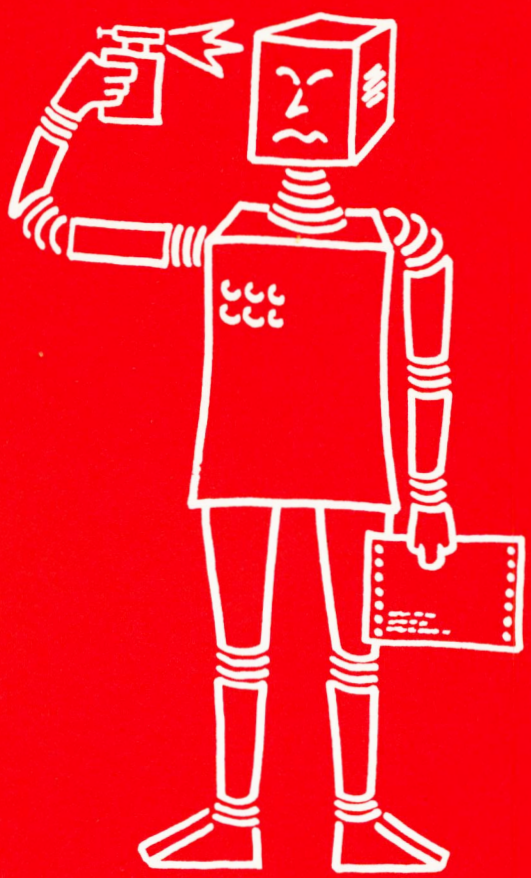
SEKI-PROJEKT

SEKI MEMO

Institut für Informatik III
Universität Bonn
Bertha-von-Suttner-Platz 6
D 5300 Bonn 1, W. Germany

Institut für Informatik I
Universität Karlsruhe
Postfach 6380
D-7500 Karlsruhe 1, W. Germany



Korrekte Implementierung abstrakter Datentypen durch Moduln in höheren Programmiersprachen

von

Ulrich Guntram

Universität Bonn
Institut für Informatik III
Bertha-von-Suttner-Platz 6
D-5300 Bonn 1

KORREKTE IMPLEMENTIERUNG ABSTRAKTER DATENTYPEN DURCH MODULN IN HÖHEREN PROGRAMMIERSPRACHEN

VON

ULRICH GUNTRAM

Memo SEKI-BN-80-09

Juni 1980

INHALTSVERZEICHNIS

0. VORBEMERKUNGEN

0.1 EINLEITUNG

Ende der 60er- und Anfang der 70er-Jahre war man im Rahmen des Strukturellen Programmierens bestrebt, in Programmiersprachen und Programme Abstraktionsmechanismen einzubauen.
Die Abstraktion wurde dabei als ein Gliederungsmittel aufgefaßt, um das Gemeinsame an einer Menge von Objekten, Situationen oder Abläufen zu erfassen.
Dies führte zum Begriff des Moduls, der ohne Kenntnis seines inneren Aufbaus in eine beliebige Umgebung eingesetzt werden kann und dessen Entwurf ohne Kenntnis der Umgebung möglich ist. Alle notwendigen Informationen müssen auf der Schnittstelle des Moduls verfügbar sein ([GOO 76]).

Damit wurde die Abstraktion zu einem Hilfsmittel des Programmentwurfs. Man versuchte daher, die Abstraktion durch sprachliche Mittel zu unterstützen.
Dies geschah etwa in der begrifflichen Erfassung und Bezeichnung einer zusammengesetzten Operation. Operationen können somit als Ganzes in eine Umgebung eingesetzt werden, wobei nur ihre Wirkung bedeutungsvoll ist.
Die sprachliche Realisierung dieser operativen Abstraktion geschieht in den Prozeduren einer operativen Programmiersprache.

Es liegt nahe, diesen Ansatz zu erweitern, indem die Abstraktion von Operationen mit einer abstrakten Erfassung der Objekte, die diese Operationen manipulieren, gekoppelt wird. Dies führt dann zum Begriff des abstrakten Datentyps, der durch eine Menge von Objekten und den auf ihnen definierten Operationen gegeben ist. Dabei wird sowohl von der Realisierung der Objekte als auch vom Aufbau der Operationen als Algorithmen völlig abstrahiert. Von einer Operation ist nur noch ihre Wechselwirkung mit anderen Operationen interessant.

Sprachlich wirkte sich dieser Ansatz aus im Klassenkonzept von SIMULA, den entsprechenden Konzepten in den SIMULA-Nachfolgern (z.B. CLU und ALPHARD) und im Entwurf neuerer Programmiersprachen wie OBJ oder CSSA. Dabei enthält die letzte Sprache schon Konzepte zur Abstraktion von Kontrollstrukturen.

Parallel zur oben beschriebenen Einführung von Abstraktionsmechanismen in Programmiersprachen wird seit Anfang der 70er-Jahre die dafür notwendige mathematische Theorie entwickelt. Das Ziel der Bemühungen in dieser Richtung ist eine formale, mathematische Definition des Begriffs "abstrakter Datentyp".
Von den verschiedenen Ansätzen, die zur Lösung dieser Frage entwickelt wurden, ist einer die algebraische Beschreibung eines Datentyps. Die mathematischen Grundlagen dafür wurden bereits in den frühen Siebzigerjahren von der ADJ-Gruppe bereitgestellt. Popularisiert wurde die Theorie der abstrakten Datentypen durch die Dissertation von Guttag:" The Specification and Application to Programming of Abstract Data Types" ([GUT 75]). Anwendung findet die Theorie der abstrakten Datentypen beim Entwurf von Programmiersprachen und Programmen sowie beim Nachweis der Korrektheit eines Programms.

Ein abstrakter Datentyp wird definiert durch eine Spezifikation, die einen syntaktischen Teil, genannt Signatur, enthält und eine Menge E von Gleichungen zwischen Termen, die aufgrund der syntaktischen Angaben gebildet werden können und in ihrer Gesamtheit eine Algebra bilden, die Termalgebra genannt wird. Die Semantik eines abstrakten Datentyps ist dann die nach einer Kongruenz, die die Gleichungen E enthält, faktorisierte Termalgebra. Bei der Definition dieser Kongruenz gibt es zwei zueinander duale Sichtweisen:

Der initiale Ansatz: Hier werden nur so viele Terme identifiziert, wie unbedingt notwendig ist, damit die Gleichungen noch gelten.

Der terminale Ansatz: Hier werden möglichst viele Terme identifiziert, die in noch anzugebender Weise in ihrer Wirkung nicht unterschieden werden können.

Der terminale Ansatz ist vorallem geeignet für die Beschreibung praktischer Verhältnisse, da ein Programmierer nicht daran interessiert ist, mehrere Realisierungen eines Objektes mit derselben Wirkung zu haben.
Beide Ansätze, der initiale und der terminale, werden in dualer Vorgehensweise kurz in Kapitel 1 vorgestellt.

Gegenstand dieser Arbeit ist nun das Verhältnis einer gegebenen algebraischen Spezifikation eines Datentyps zu einer programmiersprachlichen Realisierung.
Bisher hat lediglich die Intention und Intuition des Programmierers darüber entschieden, ob beide "dasselbe bedeuten", ob also ein Programmstück eine "korrekte" Implementierung des algebraisch spezifizierten Datentyps ist. Ein allgemeines, formales Kriterium stand seither nicht zur Verfügung, um diese Frage zu beantworten. In der vorliegenden Arbeit wird der Versuch unternommen, solch ein Kriterium zu entwickeln.

Das gewünschte Korrektheitskriterium wird hergeleitet anhand von drei Beispielsprachen. Diese sind CLU und APLPHARD als Vertreter der "konventionellen" Programmiersprachen (SIMULA-Erben). Es wird sich zeigen, daß beide Sprachen in den hier relevanten Konzepten so ähnlich sind, daß sie durch eine gemeinsame fiktive Mischsprache CLALPHARD beschrieben werden. Als Beispiel einer interaktiven Programmiersprache für asynchrone Multiprozessorsysteme dient CSSA.
Für diese Sprachen wird in Kapitel 2 eine kurze Beschreibung, abstrakte Syntax und abgemagerte denotationale Semantik gegeben.
In CLU, ALPHARD und CSSA sind die Sprachkonzepte zur Definition eines Datentyps das Cluster, Form bzw. Script. In der Semantik werden daraus Moduln erzeugt, die Paare bestehend aus einem Verhalten und einem lokalen Zustand sind. Der lokale Zustand kann durch Operationen geändert werden.
Eine Menge von Moduln bildet eine Modulgemeinschaft, in der die Moduln durch Locations identifiziert werden, was eine Umgebung sein soll (vgl. Speicherbelegung).

Das Ziel ist es nun, der Menge aller Moduln eine algebraische Struktur zu unterlegen, wobei die Struktur der Termalgebra als Orientierung dient.
Als Schwierigkeit wird sich dabei die notwendige Berücksichtigung der Umgebungen erweisen.
Die so erhaltene Algebra, die Modulalgebra, wird in 3.2 definiert.
Die Verbindung zwischen einem Term der Termalgebra und einem Element der Modulalgebra stellt die Auswertungsfunktion EVAL in 3.3 her.

Bis hierher wurde nur die Syntax eines abstrakten Datentyps berücksichtigt. Um die Gleichungen zwischen den Termen der Termalgebra in der Modulalgebra widerzuspiegeln, muß zunächst auf dieser eine Gleichheitsrelation definiert

werden, die zwei Elemente der Modulalgebra identifiziert, wenn sie in ihrer Wirkung nicht unterschieden werden können (vgl. 3.4).

Das oben erwähnte Korrektheitskriterium besteht dann darin, daß verhaltensgleiche Terme der Termalgebra verhaltensgleiche Bilder unter EVAL in der Modulalgebra haben und daß dazuhin die EVAL-Bilder unterscheidbarer Terme auch in der Modulalgebra unterscheidbar sind (vgl. 3.5).

In Kapitel 4 wird dieses Kriterium erweitert auf typparametrisierte Datentypen.
Zu diesem Zweck wird in 4.1 zunächst eine algebraische Theorie entwickelt, die ein duales Vorgehen im initialen und terminalen Fall zuläßt. Ein parametrisierter Datentyp wird dabei aufgefaßt als Transformation zwischen dem Datentyp, der als aktueller Parameter dient, und dem resultierenden aktuell parametrisierten Datentyp.

In 4.2 und 4.3 wird die Parametrisierung in den Beispielsprachen CLU, CSSA und ALPHARD untersucht und deren Syntax bzw. Semantik aus Kapitel 2 soweit erforderlich ergänzt.
Das Korrektheitskriterium für den abstrakten parametrisierten Datentyp baut auf dem aus 3.5 insofern auf, als die jeweiligen Programmstücke die aktuellen Parameterspezifikationen und die zugehörigen Spezifikationen der resultierenden parametrisierten Datentypen korrekt implementieren müssen. Da bei parametrisierten Datentypen nicht eine, sondern vier Spezifikationen bzw. Signaturen beteiligt sind, fallen die entsprechenden syntaktischen Anforderungen an die Programmkonstrukte umfangreicher aus.

Obwohl die vorliegende Arbeit ein Problem aus dem Grenzbereich von Theorie und Praxis behandelt, ist sie in dieser Form nur von akademischem Wert.
Die sich unmittelbar aus den vorliegenden Ergebnissen ergebende Aufgabe ist die Konstruktion eines Verfahrens, wie von einem gegebenen Programm konkret die Erfüllung des hier entwickelten Korrektheitskriteriums nachgewiesen werden kann.

"Nur aus der Spannung, aus dem Spiel zwischen
der Fülle der Tatsachen und den vielleicht
dazu passenden mathematischen Formeln können
die entscheidenden Fortschritte kommen".

Werner Heisenberg in
"Quantentheorie und Philosophie"

0.2 Notation

In diesem Abschnitt wird die in dieser Arbeit verwendete Notation vorgestellt. Abkürzende Schreibweisen für bestimmte Sachverhalte werden noch an späterer Stelle eingeführt.

0.2.1 Logische Operatoren und Quantoren

Quantoren:	$\forall$	Allquantor	("für alle ... ")
	$\exists$	Existenzquantor	("es existiert ...")
	$\exists_1$	eindeutiger Existenzquantor	("es existiert genau ein...")

Logische Operatoren:

$\wedge$	UND
V	ODER
-	Negation
—>	Implikation
<-->	Äquivalenz

In logischen Ausdrücken hat die Klammerung höchste Priorität; die logischen Operatoren binden in obiger Reihenfolge.

In Beweisen wird für die Implikation ("hieraus folgt") das Zeichen "==>" verwendet.
Definitionen werden eingeführt durch die Abkürzung ":gdw" (genau dann wenn).
Das Ende eines Beweises wird angezeigt durch das Zeichen "MMM".

0.2.2 Mengen

+	disjunkte Vereinigung
$N_0 := \{0,1,2,...\}$	Menge der nicht-negativen ganzen Zahlen
$N := \{1,2,3,...\}$	Menge der positiven ganzen Zahlen

Seien $A_1, A_2, ...,A_n$,$n \in N$, Mengen.

$\|A_1\|$	bezeichnet die Kardinalität von A_1
$A_1 x A_2 x ... x A_n$	ist das kartesische Produkt der Mengen $A_1,...,A_n$, dessen Elemente n-Tupel bzw. Listen der Länge n sind.

Auf die Menge an der i-ten Stelle eines kartesischen Produktes wird durch den Projektionsoperator "↓" zugegriffen:

$(\forall n\in N)(\forall K_n=A_1x\ldots xA_n)(\forall i\in N)$

$$K_n\downarrow i:=\begin{cases}A_i & \text{falls } 1\leq i\leq n\\ \text{undefiniert} & \text{sonst}\end{cases}$$

Auf die i-te Komponente einer Liste wird durch Indizierung zugegriffen:

$(\forall n\in N)(\forall l:=(a_1,\ldots a_n)\in A_1x\ldots xA_n)(\forall i\in N)$

$$l_i:=\begin{cases}a_i & \text{falls } 1\leq i\leq n\\ \text{undefiniert} & \text{sonst}\end{cases}$$

Zuweilen wird statt der Schreibweise $(a_1,\ldots,a_n)\in A^n$ als Liste auch die Darstellung $a_1\ldots.a_n$ als Wort (der von A erzeugten Worthalbgruppe mit der Konkatenation als Verknüpfung) verwendet.

ε	leeres Wort
$A^{\{0,1\}}:=\{\varepsilon\}\cup A$	
$A^+ := \bigcup_{n\in N} A^n$	
$A^* := \bigcup_{n\in N_0} A^n = A^+\cup\{\varepsilon\}$	Menge aller Wörter über A

$\perp$ ist ein allgemeines Fehlerelement bzw. ein Zeichen für "undefiniert", dessen genaue Bedeutung jeweils an Ort und Stelle angegeben wird. In Kapitel 2 wird "$\perp$" auch für die leere Liste verwendet.

0.2.3 Relationen und Funktionen

Seien A und B Mengen.
Sei $R\subseteq AxB$ eine zweistellige Relation. Dann wird für'(a,b)$\in$ R'auch die Infixschreibweise 'aRb' verwendet.

id_A	identische Funktion auf einer Menge A

Sei $f: A \longrightarrow B$, $g: B \longrightarrow C$

$g\circ f: A \longrightarrow C$	bezeichnet die Komposition von f und g
dom(f) := A	der Definitionsbereich von f
bd(f) := B	der Bildbereich von f

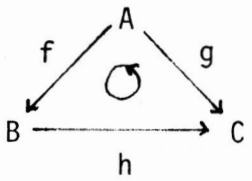
Funktionen $f: A_1x\ldots xA_n \longrightarrow A$ werden in Kapitel 2 auch aufgefaßt als Funktionen $f\in A_1 \longrightarrow [A_2 \longrightarrow [\ldots \longrightarrow [A_n \longrightarrow A]\ldots]$ wobei $A \longrightarrow B$ die Menge aller Funktionen von A nach B ist.

Abbildungsdiagramme:

$f: A \hookrightarrow B$ f injektiv

$f: A \hookrightarrow B$ f partielle Funktion, dh. $dom(f) \subseteq A$.

Seien $f: A \longrightarrow B$, $g: A \longrightarrow C$, $h: B \longrightarrow C$ mit $g = h \circ f$; dann kommutiert das Abbildungsdiagramm



1. ABSTRAKTE DATENTYPEN

In diesem Kapitel werden die für diese Arbeit notwendigen Begriffe aus der Theorie der abstrakten Datentypen eingeführt sowie die initiale und terminale Semantik von abstrakten Datentypen vorgestellt. Dabei wird von der Darstellung in [HOR 79], [RAU 79] und [ADJ 77] ausgegangen, wo auch die detaillierten Beweise der zitierten Lemmata und Sätze zu finden sind.

1.1 Grundlegende Definitionen

Ein abstrakter Datentyp kann aufgefaßt werden als eine Familie von Objektmengen, deren Elemente durch Operationen auf diesen Mengen definiert sind. Die syntaktischen Beziehungen zwischen den Objektmengen und den Operationen werden in einer Signatur wiedergegeben.

1.1.1 Signaturen und Σ-Algebren

1.1.1.1 Definition (Signatur)

Eine *Signatur* ist ein Paar (S,Σ) mit

- S ist eine Menge von *Sorten*
- $\Sigma := \{\Sigma_{w,s} : w \in S^*, s \in S\}$ ist eine Familie von paarweise disjunkten Mengen

Für $w \in S^*$, $s \in S$ heißt $f \in \Sigma_{w,s}$ *Operationssymbol der Stelligkeit, Funktionalität* (w,s).

Notation:

1.) Seien $(S,\Sigma),(S',\Sigma')$ Signaturen mit $S \subseteq S'$ und $(\forall s \in S)(\forall w \in S^*)(\Sigma_{w,s} \subseteq \Sigma'_{w,s})$.
Dann wird dafür abkürzend gschrieben: $\Sigma \subseteq \Sigma'$.

2.) Seien (S,Σ) und (S',Σ') Signaturen.
Dann ist $\Sigma \cap \Sigma' = \emptyset$ eine abkürzende Schreibweise für
$(\forall s \in S, s' \in S')(\forall w \in S^*, w' \in S'^*)\ (\Sigma_{w,s} \cap \Sigma'_{w',s'} = \emptyset)$.

3.) Sei (S,Σ) eine Signatur.
Dann ist "$f \in \Sigma$" eine abkürzende Schreibweise für
$(\exists s \in S)(\exists w \in S^*)(f \in \Sigma_{w,s})$.

1.1.1.2 Definition (Σ-Algebra)

Sei (S,Σ) eine Signatur.

Eine Σ-*Algebra* A ist ein Paar (M,G) mit

- $M=\{A_s : s\in S\}$ die Menge der *Trägermengen*
- $G=\{f_A : f\in\Sigma_{w,s},\ w\in S^*,\ s\in S\}$ eine Menge von Funktionen mit
  - $(\forall n\in N)(\forall s,s1,\ldots sn \in S)(\forall f\in\Sigma_{s1\ldots sn,s})(f_A : A_{s1}x\ldots xA_{sn} \longrightarrow A_s)$
  - $(\forall s\in S)(\forall f\in\Sigma_{\varepsilon,s})\ f_A: \longrightarrow A_s$ ist ein Element aus A_s.

1.1.1.3 Definition (Termalgebra)

Sei (S,Σ) eine Signatur.

1. $(\forall s\in S)$
   a) $T_{\Sigma,s,0} := \Sigma_{\varepsilon,s}$
   b) $(\forall k\in N_0)$

$$T_{\Sigma,s,k+1} := \{f(t_1,\ldots t_n) : n\in N;\ s1,\ldots,sn\in S,\ f\in\Sigma_{s1\ldots sn,s},\ t_i\in \bigcup_{0\leq j\leq k} T_{\Sigma,si,j}\ \ i=1\ldots n\}$$

2. $(\forall s\in S)\ T_{\Sigma,s} := \bigcup_{k\in N_0} T_{\Sigma,s,k}$, $t\in T_{\Sigma,s}$ heißt *Term der Sorte S.*

3. $T_\Sigma := (\{T_{\Sigma,s} : s\in S\},\{f_{T_\Sigma} : f\in\Sigma_{w,s},\ w\in S^*,\ s\in S\})$

heißt die durch (S,Σ) induzierte *Termalgebra.*

1.1.1.4 Definition (Unter-,Teilalgebra)

Seien (S,Σ), (S',Σ') Signaturen, A eine Σ-Algebra und A' eine Σ'-Algebra.

A' heißt *Unteralgebra* von A :gdw

1. $S'\subseteq S$
2. $(\forall s\in S')(A'_s\subseteq A_s)$
3. $(\forall n\in N_0)(\forall s,s1,\ldots,sn\in S)(\forall f\in\Sigma'_{s1\ldots sn,s})(\forall a\in A'_{s1}x\ldots xA'_{sn})$

$$(f\in\Sigma_{s1\ldots sn,s} \wedge f_{A'}(a)=f_A(a))$$

Spezielle Unteralgebren einer Algebra erhält man durch die Einschränkung:

1.1.1.5 Definition / Corollar (Einschränkung)

Seien (S,Σ) eine Signatur und A eine Σ-Algebra.

Sei (S',Σ') eine Signatur mit $S'\subseteq S$ und $(\forall w\in S'^*)(\forall s\in S')(\Sigma'_{w,s}\subseteq\Sigma_{w,s})$.

Dann ist $A|_{\Sigma'} := (\{A_s : s\in S'\},\{f_A : f\in\Sigma'_{w,s},\ w\in S'^*,\ s\in S'\})$ die

Einschränkung von A auf (S',Σ').

$A|_{\Sigma'}$ ist eine Σ'-Unteralgebra von A.

Im folgenden werden strukturerhaltende Abbildungen zwischen Σ-Algebren eingeführt.

1.1.1.6 Definition (Σ-Homomorphismen)

Seien (S,Σ) eine Signatur, A und B Σ-Algebren.

a) $H:=\{h_s:A_s \longrightarrow B_s,\ s\in S\} : A \rightarrow B$ ist *Σ-Homomorphismus* :gdw

1. $(\forall s\in S)(\forall f\in\Sigma_{\varepsilon,s})(h_s(f_A)=f_B)$
2. $(\forall n\in\mathbb{N})(\forall s,s1,\ldots sn\in S)(\forall(a1,\ldots an)\in A_{s1}\times\ldots\times A_{sn})(\forall f\in\Sigma_{s1\ldots sn,s})$
$(h_s(f_A(a1,\ldots,an))=f_B(h_{s1}(a1),\ldots,h_{sn}(an)))$

b) H heißt *Σ-Epimorphismus* :gdw H ist Σ-Homomorphismus und $(\forall s\in S)$ h_s ist surjektiv.

c) H heißt *Σ-Monomorphismus* :gdw H ist Σ-Homomorphismus und $(\forall s\in S)$ h_s ist injektiv.

d) H heißt *Σ-Isomorphismus* :gdw H ist Σ-Epi- und Σ-Monomorphismus.
Notation: $A \simeq B$.

1.1.1.7 Definition / Corollar (Komposition)

Sei (S,Σ) eine Signatur, A und B Σ-Algebren und $F: A \rightarrow B$, $H: B \longrightarrow C$ Σ-Homomorphismen.
Dann ist $H\circ F:=\{h_s\circ f_s: A_s \longrightarrow C_s,\ s\in S\}: A \longrightarrow C$ ein Σ-Homomorphismus.
Die Komposition ist assoziativ.

Beweis: [HOR 79].

1.1.1.8 Corollar

Sei (S,Σ) eine Signatur, A und B Σ-Algebren, $H:A \longrightarrow B$ ein Σ-Homomorphismus.
Dann gilt: H ist Σ-Isomorphismus $\Longleftrightarrow (\exists\overline{H}:B \rightarrow A)(H\circ\overline{H}=id_B \wedge \overline{H}\circ H=id_A)$.

Beweis:[HOR 79].

Oft ist es hilfreich, alle Algebren, die eine bestimmte Eigenschaft haben, sowie alle Homomorphismen zwischen ihnen zusammenzufassen zu einer Gesamtheit. Hierzu bietet sich der Kategorienbegriff an. Für die folgenden Definitionen vergleiche [ADJ 75].

1.1.1.9 Definition (Kategorie)

$C:=(|C|,/C/,Z,Q,\circ)$ ist *Kategorie* :gdw

1. $|C|$ und $/C/$ sind Klassen, genannt die *Objekte* bzw. *Morphismen* von C.
2. Z, Q: $/C/ \longrightarrow |C|$ sind Funktionen, genannt *Ziel* bzw. *Quelle*.

3. $\circ$:/C/x/C/ $\longrightarrow$/C/ ist eine partielle Funktion, *Komposition* genannt.
   Für $\circ(f,g)$ wird auch $f\circ g$ geschrieben.
   $(\forall f,g,h \in /C/)$ gilt
   - $f\circ g$ definiert $\Leftrightarrow$ $Z(g)=Q(f)$
   - Falls $f\circ g$ definiert, so gilt $Q(f\circ g)=Q(g)$ und $Z(f\circ g)= Z(f)$.
   - Falls $f\circ g$ und $g\circ h$ definiert, so gilt
     $(f\circ g)\circ h$, $f\circ(g\circ h)$ definiert und $(f\circ g)\circ h=f\circ(g\circ h)$.
4. $(\forall a \in |C|)(\exists id_a \in /C/)$ so daß
   - $Q(id_a)=Z(id_a)=a$
   - $(\forall f,g \in /C/: Z(f)=Q(g)=a)$ ($id_a\circ f=f \wedge g\circ id_a=g$)

1.1.1.10 Lemma

Sei (S,Σ) eine Signatur, $A:=\{A_i: i\in I\}$ eine Familie von Σ-Algebren und $H:=\{h:A_i \longrightarrow A_j,\ i,j\in I,\ h\ \Sigma\text{-Homomorphismus}\}$ die Menge aller Σ-Homomorphismen zwischen Algebren aus A. Sei $\circ$ die Komposition von Σ-Homomorphismen entsprechend 1.1.1.7. Seien $Z,Q: H \to A$, so daß

$$(\forall f\in H)(Z(f)=A_i \Leftrightarrow (\exists j\in I)(f:A_j \to A_i))$$
$$(\forall f\in H)(Q(f)=A_i \Leftrightarrow (\exists j\in I)(f:A_i \to A_j)).$$

Dann ist $(A,H,Z,Q,\circ)$ eine Kategorie.

Beweis:

Bedingung 3 in 1.1.1.9 folgt unmittelbar aus den Eigenschaften der Hintereinanderausführung $\circ$ von Σ-Homomorphismen.
In Bedingung 4 in 1.1.1.9 setze für $A_i \in A$, $i\in I$: id_{A_i} := Identität auf A_i. id_{A_i} ist Σ-Homomorphismus, so daß $id_{A_i} \in H$. Die Forderungen in 4. gelten dann trivialerweise.

MMM

Der Einfachheit halber wird im folgenden bei der Definition einer Kategorie von Σ-Algebren und Σ-Homomorphismen nur A angegeben und Z, Q, H und $\circ$ wie in 1.1.1.10 vorausgesetzt.

Nun können alle Σ-Algebren zu einer Kategorie zusammengefaßt werden.

1.1.1.11 Definition (Alg_Σ)

Sei (S,Σ) eine Signatur.

Dann ist Alg_Σ die Kategorie, deren Objekte die Σ-Algebren und deren Morphismen die Σ-Homomorphismen sind.

Strukturerhaltende Abbildungen zwischen Kategorien sind Funktoren:

1.1.1.12 Definition (Funktor)

Seien A und B Kategorien.

Ein Quadrupel $F:=(A,|F|,/F/,B)$ heißt *Funktor* :gdw

$|F|: |A| \longrightarrow |B|$, $/F/: /A/ \longrightarrow /B/$ sind Funktionen, für die gilt

1. Das Diagramm

$$\begin{array}{ccc} /A/ & \xrightarrow{Z,Q} & |A| \\ {\scriptstyle /F/}\downarrow & & \downarrow{\scriptstyle |F|} \\ /B/ & \xrightarrow[Z,Q]{} & |B| \end{array}$$

kommutiert.

2. $(\forall f,g \in /A/)(f\circ g \text{ definiert} \Longrightarrow (/F/(f)\circ/F/(g) \text{ definiert und } /F/(f)\circ/F/(g)=/F/(f\circ g))).$

3. $(\forall a \in |A|)\ (/F/(id_a)=id_{|F|(a)})$.

1.1.1.13 Definition (persistent)

Seien (S,Σ) und (S',Σ') Signaturen mit $S \subseteq S'$ und $\Sigma \subseteq \Sigma'$.

Sei A eine Kategorie von Σ-Algebren und B eine Kategorie von Σ'-Algebren.

$F: A \longrightarrow B$ sei ein Funktor.

F heißt *persistent* :gdw $(\forall a \in A)\ (a \simeq f(a)|_\Sigma)$.

Persistente Funktoren betten also ihre Argumente isomorph in deren Bilder ein.

Schließlich können Funktoren zu einer Kategorie zusammengefaßt werden, deren Morphismen die natürlichen Transformationen sind:

1.1.1.14 Definition (natürliche Transformation)

Seien C und D Kategorien und F,G: C → D Funktoren.

τ: F —>G ist eine *natürliche Transformation* :gdw

$\tau=\{\tau_c: |F|(c) \longrightarrow |G|(c), c\in C\}$ mit

$(\forall c,c'\in C)(\forall f: c \longrightarrow c' \in C) (\tau_{c'} \circ /F/(f) = /G/(f) \circ \tau_c)$ d.h.

das Diagramm

$$\begin{array}{ccc} c & & |F|(c) \xrightarrow{\tau_c} |G|(c) \\ f \downarrow & /F/(f) \downarrow & \downarrow /G/(f) \\ c' & & |F|(c') \xrightarrow[\tau_{c'}]{} |G|(c') \end{array}$$

kommutiert.

Durch die Zusammenfassung von Algebren mit bestimmten Eigenschaften zu einer Kategorie können einige dieser Algebren durch ihre Stellung in dieser Kategorie ausgezeichnet werden.

1.1.1.15 Definition (initiales und terminales Objekt)

Sei C eine Kategorie mit der Objektmenge O und der Morphismenmenge M.

$o\in O$ heißt *initial* :gdw $(\forall c\in O)(\exists_1 f\in M)(f: o \longrightarrow c)$

$o\in O$ heißt *terminal* :gdw $(\forall c\in O)(\exists_1 f\in M)(f: c \longrightarrow o)$

Der folgende Satz zeichnet die Termalgebra T_Σ aus in der Kategorie der Σ-Algebren.

1.1.1.16 Satz / Definition (Φ_A)

Sei (S,Σ) eine Signatur.

Dann ist T_Σ das bis auf Isomorphie eindeutige initiale Objekt in Alg_Σ.

Dabei wird für $A\in Alg_\Sigma$ der eindeutige Σ-Homomorphismus $\Phi_A: T_\Sigma \longrightarrow A$ bestimmt durch:

1. $(\forall s\in S)(\forall f\in \Sigma_{\varepsilon,s})(\Phi_{A,s}(f\ ) := f_A)$
2. $(\forall n\in N)(\forall s1,\ldots sn,s\in S)(\forall f\in \Sigma_{s1\ldots sn,s})(\forall (t1,\ldots,tn)\in T_{\Sigma,s1}\times\ldots\times T_{\Sigma,sn})$
   $(\Phi_{A,s}(f\ (t1,\ldots,tn)) := f_A(\Phi_{A,s1}(t1),\ldots,\Phi_{A,sn}(tn))\ )$

Beweis: [ADJ 77].

1.1.1.17 Definition (Σ-Kongruenz)

Seien (S,Σ) eine Signatur und A eine Σ-Algebra.

$\equiv=\{\equiv_s : s\in S\}$ ist Σ-*Kongruenz* auf A :gdw

1. $(\forall s\in S)$ $\equiv_s$ ist Äquivalenzrelation auf A_s
2. $\equiv$ hat die *Kongruenzeigenschaft*, d.h.

$(\forall n\in N)(\forall s,s1,\ldots,sn \in S)(\forall f\in\Sigma_{s1\ldots sn,s})$
$(\forall(a1,\ldots,an),(b1,\ldots,bn)\in A_{s1}x\ldots xA_{sn})$
$((ai\equiv_{si} bi,\ i=1\ldots n) \Longrightarrow (f_A(a1,\ldots,an) \equiv_s f_A(b1,\ldots,bn)))$

Auch für Σ-Kongruenzen gilt, daß der Durchschnitt von Σ-Kongruenzen wieder eine Σ-Kongruenz ist:

1.1.1.18 Lemma

Seien (S,Σ)eine Signatur, A eine Σ-Algebra und $\{\equiv_i : i\in I\}$ eine Menge von Σ-Kongruenzen auf A.

Dann ist $\equiv := \bigcap_{i\in I} \equiv_i := \{\equiv_s := \bigcap_{i\in I} \equiv_{i,s} : s\in S\}$ Σ-Kongruenz auf A.

Beweis: [HOR 79].

Von Interesse sind nun Algebren, die man durch Faktorisierung einer Termalgebra nach einer Kongruenz erhält.

1.1.1.19 Definition (Quotientenalgebra)

Seien (S,Σ) eine Signatur, A eine Σ-Algebra und $\equiv$ eine Σ-Kongruenz auf A.

Dann heißt $A/_\equiv := (\{A_s/_\equiv : s\in S\},\{f_{A/_\equiv} : f\in\Sigma_{w,s}, w\in S^*, s\in S\})$ die *Quotientenalgebra* von A über $\equiv$.

Dabei gilt:
1. $(\forall s\in S)(A_s/_\equiv := \{[a]_{\equiv s} : a\in A_s\})$
2. $(\forall n\in N)(\forall s,s1,\ldots,sn\in S)(\forall f\in\Sigma_{s1\ldots sn,s})$
$(\forall(a1,..,an)\in A_{s1}x..xA_{sn})$
$(f_{A/\equiv}([a1]_{\equiv s1},..,[an]_{\equiv sn}) := [f_A(a1,\ldots,an)]_{\equiv s})$

Da $\equiv$ eine Σ-Kongruenz auf A ist, sind 1. und 2. unabhängig von der Wahl der Repräsentanten, d.h. die Quotientenalgebra ist wohldefiniert.

Da T_Σ initial ist in Alg_Σ, induziert jede Σ-Algebra über den initialen Σ-Homomorphismus eine Σ-Kongruenz auf T_Σ.

1.1.1.20 Definition / Lemma ($\equiv_A$)

Seien (S,Σ) eine Signatur und A eine Σ-Algebra.

Dann bezeichnet $\equiv_A := \{\equiv_{A,s} : s\in S\}$ die Relationenfamilie auf T_Σ, die folgendermaßen durch Φ_A: $T_\Sigma \longrightarrow A$ (vgl. 1.1.1.16) induziert wird:

$(\forall s\in S)(\forall t,t'\in T_{\Sigma,s})(t\equiv_{A,s} t'$:gdw $\Phi_{A,s}(t)=\Phi_{A,s}(t'))$.

$\equiv_A$ ist eine Σ-Kongruenz auf A.

Beweis: [HOR 79].

1.1.1.21 Definition (Σ-erzeugt)

Seien (S,Σ) eine Signatur und A eine Σ-Algebra. Φ_A: $T_\Sigma \longrightarrow A$ gemäß 1.1.1.16.
A ist *Σ-erzeugt* :gdw Φ_A ist surjektiv.

Nun werden in eine Termalgebra Terme mit Variablen eingeführt.

1.1.1.22 Definition (Signatur mit Variablen, Belegung)

Sei (S,Σ) eine Signatur.

1. $(\forall s\in S)$ sei X_s eine unendliche , abzählbare Menge von *Variablensymbolen* der Sorte s, mit
   - $(\forall s'\in S)\ (s\neq s' \Longrightarrow X_s\cap X_{s'}=\emptyset)$
   - $(\forall s\in S)(\forall w\in S^*)(\forall s'\in S)\ (X_{s'}\cap\Sigma_{w,s}=\emptyset)$
2. $(S,\Sigma(X))$ ist eine *Signatur mit Variablen* :gdw
   $\Sigma(X):=\{\Sigma(X)_{w,s}:w\in S^*,s\in S\}$ mit
   - $(\forall s\in S)(\Sigma(X)_{\varepsilon,s}:=\Sigma_{\varepsilon,s}\cup X_s)$
   - $(\forall s\in S)\ (\forall w\in S^+)(\Sigma(X)_{w,s}:=\Sigma_{w,s})$
3. Die $\Sigma(X)$-Termalgebra $T_{\Sigma(X)}$ heißt Σ-Algebra $T_\Sigma(X)$ oder *freie Σ-Algebra über X* :gdw
   - $(\forall s\in S)(T_\Sigma(X)_s:=T_{\Sigma(X),s})$
   - $(\forall s\in S)(\forall w\in S^*)(\forall f\in\Sigma_{w,s})(f_{T_\Sigma(X)}:=f_{T_{\Sigma(X)}})$
4. Für eine Σ-Algebra A heißt eine Familie $a=(a_s)_{s\in S}$ mit a_s: $X_s \longrightarrow A_s$ eine Belegung der Variablen aus X in A. Notation: a: $X \longrightarrow A$.

Eine Belegung a kann in eindeutiger Weise fortgesetzt werden zu einem Σ-Homomorphismus $T_\Sigma(X) \longrightarrow A$:

1.1.1.23 Satz

Sei $(S,\Sigma(X))$ eine Signatur mit Variablen.
Dann gibt es zu jeder Σ-Algebra A und zu jeder Belegung $a:X \longrightarrow A$ genau einen Σ-Homomorphismus $H^*:T_\Sigma(X) \longrightarrow A$ mit

$$(\forall s\in S)(\forall x\in X_s)(H^*_s(x)=a_s(x))$$

Beweis: [RAU 79].

1.1.2 Spezifikationen

In diesem Abschnitt werden Signaturen um Gleichungen erweitert, die Terme miteinander identifizieren.

1.1.2.1 Definition (Gleichung)

Sei $(S,\Sigma(X))$ eine Signatur mit Variablen.
Für jede Sorte $s\in S$ heißt ein geordnetes Paar $e=(L,R)$ von Σ-Termen der Sorte s mit Variablen in X eine *Σ-Gleichung* der Sorte s.

1.1.2.2 Definition (Spezifikation)

Sei $(S,\Sigma(X))$ eine Signatur mit Variablen und $E=\{E_s : s\in S\}$ eine Menge von Gleichungen der Sorten $s\in S$.
Dann heißt (S,Σ,E) eine *Spezifikation*.

1.1.2.3 Defintion (Erfüllen einer Gleichung, (Σ,E)-Algebra)

Sei $(S,\Sigma(X))$ eine Signatur mit Variablen, $E=\{E_s : s\in S\}$ eine Menge von Gleichungen und $SP:=(S,\Sigma,E)$ die nach 1.1.2.2 zugehörige Spezifikation.

1. Eine Σ-Algebra A *erfüllt die Gleichung* $e=(L,R)\in E_s\in E$:gdw
   Für alle Belegungen $a: X \longrightarrow A$ gilt: $\hat{a}(R) = \hat{a}(L)$;
   Dabei bezeichnet $\hat{a}: T_\Sigma(X) \longrightarrow A$ den nach 1.1.1.23 eindeutig von a erzeugten Σ-Homomorphismus.
2. Eine Σ-Algebra A *erfüllt die Spezifikation SP* :gdw
   jede Gleichung $e\in E$ wird von A erfüllt.
   A heißt dann *(Σ,E)-Algebra*.

Nun werden wieder alle (Σ,E)-Algebren zu einer Kategorie zusammengefaßt:

1.1.2.4 Definition ($Alg_{\Sigma,E}$)
Sei (S,Σ,E) eine Spezifikation.
Dann ist $Alg_{\Sigma,E}$ die Kategorie, deren Objekte die (Σ,E)-Algebren und deren Morphismen die Σ-Homomorphismen unter diesen Algebren sind.

1.1.2.5 Definition/Lemma (die von einer Relation erzeugte Σ-Kongruenz)

Seien (S,Σ) eine Signatur, A eine Σ-Algebra und $R:=\{R_s\subseteq A_s\times A_s : s\in S\}$ eine Familie von Relationen auf A.
Dann gibt es eine kleinste Σ-Kongruenz auf A, die R enthält, nämlich der Durchschnitt aller Σ-Kongruenzen $\equiv$ auf A mit $R\subseteq\equiv$
Diese Σ-Kongruenz heißt *die von R erzeugte Σ-Kongruenz*.
Beweis: [RAU 79].

1.1.2.6 Definition ($\equiv_E$, Quotiententermalgebra)

Sei (S,Σ,E) eine Spezifikation.

Sei $\equiv_E$ die nach 1.1.2.5 von der Relation $B_E := \{(\hat{a}_s(L),\hat{a}_s(R)): (L,R)\in E_s \in E, s\in S,$ a ist Belegung in T_Σ und $\hat{a}$ deren eindeutige Fortsetzung nach 1.1.1.23$\}$ eindeutig erzeugte Σ-Kongruenz auf T_Σ.

Notation: Sei $R \subseteq T_\Sigma \times T_\Sigma$. Dann wird für '$B_E \subseteq R$' auch kurz '$E \subseteq R$' geschrieben.

$T_{\Sigma,E} := T_\Sigma / \equiv_E$ heißt die *Quotiententermalgebra* der Spezifikation (S,Σ,E).

1.1.2.7 Lemma

Seien (S,Σ,E) ,(S',Σ',E') zwei Spezifikationen mit $S \subseteq S'$, $\Sigma \subseteq \Sigma'$ und $E \subseteq E'$.

Dann gilt: $(\forall s\in S)(\equiv_{E,s} \subseteq \equiv_{E',s})$.

Notation: $\equiv_E \subseteq \equiv_{E'}$.

Beweis: [HOR 79].

1.1.2.8 Lemma

Sei (S,Σ,E) eine Spezifikation und A eine (Σ,E)-Algebra.

Dann gilt: $(\forall s\in S)(\equiv_{E,s} \subseteq \equiv_{A,s})$.

Notation: $\equiv_E \subseteq \equiv_A$.

Beweis:[HOR 79].

1.2 Abstrakte Datentypen in initialer Algebrasemantik

Eine Entsprechung zu 1.1.1.16 ist für $Alg_{\Sigma,E}$ der folgende Satz, der die Quotiententermalgebra in $Alg_{\Sigma,E}$ auszeichnet.

1.2.1 Satz

Sei (S,Σ,E) eine Spezifikation.

Dann ist $T_{\Sigma,E}$ initial in $Alg_{\Sigma,E}$.

Beweis:[HOR 79].

1.2.2 Corollar

Sei (S,Σ,E) eine Spezifikation.

Dann ist jedes initiale Objekt in $Alg_{\Sigma,E}$ Σ-isomorph zu $T_{\Sigma,E}$.

Beweis: [ADJ 77].

Beim initialen Ansatz bei der Behandlung abstrakter Datentypen wird diese Isomorphieklasse als Semantik eines abstrakten Datentyps gewählt, wobei $T_{\Sigma,E}$ als Repräsentant dieser Klasse verwendet wird. Zwei Terme aus $T_{\Sigma,E}$ bezeichnen dann dasselbe Objekt des i-abstrakten Datentyps, wenn sie durch E über die Kongruenz $\equiv_E$ identifiziert werden.

1.2.3 Definition (i-abstrakter Datentyp)

Sei (S,Σ,E) eine Spezifikation.
Dann heißt $T_{\Sigma,E}$ (bzw. die Σ-Isomorphieklasse von $T_{\Sigma,E}$ in $Alg_{\Sigma,E}$) der durch (S,Σ,E) definierte *i-abstrakte Datentyp*.
Notation: "i" steht für "initial". Analog wird weiter unten das Präfix "t" für "terminal" verwendet.

1.3 Abstrakte Datentypen in terminaler Algebrasemantik

In der initialen Semantik werden zwei Terme identifiziert, wenn ihre Gleichheit aus den Gleichungen einer Spezifikation ableitbar ist (Semi-Thue-Ableitung); andernfalls sind sie verschieden. Dennoch kann es Terme geben, die nicht gleich sind in der initialen Semantik, von ihrer Wirkung her aber nicht unterscheidbar sind. Dabei wird diese Wirkung bezogen auf eine ausgezeichnete Sorte "dis" (distinguished), deren zugehörige Trägermenge zwei Elemente enthält, die per definitionem verschieden sind. Insbesondere wird man im folgenden von jeder Spezifikation erwarten, daß sie diese Sorte "dis" in ihrer Signatur enthält.

1.3.1 Defintion (t-Spezifikation, konsistent, vollständig)

1. Eine Spezifikation (S,Σ,E) heißt *t-Spezifikation* :gdw
   - $dis \in S$
   - $\{tr,fl\} \subseteq \Sigma_{\varepsilon,dis}$
2. Eine t-Spezifikation (S,Σ,E) heißt *konsistent* :gdw
   $tr \neq_E fl$
3. Eine t-Spezifikation (S,Σ,E) heißt *vollständig* :gdw
   $(\forall t \in T_{\Sigma,dis})(\exists tv \in \{tr,fl\})\ (t \equiv_E tv)$

tr und fl stehen für die Wahrheitswerte "true" und "false".
Die Konsistenz einer t-Spezifikation fordert gerade die Existenz zweier verschiedener Elemente, die im weiteren als Bezugspunkt für die Verhaltensgleichheit dienen werden.
Die Vollständigkeit einer t-Spezifikation besagt, daß jeder Term der Sorte dis zu einem der Wahrheitswerte reduziert werden kann vermöge der Gleichungen E.
Eine konsistente und vollständige Spezifikation stellt somit sicher, daß $T_{\Sigma,E,dis}$ genau zwei Elemente enthält, nämlich $[tr]_{\equiv E}$ und $[fl]_{\equiv E}$.

Als nächstes wird die Σ-Kongruenz auf T_Σ definiert, die zwei Terme dann miteinander identifiziert, wenn sie bzgl. dis sich gleich verhalten. Zwei Terme gelten nur dann als nicht verhaltensgleich, wenn sie sich bzgl. dis durch E voneinander unterscheiden lassen.

1.3.2 Definition ($\sim_E$)

Sei (S,Σ,E) eine t-Spezifikation.

$\sim_E := \{\sim_{E,s} : s \in S\}$ ist eine Relationenfamilie auf T_Σ, die für $s \in S$ wie folgt induktiv definiert wird:

1. $(\forall i \in N_0)$ $\overset{i}{\sim}_{E,dis} := \equiv_{E,dis}$
2. $(\forall s \in S-\{dis\})(\forall t,t' \in T_{\Sigma,s})$

   $t \overset{0}{\sim}_{E,s} t'$:gdw $(\forall n \in N)(\forall s1,\ldots,sn \in S)(\forall f \in \Sigma_{s1\ldots sn,dis})$
   $(\forall 1 \leq j \leq n)(\forall (t1,\ldots,tn) \in T_{\Sigma,s1} x \ldots x T_{\Sigma,sn})$
   $(sj=s \implies f(t1,\ldots,tj-1,t,tj+1,\ldots,tn) \sim_{E,dis} f(t1,\ldots,tj-1,t',tj+1,\ldots,tn))$
3. $(\forall i \in N_0)$
   $(\forall s \in S-\{dis\})(\forall t,t' \in T_{\Sigma,s})$

   $t \overset{i+1}{\sim}_{E,s} t'$:gdw $(\forall n \in N)(\forall s1,\ldots sn,s' \in S)(\forall f \in \Sigma_{s1\ldots sn,s'})$
   $(\forall 1 \leq j \leq n)(\forall (t1,\ldots,tn) \in T_{\Sigma,s1} x \ldots x T_{\Sigma,sn})$
   $(sj=s \implies f(t1,\ldots,tj-1,t,tj+1,\ldots,tn) \overset{i}{\sim}_{E,s'} f(t1,\ldots,tj-1,t',tj+1,\ldots,tn))$
4. $(\forall s \in S)(\sim_{E,s} := \bigcap_{i \in N_0} \overset{i}{\sim}_{E,s})$

1.3.3 Lemma

Sei (S,Σ,E) eine t-Spezifikation, $\sim_E$ wie in 1.3.2.
Dann ist $\sim_E$ eine Σ-Kongruenz.

