View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

Deduktive Fehlersuche in Abstrakten Datentypen

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Wolfgang Ahrendt

aus Bochum

Tag der mündlichen Prüfung: 20.07.2001

Erster Gutachter: Prof. Dr. Reiner Hähnle, Chalmers University of Technology, Göteborg (Schweden)

Zweiter Gutachter: Prof. Dr. Roland Vollmar, Universität Karlsruhe

Danksagung

Zuallererst möchte ich Herrn Prof. Dr. Reiner Hähnle Dank sagen für die unwahrscheinliche Unterstützung und Förderung, die er mir sowohl als Betreuer dieser Arbeit als auch in vielerlei anderer Hinsicht hat zuteil werden lassen. Seinem unermüdlichen Einsatz, seiner Freude an wissenschaftlicher Arbeit, seinem Teamgeist und nicht zuletzt seinem Vertrauen verdanke ich sehr viel.

Ich hatte zudem das Glück, mit Herrn Prof. Dr. Peter H. Schmitt und Herrn Prof. Dr. Wolfram Menzel zwei weitere Förderer zu haben. Diese beiden haben es mir überhaupt erst ermöglicht, am Institut für Logik, Komplexität und Deduktionssysteme zu arbeiten. Sie haben mich jederzeit aktiv unterstützt und großen Wert auf eine intensive und beiderseitige Zusammenarbeit gelegt. Das dadurch entstehende Klima empfand ich als sehr anregend.

Besonders bedanken möchte ich mich an dieser Stelle auch bei Dr. Bernhard Beckert, für seine auch im Detail nicht nachlassende Schärfe in der Diskussion, viele gute Fragen, und für seine enorme Kollegialität. Desweiteren danke ich all meinen Kollegen, besonders Thomas Baar, Martin Giese, Dr. Elmar Habermalz, Lilian Heck und Dr. Matthias Ott, für zahlreiche Diskussionen und das sehr angenehme Arbeitsumfeld. Auch mein Informatiker-Freund Dr. Christoph Zenger war für mich immer ein wertvoller Gesprächspartner. Mein Dank gilt auch Sonja Pieper für den Einsatz, mit dem sie die Ergebnisse der vorliegenden Arbeit implementiert hat.

Also, I am indebted to Prof. Dr. Ryuzo Hasegawa and Prof. Dr. Hiroshi Fujita for many fruitful discussions and for actively supporting my usage of their tool MGTP. They even included new features due to my demands. I am much obliged to both.

Ich danke Herrn Prof. Dr. Roland Vollmar für die Übernahme des Korreferates und das Interesse, welches er dieser Arbeit entgegengebracht hat.

Mein besonders herzlicher Dank gilt meinen Eltern für all ihre Unterstützung bei dem Zustandekommen dieser Arbeit, wo sie als Eltern und Großeltern oft gebraucht wurden, sowie für das gründliche Korrektur-Lesen des Textes.

Für meinen Dank an meine Frau Christine und meine Kinder Michael und Philipp schließlich fehlen mir fast die Worte. Alle drei haben mir in unbeschreiblicher Weise geholfen, haben Höhen und Tiefen mit mir geteilt, und mußten unendlich viel Geduld aufbringen. Diese drei Menschen waren mir ein sicherer Rückhalt. Vielleicht läßt sich dies durch ein Detail am besten wiedergeben: wenn ich mich morgens vor der Schule von Michael verabschiede, dreht er sich immer noch einmal um, winkt und ruft:
„Tschüß Papa, gute Arbeit, und paß auf daß dein Computer nicht abstürtzt.“

Inhaltsverzeichnis

1 Einleitung **1**

2 Abstrakte Datentypen und Nicht-Folgerbarkeit **5**

2.1 Mathematische Grundkonzepte 6

2.2 Frei erzeugte Datentypen 7

2.2.1 Syntax 7

2.2.2 Semantik 13

2.3 Spezifikation frei erzeugter Datentypen 19

2.3.1 Syntax 20

2.3.2 Semantik 26

2.4 Nicht-Folgerbarkeit 36

2.5 Normalisierung von Spezifikationen 43

2.6 Diskussion 55

2.6.1 Allgemeine Algebren und Konstruktorterm-Algebren . . . 56

2.6.2 Fehlender Isomorphieabschluß 58

2.6.3 Lose Semantik vs. monomorphe Ansätze 61

3 Modell-Generierung **65**

3.1 Interpretations-Schließen 66

3.2 Erzeugung der Modell-Generierungs-Regeln 73

3.2.1 Transformation der Signatur 74

3.2.2 Transformation der Axiome 79

3.2.3 Optimierte Transformation der Literale 89

3.2.4 Transformation der Spezifikation 91

3.3 Beschränkte Instantiierung der Axiome 92

3.3.1 Grundinstantiierte Formelmengen 93
3.3.2 Transformation der Instanzen 96
3.4 Anwendung der Modell-Generierungs-Regeln 97
3.4.1 Bestandteile von Modell-Generierungs-Regeln 97
3.4.2 Die Modell-Generierungs-Prozedur 103

4 Korrektheit und Vollständigkeit 109
4.1 Eigenschaften der Transformation 110
4.2 Eigenschaften von Meta-Atom-Mengen 114
4.3 Beschränkte Modell-Korrektheit 121
4.4 Unerfüllbarkeits-Korrektheit 130
4.5 Fairneß und Modell-Vollständigkeit 138
4.5.1 Fairneß 138
4.5.2 Modell-Vollständigkeit 139
4.5.3 Nachweis der beschränkten Ast-Länge 143

5 Realisierung und Beispiele 157
5.1 Realisierung des Verfahrens 158
5.2 Beispiele 163
5.2.1 NatStack 163
5.2.2 p_Forever 177
5.2.3 MergeSort 179

6 Verwandte Arbeiten 183
6.1 Konstruktion und Repräsentation von Modellen 183
6.2 Widerlegung und Korrektur falscher Aussagen 186

7 Zusammenfassung und Ausblick 189

Literaturverzeichnis 191

Index 197

1 Einleitung

Abstrakte Datentypen (ADTs) sind ein in der Informatik weit verbreitetes Mittel zur Modellierung von Datenstrukturen und ihrer Manipulation. Zuallererst dienen sie der abstrakten Beschreibung von Datenoperationen. Solche Beschreibungen bilden eine Schnittstelle zwischen der Implementierung und der Benutzung der Operationen. Nach *innen* ist ein abstrakter Datentyp (genauer: seine Beschreibung) eine Anforderungsspezifikation für die *Implementierung* der Strukturen und Operationen. Nach *außen* hin stellt er einen Vertrag dar, d.h. eine Sammlung von Eigenschaften, auf die man sich bei der *Benutzung* der Operationen verlassen kann (bzw. können sollte).

Zur Beschreibung der Eigenschaften abstrakter Datentypen bedient man sich logischer Formelsprachen. Aus diesem Grund sind ADTs besonders geeignet für die Anwendung mathematischer Logik. Entsprechend groß ist ihre Bedeutung im Gebiet der sogenannten *formalen Methoden*, dem Zweig der Informatik, der sich erstens mit der mathematisch präzisen Modellierung von Software und Hardware beschäftigt, sowie zweitens mit der Anwendung von Computer-gestützten Beweismethoden auf solche Modellierungen.

Die vorliegende Abhandlung zielt insbesondere auf den Einsatz abstrakter Datentypen im Umfeld von formalen Methoden für *Software*. Beispielsweise ist die hier vorgestellte Methode als Baustein innerhalb der Computer-gestützten Software-Verifikation einsetzbar. Dort werden Aussagen über Programme mittels Programm-Logik zurückgeführt auf Aussagen über die im Programm verwendeten Datentypen. Entsprechend können Fehler in einem Programm zurückgeführt werden auf *fehlerhafte Aussagen* über Datentypen. Solche fehlerhaften Aussagen zu identifizieren ist das Grundanliegen, welches in dieser Abhandlung verfolgt wird.

Syntaktisch sind abstrakte Datentypen gegeben durch eine Signatur sowie durch eine Menge von Formeln, die Axiome des Datentyps. Wir verfolgen den Ansatz von *konstruktor-basierten* Datentypen, in welchen die Signatur zerfällt in Konstruktoren und (Nicht-Konstruktor-)Funktionen. Die Konstruktoren repräsentieren hierbei die Strukturelemente der Daten, während die Funktionen für Operationen auf den Daten stehen. Diese Unterscheidung ist von überragender Bedeutung für die folgenden semantischen und methodischen Diskussionen.

Die Axiome eines Datentyps sind Formeln einer Gleichungslogik und beschreiben seine *vorgegebenen Eigenschaften*. Neben den Axiomen gelten in dem Datentyp noch andere Formeln, nämlich alle, die aus den Axiomen folgen. Diese beschreiben *abgeleitete Eigenschaften* des Datentyps. Für jede Formel φ (passender Signatur) können wir die Frage stellen, ob φ aus den Axiomen AX folgt, oder abgekürzt, ob $AX \models \varphi$ zutrifft, wobei '$\models$' das Symbol für die *Folgerbarkeit* ist.

Die Untersuchung des Folgerbarkeits-Begriffs und der damit zusammenhängenden Beweisbarkeit fällt in das Gebiet der mathematischen Logik. Neben der Mathematik hat sich auch längst die Informatik dieses Gebietes angenommen, zum einen, weil die Informatik viele Anwendungsfelder für die Logik bietet (siehe die oben erwähnten formalen Methoden), zum anderen, weil umgekehrt die Logik ein Anwendungsfeld für die Informatik ist (siehe die oben erwähnten Computergestützten Beweissysteme).

Sowohl für die Theoriebildung der Logik als auch für die Beweissysteme gilt, daß ganz überwiegend der *Nachweis der Folgerbarkeit* im Mittelpunkt steht. Diese Abhandlung widmet sich hingegen der Untersuchung der *Nicht-Folgerbarkeit*, denn darin manifestieren sich die *Fehler*, von denen im Titel gesprochen wird. Wenn nämlich eine eigentlich vermutete Eigenschaft φ *nicht* aus den Axiomen eines Datenyps folgt, wenn also $AX \not\models \varphi$ gilt, *dann ist ein Fehler aufgetreten.* Entweder, dieser befindet sich in der Eigenschaft φ, die es dann zu korrigieren gelte, oder er befindet sich in den Axiomen. Letzteres ist der Fall, wenn die Eigenschaft φ der eigentlichen Intention entspricht und der Fehler darin liegt, daß die Axiome diese Intention nicht garantieren. Da sowohl Spezifikationen als auch ihre Implementierungen sehr häufig fehlerhaft sind, ist eine Unterstützung bei der Aufdeckung von Fehlern sehr relevant für die Verbesserung der Zuverlässigkeit von Systemen.

Falls für bestimmte Axiome AX und eine bestimmte Formel φ gilt, daß $AX \not\models \varphi$ zutrifft, dann könnte es sein, daß das Gegenteil von φ folgerbar ist, d.h. daß $AX \models \neg\varphi$ zutrifft.[1] In diesem Fall kämen wir, sowohl in der Theorie als auch bei den Systemen, mit klassischen Methoden zum Nachweis der Folgerbarkeit aus. Anstelle von φ muß man einfach $\neg\varphi$ beweisen. Der interessantere, weil schwierigere Fall tritt ein, wenn weder φ noch $\neg\varphi$ folgerbar sind, d.h. wenn *weder* $AX \models \varphi$ *noch* $AX \models \neg\varphi$ zutreffen. In diesem Fall kann man $AX \not\models \varphi$ nicht mit klassischen Beweisverfahren nachweisen. Was man stattdessen tun muß, wird deutlicher bei genauerer Betrachtung des Begriffs 'folgerbar'. In den meisten Ansätzen, so auch hier, bedeutet $AX \models \varphi$, daß *in allen* Modellen der Axiome (und der Signatur) die Formel φ gilt. Umgekehrt bedeutet $AX \not\models \varphi$, daß ein Modell der Axiome *existiert*, in dem φ nicht gilt. In diesem Modell gilt dann sowohl AX als auch (möglicherweise wider Erwarten) $\neg\varphi$. Um also $AX \not\models \varphi$ zu belegen, muß ein Mo-

[1] So einfach, wie es hier steht, stimmt das nur für geschlossene Formeln. Der allgemeine Fall wird später ausführlich beschrieben.

dell für $AX \cup \{\neg\varphi\}$ gefunden werden. *Die Suche nach genau solchen Modellen bildet den Kern der hier vorgestellten Methode.*

Die Betrachtung von $AX \cup \{\neg\varphi\}$ erinnert stark an die bekannten, widerspruchsbasierten Beweisverfahren, wie Resolution oder Tableaux. Dort sucht man $AX \models \varphi$ zu belegen, indem man die *Inkonsistenz* von $AX \cup \{\neg\varphi\}$ beweist. In Analogie hierzu suchen wir $AX \not\models \varphi$ zu belegen, indem wir die *Konsistenz* von $AX \cup \{\neg\varphi\}$ nachweisen, und zwar durch Angabe eines Modells. (Allerdings gelingt dies nur näherungsweise, s.u.). Die Analogie zu widerspruchsbasierten Beweisverfahren geht noch weiter. Bei der Analyse der Formelmenge $AX \cup \{\neg\varphi\}$ setzen wir ein zur Familie der Tableaux zu zählendes Verfahren namens 'model generation' (engl.) ein, allerdings nicht direkt, sondern erst nach einer substantiellen Transformation des Problems. 'model generation' ist deswegen ein für unsere Zwecke gut geeignetes Verfahren, weil es ausgerichtet ist auf *Terminierung* im Falle einer konsistenten Eingabe, während viele andere Tableau-Varianten ausgerichtet sind auf Terminierung im Falle inkonsistenter Eingaben. Da unser Hauptanliegen die Konstruktion von Modellen ist, haben die semantischen Festlegungen, wie auch immer sie aussehen, einen erheblichen Einfluß auf das weitere Vorgehen.

Zuallererst ist hier die Festlegung auf *termerzeugte Modelle* zu nennen. Gerade hierin unterscheiden sich abstrakte Datentypen von allgemeinen Theorien beispielsweise der erststufigen Prädikatenlogik. Termerzeugte Modelle sind solche, in denen alle Elemente des Grundbereiches durch Terme 'benennbar' sind. Diese Eigenschaft wird gerne auch mit 'no junk' bezeichnet. Eine termenerzeugte Semantik bietet die Möglichkeit, sogenannte Nicht-Standard-Modelle auszuschließen. Auf diese Weise können wir überhaupt erst über Datenstrukturen wie natürliche Zahlen, Listen, Bäume usw. sprechen (ohne eine höherstufige Formelsprache verwenden zu müssen). In diesem Punkt stellen wir uns ganz in die Tradition des Gebietes der algebraischen Spezifikation.

Eine weitere Festlegung, die wir treffen, ist die Verwendung der sog. *losen Semantik*. Hier werden als Semantik einer Axiomen-Menge *all* ihre (termenerzeugten) Modelle betrachtet, nicht nur ein bestimmtes, wie beim initialen Ansatz. Der 'lose' Ansatz liegt näher an der Tradition der Logik. Wir aber verwenden ihn, weil er zu Zwecken der *Anforderungsspezifikation* besser geeignet ist als der 'initiale'. Darüberhinaus erlaubt uns der lose Ansatz auf der Seite der Syntax die Verwendung beliebiger Disjunktionen und Quantoren.

Die dritte semantische Festlegung ist diejenige auf '*frei* erzeugte' Modelle. Dieser, der (universellen) Algebra entspringende Begriff hat in unserem konstruktorbasierten Ansatz die folgende Bedeutung: zwei verschiedene *Konstruktorterme* bezeichnen zwei verschiedene Elemente des Grundbereiches. Diese Eigenschaft wird auch mit 'no confusion' bezeichnet. Viele wichtige Datentypen sind 'frei', wie natürliche Zahlen, Listen oder Bäume. Nicht 'frei' sind z.B. Mengen. Die Einschränkung auf freie Datentypen wurde der Machbarkeit wegen getroffen. Sie

befreit uns von der Notwendigkeit, bei der Modellkonstruktion Äquivalenzklassen zu bilden. Der Grundbereich freier Datentypen liegt von Anfang an fest. Verschiedene Modelle unterscheiden sich 'nur' in der *Interpretation der Funktionssymbole*. Damit ist klar, was das zu beschreibende Verfahren im Kern tun wird: es realisiert eine *Suche nach Interpretationen*.

Die vorliegende Arbeit ist nicht die erste, die sich mit der Konstruktion von Modellen zum Zweck des Nachweises von Nichtfolgerbarkeit beschäftigt. Sie ist aber nach bestem Wissen des Autors der bisher einzige Beitrag, der diese Fragestellung für termerzeugte Datentypen bearbeitet. Diese informatik-nahe, anwendungsbezogene Ausrichtung hat ihren Preis. Wenn die Konstruktoren rekursiv sind, und dies ist der Normalfall, dann sind die (im vorhinein festgelegten) Grundbereiche *unendlich*. Wir sehen uns also vor der Schwierigkeit, auf einem unendlichen Bereich eine (passende) Interpretation zu finden und zu beschreiben. Prinzipiell kann man das natürlich tun, und zwar durch Angabe eines Programms (sei es in Pascal geschrieben oder ein λ-Term). Die Konstruktion eines solchen Programms kann man von einem Programmierer erwarten, nicht aber von einem automatischen Verfahren.

Da wir eine automatische Unterstützung bei der Aufdeckung von Fehlern anstreben, müssen wir hier Abstriche machen. Die Lösung besteht darin, eine Interpretation nicht komplett, sondern nur teilweise anzugeben. Damit kann zwar die Konsistenz einer Formelmenge (bei uns der Menge $AX \cup \{\neg\varphi\}$) nicht abschließend bewiesen werden. Die vorgestellte Methode ist aber ein Semi-Entscheidungsverfahren für die Konsistenz der *Instantiierung* einer Formelmenge mit beliebig großen, aber *endlichen Mengen von Instanzen*. Das bedeutet: ist eine Formelmenge gegeben (etwa $AX \cup \{\neg\varphi\}$) und ist eine endliche Menge von Instanzen für die Variablen der Formelmenge gegeben, dann gilt: falls die entsprechende Instantiierung der Formelmenge konsistent ist, dann wird von dem Verfahren ein Modell gefunden. Für die ursprüngliche Frage, ob $AX \models \varphi$ gilt, bedeutet dies: falls wir ein Modell der beschränkten Instantiierung von $AX \cup \{\neg\varphi\}$ finden, dann gibt es gute Gründe, die Gültigkeit von $AX \models \varphi$ zu bezweifeln. Da die Ausgabe des Verfahrens in (einem Ausschnitt) der Interpretation der Funktionen besteht, hat der Benutzer, oder auch der Autor eines abstrakten Datentyps einen konkreten Hinweis darauf, *warum* eine vermutete Folgerbarkeit von φ aus AX möglicherweise *nicht* besteht. Im Falle eines Fehlers tritt dieser dann für den Betrachter offen zutage.

2 Abstrakte Datentypen und Nicht-Folgerbarkeit

Dieses Kapitel behandelt die syntaktischen und semantischen Grundbegriffe frei erzeugter Datentypen und ihrer Spezifikation. Obwohl wir uns dabei im Prinzip bekannter Konzepte bedienen, so sind dennoch die Definitionen in starkem Maße zugeschnitten auf den Kontext, dem diese Arbeit gewidmet ist. Hier ist insbesondere zu nennen, daß wir uns auf *frei erzeugte* Datentypen beschränken, und daß wir speziell für diese die Nicht-Folgerbarkeit von Aussagen belegen wollen durch die Konstruktion *frei erzeugter (Gegen-)Modelle*. In diesem Kontext ist es von Vorteil, die freie Erzeugtheit nicht als Spezialfall im Rahmen allgemeinerer Datentypen zu betrachten, sondern sozusagen fest einzubauen. Inwiefern die folgenden Begriffe dies tun und welche Vorteile damit verbunden sind, das wird am Ende des Kapitels in Abschn. 2.6 im Vergleich mit klassischen Definitionen diskutiert.

Schon an dieser Stelle sei auf Folgendes hingewiesen:

Bemerkung 2.0.1 (Trennung von Funktionen und Konstruktoren)
Die Konstruktoren eines Datentyps und die daraus aufgebauten Terme sind durch die gesamte Arbeit hindurch von so herausragender Bedeutung, daß wir die Konstruktoren in ungewöhnlich scharfer Weise von den übrigen Funktionen unterscheiden. Konkret heißt dies: wir werden nicht, wie üblich, bestimmte Funktionen als Konstruktoren auszeichnen (s. z.B. [LEW96, Def. 3.24]). Stattdessen werden Funktionen und Konstruktoren völlig voneinander getrennt.

Diese Sichtweise ist zunächst einmal etwas ungewöhnlich, sie wird sich aber für unseren Kontext als sehr nützlich erweisen. Nebenbei bleibt es uns auf diese Weise erspart, durch die ganze Arbeit hindurch von ‘Nicht-Konstruktor-Funktionen’ zu sprechen, was wir ansonsten an sehr vielen Stellen tun müßten.

Ein weiteres Thema dieses Kapitels sind die ersten, vorbereitenden Schritte zur Untersuchung der Nicht-Folgerbarkeit, und zwar die Konstruktion von Gegenspezifikationen sowie deren Normalisierung.

2.1 Mathematische Grundkonzepte

Hier vereinbaren wir einige wenige Notationen für sehr elementare mathematische Grundbegriffe.

Notation 2.1.1 (Mengen)
Seien M und N Mengen und a_i Elemente. Wir verwenden die üblichen Notationen für Mengenoperationen:

- $\emptyset$ oder $\{\}$ für die leere Menge,
- $a \in M$ für die Elementbeziehung, $a \notin M$ für die Negation derselben,
- $M \subseteq N$ für die (nicht strikte) Teilmengenbeziehung, $M \nsubseteq N$ für die Negation derselben,
- $M \cup N$ für die Vereinigung,
- $M \cap N$ für den Durchschnitt,
- $M \setminus N$ für die Mengendifferenz (die Menge der Elemente von M, die nicht Element von N sind),
- $\{a_1, \ldots, a_n\}$ für diejenige endliche Menge, die ausschließlich die Elemente a_1 bis a_n enthält.
- $\{\mathit{elem} \mid \mathit{Eig}\}$ für die Menge aller Elemente mit der Eigenschaft Eig,
- $M_1 \times \ldots \times M_n$ für das kartesische Produkt der Mengen $M_1, \ldots, M_n$ (die Menge aller Tupel $\langle a_1, \ldots, a_n \rangle$ mit $a_i \in M_i$),
- $|M|$ bezeichnet die Anzahl der Elemente der Menge M. Ist M endlich, so schreiben wir auch $|M| < \infty$,
- $\mathbb{N}$ bezeichnet die Menge der natürlichen Zahlen $\{0, 1, 2, \ldots\ldots\}$,
- Zu einer endlichen Menge $M \subseteq \mathbb{N}$ bezeichnet $max(M)$ ihr maximales Element.
- Zu einer Menge M bezeichnet $\wp(M)$ die Menge aller Teilmengen von M.

⋆

Notation 2.1.2 (Familien,deren Vereinigung)
Zu einer (Index-)Menge I bezeichnet $\{\mathbf{M_i}|\ \mathbf{i} \in \mathbf{I}\}$ eine **I-indizierte Familie**. Diese besteht aus je einem Mitglied M_i pro Index $i \in I$. Ist $\mathcal{X}$ eine Familie, deren Mitglieder alle Mengen sind, dann bezeichnet $\overline{\mathcal{X}}$ die **Vereinigung aller Mengen der Familie**, d.h. $\overline{\{M_i|\ i \in I\}} = \bigcup_{i \in I} M_i$. Wenn eine Familie angegeben wird durch explizite Aufzählung ihrer Mitglieder, dann benutzen wir die gleiche Schreibweise wie für endliche Mengen (s.o.). ⋆

Notation 2.1.3 (Wörter, das leere Wort)
Zu einer Menge M ist $\boldsymbol{M^*}$ die Menge der endlichen **Wörter** über M. λ ist das **leere Wort**. Wo es die Lesbarkeit unterstützt, werden Wörter geklammert mittels '[' und ']'. ⋆

(Da in dieser Abhandlung logische Formelsprachen eine große Rolle spielen, lohnt es sich vielleicht, an dieser Stelle darauf hinzuweisen, daß die obigen Symbole für Mengenoperationen hier *nicht* Teil der logischen Objektsprache sind, sondern ausschließlich der Abkürzung natürlicher Sprache dienen. Wir betrachten in dieser Arbeit i.a. keine Logik der Mengen oder Wörter; wenn doch, dann nicht unter Verwendung dieser Symbole.)

Notation 2.1.4 (genau dann wenn)
Das Zeichen '$\Longleftrightarrow$' verwenden wir als Abkürzung für '**genau dann wenn**'.
'$:\Longleftrightarrow$' bedeutet '**per Definition genau dann wenn**'. ⋆

(Auch '$\Longleftrightarrow$' ist *nicht* Teil der logischen Objektsprache.)

2.2 Frei erzeugte Datentypen

2.2.1 Syntax

Zunächst definieren wir Signaturen für abstrakte Datentypen. Wie angekündigt, werden Konstruktoren von Anfang an von den Funktionen getrennt.

Definition 2.2.1 (Signaturen, Sorten, Konstruktoren, Funktionen)
Eine **ADT-Signatur $\boldsymbol{\Sigma}$** ist ein Tupel $(S, \mathcal{C}, \mathcal{F}, \alpha)$. Hierbei ist S eine endliche Menge von **Sorten**, $\mathcal{C} = \{C_s|\ s \in S\}$ eine Familie von disjunkten, S-indizierten Mengen von **Konstruktoren** und $\mathcal{F} = \{F_s|\ s \in S\}$ eine Familie von disjunkten, S-indizierten Mengen von **Funktionen** ($\overline{\mathcal{C}} \cap \overline{\mathcal{F}} = \emptyset$). $\overline{\mathcal{C}} \cup \overline{\mathcal{F}}$ ist endlich. Ist $c \in C_s$ bzw. $f \in F_s$, dann heißt s die **Zielsorte** von c bzw. f. Jedem Konstruktor bzw. jeder Funktion weist $\alpha : \overline{\mathcal{C}} \cup \overline{\mathcal{F}} \to S^*$ seine/ihre **Argumentsorten** zu. ⋆

Bemerkung 2.2.2 Der Begriff ‘Funktion’ wird hier immer für syntaktische Zeichen verwendet, ebenso wie der Begriff ‘Konstruktor’. Auf der Seite der Semantik werden wir von ‘Abbildungen’ sprechen, bzw. von ‘Interpretationen’, welches bestimmte, mit Funktionen assoziierte Abbildungen sind, siehe Abschn. 2.2.2.

Konstanten werden, wie üblich, als 0-stellige Funktionen aufgefaßt. Davon zu unterscheiden sind die 0-stelligen Konstruktoren.

Definition 2.2.3 (Basiskonstruktoren, Konstanten)
Sei Σ eine ADT-Signatur $(S, \mathcal{C}, \mathcal{F}, \alpha)$ mit $s \in S$. Ein **Basiskonstruktor** der Sorte s ist ein Element $c \in C_s$ mit $\alpha(c) = \lambda$. Eine **Konstante** der Sorte s ist ein Element $a \in F_s$ mit $\alpha(a) = \lambda$. ⋆

Beispiel 2.2.4 Als Beispiel betrachten wir eine Signatur $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$, mittels derer über den abstrakten Datentyp Stack gesprochen werden kann. Die Elemente des Stacks sind, der Einfachheit halber, natürliche Zahlen. Wir benötigen also zwei Sorten:

$$S = \{Nat, Stack\}$$

Alle natürlichen Zahlen werden *konstruiert* durch zero und succ (Nachfolger), alle Stacks durch nil und push:

$$\mathcal{C} = \{\{\mathsf{zero}, \mathsf{succ}\}_{Nat}, \{\mathsf{nil}, \mathsf{push}\}_{Stack}\}$$

Die Argumentsorten von succ und push sind:

$$\alpha(\mathsf{succ}) = Nat \quad \alpha(\mathsf{push}) = [Nat\ Stack]$$

zero und nil sind die Basiskonstruktoren:

$$\alpha(\mathsf{zero}) = \alpha(\mathsf{nil}) = \lambda$$

Wir betrachten folgende Funktionen: die Vorgängerfunktion *pred* auf den natürlichen Zahlen, *top* für das oberste Element eines Stacks, *pop* für das Entfernen des obersten Elementes, sowie *del* zum Löschen einer Zahl aus einem Stack. Die Funktionen werden, entsprechend der Def. 2.2.1, in $\mathcal{F}$ nach ihren Ergebnissorten ‘sortiert’:

$$\mathcal{F} = \{\{pred, top\}_{Nat}, \{pop, del\}_{Stack}\}$$

Die Argumentsorten dieser Funktionen ergeben sich aus:

$$\alpha(pred) = Nat \quad \alpha(top) = \alpha(pop) = Stack \quad \alpha(del) = [Nat\ Stack]$$

⋆

(Das Wort ‘Stack’ ist ein Beispiel für viele Fachbegriffe der Informatik, die zur englischen Sprache gehören. Man kann natürlich auch deutsche Übersetzungen verwenden. Wo dies geschieht, wird im Falle des ‘Stacks’ gerne vom ‘Keller’ gesprochen. Ein echter Keller aber ist nach keiner Seite offen, der Stack hingegen intuitiv sehr wohl. Passender wäre die wörtliche Übersetzung ‘Stapel’, die ist aber in der Terminologie der Datenstrukturen noch weniger üblich. Terminologische Probleme, die aus dem komplizierten Verhältnis von Muttersprache und einer fremdsprachlich geprägten Fachsprache

erwachsen, können in dieser Abhandlung nicht gelöst werden. Stattdessen werden die Bezeichner für Sorten, Konstruktoren und Funktionen hier schlicht so gewählt, wie sie häufig auch anderswo benutzt werden.)

Dieses Beispiel zeigt auch, daß die Notierung von Signaturen streng nach Def. 2.2.1 nicht besonders übersichtlich ist. Die Schreibwiese mit indizierten Familien von Mengen und, davon getrennt, mit α ist auch nicht als Spezifikationssprache gedacht, sondern dient der Theoriebildung, z.B. der induktiven Definition von Termen (Def. 2.2.9). Für die übersichtliche Darstellung verwendet man andere Schreibweisen, die wir später im Zusammenhang mit Spezifikationen abstrakter Datentypen einführen.

Definition 2.2.5 (Signaturerweiterung, Teilsignatur)
Seien $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ und $\Sigma' = (S', \mathcal{C}', \mathcal{F}', \alpha')$ zwei Signaturen.
Σ' ist eine **Signaturerweiterung** von Σ, kurz '$\boldsymbol{\Sigma \subseteq \Sigma'}$', falls

- $S \subseteq S'$,
- für alle $s \in S$ gilt $C_s \subseteq {C_s}'$und $F_s \subseteq {F_s}'$,
- für alle $l \in \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$ gilt $\alpha(l) = \alpha'(l)$.

Umgekehrt heißt Σ **Teilsignatur** von Σ'. $\star$

Beispiel 2.2.6 Sei $\Sigma_1 = (S_1, \mathcal{C}_1, \mathcal{F}_1, \alpha_1)$, die Signatur der natürlichen Zahlen, gegeben durch $S_1 = \{Nat\}$, $\mathcal{C}_1 = \{\{\mathsf{zero}, \mathsf{succ}\}_{Nat}\}$, $\mathcal{F}_1 = \{\{pred\}_{Nat}\}$ und $\alpha_1(\mathsf{succ}) = \alpha_1(pred) = Nat$, $\alpha_1(\mathsf{zero}) = \lambda$. Sei $\Sigma_2 = \Sigma$ aus Beispiel 2.2.4, die Signatur des Stacks natürlicher Zahlen. Dann ist Σ_1 eine Teilsignatur von Σ_2, '$\Sigma_1 \subseteq \Sigma_2$'. $\star$

Eine spezielle Erweiterung ist diejenige, in der nur Funktionssymbole hinzukommen.

Definition 2.2.7 (Funktionserweiterung)
Sei $\Sigma' = (S', \mathcal{C}', \mathcal{F}', \alpha')$ eine Signaturerweiterung von $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$.
Σ' heißt **Funktionserweiterung** von Σ, falls $S = S'$ und $\mathcal{C} = \mathcal{C}'$. $\star$

Definition 2.2.8 (Variablen)
Für eine ADT-Signatur $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ ist $\mathcal{V}(\Sigma) = \{V_s | \ s \in S\}$ eine Familie von disjunkten, S-indizierten Mengen von Variablen $(\overline{\mathcal{V}(\Sigma)} \cap (\overline{\mathcal{C}} \cup \overline{\mathcal{F}}) = \emptyset)$.
$\boldsymbol{V_s}$ ist die Menge der **Variablen der Sorte $\boldsymbol{s}$**.
$\boldsymbol{V_\Sigma} = \overline{\mathcal{V}(\Sigma)}$ ist die Menge der **$\boldsymbol{\Sigma}$-Variablen**.
(Zusätzlich verlangen wir von $\mathcal{V}$ die Verträglichkeit mit dem '$\subseteq$' auf Signaturen: falls $\Sigma \subseteq \Sigma'$, dann $V_s \subseteq {V_s}'$ für alle $s \in S$.) $\star$

Wenn im folgenden im Zusammenhang mit einer Signatur Σ die Rede ist von V_s, dann ist damit ein entsprechendes Mitglied der Familie $\mathcal{V}(\Sigma)$ gemeint.

Eine ADT-Signatur induziert verschiedene Termmengen, die im folgenden definiert werden. Einige dieser Mengen benötigen wir zum Aufbau der Formelsyntax, andere spielen bei der Beschreibung der Semantik eine große Rolle. Schließlich wird auch die in dieser Abhandlung dargelegte Methodik ausgiebig Gebrauch machen von der Charakterisierung der Terme durch diese Mengen. Wir werden unterscheiden zwischen a) allgemeinen Termen, b) Termen, die nur aus Konstruktoren und Variablen aufgebaut sind (also keine Funktionen enthalten), und c) Termen, in denen ausschließlich Konstruktoren auftreten. Außerdem werden Terme nach Sorten 'sortiert'.

Definition 2.2.9 (Terme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur.
$\mathcal{T}_\Sigma = \{T_s |\ s \in S\}$ ist die Familie von minimalen, S-indizierten Mengen, für die gilt:

- $V_s \subseteq T_s$,
- falls $l \in C_s \cup F_s$ und $\alpha(l) = \lambda$, dann $l \in T_s$,
- falls $l \in C_s \cup F_s$, $\alpha(l) = s_1 \dots s_n$ und $\langle t_1, \dots, t_n \rangle \in T_{s_1} \times \dots \times T_{s_n}$, dann $l(t_1, \dots, t_n) \in T_s$.

$\boldsymbol{T_s}$ ist die Menge der **Terme der Sorte $\boldsymbol{s}$**.
Ist $t \in T_s$, dann ist $\boldsymbol{sort}(t) = s$.
$\boldsymbol{T_\Sigma} = \overline{\mathcal{T}_\Sigma}$ ist die Menge der **$\boldsymbol{\Sigma}$-Terme**. ⋆

Definition 2.2.10 (funktionsfreie Terme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur.
$\mathcal{FFT}_\Sigma = \{FFT_s |\ s \in S\}$ ist die Familie von minimalen, S-indizierten Mengen, für die gilt:

- $V_s \subseteq FFT_s$,
- falls $c \in C_s$ und $\alpha(c) = \lambda$, dann $c \in FFT_s$,
- falls $c \in C_s$, $\alpha(c) = s_1 \dots s_n$ und $\langle t_1, \dots, t_n \rangle \in FFT_{s_1} \times \dots \times FFT_{s_n}$, dann $c(t_1, \dots, t_n) \in FFT_s$.

$\boldsymbol{FFT_s}$ ist die Menge der **funktionsfreien Terme der Sorte $\boldsymbol{s}$**.
$\boldsymbol{FFT_\Sigma} = \overline{\mathcal{FFT}_\Sigma}$ ist die Menge der **funktionsfreien $\boldsymbol{\Sigma}$-Terme**. ⋆

Definition 2.2.11 (Konstruktorterme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur.
$\mathcal{CT}_\Sigma = \{CT_s | \ s \in S\}$ ist die Familie von minimalen, S-indizierten Mengen, für die gilt:

- falls $c \in C_s$ und $\alpha(c) = \lambda$, dann $c \in CT_s$,
- falls $c \in C_s$, $\alpha(c) = s_1 \ldots s_n$ und $\langle t_1, \ldots, t_n \rangle \in CT_{s_1} \times \ldots \times CT_{s_n}$, dann $c(t_1, \ldots, t_n) \in CT_s$.

$\boldsymbol{CT_s}$ ist die Menge der **Konstruktorterme der Sorte $\boldsymbol{s}$**.
$\mathcal{CT}_\Sigma$ ist die **Familie der Σ-Konstruktorterme**.
$\boldsymbol{CT_\Sigma} = \overline{\mathcal{CT}_\Sigma}$ ist die **Menge der Σ-Konstruktorterme**. ⋆

(In den Notationen T_s, FFT_s und CT_s verbirgt sich eine leichte Ungenauigkeit. Die ausschließliche Verwendung der Sorte als Index unterschlägt die Abhängigkeit dieser Termmengen von der jeweiligen Signatur Σ. Dies erhöht die Lesbarkeit und betont den Unterschied z.B. zwischen den Termen T_s einer Sorte einerseits und allen Termen T_Σ einer Signatur andererseits. Die jeweilige Signatur ergibt sich im Text aus dem Kontext.)

Faktum 2.2.12 *Seien Σ und Σ' zwei ADT-Signaturen mit $\Sigma \subseteq \Sigma'$. Dann gilt $T_\Sigma \subseteq T_{\Sigma'}$, $FFT_\Sigma \subseteq FFT_{\Sigma'}$ und $CT_\Sigma \subseteq CT_{\Sigma'}$.*

Wir benötigen ein paar Operationen auf Termen.

Definition 2.2.13 (Argumentselektion aus Termen)
Für einen Term $t \in T_\Sigma$ mit mindestens i Argumenten bezeichnet $t{\downarrow_i}$ das **i-te Argument** von t. Es gilt also:

$$l(t_1, \ldots, t_i, \ldots, t_n){\downarrow_i} = t_i$$

⋆

Definition 2.2.14 (Größe von Konstruktortermen)
Für Konstruktorterme $ct \in CT_\Sigma$ ist $|ct|$ die **Größe von** t, definiert durch:

- falls $ct = c$, mit $\alpha(c) = \lambda$, dann gilt $|ct| = 1$
- falls $ct = c(ct_1, \ldots, ct_n)$ mit $\alpha(c) = s_1 \ldots s_n$, dann gilt $|ct| = 1 + |ct_1| + \ldots + |ct_n|$ ⋆

Intuitiv kann man die so definierte Größe von Konstruktortermen auf zweierlei Art deuten: einerseits ist sie gleich der Anzahl der Konstruktoren, andererseits gleich der Anzahl der Unterterme.

Definition 2.2.15 (auftretende Variablen)
Für Terme $t \in T_\Sigma$ ist $\boldsymbol{Var}(t)$ die Menge der darin **auftretenden Variablen**. Diese ist definiert durch:

- $Var(x) = \{x\}$, für Variablen $x \in V_\Sigma$,
- $Var(c) = Var(a) = \emptyset$, für Basiskonstruktoren $c \in \overline{\mathcal{C}}$ und Konstanten $a \in \overline{\mathcal{F}}$ ($\alpha(c) = \alpha(a) = \lambda$),
- $Var(l(t_1, \ldots, t_n)) = Var(t_1) \cup \ldots \cup Var(t_n)$ für Terme $l(t_1, \ldots, t_n) \in T_\Sigma$.

⋆

Definition 2.2.16 (Grundterme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $S = \{s_1, \ldots, s_n\}$.
Für $s_i \in S$ ist $\boldsymbol{T^0_{s_i}}$ die Menge der **Grundterme der Sorte $\boldsymbol{s_i}$**, definiert durch:

$$t \in T^0_{s_i} \quad :\Longleftrightarrow \quad t \in T_{s_i} \text{ und } Var(t) = \emptyset$$

$\boldsymbol{T^0_\Sigma}$, die Menge der **$\boldsymbol{\Sigma}$-Grundterme**, ist definiert durch $T^0_\Sigma = T^0_{s_1} \cup \ldots \cup T^0_{s_n}$. ⋆

Es gilt $CT_{s_i} \subseteq T^0_{s_i}$ und $CT_\Sigma \subseteq T^0_\Sigma$.

Aus den Definitionen 2.2.9 – 2.2.11 und der Tatsache, daß laut Def. 2.2.1 jeder Konstruktor und jede Funktion nur einer Sorte zugeordnet ist, folgt das

Faktum 2.2.17 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $t \in T_\Sigma$. Dann gilt $t \in T_s$ für genau ein $s \in S$, $t \in FFT_s$ für höchstens ein $s \in S$ und $ct \in CT_s$ für höchstens ein $s \in S$.*

Diese Eigenschaft wird in der Literatur auch als 'strongly typed' bezeichnet [LEW96, Def. 2.30]. Gemeinsam mit der Tatsache, daß aus einer Signatur keine Beziehung zwischen verschiedenen Sorten ablesbar ist, deutet dies auf eine flache (oder, anders gesagt, nicht vorhandene) Sortenhierarchie hin. Dies wird weiter unten durch die semantischen Definitionen bestätigt. Sortenhierarchien (engl. 'ordered sorts', s. z.B. [LEW96, Abschn. 12.3]) sind ein Beispiel für verschiedene Verfeinerungen des Konzeptes abstrakter Datentypen, die in dieser Abhandlung nicht berücksichtigt werden.

Der Vollständigkeit halber definieren wir noch Substitutionen auf Variablen und in Termen, und zwar genau wie üblich.

Definition 2.2.18 (Substitutionen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Eine **Substitution** (zu V_Σ) ist eine Abbildung $\sigma : V_\Sigma \to T_\Sigma$, wobei nur für endlich viele Variablen $\sigma(x) \neq x$ gilt, und wobei σ *wohlsortiert* ist, d.h. $\sigma(x) \in T_s$, falls $x \in V_s$.

Eine Substitution σ auf Variablen läßt sich erweitern zu einer Abbildung auf Termen $\overline{\sigma} : T_\Sigma \to T_\Sigma$, die postfix notiert wird ('$t\overline{\sigma}$') und definiert ist durch:

- $x\overline{\sigma} = \sigma(x)$, falls $x \in V_\Sigma$,
- $c\overline{\sigma} = c$ und $a\overline{\sigma} = a$, für Basiskonstruktoren $c \in \overline{\mathcal{C}}$ und Konstanten $a \in \overline{\mathcal{F}}$ $(\alpha(c) = \alpha(a) = \lambda)$,
- $(l(t_1, \ldots, t_n))\overline{\sigma} = l(t_1\overline{\sigma}, \ldots, t_n\overline{\sigma})$ für Terme $l(t_1, \ldots, t_n) \in T_\Sigma$. ⋆

Notation 2.2.19
Mit $[x_1/t_1, \ldots, x_n/t_n]$ wird eine Substitution σ notiert, für die gilt:

$$\sigma(x) = \begin{cases} t_i, & \text{falls } x = x_i \text{ für ein } i \in \{1, \ldots, n\} \\ x, & \text{sonst} \end{cases}$$

⋆

Wir werden ohne eine Ordnung auf Substitutionen auskommen, stattdessen muß es aber möglich sein, Substitutionen auf bestimmte Variablen einzuschränken.

Definition 2.2.20 (Einschränkung von Substitutionen)
Sei Σ eine ADT-Signatur, σ eine Substitution zu V_Σ und sei $V = \{x_1, \ldots, x_n\} \subseteq V_\Sigma$ eine endliche Menge von Variablen. Dann ist die **Einschränkung von σ auf V**, $\sigma|_V$, definiert durch:

$$\sigma|_V = [x_1/\sigma(x_1), \ldots, x_n/\sigma(x_n)]$$

⋆

2.2.2 Semantik

In diesem Abschnitt werden Funktionen einer Signatur *interpretiert*, und zwar als Abbildungen über dem sortierten Bereich der *Konstruktorterme*. Diese Interpretationen werden auf Termen fortgesetzt, wobei Funktionen und Konstruktoren unterschiedlich behandelt werden. Auf Basis der Interpretationen werden schließlich *frei erzeugte Algebren und Datentypen* definiert.

Den semantischen Definitionen müssen wir zunächst noch eine kleine Bedingung an die Signaturen vorausschicken.

Definition 2.2.21 (genug Konstruktorterme)
Eine ADT-Signatur $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ hat **genug Konstruktorterme**, falls für jede Sorte $s \in S$ die Menge ihrer Konstruktorterme mindestens ein Element enthält, d.h. $CT_s \neq \emptyset$. ⋆

Eine ADT-Signatur hat insbesondere dann genug Konstruktorterme, wenn jede Sorte mindestens einen Basiskonstruktor hat (s. Def. 2.2.3). Diese Forderung ist aber zu stark.

Beispiel 2.2.22 Betrachten wir die Signatur $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ mit:

$$S = \{Bool, BoolPair\}$$
$$\mathcal{C} = \{\{\mathsf{tt}, \mathsf{ff}\}_{Bool}, \{\mathsf{mkpair}\}_{BoolPair}\}$$
$$\mathcal{F} = \{\{\}_{Bool}, \{\}_{BoolPair}\}$$
$$\alpha(\mathsf{tt}) = \alpha(\mathsf{ff}) = \lambda$$
$$\alpha(\mathsf{mkpair}) = [Bool\ Bool]$$

Σ enthält genug Konstruktorterme. Zwar besitzt die Sorte *BoolPair* keinen Basiskonstruktor, aber die Menge ihrer Konstruktorterme enthält mindestens ein Element (z.B. $\mathsf{mkpair}(\mathsf{tt}, \mathsf{tt})$). ⋆

Die folgenden Definitionen sind nur anwendbar auf Signaturen, die genug Konstruktorterme besitzen. Diese Voraussetzung gilt stillschweigend, sowohl hier als auch bei der späteren Verwendung der hier definierten Begriffe.

Definition 2.2.23 ($\mathcal{F}$-Interpretationen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Eine **$\mathcal{F}$-Interpretation** $\mathcal{I}$ weist jeder Funktion f, mit $f \in F_s$ eine Abbildung $\mathcal{I}(f)$ zu mit:

$$\begin{array}{ll} \mathcal{I}(f) : CT_{s_1} \times \ldots \times CT_{s_n} \to CT_s, & \text{falls } \alpha(f) = s_1 \ldots s_n, \\ \mathcal{I}(f) : \qquad\qquad\qquad\qquad \to CT_s, & \text{falls } \alpha(f) = \lambda. \end{array}$$

⋆

Der Name '$\mathcal{F}$'-Interpretation betont die Tatsache, daß lediglich Funktionen ($\in \mathcal{F}$) interpretiert werden, *nicht aber Konstruktoren.* Die semantischen Bereiche, über denen interpretiert wird, sind die Konstruktorterm-Mengen (der passenden Sorten). Die zentrale Schlußfolgerung aus dieser Definition wollen wir besonders hervorheben:

Bemerkung 2.2.24 Nach Def. 2.2.23 sind die semantischen Bereiche, über denen die Funktionen einer Signatur Σ interpretiert werden, schon durch Σ selbst bestimmt, und zwar eindeutig. Es handelt sich dabei um die Konstruktorterm-Mengen CT_s (aller Sorten $s \in S$).

Die Einschränkung auf Signaturen mit genug Konstruktortermen dient demzufolge dem Zweck, nichtleere semantische Bereiche dieser Bauart zu garantieren.

Nun wollen wir die durch eine Interpretation induzierte Auswertung von Termen definieren. Dazu benötigen wir noch die Belegung von Variablen. Auch hier besteht der semantische Bereich der Elemente, mit denen Variablen 'belegt' werden, aus Konstruktortermen.

Definition 2.2.25 (Variablenbelegungen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Eine **Variablenbelegung** (zu V_Σ) ist eine Abbildung $\beta : V_\Sigma \to CT_\Sigma$, so daß, für $s \in S$ und $x \in V_s$, $\beta(x) \in CT_s$. ⋆

Für die späteren semantischen Definitionen braucht man gelegentlich lokal abgeänderte Variablenbelegungen.

Definition 2.2.26 (abgeänderte Variablenbelegungen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $s \in S$, $x \in V_s$, $ct \in CT_s$, und β eine Variablenbelegung (zu V_Σ). Dann ist die **Abänderung der Variablenbelegung** β **an der Stelle** x **durch** ct, kurz 'β_x^{ct}', definiert durch:

$$\beta_x^{ct}(y) = \begin{cases} ct, & \text{falls } y = x \\ \beta(y), & \text{sonst} \end{cases}$$

$\star$

Definition 2.2.27 (Auswertung von Termen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\mathcal{I}$ eine $\mathcal{F}$-Interpretation und β eine Variablenbelegung zu V_Σ.
Dann ist die **Auswertung** $\boldsymbol{val}_{\mathcal{I},\beta} : T_\Sigma \to CT_\Sigma$ **von Termen** definiert durch:

1. $val_{\mathcal{I},\beta}(x) = \beta(x)$, für $x \in V_\Sigma$,
2. $val_{\mathcal{I},\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(val_{\mathcal{I},\beta}(t_1), \ldots, val_{\mathcal{I},\beta}(t_n))$,
   für $f \in \overline{\mathcal{F}}$, $f(t_1, \ldots, t_n) \in T_\Sigma$,
3. $val_{\mathcal{I},\beta}(a) = \mathcal{I}(a)()$, für $a \in \overline{\mathcal{F}}$, $\alpha(a) = \lambda$,
4. $val_{\mathcal{I},\beta}(c(t_1, \ldots, t_n)) = c(val_{\mathcal{I},\beta}(t_1), \ldots, val_{\mathcal{I},\beta}(t_n))$,
   für $c \in \overline{\mathcal{C}}$, $c(t_1, \ldots, t_n) \in T_\Sigma$,
5. $val_{\mathcal{I},\beta}(c) = c$, für $c \in \overline{\mathcal{C}}$, $\alpha(c) = \lambda$. $\star$

Diese Definition ist konsistent mit Def. 2.2.23 und 2.2.25, da die Abbildungen $val_{\mathcal{I},\beta}$, $\mathcal{I}(f)$ und β alle nach CT_s gehen (für passendes s). Da die semantischen Bereiche fest sind, benötigt val nur $\mathcal{I}$ und β als Index. Wir wollen auch β weglassen, falls es nicht benötigt wird.

Definition 2.2.28 (Auswertung ohne Variablenbelegung)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\mathcal{I}$ eine $\mathcal{F}$-Interpretation.
Dann ist $\boldsymbol{val}_{\mathcal{I}} : T_\Sigma^0 \to CT_\Sigma$ (ohne den Index β!) definiert durch $val_{\mathcal{I}}(t) = val_{\mathcal{I},\beta}(t)$, f.a. $t \in T_\Sigma^0$, wobei β eine beliebig gewählte Variablenbelegungen zu V_Σ ist. $\star$

Die Wohldefiniertheit von $val_{\mathcal{I}}$ folgt aus der als 'Koinzidenzlemma' bekannten Eigenschaft von Auswertungen (s. [EFT92, Lemma 4.6]). Das Ergebnis einer Termauswertung mit $val_{\mathcal{I},\beta}$ oder $val_{\mathcal{I}}$ ist immer ein Konstruktorterm (s. insbes. Def. 2.2.23 und 2.2.25).

Aus Def. 2.2.27 folgt unmittelbar das

Faktum 2.2.29 *Sei* $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ *eine ADT-Signatur,* $\mathcal{I}$ *eine* $\mathcal{F}$*-Interpretation und* $ct \in CT_\Sigma$ *ein Konstruktorterm. Dann gilt:*

$$val_{\mathcal{I}}(ct) = ct$$

Das klassische Substitutionstheorem auf Termen (s. [EFT92, Lemma 8.3 (a)]), welches den Zusammenhang herstellt zwischen Substitutionen und Abänderungen von Variablenbelegungen, benötigen wir nur in spezialisierter Form, für Substitution mit, bzw. Abänderung durch, *Konstruktorterme*.

Faktum 2.2.30 *Sei* $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ *eine ADT-Signatur,* $t \in T_\Sigma$*,* $s \in S$*,* $x \in V_s$ $ct \in CT_s$*, und* β *eine Variablenbelegung (zu* V_Σ*). Dann gilt:*

$$val_{\mathcal{I},\beta}(t[x/ct]) = val_{\mathcal{I},\beta_x^{ct}}(t)$$

Es ist dem Faktum 2.2.29 zu verdanken, daß in Faktum 2.2.30 der Term *ct* auf der rechten Seite der Gleichung nicht erst ausgewertet werden muß.

Blicken wir noch einmal zurück auf die Definition 2.2.27. Diese kann man auffassen als Kombination von Standard-Auswertungen (s. z.B. [SA91, Def. 2.4.]) einerseits mit der Auswertung in *Herbrand-Algebren* bzw. *Term-Algebren* andererseits. Die Begriffe 'Herbrand-Algebra' und 'Term-Algebra' bedeuten genau das gleiche, nur daß ersteres ein Terminus der mathematischen Logik ist (s. z.B. [SA91, Def. 2.10.]) und letzteres der entsprechende Terminus aus dem Bereich der abstrakten Datentypen [LEW96, Def. 3.46]. In Herbrand-Algebren werden Funktionen durch sich selbst interpretiert. Ähnliches gilt bei uns für die Konstruktoren, nur daß wir in unseren Definitionen die Konstruktoren zunächst gar nicht interpretieren (mittels $\mathcal{F}$-Interpretationen), sie dann aber bei der Termauswertung einfach 'stehen lassen' (s. Fälle 4 und 5 in Def. 2.2.27). Diese, zugegebenermaßen subtile, Verschiebung der Bedeutungsbildung ermöglicht eine minimalistische Definition frei erzeugter Algebren, für die wir außer einer Signatur dann nur noch die Interpretation der (Nicht-Konstruktor-)Funktionen benötigen.

Definition 2.2.31 (frei erzeugte Algebren)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $\mathcal{I}$ eine $\mathcal{F}$-Interpretation. Dann ist das Paar $(\mathcal{CT}_\Sigma, \mathcal{I})$ eine **frei erzeugte (f.e.) Σ-Algebra**. ⋆

Es ist zu beachten, daß, bei gegebenem Σ, das $\mathcal{CT}_\Sigma$ für alle frei erzeugten Σ-Algebren das gleiche ist, nämlich die Familie $\{CT_s |\ s \in S\}$ der Konstruktorterm-Mengen. Die Definitionen enthalten somit eine gewisse Redundanz, die aber dazu beiträgt, die allgemeinere Begriffsbildung aus der Logik bzw. der algebraischen

Spezifikation auch im speziellen Fall *im wesentlichen bedeutungsgleich* zu verwenden. (Was hier 'im wesentlichen' genau bedeutet, wird in Abschn. 2.6 ausführlich erläutert.) Die eindeutige Beziehung zwischen frei erzeugten Algebren und ihren Interpretationen erlaubt uns, zwischen beiden hin und her zu wechseln.

Notation 2.2.32 Ist $\mathcal{I}$ eine $\mathcal{F}$-Interpretation, dann bezeichnet $\boldsymbol{Alg}(\mathcal{I})$ die frei erzeugte Σ-Algebra $(\mathcal{CT}_\Sigma, \mathcal{I})$. Ist $\mathcal{A} = (\mathcal{CT}_\Sigma, \mathcal{I})$ eine frei erzeugte Σ-Algebra, dann bezeichnet $\boldsymbol{Intpr}(\mathcal{A})$ die $\mathcal{F}$-Interpretation $\mathcal{I}$. ⋆

Notation 2.2.33 $\mathcal{CT}_\Sigma$ nennen wir auch ***die* Grundbereiche der f.e. Σ-Algebren** (Plural, da $\mathcal{CT}_\Sigma$ eine Familie von Mengen ist). CT_Σ $(= \overline{\mathcal{CT}_\Sigma}$, die Menge aller Konstruktorterme) nennen wir auch ***den* Grundbereich der f.e. Σ-Algebren** (Singular). CT_s (d.h. die Menge aller Konstruktorterme der Sorte s) nennen wir auch **Grundbereich der Sorte s** in den f.e. Σ-Algebren. ⋆

Diese Bezeichnungen werden dadurch gerechtfertigt, daß uns die Mengen CT_s intuitiv als *Semantik* der jeweiligen Sorte s dienen. Dazu bedarf es aber rein formal keiner 'Interpretation' der Sorten (etwa mittels '$\mathcal{I}(s) = CT_s$'). Stattdessen sorgen die Definitionen dieses Abschnitts dafür, daß die Konstruktorterm-Mengen überall die Rolle einnehmen, die üblicherweise die (die Sorten interpretierenden) Grundbereiche einnehmen. Hieraus ergibt sich auch, zusammen mit Faktum 2.2.17, daß die Sorten semantisch als disjunkt anzusehen sind. Es gibt weder Hierarchien noch Überlappungen.

Allgemeinere Abhandlungen über abstrakte Datentypen definieren zunächst *allgemeine* Algebren und erst danach die *Erzeugtheit* (genauer: Termerzeugtheit) und das *'frei'* als mögliche Eigenschaften von Algebren. Das ist dann nützlich, wenn man noch andere Algebren betrachtet und mit den erzeugten bzw. freien vergleicht. In einem solchen Duktus wäre die Aussage in Def. 2.2.31 keine Definition, sondern ein Satz! Da wir hingegen die frei erzeugten Algebren gleich passend definiert haben, gibt es an dieser Stelle formal nichts nachzuweisen. Wir können uns aber sofort davon überzeugen, daß die f.e. Σ-Algebren die Eigenschaften besitzen, die mit 'erzeugt' und 'frei' assoziiert werden. Der Grundbereich *(term-)erzeugter* Algebren darf nur solche Elemente enthalten, die man auch mit Termen benennen kann, d.h. jedes Element ist das Ergebnis der Auswertung irgendeines Termes. Dies ist die sog. 'no junk'-Eigenschaft. Im Falle der Auszeichnung von Konstruktoren müssen alle Elemente sogar schon durch Konstruktorterme benennbar sein. Bei *freien* Algebren müssen verschiedene Konstruktorterme zu verschiedenen Elementen ausgewertet werden. Dies ist die 'no confusion'-Eigenschaft. Beide Eigenschaften sind für f.e. Σ-Algebren gegeben, da hier der Grundbereich identisch ist mit der Menge aller Konstruktorterme, und da laut Faktum 2.2.29 jeder Konstruktorterm zu sich selbst ausgewertet wird.

Nun haben wir alle Hilfsmittel beisammen, um zu definieren, was wir mit 'frei erzeugten Datentypen' meinen.

Definition 2.2.34 (frei erzeugte Datentypen)
Ein **frei erzeugter Σ-Datentyp** $\mathcal{DT}$ ist eine Menge[1] frei erzeugter Σ-Algebren. ⋆

Von allen Definitionen dieses Abschnittes ist dies möglicherweise die überraschendste. Abhängig von der Vertrautheit des Lesers oder der Leserin mit dem Gebiet der abstrakten Datentypen könnten verschiedene Erwartungen an die Definition von Datentypen im allgemeinen sowie frei erzeugten Datentypen im besonderen geknüpft sein, die durch die Definition 2.2.34 nicht erfüllt werden.

Zwei Dinge könnten vor allem auffallen:

1. *Verschiedene* Algebren, die insbesondere nicht isomorph zu sein brauchen, können *einem* Datentyp angehören.
2. Es fehlt die Forderung nach einem Abschluß gegen Isomorphie, wonach mit einer Algebra auch alle dazu isomorphen zu einem Datentyp gehören.

Beide Punkte werden in Abschnitt 2.6 noch genauer diskutiert, darum beschränken wir uns hier auf einige Bemerkungen.

Zu 1.: Die Zulassung verschiedener, nicht-isomorpher Algebren erlaubt es, gewisse Dinge in einem Datentyp als 'offen gelassen' anzusehen. Dies ist besonders interessant im Zusammenhang mit *Spezifikationen* abstrakter Datentypen (s. Abschn. 2.3). Von diesen wird nicht mehr gefordert, jedes Detail festzulegen. Dieser Ansatz ist in der jüngeren Literatur über abstrakte Datentypen häufig zu finden. (Referenzen hierzu s. Abschn. 2.6.3).

Zu 2.: In diesem Punkt folgen wir, rein formal, nicht der einschlägigen Literatur. Dies bedeutet aber nicht, daß wir unter frei erzeugten Datentypen wirklich etwas anderes verstehen. Wir werden in Abschnitt 2.6.2 erklären und sogar beweisen, daß die Definition 2.2.34 (in einem noch zu präzisierenden Sinne) gleichwertig ist mit gängigen Definitionen und somit absolut verträglich ist mit der Intuition vorgebildeter Leser oder Leserinnen.

Zum Schluß definieren wir noch den Spezialfall monomorpher Datentypen.

Definition 2.2.35 (monomorphe vs. polymorphe Datentypen)
Ein f.e. Σ-Datentyp $\mathcal{DT}$ heißt **monomorph**, falls $|\mathcal{DT}| = 1$. Nicht monomorphe Datentypen heißen **polymorph**. ⋆

[1] Jede Mengen f.e. Algebren ist tatsächlich eine Menge im Sinne der axiomatischen Mengentheorie (s. z.B. [Eis71, Def. 4.7]), nicht nur eine Klasse. Dies liegt daran, daß der Grundbereich f.e. Algebren immer abzählbar ist und infolgedessen die Kardinalität von Mengen solcher Algebren beschränkt ist. Ohne jegliche Einschränkung möglicher Grundbereiche wäre das nicht der Fall, s. [LEW96, Exercise 2.3-1].

2.3 Spezifikation frei erzeugter Datentypen

In dieser Abhandlung geht es um eine Methodik, die für einen gegebenen abstrakten Datentyp und eine gegebene (evtl. fälschlicherweise vermutete) Eigenschaft aufdecken soll, ob der Datentyp die Eigenschaft *nicht* besitzt. Hierzu muß die Methodik Zugriff haben auf eine Darstellung des Datentyps, auf eine syntaktische Repräsentation, die ihn *spezifiziert*. Zur Spezifikation von abstrakten Datentpen verwendet man gemeinhin *Formeln* einer Logiksprache. Die Menge von Formeln, die einen Datentyp spezifizieren, nennt man *Axiome* des Datentyps.

Das Gebiet der Charakterisierung abstrakter Datentypen durch Formeln bzw. Formelmengen heißt *algebraische Spezifikation*. Dieses Gebiet fächert sich auf in verschiedene algebraische Spezifikationsstile, die sich unterscheiden in der zugelassenen Formelsyntax und in den semantischen Grundannahmen. Eine gute und moderne Einführung in die unterschiedlichen Facetten algebraischer Spezifikation gibt das Buch von Loeckx, Ehrich & Wolf [LEW96], das sich auch gut als Nachschlagewerk eignet.

Allen Ansätzen ist gemein, daß die Formelsprache auf *Gleichungen* basiert. Man unterscheidet die folgenden Varianten:

- *reine Gleichungslogik*: Alle Axiome sind Gleichungen, es gibt keine Disjunktionen (Oder-Verknüpfungen) oder Negationen (Ungleichungen).
- *Horn Gleichungslogik*: Axiome sind Disjunktionen zwischen beliebig vielen Ungleichungen und höchstens einer Gleichung. Man kann solche Formeln auch auffassen als Implikationen zwischen a) einer Konjunktion von Gleichungen und b) einer einzelnen Gleichung.
- *quantorenfreie Gleichungslogik*: Es sind beliebige Konjunktionen, Disjunktionen, Implikationen und Negationen zugelassen. Wie bei den ersten beiden Logiken geschieht die Allquantifikation implizit.
- *volle erststufige Gleichungslogik*: Hier sind nun alle üblichen logischen Junktoren sowie All- und (dies ist der wesentliche Unterschied) Existenzquantoren zugelassen.

In dieser Aufzählung ist jede Logik ein Spezialfall der nachfolgenden. Bereits mit quantorenfreier Gleichungslogik kann man alle berechenbaren Datentypen (das sind intuitiv die implementierbaren) spezifizieren, solange die Termerzeugtheit vorausgesetzt wird und Hilfsfunktionen zugelassen sind. Unter der Zusatzannahme *initialer* Semantik (s. Abschn. 2.6.3) lassen sich monomorphe Datentypen sogar schon mittels reiner Gleichungen spezifizieren. (Diese Aussagen sind zusammengefaßt in [Sch98, Satz 1.1], für die Einzelresultate s. [BT82] und [GH78, Theorem 1].)

Wir lassen in dieser Abhandlung zur Spezifikation von frei erzeugten Datentypen *volle erststufige Gleichungslogik* zu. Im technischen Sinne wird auch die quantorenfreie Gleichungslogik eine große Rolle spielen, aber nicht als Einschränkung auf der Spezifikationsseite. Warum eine stärkere Einschränkung für unsere Zwecke auch keinen großen Nutzen brächte, *obwohl* die meisten Beispiele für Spezifikationen aus Horn oder sogar reinen Gleichungsformeln bestehen, das wird in Abschnitt 2.6.3 erläutert.

2.3.1 Syntax

Wir definieren jetzt die Formeln, die zur Spezifikation eines frei erzeugten Datentyps verwendet werden können. Auch die Aussagen über den Datentyp, deren (Un-)Gültigkeit untersucht werden soll, sind solche Formeln. Der Spezialfall der 'Klauseln' wird gleich mitdefiniert.

Definition 2.3.1 (Literale, Klauseln, Formeln)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur.

- Lit_Σ ist die minimale Menge, für die gilt: falls $\{t, t'\} \subseteq T_s$ für ein $s \in S$, dann $\{t \doteq t', t \not\doteq t'\} \subseteq Lit_\Sigma$. $\boldsymbol{Lit_\Sigma}$ ist die Menge der **Σ-Literale**.
- Kl_Σ ist die minimale Menge, für die gilt: falls $\{lit_1, \ldots, lit_n\} \subseteq Lit_\Sigma$, dann $lit_1 \vee \ldots \vee lit_n \in Kl_\Sigma$. $\boldsymbol{Kl_\Sigma}$ ist die Menge der **Σ-Klauseln**.
- For_Σ ist die minimale Menge, für die gilt:
  - $Lit_\Sigma \subseteq For_\Sigma$,
  - falls $\varphi \in For_\Sigma$, dann $\neg\varphi \in For_\Sigma$,
  - falls $\{\varphi_1, \ldots, \varphi_n\} \subseteq For_\Sigma$, dann $\{\varphi_1 \wedge \ldots \wedge \varphi_n,\ \varphi_1 \vee \ldots \vee \varphi_n\} \subseteq For_\Sigma$,
  - falls $\varphi_1 \in For_\Sigma$ und $\varphi_2 \in For_\Sigma$, dann $\{\varphi_1 \rightarrow \varphi_2,\ \varphi_1 \leftrightarrow \varphi_2\} \subseteq For_\Sigma$,
  - falls $x \in V_\Sigma$ und $\varphi \in For_\Sigma$, dann $\{\forall x.\varphi,\ \exists x.\varphi\} \subseteq For_\Sigma$.

  $\boldsymbol{For_\Sigma}$ ist die Menge der **Σ-Formeln**. ⋆

Die Disjunktion von Formeln ist nur deswegen n-stellig definiert (und der Symetrie wegen auch die Konjunktion), damit die Σ-Klauseln eine Teilmenge der Σ-Formeln sind.

(Der Leser oder die Leserin wird bemerkt haben, daß in dieser Definition die Negation doppelt vorkommt, einmal in Form der Ungleichung '$\not\doteq$' und einmal in Form der Negation '$\neg$' allgemeiner Formeln. Diese kleine Redundanz hat kaum Nachteile, lediglich die Normalisierung von Formeln in Abschn. 2.5 erfordert zwei zusätzliche Fälle. Auf

der anderen Seite spielt im weiteren Verlauf die Negation *allgemeiner Formeln* keine große Rolle, und auf der *Ebene der Literale* werden Gleichungen und Ungleichungen in gewisser Weise gleichrangig behandelt, wie wir noch sehen werden. Dann wird es von Vorteil sein, Ungleichungen als elementar anzusehen.)

Faktum 2.3.2 *Seien Σ und Σ' zwei ADT-Signaturen mit $\Sigma \subseteq \Sigma'$.*
Dann gilt $For_\Sigma \subseteq For_{\Sigma'}$ (aufgrund von Faktum 2.2.12).

Definition 2.3.3 ((frei) auftretende Variablen)
Wir erweitern die Abbildung *Var* der **auftretenden Variablen** (s. Def. 2.2.15) auf Literale und Klauseln:

- $Var(t \doteq t') = Var(t \not\doteq t') = Var(t) \cup Var(t')$,
- $Var(lit_1 \vee \ldots \vee lit_n) = Var(lit_1) \cup \ldots \cup Var(lit_n)$, falls $lit_1 \vee \ldots \vee lit_n \in Kl_\Sigma$

Auf allgemeinen Formeln $\varphi \in For_\Sigma$ definieren wir ***frei***(φ) als Menge der darin **frei auftretenden Variablen**.

- $frei(lit) = Var(lit)$, falls $lit \in Lit_\Sigma$,
- $frei(\neg\varphi) = frei(\varphi)$,
- $frei(\varphi_1 \wedge \ldots \wedge \varphi_n) = frei(\varphi_1 \vee \ldots \vee \varphi_n) = frei(\varphi_1) \cup \ldots \cup frei(\varphi_n)$,
- $frei(\varphi_1 \rightarrow \varphi_2) = frei(\varphi_1 \leftrightarrow \varphi_2) = frei(\varphi_1) \cup frei(\varphi_2)$,
- $frei(\forall x.\varphi) = frei(\exists x.\varphi) = frei(\varphi) \setminus \{x\}$. ⋆

Definition 2.3.4 (Allabschluß, Existenzabschluß)
Sei $\varphi \in For_\Sigma$.
Dann ist $Cl_\forall(\varphi)$, der **Allabschluß von** φ, definiert durch:

$$Cl_\forall(\varphi) = \begin{cases} \forall x_1.\ldots\forall x_n.\varphi, & \text{falls } frei(\varphi) = \{x_1,\ldots,x_n\} \\ \varphi, & \text{falls } frei(\varphi) = \emptyset \end{cases}$$

Außerdem ist $Cl_\exists(\varphi)$, der **Existenzabschluß von** φ, definiert durch:

$$Cl_\exists(\varphi) = \begin{cases} \exists x_1.\ldots\exists x_n.\varphi, & \text{falls } frei(\varphi) = \{x_1,\ldots,x_n\} \\ \varphi, & \text{falls } frei(\varphi) = \emptyset \end{cases}$$

⋆

(Daß die Reihenfolge der Variablen in '$\forall x_1.\dots\forall x_n$' (bzw. '$\exists x_1.\dots\exists x_n$') durch die Definition nicht festgelegt wird, ist gerechtfertigt durch die Invarianz der Formelauswertung gegenüber der Vertauschung unmittelbar aufeinander folgender $\forall$-Quantoren (bzw. unmittelbar aufeinander folgender $\exists$-Quantoren), die sich leicht aus der noch folgenden Def. 2.3.14 ergibt.)

Später werden Literale und Klauseln, die keine Variablen enthalten, eine wichtige Rolle spielen.

Definition 2.3.5 (geschlossene Formeln, Grundklauseln, Grundliterale)
Sei Σ eine ADT-Signatur.
$\boldsymbol{For^0_\Sigma}$, die Menge der **geschlossenen $\boldsymbol{\Sigma}$-Formeln**, ist definiert durch:

$$For^0_\Sigma = \{\varphi \in For_\Sigma \mid frei(\varphi) = \emptyset\}$$

$\boldsymbol{Kl^0_\Sigma}$ und $\boldsymbol{Lit^0_\Sigma}$, die Menge der **$\boldsymbol{\Sigma}$-Grundklauseln** und der **$\boldsymbol{\Sigma}$-Grundliterale**, sind definiert durch: $Kl^0_\Sigma = Kl_\Sigma \cap For^0_\Sigma$ und $Lit^0_\Sigma = Lit_\Sigma \cap For^0_\Sigma$ ⋆

Jetzt erweitern wir die bisher auf Termen definierte Substitution auf Formeln.[2]

Definition 2.3.6 (Substitutionen in Formeln)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und σ eine Substitution (zu V_Σ). Die in Def. 2.2.18 definierte Erweiterung $\overline{\sigma}$ von σ auf Terme wird nun fortgesetzt zu einer **Substitution in Formeln** $\overline{\sigma} : For_\Sigma \to For_\Sigma$, definiert durch:

- $(t \doteq t')\overline{\sigma} = t\overline{\sigma} \doteq t'\overline{\sigma}$,
  $(t \not\doteq t')\overline{\sigma}$ ensprechend,
- $(\varphi_1 \wedge \dots \wedge \varphi_n)\overline{\sigma} = \varphi_1\overline{\sigma} \wedge \dots \wedge \varphi_n\overline{\sigma}$,
  $(\varphi_1 \vee \dots \vee \varphi_n)\overline{\sigma}$, $(\varphi_1 \to \varphi_2)\overline{\sigma}$, $(\varphi_1 \leftrightarrow \varphi_2)\overline{\sigma}$ und $(\neg\varphi)\overline{\sigma}$ ensprechend,
- $(\forall x.\varphi)[x_1/t_1, \dots, x_n/t_n] = \forall x.(\varphi[x_1/t_1, \dots, x_n/t_n])$, falls $x \notin \{x_1, \dots, x_n\}$,
- $(\forall x.\varphi)[x_1/t_1, \dots, x/t, \dots, x_n/t_n] = \forall x.(\varphi[x_1/t_1, \dots, x/x, \dots, x_n/t_n])$,
- $(\exists x.\varphi)[x_1/t_1, \dots, x_n/t_n] = \exists x.(\varphi[x_1/t_1, \dots, x_n/t_n])$, falls $x \notin \{x_1, \dots, x_n\}$.
- $(\exists x.\varphi)[x_1/t_1, \dots, x/t, \dots, x_n/t_n] = \exists x.(\varphi[x_1/t_1, \dots, x/x, \dots, x_n/t_n])$.

⋆

Die Spezifikationen, mit denen Datentypen charakterisiert werden sollen, bestehen schlicht aus einer Signatur und einer Menge von Formeln.

[2] Oftmals wird von Substitutionen in quantifizierte Formeln gefordert, daß keine Variable in den Wirkungsbereich eines entsprechenden Quantors gerät, hier $x \notin Var(t_i)$. Dies war hier nicht nötig, weil die folgenden Verwendungen der Substitution dies von sich aus garantieren.

Definition 2.3.7 (ADT-Spezifikationen, Axiome)
Eine **ADT-Spezifikation** ist ein Paar[3] $\langle \boldsymbol{\Sigma}, \boldsymbol{AX} \rangle$, wobei Σ eine ADT-Signatur ist und $AX \subseteq For_\Sigma$ eine endliche Menge von Σ-Formeln. $\boldsymbol{AX}$ heißt Menge der Axiome, oder kurz, **die Axiome** (Plural) der Spezifikation. $\star$

Notation 2.3.8 Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation.
Dann bezeichnet $Ax(\langle \Sigma, AX \rangle)$ die Menge der Axiome AX. $\star$

Die Axiome einer Spezifikation brauchen nicht geschlossen zu sein, d.h. sie können freie Variablen enthalten. Es folgen zwei Beispiele für ADT-Spezifikationen, auf die wir noch mehrmals verweisen werden.

Beispiel 2.3.9 Wir spezifizieren die natürlichen Zahlen mit Vorgängerfunktion: $\mathsf{Nat} = \langle \Sigma, AX \rangle$, wobei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ die schon erwähnte Signatur der natürlichen Zahlen ist:

$$\begin{gathered} S = \{Nat\} \\ \mathcal{C} = \{\{\mathsf{zero}, \mathsf{succ}\}_{Nat}\} \\ \mathcal{F} = \{\{pred\}_{Nat}\} \\ \alpha(\mathsf{zero}) = \lambda \\ \alpha(\mathsf{succ}) = \alpha(pred) = Nat \end{gathered}$$

Es gibt nur ein Axiom:

$$AX = \{ \quad pred(\mathsf{succ}(x)) \doteq x \quad \}$$

$\star$

Das zweite Beispiel ist eine Erweiterung des ersten, eine Spezifikation von Stacks natürlicher Zahlen. Der Übersichtlichkeit halber wiederholen wir die entprechende Signatur aus Beispiel 2.2.4.

Beispiel 2.3.10 Wir spezifizieren den Stack natürlicher Zahlen:
$\mathsf{NatStack} = \langle \Sigma, AX \rangle$, $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ mit:

$$\begin{gathered} S = \{Nat, Stack\} \\ \mathcal{C} = \{\{\mathsf{zero}, \mathsf{succ}\}_{Nat}, \{\mathsf{nil}, \mathsf{push}\}_{Stack}\} \\ \mathcal{F} = \{\{pred, top\}_{Nat}, \{pop, del\}_{Stack}\} \\ \alpha(\mathsf{zero}) = \alpha(\mathsf{nil}) = \lambda \\ \alpha(\mathsf{succ}) = \alpha(pred) = Nat \\ \alpha(\mathsf{push}) = \alpha(del) = [Nat\ Stack] \\ \alpha(top) = \alpha(pop) = Stack \end{gathered}$$

Die Axiome sind:

[3] Der Leser oder die Leserin möge entschuldigen, daß wir verschiedene Paare verschieden notieren, Strukturen mit runden und Spezifikationen mit spitzen Klammern. Dies dient der Lesbarkeit.

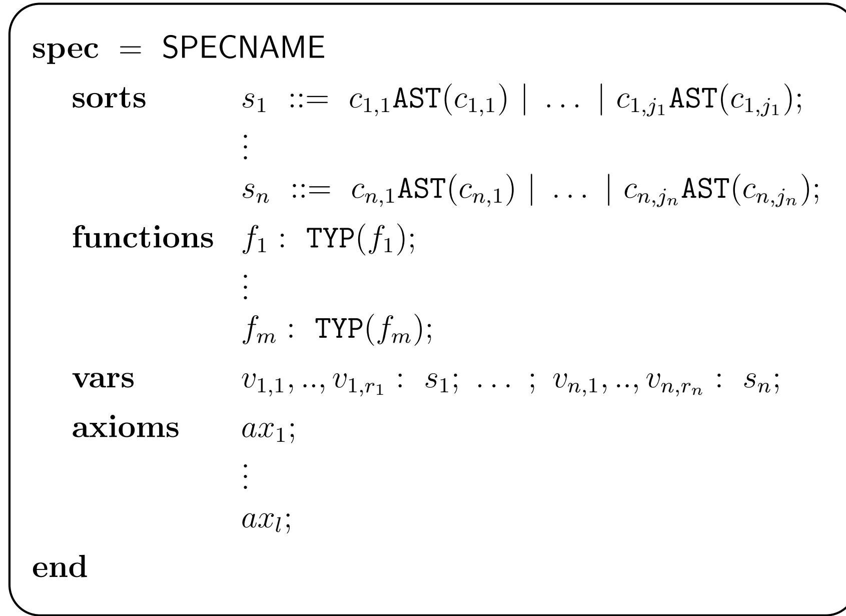


Abbildung 2.1: Schema für die konkrete Synatx von Spezifikationen

$$
\begin{aligned}
AX = \{ \quad & pred(\mathsf{succ}(n)) \doteq n, \\
& top(\mathsf{push}(n, st)) \doteq n, \\
& pop(\mathsf{push}(n, st)) \doteq st, \\
& del(n, \mathsf{push}(n, st)) \doteq st, \\
& n \not\doteq n' \rightarrow \\
& \quad del(n, \mathsf{push}(n', st)) \doteq \mathsf{push}(n', del(n, st)), \\
& del(n, \mathsf{nil}) \doteq \mathsf{nil} \quad \}
\end{aligned}
$$

⋆

Wie schon gesagt, die in den Beispielen verwendete Schreibweise der Signaturen ist nicht sehr übersichtlich. Zu Zwecken der Darstellung sowie zur einfacheren Eingabe werden Spezifikationen anders notiert, üblicherweise in einer Programmiersprachen ähnlichen Syntax, die wir *konkrete* Syntax nennen. Die meisten sog. algebraischen Spezifikationssprachen, die zur Spezifikation von abstrakten Datentypen verwendet werden, sind sich in in der Notationsweise sehr ähnlich, jedenfalls was den Kern der Standardkonstrukte betrifft. Wir verwenden hier eine konkrete Syntax, die sehr eng angelehnt ist an den neueren Sprachstandard 'CASL' (The CoFI[4] Algebraic Specification Language) [CoF98].

