry)$
`vars`	$f : map(index, entry) \quad i : index \quad x : entry$
`Horn axioms`	$New(f) \equiv ((), new)$
	$Upd(i, x)(f) \equiv (index(i), upd(i, x, f))$
	$Get(i)(f) \equiv (get(f, i), f)$

Nehmen wir z.B. die folgende Funktion $vals : list(defuse) \to list(index + entry)$, die eine Liste von Veränderungen und Benutzungen von Variablen (Elementen der Sorte $index$) in die Liste der an den Benutzungsstellen gültigen Werte überführt:

DEFUSE = BOOL `and`

`sorts`	$index \quad entry \quad defuse$
`constructs`	$def : index \times entry \to defuse$
	$use : index \to defuse$
`preds`	$\_ \not\equiv \_ : defuse \times defuse$
	Axiome für $\not\equiv$ gemäß Satz 6.2.7

USELIST1 = LIST$_{defuse/s}$[DEFUSE] `and` LIST[TRIV($index + entry$)[BOOL]] `and`
MAP[NEQ($index$),NEQ($entry$)] `and`

`defuncts`	$vals : list(defuse) \to list(index + entry)$
	$loop : list(defuse) \times map(index, entry) \to list(index + entry)$
`vars`	$L : list(defuse)\ \ i : index\ \ x : entry\ \ f : map(index, entry)$
`Horn axioms`	$vals(L) \equiv loop(L, new)$
	$loop([], f) \equiv []$
	$loop(def(i, x) : L, f) \equiv loop(L, upd(i, x, f))$
	$loop(use(i) : L, f) \equiv get(f, i) : loop(L, f)$

Hier hat die Schleifenfunktion *loop* einen Zustandsparameter, nämlich das Feld f. Auf der Basis von MAPMON lässt sich f verstecken und *loop* wird zur Schleifenfunktion *Loop* ohne Zustandsparameter:

USELIST2 = LIST$_{defuse/s}$[DEFUSE] `and`
MAPMON$_{list(index+entry)/s}$[LIST[TRIV($index + entry$)]] `and`

`defuncts`	$vals : list(defuse) \to list(index + entry)$
	$Loop : list(defuse) \to M(list(index + entry))$
`vars`	$L : list(defuse)\ \ i : index\ \ x : entry\ \ y : index + entry\ \ f, g : map(index, entry)$
	$L' : list(index + entry)$
`Horn axioms`	$vals(L) \equiv L' \ \Leftarrow\ do\ \{New;\ Loop(L)\}\ (f) \equiv (L', g)$
	$Loop(nil) \equiv return([])$
	$Loop(def(i, x) : L)\ \equiv\ do\ \{Upd(i, x);\ Loop(L)\}$
	$Loop(use(i) : L)\ \equiv\ do\ \{y \leftarrow Get(i);\ L' \leftarrow Loop(L);\ return(y : L')\}$

Die als Wertebereich einer partiellen Funktion $f : w \to s$ verwendete Summensorte $1 + s$ kann man zur **Fehlermonade** erweitern:

ERRORMON[NEQ(s),NEQ(s')]

`constructs`	$() :\to 1 + s$
	$just : s \to 1 + s$
`defuncts`	$return : s \to 1 + s$
	$fail :\to 1 + s$
	$\_^* : (s \to 1 + s') \to (1 + s \to 1 + s')$
	$\_ \ggg= \_ : 1 + s \times (s \to 1 + s') \to 1 + s'$
`vars`	$x : s\ \ e : 1 + s\ \ f : s \to 1 + s'$
`Horn axioms`	$return(x) \equiv just(x)$
	$fail \equiv ()$
	$f^*(()) \equiv ()$
	$f^*(just(x)) \equiv f(x)$
	$e \ggg= f \equiv f^*(e)$

Wir haben die Fehlermonade in der Funktion `unifyall` verwendet (siehe §4.2).

Schließlich kann jeder polymorphe Datentyp mit einer (!) Parametersorte zur Monade erweitert werden. Z.B. entspricht der Sternoperator $*$ der **Listenmonade** der *concatMap*-Funktion (siehe Beispiel 6.3.2):

LISTMON[TRIV(s)[BOOL],TRIV(s')[BOOL]] = MAPLIST[TRIV(s)[BOOL],TRIV(s')[BOOL]] `and`

`defuncts`	$return : s \to list(s)$
	$fail :\to list(s)$

$\_^* : (s \to list(s')) \to (list(s) \to list(s'))$
$\_ \gg\!= \_ : list(s) \times (s \to list(s')) \to list(s')$
`vars` $x : s \quad L : list(s) \quad f : s \to list(s')$
`Horn axioms` $return(x) \equiv [x]$
$fail \equiv []$
$f^*(L) \equiv concatMap(f, L)$
$L \gg\!= f \equiv f^*(L)$

Die oben hinter STATEMON eingeführte *do*-Notation gilt natürlich auch für ERRORMON und LISTMON. Da es Konstruktoren für $E(s)$ bzw. $list(s)$ gibt, können anstelle der Variablen $x_1, \ldots, x_n$ auch Normalformen stehen, was dazu führen kann, dass die durch die λ-Terme definierten Funktionen partiell sind. Deren Totalisierung dient die in ERRORMON und LISTMON definierte Funktion $fail$. Der Term

$$do\ \{t_1 \leftarrow m_1;\ t_2 \leftarrow m_2;\ \ldots;\ t_n \leftarrow m_n;\ m\}$$

mit Normalformen $t_1, \ldots, t_n$ steht dann nämlich für:

$$m_1 \gg\!= \lambda t_1.(m_2 \gg\!= \lambda t_2.(\ldots(m_n \gg\!= \lambda t_n.m \blacksquare \lambda u_n.fail)\ldots) \blacksquare \lambda u_2.fail) \blacksquare \lambda u_1.fail,$$

wobei vorausgesetzt wird, dass für alle $1 \leq i \leq n$ jede Normalform der Sorte von t_i eine Instanz von t_i oder u_i ist.

☞ Zeigen Sie, dass die folgende Funktion f eine Liste aller Paare (x, y) mit $x \in L$, $y \in L'$ und $x \neq y$ liefert!

PAIRLIST = LISTMON[TRIV(s)[BOOL],TRIV(s')[BOOL]] `and`
`sorts` $list(bool) \quad list(s \times s)$
`defuncts` $f : list(s) \times list(s) \to list(s \times s)$
`vars` $x, y : s \quad L, L' : list(s)$
`Horn axioms` $f(L, L') \equiv do\ \{x \leftarrow L;\ y \leftarrow L';\ true \leftarrow return(neq(x, y));\ return((x, y))\}$

Der λ-Kalkül ist wegen seiner knappen Syntax die übliche Zwischensprache für Compiler funktionaler Programme. Kapitel 11 behandelt seine operationellen (§11.1) und modelltheoretischen (§§11.2-11.4) Eigenschaften. [28], Kap. 4, liefert einen kurzen, leicht verständlichen Einstieg.

Da wir λ-Abstraktionen als Konstruktoren auffassen, können wir mit dem Narrowing-Kalkül (Def. 5.1.21) im Prinzip auch Normalform-Lösungen berechnen, die funktionale Objekte repräsentieren. Dabei kann es sich aber nur um in der gegebenen Spezifikation bereits vorhandene Funktionssymbole oder λ-Abstraktionen handeln. Wie schon in §6.6 bemerkt wurde, sind λ-Abstraktionen sind Darstellungen funktionaler Objekte, repräsentieren diese aber nicht eindeutig. Ein geeigneter Unifikationsalgorithmus müßte Terme, die funktionale Objekte repräsentieren, *modulo extensionaler Äquivalenz* unifizieren (Def. 6.7.2). Obwohl diese Unifizierbarkeit höherer Ordnung allgemein nicht entscheidbar ist, werden unter den Titeln **higher-order unification** und **higher-order narrowing** Lösungsalgorithmen entwickelt, die zumindest unter akzeptablen Einschränkungen funktionieren.

Unifikation höherer Ordnung ist auch ein Ansatz zur **deduktiven Programmsynthese**: Man spezifiziert eine Ein/Ausgabe-Relation r und versucht, Programme, die r realisieren, abzuleiten, indem man die Formel $r(x, f(x))$ in f löst. Baut das Lösungsverfahren nur auf den Regeln des Narrowing-Kalküls auf (Def. 5.1.21), dann könnte es vielleicht λ-Abstraktionen ableiten, aber keine μ-Abstraktionen, also keine rekursiven Programme. Deren Herleitung erfordert nämlich die Verwendung von **Induktionsregeln**. Das kommt daher, dass deduktive Programmsynthese Ableitungen produziert, die im Prinzip invers zu Korrektheitsbeweisen verlaufen, und diese Induktionsschritte enthalten, sobald das untersuchte Programm rekursiv ist.

Wir wollen hier nicht den λ-Kalkül bemühen, um Programmsynthese zu betreiben.[42] Vielmehr werden in Kapitel 8 einige Syntheseregeln behandelt, mit denen man Programme ableiten kann, die eine bestimmte vorgegebene syntaktische Struktur aufweisen. Zur Herleitung solcher Regeln werden tatsächlich Korrektheitsbeweise umgedreht: Man versucht eine gewünschte Ein/Ausgabe-Relation für eine noch nicht axiomatisierte Funktion f zu beweisen und trifft dabei — *eureka!* — auf geeignete Axiome für f, mit denen sich der Beweis erfolgreich zuendeführen läßt.

7 Programmverifikation

In diesem und dem folgenden Kapitel wollen wir uns näher mit Beweisregeln befassen, die zur Verifikation und Synthese von Programmen besonders geeignet sind. Wir gehen aus von abstrakten Programmen, wie sie durch die Axiome einer konstruktorbasierten Spezifikation $SP = (\Sigma, AX)$ gegeben sind. **Verifikation** bedeutet dann, als prädikatenlogische Formeln formulierte Eigenschaften einzelner Funktionen oder Prädikate als induktive Theoreme von SP nachzuweisen. Umgekehrt bedeutet Programm**synthese**, aus — zunächst unbewiesenen — Anforderungen an Funktionen oder Prädikate Axiome einer Spezifikation SP herzuleiten derart, dass die Anforderungen im Herbrandmodell von SP gelten. Programm**transformation** ist der Übergang von einer Spezifikation in eine induktiv äquivalente (Def. 6.1.3) oder — bei Signaturwechsel — in eine bzgl. der Ausgangsspezifikation monotone oder konsistente (Def. 6.1.4). In diesem Kapitel geht es nur um Verifikation. Synthese- und Transformationsschemata werden in Kapitel 8 behandelt. Eine leicht verständliche Einführung in die Thematik sowie — im Hinblick auf rein funktionale Programme — interessante Ergänzungen bietet [28], Kap. 3.

7.1 Natürliches Beweisen

Die bisher behandelten Regelsysteme sind zur praktischen Verifikation wenig geeignet. Der **Schnittkalkül** (Def. 4.2.5) diente i.w. der Definition des Herbrandmodells. Er liefert **synthetische**, **bottom-up-** oder **Vorwärts-Ableitungen** von Axiomen hin zum **Beweisziel** (**Behauptung**, **conjecture**) und produziert so oft riesige **Beweisbäume** mit vielen Zweigen, die mit nicht weiter transformierbaren, vom Beweisziel verschiedenen, Formeln enden. Insbesondere verlangt die Ableitung von Gleichungen i.a. zahlreiche Anwendungen von Kongruenzaxiomen. Der **Reduktionskalkül** (Def. 5.1.9) kommt ohne Kongruenzaxiome aus und erzeugt **analytische**, **top-down-** oder **Rückwärts-Ableitungen** von Beweiszielen hin zu Tautologien oder Widersprüchen. Beide Kalküle sind auf Beweise von Goals beschränkt. Der **Narrowing-Kalkül** (Def. 5.1.21) diente der Berechnung von Goal-Lösungen. Zu diesem Zweck wird er auch praktisch eingesetzt, allerdings oft in Verbindung mit weiteren Regeln zur Transformation von Constraints.

Analytische Kalküle dienen oft der Widerlegung. Man beweist die Formel φ, indem man $\neg\varphi$ in die Formel *False* transformiert. Manchmal ist es i.w. die Richtung der Regelanwendung, die einen Kalkül als bottom-up- bzw. top-down-Kalkül ausweist. Z.B. liefert der Modus Ponens die Umkehrung

$$\frac{G}{G \Leftarrow H, \quad H} \Uparrow$$

Von hier aus gelangt man zur Resolutionsregel, indem man $G \Leftarrow H$ auf Axiome beschränkt und diese Eigenschaft zur **Anwendbarkeitsbedingung** (**applicability condition**) der neuen Regel macht:

$$\frac{G}{H} \Uparrow \qquad \text{falls } G \Leftarrow H \text{ ein Axiom ist.}$$

I.a. sind analytische Kalküle effizienter als synthetische, weil sie weniger scheiternde Ableitungen produzieren. Noch zielgerichteter sind **sequentielle Beweise** $\varphi_1, \ldots, \varphi_n$, bei denen für alle $1 \leq i < n$ φ_{i+1} aus φ_i

[42] Ein solcher Ansatz wird z.B. in [109] ab Kapitel 11 näher ausgeführt.

durch Anwendung einer Regel entsteht. Von dieser Art sind Reduktionen und Narrowing-Ableitungen, aber Schnittkalkül-Ableitungen in der Regel nicht. Alle in diesem und den folgenden Abschnitten vorgestellten Regeln zur Programmverifikation sind ausschließlich zur Konstruktion sequentieller Beweise gedacht.

Die einzige schon in vorangegangenen Kapiteln erwähnte Regel zur Programmverifikation ist **Fixpunktinduktion** für logische Prädikate (s. §4.2) bzw. definierte Funktionen (s. §5.1). Sie wird in §7.2 in mehreren Richtungen verallgemeinert. Prämisse und Konklusion der Fixpunktinduktion sind von einem Typ prädikatenlogischer Formeln, auf den wir die Formeln eines formalen Beweises in Zukunft generell beschränken werden: endliche Konjunktionen negationsfreier Implikationen. Typische Beweisziele, die bei der Programmverifikation auftreten, sind: **Äquivalenzen von Funktionen**:

$$f(x) \equiv g(x),$$

Ein/Ausgabe-Relationen von Funktionen:

$$f(x) \equiv y \quad \Rightarrow \quad \varphi(x, y),$$

Invarianzbedingungen von Funktionen oder Transitionsrelationen:

$$\begin{aligned} \varphi(x) \quad &\Rightarrow \quad \varphi(f(x)), \\ \varphi(x) \wedge x \xrightarrow{*} y \quad &\Rightarrow \quad \varphi(y), \end{aligned}$$

Erreichbarkeitsbedingungen von Transitionsrelationen:

$$\varphi(x) \quad \Rightarrow \quad \exists\, y : x \xrightarrow{*} y \wedge \psi(y).$$

Im Folgenden geht es um Regeln und Strategien für analytische Beweise, also Transformationen eines Beweisziels in die Formel *True* oder *False* oder eine (disjunktiv oder konjunktiv verknüpfte) Menge **gelöster Formeln**

$$\exists \vec{x} : x_1 = t_1 \wedge \cdots \wedge \exists \vec{x} : x_k = t_k \wedge \forall \vec{x} : x_{k+1} \neq t_{k+1} \wedge \cdots \wedge \forall \vec{x} : x_n \neq t_n. \tag{1}$$

führen. Gelöst nennen wir eine solche Konjunktion von Gleichungen und Ungleichungen, wenn $x_1, \ldots, x_n$ verschiedene Variablen und $t_1, \ldots, t_n$ Normalformen sind und der transitive Abschluss von $\{(i, j) \mid t_i \text{ enthält } x_j\}$ nicht reflexiv ist.

Negation kommt im ganzen Beweis nur als Widerspruchsformel *False* oder in Gestalt von Komplementprädikaten vor (s. Def. 6.1.9). Das Negationssymbol ¬ taucht nirgends auf. Das schränkt die Menge möglicher Beweisziele nicht ein, weil jede prädikatenlogische Formel $\neg\varphi$ zu einer Formel ψ äquivalent ist, in der alle Negationssymbole direkt vor Atomen stehen, und ψ wiederum zu einer negationsfreien Formel mit Komplementprädikaten äquivalent ist. Man beachte, dass diese Transformation die Größe einer Formel nicht verändert. Verfahren zur Spezifikation von Komplementprädikaten liefern die Sätze 6.1.10 und 6.2.7.

Dieser Ansatz für ein Beweissystem entspricht dem natürlichen Beweisen. Wir gehen davon aus, dass zu beweisende Behauptungen als Implikationen gegeben sind, und wollen diese Form auch während der Ableitung beibehalten. Dies unterscheidet sich grundlegend von vielen Theorembeweisern, die nur stark **normalisierte Formeln** verarbeiten können. Ein Beispiel dafür ist die Normalisierung in die — von klassischen Beweisern verarbeiteten — **Klauseln**. Das sind quantorfreie Disjunktionen von Konjunktionen von **Literalen**. Ein Literal ist ein Atom $r(t)$ oder seine Negation $\neg r(t)$. Quantoren werden dabei i.w. durch **Skolemisierung** beseitigt, d.i. die wiederholte Überführung von Teilformeln $\forall x \exists y : \varphi(x, y)$ in **erfüllbarkeitsäquivalente** Formeln $\varphi(x, f(x))$, wobei f ein neu eingeführtes Funktionssymbol ist. Zwei Formeln φ und ψ sind erfüllbarkeitsäquivalent, wenn φ genau dann ein Modell hat, wenn ψ eins hat, wobei das eine oft ein Redukt des anderen ist (s. 4.1.2). Im Kontext einer konstruktorbasierten Spezifikation SP, deren Herbrandmodell $\forall x \exists y : \varphi(x, y)$ erfüllt, könnte man z.B. SP um das Axiom $f(x) \equiv y \Leftarrow \varphi(x, y)$ erweitern. Das Herbrandmodell der erweiterten Spezifikation würde

dann die skolemisierte Formel $\varphi(x, f(x))$ erfüllen. Die klassische Normalisierung prädikatenlogischer Formeln in Klauselform wird z.B. in [101], [31] und [74] ausführlich behandelt. Wir beschränken die Normalisierung auf die Anwendung elementarer Boolescher Äquivalenzen wie die Idempotenz von $\wedge$ und $\vee$ sowie die Entfernung von *True* und *False*:

$$\begin{array}{lllllll} \mathit{True} \Rightarrow \varphi & \Leftrightarrow & \mathit{True} \wedge \varphi & \Leftrightarrow & \mathit{False} \vee \varphi & \Leftrightarrow & \varphi \\ \mathit{False} \Rightarrow \varphi & \Leftrightarrow & \varphi \Rightarrow \mathit{True} & \Leftrightarrow & \mathit{True} \vee \varphi & \Leftrightarrow & \mathit{True} \\ \mathit{False} \wedge \varphi & \Leftrightarrow & \varphi \wedge \mathit{False} & \Leftrightarrow & \mathit{False} & & \end{array}$$

Absorptions- und Distributivgesetze werden schon nicht mehr automatisch angewendet, weil sie Formeln erheblich verändern und evtl. vergrößern können.

Klauseln sind selten die natürliche Darstellung eines Beweisziels. Sie sind maschinenorientiert und bieten sich daher für Inferenzsysteme an, die auf Anwendungsbereiche zugeschnitten sind, in denen Beweise in der Regel vollautomatisch geführt werden können, wie z.B. in der Schaltwerksverifikation oder, allgemeiner, der Analyse großer, aber endlicher Datenmengen. Demgegenüber sind unendliche Datenmengen, Zustandsräume, etc., meist nur mit induktiven Verfahren verifizierbar. Keines dieser Verfahren entspricht aber einem vollständigen Kalkül zur Ableitung aller gültigen Aussagen über den jeweiligen Datenbereich. Das liegt an der generellen Nichtaufzählbarkeit der induktiven Theoreme einer Spezifikation und zeigt sich beim praktischen Beweisen spätestens dann, wenn **Beweisziele verallgemeinert** oder passende **Induktionsordnungen** gefunden werden müssen (s. §7.2). Abhilfe schafft nur die **Interaktion** des automatischen Beweisers mit dem Benutzer. Dieser muss in den Ableitungsvorgang eingreifen und ihn in die "richtige Richtung" steuern können. Dazu müssen ihm neben der Ausgangsbehauptung auch die vom Beweiser erzeugten "subgoals" verständlich sein, was bei stark normalisierten Formeln selten der Fall ist.

Wir werden im folgenden versuchen, die Balance zwischen natürlichem und automatischem Beweisen zu halten. Einerseits sammeln und präzisieren wir die in Korrektheitsbeweisen *von Hand* häufig nur implizit verwendeten Regeln. Andererseits benutzen wir sie zur Erzeugung zahlreichen Beispielableitungen, die so aufgeschrieben sind, dass Möglichkeiten und Grenzen der Beweisautomatisierung erkennbar werden.

Ein Prinzip des natürlichen Schließens ist die weitgehende **Lokalisierung** von Beweisschritten. Man ersetzt wie beim Rewriting in den meisten Fällen nur Teilformeln. Die Korrektheit der lokalen Anwendung einer Beweisregel ist aber nur dann gewährleistet, wenn es sich bei der Regel um eine **Simplifikation** handelt, d.h. um eine Regel des Typs

$$\frac{\varphi}{\psi} \Updownarrow$$

Da Prämisse und Konklusion einer Simplifikation äquivalent sind, liefert ihre Anwendung auf eine beliebige Teilformel φ einer Formel $C[\varphi]$ einen korrekten Ableitungsschritt innerhalb eines top-down- (oder auch bottom-up-) Beweises:

$$\frac{\varphi}{\psi} \Updownarrow \quad \Longrightarrow \quad \frac{C[\varphi]}{C[\psi]} \Uparrow \tag{1}$$

Die Bezeichnung "Simplifikation" soll andeuten, dass die Anwendung Formeln syntaktisch vereinfacht. Typische Simplifikationen sind z.B. Reduktionen mit allen (!) Axiomen einer Standardfunktion oder -relation. Dann spricht man auch von **partieller Auswertung**.[43] Ist die zugrundeliegende Spezifikation SP funktional, dann liefert z.B. Satz 6.2.7 die folgenden Simplifikationsregeln für Ungleichungen:

Term-Splitting $\quad \dfrac{c(t_1, \ldots, t_n) \not\equiv c(u_1, \ldots, u_n)}{t_1 \not\equiv u_1 \vee \cdots \vee t_n \not\equiv u_n} \Updownarrow$ für alle Konstruktoren c von SP

Clash $\quad \dfrac{c(t) \not\equiv d(u)}{\mathit{True}} \Updownarrow$ für alle Paare (c, d) verschiedener Konstruktoren von SP

[43] "Partiell", weil der zu reduzierende Ausdruck Variablen enthalten kann.

Die "komplementären" Regeln dienen der Simplifikation von Gleichungen:[44]

Term-Splitting $\dfrac{c(t_1,\ldots,t_n) \equiv c(u_1,\ldots,u_n)}{t_1 \equiv u_1 \wedge \cdots \wedge t_n \equiv u_n} \Updownarrow$ für alle Konstruktoren c von SP

Clash $\dfrac{c(t) \equiv d(u)}{\mathit{False}} \Updownarrow$ für alle Paare (c,d) verschiedener Konstruktoren von SP

Auf eine Gleichung oder Ungleichung zwischen Normalformen ist immer Term-Splitting oder Clash anwendbar, es sei denn, eine der beiden Seiten der (Un-)Gleichung ist eine Variable. Dann kann sie u.U. mit einer der folgenden drei Regeln eliminiert werden:

Variablen-Entfernung

$$\frac{\exists x : (x \equiv t \wedge \varphi(x))}{\varphi(t)} \Updownarrow \qquad \frac{\forall x : (x \not\equiv t \vee \varphi(x))}{\varphi(t)} \Updownarrow$$

$$\frac{\forall x : ((x \equiv t \wedge \varphi(x)) \Rightarrow \psi(x))}{\varphi(t) \Rightarrow \psi(t)} \Updownarrow \qquad \frac{\forall x : (\varphi(x) \Rightarrow (x \not\equiv t \vee \psi(x)))}{\varphi(t) \Rightarrow \psi(t)} \Updownarrow$$

falls $x \notin \mathit{var}(t)$

Umgekehrt angewendet nennen wir diese Regeln **Variablen-Einführungen**.

☞ Folgern Sie die Korrektheit der dritten Regel aus der Korrektheit der zweiten!

Aussagenlogische Simplifikationen dienen oft der Entfernung oder Verkürzung von Implikationen:

Implikationsregeln

$$\frac{\varphi \wedge (\varphi \Rightarrow \psi)}{\varphi \wedge \psi} \Updownarrow \qquad \frac{\varphi_1 \Rightarrow (\varphi_2 \Rightarrow \psi)}{(\varphi_1 \wedge \varphi_2) \Rightarrow \psi} \Updownarrow$$

$$\frac{\varphi \Rightarrow (\psi_1 \wedge \psi_2)}{(\varphi \Rightarrow \psi_1) \wedge (\varphi \Rightarrow \psi_2)} \Updownarrow \qquad \frac{(\varphi_1 \vee \varphi_2) \Rightarrow \psi}{(\varphi_1 \Rightarrow \psi) \wedge (\varphi_2 \Rightarrow \psi)} \Updownarrow$$

Die erste Regel enthält den Modus ponens. Die zweite Regel beschreibt das "Currying" auf logischer Ebene. Die dritte und vierte Regel spalten eine Implikation auf und führen damit näher an die Subsumption (s.u.) heran.

Beliebige Termersetzungen sind i.a. nur dann korrekte Simplifikationen, wenn der ersetzte Term zum ursprünglichen äquivalent ist:

Termersetzung $\dfrac{\varphi(t) \wedge t \equiv u}{\varphi(u) \wedge t \equiv u} \Updownarrow$

Die folgenden Simplifikationen basieren auf Formelsubsumption, d.i. eine Relation zwischen Formeln φ und ψ, die hinreichend für die Gültigkeit der Implikation $\varphi \Rightarrow \psi$, aber im Gegensatz zu dieser entscheidbar ist:

Subsumption

$$\frac{\varphi \Rightarrow \psi}{\mathit{True}} \quad \frac{\varphi \Leftrightarrow \psi}{\psi \Rightarrow \varphi} \quad \frac{\psi \Leftrightarrow \varphi}{\psi \Rightarrow \varphi} \quad \frac{\varphi \wedge (\psi \Rightarrow \theta)}{\varphi \wedge \theta} \quad \text{falls } \varphi \ \psi \text{ subsumiert}$$

$$\frac{\varphi_1 \vee \cdots \vee \varphi_n}{\varphi_1 \vee \cdots \vee \varphi_{n-1}} \quad \text{falls } \varphi_1 \wedge \cdots \wedge \varphi_{n-1} \ \varphi_n \text{ subsumiert}$$

$$\frac{\varphi_1 \vee \cdots \vee \varphi_n}{\varphi_1 \vee \cdots \vee \varphi_{n-1}} \quad \text{falls } \varphi_n \ \varphi_1 \vee \cdots \vee \varphi_{n-1} \text{ subsumiert}$$

Wir definieren Formelsubsumption wie folgt induktiv. Sei $\sim$ die syntaktische Gleichheit von Formeln modulo der Umordnung von Argumenten permutativer Operatoren und θ eine Umbenennung von Variablen.

[44] Die Namen der wichtigsten Regeln sind oval gerahmt.

$$
\begin{array}{rcl}
\varphi \sim \psi & \Longrightarrow & \varphi \text{ subsumiert } \psi \\
\psi \text{ subsumiert } \varphi & \Longrightarrow & \neg\varphi \text{ subsumiert } \neg\psi \\
\varphi' \text{ subsumiert } \varphi \text{ und } \psi \text{ subsumiert } \psi' & \Longrightarrow & \varphi \Rightarrow \psi \text{ subsumiert } \varphi' \Rightarrow \psi' \\
\exists\ 1 \le i \le n : \varphi \text{ subsumiert } \psi_i & \Longrightarrow & \varphi \text{ subsumiert } \psi_1 \vee \cdots \vee \psi_n \\
\forall\ 1 \le i \le n : \varphi_i \text{ subsumiert } \psi & \Longrightarrow & \varphi_1 \vee \cdots \vee \varphi_n \text{ subsumiert } \psi \\
\forall\ 1 \le i \le n : \varphi \text{ subsumiert } \psi_i & \Longrightarrow & \varphi \text{ subsumiert } \psi_1 \wedge \cdots \wedge \psi_n \\
\exists\ 1 \le i \le n : \varphi_i \text{ subsumiert } \psi & \Longrightarrow & \varphi_1 \wedge \cdots \wedge \varphi_n \text{ subsumiert } \psi \\
\varphi(\vec{x}) \text{ subsumiert } \psi(\vec{x}) & \Longrightarrow & \exists\vec{x}\varphi(\vec{x}) \text{ subsumiert } \exists\vec{y}\psi(\vec{y}) \\
\varphi(\vec{x}) \text{ subsumiert } \psi(\vec{x}) & \Longrightarrow & \forall\vec{x}\varphi(\vec{x}) \text{ subsumiert } \forall\vec{y}\psi(\vec{y}) \\
\exists\ \theta, \vec{t} : \varphi\theta \sim \psi(\vec{t}) & \Longrightarrow & \varphi \text{ subsumiert } \exists\vec{x}\psi(\vec{x}) \\
\exists\ \theta, \vec{t} : \varphi(\vec{t}) \sim \psi\theta & \Longrightarrow & \forall\vec{x}\varphi(\vec{x}) \text{ subsumiert } \psi \\
\exists\ \theta, \vec{t}, \{i_1, \ldots, i_k\} \subset \{1, \ldots, n\} : (\varphi_{i_1} \wedge \cdots \wedge \varphi_{i_k})\theta \sim \psi(\vec{t}) & \Longrightarrow & \varphi_1 \wedge \cdots \wedge \varphi_n \text{ subsumiert } \exists\vec{x}\psi(\vec{x}) \\
\exists\ \theta, \vec{t}, \{i_1, \ldots, i_k\} \subset \{1, \ldots, n\} : \varphi(\vec{t}) \sim (\psi_{i_1} \vee \cdots \vee \psi_{i_k})\theta & \Longrightarrow & \forall\vec{x}\varphi(\vec{x}) \text{ subsumiert } \psi_1 \vee \cdots \vee \psi_n
\end{array}
$$

Z.B. wird $\exists y : x < y$ von $x \ge 0 \wedge x < y + 1$ subsumiert (siehe Beispiel 5.1.5).

Die o.a. Definition der Formelsubsumption deckt nicht alle Möglichkeiten ab, die Gültigkeit von $\varphi \Rightarrow \psi$ zu entscheiden. Z.B. könnte man die Bedingung

$$\exists\ \theta, \vec{t}, \{i_1, \ldots, i_k\} \subset \{1, \ldots, n\} : \varphi(\vec{t}) \sim (\psi_{i_1} \vee \cdots \vee \psi_{i_k})\theta$$

in der letzten Zeile der Definition zu

$$\exists\ \{i_1, \ldots, i_k\} \subset \{1, \ldots, n\} : \varphi \text{ subsumiert } \psi_{i_1} \vee \cdots \vee \psi_{i_k}$$

verallgemeinern. Das würde die Berechnung von "φ subsumiert ψ" erheblich verlangsamen, weil $2^n - n - 1$ rekursive Aufrufe der Relation "subsumiert" hinzukämen!

☞ Zeigen Sie die Korrektheit der Subsumptionsregeln!

Im Prinzip können Simplifikationen immer dann angewendet werden, wenn sie überhaupt anwendbar sind. Da sie die Bedeutung einer Formel nicht verändern, kann ihre Anwendung in keine *zusätzlichen* Beweis-Sackgassen führen. Das gilt allerdings nicht mehr, wenn andere für den Beweis relevante Regeln unsimplifizierte **Redices**[45] benötigen. Das gilt insbesondere für **Induktionsschritte**, wo Regeln nur auf Induktionshypothesen einer bestimmten syntaktischen Struktur und nicht auf beliebige äquivalente Formeln angewendet werden können. Die o.g. Simplifikationen sind auch in dieser Hinsicht meistens sicher und laufen deshalb in Beweisern wie KIV [97] und Expander2 [81] im Hintergrund ab, d.h. der Benutzer bekommt i.a. nur simplifizierte Formeln zu sehen.

Die Korrektheit der folgenden **Entfaltungsregeln** (**unfolding**) ergibt sich aus den in §6.1 erwähnten induktiven Theoremen

$$r(x) \iff \bigvee_{i=1}^{n} \exists X_i : (x \equiv t_i \wedge \varphi_i), \tag{2}$$

$$q(x) \iff \bigwedge_{i=1}^{n} \forall X_i : (x \equiv t_i \Rightarrow \varphi_i), \tag{3}$$

ie alle Axiome $r(t_1) \Leftarrow \varphi_1, \ldots, r(t_n) \Leftarrow \varphi_n$ für ein logisches Prädikat r bzw. alle Axiome $q(t_1) \Rightarrow \varphi_1, \ldots, q(t_n) \Rightarrow \varphi_n$ für ein Coprädikat q zusammenfassen, wobei X_i die Menge der Variablen von $r(t_i) \Leftarrow \varphi_i$ bzw. $q(t_i) \Rightarrow \varphi_i$, $1 \le i \le n$, ist.

[45] So nennt man die größten Teilformeln, die von der Regelanwendung tatsächlich betroffen sind.

Entfaltung von r
$$\frac{r(t)}{\exists X_{i_1}(t \equiv t_{i_1} \wedge \varphi_{i_1}) \vee \cdots \vee \exists X_{i_k}(t \equiv t_{i_k} \wedge \varphi_{i_k})} \Updownarrow$$
falls $t \equiv t_j$ für alle $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$ unlösbar ist (z.B. zum Clash führt)

Entfaltung von q
$$\frac{q(t)}{\forall X_{i_1}(t \equiv t_1 \Rightarrow \varphi_{i_1}) \wedge \cdots \wedge \forall X_{i_k}(t \equiv t_{i_k} \Rightarrow \varphi_{i_k})} \Updownarrow$$
falls $t \equiv t_j$ für alle $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$ unlösbar ist (z.B. zum Clash führt)

Der Grenzfall $n = 0$ liefert Regeln zur Entfernung logischer Prädikate bzw. Coprädikate:

Entfernung von r $\dfrac{r(t)}{\mathit{False}} \Updownarrow$ falls für alle Axiome $r(u) \Leftarrow \varphi$ für r, $t \equiv u$ unlösbar ist

Entfernung von q $\dfrac{q(t)}{\mathit{True}} \Updownarrow$ falls für alle Axiome $q(u) \Rightarrow \varphi$ für q, $t \equiv u$ unlösbar ist

☞ Zeigen Sie die Korrektheit dieser Regeln!

Aus (2) und Satz 5.1.20 folgt das (2) entsprechende Theorem

$$f(x) \equiv y \iff \bigvee_{i=1}^{n} \exists X_i : (x \equiv t_i \wedge y \equiv u_i \wedge \varphi_i), \tag{4}$$

das alle Axiome $f(t_1) \equiv u_i \Leftarrow \varphi_1, \ldots, f(t_n) \equiv u_n \Leftarrow \varphi_n$ für eine definierte Funktion f zusammenfaßt. X_i ist die Menge der Variablen von $f(t_i) \equiv u_i \Leftarrow \varphi_i$. (3) impliziert die Korrektheit der Entfaltungsregel für Terme:

Entfaltung von f
$$\frac{r(\ldots, f(t), \ldots)}{\exists X_{i_1}(t \equiv t_{i_1} \wedge \varphi_{i_1} \wedge r(\ldots, u_{i_1}, \ldots)) \vee \cdots \vee \exists X_{i_k}(t \equiv t_{i_k} \wedge \varphi_{i_k} \wedge r(\ldots, u_{i_k}, \ldots))} \Updownarrow$$
falls $t \equiv t_j$ für alle $j \in \{1, \ldots, n\} \setminus \{i_1, \ldots, i_k\}$ unlösbar ist (z.B. zum Clash führt)

☞ Geben Sie eine nicht-funktionale Spezifikation SP and und zeigen Sie, dass (3) kein induktives Theorem von SP ist!

Mit einer Anwendung einer Entfaltungsregel wird eine Disjunktion erzeugt, die eine zur Menge der Axiome für r bzw. f korrespondierende **Fallunterscheidung** darstellt. Das Ziel der wiederholten Anwendung ist die Entfernung von r bzw. f aus einer gegebenen Formel.[46] Übrig bleiben Gleichungen zwischen Normalformen, die dann mit Term-Splitting, Clash und Variablen-Entfernung eliminiert werden.

Die Anwendung einer Entfaltung kann man auch als parallele Resolution bzw. paralleles Narrowing mit allen Axiomen für ein Prädikat bzw. eine Funktion auffassen, wobei die zu Resolutions- bzw. Narrowingschritten gehörenden Term-Unifikationen (s. Def. 5.1.21) durch die Erzeugung der Gleichungen $t \equiv t_1, \ldots, t \equiv t_n$ (s.o.) und deren spätere Lösung ersetzt werden. Da die Lösung letzten Endes wieder auf Term-Unifikation hinausläuft, ist es effizienter, Entfaltung durch passende Varianten von Resolution bzw. Narrowing zu implementieren.[47] Um die Zahl der erzeugten Summanden noch weiter einzuschränken, werden auch *bewachte* Hornformeln verarbeitet (siehe §4.2):

(bewachte) Resolution
$$\frac{r(t)}{\bigvee_{i=1}^{k} \exists Z_i : (\vec{x} \equiv \vec{x}\sigma_i \wedge \varphi_i\sigma_i)} \Updownarrow$$

[46] Das gelingt natürlich nur, wenn die Axiome für r bzw. f nicht rekursiv sind.
[47] Expander2 nennt beide Narrowing.

falls $\gamma_1 \Rightarrow (r(u_1) \Longleftarrow \varphi_1), \ldots, \gamma_n \Rightarrow (r(u_n) \Longleftarrow \varphi_n)$ die Axiome für r sind, $\vec{x}$ eine Liste der Variablen von t ist, für alle $1 \leq i \leq k$ $t\sigma_i = u_i\sigma_i$, $\gamma_i\sigma_i \vdash \mathit{True}$ und $Z_i = var(u_i, \varphi_i)$ gilt, und für alle $k < i \leq n$ t nicht mit u_i unifizierbar ist

(bewachte) Coresolution

$$\frac{q(t)}{\bigwedge_{i=1}^{k} \forall Z_i : (\vec{x} \equiv \vec{x}\sigma_i \Rightarrow \varphi_i\sigma_i)} \Updownarrow$$

falls $\gamma_1 \Rightarrow (q(u_1) \Longrightarrow \varphi_1), \ldots, \gamma_n \Rightarrow (q(u_n) \Longrightarrow \varphi_n)$ die Axiome für q sind, $\vec{x}$ eine Liste der Variablen von t ist, für alle $1 \leq i \leq k$ $t\sigma_i = u_i\sigma_i$, $\gamma_i\sigma_i \vdash \mathit{True}$ und $Z_i = var(u_i, \varphi_i)$ gilt, und für alle $k < i \leq n$ t nicht mit u_i unifizierbar ist

(bewachtes) Narrowing

$$\frac{r(\ldots, f(t), \ldots)}{\begin{array}{l}\bigvee_{i=1}^{k} \exists Z_i : (r(\ldots, v_i\sigma_i, \ldots) \wedge \vec{x} \equiv \vec{x}\sigma_i \wedge \varphi_i\sigma_i) \vee \\ \bigvee_{i=k+1}^{l} (r(\ldots, f(t\sigma_i), \ldots) \wedge \vec{x} \equiv \vec{x}\sigma_i)\end{array}} \Updownarrow$$

falls $\gamma_1 \Rightarrow (f(u_1) \equiv v_1) \Longleftarrow \varphi_1, \ldots, \gamma_n \Rightarrow (f(u_n) \equiv v_n) \Longleftarrow \varphi_n$ die Axiome für f sind, $\vec{x}$ eine Liste der Variablen von t ist, für alle $1 \leq i \leq k$, $t\sigma_i = u_i\sigma_i$, $\gamma_i\sigma_i \vdash \mathit{True}$ und $Z_i = var(u_i, \varphi_i)$ gilt, für alle $k < i \leq l$, σ_i ein partieller Unifikator of t und u_i ist, und für alle $l < i \leq n$, t nicht mit u_i partiell unifizierbar ist.

Beim Narrowing kann die Unifikation von t mit u_i scheitern, weil t eine definierte Funktion enthält, während an derselben Stelle in u_i ein Konstruktor steht. Da die Unifikation aber später erfolgreich sein kann, nachdem weitere Narrowing-Schritte f aus t entfernt haben, simuliert diese Narrowing-Regel bezüglich der Anwendung der Axiome $\gamma_{k+1} \Rightarrow (f(u_{k+1}) \equiv v_{k+1}) \Longleftarrow \varphi_{k+1}, \ldots, \gamma_l \Rightarrow (f(u_l) \equiv v_l) \Longleftarrow \varphi_l$ die pre-Narrowing-Regel aus §5.1: wir bilden den partiellen Unifikator σ_i und ersetzen die Redexformel $\varphi(f(t))$ durch ihre σ_i-Instanz und die Gleichungen, die σ_i repräsentieren.

Inferenzregeln des Typs

$$\frac{\varphi}{\psi} \Uparrow \quad \text{bzw.} \quad \frac{\varphi}{\psi} \Downarrow$$

nennen wir **Expansionen** bzw. **Kontraktionen**. Simplifikationen sind demnach sowohl Expansionen als auch Kontraktionen. φ **expandiert** bzw. **kontrahiert** zu ψ, falls eine Folge von Anwendungen von Expansionen bzw. Kontraktionen zu ψ führt. Leider lassen sich echte Expansionen und Kontraktionen nicht auf beliebige Teilformeln eines Beweiszieles anwenden, d.h. die Implikation (1) gilt nicht, wenn man $\Updownarrow$ durch $\Uparrow$ oder $\Downarrow$ ersetzt. Sie gilt nur, wenn der Kontext keine Implikationen enthält:

$$\frac{\varphi}{\psi} \Uparrow \implies \frac{C[\varphi]}{C[\psi]} \Uparrow \qquad \frac{\varphi}{\psi} \Downarrow \implies \frac{C[\varphi]}{C[\psi]} \Downarrow \qquad \text{falls } C \text{ implikationsfrei ist} \tag{4}$$

Im Kontext einer Implikation kehrt sich die Folgerbarkeitsrichtung (senkrechter Pfeil) möglicherweise um, je nachdem, ob die Prämisse oder die Konklusion der umfassenden Implikation transformiert wird:

$$\frac{\varphi}{\psi} \Uparrow \quad \Longrightarrow \quad \frac{\varphi \Rightarrow \varphi'}{\psi \Rightarrow \varphi'} \Downarrow \ \wedge \ \frac{\varphi' \Rightarrow \varphi}{\varphi' \Rightarrow \psi} \Uparrow \tag{5}$$

$$\frac{\varphi}{\psi} \Downarrow \quad \Longrightarrow \quad \frac{\varphi \Rightarrow \varphi'}{\psi \Rightarrow \varphi'} \Uparrow \ \wedge \ \frac{\varphi' \Rightarrow \varphi}{\varphi' \Rightarrow \psi} \Downarrow \tag{6}$$

☞ Zeigen Sie die Korrektheit von (5) und (6)!

Die Anwendbarkeit einer echten Expansion oder Kontraktion auf eine Teilformel φ eines Beweisziels ψ wird demnach mitbestimmt von der *Polarität* der Position von φ innerhalb von ψ, die folgendermaßen definiert ist: Sei $\mathcal{F}$ die Menge aller Σ-Formeln.

$$\mathit{polarity} : \mathbb{N}^* \times \mathcal{F} \to \{+, -\} \qquad f : \mathbb{N}^* \times \mathcal{F} \times \{+, -\} \to \{+, -\}$$

$$
\begin{aligned}
&\mathit{polarity}(w,\psi) \;=\; f(w,\psi,+)\\
&f(\varepsilon,\varphi,\pm) \;=\; \pm\\
&f(0w,\varphi\Rightarrow\psi,\pm) \;=\; f(w,\varphi,\mp)\\
&f(1w,\varphi\Rightarrow\psi,\pm) \;=\; f(w,\psi,\pm)\\
&f(0w,\varphi\odot\psi,\pm) \;=\; f(w,\varphi,\pm) \quad \text{for all } \odot\in\{\wedge,\vee\}\\
&f(1w,\varphi\odot\psi,\pm) \;=\; f(w,\psi,\pm) \quad \text{for all } \odot\in\{\wedge,\vee\}\\
&f(0w,\neg\varphi),\pm) \;=\; f(w,\varphi,\mp)\\
&f(0w,\forall x\varphi),\pm) \;=\; f(w,\varphi,\pm) \quad \text{for all } x\in X\\
&f(0w,\exists x\varphi),\pm) \;=\; f(w,\varphi,\pm) \quad \text{for all } x\in X
\end{aligned}
$$

$\mathit{polarity}(w,\psi)$ nennen wir die **Polarität von** w **in** ψ. Die Teilformel an der Position w heisst **positiv** bzw. **negativ polarisiert**, wenn $\mathit{polarity}(w,\psi)$ den Wert $+$ bzw. $-$ hat. Dieser Begriff liefert die Einschränkung, unter der (1) von Simplifikationen auf Expansionen bzw. Kontraktionen übertragen werden kann: Sei w die Position von φ in $C[\varphi]$.[48]

$$\frac{\varphi}{\psi}\Uparrow \;\wedge\; \mathit{polarity}(w,C[\varphi])=+ \quad\Longrightarrow\quad \frac{C[\varphi]}{C[\psi]}\Uparrow \tag{7}$$

$$\frac{\varphi}{\psi}\Downarrow \;\wedge\; \mathit{polarity}(w,C[\varphi])=- \quad\Longrightarrow\quad \frac{C[\varphi]}{C[\psi]}\Uparrow \tag{8}$$

Da wir hier nur top-down-Beweise führen, genügt uns die Herleitung von Expansionen aus Expansionen oder Kontraktionen.

☞ Zeigen Sie die Korrektheit von (7) und (8) durch Induktion über die Größe von C und unter Verwendung von (5) und (6)!

Der Entfernung von Quantoren bzw. Gleichungen dienen folgende Expansionen bzw. Kontraktionen:

Quantor-Entfernung $\quad \dfrac{\exists x:\varphi(x)}{\varphi(t)}\Uparrow \qquad \dfrac{\forall x:\varphi(x)}{\varphi(t)}\Downarrow$

Gleichungs-Entfernung $\quad \dfrac{t\equiv u\wedge\varphi(t)}{\varphi(u)}\Downarrow \qquad \dfrac{(t\equiv u\wedge\varphi(t)) \;\Rightarrow\; \psi(t)}{\varphi(u) \;\Rightarrow\; \psi(u)}\Uparrow$

Eher zur interaktiven Anwendung gedacht sind auch die folgenden Regeln, mit denen eine bereits bewiesene Implikation der Form

$$at_1\wedge\cdots\wedge at_n \;\Leftarrow\; \mathit{prem} \tag{9}$$

$$at_1\vee\cdots\vee at_n \;\Leftarrow\; \mathit{prem} \tag{10}$$

$$at_1\wedge\cdots\wedge at_n \;\Rightarrow\; \mathit{conc} \tag{11}$$

$$at_1\vee\cdots\vee at_n \;\Rightarrow\; \mathit{conc} \tag{12}$$

uf eine Disjunktion bzw. eine Konjunktion angewendet werden kann. Üblicherweise sind $at_1,\ldots,at_n$ atomare Formeln. Es können aber auch beliebige Σ-Formeln sein. Expander2 [82, 81] schließt erlaubt allerdings keine Quantoren in $at_1,\ldots,at_n$, weil das die Suche nach korrekten Redices erheblich komplizieren würde.

konjunktive Resolution Anwendung von (9)

$$\frac{\varphi_1(at_1')\wedge\cdots\wedge\varphi_n(at_n')}{(\bigwedge_{i=1}^n\varphi_i(\exists\, V(\mathit{prem}\sigma\wedge\bigwedge_{x\in dom(\sigma)}x\equiv x\sigma)))}\Uparrow$$

wobei für alle $1\le i\le n$ $at_i'\sigma=at_i\sigma$ gilt und in φ_i weder Existenzquantoren noch Negations- oder Implikationssymbole vorkommen.

[48] Wie sich aus der Definition von *polarity* ergibt, wird $C[\varphi]$ bei der Positionsbestimmung von φ als Term über zwei- bzw. einstelligen Funktionssymbolen $\Rightarrow$, $\wedge$, $\vee$, $\forall x$ und $\exists x$ aufgefaßt.

disjunktive Resolution Anwendung von (10)

$$\frac{\varphi_1(at'_1) \vee \cdots \vee \varphi_n(at'_n)}{(\bigwedge_{i=1}^{n} \varphi_i(\exists\ V(prem\sigma \wedge \bigwedge_{x \in dom(\sigma)} x \equiv x\sigma)))} \Uparrow$$

wobei für alle $1 \leq i \leq n$ $at'_i\sigma = at_i\sigma$ gilt und in φ_i weder Allquantoren noch Negations- oder Implikationssymbole vorkommen.

konjunktive Coresolution Anwendung von (11)

$$\frac{\varphi_1(at'_1) \wedge \cdots \wedge \varphi_n(at'_n)}{(\bigvee_{i=1}^{n} \varphi_i(\forall\ V(\bigwedge_{x \in dom(\sigma)} x \equiv x\sigma \Rightarrow conc\sigma)))} \Downarrow$$

wobei für alle $1 \leq i \leq n$ $at'_i\sigma = at_i\sigma$ gilt und in φ_i weder Existenzquantoren noch Negations- oder Implikationssymbole vorkommen.

disjunktive Coresolution Anwendung von (12)

$$\frac{\varphi_1(at'_1) \wedge \cdots \wedge \varphi_n(at'_n)}{(\bigvee_{i=1}^{n} \varphi_i(\forall\ V(\bigwedge_{x \in dom(\sigma)} x \equiv x\sigma \Rightarrow conc\sigma)))} \Downarrow$$

wobei für alle $1 \leq i \leq n$ $at'_i\sigma = at_i\sigma$ gilt und in φ_i weder Allquantoren noch Negations- oder Implikationssymbole vorkommen.

V ist die Menge der Variablen von $prem$, die nicht in $at_1, \ldots, at_n$ auftreten. Während Resolution auf ein Atom angewendet wird (s. Def. 4.2.12 oder Def. 5.1.21), kann man mit den o.g. Regeln n Atome gleichzeitig ersetzen. Ausserdem wird hier die jeweils erzeugte Substitution als Konjunktion

$$\bigwedge_{x \in dom(\sigma)} x \equiv x\sigma$$

von Gleichungen wiedergegeben.

Die Regeln sind korrekt, wenn (9)-(12) Lemmas sind, d.h. ein induktives Theorem der zugrundeliegenden Spezifikation SP ist oder als solches vorausgesetzt wird. So lassen sich mit diesen Regeln — wie bei manuellen Beweisen üblich — Ableitungen strukturieren und man muss nicht jeden Beweis auf die Anwendung von Axiomen, also die o.a. Entfaltungsregeln beschränken. Ist SP funktional, dann ist z.B. die aus einer endlichen Menge von Normalformen gebildeten Fallunterscheidungen der Form

$$\exists X_1 x \equiv t_1 \vee \cdots \vee \exists X_n x \equiv t_n, \tag{13}$$

ein gültiges Lemma, falls für alle $1 \leq i \leq n$, t_i eine Normalform ist, X_i die Menge der Variablen von t_i ist, diese alle dieselbe Sorte wie x haben, x darin aber nicht vorkommt, und für alle $t \in NF_\Sigma$ ein $1 \leq i \leq n$ mit $t_i \leq t$ existiert (s. Def. 5.2.4). Einfachstes Beispiel (s. Bsp. 5.1.3):

$$x \equiv 0 \vee \exists\ y : x \equiv suc(y).$$

Leider enthält ein solches Lemma i.d.R. auf der linken Seite Quantoren und wäre deshalb von Expander2 nicht anwendbar. Tatsächlich kann man aber jeden Beweis einer Formel $\varphi(x)$, in dem das Lemma (13) verwendet wird, durch einen Beweis der Formel $\varphi(x) \Leftarrow P(x)$ ersetzen, in dem anstelle von (13) die folgenden Axiome für das neue Prädikat P angewendet werden:

$$P(t_i) \Leftarrow P(x_{i1}) \wedge \cdots \wedge P(x_{in_i}), \quad 1 \leq i \leq n, \tag{14}$$

wobei o.B.d.A. vorausgesetzt wird, dass $X_i = \{x_{i1}, \ldots, x_{in_i}\}$ aus Variablen der Sorte von t_i besteht. Im Beispiel:

$$P(0), \qquad P(suc(x)) \Leftarrow P(x).$$

☞ Bilden Sie entsprechende Axiome für die Sorten aus Bsp. 5.1.4 und Bsp. 5.1.5!

Wie aus den Regeln hervorgeht, sind nicht nur Instanzen der Konklusion bzw. Pramisse von (9)-(12) mögliche Redices, sondern die Atome $at_1, \ldots, at_n$ können in verschiedenen, aber nicht ganz beliebigen Formelkontexten $\varphi_1, \ldots, \varphi_n$ auftreten. Wir zeigen für die konjunktive und die disjunktive Resolution, wie die jeweilige Kontextbedingung die Korrektheit der Regeln impliziert.

Konjunktive Resolution ist offenbar korrekt bzgl. jeder Σ-Struktur mit Gleichheit, wenn aus (9) die Gültigkeit von

$$\varphi_1(at_1) \wedge \cdots \wedge \varphi_n(at_n) \quad \Leftarrow \quad \bigwedge_{i=1}^{n} \varphi_i(\exists\ V\ prem) \tag{15}$$

folgt. Wegen der Monotonie von φ_i folgt

$$\bigwedge_{i=1}^{n} \varphi_i(at_1 \wedge \cdots \wedge at_n) \tag{16}$$

aus der Prämisse von (15). (16) wiederum impliziert

$$\bigwedge_{i=1}^{n} (\varphi_i(at_1) \wedge \cdots \wedge \varphi_i(at_n)), \tag{17}$$

weil, wie man durch Induktion über φ zeigen kann,

$$\varphi(at_1 \wedge \cdots \wedge at_n) \quad \Longleftrightarrow \quad \varphi(at_1) \wedge \cdots \wedge \varphi(at_n) \tag{18}$$

gilt, sofern sich die logischen Operatoren von φ auf $\vee$, $\wedge$ und $\forall$ beschränken. (17) impliziert die Konklusion von (15). Also ist die konjunktive Resolution korrekt.

Disjunktive Resolution ist offenbar korrekt bzgl. jeder Σ-Struktur mit Gleichheit, wenn aus (10) die Gültigkeit von

$$\varphi_1(at_1) \vee \cdots \vee \varphi_n(at_n) \quad \Leftarrow \quad \bigwedge_{i=1}^{n} \varphi_i(\exists\ V\ prem) \tag{19}$$

folgt. Wegen der Monotonie von φ_i folgt

$$\bigwedge_{i=1}^{n} \varphi_i(at_1 \vee \cdots \vee at_n) \tag{20}$$

aus der Prämisse von (19). (20) wiederum impliziert

$$\bigwedge_{i=1}^{n} (\varphi_i(at_1) \vee \cdots \vee \varphi_i(at_n)), \tag{21}$$

weil, wie man durch Induktion über φ zeigen kann,

$$\varphi(at_1 \vee \cdots \vee at_n) \quad \Longleftrightarrow \quad \varphi(at_1) \vee \cdots \vee \varphi(at_n) \tag{22}$$

gilt, sofern sich die logischen Operatoren von φ auf $\wedge$, $\vee$ und $\exists$ beschränken. Durch Distribution von $\wedge$ über $\vee$ wird (21) zu:

$$\bigvee_{1 \leq i_1, \ldots, i_n \leq n} (\varphi_1(at_{i_1}) \wedge \cdots \wedge \varphi_n(at_{i_n})),$$

was schließlich zur Konklusion von (19) führt. Also ist die disjunktive Resolution korrekt.

☞ Zeigen Sie die Gültigkeit von (18) und (22) unter den genannten Kontextbedingungen durch Induktion über die Größe von φ.

☞ Zeigen Sie, dass auch konjunktive und disjunktive Coresolution bzgl. jeder Σ-Struktur mit Gleichheit korrekt sind, indem sie die Beweise der Korrektheit konjunktiver und disjunktiver Resolution dualisieren!

7.2 Induktion

Aussagen über rekursiv definierte Funktionen und Prädikate auf unendlichen Datenbereichen lassen sich i.a. nur induktiv beweisen. Rekursion in den Axiomen für ein Signatursymbol f findet sich i.a. bei der Verifikation von f als Induktion wieder. Die Mengen, über die induziert wird, bestehen i.d.R. aus Normalformen. Ist SP funktional — was wir schon in §7.1 für die Korrektheit wichtiger Regeln wie Term-Splitting und Clash vorausgesetzt haben —, dann hat jeder Grundterm genau eine SP-äquivalente Normalform, so dass man nur noch über Normalformen induzieren muss. Leider läßt sich jedoch die Menge möglicher Induktionsschemata (genauer: wohlfundierter Normalformordnungen, *entlang derer* induziert wird) nicht so auf endlich viele begrenzen, dass damit alle im Herbrandmodell gültigen Aussagen beweisbar wären. Insbesondere beim Beweis von Aussagen über mehrere Funktionen oder Prädikate ist es oft erforderlich, die Rekursionsschemata der Axiome trickreich zu einer passenden Induktionsordnung zu kombinieren. Die Menge der induktiven Theoreme von SP ist — wie oben bereits erwähnt wurde — nicht aufzählbar. M.a.W. es gibt keinen bzgl. $Her(SP)$ vollständigen Kalkül. Das widerspricht nicht der Vollständigkeit von Regelsystemen für die Prädikatenlogik wie den bekannten **Hilbert-** und **Tableau-Kalkülen**. Dort geht es nämlich um die Gültigkeit in *allen* Modellen von SP, also auch in solchen, deren Trägermengen keinen induktiven Aufbau haben und daher auch keine induktiven Beweise ihrer Eigenschaften erlauben.

Prinzipiell lassen sich drei Ansätze zum Beweis induktiver Theoreme unterscheiden:

- **Noethersche Induktion** mit dem Spezialfall **struktureller Induktion**,
- **implizite Induktion** oder **induktive Vervollständigung**,
- **Fixpunktinduktion** und **Coinduktion**.

Noethersche Induktion bezeichnet den — naheliegendsten — Ansatz, eine Behauptung $\forall x \varphi(x)$ dadurch zu beweisen, dass man eine wohlfundierte Ordnung $\ll$ auf $Her(SP)_{sort(x)}$ angibt und $\varphi(x)$ herleitet unter der Annahme, dass $\varphi(y)$ für alle $y \ll x$ gilt. Demnach ist Noethersche Induktion die Anwendung folgender Expansion:

Noethersche Induktion $$\frac{\forall x\ \varphi(x)}{\forall x\ (\forall y\ (y \ll x\ \Rightarrow\ \varphi(y))\ \Rightarrow\ \varphi(x))}\ \Updownarrow$$

Die Gültigkeit der $\Uparrow$-Richtung ergibt sich wie folgt:[49] Es gelte der Sukzedent der Induktionsregel in $Her(SP)$. Sei A die Menge aller $x \in Her(SP)_{sort(x)}$, die $\varphi(x)$ nicht erfüllen. Wir nehmen an, dass A nichtleer ist. Da $\ll$ wohlfundiert ist, hat A ein bzgl. $\ll$ minimales Element x_0. Also gilt $\varphi(y)$ für alle $y \in Her(SP)_{sort(x)}$ mit $y \ll x_0$, d.h. $x = x_0$ erfüllt die Prämisse $\forall y\ (y \ll x\ \Rightarrow\ \varphi(y))$ der äußeren Implikation des Sukzedenten. Wegen der Gültigkeit des Sukzedenten in $Her(SP)$ erfüllt daher $x = x_0$ auch die Konklusion $\varphi(x)$, was aber $x_0 \in A$ widerspricht. Also ist A leer, d.h. alle $x \in Her(SP)_{sort(x)}$ erfüllen $\varphi(x)$, m.a.W.: der Antezedent der Induktionsregel gilt in $Her(SP)$.

Man beachte, dass die Konklusion der Noetherschen Induktion die üblicherweise geforderte Induktionsverankerung enthält. Für alle bzgl. $\ll$ minimalen Elemente x ist die Prämisse des Sukzedenten der Induktionsregel nämlich eine Tautologie und damit der Sukzedent äquivalent zu $\varphi(x)$. Für die minimalen Elemente muss man φ demnach direkt zeigen.

[49] Wir folgen dem Beweis von [65], Theorem 5-10.

In ihrer allgemeinen Form ist Noethersche Induktion kaum von einem automatischer Beweiser einsetzbar, denn es fehlt jeglicher Ansatz zu einer konkreten Definition von $\ll$. Eine Anforderung an $\ll$ haben wir hier allerdings schon eingebaut. Indem die Wohlfundiertheit *der Interpretation von* $\ll$ im Herbrandmodell von SP verlangt wird, setzen wir nämlich voraus, dass $\ll$ ein Prädikat von SP ist.

Das bekannteste Beispiel Noetherscher Induktion ist die **strukturelle Induktion**, bei der Grundterme nach ihrer Größe geordnet werden. Sei $\#t$ die Anzahl der Symbole von t.

$$t \ll^{Her(SP)} t' \iff \#t < \#t'.$$

Gibt es auch nur zwei SP-äquivalente Grundterme t, t' unterschiedlicher Größe, dann gilt diese Äquivalenz jedoch für kein Prädikat $\ll$, dessen Interpretation in $Her(SP)$ eine wohlfundierte Relation ist. Wegen der impliziten Kongruenzaxiome von SP muss $\equiv_{SP}$ nämlich mit $\ll$ verträglich sein. Dann aber würde aus $\#t < \#t'$ und $t \equiv_{SP} t'$ $t' \ll^{Her(SP)} t'$ folgen und $\ll^{Her(SP)}$ wäre nicht wohlfundiert.

Strukturelle Induktion funktioniert erst bei funktionalen Spezifikationen, wo sich Grundterme nach der Größe ihrer Normalformen ordnen lassen:

$$t \ll^{Her(SP)} t' \iff \#(nf(t)) < \#(nf(t')). \tag{1}$$

☞ Erweitern Sie eine beliebige funktionale Spezifikation um Axiome für $\ll$ derart, dass (1) gilt!

Ist x im Beweisziel $\forall x \varphi(x)$ ein Tupel mehrerer Variablen, dann muss $\ll$ auf Termtupel fortgesetzt werden. Da gibt es bereits viele Möglichkeiten, aus denen man eine passende auswählen muss. Häufig eignet sich die lexikographische Erweiterung von (1) (vgl. Bsp. 7.2.2):

$$(t_1, \ldots, t_n) \ll^{Her(SP)} (t'_1, \ldots, t'_n) \iff (\#(nf(t_1)), \ldots, \#(nf(t_n)) <_{lex} (\#(nf(t'_1)), \ldots, \#(nf(t'_n)). \tag{2}$$

☞ Erweitern Sie Ihre Spezifikation von $\ll$ so, dass auch (2) gilt!

$\#$ ist eine spezielle **Gewichtsfunktion**. Eine Gewichtsfunktion w ordnet Grundtermen Elemente einer Menge A mit wohlfundierter Ordnung $<$ zu. Häufig ist $A = \mathbb{N}$ und $<$ die übliche "kleiner"-Relation. Dann wird $>$ von w auf Grundnormalformen "projiziert":

$$t \ll^{Her(SP)} t' \iff w(nf(t)) < w(nf(t')). \tag{3}$$

Die Spezifikation von $\ll$ besteht hier i.w. aus geeigneten Axiomen für w.

Theorembeweiser wie Isabelle [88, 75] und der Larch Prover [43] stellen Beweisregeln für strukturelle Induktion zur Verfügung. Der Boyer-Moore Prover [19], QuodLibet [9] und Expander2 [81] erlauben auch andere Formen Noetherscher Induktion. CLAM [22] und INKA [51] steuern Beweise mithilfe sog. *wave rules* in Richtung auf Formeln, die als Induktionshypothesen interpretiert werden können. Die Tatsache, dass ohne vorherige Ableitung von Induktionshypothesen keine Induktionsregeln angewendet werden können, ist neben der geeigneten Wahl von $\ll$ ein weiterer Grund dafür, dass die Benutzer induktiver Beweiser mit diesen interagieren müssen.

Mit Expander2 [81] kann man Noethersche Induktionsbeweise wie folgt führen. Nach Auswahl der beweisenden Formel, i.a. eine Implikation $prem \Rightarrow conc$, werden Variablen $x_1, \ldots, x_n$ angeklickt, die damit als Induktionsvariablen deklariert sind. Ausserdem erzeugt das System zwei Lemmas, die Induktionshypothesen entsprechen:

$$conc' \iff (x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n) \ \wedge \ prem' \tag{4}$$

$$prem' \implies ((x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n) \ \Rightarrow \ conc') \tag{5}$$

$conc'$ und $prem'$ werden aus $conc$ bzw. $prem$ gebildet, indem jedes Vorkommen einer Induktionsvariablen x durch x' ersetzt wird. Die **Abstiegsbedingung** $(x_1, \ldots, x_n) \gg (x'_1, \ldots, x'_n)$ ist ein Atom, wobei $\ll$ die – ggf. noch unspezifizierte – Induktionsordnung bezeichnet, $x_1, \ldots, x_n$ die ursprünglichen Induktionsvariablen sind und

$x'_1, \ldots, x'_n$ eine Kopie davon, die bei Anwendung des Lemmas auf eine Instanz von *prem* bzw. *conc* instantiiert wird. Die Anwendung von (4) bzw. (5) kann erst dann als Induktionsschritt bezeichnet werden, wenn das Redukt der Anwendung, d.h. die entsprechende Instanz der Prämisse von (4) bzw. Konklusion von (5) bewiesen, also auf *True* reduziert worden ist. Dazu muss die Induktionsordnung $\ll$ axiomatisiert werden, natürlich so, dass ihre Interpretation im Herbrandmodell der jeweiligen Spezifikation wohlfundiert ist.

Die bisher vorgestellten Regeln legen folgende Strategie beim Beweis einer Implikation nahe: Prämisse und Konklusion so lange kontrahieren bzw. expandieren, bis letztere von ersterer subsumiert wird. Sind Induktionsschritte erforderlich, dann muss die Ableitung Formeln hervorbringen, auf die die passenden Induktionshypothesen der Form (4) oder (5) anwendbar sind.