Beweis: [HOR 79].

1.3.4 Lemma

Sei (S,Σ,E) eine t-Spezifikation.
Dann gilt: $(\forall s \in S)$ $(\equiv_{E,s} \subseteq \sim_{E,s})$.
Notation: $\equiv_E \subseteq \sim_E$.

Beweis: [HOR 79].

In $\equiv_E$ werden möglichst wenig Terme identifiziert unter der Randbedingung, daß die Gleichungen aus E noch gelten. $\equiv_E$ ist damit die Kongruenz auf T_Σ, die gerade noch E enthält.

Falls (S,Σ,E) eine konsistente und vollständige t-Spezifikation ist, ist $\sim_E$ die Σ-Kongruenz auf T_Σ, in der zwei Terme derselben Sorte verschieden sind, weil ihre Gleichsetzung zusammen mit den Gleichungen aus E "tr = fl" liefern würde.
Das folgende Lemma ist eine Entsprechung zu 1.1.2.7.

1.3.5 Lemma

Seien (S,Σ,E) und (S',Σ',E') vollständige und konsistente t-Spezifikationen mit $S \subseteq S'$, $\Sigma \subseteq \Sigma'$ und $E \subseteq E'$.
Dann gilt: $(\forall s \in S)(\forall t,t' \in T_{\Sigma,s})(t \sim_E t' \implies t \sim_E t')$.
Notation: $\sim_E \subseteq \sim_E$.

Beweis: [HOR 79].

Die Umkehrung der Inklusionsrichtung im Vergleich zu 1.1.2.7 rührt daher, daß zwei Terme, die bzgl. der ursprünglichen Menge von Gleichungen unterschiedlich sind, dies auch bzgl. der erweiterten Gleichungsmenge sind.

1.3.6 Definition $(T_{\Sigma,\sim_E})$

Sei (S,Σ,E) eine t-Spezifikation.
Dann ist $T_{\Sigma,\sim_E} := T_\Sigma/\sim_E$.

$T_{\Sigma,\sim_E}$ ist wieder eine Σ-Algebra und erfüllt die Gleichungen in E:

1.3.7 Lemma

Sei (S,Σ,E) eine t-Spezifikation.
Dann ist $T_{\Sigma,\sim_E}$ eine (Σ,E)-Algebra.

Beweis: [HOR 79].

Analog zu 1.2.1 wird man nun erwarten, daß $T_{\Sigma,\sim_E}$ terminal ist in der Kategorie aller (Σ,E)-Algebren. Allerdings gilt nur eine schwächere Behauptung, die nur bestimmte (Σ,E)-Algebren zuläßt.

1.3.8 Definition (t-Mod(Σ,E), t-Imp(Σ,E))

Sei (S,Σ,E) eine vollständige und konsistente t-Spezifikation.

1. *t-Mod(Σ,E)* ist die Kategorie, deren Objekte die (Σ,E)-Algebren A sind, für die gilt:

$$A_{dis}=\{tr_A,fl_A\} \wedge tr_A \neq fl_A \quad \text{und A ist } \Sigma\text{-erzeugt.}$$

2. *t-Imp(Σ,E)* ist die Kategorie, deren Objekte die Σ-Algebren sind, für die gilt:
   - $A_{dis}=\{tr_A,fl_A\} \wedge tr_A \neq fl_A$
   - $(\forall s \in S)(\forall t,t' \in T_{\Sigma,s})(t \equiv_A t' \Longrightarrow t \sim_E t')$
   - A ist Σ-erzeugt.

Die Algebren aus t-Mod(Σ,E) erfüllen also die Axiome und modellieren $T_{\Sigma,E,dis}$ durch $\{tr_A,fl_A\}$.
Die Algebren aus t-Imp(Σ,E) brauchen die Axiome aus E nicht zu erfüllen. Zwei Terme, die vermöge des initialen Homomorphismus Φ_A demselben Objekt einer solchen Algebra entsprechen, müssen aber unter $\sim_E$ kongruent sein. Die Beziehung zwischen t-Mod(Σ,E) und t-Imp(Σ,E) wird beschrieben durch:

1.3.9 Corollar

Sei (S,Σ,E) eine vollständige und konsistente t-Spezifikation.
Dann gilt: $Ae\,|\text{t-Mod}(\Sigma,E)| \Longrightarrow Ae\,|\text{t-Imp}(\Sigma,E)|$.

Beweis: [HOR 79].

Die Quotientenalgebra $T_{\Sigma,\sim_E}$ ist nun in beiden Kategorien t-Mod(Σ,E) und t-Imp(Σ,E) terminales Objekt:

1.3.10 Satz

Sei (S,Σ,E) eine vollständige und konsistente t-Spezifikation.
Dann gilt:

$T_{\Sigma,\sim_E}$ ist terminal in t-Imp(Σ,E) und in t-Mod(Σ,E).

Beweis: [HOR 79].

In dualer Fassung zu 1.2.3 definiert die folgende Definition $T_{\Sigma,\sim_E}$ bzw. die Isomorphieklasse von $T_{\Sigma,\sim_E}$ als terminale Semantik eines abstrakten Datentyps.

1.3.11 Definition (t-abstrakter Datentyp, t-Modell, t-Implementation)

Sei SP:=(S,Σ,E) eine vollständige und konsistente t-Spezifikation.

1. $T_{\Sigma,\sim_E}$ bzw, die Σ-Isomorphieklasse von $T_{\Sigma,\sim_E}$ heißt der durch SP definierte *t-abstrakte Datentyp*.
2. $Ae\,|\text{t-Imp}(\Sigma,E)|$ heißt *t-Implementation* von $T_{\Sigma,\sim_E}$.
3. $Ae\,|\text{t-Mod}(\Sigma,E)|$ heißt *t-Modell* von $T_{\Sigma,\sim_E}$.

Zwei Objekte des t-abstrakten Datentyps sind also dann verschieden, wenn sie sich bzgl. dieses durch die Gleichungen E voneinander unterscheiden.

Dies ist die duale Sichtweise zur initialen Algebrasemantik, wo zwei Objekte des i-abstrakten Datentyps dann gleich sind, wenn sich ihre Gleichheit aus E herleiten läßt.
In der Praxis ist für einen Programmierer das Verhalten eines Exemplars eines Datentyps von Interesse, das ihm in der Regel über eine Modulschnittstelle mitgeteilt wird. Uninteressant sind für ihn Datenstrukturen, die zwar auf unterschiedliche Art erzeugt oder realisiert werden, in ihrer Wirkung nach außen aber nicht unterscheidbar sind. Da in dieser Arbeit von der Programmierpraxis ausgegangen wird, wird im weiteren Verlauf einem Datentyp stets eine terminale Semantik unterstellt.

Ein t-Modell A von $T_{\Sigma,\sim E}$ mit dem terminalen Σ-Homomorphismus $G: A \longrightarrow T_{\Sigma,\sim E}$ kann zwar für $s \in S$ ein Element $t \in T_{\Sigma,\sim E,s}$ mit zwei verschiedenen Elementen $a,a' \in A_s$ modellieren (d.h. $a \neq a' \wedge G(a)=t=G(a')$), erfüllt aber die Gleichungen aus E.

Für eine t-Implementation B von $T_{\Sigma,\sim E}$ mit dem terminalen Σ-Homomorphismus $H: B \longrightarrow T_{\Sigma,\sim E}$ und dem initialen Σ-Homomorphismus $F: T_\Sigma \longrightarrow B$ gelten die Gleichungen aus E i.a. nicht mehr, d.h. es kann für $s \in S$ $t,t' \in T_{\Sigma,s}$ geben mit $t \equiv_E t'$, aber $F(t) \neq F(t')$.

In Kapitel 4 wird bei der Behandlung parametrisierter Datentypen noch der Begriff der Erweiterung gebraucht. Man wird nämlich erwarten, daß die Algebra eines aktuellen Parameters in die Ergebnisalgebra des aktuell parametrisierten Datentyps eingebettet werden kann. Hierzu dient die folgende Definition.

1.3.12 Definiton (i-Erweiterung)

Seien $SP=(S,\Sigma,E)$ und $SP'=(S',\Sigma',E')$ Spezifikationen.

SP' ist *i-Erweiterung* von SP :gdw

1. $S \subseteq S'$
2. $\Sigma \subseteq \Sigma'$
3. $E \subseteq E'$
4. $T_{\Sigma,E}$ ist Σ-isomorph zu $T_{\Sigma',E'}|_\Sigma$.

Wenn SP' eine i-Erweiterung von SP ist, heißt auch $T_{\Sigma',E'}$ i-Erweiterung von $T_{\Sigma,E}$.

1.3.13 Definition (t-Erweiterung)

Seien $SP:=(S,\Sigma,E)$ und $SP':=(S',\Sigma',E')$ vollständige und konsistente t-Spezifikationen.

SP' ist *t-Erweiterung* von SP :gdw

1. $S \subseteq S'$
2. $\Sigma \subseteq \Sigma'$
3. $E \subseteq E'$
4. $T_{\Sigma,\sim_E}$ ist Σ-isomorph zu $T_{\Sigma',\sim_{E'}}|_{\Sigma}$.

Wenn SP' eine t-Erweiterung von SP ist, heißt auch $T_{\Sigma',\sim_{E'}}$ t-Erweiterung von $T_{\Sigma,\sim_E}$.

2. DIE BEISPIELSPRACHEN: CSSA, CLU UND ALPHARD

Die drei hier zu beschreibenden Sprachen wurden u.a. mit dem Ziel entworfen, durch Abstraktionsmechanismen die Programme und den Programmentwurf zu strukturieren.
CLU und ALPHARD sind SIMULA-Nachfolger. Die Konzepte zur Datenabstraktion - Cluster und Forms - wurden aus den SIMULA-Klassen heraus entwickelt. Dazuhin bietet CLU mit dem Iterator-Konzept die Möglichkeit zur Abstraktion einfacher Kontrollstrukturen (Wiederholungen). Da beide Sprachen in den wesentlichen Konzepten übereinstimmen, werden sie gemeinsam durch die fiktive Mischsprache CLALPHARD beschrieben.

CSSA ist eine interaktive Programmiersprache für asynchrone Multiprozessorsysteme. Konsequent wurden bei diesem Sprachentwurf Konzepte für Operations-, Daten- und Kontrollabstraktion eingeführt und heterarchische Kontrollstrukturen vorgesehen.
Für CSSA und CLU bzw. ALPHARD wird im folgenden eine abgemagerte abstrakte Syntax ([RAU 78]) und eine abgemagerte denotationale Semantik ([STOY], [RAU 78]) angegeben, die sich auf die für diese Arbeit relevanten Punkte beschränken.

Notation:

Die abstrakte Syntax wird beschrieben durch eine Menge von Prädikaten und Mengen. Ein Prädikat wird definiert durch ein oder mehrere n-Tupel, deren Komponenten durch Selektoren identifiziert werden. Die Anwendung eines Selektors auf ein n-Tupel wird in Punkt-Notation formuliert;
z.B.: ls.assign.
Bei der Definition semantischer Funktionen wird die λ-Notation verwendet ([STOY], S.121).
"If P then E1 else E2" wird abgekürzt durch "P $\longrightarrow$E1 | E2".
Anstelle von "If P then E" wird geschrieben "case P : E".
"$\perp$" bezeichnet abhängig vom Kontext ein Fehlerelement, die leere Liste oder eine leere Komponente in einem n-Tupel.

Auf die Komponenten einer Liste wird durch Indizierung des Listenbezeichners zugegriffen.

2.1 CSSA

(vgl. [CSSA 77],[CSSA 79]).

Grundlegende Objekte einer CSSA-Berechnung sind Moduln, die als Agenten bezeichnet werden. Diese werden dynamisch durch Generatoren auf Grund einer Moduldefinition, einem Script, erzeugt. Der Typ eines Agenten ist das Script, aus dem er erzeugt wurde.
Agenten können über Zugriffs- und Kommunikationswege, den Acquaintances (Bekanntschaften), zu einer Netzstruktur verknüpft werden. Der Kontrollfluß innerhalb dieses Netzes und die Kommunikation zwischen den Agenten werden durch den Austausch von Nachrichten, den Messages, gewährleistet. Über die Aktivierung eines Agenten durch eine Nachricht entscheidet der Ausgang eines Entry-Match der Nachricht gegen ein Entry-Pattern.

Das Sprachkonzept für die Datenabstraktion ist das Script.
Im Gegensatz zu den Sprachen der SIMULA-Generation werden bei der Realisierung der Datenabstraktion durch Scripts bzw. Agenten keine spracheigenen Datenstrukturen für eine interne Darstellung benötigt. Vielmehr wird etwa jede Komponente eines Exemplars eines zusammengesetzten Datentyps durch einen Agenten repräsentiert, der u.U.implizit erzeugt wird.
Beispiel: Push(s,x) kann realisiert werden durch eine implizite Erzeugung eines Moduls durch den mit s bezeichneten Modul, der einen Verweis auf das Datum 'x' enthält.

2.1.1. Abgemagerte abstrakte Syntax für CSSA

Expr = VExpr v Genagent
Genagent = (<sc: Script>,<init: Expr*>,<access: Opid+>)
Opid $\subseteq$ Id
Instr = Assign v Activate
Assign = (<ls: Id>,<rs: Expr>)
Aktivate = (<mess: (<ms: Expr*>,<opid: Opid>),<target: Id>)
Script = (<create: Id*>,<procblock:
{<id: (<entry: Id*>,<lcode: Instr*x Instr>)>||id $\in$ OI$\subseteq$Opid}>)

- VExpr sind arithmetische und logische Ausdrücke, die hier nicht näher interessieren.
- Opid ist die Menge der Operationsbezeichner.

- Genagent erzeugt einen Agenten unter Angabe
  - des zugehörigen Scripts
  - der Initialisierungsnachricht (Paramterliste)
  - der Operationen, auf die von außen zugegriffen werden darf.
- Von Interesse sind hier die Instruktionen Instr, die die globale Umgebung und/oder den lokalen Zustand eines Agenten ändern.
- Assign ist die Zuweisung des Wertes eines Ausdruckes an einen Bezeichner.
- Activate gibt die Kontrolle an den mit 'target' bezeichneten Agenten weiter und stößt in diesem die (von außen zugängliche) Operation 'opid' unter den aktuellen Parametern 'ms' an.
- Ein Script enthält ein Creation-Pattern, das hier als Liste von formalen Parametern aufgefaßt wird, und eine Menge von Operationen, die den Prozeßblock bilden. Von einer Operation wird angenommen, daß sie in ihrem Code mindestens eine Instruktion enthält.

2.1.2 Abgemagerte denotationale Semantik für CSSA

Semantische Bereiche:

$$
\begin{aligned}
m &: \text{Agent} = \text{Beh} \times \text{LState} \\
\beta &: \text{Beh} = \text{LState} \longrightarrow \text{Mess} \times \text{GEnv} \longrightarrow E \\
& \quad E = \text{Mess} \times \text{Loc} \times \text{GEnv} \\
\rho &: \text{GEnv} = \text{Loc} \longrightarrow \text{Agent} \\
\sigma &: \text{LState} = \text{Id} \longrightarrow \text{Acq} \\
aq &: \text{Acq} = \text{Id}^{*} \times (\text{Loc} + \text{Script} + \text{PrimObj}) \\
me &: \text{Mess} = \text{Acq}^{*} \times \text{Id}^{\{0,1\}}
\end{aligned}
$$

- Ein Agent ist durch sein Verhalten und seinen lokalen Zustand bestimmt.
- Das Verhalten Beh beschreibt das Input-/Output-Verhalten eines Agenten in einer bestimmten Umgebung.
- Ein Ereignis (aus E) ist das Ergebnis des Verhaltens in einer aktuellen Situation und gibt eine neue Nachricht an einen anderen (Ziel-) Agenten an sowie die neue, geänderte Umgebung.
- Eine globale Umgebung (aus GEnv) identifiziert die Agenten über Locations leLoc.
- Der lokale Zustand (aus LState) eines Agenten bindet Bekanntschaften an (lokale) Bezeichner. In CSSA gibt es ausschließlich lokale Bezeichner.

- Eine Bekannschaft (aus Acq) besteht aus einer Liste von Zugriffsrechten (Operationsbezeichner) und einem Zugriffsweg auf ein CSSA-Objekt.
- PrimObj ist die Menge der nicht nähers spezifizierten primitiven Objekte in CSSA.

Semantische Funtionen:

$\mathcal{I}$: Instr $\longrightarrow$ GEnv x LState $\longrightarrow$ GEnv x LState
$\mathcal{E}$: Expr $\longrightarrow$ GEnv x LState $\longrightarrow$ GEnv x Acq
$\mathcal{S}$: Script $\longrightarrow$ Mess$\downarrow$1 $\longrightarrow$ Agent

Hilfsfunktionen:

Follow : E $\longrightarrow$ E
Match : Id^* x Acq^* x LState $\longrightarrow$ LState $\cup$ $\{\perp\}$
Event : $Instr^*$ x Instr $\longrightarrow$ GEnv x LState $\longrightarrow$ E
Length : ($Instr^*$ + $Expr^*$ + Id^* + Acq^*) $\longrightarrow$ N
Cons : Acq x Acq^* $\longrightarrow$ Acq^*
Newloc : GEnv $\longrightarrow$ Loc

- Length bestimmt die Länge einer Liste.
- Cons hängt ein Element an eine bereits bestehende Liste an. '$\perp$' sei die leere Liste.
- Newloc beschafft aus einem (unendlichen) Vorrat von Locations eine neue Location, die abhängig von der aktuellen Umgebung bestimmt wird. Newloc sei injektiv.
- Match führt die Parameterübergabe durch und initialisiert den lokalen Zustand eines Agenten.
- Event berechnet das durch eine Operation eines Prozeßblockes bestimmte Ereignis.
- Follow berechnet zu einem Ereignis das Folgeereignis und beschreibt damit auf der Ebene des Agentennetzes die Aktivitäten (Kontrollfluß) bzw. das E/A-Verhalten eines Agenten.

Im folgenden wird vorausgesetzt, daß sich bei Änderungen eines lokalen Zustandes σ eines Moduls (β,σ) zu σ' implizit auch die momentane globale Umgebung ρ ändert, d.h. sei L:={loceLoc: ρ(loc)=(β,σ)}, dann gilt nach Änderung von σ zu σ' die globale Umgebung ρ':=λ loc.loc e L $\longrightarrow$ $(\beta,\sigma')|\rho$(loc).

$\mathcal{I}[\![$a:Assign$]\!]\rho\sigma$ =

sei x:= ls.a; e:=rs.a in
sei res:=$\mathcal{E}[\![$ e$]\!]\ \rho\sigma$ in
sei Zuweg:= res↓2 in
sei σ':=λ id.id=x ⟶ Zuweg|σ(id); ρ':= res↓1 in
(ρ,σ')

Eine Zuweisung ändert in erster Linie den lokalen Zustand.
Da die rechte Seite der Zuweisung ein Genagent-Ausdruck sein kann, ändert sich u.U. auch die globale Umgebung.

$\mathcal{I}[\![$act: Activate$]\!]\ \rho\sigma$ =

sei el:=(mess.act)↓1; t:=target.act in
n:=Length(el); ρ_1:=ρ in
sei case n>0:
ρ_{i+1}:=($\mathcal{E}[\![$ el_i $]\!]\ \rho_i\sigma$)↓1, i=1,...,n in
(ρ_{n+1},σ)

Das Ergebnis eines Activate-Aufrufs ist die durch die Abarbeitung der Parameter in der Nachricht 'mess.act' erzeugte neue globale Umgebung. Der lokale Zustand bleibt unverändert, da nur Ausdrücke interpretiert werden.

$\mathcal{E}[\![$gen: Genagent$]\!]\ \rho\sigma$ =

sei scr:= sc.gen; parm:=init.gen; actpar:=$\perp$;ρ_1:=ρ in
sei n:=Length(parm);locagent:= Newloc(ρ) in
sei case n>0:
(sei e_i:=$\mathcal{E}[\![$ $parm_i$ $]\!]\ \rho_i\sigma$ in
actpar:=Cons(e_i↓2,actpar);
ρ_{i+1}:=e_i↓1) i=1,...,n in
sei ρ':=λ loc.loc=locagent ⟶ $\mathcal{S}[\![$ scr $]\!]$ actpar |ρ_{n+1}(loc) in
(ρ',(access.gen, locagent))

Zunächst wird eine neue Location (locagent) besorgt, an die der neue Agent gebunden wird. Diesen neuen Agenten erhält man durch Interpretation des zugehörigen Scripts scr mit dem Creation-Pattern parm.
Ergebnis der Interpretation eines Genagent-Aufrufes ist die neue Umgebung ρ' und eine Bekanntschaft zu dem neuen Agenten.

```
Match(pat,aq,σ) =
                sei n:= Length(pat);k:= Length(aq)  in
                case  n ≠ k: ⊥
                case  n = k: λ id.id=pat_i, ie{1,...,n} ⟶ aq_i|σ(id)

Event⟦instrl: Instr*; instrend: Instr⟧ρσ =
                sei n:= Length(instrl)  in
                sei (ρ';σ'):= n=0 ⟶ (ρ,σ)|
                                  ℑ⟦instrl_n⟧(ℑ⟦instrl_n-1⟧...ℑ⟦instrl_1⟧ρσ)...)
                                                                  in
                case instrend≠Activate:
                             (⊥,⊥,(ℑ⟦instrend⟧ρ'σ')↓1)
                case instrend=Activate:
                             sei el:=(mess.instrend)↓1; op:=(mess.instrend)↓2;
                                 actpar:=⊥; ρ_1:=ρ'; k:=Length(el)          in
                             sei case k > 0:
                                    (sei e_i:=ℰ⟦el_i⟧ρ_iσ'  in
                                     actpar:=Cons(e_i↓2,actpar); ρ_i+1:=e_i↓1)
                                     i=1,...,k                              in
                    ((actpar,op),σ'(target.instrend)↓2,ρ_n+1)
```

Event interpretiert eine Instruktionsliste und bestimmt ausgehend von der dabei erhaltenen neuen Umgebung ein Ereignis, das davon abhängt, ob die letzte Anweisung der Instruktionsliste ein Activate-Aufruf war oder nicht.

Event bewirkt bei der Abarbeitung der Instruktionsliste i.a. einen Seiteneffekt auf den lokalen Zustand σ, der dadurch in σ' übergeht.

```
ℑ⟦sc: Script⟧aq =
                sei σ:= Match(create.sc,aq,⊥)  in
                sei β(σ):= λ me:Mess,ρ:GEnv.
                    sei σ':=Match(entry.(me↓2).procblock.sc,me↓1,σ) in
                    Event⟦code.(me↓2).procblock.sc⟧ρσ'              in
                    (β,σ)
```

Die Liste aq der aktuellen Script-Parameter wird gegen das Creation-Pattern des Scripts sc gematcht, wodurch der lokale Zustand zu σ initialisiert wird.

$\perp$ bedeutet hier die Funktion, die auf Id undefiniert ist.
Das Verhalten des aus dem Script sc unter dem Parametersatz aq erzeugten Agenten wird durch die Hilfsfunktion Event berechnet.

$$Follow((me,loc,\rho)) = ((\rho(loc)\downarrow 1)(\rho(loc)\downarrow 2))(me,\rho)$$

Follow wendet das Verhalten des durch loc identifizierten Agenten an auf die Nachricht me und die gegenwärtige Umgebung ρ im lokalen Zustand dieses Agenten.

Nach Voraussetzung wird der unter dem Seiteneffekt des Verhaltens (vgl.Event) geänderte Modul $\rho(loc)$ wieder durch die Location loc identifiziert, d.h. falls $Follow((me,loc,\rho)) = (me',loc',\rho')$, so gilt

$$\rho'(loc) = (\beta,\sigma'),$$

wo $\rho(loc) = (\beta,\sigma)$ und σ' der durch die Seiteneffekte geänderte lokale Zustand σ.

2.2 CLU und ALPHARD

([ALPH 76], [ALPH 78], [CLU 74], [CLU 77], [CLU 78])

CLU und ALPHARD enthalten als sprachliche Mittel zur Datenabstrakion das Cluster- bzw. Form-Konzept. Durch Angabe aktueller Parameter (vgl.Creation-Match in CSSA) kann mittels eines Clusters bzw. einer Form ein Modul erzeugt werden. Dieser besteht wie in CSSA aus einem lokalen Zustand und einem Verhalten. Ein Cluster bzw. eine Form besteht aus einer Schnittstelle, die Angaben zu den Parametern und die Bezeichner der von außen zugreifbaren Operationen enthält, und einem Rumpf mit den frei programmierbaren Prozeduren (Operationen).

Das Verhalten eines Moduls ist dessen E/A-Verhalten sowie die Weitergabe der Kontrolle. Diese geschieht auf konventionelle Weise durch Rückgabe über eine Return-Anweisung an die aufrufende Instanz. Eine beliebige Weitergabe der Kontrolle durch die explizite Angabe einer Zielinstanz wie in CSSA ist nicht möglich.

Intern werden Moduln durch bereits definierte Datenstrukturen realisiert. In CLU sind dies in der Regel Reihe und Verbund.

Innerhalb eines Moduls sind als freie Variable nur Bezeichner von extern definierten Moduln zulässig. Dies hat wieder eine Unterscheidung von lokalem Zustand und globaler Umgebung zur Folge.
In diesem Bindungsmodell werden Moduln zweistufig referenziert: der lokale Zustand eines Moduls gibt zu einem (lokalen) Bezeichner einen (globalen) Bezeichner einer Location an. Mittels der globalen Umgebung kann über diese Location dann der referenzierte Modul identifiziert werden.

Zusätzlich zu diesen Eigenschaften, die CLU und ALPHARD gemeinsam haben, verfügt ALPHARD noch über folgende Möglichkeiten:

1. Jeder Ausdruck ist auch Anweisung.
2. Shared-Objects sind möglich, d.h. bei der internen Realisierung mehrerer Moduln wird ein und dieselbe Datenstruktur verwendet.
3. Forms bzw. Moduln können geschachtelt werden.
4. Blockschachtelung ist möglich.
5. Als primitive Objekte treten nur Speicherzelle (raw-storage) und boolean auf.

Es ist nun die Frage zu klären, welche dieser Spracheigenschaften in die Mischsprache CLALPHARD zuzüglich zu den eingangs aufgeführten aufgenommen werden.

ad 1: Bei Ausdrücken interessieren im folgenden nur Operationsaufrufe (Call und Create); die logischen und arithmetischen Ausdrücke werden nicht nähers spezifiziert.
Da in CLU Operationsaufrufe (nicht Ausdrücke allgemein) Ausdrücke und Anweisungen sind, wird diese Regelung für CLALPHARD übernommen.

ad 2: Auf shared-objects wird verzichtet, da sie für die weiteren Zwecke eine unwesentliche Spezialisierung sind (die interne Realisierung wird nicht behandelt).

ad 3: Die Schachtelung von Forms hat drei Konsequenzen:
- Eine Beschreibung der Schachtelung von Forms und Moduln bzw. der impliziten Erzeugung von Moduln würde die Berücksichtigung von Vereinbarungen notwendig machen.
- Bei der Erzeugung eines Moduls aus einer Form, in die weitere Forms geschachtelt sind, müssen zu den inneren Forms implizit noch die entsprechenden Moduln erzeugt werden.
  Der eigentliche Vorgang der Generierung eines Moduls aus einer Form bleibt aber unverändert gegenüber der expliziten Erzeugung,

d.h. die globale Umgebung muß erweitert und der jeweilige lokale Zustand eines Moduls initialisiert werden.

- Der Aufruf einer Operation in einem geschachtelten Modul unterscheidet sich vom einfachen Call nur durch die vorgestellte Liste der Namen der umgebenden Moduln.

Die Schachtelung von Forms bedeutet somit keine wesentliche Erweiterung der Sprachkonzepte von CLU. Deswegen wird in CLALPHARD darauf verzichtet.

ad 4: Die Blockschachtelung ermöglicht die Einführung eines neuen Vereinbarungsteils in den Programmtext.
Während einer Programmberechnung bedeutet der Eintritt in einen neuen Block daher eine Änderung des lokalen Zustandes und der globalen Umgebung. Bei Blockaustritt muß der lokale Zustand vor Blockeintritt mit einer Korrektur der nicht verdeckten Größen wiederhergestellt werden.
Es kann nun Locations bzw. Moduln geben, die nicht mehr über einen lokalen Bezeichner referenzierbar sind (Halde).
Die Berücksichtigung der Blockschachtelung bringt es mit sich, daß nun anstelle eines lokalen Zustands ein Keller von lokalen Zuständen verwaltet werden muß.
Diese Mechanismen sind neu gegenüber CLU, so daß die Blockschachtelung eine wesentliche Spracherweiterung darstellt; sie wird daher in CLALPHARD aufgenommen.
Die Beschreibung des Vereinbarungsteils wird sehr einfach gehalten und ähnlich der Parameterübergabe bei Moduln behandelt.

ad 5: Die primitiven Objekte spielen in ihrer konkreten Gestalt nur bei der internen Realisierung von Moduln eine Rolle. Diese wird aber nicht genauer beschrieben, da hier die Abstraktion im Vordergrund steht. Daher wird auch auf die Beschreibung der primitiven Objekte verzichtet.

Als Ergebnis läßt sich festhalten, daß die zu untersuchende Mischsprache CLALPHARD die Spracheigenschaften von CLU, erweitert um die Blockschachtelung enthält.

Für die nun folgende Angabe der abstrakten Syntax und der denotationalen Semantik wird die Notation aus 2.1 übernommen.

2.2.1 Abgemagerte abstrakte Syntax für CLALPHARD

```
Cluster     = (<clinter: Clinterface>,<clbd:Clbody>)
Clinterface = (<cltype: Id>,<parm: Id*>,<public: Id+>)
Clbody      = (<rep: Decl>,<create: Oper>,<op: Oper>+)
Oper        = (<opid: Id>,<parm: Id*>,<code:(<stmt:Statement*,ret:Expr*>)>)
Expr        = VExpr v Call
Statement   = Assign v Call v Block
Block       = (<dec:(<id:Id>,<val:Expr>)*>,<code:Statement+>)
Call        = (<mod:Id>,<op:Id>,<parm: Expr*>)
Assign      = (<ls:Id>,<rs:Expr>)
```

- Ein Cluster besteht aus einer Schnittstelle und einem Rumpf.
  Die Schnittstelle eines Clusters enthält dessen Bezeichner, die Liste der formalen Parameter zur Aktualisierung des Clusters (d.h. Erzeugung eines Moduls aus diesem Cluster) sowie eine Auflistung der von außen zugänglichen Operationen.

  Der Rumpf eines Clusters enthält Angaben zur internen Realisierung, den Bezeichner der Operation, über die aus dem Cluster ein Modul erzeugt werden kann (Create) und die Liste der Operationen des Clusters.

  In der CLU- bzw. ALPHARD-Syntax wird die Create-Operation nicht explizit aufgeführt, da die Erzeugung eines Moduls in einer Variablenvereinbarung implizit erfolgt. Der Deutlichkeit wegen und in Analogie zu CSSA wird diese Generierungsfunktion hier aber extra in die abstrakte Syntax aufgenommen.

- Die Repäsentationsklausel gibt an, wie ein Modul intern mittels primitiver Datenstrukturen realisiert wird.'Decl' steht dabei für einen entsprechenden Vereinbarungsteil. Da die interne Realisierung hier nicht interessiert, wird auch auf eine Beschreibung des Vereinbarungsteils verzichtet.

- Eine Operation ist definiert durch ihren Bezeichner, ihre formalen Parameter und eine Liste von Anweisungen, die im Fall einer Funktion durch oBdA einen Ausdruck abgeschlossen wird. Im Falle einer Prozedur, wo nur Seiteneffekte bewirkt werden, ist 'ret' die leere Liste.

- VExpr sind wieder arithmetische und logische Ausdrücke, die hier nicht spezifiziert werden.

- Ein Call ist im Fall eines Funktionsaufrufes ein Ausdruck, ansonsten eine Anweisung.
  Ein Funktionsaufruf kann insbesondere ein Create sein; in diesem Fall bezeichnet 'mod' ein Cluster.
- Von Interesse sind wieder Anweisungen, die die globale Umgebung oder den lokalen Zustand ändern.
- Ein Block kann einen Vereinbarungsteil enthalten, der hier als Liste von Paaren (Bezeichner, Initialisierungsausdruck) dargestellt wird.

2.2.2 Abgemagerte denotationale Semantik für CLALPHARD

Semantische Bereiche:

$$\begin{aligned} &\text{Modul} = \text{Beh} \times \text{LState} \\ \beta:\ &\text{Beh} = \text{LState} \longrightarrow \text{Id} \times \text{Obj}^* \times \text{GEnv} \longrightarrow \text{GEnv} \times \text{Obj}^* \\ \rho:\ &\text{GEnv} = \text{Loc} \longrightarrow \text{Modul} \\ \sigma:\ &\text{LState} = \text{Id} \longrightarrow \text{Obj} \\ &K \in \text{LState}^* \end{aligned}$$

- Das Verhalten (aus Beh) liefert in einem bestimmten lokalen Zustand und einer globalen Umgebung unter Angabe eines Operationsbezeichners und der zugehörigen aktuellen Parameter eine Liste von Objekten als Ergebnis und ändert u.U. die globale Umgebung und (über Seiteneffekte) den lokalen Zustand.
- Die Referenzierung eines Moduls geschieht wieder zweistufig über einen (lokalen) Bezeichner und eine (globale) Location.
- K bezeichnet im folgenden einen Keller von lokalen Zuständen, der wegen der Einführung der Blockschachtelung notwendig geworden ist.
- Objekte sind Locations, Cluster und Werte von einem bestimmten Typ.

Semantische Funktionen:

$$\begin{aligned} \mathcal{I} &: \text{Statement} \longrightarrow \text{GEnv} \times \text{LState}^+ \longrightarrow \text{GEnv} \times \text{LState}^+ \\ \mathcal{E} &: \text{Expr} \longrightarrow \text{GEnv} \times \text{LState} \longrightarrow \text{GEnv} \times \text{Obj} \\ \mathcal{C} &: \text{Cluster} \longrightarrow \text{Obj}^* \longrightarrow \text{Modul} \end{aligned}$$

Hilfsfunktionen:

Effect	: $Obj^* \times Id \times Loc \times GEnv \longrightarrow GEnv \times Obj^*$
Match	: $Id^* \times Obj^* \times LState \longrightarrow LState \cup \{\bot\}$
Result	: $Statement^* \times Expr^* \longrightarrow GEnv \times LState \longrightarrow GEnv \times Obj^*$
Length	: $(Expr^* + Statement^* + Obj^*) \longrightarrow N$
Cons	: $Obj \times Obj^* \longrightarrow Obj^*$
Newloc	: $GEnv \longrightarrow Loc$
GenRep	: $Oper \times Decl \times Obj^* \longrightarrow LState$
Createstack	: $\longrightarrow LState^*$
Top	: $LState^+ \longrightarrow LState$
Pop	: $LState^+ \longrightarrow LState^*$
Push	: $LState^* \times LState \longrightarrow LState^*$

- Match, Length, Cons, Newloc sind analog den jeweiligen Hilfsfunktionen in CSSA.
- Effect entspricht in CSSA der Hilfsfunktion Follow.
  Diese Abbildung beschreibt die Aktivitäten einer Menge von Moduln auf der Ebene der globalen Umgebung.
  In einer Umgebung $\rho \in GEnv$ wird in einem Modul, der durch loc $\in$ Loc identifiziert wird, eine Operation mit der Bezeichnung ideId unter dem aktuellen Parametersatz $oleObj^*$ aufgerufen und ergibt eine u.U. geänderte Umgebung sowie (im Fall einer Funktion) eine Ergebnisliste.
- Result entspricht in CSSA die Hilfsfunktion Event und berechnet die Semantik der einzelnen Operationen bzw. das Verhalten eines Moduls.
- GenRep erzeugt abhängig vom Create-Code, der Darstellungsvorschrift und den aktuellen Parametern eines Clusters die interne Realisierung eines Moduls und initialisiert somit den lokalen Zustand dieses neu erzeugten Moduls.
  Da die interne Realisierung eines Moduls hier irrelevant ist, wird GenRep nicht näher bestimmt.
- Createstack, Top, Pop und Push dienen zur Verwaltung des Zustandkellers bei der Abarbeitung einer Blockstruktur.

Wie in 2.1.2 wird im folgenden vorausgesetzt, daß wenn sich der lokale Zustand eines Moduls ändert, daß sich dann die gegenwärtige globale Umgebung

implizit auch so ändert, daß dieser Modul nach wie vor durch die gleiche(n) Location(s) identifiziert wird.

```
𝒥⟦a: Assign⟧ρK =
                sei σ:= Top(K); x:= ls.a;  e:=rs.a  in
                sei res:=ℰ⟦e⟧ρσ                    in
                sei σ':=λ id.id=x ⟶ res↓2|σ(id);
                    ρ':= res↓1                     in
                    (ρ',Push(Pop(K),σ'))
```

Für die Interpretation einer Anweisung ist der oberste Zustand im Keller K maßgebend. Dieser Zustand wird nach der ausgeführten Zuweisung durch den neuen lokalen Zustand σ' ersetzt.

```
𝒥⟦bl: Block⟧ρK = sei n:=Length(dec.bl); σ1:=Top(K); ρ1:=ρ;
                    m:=Length(code.bl); cb:=code.bl        in
                sei case n>0:
                      (sei ei:=ℰ⟦val.(dec.bl)i⟧ρiσi        in
                      σi+1:=λ bez.bez=id.(dec.bl)i ⟶ ei↓2|σi(bez);
                      ρi+1:=ei↓1) i=1,...,n                 in

                sei K':=Push(K,σn+1)                        in
                sei (ρ",K"):=𝒥⟦cbm⟧(...𝒥⟦cb1⟧ρn+1K')...)    in
                sei  σ":= Top(K")                           in
                sei σ"':=λ bez.beze{(id.dec.bl)i:ie{1,...,n}}
                              ⟶ σ1(bez)|σ"(bez)             in

                    (ρ", Push(Pop(Pop(K")),σ"'))
```

Zunächst wird der Vereinbarungsteil abgearbeitet, wodurch einige Bezeichner des vorherigen lokalen Zustands (σ_1) evt. verdeckt werden, d.h. neue Bindungen erhalten. Diese Abarbeitung kann auch die globale Umgebung ändern, falls unter den Ausdrücken ein Create-Aufruf vorkommt.

Der neue lokale Zustand wird auf den Keller geschrieben und dann der Blockrumpf abgearbeitet.

Am Ende des Blocks muß der Keller wieder aktualisiert werden. Das oberste Kellerelement wird entfernt; in dem nun obersten Zustand im Keller (d.h. der aktuelle Zustand vor Blockeintritt) müssen alle durch den Ver-

einbarungsteil des Blocks nicht verdeckten Bezeichner auf die Werte am Ende des Blocks gesetzt werden. Dies geschieht in der Definition des Zustands $\sigma"'$, der neuer, aktueller Zustand wird (d.h. oberstes Element im Keller).

Die Gültigkeit der globalen Umgebung wird durch die Blockschachtelung nicht berührt. Allerdings ist es jetzt möglich, daß es Moduln mit einer zugehörigen Location gibt, die u.U. nicht mehr über einen (lokalen) Bezeichner referenzierbar sind (Haldenbildung). Dieser Fall tritt dann ein, wenn im Blockinnern zu einem extern definierten Cluster ein Modul erzeugt wird.

$$\mathrm{Effect}[\![ol{:}Obj^*, op{:}Id, mod{:}Loc, \rho{:}GEnv]\!] =$$

$$((\rho(mod)\downarrow 1)(\rho(mod)\downarrow 2))(op, ol, \rho)$$

Effect berechnet das Verhalten des durch 'mod' bezeichneten Moduls und wendet dieses auf 'op' und 'ol' in der Umgebung 'ρ' und im lokalen Zustand des durch 'mod' bezeichneten Moduls an.

$$\mathcal{J}[\![c{:}\ Call]\!]\rho K =$$

sei n:=Length(parm.c); $\rho_1 := \rho$;
    σ:= Top(K); actpar:=$\perp$ in
sei case n>0:
        (sei $e_i := \mathcal{E}[\![(parm.c)_i]\!]\rho_i\sigma$ in
        actpar:= Cons($e_i\downarrow 2$,actpar);
        $\rho_{i+1} := e_i\downarrow 1$) i=1,...,n in
sei ρ':=Effect(actpar,op.c,σ(mod.c),ρ_{n+1})$\downarrow 1$ in

    (ρ',Push(Pop(K),ρ'(σ(mod.c))$\downarrow 2$))

Die Wirkung einer Moduloperation wird beschrieben durch die Änderung der globalen Umgebung.
Im Fall einer Selbstaktivierung eines Moduls ändert sich auch dessen lokaler Zustand.
Zunächst werden die aktuellen Parameter der auszuführenden Operation berechnet und dann diese Operation im Zielmodul $\rho_{n+1}(\sigma(mod.c))$ aufge-

rufen. Da es sich im vorliegenden Fall um einen Prozedur- und nicht um einen Funktionsaufruf handelt, liefert der Aufruf von Effect kein Objekt als Ergebnis. Allerdings kann sich infolge von Seiteneffekten der lokale Zustand des aufgerufenen Moduls ändern.

Der Kontrollfluß zwischen den Moduln ist hierarchisch, was durch die Schachtelung des Effect-Aufrufs in die Interpretation des Prozedurrumpfes zum Ausdruck kommt.

$\mathcal{E}[\![$ce:Call$]\!]\rho\sigma$ =

sei n:= Length (parm.ce); ρ_1:=ρ; actpar:=$\perp$ in
sei case n>0:

($\underline{\text{sei}}$ e_i:=$\mathcal{E}[\![$(parm.ce)$_i]\!]\rho_i\sigma$ in
actpar:=Cons(e_i↓2,actpar);
ρ_{i+1}:=e_i↓1) i=1,...,n in

case op.ce = Create:

sei locmod:=Newloc(ρ_{n+1}); cl:=σ(mod.ce) in
sei ρ':=λ loc.loc=locmod ⟶ $\mathcal{C}[\![$cl$]\!]$actpar$|\rho_{n+1}$(loc) in

(ρ',locmod)

case op.ce ≠ Create:

Effect(actpar,op.ce,σ(mod.ce),ρ_{n+1})

Gemäß der Syntax liefert eine Funktion genau ein Ergebnis und nicht eine Liste von Objekten als Ergebnis. Damit ist auch die Funktionalität von $\mathcal{E}$ gewahrt.

Bei der Erzeugung eines Moduls wird als Ergebnis des Aufrufs die Location dieses Moduls geliefert.

Result$[\![$stmt:Statement*; ret:Expr*$]\!]\rho\sigma$ =

sei K:=Push(Createstack,σ)); n:=Length(stmt);
k:=Length(ret) in
sei (ρ',K'):= n=0 ⟶ (ρ,K)$|$
$\mathcal{I}[\![$stmt$_n]\!](\mathcal{I}[\![$stmt$_{n-1}]\!]\ldots\mathcal{I}[\![$stmt$_1]\!]\rho$K) in

case k=0:

(ρ',$\perp$)

case k=1:

$\mathcal{E}[\![\text{ret}]\!]\rho'(\text{Top}(K'))$

Result interpretiert den Code einer Operation im Rumpf eines Clusters Result bewirkt u.U. auf den lokalen Zustand σ einen Seiteneffekt. Gemäß den Vereinbarungen zur Syntax gilt, daß 'ret' entweder die leere Liste ist oder einelementig ist.

$\mathcal{C}[\![\text{cl: Cluster}]\!]\text{ol} =$

sei σ= GenRep(create.clbd.cl,rep.clbd.lc,ol) in
sei $\beta(\sigma)$:=λ bez:Id parml:Obj* ρ:GEnv.
bez $\notin$ public.clinter.cl $\longrightarrow \perp$|
sei σ':=Match(parm.bez.clbd.cl,parml,σ) in
Result(code.bez.clbd.cl)$\rho\sigma'$ in

(β,σ)

ol wird verwendet zur Initialisierung eines neu zu erzeugenden Moduls, dessen Verhalten durch die Hilfsfunktion Result bestimmt wird.

3. KORREKTE IMPLEMENTIERUNG EINES ABSTRAKTEN DATENTYPS DURCH CLUSTER BZW. SCRIPTS

In diesem Kapitel wird eine Definition bzw. ein Kriterium entwickelt, wann eine Spezifikation eines nicht typparametrisierten abstrakten Datentyps korrekt durch eine Menge von Cluster bzw. Scripts implementiert wird.

Der Schlüsselbegriff dafür ist die Modulalgebra, die ausgehend von der Semantik der Bespielsprachen aus Kapitel 2 das Verhältnis der aus den Clustern bzw. Scripts erzeugten Moduln zueinander beschreibt. Dabei wird versucht, die Signatur der Modulalgebra der Signatur aus der Datentypspezifikation möglichst weitgehend anzugleichen. Zu diesem Zweck werden in 3.1 an die Cluster/Scripts syntaktische Forderungen gestellt.

Um die Gleichungen einer Datentypspezifikation in der Modulalgebra zu berücksichtigen muß für die Elemente der Modulalgebra zunächst definiert werden, wann zwei von ihnen verhaltensgleich sind. Dies geschieht in 3.4. Die Definition der Verhaltensgleichheit geht von einem terminalen Ansatz aus, weshalb für den abstrakten Datentyp eine terminale Semantik unterstellt wird.

In 3.3 werden den einzelnen Termen der Termalgebra einer Datentypspezifikation entsprechende Elemente der Modulalgebra zugeordnet. Diese Abbildung EVAL stellt dann die Verbindung zwischen der Termalgebra und der Modulalgebra her. Das Bild der Termalgebra unter EVAL wird in 3.6 untersucht.

In 3.7 wird versucht, für den Fall einer Erweiterung einer Spezifikation die zugehörigen Modulalgebren in analoger Weise ineinander einzubetten.

3.1 Syntaktische Erfüllung einer Signatur durch eine Menge von Cluster bzw. Scripts

Sei im folgenden eine Signatur (S,Σ) gegeben.

3.1.1 Im Vereinbarungsteil eines Clusters/Scripts wird der Definitionsbereich des lokalen Zustands eines aus diesem Cluster/Script erzeugbaren Moduls bestimmt.
Variable eines bestimmten Typs stehen also entweder für lokale Größen, die als Moduln realisiert sein können, oder für Locations von Moduln, die aus extern definierten Clustern/Scripts erzeugt wurden. Diese Cluster/Scripts bzw. Datentypen treten in der Funktionalität der Prozeduren und Funktionen der Cluster/Scripts auf.

Somit kann jeder Sorte seS ein Cluster/Script (bzw. dessen Bezeichner) A'_S zugeordnet werden.

3.1.2 In CLALPHARD können Prozeduren P vorkommen, die kein Ergebnis liefern, sondern nur Seiteneffekte auf einen Modul bewirken, der aus einem Cluster erzeugt wurde, dessen Bezeichner im Definitionsbereich dieser Prozedur vorkommt. Bei der Funktionalität von P wird daher dieser Clusterbezeichner (Typ) explizit im Zielbereich von P angegeben.

Beispiel: Einfügen in eine Liste von DATA-Elementen:
INSERT: (index: INTEGER,element: DATA,liste: cvt); <code>;
Diese Prozedur hat dann die Funktionalität:
INSERT: INTEGER x DATA x LISTE —> LISTE

(cvt ist in CLU ein reserviertes Symbol und steht für 'convert', weist also auf eine Änderung des Standpunktes hin, ob ein Typ in seiner externen, abstrakten Form oder in seiner internen Darstellung gesehen wird.)