Notation 2.3.11 (konkrete Syntax von Spezifikationen)
Sei SPECNAME $= \langle \Sigma, AX \rangle$ eine ADT-Spezifikation mit $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$, wobei $S = \{s_1, \ldots, s_n\}$. Für jede Sorte s_i ist $C_{s_i} = \{c_{i,1}, \ldots c_{i,j_i}\}$ die Menge ihrer

[4]The Common Framework Initiative

```
spec = Nat
    sorts      Nat ::= zero | succ(Nat);
    functions  pred : Nat → Nat;
    vars       x : Nat;
    axioms     pred(succ(x)) ≐ x;
end
```

Abbildung 2.2: Spezifikation Nat

Konstruktoren ($j_i = |C_{s_i}|$). Für jeden Konstruktor $c \in \overline{\mathcal{C}}$ ist sein (konkret syntaktischer) Argumentsorten-Tupel, $\mathtt{AST}(c)$, definiert durch:

$$\mathtt{AST}(c) = \begin{cases} (s_{i_1}; \ldots; s_{i_k}), \text{ falls } \alpha(c) = [s_{i_1} \ldots s_{i_k}] \\ \lambda, \text{ falls } \alpha(c) = \lambda \end{cases}$$

Sei $\overline{\mathcal{F}} = \{f_1, \ldots, f_m\}$ die Menge aller Funktionen der Signatur. Für jede Funktion $f \in F_s$ ist die (konkret syntaktische) Typisierung, $\mathtt{TYP}(f)$, definiert durch[5]:

$$\mathtt{TYP}(f) = \begin{cases} s_{i_1} \times \ldots \times s_{i_k} \to s, \text{ falls } \alpha(c) = [s_{i_1} \ldots s_{i_k}] \\ s, \text{ falls } \alpha(c) = \lambda \end{cases}$$

Die Menge der Axiome sei $AX = \{ax_1, \ldots, ax_l\}$. Die Menge der in den Axiomen auftretenden Variablen einer bestimmten Sorte s_i sei $\{v_{i,1}, \ldots, v_{i,r_i}\} \subseteq V_{s_i}$.

Dann ist die **konkrete Syntax** der Spezifikation SPECNAME schematisch gegeben durch den Text in Abb. 2.1. ⋆

Angewendet auf die beiden obigen Beispielsignaturen ergeben sich die folgenden Spezifikationstexte.

Beispiel 2.3.12 Sei Nat die ADT-Spezifikation aus Beispiel 2.3.9. Ihre konkrete Syntax findet sich in Abb. 2.2. ⋆

Beispiel 2.3.13 Sei NatStack die ADT-Spezifikation aus Beispiel 2.3.10. Ihre konkrete Syntax findet sich in Abb. 2.3. ⋆

(Die Abweichungen dieser Syntax von CASL sind minimal. Um eine syntaktisch korrekte CASL-Spezifikation zu erhalten, die semantisch unseren frei erzeugten Datentypen entspricht, muß man nur die Schlüsselworte '**sorts**' bzw. '**functions**' ersetzen durch

[5]Das Zeichen '→' in der Typisierung ist zu unterscheiden von der Implikation, die durch das gleiche Zeichen dargestellt wird, aber nur in Formeln vorkommt.

spec = NatStack

sorts Nat ::= zero | succ(Nat);
$Stack$ ::= nil | push(Nat; $Stack$);

functions $pred: Nat \to Nat$;
$top: Stack \to Nat$;
$pop: Stack \to Stack$;
$del: Nat \times Stack \to Stack$;

vars $n, n' : Nat$; $st : Stack$;

axioms $pred(\mathsf{succ}(n)) \doteq n$;
$top(\mathsf{push}(n, st)) \doteq n$;
$pop(\mathsf{push}(n, st)) \doteq st$;
$del(n, \mathsf{push}(n, st)) \doteq st$;
$n \not\doteq n' \to$
$del(n, \mathsf{push}(n', st)) \doteq \mathsf{push}(n', del(n, st))$;
$del(n, \mathsf{nil}) \doteq \mathsf{nil}$;

end

Abbildung 2.3: Spezifikation NatStack

'**free types**' bzw. '**ops**' und außerdem in den Axiomen die CASL-Syntax beachten, die ein paar subtile Unterschiede zu unserer Formelsyntax aufweist. Das Ergebnis wäre eine gültige CASL-Spezifikation, aber ohne Ausnutzung zusätzlicher Konzepte, die CASL anbietet, wie Generizität, Selektoren, Infix-Notation oder Strukturierung.)

2.3.2 Semantik

Eine ADT-Spezifikation soll gerade diejenigen (frei erzeugten) Algebren charakterisieren, in denen die Axiome der Spezifikation *gelten*. Hierfür benötigt man die Auswertung von Formeln, abhängig von Interpretationen und Variablenbelegungen.

Definition 2.3.14 (Auswertung von Formeln)
Sei Σ eine ADT-Signatur, $\mathcal{I}$ eine $\mathcal{F}$-Interpretation und β eine Variablenbelegunge zu V_Σ.
Dann ist die **Auswertung** $\boldsymbol{val_{\mathcal{I},\beta}} : For_\Sigma \to \{W, F\}$ **von Formeln** definiert aufbauend auf der Auswertung $val_{\mathcal{I},\beta}$ von Termen (Def. 2.2.27):

- $val_{\mathcal{I},\beta}(t \doteq t') = \begin{cases} W, & \text{falls } val_{\mathcal{I},\beta}(t) = val_{\mathcal{I},\beta}(t') \\ F, & \text{sonst} \end{cases}$

- $val_{\mathcal{I},\beta}(t \dot{\neq} t') = \begin{cases} W, & \text{falls } val_{\mathcal{I},\beta}(t) \neq val_{\mathcal{I},\beta}(t') \\ F, & \text{sonst} \end{cases}$

- $val_{\mathcal{I},\beta}(\neg\varphi) = \begin{cases} W, & \text{falls } val_{\mathcal{I},\beta}(\varphi) = F \\ F, & \text{sonst} \end{cases}$

- $val_{\mathcal{I},\beta}(\varphi_1 \wedge \ldots \wedge \varphi_n) = \begin{cases} W, & \text{falls } val_{\mathcal{I},\beta}(\varphi_i) = W \text{ für alle } i \in \{1, \ldots, n\} \\ F, & \text{sonst} \end{cases}$

- $val_{\mathcal{I},\beta}(\varphi_1 \vee \ldots \vee \varphi_n) = \begin{cases} W, & \text{falls } val_{\mathcal{I},\beta}(\varphi_i) = W \text{ für ein } i \in \{1, \ldots, n\} \\ F, & \text{sonst} \end{cases}$

- $val_{\mathcal{I},\beta}(\varphi_1 \rightarrow \varphi_2) = val_{\mathcal{I},\beta}(\neg\varphi_1 \vee \varphi_2)$

- $val_{\mathcal{I},\beta}(\varphi_1 \leftrightarrow \varphi_2) = val_{\mathcal{I},\beta}((\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1))$

- $val_{\mathcal{I},\beta}(\forall x.\varphi) = \begin{cases} W, & \left[\begin{array}{l} \text{falls } x \in V_s \text{ und für alle Konstruktorterme} \\ ct \in CT_s \text{ gilt: } val_{\mathcal{I},\beta_x^{ct}}(\varphi) = W \end{array} \right. \\ F, & \text{sonst} \end{cases}$

- $val_{\mathcal{I},\beta}(\exists x.\varphi) = \begin{cases} W, & \left[\begin{array}{l} \text{falls } x \in V_s \text{ und für einen Konstruktorterm} \\ ct \in CT_s \text{ gilt: } val_{\mathcal{I},\beta_x^{ct}}(\varphi) = W \end{array} \right. \\ F, & \text{sonst} \end{cases}$

⋆

Da der semantische Bereich bei uns aus Termen besteht, könnte man die Auswertung von all- bzw. existenzquantifizierten Formeln auch mit Hilfe von Substitutionen definieren. Wir lassen die Definition klassisch, formulieren aber das folgende Faktum, das eine spezialisierte Version des klassischen Substitutionstheorems auf Formeln (s. [EFT92, Lemma 8.3 (b)]) darstellt.

Faktum 2.3.15 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\varphi \in For_\Sigma$, $s \in S$, $x \in V_s$ $ct \in CT_s$, und β eine Variablenbelegung (zu V_Σ). Dann gilt:*

$$val_{\mathcal{I},\beta}(\varphi[x/ct]) = val_{\mathcal{I},\beta_x^{ct}}(\varphi)$$

Wie schon bei der entsprechenden Eigenschaft von Termen (Faktum 2.2.30) ist auch hier auf der rechten Seite der Gleichung keine Auswertung von *ct* nötig, aufgrund von Faktum 2.2.29.

Damit können wir nun die Auswertung von all- bzw. existenzquantifizierten Formeln zurückführen auf die Substitution der quantifizierten Variable.

Satz 2.3.16 *Sei Σ eine ADT-Signatur, $\mathcal{I}$ eine $\mathcal{F}$-Interpretation und β eine Variablenbelegung zu V_Σ. Dann gilt:*

- $val_{\mathcal{I},\beta}(\forall x.\varphi) = \begin{cases} W, & \left[\begin{array}{l}\text{falls } x \in V_s \text{ und für alle Konstruktorterme} \\ ct \in CT_s \text{ gilt: } val_{\mathcal{I},\beta}(\varphi[x/ct]) = W\end{array}\right. \\ F, & \text{sonst}\end{cases}$
- $val_{\mathcal{I},\beta}(\exists x.\varphi) = \begin{cases} W, & \left[\begin{array}{l}\text{falls } x \in V_s \text{ und für einen Konstruktorterm} \\ ct \in CT_s \text{ gilt: } val_{\mathcal{I},\beta}(\varphi[x/ct]) = W\end{array}\right. \\ F, & \text{sonst}\end{cases}$

Beweis: Einfaches Einsetzen von Faktum 2.3.15 in Def. 2.3.14. □

Man beachte im Vergleich der Definitionen 2.2.25 und 2.2.18, daß Variablenbelegungen *fast*[6] spezielle Substitutionen sind.

Auf Basis der Formelauswertung können wir nun die Gültigkeit von Formeln in Algebren definieren.

Definition 2.3.17 (Gültigkeit in Algebren)
Sei Σ eine ADT-Signatur, $\mathcal{A} = (\mathcal{CT}_\Sigma, \mathcal{I})$ eine frei erzeugte Σ-Algebra und $\varphi \in For_\Sigma$ eine Formel. Dann ist 'φ **gilt in** $\mathcal{A}$', kurz '$\mathcal{A} \models \varphi$', definiert durch:

$$\mathcal{A} \models \varphi \quad :\Longleftrightarrow \quad \text{für alle Variablenbelegungen } \beta \text{ (zu } V_\Sigma\text{) gilt } val_{\mathcal{I},\beta}(\varphi) = W$$

Die Relation $\models$ läßt sich rechtsseitig erweitern auf Mengen von Formeln. Sei $\Phi \subseteq For_\Sigma$, dann ist '$\Phi$ **gilt in** $\mathcal{A}$', kurz '$\mathcal{A} \models \Phi$', definiert durch:

$$\mathcal{A} \models \Phi \quad :\Longleftrightarrow \quad \text{für alle } \varphi \in \Phi \text{ gilt } \mathcal{A} \models \varphi.$$

⋆

Man beachte, daß in dieser Definition der Gültigkeit die freien Variablen einer Formel implizit, d.h. semantisch, behandelt werden wie allquantifizierte Variablen.

Faktum 2.3.18 *Sei Σ eine ADT-Signatur, $\mathcal{A}$ eine frei erzeugte Σ-Algebra und $\varphi \in For_\Sigma$. Dann gilt:*

$$\mathcal{A} \models \varphi \quad \Longleftrightarrow \quad \mathcal{A} \models Cl_\forall(\varphi)$$

Die Möglichkeit der impliziten Allquantifikation erleichtert sowohl das Spezifizieren als auch die Formulierung von Vermutungen über eine Spezifikation. Durch sie werden explizite Quantoren nur selten benötigt. Ein Nachteil hiervon ist, daß das intuitive 'Gegenteil' einer mit einer Formel getroffenen Aussage nicht ganz genau mit der Negation der Formel zusammenfällt, s. Abschn. 2.4.

[6] Eine Variablenbelegung ist deswegen i.a. keine Substitution, weil nach Def. 2.2.25 $\beta(x) \neq x$ gilt, f.a. Variablen x (da $x \notin CT_\Sigma$). Dies verletzt bei unendlichem V_Σ die Bedingung '$\sigma(x) \neq x$ für höchstens endlich viele Variablen'.

Bemerkung 2.3.19 $\mathcal{A} \models \varphi$ ist nach Def. 2.3.17 nur dann wohldefiniert, wenn $\mathcal{A}$ eine Σ-Algebra und $\varphi \in \mathit{For}_\Sigma$ ist, *für dasselbe* Σ. Dennoch kann auch für verschiedene Σ und Σ' die Gültigkeit einer Formel $\varphi \in \mathit{For}_\Sigma$ wohldefiniert sein gegenüber einer Σ'-Algebra, und zwar falls $\mathit{For}_\Sigma \subseteq \mathit{For}_{\Sigma'}$. Dies trifft gerade dann zu, wenn $\Sigma \subseteq \Sigma'$ (s. Faktum 2.3.2). Dieses Detail wird wichtig im Zusammenhang mit der sog. Skolemisierung, bei der die Signatur erweitert wird.

Da Algebren und Interpretationen eindeutig aufeinander bezogen sind, kann man auch von der Gültigkeit in Interpretationen sprechen.

Definition 2.3.20 (Gültigkeit in Interpretationen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\mathcal{I}$ eine $\mathcal{F}$-Interpretation und $\varphi \in \mathit{For}_\Sigma$. Dann ist '$\varphi$ **gilt in** $\mathcal{I}$', kurz '$\mathcal{I} \models \varphi$', definiert durch:

$$\mathcal{I} \models \varphi \quad :\Longleftrightarrow \quad \mathit{Alg}(\mathcal{I}) \models \varphi$$

('$\mathcal{I} \models \Phi$' entsprechend). ⋆

Nun kommen wir zu den Modellen von Formeln bzw. Formelmengen. Eigentlich ist die Modellbeziehung nichts anderes als die Umkehrung der Gültigkeitsbeziehung. Da aber beide Begriffe für diese Abhandlung überaus wichtig sind, spendieren wir hierfür eine eigene Definition.

Definition 2.3.21 (Modelle von Formeln)
Sei Σ eine ADT-Signatur, $\mathcal{A}$ eine frei erzeugte Σ-Algebra und $\varphi \in \mathit{For}_\Sigma$ und $\Phi \subseteq \mathit{For}_\Sigma$.

$$\mathcal{A} \text{ ist } \mathbf{\Sigma\text{-Modell von}} \left\{ \begin{array}{c} \varphi \\ \\ \Phi \end{array} \right\} \quad :\Longleftrightarrow \quad \mathcal{A} \models \left\{ \begin{array}{c} \varphi \\ \\ \Phi \end{array} \right.$$

⋆

Wir betrachten also nur solche Modelle, die frei erzeugte Algebren sind. Darum verwenden wir den Terminus 'frei erzeugtes Modell' nur dort, wo es vorteilhaft erscheint, diese Einschränkung besonders hervorzuheben.

Formeln und Formelmengen, die ein Modell besitzen, heißen erfüllbar. In unserem Fall hängt dieser Begriff von Σ ab.

Definition 2.3.22 (Erfüllbarkeit)
Sei Σ eine ADT-Signatur, $\varphi \in \mathit{For}_\Sigma$ und $\Phi \subseteq \mathit{For}_\Sigma$.

φ (bzw. Φ) ist '**Σ-erfüllbar**'
$:\Longleftrightarrow$
φ (bzw. Φ) besitzt ein Σ-Modell,
d.h. es *existiert* eine frei erzeugte Σ-Algebra $\mathcal{A}$ mit $\mathcal{A} \models \varphi$ (bzw. $\mathcal{A} \models \Phi$).

⋆

Die Erfüllbarkeit wird in der Literatur üblicherweise auf zweierlei Arten definiert. Entweder sie fällt, wie hier, zusammen mit der Existenz eines Modells (s. [EFT92, Def. 4.3]). Oder man verlangt weniger, nämlich daß es eine Interpretation $\mathcal{I}$ gibt, so daß die betrachtete Formel φ unter wenigstens einer Belegung β der Variablen wahr wird: $val_{\mathcal{I},\beta}(\varphi) = W$ (s. [SA91, Def. 2.8]). Auch letztere Konvention hat viel für sich. Aber in einem Kontext wie dem unsrigen, in dem die Frage nach der Existenz von (Gegen-)Modellen im Mittelpunkt steht, ist es sehr praktisch, die Eigenschaft von Formeln, Modelle zu besitzen, mit einem Adjektiv ausdrücken zu können.

Nun können wir mit Hilfe der Modelle die Folgerbarkeit zwischen Formeln definieren.

Definition 2.3.23 (Folgerbarkeit aus Formeln)
Sei Σ eine ADT-Signatur, $\Phi \subseteq For_\Sigma$ und $\Psi \subseteq For_\Sigma$.
Dann ist 'Ψ **folgt aus** Φ (im über Σ frei Erzeugten)', kurz '$\Phi \models_\Sigma \Psi$', definiert durch:

$$\Phi \models_\Sigma \Psi \quad :\Longleftrightarrow \quad \text{jedes (f.e.) } \Sigma\text{-Modell von } \Phi \text{ ist ein } \Sigma\text{-Modell von } \Psi$$

'$\Phi \models_\Sigma \varphi$' steht für $\Phi \models_\Sigma \{\varphi\}$, und '$\varphi \models_\Sigma \psi$' steht für $\{\varphi\} \models_\Sigma \{\psi\}$. ⋆

Wenn zwei Formelmengen wechselseitig folgerbar sind, dann besitzen sie die gleichen Σ-Modelle, und heißen äquivalent:

Definition 2.3.24 (Äquivalenz von Formeln)
Sei Σ eine ADT-Signatur, $\Phi \subseteq For_\Sigma$ und $\Psi \subseteq For_\Sigma$.
'Ψ **ist äquivalent zu** Φ (im über Σ frei Erzeugten)', falls $\Phi \models_\Sigma \Psi$ und $\Psi \models_\Sigma \Phi$. ⋆

Ein Spezialfall der Folgerbarkeit ist die Allgemeingültigkeit.

Definition 2.3.25 (Allgemeingültigkeit)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $\varphi \in For_\Sigma$.
Dann ist die **Σ-Allgemeingültigkeit von Formeln** als *einstellige* Relation $\models_\Sigma$ über Formeln definiert durch:

$$\models_\Sigma \varphi \quad :\Longleftrightarrow \quad \emptyset \models_\Sigma \varphi$$

⋆

Eine Σ-allgemeingültige Formel gilt tatsächlich in allen frei erzeugten Σ-Algebren, da jede Algebra Modell der leeren Formelmenge ist.

In den Definitionen 2.3.23 und 2.3.25 ist das Σ im Index von $\models_\Sigma$ absolut notwendig, was bei dem 'Gültigkeits-$\models$' zwischen Algebren und Formeln (s. Def. 2.3.17)

nicht der Fall war. Wenn eine frei erzeugte Algebra gegeben ist, dann ist die Auswertung von Formeln in der Algebra klar bestimmt. Wenn wir aber nur Formeln haben und die Folgerbarkeit zwischen ihnen untersuchen, dann müssen wir zusätzlich wissen, *gegenüber welcher Signatur die Modelle der Formeln frei erzeugt zu sein haben.* Insbesondere hängt die Folgerbarkeit stark davon ab, welche Signatursymbole Konstruktoren sind.

Beispiel 2.3.26 Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur mit
$S = \{Nat, Bool\}$, $\mathcal{C} = \{\{\mathsf{zero}, \mathsf{succ}\}_{Nat}, \{\mathsf{tt}, \mathsf{ff}\}_{Bool}\}$, $\mathcal{F} = \{\{\}_{Nat}, \{p\}_{Bool}\}$,
$\alpha(\mathsf{succ}) = \alpha(p) = Nat$, $\alpha(\mathsf{zero}) = \alpha(\mathsf{tt}) = \alpha(\mathsf{ff}) = \lambda$.
Sei $\Phi = \{p(\mathsf{zero}) \doteq \mathsf{tt},\ p(x) \doteq \mathsf{tt} \rightarrow p(\mathsf{succ}(x)) \doteq \mathsf{tt}\}$. Dann gilt:

1. $\models_\Sigma \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))) \not\doteq \mathsf{succ}(\mathsf{zero})$ (frei)

2. $\Phi \models_\Sigma p(x) \doteq \mathsf{tt}$ (erzeugt)

Zur Abhängigkeit der Aussagen 1 und 2 von Σ lassen sich drei Dinge festhalten:

1. Wenn wir das $\models_\Sigma$ gegen die klassische Folgerung $\models$ zwischen Formeln (vgl. [EFT92, Def. 4.1]) austauschen, in der auch nicht freie und nicht erzeugte Algebren betrachtet werden, dann gelten die Folgerungsbeziehungen 1 und 2 nicht mehr.

2. Selbst bei der Einschränkung auf frei erzeugte Algebren gelten die beiden Aussagen nur dann, wenn wir ausschließlich in Σ frei erzeugte Algebren betrachten. Nehmen wir z.B. ein Σ', welches aus Σ hervorgeht durch die Verschiebung von succ von den Konstruktoren zu den Funktionen. Dann ist die obige Ungleichung nicht Σ'-allgemeingültig, d.h.
$\models_{\Sigma'} \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))) \not\doteq \mathsf{succ}(\mathsf{zero})$ gilt *nicht*.
Stattdessen ist sogar die entsprechende Gleichung Σ'-allgemeingültig, d.h.
$\models_{\Sigma'} \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))) \doteq \mathsf{succ}(\mathsf{zero})$ gilt,
da in Σ' nur ein einziger Konstruktorterm der Sorte Nat existiert ($CT_{Nat} = \{\mathsf{zero}\}$), und somit $val_{\mathcal{I},\beta}(t) = \mathsf{zero}$ für alle Terme $t \in T_{Nat}$ (unabhängig von β).

3. Betrachten wir nun noch ein Σ'' das aus Σ hervorgeht durch Hinzufügen eines weiteren Konstruktors zero' zu C_{Nat}. Dann gilt (im Kontrast zu $\Phi \models_\Sigma p(x) \doteq \mathsf{tt}$):
$\Phi \models_{\Sigma'} p(x) \doteq \mathsf{tt}$ gilt *nicht*,
da eine frei erzeugte Σ'-Algebra $\mathcal{A}$ existiert mit $\mathcal{A} \models \Phi$, aber $\mathcal{A} \not\models p(x) \doteq \mathsf{tt}$.
Dazu wählen wir das $\mathcal{A}$ z.B. so, daß
$Intpr(\mathcal{A})(p)(t) = \mathsf{tt}$ für alle $t \in \{\mathsf{zero}, \mathsf{succ}(\mathsf{zero}), \mathsf{succ}(\mathsf{succ}(\mathsf{zero})), \ldots\}$ und
$Intpr(\mathcal{A})(p)(t) = \mathsf{ff}$ für alle $t \in \{\mathsf{zero}', \mathsf{succ}(\mathsf{zero}'), \mathsf{succ}(\mathsf{succ}(\mathsf{zero}')), \ldots\}$. $\star$

Das Zeichen '$\models$' haben wir nun schon verwendet a) für die Gültigkeitsbeziehung (bzw. die Modellbeziehung) zwischen Algebren und Formeln sowie b) in Σ-indizierter Form für die Folgerbarkeit zwischen Formeln und die Allgemeingültigkeit von Formeln. Die nichtindizierte Variante wird jetzt weiter überladen, um die Gültigkeit von Formeln nicht nur in einzelnen Algebren, sondern auch in Mengen von Algebren, sprich Datentypen, auszudrücken.

Definition 2.3.27 (Gültigkeit in Datentypen)
Sei Σ eine ADT-Signatur, $\mathcal{DT}$ ein frei erzeugter Σ-Datentyp und $\varphi \in For_\Sigma$ eine Formel. Dann ist 'φ **gilt in** $\mathcal{DT}$', kurz '$\mathcal{DT} \models \varphi$', definiert durch:

$$\mathcal{DT} \models \varphi \quad :\Longleftrightarrow \quad \text{für alle } \Sigma\text{-Algebren } \mathcal{A} \in \mathcal{DT} \text{ gilt } \mathcal{A} \models \varphi.$$

⋆

Die allgemeine Erfüllbarkeit aus Def. 2.3.22 läßt sich einschränken auf Datentypen.

Definition 2.3.28 (Erfüllbarkeit in Datentypen)
Sei Σ eine ADT-Signatur, $\mathcal{DT}$ ein frei erzeugter Σ-Datentyp, $\varphi \in For_\Sigma$ und $\Phi \subseteq For_\Sigma$.

$$\begin{gathered}\varphi \text{ (bzw. } \Phi\text{) ist '}\textbf{erfüllbar in } \mathcal{DT}\text{'} \\ :\Longleftrightarrow \\ \mathcal{DT} \text{ enthält ein Modell von } \varphi \text{ (bzw. } \Phi\text{),} \\ \text{d.h. es } \textit{existiert} \text{ ein } \mathcal{A} \in \mathcal{DT} \text{ mit } \mathcal{A} \models \varphi \text{ (bzw. } \mathcal{A} \models \Phi\text{).}\end{gathered}$$

⋆

Ein Datentyp induziert eine Menge von in ihm gültigen Formeln, seine Theorie.

Definition 2.3.29 (Theorie)
Sei Σ eine ADT-Signatur, $\mathcal{DT}$ ein frei erzeugter Σ-Datentyp. Dann ist die **Theorie des Datentyps** $\mathcal{DT}$, kurz '$Th(\mathcal{DT})$', definiert durch:

$$\varphi \in Th(\mathcal{DT}) \quad :\Longleftrightarrow \quad [\ \varphi \in For_\Sigma \text{ und } \mathcal{DT} \models \varphi\ ]$$

⋆

Viel wichtiger noch ist die entgegengesetzte Richtung, die Charakterisierung von Datentypen mit Hilfe von Formelmengen, oder besser gleich mit Hilfe von Spezifikationen. Dazu definieren wir zunächst in naheliegender Weise, was ein 'Modell' einer Spezifikation ist, um dann alle Modelle einer Spezifikation zu dem 'spezifizierten' Datentyp zusammenzufassen.

Definition 2.3.30 (Modelle von Spezifikationen)
Sei $\langle\Sigma, AX\rangle$ eine ADT-Spezifikation. Dann ist jedes Σ-Modell $\mathcal{A}$ von AX ein **Modell der Spezifikation** $\langle\Sigma, AX\rangle$, kurz '$\mathcal{A} \Vdash \langle\Sigma, AX\rangle$'. D.h. es gilt:

$$\mathcal{A} \Vdash \langle\Sigma, AX\rangle \quad :\Longleftrightarrow \quad \mathcal{A} \text{ ist f.e. } \Sigma\text{-Algebra mit } \mathcal{A} \models AX$$

⋆

Wie schon bei '$\models$', erlauben wir links von '$\Vdash$' auch reine Interpretationen.

Definition 2.3.31 (erfüllende Interpretationen)
Sei $\langle\Sigma, AX\rangle$ eine ADT-Spezifikation mit $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$. Eine $\mathcal{F}$-Interpretation $\mathcal{I}$ **erfüllt die Spezifikation** $\langle\Sigma, AX\rangle$, kurz '$\mathcal{I} \Vdash \langle\Sigma, AX\rangle$', falls $Alg(\mathcal{I})$ ein Modell von $\langle\Sigma, AX\rangle$ ist, d.h.:

$$\mathcal{I} \Vdash \langle\Sigma, AX\rangle \quad :\Longleftrightarrow \quad Alg(\mathcal{I}) \Vdash \langle\Sigma, AX\rangle$$

⋆

Das Zeichen '$\Vdash$' ist nicht Standard. Da wir aber in aller Ausführlichkeit über Modelle und erfüllende Interpretation von Spezifikationen sprechen wollen, ist es wünschenswert, eine abkürzende Notation zur Verfügung zu haben. Statt ein weiteres Mal das Zeichen '$\models$' zu bemühen, wurde absichtlich ein sehr ähnliches verwendet, damit '$\models$' trotz aller Überladung auch weiterhin rechtsseitig Formeln und Formelmengen vorbehalten bleibt. Rechts von '$\Vdash$' steht hingegen immer eine Spezifikation.

Definition 2.3.32 (Erfüllbarkeit von Spezifikationen)
Sei $\langle\Sigma, AX\rangle$ eine ADT-Spezifikation mit $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$.
$\langle\Sigma, AX\rangle$ heißt **erfüllbar**, falls eine $\mathcal{F}$-Interpretation $\mathcal{I}$ existiert mit $\mathcal{I} \Vdash \langle\Sigma, AX\rangle$.
⋆

Die Modelle einer Spezifikation bilden einen Datentyp.

Definition 2.3.33 (spezifizierte Datentypen)
Sei $\langle\Sigma, AX\rangle$ eine ADT-Spezifikation.
Der durch $\langle\Sigma, AX\rangle$ **spezifizierte (f.e.) Datentyp**, kurz '$\boldsymbol{Dat}(\langle\Sigma, AX\rangle)$', ist die Menge aller Modelle von $\langle\Sigma, AX\rangle$, d.h.:

$$\mathcal{A} \in \boldsymbol{Dat}(\langle\Sigma, AX\rangle) \quad :\Longleftrightarrow \quad \mathcal{A} \Vdash \langle\Sigma, AX\rangle.$$

$\boldsymbol{Dat}(\langle\Sigma, AX\rangle)$ heißt auch '**die Semantik der Spezifikation** $\langle\Sigma, AX\rangle$'. ⋆

Erst diese Definition rechtfertigt, im nachhinein, für das Gebilde $\langle\Sigma, AX\rangle$ den Begriff 'ADT-Spezifikation' zu verwenden.

Definition 2.3.34 (Äquivalenz von Spezifikationen)
Zwei ADT-Spezifikationen $\langle \Sigma, AX \rangle$ und $\langle \Sigma, AX' \rangle$ heißen **äquivalent**, falls

$$Dat(\langle \Sigma, AX \rangle) = Dat(\langle \Sigma, AX' \rangle)$$

⋆

Nun liegt auch der Begriff der Folgerbarkeit aus einer Spezifikation auf der Hand. Dazu wird das Zeichen '$\models$' einmal mehr überladen.

Definition 2.3.35 (Folgerbarkeit aus Spezifikationen)
Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$.
Dann ist 'φ **folgt aus** $\langle \Sigma, AX \rangle$', kurz '$\langle \Sigma, AX \rangle \models \varphi$', definiert durch:

$$\langle \Sigma, AX \rangle \models \varphi \quad :\Longleftrightarrow \quad Dat(\langle \Sigma, AX \rangle) \models \varphi.$$

⋆

Notation 2.3.36 In einem Kontext, in dem die Folgerbarkeitsbeziehung

$$\langle \Sigma, AX \rangle \models \varphi$$

untersucht wird, nennen wir φ die **Vermutung**. ⋆

(Traditionell nennt man solche 'Vermutungen' im automatischen Beweisen gerne 'Theoreme'. Man spricht z.B. davon, daß man vor der Anwendung etwa der Tableau-Methode das 'Theorem' negieren muß, oder es zuerst auf den Ast holt usw. Das liegt daran, daß Beweisverfahren in der Vergangenheit mit Vorliebe auf solche Vermutungen angewendet wurden, von denen man vorneweg wußte, daß sie Theoreme sind.)

Wir haben die Folgerbarkeit aus Spezifikationen über die Gültigkeit in Datentypen definiert. Wir hätten sie natürlich genauso gut über die Folgerbarkeit zwischen Formeln definieren können. Diesen anderen Zusammenhang formulieren wir als Faktum, das zwar trivial ist, für dessen Nachweis man aber immerhin vier der obigen Definitionen anwenden muß.

Faktum 2.3.37 *Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$. Dann gilt:*

$$\langle \Sigma, AX \rangle \models \varphi \quad \Longleftrightarrow \quad AX \models_\Sigma \varphi.$$

Beispiel 2.3.38 In Abb. 2.4 wird die Spezifikation p_Forever definiert. Ihre Signatur und die Axiome sind bekannt als Σ und Φ aus Beispiel 2.3.26. Aus den Aussagen 1 und 2 jenes Beispiels und aus Faktum 2.3.37 ergeben sich die Aussagen:

1. $\mathsf{p_Forever} \models \mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))) \not\doteq \mathsf{succ}(\mathsf{zero})$

```
spec = p_Forever
    sorts       Nat  ::= zero | succ(Nat);
                Bool ::= tt | ff;
    functions   p : Nat → Bool;
    vars        x : Nat;
    axioms      p(zero) ≐ tt;
                p(x) ≐ tt → p(succ(x)) ≐ tt;
end
```

Abbildung 2.4: Spezifikation p_Forever

2. p_Forever $\models p(x) \doteq \mathsf{tt}$

Hierbei gilt die erste Folgerung aus der Spezifikation schon aufgrund der Signatur, unabhängig von den Axiomen. Die zweite Folgerung hängt hingegen sowohl von den Axiomen als auch von der Signatur ab. $\star$

Eine Formel kann auch auf eine schwächere Weise zu einer Spezifikation passen als unbedingt aus ihr zu folgen. Sie kann in bestimmten Modellen der Spezifikation gelten. Dann nennt man sie konsistent mit der Spezifikation, was intuitiv bedeutet, daß sie der Spezifikation nicht widerspricht.

Definition 2.3.39 (Konsistenz zu Spezifikationen)
Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$.

$$\varphi \text{ ist '}\textbf{konsistent zu } \langle \Sigma, AX \rangle\text{'}$$
$$:\Longleftrightarrow$$
$$Dat(\langle \Sigma, AX \rangle) \text{ enthält ein Modell von } \varphi \text{ (bzw. } \Phi\text{),}$$
$$\text{d.h. es } \textit{existiert} \text{ ein } \mathcal{A} \in \mathcal{DT} \text{ mit } \mathcal{A} \models \varphi \text{ (bzw. } \mathcal{A} \models \Phi\text{).}$$

$\star$

Faktum 2.3.40 *Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$. Dann gilt:*

$$\varphi \textit{ ist konsistent zu } \langle \Sigma, AX \rangle \quad \Longleftrightarrow \quad AX \cup \{\varphi\} \textit{ ist } \Sigma\textit{-erfüllbar.}$$

Entsprechend wird in späteren Kapiteln die Frage nach der Konsistenz durch die Konstruktion von Modellen bearbeitet.

2.4 Nicht-Folgerbarkeit

Da diese Abhandlung sich mit Fehlern in abstrakten Datentypen beschäftigt, interessieren wir uns nicht nur dafür, was *gilt* und *folgt*, sondern besonders auch dafür, was *nicht* gilt bzw. *nicht* folgt. Es sind ja gerade diese 'negativen' Sachverhalte, denen wir auf der Spur sind. Dazu sammeln wir in diesem Abschnitt Begrifflichkeiten, die den in Abschn. 2.3.2 bereitgestellten Begriffen widersprechen, und diskutieren ihre Eigenschaften.

Manchmal werden wir davon sprechen, daß in einer Algebra bzw. in einem Datentyp das *Gegenteil* einer Formel gilt. Wir präzisieren kurz, was das heißt. Das Gegenteil einer durch eine Formel ausgedrückten Aussage wird im Prinzip durch die Negation der Formel ausgedrückt. Allerdings sind noch die freien Variablen zu beachten, die in der Definition der Gültigkeit (2.3.17) behandelt werden wie allquantifizierte Variablen, vgl. Faktum 2.3.18. Wenn beispielsweise die Formel '$f(x) \doteq g(x)$' in einer Algebra gilt, dann sind die Funktionen f und g auf *jedem* Argument identisch. Ist das Gegenteil der Fall, dann müssen f und g auf *mindestens einem* Argument verschiedene Werte liefern. Dies ist gleichbedeutend mit der Gültigkeit der Formel '$\neg\forall x.\ f(x) \doteq g(x)$' bzw. '$\exists x.\ f(x) \not\doteq g(x)$', nicht mit der Gültigkeit von '$f(x) \not\doteq g(x)$'. Letzteres würde ja heißen, daß f und g auf *keinem* Argument übereinstimmen dürfen. Das Gegenteil einer Formel wäre demnach die *Negation des Allabschlusses*, bzw. jede dazu äquivalente Formel, z.B. der *Existenzabschluß der Negation*. Um notationell leicht Zugriff auf 'das Gegenteil' zu haben, wählen wir, recht beliebig, einen Repräsentanten aus diesen Möglichkeiten.

Definition 2.4.1 (Gegenteil)
Zu einer Formel φ ist das **Gegenteil von** φ, kurz '$\mathit{Contr}(\varphi)$', definiert durch:

$$\mathit{Contr}(\varphi) \;=\; \mathit{Cl}_{\exists}(\neg\varphi)$$

⋆

Faktum 2.4.2 *Für geschlossene Formeln φ gilt: $\mathit{Contr}(\varphi) = \neg\varphi$.*

Was die Gültigkeit von Formeln betrifft, so gibt es einen wesentlichen Unterschied zwischen Algebren einerseits und Datentypen, d.h. Mengen von Algebren, andererseits. In einer Algebra ist intuitiv alles festgelegt. Darum gilt für jede Formel, daß entweder sie selbst oder ihr Gegenteil in der Algebra gilt (passende Signatur vorausgesetzt). In einem Datentyp hingegen ist intuitiv nur das festgelegt, worin all seine Modelle übereinstimmen. Intuitiv nicht festgelegt sind Formeln, die in bestimmten Modellen des Datentyps gelten, in anderen aber nicht. Für solche Formeln gilt, daß *weder* sie selbst *noch* ihr Gegenteil in dem Datentyp gelten (i.S.v. Def. 2.3.27). Im folgenden präzisieren wir diese Aussagen.

Faktum 2.4.3 *Sei* Σ *eine ADT-Signatur,* $\mathcal{A}$ *eine frei erzeugte* Σ*-Algebra und* $\varphi \in For_\Sigma$*. Dann gilt immer eine der beiden Aussagen:*

$$\mathcal{A} \models \varphi$$

oder

$$\mathcal{A} \models Contr(\varphi)$$

Insbesondere gilt natürlich immer nur höchstens eine der Aussagen aus Faktum 2.4.3.

Wenn eine Formel in einer Algebra nicht gilt, dann benutzen wir dafür ein eigenes Zeichen.

Notation 2.4.4 (Ungültigkeit in Algebren)
Sei Σ eine ADT-Signatur, $\mathcal{A}$ eine frei erzeugte Σ-Algebra und $\varphi \in For_\Sigma$.
φ heißt **ungültig in** $\mathcal{A}$, kurz '$\mathcal{A} \not\models \varphi$', falls $\mathcal{A} \models \varphi$ nicht gilt[7].

('$\mathcal{A} \not\models \Phi$' ensprechend). ⋆

Faktum 2.4.5 *Sei* Σ *eine ADT-Signatur,* $\mathcal{A}$ *eine frei erzeugte* Σ*-Algebra und* $\varphi \in For_\Sigma$*. Dann gilt:*

$$\mathcal{A} \not\models \varphi \quad \Longleftrightarrow \quad \mathcal{A} \models Contr(\varphi)$$

Das Zeichen $\models$ wurde ja in Abschn. 2.3.2 mehrfach überladen. Genauso werden wir jetzt $\not\models$ überladen.

Notation 2.4.6 (Ungültigkeit in Datentypen)
Sei Σ eine ADT-Signatur, $\mathcal{DT}$ ein frei erzeugter Σ-Datentyp und $\varphi \in For_\Sigma$.
φ heißt **ungültig in** $\mathcal{DT}$, kurz '$\mathcal{DT} \not\models \varphi$', falls $\mathcal{DT} \models \varphi$ nicht gilt.

('$\mathcal{DT} \not\models \Phi$' ensprechend). ⋆

Notation 2.4.7 (Nicht-Folgerbarkeit, Nicht-Allgemeingültigkeit)
Sei Σ eine ADT-Signatur, $\varphi \in For_\Sigma$ und $\Phi \subseteq For_\Sigma$.
Wir schreiben '$\Phi \not\models_\Sigma \varphi$', falls $\Phi \models_\Sigma \varphi$ nicht gilt, und '$\not\models_\Sigma \varphi$', falls $\models_\Sigma \varphi$ nicht gilt. ⋆

Notation 2.4.8 (Nicht-Folgerbarkeit aus Spezifikationen)
Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$.
Wir schreiben '$\langle \Sigma, AX \rangle \not\models \varphi$', falls $\langle \Sigma, AX \rangle \models \varphi$ nicht gilt. In diesem Fall heißt φ '**nicht folgerbar aus** $\langle \Sigma, AX \rangle$'. ⋆

[7] $\mathcal{A} \models \varphi$ muß aber wohldefiniert sein, s. Bem. 2.3.19.

Es kann nicht genug hervorgehoben werden, daß etwas dem Faktum 2.4.5 entsprechendes weder für die Gültigkeit in Datentypen noch für die Folgerbarkeit aus Formeln bzw. Spezifikationen gilt. Obwohl diese Tatsache gemäß den Definitionen in Abschn. 2.3.2 trivial ist, soll sie hier durch ein eigenes Faktum unterstrichen werden.

Faktum 2.4.9 *Es gibt eine ADT-Signatur Σ, einen Σ-Datentyp $\mathcal{DT}$, eine Formelmenge $\Phi \subseteq For_\Sigma$ und eine Formel $\varphi \in For_\Sigma$ mit:*

1. $\mathcal{DT} \not\models \varphi \quad \text{und} \quad \mathcal{DT} \not\models \mathit{Contr}(\varphi)$
2. $\Phi \not\models_\Sigma \varphi \quad \text{und} \quad \Phi \not\models_\Sigma \mathit{Contr}(\varphi)$
3. $\langle \Sigma, \Phi \rangle \not\models \varphi \quad \text{und} \quad \langle \Sigma, \Phi \rangle \not\models \mathit{Contr}(\varphi)$

Das folgende Beispiel ist gleichzeitig ein Beweis für dieses Faktum, dient aber hauptsächlich der Illustration.

Beispiel 2.4.10 Betrachten wir die Spezifikation Nat (s. Abb. 2.2 auf S. 25). Nun untersuchen wir die Formel, die besagt, daß die Vorgängerfunktion *pred* irreflexiv ist:

$$\varphi \quad = \quad \mathit{pred}(x) \mathrel{\dot{\neq}} x$$

Das Gegenteil von φ ist:[8]

$$\mathit{Contr}(\varphi) \quad = \quad \exists x.\ \mathit{pred}(x) \doteq x$$

Die Spezifikation Nat bestimmt den Vorgänger *pred* einer natürlichen Zahl n eindeutig, vorausgesetzt, es handelt sich bei n um den Nachfolger einer anderen Zahl. Für solche Konstruktorterme n gilt $\mathit{val}_\mathcal{I}(\mathit{pred}(\mathsf{n})) \neq \mathit{val}_\mathcal{I}(\mathsf{n})$, für jede Interpretation mit $\mathcal{I} \Vdash \mathsf{Nat}$.[9] Soweit spräche dies für die Gültigkeit von φ in dem Datentyp *Dat*(Nat). Nat legt sich aber nicht fest, wie *pred*(zero) auszuwerten ist. Zwei verschiedene Nat erfüllende Interpretationen $\mathcal{I}$ können sich im Wert von $\mathcal{I}(\mathit{pred})(\mathsf{zero})$ unterscheiden (und nur darin).

Sei nun $\mathcal{I}_1$ eine Interpretation mit $\mathcal{I}_1 \Vdash \mathsf{Nat}$, wobei $\mathcal{I}_1(\mathit{pred})(\mathsf{zero}) = \mathsf{zero}$. Dann gilt wegen $\mathcal{I}_1 \models \mathit{pred}(\mathsf{zero}) \doteq \mathsf{zero}$, daß $\mathcal{I}_1 \not\models \mathit{pred}(x) \mathrel{\dot{\neq}} x$. Somit auch $\mathsf{Nat} \not\models \mathit{pred}(x) \mathrel{\dot{\neq}} x$.

Sei nun $\mathcal{I}_2$ eine Interpretation mit $\mathcal{I}_2 \Vdash \mathsf{Nat}$, wobei $\mathcal{I}_2(\mathit{pred})(\mathsf{zero}) = \mathsf{succ}(\mathsf{zero})$. Also gilt $\mathcal{I}_2 \models \mathit{pred}(\mathsf{zero}) \mathrel{\dot{\neq}} \mathsf{zero}$, und darüber hinaus für alle Konstruktorterme n,

[8] Hierbei wurde '$\neg \mathrel{\dot{\neq}}$' gleich vereinfacht zu '$\doteq$'.

[9] Warum genau ist das der Fall? Weil n ein Konstruktorterm der Form $\mathsf{succ}(\mathsf{n}')$ ist. Für diesen gilt gemäß dem einzigen Axiom, daß $\mathit{val}_\mathcal{I}(\mathit{pred}(\mathsf{succ}(\mathsf{n}'))) = \mathit{val}_\mathcal{I}(\mathsf{n}') = \mathsf{n}'$. Damit ist $\mathit{val}_\mathcal{I}(\mathit{pred}(\mathsf{succ}(\mathsf{n}'))) \neq \mathit{val}_\mathcal{I}(\mathsf{succ}(\mathsf{n}')) = \mathsf{succ}(\mathsf{n}')$, da n' und $\mathsf{succ}(\mathsf{n}')$ verschiedene Elemente des Grundbereiches sind.

die nicht zero sind, ebenfalls $\mathcal{I}_2 \models \mathit{pred}(\mathsf{n}) \not\doteq \mathsf{n}$ (s.o.). Es folgt $\mathcal{I}_2 \not\models \exists x.\ \mathit{pred}(x) \doteq x$, und somit auch $\mathsf{Nat} \not\models \exists x.\ \mathit{pred}(x) \doteq x$.

Sei $\mathcal{DT} = \mathit{Dat}(\mathsf{Nat})$ und $\Phi = \mathit{Ax}(\mathsf{Nat})$. Dann gilt zusammenfassend:

1. $\mathcal{DT} \not\models \varphi$ und $\mathcal{DT} \not\models \mathit{Contr}(\varphi)$
2. $\Phi \not\models_\Sigma \varphi$ und $\Phi \not\models_\Sigma \mathit{Contr}(\varphi)$
3. $\mathsf{Nat} \not\models \varphi$ und $\mathsf{Nat} \not\models \mathit{Contr}(\varphi)$ ⋆

Faktum 2.4.9 ist charakteristisch für einen Ansatz, in dem die Semantik einer Spezifikation aufgefaßt wird als die Menge *aller Modelle* der Spezifikation (vgl. Def. 2.3.33). Genau dies nennt man *lose Semantik* (s. Abschn. 2.6.3). Bei der sogenannten *initialen Semantik* (ebd.) ist das anders. Dort faßt man immer *genau ein Modell* als Semantik einer Spezifikation auf, und zwar das sog. 'intiale Modell'. Somit gilt trivialerweise für jede Formel und jede Spezifikation (mit initialer Semantik), daß entweder die Formel oder ihr Gegenteil aus der Spezifikation folgt (entsprechend unserem Faktum 2.4.3).

Auch bei uns kann dieser Fall eintreten, und zwar wenn ein spezifizierter Datentyp monomorph ist.

Definition 2.4.11 (monomorphe vs. polymorphe Spezifikationen)
Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation.
$\langle \Sigma, AX \rangle$ heißt **monomorph** (**polymorph**), falls $\mathit{Dat}(\langle \Sigma, AX \rangle)$ monomorph (polymorph) ist. ⋆

Vergewissern wir uns kurz, daß es überhaupt monomorphe Spezifikationen gibt.

Satz 2.4.12

1. *Es gibt eine ADT-Signatur Σ und Axiome $AX \subseteq \mathit{For}_\Sigma$, so daß $\langle \Sigma, AX \rangle$ monomorph ist.*
2. *Es gibt (sogar) eine ADT-Signatur Σ' und Axiome $AX' \subseteq \mathit{For}_{\Sigma'}$, so daß $\langle \Sigma', AX' \rangle$ monomorph ist und das einzige Modell von $\langle \Sigma', AX' \rangle$ einen unendlichen Grundbereich besitzt.*

Der erste Teil des Satzes ergibt sich natürlich aus dem zweiten. Es ist aber illustrativer, beide Fälle getrennt zu betrachten. Auch wird dadurch der Vergleich zur allgemeinen erststufigen Logik interessanter, s.u. Die Beispiele im folgenden Beweis könnten noch minimaler sein, wären dann aber etwas pathologisch.

spec = SINGLETON
  sorts $Singleton$ ::= theOneAndOnly;
  functions id : $Singleton \to Singleton$;
  axioms
end

Abbildung 2.5: Spezifikation SINGLETON

spec = NAT_ID
  sorts Nat ::= zero | succ(Nat);
  functions id : $Nat \to Nat$;
  vars x : Nat;
  axioms $id(x) \doteq x$
end

Abbildung 2.6: Spezifikation NAT_ID

Beweis: (zu Satz 2.4.12)

1. Σ und AX ergeben sich aus der Spezifikation SINGLETON in Abb. 2.5. Dann ist Dat(SINGLETON) $= \{(\mathcal{CT}_\Sigma, \mathcal{I})\}$. Hierbei ist $\mathcal{CT}_\Sigma$ sowieso nicht variabel.[10] Die einzige Möglichkeit, $\mathcal{I}$ zu bilden, ist $\mathcal{I}(id)$(theOneAndOnly) $=$ theOneAndOnly.

2. Σ' und AX' ergeben sich aus der Spezifikation NAT_ID in Abb. 2.6. Dann ist Dat(NAT_ID) $= \{(\mathcal{CT}_{\Sigma'}, \mathcal{I}')\}$, wobei $\mathcal{I}'(id)$ nur die Identität sein kann.

□

Der zweite Teil von Satz 2.4.12 ist aus Sicht der Logik nicht selbstverständlich, und dies ist auch der Grund, warum wir den Satz überhaupt formuliert haben. In der Logik erster Ordnung gilt nämlich, daß eine Formelmenge, die überhaupt ein unendliches Modell besitzt, auch noch andere unendlichen Modelle besitzt, sogar 'beliebig schlimme': Der Satz von Löwenheim, Skolem und Tarski [EFT92, Satz 2.4] besagt, daß eine Formelmenge mit unendlichen Modellen auch Modelle beliebiger Kardinalität besitzt. Ohne die semantische Festlegung auf termerzeugte

[10] $\mathcal{CT}_\Sigma = \{\{\text{theOneAndOnly}\}_{Singleton}\}$

Bereiche ist demnach an monomorphe Spezifikationen unendlicher Datenstrukturen überhaupt nicht zu denken. Prinzipiell steht der semantische Ansatz, der auch dieser Abhandlung zugrunde liegt, zwischen den Extremen reiner erststufiger Semantik einerseits und initialer Semantik andererseits. Im Gegensatz zur reinen erststufigen Logik ermöglicht uns die Termerzeugtheit, überhaupt (auch unendliche Datentypen) monomorph spezifizieren zu können. Im Gegensatz zu initialer Semantik besteht andererseits die Möglichkeit, eben nicht beliebig genau spezifizieren zu müssen, sondern beim Erstellen einer Spezifikation intuitiv verschiedene Möglichkeiten der Interpretation zuzulassen, die sich in den unterschiedlichen Algebren des spezifizierten Datentyps wiederfinden. Dies bezeichnet man als Unterspezifikation.

Definition 2.4.13 (Unterspezifikation)
Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation.
Der dadurch spezifizierte Datentyp $\mathit{Dat}(\langle \Sigma, AX \rangle)$ heißt **unterspezifiziert**, falls er polymorph ist. $\star$

Ließ sich das Faktum 2.4.5 schon nicht auf allgemeine Spezifikationen übertragen, so doch auf monomorphe.

Faktum 2.4.14 *Falls eine ADT-Spezifikation $\langle \Sigma, AX \rangle$ monomorph ist, dann gilt für $\varphi \in \mathit{For}_\Sigma$:*

$$\langle \Sigma, AX \rangle \not\models \varphi \quad \Longleftrightarrow \quad \langle \Sigma, AX \rangle \models \mathit{Contr}(\varphi)$$

Gälte dies auch für allgemeine Spezifikationen, dann würde dies den Nachweis der Nicht-Folgerbarkeit bzw. Ungültigkeit erheblich erleichtern. Denn dann könnten wir die *Nicht-Folgerbarkeit* einer Formel auf die *Folgerbarkeit* einer anderen Formel reduzieren. Genau hiervon profitieren Ansätze, die die Nicht-Folgerbarkeit in einem Rahmen untersuchen, in dem Monomorphie garantiert ist, z.B. durch die semantische Beschränkung auf initiale Modelle oder durch syntaktische Restriktionen an die Axiome (vgl. 'konstruktive Spezifikationen' [LEW96, Abschnitt 8]).

Wir aber müssen uns mit weniger zufrieden geben. Aus Def. 2.3.27 und Faktum 2.4.3 resultiert das folgende einfache, aber sehr wichtige Korollar.

Korollar 2.4.15 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\mathcal{DT}$ ein f.e. Σ-Datentyp und $\varphi \in \mathit{For}_\Sigma$. Dann gilt:*

$$\mathcal{DT} \not\models \varphi$$
$$\Longleftrightarrow$$
$$\text{es existiert eine f.e. } \Sigma\text{-Algebra } \mathcal{A} \in \mathcal{DT} \text{ mit } \mathcal{A} \models \mathit{Contr}(\varphi)$$
$$\Longleftrightarrow$$
$$\text{es existiert eine } \mathcal{F}\text{-Interpretation } \mathcal{I} \text{ mit } \mathit{Alg}(\mathcal{I}) \in \mathcal{DT} \text{ und } \mathcal{I} \models \mathit{Contr}(\varphi)$$

Für die Nicht-Folgerbarkeit aus Spezifikationen gilt entsprechend:

Korollar 2.4.16 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$. Dann gilt:*

$$\langle \Sigma, AX \rangle \not\models \varphi$$
$$\Longleftrightarrow$$
$$\textit{es existiert ein Modell } \mathcal{A} \textit{ von } \langle \Sigma, AX \rangle \text{ [11] } \textit{mit } \mathcal{A} \models \mathit{Contr}(\varphi)$$
$$\Longleftrightarrow$$
$$\textit{es existiert eine } \langle \Sigma, AX \rangle \textit{ erfüllende } \mathcal{F}\textit{-Interpretation } \mathcal{I} \text{ [12] } \textit{mit } \mathcal{I} \models \mathit{Contr}(\varphi)$$

Um also unser zentrales Anliegen zu verwirklichen, '$\langle \Sigma, AX \rangle \not\models \varphi$' nachzuweisen, müssen wir eine Interpretation im Sinne dieses Korollars finden. Weil solche Interpretationen bzw. Modelle die Folgerbarkeit einer Formel aus einer Spezifikation widerlegen, nennen wir sie Gegeninterpretationen bzw. Gegenmodelle.

Definition 2.4.17 (Gegenmodelle, Gegeninterpretationen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$.

1. Ein **Gegenmodell zu** '$\langle \Sigma, AX \rangle \models \varphi$' ist eine Σ-Algebra $\mathcal{A}$ mit: $\mathcal{A} \Vdash \langle \Sigma, AX \rangle$ und $\mathcal{A} \models \mathit{Contr}(\varphi)$.
2. Eine **Gegeninterpretation zu** '$\langle \Sigma, AX \rangle \models \varphi$' ist eine $\mathcal{F}$-Interpretation $\mathcal{I}$ mit: $\mathcal{I} \Vdash \langle \Sigma, AX \rangle$ und $\mathcal{I} \models \mathit{Contr}(\varphi)$.

Ergibt sich $\langle \Sigma, AX \rangle$ aus dem Kontext, dann sprechen wir auch von einem **Gegenmodell zu** φ bzw. einer **Gegeninterpretation zu** φ. ⋆

Nun liegt es noch nahe, $\mathit{Contr}(\varphi)$ zu AX hinzuzufügen, um nach einer Interpretation für die Gesamtmenge zu suchen.

Definition 2.4.18 (Gegenspezifikation)
Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$. Dann ist die Spezifikation $\langle \Sigma, AX \cup \mathit{Contr}(\varphi) \rangle$ die **Gegenspezifikation zu** '$\langle \Sigma, AX \rangle \models \varphi$'. ⋆

Faktum 2.4.19 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$ und $\mathcal{I}$ eine $\mathcal{F}$-Interpretation. Dann gilt:*

$$\mathcal{I} \Vdash \langle \Sigma, AX \cup \mathit{Contr}(\varphi) \rangle$$
$$\Longleftrightarrow$$
$$\mathcal{I} \textit{ ist eine Gegeninterpretation zu } \langle \Sigma, AX \rangle \models \varphi$$

[11] d.h. $\mathcal{A} \Vdash \langle \Sigma, AX \rangle$
[12] d.h. $\mathcal{I} \Vdash \langle \Sigma, AX \rangle$

Daraus folgt unmittelbar:

Faktum 2.4.20 *Sei Σ eine ADT-Signatur, $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in For_\Sigma$. Dann gilt:*

$$\langle \Sigma, AX \rangle \not\models \varphi$$
$$\Longleftrightarrow$$
$$\textit{die Gegenspezifikation zu } \langle \Sigma, AX \rangle \models \varphi \textit{ ist erfüllbar}$$

Dieses Faktum kennzeichnet den hier verfolgten Ansatz. Wir wollen den Nachweis der Nicht-Folgerbarkeit führen mit Hilfe der Konstruktion von Modellen für bestimmte Spezifikationen, die 'das Gegenteil der Vermutung behaupten'. Dazu werden wir uns zunächst mit der Konstruktion von Modellen für allgemeine ADT-Spezifikationen beschäftigen. Erst bei der Deutung solcher Modelle wird dann wieder unterschieden werden zwischen den ursprünglichen Axiomen und (dem Gegenteil) der Vermutung.

2.5 Normalisierung von Spezifikationen

Die Methode zur Konstruktion von Modellen einer ADT-Spezifikation, die in Kapitel 3 vorgestellt wird, baut darauf auf, daß die Axiome einer bestimmten syntaktischen Normalform genügen, und zwar der gleichen, die auch bei vielen bekannten Beweisverfahren zugrunde gelegt wird. Wie schon gesagt, ist diese Normalform nicht als Einschränkung zu verstehen an Spezifikationen und Vermutungen, die auf (Nicht-)Folgerbarkeit untersucht werden soll. Stattdessen ist die Normalisierung als Zwischenschritt der Methode anzusehen. Zunächst bilden wir die Gegenspezifikation zu einer Folgerungsbeziehung '$\mathsf{SPEC} \models \varphi$', dann normalisieren wir die Gegenspezifikation, um anschließend Modelle derselben zu suchen. In diesem Abschnitt geht es nur um die Normalisierung und ihre Eigenschaften. Dabei wird die Normalisierung noch nicht einmal im Detail (algorithmisch) beschreiben, sie folgt zum größten Teil einem gängigen Muster, das an vielen Stellen nachgelesen werden kann. (Natürlich werden geeignete Referenzen angegeben.) Bekanntermaßen ist der einzige Schritt, der bei Normalisierung erststufiger Formeln semantisch nicht trivial ist, die sog. *Skolemisierung*. Diese wird oft als 'nur erfüllbarkeitserhaltend' beschrieben, was beim Nachweis der *Un*erfüllbarkeit völlig ausreicht. Wir aber untersuchen die Erfüllbarkeit von Spezifikationen und interessieren uns daher stärker für Modelle, als dies bei allgemeinen Beweisverfahren der Fall ist.

Tatsächlich ist der Zusammenhang zwischen den Modellen vor und nach der Skolemisierung (auch bei klassischer erststufiger Logik) sehr eng, was zwar nicht neu ist, aber in diesem Rahmen besonders hervorgehoben werden sollte. Darüberhinaus führen die syntaktischen und semantischen Besonderheiten der frei erzeugten

Datentypen zu einer Spezialisierung des allgemeinen Konzeptes der Skolemisierung. Wir müssen die Unterscheidung zwischen Konstruktoren und Funktionen berücksichtigen und deren semantische Folgen. Aus diesen Gründen passen wir im folgenden die 'klassische' Diskussion der Skolemisierung (vgl. [Fit96, Abschn. 8.3]) auf frei erzeugte Datentypen an. Dabei legen wir uns auf eine Variante der Skolemisierung fest, die für die intuitive Deutung eventueller Gegenmodelle besonders gut geeignet erscheint.

Zunächst einmal legen wir das Ziel der Normalisierung fest: eine Spezifikation, deren Axiome allesamt Klauseln sind, d.h. Disjunktionen von Gleichungen und Ungleichungen sind (vgl. Def. 2.3.1, S. 20).

Definition 2.5.1 (normalisierte ADT-Spezifikationen)
Eine ADT-Spezifikation $\langle \Sigma, AX \rangle$ heißt **normalisiert**, falls $AX \subseteq Kl_\Sigma$. ⋆

Da $\langle \Sigma, \{ax_1, \ldots, ax_n\} \rangle$ die gleichen Modelle hat wie $\langle \Sigma, \{ax_1 \wedge \ldots \wedge ax_n\} \rangle$[13], entspricht dies der gängigen *konjunktiven Normalform.*

Die Spezifikation Nat aus Abb. 2.2 (S. 25) ist natürlich normalisiert. Die Spezifikation NatStack aus Abb. 2.3 (S. 26) hingegen ist nicht normalisiert. Sie wäre es, wenn man das Axiom '$n \not\doteq n' \rightarrow del(n, \mathsf{push}(n', st)) \doteq \mathsf{push}(n', del(n, st))$' ersetzen würde durch die äquivalente Formel '$n \doteq n' \vee del(n, \mathsf{push}(n', st)) \doteq \mathsf{push}(n', del(n, st))$'.

Nun beschreiben wir die Transformation einer beliebigen ADT-Spezifikation in eine normalisierte ADT-Spezifikation. Zunächst eliminieren wir alle Äquivalenzen, indem in den Axiomen jede Unterformel '$\varphi_1 \leftrightarrow \varphi_2$' ersetzt wird durch '$(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$'. Dies ist nötig, weil wir für den nächsten Schritt wissen müssen, welche Unterformel eines Axioms positiv und welche negativ ist, und zwar in dem folgenden Sinne:

Definition 2.5.2 (positive und negative Unterformeln[14])
Sei Σ eine ADT-Signatur und $\varphi \in For_\Sigma$ ohne '$\leftrightarrow$'.
φ ist eine positive Unterformel von φ. Ist $\varphi = \varphi_{p1} \wedge \ldots \wedge \varphi_{pn}$ oder $\varphi = \varphi_{p1} \vee \ldots \vee \varphi_{pn}$, dann ist φ_{pi} eine positive Unterformel von φ (f.a. $i \in \{1, \ldots, n\}$). Ist $\varphi = \forall x.\varphi_p$ oder $\varphi = \exists x.\varphi_p$ oder $\varphi = \varphi_n \rightarrow \varphi_p$, dann ist φ_p eine positive Unterformel von φ.
Ist $\varphi = \neg\varphi_n$ oder $\varphi = \varphi_n \rightarrow \varphi_p$, dann ist φ_n eine negative Unterformel von φ.
Ist φ_1 eine positive (negative) Unterformel von φ_2 und φ_2 eine positive (negative)

[13] Strenggenommen gilt dies nur, wenn die freien Variablen der Axiome disjunkt sind, was sich aber immer erreichen läßt.

[14] Mit dem Terminus Unterformel meinen wir ein *bestimmtes Auftreten* einer Formel innerhalb einer anderen. Treten zwei syntaktisch gleiche Formeln an verschiedenen Stellen innerhalb einer dritten Formel auf, dann betrachten wir sie als verschiedene Unterformeln.

Unterformel von φ_3, dann ist φ_1 eine positive Unterformel von φ_3. Ist φ_1 eine positive (negative) Unterformel von φ_2 und φ_2 eine negative (positive) Unterformel von φ_3, dann ist φ_1 eine negative Unterformel von φ_3.

φ ist eine Unterformel von φ', falls φ eine positive oder negative Unterformel von φ' ist. ⋆

Gemäß der Intention dieser Definition wären die unmittelbaren Unterformeln einer Äquivalenz ambivalent, weswegen sie in dem ersten Schritt eliminiert wurden.

Beim späteren Beweis der erwünschten Eigenschaften der Skolemisierung wird das folgende 'Implikations-Ersetzungs'-Lemma von Nutzen sein, das wir in leicht abgewandelter Form zitieren aus [Fit96, Theorem 8.2.4]:

Lemma 2.5.3 *Sei Σ eine ADT-Signatur und seien $\{\varphi, \varphi', \psi, \psi'\} \subseteq For_\Sigma$ Formeln, für die gilt: φ ist eine Unterformel von ψ, und ψ' geht aus ψ hervor, indem φ durch φ' ersetzt wird. Dann gilt:*

1. *Gilt $\models_\Sigma \varphi \rightarrow \varphi'$ und ist φ eine positive Unterformel von ψ, dann folgt $\models_\Sigma \psi \rightarrow \psi'$.*
2. *Gilt $\models_\Sigma \varphi' \rightarrow \varphi$ und ist φ eine negative Unterformel von ψ, dann folgt ebenfalls $\models_\Sigma \psi \rightarrow \psi'$.*

Die Aufgabe des zweiten Schritt ist es, aus einer äquivalenzfreien Spezifikation alle 'positiven' Existenzquantoren sowie 'negative' Allquantoren zu entfernen mittels Skolemisierung. Dazu konzentrieren wir uns zunächst auf die Entfernung eines einzelnen solchen Quantors.

Definition 2.5.4 (einschrittige Skolemisierung)
Sei $\mathsf{SPEC} = \langle \Sigma, AX \rangle$ eine ADT-Spezifikation mit:

- $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$
- $S = \{s_1, \ldots, s_i, \ldots, s_n\}$
- $\mathcal{F} = \{F_{s_1}, \ldots, F_{s_i}, \ldots, F_{s_n}\}$
- $AX = \{ax_1, \ldots, ax_j, \ldots, ax_m\}$
- $\exists x.\varphi$ ist eine positive (bzw. $\forall x.\varphi$ ist eine negative) Unterformel von ax_j, mit $x \in V_{s_i}$

Des weiteren sei $\mathsf{SPEC}' = \langle \Sigma', AX' \rangle$ eine ADT-Spezifikation mit:

- $\Sigma' = (S, \mathcal{C}, \mathcal{F}', \alpha')$
- $\mathcal{F}' = \{F_{s_1}, \ldots, {F_{s_i}}', \ldots, F_{s_n}\}$
- ${F_{s_i}}' = F_{s_i} \cup \{f\}$, wobei f neu gegenüber Σ (d.h. $f \notin \overline{\mathcal{C}} \cup \overline{\mathcal{F}} \cup V_\Sigma$)
- $AX' = \{ax_1, \ldots, {ax_j}', \ldots, ax_m\}$
- ${ax_j}'$ entsteht aus ax_j, indem die positive Unterformel $\exists x.\varphi$ (bzw. die negative Unterformel $\forall x.\varphi$) in ax_j ersetzt wird durch:
$$\begin{cases} \varphi[x/f], & \text{falls } \mathit{frei}(\varphi) \setminus \{x\} = \emptyset \\ \varphi[x/f(x_1, \ldots, x_k)], & \text{falls } \mathit{frei}(\varphi) \setminus \{x\} = \{x_1, \ldots, x_k\} \end{cases}$$
- $\alpha'(f) = \begin{cases} \lambda, & \text{falls } \mathit{frei}(\varphi) \setminus \{x\} = \emptyset \\ [\mathit{sort}(x_1) \ldots \mathit{sort}(x_k)], & \text{falls } \mathit{frei}(\varphi) \setminus \{x\} = \{x_1, \ldots, x_k\} \end{cases}$
- $\alpha'(l) = \alpha(l)$ f.a. $l \in \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$

Dann ist SPEC′ eine **einschrittige Skolemisierung** von SPEC.
f heißt in diesem Kontext **Skolemfunktion**.
Ist $\alpha'(f) = \lambda$, dann heißt f darüberhinaus **Skolemkonstante**. ⋆

Diese Definition wird leider durch die korrekte Behandlung der Sorten verkompliziert. Die entscheidenden Punkte aber sind:

- Es wird eine beliebige positive $\exists$- (bzw. negative $\forall$-) Formel skolemisiert. Dabei wird insbesondere *nicht* vorausgesetzt, daß die Axiome in *Pränexnormalform* vorliegen. Dies nennt man auch *strukturelle Skolemisierung* (vgl. [BEL01, Def. 5.3]). Hierdurch können überflüssige Argumentpositionen für die neue Funktion f eingespart werden (vgl. [BEL01, Example 5.1]). Darüber hinaus erhält die neue Funktion f nur die Variablen als Argumente, die in der zu ersetzenden quantifizierten Formel frei vorkommen. Hierin folgen wir der Skolemisierungs-Variante, die bei Baaz et al. *Andrews' Skolem Form* genannt wird [BEL01, Def. 5.9], s.a. [Fit96, Theorem 8.3.2]. Hierdurch werden weitere überflüssige Argumentpositionen für die neue Funktion f eingespart. Die Einsparung von Argumentpositionen für f hat Konsequenzen für die Beweiskomplexität (s. [BEL01]). Dies kommt uns zwar auch bei der Suche nach Modellen zugute, aber das ist nicht die Hauptmotivation, die spezielle Form der Skolemisierung in obiger Definition zu verwenden. Stattdessen ist die Deutung einer Interpretation für f im Hinblick auf die ursprüngliche Formel besonders intuitiv.
- Da wir in ADT-Signaturen einen Unterschied machen zwischen Konstruktoren und Funktionen, müssen wir uns entscheiden, welcher Kategorie das neue Symbol f angehört. Es mag natürlich erscheinen, Skolemsymbole wie

geschehen als Funktionen aufzufassen. Wir wollen aber betonen, daß es hier einen grundlegenden Unterschied zu normaler erststufigen Logik gibt. Dort reduziert man die Frage nach der Existenz von Modellen auf die Frage nach der Existenz von Herbrand-Modellen.[15] Bei Herbrand-Modellen aber ist jede ‘Funktion’ ein Konstruktor in unserem Sinne, so daß auch die Skolemsymbole zum Aufbau der Modell-Grundmenge beitragen. Gerade dies ist der Grund dafür, daß in erststufiger Logik mehr Formeln erfüllbar und weniger Formeln gültig sind als im Termerzeugten, was weiter unten im Beispiel 2.5.15 demonstriert werden soll. Jedenfalls sind wir gezwungen, bei der Suche nach einem Modell für SPEC' auch für f eine Interpretation zu konstruieren.

- Wie jede andere Skolemisierungs-Variante ist auch die in Def. 2.5.4 definierte nicht deterministisch. Sie ist sogar besonders indeterministisch, da die betroffenen Quantoren nicht von außen nach innen eliminiert werden müssen.

Nun wollen wir den Zusammenhang herstellen zwischen den Modellen einer Spezifikation und den Modellen ihrer (einschrittigen) Skolemisierung. Dazu benötigen wir noch ein paar Vorbereitungen.

Definition 2.5.5 (Redukt)
Sei $\Sigma' = (S, \mathcal{C}, \mathcal{F}', \alpha')$ eine Funktionserweiterung von $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ (vgl. Def. 2.2.7) und $\mathcal{A}' = (\mathcal{CT}_{\Sigma'}, \mathcal{I}')$[16] eine Σ'-Algebra. Das **Σ-Redukt von** $\mathcal{A}'$, kurz ‘$\mathcal{A}'|_\Sigma$’, ist diejenige Σ-Algebra $\mathcal{A} = (\mathcal{CT}_\Sigma, \mathcal{I})$, in der f.a. $f \in \overline{\mathcal{F}}$ gilt: $\mathcal{I}(f) = \mathcal{I}'(f)$. ⋆

Eine unmittelbare Konsequenz dieser Definition ist, daß Formeln, die sich syntaktisch auf Σ beschränken, in $\mathcal{A}'$ und $\mathcal{A}$ gleich ausgewertet werden.

Korollar 2.5.6 *Sei Σ' eine Funktionserweiterung von Σ und $\mathcal{A}'$ eine Σ'-Algebra. Sei außerdem $\Phi \subseteq For_\Sigma$, somit sowieso $\Phi \subseteq For_{\Sigma'}$ (s. Faktum 2.3.2). Dann gilt:*

$$\mathcal{A}' \models \Phi \quad \Longleftrightarrow \quad \mathcal{A}'|_\Sigma \models \Phi.$$

Nun beleuchten wir den Zusammenhang zwischen den Formeln, die in Def. 2.5.4 ‘wegsubstituiert’ werden, und ihrem Substitut. Der entsprechende Satz arbeitet mit der Vorraussetzung, daß die betrachteten Formeln ‘bereinigt’ sind, was bedeutet, daß a) freie und gebundene Variablen verschieden heißen und b) verschiedene Quantoren immer verschiedene Variablen binden. Diese Voraussetzung ist unproblematisch, da sie sich immer durch Umbenenung von Variablen herstellen läßt.

[15] Siehe [Fit96], Theorem 5.9.4, mit ‘L^{sko}’ (Def. 7.4.1) statt ‘L^{par}’.
[16] $\mathcal{CT}_{\Sigma'} = \mathcal{CT}_\Sigma$

Definition 2.5.7 (bereinigte Formeln und Spezifikationen)
Sei Σ eine ADT-Signatur.

Eine **Formel** $\varphi \in For_\Sigma$ ist **bereinigt**, wenn gilt:
Falls $\psi_1 = Q_1 x.\psi'_1$ eine Unterformel von φ ist (mit $Q_1 \in \{\forall, \exists\}$), dann gilt:
1. $x \notin \mathit{frei}(\varphi)$,
2. ist $\psi_2 = Q_2 x.\psi'_2$ eine Unterformel von φ (mit $Q_2 \in \{\forall, \exists\}$), dann handelt es sich bei ψ_1 und ψ_2 um dieselbe[17] Unterformel von φ.

Eine ADT-**Spezifikation** $\langle \Sigma, AX \rangle$ ist **bereinigt**, wenn jedes Axiom $ax \in AX$ bereinigt ist. ⋆

Lemma 2.5.8 *Sei Σ eine ADT-Signatur und $\varphi \in For_\Sigma$ eine bereinigte Formel. O.B.d.A. nehmen wir an, daß $\mathit{frei}(\varphi) \setminus \{x\} = \{x_1, \ldots, x_k\}$. Die ADT-Signatur Σ' ergebe sich aus Σ und der Formel $\exists x.\varphi$ bzw. $\forall x.\varphi$ genau wie in Def. 2.5.4 beschrieben (d.h., kurz gesagt, durch Erweiterung von Σ um eine neue Funktion f, deren Sorte und Argumentsorten sich aus den Variablen in $\exists x.\varphi$ bzw. $\forall x.\varphi$ ergibt).*

Sei $\mathcal{A}$ eine beliebige Σ-Algebra. Dann existieren Σ'-Algebren $\mathcal{A}_1$ und $\mathcal{A}_2$, so daß $\mathcal{A}_1|_\Sigma = \mathcal{A}_2|_\Sigma = \mathcal{A}$ (d.h. $\mathcal{A}_1$ und $\mathcal{A}_2$ unterscheiden sich von $\mathcal{A}$ nur durch die Interpretation von f), und so daß weiterhin gilt:

1. $\mathcal{A}_1 \models \exists x.\varphi \rightarrow \varphi[x/f(x_1, \ldots, x_k)]$
2. $\mathcal{A}_2 \models \varphi[x/f(x_1, \ldots, x_k)] \rightarrow \forall x.\varphi$

Beweis: $\mathcal{A}_1 = (\mathcal{CT}_\Sigma, \mathcal{I}_1)$ und $\mathcal{A}_2 = (\mathcal{CT}_\Sigma, \mathcal{I}_2)$ liegen fest bis auf die Interpretation von f. Für beliebige $\{ct_1, \ldots, ct_k\} \subseteq CT_\Sigma$ konstruieren wir nun $\mathcal{I}_1(f)(ct_1, \ldots, ct_k)$ und $\mathcal{I}_2(f)(ct_1, \ldots, ct_k)$. Dazu fixieren wir jeweils eine Variablenbelegung β mit $\beta(x_1) = ct_1$, …, $\beta(x_k) = ct_k$.

Wir beweisen zunächst die Aussage '*1.*', wobei wir zwei Fälle unterscheiden: a) Es gilt $val_{\mathcal{I}_1,\beta}(\exists x.\varphi) = F$. (Dies ist wohldefiniert, da f in φ nicht vorkommt.) Dann können wir für $\mathcal{I}_1(f)(ct_1, \ldots, ct_k)$ ein beliebiges Element aus CT_{s_i} wählen, und es gilt $val_{\mathcal{I}_1,\beta}(\exists x.\varphi \rightarrow \varphi[x/f(x_1, \ldots, x_k)])$. b) Es gilt $val_{\mathcal{I}_1,\beta}(\exists x.\varphi) = W$. Dann gilt laut Satz 2.3.16 für ein $ct \in CT_{s_i}$, daß $val_{\mathcal{I}_1,\beta}(\varphi[x/ct]) = W$. Für ein solches ct wählen wir $\mathcal{I}_1(f)(ct_1, \ldots, ct_k) = ct$. Dann gilt aufgrund der Kompositionalität der Formelauswertung, daß $val_{\mathcal{I}_1,\beta}(\varphi[x/f(x_1, \ldots, x_k)]) = W$, und somit auch $val_{\mathcal{I}_1,\beta}(\exists x.\varphi \rightarrow \varphi[x/f(x_1, \ldots, x_k)]) = W$.

Nun zu Assage '*2.*'. Wir unterscheiden wieder zwei Fälle: a) Es gilt $val_{\mathcal{I}_2,\beta}(\forall x.\varphi) = W$. Dann können wir für $\mathcal{I}_2(f)(ct_1, \ldots, ct_k)$ ein beliebiges Element aus CT_{s_i} wählen, und es gilt $val_{\mathcal{I}_2,\beta}(\varphi[x/f(x_1, \ldots, x_k)] \rightarrow \forall x.\varphi)$. b) Es gilt $val_{\mathcal{I}_2,\beta}(\forall x.\varphi) = F$. Dann gilt laut Satz 2.3.16 für ein $ct \in CT_{s_i}$, daß $val_{\mathcal{I}_2,\beta}(\varphi[x/ct]) = F$. Für ein

[17] vgl. Fußnote zu Def. 2.5.2

solches ct wählen wir $\mathcal{I}_2(f)(ct_1, \ldots, ct_k) = ct$. Dann gilt gemäß obigem Argument $val_{\mathcal{I}_2,\beta}(\varphi[x/f(x_1, \ldots, x_k)]) = F$, und somit $val_{\mathcal{I}_2,\beta}(\varphi[x/f(x_1, \ldots, x_k)] \to \forall x.\varphi) = W$. □

Falls $frei(\varphi) \setminus \{x\} = \emptyset$, kann man entsprechend die Existenz von $\mathcal{A}_1$ und $\mathcal{A}_2$ zeigen, so daß $\mathcal{A}_1 \models \exists x.\varphi \to \varphi[x/f]$ und $\mathcal{A}_2 \models \varphi[x/f] \to \forall x.\varphi$.

Lemma 2.5.9 *Sei Σ eine ADT-Signatur und $\psi_1 \in For_\Sigma$ eine bereinigte Formel der Form ax_j aus Def. 2.5.4 (d.h. ψ_1 enthält eine positive Unterformel der Form $\exists x.\varphi$ oder eine negative Unterformel der Form $\forall x.\varphi$). Seien Σ' bzw. ψ_2 gegeben wie Σ' bzw. $ax_j{}'$ in Def. 2.5.4. Dann existiert eine Σ'-Algebra $\mathcal{A}'$ mit $\mathcal{A}'|_\Sigma = \mathcal{A}$, so daß gilt:*

$$\mathcal{A}' \models \psi_1 \to \psi_2$$

Beweis: Falls ψ_1 eine positive Unterformel der Form $\exists x.\varphi$ enthält, dann existiert wegen Lemma 2.5.8 ein $\mathcal{A}'$ mit $\mathcal{A}'|_\Sigma = \mathcal{A}$, wobei $\mathcal{A}' \models \exists x.\varphi \to \varphi[x/f(x_1, \ldots, x_k)]$. Falls ψ_1 eine negative Unterformel der Form $\forall x.\varphi$ enthält, dann existiert wegen Lemma 2.5.8 ein $\mathcal{A}'$ mit $\mathcal{A}'|_\Sigma = \mathcal{A}$, wobei $\mathcal{A}' \models \varphi[x/f(x_1, \ldots, x_k)] \to \forall x.\varphi$. In beiden Fällen gilt dann aufgrund von Lemma 2.5.3, daß $\mathcal{A}' \models \psi_1 \to \psi_2$ gilt. □

Nach diesen Vorbereitungen formulieren wir die zwei wesentlichen Eigenschaften der einschrittigen Skolemisierung:

Satz 2.5.10 *Seien die ADT-Spezifikationen* SPEC *und ihre einschrittige Skolemisierung* SPEC′ *gegeben genau wie in Def. 2.5.4, wobei* SPEC *bereinigt ist. Dann gilt:*

1. *Ist* SPEC *erfüllbar, dann ist auch* SPEC′ *erfüllbar.*
2. *Ist $\mathcal{A}'$ ein Modell von* SPEC′*, dann ist $\mathcal{A} = \mathcal{A}'|_\Sigma$ ein Modell von* SPEC.

