Strikte Verfahren zyklischer Berechnung

vorgelegt von
Diplom-Informatiker
Baltasar Trancón y Widemann

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. H. Ehrig
Berichter:	Prof. Dr. P. Pepper
Berichter:	Prof. Dr. B. Möller

Tag der wissenschaftlichen Aussprache: 28. Februar 2007

Berlin 2008
D 83

Im Andenken an meinen Vater José Trancón Rodriguez, † 25. Dezember 2004, dem ich in jeder Hinsicht den Zugang zur Welt der Wissenschaft verdanke.

Abstract

This present work investigates theory and techniques of declarative and functional construction and processing of cyclical data. Contrary to the non-well-founded approaches forming the foundations of so-called *lazy* programming languages like Haskell or Clean, in which infinitely descending subterm chains are allowed, we replace the common meta-theory of *algebra* by its categorial dual *coalgebra*. This allows to distinguish cylicity as a particular phenomenon from proper infinity. Thus cyclic data that allow infinitly progressing observations, but are represented finitely nonetheless, can be processed with terminating computations under strict semantics.

The most successful coalgebraic approach so far lead to the development of the language Charity that employs category-theoretic constructs for both denotational and operational semantics. We pursue a contrary approach, gearing operational semantics more towards traditional strict functional, imperative or object-oriented languages.

This approach is not restricted to using just final coalgebra as the structure of semantical elements. Actual representations in machine memory are described as elements of non-final coalgebras, as well. In this framework, any primitively corecursive function can be specified as a family of coalgebras, each describing a recursion step on a single state of memory. An algorithm is given that corresponds up to bisimilarity to the unique morphism into the final coalgebra, namely the induced corecursive function. This algorithm terminates even under strict evaluation rules on all finite input states, subsuming real encodings of data graphs in memory.

We also reduce the decision procedure for predicates by depth-first search, like it is used in the resolution strategy of Prolog, to a coalgebraic representation. If deduction rules are cyclic, there are potentially several different fixpoint semantics to this resolution. We give another algorithm that decides the inclusion of formulae in one fixpoint model, given proper parametrization. We also define a class of such parameters constructively, covering all of the commonly intended fixpoints, specifically piecewise least and greatest ones.

Both algorithmic methods are presented in a way that allows implementation with the same technical means of cycle detection. Since cyclical predicates cannot be decided lazily, considerable power is gained by combining the two methods. Numerous problems on cyclical input concisely solved in this way require radical transcoding in traditional frameworks without coalgebraic semantics and cycle detection. Specifically, cell identities have to be modeled explicitly, which we consider as contradictory to the attitude of abstraction innate to a declarative paradigma.

Following a detailed analysis of the technical constraints, we define a virtual machine as operational semantics for supporting the presented methods in a programming language. This machine is modeled after standard examples like the Java machine. But it also caters for the needs of higher-order functional programming on the one hand and of cycle detection and handling on the other. We describe the existing implementation of the machine and its peripheral tools in Java. This comprises a program model, an interpreter with optional graphical user interface, an assembly language for textual program input, and a compiler suitable for both traditional ahead-of-time production of C code and just-in-time operation.

The relevance of the coalgebraic viewpoint is finally illustrated by three exemplary application domains of increasing complexity. For each of these, a collection of typical algorithms is developed both as formal specifications and machine programs, available in the examples directory of the implementation.

This work provides exhaustive and self-contained foundations for the formal specification and effective implementation of cyclical computations. The given implementation is fit for usage as a programming system on its own, or as a reference for the integration of similar methods into other systems and paradigms.

Zusammenfassung

Diese Arbeit untersucht Theorie und Techniken der deklarativen, funktionalen Erzeugung und Verarbeitung von zyklischen Daten. Anders als bei nichtfundierten Ansätzen, wie sie den sogenannten *lazy* Programmiersprachen wie Haskell und Clean zugrunde liegen, und in denen unendlich absteigende Teiltermketten gestattet sind, ersetzen wir die übliche Metatheorie *Algebra* durch ihre Dualisierung *Coalgebra*. Dadurch wird es möglich, Zyklizität als eigenständiges Phänomen von echter Unendlichkeit zu unterscheiden. Somit können auf zyklischen Daten, die unendliche fortschreitende Beobachtungen zulassen, aber endlich repräsentiert sind, terminierende Berechnungen mit strikter Semantik ausgeführt werden.

Der bisher erfolgreichste coalgebraische Ansatz führte zur Entwicklung der Sprache Charity, die sowohl in denotationeller als auch operationaler Semantik auf kategorientheoretische Konstrukte zurückgreift. Hier wird ein entgegengesetzter Ansatz verfolgt, der die operationale Semantik enger an traditionelle strikt funktionale, imperative oder objektorientierte Sprachen anlehnt.

Eine Besonderheit dieses Ansatzes ist, daß nicht nur eine finale Coalgebra als Struktur der semantischen Elemente verwendet wird. Konkrete Repräsentationen im Speicher eines Rechners werden als Elemente von nichtfinalen Coalgebren beschrieben. In diesem Rahmen läßt sich eine primitiv corekursive Funktion als eine Familie von Coalgebren darstellen, von denen jede einen Rekursionsschritt der Funktion auf einem Speicherzustand beschreibt. Es wird ein Algorithmus angegeben, der modulo Bisimilarität dem eindeutigen Homomorphismus in die finale Coalgebra, also der erzeugten corekursiven Funktion entspricht. Dieser Algorithmus terminiert auch bei strikter Auswertung auf allen endlichen Eingabezuständen, also insbesondere auf konkreten Graphdarstellungen von Daten im Speicher.

Ebenso wird das Entscheiden von Prädikaten durch Tiefensuche, wie etwa beim Resolutionsverfahren der Sprache Prolog, auf eine coalgebraische Darstellung zurückgeführt. Falls die definierenden Schlußregeln zyklisch sind, existieren potentiell verschiedene Fixpunktsemantiken. Es wird auch hier ein Algorithmus angegeben, der bei geeigneter Parametrisierung die Zugehörigkeit einer Formel zu einem Fixpunktmodell entscheidet. Außerdem wird eine Klasse von Parametern konstruktiv definiert, die die gewöhnlich intendierten Fixpunkte, nämlich stückweise kleinste und größte Fixpunkte, auswählen.

Beide algorithmischen Verfahren sind so dargestellt, daß sie sich mit den selben technischen Mitteln der Zyklenerkennung implementieren lassen. Da das Entscheiden zyklischer Prädikate *lazy* nicht möglich ist, ergibt sich durch Kombination beider Verfahren eine erhebliche Mächtigkeit. Zahlreiche Probleme auf zyklischen Eingaben, die auf diese Weise gelöst werden können, erfordern in herkömmlichen Ansätzen ohne coalgebraische Semantik und Zyklenerkennung eine radikale Umcodierung enschließlich der expliziten Modellierung von Zellenidentitäten, was dem Abstraktionsanspruch eines deklarativen Paradigmas zuwiderläuft.

Nach detaillierter Analyse der technischen Randbedingungen wird ein operationales Semantikmodell für die sprachseitige Unterstützung der entwickelten Verfahren in Form einer virtuellen Maschine definiert. Diese ist herkömmlichen Modellen wie der Java-Maschine nachempfunden, trägt aber auch den besonderen Bedürfnissen der funktionalen Programmierung höherer Ordnung einerseits und der Zyklenerkennung und -behandlung andererseits Rechnung. Die existierende Java-Implementierung der virtuellen Maschine wird beschrieben. Sie umfaßt Programmmodell und Interpreter mit grafischer Benutzeroberfläche, eine textuelle Eingabesprache, sowie einen optimierenden Compiler, der zur statischen Erzeugung von C oder im *just-in-time*-Betrieb eingesetzt werden kann.

Die Relevanz der coalgebraischen Sichtweise wird durch drei beispielhafte Anwendungsfelder von aufsteigender Komplexität illustriert, für die jeweils eine Reihe typischer Algorithmen in formaler Form und als Maschinenprogramm (im Beispielverzeichnis der Implementierung) entwickelt werden.

Die Arbeit legt also eine Basis für die formale Spezifikation und konkrete Implementierung zyklischer Verfahren. Die angegebene Implementierung kann direkt als Programmiersystem genutzt werden, oder als Referenz für die Integration solcher Verfahren in andere Systeme und Paradigmen dienen.

Inhaltsverzeichnis

I Einführung **13**

1 Motivation **17**
1.1 Zyklen . . . 17
1.1.1 Teufelskreise . . . 17
1.1.2 Zyklen und Datenstrukturen . . . 17
1.1.3 Zyklen und Wissensrepräsentation . . . 17
1.1.4 Zyklen und formale Sprachen . . . 18
1.2 Zyklenerkennung und -behandlung . . . 21
1.2.1 Algorithmische Division . . . 21
1.3 Ziele und Vorgehen . . . 23
1.3.1 Praktische Erwartungen . . . 23
1.3.2 Wissenschaftliche Agenda . . . 24
1.3.3 Sprachliche Grundlagen . . . 26

2 Algebra und Coalgebra **29**
2.1 Universelle Algebra und Coalgebra . . . 29
2.1.1 Kategorien und Funktoren für (Co)Algebra . . . 30
2.1.2 Kategorielle Algebra . . . 31
2.1.3 Kategorielle Coalgebra . . . 32
2.1.4 Beziehungen zwischen Algebren und Coalgebren . . . 33
2.2 Signaturfunktoren für Daten . . . 36
2.2.1 Initiale und Finale Modelle . . . 37
2.3 Relationen auf Coalgebren . . . 38
2.3.1 Erreichbarkeit . . . 38
2.3.2 Lokale Beziehungen . . . 40
2.3.3 Globale Beziehungen . . . 41
2.4 Polynomielle Signaturen . . . 43
2.4.1 Existenz initialer Algebren und finaler Coalgebren . . . 43
2.4.2 Kalkül der Signaturfunktoren . . . 43
2.4.3 Komposition und Dekomposition . . . 51
2.5 Algebraische und Coalgebraische Datentypen . . . 52
2.5.1 Einsortige Daten . . . 52
2.5.2 Die Kardinalitätsexplosion . . . 52
2.5.3 Mehrsortige Daten . . . 52
2.6 Terme und Coterme . . . 55
2.6.1 Allgemeine Coterme . . . 56
2.6.2 Kanonische Coterme . . . 57
2.6.3 Coterme und Kardinalität . . . 58
2.6.4 Coterme und Pattern Matching . . . 59
2.7 Nichtfundierte Algebra und Coalgebra . . . 61

2.8 Rekursive und Corekursive Funktionen . . . 62
2.8.1 Syntaktische und semantische Rekursion . . . 62
2.8.2 Primitive Rekursion . . . 62
2.8.3 Primitive Corekursion . . . 64

3 Coalgebra des Arbeitsspeichers 67
3.1 Graphische Struktur . . . 67
3.1.1 Termgraphen . . . 67
3.1.2 Zeigergraphen . . . 68
3.2 Coalgebraische Speicherzustände . . . 69
3.2.1 Direkte und Indirekte Adressierung . . . 69
3.2.2 Nullreferenzen . . . 70
3.2.3 Unverpackte Datentypen . . . 72
3.3 Operationen auf Speicherzuständen . . . 73
3.3.1 Eigenschaften . . . 73
3.3.2 Operationen . . . 75
3.3.3 Abläufe . . . 77
3.4 Speicherverwaltung . . . 78
3.4.1 Erzeugung und Initialisierung . . . 79
3.5 Anhang: Algebraische Speicherzustände . . . 81

II Algorithmische Theorie 83

4 Zyklische Funktionen 85
4.1 Primitive Corekursion und Speicherzustände . . . 85
4.1.1 Interpretation von Speicherzuständen . . . 85
4.1.2 Abstrakter Algorithmus . . . 91
4.1.3 Grundlagen der Spezifikation . . . 93
4.1.4 Algorithmischer Grundschritt . . . 94
4.2 Algorithmus ohne Zyklenerkennung . . . 95
4.2.1 Iterative Formulierung . . . 95
4.2.2 Rekursive Formulierung . . . 97
4.3 Algorithmus mit Zyklenerkennung . . . 99

5 Zyklische Prädikate 103
5.1 Logik-Kalküle . . . 103
5.1.1 Kalküle und Regeln . . . 104
5.1.2 Kombination von Kalkülen . . . 106
5.1.3 Modell-Semantik . . . 110
5.1.4 Abstraktion . . . 113
5.2 Algebraische Kalküle . . . 114
5.2.1 Iterative Formulierung . . . 115
5.2.2 Rekursive Formulierung . . . 117
5.3 Coalgebraische Kalküle . . . 118
5.3.1 Zyklenbehandlung . . . 120
5.4 Erwartungen . . . 122
5.4.1 Operationale und semantische Unabhängigkeit . . . 124
5.4.2 Graphische Deutung . . . 127
5.5 Bisimilarität . . . 128
5.6 Beispiel aus dem Compilerbau . . . 131
5.7 Zusammenfassung . . . 132

6 Randbedingungen **133**
6.1 Effiziente Zyklenbehandlung . . . 134
6.1.1 Statische Optimierung . . . 134
6.1.2 Dynamische Optimierung . . . 135
6.2 Effiziente Rekursion . . . 140
6.3 Erkennung von Quasizyklen . . . 140
6.4 Effektive Speicherverwaltung . . . 142
6.4.1 Referenzzählung . . . 142
6.4.2 Globale Suche . . . 144
6.4.3 Zellenmarkierung . . . 145

III Praktische Realisierung **147**

7 Virtuelle Maschine **151**
7.1 Anforderungen . . . 151
7.2 Architektur . . . 153
7.2.1 Speicher . . . 153
7.2.2 Vergleich . . . 156
7.3 Statisches Modell . . . 158
7.3.1 Module . . . 158
7.3.2 Typen . . . 158
7.3.3 Prozeduren . . . 163
7.3.4 Konstanten . . . 164
7.4 Dynamisches Modell . . . 166
7.4.1 Abläufe . . . 166
7.4.2 Prozeduraufruf . . . 166
7.4.3 Referenzen . . . 167
7.4.4 Operandenstack . . . 167
7.4.5 Operationen . . . 168
7.4.6 Aufrufkonventionen . . . 176
7.4.7 Prozeduren höherer Ordnung . . . 179
7.4.8 Meta-Attribute . . . 180
7.4.9 Referentielle Transparenz . . . 182
7.5 Einordnung des Typsystems . . . 184

8 Implementierung **187**
8.1 Interpreter . . . 188
8.1.1 Programmmodell . . . 188
8.1.2 Parser . . . 188
8.1.3 Ausführung . . . 189
8.2 Compiler . . . 191
8.2.1 Die Zwischensprache IMPL . . . 191
8.2.2 Transformationen . . . 193
8.2.3 Just-in-time-Codegenerierung . . . 196
8.2.4 Generierung von C . . . 196
8.3 Grafische Oberfläche . . . 198
8.3.1 Editor . . . 198
8.3.2 Visualisierung . . . 198
8.3.3 Interaktion . . . 200
8.4 Beispielbibliothek . . . 201
8.4.1 Basistypen . . . 201

8.4.2 Funktionenkalkül . . . 201
8.4.3 Ein-/Ausgabe . . . 201
8.4.4 Datenstrukturen . . . 201
8.4.5 Programmmodell . . . 201
8.5 Evaluierung . . . 202
8.5.1 Referenzproblem . . . 202
8.5.2 Realisierung . . . 202
8.5.3 Realisierungen in anderen Sprachen . . . 204
8.5.4 Ergebnisse . . . 205

9 Zusammenfassung und Ausblick 213
9.1 Zusammenfassung . . . 213
9.2 Verwandte Arbeiten . . . 216
9.2.1 Relationale Modellierung von Strukturen im Speicher . . . 216
9.2.2 Corekursion und Berechenbarkeit . . . 216
9.2.3 Coalgebraische Strukturen . . . 216
9.2.4 Fixpunktsemantik . . . 217
9.2.5 Coalgebraische Programmierung . . . 218
9.2.6 Deklarative Verfahren auf Graphen . . . 218
9.2.7 Zyklenerkennung . . . 219
9.3 Zukünftige Arbeiten . . . 220
9.3.1 Offene theoretische Fragen . . . 220
9.3.2 Offene linguistische Fragen . . . 220
9.3.3 Offene praktische Fragen . . . 222

IV Anhang: Applikationen 225

A Exakte Rationale Arithmetik 229
A.1 *b*-adische Brüche . . . 229
A.1.1 Ziffernfolgen als Datentyp . . . 230
A.1.2 Meta-Notation . . . 230
A.2 Elementare Operationen auf rationalen Zahlen . . . 231
A.2.1 Ordnung . . . 231
A.2.2 Normierung . . . 231
A.3 Arithmetik auf rationalen Zahlen . . . 232
A.3.1 Subtraktion . . . 232
A.3.2 Division . . . 233
A.4 Strukturanalyse rationaler Zahlen . . . 235
A.4.1 Bestimmung von Präfix und Periode . . . 235
A.4.2 Bruchdarstellung . . . 235

B Unendliche Listen 237
B.1 Listen als Datenstruktur . . . 237
B.2 Gewöhnliche Operationen auf unendlichen Listen . . . 238
B.2.1 Transformation der Elemente (Map) . . . 238
B.2.2 Verkettung . . . 238
B.2.3 Periodisches Einfügen und Auswählen . . . 239
B.3 Ungewöhnliche Operationen auf unendlichen Listen . . . 241
B.3.1 Quantoren . . . 241
B.3.2 Lexikographische Ordnung und Ketten . . . 241
B.3.3 Filter . . . 242

B.3.4 Sortieren . . . 243
B.3.5 Basistransformation . . . 245

C Polynomielle Subtypen **247**
C.1 Polynomielle Subtypbeziehungen . . . 248
C.1.1 Typen . . . 248
C.1.2 Typschnittstellen . . . 249
C.1.3 Subtypen . . . 250
C.1.4 Das Problem . . . 251
C.2 Lösung durch Typlogik . . . 253
C.3 Lösung durch Codierung . . . 254
C.3.1 Kompakte Codierung . . . 254
C.3.2 Subtypbeweise . . . 255
C.4 Praktische Handhabung . . . 258
C.4.1 Typkategorie . . . 258
C.4.2 Bindung dynamisch typisierter Werte . . . 261
C.4.3 Aufruf dynamisch typisierter Funktionen . . . 261
C.4.4 Optimierung . . . 261
C.5 Anwendungen . . . 263
C.5.1 Beispiele . . . 263
C.5.2 Weitere Anwendungsmöglichkeiten . . . 264

Abbildungsverzeichnis **267**

Programmverzeichnis **269**

Mathematisches Glossar **271**

Index **279**

Literaturverzeichnis **285**

Teil I

Einführung

Those who cannot remember the past are condemned to repeat it.

George Santayana[56]

In diesem ersten Teil der Arbeit soll zunächst die Zielstellung anhand von historischen Betrachtungen und paradigmatischen Beispielen geklärt werden. Anschließend wird, aufbauend auf eine etablierte kategorientheoretische Darstellung von Coalgebra, eine coalgebraische Theorie von Daten, Datentypen und Funktionen erster Ordnung formuliert. Abschließend wird die coalgebraische Sichtweise auf die extensionale Repräsentation von Daten im Arbeitsspeicher eines Rechners ausgedehnt. Es soll gezeigt werden, daß die traditionelle Codierung in Form von Speicherzellen und Referenzen natürlicherweise eine coalgebraische Semantik besitzt, die übliche algebraische Semantik dagegen eine weitergehende Abstraktion darstellt.

Kapitel 1

Motivation

1.1 Zyklen

1.1.1 Teufelskreise

Die abendländische Geistesgeschichte, der die Mathematik und damit auch die theoretische Informatik entstammen, ist durchdrungen von einem erstmals von ARISTOTELES formulierten logischen Axiom, dem *Satz vom zureichenden Grunde*. Ursprünglich eingeführt, um den logischen Zirkelschluß (*circulus in probando*) auszuschließen, wurden häufig auch selbstbezügliche Begriffsdefinitionen und andere Spielarten von Zirkularität als *circuli vitiosi* zurückgewiesen. LEIBNIZ bringt die philosophische Ablehnung der Zirkularität folgendermaßen auf den Punkt:

> *Stellen wir uns vor, das Buch über die Elemente der Geometrie sei ewig gewesen, immer sei eines vom anderen abgeschrieben worden, so leuchtet ein, daß — wenn auch der Grund für das gegenwärtige Buch in dem früheren, von dem es abgeschrieben ist, aufgezeigt werden kann — man doch, wenn man auch auf noch so viele Bücher zurückgeht, nirgends zu einem vollen Grunde gelangen wird, da man sich immer wundern kann, weshalb Bücher überhaupt und weshalb in dieser Weise geschriebene.*(sic)[30]

1.1.2 Zyklen und Datenstrukturen

Seit die Informatik Datenstrukturen systematisch und formal erfaßt hat, sind auch Zyklen bekannt:

1. Bei einigen Datenstrukturen sind sie das primäre Strukturmerkmal, etwa bei den Ringlisten.
2. Bei anderen treten sie als sekundäres Merkmal auf, etwas bei doppelt verketteten Listen und aufgefädelten Bäumen. Auch die symbolischen Verzeichnisnamen `.` und `..` eines Unix-Dateisystems gehören in diese Kategorie.
3. Bei vielen Datenstrukturen verweisen die Elemente mit Referenzen aufeinander. So werden etwa in einem Betriebssystem die zu verwaltenden Ressourcen und Entitäten durch Datensätze und ihre Beziehungen durch Referenzen dargestellt. In einem solchen Fall treten Zyklen oft als Nebeneffekt auf, da die dargestellten Relationen ja im Allgemeinen nicht zyklenfrei sind.

Die in dieser Arbeit vorkommenden Beispiele für zyklische Berechnungen werden in erste Linie solche *ad hoc* zyklischen Datenstrukturen zum Gegenstand haben.

1.1.3 Zyklen und Wissensrepräsentation

Zyklische Strukturen sind seit den ersten Anfängen der Informatik bekannt. In der Automatentheorie bilden Zyklen das einzige Mittel zur Iteration von Abläufen. Auch andere Prozessmodelle wie etwa

Petrinetze sind ohne Zyklen kaum denkbar. Auf dem Gebiet statischer Modellierung von Wissen sind Semantische Netze, E/R-Diagramme und Klassendiagramme wie etwa in UML[62] zu nennen, deren Elemente im Allgemeinen in zyklischer Relation zueinander stehen. Auch alle ontologischen Modellierungen wie etwa MOF[39] sind zwangsläufig zyklischer Natur. Schließlich ist die Reduktion ontologischer Kategorien auf absolut unabhängige Begriffe ein seit der griechischen Antike bekanntes, aber anscheinend unlösbares Problem.

Der Forschung auf dem Gebiet der sogenannten terminologischen Logiken ist die Erkenntnis zu verdanken, daß auch der unendliche Regress zyklischer Definitionen einen sinnvollen, wenn auch nicht eindeutigen, logischen Gehalt besitzt, siehe dazu [42].

1.1.4 Zyklen und formale Sprachen

Daß Programmiersprachen traditionell keine umfassenden Mittel zur Verarbeitung zyklischer Daten zur Verfügung stellen, ist umso erstaunlicher, als eine erhebliche Anzahl von Problemen bei der Modellierung formaler Sprachen selbst zyklischer Natur sind. Die Analyse und Übersetzung von Programmen ist ohne die Erkennung und Behandlung von Zyklen fast undenkbar.

Die Elemente eines Programms verweisen mit *Referenzen* aufeinander. Dadurch ergibt sich eine semantische Graphstruktur des Programms. Die Oberflächenform der Referenzen ist ein Phänomen der textuellen Syntax, die im Wesentlichen eindimensional ist. Zur Darstellung von graphischen Strukturen ist Fließtext ein unvollständiges Mittel, das durch symbolische Anknüpfungspunkte in Form von benannten Elementen und Referenzen ergänzt werden muß. Der jeweils gewählte Namen kann zwar dem menschlichen Leser Aufschluß über die Rolle des Element geben, ist aber semantisch letztendlich irrelevant.

Verweist ein Element direkt oder indirekt auf sich selbst, spricht man von *Rekursion*. In diesem Fall enthält die Graphstruktur Zyklen. Insofern ist der häufig gebrauchte Begriff *abstrakter Syntaxbaum* eigentlich ein Oxymoron, denn eine Baumstruktur kann angesichts von Rekursion nur zugesichert werden, wenn auf die Abstraktion von symbolischen Referenzen verzichtet wird. Es sollte daher von *semi-abstraktem Syntaxbaum* und *abstraktem Syntaxgraphen* als Ein- und Ausgabe der Referenzanalyse gesprochen werden.

Als Beispiel für die Allgegenwärtigkeit dieser dualen Sichten soll eine *Drosophila* der Informatik, die Fakultätsfunktion, dienen. Als Eingabe dient das folgende Haskell-Programm, dessen semi-abstrakter Syntaxbaum leicht ersichtlich ist:

$$
\begin{array}{l}
\textbf{data}\ Nat\ =\ Zero \mid Succ\ Nat \\
fac\ n\ =\ \textbf{case}\ n\ \textbf{of} \\
\quad Zero\ \ \ \ \rightarrow\ Succ\ Zero \\
\quad Succ\ m \rightarrow\ n\ *\ fac\ m
\end{array}
$$

Die Abbildungen 1.1 und 1.2 zeigen drei Stadien der Analyse der beiden Definitionen:

1. Den Term der Eingabesprache als semi-abstrakte Baumstruktur. Referenz und referiertes Objekt werden symbolisch durch gleichnamige Blätter dargestellt. Die Rekursion ist syntaktisch implizit im Vorkommen der linken in der rechten Seite.

2. Das Ergebnis der Rekursionsanalyse in algebraischer Darstellung mit Fixpunktquantoren als gerichtete azyklische Graphstruktur. Referenz und referiertes Objekt werden symbolisch durch einen einzigen Knoten dargestellt, der den Bindungsbereich einer Fixpunktvariablen repräsentiert. Die Rekursion ist semantisch implizit in der Bedeutung des `fix`-Knotens.

3. Dasselbe in zyklischer Darstellung als allgemeine Graphstruktur. Referenz und referiertes Objekt werden direkt durch den Wurzelknoten der Struktur dargestellt. Die Rekursion ist explizit in der Form des Graphen.

Die Diagramme wurden mit einer eigens für diese Arbeit entwickelten Java-Bibliothek zum automatischen und interaktiven Graphlayout erstellt. Diese findet auch in der Benutzeroberfläche der in Teil III vorzustellenden Programmierumgebung Verwendung.

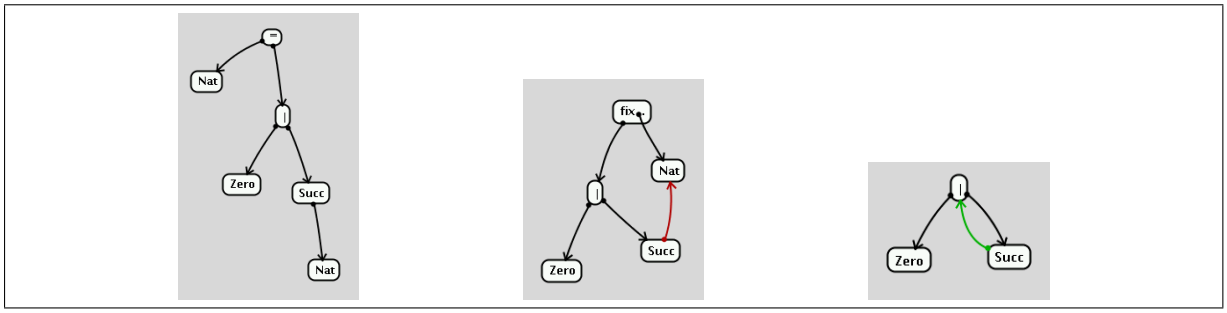


Abbildung 1.1: Namensanalyse der rekursiven Typdefinition *Nat*

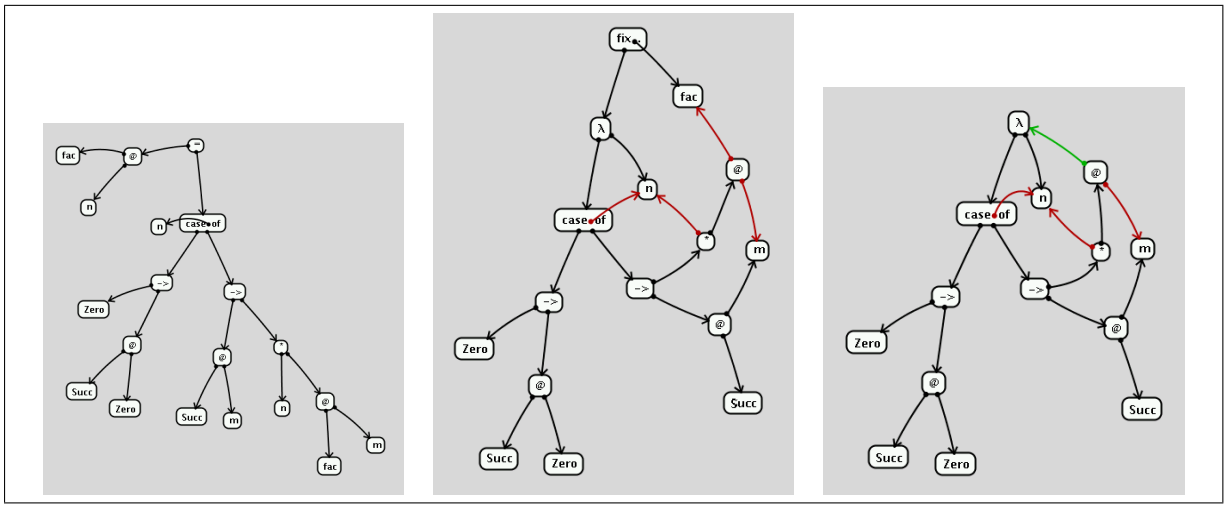


Abbildung 1.2: Namensanalyse der rekursiven Funktionsdefinition *fac*

Fast alle relevanten Programmiersprachen gestatten Rekursion in den Definitionen statischer Elemente, wie Typen und Funktionen. Rekursive dynamische Elemente, wie etwa die selbstbezügliche Initialisierung einer Variablen, werden nur von wenigen Sprachen und Systemen unterstützt. Anders als Interpreter und Compiler, werden Laufzeitsysteme nur selten mit umfassenden Fähigkeiten zum deklarativ expliziten Umgang mit Zyklen ausgestattet, sondern erfordern implizite Konstruktion, beispielsweise durch reziproke Zuweisung. Besonders flexibel sind auf diesem Gebiet noch die *lazy* funktionalen Sprachen. Sie gestatten, mittels eines rekursiven *let*-Ausdrucks zyklische Daten von statischer Form zu erzeugen, ebenso unter gewissen Voraussetzungen die strikte Sprache Scheme[26]. Die Unterstützung ist aber in beiden Fällen unvollständig, da die Wiedererkennung eines Zyklus bei seiner Traversierung nicht mit Sprachmitteln abgedeckt ist. Eben dieser Zyklenerkennung ist der nächste Abschnitt gewidmet.

Das folgende Beispiel zeigt, daß der Umgang mit zyklischen Datenstrukturen präzises Verständnis und vorsichtige Argumentation erfordert.

Beispiel 1.1 („Zyklisches“ Scheme-Programm). Bei einer Internet-Recherche zu zyklischen Datenstrukturen stößt man auf einen Artikel[28], der sich mit dem Flachklopfen geschachtelter Listen in Scheme befaßt. Die dazu verwendete Technik wird als *lazy virus* bezeichnet. Tatsächlich handelt es sich dabei um eine Emulation der typischen *lazy*-Auswertung, wie sie beispielsweise von Haskell-Implementierungen standardmäßig durchgeführt wird.

```
(define (flatten x)
  (define (does-flatten x)
    (if (not (pair? x)) x
      (cond
        ((null? (car x)) (does-flatten (cdr x)))
        ((not (pair? (car x)))
          (cons (car x) (delay (does-flatten (cdr x)))))
        (else
          (does-flatten
            (cons (caar x) (cons (cdar x) (cdr x))))))))
  (delay (does-flatten x)))

(define l (list ()))
(set-car! l l)
(force (flatten l))
```

Abbildung 1.3: „Zyklisches“ Scheme-Programm

Der Autor behauptet, das Programm arbeite mit konstantem Platzbedarf und sei auch für zyklische Listen effektiv. Bei genauerer Betrachtung stellt man aber fest, daß beide Aussagen nicht zusammen gelten, da eine zyklische Liste endlicher Ausdehnung in eine unbegrenzt wachsende *lazy* codierte Liste überführt wird. Für eine zyklische Liste höherer Ordnung, die sich selbst als erstes Element enthält, ist das Programm nicht einmal produktiv, das heißt, es wird eine *lazy* Berechnung erzeugt, die nicht terminiert. Abbildung 1.3 zeigt das Originalprogramm und den pathologischen Fall. □

Aus diesen Betrachtungen können zwei Schlüsse gezogen werden:

1. Es existieren zwei komplementäre Sichtweisen auf zyklische Daten, nämlich als *Graph* von Speicherzellen und als (unendlicher) *Baum* von begehbaren Pfaden (Term). Diese beiden Sichten sollten nicht verwechselt werden, da beim Übergang von einer zur anderen nicht alle Eigenschaften erhalten bleiben. Andererseits hat jede der Sichten ihre Vorteile. So besitzt die Baumsicht die kompaktere und elegantere Theorie, während auf der Graphsicht mehr Berechnungen effektiv sind. In dieser Arbeit sollen zunächst in Kapitel 2 beide Sichten mathematisch präzise (als finales beziehungsweise nichtfinales Modell) gefaßt werden. Der Rest der Arbeit wird sich dann mit der gewinnbringenden Kombination beider Sichten beschäftigen.

2. Zyklische Daten haben gegenüber azyklischen mehr Randfälle struktureller Art, die bei der Bewertung eines Algorithmus betrachtet werden müssen.

1.2 Zyklenerkennung und -behandlung

Für eine lokale Änderung an einer bestehenden Datenstruktur spielt es keine Rolle, ob die Struktur zyklisch ist oder nicht. Nur wenn die Struktur für eine globale Operation traversiert werden muß, zeigt sich, ob der Zyklus effektiv behandelt wird, oder ob die Traversierung unbeschränkt ergebnislos fortschreitet. In einem imperativen Kontext überwiegt häufig der erste, lokale Fall, daher kommen imperative und objektorientierte Sprachen trotzt allgegenwärtiger zyklischer Strukturen recht gut ohne eingebaute Mittel zur Erkennung und Behandlung von Zyklen aus. In einem funktionalen Kontext dagegen liegt der Schwerpunkt eindeutig auf dem zweiten, globalen Fall. Funktionale Sprachen würden daher stärker von einer Methode der automatischen Erkennung von Zyklen profitieren. Es soll gezeigt werden, daß sogar in Umkehrung funktionale Berechnungen besonders geeignet sind, mit der automatischen Erkennung und Behandlung von Zyklen ausgestattet zu werden, indem universelle Verfahren zur Zyklenbehandlung aus einem semantischen Modell purer Funktionen hergeleitet werden.

1.2.1 Algorithmische Division

Beispiel 1.2. Ein Ansatz zur automatischen Erkennung und Behandlung von Zyklen kann aus einem Algorithmus abgeleitet werden, der schon seit einigen Jahrhunderten bekannt ist, nämlich der schriftlichen Division. Man betrachte dazu folgende Beispielrechnung:

$$
\begin{array}{l}
43 : 101 = 0,\overline{4257} \\
\underline{\ 0} \\
430 \\
\underline{404} \\
\ \ 260 \\
\ \ \underline{202} \\
\ \ \ \ 580 \\
\ \ \ \ \underline{505} \\
\ \ \ \ \ \ 750 \\
\ \ \ \ \ \ \underline{707} \\
\ \ \ \ \ \ \ \ 430
\end{array}
$$

Der Algorithmus arbeitet rekursiv mit dem Divisionsrest, initial dem Dividenden, als Argument. Ein Zyklus liegt dann vor, wenn ein Rest geschachtelt noch einmal auftritt. Da der Algorithmus streng funktional lediglich von seinen Argumenten abhängt, wiederholt sich das Ergebnis ebenso zyklisch. Daher kann das Verfahren abgebrochen werden, sobald der Zyklus erkannt ist. Das Ergebnis, welches durch Schließen eines gleichförmigen Zyklus entsteht, also im Beispiel durch das Ziehen des Periodenstrichs, ist äquivalent zu dem virtuellen Ergebnis $0,425742574257\ldots$, das den Grenzwert unendlicher Traversierung ohne Beachtung des Zyklus bildet. Während sich nicht-strikte Ansätze darauf konzentrieren, alle endlichen Präfizes des letzteren möglichst effizient berechnen zu können, soll unser strikter Ansatz das vollständige erstere erzeugen. □

Der augenfälligste Vorteil eines solchen Vorgehens ist ökonomischer Natur: Der Platz- und Zeitbedarf einer Berechnung sowie gegebenenfalls das Auftreten von Seiteneffekten hängt allein von der Eingabe, nicht aber vom Beobachter ab, der im nichtstikten Ansatz die Ausführung verzögerter Berechnungen implizit kontrolliert. Dieser Vorteil ist allerdings nicht so bedeutend wie ein zweiter, weniger offensichtlicher: Steht die Zyklenerkennung zur Verfügung, und ist gewährleistet, daß zyklische Daten nicht spontan zu unendlichen degenerieren, dann können solche Daten nicht nur effektiv berechnet, sondern auch *durchsucht* werden. Eine solche Suche ist aber Voraussetzung für das Entscheiden eines rekursiven Prädikats. Man betrachte beispielsweise das folgende Problem:

Beispiel 1.3. Enthält die Dezimalentwicklung einer beliebigen gegebenen Rationalzahl die Ziffer 3?

Es bleibt im Allgemeinen nur, die Ziffern zu traversieren. Falls dabei eine 3 gefunden wird, kann sofort mit positiver Antwort abgebrochen werden. Da zum Beispiel in $43 : 101$ keine 3 enthalten ist, und damit auch keine in endlich vielen Schritten gefunden werden kann, ist dieses Problem ohne eine Form von Zyklenerkennung nicht unmittelbar entscheidbar.

Wird dagegen ein Zyklus erkannt, steht die Antwort für diesen Knoten ohne weitere Traversierung als negativ fest; wenn auf dem ersten Durchlauf durch den Zyklus bis zu seiner Erkennung keine 3 gefunden wurde, ist jeder weitere Durchlauf notwendigerweise ebenso erfolglos. □

1.3 Ziele und Vorgehen

1.3.1 Praktische Erwartungen

Dieser Abschnitt soll kurz erläutern, welche Leistung diese Arbeit aus praktischer Sicht erbringen soll und was dazu an Vorüberlegungen notwendig ist. Dementsprechend wird die Themenentwicklung hier im Vergleich zur restlichen Arbeit rückwärts vollzogen.

Zahlreiche Datenstrukturen und darüber formulierte Probleme sind weder rein algebraischer, noch rein graphtheoretischer Natur, da sie einerseits Zyklen enthalten und keine endlichen Terme darstellen, andererseits die Information von der Menge von einer Wurzel aus begehbarer Pfade getragen wird und die Graphdarstellung ausgetauscht werden kann. Folglich ist weder ein klassischer funktionaler Ansatz zur Programmierung von Algorithmen auf solchen Strukturen angemessen, noch eine Offenlegung der Graphstruktur. In ersterem terminieren in Folge der algebraischen Semantik Traversierungen im Allgemeinen nicht, da Zyklen nicht als solche, sondern als unbeschränkt tiefe Pfade wargenommen werden. In letzterem kann der eigentliche Gehalt eines Algorithmus, nämlich die in jedem Traversierungsschritt anzuwendende Operation, nicht von der Zyklenerkennung und -behandlung getrennt werden, die im Verlauf dieser Arbeit auf eine beziehungsweise zwei universelle Techniken reduziert werden.

Gesucht sind also Programmiertechniken oder besser eine einzige Programmiertechnik für funktionale Algorithmen auf als zyklische Graphen codierten Daten. Diese werden in folgenden Gedankenschritten herausgearbeitet:

1. Identifikation von typischen Anwendungsgebieten und Spezifikation von Algorithmen in diesen (siehe Teil IV). Als Gebiete betrachten wir die bereits erwähnten rationalen Zahlen, dargestellt als periodische Ziffernfolgen und ihre Arithmetik (Anhang A), zyklische Listen im Allgemeinen und Standard-Listenalgorithmen (Anhang B), sowie stellvertretend für die Probleme formaler Sprachen eine strukturelle Subtypbeziehung für freie Datentypen (Anhang C).

2. Dabei treten die zwei Arten von zyklischen Berechnungen mit jeweils einer charakteristischen Technik der Zyklenbehandlung auf, die bereits im vorigen Abschnitt angedeutet wurden, wobei die Abbildung von Zyklen auf Zyklen (wie in Beispiel 1.2) von nichtstrikten Sprachen beliebig gut simuliert werden kann, während die Abbildung von Zyklen auf Wahrheitswerte (wie in Beispiel 1.3) nur in einem zyklenerkennenden Ansatz gelingen kann.

3. Jede der beiden Techniken stellt spezifische Anforderungen an die Programmierung: Um Zyklen in der Eingabe auf Zyklen in der Ausgabe abzubilden, müssen die Elemente der Ausgabe in der richtigen, für die funktionale Programmierung aber untypischen Reihenfolge „von oben nach unten“ erzeugt werden, bei welcher ein Ergebnisknoten erzeugt wird, bevor seine Nachfolger durch rekursive Traversierung bestimmt sind; bei der Verwendung von herkömmlichen Konstruktoren zur Erzeugung von Daten ergibt sich gerade die umgekehrte Reihenfolge. Bei Suchproblemen auf zyklischen Daten muß von Fall zu Fall entschieden werden, mit welchem Wahrheitswert zur Behandlung eines erkannten Zyklus abgebrochen werden kann, ohne dabei Widersprüche zu erzeugen.

4. Es wird eine virtuelle Maschine entwickelt (Kapitel 7), die die imperative Programmierung von Algorithmen beider Arten erlaubt. Die Zyklenerkennung ist beiden Techniken gemeinsam, und wird durch das Durchsuchen des Aufrufstacks realisiert. Sie ist in der Maschinensemantik implizit. Die Zyklenbehandlung wird realisiert, indem bei einem erkannten Zyklus ein alternativer Funktionsrumpf ausgeführt wird. Für das Schließen eines Ausgabezyklus, bei dem schreibend auf den Aufrufstack zugegriffen wird, steht ein eigener Maschinenbefehl zur Verfügung. Der Aufrufstack als aktiv genutzte Datenstruktur ist also vor dem Programmierer völlig verborgen. Ein Divisionsalgorithmus wie oben sieht also etwa aus wie in Abbildung 1.4 dargestellt.

5. Die Infrastruktur der virtuellen Maschine ist in Java implementiert (Kapitel 8), so daß alle Beispielalgorithmen real analysiert und ausgeführt werden können. Die Zyklenerkennung kann in Theorie

```
fun div : digits, digits → digits =
[
        — Setze erste Ziffer
        ...
        — Rekursiver Aufruf mit Rest
        ...
]
recursively
[
        — Zyklus erkannt ⇒ schließen
        dito
]
```

Abbildung 1.4: Division im Pseudocode

(Kapitel 6) und Praxis (Abschnitt 8.5) hinreichend optimiert werden, so daß kein inakzeptabler Mehraufwand gegenüber klassischen funktionalen Ansätzen entsteht.

6. Neben der technischen Umsetzung wird gleichwertig auch die denotationelle Semantik zyklischer Verfahren untersucht. Dazu werden die beiden Zyklenbehandlungstechniken auf die theoretischen Begriffe der primitiv corekursiven Funktionen (Kapitel 4) beziehungsweise der Resolution zyklischer Logikkalküle (Kapitel 5) zurückgeführt. Vor diesem theoretischen Hintergrund wird pro Kapitel ein universeller Algorithmus für zyklische Traversierung entwickelt, der mit konkreten Schrittoperationen instanziiert werden kann, um Funktionen beziehungsweise Prädikate auf zyklischen Daten effektiv zu implementieren. Diese Algorithmen sind so formuliert, daß der universelle Anteil sich direkt mit den Zyklenerkennungs- und behandlungsmitteln der virtuellen Maschine realisieren läßt.

7. Voraussetzung für beide Modelle und insbesondere für die Beweise von Termination und Korrektheit der universellen Algorithmen ist eine Modellierung der zyklischen Daten (Kapitel 3), die sowohl die konkrete Graphstruktur als auch die abstrakte unendliche Termstruktur von Zellen und Zeigern im Speicher berücksichtigt. Während erstere die Zyklenerkennung effektiv möglich macht, bildet letztere die Semantik, bezüglich derer sich die konstruierten Algorithmen als Funktionen oder Prädikate zeigen sollen. Ferner sollen sich Daten im Speicher trotz der erwähnten Vorgehensweise, bei der partielle Strukturen erzeugt und erst mit unbeschränkter Verzögerung vervollständigt werden, referentiell transparent, also wie mathematische Objekte verhalten.

8. Als gemeinsame Meta-Theorie für Daten, Funktionen und Prädikate erweist sich die kategorielle Coalgebra als besonders adäquat, da hier alle verwendeten Konzepte, nämlich Zyklizität, Corekursion, Referentielle Transparenz und die Dualität von abstrakter und konkreter Struktur als elementare und kanonische Bestandteile auftreten (Kapitel 2). Für die geschlossene Darstellung der Gesamttheorie in diesem Rahmen wird die Reformulierung einiger bekannter algebraischer Modelle in Kauf genommen.

1.3.2 Wissenschaftliche Agenda

Diese Arbeit soll einen Beitrag zu Theorie und Praxis der zyklischen Berechnung liefern, indem folgende einzelnen Ziele identifiziert und verfolgt werden, wobei sich die Reihenfolge nun an der textuellen orientiert:

1. *Deutung und begriffliche Erweiterung der kategoriellen universellen Coalgebra als Theorie der zyklischen Daten.* Hiermit beschäftigt sich Kapitel 2, indem bekannte Sätze und Fallbeispiele universeller Coalgebra zusammengetragen, im Licht einer Datentheorie beleuchtet und um weitere, für die kompakte Formulierung späterer Kapitel notwendige Begriffe ergänzt werden.

2. *Fortsetzung des coalgebraischen Paradigmas von der rein semantischen Ebene von Daten auf die tatsächliche Repräsentation und die Dynamik von Daten im Speicher einer abstrakten oder konkreten Maschine.* Dies ist der Inhalt von Kapitel 3.

3. *Identifikation der Bedingungen für die Möglichkeit zyklischer Berechnung.* Dies geschieht in konstruktiver Weise in den Kapiteln 4 und 5 durch minimale Erweiterung von herkömmlichen Auswertungsstrategien, wie sie in operationaler Semantik strikter Sprachen, egal ob imperativ, funktional oder hybrid, Verwendung finden.

4. *Methodische Untersuchung der Voraussetzungen für zyklische Berechnung.* Es zeigt sich, daß die spezifischen Techniken längst zum Repertoire versierter Programmierer gehören, allerdings in unbewußter und undisziplinierter Form. In allen Abschnitten dieser Arbeit, durchgängig von der Theorie zur Praxis, werden theoretische Ergebnisse über diese Techniken und Beispiele für deren gewinnbringenden Einsatz entwickelt.

5. *Abgrenzung der Dualität Algebra-Coalgebra in mengentheoretisch fundierten Ansätzen von der Symmetrie nichtfundierter Ansätze.* In Semantiken auf Basis nichtfundierter Mengentheorie sind die ausgezeichneten Modelle von Algebra und Coalgebra isomorph, weshalb die Algebra als Meta-Theorie oft genügt; in fundiertem Fall dagegen sind die der Coalgebra stets (und teilweise erheblich) mächtiger. Diesem Unterschied direkt widmet sich eine kurze Gegenüberstellung am Ende von Kapitel 2. Eine detailliertere Charakterisierung erfolgt indirekt durch die Algorithmen des coalgebraischen Berechnens und Entscheidens in Kapitel 4 beziehungsweise 5. Es zeigt sich, daß die nichtfundierte Funktionsberechnung mächtiger als die strikt coalgebraische ist. Auf der Seite des Entscheidens logischer Aussagen dagegen ist der coalgebraische Ansatz im Vorteil und löst Probleme, die mit herkömmlichen Techniken nicht in eine einfache rekursive Darstellung gebracht werden können.

   Als weitere Illustration der wesentlichen Unterschiede wird in Kapitel 7 eine zyklische abstrakte Maschine entwickelt, die starke und beabsichtigte Übereinstimmungen mit herkömmlichen imperativen Modellen aufweist, während nichtfundierte Ansätze radikal andere Berechnungsmodelle propagieren.

6. *Semantische Integration von Theorie und Praxis der zyklischen Berechnung.* Dies geschieht in konstruktiver Weise, indem den mathematischen Modellierungen der theoretischen Algorithmen und der praktischen abstrakten Maschine ausführbare formale Spezifikationen in Form von Haskell-Programmen zur Seite gestellt werden. Obwohl die Darstellungen notwendigerweise voneinander abweichen, sind sie so gewählt, daß eine formale Rückführung praktischer Abläufe auf ihre theoretische Bedeutung gewährleistet ist.

7. *Nachweis der Praktikabilität der operationalen Semantik.* Dies geschieht in Bezug auf die Effektivität der Ausführbarkeit zyklischer Programme in Kapitel 7, indem eine vollständige, lauffähige Java-Implementierung der Maschine und einer textuellen Eingabesprache beschrieben wird. Auf die Effizienz der Ausführbarkeit zyklischer Programme wird in Kapitel 6 auf theoretischer Ebene bezug genommen, indem die Optimierbarkeit der spezifischen Operationen auf abstrakter Ebene untersucht wird. Dem steht auf konkreter Ebene Kapitel 8 gegenüber, wo die Java-Implementierung eines optimierenden Compilers, der sowohl zur Erzeugung nativer Maschinenprogramme als auch im *just-in-time*-Betrieb eingesetzt werden kann, beschrieben und die Performanz der erzeugten Programme empirisch beurteilt wird.

8. *Nachweis der praktischen Relevanz des ganzen Ansatzes.* Dies ist die Aufgabe von Teil IV, in dem mehrere beispielhafte Anwendungsfelder diskutiert werden. Aufbauend auf zyklischer Berechnung lassen sich nützliche Verfahren in einer kompakten und eleganten Form darstellen, wie sie bei Einsatz herkömmlicher Berechnungstechniken nicht ohne weiteres denkbar ist.

1.3.3 Sprachliche Grundlagen

1.3.3.1 Mathematik

Die mathematischen Formulierungen in dieser Arbeit basieren auf einfachen Grundlagen der als wohlbekannt vorausgesetzten Kalküle von Mengen, Relationen, Abbildungen und Kategorien. Da wie überall in der Mathematik auch hier widerstreitende Notationen und Definitionen koexistieren, wurde im Anhang ein Glossar der verwendeten mathematischen Grundbegriffe eingerichtet, das als Nachschlagewerk für in der Arbeit verwendete, aber nicht definierte Begriffe dient.

1.3.3.2 Warum Coalgebra?

Der allgemeine Stand der Technik bei der Modellierung zyklischer Strukturen sind relationale Ansätze, wie sie besonders in den Arbeiten von MÖLLER[40] Verwendung finden. Da jene Arbeiten auf klassischen, wohlverstandenen Modellierungstechniken beruhen, während hier ein weitgehend neuer Weg beschritten wird, wäre ein Vergleich an dieser Stelle noch verfrüht, er wird jedoch in Abschnitt 9.2.1 nachgeholt. Es sei vorausgeschickt, daß die bei MÖLLER sogenannte *Pointer Algebra* ebenso zutreffend auch als coalgebraische Theorie angesehen werden kann. Daß diese Sichtweise dort nicht vertreten wird, ist bedingt durch die strenge Herleitung aus den algebraischen Spezifikationstechniken, die zu diesem Zeitpunkt den Stand der Kunst darstellten. In einer nachträglichen Reinterpretation können die Ergebnisse von MÖLLER aber auch als Rechtfertigung eines coalgebraischen Ansatzes herangezogen werden. Schließlich bietet die Coalgebra als Metatheorie einiges an Mehrwert:

1. Die Coalgebra steht mit der Algebra in einer kategorientheoretischen Dualitätsbeziehung. Daher besteht eine enge Verbindung zwischen der coalgebraischen Sicht auf eine Datenstruktur, die deren Graphcharakter betont, und der algebraischen Sicht, die von der Struktur abstrahiert und einen Termcharakter zugrundelegt. Die Abstraktionsfunktion, die zwischen beiden vermittelt, ist dabei keine pragmatische Variable, sondern kanonisch festgelegt. Das erleichtert den Vergleich unseres Ansatzes mit dem der klassischen funktionalen Programmierung.

2. Coalgebra wurde, besonders in den Arbeiten von JACOBS, bereits mit großem Erfolg als Metatheorie objektorientierter Modelle eingesetzt, siehe besonders [22, 20]. Das erleichtert den Vergleich unseres Ansatzes mit dem der objektorienten Programmierung.

3. Der Begriff der Identität eines Elementes innerhalb einer Datenstruktur ist zwiespältig:

   (a) Einerseits ist die feststellbare Identität Voraussetzung für die Effektivität und die Effizienz zahlreicher Verfahren, insbesondere der Traversierung zyklischer Strukturen.

   (b) Andererseits dient eine Abstraktion von der Identität und damit von der konkreten technischen Codierung dem Niveau und dem Optimierungspotential einer Sprache.

   Zu diesem Dilemma gibt es im Wesentlichen zwei konträre Standpunkte:

   (a) Imperative und objektorientierte Sprachen legen die Identität für den Benutzer offen, und nehmen der Sprache damit alle Freiheitsgrade in der Repräsentation der Daten. Obwohl diese Explizitheit für viele Anwendungsgebiete erwünscht ist, ist sie doch besonders für die Modellierung mathematischer Gegenstände inadäquat.

   (b) Funktionale Sprachen dagegen verbergen die Identität der einzelnen Elemente vor dem Benutzer, und behalten sich damit vor, die konkrete Repräsentation statisch oder dynamisch den Gegebenheiten anzupassen. Für rein algebraische Datenstrukturen ist diese Sicht günstig, aber inhärent identitätsbehaftete Strukturen und eben insbesondere Zyklen können nicht adäquat dargestellt werden.

Die Coalgebra gestattet es, einen hybriden Standpunkt anzunehmen: Die Beziehung von identifizierbaren und abstrakten Elementen wird elegant durch Paare von nicht-finalen und finalen Coalgebren modelliert. Diese verhält sich dual zur algebraischen Beziehung von initialen Termen und ihren nicht-initialen Interpretationen. Die technische *Repräsentation* von Termen läßt sich algebraisch allerdings nur auf dem nichttrivialen Umweg über eine Graphcodierung beschreiben, während sie im coalgebraischen Fall einen natürlichen Spezialfall von Elementidentität darstellt.

1.3.3.3 Haskell

Parallel zur Entwicklung des mathematischen Gebäudes findet eine Modellspezifikation in der Programmiersprache Haskell statt. Von dieser facettenreichen Sprache werden nur wenige Aspekte verwendet, die für das Verständnis der abgebildeten Programmfragmente erforderlich sind:

1. Pure Funktionen höherer Ordnung.
2. Freie Datentypen und Typkonstruktoren höherer Ordnung.
3. Elementare Datenstrukturen: Listen und endliche Abbildungen.
4. Typklassen. Hier werden einige Erweiterungen des ghc-Systems[16] verwendet, die nicht Bestandteil des formalen Haskell-Standards[47], aber *de facto* universell akzeptiertes Sprachmittel sind, insbesondere mehrstellige Typklassen[49].
5. Funktoren, Monaden und imperative do-Notation. Hier finden insbesondere die Zustandsmonaden *State* und *StateT* Verwendung.

Andere Aspekte von Haskell, insbesondere die charakteristische nichtstrikte Semantik und *lazy*-Auswertung, sind für das Verständnis und die Ausführung des Modells irrelevant.

Im Übrigen ergeben die in den fortlaufenden Text eingestreuten Fragmente ein ausführbares Programm, das in dem zu dieser Arbeit gehörigen implementierten System vollständig und dokumentiert verfügbar ist und beispielsweise mit einer aktuellen Version des ghc geprüft und ausgeführt werden kann.

Kapitel 2

Algebra und Coalgebra

Er war ein Knabe für ein Sechstel seines Lebens; nach einem Zwölftel war ihm ein Bart gewachsen. Nach einem weiteren Siebtel heiratete er, und fünf Jahre später wurde ihm ein Sohn geschenkt. Der Sohn erreichte nur die halbe Lebensspanne des Vaters; dieser tröstete sich noch vier Jahre mit der Zahlentheorie, dann starb er.

Grabinschrift des Diophantus von Alexandria

Dieses Kapitel soll eine Einführung in die grundlegende Metatheorie der folgenden Arbeit, die Coalgebra, bieten. Dazu wählen wir die kategorientheoretische Darstellung, in der sich Algebra und Coalgebra dual verhalten. Da die Beschäftigung mit Coalgebra wegen der Allgegenwärtigkeit und Vertrautheit der Algebra psychologisch nicht unproblematisch ist, fällt diese Einführung sehr detailliert aus. Dabei werden sowohl Begriffe aus der coalgebraischen Standardliteratur zusammengetragen und in einheitlicher Notation präsentiert, als auch eigene Begriffe definiert und Sätze bewiesen, die als Metasprache für die kompakte Darstellung der späteren Kapitel notwendig sind.

Neben den zentralen Begriffen Signatur, Träger, Operation und Coalgebra werden Relationen und Konstruktionen auf Coalgebren und deren Elementen untersucht. Es wird eine Klasse von Signaturen definiert, die der Signatursyntax der klassischen formalen Algebra nachempfunden ist, und deren Anwendung zur Modellierung von Datentypen und Datenausdrücken vorgeführt. Darauf aufbauend werden coalgebraische Techniken der Funktionsdefinition untersucht.

In einigen Abschnitten werden Algebra und Coalgebra in ihrer kategorientheoretischen Dualität parallel behandelt. Andere sind dagegen spezifisch für die Coalgebra, da Algebra im Folgenden eine untergeordnete Rolle spielen wird. Zur Legitimation wird gezeigt, daß der initialen Algebra, die oft die Rolle eine kanonischen Modells spielt, dual die finale Coalgebra entspricht. Diese ist nicht nur ein adäquates Modell real existierender Datenphänomene, sondern auch eine echte Erweiterung der initialen Algebra, so daß alle coalgebraischen Überlegungen der folgenden Kapitel eine Verallgemeinerung klassischer algebraischer Programmsemantik darstellen. Am Ende des Kapitels wird noch kurz auf das Verhältnis zu nichtfundierter Algebra eingegangen, die üblicherweise herangezogen wird, wenn Probleme der Programmierung nicht mit den Mitteln fundierter Algebra beschrieben werden können.

Eingestreut in den Fluß der Definitionen werden ausgewählte Begriffe auch in Haskell modelliert, als Grundlage für die vollständige ausführbare Spezifikation der in späteren Kapiteln eingeführten Verfahren. Definitionen und Notationen für bekannte Begriffe aus der Mathematik werden zwar *ad hoc*, aber in konsistenter Weise eingeführt. Für eine Übersicht sei auf das Glossar im Anhang verwiesen.

2.1 Universelle Algebra und Coalgebra

Aus dem arithmetischen Wissen des vorderen Orients entwickelte die hellenische Hochkultur eine systematische Theorie des symbolischen Rechnens. Diese war von Anfang an geprägt von den philosophischen

Vorstellungen der Rückführung aller Dinge auf endgültige Grundlagen. Ausgereifte Theorien, die auch den unendlichen Regreß oder die wechselseitige Bedingung zulassen, sind dagegen erst im 20. Jahrhundert belegt. Hier ist als Gegenstück zur Algebra die Coalgebra zu nennen, die bisher wenig Verbreitung gefunden hat, obwohl eine Fülle von Anwendungsgebieten in der Informatik und anderen mathematischen Disziplinen bekannt sind. RUTTEN[55] charakterisiert Coalgebra als strukturelle "Theorie der Systeme".

Der in mehr als zwei Jahrtausenden gewachsenen und perfektionierten Notation der Algebra hat die Coalgebra nichts Vergleichbares entgegenzusetzen. Daraus ergeben sich auch zum Teil die Akzeptanzprobleme der Coalgebra. In den folgenden Abschnitten soll daher die klassische, syntaktische Darstellungsform der Algebra durch eine abstrakte, kategorientheoretische Form ersetzt werden. Dadurch ergibt sich die Coalgebra völlig symmetrisch durch Dualisierung.

Ein Teil der Definitionen und Sätze sind der Standardliteratur zur Coalgebra entnommen, insbesondere der Monographie [55]. Diese sind anstelle des üblichen Q.E.D.-Symbols mit einer Quellenangabe markiert. Die übrigen Abschnitte sind eigene Begriffsdefinitionen und deren Eigenschaften, die für die Formalisierung der weiteren Arbeit erforderlich sind.

2.1.1 Kategorien und Funktoren für (Co)Algebra

In den folgenden Abschnitten werden Algebra und Coalgebra in der Sprache der Kategorientheorie dargestellt:

1. Objekte und Morphismen einer *Trägerkategorie* modellieren die Trägerobjekte von (Co)Algebren beziehungsweise deren Homomorphismen.

2. Endofunktoren der Trägerkategorie modellieren Signaturen.

3. Die Klasse der (Co)Algebren zu einer Signatur und deren Homomorphismen ergibt wiederum jeweils eine Kategorie.

Welche Kandidaten als Trägerkategorien und Signaturfunktoren in Frage kommen, läßt sich auf verschiedene Weisen definieren:

1. Es können axiomatisch Eigenschaften gefordert werden, die für die weitergehenden Definitionen und Sätze notwendig sind, beispielsweise die Existenz endlicher (Co)Produkte oder (Co)Limiten von Fixpunktkonstruktionen. Dieser allgemeine Ansatz wäre für eine rein theoretische Arbeit angebracht.

2. Es können bestimmte Instanzen betrachtet werden, die für die beabsichtigte Anwendung adäquat sind. Da in dieser Arbeit der praktische Anteil mindestens gleichwertig ist, verfolgen wir diesen Weg.