3.1.3 OBdA wird vorausgesetzt, daß alle Prozeduren und Funktionen einen einfachen Zielbereich, also kein kartesisches Produkt haben. Dies bedeutet insbesondere, daß Funktionen keine Seiteneffekte haben dürfen (vgl. 3.1.2)

3.1.3.1 Funktionalität der Operationen in CSSA

Sei P eine Operation≠ Create des Scripts A'_{sn} mit den Parameterbereichen (d.h. den Typen der jeweiligen aktuellen Parameter) $A'_{s1},\ldots,A'_{sn}$ (A'_s wird also sowohl als Clusterbezeichner als auch als Bezeichner der Menge von Objekten vom Typ A'_s verwendet).
Gemäß der Semantik von CSSA (vgl.2.1.2) liefert ein Operationsaufruf als Ergebnis ein Ereignis E=(mess,target,env). Hier ist aber nur das Objekt als Ergebnis interessant, das P als abstrakte Funktion berechnet. Dieses wird in der Liste mess↓1 weitergegeben. Da die Liste mess↓1 noch weitere Informationen (etwa für den Kontrollfluß) enthalten kann, wird vereinbart, daß dieses Ergebnis die erste Komponente dieser Liste enthält und z.B. vom Typ A'_s ist. Dann hat P (bei geeigneter Indizierung) die Funktionalität

$$P: A'_{s1} x \ldots x A'_{sn} \longrightarrow A'_s.$$

3.1.3.2 Funktionalität der Operationen in CLALPHARD

Sei P eine Operation eines Clusters A'_{sn} mit den Parameterbereichen $A'_{s1},\ldots,A'_{sn-1},A'_{sn}$ und dem Ergebnisbereich A'_s.
Dann hat P die Funktionalität (bei geeigneter Indizierung):

$$P: A'_{s1} x \ldots x A'_{sn} \longrightarrow A'_s$$

Der Unterschied zu CSSA rührt daher, daß in CSSA keine interne Repräsentation der Datentypen existiert wie in CLU und ALPHARD und der zu aktivierende Agent des Scripts A'_{sn} als gesonderter Parameter beim Activate-Befehl angegeben wird.

In CLU und ALPHARD gibt es ein implizites Create, durch das ein primitives Element eines Typs erzeugt wird. Dies geschieht in einer Variablenvereinbarung mit (Standard-) Initialisierung.

Z.B.: var s=stack

Hier wird für die Variable 's' der Typ 'stack' festgelegt und gleichzeitig an s ein Exemplar des leeren Kellers gebunden.
In der Definition der Modulalgebra wird diese implizite Erzeugung eines Moduls explizit als Funktion mit der passenden Funktionalität eingeführt.

3.1.4 Hilfsfunktionen und -prozeduren eines Clusters/Scripts, die nur definitorischen Charakter haben und nicht in der Liste der von außen zugreifbaren Operationen dieses Clusters/Scripts aufgeführt sind, werden nicht berücksichtigt.

3.1.5 Insgesamt kann also gefordert werden, daß jedem $f \in \Sigma$ genau eine (Funktions-) Prozedur eines Clusters/Scripts mit einer passenden Funktionalität zugeordnet werden kann.

Die syntaktischen Forderungen aus 3.1.1 bis 3.1.5 an eine Menge von Cluster/Scripts lassen sich in der folgenden Definition präzisieren.

3.1.6 Definition (Erfüllung einer Signatur)

Sei (S,Σ) eine Signatur, C eine Menge von Cluster/Scripts.
Sei OP_C die Menge aller in den Clustern/Scripts von außen zugreifbaren Operationen (inklusive Create).
C *erfüllt die Signatur* (S,Σ) :gdw

1. es existiert α: C $\longrightarrow$ S bijektiv

und 2. es existiert γ: $OP_C \longrightarrow \Sigma$ bijektiv

so daß gilt:

$$(\forall P_f \in OP_C)(P_f : A'_{s1} \times \ldots \times A'_{sn} \longrightarrow A'_s \Longrightarrow \gamma(P_f) = f \in \Sigma_{\alpha(s1)\ldots\alpha(sn),\alpha(s)})$$

Notation: $Erf(C,S,\Sigma,\alpha,\beta)$ oder kurz: $Erf(C,S,\Sigma)$.

3.1.7 Beispiel: INTSTACK

(Der besseren Lesbarkeit wegen wird dieses (und andere) Beispiel(e) in der Original-CLU-Syntax formuliert)

(S,Σ) sei Signatur mit

$S := \{integer, stack\}$

$\Sigma := \{$

null :		$\longrightarrow$ integer ,
succ,pred :	integer	$\longrightarrow$ integer ,
createstack:		$\longrightarrow$ stack ,
Pop :	stack	$\longrightarrow$ stack ,
Top :	stack	$\longrightarrow$ integer,
Push :	integer x stack	$\longrightarrow$ stack$\}$

$C := \{cl_1, cl_2\}$ sei eine Clustermenge mit

```
cl1:= INTEGER cluster is Null, Succ, Pred;
        rep = ...;
```

```
        Null: <code>;
        Succ: (n: cvt) returns n': cvt; <code>;
        Pred: (n: cvt) returns n': cvt; <code>;

        end INTEGER

cl2:= STACK cluster is Createstack, Pop, Top, Push;
        rep = ...;
        Createstack: <code>;
        Pop: (s: cvt); <code>;
        Top: (s: cvt) returns n: INTEGER; <code>;
        Push: (n: INTEGER; s: cvt); <code>;

        end STACK
```

Die abstrakte Syntax (vgl. Kapitel 2) der beiden Cluster lautet:

cl1: ((INTEGER,(),(Null,Succ,Pred)),<rep-Code INTEGER>,(Null,(),<Null-Code>),
((Succ,(INTEGER),<Succ-Code>),(Pred,(INTEGER),<Pred-Code>))))

cl2: ((STACK,(),(Createstack,Pop,Top,Push)),<rep-Code STACK>,
(Createstack,(),<Createstack-Code>),((Pop,(STACK),<Pop-Code>),
(Top,(STACK),<Top-Code>),(Push,(INTEGER,STACK),<Push-Code>))).

Die Ergebnisbereiche der einzelnen Operationen hängen ab von dem Return-Ausdruck am Ende des Operationscodes bzw. falls dieser fehlt, ist der Ergebnisbereich gleich dem eigenen Clusterbezeichner.

OP_C = {Null, Succ, Pred, Createstack, Pop, Top, Push}

Die Operationen $P_f \in OP_C$ haben folgende Funktionalitäten:

Null	:	⟶ INTEGER
Succ, Pred	: INTEGER	⟶ INTEGER
Createstack	:	⟶ STACK
Pop	: STACK	⟶ STACK
Top	: STACK	⟶ INTEGER
Push	: INTEGER x STACK	⟶ STACK

α und γ werden auf naheliegende Weise definiert:

α:(INTEGER,STACK) ⟼ (integer,stack) komponentenweise

γ:(Null,Succ,Pred,Createstack,Pop,Top,Push) ⟼
(null,succ,pred,createstack,pop,top,push)
komponentenweise

α und γ sind wohldefiniert, bijektiv und erfüllen die Bedingungen in 3.1.6.

Damit gilt: Erf(C,S,Σ,α,γ).

3.2 Definition der Modulalgebra M_C

Sei im folgenden (S,Σ) eine Signatur und C eine Menge von Cluster /Scripts, so daß α und γ existieren mit $Erf(C,S,\Sigma,\alpha,\gamma)$, d.h. die Cluster-/Script-Menge C erfüllt die syntaktischen Forderungen aus 3.1.

Aus einem Cluster/Script können über einen Create-Aufruf Moduln erzeugt werden. Durch Anwendung von Operationen kann deren lokaler Zustand geändert werden, was zu neuen Moduln führt. Es liegt daher nahe, eine Algebra zu konstruieren, deren Elemente Moduln und deren Funktionen die Operationen dieser Moduln sind. Dabei sollte die Definition dieser Algebra sich an der der Termalgebra T_Σ orientieren.

Bei diesem Vorgehen, eine Algebra über den Moduln der Cluster-/Script-Menge C zu definieren, ergeben sich allerdings noch einige Schwierigkeiten.

Die Moduln, die Trägermengen bilden sollen, sind abhängig von den (Ergbnissen der) Operationen der zu definierenden Algebra. Andererseits können diese Operationen nur dann definiert werden, wenn die Trägermengen bereits bekannt sind. Um die Gefahr einer Zirkeldefinition zu vermeiden, wird die Modulalgebra M_C induktiv definiert. Der Induktionsanfang ist stets möglich, wie in 3.2.1 bzw. 3.2.2 ausgeführt wird.

Die Definition der Modulalgebra sollte sich möglichst an der Konstruktion der Terme der Termalgebra T_Σ orientieren. Wünschenswert wäre eine Zuordnung Term – Modul. Wegen der zweistufigen Referenzierung extern definierter Moduln ist dies aber direkt nicht möglich. Ein Modul bzw. dessen lokaler Zustand σ ist vielmehr ein Schema, das durch die Bindungen $\sigma(id)$=locaLoc – Modul durch eine globale Umgebung ausgefüllt wird.

Zu einem Modul muß also stets angegeben werden, in welcher Umgebung er betrachtet wird. Damit kann höchstens die Zuordnung erreicht werden:

Modul in bestimmter Umgebung – Term $t \in T_\Sigma$.

Dies wirkt sich auf die Definition der Modulalgebra dahingehend aus, daß nicht die Moduln selbst die Trägermengen bilden, sondern Paare bestehend aus einem Modul und einer globalen Umgebung, in der dieser Modul betrachtet wird.

Da die Elemente der Modulalgebra über die Semantik der Beispielsprachen gewonnen werden, ist es nicht möglich, dafür die Moduln selbst (bzw. Moduln in bestimmten Umgebungen) zu wählen, wie folgende Überlegung zeigt.

Seien $\overline{P}$ eine n-stellige Operation der Modulalgebra, die aus einer (Funktions-) prozedur P eines Clusters/Scripts gewonnen wurde, und (a1,...,an) ein Parametersatz von $\overline{P}$. Sei $\overline{P}$(a1,...,an)=a,wobei a über die Semanik von P berechnet wird, also über ein Verhalten ß. Nun hat ß als Argumente in der Regel aber Locations und nicht Moduln (vgl.Kapitel 2). Zu einem Modul a_i müßte also zuerst eine Location l_i bestimmt werden mit $\rho(l_i)=a_i$, wo ρ die globale Umgebung ist, in der die a_i, i=1,...,n, betrachtet werden. l_i ist nur dann eindeutig bestimmbar, wenn ρ injektiv ist, was aber i.a. nicht der Fall zu sein braucht (z.B. zweimaliges Create desselben Clusters/ Scripts).

Umgekehrt ist aber a_i eindeutig bestimmt durch l_i.
Als Elemente der Trägermengen der Modulalgebra M_C werden deswegen Paare gewählt bestehend aus einer Location und der Umgebung, in der der durch diese Location identifizierte Modul betrachtet wird.

Die Trägermengen von M_C enthalten also als Elemente Paare (l,ρ) mit $l \in Loc$ und $\rho \in GEnv$ und es gilt:

$$M_C \downarrow l \supseteq \{A_s : s \in S\} \text{ mit}$$
$$(l,\rho) \in A_s \Longrightarrow \rho(l) = \text{m Modul des Typs } A'_s.$$

Primitive Objekte, die ebenfalls Elemente von Obj sind, wurden in der Semantik in 2.1 und 2.2 nähers spezifiziert.
Es wird aber vorausgesetzt, daß alle Objekte, die nicht Scripts, Cluster oder Locations sind, Moduln sind, da sie als Daten(strukturen) durch eine Spezifikation bzw. ein Cluster/Script definierbar sein müssen.

Aus diesem Ansatz folgt insbesondere die Existenz von Moduln (ß,σ), für die $\sigma=\perp$ und ß eine konstante (Zugriffs-) Funktion ist. In CLALPHARD bedeutet dies, daß diese Moduln keine interne Realisierung mehr haben.

Da nun alle Objekte außer Cluster, Scripts und Locations Moduln sind, können diese alle über Locations identifiziert werden. Ggf. können für primitive Objekte, die als Moduln aufgefaßt werden, Standardlocations verwendet weden, die einer Standardumgebung angehören, also allen Umgebungen angehören.

In CSSA wird die erste Komponente einer Bekanntschaft, die die Zugriffsbeschränkung enthält, hier nicht verwendet. Da außerdem Bekanntschaften, deren Zugriffsweg ein Script enthält, hier irrelevant sind, können der Einfachheit halber Bekanntschaften mit ihrer zweiten Komponente identifiziert werden.

Bemerkungen zur Funktionalität des Verhaltens ß

Das Verhalten ß eines Moduls (ß,σ) ist implizit abhängig vom lokalen Zustand σ (vgl. $\mathcal{S}[\![s: Script]\!]$ in 2.1.2 und $\mathcal{C}[\![d: Cluster]\!]$ in 2.2.2).
Durch Aufspalten der Definitionsbereiche erhält man daher:

in CLALPHARD: ß: LState —> Id x Obj^* x GEnv —> GEnv x Obj^*

ist äquivalent mit: ß: LState —> GEnv —>Id x Obj^* —> GEnv x Obj^*

Unter Berücksichtigung der Tatsache, daß Obj = Loc (vgl.oben) vereinbart wurde und ideId ein Operationsbezeichner aus Opid ist, gilt:

ß: LState —> GEnv —> Opid x Loc^* —> GEnv x Loc^*

in CSSA: ß: LState —> Mess x GEnv —> Mess x Loc x GEnv

Unter Berücksichtigung der Vereinbarung, daß Bekanntschaften mit ihrer zweiten Komponente gleichgesetzt werden, primitive Objekte als Moduln aufgefaßt und über Locations identifiziert werden und nur Ereignisse interessieren, die einen Operator enthalten, erhält man:

ß: LState —> GEnv —> $(Loc + Scripts)^*$ x Opid —>
—> Mess x Loc x GEnv

3.2.1 Induktive Definition der Modulalgebra M_C für CLALPHARD

Sei (S,Σ) eine Signatur, C eine Menge von Clusters und OP_C die Menge aller in den Clustern von C öffentlichen Operationen. Es gelte $Erf(C,S,\Sigma,\alpha,\gamma)$ für ein α und γ.
$P_f \in OP_C$ sei eine (Funktions-)Prozedur eines Clusters aus C mit $\gamma(P_f)=f\in\Sigma$.

3.2.1.1 Sei $\rho_0 \in GEnv$; $\perp \notin S$.

$A^X_\perp := \{(\perp,\rho_0)\}$ ρ_0 heißt dann *Standardumgebung*.

(Die Standardumgebung kann Hilfsgrößen für die Semantik enthalten, die auf der Betrachtungsebene des abstrakten Datentyps nicht auftreten).

$(\forall s\in S)\ A^X_s := \emptyset$

$(\forall P_f \epsilon OP_C)$

3.2.1.2 P_f ist Create ohne Parameter

dann: $(\exists s \epsilon S)\ P_f: \longrightarrow A'_s$

definiere:

$P_f^X: A_{\perp}^X \longrightarrow A_s^X$

Sei $m_s := \mathcal{C}[\![A'_s]\!]$

$(\forall \rho \epsilon A_{\perp}^X \downarrow 2)$

$[a_s := \text{Newloc}(\rho);\ \rho' := \lambda\ \text{loc}.\text{loc}=a_s \longrightarrow m_s|\rho(\text{loc})$

$A_{\perp}^X := A_{\perp}^X \cup \{(\perp,\rho')\}$

$A_s^X := A_s^X \cup \{(a_s,\rho')\}$

$(\forall s' \epsilon S)\ A_{s'}^X := A_{s'}^X \cup \{(l,\rho'):\ l \epsilon \text{dom}(\rho') \cap A_{s'}^X \downarrow 1\}$ (*)

$P_f^X((\perp,\rho)) := (a_s,\rho')]$

Es existiert mindestens ein P_f, das Create ohne Parameter ist, d.h. daß $\gamma(P_f) = f \epsilon \Sigma_{\varepsilon,s}$ für ein $s \epsilon S$, da sonst die Termalgebra $T_\Sigma = \emptyset$ wäre.
Falls also $T_\Sigma \neq \emptyset$, hat die induktive Definition der Modulalgebra einen Induktionsanfang durch P_f^X.

Der Definition von A_s^X liegt die Möglichkeit zugrunde, in jeder Umgebung einen beliebigen Modul zu erzeugen, bzw. daß ein Cluster in jeder Umgebung zu einem Modul initialisiert werden kann.

$\perp$ ist hier eine undefinierte Location, die dem leeren Wort ε zugeordnet ist. Dies hat nur formale Bedeutung beim Induktionsanfang der Modulalgebra M_C.

$A_{\perp}^X \downarrow 2$ enthält alle durch die Clustermenge C erzeugbaren Umgebungen (Modulgemeinschaften).

(*) wird Lifting genannt.
Das Lifting bewirkt, daß alle Locations $l \epsilon Loc$, die in der Modulalgebra vorkommen und in der Umgebung ρ betrachtet werden (d.h. $(l,\rho) \epsilon A_s^X$, $s \epsilon S$), auch in der geänderten Umgebung ρ' betrachtet werden, falls $\rho'(l)$ definiert ist.

3.2.1.3 P_f Prozedur

d.h. $(\exists n\in N)(\exists s1,\ldots,sn,s\in S)\ P_f: A'_{s1}x\ldots x\ A'_{sn} \longrightarrow A'_s$ und $s=sn$
bei geeigneter Indizierung (vgl.3.1.3.2)

Definiere:

$$P_f^X: A_{s1}^X x\ldots x\ A_{sn}^X \longrightarrow A_s^X$$

$$(\forall((a_{s1},\rho),\ldots,(a_{sn},\rho))\in A_{s1}^X x\ldots x\ A_{sn}^X \text{ mit}$$

$$(\sigma,P_f,(a_{s1},\ldots,a_{sn-1},rep(m_s)),\rho)\in dom(\beta),\ \text{wo } m_s:=(\beta,\sigma):=\rho(a_{sn}))$$

$$[(\rho',obj):=\beta(\sigma,P_f,(a_{s1},\ldots,a_{sn-1},rep(ms)),\rho)$$

$$A_\perp^X:=A_\perp^X \cup \{(\perp,\rho')\}$$

$$(\forall s'\in S)\ A_{s'}^X:=A_{s'}^X \cup \{(l,\rho'): l\in dom(\rho') \cap A_{s',\downarrow 1}^X\}$$

$$P_f^X((a_{s1},\rho),\ldots,(a_{sn},\rho)):=(a_{sn},\rho')]$$

Das Verhalten des Moduls $m_s=\rho(a_{sn})$ wird angesetzt auf den Parametersatz bestehend aus

- der Operation P_f
- den aktuellen Parametern von P_f, wobei $rep(m_s)$ die interne Darstellung (Datenstruktur) von $\rho(a_{sn})$ sein soll
- der Umgebung ρ.

Dieses Verhalten beschreibt gerade die Seiteneffekte von P_f auf den Modul $\rho(a_{sn})$. Im allgemeinen gilt also: $\rho(a_{sn})\neq\rho'(a_{sn})$, wobei $\rho'(a_{sn})$ der unter den Seiteneffekten von P_f geänderte Modul m_s ist (vgl. die Vereinbarung zur Änderung der Umgebung bei Seiteneffekten in 2.2.2).

'obj' ist hier undefiniert, da P_f kein Ergebnis liefert, sondern nur Seiteneffekte bewirkt.

Alle Parameter von P_f^X gehören derselben Umgebung an, da nur deren Verknüpfung sinnvoll ist.

3.2.1.4 P_f ist parametrisierte Funktion

d.h. $(\exists n\in N)(\exists s1,\ldots,sn,s\in S)\ P_f: A'_{s1}x\ldots x\ A'_{sn} \longrightarrow A'_s$

Definiere: $P_f^X: A_{s1}^X x\ldots x\ A_{sn}^X \longrightarrow A_s^X$

3.2.1.4.1 P_f ist Create mit Parametern

$$(\forall((a_{s1},\rho),\ldots,(a_{sn},\rho))\in A^X_{s1}x\ldots x\ A^X_{sn})$$
$$[a_s:=Newloc(\rho)$$
$$\rho':=\lambda\ loc.loc=a_s \longrightarrow \mathcal{C}[\![A'_s]\!](a_{s1},\ldots,a_{sn})|\rho(loc)$$
$$A^X_\perp:=A^X_\perp \cup \{(\perp,\rho')\}$$
$$A^X_s:=A^X_s \cup \{(a_s,\rho')\}$$
$$(\forall s'\in S)\quad A^X_{s'}:=A^X_{s'} \cup \{(l,\rho'):\ l\in dom(\rho')\cap A^X_{s'},\downarrow 1\}$$
$$P^X_f((a_{s1},\rho),\ldots,(a_{sn},\rho)):=(a_s,\rho')]$$

3.2.1.4.2 P_f Ist Funktionsprozedur ≠ Create

$$(\forall((a_{s1},\rho),\ldots,(a_{sn},\rho))\in A^X_{s1}x\ldots x\ A^X_{sn}\ \text{mit}$$
$$(\sigma,P_f,(a_{s1},\ldots,a_{sn-1},rep(m_{sn})),\rho)\in dom(\beta).m_{sn}:=\rho(a_{sn})=:(\beta,\sigma))$$
$$[(\rho',a_s):=\beta(\sigma,P_f,(a_{s1},\ldots,a_{sn-1},rep(m_{sn})),\rho)$$
$$A^X_\perp:=A^X_\perp \cup \{(\perp,\rho')\}$$
$$A^X_s:=A^X_s \cup \{(a_s,\rho')\}$$
$$(\forall s'\in S)\ A^X_{s'}:=A^X_{s'} \cup \{(l,\rho'):\ l\in dom(\rho')\cap A^X_{s'},\downarrow 1\}$$
$$P^X_f((a_{s1},\rho),\ldots(a_{sn},\rho)):=(a_s,\rho')\ ]$$

3.2.1.5 Lemma

Sei (S,Σ) eine Signatur, C eine Menge von Clusters und OP_C die Menge aller in den Clustern von C öffentlichen Operationen. Es gelte Erf(C,S,Σ).
P^X_f, f∈Σ, sei wie in 3.2.1.2 - 3.2.1.4.
Dann gilt: $(\forall f\in\Sigma)$ P^X_f ist wohldefiniert.

Beweis:

Sei $P_f\in OP_C$.

1. P_f: $\longrightarrow A'_s$ ist Create ohne Parameter.
   Sei $\rho\in A^X_\perp\downarrow 2$. $P^X_f((\perp,\rho))=(Newloc(\rho),\rho')$.
   Dabei ist $\rho'=\lambda\ loc.loc=Newloc(\rho) \longrightarrow \mathcal{C}[\![A'_s]\!]|\rho(loc)$.
   Newloc ist injektiv; $\mathcal{C}$ und ρ sind Funtkionen
   ==> (Newloc(ρ),ρ') ist eindeutig ==> P^X_f wohldefiniert.

2. P_f: $A'_{s1} x \ldots x A'_{sn} \longrightarrow A'_s$ ist Prozedur.
   Sei $((a_{s1},\rho),\ldots,(a_{sn},\rho)),((a'_{s1},\rho'),\ldots,(a'_{sn},\rho')) \in A^X_{s1} x \ldots x A^X_{sn}$ und
   $((a_{s1},\rho),\ldots(a_{sn},\rho))=((a'_{s1},\rho'),\ldots,(a'_{sn},\rho'))$
   $\Longrightarrow (\beta,\sigma):=\rho(a_{sn})=\rho'(a'_{sn})=:(\beta',\sigma')$, da ρ und ρ' Funktionen.
   β,rep,β' sind voraussetzungsgemäß Funktionen
   $\Longrightarrow (\bar{\rho},obj):=\beta(\sigma,P_f,(a_{s1},\ldots,a_{sn-1},rep(\rho(a_{sn}))),\rho)=$
   $=\beta'(\sigma',P_f,(a'_{s1},\ldots a'_{sn-1},rep(\rho'(a'_{sn}))),\rho')=:(\bar{\rho}',obj')$
   (dabei ist $obj=obj'=\underline{\perp}$)
   $\Longrightarrow P^X_f((a_{s1},\rho),\ldots,(a_{sn},\rho))=(a_{sn},\rho)=P^X_f((a'_{si},\rho'),\ldots,(a'_{sn},\rho'))=(a'_{sn},\bar{\rho}')$
   $\Longrightarrow P^X_f$ ist wohldefiniert.

3. P_f: $A'_{s1} x \ldots x A'_{sn} \longrightarrow A'_s$ ist Create mit Parametern.
   Vgl. 1. und verwende: $\mathcal{C}[\![A'_s]\!]$: $A'_{s1} x \ldots x A'_{sn} \longrightarrow A'_s$ ist Funktion.

4. P_f: $A'_{s1} x \ldots x A'_{sn} \longrightarrow A'_s$ ist Funktionsprozedur $\neq$ Create.
   Analog 2.

MMM

3.2.2 Induktive Definition der Modulalgebra M_C für CSSA

Sei (S,Σ) eine Signatur, C eine Menge von Scripts und OP_C die Menge aller in den Scripts von C öffentlichen Operationen. Es gelte $Erf(C,S,\Sigma,\alpha,\gamma)$ für ein α und γ.
$P_f \in OP_C$ sei eine Funktion eines Scripts aus C mit $\gamma(P_f)=f \in \Sigma$.

3.2.2.1 Sei $\rho_0 \in GEnv$; $\perp \notin S$.
$A^X_\perp := \{(\perp,\rho_0)\}$ ρ_0 heißt die *Standardumgebung*.
$(\forall s \in S)\ A^X_s := \emptyset$

$(\forall P_f \in OP_C)$

3.2.2.2 P_f ist Create ohne Parameter

dann: $(\exists s \in S)\ P_f$: $\longrightarrow A'_s$

Definiere:

$$P^X_f\colon A^X_\perp \longrightarrow A^X_s$$

$(\forall \rho \in A^{X}_{\perp} \downarrow 2)$
$[a_s := \text{Newloc}(\rho)$
$\rho' := \lambda\ loc.loc = a_s \longrightarrow \mathcal{I}[\![A'_s]\!] \mid \rho(loc)$
$A^{X}_{\perp} := A^{X}_{\perp} \cup \{(\perp, \rho')\}$
$A^{X}_{s} := A^{X}_{s} \cup \{(a_s, \rho')\}$
$(\forall s' \in S)\ A^{X}_{s'} := A^{X}_{s'} \cup \{(l, \rho') : l \in \text{dom}(\rho') \cap A^{X}_{s'} \downarrow 1\}$
$P^{X}_{f}((\perp, \rho)) := (a_s, \rho')]$

3.2.2.3 P_f ist Create mit Parametern

dann $(\exists n \in N)(\exists s1, \ldots, sn, s \in S)\ P_f : A'_{s1} \times \ldots \times A'_{sn} \longrightarrow A'_s$
Definiere:

$P^{X}_{f} : A^{X}_{s1} \times \ldots \times A^{X}_{sn} \longrightarrow A^{X}_{s}$
$(\forall ((a1, \rho), \ldots, (an, \rho)) \in A^{X}_{s1} \times \ldots \times A^{X}_{sn})$
$[a_s := \text{Newloc}(\rho)$
$\rho' := \lambda\ loc.loc = a_s \longrightarrow \mathcal{I}[\![A'_s]\!](a1, \ldots, an) \mid \rho(loc)$
$A^{X}_{\perp} := A^{X}_{\perp} \cup \{(\perp, \rho')\}$
$A^{X}_{s} := A^{X}_{s} \cup \{(a_s, \rho')\}$
$(\forall s' \in S)\ A^{X}_{s'} := A^{X}_{s'} \cup \{(l, \rho') : l \in \text{dom}(\rho') \cap A^{X}_{s'} \downarrow 1\}$
$P^{X}_{f}((a1, \rho), \ldots, (an, \rho)) := (a_s, \rho')]$

3.2.2.4 P_f Ist Operation ≠ Create

In CSSA gibt es keine Unterscheidung zwischen Prozeduren und Funktionsprozeduren.
Ein weiterer Unterschied zu CLALPHARD ist, daß die Weitergabe der Kontrolle explizit programmiert werden muß und daß es i.a. keine interne Realisierung einer Datenstruktur gibt (vgl. 2.1).
Ein Operationsaufruf wird durch ein Ereignis E initiiert. Das Ergebnis dieser Operation ist Bestandteil der Nachricht des Folgeereignisses E'= Follow(E). Per Konvention sei dieses Ergebnis das erste Element der Acquaintance- bzw. Locationliste der Nachricht E'.

$(\exists n\in N)(\exists s1,\ldots,sn,s\in S)\ P_f\colon A'_{s1} \times\ldots\times A'_{sn} \longrightarrow A'_s$

Definiere:

$P^x_f\colon A^x_{s1} \times\ldots\times A^x_{sn} \longrightarrow A^x_s$

$(\forall((a1,\rho),\ldots,(an,\rho))\in A^x_{s1} \times\ldots\times A^x_{sn}$ mit

$((a1,\ldots,an-1),P_f),an,\rho)\in dom(Follow)$ bei geeigneter Indizierung)

$[(mess,l',\rho'):=Follow((((a1,\ldots,an-1),P_f,an,\rho))$

$a_s:=(mess\downarrow 1)_1$

$A^x_\perp:=A^x_\perp \cup \{(\perp,\rho')\}$

$A^x_s:=A^x_s \cup \{(a_s,\rho')\}$

$(\forall s'\in S)\ A^x_{s'}:=A^x_{s'} \cup \{(l,\rho')\colon l\in dom(\rho') \cap A^x_{s'}\downarrow 1\}$

$P^x_f((a1,\rho),\ldots,(an,\rho)):=(a_s,\rho')]$

Zur Funktionalität von P_f vgl. 3.1.3.1.
Zur impliziten Änderung der globalen Umgebung vgl. 2.1.2.

3.2.2.5 Lemma

Sei (S,Σ) eine Signatur, C eine Menge von Scripts und OP_C die Menge aller in den Scripts von C öffentlichen Operationen. Es gelte $Erf(C,S,\Sigma)$.
P^x_f, $f\in\Sigma$, sei wie in 3.2.2.2 - 3.2.2.4.
Dann gilt: $(\forall f\in\Sigma)\ P^x_f$ ist wohldefiniert.

Beweis:

Sei $P_f\in OP_C$.
1. $P_f\colon \longrightarrow A'_s$ ist Create ohne Parameter.
Sei $\rho\in A^x_\perp\downarrow 2$, $P^x_f((\perp,\rho))=(Newloc(\rho),\rho')$.
Dabei ist $\rho'=\lambda\ loc.loc=Newloc(\rho) \longrightarrow \mathfrak{I}[\![A'_s]\!]|\rho(loc)$.
Newloc ist injektiv; $\mathfrak{I}$ und ρ sind voraussetzungsgemäß Funktionen
$\Rightarrow (Newloc(\rho),\rho')$ ist eindeutig $\Rightarrow$ P^x_f wohldefiniert.

2. $P_f\colon A'_{s1} \times\ldots\times A'_{sn} \longrightarrow A'_s$ ist Create mit Parametern.
Analog 1. unter Verwendung der Eigenschaft, daß $\mathfrak{I}[\![A'_s]\!] : A'_{s1}\times\ldots\times A'_{sn} \longrightarrow A'_s$ Funktion ist.

3. $P_f: A'_{s1} \times \ldots \times A'_{sn} \longrightarrow A'_s$ ist Funktionsprozedur $\neq$ Create.
   Sei $((a1,\rho),\ldots,(an,\rho))$, $((b1,\rho'),\ldots,(bn,\rho'))\in A'_{s1}\times\ldots\times A'_{sn}$ mit
   $((a1,\rho),\ldots,(an,\rho)) = ((b1,\rho'),\ldots,(bn,\rho'))$.

   Follow ist nach Voraussetzung Funktion
   $\Longrightarrow$ Follow $(((a1,\ldots,an-1),P_f),an,\rho)$ = Follow $(((b1,\ldots,bn-1),P_f),bn,\rho')$
   $\Longrightarrow$ $P_f^X((a1,\rho),\ldots,(an,\rho)) = P_f^X((b1,\rho'),\ldots,(bn,\rho'))$
   == P_f^X ist wohldefiniert.

MMM

Für P_f^X, $f\in\Sigma$, und A_s^X, $s\in S$, definiert wie in 3.2.1 bzw. 3.2.2 gilt nun, daß P_f^X im allgemeinen nur eine partielle Funktion ist.
Es wird daher jede Menge A_s^X, $s\in S$, um ein (neues) Fehlerelement $\perp_s$ erweitert und die Definition von P_f^X, $f\in\Sigma$, so korrigiert, daß jedes P_f^X eine totale Funktion ist.

3.2.3 Definition (Modulalgebra)

Sei (S,Σ) eine Signatur, C eine Menge von Cluster /Scripts mit $Erf(C,S,\Sigma)$.
Sei $\perp\notin S$.
Seien A_s^X, $s\in S$, und P_f^X, $f\in\Sigma$, wie in 3.2.1 bzw. 3.2.2.

1. $(\forall s\in S)\ A_s := A_s^X \cup \{\perp_s\}$, wobei $\perp_s\notin A_s^X$.
2. $A_\perp := A_\perp^X$ (vgl. 3.2.1.1 bzw. 3.2.2.1)
3. $(\forall s\in S)(\forall f\in\Sigma_{\varepsilon,s})$
   $\overline{P}_f: A_\perp \longrightarrow A_s$
   $(\forall(\perp,\rho)\in A_\perp)\ \overline{P}_f((\perp,\rho)) := P_f^X((\perp,\rho))$ gemäß 3.2.1.2 bzw. 3.2.2.2
4. $(\forall n\in N)(\forall s1,\ldots,sn,s\in S)(\forall f\in\Sigma_{s1\ldots sn,s})$
   $\overline{P}_f: A_{s1}\times\ldots\times A_{sn}\longrightarrow A_s$
   $(\forall((a1,\rho),\ldots,(an,\rho))\in A_{s1}\times\ldots A_{sn})$

$$\overline{P}_f((a1,\rho),\ldots,(an,\rho)) := \begin{cases} P_f^X((a1,\rho),\ldots,(an,\rho)) & \text{gemäß 3.2.1.3/3.2.1.4 bzw. 3.2.2.3/3.2.2.4} \\ & \text{falls } ((a1,\rho),\ldots,(an,\rho))\in dom(P_f^X) \\ \perp_s & \text{sonst} \end{cases}$$

5. $M_C := (\{A_s : s \in S \cup \{\bot\}\}, \{\overline{P}_f : f \in \Sigma\})$ heißt die von C erzeugte *Modulalgebra*.

$|M_C| := \bigcup_{s \in S} A_s$.

Nach Definition 3.2.3.3 ist $\overline{P}_f : A_{s1} \times \ldots \times A_{sn} \longrightarrow A_s$ insbesondere streng bzgl. den Fehlerelementen $\bot_s$, $s \in S$, d.h. $\overline{P}_f(e1,\ldots,en) = \bot_s$, falls $e_i = \bot_{si}$ für ein $i \in \{1,\ldots,n\}$.
Diese Regelung entspricht den Fehlermeldungen eines Programm(laufs).

Der Definition der Modulalgebra kann man entnehmen, daß mit $|Loc| = \infty$ auch $|\bigcup_{s \in S} A_s| = \infty$ ist, falls es mindestens ein $s \in S$ gibt mit $\Sigma_{\varepsilon,s} \neq \emptyset$, was vorausgesetzt wurde. Sei nämlich $s \in S$ mit $f \in \Sigma_{\varepsilon,s}$. Dann können, ausgehend von $(\bot, \rho_0)$ durch wiederholte Anwendung von $\overline{P}_f$ unendlich viele Elemente in $A_\bot$ und A_s erzeugt werden.
Diese Eigenschaft, daß in jeder Umgebung ein Element durch $\overline{P}_f$ erzeugt werden kann, erschwert die Erörterung eines konkreten Beispiels einer Modulalgebra.

Wünschenswert wäre nun, daß M_C eine Σ-Algebra ist.
Dies ist aber nicht möglich, da es in M_C i.a. keine konstante Funktionen gibt und die Menge der Sorten um "$\bot$" erweitert wurde. Allerdings ist nach der Definition von M_C zu vermuten, daß M_C eine algebraische Struktur hat, die in eindeutiger Weise von der Signatur (S,Σ) und $\bot$ abhängt.
Dies legt folgende Begriffsbildung nahe.

3.2.4 Definition $((S,\Sigma)_g)$

Sei (S,Σ) eine Signatur und $g \notin S$.
$(S',\Sigma') := (S,\Sigma)_g$:gdw

1. $S' = S \cup \{g\}$
2. $\Sigma' = \{\Sigma'_{w,s} : w \in S^*, s \in S\} \cup \{\Sigma'_{g,s} : s \in S\}$ mit

$(\forall s \in S)(\exists fun_s : \Sigma_{\varepsilon,s} \longrightarrow \Sigma'_{g,s}$ bijektiv) $\wedge$

$(\forall s \in S)(\forall w \in S^+)(\exists fun_{w,s} : \Sigma_{w,s} \longrightarrow \Sigma'_{w,s}$ bijektiv) $\wedge$

$(\forall s \in S)(\forall w \in S^*)(\Sigma_{w,s} = \emptyset \iff \Sigma'_{w,s} = \emptyset)$

3.2.5 Corollar

Sei (S,Σ) eine Signatur, $g \notin S$ und $(S',\Sigma')=(S,\Sigma)_g$.

Dann ist (S',Σ') eine Signatur.

Beweis: Da (S,Σ) eine Signatur ist, folgt die Behauptung unmittelbar nach 3.2.4 und 1.1.1.1.

MMM

Nach Definition hängt $(S,\Sigma)_g$ in eindeutiger Weise von (S,Σ) und g ab, so daß weiter unten einige strukturelle Induktionsbeweise anstatt über $(S,\Sigma)_g$ auch über (S,Σ) geführt werden können.

3.2.6 Satz

Sei (S,Σ) eine Signatur, C eine Menge von Cluster /Scripts mit $Erf(C,S,\Sigma,\alpha,\gamma)$ für ein α und γ und M_C die von C erzeugte Modulalgebra. $\perp \notin S$.

Dann ist M_C eine Σ'-Algebra mit $(S',\Sigma'):=(S,\Sigma)_\perp$.

Beweis:

1. Nach 3.2.3.5 existieren für alle $s \in S'=S \cup \{\perp\}$ eine Trägermenge A_s.
2. $(\forall s \in S)\Sigma'_{\perp,s} := \{f' : f \in \Sigma_{\varepsilon,s}\}$.
   $(\forall s \in S)(\exists fun_s : \Sigma_{\varepsilon,s} \longrightarrow \Sigma'_{\perp,s}, (\forall f \in \Sigma_{\varepsilon,s})\ fun_s(f):=f')$
   $(\forall s \in S)(\forall w \in S^+)\ \Sigma'_{w,s} := \Sigma_{w,s}$
   $(\forall s \in S)(\forall w \in S^+)(fun_{w,s} := id_{\Sigma_{w,s}})$

2.1 Sei $s \in S'$, $f' \in \Sigma'_{\perp,s}$
   $\Longrightarrow (s \in S \wedge (\exists f \in \Sigma_{\varepsilon,s})\ fun_s^{-1}(f')=f)$
   $\Longrightarrow ((\exists P_f \in OP_C)\gamma(P_f)=f \wedge P_f : \longrightarrow A'_s)$
   $\Longrightarrow (\exists \overline{P}_f \in M_C \downarrow 2)\ \overline{P}_f : \longrightarrow A_s$ nach 3.2.3.3.

2.2 Seien $w=s1\ s2 \ldots sn \in S'^+$, $s \in S'$, $f' \in \Sigma'_{w,s}$.
   $\Longrightarrow (w \in S^+, s \in S) \wedge ((\exists f \in \Sigma_{w,s})\ fun_{w,s}^{-1}(f')=f)\ (f=f'!)$
   $\Longrightarrow (\exists P_f \in OP_C)\ \gamma(P_f)=f \wedge P_f : A'_{s1} \times \ldots \times A'_{sn} \longrightarrow A'_s$
   $\Longrightarrow (\exists \overline{P}_f \in M_C \downarrow 2)\ \overline{P}_f : A_{s1} \times \ldots \times A_{sn} \longrightarrow A_s$ nach 3.2.3.4.

$\Longrightarrow M_C$ ist eine Σ'-Algebra.

MMM

Ein kurzes, informelles Beispiel soll die Definition einer Modulalgebra veranschaulichen. Informell bleibt das Beispiel deshalb, weil die genaue Konstruktion einer Modulalgebra die Kenntnis der semantischen Funktionen voraussetzt. Diese können aber wegen des hier angesetzten Abstraktionsniveaus bei der Beschreibung der Beispielsprachen nicht explizit und vollständig angegeben werden. Dennoch soll zumindest der Mechanismus bei der Konstruktion einer Modulalgebra verdeutlicht werden.

Als Grundlage soll Beispiel 3.1.7 (INTSTACK) dienen.

Die Standardumgebung sei $\rho_0 := \emptyset$.
Die Menge der Locations sei $Loc := \{l_i : i \in N\}$.

Ausgehend von der Standardumgebung soll eine Umgebung konstruiert werden, die einen Keller der Länge 2 mit den Komponenten '0' und '1' enthält.

In ρ_0 wird Createstack aufgerufen; dies führt zu einer Umgebung
$\rho_1 := \{(l_1, m1)\}$ mit
$l_1 := Newloc(\rho_0)$, $m1 := \mathcal{C}[\![STACK]\!]$.
Dem entspricht in der Modulalgebra: $\overline{Createstack}((\perp, \rho_0)) =: (l_1, \rho_1)$;
$A_{stack} := \{(l_1, \rho_1)\}$.
(Eine Abbildung $\rho \in GEnv$ wird hier als zweistellige Relation geschrieben.)

In ρ_1 wird nun eine Null erzeugt, was zu ρ_2 führt:
$\rho_2 := \{(l_1, m1), (l_2, m2)\}$ mit
$l_2 := Newloc(\rho_1)$, $m2 := \mathcal{C}[\![INTEGER]\!]$;
$(l_2, \rho_2) := \overline{Null}((\perp, \rho_1))$;
$A_{integer} := \{(l_2, \rho_2)\}$; $A_{stack} := \{(l_1, \rho_1), (l_1, \rho_2)\}$.
Dabei wurde A_{stack} durch das Lifting um (l_1, ρ_2) erweitert.

Nun wird '1' erzeugt durch einen Aufruf
$\overline{Succ}((l_2, \rho_2)) =: (l_3, \rho_3)$ wobei $\rho_3(l_3) =: m3$ der Modul für '1' sein soll.
Damit gilt nun:
$\rho_3 := \{(l_1, m1), (l_2, m2), (l_3, m3)\}$,
$A_{stack} := \{(l_1, \rho_1), (l_1, \rho_2), (l_1, \rho_3)\}$, $A_{integer} := \{(l_2, \rho_2), (l_2, \rho_3), (l_3, \rho_3)\}$.
Nun wird zweimal ein Push ausgeführt, wobei zuerst '0' und dann '1' auf den Keller geschrieben wird:
$\overline{Push}((l_1, \rho_3), (l_2, \rho_3)) =: (l_1, \rho_4)$.
$\overline{Push}$ bewirkt einen Seiteneffekt auf dem Modul m1, der dadurch in m1' übergeht.

ρ_4 := $\{(l_1,m1'),(l_2,m2),(l_3,m3)\}$,
A_{stack}:= $\{(l_1,\rho_1),(l_1,\rho_2),(l_1,\rho_3),(l_1,\rho_4)\}$,
$A_{integer}$:= $\{(l_2,\rho_2),(l_2,\rho_3),(l_3,\rho_3),(l_2,\rho_4),(l_3,\rho_4)\}$.

Eine weitere Anwendung von $\overline{Push}$ soll '1' auf den Keller schreiben, der bisher mit '0' gefüllt ist:

$\overline{Push}((l_1,\rho_4),(l_3,\rho_4))$ =: (l_1,ρ_5),

dabei geht m1' durch Seiteneffekte über in m1" (beachte, daß sich vereinbarungsgemäß dadurch die Umgebung ändert, so daß $\rho_5(l_1)$=m1").

ρ_5 := $\{(l_1,m1"),(l_2,m2),(l_3,m3)\}$,
A_{stack}:= $\{(l_1,\rho_1),(l_1,\rho_2),(l_1,\rho_3),(l_1,\rho_4),(l_1,\rho_5)\}$,
$A_{integer}$:= $\{(l_2,\rho_2),(l_2,\rho_3),(l_2,\rho_4),(l_2,\rho_5),(l_3,\rho_3),(l_3,\rho_4),(l_3,\rho_5)\}$.

Anhand einer Modulalgebra bzw. deren Definition kann nun rückblickend die Semantik der Beispielsprachen noch etwas beleuchtet werden. Dabei werden im folgenden zwei Forderungen an diese Semantik entwickelt, die die Eigenschaften von Axiomen haben.

3.2.7 Definition $(Pm(Loc,\rho_0),\pi\sigma,\pi\rho)$

Sei C eine Menge von Cluster /Scripts und M_C die von C erzeugte Modulalgebra, deren Standardumgebung ρ_0 sei.

1. $Pm(Loc,\rho_0)$:= $\{\pi$: π: Loc $\longrightarrow$ Loc bijektiv $\wedge$ $\pi|_{dom(\rho_0)} = id_{dom(\rho_0)}\}$.
2. $(\forall\pi ePm(Loc,\rho_0))(\forall\sigma eLState$: σ: Id $\longrightarrow$ Loc)
   $\pi\sigma$: Id $\longrightarrow$ Loc
   $(\forall ideId)(\pi\sigma(id)$:= $\pi(loc)$:gdw $\sigma(id) = loc)$
3. $(\forall\pi ePm(Loc,\rho_0))(\forall\rho eGEnv$: ρ: Loc $\longrightarrow$ Modul)
   $\pi\rho$: Loc $\longrightarrow$ Modul
   $(\forall loceLoc)(\pi\rho(loc)$:= (β,σ) :gdw $\rho(\pi^{-1}(loc)) = (\beta,\pi^{-1}\sigma))$

Mit σ und ρ sind auch $\pi\sigma$ bzw. $\pi\rho$, $\pi ePm(Loc,\rho_0)$, wohldefiniert, da π bijektiv ist.

3.2.8 Corollar

Sei C eine Menge von Clusters/Scripts und M_C die von C erzeugte Modulalgebra mit der Standardumgebung ρ_0.
Dann gilt: $(\forall\pi ePm(Loc,\rho_0))(\forall leLoc)(\forall\rho eGEnv)$

1. $\pi(\rho(l)\downarrow 2) = (\pi\rho(\pi(l)))\downarrow 2$
2. $(\pi\rho(\pi(l)))\downarrow 1 = \rho(l)\downarrow 1$

Beweis:

Sei $\pi \in Pm(Loc,\rho_0)$, $l \in Loc$, $\rho \in GEnv$ und $\rho(l)=:(\beta,\sigma)$.
$\rho(l)=(\beta,\sigma)$ ==> $\pi(\rho(l)\downarrow 2) = \pi\sigma$.
Sei $\pi\rho(\pi(l)) =: (\beta',\sigma')$ ==> $\rho(\pi^{-1}(\pi(l))) = (\beta',\pi^{-1}\sigma')$;
$\rho(\pi^{-1}(\pi(l))) = \rho(l) = (\beta,\sigma)$ ==> $\beta=\beta' \wedge \sigma=\pi^{-1}\sigma'$
==> $\beta=\beta' \wedge \sigma'=\pi\sigma$.

MMM

Die Locations, die in den Beispielsprachen verwendet werden, dienen als Namen für Moduln (vgl. Speicheradresse - Speicherzelleninhalt).
Die konkrete Beschaffenheit dieser Namen ist irrelevant. Man wird also erwarten, daß die Semantik der Beispielsprachen "unabhängig" ist von den jeweils verwendeten Locations. Über die Menge Loc wurde in Kapitel 2 demnach auch nur die Angabe gemacht, abzählbar unendlich viele Elemente zu haben.