Beweis:

1. Sei $\mathcal{A}$ ein Modell für SPEC. Laut Lemma 2.5.9 existiert eine Σ'-Algebra $\mathcal{A}'$ mit $\mathcal{A}'|_\Sigma = \mathcal{A}$, so daß $\mathcal{A}' \models ax_j \to ax_j{}'$ gilt. Wir zeigen, daß $\mathcal{A}'$ ein Modell von SPEC′ ist. Aufgrund von Korollar 2.5.6 gelten die Axiome AX aus SPEC auch in $\mathcal{A}'$: $\mathcal{A}' \models AX$. Insbesondere gelten also die nicht veränderten Axiome: $\mathcal{A}' \models AX \setminus \{ax_j\}$. Das veränderte Axiom $ax_j{}'$ gilt auch: wegen $\mathcal{A}' \models ax_j$ und $\mathcal{A}' \models ax_j \to ax_j{}'$ folgt $\mathcal{A}' \models ax_j{}'$.
2. Sei $\mathcal{A}'$ ein Modell von SPEC′. Es gilt sicherlich (vgl. Satz 2.3.16):

   a) $\models_{\Sigma'} \varphi[x/f(x_1, \ldots, x_k)] \to \exists x.\varphi$

   b) $\models_{\Sigma'} \forall x.\varphi \to \varphi[x/f(x_1, \ldots, x_k)]$

Mit Lemma 2.5.3 folgt hieraus $\models_{\Sigma'} {ax_j}' \rightarrow ax_j$. Da $\mathcal{A}' \models {ax_j}'$, gilt also auch $\mathcal{A}' \models ax_j$ und damit insgesamt $\mathcal{A}' \models AX$ (für das ungestrichene AX!). Sei $\mathcal{A} = \mathcal{A}'|_\Sigma$. Dann folgt aus Korollar 2.5.6: $\mathcal{A} \models AX$.

□

Dieses Ergebnis überträgt sich leicht auf die vollständige Skolemisierung *aller* positiven existentiellen und negativen universellen Unterformeln.

Definition 2.5.11 (Anzahl Skolemisierbarer Unterformeln)
Sei Σ eine ADT-Signatur und $\varphi \in For_\Sigma$. φ enthalte k_1 existenzquantifizierte positive Unterformeln und k_2 allquantifizierte negative Unterformeln.[18] Dann ist $(k_1 + k_2)$ die 'Anzahl Skolemisierbarer Unterformeln' von φ. ⋆

Definition 2.5.12 (vollständige Skolemisierung)
Sei $\mathsf{SPEC} = \langle \Sigma, \{ax_1, \ldots, ax_n\} \rangle$ eine ADT-Spezifikation und su_i die Anzahl Skolemisierbarer Unterformeln von ax_i (f.a. $i \in \{1, \ldots, n\}$). SPEC' entstehe aus SPEC durch $(\sum_{i=1}^{n} su_i)$-fache Anwendung der einschrittigen Skolemisierung gemäß Def. 2.5.4. Dann ist SPEC' die **(vollständige) Skolemisierung** von SPEC. ⋆

Satz 2.5.13 *Ist* $\mathsf{SPEC} = \langle \Sigma, AX \rangle$ *eine bereinigte ADT-Spezifikation und* SPEC' *die vollständige Skolemisierung von* SPEC*, dann gilt:*

1. *Ist* SPEC *erfüllbar, dann ist auch* SPEC' *erfüllbar.*
2. *Ist* $\mathcal{A}'$ *ein Modell von* SPEC'*, dann ist* $\mathcal{A} = \mathcal{A}'|_\Sigma$ *ein Modell von* SPEC.

Beweis: '*1.*' und '*2.*' zeigt man leicht durch Induktion über die Anzahl der Anwendungen einschrittiger Skolemisierung. Für '*2.*' nutzt man dabei die Transitivität der Reduktbildung aus: Ist $\mathcal{A}_1 = \mathcal{A}_2|_{\Sigma_1}$ und $\mathcal{A}_2 = \mathcal{A}_3|_{\Sigma_2}$, dann ist $\mathcal{A}_1 = \mathcal{A}_3|_{\Sigma_1}$. □

Damit wissen wir 1. daß uns durch die Skolemisierung keine Modelle 'verloren gehen', und 2. wie wir aus den Modellen Skolemisierter Spezifikationen wieder auf Modelle der Ursprungsspezifikation zurückschließen können, indem wir nämlich die Interpretation der Skolemfunktionen schlicht 'vergessen'.

Ein konkretes Verfahren zur Konstruktion von Modellen sollte dennoch bei der Rückmeldung an den Benutzer die Interpretation der Skolemfunktionen nicht einfach vergessen, denn diese enthält durchaus wertvolle Information. Sie liefert intuitiv einen 'Zeugen' für die weg-Skolemisierte quantifizierte Unterformel, bzw. eine 'Belegung' für die weg-Skolemisierte Variable. Mit ihrer Hilfe kann man möglicherweise besser nachvollziehen, warum eine Spezifikation erfüllbar ist.

[18] vgl. Fußnote zu Def. 2.5.2

```
spec = Gegen_Irrefl
    sorts       Nat ::= zero | succ(Nat);
    functions   pred : Nat → Nat;
    vars        x, y : Nat;
    axioms      pred(succ(x)) ≐ x;
                ∃y. ¬(pred(y) ≠ y);
end
```

Abbildung 2.7: Spezifikation Gegen_Irrefl

Eine gute Möglichkeit, die Deutung von Modellen Skolemisierter Spezifikationen zu unterstützten, ist die Verwendung geeigneter Namenskonventionen bei der Einführung neuer Skolemfunktionen, so daß man diese Funktion nachträglich wieder mit der dem entspechenden Quantor in Verbindung bringen kann. Besonders wichtig wird dies beim Nachweis der Nicht-Folgerbarkeit einer Formel φ. Die entsprechende Gegenspezifikation enthält das Gegenteil von φ, $Cl_\exists(\neg\varphi)$, als Axiom. Hier sieht man auch, daß die Notwendigkeit, zu Skolemisieren, nicht unbedingt aus der ursprünglichen Spezifikation herrührt. Stattdessen müssen wir Gegenspezifikationen mindestens dann Skolemisieren, wenn die Vermutung φ freie Variablen enthält oder explizit allquantifiziert ist. Dies ist fast immer der Fall. Die Interpretation der Skolemkonstanten(!), die aus der Skolemisierung des Existenzabschlusses in $Cl_\exists(\neg\varphi)$ entstehen, liefert uns intuitiv eine 'Belegung' der freien Variablen in φ, die (in Verbindung mit einer bestimmten Interpretation der übrigen Funktionen) zur Nicht-Folgerbarkeit führt.

Beispiel 2.5.14 Betrachten wir die Spezifikation Nat (Abb. 2.2, S. 25). Wir hatten uns schon in Beispiel 2.4.10 davon überzeugt, daß die Irreflexivität der Vorgängerfunktion *pred* nicht aus Nat folgt: Nat $\not\models$ $pred(y) \neq y$. Folglich muß die Gegenspezifikation zu Nat $\models$ $pred(y) \neq y$, die in Abb. 2.7 wiedergegeben ist, ein Modell besitzen. Die Skolemisierung dieser Gegenspezifikation führt zu der in Abb. 2.8 angegebenen Spezifikation. Das zweite Axiom läßt sich vereinfachen zu $pred(sk_y) \doteq sk_y$. Diese Spezifikation besitzt genau ein Modell, d.h. genau eine erfüllende Interpretation $\mathcal{I}$. In dieser gilt: $\mathcal{I}(sk_y)() = \mathsf{zero}$ und $\mathcal{I}(pred)(\mathsf{zero}) = \mathsf{zero}$. Für die ursprüngliche Vermutung $pred(y) \neq y$ bedeutet dies intuitiv, daß die 'Belegung' von y mit zero (in Verbindung mit der genannten Interpretation von *pred*) für die Nicht-Folgerbarkeit verantwortlich ist. ⋆

Bevor wir das Thema Skolemisierung verlassen, wollen wir nocheinmal die Erfüllbarkeit in unserer termerzeugten Semantik mit der Erfüllbarkeit in reiner erststufiger Logik vergleichen, und zwar an folgendem Beispiel.

```
spec = Sko_Gegen_Irrefl
    sorts       Nat ::= zero | succ(Nat);
    functions   pred : Nat → Nat;
                sk_y : Nat;
    vars        x : Nat;
    axioms      pred(succ(x)) ≐ x;
                ¬(pred(sk_y) ≢ sk_y);
end
```

Abbildung 2.8: Spezifikation Sko_Gegen_Irrefl

Beispiel 2.5.15 Wir erweitern die Spezifikation p_Forever (Abb. 2.4, S. 35) um ein weiteres Axiom, s. Abb. 2.9, den Namen der Spezifikation um ein Fragezeichen ergänzend. p_Forever_? ist unerfüllbar: einerseits müßte in einer in Frage kommenden erfüllenden Interpretation $\mathcal{I}(p)(ct) = \mathsf{tt}$ $(*)$ gelten, f.a. Konstruktorterme ct, andererseits verlangt das letzte Axiom, daß $\mathcal{I}(p)(ct_1) \neq \mathsf{tt}$ gilt für mindestens einen Konstruktorterm ct_1. Genauso unerfüllbar ist die Skolemisierung dieser Spezifikation, in der das letzte Axiom $p(sk_y) \not\doteq \mathsf{tt}$ lautet (wobei der Signatur sk_y als neues Funktionssymbol hinzugefügt wird). Natürlich folgt die Unerfüllbarkeit der Skolemisierten Variante bereits aus Satz 2.5.13. Wir können uns aber auch direkt davon überzeugen, daß jede mögliche Interpretation von sk_y durch einen Konstruktorterm ct_2 die Gültigkeit von $\mathcal{I}(p)(ct_2) \neq \mathsf{tt}$ nach sich ziehen würde (gemäß dem Axiom $p(sk_y) \not\doteq \mathsf{tt}$), was wiederum $(*)$ (s.o.) widerspricht.

Betrachten wir nun die Skolemisierte Spezifikation aus der Perspektive reiner erststufiger Logik, d.h. ohne besondere Annahmen an die semantischen Grundmengen, wobei wir die Konstruktoren als normale Funktionen auffassen. Die so gedeutete Spezifikation ist erfüllbar durch eine Herbrand-Algebra[19]. Deren Grundmenge besteht aus Äquivalenzklassen der aus $\mathsf{tt}, \mathsf{ff}, p, \mathsf{zero}, \mathsf{succ}$ und sk_y aufgebauten Terme, und in ihr gilt $val_{\mathcal{I}}(p(sk_y)) \neq val_{\mathcal{I}}(\mathsf{tt})$ sowie $val_{\mathcal{I}}(p(\overline{t_1})) = val_{\mathcal{I}}(\mathsf{tt})$, für jeden Term $t_1 \in \{\mathsf{zero}, \mathsf{succ}(\mathsf{zero}), \ldots\}$. Diese Herbrand-Algebra ist auch ein Modell der ursprünglichen Spazifikation p_Forever_?, weswegen diese in reiner erststufiger Logik erfüllbar wäre! Es ist zu beachten, daß die Grundmenge dieses Modells von p_Forever_? nicht aus der Signatur von p_Forever_? aufgebaut wird. Die genannte Herbrand-Algebra ist zwar eine Term-Algebra, aber nur gegenüber einer durch Skolemisierung erweiterten Signatur!

Man sieht, daß im allgemeinen mehr Formeln bzw. Spezifikationen erfüllbar sind als in unserer termerzeugten Semantik. Umgekehrt sind bei uns mehr Formeln

[19] s. [SA91, Def. 2.10]

```
spec = p_Forever_?
    sorts        Nat ::= zero | succ(Nat);
                 Bool ::= tt | ff;
    functions    p : Nat → Bool;
    vars         x : Nat;
    axioms       p(zero) ≐ tt;
                 p(x) ≐ tt → p(succ(x)) ≐ tt;
                 ∃y. (p(y) ≠̇ tt);
end
```

Abbildung 2.9: Spezifikation p_Forever_?

gültig bzw. folgerbar als im allgemeinen. So gilt z.B. p_Forever $\models p(y) \doteq \mathsf{tt}$ (p_Forever ohne '?'), was unter allgemeiner Semantik nicht der Fall wäre. ⋆

Nach der Entfernung der Äquivalenzen im ersten und der vollständigen Skolemisierung im zweiten Schritt steht jetzt noch aus, in einem dritten Schritt die Menge der Axiome so umzuformen, daß nur noch Disjunktionen von Gleichungen und Ungleichungen übrig bleiben. Wir halten zunächst fest, daß die nach der vollständigen Skolemisierung verbliebenen Quantoren einfach 'gestrichen' werden können, gegebenenfalls nach Umbenennung der gebundenen Variablen. Dies liegt daran, daß nicht Skolemisierbare Quantoren in einem gewissen Sinne universell sind, so daß man statt ihrer auch die implizite All-Quantifikation verwenden kann.

Definition 2.5.16 (nicht Skolemisierbare Formeln)
Sei Σ eine ADT-Signatur und $\psi \in For_\Sigma$. ψ heißt **nicht Skolemisierbar**, falls ψ bereinigt und frei von Äquivalenzen ist, sowie keine positive Unterformel der Form $\exists x.\varphi$ und keine negative Unterformel der Form $\forall x.\varphi$ besitzt. ⋆

Lemma 2.5.17 *Sei Σ eine ADT-Signatur und $\psi \in For_\Sigma$ nicht Skolemisierbar. Sei $\forall x.\varphi$ bzw. $\exists x.\varphi$ eine Unterformel von ψ, wobei $x \notin frei(\psi)$. Desweiteren entstehe ψ' aus ψ durch Ersetzung von $\forall x.\varphi$ (bzw. $\exists x.\varphi$) durch φ. Dann gilt:*

$$\models_\Sigma \psi \leftrightarrow \psi'$$

Beweis: Zunächst überzeuge man sich, daß in einer nicht Skolemisierbaren Formel All- und Existenz-Quantoren nicht direkt aufeinander folgen können, d.h. es existieren keine Unterformeln der Form $\forall x.\exists y.\varphi$ oder $\exists x.\forall y\varphi$. Desweiteren kann man von jeder der üblichen äquivalenzerhaltenden Quantoren-Shifting-Regeln (s.

[SA91, Lemma 2.7]) leicht zeigen, daß sie die Eigenschaft der Nicht-Skolemisierbarkeit erhalten. Damit können wir nun zeigen, daß ψ äquivalent ist zu $\forall x.\psi'$: Man kann nämlich durch besagte Quantoren-Shifting-Regeln und wegen der Vertauschbarkeit zweier All- bzw. zweier Existenz-Quantoren den Quantor, der x bindet, ganz nach außen shiften. Zwar können $\forall$ und $\exists$ dabei sozusagen 'unterwegs' wechseln, aber da die Nicht-Skolemisierbarkeit erhalten bleibt, kann am Ende ganz außen nur ein All-Quantor stehen. Schließlich gilt wegen $x \notin \mathit{frei}(\psi)$ noch $\models_\Sigma (\forall x.\psi') \leftrightarrow \psi'$. □

Statt also bei der Normalisierung sog. Quantoren-Shifting betreiben zu müssen, haben wir das Quantoren-Shifting nur zu dem Zweck benutzt, die Korrektheit der schlichten Quantoren-Streichung zu zeigen.

Liegt also nach eventueller Bereinigung sowie vollständiger Skolemisierung eine Spezifikation mit nicht Skolemisierbaren Axiomen vor, dann führen folgende Schritte zu einer normalisierten Spezifikation: 1. Alle Quantoren werden gestrichen wie in Lemma 2.5.17 beschrieben. 2. Die Implikation ($\rightarrow$) und die Negation ($\neg$) werden komplett eliminiert durch Herstellung der Negations-Normalform (s. [BEL01, Def. 4.2]) und anschließende Anwendung von $\models_\Sigma \neg(t \doteq t') \leftrightarrow t \not\doteq t'$ und $\models_\Sigma \neg(t \not\doteq t') \leftrightarrow t \doteq t'$. 3. Durch Anwendung der Distributivität des $\vee$ über das $\wedge$ erreicht man für jedes Axiom eine Konjunktionen reiner Disjunktionen. 4. Die Konjunktionsglieder jedes Axioms werden in separate Axiome aufgespalten. (5. Wenn es der Übersichtlichkeit dient: Die Variablen unterschiedlicher Axiome können wieder gleich benannt werden.)

Somit ist die Normalisierung von Spezifikationen abgeschlossen. Da unsere Mothode Modelle normalisierter Gegenspezifikation suchen soll, definieren wir diese explizit.

Definition 2.5.18 (normalisierte Gegenspezifikation)
Sei $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in \mathit{For}_\Sigma$. Die **normalisierte Gegenspezifikation zu** '$\langle \Sigma, AX \rangle \models \varphi$' ist das Ergebnis der oben beschriebenen Normalisierung angewendet auf $\langle \Sigma, AX \cup \mathit{Contr}(\varphi) \rangle$. ⋆

Aus dem Faktum 2.4.20, dem Satz 2.5.13 über die vollständige Skolemisierung und der Äquivalenzerhaltung der sonstigen Normalisierungsschritte folgt:

Korollar 2.5.19 *Sei Σ eine ADT-Signatur, $\langle \Sigma, AX \rangle$ eine ADT-Spezifikation und $\varphi \in \mathit{For}_\Sigma$. Dann gilt:*

$$\langle \Sigma, AX \rangle \not\models \varphi$$
$$\Longleftrightarrow$$
die normalisierte Gegenspezifikation zu $\langle \Sigma, AX \rangle \models \varphi$ ist erfüllbar

Dieses Korollar kennzeichnet unser prinzipielles Vorgehen zur Untersuchung von '$\langle \Sigma, AX \rangle \not\models \varphi$': Wir bilden (erstens) die Gegenspezifikation zu '$\langle \Sigma, AX \rangle \models \varphi$',

normalisieren diese (zweitens) und suchen dann (drittens) nach Modellen der so entstandenen Spezifikation.

2.6 Diskussion

Die Definitionen dieses Kapitels (2), darauf wurde bereits mehrfach hingewiesen, sind speziell zugeschnitten auf unsere Bedürfnisse: die ausschließliche Behandlung *frei erzeugter* Datentypen und die Fokussierung auf die Beschreibung von Modellen. Dies betrifft insbesondere unsere strikte Trennung von Konstruktoren und Funktionen und die darauf aufbauende Definition von frei erzeugten Algebren (Def. 2.2.31), sowie die Definition von frei erzeugten Datentypen (Def. 2.2.34). Hier besteht aber noch Erklärungsbedarf. In der Literatur über abstrakte Datentypen werden sowohl die Term-Erzeugtheit (kurz 'Erzeugtheit') als auch ihr Spezialfall, die freie Erzeugtheit, formal definiert als mögliche Eigenschaften allgemeinerer Algebren. Wir aber haben weder gesagt, was allgemeine Algebren sind, noch, wann eine Algebra frei erzeugt ist. Stattdessen haben wir ein bestimmtes Gebilde einfach 'frei erzeugte Algebra' genannt. Nun wollen wir noch klären, was dieses Gebilde mit dem gleichnamigen aus der Literatur zu tun hat. Dies ist notwendig, um sicherzustellen, daß die im folgenden dargestellte Methodik auch im allgemeinverstandenen Sinne die Fragestellungen der Erfüllbarkeit bzw. Nicht-Folgerbarkeit in frei erzeugten Datentypen bearbeitet.

Um uns in einen allgemeineren Rahmen einzuordnen, müssen wir nun nachholen, was wir ja eigentlich gerade vermeiden wollten: den Schritt ins Allgemeine und von da aus zurück ins Spezielle. Bisher sind wir z.B. ausgekommen ohne den Begriff des Homomorphismus. Nun brauchen wir ihn doch noch. Wir wollen aber diesen Bruch im bisherigen Duktus auf diesen Abschnitt beschränken. Wenn der Vergleich mit den gleichlautenden Begriffen in der Literatur vollzogen ist, dann kehren wir zurück zur ausschließlichen Verwendung der bisherigen Definitionen. Darum seien diejenigen Leser, die mit der Theorie abstrakter Datentypen weniger vertraut sind, ermuntert, diesen Abschnitt (2.6) zu überspringen, um sich nicht von anderslautenden Definitionen verwirren zu lassen. Für die eingeweihten Leser hingegen könnte dieser Abschnitt dazu beitragen, evtl. entstandene Zweifel zu beseitigen, ob wir (in Kern) von 'der gleichen Sache reden'.

Der zweite Zweck dieses Abschnittes besteht darin, die Abgrenzung unserer polymorphen Semantik zu prinzipiell monomorphen Ansätzen, insbesondere dem initialen, zu unterstreichen. Diese Unterscheidung ist zwar in der einschlägigen Literatur wohletabliert, gehört aber nach Einschätzung des Autors noch nicht zur 'Folklore'. Da aber genau dieser Punkt verantwortlich ist für das Auseinanderfallen von Erfüllbarkeit und Gültigkeit, ist er in diesem Kontext so zentral, daß wir möglichen Mißverständnissen vorbeugen wollen.

2.6.1 Allgemeine Algebren und Konstruktorterm-Algebren

Die Verallgemeinerung unserer Definition frei erzeugter Algebren zu allgemeinen Algebren erreichen wir im wesentlichen durch Ignorieren des Unterschiedes zwischen Konstruktoren und Funktionen sowie der Zulassung beliebiger Mengen als Grundmengen der Sorten. Wir können dabei ohne weiteres mit unseren ADT-Signaturen arbeiten, vereinigen aber die Konstruktoren und Funktionen zu 'Operationen' (vgl. [LEW96, Def. 2.3]).

Notation 2.6.1 Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Für jedes $s \in S$ ist $OP_s = C_s \cup F_s$, außerdem ist $\mathcal{OP}_\Sigma = \{OP_s | s \in S\}$ und $OP_\Sigma = \overline{\mathcal{OP}_\Sigma}$. Die Elemente von OP_Σ heißen **Σ-Operationen**. ⋆

Definition 2.6.2 (klassische Algebren)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Das Paar $(\mathcal{D}, \mathcal{I})$ ist eine **klassische Σ-Algebra**, falls $\mathcal{D} = \{D_s | s \in S\}$ eine S-indizierte Familie von nichtleeren Mengen ist und $\mathcal{I}$ jeder Operation l, mit $l \in OP_s$, eine Abbildung $\mathcal{I}(l)$ zuweist mit:

$$\begin{array}{ll} \mathcal{I}(l) : D_{s_1} \times \ldots \times D_{s_n} \to D_s, & \text{falls } \alpha(l) = s_1 \ldots s_n. \\ \mathcal{I}(l) : \qquad\qquad\qquad\quad \to D_s, & \text{falls } \alpha(l) = \lambda. \end{array}$$

$\mathcal{I}$ heißt **$\mathcal{OP}$-Interpretation** (im Ggs. zu $\mathcal{F}$-Interpretationen, s. Def. 2.2.23). ⋆

Wir nennen diese Algebren 'klassisch', weil sie den gängigen Definitionen entsprechen, s. [LEW96], [Wir90], [EGL89] oder [EM85]. (Um in klassischen Algebren Terme auswerten zu können, müssen wir noch in unseren Definitionen von Variablenbelegungen (Def. 2.2.25) und Termauswertungen (Def. 2.2.27) den Grundbereich CT_Σ ersetzen durch $\overline{\mathcal{D}}$. Darüberhinaus indizieren wir die Auswertefunktion val für Terme mit einer Algebra statt nur mit einer Interpretation ('$val_{\mathcal{A},\beta}$' bzw. '$val_{\mathcal{A}}$') und streichen bei deren Definition (2.2.27) die Punkte '4.' und '5.'.)

Weil wir im folgenden noch über 'isomorphe' Algebren zu sprechen haben, benötigen wir den Begriff des Homomorphismus.

Definition 2.6.3 (Homomorphismus, Isomorphie)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und seien $(\mathcal{D}, \mathcal{I})$ und $(\mathcal{D}', \mathcal{I}')$ zwei klassische Σ-Algebren mit $\mathcal{D} = \{D_s | s \in S\}$ und $\mathcal{D}' = \{D'_s | s \in S\}$.
Ein **Σ-Homomorphismus von** $(\mathcal{D}, \mathcal{I})$ **nach** $(\mathcal{D}', \mathcal{I}')$ ist eine Abbildung h von $\overline{\mathcal{D}}$ nach $\overline{\mathcal{D}'}$, so daß h auf D_s, für jedes $s \in S$, folgenden Bedingungen genügt:

1. für $d \in D_s$ ist $h(d) \in D'_s$
2. $h(\mathcal{I}(l)(d_1, \ldots, d_n)) = \mathcal{I}'(l)(h(d_1), \ldots, h(d_n))$,
   für $l \in OP_s$, $\alpha(l) = s_1 \ldots s_n$ und $\langle d_1, \ldots, d_n \rangle \in D_{s_1} \times \ldots \times D_{s_n}$

3. $h(\mathcal{I}(l)) = \mathcal{I}'(l)$,
   für $l \in OP_s$ und $\alpha(l) = \lambda$

Existiert für zwei klassische Σ-Algebren $(\mathcal{D}, \mathcal{I})$ und $(\mathcal{D}', \mathcal{I}')$ ein bijektiver Homomorphismus von $(\mathcal{D}, \mathcal{I})$ nach $(\mathcal{D}', \mathcal{I}')$, dann heißen $(\mathcal{D}, \mathcal{I})$ und $(\mathcal{D}', \mathcal{I}')$ **isomorph**, kurz '$(\mathcal{D}, \mathcal{I}) \simeq (\mathcal{D}', \mathcal{I}')$'. ⋆

Um nun von erzeugten Algebren sprechen zu können, müssen bestimmte Operationen als Konstruktoren ausgezeichnet sein. Dazu reaktivieren wir unsere Menge $\mathcal{C}$.

Definition 2.6.4 ((frei) erzeugte klassische Algebren)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $(\mathcal{D}, \mathcal{I})$ eine *klassische* Σ-Algebra.

- $(\mathcal{D}, \mathcal{I})$ ist **erzeugt durch** $\mathcal{C}$, falls für alle $d \in \overline{\mathcal{D}}$ ein Konstruktorterm $ct \in CT_\Sigma$ existiert mit $val_{(\mathcal{D},\mathcal{I})}(ct) = d$.
- $(\mathcal{D}, \mathcal{I})$ ist **frei erzeugt durch** $\mathcal{C}$, falls für alle $d \in \overline{\mathcal{D}}$ *genau ein* Konstruktorterm $ct \in CT_\Sigma$ existiert mit $val_{(\mathcal{D},\mathcal{I})}(ct) = d$. ⋆

Wird in der Literatur von erzeugten Algebren gesprochen, so geschieht dies manchmal auch ohne die Auszeichnung spezieller Operationen als Konstruktoren. Im diesem Fall sind implizit *alle* Operationen Konstruktoren.

Auch wenn der Ausdruck 'erzeugt durch' dies suggeriert, impliziert obige Definition noch nicht, daß es sich bei den Elementen des Grundbereiches um Konstruktorterme handelt. Dies ist aber der Fall bei klassischen Konstruktorterm-Algebren.

Definition 2.6.5 (Konstruktorterm-Algebren)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Eine klassische Σ-Algebra $(\mathcal{D}, \mathcal{I})$ heißt **Σ-Konstruktorterm-Algebra**, falls $\mathcal{D} = \mathcal{CT}_\Sigma$ und falls die Konstruktoren von der $\mathcal{OP}$-Interpretation $\mathcal{I}$ in folgender Weise interpretiert werden:

1. $\mathcal{I}(c)(ct_1, \ldots, ct_n) = c(ct_1, \ldots, ct_n)$,
   für $c \in \overline{\mathcal{C}}$, $\alpha(c) = s_1 \ldots s_n$ und $\langle ct_1, \ldots, ct_n \rangle \in CT_{s_1} \times \ldots \times CT_{s_n}$
2. $\mathcal{I}(c)() = c$,
   für $c \in \overline{\mathcal{C}}$ und $\alpha(c) = \lambda$ ⋆

Grob gesagt werden Konstruktoren hier durch sich selbst interpretiert, aber sie *werden* interpretiert. Dies ist der einzige Unterschied zu unserer a priori Definition frei erzeugter Algebren (Def. 2.2.31). Beides ist aber gleichwertig in dem Sinne, daß Terme identisch evaluiert werden: für $val_{\mathcal{A},\beta}(x)$ und $val_{\mathcal{A},\beta}(f(t_1, \ldots, t_n))$ besteht sowieso kein Unterschied, und sogar

$$val_{\mathcal{A},\beta}(c(t_1,\ldots,t_n)) = c(val_{\mathcal{A},\beta}(t_1),\ldots,val_{\mathcal{A},\beta}(t_n))$$

gilt in beiden Fällen, in 2.2.31 per Definition und in klassischen Konstruktorterm-Algebren als Folge der obigen Definition. Diese Abweichung im Detail hatten wir vorgenommen, um die uns interessierenden Algebren minimalistisch beschreiben zu können. Semantisch aber ist dieser Unterschied bedeutungslos. Im Grunde haben wir es also schon die ganze Zeit mit Konstruktorterm-Algebren zu tun. Warum aber nennen wir sie dann ‘frei erzeugte Algebren’, wo doch nach allgemeinener Lesart die Konstruktorterm-Algebren nur ein Spezialfall der frei erzeugten sind?

Faktum 2.6.6 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Ist $(\mathcal{D}, \mathcal{I})$ eine (klassische) Σ-Konstruktorterm-Algebra, dann ist $(\mathcal{D}, \mathcal{I})$ frei erzeugt durch $\mathcal{C}$ (i.S.v. Def. 2.6.4).*

Die Antwort lautet: in jeder Isomorphieklasse frei erzeugter, klassischer Algebren liegt *genau eine* Konstruktorterm-Algebra. Diese Algebren können daher als *Repräsentanten* dienen in dem Raum aller frei erzeugten Algebren. Der Nachweis hierfür wird im folgenden Abschnitt geführt.

2.6.2 Fehlender Isomorphieabschluß

Werfen wir nochmal einen Blick auf unsere Definition (2.2.34) frei erzeugter Datentypen: „Ein frei erzeugter Σ-Datentyp $\mathcal{DT}$ ist eine Menge frei erzeugter Σ-Algebren.“ Wir hatten bereits darauf hingewiesen, daß wir hier von den Definitionen in der Literatur abweichen. Dort wird verlangt, daß ein Datentyp gegenüber Isomorphie abgeschlossen ist, d.h. mit einer Algebra ist auch jede dazu isomorphe Element des Datentyps. Dies gilt unabhängig davon, ob man von allen Algebren eines Datentyps verlangt, daß sie wechselseitig isomorph sind (vgl. [Wir90, S. 686]) oder ob man auch nichtisomorphe Algebren in ein und demselben Datentyp zuläßt (vgl. [LEW96, Def. 2.24]). Nun betrachten wir im Gegensatz zu den zitierten Definitionen ausschließlich frei erzeugte Algebren. Dennoch stellt sich die Frage nach dem Abschluß gegen Isomorphie. Es gibt nämlich durchaus frei erzeugte klassische Algebren (s. Def. 2.6.4), die isomorph, aber nicht identisch sind.

Beispiel 2.6.7 Betrachten wir die Signatur $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ mit: $S = \{Bool\}$, $\mathcal{C} = \{\{\mathsf{tt}, \mathsf{ff}\}_{Bool}\}$, $\mathcal{F} = \{\{k\}_{Bool}\}$, $\alpha(\mathsf{tt}) = \alpha(\mathsf{ff}) = \alpha(k) = \lambda$. Betrachten wir fünf Beispiele für durch $\mathcal{C}$ frei erzeugte klassische Σ-Algebren $(\mathcal{D}_i, \mathcal{I}_i)$:
1. $\mathcal{D}_1 = \{\{H, L\}_{Bool}\}$, $\mathcal{I}_1(\mathsf{tt})() = H$, $\mathcal{I}_1(\mathsf{ff})() = L$, $\mathcal{I}_1(k)() = H$.
2. $\mathcal{D}_2 = \{\{\mathsf{tt}, \mathsf{ff}\}_{Bool}\}$, $\mathcal{I}_2(\mathsf{tt})() = \mathsf{tt}$, $\mathcal{I}_2(\mathsf{ff})() = \mathsf{ff}$, $\mathcal{I}_2(k)() = \mathsf{tt}$.
3. $\mathcal{D}_3 = \{\{H, L\}_{Bool}\}$, $\mathcal{I}_3(\mathsf{tt})() = H$, $\mathcal{I}_3(\mathsf{ff})() = L$, $\mathcal{I}_3(k)() = L$.
4. $\mathcal{D}_4 = \{\{\mathsf{tt}, \mathsf{ff}\}_{Bool}\}$, $\mathcal{I}_4(\mathsf{tt})() = \mathsf{tt}$, $\mathcal{I}_4(\mathsf{ff})() = \mathsf{ff}$, $\mathcal{I}_4(k)() = \mathsf{ff}$.

5. $\mathcal{D}_5 = \{\{\mathsf{tt}, \mathsf{ff}\}_{Bool}\}$, $\mathcal{I}_5(\mathsf{tt})() = \mathsf{ff}$, $\mathcal{I}_5(\mathsf{ff})() = \mathsf{tt}$, $\mathcal{I}_5(k)() = \mathsf{tt}$.
Alle fünf Algebren sind wechselseitig verschieden. Die Beispiele sollen dreierlei demonstrieren: I) Verschiedene durch $\mathcal{C}$ frei erzeugte klassische Σ-Algebren können isomorph sein. Zum einen sind 1. und 2. isomorph, zum anderen sind 3., 4. und 5. isomorph. II) Wegen $\mathcal{I}_5(c)() \neq c$, für $c \in \{\mathsf{tt}, \mathsf{ff}\}$, ist $(\mathcal{D}_5, \mathcal{I}_5)$ keine Konstruktorterm-Algebra, obwohl die Grundmenge $\mathcal{D}_5$ aus den Konstruktortermen besteht. III) Im Beispiel gibt es zu jeder Algebra gerade eine dazu isomorphe Konstruktorterm-Algebra. Für 1. und 2. ist dies $(\mathcal{D}_2, \mathcal{I}_2)$, für 3., 4. und 5. ist dies $(\mathcal{D}_4, \mathcal{I}_4)$. ⋆

Wir werden nun nachweisen, daß es grundsätzlich in jeder Isomorphieklasse frei erzeugter klassischer Algebren *genau eine* Konstruktorterm-Algebra gibt.

Definition 2.6.8 (Isomorphieklasse)
Sei Σ eine ADT-Signatur.
Eine **Σ-Isomorphieklasse** $\mathcal{IC}$ ist eine Menge klassischer Σ-Algebren, so daß gilt:
1. aus $\mathcal{A} \in \mathcal{IC}$ und $\mathcal{A}' \in \mathcal{IC}$ folgt $\mathcal{A} \simeq \mathcal{A}'$,
2. aus $\mathcal{A} \in \mathcal{IC}$ und $\mathcal{A} \simeq \mathcal{A}'$ folgt $\mathcal{A}' \in \mathcal{IC}$. ⋆

Faktum 2.6.9 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $\mathcal{IC}$ eine Σ-Isomorphieklasse. Enthält $\mathcal{IC}$ eine über $\mathcal{C}$ frei erzeugte Algebra, dann ist jede Algebra $\mathcal{A} \in \mathcal{IC}$ über $\mathcal{C}$ frei erzeugt.*

Demnach macht es Sinn, von 'Isomorphieklassen frei erzeugter Algebren' zu sprechen.

Satz 2.6.10 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Dann gilt:*
In jeder Σ-Isomorphieklasse über $\mathcal{C}$ frei erzeugter Algebren existiert genau eine (klassische) Σ-Konstruktorterm-Algebra.

Für den Beweis des Satzes ist die folgende Definition hilfreich:

Definition 2.6.11 Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $\mathcal{A} = (\mathcal{D}, \mathcal{I})$ eine durch $\mathcal{C}$ frei erzeugte (klassische) Σ-Algebra. Für jedes Element $d \in \overline{\mathcal{D}}$ bezeichnet $ConstrTerm_{\mathcal{A}}(d)$ den (wegen der freien Erzeugtheit eindeutigen) Konstruktorterm $ct \in CT_\Sigma$ mit $val_{\mathcal{A}}(ct) = d$. ⋆

Für die Elemente d des Grundbereiches $\overline{\mathcal{D}}$ gilt $val_{\mathcal{A}}(ConstrTerm_{\mathcal{A}}(d)) = d$ $(*)$. Für Konstruktorterme $ct \in CT_\Sigma$ gilt $ConstrTerm_{\mathcal{A}}(val_{\mathcal{A}}(ct)) = ct$ $(**)$. (Die Eigenschaft $(**)$ gilt nicht für Grundterme, die Funktionen enthalten.) Beide Eigenschaften benutzen wir im folgenden Beweis des obigen Satzes.

Beweis: (des Satzes 2.6.10) Der Satz ergibt sich aus zwei Unterbehauptungen:

1. Sei $\mathcal{A}$ eine über $\mathcal{C}$ frei erzeugte Algebra. Dann *existiert* eine zu $\mathcal{A}$ isomorphe Σ-Konstruktorterm-Algebra.

2. Seien $\mathcal{A}$ und $\mathcal{B}$ zwei isomorphe Σ-Konstruktorterm-Algebren. Dann sind $\mathcal{A}$ und $\mathcal{B}$ identisch.

Zu '1.': Sei $\mathcal{A} = (\mathcal{D}, \mathcal{I})$ eine durch $\mathcal{C}$ frei erzeugte Σ-Algebra. Wir konstruieren eine zu $\mathcal{A}$ isomorphe Σ-Konstruktorterm-Algebra $\mathcal{B} = (\mathcal{CT}_\Sigma, \mathcal{I}_\mathcal{B})$. Dazu definieren wir für $l \in OP_\Sigma$: $\mathcal{I}_\mathcal{B}(l)(ct_1, \ldots, ct_n) := \mathit{ConstrTerm}_\mathcal{A}(\mathit{val}_\mathcal{A}(l(ct_1, \ldots, ct_n)))$, falls $\alpha(l) = s_1 \ldots s_n$ und $\langle ct_1, \ldots, ct_n\rangle \in CT_{s_1} \times \ldots \times CT_{s_n}$, sowie $\mathcal{I}_\mathcal{B}(l)() := \mathit{ConstrTerm}_\mathcal{A}(\mathit{val}_\mathcal{A}(l))$, falls $\alpha(c) = \lambda$. $\mathcal{B}$ ist tatsächlich eine Σ-Konstruktorterm-Algebra, da für $c \in \overline{\mathcal{C}}$: $\mathcal{I}_\mathcal{B}(c)(ct_1, \ldots, ct_n) = \mathit{ConstrTerm}_\mathcal{A}(\mathit{val}_\mathcal{A}(c(ct_1, \ldots, ct_n))) \stackrel{**}{=} c(ct_1, \ldots, ct_n)$, falls $\alpha(c) = s_1 \ldots s_n$. Für Basiskonstruktoren gilt entsprechend: $\mathcal{I}_\mathcal{B}(c)() = c$. $\mathcal{A}$ und $\mathcal{B}$ sind isomorph, da die bijektive Funktion $\mathit{ConstrTerm}_\mathcal{A}$ ein Homomorphismus von $\mathcal{A}$ nach $\mathcal{B}$ ist. Dazu zeigen wir $h(\mathcal{I}_\mathcal{A}(l)(d_1, \ldots, d_n)) = \mathcal{I}_\mathcal{B}(l)(h(d_1), \ldots, h(d_n))$ für $h := \mathit{ConstrTerm}_\mathcal{A}$.

$$\begin{aligned}
&\mathcal{I}_\mathcal{B}(l)(h(d_1), \ldots, h(d_n)) \\
&= \mathit{ConstrTerm}_\mathcal{A}(\mathit{val}_\mathcal{A}(l(h(d_1), \ldots, h(d_n)))) \\
&= \mathit{ConstrTerm}_\mathcal{A}(\mathcal{I}_\mathcal{A}(l)(\mathit{val}_\mathcal{A}(h(d_1)), \ldots, \mathit{val}_\mathcal{A}(h(d_n)))) \\
&\stackrel{*}{=} \mathit{ConstrTerm}_\mathcal{A}(\mathcal{I}_\mathcal{A}(l)(d_1, \ldots, d_n)) \\
&= h(\mathcal{I}_\mathcal{A}(l)(d_1, \ldots, d_n))
\end{aligned}$$

Für nullstellige Operationen ergibt sich $\mathcal{I}_\mathcal{B}(l)() = h(\mathcal{I}_\mathcal{A}(l)())$ entsprechend.

Zu '2.': Aus der Def. 2.6.5 wissen wir bereits, daß $\mathcal{D}_\mathcal{A} = \mathcal{D}_\mathcal{B} = \mathcal{CT}_\Sigma$. Außerdem wissen wir für $c \in \overline{\mathcal{C}}$: $\mathcal{I}_\mathcal{A}(c)(ct_1, \ldots, ct_n) = \mathcal{I}_\mathcal{B}(c)(ct_1, \ldots, ct_n) = c(ct_1, \ldots, ct_n)$, falls $\alpha(c) = s_1 \ldots s_n$, bzw. $\mathcal{I}_\mathcal{A}(c)() = \mathcal{I}_\mathcal{B}(c)() = c$, falls $\alpha(c) = \lambda$ ($\blacklozenge$). Da $\mathcal{A}$ und $\mathcal{B}$ isomorph sind, existiert ein bijektiver Homomorphismus h von $\mathcal{A}$ nach $\mathcal{B}$, es gilt also für $l \in OP_\Sigma$: $h(\mathcal{I}_\mathcal{A}(l)(d_1, \ldots, d_n)) = \mathcal{I}_\mathcal{B}(l)(h(d_1), \ldots, h(d_n))$ bzw. $h(\mathcal{I}_\mathcal{A}(l)()) = \mathcal{I}_\mathcal{B}(l)()$ ($\blacktriangledown$). Wenn wir nachweisen, daß h die Identität ist, dann folgt aus ($\blacktriangledown$), daß $\mathcal{I}_\mathcal{A}(l)(d_1, \ldots, d_n) = \mathcal{I}_\mathcal{B}(l)(d_1, \ldots, d_n)$ bzw. $\mathcal{I}_\mathcal{A}(l)() = \mathcal{I}_\mathcal{B}(l)()$, und wir sind fertig. Wir zeigen mit struktureller Induktion über die Konstruktorterme, daß $h : \overline{\mathcal{CT}_\Sigma} \to \overline{\mathcal{CT}_\Sigma}$ die Identität ist. Der Basisfall: wenn $\alpha(c) = \lambda$, dann gilt: $h(c) \stackrel{\blacklozenge}{=} h(\mathcal{I}_\mathcal{A}(c)()) \stackrel{\blacktriangledown}{=} \mathcal{I}_\mathcal{B}(c)() \stackrel{\blacklozenge}{=} c$. Der Induktionsschritt: $h(c(ct_1, \ldots, ct_n)) \stackrel{\blacklozenge}{=} h(\mathcal{I}_\mathcal{A}(c)(ct_1, \ldots, ct_n)) \stackrel{\blacktriangledown}{=} \mathcal{I}_\mathcal{B}(c)(h(ct_1), \ldots, h(ct_n)) \stackrel{\mathit{IndHyp}}{=} \mathcal{I}_\mathcal{B}(c)(ct_1, \ldots, ct_n) \stackrel{\blacklozenge}{=} c(ct_1, \ldots, ct_n)$. □

Da also jede Isomorphieklasse klassischer, frei erzeugter Algebren genau eine Σ-Konstruktorterm-Algebra enthält, kann letztere (auf der Metaebene) als 'Repräsentant' dienen für die ganze Klasse. Insbesondere reduziert sich die Frage nach der Erfüllbarkeit einer Spezifikation auf die Frage nach der Existenz einer erfüllenden Konstruktorterm-Algebra! Wir hätten diesen Sachverhalt als Suchraumeinschränkung im Raum der frei erzeugten klassischen Algebren modellieren können. Stattdessen haben wir, nicht-klassisch, frei erzeugte Algebren per Definition (2.2.31) eingeschränkt auf eine leichte Variation klassischer Konstruktorterm-Algebren. Daß wir dabei im Grunde den gleichen Erfüllbarkeitsbegriff und den

gleichen Folgerbarkeitsbegriff erhalten wie mit klassischen Definitionen, das folgt aus Satz 2.6.10. Dies ist natürlich zentral für die Anwendbarkeit der hier beschriebenen Methodik.

Weiterhin folgt aus Satz 2.6.10, daß ein Abschluß gegen Isomorphie in unserer Definition von Datentypen nichts anderes wäre als ein Abschluß gegen Identität.

2.6.3 Lose Semantik vs. monomorphe Ansätze

Ein zweites Mal wiederholen wir die Definition (2.2.34) frei erzeugter Datentypen: „Ein frei erzeugter Σ-Datentyp $\mathcal{DT}$ ist eine Menge frei erzeugter Σ-Algebren.“ Sie läßt in ein und demselben Datentyp verschiedene (nicht-isomorphe) Algebren zu. Dies ist in der ADT-Literatur nicht ungewöhnlich. Zwar werden Datentpyen teilweise auch als *eine einzelne* Isomorphieklasse von Algebren definiert ([Wir90, S. 686]). Diese Datentypen würden wir als 'monomorph' bezeichnen. Aber ein wichtiger Teil der Literatur läßt auch polymorphe abstrakte Datentypen zu ([EM85, S. 34, 2.], [EGL89, Def. 2.13], [LEW96, Def. 2.24]). Wichtiger als dieser Unterschied ist die Frage, ob man als Semantik einer Spezifikation verschiedene, nicht-isomorphe Algebren zulassen will (egal, ob man diese Algebren dann als einen oder als mehrere Datentypen auffaßt). Die Zulassung nicht-isomorpher Algebren[20] als Semantik einer Spezifikation wird im Gebiet der abstrakten Datentypen meist als *lose Semantik* (s. [Gut75]) bezeichnet. Dem gegenüber stehen zwei Ansätze, welche die Semantik von Spezifikationen auf einzelne Isomorphieklassen von Algebren beschränken. Konkret handelt es sich hierbei um die *initiale Semantik* und um *konstruktive Spezifikationen*. Beide bezeichnen wir hier (etwas lax) als *monomorphe Spezifikationsmethoden*.

Bei dem *initialen Ansatz* (s. [GTWW75]) wird die Monomorphie mit semantischen Mitteln erreicht. Dazu betrachtet man als Semantik einer Spezifikation nur diejenigen ihrer Modelle, die im Raum aller Modelle 'initial' sind, das heißt, von denen aus eindeutige Homomorphismen zu allen anderen Modellen (der Spezifikation) existieren. Diese algebraische Charakterisierung hat eine schlichte inituitive Deutung: im initialen Modell einer Spezifikation gelten genau die Gleichungen, die aus der Spezifikation folgen. Oder anders gesagt, je zwei Elemente des Grundbereiches sind 'unequal by default'. Die initialen Modelle einer Spezifikation bilden eine Isomorphieklasse. Ähnlich wie im 'frei erzeugten' Fall, wo in jeder Isomorphieklasse genau eine Konstruktortermalgebra liegt, gibt es auch hier einen eindeutigen Repräsentanten der Isomorphieklasse, und das ist der Quotient der Termalgebra gegenüber der (folgerbaren) Gleichheit, anderswo auch als 'minimales Herbrand-Modell' bezeichnet (weil die Äquivalenzklassen minimal sind). Man spricht hier auch von 'dem' initialen Modell. Beschränkt man sich in den Axiomen auf *Horn Gleichungen* (s. S. 19) oder gar auf *reine Gleichungen*, dann ist

[20] dennoch mit der Einschränkung auf termerzeugte Algebren

die Existenz initialer Modelle garantiert. Im Sinne ihrer initialen Bedeutung sind Horn- und reine Gleichungsspezifikationen sogar ausführbar, was nur möglich ist, weil jedes Detail festliegt (Monomorphie). Lassen die Axiome für sich genommen mehrere Möglichkeiten zu, so wählt die Initialität in eindeutiger Weise eine davon aus. Diese muß nicht die intendierte sein:

Beispiel 2.6.12 Die Spezifikation Nat (Abb. 2.2, S. 25) besitzt als einziges Axiom die Gleichung: $\mathit{pred}(\mathsf{succ}(x)) \doteq x$. Wir betrachten das initiale Modell (die minimale Herbrandalgebra) von Nat. Im initialen Ansatz gibt es keine Konstruktoren, wir behandeln daher zero, succ und *pred* als gleichwertige 'Operationen'. Man beachte, daß für *keinen* Term t, der nur aus zero und succ aufgebaut ist, die Gleichung $\mathit{pred}(\mathsf{zero}) \doteq t$ folgt. Darum bezeichnet $\mathit{pred}(\mathsf{zero})$ im initialen Modell ein von Null und seinen Nachfolgern verschiedenes Element. Das initiale Modell besteht genau aus den folgenden Äquivalenzklassen:
$[\mathsf{zero}],\ [\mathsf{succ}(\mathsf{zero})],\ [\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))],\ \ldots\ldots$
$[\mathit{pred}(\mathsf{zero})],\ [\mathsf{succ}(\mathit{pred}(\mathsf{zero}))],\ [\mathsf{succ}(\mathsf{succ}(\mathit{pred}(\mathsf{zero})))],\ \ldots\ldots$
$[\mathit{pred}(\mathit{pred}(\mathsf{zero}))],\ [\mathsf{succ}(\mathit{pred}(\mathit{pred}(\mathsf{zero})))],\ [\mathsf{succ}(\mathsf{succ}(\mathit{pred}(\mathit{pred}(\mathsf{zero}))))],\ \ldots$
$$\vdots$$
$$\vdots$$
Im Gegensatz hierzu wird in dem *frei erzeugten* Datentyp, der durch Nat spezifiziert wird, die Bedeutung von $\mathit{pred}(\mathsf{zero})$ nicht festgelegt (lose Semantik). Auf jeden Fall aber ist $\mathit{pred}(\mathsf{zero})$ gleich einem Term, der nur aus zero und succ aufgebaut wird. ⋆

Dieses Beispiel soll auch veranschaulichen, daß die Verwendung loser Semantik *nicht hauptsächlich* dadurch motiviert ist, daß man in den Axiomen syntaktisch quantorenfreie oder volle erststufige Gleichungslogik (s. S. 19) benutzen kann. Zwar sind die unbeschränkte Disjunktion und der Existenzquantor willkommene syntaktische Zutaten, aber auch bei reinen Gleichungs-Axiomen kann man durchaus die lose gegenüber der initialen Semantik bevorzugen, etwa in dem obigen Beispiel. Die Möglichkeit der Unterspezifikation kann sehr hilfreich sein, je nach Anwendung der Spezifikationsmethodik. *„In software development, design specifications and prototyping by executable specifications are supported by the initial approach; in the loose approach the aim is to cover the whole software development process including requirement specifications." M. Wirsing [Wir95, S. 84].* In der Entwicklung algebraischer Spezifikationssprachen seit den späten siebziger Jahren ist ein Trend zu verzeichnen von anfangs reinen 'initialen Spezifikationssprachen' hin zu 'losen Spezifikationssprachen' bzw. zu Sprachen, die beide Paradigmen unterstützen (vgl. [Wir95, Table 1]). Das vorläufige Ende dieser Entwicklung ist die schon erwähnte, quasi-standardisierte Sprache 'CASL' [CoF98].

Wenn man sich einmal entschieden hat für die Verwendung loser (termerzeugter) Semantik, dann ist es sinnvoll, volle erststufige Logik zu erlauben. *„For specifying*

requirements it soon became apparent that full first-order logic (with equality as only predicate symbol) was suitable [BDP+79].“ M. Wirsing [Wir95, S. 89]. Was den Existenzquantor betrifft, so haben wir in Abschnitt 2.5 ausführlich diskutiert, inwieweit er praktisch ‘austauschbar’ ist mit Skolemfunktionen. (Im übrigen könnten wir in unserem Zusammenhang sowieso nicht auf Existenzquantoren verzichten, da die uns interessierenden Gegenspezifikationen den Existenzabschluß einer negierten Vermutung enthalten.) Was die Zulassung der unbeschränkten Disjunktion betrifft, so zeigt es sich, daß sie keine Schwierigkeiten bereitet, die man nicht sowieso schon hat. Aufgrund der Termerzeugtheit müssen wir nämlich implizit sogar mit unendlichen Disjunktionen kämpfen: so ist beispielsweise die Termerzeugtheit der Modelle von Nat (Abb. 2.2, S. 25) äquivalent zu der folgenden unendlichen Disjunktion:

$$x \doteq \mathsf{zero} \;\vee\; x \doteq \mathsf{succ}(\mathsf{zero}) \;\vee\; x \doteq \mathsf{succ}(\mathsf{succ}(\mathsf{zero})) \;\vee\; \ldots\ldots$$

Auch wenn diese Disjunktion in der Semantik versteckt ist, so wird sie doch in der noch darzustellenden Methodik der Modellsuche sichtbar werden. Sie verursacht dort mehr Schwierigkeiten als die aus den Axiomen stammenden Disjunktionen.

Ein anderer Ansatz, welcher die Monomorphie erzwingt, aber im Gegensatz zum initialen Ansatz die erwünschten, durch Konstruktoren erzeugten Grundbereiche garantiert, besteht in der Befolgung bestimmter Muster für die Axiome. Die Bezeichnung solcher Muster ist uneinheitlich, wenn sie sich auch ähneln. Eine Referenzdefinition findet man in [LEW96, Def. 8.1], wo die entprechenden Spezifikationen bezeichnet werden als ‘*konstruktive Spezifikationen*’. (Wir verwenden den Begriff auch für verwandte Konzepte.) Ohne die Definition wiederzugeben, nennen wir nur die wesentlichen Eigenschaften: alle Axiome sind Gleichungen; jede linke Seite der Gleichung wird von einer (Nicht-Konstruktor-)Funktion angeführt; die gerichteten Gleichungen genügen einer Reduktionsordnung und sind wertebeschränkt ($Var(t') \subseteq Var(t)$ für $t \doteq t'$); schließlich ist jeder Grundterm, der nicht durch einen Konstruktor angeführt wird, Instanz *genau einer* linken Seite einer Gleichung. Andere Definitionen, die auch bedingte Gleichungen zulassen, erlauben die Verletzung der letztgenannten Restriktion, verlangen dann aber die semantische Disjunktheit der entsprechenden Bedingungen. Konstruktive Spezifikationen lassen sich auffassen als *deterministische* Programme, die eine Abbildung aller Grundterme auf Konstruktorterme berechnen. In unserem Zusammenhang sind konstruktive Ansätze deswegen erwähnenswert, weil eine Reihe von Arbeiten zur Aufdeckung und Korrektur falscher Vermutungen diesem Bereich zuzuorden sind.

Generell unterscheiden sich monomorphe Ansätze in Fragen der Erfüllbarkeit von Spezifikationen ganz wesentlich von dem polymorphen Ansatz der losen Semantik. Im monomorphen Kontext fallen Erfüllbarkeit und Gültigkeit zusammen! Hierauf gründen sich u.a. Methoden, die man unter dem Begriff ‘*Beweis durch Konsistenz*’ zusammenfaßt, vgl. [Com01][21]. In gleicher Weise fällt im monomor-

[21] Der zitierte Handbuch-Artikel heißt zwar „Inductionless Induction“, darin findet sich aber

phen Kontext die Nicht-Folgerbarkeit einer Formel zusammen mit der Folgerbarkeit ihres Gegenteils (vgl. Faktum 2.4.14). Darum kann dort die Fragestellung der Nicht-Folgerbarkeit mit klassischen Beweisverfahren behandelt werden.

In unserem prinzipiell polymorphen Kontext hingegen ist das Erfüllbarkeitsproblem nicht mit klassischen Beweismethoden zu lösen. Stattdessen werden wir nach einzelnen Modellen (von Gegenspezifikationen) suchen.

der Satz: „Inductionless Induction should rather be called *proof by consistency*“.

3 Modell-Generierung

Im letzten Kapitel wurde erläutert, wie man die Frage der Nicht-Folgerbarkeit (gilt '$\langle \Sigma, AX \rangle \not\models \varphi$'?) reduziert auf die Frage nach der Erfüllbarkeit einer normalisierten Gegenspezifikation, vgl. Korollar 2.5.19. Nun wenden wir uns der Frage zu, wie man allgemein die Erfüllbarkeit einer normalisierten Spezifikation frei erzeugter Datentypen untersuchen kann. Da Erfüllbarkeit gerade die Existenz eines Modells bedeutet, besteht die Aufgabe darin, ein Modell der Spezifikation zu konstruieren.

Die unterschiedlichen Ansätze in dem noch jungen Gebiet der Modell-Konstruktion lassen sich nach A. Leitsch [Lei00] in *semantische* und *syntaktische* Methoden einteilen. Charakteristisch für die *semantischen* Methoden ist die Aufzählung von Interpretationen. Dabei werden Interpretationen auf ähnlich elementare Weise dargestellt wie etwa Wahrheitstabellen in der Aussagelogik. Derartige Darstellungen unterliegen zwar am Ende auch irgendeiner Syntax, aber diese übernimmt gegenüber der Objektlogik eher die Rolle einer Metasprache und dient der sehr unmittelbaren semantischen Beschreibung eines Modells. Bei den *syntaktischen* Methoden hingegen ist die konstruierte Repräsentation eines Modells wiederum eine Formel der Objektlogik (oder eine endliche Menge solcher Formeln). Bei der Suche nach diesen Formeln bedienen sich die syntaktischen Methoden (Varianten) bekannter Logikkalküle.

Wir hatten im letzten Kapitel erläutert, daß frei erzeugte Algebren[1] $(\mathcal{CT}_\Sigma, \mathcal{I})$ eindeutig gegeben sind durch ihre Interpretation $\mathcal{I}$. Unser Ziel ist nun die *Konstruktion einer Interpretation* der Funktionen über dem Grundbereich der Konstruktorterme, und zwar in der Weise, daß die jeweilige Spezifikation erfüllt wird. Die im folgenden beschriebene Methode stellt Interpretationen auf eine sehr unmittelbare Weise dar und zählt darum vom Grundansatz her zu den semantischen Methoden im obigen Sinne. Gleichzeitig hat sie auch Anteile einer syntaktischen Methode, da die Interpretationen wiederum mittels logischer, atomarer Formeln dargestellt werden und die Suche nach ihnen in deduktiver Weise vonstatten geht, unter Zuhilfenahme eines logischen Kalküls. Dennoch muß hier deutlich zwischen

[1] Hier und im folgenden verwenden wir die semantischen Begriffe wieder ausschließlich im Sinne der Definitionen in den Abschnitten 2.2 bis 2.5. Mit 'frei erzeugte Algebren' beispielsweise ist der in Def. 2.2.31 definierte Begriff gemeint, nicht etwa die mit Absicht etwas anders genannten 'frei erzeugten *klassischen* Algebren' aus Abschnitt 2.6.

Interpretationstabelle für f:

$CT_{s_1} \times \ldots \times CT_{s_n}$	$\mathcal{I}(f)$
$\langle ct_{11}, \ldots, ct_{1n} \rangle$	ct_{10}
$\langle ct_{21}, \ldots, ct_{2n} \rangle$	ct_{20}
$\vdots$	$\vdots$

Interpretationstabelle für f':

$CT_{s'_1} \times \ldots \times CT_{s'_m}$	$\mathcal{I}(f')$
$\langle ct'_{11}, \ldots, ct'_{1m} \rangle$	ct'_{10}
$\langle ct'_{21}, \ldots, ct'_{2m} \rangle$	ct'_{20}
$\vdots$	$\vdots$

Abbildung 3.1: Interpretation $\mathcal{I}$ als Tabellen

den Ebenen unterschieden werden: die Logik, 'innerhalb derer' Interpretationen gesucht und repräsentiert werden, 'redet' explizit über Interpretationen, nicht über Gleichungen, wie das die Axiome der Spezifikation tun. Umgekehrt kommen in den Axiomen die Interpretationen nicht explizit vor.

3.1 Interpretations-Schließen

Zwei frei erzeugte Algebren über der gleichen Signatur $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ können sich nur in ihrer $\mathcal{F}$-Interpretation unterscheiden. Darum reduziert sich die Konstruktion eines Modells auf die Konstruktion einer Interpretation der Funktionen. Wir können uns eine $\mathcal{F}$-Interpretation vorstellen als Menge (i.a. unendlicher) Tabellen, jeweils eine für jede Funktion $f \in \overline{\mathcal{F}}$. Diese Sichtweise ist dargestellt in Abb. 3.1, am Beispiel einer Funktionenmenge $\overline{\mathcal{F}} = \{f, f'\}$ mit $\alpha(f) = s_1 \ldots s_n$ und $\alpha(f') = {s'}_1 \ldots {s'}_n$. In der ersten Spalte jeder Tabelle werden alle Tupel von Konstruktortermen der entsprechenden Sorten aufgezählt. Das geht, weil die Konstruktorterme einer Sorte aufzählbar sind und das Kreuzprodukt aufzählbarer Mengen wieder aufzählbar ist. Jedem Tupel $\langle ct_1, \ldots, ct_n \rangle$ der ersten Spalte wird in der zweiten Spalte die Anwendung $\mathcal{I}(g)(ct_1, \ldots, ct_n)$ der Interpretation der jeweiligen Funktion g auf seine Komponenten zugeordnet. (Dabei muß natürlich die Zielsorte der Funktion g in keiner Weise durch die zweite Spalte abgedeckt werden; so könnte z.B. eine konstante Interpretation in der rechten Spalte überall den gleichen Eintrag besitzen.)

Die Grundidee hinter dem im folgenden zu beschreibenden Vorgehen besteht darin, die *Zeilen dieser Interpretationstabellen* sowie *Mengen solcher Zeilen* zu verwenden als Grundbestandteile der Repräsentation von Interpretationen. Auf

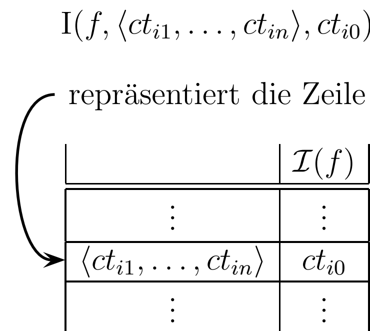


Abbildung 3.2: Zeilen der Interpretationstabelle als Atome

dieser Repräsentation wird die Suche nach (erfüllenden) Interpretationen durchgeführt. Im speziellen werden wir die Zeilen der Interpretationstabellen als *Atome* repräsentieren, gebildet mit dem dreistelligen Prädikat I, entspechend der Abbildung 3.2. Wir nennen solche Atome 'I-Atome'.

Es ist wichtig, festzustellen, daß diese Atome nicht Teil unserer Objekt-Logik sind, schon allein deswegen, weil wir gar keine Signaturen mit Prädikaten betrachten, 'I' also kein Bestandteil einer ADT-Signatur sein kann. I-Atome können weder in einer ADT-Spezifikation noch in einer Vermutung φ über diese Spezifikation verwendet werden. Da der nun zu beschreibende Mechanismus nach Interpretationen sucht, begeben wir uns in der Repräsentation der Interpretationen notwendigerweise auf die *Meta-Ebene*. Andererseits kommen wir jetzt nicht mehr, wie noch im vorangegangenen Kapitel, mit der Verwendung mathematischer Meta-Sprache aus. Wir benötigen syntaktische Objekte, auf denen der Mechanismus operieren kann. Insbesondere ist der Basis-Mechanismus, auf den wir uns stützen, ein *deduktiver*, wie der Titel dieser Abhandlung verrät. Das bedeutet, er operiert auf Formeln. Diese Formeln sind aber (aus Sicht unserer in Abschn. 2.3.1 definierten Objekt-Syntax) *Meta-Formeln*. Es besteht jedoch keine Notwendigkeit, auch auf der Meta-Ebene eine volle Formelsyntax mit Junktoren und Quantoren aufzubauen. Wir benötigen nur Atome. Darum werden wir den Terminus 'Meta-Formeln' fortan nicht mehr verwenden und nur noch von 'Meta-Atomen' sprechen. $I(g, \langle ct_1, \ldots, ct_n \rangle, ct_0)$ ist ein Beispiel für ein solches Meta-Atom. Außer den genannten I-Atomen brauchen wir noch einige andere, deren Rolle im folgenden noch dargelegt wird.

Jede *Menge* solcher I-Atome repräsentiert einen Ausschnitt einer Interpretation der Funktionen, solange die Menge *funktional* ist, das heißt, solange sie *nicht* zwei I-Atome der Form $I(g, \langle ct_1, \ldots, ct_n \rangle, ct_0)$ und $I(g, \langle ct_1, \ldots, ct_n \rangle, ct'_0)$ mit *unterschiedlichen* Konstruktortermen ct_0 und ct'_0 enthält. Eine beliebige Menge von I-Atomen nennen wir einen *Interpretations-Kandidaten*, jedenfalls solange die Funktionalität nicht sichergestellt ist. Die Suche nach Interpretationen besteht

im Kern aus dem Aufbau von Interpretations-Kandidaten durch Inferenz neuer I-Atome.

Dabei haben die inferierten I-Atome nicht immer von Anfang an die oben angedeutete Form. Zunächst können einzelne Einträge der Interpretationstabelle noch unbekannt sein. Nach ihnen muß erst noch gesucht werden. Bis dahin werden sie in den I-Atomen durch Platzhalter repräsentiert, die später ersetzt werden. Als einfaches Beispiel betrachten wir eine einstellige Funktion f von *Nat* nach *Nat* (eine Spezifikation der Sorte *Nat* findet sich auf S. 25). Es kann zum Beispiel notwendig sein, nach dem Wert der Interpretation von f an der Stelle *zero*, d.h. nach $\mathcal{I}(f)(zero)$ zu suchen. Der Mechanismus würde dann ein I-Atom der Form I(f, $\langle zero\rangle$, ?) erzeugen, wobei das letzte Argument nicht ein Fragezeichen sein wird sondern etwas, das nun erklärt werden soll. Solange der Wert von $\mathcal{I}(f)(zero)$ noch nicht bekannt ist, wissen wir nur, daß es sich hierbei um den gleichen Wert handeln muß wie bei $val_{\mathcal{I}}(f(zero))$ (vgl. die Def. 2.2.27 der Termauswertung). Der Ausdruck '$val_{\mathcal{I}}(f(zero))$' könnte also ein geeigneter Kandidat sein, den noch unbekannten Wert zunächst einmal zu repräsentieren. Um die Stelle des Fragezeichens einzunehmen, brauchen wir einen konkreten syntaktischen Ausdruck, sozusagen einen *Meta-Term*, der dem mathematisch-metasprachlichen Ausdruck '$val_{\mathcal{I}}(f(zero))$' entspricht. Dieser Meta-Term hat (im Beispiel) die Syntax $\mathsf{val}(f(zero))$. Das obige I-Atom hat also zu einem Zeitpunkt, in dem der gesuchte Wert noch gänzlich unbekannt ist, die Form I(f, $\langle zero\rangle$, $\mathsf{val}(f(zero))$).

Das Meta-Atom I(f, $\langle zero\rangle$, $\mathsf{val}(f(zero))$) hat natürlich für sich genommen noch einen sehr geringen Informationsgehalt. Schließlich ist es ja auch nicht als Ergebnis, sondern als Ausgangspunkt gedacht. Die Suche nach einem Konstruktorterm, der schließlich $\mathsf{val}(f(zero))$ ersetzen soll, wird initialisiert durch die Erzeugung eines zusätzlichen Meta-Atoms $\mathsf{search_}Nat(\mathsf{val}(f(zero)))$[2]. Intuitiv besagt dieses, daß nun nach einem Konstruktorterm der Sorte *Nat* gesucht werden soll, der die Stelle von $\mathsf{val}(f(zero))$ einnimmt. Nun wird durch das $\mathsf{search_}Nat$-Atom eine *Verzweigung* des aktuellen Interpretations-Kandidaten bewirkt, und zwar in genauso viele Äste, wie die Sorte *Nat* Konstruktoren hat, in diesem Falle also in zwei. Entweder der Wert von $\mathsf{val}(f(zero))$ ist *zero*, was wir ausdrücken durch das Meta-Atom $\mathsf{is}(\mathsf{val}(f(zero)),\ zero)$. Oder aber der gesuchte Wert ist der Nachfolger *succ* einer noch näher zu bestimmenden Zahl aus *Nat*. In letzterem Fall wird ein Meta-Atom der Form $\mathsf{is}(\mathsf{val}(f(zero)),\ succ(?))$ erzeugt, wobei wir wiederum sofort klären wollen, was an die Stelle des Fragezeichens treten soll.

Schauen wir uns das unvollständige Meta-Atom $\mathsf{is}(\mathsf{val}(f(zero)),\ succ(?))$ genauer an. Es steht dafür, daß $val_{\mathcal{I}}(f(zero)) = succ(t)$ ist, für irgend ein t. Da t das erste (und in diesem Fall einzige) Argument von $succ(t)$ ist, ist t auch das erste Argument von $val_{\mathcal{I}}(f(zero))$! Gemäß der Def. 2.2.13 gilt also: $t =$

[2]Die Verwendung unterschiedlicher Schriftarten für '*Nat*' und '$f(zero)$' einerseits bzw. für 'search' und 'val' andererseits soll andeuten, welche Bestandteile der Syntax aus der ADT-Signatur stammen, und welche für die Meta-Ebene stehen.

$val_{\mathcal{I}}(f(zero))\downarrow_1$. Das bedeutet: an die Stelle des Fragezeichens tritt ein Meta-Term, der für $val_{\mathcal{I}}(f(zero))\downarrow_1$ steht. Dieser hat die Syntax $\mathsf{arg1}(\mathsf{val}(f(zero)))$. Nunmehr können wir das eben noch unvollständige Meta-Atom komplett angeben: $\mathsf{is}(\mathsf{val}(f(zero)),\ succ(\mathsf{arg1}(\mathsf{val}(f(zero)))))$. Schließlich muß in diesem zweiten Fall der Verzweigung noch ein zweites Meta-Atom hinzugefügt werden, welches 'besagt', daß nun nach dem neuen Meta-Term $\mathsf{arg1}(\mathsf{val}(f(zero)))$ gesucht werden muß: $\mathsf{search_}Nat(\mathsf{arg1}(\mathsf{val}(f(zero))))$. (Man sieht, daß die Meta-Terme und Meta-Atome schnell unübersichtlich werden. Dies ist aber akzeptabel, da das Verfahren vollautomatisch arbeitet. Immerhin sind die Einzelschritte recht leicht zu verstehen, da die Syntax sich stark an den mathematisch-metasprachlichen Konzepten orientiert.) Zusammengenommen ergibt sich bisher das folgende Bild:

$$\mathrm{I}(f,\ \langle zero\rangle,\ \mathsf{val}(f(zero)))$$
$$\mathsf{search_}Nat(\mathsf{val}(f(zero)))$$
$$\mathsf{is}(\mathsf{val}(f(zero)),\ zero) \qquad \mathsf{is}(\mathsf{val}(f(zero)),\ succ(\mathsf{arg1}(\mathsf{val}(f(zero)))))$$
$$\mathsf{search_}Nat(\mathsf{arg1}(\mathsf{val}(f(zero))))$$

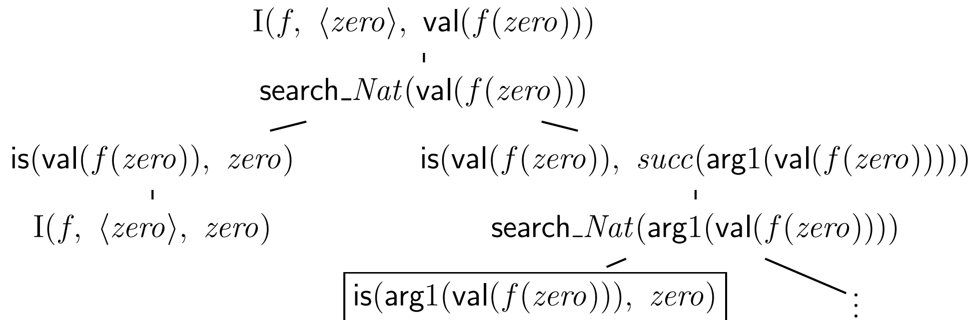
Auf der linken Seite der Verzweigung wissen wir nun schon, durch welchen Konstruktorterm $\mathsf{val}(f(zero))$ ersetzt werden soll, nämlich durch *zero*. Dieses Wissen wollen wir nun für das oberste Meta-Atom, $\mathrm{I}(f,\ \langle zero\rangle,\ \mathsf{val}(f(zero)))$, nutzen. Aus diesem und aus $\mathsf{is}(\mathsf{val}(f(zero)),\ zero)$ kann man auf dem linken Zweig das neue Meta-Atom $\mathrm{I}(f,\ \langle zero\rangle,\ zero)$ inferieren:

$$\mathrm{I}(f,\ \langle zero\rangle,\ \mathsf{val}(f(zero)))$$
$$\mathsf{search_}Nat(\mathsf{val}(f(zero)))$$
$$\mathsf{is}(\mathsf{val}(f(zero)),\ zero) \qquad \mathsf{is}(\mathsf{val}(f(zero)),\ succ(\mathsf{arg1}(\mathsf{val}(f(zero)))))$$
$$\boxed{\mathrm{I}(f,\ \langle zero\rangle,\ zero)} \qquad \mathsf{search_}Nat(\mathsf{arg1}(\mathsf{val}(f(zero))))$$

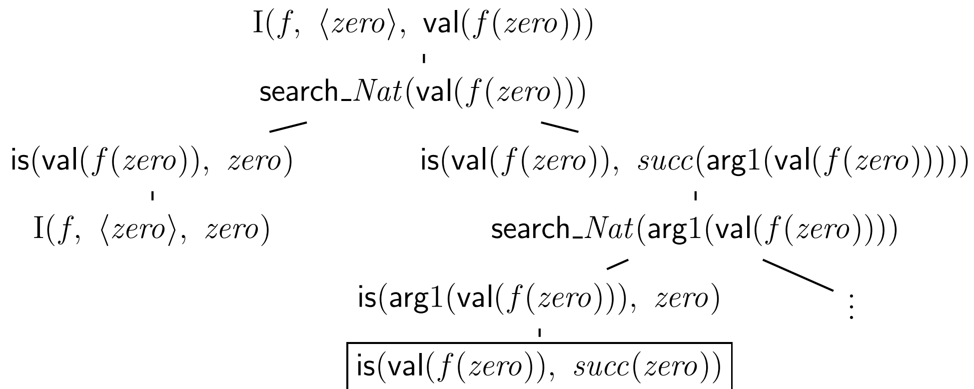
Die Meta-Terme an der Wurzel und im linken Blatt unterscheiden sich nur in ihrem letzten Argument. Die beiden dort auftretenden Terme, $\mathsf{val}(f(zero))$ und *zero*, fassen wir beide auf als Meta-Terme. Ganz allgemein bilden die Konstruktorterme die Schnittmenge zwischen normalen Termen und Meta-Termen. Dies entspricht auch völlig den im letzten Kapitel ausführlich diskutierten syntaktischen und semantischen Gegebenheiten, wonach die Konstruktorterme sowohl Terme der Objektsprache sind als auch die Elemente der semantischen Bereiche darstellen.

Fahren wir fort mit dem Aufbau eines Baumes von alternativen Interpretations-Kandidaten. Aus dem is-Atom auf der *rechten* Seite der Verzweigung können wir nur ablesen, daß der gesuchte Konstruktorterm von der Form $succ(\cdot)$ ist. Den-

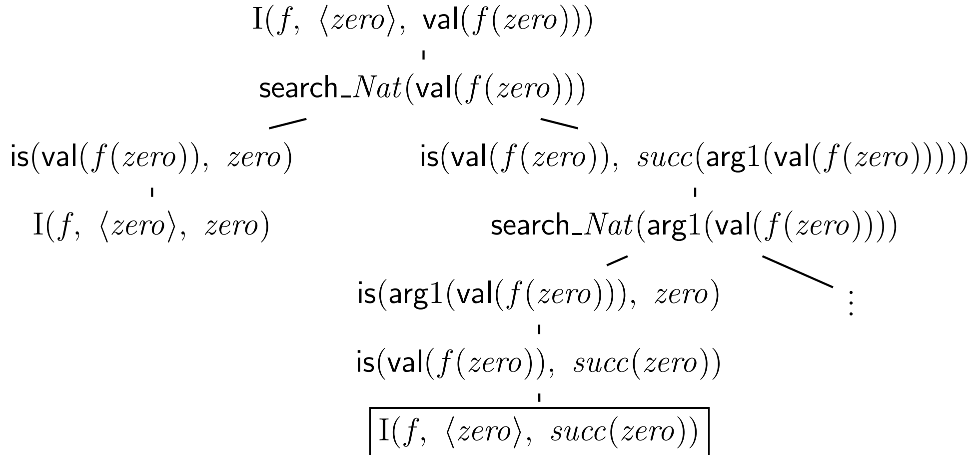
noch könnten wir das is-Atom des rechten Astes dazu benutzen, aus dem obersten I-Atom $\mathrm{I}(f,\ \langle zero\rangle,\ \boxed{succ(\mathsf{arg1}(\mathsf{val}(f(zero))))})$ herzuleiten. Dies kann sogar sinnvoll sein, um frühzeitig festzustellen, daß der Interpretations-Kandidat, der durch den rechten Ast repräsentiert wird, nicht funktional ist. Auf die Behandlung der Funktionalität kommen wir später zurück. Zunächst machen wir anders weiter, indem wir das zweite search-Atom auf dem rechten Ast wieder eine Verzweigung auslösen lassen. Wir betrachten nur den linken der jetzt neu entstehenden Zweige.



Wenn man sich das neue is-Atom, das darüberliegende is-Atom und die Wurzel des Baumes genau anschaut, dann wird klar, daß man jetzt auf irgendeine Weise zu dem I-Atom $\mathrm{I}(f,\ \langle zero\rangle,\ succ(zero))$ gelangen müßte. Technisch geht dies in zwei Schritten: zunächst inferieren wir aus dem eingerahmten und aus dem darüberliegenden is-Atom das neue Meta-Atom $\mathsf{is}(\mathsf{val}(f(zero)),\ succ(zero))$:



Und schließlich kann aus dem neuen Meta-Atom und der Wurzel das gewünschte inferiert werden.



Bei Betrachtung der Blätter dieses Baumes sieht man, daß das Verfahren mögliche endgültige Werte für zunächst vorläufige Einträge in I-Atome, hier $\mathsf{val}(f(zero))$, aufzählt. Wenn diese Aufzählung das alleinige Ziel wäre, dann wäre ein deduktives Verfahren (wie wir es hier zunächst nur angedeutet haben) nicht die geeignete Wahl. Wir haben aber schon erwähnt, daß auch die Zwischenergebnisse, wie wir sie hier erhalten, ausgenutzt werden können. Darauf kommen wir später wieder zurück.

Nun haben wir am Beispiel gesehen, wie nach zunächst unbekannten Konstruktortermen gesucht wird. In dem entstehenden Baum fassen wir jeden Ast als Interpretations-Kandidaten auf, wenngleich die Äste noch andere Atome enthalten als nur I-Atome (hier is und search, andere kommen später noch hinzu). Wir haben beim Aufbau dieses Baumes Regeln angewendet, ohne sie explizit anzugeben. Dies wird nun nachgeholt.

Der obige Baum ist im Tableau-Stil notiert. Auch die folgenden Regeln werden, zunächst, im Stile von Tableau-Regeln notiert. Versetzen wir uns an den Anfang zurück in einen Zustand, in dem der Baum nur aus einem Zweig besteht, welcher die obersten zwei Meta-Atome enthält. Die erste angewendete Regel ist gleich eine vergleichsweise komplizierte:

$$\frac{\mathsf{search_}Nat(x)}{\mathsf{is}(x,\ zero) \qquad \begin{array}{l}\mathsf{is}(x,\ succ(\mathsf{arg1}(x)))\\ \mathsf{search_}Nat(\mathsf{arg1}(x))\end{array}}$$

Die Variable x ist die Regelschema-Variable. Sie muß bei Anwendung der Regel substituiert werden, in unserem Fall mit $\mathsf{val}(f(zero))$. Die beiden Spalten unter dem Regelstrich bezeichnen wir als *Extensionen*. Intuitiv sind die verschiedenen Extensionen als verschiedene Alternativen anzusehen. Für jeden Konstruktor der Sorte *Nat* gibt es eine Extension. Innerhalb jeder Extension werden so viele neue search-Atome ‘erzeugt’, wie der jeweilige Konstruktor Argumente hat. Bei Anwendung der Regel verursacht dies eine Suche nach jedem der Argumente des Konstruktors.

Im nächsten Schritt hatten wir auf dem linkesten Ast $\mathrm{I}(f,\ \langle zero\rangle,\ zero)$ abgeleitet, und zwar aus $\mathrm{I}(f,\ \langle zero\rangle,\ \mathsf{val}(f(zero)))$ und $\mathsf{is}(\mathsf{val}(f(zero)),\ zero)$. Die entsprechende Regel lautet:

$$\frac{\begin{array}{c}\mathrm{I}(fv,\ tv,\ x)\\ \mathsf{is}(x,\ z)\end{array}}{\mathrm{I}(fv,\ tv,\ z)}$$

fv ist eine Funktions-Variable und *tv* eine Tupel-Variable. Diese Regel ist, im Gegensatz zur letzten, völlig unabhängig von der jeweiligen ADT-Signatur. Sie gehört zu einer Reihe von *Ersetzungsregeln*, die alle das is-Atom dazu verwenden, Argumente anderer Meta-Atomen zu ersetzen. (Streng genommen findet gar keine echte, destruktive 'Ersetzung' statt, denn das neue Atom kommt zusätzlich hinzu. Dies ist auch wichtig, wie man in obigem Beispiel sieht. Hier wird das I-Atom an der Wurzel auf unterschiedlichen Ästen auf unterschiedliche Weise 'ersetzt'.)