Beispiel 7.2.1 Wir zeigen die Korrektheit eines logischen Programms zur Berechnung der Partitionen einer Liste (siehe §6.2):

```
PARTITION = LIST[TRIV(entry)] and LIST{list(entry)/entry}[LIST[TRIV(entry)]] then
     defuncts  flatten:list(list(entry))->list(entry)
     preds     part:list*list(list(entry))
     vars      x,y:s  s,s':list(entry)  p:list(list(entry))
     axioms    flatten([]) = []
               flatten(s:p) = s++flatten(p)
               part([x],[[x]])
               part(x:y:s,[x]:p)  <==  part(y:s,p)
               part(x:y:s,(x:s'):p)  <==  part(y:s,s':p)
```

Die Korrektheit (der Axiome von) *part* wird durch folgende Hornformel beschrieben:

$$part(s,p) \quad \Rightarrow \quad s \equiv flatten(p). \tag{6}$$

Wir wählen s als Induktionsvariable (und markieren sie mit einem Ausrufungszeichen), werden also für einen Noetherschen Induktionsbeweis ein Prädikat $\ll$: $list(entry) \times list(entry)$ brauchen, dessen Interpretation im initialen Modell von PARTITION wohlfundiert ist. Die Induktionshypothesen (4) und (5) lauten hier demnach wie folgt:

```
s' = flatten(p')  <===  part(s',p') & !s >> s'
part(s',p')  ===>  (!s >> s' ==> s' = flatten(p'))
```

Das zweite wird an den Stellen (A) und (B) angewendet. Im jeweils folgenden Schritt wird die eben erzeugte Abstiegsbedingung durch Resolution mit dem (einzigen) Axiom $x : s \gg s$ für die Induktionsordnung sofort wieder eliminiert.

```
Derivation of:

part(s,p) ==> s = flatten(p)

Selecting induction variables at position [0,0] of the preceding formula leads to

All p:(part(!s,p) ==> !s = flatten(p))

Applying the axioms

  flatten[] = []
& flatten(s:p) = s++flatten(p)
& part([x],[[x]])
& (part(x:y:s,[x]:p) <=== part(y:s,p))
& (part(x:y:s,x:s':p) <=== part(y:s,s':p))
```

```
at positions [0,1,1],[0,0] of the preceding formula leads to

All x:(!s = [x] ==> [] = flatten[]) &
All x y s p0:(!s = x:y:s & part(y:s,p0) ==> y:s = flatten(p0)) &
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

The reducts have been simplified.

Applying the axiom resp. theorem

flatten[] = []

at position [0,0,1,1] of the preceding formula leads to

All x y s p0:(!s = x:y:s & part(y:s,p0) ==> y:s = flatten(p0)) &
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

The reducts have been simplified.

Shifting subformulas at position [0,0,0,1] of the preceding formula leads to

All x y s p0:(!s = x:y:s ==> (part(y:s,p0) ==> y:s = flatten(p0))) &
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

Applying the induction hypothesis

part(s,p) ===> (!s >> s ==> s = flatten(p))                              (A)

at position [0,0,1,0] of the preceding formula leads to

All x y s p0:(!s = x:y:s & (x:y:s >> y:s ==> y:s = flatten(p0)) ==>
              y:s = flatten(p0)) &
All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

The reducts have been simplified.

Applying the axioms

  x:s >> s
& (x >> y <=== x > y)

at position [0,0,0,1,0] of the preceding formula leads to

All x y s s' p0:(!s = x:y:s & part(y:s,s':p0) ==> y:s = s'++flatten(p0))

The reducts have been simplified.

Shifting subformulas at position [0,0,1] of the preceding formula leads to

All x y s s' p0:(!s = x:y:s ==> (part(y:s,s':p0) ==> y:s = s'++flatten(p0)))

Applying the induction hypothesis

part(s,p) ===> (!s >> s ==> s = flatten(p))                              (B)

at position [0,1,0] of the preceding formula leads to

All x y s s' p0:(!s = x:y:s & (x:y:s >> y:s ==> y:s = flatten(s':p0)) ==>
                 y:s = s'++flatten(p0))

The reducts have been simplified.
```

```
Applying the axioms

  x:s >> s
& (x >> y <=== x > y)

at position [0,0,1,0] of the preceding formula leads to

All x y s s' p0:(!s = x:y:s & y:s = flatten(s':p0) ==> y:s = s'++flatten(p0))

The reducts have been simplified.

Applying the axiom resp. theorem

s++flatten(p) = flatten(s:p)

at position [0,1,1] of the preceding formula leads to

True
```

Beispiel 7.2.2 (nach C.-P. Wirth) Selbst um einfache Aussagen, deren induktive Gültigkeit schon beim ersten Hinsehen klar ist, formal zu beweisen, sind manchmal nichttriviale Induktionsordnungen erforderlich – wie das folgende Beispiel (siehe [114], §3.2.2) zeigt.

```
PQ = NAT and
     preds     P:nat
               Q:nat*nat
               `gt`:(nat*nat)*(nat*nat)            -- lexikographische Ordnung
               >>:(nat*nat*nat)*(nat*nat*nat)      -- Induktionsordnung
               Nat:nat
     axioms    P(0)
               P(suc(x))  <==  P(x) /\ Q(x,suc(x))
               Q(x,0)
               Q(x,suc(y))  <==  P(x) /\ Q(x,y)
               (x,y) `gt` (x',y')  <==  x > x'
               (x,y) `gt` (x,y')  <==  y > y'
               (x,y,z) >> (x',y',z')  <==  x > x' /\ x > y'
               (x,y,z) >> (x',y',z')  <==  (y,z) `gt` (x',0) /\ (y,z) `gt` (y',z')
               Nat(0)
               Nat(suc(x)) <== Nat(x)
```

Wir beweisen die Formel

$$Nat(x) \wedge Nat(y) \wedge Nat(z) \quad \Rightarrow \quad p(x) \wedge q(y,z)$$

durch Noethersche Induktion, wobei alle drei Variablen x, y, z als Induktionsvariablen deklariert werden. Das Prädikat Nat schränkt die Instanzen von x, y, z auf Grundterme ein, die zu Normalformen der Sorte nat reduziert werden können. Ohne diese Einschränkung wäre die Formel nicht beweisbar.

Die Induktionshypothesen (4) und (5) lauten hier demnach wie folgt:

```
P(x') & Q(y',z')  <===  Nat(x') & Nat(y') & Nat(z')) & (!x,!y,!z) >> (x',y',z'))
Nat(x') & Nat(y') & Nat(z')  ===>  (!x,!y,!z) >> (x',y',z') ==> (P(x') & Q(y',z'))
```

Die erste wird an den Stellen (A) und (B) angewendet.

```
Derivation of

Nat(x) & Nat(y) & Nat(z) ==> P(x) & Q(y,z)
```

```
Selecting induction variables at positions [0,0,0],[0,1,0],[0,2,0] of the preceding formula leads to

Nat(!x) & Nat(!y) & Nat(!z) ==> P(!x) & Q(!y,!z)

Applying the axioms

  Q(x,0)
& (Q(x,suc(y)) <=== P(x) & Q(x,y))
& P(0)
& (P(suc(x)) <=== P(x) & Q(x,suc(x)))

at positions [1,1],[1,0] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 | Any x:(P(x) & Q(x,suc(x)) & !x = suc(x))) &
(Nat(!x) & Nat(!y) & Nat(!z) ==> !z = 0 | P(!y) & Any y:(Q(!y,y) & !z = suc(y)))

The reducts have been simplified.

Applying the induction hypothesis

P(x) & Q(y,z) <=== (Nat(x) & Nat(y) & Nat(z)) & (!x,!y,!z) >> (x,y,z)           (A)

at positions [0,1,1,0,0],[0,1,1,0,1] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 |
 Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)) & (suc(x),!y,!z) >> (x,x,suc(x)))) &
(Nat(!x) & Nat(!y) & Nat(!z) ==> !z = 0 | P(!y) & Any y:(Q(!y,y) & !z = suc(y)))

The reducts have been simplified.

Moving up quantifiers at position [1,1,1,1] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 |
 Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)) & (suc(x),!y,!z) >> (x,x,suc(x)))) &
(Nat(!x) & Nat(!y) & Nat(!z) ==> !z = 0 | Any y:(P(!y) & Q(!y,y) & !z = suc(y)))

Applying the induction hypothesis

P(x) & Q(y,z) <=== (Nat(x) & Nat(y) & Nat(z)) & (!x,!y,!z) >> (x,y,z)           (B)

at positions [1,1,1,0,0],[1,1,1,0,1] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 |
 Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)) & (suc(x),!y,!z) >> (x,x,suc(x)))) &
(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y) & (!x,!y,suc(y)) >> (!y,!y,y)))

The reducts have been simplified.

Applying the axiom resp. theorem

(x,y,z) >> (x',y',z') <=== x > x' & x > y'

at position [0,1,1,0,3] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 | Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)) & suc(x) > x)) &
(Nat(!x) & Nat(!y) & Nat(!z) ==>
```

```
 !z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y) & (!x,!y,suc(y)) >> (!y,!y,y)))

The reducts have been simplified.

Applying the axiom resp. theorem

(x,y,z) >> (x',y',z') <=== (y,z) `gt` (x',0) & (y,z) `gt` (y',z')

at position [1,1,1,1,0,2] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 | Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)) & suc(x) > x)) &
(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !z = 0 |
 Nat(!y) &
 Any y:(!z = suc(y) & Nat(y) & (!y,suc(y)) `gt` (!y,0) &
        (!y,suc(y)) `gt` (!y,y)))

The reducts have been simplified.

Applying the axioms

  ((x,y) `gt` (x',y') <=== x > x')
& ((x,y) `gt` (x,y') <=== y > y')

at positions [1,1,1,1,0,3],[1,1,1,1,0,2] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 | Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)) & suc(x) > x)) &
(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y) & suc(y) > y))

The reducts have been simplified.

Applying the axiom resp. theorem

suc(x) > x

at position [0,1,1,0,3] of the preceding formula leads to

(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !x = 0 | Any x:(!x = suc(x) & Nat(x) & Nat(suc(x)))) &
(Nat(!x) & Nat(!y) & Nat(!z) ==>
 !z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y) & suc(y) > y))

The reducts have been simplified.

Applying the axioms

  (Nat(suc(x)) <=== Nat(x))
& Nat(0)

at positions [0,1,1,0,2],[0,0,0] of the preceding formula leads to

Nat(!x) & Nat(!y) & Nat(!z) ==>
!z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y) & suc(y) > y)

The reducts have been simplified.

Applying the axiom resp. theorem

suc(x) > x
```

```
at position [1,1,1,0,2] of the preceding formula leads to

Nat(!x) & Nat(!y) & Nat(!z) ==> !z = 0 | Nat(!y) & Any y:(!z = suc(y) & Nat(y))

The reducts have been simplified.

Applying the axioms

  Nat(0)
& (Nat(suc(x)) <=== Nat(x))

at position [0,2] of the preceding formula leads to

True
```

Oft läßt sich der Beweis einer Formel φ auf keine Induktionshypothese, d.h. auf keine Instanz von φ hinführen. Vielmehr liefert er eine Instanz einer **Generalisierung** ψ von φ. ψ impliziert dann φ, ist aber nicht äquivalent zu φ. Eine geeignete Generalisierung von φ ist in vielen Fällen die konjunktive Verknüpfung von φ mit einem weiteren Beweisziel φ'. Dann ist der Einsatz von $\varphi \wedge \varphi'$-Instanzen als Induktionshypothesen im Beweis von φ korrekt, sofern sich auch φ' unter diesen Induktionsannahmen herleiten läßt.

Andere Generalisierungen von φ erhält man aus φ, indem man Teilterme von φ durch (allquantifizierte) Variablen ersetzt. Diese Art der Verallgemeinerung läßt sich zwar automatisieren, führt aber in vielen interessanten Fällen zu Formeln, die als Induktionshypothesen noch zu schwach sind. Umgekehrt besteht die Gefahr, dass man zu stark verallgemeinert und die Generalisierung nicht beweisbar ist, weil sie gar nicht gilt. Die in §4.2, §5.1 und unten behandelte Fixpunktinduktion befreit uns zwar von der Suche nach geeigneten Induktionsordnungen. Die Aufgabe, hinreichende Verallgemeinerung von Beweisziele zu finden, wird uns aber auch dort nicht abgenommen, allerdings ein klein weing stärker eingegrenzt.

Implizite Induktion basiert auf Satz 6.1.5(1): Eine Hornformel φ ist genau dann ein induktives Theorem von SP, wenn $SP \cup \varphi$ relativ konsistent bzgl. SP ist.[50] Zum Beweis der relativen Konsistenz können wir die Kriterien 6.1.5(2),(3) und 6.1.6 und 6.1.7 nicht anwenden, weil SP und $SP \cup \varphi$ dieselbe Signatur haben. Einzig 6.2.9 hat eine Chance. Dieses Kriterium ist hier aber nur dann anwendbar, wenn die Menge der Prädikate von SP bzgl. $SP \cup \varphi$ komplementabgeschlossen ist. Für Gleichheiten bedeutet das praktisch, dass SP und $SP \cup \varphi$ funktional sind (s. Satz 6.2.7). Nach Kor. 5.1.15 legt das die Bedingung nahe, dass $SP \cup \varphi$ konfluent ist. Tatsächlich besteht der übliche Ansatz impliziter Induktion im Beweis der Konfluenz von $SP \cup \varphi$. Das Bindeglied zwischen Konfluenz und Konsistenz ist hier die SP-Reduzibilität von φ:

reduzible Hornformel

Definition 7.2.3 Eine Hornformel $t\{\equiv u\} \Leftarrow H$ ist **SP-reduzibel**, wenn für alle $\sigma : X \to T_\Sigma$ aus der $(SP \cup \varphi)$-Konvergenz von $H\sigma$ folgt, dass $t\sigma$ nicht SP-reduziert ist (s. Def. 5.1.10).

Lemma 7.2.4 Sei SP eine konstruktorbasierte Spezifikation und φ eine Hornformel über SP so, dass $SP \cup \varphi$ terminiert.

(1) Ist $SP \cup \varphi$ konfluent und φ SP-reduzibel, dann ist $SP \cup \varphi$ relativ konsistent bzgl. SP.

(2) Ist SP konfluent und $SP \cup \varphi$ relativ konsistent bzgl. SP, dann ist φ SP-reduzibel.

Beweis. ☞ Übung. ❑

[50] Das gilt auch für Spezifikationen SP, die die Bedingungen 5.1.2(1)-(3) nicht erfüllen.

Die Konfluenz von $SP \cup \varphi$ schließt man dann aus der von SP mithilfe hierarchischer Kriterien[51] (s. z.B. [18]). Vergleichende Analysen von impliziten bzw. Noetherschen Induktionsbeweisen haben gezeigt, dass mit impliziter Induktion kein entscheidender Effizienzgewinn zu erreichen ist [80, 13]. Wo Noethersche Induktion geeignete Induktionsordnungen verlangt, erfordert implizite Induktion wegen der Bedingung, dass $SP \cup \varphi$ terminiert (s. Lemma 7.2.4), Reduktionsordnungen für SP, die mit φ verträglich sind. Man spricht auch von "induktiver Vervollständigung", weil unter gewissen Bedingungen die Erweiterung von $SP \cup \varphi$ um kritische Klauseln von $SP \cup \varphi$ zu einer konfluenten Spezifikation führt, aus deren Existenz dann auf die induktive Gültigkeit von φ im Herbrandmodell von SP zurückgeschlossen werden kann. Wir werden implizite Induktion hier nicht weiter behandeln. Sie wird z.B. im Induktionsbeweiser SPIKE [18] eingesetzt.

Fixpunktinduktion und implizite Induktion kommen im Gegensatz zu explizter Induktion ohne wohlfundierte Ordnungen aus. Die Korrektheit der Fixpunkt(induktions)regeln folgt direkt aus Satz 4.2.8, der u.a. besagt, dass das Herbrandmodell einer funktionalen Spezifikation SP logische Prädikate und — abgestützt auf Satz 5.1.20 — auch die Graphen definierter Funktionen als kleinste Lösungen ihrer jeweiligen Axiome interpretiert. Ein Beweisziel der Form $r(x) \Rightarrow q(x)$ ist deshalb immer dann ein induktives Theorem von SP, wenn die Interpretation von r durch $q^{Her(SP)}$ ebenfalls ein Modell von SP liefert, wenn also $q^{Her(SP)}$ die Axiome für r erfüllt. Genau das drückt die Konklusion der Fixpunktinduktion aus:

Sei SP funktional, r ein logisches Prädikat von SP, f eine definierte Funktion von SP sowie AX_r und AX_f die Mengen der Axiome, in denen r bzw. f vorkommt. Weiterhin sei vorausgesetzt, dass die Formel φ weder r noch von r abhängige Prädikate oder Funktionen q enthält. q ist von r **abhängig**, wenn r in der Prämisse eines Axioms für q vorkommt oder ein dort vorkommendes Prädikat von r abhängig ist.

Fixpunktinduktion über r

$$\frac{r(x) \;\Rightarrow\; \varphi(x)}{\bigwedge_{\psi \in AX_r} \psi[\varphi(u)/r(u) \mid r(u) \text{ kommt in } \psi \text{ vor}]} \Uparrow$$

Fixpunktinduktion über f

$$\frac{f(x) \equiv y \;\Rightarrow\; \varphi(x,y)}{\bigwedge_{\psi \in flat(AX_f)} \psi[\varphi(u,v)/f(u) \equiv v \mid f(u) \equiv v \text{ kommt in } \psi \text{ vor}]} \Uparrow$$

Die Expansionseigenschaft ($\Uparrow$) folgt aus Satz 4.2.8: Jede Interpretation von $r : w$ auf $T_{\Sigma,w}$, die AX_r erfüllt, enthält $r^{Her(SP)}$, weil $r^{Her(SP)}$ die kleinste Lösung von AX_r ist. Also gilt $r(x) \Rightarrow \varphi(x)$.

Fixpunktinduktion kann auch auf eine bedingte Behauptung wie

$$(r(x) \wedge pre(x)) \;\Rightarrow\; \varphi(x) \qquad \text{bzw.} \qquad (f(x) \equiv y \wedge pre(x)) \Rightarrow \varphi(x,y)$$

angewendet werden, weil diese zu

$$(r(x) \Rightarrow (pre(x) \Rightarrow \varphi(x)) \qquad \text{bzw.} \qquad (f(x) \equiv y \Rightarrow (pre(x) \Rightarrow \varphi(x,y))$$

äquivalent ist.

Fixpunktinduktion macht die Verwendung einer wohlfundierten Ordnung überflüssig. Sie fasst mehrere Schritte eines Noetherschen Induktionsbeweises zu einem Schritt zusammen (Anwendung von Induktionshypothesen, Beweis von Abstiegsbedingungen). Umgekehrt lassen sich Beweise durch Fixpunktinduktion in Beweise durch Noethersche Induktion umformen. Definiert man z.B. $\ll$: ss wie bei struktureller Induktion (siehe (1) oder (2)), um Eigenschaften eines Prädikats $r : s$ mit Axiomen[52]

$$r(c_1(x_1,\ldots,x_{m_1})) \;\Leftarrow\; \varphi_i(x_1,\ldots,x_{m_1})\{\wedge r(x_{k_1})\},\;\; \ldots\;, r(c_n(x_1,\ldots,x_{m_n})) \;\Leftarrow\; \varphi_i(x_1,\ldots,x_{m_n})\{\wedge r(x_{k_n})\},$$

Konstruktoren $c_1,\ldots,c_n$ und $1 \le k_i \le m_i$ zu beweisen, dann lautet die Regel der Fixpunktinduktion über r wie folgt:

$$\frac{r(x) \;\Rightarrow\; \varphi(x)}{\bigwedge_{i=1}^{n}(\varphi(c_i(x_1,\ldots,x_{m_i})) \;\Leftarrow\; \varphi_i(x_1,\ldots,x_{m_i})\{\wedge\varphi(x_{k_i})\})} \Uparrow \tag{7}$$

[51] Unsere Kriterien Satz 5.2.7 und 5.2.9 versagen, weil $SP \cup \varphi$ in der Regel nicht orthogonal ist und dieselbe Signatur wie SP hat.
[52] Geschweifte Klammern um eine Teilformel deuten an, dass diese optional ist.

Gilt der Sukzedent von (7) und ist die als transitiver Abschluss von

$$\{(c_i(t_1,\ldots,t_{m_i}),t_{k_i}) \in T_\Sigma^2 \mid Her(SP) \models \varphi_i(t_1,\ldots,t_{m_i}),\ 1 \le i \le n\}$$

definierte Ordnung $\ll$ wohlfundiert, dann könnte man die Gültigkeit von $r(x) \Rightarrow \varphi(x)$ in $Her(SP)$ auch folgendermaßen durch Noethersche Induktion beweisen: O.B.d.A. sei $t \in NF_\Sigma$ und $Her(SP) \models r(t)$. Dann gilt $SP \vdash_{cut} r(t)$. Also gibt es $1 \le i \le n$ und Grundterme $t_1,\ldots,t_{m_i}$ mit $t = c_i(t_1,\ldots,t_{m_i})$ und

$$SP \vdash_{cut} \varphi_i(t_1,\ldots,t_{m_i})\{\wedge r(t_{k_i})\}. \tag{8}$$

Entfällt die in geschweifte Klammern gesetzte Teilformel, dann lässt sich aus (8) und dem Sukzedenten von (7) die Formel $\varphi(c_i(t_1,\ldots,t_{m_i})) = \varphi(t)$ ableiten. Andernfalls folgt $SP \vdash_{cut} \varphi(t_{k_i})$ nach Induktionsvoraussetzung aus $SP \vdash_{cut} r(t_{k_i})$ und damit aus aus (8) und dem Sukzedenten von (7) die Ableitbarkeit von $\varphi(c_i(t_1,\ldots,t_{m_i})) = \varphi(t)$. Also ist $\varphi(t)$ in $Her(SP)$ gültig.

☞ Zeigen Sie, dass $\ll^{Her(SP)}$ wohlfundiert ist, wenn es ein SP-Modell gibt, das $\ll$ als wohlfundierte Relation interpretiert!

Umgekehrt kann man die Konklusion der Fixpunktinduktion als Konjunktion möglicher Induktionsschritte für den Beweis von φ auffassen. Ist sie nicht beweisbar, dann muss φ **generalisiert** werden, d.h. zu einer stärkeren Behauptung $\varphi \wedge \varphi'$ erweitert werden. Damit wird die Konklusion der Fixpunktinduktion u.U. abgeschwächt, weil die – Induktionshypothesen bei Noetherscher Induktion entsprechenden – Prämissen der transformierten Hornformeln stärker werden. r liefert die "untere Grenze" der Generalisierung, da ja $r(x) \Rightarrow (\varphi(x) \wedge \varphi'(x))$ gelten soll. φ ist die "obere Grenze". Dennoch gibt es unendlich viele Formeln φ', die

$$r(x) \quad \Rightarrow \quad \varphi(x) \wedge \varphi'(x) \quad \Rightarrow \quad \varphi(x)$$

erfüllen, und für die man alle die entsprechende Konklusion der Fixpunktinduktion

$$\bigwedge_{\psi \in AX_r} \psi[(\varphi(u) \wedge \varphi'(u))/r(u) \mid r(u) \text{ kommt in } \psi \text{ vor}]$$

erzeugen müsste – was zu einem Beweisbaum mit unendlichem Verzweigungsgrad führen würde –, um möglicherweise zu einem Beweis von $r(x) \Rightarrow \varphi(x)$ zu kommen. Das gibt zumindest einen Hinweis darauf, warum die Menge der induktiven Theoreme einer Spezifikation nicht aufzählbar ist. Die Menge der unendlich vielen Konklusionen der Fixpunktinduktion entspricht einer **Formel zweiter Stufe**:

$$\exists\, p : \bigwedge_{\psi \in AX_r} \psi[(\varphi(u) \wedge p(u))/r(u) \mid r(u) \text{ kommt in } \psi \text{ vor}] \ \wedge\ (r(x) \ \Rightarrow\ \varphi(x) \wedge p(x)).$$

Auch das begründet die Nicht-Aufzählbarkeit induktiver Theoreme.

[39] verwendet eine andere Art der Verallgemeinerung, die ebenfalls manchmal zur Abschwächung der Beweisverpflichtung führt. Sei wieder SP funktional, r ein logisches Prädikat von SP, f eine definierte Funktion von SP sowie AX_r und AX_f die Mengen der Axiome, in denen r bzw. f vorkommt. Die Ergänzungen gegenüber der Fixpunktinduktion sind unterstrichen.

starke Fixpunktinduktion über r

$$\frac{r(x) \ \Rightarrow\ \varphi(x)}{\bigwedge_{r(t) \Leftarrow \psi \in AX_r} \quad \varphi(t) \Leftarrow (\psi[(\varphi(u)\underline{\wedge r(u)})/r(u) \mid r(u) \text{ kommt in } \psi \text{ vor}]\underline{\wedge r(t)})} \Uparrow$$

starke Fixpunktinduktion über f

$$\frac{f(x) \equiv y \ \Rightarrow\ \varphi(x,y)}{\bigwedge_{f(t) \equiv u \Leftarrow \psi \in flat(AX_f)} \varphi(t,u) \Leftarrow (\psi[(\varphi(v,v')\underline{\wedge f(v) \equiv v'})/f(v) \equiv v' \mid f(v) \equiv v' \text{ kommt in } \psi \text{ vor}]\underline{\wedge f(t) \equiv u})} \Uparrow$$

Auch hier verschwinden bei der Anwendung der Regel zunächst die Vorkommen von r bzw. f in den Axiomen. In den Prämissen der transformierten Axiome werden aber ggf. neue Vorkommen von r bzw. f erzeugt, die die Prämissen verstärken.

Wir zeigen, dass auch die starke Fixpunktinduktion über $r : w$ korrekt ist. Sei $SP = (\Sigma, AX)$, Φ die von AX induzierte Schrittfunktion (siehe §4.2), r^H die Interpretation von r in $Her(SP)$, A die Σ-Herbrandstruktur, die r durch $\{t \in T_{\Sigma,w} \mid Her(SP) \models \varphi(t)\}$ und alle anderen Prädikate von SP wie im Herbrandmodell interpretiert, und $F : \wp(T_{\Sigma,w}) \to \wp(T_{\Sigma,w})$ für alle Σ-Herbrandstrukturen B definiert durch $F(r^B) = r^{\Phi(B)}$. Die starke Fixpunktinduktion über r ist offenbar genau dann korrekt, wenn

$$F(r^A \cap r^H) \cap r^H \subseteq r^A \quad \Rightarrow \quad r^H \subseteq r^A \tag{9}$$

gilt. Eine Relation r ist **F-abgeschlossen**, wenn $F(r) \subseteq r$ gilt. Da Φ und damit auch F monoton und r^H F-abgeschlossen ist, folgt

$$F(r^A \cap r^H) \subseteq F(r^A \cap r^H) \cap F(r^A \cap r^H) \subseteq F(r^A \cap r^H) \cap F(r^H) \subseteq F(r^A \cap r^H) \cap r^H \subseteq r^H \cap r^A \tag{10}$$

aus der Prämisse von (9). Damit ist $r^A \cap r^H$ F-abgeschlossen. Da r^H die kleinste F-abgeschlossene Relation ist, folgt $r^H \subseteq r^H \cap r^A$ und damit die Konklusion von (9).

Ist r verschränkt-rekursiv mit anderen Prädikaten $r_1, \ldots, r_n$ definiert, dann ist i.a. eine Behauptung φ über r auch nur "verschränkt-induktiv" mit Behauptungen über $r_1, \ldots, r_n$ beweisbar. Tatsächlich kann Fixpunktinduktion leicht zu einer Regel erweitert werden, mit deren Hilfe sich alle diese Behauptungen gleichzeitig zeigen lassen:

simultane Fixpunktinduktion über $r_1, \ldots, r_n$

$$\frac{\bigwedge_{i=1}^{n}(r_i(x) \;\Rightarrow\; \varphi_i(x))}{\bigwedge_{i=1}^{n}\bigwedge_{\psi \in AX_{r_i}} \psi[\varphi_j(u)/r_j(u) \mid r_j(u) \text{ kommt in } \psi \text{ vor}, \; 1 \le j \le n]} \Uparrow$$

Analog zur Fixpunktinduktion sei vorausgesetzt, dass $\varphi_1, \ldots, \varphi_n$ weder r_i, $1 \le i \le n$, noch von r_i abhängige Prädikate oder Funktionen enthält.

Wie lauten die entsprechende Verallgemeinerung der Fixpunktinduktion *über einer definierten Funktion f*? Möglicherweise muss diese auch mit der Fixpunktinduktion über einem Prädikat r kombiniert werden, wenn ein f und r wechselseitig-rekursiv spezifiziert sind.

Beispiel 7.2.5 Wir beweisen nochmal die Formel

$$part(L, P) \quad \Rightarrow \quad L \equiv flatten(P)$$

aus Beispiel 7.2.1, jetzt aber mit Fixpunktinduktion. Der Beweis wurde wieder mit *Expander2* erstellt [81]. Er ist um einiges kürzer und direkter als der mit Noetherscher Induktion geführte (siehe Bsp. 7.2.1). Narrowing wird hier als Sammelbegriff für Narrowing und Resolution im Sinne von §7.1 verwendet.

```
Derivation of

part(s,p) ==> s = flatten(p)

Applying fixpoint induction w.r.t.

  part([x],[[x]])
& (part(x:y:s,[x]:p) <=== part(y:s,p))
& (part(x:y:s,x:s':p) <=== part(y:s,s':p))
```

```
at position [] of the preceding formula leads to

All x:([x] = flatten[[x]]) &
All x y s p:(y:s = flatten(p) ==> x:y:s = flatten([x]:p)) &
All x y s p s':(y:s = flatten(s':p) ==> x:y:s = flatten(x:s':p))

The reducts have been simplified.

Applying the axiom resp. theorem

flatten(s:p) = s++flatten(p)

at positions [2,0,1,1],[2,0,0,1],[1,0,1,1],[0,0,1] of the preceding formula leads to

[] = flatten[]

The reducts have been simplified.

Narrowing the preceding formula leads to

True
```

Beispiel 7.2.6 (mergesort) Wir zeigen die Korrektheit eines funktionalen Sortierprogramms (*Sortieren durch Mischen*). Es wird als Extension von LIST'[ORD(entry)] spezifiziert (siehe Beispiel 6.3.2). Die Beweise setzen voraus, dass die Relation $\sim$ ("ist Permutation von") eine Kongruenzrelation ist. Axiome für $\sim$ werden nicht verwendet, aber mehrere Lemmas (`theorems`).

```
MERGESORT = LIST'[ORD(entry)] and
     defuncts  sort:list(nat)->list(entry)
               merge:list(entry)*list(entry)->list(entry)
               split:list(entry)->(list(entry)*list(entry))
     preds     ~:list(entry)*list(entry)
     vars      x,y:s  s,s1,s2,s',s1',s2':list(entry)
     axioms    sort([]) = []
               sort([x]) = [x]
               sort(x:y:s) = merge(s1',s2')
                 <==  split(s) = (s1,s2) & sort(x:s1) = s1' & sort(y:s2) = s2'
               merge([],s) = s
               merge(s,[]) = s
               merge(x:s1,y:s2) = x:merge(s1,y:s2)  <==  x <= y
               merge(x:s1,y:s2) = y:merge(x:s1,s2)  <==  x > y
               split([]) = ([],[])
               split([x]) = ([x],[])
               split(x:y:s) = (x:s1,y:s2)  <==  split(s) = (s1,s2)
     lemmas    sorted(s1) & sorted(s2)  ==>  sorted(merge(s1,s2))          (1)
               split(s) = (s1,s2)  ==>  s ~ s1++s2                         (2)
               s ~ merge(s1,s2) ==>  s ~ s1++s2                            (3)
               x:y:(s++s') ~ (x:s)++(y:s')                                 (4)
               s'++x:s ~ x:s++s'                                           (5)
               x > y ===> y <= x                                           (6)
```

Die Korrektheitsanforderungen an (die Axiome für) *sort* lauten:

$$\begin{aligned} sort(s) \equiv s' \quad &\Rightarrow \quad sorted(s'), \\ sort(s) \equiv s' \quad &\Rightarrow \quad s \sim s'. \end{aligned}$$

In beiden Teilen führt die Anwendung der Fixpunktinduktion zur Einführung eines Prädikats, das die jeweilige Konklusion $\varphi(x)$ (s.o.) wiedergibt. Die automatisch erzeugten (co-Horn-)Axiome (siehe §7.3) für diese Prädikate lauten demnach wie folgt:

```
               sort0(s,s')  ===>  sorted(s')    bzw.   sort1(s,s')  ===>  s ~ s'
```

Hier sind die Beweise:

```
Derivation of

    sort(s) = s' ==> sorted(s')

Applying fixpoint induction w.r.t.

  sort[] = []
& sort[x] = [x]
& (sort(x:(y:s)) = merge(z0,z1)  <===  split(s) = (s1,s2) & sort(x:s1) = z0 & sort(y:s2) = z1)

to the preceding tree leads to the formula

All x y s7 z0 z1 s1 s2:
(  sort0([],[])
 & sort0([x],[x])
 & ( sort0(x:(y:s7),merge(z0,z1))  <===  split(s7) = (s1,s2) & sort0(x:s1,z0) & sort0(y:s2,z1)))

Simplifying (4 steps) the preceding tree leads to new ones. The current factor is given by

sort0([],[])

Narrowing the preceding factor leads to the factor

sorted[]

Narrowing the preceding factor leads to new ones. The current factor is given by

All x: (sort0([x],[x]))

Narrowing the preceding factor leads to the factor

All x2: (sorted[x2])

Narrowing the preceding factor leads to a new formula. The current formula is given by

All x y s7 z0 z1 s1 s2:
(    split(s7) = (s1,s2) & sort0(x:s1,z0) & sort0(y:s2,z1)
 ==> sort0(x:(y:s7),merge(z0,z1)))

Applying the axiom

     sort0(s,s') ===> sorted(s')

at positions [0,1],[0,0,2],[0,0,1] of the preceding tree leads to the formula

All x y s7 z0 z1 s1 s2:
(split(s7) = (s1,s2) & sorted(z0) & sorted(z1)  ==>  sorted(merge(z0,z1)))

Applying the theorem

     sorted(s) & sorted(s') ===>  sorted(merge(s,s'))

at position [0,0] of the preceding tree leads to the formula

All x y s7 z0 z1 s1 s2:
(sorted(merge(z0,z1)) & split(s7) = (s1,s2)  ==>  sorted(merge(z0,z1)))

Simplifying (3 steps) the preceding tree leads to the formula
```

```
True

*******************************************************************************

Derivation of

    sort(s) = s' ==> s ~ s'

Applying fixpoint induction w.r.t.

  sort[] = []
& sort[x] = [x]
& (sort(x:(y:s)) = merge(z3,z4) <=== split(s) = (s1,s2) & sort(x:s1) = z3 & sort(y:s2) = z4)

to the preceding tree leads to the formula

All x y s7 z3 z4 s1 s2:
(  sort1([],[])
 & sort1([x],[x])
 & (     sort1(x:(y:s7),merge(z3,z4))
    <=== split(s7) = (s1,s2) & sort1(x:s1,z3) & sort1(y:s2,z4)))

Applying the axiom

     sort1(s,s') ===> s ~ s'

at positions [0,2,1,2],[0,2,1,1],[0,2,0],[0,1],[0,0] of the preceding tree leads to the formula

All x y s7 z3 z4 s1 s2:
(  [] ~ []
 & [x] ~ [x]
 & (x:(y:s7) ~ merge(z3,z4)  <===  split(s7) = (s1,s2) & x:s1 ~ z3 & y:s2 ~ z4))

Simplifying (13 steps) the preceding tree leads to the formula

All x y s7 s1 s2: (split(s7) = (s1,s2) ==> x:(y:s7) ~ merge(x:s1,y:s2))

Applying the theorem

     split(s) = (s1,s2) ===> s ~ s1++s2

at position [0,0] of the preceding tree leads to the formula

All x y s7 s8 s9: (s7 ~ s8++s9 ==> x:(y:s7) ~ merge(x:s8,y:s9))

Applying the theorem

     s ~ merge(s1,s2) <=== s ~ s1++s2

at position [0,1] of the preceding tree leads to the formula

All x y s7 s8 s9: (s7 ~ s8++s9 ==> x:(y:s7) ~ (x:s8)++(y:s9))

Simplifying the preceding tree leads to the formula

All x y s7 s8 s9: (s7 ~ s8++s9 ==> x:(y:s7) ~ x:(s8++(y:s9)))

Applying the theorem

x:y:(s1++s2) ~ x:(s1++(y:s2))
```

```
at position [0,1] of the preceding tree leads to the formula

All x3 y2 s7 s8 s9: (s7 ~ s8++s9 ==> True)

Simplifying (3 steps) the preceding tree leads to the formula

True
```

☞ *sorted* spezifiert aufsteigende Sortierung. Geben Sie Hornaxiome für ein Prädikat *sortedB* an, das im Herbrandmodell genau dann gilt, wenn die Argumentliste absteigend sortiert ist. Zeigen Sie

$$reverse(L) \equiv L' \quad \Rightarrow \quad (sorted(L) \Rightarrow sortedB(L'))$$

mit Fixpunktinduktion über *reverse* (siehe Bsp. 6.3.2)!

Beispiel 7.2.7 (quick- und bubblesort) Zeigen Sie analog zu Beispiel 7.2.6 die Korrektheit von *quicksort* und *bubblesort* bzgl. folgender Spezifikationen:

```
QUICKSORT = LIST'[ORD(entry)] and
     defuncts  sort:list(entry)->list(entry)
               smaller,greater:nat->(entry->bool)
     vars      x,y:nat  s,s1,s2:list(entry)
     axioms    sort([]) = []
               sort(x:s) = sort(s1)++x:sort(s2)
                     <==  filter(smaller(x),s) = s1 /\ filter(greater(x),s) = s2
               smaller(x)(y) = true  <==  x <= y
               smaller(x)(y) = false  <==  x > y
               greater(x)(y) = not(smaller(x)(y))

BUBBLESORT = LIST'[ORD(entry)] then
     defuncts  sort:list(entry)->list(entry)
               bubble:list(entry)->list(entry)
     vars      x,y:entry  s,s':list(entry)
     axioms    sort(s) = s'  <==  bubble(s) = s' /\ s = s'
               sort(s) = sort(s')  <==  bubble(s) = s' /\ s =/= s'
               bubble([]) = []
               bubble([x]) = [x]
               bubble(x:y:s) = x:bubble(y:s)  <==  x <= y
               bubble(x:y:s) = y:bubble(x:s)  <==  x > y
```

☞ Optimieren Sie BUBBLESORT durch Erweiterung des Wertebereiches von *bubble* um einen Booleschen Wert (*flag*), der angibt, ob der jeweilige Aufruf von *bubble* die Argumentliste verändert hat! Anstatt die Bedingung $s = s'$ bzw. $s \neq s'$ zu überprüfen, braucht *sort* dann nur den Wert des flags abzufragen. Zeigen Sie, dass Ihr optimiertes *bubble* zum *bubble* von BUBBLESORT äquivalent ist!

Beispiel 7.2.8 (Rucksackproblem) Gegeben sind ein Rucksack der Größe b und $n > 0$ Objekte mit Größen $a_1, \ldots, a_n$ und Nutzwerten $c_1, \ldots, c_n$. Gesucht ist eine Teilmenge M von $\{1, \ldots, n\}$ mit $\sum_{i=1}^{n} a_i x_i \leq b$ und maximalem Gesamtnutzen $c(\vec{x}) =_{def} \sum_{i=1}^{n} c_i x_i$. $\vec{x} \in \{0,1\}$ ist der charakteristische Vektor von M, d.h. $x_i = 1 \Leftrightarrow i \in \{0,1\}$. Der Aufruf $knap(n,b)$ des folgenden, leider exponentiellen Haskell-Programms *knap* berechnet $\vec{x}$ und $c(\vec{x})$ rekursiv. Vektoren werden hier als Listen implementiert.

```
knap(1,b) = if a(1) > b then ([0],0) else ([1],c(1))
knap(n,b) = if d1 > d2' then (v1++[0],d1) else (v2++[1],d2')
            where (v1,d1) = knap(n-1,b)
                  (v2,d2) = knap(n-1,b-a(n))
                  d2' = d2+c(n)
```

☞ Zeigen Sie die Korrektheit von *knap* durch Fixpunktinduktion! Die zu beweisende Behauptung lautet als Formel wie folgt:

$$knap(k,a) = \vec{x} \implies (\sum_{i=1}^{n} a_i x_i \leq b \wedge \forall \vec{y} : (\sum_{i=1}^{n} a_i y_i \leq b \Longrightarrow \sum_{i=1}^{n} c_i y_i \leq \sum_{i=1}^{n} c_i x_i)).$$

Die Methode der Tabellierung (siehe §8.6) überführt *knap* in den folgenden Algorithmus, der *knap* als Matrix darstellt:

```
knap = array' bounds [(i,f i) | i <- range bounds]
       where bounds = ((1,-1000),(n,b))
             f(1,b) = if a(1) > b then ([0],0) else ([1],c(1))
             f(n,b) = if d1 > d2' then (v1++[0],d1) else (v2++[1],d2')
                      where (v1,d1) = knap!(n-1,a)
                            (v2,d2) = knap!(n-1,b-a(n))
                            d2' = d2+c(n)
```

7.3 Coinduktion

Aus der Interpretation von q im Herbrandmodell folgt die Korrektheit einer zur Fixpunktinduktion über Prädikaten dualen Beweisregel für Coprädikate: Sei AX_q die Menge der Axiome, in denen q vorkommt. Analog zur Fixpunktinduktion sei vorausgesetzt, dass φ weder q noch von q abhängige Coprädikate enthält.

Coinduktion über q

$$\frac{\varphi(x) \;\Rightarrow\; q(x)}{\bigwedge_{\psi \in AX_q} \psi[\varphi(u)/q(u) \mid q(u) \text{ kommt in } \psi \text{ vor}]} \Uparrow$$

Auch Coinduktionsbeweise können es erforderlich machen, die jeweiligen Behauptungen zu **generalisieren**. Da das hier bedeutet, die *Prämisse* φ einer Implikation zu erweitern, muss φ *disjunktiv* mit einer Formel φ' verknüpft werden, damit die Implikation verstärkt wird. q liefert jetzt die "obere Grenze" der Generalisierung und φ die "untere Grenze", da $(\varphi(x) \vee \varphi'(x)) \Rightarrow q(x)$ gelten soll:

$$\varphi(x) \quad \Rightarrow \quad \varphi(x) \vee \varphi'(x) \quad \Rightarrow \quad q(x).$$

☞ Zeigen Sie $unsorted([1,2,3,5,4])$ erstens mit Coinduktion über *unsorted* und zweitens durch Widerlegung, d.h. durch Ableitung der Formel $sorted([1,2,3,5,4]) \Rightarrow False$ nach *True*!

Beispiel 7.3.1 (MAP implementiert STACK, cont.) Das Lemma (15) in Beispiel 6.4.3 wird mit Coinduktion bewiesen.

```
Derivation of

i > j ==> (upd(i,x,f),j) ~ (f,j)

Applying coinduction w.r.t.

s ~ s' ===> top(s) = top(s') & pop(s) ~ pop(s')

at position [] of the preceding formula leads to

All i j x f:(i > j ==> top(upd(i,x,f),j) = top(f,j)) &
All i j x f:(i > j ==>
             Any i0 j0 x0 f0:(i0 > j0 & pop(upd(i,x,f),j) = (upd(i0,x0,f0),j0) &
                              pop(f,j) = (f0,j0)))
```

```
Reducts have been simplified.

Applying the axioms

  pop(f,i) = (f,pred(i))
& top(f,i) = get(f,i)

at positions [1,0,1,0,2,0],[1,0,1,0,1,0],[0,0,1,1],[0,0,1,0] of the preceding formula leads to

All i j:(i > j ==> i > pred(j)) &
All i j x f:(i > j ==> get(upd(i,x,f),j) = get(f,j))

Reducts have been simplified.

Applying the axioms

  get(upd(i,x,f),i) = entry(x)
& (get(upd(i,x,f),j) = get(f,j) <=== i =/= j)

at position [1,0,1,0] of the preceding formula leads to

All i j:(i > j ==> i > pred(j))

Reducts have been simplified.

Applying the theorem

i > j ===> i > pred(j)

at position [0,0] of the preceding formula leads to

True
```

Durch Dualisierung der starken Fixpunktinduktion erhält man die *starke Coinduktion* [39]. Sei wieder q ein Coprädikat und AX_q die Menge der Axiome, in denen q vorkommt. Die Ergänzungen gegenüber der Coinduktion sind unterstrichen.

starke Coinduktion über q

$$\frac{\varphi(x) \;\Rightarrow\; q(x)}{\bigwedge_{q(t)\Rightarrow\psi\in AX_q} \quad \varphi(t) \Rightarrow (\psi[(\varphi(u)\underline{\vee q(u)})/q(u) \mid q(u) \text{ kommt in } \psi \text{ vor}]\underline{\vee q(t)})} \Uparrow$$

Auch hier verschwinden bei der Anwendung der Regel zunächst die Vorkommen von q in den Axiomen. In den Konklusionen der transformierten Axiome werden aber ggf. neue Vorkommen von q erzeugt, die die Konklusionen verstärken.

☞ Zeigen Sie die Korrektheit der starken Coinduktion durch Dualisierung des Beweises der Korrektheit der starken Fixpunktinduktion (siehe §7.2)!

Ist q verschränkt-rekursiv mit anderen Coprädikaten $q_1, \ldots, q_n$ definiert, dann ist i.a. eine Behauptung $\varphi(x) \Rightarrow q(x)$ auch nur "verschränkt-induktiv" mit entsprechenden Behauptungen über $\varphi_i(x) \Rightarrow q_i(x)$, $1 \leq i \leq n$, beweisbar. Tatsächlich kann Coinduktion leicht zu einer Regel erweitert werden, mit deren Hilfe sich diese Behauptungen gleichzeitig zeigen lassen:

simultane Coinduktion über $q_1, \ldots, q_n$

$$\frac{\bigwedge_{i=1}^{n}(\varphi_i(x) \;\Rightarrow\; q_i(x))}{\bigwedge_{i=1}^{n}\bigwedge_{\psi\in AX_{r_i}} \psi[\varphi_j(u)/q_j(u) \mid q_j(u) \text{ kommt in } \psi \text{ vor},\; 1 \leq j \leq n]} \Uparrow$$

Analog zur simultanen Fixpunktinduktion sei vorausgesetzt, dass $\varphi_1, \ldots, \varphi_n$ weder q_i, $1 \leq i \leq n$, noch von q_i abhängige Coprädikate enthält.

Während eine der Fixpunktinduktion zugängliche Formel $r(x) \Rightarrow \varphi(x)$ verlangt, dass das *Prädikat* r die Eigenschaft φ hat, sagt eine mit Coinduktion transformierbare Formel $\varphi(x) \Rightarrow q(x)$ aus, dass das Coprädikat q für alle *Objekte* gilt, die φ erfüllen. Die Expansion von $\varphi(x) \Rightarrow q(x)$ ist also eher eine *Auswertung* von (Instanzen von) q als ein Beweis. Wäre q ein Prädikat, dann beginnt die Expansion von $\varphi(x) \Rightarrow q(x)$ in der Regel mit Auswertungsschritten, also Entfaltungen von q. Das Gleiche gilt für eine Expansion von $q(x) \Rightarrow \varphi(x)$, falls q ein Coprädikat ist, obwohl diese auf einen *Beweis* (der Gültigkeit von φ für q) hinausläuft.

Die vier genannten Formelschemata sind demnach nicht nur paarweise mit dualen Regeln expandierbar, die Expansionen streben auch duale Ziele an: einerseits Beweise, andererseits Auswertungen. Die Symmetrien sind in Fig. 8 veranschaulicht. *Narrowing* steht dort für Entfaltungen zusammen mit Simplifikationen, die notwendig sind, um schließlich *True*, *False* oder eine Menge gelöster Formeln zu erreichen (s. §7.1).

Die Anwendung von Fixpunktinduktion/Coinduktion einerseits und Narrowing andererseits offenbart eine weitere Dualität. Im ersten Fall werden Axiome von Behauptungen verändert, im zweiten Fall ist es genau umgekehrt: Behauptungen werden von Axiomen transformiert.

Bei den Narrowing-Anwendungen auf die Formeln $\varphi(x) \Rightarrow p(x)$ und $q(x) \Rightarrow \varphi(x)$ in Fig. 8 ist die Expansion von $p(x)$ bzw. $q(x)$ gemeint.

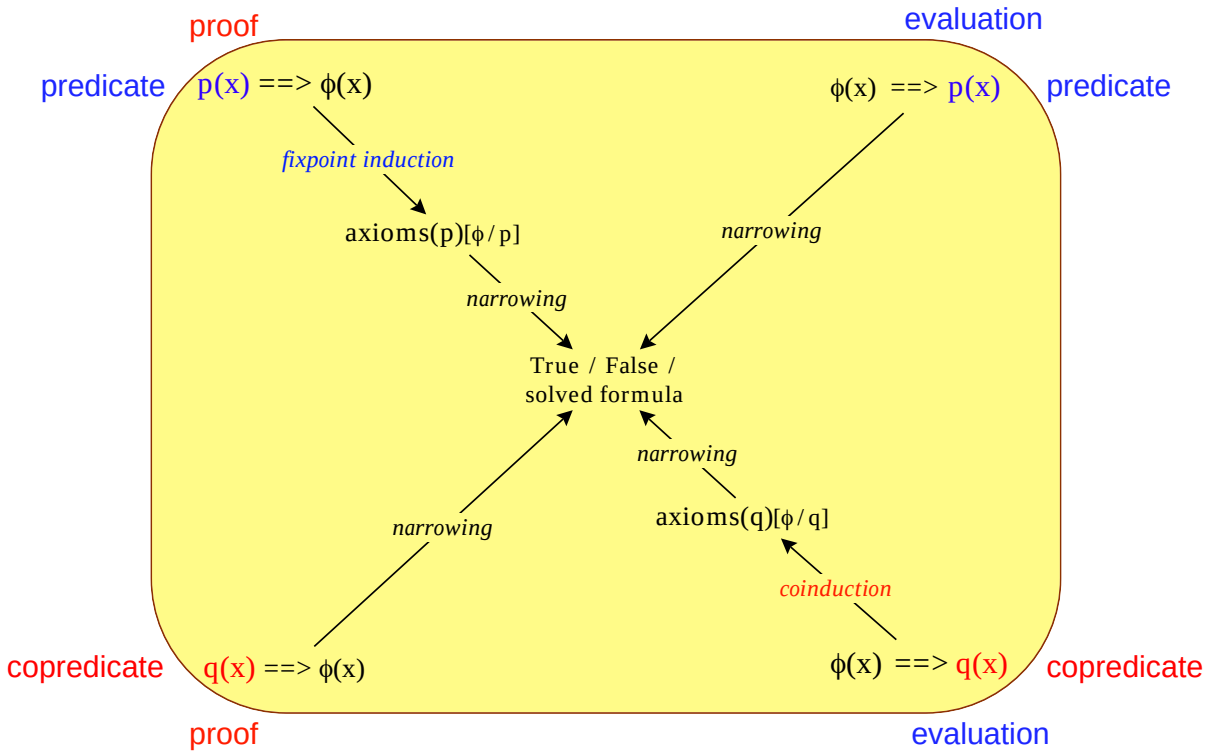


Figure 8.

Was man wie beweist/auswertet.
Je nach Redex kann mit Narrowing auch Resolution oder Coresolution gemeint sein.

7.4 Invarianzbeweise

Beispiel 7.4.1 (merge) Wir wollen Lemma (1) aus Beispiel 7.2.6 zeigen:

$$sorted(s_1) \wedge sorted(s_2) \quad \Rightarrow \quad sorted(merge(s_1, s_2)). \tag{1}$$

(1) beschreibt eine typische **Invarianzbedingung**: Wird *merge* auf sortierte Listen angewendet, dann ist auch die Ergebnisliste sortiert. Zum Beweis mit Fixpunktinduktion muss (1) generalisiert werden:

$$merge(s_1, s_2) = s \wedge sorted(s_1) \wedge sorted(s_2) \quad \Rightarrow \quad sorted(s) \wedge s \sim s1 + +s2. \tag{2}$$

(2) ist äquivalent zur Konjunktion von Lemma (1) und Lemma (3) aus Beispiel 7.2.6!

```
Derivation of

merge(s1,s2) = s & sorted(s1) & sorted(s2) ==> sorted(s) & s ~ s1++s2

Shifting subformulas at positions [0,1],[0,2] of the preceding formula leads to

merge(s1,s2) = s ==> (sorted(s1) & sorted(s2) ==> sorted(s) & s ~ s1++s2)

Applying fixpoint induction w.r.t.

merge([],s) = s & merge(s,[]) = s &
(merge(x:s,y:s') = x:z3 <=== x <= y & merge(s,y:s') = z3) &
(merge(x:s,y:s') = y:z3 <=== x > y & merge(x:s,s') = z3)

at position [] of the preceding formula leads to

All s x y s' z3:((sorted[] & sorted(s) ==> sorted(s) & s ~ []++s) &
                (sorted(s) & sorted[] ==> sorted(s) & s ~ s++[]) &
                ((sorted(x:s) & sorted(y:s') ==>
                  sorted(x:z3) & x:z3 ~ x:s++y:s') <===
                 x <= y &
                 (sorted(s) & sorted(y:s') ==> sorted(z3) & z3 ~ s++y:s')) &
                ((sorted(x:s) & sorted(y:s') ==>
                  sorted(y:z3) & y:z3 ~ x:s++y:s') <===
                 x > y &
                 (sorted(x:s) & sorted(s') ==> sorted(z3) & z3 ~ x:s++s')))

Simplifying the preceding formula (33 steps) leads to

All s x y s' z3:(x <= y & (sorted(s) & sorted(y:s') ==> sorted(z3)) &
                (sorted(s) & sorted(y:s') ==> z3 ~ s++y:s') & sorted(x:s) &
                sorted(y:s') ==>
                sorted(x:z3)) &
All s x y s' z3:(x <= y & (sorted(s) & sorted(y:s') ==> sorted(z3)) &
                (sorted(s) & sorted(y:s') ==> z3 ~ s++y:s') & sorted(x:s) &
                sorted(y:s') ==>
                x:z3 ~ x:s++y:s') &
All s x y s' z3:(x > y & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                (sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &
                sorted(y:s') ==>
                sorted(y:z3)) &
All s x y s' z3:(x > y & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                (sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &
                sorted(y:s') ==>
                y:z3 ~ x:s++y:s')

Applying the axiom resp. theorem
```

```
sorted(s) <=== sorted(x:s)

at positions [0,0,2,0,0],[0,0,1,0,0] of the preceding formula leads to 4 factors.
The current factor is given by

All s x y s' z3:(x <= y & (Any x0:(sorted(x0:s)) & sorted(y:s') ==> sorted(z3)) &
                 (Any x1:(sorted(x1:s)) & sorted(y:s') ==> z3 ~ s++y:s') &
                 sorted(x:s) & sorted(y:s') ==>
                 sorted(x:z3))

Simplifying the preceding factors (2 steps) leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & x <= y & sorted(z3) &
                 z3 ~ s++y:s' ==>
                 sorted(x:z3))

Applying the axiom resp. theorem

sorted(x:s1) <===
sorted(x:s) & sorted(y:s') & x <= y & sorted(s1) & s1 ~ s++y:s'

at position [0,1] of the preceding factors leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & x <= y & sorted(z3) &
                 z3 ~ s++y:s' ==>
                 Any s9 y0 s'0:(sorted(x:s9) & sorted(y0:s'0) & x <= y0 &
                                sorted(z3) & z3 ~ s9++y0:s'0))

Simplifying the preceding factors (2 steps) leads to 3 factors.
The current factor is given by

All s x y s' z3:(x <= y & (sorted(s) & sorted(y:s') ==> sorted(z3)) &
                 (sorted(s) & sorted(y:s') ==> z3 ~ s++y:s') & sorted(x:s) &
                 sorted(y:s') ==>
                 x:z3 ~ x:s++y:s')

Applying the axiom resp. theorem

sorted(s) <=== sorted(x:s)

at position [0,0,2,0,0] of the preceding factors leads to the factor

All s x y s' z3:(x <= y & (sorted(s) & sorted(y:s') ==> sorted(z3)) &
                 (Any x3:(sorted(x3:s)) & sorted(y:s') ==> z3 ~ s++y:s') &
                 sorted(x:s) & sorted(y:s') ==>
                 x:z3 ~ x:s++y:s')

Simplifying the preceding factors leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & x <= y &
                 (sorted(s) & sorted(y:s') ==> sorted(z3)) & z3 ~ s++y:s' ==>
                 x:z3 ~ x:s++y:s')

Decomposing the atom at position [0,1] of the preceding factors leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & x <= y &
                 (sorted(s) & sorted(y:s') ==> sorted(z3)) & z3 ~ s++y:s' ==>
                 z3 ~ s++y:s')

Simplifying the preceding factors (2 steps) leads to 2 factors.
The current factor is given by
```

```
All s x y s' z3:(x > y & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                 (sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &
                 sorted(y:s') ==>
                 sorted(y:z3))

Applying the theorem

x > y ===> y <= x

at position [0,0,0] of the preceding factors leads to the factor

All s x y s' z3:(y <= x & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                 (sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &
                 sorted(y:s') ==>
                 sorted(y:z3))

Applying the axiom resp. theorem

sorted(s) <=== sorted(x:s)

at positions [0,0,2,0,1],[0,0,1,0,1] of the preceding factors leads to the factor

All s x y s' z3:(y <= x & (sorted(x:s) & Any x5:(sorted(x5:s')) ==> sorted(z3)) &
                 (sorted(x:s) & Any x6:(sorted(x6:s')) ==> z3 ~ x:s++s') &
                 sorted(x:s) & sorted(y:s') ==>
                 sorted(y:z3))

Simplifying the preceding factors (2 steps) leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &
                 z3 ~ x:s++s' ==>
                 sorted(y:z3))

Applying the axiom resp. theorem

sorted(x:s1) <===
sorted(x:s) & sorted(y:s') & x <= y & sorted(s1) & s1 ~ s++y:s'

at position [0,1] of the preceding factors leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &
                 z3 ~ x:s++s' ==>
                 Any s13 y2 s'1:(sorted(y:s13) & sorted(y2:s'1) & y <= y2 &
                                 sorted(z3) & z3 ~ s13++y2:s'1))

Simplifying the preceding factors (3 steps) leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &
                 z3 ~ x:s++s' ==>
                 Any s13 y2 s'1:(sorted(y:s13) & z3 ~ s13++y2:s'1 &
                                 sorted(y2:s'1) & y <= y2))

Applying the theorem

s'++x:s ~ x:s++s'

at position [0,0,4] of the preceding factors leads to the factor

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x & sorted(z3) &
                 z3 = s'++x:s ==>
                 Any s13 y2 s'1:(sorted(y:s13) & z3 ~ s13++y2:s'1 &
```

```
                                sorted(y2:s'1) & y <= y2))

Simplifying the preceding factors (2 steps) leads to the factor

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>
              Any s13 y2 s'1:(sorted(y:s13) & s'++x:s ~ s13++y2:s'1 &
                              sorted(y2:s'1) & y <= y2))

Substituting s' for s13 at position [0,1,0,1,0,0] of the preceding factors
leads to the factor

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>
              Any s13 y2 s'1:(sorted(y:s') & s'++x:s ~ s'++y2:s'1 &
                              sorted(y2:s'1) & y <= y2))

Substituting x for y2 at position [0,1,0,1,0,1,0] of the preceding factors
leads to the factor

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>
              Any s13 y2 s'1:(sorted(y:s') & s'++x:s ~ s'++x:s'1 &
                              sorted(x:s'1) & y <= x))

Substituting s for s'1 at position [0,1,0,1,0,1,1] of the preceding factors
leads to the factor

All s x y s':(sorted(x:s) & sorted(y:s') & y <= x & sorted(s'++x:s) ==>
              Any s13 y2 s'1:(sorted(y:s') & s'++x:s ~ s'++x:s & sorted(x:s) &
                              y <= x))

Simplifying the preceding factors (6 steps) leads to a single formula,
which is given by

All s x y s' z3:(x > y & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                 (sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &
                 sorted(y:s') ==>
                 y:z3 ~ x:s++y:s')

Applying the theorem

x > y ===> y <= x

at position [0,0,0] of the preceding formula leads to

All s x y s' z3:(y <= x & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                 (sorted(x:s) & sorted(s') ==> z3 ~ x:s++s') & sorted(x:s) &
                 sorted(y:s') ==>
                 y:z3 ~ x:s++y:s')

Applying the axiom resp. theorem

sorted(s) <=== sorted(x:s)

at position [0,0,2,0,1] of the preceding formula leads to

All s x y s' z3:(y <= x & (sorted(x:s) & sorted(s') ==> sorted(z3)) &
                 (sorted(x:s) & Any x10:(sorted(x10:s')) ==> z3 ~ x:s++s') &
                 sorted(x:s) & sorted(y:s') ==>
                 y:z3 ~ x:s++y:s')

Simplifying the preceding formula leads to

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x &
```

```
                (sorted(x:s) & sorted(s') ==> sorted(z3)) & z3 ~ x:s++s' ==>
                y:z3 ~ x:s++y:s')

A transitivity axiom at position [0,1] of the preceding formula leads to

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x &
                (sorted(x:s) & sorted(s') ==> sorted(z3)) & z3 ~ x:s++s' ==>
                Any z4:(y:z3 ~ z4 & z4 ~ x:s++y:s'))

Applying the theorem

y:x:s++s' ~ x:s++y:s'

at position [0,1,0,1] of the preceding formula leads to

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x &
                (sorted(x:s) & sorted(s') ==> sorted(z3)) & z3 ~ x:s++s' ==>
                Any z4:(y:z3 ~ z4 & z4 = y:x:s++s'))

Substituting y:x:s++s' for z4 at position [0,1,0,1,1] of the preceding formula
leads to

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x &
                (sorted(x:s) & sorted(s') ==> sorted(z3)) & z3 ~ x:s++s' ==>
                Any z4:(y:z3 ~ y:x:s++s' & y:x:s++s' = y:x:s++s'))

Simplifying the preceding formula (6 steps) leads to

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x &
                (sorted(x:s) & sorted(s') ==> sorted(z3)) & z3 ~ x:s++s' ==>
                y:z3 ~ y:x:s++s')

Decomposing the atom at position [0,1] of the preceding formula leads to

All s x y s' z3:(sorted(x:s) & sorted(y:s') & y <= x &
                (sorted(x:s) & sorted(s') ==> sorted(z3)) & z3 ~ x:s++s' ==>
                z3 ~ x:s++s')

Simplifying the preceding formula (2 steps) leads to

True

Number of proof steps: 31
```

Beispiel 7.4.2 Wir spezifizieren den als **2-3-Bäume** bekannten Datentyp von Bäumen mit Knotenausgrad 2 oder 3 zusammen mit einer Operation *insert*, die Elemente in 2-3-Bäume einträgt und ggf. balancierte Bäume rebalanciert:

```
  2-3-TREE[ORD(s)] and NAT then
      sorts      tree
      constructs  mt:->tree                                  - leerer Baum
                  _#_#_:tree*s*tree->tree                    - Wurzel hat 2 Nachfolger
                  _#_#_#_#_:tree*s*tree*s*tree->tree         - Wurzel hat 3 Nachfolger
      defuncts    insert:s*tree->(tree*nat)
                  h:tree->nat                                - Baumhöhe
      preds       balanced:tree
      vars        x,y,z,z':s  T1,T2,T3,T4,T5:tree
      axioms      insert(x,mt) = (mt#x#mt,1)
```

```
insert(x,T1#x#T2) = (T1#x#T2,0)
insert(x,T1#x#T2#y#T3) = (T1#x#T2#y#T3,0)
insert(x,T1#y#T2#x#T3) = (T1#y#T2#x#T3,0)
insert(x,T1#y#T2) = (T3#y#T2,0)  ⇐  x < y ∧ insert(x,T1) = (T3,0)
insert(x,T1#y#T2) = (T1#y#T3,0)  ⇐  y < x ∧ insert(x,T2) = (T3,0)
insert(x,T1#y#T2#z#T3) = (T4#y#T2#z#T3,0)  ⇐  x < y ∧ insert(x,T1) = (T4,0)
insert(x,T1#y#T2#z#T3) = (T4#y#T2#z#T3,0)
                          ⇐  y < x < z ∧ insert(x,T2) = (T4,0)
insert(x,T1#y#T2#z#T3) = (T1#y#T2#z#T4,0)  ⇐  z < x ∧ insert(x,T3) = (T4,0)
insert(x,T1#y#T2) = (T3#z#T4#y#T2,0)  ⇐  x < y ∧ insert(x,T1) = (T3#z#T4,1)
insert(x,T1#y#T2) = (T1#y#T3#z#T4,0)  ⇐  y < x ∧ insert(x,T2) = (T3#z#T4,1)
insert(x,T1#y#T2#z#T3) = (T4#y#(T2#z#T3),1)
                          ⇐  x < y ∧ insert(x,T1) = (T4#z'#T5,1)
insert(x,T1#y#T2#z#T3) = ((T1#y#T4)#z'#(T5#z#T3),1)
                          ⇐  y < x < z ∧ insert(x,T2) = (T4#z'#T5,1)
insert(x,T1#y#T2#z#T3) = ((T1#y#T2)#z#T4,1)
                          ⇐  z < x ∧ insert(x,T3) = (T4#z'#T5,1)
h(mt) = 0
h(T1#x#T2) = max(h(T1),h(T2))+1
h(T1#x#T2#y#T3) = max(h(T1),h(T2),h(T3))+1
balanced(mt)
balanced(T1#x#T2) ⇐  h(T1) = h(T2) ∧ balanced(T1) ∧ balanced(T2)
balanced(T1#x#T2#y#T3)
⇐  h(T1) = h(T2) = h(T3) ∧ balanced(T1) ∧ balanced(T2) ∧ balanced(T3)
```

☞ Zeigen Sie folgende Invarianzbedingungen durch Fixpunktinduktion über *insert*:

$$\begin{aligned} insert(x,T) \equiv (T',0) \wedge balanced(T) &\Rightarrow h(T) \equiv h(T') \wedge balanced(T') \\ insert(x,T) \equiv (T',1) \wedge balanced(T) &\Rightarrow h(T)+1 \equiv h(T') \wedge balanced(T'). \end{aligned}$$ ✌

Beispiel 7.4.3 Wir spezifizieren den als **AVL-Bäume** bekannten Datentyp binärer Bäume zusammen mit einer Operation *insert*, die Elemente in AVL-Bäume einträgt und ggf. balancierte Bäume rebalanciert:

```
AVL-TREE[ORD(s)] = NAT then
    sorts  bintree   nonempty
  constructs  mt:->tree                              - leerer Baum
              _:nonempty->tree                       - nichtleerer Baum
              _#_#_:tree*s*tree->nonempty            - ausgeglichener Baum
              [_]#_#_:nonempty*s*tree->nonempty      - linkslastiger Baum
              _#_#[_]:tree*s*nonempty->nonempty      - rechtslastiger Baum
    defuncts  insert:s*tree->nonempty
              [_]#_#_:nonempty*s*tree->nonempty      - Rotation von links her
              _#_#[_]:tree*s*nonempty->nonempty      - Rotation von rechts her
              h:tree->nat                            - Baumhöhe
    preds     balanced:tree
    vars      x,y,z:s  T1,T2,T3,T4:tree
    axioms    insert(x,mt) = mt#x#mt
              insert(x,T1#x#T2) = T1#x#T2
```

```
     insert(x,T1#x#T2) = T1#x#T2
     insert(x,T1#x#T2) = T1#x#T2
     insert(x,T1#y#T2) = T3#y#T2  ⇐  x < y ∧ insert(x,T1) = T3 ∧ h(T1) = h(T3)
     insert(x,T1#y#T2) = T1#y#T3  ⇐  y < x ∧ insert(x,T2) = T3 ∧ h(T1) = h(T3)
     insert(x,T1#y#T2) = T3#y#T2  ⇐  x < y ∧ insert(x,T1) = T3 ∧ h(T1) = h(T3)
     insert(x,T1#y#T2) = T1#y#T3  ⇐  y < x ∧ insert(x,T2) = T3 ∧ h(T1) = h(T3)
     insert(x,T1#y#T2) = T3#y#T2  ⇐  x < y ∧ insert(x,T1) = T3 ∧ h(T1) = h(T3)
     insert(x,T1#y#T2) = T1#y#T3  ⇐  y < x ∧ insert(x,T2) = T3 ∧ h(T1) = h(T3)
     insert(x,T1#y#T2) = T3#y#T2  ⇐  x < y ∧ insert(x,T1) = T3 ∧ h(T1) ≠ h(T3)
     insert(x,T1#y#T2) = T1#y#T3  ⇐  y < x ∧ insert(x,T2) = T3 ∧ h(T2) ≠ h(T3)
     insert(x,T1#y#T2) = T3#y#T2  ⇐  x < y ∧ insert(x,T1) = T3 ∧ h(T1) ≠ h(T3)
     insert(x,T1#y#T2) = T1#y#T3  ⇐  y < x ∧ insert(x,T2) = T3 ∧ h(T2) ≠ h(T3)
     insert(x,T1#y#T2) = T3#y#T2  ⇐  x < y ∧ insert(x,T1) = T3 ∧ h(T1) ≠ h(T3)
     insert(x,T1#y#T2) = T1#y#T3  ⇐  y < x ∧ insert(x,T2) = T3 ∧ h(T2) ≠ h(T3)
(1)  T1#x#T2#y#T3 = T1#x#(T2#y#T3)
(2)  T1#x#T2#y#T3 = T1#x#(T2#y#T3)
(3)  T1#x#T2#y#T3#z#T4 = (T1#x#T2)#y#(T3#z#T4)
(4)  T1#x#T2#y#T3#z#T4 = (T1#x#T2)#y#(T3#z#T4)
(5)  T1#x#T2#y#T3#z#T4 = (T1#x#T2)#y#(T3#z#T4)
(6)  T1#x#T2#y#T3 = T1#x#T2#y#T3
(7)  T1#x#T2#y#T3 = (T1#x#T2)#y#T3
(8)  T1#x#T2#y#T3#z#T4 = (T1#x#T2)#y#(T3#z#T4)
(9)  T1#x#T2#y#T3#z#T4 = (T1#x#T2)#y#(T3#z#T4)
(10) T1#x#T2#y#T3#z#T4 = (T1#x#T2)#y#(T3#z#T4)
     h(mt) = 0
     h(T1#x#T2) = max(h(T1),h(T2))+1
     h(T1#x#T2) = max(h(T1),h(T2))+1
     h(T1#x#T2) = max(h(T1),h(T2))+1
     balanced(mt)
     balanced(T1#x#T2) ⇐  h(T1) = h(T2) ∧ balanced(T1) ∧ balanced(T2)
     balanced(T1#x#T2) ⇐  h(T1) = h(T2)+1 ∧ balanced(T1) ∧ balanced(T2)
     balanced(T1#x#T2) ⇐  h(T1)+1 = h(T2) ∧ balanced(T1) ∧ balanced(T2)
```

Wieder ist die wesentliche Anforderung an *insert* eine Invarianzbedingung:

$$balanced(T) \quad \Rightarrow \quad balanced(insert(x,T)).$$

☞ Zeigen Sie durch Fixpunktinduktion über *insert* und die beiden Rotationsfunktionen:

$$\begin{array}{lll} insert(x,T) \equiv T' & \Rightarrow & (balanced(T) \quad \Rightarrow \quad balanced(T')) \\ \boxed{T1}\#x\#T2 \equiv T & \Rightarrow & ((balanced(T1) \wedge balanced(T2) \wedge h(T1) \equiv h(T2)+2) \quad \Rightarrow \quad balanced(T)) \\ T1\#x\#\boxed{T2} \equiv T & \Rightarrow & ((balanced(T1) \wedge balanced(T2) \wedge h(T1)+2 \equiv h(T2)) \quad \Rightarrow \quad balanced(T)). \end{array}$$

☞ Optimieren Sie – ähnlich der Einfügeoperation bei 2-3-Bäumen (s. Bsp. 7.4.2) – *insert* durch Erweiterung des Wertebereiches der Funktion um einen Booleschen Wert (*flag*), der angibt, ob der jeweilige Aufruf von *insert*

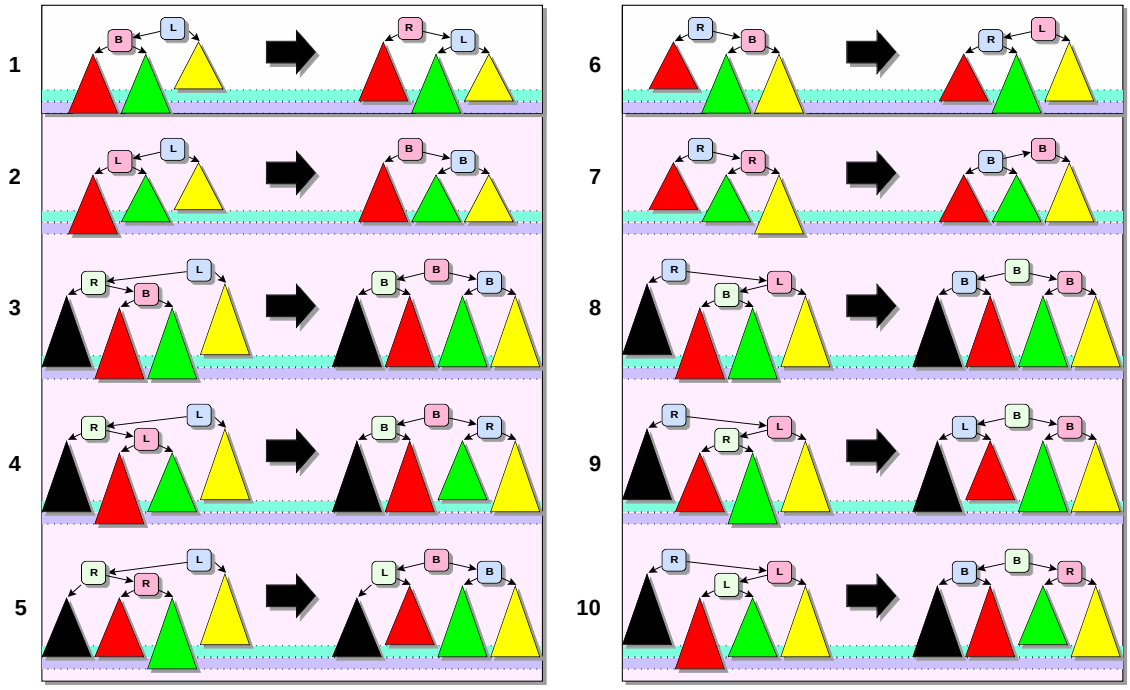


Figure 9. *Die 10 Rotationsfälle (nach A. Hallmann)*

die Höhe des Argumentbaumes verändert hat! Anstatt die Bedingungen $h(T1) = h(T3)$, $h(T1) \neq h(T3)$, etc., zu überprüfen, braucht *insert* dann nur den Wert des flags abzufragen. Zeigen Sie, dass Ihr optimiertes *insert* zum *insert* von AVL-TREE äquivalent ist!