Von den Funktionen, die die Semantik einer Beispielsprache definieren, wird man daher annehmen, daß sie invariant sind gegenüber einer Permutation der Locations. Dies veranlaßt zu folgender Forderung an die Semantik der Beispielsprachen:

3.2.9 Forderung

Für jede Funktion F der Semantik einer Beispielsprache und für jede Permutation π der Locations gilt:

F und π sind vertauschbar,

d.h. die Anwendung von F auf ein beliebiges Argument, in dem jede Location $l \in Loc$ ersetzt wurde durch $\pi(l)$, ergibt dasselbe, wie wenn man F auf das ursprüngliche Argument anwendet und in dem Ergebnis von F dann jede Location durch ihr Bild unter π ersetzt.

Diese Forderung trifft insbesondere zu auf das Verhalten β eines Moduls (β,σ), da β über die Funktionen der Semantik der zugrundeliegenden Beispielsprache definiert wurde. Außerdem wurden die Operationen $\overline{P}_f$ einer Modulalgebra über das Verhalten und die semantischen Funktionen bestimmt.
Auf der Ebene der Modulalgebren wirkt sich daher obige Forderung folgendermaßen aus:

Sei $M_C:=(\{A_s: s\in S\cup\{\perp\}\},\{\overline{P}_f: f\in\Sigma\})$ eine Modulalgebra der Cluster-/Scriptmenge C mit der Standardumgebung ρ_0.
Sei $\overline{P}_f\in M_C\downarrow 2$, $((l1,\rho),\ldots,(ln,\rho))\in dom(\overline{P}_f)$ und $(l,\rho'):=\overline{P}_f((l1,\rho),..,(ln,\rho))$.
Sei $\pi\in Pm(Loc,\rho_0)$.
Dann gilt nach 3.2.9:

$$\overline{P}_f((\pi(l1),\pi\rho),\ldots,(\pi(ln),\pi\rho)) = (\pi(l),\pi\rho').$$

Der Definition einer Modulalgebra kann man entnehmen, daß eine Umgebung $\rho\in A_\perp\downarrow 2$ ausgehend von der Standardumgebung ρ_0 durch eine Folge von Operationen $\overline{P}_f$, $f\in\Sigma$, erzeugt wurde, wobei natürlich das erste Element dieser Operationenfolge ein Create ohne Parameter sein muß.
Diese Ketten von Operationen werden noch eine besondere Rolle spielen.

3.2.10 Definition (Konkatenation, Operationskette, OP_{M_C})

a) Sei A eine nicht leere Menge und A^* die von A erzeugte freie Worthalbgruppe mit dem leeren Wort ε.
Dann sei "_" das Zeichen für die Konkatenation, d.h. $(A^*,\_,\varepsilon)$ ist ein Monoid.

b) Sei (S,Σ) eine Signatur, C eine Menge von Cluster /Scripts mit $Erf(C,S,\Sigma)$ und $M_C:=(\{A_s: s\in S\cup\{\perp\}\},\{\overline{P}_f: f\in\Sigma\})$ die von C erzeugte Modulalgebra mit der Standardumgebung ρ_0.

1. $(\forall OP\in(M_C\downarrow 2)^*)$ definiere partielle Funktion

$$OP: A_\perp\downarrow 2 \longrightarrow POT(A_\perp\downarrow 2)$$

($POT(A_\perp\downarrow 2)$ sei die Potenzmenge von $A_\perp\downarrow 2$)

durch:

a) $OP=\varepsilon$: $(\forall\rho\in A_\perp\downarrow 2)\ OP(\rho) := \{\rho\}$

b) $OP=\overline{P}_f$, $f\in\Sigma_{\varepsilon,s}$, $s\in S$: $(\forall\rho\in A_\perp\downarrow 2)\ OP(\rho) := \{(\overline{P}_f((\perp,\rho))\downarrow 2)\}$

c) $OP=\overline{P}_f\_OP'$, $f\in\Sigma_{s1\ldots sn,s}$, $n\in N_0$, $s1,\ldots,sn,s\in S$, $OP'\in (M_C\downarrow 2)^*$:

$$(\forall\rho\in A_\perp\downarrow 2)\ OP(\rho) := \{\rho'': \exists\rho''\in OP'(\rho)\,\exists((l1,\rho''),\ldots,(ln,\rho''))\in dom(\overline{P}_f) \wedge li\notin dom(\rho),\ i=1,\ldots,n \wedge \rho'=(\overline{P}_f((l1,\rho''),\ldots,(ln,\rho'')))\downarrow 2\}$$

2. $OP_{M_C} := \{OP\in(M_C\downarrow 2)^*$: OP totale Funktion auf $A_\perp\downarrow 2 \wedge ((\forall\rho\in A_\perp\downarrow 2)\ |OP(\rho)|=1)\}$

$OP\in OP_{M_C}$ heißt *Operationskette*.

Für $OP\in OP_{M_C}$ und $\rho\in A_\perp\downarrow 2$ identifiziere dann $\{\rho'\}:= OP(\rho)$ mit ρ'.

Eine Modulalgebra M_C bzw. eine Menge C von Cluster /Scripts enthalte die Programmierung eines DATA-Kellers und einer DATA'-Warteschlange. $\rho 1$ sei die Umgebung, die man ausgehend von der Standardumgebung erhält durch die Erzeugung bzw. Konstruktion eines Kellers mit n Kellerelementen d1,...,dn vom Typ DATA.
In dieser Umgebung werde nun eine Warteschlange der Länge m mit all ihren Komponenten erzeugt bzw. konstruiert, was zu einer Umgebung $\rho 2$ führe. An diesem Prozeß sind der Keller und seine Komponenten nicht beteiligt. Man wird also erwarten, daß die umgekehrte Anwendung der Operationsketten für den Keller und die Warteschlange, d.h. zuerst wird die Warteschlange ausgehend von der Standardumgebung konstruiert und dann erst der Keller, zu einer Umgebung $\rho 2'$ führt, die bis auf eine Permutation der Locations (Umbenennung) zu $\rho 2$ gleich ist.
Eine Verallgemeinerung dieser Überlegung stellt die folgende zweite Forderung dar, die sich ebenfalls an die Semantik der Beispielsprachen richtet.

3.2.11 Forderung

Sei M_C eine Modulalgebra mit $\rho_0 \in A_{\perp}\downarrow 2$ als Standardumgebung.
Sei $n \in N_0$ und $OP_i \in OP_{M_C}$, $i=1,\ldots,n$; α und β seien Permutationen der Zahlen $1,\ldots,n$. $OP := OP_{\alpha(1)} \_ \ldots \_ OP_{\alpha(n)}$, $OP' := OP_{\beta(1)} \_ \ldots \_ OP_{\beta(n)}$.
Dann gilt für alle $\rho \in A_{\perp}\downarrow 2$:

$\exists \pi \in Pm(Loc, \rho_0)$, so daß

1. $\pi OP(\rho) = OP'(\rho)$ und
2. $\pi|_{dom(\rho)} = id_{dom(\rho)}$.

3.2.11 besagt also, daß die einzelnen Operationsketten $OP_1,\ldots,OP_n$ voneinander und von den Elementen der Umgebung ρ unabhängig sind. OP und OP' angewendet auf eine Umgebung $\rho \in A_{\perp}\downarrow 2$ ergeben damit zwei Umgebungen, die bis auf eine Permutation (Umbenennung) der Locations gleich sind und die Umgebung ρ "unverfälscht" enthalten, d.h. $\pi|_{dom(\rho)} = id_{dom(\rho)}$.

3.3 Die Auswertungsfunktion EVAL

Die Konstruktion der Modulalgebra M_C zu einer gegebenen Signatur (S,Σ) in 3.2 orientierte sich an der Struktur der Terme $te T_\Sigma$. Im folgenden wird nun eine Zuordnung zwischen den Termen der Termalgebra und den Elementen der Modulalgebra vorgenommen. Diese Zuordnung bildet eine Brücke zwischen der Term- und der Modulalgebra.

3.3.1 Definition (EV, EVAL)

Sei (S,Σ) eine Signatur, C eine Menge von CLuster /Scripts, für die $Erf(C,S,\Sigma)$ gilt. $M_C := (\{A_s : se S \cup \{\bot\}\}, \{\overline{P}_f : fe\Sigma\})$ sei die von C erzeugte Modulalgebra.

3.3.1.1 $EV := \{EV_s : EV_s : T_\Sigma^* \times T_{\Sigma,s} \longrightarrow A_s,\ seS\}$

1. $(\forall seS)(\forall fe\Sigma_{\varepsilon,s})$

   1.1 $EV(\varepsilon,f) := \overline{P}_f((\bot,\rho_0))$ (ρ_0 Standardumgebung)

   1.2 $(\forall neN)(\forall t1,\ldots,tne T_\Sigma)$

   $EV(t1\ldots tn,f) := \overline{P}_f((\bot,\rho))$ mit $\rho := EV(t1\ldots tn-1,tn)\downarrow 2$

2. $(\forall neN)(\forall s1,\ldots,sn,seS)(\forall fe\Sigma_{s1\ldots sn,s})(\forall t1,\ldots,tne T_\Sigma)$

   2.1 $EV(\varepsilon,f(t1,\ldots,tn)) := \overline{P}_f((a1,\rho),\ldots,(an,\rho))$

   mit $(a1,\rho_1) := EV(\varepsilon,t1)$

   $(a2,\rho_2) := EV(t1,t2)$

   $\vdots$

   $(an,\rho_n) := EV(t1\ldots tn-1,tn)$

   $\rho := \rho_n$

   2.2 $(\forall meN)(\forall t'_1,\ldots,t'_m e T_\Sigma)$

   $EV(t'_1\ldots t'_m,f(t1,\ldots,tn)) := \overline{P}_f((a1.\rho),\ldots,(an,\rho))$

   mit $(a1,\rho_1) := EV(t'_1\ldots t'_m,t1)$

   $(a2,\rho_2) := EV(t'_1\ldots t'_m t1,t2)$

   $\vdots$

   $(an,\rho_n) := EV(t'_1\ldots t'_m t_1\ldots tn-1,tn)$

   $\rho := \rho_n$

3.3.1.2 $EVAL := \{EVAL_s : EVAL_s : T_{\Sigma,s} \longrightarrow A_s,\ seS\}$

$EVAL : T_\Sigma \longrightarrow M_C$

$(\forall te T_\Sigma)\ EVAL(t) := EV(\varepsilon,t)$

EVAL bzw. EV sind wohldefiniert, da $\overline{P}_f$, $f \in \Sigma$, wohldefiniert.

EVAL ordnet einem Term $t \in T_\Sigma$ ein Paar $(1,\rho)$ zu: t kann dabei aufgefaßt werden als abstrakte Beschreibung einer Programmberechnung. Dabei ergibt sich die Schwierigkeit, daß in einer Berechnung die Teilterme von t sequentiell erarbeitet werden, wodurch schrittweise eine globale Umgebung - ausgehend von der Standardumgebung ρ_0 - aufgebaut wird.
Die Reihenfolge, in der die Teilterme eines Terms t=f(t1,...,tn) erarbeitet werden, ist beliebig. Hier wird eine Abarbeitung der Argumente von f von links nach rechts unterstellt. Die bereits abgearbeiteten Teilterme, der Kontext, bestimmen die gegenwärtige globale Umgebung. Der Kontext wird im ersten Argument der Hilfsfunktion EV gespeichert.

Dem Übergang von (ai,ρ_i) nach $(ai+1,\rho)$, $1 \leq i \leq n-1$, $\rho=\rho_n$, in 3.3.1.1.2 liegt das Lifting zugrunde:

a) $t_{i+1} \in \Sigma_{\varepsilon,s}$, $s \in S$:

$(ai+1,\rho_{i+1}) = EV(t1...ti-1ti,ti+1) = \overline{P}_f((\perp,\rho_i))$
nach 3.3.1.1.1.2
Sei $(ai,\rho_i) = EV(t1...ti-1,ti) \in A_{si}$ für $si \in S$
$\Longrightarrow (ai,\rho_{i+1}) \in A_{si}$ per Lifting
$\Longrightarrow (ai,\rho) \in A_{si}$ per Lifting.

b) $t_{i+1}=f(t1',...,tm')$:

$(ai+1,\rho_{i+1}) = EV(t1...tn,f(t1',...,tm')) = \overline{P}_f((b1,\rho')..,(bm,\rho'))$
und $(ai,\rho') \in A_{si}$
$\Longrightarrow (ai,\rho_{i+1}) \in A_{si}$ per Lifting
$\Longrightarrow (ai,\rho) \in A_{si}$ per Lifting.

Daran sieht man, daß EVAL bzw. EV nur dann definiert ist, wenn auch jeweils das Lifting in geeigneter Weise durchführbar ist, wenn also $\overline{P}_f((a1,\rho),...,(an,\rho))$ in 3.3.1.1.2 definiert und $\neq \perp_s$ ist.
Dies hängt ab von der Programmierung der Cluster/Scripts. Daher ist es sinnvoll, an den Programmierer die Forderung zu stellen, daß es zu jedem Term $t \in T_\Sigma$ eine entsprechende Folge von definierten Operationsaufrufen in M_C gibt.

3.3.2 Definition (M_C enthält Σ)
Sei (S,Σ) eine Signatur, C eine Menge von CLuster /Scripts mit $Erf(C,S,\Sigma)$ und M_C die von C erzeugte Modulalgebra. Sei EVAL: $T_\Sigma \longrightarrow M_C$ wie in 3.3.1.
M_C *enthält* Σ :gdw $(\forall s \in S)(\forall t \in T_{\Sigma,s})$ $EVAL(t) \neq \perp_s$.

EVAL bzw. EV bewahren in M_C die Schachtelung von Operationen in einem Term $te T_\Sigma$. Dennoch ist EVAL kein Homomorphismus, da

- M_C und T_Σ verschiedene Signaturen haben
- EVAL(t) stets von der Standardumgebung ausgeht.

Dieser Mangel von EVAL, nicht homomorph zu sein, wird sich in 3.5 bei der Definition der korrekten Implementierung einer Datentypspezifikation durch eine Menge von Cluster/Scripts in einer verhältnismäßig starken Forderung an M_C auswirken.
Eine weitere Folge ist, daß das Bild von T_Σ unter EVAL i.a. keine Teilalgebra von M_C ist.

3.3.3 Beispiel zur Funktionsweise von EV

Sei S:={bool}

Σ:={ T,F : $\longrightarrow$ bool,
AND, OR: bool x bool $\longrightarrow$ bool}

Sei $te T_\Sigma$, t:= AND(OR(T,F),AND(T,T))

M_C:=($\{A_\perp, A_{bool}\}$,{$\overline{TRUE}$, $\overline{FALSE}$, $\overline{AND}$, $\overline{OR}$}) sei die von einem Cluster/Script "BOOL" erzeugte Modulalgebra.

Seien $\{\rho,\rho_1,\rho_2,\rho_0,\rho',\rho'_1,\rho'_2,\bar{\rho},\rho'',\rho''_1,\rho''_2,\rho''_{11},\rho''_{22}\} \subseteq GEnv$,
$\{a1,a2,b1,b2,c1,c2\} \subseteq Loc$.

$EVAL(t)=EV(\varepsilon,AND(OR(T,F),AND(T,T)))=\overline{AND}((a1,\rho),(a2,\rho))$ (3.3.1.1.1.2)

$(a1,\rho_1):=EV(\varepsilon,OR(T,F))=\overline{OR}((b1,\rho'),(b2,\rho'))$ (3.3.1.1.2.1)

$(b1,\rho'_1):=EV(\varepsilon,T)=\overline{TRUE}((\perp,\rho_0))$ (3.3.1.1.1.1)

$(b2,\rho'_2):=EV(T,F)=\overline{FALSE}((\perp,\bar{\rho}))$ (3.3.1.1.1.2)

mit $\bar{\rho}:=EV(\varepsilon,T)\downarrow 2=:\rho'_1$

$\Longrightarrow \rho'=\rho'_2$

$(a2,\rho_2):=EV(OR(T,F),AND(T,T))=\overline{AND}((c1,\rho''),(c2,\rho''))$ (3.3.1.1.2.2)

$(c1,\rho''_1):=EV(OR(T,F),T)=\overline{TRUE}((\perp,\rho''_{11}))$ (3.3.1.1.1.2)

mit $\rho''_{11}:=EV(\varepsilon,OR(T,F))\downarrow 2=\rho_1$

$(c2,\rho''_2):=EV(OR(T,F)T,T)=\overline{TRUE}((\perp,\rho''_{22}))$ (3.3.1.1.1.2)

mit $\rho''_{22}:=EV(OR(T,F),T)\downarrow 2=\rho''_1$

$\Longrightarrow \rho''=\rho''_2$

$\rho=\rho_2$

Die syntaktische Entsprechung zwischen der Termalgebra und der Modulalgebra M_C , nämlich $te T_\Sigma - EVAL(t)$ und $T_{\Sigma,s} - A_s, se S$, kann noch vervollständigt werden, indem der Zerlegung von T_Σ in Mengen $T_{\Sigma,s,j}$ von j-fach geschachtelten ($je N_0$) Termen (vgl. 1.1.3) der Sorte $se S$ eine Zerlegung der Trägermengen $A_s, se S$, von M_C in Mengen $A_{s,j}$, $se S$, $je N_0$, von Elementen zugeordnet wird, die durch j-fache Schachtelung von Operationsaufrufen in M_C entstanden sind.

3.3.4 Defintion (OP(n,s), R, $N_t(R,\rho)$, $A_{s,j}$)

Sei (S,Σ) eine Signatur, C eine Menge von Cluster/Scripts und $M_C := (\{A_s : se S \cup \{\perp\}\}, \{\overline{P}_f : f e \Sigma\})$ die von C erzeugte Modulalgebra.

1.1 $(\forall se S)\ OP(0,s) := \{\overline{P}_f : \overline{P}_f : A_\perp \longrightarrow A_s\} = \{\overline{P}_f : f e \Sigma_{\varepsilon,s}\}$

1.2 $(\forall ne N)(\forall se S)\ OP(n,s) := \{\overline{P}_f : (\exists s1,\ldots,sn e S)\ \overline{P}_f : A_{s1} x \ldots x A_{sn} \longrightarrow A_s\}$
$= \{\overline{P}_f : (\exists w e S^n)\ f e \Sigma_{w,s}\}$

2.1 $R \subseteq (A_\perp \downarrow 2)^2$
$(\rho,\rho') e R$:gdw $\rho = \rho'$ v
$(\exists ne N)(\exists s1,\ldots,sn, se S \cup \{\perp\})(\exists \overline{P}_f : A_{s1} x \ldots x A_{sn} \longrightarrow A_s)$
$(\exists((l1,\rho),\ldots,(ln,\rho),(l,\rho')) e A_{s1} x \ldots x A_{sn} x A_s)$
$(\overline{P}_f((l1,\rho),\ldots,(ln,\rho)) = (l,\rho'))$
für $\rho,\rho' e A_\perp \downarrow 2$.

2.2 $(\forall \rho e A_\perp \downarrow 2)$ $N_t(R,\rho) := \bigcup_{\substack{\rho' e A_\perp \downarrow 2 \\ (\rho,\rho') e R}} N_t(R,\rho') \cup \{\rho' : (\rho,\rho') e R\}$

$N_t(R,\rho)$ heißt der *transitive Nachbereich* von ρ unter R.

3. Sei $B \subseteq A_s^X$, $se S$.
$H(B) := \{(l,\rho) e A_s : (\exists (l,\overline{\rho}) e B)\ (\rho e N_t(R,\overline{\rho}))\}$

4. $(\forall se S)(\forall je N_0)$

$\overline{A}_{s,0} := \{\overline{P}_f((\perp,\rho)) : \overline{P}_f e OP(0,s),\ \rho e A_\perp \downarrow 2\}$

$\overline{A}_{s,j+1} := \{\overline{P}_f((l1,\rho),\ldots,(ln,\rho)) : \overline{P}_f e OP(n,s) : A_{s1} x \ldots x A_{sn} \longrightarrow A_s, ne N,$
$((l1,\rho),\ldots,(ln,\rho)) e A_{s1,k1} x \ldots x A_{sn,kn},$
$ki \leq j,\ i = 1,\ldots,n,\ s1,\ldots sn e S\}$

$A_{s,j} := H(\overline{A}_{s,j})$

$A_{s,j}$ enthält die Elemente aus $A_s, se S$, die durch die Anwendung von Opera-

tionen $\overline{P}_f \in M_C$ auf eine Argumentliste, deren Komponenten höchstens j-malige Schachtelungen von Operationsaufrufen sind, entstehen.

Die Anwendung von H auf $\overline{A}_{s,j}$ berücksichtigt das Lifting.
Es ist nach 3.3.4 klar, daß $M_C \downarrow 2 = \bigcup_{s \in S} \bigcup_{n \in N_0} OP(n,s) = \{\overline{P}_f : f \in \Sigma\}$ ist.

3.3.5 Lemma

Seien (S,Σ) eine Signatur, C eine Menge von Cluster/Scripts mit $Erf(C,S,\Sigma)$ und $M_C = (\{A_s : s \in S \cup \{\perp\}\}, \{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra.
Dann gilt:

1. $(\forall s \in S)(\forall A,B \subseteq A_s^X)$
   a) $B \subseteq H(B)$
   b) $A \subseteq B \Rightarrow H(A) \subseteq H(B)$
   c) $H(H(B)) = H(B)$
2. $(\forall s \in S)(\forall j \in N_0)\ A_{s,j} \subseteq A_{s,j+1}$
3. $(\forall s \in S)\ A_s = \bigcup_{j \in N_0} A_{s,j} \cup \{\perp_s\}$

Beweis:

1. a) Sei $(1,\rho) \in B$; $(\rho,\rho) \in R$ nach Definition von R $\Rightarrow \rho \in N_t(R,\rho)$
   $\Rightarrow (1,\rho) \in H(B)$ nach 3.3.4.3.
   b) Sei $(1,\rho) \in H(A) \Rightarrow (\exists (1,\overline{\rho}) \in A)(\rho \in N_t(R,\overline{\rho}))$
   $A \subseteq B \Rightarrow (1,\overline{\rho}) \in B \Rightarrow (1,\rho) \in H(B)$.
   c) $H(B) \subseteq H(H(B))$ wegen a).
   Sei $(1,\rho) \in H(H(B)) \Rightarrow (\exists (1,\overline{\rho}) \in H(B))(\rho \in N_t(R,\overline{\rho}))$
   $\Rightarrow (\exists (1,\overline{\rho}') \in B)(\overline{\rho} \in N_t(R,\overline{\rho}'))$
   $\Rightarrow \rho \in N_t(R,\overline{\rho}')$
   $\Rightarrow (1,\rho) \in H(B)$
   Damit erfüllt H die Eigenschaften einer Hüllenbildung.
2. Sei $s \in S$, $j \in N_0$.
   Nach 3.3.4.4 ist $\overline{A}_{s,j} \subseteq \overline{A}_{s,j+1}$.
   $\Rightarrow A_{s,j} \subseteq A_{s,j+1}$ wegen b).
3. Sei $s \in S$ beliebig.
   1. $\bigcup_{j \in N_0} A_{s,j} \cup \{\perp_s\} \subseteq A_s$, da in 3.3.4.3 für $(1,\rho) \in H(B)$ gefordert wurde, daß $(1,\rho) \in A_s$, wo $B \subseteq A_s^X \subseteq A_s$.

2."$\subseteq$"

Sei $(l,\rho) e A_s$.

zu zeigen: $(l,\rho) e \bigcup_{j \in N_0} A_{s,j}$

Der Beweis wird induktiv über die Definition der Modulalgebra geführt.

a) (l,ρ) in A_s aufgenommen per Create (ohne Parameter)

$\Rightarrow (l,\rho)=\overline{P}_f((\perp,\rho'))$, $\rho' e A_{\perp\downarrow 2} \Rightarrow (l,\rho) e \overline{A}_{s,0} \subseteq A_{s,0}$.

b) (l,ρ) in A_s aufgenommen als Ergebnis eines Funktions-/ Prozeduraufrufs:

$\Rightarrow (\exists n \in N)\{\exists s1,\ldots,sn,s \in S)(\exists \overline{P}_f e OP(n,s))(\exists\, j1,\ldots,jn \in N_0)$

$(\exists((l1,\rho'),\ldots,(ln,\rho')) e A_{s1,j1} x \ldots x A_{sn,jn}$ nach Induktionsvoraussetzung)

$((l,\rho) = \overline{P}_f((l1,\rho'),\ldots,(ln,\rho')))$

Sei $j=\max\{j1,\ldots,jn\} \Rightarrow (li,\rho') e A_{si,j}$, $i=1,\ldots,n$

$\Rightarrow (l,\rho) e \overline{A}_{s,j+1} \subseteq A_{s,j+1}$

c) (l,ρ) in A_s aufgenommen per Lifting:

$\Rightarrow (\exists \overline{\rho} e A_{\perp\downarrow 2})(\exists j \in N_0)((l,\overline{\rho}) e A_{s,j} \wedge (\overline{\rho},\rho) e R)$ nach Indutkionsvoraussetzung

$\Rightarrow (l,\rho) e H(A_{s,j}) = A_{s,j}$ nach 1.c).

MMM

3.3.6 Lemma

Sei (S,Σ) eine Signatur, C eine Menge von Cluster /Scripts mit $Erf(C,S,\Sigma)$ und $M_C := (\{A_s : s \in S \cup \{\perp\}\}, \{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra, die Σ enthält.

Dann gilt:

$$(\forall s \in S)(\forall j \in N_0)(\forall t \in T_{\Sigma,s,j})\ EVAL(t) e A_{s,j}.$$

Beweis:

Es wird gezeigt: $(\forall s \in S)(\forall j \in N_0)(\forall t1,\ldots,tm \in T_\Sigma)(\forall t \in T_{\Sigma,s,j})$
$(EV(t1\ldots tm,t) e A_{s,j})$

Der Beweis erfolgt induktiv über $j \in N_0$.

1. j=0, m=0: $j=0 \Rightarrow t \in \Sigma_{\varepsilon,s} \Rightarrow \overline{P}_f e OP(0,s)$

$\Rightarrow EV(\varepsilon,t)=\overline{P}_f((\perp,\rho_0))$ mit $\rho_0 e A_{\perp\downarrow 2}$ Standardumgebung

$\Rightarrow EV(\varepsilon,t) e \overline{A}_{s,0} \subseteq A_{s,0}$.

2. j=0, m>0: $j=0 \Rightarrow t \in \Sigma_{\varepsilon,s} \Rightarrow \overline{P}_f e OP(0,s)$

M_C enthält $\Sigma \Rightarrow EV(t1\ldots tm-1,tm) \downarrow 2 =: \rho e A_{\perp\downarrow 2}$

$\Rightarrow EV(t1\ldots tm,t) = \overline{P}_f((\perp,\rho)) e A_{s,0}$

3. j>0: Sei neN, t=f(t1',...,tn') mit $fe\Sigma_{s1,...,sn,s}$, s1,...,sneS.
Sei $ti'eT_{\Sigma,si,j-1}$, i=1,...,n, t1,...,tmeT_Σ.

$EV(t1...tm,f(t1',...,tn'))= \overline{P}_f((l1,\rho),...,(ln,\rho))$

mit $(l1,\rho_1):= EV(t1...tm,t1')$

$\vdots$

$(ln,\rho_n):= EV(t1...tmt1'...tn-1',tn')$

$\rho= \rho_n$

nach Induktionsvoraussetzung ist $(li,\rho_i)eA_{si,j-1}$, i=1,...,n.
M_C enthält Σ ==> $(li,\rho)eH(A_{si,j-1})=A_{si,j-1}$, i=1,...,n
==> $\overline{P}_f((l1,\rho),...,(ln,\rho))e\ \overline{A}_{s,j}{}^cA_{s,j}$.

MMM

Faßt man nun die Ergebnisse aus 3.1, 3.2 und 3.3 zusammen, so konnte auf der syntaktischen Ebene zwischen einer Datentypspezifikation (S,Σ,E) und einer Menge C von Cluster /Scripts , die die Modulalgebra $M_C:=(\{A_s: seS\cup\{\perp\}\},\{\overline{P}_f: fe\Sigma\})$ erzeugt und damit insbesondere $Erf(C,S,\Sigma,\alpha,\gamma)$ für ein α und γ gilt, folgende Verbindung hergestellt werden:

1. Die Signatur $(S,\Sigma)_\perp$ der Modulalgebra M_C hängt in eindeutiger Weise von der Signatur $(S,\Sigma)_\perp$ der Spezifikation ab und es gilt:
   - Jeder Sorte seS entspricht vermöge α ein eindeutiges Cluster/Script A'_s bzw. eine eindeutige Trägermenge A_s der Modulalgebra.
   - Jeder Operation $fe\Sigma$ entspricht vermöge γ eine eindeutige Prozedur bzw. Funktionsprozedur P_f eines Clusters/Scripts aus C bzw. eine eindeutige Operation $\overline{P}_f$ entsprechender Funktionalität aus der Modulalgebra.

2. Falls die Modulalgebra M_C Σ enthält, wird vermöge EVAL jedem Term teT_Σ ein Element (l,ρ) der Modulalgebra zugeordnet.

Als nächstes muß noch die Semantik eines abstrakten Datentyps untersucht werden. Das Ziel ist ein Kriterium, das angibt, wann die Modulalgebra M_C bzw. die Menge C von Cluster /Scripts die Spezifikation (S,Σ,E) korrekt implementiert.

Beim Übergang zur Semantik eines abstrakten Datentyps wird die Termalgebra T_Σ nach der durch die Gleichungen E erzeugten Verhaltensgleichheit $\sim_E$ faktorisiert.
Ideal wäre, man könnte den Gleichungen E bestimmte Forderungen an die

Syntax und Semantik der Cluster/Scripts aus C explizit zuordnen. Dies ist aber nicht möglich, da die Tatsache, ob eine Menge von Cluster /Scripts die Gleichungen einer Spezifikation (in einer noch anzugebenden Weise) erfüllt, nur implizit über den dynamischen Ablauf der Operationen der Cluster/Scripts ermittelt werden kann. Dies macht eine genauere Betrachtung des Verhaltens der aus den Clustern/Scripts erzeugbaren Moduln bzw. der Elemente der Modulalgebra erforderlich. Insbesondere muß für zwei Elemente der Modulalgebra geklärt werden, wann sie sich 'gleich verhalten'. Hierzu dient der nächste Abschnitt.

3.4 Verhaltensgleichheit in einer Modulalgebra

3.4.1 Definition (M_C implementiert bool)

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster/Scripts mit $Erf(C,S,\Sigma)$ und M_C die von C erzeugte Modulalgebra.

M_C *implementiert bool* :gdw

1. Es existieren zwei konstante Moduln $m_T:=(\beta_T,\sigma_T)$, $M_F:=(\beta_F,\sigma_F)\in Mod$ (d.h. die lokalen Zustände σ_T und σ_F können nicht geändert werden), für die $m_T \neq m_F$ gilt, und es existieren $l_T,l_F\in Loc$, so daß gilt:
   $(\forall \rho\in A_\perp\downarrow 2)((l_T,m_T),(l_F,m_F)\in\rho)$
2. $\{\overline{P}_T,\overline{P}_F,\overline{P}_\wedge,\overline{P}_v,\overline{P}_-\} \subseteq M_C\downarrow 2$ mit
   a) $\overline{P}_T,\overline{P}_F: A \longrightarrow A_{dis}$ mit
   $(\forall \rho\in A_\perp\downarrow 2)(\overline{P}_T((\perp,\rho)) = (l',\rho') \Longrightarrow \rho'(l') = \rho'(l_T) = m_T)$ bzw.
   $(\forall \rho\in A_\perp\downarrow 2)(\overline{P}_F((\perp,\rho)) = (l',\rho') \Longrightarrow \rho'(l') = \rho'(l_F) = m_F)$
   b) $\overline{P}_-: A_{dis} \longrightarrow A_{dis}$ mit
   $(\forall (l,\rho)\in A^x_{dis})(\overline{P}_-((l,\rho)) = (l',\rho') \Longrightarrow ((\rho(l)=m_T \Longleftrightarrow \rho'(l')=m_F)\wedge$
   $\rho(l)=m_F \Longleftrightarrow \rho'(l')=m_T)))$
   c) $\overline{P}_v: A_{dis} \times A_{dis} \longrightarrow A_{dis}$ mit
   $(\forall (l1,\rho),(l2,\rho)\in A^x_{dis})(\overline{P}_v((l1,\rho),(l2,\rho))=(l',\rho') \Longrightarrow$
   $\Longrightarrow ((\rho'(l')=m_T \Longleftrightarrow (\rho(l1)=m_T \vee \rho(l2)=m_T) \wedge$
   $(\rho'(l')=m_F \Longleftrightarrow (\rho(l1)=\rho(l2)=m_F))))$
   d) $\overline{P}_\wedge: A_{dis} \times A_{dis} \longrightarrow A_{dis}$ mit
   $(\forall (l1,\rho),(l2,\rho)\in A^x_{dis})(\overline{P}_\wedge ((l1,\rho),(l2,\rho))=(l',\rho') \Longrightarrow$
   $\Longrightarrow ((\rho'(l')=m_T \Longleftrightarrow (\rho(l1)=m_T=\rho(l2))) \wedge$
   $(\rho'(l')=m_F \Longleftrightarrow (\rho(l1)=m_F \vee \rho(l2)=m_F))))$

Nach dieser vorbereitenden Definition kann nun auf den Trägermengen einer Modulalgebra eine Verhaltensgleichheit definiert werden.

Die Verhaltensgleichheit wird bezogen auf zwei Moduln, die die Wahrheitswerte repräsentieren und per definitionem verschieden sind (vgl. m_T und m_F in 3.4.1). Zwei Elemente einer Trägermenge der Modulalgebra sollen dann verhaltensgleich sein, wenn sie bezüglich dieser zwei Moduln nicht unterschieden werden können.

Die Verhaltensgleichheit soll folgende Eigenschaften haben:

- Zwei verhaltensgleiche Elemente derselben globalen Umgebung sollen gegeneinander als Argumente von Operationen austauschbar sein.
- Auf der Ebene der Moduln besagt die Verhaltensgleichheit von $(1,\rho)$ und $(1',\rho')$, daß die entsprechenden Moduln $\rho(1)$ und $\rho'(1')$ dasselbe E/A-Verhalten haben, d.h., daß für alle von den Moduln $\rho(1)$ und $\rho'(1')$ durchführbaren Operationen gilt, daß aus der Verhaltensgleichheit der Eingabe die der Ausgabe folgt.

Die rekursive zweite Eigenschaft legt wieder eine induktive Vorgehensweise bei der Definition der Verhaltensgleichheit nahe.

Die Verhaltensgleichheit wird zunächst nur für Elemente der gleichen globalen Umgebung ρ definiert. Dieser Ansatz hat verschiedene Gründe:

- Die oben erwähnte induktive Vorgehensweise setzt beim Induktionsanfang einen Gleichheitsbegriff für die Elemente der Trägermengen voraus. Hierfür wird die komponentenweise Gleichheit gewählt.
- Faßt man eine Umgebung wieder als eine Modulgemeinschaft oder gar als Speicherbelegung auf, so ist ein Vergleich von Moduln nur in der gleichen Umgebung sinnvoll. So kann auch ein Programmierer nur Elemente der gleichen Umgebung vergleichen.

Da aber andererseits später die Verhaltensgleichheit in einer Modulalgebra M_C in noch zu bestimmender Weise die Verhaltensgleichheit $\sim_E$ in T_Σ widerspiegeln soll, muß diese Definition in einem zweiten Schritt auf Elemente unterschiedlicher Umgebungen erweitert werden. Allerdings werden für diesen Vergleich nur ganz bestimmte Umgebungen zugelassen.

3.4.2 Definition (Verhaltensgleichheit $\sim_{M_C}$)

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und $M_C := (\{A_s : s \in S \cup \{\bot\}\}, \{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiere.

1. $(\forall (l,\rho),(l',\rho) \in A^X_{dis})$

   $(l,\rho) \sim_{dis} (l',\rho)$:gdw $(\forall k \geq 0)((l,\rho) \overset{k}{\sim}_{dis} (l',\rho))$:gdw $\rho(l)=\rho(l')$

2. $(\forall s \in S-\{dis\})(\forall (l,\rho),(l',\rho) \in A^X_s)$

   a) $(l,\rho) \overset{0}{\sim}_s (l',\rho)$:gdw $(\forall n \in N)(\forall \overline{P}_f \in OP(n,dis): \exists 1 \leq i_0 \leq n : dom(\overline{P}_f) \downarrow i_0 = A_s)$
   $(\forall (li,\rho) \in A_{si}\ i=1,\ldots,n,\ i \neq i_0)$
   $(\overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho)) \sim_{dis} \overline{P}_f((l1,\rho),..,(l',\rho),..,(ln,\rho)))$

   b) $(\forall k \in N_0)$

   $(l,\rho) \overset{k+1}{\sim}_s (l',\rho)$:gdw $(\forall n \in N)(\forall s' \in S)(\forall \overline{P}_f \in OP(n,s'): \exists 1 \leq i_0 \leq n : dom(\overline{P}_f) \downarrow i_0 = A_s)$
   $(\forall (li,\rho) \in A_{si}\ i=1,\ldots,n,\ i \neq i_0)$
   $(\overline{P}_f((l1,\rho),\ ,(l_i,\rho),..,(ln,\rho)) \overset{k}{\sim}_{s'} \overline{P}_f((l1,\rho),..,(l_i',\rho),..,(ln,\rho)))$

   c) $(l,\rho) \sim_s (l',\rho)$:gdw $(\forall k \in N_0)((l,\rho) \overset{k}{\sim}_s (l',\rho))$

3. $(\forall s \in S)\ (\bot_s,\bot_s) \in \sim_s$

4. $\sim_\bot := \{(a,a): a \in A_\bot\}$

5. $\sim_{M_C} := \{\sim_s : s \in S \cup \{\bot\}\}$

3.4.2 besagt, daß zwei Elemente, die verhaltensgleich sind, gegeneinander austauschbar sind als Argumente einer Operation der Modulalgebra.

Die Induktion über k gibt an, über wieviel Stufen (Sorten) zwei Elemente (l,ρ) und (l',ρ) verglichen werden müssen, um sie bezüglich der Wahrheitswerte zu unterscheiden.

Die (noch zu zeigende) Kongruenzeigenschaft von $\sim_{M_C}$ ist eine verallgemeinerte Fassung der Gleichheit des E/A-Verhaltens.

3.4.2 erfüllt damit die eingangs an die Verhaltensgleichheit gestellten Forderungen.

Außerdem wird in 3.4.2 deutlich, daß dieser Definition ein terminaler Ansatz zugrundeliegt (vgl. 1.4.2).

Das folgende Lemma zeigt, daß die Forderung an M_C, bool zu implementieren, keine Einschränkung der gewünschten Kongruenzeigenschaft von $\sim_{M_C}$ ist.

3.4.3 Lemma

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und M_C die von C erzeugte Modulalgebra, die bool implementiere; $\sim_{dis}$ wie in 3.4.2.
Dann ist $\sim_{dis}$ eine Äuivalenzrelation auf A_{dis} und hat die Kongruenzeigenschaft bzgl. $\overline{P}_f$, $f\in\{-,v,\wedge\}$.

Beweis:

Da $\sim_{dis}$ über "=" (Gleichheit von Moduln) definiert ist, ist $\sim_{dis}$ eine Äquivalenzrelation auf A_{dis}.
Kongruenzeigenschaft bzgl. $\overline{P}_f\in\{\overline{P}_-,\overline{P}_v,\overline{P}_\wedge\}$:
Sei $(1i,\rho),(1i',\rho)\in A^X_{dis}$ mit $(1i,\rho) \sim_{dis} (1i',\rho)$, $i=1,2$.

1. Zu zeigen: $\overline{P}_-((11,\rho)) \sim_{dis} \overline{P}_-((11',\rho))$
   $(11,\rho) \sim_{dis} (11',\rho)$ ==> $\rho(11) = m_v = \rho(11')$, $v\in\{T,F\}$.
   Sei $(1,\overline{\rho}) := \overline{P}_-((11,\rho))$, $(1',\overline{\rho}') := \overline{P}_-((11',\rho))$.
   $v=T$ ==> $\overline{\rho}(1) = m_F$, $\overline{\rho}'(1') = m_F$ ==> $(1,\rho) \sim_{dis} (1',\rho)$.
   $v=F$ analog.

2. Zu zeigen: $(1,\overline{\rho}) := \overline{P}_f((11,\rho),(12,\rho)) \sim_{dis} \overline{P}_f((11',\rho),(12',\rho)) =: (1',\overline{\rho}')$,
   $f\in\{v,\wedge\}$

   a) $f=v$:
   $\overline{\rho}(1)=m_T$ <==> $(\rho(11) = m_T \vee \rho(12) = m_T)$ <==> $(\rho(11') = m_T \vee \rho(12')=m_T)$
   <==> $\overline{\rho}'(1') = m_T$
   $\overline{\rho}(1)=m_F$ <==> $(\rho(11) = m_F = \rho(12))$ <==> $(\rho(11') = m_F = \rho(12'))$
   <==> $\overline{\rho}'(1') = m_F$
   ==> $(1,\overline{\rho}) \sim_{dis} (1',\overline{\rho}')$.

   b) $f=\wedge$:
   $\overline{\rho}(1)=m_T$ <==> $(\rho(11) = m_T = \rho(12))$ <==> $(\rho(11') = m_T = \rho(12'))$
   <==> $\overline{\rho}'(1') = m_T$
   $\overline{\rho}(1)=m_F$ <==> $(\rho(11) = m_F \vee \rho(12) = m_F)$ <==> $(\rho(11') = m_F \vee \rho(12')=m_F)$
   <==> $\overline{\rho}'(1') = m_F$
   ==> $(1,\overline{\rho}) \sim_{dis} (1',\overline{\rho}')$.

3. Sei $(l1,\rho) = \perp_{dis}$ ==> $(l1',\rho) = \perp_{dis}$

   ==> $\overline{P}_-((l1,\rho)) = \perp_{dis} = \overline{P}_-((l1',\rho))$

   bzw. $\overline{P}_f((l1,\rho),(l2,\rho)) = \perp_{dis} = \overline{P}_f((l1',\rho),(l2',\rho))$, $f\in\{\vee,\wedge\}$.

Damit hat $\sim_{dis}$ die Kongruenzeigenschaft bzgl. $\overline{P}_-,\overline{P}_\vee$ und $\overline{P}_\wedge$.

■■■

3.4.4 Lemma

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$, $M_C:=(\{A_s:s\in S\cup\{\perp\}\},\{\overline{P}_f:f\in\Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiere, $\sim_{M_C}=\{\sim_s:s\in S\cup\{\perp\}\}$ wie in 3.4.2 und $(S',\Sigma'):=(S,\Sigma)_\perp$.
Dann ist $\sim_{M_C}$ eine Σ'-Kongruenz auf M_C.

Beweis:

A) Zu zeigen: $(\forall s\in S\cup\{\perp\})$ ist $\sim_s$ Äquivalenz auf A_s.
   Nach 3.4.3 ist $\sim_{dis}$ Äquivalenz auf A_{dis}.
   $\sim_\perp$ ist als Identität auf $A_\perp$ trivialerweise Äquivalenz auf $A_\perp$.
   Sei im folgenden $s\in S'-\{dis,\perp\}$.

   1. Reflexivität:

      $(\perp_s,\perp_s)\in\sim_s$ per definitionem.

      Sei $(l,\rho)\in A_s^x$.

      k=0: Sei $n\in N$, $\overline{P}_f\in OP(n,dis)$ mit $\exists 1\le i_o\le n$: $dom(\overline{P}_f)\downarrow i_o = A_s$,
      $(li,\rho)\in A_{si}$, $i=1,\ldots,n$, $i\neq i_o$.

      Da $\sim_{dis}$ reflexiv ist, gilt:
      $\overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho)) \sim_{dis} \overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho))$

      ==> $(l,\rho) \overset{0}{\sim}_s (l,\rho)$.

      Damit ist $\overset{0}{\sim}_s$ reflexiv.

      k>0: Sei $n\in N$, $s'\in S$, $\overline{P}_f\in OP(n,s')$ mit $\exists 1\le i_o\le n$: $dom(\overline{P}_f)\downarrow i_o = A_s$,
      $(li,\rho)\in A_{si}$, $i=1,\ldots,n$, $i\neq i_o$.
      Nach Induktionsvoraussetzung ist $\overset{k-1}{\sim}_s$ reflexiv
      ==> $\overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho)) \overset{k-1}{\sim}_s \overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho))$

      ==> $(l,\rho) \overset{k}{\sim}_s (l,\rho)$.

      Damit ist $\overset{k}{\sim}_s$ reflexiv für $k\in N_0$, $s\in S$.

   Insgesamt ist damit $\sim_s$ reflexiv für $s\in S\cup\{\perp\}$.

2. Symmetrie:

Sei $(l,\rho),(l',\rho) \in A_s$ mit $(l,\rho) \sim_s (l',\rho)$.

Zu zeigen: $(l',\rho) \sim_s (l,\rho)$.

k=0: Sei $n \in \mathbb{N}$, $\overline{P}_f \in OP(n,dis)$ mit $\exists 1 \leq i_o \leq n: dom(\overline{P}_f) \downarrow i_o = A_s$,
$(li,\rho) \in A_{si}$, $i=1,\ldots,n$, $i \neq i_o$.

$(l,\rho) \sim_s (l',\rho) \Longrightarrow \overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho)) \sim_{dis} \overline{P}_f((l1,\rho),..,(l',\rho),..,(ln,\rho))$

$\sim_{dis}$ ist symmetrisch

$\Longrightarrow \overline{P}_f((l1,\rho),..,(l',\rho),..,(ln,\rho)) \sim_{dis} \overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho))$

$\Longrightarrow (l',\rho) \overset{0}{\sim}_s (l,\rho)$.

k>0: Analog dem Fall k=0 unter Verwendung der Induktionsvoraussetzung, daß $\overset{k-1}{\sim}_{s'}$ symmetrisch für $s' \in S$.

Damit gilt: $\sim_s$ ist symmetrisch für $s \in S \cup \{\perp\}$.

3. Transitivität:

Sei $(l,\rho),(l',\rho),(l'',\rho) \in A_s$ mit $(l,\rho) \sim_s (l',\rho)$ und $(l',\rho) \sim_s (l'',\rho)$.

Zu zeigen: $(l,\rho) \sim_s (l'',\rho)$.

k=0: Sei $n \in \mathbb{N}$, $\overline{P}_f \in OP(n,dis)$ mit $\exists 1 \leq i_o \leq n: dom(\overline{P}_f) \downarrow i_o = A_s$,
$(li,\rho) \in A_{si}$ $i=1,\ldots,n$, $i \neq i_o$.

$(l,\rho) \sim_s (l',\rho)$

$\Longrightarrow \overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho)) \sim_{dis} \overline{P}_f((l1,\rho),..,(l',\rho),..,(ln,\rho))$

$(l',\rho) \sim_s (l'',\rho)$

$\Longrightarrow \overline{P}_f((l1,\rho),..,(l',\rho),..,(ln,\rho)) \sim_{dis} \overline{P}_f((l1,\rho),..,(l'',\rho),..,(ln,\rho))$

$\sim_{dis}$ ist transitiv

$\Longrightarrow \overline{P}_f((l1,\rho),..,(l,\rho),..,(ln,\rho)) \sim_{dis} \overline{P}_f((l1,\rho),..,(l'',\rho),..,(ln,\rho))$

$\Longrightarrow (l,\rho) \overset{0}{\sim}_s (l'',\rho)$.

k>0: Analog dem Fall k=0 unter Verwendung der Induktionsvoraussetzung, daß $\overset{k-1}{\sim}_{s'}$ transitiv ist für $s' \in S$.

Damit gilt: $\sim_s$ ist transitiv für $s \in S \cup \{\perp\}$.

Insgesamt ist also $\sim_s$ Äquivalenzrelation auf A_s für $s \in S \cup \{\perp\}$.

B) Kongruenzeigenschaft

Sei $n \in N_0$, $s \in S$ und $\overline{P}_f \in OP(n,s)$.