Als Trägerkategorie betrachten wir in dieser Arbeit ausschließlich die Kategorie **Set** der Mengen und (totalen) Abbildungen, wie auch in [55] geschehen. Für (Co)Algebra als Metatheorie reiner Daten, die mit Funktionen erster Ordnung verarbeitet werden, ist das ausreichend. Für Funktionen höherer Ordnung wäre eine tiefere, domain-theoretische Struktur, etwa eine Klasse von CPOs, erforderlich. Darauf kann aber im Rahmen dieser Arbeit nicht eingegangen werden. Mit dieser Einschränkung folgen wir einer klassischen Arbeit über konstruktive (Co)Rekursion von MEIJER et.al.[37], die auch erst in einer substantiellen Folgearbeit [38] auf höhere Ordnung verallgemeinert wurde. Für die Konsequenzen dieser Entscheidung sei auch auf Abschnitt 2.7 verwiesen. Andererseit bietet die Kategorie **Set** den Vorzug wohlbekannter, klassischer Beweistechniken, vergleiche dazu [18].

Die Klasse der Signaturfunktoren werden wir in den Abschnitten 2.2 und 2.3 zunächst axiomatisch spezifizieren, um dann in Abschnitt 2.4 eine induktive Definition anzugeben, die den üblichen Intuitionen von Signaturen genügt, und äquivalent oder zumindest recht ähnlich definiert auch in der Standardliteratur zu finden ist.

2.1.2 Kategorielle Algebra

Definition 2.1 (Algebra)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Signaturfunktor. Sei A eine Menge. Dann heißt ein Paar $\mathbb{A} = \big(A, \alpha : \Sigma(A) \to A\big)$ eine Σ-*Algebra*. Dabei nennen wir A den *Träger* und α die *Operation* oder den *Konstruktor*. [55]

Beispiel 2.2. Sei der Funktor Σ_P definiert wie folgt:

$$\Sigma_P(X) = \{\star\} \uplus X \qquad \Sigma_P(f)(\star) = \star \qquad \Sigma_P(f)(x) = f(x)$$

Dabei sei $\uplus$ die übliche disjunkte Vereinigung. Wir werden diesen Funktor später kompakt als $\mathcal{C}_1 + \mathcal{I}$ schreiben. Die folgenden Strukturen über den natürlichen Zahlen $\mathcal{N}$ sind Σ_P-Algebren:

$$\mathbb{A}_P = (\mathcal{N}, \alpha_P) \qquad \mathbb{B}_P = (\mathcal{N}, \beta_P)$$
$$\alpha_P(\star) = 0 \qquad \beta_P(\star) = 1$$
$$\alpha_P(n) = n + 1 \qquad \beta_P(n) = 2 \cdot n \qquad \square$$

In semiformaler mathematischer Notation ist es üblich, eine Algebra und ihren Träger zu identifizieren, und die Operation als implizit gegeben vorauszusetzen. Man schreibt also A, und der Leser ergänzt (mit Hilfe von Kontextinformation) zu $\mathbb{A}$.

Definition 2.3 (Algebra-Morphismus)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Signaturfunktor. Seien $\mathbb{A}_1 = (A_1, \alpha_1), \mathbb{A}_2 = (A_2, \alpha_2)$ zwei Σ-Algebren. Sei $f : A_1 \to A_2$ eine Abbildung. Dann heißt $f : \mathbb{A}_1 \to \mathbb{A}_2$ ein Σ-*Algebra-Morphismus*, falls folgendes Diagramm kommutiert:

$$\begin{array}{ccc} \Sigma(A_1) & \xrightarrow{\Sigma(f)} & \Sigma(A_2) \\ {\scriptstyle \alpha_1}\downarrow & & \downarrow{\scriptstyle \alpha_2} \\ A_1 & \xrightarrow[f]{} & A_2 \end{array}$$ [55]

Beispiel 2.4. Die Abbildung $f(n) = 2^n$ ist ein Σ_P-Algebra-Morphismus von $\mathbb{A}_P$ nach $\mathbb{B}_P$:

$$f\big(\alpha_P(\star)\big) = 2^{\alpha_P(\star)} = 2^0 = 1 = \beta_P(\star) = \beta_P\big(\Sigma_P(f)(\star)\big)$$
$$f\big(\alpha_P(n)\big) = 2^{\alpha_P(n)} = 2^{n+1} = 2 \cdot 2^n = \beta_P(2^n) = \beta_P\big(\Sigma_P(f)(n)\big) \qquad \square$$

Definition 2.5 (Algebren-Kategorie)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Signaturfunktor. Die Σ-Algebren und Σ-Algebra-Morphismen bilden eine Kategorie $\mathbf{Alg}(\Sigma)$. [55]

Definition 2.6 (Initiale Algebra)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Signaturfunktor. Eine Σ-Algebra $\mathbb{I} = (I, \iota)$ heißt *initial*, wenn zu jeder Σ-Algebra $\mathbb{A} = (A, \alpha)$ genau ein Morphismus $(\!|\mathbb{A}|\!) : \mathbb{I} \to \mathbb{A}$ existiert. Dieser heißt *Katamorphismus* von $\mathbb{A}$. [55]

Beispiel 2.7. Nach einem bekannten Satz von Peano ist $\mathbb{A}_P$ eine initiale Σ_P-Algebra und es gilt für jede Σ_P-Algebra $\mathbb{A} = (A, \alpha)$:

$$(\!|\mathbb{A}|\!)(n) = \alpha^n\big(\alpha(\star)\big) \qquad \square$$

Satz 2.8. *Die Operation einer initialen Algebra ist ein Isomorphismus.* [55]

Die initiale Algebra ist auf diesem Abstraktionsniveau von zentraler Bedeutung, da sie das *Induktionsprinzip* verkörpert. Die Existenz des Katamorphismus ermöglicht induktives *Definieren*; seine Eindeutigkeit induktives *Beweisen*, vergleiche [55].

Der duale Begriff der *finalen* Algebra spielt eine untergeordnete Rolle, da der Träger stets das finale Objekt seiner Kategorie und die Operation der zugehörige universelle Morphismus ist, in unserem Fall also die triviale einelementige Struktur.

Die sogenannte *Bananenklammer* $(\!|\_|\!)$ wurde von Bird und Meertens [5, 36] im Rahmen der funktionalen Listenprogrammierung eingeführt, und später von Meijer und Coautoren [37, 38] auf polynomielle Algebra erweitert. In diesem *Squiggol* genannten Formalismus ist es üblich, eine Algebra und ihre Operation zu identifizieren. Man schreibt also $(\!|\alpha|\!)$, und der Leser ergänzt (mit Hilfe von Typinformation) mit A zu $(\!|\mathbb{A}|\!)$.

2.1.3 Kategorielle Coalgebra

Durch Dualisierung erhalten wir die Grundbegriffe der Coalgebra:

Definition 2.9 (Coalgebra)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Signaturfunktor. Sei C eine Menge. Dann heißt ein Paar $\mathbb{C} = \big(C, \gamma : C \to \Sigma(C)\big)$ eine Σ-*Coalgebra*. Dabei nennen wir C den *Träger* und γ die *Operation* oder den *Dekonstruktor*. [55]

Beispiel 2.10. Die folgenden Strukturen sind Σ_P-Coalgebren:

$$\begin{aligned} \mathbb{C}_P &= (\mathcal{N} \cup \omega, \gamma_P) & \mathbb{D}_P &= (\mathcal{N}, \delta_P) \\ \gamma_P(0) &= \star & \delta_P(2 \cdot n) &= n \\ \gamma_P(n+1) &= n & \delta_P(2 \cdot n + 1) &= \star \\ \gamma_P(\omega) &= \omega \end{aligned}$$

□

Definition 2.11 (Coalgebra-Morphismus)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Signaturfunktor. Seien $\mathbb{C}_1 = (C_1, \gamma_1), \mathbb{C}_2 = (C_2, \gamma_2)$ zwei Σ-Coalgebren. Sei $f : C_1 \to C_2$ eine Abbildung. Dann heißt $f : \mathbb{C}_1 \to \mathbb{C}_2$ ein Σ-*Coalgebra-Morphismus*, falls folgendes Diagramm kommutiert:

$$\begin{array}{ccc} C_1 & \xrightarrow{\quad f \quad} & C_2 \\ {\scriptstyle \gamma_1}\downarrow & & \downarrow{\scriptstyle \gamma_2} \\ \Sigma(C_1) & \xrightarrow[\Sigma(f)]{} & \Sigma(C_2) \end{array}$$

[55]

Beispiel 2.12. Die folgende Abbildung ist ein Σ_P-Coalgebra-Morphismus von $\mathbb{C}_P$ nach $\mathbb{D}_P$:

$$f(n) = 2^n \qquad f(\omega) = 0$$

Es gilt:

$$\begin{aligned} \delta_P\big(f(0)\big) = \delta_P(2^0) = \delta_P(1) = \star = \Sigma_P(f)(\star) = \Sigma_P(f)\big(\gamma_P(0)\big) \\ \delta_P\big(f(n+1)\big) = \delta_P(2^{n+1}) = \delta_P(2 \cdot 2^n) = 2^n = f(n) = \Sigma_P(f)(n) \\ \delta_P\big(f(\omega)\big) = \delta_P(0) = \delta_P(2 \cdot 0) = 0 = f(\omega) = \Sigma_P(f)(\omega) \end{aligned}$$

□

Definition 2.13 (Coalgebren-Kategorie)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Funktor. Die Σ-Coalgebren und Σ-Coalgebra-Morphismen bilden eine Kategorie $\mathbf{Coalg}(\Sigma)$. [55]

Definition 2.14 (Finale Coalgebra)**.** Sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein Signaturfunktor. Eine Σ-Coalgebra $\mathbb{F} = (F, \phi)$ heißt *final*, wenn zu jeder Σ-Coalgebra $\mathbb{C} = (C, \gamma)$ genau ein Morphismus $[\![\mathbb{C}]\!] : \mathbb{C} \to \mathbb{F}$ existiert. Dieser heißt *Anamorphismus* von $\mathbb{C}$. [55]

Die *Linsenklammer* $[\![\_]\!]$ ist das duale Gegenstück der Bananenklammer.

Beispiel 2.15. $\mathbb{C}_P$ ist eine finale Σ_P-Coalgebra und es gilt für alle Σ_P-Coalgebren $\mathbb{C} = (C, \gamma)$:

$$\begin{aligned} [\![\mathbb{C}]\!](x) = n \quad &\Longleftrightarrow \quad \gamma\big(\gamma^n(x)\big) = \star \\ [\![\mathbb{C}]\!](x) = \omega \quad &\Longleftrightarrow \quad \forall\, n.\ \gamma\big(\gamma^n(x)\big) \neq \star \end{aligned}$$

□

Satz 2.16. *Die Operation einer finalen Coalgebra ist ein Isomorphismus.* [55]

Die finale Coalgebra verkörpert das wenig bekannte *Coinduktionsprinzip*, das dem Induktionsprinzip an Mächtigkeit gleichrangig ist. Der gravierende Unterschied in der Popularität der beiden Prinzipien ergibt sich nicht daraus, daß der Coinduktion keine oder weniger Intuition entspräche, sondern in ihrem Widerspruch zu den klassischen Idealen der Fundiertheit und Zyklenfreiheit.

Die *initiale* Coalgebra besteht dual zur *finalen* Algebra aus dem initialen Objekt der Trägerkategorie und dem zugehörigen universellen Morphismus, in unserem Fall ist es also die triviale leere Struktur.

2.1.4 Beziehungen zwischen Algebren und Coalgebren

In den folgenden beiden Abschnitten werden Begriffe eingeführt, die gleichermaßen auf Algebren wie auf Coalgebren anwendbar sind. Es handelt sich dabei um abstrakte Teil-Ganzes-Relationen, die, wie in der Kategorientheorie üblich, bis auf Isomorphie durch die Existenz von Mono- beziehungsweise Epimorphismen bestimmt sind. Für die betrachtete Trägerkategorie existieren ausgezeichnete Spezialfälle, die gerade die Anhebung der mengentheoretischen Begriffe von Unter- und Quotientenmenge auf (Co)Algebren darstellen. Dabei gelingt die Anhebung nur für solche Unter- und Quotientenmengen, für die die Inklusion beziehungsweise Partition verträglich mit der Operation der Struktur ist. Zur Unterscheidung nennen wir den allgemeinen Fall *abstrakt*, den Spezialfall *eigentlich*.

Im folgenden sei $\Sigma : \mathbf{Set} \to \mathbf{Set}$ ein beliebiger Signaturfunktor.

2.1.4.1 Unter(co)algebren

Definition 2.17 (Abstrakte Unter(co)algebra)**.** Sei $\mathbb{A} = (A, \alpha)$ eine Σ-(Co)Algebra. Eine Σ-(Co)Algebra $\mathbb{A}' = (A', \alpha')$ heißt *abstrakte Unter(co)algebra* von $\mathbb{A}$ (geschrieben $\mathbb{A}' \subseteq \mathbb{A}$), wenn ein Σ-(Co)Algebra-Monomorphismus $m : \mathbb{A}' \to \mathbb{A}$ existiert. □

Definition 2.18 (Eigentliche Unter(co)algebra)**.** Sei $\mathbb{A}$ eine Σ-(Co)Algebra und $\mathbb{A}' \subseteq \mathbb{A}$ eine abstrakte Unter(co)algebra. $\mathbb{A}'$ heißt *eigentliche Unter(co)algebra* (geschrieben $\mathbb{A}' \subsetneqq \mathbb{A}$), falls gilt:

1. A' ist Teilmenge von A,
2. die zugehörige Inklusionsabbildung ist ein Σ-(Co)Algebra-Morphismus $\mathrm{in} : \mathbb{A}' \to \mathbb{A}$. □

Lemma 2.19. *Sei $\mathbb{A} = (A, \alpha)$ eine Σ-(Co)Algebra. Zu jeder Teilmenge A' von A existiert höchstens eine Operation α', so daß $(A', \alpha') \subsetneqq \mathbb{A}$.* [55]

Lemma 2.20. *Sei $\mathbb{A}$ eine Σ-(Co)Algebra. Zu jeder abstrakten Unter(co)algebra $\mathbb{A}' \subseteq \mathbb{A}$ existiert genau eine isomorphe eigentliche Unter(co)algebra $\mathbb{A}'' \subsetneqq \mathbb{A}$.* [55]

Beispiel 2.21. Die bekannte Σ_P-Algebra $\mathbb{B}_P$ hat eine eigentliche Unteralgebra $\mathbb{B}'_P = (B'_P, \beta'_P)$, wobei $B'_P = \{2^n \mid n \in \mathcal{N}\}$. Auf den Beweis der Morphismuseigenschaft wird hier verzichtet. □

Definition 2.22 (Minimalität)**.** Eine Σ-(Co)Algebra $\mathbb{A}$ heißt *minimal*, wenn sie bis auf Isomorphie keine Unter(co)algebren besitzt, das heißt, wenn alle Monomorphismen $m : \mathbb{B} \to \mathbb{A}$ iso sind. Dies ist notwendig und hinreichend dafür, daß $\mathbb{A}$ keine echten eigentlichen Unter(co)algebren (also keine außer sich selbst) besitzt. [55]

Satz 2.23. *Jede initiale Σ-Algebra ist minimal.* [55]

Unter den Σ-Coalgebren ist nur die leere Coalgebra minimal. Für eine Anwendung von *relativer Minimalität* siehe Definition 2.113.

Die Untercoalgebren einer finalen Σ-Coalgebra sind von besonderem Interesse, da sie die Modelle von coalgebraischen Spezifikationen mit *Invarianten* darstellen.

Beispiel 2.24. Eine Invariante ist eine Eigenschaft, die für ein gegebenes Element und alle durch Anwendung des Dekonstruktors transitiv erreichbaren Elemente gelten muß, siehe auch Abschnitt 2.3.1. Der Dekonstruktor γ_P der bekannten finalen Σ_P-Coalgebra $\mathbb{C}_P$ ist die Vorgängerfunktion, damit ist eine Invariante auf $\mathbb{C}_P$ ein Prädikat der Form $I(n) \equiv \big(\forall\, m \leq n.\ I'(m)\big)$. Folglich gilt $\forall\, m \leq n.\ I(n) \Rightarrow I(m)$. Damit ist jede $\mathbb{C}_P$-Invariante äquivalent zu einer Mengenbildung $\{m \mid m \leq n\}$, wobei n die größte Zahl (eventuell ω) ist, für die $I(n)$ gilt. □

Eine generische eigentliche Untercoalgebra beliebiger finaler Coalgebren wird in dieser Arbeit eine zentrale Rolle spielen, nämlich die größte, deren Elemente auch in einer endlichen eigentlichen Untercoalgebra enthalten sind. Siehe dazu Abschnitt 2.3.1.

2.1.4.2 Quotienten(co)algebren

Definition 2.25 (Abstrakte Quotienten(co)algebra)**.** Sei $\mathbb{A} = (A, \alpha)$ eine Σ-(Co)Algebra. Eine Σ-(Co)Algebra $\mathbb{A}' = (A', \alpha')$ heißt *abstrakte Quotienten(co)algebra* von $\mathbb{A}$ (geschrieben $\mathbb{A}' \mid \mathbb{A}$), wenn ein Σ-(Co)Algebra-Epimorphismus $e : \mathbb{A} \to \mathbb{A}'$ existiert. □

Definition 2.26 (Eigentliche Quotienten(co)algebra)**.** Sei $\mathbb{A}$ eine Σ-(Co)Algebra und $\mathbb{A}' \subseteq \mathbb{A}$ eine abstrakte Quotienten(co)algebra. $\mathbb{A}'$ heißt *eigentliche Quotienten(co)algebra* (geschrieben $\mathbb{A}' \parallel \mathbb{A}$), falls gilt:

1. A' ist Partition von A,
2. die zugehörige Partitionierungsabbildung ist ein Σ-(Co)Algebra-Morphismus $\mathrm{in} : \mathbb{A} \to \mathbb{A}'$. □

Lemma 2.27. *Sei $\mathbb{A} = (A, \alpha)$ eine Σ-(Co)Algebra. Zu jeder Partition A' von A existiert höchstens eine Operation α', so daß $(A', \alpha') \parallel \mathbb{A}$.* [55]

Lemma 2.28. *Sei $\mathbb{A}$ eine Σ-(Co)Algebra. Zu jeder abstrakten Quotienten(co)algebra $\mathbb{A}' \mid \mathbb{A}$ existiert genau eine isomorphe eigentliche Quotienten(co)algebra $\mathbb{A}'' \parallel \mathbb{A}$.* [55]

Beispiel 2.29. Die bekannte Σ_P-Coalgebra $\mathbb{D}_P$ besitzt einen eigentlichen Quotienten:

$$\mathbb{D}'_P = (D'_P, \delta'_P)$$
$$D'_P = \big\{\{2 \cdot n\} \mid n \in \mathcal{N}\big\} \cup \big\{\{2 \cdot n + 1 \mid n \in \mathcal{N}\}\big\}$$

Man beachte den feinen Unterschied zwischen beiden Teilmengenkonstruktionen: die geraden Zahlen werden jeweils in eine eigene einelementige Äquivalenzklasse verpackt, während alle ungeraden Zahlen zu einer einzigen Äquivalenzklasse zusammengefaßt werden. Auf den Beweis der Morphismuseigenschaft wird hier verzichtet. □

Definition 2.30 (Einfachheit)**.** Eine Σ-(Co)Algebra $\mathbb{A}$ heißt *einfach*, wenn sie bis auf Isomorphie keine Quotienten(co)algebren besitzt, das heißt, wenn alle Epimorphismen $m : \mathbb{B} \to \mathbb{A}$ iso sind. Dies ist notwendig und hinreichend dafür, daß $\mathbb{A}$ keine echten eigentlichen Quotienten(co)algebren (also keine außer sich selbst) besitzt. [55]

Satz 2.31. *Jede finale Σ-Coalgebra ist einfach.* [55]

Unter den Σ-Algebren sind nur die einelementigen einfach.

Die Quotienten einer initialen Σ-Algebra sind von besonderem Interesse, da sie die Modelle von algebraischen Spezifikationen mit *Gleichungen* darstellen.

Beispiel 2.32. Die *Restklassen modulo m* bilden einen eigentlichen Quotienten der bekannten initialen Σ_P-Algebra $\mathbb{A}_P$. Dabei ergibt sich die Partitionierung schon aus der Gleichung $0 \equiv m$ durch Kongruenzbildung bezüglich α_P. Auf Kongruenzen soll aber hier nicht weiter eingegangen werden. □

2.1.4.3 Inversion

Ist die Operation einer Algebra oder Coalgebra umkehrbar, dann ergibt sich daraus ein inverses Objekt.

Definition 2.33 (Inverse (Co)Algebra)**.** Sei $\mathbb{A} = (A, \alpha)$ eine Σ-Algebra. Ist α iso, dann ist $\mathbb{A}^{-1} = (A, \alpha^{-1})$ eine Σ-Coalgebra. Dual ergibt sich eine Σ-Algebra durch Inversion einer Σ-Coalgebra. □

Im Allgemeinen gilt nicht, daß durch Inversion aus einem initialen ein finales Objekt wird, oder umgekehrt. Des weiteren gilt zwar folgender Satz, nicht aber seine Dualisierung:

Satz 2.34. *Sei $\mathbb{I}$ eine initiale Σ-Algebra und $\mathbb{F}$ eine finale Σ-Coalgebra. Dann gilt $\mathbb{I}^{-1} \subseteq \mathbb{F}$.*

Beweis. Da $\mathbb{I}$ minimal ist, ist der Anamorphismus $[\![\mathbb{I}^{-1}]\!]$ mono. □

Beispiel 2.35. Die Σ_P-Coalgebren $\mathbb{A}_P^{-1}$ und $\mathbb{C}_P$ unterscheiden sich genau um das Element ω. □

Definition 2.36. Sei $\mathbb{I}$ eine initiale Σ-Algebra und $\mathbb{F} = (F, \phi)$ eine finale Σ-Coalgebra. Das Bild $F_0 \subseteq F$ des Anamorphismus $[\![\mathbb{I}^{-1}]\!]$ heißt *algebraischer Anteil* von F. Dieser definiert eine eigentliche Untercoalgebra $\mathbb{F}_0 \subsetneqq \mathbb{F}$. □

Beispiel 2.37. ω ist in diesem Sinne eine nichtalgebraische Zahl. □

Satz 2.38. *Für jede finale Σ-Coalgebra $\mathbb{F}$ ist die Untercoalgebra $\mathbb{F}_0$ eindeutig, unabhängig von der Wahl von $\mathbb{I}$.*

Beweis. Seien $\mathbb{I}, \mathbb{I}'$ zwei initiale Σ-Algebren. Dann existiert ein Isomorphismus $\iota : \mathbb{I} \to \mathbb{I}'$, so daß $[\![\mathbb{I}]\!] = [\![\mathbb{I}']\!] \circ \iota$. Es gilt $\mathrm{codom}[\![\mathbb{I}]\!] = \mathrm{codom}([\![\mathbb{I}']\!] \circ \iota) = \mathrm{codom}[\![\mathbb{I}']\!]$. □

```
class Functor σ ⇒ Signature σ where
  (==*)        :: Eq α ⇒ σ α → σ α → Bool
  ffold        :: (α → β → α) → α → σ β → α
  fmapM        :: Monad μ ⇒ (α → μ β) → σ α → μ (σ β)
  fzipWithM_   :: Monad μ ⇒ (α → β → μ ()) → σ α → σ β → μ ()
```

Programm 2.1: Signaturfunktoren

2.2 Signaturfunktoren für Daten

Programm 2.1 zeigt die Typklasse *Signature*, welche alle Instanzen von Signaturfunktoren in der Haskell-Spezifikation subsumieren soll. Diese deklariert als Methoden einige generische Traversierungsoperationen, um die Elemente der (Co)Algebren unabhängig von der durch die Signatur gegebenen Struktur transformieren und auswerten zu können. Mit Hilfe dieser Operationen wird es möglich sein, die restlichen Programmteile dieser Arbeit generisch für alle derart modellierten Signaturen zu formulieren. Zur Beschreibung der Operationen müssen wir der Formalisierung in den folgenden Abschnitten zunächst intuitiv vorgreifen:

Ein Signaturfunktor Σ definiert eine *Struktur* (Menge mit Operationen/Relationen) über seiner Argumentmenge. Die Operationen von Σ-Algebren und Σ-Coalgebren übersetzen zwischen den *Elementen* einer Trägermenge X und Datensätzen aus $\Sigma(X)$. Diese Datensätze setzen sich zusammen aus einem durch Σ strukturierten *lokalen* Anteil und Vorkommen weiterer Elemente aus X, genannt *Referenzen*.

Die Aussage, daß einem Element x ein Datensatz mit lokalem Anteil A und Referenzen $(y_i)_{i\in I}$ zugeordnet ist, schreiben wir

$$x \mathrel{\widehat{=}} A\big((y_i)_{i\in I}\big)$$

Dabei ist I nicht im Allgemeinen endlich oder total geordnet. Für eine funktionale Modellierung in Haskell nehmen wir allerdings an, daß I endlich sei und für jede betrachtete Signatur eine kanonische totale Anordnung besitze, so das sich die Familie $(y_i)_{i\in I}$ als Liste von Referenzen darstellen läßt. Unter der pragmatischen Randbedingung, daß Signaturen zur Beschreibung von Datenstrukturen dienen, sind diese Einschränkungen adäquat.

Die Methoden der Typklasse *Signature* lassen sich drei Bereichen zuordnen:

1. Transformation eines Elementes. Hierzu dient die von der Typklasse *Functor* geerbte, pur funktionale Methode *fmap* und deren monadisches Pendant in der Kleisli-Kategorie, *fmapM*. Beide beschreiben die Anwendung einer Transformationsfunktion auf alle Elemente des referentiellen Anteils; die lokale Struktur bleibt unangetastet:

$$\begin{aligned} x \mathrel{\widehat{=}} A(y_1, \ldots, y_n) \implies \mathit{fmap}\, f\, x &\mathrel{\widehat{=}} A(f\, y_1, \ldots, f\, y_n) \\ \mathit{fmapM}\, f\, x &\mathrel{\widehat{=}} \mathbf{do} \\ &\quad y'_1 \leftarrow f\, y_1 \\ &\quad \vdots \\ &\quad y'_n \leftarrow f\, y_n \\ &\quad \mathit{return}\, A(y'_1, \ldots, y'_n) \end{aligned}$$

2. Auswertung eines Elementes. Die Methode `ffold` beschreibt die Reduktion des referentiellen Anteils in einem Monoid, also mittels einer assoziativen binären Operation und eines neutralen Elementes. Die lokale Struktur wird ignoriert:

$$x \mathrel{\widehat{=}} A(y_1, \ldots, y_n) \implies \mathit{ffold}\, (\oplus)\, e\, x = e \oplus y_1 \oplus \cdots \oplus y_n$$

```
syntacticEquiv :: (Signature σ, Eq χ) ⇒ Equiv (σ χ)
syntacticEquiv = Equiv (==*)
```

Programm 2.2: Induzierte syntaktische Gleichheit

```
type Cons σ α    = σ α → α
type Decons σ α  = α → σ α
class Signature σ ⇒ Dialgebra σ α where
  cons   :: Cons σ α
  decons :: Decons σ α

kata   :: Dialgebra σ ι ⇒ Cons σ α → ι → α
kata f = f ∘ fmap (kata f) ∘ decons
ana    :: Dialgebra σ φ ⇒ Decons σ α → α → φ
ana f  = cons ∘ fmap (ana f) ∘ f
```

Programm 2.3: Dialgebren, Kata- und Anamorphismen

3. Verknüpfung zweier Elemente. Die Methode $(==^*)$ beschreibt die Anhebung der inhärenten Gleichheit eines Elementtyps auf die Strukturebene. Die Methode *fzipWithM_* beschreibt die monadische simultane Traversierung zweier Elemente mit identischer Struktur und Anwendung einer binären monadischen Operation auf alle Paare von an gleicher Stelle vorkommenden Referenzen. Sei also:

   $x \mathrel{\widehat{=}} A(y_1, \ldots, y_n)$,$x' \mathrel{\widehat{=}} A(y'_1, \ldots, y'_n)$

   Dann gelte:

   $$x ==^* x' = y_1 == y'_1 \mathbin{\&\&} \ldots \mathbin{\&\&} y_n == y'_n$$
   $$fzipWithM\_\ (\oplus)\ x\ x' = y_1 \oplus y'_1 \gg \cdots \gg y_n \oplus y'_n$$

   Für verschieden strukturierte Elemente ist $(==^*)$ konstant *False*, während *fzipWithM_* undefiniert ist. Die strukturelle Gleichheit $(==^*)$ definiert, wie in Programm 2.2 gezeigt, für jeden Elementtyp mit Gleichheit eine Äquivalenzrelation.

2.2.1 Initiale und Finale Modelle

Für die Modellierung in Haskell nutzen wir die bereits erwähnte Tatsache, daß initiale Algebra und finale Coalgebra dort identisch sind. Folglich dient ein Haskell-Datentyp als Modell beider, einer sogenannten *Dialgebra*. Programm 2.3 zeigt die Modellierung als Typklasse, sowie die generische Implementierung von Kata- und Anamorphismen durch Umformung der Definitionsgleichungen.

```
threadData :: Signature σ ⇒ σ χ → [χ]
threadData = reverse ∘ ffold (flip (:)) []
succData   :: (Signature σ, Ord χ) ⇒ σ χ → Set χ
succData   = ffold addToSet emptySet
```

Programm 2.4: Erreichbare Umgebung

2.3 Relationen auf Coalgebren

In diesem Abschnitt sollen Beziehungen zwischen den Elementen von Coalgebren untersucht werden. Da die coalgebraische Meta-Theorie einen Mittelweg zwischen Algebra und Graphtheorie beschreiben soll, werden strukturelle Begriffe wie *Teilterm*, *Termtiefe*, *Adjazenz* und *Knotenattributierung* auf elementare coalgebraische Konzepte abgebildet. Auch die Frage der semantischen Äquivalenz verschiedener graphischer Repräsentationen eines zyklischen Datums wird geklärt.

2.3.1 Erreichbarkeit

In der klassischen, syntaxorientierten Algebra sind die Begriffe *Teil* und *Ganzes* intuitiv und anschaulich durch die Teiltermbeziehung festgelegt. Beim Übergang zur kategoriellen Coalgebra ergeben sich zwei Schwierigkeiten:

1. In der Coalgebra wird auf Grund von Zyklen die Rollentrennung von Teil und Ganzem hinfällig. Es existiert jedoch eine duale Relation, die am besten als *Erreichbarkeit* bezeichnet werden kann.
2. Durch den Verzicht auf Syntax ist ein abstraktes Kriterium für diese Relation nötig. Hier kann auf kausale Zusammenhänge aufgebaut werden.

Definition 2.39 (Kausale Erreichbarkeit)**.** Sei $\mathbb{C} = (C, \gamma)$ eine Σ-Coalgebra. Sei $x \in C$. Ein Teilträger $A \subseteq C$ heißt *x-beschreibend*, genau dann wenn $\gamma(x) \in \Sigma(A)$. Ein Element $y \in C$ heißt von x aus *unmittelbar erreichbar*, geschrieben $x \to_{\mathbb{C}} y$, genau dann wenn jeder x-beschreibende Teilträger y enthält.

$$\forall\, A \subseteq C.\ \gamma(x) \in \Sigma(A) \implies y \in A$$

Die *mittelbare* Erreichbarkeitsrelation $\to^*_{\mathbb{C}}$ ist der reflexive transitive Abschluß dieser Relation. Wir schreiben:

$$\mathfrak{s}_{\mathbb{C}}(x) = \{y \mid x \to_{\mathbb{C}} y\} \qquad \mathfrak{p}_{\mathbb{C}}(y) = \{x \mid x \to_{\mathbb{C}} y\}$$
$$\mathfrak{s}^*_{\mathbb{C}}(x) = \{y \mid x \to^*_{\mathbb{C}} y\} \qquad \mathfrak{p}^*_{\mathbb{C}}(y) = \{x \mid x \to^*_{\mathbb{C}} y\}$$
$$\mathfrak{s}^+_{\mathbb{C}}(x) = \{y \mid x \to^*_{\mathbb{C}} y\} \qquad \mathfrak{p}^+_{\mathbb{C}}(y) = \{x \mid x \to^*_{\mathbb{C}} y\}$$

Dabei nennen wir $\mathfrak{s}_{\mathbb{C}}(x)/\mathfrak{s}^*_{\mathbb{C}}(x)$ *unmittelbar* bzw. *mittelbar erreichbare Umgebung* von x. Ein Element x mit $\mathfrak{s}_{\mathbb{C}}(x) = \emptyset$ heißt *Blatt*. Ein Element x mit $\mathfrak{p}_{\mathbb{C}}(x) = \emptyset$ heißt *Wurzel*. □

Beispiel 2.40. Die unmittelbare Erreichbarkeitsrelation der bekannten Σ_P-Coalgebren sind:

$$n + 1 \to_{\mathbb{C}_P} n \qquad 2 \cdot n \to_{\mathbb{D}_P} n$$
$$\omega \to_{\mathbb{C}_P} \omega$$

□

Programm 2.4 berechnet $\mathfrak{s}(x)$ als Liste oder Menge der von $x \in \Sigma(X)$ erreichbaren Elemente. Die Funktion *threadData* ist dabei die Reduktion im freien Monoid der rückwärts konstruierten Listen. Dabei wird die Traversierungsreihenfolge als Listenreihenfolge explizit. Die reihenfolgenunabhängige Darstellung als endliche Menge, wie in *succData* implementiert, setzt technisch eine totale Ordnung voraus.

Definition 2.41 (Teilterm)**.** Sei $\mathbb{A} = (A, \alpha)$ eine initiale Σ-Algebra. Die zugehörige *Teiltermrelation* ist definiert durch die Erreichbarkeitsrelation der inversen Coalgebra: Die unmittelbar erreichbaren Elemente in $\mathbb{A}^{-1}$ heißen *unmittelbare Teilterme* in $\mathbb{A}$. □

Definition 2.42 (Konstante Signatur)**.** Ein Signaturfunktor Σ heißt *konstant*, wenn für alle Σ-Coalgebren $\mathbb{C}$ gilt $(\rightarrow_{\mathbb{C}}) = \emptyset$. □

Definition 2.43 (Grad)**.** Der *Grad* eines Signaturfunktors Σ ist die kleinste endliche oder unendliche Kardinalzahl a, so daß für alle Σ-Coalgebren $\mathbb{C} = (C, \gamma)$ und für alle Elemente $x \in C$ gilt:

$$|\mathfrak{s}_{\mathbb{C}}(x)| \leq a$$ □

Lemma 2.44. *Der Grad einer konstanten Signatur ist null.*

Beispiel 2.45. Der Grad des bekannten Signaturfunktors Σ_P ist eins. □

Der Grad ist ein Maß für die Stärke der Verzweigung beim Übergang von einem Element zu einer unmittelbar erreichbaren Umgebung, und damit auch für die Größe des zugeordneten Datensatzes. Für die Modellierung realer Daten sollte diese endlich sein.

Definition 2.46 (Finitarität)**.** Ein Signaturfunktor Σ heißt *finitär*, wenn für alle Σ-Coalgebren $\mathbb{C} = (C, \gamma)$ und für alle Elemente $x \in C$ die Menge $\mathfrak{s}_{\mathbb{C}}(x)$ endlich ist. □

Lemma 2.47. *Signaturfunktoren endlichen Grades sind finitär. Finitäre Signaturfunktoren haben einen wohldefinierten Grad von höchstens* $\aleph_0$.

Das folgende Lemma soll der induktiven Argumentation über einer Coalgebra dienen. Wie schon bei Untermenge und Quotient können die mengentheoretischen Operationen des positiven (Schnitt) und negativen (Differenz) Einschränkens einer Trägermenge nur auf die Coalgebra angehoben werden, falls sie mit deren Operation verträglich sind.

Lemma 2.48 (Licht und Schatten)**.** *Sei $\mathbb{C} = (C, \gamma)$ eine Coalgebra. Sei $x \in C$ ein beliebiges Element. Dann sind $\left(\mathfrak{s}^*_{\mathbb{C}}(x), \gamma\right)$ und $\left(C \setminus \mathfrak{p}^*_{\mathbb{C}}(x), \gamma\right)$ eigentliche Untercoalgebren von $\mathbb{C}$.*

Beweis. Jede unter $\mathfrak{s}_{\mathbb{C}}$ abgeschlossene Teilmenge von C ist per Definition Träger einer eigentlichen Untercoalgebra.

1. $\mathfrak{s}^*_{\mathbb{C}}(x)$ ist per Konstruktion abgeschlossen.
2. Für alle y mit $\mathfrak{s}_{\mathbb{C}}(y) \cap \mathfrak{p}^*_{\mathfrak{C}}(x) \neq \emptyset$ gilt $y \in \mathfrak{p}^*_{\mathfrak{C}}(x)$. Also ist auch $C \setminus \mathfrak{p}^*_{\mathfrak{C}}(x)$ abgeschlossen. □

Definition 2.49 (Zyklizität)**.** Sei $\mathbb{C} = (C, \gamma)$ eine Coalgebra.

1. Ein Element $x \in C$ heißt *zyklisch*, falls $x \in \mathfrak{s}^+_{\mathbb{C}}(x)$, sonst *azyklisch*.
2. Ein Element $x \in C$ heißt *zyklenfrei*, falls alle $y \in \mathfrak{s}^*_{\mathbb{C}}(x)$ azyklisch sind.
3. Die Coalgebra $\mathbb{C}$ heißt zyklenfrei, wenn alle Elemente $x \in C$ zyklenfrei sind. □

Beispiel 2.50. Azyklische Elemente sind nicht immer zyklenfrei. Sei folgende Erreichbarkeitsrelation gegeben:

$$a \longrightarrow b \longleftrightarrow c \longrightarrow d$$

Dann sind a und d azyklisch, aber nur d zyklenfrei. □

Lemma 2.51. *Eine Coalgebra $\mathbb{C}$ ist zyklenfrei, genau dann wenn die Relation $\rightarrow^+_{\mathbb{C}}$ antisymmetrisch ist.*

Definition 2.52 (Rationalität)**.** Sei Σ ein finitärer Signaturfunktor. Eine Σ-Coalgebra $\mathbb{C} = (C, \gamma)$ heißt *rational*, wenn für alle Elemente $x \in C$ die Menge $\mathfrak{s}^*_{\mathbb{C}}(x)$ endlich ist. □

```
localData :: Signature σ ⇒ σ α → σ ()
localData = fmap (const ())
```

Programm 2.5: Lokaler Anteil

```
compatible      :: Signature σ ⇒ σ α → σ α → Bool
compatible x x' = localData x ==* localData x'
```

Programm 2.6: Kompatibilität

Satz 2.53. *Sei Σ ein finitärer Signaturfunktor und $\mathbb{I}$ eine initiale Σ-Algebra. Die inverse Σ-Coalgebra $\mathbb{I}^{-1}$ ist rational.*

Beweis. Dual zu Lemma 2.48. Sei $\mathbb{I} = (I, \iota)$. Die Teilmenge $I_0 = \{x \in I \mid \mathfrak{s}^*_{\mathbb{I}^{-1}}(x) \text{ endlich}\}$ ist abgeschlossen unter $\mathfrak{p}_{\mathbb{I}^{-1}}$. Also ist I_0 Träger einer Unteralgebra von $\mathbb{I}$. Da $\mathbb{I}$ minimal ist, gilt $I_0 = I$. □

Als Modelle für reale Daten, die mit strikten Verfahren erzeugt und verarbeitet werden können, also Zustände eines strikt funktionalen Programms, betrachten wir nur rationale Coalgebren finitärer Signaturen.

2.3.2 Lokale Beziehungen

In einem coalgebraischen Kontext spielen die Elemente von Funktorbildern $s \in \Sigma(A)$ die Rolle von *Datensätzen* über einer Individuenmenge A. Die folgenden Definitionen sind über Datensätzen formuliert, lassen sich aber trivial auf Elemente von Coalgebren fortsetzen, denen durch die Operation der Coalgebra diese Datensätze zugeordnet werden.

Definition 2.54 (Lokalität)**.** Sei $s \in \Sigma(A)$ ein Datensatz und $(!) : A \to 1$ ein finaler Morphismus in **Set**. Dann heißt der Datensatz $\Sigma(!)(s) \in \Sigma(1)$, in welchem alle Individuen gleichgesetzt sind, der *lokale Anteil* von s. Gilt $\Sigma(!)(s) = s$, dann heißt s *rein lokal*. □

Beispiel 2.55. Sei die einelementige Menge $1 = \{\bullet\}$. Der lokale Anteil von Σ_P-Datensätzen über $\mathbb{C}_P$ ist $\Sigma_P(!)\big(\gamma_P(0)\big) = \star$, für $n > 0$ aber $\Sigma_P(!)\big(\gamma_P(n)\big) = \bullet$. Man kann also sagen, daß Positivität eine lokale, Betrag dagegen eine nichtlokale Eigenschaft von Zahlen ist, da positive Zahlen durch ihren lokalen Anteil zwar von Null unterscheidbar sind, aber nicht untereinander. □

Lemma 2.56. *Einem Blatt ist stets ein rein lokaler Datensatz zugeordnet:*

$$\mathfrak{s}_{\mathbb{C}}(x) = \emptyset \implies \Sigma(!)\big(\gamma(x)\big) = \gamma(x)$$

Für einen rein lokalen Datensatz s gilt unter allen Abbildungen f:

$$\Sigma(f)(s) = s$$

Definition 2.57 (Kompatibilität)**.** Zwei Datensätze $x \in \Sigma(A)$ und $x' \in \Sigma(A')$ heißen *kompatibel*, geschrieben $x \asymp_\Sigma x'$, falls ihre lokalen Anteile übereinstimmen:

$$x \asymp_\Sigma x' \iff \Sigma(!)(x) = \Sigma(!)(x') \qquad \square$$

Lemma 2.58. *Kompatible, rein lokale Datensätze sind identisch.*

```
type ITrans α β = [(α, β)]
transData       :: Signature σ ⇒ σ α → σ β → ITrans α β
transData x x'  = execState (fzipWithM_ update x x') []
  where
    update      :: α → β → State (ITrans α β) ()
    update y y' = modify ((y, y') :)
transCover      :: Monad μ ⇒ (α → β → μ ()) → ITrans α β → μ ()
transCover      = mapM_ ∘ uncurry
```

Programm 2.7: Individualtransformation

Lemma 2.59. *fzipWithM_ (⊕) x x' ist definiert, genau dann wenn x und x' kompatibel sind.*

Die vorausgehenden Definitionen betreffen den lokalen Anteil von Datensätzen, die folgenden dagegen komplementär den referentiellen Anteil. Dabei genügt es für den Rest der Arbeit, funktionale Beziehungen zu betrachten.

Definition 2.60 (Individualtransformation)**.** Seien $x \in \Sigma(A)$ und $x' \in \Sigma(A')$ kompatible Datensätze. Eine partielle Abbildung $f : A \nrightarrow A'$ heißt *Individualtransformation* zwischen x und x', geschrieben $f : x \to x'$, falls $\Sigma(f)(x) = x'$ gilt und f minimal ist, das heißt, keine Teilabbildung $g \sqsubset f$ mit derselben Eigenschaft existiert. □

Eine Individualtransformation $f : x \to x'$ übersetzt also unter dem Funktor Σ den referentiellen Anteil von x in den von x', wobei der lokale Anteil (dies ist die Funktoreigenschaft) erhalten bleibt.

Beispiel 2.61. Sei $\mathbb{C}_P$ die bekannte Σ_P-Coalgebra. Die einelementige partielle Abbildung f mit $f(0) = 1$ ist Individualtransformation zwischen den Datensätzen $\gamma_P(1)$ und $\gamma_P(2)$. □

Programm 2.7 modelliert Individualtransformationen extensional als Listen von Paaren. Die Berechnungsfunktion *transData* ist als monadische Operation definiert, die im Seiteneffekt die Liste aufbaut. Die Traversierungsreihenfolge ist als Listenreihenfolge explizit. Die Funktion *transCover* dient der Traversierung einer Individualtransformation mit einer monadischen Operation und wird in Kapitel 4 eine wichtige Rolle spielen.

Definition 2.62 (Prototyppräordnung)**.** Seien $x \in \Sigma(A)$ und $x' \in \Sigma(A')$ kompatible Datensätze. Dann heißt x *Prototyp* von x', geschrieben $x \trianglelefteq x'$, falls eine Individualtransformation $f : x \to x'$ existiert. □

Lemma 2.63. *Die Relation $\trianglelefteq$ ist reflexiv und transitiv.*

Definition 2.64 (Initialität)**.** Ein Datensatz $x \in \Sigma(A)$ heißt *initial*, falls zu jedem kompatiblen Datensatz $x' \in \Sigma(A')$ genau eine Individualtransformation $f : x \to x'$ existiert. Diese schreiben wir auch $x \Rightarrow x'$. □

2.3.3 Globale Beziehungen

Definition 2.65 (Bisimulation)**.** Seien $\mathbb{C}_1 = (C_1, \gamma_1)$ und $\mathbb{C}_2 = (C_2, \gamma_2)$ zwei Σ-Coalgebren. Eine Σ-Coalgebra $\mathbb{R} = (R, \rho)$ mit $R \subseteq C_1 \times C_2$ heißt *Bisimulation* zwischen $\mathbb{C}_1$ und $\mathbb{C}_2$, geschrieben $\mathbb{R} : \mathbb{C}_1 \leftrightarrow \mathbb{C}_2$, falls die Projektionen $\pi_1 : R \to C_1$ und $\pi_2 : R \to C_2$ Σ-Coalgebra-Morphismen sind:

$$\begin{array}{ccccc} C_1 & \xleftarrow{\pi_1} & R & \xrightarrow{\pi_2} & C_2 \\ \downarrow{\scriptstyle \gamma_1} & & \downarrow{\scriptstyle \rho} & & \downarrow{\scriptstyle \gamma_2} \\ \Sigma(C_1) & \xleftarrow{\Sigma(\pi_1)} & \Sigma(R) & \xrightarrow{\Sigma(\pi_2)} & \Sigma(C_2) \end{array}$$

Der Träger R kann als Relation zwischen C_1 und C_2 aufgefaßt werden. [55]

Definition 2.66 (Bisimilarität)**.** Seien $\mathbb{C}_1 = (C_1, \gamma_1)$ und $\mathbb{C}_2 = (C_2, \gamma_2)$ zwei Σ-Coalgebren. Zwei Elemente $x_1 \in C_1$, $x_2 \in C_2$ heißen *bisimilar*, geschrieben $x_1 \sim x_2$, falls eine Bisimulation $\mathbb{R} = (R, \rho)$ mit $\mathbb{R} : \mathbb{C}_1 \leftrightarrow \mathbb{C}_2$ existiert, so daß $(x_1, x_2) \in R$. [55]

Es ist nicht auf den ersten Blick ersichtlich, unter welchen Bedingungen Bisimilarität entscheidbar ist. Daher wird hier kein entsprechendes Programm angegeben. Ein effektives Verfahren für die in Kürze zu definierende Klasse von Signaturfunktoren wird sich als nützliches Korollar der in Kapitel 5 angestellten Überlegungen ergeben.

Beispiel 2.67. Sei $\mathbb{D}_P$ die bekannte Σ_P-Coalgebra. Alle ungeraden Zahlen sind hier bisimilar. □

Das folgende Kriterium ist nur notwendig, nicht hinreichend für Bisimilarität.

Lemma 2.68. *Bisimilaren Elementen sind kompatible Datensätze zugeordnet:*

$$x_1 \sim x_2 \implies \gamma_1(x_1) \asymp_\Sigma \gamma_2(x_2)$$

Beweis. Es existiert ein $r \in R$, so daß:

$$\begin{aligned}
(x_1, x_2) &= \langle \pi_1, \pi_2 \rangle(r) \\
(\gamma_1 \times \gamma_2)(x_1, x_2) &= (\gamma_1 \times \gamma_2)\big(\langle \pi_1, \pi_2 \rangle(r)\big) \\
&= \langle \gamma_1 \circ \pi_1, \gamma_2 \circ \pi_2 \rangle(r) \\
&= \big(\langle \Sigma(\pi_1), \Sigma(\pi_2) \rangle \circ \rho\big)(r) \\
\Big(\big(\Sigma(!) \times \Sigma(!)\big) \circ \big(\gamma_1 \times \gamma_2\big)\Big)(x_1, x_2) &= \Big(\big(\Sigma(!) \times \Sigma(!)\big) \circ \langle \Sigma(\pi_1), \Sigma(\pi_2) \rangle \circ \rho\Big)(r) \\
&= \big(\langle \Sigma(!) \circ \Sigma(\pi_1), \Sigma(!) \circ \Sigma(\pi_2) \rangle \circ \rho\big)(r) \\
&= \big(\langle \Sigma(! \circ \pi_1), \Sigma(! \circ \pi_2) \rangle \circ \rho\big)(r) \\
&= \big(\langle \Sigma(!), \Sigma(!) \rangle \circ \rho\big)(r) \\
\gamma_1(x_1) &\asymp_\Sigma \rho(r) \asymp_\Sigma \gamma_2(x_2)
\end{aligned}$$

□

Die Existenz finaler Coalgebren erlaubt die folgende, einfache Formulierung von Bisimilarität.

Satz 2.69. *Zwei Elemente sind genau dann bisimilar, wenn ihre finale Semantik übereinstimmt.*

Sei Σ ein Signaturfunktor mit finalen Coalgebren und $\mathbb{C} = (C, \gamma)$ und $\mathbb{D} = (D, \delta)$ zwei Σ-Coalgebren. Dann gilt für alle $c \in C, d \in D$:

$$c \sim d \iff [\![\mathbb{C}]\!](c) = [\![\mathbb{D}]\!](d)$$ [55]

Die folgende Eigenschaft wird in Kapitel 4 nützlich sein.

Lemma 2.70. *Sei Σ ein Signaturfunktor. Sei $\mathbb{C} = (C, \gamma)$ eine beliebige Σ-Coalgebra. Dann gilt:*

$$\operatorname{Ker} \gamma \subseteq \operatorname{Ker}[\![\mathbb{C}]\!]$$

Beweis. Sei $(x, y) \in \operatorname{Ker} \gamma$. Dann läßt sich eine Bisimulation (R, ρ) konstruieren:

$$\begin{aligned}
R &= \{(x, y)\} \cup \{(z, z) \mid z \in \mathfrak{s}^+_{\mathbb{C}}(x) = \mathfrak{s}^+_{\mathbb{C}}(y)\} \\
\rho(x, y) &= \gamma(x) = \gamma(y) \\
\rho(z, z) &= \gamma(z)
\end{aligned}$$

Also gilt $x \sim y$ und nach Satz 2.69 $[\![\mathbb{C}]\!](x) = [\![\mathbb{C}]\!](y)$. □

```
instance Signature Identity where
  Identity x ==* Identity x'          = x == x'
  ffold f e (Identity x)              = f e x
  fmapM f (Identity x)                = return ∘ Identity =≪ f x
  fzipWithM_ f (Identity x) (Identity y) = f x y
```

Programm 2.8: Identische Signatur

2.4 Polynomielle Signaturen

Der Funktor Σ spielt in der kategoriellen Deutung die Rolle der *Signatur*. Verschiedene Überlegungen legen eine Einschränkung der betrachteten Funktoren nahe:

1. Die Existenz initialer Algebren und finaler Coalgebren ist nicht für alle Funktoren gegeben.

2. Der syntaxorientierten Sicht klassischer Signaturen entsprechen nur einige spezielle Konstruktionen auf der Ebene von Objekten und Morphismen.

2.4.1 Existenz initialer Algebren und finaler Coalgebren

Beispiel 2.71. Gegeben sei der covariante Potenzfunktor $\mathcal{P}$ in **Set**. Es existiert keine initiale $\mathcal{P}$-Algebra oder finale $\mathcal{P}$-Coalgebra. Grund ist die Unlösbarkeit folgender Isomorphie in ZERMELO-FRAENKEL-Mengentheorie:

$$A \simeq \mathcal{P}(A)$$ □

Es sind verschieden akkurate hinreichende Bedingungen für die Existenz initialer Algebren und finaler Coalgebren zu einem gegebenen Funktor bekannt. Eine populäre Klasse solcher Funktoren sind die *polynomiellen* Funktoren, welche erzeugt werden durch den *identischen* Funktor, sowie alle *konstanten* Funktoren und endlichen *Produkte* und *Coprodukte*. Diese entsprechen der Mächtigkeit freier Datentypdefinitionen. Eine nützliche Erweiterung ist der KLEENE-Abschluß, welcher dem syntaktischen Konstrukt unbeschränkter Listen entspricht.

2.4.2 Kalkül der Signaturfunktoren

Die Klasse der *streng* polynomiellen Signaturfunktoren sei die kleinste Klasse, welche den folgenden vier Definitionen genügt.

Definition 2.72 (Identische Signatur)**.** Der identische Funktor $\mathcal{I}$ ist ein polynomieller Signaturfunktor.

$$\mathcal{I}(X) = X$$
$$\mathcal{I}(f) = f$$ □

Definition 2.73 (Konstante Signatur)**.** Der konstante Funktor $\mathcal{C}_A$ zu einer Menge A ist ein polynomieller Signaturfunktor.

$$\mathcal{C}_A(X) = A$$
$$\mathcal{C}_A(f) = \mathrm{id}_A$$ □

```
newtype Const α χ = Const α
instance Functor (Const α) where
  fmap f (Const a) = Const a
instance Eq α ⇒ Signature (Const α) where
  Const a ==* Const a'                    = a == a'
  ffold f e (Const a)                     = e
  fmapM f (Const a)                       = return (Const a)
  fzipWithM_ f (Const a) (Const a') | a == a' = return ()
```

Programm 2.9: Konstante Signatur

```
data (:*:) σ1 σ2 χ = (:*:) { project1 :: σ1 χ, project2 :: σ2 χ}
instance (Functor σ1, Functor σ2) ⇒ Functor (σ1 :*: σ2) where
  fmap f (x :*: y) = fmap f x :*: fmap f y
instance (Signature σ1, Signature σ2) ⇒ Signature (σ1 :*: σ2) where
  (x :*: y) ==* (x' :*: y')          = (x ==* x') && (y ==* y')
  ffold f e (x :*: y)                = ffold f (ffold f e x) y
  fmapM f (x :*: y)                  = do
                                         x' ← fmapM f x
                                         y' ← fmapM f y
                                         return (x' :*: y')
  fzipWithM_ f (x :*: y) (x' :*: y') = fzipWithM_ f x x' ≫ fzipWithM_ f y y'
```

Programm 2.10: Produkt von Signaturen

Definition 2.74 (Signatur-Produkt)**.** Sei $(\Sigma_i)_{i\in I}$ eine endliche Familie von polynomiellen Signaturfunktoren. Dann ist das punktweise Produkt $\prod_{i\in I}\Sigma_i$ ein polynomieller Signaturfunktor.

$$(\prod_{i\in I}\Sigma_i)(X) = \prod_{i\in I}(\Sigma_i(X))$$
$$(\prod_{i\in I}\Sigma_i)(f) = \prod_{i\in I}(\Sigma_i(f))$$

Als Spezialfall $I = \{1, 2\}$ ergibt sich das binäre Produkt:

$$(\Sigma_1 \times \Sigma_2)(X) = \Sigma_1(X) \times \Sigma_2(X)$$
$$(\Sigma_1 \times \Sigma_2)(f) = \Sigma_1(f) \times \Sigma_2(f)$$

Wir schreiben auch in gewohnter Exponentenform:

$$\Sigma^n = \prod_{i\in\{1,\ldots,n\}} \Sigma$$

Für den Randfall $I = \emptyset$ degeneriert das Produkt zu $\mathcal{C}_1$. □

Programm 2.10 definiert das binäre punktweise Produkt als freien Datentyp. Wir schreiben :*: für Typ- und Wertekonstruktor.

Programm 2.11 zeigt die universellen Operationen auf Produkten. Die Definitionen sind im Grunde trivial. Lediglich die Typen sind eher komplex und werden daher hier weggelassen.

```
tupling f g x = f x :*: g x
fproduct f g  = tupling (f ∘ project_1) (g ∘ project_2)
```

Programm 2.11: Universelle Operationen auf Produkten

```
data (:+:) σ_1 σ_2 χ = Inject_1 { part_1 :: σ_1 χ } | Inject_2 { part_2 :: σ_2 χ }
instance (Functor σ_1, Functor σ_2) ⇒ Functor (σ_1 :+: σ_2) where
  fmap f (Inject_1 x) = Inject_1 (fmap f x)
  fmap f (Inject_2 x) = Inject_2 (fmap f x)
instance (Signature σ_1, Signature σ_2) ⇒ Signature (σ_1 :+: σ_2) where
  Inject_1 x ==* Inject_1 x'                = x ==* x'
  Inject_2 x ==* Inject_2 x'                = x ==* x'
  _ ==* _                                   = False
  ffold f e (Inject_1 x)                    = ffold f e x
  ffold f e (Inject_2 x)                    = ffold f e x
  fmapM f (Inject_1 x)                      = fmapM f x >>= return ∘ Inject_1
  fmapM f (Inject_2 x)                      = fmapM f x >>= return ∘ Inject_2
  fzipWithM_ f (Inject_1 x) (Inject_1 x') = fzipWithM_ f x x'
  fzipWithM_ f (Inject_2 x) (Inject_2 x') = fzipWithM_ f x x'
```

Programm 2.12: Coprodukt von Signaturen

Definition 2.75 (Signatur-Coprodukt)**.** Sei $(\Sigma_i)_{i\in I}$ eine endliche Familie von polynomiellen Signaturfunktoren. Dann ist das punktweise Coprodukt $\coprod_{i\in I}\Sigma_i$ ein polynomieller Signaturfunktor.

$$(\coprod_{i\in I}\Sigma_i)(X) = \coprod_{i\in I}(\Sigma_i(X))$$
$$(\coprod_{i\in I}\Sigma_i)(f) = \coprod_{i\in I}(\Sigma_i(f))$$

Als Spezialfall $I = \{1, 2\}$ ergibt sich das binäre Coprodukt:

$$(\Sigma_1 + \Sigma_2)(X) = \Sigma_1(X) + \Sigma_2(X)$$
$$(\Sigma_1 + \Sigma_2)(f) = \Sigma_1(f) + \Sigma_2(f)$$

Für den Randfall $I = \emptyset$ degeneriert das Coprodukt zu $\mathcal{C}_\emptyset$. □

Programm 2.12 definiert das binäre punktweise Coprodukt als freien Datentyp. Wir schreiben :+: für den Typkonstruktor.

Programm 2.13 zeigt die universellen Operationen auf Coprodukten. Diese sind wiederum trivial bis auf ihre Typen.

Beispiel 2.76. Der bekannte Signaturfunktor Σ_P läßt sich nun punktfrei als $\mathcal{C}_1 + \mathcal{I}$ schreiben. □

2.4.2.1 Verallgemeinerung

Die Klasse der *verallgemeinert* polynomiellen Signaturen erlaubt außerdem die folgende Definition.

Definition 2.77. Sei Σ ein polynomieller Signaturfunktor. Dann ist der punktweise Listenabschluß Σ^*

```
cotupling f g (Inject1 x) = f x
cotupling f g (Inject2 x) = g x
fcoproduct f g            = cotupling (Inject1 ∘ f) (Inject2 ∘ g)
```

Programm 2.13: Universelle Operationen auf Coprodukten

```
newtype List σ χ = List [σ χ]
instance Functor σ ⇒ Functor (List σ) where
  fmap f (List xs) = List (map (fmap f) xs)
instance Signature σ ⇒ Signature (List σ) where
  List [] ==* List []           = True
  List (x : ys) ==* List (x' : ys') = (x ==* x') && (List ys ==* List ys')
  List _ ==* List _             = False
  fmapM f (List xs)             = mapM (fmapM f) xs >>= return ∘ List
  ffold f e (List xs)           = foldl (ffold f) e xs
  fzipWithM_ f (List xs) (List xs') = zipWithM_ (fzipWithM_ f) xs xs'
```

Programm 2.14: Listenabschluß von Signaturen

ebenfalls ein polynomieller Signaturfunktor.

$$\Sigma^*(X) = \Sigma(X)^*$$
$$\Sigma^*(f) = f^*$$

□

Programm 2.14 definiert den entsprechenden freien Datentyp.

In den folgenden Abschnitten werden, falls nicht explizit eingeschränkt, immer verallgemeinerte polynomielle Signaturen betrachtet. Alle folgenden Sätze über polynomielle Signaturen sind induktiv über der Struktur der Klasse unmittelbar nachweisbar. Die Ausführung der Fälle wird aus Gründen der Platzersparnis weggelassen.

2.4.2.2 Eigenschaften polynomieller Signaturfunktoren

Satz 2.78. *Polynomielle Signaturfunktoren sind* monoton*: Sei Σ ein polynomieller Signaturfunktor. Seien X, Y zwei Mengen. Dann gilt:*

$$X \subseteq Y \implies \Sigma(X) \subseteq \Sigma(Y)$$

Beweis. Durch strukturelle Induktion in Σ. □

Korollar 2.79. *Die Unterklasse der konstanten Signaturfunktoren ist unter Produkt, Coprodukt und Listenbildung abgeschlossen:*

$$\prod_{i\in I} \mathcal{C}_{A_i} = \mathcal{C}_{\prod_{i\in I} A_i} \qquad \coprod_{i\in I} \mathcal{C}_{A_i} = \mathcal{C}_{\coprod_{i\in I} A_i} \qquad {\mathcal{C}_A}^* = \mathcal{C}_{A^*}$$

Lemma 2.80. *Polynomielle Signaturfunktoren Σ sind monoton bezüglich der Kernrelation.*

$$\operatorname{Ker} f \subseteq \operatorname{Ker} g \implies \operatorname{Ker} \Sigma(f) \subseteq \operatorname{Ker} \Sigma(g)$$

Beweis. Durch strukturelle Induktion in Σ. □

Lemma 2.81. *Polynomielle Signaturfunktoren Σ kommutieren mit der Bildung von Urbildmengen. Sei $f : A \to B$ eine Abbildung.*

$$\mathrm{dom}_{\Sigma(B)}(\Sigma(f)) = \Sigma(\mathrm{dom}_B f)$$

Beweis. Durch strukturelle Induktion in Σ. □

Lemma 2.82. *Polynomielle Signaturfunktoren Σ kommutieren mit der Bildung von Wertemengen. Sei $f : A \to B$ eine Abbildung.*

$$\mathrm{codom}_{\Sigma(A)} \Sigma(f) = \Sigma(\mathrm{codom}_A f)$$

Beweis. Durch strukturelle Induktion in Σ. □

Satz 2.83. *Coalgebra-Homomorphismen über polynomiellen Signaturen erhalten Erreichbarkeit. Sei Σ ein polynomieller Signaturfunktor und $\mathbb{C} = (C, \gamma)$ und $\mathbb{D} = (D, \delta)$ zwei Σ-Coalgebren. Sei $f : \mathbb{C} \to \mathbb{D}$ ein Homomorphismus. Dann gilt für alle $x, y \in C$:*

$$x \to_{\mathbb{C}} y \iff f(x) \to_{\mathbb{D}} f(y)$$

Beweis. Zwei Richtungen sind zu zeigen:

1. $x \to_{\mathbb{C}} y \implies f(x) \to_{\mathbb{D}} f(y)$. Seien $x, y \in C$ so daß $x \to_{\mathbb{C}} y$. Sei $B \subseteq D$ beliebig. Es gelte:

   $$\delta\big(f(x)\big) \in \Sigma(B)$$

   Aus der Homomorphismuseigenschaft von f folgt:

   $$\begin{aligned} \Sigma(f)\big(\gamma(x)\big) &\in \Sigma(B) \\ \gamma(x) &\in \mathrm{dom}_{\Sigma(B)} \Sigma(f) \\ \gamma(x) &\in \Sigma(\mathrm{dom}_B f) && \text{(Lemma 2.81)} \\ y &\in \mathrm{dom}_B f && (x \to_{\mathbb{C}} y) \\ f(y) &\in B \end{aligned}$$

2. $f(x) \to_{\mathbb{D}} f(y) \implies x \to_{\mathbb{C}} y$. Dual mit Lemma 2.82. □

Definition 2.84 (Endlichkeit)**.** Ein konstanter Signaturfunktor $\mathcal{C}_A$ heißt *endlich*, falls die Menge A endlich ist. Ein streng polynomieller Signaturfunktor Σ heißt *endlich*, wenn er aus Identität, Produkt, Summe und endlichen Konstanten konstruiert werden kann.