☞ Spezifizieren Sie eine Operation $remove : s \times nonempty \rightarrow tree$, die Einträge entfernt und wie *insert* die Balanciertheit von Bäumen erhält! Da eine Löschoperation die Höhe eines Baumes verringern kann, müssen ggf. Teilbäume mehrfach rotiert werden, was zu rekursiven Aufrufen der Rotationsfunktionen führt. ✌

Beispiel 7.4.4 (Baumordnungen) Infixordnung (auch Suchbaumeigenschaft genannt) und partielle Ordnung sind zwei Arten, die Struktur eines binären Baumes auszunutzen, um Einträge zu sortieren und damit die Zugriffszeit zu verkürzen.[53] Für Suchbäume bieten sich Konstruktoren an, die den leeren Baum ausschließen.

```
TREE-ORDERS[ORD(s)] = AVL-TREE[ORD(s)] then
      sorts      searchtree
      constructs leaf:s -> searchtree
                 _&_&_:searchtree*s*searchtree -> searchtree
      defuncts   minT,maxT:searchtree -> s
                 bin:searchtree -> tree
      preds      infix:searchtree
                 parti:tree
      axioms     infix(leaf(x))
                 infix(T&x&T')  <==  infix(T) /\ maxT(T) <= x /\ x <= minT(T') /\ infix(T')
                 minT(leaf(x)) = x
                 minT(T&x&T') = (min(minT(T),min(x,minT(T'))))
                 maxT(leaf(x)) = x
                 maxT(T&x&T') = (max(maxT(T),max(x,maxT(T'))))
                 bin(leaf(x)) = mt#x#mt
                 bin(T&x&T')  = bin(T)#x#bin(T')
                 parti(mt)
                 parti(mt#x#(T1#y#T2))          <==  x <= y /\ parti(T1#x#T2)
                 parti((T1#x#T2)#y#mt)          <==  y <= x /\ parti(T1#x#T2)
```

[53] Alle Axiome mit dem Konstruktor _#_#_ gibt es implizit auch für □#_#_ und _#_#□.

```
parti((T1#x#T2)#y#(T3#z#T4))  <==  y <= x /\ parti(T1#x#T2) /\
                                   y <= z /\ parti(T3#z#T4)
```

☞ Zeigen Sie, dass die Invarianzbedingung

$$infix(T) \wedge insert(x, bin(T)) \equiv bin(T') \quad \Rightarrow \quad infix(T')$$

ein induktives Theorem von TREE-ORDERS ist, dass also *insert* die Infixordnung erhält!

☞ Zeigen Sie, dass auch Ihre spezifizierte Löschfunktion *remove* Suchbäume in Suchbäume überführt!

☞ Zeigen Sie, dass die Invarianzbedingung

$$parti(T1) \wedge parti(T2) \quad \Rightarrow \quad parti(sift(T1\#x\#T2))$$

ein induktives Theorem von HEAP ist (s. Beispiel 6.4.5)! Folgern Sie daraus die Gültigkeit von $parti(heapify(T))$! ✌

7.5 Schleifeninvarianten

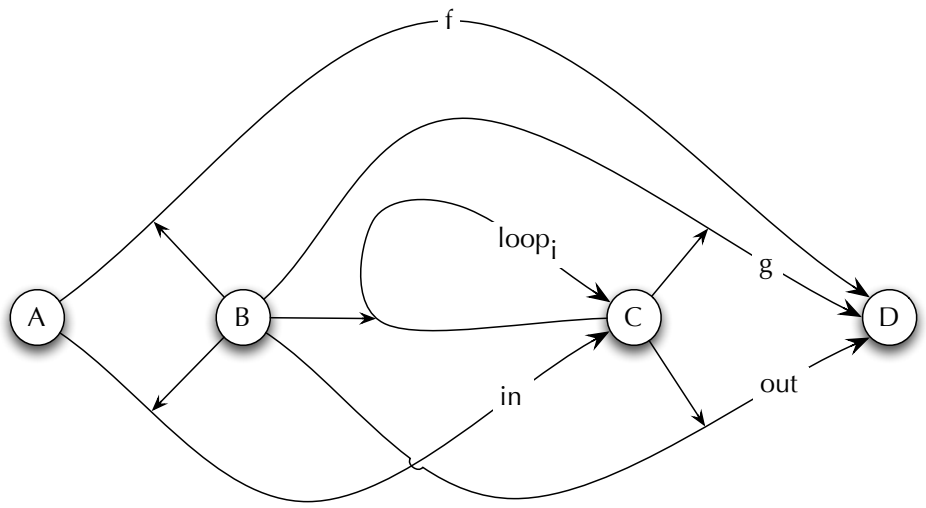


Figure 10. *Die Funktionen eines iterativen Programms: die Ausgangsfunktion f, deren Schleifenfunktion g, die Initialisierung in der Schleifenvariablen, deren mögliche Modifikationen $loop_i$ und die Projektion vom Schleifenbereich C in den Ausgabebereich D.*

Eine bestimmte Struktur der Axiome für eine Funktion oder ein Prädikat nennt man auch **Programmschema**. Eins der gebräuchlichsten ist das für iterative Programme charakteristische, das aus vier Sorten A, B, C und D, einer Funktion $f : A \times B \to D$, einer **Schleifenfunktion** $g : B \times C \to D$, Hilfsfunktionen $in : A \times B \to C$, $out : B \times C \to D$ und $loop_1, \ldots, loop_n : B \times C \to C$ sowie einer Axiomenmenge der Form

$$\begin{array}{l} f(a,b) \equiv g(b, in(a,b)) \\ g(b,c) \equiv g(b, loop_1(b,c)) \;\Leftarrow\; \delta_1(b,c) \\ \ldots \\ g(b,c) \equiv g(b, loop_n(b,c)) \;\Leftarrow\; \delta_n(b,c) \\ g(b,c) \equiv out(b,c) \;\Leftarrow\; \delta(b,c) \end{array} \tag{1}$$

besteht. Daß (1) tatsächlich einer iterativen Definition entspricht, zeigt die folgende Implementierung der Axiome als Funktionsprozedur mit Schleifenrumpf:

```
function f(a:A,b:B):D;
    begin c:C;
          c:=in(a,b);
          while δ_1(b,c) ∨ ··· ∨ δ_n(b,c)
          do if δ_1(b,c) then c:=loop_1(b,c);                    (2)
             ...
             if δ_n(b,c) then c:=loop_n(b,c)
          od;
          if δ(b,c) then return out(b,c)
    end
```

Welche Bedingung muss an die Spezifikation, in die (1) eingebettet ist, gestellt werden, damit diese Implementierung von (1) korrekt ist?

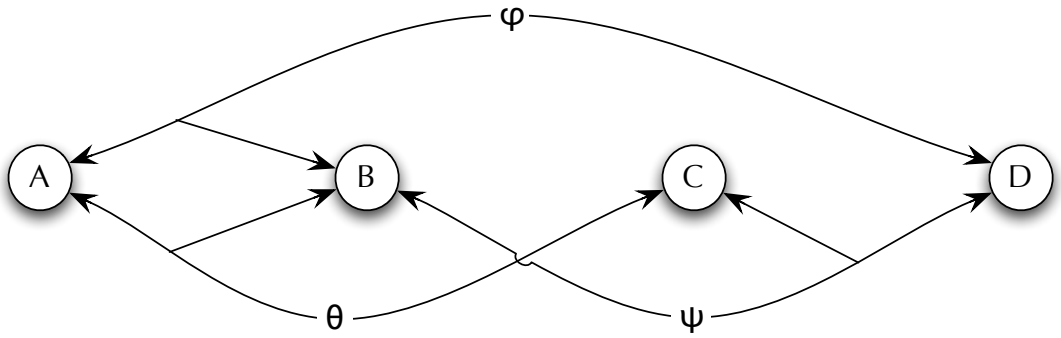


Figure 11. *Die Relationen eines iterativen Programms: die (gewünschte) Ein/Ausgaberelation φ von f, die Hoare-Invariante θ und die Subgoal-Invariante ψ, die der (gewünschten) Ein/Ausgaberelation der Schleifenfunktion g entspricht.*

Der Verifikation von Schleifenprogrammen dienen zwei komplementäre Verfahren: **Hoare-Induktion** und **Subgoal-Induktion** (siehe z.B. §10.2 oder [61]).

Subgoal-Induktion ist Fixpunktinduktion über f (Beweis der E/A-Relation $\varphi(a,b,d)$ von f) mit anschließender Fixpunktinduktion über g (Beweis der **Subgoal-Invarianten**). Die Subgoal-Invariante $\psi(b,c,d)$ setzt Werte der **Schleifenvariablen** b,c mit Werten der **Ausgabevariable** d in Beziehung. Die an ψ gestellten Bedingungen bilden die Konklusion folgender Regel:

Subgoal-Invarianten-Erzeugung

$$\frac{\begin{array}{l} f(a,b) \equiv d \;\Rightarrow\; \varphi(a,b,d) \\ \psi(b,in(a,b),d) \;\Rightarrow\; \varphi(a,b,d) \\ \wedge \quad g(b,c) \equiv d \;\Rightarrow\; \psi(b,c,d) \end{array}}{} \Uparrow$$

Der zweite Faktor der Konklusion läßt sich dann durch Fixpunktinduktion über g beweisen (siehe §7.2). Die Axiome für g sind durch (1) gegeben.

Dual dazu beschreibt eine **Hoare-Invariante** $\theta(a,b,c)$ einen Zusammenhang zwischen Werten der **Eingabevariablen** a,b und Werten der Schleifenvariable c. Eine Hoare-Invariante θ läßt sich folgendermaßen aus einer Subgoal-Invarianten ψ konstruieren:

$$\theta(a,b,c) \quad \Longleftrightarrow_{def} \quad \forall d : (\psi(b,c,d) \Rightarrow \varphi(a,b,d))$$

Umgekehrt liefert jede Hoare-Invariante θ eine Subgoal-Invariante ψ:

$$\psi(b,c,d) \quad \Longleftrightarrow_{def} \quad \forall a : (\theta(a,b,c) \Rightarrow \varphi(a,b,d))$$

Mit dieser Beziehung erhält man die Korrektheit Regel zur Erzeugung von Hoare-Invarianten aus der Korrektheit der o.g. Regel zur Erzeugung von Subgoal-Invarianten:

Hoare-Invarianten-Erzeugung

$$\frac{f(a,b) \equiv d \;\Rightarrow\; \varphi(a,b,d)}{\begin{array}{ll} & \theta(a,b,in(a,b)) \\ \wedge & g(b,c) \equiv d \;\Rightarrow\; (\theta(a,b,c) \Rightarrow \varphi(a,b,d)) \end{array}} \Uparrow$$

Der zweite Faktor der Konklusion läßt sich wieder durch Fixpunktinduktion über g beweisen.

Beide Regeln sind natürlich auch dann korrekt, wenn die Axiome für g nicht dem iterativen Schema (1) folgen. f muss allerdings wie in (1) durch das Axiom $f(a,b) \equiv g(b, in(a,b))$ definiert sein.

Beispiel 7.5.1 Aufbauend auf Bsp. 7.2.1 beweisen wir die Korrektheit eines iterativen Programms zur Zerlegung einer Liste in Teillisten fester Länge mit Hoare-Induktion.

```
PARTN = PARTITION and NAT then
     defuncts  partn:nat*list->list(list)
               g:nat*list*nat*list*list(list)->list(list)
               length:list->nat
     preds     lgOk:nat*list(list)
     vars      k,n:nat  x:s  s,s':list  p,p':list(list)
               partn(s,n) = g(n,s,0,[],[])
               g(n,[],k,s,p) = p++[s]
               g(n,x:s,k,s',p) = g(n,s,k+1,s'++[x],p) <=== k < n
               g(n,x:s,k,s',p) = g(n,x:s,0,[],p++[s']) <=== k > n
               length[] = 0
               length(x:s) = length(s)+1
               lgOk(n,[])
               lgOk(n,[s])  <===  length(s) <= n
               lgOk(n,s:s':p)  <===  length(s) = n /\ lgOk(n,s':p)
     theorems: flatten(p++p') = flatten(p)++flatten(p')
               lgOk(n,p++[s]) <=== lgOk(n,p) /\ length(s) <= n
     conjects: partn(s,n) = p ==> s = flatten(p)
               partn(s,n) = p ==> lgOk(n,p)
```

Die beiden letzten Formeln sind die Korrektheitsbedingungen an *partn*. Aus den Argumentsorten von *partn* und g folgt, dass eine Hoare-Invariante den Typ

$$nat \times list \times list \times nat \times list \times list(list)$$

haben muss. Zum Beweis von (3) definieren wir sie durch das Axiom:

$$INV(s,n,s_1,k,s_2,p) \quad \Leftarrow \quad s \equiv flatten(p + +[s_2 + +s_1]).$$

```
Derivation of

partn(s,n) = p ==> s = flatten(p)

Hoare invariant creation for the iterative program

partn(s,n) = g(n,s,0,[],[])

at position [] of the preceding formula leads to

INV(s,n,s,0,[],[]) &
(g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6) ==> s = flatten(p))

Narrowing at position [0] of the preceding formula (3 steps) leads to
```

```
True & (g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6) ==> s = flatten(p))

Reductsh have been simplified.

Simplifying the preceding formula leads to

g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6) ==> s = flatten(p)

Shifting subformulas at position [0,1] of the preceding formula leads to

g(n,z3,z4,z5,z6) = p ==> (INV(s,n,z3,z4,z5,z6) ==> s = flatten(p))

Applying fixpoint induction w.r.t.

g(n,[],k,s,p) = p++[s] &
(g(n,x:s,k,s',p) = z7 <=== k < n & g(n,s,k+1,s'++[x],p) = z7) &
(g(n,x:s,k,s',p) = z7 <=== k > n & g(n,x:s,0,[],p++[s']) = z7)

at position [] of the preceding formula leads to

All n k s p x s' z7:(All s6:(INV(s6,n,[],k,s,p) ==> s6 = flatten(p++[s])) &
                     (All s7:(INV(s7,n,x:s,k,s',p) ==> s7 = flatten(z7)) <===
                      k < n &
                      All s7:(INV(s7,n,s,k+1,s'++[x],p) ==> s7 = flatten(z7))) &
                     (All s8:(INV(s8,n,x:s,k,s',p) ==> s8 = flatten(z7)) <===
                      k > n &
                      All s8:(INV(s8,n,x:s,0,[],p++[s']) ==> s8 = flatten(z7))))

Applying the axiom resp. theorem

INV(s,n,s1,k,s2,p) <=== s = flatten(p++[s2++s1])

at positions [0,2,1,1,0,0],[0,2,0,0,0],[0,1,1,1,0,0],[0,1,0,0,0],[0,0,0,0] of the preceding formula
leads to

All n k s p x s' z7:(All s6:(s6 = flatten(p++[s++[]]) ==> s6 = flatten(p++[s])) &
                     (All s7:(s7 = flatten(p++[s'++x:s]) ==> s7 = flatten(z7)) <===
                      k < n &
                      All s7:(s7 = flatten(p++[s'++[x]++s]) ==>
                              s7 = flatten(z7))) &
                     (All s8:(s8 = flatten(p++[s'++x:s]) ==> s8 = flatten(z7)) <===
                      k > n &
                      All s8:(s8 = flatten(p++[s']++[[]++x:s]) ==>
                              s8 = flatten(z7))))

Simplifying the preceding formula (29 steps) leads to

All n k s p x s' z7:(k > n & flatten(p++[s',x:s]) = flatten(z7) ==>
                     flatten(p++[s'++x:s]) = flatten(z7))

Applying the axiom resp. theorem

flatten(p++p') = flatten(p)++flatten(p')

at positions [0,1],[0,0,1] of the preceding formula leads to

All n k s p x s' z7:(k > n & flatten(p)++flatten[s',x:s] = flatten(z7) ==>
                     flatten(p)++flatten[s'++x:s] = flatten(z7))

Applying the axiom resp. theorem
```

```
flatten(s:p) = s++flatten(p)

at positions [0,1],[0,0,1] of the preceding formula leads to

All n k s p x s' z7:(k > n & flatten(p)++s'++flatten[x:s] = flatten(z7) ==>
                     flatten(p)++s'++x:s++flatten[] = flatten(z7))

Applying the axioms

  flatten[] = []
& flatten(s:p) = s++flatten(p)

at positions [0,1],[0,0,1] of the preceding formula leads to

All n k s p x s' z7:(k > n & flatten(p)++s'++x:s++flatten[] = flatten(z7) ==>
                     flatten(p)++s'++x:s++[] = flatten(z7))

Applying the axiom resp. theorem

flatten[] = []

at position [0,0,1] of the preceding formula leads to

All n k s p x s' z7:(k > n & flatten(p)++s'++x:s++[] = flatten(z7) ==>
                     flatten(p)++s'++x:s++[] = flatten(z7))

Simplifying the preceding formula (4 steps) leads to

True

Number of proof steps: 12
```

Zum Beweis der zweiten Bedingung an *partn* definieren wir eine Hoare-Invariante durch das Axiom:

$$INV(s, n, s_1, k, s_2, p) \quad \Leftarrow \quad lgOk(n, p) \wedge k = length(s_2) \wedge k <= n$$

```
Derivation of

partn(s,n) = p ==> lgOk(n,p)

Hoare invariant creation for the iterative program

partn(s,n) = g(n,s,0,[],[])

at position [] of the preceding formula leads to

INV(s,n,s,0,[],[]) &
(g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6) ==> lgOk(n,p))

Narrowing at position [0] of the preceding formula (2 steps) leads to

True & (g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6) ==> lgOk(n,p))

Reducts have been simplified.

Simplifying the preceding formula leads to

g(n,z3,z4,z5,z6) = p & INV(s,n,z3,z4,z5,z6) ==> lgOk(n,p)

Shifting subformulas at position [0,1] of the preceding formula leads to
```

```
g(n,z3,z4,z5,z6) = p ==> (INV(s,n,z3,z4,z5,z6) ==> lgOk(n,p))

Applying fixpoint induction w.r.t.

g(n,[],k,s,p) = p++[s] &
(g(n,x:s,k,s',p) = z7 <=== k < n & g(n,s,k+1,s'++[x],p) = z7) &
(g(n,x:s,k,s',p) = z7 <=== k > n & g(n,x:s,0,[],p++[s']) = z7)

at position [] of the preceding formula leads to

All n k s p s4:(INV(s4,n,[],k,s,p) ==> lgOk(n,p++[s])) &
All n k s p x s' z7 s5:(k < n &
                        All s5:(INV(s5,n,s,k+1,s'++[x],p) ==> lgOk(n,z7)) &
                        INV(s5,n,x:s,k,s',p) ==>
                        lgOk(n,z7)) &
All n k s p x s' z7 s6:(k > n &
                        All s6:(INV(s6,n,x:s,0,[],p++[s']) ==> lgOk(n,z7)) &
                        INV(s6,n,x:s,k,s',p) ==>
                        lgOk(n,z7))

Reducts have been simplified.

Applying the axiom resp. theorem

INV(s,n,s1,k,s2,p) <=== lgOk(n,p) & k = length(s2) & k <= n

at positions [2,0,0,2],[2,0,0,1,0,0],[1,0,0,2],[1,0,0,1,0,0],[0,0,0] of the preceding formula
leads to

All n s p:(length(s) <= n & lgOk(n,p) ==> lgOk(n,p++[s])) &
All n p s' z7:(length(s') <= n & length(s') < n &
               (lgOk(n,p) & length(s')+1 <= n ==> lgOk(n,z7)) & lgOk(n,p) ==>
               lgOk(n,z7))

Reducts have been simplified.

Applying the axiom resp. theorem

lgOk(n,p++[s]) <=== lgOk(n,p) & length(s) <= n

at position [0,0,1] of the preceding formula leads to

All n p s' z7:(length(s') <= n & length(s') < n &
               (lgOk(n,p) & length(s')+1 <= n ==> lgOk(n,z7)) & lgOk(n,p) ==>
               lgOk(n,z7))

Applying the theorem

k < n ===> k+1 <= n

at position [0,0,1] of the preceding formula leads to

True

Reducts have been simplified.

Number of proof steps: 8
```

Beispiel 7.5.2 Wir beweisen die Korrektheit eines iterativen Programms zur Berechnung von Quotient und Rest der Division natürlicher Zahlen mit Expander2.

```
           DIVLOOP = NAT then
                defuncts  div:nat->nat*nat
                          loop:nat*nat*nat->nat*nat
                preds     INV:nat*nat*nat*nat           -- Hoare-Invariante
                vars      x,y,q,r:nat
                axioms    div(x,y) = loop(y,0,x)
                          loop(y,q,r) = (q,r) <=== r < y
                          loop(y,q,r) = loop(y,q+1,r-y) <=== r >= y
                          INV(x,y,q,r) <=== x = (y*q)+r

Derivation of

div(x,y) = (q,r) ==> x = (y*q)+r & r < y

Hoare invariant creation for the iterative program

div(x,y) = loop(y,0,x)

at position [] of the preceding formula leads to

INV(x,y,0,x) &
(loop(y,z1,z2) = z0 & INV(x,y,z1,z2) ==> (z0 = (q,r) ==> x = (y*q)+r & r < y))

Narrowing at position [0] of the preceding formula leads to

True &
(loop(y,z1,z2) = z0 & INV(x,y,z1,z2) ==> (z0 = (q,r) ==> x = (y*q)+r & r < y))

Reductes have been simplified.

Simplifying the preceding formula leads to

loop(y,z1,z2) = z0 & INV(x,y,z1,z2) ==> (z0 = (q,r) ==> x = (y*q)+r & r < y)

Shifting subformulas at position [0,1] of the preceding formula leads to

loop(y,z1,z2) = z0 ==>
(INV(x,y,z1,z2) ==> (z0 = (q,r) ==> x = (y*q)+r & r < y))

Applying fixpoint induction w.r.t.

(loop(y,q,r) = (q,r) <=== r < y) &
(loop(y,q,r) = z3 <=== r >= y & loop(y,q+1,r-y) = z3)

at position [] of the preceding formula leads to

All y q r z3:((All x q0 r0:(INV(x,y,q,r) ==>
                           ((q,r) = (q0,r0) ==> x = (y*q0)+r0 & r0 < y)) <===
               r < y) &
              (All x q1 r1:(INV(x,y,q,r) ==>
                           (z3 = (q1,r1) ==> x = (y*q1)+r1 & r1 < y)) <===
               r >= y &
               All x q1 r1:(INV(x,y,q+1,r-y) ==>
                           (z3 = (q1,r1) ==> x = (y*q1)+r1 & r1 < y))))

Applying the axiom resp. theorem

INV(x,y,q,r) <=== x = (y*q)+r

at positions [0,1,1,1,0,0],[0,1,0,0,0],[0,0,0,0,0] of the preceding formula leads to

All y q r z3:((All x q0 r0:(x = (y*q)+r ==>
```

```
                    ((q,r) = (q0,r0) ==> x = (y*q0)+r0 & r0 < y)) <===
 r < y) &
(All x q1 r1:(x = (y*q)+r ==>
                    (z3 = (q1,r1) ==> x = (y*q1)+r1 & r1 < y)) <===
 r >= y &
 All x q1 r1:(x = (y*(q+1))+(r-y) ==>
                    (z3 = (q1,r1) ==> x = (y*q1)+r1 & r1 < y))))

Simplifying the preceding formula (57 steps) leads to

All y q r q1 r1:(r >= y & All q1 r1:((y*(q+1))+(r-y) = (y*q1)+r1) &
                  All r1:(r1 < y) ==>
                  (y*q)+r = (y*q1)+r1)

Applying the axiom resp. theorem

(x*(y+1))+(z-x) = (x*y)+z

at position [0,0,1,0] of the preceding formula leads to

All y q r q1 r1:(r >= y & All q1 r1:((y*q)+r = (y*q1)+r1) & All r1:(r1 < y) ==>
                  (y*q)+r = (y*q1)+r1)

Simplifying the preceding formula (2 steps) leads to

True

Number of proof steps: 9
```

Beispiele 7.5.3 Es folgen einige z.T. nichttriviale iterative Programme mit ihrer jeweiligen Korrektheitsbedingung φ und Hoare-Invarianten θ. Hilfsfunktionen werden nicht explizit spezifiziert. Jede vom Programm nicht veränderte gemeinsame Komponente von Eingabe- und Schleifenvariable taucht nur einmal als Argument von θ auf.

Summe

`sum(x) = sum1(x,0)`

`sum1(y,z) = if y > 0 then sum1(y-1,y+z) else z`

$\varphi(x,z) \iff_{def} z = \sum_{i=1}^{x} i.$

$\theta(x,y,z) \iff_{def} \sum_{i=1}^{x} i = (\sum_{i=1}^{y} i) + z.$

Fakultät

`fact(x) = fact1(x,1)`

`fact1(y,z) = if y > 0 then fact1(y-1,y*z) else z`

$\varphi(x,z) \iff_{def} z = x!.$

$\theta(x,y,z) \iff_{def} x! = y! * z.$

Fibonacci-Zahlen

`fib(x) = fib1(x,0,1)`

`fibl1(x,y,z) = if x > 0 then fib1(x-1,z,y+z) else y`

$\varphi(x,z) \iff_{def}$ z ist die x-te Fibonacci-Zahl.

$\theta(n,x,y,z) \iff_{def}$ $n \geq x$ und y ist die $(n-x)$-te und z die $(n-x+1)$-te Fibonacci-Zahl.

Ganzzahlige Wurzel

`root(x) = root1(x,0,1)`

`root1(x,y,z) = if x >= z then root1(x,y+1,z+2y+3) else y`

$\varphi(x,z) \iff_{def} z = \lceil \sqrt{x} \rceil.$

$\theta(x,y,z) \iff_{def} x \geq y^2 \wedge z = (y+1)^2.$

Größter gemeinsamer Teiler

```
gcd(x,y) = gcd1(x,y)
gcd1(x,y) = if x > y then gcd1(x-y,y) else if x < y then gcd1(x,y-x) else y
```

$\varphi(x,y,z) \iff_{def} z = ggT(x,y)$.

$\theta(m,n,x,y) \iff_{def} ggT(m,n) = ggT(x,y)$.

Quotient und Rest (Beispiel 7.5.2)

```
div(x,y) = div1(y,0,x)
div1(y,q,r) = if r >= y then div1(y,q+1,r-y) else (q,r)
```

$\varphi(x,y,q,r) \iff_{def} x = y * q + r \wedge r < y$.

$\theta(x,y,q,r) \iff_{def} x = y * q + r$.

Kürzeste Wege in Graphen des Typs nat × nat → nat nach Floyd und Warshall (vgl. [77], §6.2)

```
minGraph(G) = minGraph1(G,1)
minGraph1(G,i) = if i <= N then minGraph1(λ(j,k).min(G(j,k),G(j,i)+G(i,k)),i+1) else G
```

$\varphi(G,G') \iff_{def}$ G und G' sind markierte gerichtete Graphen mit der Knotenmenge $\{1,\ldots,N\}$. Jede Kante (j,k) von G' ist markiert mit der Länge des kürzesten Weges, der in G von j nach k führt.

$\theta(G,G',i) \iff_{def}$ G und G' sind markierte gerichtete Graphen mit der Knotenmenge $\{1,\ldots,N\}$. Jede Kante (j,k) von G' ist markiert mit der Länge des kürzesten Weges, der in G von j nach k führt und nur Knoten der Menge $\{1,\ldots,i-1\}$ passiert.

Tiefensuche in Graphen des Typs $s \to list(s)$ (vgl. [77], §6.3)

```
search(G,L) = search1(G,L,[])
search1(G,[],V) = V
search1(G,x:L,V) = if x in V then search1(G,L,V) else search1(G,G(x)++L,x:V)
```

$\varphi(G,L,V) \iff_{def}$ V ist eine Liste aller (von einem Knoten) von L aus erreichbaren Knoten von G.

$\theta(G,L,L',V) \iff_{def}$ Jeder von L aus erreichbare Knoten von G liegt in V oder ist von L' aus auf einem Weg erreichbar, der keinen Knoten von V passiert.

Eine Subgoal-Invariante ist hier leichter zu finden:

$\psi(G,L,V,V') \iff_{def}$ V' besteht aus den Knoten von V und den von $L \setminus V$ aus erreichbaren Knoten von G.

Minimaler Spannbaum nach Kruskal (vgl. [77], §6.4)

```
KRUSKAL = LIST{nat/s}[NAT] and ORD(label) and LIST{edge/s}[NAT] and LIST{list(nat)/s}[LIST{nat/s}[NAT]] then
   sorts     edge = nat*label*nat
             graph = list(edge)
             partition = list(list(nat))
   defuncts  maxnode:->nat
             firstPart:->partition
             spanTree:graph->graph
             spanTree1:graph*partition*graph->graph
   preds     _<=_:edge*edge
   vars      i,j,m,n:nat  x,y:label  G,T:graph  L,L',L1,L2:list(nat)  P:partition
   axioms    (i,x,j) <= (m,y,n)  <==  x <= y
             firstPart = [[1],...,[maxnode]]
             spanTree(G) = spanTree1(sort(G),firstPart,[])
             spanTree1([],L:L':P,T) = []                    -- Hier ist G ist unzusammenhaengend.
             spanTree1(G,[L],T) = T
             spanTree1((i,x,j):G,L:L':P,T)
              = if L1 = L2 then spanTree1(G,L:L':P,T)
                           else spanTree1(G,(L1++L2):remove(L1,remove(L2,L:L':P)),(i,x,j):T)
                <==  i in L1 /\ j in L2
```

$\varphi(G,T) \quad \Longleftrightarrow_{def}$ G ist ein markierter Graph mit der Knotenmenge $\{1,\ldots,N\}$ und T ist ein minimaler Spannbaum von G.

$\theta(G,G',P,T) \Longleftrightarrow_{def}$ G und G' sind markierte Graphen mit der Knotenmenge $\{1,\ldots,N\}$ und P enthält alle Knoten von G. Jede Kante von $G \setminus G'$ verbindet zwei Knoten derselben Liste von P. Für alle Knotenlisten L von P ist der von L erzeugte Teilgraph von T ein minimaler Spannbaum des von L erzeugten Teilgraphen von G.

Optimaler Binärcode nach Huffman (vgl. [77], §7.3)

```
BINTREE = LIST[TRIV(s)] then
  sorts       tree
  constructs  leaf:list(s)->tree
              _#_#_:tree*list(s)*tree->tree
  defuncts    root:tree->list(s)
  vars        T,T':tree  cL:list(s)
  axioms      root(T#cL#T') = cL

HUFFMAN = LIST{tree/s}[BINTREE] and NAT then
  defuncts    minOf2:(s->nat)*tree*tree->tree
              minOfList:(s->nat)*list(tree)->tree
              firstList:list(s)->list(tree)
              newList:list(s)*list(tree)->list(tree)
              Huffman:(s->nat)*list(s)->tree
              Huffman1:(s->nat)*list(tree)->tree
  vars        T,T',T1,T2:tree  fr:s->nat  c:s  cL:list(s)  TL:list(tree)
  axioms      minOf2(fr,T,T') = if sum(map(fr)(root(T))) <= sum(map(fr)(root(T')))
                                then T else T'
              minOfList(fr,T:TL) = if TL = [] then T else minOf2(fr,T,minOfList(fr,TL))
              firstList([]) = []
              firstList(c:cL) = [c]:firstList(cL)
              newList(fr,T:TL) = (T1#(root(T1)++root(T2))#T2):remove(T2,TL')
                               <==  minOfList(fr,T:TL) = T1 /\ remove(T1,T:TL) = TL' /\
                                    minOfList(fr,TL') = T2
              Huffman(fr,cL) = Huffman1(fr,firstList(cL))
              Huffman1(fr,T:TL) = if TL = [] then T else Huffman1(fr,newList(fr,T:TL))
```

$\varphi(fr,cL,T) \quad \Longleftrightarrow_{def}$ T ist ein Codebaum so, dass für alle Zeichen $c,c' \in cL$ und Wege w,w' von der Wurzel von T zu dem Blatt, das mit c bzw. c' markiert ist, gilt: $fr(c) < fr(c') \;\Rightarrow\; length(w) \geq length(w')$.

$\theta(fr,cL,TL) \Longleftrightarrow_{def}$ TL ist eine Codebaumliste derart, dass für alle Zeichen $c,c' \in cL$ und Wege w,w' von der Wurzel der Bäume T,T' zu dem Blatt, das mit c bzw. c' markiert ist, gilt:
$fr(c) < fr(c') \Rightarrow\; length(w) \geq length(w')$,
$fr(c) < fr(c') \wedge length(w) = length(w') \wedge T \neq T'$
$\Rightarrow\; \sum_{c'' \in root(T)} fr(c'') \;<\; \sum_{c'' \in root(T')} fr(c'')$.

8 Deduktive Programmsynthese

Die Grundidee deduktiver Programmsynthese ist einfach: Man stellt eine Ein/Ausgabe-Relation $\varphi(x,y)$ auf, die eine noch unspezifizierte Funktion f erfüllen soll:

$$f(x) \equiv y \quad \Rightarrow \quad \varphi(x,y). \tag{1}$$

Dann wird (1) expandiert, i.w. durch Entfaltungen von Termen und Atomen von φ. Die Anwendung von (1) als Induktionshypothese liefert "rekursive Aufrufe" von f, die oft deutliche Hinweise darauf geben (*eureka!*), mit

welchen Axiomen für f die Expansion erfolgreich zuendegeführt werden kann. Deduktive Programmsynthese ist demnach in gewisser Weise komplementär zur Verifikation. Es wird immer wieder versucht, die Ableitung von Axiomen (= Programmen) für eine Funktion, die eine gewünschte Ein/Ausgabe-Relation realisiert, zu automatisieren. Sofern die erzeugten Programme nicht rekursiv, also mehr oder weniger trivial sind, geht das auch recht gut. Ein allgemeines Verfahren zur automatischen Ableitung rekursiver Programme wäre aber komplementär zu einem vollständigen Beweiskalkül für induktiver Theoreme. Da diesem die Nichtaufzählbarkeit induktiver Theoreme widerspricht, gibt es auch kein *vollständiges* Programmsyntheseverfahren (siehe auch die Bemerkungen am Ende von §6.6).

Hat man passende Axiome für f, die (1) erfüllen, in der o.g. Weise abgeleitet, dann lohnt es sich, die in der Ableitung verwendeten (Axiome und) Lemmas über Funktionen und Prädikate von φ genauer anzusehen. Da die Ableitung bzgl. aller Spezifikationen korrekt ist, die diese Lemmas und die Axiome für f erfüllen, gilt auch (1) in jeder solchen Spezifikation.

8.1 Funktionskomposition

Im ersten Beispiel geht es um die Ableitung von Axiomen für die Komposition zweier gegebener Funktionen (vgl. [91], Example 1).

```
SUMFILTER = LIST{nat/s}[NAT] then
    defuncts  sum:list(nat)->nat
              filter:(nat->bool)*list(nat)->list(nat)
              sumfilter:(nat->bool)*list(nat)->nat
    vars      n:nat  L,L':list  f:nat->bool
    axioms    sum([]) = 0
              sum(n:L) = n + sum(L)
              filter(f,[]) = []
              filter(f,n:L) = n:filter(f,L)  <==  f(n) = true
              filter(f,n:L) = filter(f,L)  <==  f(n) = false
```

Wir expandieren die Bedingung, dass $sumfilter$ die Komposition von $filter$ und sum ist, und leiten dabei Axiome für $sumfilter$ ab:[54]

$$sumfilter(f, L) \equiv sum(filter(f, L)) \tag{1}$$

Entfaltungen von $filter$

$$\begin{aligned}\vdash\quad & L \equiv [] \wedge sumfilter(f, L) \equiv sum([]),\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv sum(n : filter(f, L')) \wedge f(n) \equiv true),\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv sum(filter(f, L')) \wedge f(n) \equiv false)\end{aligned}$$

Entfaltungen von sum

$$\begin{aligned}\vdash\quad & L \equiv [] \wedge sumfilter(f, L) \equiv 0,\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv n + sum(filter(f, L')) \wedge f(n) \equiv true),\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv sum(filter(f, L')) \wedge f(n) \equiv false)\end{aligned}$$

Anwendungen einer Induktionshypothese

$$\begin{aligned}\vdash\quad & L \equiv [] \wedge sumfilter(f, L) \equiv 0,\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv n + sumfilter(f, L') \wedge f(n) \equiv true \wedge L \gg L'),\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv sumfilter(f, L') \wedge f(n) \equiv false \wedge L \gg L')\end{aligned}$$

Anwendung des Lemmas $n : L \gg L$

$$\begin{aligned}\vdash\quad & L \equiv [] \wedge sumfilter(f, L) \equiv 0,\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv n + sumfilter(f, L') \wedge f(n) \equiv true),\\ & \exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L) \equiv sumfilter(f, L') \wedge f(n) \equiv false)\end{aligned}$$

[54] Hier steht das Komma mal wieder für Disjunktionen.

Die letzte Goalmenge liefert Axiome für $sumfilter$ (*eureka!*):

```
sumfilter(f,[]) = 0
sumfilter(f,n:L) = n + sumfilter(f,L)  <==  f(n) = true
sumfilter(f,n:L) = sumfilter(f,L)  <==  f(n) = false
```

Mit ihnen läßt sich die Expansion von (1) zuendeführen:

Entfaltung von $sumfilter$

$\vdash \quad L \equiv [] \wedge 0 \equiv 0,$
$\exists n, L' : (L \equiv n : L' \wedge n + sumfilter(f, L') \equiv n + sumfilter(f, L') \wedge f(n) \equiv true),$
$\exists n, L' : (L \equiv n : L' \wedge sumfilter(f, L') \equiv sumfilter(f, L') \wedge f(n) \equiv false)$

Anwendung des Lemmas $n \equiv n$

$\vdash \quad L \equiv [] \ \vee \ \exists n, L' : (L \equiv n : L' \wedge f(n) \equiv true) \ \vee \ \exists n, L' : (L \equiv n : L' \wedge f(n) \equiv false)$

Anwendung des Lemmas $b \equiv true \vee b \equiv false$

$\vdash \quad L \equiv [] \ \vee \ \exists n, L' : L \equiv n : L'$ (2)

Anwendung des Lemmas (2)

$\vdash \quad True$

Faßt man (1) als Axiom für $sumfilter$ auf, dann ist die oben durchgeführte Expansion eine umgekehrte Entfaltung. Deshalb werden die Induktionsschritte in Programmableitungen auch **Faltungen** (**fold-Schritte**) genannt.

Wie in diesem Beispiel gibt es oft schon andere Axiome für die zu spezifierende Funktion. Dann geht es darum, die Zahl der verwendeten Hilfsfunktionen zu verringern und damit zu effizienteren Implementierungen. So wird die Länge der Reduktion eines Grundterms $sumfilter(f, L)$ in eine Normalform halbiert, wenn man die abgeleiteten Axiome anstelle von (1) benutzt.

Die Expansion von (1) läßt sich leicht verallgemeinern zur Ableitung von Axiomen für die Komposition $g \circ h$ zweier Funktionen g und h, die nach demselben Schema wie sum bzw. $filter$ spezifiziert sind. Wie lauten passende Axiomenschemata für g und h und welche Eigenschaften müssen die dort verwendeten Hilfsfunktionen haben, damit die Expansion von $(g \circ h)(f, L) \equiv g(h(f, L))$ ein entsprechendes Axiomenschema für $g \circ h$ liefert?

In [110] wird ein Algorithmus vorgestellt, der Programme für Kompositionen von n Funktionen erzeugt. Die Entfernung expliziter Funktionskompositionen kann man auch als eine Art *flattening* betrachten (s. §5.1). [110] spricht von *Entwaldung* (*deforestation*).

8.2 Tuplung

Hier geht es um die Linearisierung nicht-linear-rekursiver Funktionsdefinitionen durch Zusammenfassung mehrerer rekursiver Aufrufe zu Tupeln mithilfe einer **Tuplungsfunktion**. Betrachten wir dazu die übliche Spezifikation der Fibonacci-Zahlen:

```
FIB = NAT then
    defuncts  fib:nat->nat
              fib2:nat->(nat*nat)
    vars      n:nat
    axioms    fib(0) = 0
              fib(1) = 1
              fib(n+2) = fib(n) + fib(n+1)
```

Die Tuplungsfunktion $fib2$ soll beide rekursiven Aufrufe parallel berechnen:

$$fib2(n) \equiv (fib(n), fib(n+1)). \quad (1)$$

Wir expandieren (1) und leiten dabei Axiome für $fib2$ ab:[55]

$$fib2(n) \equiv (fib(n), fib(n+1))$$

Entfaltungen von fib

$$\vdash \quad n+1 \equiv 0 \wedge fib2(n) \equiv (fib(n), 0),$$
$$n+1 \equiv 1 \wedge fib2(n) \equiv (fib(n), 1),$$
$$\exists k : (n+1 \equiv k+2 \wedge fib2(n) \equiv (fib(n), fib(k) + fib(k+1)))$$

Clash und Term-Splitting

$$\vdash \quad n \equiv 0 \wedge fib2(n) \equiv (fib(n), 1),$$
$$\exists k : (n \equiv k+1 \wedge fib2(n) \equiv (fib(n), fib(k) + fib(k+1)))$$

Termersetzung

$$\vdash \quad n \equiv 0 \wedge fib2(0) \equiv (fib(0), 1),$$
$$\exists k : (n \equiv k+1 \wedge fib2(k+1) \equiv (fib(k+1), fib(k) + fib(k+1)))$$

Entfaltung von fib und Variablen-Einführung

$$\vdash \quad n \equiv 0 \wedge fib2(0) \equiv (0, 1),$$
$$\exists k, x, y : (n \equiv k+1 \wedge fib(k) \equiv x \wedge fib(k+1) \equiv y \wedge fib2(k+1) \equiv (y, x+y)$$

Term-Splitting

$$\vdash \quad n \equiv 0 \wedge fib2(0) \equiv (0, 1),$$
$$\exists k, x, y : (n \equiv k+1 \wedge (fib(k), fib(k+1)) \equiv (x, y) \wedge fib2(k+1) \equiv (y, x+y)$$

Anwendung einer Induktionshypothese

$$\vdash \quad n \equiv 0 \wedge fib2(0) \equiv (0, 1),$$
$$\exists k, x, y : (n \equiv k+1 \wedge fib2(k) \equiv (x, y) \wedge fib2(k+1) \equiv (y, x+y) \wedge n \gg k)$$

Anwendung des Lemmas $k+1 \gg k$

$$\vdash \quad n \equiv 0 \wedge fib2(0) \equiv (0, 1),$$
$$\exists k, x, y : (n \equiv k+1 \wedge fib2(k) \equiv (x, y) \wedge fib2(k+1) \equiv (y, x+y))$$

Die letzte Goalmenge liefert Axiome für $fib2$ (*eureka!*):

```
fib2(0) = (0,1)
fib2(n+1) = (y,x+y)  <==  fib2(n) = (x,y)
```

Mit ihnen läßt sich die Expansion von (1) zuendeführen:

Entfaltung von $fib2$

$$\vdash \quad n \equiv 0 \wedge (0, 1) \equiv (0, 1),$$
$$\exists k, x, y : (n \equiv k+1 \wedge fib2(k) \equiv (x, y))$$

Anwendung der Lemmas $z \equiv z$ und $\exists x, y : fib2(k) \equiv (x, y)$

$$\vdash \quad n \equiv 0 \ \vee \ \exists k : n \equiv k+1 \tag{2}$$

Anwendung des Lemmas (2)

$$\vdash \quad \mathit{True}$$

fib hat exponentielle, $fib2$ nur lineare Komplexität. fib ist baumartig-rekursiv, $fib2$ linear-rekursiv. $fib2$ vermeidet die Verdopplung der rekursiven Aufrufe von fib, die man erkennt, wenn man die Reduktion eines fib-Terms mit der Narrowing-Ableitung eines entsprechenden $fib2$-Constraints vergleicht (s. Def. 5.1.21):

$$fib(3) \ \longrightarrow \ fib(1) + fib(2) \ \longrightarrow \ \boxed{fib(1)} + fib(0) + \boxed{fib(1)} \ \xrightarrow{*} \ 1 + 0 + 1 \ \xrightarrow{*} \ 2.$$

[55] Wieder steht das Komma für Disjunktionen.

$$
\begin{aligned}
\langle fib2(3) \equiv z,\ id\rangle \ &\vdash\ \langle (y, x+y) \equiv z \wedge fib2(2) \equiv (x,y),\ id\rangle \\
&\vdash\ \langle (y, x+y) \equiv z \wedge (y', x'+y') \equiv (x,y) \wedge fib2(1) \equiv (x',y'),\ id\rangle \\
&\vdash\ \langle (x'+y', y'+x'+y') \equiv z \wedge \boxed{fib2(1)} \equiv (x',y'),\ id\rangle \\
&\vdash\ \langle (x'+y', y'+x'+y') \equiv z \wedge (y'', x''+y'') \equiv (x',y') \wedge fib2(0) \equiv (x'',y''),\ id\rangle \\
&\vdash\ \langle (x'+y', y'+x'+y') \equiv z \wedge (y'', x''+y'') \equiv (x',y') \wedge (0,1) \equiv (x'',y''),\ id\rangle \\
&\vdash\ \langle (x'+y', y'+x'+y') \equiv z \wedge (1, 0+1) \equiv (x',y'),\ 0/x'', 1/y''\rangle \\
&\vdash\ \langle (x'+y', y'+x'+y') \equiv z \wedge (1,1) \equiv (x',y'),\ 0/x'', 1/y''\rangle \\
&\vdash\ \langle (1+1, 1+1+1) \equiv z,\ 0/x'', 1/y'', 1/x', 1/y'\rangle \\
&\vdash\ \langle (2,3) \equiv z,\ 0/x'', 1/y'', 1/x', 1/y'\rangle \\
&\vdash\ \langle \emptyset,\ 0/x'', 1/y'', 1/x', 1/y', (2,3)/z\rangle.
\end{aligned}
$$

☞ Leiten Sie in ähnlicher Weise Axiome für eine Tuplungsfunktion *height&bal* ab, die bzgl. der Spezifikation 2-3-TREE aus Beispiel 7.4.2 oder AVL-TREE aus Beispiel 7.4.3 die Bedingung

$$height\&bal(T) \equiv (h(T), balanced(T))$$

erfüllt, wobei *balanced* die dem gleichnamigen Prädikat entsprechende Boolesche Funktion ist!

☞ Leiten Sie in ähnlicher Weise Axiome für eine Tuplungsfunktion *minmaxinf* ab, die bzgl. der Spezifikation TREE-ORDERS aus Beispiel 7.4.4 die Bedingung

$$minmaxinf(T) \equiv (minT(T), maxT(T), infix(T))$$

erfüllt, wobei *infix* die dem gleichnamigen Prädikat entsprechende Boolesche Funktion ist!

Beispiel 8.2.1 Wenden Sie die Methode der Tuplung auf die Funktion *knap* aus Beispiel 7.2.8 an, indem Sie eine Funktion *knap2(k,a)* ableiten, die die Gleichung

$$knap2(k,a) = (knap(k\text{-}1,a), knap(k\text{-}1,a\text{-}ak))$$

erfüllt!

Beispiel 8.2.2 Der bekannte Türme-von-Hanoi-Algorithmus (siehe [77], §9.2) bietet sich ebenfalls zur Tuplung an, weil auch hier gleiche rekursive Aufrufe mehrfach erzeigt werden:

```
HANOI = NAT then
     sorts          tower
     constructs     a,b,c:tower
     defuncts       trans:nat*tower*tower*tower->list(tower*tower)
     vars           n:nat  x,y,z:tower
     axioms         trans(0,x,y,z) = []
                    trans(n+1,x,y,z) = trans(n,x,z,y)++(x,z):trans(n,y,x,z)
```

$$
\begin{aligned}
& trans(2,x,y,z) \\
\longrightarrow\ & trans(1,x,z,y) + +(x,z) : trans(1,y,x,z) \\
\longrightarrow\ & \boxed{trans(0,x,y,z)} + +(x,y) : trans(0,z,x,y) + +(x,z) : trans(0,y,z,x) + +(y,z) : \boxed{trans(0,x,y,z)} \\
\xrightarrow{*}\ & (x,y) : (x,z) : (y,z) : [].
\end{aligned}
$$

☞ Leiten Sie Axiome für eine Tuplungsfunktion $trans3$ ab, die bzgl. HANOI die Bedingung

$$trans3(n,x,y,z) \equiv (trans(n,x,y,z), trans(n,z,x,y), trans(n,y,z,x))$$

erfüllt! ✌

8.3 Entrekursivierung

Aufwandsreduktion ist auch das Ziel der Überführung beliebiger Funktionsdefinitionen in iterative wie die in §7.5 behandelte. Die von einem iterativen Programm erzeugten Berechnungsfolgen entsprechen den Ableitungen von Wörtern einer **regulären Sprache**. Das Charakteristikum ist **Nicht-Selbsteinbettung** oder **tail recursion**: wenn $A \xrightarrow{+} vAw$, dann $w = \varepsilon$. Die Compilation iterativer Programme in maschinenlesbare Form ist weitaus einfacher als die Implementierung beliebiger Rekursion. Zwei Marken, zwei Sprungbefehle, und die Sache ist gelaufen... Echte Rekursion hingegen erfordert zusätzliche Datenstrukturen zur Zwischenspeicherung von Werten, Programmteilen, etc. Wie diese Datenstrukturen benutzt werden, was und wann etwas in sie eingetragen oder aus ihm herausgelesen wird, hängt von den Rekursionsschemata der übersetzten Programme ab. Im Prinzip läßt sich Entrekursivierung immer von der eigentlichen Übersetzung in eine Zielsprache trennen, indem man Entrekursivierung als Transformation (abstrakter) Quellprogramme *innerhalb* der Quellsprache realisiert.

8.3.1 Entrekursivierung mit Akkumulator. Gegeben sei eine linear-rekursive Axiomatisierung einer Funktion f:

$$\boxed{\begin{array}{l} f(x) \equiv h(x) + f(pred(x)) \;\Leftarrow\; p_1(x) \\ f(x) \equiv out(x) \;\Leftarrow\; p_2(x). \end{array}} \tag{1}$$

Bei diesem Rekursionsschema reicht unter den u.g. Voraussetzungen die Einführung eines Wert-**Akkumulators** y vom Typ der Werte von f, um zu einer iterativen Definition von f zu gelangen. Genauer gesagt geht es um iterative Axiome für eine Funktion f_1, die mit f in folgendem Zusammenhang steht:

$$\boxed{f(x) \equiv f_1(x, a),} \tag{2}$$

wobei a ein geeigneter "Anfangswert" von y ist. Da (1) nur einen Teil der Ein/Ausgabe-Relation von f_1 beschreibt, lassen sich aus dieser Gleichung noch keine Axiome für f_1 ableiten. Wir verallgemeinern deshalb (2) zu:

$$f_1(x, y) \equiv y + f(x). \tag{3}$$

Ist a linksneutral bzgl. $+$, dann folgt (2) aus (3). Wir versuchen, aus der Expansion von (3) Axiome für f_1 abzuleiten:

$$f_1(x, y) \equiv y + f(x)$$

Entfaltung von f

$$\vdash \quad \begin{array}{l} f_1(x, y) \equiv y + (h(x) + f(pred(x))) \wedge p_1(x), \\ f_1(x, y) \equiv y + out(x) \wedge p_2(x) \end{array}$$

Anwendung des Lemmas $x + (y + z) \equiv (x + y) + z$

$$\vdash \quad \begin{array}{l} f_1(x, y) \equiv (y + h(x)) + f(pred(x)) \wedge p_1(x), \\ f_1(x, y) \equiv y + out(x) \wedge p_2(x) \end{array}$$

Anwendung einer Induktionshypothese

$$\vdash \quad \begin{array}{l} f_1(x, y) \equiv f_1(pred(x), y + h(x)) \wedge p_1(x) \wedge x \gg pred(x), \\ f_1(x, y) \equiv y + out(x) \wedge p_2(x) \end{array}$$

Anwendung des Lemmas $x \gg pred(x) \Leftarrow p_1(x)$

$$\vdash \quad \begin{array}{l} f_1(x, y) \equiv f_1(pred(x), y + h(x)) \wedge p_1(x), \\ f_1(x, y) \equiv y + out(x) \wedge p_2(x) \end{array}$$

Die letzte Goalmenge liefert iterative Axiome für f_1 (*eureka!*):

$$\boxed{\begin{array}{l} f_1(x, y) \equiv f_1(pred(x), y + h(x)) \;\Leftarrow\; p_1(x) \\ f_1(x, y) \equiv y + out(x) \;\Leftarrow\; p_2(x) \end{array}} \tag{4}$$

Mit ihnen läßt sich die Expansion von (3) zuendeführen:

Entfaltung von f_1

$\vdash$ $f_1(pred(x), y + h(x)) \equiv f_1(pred(x), y + h(x)) \wedge p_1(x)$,
$y + out(x) \equiv y + out(x) \wedge p_2(x)$

Anwendung des Lemmas $x \equiv x$

$\vdash$ $p_1(x) \vee p_2(x)$

Anwendung des Lemmas $p_1(x) \vee p_2(x)$

$\vdash$ $True$

Wir fassen die Bedingungen an die Definition von f zusammen, die gelten müssen, damit (3) aus (4) folgt: $+$ ist assoziativ, a ist linksneutral bzgl. $+$ und $\ll$ ist eine wohlfundierte Relation ist derart, dass das zugrundeliegende Modell die Formeln $p_1(x) \vee p_2(x)$ und $p_1(x) \Rightarrow x \gg pred(x)$ erfüllt. ✌

☞ Leiten Sie in ähnlicher Weise die in Beispiel 7.5.2 verifizierte iterative Version der Funktion *div* ab (siehe Beispiel 5.1.5)! Hierbei ist $div(x, y) = loop(y, 0, x)$ die (2) entsprechende Initialisierung, während sich die (3) entsprechende, zu transformierende Formel aus der Hoare-Invarianten $x = (y * q) + r$ und der daraus gemäß §7.5 gebildeten Subgoal-Invarianten (also der Ein/Ausgaberelation der oben f_1 genannten Schleifenfunktion *loop*) ergibt:

$$x = (y * q) + r \quad \Longrightarrow \quad loop(y, q, r) = div(x, y).$$

8.3.2 Teilweise Entrekursivierung baumartiger Rekursion mit Akkumulatoren. Gegeben sei folgende baumartig-rekursive Axiomatisierung einer Funktion f:

$$\boxed{\begin{array}{l} f(x) \equiv h(x) + (f(pred_1(x)) * f(pred_2(x))) \;\Leftarrow\; p_1(x) \\ f(x) \equiv out(x) \;\Leftarrow\; p_2(x) \end{array}} \tag{1}$$

(7) ist ein Spezialfall von (6), bei dem g aus h, $+$ und $*$ zusammengesetzt ist. Umgekehrt verallgemeinert (8) das Schema (1) aus 8.3.1 von einem auf zwei — durch $*$ verknüpfte — rekursive Aufrufe von f. Dementsprechend folgt auch die Entwicklung einer (halb-)iterativen Version von f 8.3.1 analogen Überlegungen. Zunächst wird jeder Aufruf f zum Aufruf einer jetzt *drei*stelligen Funktion f_1 mit Anfangswerten a bzw. b *zweier* Akkumulatoren y und z:

$$\boxed{f(x) \equiv f_1(x, a, b).} \tag{2}$$

Gesucht sind Axiome für f_1. Da (9) wieder nur einen Teil der Ein/Ausgabe-Relation von f_1 beschreibt, lassen sich aus dieser Gleichung noch keine Axiome für f_1 ableiten. Wir verallgemeinern deshalb (9) zu:

$$f_1(x, y, z) \equiv (y + f(x)) * z. \tag{3}$$

Ist a linksneutral bzgl. $+$ und b rechtsneutral bzgl. $*$, dann folgt (2) aus (3). Wir versuchen, aus der Expansion von (3) Axiome für f_1 abzuleiten:

$f_1(x, y, z) \equiv (y + f(x)) * z$

Entfaltung

$\vdash$ $f_1(x, y, z) \equiv (y + (h(x) + (f(pred_1(x)) * f(pred_2(x))))) * z \wedge p_1(x)$,
$f_1(x, y, z) \equiv (y + out(x)) * z \wedge p_2(x)$

Anwendung des Lemmas $x + (y + z) \equiv (x + y) + z$

$\vdash$ $f_1(x, y, z) \equiv ((y + h(x)) + (f(pred_1(x)) * f(pred_2(x)))) * z \wedge p_1(x)$,
$f_1(x, y, z) \equiv (y + out(x)) * z \wedge p_2(x)$

Anwendung des Lemmas $x + (y * z) \equiv (x + y) * (x + z)$

$\vdash$ $f_1(x, y, z) \equiv (((y + h(x)) + f(pred_1(x))) * ((y + h(x)) + f(pred_2(x)))) * z \wedge p_1(x)$,
$f_1(x, y, z) \equiv (y + out(x)) * z \wedge p_2(x)$

Anwendung einer Induktionshypothese

$\vdash \quad f_1(x,y,z) \equiv f_1(pred_1(x), y + h(x), ((y + h(x)) + f(pred_2(x))) * z) \wedge p_1(x) \wedge x \gg pred_1(x),$
$f_1(x,y,z) \equiv (y + out(x)) * z \wedge p_2(x)$

Anwendung des Lemmas $x \gg pred_1(x) \Leftarrow p_1(x)$

$\vdash \quad f_1(x,y,z) \equiv f_1(pred_1(x), y + h(x), ((y + h(x)) + f(pred_2(x))) * z) \wedge p_1(x),$
$f_1(x,y,z) \equiv (y + out(x)) * z \wedge p_2(x)$

Anwendung einer Induktionshypothese

$\vdash \quad f_1(x,y,z) \equiv f_1(pred_1(x), y + h(x), f_1(pred_2(x), y + h(x), z)) \wedge p_1(x) \wedge x \gg pred_2(x)),$
$f_1(x,y,z) \equiv (y + out(x)) * z \wedge p_2(x)$

Anwendung des Lemmas $x \gg pred_2(x) \Leftarrow p_1(x)$

$\vdash \quad f_1(x,y,z) \equiv f_1(pred_1(x), y + h(x), f_1(pred_2(x), y + h(x), z)) \wedge p_1(x)),$
$f_1(x,y,z) \equiv (y + out(x)) * z \wedge p_2(x)$

Die letzte Goalmenge liefert Axiome für f_1 mit zumindest einem nicht-eingebetteten Aufruf von f_1:

$$\boxed{\begin{array}{l} f_1(x,y,z) \equiv f_1(pred_1(x), y + h(x), f_1(pred_2(x), y', z)) \;\Leftarrow\; p_1(x) \\ f_1(x,y,z) \equiv (y + out(x)) * z \;\Leftarrow\; p_2(x) \end{array}} \tag{4}$$

Mit ihnen läßt sich die Expansion von (10) zuendeführen:

Entfaltung

$\vdash \quad f_1(pred_1(x), y + h(x), f_1(pred_2(x), y', z)) \equiv f_1(pred_1(x), y + h(x), f_1(pred_2(x), y', z)) \wedge p_1(x),$
$(y + out(x)) * z \equiv (y + out(x)) * z \wedge p_2(x)$

Anwendung des Lemmas $x \equiv x$

$\vdash \quad p_1(x) \vee p_2(x)$

Anwendung des Lemmas $p_1(x) \vee p_2(x)$

$\vdash \quad True$

Wir fassen die Bedingungen an die Definition von f zusammen, die gelten müssen, damit (3) aus (4) folgt: $+$ ist assoziativ und distributiert über $*$, a ist linksneutral bzgl. $+$, b ist rechtsneutral bzgl. $*$ und $\ll$ ist eine wohlfundierte Relation ist derart, dass das zugrundeliegende Modell die Formeln $p_1(x) \vee p_2(x)$ und

$$p_1(x) \quad \Rightarrow \quad x \gg pred_1(x) \wedge x \gg pred_2(x)$$

erfüllt. ✌

Beispiel 8.3.3 Gegeben sei die folgende Spezifikation binärer Bäume:

```
BINTREE = LIST[TRIV(s)] and NAT then
     sorts       tree
     constructs  leaf:s->tree
                 _#_:tree*tree->tree
     defuncts    height:tree->nat
     vars        x:s  T,T':tree
     axioms      height(leaf(x)) = 1
                 height(T#T') = max(height(T),height(T'))+1
```

☞ Überführen Sie *height* unter Verwendung von zwei Akkumulatoren in halb-iterative Form! ✌

Beispiel 8.3.4 (Entrekursivierung geschachtelter Rekursion) In Beispiel 7.5.3 haben wir die iterative Version eines Algorithmus *search* zur Tiefensuche in Graphen angegeben. Hier ist eine Version mit geschachteler Rekursion (*nested recursion*):

```
SEARCH = LIST[TRIV(s)] then
       sorts     graph = s->list(s)
       defuncts  search:graph*list(s)->list(s)
                 search2:graph*list(s)*list(s)->list(s)
       vars      G:graph  x,y:s  L,V:list(s)
       axioms    search(G,L) = search2(G,L,[])
                 search2(G,[],V) = V
                 search2(G,x:L,V) = search2(G,L,V)  <==  x ∈ V
                 search2(G,x:L,V) = search2(G,L,search2(G,G(x),x:V))  <==  x ∉ V
```

Die Axiome für $search_2$ entspricht denjenigen für $search_1$ (s. 7.5.3) bis auf die rechte Seite des dritten Axioms. Würde noch die Gleichung

$$search_2(G, L, search_2(G, G(x), x : V)) \equiv search_1(G, G(x) + +L, x : V) \quad \Leftarrow \quad x \notin V \tag{1}$$

gelten, dann wären $search_1$ und $search_2$ äquivalent. (1) ist für eine induktiven Beweis wieder mal nicht allgemein genug. M.a.W.: Der Beweis von (1) benötigt eine stärkere Induktionshypothese. Insbesondere das mehrfache Auftreten der Variablen G und x macht (1) zu speziell. (1) ist jedoch eine Instanz der induktiv beweisbaren Formel

$$search_2(G, L, search_2(G, L', V)) \equiv search_1(G, L' + +L, V) \tag{2}$$

und folgt deshalb aus ihr.

☞ Zeigen Sie (1) durch Noethersche Induktion, indem Sie zuerst die rechte Seite von (1) entfalten! ✌

8.3.5 Entrekursivierung mit Keller. Erfüllt die rekursive Definition einer Funktion $f : s \to s'$ keine speziellen Bedingungen wie in 8.3.1 und 8.3.2, dann müssen die Argumente und Werte ihrer rekursiven Aufrufe in einem **Keller** zwischengespeichert werden, um sie in eine iterative Form zu bringen. Das zugrundeliegende Rekursionsschema laute wie folgt:

$$\boxed{\begin{array}{l} f(x) \equiv g(x, f(pred_1(x)), \ldots, f(pred_n(x))) \;\Leftarrow\; p_1(x) \\ f(x) \equiv out(x) \;\Leftarrow\; p_2(x) \end{array}}$$

Aus den hier auftretenden Termen ergeben sich folgende Typen der verwendeten Hilfsfunktionen:

$$\begin{array}{l} g : s \times s' \times \ldots s' \to s' \\ pred_1, \ldots, pred_n : s \to s \\ out : s \to s' \end{array}$$

Hier ist ein iteratives Programm f', das f berechnet:

```
RECSTACK = LIST[TRIV(s+s')] and specification of g,pred_1,...,pred_n,out then
    defuncts  f':s->s'
              loop:list(s+s')->s'
    vars      x:s  y,y1,...,yn:s'  L:list(s+s')
    axioms    f'(x) = loop([x])
              loop(x:L) = loop(pred_n(x):...:pred_1(x):x:L)  <==  p1(x)
              loop(x:L) = loop(out(x):L)                     <==  p2(x)
              loop(yk:...:yn:x:L) = loop(x:yk:...:yn:L)           für alle k < n
              loop(y1:...:yn:x:L) = loop(g(x,y1,...,yn):L)
              loop([y]) = y
```

☞ Zeigen Sie die Äquivalenz beider Spezifikationen von f durch Hoare-Induktion! Aus den Axiomen von RECSTACK ergeben sich folgende Bedingungen an eine Hoare-Invariante θ von f:

$$\begin{aligned}
&\theta(x,[x])\\
&\theta(x,x':L)\wedge p_1(x') \quad\Rightarrow\quad \theta(x,pred_n(x'):...:pred_1(x'):x':L)\\
&\theta(x,x':L)\wedge p_2(x) \quad\Rightarrow\quad \theta(x,out(x'):L)\\
&\theta(x,y_k:...:y_n:x':L) \quad\Rightarrow\quad \theta(x,x':y_k:...:y_n:L)\\
&\theta(x,y_1:...:y_n:x':L) \quad\Rightarrow\quad \theta(x,g(x',y_1:...:y_n):L)\\
&\theta(x,[y]) \quad\Rightarrow\quad f(x)\equiv y.
\end{aligned}$$

Es ist also ein θ zu finden, dass diese Bedingungen erfüllt.