1. n=0: Da $\sim_\perp = \{(a,a): a \in A_\perp\}$ und $\sim_s$ reflexiv, hat $\sim_{M_C}$ bzgl. $\overline{P}_f$ trivialerweise die Kongruenzeigenschaft.

2. n>0: Seien $(1i,\rho), (1i',\rho) \in A_{si}$ mit $(1i,\rho) \sim_{si} (1i',\rho)$ i=1,...,n.

Zu zeigen: $\overline{P}_f((11,\rho),\ldots,(1n,\rho)) \sim_s \overline{P}_f((11',\rho),\ldots,(1n',\rho))$.

Sei $k \in N_0$ beliebig.

Da $(11,\rho) \sim_{s1} (11',\rho)$ ==> $(11,\rho) \overset{k+1}{\sim}_{s1} (11',\rho)$

==> $\overline{P}_f((11,\rho),(12,\rho),..,(1n,\rho)) \overset{k}{\sim}_s \overline{P}_f((11',\rho),(12,\rho),..,(1n,\rho))$

$\vdots$

Da $(1n,\rho) \sim_{sn} (1n',\rho)$ ==> $(1n,\rho) \overset{k+1}{\sim}_{sn} (1n',\rho)$

==> $\overline{P}_f((11,\rho),..,(1n-1,\rho),(1n,\rho)) \overset{k}{\sim}_s \overline{P}_f((11,\rho),\ldots,(1n',\rho))$

Unter Ausnutzung der Transitivität von $\sim_s$ erhält man dann:

$\overline{P}_f((11,\rho),\ldots,(1n,\rho)) \overset{k}{\sim}_s \overline{P}_f((11',\rho),\ldots,(1n',\rho))$.

Damit hat $\sim_{M_C}$ die Kongruenzeigenschaft bzgl. $\overline{P}_f$.

Insgesamt hat $\sim_{M_C}$ die Kongruenzeigenschaft bzgl. $\overline{P}_f$, $f \in \Sigma$.

A) und B) ergeben damit den Nachweis, daß $\sim_{M_C}$ eine Σ'-Kongruenz auf M_C ist.

MMM

Von zwei verhaltensgleichen Elementen $(1,\rho)$ und $(1',\rho)$ wird man annehmen, daß sie auch in allen Umgebungen $\rho' := OP[\overline{\rho}](\rho)$ mit $OP[\overline{\rho}] \in OP_{M_C}$ verhaltensgleich sind, d.h. $(1,\rho') \sim_{M_C} (1',\rho')$, da durch $OP[\overline{\rho}]$ die Elemente der Umgebung ρ nicht beeinflußt werden, sondern lediglich die Umgebung ρ erweitert wird. Daher wird die Definition der Verhaltensgleichheit noch vervollständigt:

3.4.5 Definition ($\sim'_{M_C}$)

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und $M_C := (\{A_s : s \in S \cup \{\perp\}\}, \{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiere.

1. $(\forall s \in S \cup \{\perp\})(\forall (l,\rho),(l',\rho) \in A_s)$

   $(l,\rho) \sim'_s (l',\rho)$:gdw $(\forall OP \in OP_{M_C}, \rho' := OP(\rho))$

   $((l,\rho') \sim_s (l',\rho'))$

2. $\sim'_{M_C} := \{\sim'_s : s \in S \cup \{\perp\}\}$.

3.4.6 Lemma

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und $M_C := (\{A_s : s \in S \cup \{\perp\}\}, \{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiere.

Dann gilt: $\sim'_{M_C}$ ist Äquivalenzrelation auf M_C, d.h.

$\sim'_s$ ist Äquivalenz auf A_s für $s \in S \cup \{\perp\}$.

Beweis:

Sei $s \in S \cup \{\perp\}$.

1. Reflexivität:

   Sei $(l,\rho) \in A_s$, $OP \in OP_{M_C}$, $\rho' := OP(\rho)$.

   $\sim_s$ ist reflexiv ==> $(l,\rho') \sim_s (l,\rho')$

   ==> $(l,\rho) \sim'_s (l,\rho)$

   ==> $\sim'_s$ ist reflexiv für $s \in S \cup \{\perp\}$.

2. Symmetrie:

   Seien $(l,\rho),(l',\rho) \in A_s$ mit $(l,\rho) \sim'_s (l',\rho)$.

   Zu zeigen: $(l',\rho) \sim'_s (l,\rho)$.

   Sei $OP \in OP_{M_C}$ und $\rho' := OP(\rho)$.

   $(l,\rho) \sim'_s (l',\rho)$ ==> $(l,\rho') \sim_s (l',\rho')$.

   $\sim_s$ ist symmetrisch ==> $(l',\rho') \sim_s (l,\rho')$

   ==> $(l',\rho) \sim'_s (l,\rho)$.

   ==> $\sim'_s$ ist symmetrisch für $s \in S \cup \{\perp\}$.

3. Transitivität:

   Seien $(l,\rho),(l',\rho),(l'',\rho) \in A_s$ mit $(l,\rho) \sim'_s (l',\rho)$ und $(l',\rho) \sim'_s (l'',\rho)$.

   Zu zeigen: $(l,\rho) \sim'_s (l'',\rho)$.

   Sei $OP \in OP_{M_C}$ und $\rho' := OP(\rho)$.

   Nach Voraussetzung gilt: $(l,\rho') \sim_s (l',\rho')$ und $(l',\rho') \sim_s (l'',\rho')$.

   $\sim_s$ ist transitiv ==> $(l,\rho') \sim_s (l'',\rho')$

   ==> $(l,\rho) \sim'_s (l'',\rho)$

   ==> $\sim'_s$ ist transitiv für $s \in S \cup \{\perp\}$.

Insgesamt ist damit $\sim'_{M_C}$ eine Äquivalenz auf M_C.

■■■

3.4.7 Corollar

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und $M_C := (\{A_s : s \in S \cup \{\perp\}\}, \{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiere und die Standardumgebung ρ_0 habe.

Sei $n \in N_0$ und $OP_i \in OP_{M_C}$, $i=1,\ldots,n$ Operationsketten; α und β seien Permutationen der Zahlen $1,\ldots,n$.

$OP := OP_{\alpha(1)} \_ \cdots \_ OP_{\alpha(n)}$, $OP' := OP_{\beta(1)} \_ \cdots \_ OP_{\beta(n)}$.

Sei $s \in S \cup \{\perp\}$, $(l,\rho),(l',\rho) \in A_s$ mit $(l,\rho) \sim'_s (l',\rho)$.

Dann gilt: Es existiert $\pi \in Pm(Loc,\rho_0)$, so daß

$$\pi OP(\rho) = OP'(\rho) =: \rho' \quad \text{und}$$
$$(l, OP(\rho)) \sim'_s (l', OP(\rho)) \quad \text{und}$$
$$(\pi(l), \pi OP(\rho)) = (l,\rho') \sim_s (l',\rho') = (\pi(l'), \pi OP(\rho)).$$

Beweis:

Verwende das $\pi \in Pm(Loc,\rho_0)$ aus 3.2.11.

Dann gilt: $\pi|_{dom(\rho)} = id_{dom(\rho)}$

$$\Longrightarrow \pi(l) = l,\ \pi(l') = l'.$$

Der Rest der Behauptung folgt dann mit 3.4.5, der Definition von $\sim'_{M_C}$.

MMM

3.2.9 läßt sich in einfacher Weise erweitern auf die Operationskette einer Modulalgebra:

3.4.8 Corollar

Sei M_C eine Modulalgebra mit der Standardumgebung $\rho_0 \in A_{\perp} \downarrow 2$.
Dann gilt für alle $OP \in OP_{M_C}$ und alle $\pi \in Pm(Loc,\rho_0)$, $\rho \in A_{\perp} \downarrow 2$:

$$OP(\pi\rho) = \pi OP(\rho).$$

Beweis: Sei $\pi \in Pm(Loc,\rho_0)$, $\rho \in A_{\perp} \downarrow 2$, $n \in N$, $OP := \overline{P}_n \_ \ldots \_ \overline{P}_1 \in OP_{M_C}$.

Führe den Beweis durch Induktion über n.

n=1: $OP = \overline{P} \Longrightarrow \overline{P}$ ist Create ohne Parameter, $\overline{P}((\perp,\rho)) =: (l,\rho')$

$$\Longrightarrow \overline{P}((\pi(\perp),\rho)) = (\pi(l),\pi\rho') \text{ nach 3.2.9}$$
$$\Longrightarrow \pi\overline{P}(\rho) = \overline{P}(\pi\rho) \qquad (\overline{P} \in OP_{M_C}!).$$

Sei die Behauptung richtig für n.

Sei $OP' := \overline{P}\_OP$, $\rho_1 := OP(\rho)$.

Sei $(l,\rho_2) := \overline{P}((l1,\rho_1),\ldots,(lm,\rho_1))$ mit $((l1,\rho_1),\ldots,(lm,\rho_1)) \in dom(\overline{P}) \wedge li \notin dom(\rho)$ $i=1,\ldots,m$

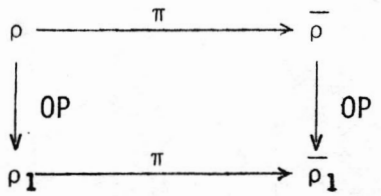
$\Longrightarrow (\pi(l),\pi\rho_2) = \overline{P}((\pi(l1),\pi\rho_1),..,(\pi(lm),\pi\rho_1))$ nach 3.2.9

$= \overline{P}((\pi(l1),\pi OP(\rho)),\ldots,(\pi(lm),\pi OP(\rho)))$

$= \overline{P}((\pi(l1),OP(\pi\rho)),\ldots,(\pi(lm),OP(\pi\rho)))$ nach Induktionsvoraussetzung.

$\Longrightarrow \pi\overline{P}\_OP(\rho) = \overline{P}\_OP(\pi\rho)$.

MMM



Graphisch dargestellt bedeutet 3.4.8, daß nebenstehendes Diagramm kommutiert.

Wie eingangs angekündigt, wird nun der Begriff der Verhaltensgleichheit erweitert für Elemente verschiedener Umgebungen.

Die Relation $\sim_{M_C}$ war definiert worden aus der Sicht eines Programmierers bzw. eines Anwenders, der zu einem bestimmten Zeitpunkt genau eine globale Umgebung (Speicherbelegung) betrachtet. Dazuhin muß aber auch der Gesichtspunkt desjenigen berücksichtigt werden, der einen abstrakten Datentyp spezifiziert und verschiedene Terme der Termalgebra vergleicht.
Sei beispielsweise (S,Σ,E) die Spezifikation eines Data-Kellers und $k \in T_{\Sigma,stack}$, t:=Pop(Push(k,Createdata)). Durch die Gleichungen in E werde k und t identifiziert. C sei eine Cluster-/Scriptmenge, die (S,Σ,E) implementieren soll.
In der von C erzeugten Modulalgebra gehören EVAL(k) und EVAL(t) in der Regel verschiedenen Umgebungen an, da EVAL(t)↓2 im allgemeinen noch den durch 'Createdata' erzeugten Modul zusätzlich zu der Modulgemeinschaft in EVAL(k)↓2 enthält. Um nun in der Modulalgebra die Elemente EVAL(k) und EVAL(t) vergleichen zu können, muß die Umgebung EVAL(k)↓2 um den durch Createdata erzeugten Modul ergänzt werden, was auf die Modulgemeinschaft in EVAL(k)↓2 keinen ändernden Einfluß hat.

Allgemein wird man beim Vergleich zweier Elemente (l,ρ) und (l',ρ') einer Modulalgebra versuchen, die Umgebungen ρ und ρ' zu einer gemeinsamen Umgebung $\overline{\rho}$ so zu erweitern, daß ρ und ρ' nicht "wesentlich verfälscht" werden und in der Umgebung $\overline{\rho}$ dann ein Vergleich zwischen $(l,\overline{\rho})$ und $(l',\overline{\rho})$ möglich ist.

Eine Möglichkeit, zwei Umgebungen $\rho,\rho' \underline{c} A_{\perp}{}^{\downarrow 2}$ einander anzugleichen besteht darin, auf ρ die Operationsfolge $\overline{OP}[\rho']$ und auf ρ' die Kette $OP[\rho]$ anzuwenden. Anschaulich entspricht dies einer Summierung der Umgebungen ρ und ρ'.
"Nicht wesentlich verfälscht" heißt dann mit 3.2.11 , daß die zwei so erhaltenen Umgebungen $OP[\rho](\rho')$ und $OP[\rho'](\rho)$ bis auf Permutation der Locations gleich sind.

Von zwei leeren Kellern z.B. wird man aber nicht nur die Verhaltensgleichheit in einer gemeinsamen Umgebung ρ_1 erwarten, sondern ebenso in allen Umgebungen $\overline{\rho}$ des transitiven Nachbereiches von ρ_1, die ρ_1 "unverfälscht enthalten", insbesondere also auch die beiden Keller.

3.4.9 Definition (potentielle Verhaltensgleichheit)

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und $M_C:=(\{A_s: s\in S\cup\{\perp\}\},\{\overline{P}_f: f\in\Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiere und die Standardumgebung ρ_0 habe.

1. $\hat{=}_{dis} := \sim'_{dis}$.
2. $\hat{=}_{\perp} := \{(a,a): a\in A_{\perp}\}$
3. $(\forall s\in S-\{dis\})(\forall(l,\rho),(l',\rho')\in A_s^X)$
   $(l,\rho)\ \hat{=}_s\ (l',\rho')$:gdw $(\exists OP1, OP2\in OP_{M_C})(\exists\pi\in Pm(Loc,\rho_0))$
   $((\pi OP1(\rho) = OP2(\rho') =:\rho_1)$
   $((\forall OP3\in OP_{M_C},\ \overline{\rho}:=OP3(\rho_1))$
   $((\pi(l),\overline{\rho})\sim'_s (l',\overline{\rho}))))$
4. $(\forall s\in S)((\perp_s,\perp_s)\in\hat{=}_s)$
5. $\hat{\equiv} := \{\hat{=}_s: s\in S\cup\{\perp\}\}$.

Von zwei potentiell verhaltensgleichen Elementen in den Umgebungen ρ bzw. ρ' wird also gefordert, daß diese Elemente in einer gemeinsamen erweiterten Umgebung ρ_1 (d.h. $OP3=\varepsilon$) bis auf Permutation der Locations bzgl. $\sim'_{M_C}$ gleich sind und dies auch in allen Folgeumgebungen $\overline{\rho}$ von ρ_1 gilt, die man durch Anwendung einer Operationskette in ρ_1 erhält.

3.4.10 Lemma

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und $M_C:=(\{A_s: s\in S\cup\{\perp\}\},\{\overline{P}_f: f\in\Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiert und die Standardumgebung ρ_0 habe.
Dann gilt: $(\forall s\in S\cup\{\perp\})$ $\hat{=}_s$ ist Äquivalenz auf A_s, d.h. $\hat{\equiv}$ ist Äquivalenz auf M_C.

Beweis:

Sei $s \in S \cup \{\bot\}$.

1. $s = \bot$: $\hat{\sim}_\bot$ ist die identische Relation auf $A_\bot$
   $\Longrightarrow$ $\hat{\sim}_\bot$ ist Äquivalenz auf $A_\bot$.

2. $s = dis$: $\hat{\sim}_{dis} = \sim'_{dis}$; $\sim'_{dis}$ ist Äquivalenz auf A_{dis} (vgl.3.4.6)
   $\Longrightarrow$ $\hat{\sim}_{dis}$ ist Äquivalenz auf A_{dis}.

3. Sei nun $s \in S - \{dis\}$.

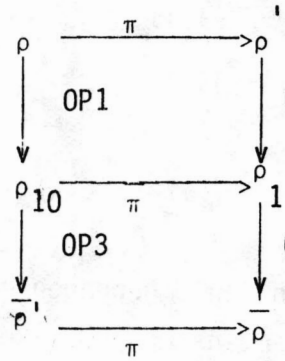
   3.1 Reflexivität:

   Setze in 3.4.9: $OP1 := OP2 := \varepsilon$, $\pi := id_{Loc}$
   $\Longrightarrow$ $\rho_1 = \rho$.
   Sei $OP3 \in OP_{M_C}$, $\overline{\rho} := OP3(\rho)$.
   Da $\sim'_s$ reflexiv ist (vgl. 3.4.6), gilt: $(l,\overline{\rho}) \sim'_s (l,\overline{\rho})$
   $\Longrightarrow$ $(l,\rho) \hat{\sim}_s (l,\rho)$.
   $(\bot_s,\bot_s) \in \hat{\sim}_s$
   $\Longrightarrow$ $\hat{\sim}_s$ ist reflexiv für $s \in S \cup \{\bot\}$.

   3.2 Symmetrie:

   Seien $(l,\rho),(l',\rho') \in A_s^X$ mit $(l,\rho) \hat{\sim}_s (l',\rho')$.
   Zu zeigen: $(l',\rho') \hat{\sim}_s (l,\rho)$.

   $(l,\rho) \hat{\sim}_s (l',\rho') \Longrightarrow \exists OP1, OP2 \in OP_{M_C}, \exists \pi \in Pm(Loc,\rho_0)$ mit



   $\pi OP1(\rho) = OP2(\rho') =: \rho_1$

   Sei $\rho_{10} := OP1(\rho)$

   $\Longrightarrow$ $\rho_{10} = \pi^{-1} \rho_1$

   bzw. $\pi^{-1} OP2(\rho') = OP1(\rho)$.

   Sei nun $OP3 \in OP_{M_C}$, $\overline{\rho}' := OP3(\rho_{10})$.

   Sei $\overline{\rho} := \pi\overline{\rho}'$

   $\Longrightarrow$ $\overline{\rho} = OP3(\rho_1)$ nach 3.4.8.

   $(l,\rho) \hat{\sim}_s (l',\rho') \Longrightarrow (\pi(l),\overline{\rho}) \sim'_s (l',\overline{\rho})$
   $\Longrightarrow$ $(\pi^{-1} \circ \pi(l), \pi^{-1}\overline{\rho}) \sim'_s (\pi^{-1}(l'), \pi^{-1}\overline{\rho})$ nach 3.2.9
   $\Longrightarrow$ $(\pi^{-1}(l'),\overline{\rho}') \sim'_s (l,\overline{\rho}')$ da $\sim'_s$ symmetrisch (3.4.6)

   $\Longrightarrow$ $(l',\rho') \sim'_s (l,\rho)$

   $\Longrightarrow \hat{\sim}_s$ ist symmetrisch für $s \in S \cup \{\bot\}$.

3.3 Transitivität:

Seien $(l,\rho),(l',\rho'),(l'',\rho'')\in A_s^X$ mit $(l,\rho) \hat{=}_s (l',\rho')$ und $(l',\rho') \hat{=}_s (l'',\rho'')$.

Zu zeigen: $(l,\rho) \hat{=}_s (l'',\rho'')$.

Nach Voraussetzung gilt:

$\exists \rho_{10} \in A_{\perp}^{\downarrow 2}$: $\rho_{10} = OP1(\rho)$ für ein $OP1 \in OP_{M_C}$

$\exists \rho_1 \in A_{\perp}^{\downarrow 2}$: $\rho_1 = OP2(\rho')$ für ein $OP2 \in OP_{M_C}$

und $\exists \pi_1 \in Pm(Loc,\rho_0)$: $\pi_1 \rho_{10} = \rho_1$;

bzw.

$\exists \rho_{20} \in A_{\perp}^{\downarrow 2}$: $\rho_{20} = OP3(\rho')$ für ein $OP3 \in OP_{M_C}$

$\exists \rho_2 \in A_{\perp}^{\downarrow 2}$: $\rho_2 = OP4(\rho'')$ für ein $OP4 \in OP_{M_C}$

und $\exists \pi_2 \in Pm(Loc,\rho_0)$: $\pi_2 \rho_{20} = \rho_2$.

Setze nun:

$\rho_3 := OP2(\rho_2)$, $\rho_{30} := OP2(\rho_{20})$

$\Longrightarrow \quad \pi_2 \rho_{30} = \rho_3$ nach 3.4.8

$\rho_{300} := OP3(\rho_1)$, $\rho_{3000} := OP3(\rho_{10})$

$\Longrightarrow \quad \pi_1 \rho_{3000} = \rho_{300}$ nach 3.4.8.

Nach 3.2.11 gilt nun:

$\exists \pi_3 \in Pm(Loc,\rho_0)$: $\pi_3 OP3\_OP2(\rho') = OP2\_OP3(\rho')$

und $\pi_3|_{dom(\rho')} = id_{dom(\rho')}$, insbesondere $\pi_3(l')=l'$

$\Longrightarrow \pi_2 \circ \pi_3 \circ \pi_1 OP3\_OP1(\rho) = OP2\_OP4(\rho'')$.

Sei nun $OP5 \in OP_{M_C}$ und $\overline{\rho} := OP5(\rho_3)$.

Zu zeigen: $(\pi_2 \circ \pi_3 \circ \pi_1(l),\overline{\rho}) \sim_s' (l'',\overline{\rho})$.

Nach Voraussetzung gilt:

$(\pi_2(l'),\rho_2) \sim_s' (l'',\rho_2)$

$\Longrightarrow (\pi_2(l'),\overline{\rho}) \sim_s' (l'',\overline{\rho})$ nach 3.4.5 (*)

Analog gilt:

$(\pi_1(l),\rho_1) \sim_s' (l',\rho_1)$; sei $\overline{\rho}' := OP5\_OP3(\rho_1)$

$\Longrightarrow (\pi_1(l),\overline{\rho}') \sim_s' (l',\overline{\rho}')$ nach 3.4.5

$\Longrightarrow (\pi_2 \circ \pi_3 \circ \pi_1(l),\pi_2 \circ \pi_3 \overline{\rho}') \sim_s' (\pi_2 \circ \pi_3(l'),\pi_2 \circ \pi_3 \overline{\rho}')$ nach 3.2.9.

$\overline{\rho}' = OP5\_OP3(\rho_1) = OP5(\rho_{300})$

$\Longrightarrow \quad \pi_2 \circ \pi_3 \overline{\rho}' = OP5(\pi_2 \circ \pi_3 \rho_{300})$ nach 3.4.8

$= OP5(\rho_3)$

$= \overline{\rho}$

Damit gilt dann:

$(\pi_2 \circ \pi_3 \circ \pi_1(l), \overline{\rho}) = (\pi_2 \circ \pi_3 \circ \pi_1(l), \pi_2 \circ \pi_3 \overline{\rho}')$

$\sim_s' (\pi_2 \circ \pi_3(l'), \pi_2 \circ \pi_3 \overline{\rho}') = (\pi_2(l'), \overline{\rho})$.

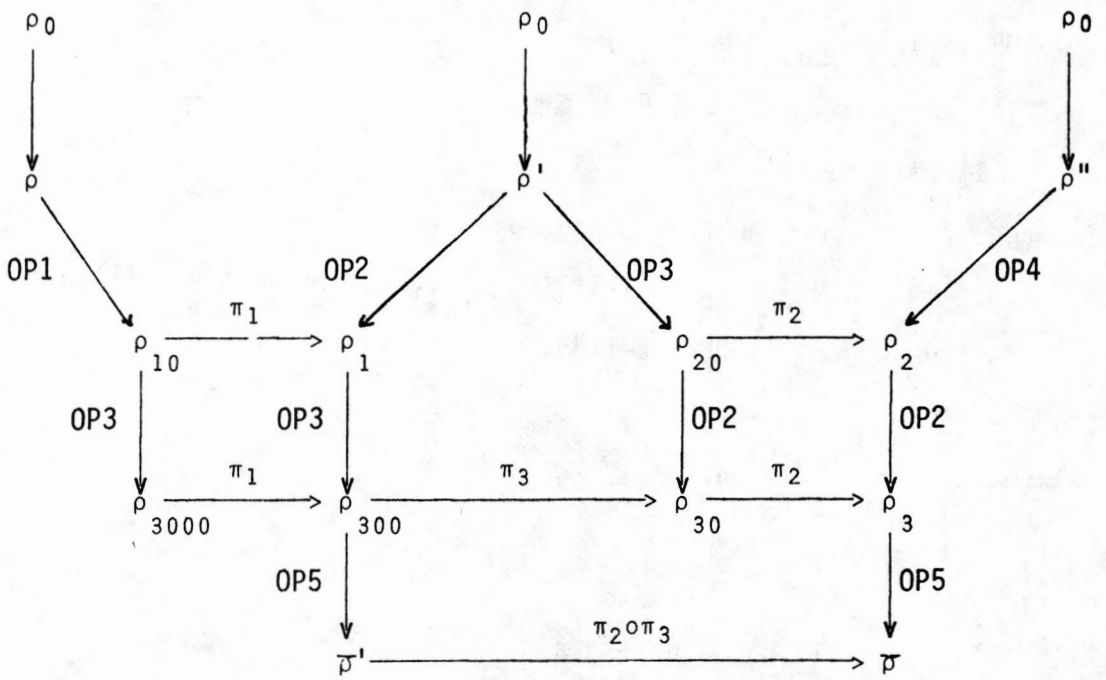
Mit (*) und der Transitivität von $\sim_s'$ erhält man dann:

$(\pi_2 \circ \pi_3 \circ \pi_1(l), \overline{\rho}) \sim_s' (l'', \overline{\rho})$

$\Longrightarrow \quad (l, \rho) \hat{=}_s (l'', \rho'')$

$\Longrightarrow \hat{=}_s$ ist transitiv für $se S \cup \{\bot\}$.

Obige Konstruktion läßt sich noch graphisch veranschaulichen:



Insgesamt ist also $\hat{=}$ eine Äquivalenz auf M_C.

MMM

3.4.11 Definition ($M_C/\hat{=}$)

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$ und $M_C := (\{A_s : se S \cup \{\bot\}\}, \{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra, die bool implementiere.

Dann: $M_C/\hat{=} := \{A_s/\hat{=}_s : se S \cup \{\bot\}\}$.

3.5.5 Definition (Korrekte Implementierung eines abstrakten Datentyps durch Clusters/Scripts)

Sei (S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster / Scripts mit $Erf(C,S,\Sigma)$. Sei M_C die von C erzeugte Modulalgebra, die bool implementiere und Σ enthalte.

Die Spezifikation (S,Σ,E) wird durch C *korrekt implementiert* :gdw

M_C erfüllt genau die Spezifikation (S,Σ,E).

Analog wird dann auch davon gesprochen, daß die Modulalgebra M_C korrekt die Spezifikation (S,Σ,E) implementiert.

3.6 Das Bild von T_Σ unter EVAL und von $T_{\Sigma,\sim E}$ unter $\overline{EVAL}$

3.6.1 Terme und Programme

Im folgenden sei (S,Σ,E) eine konsistente t-Spezifikation, $C:=\{A'_s:s\in S\}$ eine Cluster-/Script-Menge, die die Modulalgebra $M_C:=(\{A_s:s\in S\cup\{\perp\}\},\{\overline{P}_f: f\in\Sigma\})$ erzeugt und (S,Σ,E) korrekt implementiert.

Es wird informell gezeigt, daß es ausreicht, in M_C nur das Bild von T_Σ unter EVAL zu betrachten.

3.6.1.1 Sei $t\in T_\Sigma$ beliebig. Dann kann t aufgefaßt werden als ein abstraktes Programm. Da C die Spezifikation (S,Σ,E) korrekt implementiert, gelten insbesondere die syntaktischen Bedingungen an C aus 3.1. Dies bedeutet, daß jedem Funktionssymbol in t ein Prozedur-/Funktions-bzw. Operationsbezeichner in der jeweiligen Beispielprogrammiersprache entspricht, deren Aktivierungen geschachtelt sein können.

Mit dieser Zuordnung von Funktionssymbolen aus Σ zu Operationsaufrufen in der Programmiersprache und einer unterstellten Abarbeitung der Teilterme von $t\in T_\Sigma$ von links nach rechts kann einem Term $t\in T_\Sigma$ ein Programm P zugeordnet werden, das die Cluster bzw. Operationen aus C verwendet. Die Semantik des Programmes P erzeugt in M_C ein Element $(1,\rho)$, welches das Ergebnis von P und das Bild von t unter EVAL ist (vgl.3.4).

Damit entspricht einem Term $t\in T_\Sigma$ ein Programm P in einer der Beispielsprachen.

3.6.1.2 Sei nun P ein Programm in einer der Beispielsprachen, das ausschließlich Operationen P_f aufruft, die in den Clusters/Scripts aus C programmiert wurden und somit einem Funktionssymbol f der Signatur der Termalgebra T_Σ entsprechen.

Für P sollen die folgenden drei Randbedingungen gelten, die keine Einschränkung der Allgemeinheit bedeuten:

- P liefert genau ein Ergebnis eines Typs A'_s, $s\in S$, also keine Liste von Ergebnissen.
- P hat keine Eingabe.
  Jedem Programm mit den formalen Eingabeparametern par1,...,parn kann eine Klasse von Programmen ohne Eingabeparametern zugeordnet

werden, nämlich für jedes n-Tupel einer möglichen Liste von aktuellen Eingabeparametern ein Programm, in dem zunächst gerade diese n Exemplare von Datentypen erzeugt werden.
Faßt man P als Funktion auf, so gilt damit:

$$P: \longrightarrow A'_s \text{ für ein } s \in S.$$

- Im Laufe der Berechnung von P werden nur solche Elemente erzeugt, die zur Berechnung des Ergebnisses von P benötigt werden.

Unter diesen Voraussetzungen kann der Schachtelung von Aufrufen der Operationen der Cluster/Scripts aus C, die den Code des Programms P ausmachen, ein Term $t \in T_\Sigma$ zugeordnet werden, der eine Schachtelung der entsprechenden Funktionssymbole der Signatur (S,Σ) dastellt.

Damit entspricht dem Programm P ein Term $t \in T_\Sigma$, so daß die Interpretation des Programmergebnisses gleich EVAL(t) ist.

3.6.1.3 Faßt man die Überlegungen aus 3.6.1.1 und 3.6.1.2 zusammen, so erkennt man, daß es genügt, das Bild von T_Σ unter EVAL in M_C bzw. das Bild von $T_{\Sigma,\sim_E}$ unter $\overline{EVAL}$ in $M_C/_{\hat{=}}$ zu betrachten.

3.6.2 Das Bild von $T_{\Sigma,\sim_E}$ unter $\overline{EVAL}$ als Σ-Algebra

In 3.4 wurde von der potentiellen Verhaltensgleichheit $\hat{=}$ lediglich gefordert, daß sie eine Äquivalenz auf M_C ist, d.h.auf die Kongruenzeigenschaft von $\hat{=}$ wurde verzichtet.Dies hat zur Folge, daß $M_C/_{\hat{=}}$ keine algebraische Struktur besitzt.
In dem Fall, daß eine Modulalgebra M_C aber eine Spezifikation (S,Σ,E) korrekt implementiert, kann dem hier interessanten Teil von $M_C/_{\hat{=}}$, nämlich dem Bild von $T_{\Sigma,\sim_E}$ unter $\overline{EVAL}$, wieder eine algebraische Struktur unterlegt werden.

3.6.2.1 Definition/Lemma $(\overline{M}_C)$

Sei $SP:=(S,\Sigma,E)$ eine konsistente t-Spezifikation, C eine Menge von Cluster /Scripts und $M_C:=(\{A_s : s \in S \cup \{\bot\}\},\{\overline{P}_f : f \in \Sigma\})$ die von C erzeugte Modulalgebra, die SP korrekt implementiert.

1. $M_{C,\overline{EVAL}} := bd(\overline{EVAL})$

$$\overline{A}_s := bd(\overline{EVAL}) \cap A_s/_{\hat{=}} \quad , s \in S$$

2. $(\forall n\in N_0)(\forall s1,\ldots,sn,s\in S)(\forall f\in\Sigma_{s1\ldots sn,s})$

Setze:

$$\overline{\overline{P}}_f:\overline{A}_{s1}\times\ldots\times\overline{A}_{sn}\longrightarrow\overline{A}_s$$

$$\overline{\overline{P}}_f([l1,\rho 1]_{\cong},\ldots,[ln,\rho n]_{\cong}):=\overline{EVAL}([f(t1,\ldots,tn)]_{\sim_E})$$

wobei

$$([li,\rho i]_{\cong}\in\overline{A}_{si}\wedge[li,\rho i]_{\cong}=\overline{EVAL}([ti]_{\sim_E}),ti\in T_{\Sigma,si})\ i=1,..,n.$$

Dann ist $\overline{M}_C := (\{\overline{A}_s: s\in S\},\{\overline{\overline{P}}_f: f\in\Sigma\})$ eine Σ-Algebra.

Beweis:

Da $T_{\Sigma,\sim_E}$ eine Σ-Algebra ist und $\overline{EVAL}$ nach Voraussetzung injektiv ist, ist $\overline{\overline{P}}_f$, $f\in\Sigma$, wohldefiniert und $\overline{M}_C$ eine Σ-Algebra.

MMM

3.6.2.2 Corollar

Sei SP=(S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster/Scripts und M_C die von C erzeugte Modulalgebra, die SP korrekt implementiert.
Dann ist $\overline{EVAL}$: $T_{\Sigma,\sim_E}\longrightarrow\overline{M}_C$ ein Σ-Isomorphismus.

Beweis:

Nach Voraussetzung ist $\overline{EVAL}$: $T_{\Sigma,\sim_E}\longrightarrow\overline{M}_C$ bijektiv.
$\overline{EVAL}$ ist Σ-Homomorphismus:

- Sei $f\in\Sigma_{\varepsilon,s}$, $s\in S$
  ==> $\overline{\overline{P}}_f:\longrightarrow\overline{A}_s$, $\overline{\overline{P}}_f=\overline{EVAL}([f]_{\sim_E})$
- Sei $n\in N$, $s1,\ldots,sn,s\in S$, $f\in\Sigma_{s1\ldots sn,s}$; $[ti]_{\sim_E}\in T_{\Sigma,\sim_E}$, $i=1,..,n$.

$$\overline{EVAL}(f_{T_{\Sigma,\sim_E}}([t1]_{\sim_E},\ldots,[tn]_{\sim_E}))=\overline{EVAL}([f(t1,\ldots,tn)]_{\sim_E})$$
$$=\overline{\overline{P}}_f(\overline{EVAL}([t1]_{\sim_E},\ldots,\overline{EVAL}([tn]_{\sim_E}))\text{ nach 3.6.2.1.}$$

Damit ist $\overline{EVAL}$: $T_{\Sigma,\sim_E}\longrightarrow\overline{M}_C$ ein Σ-Isomorphismus.

MMM

Es liegt nun nahe, eine Art Verträglichkeit der potientiellen Verhaltensgleichheit bzgl. $\overline{P}_f$ und $\overline{\overline{P}}_f$, $f\in\Sigma$, zu fordern.
Daß dies tatsächlich gilt, zeigt folgendes Corollar.

3.6.2.3 Corollar

Sei SP:=(S,Σ,E) eine konsistente t-Spezifikation, C eine Menge von Cluster /Scripts und M_C die von C erzeugte Modulalgebra, die SP korrekt implementiert.
Dann gilt:

1. $(\forall s\in S)(\forall f\in\Sigma_{\varepsilon,s})$
   $\overline{\overline{P}}_f = [\overline{P}_f((\perp,\rho_0))]_{\cong}$,wo ρ_0 die Standardumgebung ist.
2. $(\forall n\in N)(\forall s1,\ldots,sn,s\in S)(\forall f\in\Sigma_{s1\ldots sn,s})$
   $(\forall ti\in T_{\Sigma,si}: (li,\rho i):= EVAL(ti),i=1,\ldots,n)$
   $\overline{\overline{P}}_f([l1,\rho 1]_{\cong},\ldots,[ln,\rho n]_{\cong}) = [\overline{P}_f((l1,\rho),\ldots,(ln,\rho))]_{\cong}$
   wobei $(li,\rho i'):= EV(t1\ldots ti-1,ti)$ $i=1,\ldots,n$
   $\rho:=\rho n'$

Beweis:

1. $\overline{\overline{P}}_f = \overline{EVAL}([f]_{\sim_E}) = [EVAL(f)]_{\cong} = [\overline{P}_f((\perp,\rho_0))]_{\cong}$ nach 3.6.2.1, 3.5.2 und 3.3.1.
2. Sei $n\in N$, $s1,\ldots,sn,s\in S$, $f\in\Sigma_{s1\ldots sn,s}$)
   Seien $ti\in T_{\Sigma,si}$, $(li,\rho i):= EVAL(ti)$ $i=1,\ldots,n$
   Dann sei $\overline{P}_f((l1,\rho),\ldots,(ln,\rho)):= EVAL(f(t1,\ldots,tn))$ nach 3.3.1.
   $\overline{\overline{P}}_f([l1,\rho 1]_{\cong},\ldots,[ln,\rho n]_{\cong}) = \overline{EVAL}([f(t1,\ldots,tn)]_{\sim_E})$ nach 3.6.2.1.
   $\overline{EVAL}([f(t1,\ldots,tn)]_{\sim_E}) = [EVAL(f(t1,\ldots,tn))]_{\cong}$
   $= [\overline{P}_f((l1,\rho),\ldots,(ln,\rho)]_{\cong}$ nach 3.3.1.

   $\Longrightarrow \overline{\overline{P}}_f([l1,\rho 1]_{\cong},\ldots,[ln,\rho n]_{\cong}) = [\overline{P}_f((l1,\rho),\ldots,(ln,\rho)]_{\cong}$.

MMM

3.7 Erweiterungen auf der Ebene der Modulalgebren

Für die folgenden Abschnitte seien SP:=(S,Σ,E) und SP':=(S',Σ',E') konsistente t-Spezifikationen, wobei SP' eine t-Erweiterung von SP ist.
C und C' seien Cluster-/Script-Mengen, so daß für die zugehörigen Modulalgebren M_C:=($\{A_s: s\in S\cup\{\perp\}\},\{\overline{P}_f: f\in\Sigma\}$) und $M_{C'}$:=($\{B_s: s\in S'\cup\{\perp'\}\},\{\overline{P}_{f'}: f'\in\Sigma'\}$) gilt: M_C implementiert korrekt SP und $M_{C'}$ implementiert korrekt SP'.

Die Frage ist nun, wie verhält sich M_C zu $M_{C'}$ bzw. $\overline{M}_C$ zu $\overline{M}_{C'}$?
Zu wünschen wäre, daß sich M_C in $M_{C'}$ und $\overline{M}_C$ in $\overline{M}_{C'}$ einbetten lassen analog den Einbettungen von T_Σ in $T_{\Sigma'}$ bzw. von $T_{\Sigma,\sim_E}$ in $T_{\Sigma';\sim_{E'}}$.

3.7.1 Da $S \subseteq S'$ und $\Sigma \subseteq \Sigma'$ gilt, existiert ein eindeutiger Σ-Monomorphismus

$$in: T_\Sigma \longrightarrow T_{\Sigma'}$$

nämlich:

- $(\forall s \in S)(\forall f \in \Sigma_{\varepsilon,s})(in(f_{T_\Sigma}) := f_{T_{\Sigma'}})$
- $(\forall n \in N)(\forall s1,\ldots,sn,s \in S)(\forall f \in \Sigma_{s1\ldots sn,s})$
  $(\forall (t1,\ldots,tn) \in T_{\Sigma,s1} \times \ldots \times T_{\Sigma,sn})$
  $(in(f_{T_\Sigma}(t1,\ldots,tn)) := f_{T_{\Sigma'}}(in(t1),\ldots,in(tn)))$

Es ist nach Definition klar, daß 'in' ein eindeutiger Σ-Monomorphismus ist.

3.7.2 Da SP' eine t-Erweiterung von SP ist, existiert ein (eindeutiger) Σ-Monomorphismus

$$\overline{in}: T_{\Sigma,\sim_E} \longrightarrow T_{\Sigma';\sim_{E'}}.$$

Mit den kanonischen Abbildungen

$$\kappa_E: T_\Sigma \longrightarrow T_{\Sigma,\sim_E},\ t \longmapsto [t]_{\sim_E}$$

und

$$\kappa_{E'}: T_{\Sigma'} \longrightarrow T_{\Sigma';\sim_{E'}},\ t \longmapsto [t]_{\sim_{E'}}$$

gilt:

$$\kappa_{E'} \circ in = \overline{in} \circ \kappa_E.$$

Beweis:

Führe den Beweis durch strukturelle Induktion über $t \in T_\Sigma$.

a) Sei $s \in S$, $t := f \in \Sigma_{\varepsilon,s}$.
$\overline{in}(\kappa_E(t)) = \overline{in}([f_{T_\Sigma}]_{\sim_E}) = [f_{T_{\Sigma'}}]_{\sim_{E'}} = [in(f_{T_\Sigma})]_{\sim_{E'}} = \kappa_{E'}(in(t))$.

b) Sei $n \in N$, $s1,\ldots,sn,s \in S$, $f \in \Sigma_{s1\ldots sn,s}$.
Sei $(t1,\ldots,tn) \in T_{\Sigma,s1} \times \ldots \times T_{\Sigma,sn}$.
Nach Induktionsvoraussetzung gilt: $\overline{in}(\kappa_E(ti)) = \kappa_{E'}(in(ti))$ $i=1,\ldots,n$.

$$\begin{aligned}
\Longrightarrow \overline{in}(\kappa_E(f_{T_\Sigma}(t1,\ldots,tn))) &= \overline{in}([f_{T_\Sigma}(t1,\ldots,tn)]_{\sim_E}) \\
&= \overline{in}(f_{T_{\Sigma,\sim_E}}([t1]_{\sim_E},\ldots,[tn]_{\sim_E})) \\
&= f_{T_{\Sigma';\sim_{E'}}}(\overline{in}([t1]_{\sim_E}),\ldots,\overline{in}([tn]_{\sim_E})) \\
&= f_{T_{\Sigma';\sim_{E'}}}(\kappa_{E'}(in(t1)),\ldots,\kappa_{E'}(in(tn)))
\end{aligned}$$

$$= \kappa_{E'}(f_{T_{\Sigma'}}(in(t1),\ldots,in(tn)))$$
$$= \kappa_{E'}(in(f_{T_{\Sigma}}(t1,\ldots,tn)))$$

Damit gilt: $\kappa_{E'} \circ in = \overline{in} \circ \kappa_E$.

MMM

3.7.3 Da M_C SP bzw. $M_{C'}$ SP' korrekt implementieren, existieren

$$EVAL: T_\Sigma \longrightarrow M_C \quad , \qquad \overline{EVAL}: T_{\Sigma,\sim_E} \longrightarrow M_C/_{\cong}$$
$$EVAL': T_{\Sigma'} \longrightarrow M_{C'} \quad , \qquad \overline{EVAL}': T_{\Sigma',\sim_{E'}} \longrightarrow M_{C'}/_{\cong}$$
$$\kappa: M_C \longrightarrow M_C/_{\cong} \quad , \qquad (l,\rho) \longmapsto [l,\rho]_{\cong}$$
$$\kappa': M_{C'} \longrightarrow M_{C'}/_{\cong} \quad , \qquad (l,\rho) \longmapsto [l,\rho]_{\cong}$$

und es gilt:
$$\kappa \circ EVAL = \overline{EVAL} \circ \kappa_E$$
$$\kappa' \circ EVAL' = \overline{EVAL}' \circ \kappa_{E'}$$

3.7.4 Es wird nun eine Abbildung angegeben, die die Modulalgebra M_C unter bestimmten Voraussetzungen einbettet in $M_{C'}$.

$$IN := \{IN_s : s \in S \cup \{\bot\}\} : M_C \longrightarrow M_{C'}$$

IN0: $(\forall s \in S)\ IN_s(\bot_s) := \bot'_s$

IN1: $IN_\bot((\bot,\rho_0)) := (\bot',\rho_0')$ wobei ρ_0 die Standardumgebung von M_C und ρ_0' die von $M_{C'}$ ist.

IN2: a) $(\forall n \in \mathbb{N})(\forall s \in S)(\forall \overline{P}_f \in OP(n,s))$
$(\forall((l1,\rho),\ldots,(ln,\rho)) \in (dom(IN)^n \cap dom(\overline{P}_f)))$

$IN(\overline{P}_f((l1,\rho),\ldots,(ln,\rho))) := \overline{P}_{f'}(IN_{s1}((l1,\rho)),\ldots,IN_{sn}((ln,\rho)))$

mit $in(f(t1,\ldots,tn)) = f'(in(t1),\ldots,in(tn))$, $ti \in T_{\Sigma,si}$ $i=1,..,n$.

b) $(\forall s \in S)(\forall \overline{P}_f \in OP(0,s))(\forall a \in A_\bot)$

$IN(\overline{P}_f(a)) := \overline{P}_{f'}(IN_\bot(a))$ wobei $f' := in(f)$

IN3: $(\forall n \in \mathbb{N}_0)(\forall s \in S)(\forall \overline{P}_f \in OP(n,s))(\forall((l1,\rho),\ldots,(ln,\rho)) \in dom(\overline{P}_f))$

Sei $(l,\rho') := \overline{P}_f((l1,\rho),\ldots,(ln,\rho))$,

$(\overline{l},\overline{\rho}') := \overline{P}_{f'}((\overline{l1},\overline{\rho}),\ldots,(\overline{ln},\overline{\rho})) := IN(\overline{P}_f((l1,\rho),\ldots,(ln,\rho)))$

dann:

a) $IN_\bot((\bot,\rho') := (\bot,\overline{\rho}')$

b) Sei $(l',\rho) \in A_s$, $s \in S$, und $(l',\rho') \in A_s$ per Lifting; dann:

$IN_s((l',\rho')) := (IN((l',\rho))\downarrow 1, IN((l,\rho'))\downarrow 2)$
$= (IN((l',\rho))\downarrow 1, \overline{\rho}')$

Je nachdem, wie der Programmierer die Cluster-/Scriptmenge C und C' programmiert hat, kann es möglich sein, daß IN nicht wohldefiniert ist (Widersprüche zwischen IN2 und IN3b sind möglich).
An den Programmierer ist also die Forderung zu richten, C und C' so zu programmieren, daß IN wohldefiniert und injektiv ist. Dies kann etwa durch $C \underline{c} C'$ erreicht werden.

3.7.4.1 Definition ($M_{C'}$ erweitert M_C)

Seien $SP:=(S,\Sigma,E)$ und $SP':=(S',\Sigma',E')$ konsistente t-Spezifikationen, so daß SP' eine t-Erweiterung von SP ist.
Seien C und C' Cluster-/Scriptmengen und M_C bzw. $M_{C'}$ die von C und C' erzeugten Modulalgebren, die SP bzw. SP' korrekt implementieren.
$M_{C'}$ *erweitert* M_C :gdw IN: $M_C \longrightarrow M_{C'}$ ist wohldefiniert und injektiv.

Falls IN wohldefiniert ist, gilt:

$$IN \circ EVAL = EVAL' \circ in$$

d.h. die Einbettung der Termalgebra kommutiert mit der Einbettung der Modulalgebra.

Beweis:
In der Definition von $EVAL(t)$ bzw. $EV(\varepsilon,t)$, $te T_\Sigma$, (vgl.3.3) werden folgende 3 Schritte (u.U. mehrmals) angewendet:

- Abbildung einer Konstanten $fe\Sigma_{\varepsilon,s}$, seS, in einem bestimmten Kontext $t1 \ldots tn e T_\Sigma^*$
- Lifting
- Operationsanwendung.

Es wird nun gezeigt, daß bei jedem dieser Schritte die Gleichung $IN \circ EV = EV' \circ in$ gültig bleibt; es wird also eine Induktion geführt über die Definition von EV.

a) Konstanten:
Sei seS, $fe\Sigma_{\varepsilon,s}$, $f':=in(f)$.
Induktion über den Kontext von EV:
- leerer Kontext: $IN(EV(\varepsilon,f)) = IN(\overline{P}_f((\perp,\rho_0))) = \overline{P}_{f'}((\perp',\rho_0')) = EV'(\varepsilon,f')$
- beliebiger Kontext $\neq \varepsilon$:
  Sei $n e N_0$, $ti e T_{\Sigma,si}$ mit $in(ti)=:\overline{ti}$ für $i=1,\ldots,n+1$.
  Sei $(1,\rho) := EV(t1 \ldots tn, tn+1)$
  $(\overline{1},\overline{\rho}) := EV'(\overline{t1} \ldots \overline{tn},\overline{tn+1})$
  Sei $IN((\perp,\rho)) =: (\perp',\overline{\rho})$ nach Induktionsvoraussetzung.