Satz 2.85. *Sei Σ ein endlicher polynomieller Signaturfunktor. Sei X eine endliche Menge. Dann ist $\Sigma(X)$ ebenfalls endlich.*

Beweis. Durch strukturelle Induktion in Σ. □

Definition 2.86 (Abzählbarkeit)**.** Ein konstanter Signaturfunktor $\mathcal{C}_A$ heißt *abzählbar*, falls die Menge A abzählbar ist. Ein polynomieller Signaturfunktor Σ heißt abzählbar, wenn er aus Identität, Produkt, Summe, Listenabschluß und abzählbaren Konstanten konstruiert werden kann.

Satz 2.87. *Sei Σ ein abzählbarer polynomieller Signaturfunktor. Sei X eine abzählbare Menge. Dann ist $\Sigma(X)$ ebenfalls abzählbar.*

Beweis. Durch strukturelle Induktion in Σ. □

2.4.2.3 Erreichbarkeit

Definition 2.88 (Syntaktische Erreichbarkeit)**.** Die in 2.39 kausal definierte Erreichbarkeitsumgebung läßt sich auch induktiv über der Struktur der polynomiellen Signaturfunktoren konstruieren. Die Klasse der Abbildungen $\bar{\mathfrak{s}}(\Sigma) : \Sigma(A) \to \mathcal{P}(A)$ für alle Signaturen Σ und Träger A ist definiert als:

$$\bar{\mathfrak{s}}(\mathcal{I})(x) = \{x\} \qquad \bar{\mathfrak{s}}\left(\prod_{i\in I}\Sigma_i\right)(x) = \left[\bar{\mathfrak{s}}(\Sigma_i)\right]_{i\in I}(x)$$

$$\bar{\mathfrak{s}}(\mathcal{C}_A)(x) = \emptyset \qquad \bar{\mathfrak{s}}\left(\prod_{i\in I}\Sigma_i\right)(x) = \bigcup_{i\in I}\bar{\mathfrak{s}}(\Sigma_i)\big(\pi_i(x)\big)$$

$$\bar{\mathfrak{s}}(\Sigma^*)([]) = \emptyset \qquad \bar{\mathfrak{s}}(\Sigma^*)(x:\vec{r}) = \bar{\mathfrak{s}}(\Sigma)(x) \cup \bar{\mathfrak{s}}(\Sigma^*)(\vec{r})$$

□

Die kausale Definition ist jedoch unabhängig von der betrachteten Klasse der Signaturfunktoren, und daher als allgemeinere Lösung vorzuziehen.

Satz 2.89 (Äquivalenz)**.** *Sei Σ ein polynomieller Signaturfunktor und $\mathbb{C} = (C, \gamma)$ eine Σ-Coalgebra. Dann gilt:*

$$\mathfrak{s}_{\mathbb{C}} = \bar{\mathfrak{s}}(\Sigma) \circ \gamma$$

Beweis. Durch strukturelle Induktion in Σ. □

Satz 2.90. *Sei Σ ein polynomieller Signaturfunktor. Sei $\mathcal{P}$ der kovariante Potenzfunktor. Dann ist $\bar{\mathfrak{s}}(\Sigma) : \Sigma \Rightarrow \mathcal{P}$ eine natürliche Transformation:*

$$\begin{array}{ccc} \Sigma(A) & \xrightarrow{\Sigma(f)} & \Sigma(B) \\ \Big\downarrow{\scriptstyle \bar{\mathfrak{s}}(\Sigma)} & & \Big\downarrow{\scriptstyle \bar{\mathfrak{s}}(\Sigma)} \\ \mathcal{P}(A) & \xrightarrow[\mathcal{P}(f)]{} & \mathcal{P}(B) \end{array}$$

Beweis. Durch strukturelle Induktion in Σ. □

Die Begriffe Finitarität und Grad bestimmen den Verzweigungsgrad von Pfaden in der Graphstruktur eines Elements und damit die Komplexität seiner Traversierung. Daher sind sie aus algorithmischer Sicht von besonderem Interesse.

Satz 2.91. *Polynomielle Signaturfunktoren sind finitär.*

Beweis. Durch strukturelle Induktion in Σ. □

Definition 2.92 (Null)**.** Ein polynomieller Signaturfunktor Σ heißt *null*, genau dann wenn gilt $\Sigma = \mathcal{C}_\emptyset$. □

Satz 2.93 (Nullstellen)**.** *Sei Σ ein polynomieller Signaturfunktor, der nicht null ist. Dann gilt:*

$$\Sigma(X) = \emptyset \implies X = \emptyset$$

Beweis. Durch strukturelle Induktion in Σ. □

Satz 2.94. *Der Grad eines polynomiellen Signaturfunktors ist wohldefiniert und läßt sich induktiv berechnen. Der Grad eines streng polynomiellen Signaturfunktors ist endlich.*

Beweis. Durch strukturelle Induktion in Σ:

$$g(\mathcal{C}_A) = 0 \qquad g\big(\prod_{i\in I}\Sigma_i\big) = \begin{cases} 0 & \prod_{i\in I}\Sigma_i \text{ ist null} \\ \sum_{i\in I} g(\Sigma_i) & \text{sonst} \end{cases}$$

$$g(\mathcal{I}) = 1 \qquad g\big(\coprod_{i\in I}\Sigma_i\big) = \max_{i\in I} g(\Sigma_i)$$

$$g(\Sigma^*) = \omega$$

□

2.4.2.4 Linearität

Der folgende Lineritätsbegriff ist an den formal-algebraischen oder logischen angelehnt. Linearität in diesem Sinne bedeutet, daß die Identitäten unmittelbar erreichbarer Elemente auch als eindeutige Bezeichner der Stelle ihres Vorkommens in einem lokalen Datensatz dienen können. Damit wird die Addressierung einzelner Referenzen in komplexen Ausdrücken möglich.

Definition 2.95 (Linearer Datensatz)**.** Die Klasse der *linearen* Datensätze ist wie folgt induktiv über der Struktur der polynomiellen Signaturfunktoren definiert:

1. Jeder Datensatz $a \in A$ ist linear in $\mathcal{C}_A(X)$.
2. Jeder Datensatz $x \in X$ ist linear in $\mathcal{I}(X)$.
3. Ein Datensatz $b = \iota_i(b')$ ist linear in $\big(\coprod_{i\in I}\Sigma_i\big)(X)$, genau dann wenn b' linear in $\Sigma_i(X)$ ist.
4. Ein Datensatz $b = (b_1, \ldots, b_n)$ ist linear in $\big(\prod_{i\in I}\Sigma_i\big)(X)$, genau dann wenn alle b_i linear in $\Sigma_i(X)$ und alle $\bar{\mathfrak{s}}(\Sigma_i)(b_i)$ paarweise disjunkt sind.
5. Ein Datensatz $b = [b_1, \ldots, b_n]$ ist linear in $\Sigma^*(X)$, genau dann wenn alle b_i linear in $\Sigma(X)$ und alle $\bar{\mathfrak{s}}(\Sigma)(b_i)$ paarweise disjunkt sind. □

Satz 2.96. *Jeder lineare Σ-Datensatz ist initial.*

Beweis. Durch strukturelle Induktion in Σ. □

2.4.2.5 Bisimulation

Die Klasse der Bisimulationen ist nicht so groß, wie sie auf den ersten Blick erscheint. Die geforderte Homomorphismuseigenschaft der Projektionen schränkt die Operation derart ein, daß im Falle polynomieller Signaturfunktoren nur eine einzige, generische Lösung übrig bleibt.

Definition 2.97 (Bisimulationsoperation)**.** Sei Σ ein polynomieller Signaturfunktor. Seien C und D zwei Mengen. Dann ist die partielle Abbildung $\upsilon(\Sigma) : \Sigma(C) \times \Sigma(D) \nrightarrow \Sigma(C \times D)$ definiert als die minimale Abbildung, die folgenden Gleichungen genügt:

$$\upsilon(\mathcal{I})(x,y) = (x,y) \qquad \upsilon\big(\prod_{i\in I}\Sigma_i\big)\big((x_i)_{i\in I},(y_i)_{i\in I}\big) = \big(\upsilon(\Sigma_i)(x_i,y_i)\big)_{i\in I}$$

$$\upsilon(\mathcal{C}_A)(a,a) = a \qquad \upsilon\big(\coprod_{i\in I}\Sigma_i\big)\big(\iota_i(x),\iota_i(y)\big) = \upsilon(\Sigma_i)(x,y)$$

$$\upsilon(\Sigma^*)([],[]) = [] \qquad \upsilon(\Sigma^*)(x:\vec{r},y:\vec{s}) = \upsilon(\Sigma)(x,y) : \upsilon(\Sigma^*)(\vec{r},\vec{s})$$

Die Tupelbildung sei dabei strikt: $(a_i)_{i\in I} = \bot$, falls $a_i = \bot$ für ein $i \in I$. □

Programm 2.15 zeigt die Implementierung der generischen Bisimulationsoperation, die in der funktionalen Programmierung als generische oder polytypische *zip*-Operation bekannt ist.

```
class Signature σ ⇒ PolySignature σ where
  fzip :: σ α → σ β → σ (α, β)
instance PolySignature Identity where
  fzip (Identity x) (Identity y) = Identity (x, y)
instance Eq α → PolySignature (Const α) where
  fzip (Const a) (Const a') | a == a' = Const a
instance (PolySignature σ1, PolySignature σ2) ⇒ PolySignature (σ1 :*: σ2) where
  fzip (x :*: y) (x' :*: y') = fzip x x' :*: fzip y y'
instance (PolySignature σ1, PolySignature σ2) ⇒ PolySignature (σ1 :+: σ2) where
  fzip (Inject1 x) (Inject1 x') = Inject1 (fzip x x')
  fzip (Inject2 x) (Inject2 x') = Inject2 (fzip x x')
instance PolySignature σ ⇒ PolySignature (List σ) where
  fzip (List xs) (List ys) = List (zipWith fzip xs ys)
```

Programm 2.15: Bisimulationsoperation polynomieller Signaturen

Satz 2.98. *Die Abbildung $\upsilon(\Sigma)$ ist genau auf den Σ-kompatiblen Paaren definiert.*

$$\operatorname{dom} \upsilon(\Sigma) = (\asymp_\Sigma)$$

Beweis. Durch strukturelle Induktion in Σ. □

Satz 2.99. *Die folgenden Aussagen sind äquivalent:*

1. *Die $\upsilon(\Sigma)$-erreichbaren Nachbarn eines Paares (x, y) bilden den Graphen einer Abbildung f.*
2. *f ist Individualtransformation von x nach y.*

$$f : x \to y \iff \operatorname{graph} f = \big(\bar{\mathfrak{s}}(\Sigma) \circ \upsilon(\Sigma)\big)\big((x, y)\big)$$

Beweis. Durch strukturelle Induktion in Σ. □

In der Terminologie funktionaler Programmierung entspricht die Operation υ einer generischen *zip*-Funktion. Die Methode `fzipWithM_` der Typklasse `Signature` leistet auf den polynomiellen Signaturen genau die monadische Traversierung der $\upsilon(\Sigma)$-erreichbaren Nachbarn eines Paares.

Definition 2.100. Die Abbildung $\upsilon(\Sigma)$ läßt sich auf Σ-Coalgebren $\mathbb{C} = (C, \gamma)$ und $\mathbb{D} = (D, \delta)$ fortsetzen:

$$\upsilon(\mathbb{C}, \mathbb{D}) = \upsilon(\Sigma) \circ (\gamma \times \delta)$$

Der folgende Satz ist die Grundlage coinduktiver Beweise in polynomiellen Coalgebren, also ein strukturelles Coinduktionsprinzip.

Lemma 2.101. *Jede Coalgebra mit υ als Operation ist eine Bisimulation.*
Sei Σ ein polynomieller Signaturfunktor. Seien $\mathbb{C} = (C, \gamma)$ und $\mathbb{D} = (D, \delta)$ zwei Σ-Coalgebren. Sei $\mathbb{Y} = \big(Y, \upsilon(\mathbb{C}, \mathbb{D})\big)$ eine Σ-Coalgebra mit $Y \subseteq C \times D$. Dann gilt:

$$\pi_1 : \mathbb{Y} \to \mathbb{C} \qquad\qquad \pi_2 : \mathbb{Y} \to \mathbb{D}$$

Beweis. Durch strukturelle Induktion in Σ. □

2.4.3 Komposition und Dekomposition

Auch wenn Produkt und Coprodukt in der Regel gemischt verwendet werden, so läßt sich doch eine deutliche Dualität in der Affinität zu Algebra und Coalgebra feststellen.

1. Die Algebra bevorzugt als dekomponierende Strukturierung das Coprodukt: Sei $\Sigma = \coprod_{i \in I} \Sigma_i$. Dann zerfällt jeder Σ-Konstruktor $\alpha : \Sigma(A) \to A$ in eine Familie *disjunkter Fälle*:

$$\alpha = [\alpha_i]_{i \in I} \qquad \text{wobei} \quad \alpha_i : \Sigma_i(A) \to A$$

   Umgekehrt existiert zu jeder Familie $(\mathbb{A}_i)_{i \in I}$, wobei jeweils $\mathbb{A}_i = (A_i, \alpha_i)$ eine Σ_i-Algebra ist, eine freie Produktalgebra:

$$\prod_{i \in I} \mathbb{A}_i = (\prod_{i \in I} A_i, \prod_{i \in I} \alpha_i \circ \Sigma_i(\pi_i))$$

2. Die Coalgebra dagegen bevorzugt bei der Dekomposition das Produkt: Sei $\Sigma = \prod_{i \in I} \Sigma_i$. Dann zerfällt jeder Σ-Dekonstruktor $\gamma : C \to \Sigma(C)$ in eine Familie *unabhängiger Komponenten*:

$$\gamma = \langle \gamma_i \rangle_{i \in I} \qquad \text{wobei} \quad \gamma_i : C \to \Sigma_i(C)$$

   Umgekehrt existiert zu jeder Familie $(\mathbb{C}_i)_{i \in I}$, wobei jeweils $\mathbb{C}_i = (C_i, \gamma_i)$ eine Σ_i-Coalgebra ist, eine freie Coproduktcoalgebra:

$$\coprod_{i \in I} \mathbb{C}_i = (\prod_{i \in I} C_i, \coprod_{i \in I} \Sigma_i(\iota_i) \circ \gamma_i)$$

2.5 Algebraische und Coalgebraische Datentypen

2.5.1 Einsortige Daten

Hier zunächst die Reformulierung der bekannten Beispiele rund um den Signaturfunktor Σ_P.

Beispiel 2.102 (Peano-Signatur). Sei $\Sigma = \mathcal{C}_1 + \mathcal{I}$, wobei $1 = \{\star\}$.

1. Sei $\mathbb{A} = (A, \alpha : 1 + A \to A)$ eine Σ-Algebra. Dann zerfällt der Konstruktor α in eine Konstante $\alpha_1 : 1 \to A$ und eine einstellige Abbildung $\alpha_2 : A \to A$. Die Σ-Algebra der natürlichen Zahlen $\mathbb{N} = \big(\mathcal{N}, [0, \mathit{succ}]\big)$ ist initial.

2. Sei $\mathbb{C} = (C, \gamma : C \to C + 1)$ eine Σ-Coalgebra. Dann ist der Dekonstruktor γ gegeben durch eine einstellige totalisierte (partielle) Abbildung, wobei $\bot = \star$. Die Σ-Coalgebra der *erweiterten* natürlichen Zahlen $\bar{\mathbb{N}} = (\bar{\mathcal{N}}, \mathit{pred}?)$, wobei $\bar{\mathcal{N}} = \mathcal{N} \cup \{\omega\}$, $\mathit{pred}?(0) = \iota_1(\star)$, $\mathit{pred}?(n) = \iota_2\big(\mathit{pred}(n)\big)$ für $n > 0$ und $\mathit{pred}(\omega) = \iota_2(\omega)$, ist final.

3. Die finale Coalgebra ist in diesem Fall auch rational, zu Gegenbeispielen siehe Abschnitt 2.6.3. □

Beispiel 2.103 (Binärfolgen). Sei $\Sigma = \mathcal{C}_{\{0,1\}} \times \mathcal{I}$.

1. Die initiale Σ-Algebra ist leer.

2. Die Menge der unendlichen Binärfolgen ist eine finale Σ-Coalgebra; ebenso das bekanntermaßen isomorphe *reelle* Intervall $[0 \ldots 1]$. Daraus ergibt sich die folgende wichtige Erkenntnis über Kardinalität. □

2.5.2 Die Kardinalitätsexplosion

Der Inhalt des folgenden Satzes ist zwar keine Neuigkeit, aber eine wichtige Randbedingung, wenn Elemente von Coalgebren nicht wie üblich als Modelle von Prozessen, sondern von Daten dienen sollen.

Satz 2.104. *Sei Σ ein abzählbarer polynomieller Signaturfunktor.*

1. *Initiale Σ-Algebren sind stets abzählbar.*

2. *Finale Σ-Coalgebren sind im Allgemeinen überabzählbar.*

Beweis. Man betrachte einen alternativen Kleene-Abschluß, nämlich denjenigen der Termschachtelung: $\Sigma^\star(\emptyset) = \bigcup_{k \in \mathcal{N}} \Sigma^k(\emptyset)$ ist eine initiale Σ-Algebra. Mit Satz 2.87 läßt sich die Abzählbarkeit jeder Teilmenge $\Sigma^k(\emptyset)$ induktiv zeigen. Die Indexmenge $\mathcal{N}$ der Vereinigung ist ebenfalls abzählbar. Damit ist auch $\Sigma^\star(\emptyset)$ abzählbar.

Man betrachte andererseits die Signaturfunktoren $\mathsf{Seq}_k = \mathcal{C}_{D(k)} \times \mathcal{I}$ mit $D(k) = \{0, \ldots, k-1\}$. Die Menge $S(k)$ der unendlichen $D(k)$-Folgen (k-adischen Ziffernfolgen) ist eine finale Seq_k-Coalgebra. Es gilt $|S(k)| = k^{\aleph_0}$. Als wohlbekannter Speziallfall ist die Menge $S(1)$ der unendlichen unären Ziffernfolgen einelementig, die nächstgrößere Menge $S(2)$ der unendlichen binären Ziffernfolgen bereits überabzählbar. □

2.5.3 Mehrsortige Daten

Die bisherigen Beispiele für Signaturfunktoren spiegeln nicht die gesamte Bandbreite der klassischen, syntaxorientierten algebraischen Signaturen wieder. Da ein einstelliger Funktor nur von einem Trägerobjekt abstrahiert, stellt er auch nur eine unsortierte (genauer einsortige) Signatur dar. Mehrsortige (Co)Algebren können einerseits über Klassen mehrstelliger Funktoren mit entsprechender Anzahl von $\mathcal{I}$-Selektoren realisiert, oder andererseits in zwei Schritten approximiert werden:

1. Durch Vergessen der Sorten entsteht ein einstelliger Funktor, der auch Terme mit Typfehlern gestattet.

2. Eine Typisierung durch Auszeichnung von Teilmengen des unsortierten Trägers kompensiert diese Unterspezifikation für eine gegebene (Co)Algebra.

Da wir nur initiale Algebren und finale Coalgebren als Datenmodelle betrachten, und erstere in letztere eingebettet sind, soll nun die Typisierung finaler Coalgebren untersucht werden.

Sei Σ eine klassische Signatur mit der Sortenmenge S. Sei C_s die Menge der Konstruktorsymbole von Σ zur Sorte $s \in S$. Sei $a_s : C_s \to S^*$ die Abbildung von Konstruktoren auf die Liste der Sorten ihrer Argumente. Durch Abstraktion von den Argumentsorten entsteht folgender Signaturfunktor:

$$\Sigma' = \coprod_{s \in S} \coprod_{c \in C_s} \mathcal{I}^{|a'_s(c)|}$$

Die Ergebnissorte bleibt als Injektion der äußeren Ebene erhalten. Sei nun $\mathbb{F} = (F, \phi)$ eine finale Σ'-Coalgebra. Die Typisierung von F hat dann die Form eines rekursiven Gleichungssystems, wobei jeder Sorte $s \in S$ eine Gleichung entspricht:

$$\mathcal{P}(\phi)(T_s) = \bigcup_{c \in C_s} \mathcal{P}(\iota_s \circ \iota_c)\Big(\prod_{i \in \operatorname{dom} a_s(c)} T_{a_s(c)(i)} \Big)$$

Dabei beschreibt $T_s \subseteq F$ die Elemente vom Typ s. Die Gleichung liest sich folgendermaßen: Ein Element sei vom Typ s, genau dann wenn sein Datensatz als Wurzel einen Konstruktor c mit Resultattyp s und als Argumente Elemente von den jeweiligen Argumenttypen des Konstruktors hat. Eine Typisierung ist dann eine Überdeckung von F mit einer Lösung $(T_s)_{s \in S}$.

Jede Lösung des Gleichungssystems entspricht einer Fixpunktsemantik der darin enthaltenen Rekursion, von denen potentiell viele existieren können. Da in Kapitel 5 ein isomorphes Problem in aller Ausführlichkeit behandelt werden soll, sei hier nur informell festgestellt, daß die (sortenweise) kleinste Lösung initiale Datentypen liefert, die größte Lösung dagegen finale Datentypen. Des weiteren ist die kleinste Lösung als einzige im algebraischen Anteil F_0 enthalten. Sie kann also auch als Schnitt der größten Lösung mit F_0 definiert werden. Analog erhalten wir die größte rationale Lösung als Schnitt mit dem rationalen Anteil F_*.

Beispiel 2.105. Sei Σ mit $S = \{nat, list\}$ die gemeinsame Signatur der natürlichen Zahlen nat und der Listen $list$ von solchen Zahlen, aufgespannt durch die Operationen:

$$\begin{aligned} \mathsf{zero} &: [] \to nat & \qquad \mathsf{nil} &: [] \to list \\ \mathsf{succ} &: [nat] \to nat & \qquad \mathsf{cons} &: [nat, list] \to list \end{aligned}$$

Dann ergibt sich also:

$$\begin{aligned} C_{nat} &= \{\mathsf{zero}, \mathsf{succ}\} & \qquad C_{list} &= \{\mathsf{nil}, \mathsf{cons}\} \\ a_{nat}(\mathsf{zero}) &= [] & \qquad a_{list}(\mathsf{nil}) &= [] \\ a_{nat}(\mathsf{succ}) &= [nat] & \qquad a_{list}(\mathsf{cons}) &= [nat, list] \end{aligned}$$

Die Datensätze $\Sigma'(C)$ zu einer Σ'-Coalgebra mit Träger C sind partitioniert in die Bilder der vier Konstruktoren:

$$\begin{aligned} \iota_{nat} \circ \iota_{\mathsf{zero}} &: 1 \to C & \qquad \iota_{list} \circ \iota_{\mathsf{nil}} &: 1 \to C \\ \iota_{nat} \circ \iota_{\mathsf{succ}} &: C \to C & \qquad \iota_{list} \circ \iota_{\mathsf{nil}} &: C \times C \to C \end{aligned}$$

Nun zur Lösung der Typisierungsgleichungen. Da die beiden Typen nicht gegenseitig rekursiv voneinander abhängen, können die beiden Gleichungen einzeln gelöst werden:

1. Der Typ der natürlichen Zahlen hat in F genau zwei Modelle.

$$\mathcal{P}(\phi)(T_{nat}) = \left\{(\iota_{nat} \circ \iota_{\mathsf{zero}})(\star)\right\} \cup \left\{(\iota_{nat} \circ \iota_{\mathsf{succ}})(n) \;\middle|\; n \in T_{nat}\right\}$$

   Das kleinere Modell ist isomorph zu $\mathbb{N}$, das größere, rationale ist isomorph zu $\bar{\mathbb{N}}$.

2. Der Typ der Listen hat dagegen viele Modelle.

$$\mathcal{P}(\phi)(T_{list}) = \left\{(\iota_{list} \circ \iota_{\mathsf{nil}})(\star)\right\} \cup \left\{(\iota_{list} \circ \iota_{\mathsf{cons}})(n, l) \;\middle|\; n \in T_{nat}, l \in T_{list}\right\}$$

   Dieser Gleichung ist nicht anzusehen, ob unendliche Listen dem Typ angehören sollen: Jede suffixabgeschlossene Teilmenge unendlicher Listen erzeugt eine gültige Lösung. Daher existieren mindestens sechs ausgezeichnete Modelle, nämlich kleinstes, größtes rationales und größtes Modell, multipliziert mit den beiden Kandidaten für T_{nat}. Das kleinste Modell enthält nur die endlichen Listen, das größte rationale auch die periodischen, und das größte auch die divergenten Listen. □

In späteren Abschnitten werden wir uns die Freiheit gestatten, die formale Unterscheidung von Konstruktorsymbol c und induzierter Operation ι_{c} in der Notation mit Rücksicht auf die Lesbarkeit zu vernachlässigen.

2.6 Terme und Coterme

In Umkehrung der Übersetzung zwischen syntaktischen und mengentheoretischen Signaturen besitzt jeder streng polynomielle Signaturfunktor Σ eine bis auf punktweise Isomorphie eindeutige Normalform der Gestalt:

$$\Sigma = \coprod_{i \in I} \prod_{j \in J_i} \Sigma_{ij} \qquad \text{wobei} \quad \Sigma_{ij} = \mathcal{I} \text{ oder } \Sigma_{ij} = \mathcal{C}_{A_{ij}}$$

Aus dieser Form läßt sich direkt die *Termalgebra* ablesen, welche eine initiale Σ-Algebra ist. Die Initialität in der Algebra kann also als syntaktisches Phänomen aufgefaßt werden. Der finalen Coalgebra entspricht dagagen keine unmittelbare Syntax, was das zentrale Problem bei der Entwicklung einer kanonischen coalgebraischen Notation ist. Zwar existiert für die hier betrachteten polynomiellen Signaturfunktoren eine generische Limes-Konstruktion für finale Coalgebren [59], doch auf Grund der erwähnten Kardinalitätsexplosion zeigt dieser Limes eher semantische als syntaktische Qualitäten. In der Tat wird Finalität häufig als semantisches Phänomen gehandhabt. Eine Coterm-Notation muß also auf einer nicht-finalen Coalgebra aufbauen, und leidet daher *a priori* an einer gewissen Unterbestimmtheit. Diese soll im Folgenden systematisch untersucht und beschränkt werden.

Satz 2.106 (Universelle Finale Coalgebra)**.** *Sei Σ ein polynomieller Signaturfunktor. Sei $1 = \{\star\}$ und $!(x) = \star$. Der Limes der unendlichen Folge*

$$1 \xleftarrow{\;!\;} \Sigma(1) \xleftarrow{\;\Sigma(!)\;} \Sigma^2(1) \xleftarrow{\;\Sigma^2(!)\;} \cdots$$

existiert und ist eine finale Σ-Coalgebra. [55]

Beispiel 2.107. Ein Repräsentant dieser Limeskonstruktion ist die Menge der unendlichen Folgen

$$F = \left\{ (x_0, x_1, x_2, \dots) \;\middle|\; \forall n \geq 0.\, x_n \in \Sigma^n(1) \wedge \Sigma^n(!)(x_{n+1}) = x_n \right\}$$

Diese Definition ist zunächst ebenso wenig intuitiv verständlich wie das obige Diagramm. Es fällt jedoch auf, daß die Elemente von F nicht im Allgemeinen endlich darstellbar sind. Diese Beobachtung bestärkt die Vermutung, daß die finale Coalgebra keinen geeigneten Ansatz als coalgebraische Syntax darstellt.

Eine solche Folge $(x_i) \in F$ beschreibt einen Traversierungsprozess des Raumes, welcher durch rekursive Anwendung des Dekonstruktors auf ein Element aufgespannt wird. Das Glied x_n steht dabei für den Horizont der Traversierung bei einer Tiefenbeschränkung von n. Eine finale Coalgebra stellt somit eine *black-box*-Sicht auf ein System dar, die Zustände ausschließlich über die möglichen folgenden Beobachtungen identifiziert. Bemerkenswert ist dabei, daß die Ausdrücke nicht an der Wurzel, sondern an den Blättern wachsen,[1] genauer an den Positionen, welche Vorkommen des Funktors $\mathcal{I}$ entsprechen. Auf dieses Evolutionsverhalten wird in Abschnitt 4.1.4 im Rahmen der Auswertungsstrategie für corekursive Funktionen noch detailliert eingegangen. □

Beispiel 2.108 (Rekonstruktion der erweiterten natürlichen Zahlen). Gegeben sei der aus Beispiel 2.102 bekannte Signaturfunktor $\Sigma = \mathcal{C}_1 + \mathcal{I}$. Für die Injektionen schreiben wir $\mathsf{z} : 1 \to \Sigma(X)$ und $\mathsf{s} : X \to \Sigma(X)$. Als Träger F der finalen Σ-Coalgebra ergibt sich nach obiger Konstruktion die Menge der unendlichen Folgen (x_i), für die gilt:

$$x_0 = \star$$
$$x_{i+1} \begin{cases} = (\mathsf{s}^j \circ \mathsf{z})(\star) & \text{falls } x_i = (\mathsf{s}^j \circ \mathsf{z})(\star) \\ \in \left\{ (\mathsf{s}^i \circ \mathsf{z})(\star), (\mathsf{s}^i \circ \mathsf{s})(\star) \right\} & \text{falls } x_i = \mathsf{s}^i(\star) \end{cases} \qquad (j < i \geq 1)$$

Die Menge F enthält für jede natürliche Zahl n genau eine Folge f_n, die gegen $(\mathsf{s}^n \circ \mathsf{z})(\star)$ konvergiert, sowie eine zusätzliche, divergente Folge f_ω. □

[1] Um die botanische Metapher auf die Spitze zu treiben, könnte man hier von *Farnen* anstelle von *Bäumen* sprechen.

2.6.1 Allgemeine Coterme

Definition 2.109 (Coterme und finale Semantik)**.** Sei Σ ein polynomieller Signaturfunktor. Sei C eine Trägermenge. Eine Notation für C induziert eine Notation für die Datensätze $\Sigma(C)$. Daraus ergibt sich wiederum (unter Zuhilfenahme eines Mengen- oder Funktionenkalküls) eine Notation für Abbildungen $\gamma : C \to \Sigma(C)$, und damit für Σ-Coalgebren $\mathbb{C} = (C, \gamma)$. Ein Paar $t = \mathbb{C} \bullet r$ aus einer Σ-Coalgebra $\mathbb{C} = (C, \gamma)$ und einem Element $r \in C$ heißt Σ-*Coterm* und denotiert ein eindeutiges Element der finalen Σ-Coalgebra $\mathbb{F}$, die *finale Semantik* von r:

$$\mathbb{C} \bullet r \longmapsto [\![\mathbb{C}]\!](r)$$

Seien t_1, t_2 zwei Σ-Coterme. Wir nennen t_1 und t_2 *äquivalent* und schreiben $t_1 \simeq t_2$, falls beide dasselbe Element $f \in F$ bezeichnen. □

Beispiel 2.110. Gegeben sei der aus Beispiel 2.102 bekannte Signaturfunktor $\Sigma = \mathcal{C}_1 + \mathcal{I}$. Für die Injektionen schreiben wir $\mathsf{z} : 1 \to \Sigma(X)$ und $\mathsf{s} : X \to \Sigma(X)$. Sei $\bar{\mathbb{N}}$ die bekannte finale Σ-Coalgebra. Seien a, b, c, d beliebige Elemente, die im Folgenden zu endlichen Trägermengen kombiniert werden. Einige Σ-Coterme $t_i = (C_i, \gamma_i) \bullet r_i$ und ihre finale Semantik:

i	C	γ	r	$[\![(C,\gamma)]\!](r)$
1	$\{a\}$	$\{a \mapsto \mathsf{z}(\star)\}$	a	0
2	$\{a, b\}$	$\{a \mapsto \mathsf{z}(\star), b \mapsto \mathsf{s}(a)\}$	a	0
3	$\{a, b\}$	$\{a \mapsto \mathsf{z}(\star), b \mapsto \mathsf{s}(a)\}$	b	1
4	$\{c\}$	$\{c \mapsto \mathsf{s}(c)\}$	c	ω
5	$\{c, d\}$	$\{c \mapsto \mathsf{s}(d), d \mapsto \mathsf{s}(c)\}$	c	ω
6	$\mathcal{N}^*$	*tail*	$[17, 4, 42]$	3

wobei $\mathit{tail}([]) = \mathsf{z}(\star)$ und $\mathit{tail}(\_ : t) = \mathsf{s}(t)$. □

Definition 2.111 (Coterm-Morphismus)**.** Sei Σ ein polynomieller Signaturfunktor. Seien $t_1 = \mathbb{C}_1 \bullet r_1$ und $t_2 = \mathbb{C}_2 \bullet r_2$ zwei Σ-Coterme. Dann heißt ein Σ-Coalgebra-Morphismus $f : \mathbb{C}_1 \to \mathbb{C}_2$ ein Σ-*Coterm-Morphismus* $f : t_1 \to t_2$, falls $f(r_1) = r_2$. □

Satz 2.112. *Coterm-Morphismen erhalten finale Semantik.*

Seien $\mathbb{C}_1 = (C_1, \gamma_1)$ und $\mathbb{C}_2 = (C_2, \gamma_2)$ zwei beliebige Σ-Coalgebren, $\mathbb{F} = (F, \phi)$ eine finale Σ-Coalgebra und $f : \mathbb{C}_1 \to \mathbb{C}_2$ ein Σ-Coalgebra-Morphismus. Dann gilt für alle $r \in C_1$:

$$[\![\mathbb{C}_1]\!](r) = [\![\mathbb{C}_2]\!]\big(f(r)\big)$$

Beweis. Einfache Diagrammjagd:

$$\begin{aligned}
& & \phi \circ [\![\mathbb{C}_2]\!] \circ f \\
([\![\mathbb{C}_2]\!] : \mathbb{C}_2 \to \mathbb{F}) & & = \Sigma\big([\![\mathbb{C}_2]\!]\big) \circ \gamma_2 \circ f \\
(f : \mathbb{C}_1 \to \mathbb{C}_2) & & = \Sigma\big([\![\mathbb{C}_2]\!]\big) \circ \Sigma(f) \circ \gamma_1 \\
(\text{Funktor}) & & = \Sigma\big([\![\mathbb{C}_2]\!] \circ f\big) \circ \gamma_1
\end{aligned}$$

Damit gilt insgesamt $([\![\mathbb{C}_2]\!] \circ f) : \mathbb{C}_1 \to \mathbb{F}$ und die Behauptung folgt durch Eindeutigkeit des Anamorphismus $[\![\mathbb{C}_1]\!]$. □

Die folgenden Definitionen dienen dem Ausschluß von redundanten (ununterscheidbaren) und irrelevanten Elementen.

2.6.2 Kanonische Coterme

Definition 2.113 (Einfache und minimale Coterme)**.** Ein Coterm $\mathbb{C} \bullet r$ heißt *einfach*, wenn $\mathbb{C}$ einfach ist. Elemente von $\mathbb{C}$, die in einer einfachen eigentlichen Quotientencoalgebra identifiziert werden, heißen *ununterscheidbar*.

Ein Coterm $\mathbb{C} \bullet r$ heißt *minimal*, wenn $\mathbb{C}$ keine echten eigentlichen Untercoalgebren besitzt, die r enthalten. Elemente, die in einer einfachen eigentlichen Untercoalgebra, welche r enthält, nicht enthalten sind, heißen *irrelevant*. □

Definition 2.114 (Kanonische Coterme)**.** Ein Coterm heißt *kanonisch*, wenn er einfach und minimal ist. □

Beispiel 2.115. Gegeben seien die in Beispiel 2.110 definierten Coterme. Von diesen ist:

- t_2 nicht minimal: b ist irrelevant.
- t_5 nicht einfach: c, d sind ununterscheidbar.
- t_6 weder minimal noch einfach: alle gleichlangen Listen sind ununterscheidbar; alle Listen ausser $[17, 4, 42]$, $[4, 42]$, $[42]$ und $[]$ sind irrelevant. □

Minimalität impliziert ein strukturelles Induktionsprinzip:

Lemma 2.116 (Vollständigkeit)**.** *Sei $t = (C, \gamma) \bullet r$ ein minimaler Σ-Coterm. Dann ist ganz C von r erreichbar.*

Beweis. Angenommen, es existierte ein $x \in C$, so daß $r \nrightarrow^* x$. Dann wäre nach Lemma 2.48 $(C \setminus \mathfrak{p}^*(x), \gamma)$ eine echte Untercoalgebra von (C, γ), die r enthält. □

Induktives Beweisen eines Prädikates p über einem minimalen Coterm $t = (C, \gamma) \bullet r$ funktioniert also folgendermaßen:

1. Man zeige $r \to^* x \implies p(x)$ für alle x durch Induktion, beispielsweise:
   (a) Induktionsanfang: $p(r)$.
   (b) Induktionsschritt: Angenommen $r \to^* x \implies p(x)$. Dann zeige man $x \to x' \implies p(x')$.
2. Dann gilt $p(x)$ für alle $x \in C$.

Mit den bisher eingeführten Begriffen läßt sich die Unterbestimmtheit von Cotermen auf ein Maß reduzieren, das mit der α-Äquivalenz im λ-Kalkül vergleichbar ist:

Satz 2.117. *Kanonische Coterme sind eindeutig bis auf Isomorphie.*

Beweis. Zu zeigen sind zwei Implikationen:

1. Sei t_1 ein kanonischer Coterm, t_2 ein beliebiger Coterm und $i : t_1 \to t_2$ ein Isomorphismus. Da Isomorphismen Einfachheit und Minimalität erhalten, ist t_2 ebenfalls kanonisch.
2. Seien t_1, t_2 zwei äquivalente kanonische Coterme. Dann existiert ein Isomorphismus $i : t_1 \to t_2$, falls zu jedem Element $x_1 \in C_1$ genau ein Element $x_2 \in C_2$ existiert, so daß $[\![\mathbb{C}_1]\!](x_1) = [\![\mathbb{C}_2]\!](x_2)$.
   (a) Es existiert zu jedem x_1 höchstens ein passendes x_2; ansonsten wäre $[\![\mathbb{C}_2]\!]$ nicht mono, und t_2 nicht einfach.
   (b) Es existiert zu jedem x_1 wenigstens ein passendes x_2: Zu jedem x_1 mit $r_1 \to^* x_1$ existiert ein x_2; ansonsten ist $\mathbb{C}_2$ keine Σ-Coalgebra (Beweis durch Induktion). Da t_1 minimal ist, ist das nach Lemma 2.116 bereits ganz C_1. □

Die Notation von Coalgebra-Elementen kann als Speziallfall einer coinduktiven Funktionsdefinition betrachtet werden. Auf diese wird in Abschnitt 2.8 näher eingegangen.

2.6.3 Coterme und Kardinalität

Da alle äquivalenten kanonischen Coterme auf gleichmächtigen Coalgebren aufbauen, ist die Kardinalität der Träger ein natürliches Maß für die Komplexität des denotierten Elementes der finalen Coalgebra.

Definition 2.118. Sei Σ ein Signaturfunktor und $\mathbb{C} = (C, \gamma)$ eine Σ-Coalgebra. Ein Coterm $\mathbb{C} \bullet r$ heißt *endlich*, wenn C endlich ist. Die finale Semantik $[\![\mathbb{C}]\!](r)$ heißt *endlich darstellbar*. □

Satz 2.119. *Sei Σ ein Signaturfunktor und $\mathbb{F} = (F, \phi)$ eine finale Σ-Coalgebra. Die Menge F_* der endlich darstellbaren Elemente von F definiert die größte rationale Untercoalgebra, kurz als* die *rationale Σ-Coalgebra $\mathbb{F}_*$ bezeichnet.*

Beweis. Für jedes Element $x \in F$ mit endlicher Umgebung $\mathfrak{s}^*_{\mathbb{F}}(x)$ läßt sich durch Abzählen derselben eine endliche Darstellung konstruieren. Also kann ein nicht endlich darstellbares Element nicht in einer rationalen Untercoalgebra von $\mathbb{F}$ enthalten sein. □

Korollar 2.120. *Die rationale Coalgebra steht in ihrer Mächtigkeit zwischen initialer Algebra und finaler Coalgebra:*

$$\mathbb{F}_0 \subsetneqq \mathbb{F}_* \subsetneqq \mathbb{F}$$

Die rationale Coalgebra ist die zentrale semantische Struktur für den gesamten Rest dieser Arbeit:

1. Sie umfaßt genau die Elemente, die von Maschinen mit beschränktem Speicherplatz in extensionaler Codierung, also ohne unausgewertete Anteile, dargestellt und verarbeitet werden können.
2. Erschöpfende Traversierungsverfahren haben nur auf rationalen Elementen einen endlichen Suchraum und können terminieren.

Beispiel 2.121 (Dezimahlzahlen). Sei $D = \{0, 1, \ldots, 9\}$ und $\Sigma = \mathcal{C}_D \times \mathcal{I}$. Die Menge der unendlichen Dezimalbrüche ist eine finale Σ-Coalgebra, und damit auch das *reelle* Intervall $[0 \ldots 1]$. Die endlichen Σ-Coterme denotieren genau die *rationalen* Zahlen in diesem Intervall.[2] □

Beispiel 2.122 (Automaten). Seien A, B endliche Mengen. Sei $\Sigma = \prod_A (\mathcal{I} \times \mathcal{C}_B)$. Ein Coterm $t = \big(C, \gamma : C \to \Sigma(C)\big) \bullet r$ denotiert einen Mealy-Automaten $(C, A, B, \tau, \omega, r)$. Man betrachte die folgende natürliche Isomorphie:

$$\begin{aligned} C \to \Sigma(C) = C \to \prod_A (C \times B) \\ \simeq C \to (A \to C \times B) \\ \simeq C \times A \to C \times B \end{aligned}$$

Sei $t : \big(C \to \Sigma(C)\big) \to (C \times A \to C \times B)$ die entsprechende natürliche Transformation. Dann ist $t(\gamma) = \langle \tau, \omega \rangle$.

Die endlichen Σ-Coterme denotieren genau die endlichen Mealy-Automaten. □

[2] Zu tieferen zahlentheoretischen Anwendungen von Coalgebra siehe [45].

2.6.4 Coterme und Pattern Matching

Die deklarative Dekomposition von Daten durch syntaktische Muster (*pattern matching*) ist eine spezifisch coalgebraische Technik. In klassischen algebraischen Kontexten gelingt sie deshalb, weil nur die initiale Algebra betrachtet wird, die zu einer Coalgebra invertierbar ist.

Definition 2.123 (Formale Pattern)**.** Sei V eine Menge von Variablen. Sei Σ eine polynomielle Signatur. Wir definieren die Σ-Pattern $P(\Sigma)$ zusammen mit ihrer Bedeutungsfunktion $[\![\_]\!]_\Sigma$. Für jedes Σ-Pattern $p \in P(\Sigma)$ ist $[\![p]\!]_\Sigma$ eine polymorphe partielle Abbildung, welche für alle Mengen C die Σ-Datensätze über C auf Variablenbelegungen über C abbildet:

$$[\![p]\!]_\Sigma : \Sigma(C) \nrightarrow (V \nrightarrow C)$$

Die Menge $P(\Sigma)$ ist zusammen mit ihrer Bedeutung induktiv definiert als die kleinste Menge, welche folgenden Regeln genügt:

1. Eine Variable $v \in V$ ist ein $\mathcal{I}$-Pattern.

$$[\![v]\!]_{\mathcal{I}}(x) = \{v \mapsto x\}$$

2. Eine Konstante $a \in A$ ist ein $\mathcal{C}_A$-Pattern.

$$[\![a]\!]_{\mathcal{C}_A}(a') = \begin{cases} \emptyset & a = a' \\ \bot & a \neq a' \end{cases}$$

3. Sei $\Sigma = \prod \Sigma_i$. Ein Tupel von Σ_i-Pattern p_i ist ein Σ-Pattern.

$$[\![(p_i)_{i\in I}]\!]_\Sigma\big((x_i)_{i\in I}\big) = \bigsqcup_{i\in I} [\![p_i]\!]_{\Sigma_i}(x_i)$$

4. Sei $\Sigma = \coprod \Sigma_i$. Die Injektion eines Σ_i-Patterns p_i ist ein Σ-Pattern.

$$[\![\iota_i(p)]\!]_\Sigma\big(\iota_{i'}(x)\big) = \begin{cases} [\![p]\!]_{\Sigma_i}(x) & i = i' \\ \bot & i \neq i' \end{cases}$$

5. Eine Liste von Σ-Pattern ist ein Σ^*-Pattern.

$$[\![[p_1, \ldots, p_n]]\!]_{\Sigma^*}\big([x_1, \ldots, x_{n'}]\big) = \begin{cases} \bigsqcup_{i=1}^{n} [\![p_i]\!]_\Sigma(x_i) & n = n' \\ \bot & n \neq n' \end{cases}$$

Dabei sei $\sqcup$ die vereinigte Abbildung und zusätzlich strikt, also $\bot \sqcup f = f \sqcup \bot = \bot$. Ein Pattern heißt *linear*, wenn keine Variable mehr als einmal vorkommt. □

Definition 2.124 (Ungetyptes Matching)**.** Ein Σ-Pattern p paßt *ungetypt* zu einem Σ-Datensatz x, geschrieben $p := x$, genau dann wenn die Bedeutung definiert ist:

$$p := x \iff [\![p]\!]_\Sigma(x) \neq \bot$$ □

Definition 2.125 (Getyptes Matching)**.** Sei C eine Menge und $t : V \to \mathcal{P}(C)$ eine Abbildung, welche jeder Variablen $v \in V$ einen Wertebereich $t(v) \subseteq C$ zuordnet. Ein Σ-Pattern p paßt *t-getypt* zu einem Σ-Datensatz x über C, geschrieben $p :=_t x$, genau dann wenn die Bedeutung definiert und verträglich mit der Typisierung t ist:

$$p :=_t x \iff [\![p]\!]_\Sigma(x) \neq \bot \wedge \big(\forall\, v \in \mathrm{dom}\,[\![p]\!]_\Sigma(x).\ [\![p]\!]_\Sigma(x)(v) \in t(v)\big)$$ □

Lemma 2.126. *Ein Pattern paßt ungetypt zu einem Datensatz über einer Menge C, genau dann wenn es $const_C$-getypt paßt.*

Beispiel 2.127. Sei $\Sigma = (\mathcal{C}_1 + \mathcal{I}) + (\mathcal{C}_1 + \mathcal{I} \times \mathcal{I})$ die aus dem vorigen Abschnitt bekannte kombinierte Signatur der natürlichen Zahlen und Listen. Diese besitzt bis auf Variablenumbenennung vier Pattern:

$$p_1 = \iota_{nat}\big(\iota_{\mathsf{zero}}(\star)\big) \qquad p_3 = \iota_{list}\big(\iota_{\mathsf{nil}}(\star)\big)$$
$$p_2 = \iota_{nat}\big(\iota_{\mathsf{succ}}(v)\big) \qquad p_4 = \iota_{list}\big(\iota_{\mathsf{cons}}(v_1, v_2)\big)$$

Zur besseren Lesbarkeit können Injektionen komponiert und Applikationen von $(\star)$ weggelassen werden. □

Definition 2.128 (Geschachtelte Pattern)**.** Diese Bedeutung von Pattern läßt sich auf eine Σ-Coalgebra $\mathbb{C} = (C, \gamma)$ durch Vorschalten der Operation γ fortsetzen. Damit lassen sich Pattern auch beliebig tief schachteln. Dazu wird für eine Σ-Coalgebra $\mathbb{C} = (C, \gamma)$ folgende Pattern-Regel adjungiert:

6. Ein Σ-Pattern ist ein $\mathcal{I}$-Pattern.
$$[\![p]\!]^{\mathbb{C}}_{\mathcal{I}}(x) = [\![p]\!]^{\mathbb{C}}_{\Sigma}(\gamma(x))$$ □

Dasselbe gilt für eine initiale Algebra $\mathbb{A} = (A, \alpha)$ durch Vorschalten der inversen Operation α^{-1}. Für nicht-initiale Algebren existiert keine allgemeine Fortsetzung.

Man beachte, daß Pattern induktiv definiert sind und Variablen nur als Blätter enthalten. Damit ist es nicht möglich, einen Zyklus zu spezifizieren. Die Zyklenerkennung soll einem anderen Mechanismus vorbehalten bleiben, der in Teil II ausführlich beschrieben wird.

Beispiel 2.129. Sei $\Sigma = \mathcal{C}_1 + \mathcal{I}$ die Signatur der natürlichen Zahlen und $\bar{\mathbb{N}} = (\bar{\mathcal{N}}, \mathit{pred}?)$ die bekannte finale Σ-Coalgebra. Wir schreiben direkt $\mathsf{zero}, \mathsf{succ}$ für die Injektionen. Einige Pattern und ihre Bedeutung:

$$\mathsf{zero} := 0 \qquad [\![\mathsf{zero}]\!]^{\bar{\mathbb{N}}}_{\Sigma}(0) = \emptyset$$
$$\mathsf{succ}(\mathsf{zero}) := 1 \qquad [\![\mathsf{succ}(\mathsf{zero})]\!]^{\bar{\mathbb{N}}}_{\Sigma}(1) = \emptyset$$
$$n \geq 2 \implies \mathsf{succ}\big(\mathsf{succ}(v)\big) := n \qquad [\![\mathsf{succ}\big(\mathsf{succ}(v)\big)]\!]^{\bar{\mathbb{N}}}_{\Sigma}(n) = \{v \mapsto n-2\}$$

Die folgende Eigenschaft ist ein wesentliches Argument dafür, daß die finale Coalgebra die für deklarative Sprachen adäquate Sichtweise auf Daten darstellt.

Lemma 2.130. *Sei Σ eine polynomielle Signatur. Zwei Elemente einer Σ-Coalgebra sind genau dann durch das Matching geschachtelter linearer Pattern nicht unterscheidbar, wenn sie bisimilar sind.*

$$x \sim y \iff \forall p.\ p := x \Leftrightarrow p := y$$

Beweis. Durch strukturelle Induktion in Σ. □

Bisimilare Elemente sind aber gegebenenfalls noch anhand der von einem passenden Pattern bestimmten Variablenbelegung unterscheidbar.

2.7 Nichtfundierte Algebra und Coalgebra

In dieser Arbeit eine Einführung in die nichtfundierte Algebra als Metatheorie nichtstrikter Semantik zu geben, würde zu weit führen. In diesem Zusammenhang sei auf Standardliteratur, insbesondere die Originalarbeit von Aczel[2] verwiesen.

Der wesentliche Unterschied zwischen dem hier verfolgten strikten Ansatz und seinen deutlich populäreren, nichtstrikten Gegenparts kann an einem Vergleich mit der bahnbrechenden anwendungsbezogenen Arbeit von Meijer[37] deutlich gemacht werden. Dort werden Algebra und Coalgebra in der Kategorie **CPO** der ω-CPOs und stetigen Abbildungen angesiedelt. Diese ist nicht nur ein adäquates denotationelles Modell nichtstrikter Auswertung, sondern sie hat zusätzlich die reizvolle Eigenschaft, daß initiale Algebren und finale Coalgebren dieselben Trägerobjekte besitzen.

Die dort aufgestellte Behauptung, in der Kategorie **Set** seien initiale Algebra und finale Coalgebra zwei verschiedene Welten, ist allerdings nicht uneingeschränkt zutreffend. Wie wir gezeigt haben, existiert sehr wohl eine kanonische Einbettung der einen in die andere, so daß zwar nicht von einer Gleichheit, aber von einer Teil-Ganzes-Beziehung der beiden die Rede sein kann. In dieser Beziehung steht zwischen der initialen Algebra und der finalen Coalgebra noch die rationale Coalgebra. Diese dreistufige Hierarchie hat ähnlich wie die Chomsky-Hierarchie die Eigenschaft, daß größere Klassen jeweils ausdrucksmächtiger sind, während für kleinere die ökonomischeren Modelle und Verfahren existieren.

In dieser Hierarchie steht auf der unteren Stufe die klassische, fundierte algebraische Welt mit strikter Auswertung, in der Daten extensional, also streng von Code trennbar sind. Oben steht dagegen die Welt des allgemein Unendlichen, die nur *lazy* approximativ erschlossen werden kann, indem Daten intensional mit eingebetteten Berechnungen dargestellt und nichtstrikt so weit wie nötig verarbeitet werden. Diese Arbeit widmet sich ausgiebig der mittleren Stufe, wobei der Anspruch gilt, mit möglichst wenig Mehraufwand im Vergleich zur fundierten Algebra auszukommen. Der Ansatz steht also unter der Prämisse, daß das Aufgeben der Fundierung für die Erschließung des rational Unendlichen unverhältnismäßig ist.

Allerdings sind nicht alle Argumente für einen neuen Ansatz rein ökonomischer Natur: Die nichtfundierte Algebra vereint gewissermaßen die gewohnte und leichte Handhabung der Algebra mit der Mächtigkeit der Coalgebra. Der offensichtliche Preis dafür sind die bekannten Komplikationen semantischer und implementierungstechnischer Natur, mit denen sich nichtstrikte Programmiersprachen herumschlagen. Ein weiterer, kaum beachteter Nachteil ist aber die Ununterscheidbarkeit von rationalen und unendlichen Anteilen finaler Datentypen. In Teil II dieser Arbeit werden coalgebraische Verfahren vorgestellt, die nur auf dem rationalen Anteil effektiv sind, dort aber Probleme lösen, die in nichtfundierter Algebra nicht direkt behandelt werden können.

2.8 Rekursive und Corekursive Funktionen

2.8.1 Syntaktische und semantische Rekursion

Die klassische und namensgebende Bedeutung des Begriffes der Rekursion ist, daß ein Gegenstand in seiner eigenen Definition vorkommt. Dieses Phänomen bezeichnen wir exakter als *syntaktische* Rekursion. Offensichtlich ist es nicht trivial, einer solchen rekursiven Definition eine Bedeutung zuzuordnen, so daß *semantische* Rekursion eine völlig andere Sache ist:

1. Während es bei nichtrekursiven Definitionen möglich ist, die Bedeutung des Definiendums lokal aus den Bedeutungen der Bestandteile des Definiens zu synthetisieren, muß im rekursiven Fall auf globale Techniken, wie Grenzwert- oder Fixpunktkonstruktionen zurückgegriffen werden.

2. Auch wenn eine semantische Metatheorie zur Deutung von Rekursion zur Verfügung steht, kann eine konkrete rekursive Definition bedeutungslos sein. Das berühmteste Beispiel ist wohl das *Paradoxon des Barbiers von Sevilla* von RUSSELL. Rekursive Funktionsdefinitionen, deren Auswertung in einen unendlichen Regress führt, gehören ebenfalls in diese Kategorie, und sind jedem Informatiker wohl leidvoll bekannt.

3. Zu allem Überfluß kann eine rekursive Definition auch mehr als eine Bedeutung besitzen:

   (a) Wie bereits angedeutet, können einem rekursiven Prädikat unter gewissen Bedingungen mehrere konkurrierende Modelle zugeordnet sein. Dem werden wir uns in Kapitel 5 in aller Ausführlichkeit zuwenden.

   (b) In einer fundierten mengentheoretischen Interpretation haben auch rekursive Funktionsdefinitionen zwei unterschiedliche plausible Lesarten, die sich in der Bedeutung von Argument- und Resultatstyp unterscheiden. In Anlehnung an [17] bezeichnen wir eine syntaktisch rekursive Funktionsdefinition als *semantisch rekursiv*, wenn sie auf der initialen (algebraischen) Struktur ihres Argumenttyps aufbaut, und als *semantisch corekursiv*, wenn sie auf der finalen (coalgebraischen) Struktur ihres Resultattyps aufbaut.

Auf der semantischen Ebene von Funktionsdefinitionen tritt also eine Dualisierung ein. Wo von Corekursion die Rede ist, sind üblicherweise diese dualen Begriffe gemeint. Die Verwendung des Begriffes Rekursion in der Literatur läßt sich dagegen oft nicht eindeutig der singulären syntaktischen oder der dualen semantischen Ebene zuordnen.

Anders als die etwa gleichaltrige Theorie der nichtfundierten Algebra hat sich die Coalgebra als Semantik rekursiver Typen bisher kaum durchsetzen können, was sich in einer verschwindenden Zahl von Anhängern und Publikationen niederschlägt: Am 27. Januar 2006 fand die Internet-Suchmaschine GOOGLE beispielsweise etwa 544.000 Treffer für den Suchbegriff *primitive recursion*, aber nur 593 Treffer für *primitive corecursion*. In den folgenden beiden Abschnitten sollen die beiden Sichtweisen daher noch einmal kurz an Beispielen illustriert werden. Wir beschränken uns dabei auf die Klasse der primitiv (co)rekursiven Funktionen, die sich mit den bisher vorgestellten theoretischen Mitteln besonders gut beschreiben lassen, und bis auf wenige pathologische Exemplare alle praktisch denkbaren Funktionen einschließen.

2.8.2 Primitive Rekursion

Primitiv rekursive Funktionen sind eine Klasse von berechenbaren Funktionen, die über der initialen Struktur ihres Argumentbereichs definiert sind. Nur wenige berechenbare Funktionen gehören nicht dieser Klasse an.

Eine semantisch primitiv rekursive Funktion kann in einer syntaxorientierten Sicht auf Algebra durch eine syntaktisch rekursive Definition einer bestimmten Form beschrieben werden. In kategorieller Algebra entspricht dieser Definition die Homomorphiegleichung eines Katamorphismus. Jede Algebra erzeugt

also durch ihren Katamorphismus eine primitiv rekursive Funktion. Um das Verhältnis zwischen erzeugender Algebra und erzeugter Funktion und die Wirkungsweise der definierenden Gleichung näher zu beleuchten, betrachten wir ein Beispiel, das wegen seiner Symmetrie im nächsten Abschnitt auch für primitive Corekursion herhalten kann.

Dazu betrachten wir den bereits aus Beispiel 2.105 bekannten gemeinsamen Signaturfunktor der natürlichen Zahlen und Listen von solchen:

$$\Sigma = \Sigma_{nat} + \Sigma_{list} \qquad \Sigma_{nat} = \mathcal{C}_1 + \mathcal{I} \qquad \Sigma_{list} = \mathcal{C}_1 + \mathcal{I}^2$$

Für die Termnotation schreiben wir kurz:

$$\mathsf{zero} = (\iota_{nat} \circ \iota_1)(\star) \qquad \mathsf{nil} = (\iota_{list} \circ \iota_1)(\star)$$
$$\mathsf{succ} = \iota_{nat} \circ \iota_2 \qquad \mathsf{cons} = \iota_{list} \circ \iota_2$$

Man beachte, daß es sich bei damit gebildeten Ausdrücken um Datensätze aus $\Sigma(X)$, nicht um Trägerelemente aus X handelt. Um Elemente zu konstruieren, muß in der finalen Coalgebra $\mathbb{F}$ noch die inverse Beobachtungsoperation $\phi^{-1} : \Sigma(F) \to F$, beziehungsweise im algebraischem Anteil die Operation $\iota : \Sigma(I) \to I$ der zugehörigen initialen Algebra nachgeschaltet werden. Wir schreiben $zero = \phi^{-1}(\mathsf{zero})$ und so weiter.

Die gesuchten initialen Typen *nat* und *list* ergeben sich als kleinste Lösungen des folgenden Gleichungssystems:

$$T_{nat} = \{zero\} \cup \mathcal{P}(succ)(T_{nat})$$
$$T_{list} = \{nil\} \cup \mathcal{P}(cons)(T_{nat} \times T_{list})$$

Dabei definiert T_{nat} auch eine initiale Algebra von Σ_{nat}, und T_{list} eine initiale Algebra der Signatur

$$\Sigma'_{list} = \mathcal{C}_1 + \mathcal{C}_{T_{nat}} \times \mathcal{I}$$

Nun wollen wir die Elemente einer Liste zählen, also folgende Funktion definieren:

$$length : T_{list} \to T_{nat}$$

Dies gelingt induktiv über der Signatur Σ'_{list} des Argumenttyps. Die rekursive Definition in klassischer Notation lautet:

$$length(nil) = zero \qquad length\big(cons(x, \underline{\vec{y}})\big) = succ\big(\underline{length(\vec{y})}\big)$$

Dabei sind Argument und Resultat der Rekursion unterstrichen. Eine katamorphe Definition durch eine algebraische Operation λ hat für beide nur einen gemeinsamen Platzhalter vom Resultattyp:

$$\lambda(\mathsf{nil}) = zero \qquad \lambda\big(\mathsf{cons}(x, \underline{n})\big) = succ(\underline{n})$$

Damit ergibt sich eine Σ'_{list}-Algebra $\mathbb{L} = (T_{nat}, \lambda)$. Bei genauerer Betrachtung der Operation λ fällt eine eigenartige Typisierung auf:

$$\lambda : \Sigma'_{list}(T_{nat}) \to T_{nat}$$

Der Argumenttyp ist weder der Typ der natürlichen Zahlen, noch der der Listen, sondern ein Bereich hybrider Datensätze, dessen Elemente als Wurzel den Konstruktor einer Liste, aber als Teilterme dann natürliche Zahlen besitzen. Die Operation λ transformiert also nur die oberste Termebene ihrer Argumente. Der rekursive Abschluß wird durch den Katamorphismus $(\!|\mathbb{L}|\!)$ geleistet, wie die folgende Rechnung zeigt:

$$(\!|\mathbb{L}|\!) \circ \iota = \lambda \circ \Sigma'_{list}\big((\!|\mathbb{L}|\!)\big)$$

$$\begin{aligned}(\!|\mathbb{L}|\!)(nil) &= \lambda\big(\Sigma'_{list}\big((\!|\mathbb{L}|\!)\big)(\mathsf{nil})\big)\\ &= \lambda(\mathsf{nil})\\ &= zero\end{aligned} \qquad \begin{aligned}(\!|\mathbb{L}|\!)\big(cons(x,\vec{y})\big) &= \lambda\big(\Sigma'_{list}\big((\!|\mathbb{L}|\!)\big)\big(\mathsf{cons}(x,\vec{y})\big)\big)\\ &= \lambda\big(\mathsf{cons}\big(x,(\!|\mathbb{L}|\!)(\vec{y})\big)\big)\\ &= succ\big((\!|\mathbb{L}|\!)(\vec{y})\big)\end{aligned}$$

Durch die Struktur der Definition ist eine praktische Vorgehensweise bereits fest vorgegeben: Betrachtet man ι als Laufzeitsystem einer Ausführungsumgebung und λ als Code des Rumpfs der Funktion $(\!|\mathbb{L}|\!)$, dann ist durch die komponierte Abbildung $\lambda \circ \Sigma'_{list}\big((\!|\mathbb{L}|\!)\big)$ eine Traversierung der Eingabe von den Blättern zur Wurzel (*bottom-up*) spezifiziert, da im Allgemeinen die innere Abbildung $\Sigma'_{list}\big((\!|\mathbb{L}|\!)\big)$ vor der äußeren Abbildung λ ausgewertet werden muß. In jedem Schritt wird also durch Applikation der Schrittoperation der oberste Konstruktor ersetzt, unter der Bedingung, daß zuvor die Argumente ausgewertet wurden. Da im betrachteten Datenbereich keine unendlichen Teiltermketten existieren, terminiert die Traversierung für alle finitären, also insbesondere auch für alle polynomiellen Signaturen.