Der Keller enthält also Eingaben und Ausgaben, Argumente und (Zwischen-)Werte der gegebenen Funktion f. Unter gewissen Bedingungen kann auf die Speicherung von Werten verzichtet werden. Das ist zum Beispiel dann der Fall, wenn $n=2$ gilt, f also baumartig-rekursiv ist, g nicht von x abhängt, also nur noch zweistellig (!) ist, und es ein bzgl. g rechtsneutrales Element a gibt. RECSTACK reduziert sich dann zu:

```
RECSTACK' = LIST[TRIV(s)] and specification of g,pred_1,pred_2,out then
    defuncts  f':s->s'
              loop:list(s)*s' ->s'
    vars      x:s  y,y1,y2:s'  L:list(s)
    axioms    f'(x) = loop([x],a)
              loop(x:L,y) = loop(pred_2(x):pred_1(x):L,y)  <==  p1(x)
              loop(x:L,y) = loop(L,g(out(x),y))            <==  p2(x)
              loop([],y) = y
```

Beispiel 8.3.6 Wir erweitern die Spezifikation BINTREE aus Beispiel 8.3.3:

```
FLATTEN = BINTREE then
    defuncts    flatten:tree*s->list(s)
    vars        x:s  T,T':tree
    axioms      flatten(leaf(x)) = [x]
                flatten(T#T') = flatten(T)++flatten(T')
```

☞ Bilden Sie eine iterative Version von *flatten* gemäß RECSTACK'! 🐰

8.4 Parallelisierung durch λ-Abstraktion

In Beispiel 6.7.3 wurden zwei durch jeweils eine Gleichung der Form

$$\boxed{f(x)\equiv g(x,h(x))} \tag{1}$$

definierte Funktionen optimiert, indem die Abstraktion $\lambda y.g(x,y)$ parallel zu $h(x)$ berechnet wurde:

$$g\&h(x)\equiv(\lambda y.g(x,y),h(x)). \tag{2}$$

Die Anwendung der ersten auf die zweite Komponente von $g\&h(x)$ liefert eine äquivalente Definition von f:

$$\boxed{f(x)\equiv f'(y) \quad\Leftarrow\quad g\&h(x)\equiv(f',y).} \tag{3}$$

Die Transformation von (1) nach (3) ist dann ratsam, wenn f auf komplexen Datenstrukturen wie Listen oder Bäumen operiert und die doppelte Traversierung von x mit hohem Aufwand verbunden ist. Kostet die Anwendung der ersten auf die zweite Komponente von $g\&h(x)$ vernachlässigbar wenig, dann ist f nur noch halb so teuer. Zur Ableitung von Axiomen für $g\&h$ benötigen wir eine Regel zur Entfaltung von λ-Abstraktionen:

Abstraktionsentfaltung $\dfrac{\lambda x.f(t,x)}{(\lambda p_1.u_1\sigma_1)\blacksquare\ldots\blacksquare(\lambda p_k.u_k\sigma_k)}\ \Updownarrow$

falls $x \notin var(t)$ und $f(t_1,p_1) \equiv u_1, \ldots, f(t_k,p_k) \equiv u_k$ die Axiome für f mit $t\sigma_i = t_i\sigma_i$ sind[56]

Da wir alle λ-Abstraktionen als Konstruktoren auffassen, ist diese Regel i.a. nur bzgl. des extensionalen initialen Modells der zugrundeliegenden Spezifikation korrekt (vgl. §6.6).

In Beispiel 6.7.3 lautet (2) wie folgt:

$$eq\&rev(L) \ \equiv\ (\lambda L'.eq(L,L'), rev(L)) \tag{4}$$

$$\text{bzw.}\quad rep\&min(T) \ \equiv\ (\lambda n.rep(T,n), min(T)). \tag{5}$$

Wir versuchen, aus der Expansion von (4) Axiome für $eq\&rev$ abzuleiten:

$eq\&rev(L) \equiv (\lambda L'.eq(L,L'), rev(L))$

Entfaltung

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda L'.eq(L,L'), []),$
$\exists x, L_1 : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda L'.eq(L,L'), rev(L_1) + +[x]))$

Termersetzung

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda L'.eq([],L'), []),$
$\exists x, L_1 : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda L'.eq(x : L_1, L'), rev(L_1) + +[x]))$

Abstraktionsentfaltung

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda[].true\blacksquare\lambda x : L'.false,\ []),$
$\exists x, L_1 : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda[].false\blacksquare\lambda y : L'.(eq(x,y)\ and\ eq(L_1,L')),\ rev(L_1) + +[x]))$

Anwendung des Lemmas $t \equiv (\lambda L.t)(L)$

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda[].true\blacksquare\lambda x : L'.false,\ []),$
$\exists x, L_1 : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda[].false\blacksquare\lambda y : L'.(eq(x,y)\ and\ (\lambda L'.eq(L_1,L'))(L')),\ rev(L_1) + +[x]))$

Variablen-Einführung und Termersetzung

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda[].true\blacksquare\lambda x : L'.false,\ []),$
$\exists x, L_1, L_2, g : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda[].false\blacksquare\lambda y : L'.(eq(x,y)\ and\ g(L')),\ L_2 + +[x])$
$\wedge\ g \equiv \lambda L'.eq(L_1,L') \wedge L_2 \equiv rev(L_1))$

inverses Term-Splitting

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda[].true\blacksquare\lambda x : L'.false,\ []),$
$\exists x, L_1, L_2, g : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda[].false\blacksquare\lambda y : L'.(eq(x,y)\ and\ g(L')),\ L_2 + +[x])$
$\wedge\ (g, L_2) \equiv (\lambda L'.eq(L_1,L'), rev(L_1)))$

Anwendung der Induktionshypothese $(\lambda L'.eq(L_1,L'), rev(L_1)) \equiv eq\&rev(L_1) \Leftarrow L \gg L_1$

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda[].true\blacksquare\lambda x : L'.false,\ []),$
$\exists x, L_1, L_2, g : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda[].false\blacksquare\lambda y : L'.(eq(x,y)\ and\ g(L')),\ L_2 + +[x])$
$\wedge\ (g, L_2) \equiv eq\&rev(L_1) \wedge L \gg L_1)$

Anwendung des Lemmas $x : L_1 \gg L$

$\vdash\ L \equiv [] \wedge eq\&rev(L) \equiv (\lambda[].true\blacksquare\lambda x : L'.false,\ []),$
$\exists x, L_1, L_2, g : (L \equiv x : L_1 \wedge eq\&rev(L) \equiv (\lambda[].false\blacksquare\lambda y : L'.(eq(x,y)\ and\ g(L')),\ L_2 + +[x])$
$\wedge\ (g, L_2) \equiv eq\&rev(L_1))$

Die letzte Goalmenge liefert die REPALA-Axiome für $eq\&rev$ (s. Bsp. 6.7.3). Mit ihnen läßt sich die Expansion von (4) zuendeführen:

[56] Die Korrektheit der Regel ist nur dann gesichert, wenn alle Axiome für f unbedingte Gleichungen sind.

Entfaltung
$\vdash \quad L \equiv [] \quad \vee \quad \exists x, L_1, L_2, g : (L \equiv x : L_1 \wedge (g, L_2) \equiv eq\&rev(L_1))$
Anwendung des Lemmas $\exists f, L' : (f, L') \equiv eq\&rev(L)$
$\vdash \quad L \equiv [] \quad \vee \quad \exists n, L' : L \equiv n : L'$ (6)
Anwendung des Lemmas (6)
$\vdash \quad True$

☞ Leiten Sie in analoger Weise aus einem Beweis von (5) die REPALA-Axiome für $rep\&min$ her!

☞ Verallgemeinern Sie die Herleitung der Axiome von $eq\&rev$ und $rep\&min$ zu einer Herleitung von Axiomen für eine Funktion $g\&h$, die zunächst wie in (2) definiert ist, unter der Voraussetzung, dass alle Axiome für g unbedingte Gleichungen sind!

8.5 Differenzbildung

Gegeben sei eine linear-rekursive Axiomatisierung einer Funktion $f : nat \to s$ folgender Form:

$$\boxed{\begin{aligned} &f(0) \equiv e(0) \\ &f(n+1) \equiv g(e(n+1), f(n)) \end{aligned}} \tag{1}$$

Weiterhin gebe es eine **Differenzfunktion** $\delta : s \times nat \to s$, die folgende Beziehung zwischen den Werten von e herstellt:

$$\boxed{\delta(e(n+1), n) \equiv e(n)} \tag{2}$$

Ist g assoziativ und a bzgl. g linksneutral, dann bilden die folgenden Axiome ein iteratives Programm für f:

$$\boxed{\begin{aligned} &f'(0) \equiv e(0) \\ &f'(n+1) \equiv h(n, a, e(n+1)) \\ &h(0, x, y) \equiv g(g(x, y), e(0)) \\ &h(n+1, x, y) \equiv h(n, g(x, y), \delta(y, n+1)). \end{aligned}} \tag{3}$$

Die Herleitung von (3) aus (1) und (2) findet man in [91], Example 7.

☞ Beweisen Sie die Gleichung

$$h(n, x, e(n+1)) \equiv g(g(x, e(n+1)), f(n)) \tag{4}$$

unter den o.g. Voraussetzungen über g! Aus (4) folgt die Äquivalenz von (1) und (3):

$$\begin{aligned} &f(0) \equiv e(0) \equiv f'(0), \\ &f(n+1) \equiv g(e(n+1), f(n)) \equiv g(g(a, e(n+1)), f(n)) \equiv h(n, a, e(n+1)) \equiv f'(n+1). \end{aligned}$$

Beispiel 8.5.1 Setzt man $e(n) = n^2$ und $g(x, y) = x + y$, dann ist die Quadratsumme $f(n) = \sum_{i=0}^{n} n^2$ nach Schema (1) definiert. Mit $a = 0$ und $\delta(y, n) = y - 2n - 1$ liefert Schema (2) eine iterative Definition von f mit nur einer Quadratberechnung pro Aufruf von f:

$$\begin{aligned} &f'(0) \equiv 0 \\ &f'(n+1) \equiv h(n, 0, (n+1)^2) \\ &h(0, x, y) \equiv x + y \\ &h(n+1, x, y) \equiv h(n, x + y, y - 2n - 3) \quad \text{🐰} \end{aligned}$$

8.6 Tabellierung

Um das mehrfache Auftreten gleicher rekursiver Aufrufe zu vermeiden oder allgemeiner: um baumartige Rekursion zu linearisieren, kann man anstelle der Erzeugung von Wertetupeln (s. §8.2) auch eine Tabelle der Werte rekursiver Aufrufe anlegen und wiederholte Aufrufe durch Tabellenzugriffe ersetzen. Eine solche Tabelle wird auch **Memofunktion** genannt. Sie wird in der Regel als Feld repräsentiert. Im Fall der Fibonacci-Funktion (s. §8.2) sieht sie folgendermaßen aus:

```
fibs = array' bounds [(i,f i) | i <- range bounds]
       where bounds = (0,max)
             f 0 = 0
             f 1 = 1
             f n = fibs!(n-1)+fibs!(n-2)
fib n = fibs!n
```

Die Verbindung von Rekursion und Tabellierung wird auch **dynamische Programmierung** genannt.

☞ Geben Sie eine äquivalente iterative Definition von $fibs$ an, in der die Liste in einem Akkumulator aufgebaut wird (s. 8.3.1)!

Auch wechselseitige Rekursion zwischen mehreren Funktionen läßt sich in Tabellen transformieren. Seien $f_1, \ldots, f_k$ wechselseitig-rekursiv definierte listen-erzeugende Funktionen:

$$\begin{array}{rcl} f_1(0) & \equiv & A_1 \\ f_1(n+1) & \equiv & B_{11} + + f_{i_{11}}(n) + + \cdots + + B_{1n_1} + + f_{i_{1n_1}}(n) + + C_1 \\ & & \ldots \\ f_k(0) & \equiv & A_k \\ f_k(n+1) & \equiv & B_{k1} + + f_{i_{k1}}(n) + + \cdots + + B_{kn_k} + + f_{i_{kn_k}}(n) + + C_k \end{array} \tag{1}$$

Hierbei gehöre jedes $f_{i_{jr}}$, $1 \leq j \leq k$, $1 \leq r \leq n_j$, zu $\{f_1, \ldots, f_k\}$. Zur Tabellierung aller Werte von $\{f_1, \ldots, f_k\}$ bietet sich eine zweidimensionale Matrix mat an:

```
mat = array' bounds [(i,f j r) | i@(j,r) <- range bounds]
      where bounds = ((1,0),(k,max))
            f 1 0 = A_1
            f 1 n = B_11++mat!(i_11,n-1)++B_1n_1++mat!(i_1n_1,n-1)++C_1
            ...
            f k 0 = A_k
            f k n = B_k1++mat!(i_k1,n-1)++B_kn_k++mat!(i_kn_k,n-1)++C_k
```

☞ Sei $f_1 = trans$, $f_2 = \lambda(x,y,z).trans(x,z,y)$ und $f_3 = \lambda(x,y,z).trans(y,x,z)$ (s. Bsp. 8.2.2). Schreiben Sie ein Haskell-Programm, das die entsprechende Instanz von mat berechnet!

Beispiel 8.6.1 ([91], Example 8)

```
HILBERT = NAT then
    sorts           direction
    constructs      east,west,north,south:->direction
    defuncts        hilbert:nat->list(direction)
                    f1,f2,f3,f4:nat->list(direction)
    vars            n:nat
    axioms          hilbert(n) = f1(n)
                    f1(0) = f2(0) = f3(0) = f4(0) = []
                    f1(n+1) = f4(n)++west:f1(n)++south:f1(n)++east:f2(n)
                    f2(n+1) = f3(n)++north:f2(n)++east:f2(n)++south:f1(n)
                    f3(n+1) = f2(n)++east:f3(n)++north:f3(n)++west:f4(n)
                    f4(n+1) = f1(n)++south:f4(n)++west:f4(n)++north:f3(n)
```

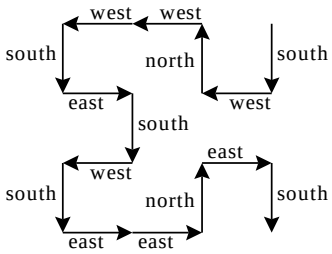


Figure 12. *hilbert(2)*

☞ Wie lautet die entsprechende Instanz von *mat*? ✌

9 Zustandsorientierte Modellierung

In den vorangegangen Kapiteln wurde gezeigt, wie mithilfe von algebraisch-logischer Konzepte Programme in abstrakter Syntax entworfen, verfeinert und verifiziert werden. Ein wesentlicher Vorteil dieser Konzepte ist die durch sie bestimmte gemeinsame Sprache für Programme/Algorithmen sowie Aussagen darüber. Dies wird besonders deutlich bei der Verifikation iterativer Programme mithilfe von Schleifeninvarianten (§7.5) und den in Kapitel 8 behandelten Syntheseverfahren. So ergaben sich die Beweisregeln der Hoare- und Subgoal-Induktion direkt aus entsprechenden Programmschemata, ohne dass — wie bei der klassischen Formulierung des Hoare-Kalküls (s. §10.2) — die verwendete Logiksprache um eine konkrete imperative Programmiersprache angereichert werden musste. Iteration ist ein funktional-logisches Konzept. Aussagen über iterative Programme lassen sich deshalb am einfachsten auf dieser Abstraktionsebene ausdrücken und beweisen. Die bekanntermaßen effiziente *Implementierung* iterativer Algorithmen mithilfe imperativer Konstrukte wie Zuweisungen und Schleifen ist für die Verifikation der Algorithmen in der Regel irrelevant. Die Frage, ob imperative Programmschemata wie 7.5(2) funktionale wie 7.5(1) korrekt implementieren, liegt auf einer anderen Ebene. Sie könnten wir im Rahmen der bisher entwickelten Theorie auch nicht beantworten. In diesem Kapitel wird der Begriff der konstruktorbasierten Spezifikation (Def. 5.1.2 u. 6.2.1) noch einmal erweitert, um u.a. imperative Konzepte zu präzisieren.

Machen wir uns einmal klar, welches Datenmodell bisher hinter konstruktorbasierten, insbesondere *funktionalen* Spezifikationen (Def. 5.1.8) gestanden hat. Funktionalität verlangt, dass alle Daten als Grundterme (Normalformen) dargestellt werden (Vollständigkeit) und dass diese Darstellung eindeutig ist (Konsistenz). Das war die Voraussetzung, um Funktionen und Prädikate mithilfe von Hornaxiomen auf den Daten rekursiv definieren und Anforderungen an die Spezifikation induktiv beweisen zu können. Trotz der Unvollständigkeit induktiver Beweisverfahren (s. §7.2) bleibt uns prinzipiell kein anderer Weg, denn wir wollen uns ja nicht nur mit endlichen Datenbereichen beschäftigen.

Bezogen sind Existenz und Eindeutigkeit der Normalformen auf die Interpretation der Gleichheiten im initialen Modell der zugrundeliegenden Spezifikation SP. Aber erst der Quotient von $Her(SP)$ nach dieser Interpretation, das initiale Modell $Ini(SP)$, interpretiert $\equiv$ als Gleichheit. Gerechnet haben wir dennoch im Herbrandmodell (siehe Kapitel 7 und 8), u.a. deshalb, weil die Gleichheit im initialen Modell nicht immer die gewünschte Datenidentität wiedergibt. So wurde in den Abschnitten 6.4 und 6.6 die Gleichheit von Daten als **starke** bzw. **extensionale Äquivalenz** definiert und in entsprechende Standardmodelle eingeführt. Wir haben dort bereits erwähnt, dass sich solche Äquivalenzrelationen mithilfe von co-Hornformeln axiomatisieren lassen, wenn man sie als größte Lösungen ihrer Axiome interpretiert. In der Terminologie von §6.3 handelt es sich demnach um Coprädikate, im Gegensatz zu Gleichheiten, deren Axiome Hornformeln sind.

Die Kongruenzaxiome machen die SP-Äquivalenz $\equiv_{SP}$ automatisch zur prädikatenverträglichen Kongruenzrelation. Diese Eigenschaft ist entscheidend für die Korrektheit wichtiger Beweisregeln wie Term-Splitting, Ent-

faltung, internem Rewriting sowie der Expansion mit bedingten Gleichungen (s. Kap. 8). Sollen diese Regeln auch für eine als Coprädikat spezifizierte Äquivalenzrelation $\sim$ anstelle von $\equiv$ benutzt werden, dann muss die Interpretation von $\sim$ im initialen Modell ebenfalls eine prädikatenverträgliche Kongruenz sein. Nun hat aber das Beispiel "starke Äquivalenz" schon gezeigt, dass manche praktisch sinnvollen Gleichheitsrelationen *nicht* mit allen Prädikaten von SP verträglich sind. In §6.4 haben wir die abgeschwächte Bedingung der **Zickzack-Verträglichkeit** eingeführt, die hinreichend ist, um Quotienten nach $\sim$ zu bilden und damit Modelle, die $\sim$ als Gleichheit interpretieren. Wie wir in §9.2 sehen werden, garantiert die Zickzack-Verträglichkeit von $\sim$ auch den Erhalt der Korrektheit o.g. Beweisregeln, wenn man $\equiv$ durch $\sim$ ersetzt, zumindest dann, wenn die Regeln auf **modale Formeln** angewendet werden.

Was ist nun die intuitive Vorstellung hinter einem Standardmodell $Her(SP)/\sim$, dessen "Datengleichheit" $\sim$ die SP-Äquivalenz zwar enthält, mit dieser aber nicht mehr übereinstimmt? Da die Existenz und Eindeutigkeit von Normalformen auf SP-Äquivalenz bezogen bleibt, gibt es i.a. verschiedene, aber $\sim$-äquivalente Normalformen, d.h. die Normalformrepräsentation eines Objektes ist nicht mehr eindeutig. Verschiedene, aber $\sim$-äquivalente Normalformen beschreiben verschiedene **Zustände** ein und desselben Objektes. Es geht hier generell um die Formalisierung von Systemen, über die man nicht anhand des statischen Aufbaus ihrer Objekte, wohl aber anhand von **Beobachtungen** ihres **Verhaltens** Aussagen machen kann (oder will). Das Verhalten setzt sich zusammen aus Werten von Objekt-*Attributen*, die den augenblicklichen Zustand des Systems kennzeichnen, und aus Zustandsveränderungen, die aus der Anwendung von **Prozeduren** und **Methoden** resultieren. Hier sind sie also, all die bekannten Begriffe aus der imperativen und objekt-orientierten Programmierwelt. Einen Datentyp, dessen Objekte in dieser indirekten Weise identifiziert werden, nennt man dann auch einen **versteckten Datentyp**. Jede Normalform einer versteckten Sorte beschreibt einen Zustand. Ihre Konstruktoren sind die Methoden, die man aufrufen muss, um ihn zu erreichen. Methoden und Prozeduren können auch nichtdeterministisch sein, d.h. wir sollten auch echte Zustandsübergangs*relationen* miteinbeziehen. Diese werden in der Regel als Automaten oder, allgemeiner, als **markierte Transitionssysteme**, englisch: **labelled transition systems (LTS)**, formalisiert.

9.1 Modallogik und ihre Derivate

Der klassische Ansatz, formal über Transitionssysteme zu reden, ist die Modallogik. Ihre Formeln sind Aussagen über einzelne Zustände. Die Gültigkeit einer modallogischen Formel hängt von einem gegebenen Zustand ab, der aber in der Formel nicht vorkommt. Demzufolge sind die Symbole zur Bezeichnung von Zuständen auch nicht Teil der Signatur, über der die Formeln gebildet werden. Wie bei imperativen und objektorientierten Programmiersprachen bleiben die Zustände im (semantischen) Hintergrund. In der Syntax tauchen sie nirgends auf.

Ein **Transitionssystem** $\mathcal{T}$ wird üblicherweise definiert als Tripel $(Z, Act, \longrightarrow)$, bestehend aus der **Zustandsmenge** Z, der **Aktionsmenge** Act und einer dreistelligen **Übergangsrelation** $\longrightarrow \subseteq Z \times Act \times Z$. Für alle $M \subseteq Act$ und $z, z' \in Z$ ist

$$z \xrightarrow{M} z' \iff_{def} \exists\, act \in M : z \xrightarrow{act} z'.$$

Intuitiv bedeutet $z \xrightarrow{M} z'$, dass eine der Aktionen von M den Zustand z in den Zustand z' überführt. $\xrightarrow{*}$ bezeichnet den reflexiv-transitiven Abschluß von $\longrightarrow =_{def} \xrightarrow{Act}$.

Die Syntax modallogischer Formeln umfaßt diejenige prädikatenlogischer Formeln. Hinzu kommen *modale Operatoren*, deren Bedeutung von $\mathcal{T}$ abhängt, sowie Konstanten und Variablen für Zustandsmengen:

modallogische Formel

Definition 9.1.1 Sei $\Sigma = (S, F, R)$ eine Signatur, X eine S-sortierte Menge von **Individuenvariablen** oder **Variablen erster Ordnung** und $\mathcal{Z}$ eine Menge von **Variablen zweiter Ordnung** für

Zustandsmengen. Die Menge der **(modallogischen) Σ-Formeln (über X und $\mathcal{Z}$)** ist induktiv definiert:

- Jedes Element von $\mathcal{Z}$ ist eine Σ-Formel.
- Für jede Teilmenge Z' von Z ist $\overline{Z'}$ eine Σ-Formel.
- Jedes Σ-Atom ist eine Σ-Formel.
- Für alle Σ-Formeln φ, ψ, $x \in X$ und $t \in T_\Sigma(X)$ sind auch $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftrightarrow \psi$, $\forall x\varphi$, $\exists x\varphi$ und $(\lambda x.\varphi)(t)$ Σ-Formeln.
- Für alle Σ-Formeln φ und $M \subseteq Act$ sind auch $[M]\varphi$, $\langle M\rangle\varphi$, $\Box\varphi$ und $\Diamond\varphi$ Σ-Formeln.
- Für alle Σ-Formeln φ und $x \in \mathcal{Z}$ sind auch $\mu x.\varphi$ und $\nu x.\varphi$ und Σ-Formeln.

$[M]$, $\langle M\rangle$, $\Box$, $\Diamond$, μ und ν nennt man **modale Operatoren**.

Die Modallogik mit den **Fixpunktoperatoren** μ und ν heisst auch **μ-Kalkül**. μ und ν binden Zustandsmengenvariablen so, wie das λ, $\forall$ und $\exists$ Variablen von X binden. Freie Variablen(vorkommen) und geschlossene Formeln sind daher analog zur Prädikatenlogik definiert. Die folgenden Äquivalenzen sind im Sinne der Semantik modallogischer Formeln (siehe Def. 9.1.2) allgemeingültig:

$$\begin{array}{lcl} \nu x.(\varphi \wedge [Act]x) & \iff & \Box\varphi \\ \mu x.(\varphi \wedge [Act]x) & \iff & False \iff \Diamond False \\ \mu x.(\varphi \vee \langle Act\rangle x) & \iff & \Diamond\varphi \\ \nu x.(\varphi \vee \langle Act\rangle x) & \iff & True \iff \Box True \end{array}$$

Kripke-Struktur, Gültigkeit

Definition 9.1.2 Sei A eine S-sortierte Menge, $Struct(A, \Sigma)$ die Klasse aller Σ-Strukturen B mit $\equiv$-Gleichheit und $B_s = A_s$ für alle $s \in S$ und $\mathcal{T} = (Z, Act, \longrightarrow)$ ein Transitionssystem.

Eine (Z, Σ)**-Kripke-Struktur** ist eine Funktion $\mathcal{A} : Z \to Struct(A, \Sigma)$, d.h. $\mathcal{A}$ ordnet jedem Zustand eine Σ-Struktur zu.

Sei φ eine modallogische Σ-Formel über X und $\mathcal{Z}$, $y \in X$, $z \in Z$, $b : X \to A$ und $c : \mathcal{Z} \to \wp(Z)$. Die Eigenschaft b, c **löst** φ **in** $\mathcal{A}, z$, geschrieben: $\mathcal{A}, z \models_{b,c} \varphi$, ist induktiv definiert über dem syntaktischen Aufbau von φ: Sei $x \in X$, $Z' \subseteq Z$, $r(t)$ ein Σ-Atom und $M \subseteq Act$.

- $\mathcal{A}, z \models_{b,c} x \iff_{def} z \in c(x)$.
- $\mathcal{A}, z \models_{b,c} \overline{Z'} \iff_{def} z \in Z'$.
- $\mathcal{A}, z \models_{b,c} r(t) \iff_{def} b^*(t) \in r^{\mathcal{A}(z)}$.[57]
- $\mathcal{A}, z \models_{b,c} \neg\varphi \iff_{def} \mathcal{A}, z \models_{b,c} \varphi$ gilt nicht.
- $\mathcal{A}, z \models_{b,c} \varphi \wedge \psi \iff_{def} \mathcal{A}, z \models_{b,c} \varphi$ und $\mathcal{A}, z \models_{b,c} \psi$.
- $\mathcal{A}, z \models_{b,c} \varphi \vee \psi \iff_{def} \mathcal{A}, z \models_{b,c} \varphi$ oder $\mathcal{A}, z \models_{b,c} \psi$.
- $\mathcal{A}, z \models_{b,c} \varphi \Rightarrow \psi \iff_{def} \mathcal{A}, z \models_{b,c} \varphi$ impliziert $\mathcal{A}, z \models_{b,c} \psi$.
- $\mathcal{A}, z \models_{b,c} \varphi \Leftrightarrow \psi \iff_{def} \mathcal{A}, z \models_{b,c} (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$.
- $\mathcal{A}, z \models_{b,c} \forall x\varphi \iff_{def} \mathcal{A}, z \models_{b[a/x],c} \varphi$ für alle $a \in A$.

[57] b^* interpretiert Funktionssymbole in $\mathcal{A}(z)$!

- $\mathcal{A}, z \models_{b,c} \exists x\varphi \iff_{def} \mathcal{A}, z \models_{b[a/x],c} \varphi$ für ein $a \in A$.
- $\mathcal{A}, z \models_{b,c} (\lambda x.\varphi)(t) \iff_{def} \mathcal{A}, z \models_{b[b^*(t)/x],c} \varphi$.
- $\mathcal{A}, z \models_{b,c} [M]\varphi \iff_{def} \mathcal{A}, z' \models_{b,c} \varphi$ für alle $z' \in Z$ mit $z \xrightarrow{M} z'$.
- $\mathcal{A}, z \models_{b,c} \langle M\rangle\varphi \iff_{def} \mathcal{A}, z' \models_{b,c} \varphi$ für ein $z' \in Z$ mit $z \xrightarrow{M} z'$.
- $\mathcal{A}, z \models_{b,c} \Box\varphi \iff_{def} \mathcal{A}, z' \models_{b,c} \varphi$ für alle $z' \in Z$ mit $z \xrightarrow{*} z'$.
- $\mathcal{A}, z \models_{b,c} \Diamond\varphi \iff_{def} \mathcal{A}, z' \models_{b,c} \varphi$ für ein $z' \in Z$ mit $z \xrightarrow{*} z'$.

Sei $x \in \mathcal{Z}$. Die Funktion

$$\begin{array}{rcl} \Phi : \wp(Z) & \longrightarrow & \wp(Z) \\ Z' & \longmapsto & \{z \in Z \mid \mathcal{A}, z \models_{b,c[Z'/x]} \varphi\} \end{array}$$

sei monoton bzgl. der Teilmengenrelation auf $\wp(Z)$. Dann existieren nach Satz 4.2.10 der kleinste Fixpunkt $lfp(\Phi)$ und der größte Fixpunkt $gfp(\Phi)$ von Φ.

- $\mathcal{A}, z \models_{b,c} \mu x.\varphi \iff_{def} z \in lfp(\Phi)$.
- $\mathcal{A}, z \models_{b,c} \nu x.\varphi \iff_{def} z \in gfp(\Phi)$.

$\mathcal{A}, z$ **erfüllt** φ oder ist ein **Modell von** φ oder φ **gilt in** $\mathcal{A}, z$, geschrieben: $\mathcal{A}, z \models \varphi$, wenn alle Paare von Belegungen $b : X \to \mathcal{A}(z)$ bzw. $c : \mathcal{Z} \to \wp(Z)$ φ lösen. $\mathcal{A}$ **erfüllt** φ, geschrieben: $\mathcal{A} \models \varphi$, wenn $\mathcal{A}, z \models \varphi$ für alle $z \in Z$ gilt. φ ist **allgemeingültig**, wenn alle (Z, Σ)-Kripke-Strukturen φ erfüllen.

$z, z' \in Z$ sind **semantisch äquivalent**, wenn für alle (Z, Σ)-Kripke-Strukturen $\mathcal{A}$ und alle modallogischen Σ-Formeln φ gilt:

$$\mathcal{A}, z \models \varphi \iff \mathcal{A}, z' \models \varphi.$$

☞ Zeigen Sie die Allgemeingültigkeit der beiden o.g. Äquivalenzen!

Den Beweis einer Behauptung $\mathcal{A}, z \models \varphi$ nennt man **lokales model checking**. Ähnlich der Interpretation einer prädikatenlogischen Formel als Relation (abgeleitetes Prädikat; siehe Def. 4.1.8) interpretiert man eine modallogische Formel – in Abhängigkeit von b und c (s.o.) – als Menge der Zustände, in denen sie gilt:

$$\varphi^{\mathcal{A}}(b, c) \quad =_{def} \quad \{z \in Z \mid \mathcal{A}, z \models_{b,c} \varphi\}.$$

Damit ist $(\mu x.\varphi)^{\mathcal{A}}$ die kleinste und $(\nu x.\varphi)^{\mathcal{A}}$ die größte Lösung der Gleichung $x = \varphi$ in (der Zustandsmengenvariablen) x. Die Bestimmung von φ^A wird **globales model checking** genannt.

Wie prädikatenlogische, so sind auch modale Operatoren Negationen von anderen:

$$\begin{array}{rcl} [M]\varphi & \iff & \neg\langle M\rangle\neg\varphi \\ \langle M\rangle\varphi & \iff & \neg[M]\neg\varphi \\ \Box\varphi & \iff & \neg\Diamond\neg\varphi \\ \Diamond\varphi & \iff & \neg\Box\neg\varphi \\ \mu x.\varphi & \iff & \neg(\nu x.\neg\varphi[\neg x/x]) \\ \nu x.\varphi & \iff & \neg(\mu x.\neg\varphi[\neg x/x]) \end{array}$$

Noch zwei nützliche Operatoren:

$$\begin{array}{rcll} enabled(M) & \iff_{def} & \langle M\rangle\, True & \text{“eine Aktion von } M \text{ ist ausführbar”} \\ disabled(M) & \iff_{def} & [M] False & \text{“keine Aktion von } M \text{ ist ausführbar”} \end{array}$$

☞ Zeigen Sie, dass die Äquivalenz

$$(\lambda x.\varphi)(t) \iff \varphi[t/x]$$

für alle prädikatenlogischen (!) Formeln φ allgemeingültig ist! Demnach hat der λ-Operator in der Prädikatenlogik dieselbe Bedeutung wie der Substitutionsoperator $[\_/\_]$, weshalb man dort λ auch gar nicht erst einführt.[58] Enthält φ jedoch modale Operatoren, dann gilt die Äquivalenz nicht mehr. Geben Sie dafür Beispiele an!

☞ Beschreiben Sie verbal, wann die Formeln $(\lambda x.[act]x \equiv u)(t)$ bzw. $(\lambda x.\langle act\rangle x \equiv u)(t)$ allgemeingültig ist!

$Z' \subseteq Z$ ist nach der o.a. Definition von $\Phi : \wp(Z) \to \wp(Z)$ genau dann ein Fixpunkt dieser Funktion, wenn gilt:

$$z \in Z' \iff \mathcal{A}, z \models_{b,c[Z'/x]} \varphi.$$

M.a.W. Z' löst die Äquivalenz $x \Leftrightarrow \varphi$ in x. Die Bedeutung von $\mu x.\varphi$ bzw. $\nu x.\varphi$ ist demnach die kleinste bzw. größte Lösung der Äquivalenz $x \Leftrightarrow \varphi$. Analog zu entsprechenden Bemerkungen in §4.2 ist Φ monoton, wenn es eine negations- und implikationsfreie Formel ψ gibt derart, dass die Äquivalenz $\varphi \Leftrightarrow \psi$ allgemeingültig ist.

☞ Zeigen Sie, dass die Formel $\nu x.[tick]x$ allgemeingültig ist!

☞ Zeigen Sie, dass es kein $\mathcal{A}, z$ gibt, dass die Formel $\mu x.\langle tick\rangle x$ erfüllt!

☞ Zeigen Sie, dass $\mathcal{A}, z \models \nu x.\langle tick\rangle x$ genau dann gilt, wenn $\mathcal{T}$ einen unendlichen Pfad $z \xrightarrow{tick} z_1 \xrightarrow{tick} z_2 \xrightarrow{tick} \ldots$ enthält!

☞ Zeigen Sie, dass $\mathcal{A}, z \models \mu x.[tick]x$ genau dann gilt, wenn alle Pfade $z \xrightarrow{tick} z_1 \xrightarrow{tick} z_2 \xrightarrow{tick} \ldots$ endlich sind!

Man spricht von **aussagenlogischer** oder **propositionaler Modallogik**, wenn Σ leer ist, also Konstanten und Variablen für Zustandsmengen die einzigen elementaren Σ-Formeln sind. Die Gültigkeit einer Σ-Formel hängt dann nur von Belegungen von $\mathcal{Z}$ ab. Nichtpropositionale Modallogik nennt man auch **prädikative** oder **Modallogik erster Stufe**.

Die Übergangsrelation eines Transitionssystem bestimmt das **Verhalten** seiner Zustände. Verhaltensäquivalenz wird meist als *Bisimilarität* formalisiert:

Bisimulation

Definition 9.1.3 Sei $\mathcal{T} = (Z, Act, \longrightarrow)$ ein Transitionssystem. Eine **$\mathcal{T}$-Bisimulation** ist eine binäre Relation $\sim$ auf Z mit folgenden Eigenschaften:

(1) $z_1 \sim z_1' \wedge z_1 \xrightarrow{a} z_2 \;\Rightarrow\; \exists z_2' \in Z : z_1' \xrightarrow{a} z_2' \wedge z_2 \sim z_2'.$

(2) $z_1 \sim z_1' \wedge z_1' \xrightarrow{a} z_2' \;\Rightarrow\; \exists z_2 \in Z : z_1 \xrightarrow{a} z_2 \wedge z_2 \sim z_2'.$

Die Relation

$$\sim_{\mathcal{T}} \;=_{def}\; \{(z, z') \in Z^2 \mid (z, z') \text{ gehört zu einer Bisimulation auf } Z\}$$

heisst **$\mathcal{T}$-Bisimilarität**.

$\mathcal{T}$ ist **endlich verzweigend**, wenn für alle $z \in Z$ die Menge der Zustände z' mit $z \xrightarrow{Act} z'$ endlich ist.

☞ Zeigen Sie, dass jede Vereinigung von Bisimulationen wieder eine Bisimulation ist! M.a.W.: $\sim_{\mathcal{T}}$ ist die größte Relation auf Z, die (1) und (2) erfüllt. Da (1) und (2) co-Hornformeln sind, kann man $\sim_{\mathcal{T}}$ als Coprädikat im Sinne von Def. 6.2.1 spezifizieren.

[58] Es geht hier um einen λ-Operator auf *Formeln*, nicht den auf *Termen*, der in §6.6 behandelt wurde!

☞ Zeigen Sie, dass $\sim_{\mathcal{T}}$ eine Äquivalenzrelation ist!

Satz 9.1.4 (Hennessy-Milner-Theorem) ([106], 5.3.2 u. 5.3.3) Sei $\mathcal{T} = (Z, Act, \longrightarrow)$ ein endlich verzweigendes Transitionssystem. Zwei Zustände $z, z' \in Z$ sind genau dann bisimilär, wenn sie semantisch äquivalent sind. ❑.

Beschreiben die Aktionen von $\mathcal{T}$ die Programme einer imperativen Sprache, dann nennt man die auf $\mathcal{T}$ aufbauende Modallogik eine **dynamische Logik** (s. z.B. [94], [111], [53]). Dynamische Logik entstand aus der in §10.2 behandelten **Zusicherungslogik** mit den wohlbekannten Regeln des **Hoare-Kalküls**. Eine Zusicherung $\{pre\}act\{post\}$ mit Vorbedingung pre, Aktion act und Nachbedingung $post$ entspricht der modallogischen Formel $pre \Rightarrow [act]post$. Als Mittel zur Verifikation imperativer Programme gut zu handhaben ist dynamische Logik u.E. nur für iterative Programme. Zusicherungsregeln für komplexere Programmkonstrukte wie rekursive Prozeduren sind z.T. mit vielen Anwendbarkeitsbedingungen versehen, die es schwermachen, die Regeln beider automatischen Beweisunterstützung zu verwenden. Die meisten in der Softwaretechnik eingesetzten Verifikationsumgebungen wie VDM [57], Z [105] und Larch [43] benutzen immer dort Konzepte dynamischer Logik, wo imperative/objekt-orientierte Sprachkonstrukte ins Spiel kommen. Charakteristisch ist dabei — wie in der Modallogik generell — die im Formalismus wiedergespiegelte Trennung der "statischen" von der "dynamischen" Ebene: Signatur und Struktur zur Formulierung von Vor- und Nachbedingungen auf der einen Seite und ein Transitionssystem mit einer Sprache für Aktionen auf der anderen Seite. Während sich VDM und Z auf der statischen Ebene mit klassischer Mengenlehre und Relationenalgebra begnügen, verlangt *Larch* (Horn-)Axiome für die in Vor- und Nachbedingungen verwendeten Funktionen. Mithilfe sog. Interface-Sprachen werden dort Zusicherungen in ausschließlich dynamische Form gebracht. Dieser wird dann automatisch in den Code der zur jeweiligen Interface-Sprache gehörigen Programmiersprache übersetzt.

Auch wenn die Aktionen von $\mathcal{T}$ keine Σ-Terme oder -Formeln sind, so enthalten sie doch in der Regel solche. Aus der üblichen Syntax imperativer Sprachen ergeben sich zwei Typen von Elementaraktionen: **(deterministische) Zuweisungen** $v := t$ eines Grundterms t an eine **dynamische Variable** oder **Programmvariable** genannte Konstante (!) v der Sorte von t sowie **(einfache) Tests** $u?$, wobei u ein Boolescher Grundterm ist. Andere Elementaraktionen sind **nichtdeterministische Zuweisungen** $v :=?$ und **reiche Tests** $\varphi?$, wobei φ eine beliebige geschlossene prädikatenlogische Σ-Formel ist. Der Ansatz der neuerdings **abstrakte Zustandsmaschinen** [1] genannten **evolvierenden Algebren** [37] basiert auf **(bedingten) Updates** $\varphi \Rightarrow f(x) := t$ als Elementaraktionen, wobei φ eine Σ-Formel ist und $f(x)$ und t Σ-Terme sind, deren (freie) Variablen zu den Komponenten von x gehören.

Wand [111] verwendet **Prozeduraufrufe** als Elementaraktionen, das sind Ausdrücke der Form $r(v; t)$, wobei $r(v, t)$ ein Grundatom und v ein Tupel von Programmvariablen ist. In Lamport's **Temporal Logic of Actions (TLA)** [55] ist eine Aktion eine geschlossene prädikatenlogische Formel $\psi(v_1, \ldots, v_n, v'_1, \ldots, v'_n)$, wobei $v_1, \ldots, v_n, v'_1, \ldots, v'_n$ Programmvariablen. Z.B. entspricht der Zuweisung $v := v + 1$ die TLA-Aktion $\psi(v, v') = (v' = v + 1)$.[59]

Mit TLA-Aktionen lassen sich vor allem die von nebenläufigen Programmen bewirkten Zustandsübergänge abstrakt beschreiben. So garantiert z.B. das folgende Programm den gegenseitigen Ausschluß beim Zugriff auf das Medium M:

```
x:=0; y:=0;
parbegin  (A) do y = 0 -> (B) (x:=1; access M; x:=0; skip) od
          (A) do x = 0 -> (B) (y:=1; access M; y:=0; skip) od
parend
```

Das Programm erlaubt vier Zustandsübergänge, die durch entsprechende TLA-Aktionen $\varphi_1, \ldots, \varphi_4$ beschrieben

[59] Diese Notation wird manchmal auch in Zusicherungen verwendet: Man schreibt z.B. $\{v > 0\}act\{v' = v + 1\}$ als Abkürzung für $\{v > 0 \wedge v = a\}act\{v = a + 1\}$ (siehe auch §10.2).

werden können. Sie verwenden neben den Programmvariablen x und y zwei Programmzähler pc_1 und pc_2, die angeben, wo sich der erste bzw. zweite Prozess bei einer Ausführung des Programms gerade befindet: am Punkt A oder am Punkt B. $\varphi_1, \ldots, \varphi_4$ lauten wie folgt:

$$
\begin{array}{ll}
(\varphi_1) & pc_1 \equiv A \wedge y \equiv 0 \wedge pc_1' \equiv B \wedge pc_2' \equiv pc_2 \wedge x' \equiv 1 \wedge y' \equiv y \\
(\varphi_2) & pc_1 \equiv B \wedge pc_1' \equiv A \wedge pc_2' \equiv pc_2 \wedge x' \equiv 0 \wedge y' \equiv y \\
(\varphi_3) & pc_2 \equiv A \wedge x \equiv 0 \wedge pc_1' \equiv pc_1 \wedge pc_2' \equiv B \wedge y' \equiv 1 \wedge x' \equiv x \\
(\varphi_4) & pc_2 \equiv B \wedge pc_1' \equiv pc_1 \wedge pc_2' \equiv A \wedge y' \equiv 0 \wedge x' \equiv x
\end{array}
$$

Zu einer imperativen Sprache gehören neben Elementaraktionen natürlich Konstruktoren zur Bildung zusammengesetzter Aktionen. Das sind i.w. die **regulären Operatoren** $+$ (*Auswahl*), $;$ (*Sequenz*) und * (*Iteration*) sowie daraus abgeleitete Operatoren wie Konditionale und bedingte Schleifen. Die aus Elementaraktionen und solchen Konstruktoren bestehenden Ausdrücke bilden die Menge Act des Transitionssystems $\mathcal{T} = (Z, Act, \longrightarrow)$ einer dynamischen Logik und zugleich die Menge der Programme, über die man in der Logik Aussagen machen will.

Dynamische Logik bezieht das zugrundeliegende Transitionssystem $\mathcal{T}$ in die Semantik modallogischer Formeln ein, indem es die Menge der Kripke-Strukturen, die Σ-Formeln interpretieren, einschränkt. Damit liefert die Übergangsrelation $\longrightarrow$ von $\mathcal{T}$ zugleich eine **operationelle Semantik** für Act-Programme. Den o.g. Elementaraktionen und Aktionskonstruktoren entsprechen dabei folgende Bedingungen an eine interpretierende Kripke-Struktur $\mathcal{A} : Z \to Mod(A, \Sigma)$: Sei v eine Konstante von Σ, $t \in T_\Sigma$, r ein Prädikat von Σ, $u \in T_{\Sigma,bool}$, $f : w \to s \in \Sigma$, $x \in X_w$, φ eine prädikatenlogischen Σ-Formel mit $var(\varphi) = \{x\}$, $t \in T_\Sigma(\{x\})_s$ und $\psi(v_1, \ldots, v_n, v_1', \ldots, v_n')$ eine TLA-Aktion.[60]

$$
\begin{array}{lrcl}
(1) & z \xrightarrow{v:=?} z' & \iff & \mathcal{A}(z)|_{\Sigma\setminus\{v\}} = \mathcal{A}(z')|_{\Sigma\setminus\{v\}} \\
(2) & z \xrightarrow{v:=t} z' & \iff & v^{\mathcal{A}(z')} = t^{\mathcal{A}(z)} \ \wedge \ \mathcal{A}(z)|_{\Sigma\setminus\{v\}} = \mathcal{A}(z')|_{\Sigma\setminus\{v\}} \\
(3) & z \xrightarrow{r(v;t)} z' & \iff & r^{\mathcal{A}(z')} = r^{\mathcal{A}(z)} \cup \{(v^{\mathcal{A}(z')}, t^{\mathcal{A}(z)})\} \ \wedge \ \mathcal{A}(z)|_{\Sigma\setminus\{r,v\}} = \mathcal{A}(z')|_{\Sigma\setminus\{r,v\}} \\
(4) & z \xrightarrow{u?} z' & \iff & \mathcal{A}(z) \models u \equiv true \ \wedge \ \mathcal{A}(z) = \mathcal{A}(z') \\
(5) & z \xrightarrow{\varphi?} z' & \iff & \mathcal{A}(z) \models \varphi \ \wedge \ \mathcal{A}(z) = \mathcal{A}(z') \\
(6) & z \xrightarrow{\varphi \Rightarrow f(x):=t} z' & \iff & \forall\, a \in A_w : \ (\mathcal{A}(z) \models \varphi \Rightarrow f^{\mathcal{A}(z')}(a) \equiv t^{\mathcal{A}(z)}) \wedge \\
 & & & \qquad (\mathcal{A}(z) \not\models \varphi \Rightarrow f^{\mathcal{A}(z')}(a) \equiv f^{\mathcal{A}(z)}(a)) \wedge \\
 & & & \qquad (\mathcal{A}(z)|_{\Sigma\setminus\{f_1,\ldots,f_n\}} = \mathcal{A}(z')|_{\Sigma\setminus\{f_1,\ldots,f_n\}}) \\
(7) & z \xrightarrow{\psi(v_1,\ldots,v_n,v_1',\ldots,v_n')} z' & \iff & (v_1^{\mathcal{A}(z)}, \ldots, v_n^{\mathcal{A}(z)}, v_1^{\mathcal{A}(z')}, \ldots, v_n^{\mathcal{A}(z')}) \in \psi^{\mathcal{A}(z')} \wedge \\
 & & & \mathcal{A}(z)|_{\Sigma\setminus\{v_1,\ldots,v_n\}} = \mathcal{A}(z')|_{\Sigma\setminus\{v_1,\ldots,v_n\}} \\
(8) & z \xrightarrow{act_1;act_2} z' & \iff & \exists z'' \in Z : (z \xrightarrow{act_1} z'' \wedge z'' \xrightarrow{act_2} z') \\
(9) & z \xrightarrow{act_1+act_2} z' & \iff & z \xrightarrow{act_1} z' \vee z \xrightarrow{act_2} z' \\
(10) & z \xrightarrow{act^*} z' & \iff & \exists n \in \mathbb{N} : z \xrightarrow{act^n} z' \\
(11) & z \xrightarrow{act^0} z' & \iff & z = z' \\
(12) & z \xrightarrow{act^{n+1}} z' & \iff & z \xrightarrow{act;act^n} z'
\end{array}
$$

Konditionale und Schleifen sind abgeleitete Aktionen:

$$
\begin{array}{rcl}
if\ t\ then\ act_1\ else\ act_2 & = & (t?; act_1) + (\neg t?; act_2) \\
while\ t\ do\ act & = & (t?; act)^*; \neg t? \\
repeat\ act\ until\ t & = & act; while\ \neg t\ do\ act
\end{array}
$$

[60] ψ wird hier als abgeleitetes Prädikat aufgefasst (vgl. Def. 4.1.8).

Besteht die Aktionsmenge von $\mathcal{T}$ beispielsweise aus den o.g. TLA-Aktionen $\varphi_1, \ldots, \varphi_4$, dann gilt in allen Kripke-Strukturen, die (7) erfüllen, folgende Formel, die die Korrektheit des $\varphi_1, \ldots, \varphi_4$ entsprechenden Programms beschreibt (gegenseitiger Ausschluß beim Zugriff auf M):

$$pc_1 \equiv A \wedge pc_2 \equiv A \wedge x \equiv 0 \wedge y \equiv 0 \quad \Rightarrow \quad \Box(pc_1 \equiv A \vee pc_2 \equiv A).$$

☞ Zeigen Sie, dass eine Zuweisung sowohl ein Prozeduraufruf als auch ein (unbedingter) Update ist!

☞ Zeigen Sie, dass folgende Formeln in allen Kripke-Strukturen gelten, die o.g. Bedingungen erfüllen:

$(\lambda x.[v := t]v \equiv x)(t)$	für Zuweisungen $v := t$
$(\lambda x.[r(v;t)]r(v,x))(t)$	für Prozeduraufrufe $r(v;t)$
$(\lambda(x_1,\ldots,x_n).[\varphi]\varphi[x_1/v_1, v_1/v_1', \ldots, x_n/v_n, v_n/v_n'])(v_1,\ldots,v_n)$	für TLA-Aktionen φ über $v_1, \ldots, v_n$

☞ Zeigen Sie, dass folgende Äquivalenzen in allen Kripke-Strukturen gelten, die o.g. Bedingungen erfüllen:

$$\begin{aligned}
[\varphi?]\psi &\iff \varphi \Rightarrow \psi \\
\langle\varphi?\rangle\psi &\iff \varphi \wedge \psi \\
[act_1; act_2]\varphi &\iff [act_1][act_2]\varphi \\
\langle act_1; act_2\rangle\varphi &\iff \langle act_1\rangle\langle act_2\rangle\varphi \\
[act_1 + act_2]\varphi &\iff [act_1]\varphi \wedge [act_2]\varphi \\
\langle act_1 + act_2\rangle\varphi &\iff \langle act_1\rangle\varphi \vee \langle act_2\rangle\varphi \\
[act^*]\varphi &\implies \varphi \wedge [act][act^*]\varphi \\
\langle act^*\rangle\varphi &\impliedby \varphi \vee \langle act\rangle\langle act^*\rangle\varphi
\end{aligned}$$

Einen anderen Ansatz zur Formalisierung imperativer Sprachen erhält man aus der Einschränkung von Z auf **Speicherzustände**, d.h. auf die Menge der Belegungen dynamischer Variablen durch Daten der jeweiligen Kripke-Struktur. Dieses Zustandsmodell wird auch häufig mit den oben aufgeführten Aktionsschemata assoziiert. Es liegt auch den CPO-Modellen imperativer Sprachen zugrunde (s. Bsp. 10.1.10). Es ist jedoch wenig abstrakt und daher kaum erweiterungsfähig im Hinblick auf höhere Sprachkonstrukte wie die Deklaration und Verwendung strukturierter dynamischer Objekte, die objektorientierte Sprachen kennzeichnen. Andererseits bilden Speicherzustände einen — ARRAY vergleichbaren (s. Bsp. 6.4.3) — Datentyp, so dass man Z und dann auch die Übergangsrelation $\longrightarrow$ zu Σ hinzunehmen und so statische *und* dynamische Komponenten im selben Formalismus behandeln kann. Mit versteckten Datentypen läßt sich diese Eigenschaft des CPO-Ansatzes auch ohne Festlegung von Z auf Speicherzustände realisieren (s. §9.2).

Mit dem Aktionskonstruktor + enthält die oben definierte imperative Sprache bereits ein Konstrukt zur nichtdeterministischen Prozesserzeugung und ist daher nicht weit entfernt von Prozesssprachen, die neben + Operatoren zur Parallelisierung, zur Synchronisation und zum Nachrichtenaustausch anbieten (s. z.B. [68], [12], [69]). Bezogen auf unsere imperative Sprache könnte man Elementaraktionen als Nachrichten betrachten, die bei der Ausführung eines Programms abgesetzt werden. Deshalb sind — zumindest in der SOS-Version der Sprache — alle Transitionsmarkierungen Elementaraktionen. Bei der Spezifikation von Prozesssprachen und dynamischen Systemen ist es aber eher üblich, ähnlich der SOS-Version vorzugehen und Transitionsmarkierungen auf die von der Quelle der jeweiligen Transition gesendete oder empfangene Nachrichten zu beschränken. Das gilt schon für die einfachsten Transitionssysteme, die endlichen Automaten, deren Transitionsmarkierungen Ein- oder Ausgabedaten darstellen.

Die Zustände des Transitionssystems einer Prozesssprache sind in der Regel Darstellungen von Prozessen, die von den jeweiligen Zuständen aus möglich sind. Der Begriff “Zustand” paßt dann nicht mehr so recht. “Aktion” hingegen würde zur Verwechslung mit den Transitionsmarkierungen führen, die weiterhin Aktionen

heissen (obwohl sie lieber Nachrichten heissen sollten; s.o.). Also spricht man von **Agenten** oder gleich von **Prozessen**. Bezogen auf ihren Gebrauch in der Modallogik sind Prozesse jedoch nichts anderes Zustände. Intuitiv interpretiert man eine Zustandsübergang $P \xrightarrow{act} Q$ zwischen Prozessen wie folgt: P kann (!) act ausführen und verhält sich nach der Ausführung von act wie Q. Die Gleichung Prozess=Zustand hat zur Folge, dass Bisimilarität als Prozess und damit als Programmäquivalenz betracht werden kann.

In dem von Milner [68] entwickelten **Prozesskalkül** CCS (*Calculus of Communicating Systems*) sind die Prozesse aus den Konstruktoren $a.$, $+$, $|$, $\setminus$ und Konstanten A zusammengesetzte Ausdrücke. Die (operationelle) Semantik von CCS ist durch folgende Übergangsrelation gegeben:[61]

$$
\begin{array}{rcll}
a(x).P \xrightarrow{a(v)} P[v/x] & & & (lesen) \\
\overline{a}(e).P \xrightarrow{\overline{a}(val(e))} P & & & (schreiben) \\
P_1 + P_2 \xrightarrow{a} Q & \iff & P_1 \xrightarrow{a} Q \ \vee \ P_2 \xrightarrow{a} Q & (auswählen) \\
P|P' \xrightarrow{a} Q|P' & \iff & P \xrightarrow{a} Q & (parallelisieren) \\
P'|P \xrightarrow{a} P'|Q & \iff & P \xrightarrow{a} Q & \\
P_1|P_2 \xrightarrow{\tau} Q_1|Q_2 & \iff & P_1 \xrightarrow{a} Q_1 \ \wedge \ P_2 \xrightarrow{\overline{a}} Q_2 & (kommunizieren) \\
P \setminus M \xrightarrow{a} Q \setminus M & \iff & P \xrightarrow{a} Q \ \wedge \ a \in Act \setminus M \setminus \{\overline{b} \mid b \in M\} & (einschränken) \\
A \xrightarrow{a} Q & \iff & P \xrightarrow{a} Q \quad \text{falls } A \text{ durch } P \text{ definiert wurde}[62] & (aufrufen)
\end{array}
$$

☞ Zeigen Sie, dass die Prozesse $a.(P_1 + P_2)$ — erst a, dann weiter mit P_1 oder P_2 — und $(a.P_1) + (a.P_2)$ — erst a, dann P_1, oder erst a, dann P_2 — bzgl. CCS nicht bisimilär sind! Diese beiden Prozesse als ungleich zu betrachten ist beabsichtigt: $a.(P_1 + P_2)$ entscheidet sich erst nach Ausführung von a zwischen zwei Unterprozessen, während $(a.P_1) + (a.P_2)$ das schon davor tut.

Temporale (linear-time) Logik dient der Formulierung von Aussagen über Zustandsfolgen und nicht über einzelne Zustände. Ist $\mathcal{T} = (Z, Act, \longrightarrow)$ ein Transitionssystem, dann heisst eine Funktion $\pi : \mathbb{N} \to Z$ **Spur** (*trace*) oder **Ablauf** (*run, fullpath*) von $\mathcal{T}$, wenn entweder für alle $i \in \mathbb{N}$ $\pi(i) \xrightarrow{Act} \pi(i+1)$ gilt oder ein $i \in \mathbb{N}$ existiert mit $\pi(i) \not\xrightarrow{Act} z$ für alle $z \in Z$, $\pi(j) \xrightarrow{Act} \pi(j+1)$ für alle $j < i$ und $\pi(j) = \pi(i)$ für alle $j > i$. Hier sind die wichtigsten temporalen Operatoren und ihre Semantik, die man als Erweiterung der modallogischen Erfüllbarkeitsrelation (s. Def. 9.1.1) betrachten kann:[63] Sei π eine Spur von $\mathcal{T}$ und $M \subseteq Act$.

❀ $\pi \models (M)\varphi \iff_{def} \pi(0) \xrightarrow{M} \pi(1)$ und $\lambda i.\pi(i+1) \models \varphi$

❀ $\pi \models \varphi \ U \ \psi \iff_{def} \exists k \in \mathbb{N} : (\lambda i.\pi(i+k) \models \psi \wedge \forall j < k : \lambda i.\pi(i+j) \models \varphi)$
"φ until ψ" (irgendwann gilt ψ auf der Spur π und bis dahin gilt φ)

❀ $\pi \models \varphi \ wU \ \psi \iff_{def} \pi \models \varphi \ U \ \psi \ \vee \ \forall k \in \mathbb{N} : \lambda i.\pi(i+k) \models \varphi$
"φ weak until ψ" ("φ until ψ" oder φ gilt auf der gesamten Spur π)

Daraus abgeleitet sind die folgenden Operatoren:

$$
\begin{array}{rcll}
X\varphi & \iff_{def} & (Act)\varphi & \text{"im nächsten Zustand gilt } \varphi\text{"} \\
executed(M) & \iff_{def} & (M)\,True & \text{"}M \text{ wird ausgeführt"} \\
F\varphi & \iff_{def} & True \ U \ \varphi & \text{"irgendwann gilt } \varphi\text{"} \\
G\varphi & \iff_{def} & \varphi \ wU \ False & \text{"}\varphi \text{ gilt immer"}
\end{array}
$$

[61] Wir verzichten hier auf die verbale Erläuterung der einzelnen Prozesskonstruktoren. Wer mehr darüber wissen will, lese [68] oder Arbeiten über **Prozessalgebra**. In §9.2 werden ähnliche Konstruktoren im Rahmen versteckter Datentypen eingeführt.

[62] Formal ist die Bedeutung von A eine Lösung der Gleichung $A = P$ in einer geeigneten Struktur. Sie existiert immer dann, wenn P **bewacht-rekursiv**, d.h. von der Form $a_1.P_1 + \cdots + a_n.P_n$ ist.

[63] $\mathcal{A}$ und die Belegungen b und c lassen wir hier der Einfachheit halber weg.

$$\begin{array}{rcll} \varphi \rightsquigarrow \psi & \Longleftrightarrow_{def} & G(\varphi \Rightarrow F\psi) & \text{“}\varphi \text{ führt zu } \psi\text{”} \\ G^{\infty}\varphi & \Longleftrightarrow_{def} & FG\varphi & \text{“}\varphi \text{ gilt fast immer”} \\ F^{\infty}\varphi & \Longleftrightarrow_{def} & GF\varphi & \text{“}\varphi \text{ gilt unendlich oft”} \end{array}$$

Eine modallogische Formel gilt für eine Spur π, wenn sie im ersten Zustand von π gilt:

❀ $\pi \models \varphi \quad \Longleftrightarrow_{def} \quad \pi(0) \models \varphi$

Fairneßbedingungen lassen sich als Kombinationen modaler und temporaler Operatoren formulieren:

$$\begin{array}{rcll} UF(M) & \Longleftrightarrow_{def} & F^{\infty}executed(M) & \text{(unbedingte Fairneß)} \\ WF(M) & \Longleftrightarrow_{def} & G^{\infty}enabled(M) \Rightarrow F^{\infty}executed(M) & \text{(schwache Fairneß)} \\ SF(M) & \Longleftrightarrow_{def} & F^{\infty}enabled(M) \Rightarrow F^{\infty}executed(M) & \text{(starke Fairneß)} \end{array}$$

☞ Beschreiben Sie die Fairneßoperatoren verbal!

Modale Aussagen über die Anfangszustände von Spuren, die durch temporale Eigenschaften charakterisiert sind, führen zur **temporalen branching-time** oder **computation-tree Logik (CTL)**:

❀ $z \models E\varphi \quad \Longleftrightarrow_{def} \quad \exists \pi : (\pi(0) = z \wedge \pi \models \varphi)$

❀ $z \models E_{\infty}\varphi \quad \Longleftrightarrow_{def} \quad \exists \textit{ unendliches } \pi : (\pi(0) = z \wedge \pi \models \varphi)$

❀ $z \models A\varphi \quad \Longleftrightarrow_{def} \quad \forall \pi : (\pi(0) = z \Rightarrow \pi \models \varphi)$

❀ $z \models A_{\infty}\varphi \quad \Longleftrightarrow_{def} \quad \forall \textit{ unendlichen } \pi : (\pi(0) = z \Rightarrow \pi \models \varphi)$

☞ Zeigen Sie folgende Äquivalenzen:

$$\begin{array}{rcl} A(M)\varphi & \Longleftrightarrow & [M]\varphi \\ E(M)\varphi & \Longleftrightarrow & \langle M\rangle\varphi \\ AG\varphi & \Longleftrightarrow & \Box\varphi \\ EF\varphi & \Longleftrightarrow & \Diamond\varphi \\ AF\varphi & \Longleftrightarrow & \nu x.(\varphi \vee [Act]x) \\ AF\varphi & \Longleftrightarrow & \mu x.(\varphi \vee (enabled(Act) \wedge [Act]x)) \\ A_{\infty}F\varphi & \Longleftrightarrow & \mu x.(\varphi \vee [Act]x) \\ EG\varphi & \Longleftrightarrow & \mu x.(\varphi \wedge (disabled(Act) \vee \langle Act\rangle x)) \\ EG\varphi & \Longleftrightarrow & \nu x.(\varphi \wedge (enabled(Act) \Rightarrow \langle Act\rangle x)) \\ E_{\infty}G\varphi & \Longleftrightarrow & \nu x.(\varphi \wedge \langle Act\rangle x) \\ A(\varphi\ U\ \psi) & \Longleftrightarrow & A(\varphi\ wU\ \psi) \wedge AF\psi \\ A(\varphi\ U\ \psi) & \Longleftrightarrow & \mu x.(\psi \vee (\varphi \wedge enabled(Act) \wedge [Act]x)) \\ A(\varphi\ wU\ \psi) & \Longleftrightarrow & \nu x.(\psi \vee (\varphi \wedge [Act]x)) \\ A(\varphi \rightsquigarrow \psi) & \Longleftrightarrow & \nu x.((\varphi \Rightarrow AF\psi) \wedge [Act]x) \\ E(\varphi\ U\ \psi) & \Longleftrightarrow & \mu x.(\psi \vee (\varphi \wedge \langle Act\rangle x)) \\ E(\varphi\ wU\ \psi) & \Longleftrightarrow & E(\varphi\ wU\ \psi) \vee EG\varphi \\ E(\varphi\ wU\ \psi) & \Longleftrightarrow & \nu x.(\psi \vee (\varphi \wedge (enabled(Act) \Rightarrow \langle Act\rangle x))) \\ E(\varphi \rightsquigarrow \psi) & \Longleftrightarrow & \mu x.((\varphi \wedge EF\psi) \vee \langle Act\rangle x) \end{array}$$

9.2 Spezifikationen mit versteckten Sorten

Effiziente Beweisverfahren modallogischer Formeln beschränken sich bisher meist auf propositionale Formeln über einem Transitionssystem $\mathcal{T}$ mit unstrukturierten Aktionen und einer endlichen oder als kontextfreie Sprache beschreibbaren Zustandsmenge (*pushdown-Prozesse*). $\mathcal{T}$ ist hierbei die einzige Datenstruktur, über die Aussagen gemacht und bewiesen werden. Wie wir gesehen haben, führt dynamische Logik i.w. den induktiven Aufbau von Aktionen ein, während strukturell-operationelle Semantiken sowie Prozesskalküle Zustände als Terme darstellen. Die den Aufbau von Aktionen und Zuständen bestimmenden Datenstrukturen bleiben aber außerhalb der modallogischen Beschreibung. Diese befaßt sich i.w. mit den *Übergängen* zwischen "Welten" (= Zuständen = Σ-Strukturen), d.h. mit der Übergangsrelation von $\mathcal{T}$, aber kaum, mit dem, was in einer einzelnen Welt passiert. Nichtpropositionale Modallogik läßt das zwar zu, wird aber — vermutlich aufgrund ihrer syntaktischen und semantischen Komplexität — sehr viel seltener benutzt als propositionale Modallogik. Stattdessen werden zur Formulierung und Verifikation von Aussagen über einzelne Zustände eigene nichtmodale Formalismen verwendet.

Bezieht man noch temporale Logik mit ihren Aussagen über Zustandsfolgen mit ein, dann sind es schon mindestens drei Datentypen, die die Gültigkeit von Σ-Formeln bestimmen, ohne dass sie selbst in Σ vorkommen: Aktionen, Zustände und Abläufe. Letztere gehören zur Klasse der **unendlichen Objekte**, die sich — wie wir im folgenden zeigen werden — ähnlich den bisher behandelten endlichen Objekten als Grundterme darstellen lassen. Alles was sich in Modallogik an **dynamischen Eigenschaften** von Objekten ausdrücken läßt, kann auch in mehrsortiger Prädikatenlogik formuliert werden, wenn man eine Spezifikation von $\mathcal{T}$ und den jeweils zu untersuchenden modalen Prädikaten zugrundelegt.

Nicht nur SOS-Regeln und Prozesskalküle spezifizieren Transitionssysteme. Auch viele andere Formalismen zur Darstellung dynamischer und verteilter Systeme, wie **Petrinetze**, deren Zustände aus Platzmarkierungen bestehen, **Ein/Ausgabe-** und **kommunizierende Automaten**, **Statecharts** sowie auf einer **Rewriting-Logik** basierende Spezifikationen (s. [67]) sind im Grunde alles endliche Repräsentation von Transitionssystemen. Als Datentypen betrachtet gehören Transitionssysteme zur Klasse der **Spezifikationen mit versteckten Sorten**, die sich von den bisher betrachteten darin unterscheiden, dass sich die Identität ihrer Objekte nicht über deren Normalformen, sondern über eine **Verhaltensäquivalenz** definiert. Abstrakte Syntax verlangt zwar auch für "versteckte" Objekte Konstruktoren, aus denen man sie zusammensetzen kann. Ein verstecktes Objekt kann aber viele Normalformen haben. Man sollte diese als verschiedene **Namen** für dasselbe Objekt auffassen, über die man auf das Objekt zugreift, es beobachtet, zur Zustandsänderung (**Reaktion**) veranlaßt, danach erneut beobachtet und so immer mehr über seine Identität erfährt. Genau diese Identität wird bei einem Transitionssystem $\mathcal{T} = (Z, Act, \longrightarrow)$ durch die $\mathcal{T}$-Bisimilarität beschrieben (s. Def. 9.1.4).

Es sind aber nicht nur Zustände, Prozesse, Abläufe und andere unendliche Objekte (s.o.), deren Identität sich nur durch Beobachtung ihres Verhaltens erschließen läßt. Auch Objekte **permutativer Datentypen** wie Mengen, Multimengen und Felder (= Funktionen auf einer Indexmenge), egal ob endlich oder unendlich, haben keine eindeutigen Normalformen. Zur Definition der Verhaltensäquivalenz braucht man hier allerdings keine Übergangsrelation. Es genügt die Anwendung von **Destruktoren**, um festzustellen, ob zwei Objekte äquivalent sind: das Enthaltensein eines Elementes in einer Menge, die Häufigkeit eines Elementes in einer Multimenge, der Eintrag unter einem Feldindex sind Werte von Destruktoren, die die Identität einer Menge, einer Multimenge bzw. eines Feldes bestimmen.

Die Eigenschaften von Objekten, deren Identität durch Destruktoren und erreichbare Zustände, aber nicht durch die Struktur ihrer Normalformen bestimmt ist, lassen sich in der Regel nicht durch Induktion über dieser Struktur beweisen. Mithilfe der in Kapitel 10 auf partielle Funktionen angewendeten CPO-Theorie sind auch unendliche Objekte, insbesondere unendliche Listen (**Ströme**) modelliert worden (s. §11.2). Sie werden dort — ähnlich der Kleene'schen Charakterisierung kleinster relationaler Fixpunkte (s. Satz 4.2.11) — als

Suprema endlicher Approximationen dargestellt. Beweisen lassen sich im Rahmen von CPO-Modellierungen aber nur **zulässige Aussagen**, die ein unendliches Objekt immer dann erfüllt, wenn sie für dessen endliche Approximationen gelten (s. Def. 10.1.5). Die Dynamik eines Objektes betreffende Bedingungen sind aber häufig nicht von dieser Art. Insbesondere viele temporale Formeln (s. §9.1) beschreiben Aspekte des Verhalten "im Unendlichen", auf die man nicht aus dem Verhalten "im Endlichen" schließen kann.