Dann gilt:

$IN(EV(t1...tn+1,f)) = IN(\overline{P}_f((\perp,\rho))) = \overline{P}_{f'}(IN_{\perp}((\perp,\rho)))$ (IN2)

$= \overline{P}_{f'}((\perp',\overline{\rho})) = EV'(in(t\overline{1})...in(tn+1),in(f))$ (IN3a)

b) Lifting:

Sei neN_0, $tieT_{\Sigma,si}$ mit $in(ti)=:\overline{ti}$ für i=1,...,n+1.

Seien $(a,\rho):= EV(t1...tn,tn+1)$, $(\overline{a},\overline{\rho}) := EV'(\overline{t1}...\overline{tn},\overline{tn+1})$,

$IN((a,\rho))= (\overline{a},\overline{\rho})$ nach Induktionsvoraussetzung.

Sei teT_{Σ} mit $in(t)=:\overline{t}$;

$(a',\rho'):= EV(t1...tn+1,t)$, $(\overline{a}',\overline{\rho}'):= EV'(\overline{t1}...\overline{tn+1},\overline{t})$,

$IN((a',\rho')) = (\overline{a}',\overline{\rho}')$ nach Induktionsvoraussetzung.

Seien (a,ρ')in M_C und $(\overline{a},\overline{\rho}')$ in $M_{C'}$ per Lifting.

Dann gilt nach IN3a und nach Voraussetzung:

$IN((a,\rho') = (IN(EV(t1...tn,tn+1))\downarrow 1, IN(EV(t1...tn+1,t))\downarrow 2)$

$= (\overline{a},\overline{\rho}')$.

c) Operationsanwendung:

Beweis induktiv über teT_{Σ}.

- $t=fe\Sigma_{\varepsilon,s}$, seS: vgl a).
- Seien neN, meN_0, seS; $tieT_{\Sigma,si}$ und $\overline{ti}:=in(ti)$ für i=1,...,m, feOP(n,s).
  Seien $ti'eT_{\Sigma,si}$, i=1,...,n; $f'(\overline{t1}',...,\overline{tn}'):= in(f(t1',...,tn'))$.
  Sei $\overline{P}_f((a1,\rho),...,(an,\rho)):= EV(t1...tm,f(t1'...tn'))$,
  $\overline{P}_{f'}((\overline{a1},\overline{\rho}),...,(\overline{an},\overline{\rho})):= EV'(\overline{t1}...\overline{tm},f'(\overline{t1}',...,\overline{tn}'))$
  Nach Induktionsvoraussetzung sei $IN((ai,\rho)) = (\overline{ai},\overline{\rho})$ für i=1,...,n.

  Dann gilt nach IN2:

$$\begin{aligned} IN(EV(t1...tm,f(t1,...,tn))) &= IN(\overline{P}_f((a1,\rho),...,(an,\rho)) \\ &= \overline{P}_{f'}(IN((a1,\rho)),...,IN((an,\rho))) \\ &= \overline{P}_{f'}((\overline{a1},\overline{\rho}),...,(\overline{an},\overline{\rho})) \\ &= EV'(\overline{t1}...\overline{tm},f'(\overline{t1}',...,\overline{tn}')) \\ &= EV'(in(t1),..,in(tm),in(f(t1',..,tn'))) \end{aligned}$$

Damit gilt : $IN \circ EV = EV' \circ in$, insbesondere also

$IN \circ EVAL = EVAL' \circ in$.

MMM

Auf eine Abbildung zwischen zwei $\Sigma_{\perp}$-Algebren, wo $(S_{\perp},\Sigma_{\perp})=(S,\Sigma)_{\perp}$, kann der Homomorphiebegriff aus 1.1.1.6 nicht angewendet werden, da für diese Algebren keine konstante Funktionen definiert wurden . In der folgenden Definition wird daher der Homomorphiebegriff eingeführt für Abbildungen zwischen $\Sigma_{\perp}$-Algebren.

3.7.4.2 Definition ($\Sigma_\perp$-Homomorphismus)

Sei (S,Σ) eine Signatur, $\perp \notin S$ und $(S_\perp,\Sigma_\perp):=(S,\Sigma)_\perp$ (vgl.3.2.4).
Seien A,B $\Sigma_\perp$-Algebren und $H:=\{h_s: h_s: A_s \longrightarrow B_s,\ s\in S\cup\{\perp\}\}: A \longrightarrow B$.
H heißt $\Sigma_\perp$-*Homomorphismus* :gdw

1. $(\forall s\in S)(\forall f\in\Sigma_{\varepsilon,s})(\forall a\in A_\perp)$
   $H(f_A(a)) = f_B(h_\perp(a))$
2. $(\forall n\in N)(\forall s1,\ldots,sn,s\in S)(\forall f\in\Sigma_{s1\ldots sn,s})$
   $(\forall(a1,\ldots,an)\in A_{s1}x\ldots x\ A_{sn})$
   $H(f_A(a1,\ldots,an)) = f_B(h_{s1}(a1),\ldots,h_{sn}(an))$

Beachtenswert ist, daß außer der Bedingung 1. über $h_\perp$ keine Aussagen gemacht werden.

Mit diesem Homomorphiebegriff für $\Sigma_\perp$-Algebren gilt nun:

IN: $M_C \longrightarrow M_{C'}|_{\Sigma_\perp}$ ist $\Sigma_\perp$-Homomorphismus, falls $M_{C'}$ M_C erweitert.

Beweis:

1. Sei $s\in S$, $f\in\Sigma_{\varepsilon,s}$ $\Longrightarrow \overline{P}_f\in OP(0,s)$.
   Sei $a\in A_\perp$. Nach IN2b gilt dann:
   $IN(\overline{P}_f(a)) = \overline{P}_{f'}(IN_\perp(a))$, wobei $f':=in(f)$.
2. Sei $n\in N;s1,\ldots,sn,s\in S$, $f\in\Sigma_{s1\ldots sn,s}$ $\Longrightarrow \overline{P}_f\in OP(n,s)$.
   Sei $(a1,\ldots,an)\in A_{s1}\ x\ldots x\ A_{sn}$.
   Dann gilt nach IN2a:
   $IN(\overline{P}_f(a1,\ldots,an)) = \overline{P}_{f'}(IN_{s1}(a1),\ldots,IN_{sn}(an))$
   mit $f'(t1,..,tn'):=in(f(t1,\ldots,tn))$,
   $(t1,\ldots,tn)\in T_{\Sigma,s1}x\ldots x\ T_{\Sigma,sn}$.

Damit ist IN: $M_C \longrightarrow M_{C'}|_{\Sigma_\perp}$ ein $\Sigma_\perp$-Homomorphismus.

⋈⋈⋈

3.7.5 Analog der von 'in' auf $T_{\Sigma,\sim_E}$ induzierten Abbildung $\overline{in}$ (vgl. 3.7.2) wird nun eine Abbildung auf $\overline{M}_C$ definiert:

$\overline{IN}: \overline{M}_C \longrightarrow M_{C'}/_{\cong}$

$$(\forall s \in S)(\forall [l,\rho]_{\hat{=}} \in \overline{A}_s)$$

$$\overline{IN}([l,\rho]_{\hat{=}}) := \overline{EVAL}' \circ \overline{in} \circ \overline{EVAL}^{-1}([l,\rho]_{\hat{=}})$$

$$= \overline{EVAL}'(\overline{in}(\overline{EVAL}^{-1}([l,\rho]_{\hat{=}})$$

$\overline{IN}$ ist wohldefiniert, da $\overline{EVAL}$ bijektiv auf seinem Bild ist.

Da $\overline{EVAL}^{-1}$, $\overline{in}$ und $\overline{EVAL}'$ Σ-Monomorphismen sind, ist auch $\overline{IN}$ ein Σ-Monomorphismus (vgl.3.6.2).

Nach Definition gilt: $\overline{EVAL}' \circ \overline{in} = \overline{IN} \circ \overline{EVAL}$.

$\overline{IN}$ existiert unabhängig von IN, d.h. $M_{C'}$ muß nicht unbedingt M_C erweitern.

3.7.6 Analog der Vertauschbarkeit der Einbettungen in und $\overline{in}$ gilt für IN und $\overline{IN}$, falls $M_{C'}$, M_C erweitert:

$$\overline{IN} \circ \kappa_{EVAL} = \kappa' \circ IN_{EVAL}$$

wobei κ_{EVAL} bzw. IN_{EVAL} die Einschränkung von κ bzw. IN auf das Bild von M_C unter EVAL ist.

Beweis:

Sei $(l,\rho) \in bd(EVAL) \Longrightarrow (\exists t \in T_\Sigma)(EVAL(t) = (l,\rho))$

$\Longrightarrow \kappa'(IN((l,\rho))) = \kappa'(IN(EVAL(t)))$

$= \kappa'(EVAL'(in(t)))$ (3.7.4)

$= \overline{EVAL}'(\kappa_{E'}(in(t)))$ (3.7.3)

$= \overline{EVAL}'(\overline{in}(\kappa_E(t)))$ (3.7.2)

$= \overline{EVAL}'(\overline{in}(\overline{EVAL}^{-1}(\kappa((l,\rho)))))$ (3.7.3)

$= \overline{IN}(\kappa((l,\rho)))$ (3.7.5)

MMM

Nach 3.7.5 ist klar, daß $bd(\overline{IN}) \subseteq bd(\overline{EVAL}') = \overline{M}_{C'}$.

Im folgenden Satz werden nun die Ergebnisse aus 3.7.1 bis 3.7.6 zusammengefaßt. Dieser Satz bringt nochmals deutlich die Parallelität zwischen der Konstruktion einer Termalgebra T_Σ und einer Modulalgebra M_C zum Ausdruck.

3.7.7 Satz

Seien SP:=(S,Σ,E)und SP':=(S',Σ',E') konsistente t-Spezifikationen, so daß SP' eine t-Erweiterung von SP ist. $(S_\perp,\Sigma_\perp):=(S,\Sigma)_\perp$.
Seien C und C' Cluster-/Scriptmengen und $\overline{M}_C$ bzw. $\overline{M}_{C'}$ die von C bzw C' erzeugten Modulalgebren und M_C bzw. $M_{C'}$ implementiere korrekt SP bzw. SP'.
Dann gilt:

1. Es existiert ein Σ-Monomorphismus

$$\overline{IN}: \overline{M}_C \longrightarrow \overline{M}_{C'}$$

d.h. $\overline{M}_C$ läßt sich in $\overline{M}_{C'}$ einbetten.

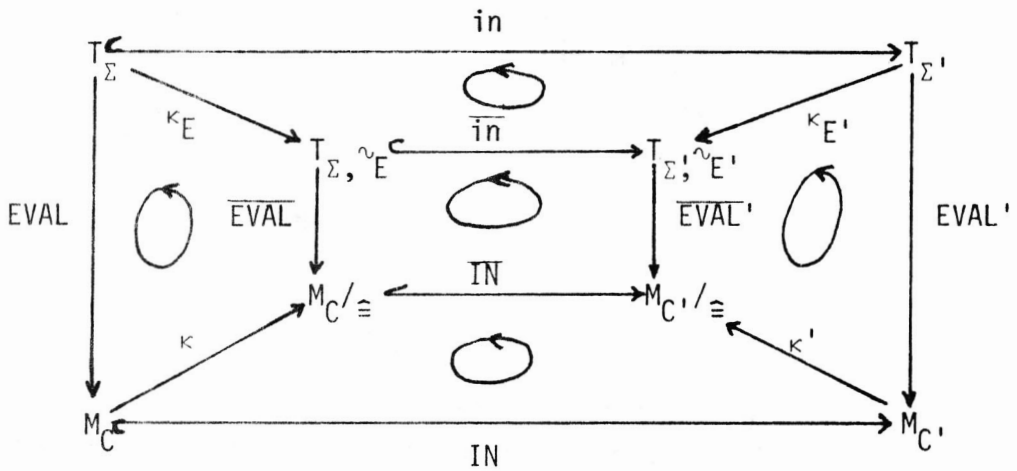
2. Es existieren

$$\begin{aligned}
&in: T_\Sigma \longrightarrow T_{\Sigma'} && \Sigma\text{-monomorph}\\
&\overline{in}: T_{\Sigma,\sim E} \longrightarrow T_{\Sigma';\sim E'} && \Sigma\text{-monomorph}\\
&\kappa_E: T_\Sigma \longrightarrow T_{\Sigma,\sim E} && \Sigma\text{-epimorph}\\
&\kappa_{E'}: T_{\Sigma'} \longrightarrow T_{\Sigma';\sim E'} && \Sigma'\text{-epimorph}\\
&\kappa: M_C \longrightarrow M_C/_{\cong}\\
&\kappa': M_{C'} \longrightarrow M_{C'}/_{\cong}
\end{aligned}$$

so daß in dem Fall, daß $M_{C'}$, M_C erweitert,

$$IN: M_C \longrightarrow M_{C'} \qquad \Sigma_\perp\text{- monomorph existiert,}$$

so daß die folgenden Diagramme kommutieren:



Beweis:

a) Vgl. 3.7.5.

b) Vgl. 3.7.1 - 3.7.6.

⋈⋈⋈

4. KORREKTE IMPLEMENTIERUNG EINES ABSTRAKTEN TYP-PARAMETRISIERTEN DATENTYPS DURCH CLUSTER

In diesem Kapitel wird der Implementierungsbegriff aus Kapitel 3 erweitert für den Fall von typparametrisierten Datentypen.
Hierzu bedarf es zunächst einer algebraischen Theorie abstrakter typparametrisierter Datentypen. Diese wird in 4.1 entwickelt. Dabei wird nicht der für den initialen Fall bereits vorliegende Ansatz der ADJ-Gruppe ([ADJ 78], [EKTWW 79]) übernommen, sondern der in [HOR 80], der sich gleichermaßen im initialen wie im terminalen Fall anwenden läßt.

In 4.2 werden die Konzepte zur Typparametrisierung von Clusters, Forms und Scripts in CLU, ALPHARD bzw. CSSA untersucht. Im weiteren werden dann nur noch CLU und ALPHARD betrachtet, da die Parametrisierung in der gegenwärtigen Version von CSSA noch erhebliche Mängel aufweist (diese sollen in der zur Zeit in Entwicklung befindlichen Version CSSA-M beseitigt werden).

In 4.3 wird dann gezeigt, daß aufgrund der Ähnlichkeit von CLU und ALPHARD beide Sprachen wieder durch dieselbe abstrakte Syntax und denotationale Semantik beschrieben werden können (parametrisierte Version von CLALPHARD).

In 4.4 wird schließlich definiert, wann eine Clustermenge eine parametrisierte Datentypspezifikation korrekt implementiert, nachdem an die Syntax dieser Cluster, ausgehend von der Signatur einer Spezifikation, Forderungen entwickelt und gestellt wurden.

4.1 Abstrakte parametrisierte Datentypen

Ein parametrisierter Datentyp wird hier nicht als ein Datentyp, d.h. als eine Algebra bzw. Isomorphieklasse von Algebren aufgefaßt, sondern als eine Transformation, die die Algebra einer aktuellen Parameterspezifikation überführt in die zugehörige Ergebnisalgebra des aktuell parametrisierten Datentyps. Gesucht ist also ein Funktor zwischen der Kategorie der zulässigen Algebren der aktuellen Parameterspezifikationen und der Kategorie der zugehörigen aktuell parametrisierten Datentypen.

4.1.1 Definition (Kombination)

Sei (S,Σ) eine Signatur.
Sei $S' \cap S = \emptyset$, $\Sigma' : \subseteq \{\Sigma^{o}_{w,s} : we(S \cup S')^*, se S \cup S', (\forall ue S^*)(\forall re S)(\Sigma_{u,r} \cap \Sigma^{o}_{u,r} = \emptyset\}$

1. $(\overline{S},\overline{\Sigma}) := (S,\Sigma) + (S',\Sigma') := (S \cup S', \Sigma \cup \Sigma')$ ist die *Kombination* von (S,Σ) und (S',Σ') und es gilt dann: $(S,\Sigma) \leq (\overline{S},\overline{\Sigma})$.

2. Sei E eine Menge von Gleichungen über (S,Σ), E' eine Menge von Gleichungen über $(S \cup S', \Sigma \cup \Sigma')$, wobei $(S,\Sigma) \leq (S \cup S', \Sigma \cup \Sigma')$.
   Dann ist $(\overline{S},\overline{\Sigma},\overline{E}) := (S,\Sigma,E) + (S',\Sigma', E') := (S \cup S', \Sigma \cup \Sigma', E \cup E')$
   die *Kombination* von (S,Σ,E) und (S',Σ',E').

Bei einem parametrisierten Datentypen werden nun vier Spezifikationen unterschieden:

(S,Σ,E)	Spezifikation des formalen Parameters
$(S,\Sigma,E)+(S';\Sigma';E')$	Spezifikation des formal parametrisierten DT
$(S,\Sigma,E)+(S'';\Sigma'';E'')$	Spezifikation des aktuellen Parameters
$(S,\Sigma,E)+(S';\Sigma';E')+(S'';\Sigma'';E'')$	Spezifikation des aktuell parametrisierten DT

Zwischen den Spezifikationen bestehen somit folgende Zusammenhänge:

	Parameter	$+(S';\Sigma';E')$	parametrisierter Datentyp
formal	(S,Σ,E)		$(S,\Sigma,E)+(S';\Sigma';E')$
$+(S'';\Sigma'';E'')$			
aktuell	$(S,\Sigma,E)+(S'';\Sigma'';E'')$		$(S,\Sigma,E)+(S';\Sigma';E')+(S'';\Sigma'';E'')$

4.1.2 Definition (syntaktisch zulässig)

Seien (S,Σ), $(S \cup S', \Sigma \cup \Sigma')$ und $(S \cup S'', \Sigma \cup \Sigma'')$ Signaturen mit
$S \cap S' = S \cap S'' = S' \cap S'' = \Sigma \cap \Sigma' = \Sigma \cap \Sigma'' = \Sigma' \cap \Sigma'' = \emptyset$.
Dann heißt (S'',Σ'') *syntaktisch zulässig* für (S',Σ') über (S,Σ).

Notation: $SynZul(S'';\Sigma'';S,\Sigma,S';\Sigma')$.

4.1.3 Corollar

Seien (S,Σ), $(S\cup S',\Sigma\cup\Sigma')$ und $(S\cup S'',\Sigma\cup\Sigma'')$ Signaturen mit
$S\cap S'=S\cap S''=S'\cap S''=\Sigma\cap\Sigma'=\Sigma\cap\Sigma''=\Sigma'\cap\Sigma''=\emptyset$.
Dann gilt:

1. SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$ <==> SynZul$(S',\Sigma',S,\Sigma,S'',\Sigma'')$
2. SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$ ==> $(S\cup S'\cup S'',\Sigma\cup\Sigma'\cup\Sigma'')$ ist Signatur.

Der Beweis folgt unmittelbar aus 4.1.2.

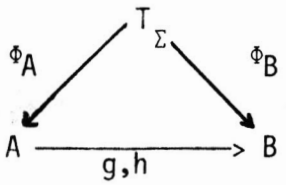
4.1.4 Lemma

Sei (S,Σ) eine Signatur, A,B Σ-erzeugte Σ-Algebren.

Dann gilt:
1. Es existiert höchstens ein Σ-Homomorphismus $g:A \longrightarrow B$.
2. Falls $g: A \longrightarrow B$, g Σ-homomorph, existiert, so ist g surjektiv.
3. Falls $g: A \longrightarrow B$, $h: B \longrightarrow A$ als Σ-Homomorphismen existieren, so ist $A\simeq B$.

Beweis:

1.



Seien $g,h: A \longrightarrow B$ Σ-Homomorphismen.
T_Σ ist initial in der Kategorie aller Σ-erzeugten Σ-Algebren
$\Phi_A: T_\Sigma \longrightarrow A$, $\Phi_B: T_\Sigma \longrightarrow B$ sind eindeutige Σ-Homomorphismen. Mit g und h sind auch $g\circ\Phi_A$, $h\circ\Phi_A: T_\Sigma \longrightarrow B$ Σ-Homomorphismen.
==> $g\circ\Phi_A = \Phi_B = h\circ\Phi_A$.
Sei nun $a\in A$ beliebig ==> es existiert $t\in T_\Sigma$ mit $\Phi_A(t)=a$, da A Σ-erzeugt ist.
==> $g(a) = g(\Phi_A(t)) = h(\Phi_A(t)) = h(a)$
==> $g = h$.

2. Sei $g: A \longrightarrow B$ Σ-homomorph.
A,B sind Σ-erzeugt ==> Φ_A,Φ_B surjektiv ==> g surjektiv.

3. Seien $g: A \longrightarrow B$, $h: B \longrightarrow A$ Σ-Homomorphismen.
Sei $a\in A$, $t\in T_\Sigma$ mit $\Phi_A(t)=a$
Dann gilt: $h\circ g(a) = h(g(a)) = h(g(\Phi_A(t))) = h(\Phi_B(t)) = \Phi_A(t) = a$
==> $h\circ g = id_A$
analog erhält man: $g\circ h = id_B$
==> $g = h^{-1}$
==> $A\simeq B$.

■

Im folgenden wird eine Theorie der abstrakten parametrisierten Datentypen entwickelt; zunächst für den Fall, daß den Datentypen eine initiale Semantik unterstellt wird, dann für den terminalen Fall.

4.1.5 Initialer Fall

4.1.5.1 Definition ($Alg'_{\Sigma,E}$)

Sei (S,Σ,E) eine Spezifikation.

$Alg'_{\Sigma,E}$ sei die Kategorie mit

- $|Alg'_{\Sigma,E}|$:= Klasse aller Σ-erzeugten Σ-Algebren A, die E erfüllen ($\equiv_E \subseteq \equiv_A$).
- $/Alg'_{\Sigma,E}/$:= Klasse aller zugehörigen Σ-Homomorphismen

4.1.5.2 Defintion / Lemma

Sei $\{A_i : i\in I\}$ eine Familie von Kategorien mit $|A_i|\cap|A_j| = /A_i/\cap/A_j/ = \emptyset$ für $i,j\in I, i\neq j$.

$\bigoplus_{i\in I} A_i$ ist eine Kategorie mit

- $|\bigoplus_{i\in I} A_i| := \bigcup_{i\in I} |A_i|$
- $/\bigoplus_{i\in I} A_i/ := \bigcup_{i\in I} /A_i/$

Da die einzelnen Objekt- und Morphismenmengen disjunkt sind, ist mit A_i, $i\in I$, auch $\bigoplus_{i\in I} A_i$ eine Kategorie.

4.1.5.3 Defintion (parametrisierte i-Spezifikation, (Σ,E,Σ',E')-i-Transformer, i-Trans(Σ,E,Σ',E'))

Seien (S,Σ,E), $(S\cup S',\Sigma\cup\Sigma',E\cup E')$ Spezifikationen mit $S\cap S' = \Sigma\cap\Sigma' = \emptyset$.

1. $Alg_{\Sigma,E,\Sigma'} := \bigoplus_{(S'',\Sigma'') \text{ mit } SynZul(S'',\Sigma'',S,\Sigma,S',\Sigma')} Alg'_{\Sigma\cup\Sigma'',E}$

2. $(S,\Sigma,E,S',\Sigma',E')$ heißt *parametrisierte i-Spezifikation* :gdw

$$(\forall(S'',\Sigma'')\ SynZul(S'',\Sigma'',S,\Sigma,S',\Sigma'))(\forall A e |Alg_{\Sigma,E,\Sigma'}|)$$
$$((\equiv_{E\cup E',A}|_{T_{\Sigma\cup\Sigma''}} \subseteq \equiv_A) \wedge ((\forall s e S)(\forall t e T_{\Sigma\cup\Sigma',\Sigma'',s})(\exists t' e T_{\Sigma\cup\Sigma'',s})(t \equiv_{E\cup E',A} t')))$$

wo $\equiv_{E\cup E',A}$ die kleinste Kongruenz auf $T_{\Sigma\cup\Sigma',\Sigma''}$ ist, die E, E' und die durch den initialen Epimorphismus $\Phi_A : T_{\Sigma\cup\Sigma''} \longrightarrow A$ induzierte Kongruenz $\equiv_A$ enthält.

3. Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation.

   3.1 Ein *(Σ,E,Σ',E')-i-Transformer* F ist ein persistenter Funktor

   $$F: Alg_{\Sigma,E,\Sigma'} \longrightarrow Alg_{\Sigma\cup\Sigma',E\cup E',\Sigma'} \quad \text{mit}$$

   $$(\forall(S'',\Sigma''):SynZul(S'',\Sigma'',S,\Sigma,S',\Sigma'))(\forall Ae|Alg_{\Sigma,E,\Sigma'}|\cap|Alg'_{\Sigma\cup\Sigma''}|)(F(A)e|Alg_{\Sigma\cup\Sigma',\Sigma\cup\Sigma''}|)$$

   3.2 *i-Trans(Σ,E,Σ',E')* ist die Kategorie, deren Objekte gerade die (Σ,E,Σ',E')-i-Transformer sind und deren Morphismen die natürlichen Transformationen dieser Funktoren sind.

4. Sei (S'',Σ'') mit $SynZul(S'',\Sigma'',S,\Sigma,S',\Sigma')$, $Ae|Alg_{\Sigma\cup\Sigma'',E}|$.

   $T := T_{\Sigma\cup\Sigma'\cup\Sigma''}$

   $\hat{=}_A := \equiv_{E\cup E',A}$.

Eine parametrisierte i-Spezifikation $(S,\Sigma,E,S',\Sigma',E')$ muß also so beschaffen sein, daß für alle Algebren $Ae|Alg_{\Sigma,E,\Sigma'}|$ einer aktuellen Parameterspezifikation gilt:

- A ist $\Sigma\cup\Sigma''$-erzeugt.
- A erfüllt E, die Gleichungen der aktuellen Parameterspezifikation
- A erfüllt die Konsistenzbedingung: $\equiv_{E\cup E',A}|_{T_{\Sigma\cup\Sigma''}} \subseteq \equiv_A$.

  Da nach Voraussetzung bereits $\equiv_A \subseteq \equiv_{E\cup E',A}|_{T_{\Sigma\cup\Sigma''}}$ gilt, ist damit sogar $\equiv_A = \equiv_{E\cup E',A}|_{T_{\Sigma\cup\Sigma''}}$.

  Dies bedeutet, daß die Gleichungen E' des formal parametrisierten Datentyps sich nicht auf den aktuellen Parameter auswirken.
- A erfüllt die Vollständigkeitsbedingung:

  $$(\forall s\in S)(\forall t\in T_s)(\exists t'\in T_{\Sigma\cup\Sigma'',s})(t \equiv_{E\cup E',A} t').$$

  Dies bedeutet, daß alle Terme der Termalgebra $T_{\Sigma\cup\Sigma'\cup\Sigma''}$, die von einer Sorte der Spezifikation des formalen Parameters sind, bzgl. $\hat{=}_A$ kongruent sind zu einem Term der Termalgebra des aktuellen Parameters. D.h., beim Übergang vom aktuellen Parameter zum aktuell parametrisierten Datentyp werden die Trägermengen der Sorten $s\in S$ nicht erweitert.

Die Vollständigkeitsbedingung in 4.1.5.3.2 läßt sich verallgemeinern:

4.1.5.4 Corollar

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation, (S'',Σ'') mit $SynZul(S'',\Sigma'',S,\Sigma,S',\Sigma')$, $Ae|Alg_{\Sigma,E,\Sigma'}|\cap|Alg_{\Sigma\cup\Sigma''}|$.
Dann gilt:

$$(\forall s\in S\cup S'')(\forall t\in T_{\Sigma\cup\Sigma'\cup\Sigma'',s})(\exists t'\in T_{\Sigma\cup\Sigma'',s})(t \equiv_{E\cup E',A} t')$$

Beweis:

seS: die Behauptung gilt, da $(S,\Sigma,E,S',\Sigma',E')$ parametrisierte i-Spezifikation ist.

seS": sei $te T_s$. Der Beweis erfolgt durch strukturelle Induktion.

1. $t=fe(\Sigma\cup\Sigma'\cup\Sigma")_{\varepsilon,s}$ ==> $fe\Sigma"_{\varepsilon,s}$ ==> $fe T_{\Sigma\cup\Sigma",s}$ (setze also t':=t).
2. $t=f(t1,\ldots,tn)$, $fe(\Sigma\cup\Sigma'\cup\Sigma")_{s1\ldots sn,s}$, $s1,\ldots,sn\in S\cup S'\cup S"$, $ti\in T_{si}$, $i=1,\ldots,n$.
   $se S"$ ==> $fe\Sigma"_{s1\ldots sn,s}$ ==> $s1,\ldots,sn\in S\cup S"$
   daraus folgt nach Induktionsvoraussetzung:
   zu $ti\in T_{si}$ existiert $ti'\in T_{\Sigma\cup\Sigma",si}$ mit $ti \equiv_{E\cup E';A} ti'$, $i=1,\ldots,n$.
   $\equiv_{E\cup E';A}$ ist $\Sigma\cup\Sigma'\cup\Sigma"$-Kongruenz
   ==> $t=f(t1,\ldots,tn) \equiv_{E\cup E';A} f(t1',\ldots,tn') =:t'$ und $t'\in T_{\Sigma\cup\Sigma",s}$.

MMM

Die Algebra $Ae|Alg_{\Sigma,E,\Sigma'}|$ (wo $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation ist) eines aktuellen Parameters kann in die Quotientenalgebra $T/_{\hat{=}_A}$ eingebettet werden:

4.1.5.5 Lemma

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation, $(S",\Sigma")$ mit $SynZul(S",\Sigma",S,\Sigma,S',\Sigma')$. Sei $Ae|Alg'_{\Sigma\cup\Sigma",E}|\cap|Alg_{\Sigma,E,\Sigma'}|$.
Dann gilt:

$$(\forall se S\cup S")\ \text{existiert}\ \Psi_s: A_s \longrightarrow T/_{\hat{=}_A|_{\Sigma\cup\Sigma'},s}$$
$$(\forall a\in A)\quad a \longmapsto [t]_{\hat{=}_A}\ \text{mit}\ te T_{\Sigma\cup\Sigma",s},\ \Phi_A(t)=a$$

und $\Psi=\{\Psi_s : se S\cup S"\}$ ist ein $\Sigma\cup\Sigma"$-Isomorphismus.

Beweis:

1. Ψ ist wohldefiniert:
   A ist $\Sigma\cup\Sigma"$-erzeugt ==> zu $a\in A$ existiert $te T_{\Sigma\cup\Sigma"}$ mit $\Phi_A(t)=a$.
   Sei nun $t\equiv_A t'$ mit $t,t'\in T_{\Sigma\cup\Sigma"}$.
   Nach Voraussetzung ($\equiv_A \subseteq \hat{=}_A$) folgt dann: $t\hat{=}_A t'$
   ==> Ψ ist wohldefiniert.

2. Ψ ist injektiv:
   Sei $se S\cup S"$, $a,a'\in A$.
   Sei $[t]_{\hat{=}_A} = \Psi_s(a)=\Psi_s(a') = [t']_{\hat{=}_A}$ mit $t,t'\in T_{\Sigma\cup\Sigma",s}$, $\Phi_A(t)=a$ und $\Phi_A(t')=a'$.

$\Longrightarrow t \hat{=}_A t'$. Nach Voraussetzung (parametrisierte i-Spezifikation) folgt dann:

$t \equiv_A t' \Longrightarrow a = a' \Longrightarrow \psi$ ist injektiv.

3. Ψ ist ein $\Sigma \cup \Sigma"$-Homomorphismus:

A ist $\Sigma \cup \Sigma"$-erzeugt. Führe also den Beweis durch strukturelle Induktion über $te T_{\Sigma \cup \Sigma"}$.

a) Sei $t = fe(\Sigma \cup \Sigma")_{\varepsilon,s}$, $se S \cup S"$ und $a := \Phi_A(t)$.

$\Longrightarrow \Psi_s(a) = [t]_{\hat{=}_A} = t_{T/\hat{=}_A}$.

b) Sei $t = f(t1,\ldots,tn)$, $fe(\Sigma \cup \Sigma")_{s1\ldots sn,s}$, $s1,\ldots,sn,se S \cup S"$, $ti e T_{\Sigma \cup \Sigma",si}$, $i=1,\ldots,n$.

$a := \Phi_A(t) = \Phi_A(f(t1,\ldots,tn)) = f_A(\Phi_A(t1),\ldots,\Phi_A(tn))$.

Sei nach Induktionsvoraussetzung $\Phi_A(ti)=ai$ bzw. $\Psi(ai)=[ti]_{\hat{=}_A}$, $i=1,..,n$.

$\Longrightarrow \Psi(f_A(a1,\ldots,an)) = [f(t1,\ldots,tn)]_{\hat{=}_A} = f_{T/\hat{=}_A}([t1]_{\hat{=}_A},\ldots,[tn]_{\hat{=}_A})$

$= f_{T/\hat{=}_A}(\Psi(a1),\ldots,\Psi(an))$.

Damit ist Ψ ein $\Sigma \cup \Sigma"$-Homomorphismus.

4. Ψ_s ist surjektiv für $se S \cup S"$:

Sei $se S \cup S"$, $[t]_{\hat{=}_A} e T/\hat{=}_A$ mit $te T_s$.

Zu zeigen: $(\exists a e A_s)\ (\Psi_s(a) = [t]_{\hat{=}_A})$

Nach Voraussetzung und nach 4.1.5.4 existiert zu t ein $t' e T_{\Sigma \cup \Sigma",s}$: $t \hat{=}_A t'$.

Setze $a := \Phi_A(t') \Longrightarrow \Psi_s(a) = [t']_{\hat{=}_A} = [t]_{\hat{=}_A}$.

Da $te T/\hat{=}_A, s$, $se S \cup S"$ beliebig, ist Ψ_s surjektiv für $se S \cup S"$.

MMM

4.1.5.6 Lemma

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation, $(S",\Sigma")$ mit SynZul$(S",\Sigma",S,\Sigma,S',\Sigma')$.

Sei $A,B e |Alg_{\Sigma,E,\Sigma'}| \cap |Alg'_{\Sigma \cup \Sigma"}|$, $g: A \longrightarrow B$ ein $\Sigma \cup \Sigma"$-Homomorphismus.

Dann gibt es genau einen $\Sigma \cup \Sigma' \cup \Sigma"$-Homomorphismus

$$\overline{g}: T_{\Sigma \cup \Sigma' \cup \Sigma"}/_{\equiv_{E \cup E',A}} \longrightarrow T_{\Sigma \cup \Sigma' \cup \Sigma"}/_{\equiv_{E \cup E',B}}$$

mit $[t]_{\hat{=}_A} \longmapsto [t]_{\hat{=}_B}$ $te T$

Beweis:

Zu zeigen: $\overline{g}$: $[t]_{\hat{=}_A} \longmapsto [t]_{\hat{=}_B}$, teT, ist wohldefiniert.

Φ_A: $T_{\Sigma\cup\Sigma''} \longrightarrow A$, Φ_B: $T_{\Sigma\cup\Sigma''} \longrightarrow B$. Sei $t,t' \in T_{\Sigma\cup\Sigma''}$ mit $t \equiv_A t'$

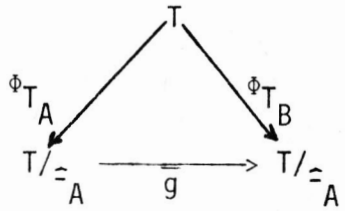
$\Longrightarrow g\circ\Phi_A(t) = g\circ\Phi_A(t')$.

Andererseits gilt (vgl. 4.1.4): $\Phi_B(t) = g\circ\Phi_A(t) = g\circ\Phi_A(t') = \Phi_B(t')$

$\Longrightarrow t \equiv_B t' \Longrightarrow \equiv_A \subseteq \equiv_B$.

Für $\hat{=}_B$ gilt: $\equiv_{E\cup E'} \subseteq \hat{=}_B$, $\equiv_B \subseteq \hat{=}_B$ $\Longrightarrow$ $\equiv_{E\cup E'} \subseteq \hat{=}_B$, $\equiv_A \subseteq \equiv_B \subseteq \hat{=}_B$

$\Longrightarrow \hat{=}_A \subseteq \hat{=}_B$, da $\hat{=}_A$ die kleinste Kongruenz ist, die $\equiv_{E\cup E'}$ und $\equiv_A$ enthält.



Nach 4.1.4 existiert höchstens ein $\overline{g}$: $T/_{\hat{=}_A} \longrightarrow T/_{\hat{=}_B}$ und es gilt : $\overline{g}\circ\Phi_{T_A} = \Phi_{T_B}$

$\Longrightarrow \overline{g}([t]_{\hat{=}_A}) = \overline{g}(\Phi_{T_A}(t)) = \Phi_{T_B}(t) = [t]_{\hat{=}_B}$, teT

MMM

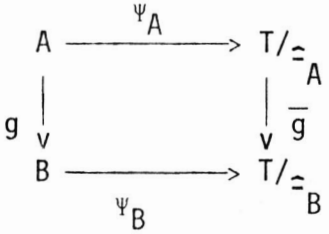
4.1.5.7 Corollar

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation,
(S'',Σ'') mit SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$.
Seien $A,B \in |Alg_{\Sigma,E,\Sigma'}| \cap |Alg'_{\Sigma\cup\Sigma''}|$, $\overline{g}$: $T/_{\equiv_{E\cup E'};A} \longrightarrow T/_{\equiv_{E\cup E'};B}$ wie in 4.1.5.6.
Zu A bzw. B existieren Ψ_A: $A \longrightarrow T/_{\hat{=}_A}$ bzw. Ψ_B: $B \longrightarrow T/_{\hat{=}_B}$ $\Sigma\cup\Sigma''$-Isomorphismen gemäß 4.1.5.5.
Dann gilt:

$$\overline{g}_s = \Psi_{B,s}\circ g_s \circ \Psi_{A,s}^{-1} \quad \text{für } s \in S\cup S''.$$

Beweis:

Die Behauptung folgt direkt aus 4.1.4:



$\overline{g}\circ\Psi_A = \Psi_B\circ g$: $A \longrightarrow T/_{\hat{=}_B}$

MMM

Damit kann ein $\Sigma\cup\Sigma''$-Homomorphismus zwischen zwei Algebren A und B der aktuellen Parameter-Spezifikation fortgesetzt werden (mittels den Einbettung-

en ψ_A und ψ_B) zu einem $\Sigma\cup\Sigma'\cup\Sigma''$-Homomorphismus der Quotientenalgebren $T/\hat{=}_A$ und $T/\hat{=}_B$ der Ergebnisspezifikationen .

Die Ergebnisse aus 4.1.5.5. und 4.1.5.6 werden nun zusammengefaßt, um die eingangs erwähnte Transformation zwischen Algebren einer Spezifikation eines aktuellen Parameters und den entsprechenden aktuell parametrisierten Datentypen zu konstruieren als Funktor zwischen den Kategorien $Alg_{\Sigma,E,\Sigma'}$ und $Alg_{\Sigma\cup\Sigma;E\cup E';\Sigma'}$ (wo $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation ist).

Dieser Funktor wird sogar persistent und initial in i-Trans$(\Sigma,E,\Sigma;E')$ sein.

4.1.5.8 Satz

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation,
(S'',Σ'') mit SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$.
Seien $A,B\in|Alg_{\Sigma,E,\bar{\Sigma}'}|\cap|Alg'_{\Sigma\cup\Sigma''}|$, $g: A \longrightarrow B$ ein $\Sigma\cup\Sigma''$-Homomorphismus.

$$F: Alg_{\Sigma,E,\Sigma'} \longrightarrow Alg_{\Sigma\cup\Sigma;E\cup E';\Sigma'}$$
$$(\forall A\in|Alg_{\Sigma,E,\Sigma'}|) \quad A \longmapsto T_{\Sigma\cup\Sigma'\cup\Sigma''}/_{\equiv_{E\cup E';A}}$$
$$g \longmapsto \bar{g} \text{ gemäß 4.1.5.6}$$

Dann ist F initial in i-Trans(Σ,E,Σ',E').

Beweis:

1. $T/\hat{=}_A \in|Alg_{\Sigma\cup\Sigma;E\cup E';\Sigma'}|\cap|Alg'_{\Sigma\cup\Sigma'\cup\Sigma''}|$:
   $T/\hat{=}_A$ ist $\Sigma\cup\Sigma'\cup\Sigma''$-erzeugte $\Sigma\cup\Sigma'\cup\Sigma''$-Algebra und erfüllt die Gleichungen $E\cup E'$ (vgl.: $\hat{=}_A = \equiv_{E\cup E';A}$).
2. F ist persistenter Funktor:
   a) Sei $A\in|Alg_{\Sigma,E,\Sigma'}|$.
      Sei $g=id_A$ ==> $\bar{g}=F(g)=id_{F(A)}$ nach 4.1.5.6.
   b) Seien $A,B,C\in|Alg_{\Sigma,E,\Sigma'}|\cap|Alg'_{\Sigma\cup\Sigma''}|$, $g: A \longrightarrow B$, $h: B \longrightarrow C$ seien $\Sigma\cup\Sigma''$-Homomorphismen, $t\in T$.
      $F(h\circ g)([t]_{\hat{=}_A}) = [t]_{\hat{=}_C} = F(h)([t]_{\hat{=}_B}) = F(h)\circ F(g)([t]_{\hat{=}_A})$
      ==> $F(h\circ g) = F(h)\circ F(g)$
      ==> F ist Funktor.
   c) F ist persistent nach 4.1.5.5.
   Damit gilt: $F\in i\text{-}Trans(\Sigma,E,\Sigma',E')$.

3. F ist initial in i-Trans(Σ,E,Σ',E'):

Sei Gei-Trans(Σ,E,Σ',E').

Setze: τ: F $\longrightarrow$ G, τ_A:F(A) $\longrightarrow$ G(A), Ae|$Alg_{\Sigma,E,\Sigma'}$|,

$$\tau_A([t]_{\cong_A}) := \Phi_{G(A)}(t), \quad [t]_{\cong_A} \in T/_{\cong_A}.$$

a) τ_A ist wohldefiniert:

$E \cup E' \subseteq \equiv_{G(A)}$ und $\equiv_A \subseteq \equiv_{G(A)}$, da G persistent

$\Longrightarrow \equiv_{E \cup E'; A} = \widehat{\equiv}_A \subseteq \equiv_{G(A)}$

$\Longrightarrow \tau_A$ ist wohldefiniert.

b) τ_A ist $\Sigma \cup \Sigma' \cup \Sigma$"-Homomorphismus, da $\Phi_{G(A)}$ ein $\Sigma \cup \Sigma' \cup \Sigma$"-Homomorphismus ist.

c) τ_A ist eindeutig nach 4.1.4.

d) τ ist natürliche Transformation:

$$\begin{array}{ccc} A & F(A) & \xrightarrow{\tau_A} G(A) \\ g \downarrow & \overline{g} \downarrow & \downarrow G(\overline{g}) \\ B & F(B) & \xrightarrow[\tau_B]{} G(B) \end{array}$$

$\overline{g}$, $G(\overline{g})$, τ_A und τ_B sind $\Sigma \cup \Sigma' \cup \Sigma$"-Homomorphismen

$\Longrightarrow G(\overline{g}) \circ \tau_A$: F(A) $\longrightarrow$ G(B) und

$\tau_B \circ \overline{g}$: F(A) $\longrightarrow$ G(B) sind

ebenfalls $\Sigma \cup \Sigma' \cup \Sigma$"-Homomorphismen.

F(A),G(A),F(B) und G(B) sind $\Sigma \cup \Sigma' \cup \Sigma$"-erzeugt

$\Longrightarrow G(\overline{g}) \circ \tau_A = \tau_B \circ \overline{g}$

$\Longrightarrow \tau$ ist natürliche Transformation.

Damit ist F initial in i-Trans(Σ,E,Σ',E').

⋈⋈⋈

Wünschenswert wäre, daß der initiale Funktor F die initiale (Quotienten-) Algebra einer Spezifikation eines aktuellen Parameters überführt in die initiale (Quotienten-)Algebra der aktuellen resultierenden Spezifikation und daß diese eine i-Erweiterung der aktuellen Parameter-Algebra ist. Insbesondere setzt dies voraus, daß die entsprechenden initialen Algebren in der Kategorie der zulässigen aktuellen Parameter bzw. der Kategorie der zugehörigen aktuell resultierenden Datentypen liegen.

4.1.5.9 Satz

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation, (S'',Σ'') mit SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$, $(S\cup S'',\Sigma\cup\Sigma'',E\cup E'')$ Spezifikation und Fei-Trans(Σ,E,Σ',E') wie in 4.1.5.8.
Dann gilt:

$$T_{\Sigma\cup\Sigma'';E\cup E''} \in |Alg_{\Sigma,E,\Sigma'}| \text{ und}$$
$$F(T_{\Sigma\cup\Sigma'';E\cup E''}) = T_{\Sigma\cup\Sigma'\cup\Sigma'';E\cup E'\cup E''} \text{ ist eine}$$
$$\text{i-Erweiterung von } T_{\Sigma\cup\Sigma'';E\cup E''}.$$

Beweis:

1. $A:=T_{\Sigma\cup\Sigma'';E\cup E''}\in|Alg_{\Sigma,E,\Sigma'}|$;

   A ist $\Sigma\cup\Sigma''$-erzeugte $\Sigma\cup\Sigma''$-Algebra und erfüllt die Gleichungen in E
   $\Longrightarrow A\in|Alg_{\Sigma,E,\Sigma'}|$.

2. $F(A)=T_{\Sigma\cup\Sigma'\cup\Sigma'';E\cup E'\cup E''}$:

   Zu zeigen: $\hat{=}_A = \equiv_{E\cup E'\cup E''}$

   "$\subseteq$": $\hat{=}_A = \equiv_{E\cup E';A}$ ist die kleinste Kongruenz, die $E\cup E'$ und $\equiv_A$ enthält.
   $\equiv_{E\cup E'} \subseteq \equiv_{E\cup E'\cup E''}$, $\equiv_A = \equiv_{E\cup E''} \subseteq T^2_{\Sigma\cup\Sigma''}$
   $\Longrightarrow \equiv_A \subseteq \equiv_{E\cup E''}|_{T_{\Sigma\cup\Sigma''}}$
   $\Longrightarrow \equiv_A \subseteq \equiv_{E\cup E'\cup E''}$
   $\Longrightarrow \hat{=}_A \subseteq \equiv_{E\cup E'\cup E''}$.

   "$\supseteq$": Klar ist nach Definition: $E\cup E' \subseteq \hat{=}_A = \equiv_{E\cup E';A}$.
   Noch zu zeigen: $E'' \subseteq \hat{=}_A$.
   Sei $(L,R)\in E''$, $E\cup E'\cup E''\subseteq T^2_{\Sigma\cup\Sigma'\cup\Sigma''}(X)$, $X:= \bigcup_{s\in S\cup S'\cup S''} X_s$
   Sei $\sigma: X \longrightarrow T_{\Sigma\cup\Sigma'\cup\Sigma''}$ eine Belegung.
   Zu σ existiert nach 1.1.1.23 eine eindeutige Fortsetzung
   $\hat{\sigma}: T_{\Sigma\cup\Sigma'\cup\Sigma''}(X) \longrightarrow T_{\Sigma\cup\Sigma'\cup\Sigma''}$.
   Sei nun $\hat{\sigma}(L)=:t$, $\hat{\sigma}(R)=:t'$ mit $t,t'\in T$.

   Zu zeigen: Es existiert eine Belegung σ': $X \longrightarrow T_{\Sigma\cup\Sigma'\cup\Sigma''}$ derart, daß $\hat{\sigma}'(L)=:\overline{t}$, $\hat{\sigma}'(R)=:\overline{t}'$ mit $\overline{t},\overline{t}'\in T_{\Sigma\cup\Sigma''}$ und $t\hat{=}_A\overline{t}$, $t'\hat{=}_A\overline{t}'$ (*)
   (wobei $\hat{\sigma}'$: $T_{\Sigma\cup\Sigma'\cup\Sigma''}(X) \longrightarrow T_{\Sigma\cup\Sigma'\cup\Sigma''}$ die Fortsetzung von σ' auf $T_{\Sigma\cup\Sigma'\cup\Sigma''}(X)$ ist.)