Beispiel 2.131 (Rekursives Zählen). Das primitiv rekursive Zählen einer zweielementigen Liste $[17, 4]$:

$$\begin{aligned}(\!|\mathbb{L}|\!)\big(cons(17, cons(4, nil))\big) &= \lambda\big(\mathsf{cons}\big(17,(\!|\mathbb{L}|\!)\big(cons(4,nil)\big)\big)\big)\\ &= \lambda\big(\mathsf{cons}\big(17,\lambda\big(\mathsf{cons}\big(4,(\!|\mathbb{L}|\!)(nil)\big)\big)\big)\big)\\ &= \lambda\big(\mathsf{cons}\big(17,\lambda(\mathsf{cons}(4,zero))\big)\big)\\ &= \lambda\big(\mathsf{cons}\big(17,succ(zero)\big)\big)\\ &= succ\big(succ(zero)\big)\end{aligned}$$

□

2.8.3 Primitive Corekursion

Für den dualen Begriff der primitiven Corekursion können wir große Teile des Beispielszenarios wiederverwenden. Funktionen wie *length*, die zugleich primitiv rekursiv und corekursiv sind, haben eine Reihe angenehmer Eigenschaften. Diese werden in Kapitel 6 noch eine Rolle spielen.

Die initialen Typen erweitern wir zu finalen, also zu den größten Lösungen des Typgleichungssystems. Dementsprechend schreiben wir Konstruktoren $zero = \phi^{-1}(\mathsf{zero})$ und so weiter. Die anamorphe Definition durch eine coalgebraische Operation sieht der katamorphen recht ähnlich, nur daß die Rekursion durch einen Platzhalter von Argumenttyp markiert wird:

$$\lambda'(nil) = \mathsf{zero} \qquad \lambda'\big(cons(x,\vec{y})\big) = \mathsf{succ}(\underline{\vec{y}})$$

Der Typ dieser Operation ist ebenfalls hybrid:

$$\lambda' : T_{\mathsf{list}} \to \Sigma_{nat}(T_{list})$$

Jedem Argument wird also ein Datensatz zugeordnet, der als Wurzel bereits den Konstruktor einer natürlichen Zahl, als Argumente aber noch Listen hat. Wir vergewissern uns, daß die Σ_{nat}-Coalgebra $\mathbb{L}' = (T_{list}, \lambda')$ tatsächlich die gewünschte Funktion erzeugt:

$$\phi \circ [\![\mathbb{L}']\!] = \Sigma_{nat}\big([\![\mathbb{L}']\!]\big) \circ \lambda'$$

$$\begin{aligned}\phi\big([\![\mathbb{L}']\!](nil)\big) &= \Sigma_{nat}\big([\![\mathbb{L}']\!]\big)\big(\lambda'(nil)\big)\\ &= \Sigma_{nat}\big([\![\mathbb{L}']\!]\big)(\mathsf{zero})\\ &= \mathsf{zero}\end{aligned} \qquad \begin{aligned}\phi\big([\![\mathbb{L}']\!]\big(cons(x,\vec{y})\big)\big) &= \Sigma_{nat}\big([\![\mathbb{L}']\!]\big)\big(\lambda'\big(cons(x,\vec{y})\big)\big)\\ &= \Sigma_{nat}\big([\![\mathbb{L}']\!]\big)\big(\mathsf{succ}(\vec{y})\big)\\ &= \mathsf{succ}\big([\![\mathbb{L}']\!](\vec{y})\big)\end{aligned}$$

Die durch diese Definition festgelegte Traversierungsreihenfolge verhält sich umgekehrt zur primitiven Rekursion, also von der Wurzel zu den Blättern (*top-down*). Entsprechend der Sequenz $\Sigma_{nat}\big([\![\mathbb{L}']\!]\big) \circ \lambda'$

wird zuerst in jedem Schritt der Wurzelkonstruktor ersetzt. Die Argumente bleiben zunächst stehen, um später durch corekursive Aufrufe ersetzt zu werden.

Da die Ergebnisse einer so definierten Funktion als Elemente eines finalen Datentyps unendlich sein können, kann nicht einfach von Termination im Sinne der Reduktion zu einer vollen Normalform in endlicher Zeit gesprochen werden. Der duale Begriff ist die in [60] eingeführte *Produktivität*. Demnach heißt eine Berechnung produktiv, wenn jeder erreichbare Teilausdruck in endlicher Zeit zu einer schwachen Kopf-Normalform reduziert werden kann, das heißt, wenn die Errechnung jedes vorkommenden Konstruktors ohne Reduktion der Argumente terminiert. Wir werden später zeigen, daß im Bereich der rationalen Datentypen Termination und Produktivität zusammenfallen, und die Vorzüge von beiden effektiv kombiniert werden können.

Beispiel 2.132 (Corekursives Zählen (endlich)). Das primitiv corekursive Zählen einer zweielementigen Liste $[17, 4]$:

$$\begin{aligned}[\![\mathbb{L}']\!]\big(cons(17, cons(4, nil))\big) &= succ\big([\![\mathbb{L}']\!]\big(cons(4, nil)\big)\big)\\ &= succ\big(succ\big([\![\mathbb{L}']\!](nil)\big)\big)\\ &= succ(succ(zero))\end{aligned}$$

Hier fällt auf, daß bei der Reduktion jeweils die Kontextinformation umgebender corekursiver Aufrufe sofort entfallen kann: keiner der Ausdrücke auf der rechten Seite enthält eine unreduzierte Anwendung von λ'. Dieser Effekt wird bei der Übersetzung von Programmiersprachen für die Transformation von Rekursionen in Schleifen (Elimination von *tail*-Aufrufen) ausgenutzt, wenn die Rekursion nicht direkt in *tail*-Position, sondern unter einem Konstruktor steht, wie es für die Zählfunktion der Fall ist. Tatsächlich ist nur die primitive Corekursion, nicht aber die primitive Rekursion, für solche erheblichen Effizienzsteigerungen geeignet. Bei der sogenannten Elimination von *tail*-Aufrufen *modulo Konstruktor* zur Optimierung primitiv rekursiver Funktionen handelt es sich um nichts anderes als eine Umformung zur primitiven Corekursion. □

Corekursion bietet zusätzlich noch den Mehrwert des erweiterten finalen Wertebereichs.

Beispiel 2.133 (Corekursives Zählen (unendlich)). Das Zählen der Liste aller Primzahlen $\vec{p} = [2, 3, 5, \dots]$:

$$\begin{aligned}[\![\mathbb{L}']\!]\big(cons(2, \dots)\big) &= succ\big([\![\mathbb{L}']\!]\big(cons(3, \dots)\big)\big)\\ &= succ(succ(\dots))\end{aligned}$$

Anscheinend ergibt sich eine unendliche große Zahl. Diese Annahme läßt sich auch formal untersuchen, und zwar innerhalb des finalen Bezugssystems ohne Anwendung von Grenzwertbegriffen oder ähnlichem.

1. Sei für alle $k \geq 0$ $\vec{q}_k$ der Rest von $\vec{p}$ nach dem Streichen der ersten k Primzahlen, also $\vec{q}_0 = \vec{p}$ und $\vec{q}_k = p_{k+1} : \vec{q}_{k+1}$, wobei q_k die k-te Primzahl ist. Es gilt $\phi(\vec{q}_k) \in \operatorname{codom} \mathsf{cons}$, also $[\![\mathbb{L}']\!](\vec{q}_k) \in \operatorname{codom} \mathsf{succ}$.

2. Sei andererseits $\omega \in T_{nat}$ mit $\phi(\omega) = \mathsf{succ}(\omega)$. Bekanntermaßen enthält der finale Datentyp genau diese eine unendliche Zahl.

Die Menge $Y = \big\{\big([\![\mathbb{L}']\!](\vec{q}_k), \omega\big) \mid k \geq 0\big\}$ ist also punkweise kompatibel. Insgesamt ist $\mathbb{Y} = \big(Y, \upsilon(\mathbb{F}, \mathbb{F})\big)$ eine Bisimulation, und wegen der Finalität gilt $[\![\mathbb{L}']\!](\vec{q}_k) = \omega$. Der Nachweis der Bisimulationseigenschaft umfaßt allerdings einen unendlichen Suchraum, und ist daher eher der Metaebene der Verifikation als der Objektebene eines programmierten Tests zuzuordnen. □

Der erheblich erweiterte finale Datenbereich wird mit dem Problem der Überabzählbarkeit erkauft:

1. Viele Elemente, die zwar durch eine produktive Berechnung beschrieben werden können, besitzen aber keine endliche extensionale Repräsentation als Terme oder Coterme, beispielsweise die Liste der Primzahlen. Folglich können Daten nicht unabhängig von Berechnungen betrachtet werden.

2. Semantische Gleichheit ist im Allgemeinen unentscheidbar.

Diese schwerwiegenden Konsequenzen, die in der nichtstrikten Programmierung hingenommen werden, gelten allerdings nicht im eingeschränkten Bereich der rationalen Datentypen. Dieser ist vollständig durch endliche Coterme beschrieben, die wie die Terme initialer Datentypen erschöpfend aufgespannt und traversiert werden können.

Beispiel 2.134 (Corekursives Zählen (rational)). Das Zählen einer Liste $\vec{u}$ von Einsen, also $\phi(\vec{u}) = \mathsf{cons}(1, \vec{u})$:

$$\begin{aligned}\phi\big([\![\mathbb{L}']\!](\vec{u})\big) &= \Sigma\big([\![\mathbb{L}']\!]\big)\big(\lambda'\big(cons(1,\vec{u})\big)\big)\\ &= \Sigma\big([\![\mathbb{L}']\!]\big)\big(\mathsf{succ}(\vec{u})\big)\\ &= \mathsf{succ}\big([\![\mathbb{L}']\!](\vec{u})\big)\end{aligned}$$

Also gilt $\phi(\omega) = \mathsf{succ}(\omega)$ und $\phi\big([\![\mathbb{L}']\!](\vec{u})\big) = \mathsf{succ}\big([\![\mathbb{L}']\!](\vec{u})\big)$. Die Bisimilarität der beiden läßt sich durch endliche Traversierung der beiden Cotermstrukturen nachweisen. Dieser Vorgang ist automatisierbar und kann zum Entscheiden von Prädikaten auf der Objektebene einer Programmiersprache dienen. □

Kapitel 4 wird sich mit einem allgemeinen Verfahren zur Implementierung rational corekursiver Funktionen befassen, Kapitel 5 mit dem Entscheiden von Prädikaten auf rationalen Daten.

Kapitel 3

Coalgebra des Arbeitsspeichers

3.1 Graphische Struktur

Der Zustand eines Arbeitsspeichers läßt sich adäquat mit graphischen Metaphern beschreiben:

- Ein *Adressraum* liefert *Adressen*, die zur Identifikation von *Zellen* dienen. Die Menge der Adressen, welche tatsächlich lebende Zellen bezeichnen, stellt die Knotenmenge des Speichergraphen dar.
- Der Inhalt einer Zelle besteht aus *atomaren* Anteilen, die durch ihren numerischen Wert codierte Information tragen, und *referentiellen* Anteilen, die die Adressen anderer Zellen enthalten.
  - Der atomare Zelleninhalt kann als Knotenbeschriftung aufgefaßt werden.
  - Der referentielle Zelleninhalt spezifiziert die Nachfolger eines Knotens. Somit repräsentieren gespeicherte Adressen die Endpunkte gerichteter Kanten.

Die Algebra der Zelleninhalte alleine ist keine ausreichende Metatheorie für die Beschreibung von Speicherzuständen:

1. Baumförmige Strukturen können in fundierter Algebra abstrakt als Terme dargestellt werden.
2. Andere Strukturen können nicht exakt, sondern nur approximiert durch unendliche Bäume in nicht fundierter Algebra als Terme dargestellt werden.
3. Graphstrukturen mit expliziten Zyklen oder gemeinsamen Anteilen (*sharing*) können konkret nur durch Offenlegung der Zellenidentitäten und deren Belegung dargestellt werden.

Dabei ist die Beziehung zwischen konkreter und abstrakter Darstellung eine komplexe Konstruktion, die bei genauerer Betrachtung exakt dem coalgebraischen Anamorphismus entspricht. In diesem Kapitel soll eine geschlossene, pur coalgebraische Speichertheorie entwickelt werden. Parallel zur mathematischen Spezifikation wird eine ausführbare Haskell-Modellierung entwickelt, welche die Grundlage der in späteren Kapiteln folgenden Algorithmen bildet.

3.1.1 Termgraphen

Die populärsten Modelle für die operationale Semantik nichtstrikter Programmiersprachen, beispielsweise die G-Maschine[46], beruhen auf der Reduktion von Graphen, genauer von sogenannten *Termgraphen*. Spezifisch für die Termgraphen ist dabei, daß die Umgebung eines Knotens nicht einfach eine Menge von Nachbarknoten ist, sondern eine Struktur besitzt. Ein Termgraph ist also nichts anderes als eine Coalgebra, wobei die Struktur dem lokalen Anteil und die Nachbarschaft von Knoten der Erreichbarkeit von Elementen entspricht. Durch Auszeichnung einer Wurzel wird der Termgraph sogar zum Coterm.

3.1.2 Zeigergraphen

Auch das imperative Paradigma bedient sich der Graphmetapher, um Datenstrukturen zu beschreiben, die Querverweise in Form von Adressen der Nachbarzellen, sogenannten Zeigern, enthalten. Wie man auf der Nachbarschaftsrelation der Knoten formale Spezifikationen imperativer Verfahren aufbauen kann, ist bei MÖLLER nachzulesen, so beispielsweise in [41] die Herleitung typischer Algorithmen auf verketten Listen. Unter dem funktionalen Paradigma, in dem die Adresse einer Zelle lediglich ein technisches Vehikel, nicht aber eine durch den Benutzer meßbare Größe sein soll, ist aber eine coalgebraische Sicht der relational-algebraischen aus folgenden Gründen vorzuziehen:

1. In realen Maschinen ist der Adressraum des Speichers gegeben, der Inhalt der adressierten Zellen davon kausal abhängig. Auch ist es üblich, eine Referenz nur in der referierenden Zelle zu speichern, nicht aber in der referierten. Die so bevorzugte Leserichtung der Nachbarschaftsrelation wird durch die Operation einer Speichercoalgebra adäquat widergespiegelt.

2. Die Identität einer Zelle in Form ihrer Adresse soll vor dem Benutzer aus mehreren Gründen verborgen werden: Relokation durch Speicherverwaltung, Mehrfachverwendung eines Teilwerts (*sharing*) und die sichere Simulation von Zuweisungen durch partielles Kopieren sollen transparent erscheinen. Andererseits ist die Zellenidentität eine für die Sprachimplementierung wertvolle Information, die bekanntermaßen zur Effizienzsteigerung verwendet werden kann, wie etwa als hinreichendes Kriterium für Gleichheit oder als Suchschlüssel für gespeicherte vorberechnete Ergebnisse (*memoization*). Zur Erkennung von Zyklen ist sie sogar absolut notwendig, deshalb versagen algebraische Ansätze hier, es sei denn, sie legen die Identität in dem einer Zelle zugeordneten Datensatz offen.

   Dieses Dilemma wird elegant durch eine doppelte coalgebraische Modellierung in Form einer nichtfinalen Coalgebra der Repräsentation und ihrer finalen Semantik gelöst.

3.2 Coalgebraische Speicherzustände

3.2.1 Direkte und Indirekte Adressierung

Im einfachsten Fall einer coalgebraischen Modellierung werden Knotenbeschriftung und Nachfolger durch eine einzige Operation spezifiziert. Die Strukturierung der Werte ist durch einen polynomiellen Signaturfunktor Σ beschrieben:

- Atomare Anteile vom Typ A entsprechen Vorkommen von $\mathcal{C}_A$ in Σ.
- Referentielle Anteile entsprechen Vorkommen von $\mathcal{I}$ in Σ.

Definition 3.1 (Direkte Adressierung)**.** Sei Σ ein Signaturfunktor. Sei $\mathcal{A}$ eine abzählbar unendliche Menge von *Adressen*. Ein *Speicherzustand mit direkter Addressierung* über Σ und $\mathcal{A}$ ist definiert durch ein Paar $\mathbb{A} = \left(\sqrt{A}, \alpha\right)$ mit folgenden Komponenten:

1. eine Multimenge $\sqrt{A} : \mathcal{N}^{\mathcal{A}}$, genannt die *Wurzeln* ($\sqrt{A}$ ist ein zusammenhängendes Symbol, nicht die Anwendung einer Operation $\sqrt{\ }$),
2. eine partielle Abbildung $\alpha : \mathcal{A} \rightharpoonup \Sigma(\mathcal{A})$, genannt *Belegung*,

Der Speicherzustand heißt *legal*, falls eine Trägermenge $A \supseteq \operatorname{dom}\sqrt{A}$ existiert, so daß (A, α) eine rationale Σ-Coalgebra ist. Dies beinhaltet insbesondere $A \subseteq \operatorname{dom}\alpha$. Die eindeutige minimale schreiben wir ebenfalls als $\mathbb{A}$. Für diese gilt $A = \mathcal{P}(\rightarrow_{\mathbb{A}}^{*})(\operatorname{dom}\sqrt{A})$. Die Elemente von A heißen *lebende* Zellen. □

Für die Modelle realer Speicherzustände gilt außerdem, daß sie nur endlich viele belegte Zellen besitzen.

Lemma 3.2. *Sei $\mathbb{A} = \left(\sqrt{A}, \alpha\right)$ ein legaler Speicherzustand. Ist $\sqrt{A}$ endlich, dann ist auch die zugehörige Coalgebra endlich.*

Aus der coalgebraischen Struktur eines legalen Speicherzustandes folgt insbesondere, daß in lebenden Zellen keine ungültigen (*dangling*) Referenzen existieren. Dies ist offensichtlich ein stark idealisiertes Speichermodell, wie es einem deklarativen Programmierstil angemessen ist.

Der Nachteil der direkt adressierten Modellierung ist, daß Kanten keine unmittelbare Identität besitzen. Eine Zustandsübergangsrelation, bei welcher der Endpunkt einer einzelnen Kante verändert wird (durch *Zuweisung* einer Referenz), kann nicht auf einfache Weise spezifiziert werden.

Dieses Problem läßt sich durch die Einführung eines zweiten Addressraumes lösen, welcher anstellen von Zellen *Variable* identifizert. Eine Variable ist dabei ein Bestandteil einer Zelle, welcher genau eine Referenz speichert. Der Inhalt einer Zelle wird dann aufgelöst in seinen atomaren Anteil und die Adressen der Variablen, welche den referentiellen Anteil ausmachen, so daß diese unabhängig voneinander betrachtet werden können.

Definition 3.3 (Indirekte Adressierung)**.** Sei Σ ein Signaturfunktor. Seien $\mathcal{A}, \mathcal{B}$ zwei abzählbar unendliche Mengen von *Adressen*. Ein Σ-*Speicherzustand mit indirekter Addressierung* über Σ, $\mathcal{A}$ und $\mathcal{B}$ ist definiert durch ein Tripel $\mathbb{A} = \left(\sqrt{A}, \alpha_1, \alpha_2\right)$ mit folgenden Komponenten:

1. eine Multimenge $\sqrt{A} : \mathcal{N}^{\mathcal{A}}$, genannt die *Wurzeln*,
2. eine partielle Abbildung $\alpha_1 : \mathcal{A} \rightharpoonup \Sigma(\mathcal{B})$, genannt *Zellenbelegung*,
3. eine partielle Abbildung $\alpha_2 : \mathcal{B} \rightharpoonup \mathcal{A}$, genannt *Variablenbelegung*.

Als approximative Entsprechung der direkten Belegung definieren wir $\alpha_{12} = \Sigma(\alpha_2) \circ \alpha_1$. Dabei ist zu beachten, daß $\Sigma(\alpha_2)$ nur für ein totalisiertes α_2 definiert ist, also $\alpha_{12} : \mathcal{A} \rightharpoonup \Sigma(\mathcal{A} \uplus \{\bot\})$. Wir wählen eine *strikte* Einbettung $\alpha : \mathcal{A} \rightharpoonup \Sigma(\mathcal{A})$ mit:

$$\alpha(x) = \begin{cases} \alpha_{12}(x) & \alpha_{12}(x) \in \Sigma(\mathcal{A}) \\ \bot & \text{sonst} \end{cases}$$

```
newtype Cell  = Cell Int deriving (Enum, Eq, Ord, Show)
newtype Field = Field Int deriving (Enum, Eq, Ord, Show)
```

Programm 3.1: Zellen- und Variablenadressen

Oder äquivalent, mit Hilfe syntaktischer Erreichbarkeit ausgedrückt: $\alpha(x) = \bot \iff \bot \in \bar{\mathfrak{s}}(\Sigma)\big(\alpha_{12}(x)\big)$.

Der Speicherzustand heißt *legal*, falls gilt:

1. Es existiert eine Trägermenge $A \supseteq \operatorname{dom}\sqrt{A}$, so daß (A, α) eine rationale Σ-Coalgebra ist. Wie oben ist die minimale Lösung eindeutig und wird ebenfalls als $\mathbb{A}$ geschrieben.

2. Für alle definierten Zellen $x \in \operatorname{dom}\alpha_1$ ist der Inhalt $\alpha_1(x)$ linear gemäß Definition 2.95. Damit ist gewährleistet, daß die Zelle x durch Wahl eines geeigneten α_2 mit jedem kompatiblen Inhalt belegt werden kann. Dies entspricht auch der Intuition, daß sich Variablen innerhalb einer Zelle nicht überlappen.

3. Die Variablenmengen von definierten Zellen $x, y \in \operatorname{dom}\alpha_1$ sind paarweise disjunkt:

$$x \neq y \implies \bar{\mathfrak{s}}(\Sigma)\big(\alpha_1(x)\big) \cap \bar{\mathfrak{s}}(\Sigma)\big(\alpha_1(y)\big) = \emptyset$$

   Daraus ergibt sich eine Abbildung $z_\alpha : \operatorname{dom}\alpha_2 \to \mathcal{A}$, welche jeder belegten Variable die umgebende Zelle zuordnet:

$$z_\alpha(r) = x \iff r \in \bar{\mathfrak{s}}(\Sigma)\big(\alpha_1(x)\big)$$

Dies entspricht auch der Intuition, daß sich Zellen nicht überlappen. Die Elemente der Trägermenge $A \subseteq \mathcal{A}$ der zu einem legalen Speicherzustand gehörigen Coalgebra heißen *lebende Zellen*. Die Elemente der Menge $\bigcup \mathcal{P}\big(\bar{\mathfrak{s}}(\Sigma) \circ \alpha_1\big)(A) \subseteq \mathcal{B}$ heißen *lebende Variable*. □

Direkte Addressierung kann durch indirekte Adressierung simuliert werden: Sei $\mathbb{A} = (\sqrt{A}, \alpha)$ ein legaler direkt adressierter Speicherzustand über Σ und $\mathbb{A}$. Dann läßt sich ein äquivalenter, legaler indirekt adressierter Speicherzustand $(\sqrt{A}, \alpha_1, \alpha_2)$ wie folgt konstruieren:

1. Setze $\mathcal{B} = \mathcal{A} \times \mathcal{N}$.

2. Wähle für jedes $x \in \operatorname{dom}\alpha$ eine injektive Aufzählung $i_x : \bar{\mathfrak{s}}(\Sigma)\big(\alpha(x)\big) \to \mathcal{N}$.

3. Setze $\alpha_1(x) = \Sigma(const_x \times i_x)\big(\alpha(x)\big)$.

4. Setze $\alpha_2\big((x, j)\big) = i_x^{-1}(j)$.

Umgekehrt gilt per Definition, daß ein legaler indirekt adressierter Speicherzustand $(\sqrt{A}, \alpha_1, \alpha_2)$ stets unmittelbar einen legalen direkt adressierten Speicherzustand $(\sqrt{A}, \alpha)$ definiert.

Im Folgenden können daher alle Speicherzustände je nach Kontext als direkt oder indirekt adressiert betrachtet werden.

3.2.2 Nullreferenzen

Beim Vergleich des indirekt adressierten coalgebraischen Modells mit einem traditionellen Speichermodell, etwa dem der Sprache C, fällt eine Einschränkung auf: Variable können traditionell neben gültigen Adressen auch einen undefinierten Wert, die *Nullreferenz* enthalten. Diese erfüllt zwei verschiedene Zwecke, welche in einem präzisen formalen System nicht miteinander verwechselt werden sollten:

```
module Heap where
import Signature
data Signature σ ⇒ Heap σ = Heap {
  nextCellHeap  :: Cell,
  nextFieldHeap :: Field,
  cellsHeap     :: FiniteMap Cell (σ Field),
  fieldsHeap    :: FiniteMap Field Cell
}
emptyHeap :: Signature σ ⇒ Heap σ
emptyHeap = Heap (Cell 0) (Field 0) emptyFM emptyFM
type HeapOp σ = State (Heap σ)
```

Programm 3.2: Speicherzustände

```
carrierHeap   :: Signature σ ⇒ Heap σ → Set Cell
carrierHeap h = mkSet (keysFM (cellsHeap h))
opnHeap       :: Signature σ ⇒ Heap σ → Cell → σ Cell
opnHeap h     = let
                  opn1 = lookupWithDefaultFM (cellsHeap h) undefined
                  opn2 = lookupWithDefaultFM (fieldsHeap h) undefined
                in
                fmap opn2 ∘ opn1
```

Programm 3.3: Speicherzustand als Coalgebra

```
freshCell    :: Signature σ ⇒ HeapOp σ Cell
freshCell    = State doIt
  where
    doIt s = let c = nextCellHeap s
             in (c, s { nextCellHeap = succ c })
freshField   :: Signature σ ⇒ HeapOp σ Field
freshField   = ...

setCell      :: Signature σ ⇒ Cell → σ Field → HeapOp σ ()
setCell x y  = modify doIt
  where
    doIt s = let cs = cellsHeap s
             in s { cellsHeap = addToFM cs x y }
setField     :: Signature σ ⇒ Field → Cell → HeapOp σ ()
setField v x = ...
```

Programm 3.4: Aufbau von Belegungen

```
getCell          :: Signature σ ⇒ Cell → HeapOp σ (σ Field)
getCell x        = State doIt
  where
    doIt h = (lookupWithDefaultFM (cellsHeap h) undefined x, h)
getField         :: Signature σ ⇒ Field → HeapOp σ Cell
getField v       = ...

getCellFields    :: Signature σ ⇒ Cell → HeapOp σ (σ Cell)
getCellFields x = getCell x ≫= fmapM getField
```

Programm 3.5: Zugriff auf Belegungen

1. Sie repräsentiert illegale oder uninitialisierte Zwischenzustände einer Variablen, also vorübergehende Verletzungen der Invariante, daß alle lebenden Variablen belegt sein sollen. Von solchen Situationen kann in einer höheren Sicht auf Speicherzustände abstrahiert werden, indem nur solche Übergänge betrachtet werden, nach welchen die Invariante (wieder) gilt.

2. Sie repräsentiert die implizite Erweiterung des dauerhaften Wertebereichs einer Variablen. Diese Verwendung läßt sich besser explizit durch Verwendung des Signaturfunktors $\mathcal{I} + \mathcal{C}_1$ anstelle von $\mathcal{I}$ ausdrücken.

3.2.3 Unverpackte Datentypen

In der Praxis existieren neben den Adressen von Speicherzellen auch andere Typen von Daten, deren Codierung vom Prozessor selbst unterstützt wird. Solche Datentypen, deren Werte nicht auf Zelleninhalte verweisen, sondern unabhängig vom Speicherinhalt für sich selbst stehen, werden als *unverpackt* (*unboxed*) bezeichnet.

Unverpackte Datentypen lassen sich leicht orthogonal zu Speicherzellen in ein coalgebraisches Modell integrieren. Sei $\mathbb{A} = \left(\sqrt{A}, \alpha\right)$ ein direkt oder indirekt adressierter Speicherzustand über Σ und $\mathcal{A}$. Werte eines zusätzlichen, unverpackten Typs T

1. werden Adressen gleichgeordnet: $\mathcal{A} \longmapsto \mathcal{A} + T$,

2. werden Zelleninhalten gleichgeordnet: $\Sigma \longmapsto \Sigma + \mathcal{C}_T$,

3. sind global verfügbar: $\sqrt{A} \longmapsto \sqrt{A} + T$,

4. stehen für sich selbst: $\alpha \longmapsto \alpha + id$.

3.3 Operationen auf Speicherzuständen

Operationen auf Speicherzuständen sollen als (paremetrische Familien von) Relationen modelliert werden, die einen Vor- und einen Nachzustand, sowie gegebenenfalls weitere Parameter in Beziehung setzen. Zunächst sollen elementare Eigenschaften von solchen Operationen eingeführt werden. Anschließend wird ein minimaler Satz von Grundoperationen zur Erzeugung beliebiger Speicherzustände definiert. Diese enthalten unter anderem beliebige Zuweisungen, sind also eher als operationale Semantik einer imperativen Sprache anzusehen. Da hier aber die wesentlicher eingeschränktere Modellierung funktionaler Programme im Vordergrund steht, werden keine Anstrengungen unternommen, um die Eigenschaften beliebiger Operationsfolgen abzuleiten. Stattdessen werden rigide hinreichende Bedingungen formuliert, so daß gewünschte Eigenschaften trivial gelten.

3.3.1 Eigenschaften

3.3.1.1 Sicherheit

Definition 3.4 (Sicherheit)**.** Eine binäre Relation auf Speicherzuständen heißt *sicher*, falls aus der Legalität des Vorzustands die Legalität des Nachzustands folgt. □

Eng mit den Begriffen der Legalität und Sicherheit ist die folgende Eigenschaft verbunden.

Definition 3.5 (Lebensfähigkeit)**.** Sei $\mathbb{A} = \left(\sqrt{A}, \alpha\right)$ ein legaler Speicherzustand. Eine Zelle x heißt in $\mathbb{A}$ *lebensfähig*, falls $\left(\sqrt{A} \cup \{x\}, \alpha\right)$ ebenfalls ein legaler Speicherzustand ist. □

Lemma 3.6. *Lebende Zellen in legalen Speicherzuständen sind stets lebensfähig.*

3.3.1.2 Referentielle Transparenz

Unter *referentieller Transparenz* versteht man die Eigenschaft eines Programms oder Ablaufs, daß die Bedeutung eines Ausdrucks, der Referenzen (auf Speicherzellen) enthält, sich nicht dynamisch ändert. Diese Eigenschaft ist notwendig für die korrekte Implementierung mathematischer Objekte (Terme, Coterme, Funktionen) mit Hilfe von Speicherzellen und Referenzen. Im Kontext des hier definierten coalgebraischen Speichermodells sind die potentiellen Referenzen die Adressen lebender Zellen, und ihre Bedeutung die zugeordneten Coterme.

Definition 3.7 (Referentielle Transparenz)**.** Der Übergang zwischen zwei legalen Speicherzuständen $\mathbb{A} \mapsto \mathbb{A}'$ heißt *referentiell transparent*, wenn er verträglich mit der eigentlichen Untercoalgebrenordnung $\subsetneqq$ ist: $\mathbb{A} \subsetneqq \mathbb{A}'$.

Eine binäre Relation R auf Speicherzuständen heißt *referentiell transparent*, wenn jeder Übergang $\mathbb{A}\, R\, \mathbb{A}'$ referentiell transparent ist.

Eine Folge $(\mathbb{A}^{(i)})$ von Speicherzuständen heißt *referentiell transparent*, wenn sie monoton bezüglich $\subsetneqq$ ist, also alle Übergänge $\mathbb{A}^{(i)} \mapsto \mathbb{A}^{(i+1)}$ transparent sind. □

```
heapPoset :: Signature σ ⇒ Poset (Heap σ)
heapPoset = Poset (≤) emptyHeap
  where
    h1 ≤ h2     = posetLE cellsPoset (dataHeap h1) (dataHeap h2)
    cellsPoset  = genericFlatMapPoset syntacticEquiv
    dataHeap h  = (carrierHeap h, opnHeap h)
```

Programm 3.6: Untercoalgebraordnung von Speicherzuständen

Die Relation referentieller Transparenz ist eine partielle Ordnung auf den Σ-Speicherzuständen. Programm 3.6 zeigt die Implementierung.

3.3.1.3 Funktionale Abhängigkeiten

Die in diesem Abschnitt eingeführten Eigenschaften formalisieren das Verhalten von mehrstelligen Relationen unter partieller Belegung ihrer Parameter. Die Vervollständigung der Belegung zu einem Punkt der Relation kann als Lösung einer Instanz des durch die Relation beschriebenen allgemeinen Problems aufgefaßt werden. Von besonderem Interesse sind Belegungen, deren Vervollständigung eindeutig oder immer möglich ist. Dazu definieren wir zunächst eine Operation, die es gestattet, die Komponenten eines Tupels *ad hoc* zu partitionieren.

Definition 3.8 (Offenes Tupel)**.** Sei $(A_i)_{i\in I}$ eine Familie von Mengen. Seien I_1, I_2 disjunkte Indexmengen mit $I = I_1 \uplus I_2$. Dann ist der *offene Tupelkonstruktor* wie folgt definiert:

$$\& : \prod_{i\in I_1} A_i \times \prod_{i\in I_2} A_i \to \prod_{i\in I} A_i$$

$$\pi_i(t_1 \,\&\, t_2) = \begin{cases} \pi_i(t_1) & i \in I_1 \\ \pi_i(t_2) & i \in I_2 \end{cases}$$ □

Lemma 3.9. *Der Konstruktor* & *ist assoziativ und kommutativ:*

$$t_1 \,\&\, (t_2 \,\&\, t_3) = (t_1 \,\&\, t_2) \,\&\, t_3 \qquad\qquad t_1 \,\&\, t_2 = t_2 \,\&\, t_1$$

Definition 3.10 (Abhängigkeiten)**.** Sei $(A_i)_{i\in I}$ eine Familie von Mengen und $R \subseteq \prod_{i\in I} A_i$ eine Relation. Ein Paar I^+, I^- von Indexmengen mit $I = I^+ \uplus I^-$ heißt

1. *partielle funktionale Abhängigkeit*, geschrieben $I^+ \twoheadrightarrow I^-$, falls gilt

$$\forall t^+ \in \prod_{i\in I^+} A_i.\ \left|\{t^- \in \prod_{i\in I^-} A_i \mid t^+ \,\&\, t^- \in R\}\right| \leq 1$$

2. *(totale) funktionale Abhängigkeit*, geschrieben $I^+ \to I^-$, falls gilt

$$\forall t^+ \in \prod_{i\in I^+} A_i.\ \left|\{t^- \in \prod_{i\in I^-} A_i \mid t^+ \,\&\, t^- \in R\}\right| = 1$$

3. *lösbare Abhängigkeit*, geschrieben $I^+ \dashrightarrow I^-$, falls gilt

$$\forall t^+ \in \prod_{i\in I^+} A_i.\ \left|\{t^- \in \prod_{i\in I^-} A_i \mid t^+ \,\&\, t^- \in R\}\right| \geq 1$$ □

Relationen werden üblicherweise als Teilmengen eines mit $\times$ gebildeten Produktes aufgefaßt. Die Indexmenge wäre im zweistelligen Fall $I = \{1, 2\}$, oder allgemeiner im n-stelligen Fall $\{1 \ldots n\}$. Falls eine Relation aber syntaktisch als Prädikat eingeführt wird, kann man zur besseren Lesbarkeit anstlle der Positionsnummern die Namen der formalen Parameter als Indizes auffassen.

Beispiel 3.11 (Indexnamen und Abhängigkeiten). Sei die Relation R mit dem Prädikat $R(x, y, z)$ notiert, für $x \in A_x$, $y \in A_y$ und $z \in A_z$. Dann ist $R \subseteq \prod_{i\in I} A_i$, wobei $I = \{x, y, z\}$. Ferner ist $R(a, b, c)$ Kurzschreibweise für

$$\exists\, r \in R.\ \pi_x(r) = a \wedge \pi_y(r) = b \wedge \pi_z(r) = c$$

Die Eigenschaft, daß die Belegung von x und y eindeutig die Belegung von z bestimmt, schreibt man dann

$$\{x, y\} \to \{z\} \qquad \text{oder einfach} \qquad x, y \to z$$ □

```
freshFields :: Signature σ ⇒ σ α → HeapOp σ (σ Field)
freshFields = fmapM (const freshField)
newCell     :: Signature σ ⇒ σ α → HeapOp σ (Cell, ITrans Field α)
newCell s   = do
                x ← freshCell
                t ← freshFields s
                setCell x t
                return (x, transData t s)
```

Programm 3.7: Erzeugen einer Zelle

3.3.1.4 Leben und Tod

Eine Zelle x kann auf verschiedene Weisen zwischen den Zuständen *lebend* und *tot* wechseln:

1. Eine tote Zelle wird lebendig, wenn ihre Adresse in einer lebenden Variablen gespeichert oder zur Wurzel gemacht wird, oder wenn eine diese Adresse enthaltende Variable lebendig wird.
2. Eine lebende Zelle stirbt, sobald alle Referenzen auf sie aus der Wurzelmenge entfernt, beziehungsweise die enthaltenden Variablen überschrieben oder tot sind.

3.3.2 Operationen

Für die folgenden Definitionen seien

1. $\mathbb{A} = \left(\sqrt{A}, \alpha_1, \alpha_2\right)$ und $\mathbb{A}' = \left(\sqrt{A'}, \alpha_1', \alpha_2'\right)$ beliebige Speicherzustände über Σ, $\mathcal{A}$ und $\mathcal{B}$,
2. $x \in \mathcal{A}, y \in \mathcal{A} \uplus \{\bot\}$ beliebige Zellenaddressen,
3. $v \in \mathcal{B}$ eine beliebige Variablenadresse,
4. $s \in \Sigma(B)$ ein beliebiger linearer Zelleninhalt.

3.3.2.1 Erzeugen einer Zelle

Definition 3.12. Wir schreiben $\mathbb{A} \overset{\mathsf{n}\, x:=s}{\longmapsto} \mathbb{A}'$, falls gilt:

1. Die Wurzeln bleiben unverändert: $\sqrt{A} = \sqrt{A'}$.
2. x ist im Nach-, aber nicht im Vorzustand belegt; ansonsten bleibt die Zellenbelegung unverändert: $\alpha_1' = \alpha_1 \uplus \{x \mapsto s\}$.
3. Die Variablen in s sind frisch: $\bigcup \mathcal{P}\left(\bar{\mathfrak{s}}(\Sigma)\right)(\operatorname{dom} \alpha_1) \cap \bar{\mathfrak{s}}(\Sigma)(s) = \emptyset$.
4. Die Variablenbelegung bleibt unverändert. Insbesondere sind die Variablen in s in Vor- und Nachzustand nicht belegt:

$$\alpha_2 = \alpha_2' \qquad\qquad \operatorname{dom} \alpha_2 \cap \bar{\mathfrak{s}}(\Sigma)(s) = \emptyset \qquad\qquad \square$$

Lemma 3.13. *Das Erzeugen einer Zelle ist sicher und transparent.*

Lemma 3.14. *Für einen Übergang $\mathbb{A} \overset{\mathsf{n}\, x:=s}{\longmapsto} \mathbb{A}'$ gelten die folgenden Abhängigkeiten:*

$$\mathbb{A}, x, s \nrightarrow \mathbb{A}' \qquad\qquad \mathbb{A}', x \nrightarrow \mathbb{A}, s$$
$$\mathbb{A}, \mathbb{A}' \nrightarrow x, s \qquad\qquad \mathbb{A}, s \dashrightarrow \mathbb{A}', x$$

Die letzte, lösbare Abhängigkeit ist für die Konstruktion von Algorithmen von besonderem Interesse, da sie ausdrückt, daß für jeden zu speichernden Inhalt stets eine Zelle gefunden werden kann.

3.3.2.2 Löschen einer Zelle

Definition 3.15. Wir schreiben $\mathbb{A} \overset{\mathsf{d}\,x}{\longmapsto} \mathbb{A}'$, falls gilt:

1. Die Wurzeln bleiben unverändert: $\sqrt{A} = \sqrt{A'}$.
2. x ist im Vor-, aber nicht im Nachzustand belegt: $\operatorname{dom}\alpha_1 = \operatorname{dom}\alpha_1' \uplus \{x\}$.
3. Dasselbe gilt für die Variablen in x: $\operatorname{dom}\alpha_2 = \operatorname{dom}\alpha_2' \uplus \bar{\mathfrak{s}}(\Sigma)\big(\alpha_1(x)\big)$.
4. Ansonsten bleiben die Belegungen unverändert:

$$\alpha_1' \sqsubseteq \alpha_1 \qquad\qquad \alpha_2' \sqsubseteq \alpha_2$$

□

Lemma 3.16. *Das Löschen einer Zelle ist sicher und transparent, falls sie tot ist.*

Lemma 3.17. *Für einen Übergang $\mathbb{A} \overset{\mathsf{d}\,x}{\longmapsto} \mathbb{A}'$ gelten folgende Abhängigkeiten:*

$$\mathbb{A}, x \nrightarrow \mathbb{A}' \qquad\qquad \mathbb{A}, \mathbb{A}' \nrightarrow x$$

3.3.2.3 Zuweisen einer Variablen

Definition 3.18. Wir schreiben $\mathbb{A} \overset{v:=y}{\longmapsto} \mathbb{A}'$, falls gilt:

1. Wurzeln und Zellenbelegung bleiben unverändert: $\sqrt{A} = \sqrt{A'}$, $\alpha_1 = \alpha_1'$.
2. v ist Bestandteil einer Zelle x: $x = z_\alpha(v) \neq \bot$.
3. v ist im Nachzustand mit y belegt, ansonsten bleibt die Variablenbelegung unverändert:

$$\alpha_2' = \begin{cases} \alpha_2 \ominus \{v\} & y = \bot \\ \alpha_2 \oplus \{v \mapsto y\} & y \neq \bot \end{cases}$$

Die Zuweisung heißt *redundant*, falls bereits im Vorzustand $\beta(v) = y$ gilt. □

Lemma 3.19. *Für redundante Zuweisungen $\mathbb{A} \overset{v:=y}{\longmapsto} \mathbb{A}'$ gilt $\mathbb{A} = \mathbb{A}'$.*

Lemma 3.20. *Die Sicherheit und Transparenz einer irredundanten Zuweisung $\mathbb{A} \overset{v:=y}{\longmapsto} \mathbb{A}'$ hängt vom Status der veränderten Zelle $x = z_\alpha(v)$ ab:*

1. *Ist x lebendig, dann ist die Zuweisung nur sicher, falls y lebensfähig ist; aber keinesfalls transparent.*
2. *Ist x tot, dann ist die Zuweisung einer beliebigen Zelle y sicher und transparent.*

Lemma 3.21. *Für einen irredundanten Übergang $\mathbb{A} \overset{v:=y}{\longmapsto} \mathbb{A}'$ gelten die folgenden Abhängigkeiten:*

$$\mathbb{A}, v, y \nrightarrow \mathbb{A}' \qquad\qquad \mathbb{A}', v \nrightarrow \mathbb{A}, y$$
$$\mathbb{A}, \mathbb{A}' \nrightarrow v, y$$

3.3.2.4 Hinzufügen einer Wurzel

Definition 3.22. Wir schreiben $\mathbb{A} \overset{\mathsf{p}\,x}{\longmapsto} \mathbb{A}'$, falls gilt:

1. x ist im Nachzustand einmal mehr Wurzel als im Vorzustand: $\sqrt{A'} = \sqrt{A} + \{x\}$.
2. x ist im Vor- und Nachzustand belegt: $x \in \operatorname{dom}\alpha \cap \operatorname{dom}\alpha'$.
3. Die Belegungen bleiben unverändert:

$$\alpha_1 = \alpha'_1 \qquad\qquad \alpha_2 = \alpha'_2$$

Das Hinzufügen heißt *redundant*, falls x bereits im Vorzustand Wurzel ist. □

Lemma 3.23. *Für redundantes Hinzufügen $\mathbb{A} \overset{\mathsf{p}\,x}{\longmapsto} \mathbb{A}'$ gilt, daß die Coalgebren von $\mathbb{A}$ und $\mathbb{A}'$ gleich sind.*

Lemma 3.24. *Das irredundante Hinzufügen einer Wurzel ist sicher und transparent, falls sie lebensfähig ist.*

3.3.2.5 Entfernen einer Wurzel

Definition 3.25. Wir schreiben $\mathbb{A} \overset{\mathsf{v}\,x}{\longmapsto} \mathbb{A}'$, falls gilt:

1. x ist im Vorzustand einmal mehr Wurzel als im Nachzustand: $\sqrt{A} = \sqrt{A'} + \{x\}$.
2. x ist im Vor- und Nachzustand belegt: $x \in \operatorname{dom}\alpha \cap \operatorname{dom}\alpha'$.
3. Die Belegungen bleiben unverändert:

$$\alpha_1 = \alpha'_2 \qquad\qquad \alpha_2 = \alpha'_2$$

Das Entfernen heißt *redundant*, falls x auch im Nachzustand noch Wurzel ist. □

Lemma 3.26. *Für redundantes Entfernen $\mathbb{A} \overset{\mathsf{v}\,x}{\longmapsto} \mathbb{A}'$ gilt, daß die Coalgebren von $\mathbb{A}$ und $\mathbb{A}'$ gleich sind.*

Lemma 3.27. *Das irredundante Entfernen einer Wurzel ist stets sicher. Es ist transparent, falls die Zelle dabei nicht stirbt.*

3.3.3 Abläufe

Die Grundoperationen lassen sich zu einer Relationenalgebra von Abläufen kombinieren, wobei die Komposition die wichtigste Verknüpfung darstellt.

Definition 3.28 (Sequenz)**.** Seien r, s zwei binäre Relationen auf Speicherzuständen. Wir schreiben $\mathbb{A} \overset{r;s}{\longmapsto} \mathbb{A}'$, falls ein Zwischenzustand $\mathbb{A}''$ existiert, so daß $\mathbb{A} \overset{r}{\mapsto} \mathbb{A}'' \overset{s}{\mapsto} \mathbb{A}'$. □

3.4 Speicherverwaltung

Die Grundlage einer automatischen Speicherverwaltung (*garbage collection*) besteht aus zwei Komponenten:

1. Das implizite Löschen von toten Zellen. Da einmal lebendige Zellen in transparenten Abläufen unsterblich sind, ist als Auslöser ein nichttransparenter Eingriff erforderlich:

2. Das explizite Entfernen von Wurzeln.

Eine transparente Speicherverwaltung ist offensichtlich trotzdem möglich, denn die Laufzeitumgebungen pur funktionaler Programmiersprachen beherrschen diesen Trick seit Jahrzehnten. Entscheidend ist dabei, daß Wurzeln nicht beliebig hinzugefügt und entfernt werden, sondern eine geschachtelte Lebensdauer besitzen. Technisch entspricht dieser Eigenschaft die Speicherung von Variablenbindungen auf einem *Stack*.

Definition 3.29 (Beschränkte Transparenz)**.** Seien $\mathbb{A} = \left(\sqrt{A}, \alpha\right)$ und $\mathbb{A}' = \left(\sqrt{A'}, \alpha'\right)$ zwei legale Speicherzustände über Σ und $\mathcal{A}$. Sei $S \subseteq A$. Der Übergang $\mathbb{A} \mapsto \mathbb{A}'$ heißt *S-beschränkt referentiell transparent*, falls gilt:

$$x \in S \implies \alpha(x) = \alpha'(x)$$

Zur Unterscheidung nennen wir Definition 3.7 hier *unbeschränkte* Transparenz. □

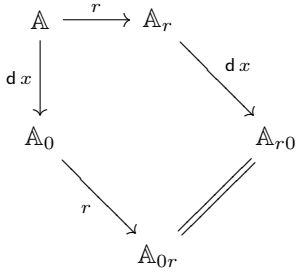
Lemma 3.30. *Ist der Übergang $\mathbb{A} \mapsto \mathbb{A}'$ S-beschränkt transparent, dann ist er auch S'-beschränkt transparent für alle $S' \subset S$. Außerdem ist er unbeschränkt transparent, genau dann wenn er A-beschränkt transparent ist.*

Satz 3.31. *Seien $\mathbb{A}$ und $\mathbb{A}'$ legaler Anfangs- und Endzustand eines (beliebig komplexen) S-beschränkt transparenten Ablaufs. Dann ist jedes anschließende Löschen einer Wurzel $\mathbb{A}' \overset{\forall x}{\longmapsto} \mathbb{A}''$ mit $x \in \sqrt{A'} - \sqrt{A}$ ebenfalls S-beschränkt transparent.*

Beweis. Das Löschen einer Wurzel x ist nur dann nicht transparent, wenn dadurch die Zelle x stirbt. Diese Zelle ist also vor dem Löschen nur über diese Wurzel erreichbar. Da $x \in \sqrt{A'}$, aber $x \notin \sqrt{A}$, gilt $x \notin A$ und damit $x \notin S$. □

Die lokalen Variablenbindungen einer Nebenrechnung können also entfernt werden, ohne die Transparenz der Hauptrechnung zu beeinträchtigen. Auch die Strategie, nach welcher tote Zellen zwischen Berechnungsschritten gelöscht werden, ist irrelevant, sofern tote Zellen nicht wieder lebendig werden.

Satz 3.32. *Das sichere Löschen einer Zelle kommutiert mit jeder anderen sicheren Operation r, sofern die Zelle vorher und nachher tot ist.*



Beweis. Sei $\mathbb{A}$ legal. Sei die Zelle x tot in $\mathbb{A}$ und $\mathbb{A}_r$, damit die Anwendung von $\mathsf{d}\,x$ sicher ist. Dann ist x ebenfalls tot in $\mathbb{A}_0$ und $\mathbb{A}_{r0}$. Sei x auch tot in $\mathbb{A}_{0r}$. Dann sind alle Zustände legal, und $\mathbb{A}_{0r}$ und $\mathbb{A}_{r0}$ bezeichnen dieselbe Coalgebra.

Fallunterscheidung über die oben definierten Operationen r ergibt, daß die Belegtheit oder der Inhalt invariant toter Zellen keinen Einfluß auf die Coalgebra des Nachzustandes hat. □

Allerdings kann das Löschen toter Zellen in dem hier beschriebenen Speichermodell nicht völlig beliebig mit Berechnungsschritten vermischt werden: Nach dem Erzeugen einer Zelle sind in der Regel Zuweisungsschritte erforderlich, bevor die Zelle lebensfähig ist. In einem legalen Ablauf ist also eine unvollständig initialisierte Zelle qua Definition tot. Falls eine solche Zelle in einem späteren Berechnungsschritt lebendig gemacht werden soll, darf sie offensichtlich nicht zwischendurch gelöscht werden.

Verfeinert man das Speichermodell mit einer zweiten Wurzelmenge, und verbietet man das Löschen von Zellen, die über diese zweite Wurzelmenge erreichbar sind, dann kann eine Berechnung so spezifiziert werden, daß der Ablauf an jeder beliebigen Stelle korrekt zur Speicherbereinigung unterbrochen werden kann. Da Strategien zur Speicherverwaltung aber nicht Gegenstand dieser Arbeit sind, wird das Problem in den späteren Kapiteln ignoriert.

3.4.1 Erzeugung und Initialisierung

Das Problem, daß Zellen zwischen ihrer Erzeugung und ihrer vollständigen Initialisierung in einem illegalen Zwischenzustand existieren, ist nicht neu. Beim Aufbau zyklischer Datenstrukturen ist es praktisch unvermeidbar. Anhand verschiedener Strategien zu seiner Lösung läßt sich der hier verfolgte strikt funktionale Ansatz in operationaler Hinsicht gut einordnen:

3.4.1.1 Objektorientierter Ansatz

Objektorientierte Sprachen versuchen, dem Problem durch die Einführung von *Konstruktoren* zu begegnen. Diese sind aber keine atomaren Operationen wie die algebraischen Konstruktoren, sondern lediglich Initialisierungsmethoden, deren Aufruf an die Erzeugung eines Objektes gebunden ist. Dem Aufrufer eines Konstruktors wird zugesichert, daß nach der Rückkehr sowohl die Erzeugung als auch die Initialisierung des Objektes abgeschlossen ist. Dieser pragmatische Ansatz ist aus zweierlei Gründen unbefriedigend:

1. Da objektorientierte Sprachen in der Praxis eine strikte Auswertungsstrategie besitzen, lassen sich mit Hilfe von Konstruktoren alleine keine zyklischen Strukturen erzeugen. Hier muß wieder auf das intransparente Verfahren der nachträglichen Zuweisung zurückgegriffen werden, wie es auch in primitiven imperativen Sprachen das Mittel der Wahl ist.

2. Konstruktoren dürfen, wie andere Methoden auch, geschachtelte Aufrufe enthalten. Damit ist es möglich, Methoden im Kontext eines unvollständig initialisierten Objektes aufzurufen. Auf dem Umweg über Vererbung und Überschreiben von Methoden kann dieser Effekt sogar per unsichtbarer Fernwirkung eintreten und auf bizarre Weise mit anderen Sprachmechanismen interagieren, wie das folgende Beispiel zeigen wird. Partiell wird dem in der Praxis dadurch begegnet, daß vor der Initialisierung durch den Konstruktor bereits eine Vorbelegung mit Standardwerten erfolgt, ohne daß das Problem aber grundsätzlich vermeidbar wäre.

Beispiel 3.33. Man betrachte folgendes, nicht offensichtlich pathologische Java-Programm:

```
class A {
  int y;
  A(int x) { y = f(x); }
  int f(int x) { return x; }
}

class B {
  int z = 2;
  A a = new A(3) { int f(int x) { return z; } };
}
```

Bei der Erzeugung eines `B`-Objektes stürzt das Programm ab! Das liegt daran, daß das freie Vorkommen von `z`, eigentlich `this.z`, im Rumpf der überschriebenen Methode `f` per λ-Lifting auf ein zusätzliches, unsichtbares Attribut der abgeleiteten anonymen Klasse abgebildet wird, welches die `this`-Referenz des `B`-Objektes kapselt. Dieses ist aber zum Zeitpunkt der Ausführung von `f` noch nicht initialisiert, und hat daher den Standardwert `null`, so daß der Zugriff auf das Attribut `z` fehlschlägt. □

3.4.1.2 Nichtstrikter Ansatz

In *lazy* Sprachen wird das Problem auf sehr elegante Weise umgangen: Eine zyklische Datenstruktur wird hier durch eine rekursive `let`-Bindung erzeugt. Also ist der Zyklus bereits im erzeugenden Code enthalten, und Compiler und Laufzeitsystem behalten die Kontrolle über die transparente Erzeugung des Ausdrucks, wobei zwar die Zyklenstruktur aufgebaut wird, aber im Allgemeinen keine Initialisierung stattfindet. Diese wird per *lazy* Reduktion bei Bedarf und auf der Stelle durchgeführt, ohne die Transparenz für den Benutzer zu kompromittieren.

Eine auf operationaler Ebene derart saubere Lösung ist anscheinend notwendigerweise mit nichtstrikter Semantik verbunden. Daraus wird sich in den folgenden Kapiteln eine Diskrepanz zwischen der theoretischen Erzeugung von Zyklen mittels primitiver Corekursion und deren Operationalisierung mittels Zuweisungen ergeben. Die dabei auftretenden, partiell initialisierten Objekte werden in Teil III bei der Spezifikation einer operationalen Semantik eine zentrale Rolle spielen.

3.5 Anhang: Algebraische Speicherzustände

Geht man von der hier beschriebenen coalgebraischen Sichtweise auf Speicherzellen aus, dann ist bis zu der traditionellen algebraischen Sichtweise ein erheblicher Abstraktionsweg zurückzulegen. Sei Σ ein Signaturfunktor. Sei $\mathbb{I}$ eine initiale Σ-Algebra und $\mathbb{F}$ eine finale Σ-Coalgebra. Sei $\mathbb{A}$ ein Σ-Speicherzustand.

1. Zunächst ordnet man jeder Zellenadresse $x \in A$ ihr finales Abbild zu:

$$x' = [\![\mathbb{A}]\!](x)$$

   Übrig bleibt die coalgebraische Struktur von $\mathbb{A}$ modulo Bisimilarität, also die finale Semantik des Speicherzustands. Information über die konkrete Repräsentation geht verloren.

2. Da $\mathbb{A}$ rational ist, gilt $[\![\mathbb{A}]\!] : \mathbb{A} \to \mathbb{F}_*$. Damit unterteilt sich die Adressmenge A in einen zyklenfreien Anteil A_0 und einen zyklischen Anteil A_∞, so daß:

$$[\![\mathbb{A}]\!] : A_0 \to F_0 \qquad\qquad [\![\mathbb{A}]\!] : A_\infty \to F_* \setminus F_0$$

3. Der zyklenfreie Anteil hat Urbilder in der initialen Algebra:

$$x'' = [\![\mathbb{I}^{-1}]\!]^{-1}(x')$$

4. Für den zyklischen Anteil gibt es, je nach Stil der algebraischen Semantik, verschiedene Deutungen:

   (a) Eine fundierte Semantik weist zyklische Objekte als ungültige Repräsentationen zurück. In einer strikten, pur funktionalen Sprache sind solche Objekte *a priori* ausgeschlossen. In einer Sprache mit Seiteneffekten können sie auftreten und führen üblicherweise zur Nichttermination rekursiver Funktionen.

   (b) Eine nichtfundierte Semantik ordnet zyklische Objekte genau wie unbeschränkte Berechnungen dem Reich des Unendlichen zu. Eine Unterscheidung der beiden Phänomene ist algebraisch nicht möglich.

   (c) Eine zweistufig konkrete und abstrakte Semantik wie in [41] kann bestimmten Formen zyklischer Strukturen durch geeignete Wahl einer speziellen Abstraktionsfunktion eine endliche algebraische Darstellung zuweisen.

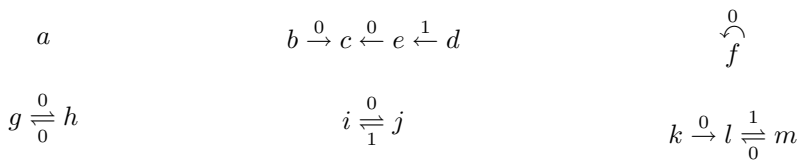
Beispiel 3.34. Sei $\Sigma = \mathcal{C}_1 + \mathcal{C}_2 \times \mathcal{I}$ die Signatur der Binärfolgen, wobei $2 = \{0, 1\}$. Wir schreiben für die Injektionen:

$$\mathsf{nil} = \iota_1(\star) \qquad\qquad \mathsf{cons} = \iota_2$$

Damit läßt sich nun ein direkt adressierter Speicherzustand $\mathbb{A} = (\sqrt{A}, \alpha)$ spezifizieren. Wir schreiben $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ für Zellenadressen. Neben den finalen und initalen Abstraktionen x', x'' betrachten wir die in [41] verwendete Abstraktion x''' zyklischer Listen, nämlich das längste Anfangsstück, bei dem keine Zelle

mehrmals traversiert wird.

x	$\alpha(x)$	$x' = [\![\mathbb{A}]\!](x)$	$x'' = [\![\mathbb{I}^{-1}]\!]^{-1}(x')$	x'''
a	nil	$[]$	$[]$	$[]$
b	cons(0, c)	$[0]$	$[0]$	$[0]$
c	nil	$[]$	$[]$	$[]$
d	cons(1, e)	$[1,0]$	$[1,0]$	$[1,0]$
e	cons(0, c)	$[0]$	$[0]$	$[0]$
f	cons(0, f)	$[0,\ldots$	$\bot$	$[0]$
g	cons(0, h)	$[0,\ldots$	$\bot$	$[0,0]$
h	cons(0, g)	$[0,\ldots$	$\bot$	$[0,0]$
i	cons(0, j)	$[0,1,\ldots$	$\bot$	$[0,1]$
j	cons(1, i)	$[1,0,\ldots$	$\bot$	$[1,0]$
k	cons(0, l)	$[0,1,\ldots$	$\bot$	$[0,1,0]$
l	cons(1, m)	$[1,0,\ldots$	$\bot$	$[1,0]$
m	cons(0, l)	$[0,1,\ldots$	$\bot$	$[0,1]$



Dabei gelten folgende Beobachtungen:

1. Im endlichen Fall stimmen alle drei Abstraktionen überein.
2. Graphische Isomorphie ist hinreichend für Identifikation in allen drei Abstraktionen.
3. Die Präfixabstraktion x''' unterscheidet einige Elemente, die von der finalen Abstraktion identifiziert werden (so $\mathsf{i}''' \neq \mathsf{k}'''$), aber nicht alle (so $\mathsf{g}''' = \mathsf{h}'''$). Der Grund ist, daß der pragmatische Begriff zyklischer Listen hinter x''' nicht der finale ist, sondern einen in der imperativen Programmierung relevanten Spezialfall darstellt, nämlich den stark zusammenhängenden Fall eines einzigen großen Zyklus, dessen Länge ebenfalls relevant ist. Nicht abstrahiert wird dagegen von Abwicklungen innerhalb des Zyklus oder als Präfix.