Danach hatten wir auf den rechten Ast die schon bekannte search_*Nat*-Regel angewendet. Im darauf folgenden Schritt entstand $\mathsf{is}(\mathsf{val}(f(zero)),\ succ(zero))$, und zwar mit der Regel:

$$\frac{\begin{array}{c}\mathsf{is}(x,\ succ(y))\\ \mathsf{is}(y,\ z)\end{array}}{\mathsf{is}(x,\ succ(z))}$$

Hier sollte man etwas genauer hinschauen. Diesmal wird nicht ein direktes Argument des Meta-Terms ersetzt, sondern der Term *innerhalb* des Konstruktors *succ*. Solche Regeln benötigen wir für alle Argumentpositionen aller Konstruktoren, denn nur auf diese Weise kann die Suche nach Argumenten eines Terms beitragen zur Suche nach dem Term selbst.

Wir werden solche Regeln im folgenden nicht zwei-, sondern eindimensional notieren. Statt also im Tableau-Stil

$$\frac{\begin{array}{c}at_1\\ \vdots\\ at_n\end{array}}{\begin{array}{ccc}at_{11} & & at_{m1}\\ \vdots & \cdots & \vdots\\ at_{1n_1} & & at_{mn_m}\end{array}}$$

zu schreiben (für eine Regel mit n Atomen in der Prämisse und m Extensionen, wobei die i-te Extension n_i Atome hat), schreiben wir linear:

$$at_1, \ldots, at_n \ \rightarrow\ at_{11}, \ldots, at_{1n_1}\ ;\ \ldots\ ;\ at_{m_1}, \ldots, at_{mn_m}\ .$$

In einer solchen Regel liest man das Komma als „und“, den Strichpunkt dagegen als „oder“.

Eine lineare Syntax ist schon deswegen vorteilhaft, weil die Regeln durch eine Transformation erst erzeugt werden (abhängig von einer Spezifikation). Für die Definition einer solchen Transformation ist eine lineare Schreibweise sehr hilfreich. Die Verwendung der obigen linearen Syntax *im speziellen* ist motiviert durch den Einsatz eines bestimmten Systems zur Ausführung der Regeln, welches genau diese Syntax verlangt. Bei diesem System handelt es sich um MGTP (Model Generation Theorem Prover) [FH91].

Die Benennung der Regel-Bestandteile ergibt sich aus dem folgenden Schaubild:

$$\underbrace{at_1, \ldots, at_n}_{\text{Prämisse}} \rightarrow \overbrace{\underbrace{at_{11}, \ldots, at_{1n_1}}_{\text{1.Extension}} \; ; \; \ldots \; ; \; \underbrace{at_{m_1}, \ldots, at_{mn_m}}_{\text{m.Extension}}}^{\text{Konklusion}} .$$

Eine formale Definition von Modell-Generierungs-Regeln folgt später. Es sei aber schon hier bemerkt, daß in der Konklusion nur solche Variablen auftreten dürfen, die schon in der Prämisse auftreten. Diese Eigenschaft wird in der Literatur als '*Werte-Beschränktheit*' (engl. '*range-restrictedness*') bezeichnet. In ihrer Konsequenz werden bei keiner Anwendung einer Regel Variablen erzeugt, so daß die bei Anwendung der Regeln erzeugten Meta-Atome *grund*, d.h. variablenfrei sind.

Die Prämisse kann leer sein (die Regel beginnt dann mit „→“). Dann spricht man von einer *positiven Regel*, im Falle einer einzigen Extension auch von einem *Faktum*. Aufgrund der Werte-Beschränktheit müssen alle Atome einer positiven Regel frei von Variablen sein. Auch die Konklusion kann leer sein (die Regel endet dann auf „→ .“). Intuitiv bedeutet dies, daß die Anwendbarkeit der Regel auf einen Modell-Kandidaten zur *Ablehnung* desselben führt. Solche Regeln heißen *negativ*. Leere Extensionen werden nicht betrachtet, diese können immer weggelassen werden.

3.2 Erzeugung der Modell-Generierungs-Regeln

Im letzten Abschnitt haben wir am Beispiel drei Regeln für die Modell-Generierung vorgestellt. Zwei davon hängen von der jeweiligen ADT-Signatur ab. In diesem Abschnitt werden nun alle für die Modell-Generierung nötigen Regeln eingeführt, wir nennen sie *Modell-Generierungs-Regeln* (MG-Regeln). Diese sind nicht statisch gegeben, sondern von der Spezifikation abhängig, und zwar so stark, daß wir die Menge der MG-Regeln auffassen als das Ergebnis einer *Transformation der Spezifikation*. Insbesondere werden auch die Axiome transformiert, von Σ-Formeln in MG-Regeln. Zunächst aber betrachten wir diejenigen MG-Regeln, die sich aus der Signatur ergeben.

3.2.1 Transformation der Signatur

Zunächst verallgemeinern wir die schon eingeführten Regeln. Die $\mathsf{search_}Nat$-Regel aus dem letzten Abschnitt lautet in linearer Schreibweise:

$$\mathsf{search_}Nat(x) \;\rightarrow\; \mathsf{is}(x, zero) \;;\; \mathsf{is}(x, succ(\mathsf{arg1}(x))),\; \mathsf{search_}Nat(\mathsf{arg1}(x)) \;.$$

Wenn wir nun nicht mehr eine bestimmte, sondern ab sofort eine beliebige ADT-Spezifikation $\langle \Sigma, AX \rangle$ betrachten, dann muß *Nat* ersetzt werden durch eine beliebige Sorte $s \in S$. Statt für *zero* und *succ* werden für die Konstruktoren der Sorte s Extensionen erzeugt, in denen jeweils auch eine Suche nach allen Argumenten des Konstruktorterms initiiert wird. Wir definieren eine entsprechende Transformation $\mathfrak{TransSort}_\Sigma(s)$ einer Sorte $s \in S$.

Definition 3.2.1 (Transformation einer Sorte)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, sei $s \in S$ und $C_s = \{c_1, \ldots, c_n\}$, wobei $|\alpha(c_i)| \leq |\alpha(c_j)|$ für $i \leq j$. Die Modell-Generierungs-Regel für die Suche nach Elementen der Sorte s ist definiert durch:

$$\begin{array}{rcc} \mathfrak{TransSort}_\Sigma(s) & = & \mathsf{search_}s(x) \\ & \rightarrow & \mathfrak{TransKonstr}_\Sigma(x, c_1) \\ & & ; \\ & & \vdots \\ & & ; \\ & & \mathfrak{TransKonstr}_\Sigma(x, c_n) \;. \end{array}$$

⋆

Man beachte die Strichpunkte zwischen den Extensionen der Regel. Die einzelnen Extensionen ergeben sich aus der folgenden Definition.

Definition 3.2.2 (Transformation eines Konstruktors)
Sei Σ eine ADT-Signatur und $c \in C_\Sigma$.

1. falls $\alpha(c) = \lambda$, dann:

$$\mathfrak{TransKonstr}_\Sigma(x, c) \;=\; \{\ \mathsf{is}(x, c)\ \}$$

2. falls $\alpha(c) = s_1 \ldots s_n$, dann:

$$\begin{array}{rcc} \mathfrak{TransKonstr}_\Sigma(x, c) & = & \{\ \mathsf{is}(x, c(\mathsf{arg1}(x), \ldots, \mathsf{arg}n(x)))\ \} \\ & & \cup \\ & & \{\ \mathsf{search_}s_1(\mathsf{arg1}(x))\ \} \\ & & \cup \\ & & \vdots \\ & & \cup \\ & & \{\ \mathsf{search_}s_n(\mathsf{arg}n(x))\ \} \end{array}$$

⋆

Die Transformation $\mathfrak{TransKonstr}_\Sigma(x, c)$ eines jeden Konstruktors $c \in CT_s$ bildet je eine Extension der Regel $\mathfrak{TransSort}_\Sigma(s)$. Wie angekündigt, werden bei der *Notation* konkreter Regeln, d.h. Instanzen dieser Definition, die Mengenklammern weggelassen. Trotzdem sind die Extensionen Mengen, hier wie in den später definierten Regeln.

Eine Instanz der obigen Regel, das Ergebnis von $\mathfrak{TransSort}_\Sigma(Nat)$, kennen wir schon. Wir zeigen hier noch ein Beispiel, die Transformation $\mathfrak{TransSort}_\Sigma(Stack)$ der Sorte *Stack* (definiert in der Spez. NatStack, S. 26).

$$\begin{array}{rcc} \mathsf{search_}Stack(x) & \rightarrow & \mathsf{is}(x, nil) \ ; \\ & & \mathsf{is}(x, push(\mathsf{arg1}(x), \mathsf{arg2}(x))), \\ & & \mathsf{search_}Nat(\mathsf{arg1}(x)), \\ & & \mathsf{search_}Stack(\mathsf{arg2}(x)) \ . \end{array}$$

(Vorsicht: das erste Trennungzeichen in der Konklusion ist ein ';' und trennt die beiden Extensionen, die weiteren Trennungszeichen sind Kommata ',' und trennen die Atome der zweiten Extension.)

In dieser Weise werden alle Sorten der jeweiligen Signatur transformiert.

Definition 3.2.3 (Transformation der Sorten)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die Modell-Generierungs-Regeln für die Sorten in S sind definiert durch:

$$\mathfrak{TransSorts}_\Sigma(S) \quad = \quad \{\mathfrak{TransSort}_\Sigma(s) \mid s \in S\}$$

$\star$

Neben der search-Regel haben wir im letzten Abschnitt noch zwei weitere Regeln kennengelernt, die beide zur Gruppe der Ersetzungsregeln zu zählen sind. In der linearen Schreibweise lauten diese:

$$\mathrm{I}(fv, tv, x),\ \mathsf{is}(x, z) \ \rightarrow\ \mathrm{I}(fv, tv, z) \ .$$

und

$$\mathsf{is}(x, succ(y)),\ \mathsf{is}(y, z) \ \rightarrow\ \mathsf{is}(x, succ(z)) \ .$$

Die zweite dieser Regeln ist eine mögliche Instanz eines allgemeinen Musters, nach welchem jedes Argument eines jeden Konstruktors mit Hilfe von is ersetzt werden kann. Die erste Regel hingegen ist allgemein genug und wird unverändert in die Definition der Transformation übernommen. Wir benötigen aber darüber hinaus weitere Ersetzungsregeln für I-Atome. Jedes I-Atom, welches in einem Interpretations-Kandidaten auftritt, besitzt als zweites Argument ein Tupel von Meta-Termen. Auch diese müssen ersetzt werden können. Wenn z.B. ein Meta-Atom $\mathrm{I}(f, \langle \mathsf{val}(a) \rangle, zero)$ vorliegt und außerdem $\mathsf{is}(\mathsf{val}(a), succ(zero))$, dann wollen wir daraus ableiten, daß $\mathrm{I}(f, \langle succ(zero) \rangle, zero)$ 'gilt'. Dies ginge mit der Regel:

$$\mathrm{I}(fv, \langle x \rangle, y),\ \mathsf{is}(x, z) \ \rightarrow\ \mathrm{I}(fv, \langle z \rangle, y) \ .$$

Solche Regeln brauchen wir auch für mehrstellige Tupel, und dort dann je eine pro Argumentposition.

Wir definieren nun die Menge aller Ersetzungsregeln $\mathfrak{Ersetzung}_\Sigma$, in Abhängigkeit von der jeweiligen Signatur Σ. Wir werden dabei die Ersetzung in zwei noch unbekannten Typen von Meta-Atomen gleich mitdefinieren. Es handelt sich hierbei um same- bzw. different-Atome, die aus der Transformation der Objekt-(Un-)Gleichheit stammen und im Anschluß diskutiert werden.

Definition 3.2.4 (Ersetzungsregeln)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die Regeln zur Ersetzung von Meta-Termen in Meta-Atomen mittels is sind definiert durch:

$$\mathfrak{Ersetzung}_\Sigma = \mathfrak{ErsetzSame} \cup \mathfrak{ErsetzDiff} \cup \mathfrak{ErsetzI}_\Sigma \cup \mathfrak{ErsetzIs}_\Sigma$$

- $\mathfrak{ErsetzSame}$ ist die Menge:

$$\{\ \mathsf{same}(x,y),\ \mathsf{is}(x,z)\ \rightarrow\ \mathsf{same}(z,y)\ .\ ,\quad \mathsf{same}(x,y),\ \mathsf{is}(y,z)\ \rightarrow\ \mathsf{same}(x,z)\ .\ \}$$

- $\mathfrak{ErsetzDiff}$ ist die Menge:

$$\{\ \mathsf{different}(x,y),\ \mathsf{is}(x,z)\ \rightarrow\ \mathsf{different}(z,y)\ .\ ,\quad \mathsf{different}(x,y),\ \mathsf{is}(y,z)\ \rightarrow\ \mathsf{different}(x,z)\ .\ \}$$

- $\mathfrak{ErsetzI}_\Sigma$ ist minimal definiert durch:

  1. $\mathrm{I}(fv, tv, x),\ \mathsf{is}(x,z)\ \rightarrow\ \mathrm{I}(fv, tv, z)\ .\ \in \mathfrak{ErsetzI}_\Sigma$
  2. falls eine Funktion $f \in \overline{\mathcal{F}}$ existiert mit $|\alpha(f)| = n$, dann gilt:

$$\begin{array}{c}\{\ \mathrm{I}(fv, \langle x_1, \ldots, x_n\rangle, y),\ \mathsf{is}(x_1, z)\ \rightarrow\ \mathrm{I}(fv, \langle z, \ldots, x_n\rangle, y)\ .\ ,\\ \vdots \\ \mathrm{I}(fv, \langle x_1, \ldots, x_n\rangle, y),\ \mathsf{is}(x_n, z)\ \rightarrow\ \mathrm{I}(fv, \langle x_1, \ldots, z\rangle, y)\ .\ \}\\ \subseteq \\ \mathfrak{ErsetzI}_\Sigma\end{array}$$

- $\mathfrak{ErsetzIs}_\Sigma$ ist minimal definiert durch:
  für jeden Konstruktor $c \in \overline{\mathcal{C}}$ mit $|\alpha(c)| = n$ gilt:

$$\begin{array}{c}\{\ \mathsf{is}(x, c(y_1, \ldots, y_n)),\ \mathsf{is}(y_1, z)\ \rightarrow\ \mathsf{is}(x, c(z, \ldots, y_n))\ .\ ,\\ \vdots \\ \mathsf{is}(x, c(y_1, \ldots, y_n)),\ \mathsf{is}(y_n, z)\ \rightarrow\ \mathsf{is}(x, c(y_1, \ldots, z))\ .\ \}\\ \subseteq \\ \mathfrak{ErsetzIs}_\Sigma\end{array}$$

⋆

Zum Abschluß der signaturabhängigen MG-Regeln führen wir die Behandlung der Gleichheit und der Ungleichheit im Mechanismus der Modell-Generierung ein. Das Meta-Atom $\mathsf{same}(t_1, t_2)$ besagt, daß die beiden Meta-Terme t_1 und t_2 die gleiche Bedeutung haben. Falls es sich bei beiden um reine Konstruktorterme handelt, dann müssen sie syntaktisch identisch sein. Aber auch z.B. $\mathsf{val}(a)$ und *zero* können das gleiche bedeuten, z.B. falls nämlich $val_{\mathcal{I}}(a) = zero$ eine zutreffende Aussage ist. Ein same-Atom kann der Transformation der Axiome entstammen oder aber der Forderung nach Funktionalität, s.u. Die Regeln zur Behandlung von same basieren auf dem Vergleich der Konstruktoren. Ein Beispiel für eine solche Regel wäre

$$\mathsf{same}(succ(x),\ zero) \ \rightarrow \ .$$

Dies ist ein Beispiel für eine Regel, die den Modell-Kandidaten, auf den sie anwendbar ist, *verwirft* (im Tableau-Sprachgebrauch würde man sagen: der entsprechende Ast wird *geschlossen*). Verschiedene führende Konstruktoren der Argumente eines same-Atoms führen also zur Ablehnung. Sind die führenden Konstruktoren hingegen gleich, dann müssen ihre Argumente überprüft werden. Ein Beispiel hierfür:

$$\mathsf{same}(push(x_1, x_2),\ push(y_1, y_2)) \ \rightarrow \ \mathsf{same}(x_1, y_1),\ \mathsf{same}(x_2, y_2)\ .$$

Auch hier, bei der 'Überprüfung' der Gleichheit, wäre der Einsatz von Deduktion übertrieben, ginge es nur um den Vergleich zweier Konstruktorterme. Da aber unsere Meta-Terme auch noch aus anderen Bestandteilen bestehen, wie val- und $\mathsf{arg}i$-Termen, nach deren Ersetzung erst noch gesucht wird, ist es nötig, die Meta-Terme ihrer Struktur nach zu analysieren. Auf diese Weise kann man z.B. einen Modell-Kandidaten, der $\mathsf{same}(succ(\mathsf{val}(a)), zero)$ enthält, verwerfen (mit der obigen Regel), obwohl die Suche nach einer Ersetzung für $\mathsf{val}(a)$ noch gar nicht abgeschlossen ist.

Analog zur Gleichheit läßt sich auch die *Un*gleichheit durch Meta-Atome ausdrücken, und zwar durch solche der Form $\mathsf{different}(t_1, t_2)$. Wir geben auch hier zwei Beispielregeln an.

$$\mathsf{different}(zero, zero) \ \rightarrow \ .$$

$$\mathsf{different}(push(x_1, x_2),\ push(y_1, y_2)) \ \rightarrow \ \mathsf{different}(x_1, y_1);\ \mathsf{different}(x_2, y_2)\ .$$

Es ist zu beachten, daß die zweite Regel verzweigt! Wenn zwei durch *push* angeführte Meta-Terme verschieden sein sollen, dann müssen die ersten *oder* die zweiten Argumente verschieden sein.

Regeln wie die obigen realisieren die *freie Erzeugtheit* der Grundbereiche: zwei verschiedene Konstruktorterme sind semantisch ungleich. Darum nennen wir auch die nun zu definierende Menge von MG-Regeln 'Regeln für die freie Erzeugtheit'.

Definition 3.2.5 (Regeln für die freie Erzeugtheit)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die Regeln für die freie Erzeugtheit sind definiert durch:

$$\mathfrak{FreiErz}_\Sigma = \mathfrak{TestSame}_\Sigma \cup \mathfrak{TestDiff}_\Sigma$$

- $\mathfrak{TestSame}_\Sigma$ ist minimal definiert durch:
  1. Für je zwei verschiedene Konstruktoren der gleichen Sorte $s \in S$, $\{c_1, c_2\} \subseteq C_s$ mit $c_1 \neq c_2$ und $|\alpha(c_1)| = n$, $|\alpha(c_2)| = m$, gilt:
     - falls $n \neq 0$ und $m \neq 0$, dann ist
       $\mathsf{same}(c_1(x_1, \ldots, x_n),\ c_2(y_1, \ldots, y_m)) \ \rightarrow\ .\ \in \mathfrak{TestSame}_\Sigma$
     - falls $n \neq 0$ und $m = 0$, dann ist
       $\{\ \mathsf{same}(c_1(x_1, \ldots, x_n),\ c_2) \ \rightarrow\ .\ ,$
       $\mathsf{same}(c_2,\ c_1(x_1, \ldots, x_n)) \ \rightarrow\ .\ \} \subseteq \mathfrak{TestSame}_\Sigma$
     - falls $n = m = 0$, dann ist
       $\mathsf{same}(c_1,\ c_2) \ \rightarrow\ .\ \in \mathfrak{TestSame}_\Sigma$
  2. Für jeden Konstruktor $c \in C_s$ mit $|\alpha(c)| = n \neq 0$ gilt:
     $$\left.\begin{array}{l}\mathsf{same}(c(x_1, \ldots, x_n),\ c(y_1, \ldots, y_n)) \\ \rightarrow\quad \mathsf{same}(x_1,\ y_1),\ \ldots,\ \mathsf{same}(x_n,\ y_n)\ .\end{array}\right\} \in \mathfrak{TestSame}_\Sigma$$
- $\mathfrak{TestDiff}_\Sigma$ ist minimal definiert durch:

  Für jeden Konstruktor $c \in C_s$ gilt:
  - falls $\alpha(c) = \lambda$, dann ist
    $\mathsf{different}(c,\ c) \ \rightarrow\ .\ \in \mathfrak{TestDiff}_\Sigma$
  - falls $|\alpha(c)| = n \neq 0$, dann ist:
    $$\left.\begin{array}{l}\mathsf{different}(c(x_1, \ldots, x_n),\ c(y_1, \ldots, y_n)) \\ \rightarrow\quad \mathsf{different}(x_1,\ y_1) \\ \qquad ; \\ \qquad \vdots \\ \qquad ; \\ \qquad \mathsf{different}(x_n,\ y_n)\ .\end{array}\right\} \in \mathfrak{TestDiff}_\Sigma$$

⋆

Bei der Definition von $\mathfrak{TestSame}_\Sigma$ ist zu beachten, daß für die ablehnenden same-Regeln, d.h. bei ungleichen Konstruktoren, die jeweilige Regel mit vertauschten *same*-Argumenten auch in $\mathfrak{TestSame}_\Sigma$ enthalten ist. In einem Fall haben wir das explizit so definiert, in den zwei anderen Fällen erreicht man dies schlicht durch die umgekehrte Zuordnung der Indizes zu den Konstruktoren.

Zwischen den same- bzw. different-Regeln zum Vergleich von Termen mit identischem führenden Konstruktor besteht ein wichtiger Unterschied. Die same-Regel

ist konjunktiv, während die different-Regel disjunktiv, d.h. verzweigend ist. Intuitiv verlangt die same-Regel die 'Gleichheit' aller Argumente, während die different-Regel die 'Ungleichheit' nur eines Argumentes verlangt.

Damit haben wir nun alle MG-Regeln definiert, die allein aus der Signatur Σ einer Spezifikation hervorgehen.

Definition 3.2.6 (Transformation der Signatur)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die Modell-Generierungs-Regeln für diese Signatur sind definiert durch:

$$\mathfrak{TransSig}(\Sigma) = \mathfrak{TransSorts}_\Sigma(S) \cup \mathfrak{Ersetzung}_\Sigma \cup \mathfrak{FreiErz}_\Sigma$$

$\star$

3.2.2 Transformation der Axiome

Unser eigentliches Ziel ist ja die Erzeugung des Modells einer Spezifikation (insbesondere einer Gegenspezifikation). Es sei daran erinnert, daß die Spezifikationen, für die wir nach Modellen suchen, normalisiert sind, d.h. ihre Axiome bestehen aus Disjunktionen von Gleichungen und Ungleichungen. Bisher haben wir in den MG-Regeln nur die Signatur berücksichtigt. Aus dieser läßt sich ablesen, für welche Funktionen und über welchem Grundbereich nach Interpretationen gesucht werden muß. Im Zentrum des Interesses steht aber, daß wir nach einem Modell suchen, welches die Axiome der Spezifikation erfüllt. In dem hier verfolgten Ansatz wird jedes Axiom transformiert zu einer MG-Regel. Der folgende Abschnitt erklärt, in welcher Weise dies geschieht. Zunächst werden die zugrundeliegenden Ideen erläutert, bevor die Transformation präzise definiert wird.

Die Grundbestandteile der normalisierten Axiome sind Gleichungen und Ungleichungen. Konzentrieren wir uns zunächst auf Gleichungen. Die Tabelle 3.1 gibt einen ersten Eindruck von dem Zusammenhang zwischen Gleichungen in den Axiomen und I-Atomen in den Regeln. Hier stehen die f_i für Funktionen, die ct_i für Konstruktorterme. k_i bzw. x stehen für natürlichsprachlich existenz- bzw. allquantifizierte Konstruktorterm-Variablen.

In der ersten Zeile der Tabelle ist die Entsprechung zwischen der Gleichung und dem I-Atom ganz offensichtlich. Die zweite Zeile verdeutlicht: will man die Gleichung (links) erfüllen, dann muß man nach einem Konstruktorterm k suchen, so daß $\mathrm{I}(f_1, \langle ct_1 \rangle, k)$ und $\mathrm{I}(f_2, \langle ct_2 \rangle, k)$ in den zu konstruierenden Interpretations-Kandidaten aufgenommen werden können, *ohne* zur Ablehnung desselben zu führen. Um die Suche zu initiieren, werden zunächst zwei I-Atome erzeugt, die jeweils im letzten Argument einen Platzhalter für den zu suchenden Konstruktorterm enthalten. Als Platzhalter bietet sich hier sowohl $\mathsf{val}(f_1(ct_1))$ als auch

Objekt-Gleichheit	I-Atome
Grund-Fall:	
$f(ct_1) \doteq ct_2$	$\mathrm{I}(f, \langle ct_1 \rangle, ct_2)$
(beidseitig Funktionen:) $f_1(ct_1) \doteq f_2(ct_2)$	*Es existiert* ein Konstruktorterm k, so daß $\mathrm{I}(f_1, \langle ct_1 \rangle, k)$ und $\mathrm{I}(f_2, \langle ct_2 \rangle, k)$ gilt.
(geschachtelte Funktionen:) $f_1(f_2(ct_1)) \doteq ct_2$	*Es existiert* ein Konstruktorterm k, so daß $\mathrm{I}(f_2, \langle ct_1 \rangle, k)$ und $\mathrm{I}(f_1, \langle k \rangle, ct_2)$ gilt.
(gemischt:) $f_1(f_2(ct_1)) \doteq f_3(ct_2)$	*Es existieren* Konstruktorterme k_1, k_2, so daß $\mathrm{I}(f_2, \langle ct_1 \rangle, k_1)$, $\mathrm{I}(f_1, \langle k_1 \rangle, k_2)$ und $\mathrm{I}(f_3, \langle ct_2 \rangle, k_2)$.
Variablen-Fall:	
$f(x) \doteq ct$	*Für alle* Konstruktorterme x gilt $\mathrm{I}(f, \langle x \rangle, ct)$.
(beidseitig Funktionen:) $f(x, y) \doteq f(y, x)$	*Für alle* Konstruktorterme x und y *existiert* ein Konstruktorterm k, so daß $\mathrm{I}(f, \langle x, y \rangle, k)$ und $\mathrm{I}(f, \langle y, x \rangle, k)$ gilt.
(geschachtelte Funktionen:) ⋮	⋮ ⋮

Tabelle 3.1: Beispiele für Gleichheiten und korrespondierende I-Atome

$\mathsf{val}(f_2(ct_2))$ an. Wir werden, recht willkürlich, $\mathsf{val}(f_2(ct_2))$ verwenden. Diesen Meta-Term gilt es später zu ersetzen. Dafür muß eine Suche initiiert werden, durch das Meta-Atom $\mathsf{search_}s(\mathsf{val}(f_2(ct_2)))$, wobei s die Zielsorte von f_1 und f_2 ist. Das bedeutet: ist

$$f_1(ct_1) \doteq f_2(ct_2)$$

ein Axiom (was ungewöhnlich wäre, da die Gleichung keine Variablen enthält), dann würde die Transformation dieses Axioms resultieren in der MG-Regel:

$$\begin{aligned} \rightarrow \quad & \mathrm{I}(f_1, \langle ct_1 \rangle, \mathsf{val}(f_2(ct_2))), \\ & \mathrm{I}(f_2, \langle ct_2 \rangle, \mathsf{val}(f_2(ct_2))), \\ & \mathsf{search_}s(\mathsf{val}(f_2(ct_2))) \ . \end{aligned}$$

Da die Prämisse dieser Regel leer ist, kann sie unmittelbar angewendet werden. Darum finden sich diese drei Meta-Atome der Konklusion in jedem Interpretations-Kandidaten wieder (falls dieser nicht zuvor schon abgelehnt wird).

Die eben angegebene, etwas unsymmetrische Transformation der Gleichung in eine MG-Regel ist zwar günstig für die Effizienz des Verfahrens, jedoch von Nachteil für die Definition und die theoretische Behandlung der Transformation. Eine

symmetrische Version wäre:

$$\begin{aligned}\rightarrow\quad & \mathsf{same}(\mathsf{val}(f_1(ct_1)), \mathsf{val}(f_2(ct_2))),\\ & \mathrm{I}(f_1, \langle ct_1\rangle, \mathsf{val}(f_1(ct_1))),\\ & \mathrm{I}(f_2, \langle ct_2\rangle, \mathsf{val}(f_2(ct_2))),\\ & \mathsf{search_}s(\mathsf{val}(f_1(ct_1))),\\ & \mathsf{search_}s(\mathsf{val}(f_2(ct_2)))\ .\end{aligned}$$

Nach Anwendung dieser Regel, wenn alle fünf Meta-Atome auf dem Ast sind, wird aufgrund der zwei search–Atome nach zwei Konstruktortermen gesucht. Die Zwischenergebnisse werden durch Anwendung von Ersetzungsregeln in die beiden I-Atome und in das same-Atom übertragen. Mit Hilfe der same-Regeln (vgl. Def. 3.2.5) werden dann solche Paare von Zwischenergebnissen abgelehnt, die aufgrund der schon bekannten Konstruktoren notwendigerweise verschieden sind.

Es liegt auf der Hand, warum die erste Variante effizienter ist: Anstatt nach *zwei* Konstruktortermen zu suchen und dabei unterschiedliche Ergebnisse immer wieder mit Hilfe der same-Regeln abzulehnen, wird gleich nach nur *einem* Konstruktorterm gesucht.

Die beiden hier am Beispiel skizzierten Möglichkeiten zur Transformation von Gleichungen werden im folgenden unterschiedliche Rollen spielen. Für die Theorie des Verfahrens ist die symmetrische Version wesentlich günstiger. Sie ermöglicht eine klare Trennung zwischen der Transformation der Gleichungen und Ungleichungen einerseits sowie der beteiligten Terme andererseits, s.u. Die ohnehin recht komplizierten Beweise werden nicht durch zusätzliche Fallunterscheidungen belastet. Die kompaktere, unsymmetrische Transformation hingegen wird die größere Rolle spielen bei dem praktischen Einsatz der Methode, da sie effizienter ist. Auch bei der Angabe von Beispiel-Transformationen zu Erklärungszwecken werden wir häufig die kompaktere optimierte Variante verwenden.

Betrachten wir nun die dritte Zeile der Tabelle 3.1. Diese soll demonstrieren, daß es i.a. nötig ist, Funktionsterme rekursiv zu transformieren. Diesmal muß nach einem Konstruktorterm gesucht werden, der das Ergebnis der Interpretation der inneren Funktion f_2 auf dem Argument ct_1 ist und gleichzeitig selbst wieder als Argument für die Interpretation von f_1 verwendet wird. Ist also

$$f_1(f_2(ct_1)) \doteq ct_2$$

ein Axiom, dann resultiert die (optimierte) Transformation in folgender Regel:

$$\begin{aligned}\rightarrow\quad & \mathrm{I}(f_2, \langle ct_1\rangle, \mathsf{val}(f_2(ct_1))),\\ & \mathrm{I}(f_1, \langle \mathsf{val}(f_2(ct_1))\rangle, ct_2),\\ & \mathsf{search_}s(\mathsf{val}(f_2(ct_1)))\ .\end{aligned}$$

Schließlich treten in der vierten Zeile der Tabelle 3.1 die beiden Fälle gleichzeitig auf: links und rechts der Gleichung stehen Funktionsterme, und innerhalb des linken Funktionsterms tritt ein weiterer Funktionsterm auf. Diesmal muß nach

zwei Konstruktortermen gesucht werden. Sei also

$$f_1(f_2(ct_1)) \doteq f_3(ct_2)$$

ein Axiom, dann ist die transformierte Regel:

$$\begin{aligned}\rightarrow\ & \mathrm{I}(f_2, \langle ct_1\rangle, \mathsf{val}(f_2(ct_1))),\\ & \mathrm{I}(f_1, \langle \mathsf{val}(f_2(ct_1))\rangle, \mathsf{val}(f_3(ct_2))),\\ & \mathrm{I}(f_3, \langle ct_2\rangle, \mathsf{val}(f_3(ct_2))),\\ & \mathsf{search_}s(\mathsf{val}(f_2(ct_1))),\\ & \mathsf{search_}s'(\mathsf{val}(f_3(ct_2)))\ .\end{aligned}$$

Bisher haben wir nur den Fall variablenfreier Axiome betrachtet, welche aber nur selten auftreten.[3] Ihre Betrachtung diente eher dem 'seperation of concerns'. Die Variablen der Axiome bringen ein ganz anderes Moment in die Transformation ein, welches wir nun diskutieren wollen.

Die vorletzte Zeile der Tabelle 3.1 behandelt die Gleichung $f(x) \doteq ct$. Wenn diese in einem normalisierten Axiom auftritt, dann ist x implizit allquantifiziert. Entsprechend sollte für *jeden* Konstruktorterm ct' (passender Sorte) das Meta-Atom $\mathrm{I}(f, \langle ct'\rangle, ct)$, in welchem x durch ct' *ersetzt* wird, 'gelten'. Diese *Ersetzung* wird mit Hilfe von bestimmten MG-Regeln und deren Anwendung realisiert. Wir geben gleich eine solche Regel an und diskutieren sie im Anschluß. Sei $f(x) \doteq ct$ ein Axiom und sei s sie Sorte von x, dann resultiert die Transformation des Axioms in folgender MG-Regel:

$$s(x) \ \rightarrow\ \mathrm{I}(f, \langle x\rangle, ct)\ .$$

Hier verwenden wir die Sorte s der Objekt-Signatur als Prädikat zur Bildung eines (Meta-)Atoms. Dies ist eine bekannte Technik, die oftmals verwendet wird, um Sorten-Information in einem sortenfreien Rahmen zu repräsentieren. U.a. diesen Zweck erfüllt das Sortenprädikat auch hier. Wichtiger jedoch ist die *Steuerung der Instantiierung* von Axiomen. Die obige Regel hat ja eine operationale Bedeutung: enthält ein Modell-Kandidat das Meta-Atom $s(ct')$ (wobei ct' ein Konstruktorterm ist), dann kann das Meta-Atom $\mathrm{I}(f, \langle ct'\rangle, ct)$ zu dem Modell-Kandidaten hinzugefügt werden.

Generell werden Axiome mit freien Variablen nach dem (zunächst) gleichen Muster transformiert, das wir in den bisherigen Beispielen schon angedeutet haben. Variablen werden dabei, so wie bisher schon die Konstruktorterme, unverändert übernommen. Die Meta-Atome der Konklusion der Regel enthalten dann die gleichen Variablen wie das Axiom. Schließlich werden diese Variablen aber gewissermaßen 'gebunden', indem die entsprechenden Sorten-Atome in die Prämisse der Regel aufgenommen werden.

[3] Allerdings tritt in den uns interessierenden *Gegenspezifikationen* regelmäßig *ein* variablenfreies Axiom auf, welches das skolemisierte Gegenteil der untersuchten Vermutung darstellt.

Wenden wir dieses Prinzip noch auf die letzte Zeile der Tabelle 3.1 an. Die Gleichung $f(x,y) \doteq f(y,x)$ formalisiert die Kommutativität der Funktion f. Sei diese Gleichung ein Axiom. Entsprechend dem in den obigen Beispielen diskutierten Muster besteht die *Konklusion* der zu bildenden MG-Regel aus den Meta-Atomen $\mathrm{I}(f, \langle x,y \rangle, \mathsf{val}(f(y,x)))$, $\mathrm{I}(f, \langle y,x \rangle, \mathsf{val}(f(y,x)))$ und $\mathsf{search_}s(\mathsf{val}(f(y,x)))$. (Wieder benutzen wir ein und denselben val-Term für beide I-Atome, und nur nach diesem müssen wir suchen.) Damit ist die Regel aber noch nicht komplett. Die freien Variablen müssen noch durch Sorten-Atome in der Prämisse 'gebunden' werden. Das Ergebnis lautet:

$$\begin{array}{lll} s(x),\ s(y) & \rightarrow & \mathrm{I}(f, \langle x,y \rangle, \mathsf{val}(f(y,x))), \\ & & \mathrm{I}(f, \langle y,x \rangle, \mathsf{val}(f(y,x))), \\ & & \mathsf{search_}s(\mathsf{val}(f(y,x)))\ . \end{array}$$

(Die Argumente einer kommutativen Funktion müssen notwendigerweise die gleiche Sorte besitzen.) Betrachten wir noch kurz die Wirkung dieser Regel. Enthält der aktuelle Modell-Kandidat zwei Meta-Atome $s(ct_1)$ und $s(ct_2)$ (für Konstruktorterme ct_1 und ct_2), dann werden die folgenden Meta-Atome zum Modell-Kandidaten hinzugenommen:

$$\begin{array}{c} \mathrm{I}(f, \langle ct_1, ct_2 \rangle, \mathsf{val}(f(ct_2, ct_1))), \\ \mathrm{I}(f, \langle ct_2, ct_1 \rangle, \mathsf{val}(f(ct_2, ct_1))) \\ \text{und} \\ \mathsf{search_}s(\mathsf{val}(f(ct_2, ct_1))). \end{array}$$

Dies ist die richtige Stelle, um auf einen wichtigen Punkt hinzuweisen. Die Verwendung des Meta-Termes $\mathsf{val}(f(y,x))$ mit den Variablen y und x in der MG-Regel hat den Effekt, daß bei Anwendungen dieser Regel für *jedes* Paar ct_2 und ct_1 ein *neuer* Platzhalter erzeugt wird. Die *Verwendung der Variablen* in den val-Termen kodiert demnach die Erzeugung neuer Platzhalter nach Bedarf. Die *Verwendung der Terme*, wie hier $f(y,x)$, als Parameter von val, hat darüberhinaus den Vorteil, daß Platzhalter wiederverwendet werden können, wo dies semantisch gerechtfertigt ist. Falls z.B. die beiden Terme $g(x)$ und $g(y)$ in zwei unterschiedlichen Axiomen einer Spezifikation auftreten, dann enthalten die beiden resultierenden Regeln die Meta-Terme $\mathsf{val}(g(x))$ und $\mathsf{val}(g(y))$, sowie jeweils die entsprechenden search-Atome. Falls dann beide Regeln mit dem gleichen Konstruktorterm ct instantiiert werden, dann entsteht in beiden Fällen derselbe Grund-Meta-Term $\mathsf{val}(g(ct))$, für den auch nur eine Suche initiiert wird!

Das Konzept der val-Terme ähnelt der Verwendung von ϵ-Termen in 'freie-Variablen-Tableaux', welche von M. Giese und dem Autor gemeinsam untersucht wurden [GA99].

Bisher haben wir nur die Transformation von Gleichungen betrachtet. Die Transformation von Ungleichungen erklärt sich auch am besten an einem Beispiel. Auf

S. 81 hatten wir die symmetrische MG-Regel für die Gleichung $f_1(ct_1) \doteq f_2(ct_2)$ angegeben. Sei die entsprechende *Ungleichung*

$$f_1(ct_1) \not\doteq f_2(ct_2)$$

ein Axiom, dann ist die resultierende MG-Regel fast identisch mit derjenigen auf S. 81, nur daß an Stelle des same-Terms der entsprechende different-Term steht:

$$\begin{aligned} \rightarrow\ & \mathsf{different}(\mathsf{val}(f_1(ct_1)), \mathsf{val}(f_2(ct_2))), \\ & \mathrm{I}(f_1, \langle ct_1 \rangle, \mathsf{val}(f_1(ct_1))), \\ & \mathrm{I}(f_2, \langle ct_2 \rangle, \mathsf{val}(f_2(ct_2))), \\ & \mathsf{search_}s(\mathsf{val}(f_1(ct_1))), \\ & \mathsf{search_}s(\mathsf{val}(f_2(ct_2)))\ . \end{aligned}$$

Im Unterschied zur Behandlung der Gleichheit gibt es hier nichts zu optimieren. Diesmal müssen wir zwei verschidene val-Terme betrachten und getrennt nach sie ersetzenden Konstruktortermen suchen. Wir können nicht, wie noch bei der Gleichheit, die beiden Meta-Terme im Vornherein miteinander identifizieren.

Die Spezifikationen, für die wir nach Modellen suchen, sind normalisiert. Laut Def. 2.5.1 sind die Axiome dann Klauseln, d.h. Disjunktionen von Literalen (Gleichungen und Ungleichungen). Das letzte noch nicht diskutierte Strukturelement der zu transformierenden Axiome ist demnach die Disjunktion. Wie schon im letzten Kapitel angedeutet, verursacht die Zulassung disjunktiver Axiome keine zusätzlichen Schwierigkeiten für die Methode. Schon allein die bereits diskutierte Suche nach Konstruktortermen verlangt nach Verzweigungen im Baum der Modell-Kandidaten, vgl. die disjunktiven search-Regeln (Beispiel auf S. 71 und S. 75). Disjunktive Axiome sind dann nur noch eine weitere mögliche Ursache für Verzweigungen. Jedes Disjunktionsglied eines Axioms resultiert in genau einer Extension der transformierten MG-Regel (syntaktisch getrennt durch den Strichpunkt, s.u. Def. 3.2.8).

Nach diesen beispielgetriebenen Vorbereitungen werden wir nun präziser und definieren, top-down, die Bestandteile der Transformation von Axiomen.

Definition 3.2.7 (Transformation der Axiome)
Sei Σ eine ADT-Signatur und sei $AX \subseteq Kl_\Sigma$. Die Modell-Generierungs-Regeln für AX sind definiert durch:

$$\mathfrak{TransAxioms}_\Sigma(AX) \quad = \quad \{\mathfrak{TransAxiom}_\Sigma(ax) \mid ax \in AX\}$$

⋆

Definition 3.2.8 (Transformation eines Axioms)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und sei $ax \in Kl_\Sigma$ mit $ax = lit_1 \vee \ldots \vee lit_n$,

$Var(ax) = \{x_1, \ldots, x_m\}$ und $sort(x_i) = s_i$. Die Modell-Generierungs-Regel für ax ist definiert durch:

$$\begin{array}{rcl} \mathfrak{TransAxiom}_\Sigma(ax) & = & s_1(x_1)\ ,\ \ldots\ ,\ s_m(x_m) \\ & \rightarrow & \mathfrak{TransLit}_\Sigma(lit_1) \\ & & ; \\ & & \vdots \\ & & ; \\ & & \mathfrak{TransLit}_\Sigma(lit_n)\ . \end{array}$$

⋆

Ehe wir uns der Transformation der Literale zuwenden, führen wir noch eine nützliche Abkürzung ein, die eine sonst ständig zu treffende Fallunterscheidung sozusagen auslagert. Wir haben in den obigen Beispielen gesehen, daß Terme, die Funktionen enthalten, durch val() zu kapseln sind, was intuitiv für die (verzögerte) Auswertung der Terme steht. Konstruktorterme hingegen werden unverändert von den Axiomen in die Regeln übernommen, wie auch die Variablen, da diese bei Anwendung der Regeln mit Konstruktortermen instantiiert werden. Noch allgemeiner können alle Unterterme der Axiome, die keine Funktionen enthalten, also nur aus Konstruktoren und Variablen bestehen, innerhalb der MG-Regeln durch sich selbst repräsentiert werden. Diese *funktionsfreien* Terme einer Signatur Σ wurden in Def. 2.2.10 definiert als die Menge FFT_Σ.

Definition 3.2.9 (Repräsentation der Terme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $t \in T_\Sigma$.

$$\mathfrak{Rep}(t) \quad = \quad \begin{cases} t & \text{falls } t \in FFT_\Sigma \\ \mathsf{val}(t) & \text{sonst} \end{cases}$$

⋆

Damit läßt sich nun die Transformation der Literale definieren, zunächst symmetrisch, also unoptimiert. Es sei daran erinnert, daß jedes $\mathfrak{TransLit}_\Sigma(lit_i)$ eine Extension der Regel $\mathfrak{TransAxiom}_\Sigma(ax)$ darstellt, und daß wir Extensionen als *Mengen* (von Meta-Termen) auffassen.

Definition 3.2.10 (Transformation der Literale)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $t_1, t_2 \in T_s$ für ein $s \in S$.

$$\begin{array}{rcl} \mathfrak{TransLit}_\Sigma(t_1 \doteq t_2) & = & \{\ \mathsf{same}(\mathfrak{Rep}(t_1), \mathfrak{Rep}(t_2))\ \} \\ & & \cup \\ & & \mathfrak{TransTerm}_\Sigma(t_1) \\ & & \cup \\ & & \mathfrak{TransTerm}_\Sigma(t_2) \end{array}$$

$$\mathfrak{TransLit}_\Sigma(t_1 \not\doteq t_2) \quad = \quad \{\ \mathsf{different}(\mathfrak{Rep}(t_1), \mathfrak{Rep}(t_2))\ \} \cup \mathfrak{TransTerm}_\Sigma(t_1) \cup \mathfrak{TransTerm}_\Sigma(t_2)$$

⋆

Komplizierter, weil rekursiv, ist die Transformation $\mathfrak{TransTerm}_\Sigma$ der Terme. Hier müssen wir eine wichtige Fallunterscheidung treffen. Bei funktionsfreien Termen gibt es nichts zu tun. Enthält ein Term t Funktionen, dann unterscheiden wir zwei Fälle.

1. Hat t die Form $f(t_1, \ldots, t_n)$, wobei f eine Funktion ist, dann treffen wir eine Aussage der Form:

   „f wird an der Stelle der (Auswertung der) Terme $t_1, \ldots, t_n$ interpretiert zu einem Konstruktorterm, den es nun zu suchen gilt.“

   Beispiel:

$$\mathit{del}(n, \mathit{push}(m, \mathit{st}))$$

$$\text{wird transformiert zu}$$

$$\mathrm{I}(\mathit{del}, \langle n, \mathit{push}(m, \mathit{st})\rangle, \mathsf{val}(\mathit{del}(n, \mathit{push}(m, \mathit{st}))))$$

$$\text{und}$$

$$\mathsf{search_}\mathit{Stack}(\mathsf{val}(\mathit{del}(n, \mathit{push}(m, \mathit{st}))))$$

2. Hat t die Form $c(t_1, \ldots, t_n)$, wobei c ein Konstruktor ist, dann müssen wir (auf dieser Ebene) keine Suche anstoßen, sondern können die folgende Aussage treffen:

   „Die Auswertung von t ist der Konstruktor c angewendet auf die (Auswertungen der) Terme $t_1, \ldots, t_n$.“

   Beispiel:

$$\mathit{push}(m, \mathit{del}(n, \mathit{st}))$$

$$\text{wird transformiert zu}$$

$$\mathsf{is}(\mathsf{val}(\mathit{push}(m, \mathit{del}(n, \mathit{st}))), \mathit{push}(m, \mathsf{val}(\mathit{del}(n, \mathit{st}))))$$

In beiden Fällen muß die Analyse rekursiv auf den Untertermen fortgesetzt werden. Der obere Beispielterm $\mathit{del}(n, \mathit{push}(m, \mathit{st}))$ besitzt nur funktionsfreie Unterterme, dort ist man fertig. Der untere Beispielterm $\mathit{push}(m, \mathit{del}(n, \mathit{st}))$ besitzt einen Unterterm, der nicht funktionsfrei ist, nämlich $\mathit{del}(n, \mathit{st})$. Wendet man die obige Fallunterscheidung auch auf diesen an, dann tritt wieder der erste Fall ein, und

$$del(n, st)$$

wird transformiert zu

$$\mathrm{I}(del, \langle n, st\rangle, \mathsf{val}(del(n, st)))$$

und

$$\mathsf{search_}Stack(\mathsf{val}(del(n, st)))$$

In der Formulierung der beiden Fälle ist die Phrase „Auswertungen der“ eingeklammert, weil es von den Untertermen abhängt, ob sie erst noch „ausgewertet“ werden müssen oder nicht. Statt in der folgenden Definition die entsprechenden Fälle zu unterscheiden, wenden wir die oben definierte Abkürzung $\mathfrak{Rep}$ auf alle Unterterme an.

Definition 3.2.11 (Transformation der Terme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $t \in T_\Sigma$.

1. Falls $t \in FFT_\Sigma$, dann:

$$\mathfrak{TransTerm}_\Sigma(t) \quad = \quad \emptyset$$

2. Falls $t = a$ mit $a \in F_s$, $\alpha(a) = \lambda$, dann:

$$\mathfrak{TransTerm}_\Sigma(t) \quad = \quad \{\ \mathrm{I}(a, \langle\rangle, \mathsf{val}(a))\ ,\ \mathsf{search_}s(\mathsf{val}(a))\ \}$$

3. Falls $t = f(t_1, \ldots, t_n)$ mit $f \in F_s$, $\alpha(f) = s_1 \ldots s_n$, dann:

$$\mathfrak{TransTerm}_\Sigma(t) \quad = \quad \{\ \mathrm{I}(f, \langle \mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, \mathsf{val}(t))\ \} \cup \{\ \mathsf{search_}s(\mathsf{val}(t))\ \} \cup \mathfrak{TransTerm}_\Sigma(t_1) \cup \vdots \cup \mathfrak{TransTerm}_\Sigma(t_n)$$

4. Falls $t = c(t_1, \ldots, t_n)$ mit $c \in C_s$, $\alpha(c) = s_1 \ldots s_n$, *wobei* $t \notin FFT_\Sigma$, dann:

$$\mathfrak{TransTerm}_\Sigma(t) \quad = \quad \{\ \mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)))\ \} \cup \mathfrak{TransTerm}_\Sigma(t_1) \cup \vdots \cup \mathfrak{TransTerm}_\Sigma(t_n)$$

⋆

In dieser Definition werden f, n, s und c syntaktisch eingesetzt. Falls z.B. *del* eine 2-stellige Funktion mit Ergebnissorte *Stack* ist, dann entsteht bei der Transformation eines durch *del* angeführten Terms (Fall 3) die Syntax:

$$\mathrm{I}(del, \langle \ldots, \ldots \rangle, \ldots)\ ,\ \mathsf{search_}Stack(\ldots)\ ,\ \ldots$$

$\mathfrak{TransTerm}_\Sigma$ ist auf T_Σ wohldefiniert, da für jeden Term $t \in T_\Sigma$ genau einer der drei folgenden Fälle zutrifft:

1. Entweder t enthält keine Funktionssymbole ($\in FFT_\Sigma$), oder
2. das führende Symbol von t ist ein Funktionssymbol, oder
3. das führende Symbol von t ist ein Konstruktor *und* mindestens ein Argumentterm von t enthält Funktionssymbole ($\{t_1, \ldots, t_n\} \not\subseteq FFT_\Sigma$).

Man beachte, daß $\mathfrak{TransTerm}_\Sigma(t)$ genau dann $\emptyset$ liefert, wenn $\mathfrak{Rep}(t) = t$ ist. Wenn also im Resultat von $\mathfrak{TransLit}_\Sigma$ ein Argument von same bzw. $\mathsf{different}$ durch sich selbst repräsentiert wird, dann ist die entsprechende Anwendung von $\mathfrak{TransTerm}_\Sigma$ redundant, was hier modelliert wird durch den Rückgabewert $\emptyset$.

In den anderen Fällen, in denen $\mathfrak{TransLit}_\Sigma$ den ‘Platzhalter’ $\mathsf{val}(t)$ als Argument von same bzw. $\mathsf{different}$ einführt, stellt $\mathfrak{TransTerm}_\Sigma(t)$ weitere Informationen über eben diesen Platzhalter $\mathsf{val}(t)$ bereit.

Zum Zweck der exemplarischen Transformation eines Axioms setzen wir auf die vorangegangene Beispiel-Transformation der beiden Terme $del(n, push(m, st))$ und $push(m, del(n, st))$ auf. Diese Terme sind Bestandteil des folgenden Axiomes ax:

$$ax \qquad = \qquad n \doteq m \ \vee\ del(n, push(m, st)) \doteq push(m, del(n, st))$$

ax ist die normalisierte Variante eines Axioms der Spezifikation NatStack (vgl. S. 26). Die einzelnen Bestandteile der Transformation dieses Axioms sind aus der obigen Diskussion schon bekannt. Zusammengefaßt lautet das Ergebnis von $\mathfrak{TransAxiom}_\Sigma(ax)$:

$$\begin{gathered} Nat(n),\ Nat(m),\ Stack(st) \\ \rightarrow \\ \mathsf{same}(n, m)\ ; \\ \mathsf{same}(\mathsf{val}(del(n, push(m, st))),\ \mathsf{val}(push(m, del(n, st)))), \\ \mathrm{I}(del, \langle n, push(m, st)\rangle, \mathsf{val}(del(n, push(m, st)))), \\ \mathsf{search_}Stack(\mathsf{val}(del(n, push(m, st)))), \\ \mathsf{is}(\mathsf{val}(push(m, del(n, st))), push(m, \mathsf{val}(del(n, st)))), \\ \mathrm{I}(del, \langle n, st\rangle, \mathsf{val}(del(n, st))), \\ \mathsf{search_}Stack(\mathsf{val}(del(n, st)))\ . \end{gathered}$$

Das Ergebnis der Transformation ist recht länglich. Es sollte dabei nicht vergessen werden, daß sowohl die Transformation als auch die Ausführung des Ergebnisses automatisch geschieht. Aber auch dem automatischen Verfahren kommt die Optimierung sehr zugute, die in den einführenden Beispielen teilweise schon verwendet wurde und im folgenden Abschnitt definiert wird.

3.2.3 Optimierte Transformation der Literale

In der bis hierher definierten Transformation wird für jeden Term t, der mit einem Funktionssymbol beginnt, ein Platzhalter `val(t)` eingeführt, nach dessen Wert später gesucht werden muß ('$\mathsf{search_}s(\mathsf{val}(t))$'). Infolgedessen werden auch bei einer Gleichung $t_1 \doteq t_2$, bei der beide Terme t_1 und t_2 mit einem Funktionssymbol beginnen, *zwei* Platzhalter $\mathsf{val}(t_1)$ und $\mathsf{val}(t_2)$ eingesetzt, nach deren Wert dann getrennt gesucht wird.

Dies ist zwar korrekt, da ungleiche Kombinationen mittels '`same` ' wieder verworfen werden (s. Def. 3.2.5), aber der Aufwand ist unnötig groß. Es wäre besser, nur *einen* Platzhalter einzuführen, der *beide* Terme repräsentiert (dies ist semantisch gerechtfertigt durch die Gleichheit '$\doteq$'), um so nur *eine* Suche zu initiieren. Auch in dem Fall, daß bei einer Gleichung $t_1 \doteq t_2$ nur einer der beiden Terme t_1 und t_2 mit einem Funktionssymbol beginnt, kann ein Platzhalter (und die Suche nach entprechenden Werten) eingespart werden. In beiden Fällen kann man den Repräsentanten eines mit einem Funktionssymbol beginnenden Terms, z.B. $\mathfrak{Rep}(t_1)$, durch den Repräsentanten des anderen Terms, in diesem Fall $\mathfrak{Rep}(t_2)$, ersetzen (unabhängig davon, ob auch t_2 mit einem Funktionssymbol beginnt). Der 'Vergleich' mittels `same` kann dann gänzlich entfallen.

Einige kleine Beispiele hierfür hatten wir schon in Abschnitt 3.2.2 gesehen. Betrachten wir, noch vor einer Definition, die optimierte Transformation des gerade eben behandelten Axioms ax. D.h. es gilt wieder

$$ax \quad = \quad n \doteq m \;\vee\; del(n, push(m, st)) \doteq push(m, del(n, st))$$

Oftmals, so auch hier, wird nur einer der beiden Terme beidseits der Gleichung von einem Funktionssymbol angeführt. Nur dieser Term verursacht (schon auf oberster Ebene) die Suche nach einem Konstruktorterm (Fall 2. und 3. der Definition 3.2.11). Darum wird vorzugsweise der zu diesem Term gehörende Platzhalter wegoptimiert.

Die optimierte Transformation von ax ist gegeben durch:

$$\begin{gathered} Nat(n),\ Nat(m),\ Stack(st) \\ \rightarrow \\ \mathsf{same}(n,m)\ ; \\ \mathrm{I}(del, \langle n, push(m,st)\rangle, \mathsf{val}(push(m, del(n,st)))), \\ \mathsf{is}(\mathsf{val}(push(m, del(n,st))), push(m, \mathsf{val}(del(n,st)))), \\ \mathrm{I}(del, \langle n, st\rangle, \mathsf{val}(del(n,st))), \\ \mathsf{search_}Stack(\mathsf{val}(del(n,st)))\ . \end{gathered}$$

Es folgt die formale Definition der optimierten Transformation. Was im Lichte der Beispiele als Vereinfachung wirkt, führt auf der Seite der präzisen Definition zu einer Verkomplizierung. Darum sei der Leser/ die Leserin ermuntert, die folgende Definition eher flüchtig zu lesen. Man kann sich hier mit dem Eindruck zufrieden geben, daß im Vergleich zu den Definitionen des letzten Abschnitts die Ebene der Literale-Transformation und diejenige der Term-Transformation vermischt werden, und das auch nur in bestimmten Fällen. Diese Fälle treten aber in üblichen Axiomenmengen sehr häufig auf!

Definition 3.2.12 (Optimierte Transformation der Literale)
Aus Abschnitt 3.2.2 werden die Definitionen von $\mathfrak{Rep}$, $\mathfrak{TransAxiom}_\Sigma$ und $\mathfrak{TransTerm}_\Sigma$ ohne Änderung übernommen. Ebenfalls übernommen wird die Definition von $\mathfrak{TransLit}_\Sigma$ auf *Ungleichungen* ($\mathfrak{TransLit}_\Sigma(t_1 \neq t_2)$).

Auf *Gleichungen* ist $\mathfrak{TransLit}_\Sigma$ wie folgt definiert:
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $t_1, t_2 \in T_s$ für ein $s \in S$.

1. Falls $t_1 = f(t_{11}, \ldots, t_{1n})$, mit $f \in F_s$, dann:

$$\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2) = \begin{gathered} \{\mathrm{I}(f, \langle \mathfrak{Rep}(t_{11}), \ldots, \mathfrak{Rep}(t_{1n})\rangle, \mathfrak{Rep}(t_2))\} \\ \cup \\ \mathfrak{TransTerm}_\Sigma(t_{11}) \\ \cup \\ \vdots \\ \cup \\ \mathfrak{TransTerm}_\Sigma(t_{1n}) \\ \cup \\ \mathfrak{TransTerm}_\Sigma(t_2) \end{gathered}$$

2. Falls a) $t_1 \in V_s$ *oder* $t_1 = c(t_{11}, \ldots, t_{1n})$ mit $c \in C_s$,
 und
 b) $t_2 = f(t_{21}, \ldots, t_{2m})$ mit $f \in F_s$,
 dann:

$$\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2) = \mathfrak{TransLit}_\Sigma(t_2 \doteq t_1)$$

3. Falls a) $t_1 \in V_s$ *oder* $t_1 = c(t_{11}, \ldots, t_{1n})$ mit $c \in C_s$,
   und
   b) $t_2 \in V_s$ *oder* $t_2 = c'(t_{21}, \ldots, t_{2n})$ mit $c' \in C_s$,
   dann:

$$\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2) = \{\mathsf{same}(\mathfrak{Rep}(t_1), \mathfrak{Rep}(t_2))\} \cup \mathfrak{TransTerm}_\Sigma(t_1) \cup \mathfrak{TransTerm}_\Sigma(t_2)$$

⋆

Der 3. Fall, in dem keiner der beiden Terme mit einem Funktionssymbol beginnt, ist identisch mit der unoptimierten Definition von $\mathfrak{TransLit}_\Sigma$.

Im 1. Fall, in dem t_1 mit einem Funktionssymbol beginnt, läßt sich die Definition als Abkürzung der folgenden Prozedur auffassen: Man wendet $\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2)$ so an wie in Fall 3 beschrieben. In dem Ergebnis werden $\mathfrak{Rep}(t_1)$, $\mathfrak{Rep}(t_2)$ und $\mathfrak{TransTerm}_\Sigma(t_2)$ zunächst nicht expandiert, sondern lediglich $\mathfrak{TransTerm}_\Sigma(t_1)$ (gemäß Def. 3.2.11, Fall 2. un 3.). Danach wird überhaupt nicht weiter expandiert. Stattdessen wird nun überall $\mathfrak{Rep}(t_1)$ durch $\mathfrak{Rep}(t_2)$ ersetzt (was zulässig ist wegen '$t_1 \doteq t_2$'). Jetzt lautet das erste Konjunktionsglied $\mathsf{same}(\mathfrak{Rep}(t_2), \mathfrak{Rep}(t_2))$. Dieses ist selbstverständlich redundant und kann weggelassen werden. Ebenfalls redundant ist nun $\mathsf{search_}s(\mathfrak{Rep}(t_1))$, da $\mathfrak{Rep}(t_1)$ gar nicht mehr vorkommt. Dieses Konjunktionsglied wird ebenfalls weggelassen. Das Endergebnis dieser Prozedur entspricht genau der Definition von Fall 1.

Fall 2 besagt, daß die Terme vertauscht werden, falls nur der linke mit einem Funktionssymbol anfängt.

3.2.4 Transformation der Spezifikation

Nachdem nun definiert wurde, inwiefern erstens aus der Signatur und zweitens aus den Axiomen einer Spezifikation Regeln zur Generierung von Modellen hervorgehen, fehlen uns nur noch zwei Regeln zur Komplettierung der Spezifikations-Transformation (abgesehen von der Frage der Instantiierung, die wir seperat behandeln, s.u.). Die noch fehlenden Regeln haben die Aufgabe, sicherzustellen, daß sowohl die I-Atome als auch die is-Atome rechtseindeutig, d.h. *funktional* sind. Die folgende Menge von Regeln ist unabhängig von Σ oder AX.

Definition 3.2.13 (Regeln für die Funktionalität)
Die beiden Regeln, die für saturierte Äste des Modell-Generierungs-Baumes die Funktionalität des I- und des is-Atoms garantieren, sind gegeben durch:

$$\mathfrak{Funktionalitaet} \;=\; \{\; \mathrm{I}(fv, tv, z),\ \mathrm{I}(fv, tv, z') \;\rightarrow\; \mathsf{same}(z, z')\ .\ , \quad \mathsf{is}(x, y),\ \mathsf{is}(x, y') \;\rightarrow\; \mathsf{same}(y, y')\ .\ \}$$

⋆

Somit können Spezifikationen komplett transformiert werden.

Definition 3.2.14 (Transformation einer Spezifikation)
Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation. Die Modell-Generierungs-Regeln für diese Spezifikation sind definiert durch:

$$\mathfrak{TransSpec}(\langle \Sigma, AX \rangle) = \mathfrak{TransSig}(\Sigma) \cup \mathfrak{TransAxioms}_{\Sigma}(AX) \cup \mathfrak{Funktionalitaet}$$

⋆

3.3 Beschränkte Instantiierung der Axiome

Aus der Diskussion des letzten Abschnitts, insbesondere aus der Transformation der Axiome (Def. 3.2.8), ist schon deutlich geworden, daß die transformierten Axiome erst dann '*angewendet*' werden, wenn für jede ihrer freien Variablen eine Instanz 'vorliegt'. Eine Instanz *inst* 'liegt' dann in einem Modell-Kandidaten 'vor', wenn $s(inst)$ Element des Modell-Kandidaten ist (wobei s die Sorte von *inst* ist). In diesem Fall sprechen wir auch davon, daß *inst* in dem Modell-Kandidaten *präsent* ist. Wenn wir dafür sorgen, daß ein Sorten-Atom der Form $s(\cdot)$ immer einen Konstruktorterm zum Argument hat, dann werden die transformierten Axiome im Laufe der Modell-Generierung immer mit Konstruktortermen instantiiert (durch Anwendung einer Axiom-Regel, s. Def. 3.2.8). Um der Semantik von Spezifikationen völlig zu entsprechen, benötigten wir im Prinzip die Präsenz *aller* Konstruktorterme in jedem Modell-Kandidaten. Da es aber i.a. unendlich viele Konstruktorterme gibt, ist die Konstruktion solcher Modell-Kandidaten in endlicher Zeit nicht möglich. Gerade der Fall, der uns am meisten interessiert, in dem nämlich ein Modell der (Gegen-)Spezifikation existiert, würde nicht terminieren. Damit hätten wir die gleiche Situation wie schon bei gewöhnlichen Tableaux.

Wir aber wollen ein Verfahren, welches, angewendet auf erfüllbare (Gegen-)Spezifikationen, auf jeden Fall terminiert. Um dies zu erreichen, werden lediglich *endliche* Instanzen-Mengen *Inst* betrachtet. Der Preis hierfür ist, daß die gefundenen Modelle eigentlich nur Modelle der mit *Inst* instantiierten Axiome sind. Das bedeutet: das Verfahren findet zwar *alle* Modelle, aber möglicherweise *zu viele*. Aus diesem Grund muß der Benutzer/die Benutzerin die gefundenen Modelle selbst abschließend beurteilen. Hierbei wird er/sie unterstützt durch eine Aufbereitung des Ergebnisses, die in Abschnitt 5.1 beschrieben werden wird. Wichtig hierbei ist auch die Möglichkeit, die Modell-Generierung für unterschiedliche (z.B. größer werdende) Instanzen-Mengen durchzuführen und die Ergebnisse zu vergleichen.

3.3.1 Grundinstantiierte Formelmengen

Da das Verfahren, welches hier beschrieben wird, Modelle grundinstantiierter Spezifikationen konstruiert, ist es sowohl für das Verständnis als auch im Hinblick auf die Diskussion der Korrektheit und Vollständigkeit hilfreich, eine Notation zur Verfügung zu stellen, die es erlaubt, in einfacher Weise über instantiierte Formeln und Formelmengen zu sprechen.

Definition 3.3.1 (Formel-Instantiierung mit Konstr.-term-Mengen)
Sei $Inst \subseteq CT_\Sigma$ eine Teilmenge der Konstruktorterme und $\varphi \in For_\Sigma$ mit $frei(\varphi) = \{x_1, \ldots, x_n\}$. Die **Instantiierung von φ mit *Inst***, kurz $[\varphi/Inst]$, ist definiert durch:

$$\psi \in [\varphi/Inst]$$
$$:\Longleftrightarrow$$
$$\psi = \varphi[x_1/ct_1, \ldots, x_n/ct_n] \text{ für Konstruktorterme } ct_1, \ldots, ct_n$$
$$\text{mit } ct_i \in Inst \text{ und } sort(ct_i) = sort(x_i)$$

⋆

Faktum 3.3.2 *$[\varphi/Inst]$ sind geschlossene Formeln: $[\varphi/Inst] \subseteq For^0_\Sigma$*

Satz 3.3.3 *Falls Inst endlich ist, dann ist auch $[\varphi/Inst]$ endlich.*

Beweis: Unter *Nichtbeachtung* der Wohlsortiertheit kann man nicht mehr als $|frei(\varphi)|^{|Inst|}$ Formeln der Form $\varphi[x_1/ct_1, \ldots, x_n/ct_n]$ konstruieren, d.h. es gilt $|[\varphi/Inst]| \leq |frei(\varphi)|^{|Inst|}$. □

Nun gilt es, die Beziehung zwischen Formeln und ihren Instantiierungen zu beleuchten. Tatsächlich ist jede Formel φ äquivalent zu der Menge ihrer Instantiierungen mit allen Konstruktortermen, $[\varphi/CT_\Sigma]$. Dies gilt aber *nur* deshalb, weil wir eine termerzeugte Semantik zugrunde legen. Den entsprechenden Satz beweist man durch Induktion über die Anzahl der freien Variablen in φ. Im Induktionsschritt benötigt man das folgende Lemma:

Lemma 3.3.4 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur, sei $\varphi \in For_\Sigma$ und $x \in V_\Sigma$ mit $sort(x) = s$. Sei außerdem $\Psi = \{\varphi[x/ct] \mid ct \in CT_s\}$.*
Dann sind $\{\varphi\}$ und Ψ Σ-äquivalent.

Beweis: Wie zeigen: Ist $\mathcal{I}$ eine $\mathcal{F}$-Interpretation, dann gilt $\mathcal{I} \models \Psi \Longleftrightarrow \mathcal{I} \models \varphi$.

$\mathcal{I} \models \Psi$
$\Longleftrightarrow$
f.a. $ct \in CT_s$ gilt $\mathcal{I} \models \varphi[x/ct]$

$\Longleftrightarrow$
f.a. $ct \in CT_s$ gilt, daß f.a. β gilt: $val_{\mathcal{I},\beta}(\varphi[x/ct]) = W$
$\Longleftrightarrow$
f.a. β gilt, daß f.a. $ct \in CT_s$ gilt: $val_{\mathcal{I},\beta}(\varphi[x/ct]) = W$
$\Longleftrightarrow$ (Substitutionstheorem f. Formeln, Faktum 2.3.15)
f.a. β gilt, daß f.a. $ct \in CT_s$ gilt: $val_{\mathcal{I},\beta_x^{ct}}(\varphi) = W$
$\Longleftrightarrow$ (!)
f.a. β gilt: $val_{\mathcal{I},\beta}(\varphi) = W$
$\Longleftrightarrow$
$\mathcal{I} \models \varphi$ □

Satz 3.3.5 *Sei $\varphi \in For_\Sigma$. Dann gilt:*

$$[\varphi/CT_\Sigma] \text{ ist } \Sigma\text{-äquivalent zu } \{\varphi\}.$$

Beweis: Wir zeigen durch Induktion über die Anzahl der freien Variablen in φ, daß $\mathcal{I} \models [\varphi/CT_\Sigma] \Longleftrightarrow \mathcal{I} \models \varphi$. Aus notationellen Gründen betrachten wir zwei Induktionsanfänge.

Ind.-Anfang:
Falls $frei(\varphi) = \emptyset$, dann ist die Beh. trivial.
Falls $frei(\varphi) = \{x\}$, dann folgt die Beh. aus Lemma 3.3.4.

Ind.-Schritt:
Sei $frei(\varphi) = \{x_1, \ldots, x_n\}$ mit $sort(x_n) = s$, dann gilt die folgende Kette von Äquivalenzen:

$\mathcal{I} \models [\varphi/CT_\Sigma]$
$\Longleftrightarrow$
$\mathcal{I} \models \varphi[x_1/ct_1, \ldots, x_n/ct_n]$ f.a. Konstruktorterme $ct_1, \ldots, ct_n$ mit $ct_i \in Inst$ und $sort(ct_i) = sort(x_i)$
$\Longleftrightarrow$
f.a. Konstruktorterme $ct_1, \ldots, ct_n$ mit $ct_i \in Inst$ und $sort(ct_i) = sort(x_i)$ gilt: $\mathcal{I} \models \varphi[x_1/ct_1, \ldots, x_{n-1}/ct_{n-1}][x_n/ct_n]$
$\Longleftrightarrow$ (wg. Lemma 3.3.4)
f.a. Konstruktorterme $ct_1, \ldots, ct_{\underline{n-1}}$ mit $ct_i \in Inst$ und $sort(ct_i) = sort(x_i)$ gilt: $\mathcal{I} \models \varphi[x_1/ct_1, \ldots, x_{n-1}/ct_{n-1}]$
$\Longleftrightarrow$ (wg. Ind.-Hyp.)
$\mathcal{I} \models \varphi$ □

Da wir uns eigentlich für die Instantiierung einer Formelmenge, nämlich der Axiome, interessieren, folgen nun die entsprechende Definition und der entsprechende Satz für Formelmengen.

Definition 3.3.6 (Instantiierung von Formelmengen)
Sei $Inst \subseteq CT_\Sigma$ eine Teilmenge der Konstruktorterme und $\Phi \subseteq For_\Sigma$. Die **Instantiierung von Φ mit *Inst***, kurz $[\Phi/Inst]$, ist definiert durch:

$$[\Phi/\mathit{Inst}] = \bigcup_{\varphi\in\Phi}[\varphi/\mathit{Inst}]$$

⋆

Aus dieser Definition und Satz 3.3.3 folgt:

Korollar 3.3.7 *Sind $\Phi \subseteq For_\Sigma$ und $Inst \subseteq CT_\Sigma$ beide endlich, dann ist auch $[\Phi/Inst]$ endlich.*

Satz 3.3.8 *Sei $\Phi \subseteq For_\Sigma$. Dann gilt:*
$[\Phi/CT_\Sigma]$ ist Σ-äquivalent zu Φ.

Beweis: Wir zeigen, daß $\mathcal{I} \models \Phi \Longleftrightarrow \mathcal{I} \models [\Phi/CT_\Sigma]$ gilt.

$\mathcal{I} \models \Phi$
$\Longleftrightarrow$
f.a. $\varphi \in \Phi$ gilt: $\mathcal{I} \models \varphi$
$\Longleftrightarrow$ (Satz 3.3.5)
f.a. $\varphi \in \Phi$ gilt: $\mathcal{I} \models [\varphi/CT_\Sigma]$
$\Longleftrightarrow$
$\mathcal{I} \models \bigcup_{\varphi\in\Phi}[\varphi/CT_\Sigma]$
$\Longleftrightarrow$
$\mathcal{I} \models [\Phi/CT_\Sigma]$ □

Korollar 3.3.9 *Sei $\Phi \subseteq For_\Sigma$ und $Inst \subseteq CT_\Sigma$. Dann gilt:*
$$\Phi \models_\Sigma [\Phi/\mathit{Inst}]$$

Wir haben also gezeigt, daß man die Axiome theoretisch austauschen könnte gegen unendlich viele grundinstantiierte Formeln. Unser Verfahren benutzt natürlich nur endlich viele Instanzen, praktisch sogar ziemlich wenige. Dennoch ist es wichtig, daß wir uns darüber klar geworden sind, welche Bedeutung die gesamte Menge $[\Phi/CT_\Sigma]$ hat, deren endliche Teilmengen $[\Phi/Inst]$ wir behandeln können.

Für theoretische Zwecke ist die Betrachtung allgemeiner Instanzen-Mengen nützlich. Für den realen Einsatz jedoch ist es besser, auf einfache Weise interessierende Mengen von Instanzen benennen zu können. Wir tun dies durch Angabe einer Obergrenze für die Anzahl der Konstruktoren in den Instanzen.

Definition 3.3.10 (größenbeschränkte Instantiierung von Formeln)
Sei $\varphi \in For_\Sigma$, $\Phi \subseteq For_\Sigma$ und $n \in \mathbb{N}$. Die **Instantiierung von φ bzw. Φ mit Konstruktortermen der maximalen Größe** n, kurz $\varphi_{\leq n}$ bzw. $\Phi_{\leq n}$, ist definiert durch:[4]

[4]Für $|ct|$ s. Def. 2.2.14.

$$\varphi_{\leq n} = [\varphi/\{ct \in CT_\Sigma \mid |ct| \leq n\}]$$
bzw.
$$\Phi_{\leq n} = \bigcup_{\varphi \in \Phi} \varphi_{\leq n}$$

⋆

Das realisierte Verfahren konstruiert, bei gegebener Spezifikation $\langle \Sigma, AX \rangle$ und gegebener Obergrenze n, ein Modell für $\langle \Sigma, AX_{\leq n} \rangle$ (falls $\langle \Sigma, AX_{\leq n} \rangle$ erfüllbar ist). Für die Theorie betrachten wir ganz allgemein die Konstruktion von Modellen von $\langle \Sigma, [AX/\mathit{Inst}] \rangle$ (mit beliebige Instanzen-Mengen *Inst*). In beiden Fällen benötigen wir MG-Regeln, welche die jeweiligen Instanzen in jedem Modellkandidaten bereitstellen. Diese Regeln folgen im kommenden Abschnitt.

3.3.2 Transformation der Instanzen

Um die Frage der Instantiierung möglichst modular zu der allgemeinen Behandlung von Signaturen und Axiomen zu halten, betrachten wir neben der Transformation $\mathfrak{TransSpec}$, die nur die Spezifikation berücksichtigt, eine separate Transformation für die Bereitstellung von Instanzen in der Modellgenerierung.

Dazu formulieren wir für theoretische Zwecke (Korrektheit, Vollständigkeit) zunächst eine sehr allgemeine Transformation $\mathfrak{TransInst}_\Sigma$ von vorgegebenen Instanzen-Mengen $\mathit{Inst} \subseteq CT_\Sigma$ in entsprechende Modell-Generierungs-Regeln. Als Regelmenge für die Modell-Generierungs-Prozedur dient dann zunächst:

$$\boxed{R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(\mathit{Inst})}$$

Definition 3.3.11 (Transformation einer Instanzen-Menge)
Sei Σ eine ADT-Signatur und sei $\mathit{Inst} \subseteq CT_\Sigma$ eine endliche Menge von Konstruktortermen. Die Transformation der Instanzen-Menge *Inst* in Modell-Generierungs-Regeln ist definiert durch:

$$r \in \mathfrak{TransInst}_\Sigma(\mathit{Inst})$$
$$:\Longleftrightarrow$$
$$r = \ \rightarrow\ s(ct)\ .\ \text{für ein } ct \in CT_s \text{ mit } ct \in \mathit{Inst}.$$

⋆

Wenn für diese Regelmenge R die zentralen Sätze abgeleitet sind, dann sind sie auch unmittelbar anwendbar auf eine für den praktischen Einsatz besser geeignete Verfeinerung von $\mathfrak{TransInst}_\Sigma$, die wir $\mathfrak{MaxInst}_\Sigma$ nennen. $\mathfrak{MaxInst}_\Sigma$ erhält nun noch eine natürliche Zahl als Parameter, die als Beschränkung der Größe betrachteter Instanzen dient. Somit entfällt die Notwendigkeit, Instanzen-Mengen erst zu bilden und dann als Ganzes der Transformation $\mathfrak{TransInst}_\Sigma$ zu übergeben. Die zentralen Sätze sind dann unmittelbar auf $\mathfrak{MaxInst}_\Sigma$ anwendbar.

Definition 3.3.12 (Transformation einer Größen-Beschränkung für Instanzen)
Sei Σ eine ADT-Signatur und $n \in \mathbb{N}$. Die Transformation der Größen-Beschränkung n für Instanzen ist gegeben durch:

$$r \in \mathfrak{MaxInst}_\Sigma(n)$$
$$:\Longleftrightarrow$$
$$r = \ \rightarrow \ s(ct)\ .\ \text{für ein } ct \in CT_s \text{ mit } |ct| \leq n.$$

⋆

3.4 Anwendung der Modell-Generierungs-Regeln

3.4.1 Bestandteile von Modell-Generierungs-Regeln

Nachdem wir nun die Regeln zur Generierung von Modellen schon eingeführt haben, ohne ihre Bestandteile formal zu definieren, ist es im Hinblick auf den weiteren Verlauf notwendig, präziser darauf einzugehen, wie solche Regeln aufgebaut sind und aus welchen Bestandteilen sie bestehen. Da der später zu beschreibende Mechanismus zur 'Abarbeitung' der Regeln ganz wesentlich auf dem Konzept der Substitution basiert, verallgemeinern wir hier dieses schon bekannte Konzept, um es auch auf die verwendeten 'Meta-Terme' (z.B. $\mathsf{val}(t)$) und 'Meta-Atome' (z.B. $\mathrm{I}(f, \langle t\rangle, t')$) anwenden zu können.

Die in den Regeln verwendeten Terme weisen über die Bestandteile von T_Σ hinaus zwei zusätzliche Bauelemente auf: erstens val, was im intuitiven Sinne für die 'noch nicht ausgewertete' Auswertefunktion $val_\mathcal{I}$ steht, zweitens $\mathsf{arg}n$, was im intuitiven Sinne für die Selektion von Argumenten aus einem Term steht (vgl. Def. 2.2.13). Das n in $\mathsf{arg}n$ ist eine natürliche Zahl kleiner gleich der maximalen Konstruktor-Stelligkeit der betrachteten Signatur.

Definition 3.4.1 (maximale Konstruktor-Stelligkeit)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die **maximale Stelligkeit der Konstruktoren** in Σ, $max\alpha(\Sigma)$, ist definiert durch:

$$max\alpha(\Sigma) = max(\{|\alpha(c)| \mid c \in \overline{\mathcal{C}}\})$$

⋆

Damit können wir nun Meta-Terme definieren, bestehend aus Variablen, Konstruktoren, Funktionen, val und $\mathsf{arg}n$. Strukturelle Bedingungen an die Meta-Terme, wie sie im Abschnitt 3.2 besprochen wurden (z.B. die notwendige Kapselung von Funktionstermen mittels val), sind zunächst noch nicht berücksichtigt.

Definition 3.4.2 (Meta-Terme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die Menge der **Meta-Terme**, MT_Σ, ist minimal definiert durch:

- $V_\Sigma \subseteq MT_\Sigma$,
- falls $l \in \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$ und $\alpha(l) = \lambda$, dann $l \in MT_\Sigma$,
- falls $l \in \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$, $|\alpha(l)| = n$ und $\{t_1, \ldots, t_n\} \subseteq MT_\Sigma$, dann $l(t_1, \ldots, t_n) \in MT_\Sigma$,
- falls $t \in MT_\Sigma$, dann $\mathsf{val}(t) \in MT_\Sigma$,
- falls $t \in MT_\Sigma$ und $n \in \{1, \ldots, max\alpha(\Sigma)\}$, dann $\mathsf{arg}n(t) \in MT_\Sigma$.