Eine beliebte Klassifizierung dynamischer Eigenschaften liefert die Unterscheidung in **Sicherheitsbedingungen** (*safety conditions*) auf der eine Seite und **Lebendigkeitsbedingungen** (*liveness conditions*) auf der anderen Seite. Nach [14], S. 94, schließt eine Sicherheitsbedingung das Auftreten von etwas Schlechtem aus, während eine Lebendigkeitsbedingung das Auftreten von etwas Gutem garantiert. Eine Sicherheitsbedingung wird i.a. induktiv über dem Aufbau von Daten (Normalformen) definiert. Das "irgendwann" einer Lebendigkeitsbedingung deutet an, dass es sich hierbei oft um Eigenschaften unendlicher Prozesse handelt. Sicherheitsbedingungen sind noch eher zulässig im obigen Sinne als Lebendigkeitsbedingungen. Wie soll man z.B. feststellen, ob jede Folge von Zustandsänderungen eines dynamischen Objekts terminiert, wenn man nur endliche Teilfolgen betrachtet?

Geht man davon aus, dass sich alle relevanten Sicherheits- und Lebendigkeitsbedingungen als modallogische Formeln darstellen lassen, dann kann man sie auch prädikatenlogisch repräsentieren. Dazu werden das jeweils zugrundeliegende Transitionssystem $\mathcal{T} = (Z, Act, \rightarrow)$ als dreistelliges Prädikat und die modalen Operatoren als einstellige (Zustands-)Prädikate zweiter Ordnung einer Spezifikation mit Coprädikaten (siehe Def. 6.2.1) wiedergegeben. Modallogische Gültigkeitsaussagen $z \models \varphi$ ("φ gilt im Zustand z") lassen sich wie folgt in prädikatenlogische Formeln über einer solchen Spezifikation übersetzen:

$$
\begin{array}{lcl}
\mathit{compile}(z \models x) & = & r_x(z) \quad \text{für Zustandsmengenkonstanten oder -variablen } x \\
\mathit{compile}(z \models \varphi \wedge \psi) & = & \mathit{compile}(z \models \varphi) \wedge \mathit{compile}(z \models \psi) \\
\mathit{compile}(z \models \varphi \vee \psi) & = & \mathit{compile}(z \models \varphi) \vee \mathit{compile}(z \models \psi) \\
\mathit{compile}(z \models \langle M \rangle \varphi) & = & \exists z'(z \xrightarrow{M} z' \wedge \mathit{compile}(z' \models \varphi)) \\
\mathit{compile}(z \models [M] \varphi) & = & \forall z'(z \xrightarrow{M} z' \Rightarrow \mathit{compile}(z' \models \varphi)) \\
\mathit{compile}(z \models \mu x.(\varphi_1 \vee \cdots \vee \varphi_n)) & = & r_x(z), \text{ wobei } r_x \text{ durch die Hornaxiome}^{64} \\
 & & r_x(z) \Leftarrow \mathit{compile}(z \models \varphi_1), \ldots, r_x(z) \Leftarrow \mathit{compile}(z \models \varphi_n) \\
 & & \text{spezifiziert wird} \\
\mathit{compile}(z \models \nu x.(\varphi_1 \wedge \cdots \wedge \varphi_n)) & = & r_x(z), \text{ wobei } r_x \text{ durch die co-Hornaxiome} \\
 & & r_x(z) \Rightarrow \mathit{compile}(z \models \varphi_1), \ldots, r_x(z) \Rightarrow \mathit{compile}(z \models \varphi_n) \\
 & & \text{spezifiziert wird}
\end{array}
$$

Um andere modallogische Gültigkeitsaussagen zu übersetzen, transformiere man sie zunächst gemäß den Äquivalenzen von §9.1 und wende *compile* auf die Ergebnisse an.

Beispiel 9.2.1 Die folgende Spezifikation (in der Syntax von Expander2) enthält ein Transitionssystem mit der Zustandsmenge $\{1, 2, 3, 4\}$, der Aktionsmenge $\{a, b\}$, der Übergangsrelation *trans* (dargestellt als Adjazenzliste), der mit *compile* in Prädikate (X und Y) übersetzten modallogischen Formel

$$\nu x.(\mu y.(\langle a \rangle True \vee \langle b \rangle y) \wedge [b]x) \tag{1}$$

und den wichtigsten der in §9.1 definierten modalen Operatoren.

```
-- MODSIG

preds:      P Q R P' Q' R' true dia EF enabled EG Y
```

[64] Die Prämisse einer Hornformel und die Konklusion einer co-Hornformel können hier allgemeiner sein als in Def. 4.2.1 bzw. 6.2.1 gefordert.

```
copreds:    false box AG disabled AF X
defuncts:   states trans
fovars:     x y z st st' st1 st2 st3 st4        -- Variablen erster Ordnung
hovars:     P Q R                               -- Variablen hoeherer Ordnung

-- MODSPEZ

trans = [(2,"b",[1,3]),(3,"b",3),(3,"a",4),(4,"b",3)]

 & (X(st) ===> Y(st) & box("b")(X)(st))
 & (Y(st) <=== dia("a")(true)(st) | dia("b")(Y)(st))

 & true(st)
 & (dia(x)(P)(st) <=== (st,x,st') `in` trans & P(st'))
 & (EF(P)(st) <=== P(st))
 & (EF(P)(st) <=== (st,x,st') `in` trans & EF(P)(st'))
 & (enabled(st) <=== (st,x,st':_) `in` trans)
 & (EG(P)(st) <=== enabled(st) & P(st))
 & (EG(P)(st) <=== (st,x,st') `in` trans & P(st') & EG(P)(st'))
 & (false(st) ===> False)
 & (box(x)(P)(st) ===> ((st,x,st') `in` trans ==> P(st')))
 & (AG(P)(st) ===> P(st))
 & (AG(P)(st) ===> ((st,x,st') `in` trans ==> AF(P)(st')))
 & (disabled(st) ===> ((st,x,st') `in` trans ==> False))
 & (AF(P)(st) ===> (disabled(st) ==> P(st)))
 & (AF(P)(st) ===> ((st,x,st') `in` trans ==> P(st) | AF(P)(st')))
```

Wir beweisen, dass (1) in den Zuständen 3 und 4 gilt. Die entsprechende prädikatenlogische Formel lautet $X(3) \wedge X(4)$. Da X als Coprädikat spezifiziert ist, bietet sich ein Beweis durch Coinduktion an:

```
X(3)& X(4)

Coinduction w.r.t.  X(st5) ===> Y(st5) & box("b")(X)(st5)
applied to the preceding formula leads to the formula

All st5: (st5 = 3 | st5 = 4 ===> Y(st5) & box("b")(X0)(st5))

Simplification applied to the preceding formula leads to the formula

Y(3) & box("b")(X0)(3) & box("b")(X0)(4) & Y(4)

Simplification applied to the preceding formula leads to a new one.
The current factor is given by

Y(3)

Narrowing applied to the preceding formula leads to the factor

dia("a")(true)(3) | dia("b")(Y)(3)

Narrowing applied to the preceding formula leads to the factor

Any st'0: ((3,"a",st'0) `in` trans & true(st'0)) | dia("b")(Y)(3)

Narrowing applied to the preceding formula leads to the factor

true(4) | dia("b")(Y)(3)

Narrowing applied to the preceding formula leads to a new one.
The current factor is given by
```

```
box("b")(X0)(3)

Narrowing applied to the preceding formula leads to the factor

All st'1: ((3,"b",st'1) `in` trans ==> X0(st'1))

Narrowing applied to the preceding formula leads to the factor

X0(3)                                                          (A)

Narrowing applied to the preceding formula leads to a new one.
The current factor is given by

box("b")(X0)(4)

Narrowing applied to the preceding formula leads to the factor

All st'2: ((4,"b",st'2) `in` trans ==> X0(st'2))

Narrowing applied to the preceding formula leads to the factor

X0(3)                                                          (B)

Narrowing applied to the preceding formula leads to a new one.
The current formula is given by

Y(4)

Narrowing applied to the preceding formula leads to a new one.
The current summand is given by

dia("a")(true)(4)

Narrowing applied to the preceding formula leads to the summand

Any st'3: ((4,"a",st'3) `in` trans & true(st'3))

Narrowing applied to the preceding formula leads to a new one.
The current formula is given by

dia("b")(Y)(4)

Narrowing applied to the preceding formula leads to the formula

Any st'4: ((4,"b",st'4) `in` trans & Y(st'4))

Narrowing applied to the preceding formula leads to the formula

Y(3)

Narrowing applied to the preceding formula leads to a new one.
The current summand is given by

dia("a")(true)(3)

Narrowing applied to the preceding formula leads to the summand

Any st'5: ((3,"a",st'5) `in` trans & true(st'5))

Narrowing applied to the preceding formula leads to the summand

true(4)
```

```
Narrowing applied to the entire formula leads to True.
```

Das beim Coinduktionsschritt eingeführte Prädikat $X0$ ist durch die Hornformel $X0(z1) \Leftarrow (z1 = 3 \vee z1 = 4)$ spezifiziert. Käme X im Axiom für X nur an applizierbaren Positionen vor (wie z.B. in $X(st5)$), dann bräuchte $X0$ gar nicht erzeugt werden. Die Position von X in $box(b)(X)(st5)$ ist aber nicht applizierbar. Deshalb wird dort $X0$ eingesetzt und erst entfaltet, wenn eine applizierbare Position erreicht ist. Das geschieht bei (A) und (B). ✌

Um auch Zustandsäquivalenzen und andere **Verhaltensäquivalenzen** im Rahmen konstruktorbasierter Spezifikationen formulieren und beweisen zu können, erweitern wir Def. 5.1.2 nach der Einführung von Coprädikaten in Def. 6.2.1 ein zweites Mal:

Spezifikation mit versteckten Sorten

Definition 9.2.2 Sei $SP = (\Sigma, AX \cup coAX)$ eine Spezifikation mit Coprädikaten (s. Defs. 5.1.2 und 6.2.1). Sei $\Sigma = (S, F, R)$ und $S = visS \uplus hidS$. Die Sorten von $visS$ heissen **sichtbare**, die von $hidS$ **versteckte Sorten**. Für alle Konstruktoren $c : w \to s$ gelte: $s \in visS \Rightarrow w \in visS^*$, d.h. umgekehrt: Alle Konstruktoren mit versteckten Argumenten erzeugen versteckte Daten.

Außerdem gebe es für alle $s \in hidS$ unter den definierten Funktionen und Prädikaten eine Menge von **Destruktoren** $f : sw\{\to s'\}$, weiterhin unter den Prädikaten eine Menge von **Transitionsprädikaten** $\delta : sws'$ und schließlich unter den Coprädikaten eine **Verhaltensgleichheit** $\sim_s : ss$. Im Unterschied dazu heisst $\equiv_s$ fortan **Strukturgleichheit**. Logische Prädikate, die keine Transitionsprädikate sind, heissen **lokale Prädikate**.

Gemäß Def. 6.2.1 gehören die Axiome für ein Coprädikat zur Menge $coAX$ der co-Hornaxiome von SP. Axiome für Verhaltensgleichheiten heissen **Verhaltensaxiome** und lauten wie folgt: Sei $s \in hidS$.

$x \sim_s y \;\Rightarrow\; f(x,z) \sim f(y,z)$	für alle funktionalen Destruktoren $f : sw\{\to s'\}$,
$r(x,z) \;\Rightarrow\; (r(x,z) \;\Rightarrow\; r(y,z))$	für alle lokalen Prädikaten $r : sw$ unter den relationalen Destruktoren,
$x \sim_s y \;\Rightarrow\; (\delta(x,z,x') \;\Rightarrow\; \exists y'(\delta(y,z,y') \wedge x' \sim y'))$	und
$x \sim_s y \;\Rightarrow\; (\delta(y,z,y') \;\Rightarrow\; \exists x'(\delta(x,z,x') \wedge x' \sim y'))$	für alle Transitionsprädikate $\delta : sws'$ unter den relationalen Destruktoren.

Die starke Äquivalenz $\approx_{SP}$ (Def. 6.5.1), die extensionale Äquivalenz $\asymp_{SP}$ (Def. 6.7.2) und auch die Bisimilarität eines Transitionssystems $\mathcal{T}$ (Def. 9.1.3) lassen sich durch Verhaltensaxiome spezifizieren.

Verhaltensmodell

Definition 9.2.3 Sei $SP = (\Sigma, AX \cup coAX)$ eine Spezifikation mit versteckten Sorten. Die Interpretation der Verhaltensgleichheit im Herbrandmodell von SP heisst SP**-Verhaltensäquivalenz** und wird mit $\sim_{SP}$ bezeichnet. SP is **verhaltenskongruent**, wenn $\sim_{SP}$ mit allen Transitionsprädikaten von Σ zickzack-verträglich (Def. 6.5.3) und mit allen übrigen logischen Prädikaten und allen Funktionen von Σ verträglich ist. Unter dieser Bedingung ist der Quotient

$$Beh(SP) \quad =_{def} \quad Her(SP)/\sim_{SP}$$

ein Modell von SP und heisst **Verhaltensmodell von** SP.

Ist SP modal im Sinne von Def. 9.2.13, dann ist $Beh(SP)$ tatsächlich ein Modell von SP.

Da starke und extensionale Äquivalenz durch Verhaltensaxiome spezifiziert werden können, sind die Quotienten des Herbrandmodells nach diesen Äquivalenzen Verhaltensmodelle.

Beispiel 9.2.4

FLAG

`hidsorts`	$flag$
`constructs`	$new :\rightarrow flag$
	$up, down, rev : flag \rightarrow flag$
`destructs`	$up? : flag \rightarrow bool$
`vars`	$b : bool \quad x, y : flag$
`Horn axioms`	$up?(new) \equiv true$
	$up?(up(x)) \equiv true$
	$up?(down(x)) \equiv false$
	$up?(rev(x)) \equiv not(up?(x))$

☞ Zeigen Sie mithilfe der Coinduktionsregel aus §7.3, dass $rev(rev(x)) \sim x$ ein induktives Theorem von FLAG ist! ✌

Beispiel 9.2.5 (Mengen, Multimengen und Felder) Auch wenn nur endliche Objekte darzustellen sind, ist SP-Äquivalenz keine geeignete Modellierung der Identität *permutativer Daten*, weil solche Daten sich gerade dadurch auszeichnen, dass sie in der Regel mehrere semantisch äquivalente Normalformrepräsentationen haben. Von den jeweiligen Konstruktor*namen* einmal abgesehen, sind die Normalformen einer n-elementigen Menge, einer n-elementigen Multimengen oder eines Feldes n Einträgen nicht anderes als n-elementige Listen. Man könnte die Axiome von LIST (siehe Bsp. 6.3.2) um die folgenden Gleichungen zwischen Normalformen ergänzen und damit Listen zu (1) Mengen (*sets*), (2) Multimengen (*bags*) bzw. (3) Felder (*maps*) "abstrahieren":

$$
\begin{array}{lll}
(1) & x : y : L \equiv y : x : L & \\
 & x : x : L \equiv x : L & \\
(2) & x : y : L \equiv y : x : L & \\
(3) & (i, x) : (i, y) : L \equiv (i, x) : L & \\
 & (i, x) : (j, y) : L \equiv (j, y) : (i, x) : L & \Leftarrow \quad i \not\equiv j
\end{array}
$$

Bei (3) hat ":" die intuitive Bedeutung einer Update-Funktion: $(i, x) : L$ setzt x im Feld L an die Stelle i.

Normalformaxiome verletzen die Bedingung 5.1.2(2) an eine konstruktorbasierte Spezifikation SP sowie die Korrektheit fast aller wichtigen Beweisregeln (siehe Kapitel 7). Alle Versuche, das Beweisen *modulo Normalformgleichungen* zu automatisieren, sind wegen des hohen technischen Aufwandes entsprechender Verfahren mehr oder weniger gescheitert. Man kann auf Normalformaxiome wie (1)-(4) verzichten, weil sie Verhaltensäquivalenzen beschreiben, die sich auch im Rahmen einer Spezifikation mit versteckten Sorten axiomatisieren lassen. Zur Beschreibung von Feldern benötigen wir neben der Parameterspezifikation ENTRY (Bsp. 6.3.2) eine Parameterspezifikation INDEX für Feldindizes:

INDEX

`sorts`	$index$
`preds`	$\_ \not\equiv \_ : index \times index$
`vars`	$i, j : index$
`axioms`	$i \equiv j \ \vee \ i \not\equiv j$

$$\neg(i \equiv j \ \wedge \ i \not\equiv j)$$

PERMUT[TRIV(*entry*),TRIV(*index*)]

`vissorts`	$1 \quad 1 + entry$
`hidsorts`	$set = set(entry) \quad bag = bag(entry) \quad map = map(index, entry)$
`constructs`	$() :\rightarrow 1$
	$\emptyset :\rightarrow set$
	$insert : entry \times set \rightarrow set$
	$\_ \cup \_ : set \times set \rightarrow set$
	$\_ \cap \_ : set \times set \rightarrow set$
	$empty :\rightarrow bag$
	$add : entry \times bag \rightarrow bag$
	$new :\rightarrow map$
	$upd : index \times entry \times map \rightarrow map$ *"update"*
`destructs`	$\_ \in \_ : entry \times set \rightarrow bool$
	$freq : bag \times entry \rightarrow nat$ *"frequency"* (Häufigkeit eines Elementes im *bag*)
	$get : map \times index \rightarrow 1 + entry$
`copreds`	$\_ \subseteq \_ : set \times set$
`vars`	$x, y : entry \quad s, s' : set \quad b : bag \quad f : map \quad i, j : index$
`Horn axioms`	$x \in insert(y, s) \ \equiv \ eq(x, y) \ or \ (x \in s)$
	$x \in s \cup s' \ \equiv \ (x \in s) \ or \ (x \in s')$
	$x \in s \cap s' \ \equiv \ (x \in s) \ and \ (x \in s')$
	$freq(empty, x) \equiv 0$
	$freq(add(x, b), y) \equiv freq(b, y) + 1$
	$get(new, i) \equiv \kappa_1()$
	$get(upd(i, x, f), i) \equiv (x)$
	$get(upd(i, x, f), j) \equiv get(f, j) \quad \Leftarrow \quad i \not\equiv j$
`co-Horn axioms`	$s \subseteq s' \quad \Rightarrow \quad (x \in s \ \Rightarrow \ x \in s')$

☞ Wie lauten die Verhaltensaxiome von PERMUT (s. Def. 9.2.2)?

Die Behauptung, PERMUT spezifiziere Mengen, Multimengen und Felder, bedeutet präzise Folgendes: Seien A und A' Modelle der Parameterspezifikationen ENTRY bzw. INDEX und B das Verhaltensmodell der Aktualisierung von PERMUT durch A und A' (s. Def. 6.3.1). Dann ist

- B_{set} isomorph zur Menge aller endlichen Teilmengen von A,
- B_{bag} isomorph zur Menge aller endlichen Multimengen mit Elementen aus A,
- B_{map} isomorph zur Menge aller Funktionen von A' nach $A \cup \{\bot\}$ mit höchstens endlich vielen Werten $\neq \bot$.

☞ Zeigen Sie, dass die folgenden Atome induktive Theoreme von PERMUT sind:

$$\begin{aligned}
&insert(x, insert(y, s)) \ \sim \ insert(y, insert(x, s)) \\
&insert(x, insert(x, s)) \ \sim \ insert(x, s) \\
&add(x, add(y, b)) \ \sim \ add(y, add(x, b)) \\
&upd(i, x, upd(i, y, f)) \ \sim \ upd(i, x, f) \\
&upd(i, x, upd(j, y, f)) \ \sim \ upd(j, y, upd(i, x, f)) \quad \Leftarrow \quad i \not\equiv j
\end{aligned}$$

☞ Zeigen Sie, dass die PERMUT-Verhaltensäquivalenz mit den Konstruktoren (!) $\cup$ und $\cap$ verträglich ist!

☞ Erweitern Sie PERMUT um Konstruktoren zur Bildung von Normalformen, die unendliche Mengen, Multimengen bzw. Felder mit unendlich vielen Werten $\neq \bot$ bezeichnen! ✌

Beispiel 9.2.6 (Spezifikation von $\mathbb{Z}$) Auch die Gleichheit ganzer Zahlen läßt sich durch Normalformaxiome beschreiben:

$$(x+1)-1 \;\equiv\; x \qquad\qquad (x-1)+1 \;\equiv\; x$$

Auch diese Axiome werden zu Theoremen, wenn man $\equiv$ als Verhaltensgleichheit spezifiziert:

INT

`hidsorts`	int		
`constructs`	$0, 1 :\to int$		
	$\_ + \_ : int \times int \to int$		
	$\_ - \_ : int \times int \to int$		
`destructs`	$pred, succ : int \to int$		
	$zero : int$		
`vars`	$x, y : int$		
`Horn axioms`	$succ(0) \equiv 1$	$pred(0) \equiv 0 - 1$	$zero(0)$
	$succ(1) \equiv 1 + 1$	$pred(1) \equiv 0$	
	$succ(x + y) \equiv (x + y) + 1$	$pred(x + y) \equiv (x + y) - 1$	
	$succ(x - y) \equiv (x - y) + 1$	$pred(x - y) \equiv (x - y) - 1$	

Das Verhaltensmodell von INT ist isomorph zur Menge $\mathbb{Z}$ der ganzen Zahlen.

☞ Zeigen Sie, dass die o.a. Normalformaxiome induktive Theoreme von INT sind, wenn man $\equiv$ durch $\sim$ ersetzt!

☞ Da die Verhaltensäquivalenz die größte Lösung der Verhaltensaxiome im Herbrandmodell ist, wären alle int-Terme verhaltensäquivalent, wenn man den Destruktor $zero$ wegließe. Zeigen Sie das mit Coinduktion über $\sim$! ✌

Beispiel 9.2.7 (Semantik von IPF) Die bisher nicht formalisierte Semantik der imperativen Sprache IPF (s. Bsp. 3.3, Bsp. 3.8 und die Bemerkung im Anschluß an Bsp. 4.1.5) soll als Verhaltensmodell einer Spezifikation definiert werden. Dazu setzen wir eine Spezifikation INT Boolescher und ganzzahliger Arithmetik voraus, betrachten int und var als Untersorten von exp sowie $bool$ als Untersorte von $boolexp$ (s. §4.1) und benutzen PERMUT (Bsp. 9.2.5), um Speicherzustände als Felder mit var-Indizes und int-Einträgen darzustellen. Den Kern der Semantik von IPF liefern Axiome für die Transitionsrelation $\longrightarrow$ auf Speicherzuständen, was einer klassischen operationellen Semantik entspricht (s. §9.1).

IPF_EVAL = PERMUT$_{int/entry}$[INT,TRIV(var)] `then`

`vissorts`	$exp \;\; boolexp \;\; com$
`hidsorts`	$state = map(var, int)$
`constructs`	$\_ : int \to exp$
	$\_ : var \to exp$
	$\_ : bool \to exp$
	$\_ := \_ : var \times exp \to com$
	$\_? : boolexp \to com$
	$\_;\_ : com \times com \to com$
	$\_ + \_ : com \times com \to com$
	$\_^* : com \to com$
	$add : exp \times exp \to exp$

	$equal, greater : exp \times exp \to boolexp$
	$Not : boolexp \to boolexp$
	$And : boolexp \times boolexp \to boolexp$
	$\_ : \_ : state \times com \to state$
`defuncts`	$skip, fail :\to com$
	$if\_then\_else\_ : boolexp \times com \times com \to com$
	$while\_do\_ : boolexp \times com \to com$
	$repeat\_until\_ : com \times boolexp \to com$
	$eval : exp \to int$
	$eval : boolexp \to bool$
`transpreds`	$\_ \xrightarrow{-} \_ : state \times com \times state$
`copreds`	$\_ \approx \_ : com \times com$
`vars`	$s, s', s'' : state \;\; x : var \;\; e, e' : exp \;\; be, be' : boolexp \;\; c, c' : com \;\; i : int \;\; b : bool$
`Horn axioms`	$prog(new) \equiv skip$
	$prog(upd(x, i, s)) \equiv prog(s)$
	$prog(s : c) \equiv c; prog(s)$
	$get(s : c, x) \equiv get(s, x)$
	$s \xrightarrow{x:=e} upd(x, eval(s, e), s)$
	$s \xrightarrow{be?} s \;\; \Leftarrow \;\; eval(s, be) \equiv true$
	$s \xrightarrow{c;c'} s' \;\; \Leftarrow \;\; s \xrightarrow{c} s'' \wedge s'' \xrightarrow{c'} s'$
	$s \xrightarrow{c+c'} s' \;\; \Leftarrow \;\; s \xrightarrow{c} s'$
	$s \xrightarrow{c+c'} s' \;\; \Leftarrow \;\; s \xrightarrow{c'} s'$
	$s \xrightarrow{c^*} s$
	$s \xrightarrow{c^*} s' \;\; \Leftarrow \;\; s \xrightarrow{c;c^*} s'$
	$skip \equiv true?$
	$fail \equiv false?$
	$if\ be\ then\ c\ else\ c' \;\equiv\; (be?; c) + (Not(be)?; c')$
	$while\ be\ do\ c \;\equiv\; (be?; c)^*; Not(be)?$
	$repeat\ c\ until\ be \;\equiv\; c; while\ Not(be)\ do\ c$
	$eval(s, x) \equiv get(s, x)$
	$eval(s, i) \equiv i$
	$eval(s, add(e, e')) \equiv eval(s, e) + eval(s, e')$
	$eval(s, b) \equiv b$
	$eval(s, equal(e, e')) \equiv true \;\; \Leftarrow \;\; eval(s, e) \equiv eval(s, e')$
	$eval(s, equal(e, e')) \equiv false \;\; \Leftarrow \;\; eval(s, e) \not\equiv eval(s, e')$
	$eval(s, greater(e, e')) \equiv true \;\; \Leftarrow \;\; eval(s, e) > eval(s, e')$
	$eval(s, greater(e, e')) \equiv false \;\; \Leftarrow \;\; eval(s, e) \leq eval(s, e')$
	$eval(s, Not(be)) \equiv not(eval(s, be))$
	$eval(s, And(be, be')) \equiv (eval(s, be)\ and\ eval(s, be'))$
`co-Horn axioms`	$c \approx c' \;\; \Rightarrow \;\; (s \xrightarrow{c} s' \;\Rightarrow\; \exists\, s'' : (s \xrightarrow{c'} s'' \wedge s' \sim s''))$
	$c \approx c' \;\; \Rightarrow \;\; (s \xrightarrow{c'} s' \;\Rightarrow\; \exists\, s'' : (s \xrightarrow{c} s'' \wedge s' \sim s''))$

Zusicherungen (s. §§8.1 und 9.2) lassen sich als prädikatenlogische Formeln über IPF_EVAL darstellen. Seien s und s' Variablen der Sorte *state* und sei c ein IPF-Programm (= Grundterm der Sorte *com*). Dann beschreibt

$$pre(s) \wedge s \xrightarrow{c} s' \;\; \Rightarrow \;\; post(s')$$

eine Ein/Ausgabe-Relation von c: Gilt die **Vorbedingung** *pre* im Zustand s und überführt c den Zustand s in

den Zustand s', dann erfüllt s' die **Nachbedingung** *post*.

Die in Def. 10.2.5 im Rahmen der CPO-Semantik definierte **schwächste** bzw. **schwächste liberale Vorbedingung von** $(c, post)$ läßt sich als IPF_EVAL-Prädikat durch die Hornaxiome

$$wp(s) \Leftarrow \exists\, s' : (s \xrightarrow{c} s' \wedge wlp(s)) \quad \text{bzw.} \quad wlp(s) \Leftarrow \forall\, s' : (s \xrightarrow{c} s' \Rightarrow post(s'))$$

spezifizieren. Die Regeln des **Hoare-Kalküls** (Def. 10.2.1) zum Nachweis von Zusicherungen entsprechen folgenden bzgl. des initialen Modells von IPF_EVAL korrekten Expansionsregeln :

Zuweisungsregel
$$\frac{pre(s) \wedge s \xrightarrow{x:=e} s' \quad \Rightarrow \quad post(s')}{pre(s) \quad \Rightarrow \quad post(s)[e/x]} \Uparrow$$

Sequenzregel
$$\frac{pre(s) \wedge s \xrightarrow{c;c'} s' \quad \Rightarrow \quad post(s')}{pre(s) \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad q(s'), \qquad q(s) \wedge s \xrightarrow{c'} s' \quad \Rightarrow \quad post(s')} \Uparrow$$

Verzweigungsregel
$$\frac{pre(s) \wedge s \xrightarrow{if\ be\ then\ c\ else\ c'} s' \quad \Rightarrow \quad post(s')}{\begin{array}{l} pre(s) \wedge eval(s, be) = true \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad post(s'), \\ pre(s) \wedge eval(s, be) = false \wedge s \xrightarrow{c'} s' \quad \Rightarrow \quad post(s') \end{array}} \Uparrow$$

Schleifenregel
$$\frac{pre(s) \wedge s \xrightarrow{while\ be\ do\ c} s' \quad \Rightarrow \quad post(s')}{\begin{array}{c} pre(s) \quad \Rightarrow \quad inv(s), \\ inv(s) \wedge eval(s, be) = true \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad inv(s'), \\ inv(s) \wedge eval(s, be) = false \quad \Rightarrow \quad post(s) \end{array}} \Uparrow$$

inv entspricht der in §7.5 im Rahmen der Verifikation funktionaler iterativer Programme behandelten Hoare-Invariante. Wie die folgende Regel zeigt, kann *inv* auch den Nachweis der Termination einer Schleife unterstützen:

Terminationsregel
$$\frac{pre(s) \quad \Rightarrow \quad s \xrightarrow{while\ be\ do\ c} s'}{\begin{array}{c} pre(s) \quad \Rightarrow \quad inv(s), \\ inv(s) \wedge eval(s, be) = true \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad s \gg s' \wedge inv(s') \end{array}} \Uparrow$$

falls $\ll$: $state \times state$ im initialen Modell von IPF_EVAL eine wohlfundierte Interpretation hat

☞ Zeigen Sie die Korrektheit des IPF-Programms

$$c \quad = \quad z := 1;\ while\ y > 0\ do\ (z := z * y;\ y := y - 1)$$

zur Berechnung der Fakultätsfunktion! Expandieren Sie zu diesem Zweck die Zusicherung

$$get(s, y) \equiv x \wedge x \geq 0 \wedge s \xrightarrow{c} s' \quad \Rightarrow \quad get(s', z) \equiv x!$$

mit der Zuweisungs-, Sequenz- und Schleifenregel! Die bei der Anwendung der Schleifenregel einzusetzende Invariante lautet:

$$x! \equiv get(s, y)! * get(s, z)$$

(vgl. Bsp. 7.5.3). ✌

Beispiel 9.2.8 (Kleiner Bankverkehr) An folgendem Beispiel wurden mehrere formale Ansätze objektorientierter Spezifikation veranschaulicht (vgl. [34], [67]). Die zwischen Objekten austauschbaren Nachrichten

bilden die Aktionen der von objektorientierten Programmen erzeugten Transitionssysteme. Intuitiv gesprochen, beschreiben diese das Laufzeitverhalten der Programme. Wir spezifizieren die Dynamik des Bankverkehrs analog zu IPF_EVAL (Bsp. 9.2.7). Anstelle von Speicherzuständen gibt es hier (Mengen von) Kontoobjekte(n), anstelle von Kommandos *Methoden*sequenzen, die von einzelnen Objekten aufgerufen werden können. Die Destruktoren *name*, *bal*(*ance*) und *record* bestimmen den jeweils aktuellen Zustand eines Kontoobjektes: Zwei Objekte sind verhaltensäquivalent, wenn sie dieselben *name*-, *bal*- und *record*-Werte haben. Während das Transitionsprädikat $\longrightarrow$: $acc \times com \times acc$ die möglichen Zustandsübergange einzelner Objekte festlegt, setzt $\Longrightarrow$: $set(acc) \times com \times set(acc)$ jenes Prädikat auf Objekt*mengen* fort. Damit läßt sich dann auch die Kommunikation zwischen Objekten beschreiben. Zur Verhaltensäquivalenz von Mengen vgl. Bsp. 9.2.5).

TRANSACTION = NAT and
`vissorts` $entry \quad transaction$
`constructs` $from, to : entry \times nat \to transaction$

ACC_STATE = $\text{LIST}_{transaction/s}$[TRANSACTION] `then`

`hidsorts`	acc
`constructs`	$new : entry \to acc$
	$credit, debit : acc \times nat \times entry \to acc$
`destructs`	$name : acc \to entry$
	$bal : acc \to nat$
	$record : acc \to list(transaction)$
`vars`	$x : entry \quad n : nat \quad a : acc$
`Horn axioms`	$name(new(x)) \equiv x$
	$bal(new(x)) \equiv 0$
	$record(new(x)) \equiv nil$
	$name(credit(a, n, x)) \equiv name(a)$
	$bal(credit(a, n, x)) \equiv bal(a) + n$
	$record(credit(a, n, x)) \equiv from(x, n) : record(a)$
	$name(debit(a, n, x)) \equiv name(a)$
	$bal(debit(a, n, x)) \equiv bal(a) - n$
	$record(debit(a, n, x)) \equiv to(x, n) : record(a)$

ACC_LOCAL = ACC_STATE `then`

`vissorts`	com
`constructs`	$deposit, withdraw : nat \to com$
	$send, receive : nat \times entry \to com$
	$bal? : nat \to com$
`transpreds`	$\_ \stackrel{-}{\longrightarrow} \_ : acc \times com \times acc$
`vars`	$x : entry \quad n : nat \quad a : acc \quad c : com$
`Horn axioms`	$a \stackrel{deposit(n)}{\longrightarrow} credit(a, n, name(a))$
	$a \stackrel{withdraw(n)}{\longrightarrow} debit(a, n, name(a)) \quad \Leftarrow \quad n \leq bal(a)$
	$a \stackrel{send(n,x)}{\longrightarrow} debit(a, n, x) \quad \Leftarrow \quad n \leq bal(a)$
	$a \stackrel{receive(n,x)}{\longrightarrow} credit(a, n, x)$
	$a \stackrel{bal?(n)}{\longrightarrow} a \quad \Leftarrow \quad n \equiv bal(a)$

ACC_GLOBAL = ACC_LOCAL `and` PERMUT[$entry \mapsto acc$] `then`

`constructs`	$open, close : entry \to com$
	$\_.\_ : entry \times com \to com$

	$\_;\_ : com \times com \to com$
`defuncts`	$transfer : entry \times nat \times entry \to com$
`transpreds`	$\_ \Longrightarrow \_ : set(acc) \times com \times set(acc)$
`preds`	$\_ \notin \_ : entry \times set(acc)$
`copreds`	$\_ \approx \_ : com \times com$
`vars`	$x, y : entry \quad a, a' : acc \quad as, bs, as', bs' : set(acc) \quad c, c' : com$
`Horn axioms`	$as \overset{open(x)}{\Longrightarrow} as \cup \{new(x)\} \quad \Leftarrow \quad x \notin as$
	$as \overset{close(x)}{\Longrightarrow} as \setminus \{a\} \quad \Leftarrow \quad name(a) \equiv x$
	$as \overset{x.c}{\Longrightarrow} as \setminus \{a\} \cup \{a'\} \quad \Leftarrow \quad a \xrightarrow{c} a' \wedge a \in as \wedge name(a) \equiv x$
	$as \overset{c;c'}{\Longrightarrow} as' \quad \Leftarrow \quad as \overset{c}{\Longrightarrow} bs \wedge bs \overset{c'}{\Longrightarrow} as'$
	$transfer(x, n, y) \equiv x.send(n, y);\ y.receive(n, x)$
	$x \notin as \quad \Leftarrow \quad name(a) \equiv x \wedge a \notin as$
`co-Horn axioms`	$c \approx c' \quad \Rightarrow \quad (as \overset{c}{\Longrightarrow} bs \ \Rightarrow \ \exists\, bs' : (as \overset{c'}{\Longrightarrow} bs' \wedge bs \sim bs'))$
	$c \approx c' \quad \Rightarrow \quad (as \overset{c'}{\Longrightarrow} bs' \ \Rightarrow \ \exists\, bs : (as \overset{c}{\Longrightarrow} bs \wedge bs \sim bs'))$

Kontext

Definition 9.2.9 Ein Symbol ist **sichtbar** bzw. **versteckt**, wenn alle seine Argument- und Wertesorten sichtbar sind bzw. mindestens eine dieser Sorten versteckt ist. Ein Term ist **sichtbar** bzw. **versteckt**, wenn er eine sichtbare bzw. versteckte Sorte hat. Ein sichtbarer Term, der genau eine Variable einer versteckten Sorte s enthält, heisst **s-Kontext**. Die **kontextuelle SP-Äquivalenz** $\sim^c_{SP}$ auf T_Σ stimmt auf sichtbaren Termen mit $\equiv_{SP}$ überein und ist auf versteckten Termen wie folgt definiert: Seien $t, t' \in T_{\Sigma,s}$ und $x \in X_s$.

$$t \sim^c_{SP} t' \quad \Longleftrightarrow_{def} \quad u[t/x] \equiv_{SP} u[t'/x] \text{ für alle } s\text{-Kontexte } u \in T_\Sigma(\{x\}).$$

$\sim^c_{SP}$ stimmt mit der SP-Verhaltensäquivalenz überein, wenn SP wie im folgenden Beispiel keine Transitionsprädikate enthält.

Beispiel 9.2.10 Wir spezifizieren den Datentyp *unendlicher* Ströme:

STREAM[TRIV($entry$)] = LIST[TRIV($entry$)] `and` NAT `then`

`hidsorts`	$stream(entry)$
`constructs`	$\_\&\_ : entry \times stream(entry) \to stream(entry)$
	$odds : stream \to stream(entry)$
	$zip : stream(entry) \times stream(entry) \to stream(entry)$
	$map : (entry \to entry) \times stream(entry) \to stream(entry)$
`destructs`	$head : stream(entry) \to entry$
	$tail : stream(entry) \to stream(entry)$
`defuncts`	$\_\#\_ : list(entry) \times stream(entry) \to stream(entry)$
	$evens : stream(entry) \to stream(entry)$
	$firstn : nat \times stream(entry) \to list(entry)$
	$nthtail : nat \times stream(entry) \to stream(entry)$
`preds`	$exists : (entry \to bool) \times stream(entry)$
`copreds`	$forall : (entry \to bool) \times stream(entry)$
	$fair : (entry \to bool) \times stream(entry)$
`vars`	$m, n : nat \quad x, y : entry \quad L : list(entry) \quad s, s' : stream(entry)$

	$f : entry \to entry \quad g : entry \to bool$	
`Horn axioms`	$head(x\&s) \equiv x$	$tail(x\&s) \equiv s$
	$head(zip(s, s')) \equiv head(s)$	$tail(zip(s, s')) \equiv zip(s', tail(s))$
	$head(odds(s)) \equiv head(s)$	$tail(odds(s)) \equiv odds(tail(tail(s))$
	$head(map(f, s)) \equiv f(s)$	$tail(map(f, s)) \equiv map(f, tail(s))$
	$nil\#s \equiv s$	
	$(x : L)\#s \equiv x\&(L\#s)$	
	$evens(s) \equiv odds(tail(s))$	
	$firstn(0, s) \equiv nil$	
	$firstn(n + 1, s) \equiv head(s) : firstn(n, tail(s))$	
	$nthtail(0, s) \equiv s$	
	$nthtail(n + 1, s) \equiv tail(nthtail(n, s))$	
	$exists(g, s) \Leftarrow g(head(s)) \equiv true$	
	$exists(g, s) \Leftarrow exists(g, tail(s))$	
`co-Horn axioms`	$forall(g, s) \Rightarrow g(head(s)) \equiv true \wedge forall(g, tail(s))$	
	$fair(g, s) \Rightarrow exists(g, s) \wedge fair(g, tail(s))$	

NATSTREAM = $\text{STREAM}_{nat/entry}$[NAT] `then`

`constructs`	$blink :\to stream(nat)$	
	$nats : nat \to stream(nat)$	
`Horn axioms`	$head(blink) \equiv 0$	$tail(blink) \equiv 1\&blink$
	$head(nats(n)) \equiv n$	$tail(nats(n)) \equiv nats(n + 1)$

Im Verhaltensmodell von STREAM entspricht der Konstruktor & dem Konstruktor : endlicher Listen. *blink* bezeichnet den Strom, der, beginnend mit 0, abwechselnd Nullen und Einsen enthält. $nats(n)$ erzeugt den Strom aller natürlichen Zahlen, beginnend mit n. $odds(s)$ und $evens(s)$ liefern die Ströme der Elemente von s an ungeradzahligen bzw. geradzahligen Positionen. *zip* mischt zwei Ströme durch alternatives Anfügen von Elementen des einen bzw. des anderen Stroms. # konkateniert eine Liste und einen Strom zu einem neuen Strom. *head*, *tail*, $firstn$, *nthtail*, *map*, *exists* and $forall$ haben die gleiche Bedeutung die entsprechenden Funktionen auf endlichen Listen. $fair(g, s)$ gilt genau dann, wenn g für unendlich viele Elemente von s gilt.

Jeder *stream*-Kontext hat die Form $head(tail(\ldots(tail(s))\ldots))$. Demnach gilt für alle $t, t' \in T_{\Sigma,stream}$,

$$t \sim_{\text{STREAM}} t' \quad \Longleftrightarrow \quad \forall i \in \mathbb{N} : head(tail^i(t)) \equiv_{\text{STREAM}} head(tail^i(t')). \tag{1}$$

☞ Zeigen Sie (1) durch Coinduktion (s. §7.3)! ✌

So wie das Verhaltensmodell von PERMUT (s. 9.2.5) nicht alle Mengen, Multimengen und Felder von Elementen der Sorte *entry* enthält, so sind im Verhaltensmodell von STREAM nicht alle unendlichen Ströme durch Grundterme repräsentiert. Mit zusätzlichen Konstruktoren können weitere Ströme eingeführt werden, niemals jedoch alle, denn die Menge der Ströme von Elementen aus einer Menge A ist überabzählbar, schon wenn A nur zwei Elemente enthält. Das übliche Modell aller unendlichen Ströme von Elementen aus A ist die Menge $[\mathbb{N} \to A]$ der Funktionen von $\mathbb{N}$ nach A, zusammen mit geeigneten Interpretationen der Signatursymbole von STREAM. Man kann das Verhaltensmodell von STREAM in $[\mathbb{N} \to A]$ einbetten, d.h. es gibt einen injektiven Homomorphismus

$$h : Beh(\text{STREAM}) \to [\mathbb{N} \to A].$$

Aus (1) folgt nämlich, dass h mit

$$h([t])(i) \quad =_{def} \quad head(tail^i(t))^A$$

für alle $i \in \mathbb{N}$ wohldefiniert ist.

Beispiel 9.2.11 Im Gegensatz zu STREAM (s. Bsp. 9.2.10) werden in der folgenden Spezifikation auch endliche Ströme durch die Sorte $stream(entry)$ repräsentiert. Das ergibt sich aus der Ersetzung der Destruktoren *head* and *tail* von STREAM durch das Transitionsprädikat $\longrightarrow$: $stream(entry) \times entry \times stream(entry)$ (s. Def. 9.2.2).

RSTREAM[TRIV($entry$)] = LIST[TRIV($entry$)] `and` NAT `then`

`hidsorts` $stream(entry)$

`constructs`
$\_\&\_ : entry \times stream(entry) \to stream(entry)$
$empty :\to stream(entry)$
$odds, evens : stream(entry) \to stream(entry)$
$zip : stream \times stream(entry) \to stream(entry)$
$map : (entry \to entry) \times stream(entry) \to stream(entry)$
$filter : (entry \to bool) \times stream(entry) \to stream(entry)$

`destructs`
$disabled : stream(entry)$
$\_ \xrightarrow{\_} \_ : stream(entry) \times entry \times stream(entry)$

`preds`
$enabled, finite : stream(entry)$
$first : (entry \to bool) \times stream(entry) \times nat$
$exists : (entry \to bool) \times stream(entry)$
$length : stream(entry) \times nat$

`copreds`
$fair : (entry \to bool) \times stream(entry)$
$infinite : stream(entry)$
$forall : (entry \to bool) \times stream(entry)$

`vars`
$n : nat \quad x, y : entry \quad L : list(entry) \quad s, s', t, t' : stream(entry)$
$f : entry \to entry \quad g : entry \to bool$

`Horn axioms`
$x\&s \xrightarrow{x} s$
$odds(s) \xrightarrow{x} odds(t) \Leftarrow s \xrightarrow{x} s' \wedge s' \xrightarrow{y} t$
$evens(s) \xrightarrow{x} odds(t) \Leftarrow s \xrightarrow{y} s' \wedge s' \xrightarrow{x} t$
$s@s' \xrightarrow{x} t@s' \Leftarrow s \xrightarrow{x} t$
$s@s' \xrightarrow{x} s@t \Leftarrow disabled(s) \wedge s' \xrightarrow{x} t$
$zip(s, s') \xrightarrow{x} zip(s', t) \Leftarrow s \xrightarrow{x} t$
$zip(s, s') \xrightarrow{x} zip(s, t) \Leftarrow disabled(s) \wedge s' \xrightarrow{x} t$
$map(f, s) \xrightarrow{f(x)} map(f, t) \Leftarrow s \xrightarrow{x} t$
$filter(g, s) \xrightarrow{x} filter(g, t) \Leftarrow s \xrightarrow{x} t \wedge g(x) \equiv true$
$filter(g, s) \xrightarrow{y} t' \Leftarrow s \xrightarrow{x} t \wedge g(x) \equiv false \wedge filter(g, t) \xrightarrow{y} t'$
$enabled(s) \Leftarrow s \xrightarrow{x} t$
$disabled(empty)$
$disabled(odds(s)) \Leftarrow disabled(s)$
$disabled(evens(s)) \Leftarrow disabled(s)$
$disabled(s@s') \Leftarrow disabled(s) \wedge disabled(s')$
$disabled(zip(s, s')) \Leftarrow disabled(s) \wedge disabled(s')$
$disabled(map(f, s)) \Leftarrow disabled(s)$
$disabled(filter(g, s)) \Leftarrow disabled(s)$
$disabled(filter(g, s)) \Leftarrow s \xrightarrow{x} t \wedge g(x) \equiv false \wedge disabled(filter(g, t))$
$length(s, 0) \Leftarrow disabled(s)$
$length(s, n + 1) \Leftarrow s \xrightarrow{x} t \wedge length(t, n)$
$finite(s) \Leftarrow length(s, n)$
$first(g, s, 0) \Leftarrow s \xrightarrow{x} t \wedge g(x) \equiv true$
$first(g, s, n + 1) \Leftarrow s \xrightarrow{x} t \wedge g(x) \equiv false \wedge first(g, t, n)$

$$
\begin{array}{ll}
 & exists(g,s) \;\Leftarrow\; first(g,s,n) \\
\texttt{co-Horn axioms} & infinite(s) \;\Rightarrow\; \exists x,t : (s \xrightarrow{x} t \wedge infinite(t)) \\
 & forall(g,s) \;\Rightarrow\; (s \xrightarrow{x} t \;\Rightarrow\; (g(x) \equiv true \wedge forall(g,t))) \\
 & fair(g,s) \;\Rightarrow\; exists(g,s) \\
 & fair(g,s) \;\Rightarrow\; (s \xrightarrow{x} t \;\Rightarrow\; fair(g,t))
\end{array}
$$

$$
\begin{array}{lll}
\text{NATRSTREAM} = \text{RSTREAM}_{nat/entry}[\text{NAT}] \texttt{ then} & & \\
\quad \texttt{constructs} & blink :\rightarrow stream(nat) & \\
 & nats : nat \rightarrow stream(nat) & \\
\quad \texttt{Horn axioms} & blink \xrightarrow{0} 1\&blink & \\
 & nats(n) \xrightarrow{n} nats(n+1) &
\end{array}
$$

Im Verhaltensmodell von RSTREAM gilt $s \xrightarrow{x} t$ genau dann, wenn x das erste Element und t der Rest von s ist. $filter(g,s)$ entfernt alle Elemente aus s, die g nicht erfüllen. Die Prädikate *disabled* und *enabled* sowie $finite$ und $infinite$ grenzen leere von nichtleeren bzw. endliche von unendlichen Strömen ab. $first(g,s,n)$ gilt genau dann, wenn das $(n+1)$-te Element von s die Bedingung g erfüllt. $length(s,n)$ gilt genau dann, wenn s aus n Elementen besteht. Die anderen Signatursymbole werden wie die gleichlautenden Symbole von STREAM interpretiert (s. Bsp. 9.2.10). ✌

Aus den Verhaltensaxiomen von SP folgt noch nicht, dass SP verhaltenskongruent ist und damit das Verhaltensmodell $Beh(SP)$ existiert (s. Def. 9.2.3). Selbst wenn das der Fall ist, kann es noch sein, dass $Beh(SP)$ kein Modell von SP ist. Enthält SP weder versteckte Sorten noch Coprädikate, dann ist nach Satz **??** die Regularität von SP hinreichend, um beide Anforderungen sicherzustellen. Andernfalls muss SP weitere Bedingungen erfüllen. Diese Bedingungen sind zwar weitgehend syntaktisch, aber zum Teil sehr technisch. Sie ergeben sich aus einer genauen Analyse von Ergebnissen der Modallogik, Prozessalgebra (s. §9.1) und der Theorie der Coalgebren (s.u.).

verhaltensinvariante Formel

Definition 9.2.12 Sei $SP = (\Sigma, AX)$ eine Spezifikation mit versteckten Sorten. Eine Formel $\varphi(x)$ mit der freien Variable x ist **verhaltensinvariant**, wenn für alle $t, t' \in T_\Sigma$ gilt:

$$Her(SP) \;\models\; \varphi(t) \wedge t \sim t' \quad\Rightarrow\quad Her(SP) \;\models\; \varphi(t').$$

Wäre $\sim_{SP}$ eine prädikatenverträgliche Kongruenzrelation (s. Def. 4.1.2), dann wäre jede prädikatenlogische Σ-Formel verhaltensinvariant. In der Regel ist $\sim_{SP}$ aber nur verhaltenskongruent (s. Def. 9.2.3) und demzufolge kann die Gültigkeit einer Formel verletzt werden, wenn einer ihrer Teilterme durch einen verhaltensäquivalenten Term ersetzt wird. Ist die Formel verhaltensinvariant, dann kann das nicht passieren. Die folgende Definition liefert u.a. syntaktische Bedingungen für Verhaltensinvarianz.

modale Spezifikation

Definition 9.2.13 Eine Formel ist **poly-modal**, wenn sie zu einer mit folgenden Regeln aufgebauten Formel äquivalent ist:

- Ein Atom $r(t)$ mit lokalem Prädikat r ist poly-modal.
- Sind φ und ψ poly-modal, dann sind $\neg\varphi$, $\varphi \wedge \psi$ und $\exists x\varphi$ für alle $x \in X$ poly-modal.
- Ist $\delta(t,x)$ ein Atom mit Transitionsprädikat δ, $x \in X \setminus var(t)$ und φ poly-modal, dann ist $\exists x(\delta(t,x) \wedge \varphi)$ poly-modal.

Eine Formel φ ist **schwach modal mit Ausgabe** $out(\varphi) \subseteq X$, wenn sie zu einer mit folgenden Regeln aufgebauten Formel äquivalent ist:

- Eine poly-modale Formel ist schwach modal mit Ausgabe $\emptyset$.
- Ein Atom $\delta(t, x)$ mit Transitionsprädikat δ und $x \in X \setminus var(t)$ ist schwach modal mit Ausgabe $\{x\}$.
- Sind φ und ψ schwach modal mit disjunkten Ausgaben Y bzw. Z, dann ist $\varphi \wedge \psi$ schwach modal mit Ausgabe $Y \cup Z$.
- Ist φ schwach modal mit Ausgabe Y und $x \in X$, dann ist $\exists x \varphi$ schwach modal mit Ausgabe $Y \setminus \{x\}$.

Eine Spezifikation SP ist **modal**, wenn

- für alle Hornaxiome $r(t) \Leftarrow \varphi$ für ein lokales Prädikat r und alle Hornaxiome $\delta(t, u) \Leftarrow \varphi$ für ein Transitionsprädikat $\delta : ws$, φ schwach modal mit einer von $var(t)$ disjunkten Ausgabe ist,
- für alle co-Hornaxiome $p \Rightarrow \varphi$, φ poly-modal ist.

Der Begriff "modale Formel" wurde gewählt, weil es sich hier u.a. um diejenigen prädikatenlogischen Formeln handelt, die entstehen, wenn modallogische Aussagen mit der oben definierten Funktion *compile* übersetzt werden.

Satz 9.2.14 ([85], Theorem 3.8(3)) Sei $SP = (\Sigma, AX)$ eine verhaltenskongruente Spezifikation (s. Def. 9.2.3). Dann sind alle poly-modalen Σ-Formeln verhaltensinvariant. ❑

Daraus ergibt sich die Korrektheit folgender Regeln: Seien φ und ψ polymodale Formeln.

Entfernung von Verhaltensgleichungen $\dfrac{t \sim u \wedge \varphi(t)}{\varphi(u)} \Downarrow$ $\qquad$ $\dfrac{(t \sim u \wedge \varphi(t)) \;\Rightarrow\; \psi(t)}{\varphi(u) \;\Rightarrow\; \psi(u)} \Uparrow$

bild-endliche Spezifikation

Definition 9.2.15 Sei G ein Σ-Goal.

$$\mathcal{S}(G) \;=_{def}\; \{\tau : var(G) \to NF_\Sigma \mid Her(SP) \models G\tau\}$$

ist die Menge der **Normalform-Lösungen von** G. Eine Goalmenge $\exists X_1 G_1 \vee \ldots \exists X_n G_n$ ist **bild-endlich**, wenn für alle $1 \leq i \leq n$ G_i ein Goal H ohne Coprädikate enthält derart, dass H mit G_i übereinstimmt oder nichtleer ist und für alle $\sigma : X \to NF_\Sigma$ die Menge der Normalform-Lösungen von $H\sigma_{X \setminus X_i}$ endlich ist. Eine Spezifikation $SP = (\Sigma, AX \cup coAX)$ ist **bild-endlich**, wenn für alle $p \Rightarrow (G \Rightarrow \varphi) \in coAX$, φ bild-endlich ist.

Der Begriff "bild-endlich" ist durch die speziellen Goals der Form $\exists y : x \to y \wedge H(y)$ motiviert, die im Sinne von Def. 9.2.15 gerade dann bild-endlich sind, wenn das $\to$ bild-endlich ist, d.h. für alle (Grundterme) t höchstens endlich viele u mit $t \to u$ existieren.

Alle Beispielspezifikationen dieses Abschnitts sind bild-endlich. Die Konklusion des folgenden Axioms für das Prädikat $fair$ von STREAM ist nicht bild-endlich:

$$fair(g, s) \;\Rightarrow\; \exists n, s' : (nthtail(n, s) \equiv s' \wedge head(s') \equiv x \wedge g(x) \equiv true \wedge fair(g, tail(s'))).$$

Mit den Bezeichnungen von Def. 9.2.15 ist hier $n = 1$ und $X_1 = \{n, s'\}$. H muss ein Teilgoal von $nthtail(n, s) \equiv s' \wedge head(s') \equiv x \wedge g(x) \equiv true$ sein. Es gibt aber $\sigma : X \to NF_\Sigma$ derart, dass $H\sigma_{X\setminus X_i} = H[\sigma(s)/s]$ unendlich viele Normalform-Lösungen hat. Erweitert man jedoch H um das Atom $forall(not \circ g, firstn(n, s))$, dann wird das Axiom bild-endlich, weil dann $head(s')$ das erste Element of s ist, das g erfüllt, so dass $H[\sigma(s)/s]$ nur noch höchstens eine Normalform-Lösung hat.

coinduktive Spezifikation

Definition 9.2.16 Sei $SP = (\Sigma, AX)$ eine Spezifikation mit versteckten Sorten. $t \in T_\Sigma(X)^+$ ist **objekt-normal**, wenn t aus sichtbaren Normalformen und versteckten Variablen besteht. Ein Atom $\delta(t, a, u)$ ist **beobachtend**, wenn δ ein Transitionsprädikat ist oder $\delta(t, a, u) = (f(t, a)\{\equiv u\})$ und f ein Destruktor ist. Ein Atom $\delta(t, u)$ ist **nicht-beobachtend**, wenn δ ein Transitionsprädikat, aber kein Destruktor ist oder $\delta(t, a, u) = (f(t)\{\equiv u\})$ und f eine definierte Funktion oder ein lokales Prädikat, aber kein Destruktor ist. Ein Goal ist **nicht-beobachtend**, wenn es aus nicht-beobachtenden Atomen besteht. Eine Hornformel $p \Leftarrow \varphi$ ist **coinduktiv**, wenn entweder $p = \delta(t, u)$ nicht-beobachtend und t objekt-normal ist oder $\delta(t, a, u)$ beobachtend ist,

$$\varphi \quad = \quad G \wedge \delta_1(t_1, a_1, u_1) \wedge G_1 \wedge \cdots \wedge \delta_n(t_n, a_n, u_n) \wedge G_n$$

und die folgenden Bedingungen gelten: Sei $V_0 = var(t, a, G)$ und für alle $1 \leq i \leq n$, $V_i = V_{i-1} \cup var(a_i, u_i, G_i)$.

(1) t ist objekt-normal oder es gibt einen Konstruktor c und eine Objekt-Normalform t' mit $t = c(t')$, a ist objekt-normal, G ist schwach modal (s. Def. 9.2.13) und nicht-beobachtend, $var(u) \subseteq V_n$ und $out(G) \cap var(t, a) = \emptyset$.

(2) Für alle $1 \leq i \leq n$ ist $\delta_i(t_i, a_i, u_i)$ beobachtend, (t_i, a_i) normal, u_i objekt-normal, G_i schwach modal und nicht-beobachtend, $var(t_i) \subseteq V_{i-1}$, $(var(u_i) \cup out(G_i)) \cap (V_{i-1} \cup var(a_i, u_i)) = \emptyset$, und alle versteckten Variablen von a_i kommen in a vor.

Eine co-Hornformel $r(t) \Rightarrow \varphi$ ist **coinduktiv**, wenn t objekt-normal ist. SP ist **coinduktiv**, wenn

(3) alle Axiome für versteckte Symbole coinduktiv sind,

(4) alle Axiome für Destruktoren oder Transitionsprädikate außer diesen nur sichtbare Symbole enthalten.

Satz 9.2.17 ([85], Theoreme 3.9(b) und 5.15; [87], Theorem 5.4) Eine funktionale, bild-endliche und coinduktive Spezifikation SP ist verhaltenskongruent. Ist SP darüberhinaus modal, dann erfüllt das Verhaltensmodell von SP alle Axiome der Spezifikation. ❑

Eine entscheidende Bedingung, die ein Axiom $f(t, u)\{\equiv v\} \Leftarrow \varphi$ für einen Destruktor oder ein Transitionsprädikat f coinduktiv macht, ist das "Muster" von t: nur das äußerste Symbol von t darf ein Konstruktor sein, alle anderen Symbole müssen sichtbar sein oder Variablen. Ist f weder Destruktor noch Transitionsprädikat, dann muss t sogar aus Variablen bestehen. Man kann den Begriff einer Objekt-Normalform verallgemeinern und darin auch versteckte Konstruktoren zulassen, für die allerdings die Regeln *Term-Splitting* und *Clash* (s. §7.1) auch bzgl. Verhaltens- anstelle von Strukturgleichheit gelten müssen. Da in jedem Fall die Bedingungen an Axiome für Destruktoren und Transitionsprädikate weniger restriktiv sind als jene an Axiome für andere Symbole, kann man zunächst versuchen, möglichst viele Symbole als Destruktoren oder Transitionsprädikate zu deklarieren. Je mehr das sind, desto größer wird allerdings die Menge der Verhaltensaxiome (s. Def. 9.2.2) und damit die Anzahl der Fälle, die bei Entfaltungen von Verhaltensgleichungen oder bei Coinduktion über $\sim$ erzeugt werden (s. §7.3).

☞ Zeigen Sie, dass alle Beispielspezifikationen dieses Kapitels coinduktiv sind!

Um Anforderungen von in STREAM oder RSTREAM spezifizierten Strömen zu beweisen, benötigen wir i.w. die in den §§7.1 - 7.3 eingeführten Entfaltungs- und Fixpunktregeln. Insbesondere Coinduktion über einer Verhaltensgleichheit wird häufig benötigt. Da die Interpretation von $\sim$ im Herbrandmodell von SP eine Äquivalenzrelation ist (Beweis!), lässt sich die Coinduktion über $\sim$ (siehe §7.3) folgendermaßen verallgemeinern: Sei $AX_\sim$ die Menge der Verhaltensaxiome von SP.

Coinduktion über $\sim$

$$\frac{\varphi(x,y) \;\Rightarrow\; x \sim y}{\bigwedge_{t\sim u \Rightarrow \psi \in AX_\sim} \quad \varphi(t,u) \Rightarrow \psi[(\varphi_\sim(v,w))/(v \sim w) \mid (v \sim w) \text{ kommt in } \psi \text{ vor}] \Uparrow}$$

wobei das Prädikat $\varphi_\sim$ durch folgende Hornaxiome definiert ist:

$$\begin{array}{lcl} \varphi_\sim(x,y) & \Leftarrow & \varphi(x,y) \\ \varphi_\sim(x,x) & & \\ \varphi_\sim(x,y) & \Leftarrow & \varphi_\sim(y,x) \\ \varphi_\sim(x,z) & \Leftarrow & \varphi_\sim(x,y) \wedge \varphi_\sim(y,z) \end{array}$$

Beispiel 9.2.18 Wir wollen zeigen, dass der Strom *blink* fair bzgl. $eq(0)$ ist, dass also das Atom $fair(eq(0), blink)$ ein induktives Theorem von STREAM (Bsp. 9.2.10). Um sie zu beweisen, müssen wir die Behauptung verallgemeinern und gleichzeitig zeigen, dass auch der Strom $1 : blink$ fair bzgl. $eq(0)$ ist. Der Beweis wurde wieder mit *Expander2* erstellt [81].

```
fair(eq(0),blink) & fair(eq(0),1:blink)

Applying coinduction w.r.t.

fair(F0,s0)  ===>  exists(F0,s0) & fair(F0,tail(s0))

to the preceding tree leads to the formula

All F0 s0:
(       F0 = eq(0) & s0 = blink
      | F0 = eq(0) & s0 = 1:blink
 ===>   exists(F0,s0)
      & (  F0 = eq(0) & tail(s0) = blink
         | F0 = eq(0) & tail(s0) = 1:blink))

Simplifying (57 steps) the preceding tree leads to the summand

  tail(1:blink) = blink
& tail(blink) = blink
& exists(eq(0),blink)
& exists(eq(0),1:blink)

Adding the other summands leads to

    tail(1:blink) = blink & tail(blink) = blink
  & exists(eq(0),blink) & exists(eq(0),1:blink)
|   tail(1:blink) = 1:blink & tail(blink) = blink
  & exists(eq(0),blink) & exists(eq(0),1:blink)
|   tail(1:blink) = blink & tail(blink) = 1:blink
  & exists(eq(0),blink) & exists(eq(0),1:blink)
|   tail(1:blink) = 1:blink & tail(blink) = 1:blink
  & exists(eq(0),blink) & exists(eq(0),1:blink)
```

```
Narrowing (20 steps) the entire formula leads to True.
```

Man beachte die unterschiedliche Art der Aussagen, die mit Fixpunktinduktion bzw. Coinduktion bewiesen werden. Im ersten Fall wird eine Formel $r(x) \Rightarrow \varphi(x)$ bewiesen, wobei r gegeben ist und φ eine an r gestellte Anforderung ist. Ist r der Graph einer Funktion f, dann beschreibt r die gewünschte Ein/Ausgaberelation von f (s §7.2). Demgegenüber wird Coinduktion zum Beweis von Formeln $r(x) \Leftarrow \varphi(x)$ verwendet, die aussagen, dass r für alle durch φ beschriebenen Objekte gilt. Coinduktion dient demnach eher dem Nachweis von Eigenschaften von Daten, während mit Fixpunktinduktion Eigenschaften von Funktionen und Relationen bewiesen werden. Zeigt das eine formale Dualität zwischen funktional-logischer Spezifikation einerseits und objektorientierter Spezifikation andererseits auf?

☞ Erweitern Sie STREAM (Bsp. 9.2.10) um Axiome für die beiden Ströme 0^ω und 1^ω, die aus Nullen bzw. Einsen bestehen, und zeigen Sie, dass die Äquivalenz $zip(0^\omega, 1^\omega) \sim blink$ ein induktives Theorem von STREAM ist!

☞ Zeigen Sie $odds(blink) \sim 0^\omega$ und $evens(blink) \sim 1^\omega$!

Beispiel 9.2.19 (s. Bsp. 9.2.11) Wir zeigen, dass

$$fair(g \circ f, s) \quad \Rightarrow \quad filter(g, map(f, s)) \;\sim\; map(f, filter(g \circ f, s)) \tag{1}$$

ein induktives Theorem von RSTREAM ist i.w. durch Coinduktion über $\sim$ und induktive Kontraktion entlang einer wohlfundierten Ordnung $\ll$ (s. §7.2), die wie folgt spezifiziert ist:

$$(g, f, s) \gg (g, f, t) \quad \Leftarrow \quad first(g \circ f, s, m) \wedge first(g \circ f, t, n) \wedge m > n.$$

Außerdem erweitern wir RSTREAM um ein Hilfsprädikat $p : stream \times stream \times (entry \to bool) \times (entry \to bool)$:

$$p(t, u, g, f) \quad \Leftarrow \quad t \equiv filter(g, map(f, s)) \wedge u \equiv map(f, filter(g \circ f, s)) \wedge fair(g \circ f, s),$$

und setzen die Gültigkeit folgender Lemmas voraus:

$$s \xrightarrow{x} t \wedge fair(g, s) \quad \Rightarrow \quad fair(g, t) \tag{A}$$
$$s \xrightarrow{x} t \wedge g(x) \equiv false \wedge fair(g, s) \quad \Rightarrow \quad (g, s) \gg (g, t). \tag{B}$$

☞ Zeigen Sie (A) und (B)!