Es gilt: $(L,R) \in E'' \Longrightarrow R,L \in T_{\Sigma \cup \Sigma''}(X)$

d.h. alle Operationssymbole, die in L oder R vorkommen, sind aus $\Sigma \cup \Sigma''$

$\Longrightarrow$ Teilterme $t_o \in T\text{-}T_{\Sigma \cup \Sigma''}$ von t oder t' können nur durch die Belegung $\sigma(x)=t_o$ einer Variablen x in L bzw. R entstehen.

Sei $t_x := \sigma(x)$ für $x \in X$.

Sei $X' := \{x \in X$: x kommt in L oder R als Variable vor und $\sigma(x)=t_x \in T\text{-}T_{\Sigma \cup \Sigma''}\}$.

$X'_s := X' \cap X_s$, $s \in S \cup S' \cup S''$

$L,R \in T_{\Sigma \cup \Sigma''}(X) \Longrightarrow X' = \bigcup_{s \in S \cup S''} X'_s$

$\Longrightarrow (\forall x \in X')(\exists s \in S \cup S'')\ (\sigma(x)=t_x \in T_s)$

$\Longrightarrow (\forall s \in S \cup S'')(\forall x \in X'_s)(\exists t'_x \in T_{\Sigma \cup \Sigma'';s})\ t_x \hat{=}_A t'_x$ nach 4.1.5.4.

setze: σ': $X \longrightarrow T_{\Sigma \cup \Sigma' \cup \Sigma''}$

$$(\forall x \in X)\sigma'(x) := \begin{cases} t'_x \quad t'_x \in T_{\Sigma \cup \Sigma''} \text{ mit } t_x \hat{=}_A t'_x & \text{falls } x \in X' \\ \sigma(x) & \text{sonst} \end{cases}$$

Mit σ ist auch σ' eine Belegung. Sei $\hat{\sigma}'$ die Fortsetzung von σ' auf T(X). Dann gilt nach Konstruktion: $\hat{\sigma}'(L)$, $\hat{\sigma}'(R) \in T_{\Sigma \cup \Sigma''}$.

Da $\hat{=}_A$ $\Sigma \cup \Sigma' \cup \Sigma''$-Kongruenz ist, gilt ferner: $\overline{t} := \hat{\sigma}'(L) \hat{=}_A \hat{\sigma}(L) = t$

und $\overline{t}' := \hat{\sigma}'(R) \hat{=}_A \hat{\sigma}(R) = t'$.

Damit ist (*) bewiesen.

$\Longrightarrow t \hat{=}_A \overline{t} \equiv_A \overline{t}' \hat{=}_A t' \Longrightarrow t \hat{=}_A t' \Longrightarrow E'' \subseteq \hat{=}_A$

$\Longrightarrow E \cup E' \cup E'' \subseteq \hat{=}_A$

$\Longrightarrow \equiv_{E \cup E' \cup E''} \subseteq \hat{=}_A$.

Mit 4.1.5.5 ist dann $F(T_{\Sigma \cup \Sigma';E \cup E''})$ eine i-Erweiterung von $T_{\Sigma \cup \Sigma';E \cup E''}$.

■■■

Sei (S_o,Σ_o,E_o) eine Spezifikation und $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte i-Spezifikation. Den Σ_o-Algebren, die E_o erfüllen, entsprechen dann im parametrisierten Fall die i-Transformer Hei-Trans(Σ,E,Σ',E').

Insbesondere entspricht der initialen Quotientenalgebra T_{Σ_o,E_o} der initiale Funktor F in i-Trans(Σ,E,Σ',E').

Diese Parallelität legt folgende Definition nahe:

4.1.5.10 Definition (i-abstrakte parametrisierte Datentyp)

Sei PS:=(S,Σ,E,S',Σ',E') eine parametrisierte i-Spezifikation.
Dann heißt der in i-Trans(Σ,E,Σ',E') initiale Funktor der durch PS spezifizierte *i-abstrakte Datentyp*.

4.1.5.11 Unterschied zum ADJ-Ansatz

Der Unterschied zum ADJ-Ansatz (vgl. [ADJ 78], [EKTWW 79]) liegt in der konsequenten Betonung der Transformationseigenschaft eines parametrisierten Datentyps und im verwendeten Formalismus.

Die Betonung der Transformation zwischen Datentypen findet ihren Niederschlag in der Definition von i-Trans(Σ,E,Σ',E') und der Auszeichnung des initialen (bzw. terminalen) Funktors in dieser Kategorie. Dabei werden von Anfang an alle möglichen Algebren der Spezifikation des aktuellen Parameters in der Definition von $Alg_{\Sigma,E,\Sigma'}$ berücksichtigt. Daraus folgt, daß ein Funktor aus i-Trans(Σ,E,Σ',E') als Argument die gesamte Algebra eines aktuellen Parameters hat und nicht nur deren Σ-Einschränkung (d.h. sich also nicht nur auf die Eigenschaften des formalen Parameters bezieht).
Die Konsistenz- und Vollständigkeitsbedingungen in 4.1.5.3, die die Menge aller Spezifikationen einschränken, entsprechen der Forderung in Theorem 10, [ADJ 78], daß der Funktor F persistent ist, was ebenfalls die Menge der möglichen parametrisierten Spezifikationen einschränkt.

4.1.6 Terminaler Fall

Dual zum initialen Fall wird nun unter dem terminalen Gesichtspunkt die Theorie der abstrakten parametrisierten Datentypen entwickelt.
Zunächst wird die Verhaltensgleichheit $\sim_E$ aus Kapitel 1 verallgemeinert.

4.1.6 Definition ($\sim_\equiv$)

Sei (S,Σ,E) eine t-Spezifikation und ≡ eine Σ-Kongruenz auf T_Σ.
$\sim_\equiv \subseteq T_\Sigma \times T_\Sigma$ wird wie folgt definiert:

1. $\sim_{\equiv,dis} := \equiv_{dis}$
2. $(\forall s \in S-\{dis\})(\forall t,t' \in T_{\Sigma,s})$

   a) $t \overset{0}{\sim}_{\equiv,s} t'$:gdw $(\forall n \in N)(\forall s1,\ldots,sn \in S)(\forall f \in \Sigma_{s1\ldots sn,dis})$
   $(\forall 1 \leq j \leq n)(\forall (t1,\ldots tn) \in T_{\Sigma,s1} \times \ldots \times T_{\Sigma,sn})$
   $(s_j = s \Rightarrow f(t1,..,t,..,tn) \sim_{\equiv,dis} f(t1,..,t',..,tn))$

b) $(\forall i\in N_0)$

$t \overset{i+1}{\sim}_{\equiv,s} t'$:gdw $(\forall n\in N)(\forall s1,\ldots,sn,s'\in S)(\forall f\in \Sigma_{s1\ldots sn,s'})$

$(\forall 1\leq j\leq n)(\forall (t1,\ldots,tn)\in T_{\Sigma,s1}\times\ldots\times T_{\Sigma,sn})$

$(s_j=s \Longrightarrow f(t1,..,t,..,tn) \overset{i}{\sim}_{\equiv,s} f(t1,..,t',..,tn))$

3. $(\forall s\in S)\ \sim_{\equiv,s} := \bigcap_{i\in N_0} \overset{i}{\sim}_{\equiv,s}$

4. $\sim_{\equiv} := \{\sim_{\equiv,s} : s\in S\}$.

4.1.6.2 Lemma

Sei (S,Σ,E) eine t-Spezifikation und $\equiv$ Σ-Kongruenz auf T_Σ.
Dann ist $\sim_{\equiv}$ eine Σ-Kongruenz auf T_Σ.

Beweis:

1. $\sim_{\equiv}$ ist eine Familie von Äquivalenzrelationen:

Reflexivität:

Führe den Beweis durch Induktion über $i\in N_0$:

Zu zeigen: $(\forall i\in N_0)(\forall s\in S)(\forall t\in T_{\Sigma,s})\ t \overset{i}{\sim}_{\equiv,s} t$

i=0: Sei $s\in S$, $t\in T_{\Sigma,s}$.

$s=dis \Longrightarrow t\sim_{\equiv,dis} t$, da $\equiv$ Kongruenz.

$s\neq dis$: Sei $n\in N$, $s1,\ldots sn,s\in S$, $1\leq j\leq n$, $ti\in T_{\Sigma,si}$, $i=1,\ldots,n$,
$f\in \Sigma_{s1\ldots sn,dis}$, $sj=s$.
Dann gilt:
$f(t1,\ldots,tn)\sim_{\equiv,dis} f(t1,\ldots,tn)$, da $\sim_{\equiv,dis}$ reflexiv ist
$\Longrightarrow t \overset{0}{\sim}_{\equiv,s} t$.

i>0: Sei $\overset{k}{\sim}_{\equiv,s}$ reflexiv für $k\leq i-1$ und $s\in S$.
Sei $s\in S$, $t\in T_{\Sigma,s}$.
$s=dis \Longrightarrow$ fertig.
$s\neq dis$: Sei $n\in N$, $s1,\ldots,sn,s'\in S$, $f\in \Sigma_{s1\ldots sn,s'}$, $ti\in T_{\Sigma,si}$, $i=1,..,n$.
$s_j=s$, dann
$f(t1,\ldots,t,\ldots,tn) \overset{i-1}{\sim}_{\equiv,s'} f(t1,\ldots,t',\ldots,tn)$ nach Induktionsvoraussetzung

$\Longrightarrow \sim_{\equiv,s}$ ist reflexiv für $s\in S$.

Analog wird ausgehend von der Symmetrie bzw. Transitivität von $\equiv_{dis}$ induktiv die Symmetrie und Transitivität von $\sim_{\equiv}$ bewiesen.

2. Kongruenzeigenschaft:

Sei $n\in\mathbb{N}$, $s1,\ldots,sn,s\in S$, $f\in\Sigma_{s1\ldots sn,s}$, $ti,ti'\in T_{\Sigma,si}$ mit $ti\sim_{\equiv} ti'$ $i=1,\ldots,n$.
Da $(\forall i\in N_0)(\forall 1\leq j\leq n)(t_j \overset{i+1}{\sim}_{\equiv,s} tj')$, gilt:

$$f(t1,t2,\ldots,tn)\overset{i}{\sim}_{\equiv,s} f(t1',t2,\ldots,tn)\overset{i}{\sim}_{\equiv,s} f(t1',t2',..,tn)\overset{i}{\sim}_{\equiv,s}\ldots$$

$$\overset{i}{\sim}_{\equiv,s} f(t1',t2',\ldots,tn')$$

$$\Longrightarrow (\forall i\in N_0)\ f(t1,\ldots,tn)\ \overset{i}{\sim}_{\equiv,s}\ f(t1',\ldots,tn')$$

Insgesamt gilt damit: $\sim_{\equiv}$ ist Σ-Kongruenz.

MMM

4.1.6.3 Lemma

Sei (S,Σ,E) eine t-Spezifikation, $\equiv_1$ und $\equiv_2$ Σ-Kongruenzen auf T_Σ.
Dann gilt:

$$\sim_{\equiv_1,dis}\ \subseteq\ \sim_{\equiv_2,dis} \Longrightarrow \sim_{\equiv_1}\ \subseteq\ \sim_{\equiv_2}$$

Beweis:

Zu zeigen: $\sim_{\equiv_1,s}\ \subseteq\ \sim_{\equiv_2,s}$ für $s\in S$.
Der Beweis erfolgt induktiv über die Definition von $\sim_{\equiv_1}$ bzw. $\sim_{\equiv_2}$.

1. Sei $s\in S$, $t,t'\in T_{\Sigma,s}$ mit $t\overset{0}{\sim}_{\equiv_1} t'$

$$\Longrightarrow (\forall n\in N)(\forall s1,\ldots,sn\in S)(\forall f\in\Sigma_{s1\ldots sn,dis})(\forall(t1,\ldots,tn)\in T_{\Sigma,s1}\times\ldots\times T_{\Sigma,sn})$$
$$(\forall 1\leq j\leq n)((s=s_j \Longrightarrow f(t1,\ldots,t,..,tn)\ \overset{i}{\sim}_{\equiv_1,dis}\ f(t1,\ldots,t',\ldots,tn))$$

Da $\sim_{\equiv_1,dis}\ \subseteq\ \sim_{\equiv_2,dis}$ folgt: $t\overset{0}{\sim}_{\equiv_2} t'$

$$\Longrightarrow \overset{0}{\sim}_{\equiv_1,s}\ \subseteq\ \overset{0}{\sim}_{\equiv_2,s} \quad \text{für } s\in S,$$

2. Sei nach Induktionsvoraussetzung $\overset{i}{\sim}_{\equiv_1,s}\ \subseteq\ \overset{i}{\sim}_{\equiv_2,s}$ für $s\in S$, $i\geq 0$.

Sei $s\in S$, $t,t'\in T_{\Sigma,s}$ mit $t\overset{i+1}{\sim}_{\equiv_1} t'$.

$$\Longrightarrow((\forall n\in N)(\forall s1,\ldots,sn,s'\in S)(\forall f\in\Sigma_{s1\ldots sn,s'})(\forall(t1,\ldots,tn)\in T_{\Sigma,s1}\times\ldots\times T_{\Sigma,sn})$$
$$(\forall 1\leq j\leq n)(s=s_j \Longrightarrow f(t1,..,t,..,tn)\ \overset{i}{\sim}_{\equiv_1,s} f(t1,..,t',..,tn))$$
$$\Longrightarrow f(t1,..,t,..,tn)\overset{i}{\sim}_{\equiv_2,s} f(t1,..,t',..,tn))$$
$$\Longrightarrow t\ \overset{i+1}{\sim}_{\equiv_2}\ t'$$
$$\Longrightarrow \overset{i+1}{\sim}_{\equiv_1}\ \subseteq\ \overset{i+1}{\sim}_{\equiv_2}$$
$$\Longrightarrow \overset{i}{\sim}_{\equiv_1,s}\ \subseteq\ \overset{i}{\sim}_{\equiv_2,s} \quad \text{für } i\geq 0,\ s\in S \Longrightarrow \sim_{\equiv_1}\ \subseteq\ \sim_{\equiv_2}.$$

MMM

4.1.6.4 Lemma

Sei (S,Σ,E) eine t-Spezifikation, $\equiv$ eine Σ-Kongruenz auf T_Σ.
Dann gilt: $\equiv \subseteq \sim_{\equiv}$.

Beweis:

Der Beweis erfolgt induktiv über die Definition von $\sim_{\equiv}$.

1. Nach Definition von $\sim_{\equiv}$ ist $\equiv_{dis} = \sim_{\equiv,dis}$.

2. Sei $t \equiv t'$, $t,t' \in T_{\Sigma,s}$, $s \in S - \{dis\}$.

   a) Sei $f \in \Sigma_{s1\ldots sn,dis}$, $s1,\ldots,sn \in S$, $n \in N$ und $s_j = s$ für ein je $1,\ldots,n$.
   Sei $ti \in T_{\Sigma,si}$ $i=1,\ldots,j-1,j+1,\ldots,n$.
   $\equiv$ ist Σ-Kongruenz ==> $f(t1,..,t,..,tn) \equiv_{dis} f(t1,..,t',..,tn)$
   Mit 1. folgt: $f(t1,..,t,..,tn) \sim_{\equiv,dis} f(t1,..,t',..,tn)$
   ==> $t \overset{0}{\sim}_{\equiv,s} t'$
   ==> $\equiv_s \subseteq \overset{0}{\sim}_{\equiv,s}$ für $s \in S$.

   b) Sei nach Induktionsvoraussetzung: $\equiv \subseteq \overset{i}{\sim}_{\equiv,\bar{s}}$, $i \geq 0$, $\bar{s} \in S$.
   Sei $n \in N$, $s1,\ldots,sn,s' \in S$, $s_j = s$, $f \in \Sigma_{s1\ldots sn,s'}$, $ti \in T_{\Sigma,si}$, $i=1,..,j-1,j+1,..,n$
   $\equiv$ ist Kongruenz ==> mit $t \equiv_s t'$ gilt auch:
   $f(t1,..,t,..,tn) \equiv_{s'} f(t1,..,t',..,tn)$
   ==> $f(t1,..,t,..,tn) \overset{i}{\sim}_{\equiv,s'} f(t1,..,t',..tn)$ nach Induktionsvoraussetzung.
   f war beliebig ==> $t \overset{i+1}{\sim}_{\equiv,s} t'$
   ==> $\equiv_s \subseteq \overset{i+1}{\sim}_{\equiv,s}$,$s \in S$

Insgesamt gilt damit: $\equiv \subseteq \sim_{\equiv}$.

MMM

4.1.6.5 Definition ($Mod'_{\Sigma,E}$, $Mod_{\Sigma,E,\Sigma'}$, parametrisierte t-Spezifikation, (Σ,E,Σ',E')-t-Transformer, t-Trans(Σ,E,Σ',E')

Sei (S,Σ,E) eine vollständige t-Spezifikation.

1. $Mod'_{\Sigma,E} := \{A \in Alg'_{\Sigma,E} \mid : A_{dis} = \{tt_A, ff_A\} \wedge |A_{dis}| = 2 \wedge \equiv_A = \sim_A\}$
   wobei $\sim_A := \sim_{\equiv_A}$.

2. Sei $(S \cup S', \Sigma \cup \Sigma', E \cup E')$ eine t-Spezifikation mit $S \cap S' = \Sigma \cap \Sigma' = \emptyset$ und (S,Σ,E) konsistente t-Spezifikation.

   2.1 $Mod_{\Sigma,E,\Sigma'} := \bigoplus_{(S'',\Sigma'') \text{ mit } Syn\Sigma ul(S'',\Sigma'',S,\Sigma,S',\Sigma')} Mod'_{\Sigma\cup\Sigma'',E}$

2.2 $(S,\Sigma,E,S',\Sigma',E')$ ist *parametrisierte t-Spezifikation* :gdw

$(\forall(S'',\Sigma'')\colon \mathrm{SynZul}(S'',\Sigma'',S,\Sigma,S',\Sigma'))(\forall Ae|\mathrm{Mod}_{\Sigma,E,\Sigma'}|\cap|\mathrm{Alg}_{\Sigma\cup\Sigma''}|)$

$((\sim_{E\cup E';A}|T_{\Sigma\cup\Sigma''} = \equiv_A)\wedge((\forall s\in S)(\forall te T_{\Sigma\cup\Sigma\cup\Sigma'';s})(\exists t' e T_{\Sigma\cup\Sigma'';s})$

$(t \sim_{E\cup E';A} t')))$

wobei $\sim_{E\cup E';A} := \sim_{\equiv_{E\cup E';A}}$.

$\hat{\sim}_A := \sim_{E\cup E';A} = \sim_{\equiv_{E\cup E';A}}$.

3. Sei $(S,\Sigma, E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation.

3.1 Ein *(Σ,E,Σ',E')-t-Transformer* F ist ein persistenter Funktor

$F\colon \mathrm{Mod}_{\Sigma,E,\Sigma'} \longrightarrow \mathrm{Mod}_{\Sigma\cup\Sigma',E\cup E',\Sigma'}$ mit

$(\forall(S'',\Sigma'')\colon \mathrm{SynZul}(S'',\Sigma'',S,\Sigma,S',\Sigma'))(\forall Ae|\mathrm{Mod}_{\Sigma,E,\Sigma'}|\cap|\mathrm{Alg}_{\Sigma\cup\Sigma''}|)$

$(F(A)e|\mathrm{Mod}'_{\Sigma\cup\Sigma'\cup\Sigma''}|)$.

3.2 *t-Trans(Σ,E,Σ',E')* ist die Kategorie, deren Objekte gerade die (Σ,E,Σ',E')-t-Transformer sind und deren Morphismen die natürlichen Transformationen dieser Funktoren sind.

Als Modelle für die Spezifikation eines aktuellen Parameters werden also nur Algebren A zugelassen, bei denen die Verhaltensgleichheit mit der (initialen) Gleichheit zusammenfällt ($\equiv_A = \sim_A$, vgl. 4.1.6.5).

Die Anforderungen an eine parametrisierte t-Spezifikation (Konsistenz- und Vollständigkeitsbedingungen) entsprechen denen im initialen Fall.

In 4.1.6.5.2 wird nicht gefordert, daß $(S\cup S',\Sigma\cup\Sigma',E\cup E')$ eine konsistente t-Spezifikation ist. Dies ergibt sich nämlich aus der Konsistenz von (S,Σ,E) und der Konsistenzbedingung in 4.1.6.5.2.2.

4.1.6.6 Lemma

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation.
Dann ist $(S\cup S',\Sigma\cup\Sigma',E\cup E')$ eine konsistente t-Spezifikation.

Beweis:

Zu zeigen: $tr \not\equiv_{E\cup E'}$ fl.

Setze $(S'',\Sigma''):=(\emptyset,\emptyset)$. Dann gilt trivialerweise $\mathrm{SynZul}(S'',\Sigma'',S,\Sigma,S',\Sigma')$.
$A:= T_{\Sigma,\sim_E}.\ Ae|\mathrm{Mod}_{\Sigma,E,\Sigma'}|$, da (S,Σ,E) konsistente, vollständige t-Spezifikation ist.
(d.h. insbesondere: $|A_{dis}|=2$), A ist Σ-erzeugt und $\equiv_A = \sim_A = \sim_E$ ($\sim_A := \sim_{\equiv_A}$, $\sim_E := \sim_{\equiv_E}$).

$\sim_{E\cup E';A} = \sim_{E\cup E'}$.

Annahme: $(S\cup S',\Sigma\cup\Sigma',E\cup E')$ sei nicht konsistent, d.h. $tr \equiv_{E\cup E'} fl$
$\Longrightarrow tr \sim_{E\cup E';dis} fl$.
Mit der Konsistenzbedingung für die parametrisierte t-Spezifikation folgt dann, da $tr,fl \in T_\Sigma$:
$tr \equiv_A fl$ bzw. $tr \sim_E fl$
$\Longrightarrow tr \equiv_E fl$. Dies ist aber ein Widerspruch zur Konsistenz der t-Spezifikation (S,Σ,E).

Also gilt: $tr \not\equiv_{E\cup E'} fl$, d.h. $(S\cup S',\Sigma\cup\Sigma',E\cup E')$ ist eine konsistente t-Spezifikation.

MMM

Analog dem initialen Fall gilt die Vollständigkeitsbedingung in 4.1.6.5.2.2 schon für alle Sorten $s\in S\cup S''$:

4.1.6.7 Lemma

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation, (S'',Σ'') mit $SynZul(S'',\Sigma'',S,\Sigma,S',\Sigma')$ und $Ae|Mod_{\Sigma,E,\Sigma'}|\cap|Alg_{\Sigma\cup\Sigma''}|$.
Dann gilt:

$$(\forall s\in S\cup S'')(\forall t\in T_{\Sigma\cup\Sigma'\cup\Sigma'',s})(\exists t'\in T_{\Sigma\cup\Sigma'',s})(t \sim_{E\cup E';A} t')$$

Beweis:

$s\in S$: Dann gilt die Behauptung, da $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation ist.

$s\in S''$: Sei $t\in T_s$. Der Beweis wird geführt durch strukturelle Induktion über t.

1. $t=f\in(\Sigma\cup\Sigma'\cup\Sigma'')_{\varepsilon,s}$
   $\Longrightarrow f\in\Sigma''_{\varepsilon,s} \Longrightarrow f\in T_{\Sigma\cup\Sigma'',s}$ (Setze also $t':=t$).
2. $t=f(t1,\ldots,tn)$, $f\in(\Sigma\cup\Sigma'\cup\Sigma'')_{s1\ldots sn,s}$, $s1,\ldots,sn\in S\cup S'\cup S''$, $(t1,\ldots,tn)\in T_{s1}\times\ldots\times T_{sn}$.
   $s\in S'' \Longrightarrow f\in\Sigma''_{s1\ldots sn,s} \Longrightarrow s1,\ldots,sn\in S\cup S''$
   Nach Induktionsvoraussetzung gilt:
   $(\forall 1\leq i\leq n)(\exists ti'\in T_{\Sigma\cup\Sigma'',si})(ti \sim_{E\cup E';A} ti')$
   $\sim_{E\cup E';A}$ ist $\Sigma\cup\Sigma'\cup\Sigma''$-Kongruenz
   $\Longrightarrow t=f(t1,\ldots,tn) \sim_{E\cup E';A} f(t1',\ldots,tn') =:t'$ und $t'\in T_{\Sigma\cup\Sigma'',s}$.

MMM

Analog zu 4.1.5.5 und 4.1.5.6 geben 4.1.6.8 und 4.1.6.9 die duale Version für den terminalen Fall wieder.

4.1.6.8 Lemma

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation, $(S",\Sigma")$ mit $SynZul(S",\Sigma",S,\Sigma,S',\Sigma')$.
Sei $Ae|Mod_{\Sigma,E,\Sigma'}| \cap |Alg_{\Sigma\cup\Sigma"}|$.
Dann ist

$$\Psi := \{\Psi_s : s \in S \cup S"\}$$

$$\Psi_s : A_s \longrightarrow T/_{\tilde{\sim}_A,s}, \quad s \in S \cup S"$$

$$(\forall a \in A) \qquad a \longmapsto [t]_{\tilde{\sim}_A} \text{ mit } a=\Phi_A(t),\ t \in T_{\Sigma\cup\Sigma";s}$$

ein $\Sigma\cup\Sigma"$-Isomorphismus.

Beweis:

1. Ψ ist wohldefiniert:

   A ist $\Sigma\cup\Sigma"$-erzeugt ==> zu $a \in A$ existiert $t \in T_{\Sigma\cup\Sigma"}$ mit $\Phi_A(t)=a$.
   Sei nun $t \equiv_A t'$ für ein $t' \in T_{\Sigma\cup\Sigma"}$
   ==> $t \tilde{\sim}_A t'$ nach Voraussetzung (vgl.: $Ae|Mod_{\Sigma,E,\Sigma'}|$, $(S,\Sigma,E,S',\Sigma',E')$ ist parametrisierte t-Spezifikation)
   ==> $[t]_{\tilde{\sim}_A} = [t']_{\tilde{\sim}_A}$
   ==> Ψ ist wohldefiniert.

2. Ψ ist injektiv:

   Sei $s \in S \cup S"$, $a,a' \in A_s$.
   Sei $[t]_{\tilde{\sim}_A} = \Psi_s(a) = \Psi_s(a') = [t']_{\tilde{\sim}_A}$ mit $t,t' \in T_{\Sigma\cup\Sigma";s}$, $\Phi_A(t)=a$, $\Phi_A(t')=a'$
   ==> $[t] \tilde{\sim}_A [t']$. Nach Voraussetzung gilt dann:
   $t \equiv_A t'$
   ==> $a = \Phi_A(t) = \Phi_A(t') = a'$
   ==> Ψ ist injektiv.

3. Ψ ist $\Sigma\cup\Sigma"$-Homomorphismus:
   A ist $\Sigma\cup\Sigma"$-erzeugt. Führe also den Beweis durch strukturelle Induktion über $t \in T_{\Sigma\cup\Sigma"}$.

   a) Sei $t=f \in (\Sigma\cup\Sigma")_{\varepsilon,s}$, $s \in S \cup S"$ und $a:=\Phi_A(t)$
      ==> $\Psi_s(a) = [t]_{\tilde{\sim}_A} = t_{T/\tilde{\sim}_A,s}$.

b) Sei $t=f(t1,\ldots,tn)$, $fe(\Sigma\cup\Sigma")_{s1\ldots sn,s}$, $s1,\ldots,sn,se S\cup S"$ und
$(t1,\ldots,tn)eT_{\Sigma\cup\Sigma";s1}x\ldots x\,T_{\Sigma\cup\Sigma";sn}$.
$a:= \Phi_A(t) = \Phi_A(f(t1,\ldots,tn)) = f_A(\Phi_A(t1),\ldots,\Phi_A(tn))$
Sei nach Induktionsvoraussetzung:
$\Phi_A(ti) = ai$ bzw. $\Psi_{si}(ai) = [ti]_{\tilde{\sim}_A}$, $i=1,\ldots,n$.
$\Longrightarrow \Psi(f_A(a1,\ldots,an)) = [f(t1,\ldots,tn)]_{\tilde{\sim}_A} = f_{T/\tilde{\sim}_A}([t1]_{\tilde{\sim}_A},\ldots,[tn]_{\tilde{\sim}_A})$
$= f_{T/\tilde{\sim}_A}(\Psi_{s1}(a1),\ldots,\Psi_{sn}(an))$.

Damit ist Ψ ein $\Sigma\cup\Sigma"$-Homomorphismus.

3. Ψ_s ist surjektiv für $se S\cup S"$:

Sei $se S\cup S"$, $[t]_{\tilde{\sim}_A} eT/\tilde{\sim}_A$, teT_s.

Zu zeigen: Es existiert aeA_s, so daß $\Psi_s(a) = [t]_{\tilde{\sim}_A}$.

Nach Voraussetzung und nach 4.1.6.7 existiert zu t ein $t'eT_{\Sigma\cup\Sigma";s}$ mit $t \tilde{\sim}_A t'$.

Setze: $a:= \Phi_A(t') \Longrightarrow \Psi_s(a) = [t']_{\tilde{\sim}_A} = [t]_{\tilde{\sim}_A}$.

Damit ist Ψ_s surjektiv für $se S\cup S"$.

Insgesamt ist Ψ ein $\Sigma\cup\Sigma"$-Isomorphismus.

⋈⋈⋈

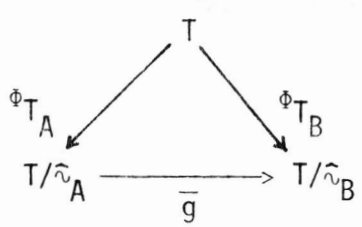
4.1.6.9 Lemma

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation,
$(S",\Sigma")$ mit $SynZul(S",\Sigma",S,\Sigma,S',\Sigma')$.
Sei $A,Be\,|Mod_{\Sigma,E,\Sigma'}|\cap|Alg_{\Sigma\cup\Sigma"}|$, $g: A \longrightarrow B$ ein $\Sigma\cup\Sigma"$-Homomorphismus.
Dann gibt es genau einen $\Sigma\cup\Sigma'\cup\Sigma"$-Homomorphismus

$$\overline{g}: T_{\Sigma\cup\Sigma'\cup\Sigma"}/\tilde{\sim}_A \longrightarrow T_{\Sigma\cup\Sigma'\cup\Sigma"}/\tilde{\sim}_B$$

nämlich $(\forall teT)$ $[t]_{\tilde{\sim}_A} \longmapsto [t]_{\tilde{\sim}_B}$
und $\overline{g}$ ist schon ein $\Sigma\cup\Sigma'\cup\Sigma"$-Isomorphismus.

Beweis:



Nach 4.1.4 existiert höchstens ein $\overline{g}: T/\tilde{\sim}_A \longrightarrow T/\tilde{\sim}_B$
und für $\overline{g}$ gilt: $\overline{g}\circ\Phi_{T_A} = \Phi_{T_B}$

$\Longrightarrow \overline{g}([t]_{\tilde{\sim}_A}) = \overline{g}(\Phi_{T_A}(t)) = \Phi_{T_B}(t) = [t]_{\tilde{\sim}_B}$, teT.

Noch zu zeigen: $\bar{g}: [t]_{\hat{\sim}_A} \longmapsto [t]_{\hat{\sim}_B}$, teT, ist wohldefiniert.

Zeige dazu: $\hat{\sim}_A = \hat{\sim}_B$.

Nach 4.1.6.3 genügt es zu zeigen: $\hat{\sim}_{A,dis} = \hat{\sim}_{B,dis}$.

$\hat{\sim}_{A,dis} = \equiv_{E\cup E';A,dis}$, $\hat{\sim}_{B,dis} = \equiv_{E\cup E';B,dis}$.
$E\cup E' \subseteq \equiv_{E\cup E';B}$.

Sei $\Phi_A: T_{\Sigma\cup\Sigma"} \longrightarrow A$, $\Phi_B: T_{\Sigma\cup\Sigma"} \longrightarrow B$ $\Sigma\cup\Sigma"$-homomorph.

Sei $t,t'eT_{\Sigma\cup\Sigma"}$ mit $t \equiv_A t'$
$\Longrightarrow \Phi_B(t) = g\circ\Phi_A(t) = g\circ\Phi_A(t') = \Phi_B(t')$

$\Longrightarrow t \equiv_B t'$

$\Longrightarrow \equiv_A \subseteq \equiv_B$

$\Longrightarrow \equiv_A \subseteq \equiv_B \subseteq \equiv_{E\cup E';B}$

$\Longrightarrow \equiv_{E\cup E';A} \subseteq \equiv_{E\cup E';B}$

$\Longrightarrow \hat{\sim}_{A,dis} \subseteq \hat{\sim}_{B,dis}$.

Nun gilt weiter: $|A_{dis}| = |B_{dis}| = 2 \Longrightarrow |T/\hat{\sim}_{A,dis}| = |T/\hat{\sim}_{B,dis}| = 2$.

Sei $T/\hat{\sim}_{A,dis} = \{[tr]_{\hat{\sim}_A}, [fl]_{\hat{\sim}_A}\}$, $T/\hat{\sim}_{B,dis} = \{[tr]_{\hat{\sim}_B}, [fl]_{\hat{\sim}_B}\}$
$\Longrightarrow [tr]_{\hat{\sim}_A} \cap [tr]_{\hat{\sim}_B} \neq \emptyset$, $[fl]_{\hat{\sim}_A} \cap [fl]_{\hat{\sim}_B} \neq \emptyset$.
Sei $t1,t2eT_{dis}$ mit $t1 \hat{\sim}_B t2$.

OBdA sei $t1 \hat{\sim}_B tr \hat{\sim}_B t2$.

Annahme: $t1 \not\hat{\sim}_A t2$, also etwa oBdA $t1 \hat{\sim}_A tr$, $t2 \hat{\sim}_A fl$

$\hat{\sim}_A \subseteq \hat{\sim}_B \Longrightarrow t1 \hat{\sim}_B tr$, $t2 \hat{\sim}_B fl \Longrightarrow tr \hat{\sim}_A fl$
Dies ist ein Widerspruch zur Konsistenz.

$\Longrightarrow t1 \hat{\sim}_A' t2 \Longrightarrow \hat{\sim}_{B,dis} \subseteq \hat{\sim}_{A,dis}$.

$\Longrightarrow \hat{\sim}_{A,dis} = \hat{\sim}_{B,dis}$.

Mit 4.1.6.3 folgt dann: $\hat{\sim}_A = \hat{\sim}_B$

$\Longrightarrow \bar{g}$ ist wohldefiniert.

Insbesondere gilt dann aber: $T/\hat{\sim}_A$ ist $\Sigma\cup\Sigma'\cup\Sigma"$-isomorph zu $T/\hat{\sim}_B$,
d.h. $\bar{g}$ ist (nach 4.1.4) schon $\Sigma\cup\Sigma'\cup\Sigma"$-Isomorphismus.

⋈⋈⋈

4.1.6.10 Corollar

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation,
(S'',Σ'') mit SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$.
Seien $A,Be|Mod_{\Sigma,E,\Sigma'}|\cap|Alg_{\Sigma\cup\Sigma''}|$, $g: A \longrightarrow B$ ein $\Sigma\cup\Sigma''$-Homomorphismus.
Dann ist g schon ein $\Sigma\cup\Sigma''$-Isomorphismus, d.h. $A\simeq B$.

Beweis:
Mit 4.1.6.8 und 4.1.4 ist $g = \Psi_B^{-1} \circ \bar{g} \circ \Psi_A$.
$\bar{g}|_{T_{\Sigma\cup\Sigma''}}$ ist $\Sigma\cup\Sigma''$-Isomorphismus, ebenso Ψ_A und Ψ_B.

⋈⋈⋈

Alle Modelle eines aktuellen Parameters, die homomorph sind, sind also schon isomorph.

Die Ergebnisse aus 4.1.6.8 und 4.1.6.9 werden nun zusammengefaßt, um einen persistenten Funktor zwischen den Katrgorien $Mod_{\Sigma,E,\Sigma'}$ und $Mod_{\Sigma\cup\Sigma';E\cup E';\Sigma'}$ (wobei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation ist) anzugeben, der dazuhin in t-Trans(Σ,E,Σ',E') terminal ist.

4.1.6.11 Satz

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation,
(S'',Σ'') mit SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$.
Seien $A,Be|Mod_{\Sigma,E,\Sigma'}|\cap|Alg_{\Sigma\cup\Sigma''}|$, $g: A \longrightarrow B$ ein $\Sigma\cup\Sigma''$-Homomorphismus.

$$F: |Mod_{\Sigma,E,\Sigma'}| \longrightarrow |Mod_{\Sigma\cup\Sigma',E\cup E',\Sigma'}|$$
$$A \longmapsto T/\hat{\sim}_A$$
$$g \longmapsto \bar{g} \text{ gemäß 4.1.6.9.}$$

Dann ist F terminal in t-Trans(Σ,E,Σ',E').

Beweis:

1. $T/\hat{\sim}_A \in |Mod_{\Sigma\cup\Sigma';E\cup E';\Sigma'}|\cap|Alg_{\Sigma\cup\Sigma'\cup\Sigma''}|$:

   $T/\hat{\sim}_A$ ist $\Sigma\cup\Sigma'\cup\Sigma''$-erzeugte $\Sigma\cup\Sigma'\cup\Sigma''$-Algebra.

   $\equiv_{E\cup E'} \overset{C}{=} \equiv_{E\cup E';A} \overset{C}{\sim} \sim_{E\cup E';A} \overset{=}{=} \hat{\sim}_A$

   ==> $T/\hat{\sim}_A$ erfüllt die Gleichungen in $E\cup E'$.

   $\equiv_{T/\hat{\sim}_A} \overset{=}{\sim} \sim_{T/\hat{\sim}_A}$

   Damit ist $T/\hat{\sim}_A \in |Mod'_{\Sigma\cup\Sigma';E\cup E'}|$; da $(S,\Sigma,E,S',\Sigma',E')$ parametrisierte

t-Spezifikation ist, gilt: $T/\hat{\sim}_A \in |Mod_{\Sigma\cup\Sigma';E\cup E';\Sigma'}| \cap |Alg_{\Sigma\cup\Sigma'\cup\Sigma"}|$

2. F ist persistenter Funktor:

a) Sei $A \in |Mod_{\Sigma,E,\Sigma'}|$.
Sei $g = id_A$ ==> $\bar{g} = F(g) = id_{F(A)}$ nach 4.1.6.9.

b) Seien $A,B,C \in |Mod_{\Sigma,E,\Sigma'}| \cap |Alg_{\Sigma\cup\Sigma"}|$, $g: A \longrightarrow B$, $h: B \longrightarrow C$ $\Sigma\cup\Sigma"$-Homomorphismen.
$F(h\circ g)([t]_{\hat{\sim}_A}) = [t]_{\hat{\sim}_C} = F(h)([t]_{\hat{\sim}_B}) = F(h)\circ F(g)([t]_{\hat{\sim}_A})$, $t \in T$
==> $F(h\circ g) = F(h)\circ F(g)$
== F ist Funktor.

c) F ist persistent nach 4.1.6.8.

3. F ist terminal in t-Trans(Σ,E,Σ',E'):
Sei $G \in |t\text{-}Trans(\Sigma,E,\Sigma',E')|$.
Setze: $\tau: G \longrightarrow F$

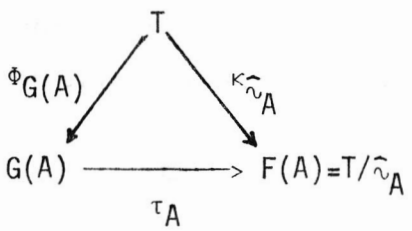
$\tau_A: G(A) \longrightarrow F(A)$, $A \in |Mod_{\Sigma,E,\Sigma'}|$

$\tau_A(\Phi_{G(A)}(t)) := [t]_{\hat{\sim}_A}$, $t \in T$

a) τ_A ist wohldefiniert:
Zu zeigen: $\equiv_{G(A),s} \subseteq \hat{\sim}_{A,s}$ für $s \in S\cup S'\cup S"$

s=dis: F,G sind persistent
==> $\sim_{G(A),dis} = \equiv_{G(A),dis} = \hat{\sim}_{A,dis}$
Mit 4.1.6.3 folgt dann: $\sim_{G(A),s} = \hat{\sim}_{A,s}$, $s \in S\cup S'\cup S"$
==> $\sim_{G(A)} = \hat{\sim}_A$
==> $\equiv_{G(A)} \subseteq \sim_{G(A)} = \hat{\sim}_A$
==> τ_A ist wohldefiniert.

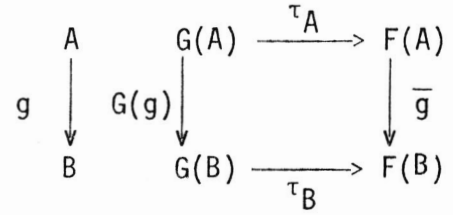
b) τ_A ist eindeutiger $\Sigma\cup\Sigma'\cup\Sigma"$-Homomorphismus:



$\kappa_{\hat{\sim}_A}$ ist der durch die $\Sigma\cup\Sigma'\cup\Sigma"$-Kongruenz $\hat{\sim}_A$ induzierte $\Sigma\cup\Sigma'\cup\Sigma"$-Homomorphismus.

Nach Definition von τ_A kommutiert das Diagramm und τ_A ist $\Sigma\cup\Sigma'\cup\Sigma"$-homomorph.
Da G(A), F(A) $\Sigma\cup\Sigma'\cup\Sigma"$-erzeugt sind, ist mit 4.1.4 τ_A eindeutig.

c) τ ist natürliche Transformation:



G(g), τ_A, τ_B und $\bar{g}$ sind $\Sigma\cup\Sigma'\cup\Sigma''$-homomorph

$\Longrightarrow$ $\bar{g}\circ\tau_A$: G(A) $\longrightarrow$ F(B) und $\tau_B\circ G(g)$: G(A) $\longrightarrow$ F(B) sind ebenfalls $\Sigma\cup\Sigma'\cup\Sigma''$-Homomorphismen.

G(A) und F(B) sind $\Sigma\cup\Sigma'\cup\Sigma''$-erzeugt $\Longrightarrow$ $\bar{g}\circ\tau_A = \tau_B\circ G(g)$

$\Longrightarrow$ τ ist natürliche Transformation.

Damit ist F terminal in t-Trans(Σ, E,Σ',E').

MMM

4.1.6.12 Corollar

Sei (S,Σ,E,S',Σ',E') eine parametrisierte t-Spezifikation und $F\in|$t-Trans(Σ,E,Σ',E')$|$ wie in 4.1.6.11.
Dann ist F auch initial in t-Trans(Σ,E,Σ',E'), d.h. F ist bis auf Isomorphie der einzige t-Transformer in t-Trans(Σ,E,Σ',E').

Beweis:

Nur noch zu zeigen: F ist initial in t-Trans(Σ,E,Σ',E').
Sei $G\in|$t-Trans(Σ,E,Σ',E')$|$, $A\in|Mod_{\Sigma,E,\Sigma'}|$.
Setze: $\bar{\tau}$: F $\longrightarrow$ G, $\bar{\tau}_A$: F(A) $\longrightarrow$ G(A)

$$\bar{\tau}_A([t]_{\hat{\sim}_A}) := \Phi_{G(A)}(t) \quad , \ t\in T.$$

$\bar{\tau}_A$ ist wohldefiniert, da $\equiv_{G(A)} = \hat{\sim}_A$, denn:

$\sim_{G(A)} = \hat{\sim}_A$ nach Beweis zu 4.1.6.11 (Teil 3a).

$G\in|$t-Trans(Σ,E,Σ',E')$|\Longrightarrow$ $G(A)\in|Mod'_{\Sigma\cup\Sigma'\cup\Sigma''}|$ nach 4.1.6.5

$\Longrightarrow$ $\equiv_{G(A)} = \sim_{G(A)}$ $\Longrightarrow$ $\equiv_{G(A)} = \hat{\sim}_A$.

$\Longrightarrow$ $\bar{\tau}_A = \tau_A^{-1}$ $\Longrightarrow$ $\bar{\tau}_A$ ist eindeutiger $\Sigma\cup\Sigma'\cup\Sigma''$-Homomorphismus (dabei sei τ_A wie im Beweis von 4.1.6.11) und $\bar{\tau}$ ist natürliche Transformation.

MMM

Wie im initialen Fall gilt nun, daß der terminale Funktor F die terminale (Quotienten-) Algebra der Spezifikation eines aktuellen Parameters überführt in die terminale Quotientenalgebra der resultierenden aktuellen Spezifikation und daß sich die Algebra des aktuellen Parameters in die der Ergebnisspezifikation einbetten läßt:

4.1.6.13 Satz

Sei $(S,\Sigma,E,S',\Sigma',E')$ eine parametrisierte t-Spezifikation,
$(S",\Sigma")$ mit $SynZul(S",\Sigma",S,\Sigma,S',\Sigma')$.
$SP:=(S\cup S",\Sigma\cup\Sigma",E\cup E")$ sei eine konsistente und vollständige t-Spezifikation.
Dann gilt:

$$T_{\Sigma\cup\Sigma";\sim_{E\cup E"}} \in |Mod_{\Sigma,E,\Sigma'}| \quad \text{und}$$

$$F(T_{\Sigma\cup\Sigma";\sim_{E\cup E"}}) = T_{\Sigma\cup\Sigma'\cup\Sigma";\sim_{E\cup E'\cup E"}}$$

ist eine t-Erweiterung von $T_{\Sigma\cup\Sigma";\sim_{E\cup E"}}$ (dabei ist F definiert wie in 4.1.6.11).

Beweis:

1. $A:=T_{\Sigma\cup\Sigma";\sim_{E\cup E"}} \in |Mod_{\Sigma,E,\Sigma'}|$:

   A ist $\Sigma\cup\Sigma"$-erzeugte $\Sigma\cup\Sigma"$-Algebra und erfüllt die Gleichungen in E,
   da $\equiv_E \subseteq \equiv_{E\cup E"} \subseteq \sim_{E\cup E"}$.
   $|A_{dis}| = 2$, da SP vollständige und konsistente t-Spezifikation ist.
   Weiter gilt: $\equiv_A = \sim_A$.
   $\Longrightarrow A \in |Mod_{\Sigma,E,\Sigma'}|$.

2. $F(A) = T_{\Sigma\cup\Sigma'\cup\Sigma";\sim_{E\cup E'\cup E"}}$:

   Zu zeigen: $\hat{\sim}_A = \sim_{E\cup E';A} \overset{!}{=} \sim_{E\cup E'\cup E"}$.

   Zeige hierzu: $\hat{\sim}_{A,dis} = \equiv_{E\cup E';A,dis} \overset{!}{=} \equiv_{E\cup E'\cup E";dis} = \sim_{E\cup E'\cup E";dis}$

   "$\subseteq$": $\equiv_{E\cup E';dis} \subseteq \equiv_{E\cup E'\cup E";dis}$

   $\equiv_{A,dis} = \equiv_{E\cup E";dis} \subseteq T^2_{\Sigma\cup\Sigma";dis} \Longrightarrow \equiv_{A,dis} \subseteq \equiv_{E\cup E'\cup E";dis}$.

   $\equiv_{E\cup E';A}$ ist die kleinste Kongruenz, die $E\cup E'$ und $\equiv_A$ enthält

   $\Longrightarrow \hat{\sim}_{A,dis} = \equiv_{E\cup E';A,dis} \subseteq \equiv_{E\cup E'\cup E";dis} = \sim_{E\cup E'\cup E";dis}$.

   Mit 4.1.6.3 folgt dann die Behauptung.

   "$\supseteq$":

   Es gilt: $E_{dis} \cup E'_{dis} \subseteq \equiv_{E\cup E';A,dis} = \hat{\sim}_{A,dis}$.