⋆

Definition 3.4.3 (auftretende Variablen, Grund-Meta-Terme)
Auch für Meta-Terme $t \in MT_\Sigma$ ist $Var(t)$ die Menge der darin auftretenden Variablen, definiert wie in Def. 2.2.15 und zusätzlich:

$$Var(\mathsf{arg}n(t)) = Var(\mathsf{val}(t)) = Var(t)$$

Die Menge der **Grund-Meta-Terme** MT^0_Σ ist definiert durch:

$$MT^0_\Sigma = \{t \in MT_\Sigma \mid Var(t) = \emptyset\}$$

⋆

Die Definition der Meta-Terme berücksichtigt keine Sorten und ist auch sonst ist sehr viel allgemeiner als die Form der tatsächlich bei der Modellkonstruktion auftretenden Meta-Terme, die einem spezielleren, nicht ganz simplen Muster folgen. Der Grund, die Terme nicht gleich nach dem Muster zu definieren, ist, daß die Substitution dann komplizierter zu definieren ist. Man bräuchte dann so etwas wie zulässige Substitutionen, die das Muster erhalten. Stattdessen lassen wir allgemeinere Terme zu und erhalten das Muster als Invariante bei der Modellkonstruktion, was dann zu beweisen sein wird. Neben den theoretischen Vorteilen ist ein sehr wichtiger Aspekt hierbei, daß wir in der *praktischen Realisierung* ein Werkzeug einsetzen können, das allgemeines Matching verwendet, ohne daß deswegen eine Korrektheitslücke entsteht.

Trotzdem definieren wir schon an dieser Stelle das angesprochene Muster für variablenfreie Meta-Terme, ohne uns aber in den folgenden Definitionen darauf zu stützen. Theoretisch wird es erst wieder aufgegriffen, wenn im Rahmen der Vollständigkeits-Betrachtungen die Befolgung des Musters benötigt und deswegen auch bewiesen wird. Bis dahin dient die folgende Definition dem Verständnis dafür, welche Art von Meta-Termen wirklich auftreten.

Definition 3.4.4 (wohlgeformte Grund-Meta-Terme)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die Menge der **wohlgeformten Grund-Meta-Terme**, wMT_Σ, ist minimal definiert durch:

- ist $t \in T^0_\Sigma \setminus CT_\Sigma$ (d.h. t enthält eine Funktion),
  dann ist $\mathsf{val}(t) \in wMT_\Sigma$,
- ist $\mathsf{val}(t) \in wMT_\Sigma$,
  dann ist $\mathsf{arg}n(t) \in wMT_\Sigma$, für alle $n \in \{1, \ldots, max\alpha(\Sigma)\}$,
- ist $\mathsf{arg}n(t) \in wMT_\Sigma$,
  dann ist $\mathsf{arg}m(\mathsf{arg}n(t)) \in wMT_\Sigma$, für alle $m \in \{1, \ldots, max\alpha(\Sigma)\}$,
- ist $c \in \overline{\mathcal{C}}$, mit $\alpha(c) = \lambda$, dann ist $c \in wMT_\Sigma$,
- ist $\{t_1, \ldots, t_n\} \subseteq wMT_\Sigma$ und ist $c \in \overline{\mathcal{C}}$, $|\alpha(c)| = n$,
  dann ist $c(t_1, \ldots, t_n) \in wMT_\Sigma$.

Die Elemente von wMT_Σ bezeichnen wir als **wohlgeformte Grund-Meta-Terme**, oder etwas kürzer auch als wohlgeformte Meta-Terme. $\star$

Korollar 3.4.5 *Es gilt:*

$$CT_\Sigma = (FFT_\Sigma \cap T^0_\Sigma) \subseteq wMT_\Sigma \subseteq MT^0_\Sigma \subseteq MT_\Sigma$$

In wohlgeformten Meta-Termen treten Funktionen nur innerhalb von $\mathsf{val}(\cdot)$ auf. Das Argument von $\mathsf{val}(\cdot)$ *muß* sogar ein Funktionsterm sein! val hat gerade diesen Sinn, Funktionsterme zu kapseln, weil man auf der Meta-Ebene über ihre 'Auswertung' redet. Reine Konstruktorterme bleiben ungekapselt, da sie mit ihrer eigenen Auswertung übereinstimmen. Darum betrachten wir sie gleichzeitig als Objekt- und als Meta-Terme.

Die Definition der Wohlgeformtheit läßt immer noch mehr Terme zu als tatsächlich auftreten. Z.B. werden die Sorten nicht per Definition beachtet. Dies wäre auch nicht lokal für Meta-Terme definierbar. Beispielsweise kann die 'Sorte' von $\mathsf{arg1}(\mathsf{val}(t))$ von anderen Meta-Atomen abhängen, aus denen hervorgeht, zu welchem Konstruktorterm t evaluiert.

(Wie schon angekündigt, setzten die nun folgenden Definitionen die Wohlgeformtheit von Meta-Termen nicht voraus.)

Als nächstes wollen wir, aufbauend auf den Meta-Termen, die in den Regeln verwendeten Meta-Atome definieren, wie z.B. is- oder I-Atome. Speziell für die I-Atome benötigen wir aber noch andere Bestandteile als bloße Meta-Terme. Für das erste Argument muß es die Möglichkeit geben, Funktionen ohne Argumente selbst als Argumente des I-Atoms zu verwenden. Für das zweite Argument ist

die Bildung von Tupeln vonnöten. Ein Blick z.B. auf die Regel für die Funktionalität von I, '$\mathrm{I}(fv, tv, z),\ \mathrm{I}(fv, tv, z') \ \rightarrow\ \mathsf{same}(z, z')$.', zeigt, daß wir sowohl für Funktionen als auch für Tupel sogar noch Variablen brauchen. Allerdings kommen wir mit jeweils einer aus, nämlich mit *fv* als Funktionsvariable und *tv* als Tupel-Variable.

Definition 3.4.6 (Funktions-Ausdrücke)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur.
fv ist eine (die einzige) **Funktionsvariable** ($fv \notin V_\Sigma \cup \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$).
$FA_\Sigma = \{fv\} \cup \overline{\mathcal{F}}$ ist die Menge der (Σ)-**Funktionsausdrücke**.
Die Menge der auftretenden Variablen ist offensichtlich definiert:

$$Var(fa) = \begin{cases} \{fv\} & \text{falls } fa = fv \\ \emptyset & \text{falls } fa \in \overline{\mathcal{F}} \end{cases}$$

⋆

Nun definieren wir Tupel von Termen.

Definition 3.4.7 (Meta-Term-Tupel)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur.
tv ist eine (die einzige) **Tupelvariable** ($tv \notin V_\Sigma \cup \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$).
Die Menge der **Meta-Term-Tupel**, MTT_Σ, ist minimal definiert durch:

- $tv \in MTT_\Sigma$
- $\langle\rangle \in MTT_\Sigma$
- wenn $\{t_1, \ldots, t_n\} \subseteq MT_\Sigma$, dann ist $\langle t_1, \ldots t_n\rangle \in MTT_\Sigma$

Die Menge der auftretenden Variablen ist definiert durch:
$Var(tv) = \{tv\},\ Var(\langle\rangle) = \emptyset,\ Var(\langle t_1, \ldots, t_n\rangle) = Var(t_1) \cup \ldots \cup Var(t_n)$ ⋆

Die Variablen in den Meta-Termen sind im wesentlichen die 'normalen' Variablen aus der Menge V_Σ, zuzüglich der Variablen *fv* und *tv*. All diese Variablen werden in den Regeln verwendet. Wir nennen sie dann 'Meta-Variablen', und zwar aus zweierlei Gründen: erstens, weil sie hier die Rolle von 'Regel-Schema-Variablen' einnehmen, und zweitens, weil sie durch Meta-Terme *substituiert* werden:

Definition 3.4.8 (Meta-Variablen, Meta-Variablen-Substitution)
Sei Σ eine ADT-Signatur. Die Menge der Meta-Variablen MV_Σ ist definiert durch:

$$MV_\Sigma = V_\Sigma \cup \{fv, tv\}$$

Eine Meta-Variablen-Substitution (MV-Substitution) ist eine Abbildung

$$\sigma : MV_\Sigma \rightarrow MT_\Sigma \cup FA_\Sigma \cup MTT_\Sigma,$$

wobei nur für endlich viele Variablen $\sigma(x) \neq x$ gilt, und wobei:

$$\sigma(fv) \in FA_\Sigma,\ \sigma(tv) \in MTT_\Sigma \text{ und } \sigma(x) \in MT_\Sigma, \text{ falls } x \in V_\Sigma.$$

⋆

Man beachte, daß wegen $T_\Sigma \subseteq MT_\Sigma$ auch normale Substitutionen als MV-Substitutionen aufgefaßt werden können (mit $\sigma(fv) = fv$ und $\sigma(tv) = tv$). Diesen Umstand werden wir noch ausnutzen. Die Verwendung normaler Variablen als Meta-Variablen hat u.a. auch den Vorteil, daß die Transformation von Regeln aus Axiomen einer Spezifikation erheblich einfacher wird.

Nun stehen uns alle syntaktischen Zutaten zur Verfügung, die zur Definition von Meta-Atomen benötigt werden.

Definition 3.4.9 (Meta-Atome)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur. Die Menge der **Meta-Atome**, MA_Σ, ist minimal definiert durch:

- wenn $t \in MT_\Sigma$ und $s \in S$, dann ist $\{s(t),\ \mathsf{search_}s(t)\} \subseteq MA_\Sigma$
- wenn $\{t_1, t_2\} \subseteq MT_\Sigma$, dann ist
  $\{\mathsf{is}(t_1, t_2),\ \mathsf{same}(t_1, t_2),\ \mathsf{different}(t_1, t_2)\} \subseteq MA_\Sigma$
- wenn $fa \in FA_\Sigma$ ein Funktionsausdruck, $mtt \in MTT_\Sigma$ ein Meta-Term-Tupel und $t \in MT_\Sigma$, dann ist $\mathrm{I}(fa, mtt, t) \in MA_\Sigma$

Die Menge der in Meta-Atomen auftretenden Variablen ist definiert durch:

- $Var(s(t)) = Var(\mathsf{search_}s(t)) = Var(t)$
- $Var(\mathsf{is}(t_1, t_2)) = Var(\mathsf{same}(t_1, t_2)) = Var(\mathsf{different}(t_1, t_2)) = Var(t_1) \cup Var(t_2)$
- $Var(\mathrm{I}(fa, mtt, t)) = Var(fa) \cup Var(mtt) \cup Var(t)$

Die Menge der **Grund-Meta-Atome**, MA^0_Σ, ist definiert durch:

$$t \in MA^0_\Sigma \quad :\Longleftrightarrow \quad t \in MA_\Sigma \text{ und } Var(t) = \emptyset$$

Die Menge der in Mengen von Meta-Atomen auftretenden Variablen ist definiert durch: Sei $A \subseteq MA$, dann gilt:

$$x \in Var(A) \quad :\Longleftrightarrow \quad x \in Var(at) \text{ für ein } at \in A$$

$\star$

Wir erweitern die Meta-Variablen-Substitutionen auf Meta-Terme und Meta-Atome:

Definition 3.4.10 (Substitution in Meta-Termen und Meta-Atomen)
Sei Σ eine ADT-Signatur und σ eine Meta-Variablen-Substitution. Wir erweitern σ zu einer postfix notierten Abbildung $\overline{\sigma}$ auf Meta-Termen:

- $x\overline{\sigma} = \sigma(x)$ und $l\overline{\sigma} = l$, für $x \in V_\Sigma$ und $l \in \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$

- $(l(t_1,\ldots,t_n))\overline{\sigma} = l(t_1\overline{\sigma},\ldots,t_n\overline{\sigma})$, falls $l(t_1,\ldots,t_n) \in MT_\Sigma$
- $(\mathsf{val}(t))\overline{\sigma} = \mathsf{val}(t\overline{\sigma})$
- $(\mathsf{arg}n(t))\overline{\sigma} = \mathsf{arg}n(t\overline{\sigma})$

Weiterhin erweitern wir $\overline{\sigma}$ weiter zu einer Abbildung auf Meta-Term-Tupeln und Funktionsausdrücken.

- $(tv)\overline{\sigma} = \sigma(tv)$
- $\langle\rangle\overline{\sigma} = \langle\rangle$
- $\langle t_1,\ldots,t_n\rangle\overline{\sigma} = \langle t_1\overline{\sigma},\ldots,t_n\overline{\sigma}\rangle$
- $(fv)\overline{\sigma} = \sigma(fv)$
- $f\overline{\sigma} = f$, falls $f \in \overline{\mathcal{F}}$

Weiterhin erweitern wir $\overline{\sigma}$ weiter zu einer Abbildung auf Meta-Atomen.

- $(s(t))\overline{\sigma} = s(t\overline{\sigma})$, $\quad (\mathsf{search_}s(t))\overline{\sigma} = \mathsf{search_}s(t\overline{\sigma})$
- $(\mathsf{is}(t_1,t_2))\overline{\sigma} = \mathsf{is}(t_1\overline{\sigma},t_2\overline{\sigma})$, $\quad (\mathsf{same}(t_1,t_2))\overline{\sigma} = \mathsf{same}(t_1\overline{\sigma},t_2\overline{\sigma})$, $(\mathsf{different}(t_1,t_2))\overline{\sigma} = \mathsf{different}(t_1\overline{\sigma},t_2\overline{\sigma})$
- $(\mathrm{I}(fa,mtt,t))\overline{\sigma} = \mathrm{I}(fa\overline{\sigma},mtt\overline{\sigma},t\overline{\sigma})$

Schließlich erweitern wir $\overline{\sigma}$ weiter zu einer Abbildung auf Mengen von Meta-Atomen:

$$\text{Sei } \{at_1,\ldots,at_n\} \subseteq MA_\Sigma, \text{ dann ist } \{at_1,\ldots,at_n\}\overline{\sigma} = \{at_1\overline{\sigma},\ldots,at_n\overline{\sigma}\}$$

. ⋆

Nun werden wir die bisher informell eingeführte allgemeine Form von Modell-Generierungs-Regeln präzise definieren. Dies dient vor allen Dingen dem Zweck, die Modell-Generierungs-Prozedur genau definieren zu können. Zunächst wiederholen wir zum besseren Verständnis die schon bekannte, informell eingeführte Struktur von Regeln:

$$\underbrace{at_1,\ldots,at_n}_{\text{Prämisse}} \rightarrow \overbrace{\underbrace{at_{11},\ldots,at_{1n_1}}_{\text{1.Extension}}\ ;\ \ldots\ ;\ \underbrace{at_{m_1},\ldots,at_{mn_m}}_{\text{m.Extension}}}^{\text{Konklusion}}\ .$$

Wir können darauf verzichten, innerhalb einer Prämisse oder innerhalb der Extension eine Reihenfolge der Meta-Atome zu betrachten. Wir behandeln daher

Prämissen und Extensionen als Mengen und vereinbaren, daß in der konkreten Syntax der Regeln die Mengenklammern weggelassen werden können. (Die Behandlung als Menge ist auch von Vorteil bei der Transformation.) Hingegen fassen wir die Konklusion nicht als Menge auf, sondern betrachten hier eine Reihenfolge. Die in der Realisierung benutzte Beweisprozedur 'behandelt' nämlich die Extensionen von links nach rechts, und dies machen wir uns zunutze, um einfachere Modelle zuerst zu konstruieren.

Definition 3.4.11 (Modell-Generierungs-Regeln)
Sei Σ eine ADT-Signatur. Sei Pr eine endliche Menge von Meta-Atomen, $Pr \subseteq MA_\Sigma$, $|Pr| < \infty$. Sei $\{Ext_1, \ldots, Ext_n\}$ bzw. $\{\}$ eine Menge von Meta-Atom-Mengen, $Ext_i \subseteq MA_\Sigma$, $0 < |Ext_i| < \infty$, und es gelte $Var(Ext_1) \cup \ldots \cup Var(Ext_n) \subseteq Var(Pr)$ (Wertebeschränktheit). Dann ist:

$$Pr \ \rightarrow \ Ext_1 \ ; \ \ldots \ ; \ Ext_n \ . \quad \text{bzw.} \quad Pr \ \rightarrow \ .$$

eine Σ-**Modell-Generierungs-Regel**, kurz **MG-Regel**.

Ist $r = Pr \ \rightarrow \ Ext_1 \ ; \ \ldots \ ; \ Ext_n$. eine MG-Regel, dann erhält man die Prämisse sowie die *Menge* der Extensionen von r durch die Abbildungen *prämisse* bzw. *extensionen*, d.h.

$$\textit{prämisse}(r) = Pr \quad \text{und} \quad \textit{extensionen}(r) = \{Ext_1, \ldots, Ext_n\}.$$

⋆

Laut Definition handelt es sich bei Pr und Ext_i formal um Mengen. Wie schon gewohnt lassen wir aber in der *Notation* dieser Mengen innerhalb der Regeln die Mengenklammern weg.

Jetzt können wir feststellen, daß das Ergebnis der in Abschn. 3.2 definierten Transformation tatsächlich eine Menge von MG-Regeln ist.

Faktum 3.4.12 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, $Inst \subseteq CT_\Sigma$ und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$. Dann ist jedes $r \in R$ eine MG-Regel.*

3.4.2 Die Modell-Generierungs-Prozedur

Die Prozedur zur Erzeugung von Modellen (erfüllenden Interpretationen) von ADT-Spezifikationen, die wir nun einführen, wird in der Literatur (z.B. [FH91]) bezeichnet mit dem englischen Begriff 'model generation', und geht zurück auf R. Manthey und F. Bry [MB88].[5] Die beiden zitierten Papiere beschreiben insbesondere Implementierungen dieser Prozedur. Die ältere Implementierung ist das in Prolog geschriebene System 'SATCHMO' (SATisfiability CHecking by MOdel

[5] Die allgemeineren Hyper-Tableaux wurden bereits von F. Brown [Bro78] betrachtet.

generation) [MB88]. Zur systemischen Umsetzung der in der vorliegenden Abhandlung beschriebenen Methode wurde das etwas jüngere und bis zum heutigen Tage intensiv weiter entwickelte System 'MGTP' (model generation theorem prover) [FH91] verwendet, für das seit ca. drei Jahren eine an der Universität von Fukuoka entwickelte, hocheffiziente Re-Implementierung in Java vorliegt [FH98].

Die nun folgende Einführung der Modell-Generierungs-Prozedur ist *in der Darstellung* speziell auf unsere Bedürfnisse abgestimmt. Wir erwähnen hier nur kurz die in der Hauptsache terminologischen Unterschiede. Die von uns bisher und auch weiterhin als 'MG-Regeln' bezeichneten syntaktischen Konstrukte der Form

$$at_1, \ldots, at_n \;\rightarrow\; at_{11}, \ldots, at_{1n_1} \;;\; \ldots \;;\; at_{m_1}, \ldots, at_{mn_m} \;.$$

werden dort als 'range restricted clauses' (wertebeschränkte Klauseln) bezeichnet. Dabei wird folgende Sichtweise eingenommen: diese Klauseln bilden eine Formelmenge, deren Erfüllbarkeit durch 'model generation' überprüft wird. Als 'model' wird hierbei eine widerspruchsfreie Menge von *Grund*-Atomen angesehen, die in einem gewissen Sinne gegenüber den Klauseln *saturiert* ist.

Wir hingegen nehmen eine etwas andere Sichtweise ein. Das obige syntaktische Konstrukt dient der Notation von Regeln eines theoriespezifischen, aus der Spezifikation transformierten Kalküls, und der 'model generation'-Mechanismus nimmt die Rolle einer Beweisprozedur für diesen Kalkül ein. Wir verwenden also die 'range restricted clauses' gewissermaßen als eine deklarative Programmiersprache (vergleichbar mit Prolog), die es uns erlaubt, Regeln eines Kalküls sehr direkt zu notieren und ausführen zu lassen.[6] Den Begriff 'Klausel' verwenden wir außerdem schon auf der Ebene der (normalisierten) Spezifikationen.

Wir vermeiden also in unserem Zusammenhang die Betrachtung verschiedener Formel-Ebenen. Echte Formeln, auch 'Objekt-Formeln' genannt, finden sich bei uns nur in den Spezifikationen. Die Suche nach Modellen dieser Spezifikationen wird nicht durch *Formeln*, sondern durch *Regeln* repräsentiert. Die Regeln sind in der schon erklärten Weise aufgebaut aus *Meta*-Termen und *Meta*-Atomen. Wie schon gesagt, sind die in den Regeln auftretenden Variablen ihrer Natur nach Regel-Schema-Variablen. Die Modell-Generierungs-Prozedur baut intuitiv einen *Baum* auf, dessen Knoten *Grund*-Meta-Atome sind. Die Äste des Baumes bezeichnen wir *nicht direkt* als 'Modell'. Stattdessen sehen wir speziell in den I-Atomen eines Astes eine *Repräsentation* von Modellen der betrachteten Spezifikation. Solche Unterschiede erscheinen auf den ersten Blick subtil und sollen uns nicht weiter beschäftigen. Diese kurze Diskussion sollte lediglich den Zusammenhang mit der Terminologie in der Literatur herstellen.

[6] Der Hauptunterschied zu Prolog ist, daß die Implikation dort operational rückwärts, bei 'model generation' hingegen vorwärts 'ausgeführt' wird. Dies wird im folgenden noch deutlich werden.

Ein weiterer Unterschied ist, daß in der Literatur nur einelementige Extensionen betrachtet werden. Da aber das von uns verwendete Werkzeug MGTP mehrelementige Extensionen unterstützt und wir diese auch dringend brauchen, beschreiben wir sie hier auch auf der Ebene der Theorie.

Definition 3.4.13 (MG-Äste, MG-Bäume)
Sei Σ eine ADT-Signatur.

Ein Σ-**Modell-Generierungs-Ast** A ist eine Menge von Grund-Meta-Atomen, $A \subseteq MA_\Sigma^0$.

Ein Σ-**Modell-Generierungs-Baum** ist entweder eine Sequenz $\lceil A_1 | \ldots | A_n \rceil$ von $n > 0$ MG-Ästen oder die leere Sequenz $\lceil\ \rceil$.

Die *Menge* der Äste eines MG-Baumes B, *äste*(B), ist definiert durch:

$$\textit{äste}(B) \quad = \quad \begin{cases} \emptyset & \text{falls } B = \lceil\ \rceil \\ \{A_1, \ldots, A_n\} & \text{falls } B = \lceil A_1 | \ldots | A_n \rceil \end{cases}$$

Die Menge der Atome eines MG-Baumes B, *atome*(B), ist definiert durch:

$$\textit{atome}(B) \quad = \quad \begin{cases} \emptyset & \text{falls } B = \lceil\ \rceil \\ A_1 \cup \ldots \cup A_n & \text{falls } B = \lceil A_1 | \ldots | A_n \rceil \end{cases}$$

⋆

Rein formal sind diese Bäume keine Bäume, sondern Sequenzen von Mengen. Die Bezeichnung ‘Baum’ dient der Anschauung. Die Zeichen $\lceil$ und $\rceil$ sollen die in der Informatik verbreitete Vorstellung stützen, daß Bäume nach ‘unten’ wachsen bzw. ‘oben’ zusammenlaufen.[7]

In Vorbereitung der eigentlichen Modell-Generierungs-Prozedur wird nun beschrieben, wann ein Ast gegenüber einer Regelmenge R als saturiert gilt, wie ein Ast erweitert werden kann und wann er verworfen werden muß.

Definition 3.4.14 (R-Saturiertheit eines Astes)
Sei R eine Menge von Modell-Generierungs-Regeln und sei $A \subseteq MT_\Sigma^0$ ein MG-Ast.

A ist R-saturiert
$:\Longleftrightarrow$
für jede Regel $r \in R$ und jede Meta-Variablen-Substitution σ gilt:

1. falls $r = Pr \rightarrow .$, dann gilt $(Pr)\sigma \not\subseteq A$.

[7] Bekanntlich wachsen in der Informatik die Bäume nicht in den Himmel.

2. falls $r = Pr \rightarrow Ext_1; \ldots ; Ext_n.$, dann gilt:
   wenn $(Pr)\sigma \subseteq A$, dann gilt für ein $i \in \{1, \ldots, n\}$, daß $Ext_i\sigma \subseteq A$.

⋆

Intuitiv ist ein Ast genau dann R-saturiert, wenn keine Regel aus R mehr auf ihn anwendbar ist. In diesem Fall gibt es entweder keine Regel-Prämisse, die auf den Ast 'matcht'; oder aber es gibt eine Regel mit 'matchender' Prämisse, deren Anwendung *redundant* ist. Eine Regelanwendung ist redundant, wenn *einer* der neu entstehenden Äste mit dem alten Ast übereinstimmt, weil die entsprechende Extension nach Anwendung der Substitution schon Teil des alten Astes ist.

Definition 3.4.15 (Erweiterung eines Astes)
Sei $B = \lceil A_1 | \ldots | A_n \rceil$ ein MG-Baum mit $n > 0$, sei $A_i \in \{A_1, \ldots, A_n\}$, sei $r = Pr \rightarrow Ext_1; \ldots ; Ext_m.$ eine MG-Regel und σ eine Meta-Variablen-Substitution.
Es gilt: falls (Pr)$\sigma \subseteq A_i$, dann ist die **Erweiterung von A_i in B mittels r und σ**, kurz $Erw(A_i, B, r, \sigma)$, definiert durch

$$Erw(A_i, B, r, \sigma) := \lceil A_1 | \ldots | A_{i-1} | A_i \cup (Ext_1)\sigma | \ldots | A_i \cup (Ext_m)\sigma | A_{i+1} | \ldots | A_n \rceil$$

⋆

Definition 3.4.16 (Verwerfen eines Astes)
Sei $B = \lceil A_1 | \ldots | A_n \rceil$ ein MG-Baum mit $n > 0$, sei $A_i \in \{A_1, \ldots, A_n\}$, sei $r = Pr \rightarrow .$ eine MG-Regel und σ eine Meta-Variablen-Substitution.
Es gilt: falls (Pr)$\sigma \subseteq A_i$, dann ist das **Verwerfen des Astes A_i in B mittels r und σ**, kurz $Verw(A_i, B, r, \sigma)$, definiert durch

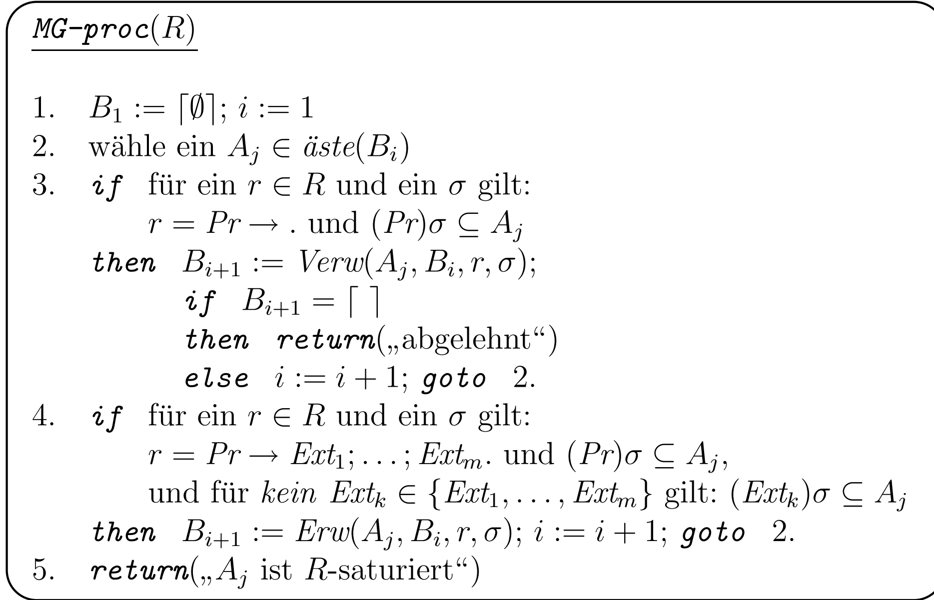
$$Verw(A_i, B, r, \sigma) := \begin{cases} \lceil A_1 | \ldots | A_{i-1} | A_{i+1} | \ldots | A_n \rceil & \text{falls } n > 1 \\ \lceil\ \rceil & \text{falls } n = 1 \end{cases}$$

⋆

Mit Hilfe dieser Operationen auf Bäumen können wir nun die allgemeine Modell-Generierungs-Prozedur angeben.

Definition 3.4.17 (allgemeine Modell-Generierungs-Prozedur)
Die allgemeine Modell-Generierungs-Prozedur ist die in Abb. 3.3 in Pseudo-Code wiedergegebene indeterministische Prozedur ***MG-proc***(R), wobei der Eingabe-Parameter R eine Menge von MG-Regeln ist. ⋆

Der letzte Teil der etwas länglichen ***if***-Bedingung in Schritt 4. von ***MG-proc*** dient dazu, die Modell-Konstruktion dann abzubrechen, wenn die einzige noch mögliche Regelanwendung redundant ist in dem oben beschriebenen Sinne. Dies garantiert die Korrektheit der Antwort „A_j ist R-saturiert“, gemäß dem folgenden Satz.



MG-proc(R)

1. $B_1 := \lceil\emptyset\rceil$; $i := 1$
2. wähle ein $A_j \in \ddot{a}ste(B_i)$
3. ***if*** für ein $r \in R$ und ein σ gilt:
   $r = Pr \rightarrow .$ und $(Pr)\sigma \subseteq A_j$
   then $B_{i+1} := Verw(A_j, B_i, r, \sigma)$;
   if $B_{i+1} = \lceil\ \rceil$
   then ***return***(„abgelehnt")
   else $i := i + 1$; ***goto*** 2.
4. ***if*** für ein $r \in R$ und ein σ gilt:
   $r = Pr \rightarrow Ext_1; \ldots; Ext_m.$ und $(Pr)\sigma \subseteq A_j$,
   und für *kein* $Ext_k \in \{Ext_1, \ldots, Ext_m\}$ gilt: $(Ext_k)\sigma \subseteq A_j$
   then $B_{i+1} := Erw(A_j, B_i, r, \sigma)$; $i := i + 1$; ***goto*** 2.
5. ***return***(„A_j ist R-saturiert")

Abbildung 3.3: die allgemeine Modell-Generierungs-Prozedur

Satz 3.4.18 *Sei R eine Menge von Modell-Generierungs-Regeln. Dann gilt: falls* MG-proc(R) *mit der Ausgabe „A ist R-saturiert" stoppt, dann ist A tatsächlich R-saturiert i.S.d. Def. 3.4.14.*

Beweis: Die einzige Möglichkeit, den Schritt 5. zu erreichen, besteht darin, daß die *if*-Bedingungen in Schritt 3. und 4. verletzt sind. Die Negation dieser Bedingungen entspricht gerade der Definition der R-Saturiertheit. □

Der Zustandsraum, auf dem die Prozedur *MG-proc* operiert, besteht aus den MG-Bäumen. Entsprechend fassen wir als 'Lauf' der Prozedur eine Folge von Bäumen auf.

Definition 3.4.19 (Läufe von *MG-proc*)
Sei R gegeben. Jede bei einer indeterministischen Ausführung von *MG-proc*(R) entstehende, endliche bzw. unendliche Folge $B_1, \ldots, B_e$ bzw. $B_1, B_2, \ldots$ von Bäumen bezeichnet man als (möglichen) **Lauf** von *MG-proc*(R). Der Lauf heißt **terminierend** gdw. er endlich ist. Ensteht bei einer indeterministischen Ausführung von *MG-proc*(R) ein terminierender Lauf L, dann bezeichenen wir die Ausgabe von *MG-proc*(R) auch als **Ausgabe des Laufes** L.
Ein terminierender Lauf $B_1, \ldots, B_e$ **terminiert durch die Saturierung eines Astes**, falls $B_e \neq \lceil\ \rceil$. ⋆

Jeder Lauf, der mit der Saturierung eines Astes terminiert, hat die Ausgabe „A ist R-saturiert", für einen Ast A.

Nun können wir noch, zum Abschluß dieses Kapitels, unsere Methode zur Generierung von Modellen (instantiierter) normalisierter Spezifikationen zusammenfassen, und zwar in einer notationell sehr prägnanten Weise.

Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation und sei *Inst* eine endliche Teilmenge von CT_Σ. Die Suche nach einem Modell von $\langle \Sigma, [AX/Inst] \rangle$ besteht in der Ausführung von:

$$\mathtt{MG\text{-}proc}(\ \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)\ )$$

Für die Realisierung wird später $\langle \Sigma, [AX/Inst] \rangle$ durch $\langle \Sigma, AX_{\leq n} \rangle$ ersetzt, sowie $\mathfrak{TransInst}_\Sigma(Inst)$ durch $\mathfrak{MaxInst}_\Sigma(n)$.

4 Korrektheit und Vollständigkeit

In diesem Kapitel werden die zentralen Eigenschaften der in Kapitel 3 beschriebenen Methode formuliert und bewiesen. Konkret geht es darum, präzise den Zusammenhang zu untersuchen zwischen dem Ergebnis der Modell-Generierungs-Prozedur ***MG-proc*** (angewendet auf die Transformation einer Spezifikation und einer Instanzen-Menge) und der ursprünglichen Spezifikation.

Wir fassen hier kurz und recht informell die Hauptaussagen zusammen, denen dieses Kapitel gewidmet ist. Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation und sei *Inst* eine endliche Teilmenge von CT_Σ. Sei weiterhin R die folgende Menge von MG-Regeln:

$$R \;=\; \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(\mathit{Inst})$$

Dann gilt:

- *beschränkte Modell-Korrektheit:*
  Falls ***MG-proc***(R) mit der Ausgabe „A ist R-saturiert“ terminiert, wobei A ein MG-Ast ist, dann ist die instantiierte Spezifikation $\langle \Sigma, [AX/\mathit{Inst}] \rangle$ erfüllbar. Außerdem kann man aus A eine Interpretation $\mathcal{I}$ ablesen mit $\mathcal{I} \Vdash \langle \Sigma, [AX/\mathit{Inst}] \rangle$.
  (Vgl. Abschn. 4.3.)

- *Unerfüllbarkeits-Korrektheit:*
  Falls ***MG-proc***(R) mit der Ausgabe „abgelehnt“ terminiert, dann ist die Spezifikation $\langle \Sigma, AX \rangle$ unerfüllbar.
  (Vgl. Abschn. 4.4.)

- *Modell-Vollständigkeit:*
  Falls $\langle \Sigma, AX \rangle$ erfüllbar ist, dann terminiert ***MG-proc***(R) mit der Ausgabe „A ist R-saturiert“, und es gilt $\mathcal{I} \Vdash \langle \Sigma, [AX/\mathit{Inst}] \rangle$ für jede Interpretation $\mathcal{I}$, die man aus A ablesen kann. (Dies gilt unabhängig von der Wahl von *Inst*!)
  (Vgl. Abschn. 4.5.)

Die ersten zwei Abschnitte dieses Kapitels dienen der Vorbereitung auf die Beweise dieser Aussagen. Darin werden wichtige Eigenschaften der Transformation sowie von Meta-Atom-Mengen (MG-Ästen) zusammengetragen.

4.1 Eigenschaften der Transformation

Hier formulieren und beweisen wir, noch losgelöst vom Mechanismus der Regelanwendung, einige Eigenschaften der puren Transformation, die später für die Beweise in den Abschnitten 4.3 bis 4.5 ausgenutzt werden. Insbesondere zeigen wir, daß die Transformation (von Termen und Literalen) einerseits und die Meta-Variablen-Substitution andererseits *kommutieren*. Dies ermöglicht es uns, viele zentrale Aussagen als Eigenschaften der Transformation von *Grundtermen* und *Grundliteralen* zu treffen und erst sehr spät auf den Nicht-Grund-Level zu heben.

Wir verweisen nochmals kurz auf die Definitionen, auf die sich die folgenden Aussagen beziehen: $\mathfrak{Rep}$: Def. 3.2.9, S. 85; $\mathfrak{TransLit}_\Sigma$: Def. 3.2.10, S. 85; $\mathfrak{TransTerm}_\Sigma$: Def. 3.2.11, S. 87.

Zunächst weisen wir nach, daß die Transformation eines Grundtermes auch die Transformation seiner Unterterme enthält. Gleiches gilt für Literale.

Definition 4.1.1 (Unterterme von Termen und Literalen)
Die Abbildung $unterTerme : T_\Sigma \cup Lit_\Sigma \to \wp(T_\Sigma)$ ist definiert durch:

- falls $t \in V_\Sigma \cup \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$, dann ist $unterTerme(t) = \{t\}$,
- falls $t = l(t_1, \ldots, t_n) \in T_\Sigma$, dann ist $unterTerme(t) = \{t\} \cup unterTerme(t_1) \cup \ldots \cup unterTerme(t_n)$,
- $unterTerme(t_1 \doteq t_2) = unterTerme(t_1 \not\doteq t_2)$ $= unterTerme(t_1) \cup unterTerme(t_2)$.

⋆

Satz 4.1.2 *Sei $t \in T^0_\Sigma$. Dann gilt f.a. $ut \in unterTerme(t)$:*

$$\mathfrak{TransTerm}_\Sigma(ut) \subseteq \mathfrak{TransTerm}_\Sigma(t)$$

Beweis: Strukturelle Induktion über t:

Ind.-Anfang:
Sei $t \in V_\Sigma \cup \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$, dann ist $unterTerme(t) = \{t\}$, darum $ut = t$.

Ind.-Schritt:
Sei $t = l(t_1, \ldots, t_n) \in T^0_\Sigma$. Wir zeigen zunächst, daß für jedes $t_i \in \{t_1, \ldots, t_n\}$ gilt: $\mathfrak{TransTerm}_\Sigma(t_i) \subseteq \mathfrak{TransTerm}_\Sigma(t)$ (*). Falls $t \notin CT_\Sigma$, dann ergibt sich dies unmittelbar aus der Def. 3.2.11 von $\mathfrak{TransTerm}_\Sigma$ (2.-4.). Falls $t \in CT_\Sigma$, dann ist auch $t_i \in CT_\Sigma$ und $\mathfrak{TransTerm}_\Sigma(t_i) = \mathfrak{TransTerm}_\Sigma(t) = \emptyset$, also gilt wiederum $\mathfrak{TransTerm}_\Sigma(t_i) \subseteq \mathfrak{TransTerm}_\Sigma(t)$. Damit ist (*) gezeigt. Sei nun $ut \in unterTerme(t)$. In dem Falle $ut = t$ ist die Beh. trivial. Sei also $ut \in$

$unterTerme(t_i)$, für ein $t_i \in \{t_1, \ldots, t_n\}$. Laut Ind.-Hyp. gilt $\mathfrak{TransTerm}_\Sigma(ut) \subseteq \mathfrak{TransTerm}_\Sigma(t_i)$. Mit (*) und der Transitivität von $\subseteq$ folgt $\mathfrak{TransTerm}_\Sigma(ut) \subseteq \mathfrak{TransTerm}_\Sigma(t)$. □

Satz 4.1.3 *Sei $lit \in Lit^0_\Sigma$. Dann gilt f.a. $ut \in unterTerme(lit)$:*

$$\mathfrak{TransTerm}_\Sigma(ut) \subseteq \mathfrak{TransLit}_\Sigma(lit)$$

Beweis: $lit \in Lit^0_\Sigma$ impliziert $lit = t_1 \doteq t_2$ oder $lit = t_1 \not\doteq t_2$. In beiden Fällen gilt nach Def. 3.2.10, daß $(\mathfrak{TransTerm}_\Sigma(t_1) \cup \mathfrak{TransTerm}_\Sigma(t_2)) \subseteq \mathfrak{TransLit}_\Sigma(lit)$ (*). Aus $ut \in unterTerme(lit)$ folgt $ut \in (unterTerme(t_1) \cup unterTerme(t_2))$. Mit Satz 4.1.2 folgt $\mathfrak{TransTerm}_\Sigma(ut) \subseteq \mathfrak{TransTerm}_\Sigma(t_1)$ oder $\mathfrak{TransTerm}_\Sigma(ut) \subseteq \mathfrak{TransTerm}_\Sigma(t_2)$. Mit (*) folgt $\mathfrak{TransTerm}_\Sigma(ut) \subseteq \mathfrak{TransLit}_\Sigma(lit)$. □

Im folgenden geht es um die Kommutativität von $\mathfrak{TransTerm}_\Sigma$ (bzw. $\mathfrak{TransLit}_\Sigma$) und Substitutionen. Diese gilt aber nur für solche Substitutionen, welche alle freien Variablen des entsprechenden Terms (bzw. des entsprechenden Literals) durch Konstruktorterme ersetzen, sogenannte 'vollständige Konstruktorterm-Substitutionen'.

Definition 4.1.4 (vollständige Konstruktorterm-Substitutionen)
Sei $t \in T_\Sigma$ und σ eine Substitution zu V_Σ. σ ist eine **vollständige Konstruktorterm-Substitution für** t, falls $\sigma(x) \in CT_\Sigma$, f.a. $x \in Var(t)$. ⋆

Das Ergebnis der Anwendung einer vollständigen Konstruktorterm-Substitution ist ein Grundterm aus T^0_Σ, aber nicht notwendigerweise ein Konstruktorterm, da alle Funktionen, die in t auftreten, auch in $t\sigma$ auftreten.

Satz 4.1.5 *Sei $t \in T_\Sigma$ und sei σ eine vollständige Konstruktorterm-Substitution für t. Dann gilt:*

$$(\mathfrak{TransTerm}_\Sigma(t))\sigma = \mathfrak{TransTerm}_\Sigma(t\sigma)$$

Beweis: Strukturelle Induktion über t:

1. Sei $t = x \in V_\Sigma$. Da $t \in FFT_\Sigma$, ist $\mathfrak{TransTerm}_\Sigma(t) = \emptyset$. Also ist auch $\mathfrak{TransTerm}_\Sigma(t)\sigma = \emptyset$. Andererseits ist $t\sigma = \sigma(x) \in CT_\Sigma$. Da $CT_\Sigma \subseteq FFT_\Sigma$, ist $\mathfrak{TransTerm}_\Sigma(t\sigma) = \emptyset$.

2. Sei $t = a$, $a \in \overline{\mathcal{F}}$ mit $\alpha(a) = \lambda$. Dann ist $Var(\mathfrak{TransTerm}_\Sigma(t)) = \emptyset$, daher gilt: $(\mathfrak{TransTerm}_\Sigma(t))\sigma = \mathfrak{TransTerm}_\Sigma(t) = \mathfrak{TransTerm}_\Sigma(t\sigma)$.

3. Sei $t = c(t_1, \ldots, t_n)$, $c \in \overline{\mathcal{C}}$ mit $\alpha(c) = s_1 \ldots s_n$. Es gilt $(c(t_1, \ldots, t_n))\sigma = c(t_1\sigma, \ldots, t_n\sigma)$ (*). Da σ eine vollst. Konstr.-term-Subst. für t ist, ist σ auch eine vollst. Konstr.-term-Subst. für alle $t_i \in \{t_1, \ldots, t_n\}$. Darum ist die Induktionshypothese auf alle t_i anwendbar. Es gilt:

$$(\mathfrak{TransTerm}_\Sigma(t))\sigma = \begin{array}{c} \{\ (\mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n))))\sigma\ \} \\ \cup \\ (\mathfrak{TransTerm}_\Sigma(t_1))\sigma \\ \cup \\ \vdots \\ \cup \\ (\mathfrak{TransTerm}_\Sigma(t_n))\sigma \end{array}$$

Durch Propagieren von σ in das is-Atom, wegen $(\mathfrak{Rep}(t))\sigma = \mathfrak{Rep}(t\sigma)$ sowie durch Anwendung der Induktions-Hypothese auf $t_i \in \{t_1, \ldots, t_n\}$ folgt:

$$\begin{array}{c} (\mathfrak{TransTerm}_\Sigma(t))\sigma \\ = \\ \{\ \mathsf{is}(\mathsf{val}(t\sigma), c(\mathfrak{Rep}(t_1\sigma), \ldots, \mathfrak{Rep}(t_n\sigma)))\ \} \\ \cup \\ \mathfrak{TransTerm}_\Sigma(t_1\sigma) \\ \cup \\ \vdots \\ \cup \\ \mathfrak{TransTerm}_\Sigma(t_n\sigma) \\ \overset{*}{=} \\ \{\ \mathsf{is}(\mathsf{val}(c(t_1\sigma, \ldots, t_n\sigma)), c(\mathfrak{Rep}(t_1\sigma), \ldots, \mathfrak{Rep}(t_n\sigma)))\ \} \\ \cup \\ \mathfrak{TransTerm}_\Sigma(t_1\sigma) \\ \cup \\ \vdots \\ \cup \\ \mathfrak{TransTerm}_\Sigma(t_n\sigma) \\ = \\ \mathfrak{TransTerm}_\Sigma(c(t_1\sigma, \ldots, t_n\sigma)) \\ \overset{*}{=} \\ \mathfrak{TransTerm}_\Sigma(t\sigma) \end{array}$$

4. Der Fall $t = f(t_1, \ldots, t_n)$, $f \in \overline{\mathcal{F}}$ mit $\alpha(f) = s_1 \ldots s_n$ verläuft völlig analog zum letzten Fall.

□

Nun zeigen wir die Kommutativität von $\mathfrak{TransLit}_\Sigma$ und vollständigen Konstruktorterm-Substitutionen.

Satz 4.1.6 *Sei $lit \in Lit_\Sigma$ ein Literal mit $lit = t_1 \doteq t_2$ oder $lit = t_1 \not\doteq t_2$ und sei σ eine vollständige Konstruktorterm-Substitution sowohl für t_1 als auch für t_2. Dann gilt:*

$$(\mathfrak{TransLit}_\Sigma(lit))\sigma = \mathfrak{TransLit}_\Sigma((lit)\sigma)$$

Beweis: O.B.d.A. sei $lit = t_1 \doteq t_2$. Dann gilt:

$$\begin{array}{rcc} (\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2))\sigma & = & \{\ \mathsf{same}(\mathfrak{Rep}(t_1), \mathfrak{Rep}(t_2))\ \}\sigma \\ & & \cup \\ & & (\mathfrak{TransTerm}_\Sigma(t_1))\sigma \\ & & \cup \\ & & (\mathfrak{TransTerm}_\Sigma(t_2))\sigma \end{array}$$

Wegen $(\mathfrak{Rep}(t))\sigma = \mathfrak{Rep}(t\sigma)$ und Satz 4.1.5 gilt dann:

$$\begin{array}{rcc} (\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2))\sigma & = & \{\ \mathsf{same}(\mathfrak{Rep}(t_1\sigma), \mathfrak{Rep}(t_2\sigma))\ \} \\ & & \cup \\ & & \mathfrak{TransTerm}_\Sigma(t_1\sigma) \\ & & \cup \\ & & \mathfrak{TransTerm}_\Sigma(t_2\sigma) \\ & = & \mathfrak{TransLit}_\Sigma(t_1\sigma \doteq t_2\sigma) \\ & = & \mathfrak{TransLit}_\Sigma((t_1 \doteq t_2)\sigma) \end{array}$$

□

Schließlich formulieren wir noch ein paar einfache Sätze über den Zusammenhang zwischen Transformationen und freien Variablen.

Satz 4.1.7 *Für $t \in T_\Sigma$ gilt: $Var(\mathfrak{Rep}(t)) = Var(t)$.*

Beweis: Triv. nach Def. 3.2.9. □

Satz 4.1.8 *Für $t \in T_\Sigma$ gilt: $Var(\mathfrak{TransTerm}_\Sigma(t)) \subseteq Var(t)$.*

Beweis: Durch strukturelle Induktion über t:

Ind.-Anfang:
Ist $t \in V_\Sigma \cup \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$, dann folgt aus der Definition von $\mathfrak{TransTerm}_\Sigma$ (Fall 1 und 2) unmittelbar $Var(\mathfrak{TransTerm}_\Sigma(t)) = \emptyset$.

Ind.-Schritt:
Ist $t = l(t_1, \ldots, t_n)$, $l \in \overline{\mathcal{C}} \cup \overline{\mathcal{F}}$, $\alpha(l) = s_1, \ldots, s_n$. Dann gilt:

$$\begin{aligned} &Var(\mathfrak{TransTerm}_\Sigma(t)) \quad = \\ &Var(\mathfrak{Rep}(t_1)) \cup \ldots \cup Var(\mathfrak{Rep}(t_n)) \cup Var(\mathsf{val}(t)) \\ &\cup Var(\mathfrak{TransTerm}_\Sigma(t_1)) \cup \ldots \cup Var(\mathfrak{TransTerm}_\Sigma(t_n)) \end{aligned}$$

Da a) $Var(\mathfrak{Rep}(t_i)) = Var(t_i)$, b) $Var(\mathsf{val}(t)) = Var(t)$, c) $Var(t_i) \subseteq Var(t)$ und wegen der Ind.-Hyp. $Var(\mathfrak{TransTerm}_\Sigma(t_i)) \subseteq Var(t_i)$ gilt $Var(\mathfrak{TransTerm}_\Sigma(t)) \subseteq Var(t)$. □

Satz 4.1.9 *Für* $lit \in Lit_\Sigma$ *gilt:* $Var(\mathfrak{TransLit}_\Sigma(lit)) = Var(lit)$.

Beweis: O.B.d.A. sei $lit = t_1 \doteq t_2$. Dann gilt:
$Var(\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2))$
$= Var(\mathfrak{Rep}(t_1)) \cup Var(\mathfrak{Rep}(t_2)) \cup Var(\mathfrak{TransTerm}_\Sigma(t_1)) \cup Var(\mathfrak{TransTerm}_\Sigma(t_2))$
$= Var(t_1) \cup Var(t_2) \cup Var(\mathfrak{TransTerm}_\Sigma(t_1)) \cup Var(\mathfrak{TransTerm}_\Sigma(t_2))$
Mit Satz 4.1.8 folgt dann:
$Var(\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2)) = Var(t_1) \cup Var(t_2) = Var(t_1 \doteq t_2)$. □

4.2 Eigenschaften von Meta-Atom-Mengen

Die Prozedur `MG-proc` baut einen MG-Baum auf, d.h. eine Sequenz von MG-Ästen (vergl. Def. 3.4.13). Formal ist ein MG-Ast A nichts anderes als eine Menge von Grund-Meta-Atomen ($A \subseteq MA^0_\Sigma$). Die Bezeichnung 'MG-Ast' verwenden wir aber bevorzugt für solche Mengen von Grund-Meta-Atomen, die tatsächlich von `MG-proc` konstruiert werden. Im Interesse einer klaren Argumentation treten wir hier zunächst einen Schritt zurück und diskutieren in diesem Abschnitt *mögliche* Eigenschaften allgemeiner Meta-Atom-Mengen. Erst im nächsten Abschnitt (4.3) werden wir dann beweisen, daß die von `MG-proc` *tatsächlich konstruierten* (saturierten) MG-Äste genau diese Eigenschaften besitzen, woraus dann die Modell-Korrektheit des Verfahrens folgt.

Eine erste interessante Eigenschaft von Meta-Atom-Mengen ist ihre Rechtseindeutigkeit, oder 'Funktionalität', bzgl. der is und I-Atome.

Definition 4.2.1 (funktionale Mengen von Meta-Atomen)
Eine Menge $A \subseteq MA^0_\Sigma$ von Grund-Meta-Atomen heißt

- **val-funktional**, wenn f.a. *Konstruktorterme* $ct \in CT_\Sigma$ und $ct' \in CT_\Sigma$ gilt: falls $\mathsf{is}(\mathsf{val}(t), ct) \in A$ und $\mathsf{is}(\mathsf{val}(t), ct') \in A$ für ein $t \in T^0_\Sigma$, dann ist $ct = ct'$.
- **I-funktional**, wenn f.a. *Konstruktorterme* $ct \in CT_\Sigma$ und $ct' \in CT_\Sigma$ gilt: falls $\mathrm{I}(f, \langle ct_1, \ldots, ct_n \rangle, ct) \in A$ und $\mathrm{I}(f, \langle ct_1, \ldots, ct_n \rangle, ct') \in A$ für $f \in \overline{\mathcal{F}}$ und $\{ct_1, \ldots, ct_n\} \subseteq CT_\Sigma$ (bzw. $\mathrm{I}(f, \langle\rangle, ct) \in A$ und $\mathrm{I}(f, \langle\rangle, ct') \in A$), dann ist $ct = ct'$.

- **funktional**, wenn A sowohl val-funktional als auch I-funktional ist.

$\star$

Nun beschäftigen wir uns damit, wie man aus einer Menge von Meta-Atomen eine 'entsprechende' Interpretation $\mathcal{I}$ ablesen kann.

Definition 4.2.2 (entsprechende Interpretationen)
Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $A \subseteq MA^0_\Sigma$ eine Menge von Grund-Meta-Atomen. Eine $\mathcal{F}$-Interpretation $\mathcal{I}$ **entspricht** A, wenn folgende Bedingungen erfüllt sind:

1. Ist $\mathrm{I}(f, \langle ct_1, \ldots, ct_n \rangle, ct) \in A$, für $f \in F_s$, $\alpha(f) = s_1 \ldots s_n$ und es gilt $\langle ct_1, \ldots ct_n, ct \rangle \in CT_{s_1} \times \ldots \times CT_{s_n} \times CT_s$, dann ist $\mathcal{I}(f)(ct_1, \ldots ct_n) = ct$.
2. Ist $\mathrm{I}(a, \langle\rangle, ct) \in A$, für $a \in F_s$, $\alpha(f) = \lambda$ und es gilt $ct \in CT_s$, dann ist $\mathcal{I}(a)() = ct$.

$\star$

Die Interpretation $\mathcal{I}$ wird aus A ausgelesen, soweit dies möglich ist. Zwei Dinge sind hier zu beachten:

1. $\mathcal{I}$ richtet sich nur nach solchen I-Atomen, welche im letzten Argument bzw. im Tupel Konstruktorterme enthalten. In dem in Kapitel 3 beschriebenen Verfahren werden aber auch solche I-Atome wie z.B. $\mathrm{I}(f, \langle 0 \rangle, \mathsf{val}(f(0)))$ gebildet. Hieraus ist jedoch noch nicht ablesbar, zu welchem Konstruktorterm f an der Stelle 0 interpretiert wird. Erst, wenn z.B. $\mathsf{is}(\mathsf{val}(f(0)), 1)$ abgeleitet ist und durch Ersetzungsregeln $\mathrm{I}(f, \langle 0 \rangle, 1)$ entsteht, ist ablesbar, welchen Wert $\mathcal{I}(f)(0)$ hat (nämlich 1).
2. Nach dieser Definition kann einer Meta-Atom-Menge A nur dann eine Interpretation *entsprechen*, wenn A I-funktional ist, da $\mathcal{I}$ sonst ambivalent definiert ist. Wenn A aber I-funktional ist, dann gibt es *immer* eine entsprechende Interpretation $\mathcal{I}$. Dies halten wir im folgenden Satz fest.

Satz 4.2.3 *Sei $\Sigma = (S, \mathcal{C}, \mathcal{F}, \alpha)$ eine ADT-Signatur und $A \subseteq MA^0_\Sigma$. Dann gilt: Wenn A I-funktional ist, dann ex. eine A entsprechende $\mathcal{F}$-Interpretation $\mathcal{I}$.*

Beweis: Wir konstruieren eine A entsprechende $\mathcal{F}$-Interpretation $\mathcal{I}$.

- $\mathcal{I}(f)(ct_1, \ldots ct_n)$ wird auf ct gesetzt, falls $\mathrm{I}(f, \langle ct_1, \ldots, ct_n \rangle, ct) \in A$. Hierbei ist ct wohldefiniert aufgrund der I-Funktionalität von A.

- $\mathcal{I}(f)(ct_1, \dots ct_n)$ wird auf ein beliebiges $ct \in CT_\Sigma$ gestzt, falls für kein $ct' \in CT_\Sigma$ gilt, daß $\mathrm{I}(f, \langle ct_1, \dots, ct_n \rangle, ct') \in A$.

□

I.a. können zu einer I-funktionalen Menge A unendlich viele entsprechende Interpretationen existieren.

Nachdem nun der Zusammenhang hergestellt ist zwischen I-Atomen und der Interpretation $\mathcal{I}$, setzen wir nun is-Atome der Form $\mathsf{is}(\mathsf{val}(t), ct)$ und die Auswertefunktion $val_\mathcal{I}$ miteinander in Beziehung. Dieses Unterfangen gestaltet sich leider etwas komplizierter als das letzte. In der Weise, in der auf der Seite der Semantik $val_\mathcal{I}$ und $\mathcal{I}$ miteinander verwoben sind, müssen auch die is- und die I-Atome zueinander 'passen'. Diese Eigenschaft gilt es jetzt zu formalisieren.

Die erste Definition in diesem Zusammenhang stellt sich zunächst recht 'dumm', und liest aus jeder beliebigen Menge von Meta-Atomen heraus, wie (gewisse) Terme 'evaluiert'[1] werden. Dabei ist anfangs weder die Eindeutigkeit garantiert noch die Korrespondenz zu irgendeiner semantischen Interpretation $\mathcal{I}$.

Definition 4.2.4 (Term-Evaluation in Meta-Atom-Mengen)
Sei $A \in MA^0_\Sigma$, $t \in T^0_\Sigma$ und $ct \in CT_\Sigma$.
t **evaluiert in** A **zu** ct genau dann, wenn gilt:

1. falls $t \notin CT_\Sigma$, dann ist $\mathsf{is}(\mathsf{val}(t), ct) \in A$.
2. falls $t \in CT_\Sigma$, dann gilt $ct = t$.

⋆

Intuitiv bedeutet dies, daß wir (1.) die Auswertung des Termes t aus einem passenden is-Atom in A ablesen, wobei wir uns (2.) für jeden Konstruktorterm ct das Atom $\mathsf{is}(\mathsf{val}(ct), ct)$ zu A 'hinzudenken', auch wenn es eigentlich nicht in A enthalten ist.

Damit die Evaluation von Termen durch Meta-Atom-Mengen mit der semantischen Auswertung $val_\mathcal{I}$ von Termen korrespondiert, fehlen uns noch drei Eigenschaften:

1. I.a. evaluiert nicht jede Meta-Atom-Menge jeden Term. (Z.B. ist die Evaluation in der leeren Meta-Atom-Menge nur für Konstruktorterme definiert.) Wir müssen uns also die Möglichkeit verschaffen, von einer Meta-Atom-Menge verlangen zu können, daß sie bestimmte Terme inklusive all ihrer Unterterme (verg. Def. 4.2.5) evaluiert.

[1] Wir erlauben uns hier eine natürlichsprachliche Überladung des Begriffs 'evaluieren'.

2. Die obige Definition garantiert noch nicht die Eindeutigkeit der Evaluation in A. Diese Eindeutigkeit wird jetzt als Anforderung an Meta-Atom-Mengen formuliert.

3. Die Evaluation von Termen durch Mengen von Meta-Atomen korrespondiert nur dann mit dem semantischen $val_{\mathcal{I}}$, wenn sie zu einer Interpretation paßt, d.h. wenn überhaupt eine Interpretation *existiert*, zu der sie paßt. Um dies zu garantieren, repräsentieren wir neben der Evaluation auch die *Interpretation* von Funktionen als Meta-Atome und verlangen (weiter unten), daß diese Repräsentation *funktional* ist.

Definition 4.2.5 (Unterterm-Menge)
$UT \subseteq T_\Sigma$ ist eine **Unterterm-Menge**, falls $unterTerme(t) \subseteq UT$ für alle $t \in UT$.
⋆

Eine Unterterm-Menge ist also gegen Unterterme abgeschlossen. Insbesondere ist $unterTerme(t)$ bzw. $unterTerme(lit)$ immer eine Unterterm-Menge, wie man leicht zeigen kann. Nun können wir einige schon erwähnte Eigenschaften von Meta-Atom-Mengen einfordern. Wir beziehen uns dabei auf Unterterm-Mengen UT. (Später werden dies die Mengen aller Unterterme der Axiome sein.) Wir sagen, daß eine Meta-Atom-Menge A dann *'adäquat'* ist für UT, wenn jeder Term $t \in UT$ in A *evaluiert* wird und wenn außerdem zu der in A repräsentierten *Evaluation* von t auch eine passende *Interpretation* (der entsprechenden Funktion auf den entsprechenden Argumenten) in A repräsentiert ist.

Definition 4.2.6 (adäquate Meta-Atom-Mengen)
Sei $UT \subseteq T^0_\Sigma$ eine Unterterm-Menge. Eine Menge $A \subseteq MA^0_\Sigma$ von Meta-Atomen ist **adäquat für** UT, wenn f.a. $t \in UT$ gilt:

1. falls $t = a$, mit $a \in F_s$ und $\alpha(a) = \lambda$, dann gibt es ein $ct \in CT_\Sigma$, so daß $\mathsf{is}(\mathsf{val}(a), ct) \in A$ und $\mathrm{I}(a, \langle\rangle, ct) \in A$.

2. falls $t = f(t_1, \ldots, t_n)$, mit $f \in F_s$ und $\alpha(f) = s_1 \ldots s_n$, und falls f.a. $t_i \in \{t_1, \ldots, t_n\}$ der Term t_i in A zu ct_i evaluiert[2], dann gibt es ein $ct \in CT_\Sigma$, so daß $\mathsf{is}(\mathsf{val}(f(t_1, \ldots, t_n)), ct) \in A$ und $\mathrm{I}(f, \langle ct_1, \ldots, ct_n\rangle, ct) \in A$.

3. falls $t = c(t_1, \ldots, t_n)$, mit $c \in C_s$, $\alpha(c) = s_1 \ldots s_n$ und $t \notin CT_\Sigma$ und falls f.a. $t_i \in \{t_1, \ldots, t_n\}$ der Term t_i in A zu ct_i evaluiert, dann ist $\mathsf{is}(\mathsf{val}(c(t_1, \ldots, t_n)), c(ct_1, \ldots, ct_n)) \in A$.

⋆

[2] Vgl. Def. 4.2.4.

Man beachte, daß wir nur für $t \notin CT_\Sigma$ irgendwelche Forderungen aufstellen. Die Adäquatheit einer Meta-Term-Menge A für eine Unterterm-Menge UT garantiert u.a., daß jeder Term $t \in UT$ in A zu einem Konstruktorterm (passender Sorte) evaluiert:

Satz 4.2.7 *Sei $UT \subseteq T^0_\Sigma$ eine Unterterm-Menge und sei $A \subseteq MA^0_\Sigma$ adäquat für UT. Dann gilt für alle $t \in UT$:*
falls $sort(t) = s$, dann gilt für ein $ct \in CT_s$, daß t in A zu ct evaluiert.

Beweis: Strukturelle Induktion über t:

1. Ist $t = c$, $c \in C_s$ und $\alpha(c) = \lambda$, dann ist $t \in CT_\Sigma$ und die Beh. gilt trivialerweise. (vgl. Def. 4.2.4).
2. Ist $t = a$, mit $a \in F_s$ und $\alpha(a) = \lambda$, dann folgt aus der Adäquatheit (1.) die Beh.
3. Sei $t = c(t_1, \ldots, t_n)$, mit $c \in C_s$ und $\alpha(c) = s_1 \ldots s_n$. Laut Ind.-Hyp. existiert für jedes $t_i \in \{t_1, \ldots, t_n\}$ ein ct_i, so daß t_i in A zu ct_i evaluiert.
   a) falls $c(t_1, \ldots, t_n) \notin CT_\Sigma$, dann folgt mit der Adäquatheit (3.) bei Wahl von $ct = c(ct_1, \ldots, ct_n)$ die Beh.
   b) falls $c(t_1, \ldots, t_n) \in CT_\Sigma$, dann gilt die Beh. triv. (s. (1.)).
4. Sei $t = f(t_1, \ldots, t_n)$, mit $f \in F_s$ und $\alpha(f) = s_1 \ldots s_n$. Dann gilt die Ind.-Hyp. für jedes $t_i \in \{t_1, \ldots, t_n\}$. Mit der Adäquatheit (2.) folgt die Beh. für t.

□

Man beachte, daß hier die strukturelle Induktion nur deshalb funktioniert, weil UT eine Unterterm-Menge, also gegenüber Untertermen abgeschlossen ist. Wenn UT außerdem val-funktional ist, dann kann man obigen Satz verschärfen zu „... genau ein ct ...“:

Satz 4.2.8 *Sei $UT \subseteq T^0_\Sigma$ eine Unterterm-Menge und sei $A \subseteq MA^0_\Sigma$ adäquat für UT. Falls UT val-funktional ist, dann gibt es für jeden Term $t \in UT$ genau einen Konstruktorterm $ct \in CT_\Sigma$, so daß t in A zu ct evaluiert.*

Beweis: Satz 4.2.7 garantiert die Existenz eines solchen ct. Es bleibt noch dessen Eindeutigkeit zu zeigen. Nehmen wir also an:
t evaluiert in A zu ct und t evaluiert in A zu ct'.
Falls nun $t \in CT_\Sigma$, dann folgt aus Def. 4.2.4, daß $ct = t$ und $ct' = t$, also $ct = ct'$.

Falls andererseits $t \notin CT_\Sigma$, dann folgt aus Def. 4.2.4, daß $\mathsf{is}(\mathsf{val}(t), ct) \in M$ und $\mathsf{is}(\mathsf{val}(t), ct') \in M$. Aus der val-Funktionalität folgt $ct = ct'$. □

Nun sind wir in der Lage, zu zeigen, daß für *adäquate, funktionale* Meta-Atom-Mengen A gilt: Wenn wir aus A eine (A 'entsprechende') Interpretation $\mathcal{I}$ ablesen, dann ist $\mathsf{is}(\mathsf{val}(t), ct) \in A$ gleichbedeutend mit $val_\mathcal{I}(t) = ct$, für genau dieses $\mathcal{I}$!

Satz 4.2.9 *Sei $UT \subseteq T^0_\Sigma$ eine Unterterm-Menge, sei $A \subseteq MA^0_\Sigma$ eine funktionale, für UT adäquate Menge von Meta-Atomen, und sei $\mathcal{I}$ eine A entsprechende $\mathcal{F}$-Interpretation. Dann gilt für jeden Term $t \in UT$:*
falls t in A zu ct evaluiert, dann ist $val_\mathcal{I}(t) = ct$.

Beweis: Zunächsteinmal ist zu beachten, daß der Satz nur wegen der I-Funktionalität von A wohlgeformt ist, denn nur dann existiert überhaupt eine A entsprechende Interpretation $\mathcal{I}$. Wir beweisen die Beh. durch strukturelle Induktion über t, was möglich ist, da UT mit t auch die Unterterme von t enthält, und somit die Adäquatheit der Menge A auch für die Argumente des Termes t gilt.

1. Ist $t = c$, $c \in C_s$ und $\alpha(c) = \lambda$, dann ist $t \in CT_\Sigma$, weswegen $val_\mathcal{I}(t) = t$ ist und t in A zu T evaluiert (lt. Def. 4.2.4).

2. Sei $t = a$, mit $a \in F_s$ und $\alpha(a) = \lambda$. Wir nehmen an, daß a in A zu ct evaluiert. Da $a \notin CT_\Sigma$, heißt dies, daß $\mathsf{is}(\mathsf{val}(a), ct) \in A$. (Daraus wollen wir $val_\mathcal{I}(a) = ct$ folgern.) Aus der Adäquatheit (1.) folgt, daß für ein $ct_0 \in CT_\Sigma$ gilt, daß $\mathsf{is}(\mathsf{val}(a), ct_0) \in A$ und $\mathrm{I}(a, \langle\rangle, ct_0) \in A$. Aus der val-Funktionalität folgt, daß $ct_0 = ct$, also $\mathrm{I}(a, \langle\rangle, ct) \in A$. Da $\mathcal{I}$ A entspricht, gilt $\mathcal{I}(a)() = ct$. Dann ist $val_\mathcal{I}(a) = \mathcal{I}(a)() = ct$.

3. Sei $t = c(t_1, \ldots, t_n)$, mit $c \in C_s$ und $\alpha(c) = s_1 \ldots s_n$. Falls nun $t \in CT_\Sigma$, dann folgt die Beh. wie in Fall 1. Sei also $c(t_1, \ldots, t_n) \notin CT_\Sigma$. Wir nehmen an, daß $c(t_1, \ldots, t_n)$ in A zu ct evaluiert, d.h. (wg. $c(t_1, \ldots, t_n) \notin CT_\Sigma$) daß $\mathsf{is}(\mathsf{val}(c(t_1, \ldots, t_n)), ct) \in A$. Da UT eine Unterterm-Menge ist und wegen der val-Funktionalität von A ist Satz 4.2.8 auf die Argumente von t anwendbar. Demnach ex. für jeden Term $t_i \in \{t_1, \ldots, t_n\}$ genau ein ct_i, so daß t_i in A zu ct_i evaluiert (*). Aus der Adäquatheit (3.) folgt, daß $\mathsf{is}(\mathsf{val}(c(t_1, \ldots, t_n)), c(ct_1, \ldots, ct_n)) \in A$. Aus der val-Funktionalität folgt, daß $c(ct_1, \ldots, ct_n) = ct$ (**). Laut Ind.-Hyp. gilt mit (*), daß $val_\mathcal{I}(t_i) = ct_i$, f.a. $t_i \in \{t_1, \ldots, t_n\}$. Dann haben wir:
$val_\mathcal{I}(c(t_1, \ldots, t_n)) = c(val_\mathcal{I}(t_1), \ldots, val_\mathcal{I}(t_n)) = c(ct_1, \ldots, ct_n) \stackrel{**}{=} ct.$

4. Sei $t = f(t_1, \ldots, t_n)$, mit $f \in F_s$ und $\alpha(f) = s_1 \ldots s_n$. Wir nehmen an, daß $f(t_1, \ldots, t_n)$ in A zu ct evaluiert, d.h. (wg. $f(t_1, \ldots, t_n) \notin CT_\Sigma$), daß $\mathsf{is}(\mathsf{val}(f(t_1, \ldots, t_n)), ct) \in A$. Da UT eine Unterterm-Menge ist und wegen

der val-Funktionlität von A ist Satz 4.2.8 auf die Argumente von t anwendbar. Demnach ex. für jeden Term $t_i \in \{t_1, \ldots, t_n\}$ genau ein ct_i, so daß t_i in A zu ct_i evaluiert (*). Aus der Adäquatheit (2.) folgt, daß für ein ct_0 gilt: $\mathsf{is}(\mathsf{val}(f(t_1, \ldots, t_n)), ct_0) \in A$ und $\mathrm{I}(f, \langle ct_1, \ldots, ct_n\rangle, ct_0) \in A$. Aus der val-Funktionalität folgt, daß $ct_0 = ct$, also $\mathrm{I}(f, \langle ct_1, \ldots, ct_n\rangle, ct) \in A$. Da $\mathcal{I}$ A entspricht, gilt $\mathcal{I}(f)(ct_1, \ldots, ct_n) = ct$. (**) Laut Ind.-Hyp. gilt mit (*), daß $val_{\mathcal{I}}(t_i) = ct_i$, f.a. $t_i \in \{t_1, \ldots, t_n\}$. Dann haben wir:
$val_{\mathcal{I}}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(val_{\mathcal{I}}(t_1), \ldots, val_{\mathcal{I}}(t_n)) = \mathcal{I}(f)(ct_1, \ldots, ct_n) \overset{**}{=} ct.$

□

Bis hierher können wir aus adäquaten Mengen Interpretationen und Evaluationen (Auswertungen) ablesen, und wir haben bewiesen, daß unter Annahme der Funktionalität beides zusammenpaßt. Nun geht es darum, wann eine abgelesene Interpretation/Evaluation bestimmte Formeln erfüllt, so daß die der Interpretation entsprechende Algebra ein *Modell* der Formeln ist. Zunächst betrachten wir hierbei nur Grundliterale.

Wir überladen jetzt den Begriff 'erfüllt' und verwenden ihn für eine mögliche Beziehung zwischen Meta-Atom-Mengen und Grundliteralen. Anschließend zeigen wir, daß dieses 'erfüllt' mit dem '$\models$' zwischen (echten) Interpretationen und Literalen korrespondiert.

Definition 4.2.10 (Erfüllung von Litaralen durch M.-A.-Mengen)
Sei $lit \in Lit^0_\Sigma$ und $A \subseteq MA^0_\Sigma$.
A **erfüllt** lit, wenn gilt:

1. A ist adäquat für $unterTerme(lit)$,
2. A ist funktional,
3. falls $lit = t \doteq t'$, t in A zu ct evaluiert und t' in A zu ct' evaluiert, dann gilt $ct = ct'$,
4. falls $lit = t \not\doteq t'$, t in A zu ct evaluiert und t' in A zu ct' evaluiert, dann gilt $ct \neq ct'$.

⋆

Da die Funktionalität sowie der recht gehaltvolle Adäquatheitsbegriff vorsorglich in diese Definition mit aufgenommen wurden, können wir den folgenden Satz sehr prägnant formulieren.

Satz 4.2.11 *Sei $lit \in Lit^0_\Sigma$ und $A \subseteq MA^0_\Sigma$. Dann gilt:*
Falls A lit erfüllt, dann gilt $\mathcal{I} \models lit$ für jede A entsprechende $\mathcal{F}$-Interpretation $\mathcal{I}$.

Beweis: Wir müssen zeigen, daß $val_{\mathcal{I}}(lit) = W$.
Sei $lit = t \doteq t'$ oder $lit = t \not\doteq t'$. Da $\{t, t'\} \subseteq unterTerme(lit)$ und A adäquat für $unterTerme(lit)$ sowie val-funktional ist, gilt nach Satz 4.2.8 für genau ein ct und genau ein ct', daß t in A zu ct evaluiert und t' in A zu ct' evaluiert. Nach Satz 4.2.9 gilt dann $val_{\mathcal{I}}(t) = ct$ und $val_{\mathcal{I}}(t') = ct'$.

1. Falls nun $lit = t \doteq t'$, dann folgt aus obiger Def. (3.) $ct = ct'$. Dann gilt $val_{\mathcal{I}}(t) = val_{\mathcal{I}}(t')$ und somit $val_{\mathcal{I}}(t \doteq t') = W$.

2. Falls nun $lit = t \not\doteq t'$, dann folgt aus obiger Def. (4.) $ct \neq ct'$. Dann gilt $val_{\mathcal{I}}(t) \neq val_{\mathcal{I}}(t')$ und somit $val_{\mathcal{I}}(t \not\doteq t') = W$.

□

4.3 Beschränkte Modell-Korrektheit

In diesem Abschnitt wollen wir nachweisen, daß aus

$$R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_{\Sigma}(Inst)$$

und der Terminierung von *MG-proc*(R) mit Ausgabe „A ist R-saturiert“ die Erfüllbarkeit von $\langle \Sigma, [AX/Inst] \rangle$ folgt, und daß man die erfüllende Interpretation $\mathcal{I}$ aus A ablesen kann. Dabei können wir von den Definitionen und Sätzen des letzten Abschnitts profitieren, wenn wir hier zeigen, daß R-saturierte MG-Äste *funktional* sind und darüberhinaus für die Unterterme von $[AX/Inst]$ *adäquat*.

Der Nachweis der Adäquatheit ist allerdings nicht trivial. Betrachten wir kurz eine der Bedingungen aus der Definition 4.2.6 der Adäquatheit. „1. falls $t = a$, mit $a \in F_s$ und $\alpha(a) = \lambda$, dann gibt es ein $ct \in CT_{\Sigma}$, so daß $\mathsf{is}(\mathsf{val}(a), ct) \in A$ und $\mathrm{I}(a, \langle\rangle, ct) \in A$.“ Dem gegenüber stehen die Regeln, die aus der Transformation von Termen (Def. 3.2.11) entstehen: „2. Falls $t = a$ mit $a \in F_s$, $\alpha(a) = \lambda$, dann: $\mathfrak{TransTerm}_{\Sigma}(t) = \{\mathrm{I}(a, \langle\rangle, \mathsf{val}(a)),\ \mathsf{search_}s(\mathsf{val}(a))\}$.“ D.h.: Was wir haben, ist $\mathsf{search_}s(\mathsf{val}(a))$, was wir aber brauchen, ist $\mathsf{is}(\mathsf{val}(a), ct)$, für ein ct. Tatsächlich enthält jeder saturierte MG-Ast mit $\mathsf{search_}s(t)$ auch $\mathsf{is}(t, ct)$, für ein ct. Dies gilt es nun zu zeigen. Der Beweis hierfür benutzt das Prinzip der noetherschen Induktion, unter Verwendung einer noetherschen Ordnung $\sqsubseteq$ auf Meta-Termen $\in MT_{\Sigma}$. Dazu definieren wir zunächst eine Relation $\prec$ auf MT_{Σ}, deren reflexive, transitive Hülle $\sqsubseteq$ ist.

Definition 4.3.1 (Relation $\prec$)
Seien $\{t_1, t_2\} \subseteq MT_{\Sigma}$ Meta-Terme.

$$t_1 \prec t_2 \quad :\Longleftrightarrow \quad t_1 = \mathsf{arg}n(t_2), \text{ für ein } n \in \mathbb{N}$$

⋆

Es mag zunächst überraschen, daß der Meta-Term $\mathsf{arg}n(t)$ im Sinne von $\prec$ kleiner ist als der Meta-Term t, obwohl er syntaktisch größer ist. Diese Ordnung macht aber Sinn, wenn man bedenkt, daß der Meta-Term $\mathsf{arg}n(t)$ in einem gewissen Sinne für ein *Argument* von t steht. Die Eigenschaft der folgenden Ordnung $\sqsubseteq$, noethersch zu sein, reflektiert somit die Tatsache, daß ein Term nicht unendlich viele Unterterme haben kann.

Definition 4.3.2 (Ordnungen $\sqsubseteq$, $\sqsubset$)
$\sqsubseteq$ ist die reflexive, transitive Hülle[3] von $\prec$.
$\sqsubset$ ist definiert durch: $\quad t_1 \sqsubset t_2 \quad :\Longleftrightarrow \quad t_1 \sqsubseteq t_2$ und $t_1 \neq t_2$ $\star$

Satz 4.3.3 *Sei A eine endliche Teilmenge von MT_Σ. Dann ist $\sqsubseteq$ auf A eine noethersche Ordnung*[4].

Beweis: Zunächst muß man zeigen, daß $\sqsubseteq$ auf A eine partielle Ordnung ist. $\sqsubseteq$ ist per Def. reflexiv und transitiv, bleibt nur die Antisymmetrie zu zeigen. Diese ist leicht nachweisbar per Induktion über Ketten der Form $t_1 \prec \ldots \prec t_n$. (Dieses Argument funktioniert, weil wegen der Endlichkeit von A auch solche Ketten endlich sind.) Somit ist $\sqsubseteq$ auf A eine partielle Ordnung. Zur 'noethersch' fehlt dann nur noch die Wohlfundiertheit von $\sqsubseteq$, d.h. die Eigenschaft, daß es keine unendlich absteigende $\sqsubset$-Kette gibt (vgl. [LS84, Theorem 1.11]). Dies folgt aber trivial aus der Endlichkeit von A. $\square$

Aus diesem Satz folgt, daß man auf endliche Meta-Atom-Mengen $A \subseteq MT_\Sigma$ das Beweisprinzip der *noetherschen Induktion*, (mit $\sqsubseteq$ als Ordnung) anwenden kann. Dieses Beweisprinzip geben wir in dem folgenden Satz wieder (entnommen aus [LS84, Theorem 1.13], Beweis siehe dort).

Satz 4.3.4 (noethersche Induktion)
Sei $\sqsubseteq$ eine noethersche Ordnung auf einer Menge M. Um für alle Elemente von M zu zeigen, daß sie eine Eigenschaft P besitzen, genügt es zu zeigen:

1. *P gilt für alle minimalen Elemente von M.*

2. *Für jedes Element $e \in M$ gilt: Wenn e nicht minimal ist und für alle $e' \in M$ mit $e' \sqsubset e$ gilt P, dann gilt P auch für e.*

Wir werden die noethersche Induktion dazu verwenden, den folgenden Satz zu beweisen. (Es sei daran erinnert, daß 'MG-Äste' nichts anderes als Mengen von Grund-Meta-Atomen sind.)

[3]vgl. [LS84], Abschn. 1.1.1, 'closure'
[4]ebd., Abschn. 1.2.2., 'well-founded set'

Satz 4.3.5 *Sei* $\langle \Sigma, AX \rangle$ *eine normalisierte ADT-Spezifikation, sei* $\mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \subseteq R$ *und sei* $A \subseteq MA^0_\Sigma$ *ein endlicher, R-saturierter MG-Ast. Dann gilt:*
Wenn $\mathsf{search_}s(t) \in A$, *dann ist auch* $\mathsf{is}(t, ct) \in A$, *für ein* $ct \in CT_\Sigma$.

Beweis: Hilfsdefinition: *searched*(A) bezeichnet die Menge der in A 'gesuchten' Meta-Terme:

$$t \in searched(A) \quad :\Longleftrightarrow \quad \mathsf{search_}s(t) \in A \text{ für ein } s \in S$$

Da A endlich ist, ist auch *searched*(A) endlich. Dann ist $\sqsubseteq$ eine noethersche Ordnung über *searched*(A). Wir beweisen den Satz mit noetherscher Induktion über *searched*(A):
Sei $t \in searched(A)$, dann gilt für ein $s \in S$: $\mathsf{search_}s(t) \in A$. Sei $C_s = \{c_1, \ldots, c_n\}$ die Menge der Konstruktoren der Sorte s. Dann ist:

$$\mathsf{search_}s(x) \ \rightarrow \ \mathfrak{TransKonstr}_\Sigma(x, c_1); \ldots ; \mathfrak{TransKonstr}_\Sigma(x, c_n) \ . \ \in R$$

Wegen der R-Saturiertheit und $\mathsf{search_}s(t) \in A$ gilt für ein $c_i \in \{c_1, \ldots, c_n\}$, daß $\mathfrak{TransKonstr}_\Sigma(t, c_i) \subseteq A$. Noethersche Induktion:

1. t sei ein $\sqsubseteq$-minimales Element in *searched*(A). Dann muß $\alpha(c_i) = \lambda$ sein. (Sonst wäre laut Def. 3.2.2 $\mathsf{search_}s_1(\mathsf{arg1}(t)) \in A$, und da $\mathsf{arg1}(t) \prec t$, wäre t nicht minimal in *searched*(A).) Folglich ist $\mathfrak{TransKonstr}_\Sigma(t, c_i) = \{\mathsf{is}(t, c_i)\}$. Mit $ct = c_i$ folgt die Beh.