□

Die zusätzlichen Freiheitsgrade der finalen Interpretation im Vergleich zu solchen, die einen Teil der konkreten graphischen Struktur erhalten, werden in den folgenden Kapiteln noch eine erhebliche Rolle spielen. Die in Kapitel 4 zu entwickelnden Verfahren sind bis auf finale Semantik und damit relativ lose spezifiziert. Daraus ergeben sich weitreichende und für den praktischen Einsatz auch notwendige Optimierungspotentiale, die in Kapitel 6 ausführlich untersucht werden sollen.

Teil II

Algorithmische Theorie

You're caught in a vicious circle
surrounded by your so-called friends
You're caught in a vicious circle
and it looks like it will never end

You're caught in a vicious circle
You're caught in a vicious circle
You're caught in a vicious circle
You're caught in a vicious circle
surrounded by all of your friends

Lou Reed [52]

Kapitel 4

Berechnen zyklischer Funktionen durch primitive Corekursion

In diesem Kapitel soll die Berechnung primitiv corekursiver Funktionen als Übergang zwischen Speicherzuständen formal definiert und algorithmisch umgesetzt werden.

1. Zuerst muß ein Formalismus geschaffen werden, in dem die erzeugende Coalgebra $\mathbb{C}$ eines Anamorphismus $[\![\mathbb{C}]\!]$ als Operation auf bestimmten oder beliebigen Speicherzuständen beschrieben werden kann.

2. Beim corekursiven Abstieg ist es außerdem möglich, daß ein Zyklus entsteht, da die transitive Erreichbarkeitsrelation anders als im algebraisch rekursiven Fall nicht zyklenfrei sein muß. Solche Zyklen müssen erkannt und effektiv behandelt werden, damit die Berechnung terminieren kann.

Beide Phasen werden von durchgängigen Beispielen begleitet. Parallel werden Haskell-Programme entwickelt, die erstens streng formale Spezifikationen und zweitens ausführbare Implementierungen darstellen. Diese haben allerdings reinen Referenzcharakter. Ein Rahmen für effiziente Implementierungen wird Gegenstand der restlichen Arbeit ab dem übernächsten Kapitel sein.

4.1 Primitive Corekursion und Speicherzustände

4.1.1 Interpretation von Speicherzuständen

Bei der Implementierung einer primitiv corekursiven Funktion muß eine Repräsentation der Bildmenge, welche Teil einer finalen Coalgebra ist, gewählt werden. Aus naheliegenden Gründen betrachten wir Coterme, deren aufspannende Coalgebra ein Speicherzustand ist. Dabei treten die für Coterme charakteristischen Freiheitsgrade in der Repräsentation auf. Die Implementierung einer primitiv corekursiven Funktion ist also lediglich bis auf Äquivalenz der berechneten Coterme spezifiziert; die Wahl der konkreten Cotermdarstellung soll jedoch transparent sein.

Damit ergibt sich ein fundamentales Problem, falls die erzeugten Daten anschließend weiterverarbeitet werden sollen: Funktionen dürfen nicht die Cotermstruktur ihrer Eingaben, sondern lediglich deren finale Semantik berücksichtigen.[1] Wir gehen dazu in mehreren Näherungsschritten vor:

1. Zuerst werden *Interpretationen* als allgemeine Klasse von Coalgebren über Speicherzuständen definiert, die geeignet ist, beliebige primitiv corekursive Funktionen über gespeicherten Daten zu erzeugen.

[1] In die Metasprache der Programmiersprache C übersetzt heißt das: Auf Zeigern ist lediglich Dereferenzierung mittels $*$, nicht aber arithmetische oder relationale Operationen gestattet.

2. Diese Klasse wird eingeschränkt auf *sekundäre* Interpretationen, die sich in einen Speicherzustands-spezifischen und einen wiederverwendbaren Anteil faktorisieren lassen. Damit wird es möglich, primitiv corekursive Funktionsdefinitionen auf beliebige Startzustände anzuwenden.

3. Eine a priori unabhängige Einschränkung ist die auf *abstrakte* Interpretationen, welche Bisimilarität erhalten. Damit ist gewährleistet, daß die Freiheitsgrade der Datenrepräsentation keine beobachtbaren Auswirkungen haben, und somit die Implementierung, obwohl als nichtdeterministische Konstruktion formuliert, tatsächlich bis auf Bisimilarität eine Funktion darstellt. Es zeigt sich, daß abstrakte Interpretationen außerdem stets auch sekundär sind.

Definition 4.1 (Interpretation)**.** Seien Σ, In, Ex Signaturfunktoren. Sei $\mathbb{A} = \big(\sqrt{A}, \alpha\big)$ ein legaler Speicherzustand über Σ. Eine Ex-Coalgebra $\mathbb{C} = (C, \gamma)$ mit $C \subseteq In(A)$ heißt $(In \to Ex)$-*Interpretation* über $\mathbb{A}$. Wir schreiben auch $\mathbb{C} : In \xrightarrow{\mathbb{A}} Ex$.

$$\Sigma(A) \xleftarrow{\alpha} A \xmapsto{In} In(A) \supseteq C \xrightarrow{\gamma} Ex(C) \qquad \square$$

Eine Interpretation spielt die Rolle einer syntaktisch rekursiven Funktionsdefinition:

1. Ihre Eingaben sind Datensätze zur Signatur In über den Zellenadressen A eines Speicherzustandes. Diese können als Funktionsaufrufe verstanden werden, in denen eventuell Referenzen auf gespeicherte Daten vorkommen.

2. Diese werden auf Datensätze zur Signatur Ex abgebildet, die als Funktionsrümpfe verstanden werden können. Anstelle von Zellenadressen aus A enthalten diese allerdings als referentiellen Anteil wiederum Datensätze aus $In(A)$, welche als rekursive Aufrufe gedeutet werden sollen. Die Abbildung $\bar{\mathfrak{s}}(Ex) \circ \gamma$ definiert also die *Rekurrenzrelation*.

Beispiel 4.2 (Interpretationen). Die Funktoren In, Ex spielen die Rolle der *Ein-* bzw. *Ausgabesyntax* einer Interpretation.

1. Einstellige Interpretationsabbildungen werden durch $In = \mathcal{I}$ charakterisiert.

   Sei $\Sigma = \mathcal{C}_{\mathcal{N}}$, enthalten also alle Speicherzellen natürliche Zahlen. Sei ferner $\mathcal{A} = \mathcal{N}$, so daß Zellen ebenfalls über natürliche Zahlen adressiert werden. Sei $\mathbb{A} = (A, \alpha)$ ein Speicherzustand über Σ und $\mathcal{A}$. Sei $Ex = \mathcal{C}_1 + \mathcal{I} \times \mathcal{C}_{\mathcal{N}} \times \mathcal{I}$ die Signatur der binären $\mathcal{N}$-gewichteten Bäume. Dann definiert ein Adressintervall $I = \{i \dots j\}$ ein Array natürlicher Zahlen im Speicher, und eine $(In \to Ex)$-Interpretation $\mathbb{BB}_I = (I, \beta_I)$ als endlicher Binärbaum:

   $$\beta_I(a) = \begin{cases} \iota_2\big((a', \alpha(a), a' + 1)\big) & a \in I \\ \iota_1(\star) & a \notin I \end{cases} \qquad\qquad a' = 2a - i + 1$$

2. n-stellige Prädikate werden durch $In = \mathcal{I}^n$ und $Ex = \mathcal{C}_{\{w,f\}}$ charakterisiert.

   Sei Σ ein Signaturfunktor und $\mathbb{A} = (A, \alpha)$ ein Σ-Speicherzustand. Sei $n = 2$. Dann ist die *flache Gleichheit* $\mathbb{EQ} = (A \times A, \varepsilon)$ wie folgt definiert:

   $$\varepsilon\big((a_1, a_2)\big) = \begin{cases} w & \alpha(a_1) = \alpha(a_2) \\ f & \alpha(a_1) \neq \alpha(a_2) \end{cases}$$

   Diese Interpretation hat über die Beispielfunktion hinaus nur wenig Bedeutung. Für die ernsthafte Behandlung von Logik über zyklischen Datenstrukturen sei auf das folgende Kapitel verwiesen.

3. Von besonderem Interesse im Zusammenhang mit rekursiven Funktionsdefinitionen sind Interpretationen mit $Ex = \Sigma$. Sei $\mathbb{C} = (C, \gamma)$ eine solche $(In \to \Sigma)$-Interpretation. Dann gilt $\operatorname{codom} \gamma \subseteq \Sigma(In(A))$. Ein Ausgabeausdruck besteht also aus einem äußeren, lokalen Σ-Datensatz, der an den

Argumentpositionen In-Ausdrücke enthält, die wiederum als Eingaben derselben Interpretation dienen können.

Sei $\Sigma = \mathcal{C}_1 + \mathcal{I}$ die PEANO-Signatur der natürlichen Zahlen. Wir schreiben z für $\iota_1(\star)$ und $\mathsf{s}(x)$ für $\iota_2(x)$. Sei $\mathbb{A} = (A, \alpha)$ ein Σ-Speicherzustand. Sei $In = \mathcal{I} \times \mathcal{I}$. Dann ist der Rekursionsschritt der Addition zweier Zahlen als $(In \to \Sigma)$-Interpretation $\mathbb{S} = (A \times A, \sigma)$ wie folgt definiert:

$$\sigma\big((a_1, a_2)\big) = \begin{cases} \mathsf{z} & \alpha(a_1) = \mathsf{z} \wedge \alpha(a_2) = \mathsf{z} \\ \mathsf{s}\big((\mathsf{z}, a_2')\big) & \alpha(a_1) = \mathsf{z} \wedge \alpha(a_2) = \mathsf{s}(a_2') \\ \mathsf{s}\big((a_1', a_2)\big) & \alpha(a_1) = \mathsf{s}(a_1') \end{cases}$$

4. Sei $\Sigma = (\mathcal{C}_1 + \mathcal{I}) + (\mathcal{C}_1 + \mathcal{I} \times \mathcal{I})$ die gemeinsame Signatur der natürlichen Zahlen und der Binärbäume. Wir fassen die Bäume als ungetypte Listen wie in LISP auf, und schreiben die Konstanten als z und nil, die Konstruktoren als s und cons. Sei $\mathbb{A} = (A, \alpha)$ ein Σ-Speicherzustand. Sei $In = \mathcal{I}$ und $Ex = \Sigma$. Dann ist der Rekursionsschritt der Zählung der Elemente von Listen als Interpretation $\mathbb{L} = (L, \lambda)$ wie folgt definiert:

   (a) L umfaßt die Instanzen des finalen Datentyps über nil und cons in $\mathbb{A}$, also die größte Lösung der Urbildgleichung
   $$L = \overline{\mathcal{P}}(\alpha)\big(\{\mathsf{nil}\} \cup \mathcal{P}(\mathsf{cons})(A \times L)\big)$$

   (b) Die Operation λ ersetzt gegenüber α die Injektionen der Listen durch die der Zahlen:
   $$\lambda(a) = \begin{cases} \mathsf{z} & \alpha(a) = \mathsf{nil} \\ \mathsf{pred}(a_2) & \alpha(a) = \mathsf{cons}(a_1, a_2) \end{cases}$$

□

Definition 4.3 (Sekundäre Interpretation)**.** Eine $(In \to Ex)$-Interpretation $\mathbb{C} = (C, \gamma)$ über einem Σ-Speicherzustand $\mathbb{A} = (\sqrt{A}, \alpha)$ heißt *sekundär*, wenn eine Abbildung γ_0 existiert, so daß $\gamma = \gamma_0 \circ In(\alpha)$.

Das Paar $\mathbb{C}_0 = (C, \gamma_0)$ heißt *Rumpf* der Interpretation. Dieser definiert eine partielle Abbildung von Speicherzuständen auf Interpretationen: Sei $\mathbb{A}' = \big(\sqrt{A'}, \alpha'\big)$ ein beliebiger Σ-Speicherzustand. Wir schreiben $\mathbb{C}_0(\mathbb{A}') = \big(C, \gamma_0 \circ In(\alpha')\big)$, falls diese Struktur eine Interpration über $\mathbb{A}'$ ist. □

Lemma 4.4. *Eine mittels Pattern definierte Interpretation ist sekundär.*

Beispiel 4.5 (Sekundäre Interpretationen). Hier die Untersuchung der bereits bekannten Beispielinterpretationen auf Sekundarität:

1. Die Binärbaum-Interpretation $\mathbb{BB}_I$ ist im Allgemeinen nicht sekundär: Die Operation β_I ist injektiv. Falls aber α nicht auf ganz I injektiv ist, dann auch nicht $In(\alpha)$, und es existiert kein passendes β_0, so daß $\beta_0 \circ In(\alpha) = \beta_I$.

2. Die flache Gleichheit $\mathbb{EQ}$ ist sekundär, und es gilt erwartungsgemäß:
   $$\varepsilon_0\big((e, e')\big) = \begin{cases} w & e = e' \\ f & e \neq e' \end{cases}$$

3. Der Additionsschritt $\mathbb{S}$ ist sekundär. Der Rumpf hat bereits mehr Ähnlichkeit mit einem funktionalen Programm im *pattern matching*-Stil:
   $$\begin{aligned} \sigma_0(\mathsf{z}, \mathsf{z}) &= \mathsf{z} \\ \sigma_0\big((\mathsf{z}, \mathsf{s}(x))\big) &= \mathsf{s}\big((\mathsf{z}, x)\big) \\ \sigma_0\big((\mathsf{s}(x), y)\big) &= \mathsf{s}\big((x, y)\big) \end{aligned}$$

4. Der Zählschritt $\mathbb{L}$ ist sekundär. Für den Rumpf gilt:

$$\lambda_0(\mathsf{nil}) = \mathsf{z}$$
$$\lambda_0\big(\mathsf{cons}(a_1, a_2)\big) = \mathsf{s}(a_2)$$

□

Der Rumpf einer sekundären Interpretation kann auch als Abstraktion über einem referentiell transparenten Ablauf verstanden werden:

Lemma 4.6 (Invarianz)**.** *Seien $\mathbb{A}, \mathbb{A}'$ Speicherzustände. Sei $\mathbb{C}_0 = (C, \gamma_0)$ der Rumpf einer sekundären Interpretation. Wenn der Übergang $\mathbb{A} \mapsto \mathbb{A}'$ referentiell transparent und $\mathbb{C}_0(\mathbb{A})$ definiert ist, dann gilt $\mathbb{C}_0(\mathbb{A}) = \mathbb{C}_0(\mathbb{A}')$.*

Beweis. Folgt unmittelbar aus Definition 3.7. □

Kommen wir nun zu unserem eigentlichen Anliegen, der Unabhängigkeit von der Repräsentation der Coterme.

Definition 4.7 (Abstrakte Interpretation)**.** Seien Σ, In, Ex Signaturfunktoren. Sei $\mathbb{A} = \big(\sqrt{A}, \alpha\big)$ ein legaler Σ-Speicherzustand. Sei $\mathbb{F} = (F, \phi)$ eine finale Σ-Coalgebra. Eine $(In \to Ex)$-Interpretation $\mathbb{C}$ über $\mathbb{A}$ heißt *abstrakt*, falls sie verträglich mit der finalen Semantik von $\mathbb{A}$ ist, falls also eine Ex-Coalgebra $\overline{\mathbb{C}} = (\overline{C}, \overline{\gamma})$ mit $\overline{C} \subseteq In(F)$ existiert, so daß $In\big([\![\mathbb{A}]\!]\big) : C \to \overline{C}$ ein Ex-Homomorphismus ist (unten im Diagramm):

$$\begin{array}{ccc}
\Sigma(A) & \xrightarrow{\Sigma\left(\{\mathbb{A}\}\right)} & \Sigma(F) \\
{\scriptstyle\alpha}\uparrow & & \uparrow{\scriptstyle\phi} \\
A & \xrightarrow[\{\mathbb{A}\}]{} & F \\
{\scriptstyle In}\downarrow & & \downarrow{\scriptstyle In} \\
In(A) & \xrightarrow[In\left(\{\mathbb{A}\}\right)]{} & In(F) \\
{\scriptstyle\subseteq}\hat{} & & \hat{}{\scriptstyle\subseteq} \\
C & \xrightarrow[In\left(\{\mathbb{A}\}\right)]{} & \overline{C} \\
{\scriptstyle\gamma}\downarrow & & \downarrow{\scriptstyle\overline{\gamma}} \\
Ex(C) & \xrightarrow[Ex\left(In\left(\{\mathbb{A}\}\right)\right)]{} & Ex(\overline{C})
\end{array}$$

Dabei heißt $\overline{\mathbb{C}}$ eine *finale Spezifikation* von $\mathbb{C}$. □

Die Motivation für die Einführung abstrakter Interpretationen läßt sich auf folgendes Dilemma reduzieren: Konstruktion des Anamorphismus steht nur auf der finalen Coalgebra (rechts im Diagramm), effektive Repräsentation von Daten nur auf nicht-finalen Coalgebren (links im Diagramm) zur Verfügung. Die coalgebraische Spezifikation eines entsprechenden Algorithmus muß also beide Ebenen zueinander in kohärente Beziehung setzen. Die Definition wirkt folgendermaßen:

1. Die Trägermenge C einer abstrakten Interpretation ist wie gehabt eine Menge von Datensätzen zur Signatur In über Zellenadressen aus A, der referentielle Anteil eines Datensatzes x enthält also physikalisch gespeicherte, nichtfinale Coterme.

2. Die Trägermenge $\overline{C}$ der finalen Spezifikation ist eine Menge von Datensätzen zur selben Signatur, aber mit Elementen der finalen Coalgebra im referentiellen Anteil. Diese Menge enthält mindestens das Bild von C unter der Abbildung $In\big([\![\mathbb{A}]\!]\big)$, welche die nichtfinalen Coterme aus dem Speicher jeweils durch ihre finale Semantik ersetzt.

3. Jedem Eingabedatensatz $\bar{x} = In\big([\![\mathbb{A}]\!]\big)(x)$ in diesem Bild ist eindeutig ein Ausgabedatensatz zugeordnet. Genauer gesagt derselbe, der auch entsteht, wenn man alle Speicheradressen in $\gamma(x)$, in geschachtelten Datensätzen zu den Signaturen Ex und In, durch ihre finale Semantik ersetzt. Die Operation γ muß also semantisch äquivalente Speicherzellen gleich behandeln.

Satz 4.8 (Eindeutigkeit)**.** *Die minimale finale Spezifikation einer abstrakten Interpretation ist eindeutig bestimmt.*

Beweis. Wähle $\overline{C} = \mathrm{codom}_{C \to In(F)}\, In\big([\![\mathbb{A}]\!]\big)$. Dies ist offensichtlich die eindeutige kleinste Trägermenge. Die Eindeutigkeit der Operation folgt durch Widerspruch: Sei $\overline{\gamma} : \overline{C} \to Ex(\overline{C})$ eine Lösung des obigen Diagramms. Sei $\widetilde{\gamma} : \overline{C} \to Ex(\overline{C})$ eine beliebige Operation mit $\overline{\gamma} \neq \widetilde{\gamma}$. Sei nun $x \in \overline{C}$ eine Stelle, so daß $\overline{\gamma}(x) \neq \widetilde{\gamma}(x)$. Aus der Minimalität von $\overline{C}$ folgt, daß ein $w \in C$ existiert mit $In\big([\![\mathbb{A}]\!]\big)(w) = x$. Also gilt auch:

$$\overline{\gamma} \circ In\big([\![\mathbb{A}]\!]\big) \neq \widetilde{\gamma} \circ In\big([\![\mathbb{A}]\!]\big)$$

Damit ist $\widetilde{\gamma}$ keine Lösung. □

Satz 4.9. *Jede abstrakte Interpretation ist sekundär.*

Beweis. Sei $\mathbb{C} = (C, \gamma)$ eine abstrakte $(In \to Ex)$-Interpretation über einem Σ-Speicherzustand $\mathbb{A} = \big(\sqrt{A}, \alpha\big)$. Sei $\overline{\mathbb{C}} = (\overline{C}, \overline{\gamma})$ die minimale finale Spezifikation von $\mathbb{C}$. Zu zeigen ist die Existenz einer Abbildung γ_0, so daß:

$$\begin{array}{ccc} C & \xrightarrow{In\left([\![\mathbb{A}]\!]\right)} & \overline{C} \\ {\scriptstyle In(\alpha)}\downarrow & & \downarrow{\scriptstyle \overline{\gamma}} \\ Ex(C) & \xrightarrow[Ex\left(In\left([\![\mathbb{A}]\!]\right)\right)]{} & Ex(\overline{C}) \end{array} \qquad \gamma_0 : C \to Ex(C)$$

Eine solche Abbildung γ_0 existiert, falls gilt:

1. $\mathrm{Ker}\ In(\alpha) \subseteq \mathrm{Ker}\ \overline{\gamma} \circ In\big([\![\mathbb{A}]\!]\big)$. Folgt aus Lemmata 2.70 und 2.80.
2. $\mathrm{codom}\ \overline{\gamma} \circ In\big([\![\mathbb{A}]\!]\big) \subseteq \mathrm{codom}\ Ex\big(In\big([\![\mathbb{A}]\!]\big)\big)$. Aus der Minimalität von $\overline{C}$ folgt mit Lemma 2.82 $\mathrm{codom}\ Ex\big(In\big([\![\mathbb{A}]\!]\big)\big) = Ex(\overline{C})$. Offensichtlich gilt $\mathrm{codom}\ \overline{\gamma} \circ In\big([\![\mathbb{A}]\!]\big) \subseteq \mathrm{codom}\ \overline{\gamma} \subseteq Ex(\overline{C})$.

□

Beispiel 4.10 (Abstrakte Interpretationen). Hier die Untersuchung der bereits bekannten Interpretationen.

1. Die Binärbaum-Interpretation $\mathbb{BB}_I$ ist nicht sekundär, und daher auch nicht abstrakt.
2. Die flache Gleichheit $\mathbb{EQ}$ ist im Allgemeinen nicht abstrakt: Sei $\Sigma = \mathcal{I}$. Die einelementige Struktur $\mathbb{F} = (\{\star\}, !)$ ist eine finale Σ-Coalgebra. Sei $\mathbb{A} = (\{x, y\}, \alpha)$ zweielementig mit $\alpha(x) = x$ und $\alpha(y) = y$. Dann gilt:

$$\varepsilon\big((x,x)\big) = w \qquad\qquad \varepsilon\big((x,y)\big) = f$$

Einsetzen der Paares (x, x) in die Abstraktionsgleichung ergibt:

$$\begin{aligned} \overline{\varepsilon}\big(In\big([\![\mathbb{A}]\!]\big)\big((x,x)\big)\big) &= Ex\big(In\big([\![\mathbb{A}]\!]\big)\big)\big(\varepsilon\big((x,x)\big)\big) \\ \overline{\varepsilon}\big(([\![\mathbb{A}]\!](x), [\![\mathbb{A}]\!](x))\big) &= \mathcal{C}_{\{w,f\}}\big(In\big([\![\mathbb{A}]\!]\big)\big)\big(\varepsilon\big((x,x)\big)\big) \\ \overline{\varepsilon}\big((\star,\star)\big) &= \mathrm{id}\big(\varepsilon\big((x,x)\big)\big) \\ &= \varepsilon\big((x,x)\big) \\ &= w \end{aligned}$$

Analog ergibt sich $\overline{\varepsilon}\big((\star,\star)\big) = \varepsilon\big((x,y)\big) = f$. Also existiert keine finale Spezifikation.

3. Der Additionsschritt $\mathbb{S}$ ist abstrakt: Sei $\bar{\mathbb{N}}$ die aus Beispiel 2.102 bekannte finale Σ-Coalgebra der erweiterten natürlichen Zahlen. Die Σ-Coalgebra $\overline{\mathbb{S}} = (\bar{\mathcal{N}} \times \bar{\mathcal{N}}, \overline{\sigma})$ ist finale Spezifikation, wobei gilt:

$$\begin{aligned} \overline{\sigma}\big((0,0)\big) &= \mathsf{z} \\ \overline{\sigma}\big((0,n)\big) &= \mathsf{s}\big((0, \mathit{pred}(n))\big) && n > 0 \\ \overline{\sigma}\big((n,m)\big) &= \mathsf{s}\big((\mathit{pred}(m), n)\big) && m, n > 0 \end{aligned}$$

   Damit gelten folgende Gesetze, die plausibel machen, daß es sich tatsächlich um Addition handelt, für alle $m, n \in \bar{\mathcal{N}}$:

$$\begin{aligned} \overline{\sigma}\big((m,n)\big) = \mathsf{z} \quad &\Longrightarrow \quad m + n = 0 \\ \overline{\sigma}\big((m,n)\big) = \mathsf{s}\big((m',n')\big) \quad &\Longrightarrow \quad m + n = 1 + m' + n' \end{aligned}$$

4. Der Zählschritt $\mathbb{L}$ ist abstrakt: Die Operation λ hat als Urbildbereich den finalen Datentyp über nil und cons. Die Struktur $\big(D^* \cup D^\omega, \langle \mathit{head}, \mathit{tail} \rangle\big)$ der endlichen und unendlichen Listen über D ist eine finale Coalgebra dieser Teilsignatur. Die Σ-Coalgebra $\overline{\mathbb{L}} = (D^* \cup D^\omega, \overline{\lambda})$ ist finale Spezifikation, wobei gilt:

$$\begin{aligned} \overline{\lambda}([]) &= \mathsf{z} \\ \overline{\lambda}(d : \vec{x}) &= \mathsf{s}(\vec{x}) \end{aligned}$$

Im nächsten Abschnitt werden wir die finale Spezifikation von abstrakten Interpretationen zur Erzeugung von Anamorphismen nutzen.

4.1.2 Abstrakter Algorithmus

Ein allgemeines Verfahren zur punktweisen Berechnung primitiv corekursiver Funktionen läßt sich mit Hilfe abstrakter Interpretationen als Konstruktion mit folgenden Eigenschaften charakterisieren:

1. Gegeben ist ein Eingabeausdruck x zur Syntax In über einem endlichen Anfangszustand $\mathbb{A}$ und eine abstrakte $(In \to \Sigma)$-Interpretation $\mathbb{C} = (C, \gamma)$ über $\mathbb{A}$ mit $x \in C$.

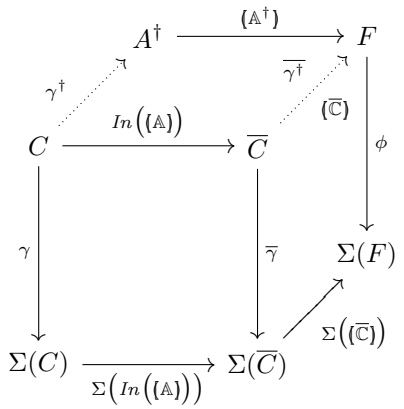
2. Gesucht ist ein Ausgabeelement x' und ein referentiell transparenter Übergang $\mathbb{A} \mapsto \mathbb{A}'_x$. Diese ergeben einen Ausgabecoterm $\mathbb{A}'_x \bullet x'$, so daß der Anamorphismus der finalen Spezifikation von $\mathbb{C}$ zwischen den finalen Semantiken von Ein- und Ausgabe übersetzt:

$$\begin{array}{ccc} \{x\} & \dashrightarrow & \{x'\} \\ {\scriptstyle In(\{\mathbb{A}\})}\downarrow & & \downarrow{\scriptstyle \{\mathbb{A}'_x\}} \\ \overline{C} & \xrightarrow[\{\overline{\mathbb{C}}\}]{} & F \end{array}$$

Die Verallgemeinerung auf alle $x \in C$ kann als virtueller Speicherzustand $\mathbb{A}^\dagger = (A^\dagger, \alpha^\dagger)$ mit $\mathbb{A}^\dagger = \bigcup_{x \in C} \mathbb{A}'_x$ vorgestellt werden, welcher den gesamten Wertebereich der Konstruktion abdeckt, indem quasi alle Elemente simultan konstruiert werden. Die Konstruktion selbst stellt auf ihrem Endzustand eine abstrakte $(In \to \mathcal{I})$-Interpretation $\mathbb{C}^\dagger$ dar. Die gewünschte Semantik ist durch die finale Spezifikation $\overline{\mathbb{C}^\dagger} = (\overline{C^\dagger}, \overline{\gamma^\dagger})$ gegeben, deren Operation der Anamorphismus der finalen Spezifikation der Eingabeinterpretation sein soll:

$$\overline{\gamma^\dagger} = [\![\overline{\mathbb{C}}]\!]$$

Der Zusammenhang ist in folgendem Diagramm dargestellt, wobei vorne die Abstraktionseigenschaft von $\mathbb{C}$ konstruiert ist, oben die Verallgemeinerung des vorherigen Diagramms auf $\mathbb{C}^\dagger$ und rechts der gewünschte Anamorphismus $[\![\overline{\mathbb{C}}]\!]$:



Durch Abstraktion von $\mathbb{A}$ entsteht die eigentliche, wiederverwendbare Implementierung. Dies läßt sich dadurch erreichen, daß der Rumpf $\gamma_0^\dagger$ der (sekundären) Ausgabeinterpretation $\mathbb{C}^\dagger$ alleine aus dem Rumpf γ_0 der Eingabeinterpretation $\mathbb{C}$ abgeleitet wird.

Lemma 4.6 ermöglicht die Implementierung von Interpretationen auf Basis echter Speicherzugriffe: Applikationen der Interpretation $\mathbb{C}$ können zu einem beliebigen Zeitpunkt einer referentiell transparenten Berechnung ausgewertet werden. Es ist nicht nötig, den Anfangszustand der Berechnung, auf welchem die Interpretation eigentlich definiert ist, „einzufrieren“.

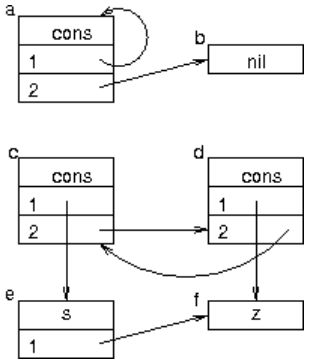
Beispiel 4.11 (Corekursive Berechnung). Als durchgängiges Beispiel für alle algorithmischen Entwicklungsstadien verwenden wir die bereits bekannte Interpretation $\mathbb{L} = (L, \lambda)$, welche die Elemente einer Liste zählt. Sei $\Sigma = (\mathcal{C}_1 + \mathcal{I}) + (\mathcal{C}_1 + \mathcal{I} \times \mathcal{I})$. Sei $In = \mathcal{I}$ und $Ex = \Sigma$. Sei außerdem $\mathcal{A} \supset \{\mathsf{a}, \ldots, \mathsf{z}\}$ und

$\mathcal{B} \supset \{\mathsf{a}_i, \ldots, \mathsf{z}_i \mid i \in \mathcal{N}\}$. Wir schreiben also Zellenadressen als Kleinbuchstaben und Variablenadressen als indizierte Kleinbuchstaben.

Außerdem seien im Initialzustand $\mathbb{A} = (A, \alpha_1, \alpha_2)$ einige Daten bereits gegeben: Sei $A = \{\mathsf{a}, \ldots, \mathsf{f}\}$, wobei

$$\begin{array}{lll}
\alpha_1(\mathsf{a}) = \mathsf{cons}(\mathsf{a}_1, \mathsf{a}_2) & \alpha_2(\mathsf{a}_1) = \mathsf{a} & \alpha_2(\mathsf{a}_2) = \mathsf{b} \\
\alpha_1(\mathsf{b}) = \mathsf{nil} & & \\
\alpha_1(\mathsf{c}) = \mathsf{cons}(\mathsf{c}_1, \mathsf{c}_2) & \alpha_2(\mathsf{c}_1) = \mathsf{e} & \alpha_2(\mathsf{c}_2) = \mathsf{d} \\
\alpha_1(\mathsf{d}) = \mathsf{cons}(\mathsf{d}_1, \mathsf{d}_2) & \alpha_2(\mathsf{d}_1) = \mathsf{f} & \alpha_2(\mathsf{d}_2) = \mathsf{c} \\
\alpha_1(\mathsf{e}) = \mathsf{s}(\mathsf{e}_1) & \alpha_2(\mathsf{e}_1) = \mathsf{f} & \\
\alpha_1(\mathsf{f}) = \mathsf{z} & &
\end{array}$$

Somit repräsentiert



1. a eine Liste, die sich selbst als einziges Element enthält,
2. b eine leere Liste,
3. c eine unendliche Liste $101010\ldots$,
4. d eine unendliche Liste $010101\ldots$,
5. e eine 1,
6. f eine 0.

Dann gilt:

$$\begin{array}{lll}
L = \{\mathsf{a}, \ldots, \mathsf{d}\} & \lambda(\mathsf{a}) = \mathsf{s}(\mathsf{b}) & \lambda(\mathsf{b}) = \mathsf{z} \\
& \lambda(\mathsf{c}) = \mathsf{s}(\mathsf{d}) & \lambda(\mathsf{d}) = \mathsf{s}(\mathsf{c})
\end{array}$$

Alle Kandidaten-Algorithmen werden im folgenden auf die Eingaben a und c angewendet. □

```
module Ana where
import Signature
import Heap
type Interp σ ξ ω     = ξ Cell → HeapOp σ (ω (ξ Cell))
type Function σ ξ     = Field → ξ Cell → HeapOp σ ()
type Ana              = (Signature σ, Signature ξ) ⇒ Interp σ ξ σ → Function σ ξ
```

Programm 4.1: Typdefinitionen für Berechnungsalgorithmen

4.1.3 Grundlagen der Spezifikation

Die ausführbare Spezifikation *Ana* der implementierten Corekursion baut auf den Spezifikationsmodulen *Signature* und *Heap* auf.

Interpretationen und Funktionen als Operationen auf Speicherzuständen werden durch folgende Typen spezifiziert:

1. Der Typkonstruktor *Interp* beschreibt sekundäre $(In \to Ex)$-Interpretationen über Σ-Speicherzuständen als Zustandsmonaden.

2. Der Typkonstruktor *Function* beschreibt *In*-Funktionen als speicherverändernde Prozeduren, welche mit einer Variable instanziiert werden, der das Ergebnis zuzuweisen ist.

3. Der Typ *Ana* schließlich beschreibt Konstruktionen, die einer $(In \to \Sigma)$-Interpretation eine *In*-Funktion zuordnen.

Anders als bisher beschrieben, werden Interpretationen hier selbst als potentielle Speicherzustandsübergänge modelliert. Sie können also den Speicherinhalt nicht nur beobachten, sondern auch verändern. Dabei wird referentielle Transparenz vorausgesetzt. Dann kann eine Interpretation mit Seiteneffekten wie folgt auf eine ohne Seiteneffekte zurückgeführt werden:

Sei $\mathbb{A}_0 \mapsto \mathbb{A}_1 \mapsto \cdots$ ein referentiell transparenter Ablauf. Bezeichne $\mathbb{A}_i \xmapsto{\mathrm{i}\, y=\gamma(x)} \mathbb{A}_{i+1}$ ein Anwendungsschritt der sekundären Interpretation $\mathbb{C} = (C, \gamma)$ mit $\gamma = \gamma_0 \circ In(\alpha_i)$. Dann gilt nach Lemma 4.6 auch $y = \gamma_0 \circ In(\alpha_j)$ für alle $j > i$.

Wenn also durch die Interpretation neue lebende Speicherzellen erzeugt werden, was der einzig mögliche Seiteneffekt einer referentiell transparenten Operation ist, dann können diese Zellen dem Anfangszustand hinzugefügt werden, um eine äquivalente, effektfreie Interpretation zu erhalten. Dabei ist nur zu beachten, daß bei unendlichen Abläufen aus einem endlichen Anfangszustand durch Hinzufügen der im Seiteneffekt entstandenen Zellen ein unendlicher werden kann. Für eine konkrete Implementierung, die endliche Speicherzustände voraussetzt, ist das Ignorieren von Seiteneffekten also nur bei Termination zulässig. Da wir ohnehin eine strikte Implementierung im Sinn haben, ist dies gewährleistet.

```
gen              :: Signature σ ⇒ Field → σ α → HeapOp σ (ITrans Field α)
gen v s          = do
                       (y, f) ← newCell s
                       setField v y
                       return f
genInterp c v x = gen v =≪ c x
```

Programm 4.2: Algorithmischer Grundschritt (coalgebraisch)

4.1.4 Algorithmischer Grundschritt

Die Grundoperation aller Implementierungen von Corekursion verhält sich erwartungsgemäß dual zu den *Konstruktoren* der Algebra. Sei A eine Wertemenge. Gegeben eine Variable $v \in \mathcal{B}$ und ein Zelleninhalts-Datensatz $s \in \Sigma(A)$, wird eine Zelle y mit zu s kompatiblem Inhalt t erzeugt, deren referentieller Anteil aber zunächst uninitialisiert bleibt. Der Datensatz s ist per Konstruktion initial. Die Adresse y wird v zugewiesen. Als Nebenprodukt fällt die Individualtransformation $f = t \Rightarrow s$ ab. Wir schreiben:

$$\mathbb{A} \xrightarrow{\mathsf{Gen}(v,s)\mapsto f} \mathbb{A}' \iff \mathbb{A} \xrightarrow{\mathsf{n}\, y:=t; v:=y} \mathbb{A} \wedge f = (t \Rightarrow s)$$

In der speziellen Anwendung auf eine $(In \to \Sigma)$-Interpretation $\mathbb{C} = (C, \gamma)$ ist statt des Datensatzes s ein Eingabeterm $x \in C \subseteq In(\mathcal{A})$ gegeben, und es gilt $A = C$ und $s = \gamma(x)$. Programm 4.2 zeigt die Implementierung. Der corekursive Abschluß wird später in der geschachtelten Anwendung auf die Elemente von $\mathrm{graph}\, f \subseteq \mathcal{B} \times \Sigma(A)$ bestehen. Die im vorigen Abschnitt erwähnte Interpretation mit Seiteneffekten ist in die monadische Hilfsfunktion *genInterp* gekapselt.

Zum Vergleich: Eine algebraischer Konstruktor verbindet Erzeugung einer Zelle und Initialisierung der referentiellen Anteils, beinhaltet aber nicht die Speicherung der Zellenadresse. Programm 4.3 zeigt die Implementierung.

In den folgenden Abschnitten soll nun basierend auf dem definierten Grundschritt ein Algorithmus zur primitiven Corekursion entwickelt werden. Dabei stellt sich heraus, daß sich ein solcher Algorithmus idealerweise rekursiv formulieren läßt. Um möglicher Verwirrung von Bezugsebenen vorzubeugen, wird der rekursiven eine iterative Darstellung vorausgeschickt, um so die Trennung der Rekursion auf der technischen und der Corekursion auf der inhaltlichen Ebene deutlich zu machen.

```
cons             :: Signature σ ⇒ σ Cell → HeapOp σ Cell
cons s           = do
                       (y, f) ← newCell s
                       transCover setField f
                       return y
consInterp c x = cons =≪ c x
```

Programm 4.3: Algorithmischer Grundschritt (algebraisch)

```
ana1          :: Ana
ana1 c v0 x0 = loop [(v0, x0)]
  where
    loop []           = return ()
    loop ((v, x) : r) = do
                          f ← genInterp c v x
                          loop (f ++ r)
```

Programm 4.4: Iterativer Berechnungsalgorithmus

4.2 Algorithmus ohne Zyklenerkennung

4.2.1 Iterative Formulierung

Eine erste Näherung eines passenden Algorithmus für eine abstrakte Interpretation $\mathbb{C} = (C, \gamma)$ läßt sich auf erstaunlich einfache Art konstruieren. Der Algorithmus $\mathsf{Ana}_1(\mathbb{C})$ verwaltet dazu

1. einen aktuellen Speicherzustand $\mathbb{A}$ über Σ, $\mathcal{A}$ und $\mathcal{B}$,
2. eine Arbeitsliste $\vec{s} = (\mathcal{B} \times C)^*$, die als zeitliche Abfolge von zukünftigen Inkarnationen vorgestellt werden kann. Jedes Element (v, x) steht für einen inkarnierten Schritt des Algorithmus, wobei x den Eingabeausdruck über der Cotermfamilie $(\mathbb{A} \bullet y)_{y \in \mathfrak{s}_{\mathbb{C}}(x)}$ darstellt, und v die Stelle, an welcher das Ergebnis abgelegt werden soll.

Am Anfang gilt $\vec{s} = [(v_0, x_0)]$. Ein Schritt ist wie folgt definiert. Ist die Liste $\vec{s}$ leer, dann terminiert der Algorithmus. Andernfalls:

1. Zerlege $\vec{s} = (v, x) : \vec{r}$.
2. Führe einen Grundschritt $\mathsf{Gen}\big(v, \gamma(x)\big) \mapsto f$ mit Nachzustand $\mathbb{A}'$ aus. Zähle den Graphen der Individualtransformation f als Liste $\vec{f}$ in beliebiger Reihenfolge auf.
3. Iteriere mit $\mathbb{A}'$ und $\vec{s}' = \vec{f} +\!\!+ \vec{r}$.

Im Endzustand $\mathbb{A}'$ verweist dann v_0 auf das konstruierte Ergebnis $y_0 = \mathbb{A}' \bullet \alpha_2'(v_0)$.

Satz 4.12. *Der Algorithmus* $\mathsf{Ana}_1(\mathbb{C})$ *ist sicher und referentiell transparent, wenn die Variable* v_0 *im Startzustand nicht lebendig ist.*

Beweis. Man betrachte einen beliebigen Schritt des Algorithmus mit Vorzustand $\mathbb{A}, \vec{s}$ und Nachzustand $\mathbb{A}', \vec{s}'$. Sei außerdem $\vec{s} = (v, x) : \vec{r}$ und $\vec{s}' = \vec{f} +\!\!+ \vec{r}$. Dann gilt:

1. Wenn v in $\mathbb{A}$ tot ist, dann sind alle Variablen aus $\vec{f}$ in $\mathbb{A}'$ tot.
2. Wenn eine Variable aus $\vec{r}$ in $\mathbb{A}$ tot ist, dann ist sie auch in $\mathbb{A}'$ tot.
3. Das Ergebnis $\alpha_2'(v)$ ist lebensfähig in jedem Zustand $\mathbb{A}'' \supseteq \mathbb{A}'$, der alle Variablen aus $\vec{f}$ mit lebensfähigen Zellen belegt.

Der Algorithmus weist erzeugte Zellen also stets nur toten Variablen zu, und erweitert damit die Coalgebra des Speicherzustands nicht. Erzeugte Ergebnisse sind aber lebensfähig bis auf die Stellen, die explizit durch hinzugekommene Elemente der Arbeitsliste markiert werden. Folglich ist das Endergebnis, bei dem die Arbeitsliste ja leer wird, vollständig lebensfähig, und kann an höherer Stelle referentiell transparent erreichbar gemacht werden. □

Beispiel 4.13 (Corekursive Berechnung). In Anwendung auf die oben eingeführten Beispiele ergeben sich Abläufe, die hier tabellarisch dargestellt sind. Jede Zeile zeigt für jeden Schritt

1. die aktuelle Arbeitsliste,
2. die aktuellen Parameter v, x,
3. die Anwendendung der Interpretation λ,
4. die durch den Grundschritt erzeugten zusätzlichen Belegungen im Nachzustand und
5. den Graphen der Individualtransformation, der zum Rest der Arbeitsliste für den nächsten Schritt hinzugefügt wird.

$\mathsf{Ana}_1(\mathbb{L})$ angewandt auf einen Startzustand $[(\mathsf{y}_1, \mathsf{a})]$ terminiert korrekt, während die Anwendung auf $[(\mathsf{y}_2, \mathsf{c})]$ auf konstantem Listenplatz divergiert und eine unbeschränkt absteigende Kette von s-Zellen erzeugt.

$\vec{s}$	v	x	$\lambda(x)$	$\mathbb{A}'$	$\vec{f}$
$[(\mathsf{y}_1, \mathsf{a})]$	y_1	a	$\mathsf{s(b)}$	$\alpha_1'(\mathsf{g}) = \mathsf{s}(\mathsf{g}_1)$ $\alpha_2'(\mathsf{y}_1) = \mathsf{g}$	$[(\mathsf{g}_1, \mathsf{b})]$
$[(\mathsf{g}_1, \mathsf{b})]$	g_1	b	z	$\alpha_1'(\mathsf{h}) = \mathsf{z}$ $\alpha_2'(\mathsf{g}_1) = \mathsf{h}$	$[]$
$[]$					
$[(\mathsf{y}_2, \mathsf{c})]$	y_2	c	$\mathsf{s(d)}$	$\alpha_1'(\mathsf{i}) = \mathsf{s}(\mathsf{i}_1)$ $\alpha_2'(\mathsf{y}_2) = \mathsf{i}$	$[(\mathsf{i}_1, \mathsf{d})]$
$[(\mathsf{i}_1, \mathsf{d})]$	i_1	d	$\mathsf{s(c)}$	$\alpha_1'(\mathsf{j}) = \mathsf{s}(\mathsf{j}_1)$ $\alpha_2'(\mathsf{i}_1) = \mathsf{j}$	$[(\mathsf{j}_1, \mathsf{c})]$
$[(\mathsf{j}_1, \mathsf{c})]$	j_1	c	$\mathsf{s(d)}$	$\alpha_1'(\mathsf{k}) = \mathsf{s}(\mathsf{k}_1)$ $\alpha_2'(\mathsf{j}_1) = \mathsf{k}$	$[(\mathsf{k}_1, \mathsf{d})]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

```
ana2  :: Ana
ana2 c = recur
  where
    recur v x  =  genInterp c v x  >>=  transCover recur
```

Programm 4.5: Rekursiver Berechnungsalgorithmus ohne Zyklenerkennung

4.2.2 Rekursive Formulierung

Der oben formulierte Algorithmus Ana_1 ist leicht als Tiefensuche zu erkennen. Eine äquivalente rekursive Formulierung vereinfacht die Argumentation hinsichtlich Korrektheit und Termination.

Ein rekursiver Schritt $\mathsf{Ana}_2(\mathbb{C})(v,x)$ mit $\mathbb{C} = (C,\gamma)$ ist wie folgt definiert:

1. Führe einen Grundschritt $\mathsf{Gen}\big(v,\gamma(x)\big) \mapsto f$ aus. Zähle den Graphen der Individualtransformation f als Liste $\vec{f}$ in beliebiger Reihenfolge auf.
2. Führe $\mathsf{Ana}_2(\mathbb{C})$ rekursiv nacheinander auf allen Elementen von $\vec{f}$ aus.

Wir schreiben also:

$$\mathbb{A} \xrightarrow{\mathsf{Ana}_2(\mathbb{C})(v,x)} \mathbb{A}' \iff \mathbb{A} \xrightarrow{\mathsf{Gen}(v,\gamma(x))\mapsto\{(v_1,x_1),\ldots,(v_n,x_n)\};\mathsf{Ana}_2(\mathbb{C})(v_1,x_1);\ldots;\mathsf{Ana}_2(\mathbb{C})(v_1,x_1)} \mathbb{A}'$$

Programm 4.5 zeigt die Implementierung. Die rekursive Traversierung wird durch die Hilfsfunktion *transCover* geleistet. Der monadische Sequenzoperator $\gg=$ macht die Präordnung der Traversierung explizit.

Satz 4.14 (Termination)**.** *Der Algorithmus* $\mathsf{Ana}_2(\mathbb{C})$ *terminiert ausgehend von Startzustand* $\mathbb{A}$ *und* (v,x)*, genau dann wenn die Teilinterpretation* $\mathfrak{s}^*_{\mathbb{C}}(x)$ *zyklenfrei ist.*

Beweis. Man betrachte die Rekurrenzrelation $\rightsquigarrow$ des Algorithmus Ana_2 auf seinem Parameter x unter Vernachlässigung von v. Es gelte $x \rightsquigarrow x'$. Dann gilt auch:

$$\begin{array}{rc} & x \to_{\mathbb{C}} x' \\ \text{(Zyklenfreiheit)} & \mathfrak{s}^*_{\mathbb{C}}(x) \supset \mathfrak{s}^*_{\mathbb{C}}(x') \end{array}$$

Da $\mathbb{C}$ rational ist, ist $\mathfrak{s}^*_{\mathbb{C}}(x)$ stets endlich. Also ist $w(x) = |\mathfrak{s}^*_{\mathbb{C}}(x)|$ eine Terminationsfunktion.

Für einen Zyklus $y \to^+_{\mathbb{C}} y$ ergibt sich dagegen eine unendliche Rekursion. □

Satz 4.15 (Partielle Korrektheit)**.** *Der Algorithmus* $\mathsf{Ana}_2(\mathbb{C})$ *ist partiell korrekt. Seien als Anfangszustand* $\mathbb{A}$ *und* (v,x) *gegeben. Wenn* $\mathsf{Ana}_2(\mathbb{C})$ *im Endzustand* $\mathbb{A}'$ *terminiert, dann gilt:*

$$[\![\mathbb{A}']\!]\big(\alpha'_2(v)\big) = [\![\overline{\mathbb{C}}]\!]\big(\mathit{In}\big([\![\mathbb{A}]\!]\big)(x)\big)$$

Beweis. Durch vollständige Induktion über die Umgebungen $\mathfrak{s}^*_{\mathbb{C}}(\_)$. Die Fundiertheit folgt aus dem Beweis für Satz 4.14.

Es gelte $\mathbb{A} \xrightarrow{\mathsf{Ana}_2(\mathbb{C})(v,x)} \mathbb{A}'$. Also existieren y, t und f, so daß:

$$\begin{array}{rcl} \text{(Definition 3.18)} & \alpha'_2(v) = y & \text{(a)} \\ \text{(Definition 3.12)} & \alpha'_1(y) = t & \text{(b)} \\ \text{(Rekursion)} & \Sigma(f)(t) = \gamma(x) & \text{(c)} \end{array}$$

Durch Anwendung der Induktionsannahme auf alle Elemente von $\mathrm{graph}\, f$ ergibt sich:

$$[\![\mathbb{A}']\!] \circ \alpha'_2 = [\![\overline{\mathbb{C}}]\!] \circ \mathit{In}\big([\![\mathbb{A}]\!]\big) \circ f \qquad \text{(i)}$$

Systematische Diagrammjagd:

$$
\begin{aligned}
& \big(\phi \circ [\![\mathbb{A}']\!] \circ \alpha'_2\big)(v) \\
\text{(a)} \quad & = \big(\phi \circ [\![\mathbb{A}']\!]\big)(y) \\
\text{(Anamorphismus)} \quad & = \big(\Sigma\big([\![\mathbb{A}']\!]\big) \circ \alpha'\big)(y) \\
\text{(Definition 3.3)} \quad & = \big(\Sigma\big([\![\mathbb{A}']\!]\big) \circ \Sigma(\alpha'_2) \circ \alpha'_1\big)(y) \\
\text{(Funktor)} \quad & = \big(\Sigma\big([\![\mathbb{A}']\!] \circ \alpha'_2\big) \circ \alpha'_1\big)(y) \\
\text{(b)} \quad & = \Sigma\big([\![\mathbb{A}']\!] \circ \alpha'_2\big)(t) \\
\text{(i)} \quad & = \Sigma\big([\![\overline{\mathbb{C}}]\!] \circ In\big([\![\mathbb{A}]\!]\big) \circ f\big)(t) \\
\text{(Funktor)} \quad & = \big(\Sigma\big([\![\overline{\mathbb{C}}]\!]\big) \circ \Sigma\big(In\big([\![\mathbb{A}]\!]\big)\big) \circ \Sigma(f)\big)(t) \\
\text{(c)} \quad & = \big(\Sigma\big([\![\overline{\mathbb{C}}]\!]\big) \circ \Sigma\big(In\big([\![\mathbb{A}]\!]\big)\big) \circ \gamma\big)(x) \\
\text{(Definition 4.7)} \quad & = \Big(\Sigma\big([\![\overline{\mathbb{C}}]\!]\big) \circ \overline{\gamma} \circ \Sigma\big(In\big([\![\mathbb{A}]\!]\big)\big)\Big)(x) \\
\text{(Anamorphismus)} \quad & = \Big(\phi \circ [\![\overline{\mathbb{C}}]\!] \circ \Sigma\big(In\big([\![\mathbb{A}]\!]\big)\big)\Big)(x)
\end{aligned}
$$

Durch Kürzen des Isomorphismus ϕ folgt die Behauptung. □

Beispiel 4.16 (Corekursive Berechnung). Da die Beispielinterpretation $\mathbb{L}$ eine linear rekursive Funktion generiert, können auch die Beispielabläufe für Ana_2 in tabellarischer Form dargestellt werden:

v	x	$\lambda(x)$	$\mathbb{A}'$	$\vec{f}$
y_1	a	$\mathsf{s(b)}$	$\alpha'_1(\mathsf{g}) = \mathsf{s}(\mathsf{g}_1)$ $\alpha'_2(\mathsf{y}_1) = \mathsf{g}$	$[(\mathsf{g}_1, \mathsf{b})]$
g_1	b	z	$\alpha'_1(\mathsf{h}) = \mathsf{z}$ $\alpha'_2(\mathsf{g}_1) = \mathsf{h}$	$[]$
y_2	c	$\mathsf{s(d)}$	$\alpha'_1(\mathsf{i}) = \mathsf{s}(\mathsf{i}_1)$ $\alpha'_2(\mathsf{y}_2) = \mathsf{i}$	$[(\mathsf{i}_1, \mathsf{d})]$
i_1	d	$\mathsf{s(c)}$	$\alpha'_1(\mathsf{j}) = \mathsf{s}(\mathsf{j}_1)$ $\alpha'_2(\mathsf{i}_1) = \mathsf{j}$	$[(\mathsf{j}_1, \mathsf{c})]$
j_1	c	$\mathsf{s(d)}$	$\alpha'_1(\mathsf{k}) = \mathsf{s}(\mathsf{k}_1)$ $\alpha'_2(\mathsf{j}_1) = \mathsf{k}$	$[(\mathsf{k}_1, \mathsf{d})]$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

```
ana2a  :: Ana
ana2a c = recur []
  where
    recur  s v x  =  let recur' =  recur ((v, x) : s)
                     in genInterp c v x >>= transCover recur'
```

Programm 4.6: Rekursiver Berechnungsalgorithmus mit Zyklenprotokollierung

4.3 Algorithmus mit Zyklenerkennung

Um einen Zyklus erfolgreich zu erkennen und zu behandeln, ist zweierlei Information nötig:

1. Zur *Erkennung* der Pfad der rekursiv besuchten Eingaben. Genau dann wenn ein Ausdruck x zweimal vorkommt, liegt ein Zyklus vor.
2. Zur *Behandlung* die zugehörigen Ergebnisvariablen. Seien (v', x) und (v, x) zwei ineinander geschachtelte Ausrufe. Um den Zyklus abschließen zu können, müssen v' und v gleich belegt werden, indem der Wert der äußeren, bereits belegten Variablen v' in die innere, zum Zeitpunkt der Zyklenerkennung noch unbelegte Variable v kopiert wird.

Es bietet sich also an, eine Informationsquelle zu nutzen, die in den meisten Implementierungen strikter Berechnung ohnehin vorhanden ist: Der *Stack* der Prozedurinkarnationen (*activation records*). Programm 4.6 zeigt die grundsätzliche Implementierung. Hier wird der Stack als zusätzliches Argument $\vec{s}$ explizit gemacht. Man beachte jedoch, daß $\vec{s}$ eine etwas andere Bedeutung als bei Algorithmus Ana_1 hat: Hier enthält der Stack von jeder Individualtransformation f immer nur ein Element gleichzeitig.

Der folgende Algorithmus Ana_3 fügt dann die eigentliche Erkennung und Behandlung des Zyklus hinzu. Programm 4.7 zeigt die Implementierung. Dazu wird die durch den Signaturfunktor *In* induzierte Gleichheit der Eingaben ($==^*$) verwendet. Zur besseren Lesbarkeit der schrittweisen Verfeinerung werden jeweils die unveränderten Anteile der Algorithmen hellgrau dargestellt.

Beispiel 4.17 (Corekursive Berechnung). Der Aublauf der Beispielrechnungen für Ana_{2a} entspricht dem für Ana_2, außer daß sich zusätzlich die besuchten Paare auf dem Stack akkumulieren. In der mit $(*)$ markierten Zeile liegt bereits ein Zyklus vor, da sich die aktuelle Eingabe c bereits auf dem Stack befindet.

$\vec{s}$	v	x	$\lambda(x)$	$\mathbb{A}'$	$\vec{f}$	
$[]$	y_1	a	$\mathsf{s(b)}$	$\alpha'_1(\mathsf{g}) = \mathsf{s}(\mathsf{g}_1)$ $\alpha'_2(\mathsf{y}_1) = \mathsf{g}$	$[(\mathsf{g}_1, \mathsf{b})]$	
$[(\mathsf{y}_1, \mathsf{a})]$	g_1	b	z	$\alpha'_1(\mathsf{h}) = \mathsf{z}$ $\alpha'_2(\mathsf{g}_1) = \mathsf{h}$	$[]$	
$[]$	y_2	c	$\mathsf{s(d)}$	$\alpha'_1(\mathsf{i}) = \mathsf{s}(\mathsf{i}_1)$ $\alpha'_2(\mathsf{y}_2) = \mathsf{i}$	$[(\mathsf{i}_1, \mathsf{d})]$	
$[(\mathsf{y}_2, \mathsf{c})]$	i_1	d	$\mathsf{s(c)}$	$\alpha'_1(\mathsf{j}) = \mathsf{s}(\mathsf{j}_1)$ $\alpha'_2(\mathsf{i}_1) = \mathsf{j}$	$[(\mathsf{j}_1, \mathsf{c})]$	
$[(\mathsf{i}_1, \mathsf{d}), (\mathsf{y}_2, \mathsf{c})]$	j_1	c	$\mathsf{s(d)}$	$\alpha'_1(\mathsf{k}) = \mathsf{s}(\mathsf{k}_1)$ $\alpha'_2(\mathsf{j}_1) = \mathsf{k}$	$[(\mathsf{k}_1, \mathsf{d})]$	$(*)$
$[(\mathsf{j}_1, \mathsf{c}), (\mathsf{i}_1, \mathsf{d}), (\mathsf{y}_2, \mathsf{c})]$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Ein rekursiver Schritt $\mathsf{Ana}_3(\mathbb{C})(\vec{s}, v, x)$ mit Zyklenerkennung ist wie folgt definiert:

1. Falls der Stack $\vec{s}$ bereits ein Paar (v', x) enthält, kopiere lediglich das Ergebnis von v' nach v. Andernfalls erweitere den Stack zu $(v, x) : \vec{s}$ und fahre wie oben fort:

```
ana3 :: Ana
ana3 c = recur []
  where
    recur s v x = let cycle (_, x') = x ==* x'
                      recur'        = recur ((v, x) : s)
                  in case find cycle s of
                       Just (v', _) → getField v' >>= setField v
                       Nothing      → genInterp c v x >>= transCover recur'
```

Programm 4.7: Rekursiver Berechnungsalgorithmus mit Zyklenbehandlung

2. Führe einen Grundschritt $\mathsf{Gen}\big(v, \gamma(x)\big) \mapsto f$ aus. Zähle den Graphen der Individualtransformation f als Liste $\vec{f}$ in beliebiger Reihenfolge auf.

3. Führe $\mathsf{Ana}_3(\mathbb{C})$ rekursiv nacheinander auf allen Elementen von $\vec{f}$ mit dem erweiterten Stack aus.

Dieser Algorithmus generalisiert das bisherige Verfahren auf beliebige, auch zyklische, endliche Objekte.

Satz 4.18 (Termination)**.** *Der Algorithmus* $\mathsf{Ana}_3(\mathbb{C})$ *terminiert ausgehend von Startzustand* $\mathbb{A}$*, beliebigem Stack und* (v, x)*.*

Beweis. Man betrachte die Rekurrenzrelation $\rightsquigarrow$ des Algorithmus Ana_3 auf seinen Parametern $\vec{s}$ und x unter Vernachlässigung von v. Außerdem sei die Menge $M(\vec{s}) \subseteq C$ der markierten Eingabeelemente definiert als $M(\vec{s}) = \{x \mid \exists\, v.\ (v, x) \in \operatorname{codom} \vec{s}\}$.

Es gelte nun $(\vec{s}, x) \rightsquigarrow (\vec{s}', x')$. Dann gilt auch:

$$\mathfrak{s}^*_{\mathbb{C}}(x') \subseteq \mathfrak{s}^*_{\mathbb{C}}(x) \qquad\qquad M(\vec{s}) \subset M(\vec{s}')$$

Definiert man nun die unmarkierte Umgebung $W(\vec{s}, x) = \mathfrak{s}^*_{\mathbb{C}}(x) \setminus M(\vec{s})$, dann gilt

$$W(\vec{s}', x') \subset W(\vec{s}, x)$$

Da $\mathbb{C}$ rational ist, ist $\mathfrak{s}^*_{\mathbb{C}}(x)$ stets endlich. Also ist $w(\vec{s}, x) = |W(\vec{s}, x)|$ eine Terminationsfunktion. □

Satz 4.19 (Partielle Korrektheit)**.** *Der Algorithmus* $\mathsf{Ana}_3(\mathbb{C})$ *ist partiell korrekt. Seien als Anfangszustand* $\mathbb{A}$*, ein leerer Stack und* (v, x) *gegeben. Wenn* $\mathsf{Ana}_3(\mathbb{C})$ *im Endzustand* $\mathbb{A}'$ *terminiert, dann gilt:*

$$[\![\mathbb{A}']\!]\big(\alpha'_2(v)\big) = [\![\overline{\mathbb{C}}]\!]\big(\mathit{In}\big([\![\mathbb{A}]\!]\big)(x)\big)$$

Beweis. Da es sich hier um ein corekursives Verfahren handelt, ist auch eine coinduktive Beweistechnik angebracht. Der Beweis unterteilt sich in mehrere Schritte:

1. *Formulierung einer Farbmetapher.* Wir betrachten die Relation R zwischen Eingaben des Algorithmus und den als Ausgabe belegten Zellen. Diese zerfällt in eine Reihe ausgezeichneter Teilmengen mit stufenweise stärkeren Eigenschaften, die durch Farben bezeichnet werden:

   (a) Ein Paar (x, y) heißt *weiß*, wenn keine der folgenden Eigenschaften zutrifft. Nicht weiße Paare heißen *gefärbt.*

   (b) Ein Paar (x, y) heißt *hellgrau*, wenn x und y kompatible lokale Datensätze zugeordnet sind, und keine der folgenden Eigenschaften zutrifft. Nicht hellgraue gefärbte Paare heißen *dunkel.*

   (c) Ein Paar (x, y) heißt *dunkelgrau*, wenn außerdem alle Paare in der unmittelbaren Umgebung $\big(\bar{\mathfrak{s}}(\Sigma) \circ \upsilon(\mathbb{A}', \mathbb{A}')\big)\big((x, y)\big)$ gefärbt sind.