   Noch zu zeigen: $E"_{dis} \subseteq \hat{\sim}_A$.

   Sei $E\cup E'\cup E" \subseteq T^2(X)$, $X = \bigcup_{s \in S\cup S'\cup S"} X_s$.

   Sei $(L,R) \in E"_{dis}$, σ eine Belegung in T mit $\hat{\sigma}(L)=t$, $\hat{\sigma}(R)=t'$, $t,t' \in T_{dis}$,
   wobei $\hat{\sigma}$ die Fortsetzung von σ auf $T(X)$ ist.

Zu zeigen: Es existiert eine Belegung σ': $X \longrightarrow T_{\Sigma\cup\Sigma'\cup\Sigma"}$ derart, daß $\hat{\sigma}'(L)=:\overline{t}$, $\hat{\sigma}'(R)=:\overline{t}'$, $\overline{t},\overline{t}' \in T_{\Sigma\cup\Sigma";dis}$ und $t \;\hat{\sim}_A\; \overline{t}$, $t' \;\hat{\sim}_A\; \overline{t}'$ (*)
(wobei $\hat{\sigma}'$: $T(X) \longrightarrow T$ die Fortsetzung von σ' auf $T(X)$ ist).

Es ist $(L,R)\in E"_{dis}$ ==> $R,L\in T_{\Sigma\cup\Sigma";dis}(X)$, d.h. alle Operationssymbole, die in L oder R vorkommen, sind aus $\Sigma\cup\Sigma"$.
==> Teilterme $t_o \in T\text{-}T_{\Sigma\cup\Sigma"}$ von t oder t' können nur durch die Belegung $\sigma(x)=t_o$ einer Variablen x in L bzw. R entstehen.
Sei $t_x:=\sigma(x)$ für $x\in X$.
Sei $X':=\{x\in X$: x kommt als Variable in L oder R vor und

$$\sigma(x)=t_x\in T\text{-}T_{\Sigma\cup\Sigma"}\}.$$

$X'_S:=X'\cap X_S$, $S\in S\cup S'\cup S"$.

$$L,R\in T_{\Sigma\cup\Sigma";dis}(X) \;\Longrightarrow\; X'=\bigcup_{S\in S\cup S"} X'_S$$

==> $(\forall x\in X)(\forall S\in S\cup S")\ (\sigma(x)=t_x\in T_S)$
==> $(\forall S\in S\cup S")(\forall x\in X'_S)(\exists t'_x\in T_{\Sigma\cup\Sigma";S})(t_x \;\hat{\sim}_A\; t'_x)$ gemäß 4.1.6.7.
Setze:

$$\sigma'\colon X \longrightarrow T_{\Sigma\cup\Sigma'\cup\Sigma"}$$

$$\sigma'(x):=\begin{cases} t'_x & t'_x\in T_{\Sigma\cup\Sigma"} \text{ mit } t'_x \;\hat{\sim}_A\; t_x \quad \text{falls } x\in X' \\ \sigma(x) & \text{sonst} \end{cases}$$

Mit σ ist auch σ' eine Belegung. Sei $\hat{\sigma}'$ die (eindeutige) Fortsetzung von σ' auf $T(X)$. Dann gilt nach Konstruktion:
$\hat{\sigma}'(L)$, $\hat{\sigma}'(R)\in T_{\Sigma\cup\Sigma";dis}$.
Da $\hat{\sim}_A$ eine $\Sigma\cup\Sigma'\cup\Sigma"$-Kongruenz ist, gilt weiter:
$\overline{t}:=\hat{\sigma}'(L) \;\hat{\sim}_A\; \hat{\sigma}(L) = t$, $\overline{t}':=\hat{\sigma}'(R) \;\hat{\sim}_A\; \hat{\sigma}(R) = t'$.

Damit ist (*) bewiesen.
==> $t \;\hat{\sim}_{A,dis}\; \overline{t} \;\equiv_{A,dis}\; \overline{t}' \;\hat{\sim}_{A,dis}\; t'$
==> $t \;\hat{\sim}_{A,dis}\; t'$
==> $E"_{dis} \subseteq \hat{\sim}_{A,dis}$
==> $\sim_{E\cup E'\cup E";dis} \subseteq \hat{\sim}_{A,dis}$.
Nach 4.1.6.3 ist dann $\sim_{E\cup E'\cup E"} \subseteq \hat{\sim}_A$.

Insgesamt ist also: $\sim_{E\cup E'\cup E"} = \hat{\sim}_A$
==> $T_{\Sigma\cup\Sigma'\cup\Sigma";\sim_{E\cup E'\cup E"}} = T_{\Sigma\cup\Sigma'\cup\Sigma";\hat{\sim}_A}$
Mit 4.1.6.8 ist dann $F(T_{\Sigma\cup\Sigma";\sim_{E\cup E"}})$ eine t-Erweiterung von $T_{\Sigma\cup\Sigma";\sim_{E\cup E"}}$.

MMM

Abschließend kann analog dem initialen Fall ein t-abstrakter parametrisierter Datentyp definiert werden.

4.1.6.14 Definition (t-abstrakter parametrisierter Datentyp)

Sei PS:=(S,Σ,E,S',Σ',E') eine parametrisierte t-Spezifikation.
Dann heißt der in t-Trans(Σ,E,Σ',E') terminale Funktor der durch SP spezifizierte *t-abstrakte parametrisierte Datentyp*.

Damit sind nun die begrifflichen Grundlagen für die nächsten Abschnitte gelegt.

4.2 Parametrisierung in den Beispielsprachen

Sowohl in CSSA als auch in CLU und ALPHARD ist die Möglichkeit der Typparametrisierung eines Scripts bzw. Clusters bzw. einer Form gegeben.

In CSSA ist dieses Konzept aber noch nicht ausgereift, so daß in größerem Umfang nur auf die Parametrisierung in CLU und ALPHARD eingegangen wird. Dabei wird sich zeigen, daß es möglich ist, beiden Sprachen wieder die gleiche abstrakte Syntax und Semantik zu unterlegen,

4.2.1 CSSA

Syntaktisch besteht die Möglichkeit der Parametrisierung:

type id is script (..., elemtype,...assert elemtype is type)

Z.B.: type stack is script (elemtype assert elemtype is type)

Bei der Aktualisierung müßte eine Kopie dieses Scripts angelegt und mit dem Verweis auf den aktuellen Typparameter (in der Regel eine Script-Acquaintance) versehen werden. (Alternativ ist ein speichersparendes Verfahren ohne Kopie, daür aber mit einer Parameter-Area möglich). Gerade dieser Mechanismus fehlt aber in CSSA.

Außerdem kennt CSSA keine Unterscheidung zwischen Typen und Typgeneratoren. Ein Script bezeichnet somit stets einen Typ. Dies bedeutet insbesondere, daß auch ein formal typparametrisiertes Script einen Datentyp bezeichnet und nicht eine Klasse von Datentypen.

Aufgrund dieser Mängel wird im weiteren auf CSSA nicht eingegangen. In

einer vorgesehenen neuen Version von CSSA (CSSA-M) sollen diese Mängel behoben werden, so daß dann eine echte Typparametrisierung von Datentypen möglich sein wird.

4.2.2 CLU (vgl.:[CLU 78])

4.2.2.1 Clusterdefinition

<cluster> ::= <Idn>=cluster[<Parm>*] is <Opid>*<body>
<parm> ::= <Idn>+:(<Typespec>|type)

Dabei gibt 'type' an, daß der voranstehende Bezeichner ein formaler Typparameter ist.

Beispiel: stack = cluster (elemtype: type) is push, pop, top;
<body>

Für 'elemtype' kann bei der Aktualisierung ein beliebiger Datentyp eingesetzt werden, d.h. der Bezeichner eines CLusters, das nicht selbst (formal) typparametrisiert ist.

4.2.2.2. Aktualisierung, Aufruf

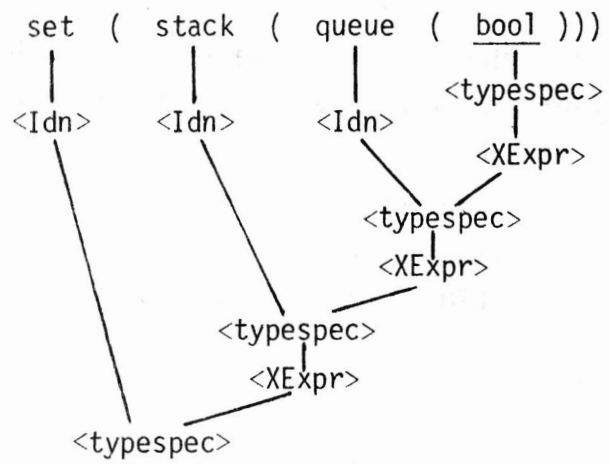
<invoc> ::= <primary> (<expr>*)
<primary> ::= <Idn>[<XExpr>*]|<typespec>$<Idn>[<XExpr>*]|<primary>.<Idn>|
<primary>[<expr>]|<invoc>|...
<XExpr> ::= <expr>|<typespec>
<typespec> ::= <simpletype>|<Idn>[<XExpr>*]|...
<simpletype> ::= bool|int|char

Ein Prozeduraufruf (<invoc>) ist sowohl eine Anweisung als auch ein Ausdruck. <primary>ist eine Operation, die insbesondere auch Ergebnis einer Funktion sein kann.
<typespec> enthält die Aktualisierung eines Typparameters:

<typespec> ::= <Idn>[<XExpr>*]

Durch die Gleichstellung von aktualisierten parametrisierten Typen und den Standardtypen (bool ...) kommt schon in der Syntax klar zum Ausdruck, daß die Ersetzung eines formalen Typparameters in einem parametrisierten Cluster durch einen aktuellen Parameter als Ergebnis einen Datentyp liefert.

Beispiel zur Aktualisierung:



In CLU ist nur Call-by-value und nicht Call-by-reference vorgesehen. Eine Aktualisierung wie etwa

set(stack(elemtype))

ist somit nicht möglich.

4.2.3 ALPHARD (vgl.: [ALPH 78])

4.2.3.1 Formdefinition

<form-decl> ::= form <idn>(<formals>*) is <form-body>
<formals> ::= <type formals>|...
<type formals>::= <idn>+: form

form gibt hier an, daß die in der Bezeichner-Liste aufgeführten formalen Parameter für beliebige Typen stehen.

Beispiel:

form BSTACK (elemtype: form, maxsize: INTEGER) is ...
Für 'elemtype' kann ein beliebiger Typ eingesetzt werden.

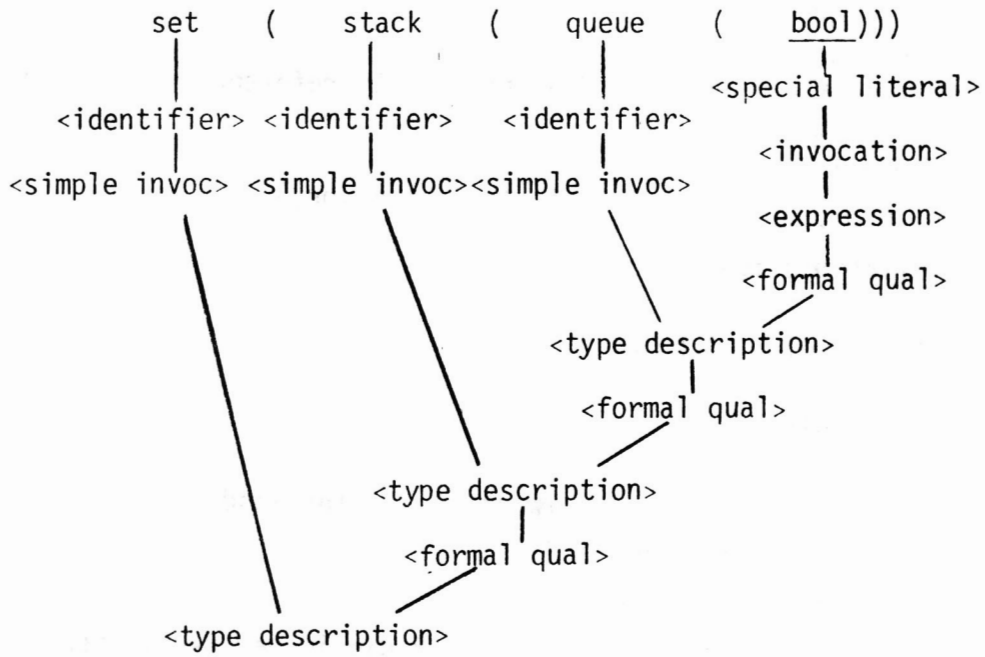
4.2.3.2 Aktualisierung, Aufruf

<invocation> ::= <simple invocation>(<actual>*)|<special literal>
<simple invocation>::= Operationsbezeichner oder Bezeichner einer Form
<actual> ::= <type description>|<expression>
<special literal> ::= bool|integer|char|string
<type description> ::= <simple invocation>({<formal qual>,}+)

<formal qual> ::= <expression>|{<identifier>:}*<type description>
<expression> ::= <invocation> ...

Ein Aufruf (<invocation>) ist wieder sowohl Anweisung als auch Ausdruck.
<type description> beschreibt eine Klasse von Objekttypen, die an einen Bezeichner gebunden werden können.
Die Abarbeitung einer <type description> ergibt einen Typ. Somit enthält <type description> wieder die Aktualisierung.
Damit ergibt sich auch, daß in ALPHARD nur Call-by-value möglich ist.

Beispiel zur Aktualisierung:



4.2.4 CLALPHARD

Der vorstehenden Beschreibung der Parametrisierung in CLU und ALPHARD kann man entnehmen, daß sowohl bei der Definition als auch bei der Aktualisierung eines parametrisierten Datentyps in beiden Sprachen keine wesentlichen Unterschiede bestehen.
Bei der Vereinbarung werden die Typparameter lediglich als solche ausgewiesen (type bzw. form) und evt. durch eine WHERE- bzw.ASSERT-Klausel ergänzt, die notwendige Bedingungen des aktuellen Parameters beinhaltet.

Die Aktualisierung erfolgt als Typdeskription in einer Vereinbarung (z.B:: s: stack(bool)) und impliziert ein Create des aktualisierten Typs.
In beiden Sprachen wird nach Call-by-value aktualisiert.

4.3 Auswirkungen der Parametrisierung in CLALPHARD auf die abstrakte Syntax und Semantik

Im wesentlichen wird die abstrakte Syntax und Semantik von CLALPHARD aus 2.2 übernommen und nur dort geändert und ergänzt, wo dies die Parametrisierung erforderlich macht.

4.3.1 Änderungen der abstrakten Syntax

Die formalen Typparameter ('type') werden gesondert in die Cluster-Schnittstelle aufgenommen:

Clinterface = (<cltype:Id>,<parm:Id*>,<typparm:Id*>,<public:Id+>)

Seither, im unparametrisierten Fall, wurden die Typen der formalen bzw. aktuellen Parameter nicht berücksichtigt; ein formaler Parameter war ein Bezeichner. Insbesondere wurde damit ein Type-Checking überflüssig. Diese Regelung wird hier im parametrisierten Fall beibehalten.

Bei einer Aktualisierung müssen die aktuellen Typparameter angegeben werden, was wieder gesondert geschieht:

Call = (<mod:Id>,<op:Id>,<parm:Expr*>,<typparm:Expr*>)

Falls ein Operationsaufruf kein Create ist, wird für 'typparm' die leere Liste eingesetzt. Ein aktueller Parameter ist entsprechend der Syntax von CLU und ALPHARD ein Ausdruck (<primary>bzw. <invocation>).

4.3.2 Änderungen der denotationalen Semantik

Semanatische Bereiche:

Als Objekte (Obj) sind jetzt nicht nur Locations möglich, sondern auch Cluster-Bezeichner, da zum Beispiel das Ergebnis eines Ausdrucks der Bezeichner eines Clusters sein kann. Daher gilt nun:

Obj := Loc + Id

Semantische Funktionen:

Entsprechend dem zweiten Parametersatz in der Cluster-Schnittstelle muß der Definitionsbereich von $\mathcal{C}$ erweitert werden:

$\mathcal{C}'$: Cluster $\longrightarrow$ Obj* x Obj* $\longrightarrow$ Modul

bzw. genauer:

$$\mathcal{C}'\text{: Cluster} \longrightarrow \text{Loc}^* \times \overline{\text{Id}}^* \longrightarrow \text{Modul}$$

wobei $\overline{\text{Id}} \subseteq \text{Id}$ die Menge aller Clusterbezeichner sei .
$\mathcal{C}'$ enthält somit implizit den Prozeß der Aktualisierung. Diese wird nun explizit durchgeführt durch eine Hilfsfunktion

$$\text{AKT: Cluster} \longrightarrow \overline{\text{Id}}^* \longrightarrow \text{Cluster}$$

AKT ersetze textuell in einem typparametrisierten Cluster cl alle Vorkommnisse eines formalen Typparameters durch den entsprechenden Clusterbezeichner aus der aktuellen Parameterliste typparm: $\overline{\text{Id}}^*$ und ersetzt abschließend in der Schnittstelle des Clusters cl 'typparm' durch die leere Liste.
Ist cl nicht typparametrisiert, so ist AKT die Identität.
Das Ergebnis von AKT ist somit ein nicht typparametrisiertes Cluster (call-by-value).
Auf diese Cluster kann dann $\mathcal{C}$ aus 2.2.2 angewendet werden:

$\mathcal{C}' = \mathcal{C} \circ \text{AKT}$ d.h. $\mathcal{C}'(\text{cl}) = \mathcal{C}(\text{AKT}(\text{cl}))$, wo cl ein (evt.)typparametrisiertes Cluster ist.

Nun müssen vor einer Aktualisierung noch die aktuellen Typparameter interpretiert werden; zu diesem Zweck wird $\mathcal{C}[\![\text{ce: Call}]\!]$ abgeändert:

$\mathcal{C}[\![\text{ce: Call}]\!]\rho\sigma =$

sei n:=Length(parm.ce); n':=Length(typparm.ce);
$\rho_1 := \rho$; me:=me':=$\perp$; L:={l∈Loc: $\rho_1(l) = \rho_1(\sigma(\text{mod.ce}))$} in
sei case n=0: $\rho_{n+1} := \rho$
case n>0:
(sei $e_i := \mathcal{C}[\![(\text{parm.ce})_i]\!]\rho_i\sigma$ in
me:=Cons($e_i \downarrow 2$,me);
$\rho_{i+1} := e_i \downarrow 1$) i=1,..,n in

sei case n'=0: $\rho_{n+n'+1} := \rho_{n+1}$
case n'>0: (sei $e'_i := \mathcal{C}[\![(\text{typparm.ce})_i]\!]\rho_{n+i}\sigma$ in
me':=Cons($e'_i \downarrow 2$,me');
$\rho_{n+i+1} := e'_i \downarrow 1$) i=1,...,n' in

case op.ce=Create:

sei locmod:=Newloc($\rho_{n+n'+1}$);
cl:=σ(mod.ce) in

sei ρ'=λloc.loc=locmod $\longrightarrow$ $\mathcal{C}'[\![cl]\!]$(me,me')|

$\rho_{n+n'+1}$(loc) in

(ρ',locmod)

case op.ce≠Create:

unverändert wie in 2.2.2.

4.4 Korrekte Implementierung abstrakter typparametrisierter Datentypen durch Cluster

In diesem Abschnitt werden zunächst für eine Menge von zum Teil typparametrisierten Clustern syntaktische Forderungen entwickelt, ähnlich wie dies in 3.1 für den unparametrisierten Fall geschehen ist.
Eine besondere Rolle spielt die Frage, welchen Typ man als aktuellen Parameter für einen formalen Typparameter eines Clusters einsetzen soll, wenn die Spezifikation (S",Σ",E")+(S,Σ,E) eines aktuellen Parameters gegeben ist.

Abschließend wird dann ausgehend von dem Implementierungsbegriff aus 3.4 angegeben, wann eine Menge von parametrisierten Clustern einen abstrakten parametrisierten Datentyp korrekt implementiert.

Zunächst wird der Vorgang der Parametrisierung und Aktualisierung auf der algebraischen und auf der programmiersprachlichen Ebene informell erläutert.

Parametrisierung auf der Ebene der algebraischen Spezifikation:

Ausgegangen wird von einer fest gegebenen Spezifikation (S,Σ,E) des formalen Parameters.
Diese Spezifikation wird um (S',Σ',E'), die Angaben für den parametrisierten Datentypen, ergänzt und ergibt die Kombination (S,Σ,E)+(S',Σ',E'), die Spezifikation des formal parametrisierten Datentyps.
Durch Ergänzung von (S,Σ,E) um (S",Σ",E"), die zusätzlichen Angaben für den aktuellen Parameter, erhält man (S,Σ,E)+(S",Σ",E") als Spezifikation des aktuellen Parameters. Allein (S",Σ",E") ist variabel (vgl.Einleitung zu 4.1).

Parametrisierung auf der programmiersprachlichen Ebene der Cluster:

Ausgegangen wird von einem (oder mehreren) typparametrisierten Cluster pcl mit den formalen Typparametern $formparm_i$, i=1,...,k.
pcl bildet zusammen mit den Clustern $cl_1,...,cl_n$, die die in pcl zitierten Typen definieren, eine Clustermenge C_p, die zusammen mit den formalen Parametern $formparm_i$, i=1,...,k, später mit $(S,\Sigma,E)+(S',\Sigma',E')$ in Verbindung gebracht wird.
Sowohl in CLU als auch in ALPHARD besteht die Möglichkeit, in einer WHERE- bzw. ASSERT-Klausel an die formalen bzw. aktuellen Parameter Bedingungen zu stellen, wie etwa das Vorhandensein bestimmter Operationen mit bestimmten Eigenschaften. Diese Bedingungen entsprechen (z.T.) denen in der Spezifikation (S,Σ,E) des formalen Parameters.

Für jeden formalen Parameter $formparm_i$, i=1,...,k, wird nun ein Cluster acl_i, i=1,...,k, dessen Definition u.U. noch die Definition weiterer Cluster $c_{i,j}$, j=1,...,n_i, erforderlich macht, als aktueller Parameter angegeben.
Den aktuell parametrisierten Datentyp apT erhält man dann durch Ersetzung der formalen Parameter in pcl durch die Bezeichner der entsprechenden Cluster acl_i und die Vereinigung von {apT} mit der Menge aller Cluster acl_i,i=1,...,k, und der Menge all der Cluster, die definitorische (Hils-) Funktion haben, also$\{ c_{i,j}: i=1,...,k, j=1,...,n_i\}$.

4.4.1 Definition (SErf(C,S',Σ',S,Σ,α,γ))

Sei (S,Σ) eine Signatur und (S',Σ') mit $(S,\Sigma) \leq (S\cup S',\Sigma\cup\Sigma')$.
Sei C eine Menge von (zum Teil) typparametrisierten Clustern.
OP_C sei die Menge aller öffentlichen Operationen (inklusive Creates), die in den Clustern in C vorkommen und FP_C sei die Menge aller formalen Typparameter, die in den Clustern in C vorkommen.
C *erfüllt schwach die Signatur*$(S\cup S',\Sigma\cup\Sigma')$ unter (S,Σ) :gdw

1. es existiert $\alpha: C \cup FP_C \longrightarrow S\cup S'$ mit
   $\alpha(FP_C) \subseteq S$
   $\alpha|_C: C \longrightarrow S'$ ist bijektiv
2. es existiert $\gamma: OP_C \longrightarrow \Sigma'$ bijektiv, so daß gilt:

$$(\forall P_f \in OP_C)(P_f: A'_{s1}x...x\ A'_{sn} \longrightarrow A'_s \ ==>\ \gamma(P_f)=fe\Sigma'_{\alpha(s1)...\alpha(sn),\alpha(s)}).$$

Notation: SErf(C,S',Σ',S,Σ,α,γ).

4.4.1 enthält für den parametrisierten Fall die syntaktischen Forderungen an eine Menge von Cluster und entspricht damit 3.1.6 für den unparametrisierten Fall.

Forderung 1 in 4.4.1 verlangt, daß ein formaler Parameter auf eine Sorte der Signatur (S,Σ) der formalen Parameter abgebildet wird und daß jedem (typparametrisierten) Cluster eine Sorte der Erweiterung von S um S' zugeordnet wird und umgekehrt.

Die zweite Forderung in 4.4.1 besagt, daß Operationen der typparametrisierten Cluster nur auf Operationssymbole der zu Σ hinzugefügten Menge Σ' von Funktionssymbolen abgebildet werden.
Um eine sinnvolle Definition dafür angeben zu können, wann ein abstrakter parametrisierter Datentyp durch eine Menge von (zum Teil) parametrisierten Clustern korrekt implementiert wird, ist es notwendig, die Ersetzung der formalen Typparameter durch aktuelle Parameter (d.h. Bezeichner von nicht typparametrisierten Clustern) zu beschreiben. Die Schwierigkeit dabei ist, daß nicht ohne weiteres klar ist, welcher aktuelle Parameter für welchen formalen einzusetzen ist, wenn auf der algebraischen Ebene bereits eine Aktualisierung vorgenommen worden ist.

4.4.2 Beispiel REIHE(DATA)

(S,Σ) sei eine Signatur mit

$S := \{data\}$

$\Sigma := \emptyset$

(S,Σ) gibt die syntakischen Eigenschaften des/der formalen Parameter(s) an.

(S',Σ'):

$S' := \{reihe, nat, bool\}$

$\Sigma' := \{$

creater:		⟶ reihe,
read	: reihe x nat	⟶ data,
assign	: reihe x data x nat	⟶ reihe,
null	:	⟶ nat,
succ	: nat	⟶ nat,
eq	: nat x nat	⟶ bool,
t,f	:	⟶ bool,
not	: bool	⟶ bool $\}$

$(S,\Sigma)+(S',\Sigma')$ ist eine Kombination von (S,Σ) und (S',Σ') und gibt die syntaktischen Eigenschaften des formal parametrisierten Datentyps an.

(S",Σ"):

S":= ∅

Σ":= {zero : —> data,
succint, predint: data —> data}

(S,Σ)+(S",Σ") ist eine Kombination von (S,Σ) und (S",Σ") und gibt die syntaktischen Eigenschaften des aktuellen Parameters an.

Zugehörige Cluster:

```
pcl = REIHE: cluster (elemtype: type) is Creater, Read, Assign;
             rep (paramelem: type) ... eltype: type ...;
             Creater: ... r:rep(elemtype) ... ;
             Read: (r: cvt, n: NAT) returns r.eltype; <code> ;
             Assign: (r:cvt, d: r.eltype, n: NAT); <code> ;
             end REIHE
```

Zur Definition von REIHE wird noch NAT, und zu dessen Definition BOOL benötigt. Damit:

C_p:= {REIHE, NAT, BOOL}.

```
cl1 = NAT  : cluster () is Null, Succ, Eq;
             rep  <code>;
             Null: <code>;
             Succ: (d: cvt) returns d': cvt; <code>;
             Eq  : (d: cvt, d': cvt) returns b: BOOL; <code>;
             end NAT

cl2 = BOOL : cluster () is True, False, Not;
             rep   <code>;
             True : <code>;
             False: <code>;
             Not  : (b: cvt) returns b': cvt; <code>;
             end BOOL
```

Aktueller Parameter:

```
acl = DATA : cluster () is Zero, Succint, Predint;
             rep     <code>;
             Zero   : <code>;
```

Succint: (d: cvt) returns d': cvt; <code>;
Predint: (d: cvt) returns d': cvt; <code>;

end DATA

Zur Definition von acl werden keine weiteren Cluster mehr benötigt.
Damit:

$C_a := \{acl\}$

Intuitiv liegt es nahe, für 'elemtype' in 'REIHE' den Clusterbezeichner 'DATA' einzusetzen. Würde man diesem aktualisierten Cluster noch die Cluster NAT, BOOL und DATA hinzufügen, so hätte man (bei geeigneter Programmierung) eine Implementierung des abstrakten Datentyps 'reihe(integer)'.

Es gilt:

$C_p := \{REIHE, NAT, BOOL\}$ erfüllt schwach die Signatur $(S \cup S', \Sigma \cup \Sigma')$ unter (S,Σ).

Beweis:

Setze: α: (REIHE, elemtype, NAT, BOOL) $\longmapsto$ (reihe, data, nat, bool) komponentenweise.

γ: (Creater,Read,Assign,Null,Succ,Eq,True,False,Not) $\longmapsto$
$\longmapsto$ (creater,read,assign,null,succ,eq,t,f,not) komponentenweise.

α und γ erfüllen offensichtlich die Bedingungen aus 4.4.1.

MMM

Außerdem gilt:

$C_a := \{DATA\}$ erfüllt die Signatur $(S,\Sigma)+(S",\Sigma")$.

Beweis:

Setze: $\overline{\alpha}$(DATA):=data

$\overline{\gamma}$: (Zero,Succint,Predint) $\longmapsto$ (zero,succint,predint) komponentenweise.

$\overline{\alpha}$ und $\overline{\gamma}$ erfüllen die Bedingungen aus 3.1.6.

MMM

Für die Aktualisierung eines typparametrisierten Clusters durch nicht-typparametrisierte Clusters kann man nun die folgende vorläufige Definition angeben:

4.4.3 Definition (Aktualisierung)

Sei clp ein typparametrisiertes Cluster.
Sei FP_{clp} die Menge der formalen Typparameter in clp.
Sei Ca $\neq \emptyset$ eine Menge von nicht-typparametrisierten Clustern.
cl ist *Aktualisierung* von clp unter Ca, falls cl aus clp auf folgende Art und Weise entsteht:

1. $\forall$formparmeFP_{clp}: ersetze 'formparm' im gesamten Cluster clp textuell durch einen Bezeichner eines Clusters aus Ca.
2. Streiche in der Schnittstelle von clp die formalen Typparameter.

4.4.2 Beispiel (Fortsetzung)

Eine bzw. die erwartete Aktualisierung von REIHE ist die, die man durch Einsetzung von 'DATA' für 'elemtype' erhält.
Dies ist aber nicht die einzig mögliche Einsetzung. In Frage kommen dafür noch NAT und BOOL, was aber offensichtlich in der abstrakten Spezifikation nicht beabsichtigt wurde.
Wie können diese Möglichkeiten der Aktualisierung eingeschränkt werden?
Vergleiche hierzu die Signatur (S',Σ') und die Funktionalität der Funktionen bzw. Prozeduren von REIHE:

C_p erfüllt schwach die Signatur (S',Σ') unter (S,Σ).
Damit kommen als aktuelle Parameter für 'elemtype' nur in Frage:
$\bar{\alpha}^{-1}(\alpha(\text{elemtype}))\in C_a$, wo α die Bedingungen aus 4.4.1 und $\bar{\alpha}$ die aus 3.1.6 erfüllen (C_a erfüllt (S,Σ)+(S",Σ")).

(Ende Beispiel 4.4.2)

In Beispiel 4.4.2 war α bijektiv. Im allgemeinen ist dies jedoch nicht der Fall. Dies liegt dann daran, daß (S,Σ) für das parametrisierte Cluster zuviel Information enthält. Zur Veranschaulichung dient das folgende Beispiel.

4.4.4 Beispiel SET(data)

Sei (S,Σ) eine Signatur mit

S:= {data,bool}
Σ:= {eq : data x data —> bool,
t,f: —> bool,
not: bool —> bool}

Von dem formalen Parameter wird also insbesondere gefordert, daß für ihn ein Gleichheitsprädikat definiert ist.

```
(S',Σ'):   S':= {set}
           Σ':= {emptyset     :              —> set,
                 insert,remove: set x data —> set}

           (S,Σ)+(S',Σ') ist Kombination.

(S",Σ"):   S":= {integer}
           Σ":= {null       :                    —> integer,
                 succ,pred  : integer            —> integer,
                 eqint      : integer x integer —> bool,
                 push       : data x integer    —> data,
                 pop        : data              —> data,
                 top        : data              —> integer,
                 Createstack:                    —> data}

           (S,Σ)+(S",Σ") ist Kombination.
```

(S,Σ)+(S',Σ')+(S",Σ") wird aufgefaßt als die Signatur des aktuell parametrisierten Datentyps 'set(stack(integer))'.

Zugehörige Cluster:

```
pcl = SET  : cluster (elemtype: type) is Emptyset,Insert,Remove
                where elemtype has  Eq,T,F,Not;

             rep (paramelem: type) ... eltype: type ... ;
             Emptyset: ... s:rep(elemtype) ...;
             Insert: (s: cvt, d: s.eltype); <code>;
             Remove: (s: cvt, d: s.eltype); <code>;

             end SET
```

C_p:= {SET}

Aktueller Parameter:

```
acl = DATA : cluster () is Createstack, Push,Pop,Top,Eq;

             rep          <code>;
             Createstack: <code>;
             Push       : (st: cvt, i: INT); <code>;
             Pop        : (st: cvt); <code>;
             Top        : (st: cvt) returns n: INT; <code>;
             Eq         : (st: cvt, st': cvt) returns b: BOOL; <code>;

             end DATA
```

Zur Definition von DATA wird noch INT und BOOL benötigt:

```
c1 = INT  : cluster () is Null, Succ,Pred,Eqint;
            rep    <code>;
            Null : <code>;
            Succ : (n: cvt) returns n': cvt; <code>;
            Pred : (n: cvt) returns n': cvt; <code>;
            Eqint: (n: cvt, n': cvt) returns b: BOOL; <code>;
            end INT

c2 = BOOL : cluster () is  True, False, Not;
            rep  : <code>;
            True : <code>;
            False: <code>;
            Not  : (b: cvt) returns b': cvt; <code>;
            end BOOL
```

C_a:= {DATA,INT,BOOL}

Die Informationen, die die Signatur (bzw. Spezifikation) des formalen Parameters enthält, werden bei der Definition der zugehörigen Cluster erst beim aktuellen Parameter wiedergegeben.

Nun gilt:

C_p:= {SET} erfüllt schwach die Signatur (S',Σ') unter (S,Σ).

Beweis:

Setze: α(SET):=set, α(elemtype):=data

γ: (Emptyset,Insert,Remove) $\longrightarrow$ (emptyset,insert,remove) komponentenweise.

α und γ erfüllen offensichtlich die Bedingungen in 4.4.1.

MMM

Weiter gilt:

C_a={DATA,INT,BOOL} erfüllt die Signatur $(S,\Sigma)+(S'',\Sigma'')$.

Beweis:

Setze: $\overline{\alpha}$: (DATA,INT,BOOL) $\longmapsto$ (data,integer,bool) komponentenweise.

$\overline{\gamma}$: (Createstack,Push,Pop,Top,Eq,Null,Succ,Pred,Eqint,True,False,Not) $\longmapsto$ (createstack,push,pop,top,eq,null,succ,pred,eqint,t,f,not) komponentenweise.

$\overline{\alpha}$ und $\overline{\gamma}$ erfüllen die Bedingungen in 3.1.6.

MMM

In obigem Beispiel kann u.a. angenommen werden, daß DATA = STACK(INT) die Aktualisierung des parametrisierten Clusters STACK(elemtype) ist. Auf diese Weise werden bei der Aktualisierung auch geschachtelte Parametrisierungen (Call-by-value) berücksichtigt.

4.4.5 Definition (Aktualisierung von C_p unter C_a mittels α, $\overline{\alpha}$)

Sei (S,Σ) eine Signatur mit $(S,\Sigma)\leq(S,\Sigma)+(S',\Sigma')$, $(S,\Sigma)\leq(S,\Sigma)+(S'',\Sigma'')$ und SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$.
Sei C_a eine Menge von (nicht-typparametrisierten) Clustern,
C_p eine Menge von (zum Teil) typparametrisierten Clustern,
FP_{cl} die Menge der in cleC_p vorkommenden formalen Typparameter.
Sei SErf$(C_p,S',\Sigma',S,\Sigma,\alpha,\gamma)$ und Erf$(C_a,S\cup S'',\Sigma\cup\Sigma'',\overline{\alpha},\overline{\gamma})$ für geeignete Abbildungen α,γ bzw. $\overline{\alpha},\overline{\gamma}$ (vgl. 4.4.1 und 3.1.6).

C ist *Aktualisierung von C_p unter C_a mittels* $\alpha,\overline{\alpha}$:gdw

Für alle typparametrisierten Cluster cleC_p gilt:

1. Für alle formparmeFP_{cl}: ersetze 'formparm' im gesamten Cluster cl textuell und konsistent durch den Bezeichner von $\overline{\alpha}^{-1}(\alpha(\text{formparm}))$.
2. Streiche in der Schnittstelle von cl die formalen Typparameter.

Notation: C:= AKTUAL$(C_p,C_a,\alpha,\overline{\alpha})$.

Wünschenswert wäre die Verträglichkeit von Erf, SErf und AKTUAL, d.h. daß die Aktualisierung einer Menge von parametrisierten Clustern, die $(S,\Sigma)+(S',\Sigma')$ schwach erfüllen, durch eine Menge von nicht parametrisierten Clustern, die $(S,\Sigma)+(S'',\Sigma'')$ erfüllen, eine Clustermenge ergibt, die die resultierende Signatur $(S,\Sigma)+(S',\Sigma')+(S'',\Sigma'')$ erfüllt.

Dies gilt tatsächlich, wie folgendes Lemma zeigt.

4.4.6 Lemma

Sei (S,Σ) eine Signatur mit $(S,\Sigma)\leq(S',\Sigma')+(S,\Sigma)$, $(S,\Sigma)\leq(S,\Sigma)+(S'',\Sigma'')$ und SynZul$(S'',\Sigma'',S,\Sigma,S',\Sigma')$.
Sei C_a eine Menge von nicht-typparametrisierten Clustern,
C_p eine Menge von (zum Teil) typparametrisierten Clustern und
es existieren $\alpha,\gamma,\overline{\alpha},\overline{\gamma}$, so daß SErf$(C_p,S',\Sigma',\alpha,\gamma)$ und Erf$(C_a,S\cup S'',\Sigma\cup\Sigma'',\overline{\alpha},\overline{\gamma})$
C:= AKTUAL$(C_p,C_a,\alpha,\overline{\alpha})$.
Dann gilt:

Erf$(C\cup Ca,S\cup S'\cup S'',\Sigma\cup\Sigma'\cup\Sigma'',\alpha 1,\gamma 1)$, wobei

$\alpha 1$: $C \cup C_a \longrightarrow S \cup S' \cup S''$

$\alpha 1|_{C_a} := \overline{\alpha}$, $\alpha 1|_C := \alpha$

$\gamma 1$: $OP_C \cup OP_{C_a} \longrightarrow \Sigma \cup \Sigma' \cup \Sigma''$

$\gamma 1|_{OP_{C_a}} := \overline{\gamma}$, $\gamma 1|_{OP_C} := \gamma$

(OP_C bzw. OP_{C_a} ist die Menge aller Bezeichner von öffentlichen Operationen in den Clustern von C bzw. C_a)

Beweis:

Sei oBdA $S' \cap S'' = \Sigma' \cap \Sigma'' = C_a \cap C_p = OP_{C_a} \cap OP_{C_p} = \emptyset$, was stets durch Umbenennung erreicht werden kann.

(C, C_p und C_a werden sowohl für die Menge der Cluster als auch die Menge der zugehörigen Clusterbezeichner verwendet).

==> $\alpha 1, \gamma 1$ sind wohldefiniert; $\gamma 1$ ist bijektiv, da γ und $\overline{\gamma}$ bijektiv sind.

$\alpha 1$ ist bijektiv, denn: $\alpha 1|_{C_a} = \overline{\alpha} : C_a \longrightarrow S \cup S''$ ist bijektiv und

$\alpha 1|_C = \alpha : C \longrightarrow S'$ ist bijektiv nach Voraussetzung.

(Beachte: $C = C_p$, falls C und C_p als Mengen von Clusterbezeichnern aufgefaßt werden.)

Verträglichkeit mit der Funktionalität:

Sei $P_f \in OP_C + OP_{C_a}$, $P_f: A'_{s1} \times \ldots \times A'_{sn} \longrightarrow A'_s$, $s1, \ldots, sn, s \in S \cup S' \cup S''$.

1. $P_f \in OP_{C_a}$ ==> $\gamma 1(P_f) = \overline{\gamma}(P_f) = f \in \Sigma_{\overline{\alpha}(s1)\ldots\overline{\alpha}(sn),\overline{\alpha}(s)} \cup \Sigma''_{\overline{\alpha}(s1)\ldots\overline{\alpha}(sn),\overline{\alpha}(s)}$

$= \Sigma_{\alpha 1(s1)\ldots\alpha 1(sn),\alpha 1(s)} \cup \Sigma''_{\alpha 1(s1)\ldots\alpha 1(sn),\alpha 1(s)}$

2. $P_f \in OP_C = OP_{C_p}$

==> $\gamma 1(P_f) = \gamma(P_f)) f \in \Sigma_{\alpha(s1)\ldots\alpha(sn),\alpha(s)} \cup \Sigma'_{\alpha(s1)\ldots\alpha(sn),\alpha(s)}$

$= \Sigma_{\alpha 1(s1)\ldots\alpha 1(sn),\alpha 1(s)} \cup \Sigma'_{\alpha 1(s1)\ldots\alpha 1(sn),\alpha 1(s)}$

MMM

Es kann nun definiert werden, wann ein abstrakter parametrisierter Datentyp durch eine Menge von (zum Teil) typparametrisierte Cluster korrekt implementiert wird.

4.4.7 Defintion (Korrekte Implementierung eines typparametrisierten Datentyps durch eine Menge von Cluster)

Sei PS:=(S,Σ,E,S',Σ',E') eine parametrisierte t-Spezifikation.
Sei Cp eine Menge von (zum Teil) typparametrisierten Clustern.
Cp *implementiert korrekt* PS :gdw

$\forall$(S",Σ",E") mit 1. SynZul(S",Σ",S,Σ,S',Σ')
2. (Sa,Σa,Ea):=(S,Σ,E)+(S",Σ",E") ist eine konsistente t-Spezifikation
3. (Sr,Σr,Er):=(Sa,Σa,Ea)+(S',Σ',E') ist eine konsistente t-Spezifikation

CL_a:={Ca: Ca Menge von Clustern, die (Sa,Σa,Ea) korrekt implementiert}
gilt:

$(\forall Ca \in CL_a)(\exists \alpha 1,\gamma 1,\overline{\alpha}1,\overline{\gamma}1: SErf(Cp,S',\Sigma',S,\Sigma,\alpha 1,\gamma 1) \wedge Erf(Ca,Sa,\Sigma a,\overline{\alpha}1,\overline{\gamma}1))$

C:=AKTUAL(Cp,Ca,α1,$\overline{\alpha}$1) implementiert korrekt (Sr,Σr,Er).

Sei PS:=(S,Σ,E,S',Σ',E') eine parametrisierte t-Spezifikation und Cp eine Menge von typparametrisierten Clustern. Dann ist eine notwendige Voraussetzung dafür, daß Cp PS korrekt implementiert, daß es α und γ gibt, so daß SErf(Cp,S',Σ',S,Σ,α,γ), d.h.Cp erfüllt die notwendigen syntaktischen Bedingungen, die durch (S∪S',Σ∪Σ') vorgegeben sind.

4.1.6.13 besagt, daß die Algebra einer aktuellen Parameterspezifikation unter bestimmten Bedingungen in die Algebra der Ergebnisspezifikation eingebettet werden kann. Dies gilt in ähnlicher Weise auch für die entsprechenden $\overline{EVAL}$-Bilder dieser Algebren, also im Bereich der (nach ≅faktorisierten) Modulalgebren.

4.4.8 Satz

Sei PS:=(S,Σ,E,S',Σ',E') eine parametrisierte t-Spezifikation,
Cp eine Clustermenge, die PS korrekt implementiert.
Sei SP:=(S∪S",Σ∪Σ",E∪E") eine konsistente und vollständige t-Spezifikation mit SynZul(S",Σ",S,Σ,S',Σ') und Ca eine Clustermenge, die SP korrekt implementiert. C:=AKTUAL(Cp,Ca,α,α') für zwei Abbildungen α und α'.
M_C erweitere M_{Ca}, wobei M_C, M_{Ca} die von C bzw. Ca erzeugten Modulalgebren sind.
Dann exisiert ein Σ∪Σ"-Monomorphismus

$$H': \overline{M}_{Ca} \longrightarrow \overline{M}_C$$

Beweis:

Nach 4.1.6.13 ist $T_{\Sigma\cup\Sigma'\cup\Sigma";\sim E\cup E'\cup E"} = F(T_{\Sigma\cup\Sigma";\sim E\cup E"})$ eine t-Erweiterung von $T_{\Sigma\cup\Sigma";\sim E\cup E"}$,

Nach 3.7.7 existiert dann der $\Sigma\cup\Sigma"$-Monomorphismus

$$H': \overline{M}_{Ca} \longrightarrow \overline{M}_{C}$$

⋈⋈⋈

5. LITERATURVERZEICHNIS

[ADJ 75] Goguen, Thatcher,Wagner, Wright:
"An Introduction to Categories, Algebraic Theories and Algebras"
IBM Research Report RC 5369, April 1975

[ADJ 77] Goguen,Thatcher, Wagner:
"An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types"
IBM Research Report RC 6487, New York 1977

[ADJ 78] Thatcher, Wagner, Wright:
"Data Type Specification: Parametrization and the Power of Specification Techniques"
Proceedings, SIGART 10th Annual Symposium on Theory of Computation, p.119 - 132, May 1978

[ALPH 76] Mary Shaw:
"Abstraction and Verification in ALPHARD: Design and Verification of a Tree Handler"
Carnegie-Mellon-University Pittsburgh
Department of Computer Science, June 1976

[ALPH 78] W.A. Wulf (Editor):
"An Informal Definition of ALPHARD"
Carnegie-Mellon-University Pittsburgh
Department of Computer Science, CMU-CS-78-105, February 1978

[CLU 74] B.Liskov, S.Zilles:
"Programming with Abstract Data Types"
ACM Sigplan Notices Vol.9,Nr.4,P.50-59
April 1974

[CLU 77] Liskov,Snyder,Atkinson,Schaffert:
"Abstraction Mechanisms in CLU"
Communications of the ACM Vol.20, Nr.8, P.564-576
August 1977

[CLU 78] Craig, Schaffert:
"A Formal Definition of CLU"
Massachusetts Institute of Technology
Laboratory for Computer Science MIT/LCS/TR-193

[CSSA 77] H.L.Fischer:
"A VDL-Machine for CSSA"
Memo SEKI-77-02 (Diplomarbeit)
Institut für Informatik III, Universität Bonn
Institut für Informatik I, Universität Karlsruhe

[CSSA 79] H.L.Fischer,P.Raulefs:
"Design Rationale for the Interactive Programming Language CSSA"
C.Fachtagung 'Programmiersprachen und Programmiersprachenentwurf'
Springer-Verlag Nr.25, S.111-124

[GOO 76] G.Goos:
"Programmkonstruktion"
Vorlesungsskript Universität Karlsruhe, S.70-79, 110-124
September 1976

[GUT 75] J.Guttag:
"The Specification and Application to Programming of Abstract Data Types"
Technical Report CSRG-59 (Ph.D.Thesis)
September 1975

[HOR 79] G.Hornung:
"Einige Probleme der Algebrasemantik abstrakter Datentypen"
Universität Bonn, Institut für Informatik III
Memo SEKI-BN-79-07 (Diplomarbeit)
Oktober 1979

[HOR 80] Persönliches Gespräch mit Günter Hornung
Universität Bonn, Institut für Informatik III
Februar/März 1980

[RAU 78] P.Raulefs:
"Programmiersprachen und Übersetzerbau II"
Vorlesungsnotizen WS78/79, S.3.28/3.29, 4.50-4.70
Universität Bonn, Institut für Informatik III

[RAU 79] P.Raulefs:
"Einführung in die Theorie der Datenstrukturen"
Vorlesungsnotizen, SS 1979
Universität Bonn, Institut für Informatik III

[STOY] Stoy:
"The Scott/Strachey-Approach to the Mathematical Semantics of Programming Languages"
MIT Press, Hearings in Computer Science, Nr.1
1977