Beweis von (1):

$fair(g \circ f, s) \;\Rightarrow\; filter(g, map(f, s)) \sim map(f, filter(g \circ f, s))$

Variablen-Einführung

$\vdash \; p(t, u, g, f) \;\Rightarrow\; t \sim u$

Coinduktion über $\sim$

$\vdash \; p(t, u, g, f) \wedge t \xrightarrow{x} s_1 \;\Rightarrow\; \exists s_2 (u \xrightarrow{x} s_2 \wedge p(s_1, s_2, g, f))$,
$p(t, u, g, f) \wedge u \xrightarrow{x} s_2 \;\Rightarrow\; \exists s_1 (t \xrightarrow{x} s_2 \wedge p(s_1, s_2, g, f))$,
$p(t, u, g, f) \wedge disabled(t) \;\Rightarrow\; disabled(u)$,
$p(t, u, g, f) \wedge disabled(u) \;\Rightarrow\; disabled(t)$

Variablen-Entfernung

$$\vdash \; filter(g, map(f, s)) \xrightarrow{x} s_1 \wedge fair(g \circ f, s) \;\Rightarrow\; \exists s_2 (map(f, filter(g \circ f, s)) \xrightarrow{x} s_2 \wedge p(s_1, s_2, g, f)), \tag{2}$$
$$map(f, filter(g \circ f, s)) \xrightarrow{x} s_2 \wedge fair(g \circ f, s) \;\Rightarrow\; \exists s_1 (filter(g, map(f, s)) \xrightarrow{x} s_1 \wedge p(s_1, s_2, g, f)), \tag{3}$$
$$disabled(filter(g, map(f, s))) \wedge fair(g \circ f, s) \;\Rightarrow\; disabled(map(f, filter(g \circ f, s))), \tag{4}$$
$$disabled(map(f, filter(g \circ f, s))) \wedge fair(g \circ f, s) \;\Rightarrow\; disabled(filter(g, map(f, s))) \tag{5}$$

Entfaltung

$\vdash\ (map(f,s) \xrightarrow{x} s_3 \wedge g(x) \equiv true \wedge s_1 \equiv filter(g,s_3) \wedge fair(g \circ f,s)) \vee$
$(map(f,s) \xrightarrow{y} s_3 \wedge g(y) \equiv false \wedge filter(g,s_3) \xrightarrow{x} s_1 \wedge fair(g \circ f,s))$
$\Rightarrow\ \exists s_2(map(f,filter(g \circ f,s)) \xrightarrow{x} s_2 \wedge p(s_1,s_2,g,f)),$
$filter(g \circ f,s) \xrightarrow{y} s_3 \wedge x \equiv f(y) \wedge s_2 \equiv map(f,s_3) \wedge fair(g \circ f,s)$
$\Rightarrow\ \exists s_1(filter(g,map(f,s)) \xrightarrow{x} s_1 \wedge p(s_1,s_2,g,f)),$
$(disabled(map(f,s)) \wedge fair(g \circ f,s)) \vee$
$(map(f,s) \xrightarrow{x} s_1 \wedge g(x) \equiv false \wedge disabled(filter(g,s_1)) \wedge fair(g \circ f,s))$
$\Rightarrow\ disabled(filter(g \circ f,s)),$
$disabled(filter(g \circ f,s)) \wedge fair(g \circ f,s)$
$\Rightarrow\ disabled(map(f,s))\ \vee\ \exists x,s_1(map(f,s) \xrightarrow{x} s_1 \wedge g(x) \equiv false \wedge disabled(filter(g,s_1)))$

Entfaltung

$\vdash\ (s \xrightarrow{y} s_4 \wedge x \equiv f(y) \wedge s_3 \equiv map(f,s_4) \wedge g(x) \equiv true \wedge s_1 \equiv filter(g,s_3) \wedge fair(g \circ f,s)) \vee$
$(s \xrightarrow{z} s_4 \wedge y \equiv f(z) \wedge s_3 \equiv map(f,s_4) \wedge g(y) \equiv false \wedge filter(g,s_3) \xrightarrow{x} s_1 \wedge fair(g \circ f,s))$
$\Rightarrow\ \exists s_2(map(f,filter(g \circ f,s)) \xrightarrow{x} s_2 \wedge p(s_1,s_2,g,f)),$
$filter(g \circ f,s) \xrightarrow{y} s_3 \wedge x \equiv f(y) \wedge s_2 \equiv map(f,s_3) \wedge fair(g \circ f,s)$
$\Rightarrow\ \exists s_1(filter(g,map(f,s)) \xrightarrow{x} s_1 \wedge p(s_1,s_2,g,f)),$
$(disabled(s) \wedge fair(g \circ f,s)) \vee$
$(s \xrightarrow{y} s_2 \wedge x \equiv f(y) \wedge s_1 \equiv map(f,s_2) \wedge g(x) \equiv false \wedge disabled(filter(g,s_1)) \wedge fair(g \circ f,s))$
$\Rightarrow\ disabled(filter(g \circ f,s)),$
$disabled(filter(g \circ f,s) \wedge fair(g \circ f,s))$
$\Rightarrow disabled(s) \vee$
$\exists x,y,s_1,s_2(s \xrightarrow{y} s_2 \wedge x \equiv f(y) \wedge s_1 \equiv map(f,s_2) \wedge g(x) \equiv false \wedge disabled(filter(g,s_1)))$

Implikationsregel und Variablen-Entfernung

$\vdash\ s \xrightarrow{y} s_4 \wedge g(f(y)) \equiv true \wedge fair(g \circ f,s)$
$\Rightarrow\ \exists s_2(map(f,filter(g \circ f,s)) \xrightarrow{f(y)} s_2 \wedge p(filter(g,map(f,s_4)),s_2)),$ (6)
$s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g,map(f,s_4)) \xrightarrow{x} s_1 \wedge fair(g \circ f,s)$
$\Rightarrow\ \exists s_2(map(f,filter(g \circ f,s)) \xrightarrow{x} s_2 \wedge p(s_1,s_2,g,f)),$ (7)
$filter(g \circ f,s) \xrightarrow{y} s_3 \wedge fair(g \circ f,s)$
$\Rightarrow\ \exists s_1(filter(g,map(f,s)) \xrightarrow{f(y)} s_1 \wedge p(s_1,map(f,s_3),g,f)),$ (8)
$(disabled(s) \wedge fair(g \circ f,s)) \vee$
$(s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g,map(f,s_2))) \wedge fair(g \circ f,s))$
$\Rightarrow\ disabled(filter(g \circ f,s)),$ (9)
$disabled(filter(g \circ f,s)) \wedge fair(g \circ f,s)$
$\Rightarrow disabled(s)\ \vee\ \exists y,s_2(s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g,map(f,s_2))))$ (10)

Entfaltung

$\vdash\ s \xrightarrow{y} s_4 \wedge g(f(y)) \equiv true \wedge fair(g \circ f,s)$
$\Rightarrow\ \exists s_2,s_3(filter(g \circ f,s) \xrightarrow{y} s_3 \wedge s_2 \equiv map(f,s_3) \wedge p(filter(g,map(f,s_4)),s_2)),$ (6)
(7),(8),(9),(10)

Resolution mit einem Axiom für $\longrightarrow$ und Variablen-Entfernung

$\vdash\ s \xrightarrow{y} s_4 \wedge filter(g \circ f,s) \xrightarrow{y} filter(g \circ f,s_4) \wedge fair(g \circ f,s)$
$\Rightarrow\ \exists s_3(filter(g \circ f,s) \xrightarrow{y} s_3 \wedge p(filter(g,map(f,s_4)),map(f,s_3))),$ (6)
(7),(8),(9),(10)

Entfaltung

$\vdash\ s \xrightarrow{y} s_4 \wedge filter(g \circ f,s) \xrightarrow{y} filter(g \circ f,s_4) \wedge fair(g \circ f,s)$
$\Rightarrow\ filter(g \circ f,s) \xrightarrow{y} filter(g \circ f,s_4) \wedge fair(g \circ f,s_4),$ (6)
(7),(8),(9),(10)

Coresolution mit (A) (siehe §7.1)

$\vdash\ filter(g \circ f, s) \xrightarrow{y} filter(g \circ f, s_4) \wedge fair(g \circ f, s_4)$
$\quad \Rightarrow\ filter(g \circ f, s) \xrightarrow{y} filter(g \circ f, s_4) \wedge fair(g \circ f, s_4),$ (6)
(7),(8),(9),(10)

Subsumption

$\vdash\ s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g, map(f, s_4)) \xrightarrow{x} s_1 \wedge fair(g \circ f, s)$
$\quad \Rightarrow\ \exists s_2(map(f, filter(g \circ f, s)) \xrightarrow{x} s_2 \wedge p(s_1, s_2, g, f)),$ (7)
(8),(9),(10)

Coresolution mit (A) und (B)

$\vdash\ (g \circ f, s) \gg (g \circ f, s_4) \wedge s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g, map(f, s_4)) \xrightarrow{x} s_1 \wedge fair(g \circ f, s_4)$
$\quad \Rightarrow\ \exists s_2(map(f, filter(g \circ f, s)) \xrightarrow{x} s_2 \wedge p(s_1, s_2, g, f)),$ (7)
(8),(9),(10)

Coresolution mit Induktionshypothese (2)

$\vdash\ s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge map(f, filter(g \circ f, s_4)) \xrightarrow{x} s_3 \wedge p(s_1, s_3)$
$\quad \Rightarrow\ \exists s_2(map(f, filter(g \circ f, s)) \xrightarrow{x} s_2 \wedge p(s_1, s_2, g, f)),$ (7)
(8),(9),(10)

Entfaltung

$\vdash\ s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g \circ f, s_4) \xrightarrow{y} s_5 \wedge x \equiv f(y) \wedge s_3 \equiv map(f, s_5) \wedge p(s_1, s_3)$
$\quad \Rightarrow\ \exists s_2(map(f, filter(g \circ f, s)) \xrightarrow{x} s_2 \wedge p(s_1, s_2, g, f)),$ (7)
(8),(9),(10)

Variablen-Entfernung

$\vdash\ s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g \circ f, s_4) \xrightarrow{y} s_5 \wedge p(s_1, map(f, s_5), g, f)$
$\quad \Rightarrow\ \exists s_2(map(f, filter(g \circ f, s)) \xrightarrow{f(y)} s_2 \wedge p(s_1, s_2, g, f)),$ (7)
(8),(9),(10)

Coresolution mit Axiomen für $\longrightarrow$

$\vdash\ map(f, filter(g \circ f, s)) \xrightarrow{f(y)} map(f, s_5) \wedge p(s_1, map(f, s_5), g, f)$
$\quad \Rightarrow\ \exists s_2(map(f, filter(g \circ f, s)) \xrightarrow{f(y)} s_2 \wedge p(s_1, s_2, g, f)),$ (7)
(8),(9),(10)

Subsumption

$\vdash\ filter(g \circ f, s) \xrightarrow{y} s_3 \wedge fair(g \circ f, s)$
$\quad \Rightarrow\ \exists s_1(filter(g, map(f, s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f, s_3), g, f)),$ (8)
(9),(10)

Entfaltung

$\vdash\ (s \xrightarrow{y} s_4 \wedge g(f(y)) \equiv true \wedge s_3 \equiv filter(g \circ f, s_4) \wedge fair(g \circ f, s)) \vee$
$(s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g \circ f, s_4) \xrightarrow{y} s_3 \wedge fair(g \circ f, s))$
$\quad \Rightarrow\ \exists s_1(filter(g, map(f, s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f, s_3), g, f)),$ (8)
(9),(10)

Implikationsregel und Variablen-Entfernung

$\vdash\ s \xrightarrow{y} s_4 \wedge g(f(y)) \equiv true \wedge fair(g \circ f, s)$
$\quad \Rightarrow\ \exists s_1(filter(g, map(f, s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f, filter(g \circ f, s_4)), g, f)),$ (11)
$s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g \circ f, s_4) \xrightarrow{y} s_3 \wedge fair(g \circ f, s)$
$\quad \Rightarrow\ \exists s_1(filter(g, map(f, s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f, s_3), g, f)),$ (12)
(9),(10)

Coresolution mit Axiomen für $\longrightarrow$

$\vdash\ s \xrightarrow{y} s_4 \wedge filter(g, map(f, s)) \xrightarrow{f(y)} filter(g, map(f, s_4)) \wedge fair(g \circ f, s)$
$\quad \Rightarrow\ \exists s_1(filter(g, map(f, s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f, filter(g \circ f, s_4)), g, f)),$ (11)

(9),(10),(12)

Entfaltung

$\vdash\ s \xrightarrow{y} s_4 \wedge filter(g, map(f,s)) \xrightarrow{f(y)} filter(g, map(f,s_4)) \wedge fair(g \circ f, s)$
$\Rightarrow\ filter(g, map(f,s)) \xrightarrow{f(y)} filter(g, map(f,s_4)) \wedge fair(g \circ f, s_4),$ (11)

(9),(10),(12)

Coresolution mit (A)

$\vdash\ filter(g, map(f,s)) \xrightarrow{f(y)} filter(g, map(f,s_4)) \wedge fair(g \circ f, s_4)$
$\Rightarrow\ filter(g, map(f,s)) \xrightarrow{f(y)} filter(g, map(f,s_4)) \wedge fair(g \circ f, s_4),$ (11)

(9),(10),(12)

Subsumption

$\vdash\ s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g \circ f, s_4) \xrightarrow{y} s_3 \wedge fair(g \circ f, s)$
$\Rightarrow\ \exists s_1 (filter(g, map(f,s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f,s_3), g, f)),$ (12)

(9),(10)

Coresolution mit (A) und (B)

$\vdash\ (g \circ f, s) \gg (g \circ f, s_4) \wedge s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g \circ f, s_4) \xrightarrow{y} s_3 \wedge fair(g \circ f, s_4)$
$\Rightarrow\ \exists s_1 (filter(g, map(f,s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f,s_3), g, f)),$ (12)

(9),(10)

Coresolution mit einem Axiom für $\longrightarrow$

$\vdash\ (g \circ f, s) \gg (g \circ f, s_4) \wedge s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge map(f, filter(g \circ f, s_4)) \xrightarrow{f(y)} map(f, s_3) \wedge fair(g \circ f, s_4)$
$\Rightarrow\ \exists s_1 (filter(g, map(f,s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f,s_3), g, f)),$ (12)

(9),(10)

Coresolution mit Induktionshypothese (3)

$\vdash\ s \xrightarrow{z} s_4 \wedge g(f(z)) \equiv false \wedge filter(g, map(f,s_4)) \xrightarrow{f(y)} s_2 \wedge p(s_2, map(f,s_3), g, f)$
$\Rightarrow\ \exists s_1 (filter(g, map(f,s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f,s_3), g, f)),$ (12)

(9),(10)

Coresolution mit Axiomen für $\longrightarrow$

$\vdash\ filter(g, map(f,s)) \xrightarrow{f(y)} s_2 \wedge p(s_2, map(f,s_3), g, f)$
$\Rightarrow\ \exists s_1 (filter(g, map(f,s)) \xrightarrow{f(y)} s_1 \wedge p(s_1, map(f,s_3), g, f)),$ (12)

(9),(10)

Subsumption

$\vdash\ (disabled(s) \wedge fair(g \circ f, s)) \vee$
$(s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f,s_2))) \wedge fair(g \circ f, s))$
$\Rightarrow\ disabled(filter(g \circ f, s)),$ (9)

(10)

Implikationsregel

$\vdash\ disabled(s) \wedge fair(g \circ f, s) \ \Rightarrow\ disabled(filter(g \circ f, s)),$
$s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f,s_2))) \wedge fair(g \circ f, s)$
$\Rightarrow\ disabled(filter(g \circ f, s))$ (13)

(10)

Coresolution mit einem Axiom für *disabled*

$\vdash\ disabled(filter(g \circ f, s)) \wedge fair(g \circ f, s) \ \Rightarrow\ disabled(filter(g \circ f, s)),$

(10),(13)

Subsumption

$\vdash\ s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f,s_2))) \wedge fair(g \circ f, s)$
$\Rightarrow\ disabled(filter(g \circ f, s))$ (13)

(10)

Coresolution mit (A) und (B)

$\vdash \; (g \circ f, s) \gg (g \circ f, s_2) \wedge s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))) \wedge fair(g \circ f, s_2)$
$\Rightarrow \; disabled(filter(g \circ f, s))$ (13)
(10)

Coresolution mit Induktionshypothese (4)

$\vdash \; s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(map(f, filter(g \circ f, s_2))) \; \Rightarrow \; disabled(filter(g \circ f, s))$ (13)
(10)

Entfaltung

$\vdash \; s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g \circ f, s_2)) \; \Rightarrow \; disabled(filter(g \circ f, s))$ (13)
(10)

Coresolution mit einem Axiom für *disabled*

$\vdash \; disabled(filter(g \circ f, s)) \; \Rightarrow \; disabled(filter(g \circ f, s))$ (13)
(10)

Subsumption

$\vdash \; disabled(filter(g \circ f, s)) \wedge fair(g \circ f, s)$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))))$ (10)

Entfaltung

$\vdash \; (disabled(s) \wedge fair(g \circ f, s)) \vee$
$(s \xrightarrow{x} s_1 \wedge g(f(x)) \equiv false \wedge disabled(filter(g \circ f, s_1)) \wedge fair(g \circ f, s))$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))))$

Implikationsregel

$\vdash \; disabled(s) \wedge fair(g \circ f, s)$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2)))),$ (14)
$s \xrightarrow{x} s_1 \wedge g(f(x)) \equiv false \wedge disabled(filter(g \circ f, s_1)) \wedge fair(g \circ f, s)$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))))$ (15)

Subsumption

$\vdash \; s \xrightarrow{x} s_1 \wedge g(f(x)) \equiv false \wedge disabled(filter(g \circ f, s_1)) \wedge fair(g \circ f, s)$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))))$ (15)

Coresolution mit einem Axiom für *disabled*

$\vdash \; s \xrightarrow{x} s_1 \wedge g(f(x)) \equiv false \wedge disabled(map(f, filter(g \circ f, s_1))) \wedge fair(g \circ f, s)$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))))$

Coresolution mit (A) und (B)

$\vdash \; (g \circ f, s) \gg (g \circ f, s_1) \wedge s \xrightarrow{x} s_1 \wedge g(f(x)) \equiv false \wedge disabled(map(f, filter(g \circ f, s_1))) \wedge fair(g \circ f, s_1)$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))))$

Coresolution mit Induktionshypothese (5)

$\vdash \; s \xrightarrow{x} s_1 \wedge g(f(x)) \equiv false \wedge disabled(filter(g, map(f, s_1)))$
$\Rightarrow disabled(s) \; \vee \; \exists y, s_2 (s \xrightarrow{y} s_2 \wedge g(f(y)) \equiv false \wedge disabled(filter(g, map(f, s_2))))$

Subsumption

$\vdash \; True$ ✌

☞ Formulieren Sie den Beweis von Beispiel 9.2.19 um in einen Beweis des Theorems

$$finite(s) \quad \Rightarrow \quad filter(g, map(f, s)) \; \sim \; map(f, filter(g \circ f, s))! \tag{16}$$

Verwenden Sie dazu die Stromordnung $\gg'$ mit dem Axiom

$$s \gg' t \quad \Leftarrow \quad length(s, m) \wedge length(s, n) \wedge m > n$$

und die Lemmas:

$$s \xrightarrow{x} t \wedge finite(s) \quad \Rightarrow \quad finite(t) \tag{C}$$
$$s \xrightarrow{x} t \wedge finite(s) \quad \Rightarrow \quad s \gg' t. \tag{D}$$

☞ Zeigen Sie (C) und (D)!

In §5.1 haben wir konstruktorbasierte Spezifikationen eingeführt, in §7.3 um Coprädikate, die als größte Lösungen ihrer Axiome interpretiert werden, ergänzt und schließlich in diesem Abschnitt versteckte Sorten, Destruktoren und Transitionsprädikate hinzugefügt, um zustandsorientierte Modelle zu spezifizieren. Das ist noch nicht das Ende der Fahnenstange! Immer noch sind nämlich alle Elemente des Herbrandmodells einer Spezifikation termerzeugt. Das bedeutet, dass wir eine syntaktische Struktur aller untersuchten Daten voraussetzen bzw. festlegen müssen und uns darauf bei der Verifikation auch beziehen, wie die Induktionsschritte im letzten Beispiel zeigen. Mit der Einführung von Verhaltensgleichheiten haben wir lediglich erreicht, dass auch Daten mit unterschiedlichen Normalformen als äquivalent betrachtet werden können. In vielen Anwendungen tauchen aber oft Typen von Objekten auf, über deren syntaktische Struktur man nun wirklich nichts sagen kann oder will.

"Nichts sagen können" gilt insbesondere für jede überabzählbare Menge, zumindest dann, wenn man versucht, jedes ihrer Elemente durch eine (variablenfreie) Normalform über einer gegebenen *endlichen* Signatur zu repräsentieren. Genaugenommen werden in den Beispielen 9.2.10 und 9.2.11 ja *nur abzählbare* Teilmengen aller Ströme von Elementen einer beliebigen festen Menge spezifiziert, obwohl man auf der Basis dieser Spezifikation eigentlich über alle Ströme reden will und nicht nur die durch Terme benannten wie *blink* oder $nats(5)$.

"Nichts sagen wollen" gilt vermutlich für die Menge der Instanzen einer oder mehrerer Klassen eines objektorientierten Programms. Das Beispiel 9.2.8 ist einfach insofern als dort alle Destruktoren sichtbare Werte haben und damit — in objektorientierter Terminologie — *Attribute* repräsentieren. Sobald aber eine Klasse(nsorte) Destruktoren mit versteckten Werten hat, kann man die Objekte meist nicht durch Tupel von Attributwerten darstellen.[65] Die versteckten Werte stehen ja dann für weitere Klasseninstanzen, und es würde schnell eine zyklische Struktur entstehen, wollte man alle über solche Verweis(?)-Ketten erreichbare Attribute aufsammeln und in einen Term stecken. Solche zyklischen Strukturen kann man zwar zu unendlichen Bäumen *entfalten*. Die bilden aber kein Herbrandmodell. Wie verbreitet Destruktoren mit versteckten Werten sind, erkennt man z.B. daran, dass jede Assoziation in einem UML-Klassendiagramm einem solchen Destruktor entspricht.

Die nach dem augenblicklichen Stand der Forschung geeignetsten Modelle nicht termerzeugter Datenbereiche sind **finale Coalgebren**. Deren Eigenschaften sind tatsächlich komplementär zu denen von Herbrandmodellen. Philosophisch betrachtet bestehen Herbrandmodelle aus Teilchen und finale Coalgebren aus Wellen. Dabei entspricht eine Welle der *Beobachtungs*-Funktion, die einen ausschließlich aus Destruktoren zusammengesetzten *Kontext* interpretiert (s. Def. 9.2.9). Ein Element der finalen Coalgebra ist ein — in der Regel unendliches — Tupel von Beobachtungsfunktionen, für jeden Kontext eine Funktion. Ordnet man jedem Kontext ein eigenes Funktionssymbol zu und bildet man daraus eine Signatur Σ, dann entspricht jede Σ-Algebra einem *Element* der finalen Coalgebra. Hier schließt sich der Kreis zu Kripke-Strukturen, die ja auch viele Modelle zu einem zusammenfassen. Wir wollen das hier nicht weiter ausführen und verweisen stattdessen die interessierte Leserin auf Artikel wie [52, 87, 100]. In [87] werden Teile von UML-Modellen in (co)algebraische Spezifikationen übersetzt. Signatur und Axiome einer coalgebraischen Spezifikation gleichen denen einer konstruktorbasierten. Der Unterschied (oder besser: die Ergänzung, denn natürlich will man beides kombinieren) liegt einzig in der Semantik und hier vor allem in der Datenrepräsentation.

Coalgebren stammen ursprünglich aus der **Kategorientheorie**, die im Kapitel 11 dieses Skriptes zur Definition der Semantik des λ-Kalküls eingesetzt wird. In §11.5 werden deshalb kurz auch Coalgebren angesprochen. Als Einführung in kategorientheoretische Begriffe ist Teil V von [26] zu empfehlen.

[65] In Programmiersprachen kommen solche Tupel unter der Bezeichnung *Verbünde* oder *Records* vor.

10 CPO-Theorie

Dieses und das folgende Kapitel sind als Anhang zu verstehen, in dem der klassische Ansatz zur Semantik-Definition rekursiver und imperativer Programme sowie gleichungsdefinierter Datentypen vorgestellt wird. Frühere Kapitel benutzen keine Begriffe oder Ergebnisse aus diesem Anhang.

10.1 CPO-Semantik

Definition 10.1.1 (CPO) Sei A eine Menge. Eine binäre Relation $\leq$ auf A heißt **Halbordnung**, falls für alle $a, b, c \in A$ gilt:

- $a \leq a$, (Reflexivität)
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$, (Transitivität)
- $a \leq b \wedge b \leq a \Rightarrow a = b$. (Antisymmetrie)

$a \in A$ heisst **Supremum** einer Teilmenge B von A, wenn gilt:

- $\forall b \in B : b \leq a$, ($a$ ist obere Schranke von B)
- $\forall c \in A : \ ((\forall b \in B : b \leq c) \Rightarrow a \leq c)$. ($a$ ist kleinste obere Schranke von B)

Eine Folge $\{a_i\}_{i\in\mathbb{N}} \subseteq A$ heißt **Kette von A bzgl. $\leq$**, falls $a_i \leq a_{i+1}$ für alle $i \in \mathbb{N}$ gilt. A ist **kettenendlich**, wenn jede Kette $\{a_i\}_{i\in\mathbb{N}}$ von A nur endlich viele Elemente enthält. A ist ein **CPO** (*complete partial order*), wenn jede Kette $\{a_i\}_{i\in\mathbb{N}}$ von A ein Supremum $sup\{a_i\}_{i\in\mathbb{N}}$ hat.

Sei A eine Menge und $\bot \notin A$. Der CPO B heißt **flache Erweiterung von** A, geschrieben: $A_\bot$, falls $B = A \cup \{\bot\}$ und für alle $a, b \in B$

$$a \leq b \quad \Longleftrightarrow \quad a = b \text{ oder } a = \bot$$

gilt. Damit ist insbesondere $\bot$ das kleinste Element von B (bzgl. $\leq$). ❑

Definition 10.1.2 (Monotonie, Stetigkeit) Eine Funktion $f : A \to B$ zwischen zwei CPOs ist **monoton**, falls für alle $a, b \in A$ mit $a \leq b$ $f(a) \leq f(b)$ gilt. Die Menge der monotonen Funktionen von A nach B bezeichnen wir mit $[A \Rightarrow B]$. f ist **stetig**, falls für jede Kette $\{a_i\}_{i\in\mathbb{N}}$ von A $\{f(a_i)\}_{i\in\mathbb{N}}$ eine Kette von B ist und

$$f(sup\{a_i\}_{i\in\mathbb{N}}) = sup\{f(a_i)\}_{i\in\mathbb{N}}$$

gilt. f ist **$\bot$-erhaltend**, falls A und B kleinste Elemente $\bot^A$ bzw. $\bot^B$ haben und $f\ \bot^A$ nach $\bot^B$ abbildet. Die Menge der $\bot$-erhaltenden Funktionen von A nach B bezeichnen wir mit $[A \stackrel{\bot}{\to} B]$. ❑

Lemma 10.1.3 Eine Funktion $f : A \to B$ ist genau dann stetig, wenn sie monoton ist und für jede Kette $\{a_i\}_{i\in\mathbb{N}}$ von A

$$f(sup\{a_i\}_{i\in\mathbb{N}}) \leq sup\{f(a_i)\}_{i\in\mathbb{N}}$$

gilt. Ist A kettenendlich, dann sind alle monotonen Funktionen auf A stetig. ❑

Satz 10.1.4 (Verallgemeinerung von 4.2.11) Sei A ein CPO mit kleinstem Element $\bot$. Jede stetige Funktion $\phi : A \to A$ hat einen kleinsten Fixpunkt $lfp(\phi)$, also $\phi(lfp(\phi)) = lfp(\phi)$ und $lfp(\phi) \leq a$ für alle $a \in A$ mit $\phi(a) = a$. $lfp(\phi)$ ist definiert durch $lfp(\phi) = sup\{\phi^i(\bot)\}_{i\in\mathbb{N}}$. ❑

Aus diesem Satz erhält man eine auf CPOs zugeschnittene Induktionsregel, die wie die in §7.2 behandelte Fixpunktinduktion aus Kleene's Fixpunktsatz (hier der Version 10.1.4) abgeleitet ist:

Definition und Satz 10.1.5 (Scott-Induktion) (s. [61]) Sei A ein CPO mit kleinstem Element $\bot$. Eine Formel $F(x)$ heißt **zulässig**, wenn für alle Ketten $\{a_i\}_{i\in\mathbb{N}} \subseteq A$ gilt:

$$(\forall i \in \mathbb{N} : a_i \models F(x)) \quad \Rightarrow \quad sup\{a_i\}_{i\in\mathbb{N}} \models F(x).$$

Sei F eine zulässige Σ-Formel und $\phi : A \to A$ eine stetige Funktion. Nach Satz 10.1.4 hat ϕ den kleinsten Fixpunkt $lfp(\phi) =_{def} sup\{\phi^i(\bot)\}_{i\in\mathbb{N}}$. Die folgende Regel ist korrekt bzgl. A:

Scott-Induktion $$\frac{F(lfp(\phi))}{F(\bot) \ \wedge \ \forall x(F(x) \Rightarrow F(\phi(x)))} \Uparrow \ \square$$

Das Objekt, über das man mit F eine Aussage macht, muss also zunächst als kleinster Fixpunkt einer stetigen Funktion beschrieben werden, bevor man Scott-Induktion anwenden kann (s.u. Lemma 10.2.3).

Ein Fixpunkt a von ϕ ist gleichbedeutend mit einer Lösung der Gleichung $\phi(x) = x$. Die abstrakte Syntax der imperativen Sprache IPF (3.8) enthält u.a. die Funktionssymbole $loop : boolexp \times com \to com$ und $repeat : com \times boolexp \to com$, deren Bedeutung als Lösung der Gleichungen

$$loop(b, c) = cond(b, seq(c, loop(b, c)), skip) \quad \text{bzw.} \quad repeat(c, b) = seq(c, cond(b, skip, repeat(c, b)))$$

in den Variablen (!) $loop$ bzw. $repeat$ definiert werden soll. Nach Satz 10.1.4 existieren solche Lösungen, falls $loop$ und $repeat$ als Elemente eines CPOs $Dom \subseteq [A_{boolexp\times com} \to A_{com}]$ interpretierbar und die Funktionen $\phi, \psi : Dom \to Dom$ mit

$$\phi(f)(b, c) \ =_{def} \ cond^A(b, seq^A(c, f(b, c)), skip^A) \quad \text{bzw.} \quad \psi(f)(c, b) \ =_{def} \ seq^A(c, cond^A(b, skip^A, f(c, b)))$$

stetig sind. Dazu brauchen wir erst noch mehr Theorie.

Definition 10.1.6 (Fortsetzungen von $\leq$) Seien A_i, $1 \leq i \leq n$, CPOs mit den Halbordnungen $\leq^{A_i}$, $1 \leq i \leq n$. Dann wird durch

$$(a_1, \ldots, a_n) \leq (b_1, \ldots, b_n) \iff_{def} a_1 \leq^{A_1} b_1, \ldots, a_n \leq^{A_n} b_n$$

eine Halbordnung $\leq^A$ auf $A =_{def} A_1 \times \ldots \times A_n$ definiert. A ist ein CPO, wobei für alle Ketten $\{a_i\}_{i\in\mathbb{N}} = \{(a_{i,1}, \ldots, a_{i,n})\}_{i\in\mathbb{N}} \subseteq A$

$$sup\{a_i\}_{i\in\mathbb{N}} = (sup\{a_{i,1}\}_{i\in\mathbb{N}}, \ldots, sup\{a_{i,n}\}_{i\in\mathbb{N}})$$

gilt. Hat A_i, $1 \leq i \leq n$, das kleinste Element $\bot^{A_i}$, dann hat A das kleinste Element $(\bot^{A_1}, \ldots, \bot^{A_n})$. $f : A \to B$ ist **strikt**, wenn für alle $(a_1, \ldots, a_n) \in A$ und $1 \leq i \leq n$ $f(a_1, \ldots, a_{i-1}, \bot, a_{i+1}, \ldots, a_n) = \bot$ gilt.

Seien A eine Menge und B ein CPO mit der Halbordnung $\leq^B$. Dann wird durch

$$f \leq g \iff_{def} f(a) \leq g(a) \text{ für alle } a \in A$$

eine Halbordnung $\leq^C$ auf $C =_{def} [A \to B]$ definiert. C ist ein CPO, wobei für alle Ketten $\{f_i\}_{i\in\mathbb{N}} \subseteq C$ und $a \in A$

$$(sup\{f_i\}_{i\in\mathbb{N}})(a) = sup\{f_i(a)\}_{i\in\mathbb{N}}$$

gilt. Hat B das kleinste Element $\bot$, dann hat C das kleinste Element Ω mit $\Omega(a) = \bot$ für alle $a \in A$. Auch die Menge

$$[A \Rightarrow B]$$

der monotonen Funktionen von A nach B ist ein CPO, weil $sup\{f_i\}_{i\in\mathbb{N}}$ selbst monoton (und stetig) ist (vgl. [29], Lemma 3.13). Das gleiche gilt für die Menge $[A \stackrel{\bot}{\to} B]$ der $\bot$-erhaltenden Funktionen von A nach B. $\square$

Lemma 10.1.7 (Verallgemeinerung des *lub-Lemmas* in [65], S. 365) Sei $\{f_i\}_{i\in\mathbb{N}}$ eine Kette eines CPOs der Form $[A_1 \to [A_2 \to \ldots [A_n \to A_{n+1}] \ldots]]$, wobei A_{n+1} ein kettenendlicher CPO ist. Dann gibt es für alle $a = (a_1, \ldots, a_n) \in A_1 \times \ldots \times A_n$ ein $j > 0$ mit $sup\{f_i(a_1) \ldots (a_n)\}_{i\in\mathbb{N}} = f_k(a_1) \ldots (a_n)$ für alle $k \geq j$. $\square$

Definition 10.1.8 (monotone und stetige Algebren) Sei Σ eine Signatur mit Sortenmenge S. Eine Σ-Algebra A ist **monoton** bzw. **stetig**, wenn für alle $s \in S$ A_s ein CPO ist und für alle $f : w \to s \in \Sigma$ f^A eine monotone bzw. stetige Funktion ist.

Sei $\Phi = \{F_1 : w_1 \to s_1, \ldots, F_k : w_k \to s_k\}$ eine Menge S^+-sortierter Funktionssymbole, $\Sigma' = (S, F \cup \Phi)$ und *A eine monotone Σ-Algebra, deren Trägermengen die in Lemma 10.1.7 vorausgesetzte Form haben.* Für alle $1 \leq i \leq k$ und $f_i \in [A_{w_i} \Rightarrow A_{s_i}]$ ist die Σ'-Algebra $A(f_1, \ldots, f_k)$ wie folgt definiert:

- Für alle $op : w \to s \in \Sigma$, $op^{A(f_1,\ldots,f_k)} = op^A$.
- Für alle $1 \leq i \leq k$, $F_i^{A(f_1,\ldots,f_k)} = f_i$.

Für alle $1 \leq i \leq k$ sei $t_i : w_i \to s_i \in T_{\Sigma'}$ und $t = t_1 \times \ldots \times t_k$. Dann ist die Funktion

$$t_\Phi^A : [A_{w_1} \Rightarrow A_{s_1}] \times \ldots \times [A_{w_k} \Rightarrow A_{s_k}] \to [A_{w_1} \Rightarrow A_{s_1}] \times \ldots \times [A_{w_k} \Rightarrow A_{s_k}]$$

definiert durch: $t_\Phi^A(f_1, \ldots, f_k) = t^{A(f_1,\ldots,f_k)}$.

Satz 10.1.9 t_Φ^A ist stetig.

Beweis. Wir zeigen die Behauptung durch Induktion über dem Aufbau von t. Sei $\{g_j\}_{j\in\mathbb{N}}$ eine Kette von $Dom_{w_1\ldots w_k}$.

Fall 1: Für alle Konstanten op von $\Sigma \cup C$ ist $op_\Phi(f)(a)$ unabhängig von f. Also ist op_Φ stetig.

Fall 2: $t = op(u)$ für $op : w \to s \in \Sigma$ und $u \in T_{\Sigma,w}$. Wir zeigen zunächst, dass t_Φ monoton ist. Seien $f, g \in Dom_{w_1\ldots w_k}$ mit $f \leq g$. Nach Induktionsvoraussetzung ist u_Φ monoton. Da op^A monoton ist, folgt

$$t_\Phi(f)(a) = op^A(u_\Phi(f)(a)) \leq op^A(u_\Phi(g)(a)) = t_\Phi(g)(a).$$

Wegen der Monotonie von u_Φ ist $\{u_\Phi(g_j)\}_{j\in\mathbb{N}}$ eine Kette. Daher gibt es nach Lemma 10.1.7 ein $J > 0$ mit $sup\{u_\Phi(g_j)\}_{j\in\mathbb{N}}(a) = u_\Phi(g_n)(a)$ für alle $n \geq J$. Man erhält

$$\begin{aligned} t_\Phi(sup\{g_j\}_{j\in\mathbb{N}})(a) &= op^A(u_\Phi(sup\{g_j\}_{j\in\mathbb{N}})(a)) \overset{Induktionsvor.}{=} op^A(sup\{u_\Phi(g_j)\}_{j\in\mathbb{N}}(a)) \\ &= op^A(u_\Phi(g_J)(a)) = t_\Phi(g_J)(a) \leq sup\{t_\Phi(g_j)\}_{j\in\mathbb{N}}(a), \end{aligned}$$

weil wegen der Monotonie von t_Φ $\{t_\Phi(g_j)\}_{j\in\mathbb{N}}$ eine Kette ist.

Fall 3: $t = F_i$ für ein $1 \leq i \leq k$ mit $v_i = \epsilon$. Dann gilt

$$t_\Phi(sup\{g_j\}_{j\in\mathbb{N}})(a) = (sup\{g_j\}_{j\in\mathbb{N}})_i(a) = sup\{g_{j,i}\}_{j\in\mathbb{N}}(a) = sup\{t_\Phi(g_j)\}_{j\in\mathbb{N}}(a).$$

Fall 4: $t = F_i(u)$ für ein $1 \leq i \leq k$ und $u \in T_{\Sigma,v_i}$. Wir zeigen wieder zunächst, dass t_Φ monoton ist. Seien $f, g \in Dom_{w_1\ldots w_k}$ mit $f \leq g$. Nach Induktionsvoraussetzung ist u_Φ monoton. Da f_i monoton ist, folgt

$$t_\Phi(f)(a) = f_i(u_\Phi(f)(a)) \leq f_i(u_\Phi(g)(a)) = t_\Phi(g)(a).$$

Wegen der Monotonie von u_Φ ist $\{u_\Phi(g_j)\}_{j\in\mathbb{N}}$ eine Kette. Daher gibt es nach Lemma 10.1.7 ein $J > 0$ mit $sup\{u_\Phi(g_j)\}_{j\in\mathbb{N}}(a) = u_\Phi(g_n)(a)$ für alle $n \geq J$. Ebenfalls nach Lemma 10.1.7 existiert ein $K > 0$ mit

$$sup\{g_{j,i}\}_{j\in\mathbb{N}}(sup\{u_\Phi(g_j)\}_{j\in\mathbb{N}}(a)) = g_{n,i}(sup\{u_\Phi(g_j)\}_{j\in\mathbb{N}}(a))$$

für alle $n \geq K$. Sei $h = sup\{(g_{j_1,1}, \ldots, g_{j_k,k}) \mid j_i \in \{J, K\},\ \forall r \neq i : j_r = J\}$. Man erhält

$$\begin{aligned} t_\Phi(sup\{g_j\}_{j\in\mathbb{N}})(a) &= (sup\{g_j\}_{j\in\mathbb{N}})_i(u_\Phi(sup\{g_j\}_{j\in\mathbb{N}})(a)) \\ = sup\{g_{j,i}\}_{j\in\mathbb{N}}(u_\Phi(sup\{g_j\}_{j\in\mathbb{N}})(a)) &\overset{Induktionsvor.}{=} sup\{g_{j,i}\}_{j\in\mathbb{N}}(sup\{u_\Phi(g_j)\}_{j\in\mathbb{N}}(a)) \\ = h_i(sup\{u_\Phi(g_j)\}_{j\in\mathbb{N}}(a)) = h_i(u_\Phi(h)(a)) &= t_\Phi(h)(a) \leq sup\{t_\Phi(g_j)\}_{j\in\mathbb{N}}(a), \end{aligned}$$

weil wegen der Monotonie von t_Φ $\{t_\Phi(g_j)\}_{j\in\mathbb{N}}$ eine Kette ist.

Da $a \in A_v$ beliebig ist, folgt aus den Ungleichungen der Fälle (2) und (4) die Monotonie von t_Φ sowie $t_\Phi(sup\{g_j\}_{j\in\mathbb{N}}) \leq sup\{t_\Phi(g_j)\}_{j\in\mathbb{N}}$ und damit nach Lemma 10.1.3 die Stetigkeit von t_Φ. ❑

Beispiel 10.1.10 (CPO-Semantik der imperativen Sprache IPF) Sei Σ die abstrakte Syntax von $IPF \setminus \{loop, repeat\}$ (vgl. 3.8) und $State = [X \to \mathbb{Z}]_\perp$. Die folgende Σ-Algebra A ist monoton:

- $A_{const} = \mathbb{Z}$, $A_{var} = X$ (Die Halbordnungen auf $\mathbb{Z}$ bzw. X sind die Gleichheitsrelationen auf diesen Mengen.)
- $A_{exp} = [State \overset{\perp}{\to} \mathbb{Z}_\perp]$, $A_{boolexp} = [State \overset{\perp}{\to} \{true, false\}_\perp]$, $A_{com} = [State \overset{\perp}{\to} State]$
- $i^A = i$ für alle $i \in \mathbb{Z} \cup \{true, false\} \cup X$
- $C^A(i)(s) = i$ für alle $i \in \mathbb{Z}$ und $s \in State \setminus \{\perp\}$
  $C^A(i)(\perp) = \perp$ für alle $i \in \mathbb{Z}$
- $V^A(x)(s) = s(x)$ für alle $x \in X$ und $s \in State$
- $add^A(f,g)(s) = \begin{cases} f(s)+g(s) & \text{falls } f(s)\neq\perp \text{ und } g(s)\neq\perp \\ \perp & \text{sonst} \end{cases}$ für alle $f, g \in A_{exp}$ und $s \in State$
- $eq^A(f,g)(s) = \begin{cases} f(s)=g(s) & \text{falls } f(s)\neq\perp \text{ und } g(s)\neq\perp \\ \perp & \text{sonst} \end{cases}$ für alle $f, g \in A_{exp}$ und $s \in State$
- *greater* analog
- $True^A(s) = true$, $False^A(s) = false$
- $and^A(f,g)(s) = \begin{cases} f(s)\wedge g(s) & \text{falls } f(s)\neq\perp \text{ und } g(s)\neq\perp \\ \perp & \text{sonst} \end{cases}$ für alle $f, g \in A_{boolexp}$ und $s \in State$
- $not^A(f)(s) = \begin{cases} \neg f(s) & \text{falls } f(s)\neq\perp \\ \perp & \text{sonst} \end{cases}$ für alle $f \in A_{boolexp}$ und $s \in State$
- $assign^A(x,f)(s)(y) = \begin{cases} f(s) & \text{falls } x = y \\ s(y) & \text{sonst} \end{cases}$ für alle $x, y \in X$, $f \in A_{exp}$ und $s \in State$
- $seq^A(f,g)(s) = g(f(s))$ für alle $f, g \in A_{com}$ und $s \in State$
- $cond^A(f,g,h)(s) = \begin{cases} g(s) & \text{falls } f(s) = true \\ h(s) & \text{falls } f(s) = false \\ \perp & \text{sonst} \end{cases}$ für alle $f \in A_{boolexp}$, $g, h \in A_{com}$ und $s \in State$
- $skip^A(s) = s$ für alle $s \in State$.
- Wir setzen $\Phi = \{loop\}$ und $C = \{b :\to boolexp, c :\to com\}$, $Dom = [A_{boolexp\times com} \Rightarrow A_{com}]$ und $t = cond(b, seq(c, loop(b,c)), skip)$ (vgl. Def. 10.1.8). Wegen der Monotonie von A ist $t_{loop} : Dom \to Dom$ nach Satz 10.1.9 stetig, hat also nach Satz 10.1.4 den kleinsten Fixpunkt $loop^A = sup\{(t_{loop})^i(\Omega)\}_{i\in\mathbb{N}}$. t_{loop} ist definiert durch (vgl. 10.1.8)

$$t_{loop}(f)(g,h)(s) = \begin{cases} f(g,h)(h(s)) & \text{falls } g(s) = true \\ s & \text{falls } g(s) = false \\ \perp & \text{sonst} \end{cases}$$

für alle $f \in Dom$, $g \in A_{boolexp}$, $h \in A_{com}$ und $s \in State$. Da $loop^A$ Fixpunkt von t_{loop} ist, erhält man $loop^A(g,h)(s) = t_{loop}(loop^A)(g,h)(s)$, also

$$loop^A(g,h)(s) = \begin{cases} loop^A(g,h)(h(s)) & \text{falls } g(s) = true \\ s & \text{falls } g(s) = false \\ \perp & \text{sonst} \end{cases}$$

- Analog erhält man $repeat^A$ als Fixpunkt von $seq(c, cond(b, skip, repeat(c, b)))_{repeat}$ und damit

$$repeat^A(h,g)(s) = \begin{cases} h(s) & \text{falls } g(h(s)) = true \\ repeat^A(h,g)(h(s)) & \text{falls } g(h(s)) = false \\ \bot & \text{sonst} \end{cases}$$

Die Gleichungen $loop(b,c) = cond(b, seq(c, loop(b,c)), skip)$ kann als Deklaration einer rekursiven Prozedur aufgefäßt werden. Allgemeiner beschreiben eine Menge $\Phi = \{F_1 : v_1 \to com, \ldots, F_k : v_k \to com\}$ von Funktionssymbolen und ein Gleichungssystem

$$Decl(\Phi) \;=\; \{F_1(e_1) = t_1, \ldots, F_k(e_k) = t_k\}$$

über $\Sigma \cup \Phi$ eine Menge evtl. verschränkt-rekursiver rekursiver Prozeduren. Für alle $1 \le i \le k$ sei $Dom = [A_{v_i} \Rightarrow A_{com}]$. Hier liefern die Sätze 10.1.9 und 10.1.4 für $t = t_1 \times \ldots \times t_k$ den kleinsten Fixpunkt $lfp(t_\Phi) = sup\{t^i_\Phi(\Omega)\}_{i\in\mathbb{N}}$, der das Gleichungssystem $Decl(\Phi)$ in $(F_1, \ldots, F_k)$ löst.

Z.B. lautet die Fakultätsfunktion als rekursive IPF-Prozedur in konkreter bzw. abstrakter Syntax wie folgt:

$Fact(x,y) = if\ x = C(0)\ then\ y := 1\ else\ (Fact(x-1,y);\ y := x * y)$
$Fact(x,y) = cond(eq(x, C(0)), assign(y, C(1)), seq(Fact(sub(x, C(1)), y), assign(y, mul(x, V(y)))))$ ✌

In analoger Weise kann die funktionale Sprache FPF (vgl. 3.2/3.7) um rekursive Programme für partielle Funktionen erweitert werden (vgl. [66], 6.3):

Beispiel 10.1.11 (CPO-Semantik der funktionalen Sprache FPF) Sei Σ die abstrakte Syntax von FPF (vgl. 3.7). Die in Bsp. 4.1.5 definierte Σ-Algebra A ist monoton, sofern wir die Gleichheit auf A_{const} als Halbordnung auf dieser Menge wählen und A_{fun} zu $[A_{const,\bot} \stackrel{\bot}{\to} A_{const,\bot}]$ erweitern. Als Element dieser Menge wird eine partielle Funktion f auf A_{const} erweitert um die Definitionen $f(\bot) = \bot$ und $f(a) = \bot$ für alle $a \in A_{const}$, für die f nicht definiert ist. Ein rekursives FPF-Programm ist gegeben durch eine Menge $\Phi = \{F_1 : \to fun, \ldots, F_k : \to fun\}$ von Konstanten und ein Gleichungssystem

$$Decl(\Phi) \;=\; \{F_1 = t_1, \ldots, F_k = t_k\}$$

über $\Sigma \cup \Phi$. Z.B. lautet die Fakultätsfunktion als rekursives FPF-Programm in konkreter bzw. abstrakter Syntax wie folgt:

$fact = if\ eq \circ [id, \equiv 0]\ then\ \equiv 1\ else\ * \circ [id, fact \circ - \circ [id, \equiv 1]]$
$fact = cond(comp(eq, prod(id, C(0))), C(1), comp(mul, prod(id, comp(fact, comp(sub, prod(id, C(1)))))))$

Da A monoton ist und B_{fun} eine Teilmenge von $[A_{const,\bot} \Rightarrow A_{const,\bot}]$, ist für $t = (t_1, \ldots, t_k)$ die Funktion $t_\Phi : B^k_{fun} \to B^k_{fun}$ nach Satz 10.1.9 stetig, hat also nach Satz 10.1.4 den kleinsten Fixpunkt $lfp(t_\Phi) = sup\{t^i_\Phi(\Omega)\}_{i\in\mathbb{N}}$, der das Gleichungssystem $Decl(\Phi)$ in $(F_1, \ldots, F_k)$ löst. ✌

FPF ist übrigens nicht so mächtig wie Lisp, Haskell oder andere heutige funktionale Sprachen. Die als Daten verwendeten Funktionen von FPF, d.h. die Elemente von A_{fun}, sind nämlich immer nur *erster* Ordnung.

10.2 Zusicherungslogik

Dieser Vorläufer der Modallogik (vgl. §9.1) zur Verifikation imperativer Programme basiert auf der (auf weitere Sprachkonstrukte ausgedehnten) CPO-Semantik von IPF (vgl. 10.1.10). Entscheidend ist dort die Definition von *State* als (flache Erweiterung der) Menge $[X \to \mathbb{Z}]$ von **Speicherzuständen**. Zusicherungslogik ist eine Modallogik (vgl. 9.1.1), deren Transitionssystem aus Speicherzuständen und den Aktionen einer imperativer Programmiersprache, z.B. IPF, besteht:

Definition 10.2.1 (Zusicherungslogik) Sei Σ eine Signatur, die die abstrakte Syntax von IPF, $\Sigma(IPF)$, enthält (vgl. 3.8) und A eine Σ-Struktur, deren $\Sigma(IPF)$-Redukt (s. §6.2) mit der CPO-Semantik von IPF übereinstimmt (s. 10.1.10), wobei A_{exp} jetzt ein beliebiger Datenbereich ist und *State* dementsprechend durch $A^X =_{def} [X \to A_{exp}]_\perp$ definiert ist.

Eine **Zusicherung** ist ein Ausdruck der Form $\{F\}C\{G\}$, wobei F und G Σ-Formeln über X (!) sind und C ein Term der Sorte *com* ist. Der **Wert** F^A **von** F ist definiert als Menge aller $b : X \to A$ mit $b \models F$ (s. 4.1.8). C ist **partiell korrekt bzgl.** (F, G) oder A **erfüllt** $\{F\}C\{G\}$, geschrieben: $A \models \{F\}C\{G\}$, falls $C^A(F^A) \subseteq G^A \cup \{\perp\}$.[66] C ist **total korrekt** bzgl. (F, G), geschrieben: $A \models_{tot} \{F\}C\{G\}$, falls $C^A(F^A) \subseteq G^A$ (C^A "terminiert" auf F^A).

Der **Hoare-Kalkül** besteht aus folgenden Inferenzregeln zur Transformation von Zusicherungen und Σ-Formeln über X.

Zuweisungsregel
$$\frac{\{F\}\ assign(x,t)\ \{G(x)\}}{F \Rightarrow G(t)} \Uparrow$$

Sequenzregel
$$\frac{\{F\}\ seq(C,C')\ \{H\}}{\{F\}\ C\ \{G\},\quad \{G\}\ C'\ \{H\}} \Uparrow$$

Verzweigungsregel
$$\frac{\{F\}\ cond(B,C,C')\ \{G\}}{\{F \wedge B = true\}\ C\ \{G\},\quad \{F \wedge B = false\}\ C'\ \{G\}} \Uparrow$$

Schleifenregel
$$\frac{\{F\}\ loop(B,C)\ \{G\}}{F \Rightarrow I,\quad \{I \wedge B = true\}\ C\ \{I\},\quad (I \wedge B = false) \Rightarrow G} \Uparrow \text{❑}$$

Man beachte, dass B ein Boolescher Ausdruck, genauer gesagt: ein Term der Sorte *boolexp* ist und keine prädikatenlogische Formel.

Satz 10.2.2 (Der Hoare-Kalkül ist korrekt) Sei A wie in Def. 10.2.1. Jede Regel des Hoare-Kalküls ist korrekt bzgl. A, d.h. die jeweilige Prämisse ist gültig in A, sofern die jeweiligen Konklusionen in A gültig sind. ❑

Lemma 10.2.3 (Korrektheit der Schleifenregel) Seien Σ und A wie in Def. 10.2.1. Für $B \in T_\Sigma(X)_{boolexp}$ bezeichne $\underline{B}^A$ die Menge $\{b \in A^X \mid b \models B = true\}$ und $\overline{B}^A$ die Menge $\{b \in A^X \mid b \models B = false\}$. Zu zeigen ist:

(i) Gilt $F^A \subseteq I^A$, $C^A(I^A \cap \underline{B}^A) \subseteq I^A \cup \{\perp\}$ und $(I^A \cap \overline{B}^A) \subseteq G^A$, dann auch $loop^A(B^A, C^A)(F^A) \subseteq G^A \cup \{\perp\}$.

Es gelte die Prämisse von (i). Nach Def. von $loop^A$ (vgl. 10.1.10) und wegen $F^A \subseteq I^A$ läßt sich die Konklusion von (i) durch Scott-Induktion bzgl. der folgenden Formel P zeigen, da P nach Lemma 10.2.4 (s.u.) zulässig ist:

$$P(\alpha) \quad =_{def} \quad \alpha(B^A, C^A)(I^A) \subseteq G^A \cup \{\perp\}.$$

Da t_{loop} (vgl. 10.1.10) stetig ist und $loop^A$ der kleinste Fixpunkt von t_{loop} ist, folgt die Konklusion von (i) nach Satz 10.1.5 aus (ii) und (iii):

(ii) $\Omega \models P$,

(iii) $\alpha \models P$ impliziert $t_{loop}(\alpha) \models P$.

(ii) gilt wegen $\Omega(B^A, C^A)(I^A) = \{\perp\}$. Es gelte $\alpha \models P$, also $\alpha(B^A, C^A)(I^A) \subseteq G^A \cup \{\perp\}$. Zu zeigen ist $t_{loop}(\alpha) \models P$, also:

(iv) $t_{loop}(\alpha)(B^A, C^A)(I^A) \subseteq G^A \cup \{\perp\}$.

[66] Zur Def. von C^A vgl. 4.1.6.

Sei also $b \in I^A$. Aus der Def. von t_{loop} (vgl. 10.1.10) folgt:

$$t_{loop}(\alpha)(B^A, C^A)(b) = \begin{cases} \alpha(B^A, C^A)(C^A(b)) & \text{falls } B^A(b) = true \\ b & \text{falls } B^A(b) = false \\ \bot & \text{sonst.} \end{cases}$$

Fall 1: $B^A(b) = true$. Dann ist $b \in \underline{B}^A$, also $C^A(b) \in I^A \cup \{\bot\}$ (Prämisse von (i)). Die Def. von t_{loop} liefert

$$t_{loop}(\alpha)(B^A, C^A)(b) = \alpha(B^A, C^A)(C^A(b)) \in \alpha(B^A, C^A)(I^A \cup \{\bot\}) \subseteq G^A \cup \{\bot\}.$$

Fall 2: $B^A(b) = false$. Dann ist $b \in \overline{B}^A$, also $b \in G^A$ (Prämisse von (i)). Die Def. von t_{loop} liefert

$$t_{loop}(\alpha)(B^A, C^A)(b) = b \in G^A.$$

Fall 3: $B^A(b) = \bot$. Dann ist $t_{loop}(\alpha)(B^A, C^A)(b) = \bot$.

In allen drei Fällen ist also $b \in G^A \cup \{\bot\}$, womit (iv) bewiesen ist. ❑

Satz 10.2.4 (zulässige Formeln) Sei A eine monotone Σ-Algebra. Eine Σ-Formel $t \leq u$ mit $t, u \in T_\Sigma(X)$, ist zulässig, wenn für alle Funktionssymbole $f \in \Sigma$, die in $t \leq u$ vorkommen, f^A stetig ist.

Beweis. Sei $\{a_i\}_{i\in\mathbb{N}}$ eine Kette von A so, daß $a_i \models F$ für alle $i \in \mathbb{N}$ gilt. Dann ist $t^A(a_i) = a_i^*(t) \leq a_i^*(u) = u^A(a_i)$ (vgl. 4.1.6). Also ist $sup\{u^A(a_i)\}_{i\in\mathbb{N}}$ eine obere Schranke aller $t^A(a_i)$, $i \in \mathbb{N}$. Daraus folgt

$$sup\{t^A(a_i)\}_{i\in\mathbb{N}} \leq sup\{u^A(a_i)\}_{i\in\mathbb{N}},$$

da $sup\{u^A(a_i)\}_{i\in\mathbb{N}}$ die *kleinste* obere Schranke ist. Da für alle Funktionssymbole $f \in \Sigma$, die in $t \leq u$ vorkommen, f^A stetig ist und Kompositionen stetiger Funktionen wieder stetig sind, erhält man

$$(sup\{a_i\}_{i\in\mathbb{N}})^*(t) = t^A(sup\{a_i\}_{i\in\mathbb{N}}) = sup\{t^A(a_i)\}_{i\in\mathbb{N}} \leq sup\{u^A(a_i)\}_{i\in\mathbb{N}} = u^A(sup\{a_i\}_{i\in\mathbb{N}}) = (sup\{a_i\}_{i\in\mathbb{N}})^*(u).$$

Also gilt $sup\{a_i\}_{i\in\mathbb{N}} \models F$. ❑

Das Prädikat *partial* mit

$$partial(f : \mathbb{N} \to \mathbb{N}_\bot) \iff_{def} \exists n \in \mathbb{N} : f(n) = \bot$$

ist beispielsweise *nicht* zulässig, weil *partial* zwar für jede Kette $\{f_i : \mathbb{N} \to \mathbb{N}_\bot\}$ mit $f_i(n) = \bot \Leftrightarrow n \geq i$, aber nicht für deren Supremum gilt.

Ein Kalkül ist **vollständig** bzgl. einer Klasse C von Strukturen, wenn man alle in allen (!) $A \in \mathcal{C}$ gültigen Formeln mit seinen Regeln ableiten kann. Ein solcher Kalkül erfreut den Informatiker, sieht er darin doch die Chance, die Gültigkeit von Formeln zu entscheiden. Leider liefert aber die Vollständigkeit noch kein Entscheidungsverfahren, sondern nur — falls die Regelmenge endlich ist — ein *Aufzählungs-* oder *Semi-Entscheidungsverfahren* für gültige Formeln. Man kann die Vollständigkeit eines Kalküls häufig durch Induktion über der Struktur der gültigen Formeln beweisen.

Der Hoare-Kalkül ist vollständig bzgl. jeder Signatur Σ und jeder Struktur A im Sinne von Def. 10.2.1, sofern man für alle Programme C und Nachbedingungen G die *schwächste Vorbedingung* von (C, G) formulieren kann:

Definition und Satz 10.2.5 (Cook's Theorem: Vollständigkeit des Hoare-Kalküls) Seien Σ und A wie in Def. 10.2.1, $C \in T_\Sigma(X)_{com}$ und G eine Σ-Formel. Die **schwächste Vorbedingung** bzw. **schwächste liberale Vorbedingung** von (C, G) ist eine Σ-Formel F mit $F^A = (C^A)^{-1}(G^A)$ bzw. $F^A = (C^A)^{-1}(G^A \cup \{\bot\})$.

A heißt **ausdrucksfähig**, wenn für alle $C \in T_\Sigma(X)_{com}$ und Σ-Formeln G die schwächste liberale Vorbedingung von (C, G) existiert, intuitiv: wenn alle schwächsten liberalen Vorbedingungen in A formulierbar sind.

Ist A ausdrucksfähig und gilt $A \models \{F\}C\{G\}$, dann ist $\{F\}C\{G\}$ aus in A gültigen Σ-Formeln mit den Regeln des Hoare-Kalküls ableitbar.

Beweis durch Induktion über dem Aufbau von C (vgl. [61], Thm. 8.11). ❑

10.3 Reduktionssemantik rekursiver Programme

Definition 10.3.1 Sei $\longrightarrow$ eine binäre Relation auf einer Menge A. Dann bezeichnet $\xrightarrow{+}$ den transitiven und $\xrightarrow{*}$ den reflexiv-transitiven Abschluß von $\longrightarrow$. Sei $a \in A$. $b \in A$ heißt **Redukt von a bzgl.** $\longrightarrow$, falls $a \xrightarrow{*} b$ gilt. b ist eine **Normalform von a bzgl.** $\longrightarrow$, wenn es außerdem kein $c \in A$ mit $b \longrightarrow c$ gibt. $\longrightarrow$ heißt **schwach normalisierend**, wenn jedes $a \in A$ eine Normalform bzgl. $\longrightarrow$ hat. $\longrightarrow$ heißt **stark normalisierend**, wenn $\longrightarrow^{-1}$ wohlfundiert ist.

$\longrightarrow$ heißt **konfluent**, wenn für alle $a, b, c \in A$ mit $a \xrightarrow{*} b$ und $a \xrightarrow{*} c$ ein $d \in A$ mit $b \xrightarrow{*} d$ und $c \xrightarrow{*} d$ gibt. $\longrightarrow$ heißt **stark konfluent**, wenn für alle $a, b, c \in A$ mit $a \longrightarrow b$ und $a \longrightarrow c$ ein $d \in A$ mit $b \longrightarrow d$ und $c \longrightarrow d$ gibt. $\longrightarrow$ heißt **lokal konfluent**, wenn für alle $a, b, c \in A$ mit $a \longrightarrow b$ und $a \longrightarrow c$ ein $d \in A$ mit $b \xrightarrow{*} d$ und $c \xrightarrow{*} d$ gibt. ❑

Satz 10.3.2 Seien $\longrightarrow$ und $\Longrightarrow$ zwei binäre Relationen auf einer Menge A, deren reflexiv-transitive Abschlüsse übereinstimmen.

(i) Ist $\Longrightarrow$ stark konfluent, dann ist $\longrightarrow$ konfluent.

(ii) Ist $\longrightarrow$ lokal konfluent und stark normalisierend, dann ist $\longrightarrow$ konfluent.

(iii) Ist $\longrightarrow$ konfluent, dann hat jedes Element von A höchstens eine Normalform bzgl. $\longrightarrow$.

Beweis. (i) zeigt man durch Induktion über die Anzahl der $\longrightarrow$-Schritte, aus denen sich zwei Reduktionen $a \longrightarrow b$ und $a \longrightarrow c$ zusammensetzen. (ii) erhält man durch Noethersche Induktion entlang $\longrightarrow$ (vgl. §7.2). Zu (iii): Hat a zwei Normalformen b und c, dann gilt $a \longrightarrow b$ und $a \longrightarrow c$. Ist $\longrightarrow$ konfluent, dann gibt es ein $d \in A$ mit $b \xrightarrow{*} d$ und $c \xrightarrow{*} d$, was der Voraussetzung widerspricht, daß b und c Normalformen bzgl. $\longrightarrow$ sind. ❑

Definition 10.3.3 (Reduktionsregeln für rekursive Programme) Sei Σ eine Signatur mit Sortenmenge S, A eine monotone Σ-Algebra, $\Sigma' = \Sigma \cup \{A_s \mid s \in S\}$, $\Phi = \{F_1 : v_1 \to w_1, \ldots, F_k : v_k \to w_k\}$ (s. 10.1.8) und

$$Decl(\Phi) = \{F_1(x_1) = t_1, \ldots, F_k(x_k) = t_k\}$$

ein Gleichungssystem über $\Sigma \cup \Phi$. $\longrightarrow_\Phi$ bezeichnet die von folgenden Reduktionsregeln erzeugte Reduktionsrelation (vgl. 5.1.10):

unfold-Regeln $\quad F_i(x_i) \longrightarrow t_i \quad$ für alle $1 \le i \le k$

branch-Regeln $\quad if\ true\ then\ x\ else\ y \longrightarrow x$
$\qquad\qquad\qquad\quad if\ false\ then\ x\ else\ y \longrightarrow y$

simplify-Regeln $\quad op(a) \longrightarrow op^A(a) \quad$ für alle $op : w \to s \in \Sigma \setminus \Phi \setminus \{if\_then\_else\}$ und $a \in A_w$ ❑

Diese Regeln liefern eindeutige Normalformen, was sich nur mithilfe von 10.3.2(i)/(iii) zeigen läßt, weil $\longrightarrow_\Phi$ wegen der beliebigen Rekursion in $Decl(\Phi)$ i.a. nicht normalisierend ist.

Wir definieren eine parallele Reduktionsrelation $\Longrightarrow$ auf $T_{\Sigma'}(X)$ wie folgt:

- $t \Longrightarrow t$.
- $t \longrightarrow_\Phi t \quad \Rightarrow \quad t \Longrightarrow t$.
- $t_1 \Longrightarrow t'_1, \ldots, t_n \Longrightarrow t'_n \quad \Rightarrow \quad op(t_1, \ldots, t_n) \Longrightarrow op(t'_1, \ldots, t'_n) \quad$ für alle $op : w \to s \in \Sigma$.

$\Longrightarrow$ ist stark konfluent und es gilt $\xrightarrow{*}{=}\xrightarrow{*}$. Also kann nach Satz 10.3.2(i)/(iii) jedem normalisierbaren $t \in T_{\Sigma'}(X)$ seine Normalform als Wert zugeordnet werden.

Definition 10.3.4 (Reduktionssemantik rekursiver Programme) Es gelten die Voraussetzungen von 10.3.3. Sei $t \in T_{\Sigma'}$. $nf(t)$ bezeichne die Normalform von t bzgl. $\longrightarrow_\Phi$. An den Reduktionsregeln von $\longrightarrow_\Phi$ erkennt man sofort, daß $nf(t)$ ein Element von $A \setminus \{\bot\}$ ist. Für alle $1 \leq i \leq k$ ist die **Reduktionssemantik von** F_i definiert als Funktion $F_i^{red(A)} : A_{u_i} \to A_{v_i}$ mit

$$F_i^{red(A)}(a) =_{def} \begin{cases} nf(F_i(a)) & \text{falls } nf(F_i(a)) \text{ existiert} \\ \bot & \text{sonst } \square \end{cases}$$

Für jeden Term t, der eine Normalform hat, kann diese durch folgende Reduktionsstrategie aus t abgeleitet werden: branch- und simplify-Regel werden stets vor der unfold-Regel angewendet und bei deren Anwendung wird der am weitesten links stehende Redex ersetzt. Man zeigt das, indem man beweist, daß diese **call-by-name-Strategie** *sicher* ist (s. 10.3.6).

Definition 10.3.5 (Reduktionsstrategie) Es gelten die Voraussetzungen von 10.3.3.

Zunächst definieren wir für alle $t \in T_\Sigma$ die Funktion t_Φ wie in 10.1.8 mit dem einzigen Unterschied, daß t_Φ jetzt für *jedes einzelne Vorkommen* eines Symbols von Φ in t ein entsprechendes funktionales Argument hat. Tritt also in t genau k-mal ein Symbol von Φ auf, dann hat t_Φ k funktionale Argumente $f_1, \ldots, f_k$, auch wenn mehreren davon dasselbe Symbol von Φ entspricht.

Eine **Berechnungsfolge von** $t \in T_{\Sigma'}$ ist eine Kette $\{t_i\}_{i\in\mathbb{N}}$ von $T_{\Sigma'}$ bzgl. $\xrightarrow{+}_\Phi$ mit $t_0 = t$. Eine **Reduktionsstrategie** ist eine Funktion RS, die jedem $t \in T_{\Sigma'}$ eine Berechnungsfolge von t zuordnet. RS kann wiederum auf eine spezielle Strategie $Unfold(RS)$ abgebildet werden, die nur die $unfold$-Schritte von RS ausführt, d.h. für alle $t \in T_{\Sigma'}$ und $i \in \mathbb{N}$ gilt:

(i) $Unfold(RS)(t)_{i+1}$ entsteht aus $Unfold(RS)(t)_i$ durch Anwendung der $unfold$-Regel auf $Unfold(RS)(t)_i$,

(ii) $RS(t)_{i,\Phi} = Unfold(RS)(t)_{i,\Phi}$.

(ii) bedeutet, daß die im i-ten Schritt von RS bzw. $Unfold(RS)$ erzeugten Terme semantisch äquivalent sind. Für alle $1 \leq j \leq k$ ist die RS**-Semantik von** F_j definiert als Funktion $F_j^{RS} : A_{v_j} \to A_{w_j}$ mit

$$F_j^{RS}(a) =_{def} sup\{RS(F_j(a))_{i,\Phi}(\Omega)\}_{i\in\mathbb{N}}. \ \square$$

Sei RS eine beliebige Reduktionsstrategie, $t \in T_{\Sigma'}$ und $i \in \mathbb{N}$. Da für alle $op : w \to s \in \Sigma$, op^A monoton ist, gilt

$$Unfold(RS)(t)_{i,\Phi}(\Omega) \leq Unfold(RS)(t)_{i+1,\Phi}(\Omega),$$

also wegen (ii) auch

(iii) $RS(t)_{i,\Phi}(\Omega) \leq RS(t)_{i+1,\Phi}(\Omega)$.

Diejenige Reduktionsstrategie RS, bei der $RS(t)_{i+1}$ aus $RS(t)_i$ dadurch entsteht, daß die $unfold$-Regel auf alle Vorkommen eines Symbols von Φ in $RS(t)_i$ gleichzeitig angewendet wird, heißt **full-substitution-Strategie** und wird mit FS bezeichnet. Ein Reduktionsschritt gemäß der full-substitution-Strategie entspricht genau einer Anwendung der Funktion t_Φ. Für alle $1 \leq j \leq k$ und $a \in A_{v_j}$ ist nämlich

(iv) $FS(F_j(a))_{i,\Phi}(\Omega) = proj_j(t_\Phi^i(\Omega))(a)$,

d.h. der Wert des von der full-substitution-Strategie nach i Schritten aus $F_j(a)$ abgeleiteten Terms $FS(F_j(a))_i$ ist gerade die auf a angewendete F_j-Komponente von $t_\Phi^i(\Omega)$. Aus (iv) erhält man sofort

(v) $F_j^{FS}(a) = proj_j(lfp(t_\Phi))(a)$.

Außerdem ist FS die "schnellste" Strategie, d.h. für jede Reduktionsstrategie RS gilt

$$Unfold(RS)(F_j(a))_i \xrightarrow{*}_\Phi FS(F_j(a))_i,$$

also wegen (ii) und (iii)

$$RS(F_j(a))_{i,\Phi}(\Omega) \leq FS(F_j(a))_{i,\Phi}(\Omega).$$

Daraus folgt wegen (v)

(vi) $F_j^{RS}(a) \leq proj_j(lfp(t_\Phi))(a)$.

Definition 10.3.6 Eine Reduktionsstrategie heißt **Fixpunktstrategie**, wenn für alle $1 \leq j \leq k$ $F_j^{RS} = F_j^A =_{def} proj_j(lfp(t_\Phi))$ gilt.

Sei $1 \leq j \leq k$, $a \in A_{u_j}$ und $RS(F_j(a)) = \{t_i\}_{i\in\mathbb{N}}$. O.B.d.A. seien die Argumente $f_1, \ldots, f_k$ von $t_{i,\Phi}$ so angeordnet, daß die beim Reduktionsschritt von t_i nach t_{i+1} ersetzten Symbole von Φ den ersten r Komponenten von $(f_1, \ldots, f_k)$ entsprechen (vgl. 10.3.5). RS ist eine **sichere Strategie**, wenn für alle $i \in \mathbb{N}$ gilt: Enthält t_i Symbole aus Φ, dann ist

$$t_{i,\Phi}(\Omega, \ldots, \Omega, f_{r+1}, \ldots, f_n) = \bot. \quad \square$$

Ist RS sicher und t ein "undefinierter Term", dann terminiert die mit RS durchgeführte Reduktion von t nicht.