(d) Ein Paar (x, y) heißt *schwarz*, wenn außerdem alle Paare in der unmittelbaren Umgebung $\big(\bar{\mathfrak{s}}(\Sigma) \circ \upsilon(\mathbb{A}', \mathbb{A}')\big)\big((x, y)\big)$ schwarz sind. Äquivalent gilt, daß alle Paare in der transitiven Umgebung gefärbt sind.

2. *Geeignete Umcodierung.* Anstelle der Relation $R \subseteq \mathit{In}(\mathcal{A}) \times \mathcal{A}$ betrachten wir die Relation $\widetilde{R} \subseteq \mathcal{B} \times \mathit{In}(\mathcal{A})$ zwischen Ergebnisvariablen und Eingaben. Diese entspricht unmittelbar den Parametern des Algorithmus. Im Endzustand $\mathbb{A}' = (A', \alpha_1', \alpha_2')$ gilt:

$$(v, x) \in \widetilde{R} \iff \big(x, \alpha_2'(v)\big) \in R$$

Und die der unmittelbaren Umgebung entsprechenden Paare ergeben sich aus der Individualtransformation von Ausgabe zu Eingabe:[2]

$$\big((\alpha_1' \circ \alpha_2')(v) \Rightarrow x\big)(v') = x' \iff \big(x', \alpha_2'(v')\big) \in \big(\bar{\mathfrak{s}}(\Sigma) \circ \upsilon(\mathbb{A}', \mathbb{A}')\big)\big((x, \alpha_2'(v))\big)$$

Da der Algorithmus rekursiv stets mit frischen Variablen aufgerufen wird, ist die Relation $\widetilde{R}$ rechtseindeutig und beschreibt den Graphen einer partiellen Abbildung.

3. *Aufstellen einer Invariante.* Der Stack enthält nur gefärbte Paare.

4. *Formulieren einer Nachbedingung.* Ein Aufruf von $\mathsf{Ana}_3(\mathbb{C})(\vec{s}, v, x)$ heiße *zyklisch*, falls der Stack $\vec{s}$ ein Paar (v', x) enthält, sonst *azyklisch*. Dann gilt:

   (a) Jeder Aufruf färbt seine Parameter.

   (b) Ein azyklischer Aufruf färbt seine Parameter dunkel.

5. *Induktiver Beweis von Invariante und Nachbedingung.*

   (a) Beim Aufruf von $\mathsf{Ana}_3(\mathbb{C})(\vec{s}, v, x)$ ist das Paar (v, x) im Allgemeinen ungefärbt, und alle Paare in $\vec{s}$ gefärbt.

   (b) Enthält der Stack $\vec{s}$ ein Paar (v', x), so wird v' nach v kopiert. Da (v', x) per Invariante gefärbt ist, wird (v, x) ebenfalls gefärbt.

   (c) Andernfalls wirf durch Ausführung von $\mathsf{Gen}\big(v, \gamma(x)\big) \mapsto f$ das Paar (v, x) gefärbt. Da der Algorithmus transparent ist, bleibt diese Färbung stets erhalten.[3] Damit gilt die Invariante für den Stack $(v, x) : \vec{s}$ unmittelbarer rekursiver Aufrufe.

   Der Graph $\vec{f}$ der Individualtransformation f, also die gesamte unmittelbare Umgebung, wird rekursiv traversiert und dabei per Induktionsannahme gefärbt. Also wird das Paar (v, x) dunkel.

6. *Anwendung der Invariante auf den Endzustand.* Man betrachte den Endzustand der Berechnung $\mathsf{Ana}_3(\mathbb{C})([], v, x)$. Der Algorithmus hat alle von (v, x) transitiv erreichbaren Paare traversiert. Da der Stack initial leer ist, hat für alle erreichbaren Paare mindestens ein azyklischer Aufruf stattgefunden. Also ist (v, x) schwarz. Daraus folgt nach Lemma 2.101 die Bisimilarität $\alpha_2'(v) \sim x$. Das Problem, das durch die hier vorgestellten Algorithmen gelöst wird, kann also auch, ohne Bezug zur finalen Semantik, als Konstruktion eines bisimilaren Coterms im Speicher verstanden werden.

7. *Coinduktive Folgerung.* Wir zeigen zuerst, daß durch Vorschalten der Abstraktion $\mathit{In}\big([\![\mathbb{A}]\!]\big)$ die Bisimilarität erhalten bleibt.

[2] Man beachte den leider sehr geringen typographischen Unterschied zwischen der Variable v und der Bisimulationsoperation υ!

[3] Einer der seltenen Fälle, in denen unabhängig voneinander gewählte Metaphern sich konsistent ergänzen; gemeint ist natürlich die Folgerung auf der mathematischen Ebene.

(a) Sei $In = \mathcal{I}$. Es gilt:

$$\begin{aligned}
& & \alpha'_2(v) &\sim x \\
& \text{(Satz 2.69)} & [\![\mathbb{A}^\dagger]\!](\alpha'_2(v)) &= [\![\mathbb{C}]\!](x) \\
& \text{(Eindeutigkeit des Anamorphismus)} & [\![\mathbb{A}^\dagger]\!](\alpha'_2(v)) &= ([\![\overline{\mathbb{C}}]\!] \circ [\![\mathbb{A}]\!])(x) \\
& \text{(Satz 2.69)} & \alpha'_2(v) &\sim [\![\mathbb{A}]\!](x)
\end{aligned}$$

(b) Alle anderen Fälle folgen durch strukturelle Induktion in In.

Damit folgt die Behauptung aus Satz 2.69:

$$\begin{aligned}
\alpha'_2(v) &\sim In([\![\mathbb{A}]\!])(x) \\
[\![\mathbb{A}']\!](\alpha'_2(v)) &= [\![\overline{\mathbb{C}}]\!](In([\![\mathbb{A}]\!])(x))
\end{aligned}$$

□

Beispiel 4.20 (Corekursive Berechnung). Der Aublauf der Beispielrechnungen für Ana_3 entspricht im Wesentlichen dem für Ana_{2a}. Ein entstehender Zyklus wird anhand des Stacks erkannt und durch Kopieren des passenden Ergebnisses behandelt.

$\vec{s}$	v	x	$\lambda(x)$	$\mathbb{A}'$	$\vec{f}$
$[]$	y_1	a	$\mathsf{s(b)}$	$\alpha'_1(\mathsf{g}) = \mathsf{s}(\mathsf{g}_1)$ $\alpha'_2(\mathsf{y}_1) = \mathsf{g}$	$[(\mathsf{g}_1, \mathsf{b})]$
$[(\mathsf{y}_1, \mathsf{a})]$	g_1	b	z	$\alpha'_1(\mathsf{h}) = \mathsf{z}$ $\alpha'_2(\mathsf{g}_1) = \mathsf{h}$	$[]$
$[]$	y_2	c	$\mathsf{s(d)}$	$\alpha'_1(\mathsf{i}) = \mathsf{s}(\mathsf{i}_1)$ $\alpha'_2(\mathsf{y}_2) = \mathsf{i}$	$[(\mathsf{i}_1, \mathsf{d})]$
$[(\mathsf{y}_2, \mathsf{c})]$	i_1	d	$\mathsf{s(c)}$	$\alpha'_1(\mathsf{j}) = \mathsf{s}(\mathsf{j}_1)$ $\alpha'_2(\mathsf{i}_1) = \mathsf{j}$	$[(\mathsf{j}_1, \mathsf{c})]$
$[(\mathsf{i}_1, \mathsf{d}), (\mathsf{y}_2, \mathsf{c})]$	j_1	c		$\alpha'_2(\mathsf{j}_1) = \alpha'_2(\mathsf{y}_2) = \mathsf{i}$	

□

Damit ist die strikte Implementierung primitiver Corekursion korrekt und für rationale Datentypen, repräsentiert durch endliche Coterme auch vollständig abgeschlossen. Lediglich unendliche Coterme können nicht „berechnet“ werden; dieses Privileg bleibt nichtstrikten Implementierungen vorbehalten. Im folgenden Kapitel wird dagegen eine Technik behandelt, die in rein algebraischen Systemen unabhängig von Striktheit *nicht* anwendbar ist.

Kapitel 5

Entscheiden zyklischer Prädikate durch coalgebraische Logik

In diesem Kapitel soll das Entscheiden logischer Prädikate untersucht werden. Im Vergleich zum Berechnen einer zyklischen Funktion treten dabei zwei zusätzliche Probleme auf:

1. Es genügt nicht, einen Zyklus in der Eingabe des Verfahrens auf einen Zyklus in der Ausgabe abzubilden. Stattdessen muß der Zyklus zu einem einzigen stabilen Wahrheitswert kollabieren. Dies gelingt, indem das einmalige Durchlaufen eines Zyklus auf eine idempotente BOOLEsche Operation abgebildet wird.

2. Wenn in einer logischen Definition Zyklen auftreten, besitzt sie im Allgemeinen verschiedene potentiell intendierte Modelle, unter denen eines ausgewählt werden muß. Eine gründliche Untersuchung des Phänomens findet sich in [42], wo jedoch wesentlich weniger feine Abstufungen unterschieden werden, als in der hier entwickelten, alternativen Theorie.

Andererseits ist die Lösbarkeit des gesamten Problems durchaus relevant, da sie einen erheblichen Mehrwert des strikten Ansatzes zur Zyklenbehandlung darstellt: Ein Prädikat über zyklischen Eingaben, welches nicht vollständig auf endliche Anfangsstücke reduziert werden kann, ist mit einem nichtstrikten algebraischen Ansatz nicht zu entscheiden. Als elementares Beispiel kann die Klassifikation einer Liste als endlich oder unendlich dienen.

Die Untersuchungen in diesem Kapitel beginnen mit einer Definition von logischen Kalkülen, ihrer Strukturierbarkeit und Semantik. Anschließend wird den zugrundeliegenden Formelmengen eine algebraische beziehungsweise coalgebraische Struktur aufgeprägt, aus der sich Entscheidungsverfahren ableiten lassen. Die Entwicklung eines allgemeinen, zyklentauglichen Verfahrens wird analog zum vorigen Kapitel in mehrere, leichter nachvollziehbare Schritte unterteilt. Dabei werden dieselben Techniken zur Zyklenerkennung eingesetzt, so daß sich beide Verfahren in einem gemeinsamen Rahmen implementieren lassen. Abschließend wird auf das Problem der semantischen Unterbestimmtheit und auf Techniken zur Spezifikation eines intendierten Modells eingegangen.

Parallel zur umgangssprachlichen und mathematischen Spezifikation der entwickelten Lösungen werden ausführbare Haskell-Modelle vorgestellt.

5.1 Logik-Kalküle

Im folgenden soll von den meisten syntaktischen Phänomenen der Logik abstrahiert werden, nämlich von *Variablen und Quantifikation*, sowie *Junktoren*, insbesondere der *Negation*. Übrig bleibt die Untersuchung von *operationalen Deduktionssystemen* (HILBERT-Kalkülen) auf *atomaren Formeln*. Um Formeln in dem

gleichen Rahmen repräsentieren zu können wie andere Daten, betrachten wir hier speziell Coterme, also Elemente von Coalgebren über polynomiellen Signaturfunktoren.

5.1.1 Kalküle und Regeln

Definition 5.1 (Kalkül)**.** Sei Σ eine Signatur und $\mathbb{C} = (C, \gamma)$ eine Σ-Coalgebra. Ein $\mathbb{C}$-Kalkül $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ist definiert durch eine *Formelmenge* $K \subseteq C$ und eine *Ableitungsrelation* $\vdash_{\mathbb{K}} : K^* \leftrightarrow K$, wobei gelte:

1. K ist Träger einer Untercoalgebra von $\mathbb{C}$.

2. Zu jeder rechten Seite $\psi \in K$ existiert nur eine endliche Menge linker Seiten $\overline{\mathcal{P}}(\vdash_{\mathbb{K}})(\{\psi\})$.

Bezüglich einer Ableitung $\vec{\varphi} \vdash_{\mathbb{K}} \psi$ heißen die Elemente der Liste $\vec{\varphi} = [\varphi_1, \ldots, \varphi_n]$ die *Prämissen* und ψ die *Konklusion* der Ableitung. □

Definition 5.2 (Regel)**.** Falls möglich, wird die Ableitungsrelation eines Kalküls gerne in Form von schematischen Regeln dargestellt. Dazu werden Σ-Pattern zur Spezifikation von Formeln aus der Σ-Coalgebra $\mathbb{C} = (C, \gamma)$ eingesetzt. Eine Regel hat also die Form:

$$\frac{p_1 \quad \cdots \quad p_n}{q}$$

Dabei sind die p_i Prämissen-Pattern und q ist das Konklusions-Pattern. Einer solchen Regel sind unter einer Typisierung $t : V \to \mathcal{P}(K)$ als Ableitungsrelation alle Paare $\big([\varphi_1, \ldots, \varphi_n], \psi\big)$ zugeordnet, für die gilt:

1. Die Pattern sind einzeln t-passend.

$$p_1 :=_t \varphi_1 \qquad \ldots \qquad p_n :=_t \varphi_n \qquad q :=_t \psi$$

2. Die Bedeutungen der Pattern sind widerspruchsfrei.

$$[\![p_1]\!]^{\mathbb{C}}_{\Sigma}(\varphi_1) \sqcup \cdots \sqcup [\![p_n]\!]^{\mathbb{C}}_{\Sigma}(\varphi_n) \sqcup [\![q]\!]^{\mathbb{C}}_{\Sigma}(\psi) \neq \bot$$

   Dadurch ist gewährleistet, daß gleichlautende Variablen in den beteiligten Formeln übereinstimmend gebunden sind.

Die Ableitungsrelation des Kalküls ist definiert als die Vereinigung der Relationen aller seiner Regeln. Die zugrundeliegende Coalgebra $\mathbb{C}$ und die Typisierung t wird entweder gar nicht oder implizit durch den Kontext spezifiziert. □

Man beachte, daß die Ableitungsrelation nicht den transitiven Abschluß über den Regeln beinhaltet, siehe dazu Abschnitt 5.1.3.

Zur einfachen Notation von Pattern verwenden wir unterschiedliche Schrifttypen: x, y stellen Variablen dar; a, b Injektionen. Somit ist $\mathsf{a}(\mathsf{b})$ ein Grundterm, $\mathsf{a}(x)$ ein offener Term. Es werden nur kartesische Produkte verwendet, so daß sich die gewohnte Termsyntax ergibt.

Beispiel 5.3 (Regeln). Einige Regeln, die im folgenden zu beispielhaften Kalkülen kombiniert werden sollen:

$$\frac{}{\mathsf{zero}}\,R_0 \qquad \frac{n}{\mathsf{succ}(n)}\,R_1 \qquad \frac{\mathsf{succ}(n)}{n}\,R_2$$

$$\frac{\mathsf{u}(n)}{\mathsf{g}\big(\mathsf{succ}(n)\big)}\,B_1 \qquad \frac{\mathsf{g}(n)}{\mathsf{u}\big(\mathsf{succ}(n)\big)}\,B_2$$

$$\frac{\mathsf{ping}}{\mathsf{ping}}\,P_1 \qquad \frac{\mathsf{ping}}{\mathsf{pong}}\,P_2 \qquad \frac{\mathsf{pong}}{\mathsf{ping}}\,P_3 \qquad \frac{\mathsf{pong}}{\mathsf{pong}}\,P_4$$

$$\frac{}{\big(\mathsf{zero},\mathsf{zero}\big)}\,J_0 \qquad \frac{}{\big(\mathsf{zero},\mathsf{succ}(n)\big)}\,J_1 \qquad \frac{(m,n)}{\big(\mathsf{succ}(m),\mathsf{succ}(n)\big)}\,J_2$$

$$\frac{}{\mathsf{o}^*(\mathsf{nil},\mathsf{nil})}\,L_0 \qquad \frac{}{\mathsf{o}^*\big(\mathsf{nil},\mathsf{cons}(x,\vec{r})\big)}\,L_1$$

$$\frac{\mathsf{o}(x,y)}{\mathsf{o}^*\big(\mathsf{cons}(x,\vec{r}),\mathsf{cons}(y,\vec{s})\big)}\,L_2 \qquad \frac{\mathsf{e}(x,y)\quad \mathsf{o}^*(\vec{r},\vec{s})}{\mathsf{o}^*\big(\mathsf{cons}(x,\vec{r}),\mathsf{cons}(y,\vec{s})\big)}\,L_3$$

In gewohnter Termnotation wurde die Applikation des $\mathcal{C}_1$-Patterns ($\star$) weggelassen.

Man beachte, daß einige Regeln keine Prädikatsymbole, sondern lediglich deren Argumente (Datensätze über Individuen) enthalten. Die Zuordnung zu einem Prädikat ergibt sich pragmatisch dadurch, daß ein Kalkül zu dessen Spezifikation herangezogen wird. In ihrer abstrakten Form lesen sich die Formeln in etwa wie folgt:

1. R_0 besagt: *Die hier spezifizierte Eigenschaft trifft auf alle durch das Pattern* zero *beschriebenen Individuen zu.*

2. R_1 besagt: *Die hier spezifizierte Eigenschaft trifft auf ein durch das Pattern* $\mathsf{succ}(n)$ *beschriebenes Individuum zu, wenn sie auf das Individuum zutrifft, mit dem n jeweils belegt ist.*

Wie die Regeln zu Kalkülen zusammengesetzt werden, die nützliche Prädikate spezifizieren, und was diese genau bedeuten, wird in den Beispielen 5.6 und 5.17 ausgeführt.

5.1.2 Kombination von Kalkülen

In diesem Abschnitt sollen einige Konstruktionen für zusammengesetzte Kalküle beschrieben werden. Diese werden im Folgenden allerdings nicht praktisch eingesetzt, um konkrete Kalküle stückweise zu spezifizieren. Stattdessen wird gezeigt werden, daß einige für die folgenden Abschnitte relevante Eigenschaften über Konstruktionen hinweg erhalten bleiben. Die Dekomposition eines Kalküls stellt also eine Beweistechnik für solche Eigenschaften dar. Zur Anwendung siehe Abschnitt 5.4.

Definition 5.4 (Vereinigung)**.** Die *Vereinigung* von Kalkülen ist folgendermaßen definiert: Sei Σ eine Signatur und $\mathbb{C} = (C, \gamma)$ eine Σ-Coalgebra. Sei I eine endliche Indexmenge und $\left(\mathbb{K}_i = (K_i, \vdash_{\mathbb{K}_i})\right)_{i \in I}$ eine Familie von $\mathbb{C}$-Kalkülen.

$$\bigcup_{i \in I} \mathbb{K}_i = \left(\bigcup_{i \in I} K_i, \bigcup_{i \in I} \vdash_{\mathbb{K}_i} \right)$$

$\bigcup \mathbb{K}_i$ ist ebenfalls ein $\mathbb{C}$-Kalkül. □

Auf der Ebene von Regeln entspricht die Vereinigung der Kalküle der Vereinigung der erzeugenden Regelmengen. Dabei können neue Zyklen (Rekursionen) entstehen.

Definition 5.5 (Coprodukt)**.** Das *Coprodukt* von Kalkülen ist folgendermaßen definiert: Sei I eine endliche Indexmenge. Sei für jedes $i \in I$ Σ_i eine Signatur, $\mathbb{C}_i$ eine Σ_i-Coalgebra und $\mathbb{K}_i = (K_i, \vdash_{\mathbb{K}_i})$ ein $\mathbb{C}_i$-Kalkül. Dann ist der Coproduktkalkül wie folgt definiert:

$$\coprod_{i \in I} \mathbb{K}_i = \left(\coprod_{i \in I} K_i, \widetilde{\coprod_{i \in I}} \vdash_{\mathbb{K}_i} \right)$$

Dabei wird die Ableitungsrelation auf das Coprodukt angehoben:

$$\widetilde{\coprod_{i \in I}} \vdash_{\mathbb{K}_i} = \bigcup_{i \in I} \mathcal{P}(\iota_i^* \times \iota_i)(\vdash_{\mathbb{K}_i})$$

Damit existiert eine Ableitung $[\varphi_1, \ldots, \varphi_n] \vdash_{\coprod \mathbb{K}_i} \psi$ genau dann wenn ein $i \in I$ existiert, so daß:

$$\varphi_k = \iota_i(\varphi'_k) \qquad \psi_k = \iota_i(\psi') \qquad [\varphi'_1, \ldots, \varphi'_n] \vdash_{\mathbb{K}_i} \psi'$$

$\coprod \mathbb{K}_i$ ist ein $(\coprod \mathbb{C}_i)$-Kalkül. □

Auf der Ebene von Regeln entspricht die Vereinigung der Kalküle der Vereinigung der erzeugenden Regelmengen, nachdem jede in der i-ten Regelmenge vorkommende Formel oder Teilformel φ durch $\iota_i(\varphi)$ ersetzt wurde. Dabei können keine neuen Zyklen entstehen.

Die binären Spezialfälle $I = \{1, 2\}$ schreiben wir $\mathbb{K}_1 \cup \mathbb{K}_2$ beziehungsweise $\mathbb{K}_1 + \mathbb{K}_2$. Den unären Spezialfall $I = \{1\}$ des Coprodukts mit der Injektion ι_1 schreiben wir $\iota_1(\mathbb{K})$.

Beispiel 5.6 (Kombinierte Kalküle). Einige Lesarten und Kombinationsmöglichkeiten der oben eingeführten Regeln. Dabei wird nur über die Korrektheit der Regeln für ein gegebenes Problem argumentiert. Für den Nachweis der behaupteten Semantik einschließlich Vollständigkeit sei auf spätere Abschnitte verwiesen.

1. Über der Signatur $\Sigma = \mathcal{C}_1 + \mathcal{I}$ der natürlichen Zahlen mit den Injektionen $\mathsf{zero}, \mathsf{succ}$ und den beiden bereits aus Beispiel 2.102 hinlänglich bekannten Σ-Coalgebren $\mathbb{N}$ und $\bar{\mathbb{N}}$ konstruieren wir einige Kalküle zur Spezifikation von Zahleigenschaften. Dabei sei jeweils $\mathbb{K}$ ein $\mathbb{N}$-Kalkül, und $\bar{\mathbb{K}}$ ein analog definierter $\bar{\mathbb{N}}$-Kalkül.

   (a) Die Kalküle $\mathbb{F}, \bar{\mathbb{F}}$, definiert durch das Regelsystem $R_0 \cup R_1$ charakterisieren endliche Zahlen: Die Null ist endlich. Ist eine Zahl n endlich, dann ist auch ihr Nachfolger endlich.

(b) Die Kalküle $\mathbb{G}, \bar{\mathbb{G}}$, definiert *nur* durch die Regel R_1, charakterisieren unendliche Zahlen: Ist eine Zahl n unendlich, dann ist auch ihr Nachfolger unendlich.

(c) Die Vereinigung beider in einem gemeinsamen System gelingt mit $\mathbb{H} = \mathbb{F} + \mathbb{G}$. Wir schreiben $\mathsf{fin}, \mathsf{inf}$ für die beiden Injektionen. Damit ergibt sich das Regelsystem:

$$\frac{}{\mathsf{fin}(\mathsf{zero})}\,H_1 \qquad \frac{\mathsf{fin}(n)}{\mathsf{fin}\big(\mathsf{succ}(n)\big)}\,H_2 \qquad \frac{\mathsf{inf}(n)}{\mathsf{inf}\big(\mathsf{succ}(n)\big)}\,H_3$$

(d) Der Kalkül $\mathbb{G}'$, definiert durch die Regel R_2 gleicht scheinbar $\mathbb{G}$. Der semantische Unterschied wird im nächsten Abschnitt erläutert.

(e) Der Kalkül $\mathbb{B} = \mathsf{g}(R_0) \cup B_1 \cup B_2$ unterscheidet gerade und ungerade Zahlen: Die Null ist gerade. Ist eine Zahl n gerade, dann ist ihr Nachfolger ungerade, und umgekehrt.

2. Sei $P = \{\mathsf{ping}, \mathsf{pong}\}$. Über der minimalen Signatur $\Sigma = \mathbb{C}_P$ und deren finaler Coalgebra (P, id) definieren wir die pathologischen Pingpong-Kalküle $\mathbb{P}_\times = P_2 \cup P_3$ und $\mathbb{P}_{\|} = P_1 \cup P_4$. Diese sind mit keiner praktischen Anwendung verbunden. In Abschnitt 5.4 werden sie als Gegenbeispiele für eine Annahme dienen.

3. Der Kalkül $\mathbb{J} = J_1 \cup J_2$ charakterisiert die Ordnung $<$ natürlicher Zahlen: Die Null ist die kleinste Zahl. Die Nachfolgeroperation ist monoton. Eine Erweiterung um die Regel J_0 zum Kalkül $\mathbb{J}'$ ergibt die Ordnung $\leq$. Wie oben bilden wir auch $\bar{\mathbb{J}}, \bar{\mathbb{J}}'$.

   Eine natürliche zugrundeliegende Coalgebra für diese Kalküle nicht leicht zu finden, da es sich bei den Formelelementen nicht um einzelne Zahlen wie bei den vorherigen Beispielen, sondern um Paare handelt. Ein allgemeines Verfahren zur Konstruktion von Coalgebren für mehrstellige Prädikate aus Coalgebren für deren Argumente wird in Abschnitt 5.1.2.1 vorgestellt.

4. Der Kalkül $\mathbb{L} = L_1 \cup L_2 \cup L_3$ charakterisiert die lexikographische Ordnung $<$ von Listen: Die leere Liste ist die kleinste Liste. Nichtleere Listen sind nach dem ersten Element oder bei Gleichheit desselben nach der Restliste zu ordnen. Das Hinzufügen der Regel L_0 ergibt den Kalkül $\mathbb{L}'$ und die Ordnung $\leq$.

   Als Obercoalgebra für die auftretenden Individuen betrachten wir die finale Coalgebra der aus Abschnitt 2.5.3 bekannten gemeinsamen Signatur der natürlichen Zahlen und Listen. Die Typisierung für die Variablen x, y sei der initiale Zahlentyp in dieser Coalgebra. Die Typisierung für die Variablen $\vec{r}, \vec{s}$ sei im Kalkül $\mathbb{L}$ der initiale beziehungsweise im Kalkül $\bar{\mathbb{L}}$ der finale Listentyp.

   An diesen Kalkülen fallen zwei weitere Besonderheiten auf:

   (a) Die Gleichheit des ersten Elementes in Regel L_3 wird nicht über eine gemeinsame Variable, sondern über ein Prädikat e ausgedrückt. Der Grund wird in Abschnitt 5.1.4 deutlich.

   (b) Für die Ableitung der Prädikate o, e existiert keine Regel. Es handelt sich also um einen partiellen Kalkül, der noch um geeignete Ableitungen erweitert werden muß. Dies ist Gegenstand von Abschnitt 5.1.2.2. □

5.1.2.1 Mehrstelligkeit

Um Coalgebren für mehrstellige Prädikate zu konstruieren, kann eine Produktkonstruktion aus der Automatentheorie verallgemeinert werden.

Definition 5.7 (Asynchrones Produkt)**.** Sei I eine endliche Menge von Indizes und für jedes $i \in I$ sei $\mathbb{C}_i = (C_i, \gamma_i)$ eine Σ_i-Coalgebra. Dann konstruiere man eine Σ-Coalgebra $\mathbb{C} = (C, \gamma)$ wie folgt, genannt das *asynchrone Produkt*:

$$\Sigma = \prod_{i \in I} \Sigma_i \qquad C = \prod_{i \in I} C_i$$

$$\gamma(x) = \Big(\Sigma_i\big(u_i(x)\big)\big(\gamma_i(\pi_i(x))\big)\Big)_{i \in I}$$

Dabei sei $u_i(x)(y)$ das Überschreiben der i-ten Komponente des Tupels x mit dem Wert y:

$$\pi_j\big(u_i(x)(y)\big) = \begin{cases} y & i = j \\ \pi_j(x) & i \neq j \end{cases}$$

Wir schreiben auch $\bigotimes_{i \in I} \mathbb{C}_i$ für $\mathbb{C}$. □

Lemma 5.8. *Es gilt:*

$$\pi_i\big(\gamma(x)\big) \asymp_\Sigma \gamma_i\big(\pi_i(x)\big)$$
$$\pi_i(x) \to_{\mathbb{C}_i} y \iff x \to_{\mathbb{C}} u_i(x)(y)$$

Die Operation γ liefert also ein Tupel von Datensätzen, die komponentenweise lokal dem Ergebnis der Einzeloperationen γ_i entsprechen. Enthält das Bild $\gamma_i(x)$ eine Referenz y, dann enthält $\gamma(x)$ stattdessen ein Tupel aus y und den übrigen Komponenten von x. Die Abbildung $\pi_i \circ \gamma$ stellt also die Operation einer Σ_i-Coalgebra dar, welche auf dem Produktraum der Träger einen Schritt in der i-ten Dimension beschreibt. Durch Komposition lassen sich diese Einzelschritte zu beliebigen Linearkombinationen zusammensetzen.

Korollar 5.9. *Das asynchrone Produkt erhält die Endlichkeit von Trägermengen und $\mathfrak{s}^*$-Umgebungen.*

Beispiel 5.10. Das asynchrone Produkt $\mathbb{N} \otimes \mathbb{N}$ besitzt folgende Operation:

$$(0,0) \mapsto (\mathsf{zero}, \mathsf{zero}) \qquad (m+1,0) \mapsto \big(\mathsf{succ}(m,0), \mathsf{zero}\big)$$
$$(0,n+1) \mapsto \big(\mathsf{zero}, \mathsf{succ}(0,n)\big) \qquad (m+1,n+1) \mapsto \big(\mathsf{succ}(m,n+1), \mathsf{succ}(m+1,n)\big)$$

□

5.1.2.2 Bedingte Kalküle

Die folgende Konstruktion läßt sich am besten mit der softwaretechnischen Metapher der *Modularisierung* beschreiben: Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein Kalkül. Dieser soll nun als ein Modul eines logischen Programms aufgefaßt werden. Dabei gilt ein Vorkommen einer Formel als Prämisse als *Verwendung*, ein Vorkommen als Konklusion dagegen als *Definition*.

Definition 5.11 (Schnittstellen)**.** Man betrachte eine Partitionierung der Formelmenge $K = K_- \uplus K_+$. Wir nennen K_+ die *Importschnittstelle* und K_- die *Exportschnittstelle* des Kalküls. Um dieser Benenennung gerecht zu werden, fordern wir, daß der Kalkül keine importierten Formeln definiert. Eine Partitionierung in Schnittstellen heißt *gültig*, wenn gilt:

$$\vdash_{\mathbb{K}} \subseteq K^* \times K_-$$ □

Definition 5.12 (Bedingung)**.** Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein Kalkül mit den gültigen Schnittstellen $K = K_- \uplus K_+$. Sei $H \subseteq K_+$ eine Formelmenge. Dann ist der *bedingte* Kalkül $\mathbb{K} \mid H$ definiert als:

$$\mathbb{K} \mid H = \big(K, \vdash_{\mathbb{K}} \cup (\{[]\} \times H)\big)$$ □

In der Modul-Metapher entspricht die Bedingung dem *Import* einer Menge von Hypothesen in einen Kalkül. Von besonderem Interesse sind hier Fälle, wo diese Formelmenge wiederum durch einen Kalkül erzeugt wurde. Hierfür sei auf den nächsten Abschnitt verwiesen.

Beispiel 5.13 (Bedingung). Der Kalkül $\mathbb{L}$ importiert Formeln der Form $\mathsf{o}(x,y), \mathsf{e}(x,y)$ und exportiert Formeln der Form $\mathsf{o}^*(\vec{l}, \vec{m})$. □

Lemma 5.14. *Bedingung distributiert über Coprodukt. Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein Coproduktkalkül. Seien K_-, K_+ gültige Schnittstellen von $\mathbb{K}$. Sei $H \subseteq K_+$ eine Formelmenge. Alle Merkmale des bedingten Kalküls $\mathbb{K} \mid H$ zerfallen in Coprodukte:*

$$\mathbb{K} = \coprod_{i \in I} \mathbb{K}_i \qquad K_- = \coprod_{i \in I} K_{i-}$$
$$H = \coprod_{i \in I} H_i \qquad K_+ = \coprod_{i \in I} K_{i+}$$

Für alle i ist $\mathbb{K}_i \mid H_i$ ebenfalls ein gültiger bedingter Kalkül.

5.1.3 Modell-Semantik

Die Semantik der logischen Sprache soll wie üblich durch die Betrachtung von Modellen beschrieben werden. Anders als bei gewöhnlichen, axiomatisierten Logiken untersuchen wir hier aber isolierte Kalküle ohne einen dahinter liegenden universellen Gültigkeitsbegriff. Daher werden auch nicht Modelle an sich, sondern Modelle eines konkreten Kalküls betrachtet. Das hat einige Vereinfachungen zur Folge:

1. Korrektheit und Vollständigkeit sind trivial gegeben.
2. Ableitbarkeit und Gültigkeit sind dasselbe.

Definition 5.15 (Modell)**.** Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein Kalkül. Eine Formelmenge $M \subseteq K$ heißt *Modell* von $\mathbb{K}$, geschrieben $M \models \mathbb{K}$, falls M invariant unter Ableitung in $\mathbb{K}$ ist:

$$M = \mathcal{P}(\vdash_{\mathbb{K}})(M^*)$$

Die Klasse aller Modelle von $\mathbb{K}$ schreiben wir als $\mathcal{M}(\mathbb{K})$. □

Lemma 5.16. *In punktweiser Notation bedeutet diese Definition:*

$$\psi \in M \iff \exists\, \varphi_1, \ldots, \varphi_n.\ \big([\varphi_1, \ldots, \varphi_n] \vdash_{\mathbb{K}} \psi \wedge \forall\, i.\ \varphi_i \in M\big)$$

Diese Definition verallgemeinert die klassische, konstruktive Definition über Ableitungsbäumen, insofern als sie keine Annahmen über die Fundiertheit von Ableitungen macht. Eine Ableitung $\vec{\varphi} \vdash \psi$ gestattet die üblichen Schlußweisen:

1. Das positive Schließen von $\vec{\varphi} \in M^*$ auf $\psi \in M$ entspricht der klassischen logischen Form des *modus ponens*.
2. Das negative Schließen von $\psi \notin M$ auf $\vec{\varphi} \notin M^*$ entspricht der klassischen logischen Form des *modus tollens*.

Beispiel 5.17. Die Modelle der oben definierten Kalküle. Diese können mit den oben erwähnten Schlußtechniken verifiziert werden. Falls ein intendiertes Modell existiert, also eines, das mit der Intuition des spezifierten Prädikats übereinstimmt, ist dieses jeweils unterstrichen:

1. Die Zahlen-Kalküle besitzen nur wenige Modelle:

$\mathbb{K}$	$\mathcal{M}(\mathbb{K})$	$\mathcal{M}(\bar{\mathbb{K}})$
$\mathbb{F}$	$\{\underline{\mathcal{N}}\}$	$\{\underline{\mathcal{N}}, \bar{\mathcal{N}}\}$
$\mathbb{G}$	$\{\underline{\emptyset}\}$	$\{\emptyset, \underline{\{\omega\}}\}$
$\mathbb{G}'$	$\{\emptyset, \mathcal{N}\}$	$\{\emptyset, \mathcal{N}, \{\omega\}, \bar{\mathcal{N}}\}$
$\mathbb{H}$	$\{\underline{\mathbb{H}_0} = \mathcal{P}(\mathsf{fin})(\mathcal{N})\}$	$\left\{\begin{array}{c}\mathbb{H}_0, \underline{\mathbb{H}_0 \cup \{\mathsf{inf}(\omega)\}},\\ \mathbb{H}_0 \cup \{\mathsf{fin}(\omega)\}, \mathbb{H}_0 \cup \{\mathsf{fin}(\omega), \mathsf{inf}(\omega)\}\end{array}\right\}$
$\mathbb{B}$	$\{\underline{\mathbb{B}_0} = \{\mathsf{g}(2n), \mathsf{u}(2n+1) \mid n \in \mathcal{N}\}\}$	$\{\mathbb{B}_0, \mathbb{B}_0 \cup \{\mathsf{g}(\omega), \mathsf{u}(\omega)\}\}$

2. Die Pingpong-Kalküle besitzen bis zu vier Modelle:

$\mathbb{K}$	$\emptyset \models \mathbb{K}$	$\{\mathsf{ping}\} \models \mathbb{K}$	$\{\mathsf{pong}\} \models \mathbb{K}$	$\{\mathsf{ping}, \mathsf{pong}\} \models \mathbb{K}$
$\mathbb{P}_\times$	×	–	–	×
$\mathbb{P}_\parallel$	×	×	×	×

3. Die Modelle der Ordnungen auf Zahlen:

$\mathbb{K}$	$\mathcal{M}(\mathbb{K})$	$\mathcal{M}(\bar{\mathbb{K}})$
$\mathbb{J}$	$\{\underline{\mathbb{J}_0 = \{(m,n) \mid m,n \in \mathcal{N}; m < n\}}\}$	$\{\underline{\mathbb{J}_1 = \mathbb{J}_0 \cup \{(m,\omega) \mid m \in \mathcal{N}\}}, \mathbb{J}_1 \cup \{(\omega,\omega)\}\}$
$\mathbb{J}'$	$\{\underline{\mathbb{J}'_0 = \{(m,n) \mid m,n \in \mathcal{N}; m \leq n\}}\}$	$\{\mathbb{J}'_1 = \mathbb{J}'_0 \cup \{(m,\omega) \mid m \in \mathcal{N}\}, \underline{\mathbb{J}'_1 \cup \{(\omega,\omega)\}}\}$

4. Die Ordnungen auf Listen besitzen im Rohzustand nur triviale Modelle, die sich für $\mathbb{L}, \bar{\mathbb{L}}$ durch das Pattern $\mathsf{o}^*\big(\mathsf{nil}, \mathsf{cons}(x, \vec{r})\big)$, für $\mathbb{L}', \bar{\mathbb{L}}'$ durch das Pattern $\mathsf{o}^*(\mathsf{nil}, \vec{s})$ erzeugen lassen.

   Durch den Import geeigneter Prädikate o, e werden die Modelle komplexer. Der Einfachheit halber betrachten wir nur endliche Listen von höchstens zwei Elementen und unendliche Listen, die sich nach einem Element wiederholen. Außerdem schränken wir den Bereich der Listenelemente auf die Zahlen $0, 1$ ein. Als Importmenge H wählen wir:

$$\{\mathsf{o}(0,1), \mathsf{e}(0,0), \mathsf{e}(1,1)\}$$

   Dann umfaßt das Prädikat o^* in allen Modellen die folgende Kette und ihren transitiven Abschluß:

$$[] < [0] < [0,0] < 0 : \ldots < [1] < [1,0] < [1,1] < 1 : \ldots$$

   In $\mathbb{L}', \bar{\mathbb{L}}'$ gilt erwartungsgemäß auch der reflexive Abschluß, zumindest auf den endlichen Listen. Für die unendlichen Listen $0 : \ldots$ und $1 : \ldots$ sind die entsprechenden Formeln unabhängig voneinander optional, somit ergeben sich kombinatorisch für $\mathbb{L}'$ und $\bar{\mathbb{L}}'$ je vier verschiedene Modelle, wobei für $\mathbb{L}'$ das kleinste und für $\bar{\mathbb{L}}'$ das größte intendiert ist. □

Die Modelle eines Kalküls $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ lassen sich im vollständigen Verband $\mathcal{P}(K)$ anordnen. Dies legt eine Charakterisierung über Fixpunktgleichungen nahe.

Definition 5.18 (Ableitungsoperator)**.** Der Operator $\Phi_{\mathbb{K}} : \mathcal{P}(K) \to \mathcal{P}(K)$ ist definiert als:

$$\Phi_{\mathbb{K}}(M) = \mathcal{P}(\vdash_{\mathbb{K}})(M^*)$$

Die Modelle von $\mathbb{K}$ sind per Konstruktion genau die Fixpunkte von $\Phi_{\mathbb{K}}$. □

Lemma 5.19. *Φ_K ist monoton.*

Beweis. Die Operatoren $\cdot^*$ und $\mathcal{P}(\vdash_{\mathbb{K}})$ sind monoton, also auch ihre Komposition. □

Korollar 5.20. *Nach einem bekannten Satz von* Tarski *ist $\mathcal{M}(\mathbb{K})$ ein vollständiger Unterverband von $\mathcal{P}(K)$. Insbesondere ist $\mathcal{M}(\mathbb{K})$ nicht leer.*

Besitzt ein Kalkül mehrere Modelle, dann muß eines ausgewählt werden, um dem Kalkül eine eindeutige Semantik zu verleihen. In der klassischen Logik stand außer Frage, daß das kleinste Modell das ausgezeichnete ist. Wie man an obigen Beispielen sehen kann, sind auch größte Modelle oft plausible Semantiken. Im Fall des komplexen Kalküls $\bar{\mathbb{H}}$ ist sogar ein mittleres Modell das angestrebte. Ein universelles Verfahren, um für eine Klasse von Kalkülen möglichst viele verschiedene Modelle auf einheitliche Weise zu entscheiden, wird Gegenstand der restlichen Kapitels sein.

5.1.3.1 Gültigkeit

Definition 5.21 (Implikation, Äquivalenz und Gültigkeit)**.** Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein Kalkül. Für Formeln $\varphi, \varphi_1, \ldots, \varphi_n, \psi \in K$ schreiben wir:

1. $\varphi_1, \ldots, \varphi_n$ *implizieren* ψ in $\mathbb{K}$ ($\varphi_1, \ldots, \varphi_n \Vdash_{\mathbb{K}} \psi$), falls $\{\varphi_1, \ldots, \varphi_n\} \subseteq M \implies \psi \in M$ für alle Modelle $M \in \mathcal{M}(\mathbb{K})$.

2. φ, ψ sind *äquivalent* in $\mathbb{K}$ ($\varphi \equiv_{\mathbb{K}} \psi$), falls $\varphi \Vdash_{\mathbb{K}} \psi$ und $\psi \Vdash_{\mathbb{K}} \varphi$.
3. φ ist *gültig* in $\mathbb{K}$ ($\Vdash_{\mathbb{K}} \varphi$), falls $\varphi \in M$ für alle Modelle $M \in \mathcal{M}(\mathbb{K})$.
4. φ ist *ungültig* in $\mathbb{K}$ ($\Vdash_{\mathbb{K}} \neg\varphi$), falls $\varphi \notin M$ für alle Modelle $M \in \mathcal{M}(\mathbb{K})$. □

Lemma 5.22. *Die Ableitungsrelation ist Spezialfall der Implikationsrelation:*

$$\varphi_1, \ldots, \varphi_n \vdash_{\mathbb{K}} \psi \implies \varphi_1, \ldots, \varphi_n \Vdash_{\mathbb{K}} \psi$$

Lemma 5.23. *Das kleinste Modell eines Kalküls enthält genau alle gültigen, das größte Modell genau alle nicht ungültigen Formeln. Da zu jedem Kalkül ein Modell existiert, gilt:*

$$\Vdash_{\mathbb{K}} \psi \implies \nVdash_{\mathbb{K}} \neg\psi \qquad\qquad \Vdash_{\mathbb{K}} \neg\psi \implies \nVdash_{\mathbb{K}} \psi$$

Definition 5.24 (Determination)**.** Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein Kalkül.

1. Eine Formel $\varphi \in K$ ist *determiniert* in $\mathbb{K}$, genau dann wenn sie gültig oder ungültig ist.
2. Der Kalkül $\mathbb{K}$ ist *determiniert*, genau dann wenn alle Formeln $\varphi \in K$ determiniert sind, er also genau ein Modell besitzt. □

Die modelltheoretische Semantik erlaubt auch eine Fortführung der Begriffe vom Coprodukt und bedingtem Kalkül.

Lemma 5.25. *Sei $(\mathbb{K}_i)_{i\in I}$ eine Familie von Kalkülen. Die Coprodukte von Modellen sind genau die Modelle des Coproduktkalküls:*

$$\{\coprod_{i\in I} M_i \mid \forall\, i.\ M_i \in \mathcal{M}(\mathbb{K}_i)\} = \mathcal{M}(\coprod_{i\in I} \mathbb{K}_i)$$

Lemma 5.26. *Sei $\mathbb{K} \mid H$ ein bedingter Kalkül. Dann gilt für alle importierten Formeln $\varphi \in K_+$:*

$$\Vdash_{\mathbb{K}} \varphi \iff \varphi \in H$$
$$\nVdash_{\mathbb{K}} \varphi \iff \varphi \notin H$$

Definition 5.27. Sei $\mathbb{K}$ ein determinierter und $\mathbb{K}'$ ein beliebiger Kalkül, so daß $K \subseteq K'_+$. Sei M das eindeutige Modell von $\mathbb{K}$. Dann schreiben wir für den bedingten Kalkül $\mathbb{K}' \mid M$ auch $\mathbb{K}' \mid \mathbb{K}$. □

In diesem Abschnitt wird ein streng konstruktiver Gültigkeitsbegriff verwendet: Eine Formel ist gültig, wenn sie in endlich vielen Schritten aus gültigen Formeln abgeleitet werden kann. Sie ist ungültig, wenn sie weder in endlich noch in unendlich vielen Schritten aus gültigen Formeln abgeleitet werden kann, also die Ableitbarkeit in endlich vielen Schritten widerlegbar ist. Ob eine nichtdeterminierte Formel in einem konkreten Modell enthalten ist oder nicht, ist daher eine Frage der Zulässigkeit unendlicher Ableitungen. Dies ist eine pragmatische Frage und kann für alle Formeln gleich, oder auch in einem gewissen Rahmen unterschiedlich beantwortet werden. Darauf wird in Abschnitt 5.4 detailliert eingegangen.

5.1.4 Abstraktion

Analog zum Begriff der abstrakten Interpretation definieren wir auch eine Klasse von Kalkülen, die kompatibel mit finaler Semantik sind. Diese Eigenschaft ist wichtig, damit Kalküle praktisch zur eindeutigen Definition von Prädikaten über den Ergebnissen primitiv corekursiver Funktionen eingesetzt werden können, da diese in der Implementierung ja nur final, das heißt bis auf Bisimilarität festgelegt sind.

Definition 5.28 (Abstrakter Kalkül)**.** Ein Kalkül $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ heißt abstrakt, wenn alle bisimilaren Formeln in K äquivalent sind. □

Satz 5.29. *Sei Σ eine Signatur. Sei $\mathbb{C} = (C, \gamma)$ eine beliebige und $\mathbb{F} = (F, \phi)$ eine finale Σ-Coalgebra. Wenn ein $\mathbb{C}$-Kalkül $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ abstrakt ist, existiert ein $\mathbb{F}$-Kalkül $\overline{\mathbb{K}} = (\overline{K}, \vdash_{\overline{\mathbb{K}}})$, so daß $\mathcal{P}(\llbracket\mathbb{C}\rrbracket) : \mathcal{M}(\mathbb{K}) \to \mathcal{M}(\overline{\mathbb{K}})$ eine bijektive Abbildung ist.*

Beweis. Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein abstrakter $\mathbb{C}$-Kalkül. Dann konstruiere man $\overline{\mathbb{K}}$ wie folgt:

$$\overline{K} = \mathcal{P}(\llbracket\mathbb{C}\rrbracket)(K)$$
$$\vdash_{\overline{\mathbb{K}}} = \mathcal{P}(\llbracket\mathbb{C}\rrbracket^* \times \llbracket\mathbb{C}\rrbracket)(\vdash_{\mathbb{K}})$$

Sei nun M ein Modell von $\mathbb{K}$, also:

$$M = \Phi_{\mathbb{K}}(M)$$

Dann ist per Konstruktion $\overline{M} = \mathcal{P}(\llbracket\mathbb{C}\rrbracket)(M)$ ein Modell von $\overline{\mathbb{K}}$:

$$\overline{M} = \Phi_{\overline{\mathbb{K}}}(\overline{M})$$

Sei nun umgekehrt $\overline{M}$ ein Modell von $\overline{\mathbb{K}}$. Dann ist ein eindeutiges Urbild M mit $\mathcal{P}(\llbracket\mathbb{C}\rrbracket)(M) = \overline{M}$ festgelegt:

1. Für alle Formeln $\overline{\varphi} \in \overline{K} \setminus \overline{M}$ gilt, daß die Urbilder φ mit $\llbracket\mathbb{C}\rrbracket(\varphi) = \overline{\varphi}$ nicht in M liegen können.
2. Für die übrigen Formeln $\overline{\varphi} \in \overline{M}$ gilt, daß von den Urbildern φ mit $\llbracket\mathbb{C}\rrbracket(\varphi) = \overline{\varphi}$ eines in M liegen muß. Da $\mathbb{K}$ per Annahme abstrakt ist, müssen auch die anderen, bisimilaren Urbilder in M liegen.

Damit sind alle Formeln $\varphi \in K$ bezüglich ihres Enthaltenseins in M festgelegt. □

Der Kalkül $\overline{\mathbb{K}}$ heißt *finale Spezifikation* von $\mathbb{K}$.

Satz 5.30. *Ein durch eine Regel mit linearen Pattern definierter Kalkül ist abstrakt.*

Beweis. Folgt direkt aus Lemma 2.130. □

Beispiel 5.31. Alle bereits bekannten Beispielkalküle sind per Konstruktion abstrakt. □

Definition 5.32. Abstraktion läßt sich auf die Kalkül-Operationen fortsetzen:

1. Die Vereinigung von Kalkülen $(\mathbb{K}_i)_{i \in I}$ heißt *abstrakte Vereinigung*, wenn alle K_i unter Bisimilarität abgeschlossen sind.
2. Die Konstruktion eines bedingten Kalküls $\mathbb{K}|H$ heißt *abstrakte Bedingung*, wenn die Exportschnittstelle K_- und die Formelmenge H unter Bisimilarität abgeschlossen sind.

Lemma 5.33. *Es gelten folgende Erhaltungseigenschaften:*

1. *Die abstrakte Vereinigung abstrakter Kalküle ergibt einen abstrakten Kalkül.*
2. *Das Coprodukt abstrakter Kalküle ergibt stets einen abstrakten Kalkül.*
3. *Die abstrakte Bedingung eines abstrakten Kalküls ergibt einen abstrakten Kalkül.*

In den folgenden Abschnitten sollen nun Klassen von abstrakten Kalkülen anhand der Komplexität ihrer finalen Spezifikation untersucht werden. Dabei werden die bereits ausführlich diskutierten Stufen *initial*, *rational* und *final* unterschieden.

5.2 Algebraische Kalküle

Von primärem Interesse als logische Berechnungsmodelle sind Kalküle, die aufgrund ihrer Struktur bereits einen Entscheidungsalgorithmus für ein intendiertes Modell festlegen, die also ein geschlossenes logisches Programm darstellen.

Definition 5.34 (Primitiv Rekursiver Kalkül)**.** Sei Σ eine Signatur. Sei $\mathbb{C}$ eine beliebige und $\mathbb{F}$ eine finale Σ-Coalgebra. Damit ist $\mathbb{F}_0^{-1}$ eine initiale Σ-Algebra. Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein abstrakter $\mathbb{C}$-Kalkül. $\mathbb{K}$ heißt *primitiv rekursiv*, falls gilt:

1. Die finale Spezifikation ist initial: $\overline{K} \subseteq F_0$.
2. Für alle Ableitungen in $\overline{\mathbb{K}}$ gilt, daß die Prämissen stets echte Teilterme der Konklusion sind:

$$[\varphi_1, \ldots, \varphi_n] \vdash_{\overline{\mathbb{K}}} \psi \implies \{\varphi_1, \ldots, \varphi_n\} \subseteq \mathfrak{s}^+_{\mathbb{F}_0}(\psi) \qquad \square$$

Lemma 5.35. *Da Homomorphismen Zyklen erhalten, ist $\mathbb{K}$ ebenfalls zyklenfrei.*

Beispiel 5.36. Von den Beispielkalkülen sind jeweils die initialen Varianten $\mathbb{F}$, $\mathbb{G}$, $\mathbb{H}$, $\mathbb{B}$, $\mathbb{J}$ und $\mathbb{L}$ primitiv rekursiv. Daneben gilt:

1. $\bar{\mathbb{F}}$ und $\bar{\mathbb{G}}$ sind nicht primitiv rekursiv, da sie eine zyklische Ableitung $[\omega] \vdash \omega$ enthalten. Dasselbe gilt für $\bar{\mathbb{H}}$ wegen des Zyklus $[\mathsf{g}(\omega)] \vdash \mathsf{u}(\omega)$ und $[\mathsf{u}(\omega)] \vdash \mathsf{g}(\omega)$. Auch $\bar{\mathbb{J}}$, $\bar{\mathbb{J}}'$, $\bar{\mathbb{L}}$ und $\bar{\mathbb{L}}'$ haben Zyklen.
2. $\mathbb{G}'$ und $\bar{\mathbb{G}}'$ sind nicht primitiv rekursiv, da sie unendliche Ketten $[n+1] \vdash n$ enthalten.
3. In $\mathbb{P}_\times$ und $\mathbb{P}_\parallel$ sind die Prämissen nicht Teilterme der Konklusion. Abhilfe würde hier eine andere, wenn auch sehr artifizielle Coalgebra schaffen:

$$\begin{aligned} \Sigma &= \mathcal{C}_P \times \mathcal{I} \times \mathcal{I} & \gamma(\mathsf{ping}) &= (\mathsf{ping}, \mathsf{ping}, \mathsf{pong}) \\ C &= P & \gamma(\mathsf{pong}) &= (\mathsf{pong}, \mathsf{ping}, \mathsf{pong}) \end{aligned}$$

   Allerdings besitzt diese notwendigerweise Zyklen. $\square$

Satz 5.37. *Coprodukt und abstrakte Bedingung erhalten die primitive Rekursivität.*

Beweis. Der Ergebniskalkül ist per Konstruktion abstrakt. Bleibt noch die Zyklenfreiheit und die Erreichbarkeit der Prämissen zu zeigen.

Coprodukt analog zur Auszeichnung initialer Typen in einer mehrsortigen Signatur. Sei für alle $i \in I$ der Kalkül $\mathbb{K}_i$ primitiv rekursiv. Sei $\mathbb{F}_i$ eine entsprechende finale Σ_i-Coalgebra und $\mathbb{A}_i = (A_i, \alpha_i)$ die zu $\mathbb{F}_{i,0}$ inverse initiale Σ_i-Algebra. Bilde das Coprodukt der Signaturen $\Sigma = \coprod(\Sigma_i)$ und eine initiale Σ-Algebra $\mathbb{A} = (A, \alpha)$. Dann kann zu jedem $\mathbb{A}_i$ eine Unteralgebra $\mathbb{A}'_i \subseteqq \mathbb{A}$ als Bild eines Katamorphismus $(\!|\mathbb{A}^*_I|\!)$ erzeugt werden, indem $\mathbb{A}$ kurzerhand zu einer Σ_i-Algebra $\mathbb{A}^*_i = (A, \alpha \circ \iota_i)$ vereinfacht wird:

$$\begin{array}{ccc} \Sigma_i(A_i) & \xrightarrow{\Sigma_i\left((\!|\mathbb{A}^*_i|\!)\right)} & \Sigma_i(A) \\ \Big\downarrow{\scriptstyle \alpha_i} & & \Big\downarrow{\scriptstyle \iota_i} \\ & & \Sigma(A) \\ & & \Big\downarrow{\scriptstyle \alpha} \\ A_i & \xrightarrow[(\!|\mathbb{A}^*_i|\!)]{} & A \end{array}$$

Dabei bleibt die Teiltermbeziehung für alle $i \in I$ erhalten:

$$\varphi \in \mathfrak{p}_{\mathbb{A}_i}(\psi) \iff \iota_i(\varphi) \in \mathfrak{p}_{\mathbb{A}}\big(\iota_i(\psi)\big)$$

```
type Pred α = α → Bool
type Calc α = α → [[α]]
type Res    = Eq α ⇒ Calc α → Pred α
```

Programm 5.1: Typdefinitionen für Entscheidungsalgorithmen

Damit folgt aus der primitiven Rekursivität von $\mathbb{K}_i$:

$$\big[\iota_i(\varphi_1),\ldots,\iota_i(\varphi_n)\big] \vdash_{\mathbb{K}} \psi \implies \big\{\iota_i(\varphi_1),\ldots,\iota_i(\varphi_n)\big\} \subseteq \mathfrak{p}_{\mathbb{A}}\big(\iota_i(\psi)\big)$$

Das sind aber bereits alle Ableitungen in $\mathbb{K} = \coprod(\mathbb{K}_i)$. Durch Invertierung von $\mathbb{A}$ zu $\mathbb{F}_0$ folgt die Behauptung.

Für bedingte Kalküle gelten die Behauptungen trivialerweise, da die hinzugefügten Regeln keine Prämissen besitzen. □

Auf primitiv rekursive Kalküle ist eine Variante des klassischen Resolutionsverfahrens für HORN-Formeln anwendbar, wie es auch zur Abarbeitung logischer Programme verwendet wird.

Voraussetzung ist die Berechenbarkeit der endlichen Urbildmenge $\overline{\mathcal{P}}(\vdash_{\mathbb{K}})\big(\{\psi\}\big)$ zu jeder Formel $\psi \in K$. Da die Elemente dieser Menge sequentiell abgearbeitet werden sollen, fordern wir stattdessen eine berechenbare Funktion $\kappa : K \to (K^*)^*$, so daß gilt:

$$\kappa(\psi) = [\vec{\varphi}_1,\ldots,\vec{\varphi}_n] \Longleftrightarrow \overline{\mathcal{P}}(\vdash_{\mathbb{K}})\big(\{\psi\}\big) = \{\vec{\varphi}'_1,\ldots,\vec{\varphi}'_n\}$$

Die Reihenfolge oder Injektivität der Aufzählung wird dabei keine Rolle für die Korrektheit des Verfahrens spielen.

Um die Notation möglichst ähnlich zum vorigen Kapitel zu halten, identifizieren wir Mengen mit ihren charakteristischen Abbildungen. Es gilt: die Menge ist entscheidbar, genau dann wenn die Abbildung berechenbar ist. Außerdem wird zunächst eine iterative Formulierung zur Klärung der operationalen Grundlagen präsentiert, während die Weiterentwicklung zum zyklentauglichen Verfahren anhand einer abstrakteren, rekursiven Formulierung erfolgt.

5.2.1 Iterative Formulierung

Definition 5.38 (Iterativer Entscheidungsalgorithmus Res_1)**.** Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein primitiv rekursiver Kalkül mit Ableitungsfunktion κ. Die Gültigkeit einer Formel $\psi \in K$ läßt sich mittels iterativer Tiefensuche entscheiden. Der Algorithmus $\mathsf{Res}_1(\mathbb{K})$ verwaltet dazu eine zweidimensionale Liste $S \in (K^*)^*$ von atomaren Aussagen, welche als disjunktive Normalform einer zur Eingabe ψ äquivalenten, komplexen Formel vorgestellt werden kann.

1. Im Startzustand gilt $S = \big[[\psi]\big]$. Der Nachfolgezustand S' berechnet sich aus S in jedem Schritt wie folgt:
2. Ist $S = \big[\big]$, dann gilt bereits $\mathsf{Res}_1(\mathbb{K})(\psi) = f$.
3. Andernfalls zerlege $S = \vec{\sigma} : T$.
   (a) Ist $\vec{\sigma} = []$, dann gilt bereits $\mathsf{Res}_1(\mathbb{K})(\psi) = w$.
   (b) Andernfalls zerlege $\vec{\sigma} = \varrho : \vec{\sigma}'$. Bestimme $\big[\vec{\varphi}_1,\ldots,\vec{\varphi}_k\big] = \kappa(\varrho)$. Iteriere mit $S' = \big[\vec{\varphi}_1 +\!\!+ \vec{\sigma}', \ldots, \vec{\varphi}_k +\!\!+ \vec{\sigma}'\big] +\!\!+ T$.

```
res1      :: Res
res1 k p = loop [[p]]
  where
    loop []           = False
    loop ([] : _)     = True
    loop ((r : s) : t) = loop (map (++ s) (k r) ++ t)
```

Programm 5.2: Iterativer Entscheidungsalgorithmus

Die Ableitungsfunktion κ zu einem $\mathbb{C}$-Kalkül $\mathbb{K}$ kann auch als Operation einer $\big(\mathcal{I} \to (\mathcal{I}^*)^*\big)$-Interpretation über $\mathcal{C}$ verstanden werden. Die Umgebung $\mathfrak{s}^*_{\mathbb{K}}(\psi)$ einer Formel ψ beschränkt den Suchraum für das Entscheiden von ψ. Anders als bei der corekursiven Funktionsberechnung müssen aber im Allgemeinen nicht alle erreichbaren Elemente auch traversiert werden, da das Ergebnis einer mehrstelligen Kon- oder Disjunktion bereits nach der Auswertung eines Teils der Argumente feststehen kann. Im Algorithmus Res_1 läßt sich dieser Effekt genau lokalisieren:

1. Im Fall $S = [] : T$ (leere Konjunktion) werden die restlichen Klauseln in T ignoriert.
2. Im Fall $\kappa(\varrho) = []$ (leere Disjunktion) werden die restlichen Formeln in $\vec{\sigma}'$ ignoriert.

Satz 5.39. *Sei $\mathbb{K}$ ein primitiv rekursiver Kalkül. Der Algorithmus $\mathsf{Res}_1(\mathbb{K})$ terminiert.*

Beweis. Durch Wahl einer geeigneten Terminationsfunktion $W : (K^*)^* \to \mathcal{N}$.

$$\begin{aligned} W\big([\vec{\varphi}_1, \ldots, \vec{\varphi}_n]\big) &= \sum_{i=1}^{n} \vec{w}(\vec{\varphi}_i) \\ \vec{w}\big([\varphi_1, \ldots, \varphi_m]\big) &= \prod_{i=1}^{n} w(\varphi_i) && > 0 \\ w(\psi) &= 1 + \sum_{\vec{\varphi} \vdash_K \psi} \vec{w}(\vec{\varphi}) && > 0 \end{aligned}$$

$w(\varphi)$ ist wegen der Zyklenfreiheit wohldefiniert. Es gilt:

$$\begin{aligned} S &= (\varrho : \vec{\sigma}') : T & S' &= [\vec{\varphi}_1 +\!\!+ \vec{\sigma}', \ldots, \vec{\varphi}_n +\!\!+ \vec{\sigma}'] +\!\!+ T \\ W(S) &= w(\varrho) \cdot \vec{w}(\vec{\sigma}') + W(T) & W(S') &= \left(\sum_{i=1}^{n} \vec{w}(\vec{\varphi}_i) \cdot \vec{w}(\vec{\sigma}')\right) + W(T) \\ &= \left(1 + \sum_{i=1}^{n} \vec{w}(\vec{\varphi}_i)\right) \cdot \vec{w}(\vec{\sigma}') + W(T) \qquad > & &= \left(\sum_{i=1}^{n} \vec{w}(\vec{\varphi}_i)\right) \cdot \vec{w}(\vec{\sigma}') + W(T) \end{aligned}$$

□

Satz 5.40. *Sei $\mathbb{K}$ ein primitiv rekursiver Kalkül. Der Algorithmus $\mathsf{Res}_1(\mathbb{K})$ ist partiell korrekt.*

$$\mathsf{Res}_1(\mathbb{K})(\psi) = \begin{cases} w & \Vdash_{\mathbb{K}} \psi \\ f & \Vdash_{\mathbb{K}} \neg\psi \end{cases}$$

Beweis. Man interpretiere den Zustand S wie oben angedeutet:

$$\begin{aligned} Q([\vec{\varphi}_1, \ldots, \vec{\varphi}_n]) &\Longleftrightarrow \exists\, i.\ q(\vec{\varphi}_i) \\ q(\varphi_1, \ldots, \varphi_n) &\Longleftrightarrow \forall\, i.\ \Vdash_K \varphi_i \end{aligned}$$

```
res2  :: Res
res2 k = recur
  where
    recur p  =  dnf recur (k p)
dnf   = any ∘ all
```

Programm 5.3: Rekursiver Entscheidungsalgorithmus ohne Zyklenerkennung

Nach Lemma 5.16 gilt $\Vdash_{\mathbb{K}} \psi \Longleftrightarrow Q\big(k(\psi)\big)$.