2. t sei nicht $\sqsubseteq$-minimal in *searched*(A). Dann kann $\alpha(c_i) = \lambda$ eigentlich nicht gelten. Statt dies zu zeigen, weist man genau wie in Fall (1.) nach, daß aus $\alpha(c_i) = \lambda$ die Beh. folgen würde. Sei also $\alpha(c_i) = s_1 \ldots s_m$. Dann ist

$$\begin{array}{rcl} \mathfrak{TransKonstr}_\Sigma(t, c_i) & = & \{ \ \mathsf{is}(t, c_i(\mathsf{arg1}(t), \ldots, \mathsf{arg}n(t))) \ \} \\ & & \cup \\ & & \{ \ \mathsf{search_}s_1(\mathsf{arg1}(t)) \ \} \\ & & \cup \\ & & \vdots \\ & & \cup \\ & & \{ \ \mathsf{search_}s_m(\mathsf{arg}m(t)) \ \} \end{array}$$

   Da $\mathsf{arg1}(t) \sqsubseteq t$ und $\ldots$ und $\mathsf{arg}m(t) \sqsubseteq t$, folgt aus der Ind.-Hyp., daß für gewisse Konstruktorterme $\langle ct_1, \ldots, ct_m \rangle \in CT_{s_1} \times \ldots \times CT_{s_m}$ gilt, daß $\{\mathsf{is}(\mathsf{arg1}(t), ct_1), \ldots, \mathsf{is}(\mathsf{arg}m(t), ct_m)\} \subseteq A$. Aus

$$\begin{array}{l} \{ \ \mathsf{is}(x, c_i(y_1, \ldots, y_m)), \ \mathsf{is}(y_1, z) \ \rightarrow \ \mathsf{is}(x, c_i(z, \ldots, y_m)) \ . \ , \\ \qquad \vdots \\ \quad \mathsf{is}(x, c_i(y_1, \ldots, y_m)), \ \mathsf{is}(y_n, z) \ \rightarrow \ \mathsf{is}(x, c_i(y_1, \ldots, z)) \ . \ \} \quad \subseteq R \end{array}$$

   und der R-Saturiertheit von A folgt, daß auch

$$\begin{array}{l} \{ \ \mathsf{is}(t, c_i(ct_1, \mathsf{arg2}(t), \ldots, \mathsf{arg}m(t))) \ . \ , \\ \quad \mathsf{is}(t, c_i(ct_1, ct_2, \ldots, \mathsf{arg}m(t))) \ . \ , \\ \qquad \vdots \\ \qquad \mathsf{is}(t, c_i(ct_1, \ldots, ct_m)) \ . \ \} \qquad \subseteq A \end{array}$$

Mit $ct = c_i(ct_1, \ldots, ct_m)$ folgt die Beh.

□

Nun zeigen wir, daß ein R-saturierter Ast, welcher für ein Grund-Literal *lit* die Meta-Atome aus $\mathfrak{TransLit}_\Sigma(lit)$ enthält (dieser Zustand wird durch Anwendung einer Axiomen-Regel erreicht), adäquat ist für die Unterterme des Literals.

Satz 4.3.6 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, sei $\mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \subseteq R$ und sei $A \subseteq MA^0_\Sigma$ ein endlicher, R-saturierter MG-Ast. Sei außerdem $lit \in Lit^0_\Sigma$ ein Grundliteral und sei $\mathfrak{TransLit}_\Sigma(lit) \subseteq A$. Dann gilt:*
A ist adäquat für $unterTerme(lit)$.

Beweis: Wir müssen zeigen, daß f.a. Terme $t \in unterTerme(lit)$ die Bedingungen (1.) – (3.) aus Def. 4.2.6 gelten. Fallunterscheidung über $t \in unterTerme(lit)$:

1. Sei $t \in CT_\Sigma$. Dann gibt es nach Def. 4.2.6 nichts zu zeigen.

2. Sei $t = a$, mit $a \in F_s$ und $\alpha(a) = \lambda$. Laut Def. 4.2.6 müssen wir zeigen, daß es ein $ct \in CT_\Sigma$ gibt, so daß $\mathsf{is}(\mathsf{val}(a), ct) \in A$ und $\mathrm{I}(a, \langle\rangle, ct) \in A$. Wegen $\mathfrak{TransLit}_\Sigma(lit) \subseteq A$ und Satz 4.1.3 ist $\mathfrak{TransTerm}_\Sigma(t) \subseteq A$, mit $\mathfrak{TransTerm}_\Sigma(t) = \{\mathrm{I}(a, \langle\rangle, \mathsf{val}(a)),\ \mathsf{search_}s(\mathsf{val}(a))\}$. Weil A R-saturiert und endlich ist, folgt laut Satz 4.3.5 aus $\mathsf{search_}s(\mathsf{val}(a)) \in A$, daß $\mathsf{is}(\mathsf{val}(a), ct_0) \in A$, für ein $ct_0 \in CT_s$. Da $\mathrm{I}(fv, tv, x),\ \mathsf{is}(x, z)\ \rightarrow\ \mathrm{I}(fv, tv, z)\ .\ \in R$, folgt aus der R-Saturiertheit, daß $\mathrm{I}(a, \langle\rangle, ct_0) \in A$. Also ist $\mathsf{is}(\mathsf{val}(a), ct_0) \in A$ und $\mathrm{I}(a, \langle\rangle, ct_0) \in A$, für ein ct_0.

3. Sei $t = c(t_1, \ldots, t_n)$, mit $c \in C_s$ und $\alpha(c) = s_1 \ldots s_n$, wobei $t \notin CT_\Sigma$. Nach Def. 4.2.6 (3.) müssen wir zeigen, daß Folgendes gilt:
   Falls f.a. $t_i \in \{t_1, \ldots, t_n\}$ der Term t_i in A zu ct_i evaluiert, dann ist $\mathsf{is}(\mathsf{val}(c(t_1, \ldots, t_n)), c(ct_1, \ldots, ct_n)) \in A$ (*). Nehmen wir also an, daß t_i in A zu ct_i evaluiert. Wegen $\mathfrak{TransLit}_\Sigma(lit) \subseteq A$ und aufgrund von Satz 4.1.3 ist $\mathfrak{TransTerm}_\Sigma(t) \subseteq A$. Laut Def. von $\mathfrak{TransTerm}_\Sigma$ ist
   $\mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n))) \in \mathfrak{TransTerm}_\Sigma(t)$, somit gilt auch
   $\mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n))) \in A$ (**).
   Betrachten wir die folgenden Meta-Atome:
   $$\begin{aligned} at_0 &= \mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n))) \\ &\vdots \\ at_i &= \mathsf{is}(\mathsf{val}(t), c(ct_1, \ldots, ct_i, \mathfrak{Rep}(t_{i+1}), \ldots, \mathfrak{Rep}(t_n))) \\ &\vdots \\ at_n &= \mathsf{is}(\mathsf{val}(t), c(ct_1, \ldots, ct_n)) \end{aligned}$$
   Wir müssen zeigen, daß $at_n \in A$. Wir zeigen allgemeiner, daß

$\{at_0, at_1, \ldots, at_n\} \subseteq A$, und zwar induktiv.
Der *Ind.-Anf.*, $at_0 \in A$ ergibt sich aus (**).
Im *Ind.-Schritt* nehmen wir an, es gilt:
$at_{i-1} = \mathsf{is}(\mathsf{val}(t), c(ct_1, \ldots, ct_{i-1}, \mathfrak{Rep}(t_i), \ldots, \mathfrak{Rep}(t_n))) \in A$.
Wir unterscheiden zwei Fälle:

a) $t_i \in CT_\Sigma$. Dann ist laut Def. 4.2.4 $t_i = ct_i$, und laut der Def. von $\mathfrak{Rep}$ gilt $\mathfrak{Rep}(t_i) = \mathfrak{Rep}(ct_i) = ct_i$. Dann ist $at_i = at_{i-1}$, also $at_i \in A$.

b) $t_i \notin CT_\Sigma$. Dann ist $\mathfrak{Rep}(t_i) = \mathsf{val}(t_i)$ und
$at_{i-1} = \mathsf{is}(\mathsf{val}(t), c(ct_1, \ldots, ct_{i-1}, \mathsf{val}(t_i), \ldots, \mathfrak{Rep}(t_n)))$. Weil $t_i \notin CT_\Sigma$, folgt aus 't_i evaluiert in A zu ct_i', daß $\mathsf{is}(\mathsf{val}(t_i), ct_i) \in A$. Nun ist
$\mathsf{is}(x, c(y_1, \ldots, y_i, \ldots, y_n)),\ \mathsf{is}(y_i, z) \ \rightarrow\ \mathsf{is}(x, c(y_1, \ldots, z, \ldots, y_n))\ . \in R$.
Aus der R-Saturiertheit folgt dann mit $at_{i-1} \in A$ auch $at_i \in A$.

4. Sei $t = f(t_1, \ldots, t_n)$, mit $f \in F_s$ und $\alpha(f) = s_1 \ldots s_n$. Nach Def. 4.2.6 (2.) müssen wir zeigen, daß Folgendes gilt:
Falls f.a. $t_i \in \{t_1, \ldots, t_n\}$ der Term t_i in A zu ct_i evaluiert, dann gibt es ein $ct \in CT_\Sigma$, so daß $\mathsf{is}(\mathsf{val}(f(t_1, \ldots, t_n)), ct) \in A$ und $\mathrm{I}(f, \langle ct_1, \ldots, ct_n\rangle, ct) \in A$.
Nehmen wir also an, daß t_i in A zu ct_i evaluiert. Wegen $\mathfrak{TransLit}_\Sigma(lit) \subseteq A$ und aufgrund von Satz 4.1.3 ist $\mathfrak{TransTerm}_\Sigma(t) \subseteq A$. Laut Def. von $\mathfrak{TransTerm}_\Sigma$ ist $\mathsf{search_}s(\mathsf{val}(t)) \in \mathfrak{TransTerm}_\Sigma(t)$, somit $\mathsf{search_}s(\mathsf{val}(t)) \in A$. Weil A R-saturiert und endlich ist, folgt aus $\mathsf{search_}s(\mathsf{val}(t)) \in A$ mit Satz 4.3.5, daß $\mathsf{is}(\mathsf{val}(t), ct_0) \in A$ (*), für ein $ct_0 \in CT_s$. Bleibt noch nachzuweisen, daß auch $\mathrm{I}(f, \langle ct_1, \ldots, ct_n\rangle, ct_0) \in A$, für das gleiche ct_0. Aus der Def. von $\mathfrak{TransTerm}_\Sigma$ folgt, daß
$\mathrm{I}(f, \langle \mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, \mathsf{val}(t)) \in \mathfrak{TransTerm}_\Sigma(t)$, somit gilt auch
$\mathrm{I}(f, \langle \mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, \mathsf{val}(t)) \in A$. Da
$\mathrm{I}(fv, tv, x),\ \mathsf{is}(x, z) \ \rightarrow\ \mathrm{I}(fv, tv, z)\ . \in R$, folgt aus der R-Saturiertheit, daß
$\mathrm{I}(f, \langle \mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, ct_0) \in A$.
Betrachten wir die folgenden Meta-Atome:

$$
\begin{aligned}
at_0 &= \mathrm{I}(f, \langle \mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, ct_0) \\
&\vdots \\
at_i &= \mathrm{I}(f, \langle ct_1, \ldots, ct_i, \mathfrak{Rep}(t_{i+1}), \ldots, \mathfrak{Rep}(t_n)\rangle, ct_0) \\
&\vdots \\
at_n &= \mathrm{I}(f, \langle ct_1, \ldots, ct_n\rangle, ct_0)
\end{aligned}
$$

Wir müssen zeigen, daß $at_n \in A$. Wir zeigen allgemeiner, daß $\{at_0, at_1, \ldots, at_n\} \subseteq A$. Der Induktionsbeweis verläuft analog zu dem Induktionsbeweis in Fall (3.), diesmal unter Ausnutzung von
$\mathrm{I}(fv, \langle x_1, \ldots, x_i, \ldots, x_n\rangle, y),\ \mathsf{is}(x_i, z) \ \rightarrow\ \mathrm{I}(fv, \langle x_1, \ldots, z, \ldots, x_n\rangle, y)\ . \in R$

□

Als nächstes zeigen wir, daß die Regeln für die freie Erzeugtheit auf Konstruktortermen das leisten, was man von ihnen erwartet.

Satz 4.3.7 *Sei $\langle\Sigma, AX\rangle$ eine normalisierte ADT-Spezifikation, sei $\mathfrak{TransSpec}(\langle\Sigma, AX\rangle) \subseteq R$, sei $A \subseteq MA^0_\Sigma$ ein R-saturierter MG-Ast und sei $\{ct, ct'\} \subseteq CT_s$. Dann gilt:*

1. *falls* $\mathsf{same}(ct, ct') \in A$, *dann ist* $ct = ct'$,
2. *falls* $\mathsf{different}(ct, ct') \in A$, *dann ist* $ct \neq ct'$.

Beweis: O.B.d.A. seien $C_s = \{c_1, \ldots, c_n\}$ die Konstruktoren der Sorte s.
Wir zeigen (1.) durch strukturelle Induktion über ct:
Ind.-Anf.: $ct = c_i$, $c_i \in C_s$ und $\alpha(c_i) = \lambda$.
Für jedes beliebige $j \neq i$ ($j \in \{1, \ldots, n\}$, gilt $\mathsf{same}(c_i,\ c_j)\ \rightarrow\ .\ \in R$ bzw. $\mathsf{same}(c_i,\ c_j(x_1, \ldots, x_m))\ \rightarrow\ .\ \in R$. Aus der R-Saturiertheit von A folgt, daß die Prämissen dieser Regeln allesamt nicht auf A 'matchen', weswegen der führende Konstruktor von ct' nicht c_j mit $j \neq i$ sein kann. Darum gilt $ct' = c_i = ct$.
Ind.-Schritt: Sei $ct = c_i(ct_1, \ldots, ct_m)$ mit $c_i \in C_s$ und $\alpha(c_i) = s_1 \ldots s_m$. Wie im Ind.-Anf. folgt aus der R-Saturiertheit von A, daß ct' den gleichen führenden Konstruktor hat wie ct, d.h. es gilt: $ct' = c_i(ct'_1, \ldots, ct'_m)$ für gewisse $ct'_i \in CT_{s_i}$.
Es gilt:
$\mathsf{same}(c_i(x_1, \ldots, x_m),\ c_i(y_1, \ldots, y_m))\ \rightarrow\ \mathsf{same}(x_1, y_1),\ \ldots,\ \mathsf{same}(x_m, y_m)\ .\ \in R$.
Aus der R-Saturiertheit von A folgt dann $\{\mathsf{same}(ct_1, ct'_1),\ \ldots,\ \mathsf{same}(ct_m, ct'_m)\} \subseteq A$. Aus der Ind.-Hyp. folgt $ct_i = ct'_i$, somit auch $ct = ct'$. Damit ist (1.) gezeigt.
Nun zeigen wir (2.), wieder durch strukturelle Induktion über ct:
Ind.-Anf.: $ct = c_i$, $c_i \in C_s$ und $\alpha(c_i) = \lambda$.
Es gilt $\mathsf{different}(c_i,\ c_i)\ \rightarrow\ .\ \in R$. Aus der R-Saturiertheit von A folgt, daß $ct' \neq c_i$, also $ct' \neq ct$.
Ind.-Schritt: Sei $ct = c_i(ct_1, \ldots, ct_m)$ mit $c_i \in C_s$ und $\alpha(c_i) = s_1 \ldots s_m$. Es gibt zwei Fälle: a) Der führende Konstruktor von ct' ist c_j mit $j \neq i$. Dann gilt sofort $ct' \neq ct$. b) Sei $ct' = c_i(ct'_1, \ldots, ct'_m)$ für gewisse $ct'_i \in CT_{s_i}$. Es gilt:

$$\left.\begin{array}{lll} \mathsf{different}(c_i(x_1, \ldots, x_m),\ c_i(y_1, \ldots, y_m)) & \rightarrow & \mathsf{different}(x_1,\ y_1) \\ & & ; \\ & & \vdots \\ & & ; \\ & & \mathsf{different}(x_m, y_m)\ . \end{array}\right\} \in R.$$

(Man beachte die Strichpunkte in der Konklusion. Diese Regel ist disjunktiv!)
Aus der R-Saturiertheit von A folgt dann, daß *für ein* $j \in \{1, \ldots, n\}$ gilt, daß $\mathsf{different}(ct_j, ct'_j) \in A$. Aus der Ind.-Hyp. folgt $ct_i \neq ct'_i$, somit auch $ct \neq ct'$. Damit ist (2.) gezeigt. □

Die gerade bewiesene Tatsache, daß die same- bzw. different-Regeln die an sie gesetzen Erwartungen erfüllen, können wir an zwei Stellen ausnutzen: Zunächsteinmal, um zu zeigen, daß saturierte Äste funktional sind, und danach, um nachzuweisen, daß ein saturierter Ast, der die Transformation eines Literals enthält, dieses Literal auch 'erfüllt'.

Satz 4.3.8 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, sei $\mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \subseteq R$ und sei $A \subseteq MA^0_\Sigma$ ein R-saturierter MG-Ast. Dann ist A funktional.*

Beweis: Wir zeigen zunächst, daß A val-funktional ist.
Angenommen, für $ct \in CT_\Sigma$ und $ct' \in CT_\Sigma$ gilt, daß $\mathsf{is}(\mathsf{val}(t), ct) \in A$ und $\mathsf{is}(\mathsf{val}(t), ct') \in A$ für ein $t \in T^0_\Sigma$. Es gilt: $\mathsf{is}(x, y),\ \mathsf{is}(x, y') \ \rightarrow\ \mathsf{same}(y, y')\ . \in R$. Da A R-saturiert ist, gilt $\mathsf{same}(ct, ct') \in A$. Aus Satz 4.3.7 folgt, daß $ct = ct'$.
Die I-Funktionalität von A zeigt man völlig analog, mit Hilfe von
$\mathrm{I}(fv, tv, z),\ \mathrm{I}(fv, tv, z') \ \rightarrow\ \mathsf{same}(z, z')\ . \in R$. □

Sei $ax = lit_1 \vee \ldots \vee lit_n$ ein Axiom. Wenn bei der Konstruktion eines Modells ein transformiertes Axiom angewendet wird (vgl. Def. 3.2.8), dann verzweigt der aktuelle Ast in n Unteräste. Jeder der Äste entsteht dann durch Erweiterung des alten Astes um eine instantiierte Extension $\mathfrak{TransLit}_\Sigma(lit^0_i)$ (lit^0_i ist eine Grundinstanz von lit_i). Angenommen, der Ast mit $\mathfrak{TransLit}_\Sigma(lit^0_i)$ saturiert im weiteren Verlauf der Modell-Konstruktion. Dann erwarten wir, daß dieser Ast das Grund-Litaral lit^0_i erfüllt, im Sinne von Def. 4.2.10.

Satz 4.3.9 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, sei $\mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \subseteq R$, sei $A \subseteq MA^0_\Sigma$ ein endlicher, R-saturierter MG-Ast und sei $lit \in Lit^0_\Sigma$ ein Grund-Literal. Dann gilt: falls $\mathfrak{TransLit}_\Sigma(lit) \subseteq A$, dann erfüllt A lit.*

Beweis: Um zu zeigen, daß A lit erfüllt, muß man die Bedingungen (1.) bis (4.) der Def. 4.2.10 nachweisen.

1. Laut Satz 4.3.6 ist A adäquat für $unterTerme(lit)$.

2. Laut Satz 4.3.8 ist A funktional.

3. Sei $lit = t_1 \doteq t_2$, t_1 evaluiere in A zu ct_1 und t_2 evaluiere in A zu ct_2. Wir müssen zeigen, daß $ct_1 = ct_2$ ist. Nach der Def. von $\mathfrak{TransLit}_\Sigma$ gilt: $\mathsf{same}(\mathfrak{Rep}(t_1), \mathfrak{Rep}(t_2)) \in \mathfrak{TransLit}_\Sigma(lit)$, also auch $\mathsf{same}(\mathfrak{Rep}(t_1), \mathfrak{Rep}(t_2)) \in A$. In einem ersten Schritt zeigen wir, daß $\mathsf{same}(ct_1, \mathfrak{Rep}(t_2)) \in A$. Wir unterscheiden zwei Fälle:
   a) $t_1 \in CT_\Sigma$: Dann ist laut Def. 4.2.4 $t_1 = ct_1$, und es gilt: $\mathfrak{Rep}(t_1) = \mathfrak{Rep}(ct_1) = ct_1$. Damit ist $\mathsf{same}(ct_1, \mathfrak{Rep}(t_2)) \in A$.
   b) $t_1 \notin CT_\Sigma$: Dann ist $\mathfrak{Rep}(t_1) = \mathsf{val}(t_1)$ und es gilt: $\mathsf{same}(\mathsf{val}(t_1), \mathfrak{Rep}(t_2)) \in A$. Wegen $t_1 \notin CT_\Sigma$ und Def. 4.2.4 gilt: $\mathsf{is}(\mathsf{val}(t_1), ct_1) \in A$.
   Da R die Regel $\mathsf{same}(x, y),\ \mathsf{is}(x, z) \ \rightarrow\ \mathsf{same}(z, y)\ .$ enthält, gilt wegen der R-Saturiertheit auch $\mathsf{same}(ct_1, \mathfrak{Rep}(t_2)) \in A$. (Ende b)
   In einem zweiten Schritt kann dann in analoger Weise gefolgert werden, daß mit $\mathsf{same}(ct_1, \mathfrak{Rep}(t_2)) \in A$ auch $\mathsf{same}(ct_1, ct_2) \in A$. Aus dem Satz 4.3.7 folgt dann, daß $ct_1 = ct_2$.

4. Sei $lit = t_1 \not\doteq t_2$, t_1 evaluiere in A zu ct_1 und t_2 evaluiere in A zu ct_2. Völlig analog zu (3.) kann man schließen, daß $\mathsf{different}(ct_1, ct_2) \in A$, weswegen $ct_1 \neq ct_2$.

□

Bisher haben wir uns in gewisser Weise mit MG-Ästen beschäftigt, auf welche die transformierten Axiome bereits angewendet wurden. Um uns nun der Modell-Korrektheit zu nähern, müssen wir uns noch mit der Anwendung der transformierten Axiome selbst auseinandersetzen. Hierbei kommen jetzt die Instanzen mit ins Spiel, deren 'Präsenz' auf einem MG-Ast (in Form von Sorten-Atomen) Voraussetzung ist für die Anwendung eines transformierten Axioms. Die Präsenz der Instanzen wird durch die Saturierung eines Astes gegenüber den Regeln in $\mathfrak{TransInst}_\Sigma(\mathit{Inst})$ erreicht (vgl. Def. 3.3.11).

Satz 4.3.10 *Sei Inst $\subseteq CT_\Sigma$, $\mathfrak{TransInst}_\Sigma(\mathit{Inst}) \subseteq R$ und sei A R-saturiert. Dann gilt f.a. $ct \in \mathit{Inst}$: Ist $sort(ct) = s$, dann ist $s(ct) \in A$.*

Beweis: Für jedes $ct \in \mathit{Inst}$ gilt: $\rightarrow\ s(ct)\ . \in R$, falls $ct \in CT_s$. Aus der R-Saturiertheit von A und da die Prämisse der Regel leer ist, folgt unmittelbar: $s(ct) \in A$. □

Nun betrachten wir die volle Regelmenge R, auf die in unserem Verfahren die Prozedur ***MG-proc*** angewendet wird:

$$R \;=\; \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(\mathit{Inst})$$

Wir können nun zeigen, daß ein R-saturierter Ast *ein* Disjunktionsglied jeder Formel aus $[AX/\mathit{Inst}]$ repräsentiert.

Satz 4.3.11 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, Inst $\subseteq CT_\Sigma$ und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(\mathit{Inst})$. Weiterhin sei $\varphi \in [AX/\mathit{Inst}]$, mit $\varphi = lit_1^0 \vee \ldots \vee lit_n^0$ und A ein R-saturierter Ast.*
Dann gilt für ein $lit_i^0 \in \{lit_1^0, \ldots, lit_n^0\}$, daß $\mathfrak{TransLit}_\Sigma(lit_i^0) \subseteq A$.

Beweis: Aus $\varphi \in [AX/\mathit{Inst}]$ folgt $\varphi \in [ax/\mathit{Inst}]$ für ein $ax \in AX$. Seien $\mathit{frei}(ax) = \{x_1, \ldots, x_m\}$ die freien Variablen das Axioms ax, mit $sort(x_i) = s_i$. Dann ist $\varphi = ax[x_1/ct_1, \ldots, x_m/ct_m]$ für bestimmte Konstruktorterme $ct_1, \ldots, ct_m$ mit $ct_i \in CT_{s_i}$ und $ct_i \in \mathit{Inst}$. Da $\varphi = lit_1^0 \vee \ldots \vee lit_n^0$, ist $ax = lit_1 \vee \ldots \vee lit_n$, für gewisse Literale $lit_i \in Lit$ mit $lit_i^0 = lit_i[x_1/ct_1, \ldots, x_m/ct_m]$ (*). Da $ax \in AX$, ist $\mathfrak{TransAxiom}_\Sigma(ax) \in \mathfrak{TransAxioms}_\Sigma(AX)$, also gilt $\mathfrak{TransAxiom}_\Sigma(ax) \in R$, mit

$$\begin{aligned}\mathfrak{TransAxiom}_\Sigma(ax) \;=\;& s_1(x_1)\ ,\ \ldots\ ,\ s_m(x_m)\\ &\rightarrow\ \mathfrak{TransLit}_\Sigma(lit_1)\\ &\qquad ;\\ &\qquad \vdots\\ &\qquad ;\\ &\quad \mathfrak{TransLit}_\Sigma(lit_n)\ .\end{aligned}$$

Wegen der R-Saturiertheit und Satz 4.3.10 gilt $\{s_1(ct_1), \ldots, s_m(ct_m)\} \subseteq A$. Da $\{s_1(ct_1), \ldots, s_m(ct_m)\} = \{s_1(x_1), \ldots, s_m(x_m)\}[x_1/ct_1, \ldots, x_m/ct_m]$, folgt (wiederum wegen der R-Saturiertheit) für ein $i \in \{1, \ldots, n\}$, daß
$(\mathfrak{TransLit}_\Sigma(lit_i))[x_1/ct_1, \ldots, x_m/ct_m] \subseteq A$. Dies ist die Stelle, an der wir die Substitutivität von $\mathfrak{TransLit}_\Sigma$ benötigen, s. Satz 4.1.6. Hieraus folgt
$(\mathfrak{TransLit}_\Sigma(lit_i[x_1/ct_1, \ldots, x_m/ct_m])) \subseteq A$. Wegen (*) gilt dann:
$\mathfrak{TransLit}_\Sigma(lit_i^0) \subseteq A$. □

Nun können wir unmittelbar folgern, daß von einem R-saturierten Ast immer *ein* Konjunktionsglied jeder Formel in $[AX/Inst]$ erfüllt wird. (Noch reden wir von der Erfüllung durch einen *Ast*. Erst im Anschluß ziehen wir daraus die Schlußfolgerungen für semantische Interpretationen.)

Satz 4.3.12 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, $Inst \subseteq CT_\Sigma$ und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$. Weiterhin sei $\varphi \in [AX/Inst]$, mit $\varphi = lit_1^0 \vee \ldots \vee lit_n^0$ und A ein R-saturierter Ast.*
Dann gilt für ein $lit_i^0 \in \{lit_1^0, \ldots, lit_n^0\}$, daß A lit_i^0 erfüllt.

Bemerkung 4.3.13 Vor dem (sehr kurzen) Beweis dieser Aussage überzeugen wir uns erst von der Wohlgeformtheit des Satzes: Da $\langle \Sigma, AX \rangle$ normalisiert ist, ist $AX \subseteq Kl_\Sigma$. Darum ist auch $[AX/Inst] \subseteq Kl_\Sigma$, insbesondere ist $[AX/Inst] \subseteq Kl_\Sigma^0$. Daher hat jedes $\varphi \in [AX/Inst]$ die Form $lit_1^0 \vee \ldots \vee lit_n^0$. Deshalb macht der Satz eine Aussage für *jede* Formel in $[AX/Inst]$. Außerdem gilt $lit_i^0 \in Lit_\Sigma^0$. Darum ist der Ausdruck 'A erfüllt lit_i^0' wohlgeformt.

Beweis: Der obige Satz folgt direkt aus den Sätzen 4.3.9 und 4.3.11. □

Schließlich können wir von der gerade gezeigten Erfüllung *einzelner Litarale* durch einen *Ast* in einem Doppelschritt übergehen zu der Erfüllung *aller instanziierten Axiome* durch eine *semantische Interpretation.* Wir erhalten den Satz über die Korrektheit der gefundenen Modelle bezüglich einer beschränkten Instantiierung der Axiome:

Satz 4.3.14 (instanzenbeschränkte Modell-Korrektheit)
Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, $Inst \subseteq CT_\Sigma$ endlich und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$. Falls `MG-proc` (R) mit der Ausgabe „A ist R-saturiert“ terminiert, für einen MG-Ast $A \in MA_\Sigma^0$, dann gilt:

1. *$[AX/Inst]$ ist erfüllbar,*

2. *für jede A entsprechende $\mathcal{F}$-Interpretation $\mathcal{I}$ gilt: $\mathcal{I} \models [AX/Inst]$.*

Beweis: Zunächst halten wir fest, daß laut Satz 3.4.18 aus der Ausgabe „A ist R-saturiert“ tatsächlich folgt, daß A R-saturiert ist. Sei nun φ ein beliebige Formel aus $[AX/\mathit{Inst}]$. Laut Bemerkung 4.3.13 hat φ die Form $\varphi = \mathit{lit}_1^0 \vee \ldots \vee \mathit{lit}_n^0 \in \mathit{Kl}_\Sigma^0$. Laut Satz 4.3.12 gilt für ein $\mathit{lit}_i^0 \in \{\mathit{lit}_1^0, \ldots, \mathit{lit}_n^0\}$, daß A lit_i^0 erfüllt. Laut Satz 4.2.11 gilt dann für jede A entsprechende Interpretation $\mathcal{I}$: $\mathcal{I} \models \mathit{lit}_i^0$. Nach der Semantik der Disjunktion gilt für ein solches $\mathcal{I}$ auch: $\mathcal{I} \models \varphi$. Da φ beliebig aus $[AX/\mathit{Inst}]$ gewählt war, gilt f.a. $\varphi \in [AX/\mathit{Inst}]$: $\mathcal{I} \models \varphi$. Demnach gilt auch $\mathcal{I} \models [AX/\mathit{Inst}]$, für jede A entsprechende Interpretation $\mathcal{I}$. Somit ist (2.) bewiesen. Um noch (1.) zu beweisen, fehlt nur noch die Existenz einer A entsprechenden Interpretation $\mathcal{I}$. Nach dem Satz 4.3.8 ist A funktional, und es folgt mit Satz 4.2.3, daß eine A entsprechende Interpretation $\mathcal{I}$ existiert. □

Da das tatsächlich eingesetzte Verfahren nicht mit expliziten Instanzen-Mengen arbeitet, sondern nur mit einer Größen-Beschränkung für Instanzen, behandeln wir abschließend noch diese Variante.

Satz 4.3.15 *Sei $n \in \mathbb{N}$, $\mathfrak{MaxInst}_\Sigma(n) \subseteq R$ und sei A R-saturiert. Dann gilt f.a. ct mit $|ct| \leq n$: Ist $sort(ct) = s$, dann ist $s(ct) \in A$.*

Beweis: Analog zu dem Beweis von 4.3.10 unter Verwendung der Def. 3.3.12. □

Satz 4.3.16 (größenbeschränkte Modell-Korrektheit)
Sei $\langle \Sigma, AX \rangle$ eine norm. ADT-Spez., $n \in \mathbb{N}$ und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{MaxInst}_\Sigma(n)$. Falls `MG-proc` (R) mit der Ausgabe „A ist R-saturiert“ terminiert, für einen MG-Ast $A \in MA_\Sigma^0$, dann gilt:

1. *$AX_{\leq n}$ ist erfüllbar,*
2. *für jede A entsprechende $\mathcal{F}$-Interpretation $\mathcal{I}$ gilt: $\mathcal{I} \models AX_{\leq n}$.*

Beweis: Wir erinnern daran, daß $\varphi_{\leq n} = [\varphi/\{ct \in CT_\Sigma \mid |ct| \leq n\}]$ ist und $\Phi_{\leq n} = \bigcup_{\varphi \in \Phi} \varphi_{\leq n}$. Demnach folgt dieser Satz *fast* als Spezialfall von Satz 4.3.14, mit $\mathit{Inst} = \{ct \in CT_\Sigma \mid |ct| \leq n\}$. Es ist lediglich zu beachten, daß hier mit $\mathfrak{MaxInst}_\Sigma(n)$ gearbeitet wird anstelle von $\mathfrak{TransInst}_\Sigma(\mathit{Inst})$, d.h., wir verwenden Satz 4.3.15 anstelle von Satz 4.3.10. □

4.4 Unerfüllbarkeits-Korrektheit

Im letzten Abschnitt waren wir davon ausgegangen, daß die Modell-Generierung mit der Saturierung eines Astes terminiert. Hieraus war zu folgern, daß ein Modell der (instantiierten Spezifikation) existiert. Wir mußten dabei von gegebenen

Meta-Atom-Mengen (mit bestimmten Eigenschaften) auf semantische Interpretationen schließen. In diesem und dem folgenden Abschnitt müssen wir nun den umgekehrten Weg gehen. Diesmal setzen wir die Erfüllbarkeit der Axiome, d.h. die Existenz einer die Axiome erfüllenden Interpretation, voraus, und schließen von da aus auf die Beschaffenheit der konstruierten MG-Bäume und -Äste. Die Elemente der MG-Äste, die Meta-Atome, repräsentieren ja Aussagen über Interpretationen. Ist eine Interpretation $\mathcal{I}$ vorgegeben, dann kann man die Meta-Atome (wie z.B. $\mathrm{I}(f, \langle 0 \rangle, 1)$) als Aussage über $\mathcal{I}$ *deuten*. Diese Aussagen können dann zutreffen oder auch nicht.

Die zentrale Eigenschaft, die es nun abzuleiten gilt, ist die folgende: Wenn eine erfüllende Interpretation $\mathcal{I}$ der Axiome existiert, dann gibt es zu jedem Zeitpunkt der Modell-Konstruktion immer *mindestens einen* MG-Ast A, so daß die Deutung aller Meta-Atome in A mittels $\mathcal{I}$ *zutrifft* (*). Gelingt es, dies zu zeigen, dann folgen zwei wichtige Aussagen über die Modell-Konstruktion:

1. Es können nicht alle Äste verworfen werden. Per Kontraposition folgt sofort die *Unerfüllbarkeits-Korrektheit*: Wenn doch alle Äste verworfen werden, und ***MG-proc*** mit „abgelehnt“ anwortet, dann kann keine die Axiome erfüllende Interpretation existiert haben.

2. Wenn man noch zeigt, daß jeder MG-Ast, bei dem die Deutung aller Meta-Atome mittels $\mathcal{I}$ zutrifft, eine bestimme Maximal-Länge nicht überschreiten kann, dann muß eine *Ast-faire* Verfeinerung von ***MG-proc*** in endlicher Zeit einen saturierten Ast finden, und mit der Ausgabe „A ist R-saturiert“ terminieren. Dies ist die *Modell-Vollständigkeit*.

In diesem Abschnitt formalisieren und beweisen wir die Aussage (*). Fast beiläufig können wir daraus die Aussage (1.) folgern. Der kommende Abschnitt baut auf die Aussage (*) auf. Dort wird der Begriff der ‘Fairneß’ präzisiert und die Terminierung der fairen Modell-Suche nachgewiesen.

Um Meta-Atome mittels eines vorgegebenen $\mathcal{I}$ deuten zu können, müssen zuerst ihre Bestandteile, die Meta-Terme, mittels $\mathcal{I}$ ‘ausgewertet’ werden. Hier und im folgenden gilt es zu beachten, daß MG-Äste Mengen von *Grund*-Meta-Atomen sind. Die vorkommenden Meta-Terme sind folglich auch *grundiert*. Wir definieren die Auswertung nur auf *wohlgeformten* Meta-Termen (s. Def. 3.4.4). Diese sind insbesondere grundiert.

Definition 4.4.1 (Auswertung von Grund-Meta-Termen)
Sei $\mathcal{I}$ gegeben, die **Auswertung von Grund-Meta-Termen** mittels $\mathcal{I}$, $mtval_{\mathcal{I}} : wMT_{\Sigma} \rightarrow CT_{\Sigma}$, ist definiert durch:

1. $mtval_{\mathcal{I}}(\mathsf{val}(t)) = val_{\mathcal{I}}(t)$

2. $mtval_{\mathcal{I}}(\mathsf{arg}n(t)) = (mtval_{\mathcal{I}}(t))\downarrow_n$
3. $mtval_{\mathcal{I}}(c) = c$, für $c \in \overline{\mathcal{C}}$, $\alpha(c) = \lambda$
4. $mtval_{\mathcal{I}}(c(t_1, \ldots, t_n)) = c(mtval_{\mathcal{I}}(t_1), \ldots, mtval_{\mathcal{I}}(t_n))$,
   für $c \in \overline{\mathcal{C}}$, $\alpha(c) = s_1, \ldots, s_n$

⋆

Intuitiv löst also val eine Auswertung mittels $val_{\mathcal{I}}$ aus, arg$n$ steht für die n-te Argumentposition und die Konstruktoren für sich selbst. Das Ergebnis von $mtval_{\mathcal{I}}$ ist immer ein Konstruktorterm! Es ist leicht zu sehen, daß Konstruktorterme unter $mtval_{\mathcal{I}}$ unverändert bleiben: $mtval_{\mathcal{I}}(ct) = ct$ ($ct \in CT_\Sigma$).

Mit Hilfe von $mtval_{\mathcal{I}}$ laßt sich nun definieren, wie die Meta-Atome unter einer vorgegebenen Interpretation $\mathcal{I}$ zu deuten sind. Das Ungewöhnliche an der folgenden Definition ist, daß das Ergebnis der definierten Deutung jeweils eine mathematisch meta-sprachliche Aussage ist.

Definition 4.4.2 (Deutung von Grund-Meta-Atomen)
Sei $\mathcal{I}$ vorgegeben. Die $\mathcal{I}$ entsprechende **Deutung von Grund-Meta-Atomen** als meta-sprachliche Aussagen ist folgendermaßen definiert:
$Deut_{\mathcal{I}}(s(ct)) \;\equiv\; [ct \in CT_s]$
$Deut_{\mathcal{I}}(\mathsf{search_}s(t)) \;\equiv\; [mtval_{\mathcal{I}}(t) \in CT_s]$
$Deut_{\mathcal{I}}(\mathsf{is}(t_1, t_2)) \;\equiv\; [mtval_{\mathcal{I}}(t_1) = mtval_{\mathcal{I}}(t_2)]$
$Deut_{\mathcal{I}}(\mathsf{same}(t_1, t_2)) \;\equiv\; [mtval_{\mathcal{I}}(t_1) = mtval_{\mathcal{I}}(t_2)]$
$Deut_{\mathcal{I}}(\mathsf{different}(t_1, t_2)) \;\equiv\; [mtval_{\mathcal{I}}(t_1) \neq mtval_{\mathcal{I}}(t_2)]$
$Deut_{\mathcal{I}}(\mathrm{I}(a, \langle\rangle, t)) \;\equiv\; [\mathcal{I}(a)() = mtval_{\mathcal{I}}(t)]$
$Deut_{\mathcal{I}}(\mathrm{I}(f, \langle t_1, \ldots, t_n\rangle, t)) \;\equiv\; [\mathcal{I}(f)(mtval_{\mathcal{I}}(t_1), \ldots, mtval_{\mathcal{I}}(t_n)) = mtval_{\mathcal{I}}(t)]$
Hierbei wird $mtval_{\mathcal{I}}$ auf der rechten Seite rekursiv aufgelöst, bis ein gänzlich meta-sprachlicher Ausdruck entsteht. ⋆

Der jeweils entstehende Ausdruck kann zutreffen oder auch nicht. Um ganze MG-Äste, aber auch z.B. Regelprämissen deuten zu können, erweitern wir $Deut_{\mathcal{I}}$ auf Mengen $A \in MA^0_\Sigma$:

Definition 4.4.3 (zutreffende Deutung von Grund-MA-Mengen)
Sei $\mathcal{I}$ vorgegeben, und sei $A \in MA^0_\Sigma$. Wir sagen, die **Deutung der Menge** A **trifft zu**, falls $Deut_{\mathcal{I}}(at)$ zutrifft f.a. $at \in A$. ⋆

Insbesondere trifft $Deut_{\mathcal{I}}(\emptyset)$ zu, für jedes $\mathcal{I}$.

Unser Ziel ist es nun, zu zeigen, daß sich das Zutreffen der gedeuteten Prämissen einer Regel auf *mindestens eine* der Regel-Extensionen vererbt. Zur Vorbereitung darauf zeigen wir noch zweierlei: a) $\mathfrak{Rep}$ wird wie $val_{\mathcal{I}}$ gedeutet, und b)

die Deutung derjenigen Meta-Atome, die aus der Transformation von Termen $\mathfrak{TransTerm}_\Sigma$ hervorgehen, trifft immer zu, unabhängig von $\mathcal{I}$.

Satz 4.4.4 *Sei $\mathcal{I}$ eine $\mathcal{F}$-Interpretation und $t \in T^0_\Sigma$ ein Grund-Term. Dann gilt:*

$$mtval_\mathcal{I}(\mathfrak{Rep}(t)) = val_\mathcal{I}(t).$$

Beweis: Sei $t \notin FFT_\Sigma$ (t enthält Funktionen), dann ist $\mathfrak{Rep}(t) = \mathsf{val}(t)$, und aus der Def. der Deutung folgt die Beh. Sei $t \in FFT_\Sigma$. Dann ist $\mathfrak{Rep}(t) = t$, demnach $mtval_\mathcal{I}(\mathfrak{Rep}(t)) = mtval_\mathcal{I}(t)$. Da $FFT_\Sigma \cap T^0_\Sigma = CT_\Sigma$, ist $t \in CT_\Sigma$. Daraus folgt $mtval_\mathcal{I}(t) = t$ sowie $val_\mathcal{I}(t) = t$. Zusammen gilt $mtval_\mathcal{I}(\mathfrak{Rep}(t)) = mtval_\mathcal{I}(t) = t = val_\mathcal{I}(t)$. □

Satz 4.4.5 *Sei $\mathcal{I}$ eine $\mathcal{F}$-Interpretation und $t \in T^0_\Sigma$ ein Grund-Term. Dann gilt:*

$$Deut_\mathcal{I}(\mathfrak{TransTerm}_\Sigma(t)) \text{ trifft zu.}$$

Beweis: Strukturelle Induktion:

1. Falls $t \in CT_\Sigma$, dann ist $\mathfrak{TransTerm}_\Sigma(t) = \emptyset$.

2. Falls $t = a$ mit $a \in F_s$, $\alpha(a) = \lambda$, dann ist
   $\mathfrak{TransTerm}_\Sigma(t) = \{\mathrm{I}(a, \langle\rangle, \mathsf{val}(a)),\ \mathsf{search_}s(\mathsf{val}(a))\}$.
   $Deut_\mathcal{I}(\mathrm{I}(a, \langle\rangle, \mathsf{val}(a))) \equiv [\mathcal{I}(a)() = mtval_\mathcal{I}(\mathsf{val}(a))] \equiv [\mathcal{I}(a)() = val_\mathcal{I}(a)]$
   $Deut_\mathcal{I}(\mathsf{search_}s(\mathsf{val}(a))) \equiv [mtval_\mathcal{I}(\mathsf{val}(a)) \in CT_s] \equiv [val_\mathcal{I}(a) \in CT_s]$
   Beide Ausdrücke treffen offensichtlich zu.

3. Falls $t = f(t_1, \ldots, t_n)$ mit $f \in F_s$, $\alpha(f) = s_1 \ldots s_n$, dann ist
   $\mathfrak{TransTerm}_\Sigma(t) = \{\mathrm{I}(f, \langle\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, \mathsf{val}(t)),\ \mathsf{search_}s(\mathsf{val}(t))\}$
   $\cup\, \mathfrak{TransTerm}_\Sigma(t_1) \cup \ldots \cup \mathfrak{TransTerm}_\Sigma(t_n)$
   $Deut_\mathcal{I}(\mathfrak{TransTerm}_\Sigma(t_i))$ trifft aufgrund der Ind.-Hyp. zu.
   $Deut_\mathcal{I}(\mathsf{search_}s(\mathsf{val}(t))) \equiv [val_\mathcal{I}(t) \in CT_s]$ (s.o.) trifft zu.
   $Deut_\mathcal{I}(\mathrm{I}(f, \langle\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, \mathsf{val}(t)))$
   $\equiv [\mathcal{I}(f)(mtval_\mathcal{I}(\mathfrak{Rep}(t_1)), \ldots, mtval_\mathcal{I}(\mathfrak{Rep}(t_n))) = mtval_\mathcal{I}(\mathsf{val}(t))]$
   $\equiv [\mathcal{I}(f)(val_\mathcal{I}(t_1), \ldots, val_\mathcal{I}(t_n)) = val_\mathcal{I}(t)]$.
   Dies trifft zu nach der Def. von $val_\mathcal{I}$.

4. Falls $t = c(t_1, \ldots, t_n)$ mit $c \in C_s$, $\alpha(c) = s_1 \ldots s_n$, *wobei* $t \notin CT_\Sigma$, dann ist
   $\mathfrak{TransTerm}_\Sigma(t) = \{\mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)))\}$
   $\cup\, \mathfrak{TransTerm}_\Sigma(t_1) \cup \ldots \cup \mathfrak{TransTerm}_\Sigma(t_n)$
   $Deut_\mathcal{I}(\mathfrak{TransTerm}_\Sigma(t_i)$ trifft wieder aufgrund der Ind.-Hyp. zu.
   $Deut_\mathcal{I}(\mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n))))$
   $\equiv [mtval_\mathcal{I}(\mathsf{val}(t)) = mtval_\mathcal{I}(c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)))]$
   $\equiv [val_\mathcal{I}(t) = c(val_\mathcal{I}(t_1), \ldots, val_\mathcal{I}(t_n))]$
   Auch dies trifft zu nach der Def. von $val_\mathcal{I}$.

□

Nun können wir den zentralen Satz dieses Abschnitts zeigen, wonach sich bei Anwendung einer Regel das Zutreffen der Deutung von der Regel-Prämisse auf *mindestens eine* der Regel-Extensionen vererbt.

Satz 4.4.6 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, $Inst \subseteq CT_\Sigma$ und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$. Weiterhin sei $\mathcal{I}$ eine (feste) $\mathcal{F}$-Interpretation, welche die Axiome erfüllt: $\mathcal{I} \models AX$. Dann gilt für jede MG-Regel $r \in R$: falls für eine Meta-Variablen-Substitution σ gilt, daß $(prämisse(r))\sigma \in MA^0_\Sigma$ und daß $Deut_\mathcal{I}((prämisse(r))\sigma)$ zutrifft, dann gibt es eine Extension $Ext \in extensionen(r)$, so daß auch $Deut_\mathcal{I}((Ext)\sigma)$ zutrifft.*

In dem folgenden Beweis wird jeweils beim Nachweis von $Deut_\mathcal{I}((Ext)\sigma)$ implizit mitbewiesen, daß $Deut_\mathcal{I}((Ext)\sigma)$ *wohlgeformt* ist. Daraus folgt insbesondere auch die *Wohlsortiertheit* von $(Ext)\sigma$. Umgekehrt können wir jeweils die Wohlsortiertheit von $(prämisse(r))\sigma$ und die Wohlgeformtheit von $Deut_\mathcal{I}((prämisse(r))\sigma)$ voraussetzen.

Beweis: Wir treffen eine große Fallunterscheidung über die verschiedenen Regelmengen, die sich ganz natürlich aus der Def. von $\mathfrak{TransSpec}$ und $\mathfrak{TransInst}_\Sigma$ ergeben. Für positive Regeln, die keine Prämisse besitzen und darum auch keine Variablen enthalten, müssen wir nur zeigen, daß die Deutung einer (meist der einzigen) Extension mit $Deut_\mathcal{I}$ zutrifft.

Fall 1.: $r \in \mathfrak{TransInst}_\Sigma(Inst)$
$r = \rightarrow\ s(ct)$. für ein $ct \in Inst \cap CT_s$. $Deut_\mathcal{I}(s(ct)) \equiv [ct \in CT_s]$, was zutrifft.

Fall 2.: $r \in \mathfrak{TransAxioms}_\Sigma(AX)$
$r \in \mathfrak{TransAxiom}_\Sigma(ax)$ für ein $ax \in AX$. Sei $ax = lit_1 \vee \ldots \vee lit_n$, $Var(ax) = \{x_1, \ldots x_m\}$ und $sort(x_i) = s_i$. Es gilt:

$$\begin{array}{rcl} \mathfrak{TransAxiom}_\Sigma(ax) & = & s_1(x_1)\ ,\ \ldots\ ,\ s_m(x_m) \\ & & \rightarrow\ \ \mathfrak{TransLit}_\Sigma(lit_1) \\ & & \qquad ; \\ & & \qquad \vdots \\ & & \qquad ; \\ & & \quad \mathfrak{TransLit}_\Sigma(lit_n)\ . \end{array}$$

Sei nun σ eine MV-Substitution, so daß $Deut_\mathcal{I}(\{s_1(x_1), \ldots, s_m(x_m)\}\sigma)$ zutrifft. Wir müssen zeigen, daß für ein $j \in \{1, \ldots, n\}$ $Deut_\mathcal{I}(\mathfrak{TransLit}_\Sigma(lit_j)\sigma)$ zutrifft. Wegen $(s_i(x_i))\sigma = s_i(\sigma(x_i))$ trifft $Deut_\mathcal{I}(s_i(\sigma(x_i)))$ zu f.a. $i \in \{1, \ldots m\}$. Nach Def. 4.4.2 gilt dann $\sigma(x_i) \in CT_{s_i}$ (*). σ ist ja eine MV-Substitution und als solche nicht per definitionem wohlsortiert. Aber aufgrund von (*) ist die Einschränkung σ' von σ auf $Var(ax)$, $\sigma' = \sigma|_{Var(ax)}$ wohlsortiert, also eine ganz gewöhnliche Substitution im Sinne von Def. 2.2.18. Damit gelten für σ' alle Sätze, die wir für gewöhnliche Substitutionen gezeigt haben.
Nun gilt $\mathcal{I} \models ax$. Nach Satz 3.3.5 gilt dann $\mathcal{I} \models [ax/CT_\Sigma]$, wegen (*) insbesondere auch $\mathcal{I} \models (ax)\sigma'$. Wegen $(ax)\sigma' = lit_1\sigma' \vee \ldots \vee lit_n\sigma'$ und der Semantik von

'$\vee$' gilt auch $\mathcal{I} \models lit_j\sigma'$ für ein $j \in \{1,\ldots,n\}$. Für dieses j zeigen wir zunächst, daß $Deut_{\mathcal{I}}((\mathfrak{TransLit}_\Sigma(lit_j))\sigma')$ zutrifft. O.B.d.A. sei $lit_j = t_1 \doteq t_2$ (der Fall $lit_j = t_1 \not\doteq t_2$ geht völlig analog). Da σ' eine vollständige Konstruktorterm-Substitution für t_1 und t_2 ist, gilt lt. Satz 4.1.6 $(\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2))\sigma' = \mathfrak{TransLit}_\Sigma((t_1 \doteq t_2)\sigma')$. Diese Menge ist gleich $\{\mathsf{same}(\mathfrak{Rep}(t_1\sigma'),\mathfrak{Rep}(t_2\sigma'))\} \cup \mathfrak{TransTerm}_\Sigma(t_1\sigma') \cup \mathfrak{TransTerm}_\Sigma(t_2\sigma')$. Da $Deut_{\mathcal{I}}(\mathfrak{TransTerm}_\Sigma(t))$ für beliebige Terme t immer zutrifft (Satz 4.4.5), müssen wir nur noch zeigen, daß
$Deut_{\mathcal{I}}(\mathsf{same}(\mathfrak{Rep}(t_1\sigma'),\mathfrak{Rep}(t_2\sigma'))) \equiv [mtval_{\mathcal{I}}(\mathfrak{Rep}(t_1\sigma')) = mtval_{\mathcal{I}}(\mathfrak{Rep}(t_2\sigma'))]$ zutrifft. Nach Satz 4.4.4 gilt dies genau dann, wenn $val_{\mathcal{I}}(t_1\sigma') = val_{\mathcal{I}}(t_2\sigma')$. Dies aber ist eine unmittelbare Folge aus $\mathcal{I} \models lit_j\sigma'$ (s.o.). Somit ist gezeigt, daß $Deut_{\mathcal{I}}((\mathfrak{TransLit}_\Sigma(lit_j))\sigma')$ zutrifft.
Laut Satz 4.1.9 gilt $Var(\mathfrak{TransLit}_\Sigma(lit_j)) = Var(lit_j)$, und wegen $Var(lit_j) \subseteq Var(ax)$ und $\sigma' = \sigma|_{Var(ax)}$ gilt $(\mathfrak{TransLit}_\Sigma(lit_j))\sigma' = (\mathfrak{TransLit}_\Sigma(lit_j))\sigma$. Somit gilt auch, daß $Deut_{\mathcal{I}}((\mathfrak{TransLit}_\Sigma(lit_j))\sigma)$ zutrifft.

Fall 3.: $r \in \mathfrak{TransSorts}_\Sigma(S)$
$r \in \mathfrak{TransSort}_\Sigma(s)$ für ein $s \in S$. Seien $C_s = \{c_1,\ldots,c_n\}$ die Konstruktoren von s. Wir nehmen an, daß $Deut_{\mathcal{I}}((\mathsf{search_}s(x))\sigma) \equiv [\sigma(x) \in CT_s]$ zutrifft. Wir müssen zeigen, daß für ein $i \in \{1,\ldots,n\}$ $Deut_{\mathcal{I}}((\mathfrak{TransKonstr}_\Sigma(x,c_i))\sigma)$ zutrifft. Wegen $\sigma(x) \in CT_s$ gilt entweder a) $\sigma(x) = c_i$ für ein $c_i \in C_s$ mit $\alpha(c_i) = \lambda$, oder b) $\sigma(x) = c_i(ct_1,\ldots,ct_m)$ für ein $c_i \in C_s$ mit $\alpha(c_i) = s_1 \ldots s_m$. Für dieses i zeigen wir, daß $Deut_{\mathcal{I}}((\mathfrak{TransKonstr}_\Sigma(x,c_i))\sigma)$ zutrifft.
Falls $\alpha(c_i) = \lambda$, dann ist $(\mathfrak{TransKonstr}_\Sigma(x,c_i))\sigma = \{\mathsf{is}(\sigma(x),c_i)\}$, und
$Deut_{\mathcal{I}}(\mathsf{is}(\sigma(x),c_i)) \equiv [mtval_{\mathcal{I}}(\sigma(x)) = mtval_{\mathcal{I}}(c_i)] \equiv [\sigma(x) = c_i]$. Dies trifft zu wg. a).
Falls $\alpha(c_i) = s_1 \ldots s_m$, dann ist

$$(\mathfrak{TransKonstr}_\Sigma(x,c_i))\sigma = \{\mathsf{is}(\sigma(x),c(\mathsf{arg1}(\sigma(x)),\ldots,\mathsf{arg}m(\sigma(x)))),\\ \mathsf{search_}s_1(\mathsf{arg1}(\sigma(x))),\ldots,\mathsf{search_}s_m(\mathsf{arg}m(\sigma(x)))\}.$$

$Deut_{\mathcal{I}}(\mathsf{is}(\sigma(x),c_i(\mathsf{arg1}(\sigma(x)),\ldots,\mathsf{arg}m(\sigma(x)))))$
$\equiv [mtval_{\mathcal{I}}(\sigma(x)) = mtval_{\mathcal{I}}(c_i(\mathsf{arg1}(\sigma(x)),\ldots,\mathsf{arg}m(\sigma(x))))]$
$\equiv [\sigma(x) = c_i(mtval_{\mathcal{I}}(\mathsf{arg1}(\sigma(x))),\ldots,mtval_{\mathcal{I}}(\mathsf{arg}m(\sigma(x))))]$
$\equiv [\sigma(x) = c_i(\sigma(x){\downarrow_1},\ldots,\sigma(x){\downarrow_m})]$ (*)
Da $\sigma(x) = c_i(ct_1,\ldots,ct_m)$ (siehe b)), ist die Gleichung (*) wohlgeformt, denn die $\sigma(x){\downarrow_1},\ldots,\sigma(x){\downarrow_m}$ sind wohlgeformte Terme. Mit $\sigma(x){\downarrow_j} = ct_j$ folgt, daß (*) zutrifft. Nun bleibt nur noch zu zeigen, daß $Deut_{\mathcal{I}}(\mathsf{search_}s_j(\mathsf{arg}j(\sigma(x)))) \equiv [mtval_{\mathcal{I}}(\mathsf{arg}j(\sigma(x))) \in CT_{s_j}] \equiv [\sigma(x){\downarrow_j} \in CT_{s_j}]$ zutrifft, f.a. $j \in \{1,\ldots,m\}$. Dies aber gilt nach b).

Fall 4.: $r \in \mathfrak{Funktionalitaet}$
Sei $r = \mathrm{I}(fv,tv,z),\mathrm{I}(fv,tv,z') \rightarrow \mathsf{same}(z,z')$.
Wir nehmen an, daß $Deut_{\mathcal{I}}(\{(\mathrm{I}(fv,tv,z))\sigma,(\mathrm{I}(fv,tv,z'))\sigma\})$ zutrifft. Wir müssen zeigen, daß $Deut_{\mathcal{I}}(\mathsf{same}(z\sigma,z'\sigma)) \equiv [mtval_{\mathcal{I}}(\sigma(z)) = mtval_{\mathcal{I}}(\sigma(z'))]$ (*) zutrifft. Da $\mathrm{I}(fv,tv,z)\sigma$ und $\mathrm{I}(fv,tv,z')\sigma$ Grund-Meta-Atome sind, ist $\sigma(fv) \in \overline{\mathcal{F}}$ und $\sigma(tv)$ ein Tupel von Grund-Meta-Termen. O.B.d.A. sei $\sigma(fv) = f \in F_s$ mit $\alpha(f) =$

$s_1 \ldots s_n$. Wegen der Wohlgeformtheit der Aussage $Deut_{\mathcal{I}}((\mathrm{I}(fv, tv, z))\sigma)$ ist dann $\sigma(tv) = \langle t_1, \ldots, t_n \rangle$. Es gilt
$Deut_{\mathcal{I}}((\mathrm{I}(fv, tv, z))\sigma) \equiv [\mathcal{I}(f)(mtval_{\mathcal{I}}(t_1), \ldots, mtval_{\mathcal{I}}(t_n)) = mtval_{\mathcal{I}}(\sigma(z))]$ und
$Deut_{\mathcal{I}}((\mathrm{I}(fv, tv, z'))\sigma) \equiv [\mathcal{I}(f)(mtval_{\mathcal{I}}(t_1), \ldots, mtval_{\mathcal{I}}(t_n)) = mtval_{\mathcal{I}}(\sigma(z'))]$.
Da beides zutrifft, trifft auch (*) zu.
Der Fall $r = \mathsf{is}(x, y), \mathsf{is}(x, y') \rightarrow \mathsf{same}(y, y')$. verläuft entsprechend.

Fall 5.: $r \in \mathfrak{Ersetzung}_{\Sigma}$
Hier ist der Beweis trivial, durch unmittelbares Einsetzen mit der jeweiligen aus $Deut_{\mathcal{I}}(\mathsf{is}(t_1, t_2))$ gewonnenen Gleichung $[mtval_{\mathcal{I}}(t_1) = mtval_{\mathcal{I}}(t_2)]$.

Fall 6.: $r \in \mathfrak{FreiErz}_{\Sigma}$
In $\mathfrak{FreiErz}_{\Sigma}$ befinden sich u.a. die einzigen negativen Regeln (Regeln mit leerer Konklusion) der gesamten Regelmenge R. Auch für diese müssen wir zeigen, daß, wann immer $Deut_{\mathcal{I}}((prämisse(r))\sigma)$ zutrifft, für eine Extension $Ext \in extensionen(r)$ $Deut_{\mathcal{I}}((Ext)\sigma)$ zutrifft. Da aber $extensionen(r) = \emptyset$ ist, müssen wir die Annahme, daß $Deut_{\mathcal{I}}((prämisse(r))\sigma)$ zutrifft, ad absurdum führen. Bei nicht-negativen Regeln verfahren wir wie gehabt. Exemplarisch für ganz $\mathfrak{FreiErz}_{\Sigma}$ behandeln wir nur zwei Regeltypen.
a) Sei $r = \mathsf{same}(c_1(x_1, \ldots, x_n), c_2(y_1, \ldots, y_m)) \rightarrow$. , wobei $c_1 \neq c_2$.
Wir nehmen $(\mathsf{same}(c_1(x_1, \ldots, x_n), c_2(y_1, \ldots, y_m)))\sigma \in MA^0_{\Sigma}$ an.
Würde $Deut_{\mathcal{I}}((\mathsf{same}(c_1(x_1, \ldots, x_n), c_2(y_1, \ldots, y_m)))\sigma)$ zutreffen, dann auch
$[c_1(mtval_{\mathcal{I}}(\sigma(x_1)), \ldots, mtval_{\mathcal{I}}(\sigma(x_n))) = c_2(mtval_{\mathcal{I}}(\sigma(y_1)), \ldots, mtval_{\mathcal{I}}(\sigma(y_m)))]$.
Daraus folgt insbesondere $c_1 = c_2$. Widerspruch.
b) Sei
$r = \mathsf{different}(c(x_1, \ldots, x_n), c(y_1, \ldots, y_n)) \rightarrow \mathsf{different}(x_1, y_1); \ldots; \mathsf{different}(x_n, y_n)$.
Wir nehmen an, daß $(\mathsf{different}(c(x_1, \ldots, x_n), c(y_1, \ldots, y_n)))\sigma \in MA^0_{\Sigma}$ und daß $Deut_{\mathcal{I}}((\mathsf{different}(c(x_1, \ldots, x_n), c(y_1, \ldots, y_n)))\sigma)$
$\equiv [c(mtval_{\mathcal{I}}(\sigma(x_1)), \ldots, mtval_{\mathcal{I}}(\sigma(x_n))) \neq c(mtval_{\mathcal{I}}(\sigma(y_1)), \ldots, mtval_{\mathcal{I}}(\sigma(y_n)))]$
zutrifft. Dann aber muß $mtval_{\mathcal{I}}(\sigma(x_i)) \neq mtval_{\mathcal{I}}(\sigma(y_i))$ gelten für mindestens ein $i \in \{1, \ldots, n\}$. Dies ist gleichbedeutend damit, daß $Deut_{\mathcal{I}}((\mathsf{different}(x_i, y_i))\sigma)$ zutrifft für ein i, was zu zeigen war. □

Im folgenden nehmen wir an, die untersuchte Spezifikation sei erfüllbar mittels $\mathcal{I}$. Aus der gerade bewiesenen Eigenschaft einzelner Regelanwendungen können wir jetzt induktiv schließen, daß jeder beim Ablauf von ***MG-proc***(R) konstruierte Baum (wir nennen solche Bäume R-erreichbar) die Eigenschaft hat, daß für mindestens einen Ast die Deutung mittels $\mathcal{I}$ zutrifft. Vorher definieren wir noch (unter Verweis auf die Def. 3.4.19 von 'Läufen') 'R-erreichbare Bäume' sowie 'Iterationen' und darin 'ausgewählte Äste'.

Definition 4.4.7 (R-erreichbare Bäume und Äste)
Sei R eine Menge von MG-Regeln. Ein MG-**Baum** B ist R-**erreichbar**, falls eine der folgenden Bedingungen erfüllt ist.

1. Es existiert ein terminierender Lauf $B_1, \ldots, B_e$ von $\mathtt{MG\text{-}proc}(R)$ und es ist $B = B_i$ für ein $i \in \{1, \ldots, e\}$.
2. Es existiert ein nicht-terminierender Lauf $B_1, B_2, \ldots\ldots$ von $\mathtt{MG\text{-}proc}(R)$ und es ist $B = B_i$ für ein $i \geq 1$.

Ein MG-**Ast** A ist R**-erreichbar**, falls $A \in \mathit{äste}(B)$ für einen R-erreichbaren MG-Baum. ⋆

Definition 4.4.8 (Iterationen, ausgewählte Äste)
Sei R gegeben, sei $B_1, \ldots, B_e$ bzw. $B_1, B_2, \ldots\ldots$ ein Lauf von $\mathtt{MG\text{-}proc}(R)$ und sei $1 \leq i < e$ bzw. $i \geq 1$. Der Berechnungsschritt beim Übergang von B_i nach B_{i+1} heißt i**-te Iteration** des Laufes. Der dabei in Schritt 2. ausgewählte Ast heißt **in der i-ten Iteration ausgewählter Ast**. ⋆

Satz 4.4.9 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, $Inst \subseteq CT_\Sigma$ und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$. Weiterhin sei $\mathcal{I}$ eine (feste) $\mathcal{F}$-Interpretation, welche die Axiome erfüllt: $\mathcal{I} \models AX$. Dann gilt für jeden R-erreichbaren MG-Baum B, daß ein Ast $A \in \mathit{äste}(B)$ existiert, dessen Deutung mittels $\mathcal{I}$, $\mathit{Deut}_\mathcal{I}(A)$, zutrifft.*

Beweis: Induktion über die Iterationen von $\mathtt{MG\text{-}proc}(R)$.
Ind.-Anf.: $B_1 = \lceil \emptyset \rceil$
$\mathit{Deut}_\mathcal{I}(\emptyset)$ trifft trivialerweise zu.

Ind.-Schritt: $B_i \;\rightarrow\; B_{i+1}$
Sei $B_i = \lceil A_1 | \ldots | A_n \rceil$. Laut Ind.-Hyp. existiert ein Ast $A_k \in \{A_1, \ldots, A_n\}$, so daß $\mathit{Deut}_\mathcal{I}(A_k)$ zutrifft.
Fall a) $B_{i+1} = \mathit{Erw}(A_j, B, r, \sigma)$ für ein $A_j \in \{A_1, \ldots, A_n\}$. Laut $\mathtt{MG\text{-}proc}$ gilt für ein r mit $r = \mathit{Pr} \rightarrow \mathit{Ext}_1; \ldots; \mathit{Ext}_m.$ und ein σ, daß $(\mathit{Pr})\sigma \subseteq A_j$, und damit:
$B_{i+1} = \lceil A_1 | \ldots | A_{i-1} | A_j \cup (\mathit{Ext}_1)\sigma | \ldots | A_j \cup (\mathit{Ext}_m)\sigma | A_{i+1} | \ldots | A_n \rceil$.
Falls nun $k \neq j$, dann ist $A_k \in \mathit{äste}(B_{i+1})$ und wir sind fertig. Sei also $k = j$, dann trifft mit $\mathit{Deut}_\mathcal{I}(A_j)$ auch $\mathit{Deut}_\mathcal{I}((\mathit{Pr})\sigma)$ zu. Laut Satz 4.4.6 trifft dann auch $\mathit{Deut}_\mathcal{I}((\mathit{Ext}_n)\sigma)$ zu, für ein $n \in \{1, \ldots, m\}$. Insgesamt trifft dann für den neuen Ast $A_j \cup (\mathit{Ext}_n)\sigma$ die Deutung $\mathit{Deut}_\mathcal{I}(A_j \cup (\mathit{Ext}_n)\sigma)$ zu.
Fall b) $B_{i+1} = \mathit{Verw}(A_j, B, r, \sigma)$ für ein $A_j \in \{A_1, \ldots, A_n\}$. Laut $\mathtt{MG\text{-}proc}$ gilt für ein r mit $r = \mathit{Pr} \rightarrow \;.$ und ein σ, daß $(\mathit{Pr})\sigma \subseteq A_j$. Wäre nun $k = j$, dann würde $\mathit{Deut}_\mathcal{I}(A_j)$ zutreffen, also auch $\mathit{Deut}_\mathcal{I}((\mathit{Pr})\sigma)$. Laut Satz 4.4.6 müßte dann eine Extension $\mathit{Ext} \in \mathit{extensionen}(r)$ existieren, so daß $\mathit{Deut}_\mathcal{I}((\mathit{Ext})\sigma)$ zutrifft. Dies steht im Widerspruch zu $\mathit{extensionen}(r) = \emptyset$. Daher muß $k \neq j$ sein. Dann aber ist, wegen $B_{i+1} = \lceil A_1 | \ldots | A_{j-1} | A_{j+1} | \ldots | A_n \rceil$, $A_k \in \mathit{äste}(B_{i+1})$. □

Wie schon in der Einleitung zu diesem Abschnitt erläutert, ist die gerade bewiesene Eigenschaft R-erreichbarer Bäume der Schlüssel sowohl für die Unerfüllbarkeits-Korrektheit (s. der folgende Satz) als auch für die Modell-Vollständigkeit (s. der kommende Abschnitt) der Methode.

Satz 4.4.10 (Unerfüllbarkeits-Korrektheit)
Sei $\langle \Sigma, AX \rangle$ *eine normalisierte ADT-Spezifikation,* $Inst \subseteq CT_\Sigma$ *und sei* $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$*. Dann gilt:*
falls `MG-proc` *mit der Ausgabe „abgelehnt“ terminiert, dann ist* $\langle AX, \Sigma \rangle$ *unerfüllbar.*

Beweis: Da `MG-proc` mit der Ausgabe „abgelehnt“ terminiert, existiert ein terminierender Lauf $B_1, \ldots, B_e$ mit $B_e = \lceil\ \rceil$, d.h. $B_e = \lceil\ \rceil$ ist R-erreichbar. Wir nehmen an, AX sei erfüllbar, d.h. es existiert eine $\mathcal{F}$-Interpretation $\mathcal{I}$ mit $\mathcal{I} \models AX$. Laut Satz 4.4.9 gilt für jeden R-erreichbaren Baum B_i, also auch für $\lceil\ \rceil$, daß ein Ast $A \in \textit{äste}(B_i)$ existiert, dessen Deutung mittels $\mathcal{I}$, $Deut_{\mathcal{I}}(A)$, zutrifft. Mit $\textit{äste}(\lceil\ \rceil) = \emptyset$ folgt der Widerspruch. □

4.5 Fairneß und Modell-Vollständigkeit

4.5.1 Fairneß

Um im Falle der Erfüllbarkeit der untersuchten Spezifikation die Terminierung des Verfahrens zu garantieren, muß eine zusätzliche Forderung an die indeterministische Prozedur `MG-proc` gestellt werden. Der Indeterminismus muß in der Weise eingeschränkt werden, daß die Prozedur keinen konstruierten Ast für alle Ewigkeiten übergeht. Dieses Verhalten bezeichnet man ‘Ast-Fairneß’, und dies ist auch in anderen Beweisverfahren eine typische Voraussetzung für Vollständigkeitsaussagen.

Das für die Realisierung des vorgestellten Verfahrens eingesetzte Werkzeug namens ‘MGTP’ (s. Kapitel 5) implementiert *keine* ast-faire Modell-Konstruktion, sondern eine ‘von-links-nach-rechts’-Strategie. Von daher beweisen wir strenggenommen nicht die Vollständigkeit unserer Realisierung. Wir zeigen aber, daß der in der Arbeit vorgestellte transformative Kalkül seinen Beitrag zur Vollständigkeit leistet: sobald er von einer ast-fairen Verfeinerung von `MG-proc` ausgeführt wird, garantiert er die terminierende Konstruktion eines Modells (falls die untersuchte Spezifikation erfüllbar ist).

In der Praxis findet auch die Realisierung mit MGTP die gesuchten Modelle der (Gegen-)Spezifikationen (und damit den gesuchten Fehler). Dies hat zwei Gründe: Erstens ist die Ursache der Fehler häufig in den Randfällen zu finden, die bei einer ‘von-links-nach-rechts’-Strategie zuerst behandelt werden. Man beachte, daß wir aus diesem Grund die Extensionen der search-Regeln so geordnet haben, daß die Konstruktoren mit weniger Argumenten in der Konklusion weiter links stehen (s. Def. 3.2.1). Zweitens sind die Regeln so konzipiert, daß Äste, die kein Modell repräsentieren, möglichst früh verworfen werden. Dies wird dadurch erreicht, daß

die same- und different-Regeln schon aufgrund der führenden Konstruktoren Äste verwerfen können, noch bevor die Platzhalter in den Argumenten beseitigt sind.

Nun präzisieren wir, was 'Ast-Fairneß', kurz 'Fairneß' bedeutet.

Definition 4.5.1 (faire Modell-Generierungs-Prozedur)
Eine **faire MG-Prozedur** ist eine Verfeinerung `fair-MG-proc` von `MG-proc` in der Weise, daß die folgende Eigenschaft von Läufen von `fair-MG-proc` garantiert wird:
Falls $B_1, B_2, \ldots\ldots$ ein nicht-terminierender Lauf von `fair-MG-proc` ist, dann gilt für jedes $i \geq 1$: Ist $B_i = \lceil A_1 | \ldots | A_n \rceil$, dann gibt es für jedes j mit $1 \leq j \leq n$ ein $i' \geq i$, so daß A_j in der i'-ten Iteration ausgewählt wird. ⋆

4.5.2 Modell-Vollständigkeit

Das Ziel der nun folgenden Argumentation ist zunächst, zu zeigen, daß in nicht-terminierenden Läufen von `fair-MG-proc` jeder Ast jede beliebige Länge überschreitet. Dies läßt sich ausdrücken mit Hilfe der 'minimalen Astlänge' von Bäumen, die jede beliebige Grenze überschreitet.

Definition 4.5.2 (minimale Astlänge, minimale Äste)
Sei B ein nichtleerer Baum ($B \neq \lceil\ \rceil$) mit $B = \lceil A_1 | \ldots | A_n \rceil$. Seine minimale Astlänge *minAstLänge*(B) ist das Minimum der Menge $\{|A_1|, \ldots, |A_n|\}$.
Die Menge der minimalen Äste von B, *minÄste*(B), ist definiert durch:
minÄste$(B) = \{A \in$ *äste*$(B) \mid |A| =$ *minAstLänge*$(B)\}$ ⋆

Wir sprechen von der 'Länge' der Äste, obwohl es sich bei den Ästen formal um Mengen handelt, und die sogen. 'Länge' eigentlich die Größe dieser Mengen bezeichnet.

Satz 4.5.3 *Sei $B_1, \ldots, B_e$ bzw. $B_1, B_2, \ldots\ldots$ ein Lauf von* `MG-proc` *(R) und sei $1 \leq i < e$ bzw. $i \geq 1$. Dann gilt:*
falls $B_{i+1} \neq \lceil\ \rceil$, dann gilt minAstLänge$(B_i) \leq$ minAstLänge(B_{i+1}).

Beweis: Entsteht B_{i+1} durch Streichung eines Astes, dann gibt es entweder einen minimalen Ast, der nicht gestrichen wird, und die minimale Astlänge bleibt unverändert, oder es wird ein Ast gestrichen, der echt kleiner als alle anderen ist, und die minimale Astlänge wird größer.
Entsteht B_{i+1} durch Erweiterung eines Astes, dann sind die neuen Äste $A_i \cup (Ext_1)\sigma, \ldots, A_i \cup (Ext_m)\sigma$ höchstens länger als A_i (sie sind sogar echt länger). □

Korollar 4.5.4 *Falls $B_1, B_2, \ldots$ ein nicht-terminierender Lauf von* `MG-proc` *(R) ist, dann gilt für $j \geq i$: minAstLänge$(B_j) \geq$ minAstLänge(B_i).*

Satz 4.5.5 *Sei* $B_1, \ldots, B_e$ *bzw.* $B_1, B_2, \ldots\ldots$ *ein Lauf von* **MG-proc** (R) *und sei* $1 \leq i < e$ *bzw.* $i \geq 1$*. Dann gilt:*
falls $B_{i+1} \neq \lceil\ \rceil$ *und* $minAstLänge(B_{i+1}) = minAstLänge(B_i)$*, dann folgt* $minÄste(B_{i+1}) \subseteq minÄste(B_i)$*.*

Beweis: Wird ein nichtminimaler Ast ausgewählt, dann wird dieser entweder verworfen oder erweitert. In beiden Fällen ist $minÄste(B_{i+1}) = minÄste(B_i)$. Wird aber ein minimaler Ast A augewählt, dann gilt (ob A gestrichen oder erweitert wird) $A \notin äste(B_{i+1})$. Da aber nach Voraussetzung $minAstLänge(B_{i+1}) = minAstLänge(B_i)$, war A nicht der einzige minimale Ast und es gilt:
$minÄste(B_{i+1}) = minÄste(B_i) \setminus A$. □

Korollar 4.5.6 *Falls* $B_1, B_2, \ldots$ *ein nicht-terminierender Lauf von* **MG-proc** (R) *ist, dann gilt für* $j \geq i$*: falls* $minAstLänge(B_j) = minAstLänge(B_i)$*, dann ist* $minÄste(B_j) \subseteq minÄste(B_i)$*.*

Jetzt können wir zeigen, daß die minimale Astlänge in einem nicht-terminierenden Lauf auf Dauer immer (echt) größer wird.

Satz 4.5.7 *Sei* **fair-MG-proc** *eine faire MG-Prozedur, sei* R *eine Menge von MG-Regeln und* $B_1, B_2, \ldots\ldots$ *ein nicht-terminierender Lauf von* **fair-MG-proc** *. Dann gibt es für jedes* $i \geq 1$ *ein* $j > i$ *mit:*
$minAstLänge(B_j) > minAstLänge(B_i)$*.*

Beweis: Induktion über die Anzahl minimaler Äste in B_i, $|minÄste(B_i)|$.

Ind.-Anf.: $|minÄste(B_i)| = 1$
Sei $minÄste(B_i) = \{A_h\}$. Wegen der Fairneß gibt es ein $i' \geq i$, so daß A_h in der i'-ten Iteration ausgewählt wird. Da $A_h \in äste(B_{i'})$, gilt $minAstLänge(B_{i'}) \leq minAstLänge(B_i)$.
Nach Korollar 4.5.4 gilt $minAstLänge(B_{i'}) \geq minAstLänge(B_i)$, zusammen also $minAstLänge(B_{i'}) = minAstLänge(B_i)$. Aus Korollar 4.5.6 folgt dann:
$minÄste(B_{i'}) \subseteq minÄste(B_i) = \{A_h\}$, d.h. $minÄste(B_{i'}) = \{A_h\}$. In der i'-ten Iteration wird nun A_h verworfen oder durch (längere) Äste ersetzt. Da A_h der einzige minimale Ast von $B_{i'}$ war, gilt in $B_{i'+1}$:
$minAstLänge(B_{i'+1}) > minAstLänge(B_{i'}) = minAstLänge(B_i)$.
Im Ind.-Anf. wählen wir dann $j = i' + 1$.

Ind.-Schritt: $|minÄste(B_i)| = m$, $m > 1$
Wegen der Fairneß wird jeder der Äste in $minÄste(B_i)$ in einer Iteration größergleich i ausgewählt. Sei $A_h \in äste(B_i)$ derjenige minimale Ast, der als erster in dem Teil-Lauf $B_i, B_{i+1}, \ldots$ ausgewählt wird. Sei i' derjenige Index ($i' \geq i$), so daß A_h in der i'-ten Iteration ausgewählt wird. Da $minÄste(B_i) \subseteq äste(B_{i'})$, gilt

$minAstLänge(B_{i'}) \leq minAstLänge(B_i)$.
Nach Korollar 4.5.4 gilt $minAstLänge(B_{i'}) \geq minAstLänge(B_i)$, zusammen also $minAstLänge(B_{i'}) = minAstLänge(B_i)$. Aus Korollar 4.5.6 folgt dann: $minÄste(B_{i'}) \subseteq minÄste(B_i)$, und wegen $minÄste(B_i) \subseteq äste(B_{i'})$ gilt $minÄste(B_{i'}) = minÄste(B_i) = m > 1$. In der i'-ten Iteration wird nun A_h verworfen oder durch (längere) Äste ersetzt. Daher ist $minÄste(B_{i'+1}) = minÄste(B_{i'}) \setminus A_h$ und $minAstLänge(B_{i'+1}) = minAstLänge(B_{i'})$. Da also $|minÄste(B_{i'+1})| = m - 1$, greift die Ind.-Hyp., nach der für $i' + 1$ ein j existiert mit $minAstLänge(B_j) > minAstLänge(B_{i'+1})$.
Wegen $minAstLänge(B_{i'+1}) = minAstLänge(B_{i'}) = minAstLänge(B_i)$, existiert auch für i ein (das selbe) j mit $minAstLänge(B_j) > minAstLänge(B_i)$. □

Da die minimalen Äste bei nicht-terminierenden Läufen also immer länger werden, ist intuitiv schon klar, daß die Äste jede beliebige Grenze m überschreiten. Wir formulieren dies noch als Satz, da wir diese Form der Aussage für den Vollständigkeitssatz verwenden werden.