Satz 10.3.7 (Äquivalenz von Reduktions- und Fixpunktsemantik rekursiver Programme) Sei RS eine Fixpunktstrategie, $1 \leq j \leq k$ und $a \in A_{v_j}$. Hat $F_j(a)$ eine Normalform bzgl. $\longrightarrow_\Phi$, dann wird diese von RS erreicht, d.h.

$$F_j^{RS}(a) = F_j^{red(A)}(a).$$

Beweis. Sei nf die Normalform von $F_j(a)$ bzgl. $\longrightarrow_\Phi$. Angenommen, nf wird von RS nicht erreicht. Dann enthält jeder Term von $RS(F_j(a))$ mindestens ein Symbol von Φ. Also ist $F_j^{RS}(a) = \bot$. Da nf aber existiert, gibt es eine andere Strategie RS', die nf erreicht, d.h. $F_j^{RS'}(a) = nf \neq \bot$. Aus (vi) folgt ein Widerspruch:

$$\bot \neq nf = F_j^{RS'}(a) \leq F_j^A(a) = F_j^{RS}(a) = \bot. \quad \square$$

Satz 10.3.8 ([65], Theorem 5-4) Eine sichere Reduktionsstrategie RS ist eine Fixpunktstrategie.

Beweis. Angenommen, RS ist keine Fixpunktstrategie. Dann gibt es $1 \leq j \leq k$, $a \in A_{v_j}$ und $i \in \mathbb{N}$ so, daß einerseits $F_j^A(a) = proj_j(\tau_\Phi^i(\Omega))(a) \neq \bot$ und andererseits $F_j^{RS}(a) = \bot$ gilt. Sei $RS(F_j(a)) = \{u_r\}_{r\in\mathbb{N}}$ und $Unfold(RS)(F_j(a)) = \{t_r\}_{r\in\mathbb{N}}$. Aus (iv) folgt

$$FS(F_j(a))_{i,\Phi}(\Omega) \neq \bot.$$

Wir stellen uns die mit der *unfold*-Regel aus $F_j(a)$ ableitbaren Terme als Bäume vor und setzen o.B.d.A. voraus, daß jede Gleichung von $Decl(\Phi)$ die Form $F(x) = g(F_{i_1}(t_1), \ldots, F_{i_n}(t_n))$ hat. Sei $t =_{def} FS(F_j(a))_i$. Jeder Weg von der Wurzel von t zum ersten Symbol von Φ hat die Länge i. Wegen $F_j^{RS}(a) = \bot$ existiert außerdem $r \in \mathbb{N}$ derart, daß t_r Symbole aus Φ enthält, beim Reduktionsschritt von t_r nach t_{r+1} aber nur noch solche Teilterme ersetzt werden, deren Wurzel mindestens i Kanten von der Wurzel von t_r entfernt ist, weil es auf kürzeren Wegen keine Symbole von Φ mehr gibt. Ersetzt man die restlichen Teilterme non t_r, deren Wurzel ein Symbol von Φ ist, durch t, dann erhält man einen Term t'_r, in dem jedes Symbol von Φ mindestens i Kanten von der Wurzel entfernt ist. Aus der Konstruktion von t'_r folgt, daß dieser Term durch Anwendungen der *unfold*-Regel aus t ableitbar ist, so daß $t_\Phi(\Omega) \leq t'_{r,\Phi}(\Omega)$ gilt, was schließlich einen Widerspruch zur Sicherheit von RS impliziert:

$$\bot \neq t_\Phi(\Omega) \leq t'_{r,\Phi}(\Omega) = t_{r,\Phi}(\Omega, \ldots, \Omega, t_\Phi(\Omega), \ldots, t_\Phi(\Omega)) \overset{(ii)}{=} u_{r,\Phi}(\Omega, \ldots, \Omega, t_\Phi(\Omega), \ldots, t_\Phi(\Omega)) = \bot. \quad \square$$

Mit dem Sicherheitskriterium läßt sich nun zeigen, daß die o.g. call-by-name-Strategie eine Fixpunktstrategie ist:

Sei $1 \leq j \leq k$, $a \in A_{u_j}$, $RS(F_j(a)) = \{t_i\}_{i\in\mathbb{N}}$ und $i \in \mathbb{N}$ so, daß t_i mindestens ein Symbol von Φ enthält. Ein Term t heiße **simplifiziert**, wenn die branch- oder simplify-Regel auf t nicht anwendbar ist. $F_j(a)$ ist simplifiziert, so daß nach Definition der call-by-name-Strategie auch t_i simplifiziert ist. Deshalb gilt einer der folgenden drei Fälle:

Fall 1: $t_i = F_j(u_1, \ldots, u_m)$ für ein $1 \leq j \leq k$ und $u_1, \ldots, u_m \in T_{\Sigma'}$. Dann ersetzt RS genau dieses Vorkommen von F_j im Schritt von t_i nach t_{i+1}. Also gilt

$$t_{i,\Phi}(\Omega, \ldots, \Omega, f_{r+1}, \ldots, f_n) = \Omega(\Omega, \ldots, \Omega, f_{r+1}, \ldots, f_n) = \bot.$$

Fall 2: $t_i = if\ p\ then\ u\ else\ v$ für $p, u, v \in T_{\Sigma'}$. Da t_i simplifiziert ist, enthält p Symbole von Φ. Da RS das am weitesten links stehende dieser Symbole im Schritt von t_i nach t_{i+1} ersetzt und alle Funktionen von p "oberhalb" dieses Symbols strikt sind[67], gibt es $a, b \in A$ mit

$$t_{i,\Phi}(\Omega, \ldots, \Omega, f_{r+1}, \ldots, f_n) = if^A \bot\ then\ a\ else\ b = \bot.$$

Fall 3: $t_i = op(u_1, \ldots, u_m)$ für ein $op \in \Sigma \setminus \Phi \setminus \{if\_then\_else\}$ und $u_1, \ldots, u_m \in T_{\Sigma'}$. Da t_i mindestens ein Symbol von Φ enthält, gilt das auch für $(u_1, \ldots, u_m)$. Das am weitesten links stehende dieser Symbole gehöre zu u_j. Da RS genau dieses Symbol im Schritt von t_i nach t_{i+1} ersetzt und alle Funktionen von u_j "oberhalb" dieses Symbols strikt sind[68], gibt es $a_1, \ldots, a_m \in A$ mit $a_j = \bot$ und

$$t_{i,\Phi}(\Omega, \ldots, \Omega, f_{r+1}, \ldots, f_n) = op^A(a_1, \ldots, a_m) = \bot.$$

11 λ-Terme, Typen und Bereiche

Der λ-Kalkül ist die Basis aller Programmiersprachen, in denen Funktionen beliebiger Ordnung als Objekte definiert und verwendet werden können. FPF (vgl. 4.1.5) hat diese Eigenschaft nicht. Deren Objekte sind Konstanten (die Elemente von B_{const}) sowie Funktionen erster Ordnung (die Elemente von B_{fun}). IPF erlaubt überhaupt keine Funktionen als Objekte. Zwar werden die Sorten *exp*, *boolexp* und *com* als Funktionenräume interpretiert (vgl. 9.2.7). Ein Term der Sorte *exp*, *boolexp* oder *com* kann in einem IPF-Programm jedoch nicht als Darstellung einer Funktion verwendet werden.

Man spricht vom *λ-Kalkül*, weil die Semantik dieser Sprache zunächst über ein Regelsystem definiert wurde. Später stellte man dann die Frage, welche semantischen Bereiche durch λ-Ausdrücke repräsentiert werden.

11.1 Typen und λ-Kalkül

Definition 11.1.1 (Typen) Sei S eine Menge von Sorten und TX eine Menge von **Typvariablen**[69]. Die Menge $Type_S(TX)$ der **Typen über** S ist induktiv definiert:

- Jede Sorte von S ist ein Typ über S.
- Jede Typvariable von TX ist ein Typ über S.

[67] Genaugenommen kann unter diesen Funktionen wieder das (nicht-strikte) Konditional sein. Dann steht das zu ersetzende Symbol von Φ aber im Booleschen Argument des Konditionals, weil t_i simplifiziert ist.
[68] dto.
[69] i.a. griechische Buchstaben

- Für alle $t, t', t_1, \ldots, t_n \in Type_S(TX)$ und $\alpha \in TX$ sind auch $t_1 + \cdots + t_n$, $t_1 \times \cdots \times t_n$, $t \to t'$, $\mu\alpha t$, $\forall\alpha t$ und $t_{\alpha \leftarrow t'}$[70] Typen über S.

Eine Typvariable α **kommt in** einem Typ t **gebunden vor**, falls t einen Teiltyp der Form $\mu\alpha t'$ oder $\forall\alpha t'$ enthält. $\alpha \in TX$ ist eine **freie Variable von** t, wenn α in t nicht nur gebunden vorkommt. Ein Typ heißt **geschlossen**, wenn er keine freien Variablen enthält. Die Menge der geschlossenen Typen über S bezeichnen wir mit $CLType_S(TX)$. ❑

Definition 11.1.2 (Typmodell) Sei S eine Menge von Sorten und TX eine Menge von Typvariablen. Ein **Typmodell** A **über** S ist eine $CLType_S(TX)$-sortierte Menge mit folgenden Eigenschaften:

(i) $A_{t_1+\cdots+t_n} \cong A_{t_1} \uplus \cdots \uplus A_{t_n}$[71] und $A_{t_1\times\cdots\times t_n} \cong A_{t_1} \times \ldots \times A_{t_n}$ für alle $t_1, \ldots, t_n \in CLType_S(TX)$,

(ii) $\emptyset \neq A_{t\to t'} \subseteq [A_t \to A_{t'}]$ für alle $t, t' \in CLType_S(TX)$,

(iii) $A_{\mu\alpha t} \cong A_{t_{\alpha\leftarrow\mu\alpha t}}$ für alle $\alpha \in TX$ und $t \in CLType_S(TX)$,

(iv) $\emptyset \neq A_{\forall\alpha t} \subseteq \cap\{A_{t_{\alpha\leftarrow t'}} \mid t' \in CLType_S(TX)\}$ für alle $\alpha \in TX$ und $t \in CLType_S(TX)$. ❑

Daß bei (ii) und (iv) nur Inklusion und nicht Isomorphie verlangt wird, hängt ausschließlich mit der Interpretation polymorpher Typen $\forall\alpha t$ zusammen, für die man bisher kein "besseres" Modell gefunden hat (vgl. 5.5). Nichtleer sollten $A_{t\to t'}$ und $A_{\forall\alpha t}$ allerdings sein, weil sonst alle Formeln mit freien Variablen des Typs $t \to t'$ bzw. $\forall\alpha t$ in A gültig wären.

Definition 11.1.3 (polymorphe Signatur und Algebra) Sei TX eine Menge von Typvariablen. Eine **polymorphe Signatur** ist ein Paar $\Sigma = (S, C)$, bestehend aus einer Menge von Sorten und einer $CLType_S(TX)$-sortierten Menge C von **Konstanten**. Anstelle von $c \in C_t$ schreiben wir $c : t \in C$.

Eine **polymorphe** Σ**-Algebra** ist ein Paar (A, C^A), bestehend aus einem Typmodell A über S und einer Interpretation C^A von C in A, d.h. für alle $c : t \in C$ gibt es $c^A \in A_t$. ❑

Eine polymorphe Signatur $\Sigma = (S, C)$ mit

$$C_t \neq \emptyset \iff t \in S \vee \exists s_1, \ldots, s_n, s \in S : t = (s_1 \times \cdots \times s_n \to s)$$

entspricht einer Signatur im Sinne von Def. 3.5.

Definition 11.1.4 (λ-Terme) Sei $\Sigma = (S, C)$ eine polymorphe Signatur und X eine Menge von **Individuenvariablen**. Die Menge $\Lambda_\Sigma(X)$ der **λ-Terme über** Σ **und** X ist induktiv definiert:

- $C \cup X \subseteq \Lambda_\Sigma(X)$,
- $e, e', e_1, \ldots, e_n \in \Lambda_\Sigma(X)$, $x \in X \Rightarrow \lambda x.e$, $\mu x.e$, $(e\ e')$, $(e_1, \ldots, e_n)$, $e_{x\leftarrow e'} \in \Lambda_\Sigma(X)$.

$e_{x\leftarrow e'}$ bezeichnet denjenigen Term, der aus e entsteht, wenn alle in e gebundenen Vorkommen freier Variablen von e' in neue Variablen umbenannt werden und anschließend jedes Vorkommen von x in e durch e' ersetzt wird (vgl. §6.5). ❑

Im Gegensatz zur Sortierung von Termen über monomorphen Signaturen ist die Typisierung von λ-Termen nicht durch den Aufbau der Terme festgelegt (vgl. 3.9). Man kann typ-inkonsistente, nicht **typisierbare** λ-Terme bilden. Typisierbar ist ein Term genau dann, wenn sein Typ mit den folgenden Inferenzregeln ableitbar ist.

[70] $t_{\alpha\leftarrow t'}$ bezeichnet den Typ t'', der aus t entsteht, indem man dort alle gebundenen Vorkommen freier Variablen von t' in neue Variablen umbenennt und dann jedes Vorkommen von α durch t' ersetzt.

[71] disjunkte Vereinigung

Definition 11.1.5 (typisierbare λ-Terme) Sei $\Sigma = (S, C)$ eine polymorphe Signatur, X eine Menge von Individuenvariablen und TX eine Menge von Typvariablen. Eine **Typannahme** ist eine partielle Funktion $\sigma : X \to Type_S(TX)$. Man schreibt σ auch als Menge der Ausdrücke $x : t$ mit $\sigma(x) = t$. Ein **Typurteil** ist ein Ausdruck der Form $\sigma \vdash e : t$, wobei σ eine Typannahme, e ein λ-Term über X und t ein Typ über S ist. $\sigma \vdash e : t$ soll bedeuten, daß unter der Annahme σ der Term e den Typ t hat.

Der **Curry-Kalkül** zur Ableitung von Typurteilen besteht aus aus folgenden **Inferenzregeln**: Sei σ ein Typzustand, $e, e', e_1, \ldots, e_n \in \Lambda_\Sigma(X)$ und $t, t', t_1, \ldots, t_n \in Type_S(TX)$, $x \in X$ und $\alpha \in TX$.

Konstantenaxiome $\sigma \vdash c : t$ für alle $c : t \in C$

Variablenaxiome $\sigma \vdash x : t$ falls $\sigma(x) = t$

Tupelregel $$\frac{\sigma \vdash (e_1, \ldots, e_n) : t_1 \times \cdots \times t_n}{\sigma \vdash e_1 : t_1, \ldots, \sigma \vdash e_n : t_n} \Uparrow$$

Abstraktionsregeln $$\frac{\sigma \vdash \lambda x.e : t \to t'}{\sigma \cup \{x : t\} \vdash e : t'} \Uparrow \qquad \frac{\sigma \vdash \mu x.e : t \to t'}{\sigma \cup \{x : t \to t'\} \vdash e : (t \to t') \to (t \to t')} \Uparrow$$

Applikationsregel $$\frac{\sigma \vdash (e\ e') : t'}{\sigma \vdash e : t \to t',\quad \sigma \vdash e' : t} \Uparrow$$

Substitutionsregel $$\frac{\sigma \vdash e'_{x \leftarrow e} : t'}{\sigma \vdash e : t,\quad \sigma \cup \{x : t\} \vdash e' : t'} \Uparrow$$

Datentypregeln $$\frac{\sigma \vdash e : \mu\alpha t}{\sigma \vdash e : t_{\alpha \leftarrow \mu\alpha t}} \Uparrow \qquad \frac{\sigma \vdash e : t_{\alpha \leftarrow \mu\alpha t}}{\sigma \vdash e : \mu\alpha t} \Uparrow$$

Polymorphieregeln $$\frac{\sigma \vdash e : \forall\alpha t}{\sigma \vdash e : t} \Uparrow \text{ falls } \alpha \text{ nicht frei in } \sigma(X) \qquad \frac{\sigma \vdash e : t_{\alpha \leftarrow t'}}{\sigma \vdash e : \forall\alpha t} \Uparrow$$

Ein λ-Term e heißt **typisierbar**, wenn mit dem Curry-Kalkül ein Typurteil der Form $\emptyset \vdash e : t$ ableitbar ist. ❑

Definition 11.1.6 (Reduktionsregeln des λ-Kalküls) Sei $\Sigma = (S, C)$ eine polymorphe Signatur und (A, C^A) eine polymorphe Σ-Algebra. Die Reduktionsregeln des λ-Kalküls lauten wie folgt: Sei $e, e' \in \Lambda_\Sigma(X)$ und $x \in X$.

β-Regel $(\lambda x e\ e') \longrightarrow e_{x \leftarrow e'}$ falls keine freien Variablen von e' in e gebunden vorkommen

η-Regel $\lambda x(e\ x) \longrightarrow e$ falls x keine freie Variable von e ist

δ-Regel $(c\ e) \longrightarrow c^A(e)$ für alle $c \in C$ mit $c^A(e) \in \Lambda_\Sigma(X)$ ❑

Mit der δ-Regel können nicht-strikte Standardfunktionen von C^A (vgl. 10.1.6) **lazy ausgewertet**, d.h. auf ggf. nur unvollständig ausgewertete Argumente angewendet werden. Alternativ kann man strikte und nicht-strikte Funktionen — wie das zur Definition rekursiver Funktionen i.a. benötigte Konditional if_then_else — auch durch λ-Terme definieren. So erhält man mit folgenden Definitionen die übliche Bedeutung von if_then_else, d.h. die Terme $if\ true\ then\ e\ else\ e'$ und $if\ false\ then\ e\ else\ e'$ lassen sich mit der β-Regel zu e bzw. e' reduzieren:

- $true =_{def} \lambda x(\lambda y x)$
- $false =_{def} \lambda x(\lambda y y)$
- $if\_then\_else =_{def} \lambda x(\lambda y(\lambda z((x\ y)z)))$

Tatsächlich sind alle berechenbaren Funktionen λ-definierbar (vgl. [20], Theorem 11.44).

$\longrightarrow_\lambda$ bezeichnet im folgenden die von den Reduktionsregeln des λ-Kalküls erzeugte Reduktionsrelation. Man beachte, daß *diese* Reduktionsregeln keine Variablen im Sinne der vorangegangenen Definition enthalten, d.h. im Falle des λ-Kalküls läßt sich die erzeugte Reduktionsrelation ohne Verwendung von Substitutionen definieren:

- $e \longrightarrow_\lambda e' \iff_{def}$ Es gibt $c \in \Lambda_\Sigma(X)$, $(l,r) \in R$ und $x \in var(c)$ mit $e = c_{x\leftarrow l}$ und $e' = c_{x\leftarrow r}$.

$\longrightarrow_\lambda$ ist weder stark noch schwach normalisierend. Z.B. hat der Term $(\lambda x(xx))(\lambda x(xx))$ keine Normalform, weil er mit der β-Regel zu sich selbst reduzierbar ist. Zwei Ergebnisse sind jedoch festzuhalten:

- Für jeden λ-Term e, der eine Normalform hat, kann diese durch folgende **Reduktionsstrategie** aus e abgeleitet werden: Die η- und δ-Regeln werden vor der β-Regel angewendet und bei deren Anwendung wird der am weitesten links stehende Redex ersetzt (**Satz von Curry**).
- Jeder typisierbare λ-Term hat eine Normalform.

Da $\longrightarrow_\lambda^{-1}$ nicht wohlfundiert ist, läßt sich die Eindeutigkeit von $\longrightarrow_\lambda$-Normalformen nicht mithilfe der Konfluenzkriterien aus §5.2 zeigen. Stattdessen definiert man induktiv die **parallele Ein-Schritt-Relation** $\Longrightarrow$: Sei $e, e', e_1, \ldots, e_n, e'_1, \ldots, e'_n, f, f' \in \Lambda_\Sigma(X)$ und $x \in X$.

- $e \Longrightarrow e$.
- $e \Longrightarrow e'$ impliziert $\lambda x e \Longrightarrow \lambda x e'$.
- $e \Longrightarrow e'$ und $f \Longrightarrow f'$ impliziert $(e\ f) \Longrightarrow (e'\ f')$.
- $e_1 \Longrightarrow e'_1, \ldots, e_n \Longrightarrow e'_n$ impliziert $(e_1, \ldots, e_n) \Longrightarrow (e'_1, \ldots, e'_n)$.
- $e \Longrightarrow e'$ und $f \Longrightarrow f'$ impliziert $\lambda x(e\ f) \Longrightarrow e'_{x\leftarrow f'}$, falls x eine freie Variable von e und e' ist.
- $e \Longrightarrow e'$ impliziert $\lambda x(e\ x) \Longrightarrow e'$, falls x keine freie Variable von e ist.
- $e \Longrightarrow e'$ impliziert $c^A(e) \Longrightarrow c^A(e')$ für alle $c \in C$ mit $c^A(e) \in \Lambda_\Sigma(X)$.

Satz 11.1.7 ($\Longrightarrow$ ist stark konfluent) Sei $\Longrightarrow$ wie oben definiert.

(i) $e \Longrightarrow e'$ und $f \Longrightarrow f'$ impliziert $e_{x\leftarrow f} \Longrightarrow e'_{x\leftarrow f'}$ für alle $x \in X$.

(ii) $\Longrightarrow$ ist stark konfluent, also: $e \Longrightarrow e'$ und $e \Longrightarrow f'$ impliziert $e' \Longrightarrow f$ und $f' \Longrightarrow f$ für ein $f \in \Lambda_\Sigma(X)$.

Beweis. Beide Behauptungen zeigt man durch Induktion über den Aufbau der jeweiligen Prämisse. (i) wird im Beweis von (ii) benötigt. Liegt beispielsweise der Fall

$$\lambda x(e\ f) \Longrightarrow e'_{x\leftarrow f'} \quad \text{und} \quad \lambda x(e\ f) \Longrightarrow e''_{x\leftarrow f''}$$

vor, dann gibt es nach Induktionsvoraussetzung λ-Terme e''' und f''' mit $e' \Longrightarrow e'''$, $e'' \Longrightarrow e'''$, $f' \Longrightarrow f'''$ und $f'' \Longrightarrow f'''$. Daraus folgt wegen (i):

$$e'_{x\leftarrow f'} \Longrightarrow e'''_{x\leftarrow f'''} \quad \text{und} \quad e''_{x\leftarrow f''} \Longrightarrow e'''_{x\leftarrow f'''}. \ \square$$

Normalformen von λ-Termen sind also eindeutig, so daß mit den Voraussetzungen von 11.1.6 jedem normalisierbaren $e \in \Lambda_\Sigma(X)$ seine Normalform als Wert zugeordnet werden kann:

Definition 11.1.8 (Reduktionssemantik von λ-Termen) Sei $\Sigma = (S, C)$ eine polymorphe Signatur und (A, C^A) eine polymorphe Σ-Algebra. Die **Reduktionssemantik** $e^{red(A)}$ eines λ-Terms e ist definiert als die Normalform von e bzgl. $\longrightarrow_\lambda$, falls sie existiert. Andernfalls ist $e^{red(A)}$ undefiniert. $\square$

11.2 Rekursive Bereiche

Die Anforderungen an ein Typmodell (vgl. 11.1.2) werden natürlich nicht von jeder beliebigen Interpretation der einzelnen Typen erfüllt. Insbesondere für **funktionale Typen** der Form $t \to t'$, **rekursive Typen** der Form $\mu\alpha t$ und **polymorphe Typen** der Form $\forall\alpha t$ bedarf es einigen modell-theoretischen Aufwandes, um überhaupt Mengen $A_{t\to t'}$, $A_{\mu\alpha t}$ und $A_{\forall\alpha t}$ zu finden, die 11.1.2(ii), (iii) bzw. (iv) erfüllen.[72]

Beginnen wir mit den rekursiven Typen. Der geforderte Isomorphismus $A_{\mu\alpha t} \cong A_{t_{\alpha\leftarrow\mu\alpha t}}$ ähnelt einer Gleichung $F = t$ eines rekursiven FPF-Programms (vgl. 10.1.11). So wie man deren Lösung als Fixpunkt der Funktion $t(f) =_{def} t^A_{F\leftarrow f}$ konstruiert, geht man auch bei der Lösung von **Bereichsgleichungen** zwischen Typen vor, da $A_{\mu\alpha t} \cong A_{t_{\alpha\leftarrow\mu\alpha t}}$ gerade bedeutet, daß $A_{\mu\alpha t}$ eine Lösung der Gleichung $\alpha = t$ in α ist. Die Funktion τ, deren Fixpunkt eine solche Lösung wäre, ist, grob gesagt, durch $\tau(C) = t^A_{\alpha\leftarrow C}$ definiert. Da hier jedes Element C im Definitionsbereich von τ eine **Menge** ist, wird die gesamte CPO-Theorie (vgl. §2.2) auf die Ebene von Mengensystemen oder **Kategorien** gehoben. τ wird dabei zum **Funktor**.

Definition 11.2.1 (Kategorie, Funktor) Eine **Kategorie** $\mathcal{K}$ besteht aus

- einer Klasse $Obj(\mathcal{K})$ von $\mathcal{K}$**-Objekten**,
- zu je zwei $\mathcal{K}$-Objekten A, B einer Menge $\mathcal{K}(A, B)$ von $\mathcal{K}$**-Morphismen von** A **nach** B,
- einer assoziativen **Komposition**

$$\circ : \mathcal{K}(A, B) \times \mathcal{K}(B, C) \longrightarrow \mathcal{K}(A, C)$$

$$(f, g) \longmapsto g \circ f$$

- und zu jedem $\mathcal{K}$-Objekt A einer **Identität** $id_A \in \mathcal{K}(A, A)$, so daß für alle $B \in Obj(\mathcal{K})$ und $f \in \mathcal{K}(A, B)$ $f \circ id_A = f = id_B \circ f$ gilt.

Ist $\mathcal{K}$ gegeben, dann schreibt man $f : A \to B$ anstelle von $f \in \mathcal{K}(A, B)$. A heißt **Quellobjekt** und B **Zielobjekt** von f. $f : A \to B$ ist ein **Isomorphismus**, wenn es ein $g : B \to A$ mit $f \circ g = id_B$ und $g \circ f = id_A$ gibt. A und B sind **isomorph**, geschrieben: $A \cong B$, falls es einen Isomorphismus $f : A \to B$ oder, was äquivalent ist, einen Isomorphismus $f : B \to A$ gibt.

Set bezeichnet die Kategorie, deren Objekte Mengen und deren Morphismen Abbildungen sind. Für alle $n \geq 1$ bezeichnet $\mathbf{Set}^n$ die Kategorie, deren Objekte n-Tupel von Mengen und deren Morphismen n-Tupel von Abbildungen sind. ❑

Definition 11.2.2 (Funktor) Seien $\mathcal{K}$ und $\mathcal{L}$ zwei Kategorien. Ein **Funktor** $F : \mathcal{K} \to \mathcal{L}$ bildet jedes $\mathcal{K}$-Objekt auf ein $\mathcal{L}$-Objekt und jeden $\mathcal{K}$-Morphismus $f : A \to B$ auf einen $\mathcal{L}$-Morphismus $F(f) : F(A) \to F(B)$ ab derart, daß für alle $\mathcal{K}$-Objekte A und alle komponierbaren $\mathcal{K}$-Morphismen f, g gilt:

- $F(id_A) = id_{F(A)}$,
- $F(g \circ f) = F(g) \circ F(f)$. ❑

Beispiel 11.2.3 (map-Funktor) Die Haskell-Funktion $map :' a \to' b \to' a\ list \to' b\ list$ ist ein Funktor von der Kategorie Set der Mengen in die Kategorie Mon der Monoide: Für alle $A \in Set$ und alle Abbildungen $f : A \to B$ ist

$$map(A) =_{def} (A^*, ++, nil)$$

$$map(f)([a_1, \ldots, a_n]) =_{def} [f(a_1), \ldots, f(a_n)]$$

[72] Im ersten und zweiten Fall folgen wir i.w. [60], im dritten i.w. [64] (vgl. auch [108], §15.5).

$Set(A,B)$ ist also die Menge aller Abbildungen von A nach B, während $Mon(C,D)$ als Menge aller Monoid-Homomorphismen vom Monoid $(C,*^C,1^C)$ ins Monoid $(D,*^D,1^D)$ definiert wird. 🐰

Definition 11.2.4 (initiales Objekt, ω-Diagramm, Cokegel, Colimes, ω-Kategorie) Sei $\mathcal{K}$ eine Kategorie. Ein $\mathcal{K}$-Objekt I heißt **initiales Objekt von** $\mathcal{K}$, falls für jedes $\mathcal{K}$-Objekt A genau ein $\mathcal{K}$-Morphismus $ini_A : I \to A$ existiert.

Eine Folge $\Delta = \{f_i : A_i \to A_{i+1}\}_{i\in\mathbb{N}}$ von $\mathcal{K}$-Morphismen heißt **ω-Diagramm von** $\mathcal{K}$. Eine Folge $\mu = \{\mu_i : A_i \to A\}_{i\in\mathbb{N}}$ von $\mathcal{K}$-Morphismen mit $\mu_i = \mu_{i+1} \circ f_i$ für alle $i \in \mathbb{N}$ heißt **Cokegel von** Δ. A heißt **Zielobjekt** des Cokegels μ.

Ein Cokegel $\nu = \{\nu_i : A_i \to C\}_{i\in\mathbb{N}}$ von Δ heißt **Colimes von** Δ, wenn für jeden Cokegel $\mu = \{\mu_i : A_i \to A\}_{i\in\mathbb{N}}$ von Δ genau ein $\mathcal{K}$-Morphismus $col_A : C \to A$ existiert mit $col_A \circ \nu_i = \mu_i$ für alle $i \in \mathbb{N}$.

$\mathcal{K}$ heißt **ω-Kategorie**, falls $\mathcal{K}$ ein initiales Objekt hat und jedes ω-Diagramm von $\mathcal{K}$ einen Colimes hat. ❑

Jede Kategorie hat *bis auf Isomorphie* höchstens ein initiales Objekt und höchstens einen Colimes eines ω-Diagramms.

Satz 11.2.5 (Konstruktion von Colimiten in Set**)** Ein Cokegel $\{\nu_i : A_i \to C\}_{i\in\mathbb{N}}$ eines ω-Diagramms $\Delta = \{f_i : A_i \to A_{i+1}\}_{i\in\mathbb{N}}$ von Set ist genau dann der Colimes von Δ, wenn C isomorph zum Quotienten der disjunkten Vereinigung aller A_i ist:

$$C \;\cong\; (\biguplus_{i\in\mathbb{N}} A_i)/\!\sim,$$

wobei $\sim$ die von allen Paaren $(f_i(a_i), a_i)$ erzeugte Äquivalenzrelation auf $A =_{def} \biguplus_{i\in\mathbb{N}} A_i$ ist. ν_i ist dann die Komposition aus der Einbettung von A_i in A und der natürlichen Abbildung $nat : A \to A/\!\sim$, d.h. $\nu_i(a) =_{def} [a]_\sim$ für alle $a \in A_i$.[73] ❑

Definition 11.2.6 (ω-Funktor) Seien $\mathcal{K}$ und $\mathcal{L}$ zwei ω-Kategorien. Sei $\nu = \{\nu_i : A_i \to C\}_{i\in\mathbb{N}}$ der Colimes eines ω-Diagramm $\Delta = \{f_i : A_i \to A_{i+1}\}_{i\in\mathbb{N}}$. Ein Funktor $F : \mathcal{K} \to \mathcal{L}$ heißt **ω-Funktor**, wenn F **Colimiten erhält**, d.h. wenn $F(\nu) = \{F(\nu_i) : F(A_i) \to F(C)\}_{i\in\mathbb{N}}$ der Colimes des ω-Diagramms $F(\Delta) = \{F(f_i) : F(A_i) \to F(A_{i+1})\}_{i\in\mathbb{N}}$ ist. ❑

Man erkennt folgende Korrespondenzen zwischen ordnungstheoretischen Begriffen einerseits und kategorientheoretischen Begriffen andererseits:

Menge	Objektklasse
Halbordnung	Morphismenmenge
kleinstes Element	initiales Objekt
Kette	ω-Diagramm
obere Schranke	Cokegel
Supremum	Colimes
CPO	ω-Kategorie
Funktion	Funktor
stetige Funktion	ω-Funktor

Fehlt nur noch die kategorielle Version des Kleeneschen Fixpunktsatzes (vgl. 10.1.4), für deren Ableitung noch der Begriff der F-Algebra verwendet wird.

Definition 11.2.7 (F**-Algebra)** Sei $F : \mathcal{K} \to \mathcal{K}$ ein Funktor und A ein $\mathcal{K}$-Objekt. Eine **F-Algebra** ist ein $\mathcal{K}$-Morphismus $\alpha : F(A) \to A$. F-Algebren bilden die Kategorie $\mathbf{Alg}_F$. Ein Alg_F-Morphismus von der F-Algebra $\alpha : F(A) \to A$ in die F-Algebra $\beta : F(B) \to B$ ist ein $\mathcal{K}$-Morphismus $f : A \to B$ mit $f \circ \alpha = \beta \circ F(f)$. ❑

Angenommen, Alg_F hat eine initiales Objekt $\iota : F(I) \to I$. Dann gibt es einen Alg_F-Morphismus

$$f \;:\; [F(I) \xrightarrow{\iota} I] \;\longrightarrow\; [F(F(I)) \xrightarrow{F(\iota)} F(I)].$$

[73] Diese Colimeskonstruktion kann man entsprechend auch auf Diagramme anwenden, die keine ω-Diagramme sind.

Also gilt $\iota \circ f \circ \iota = \iota \circ F(\iota) \circ F(f) = \iota \circ F(\iota \circ f)$. Wegen $id_I \circ \iota = \iota = \iota \circ id_{F(I)} = \iota \circ F(id_I)$ sind id_I und $\iota \circ f$ zwei Alg_F-Morphismen von ι nach ι. Wegen der Eindeutigkeit eines solchen Morphismus gilt $id_I = \iota \circ f$. Da f ein Alg_F-Morphismus ist, gilt dann auch $f \circ \iota = F(\iota) \circ F(f) = F(\iota \circ f) = F(id_I) = id_{F(I)}$. Also ist ι ein Isomorphismus.

Damit haben wir gezeigt:

Lemma 11.2.8 (Initiale Algebren sind Isomorphismen) Sei $F : \mathcal{K} \to \mathcal{K}$ ein Funktor derart, daß die Kategorie Alg_F ein initiales Objekt $\iota : F(I) \to I$ hat. Dann sind I und $F(I)$ isomorph. ❑

Satz 11.2.9 (Kategorieller Fixpunktsatz) Sei $\mathcal{K}$ eine ω-Kategorie, J das initiale Objekt von $\mathcal{K}$ und $F : \mathcal{K} \to \mathcal{K}$ ein ω-Funktor. Dann gibt es ein Objekt I in $\mathcal{K}$ mit $F(I) \cong I$, das das Zielobjekt sowohl der initialen F-Algebra als auch des Colimes des ω-Diagramms $\{F^i(ini_{F(J)})\}_{i\in\mathbb{N}}$ ist.

Beweis. Lemma 11.2.8 und folgender Satz. ❑

Satz 11.2.10 (ω-Funktoren liefern initiale Algebren) ([60], Theorem 2.1; [95], Lemma 3.4.4) Sei $\mathcal{K}$ eine ω-Kategorie, J das initiale Objekt von $\mathcal{K}$ und $F : \mathcal{K} \to \mathcal{K}$ ein ω-Funktor. Dann hat Alg_F ein initiales Objekt $\iota : F(I) \to I$. Dabei ist I das Zielobjekt des Colimes $\{\nu_i : F^i(J) \to I\}_{i\in\mathbb{N}}$ des ω-Diagramms

$$\{F^i(ini_{F(J)}) : F^i(J) \to F^{i+1}(J)\}_{i\in\mathbb{N}}$$

und $F(I)$ das Zielobjekt des Cokegels $\{F(\nu_i) : F^{i+1}(J) \to F(I)\}_{i\in\mathbb{N}}$ des ω-Diagramms $\{F^{i+1}(ini_{F(J)})\}_{i\in\mathbb{N}}$, der als Bild eines Colimes unter dem ω-Funktor F selbst Colimes ist. Damit erhält man ι als eindeutigen $\mathcal{K}$-Morphismus von $F(I)$ nach I mit $\iota \circ F(\nu_i) = \nu_{i+1}$ für alle $i \in \mathbb{N}$.

Beweis. Es bleibt zu zeigen, dass ι initial in Alg_F ist. Sei $\alpha : F(A) \to A$ eine F-Algebra. Da J initial in $\mathcal{K}$ ist, hat das ω-Diagramm $\{F^i(ini_{F(J)})\}_{i\in\mathbb{N}}$ den Cokegel $\{F(\mu_i) : F^i(J) \to A\}_{i\in\mathbb{N}}$ mit $\mu_0 = ini_A$ und $\mu_i = \alpha \circ F(\mu_{i-1})$ für alle $i > 0$. Da $\{\nu_i : F^i(J) \to I\}_{i\in\mathbb{N}}$ der Colimes von $\{F^i(ini_{F(J)})\}_{i\in\mathbb{N}}$ ist, gibt es einen eindeutigen $\mathcal{K}$-Morphismus von $col_A : I \to A$ mit $col_A \circ \nu_i = \mu_i$ für alle $i \in \mathbb{N}$. Man erhält

$$col_A \circ \iota \circ F(\nu_i) = col_A \circ \nu_{i+1} = \mu_{i+1} = \alpha \circ F(\mu_i) = \alpha \circ F(col_A \circ \nu_i) = \alpha \circ F(col_A) \circ F(\nu_i),$$

also

$$col_A \circ \iota = \alpha \circ F(col_A), \tag{1}$$

weil $\{F(\nu_i) : F^{i+1}(J) \to F(I)\}_{i\in\mathbb{N}}$ ein Colimes ist und es deshalb nur einen $\mathcal{K}$-Morphismus $\gamma : F(I) \to A$ mit $\gamma \circ F(\nu_i) = col_A \circ \iota \circ F(\nu_i)$ gibt. Wegen (1) ist col_A ein Alg_F-Morphismus. Ist col_A der einzige, der (1) erfüllt? Ja, weil col_A der einzige $\mathcal{K}$-Morphismus ist, der $col_A \circ \nu_i = \mu_i$ erfüllt. Für alle $\theta : I \to A$ mit $\theta \circ \iota = \alpha \circ F(\theta)$ gilt nämlich $\theta \circ \nu_0 = ini_A = \mu_0$ und

$$\theta \circ \nu_{i+1} = \theta \circ \iota \circ F(\nu_i) = \alpha \circ F(\theta) \circ F(\nu_i) = \alpha \circ F(\theta \circ \nu_i) \overset{Ind.vor.}{=} \alpha \circ F(\mu_i) = \mu_{i+1},$$

also $\theta \circ \nu_i = \mu_i$ für alle $i \in \mathbb{N}$. ❑

Um rekursive Typen zu interpretieren, könnten wir also von einer beliebigen ω-Kategorie ausgehen. Da wir es aber mit bestimmten "Typkonstruktoren" zu tun haben, aus denen sich ein rekursiver Typ zusammensetzt, bleibt die Frage, auf welcher ω-Kategorie gerade diese Typkonstruktoren als ω-Funktoren interpretiert werden können. Gemäß 11.1.1 haben wir es mit den Typkonstruktoren $+$ (disjunkte Vereinigung oder **Summe**), $\times$ (Produkt) und $\to$ (Funktionskonstruktor) zu tun. In Haskell werden rekursive Typen wie folgt definiert:

$$\texttt{data}\ \alpha\ =\ con_1\ typ_1 \mid \ldots \mid con_n\ typ_n$$

In der Notation von 11.1.1 lautet diese Definition wie folgt:

$$\mu\alpha(typ_1 + \cdots + typ_n).$$

Die Konstruktoren $con_1, \ldots, con_n$ sind aus abstrakter Sicht nur dazu da, um die Summanden $typ_1, \ldots, typ_n$ voneinander "disjunkt zu machen".

Etwas allgemeiner betrachten wir endliche Mengen DT von Typen der Form[74]

$$DT = \{\mu\beta_1.t_1, \ldots, \mu\beta_k.t_k\},$$

wobei für alle $1 \leq i \leq k$ weder $\forall$ noch μ in t_i vorkommt. In Haskell entspricht DT der Datentypdefinition

$$\begin{array}{lrcl} \texttt{data} & comp(\beta_1) & = & comp(t_1) \\ & \ldots & & \\ \texttt{data} & comp(\beta_k) & = & comp(t_k), \end{array}$$

wobei $comp$ definiert ist durch:

- $comp(t_1 \times \cdots \times t_n) = comp(t_1) * \cdots * comp(t_n)$
- $comp(t_1 + \cdots + t_n) = con_1 \texttt{ of } comp(t_1) \mid \ldots \mid con_n \texttt{ of } comp(t_n)$
- $comp(t \to t') = comp(t) \to comp(t')$
- $comp(\beta_i) = (\alpha_{i1}, \ldots, \alpha_{in_i})\ \beta_i$ für alle $1 \leq i \leq k$
- $comp(\alpha) = \alpha$ für alle Typvariablen außer $\beta_1, \ldots, \beta_k$
- $comp(s) = s$ für alle $s \in S$

und $\alpha_{i1}, \ldots, \alpha_{in_i}$ die in t_i vorkommenden Typvariablen außer $\beta_1, \ldots, \beta_k$ sind.

Definition 11.2.11 (rekursiver Datentyp) Sei S eine Menge von Sorten. Eine wie oben definierte Menge $DT = \{\mu\beta_1.t_1, \ldots, \mu\beta_k.t_k\}$ von Typen über S derart, daß für alle $1 \leq i \leq k$ weder $\forall$ noch μ in t_i vorkommt, heißt **rekursiver Datentyp über** S. $t_1, \ldots, t_n$ sind die **Rumpftypen** von DT. Alle in DT vorkommenden Typvariablen außer $\beta_1, \ldots, \beta_k$ heißen **Parametervariablen** von DT. ❑

Zu jeder Kategorie $\mathcal{K}$ bezeichnet $\mathcal{K}^{op}$[75] die zu $\mathcal{K}$ **duale Kategorie**. Sie ergibt sich aus $\mathcal{K}$ durch "Umkehrung der Pfeile", d.h. $Obj(\mathcal{K}^{op}) =_{def} Obj(\mathcal{K})$ und für alle $A, B \in Obj(\mathcal{K})$, $\mathcal{K}^{op}(A, B) =_{def} \mathcal{K}(B, A)$.

Definition 11.2.12 (Typkonstruktoren als Funktoren auf Set) Wir definieren die Typkonstruktoren $+$ und $\times$ als Funktoren auf Set bzw. Set^{op}.

Summe $+ : Set \times Set \to Set$

$$A + B =_{def} A \uplus B$$

$$(f : A \to A', g : B \to B') \quad \stackrel{+}{\longmapsto} \quad f + g : A + B \to A' + B'$$

$$(f + g)(c) =_{def} \begin{cases} f(c) & \text{falls } c \in A \\ g(c) & \text{falls } c \in B \end{cases}$$

Produkt $\times : Set \times Set \to Set$

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

$$(f : A \to A', g : B \to B') \quad \stackrel{\times}{\longmapsto} \quad f \times g : A \times B \to A' \times B'$$

$$(f \times g)(a, b) =_{def} (f(a), g(b))$$

[74] Statt $\forall\alpha(t)$, $\mu\alpha(t)$ bzw. $\lambda x(t)$ schreibt man auch $\forall\alpha.t$, $\mu\alpha.t$ bzw. $\lambda x.t$.
[75] $\mathcal{K}$ *opposite*

Funktionskonstruktor $\to$: $Set^{op} \times Set \to Set$

$$A \to B =_{def} \text{Menge der Abbildungen von } A \text{ nach } B \ (= \ Set(A,B))$$

$$(f : A' \to A, g : B \to B') \ \stackrel{\to}{\longmapsto} \ f \to g : (A \to B) \to (A' \to B')$$

$$(f \to g)(h : A \to B) =_{def} g \circ h \circ f : A' \to B' \quad \square$$

Der Funktionskonstruktor ist hier auf *zwei* Kategorien definiert und daher kein ω-Funktor. Es gilt aber:

Satz 11.2.13 Set und Set^n (vgl. 11.2.1) sind ω-Kategorien. $+$ und $\times$ und deren Kompositionen sind ω-Funktoren auf Set.

Beweis. Das n-Tupel $(\emptyset, \ldots, \emptyset)$ ist das initiale Objekt von Set^n. Colimiten von ω-Diagrammen in Set^n werden analog zu denen in Set konstruiert (vgl. 11.2.5). $\square$

Warum ist eine F-Algebra eigentlich eine Algebra? Weil F-Algebren Verallgemeinerungen der (Mengen-)Algebren von 4.1.2 sind. Sei Σ eine Signatur mit Sortenmenge $S = \{s_1, \ldots, s_n\}$. Dann gibt es den Funktor $T : Set^n \to Set^n$ mit

$$T(A_{s_1}, \ldots, A_{s_n}) \ =_{def} \ (\biguplus_{op_1 : w_1 \to s_1 \in \Sigma} A_{w_1}, \ldots, \biguplus_{op_n : w_n \to s_n \in \Sigma} A_{w_n})$$

und einer Σ-Algebra A entspricht die T-Algebra $\delta : T(A_{s_1}, \ldots, A_{s_n}) \to (A_{s_1}, \ldots, A_{s_n})$ mit

$$\delta((op_1, a_1), \ldots, (op_n, a_n)) \ =_{def} \ (op_1^A(a_1), \ldots, op_n^A(a_n)).$$

Als Komposition der Funktoren $+$ und $\times$ auf Set ist T nach Satz 11.2.13 ein ω-Funktor. Also liefert Satz 11.2.9 ein Objekt I in Set^n mit $T(I) \cong I$, das das Zielobjekt sowohl der initialen T-Algebra als auch des Colimes des ω-Diagramms $\{T^i(ini_{T(\emptyset,\ldots,\emptyset)})\}_{i \in \mathbb{N}}$ ist. Aus der Konstruktion des Zielobjektes eines Colimes in Set^n folgt, daß I und T_Σ isomorph sind. Demzufolge heißt T_Σ auch **initiale Σ-Algebra**.

Korollar 11.2.14 Sei S eine Menge von Sorten, DT ein rekursiver Datentyp über S (vgl. 11.2.11), dessen Rumpftypen keinen Funktionskonstruktor enthalten, $\alpha_1, \ldots, \alpha_n$ die Parametervariablen von t_i, A eine $(S, \emptyset)$-Algebra und $B = (B_1, \ldots, B_n) \in Set^n$. Dann ist für alle Rumpftypen t von DT ein Funktor

$$t_{B,DT} \ : \ Set^k \ \longrightarrow \ Set$$

wie folgt induktiv definiert[76]: Sei $C = (C_1, \ldots, C_k) \in Set^k$.

- $(\alpha_i)_{B,DT}(C) =_{def} B_i$ für alle $1 \le i \le n$.
- $(\beta_i)_{B,DT}(C) =_{def} C_i$ für alle $1 \le i \le k$.
- $s_{B,DT}(C) =_{def} A_s$ für alle $s \in S$.
- $(t_1 + \cdots + t_n)_{B,DT}(C) =_{def} t_{1,B,DT}(C) + \cdots + t_{n,B,DT}(C)$.
- $(t_1 \times \ldots \times t_n)_{B,DT}(C) =_{def} t_{1,B,DT}(C) \times \ldots \times t_{n,B,DT}(C)$.

$t_{B,DT}$ ist ein ω-Funktor, so daß für die k Rumpftypen $t_1, \ldots, t_k$ von DT auch

$$\tau_{B,DT} \ : \ Set^k \ \longrightarrow \ Set^k$$

mit

$$\tau_{B,DT}(C) \ =_{def} \ (t_{1,B,DT}(C), \ldots, t_{k,B,DT}(C))$$

[76] Die Definition von $t_{B,DT}$ folgt einem ähnlichen Schema wie die Definition von t_Φ (vgl. 10.1.8)

ein ω-Funktor ist.

Beweis. Da sich $t_{B,DT}$ und $\tau_{B,DT}$ aus den Funktoren $+$ und $\times$ zusammensetzen, folgt die Behauptung aus Satz 11.2.13. ❑

Der kategorielle Fixpunktsatz 11.2.9 liefert nun für alle Mengen B und $\tau = \tau_{B,DT}$ ein Objekt $Dom_{B,DT}$ in Set^k mit $\tau(Dom_{B,DT}) \cong Dom_{B,DT}$, das mit dem Zielobjekt des Colimes des ω-Diagramms

$$\{f_i\}_{i\in\mathbb{N}} =_{def} \{\tau^i(ini_{T(\emptyset)}) : \tau^i(\emptyset) \to \tau^{i+1}(\emptyset)\}_{i\in\mathbb{N}}$$

übereinstimmt. Wir nennen $Dom_{B,DT}$ die **Semantik von (B,DT)**.

Nach Satz 11.2.5 ist $Dom_{B,DT}$ isomorph zum Quotienten $(\biguplus_{i\in\mathbb{N}} \tau^i(\emptyset))/\sim$, wobei $\sim$ die von allen Paaren $(f_i(a_i), a_{i+1})$ erzeugte Äquivalenzrelation auf $\biguplus_{i\in\mathbb{N}} \tau^i(\emptyset)$ ist. Definiert man nun

- $A_{\alpha_i} =_{def} B_i$,
- $A_{\beta_i} =_{def} A_{\mu\beta_i t_i} =_{def} proj_i(Dom_{B,DT})$ (= i-te Komponente von $Dom_{B,DT}$),
- $A_{t_1+\cdots+t_n} =_{def} A_{t_1} + \cdots + A_{t_n}$,
- $A_{t_1\times\cdots\times t_n} =_{def} A_{t_1} \times \ldots \times A_{t_n}$,

dann gelten zumindest schon mal die Forderungen (i) und (iii) an ein Typmodell A (vgl. 11.1.2). Insbesondere erhält man

$$A_{\mu\beta_i t_i} = proj_i(Dom_{B,DT}) = proj_i(\tau(Dom_{B,DT})) = t_{i,B,DT}(Dom_{B,DT})$$
$$= (t_i)_{\beta_i \leftarrow \mu\beta_i t_i, B, DT}(Dom_{B,DT}) = A_{(t_i)_{\beta_i\leftarrow\mu\beta_i t_i}}.$$

Anders ausgedrückt, $Dom_{B,DT} = (proj_1(Dom_{B,DT}), \ldots, proj_k(Dom_{B,DT}))$ liefert eine Lösung des Gleichungssystems

$$\beta_1 = (t_1)_{B,DT}(\beta_1, \ldots, \beta_k), \quad \ldots, \quad \beta_k = (t_k)_{B,DT}(\beta_1, \ldots, \beta_k).$$

Beispiel 11.2.15 (endliche Listen) Sei

$$List \;=\; \{\mu list(1 + (\alpha \times list))\},$$

Σ eine Signatur und A eine Σ-Algebra mit $A_1 = \{nil\}$. Dann ist $Dom_{B,List}$ isomorph zur Menge aller Listen von Elementen aus B. Das ergibt sich aus der Darstellung von $Dom_{B,List}$ als Quotient $(\biguplus_{i\in\mathbb{N}} \tau^i_{B,List}(\emptyset))/\sim$, wobei $\sim$ die von allen Paaren $(f_i(a_i), a_{i+1})$ erzeugte Äquivalenzrelation auf $\biguplus_{i\in\mathbb{N}} \tau^i_{B,List}(\emptyset)$ ist (s.o.). Die **Approximation** $\tau^i_{B,List}(\emptyset)$ ist isomorph zur Menge aller endlichen Listen mit höchstens $i-1$ Elementen. Die Abbildung

$$\tau^i_{B,List}(\emptyset) \xrightarrow{f_i} \tau^{i+1}_{B,List}(\emptyset)$$

ist die Inklusion der Menge der Listen mit höchstens $i-1$ Elementen in die Menge der Listen mit höchstens i Elementen. Folglich enthält $Dom_{B,List}$ nur endliche Listen. 🐰

Beispiel 11.2.16 (endliche binäre Bäume) Sei

$$Bintree \;=\; \{\mu bintree(mt + (bintree \times \alpha \times bintree))\},$$

Σ eine Signatur und A eine Σ-Algebra mit $A_{mt} = \{mt\}$. Dann ist $Dom_{B,Bintree}$ isomorph zur Menge aller endlichen binären Bäume mit Knoteneinträgen aus B. Das ergibt sich wie bei $Dom_{B,List}$ aus der Darstellung von $Dom_{B,Bintree}$ als Quotient $(\biguplus_{i\in\mathbb{N}} \tau^i_{B,Bintree}(\emptyset))/\sim$, wobei $\sim$ die von allen Paaren $(f_i(a_i), a_{i+1})$ erzeugte Äquivalenzrelation auf $\biguplus_{i\in\mathbb{N}} \tau^i_{B,Bintree}(\emptyset)$ ist. Die Approximation $\tau^i_{B,Bintree}(\emptyset)$ ist isomorph zur Menge aller binären Bäume mit maximaler Tiefe $i-1$. 🐰

Beispiel 11.2.17 (endliche Bäume mit beliebigem endlichen Ausgrad) Sei

$$Tree \;=\; \{\mu tree(\alpha \times treelist),\ \mu treelist(1 + (tree \times treelist))\},$$

Σ eine Signatur und A eine Σ-Algebra mit $A_1 = \{nil\}$. Dann ist die *tree*- bzw. *treelist*-Komponente von $Dom_{B,Tree}$ isomorph zur Menge aller endlichen Bäume mit Knoteneinträgen aus B bzw. aller endlichen Listen solcher Bäume. Die Approximation $proj_{tree}(\tau^i_{B,Tree}(\{\bot\}))$ ist isomorph zur Menge aller Bäume mit maximaler Tiefe $i-1$, während

$$proj_{treelist}(\tau^i_{B,Tree}(\emptyset))$$

isomorph ist zur Menge aller Listen von höchstens $i-1$ Bäumen mit maximaler Tiefe $i-1$. Lehmann und Smyth zeigen, daß $Dom_{B,Bintree}$ und $proj_{treelist}(Dom_{B,Tree})$ isomorph sind ([60], Cor. 2). Das entspricht der bekannten Darstellung beliebiger Bäume als binäre Bäume (siehe Funktion `tree2bintree` in [77], Kap. 9). ✌

11.3 Funktionale Bereiche

Zur Definition der Semantik einer Sprache ohne Funktionskonstruktor und Polymorphie könnten wir uns demnach mit der ω-Kategorie *Set* begnügen. Die Verwendung von $\to$ macht es jedoch erforderlich, *Set* durch folgende ω-Kategorie zu ersetzen:

Definition 11.3.1 ($\mathbf{CPO}^E$) Die Objekte der Kategorie $\mathbf{CPO}^E$ sind alle CPOs mit kleinstem Element. Die Morphismen $f : A \to B$ von CPO^E sind alle Paare $f = (f^L : A \to B, f^R : B \to A)$ stetiger Funktionen mit $f^R \circ f^L = id_A$ und $f^L \circ f^R \leq id_B$. f^L heißt **Einbettung**, f^R **Retraktion** oder **Projektion**.[77] Die Morphismenkomposition in CPO^E wird auf die Komposition stetiger Funktionen zurückgeführt:

$$(f^L : B \to C, f^R : C \to B) \circ (g^L : A \to B, g^R : B \to A) \;=_{def}\; (f^L \circ g^L : A \to C, g^R \circ f^R : C \to A). \;\text{❑}$$

Proposition 11.3.2 Die Komponentenfunktionen f^L und f^R eines CPO^E-Morphismus $f : A \to B$ sind $\bot$-erhaltend (vgl. 10.1.1).

Beweis. Für alle $b \in B$ gilt $f^L(\bot^A) \leq f^L(f^R(b)) \leq b$, also $\bot^B = f^L(\bot^A)$ und damit auch $f^R(\bot^B) = f^R(f^L(\bot^A)) = \bot^A$. ❑

Colimesmorphismen in CPO^E sind durch eine Eigenschaft der Suprema ihrer Komponentenfunktionen charakterisiert:

Satz 11.3.3 (Konstruktion von Colimiten in CPO^E) Ein Cokegel $\{\mu_i : A_i \to C\}_{i\in\mathbb{N}}$ eines ω-Diagramms $\Delta = \{f_i : A_i \to A_{i+1}\}_{i\in\mathbb{N}}$ von CPO^E ist genau dann der Colimes von Δ, wenn gilt:

(i) $\mu^L_i \circ \mu^R_i \leq \mu^L_{i+1} \circ \mu^R_{i+1}$ für alle $i \in \mathbb{N}$,

(ii) $sup\{\mu^L_i \circ \mu^R_i\}_{i\in\mathbb{N}} = id_C$.

Das Zielobjekt C des Colimes von Δ ist gegeben durch die Menge aller Folgen $\{a_i\}_{i\in\mathbb{N}}$ mit $a_i = f^R_i(a_{i+1}) \in A_i$ für alle $i \in \mathbb{N}$. ❑

Daraus gewinnt man ein Kriterium für ω-Funktoren auf CPO^E:

Satz 11.3.4 (Kriterium für ω-Funktoren auf CPO^E) Sei $F : CPO^E \to CPO^E$ ein Funktor derart, daß für alle Folgen $\{\mu_i : A_i \to C\}_{i\in\mathbb{N}}$ von CPO^E-Morphismen mit (i), (ii) und

(iii) $F(\mu_i)^L \circ F(\mu_i)^R \leq F(\mu_{i+1})^L \circ F(\mu_{i+1})^R$ für alle $i \in \mathbb{N}$

[77] weshalb CPO^E in [60] *CPO-PR* genannt wird

auch

(iv) $sup\{F(\mu_i)^L \circ F(\mu_i)^R\}_{i\in\mathbb{N}} = id_{F(C)}$.

gilt, dann ist F ein ω-Funktor.

Beweis. Sei $\mu = \{\mu_i : A_i \to C\}_{i\in\mathbb{N}}$ der Colimes eines ω-Diagramms $\Delta = \{f_i : A_i \to A_{i+1}\}_{i\in\mathbb{N}}$. Nach Satz 11.3.3 gilt (i) und (ii). Da μ Colimes von Δ ist, gilt $\mu_i = \mu_{i+1} \circ f_i$ für alle $i \in \mathbb{N}$. Daraus folgt $F(\mu_i) = F(\mu_{i+1} \circ f_i) = F(\mu_{i+1}) \circ F(f_i)$, d.h. $\{F(\mu_i)\}_{i\in\mathbb{N}}$ ist ein Cokegel von $F(\Delta) = \{f_i : A_i \to A_{i+1}\}_{i\in\mathbb{N}}$ und es gilt nach Definition der Morphismenkomposition in CPO^E:

(v) $F(\mu_i)^L = F(\mu_{i+1})^L \circ F(f_i)^L$ und $F(\mu_i)^R = F(f_i)^R \circ F(\mu_{i+1})^R$.

Da $F(f_i)$ CPO^E-Morphismus ist, gilt $F(f_i)^L \circ F(f_i)^R \leq id$ für alle $i \in \mathbb{N}$. Damit folgt (iii) aus (v):

$$F(\mu_i)^L \circ F(\mu_i)^R = F(\mu_{i+1})^L \circ F(f_i)^L \circ F(f_i)^R \circ F(\mu_{i+1})^R \leq F(\mu_{i+1})^L \circ F(\mu_{i+1})^R,$$

also nach Voraussetzung auch (iv). Aus (iii) und (iv) folgt, wieder nach Satz 11.3.3, daß $\{F(\mu_i)\}_{i\in\mathbb{N}}$ der Colimes von $F(\Delta)$ ist. ❑

Mit diesem Kriterium läßt sich einfach zeigen, daß die Typkonstruktoren $+$, $\times$ und $\to$ als ω-Funktoren auf CPO^E darstellbar sind:

Definition 11.3.5 (Typkonstruktoren als Funktoren auf CPO^E) Wir definieren $+$, $\times$ und $\to$ zunächst als Funktoren auf der Kategorie **CPO** aller CPOs mit kleinstem Element als Objekte und aller stetigen und $\perp$-erhaltenden Funktionen als Morphismen.

Summe $+ : CPO \times CPO \to CPO$

$$A + B =_{def} (A \setminus \{\perp^A\}) \uplus (B \setminus \{\perp^B\}) \uplus \{\perp\}$$

$$(f : A \to A', g : B \to B') \quad \overset{+}{\longmapsto} \quad f + g : A + B \to A' + B'$$

$$(f + g)(c) =_{def} \begin{cases} f(c) & \text{falls } c \in A \\ g(c) & \text{falls } c \in B \\ \perp & \text{sonst} \end{cases}$$

Produkt $\times : CPO \times CPO \to CPO$

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

$$(f : A \to A', g : B \to B') \quad \overset{\times}{\longmapsto} \quad f \times g : A \times B \to A' \times B'$$

$$(f \times g)(a, b) =_{def} (f(a), g(b))$$

Funktionskonstruktor $\to : CPO^{op} \times CPO \to CPO$

$$A \to B =_{def} \text{Menge der } \perp\text{-erhaltenden stetigen Funktionen von } A \text{ nach } B\ (=\ CPO(A, B))$$

$$(f : A' \to A, g : B \to B') \quad \overset{\to}{\longmapsto} \quad f \to g : (A \to B) \to (A' \to B')$$

$$(f \to g)(h : A \to B) =_{def} g \circ h \circ f : A' \to B'$$

Produkt und Funktionskonstruktor sind so definiert, daß CPO eine **monoidale Kategorie** wird, d.h. für alle $A, B, C \in CPO$ gibt es eine Bijektion zwischen der Menge der Morphismen von $A \times B$ nach C und der Menge der Morphismen von A nach $B \to C$ (Currying!):

$$CPO(A \times B, C) \;\cong\; CPO(A, B \to C)$$

(vgl. [104], S. 775). Da die Morphismenzuordnung von $\to$ aber immer noch Morphismen aus zwei verschiedenen Kategorien verlangt, gehen wir jetzt von CPO nach CPO^E über und erweitern die Funktoren $+$, $\times$ und $\to$ auf CPO zu Funktoren $\oplus$, $\otimes$ und $\ominus$ auf CPO^E:

Summe $\oplus : CPO^E \times CPO^E \to CPO^E$

$$A \oplus B =_{def} A + B$$

$$((f^L, f^R) : A \to A', (g^L, g^R) : B \to B') \quad \overset{\oplus}{\longmapsto} \quad (f^L + g^L, f^R + g^R) : A + B \to A' + B'$$

Produkt $\otimes : CPO^E \times CPO^E \to CPO^E$

$$A \otimes B =_{def} A \times B$$

$$((f^L, f^R) : A \to A', (g^L, g^R) : B \to B') \quad \overset{\otimes}{\longmapsto} \quad (f^L \times g^L, f^R \times g^R) : A \times B \to A' \times B'$$

Funktionskonstruktor $\ominus : CPO^E \times CPO^E \to CPO^E$

$$A \ominus B =_{def} A \to B$$

$$((f^L, f^R) : A \to A', (g^L, g^R) : B \to B') \quad \overset{\ominus}{\longmapsto} \quad (h^L, h^R) : (A \to B) \to (A' \to B')$$

$$h^L(\phi : A \to B) =_{def} g^L \circ \phi \circ f^R : A' \to B'$$

$$h^R(\psi : A' \to B') =_{def} g^R \circ \psi \circ f^L : A \to B \quad \text{❑}$$

Satz 11.3.6 CPO^E ist eine ω-Kategorie. $\oplus$, $\otimes$ und $\ominus$ und deren Kompositionen sind ω-Funktoren.

Beweis. $\{\bot\}$ ist das initiale Objekt von CPO^E. Die Konstruktion von Colimiten von ω-Diagrammen in CPO^E ist in Satz 11.3.3 angegeben. Daß $\oplus$, $\otimes$ und $\ominus$ Colimiten erhalten, läßt sich leicht mithilfe von Satz 11.3.4 zeigen. ❑

Korollar 11.3.7 Sei S eine Menge von Sorten, DT ein rekursiver Datentyp über S (vgl. 11.2.11), $\alpha_1, \ldots, \alpha_n$ die Parametervariablen von t_i, A eine stetige $(S, \emptyset)$-Algebra (vgl. 10.1.8) und $B = (B_1, \ldots, B_n) \in CPO^n$. Dann ist für alle Rumpftypen t von DT ein Funktor

$$t_{B,DT} \;:\; (CPO^E)^k \;\longrightarrow\; CPO^E$$

wie folgt induktiv definiert[78]: Sei $C = (C_1, \ldots, C_k) \in CPO^k$.

- $(\alpha_i)_{B,DT}(C) =_{def} B_i$ für alle $1 \le i \le n$.
- $(\beta_i)_{B,DT}(C) =_{def} C_i$ für alle $1 \le i \le k$.
- $s_{B,DT}(C) =_{def} A_{s,\bot}$ für alle $s \in S$.
- $(t_1 + \cdots + t_n)_{B,DT}(C) =_{def} t_{1,B,DT}(C) \oplus \cdots \oplus t_{n,B,DT}(C)$.
- $(t_1 \times \ldots \times t_n)_{B,DT}(C) =_{def} t_{1,B,DT}(C) \otimes \cdots \otimes t_{n,B,DT}(C)$.

[78] Die Definition von $t_{B,DT}$ folgt einem ähnlichen Schema wie die Definition von t_Φ (vgl. 10.1.8)

- $(t \to t')_{B,DT}(C) =_{def} t_{B,DT}(C) \ominus t'_{B,DT}(C)$.

$t_{B,DT}$ ist ein ω-Funktor, so daß für die k Rumpftypen $t_1, \ldots, t_k$ von DT auch

$$\tau_{B,DT} \; : \; (CPO^E)^k \; \longrightarrow \; (CPO^E)^k$$

mit

$$\tau_{B,DT}(C) \; =_{def} \; (t_{1,B,DT}(C), \ldots, t_{k,B,DT}(C))$$

ein ω-Funktor ist.

Beweis. Da sich $t_{B,DT}$ aus den Funktoren $\oplus$, $\otimes$ und $\ominus$ zusammensetzt, folgt die Behauptung aus Satz 11.3.6. ❑

Der kategorielle Fixpunktsatz 11.2.9 liefert nun für alle CPOs B mit kleinstem Element und $\tau = \tau_{B,DT}$ ein Objekt $Dom_{B,DT}$ in $(CPO^E)^k$ mit $\tau(Dom_{B,DT}) \cong Dom_{B,DT}$, das mit dem Zielobjekt des Colimes des ω-Diagramms

$$\{f_i\}_{i\in\mathbb{N}} =_{def} \{\tau^i(ini_{T(\{\bot\})}) : \tau^i(\{\bot\}) \to \tau^{i+1}(\{\bot\})\}_{i\in\mathbb{N}}$$

übereinstimmt. Wir nennen $Dom_{B,DT}$ wie in §11.2 die **Semantik von (B,DT)**.