Dann zeige man, daß für jeden Übergang $S \to S'$ des Algorithmus $Q(S) \Longleftrightarrow Q(S')$ gilt: Sei $S = (\varrho : \vec{\sigma}) : T$. Dann ergeben sich zwei Fälle:

1. Es gilt $q(\varrho : \vec{\sigma})$. Also gilt $\Vdash_{\mathbb{K}} \varrho$ und es existiert eine Klausel $\vec{\varphi}_i$ mit $\vec{\varphi}_i \vdash_{\mathbb{K}} \varrho$, so daß $q(\vec{\varphi}_i)$ gilt. Damit gilt auch $q(\vec{\varphi}_i +\!\!+ \vec{\sigma})$. Also gelten sowohl $Q(S)$ als auch $Q(S')$.
2. Andernfalls gilt $Q(S) \Longleftrightarrow Q(T)$. Für keine Klausel $\vec{\varphi}_i$ mit $\vec{\varphi}_i \vdash_{\mathbb{K}} \varrho$ gilt $q(\vec{\varphi})$, also auch nicht $Q\big([\vec{\varphi}_1 +\!\!+ \vec{\sigma}, \ldots, \vec{\varphi}_n +\!\!+ \vec{\sigma}]\big)$. Damit gilt $Q(T) \Longleftrightarrow Q(S')$.

Man betrachte nun die möglichen Endzustände für den Ablauf von $\mathsf{Res}_1(\mathbb{K})(\psi)$:

1. $\big[[\psi]\big] \to^* \big[\big]$. Also gilt $\Vdash_{\mathbb{K}} \psi \Longleftrightarrow \bot$, und $\mathsf{Res}_1(\mathbb{K})(\psi) = f$ ist korrekt.
2. $\big[[\psi]\big] \to^* [] : T$. Also gilt $\Vdash_{\mathbb{K}} \psi \Longleftrightarrow \top$, und $\mathsf{Res}_1(\mathbb{K})(\psi) = w$ ist korrekt. □

Korollar 5.41. *Jeder primitiv rekursive Kalkül ist determiniert.*

Aus dem Beweis der partiellen Korrektheit läßt sich ein alternativer Algorithmus ableiten, der zwar komplexere formale Mittel zu seiner Spezifikation benötigt, aber wesentlich näher an einer funktionalen Implementierung liegt, und die ineffiziente Vervielfachung der Restklausel $\vec{\sigma}'$ vermeidet, die nur der Wahrung der disjunktiven Normalform dient.

5.2.2 Rekursive Formulierung

Definition 5.42 (Rekursiver Entscheidungsalgorithmus Res_2)**.** Um die Gültigkeit einer Formel $\psi \in K$ in einem primitiv rekursiven Kalkül $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ mit Ableitungsfunktion κ zu entscheiden, suche in $[\vec{\varphi}_1, \ldots, \vec{\varphi}_k] = \kappa(\psi)$ ein Element $\vec{\varphi}_i = [\varphi_{i,1}, \ldots, \varphi_{i,m}]$, so daß sich rekursiv $\mathsf{Res}_2(\mathbb{K})(\varphi_{i,j}) = w$ für alle j ergibt. Ist die Suche erfolgreich, gilt $\mathsf{Res}_2(\mathbb{K})(\psi) = w$, sonst $\mathsf{Res}_2(\mathbb{K})(\psi) = f$. □

Satz 5.43. *Die Algorithmen* Res_1 *und* Res_2 *sind äquivalent.*

Beweis. Es gelten folgende Invarianten:

1. $\mathsf{Res}_2(\mathbb{K})(\varrho) = w$ gilt, genau dann wenn jeder Zustand $S = \big[[\varrho] +\!\!+ \vec{\sigma}'\big] +\!\!+ T$ unter $\mathsf{Res}_1(\mathbb{K})$ in endlich vielen Schritten in einen Nachzustand $[\vec{\sigma}'] +\!\!+ T' +\!\!+ T$ übergeht.
2. $\mathsf{Res}_2(\mathbb{K})(\varrho) = f$ gilt, genau dann wenn jeder Zustand $S = \big[[\varrho] +\!\!+ \vec{\sigma}'\big] +\!\!+ T$ unter $\mathsf{Res}_1(\mathbb{K})$ in endlich vielen Schritten in den Nachzustand T übergeht.

Der Beweis erfolgt induktiv über $\mathbb{F}_0$. Die Argumentation entspricht dem Beweis zu Satz 5.40. Durch Spezialisierung auf $\vec{\sigma}' = []$ und $T = \big[\big]$ folgt die Behauptung. □

In einer algebraischen Sichtweise wäre die hier vorgenommene Einschränkung auf die initiale Algebra als Träger eines abstrakten Kalküls durchaus natürlich. In einer coalgebraischen Sichtweise dagegen stellt sich die Frage, inwieweit diese Einschränkung fallengelassen werden kann.

5.3 Coalgebraische Kalküle

Definition 5.44 (Primitiv Corekursiver Kalkül)**.** Sei Σ eine Signatur. Sei $\mathbb{C}$ eine Σ-Coalgebra. Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein $\mathbb{C}$-Kalkül. $\mathbb{K}$ heißt *primitiv corekursiv*, falls für alle Ableitungen in $\mathbb{K}$ gilt, daß die Prämissen stets von der Konklusion aus erreichbar sind:

$$\varphi_1, \ldots, \varphi_n \vdash_{\mathbb{K}} \psi \implies \{\varphi_1, \ldots, \varphi_n\} \subseteq \mathfrak{s}^+_{\mathbb{C}}(\psi) \qquad \square$$

Ein primitiv corekursiver Kalkül muß nicht aus technischen Gründen abstrakt sein. Die in Abschnitt 5.1.4 diskutierten semantischen Gründe sind aber gültig.

Beispiel 5.45. Für die bekannten Beispielkalküle gilt:

1. $\mathbb{P}_\times$ und $\mathbb{P}_\parallel$ lassen sich, wie in Beispiel 5.36 gezeigt, zu primitiv corekursiven Kalkülen umformen.
2. Auch für $\mathbb{G}'$ und $\bar{\mathbb{G}}'$ existiert eine alternative Coalgebra mit $\Sigma = \mathcal{I}$ und $\gamma = \mathit{succ}$, über welcher sie primitiv corekursiv sind.
3. Alle anderen Beispielkalküle sind offensichtlich primitiv corekursiv. $\square$

Satz 5.46 (Konstruktivität)**.** *Vereinigung, Coprodukt und Bedingung erhalten die primitive Corekursivität.*

Beweis. Zeige die Erreichbarkeit der Prämissen. Für die Vereinigung gilt diese trivial, da die Prämissen pro Konklusion vereinigt werden. Ebenso für die Bedingung, da hinzugefügte Ableitungen keine Prämissen besitzen.

Im Coproduktfall erhält das freie Coprodukt der Coalgebren die Erreichbarkeit: Sei für jedes $i \in I$ Σ_I eine Signatur, $\mathbb{C}_i$ eine Σ_i-Coalgebra und $\mathbb{K}_i$ ein $\mathbb{C}_i$-Kalkül. Sei $\mathbb{C} = \coprod_{i \in I} \mathbb{C}_i$. Dann gilt:

$$\psi \to_{\mathbb{C}} \varphi \iff \exists\, i \in I.\ \psi = \iota_i(\psi') \wedge \varphi = \iota_i(\varphi') \wedge \psi' \to_{\mathbb{C}_i} \varphi'$$

Von rechts nach links folgt die Behauptung. $\square$

Primitive Corekursivität eignet sich allerdings nicht direkt als Grundlage eines Entscheidungsverfahrens, da der zu einer Formel $\psi \in K$ gehörige Suchraum $\mathfrak{s}^*_{\mathbb{C}}(\psi)$ im Allgemeinen nicht endlich ist.

Definition 5.47 (Rational Corekursiver Kalkül)**.** Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein primitiv corekursiver $\mathbb{C}$-Kalkül. $\mathbb{K}$ heißt *rational corekursiv*, falls für jede Formel $\psi \in K$ die Umgebung $\mathfrak{s}^*_{\mathbb{C}}(\psi)$ endlich ist.

Lemma 5.48. *Jeder primitiv rekursive Kalkül ist rational corekursiv. Jeder rational corekursive Kalkül hat eine rationale Spezifikation:*

$$\overline{K} \subseteq F_*$$

Beispiel 5.49. Der Umkehrschluß, daß jeder Kalkül mit rationaler Spezifikation rational corekursiv sei, gilt nicht: Man betrachte die $\mathcal{I}$-Coalgebra $\mathbb{O} = (\mathcal{N}, \mathit{succ})$, über welcher $\mathbb{G}'$ primitv corekursiv ist. Da finale $\mathcal{I}$-Coalgebren einelementig sind, ist jede finale Spezifikation zu dieser Signatur rational. Die Umgebungen aller Elemente von $\mathbb{O}$ sind aber unendlich. $\square$

Satz 5.50 (Konstruktivität)**.** *Abstrakte Vereinigung, Coprodukt und abstrakte Bedingung erhalten die rationale Corekursivität.*

Beweis. Der Ergebniskalkül ist nach Satz 5.46 bereits primitiv corekursiv. Bleibt also nur die Rationalität zu zeigen, also die Abwesenheit unendlicher Umgebungen. Die Kardinalität von Umgebungen bleibt aber unter Vereinigung und Coprodukt von Coalgebren und unter Bedingung erhalten. $\square$

Die folgende Eigenschaft zeigt, daß rationale Corekursivität gegenüber primitiver Rekursivität eine Verallgemeinerung von durchaus praktischer Relevanz darstellt.

```
res_2a    :: Res
res_2a k = recur []
  where
    recur  s p =  let recur' =  recur (p : s)
                  in  dnf  recur' (k p)
```

Programm 5.4: Rekursiver Entscheidungsalgorithmus mit Zyklenprotokollierung

Korollar 5.51. *Ein als Vereinigung von Pattern-Regeln konstruierter Kalkül ist rational, wenn alle seine Regeln rational sind, sprich jeweils für sich genommen einen rationalen Kalkül erzeugen.*

Die im vorigen Abschnitt vorgestellen Algorithmen sind auf rational corekursive Kalküle nicht effektiv anwendbar, da sie für zyklische Formeln im Allgemeinen nicht terminieren. Analog zum Übergang von der rekursiven zur corekursiven Funktionsberechnung muß eine zusätzliche Protokollierung bereits besuchter Elemente erfolgen, um Zyklen behandeln zu können.

Das Entscheiden von Modellen rational corekursiver Kalküle ist für den gesamten Ansatz dieser Arbeit von zentraler Bedeutung: Ein primitiv corekursiver Kalkül kann stets auf seinen rationalen Anteil eingeschränkt werden. Für die konkrete Anwendung zur Beschreibung von Entscheidungsverfahren auf Speicherzuständen gilt sogar: Jeder Kalkül, der durch ein System von Regeln aus linearen Pattern über einem endlichen Speicherzustand erzeugt wird, ist rational corekursiv. Damit ist ein Entscheidungsalgorithmus vollständig für das mit den Mitteln von Kapitel 4 erzeugte Datenuniversum.

```
type ERes = Eq α ⇒ Calc α → Pred α → Pred α

res3          :: ERes
res3 k e      = recur []
  where
    recur s p = let recur' = recur (p : s)
                in if any (== p) s then
                     e p
                   else
                     dnf recur' (k p)
```

Programm 5.5: Rekursiver Entscheidungsalgorithmus mit Zyklenbehandlung

5.3.1 Zyklenbehandlung

Anders als im Falle der zyklischen Funktionsberechnung ist es hier mit der Erkennung von Zyklen nicht getan: Zur Behandlung eines erkannten Zyklus muß dieser auf einen Wahrheitswert abgebildet, also die zyklische Ableitung entweder gestattet oder zurückgewiesen werden. Dabei ist nicht offensichtlich

1. welche der beiden Alternativen zum intendierten Modell führt,
2. ob eine gegebene Entscheidungsstrategie überhaupt ein Modell generiert.

Wir stellen zunächst einen Algorithmus vor, der diese Fragen offen läßt, um uns ihnen im Anschluß wieder zuzuwenden.

Definition 5.52. Der rekursive, zyklenbehandelnde Algorithmus $\mathsf{Res}_3(\mathbb{K}, E)$ wird neben einem Kalkül $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ mit einer entscheidbaren Formelmenge $E \subseteq K$ parametrisiert. Die Menge E nennen wir die *Erwartung* an $\mathbb{K}$.

Als Eingabe erhält der Algorithmus eine Inkarnationsliste $\vec{s} \in K^*$ und eine Formel $\psi \in K$. Es werden zwei Fälle unterschieden:

1. Es liegt ein Zyklus vor: $\psi \in \operatorname{codom} \vec{s}$. Dann setze
$$\mathsf{Res}_3(\mathbb{K}, E)(\vec{s}, \psi) = \begin{cases} w & \psi \in E \\ f & \psi \notin E \end{cases}$$
2. Es liegt kein Zyklus vor: $\psi \notin \operatorname{codom} \vec{s}$. Dann fahre fort wie Algorithmus Res_2, aber mit Zyklenprotokollierung. Suche dazu in $[\vec{\varphi}_1, \ldots, \vec{\varphi}_k] = \kappa(\psi)$ ein Element $\vec{\varphi}_i = [\varphi_{i1}, \ldots, \varphi_{im}]$, so daß sich rekursiv $\mathsf{Res}_3(\mathbb{K}, E)(\psi : \vec{s}, \varphi_{ij}) = w$ für alle j ergibt. Ist die Suche erfolgreich, gilt $\mathsf{Res}_3(\mathbb{K}, E)(\vec{s}, \psi) = w$, sonst $\mathsf{Res}_3(\mathbb{K}, E)(\vec{s}, \psi) = f$.

Die Menge $M(\mathbb{K}, E) = \{\psi \in K \mid \mathsf{Res}_3(\mathbb{K}, E)([], \psi) = w\}$ heißt von $\mathbb{K}$ und E generiert. □

Satz 5.53 (Termination)**.** *Sei $\mathbb{K}$ ein rational corekursiver $\mathbb{C}$-Kalkül und E eine Erwartung an $\mathbb{K}$. Der Algorithmus $\mathsf{Res}_3(\mathbb{K}, E)$ terminiert.*

Beweis. Analog zu Algorithmus Ana_3. Man betrachte die Rekurrenzrelation $\rightsquigarrow$ des Algorithmus Res_3 auf seinen Argumenten $\vec{s}$ und ψ. Es gelte $(\vec{s}, \psi) \rightsquigarrow (\vec{s}', \psi')$. Dann folgt:

$$\vec{s}' = \psi : \vec{s} \qquad \psi \to_{\mathbb{C}} \psi'$$
$$\operatorname{codom} s \subset \operatorname{codom} s' \qquad \mathfrak{s}^*_{\mathbb{C}}(\psi') \subseteq \mathfrak{s}^*_{\mathbb{C}}(\psi)$$

Es sind zwei Fälle zu unterscheiden:

1. Azyklischer Fall $\psi' \nrightarrow_{\mathbb{C}}^{+} \psi$. Dann gilt $\psi \notin \mathfrak{s}_{\mathbb{C}}^{*}(\psi') \subset \mathfrak{s}_{\mathbb{C}}^{*}(\psi)$.
2. Zyklischer Fall $\psi' \rightarrow_{\mathbb{C}}^{+} \psi$. Dann gilt $\psi \in \mathfrak{s}_{\mathbb{C}}^{*}(\psi') = \mathfrak{s}_{\mathbb{C}}^{*}(\psi)$.

Definiert man also die unmarkierte Umgebung $W(\vec{s}, \psi) = \mathfrak{s}_{\mathbb{C}}^{*}(\psi) \setminus \operatorname{codom} \vec{s}$, dann gilt:

$$W(\vec{s}', \psi') \subset W(\vec{s}, \psi)$$

Da die Umgebung $\mathfrak{s}_{\mathbb{C}}^{*}(\psi)$ stets endlich ist, ist $w(\vec{s}, \psi) = |W(\vec{s}, \psi)|$ eine Terminationsfunktion. □

Satz 5.54 (Idempotenz)**.** *Sei* $\mathbb{K}$ *ein rational corekursiver* $\mathbb{C}$*-Kalkül und* M_0 *ein Modell von* $\mathbb{K}$*. Dann gilt:*

$$M(\mathbb{K}, M_0) = M_0$$

Beweis. Zeige für beliebigen Stack $\vec{s}$, daß $\mathsf{Res}_3(\mathbb{K}, M_0)(\vec{s}, \psi) = w$ gilt, genau dann wenn $\psi \in M$. Beweis durch Induktion in der Rekursionstiefe.

1. $\psi \in \operatorname{codom} \vec{s}$. Dann gilt die Behauptung unmittelbar.
2. $\psi \notin \operatorname{codom} \vec{s}$. Sei $K_\psi = \mathfrak{s}_{\mathbb{C}}^{+}(\psi)$. Für alle $\varphi \in K_\psi$ gelte per Induktionsannahme:

   $$\mathsf{Res}_3(\mathbb{K}, M_0)(\psi : \vec{s}, \varphi) = w \iff \varphi \in M_0$$

   Per Definition des Algorithmus gilt dann:

   $$\mathsf{Res}_3(\mathbb{K}, M_0)(\vec{s}, \psi) = w \iff \psi \in \Phi_{\mathbb{K}}(M_0 \cap K_\psi)$$

   Da $\mathbb{K}$ insbesondere primitiv corekursiv ist, ist K_ψ bereits der ganze Suchraum für ψ. Also gilt:

   $$\psi \in \Phi_{\mathbb{K}}(M_0 \cap K_\psi) \iff \psi \in \Phi_{\mathbb{K}}(M_0) = M_0$$ □

Korollar 5.55. *Der Algorithmus* Res_3 *ist vollständig: Alle überhaupt entscheidbaren Modelle eines rational corekursiven Kalküls lassen sich auch mit diesem Algorithmus entscheiden.*

Dieses Ergebnis ist offensichtlich unter praktischen Gesichtspunkten unbefriedigend, da es nicht gestattet, eine Entscheidungsprozedur für ein Modell aus einer *einfacheren* Entscheidungsprozedur für eine Erwartung zu konstruieren.

Die Korrektheit des Algorithmus ist dagegen nicht einmal eine absolute Eigenschaft, da ja nicht geklärt ist, welches Modell eines Kalküls der Algorithmus entscheiden soll. Wie oben festgestellt, muß noch untersucht werden, welche Belegungen des Parameters E zulässig sind, und welche Modelle jeweils resultieren. Damit wird sich der folgende Abschnitt beschäftigen. Auch die offene Frage des vorigen Absatzes, inwieweit sich Modelle durch einfach entscheidbare Erwartungen spezifizieren lassen, wird beantwortet werden.

5.4 Erwartungen

Definition 5.56 (Konsistenz)**.** Eine Erwartung E an einen Kalkül $\mathbb{K}$ heißt *konsistent*, genau dann wenn die generierte Menge $M(\mathbb{K}, E)$ ein Modell von $\mathbb{K}$ ist. E heißt ferner *stark konsistent*, genau dann wenn sie auch für alle bedingten Kalküle $\mathbb{K} \mid H$ konsistent ist. □

Die Tatsache, daß diese Definition überhaupt angegeben wird, läßt bereits vermuten, daß nicht alle Erwartungen in diesem Sinne konsistent sind.

Beispiel 5.57 (Inkonsistente Erwartung). Man betrachte den Kalkül $\mathbb{K} = PP_\times$ und die Erwartung $E = \{\mathsf{ping}\}$. Es gilt:

$$\begin{aligned} &\mathsf{Res}_3(\mathbb{K}, E)([], \mathsf{ping}) && \mathsf{Res}_3(\mathbb{K}, E)([], \mathsf{pong}) \\ &= \mathsf{Res}_3(\mathbb{K}, E)([\mathsf{ping}], \mathsf{pong}) && = \mathsf{Res}_3(\mathbb{K}, E)([\mathsf{pong}], \mathsf{ping}) \\ &= \mathsf{Res}_3(\mathbb{K}, E)([\mathsf{pong}, \mathsf{ping}], \mathsf{ping}) && = \mathsf{Res}_3(\mathbb{K}, E)([\mathsf{ping}, \mathsf{pong}], \mathsf{pong}) \\ &= w && = f \end{aligned}$$

$M(\mathbb{K}, E)$ ist kein Modell von $\mathbb{K}$. □

Es ist im Allgemeinen schwierig, die Konsistenz einer Erwartung nachzuweisen, oder eine einfach entscheidbare Erwartung für ein intendiertes Modell zu finden. Statt eines universellen Verfahrens bieten wir hier eine partielle, konstruktive Antwort, indem wir zeigen, daß

1. sich ausgezeichnete, oft intendierte Modelle durch triviale Erwartungen generieren lassen,
2. die Konsistenz von Erwartungen durch Kalkülkonstruktionen kompositional erhalten wird.

Legitimiert wird der Ansatz durch die Vermutung, daß alle Modelle von Interesse durch triviale Erwartungen und die bekannten Kalkülkonstruktionen erzeugt werden.

Satz 5.58. *Sei $\mathbb{K} = (K, \vdash_\mathbb{K})$ ein rational corekursiver Kalkül. Die beiden trivialen Erwartungen $\emptyset$ und K sind konsistent.*

Beweis. Zu zeigen ist für $E \in \{\emptyset, K\}$:

$$\begin{aligned} M(\mathbb{K}, E) &= \Phi_\mathbb{K}\big(M(\mathbb{K}, E)\big) \\ \mathsf{Res}_3(\mathbb{K}, E)([], \psi) = w &\iff \psi \in \Phi_\mathbb{K}\big(M(K, E)\big) \\ &\iff \exists\, \vec{\varphi} \in M(K, E)^*.\ \vec{\varphi} \vdash_\mathbb{K} \psi \\ &\iff \exists\, \varphi_1, \ldots, \varphi_n.\ [\varphi_1, \ldots, \varphi_n] \vdash_\mathbb{K} \psi \wedge \forall\, i.\ \mathsf{Res}_3(\mathbb{K}, E)([], \varphi_i) = w \qquad (1) \end{aligned}$$

Aus der Definition des Algorithmus folgt unmittelbar:

$$\mathsf{Res}_3(\mathbb{K}, E)([], \psi) = w \iff \exists\, \varphi_1, \ldots, \varphi_n.\ [\varphi_1, \ldots, \varphi_n] \vdash_\mathbb{K} \psi \wedge \forall\, i.\ \mathsf{Res}_3(\mathbb{K}, E)([\psi], \varphi_i) = w \qquad (2)$$

Bleibt noch die Äquivalenz der rechten Seiten (1) und (2) zu zeigen.

1. Sei $E = \emptyset$.
   (a) Die Implikation (2) $\implies$ (1) folgt aus dem Lemma
   $$\mathsf{Res}_3(\mathbb{K}, \emptyset)([\psi], \varphi) \implies \mathsf{Res}_3(\mathbb{K}, \emptyset)([], \varphi)$$
   welches durch eine elementare Induktion in der Rekursionstiefe beweisbar ist.

(b) Die Umkehrung (1) $\Longrightarrow$ (2) wird durch einen etwas komplexeren Widerspruch gezeigt. Angenommen, man fände ein allgemeineres Beispiel der folgenden Form:

$$\exists\, \varphi_1, \ldots, \varphi_n.\ [\varphi_1, \ldots, \varphi_n] \vdash_K \psi \wedge \forall\, i.\ \mathsf{Res}_3(\mathbb{K}, \emptyset)(\vec{s}, \varphi_i) = w$$
$$\forall\, \varphi_1, \ldots, \varphi_n.\ [\varphi_1, \ldots, \varphi_n] \vdash_K \psi \Rightarrow \exists\, i.\ \mathsf{Res}_3(\mathbb{K}, \emptyset)(\psi : \vec{s}, \varphi_i) = f$$

Dann gäbe es also ein $\varphi_i \in \mathfrak{s}^+_{\mathbb{C}}(\psi)$, so daß

$$\varphi_i \notin \operatorname{dom} \vec{s} \qquad \mathsf{Res}_3(\mathbb{K}, \emptyset)(\vec{s}, \varphi_i) = w \qquad \mathsf{Res}_3(\mathbb{K}, \emptyset)(\psi : \vec{s}, \varphi_i) = f$$

Das wäre aber nur möglich, falls φ_i rekursiv von ψ abhinge, falls also ein Zwischenstack $\vec{t}$ existierte, so daß

$$\mathsf{Res}_3(\mathbb{K}, \emptyset)(\vec{t} +\!\!+ \varphi_i : \vec{s}, \psi) = w$$

Hierfür ist wiederum eine Ableitung $[\varphi'_1, \ldots, \varphi'_m] \vdash_{\mathbb{K}} \psi$ notwendig, so daß

$$\forall\, j.\ \mathsf{Res}_3(\mathbb{K}, \emptyset)(\psi : \vec{t} +\!\!+ \varphi_i : \vec{s}, \varphi'_j) = w$$

Das heißt aber, daß φ_i nicht in $\varphi'_1, \ldots, \varphi'_m$ enthalten sein kann. Ersetze nun in obiger Annahme $\varphi_1, \ldots, \varphi_n$ durch $\varphi'_1, \ldots, \varphi'_m$ und iteriere. Zu jedem ausgewählten Element $\varphi_i \in \mathfrak{s}_{\mathbb{C}}(\psi)$ folgt stets die Existenz eines weiteren, von allen bisher gewählten verschiedenen. Das ist ein Widerspruch zur Rationalität von $\mathbb{C}$. Durch Spezialisierung auf $\vec{s} = []$ folgt die behauptete Implikation.

2. Sei $E = K$. Hier gelten wegen der Symmetrie des Potenzverbandes $\mathcal{P}(K)$ die dualen Argumente: (1) $\Longrightarrow$ (2) ist trivial und (2) $\Longrightarrow$ (1) folgt durch konstruierten Widerspruch. □

Korollar 5.59. *Die Erwartungen $\emptyset$ und K sind stets auch stark konsistent.*

Die trivialen Erwartungen sind nicht nur stets konsistent, sondern generieren auch die beiden ausgezeichneten Modelle.

Satz 5.60. *Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein rational corekursiver Kalkül. $M(\mathbb{K}, \emptyset)$ ist das kleinste, $M(\mathbb{K}, K)$ das größte Modell von $\mathbb{K}$.*

Beweis. Weitgehend analog zum Widerspruchsbeweis des vorigen Satzes. Aus Symmetriegründen wird wieder nur $E = \emptyset$ betrachtet. Sei M ein Modell von $\mathbb{K}$. Angenommen, es existiere ein Beispiel der Form:

$$\mathsf{Res}_3(\mathbb{K}, \emptyset)(\vec{s}, \psi) = w \qquad \psi \notin M$$

Dann folgt aus der Definition des Algorithmus:

$$\exists\, \varphi_1, \ldots, \varphi_n.\ [\varphi_1, \ldots, \varphi_n] \vdash_K \psi \wedge \forall\, i.\ \mathsf{Res}_3(\mathbb{K}, \emptyset)(\psi : \vec{s}, \varphi_i) = w$$

Außerdem kann ψ nicht in $\varphi_1, \ldots, \varphi_n$ enthalten sein. Andererseits folgt aus der Fixpunkteigenschaft von M mit dem *modus tollens*:

$$\forall\, \varphi_1, \ldots, \varphi_n.\ [\varphi_1, \ldots, \varphi_n] \vdash_K \psi \Rightarrow \exists\, i.\ \varphi_i \notin M$$

Durch Iteration kann unbeschränkt auf die Existenz von stets neuen Elementen in $\mathfrak{s}^+_{\mathbb{C}}(\psi)$ geschlossen werden, was der Rationalität von $\mathbb{C}$ widerspricht. □

Definition 5.61. Sei $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ ein rationaler Kalkül. Wir schreiben:

$$\lfloor \mathbb{K} \rfloor = M(\mathbb{K}, \emptyset) \qquad \lceil \mathbb{K} \rceil = M(\mathbb{K}, K)$$

Für determinierte Kalküle schreiben wir auch:

$$[\mathbb{K}] = \lfloor \mathbb{K} \rfloor = \lceil \mathbb{K} \rceil$$

□

```
lfp, gfp :: Res
lfp      = flip res3 (const False)
gfp      = flip res3 (const True )
```

Programm 5.6: Größtes und kleinstes Modell

Satz 5.62 (Monotonie). *Der Algorithmus* Res_3 *ist monoton im Erwartungsparameter:*

$$E \subseteq E' \implies M(\mathbb{K}, E) \subseteq M(\mathbb{K}, E')$$

Beweis. Durch Induktion in der Rekursionstiefe analog zu Satz 5.54. □

Die Importschnittstelle eines Kalküls bestimmt nicht nur die möglichen Bedingungen, sondern auch, welcher Teil der Erwartung überhaupt relevant ist.

Lemma 5.63 (Koinzidenz). *Sei* $\mathbb{K} = (K, \vdash_{\mathbb{K}})$ *ein Kalkül. Sei* $K_- \uplus K_+$ *eine gültige Partitionierung von* K *und* $E_1, E_2 \subseteq K$ *zwei Erwartungen. Falls* $E_1 \cap K_- = E_2 \cap K_-$ *gilt und* E_1 *(stark) konsistent ist, dann ist auch* E_2 *(stark) konsistent.*

Beweis. Betrachte einen beliebigen bedingten Kalkül $\mathbb{K} \,|\, H$ mit $H \subseteq K_+$. Das Ergebnis des Algorithmus für eine importierte Formel $\psi_+ \in K_+$ ist festgelegt:

$$\mathsf{Res}_3(\mathbb{K}, E)(\vec{s}, \psi_+) = \begin{cases} w & \psi_+ \in H \\ f & \psi_+ \notin H \end{cases}$$

Insbesondere hängt es weder von $\vec{s}$ noch von E ab. Auch das Ergebnis für eine exportierte Formel $\psi_- \in K_-$ hängt nicht von $E \cap K_+$ ab. Folglich gilt:

$$\mathsf{Res}_3(\mathbb{K}, E_1) = \mathsf{Res}_3(\mathbb{K}, E_2)$$ □

5.4.1 Operationale und semantische Unabhängigkeit

Die folgenden Sätze besagen, daß konsistente Erwartungen für die Bestandteile von Coprodukt- und bedingten Kalkülen unabhängig gewählt werden können. Die Dekomposition eines Kalküls in Einzelteile bestimmt also über die Granularität der mittels Res_3 korrekt entscheidbaren gemischten Fixpunkte.

Satz 5.64 (Coprodukt). *Das Entscheiden eines Coproduktkalküls zerfällt in Einzelfälle. Sei für alle* $i \in I$ $\mathbb{K}_i = (K_i, \vdash_{\mathbb{K}_i})$ *ein rational corekursiver Kalkül und* E_i *eine Erwartung an* $\mathbb{K}_i$*. Dann gilt für alle* $j \in I$*:*

$$\mathsf{Res}_3(\coprod_{i \in I} \mathbb{K}_i, \coprod_{i \in I} E_i)\big([], \iota_j(\psi)\big) = \mathsf{Res}_3(\mathbb{K}_j, E_j)([], \psi)$$

Beweis. Betrachte alle Stacks aus der Sprache $S_j = \mathcal{P}(\iota_j^*)(K_j^*)$ der Formeln der j-ten Komponente. Für alle $\iota_j^*(\vec{s}) \in S_j$ läßt sich durch Induktion in der Rekursionstiefe zeigen:

$$\mathsf{Res}_3(\coprod_{i \in I} \mathbb{K}_i, \coprod_{i \in I} E_i)\big(\iota_j^*(\vec{s}), \iota_j(\psi)\big) = \mathsf{Res}_3(\mathbb{K}_j, E_j)(\vec{s}, \psi)$$

Die Behauptung ist der Spezialfall $[] \in S_j$. □

```
res3a :: Eq α ⇒ Calc α → Pred α → Calc α → Pred α → Pred α
res3a k e k' e' = recur []
  where
    recur s p = if k' p == [] then
                  let recur' = recur (p : s)
                   in if any (== p) s then e p
                      else dnf recur' (k p)
                else
                  res3 k' e' p
```

Satz 5.65 (Bedingung). *Ein Modell, das in einen bedingten Kalkül geschachtelt ist, kann geschachtelt entschieden werden. Seien $\mathbb{K} = (K, \vdash_{\mathbb{K}}), \mathbb{K}' = (K', \vdash_{\mathbb{K}'})$ rational corekursive Kalküle und E' konsistente Erwartung an $\mathbb{K}'$. Sei K_+, K_- eine gültige Schnittstelle von $\mathbb{K}$ und $K' \subseteq K_+$. Dann gilt für jede stark konsistente Erwartung E an $\mathbb{K}$:*

$$\mathsf{Res}_3\big(\mathbb{K} \mid M(\mathbb{K}', E'), E\big)(\vec{s}, \psi) = \mathsf{Res}_{3a}\big((\mathbb{K}, E) \mid (\mathbb{K}', E')\big)(\vec{s}, \psi)$$

wobei $\mathsf{Res}_{3a}\big((\mathbb{K}, E) \mid (\mathbb{K}', E')\big)(\vec{s}, \psi)$ wie folgt definiert ist:

1. *Falls $\psi \in K_-$, verfahre wie bei $\mathsf{Res}_3(\mathbb{K}, E)$, außer daß für Rekursion wieder $\mathsf{Res}_{3a}\big((\mathbb{K}, E) \mid (\mathbb{K}', E')\big)$ zu verwenden ist.*
2. *Falls $\psi \in K'$, verfahre vollständig wie bei $\mathsf{Res}_3(\mathbb{K}', E')$.*
3. *Andernfalls, also für $\psi \in K_+ \setminus K'$, liefere stets f.*

Beweis. Betrachte alle Stacks $\vec{s}$ aus der Sprache $S = K_-^*$ der nicht importierten Formeln. Induktion in der rekursiven Struktur des Algorithmus.

1. Induktionsfall $\psi \in K_-$. Hier verhält sich Res_{3a} in der Rekurenzrelation genau wie Res_3. Per Induktionsannahme korrekte Teilergebnis werden also korrekt verknüpft. Außerdem bleibt die Eigenschaft $\vec{s} \in S$ für den Stack $\psi : \vec{s}$ unmittelbar rekursiver Aufrufe erhalten, so daß sie für den nächsten Fall angenommen werden kann.
2. Basisfall $\psi \in K'$. Hier gilt:
$$\begin{aligned}\mathsf{Res}_{3a}(\mathbb{K}', E')(\vec{s}, \psi) &= \mathsf{Res}_3(\mathbb{K}', E')(\vec{s}, \psi) \\ &= \mathsf{Res}_3(\mathbb{K}', E')([], \psi)\end{aligned}$$
da weder ψ noch ein rekursives Argument in $\vec{s}$ enthalten sein kann.
3. Basisfall $\psi \in K_+ \setminus K'$. Hier existiert im bedingten Kalkül $\mathbb{K} \mid M(\mathbb{K}', E')$ keine Regel für ψ, also ist f das korrekte Ergebnis.

□

Programm 5.4.1 zeigt eine leicht variierte Implementierung, die ohne Elementtest auskommt.
Die Unabhängigkeit läßt sich von der operationalen auf die modelltheoretische Ebene fortsetzen.

Satz 5.66 (Coprodukt). *Sei für alle $i \in I$ $\mathbb{K}_i = (K_i, \vdash_{\mathbb{K}_i})$ ein rational corekursiver Kalkül, $E_i \subseteq K_i$ eine Erwartung und $M_i \subseteq K_i$ ein Modell. Dann erhält die Coproduktbildung Konsistenz und starke Konsistenz.*

Beweis. Nach Satz 5.64 entscheidet der Algorithmus Res_3 einen Coproduktkalkül komponentenweise. Sei $\mathbb{K}$ ein Kalkül und E eine Erwartung an $\mathbb{K}$. Ist $\mathbb{K}$ ein Coprodukt, dann ist E ebenfalls ein Coprodukt, und es gilt:

$$\mathbb{K} = \coprod_{i\in I} \mathbb{K}_i \wedge E = \coprod_{i\in I} E_i \implies M(\mathbb{K}, E) = \coprod_{i\in I} M(\mathbb{K}_i, E_i)$$

Ist für ein i die Teilerwartung E_i konsistent, dann ist das Teilergebnis $M(\mathbb{K}_i, E_i)$ Modell des Teilkalküls $\mathbb{K}_i$. Da das Coprodukt von Modellen ebenfalls ein Modell ist, ist E konsistent.

Sei nun H eine gültige Bedingung an $\mathbb{K}$. Diese zerfällt ebenfalls in das Coprodukt einer Familie von H_i. Ist für ein i die Teilerwartung E_i stark konsistent für die $\mathbb{K}_i$, dann ist sie auch konsistent für $\mathbb{K}_i \mid H_i$. Folglich ist E auch konsistent für $\mathbb{K} \mid H$. □

Satz 5.67 (Bedingung)**.** *Seien $\mathbb{K}, \mathbb{K}'$ rational corekursive Kalküle. Sei E konsistente Erwartung an $\mathbb{K}$ und E' stark konsistente Erwartung an $\mathbb{K}'$. Sei $K \subseteq K'_+$. Dann ist E' auch stark konsistente Erwartung an $\mathbb{K}' \mid M(\mathbb{K}, E)$.*

Beweis. Folgt unmittelbar aus der Definition der starken Konsistenz. □

Beispiel 5.68. Die intendieren Modelle aus Beispiel 5.17 sind:

$$\begin{aligned} [\![\mathbb{F}]\!] &\models \mathbb{F} & \lfloor[\![\bar{\mathbb{F}}]\!]\rfloor &\models \bar{\mathbb{F}} \\ [\![\mathbb{G}]\!] &\models \mathbb{G} & \lceil[\![\bar{\mathbb{G}}]\!]\rceil &\models \bar{\mathbb{G}} \\ [\![\mathbb{H}]\!] &\models \mathbb{H} & \lfloor[\![\bar{\mathbb{F}}]\!]\rfloor + \lceil[\![\bar{\mathbb{G}}]\!]\rceil &\models \bar{\mathbb{H}} \\ [\![\mathbb{J}]\!] &\models \mathbb{J} & \lfloor[\![\bar{\mathbb{J}}]\!]\rfloor &\models \bar{\mathbb{J}} \\ [\![\mathbb{J}']\!] &\models \mathbb{J}' & \lceil[\![\bar{\mathbb{J}}']\!]\rceil &\models \bar{\mathbb{J}}' \end{aligned}$$

Man beachte insbesondere das nicht-extremale intendierte Modell von $\bar{\mathbb{H}}$.

Um Ordnungskalküle auf Listen von natürlichen Zahlen zu spezifizieren, führen wir zunächst Gleichheitskalküle natürlicher Zahlen $\mathbb{E}, \bar{\mathbb{E}} = J_0 \cup J_2$ ein. Deren intendierte Modelle sind wie oben:

$$[\![\mathbb{E}]\!] \models \mathbb{E} \qquad\qquad \lceil[\![\bar{\mathbb{E}}]\!]\rceil \models \bar{\mathbb{E}}$$

Damit ergeben sich als Bedingungen zum Import initialer bzw. erweiterter natürlicher Zahlen:

$$H = \mathcal{P}(\mathsf{o})\big([\![\mathbb{J}]\!]\big) \cup \mathcal{P}(\mathsf{e})\big([\![\mathbb{E}]\!]\big) \qquad\qquad \bar{H} = \mathcal{P}(\mathsf{o})\big(\lfloor[\![\bar{\mathbb{J}}]\!]\rfloor\big) \cup \mathcal{P}(\mathsf{e})\big(\lceil[\![\bar{\mathbb{E}}]\!]\rceil\big)$$

Zwei Ordnungen $<, \leq$ über zwei Listentypen über zwei Zahlentypen ergeben acht Kalküle:

$$\begin{array}{cccc} \mathbb{L} \mid H & \bar{\mathbb{L}} \mid H & \mathbb{L} \mid \bar{H} & \bar{\mathbb{L}} \mid \bar{H} \\ \mathbb{L}' \mid H & \bar{\mathbb{L}}' \mid H & \mathbb{L}' \mid \bar{H} & \bar{\mathbb{L}}' \mid \bar{H} \end{array}$$

□

Beispiel 5.69. Als Beispielkalkül für eine konkrete Berechnung mit Res_3 betrachten wir die Relation $\leq$ auf rationalen Listen auf erweiterten natürlichen Zahlen, also den Kalkül $\mathbb{K} = \bar{\mathbb{L}}' \mid \bar{H}$ und dessen größtes Modell. Als verfügbare Daten betrachten wir eine kleine Coalgebra $\mathbb{A} = (A, \alpha)$ zur gemeinsamen Signatur der natürlichen Zahlen und Listen, wobei A und α durch folgende Tabelle definiert sind:

$$\begin{array}{lll} \mathsf{a} \mapsto \mathsf{z} & \mathsf{e} \mapsto \mathsf{cons}(\mathsf{c}, \mathsf{f}) & \mathsf{h} \mapsto \mathsf{cons}(\mathsf{d}, \mathsf{i}) \\ \mathsf{b} \mapsto \mathsf{s}(\mathsf{a}) & \mathsf{f} \mapsto \mathsf{cons}(\mathsf{b}, \mathsf{g}) & \mathsf{i} \mapsto \mathsf{cons}(\mathsf{b}, \mathsf{k}) \\ \mathsf{c} \mapsto \mathsf{s}(\mathsf{c}) & \mathsf{g} \mapsto \mathsf{cons}(\mathsf{b}, \mathsf{g}) & \mathsf{k} \mapsto \mathsf{cons}(\mathsf{b}, \mathsf{i}) \\ \mathsf{d} \mapsto \mathsf{s}(\mathsf{d}) & & \end{array}$$

Diese kann beispielsweise als Speicherzustand codiert sein. Da beim Entscheiden von Prädikaten keine neuen Zellen entstehen, kann hier auf indirekte Adressierung verzichtet werden. Die Listen e und h sind

bei genauem Hinsehen bisimilar, unterscheiden sich aber in der Repräsentation des ersten Elementes und des anschließenden Zyklus.

Das folgende Diagramm zeigt den rekursiven Ablauf der Entscheidung von $\mathsf{o}^*(\mathsf{e},\mathsf{h}) \in \lceil\mathbb{K}\rceil$ mittels Res_3. Die Schachtelung gemäß Satz 5.64 und 5.65 ist implizit. Disjunktionen sind durch geschweifte, Konjunktionen durch eckige Klammern dargestellt. Jeder Rekursionsschritt ist mit der entsprechenden Kalkülregel gekennzeichnet. Die Abbruchfälle infolge Zyklenerkennung sind markiert und die zugehörigen Paare gleicher Inkarnationen unterstrichen.

$$
\mathsf{o}^*(\mathsf{e},\mathsf{h}) \begin{cases}
\xrightarrow{L_2} \underline{\mathsf{o}(\mathsf{c},\mathsf{d})} \xrightarrow{\mathsf{o}(J_2)} \underline{\mathsf{o}(\mathsf{c},\mathsf{d})} \xrightarrow{\lfloor\circlearrowleft\rfloor} f \\
\xrightarrow{L_3} \left[\begin{array}{l}
\underline{\mathsf{e}(\mathsf{c},\mathsf{d})} \xrightarrow{\mathsf{e}(J_2)} \underline{\mathsf{e}(\mathsf{c},\mathsf{d})} \xrightarrow{\lceil\circlearrowleft\rceil} w \\
o^*(f,i) \begin{cases}
\xrightarrow{L_2} o(b,b) \xrightarrow{o(J_2)} o(a,a) \longrightarrow f \\
\xrightarrow{L_3} \left[\begin{array}{l}
e(b,b) \xrightarrow{e(J_2)} e(a,a) \xrightarrow{e(J_0)} w \\
\underline{o^*(g,k)} \begin{cases}
\xrightarrow{L_2} o(b,b) \dashrightarrow f \\
\xrightarrow{L_3} \left[\begin{array}{l}
e(b,b) \dashrightarrow w \\
o^*(g,i) \begin{cases}
\xrightarrow{L_2} o(b,b) \dashrightarrow f \\
\xrightarrow{L_3} \left[\begin{array}{l} e(b,b) \dashrightarrow w \\ \underline{o^*(g,k)} \xrightarrow{\lceil\circlearrowleft\rceil} w \end{array}\right.
\end{cases}
\end{array}\right.
\end{cases}
\end{array}\right.
\end{cases}
\end{array}\right.
\end{cases}
$$

□

5.4.2 Graphische Deutung

Es ist kein Zufall, daß gerade Coprodukt und Bedingung sich kompositional zur Wahl von Erwartungen verhalten. Veranschaulicht man sich die Rekursionsstruktur eines komplexen Kalküls als Graph, dann zerfällt dieser in ein Netz stark zusammenhängender Komponenten.

1. Das Coprodukt beschreibt eine Schar unzusammenhängender Komponenten.
2. Die Bedingung beschreibt ein Paar einseitig zusammenhängender Komponenten.
3. Die lokalen Erwartungen an verschiedene Komponenten sind unabhängig.
4. Innerhalb einer Komponente ist im Allgemeinen nur eine triviale Erwartung konsistent.

Aus der relativ abstrakten Sicht der rekursiven Abhängigkeit ist damit die Struktur eines Kalküls vollständig beschrieben.

5.5 Bisimilarität

Die Bisimilarität ist nicht nur ein grundlegendes zweistelliges Prädikat, das generisch für alle Signaturen definiert ist. In einem System, in dem Daten mit finaler Semantik in nichtfinalen Coalgebren repräsentiert werden, ist sie ein essentieller Bestandteil der semantischen Abstraktion. Dieser Abschnitt soll zeigen, wie Bismilarität als Kalkül-Konstruktion spezifiziert werden kann.

Definition 5.70 (Bisimilaritätskalkül)**.** Sei Σ ein beliebiger polynomieller Signaturfunktor und $\mathbb{C} = (C, \gamma)$ eine rationale Σ-Coalgebra. Der Kalkül $\mathbb{Y}(\Sigma, \mathbb{C})$ wird durch eine Ableitungsfunktion $\kappa(\Sigma, \mathbb{C}) : C \to (C^*)^*$ induktiv über der Struktur von Σ definiert, wobei beim Abstieg zu einer Teilsignatur Σ' jeweils die Coalgebra zu einer Σ'-Coalgebra spezialisiert wird:

1. Konstante Anteile bisimilarer Elemente müssen identisch sein.

$$\kappa\big(\mathcal{C}_A, (C, \gamma)\big)(x, y) = \begin{cases} [[]] & \gamma(x) = \gamma(y) \\ [] & \gamma(x) \neq \gamma(y) \end{cases}$$

2. Rekursive Anteile bisimilarer Elemente müssen bisimilar sein.

$$\kappa\big(\mathcal{I}, (C, \gamma)\big)(x, y) = \big[[(\gamma(x), \gamma(y))]\big]$$

3. Bisimilare Tupel müssen komponentenweise bisimilar sein. Dazu wird die Beobachtung auf jeweils eine Komponente eingeschränkt, und die erhaltenen Prämissen konjugiert.

$$\kappa\big(\prod_{i \in I} \Sigma_i, (C, \gamma)\big)(x, y) = \vec{\prod_{i \in I}} \kappa\big(\Sigma_i, (C, \pi_i \circ \gamma)\big)(x, y)$$

   Dabei bezeichnet $\vec{\prod}$ das endliche Listenprodukt, welches durch folgendes binäre Listenprodukt $\vec{\times}$ bis auf Reihenfolge definiert ist:

$$[] \mathbin{\vec{\times}} R = []$$
$$(\vec{l} : S) \mathbin{\vec{\times}} [\vec{r}_1, \ldots, \vec{r}_n] = [\vec{l} +\!\!+ \vec{r}_1, \ldots, \vec{l} +\!\!+ \vec{r}_n] +\!\!+ (S \mathbin{\vec{\times}} R)$$

   Unter der aus Satz 5.40 bekannten Interpretation Q als disjunktive Normalform beschreibt dieses Listenprodukt in jeder Reihenfolge genau die Konjunktion.

4. Bisimilare Cotupel müssen identische Injektionen bisimilarer Argumente sein. Dazu wird die Trägermenge auf mit der jeweiligen Injektion gebildete Beobachtungen eingeschränkt, und die erhaltenen Prämissen disjungiert.

$$\kappa\big(\coprod_{i \in I} \Sigma_i, (C, \gamma)\big)(x, y) = \mathop{+\!\!+}_{i \in I} \kappa'\big(\Sigma_i, (C_i, \iota_i^{-1} \circ \gamma)\big)(x, y)$$

   Dabei ist C_i die Einschränkung auf die i-te Injektion:

$$C_i = \overline{\mathcal{P}}(\gamma)(\operatorname{codom} \iota_i)$$

   Die Funktion κ' erweitert den Definitionbereich von κ beliebig:

$$\kappa'\big(\Sigma, (C, \gamma)\big)(x, y) = \begin{cases} \kappa\big(\Sigma, (C, \gamma)\big) & x, y \in C \\ [] & \text{sonst} \end{cases}$$

   Unter Interpretation Q beschreibt die Konkatenation genau die Disjunktion.

5. Für Listensignaturen kann eine Definition aus Produkt und Coprodukt abgeleitet werden:

$$\Sigma^* = \coprod_{n \in \mathcal{N}} \Sigma^n$$

□

Lemma 5.71. *Die Funktion $\kappa(\Sigma, \mathbb{C})$ modelliert die aus Abschnitt 2.4.2.5 bekannte, generische Bisimulationsoperation $\upsilon(\Sigma)$:*

$$\begin{aligned} \kappa(\Sigma,\mathbb{C})(x,y) = [] \quad &\iff \quad \upsilon(\Sigma)(x,y) = \bot \\ \kappa(\Sigma,\mathbb{C})(x,y) = [\vec{\varphi}] \quad &\iff \quad \mathfrak{s}(\Sigma)\big(\upsilon(\Sigma)(x,y)\big) = \operatorname{codom}\vec{\varphi} \end{aligned}$$

Beweis. Durch strukturelle Induktion in Σ. Zeige simultan $\big|\kappa(\Sigma,\mathbb{C})(x,y)\big| \leq 1$.

1. Konstanter und identischer Fall sind trivial.

2. Im Coproduktfall sind alle Teilergebnisse leer, falls x und y aus verschiedenen Injektionen stammen. Andernfalls ist genau ein j-tes Teilergebnis nichtleer. Dann gilt:

$$\mathop{+\!\!+}_{i \in I} \kappa'\big(\Sigma_i, (C_i, \iota_i^{-1} \circ \gamma)\big)(x,y) = \kappa\big(\Sigma_j, (C_j, \iota_j^{-1} \circ \gamma)\big)(x,y)$$

3. Im Produktfall gelten die Rechenregeln:

$$[] \mathbin{\vec{\times}} R = [] \qquad S \mathbin{\vec{\times}} [] = [] \qquad [\vec{s}] \mathbin{\vec{\times}} [\vec{r}] = [\vec{s} +\!\!+ \vec{r}]$$

   Folglich ist das Ergebnis leer, falls irgendein Teilergebnis leer ist. Ansonsten enthält das Ergebnis genau die Konkatenation der einzigen Elemente der Teilergebnisse. □

Satz 5.72. *Der Bisimilaritätskalkül ist korrekt: Das größte Modell $Y = \lceil \mathbb{Y}(\Sigma,\mathbb{C}) \rceil$ ist die Bisimilaritätsrelation auf der Σ-Coalgebra $\mathbb{C}$.*

Beweis. Zeige zunächst, daß $\big(Y, \upsilon(\Sigma)\big)$ eine Σ-Coalgebra und damit eine Bisimulation ist:

$$\begin{gathered} \upsilon(\Sigma) : Y \to \Sigma(Y) \\ \psi \in Y \implies \underbrace{\psi \in \operatorname{dom}\upsilon(\Sigma)}_{(1)} \wedge \underbrace{\mathfrak{s}(\Sigma)\big(\upsilon(\Sigma)(\psi)\big) \subseteq Y}_{(2)} \end{gathered}$$

(1) folgt aus Lemma 5.71.

(2) Aus $\psi \in Y$ folgt $\mathsf{Res}_3\big(\mathbb{Y}(\Sigma,\mathbb{C}), C\big)([], \psi) = w$ und damit auch $\mathsf{Res}_3\big(\mathbb{Y}(\Sigma,\mathbb{C}), C\big)(\vec{s}, \psi) = w$ für beliebige $\vec{s}$. Aus Lemma 5.71 folgt außerdem, daß $\mathsf{Res}_3\big(\mathbb{Y}(\Sigma,\mathbb{C}), C\big)([\psi], \varphi) = w$ für alle $\varphi \in \mathfrak{s}(\Sigma)\big(\upsilon(\Sigma)(\psi)\big)$ gelten muß.

   Das gesuchte $\mathsf{Res}_3\big(\mathbb{Y}(\Sigma,\mathbb{C}), C\big)([], \varphi)$ unterscheidet sich von letzterem nur durch zusätzliche rekursive Aufrufe $\mathsf{Res}_3\big(\mathbb{Y}(\Sigma,\mathbb{C}), C\big)(\vec{s}, \psi)$, die aber wie oben festgestellt stets w liefern.

Zeige nun, daß jedes Paar bisimilarer Elemente in Y enthalten ist. Für Elemente $\psi \in Y'$ einer Bisimulation $\big(Y', \upsilon(\Sigma)\big)$ gilt, daß $\upsilon(\Sigma)(\psi)$ definiert ist und alle Paare der Umgebung $\mathfrak{s}(\Sigma)\big(\upsilon(\Sigma)(\psi)\big)$ ebenfalls in Y' sind. Per Induktion in der verbleibenden Rekursionstiefe $w(\vec{s}, \psi)$ kann dann gezeigt werden, daß $\mathsf{Res}_3\big(\mathbb{Y}(\Sigma,\mathbb{C}), C\big)(\vec{s}, \psi) = w$ gilt. □

Korollar 5.73. *Der Bisimilaritätskalkül ist abstrakt.*

Beispiel 5.74. Sei $\Sigma = \mathcal{C}_1 + \mathcal{I}$ die bekannte Signatur der natürlichen Zahlen. Sei $\bar{\mathbb{N}} = (\bar{\mathcal{N}}, pred?)$ die bekannte finale und rationale Σ-Coalgebra. Dann gilt für alle $m, n \in \bar{\mathcal{N}}$:

$$\kappa(\Sigma, \bar{\mathbb{N}})(m,n) = \kappa'\big(\mathcal{C}_1, (\{0\}, !)\big)(m,n) +\!\!+ \kappa'\big(\mathcal{I}, (\bar{\mathcal{N}} \setminus \{0\}, pred)\big)(m,n)$$

Seien nun $m, n \neq 0$. Dann ergibt sich:

$$\begin{aligned}
\kappa(\Sigma, \bar{\mathbb{N}})(0,0) &= \kappa\big(\mathcal{C}_1, (\{0\}, !)\big)(0,0) +\!\!+ [] && = [[]] \\
\kappa(\Sigma, \bar{\mathbb{N}})(m,0) &= [] +\!\!+ [] && = [] \\
\kappa(\Sigma, \bar{\mathbb{N}})(0,n) &= [] +\!\!+ [] && = [] \\
\kappa(\Sigma, \bar{\mathbb{N}})(m,n) &= [] +\!\!+ \kappa\big(\mathcal{I}, (\bar{\mathcal{N}} \setminus \{0\}, pred)\big)(m,n) && = [[(pred(m), pred(n))]]
\end{aligned}$$

□

Beispiel 5.75. Sei $\Sigma = \mathcal{C}_1 + \mathcal{C}_A \times \mathcal{I}$ die bekannte Signatur der A-Listen. Sei $\mathbb{C} = (C, \gamma)$ eine beliebige Σ-Coalgebra. Wir schreiben $\mathsf{nil}, \mathsf{cons}(a,l)$ für die Pattern und definieren die beiden Selektoren:

$$\begin{array}{lll}
car : C \nrightarrow A & car(\mathsf{nil}) = \bot & car\big(\mathsf{cons}(a,l)\big) = a \\
cdr : C \nrightarrow C & cdr(\mathsf{nil}) = \bot & cdr\big(\mathsf{cons}(a,l)\big) = l
\end{array}$$

Für alle Elemente x, y gilt:

$$\begin{aligned}
\kappa(\Sigma, \mathbb{C})(x,y) &= \kappa'(\mathcal{C}_1, \mathbb{C}_{\mathsf{nil}})(x,y) +\!\!+ \kappa'(\mathcal{C}_A \times \mathcal{I}, \mathbb{C}_{\mathsf{cons}})(x,y) \\
\mathbb{C}_{\mathsf{nil}} &= (C_{\mathsf{nil}}, !) \qquad C_{\mathsf{nil}} = \{x \in C \mid \mathsf{nil} := x\} \\
\mathbb{C}_{\mathsf{cons}} &= \big(C_{\mathsf{cons}}, \langle car, cdr\rangle\big) \quad C_{\mathsf{cons}} = \{x \in C \mid \mathsf{cons}(_,_) := x\}
\end{aligned}$$

Für die beiden Varianten gilt ferner:

$$\begin{aligned}
\kappa(\mathcal{C}_1, \mathbb{C}_{\mathsf{nil}})(x,y) &= [[]] \\
\kappa(\mathcal{C}_A \times \mathcal{I}, \mathbb{C}_{\mathsf{cons}})(x,y) &= \kappa\big(\mathcal{C}_A, (C_{\mathsf{cons}}, car)\big)(x,y) \vec{\times} \kappa\big(\mathcal{I}, (C_{\mathsf{cons}}, cdr)\big)(x,y)
\end{aligned}$$

Folglich sind alle Elemente, auf die das Pattern nil paßt, bisimilar. Elemente, auf die die Pattern $\mathsf{cons}(a,l)$ beziehungsweise $\mathsf{cons}(a',l')$ passen, sind genau dann bisimilar, wenn $a = a'$ und $l \sim l'$. □

5.6 Beispiel aus dem Compilerbau

Die in Abschnitt 1.1.4 aufgestellte Behauptung, zyklische Probleme würden auch und besonders im Rahmen der Analyse und Übersetzung formaler Sprachen auftreten, soll hier an einem kleinen, aber nichttrivialen Beispiel belegt werden: *Globale Konstanten* sind ein bewährtes Mittel, um von statischen Parametern eines Programms in dokumentierter und typsicherer Weise zu abstrahieren. Damit die Verwendung dieses Sprachmittels bei der Compilation nicht mit Einbußen an Optimierungspotential erkauft wird, ist es wichtig, die Werte von Konstanten soweit möglich zur Übersetzungszeit zu ermitteln und an Verwendungsstellen zu berücksichtigen.

In traditionellen Sprachen mittleren Niveaus wie C und C++ existiert dafür eine eigene Sprachebene, die durch einen Makro-Präprozessor mittels textueller Substitution ausgewertet wird. Diese primitive Substitution hat allerdings die Nachteile, daß weder Typsicherheit a priori festgestellt, noch die Vervielfältigung von Code ausgeschlossen werden kann. Wertet man dagegen Konstantenausdrücke der eigentlichen Sprachebene schon bei der Übersetzung aus, dann ergibt sich das Problem der potentiellen Nichttermination, das für TURING-vollständige Objektsprachen bekanntermaßen nicht entscheidbar ist.

Für eine Analyse der Konstanten eines Programms in Bezug auf die unbedenkliche Auswertbarkeit zur Übersetzungszeit treffen wir einige Modellannahmen:

1. Sei zu jeder Konstanten c vom Typ t ihr Wert in Form einer Initialisierungsprozedur $\mathsf{init}_c : 1 \to t$ gegeben.

2. Es existieren hinreichend akkurate konservative Abschätzungsverfahren, ob eine gegebene Initialisierung

   (a) Seiteneffekte verursacht, und

   (b) unter der Annahme, daß alle vorkommenden Konstanten bereits initialisiert seien, terminiert.

3. Die Verwendung einer Konstanten in einer Initialisierung kann potentiell immer deren rekursive Initialisierung auslösen.

4. Konstanten können zyklisch definiert sein.

5. Initialisierungsprozeduren können über Zyklenbehandlung verfügen.

Gesucht ist nun ein Verfahren, daß ermittelt, ob eine gegebene Konstante zur Übersetzungszeit initialisiert werden kann, ohne daß Seiteneffekte auftreten oder es bei der rekursiven Initialisierung verwendeter Konstanten zur Nichttermination kommt. Eine solche Konstante nennen wir *harmlos*.

Beispiel 5.76 (Harmlosigkeits-Analyse). Sei C die endliche Menge der Konstanten eines Programms. Sei p ein Prädikat auf C, welches die Termination und Seiteneffektfreiheit von Konstanten konservativ abschätzt. Es gelte also $p(c)$ nur für nachweislich terminierendes und seiteneffektfreies init_c, ansonsten $\neg p(c)$. Sei ferner $\delta : C \to C^*$ eine Funktion, welche jeder Konstanten c eine Aufzählung aller in der Definition von c vorkommenden Konstanten zuordnet. Dann kann in erster Näherung das Harmlosigkeitsprädikat q durch folgenden Kalkül spezifiziert werden:

$$\frac{q(d_1) \quad \cdots \quad q(d_n)}{q(c)} \text{ falls } \delta(c) = [d_1, \ldots, d_n] \text{ und } p(c)$$

Als geeignete Erwartung kommt keiner der Extremfälle in Frage, da ein Zyklus von Konstanten nur harmlos ist, wenn eine Zyklenbehandlung installiert ist. Sei y ein weiteres Prädikat auf C, so daß die Initialisierungsfunktion einer Konstanten c eine terminierende Zyklenbehandlung besitzt, falls $y(c)$. Eine gemischte Erwartung $E = \{c \mid c \in C, y(c)\}$ ist allerdings im Allgemeinen nicht konsistent, wie man sich an Beispiel 5.57 leicht veranschaulichen kann.