Satz 4.5.8 *Sei* `fair-MG-proc` *eine faire MG-Prozedur, R eine Menge von MG-Regeln und* $B_1, B_2, \ldots$ *ein nicht-terminierender Lauf von* `fair-MG-proc`*. Dann existiert für jedes* $m \in \mathbb{N}$ *ein* $i \in \mathbb{N}$*, so daß* $minAstLänge(B_i) > m$*, also alle Äste von* B_i *länger als* m *sind.*

Beweis: Vollst. Induktion über m.

Ind.-Anf.: $m = 0$
Da $B_1 = \lceil \emptyset \rceil$, ist $minAstLänge(B_1) = 0$. Laut Satz 4.5.7 gibt es ein $i > 1$ mit $minAstLänge(B_i) > minAstLänge(B_1) = 0 = m$.

Ind.-Schritt: $m \to m + 1$
Laut Ind.-Hyp. ex. ein i mit $minAstLänge(B_i) > m$. Wir zeigen, daß auch ein i' ex. mit $minAstLänge(B_{i'}) > m + 1$. Fallunterscheidung:
a) Falls schon $minAstLänge(B_i) > m + 1$, dann setzen wir $i' = i$.
b) Falls $minAstLänge(B_i) = m + 1$, dann ex. nach Satz 4.5.7 ein $j > i$ mit $minAstLänge(B_j) > minAstLänge(B_i) = m + 1$. Wir setzen $i' = j$. □

Da nun gezeigt ist, daß ein nicht-terminierender Lauf der fairen Modell-Generierungs-Prozedur jede Schranke für die Länge von Ästen durchbricht, müssen wir 'nur noch' zeigen, daß sich aus der Erfüllbarkeit der untersuchten Spezifikation *doch eine Schranke für die Länge eines bestimmten Astes* ergibt. Dabei handelt es sich gerade um denjenigen Ast A, für den $Deut_{\mathcal{I}}(A)$ zutrifft. Da es laut Satz 4.4.9 im Falle der Erfüllbarkeit immer einen solchen Ast geben muß, würde die Annahme der Nicht-Terminierung zum Widerspruch führen. Daraus folgt die Terminierung im Falle der Erfüllbarkeit untersuchter Spezifikationen. Da wir die Terminierung mit der Ausgabe „abgelehnt“ ausschließen können (aufgrund der Unerfüllbarkeits-Korrektheit, Satz 4.4.10), muß die Ausgabe „A ist R-saturiert“

lauten, wobei A gerade die erfüllende Interpretation repräsentiert (aufgrund der Modell-Korrektheit, Satz 4.3.14). Insgesamt wird also für jede erfüllbare Spezifikation ein Modell gefunden! Dies genau ist die Modell-Vollständigkeit.

Der einzige nichttriviale Punkt bei der gerade skizzierten Argumentation ist der Nachweis der Längen-Schranke, der im Detail recht aufwendig ist. Um die Argumentationskette, die uns zur Modell-Vollständigkeit führt, nicht allzulange zu unterbrechen, wird der folgende Satz zur Beschränkung bestimmter Äste zunächst ohne Beweis wiedergegeben und für den anschließenden Vollständigkeitsbeweis benutzt. Der dann noch ausstehende Beweis wird in einem eigenen Unterabschnitt (4.5.3) nachgeholt.

Satz 4.5.9 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, Inst $\subseteq CT_\Sigma$ eine endliche Menge und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$. Weiterhin sei $\langle \Sigma, AX \rangle$ erfüllbar und $\mathcal{I}$ eine (feste) $\mathcal{F}$-Interpretation mit $\mathcal{I} \models AX$.*
Dann existiert ein $m \in \mathbb{N}$, so daß für jeden R-erreichbaren Ast A gilt: Falls $Deut_\mathcal{I}(A)$ zutrifft, dann ist $|A| \leq m$.

Beweis: Siehe Abschn. 4.5.3, S. 155. □

Satz 4.5.10 (Modell-Vollständigkeit)
Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, Inst $\subseteq CT_\Sigma$ eine endliche Menge und sei $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$. Weiterhin sei `fair-MG-proc` *eine faire MG-Prozedur. Dann gilt:*
Falls $\langle \Sigma, AX \rangle$ erfüllbar ist, dann terminiert jeder Lauf von `fair-MG-proc` *(R) mit der Ausgabe „A ist R-saturiert“, wobei für jede A entsprechende Interpretation $\mathcal{I}$ gilt: $\mathcal{I} \models [AX/Inst]$.*

Beweis: Wir nehmen die Erfüllbarkeit von $\langle \Sigma, AX \rangle$ an. Dann existiert eine Interpretation $\mathcal{I}'$ mit $\mathcal{I}' \models AX$. Ein solches $\mathcal{I}'$ halten wir jetzt fest. Zunächst zeigen wir durch Widerspruch, daß `fair-MG-proc`(R) immer terminiert. Nehmen wir also an, `fair-MG-proc`(R) terminiere nicht immer, d.h. es gibt einen nichtterminierenden Lauf $B_1, B_2, \ldots$ von `fair-MG-proc`(R). Laut Satz 4.4.9 gibt es in jedem Baum B_i $(i \geq 1)$ einen Ast A, so daß $Deut_{\mathcal{I}'}(A)$ (mit diesem $\mathcal{I}'$) zutrifft. Zusammen mit Satz 4.5.9 gibt es in jedem Baum B_i $(i \geq 1)$ einen Ast mit $|A| \leq m$, wobei m in Abhängigkeit von dem festen $\mathcal{I}'$ auch fest ist. Nach Satz 4.5.8 aber gilt auch für m, daß es ein $i \in \mathbb{N}$ gibt mit $minAstLänge(B_i) > m$. Widerspruch.
Jetzt wissen wir, daß jeder Lauf von `fair-MG-proc`(R) terminiert. Aus der Unerfüllbarkeits-Korrektheit (Satz 4.4.10) folgt, daß die Ausgabe nicht „abgelehnt“ sein kann. Also lautet die Ausgabe „A ist R-saturiert“, und aus der Modell-Korrektheit (Satz 4.3.14) folgt, daß für jede A entsprechende Interpretation $\mathcal{I}$ gilt: $\mathcal{I} \models [AX/Inst]$. □

4.5.3 Nachweis der beschränkten Ast-Länge

Nun arbeiten wir auf den Beweis von Satz 4.5.9 hin. Unter der Annahme der Erfüllbarkeit der untersuchten Spezifikationen soll gezeigt werden, daß *solche* Äste A, für die $Deut_{\mathcal{I}}(A)$ zutrifft, eine bestimmte Maximallänge nicht überschreiten können. Wir verfolgen in diesem Abschnitt nur das Ziel, *irgendeine* Grenze zu finden. Um die ohnehin recht aufwendige Argumentation nicht noch zusätzlich zu verkomplizieren, ist die Abschätzung *äußerst grob* und erhebt in dieser Form nicht den Anspruch einer Komplexitätsanalyse. Es geht ausschließlich darum, die schiere Existenz einer Obergrenze nachzuweisen, auch wenn diese um Größenordnungen oberhalb der Länge tatsächlich konstruierter Äste liegt.

Das zentrale Argument der folgenden Diskussion wird sein, daß, abhängig von der Spezifikation und einer vorgegebenen Interpretation $\mathcal{I}$, nur eine bestimmte, begrenzbare Menge von Meta-Termen in dem fraglichen Ast A auftreten kann. Erstens haben die Meta-Terme nur eine bestimmte Form, insbesondere sind sie wohlgeformt (vgl. Def. 3.4.4). Zweitens können innerhalb von $\mathsf{val}()$ nur Unterterme der instantiierten Axiome auftreten. Und drittens gibt es einen Zusammenhang zwischen der Größe der ausgewerteten Unterterme der instantiierten Axiome und der Schachtelungstiefe von $\mathsf{arg}n$-Termen. Insgesamt können also nur beschränkt viele Meta-Terme gebildet werden (in A). Dadurch ist auch die Menge der sie enthaltenden Meta-Atome begrenzt.

Nun diskutieren wir die mögliche Form von Meta-Termen. Eine wichtige Rolle hierbei spielt eine Unterstruktur wohlgeformter Meta-Terme, die sogenannten argval-Terme. (Für die Def. der maximalen Stelligkeit von Konstruktoren, $max\alpha(\Sigma)$, s. 3.4.1.)

Definition 4.5.11 (argval-Terme)
Ist ein wohlgeformter Meta-Term $t \in wMT_{\Sigma}$ von der Form
$t = \underbrace{\mathsf{arg}n_k(\ldots(\mathsf{arg}n_1}_{k \geq 0}(\mathsf{val}(t')))\ldots)$, wobei $\{n_1, \ldots, n_k\} \subseteq \{1, \ldots, max\alpha(\Sigma)\}$,
dann ist t ein **argval-Term** der Tiefe k und mit **Basis** t'.
$\mathsf{val}(t')$ ist ein argval-Term der Tiefe 0. Solche argval-Terme heißen insbesondere **val-Terme**. $\star$

Gemäß der Def. 3.4.4 ist die Basis eines argval-Terms immer ein Funktionsterm aus $T^0_{\Sigma} \setminus CT_{\Sigma}$, sie entält also Funktionen. Wir benötigen später eine Verbindung zwischen der Semantik von argval-Termen einerseits und und der Semantik ihrer Basis andererseits.

Satz 4.5.12 *Sei $t \in MT^0_{\Sigma}$ ein argval-Term mit Basis t', d.h.*

$$t = \mathsf{arg}n_k(\ldots(\mathsf{arg}n_1(\mathsf{val}(t')))\ldots)$$

und sei $\mathcal{I}$ eine $\mathcal{F}$-Interpretation. Falls $mtval_{\mathcal{I}}(t)$ wohldefiniert ist, dann gilt:

$$mtval_{\mathcal{I}}(t) \in unterTerme(val_{\mathcal{I}}(t'))$$

Beweis: Induktion über $k \geq 0$. Die Indizes verlaufen mit Absicht absteigend, weil die Induktion von innen nach außen argumentiert.
$k = 0$:
$t = \mathsf{val}(t')$. Dann ist $mtval_{\mathcal{I}}(t) = mtval_{\mathcal{I}}(\mathsf{val}(t')) = val_{\mathcal{I}}(t')$.
$k \to k+1$:
$t = \mathsf{arg}n_{k+1}(\mathsf{arg}n_k(\ldots(\mathsf{arg}n_1(\mathsf{val}(t')))\ldots))$.
Dann ist $mtval_{\mathcal{I}}(t) = (mtval_{\mathcal{I}}(\mathsf{arg}n_k(\ldots(\mathsf{arg}n_1(\mathsf{val}(t')))\ldots)))\downarrow_{n_{k+1}}$
Demnach ist $mtval_{\mathcal{I}}(t) \in unterTerme(mtval_{\mathcal{I}}(\mathsf{arg}n_k(\ldots(\mathsf{arg}n_1(\mathsf{val}(t')))\ldots)))$.
Nach der Ind.-Hyp. ist
$mtval_{\mathcal{I}}(\mathsf{arg}n_k(\ldots(\mathsf{arg}n_1(\mathsf{val}(t')))\ldots)) \in unterTerme(val_{\mathcal{I}}(t'))$.
Aus der Transitivität der Untertermbeziehung folgt die Behauptung. □

Für die folgende Diskussion benötigen wir leider noch vier Abbildungen auf Meta-Atomen bzw. wohlgeformten Meta-Termen. (Es sei nochmals auf die Definitionen wohlgeformter Meta-Terme wMT_{Σ} (3.4.4) verwiesen.) Wir notieren die Definitionen in verkürzter Form:

Definition 4.5.13

Top-level Terme von Meta-Atomen
$topTerme(s(t)) = topTerme(\mathsf{search_}s(t)) = \{t\}$,
$topTerme(\mathsf{is}(t_1, t_2)) = topTerme(\mathsf{same}(t_1, t_2))$
$\quad = topTerme(\mathsf{different}(t_1, t_2)) = \{t_1, t_2\}$,
$topTerme(\mathrm{I}(f, \langle\rangle, t)) = \{t\}$,
$topTerme(\mathrm{I}(f, \langle t_1, \ldots, t_n\rangle, t)) = \{t_1, \ldots, t_n, t\}$

Meta-Unter-Terme von Meta-Atomen und Meta-Termen
$metaUnterterme(at) = \bigcup_{t \in topTerme(at)} metaUnterterme(t)$
$metaUnterterme(c) = \{c\}$,
$metaUnterterme(c(t_1, \ldots, t_n))$
$\quad = \{c(t_1, \ldots, t_n)\} \cup metaUnterterme(t_1) \cup \ldots \cup metaUnterterme(t_n)$,
$metaUnterterme(\mathsf{arg}n(t)) = \{\mathsf{arg}n(t)\} \cup metaUnterterme(t)$,
$metaUnterterme(\mathsf{val}(t)) = \{\mathsf{val}(t)\}$

Anzahl unabhängiger argval-Terme in Meta-Termen
$anzUnabhArgvalTerme(c) = 0$,
$anzUnabhArgvalTerme(c(t_1, \ldots, t_n))$
$\quad = anzUnabhArgvalTerme(t_1) + \ldots + anzUnabhArgvalTerme(t_n)$,
$anzUnabhArgvalTerme(\mathsf{arg}n(t)) = 1$,
$anzUnabhArgvalTerme(\mathsf{val}(t)) = 1$

Anzahl äußerer Konstruktoren in Meta-Termen
$anzÄußKonstr(c) = 1$,
$anzÄußKonstr(c(t_1, \ldots, t_n))$
$\quad = 1 + anzÄußKonstr(t_1) + \ldots + anzÄußKonstr(t_n)$,

$anz\ddot{A}u\beta Konstr(\mathsf{arg}n(t)) = 0,$
$anz\ddot{A}u\beta Konstr(\mathsf{val}(t)) = 0$ $\star$

Entscheidend ist jeweils, wo die Rekursion endet. Die Abbildung *topTerme* ist auf Meta-Atomen definiert und folglich nicht rekursiv. Die Abbildung *metaUnterterme* 'schaut' in einen val-Term nicht hinein, denn dessen Argument ist kein Meta-Term, sondern ein Funktions-Term. Die Abbildung *anzUnabhArgvalTerme* 'schaut' nichteinmal in die argval-Terme hinein. *anzÄußKonstr* zählt nur die Konstruktoren *außerhalb* der argval-Terme.

Gemäß dieser Definition sind $anzUnabhArgvalTerme(t)$ und $anz\ddot{A}u\beta Konstr(t)$ beide kleiner als derjenige Konstruktorterm, zu dem t mittels $mtval_{\mathcal{I}}$ ausgewertet wird. Diese Tatsache formulieren wir als Satz, da sie später für die angestrebte Größenabschätzung gebraucht wird. (Vergl. Def. 2.2.14 von $|ct|$)

Satz 4.5.14 *Sei $t \in wMT_\Sigma$ ein wohlgeformter Meta-Term und $\mathcal{I}$ eine $\mathcal{F}$-Interpretation. Dann gilt:*

1. $anz\ddot{A}u\beta Konstr(t) \leq |mtval_{\mathcal{I}}(t)|$
2. $anzUnabhArgvalTerme(t) \leq |mtval_{\mathcal{I}}(t)|$

Beweis: Wir zeigen allgemeiner
$anz\ddot{A}u\beta Konstr(t) + anzUnabhArgvalTerme(t) \leq |mtval_{\mathcal{I}}(t)|$, durch strukturelle Induktion über t. (Man beachte $|ct| \geq 1$ (*) f.a. $ct \in CT_\Sigma$.)

- $t = \mathsf{val}(t')$, dann ist:
  $anz\ddot{A}u\beta Konstr(t) + anzUnabhArgvalTerme(t) = 0 + 1 \overset{*}{\leq} |mtval_{\mathcal{I}}(t)|$
- $t = \mathsf{arg}m(t'))$, dann ist:
  $anz\ddot{A}u\beta Konstr(t) + anzUnabhArgvalTerme(t) = 0 + 1 \overset{*}{\leq} |mtval_{\mathcal{I}}(t)|$
- $t = c$, dann ist:
  $anz\ddot{A}u\beta Konstr(t) + anzUnabhArgvalTerme(t) = 1 + 0 \overset{*}{\leq} |mtval_{\mathcal{I}}(t)|$
- $t = c(t_1, \ldots, t_n))$, dann ist:
  $anz\ddot{A}u\beta Konstr(t) + anzUnabhArgvalTerme(t)$
  $= 1 + anz\ddot{A}u\beta Konstr(t_1) + \ldots + anz\ddot{A}u\beta Konstr(t_n)$
  $\quad + anzUnabhArgvalTerme(t_1) + \ldots + anzUnabhArgvalTerme(t_n)$
  $\overset{IndHyp}{\leq} 1 + |mtval_{\mathcal{I}}(t_1)| + \ldots + |mtval_{\mathcal{I}}(t_n)|$
  $= |c(mtval_{\mathcal{I}}(t_1), \ldots, mtval_{\mathcal{I}}(t_n))| = |mtval_{\mathcal{I}}(c(t_1, \ldots, t_n))| = |mtval_{\mathcal{I}}(t)|$

$\square$

Als Grundlage für die Diskussion von Eigenschaften der Meta-Terme bzw. -Atome in *Bäumen* halten wir zunächst fest, welcher Art die Meta-Terme bzw. -Atome sind, die bei der Transformation eines Termes entstehen. Wie schon in den vergangenen Abschnitten ermöglicht es die Substitutivität von $\mathfrak{TransTerm}_\Sigma$, lediglich die Transformation von Grundtermen zu betrachten. Wir schicken noch ein Hilfslemma über $\mathfrak{Rep}(t)$ voraus.

Lemma 4.5.15 *Sei $t \in T^0_\Sigma$ und $t' \in metaUnterterme(\mathfrak{Rep}(t))$. Dann ist t' ein wohlgeformter Meta-Term, also $t' \in wMT_\Sigma$.*

Beweis: Fallunterscheidung:
a) $t \in CT_\Sigma$.
Dann ist $\mathfrak{Rep}(t) = t$, und $metaUnterterme(\mathfrak{Rep}(t)) = unterTerme(t) \subseteq CT_\Sigma$. Lt. Korollar 3.4.5 ist $CT_\Sigma \subseteq wMT_\Sigma$, daher $metaUnterterme(\mathfrak{Rep}(t)) \subseteq wMT_\Sigma$.
b) $t \notin CT_\Sigma$.
Dann ist $\mathfrak{Rep}(t) = \mathsf{val}(t)$. Wegen $metaUnterterme(\mathsf{val}(t)) = \{\mathsf{val}(t)\}$ gilt die Beh.
□

Satz 4.5.16 *Sei $t \in T^0_\Sigma$ und sei $at \in \mathfrak{TransTerm}_\Sigma(t)$. Dann gilt:*

1. a) *Falls $\mathsf{val}(t') \in metaUnterterme(at)$, dann gilt $t' \in unterTerme(t)$.*
   b) *Falls $ct \in metaUnterterme(at)$ (für ein $ct \in CT_\Sigma$), dann gilt $ct \in unterTerme(t)$.*
2. *Falls $t' \in metaUnterterme(at)$, dann ist t' wohlgeformt: $t' \in wMT_\Sigma$.*
3. *Falls $at = \mathsf{search_}s(t_1)$, dann ist $t_1 = \mathsf{val}(t')$ für ein $t' \in T^0_\Sigma \setminus CT_\Sigma$.*
4. *Falls $at = \mathsf{is}(t_1, t_2)$, dann ist $t_1 = \mathsf{val}(t')$ für ein $t' \in T^0_\Sigma \setminus CT_\Sigma$.*

Beweis: Strukturelle Induktion über $t \in T^0_\Sigma$.

- Falls $t \in CT_\Sigma$, dann ist $\mathfrak{TransTerm}_\Sigma(t) = \emptyset$.
- Falls $t = a$ mit $a \in F_s$, $\alpha(a) = \lambda$, dann ist
  $\mathfrak{TransTerm}_\Sigma(t) = \{\mathrm{I}(a, \langle\rangle, \mathsf{val}(a)),\ \mathsf{search_}s(\mathsf{val}(a))\}$.
  Wir überprüfen die Bedingungen 1. – 4.
  1. a) $t' = a \in unterTerme(a)$. b) triv.
  2. $t' = \mathsf{val}(a) \in wMT_\Sigma$
  3. insbes. ist $a \in T^0_\Sigma \setminus CT_\Sigma$
  4. gilt trivialerweise

- Falls $t = c(t_1, \ldots, t_n)$ mit $c \in C_s$, $\alpha(c) = s_1 \ldots s_n$, *wobei* $t \notin CT_\Sigma$, dann ist
$\mathfrak{TransTerm}_\Sigma(t) = \{\mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)))\}$
$\cup\, \mathfrak{TransTerm}_\Sigma(t_1) \cup \ldots \cup \mathfrak{TransTerm}_\Sigma(t_n)$
Falls $at \in \mathfrak{TransTerm}_\Sigma(t_i)$, dann folgen die Bed. 1. – 4. aus der Ind.-Hyp. (im Fall von Bed. 1. unter Beachtung von $unterTerme(t_i) \subseteq unterTerme(t)$).
Sei also $at = \mathsf{is}(\mathsf{val}(t), c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)))$.
Wir überprüfen die Bedingungen 1. – 4.
1. a) Sei $\mathsf{val}(t') \in metaUnterterme(at)$, dann ist entweder
$t' = t \in unterTerme(t)$, oder $\mathsf{val}(t') \in metaUnterterme(\mathfrak{Rep}(t_i))$, für ein t_i.
Im letzten Fall gilt wg. Def. von $\mathfrak{Rep}$, daß $t' = t_i \in unterTerme(t)$.
1. b) Sei $ct \in metaUnterterme(at)$. Dann ist $ct \in metaUnterterme(\mathfrak{Rep}(t_i))$, für ein $t_i \in CT_\Sigma$. Dann ist $ct \in unterTerme(t_i) \subseteq unterTerme(t)$.
2. $\mathsf{val}(t)$ ist wohlgeformt wg. $t \notin CT_\Sigma$, und wegen Lemma 4.5.15 sind f.a. t_i die Meta-Unterterme von $\mathfrak{Rep}(t_i)$ wohlgeformt, darum ist auch
$c(\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n))$ wohlgeformt.
3. und 4. gelten trivialerweise.

- Falls $t = f(t_1, \ldots, t_n)$ mit $f \in F_s$, $\alpha(f) = s_1 \ldots s_n$, dann ist
$\mathfrak{TransTerm}_\Sigma(t) = \{\mathrm{I}(f, \langle\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, \mathsf{val}(t)),\ \mathsf{search_}s(\mathsf{val}(t))\}$
$\cup\, \mathfrak{TransTerm}_\Sigma(t_1) \cup \ldots \cup \mathfrak{TransTerm}_\Sigma(t_n)$
Falls $at \in \mathfrak{TransTerm}_\Sigma(t_i)$, dann folgen 1. – 4. aus der Ind.-Hyp. (s.o.).
Falls $at = \mathsf{search_}s(\mathsf{val}(t))$, dann folgen mit $t \notin CT_\Sigma$ sofort die Bedingungen 1. – 3., Bedingung 4. ist trivial erfüllt.
Sei also im folgenden $at = \mathrm{I}(f, \langle\mathfrak{Rep}(t_1), \ldots, \mathfrak{Rep}(t_n)\rangle, \mathsf{val}(t))$. Wir überprüfen die Bedingungen 1. – 4.
1. a) Sei $\mathsf{val}(t') \in metaUnterterme(at)$, dann ist entweder
$t' = t \in unterTerme(t)$, oder $\mathsf{val}(t') \in metaUnterterme(\mathfrak{Rep}(t_i))$, für ein t_i.
Im letzten Fall gilt wg. Def. von $\mathfrak{Rep}$, daß $t' = t_i \in unterTerme(t)$.
1. b) Sei $ct \in metaUnterterme(at)$. Dann ist $ct \in metaUnterterme(\mathfrak{Rep}(t_i))$, für ein $t_i \in CT_\Sigma$. Dann ist $ct \in unterTerme(t_i) \subseteq unterTerme(t)$.
2. $\mathsf{val}(t)$ ist wohlgeformt wg. $t \notin CT_\Sigma$, und wegen Lemma 4.5.15 sind f.a. t_i die Meta-Unterterme von $\mathfrak{Rep}(t_i)$ wohlgeformt.
3. und 4. gelten trivialerweise.

□

Diese Eigenschaften der Transformation werden ausgenutzt im Beweis des folgenden Satzes, der verschiedene Eigenschaften von Meta-Termen und Meta-Atomen in R-erreichbaren Bäumen zusammenträgt. Diese Aussagen können nicht alle auf verschiedene Sätze verteilt werden, weil sie induktiv über Iterationen von `MG-proc`(R) zu beweisen sind und wechselseitig voneinander abhängen.

Satz 4.5.17 *Sei* $\langle \Sigma, AX \rangle$ *eine normalisierte ADT-Spezifikation,* $Inst \subseteq CT_\Sigma$ *und* $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$.
Ist B *ein* R*-erreichbarer Baum, dann gilt für alle* $at \in atome(B)$*:*

1. *Falls* $\mathsf{val}(t) \in metaUnterterme(at)$*, dann gilt* $t \in unterTerme([AX/Inst])$.
2. *Falls* $t \in metaUnterterme(at)$*, dann ist* t *wohlgeformt:* $t \in wMT_\Sigma$.
3. *Falls* $at = \mathsf{search_}s(t)$*, dann ist* t *ein argval-Term.*
4. *Falls* $at = \mathsf{is}(t_1, t_2)$*, dann ist* t_1 *ein argval-Term.*
5. *Falls* $at = s(t)$*, dann ist* $t \in CT_s \cap Inst$.

Beweis: Induktion über die Iterationen von ***MG-proc***(R).

Ind.-Anf.: $B_1 = \lceil \emptyset \rceil$
Wegen $atome(B_1) = \emptyset$ gelten die Behauptungen trivialerweise.

Ind.-Schritt: $B_i \;\rightarrow\; B_{i+1}$
Sei A der in der i-ten Iteration ausgewählte Ast.
Falls $B_{i+1} = Verw(A, B, r, \sigma)$, dann ist $atome(B_{i+1}) \subseteq atome(B_i)$. und die Beh. folgt aus der Ind.-Hyp.
Sei also $B_{i+1} = Erw(A, B, r, \sigma)$, mit $r = Pr \rightarrow Ext_1; \ldots; Ext_k$. . Es gilt:
$(\mathrm{Pr})\sigma \subseteq atome(B_i)$ und $atome(B_{i+1}) = atome(B_i) \cup \{(Ext_1)\sigma\} \cup \ldots \cup \{(Ext_k)\sigma\}$. Aufgrund der Ind.-Hyp. müssen wir die Eigenschaften 1. – 5. nur für $at \in \{(Ext_1)\sigma\} \cup \ldots \cup \{(Ext_k)\sigma\}$ zeigen, und dürfen dabei insbesondere voraussetzen, daß 1. – 5. bereits für $(\mathrm{Pr})\sigma$ gilt. Wir treffen eine große Fallunterscheidung über die verschiedenen Regelmengen, die sich ganz natürlich aus der Def. von $\mathfrak{TransSpec}$ und $\mathfrak{TransInst}_\Sigma$ ergeben. Im folgenden ist immer $ax \in \{(Ext_1)\sigma\} \cup \ldots \cup \{(Ext_k)\sigma\}$. Je nach der Form von at müssen wir nur die passenden der Bedingungen 3. – 5. zeigen. Die Bed. 1. u. 2. müssen wir aber in jedem Fall nachweisen.

Fall 1.: $r \in \mathfrak{TransInst}_\Sigma(Inst)$
$r = \rightarrow\; s(ct)$. für ein $ct \in Inst \cap CT_s$. Dann ist $at = s(t)$ mit $t \in CT_s \cap Inst$ (Bed. 5.) und die $unterTerme(s(ct))$ sind wohlgeformt (Bed. 2.).

Fall 2.: $r \in \mathfrak{TransAxioms}_\Sigma(AX)$
$r \in \mathfrak{TransAxiom}_\Sigma(ax)$ für ein $ax \in AX$. Sei $ax = lit_1 \vee \ldots \vee lit_n$, $Var(ax) = \{x_1, \ldots x_m\}$. Da $(\mathrm{Pr})\sigma = \{s_1(x_1), \ldots, s_m(x_m)\} \subseteq atome(B_i)$, gilt wg. der Ind.-Hyp., daß $\sigma(x_j) \in CT_{s_j} \cap Inst$, d.h. insbesondere $sort(\sigma(x_j)) = s_j = sort(x_j)$. Darum ist die Einschränkung σ' von σ auf $Var(ax)$, $\sigma' = \sigma|_{Var(ax)}$, eine gewöhnliche, wohlsortierte Substitution zu V_Σ. Da $\sigma'(x_j) = CT_{s_j} \cap Inst$ für jedes $x_j \in Var(ax)$, ist $ax\sigma' \in [ax/Inst] \subseteq [AX/Inst]$ (*). Wir haben $at \in (\mathfrak{TransLit}_\Sigma(lit))\sigma$ für ein $lit \in \{lit_1, \ldots, lit_n\}$. Da $Var(\mathfrak{TransLit}_\Sigma(lit)) = Var(lit) \subseteq Var(ax)$ und wegen $\sigma' = \sigma|_{Var(ax)}$, gilt $(\mathfrak{TransLit}_\Sigma(lit))\sigma = (\mathfrak{TransLit}_\Sigma(lit))\sigma'$. O.B.d.A. sei

$lit = t_1 \doteq t_2$ (der Fall $lit_j = t_1 \not\doteq t_2$ geht völlig analog). Da σ' eine vollständige Konstruktorterm-Substitution für t_1 und t_2 ist, gilt lt. Satz 4.1.6 ($\mathfrak{TransLit}_\Sigma(t_1 \doteq t_2))\sigma' = \mathfrak{TransLit}_\Sigma((t_1\sigma' \doteq t_2\sigma'))$. Für $at \in \mathfrak{TransLit}_\Sigma((t_1\sigma' \doteq t_2\sigma'))$ ergeben sich zwei Fälle.
a) $at = \mathsf{same}(\mathfrak{Rep}(t_1\sigma'), \mathfrak{Rep}(t_2\sigma'))$. Zunächst gelten für alle
$t \in metaUnterterme(at)$, daß $t \in metaUnterterme(\mathfrak{Rep}(t_1\sigma'))$ oder
$t \in metaUnterterme(\mathfrak{Rep}(t_2\sigma'))$. O.B.d.A. sei $t \in metaUnterterme(\mathfrak{Rep}(t_1\sigma'))$. Da $t_1\sigma' \in T^0_\Sigma$, folgt mit Lemma 4.5.15, daß t wohlgeformt ist, d.h. die Bed. 2. ist erfüllt. Um auch die 1. Bedingung zu zeigen, nehmen wir an, es sei $t = \mathsf{val}(t')$ für ein passendes t'. Da $t_1\sigma' \in T^0_\Sigma$, ist der einzige val-Term, den $\mathfrak{Rep}(t_1\sigma')$ enthalten kann, $\mathsf{val}(t_1\sigma')$. Da $t_1\sigma' \subseteq unterTerme(ax\sigma')$ und wegen $ax\sigma' \in [AX/Inst]$ (*) ist dann $t' = t_1\sigma' \in unterTerme([AX/Inst])$.
b) $at \in \mathfrak{TransTerm}_\Sigma(t_1\sigma')$ oder $at \in \mathfrak{TransTerm}_\Sigma(t_2\sigma')$
O.B.d.A. sei $at \in \mathfrak{TransTerm}_\Sigma(t_1\sigma')$. Jetzt wenden wir den Satz 4.5.16 an. Aus dessen Bedingungen 2. – 4. folgen unmittelbar die Bed. des hier zu beweisenden Astes. Aus der dortigen Bed. 1.a) folgt: Wenn $\mathsf{val}(t') \in metaUnterterme(at)$, dann gilt $t' \in unterTerme(t_1\sigma')$. Da $t_1\sigma' \in unterTerme([AX/Inst])$ (s.o.) folgt $t' \in unterTerme([AX/Inst])$ (Bed. 1.). Für kein ct gilt, daß $s(ct) \in \mathfrak{TransTerm}_\Sigma(t_1\sigma')$, weswegen die Bed. 5) trivial gilt.

Fall 3.: $r \in \mathfrak{Funktionalitaet}$
Hier gibt es zwei Regeln. In beiden Fällen ist $at = \mathsf{same}(t, t')$ für zwei Meta-Terme $\{t, t'\} \subseteq MT^0_\Sigma$. Man sieht leicht, daß sowohl t als auch t' Meta-Unter-Terme von Atomen in B_i sind. Damit erledigen sich die Bedingungen 1. und 2. Die anderen folgen trivial.

Fall 4.: $r \in \mathfrak{Ersetzung}_\Sigma$
Wir betrachten nur eine Regel exemplarisch:
$r = \mathsf{is}(x, c(y_1, \ldots, y_n)),\ \mathsf{is}(y_1, z) \ \rightarrow\ \mathsf{is}(x, c(z, \ldots, y_n))$. Das in B_{i+1} einzig hinzugekommene Atom ist $\mathsf{is}(\sigma(x), c(\sigma(z), \ldots, \sigma(y_n)))$. Der einzig neue Meta-Term ist dann $c(\sigma(z), \ldots, \sigma(y_n))$. Dessen Wohlgeformtheit folgt leicht aus der Wohlgeformtheit der Argumente. Damit folgt Bed. 2.. Da der einzig neue Meta-Unter-Term kein val-Term ist, folgt die Bed 1. aus der Ind.-Hyp. Weiterhin ist nur noch die Bed. 4. anwendbar. Hier müssen wir zeigen, daß $\sigma(x)$ ein argval-Term ist. Dies folgt aber aus der Ind.-Hyp., da $\mathsf{is}(\sigma(x), c(\sigma(y_1), \ldots, \sigma(y_n))) \in atome(B_i)$.

Fall 5.: $r \in \mathfrak{FreiErz}_\Sigma$
Da hier keine neuen Meta-Terme hinzukommen, ist hier der Nachweis sämtlicher Bedingungen trivial.

Fall 6.: $r \in \mathfrak{TransSorts}_\Sigma(S)$
$r \in \mathfrak{TransSort}_\Sigma(s)$ für ein $s \in S$. Wir wissen: $\mathsf{search_}s(\sigma(x)) \in atome(B_i)$. Sei $at \in \mathfrak{TransKonstr}_\Sigma(\sigma(x), c_j)$, für $c_j \in \{c_1, \ldots, c_n\}$. Fallunterscheidung:
a1) $\alpha(c_j) = s_1 \ldots s_m$ und $at = \mathsf{is}(\sigma(x), c_j(\mathsf{arg1}(\sigma(x)), \ldots, \mathsf{arg}m(\sigma(x))))$.
Die einzig 'neuen' Meta-Unter-Terme sind: $\mathsf{arg1}(\sigma(x)), \ldots, \mathsf{arg}m(\sigma(x))$ und

$c_j(\mathsf{arg1}(\sigma(x)), \ldots, \mathsf{arg}m(\sigma(x)))$. Damit können wir die Bed. 1 aus der Ind.-Hyp. übernehmen. Nach der Def. der Wohlgeformtheit und der Ind.-Hyp. sind die neuen Meta-Terme auch wohlgeformt, es gilt also Bed. 2.. Die Bed. 4., für die wir zeigen müssen, daß $\sigma(x)$ ein argval-Term ist, ergibt sich diesmal aus der Bedingung 3.(!) der Ind.-Hyp. Die anderen Bed. sind trivial.
a2) $\alpha(c_j) = s_1 \ldots s_m$ und $st = \mathsf{search_}s_u(\mathsf{arg}u(\sigma(x)))$.
Daß $\mathsf{arg}u(\sigma(x))$ wohlgeformt ist, haben wir schon in a) gezeigt (Bed. 2.). Auch die Bed. 1. ergibt sich wie in a). Da mit $\sigma(x)$ auch $\mathsf{arg}u(\sigma(x))$ ein argval-Term ist, ergibt sich die die Bed. 3. aus der Bed. 3. der Ind.-Hyp., 4. und 5. sind nicht anwendbar.
b) $\alpha(c_j) = \lambda$ und $at = \mathsf{is}(\sigma(x), c_j)$.
Hier ist c_j der einzige neue Meta-Term, und selbiger ist sicherlich wohlgeformt (Bed. 2.). Die Bed. 4. ergibt sich wieder aus der Bedingung 3. der Ind.-Hyp. Die anderen Bed. sind trivial.

□

Nachdem wir jetzt einige Eigenschaften von Meta-Atomen (und darin enthaltenen Meta-Termen) nachgewiesen haben, die für *jedes* beliebige Atom eines R-erreichbaren Baumes gelten, geht es im folgenden um solche Eigenschaften, die *gerade auf den* Ästen A gelten, deren Deutung $Deut_{\mathcal{I}}(A)$ mit einer gegebenen Interpretation $\mathcal{I}$ zutrifft. Wir stellen einen Zusammenhang her zwischen den 'Top-Termen' der Meta-Atome in diesem Ast und den Untertermen der instantiierten Axiome. Dieser Zusammenhang beruht auf der beidseitigen Auswertung der Terme mittels $\mathcal{I}$, auf der einen Seite mit $mtval_{\mathcal{I}}$ und auf der anderen Seite mit $val_{\mathcal{I}}$. Es handelt sich nicht ganz um eine 'eins-zu-eins'-Entsprechung. Auf der Seite der instantiierten Axiome muß gleich zweimal die Bildung von Unter-Termen berücksichtigt werden, und zwar einmal vor und einmal nach der Auswertung mit $val_{\mathcal{I}}$. Der folgende Satz besagt, in natürlicher Sprache ausgedrückt:

> *Die Auswertung eines jeden 'Top-Termes' in A ist gleich einem Unterterm der Auswertung eines Unterterms der instantiierten Axiome.*

Einfacher ist es leider nicht. Aber dieser Sachverhalt ermöglicht es uns (weiter unten), von den ausgewerteten Untertermen der instantiierten Axiome auf eine Beschränkung der Anzahl möglicher Top-Terme in A zu schließen.

Satz 4.5.18 *Sei $\langle \Sigma, AX \rangle$ eine normalisierte ADT-Spezifikation, $Inst \subseteq CT_\Sigma$ und $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_\Sigma(Inst)$.
Sei weiterhin B ein R-erreichbarer Baum, sei $A \in äste(B)$ und $\mathcal{I}$ eine $\mathcal{F}$-Interpretation, wobei $Deut_{\mathcal{I}}(A)$ zutrifft. Ist nun $at \in A$ ein Meta-Atom in A, welches kein Sorten-Atom ist ($at \neq s(ct)$), und ist $t \in topTerme(at)$, dann existiert ein $t' \in unterTerme([AX/Inst])$, so daß $mtval_{\mathcal{I}}(t) \in unterTerme(val_{\mathcal{I}}(t'))$.*

Beweis: Induktion über die Iterationen von $\mathtt{MG\text{-}proc}(R)$.

Ind.-Anf.: $B_1 = \lceil\emptyset\rceil$
In diesem Fall gilt die Behauptung trivialerweise.

Ind.-Schritt: $B_i \;\rightarrow\; B_{i+1}$
Sei $A \in \textit{äste}(B_{i+1})$. Falls bereits $A \in \textit{äste}(B_i)$, dann folgt die Beh. aus der Ind.-Hyp. Sei also $A \in \textit{äste}(B_{i+1}) \setminus \textit{äste}(B_i)$. Dann muß $B_{i+1} = \mathit{Erw}(A', B_i, r, \sigma)$ sein für ein $A' \in \textit{äste}(B_i)$, mit $r = Pr \rightarrow \mathit{Ext}_1; \ldots; \mathit{Ext}_n.$ und $A = A' \cup \{(\mathit{Ext}_j)\sigma\}$ für ein $j \in \{1, \ldots, n\}$. Sei also $at \in A$. Falls $at \in A'$, dann folgt die Beh. aus der Ind.-Hyp. Sei also $at \in \{(\mathit{Ext}_j)\sigma\}$. Es sei noch einmal daran erinnert, daß wir davon ausgehen, daß $\mathit{Deut}_{\mathcal{I}}(A)$ zutrifft. Wegen $at \in A$ trifft dann auch $\mathit{Deut}_{\mathcal{I}}(at)$ zu. Diesmal treffen wir die Fallunterscheidung über die Form von at. Aufgrund der schon getroffenen Vorbereitungen in vorangegangenen Sätzen müssen wir hier nicht mehr in jedem Fall über die Regelanwendung der i-ten Iteration argumentieren.

- Sei $at = \mathsf{search_}s(t)$.
  Laut Satz 4.5.17, Bed. 3., ist t ein argval-Term. Hier können wir t' wählen als die Basis des argval-Termes t, denn laut Satz 4.5.17, Bed. 1., ist dann $t' \in \mathit{unterTerme}([AX/\mathit{Inst}])$ und nach Satz 4.5.12 gilt
  $\mathit{mtval}_{\mathcal{I}}(t) \in \mathit{unterTerme}(\mathit{val}_{\mathcal{I}}(t'))$.

- Sei $at = \mathsf{is}(t_1, t_2)$.
  a) Sei zunächst $t = t_1$. Laut Satz 4.5.17, Bed. 4., ist t_1 ein argval-Term, und nach der gleichen Argumentation wie im letzten Fall existiert ein t' (wieder die Basis des argval-Terms t_1), für welches gilt:
  $t' \in \mathit{unterTerme}([AX/\mathit{Inst}])$ und $\mathit{mtval}_{\mathcal{I}}(t) \in \mathit{unterTerme}(\mathit{val}_{\mathcal{I}}(t'))$.
  b) Sei nun $t = t_2$.
  Da $\mathit{Deut}_{\mathcal{I}}(\mathsf{is}(t_1, t_2))$ zutrifft, gilt $[\mathit{mtval}_{\mathcal{I}}(t_1) = \mathit{mtval}_{\mathcal{I}}(t_2)]$. Darum können wir für t_2 *das gleiche* t' wählen wie für t_1, nämlich die Basis des argval-Terms t_1! Wieder gilt:
  $t' \in \mathit{unterTerme}([AX/\mathit{Inst}])$ und $\mathit{mtval}_{\mathcal{I}}(t) \in \mathit{unterTerme}(\mathit{val}_{\mathcal{I}}(t'))$.
  Der Leser/die Leserin möge bitte die sich an den Beweis anschließende Bemerkung zu genau diesem Fall beachten.

- Sei $at = \mathsf{same}(t_1, t_2)$.
  O.B.d.A. ist $t = t_1$. Wir treffen eine Fallunterscheidung über die Regel r, aus der at in der i-ten Iteration entstanden sein kann.
  a) $r = \mathfrak{TransAxiom}_{\Sigma}(ax)$ für ein $ax \in AX$, mit $ax = \mathit{lit}_1 \vee \ldots \vee \mathit{lit}_n$. Da $at \in \{(\mathit{Ext}_j)\sigma\}$ ist, muß lit_j von der Form $t'_1 \doteq t'_2$ sein, und $t_1 = (\mathfrak{Rep}(t'_1))\sigma = \mathfrak{Rep}(t'_1\sigma)$. Nun wählen wir $t' = t'_1\sigma$. Betrachten wir t' näher. Wenn wir zeigen können (s.u.), daß $t' \in \mathit{unterTerme}([AX/\mathit{Inst}])$ (*), dann ist t' die passende Wahl. Denn laut Satz 4.4.4 ist $\mathit{mtval}_{\mathcal{I}}(\mathfrak{Rep}(t')) = \mathit{val}_{\mathcal{I}}(t')$, d.h. sowieso $\mathit{mtval}_{\mathcal{I}}(t) = \mathit{mtval}_{\mathcal{I}}(\mathfrak{Rep}(t')) \in \mathit{unterTerme}(\mathit{val}_{\mathcal{I}}(t'))$, was zu

zeigen war. Bleibt noch (*) zu zeigen. Betrachten wir die Prämisse der Regel $\mathfrak{TransAxiom}_\Sigma(ax)$: $Pr = \{s_1(x_1), \ldots, s_m(x_m)\}$. Wir wissen $Pr\sigma = \{s_1(\sigma(x_1)), \ldots, s_m(\sigma(x_m))\} \subseteq A$. Nach Satz 4.5.17, Bed. 5., ist dann $\sigma(x_i) \in CT_{s_i} \cap Inst$. D.h. die Einschränkung σ' von σ auf $Var(ax)$, $\sigma' = \sigma|_{Var(ax)}$, ist eine gewöhnliche, wohlsortierte Substitution zu V_Σ. Darum und weil $t'_1 \in unterTerme(ax)$, ist $t'_1\sigma = t'_1\sigma' \in unterTerme(ax\sigma')$. Wg. $\sigma'(x_i) \in Inst$ ist $ax\sigma' \in [ax/Inst] \subseteq [AX/Inst]$. Insgesamt ist $t' = t'_1\sigma = t'_1\sigma' \in unterTerme([AX/Inst])$, w.z.b.w. (*).
b) $r \in \mathfrak{Funktionalitaet} \cup \mathfrak{ErsetzSame}$.
Hier kann man leicht sehen, daß in dem Baum B_{i+1} keine Top-Terme auftreten, die nicht schon in B_i auftreten. Somit folgt die Beh. aus der Ind.-Hyp.
c) $r \in \mathfrak{TestSame}_\Sigma$. Hier ist
$r = \mathsf{same}(c(x_1, \ldots, x_n), c(y_1, \ldots, y_n)) \to \mathsf{same}(x_1, y_1), \ldots, \mathsf{same}(x_n, y_n)$.
Dann ist $at \in \{\mathsf{same}(x_1, y_1), \ldots, \mathsf{same}(x_n, y_n)\}\sigma$. O.B.d.A. sei
$at = \mathsf{same}(\sigma(x_1), \sigma(y_1))$ und weiterhin o.B.d.A. sei $t = \sigma(x_1)$. Aus der Ind.-Hyp. wissen wir, daß zu $(c(x_1, \ldots, x_n))\sigma$ ein $t' \in unterTerme([AX/Inst])$ existiert mit $mtval_{\mathcal{I}}((c(x_1, \ldots, x_n))\sigma) \in unterTerme(val_{\mathcal{I}}(t'))$ (*). Da
$mtval_{\mathcal{I}}((c(x_1, \ldots, x_n))\sigma) = mtval_{\mathcal{I}}(c(\sigma(x_1), \ldots, \sigma(x_n)))$
$= c(mtval_{\mathcal{I}}(\sigma(x_1)), \ldots, mtval_{\mathcal{I}}(\sigma(x_n)))$, gilt auch
$mtval_{\mathcal{I}}(\sigma(x_1)) \in unterTerme(mtval_{\mathcal{I}}((c(x_1, \ldots, x_n))\sigma))$.
Wegen (*) gilt dann auch:
$mtval_{\mathcal{I}}(t) = mtval_{\mathcal{I}}(\sigma(x_1)) \in unterTerme(val_{\mathcal{I}}(t'))$.

- Sei $at = \mathsf{different}(t_1, t_2)$.
  Hier kommt $r \in \mathfrak{TransAxioms}_\Sigma \cup \mathfrak{TestDiff}_\Sigma$ in Frage. Diese Fälle verlaufen völlig analog zu a) und c) bei $at = \mathsf{same}(t_1, t_2)$.

- Sei $at = \mathrm{I}(f, \langle t_1, \ldots, t_m \rangle, t_0)$.
  Hier kommen als Regeln, aus denen at in der i-ten Iteration entstanden sein kann, $\mathfrak{TransAxioms}_\Sigma(AX) \cup \mathfrak{ErsetzI}_\Sigma$ in Frage. Für $r \in \mathfrak{ErsetzI}_\Sigma$ gilt das gleiche wie bei b) im same-Fall. Darum betrachten wir hier nur $r \in \mathfrak{TransAxioms}_\Sigma(AX)$.
  Sei also $r = \mathfrak{TransAxiom}_\Sigma(ax)$ für ein $ax \in AX$, mit $ax = lit_1 \vee \ldots \vee lit_n$. Da $at \in \{(Ext_j)\sigma\}$ ist, ist $at \in (\mathfrak{TransLit}_\Sigma(lit_j))\sigma$. Sei $lit_j = t'_1 \doteq t'_2$ oder $lit_j = t'_1 \not\doteq t'_2$. In beiden Fällen ist $at \in \mathfrak{TransTerm}_\Sigma(t'_1)$ oder $at \in \mathfrak{TransTerm}_\Sigma(t'_2)$. O.B.d.A. sei $at \in \mathfrak{TransTerm}_\Sigma(t'_1)$. Aus der Diskussion von same, Fall a) können wir Folgendes übernehmen: $\sigma' = \sigma|_{Var(ax)}$ ist eine gewöhnliche, wohlsortierte Substitution (*), und $t'_1\sigma = t'_1\sigma' \in unterTerme([AX/Inst])$. Wegen (*) folgt aus Satz 4.1.5
  $(\mathfrak{TransTerm}_\Sigma(t'_1))\sigma' = \mathfrak{TransTerm}_\Sigma(t'_1\sigma') = \mathfrak{TransTerm}_\Sigma(t'_1\sigma)$. Nun müssen wir für alle $t \in \{t_1, \ldots, t_m, t_0\}$ zeigen, daß *jeweils* ein
  $t' \in unterTerme([AX/Inst])$ existiert mit
  $mtval_{\mathcal{I}}(t) \in unterTerme(val_{\mathcal{I}}(t'))$.

Betrachten wir zuerst $t = t_0$:
Wir wissen: t_0 hat die Form $t_0 = \mathsf{val}(t_0'')$. Da $at \in (\mathfrak{TransTerm}_\Sigma(t_1'))\sigma$, gilt nach Satz 4.5.16, 1.a), daß $t_0'' \in \mathit{unterTerme}(t_1'\sigma) \subseteq \mathit{unterTerme}([AX/\mathit{Inst}])$.
Wählen wir also $t' = t_0''$. Es gilt:
$\mathit{mtval}_\mathcal{I}(t) = \mathit{mtval}_\mathcal{I}(\mathsf{val}(t')) = \mathit{val}_\mathcal{I}(t') \in \mathit{unterTerme}(\mathit{val}_\mathcal{I}(t'))$.
Nun betrachten wir o.B.d.A. $t = t_1$:
Wir wissen: $t_1 = \mathfrak{Rep}(t_1'')$. Ist nun $\mathfrak{Rep}(t_1'') = \mathsf{val}(t_1'')$, dann ergibt sich mit $t' = t_1''$ die gleiche Argumentation wie bei $t = t_0$. Andernfalls ist $t_1 = \mathfrak{Rep}(t_1'') = t_1''$ und $t_1 \in CT_\Sigma$. Wegen $at \in \mathfrak{TransTerm}_\Sigma(t_1'\sigma)$ und nach Satz 4.5.16, 1.b), gilt dann $t_1 \in \mathit{unterTerme}(t_1'\sigma) \subseteq \mathit{unterTerme}([AX/\mathit{Inst}])$.
Wir wählen also diesmal t' als t_1 selbst. Es gilt:
$\mathit{mtval}_\mathcal{I}(t) = \mathit{mtval}_\mathcal{I}(t_1) = t_1 = \mathit{val}_\mathcal{I}(t_1) \in \mathit{unterTerme}(\mathit{val}_\mathcal{I}(t_1))$.

□

Die zentrale Idee zu diesem Beweis sowie zum Vollständigkeitsbeweis überhaupt verbirgt sich in dem recht unscheinbaren Unterfall 'b)' des Punktes „Sei $at = \mathsf{is}(t_1, t_2)$ ...". Dies ist die einzige Stelle in der obigen Fallunterscheidung, in der das Zutreffen von $\mathit{Deut}_\mathcal{I}(A)$ ausgenutzt wird, und zwar dazu, die günstigen Eigenschaften der argval-Terme, die das erste is-Argument belegen, auf das zweite Argument zu übertragen. Von da aus werden sie durch die Ersetzungsregeln überall hin propagiert. Die argval-Terme sind durch ihre den Axiomen entstammende Basis eng mit der Spezifikation verknüpft. Das is-Atom fungiert als Brücke, diese Verbindung zur Spezifikation auch den anderen Meta-Termen 'zugänglich zu machen'. Dies funktioniert aber nur auf einem Ast, auf dem das gedeutete is-Atom auch 'zutrifft'.

Den eben hergestellten Zusammenhang zwischen Top-Termen eines Astes und der Auswertung von Termen der Spezifikation können wir nun verwenden, um die Länge von Ästen A (in denen $\mathit{Deut}_\mathcal{I}(A)$ zutrifft) sehr grob abzuschätzen.

Dafür müssen wir u.a. die relativ zu einer Termmenge maximale Auswertung bestimmen.

Definition 4.5.19 (maximale Größe ausgewerteter Terme)
Sei $\mathcal{I}$ eine $\mathcal{F}$-Interpretation.
Die Abbildung $\mathit{maxval}_\mathcal{I} : \wp(T_\Sigma^0) \rightarrow \mathbb{N}$ weist jeder endlichen Menge von Σ-Grund-Termen die maximale Größe der Auswertung dieser Terme (mit $\mathcal{I}$) zu, formal:
Sei $T \subseteq T_\Sigma^0$ endlich.

$$\begin{gathered} \mathit{maxval}_\mathcal{I}(T) = n \\ :\Longleftrightarrow \\ |\mathit{val}_\mathcal{I}(t)| = n \text{ für ein } t \in T \\ \text{und} \\ \text{f.a. } t' \in T \text{ gilt } |\mathit{val}_\mathcal{I}(t')| \leq n \end{gathered}$$

⋆

Das Ergebnis der Auswertung mit $val_{\mathcal{I}}$ ist ja immer ein Konstruktorterm ct. Seine 'Größe' $|ct|$ (siehe Def. 2.2.14) kann intuitiv auf zweierlei Art gedeutet werden. $|ct|$ bezeichnet in gleicher Weise die Anzahl der Konstruktoren in ct wie auch die Anzahl der Unterterme von ct. Beide Vorstellungen sind im folgenden hilfreich. Die maximale Größe ausgewerteter Unterterme der (instantiierten) Axiome,

$$maxval_{\mathcal{I}}(unterTerme([AX/Inst])),$$

kann als obere Schranke dienen für drei Merkmale von Meta-Termen:

- die *Anzahl* auftretender Konstruktoren (außerhalb der argval-Terme) (s.u. Satz 4.5.20(3))
- die *Anzahl* unabhängig auftretender argval-Terme (s.u. Satz 4.5.20(4))
- die *Tiefe* auftretender argval-Terme (s.u. Satz 4.5.20(2))

Dies wird jetzt nachgewiesen und anschließend dafür benutzt, die Existenz der in Satz 4.5.3 behaupteten Schranke zu begründen.

Satz 4.5.20 *Sei* $\langle \Sigma, AX \rangle$ *eine normalisierte ADT-Spezifikation,* $Inst \subseteq CT_{\Sigma}$ *endlich und* $R = \mathfrak{TransSpec}(\langle \Sigma, AX \rangle) \cup \mathfrak{TransInst}_{\Sigma}(Inst)$.
Sei weiterhin B *ein* R*-erreichbarer Baum, sei* $A \in äste(B)$ *und* $\mathcal{I}$ *eine* $\mathcal{F}$*-Interpretation, wobei* $Deut_{\mathcal{I}}(A)$ *zutrifft. Sei außerdem* $at \in A$ *ein Meta-Atom in* A*, welches kein Sorten-Atom ist* ($at \neq s(ct)$)*, und* $t \in topTerme(at)$*. Dann gilt:*

1. $|mtval_{\mathcal{I}}(t)| \leq maxval_{\mathcal{I}}(unterTerme([AX/Inst]))$.
2. $anzÄußKonstr(t) \leq maxval_{\mathcal{I}}(unterTerme([AX/Inst]))$.
3. $anzUnabhArgvalTerme(t) \leq maxval_{\mathcal{I}}(unterTerme([AX/Inst]))$.
4. *Falls* $t' \in metaUnterterme(t)$ *ein argval-Term der Form* $t' = \mathsf{arg}n_k(\ldots(\mathsf{arg}n_1(\mathsf{val}(t'')))\ldots)$ *ist, mit Tiefe* $k \geq 0$*, dann gilt:* $k \leq maxval_{\mathcal{I}}(unterTerme([AX/Inst]))$.

Beweis:
1. Nach Satz 4.5.18 existiert ein $t' \in unterTerme([AX/Inst])$ mit $mtval_{\mathcal{I}}(t) \in unterTerme(val_{\mathcal{I}}(t'))$. Wenn aber $mtval_{\mathcal{I}}(t)$ ein Unterterm von $val_{\mathcal{I}}(t')$ ist, dann ist sicherlich $|mtval_{\mathcal{I}}(t)| \leq |val_{\mathcal{I}}(t')|$. Nach der Def. von $maxval_{\mathcal{I}}$ ist außerdem $|val_{\mathcal{I}}(t')| \leq maxval_{\mathcal{I}}(unterTerme([AX/Inst]))$. Demnach ist auch $|mtval_{\mathcal{I}}(t)| \leq maxval_{\mathcal{I}}(unterTerme([AX/Inst]))$.

2. Aus 1. und $anzÄußKonstr(t) \leq |mtval_{\mathcal{I}}(t)|$ (Satz 4.5.14(1)) folgt unmittelbar die Behauptung.

3. Aus 1. und $anzUnabhArgvalTerme(t) \leq |mtval_{\mathcal{I}}(t)|$ (Satz 4.5.14(2)) folgt unmittelbar die Behauptung.

4. Da $Deut_{\mathcal{I}}(A)$ zutrifft, trifft auch $Deut_{\mathcal{I}}(at)$ zu. Insbesondere ist $Deut_{\mathcal{I}}(at)$ wohldefiniert, somit auch $mtval_{\mathcal{I}}$ auf allen Meta-Unter-Termen von at. Insbesondere ist $mtval_{\mathcal{I}}$ wohldefiniert auf t' und allen Untertermen von t'. Betrachten wir nun die Menge:

$$\begin{array}{rcr} metaUnterterme(t') & = & \{\mathsf{arg}n_k(\ldots\ldots\ldots(\mathsf{arg}n_1(\mathsf{val}(t''))) \ldots), \\ & & \mathsf{arg}n_{k-1}(\ldots(\mathsf{arg}n_1(\mathsf{val}(t''))) \ldots), \\ & & \vdots \\ & & \mathsf{arg}n_1(\mathsf{val}(t'')), \\ & & \mathsf{val}(t'')\ \} \end{array}$$

Dann betrachten wir die Auswertungen dieser Menge:
$M = \{mtval_{\mathcal{I}}(mt) \mid mt \in metaUnterterme(t')\}$.
Es gilt: $|M| = k+1$, da $mtval_{\mathcal{I}}(t_1) \neq mtval_{\mathcal{I}}(t_2)$ f.a. $t_1, t_2 \in metaUnterterme(t')$.
Außerdem gilt nach Satz 4.5.12, daß $M \subseteq unterTerme(val_{\mathcal{I}}(t''))$, also $|M| \leq |val_{\mathcal{I}}(t'')|$.
Laut Satz 4.5.17, 1., gilt außerdem $t'' \in unterTerme([AX/Inst]))$. Zusammen gilt:
$k \leq k+1 = |M| \leq |val_{\mathcal{I}}(t'')| \leq maxval_{\mathcal{I}}(unterTerme([AX/Inst])$. □

Mit der Gültigkeit von Satz 4.5.20 ist intuitiv schon klar, daß ein Ast, auf dem $Deut_{\mathcal{I}}$ zutrifft, eine gwisse Maximallänge nicht überschreiten kann. Man kann eben nur eine bestimmte Anzahl von Atomen bilden, wenn die darin auftretenden Terme beschränkt sind, sowohl in der Anzahl der Konstruktoren und argval-Terme, als auch in der Tiefe der argval-Terme.

Wir beweisen nun den entsprechenden Satz 4.5.3 aus dem letzten Abschnitt. Formal handelt es sich nur um eine Beweisskizze in dem Sinne, daß wir keine konkrete Schranke ausrechnen, sondern uns lediglich davon überzeugen, daß man eine konkrete Schranke ausrechnen kann.

Beweis: (von Satz 4.5.9)
Beweisskizze:

Zunächst überzeugen wir uns davon, daß die Anzahl der verschiedenen möglichen Top-Terme in dem Atom $at \in A$ begrenzbar ist. Wir wissen aus Satz 4.5.17, daß alle Meta-Unter-Terme von at wohlgeformt sind, also insbesondere auch die Top-Terme. Wohlgeformte Meta-Terme bestehen aber nur aus Konstruktoren und argval-Termen. Um möglichst einfach argumentieren zu können, skizzieren wir eine *extrem grobe* obere Schranke für die Anzahl verschiedener möglicher Top-Terme.

Nach Satz 4.5.20 ist die Tiefe möglicher argval-Terme begrenzt, und da die Basis jedes argval-Terms immer ein Unterterm von $[AX/Inst]$ ist (s. Satz 4.5.17), läßt

sich sicherlich auch die Menge der konstruierbaren argval-Terme beschränken. Nun fassen wir die wohlgeformten Meta-Terme auf als Wörter, und zwar mit folgenden Terminalsymbolen: Konstruktoren, Kommata, Klammern und (!) argval-Terme.

Nun wissen wir aus Satz 4.5.20, daß in den Top-Termen von *at* sowohl die Anzahl der Konstruktoren als auch die Anzahl unabhängiger argval-Terme begrenzt ist, durch $maxval_{\mathcal{I}}(unterTerme([AX/Inst]))$. Gleiches gilt für die Anzahl der Kommata, die Anzahl der öffnenden und die Anzahl der schließenden Klammern. Daher ist die Menge der Wörter begrenzt, die insbesondere auch die Menge der möglichen Top-Terme enthält. Wir haben also eine (furchtbar grobe) Schranke für die Menge möglicher Top-Terme.

Es ist klar, daß sich infolgedessen auch die Menge der Meta-Atome, die diese Top-Terme enthalten, begrenzen läßt. Hierbei ist noch folgendes zu beachten. Die I-Atome enthalten, außer dem Top-Term im letzten Argument, noch zwei andere Arten von Argumenten. Erstens Tupel von Top-Termen, welche beschränkt sind durch die maximale Stelligkeit $max\alpha(\Sigma)$ der Signatur. Zweitens Funktionen, welche aber auch beschränkt sind durch die Signatur.

Zuletzt ist noch darauf hinzuweisen, daß die oben geführte Diskussion eigentlich nur für $at \neq s(ct)$ greift (s. Satz 4.5.18). Aber die Menge der Atome der Form $s(ct)$ ist laut Satz 4.5.17 auch begrenzt, und zwar durch $|Inst|$. □

5 Realisierung und Beispiele

In diesem Kapitel wird erläutert, wie der bis hierher theoretisch beschriebene Ansatz zur *deduktiven Fehlersuche in abstrakten Datentypen* praktisch umgesetzt wurde. Desweiteren wird die Anwendung dieser Realisierung auf einige Beispiele dokumentiert.

Zur eigentlichen *Ausführung* der regelgesteuerten Modell- (und damit Fehler-) suche wurde ein ausgereiftes, an der Universität Fukuoka in Japan entwickeltes Werkzeug namens MGTP eingesetzt. Sehr vorläufig ist jedoch die spezielle, auf das Anwenderszenario der (möglicherweise) fehlerhaften Datentypen zugeschnittene *Benutzungsschnittstelle*. Sie ist zwar dafür geeignet, das Verfahren mit beliebigen Beispielen zu testen, aber sie bietet noch nicht die angestrebte Einbettung der vorgestellten Methodik in einen Kontext, in welchem fehlerhafte Aussagen über Datentypen ganz natürlich anfallen. Mittelfristig ist eine systemische Einbettung des Verfahrens in ein Werkzeug zur formalen Software-Entwicklung vorgesehen. Im speziellen handelt es sich hierbei um das KeY-System [ABB^{+}00], welches parallel zur vorliegenden Arbeit an der Universität Karlsruhe entwickelt wurde und wird. Erst eine solche Einbettung und der damit ermöglichte Einsatz in konkreten Fallstudien kann den Nachweis der praktischen Verwertbarkeit des Verfahrens erbringen. Die im folgenden dokumentierten Tests erheben diesen Anspruch nicht. Dennoch sind sie essentiell für die Beurteilung der prinzipiellen Tragfähigkeit des verfolgten Ansatzes.

Ein weiterer problematischer Punkt ist die Auswahl von Beispielen zum Test des Verfahrens. Die Schwierigkeit besteht in der mangelnden Verfügbarkeit echter, tatsächlich aufgetretener Fehlerfälle, für die die Methode gedacht ist. Denn: so oft Fehler auftreten, so selten werden sie dokumentiert. Es liegt in der Natur der Sache, daß Fehler (in Programmen, in Spezifikationen oder wo auch immer), sobald sie zutage treten, schnellstens beseitigt werden, nach Möglichkeit in aller Stille.

Aus diesem Grunde sind die fehlerhaften Aussagen, auf die das Verfahren in den hier dokumentierten Tests angewendet wurde, allesamt konstruiert, bis auf eine. Der einzige nachweisbar authentische Fehler, den wir im folgenden behandeln, stammt aus einem an der Universität Ulm erschienenen Technischen Bericht [RST00]. Dort tritt er auf in einer Spezifikation, die das tragende Beispiel der

Arbeit darstellt. Bemerkenswert ist, daß sich dieser Bericht mit ‘Fehlersuche in Formalen Spezifikationen’ beschäftigt. Dennoch haben die Autoren diesen Fehler in dem tragenden Beispiel nicht entdeckt. Diese Feststellung ist nicht als Vorwurf gemeint. Vielmehr demonstriert sie, daß, selbst bei geschärftem Blick des Lesers bzw. Autors einer Spezifikation, die Entdeckung subtiler Fehler schwierig ist und eine automatische Unterstützung hierbei sehr wünschenswert wäre.

5.1 Realisierung des Verfahrens

In den vorangegangenen Kapiteln wurde ein Ansatz zur Identifikation fehlerhafter Aussagen über ADT-Spezifikationen vorgestellt. Konzeptionell lassen sich drei Schritte unterscheiden. Gegeben sei zunächst eine ADT-Spezifikation $\langle \Sigma, AX \rangle$ und eine Vermutung φ. Der Versuch, festzustellen, ob φ *nicht* aus der Spezifikation folgt, d.h. ob $\langle \Sigma, AX \rangle \not\models \varphi$ zutrifft, gliedert sich in die folgenden Schritte:

1. Bildung einer normalisierten Gegenspezifikation (vgl. Abschnitte 2.4 und 2.5), diese nennen wir SPEC'.

2. Transformation von SPEC' in eine Menge von Modell-Generierungs-Regeln. Erweiterung dieser Regel-Menge um Regeln zur Bereitstellung von Instanzen, bei vorgegebener Instanzen-Menge *Inst*.

3. Konstruktion eines Modells (einer erfüllenden Interpretation) der mit *Inst* instantiierten Gegenspezifikation.

Darüberhinaus wurde eine Eingabemöglichkeit für Spezifikationen und Vermutungen realisiert, sowie eine rudimentäre Nachbereitung des gefundenen Modells. Im folgenden beschreiben wir die systemische Umsetzung dieser Schritte kurz im einzelnen.

Eingabe der Spezifikation und Vermutung

Da die Methode gedacht ist als Baustein eines größeren Systems, in welchem die angezielten Fehler direkt auftreten, schien es weder sinnvoll noch ökonomisch, ein eigenes Front-End zu erstellen. Stattdessen wurde, stellvertretend, eine Schnittstelle implementiert zu einem existierenden Werkzeug namens IBiJa ([Hab00]). Dies ist ein in Java implementiertes Programm zum interaktiven Beweisen von (zutreffenden) Aussagen über abstrakte Datentypen. Es bietet eine fensterorientierte, menuegesteuerte Benutzerschnittstelle zur Eingabe von Spezifikationen und Vermutungen (dort optimistisch ‘Lemmata’ genannt), und darüber hinaus,

als Haupt-Eigenschaft, eine interaktive Beweisumgebung. Unsere Realisierung benutzt nur die Eingabe, d.h. insbesondere den *Parser* für Spezifikationen und Vermutungen (Lemmata). Eine Normalisierung von Spezifikationen wurde zunächst nicht implementiert, d.h. der Benutzer/die Benutzerin muß in IBiJa normalisierte Spezifikationen eingeben. In der Praxis bedeutet dies meist nur, daß Formeln der Form $\varphi \rightarrow \psi$ ersetzt werden müssen durch $\neg\varphi \vee \psi$. (Allerdings muß sich der Benutzer/die Benutzerin *nicht* um die Umwandlung der jeweiligen *Vermutung* in ihr normalisiertes Gegenteil kümmern, s.u.)

Transformation

Der annotierte abstrakte Syntax-Baum, der von dem IBiJa-Parser erzeugt wird, bildet die grundlegende Datenstruktur für die nun zu leistende Transformation. Auf dieser Datenstruktur werden in Vorbereitung der eigentlichen Transformation zunächst die Negierung, der Existenzabschluß und die Skolemisierung der Vermutung ausgerechnet. Auf der Grundlage der so erweiterten Spezifikation wird die Transformation in Modell-Generierungs-Regeln durchgeführt. Die Implementierung dieser Transformation besteht in einem Java-Programm, welches MG-Regeln erzeugt, die der Eingabesyntax des verwendeten Werkzeugs zur Ausführung der Regeln genügen. Im großen und ganzen stimmt diese Syntax mit der in Def. 3.4.11 definierten überein. Die wenigen minimalen Abweichungen hiervon (s.u.) dienen der Berücksichtigung von Namenskonventionen in MGTP. Um einen Vergleich zwischen der symmetrischen und der optimierten Transformation der Literale zu ermöglichen (vg. Def. 3.2.10 und 3.2.12), wurden beide implementiert. Die Auswahl der jeweiligen Variante wird durch eine Konfigurationsdatei gesteuert. Als Standard wird die optimierte Transformation ausgeführt.

Modellkonstruktion

Zur Abarbeitung der erzeugten Regelbasis wird das schon erwähnte Werkzeug MGTP (Model Generation Theorem Prover) eingesetzt. Dieses wurde ursprünglich im Kontext des japanischen ‘fifth generation’-Programms enwickelt und realisiert eine konstruktive Überprüfung der Konsistenz von Wissensbasen, die sich als ‘range restricted clauses’ (wertebeschränkte Klauseln) darstellen lassen. Unser in der Transformation erzeugtes Regelsystem stellt eine solche Wissensbasis dar.[1] Der Zusammenhang zwischen unseren Meta-Atomen und der Semantik abstrakter Datentypen ist für MGTP bedeutungslos. Das Programm richtet sich bei der Erweiterung oder Ablehnung von (Modell-)Ästen nach nichts anderem als nach vorgegebenen Regeln. Im Kern handelt es sich bei MGTP um eine (inzwischen) hocheffiziente Java-Implementierung der in Abb. 3.3, S. 107 in Pseudo-Code notierten Prozedur.

[1] In der MGTP-Welt werden diese Regeln als ‘clauses’ bezeichnet.

Neben der Verwendung gewöhnlicher Atome in den Regeln (clauses) erlaubt MGTP die Benutzung einiger 'extra-logischer' Instruktionen. Dazu gehört die Verwendung einfacher Java-Integer-Arithmetik und ein Test auf syntaktische Ungleichheit von Termen. Diese beiden Möglichkeiten werden von dem Verfahren ausgenutzt, und zwar in folgender Weise: Blicken wir nochmals zurück auf die Behandlung von Instanzen in der Transformation. Für die Theoriebildung hatten wir in Def. 3.3.11 die Transformation beliebiger Mengen von Instanzen betrachtet. In der darauffolgenden Def. 3.3.12 wurde dieses verfeinert zu der Transformation einer Größenbeschränkung n für Instanzen. Laut dieser Def. wird für jeden Konstruktorterm, der klein genug ist, die Regel '$\rightarrow\ s(ct)$.' erzeugt. Im Ergebnis werden all diese s-Atome bei der Modellkonstruktion erzeugt. Dies kann man unter Ausnutzung der Integer-Arithmetik in MGTP auch mit weniger Regeln erreichen. Wir erläutern dies exemplarisch für das Beispiel NatStack.

Die Regeln werden hier so wiedergegeben, wie sie automatisch erzeugt wurden. MGTP sieht Ausdrücke, die mit einem Großbuchstaben beginnen, als Variablen an. Der Unterstrich „_" dient als anonyme Variable, die auf der rechten Seite der Regel nicht mehr verwendet wird. Zahlen, die für sich alleine stehen, also nicht an Variablen angehängt sind (hier `1` und `4`), werden als Integer-Werte im programmiersprachlichen Sinne erkannt. Sie können mittels `+`, `<=` bzw. `=` addiert, verglichen bzw. zugewiesen werden. Extralogische Instruktionen werden durch Klammerung `{}` als solche gekennzeichnet. `%` ist ein Kommentarzeichen. `->` trennt die Prämisse einer Regel von der Konklusion. (Bisher hatten wir „→" verwendet.)

```
%Domain Boundary

-> max(4) .

%Generating Domain for nat

-> gennat(zero,1) .
max(M),gennat(X0,Y0),{Y1=1+Y0},{1+Y0<=M} -> gennat(succ(X0),Y1) .

%Generating Domain for stack

-> genstack(nil,1) .
max(M),gennat(X0,Y0),genstack(X1,Y1),{Y2=1+Y0+Y1},{1+Y0+Y1<=M}
  -> genstack(push(X0,X1),Y2) .

%Redirecting the Domain from GenSort ...

gennat(X,_) -> nat(X) .
genstack(X,_) -> stack(X) .
```

Die Regeln dienen dem Zweck, für alle Konstruktorterme *ct* der beiden Sorten *Nat* und *Stack* (vgl. Spezifikation `NatStack` auf S. 164), die *höchstens* vier Konstruktoren besitzen, das Meta-Atom `nat(`*ct*`)` bzw. `stack(`*ct*`)` zu erzeugen. Dazu werden zunächst Atome der Form `gennat(`*ct*`,`*n*`)` bzw. `genstack(`*ct*`,`*n*`)` erzeugt, wobei die Integer-Zahl *n* gleich der Anzahl der Konstruktoren in *ct* ist. Die Instruktionen `{Y1=1+Y0}` und `{Y2=1+Y0+Y1}` dienen dazu, bei Anwendung der Regel die Größe des neu entstehenden Konstruktorterms zu bestimmen. Die Testinstruktionen `{1+Y0<=M}` und `{1+Y0+Y1<=M}` *verhindern die Regelanwendung*, falls die Testbedingung verletzt ist, falls also der entstehende Konstruktorterm die zulässige Maximalgröße `M` überschreiten würde. Der Wert von `M` ist deswegen nicht 'hart' in die Regel kodiert, damit eine Veränderung dieser Grenze der Abänderung nur einer Regel bedarf, nämlich der Regel '`-> max(4) .`'. Somit ist die Festlegung einer Maximalgröße von Instanzen modular zur eigentlichen Transformation von Spezifikationen.

Eine weitere extralogische Instruktion von MGTP verwenden wir dafür, die Regeln zur Sicherstellung der Funktionalität (vgl. Def. 3.2.13) zu optimieren. Statt

```
intpr(F,X,Y),intpr(F,X,Y1) -> same(Y,Y1) .
is(X,Y),is(X,Y1) -> same(Y,Y1) .
```

werden die Regeln

```
intpr(F,X,Y),intpr(F,X,Y1),{Y\==Y1} -> same(Y,Y1) .
is(X,Y),is(X,Y1),{Y\==Y1} -> same(Y,Y1) .
```

erzeugt. Hierbei ist `intpr` das Meta-Prädikat, welches wir bisher (in Kapitel 3) mit I bezeichet haben. (`I` ist nicht geeignet, da es von MGTP als Variable angesehen wird.) `{Y\==Y1}` bewirkt, daß die Regel nur angewendet wird, wenn die auf `Y` und `Y1` matchenden Meta-Terme rein syntaktisch verschieden sind. Dies ist korrekt, weil die Anwendung von `same` auf zweimal den gleichen Meta-Term *nicht* zur Ablehnung eines Astes führen kann (vgl. Def. 3.2.5) und darum auch nichts beiträgt . Es ist in diesem Zusammenhang sehr wichtig, zwischen der syntaktischen und der semantischen Gleichheit zu unterscheiden. Die obigen Regeln sind zwar nur anwendbar, wenn `Y` und `Y1` syntaktisch verschieden sind, aber damit ist noch nicht entschieden, ob sie auch semantisch ungleich sind (sonst könnte man ja sofort ablehnen, ohne erst `same(Y,Y1)` zu erzeugen). In beiden Meta-Terme können noch `val`- oder `arg`*n*-Terme als Platzhalter auftreten, nach deren Ersetzung noch gesucht wird. In diesem Falle wird die 'Gültigkeit' von `same(Y,Y1)` erst später 'entschieden'.

Der Effekt dieser Optimierung ist dramatisch. Ohne die Vergleichs-Instruktion löst *jedes einzelne* `intpr`-Atom einen Vergleich seines letzten Argumentes mit sich selbst aus. Da ein `same`-Atom i.a. weitere `same`-Atome erzeugt, verursacht dies erheblich größere Modell-Äste als notwendig.

Abgesehen von den gerade diskutierten Abweichungen führt MGTP genau die in Abschn. 3.2 definierten Regeln aus, und zwar in der Weise, wie dies abstrakt in der Pseudo-Code-Prozedur *MG-proc* (S. 107) beschrieben ist.

Aufbereitung der Ausgabe

MGTP gibt die lineare Darstellung eines Baumes zurück, der entweder nur geschlossene Äste besitzt oder höchstens einen saturierten, offenen Ast, welcher dann der rechteste ist. Zusätzlich enthält die Antwort im ersten Fall '`UNSAT`', im zweiten Fall '`SAT`'. Falls MGTP mit der Antwort '`SAT`' terminiert, dann kann aus dem einzig offenen Ast die Interpretation abgelesen werden, welche die instantiierte Gegenspezifikation erfüllt. Diese Interpretation hat zwei wesentliche Eigenschaften:

- Sie wertet die Vermutung φ zu F (falsch) aus, da sie mit der Gegenspezifikation auch das Gegenteil von φ erfüllt.
- Sie erfüllt eine Grund-Instantiierung der Spezifikation $\langle\Sigma, AX\rangle$, und zwar mit Instanzen bis zu der mit '`-> max(`n`) .`' festgelegten Größe.

Damit ist noch nicht endgültig bewiesen, daß $\langle\Sigma, AX\rangle \not\models \varphi$ zutrifft. Der Benutzer/die Benutzerin muß selbst entscheiden, ob sich die gefundene Interpretation zu einer erfüllenden Interpretation für die echte, uninstantiierte Spezifikation $\langle\Sigma, AX\rangle \not\models \varphi$ erweitern läßt. Um so wichtiger ist eine vernünftige Rückmeldung.

Der saturierte, offene Ast besteht aber zunächst einmal aus einer Menge von `intpr`-, `is`-, `search`-, `same`-, `different`- und Sorten-Atomen. Zur Deutung des Ergebnisses benötigen wir aber (fast) nur die `intpr`-Atome. Die Suche nach genau diesen Atomen (in Kapitel 3 nannten wir sie I-Atome) war ja der Ausgangspunkt für die Entwicklung des Verfahrens. Darüberhinaus interessieren im Ergebnis nicht alle, sondern nur bestimmte dieser Atome. Im Laufe der Suche erzeugt das Verfahren vielfach Atome mit Platzhaltern (`val`- oder `arg`n-Termen), die im weiteren Verlauf ersetzt werden. Am Ende interessieren nur die vollständig ersetzten `intpr`-Atome. Diese werden also herausgefiltert. Von besonders großem Wert ist ein Spezialfall der `intpr`-Atome: hat es die Form `intpr(sk_x,tup0,`ct`)`[2], dann können wir aus der Namenskonvention für Skolemfunktionen zurückschließen, daß `sk_x` die Skolemisierung der Variablen x aus der Vermutung φ ist. Für diese Vermutung heißt das: *die Belegung der Variable x mit ct* führt (zusammen mit weiteren Variablenbelegungen und der gefundenen Interpretation von Funktionen) zum Wahrheitswert F (falsch). Diese primitive, aber aus Benutzersicht

[2] Da MGTP keine Tupel kennt, simulieren wir sie durch Terme. `tup0` steht für den nullstelligen Tupel.

sehr hilfreiche Umdeutung der Interpretation von Skolemkonstanten in Belegungen von Variablen der ursprünglichen Vermutung wird in der Aufbereitung der MGTP-Ausgabe vorgenommen. Auf diese Weise bleibt das Konzept der skolemisierten Gegenspezifikation weitgehend unsichtbar. Schließlich wird noch ausgenutzt, daß ein Meta-Atom der Form `is(val(`t`),`ct`)`, in welchem ct ein reiner Konstruktorterm ist, gedeutet werden kann als die Aussage $\mathsf{val}_{\mathcal{I}}(t) = ct$. Tritt in t eine Skolemkonstante auf (und enthielt die ursprüngliche Spezifikation keine Existenzquantoren), dann 'stammt' der Term aus der Vermutung φ. Somit ist seine Auswertung unter der gefundenen Interpretation von besonderem Interesse. Entsprechend wird eine 'evaluation of conjecture subterms' ausgegeben. Auch hier werden die Skolemkonstanten stillschweigend durch die entsprechenden Variablen der Vermutung ersetzt.

Einen exemplarischen Eindruck von der Ausgabe des realisierten Verfahrens kann man anhand der Beispielläufe im folgenden Abschnitt gewinnen.