Nach Satz 11.3.3 ist $Dom_{B,DT}$ isomorph zur Menge aller Folgen $\{a_i\}_{i\in\mathbb{N}}$ mit $a_i = f_i^R(a_{i+1}) \in \tau^i(\{\bot\})$ für alle $i \in \mathbb{N}$. Definiert man nun

- $A_{\alpha_i} =_{def} B_i$,
- $A_{\beta_i} =_{def} A_{\mu\beta_i t_i} =_{def} proj_i(Dom_{B,DT})$ (= i-te Komponente von $Dom_{B,DT}$),
- $A_{t_1+\cdots+t_n} =_{def} A_{t_1} \oplus \cdots \oplus A_{t_n}$,
- $A_{t_1\times\cdots\times t_n} =_{def} A_{t_1} \otimes \cdots \otimes A_{t_n}$,
- $A_{t\to t'} =_{def} A_t \ominus A_{t'}$,

dann gelten jetzt zumindest die Forderungen (i), (ii) und (iii) an ein Typmodell A (vgl. 11.1.2). Der wesentliche Zugewinn von CPO^E gegenüber Set ist die Interpretierbarkeit rekursiver Typen mit Funktionskonstruktor wie $\mu fun(fun \to fun)$, das Beispiel, mit dem alles begann: Wenn ein Programm eine Funktion darstellt und diese selbst als Objekt benutzt wird, dann hat man es offenbar mit einem Bereich zu tun, der die Gleichung $B = [B \to B]$ in B löst. Aber auch nichtfunktionale rekursive Datentypen wie 11.2.15, 11.2.16 und 11.2.17 bekommen oft eine andere Semantik, wenn wir die entsprechenden Bereichsgleichungen nicht in Set, sondern in CPO^E lösen:

Beispiel 11.3.8 (Ströme) (vgl. [77], Kap. 13) Sei

$$Stream \; = \; \{\mu stream(\alpha \times stream)\},$$

Σ eine Signatur und A eine stetige Σ-Algebra. Dann ist $Dom_{B,Stream}$ isomorph zur Menge aller Ströme von Elementen aus B. Das ergibt sich aus der Darstellung von $Dom_{B,Stream}$ als Menge aller Folgen $\{a_i\}_{i\in\mathbb{N}}$ mit $a_i = f_i^R(a_{i+1}) \in \tau^i_{B,Stream}(\{\bot\})$ für alle $i \in \mathbb{N}$ (s.o.). Die **Approximation** $\tau^i_{B,Stream}(\{\bot\})$ ist isomorph zur Menge aller Listen mit höchstens $i-1$ Elementen. Der CPO^E-Morphismus

$$\tau^i_{B,List}(\{\bot\}) \; \xrightarrow{f_i} \; \tau^{i+1}_{B,List}(\{\bot\})$$

ist wie folgt definiert: $f_i^L(a) = f_i^R(a) =_{def} a$ für alle Listen a mit höchstens $i-1$ Elementen, $f_i^R(a) =_{def} \bot$ für alle Listen a mit genau i Elementen. Folglich enthält $Dom_{B,List}$ endliche und unendliche Listen, wobei eine endliche Liste mit n Elementen als eine Folge $\{a_i\}_{i\in\mathbb{N}}$ mit $a_i = \bot$ für alle $i > n$ repräsentiert ist. 🐰

Beispiel 11.3.9 (binäre Bäume) Sei

$$Bintree \;=\; \{\mu bintree(bintree \times \alpha \times bintree)\},$$

Σ eine Signatur und A eine stetige Σ-Algebra. Dann ist $Dom_{B,Bintree}$ isomorph zur Menge aller endlichen und unendlichen binären Bäume mit Knoteneinträgen aus B. Das ergibt sich wie bei $Dom_{B,Stream}$ aus der Darstellung von $Dom_{B,Bintree}$ als Menge aller Folgen $\{a_i\}_{i\in\mathbb{N}}$ mit $a_i = f_i^R(a_{i+1})$ für alle $i \in \mathbb{N}$. Die Approximation $\tau^i_{B,Bintree}(\{\bot\})$ ist isomorph zur Menge aller binären Bäume mit maximaler Tiefe $i-1$. 🐰

Beispiel 11.3.10 (Bäume mit beliebigem Ausgrad) Sei

$$Tree \;=\; \{\mu tree(\alpha \times treestream),\ \mu treestream(tree \times treestream)\},$$

Σ eine Signatur und A eine stetige Σ-Algebra. Dann sind die Komponenten

$$proj_{tree}(Dom_{B,Tree}) \;\text{ und }\; proj_{treestream}(Dom_{B,Tree})$$

von $Dom_{B,Tree}$ isomorph zur Menge aller endlichen und unendlichen Bäume mit Knoteneinträgen aus B bzw. aller Ströme solcher Bäume. Die Approximation $proj_{tree}(\tau^i_{B,Tree}(\{\bot\}))$ ist isomorph zur Menge aller Bäume mit maximaler Tiefe $i-1$, während

$$proj_{treestream}(\tau^i_{B,Tree}(\{\bot\}))$$

isomorph ist zur Menge aller Listen von höchstens $i-1$ Bäumen mit maximaler Tiefe $i-1$. 🐰

Damit sind alle — z.B. in Haskell auftretenden — **monomorphen** Produkt-, Funktions- und rekursiven Typen interpretiert. Es fehlt aber noch Polymorphie, denn jede freie Typvariable eines rekursiven Datentyps DT wird von $Dom_{B,DT}$ durch eine Komponente von B fest interpretiert.

11.4 Polymorphe Bereiche

Um den Typ $\forall \alpha t$ zu interpretieren, müssen wir einen **universalen Bereich** konstruieren, der die Interpretationen aller geschlossenen Typen umfaßt. Um wirklich *alle* geschlossenen Typen im Sinne von 11.1.1, d.h. die beliebige Verschachtelung von Polymorphie ($\forall$) und Rekursion (μ) zuzulassen, müßten wir rekursive Typen in metrischen Räumen mithilfe des Banachschen Fixpunktsatzes interpretieren (vgl. [64]). Das wäre der im vorigen Abschnitt eingeführten Semantik von Datentypen zwar ähnlich, aber doch mit neuer Theorie verbunden, auf die wir verzichten können, wenn wir uns auf funktionale Sprachen beschränken, in denen rekursive Typen vor ihrer Verwendung deklariert werden müssen, wie es in allen implementierten Sprachen der Fall ist. Dann enthält nämlich jedes Programm höchstens einen rekursiven Datentyp im Sinne von 11.2.11 DT, während die Typen aller im Programm auftretenden Objekte aus Basistypen, $\times$, $\to$ und Elementen von DT zusammengesetzt sind.

Definition 11.4.1 (Haskell-Typen) Sei S eine Menge von Sorten,

$$DT = \{\mu\beta_1.t_1, \ldots, \mu\beta_k.t_k\}$$

ein rekursiver Datentyp über S mit den Parametervariablen $\alpha_1, \ldots, \alpha_n$ (s. 11.2.11) und TX eine Menge von Typvariablen mit $TX \cap \{\beta_1, \ldots, \beta_k\} = \emptyset$. Die Menge der **Haskell-Typen über** (S, DT) ist induktiv definiert:

- Jede Sorte von S ist ein Haskell-Typ über (S, DT).
- Jede Typvariable von TX ist ein Haskell-Typ über (S, DT).
- Für alle $1 \le i \le k$ und Haskell-Typen $p_1, \ldots, p_n$ über (S, DT) ist die **Instanz** $\beta_i(p_1, \ldots, p_n)$ von β_i ein Haskell-Typ über (S, DT).

- Für alle Haskell-Typen $t, t', t_1, \ldots, t_n$ über (S, DT) und $\alpha \in TX$ sind auch $t_1 \times \cdots \times t_n$, $t \to t'$ und $\forall \alpha t$ Haskell-Typen über (S, DT). ❑

Der Unterschied zwischen Typen (vgl. 11.1.1) und Haskell-Typen auf der Bereichsebene entspricht auf der Funktionsebene demjenigen zwischen Sprachen ohne und solchen mit μ-Abstraktion (s. §6.5). μ-Abstraktion ist das syntaktische Mittel zur Definition einer rekursiven Funktion innerhalb eines Terms. Die rekursive Funktion braucht nicht separat "deklariert" zu werden, bevor sie unter einem bestimmten Namen in Termen verwendet wird. Der μ-Operator läßt sich auf den Fixpunktkombinator Y zurückführen (s. §6.5), der wiederum einer stetigen Funktion f deren kleinsten Fixpunkt zuordnet (vgl. 10.1.4):

$$\mu x.e \;=\; Y(\lambda x.e) \;=\; (lfp \circ Abs_x)(e),$$

wobei der **Abstraktionsoperator** Abs_x durch $Abs_x(e) = \lambda x.e$ definiert ist.

Yf entspricht dem λ-Term $\lambda x(f(xx))\lambda x(f(xx)))$ (s. §6.6), so dass Y eigentlich durch die folgende Haskell-Funktion definiert sein sollte:

```
Y f = (\x -> f (x x)) (\x -> f (x x))
```

Da Yf nicht typisierbar ist, ist diese Definition jedoch syntaktisch inkorrekt. Erst unter Verwendung eines rekursiven Datentyps läßt sich Y in einer getypten Sprache wie Haskell definieren (zur ML-Version siehe [96], S. 377):

```
data CFun a = C (CFun a -> a)
Y :: (a -> a) -> a
Y f = (\C x -> f (x (C x))) (C (\C x -> f (x (C x))))
```

x bildet jetzt den Typ `CFun a` auf den Typ `a` ab.

Als Anekdote am Rande sei vermerkt, daß die Fixpunktgleichung $f(Yf) = Yf$ als die berühmte **Russellsche Antinomie** interpretiert werden kann, die mengentheoretisch wie folgt ausgedrückt wird:

$$\{x \mid x \notin x\} \notin \{x \mid x \notin x\} \overset{RA}{\iff} \{x \mid x \notin x\} \in \{x \mid x \notin x\}.$$

Interpretiert man die Applikation $(e\ e')$ als Formel $e' \in e$, die Abstraktion λxe als Menge $\{x \mid x \models e\}$ und f als Negation, dann gilt tatsächlich $f(Yf) \Leftrightarrow Yf$:

$$f(Yf) \iff \neg Yf \iff \neg(\lambda x(f(xx))\lambda x(f(xx))) \iff \lambda x(f(xx)) \notin \lambda x(f(xx))$$

$$\iff \{x \mid f(xx)\} \notin \{x \mid f(xx)\} \iff \{x \mid \neg(xx)\} \notin \{x \mid \neg(xx)\} \iff \{x \mid x \notin x\} \notin \{x \mid x \notin x\}$$

$$\overset{RA}{\iff} \{x \mid x \notin x\} \in \{x \mid x \notin x\} \iff \{x \mid \neg(xx)\} \in \{x \mid \neg(xx)\} \iff \{x \mid f(xx)\} \in \{x \mid f(xx)\}$$

$$\iff \lambda x(f(xx)) \in \lambda x(f(xx)) \iff (\lambda x(f(xx))\lambda x(f(xx))) \iff Yf.$$

Da der μ-Operator durch die Gleichung $\mu x = Y \circ Abs_x$ definiert ist (s.o.), wird μ-Abstraktion *auf der Funktionsebene* semantisch dadurch begründet, daß für jeden CPO C, die Funktionen $Y : [C \to C] \to C$ und $Abs_x : [C \times C \to C] \to [C \to [C \to C]]$ stetig sind (vgl. z.B. [61], Theoreme 4.23 und 4.32). Analog dazu brauchen wir zur Definition der Semantik von Haskell-Typen den Nachweis, daß entsprechende Fixpunkt- und Abstraktionsoperatoren *auf der Bereichsebene* ω-Funktoren sind.

Definition 11.4.2 (Fixpunkt- und Abstraktionsfunktor) Seien $\mathcal{K}$ und $\mathcal{L}$ zwei Kategorien und seine F und G zwei Funktoren von $\mathcal{K}$ nach $\mathcal{L}$. Eine **natürliche Transformation** η von F nach G, geschrieben:

$\eta : F \to G$, ist eine Funktion, die jedem $A \in Obj(\mathcal{K})$ einen $\mathcal{L}$-Morphismus $\eta_A : F(A) \to G(A)$ zuordnet so, daß für alle $\mathcal{K}$-Morphismen $f : A \to B$

$$G(f) \circ \eta_A = \eta_B \circ F(f)$$

gilt. $[\mathcal{K} \to \mathcal{L}]$ bezeichnet die Kategorie, deren Objekte die ω-Funktoren von $\mathcal{K}$ nach $\mathcal{L}$ und deren Morphismen die natürlichen Transformationen zwischen diesen Funktoren sind. Sei $\mathcal{C}$ eine ω-Kategorie.

Fixpunktfunktor $\quad Y : [\mathcal{C} \to \mathcal{C}] \to \mathcal{C}$

Sei J das initiale Objekt von $\mathcal{C}$ (vgl. 11.2.10).

$Y(F) \;=_{def}$ Zielobjekt des Colimes $\{\mu_{F,i}\}_{i\in\mathbb{N}}$ des ω-Diagramms $\{F^i(ini_{F(J)})\}_{i\in\mathbb{N}}$.

$Y(\eta : F \to G) \;=_{def}$ eindeutiger $\mathcal{C}$-Morphismus $\phi : Y(F) \to Y(G)$

mit $\phi \circ \mu_{F,i} = \mu_{G,i} \circ f_i$ für alle $i \in \mathbb{N}$, wobei $f_0 =_{def} id_J$ und $f_{i+1} =_{def} \eta_{G^i(J)} \circ F(f_i)$.

Abstraktionsfunktor $\quad Abst : [\mathcal{K} \times \mathcal{L} \to \mathcal{C}] \to [\mathcal{K} \to [\mathcal{L} \to \mathcal{C}]]$

$Abst(F)(A)(B) \;=_{def} F(A, B)$.

$Abst(F)(f : A \to A') \;=_{def} \eta : (Abst(F)(A) : \mathcal{L} \to \mathcal{C}) \to (Abst(F)(A') : \mathcal{L} \to \mathcal{C})$,

wobei $\eta_B =_{def} F(f, id_B) : F(A, B) \to F(A', B)$ für alle $B \in Obj(\mathcal{L})$.

$(Abst(\eta : F \to G)_A)_B \;=_{def} \eta_{(A,B)} : F(A, B) \to G(A, B)$ für alle $A \in Obj(\mathcal{K})$ und $B \in Obj(\mathcal{L})$. ❑

Satz 11.4.3 Seien $\mathcal{K}$ und $\mathcal{L}$ beliebige Kategorien und $\mathcal{C}$ eine ω-Kategorie. Dann gilt:

(i) $[\mathcal{K} \to \mathcal{C}]$ ist eine ω-Kategorie.

(ii) Der Fixpunktfunktor $Y : [\mathcal{C} \to \mathcal{C}] \to \mathcal{C}$ ist ein ω-Funktor.

(iii) Der Abstraktionsfunktor $Abst : [\mathcal{K} \times \mathcal{L} \to \mathcal{C}] \to [\mathcal{K} \to [\mathcal{L} \to \mathcal{C}]]$ ist ein Isomorphismus.

Beweis. (i) und (ii) sind in [60], Kap. 4, bewiesen (Lemma 1 bzw. Theorem 4.1). Insbesondere ist das Morphismenbild $Y(\eta : F \to G)$ (s.o.) wohldefiniert, weil, wie man leicht nachrechnet, die Folge $\{\mu_{G,i} \circ f_i\}_{i\in\mathbb{N}}$ ein Cokegel des ω-Diagramms $\{F^i(ini_{F(J)})\}_{i\in\mathbb{N}}$ ist, ϕ also dem eindeutigen Morphismus $ini_{Y(G)} : Y(F) \to Y(G)$ entspricht (vgl. 11.2.4). (iii) ist in [46] gezeigt (Theorem 15.9). ❑

Sei $DT = \{\mu\beta_1.t_1, \ldots, \mu\beta_k.t_k\}$ ein rekursiver Datentyp über S mit den Parametervariablen $\alpha_1, \ldots, \alpha_n$. Der universale Bereich zur Interpretation von Haskell-Typen über (S, DT) wird wie folgt konstruiert. In Korollar 11.3.7 haben wir für ein festes $B \in CPO^n$ den ω-Funktor

$$\tau_{B,DT} \;:\; (CPO^E)^k \;\longrightarrow\; (CPO^E)^k$$

gebildet. Der kategorielle Fixpunktsatz 11.2.9 liefert einen Fixpunkt $Dom_{B,DT}$ von $\tau_{B,DT}$ (vgl. §11.3). Wir definieren nun

$$\tau_{DT} \;:\; (CPO^E)^n \times (CPO^E)^k \;\longrightarrow\; (CPO^E)^k$$

durch $\tau_{DT}(B, C) =_{def} \tau_{B,DT}(C)$. Nach Satz 11.4.3(ii) und (iii) ist der Fixpunktfunktor $Y : [(CPO^E)^k \to (CPO^E)^k] \to (CPO^E)^k$ ein ω-Funktor und der Abstraktionsfunktor

$$Abst : [(CPO^E)^n \times (CPO^E)^k \to (CPO^E)^k] \to [(CPO^E)^n \to [(CPO^E)^k \to (CPO^E)^k]]$$

ein Isomorphismus. Also gilt das auch für deren Komposition zum **Semantikfunktor** $Sem(DT)$ **von** DT:

$$Sem(DT) \;:\; (CPO^E)^n \;\overset{Abst(\tau_{DT})}{\longrightarrow}\; [(CPO^E)^k \to (CPO^E)^k] \;\overset{Y}{\longrightarrow}\; (CPO^E)^k$$

und dessen Verknüpfung mit einer Projektion $proj_i,\ 1 \leq i \leq k$:

$$Sem_i(DT)\ :\ CPO^E \overset{Sem(DT)}{\longrightarrow} (CPO^E)^k \overset{proj_i}{\longrightarrow} CPO^E.$$

Nach dieser Definition sind $Sem(DT)(B)$ und der Fixpunkt $Dom_{B,DT}$ von $\tau_{B,DT}$ isomorph. $Sem(DT)(B)$ liefert also wie $Dom_{B,DT}$ eine Lösung des Gleichungssystems

$$\beta_1 = (t_1)_{B,DT}(\beta_1,\ldots,\beta_k), \quad \ldots, \quad \beta_k = (t_k)_{B,DT}(\beta_1,\ldots,\beta_k)$$

(vgl. §11.3). Durch den Übergang von $Dom_{B,DT}$ zum Funktor $Sem(DT)$ haben wir den Parameter B aus $Dom_{B,DT}$ "herausgelöst". Außerdem liefert die Projektion $Sem_i(DT)$ von $Sem(DT)$ einen ω-Funktor, der die Typvariable β_i in entsprechender Weise als Typkonstruktor interpretiert wie die ω-Funktoren $\oplus$, $\otimes$ und $\ominus$ die Typkonstruktoren $+$, $\times$ bzw. $\to$ interpretieren (vgl. 11.3.5).

Definition 11.4.4 (universaler Bereich) Sei S eine Menge von Sorten, A eine stetige $(S,\emptyset)$-Algebra (vgl. 10.1.8) und $DT = \{\mu\beta_1.t_1,\ldots,\mu\beta_k.t_k\}$ ein rekursiver Datentyp über S mit den Parametervariablen $\alpha_1,\ldots,\alpha_n$. Nach Satz 11.3.6 ist der Funktor $\tau_{All} : CPO^E \to CPO^E$ mit

$$\tau_{All}(B) \ =_{def}\ (\bigoplus_{s\in S} A_{s,\perp}) \oplus (B \otimes B) \oplus (B \ominus B) \oplus (\bigoplus_{i=1}^{k} Sem_i(DT)(B,\ldots,B))$$

ein ω-Funktor. Also liefert der kategorielle Fixpunktsatz (11.2.9) einen CPO Dom_A mit $\tau_{All}(Dom_A) \cong Dom_A$. Wir nennen Dom_A den **universalen Bereich über** A. Sei

$$Sum_A \ =_{def}\ (\bigoplus_{s\in S} A_{s,\perp}) \oplus (Dom_A \otimes Dom_A) \oplus (Dom_A \ominus Dom_A) \oplus (\bigoplus_{i=1}^{k} Sem_i(DT)(Dom_A,\ldots,Dom_A)).$$

Zu jedem Summanden C von Sum_A gibt es eine **Einbettung** in_C **von** C **in** Dom_A. Für alle $s \in S$ und $1 \leq i \leq k$ bezeichne in_s die Einbettung von $A_{s,\perp}$, in_i die Einbettung von $Sem_i(DT)(Dom_A)$ und $in_\otimes$ bzw. $in_\ominus$ die Einbettung von $Dom_A \otimes Dom_A$ bzw. $Dom_A \ominus Dom_A$ in Dom_A. ❑

Mithilfe von Dom_A können wir A zu einem Typmodell erweitern, das zumindest für Haskell-Typen über (S,DT) die Bedingungen 11.1.2(i)-(iv) erfüllt. Für jeden solchen Typ t wird A_t als eine Teilmenge von Dom_A definiert. Um die im Anschluß an 11.1.2 genannten Probleme zu vermeiden, die aufträten, wenn einige Typen durch die leere Menge interpretiert würden, sind alle zugeordneten Teilmengen von Dom_A **Ideale** von Dom_A.

Definition 11.4.5 (Ideal) Eine nichtleere Teilmenge I eines CPO B heißt **Ideal von** B, wenn

- I **nach unten abgeschlossen ist**, d.h. für alle $a \in B$ und $b \in I$ mit $a \leq b$ ist $a \in I$,
- das Supremum jeder Kette $\{a_i\}_{i\in\mathbb{N}} \subseteq I$ in I liegt.

Die Menge aller Ideale von D bezeichnen wir mit $\mathcal{I}(D)$. ❑

Das kleinste Ideal eines CPO B mit kleinstem Element $\perp$ ist durch $\{\perp\}$ gegeben, das größte Ideal ist B selbst. $\mathcal{I}(B)$ ist **abgeschlossen gegenüber Schnitten**, d.h. der Schnitt einer beliebigen Menge von Idealen von B ist wieder ein Ideal von B.

Definition 11.4.6 (Semantik von Haskell-Typen) Sei S eine Menge von Sorten, A eine stetige $(S,\emptyset)$-Algebra, $DT = \{\mu\beta_1.t_1,\ldots,\mu\beta_k.t_k\}$ ein rekursiver Datentyp über S, TX eine Menge von Typvariablen mit $TX \cap \{\beta_1,\ldots,\beta_k\} = \emptyset$ und Dom_A der universale Bereich über A. Für jeden Haskell-Typ t über (S,DT) und jede Belegung b von TX in $\mathcal{I}(Dom_A)$ ist die **Semantik** $Sem_A(t)(b)$ **von** t **bzgl.** (b,A) eine induktiv über dem Aufbau von Haskell-Typen definiertes Ideal von Dom_A (vgl. 11.4.1 und 11.4.4):

(i) $Sem_A(s)(b) =_{def} in_s(A_{s,\perp})$ für alle $s \in S$.

(ii) $Sem_A(\alpha)(b) =_{def} b(\alpha)$ für alle $\alpha \in TX$.

(iii) $Sem_A(\beta_i(p_1,\ldots,p_n))(b) =_{def} \bigcap\{I \in \mathcal{I}(Dom_A) \mid (in_i \circ Sem_i(DT)(inc)^L)(Sem_i(DT)(P)) \subseteq I\}$
für alle $1 \leq i \leq k$, wobei $P =_{def} Sem_A(p_1 \times \cdots \times p_n)(b)$ und der CPO^E-Morphismus $inc : P \to Dom_A$ durch $inc^L(c) = inc^R(c) = c$ für alle $c \in P$ und $inc^R(c) = \bot$ für alle $c \in Dom_A \setminus P$ definiert ist (vgl. 11.3.1).

(iv) $Sem_A(t_1 \times \cdots \times t_r)(b) =_{def} in_\otimes(Sem_A(t_1)(b) \otimes Sem_A(t_2 \times \ldots \times t_r)(b))$ für alle Haskell-Typen $t_1,\ldots,t_r$.

(v) $Sem_A(t \to t')(b) =_{def} in_\ominus(\{f \in Dom_A \ominus Dom_A \mid f(Sem_A(t)(b)) \subseteq Sem_A(t')(b)\})$
für alle Haskell-Typen t, t'.

(vi) $Sem_A(\forall\alpha t)(b) =_{def} \bigcap_{I \in \mathcal{I}(Dom_A)}\{Sem_A(t)(b_{\alpha\leftarrow I})\}$ für alle $\alpha \in TX$ und Haskell-Typen t. ❑

Sei $a \in A$. (ii), (v) und (vi) liefern z.B.

$$Sem_A(\forall\alpha.\alpha \to \alpha)(b) = \bigcap_{I \in \mathcal{I}(Dom_A)} \{Sem_A(\alpha \to \alpha)(b_{\alpha\leftarrow I})\}$$

$$= \bigcap_{I \in \mathcal{I}(Dom_A)} \{in_\ominus(\{f \in Dom_A \ominus Dom_A \mid f(Sem_A(\alpha)(b_{\alpha\leftarrow I})) \subseteq Sem_A(\alpha)(b_{\alpha\leftarrow I})\})\}$$

$$= \bigcap_{I \in \mathcal{I}(Dom_A)} \{in_\ominus(\{f \in Dom_A \ominus Dom_A \mid f(I) \subseteq I\})\}$$

$$= in_\ominus(\{f \in Dom_A \ominus Dom_A \mid f(\{a,\bot\}) \in \{a,\bot\}\}) = in_\ominus(\{f \in Dom_A \ominus Dom_A \mid f(a) \in \{a,\bot\}\}).$$

Die letzte Gleichung gilt wegen $f(\bot) = \bot$ für alle $f \in Dom_A \ominus Dom_A$.

Satz 11.4.7 Unter den Voraussetzungen von Def. 11.4.6 ist $Sem_A(t)(b)$ für jeden Haskell-Typ t über (S, DT) und jede Belegung $b : TX \to \mathcal{I}(Dom_A)$ ein Ideal von Dom_A.

Beweis. Zunächst ist zu bemerken, daß für alle Ideale I eines Summanden C von Σ_A (s.o.) das Bild von I unter der Einbettung $in_C : C \to Dom_A$ ein Ideal von Dom_A ist. Weiterhin sieht man sofort, daß in den Fällen 11.4.6(i), (ii) und (iv) ein Ideal von Dom_A definiert wird. In den Fällen (iii) und (vi) wird ein Ideal von Dom_A definiert, weil $\mathcal{I}(Dom_A)$ gegenüber Schnitten abgeschlossen ist. Im Fall (v) folgt das aus der Tatsache, daß für alle CPOs B und $I, J \in \mathcal{I}(B)$ die Menge $\{f \in B \ominus B \mid f(I) \subseteq J\}$ ein Ideal von $B \ominus B$ ist (vgl. [108], Lemma 15.5.4).

Das im Fall (iii) die Menge $(in_i \circ Sem_i(DT)(inc)^L)(Sem_i(DT)(P))$ umfassende Ideal I von Dom_A existiert, weil zumindest Dom_A selbst ein Ideal von Dom_A ist. $Sem_i(DT)(inc)$ setzt die Einbettung inc von $Sem_A(p_1 \times \cdots \times p_n)(b)$ in Dom_A fort zur Einbettung von $Sem_i(DT)(Sem_A(t)(b))$ in $F_i(Dom_A)$. ❑

Damit haben wir ein Modell von Haskell-Typen konstruiert, in dem λ-Terme und Typurteile (vgl. 11.1.4 bzw. 11.1.5) interpretiert und die Korrektheit des λ-Kalküls und des Curry-Kalküls (vgl. 11.1.6 bzw. 11.1.5) bewiesen werden kann:

Satz 11.4.8 (Modelle von Haskell-Typen) Seien die Voraussetzungen von Def. 11.4.6 gegeben und sei b eine beliebige Belegung von TX in $\mathcal{I}(Dom_A)$. Mit $A_t =_{def} Sem_A(t)(b)$ wird A zu einem Modell aller geschlossenen Haskell-Typen t über (S, DT) erweitert, das 11.1.2(ii)-(iv) erfüllt. Ein so gebildetes Typmodell heißt **Haskell-Typmodell über** (S, DT).

Sei A ein Haskell-Typmodell über (S, DT), C eine Menge von Konstanten, deren Typen geschlossene Haskell-Typen über (S, DT) sind (vgl. 11.1.3). Dann nennen wir eine polymorphe Signatur $\Sigma = (S, C)$ eine **Haskell-Signatur über** (S, DT) und eine polymorphe Σ-Algebra (A, C^A) eine **Haskell-Algebra über** Σ. ❑

Die Trägermenge einer Σ-Haskell-Algebra (A, C^A) ist demnach eine T-sortierte Menge von Idealen von Dom_A, wobei T die Menge aller geschlossenen Haskell-Typen über (S, DT) ist. Folglich wird jede Konstante von C durch ein Element eines dieser Ideale interpretiert.

Definition 11.4.9 (Bereichssemantik von λ-Termen) Sei $\Sigma = (S, C)$ eine Haskell-Signatur über (S, DT), X eine Menge von Individuenvariablen, (A, C^A) eine Haskell-Algebra über Σ. Für jeden λ-Term e über Σ und X und jede Belegung b von X in Dom_A ist die **Semantik** $Sem_A(e)(b)$ **von** e **bzgl.** (b, A) ein induktiv über dem Aufbau von λ-Termen definiertes Element von Dom_A (vgl. 11.1.4):

- $Sem_A(c)(b) =_{def} c^A$ für alle $c \in C$.
- $Sem_A(x)(b) =_{def} b(x)$ für alle $c \in X$.
- $Sem_A(\lambda x.e)(b)(a) =_{def} Sem_A(e)(b_{x \leftarrow a})$ für alle $a \in Dom_A$.
- $Sem_A((e\ e'))(b) =_{def} \begin{cases} Sem_A(e)(b)(Sem_A(e')(b)) & \text{falls } Sem_A(e)(b) \in in_{\ominus}(Dom_A \ominus Dom_A) \\ \text{undefiniert} & \text{sonst.} \end{cases}$
- $Sem_A(e_1, \ldots, e_n)(b) =_{def} (Sem_A(e_1)(b), \ldots, Sem_A(e_n)(b))$.
- $Sem_A(e_{x \leftarrow e'})(b) =_{def} Sem_A(e)(b_{x \leftarrow Sem_A(e')})$.
- $Sem_A(\mu x.e) =_{def} lfp \circ Sem_A(\lambda x.e)$[79] (vgl. 10.1.4). ❑

Da der Fixpunktoperator wegen der call-by-value-Strategie von Haskell nur auf höhere Funktionen angewendet werden kann (vgl. §11.4), ergibt sich aus der Semantik von $\mu x.e$, daß x und e denselben *funktionalen* Typ haben müssen, e also von der Form $\lambda y.e'$ sein muß (siehe auch §6.5).

Satz 11.4.10 (Äquivalenz von Bereichs- und Reduktionssemantik von λ-Termen, Korrektheit des λ-Kalküls) (vgl. 11.1.8) Sei $\Sigma = (S, C)$ eine Haskell-Signatur über (S, DT) und (A, C^A) eine Haskell-Algebra über Σ. Dann gilt für alle typisierbaren λ-Terme e über Σ:

$$Sem_A(e) = Sem_A(e^{red(A)}).$$

Beweis. In §11.1 wurde erwähnt, daß jeder typisierbare λ-Term eine Normalform hat. Zu zeigen bleibt, daß die von den Reduktionsregeln des λ-Kalküls erzeugte Reduktionsrelation $\longrightarrow_\lambda$ **korrekt bzgl.** A ist, d.h.

$$e \longrightarrow_\lambda e' \quad \text{impliziert} \quad Sem_A(e) = Sem_A(e').$$

Der Beweis dieser Implikation zerfällt in zwei Teile:

(i) dem Nachweis, daß $Sem_A(e) = Sem_A(e')$ für alle $\longrightarrow_\lambda$ erzeugenden Paare (e, e') gilt, also (vgl. 11.1.6):

- $Sem_A((\lambda x e\ e')) = Sem_A(e_{x \leftarrow e'})$, falls keine freien Variablen von e' in e gebunden vorkommen,
- $Sem_A(\lambda x(e\ x)) = Sem_A(e)$, falls x keine freie Variable von e ist,
- $Sem_A((c\ d)) = Sem_A(c^A(d^A))$ für alle $c : t \to t', d : t \in C$ mit $c^A(d^A) \in C$,

(ii) dem Nachweis, daß für alle $c \in \Lambda_\Sigma(X)$ und $x \in var(c)$ gilt:

- $Sem_A(e) = Sem_A(e')$ impliziert $Sem_A(c_{x \leftarrow e}) = Sem_A(c_{x \leftarrow e})$.

[79] Analog wurde oben die Semantik eines rekursiven Datentyps DT (vgl. 11.2.11) als Komposition $Y \circ Abst(\tau_{DT})$ von Fixpunktfunktor und Bild des Abstraktionsfunktor definiert.

(ii) erhält man durch Induktion über die Größe von e bzw. c, weil Sem_A induktiv über dem Aufbau von λ-Termen definiert ist. (i) sei dem Leser überlassen. ❑

Definition 11.4.11 (Semantik von Typurteilen) (vgl. 11.1.5) Sei $\Sigma = (S, C)$ eine Haskell-Signatur über (S, DT), (A, C^A) eine Haskell-Algebra über Σ und σ eine partielle Funktion von X in die Menge der Haskell-Typen über (S, DT). Ein Typurteil $\sigma \vdash e : t$ ist **gültig in** A, falls für jede Belegung $b : X \to Dom_A$ und jede Belegung $c : TX \to \mathcal{I}(Dom_A)$ gilt:

$$b(x) \in Sem_A(t')(c) \text{ für alle } x : t' \in \sigma \quad \text{impliziert} \quad Sem_A(e)(b) \in Sem_A(t)(c).$$

Wir schreiben dann: $A \models \sigma \vdash e : t$. ❑

Satz 11.4.12 (Der Curry-Kalkül ist korrekt) (vgl. 11.1.5) Sei $\Sigma = (S, C)$ eine Haskell-Signatur über (S, DT) und (A, C^A) eine Haskell-Algebra über Σ. Jede Regel des Curry-Kalküls ist korrekt bzgl. A, d.h. die jeweilige Prämisse ist gültig in A, sofern die jeweiligen Konklusionen in A gültig sind.

Beweis. Mehr oder weniger vollständige Beweise findet man in [108] (Satz 15.5.7) und [64] (Theorem 11). Als Beispiel zeigen wir hier die Korrektheit der ersten Polymorphieregel des Curry-Kalküls:

$$\frac{\sigma \vdash e : \forall \alpha t}{\sigma \vdash e : t} \Uparrow \qquad \text{falls } \alpha \text{ nicht frei in } \sigma(X)$$

Für alle $b : X \to Dom_A$ und $c : TX \to \mathcal{I}(Dom_A)$ gelte $(b, c, A) \models \sigma \vdash e : t$. Weiterhin sei α eine Typvariable, die in $\sigma(X)$ nicht frei vorkommt, $b : X \to Dom_A$ und $c : TX \to \mathcal{I}(Dom_A)$ derart, daß $b(x) \in Sem_A(t')(c)$ für alle $x : t' \in \sigma$ gilt. Zu zeigen ist $Sem_A(e)(b) \in Sem_A(\forall \alpha t)(c)$, also $Sem_A(e)(b) \in Sem_A(t)(c_{\alpha \leftarrow I})$ für alle $I \in \mathcal{I}(Dom_A)$ nach Def. 11.4.6(vi). Nach der zweiten Voraussetzung gibt es kein $x : t' \in \sigma$ so, daß α in t' frei vorkommt. Demnach gilt $Sem_A(t')(c) = Sem_A(t')(c_{\alpha \leftarrow I})$ für alle $x : t' \in \sigma$ und alle $I \in \mathcal{I}(Dom_A)$. Also folgt $Sem_A(e)(b) \in Sem_A(t)(c_{\alpha \leftarrow I})$ aus der ersten Voraussetzung. ❑

11.5 Coalgebren

So abstrakt die in den bisher betrachteten Theorien definierten semantischen Bereiche auch sein mögen, immer war es das Ziel, Daten zu **konstruieren**, um Aussagen darüber möglichst durch strukturelle Induktion über dem Aufbau der Bereiche beweisen zu können. Damit entspricht ein rekursiver Datentyp DT ohne Funktionskonstruktor (s. 11.2.11) einer konstruktorbasierten Spezifikation SP und $Dom_{B,DT}$ dem Standardmodell von SP (s. §5.1). Der Isomorphismus $\tau(Dom_{B,DT}) \cong Dom_{B,DT}$ setzt sich dabei oft aus den Konstruktoren von SP zusammen.

Das Schöne an konstruktorbasierten, endlich erzeugten Datentypen ist neben ihrer direkten Implementierbarkeit die Möglichkeit, Programme, die auf ihnen arbeiten, mithilfe **struktureller Induktion** oder, allgemeiner, **Noetherscher Induktion** zu verifizieren (s. §7.2). Hier wird über dem *Aufbau von Daten* induziert, während **Scott-Induktion** über *Berechnungslängen* induziert (s. 10.1.5). Der Übergang von $\phi^i(\bot)$ zu $\phi^{i+1}(\bot)$ ist nämlich nicht anderes als ein Berechnungsschritt in der Auswertung von $lfp(\phi) = sup\{\phi^i(\bot)\}_{i \in \mathbb{N}}$. Unendliche Daten wie Ströme oder Prozesse lassen sich ebenfalls als Suprema endlicher Approximationen beschreiben. So kann z.B. das Standardmodell einer konstruktorbasierten Listenspezifikation zum CPO der Ströme *vervollständigt* werden. Die Elemente dieses CPO sind Äquivalenzklassen *unendlicher* Terme. Er ist isomorph zu dem in Bsp. 11.3.8 angegebenen Bereichsmodell $Dom_{B,Stream}$.

Bei einer Prozeßbeschreibung mithilfe von Strömen (Spursemantik; s. §9.1) geht es of um den Nachweis von **Fairneßbedingungen** wie das in Prädikat $fair$ von STREAM (s. 9.2.11). Fairneßbedingungen gehören zur Klasse der **Lebendigkeitseigenschaften**, die, wie wir in §9.2 gesehen haben, dual zu den **Sicherheitseigenschaften** einer konstruktorbasierten Spezifikation sind. So kann man Sicherheitseigenschaften i.a. mit Hornformeln und Lebendigkeitseigenschaften mit co-Hornformeln axiomatisieren. Auf semantischer Ebene er-

kennt man eine Dualität zwischen **initialen Modellen** endlicher Datentypen mit induktiv aufgebauten Daten und **finalen** oder **terminalen Modellen** unendlicher Datentypen.

Ein initiales Modell ist eine *white box*. Seine Elemente sind Äquivalenzklassen von Termen, die den Aufbau der beschriebenen Daten offenlegen. Die Äquivalenz zweier Daten läßt sich aus den gegebenen Axiomen *ableiten*. Ein finales Modell ist eine *black box*. Der Aufbau seiner Elemente ist unbekannt. Man kann sie beobachten, Messungen über sie anstellen und sie so voneinander unterscheiden. Gleich ist, was dieselben Meßergebnisse liefert. Gleich ist, wovon man *ableiten* kann, daß es *nicht verschieden* ist. Man spricht dann von **Beobachtungs-** oder **Verhaltensäquivalenz** (vgl. Kap. 10). Die dualen Äquivalenzbegriffe im initialen Modell einerseits und im finalen Modell andererseits sind ein Spezialfall der komplementären Gültigkeitsbegriffe, die wiederum aus den Konstruktionen der beiden Modelle als kleinster bzw. größter Fixpunkt folgen (im Sinne von Satz 4.2.11). Im initialen Modell ist gültig, was (induktiv) beweisbar ist. Im terminalen Modell ist gültig, was nicht widerlegbar ist.

Definition 11.5.1 (terminales Objekt) (vgl. 11.2.4) Sei $\mathcal{K}$ eine Kategorie. Ein $\mathcal{K}$-Objekt T heißt **terminales Objekt von** $\mathcal{K}$, falls für jedes $\mathcal{K}$-Objekt A genau ein $\mathcal{K}$-Morphismus $fin_A : A \to T$ existiert. ❑

Das finale Modell $Fin(SP)$ einer konstruktorbasierten Spezifikation SP (s. §9.2) ist ein terminales Objekt in einer Kategorie gewisser *hierarchischer* Σ-Strukturen.

So wie unendliche Datenstrukturen, nebenläufige Prozesse und objektorientierte Sprachkonstrukte in der Praxis in engem Zusammenhang stehen, so hat auch deren Formalisierung in Gestalt terminaler Objekte eine gemeinsame Grundlage. §9.2 legt nahe, dass der finale Ansatz für zustandsorientierte Programme allgemeinere Verifikationsmethoden bietet als der CPO-orientierte. Das wesentliche Beweisprinzip des letzteren ist die Scott-Induktion, mit der aber nur *zulässige* Formeln gezeigt werden können (s. 10.1.5). Lebendigkeitseigenschaften sind aber i.a. nicht zulässig, weil sie Aussagen über *unendliche* Objekte sind, die gerade nicht von den *endlichen* Approximationen der Objekte auf sie selbst (als Suprema der Approximationen) übertragen werden können.

Die in §9.2 diskutierte Dualität zwischen Initialität und Induktion einerseits und Finalität und Coinduktion andererseits zeigt sich ganz deutlich bei den entsprechenden kategorientheoretischen Begriffen. So läßt sich z.B. das finale Modell von STREAM (s. 9.2.11) als terminale *Coalgebra* darstellen.

Definition 11.5.2 (F-Coalgebra) (vgl. 11.2.7) Sei $F : \mathcal{K} \to \mathcal{K}$ ein Funktor und A ein $\mathcal{K}$-Objekt. Eine **F-Coalgebra** ist ein $\mathcal{K}$-Morphismus $\alpha : A \to F(A)$. F-Coalgebren bilden die Kategorie $\mathbf{Coalg}_F$. Ein $Coalg_F$-Morphismus von der F-Coalgebra $\alpha : A \to F(A)$ in die F-Coalgebra $\beta : B \to F(B)$ ist ein $\mathcal{K}$-Morphismus $f : A \to B$ mit $F(f) \circ \alpha = \beta \circ f$. ❑

Die zu 11.2.4 und 11.2.7 dualen Definitionen 11.5.1 bzw. 11.5.2 liefern ein zu 11.2.8 duales Lemma:

Lemma 11.5.3 (Terminale Coalgebren sind Isomorphismen) Sei $F : \mathcal{K} \to \mathcal{K}$ ein Funktor derart, daß die Kategorie $Coalg_F$ ein terminales Objekt $\theta : T \to F(T)$ hat. Dann sind T und $F(T)$ isomorph. ❑

Entsprechend kann man die Definitionen eines ω-Diagramms, einer ω-Kategorie (vgl. 11.2.4) und eines ω-Funktors (vgl. 11.2.6) dualisieren zum **Co-ω-Diagramm**, zur **Co-ω-Kategorie** bzw. zum **Co-ω-Funktor** (Übung!), womit wir einen zu 11.2.9 dualen kategoriellen Fixpunktsatz erhalten:

Satz 11.5.4 (Kategorieller Fixpunktsatz II) Sei $\mathcal{K}$ eine Co-ω-Kategorie, J das terminale Objekt von $\mathcal{K}$ und $F : \mathcal{K} \to \mathcal{K}$ ein Co-ω-Funktor. Dann gibt es ein Objekt T in $\mathcal{K}$ mit $F(T) \cong T$, das das Quellobjekt sowohl der terminalen F-Coalgebra als auch des **Limes**[80] des Co-ω-Diagramms $\{F^i(fin_J)\}_{i\in\mathbb{N}}$ ist (vgl. 11.5.1). ❑

Beispiel 11.5.5 (Ströme) Sei A eine Menge. Die Kategorie Set der Mengen und Abbildungen ist eine Co-ω-Kategorie. Der Funktor $F : Set \to Set$ mit $F(B) =_{def} A \times B$ für alle Mengen B und $F(f) =_{def} id_A \times f$ für alle Abbildungen $f : B \to C$ ist ein Co-ω-Funktor. Folglich sind F-Coalgebren gerade die Abbildungen der

[80] das ist die zum Colimes duale Konstruktion (vgl. 11.2.4)

Form $f : B \to A \times B$. Jede solche Abbildung ist eindeutig bestimmt durch zwei Projektionen $head_f : B \to A$ und $tail_f : B \to B$: $f(b) = (head_f(b), tail_f(b))$ für alle $b \in B$. Nach Satz 11.5.4 hat $Coalg_F$ ein terminales Objekt $\theta = (head_\theta, tail_\theta) : T \to A \times T$.

Wie initiale, so sind auch terminale Objekte eindeutig. Bezogen auf die terminale F-Coalgebra heißt das: Jede F-Coalgebra $fin : D \to A \times D$ derart, daß es zu jeder F-Coalgebra $\alpha : B \to A \times B$ eine eindeutige Abbildung $f : B \to D$ mit $F(f) \circ \alpha = fin \circ f$ gibt (vgl. 11.5.2), ist isomorph zu θ, d.h. T und D sind isomorph. Wie man leicht nachrechnet, hat die **F-Coalgebra $fin : D \to A \times D$ der Ströme** oder unendlichen Listen über A mit

- $D =_{def} [\mathbb{N} \to A]$,
- $head_{fin}(s) =_{def} s(0)$ für alle $s : \mathbb{N} \to A$,
- $(tail_{fin}(s))(n) =_{def} s(n+1)$ für alle $s : \mathbb{N} \to A$ und $n \in \mathbb{N}$

genau diese Eigenschaft. D ist also isomorph zu T, so daß nach Lemma 11.5.3 auch D und $F(D) = A \times D$ isomorph sind. Wir haben also wie im Beispiel 11.3.8 die Bereichsgleichung $D = A \times D$ in D gelöst, und die Lösung ist wieder die Menge der Ströme von Elementen einer gegebenen Datenmenge.[81] Jetzt haben wir die Lösung aber schon in Set und nicht erst in der komplizierten Kategorie CPO^E erhalten. In Set konnten wir zwar die entsprechende Gleichung $D = 1 + (A \times D)$ lösen (vgl. 11.2.15), erhielten dabei aber nur die Menge der *endlichen* Listen von Elementen aus A als Lösung. 🐰

Es ist also tatsächlich so: was im konstruktorbasierten Ansatz erst durch die *Vervollständigung* von Set nach CPO^E gelang, nämlich unendliche Datenstrukturen zu "modellieren", erreicht man im beobachtungsorientierten Ansatz schon in Set. Stellt sich die Frage: Sind *alle* terminalen Coalgebren Vervollständigungen initialer Algebren? Nein! Es gibt durchaus informatikrelevante Entitäten, die man nicht als endliche Strukturen oder deren Fortsetzungen ins Unendliche (Suprema) beschreiben kann. Dazu gehören zum Beispiel Potenzmengen.

Modelltheoretisch ist der beobachtungsorientierte Ansatz schon recht gut untersucht, beweistheoretisch noch sehr wenig. Da, wie oben angedeutet, insbesondere verteilte Systeme und nebenläufige Prozesse naturgemäß in diesem Ansatz formalisierbar sind, wären für die Verifikation solcher Systeme beobachtungsorientierte Beweisverfahren von großer Bedeutung. Dazu müssen zunächst einmal Kalküle entwickelt werden, die bzgl. terminaler Modelle korrekt und vielleicht sogar vollständig sind. Wie definiert man z.B. Funktionen in einem terminalen Modell, das nicht, wie ein initiales, induktiv aufgebaut ist? Wieder ist es die Algebra-Coalgebra-Dualität, die eine Antwort auf diese Frage gibt.

Betrachten wir dazu noch einmal Beispiel 11.2.15: Die dort konstruierte Lösung E der Gleichung $E = 1 + (A \times E)$ ist nach Satz 11.2.9 das Zielobjekt der initialen G-Algebra, wobei G eine Menge B nach $1 + (A \times B)$ abbildet. Eine G-Algebra ist also eine Abbildung der Form $g : 1 + (A \times B) \to B$. Jede solche Abbildung ist eindeutig bestimmt durch zwei Inklusionen $1_g : 1 \to B$ und $cons_g : A \times B \to B$: $g(1) = 1_g(1)$ und $g((a,b)) = cons_g(a,b)$ für alle $a \in A$ und $b \in B$. Bezüglich der Darstellung des Zielobjektes der initialen G-Algebra $ini : 1 + (A \times E) \to E$ als Menge $E = A^*$ der endlichen Listen über A ist ini wie folgt definiert:

- $1_{ini}(1) =_{def} nil$,
- $cons_{ini}(a, L) =_{def} a : L$ für alle $a \in A$ und $L \in A^*$.

Da ini initial ist, gibt es zu jeder G-Algebra $\alpha : 1 + (A \times B) \to B$ eine eindeutige Abbildung $g : A^* \to B$ mit $g \circ ini = \alpha \circ G(g)$ (vgl. 11.2.7). Aus Anwendungen dieser Gleichung erhält man: Es gibt eine eindeutige Abbildung $g : A^* \to B$ mit

$$g(nil) = g(1_{ini}(1)) = g(ini(1)) = \alpha(G(g)(1)) = \alpha(1) = 1_\alpha(1),$$
$$g(a : L) = g(cons_{ini}(a, L)) = g(ini((a, L))) = \alpha(G(g)((a, L))) = \alpha((a, g(L))) = cons_\alpha(a, g(L))$$

[81] hier A, in 11.3.8 hieß diese Menge B.

für alle $a \in A$ und $L \in A^*$. Diese beiden Gleichungen definieren also die Funktion g eindeutig. Das ist eine Begründung dafür, daß Funktionen tatsächlich durch strukturelle Induktion (hier auf der Listenstruktur) definierbar sind. Jede Abbildung $g : E \to B$ mit $g \circ ini = \alpha \circ G(g)$ ist induktiv definiert, und umgekehrt: Für jede induktiv definierte Funktion $g : E \to B$ gilt $g \circ ini = \alpha \circ G(g)$.

Beispiel 11.5.6 (Konkatenation zweier Listen) Die Gleichungen

- $\mathbf{conc}(nil)(L') = L'$
- $\mathbf{conc}(a : L)(L') = a :\mathbf{conc}(L)(L')$

definieren $conc : A^* \to [A^* \to A^*]$ als eindeutige Abbildung vom Zielobjekt der G-Algebra ini (s.o.) in das Zielobjekt der G-Algebra $\alpha : 1 + (A \times B) \to B$ mit

- $B =_{def} [A^* \to A^*]$,
- $1_\alpha(1)(L') =_{def} L'$ für alle $L' \in A^*$,
- $cons_\alpha(a, L)(L') =_{def} a : (L + +L')$ für alle $a \in A$ und $L, L' \in A^*$. ✌

Fazit: **Induktion = Initialität**! Also auch **Terminalität = Coinduktion**? Aber was ist eine coinduktive Definition? Sehen wir uns dazu Beispiel 11.5.5 an. Da fin (s.o.) terminal ist, gibt es zu jeder F-Coalgebra $\alpha : B \to A \times B$ eine eindeutige Abbildung $f : B \to [\mathbb{N} \to A]$ mit $F(f) \circ \alpha = fin \circ f$. Aus Anwendungen dieser Gleichung erhält man: Es gibt eine eindeutige Abbildung $f : B \to D$ mit

$$\begin{aligned}
&head_{fin}(f(b)) = proj_1(fin(f(b))) = proj_1(F(f)(\alpha(b))) = proj_1(F(f)(head_\alpha(b), tail_\alpha(b)))\\
&= proj_1(head_\alpha(b), f(tail_\alpha(b))) = head_\alpha(b),\\
&tail_{fin}(f(b)) = proj_2(fin(f(b))) = proj_2(F(f)(\alpha(b))) = proj_2(F(f)(head_\alpha(b), tail_\alpha(b)))\\
&= proj_2(head_\alpha(b), f(tail_\alpha(b))) = f(tail_\alpha(b))
\end{aligned}$$

für alle $b \in B$.

Beispiel 11.5.7 (Alternierendes Mischen zweier Ströme) Die Gleichungen

- $head_{fin}(\mathbf{zips}(s, s')) = head_{fin}(s)$
- $tail_{fin}(\mathbf{zips}(s, s')) = \mathbf{zips}(s', tail_{fin}(s))$

definieren $zips : [\mathbb{N} \to A] \times [\mathbb{N} \to A] \to [\mathbb{N} \to A]$ als eindeutige Abbildung vom Quellobjekt der F-Coalgebra $\alpha : B \to A \times B$ in das Quellobjekt der F-Coalgebra fin (s.o.), wobei α wie folgt definiert ist:

- $B =_{def} [\mathbb{N} \to A] \times [\mathbb{N} \to A]$,
- $head_\alpha(s, s') =_{def} s(0)$ für alle $s, s' : \mathbb{N} \to A$,
- $tail_\alpha(s, s') =_{def} (s', tail_{fin}(s))$ für alle $s, s' : \mathbb{N} \to A$. ✌

Terminale Coalgebren lassen sich in O'Haskell [76] als Records (Typkonstruktor `struct`) implementieren. Die allgemeinen Schemata einer Recordtypdefinition, einer Recorddefinition und von Selektoraufrufen lauten wie folgt:

```
struct Record = selector1 :: type1
                selector2 :: type2
                ...

record = struct selector1 = term1
                selector2 = term2
                ...

a = record.selector1
b = record.selector2
```

Eine Funktion ist coinduktiv definiert, wenn sie in einen Recordtyp abbildet.

Beispiel 11.5.8 (unendliche Listen)

```
struct Infseq a = head_ :: a
                  tail_ :: Infseq a

x `app` s = struct head_ = x
                   tail_ = s

blink     = struct head_ = 0
                   tail_ = 1 `app` blink
```

Beispiel 11.5.9 (Ströme: endliche und unendliche Listen)

```
struct Stream a = ht :: Maybe (a,Stream a)

nats n     = struct ht = Just (n,nats (n+1))

s @@ s'    = struct ht = case s.ht of Just (x,t) -> Just (x,t@@s')
                                     _ -> s'.ht

zips s s'  = struct ht = case s.ht of Just (x,t) -> Just (x,zips s' t)
                                     _ -> s'.ht

maps f s   = struct ht = do (x,t) <- s.ht
                            Just (f s,maps f t)

exists g s = case s.ht of Just (x,t) -> g x || exists g t
                          _ -> False

nth 0 s    = do (x,t) <- s.ht
                Just x
nth n s    = do (_,t) <- s.ht
                x <- nth (n-1) t
                Just x
```

Beispiel 11.5.10 (Multimengen)

```
struct Bag a = card :: a -> Int

mtBag       = struct card = const 0

single x    = struct card y = if y == x then 1 else 0

b `union` c = struct card x = b.card x + c.card x
```

O'Haskell erlaubt subtyping (vgl. §4.1), sowohl für konstruktorbasierte Datentypen als auch für Records. Ein Subtyp `DatatypeS` eines Datentyps `Datatype` vergrößert durch Hinzunahme von Konstruktoren die durch `Datatype` denotierte Datenmenge. Ein Subtyp `RecordS` eines Records `Record` verkleinert durch Hinzunahme von Selektoren die durch `Record` denotierte Datenmenge. **Subtyping**

```
data DatatypeS > Datatype = constructorS1 typeS11 ... typeS1nS1 |
                            constructorS2 typeS21 ... typeS2nS2 |
                            ...

struct RecordS < Record = selectorS1 :: typeS1
```

```
                        selectorS2 :: typeS2
                        ...
```

Records lassen sich in O'Haskell mit dem Typkonstruktor `Template` zu Klassen erweitern. Die Erweiterung besteht in Zustandsvariablen, die nach dem Schlüsselwort `template` eingeführt und gleichzeitig initialisiert werden. Dann werden die Methoden der Klasse als Selektoren eines (zuvor deklarierten) Records definiert. Sie dürfen Zustandsvariablen verändern und müssen einen der monadischen Ergebnistypen `Action` oder `Request a` haben.

```
Action     < Cmd ()
Request a  < Cmd a
Template a < Cmd a

struct Methods = method1 :: type11 ... type1n1 -> Action
                 method2 :: type21 ... type2n2 -> Request type2
                 ...

class :: type1 -> type2 -> ... -> Template Record
```

Es folgen vier Schemata zur Definition der Klasse `class`. Falls Initialisierungen von Zustandsvariablen bzw. Methodendefinitionen Hilfsfunktionen benötigen, muss man deren Definition bei (1) bzw. (2) unterbringen. Bei wechselseitiger Rekursion von Methoden und anderen Funktionen muss man dem vierten Schema folgen und letztere bei (3) unterbringen. Selbstverständlich kann das erste Schema mit dem zweiten oder dritten kombiniert werden.

```
class x1 x2 ... = template -- initializations of state variables
                           stateVar1 := term1
                           stateVar2 := term2
                           ...
                  in struct -- method definitions
                            method1 = action  monad_term1
                            method2 = request monad_term2
                            ...

class x1 x2 ... = let -- local definitions                          (1)
                  in template -- initializations of state variables
                     in struct -- method definitions

class x1 x2 ... = template -- initializations of state variables
                  in let -- local definitions                      (2)
                     in struct -- method definitions

class x1 x2 ... = template -- initializations of state variables
                  in let -- local definitions                      (3)
                         -- method definitions
                     in struct ..Record

a <- class a1 a2 ...
```

Literatur

[1] *Abstract State Machines*, `www.eecs.umich.edu/gasm`

[2] S. Alagic, M.A. Arbib, *The Design of Well-structured and Correct Programs*, Springer 1978

[3] S. Antoy, R. Echahed, M. Hanus,*A Needed Narrowing Strategy*, Journal of the ACM 47 (2000) 776-822

[4] K.R. Apt, *Logic Programming*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 493-574

[5] K.R. Apt, E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer 1991; deutsch: *Programmverifikation*, Springer 1994

[6] `www-formal.stanford.edu/clt/ARS/systems.html` enthält Links zu einer Vielzahl zur Zeit verfügbarer automatischer Beweiser (*automated reasoning systems*)

[7] E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Report, Springer 1999

[8] J. Avenhaus, *Reduktionssysteme*, Springer 1995

[9] J. Avenhaus, U. Kühler, T. Schmidt-Samoa, C.-P. Wirth, *How to Prove Inductive Theorems? QuodLibet!*, Proc. CADE '03, Springer LNAI 2741 (2003) 328-333

[10] F. Baader, T. Nipkow, *Term Rewriting and All That*, Cambridge University Press 1998

[11] J. Backus, *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Comm. ACM 21 (1978) 613-641

[12] J.C.M. Baeten, W.P. Weijland, *Process Algebra*, Cambridge University Press 1990

[13] R. Barnett, D. Basin, J. Hesketh, *A Recursion Planning Analysis of Inductive Completion*, Report MPI-I-94-230, Max-Planck-Institut für Informatik Saarbrücken 1994

[14] E. Best, *Semantik: Theorie sequentieller und paralleler Programmierung*, Vieweg 1995

[15] Ch. Beierle, G. Kern-Isberner, *Methoden wissensbasierter Systeme*, Vieweg 2003

[16] K.H. Bläsius, H.-J. Bürckert, *Deduktionssysteme*, 2. Auflage, Oldenbourg 1992

[17] M. Bidoit, P.D. Mosses, *CASL User Manual*, Springer LNCS 2900 (2004)

[18] A. Bouhoula, E. Kounalis, M. Rusinowitch, *Automated Mathematical Induction*, J. Logic and Computation 5 (1995) 631-668

[19] R.S. Boyer, J.S. Moore, *A Computational Logic*, Academic Press 1979

[20] W.S. Brainerd, L.H. Landweber, *Theory of Computation*, John Wiley & Sons 1974

[21] R. Bündgen, *Termersetzungssysteme: Theorie, Implementierung, Anwendung*, Vieweg 1998

[22] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, A. Smaill, *Rippling: A Heuristic for Guiding Inductive Proofs*, J. Artificial Intelligence 62 (1993) 185-253

[23] A.J.T. Davie, *An Introduction to Functional Programming System using Haskell*, Cambridge University Press 1992

[24] R. Diaconescu, K. Futatsugi, *CafeOBJ Report*, World Scientific 1998

[25] H.-D. Ehrich, M. Gogolla, U.W. Lipeck, *Algebraische Spezifikation abstrakter Datentypen*, Teubner 1989

[26] H. Ehrig, B. Mahr, F. Cornelius, M. Große-Rhode, P. Zeitz, *Mathematisch-strukturelle Grundlagen der Informatik*, Springer 1999

[27] E.A. Emerson, *Temporal and Modal Logic*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 995-1072

[28] M. Erwig, *Grundlagen funktionaler Programmierung*, Oldenbourg 1999

[29] Elfriede Fehr, *Semantik von Programmiersprachen*, Springer 1989

[30] A.J. Field, P.G. Harrison, *Functional Programming*, Addison-Wesley 1988

[31] M. Fitting, *First-order Logic and Automated Theorem Proving*, Springer 1990

[32] M. Fitting, R. Mendelsohn, *First-order Modal Logic*, Kluwer 1998

[33] Th. Frühwirth, S. Abdennadher, *Constraint-Programmierung*, Springer 1997

[34] J.A. Goguen, J. Meseguer, *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, in: B. Shriver, P. Wegner, eds., Research Directions in Object-Oriented Programming, MIT Press (1987) 417-477

[35] J.A. Goguen, G. Malcolm, *Algebraic Semantics of Imperative Programs*, The MIT Press 1996

[36] J.A. Goguen, G. Malcolm, *A Hidden Agenda*, UCSD Technical Report CS97-538, 1997, siehe `www-cse.ucsd.edu/users/goguen/projs/halg.html`

[37] P. Glavan, D. Rosenzweig, *Communicating Evolving Algebras*, Proc. CSL'92, Springer LNCS 702 (1993) 186-215

[38] A.D. Gordon, *A Tutorial on Co-induction and Functional Programming*, Proc. Functional Programming Glasgow 1994, Springer (1995) 78-95,
`ftp://ftp.cl.cam.ac.uk/papers/adg/fp94.ps.gz`

[39] Andrew D. Gordon, *Bisimilarity as a Theory of Functional Programming*, Theoretical Computer Science 228 (1999) 5-47

[40] S. Gregory, *Parallel Logic Programming in PARLOG*, Addison-Wesley (1987) 202-260

[41] J.F. Groote, F. Vaandrager, *Structured Operational Semantics and Bisimulation as a Congruence*, Information and Computation 100 (1992) 202-260

[42] C.A. Gunter, *Semantics of Programming Languages*, The MIT Press 1992

[43] J.V. Guttag, J.J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer 1993

[44] T. Hagino, *Codatatypes in ML*, J. Symbolic Computation 8 (1989) 629-650

[45] C. Hankin, *Lambda Calculi: A Guide for Computer Scientists*, Clarendon Press 1994

[46] H. Herrlich, G.E. Strecker, *Category Theory*, Allyn and Bacon 1973

[47] R. Hinze, *Einführung in die funktionale Programmierung mit Miranda*, Teubner 1992

[48] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall 1985

[49] H. Hofbauer, R. Kutsche, *Grundlagen des maschinellen Beweisens*, 2. Auflage, Vieweg 1991

[50] P. Hudak, J. Peterson, J.H. Fasel, *A Gentle Introduction to Haskell 98*, www.haskell.org/tutorial, Yale and Los Alamos 2000

[51] D. Hutter, *Guiding Induction Proofs*, Proc. CADE '90, Springer LNCS 449 (1990) 147-161

[52] B. Jacobs, J. Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (June 1997) 222-259, `www.cs.kun.nl/~bart/PAPERS/JR.ps.Z`

[53] D. Kozen, J.Tiuryn, *Logic of Programs*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 789-840

[54] H.-J. Kreowski, *Logische Grundlagen der Informatik*, Oldenbourg 1991 (in der Lehrbuchsammlung erhältlich)

[55] L. Lamport, *The Temporal Logic of Actions*, Transactions of Progr. Lang. and Syst. 16 (1994) 308-320

[56] P.J. Landin, *The Mechanical Evaluation of Expressions*, BCS Computing Journal 6 (1964) 872-923

[57] P.G Larsen, *The VDM Bibliography*, IFAD Denmark (1996) `ftp://ftp.ifad.dk/pub/vdm/vdmbib.ps.gz`

[58] J. v. Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, Elsevier 1990, Kap. 10-12

[59] J.W. Lloyd, *Foundations of Logic Programming*, 2nd edition, Springer (1987)

[60] D.J. Lehmann, M.B. Smyth, *Algebraic Specification of Data Types: A Synthetic Approach*, Mathematical Systems Theory 14 (1981) 97-139

[61] J. Loeckx, K. Sieber, *The Foundations of Program Verification*, Teubner 1987

[62] J. Loeckx, H.-D. Ehrich, M. Wolf, *Specification of Abstract Data Types*, Wiley-Teubner 1996

[63] R. Loogen, *Integration funktionaler und logischer Programmiersprachen*, Oldenbourg 1995

[64] D. MacQueen, G. Plotkin, R. Sethi, *An Ideal Model for Recursive Polymorphic Types*, Information and Control 71 (1986) 95-130

[65] Z. Manna, *Mathematical Theory of Computation*, McGraw-Hill 1974

[66] E.G. Manes, M.A. Arbib, *Algebraic Approaches to Program Semantics*, Springer 1986

[67] J. Meseguer, *A Logical Theory of Concurrent Objects and its Realization in the Maude Language*, in: G. Agha, P. Wegner, A. Yonezawa, eds., Research Directions in Object-Based Concurrency, MIT Press (1993) 313-389

[68] R. Milner, *Communication and Concurrency*, Prentice-Hall 1989

[69] R. Milner, J. Parrow, D. Walker, *A Calculus of Mobile Processes*, Information and Control 100 (1992) 1-77

[70] J.C. Mitchell, *Foundations of Programming Languages*, The MIT Press 1996

[71] B. Möller, H. Partsch, S. Schuman, eds., *Formal Program Development*, IFIP WG 2.1 State-of-the-Art Report, Springer LNCS 755 (1993)

[72] A. Nerode, R.A. Shore, *Logic for Applications*, 2nd edition, Springer (1997)

[73] M. Müller-Olm, D. Schmidt, B. Steffen, *Model Checking: A Tutorial Introduction*, Proc. SAS '99, Springer LNCS 1694 (1999) 330-394

[74] S.-H. Nienhuys-Cheng, R. de Wolf, *Foundations of Inductive Logic Programming*, Springer LNAI 1228 (1997)

[75] T. Nipkow, L.C.Paulson, M. Wenzel, *Isabelle/HOL*, Springer LNCS 2283 (2002)

[76] M.P. Jones, J. Nordlander, B. v. Sydow, M. Carlsson, *A Survey of O'Haskell*, www.cs.chalmers.se/ nordland/ohaskell/survey.html, 2001

[77] P. Padawitz, *Einführung ins funktionale Programmieren*, Vorlesungsskript, fldit-www.cs.uni-dortmund.de/∼peter/ProgNeu.ps.gz

[78] P. Padawitz, *Übersetzerbau*, Vorlesungsskript, fldit-www.cs.uni-dortmund.de/∼peter/Cbau.pdf

[79] P. Padawitz, *Computing in Horn Clause Theories*, Springer 1988

[80] P. Padawitz, *Deduction and Declarative Programming*, Cambridge University Press 1992

[81] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, fldit-www.cs.uni-dortmund.de/∼peter/Expander2.html

[82] P. Padawitz, *Expander2: Towards a Workbench for Interactive Formal Reasoning*, fldit-www.cs.uni-dortmund.de/∼peter/Expander2/Chiemsee.pdf

[83] P. Padawitz, *Inductive Theorem Proving for Design Specifications*, J. Symbolic Computation 21 (1996) 41-99

[84] P. Padawitz, *Proof in Flat Specifications*, in [7]

[85] P. Padawitz, *Swinging Types = Functions + Relations + Transition Systems* (siehe `fldit-www.cs.uni-dortmund.de/∼peter/Swinging.html`)

[86] P. Padawitz, *Swinging Types at Work*, fldit-www.cs.uni-dortmund.de/∼peter/BehExa.pdf

[87] P. Padawitz, *Dialgebraic Specification and Modeling*, fldit-www.cs.uni-dortmund.de/∼peter/Dialg.pdf

[88] L.C. Paulson, *Isabelle: A Generic Theorem Prover*, Springer LNCS 828 (1994)

[89] L.C. Paulson, *ML for the Working Programmer*, 2nd edition, Cambridge University Press 1996

[90] A. Pettorossi, *Derivation of Efficient Programs for Computing Sequences of Actions*, Theoretical Computer Science 53 (1987) 151-167

[91] A. Pettorossi, M. Proietti, *Rules and Strategies for Program Transformation*, in [71], 263-304

[92] A. Pettorossi, M. Proietti, *A Comparative Revisitation of Some Program Transformation Techniques*, in O. Danvy, R. Glück, P. Thiemann, eds, Partial Evaluation, Dagstuhl Seminar, Springer LNCS 1110 (1996) 355-385

[93] A. Pettorossi, A. Skowron, *Higher-Order Generalization in Program Derivation*, Proc. TAPSOFT '87, Springer LNCS 250 (1987) 182-196

[94] V. Pratt, *Semantical Considerations of Floyd-Hoare Logic*, Proc. IEEE FOCS '76 (1976) 109-121

[95] B.C. Pierce, *Basic Category Theory for Computer Scientists*, The MIT Press 1991

[96] C. Reade, *Elements of Functional Programming*, Addison-Wesley 1989, Kap. 10 und 12

[97] W. Reif, G. Schellhorn, K. Stenzel, M. Balser. *Structured Specifications and Interactive Proofs with KIV*, in: W. Bibel and P. H. Schmitt, eds., Automated Deduction - A Basis for Applications, Vol 2., Kluwer Academic Publishers 1998

[98] M.M. Richter, *Logikkalküle*, Teubner 1978

[99] M.M. Richter, *Prinzipien der Künstlichen Intelligenz*, Teubner 1989

[100] J.J.M.M. Rutten, *Universal Coalgebra: A Theory of Systems*, Report CS-R9652, CWI, SMC Amsterdam 1996

[101] U. Schöning, *Logik für Informatiker*, BI 1989

[102] R. Sethi, *Programming Languages - Concepts and Constructs*, Addison-Wesley 1989

[103] D. Siefkes, *Formalisieren und Beweisen: Logik für Informatiker*, Vieweg 1992

[104] M.B. Smyth, G.D. Plotkin, *The Category-Theoretic Solution of Recursive Domain Equations*, Siam J. Computing 11 (1982) 761-783

[105] J.M. Spivey, *The Z Notation: A Reference Manual*, Oriel College, Oxford (1998), `spivey.oriel.ox.ac.uk/ mike/zrm`

[106] C. Stirling, *Modal and Temporal Logics*, in: S. Abramsky et al., eds., Handbook of Logic in Computer Science, Clarendon Press (1992) 477-563

[107] C. Stirling, *Modal and Temporal Logics for Processes*, in: F. Moller, B. Birtwistle, eds., Logics for Concurrency, Springer LNCS 1043 (1996) 149-237

[108] P. Thiemann, *Grundlagen der funktionalen Programmierung*, Teubner 1994

[109] R. Turner, *Constructive Foundations for Functional Languages*, McGraw-Hill 1991

[110] Ph. Wadler, *Deforestation: Transforming Programs to Eliminate Trees*, Proc. ESOP '88, Springer LNCS 300 (1988) 344-358

[111] M. Wand, *Specifications, Models, and Implementations of Data Abstractions*, Theoretical Computer Science 20 (1982) 3-32

[112] D. Watt, *Programming Language Concepts and Paradigms*, Prentice Hall 1990; deutsch: *Programmiersprachen - Konzepte und Paradigmen*, Hanser 1996

[113] M. Wirsing, *Algebraic Specification*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 675-788

[114] C.-P. Wirth, *Descente Infinie + Deduction*, Logic Journal of the IGPL 12, No. 1 (2004) 1-96

Index