5.2 Beispiele

In diesem Abschnitt wir nun der Einsatz des Verfahrens an Beispielen erläutert. Dabei wird das Ergebnis sowohl der Transformation als auch der Modell-Konstruktion dokumentiert. Eine erste Gruppe von Beispielen besteht in fehlerhaften Aussagen über die schon bekannte NatStack-Spezifikation. Diese sind, wie schon erwähnt, konstruiert. In all diesen Beispielen liegt das Problem im Randfall begründet. Aus der persönlichen Erfahrung mit datentyp-basierter Programmverifikation hält der Autor solche Fehler für typisch. Es bleibt aber dem Leser/der Leserin überlassen, sich dieser Meinung anzuschließen. Danach werden anhand einer zweiten Spezifikation die Grenzen des vorgestellten Verfahrens exemplarisch verdeutlicht. Schließlich behandeln wir einen echten, in der Literatur vorgefundenen Fehler in einer Spezifikation eines 'merge sort' Algorithmus.

Die Beispiele werden mit abnehmender Ausführlichkeit besprochen, da bestimmte Erläuterungen zur Transformation bzw. zur Ausgabe für alle Beispiele gleichermaßen gelten.

5.2.1 NatStack

Wir hatten in Kapitel 2 die Beispielspezifikation NatStack kennengelernt. Sie stellt natürliche Zahlen bereit sowie Stacks natürlicher Zahlen. Spezifizierte Funktionen sind der Vorgänger *pred* einer natürlichen Zahl, das oberste Element *top* und der Rest *pop* eines Stacks, sowie eine Löschoperation *del* auf Stacks. Zur leichteren Nachvollziehbarkeit der im folgenden angegebenen Transformation(en)

spec = NatStack
  sorts Nat ::= zero | succ(Nat);
    $Stack$ ::= nil | push(Nat; $Stack$);
  functions $pred: Nat \to Nat$;
    $top: Stack \to Nat$;
    $pop: Stack \to Stack$;
    $del: Nat \times Stack \to Stack$;
  vars $n, n': Nat$; $st: Stack$;
  axioms $pred(\mathsf{succ}(n)) \doteq n$;
    $top(\mathsf{push}(n, st)) \doteq n$;
    $pop(\mathsf{push}(n, st)) \doteq st$;
    $del(n, \mathsf{push}(n, st)) \doteq st$;
    $n \not\doteq n' \to$
      $del(n, \mathsf{push}(n', st)) \doteq \mathsf{push}(n', del(n, st))$;
    $del(n, \mathsf{nil}) \doteq \mathsf{nil}$;
end

Abbildung 5.1: Spezifikation NatStack

wiederholen wir die komplette Spezifikation in Abb. 5.1. Vor Anwendung der Transformation wird noch das Axiom $n \not\doteq n' \to \ldots$ ersetzt durch $n \doteq n' \vee \ldots$.

Optimierte Transformation

In Def. 3.2.14 hatten wir die Transformation $\mathfrak{TransSpec}$ von Spezifikationen definiert. Die Java-Implementierung von $\mathfrak{TransSpec}$ wird nun angewendet auf die Spezifikation NatStack. Dabei wird, gemäß den Standardeinstellungen, die optimierte Variante von $\mathfrak{TransLit}_\Sigma$ (vgl. Def. 3.2.12) verwendet. Das Resultat wird im folgenden wiedergegeben. Die Behandlung einer Vermutung wird zunächst einmal ausgelagert. Es sei nochmals darauf hingewiesen, daß `intpr( , , )` für I(, ,) steht, `tupn`$(t_1, \ldots, t_n)$ für $\langle t_1, \ldots, t_n \rangle$ und `_` für anonyme Variablen. `%`-Zeilen sind Kommentare.

```
%Generating Domain for nat

-> gennat(zero,1) .
max(M),gennat(X0,Y0),{Y1=1+Y0},{1+Y0<=M} -> gennat(succ(X0),Y1) .

%Generating Domain for stack
```

```
-> genstack(nil,1) .
max(M),gennat(X0,Y0),genstack(X1,Y1),{Y2=1+Y0+Y1},{1+Y0+Y1<=M}
        -> genstack(push(X0,X1),Y2) .

%Redirecting the Domain from GenSort ...

gennat(X,_) -> nat(X) .
genstack(X,_) -> stack(X) .

%Search Domain for certain elements:

search_nat(X)
 ->
is(X,zero);
is(X,succ(arg(X))),search_nat(arg(X)) .

search_stack(X)
 ->
is(X,nil);
is(X,push(arg1(X),arg2(X))),
        search_nat(arg1(X)),search_stack(arg2(X)) .

%Rewriting with same

same(X,Y),is(X,Z) -> same(Z,Y) .
same(X,Y),is(Y,Z) -> same(X,Z) .

%Rewriting with different

different(X,Y),is(X,Z) -> different(Z,Y) .
different(X,Y),is(Y,Z) -> different(X,Z) .

%Functionality

intpr(F,X,Y),intpr(F,X,Y1),{Y\==Y1} -> same(Y,Y1) .
is(X,Y),is(X,Y1),{Y\==Y1} -> same(Y,Y1) .

%Result rewriting

intpr(F,X,Y),is(Y,Y1) -> intpr(F,X,Y1) .

%Rewriting for functions with 1 arguments
```

```
intpr(F,tup1(X0),Y),is(X0,Z) -> intpr(F,tup1(Z),Y) .

%Rewriting for functions with 2 arguments

intpr(F,tup2(X0,X1),Y),is(X0,Z) -> intpr(F,tup2(Z,X1),Y) .
intpr(F,tup2(X0,X1),Y),is(X1,Z) -> intpr(F,tup2(X0,Z),Y) .

%Rewrite Rules

is(X,succ(Y0)),is(Y0,Z) -> is(X,succ(Z)) .
is(X,push(Y0,Y1)),is(Y0,Z) -> is(X,push(Z,Y1)) .
is(X,push(Y0,Y1)),is(Y1,Z) -> is(X,push(Y0,Z)) .

%For each constructor 'freely generated' backwards rules:

same(zero,succ(_)) ->  .
different(zero,zero) ->  .
same(succ(_),zero) ->  .
same(succ(X0),succ(Y0)) -> same(X0,Y0) .
different(succ(X0),succ(Y0)) -> different(X0,Y0) .
same(nil,push(_,_)) ->  .
different(nil,nil) ->  .
same(push(_,_),nil) ->  .
same(push(X0,X1),push(Y0,Y1)) -> same(X0,Y0),same(X1,Y1) .
different(push(X0,X1),push(Y0,Y1)) ->
         different(X0,Y0);
         different(X1,Y1) .

%The Generated Axioms

%axiom-pred{ ( pred( succ( N ) ) = N )}

nat(N) -> intpr(pred,tup1(succ(N)),N) .

%axiom-top{ ( top( push( N, ST ) ) = N )}

nat(N),stack(ST) -> intpr(top,tup1(push(N,ST)),N) .

%axiom-pop{ ( pop( push( N, ST ) ) = ST )}

nat(N),stack(ST) -> intpr(pop,tup1(push(N,ST)),ST) .
```

```
%axiom-del1{ ( del( N, push( N, ST ) ) = ST )}

nat(N),stack(ST) -> intpr(del,tup2(N,push(N,ST)),ST) .

%axiom-del2{ ( del( N, nil ) = nil )}

nat(N) -> intpr(del,tup2(N,nil),nil) .

%axiom-del3{ ( ( N = M ) \/ ( del( N, push( M, ST ) ) =
%                              push( M, del( N, ST ) ) ) )}

nat(M),nat(N),stack(ST) -> same(N,M);
        intpr(del,tup2(N,push(M,ST)),val(push(M,del(N,ST)))),
        is(val(push(M,del(N,ST))),push(M,val(del(N,ST)))),
        intpr(del,tup2(N,ST),val(del(N,ST))),
        search_stack(val(del(N,ST))) .
```

Die ersten Regeln, zur Behandlung von `gennat` und `genstack`, zählen zur Ausgabe von $\mathfrak{TransSpec}$, weil sie signaturabhängig sind. Verbleibt nur noch $\mathfrak{MaxInst}_\Sigma$, die Transformation der Größenbeschränkung für Instanzen. Nach der Diskussion im letzten Abschnitt liefert $\mathfrak{MaxInst}_\Sigma(n)$ nun lediglich:

```
%Domain Boundary

-> max(n) .
```

wobei n entprechend zu ersetzen ist.

Symmetrische Transformation

Es ist sehr instruktiv, das obige Ergebnis der Axiomen-Transformation zu vergleichen mit einer Transformation, bei der die unoptimierte, symmetrische Übersetzung der Literale gemäß Def. 3.2.10 verwendet wird. Obwohl dies Platz kostet, geben wir auch hier die Transformation sämtlicher Atome an (selbstverständlich unter Weglassung der axiomen-unabhängigen Regeln), weil damit deutlich wird, daß *jedes* der Axiome in NatStack eine Gleichung enthält, das zu dem von der Optimierung betroffenen Typus gehört.

```
%The Generated Axioms

%axiom-pred{ ( pred( succ( N ) ) = N )}

nat(N) -> same(val(pred(succ(N))),N),
```

```
          intpr(pred,tup1(succ(N)),val(pred(succ(N)))),
          search_nat(val(pred(succ(N)))) .

%axiom-top{ ( top( push( N, ST ) ) = N )}

nat(N),stack(ST) -> same(val(top(push(N,ST))),N),
          intpr(top,tup1(push(N,ST)),val(top(push(N,ST)))),
          search_nat(val(top(push(N,ST)))) .

%axiom-pop{ ( pop( push( N, ST ) ) = ST )}

nat(N),stack(ST) -> same(val(pop(push(N,ST))),ST),
          intpr(pop,tup1(push(N,ST)),val(pop(push(N,ST)))),
          search_stack(val(pop(push(N,ST)))) .

%axiom-del1{ ( del( N, push( N, ST ) ) = ST )}

nat(N),stack(ST) -> same(val(del(N,push(N,ST))),ST),
          intpr(del,tup2(N,push(N,ST)),val(del(N,push(N,ST)))),
          search_stack(val(del(N,push(N,ST)))) .

%axiom-del2{ ( del( N, nil ) = nil )}

nat(N) -> same(val(del(N,nil)),nil),
          intpr(del,tup2(N,nil),val(del(N,nil))),
          search_stack(val(del(N,nil))) .

%axiom-del3{ ( ( N = M ) \/ ( del( N, push( M, ST ) ) =
%                             push( M, del( N, ST ) ) ) )}

nat(M),nat(N),stack(ST) -> same(N,M);
          same(val(del(N,push(M,ST))),val(push(M,del(N,ST)))),
          intpr(del,tup2(N,push(M,ST)),val(del(N,push(M,ST)))),
          search_stack(val(del(N,push(M,ST)))),
          is(val(push(M,del(N,ST))),push(M,val(del(N,ST)))),
          intpr(del,tup2(N,ST),val(del(N,ST))),
          search_stack(val(del(N,ST))) .
```

Jede dieser Regeln enthält im Vergleich zu der entprechenden optimierten Variante ein zusätzliches `same`- und ein zusätzliches `search`-Atom. Insgesamt enthalten die Regeln im Resultat der ersten Transformation ein `search`-Atom, diejenigen im Resultat der zweiten hingegen sieben! Entsprechend aufwendiger, obwohl ergebnisgleich, ist die Modellkonstruktion.

Die succ-pred-Vermutung

Wir wenden uns nun der ersten Vermutung zu und ihrer Untersuchung durch das Verfahren. Das erste Axiom in NatStack, $\mathit{pred}(\mathsf{succ}(n)) \doteq n$, besagt, daß *pred* die Inverse von succ ist. Man könnte vermuten, daß auch das Umgekehrte der Fall ist. Sei also

$$\varphi \;=\; \mathsf{succ}(\mathit{pred}(n)) \doteq n$$

unsere Vermutung. In einem konkreten Anwendungsszenario könnte φ sowohl explizit aufgestellt worden sein oder auch als Unterziel eines Beweisversuches zum Nachweis einer komplizierteren Vermutung anfallen.

Betrachten wir zunächst die automatisch erzeugte Transformation der Vermutung.[3]

```
%The Conjecture

%lemma{ ( succ( pred( N ) ) = N )}

-> different(val(succ(pred(sk_n))),val(sk_n)),
        is(val(succ(pred(sk_n))),succ(val(pred(sk_n)))),
        intpr(pred,tup1(val(sk_n)),val(pred(sk_n))),
        search_nat(val(pred(sk_n))),
        intpr(sk_n,tup0,val(sk_n)),
        search_nat(val(sk_n)) .
```

Man beachte: die resultierende Regel ist das Ergebnis der Anwendung von $\mathfrak{TransAxioms}_\Sigma$ (Def. 3.2.8) auf die (automatisch erzeugte) *Skolemisierung* des *Existenzabschlusses* der *Negation* von φ. Deswegen enthält die Regel keine Variable, die es bei Anwendung zu substituieren gälte, und aus dem gleichen Grund ist die Prämisse der Regel leer, ganz gemäß der Def. 3.2.8. Während die Variablen (in den anderen Regeln) bei Regel-Anwendung durch Instanzen der vorgegebenen Maximalgröße instantiiert werden, stößt die Anwendung obiger Regel die *Suche* nach einer Interpretation der Skolemkonstante `sk_n` an, vgl. die Atome `intpr(sk_n,tup0,val(sk_n))` und `search_nat(val(sk_n))`. Mögliche Ersetzungen für `val(sk_n)` werden mit Hilfe der search-Regeln konstruiert. (Sie sind in keiner Weise durch '`-> max(`n`) .`' beschränkt.)

Bevor wir MGTP auf all diese Regeln anwenden, werfen wir zunächst noch einen Blick auf die in der Vermutung verwendete Signatur. Intuitiv ist schnell klar, daß die Frage nach der Folgerbarkeit der Vermutung überhaupt nichts mit Stacks zu tun hat, somit auch nicht mit den Axiomen der Spezifikation, die Stack-Funktionen axiomatisieren. Wenn dies zutrifft, dann ist $\langle\Sigma, AX\rangle \models \varphi$ äquivalent zu $\mathit{pred}(\mathsf{succ}(n)) \doteq n \models_\Sigma \varphi$, somit auch $\langle\Sigma, AX\rangle \not\models \varphi$ äquivalent zu

[3] Die optimistische Bezeichnung „`lemma`" in der Kommentarzeile stammt aus IBiJa.

$pred(\mathsf{succ}(n)) \doteq n \not\models_\Sigma \varphi$ (da $pred(\mathsf{succ}(n)) \doteq n$ das einzige Axiom ist, welches nach Streichung der Stack-Axiome übrigbleibt). Es würde den Suchraum erheblich einschränken, wenn man dieses Wissen nutzen könnte. Dies gilt in gleicher Weise für Modellsuche wie auch für herkömmliche Beweissuche. Aus diesem Grunde wurden im Umfeld von Beweissystemen auch bereits Techniken entwickelt, die *syntaktisch einfach feststellbare* Unabhängigkeiten zwischen Vermutungen und Axiomen analysieren und zur sog. *Axiomenreduktion* ausnutzen [RS98]. Auf diese Techniken könnte man bei der Integration der vorgestellten Methode in ein Spezifikations- und Beweiser-Umfeld zurückgreifen. In der vorliegenden 'stand alone'-Realisierung wurde zu Testzwecken die Möglichkeit geschaffen, diejenigen Axiome (bzw. deren Transformation), die für die Folgerbarkeit der aktuellen Vermutung keine Relevanz haben, mit Hilfe einer Konfigurationsdatei quasi auszublenden. Diese Möglichkeit wurde in den dokumentierten Tests auch genutzt, was erklärt, warum z.B. in der folgenden MGTP-Ausgabe die Stack-Funktionen nicht auftauchen.

Nun betrachten wir, wenigstens einmal für dieses simple Beispiel, die unbearbeitete, vollständige Ausgabe von MGTP, wenn es mit den gesammelten, oben angegebenen Regeln (ohne die unoptimierten Versionen) aufgerufen wird. Die Stack-Regeln waren dabei 'ausgeblendet' (s.o.). Um die Ausgabe hier kurz zu halten, wurde mit '`-> max(2) .`' eine sehr kleine Obergrenze für die Größe der Konstruktorterme vorgegeben. Die folgende Ausgabe kann von dem Benutzer an- oder abgeschaltet werden, als Zusatz zu der aufbereiteten Ausgabe (s.u.). Um dem Leser/der Leserin das Nachvollziehen der Regelanwendungen zu erleichtern, fügen wir unten noch eine entsprechende Erklärung an.

Mgtp Output for this example:

```
Proving . . .
1: max(2).
2: gennat(zero,1).
3: genstack(nil,1).
4: different(val(succ(pred(sk_n))),val(sk_n)).
5: is(val(succ(pred(sk_n))),succ(val(pred(sk_n)))).
6: intpr(pred,tup1(val(sk_n)),val(pred(sk_n))).
7: search_nat(val(pred(sk_n))).
8: intpr(sk_n,tup0,val(sk_n)).
9: search_nat(val(sk_n)).
10: gennat(succ(zero),2).
11: nat(zero).
12: stack(nil).
13: different(succ(val(pred(sk_n))),val(sk_n)).
14: nat(succ(zero)).
15: intpr(pred,tup1(succ(zero)),zero).
```

```
16: intpr(pred,tup1(succ(succ(zero))),succ(zero)).
17: is(val(pred(sk_n)),zero).
18: intpr(pred,tup1(val(sk_n)),zero).
19: is(val(succ(pred(sk_n))),succ(zero)).
20: same(zero,val(pred(sk_n))).
21: same(val(pred(sk_n)),zero).
22: different(succ(zero),val(sk_n)).
23: same(succ(zero),succ(val(pred(sk_n)))).
24: same(succ(val(pred(sk_n))),succ(zero)).
25: same(zero,zero).
26: is(val(sk_n),zero).
27: different(succ(zero),zero).
28: different(succ(val(pred(sk_n))),zero).
29: different(val(succ(pred(sk_n))),zero).
30: intpr(sk_n,tup0,zero).
31: intpr(pred,tup1(zero),zero).
32: intpr(pred,tup1(zero),val(pred(sk_n))).
33: same(zero,val(sk_n)).
34: same(val(sk_n),zero).
pModel: true
nModel: []

Number of models: 1
Number of failed branches: 0
Number of total branches: 1
Number of total atoms: 34
SAT
Proving time: 9 msec
```

Erklärung hierzu (nicht Teil der Implementierung):

2, 3, 10, 11, 12 und 14 entstammen den Domain-Generierungs-Regeln,
4 – 9 negierte Vermutung,
13: rewriting 4 mit 5,
15: `axiom-pred` und 11,
16: `axiom-pred` und 14,
17: `search_nat` und 7,
18: rewriting 6 mit 17,
19: rewriting 5 mit 17,
20: Funktionalität auf 18 und 6,
21: Funktionalität auf 6 und 18,
22: rewriting 4 mit 19,
23: Funktionalität auf 19 und 5,
24: Funktionalität auf 5 und 19,

25: rewriting 20 mit 17,
26: `search_nat` und 9,
27 – 32: rewriting 22, 13, 4, 8, 18 und 6 mit 26,
33: Funktionalität auf 31 und 32,
34: Funktionalität auf 32 und 31.

Die obige Ausgabe stellt im Prinzip einen Baum dar, in diesem Fall mit nur einem Ast. Die wichtigste Information steht ganz unten: '`SAT`'. Es wurde also ein Modell gefunden. Man kann anhand des ausgegebenen Astes die Anwendung der oben angegebenen Regeln nachvollziehen, was wir hier nicht tun wollen. Nur soviel: erst werden die sog. Fakten, die Regeln mit leerer Prämisse angewendet, dann mit den drei Atomen `nat(zero)`, `stack(nil)`, `nat(succ(zero))` die Instanzen bereitgestellt, die kleiner gleich zwei Konstruktoren besitzen. Diese lösen dann die Anwendung der Axiomregeln (hier nur eine) aus sowie die Anwendung der aus der Vermutung resultierenden Regel. Auf die so entstandene Menge von Axiomen werden dann weitere Regeln angewendet. Zwar treten Verzweigungen auf, diese werden aber erst sichtbar, wenn die jeweils andere Alternative auch noch 'besucht' wird, was hier nicht notwendig ist.

Diese Ausgabe, ob sie nun angezeigt wird oder nicht, wird nach den im letzten Abschnitt besprochenen Prinzipien aufbereitet. Das Ergebnis kann wahlweise als im ASCII- oder in einem LaTeX-Format ausgegeben werden. Natürlich wird hier die LaTeX-Version wiedergegeben. Sie hebt sich vom übrigen Text dadurch ab, daß sie englisch ist.

Counter interpretation for the conjecture:
```
 succ( pred( N ) ) = N


15: intpr(pred,tup1(succ(zero)),zero)
16: intpr(pred,tup1(succ(succ(zero))),succ(zero))
30: intpr(sk_n,tup0,zero)
31: intpr(pred,tup1(zero),zero)
```

The above interpretation satisfies the axioms,
if instantiated by constructor terms with less than 3 constructors!

Summary:

the conjecture `succ( pred( N ) ) = N`
is violated by the following variable assignment:

```
N : zero
```

and by the following evaluation of conjecture subterms:

```
pred(N) : zero
succ(pred(N)) : succ(zero)
```

Ganz oben sieht man diejenigen Zeilen der MGTP-Ausgabe, die keine Platzhalter enthalten. Aus diesen kann man die vorgeschlagene Interpretation der Funktionen auf den (für die Erfüllbarkeit der instantiierten Spezifikation relevanten) Konstruktorterm-Tupeln ablesen.

Die Notiz darunter weist darauf hin, daß hiermit eine bestimmte, beschränkte Grundinstantiierung der Axiome nachgewiesenermaßen erfüllt wird durch die angegebene, *die Vermutung widerlegende Interpretation*. Dies gilt aber nicht unbedingt für die (uninstantiierte) Spezifikation! Diese Lücke muß der Benutzer/die Benutzerin mit seiner/ihrer Intuition füllen. Er/sie muß beurteilen, ob der vermeintliche Fehler wirklich ein Fehler ist, oder ob der Blick auf die Instanzen oberhalb der vorgegebenen Grenze zeigt, daß der Vorschlag des Systems zu relativieren ist. Daß dies notwendig sein kann, wird in Abschn. 5.2.2 demonstriert.

Die in der 'Summary:' zu findenden Informationen könnte man aus den darüber stehenden `intpr`-Atomen extrahieren (unter Umdeutung von `sk_n` zu `N`). Sie lassen sich aber viel einfacher berechnen. In der vollständigen MGTP-Ausgabe (s.o.) finden sich u.a. die Atome:
`is(val(sk_n),zero)`,
`is(val(succ(pred(sk_n))),succ(zero))` und
`is(val(pred(sk_n)),zero)`.
Aus diesen liest die Ausgabeaufbereitung unmittelbar die oben wiedergegebene Information aus und formuliert sie um in der 'Sprache' der Vermutung. Dieser für die Programmierung einer benutzerfreundlichen Ausgabe sehr nützliche 'Nebeneffekt' des Verfahrens verdeutlicht erneut das Zusammenspiel zwischen den `intpr`- und den `is`-Atomen in dem vorgestellten Ansatz.

Betrachten wir noch abschließend das Ergebnis. Die Anwendung der angegebenen Variablenzuweisung und der Untertermauswertungen auf die Vermutung zeigt sofort, daß die Gleichung hierdurch verletzt wird.

In Deutung dieses Ergebnisses kann sich der Benutzer/ die Benutzerin überlegen, daß die vom System vorgeschlagene Belegung der Variablen `N` mit `zero` *in jedem Fall* die vermutete Gleichung verletzt, sogar unabhängig von der Interpretation der Funktion `pred`. Im allgemeinen muß das nicht so sein.

Auswirkung der Optimierung

Am gleichen Beispiel, aber mit einer Verschiebung der Größenbeschränkung, demonstrieren wir noch den meßbaren Effekt der Optimierung der Literal-Transfor-

mation. Die MGTP-Ausgabe stellt entsprechende Zahlen[4] bereit. In den beiden zu vergleichenden Läufen ist die maximale Konstruktortermgröße jeweils auf 7 eingestellt. Unter Verwendung der Ergebnisse der symmetrischen Transformation (Def. 3.2.12) ergeben sich folgende Zahlen:

```
Number of models: 1
Number of failed branches: 21
Number of total branches: 22
Number of total atoms: 773
SAT
Proving time: 24459 msec
```

Dieser Lauf produziert einen echten Baum mit 22 Ästen, von denen alle außer dem letzten verworfen werden. Benutzt MGTP hingegen die Regeln, die sich aus der optimierten Variante der Transformation (Def. 3.2.10) ergeben, dann benötigt der entsprechende Lauf in diesem Beispiel ca. ein fünfhundertstel der Zeit, und der entstehende Baum besitzt nur einen Ast.

```
Number of models: 1
Number of failed branches: 0
Number of total branches: 1
Number of total atoms: 73
SAT
Proving time: 49 msec
```

Die push-pop-Vermutung

Die nächste Vermutung bezüglich der Spezifikation NatStack, die wir betrachten, ist nun eine Aussage φ über Stacks. Sie lautet:

$$\varphi \;=\; \mathsf{push}(\mathit{top}(\mathit{st}), \mathit{pop}(\mathit{st})) \doteq \mathit{st}$$

Natürlichsprachlich heißt dies: wenn das oberste Element eines Kellers auf den Rest des Kellers gelegt wird, dann ergibt dies den ursprünglichen Keller. Die Transformation der Vermutung ergibt die Regel:

```
%The Conjecture

%lemma{ ( push( top( ST ), pop( ST ) ) = ST )}

->
different(val(push(top(sk_st),pop(sk_st))),val(sk_st)),
```

[4] Rechner-Konfiguration für die Testläufe: Linux, 400 Mhz Pentium II, 196 MB Arbeitsspeicher.

```
is(val(push(top(sk_st),pop(sk_st))),
   push(val(top(sk_st)),val(pop(sk_st)))),
intpr(top,tup1(val(sk_st)),val(top(sk_st))),
search_nat(val(top(sk_st))),
intpr(sk_st,tup0,val(sk_st)),
search_stack(val(sk_st)),
intpr(pop,tup1(val(sk_st)),val(pop(sk_st))),
search_stack(val(pop(sk_st))) .
```

Der Aufruf von MGTP mit dieser Regel und den unveränderten Regeln aus der transformierten Spezifikation NatStack resultiert (bei der Obergrenze von drei Konstruktoren) in folgender Ausgabe.

Counter interpretation for the conjecture:
`push( top( ST ), pop( ST ) ) = ST`

```
17: intpr(top,tup1(push(zero,nil)),zero)
18: intpr(pop,tup1(push(zero,nil)),nil)
19: intpr(top,tup1(push(succ(zero),nil)),succ(zero))
20: intpr(pop,tup1(push(succ(zero),nil)),nil)
35: intpr(sk_st,tup0,nil)
37: intpr(top,tup1(nil),zero)
45: intpr(pop,tup1(nil),nil)
```

The above interpretation satisfies the axioms,
if instantiated by constructor terms with less than 3 constructors!

Summary

the conjecture `push( top( ST ), pop( ST ) ) = ST`
is violated by the following variable assignment:

```
ST : nil
```

and by the following evaluation of conjecture subterms:

```
top(ST) : zero
pop(ST) : nil
push(top(ST),pop(ST)) : push(zero,nil)
```

Wieder kann man durch Einsetzen der Werte in die vermutete Gleichung sehen, warum diese verletzt wird. Vielleicht hat der Leser/die Leserin schon bei der Vorstellung der Vermutung bemerkt, daß diese von einem ähnlichen Typus ist wie die succ-pred-Vermutung. Auch hier reicht schon die vorgeschlagene Belegung der Variablen aus, um die Gleichung zu invalidieren. In den noch folgenden Beispielen ist dies nicht mehr so!

Die del-top-Vermutung

Diesmal betrachten wir eine Vermutung φ über NatStack, bei deren Analyse man nicht mehr um die Interpretation der Funktionen herumkommt. Sie lautet:

$$\varphi \;=\; del(top(st), st) \doteq pop(st)$$

Diese Formel besagt, daß das Löschen des obersten Elementes aus einem Stack mittels *del* der Entfernung des obersten Elementes mittels *pop* gleichkommt. Der Leser/die Leserin sei ermuntert, kurz innezuhalten, um φ und die Spezifikation NatStack (S. 164) genauer zu betrachten. Gilt NatStack $\models \varphi$ oder NatStack $\not\models \varphi$? Die Behandlung des Problems als Beispiel in dieser Abhandlung läßt Letzteres vermuten. Aber warum könnte NatStack $\not\models \varphi$ gelten?

Die Anwendung unseres Verfahrens resultiert in folgender Ausgabe (wir geben nur die 'summary' wieder):

Summary

the conjecture `del( top( ST ), ST ) = pop( ST )`
is violated by the following variable assignment:

```
ST : nil
```

and by the following evaluation of conjecture subterms:

```
del(top(ST),ST) : nil
top(ST) : zero
pop(ST) : push(zero,nil)
```

Der möglicherweise überraschende Teil der Antwort ist die Kombination von `pop(ST) : push(zero,nil)` und `ST : nil`. Setzt man die zweite in die erste ein, erhält man `pop(nil) : push(zero,nil)`. Diese Auswertung ist konsistent zu den Axiomen, da dort die Interpretation von *pop* auf nil, $\mathcal{I}(pop)(\mathsf{nil})$, nicht spezifiziert ist! Diesmal reicht die erste von dem Verfahren betrachtete Möglichkeit, $\mathcal{I}(pop)(\mathsf{nil}) = \mathsf{nil}$ zu setzen,[5] nicht aus, um die Vermutung zu widerlegen. Darum versucht das Verfahren die zweite Möglichkeit, $\mathcal{I}(pop)(\mathsf{nil}) = \mathsf{push}(\mathsf{zero}, \mathsf{nil})$, und damit gelingt die Invalidierung der Vermutung. Diese Interpretation von *pop*, nach der die Entfernung des obersten Elementes eines leeren Kellers gleichbedeutend ist mit der Einfügung einer Null, ist gemäß der Intention der Spezifikation sicherlich abwegig. In dieser Situation gibt es zwei Möglichkeiten der Verbesserung: entweder wir verstärken die Spezifikation, um die Unterspezifikation von *pop* zu beheben, oder wir schwächen die Vermutung ab, etwa durch: $\varphi' = st \not\doteq \mathsf{nil} \rightarrow \varphi$.

[5] Vgl. die erzeugte `search_stack`-Regel

del unterspezifiziert

Nun bauen wir künstlich einen Fehler in die Spezifikation NatStack ein. NatStack′ gehe aus NatStack hervor durch Streichung des Axioms $del(n, \mathsf{nil}) \doteq \mathsf{nil}$. Damit sind jetzt sämtliche Funktionen der Spezifikation unterspezifiziert. Bei den anderen Funktionen kann dies durchaus beabsichtigt sein, da dort nur der jeweilige Randfall (zero bzw. nil) betroffen ist. Bei *del* aber ist nun der *Basisfall* der rekursiven Spezifikation unterspezifiziert, womit die Interpretation auf keinem Argumenttupel mehr festgelegt ist. Dies wird sich typischerweise bemerkbar machen bei dem scheiternden Versuch, eine Aussage über *del* induktiv zu beweisen. Der Beweisversuch könnte sogar gerade das gestrichene Axiom als ungelöstes Unterziel besitzen. Auf dieses ungelöste Unterziel $del(n, \mathsf{nil}) \doteq \mathsf{nil}$ kann man dann unsere Methode anwenden. Die Ausgabe lautet:

Summary

the conjecture `del( N, nil ) = nil`
is violated by the following variable assignment:

```
N : zero
```

and by the following evaluation of conjecture subterms:

```
del(N,nil) : push(zero,nil)
```

Wenn man wieder beide Aussagen zusammensetzt, erhält man
`del(zero,nil) : push(zero,nil)`,
was in zweifacher Hinsicht unsinnig ist: der Stack ist durch die Löschung eines Elementes größer geworden, und er enthält insbesondere das gelöschte Element.

5.2.2 p_Forever

Nun wenden wir uns einem Beispiel zu, in dem die getroffene Vermutung zutrifft, von unserer Methode aber invalidiert wird, aufgrund der beschränkten Instantiierung. Die Spezifikation p_Forever in Abb. 5.2, die wir schon in Abschn. 2.3.2 behandelt hatten, spezifiziert eine boolsche Funktion p, die auf jeder natürlichen Zahl ‘gilt’.

Wir untersuchen die – absolut zutreffende – Vermutung

$$\varphi \;=\; p(y) \doteq \mathsf{tt}$$

Bei einer Größenbeschränkung von zwei Konstruktoren lautet die Antwort des Systems:

Counter interpretation for the conjecture:
`p( Y ) = tt`

$$
\begin{array}{lll}
\textbf{spec} = \textsf{p_Forever} & & \\
\quad \textbf{sorts} & \mathit{Nat} ::= \textsf{zero} \mid \textsf{succ}(\mathit{Nat}); & \\
 & \mathit{Bool} ::= \textsf{tt} \mid \textsf{ff}; & \\
\quad \textbf{functions} & p: \ \mathit{Nat} \to \mathit{Bool}; & \\
\quad \textbf{vars} & x: \ \mathit{Nat}; & \\
\quad \textbf{axioms} & p(\textsf{zero}) \doteq \textsf{tt}; & \\
 & p(x) \doteq \textsf{tt} \to p(\textsf{succ}(x)) \doteq \textsf{tt}; & \\
\textbf{end} & &
\end{array}
$$

Abbildung 5.2: Spezifikation p_Forever

```
5:  intpr(p,tup1(zero),tt).
31: intpr(p,tup1(succ(zero)),tt).
32: intpr(p,tup1(succ(succ(zero))),tt).
96: intpr(sk_y,tup0,succ(succ(succ(zero)))).
97: intpr(p,tup1(succ(succ(succ(zero)))),ff).
```

The above interpretation satisfies the axioms,
if instantiated by constructor terms with less than 3 constructors!

Summary:

the conjecture `p( Y ) = tt`
is violated by the following variable assignment:

```
Y : succ(succ(succ(zero)))
```

and by the following evaluation of conjecture subterms:

```
p(Y) : ff
```

Durch die Beschränkung auf höchstens zwei Konstruktoren ist p nur auf Argumenten kleiner gleich succ(succ(zero)) festgelegt. Die Interpretation von `sk_y` muß nur weit genug 'ausweichen', um die negierte Vermutung zu erfüllen. Daran ändert sich natürlich nichts, wenn die Obergrenze angehoben wird. Bei einem Lauf mit der Obergrenze drei lautet die Ausgabe
`Y : succ(succ(succ(succ(zero))))`,
bei vier entsprechend
`Y : succ(succ(succ(succ(succ(zero)))))`,
usw.

Dies ist ein Fall, in dem die Antwort „the conjecture is violated by ...“ irreführend ist, und die darüberstehende Warnung „... satisfies the axioms, if instantiated by constructor terms with less than n constructors!“ ernstgenommen werden muß.

Für den Benutzer sollte es ein Warnzeichen sein, daß die vorgeschlagene, die Vermutung widerlegende Interpretation bzw. Variablenbelegung sich mit der Größe betrachteter Instanzen mitändert!

5.2.3 MergeSort

Nun wenden wir uns, wie angekündigt, einer fehlerhaften Spezifikation zu, die nicht zum Zweck der Demonstration unserer Methode entstanden ist. Sie findet sich in einem an der Universität Ulm erschienenen Technischen Bericht mit dem Titel „Fehlersuche in Formalen Spezifikationen“ [RST00]. Aus diesem Titel darf man aber keine voreiligen Schlüsse in Bezug auf den nun zu demonstrierenden Fehler ziehen. Er wird dort in keiner Weise behandelt und bleibt verborgen, obwohl die Spezifikation häufig wiederkehrt. Daß wir den Fehler im folgenden behandeln, soll aber niemanden bloßstellen. Ein unbefangener, offener Umgang mit Fehlern ist ein wichtiger Faktor bei der Entwicklung von Methoden zur Fehler-Entdeckung.

In Abb. 5.3 ist die ADT-Spezifikation MergeSort wiedergegeben, die einen Algorithmus zur Sortierung einer Liste natürlicher Zahlen spezifiziert. *less* ist die transitive, irreflexive Hülle der Nachfolger-Relation. *app* (append) hängt zwei Listen aneinander. Mittels *merge* werden zwei Listen verschmolzen in der Weise, daß in jedem Schritt das kleinere der vordersten Elemente in die neue Liste eingefügt wird (wobei bei Gleichheit der vordersten Elemente die zweite Liste bevorzugt wird). *sort* schließlich zerteilt eine Liste an einer beliebigen Stelle in zwei Listen und sortiert diese getrennt, um sie danach mittels *merge* wieder zusammenzufügen.

(Die Originalspezifikation [RST00, Abb 2.2] ist generisch. Die Listen haben dort nicht, wie hier, einen konkreten Elementtyp, sondern sind über dem Elementtyp parametrisiert. Entsprechend sind die Axiome für *less* ausgelagert. Da wir hier keine Parametrisierung betrachten, verwenden wir in MergeSort als Beispiel für eine geordnete Sorte die natürlichen Zahlen. Außerdem sind in MergeSort ein paar Funktionen aus dem Original weggelassen, die hier nicht von Interesse sind.)

Auch hier sei der Leser/die Leserin herzlich eingeladen, das schon angekündigte Problem in MergeSort vor dem Weiterlesen selbst zu entdecken.

Betrachten wir nun die folgende Vermutung φ über diese Spezifikation:

$$\varphi \;=\; sort(\mathsf{cons}(n, \mathsf{empty})) \doteq \mathsf{cons}(n, \mathsf{empty})$$

sort soll die einelementige Liste nicht verändern. Dies scheint plausibel. Wir wenden nun das beschriebene Verfahren auf φ an. Die Transformation von MergeSort

spec = MergeSort

sorts Nat ::= zero | succ(Nat);
$List$ ::= empty | cons(Nat, $List$);
$Bool$ ::= tt | ff;

functions $less: \ Nat \times Nat \to Bool$;
$app: \ List \times List \to List$;
$merge: \ List \times List \to List$;
$sort: \ List \to List$;

vars $n, n', n'': \ Nat; \ l, l': \ List$;

axioms
$less(n, \mathsf{succ}(n)) \doteq \mathsf{tt}$;
$less(n, n') \doteq \mathsf{tt} \wedge less(n', n'') \doteq \mathsf{tt} \to less(n, n'') \doteq \mathsf{tt}$;
$less(n, n) \doteq \mathsf{ff}$;
$app(\mathsf{empty}, l) \doteq l$;
$app(\mathsf{cons}(n, l), l') \doteq \mathsf{cons}(n, app(l, l'))$;
$merge(\mathsf{empty}, l') \doteq l'$;
$merge(l, \mathsf{empty}) \doteq l$;
$less(n, n') \doteq \mathsf{tt} \to$
$\quad merge(\mathsf{cons}(n, l), \mathsf{cons}(n', l'))$
$\quad \doteq \mathsf{cons}(n, merge(l, \mathsf{cons}(n', l')))$;
$less(n, n') \doteq \mathsf{ff} \to$
$\quad merge(\mathsf{cons}(n, l), \mathsf{cons}(n', l'))$
$\quad \doteq \mathsf{cons}(n', merge(\mathsf{cons}(n, l), l'))$;
$sort(\mathsf{empty}) \doteq \mathsf{empty}$;
$sort(app(l, l')) \doteq merge(sort(l), sort(l'))$;

end

Abbildung 5.3: Spezifikation MergeSort

soll hier nicht wiedergegeben werden; sie besteht aus fünfzig Regeln, die teilweise recht länglich sind, da einige Axiome mehrere Vorkommen von (zu interpretierenden) Funktionen besitzen. Stattdessen folgt hier lediglich die Transformation der Vermutung:

```
%The Conjecture

%lemma{ ( sort( cons( N, empty ) ) = cons( N, empty ) )}

-> different(val(sort(cons(sk_n,empty))),val(cons(sk_n,empty))),
        intpr(sort,tup1(val(cons(sk_n,empty))),
              val(sort(cons(sk_n,empty)))),
        search_list(val(sort(cons(sk_n,empty)))),
        is(val(cons(sk_n,empty)),cons(val(sk_n),empty)),
        intpr(sk_n,tup0,val(sk_n)),
        search_nat(val(sk_n)) .
```

Entsprechend der größeren Regelmenge ist auch die Ausführung der Regeln aufwendiger als bei den letzen Beispielen. Darüberhinaus kann hier kein Axiom 'ausgeblendet' werden (ob von Hand oder automatisch), da die in der Vermutung benutzte Funktion *sort* von allen Axiomen abhängt. In der Folge kann MGTP diesmal nur mit einer sehr kleinen Obergrenze von Instanzen für die Variablen der Axiome zurechtkommen.

In diesem Zusammenhang sollte aber beachtet werden, und dies gilt ganz allgemein, daß die Größenbeschränkung nur die *Instantiierung der Variablen* betrifft. In den so instantiierten *Axiomen* kommen auch größere Terme vor. Wenn z.B. ein Axiom den Unterterm $\mathsf{cons}(n, l)$ besitzt und die Variablen mit Konstruktortermen instantiiert werden, die aus höchstens drei Konstruktoren bestehen, dann bestehen die Instanzen von $\mathsf{cons}(n, l)$ aus bis zu sieben Konstruktoren. Es sei auch noch einmal darauf hingewiesen, daß für Konstruktorterme, nach denen MGTP durch Abarbeitung der Regeln sucht, keine Größenbeschränkung existiert.

Bei einer Beschränkung auf Instanzen der Maximalgröße zwei gibt das Verfahren die folgende Ausgabe zurück:

Summary

the conjecture
`sort( cons( N, empty ) ) = cons( N, empty )`
is violated by the following variable assignment:

```
N : zero
```

and by the following evaluation of conjecture subterms:

```
sort(cons(N,empty)) : empty
cons(N,empty) : cons(zero,empty)
```

Die Überraschung (wenn sie denn eine ist) liegt in der Zeile:
`sort(cons(N,empty)) : empty`.
Tatsächlich ist in MergeSort überhaupt nicht spezifiziert, wie die einelementige Liste zu sortieren ist. Diese läßt sich nämlich nur zerteilen in sich selbst und die leere Liste; beide gilt es dann zu sortieren. Das Sortierproblem wird somit nur auf sich selbst zurückgeführt. Die Konsequenzen sind fatal. Selbst wenn niemand je eine einelementige Liste sortieren will, so wird doch die Sortierung *jeder* Liste (außer der leeren) durch die Axiome zurückgefürt auf die Sortierung der einelementigen. Auch wenn die Vermutung φ nicht explizit aufgestellt würde, so würden doch andere, sinnvollere Aussagen (beispielsweise über die Invarianz der Länge gegenüber *sort*) auf diese oder eine ähnliche zurückgeführt werden müssen.

Das Interessante an dieser Spezifikation ist, daß sie fehlerhaft ist, *weil* sie (teilweise) einem gängigen Muster folgt. Normalerweise benötigen rekursive Definitionen gerade *einen* Basisfall. Hier aber wären *zwei* Basisfälle vonnöten gewesen, obwohl die Sorte *List* nur einen Basiskonstruktor besitzt.

Auch in diesem Beispiel gilt, daß das Problem durch die *alleinige* Suche nach Variablenbelegungen nicht aufgedeckt werden würde. Jedoch durch die Suche nach einer Interpretation war es möglich, eine Termauswertung zu bestimmen, welche der Spezifizierer nicht im Sinn hatte und die eine grundlegende intendierte Eigenschaft widerlegt.

6 Verwandte Arbeiten

In diesem Kapitel stellen wir den Zusammenhang her zwischen dem vorliegenden Ansatz und anderen Arbeiten, die sich ähnlichen Fragestellungen widmen. Die jeweils unterschiedlichen Berührungspunkte lassen sich hierbei den folgenden zwei Themen zuordnen:

- Konstruktion und Repräsentation von Modellen
- Widerlegung bzw. Verbesserung falscher Aussagen

In der folgenden Diskussion werden die näher verwandten Ansätze in aller Kürze erläutert und mit dem Hiesigen verglichen. Bei den weiter entfernt liegenden Ansätzen beschränken wir uns darauf, den (möglicherweise großen) Unterschied zu begründen und einige Hauptvertreter zu benennen.

6.1 Konstruktion u. Repräsentation von Modellen

Hier betrachten wir Ansätze zur Konstruktion und Repräsentation von Modellen für Formeln erststufiger Logik, obwohl der semantische Rahmen der termerzeugten Datentypen über die Ausdrucksmächtigkeit erststufiger Logik hinausgeht (vgl. Kapitel 2). Der semantische Unterschied besteht aber darin, daß wir uns gegenüber reiner erststufiger Logik auf *bestimmte* Modelle beschränken, unter Berücksichtigung der Konstruktoren. Dies hat zwar enorme Konsequenzen für den Folgerbarkeitsbegriff (z.B. sind induktiv beweisbare Aussagen gültig), ändert aber nichts daran, daß wir auch hier *Modelle erststufiger Formeln* betrachten (bestimmte eben). Daher werfen wir im folgenden einen Blick auf andere Verfahren zur Konstruktion erststufiger Modelle.

Beinahe alle Arbeiten auf dem Gebiet der automatischen Modell-Konstruktion stammen aus den letzten zehn Jahren. Wir hatten schon in der Einleitung zu Kapitel 3 erwähnt, daß sich die Ansätze einteilen lassen in *semantische* und *syntaktische* Methoden ([Lei00]).

Semantische Methoden basieren auf einer expliziten Konstruktion von Interpretationen. Beschränkt man sich auf endliche Modelle fester Größe, dann kann man (im Prinzip) in endlicher Zeit alle möglichen Interpretationen konstruieren und auf Erfüllung der Axiome überprüfen. Systeme, die nach diesem Grundprinzip arbeiten, sind FINDER [Sla94] und SEM [ZZ96]. Da man jeden endlichen Grundbereich mit einem Anfangsstück der natürlichen Zahlen identifizieren kann, wird bei diesen Systemen jeder Sorte in der Signatur ihre Größe mitgegeben, so z.B. (in SEM-Syntax):

```
( pigeon [30] )
( hole [29] )
```

Für die Funktionen zwischen diesen Bereichen, z.B. für `h : pigeon -> hole`, werden Interpretationen gesucht, hier $\mathcal{I}(\mathtt{h}) : \{1, \ldots, 30\} \rightarrow \{1, \ldots, 29\}$.

Auch der in dieser Abhandlung vorgestellte Ansatz gehört zu den semantischen Methoden in dem Sinne, daß explizit Interpretationen konstruiert werden. Auch bei uns sind die Bereiche durch die Signatur vorgegeben, allerdings in einer viel strukturierteren Weise, denn sie können miteinander verschränkt sein. Das für Datenstrukturen so typische Phänomen, daß gewisse Bereiche zum Aufbau anderer Bereiche beitragen, ist nur mit Sortenobergrenzen nicht darstellbar. Ein weiterer, damit zusammenhängender Unterschied ist, daß bei uns die Bereiche in der Regel unendlich sind. Darum können wir auch die Interpretationen nicht erschöpfend beschreiben. Dies ist der Preis für unsere Ausrichtung auf Datentypen.

Auch bei dem in dieser Abhandlung beschriebenen Verfahren spielen Obergrenzen eine Rolle. Darum sollten wir kurz klären, was diese Obergrenzen mit den Sortengrenzen für endliche Bereiche (wie '`pigeon [30]`') zu tun haben und ob man FINDER bzw. SEM auch für unsere Zwecke einsetzen könnte. Nach dem in Kapitel 3 beschriebenen Vorgehen wird (unabhängig von der Signatur) eine Obergrenze vorgegeben für die Größe von Termen, mit denen die Axiome instantiiert werden. Dadurch entstehen – auf einem Ast – nur beschränkt viele I-Atome. Diese enthalten anfangs noch Platzhalter, nach deren Ersetzung gesucht werden muß. Diese Suche ist aber *nicht* durch besagte Grenze beschränkt, einzusetzende Werte können theoretisch beliebig 'groß' sein. Nur hierdurch wird garantiert, daß bei Anhebung der Obergrenze höchstens Modelle wegfallen, aber keine neuen hinzukommen. Für die Fehlersuche bedeutet dies, daß höchstens zu viele, nie aber zu wenige Fehler gefunden werden. Diese unterschiedliche Behandlung von Instanzen (für Axiomen-Variablen) und Werten (der Interpretation) läßt sich mit Systemen wie FINDER oder SEM nicht erreichen.

Syntaktische Methoden zeichnen sich erstens dadurch aus, daß klassische Beweiskalküle Verwendung finden (bzw. Varianten davon). Zweitens wird das gesuchte Modell durch endlich viele Formeln in der (evtl. erweiterten) Syntax der

Objektlogik repräsentiert.[1] Das Modell selbst (nicht unbedingt seine Repräsentation) wird aufgefaßt als die Menge aller Grund-Atome, die darin gelten.

Die Ansätze von T. Tammet [Tam91] sowie von C. Fermüller und A. Leitsch [FL96] basieren auf gescheiterten terminierten Resolutions-Beweisen ('gescheitert' ist dann der Nachweis der Unerfüllbarkeit einer Klausel-Menge). Während Tammets Methode endliche Mengen von Grund-Atomen erzeugt, ist das Ergebnis der Methode von Fermüller und Leitsch eine endliche Menge von variablen-behafteten Atomen, deren i.a. unendliche Instantiierung das Modell darstellt.

Ein auf R. Caferra und N. Zabel [CZ92] zurückgehender Ansatz namens RAMC (Refutation And Model Construction) basiert auch auf Resolution, aber erweitert den Kalkül ganz wesentlich. Statt auf reinen Klauseln arbeiten die Regeln auf mit Gleichungs-Constraints annotierten Klauseln. Diese Gleichungen sind jedoch nicht Teil der Objekt-Logik, sondern dienen der Repräsentation von Modellen. Die Gleichungs-Constraints werden *gleichzeitig* mit den Klauseln deduziert. Am Ende des scheiternden Beweises kann aus ihnen das Modell der dem Beweis zugrundeliegenden Objekt-Formel abgelesen werden. Dieser Ansatz wurde in etlichen Publikationen der beiden genannten Autoren sowie N. Peltier weiterentwickelt, s. [CP00] für eine Übersicht. U.a. wurde er auch auf Tableaux übertragen [CZ93, Pel97b].

Eine andere Möglichkeit, die Syntax der Objektlogik für Zwecke der Modell-Repräsentation zu erweitern, besteht in der Verwendung von Term-Exponenten. Mit diesen lassen sich Modelle wie z.B. $\{even(succ^{2n}(x))|n \in \mathbb{N}\}$ darstellen. Zu nennen sind hier die Arbeiten von S. Klingenbeck [Kli97] und N. Peltier [Pel97a].

Ein weiterer Vertreter syntaktischer Methoden ist die Familie der Hyper-Tableaux, welche zurückgehen auf F. Brown [Bro78]. Eine spezielle Ausprägung von Hyper-Tableaux, genannt *model generation*, wurde realisiert in den Systemen SATCHMO (von R. Manthey und F. Bry [MB88]) und MGTP (von H. Fujita und R. Hasegawa [FH91, HFK97]). In Abschn. 3.4.2, speziell Abb. 3.3, haben wir dieses Verfahren ausführlich eingeführt. Dieses Grundkonzept wurde von P. Baumgartner, U. Furbach und I. Niemelä verfeinert [BFN96], vor allen Dingen durch die Einführung von 'freien Variablen' und voller Unifikation.

Obwohl das in dieser Abhandlung beschriebene Verfahren die MGTP-Implementierung von model generation verwendet, arbeiten wir zunächst einmal die Unterschiede heraus zwischen unserem Ansatz und den obigen syntaktischen Methoden *einschließlich* model generation. Die oben genannten Methoden dienen überwiegend der Modellbildung für *gleichungsfreie* Formeln erststufiger Logik. Dies gilt auch für RAMC, weil dort nicht Modelle *für* Gleichungen gesucht, sondern Modelle (gleichungsfreier Formeln) *durch* Gleichungen repäsentiert werden. Dagegen sind unsere Spezifikationen ganz und gar gleichungsbasiert. Desweiteren beschäftigen sich alle genannten Ansätze mit reiner erststufiger Logik, und

[1] Diesbezüglich spielt der RAMC-Ansatz eine Sonderrolle, s.u.

sie konstruieren *Herbrandmodelle* bzw. deren Repräsentation. In die Sprache der Datentypen übersetzt heißt dies, daß die Modelle frei termerzeugt sind über einer Signatur, in der *alle* Funktionen und Konstanten als Konstruktoren aufgefaßt werden, sogar die Skolemsymbole.[2] Diese Modelle aber können wir nicht als Zeugen für die in Def. 2.3.32 definierte Erfüllbarkeit von Spezifikationen verwenden. Aus beiden Gründen können die sehr ausgereiften Techniken und Verfeinerungen der syntaktischen Methoden nicht ad hoc angewandt werden. Dennoch gibt es hier sicherlich einiges Potential, unseren noch jungen Ansatz im Lichte der obigen Ansätze weiter zu verfeinern.

Trotz dieser Abgrenzung muß man auch sehen, daß wir uns ganz wesentlich auf eine dieser Methoden, nämlich model generation (MGTP) gestützt haben. Insofern ist unsere Methode eine *deduktive* (vgl. den Titel der Arbeit). Allerdings haben wir vorher die Ebene gewechselt. Auf der Ebene der Regeln, die das Ergebnis der Transformation darstellen, gibt es keine Gleichungen mehr, und die Atome, die für uns die Rolle von Meta-Atomen einnehmen, werden mit einer ganz anderen 'Signatur' gebildet als die Gleichungsatome in den Axiomen. Der oben beschriebenen Sichtweise von model generation bzw. Hyper-Tableaux käme es näher, wenn wir unsere Regeln als Formeln auffaßten und MGTP nach der Erfüllbarkeit dieser Formeln fragen würden. Um aber nicht noch eine gewöhnliche, zweite Formelsemantik auf der Meta-Ebene einführen zu müssen, hatten wir die MGTP-Eingabe-Klauseln als Regeln aufgefaßt, denen die Prozedur ***MG-proc*** eher eine operationale denn eine deklarative Semantik gibt.

6.2 Widerlegung u. Korrektur falscher Aussagen

Was die generelle Thematik der Widerlegung falscher Aussagen angeht, so gehören die im letzten Abschnitt genannten Ansätze allesamt dazu, da man die Frage der Nicht-Folgerbarkeit immer auf ein Konsistenzproblem zurückführen kann. In diesem Abschnitt soll es nun eher um den Umgang mit falschen Aussagen im Umfeld von Datentypen (im weitesten Sinne) gehen.

Nur kurz, weil wenig verwandt, erwähnen wir die Arbeiten aus dem Gebiet der Termersetzungssysteme. Die dort zugrundeliegende sog. 'Rewrite-Semantik' hat mehr Gemeinsamkeiten mit initialer Semantik als mit unserer losen Semantik. Sie ist i.a. nicht konstruktorbasiert, und sie ist *monomorph*. Darum fallen Konsistenz und Folgerbarkeit gewissermaßen zusammen, so daß sich hier ganz andere Möglichkeiten bieten, Nichtfolgerbarkeit zu überprüfen. In diesem Rahmen gibt es viele Arbeiten zum Thema 'Beweis durch Konsistenz', z.B. L. Bachmair [Bac88]. Einen Überblick über den Stand dieses Gebietes gibt [Com01]. Dort wird auch

[2] Vgl. die Diskussion zur initialen Semantik in Abschn. 2.6.3.

der Zusammenhang zwischen 'Beweis durch Konsistenz' und 'induktionsloser Induktion' hergestellt.

Von der *Fragestellung* her näher verwandt mit unserem Ansatz sind Arbeiten aus dem Bereich der Induktionsautomatisierung. Dort entspringt das Interesse an der Behandlung falscher Aussagen vor allen Dingen der automatischen Erzeugung derselben durch Übergeneralisierung. Zwar wird auch in diesem Rahmen gewöhnlich Monomorphie garantiert, aber durch syntaktische Beschränkung auf *konstruktive Spezifikationen* (vgl. die Diskussion auf S. 63). Semantisch aber handelt es sich um konstruktorbasierte Spezifikationen, weswegen auch ähnliche Beispiele betrachtet werden, Zahlen, Listen usw. Aus diesem Umfeld gibt es eine Arbeit von M. Protzen [Pro92], in der ein automatisches Verfahren vorgestellt wird zum Nachweis von Nichtfolgerbarkeit. Da aber Monomorphie garantiert ist, ist $\not\models \varphi$ äquivalent zu $\models \neg Cl_\exists(\varphi)$. Zum Nachweis letzterer Aussage reicht es aus, Variablenbelegungen zu finden, so daß das instantiierte φ falsch wird. Andere Arbeiten aus dem Gebiet der Induktionsautomatisierung beschäftigen sich mit der *Korrektur* falscher Aussagen. Zu nennen sind hier R. Monroy, A. Bundy und A. Ireland [MBI94], die dieses Thema mit Methoden der 'Beweisplanung' bearbeiten, sowie wieder Protzen [Pro96]. Grob gesagt geht es darum, einer Aussage φ, deren Beweis nicht gelingt, eine Bedingung ψ voranzustellen, so daß der Beweis von $\psi \to \varphi$ gelingt. In beiden Ansätze werden die vielfältigen Sonder-Informationen, die der Induktionsbeweiser neben den reinen Formeln verwalten muß, zur Konstruktion von ψ ausgenutzt.

Vom syntaktischen und semantischen Rahmen her noch näher an unserer Fragestellung ist ein Ansatz von A. Thums, W. Reif und G. Schellhorn [Thu98, RST00, RST01]. Auch hier geht um das Auffinden von Fehlern in termerzeugten Datentypen, und diesmal auch unter loser Semantik. Darüberhinaus ist der Rahmen sogar noch allgemeiner als unserer, da außer den frei erzeugten auch nicht frei erzeugte Datentypen betrachtet werden. Dennoch beschränkt sich dieser Ansatz völlig auf die Suche nach Variablenbelegungen. Bei der Suche nach Belegungen der Variablen entstehen Bedingungen für die Gültigkeit der negierten Formel. Die Konsistenz dieser Bedingungen wird weder vom System überprüft, noch unterstützt es (etwa in interaktiver Weise) die Untersuchung der Konsistenz. Diesem und unserem Ansatz ist gemein, daß die Nicht-Folgerbarkeit nicht abschließend bewiesen wird. Bei beiden Ansätzen muß letztlich der Benutzer/die Benutzerin entscheiden über die Konsistenz der Antwort des Systems. Bei unserem Ansatz wird aber genau dieser Schritt unterstützt durch die Konstruktion der Interpretationen und Evaluationen. Gerade dadurch ist es möglich, wie in Abschn. 5.2.3 dokumentiert, den unbeabsichtigten Fehler in der Spezifikation MergeSort in [RST00] aufzudecken, der durch reine Variablenbelegungen nicht zu identifizieren ist.

7 Zusammenfassung und Ausblick

In dieser Abhandlung wurde ein Ansatz vorgestellt, der geeignet ist, fehlerhafte Aussagen über freie abstrakte Datentypen als solche zu identifizieren. Die Methode besteht im Kern aus der Suche nach einem Modell der entsprechenden Gegenspezifikation. Die Modellsuche ist ganz und gar zugeschnitten auf die speziellen semantischen Gegebenheiten frei erzeugter Datentypen. Der Nachweis der Identifizierbarkeit frei erzeugter Datentypen mit Konstruktorterm-Algebren rechtfertigt die Fixierung des Grundbereiches in Form der Menge der Konstruktorterme, so daß die Modellsuche einzig und allein aus der Suche nach der *Interpretation* der Funktionen besteht. Zur Steuerung der Konstruktion von Interpretationen wurde das Schema eines theoriespezifischen Kalküls entwickelt. Dessen Regeln werden erzeugt durch eine Transformation der Axiome, der Signatur, der angezweifelten Vermutung sowie einer vorgegebenen Maximalgröße für Variablen-Instantiierungen. Die resultierenden Regeln werden 'ausgeführt' durch eine in der Literatur als 'model generation' bekannte Prozedur. Für die Realisierung des Ansatzes wurde eine in Japan entwickelte Implementierung dieser Prozedur eingesetzt. Falls ein Modell, d.h. eine erfüllende Interpretation gefunden wird, so wird dieses aufbereitet im Hinblick auf die Deutung der ursprünglichen Vermutung.

Aufgrund der Beschränkung der Variableninstantiierung kann das Verfahren allein die Konsistenz der Gegenspezifikation nicht abschließend sicherstellen. Entsprechend kann auch die Nichtfolgerbarkeit einer Vermutung nicht abschließend garantiert werden. Der Benutzer/die Benutzerin muß entscheiden, ob die aufbereitete Ausgabe tatsächlich auf einen Fehler hinweist oder die vorgeschlagene Interpretation lediglich der beschränkten Instantiierung zuzuschreiben ist. Im Falle, daß er/sie sich nicht sicher ist, kann die Maximalgröße für Instanzen angehoben und die Ausführung der Regeln wiederholt werden. Ändern sich die die Vermutung invalidierenden Variablenbelegungen und Termauswertungen hierdurch nicht, dann spricht dies sehr für die Korrektheit der Antwort, d.h. für die Nichtfolgerbarkeit der Vermutung. Ändern sich aber diese Werte mit der Instanzengröße, dann sollte man die Folgerbarkeit der Vermutung wieder ernsthaft in Erwägung ziehen.

Wie man in den Beispielen gesehen hat, kann das Verfahren nur mit recht kleinen Maximalgrößen arbeiten, aufgrund der kombinatorischen Explosion der Anzahl möglicher Interpretationen. Dies ändert aber nichts daran, daß Gegenmodelle

gefunden werden und tatsächliche Fehler offen zu Tage treten. In dem Beispiel p_Forever, in dem das System einen vermeintlichen Fehler zu finden glaubt, der aber keiner ist, würde die Beherrschung größer Instanzenmengen auch nichts helfen, da die vorgeschlagene Belegung der Variablen Y jeder endlichen Instanzenmenge „ausweichen“ würde.

Die Pragmatik, daß alle interessanten Effekte entweder schon in einem sehr kleinen Bereich oder erst beim Übergang vom Endlichen ins Unendliche auftreten, wäre ein lohnenswertes Thema für weitere theoretische Untersuchungen. Würde man allerdings z.B. Zahlentheorie mit abstrakten Datentypen formalisieren, was man tun kann, dann träfe diese Pragmatik nicht mehr zu. Aber bei den in der Informatik verbreiteten Datenstrukturen wie Listen, Bäumen, Arrays, Hashtabellen usw., jeweils mitsamt den interessierenden Operationen darauf, scheint es sich so zu verhalten.

Bei der Beschreibung der Realisierung hatten wir schon hingewiesen auf die angestrebte Einbettung des Verfahrens in den Kontext größerer Systeme, in denen Beweisverpflichtungen über Datentypen anfallen. Diese könnten dann sowohl mit klassischen, ‘optimistischen’ Beweisverfahren als auch mit dem vorgestellten, eher pessimistischen Ansatz bearbeitet werden. Dabei sollte beides nicht nebeneinander stehen, sondern eng verzahnt werden. Insbesondere in einem System, in dem Induktionsbeweise über generierte Datentypen (auch) interaktiv geführt werden, läßt sich die Methode gut einbinden, da die abschließende Bewertung der Antwort Teil der Interaktion sein kann.

Aus der Verwendung in Fallstudien können sich noch einige Veränderungen der bisher getroffenen Festlegungen ergeben. Z.B. sind parametrisierte Sorten noch nicht berücksichtigt. Den Parametersorten fehlen die für das Verfahren so zentralen Konstruktoren. Im Prinzip entspricht es dem Geist der Modellgenerierung, den Parameter im Zuge der Methode zu aktualisieren. Für die Aktualisierung scheinen die natürlichen Zahlen ein geeigneter Kandidat zu sein. Sie stellen den einfachst möglichen unendlichen Datentyp dar. Dies kann zwar theoretisch zu Problemen führen, wenn nämlich die Axiome des Parameters nur endliche Modelle zulassen. Dies wird aber zu selten der Fall sein, um den Nutzen der Verwendung von Parametern aufzuwiegen.

Nicht zuletzt handelt es sich bei dem vorgestellten Ansatz um eine grundlagenorientierte Arbeit, auf die, so hofft der Autor, aufgebaut werden kann. Die spezielle Lösung etwa des Instanzenproblems wurde völlig modular gehalten von den Grundprinzipien der Modell-Generierung für (freie) termerzeugte Datentypen. Die semantik-orientierte Methodik, die sich festmacht an dem Zusammenspiel zwischen Interpretations-Atomen I(), ‘noch unausgewerteten’ Auswertungs-Termen val(), sowie deren Argumenten arg(), versteht der Autor als eigenständigen Beitrag zu dem an Bedeutung gewinnenden Gebiet der Modell-Konstruktion.

Literaturverzeichnis

[ABB+00] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In Manuel Ojeda-Aciego, Inma P. de Guzmán, Gerhard Brewka, and Luís Moniz Pereira, editors, *Proc. 8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, volume 1919 of *LNAI*, pages 21–36. Springer-Verlag, October 2000.

[Ahr00] Wolfgang Ahrendt. A basis for model computation in free data types. In *Proceedings, CADE-17 Workshop on Model Computation - Principles, Algorithms, Applications, Pittsburgh, Pennsylvania, USA*, 2000.

[Bac88] Leo Bachmair. Proof by consistency in equational theories. In *Proc. Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland*, pages 228–233. IEEE Press, 1988.

[BDP+79] M. Broy, W. Dosch, H. Partsch, P. Pepper, and M. Wirsing. Existential quantifiers in abstract data types. In Hermann A. Maurer, editor, *Automata, Languages and Programming, Sixth Colloquium, Graz, 1997*, volume 71 of *LNCS*, pages 73–86. Springer-Verlag, 1979.

[BEL01] Matthias Baaz, Uwe Egly, and Alexander Leitsch. Normal form transformations. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 5, pages 273–333. Elsevier Science Publishers, 2001.

[BFN96] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orłowska, editors, *Proc. European Workshop: Logics in Artificial Intelligence, JELIA*, volume 1126 of *LNAI*, pages 1–17. Springer-Verlag, 1996.

[Bro78] Frank Malloy Brown. Towards the automation of set theory and its logic. *Artificial Intelligence*, 10(3):281–316, 1978.

[BT82] J.A. Bergstra and J.V. Tucker. The completeness of the algebraic specification methods for computable data types. *Information and Control*, 54:186–200, 1982.

[CoF] CoFI. The Common Framework Initiative for algebraic specification and development, Elektronische Archive. Notizen und Dokumente verfügbar unter http://www.brics.dk/Projects/CoFI/.

[CoF98] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [CoF], October 1998.

[Com01] Hubert Comon. Inductionless induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 14, pages 911–960. Elsevier Science Publishers, 2001.

[CP00] Ricardo Caferra and Nicolas Peltier. Combining enumeration and deductive techniques in order to increase the class of constructible infinite models. *Journal of Symbolic Computation*, 29:177–211, 2000.

[CZ92] Ricardo Caferra and Nicolas Zabel. A method for simultaneous search for refutations and models by equational constraint solving. *Journal of Symbolic Computation*, 13:613–641, 1992.

[CZ93] Ricardo Caferra and Nicolas Zabel. A tableaux method for systematic simultaneous search for refutations and models using equational problems. *Journal of Logic and Computation*, 3(1):3–26, 1993.

[EFT92] Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Einführung in die mathematische Logik*. BI-Wissenschaftsverlag, 1992. 3., erw. Auflage.

[EGL89] Hans-Dieter Ehrich, Martin Gogolla, and Udo Walter Lipeck. *Algebraische Spezifikation abstrakter Datentypen*. Teubner, 1989.

[Eis71] Murray Eisenberg. *Axiomatic Theory of Sets and Classes*. Holt, Rinehart and Winston, 1971.

[EM85] Hartmut Ehring and Bernd Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[FH91] Hiroshi Fujita and Ryuzo Hasegawa. A model generation theorem prover in KL1 using a ramified-stack algorithm. In Koichi Furukawa, editor, *Proceedings 8th International Conference on Logic Programming, Paris/France*, pages 535–548. MIT Press, 1991.

[FH98] Hiroshi Fujita and Ryuzo Hasegawa. Implementing a Model-Generation based Theorem Prover MGTP in Java. Research Reports on Information Science and Electrical Engineering Vol. 3, No. 1, pp. 63–68, Kyushu University, 1998.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, second edition, 1996.

[FL96] Christian Fermüller and Alexander Leitsch. Hyperresolution and automated model building. *Journal of Logic and Computation*, 6(2), 1996.

[GA99] Martin Giese and Wolfgang Ahrendt. Hilbert's ϵ-terms in Automated Theorem Proving. In Neil V. Murray, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference (TABLEAUX'99), Saratoga Springs/NY USA*, volume 1617 of *LNAI*, pages 171–185. Springer-Verlag, 1999.

[GH78] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GTWW75] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Abstract data types as initial algebras and the correctness of data representation. In A. Klinger, editor, *Computer Graphics, Pattern Recognition and Data Structures*, pages 89–93. IEEE, 1975.

[Gut75] J.V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* University of Toronto, Computer Science Dept. Report CSRG-59, 1975.

[Hab00] Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln.* Shaker Verlag, 2000. Dissertation, Universität Karlsruhe.

[HFK97] Ryuzo Hasegawa, Hiroshi Fujita, and Miyuki Koshimura. MGTP: a model generation theorem prover—its advanced features and applications. In Didier Galmiche, editor, *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Pont-à-Mousson, France*, volume 1227 of *LNAI*, pages 1–15. Springer-Verlag, 1997.

[Kli97] Stefan Klingenbeck. *Counter Examples in Semantic Tableaux.* Diski 51, infix Verlag, 1997. Dissertation, Universität Karlsruhe.

[Lei00] Alexander Leitsch. Syntactic model building by calculi. In *Proceedings, CADE-17 Workshop on Model Computation - Principles, Algorithms, Applications, Pittsburgh, Pennsylvania, USA*, 2000.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types.* Wiley/Teubner, 1996.

[LS84] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification.* Wiley/Teubner, 1984.

[MB88] Rainer Manthey and François Bry. SATCHMO: A theorem prover implemented in Prolog. In *Proceedings 9th Conference on Automated Deduction*, volume 310 of *LNCS*, pages 415–434. Springer-Verlag, 1988.

[MBI94] Raul Monroy, Alan Bundy, and Andrew Ireland. Proof plans for the correction of false conjectures. In Frank Pfenning, editor, *Proc. 5th International Conference on Logic Programming and Automated Reasoning, Kiev, Ukraine*, volume 822 of *LNAI*, pages 54–68. Springer-Verlag, 1994.

[Pel97a] Nicolas Peltier. Increasing the capabilities of model building by constraint solving with terms with integer exponents. *Journal of Symbolic Computation*, 24(1):59–101, 1997.

[Pel97b] Nicolas Peltier. Simplifying and generalizing formulae in tableaux: pruning the search space and building models. In Didier Galmiche, editor, *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods, Pont-à-Mousson, France*, volume 1227 of *LNAI*, pages 313–327. Springer-Verlag, 1997.

[Pro92] Martin Protzen. Disproving conjectures. In D. Kapur, editor, *Proc. 11th Conference on Automated Deduction, Albany/NY, USA*, volume 607 of *LNAI*, pages 340–354. Springer-Verlag, 1992.

[Pro96] Martin Protzen. Patching faulty conjectures. In Michael McRobbie and John Slaney, editors, *Proc. 13th Conference on Automated Deduction, New Brunswick/NJ, USA*, volume 1104 of *LNCS*, pages 77–91. Springer-Verlag, 1996.

[RS98] Wolfgang Reif and Gerhard Schellhorn. Theorem Proving in Large Theories. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction—A Basis for Applications*, volume III, chapter 5, pages 225–241. Kluwer Academic Publishers, Dordrecht, 1998.

[RST00] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Fehlersuche in Formalen Spezifikationen. Ulmer Informatik-Berichte 2000-06, Universität Ulm, Fakultät für Informatik, 2000.

[RST01] Wolfgang Reif, Gerhard Schellhorn, and Andreas Thums. Flaw detection in formal specifications. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Automated Reasoning, First International Joint Conference, IJCAR 2001 Siena, Italy, June 18-23, 2001 Proceedings*, volume 2083 of *LNAI*. Springer-Verlag, 2001.

[SA91] Volker Sperschneider and Grigorios Antoniou. *Logic, a Foundation for Computer Science*. Addison-Wesley, 1991.

[Sch98] Arno Schönegge. *Spezifizierbarkeit berechenbarer Datentypen*. Shaker Verlag, 1998. Dissertation, Universität Karlsruhe.

[Sla94] John Slaney. FINDER: finite domain enumerator. In Alan Bundy, editor, *Proc. 12th International Conference on Automated Deduction, CADE-12, Nancy/France*, volume 814 of *LNCS*, pages 798–801. Springer-Verlag, 1994.

[Tam91] Tanel Tammet. Using resolution for deciding solvable classes and building finite models. In *Baltic Computer Science*, volume 502 of *LNCS*, pages 33–64. Springer-Verlag, 1991.

[Thu98] Andreas Thums. Fehlersuche in Formalen Spezifikationen. Diplomarbeit, Fakultät für Informatik, Universität Ulm, 1998.

[Wir90] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Formal Models and Semantics. Elsevier Science Publishers, 1990.

[Wir95] Martin Wirsing. Algebraic specification languages: An overview. In Egidio Astesiano, Giana Reggio, and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification, 10th Workshop on Specifications of Abstract Data Types, S. Margherita, 1994*, volume 906 of *LNCS*, pages 81–115. Springer-Verlag, 1995.

[ZZ96] Jian Zhang and Hantao Zhang. Generating models by SEM. In Michael McRobbie and John Slaney, editors, *Proc. 13th Conference on Automated Deduction, New Brunswick/NJ, USA*, volume 1104 of *LNCS*, pages 309–327. Springer-Verlag, 1996.

Index

Adäquatheit
  von Meta-Atom-Mengen, 117
ADT-Signatur, 7
ADT-Spezifikationen, 23
  normalisierte, 44
Algebren
  frei erzeugte, 16
  klassische, 56
Allabschluß, 21
Allgemeingültigkeit, 30
α, 7
anzÄußKonstr, 144
anzUnabhArgvalTerme, 144
Äquivalenz
  von Formeln, 30
  von Spezifikationen, 34
Argumentselektion, 11
argval-Terme, 143
Auswertung
  von Formeln, 26
  von Grund-Meta-Termen, 131
  von Termen, 15
Axiome, 23

Basiskonstruktoren, 8

$\mathcal{C}$, 7
$Cl_\forall$, 21
$Cl_\exists$, 21
Contr, 36
C_s, 7
CT_s, 11
CT_Σ, 11

Datentypen
  frei erzeugte, 7, 18
  monomorphe, 18
  polymorphe, 18
  spezifizierte, 33
$Deut_\mathcal{I}$, 132
Deutung
  von Grund-Meta-Atomen, 132
  zutreffende, 132

Einschränkung
  von Substitutionen, 13
Erfüllbarkeit, 29
  in Datentypen, 32
  von Spezifikationen, 33
Erfüllung
  von Litaralen, 120
$\mathfrak{ErsetzDiff}$, 76
$\mathfrak{ErsetzI}$, 76
$\mathfrak{ErsetzIs}$, 76
$\mathfrak{ErsetzSame}$, 76
$\mathfrak{Ersetzung}$, 76
Ersetzungsregeln, 76
Erweiterung
  eines Astes, 106
Evaluation
  in Meta-Atom-Mengen, 116
Existenzabschluß, 21

$\mathcal{F}$, 7
$\mathcal{F}$-Interpretationen, 14
Familien, 7
  deren Vereinigung $\overline{\mathcal{X}}$, 7
FFT_s, 10
FFT_Σ, 10
Folgerbarkeit
  aus Formeln, 30
  aus Spezifikationen, 34
Formeln, 20

geschlossene, 22
For_Σ, 20
For^0_Σ, 22
frei, 21
$\mathfrak{FreiErz}$, 78
F_s, 7
funktional, 115
$\mathfrak{Funktionalitaet}$, 91
Funktionen, 7
Funktions-Ausdrücke, 100
Funktionserweiterung, 9
Funktionsvariable, 100
fv, 100

Gegeninterpretationen, 42
Gegenmodelle, 42
Gegenspezifikation, 42
normalisierte, 54
Gegenteil, 36
Größe
von Konstruktortermen, 11
Grund-Meta-Atome, 101
Grund-Meta-Terme, 98
wohlgeformte, 99
Grundklauseln, 22
Grundliterale, 22
Grundterme, 12
Gültigkeit
in Datentypen, 32
in Interpretationen, 29
Gultigkeit
in Algebren, 28

Homomorphismus, 56

I-funktional, 114
Instantiierung
von Formelmengen, 94
von Formeln, 93
größenbeschränkte, 95
Interpretationen, 14
entsprechende, 115
erfüllende, 33
Isomorphie, 56
Iterationen, 137

Klauseln, 20
Kl_Σ, 20
Kl^0_Σ, 22
Konsistenz
zu Spezifikationen, 35
Konstanten, 8
Konstruktor-Stelligkeit
maximale, 97
Konstruktoren, 7
Konstruktorterm-Algebren, 57
Konstruktorterm-Substitution
vollständige, 111
Konstruktorterme, 11
genug, 13

λ, 7
Läufe, 107
Literale, 20
Lit_Σ, 20
Lit^0_Σ, 22
lose Semanik, 61

MA_Σ, 101
MA^0_Σ, 101
$max\alpha$, 97
$\mathfrak{MaxInst}$, 97
maxval, 153
MergeSort, 179
Meta-Atome, 101
Meta-Term-Tupel, 100
Meta-Terme, 98
Meta-Variablen, 100
Meta-Variablen-Substitution, 100
metaUnterterme, 144
`MG-proc`, 106
MGTP, 104, 159
minimale Äste, 139
minimale Astlänge, 139
Modell-Generierung, 65
Modell-Generierungs-Ast, 105
Modell-Generierungs-Baum, 105
Modell-Generierungs-Prozedur, 103

  - allgemeine, 106
  - faire, 139
- Modell-Korrektheit
  - beschränkte, 109
  - größenbeschränkte, 130
  - instanzenbeschränkte, 129
- Modell-Vollständigkeit, 109, 142
- Modelle
  - von Formeln, 29
  - von Spezifikationen, 33
- monomorphe Ansätze, 61
- MT_Σ, 98
- MT^0_Σ, 98
- $mtval_{\mathcal{I}}$, 131

- NatStack, 163
- Nicht-Allgemeingültigkeit, 37
- Nicht-Folgerbarkeit, 36, 37
  - aus Spezifikationen, 37
- Normalisierung
  - von Spezifikationen, 43

- p_Forever, 177
- $[\Phi/Inst]$, 94
- $[\varphi/Inst]$, 93
- $\Phi_{\leq n}$, 95
- $\varphi_{\leq n}$, 95

- R-erreichbare
  - Äste, 136
  - Bäume, 136
- R-Saturiertheit, 105
- Redukt, 47
- Regeln
  - f. freie Erzeugtheit, 78
  - f. die Funktionalität, 91
- Rep, 85
- Repräsentation, 85

- S, 7
- semantische Methoden, 184
- Σ, 7
- Signaturen, 7
- Signaturerweiterung, 9
- Skolemisierung
  - einschrittige, 45
  - vollständige, 50
- *sort*, 10
- Sorten, 7
- Spezifikationen
  - monomorphe, 39
  - polymorphe, 39
- Substitution, 12
  - in Meta-Atomen, 101
  - in Meta-Termen, 101
- syntaktische Methoden, 184

- Teilsignatur, 9
- Terme, 10
  - funktionsfreie, 10
- TestDiff, 78
- TestSame, 78
- Theorie, 32
- $t{\downarrow}_i$, 11
- *topTerme*, 144
- TransAxiom, 84
- TransAxioms, 84
- Transformation
  - der Axiome, 84
  - der Literale, 85
    - optimierte, 90
  - der Signatur, 79
  - der Sorten, 75
  - der Terme, 87
  - einer Größen-Beschränkung, 97
  - einer Instanzen-Menge, 96
  - einer Sorte, 74
  - eines Axioms, 84
  - eines Konstruktors, 74
- TransInst, 96
- TransKonstr, 74
- TransLit, 85, 90
- TransSig, 79
- TransSort, 74
- TransSorts, 75
- TransSpec, 92
- TransTerm, 87

T_s, 10
T_s^0, 12
T_Σ, 10
T_Σ^0, 12
tv, 100

Unerfüllbarkeits-Korrektheit, 109, 138
Ungültigkeit
  in Algebren, 37
  in Datentypen, 37
Unterformeln
  negative, 44
  positive, 44
Unterspezifikation, 41
Unterterm-Menge, 117
Unterterme, 110

val-funktional, 114
val-Terme, 143
$val_{\mathcal{I},\beta}$, 15, 26
Var, 21, 98
Variablen, 9
  auftretende, 12, 21, 98
  frei auftretenden, 21
Variablenbelegen
  abgeänderte, 15
Variablenbelegungen, 14
Verwerfen
  eines Astes, 106
V_s, 9
V_Σ, 9

wMT_Σ, 99
Wort
  das leere, 7
Wörter, 7