Um für jeden Zyklus zu einer konsistenten Erwartung zu kommen, kann man den praktischen Effekt nutzen, daß die Termination bereits dann für alle Elemente des Zyklus gewährleistet ist, wenn wenigstens eines davon eine Zyklenbehandlung besitzt. Diese Information kann leicht beim Abstieg propagiert werden:

$$\frac{q(d_1, z') \quad \cdots \quad q(d_n, z')}{q(c, z)} \text{ falls } \delta(c) = [d_1, \ldots, d_n] \text{ und } p(c), \text{ wobei } z' = y(c) \vee z$$

Mit der Erwartung $E = \{q(c, w) \mid c \in C\}$ beschreibt dieser Kalkül die Harmlosigkeit einer Konstanten c als $q(c, f)$. □

5.7 Zusammenfassung

In diesem Kapitel haben wir die innere Struktur von elementaren Kalkülen untersucht. Dazu wurde den Formelmengen eine algebraische, vorzugsweise coalgebraische Struktur aufgeprägt, um Zyklen in der bereits bekannten Form charakterisieren zu können. Die aufgeprägte Struktur kann dabei rein virtuell sein, oder praktisch durch Traversierung einer Datenstruktur entstehen. Damit rückt das Entscheiden solch zyklischer Prädikatdefinitionen technisch in die Nähe der Berechnung zyklischer Funktionen, wie im vorigen Kapitel behandelt. Und in der Tat läßt sich die dort entwickelte Technik der Zyklenerkennung übertragen, um zu Algorithmen zu gelangen, die nicht nur konzeptionell analog sind, sondern auch eine Implementierung teilen können.

Dem Problem der semantischen Mehrdeutigkeit zyklischer Schlußregeln wurde durch eine zusätzliche Parametrisierung des Entscheidungsalgorithmus mit einer Erwartung begegnet. Nicht jede Belegung dieses Parameters ist korrekt, und der Nachweis im Allgemeinen nicht trivial. Anstelle eines Beweisverfahrens haben wir eine konstruktive Methode vorgestellt, das intendierte Modell eines komplexen Kalküls aus kanonischen Modellen seiner Bestandteile zusammenzusetzen, wobei die betrachteten Kalkülkonstruktoren die häufigsten Fälle abdecken, aber keinen Anspruch auf Vollständigkeit erheben.

In der Kombination von zyklenfähiger Funktionsberechnung und Prädikatentscheidung zeigt sich erst der primäre Mehrwert des strikten Ansatzes, da hiermit Probleme gelöst werden können, die in einem nichtstrikten algebraischen System nicht lösbar sind, zumindest nicht ohne die Zyklen zuvor strukturell offenzulegen und durch nichttriviale Graphcodierung zu eliminieren. Im nächsten Kapitel soll nun noch belegt werden, daß die Zyklenbehandlung keine allgemein unannehmbaren Nachteile mit sich bringt, indem technische Detailprobleme und Optimierungspotential untersucht werden.

Kapitel 6

Randbedingungen

In den vorangehenden Kapiteln wurden theoretische Algorithmen zum zyklischen Berechnen von Funktionen und Entscheiden von Prädikaten auf coalgebraischer Basis entwickelt. Es handelt sich dabei um echte Erweiterungen der klassischen, fundierten algebraischen Verfahren. Bevor jedoch in einem praktischen System die wohlerforschten algebraischen Berechnungstechniken durch coalgebraische ausgetauscht werden können, müssen noch einige Details untersucht werden:

1. Läßt sich der zusätzliche Aufwand für Zyklenprotokollierung und -erkennung auf ein akzeptables Maß beschränken? Was geschieht insbesondere, wenn die erweiterten coalgebraischen Techniken auf algebraische Daten angewendet werden?

2. Wie interferiert die Zyklenprotokollierung mit der Elimination von Funktionsinkarnationen (*tail-call*), die eine zentrale Voraussetzung für effiziente Sprünge und Schleifen in funktionalen Programmen ist?

3. Welche Auswirkung haben Interpretationen, die im Seiteneffekt den Speicherinhalt verändern, auf Korrektheit, Termination und Aufwand der Verfahren?

4. Was bedeutet die Konstruktion zyklischer Daten für die automatische Speicherverwaltung?

Jeder dieser Fragen ist im Folgenden ein Abschnitt gewidmet.

6.1 Effiziente Zyklenbehandlung

Eine naive Implementierung der Zyklenerkennung führt bei jeder Funktionsinkarnation zu einer vollständigen Traversierung des Aufrufstacks. Daraus ergibt sich ein Aufwand proportional zum Integral der Stacktiefe über die Programmlaufzeit, gemessen in Funktionsaufrufen. Bei beschränkter Größe der einzelnen Blöcke auf dem Stack ist dessen Tiefe linear beschränkt, der Aufwand ist also schlimmstenfalls quadratisch in der Zahl der Funktionsaufrufe. Das ist für realistisch verschachtelte Programme inakzeptabel. Insbesondere unbefriedigend ist ein solch erheblicher Mehraufwand bei Programmen, die überwiegend azyklische Daten behandeln und daher auch mit den herkömmlichen Techniken abgearbeitet werden könnten. Dem kann man auf zweierlei Art begegnen:

1. Erweiterte Berechnungstechniken werden explizit nur für Unterprogramme angewendet, die sie auch erfordern. Dafür sind die hier vorgestellten Verfahren gut geeignet. Einerseits werden kaum Annahmen über die Repräsentation von Eingaben gemacht, sofern eine geeignete Interpretation zur Verfügung steht. Andererseits lassen sich die Ausgaben leicht auf dieselbe Repräsentation abbilden, da alle Aktivitäten streng lokal ablaufen. Es muß nicht auf eingebettete verzögerte Berechnungen, wie etwa bei einer *lazy*-Auswertung, Rücksicht genommen werden.

2. Die Techniken werden zwar universell eingesetzt, aber um geeignete Optimierungen ergänzt, die unnötigen Mehraufwand weitgehend eliminieren. Dieser theoretisch reizvollere Ansatz wird im Folgenden weiter ausgeführt.

6.1.1 Statische Optimierung

6.1.1.1 Azyklische Funktionen

Ein Programmteil, der nicht dazu bestimmt ist, auf zyklischen Daten zu operieren, muß auch keine Zyklenbehandlung vorsehen. Verfeinert man die Granularität auf einzelne Funktionen, so ergibt sich ein hybrides Berechnungsverfahren, welches je nach aufgerufener Funktion einen Schritt nach $\mathsf{Ana}_2/\mathsf{Res}_2$ oder $\mathsf{Ana}_3/\mathsf{Res}_3$ ausführt. Technisch läßt sich ein solches Verfahren entweder über zwei getrennte Stacks implementieren, oder über eine explizite Verkettung der potentiell zyklischen Aufrufe. Bei der Zyklenerkennung kann dann diese Verkettung zurückverfolgt, und somit alle nichtzyklischen Aufrufe übersprungen werden.

6.1.1.2 Aufrufgraph

Damit ein Aufruf überhaupt in einen Zyklus führen kann, muß eine Funktion in der Lage sein, sich direkt oder indirekt rekursiv selbst aufzurufen.

1. In Programmiersprachen mit statischer Bindung kann dies gegebenenfalls durch eine statische Kontrollflußanalyse ausgeschlossen werden. Dazu wird der statische Aufrufgraph gebildet und auf stark zusammenhängende Komponenten untersucht. Nur Aufrufe innerhalb einer Komponente können Bestandteil eines Zyklus sein.

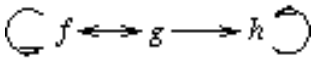
2. Bei semi-dynamischer Bindung, wie etwa durch virtuelle Methoden in objektorientierten Sprachen oder Funktionsvariablen in funktionalen Sprachen höherer Ordnung, ist im allgemeinen Fall eine komplexere Kombination aus Kontroll- und Datenflußanalyse erforderlich, siehe [58]. Die meisten praktischen Programmiersprachen unterstützen semi-dynamische Bindung. Code, der nachweislich keine dynamische Bindung nutzt, ist aber weiterhin statisch analysierbar.

3. Im Fall von voll dynamischer Bindung, durch dynamisches Laden, dynamische Codeerzeugung oder Reflektion, ist eine Analyse kaum noch möglich.

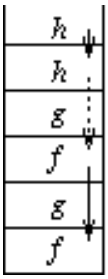
Steht fest, daß ein Funktionsaufruf nicht Teil eines Zyklus sein kann, dann muss bei Zyklenerkennung keiner seiner Vorgänger auf dem Stack beachtet werden. Die oben beschriebene Organisation des Stacks

mit expliziter Verkettung macht es möglich, an dieser Stelle die Verkettung auszusetzen, wie folgendes Beispiel illustriert.

Beispiel 6.1. Sei ein Programm mit drei Funktionen f, g, h und folgendem statischen Aufrufgraph gegeben:



Dabei haben f und h Zyklenerkennung, g aber nicht. Ein möglicher Aufrufstack bei der Ausführung könnte wie folgt aussehen. Dabei ist die Rückwärtsverkettung der zyklischen Aufrufe durch Pfeile dargestellt. Der gepunktete Pfeil in der Mitte kann nach Analyse des Aufrufgraphen entfernt werden:



6.1.2 Dynamische Optimierung

Zur Laufzeit ergeben sich wesentlich weitergehende Optimierungsmöglichkeiten. Als Motivation dienen die folgenden Überlegungen:

1. Vor der Erkennung eines Zyklus vom Umfang n liegen notwendigerweise $(n-1)$ als potentiell zyklisch eingestufte Aufrufe, bei denen die Zyklenerkennung aber fehlschlägt. Damit ergibt sich quadratischer Aufwand für unproduktive administrative Arbeiten, die der eigentlichen Berechnung nicht dienen.

2. Die rationale Coalgebra als Definitionsbereich zyklischer Funktionen umfaßt auch die initiale Algebra. Es sollte also eine einzige Funktionsdefinition zur Anwendung sowohl auf zyklische als auch auf azyklische Daten dienen, wobei im azyklischen Fall die Zyklenerkennung offensichtlich überflüssig ist. Die beiden Fälle können im Allgemeinen aber nur dynamisch unterschieden werden.

Beide Argumente führen zur selben Forderung: Die Anzahl der Zyklenerkennungsschritte soll auf ein Minimum, vorzugsweise auf genau einen pro Zyklus, beschränkt werden. Dieses Problem ist nicht allgemein lösbar, da dazu die Rekurrenzrelation der Funktionen analysiert werden müßte, was auch das Halteproblem einschließt. Es existiert allerdings eine große Klasse von Funktionen, für die eine gute, praktisch einsetzbare Approximation gefunden werden kann.

6.1.2.1 Homomorphe Interpretationen

Definition 6.2 (Homomorphe Interpretation)**.** Eine Interpretation $\mathbb{C}$ heißt *homomorph*, wenn der referentielle Anteil ihrer Ausgabe stets nur von der Eingabe unmittelbar erreichbare Elemente enthält:

$$x \rightarrow_{\mathbb{C}} y \implies \bar{\mathfrak{s}}(In)(y) \subseteq \mathcal{P}(\mathfrak{s}_{\mathbb{A}})\big(\bar{\mathfrak{s}}(In)(x)\big)$$

Im Fall einer Funktion, die durch homomorphe Interpretationen definiert ist, kann die erforderliche Information über die Rekurrenzrelation an den Referenzstrukturen im Speicher abgelesen werden.

Satz 6.3. *Sei (C, γ) eine homomorphe Interpretation. Sei x eine Eingabe, die zu einem Zyklus führt. Falls x überhaupt Referenzen enthält, liegt ein Zyklus im Speicher zugrunde:*

$$x \rightarrow_{\mathbb{C}}^{+} x \wedge \bar{\mathfrak{s}}(In)(x) = \emptyset \implies \exists\, x_0 \in \bar{\mathfrak{s}}(In)(x).\ x_0 \rightarrow_{\mathbb{A}}^{+} x_0$$

Beweis. Durch Anwendung von Definition 6.2 auf eine Kante $y \rightarrow_{\mathbb{C}} y'$ des Zyklus gilt:

1. Wenn y' Referenzen enthält, dann auch y.
2. Für jede Referenz r' in y' existiert eine Referenz r in y, so daß $r \rightarrow_{\mathbb{A}} r'$.

Im transitiven Abschluß über den gesamten Zyklus ergibt sich: Für jede Referenz r' in x existiert eine Referenz r in x, so daß $r \rightarrow_{\mathbb{A}}^{+} r'$. Die Menge $\bar{\mathfrak{s}}(In)(x)$ solcher Referenzen ist aber endlich, und kann daher nicht zyklenfrei sein. □

Die in Kapitel 4 verwendeten Beispielinterpretationen $\mathbb{S}$ und $\mathbb{L}$ sind homomorph. Viele weitere rekursive Standardfunktionen haben eine homomorphe Darstellung. In den Implementierungen der in Teil IV vorgestellten komplexeren Beispiele sind diese hervorgehoben und im Quelltext mit dem Attribut `hom` markiert. Die Implementierungen (Interpreter und Compiler) der in Teil III spezifizierten virtuellen Maschine erkennen dieses Attribut und beherrschen die hier beschriebenen Optimierungen.

6.1.2.2 Homomorphie und die Größe von Zyklen

Die Homomorphie-Eigenschaft bedeutet, daß algorithmisch produzierte Zyklen kausal abhängig sind von im Vorzustand bereits im Speicher vorhandenen Zyklen. Diese Abhängigkeit geht soweit, daß eine Relation zwischen den Größen von Ein- und Ausgabezyklen hergestellt werden kann. Dazu müssen wir Zyklen zunächst quantifizieren, wie von Graphen gewohnt.

Definition 6.4. Sei $\mathbb{C} = (C, \gamma)$ eine Coalgebra.

1. Eine Liste $[x_0, \ldots, x_n] \in C^*$ von Elementen heißt $\mathbb{C}$-*Pfad*, wenn gilt: $x_0 \rightarrow_{\mathbb{C}} x_1 \rightarrow_{\mathbb{C}} \cdots \rightarrow_{\mathbb{C}} x_n$.
2. Ein $\mathbb{C}$-Pfad heißt $\mathbb{C}$-*Zyklus*, wenn gilt: $x_0 = x_n$ und $n \neq 0$.
3. Ein $\mathbb{C}$-Zyklus heißt *minimal*, wenn keiner seiner echten Teilpfade ein $\mathbb{C}$-Zyklus ist.
4. Für minimale Zyklen heißt n die *Größe* des Zyklus.

□

Die Details der Größenrelation hängen von der Rekurrenzrelation der Interpretation ab. Hier einige einfache Fälle, bei denen stets $\mathbb{C} = (C, \gamma)$ eine $(In \rightarrow \Sigma)$-Interpretation über dem Σ-Speicherzustand $\mathbb{A}$ sei:

1. Für alle $x \in C$ sei $|\bar{\mathfrak{s}}(In)(x)| \leq 1$. Das ist insbesondere erfüllt, wenn In vom Grad 1 ist. Die Eingabe der Interpretation verweise also auf höchstens eine Speicherzelle. Dann entspricht jedem $\mathbb{C}$-Zyklus ein $\mathbb{A}$-Zyklus gleicher Größe.
2. In mehrdimensionalen Fällen, also bei höheren Graden von In, können sich die Größen von Zyklen multiplikativ kombinieren. So entsteht beim simultanen corekursiven Abstieg in mehreren zyklischen Argumenten ein Zyklus, dessen Größe sich nach dem *chinesischen Restsatz* bestimmen läßt, im Extremfall aber das kleinste gemeinsame Vielfache der einzelnen Größen beträgt.

6.1.2.3 Markierte Referenzen

Markiert man eine Auswahl von Referenzen im Speicher, so daß jeder Zyklus eine markierte Referenz enthält, dann kann die Zyklenerkennung auf Schritte beschränkt werden, die mindestens eine markierte Referenz überschreiten, ohne daß die Termination gefährdet wird. Insbesondere können zyklenfreie Daten markierungsfrei repräsentiert und komplett ohne Zyklenerkennung abgearbeitet werden. Der Aufwand des Tests auf Markierung ist bei geeigneter Implementierung sehr gering.

Die 1-Bit-Markierung eines Speichzustands läßt sich leicht modellieren:

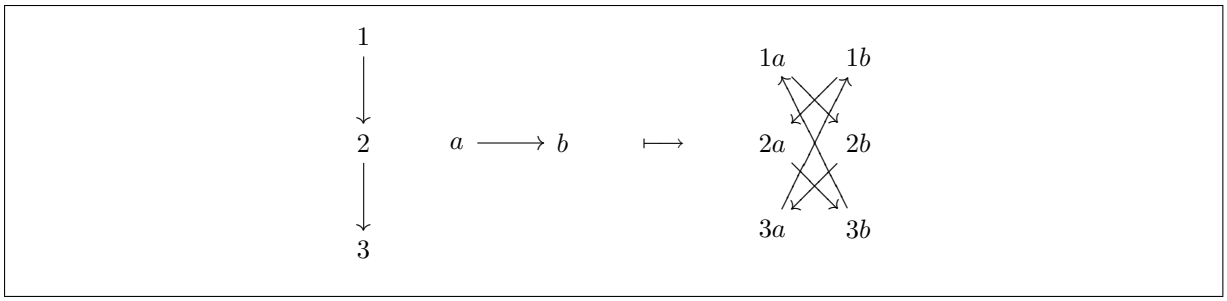


Abbildung 6.1: Multiplikation von Zyklen

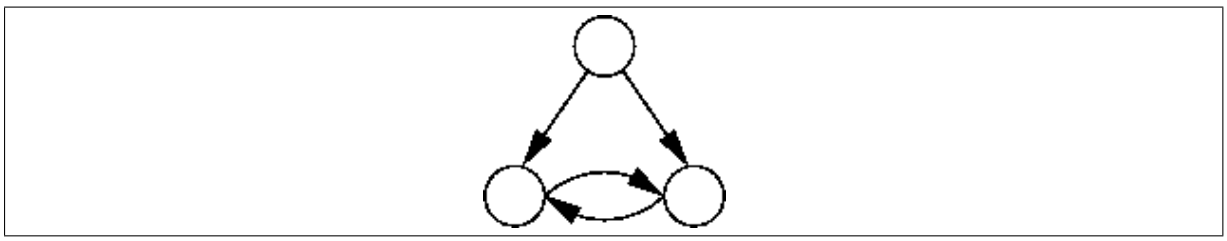


Abbildung 6.2: Zyklus mit mehreren Eintrittspunkten

Definition 6.5 (Markierter Speicherzustand)**.** Sei $\mathbb{A} = (A, \alpha)$ ein Speicherzustand über dem Adressraum $\mathcal{A}$. Dann ist der *markierte* Speicherzustand $\dot{\mathbb{A}} = (\dot{A}, \dot{\alpha})$ über dem Adressraum $\dot{\mathcal{A}}$ wie folgt definiert:

$$\dot{\mathcal{A}} = \{0,1\} \times \mathcal{A}$$
$$\dot{A} = \{0,1\} \times A$$
$$\dot{\alpha}\big((i,x)\big) = \alpha(x)$$

Dabei heißt $(0,x)$ eine *unmarkierte* Referenz auf x, $(1,x)$ eine *markierte* Referenz auf x. □

Die übliche Implementierung auf physikalischen Maschinen besteht in der Verwendung eines Bits des Adresswortes. Da die meisten aktuellen Prozessoren Wörter an geraden Byte-Adressen speichern, steht das niederwertigste Bit zu diesem Zweck zur Verfügung.

6.1.2.4 Markierungsstrategien

1. Jeder Zyklus enthalte eine markierte Referenz. Anders ausgedrückt, jeder unmarkierte Pfad sei zyklenfrei. Diese Eigenschaft ist für die Korrektheit der folgenden Optimierungen notwendig, und muß als Invariante jedes Ablaufs erhalten werden. Die folgenden Eigenschaften dagegen stellen lediglich Qualitätskriterien dar:

2. Jeder Zyklus enthalte möglichst wenige markierte Referenzen.

3. Der Abstand zwischen dem Ziel einer markierten Referenz und dem (Wieder-)Eintrittspunkt in den Zyklus sei möglichst gering, im Idealfall seien beide identisch.

Die letzteren beiden Eigenschaften besitzen nur dann ein gemeinsames globales Optimum, wenn Zyklen nicht mehrere Eintrittspunkte haben können (siehe Abbildung 6.2). Ob dies für reale Daten der Fall sein kann, ist stark anwendungsspezifisch.

Die Auswirkungen der Verletzung der beiden Qualitätskriterien sind dual:

1. Fehlplazierte Markierungen bewirken, daß die Zyklenerkennung zum falschen Zeitpunkt gestartet wird. Da der Zyklus zu diesem Zeitpunkt noch nicht vollständig ist, wird er weiter durchlaufen und

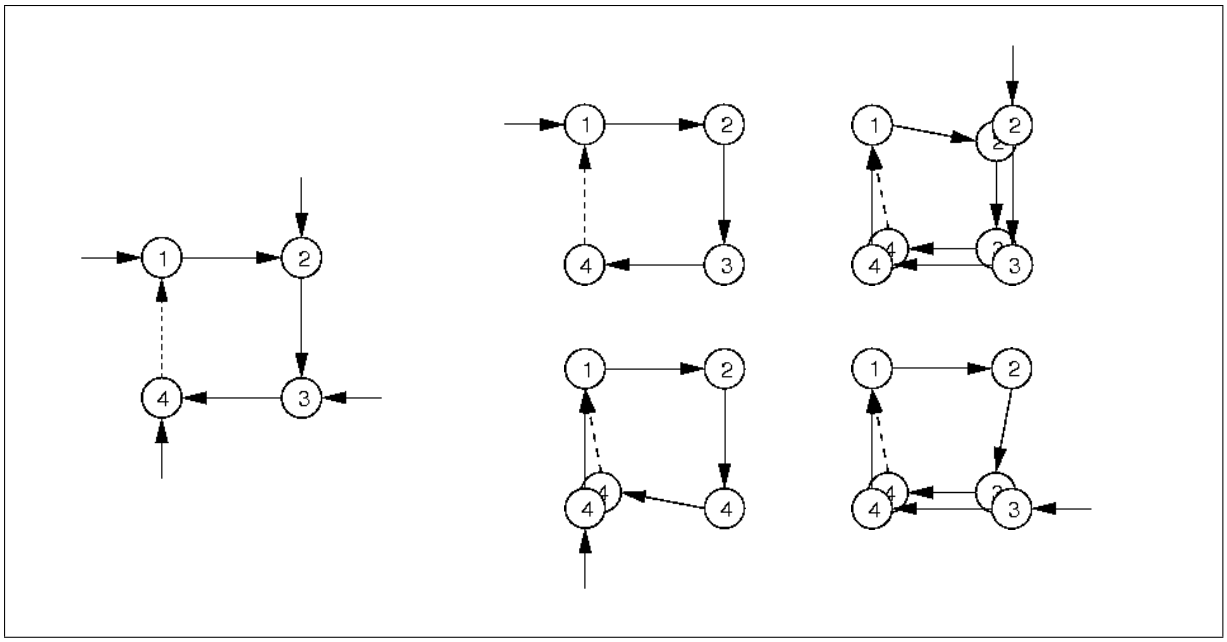


Abbildung 6.3: Plazierung von Markierungen

erst nach einer weiteren vollen Umdrehung erkannt und behandelt. Das kann dazu führen, daß redundante Repräsentationen im Speicher erzeugt werden, in denen ein Zyklus zum Teil „abgewickelt" wurde. Dabei handelt es sich um eine gemäßigte, weil endliche, Form der Auffaltung, wie sie bei der *lazy*-Erzeugung von Datenstrukturen auftreten kann.

2. Überflüssige Markierungen haben redundante Suche nach Zyklen und damit suboptimale Laufzeiten zur Folge. Andererseits kann diese Redundanz die Effekte von Fehlplazierung mildern.

Abbildung 6.3 zeigt links einen Zyklus mit vier Eintrittspunkten und einer Markierung. Rechts sind die vier Bilder der homomorphen Kopierfunktion, definiert durch die identische Interpretation, bei eingeschalteter Optimierung dargestellt. Man erkennt, daß der Präfix vor dem Ziel der markierten Referenz abgewickelt wird.

6.1.2.5 Effektive Markierung

Bleibt noch die Frage zu klären, inwieweit die Markierungsinvariante effektiv aufrechterhalten werden kann. In einem imperativen System, in dem Zyklen implizit durch wechselseitige Zuweisung entstehen, gibt es offensichtlich Probleme. Diese gelten aber nicht für einen funktionalen Kontext wie den hier beschriebenen. Die durch die hier betrachteten Algorithmen zugewiesenen Referenzen sind stets frisch und müssen nicht markiert werden. Die einzige Ausnahme ist die Zyklenbehandlung in Algorithmus Ana_3, die ja gerade zur Herstellung von Zyklen gedacht ist. Es genügt also, die bei der Zyklenbehandlung entstehenden Kopien zu markieren, um die Invariante zu sichern. Die so entstehende Verteilung der Markierungen genügt auch den oben aufgestellten Qualitätsanforderungen. Falls Zyklen nur einen Eintrittspunkt haben können, ist sie sogar optimal.

Als Nebenprodukt ergibt sich ein vollständiges Verfahren, um eine unmarkierte Datenstruktur zu markieren:

1. Als konservative Abschätzung markiere man zunächst *alle* Kanten.

2. Dann kopiere man die Struktur mit der corekursiven Kopierfunktion, die durch die identische Interpretation generiert wird.

Da die in Teil III vorgestellte virtuelle Maschine prinzipiell auch beliebige Zuweisungen gestattet, ist für solche das Meta-Attribut `tag` zulässig, um eine Kante eines ad-hoc, also durch wechselseitige Zuweisung gebildeten Zyklus zu markieren.

In Abschnitt 6.4.1 wird ein Verfahren vorgestellt, das eigentlich zur Speicherverwaltung gedacht ist. Es kann jedoch auch dazu verwendet werden, sicherzustellen, daß direkt vor dem Eintrittspunkt in einen Zyklus eine markierte Referenz liegt.

6.1.2.6 Homomorphe Kalküle

Für einen rational corekursiver Kalkül, dessen Ableitungsfunktion eine homomorphe Interpretation erzeugt, gelten dieselben grundsätzlichen Überlegungen. Auch hier kann eine Kantenmarkierung dazu dienen, überflüssige Zyklenerkennung einzusparen. Allerdings ist der Nutzeffekt hier im Vergleich zur Funktionsberechnung auf die Laufzeit eingeschränkt, da beim Entscheidungsalgorithmus nicht von einer Größenreduktion des Ergebnisses die Rede sein kann.

6.2 Effiziente Rekursion

Die unbeschränkte Wiederholung von Abläufen auf beschränkten Platz (Schleife) kann nur dann als Schachtelung von Funktionsaufrufen ausgedrückt werden, wenn nicht für jeden Aufruf frischer Speicherplatz auf dem Stack belegt wird. Um diesen Platzbedarf zu eliminieren, wird die Technik des *tail*-Aufrufs angewendet: Ist die letzte Operation eines Funktionsaufrufs ein weiterer Funktionsaufruf, dann kann dieser den Stack-Speicher seines Vorgängers „erben“, falls der Speicherbedarf beider kompatibel ist. Ob Kompatibilität für jede Kombination von Aufrufer und Aufgerufenem besteht, ist von der Implementierung abhängig. Die Sprache SCHEME beispielsweise schreibt allgemeine Kompatibilität in ihrer Spezifikation [26] ausdrücklich vor. Sind Aufrufer und Aufgerufener identisch, ist die Kompatibilität fast in allen Implementierungen gewährleistet. Die Behandlung von solchen direkten *tail*-Rekursionen ist daher Bestandteil optimierender Compiler für alle gängigen Sprachen.

Die typisch coalgebraische Auswertungsstrategie von der Wurzel zu den Blättern hat gravierende Auswirkungen auf die Möglichkeit von *tail*-Aufrufen:

1. Einerseits ist diese Reihenfolge der Operationen natürlicherweise bestens für diese Optimierung geeignet, da Konstruktoren stets vor den corekursiven Unteraufrufen ausgeführt werden. Für jede corekursive Funktion kann daher ein Ast der Rekursion in *tail*-Position gebracht werden. Im Fall einer linearen Rekursion kann sogar die gesamte Berechnung *tail*-rekursiv als Schleife ausgeführt werden.

   Im Gegensatz dazu steht bei algebraischen Auswertungsstrategien der Konstruktor der Ergebniswurzel an letzter Stelle. Die Funktion muß auf imperativer Ebene transformiert werden, um zu einer *tail*-rekursiven Form zu gelangen. Diese Transformation beherrschen nicht nur optimierende Compiler für deklarative Sprachen, sondern auch versierte Programmierer imperativer Sprachen. In C/C++ gehört sie geradezu zum guten Ton. Siehe [40] für Beispiele aus der Listenprogrammierung.

2. Andererseits steht die aufrufende Funktionsinkarnation nach dem Überschreiben ihres Stack-Speichers nicht mehr für die Zyklenerkennung zur Verfügung. Obwohl sich für corekursive Abläufe *tail*-Aufrufe geradezu aufdrängen, können sie nur dann sicher verwendet werden, wenn ausgeschlossen ist, daß der Aufrufer den Eintrittspunkt in einen Zyklus darstellt.

Die Lösung dieses Dilemmas besteht in einer Erweiterung der im letzten Abschnitt vorgestellten Optimierung. Man markiere Speicherzellen, falls auf sie mit markierten Referenzen verwiesen wird. Dies kann durch exakte Referenzzählung geschehen, oder durch ein einzelnes Bit. Im zweiten Fall kann ein Überlauf des 1-Bit-Zählers nicht erkannt werden, und daher die Zellenmarkierung beim Löschen einer markierten Referenz nicht zurückgesetzt werden. Im Kontext referentiell transparenter Abläufe stellt das aber kein Problem dar, und das Löschen überflüssiger Zellenmarkierungen kann der automatischen Speicherverwaltung überlassen werden.

Falls nun alle Zellen, auf die eine markierte Referenz verweist, markiert sind, dann können durch homomorphe Interpretationen generierte Funktionen weiter optimiert werden. Wird nämlich die Zyklenerkennung für alle Aufrufe, die keine markierte Referenz überschreiten, unterdrückt, dann gilt das insbesondere für Aufrufe, die nur unmarkierte Zellen als Eingabe erhalten. Wenn für solche Aufrufe aber keine Zyklenerkennung stattfindet, dann müssen sie auch nicht für die gesamte Dauer (co)rekursiver Unteraufrufe auf dem Stack verzeichnet bleiben. Also kann in diesem Fall der letzte Unteraufruf als *tail*-Aufruf ausgeführt werden.

Da zyklenfreie Daten aus unmarkierten Zellen bestehen, sind hier immer *tail*-Aufrufe möglich. Insbesondere kann hier lineare Rekursion, implementiert als *tail*-Rekursion, als echte Schleife ablaufen.

6.3 Erkennung von Quasizyklen

Eine abstrakte Interpretation auf einem festen Speicherzustand ist notwendigerweise rational, so daß Berechnungen mit Zyklenerkennung auch terminieren. In einer Implementierung kann es aber zweckmäßig sein, daß die Ausführung eines Interpretationsschritts neue Zellen erzeugt. Die Spezifikation in

Kapitel 4 sieht diesen Fall ausdrücklich vor. Kommen Referenzen auf solche frischen Zellen im Ergebnis eines Schritts vor, dann werden diese von folgenden Schritten traversiert. So ist es möglich, daß zwar die finale Semantik der Interpretation rational ist, aber auf der konkreten Ebene eine unendliche Spirale von verschiedenen Repräsentanten periodisch wiederkehrender Bisimilaritätsklassen, ein sogenannter Quasizyklus, entsteht. In diesem Fall versagt die Zyklenerkennung auf der konkreten Ebene.

Für polynomielle Datentypen existiert aber eine generisches Entscheidungsverfahren für Bisimilarität. Dieses kann mit den bereits bekannten Mitteln implementiert werden, nämlich als größter Fixpunkt eines Kalküls, der die aus Abschnitt 2.4.2.5 bekannte, generische Bisimulationsoperation $v(\Sigma)$ spezifiziert. Somit ist es möglich, die Zyklenerkennung nicht über einen Referenzvergleich, sondern modulo Bisimilarität durchzuführen. Da der Aufwand um ein Vielfaches größer ist, sollte diese Verallgemeinerung auf notwendige Fälle beschränkt werden. Die im folgenden Teil beschriebene virtuelle Maschine besitzt dafür ein Meta-Attribut `quasi`, mit dem für eine gegebene Funktion die Zyklenerkennung modulo Bisimilarität aktiviert wird.

Ein typisches Beispiel für quasizyklische Berechnung wird der bereits aus Kapitel 1 bekannte und in Abschnitt A.3.2 allgemein spezifizierte rationale Divisionsalgorithmus darstellen.

6.4 Effektive Speicherverwaltung

In diesem Abschnitt soll der Einfluß der typisch corekursiven Berechnungsstrategien auf automatische Speicherverwaltung untersucht werden. Dabei wird sich zeigen, daß sich für jeden Ansatz spezifische Anforderungen ergeben, die für algebraisch funktionale oder objektorientierte Sprachen typischerweise nicht gelten.

6.4.1 Referenzzählung

Obwohl referenzzählende Speicherverwaltung von den wenigsten Plattformentwicklern als zeitgemäß betrachtet wird, ist sie doch aus vielen Bereichen, wie etwa Echtzeit- und verteilten Systemen, nicht wegzudenken. Bekanntermaßen ist einfaches Zählen von Referenzen nicht geeignet, um zyklische Daten zu verwalten, da in streng zusammenhängenden Komponenten die Zähler niemals auf Null sinken. Das Problem kann auf verschiedene Weisen behoben werden:

1. Referenzzählen wird nur zur schnellen lokalen Freigabe azyklischer unerreichbarer Zellen verwendet. Unerreichbare Komponenten werden periodisch mit einen zweiten, globalen Verfahren gesucht und eliminiert.

2. Beim Löschen einer Referenz, die den Eintrittspunkt in eine Komponente darstellen könnte, werden die inneren Referenzen der Komponente spekulativ entfernt. Werden dabei alle Zellen isoliert, ist die Komponente unerreichbar. Ansonsten müssen die inneren Referenzen wieder hergestellt werden. Die Untersuchung der potentiell unerreichbaren Komponente geschieht üblicherweise durch lokale Tiefensuche, entweder sofort wie bei MARTÍNEZ[35] oder verzögert wie bei LINS[33].

Kritisch bleibt beim zweiten Ansatz die Entscheidung, wann eine Komponente unerreichbar werden könnte. Im allgemeinen Fall ist dies immer möglich, wenn ein Referenzzähler auf eine Zahl größer Null sinkt. Mit Hilfe der Markierungen aus Abschnitt 6.1.2 läßt sich die Abschätzung verbessern, indem heuristisch einige Situationen von der Pflicht zur lokalen Suche ausgenommen werden, ohne daß der letzte Eintrittspunkt der Komponente übersehen wird.

Die heuristische Verfeinerung läßt sich am besten formal als Graphalgorithmus beschreiben. Sei (V, E, s, t) ein gerichteter Graph mit einer *Knotenmenge* V, einer *Kantenmenge* E sowie zwei *Inzidenzabbildungen* $s, t : E \to V$ für Start- und Zielknoten einer Kante. Die Umkehrungen dieser Abbildungen schreiben wir als $in, ex : V \to \mathcal{P}(E)$, wobei für alle $v \in V$ gilt:

$$in(v) = \{e \in E \mid t(e) = v\} \qquad ex(v) = \{e \in E \mid s(e) = v\}$$

Dabei nennen wir $in(v)$ die *einlaufenden* und $ex(v)$ die *auslaufenden* Kanten. Damit können die zwei zentralen Relationen definiert werden, die *Erreichbarkeit* $(\to)$ und der *starke Zusammenhang* $(\approx)$.

$$v \to v' \iff ex(v) \cap in(v') \neq \emptyset \qquad v \approx v' \iff v \to^* v' \wedge v' \to^* v$$

Dabei ist $(\approx)$ eine Äquivalenzrelation. Die Klassen $[v]_\approx \subseteq V$ heißen *stark zusammenhängende Komponenten* des Graphen. Damit zerfallen die Kanten in *Intra-Kanten* $E_\approx$ und *Inter-Kanten* E_x:

$$E_\approx = \{e \in E \mid s(e) \approx t(e)\} \qquad E_x = \{e \in E \mid s(e) \not\approx t(e)\}$$

Von besonderem Interesse sind *zyklische* Komponenten, also solche, die Intra-Kanten enthalten. Außerdem definieren wir die Klasse der *Eingangsknoten* V_i:

$$V_i = \{v \in V \mid in(v) \cap E_x \neq \emptyset\}$$

Für die automatische Speicherverwaltung nehmen wir die Existenz eines ausgezeichneten *Wurzelknotens* $r \in V$ an. Die Aufgabe ist dann, die von r aus transitiv erreichbaren Komponenten von den unerreichbaren zu unterscheiden. Das Verfahren [35] löst dieses Problem inkrementell beim Löschen einer Kante e:

1. Bestimme den Zielknoten $v = t(e)$ und die Restkantenmenge $R = in(v) \setminus \{e\}$. Gilt $R = \emptyset$, dann wird die azyklische Komponente $\{v\}$ unerreichbar. Lösche rekursiv alle Kanten $e' \in ex(v)$.

2. Gilt aber $R \neq \emptyset$, dann traversiere den von v transitiv erreichbaren Subgraphen wie folgt:

   (a) Lösche zuerst spekulativ alle von v transitiv erreichbaren Kanten.

   (b) Gilt danach für ein v' mit $v \rightarrow^* v'$ immer noch $in(v') \neq \emptyset$, dann ist v Bestandteil einer anderweitig erreichbaren zyklischen Komponente. Füge dann alle von v' transitiv erreichbaren spekulativ gelöschten Kanten wieder hinzu.

   (c) Alle v', für die jetzt $in(v') = \emptyset$ gilt, sind unerreichbar.

In der Praxis werden nicht die Mengen R und $in(v')$ bestimmt, sondern lediglich ihre Größe durch einen *Referenzzähler* $rc(v) = |in(v)|$ verfolgt, der auf Null getestet wird.

Das Löschen einer Kante, auf deren Zielknoten noch weitere Kanten verweisen, kann im Fall $R \neq \emptyset$ potentiell sehr viel, nicht immer produktive Arbeit bedeuten. Dabei kann eine erreichbare Komponente durch das Löschen einer Kante nur dann unerreichbar werden, wenn es sich um eine Inter-Kante handelt. Leider kann diese Frage nur durch eine globale Traversierung des Graphen entschieden werden. Da es aber gerade um die Entwicklung eines lokalen Verfahrens geht, werden wir uns mit einer Approximation behelfen. Dazu führen wir eine zweite Partitionierung der Kanten in einen *positiven* Anteil E_p und einen *negativen* Anteil E_n ein. Anders als bei dem ähnlichen Ansatz von BROWNBRIDGE[7] ist diese Aufteilung willkürlich und nicht unmittelbar durch die Graphstruktur bestimmt. Das spekulative Löschen von Kanten soll nur dann stattfinden, wenn $R \subseteq E_n$. Ein Knoten wird also willkürlich als erreichbar angesehen, solange eine positive Kante auf ihn verweist. Technisch bedeutet dieser Eingriff die Aufteilung des einen Referenzzählers $rc(v)$ in positiven und negativen Anteil:

$$rc_p(v) = |in(v) \cap E_p| \qquad\qquad rc_n(v) = |in(n) \cap E_n|$$

Die Aufteilung in positive und negative Kanten ist natürlich keineswegs beliebig:

1. Offensichtlich ist das Verfahren nicht korrekt, wenn ein Zyklus nur aus positiven Kanten besteht.

2. Andererseits bringt jede positive Kante einen Effizienzgewinn.

Im folgenden soll also eine Strategie zur Aufteilung in positive und negative Kanten gefunden werden, die korrekt ist, möglichst viele positive Kanten zuläßt und bei Veränderungen des Graphen inkrementell mit geringem Aufwand angepaßt werden kann. Dazu definieren wir zunächst die Klassen der *skeptischen* Knoten V_s und der *negativen* Knoten V_n:

$$V_s = \{v \in V \mid in(v) \cap E_p = \emptyset \Rightarrow ex(v) \cap E_p = \emptyset\}$$
$$V_n = \{v \in V \mid in(v) \cap E_n \neq \emptyset\}$$

Ein Knoten ist also skeptisch, falls er nur dann positive auslaufende Kanten besitzt, wenn er auch positive einlaufende Kanten besitzt, und negativ, falls er negative einlaufende Kanten besitzt. Außerdem definieren wir *positive Erreichbarkeit*:

$$v \rightarrow_p v' \iff ex(v) \cap in(v') \cap E_p \neq \emptyset$$

Nun stellen wir zwei Invarianten auf:

$$v \not\rightarrow_p^+ v \qquad\qquad V_n \subseteq V_s$$

Das bedeutet im Klartext:

1. Es sollen also keine ausschließlich positive Zyklen existieren. Diese erste Invariante muß bei der Konstruktion eines Zyklus erfüllt werden, beispielsweise, indem die bereits aus Abschnitt 6.1.2 bekannten Markierung die negativen Kanten bestimmt.

2. Alle negativen Knoten sollen skeptisch sein. Diese zweite Invariante muß von einem Knoten v aus propagiert werden, wenn die erste negative Kante auf v eingefügt oder die letzte positive Kante auf v gelöscht wird. Dabei sind alle auslaufenden positiven Kanten durch negative zu ersetzen. Diese Propagation ist effizienter als das spekulative Löschen:

   (a) Sie setzt sich potentiell rekursiv fort, stoppt aber bei Knoten mit einlaufenden oder ohne auslaufende positive Kanten.

   (b) Jede Kante wird während ihrer Lebensdauer höchstens einmal ersetzt. Eine weitere Markierung ist weder während der Propagation noch danach nötig.

   Die Propagation läßt sich in die zweite Phase des spekulativen Löschens integrieren, indem vormals positive Kanten als negative wiederhergestellt werden, solange kein Knoten mit einlaufenden positiven Kanten auf dem Weg liegt.

Damit sind alle Ziele bis auf den Nachweis der Korrektheit erreicht. Um diesen anzutreten, definieren wir die zusätzliche Klasse der *verdächtigen* Knoten $V_?$:

$$V_? = \{v \in V \mid in(v) \cap E_\approx \subseteq E_n\}$$

Ein Knoten ist also verdächtig, wenn alle einlaufenden Intra-Kanten negativ sind. Dann wird spätestens beim Löschen der letzten einlaufenden Inter-Kante das spekulative Löschen ausgelöst, und die Unerreichbarkeit erkannt. Es ist also zu zeigen, daß der letzte erreichbare Eingangsknoten einer Komponente verdächtig ist.

Dazu betrachten wir die Relation $\rightarrow_p^+$ auf den Knoten einer zyklischen Komponente W. Aufgrund der ersten Invariante ist diese eine partielle Ordnung auf W. Nun zeigen wir induktiv, daß alle diesbezüglich minimalen Eingangsknoten verdächtig sind:

1. Die absolut minimalen Knoten sind per Konstruktion negativ und verdächtig.

2. Sei v_i ein Eingangsknoten und alle Intra-Vorgänger $v \in W$ mit $v \rightarrow_p v_i$ per Induktionsannahme negativ, verdächtig und keine Eingangsknoten. Folglich besitzten sie keine positiven einlaufenden Kanten. Nach der zweiten Invariante sind sie als negative Knoten skeptisch. Also besitzt v_i keine positiven einlaufenden Intra-Kanten und ist damit verdächtig.

Eine Komponente und ihre auslaufenden Inter-Kanten werden gelöscht, sobald die Komponente unerreichbar ist. Alle verbleibenden Inter-Kanten tragen also notwendigerweise zur Erreichbarkeit bei. Wird eine Komponente durch das Löschen einer Inter-Kante unerreichbar, dann verweist diese zwangsläufig auf einen minimalen und damit verdächtigen Eingangsknoten. Zur weiteren Effizienzsteigerung kann die Spekulation auch noch wie bei [33] verzögert werden.

6.4.2 Globale Suche

Anstelle von Referenzzählern können auch globale Erreichbarkeitsanalysen durch Traversierung zur Erkennung unerreichbarer Zellen eingesetzt werden. Zyklen verursachen hier deutlich geringere Probleme. Außerdem kann eine solche Traversierung mit dem Verschieben der erreichbaren Zellen kombiniert werden, wodurch die Freigabe des restlichen Speichers trivial wird. Darauf basieren die kompaktierende und die kopierende Speicherverwaltung, die zu den beliebtesten Verfahren zählen.

Sowohl für die Erreichbarkeitsanalyse, als auch für das Verschieben von Zellen ist es problematisch, wenn Referenzen nicht auf Zellen, sondern auf deren Bestandteile verweisen, wie es bei den in Kapitel 4 allgegenwärtigen Variablenadressen der Fall ist. Dieses Problem kann auf zwei Arten gelöst werden:

1. Eine Referenz auf eine einzelne Variable innerhalb einer Zelle wird aufgeteilt in eine Referenz auf die Zelle und einen Index, der die Variable selektiert. Dieser Ansatz ist allgemein und funktioniert auch auf Plattformen, die (eben aus Gründen der Speicherverwaltung) die Adressierung einzelner

Variablen gar nicht zulassen, wie etwa die JVM. Der Preis dafür ist die Interpretation des Index zur Laufzeit. Im Fall der JVM bedeutet das eine Fallunterscheidung oder ein Array-Zugriff mit Intervallprüfung.

2. Im speziellen Fall transparenter Abläufe können zwei legale Verwendungen von solchen Teilreferenzen unterschieden werden:

   (a) Während der Initialisierung einer Zelle existieren Teilreferenzen, die für Schreibzugriffe genutzt werden. Da jede erzeugte Zelle irgendwann vor ihrem Tod vollständig initialisiert wird, ist es zulässig, unvollständig initialisierte Zellen von Freigabe und Verschiebung auszuschließen. Dann können die Teilreferenzen in dieser Phase für die Erreichbarkeitsanalyse ignoriert werden.

   (b) Nach Initialisierung einer Zelle können Teilreferenzen höchstens noch für Lesezugriffe verwendet werden. Diese können eliminiert werden, falls es gelingt, alle Lesezugriffe in einem Ablauf als atomare Lesezugriffe auf Zellenreferenzen (Anwendungen von α) darzustellen. In einer funktionalen Eingabesprache, die keine Variablenadressen kennt, ist dies leicht zu realisieren.

6.4.3 Zellenmarkierung

Für die Optimierung homomorpher Berechnungen ist es wichtig, daß eine Zelle die Information trägt, ob eventuell markierte Referenzen auf sie verweisen.

1. Bei der Verwendung einer referenzzählenden Speicherverwaltung kann diese Information, wie in Abschnitt 6.4.1 beschrieben, als separater Referenzzähler verwaltet werden.

2. Bei einer traversierenden Speicherverwaltung bietet es sich dagegen an, als Approximation einen sogenannten 1-Bit-Referenzzähler zu verwenden. Dieser wird gesetzt, sobald eine markierte Referenz erzeugt wird. Offensichtlich sind alle Zählerstände größer Null ununterscheidbar, und der Zähler kann nicht zurückgesetzt werden, wenn eine markierte Referenz gelöscht wird. Die Erreichbarkeitsanalyse der Speicherverwaltung kann aber leicht um diese Aufgabe erweitert werden, indem zusätzlich zur Erreichbarkeit auch die unmittelbare Erreichbarkeit über eine markierte Kante berechnet wird.

Teil III

Praktische Realisierung

I wish to God these calculations had been executed by steam.
Charles Babbage[4]

In diesem Teil der Arbeit soll die in Teil II entwickelte Theorie zyklischer Funktionen und Prädikate in ein operationales Berechnungsmodell umgesetzt werden. Damit soll es ermöglicht werden, die theoretischen Algorithmen für die Programmierung praktisch verfügbar zu machen, ohne daß Programme in der rigiden Form von Speicher-Interpretationen formuliert werden müssen. Anstelle von universellen Algorithmen und deren Parametrisierung mit Interpretationen wird eine imperative Sprache gesetzt, in der konkrete Instanzen der Algorithmen beschrieben werden können.

Dazu wird zunächst in Kapitel 7 eine virtuelle Maschine spezifiziert, deren Entwurf sich weitgehend aus den Anforderungen der Theorie ergibt. Diese definiert ein objektorientiertes Modell von Programmen und deren operationale Semantik. Programme können in einer textuellen Eingabesprache denotiert werden, die die Struktur und den Befehlssatz der Maschine abbildet. Auch wenn dabei einige Zugeständnisse an die Ergonomie für den menschlischen Leser und Verfasser gemacht werden, handelt es sich nicht um eine vollwertige Programmiersprache, sondern um eine rohe technische Notation, wie sie der Bedienung eines Forschungsprototypen angemessen ist. Ein vorgeschalteter Compilationsschritt, der Maschinenprogramme aus einer benutzerfreundlichen Hochsprache generiert, ist nicht Gegenstand dieser Arbeit. Hier sind zuvor noch einige offene Probleme zu lösen, wie in Kapitel 9 ausgeführt wird.

Die Realisierung der Maschine V→M und ihre Erweiterung zu einer Programmierumgebung ist, wie in Kapitel 8 beschrieben, in Java implementiert und umfaßt:

1. eine Implementierung des Programmmodells,
2. eine Erweiterung um ein Speichermodell zu einem Zustandsmodell,
3. einen Interpreter zur Simulation von Abläufen auf dem Zustandsmodell,
4. einen Parser der Eingabesprache TASM,
5. einen optimierenden Compiler in die Zwischenrepräsentation IMPL,
6. einen Codegenerator, der IMPL in C übersetzt,
7. einen *just-in-time*-Compiler für IMPL als Ergänzung des Interpreters,
8. eine grafische Benutzeroberfläche zum interaktiven Experimentieren mit allen Werkzeugen.

Das Gesamtsystem hat den Namen Malice, der in Abschnitt 9.2.5.1 erklärt wird. Außerdem existieren diverse Beispielprogramme, von Standardbeispielen der funktionalen Programmierung bis hin zu komplexen Anwendungen zyklischer Berechnungen, wie sie in Teil IV beschrieben sind.

Kapitel 7

Virtuelle Maschine

Die Spezifikation der virtuellen Maschine V→M erfolgt simultan auf mehreren Ebenen:

1. semiformale Spezifikation der Maschine in Prosa und mathematischer Notation,
2. formale Spezifikation der Maschine durch eine Referenzimplementierung in Haskell,
3. formale Spezifikation der textuellen Eingabesprache TASM in einer Form von EBNF,
4. informative Beispiele zur Maschinenspezifikation in TASM.

Um die Zuordnung einzelner Fragmente zu den Ebenen zu erleichtern, wird folgende Konvention verwendet:

1. Haskell-Code steht, wie in allen bisherigen Kapiteln, nur in optisch abgegrenzten Schaukästen.
2. TASM-Code steht in explizit als Beispiel ausgewiesenen Absätzen.
3. EBNF-Regeln stehen wie Formeln im Fließtext.

Die EBNF-Regeln verwenden die Superskripte $^?, ^+$ für optionale und wiederholbare Elemente, das Superskript * für $^{+?}$, sowie die Notation $(a\#b)^+$ für Sequenzen der Form $abab\cdots ba$.

Die Komplexität der Haskell-Programmfragmente wächst naturgemäß mit dem Übergang von Theorie zu Praxis. Soweit wie möglich wird dem durch Aufteilung in kleinere Fragmente entgegengewirkt. Aus demselben Grund wurde auf eine aktive Diagnose von Programmfehlern verzichtet, so daß der Ablauf nicht wohlgeformter Programme undefiniert ist oder ohne genaue Fehlermeldung abbricht.

7.1 Anforderungen

Aus den vorhergehenden Kapitel ergeben sich explizit oder implizit einige Anforderungen für den Entwurf der virtuellen Maschine:

1. Zyklische Berechnungen wurden bisher deklarativ dargestellt. Die Maschine soll aus diesem Grund und generell im Sinne des Programmierers einen deklarativen Programmierstil so weit wie möglich unterstützen.
2. Das typisch corekursive Vorgehen, bei dem Erzeugung und Initialisierung von Zellen voneinander getrennt ablaufen, und bei dem jede Berechnung explizit im Kontext einer das Ergebnis aufnehmenden Variablen steht, ist nicht direkt vereinbar mit bekannten abstrakten Maschinenparadigmen wie Graphreduktion.
3. Die Erkennung und Behandlung von Zyklen soll integraler Bestandteil des Maschinenkonzepts sein.

Daraus lassen sich schon im Vorfeld des Entwurfs Schlüsse ziehen:

1. Anders als ein primitiv rekursiver Ablauf, der gut als Reduktion vorgestellt werden kann, läßt sich ein primitiv corekursiver Ablauf am natürlichsten als Sequenz von Zuweisungen begreifen. Daher ist ein grundsätzlich imperatives Maschinenparadigma naheliegend.

2. Damit die Beschreibungsebene nicht so niedrig wird, daß ein Maschinenprogramm vom Menschen nicht mehr mit befriedigender Produktivität geschrieben werden kann, sollte die Maschine mit wohlverstandenen Abstraktionsmitteln aufgewertet werden:

   (a) Statische Typisierung und benutzerdefinierte Datentypen abstrahieren vom Problem der Codierung von Daten.

   (b) Parametrische Polymorphie und Funktionen höherer Ordnung erlauben Wiederverwendung algorithmischer Bausteine.

   (c) Speicherverwaltung sollte implizit bleiben und in Implementierungen der Maschine automatisch gelöst werden.

3. Die corekursive Konstruktion, also Erzeugung partiell uninitialisierter Daten und die Delegation der Initialisierung sollte ebenso gekapselt werden können wie die Berechnung totaler Daten in Funktionen gekapselt wird. Beide Sichtweisen sollen möglichst durch geeignete Aufrufkonventionen integriert werden.

4. Die Delegation von Zuweisungen erfordert die universelle Adressierbarkeit von Variablen (*first-class references*). Die Typsicherheit soll dadurch nicht verletzt werden.

5. Auch wenn Funktionen höherer Ordnung unterstützt werden, basiert die Zyklenerkennung doch auf einem Identitätsbegriff, der nur bei Funktionen erster Ordnung wiederzufinden ist. Diese sollten daher die dominierende Sichtweise auf ein Maschinenprogramm darstellen.

Eine Maschine, die diese Anforderungen weitestgehend erfüllt, soll im Rest dieses Kapitels vorgestellt werden. Die praktische Eignung ist durch die umfangreiche Beispielsammlung, welche mit der Implementierung ausgeliefert wird, belegt.

```
data Memory = Memory {
  memHeap      :: Heap Univ,
  memStack     :: Stack,
  memConstants :: FiniteMap Constant Value
}
emptyMemory = Memory {
  memHeap      = emptyHeap,
  memStack     = emptyStack,
  memConstants = emptyFM
}
```

Programm 7.1: Speicher der virtuellen Maschine

7.2 Architektur

7.2.1 Speicher

Der Zustandsraum der virtuellen Maschine wird im Wesentlichen durch drei Speicherbereiche aufgespannt:

1. Der globale Arbeitsspeicher (*Heap*), in dem strukturierte Daten wie Tupel, Cotupel und partielle Prozedurinkarnationen (*Closures*) gespeichert werden.

2. Der lokale Arbeitsspeicher (*Stack*), in dem temporäre Werte wie Bindungen von Parametern und lokalen Variablen und anonyme Zwischenergebnisse in Form von primitiven Werten und Referenzen gespeichert werden.

3. Das Programm, ein abstrakter Syntaxgraph der definierten Typen, Konstanten und Prozeduren.

Im stark idealisierten Grundmodell sind Heap und Stack unbeschränkt. Im Sinne einer Ressourcenbeschränkung kann angenommen werden, daß der Heap automatisch verwaltet wird, wobei nicht mehr erreichbare Zellen wiederverwendet werden können, sobald ihr Zustand erkannt worden ist (*garbage collection*). Werte auf dem Stack dagegen besitzen eine klar definierte maximale Lebensdauer, nach deren Ablauf der Speicherbereich sofort wieder zur Verfügung steht. In wieweit das Programm einer dynamischen Veränderung unterliegen kann, und ob diese eine monotone Erweiterung sein muß, bleibt hier unspezifiziert. Ebenso ist unspezifiziert, ob das Programm einen eigenen Speicherbereich darstellt, oder ob es ganz oder in Teilen auf Heap oder Stack abgebildet wird.

Programm 7.1 zeigt den entsprechenden Haskell-Datentyp. Bei *Heap* handelt es sich um die aus Kapitel 3 bekannte Struktur. Die Typen *Univ*, *Stack*, *Constant* und *Value* werden in den folenden Abschnitten definiert.

7.2.1.1 Heap

Der Heap-Speicher der Maschine entspricht ganz der Modellierung von Kapitel 3. An Stelle von spezifischen Signaturen für einzelne Datentypen wird eine universelle Signatur verwendet. Diese unterscheidet drei Klassen von Zellen:

Tupel entsprechen den Zellen zu Signaturen der Form $\prod_{i \in I} \mathcal{I}$. Ein Tupel enthält also eine I-indizierte Familie von Referenzen, der lokale Anteil ist leer.

Die Indexmenge eines Tupels ändert sich nach der Erzeugung nicht mehr. Konkrete Implementierungen können Tupel daher als zusammenhängende Speicherbereiche repräsentieren.

```
type Label                = String
type Family               = FiniteMap Label
data Univ α               = UTuple (Family α)
                          | UCotuple Label (Maybe α)
                          | UClosure Function [α]
instance Functor Univ       where ...
instance Signature Univ     where ...
```

Programm 7.2: Universelle Signatur der virtuellen Maschine

Cotupel entsprechen den Zellen zu Signaturen der Form $\coprod_{i \in I} \Sigma_i$, wobei $\Sigma_i = \mathcal{I}$ oder $\Sigma_i = \mathcal{C}_1$. Ein Cotupel enthält also einen Index aus I als lokalen Anteil, sowie höchstens eine Referenz, genannt der *Rumpf*.

Der Index eines Cotupels sowie das Vorhandensein eines Rumpfes ändert sich nach der Erzeugung nicht mehr. Konkrete Implementierungen können Cotupel ohne Rumpf daher als Aufzählungskonstanten ohne eigenen Speicherbereich repräsentieren.

Closures haben keine Entsprechung unter den polynomiellen Signaturen, sondern dienen der Kapselung von Funktionen und Funktionsapplikationen als Werten. Eine Closure enthält eine Prozedur als lokalen Anteil, sowie eine Liste von Referenzen, die den Präfix einer Belegung der Eingabeparameter der Prozedur darstellt.

Die Anzahl der Parameterwerte in einer Closure ändert sich nach der Erzeugung nicht mehr. Konkrete Implementierungen können Closures daher als spezielle Tupel repräsentieren.

Programm 7.2 zeigt die Modellierung des Signaturfunktors in Haskell. Indizes werden durch Strings dargestellt. Der Einfachheit halber wird angenommen, daß Closures stets durch eine einzige zusammmenhängende Zelle repräsentiert werden (*flache Closures*). Für eine praktische Implementierung kann unter Umständen eine rückwärts verkettete Aufteilung der Parameterliste auf mehrere Zellen (*geschachtelte Closures*) geeigneter sein. Für die Arbeitsweise der Maschine ist diese Unterscheidung irrelevant.

7.2.1.2 Stack

Der Stack-Speicher ist in *Frames* unterteilt, von denen jeder einer Prozedurinkarnation entspricht. Für eine genauere Spezifikation des Aufbaus sei auf Abschnitt 7.4 verwiesen.

Der Raum der auf dem Stack speicherbaren Werte ist eine Erweiterung dessen, was in Heap-Zellen gespeichert werden kann. Während Heap-Variablen stets nur auf Heap-Zellen verweisen, können Stack-Variablen sowohl auf Heap-Zellen, als auch auf Heap-Variablen und andere Stack-Variablen verweisen. Daraus ergibt sich als rudimentäres Typsystem eine Ordnungszahl der Referenz, die sowohl der Variablen als auch dem gespeicherten Inhalt zugeschrieben wird:

1. Heap-Variablen und Stack-Variablen, die auf Heap-Zellen verweisen, sind erster Ordnung.
2. Stack-Variablen, die auf Variablen n-ter Ordnung verweisen, sind $(n+1)$-ter Ordnung. Insbesondere sind Stack-Variablen, die auf Heap-Variablen verweisen, zweiter Ordnung.
3. Zyklische Verweise werden durch das Stack-Typsystem der Maschine ausgeschlossen, so daß jede Variable in einem typkorrekten Programm von endlicher Ordnung ist. Mehr dazu in Abschnitt 7.3

Referenzen bis zu dritter Ordnung finden in der Maschine selbst Verwendung. Dem Benutzer ist es prinzipiell gestattet, mit Referenzen höherer Ordnung zu arbeiten. Anwendungsfälle von praktischem Nutzen sind allerdings nicht leicht vorstellbar.

Programm 7.4 zeigt die Modellierung von Stack-Referenzen in Haskell.

```
data Stack = Stack {
  stackFrames   :: [Frame],
  stackNextSlot :: Slot,
  stackSlots    :: FiniteMap Slot Value
}
emptyStack = Stack {
  stackFrames   = [],
  stackNextSlot = Slot 0,
  stackSlots    = emptyFM
}
```

Programm 7.3: Der Stack der Maschine

```
newtype Slot = Slot Int deriving (Enum, Eq, Ord, Show)
data Value   = Plain { fromPlain :: Cell }
             | HeapRef { fromHeapRef :: Field }
             | StackRef { fromStackRef :: Slot }
               deriving (Eq, Ord, Show)
```

Programm 7.4: Werteuniversum der virtuellen Maschine

7.2.1.3 Programm

Programm 7.5 zeigt die Modellierung von Maschinenprogrammen. Dabei sind lediglich die global bekannten Objekte der drei Kategorien Typen, Konstanten und Funktionen direkt erfasst. Das eigentliche Programm umfaßt auch alle transitiv erreichbaren Objekte. Im Sinne einer modularen Sprache kann man den Datentyp *Program* als Exportschnittstelle eines Moduls auffassen.

In der Eingabesprache TASM wird ein Programm durch eine Folge von Deklarationen denotiert. Die Namen der deklarierten Gegenstände dienen als Trägermenge einer Coalgebra, in der Referenzen als Coterme aufgefaßt werden können.

$$
\begin{array}{lcl}
\mathit{unit} & \rightarrow & \mathit{decl}^* \\
\mathit{decl} & \rightarrow & \mathit{typedecl} \mid \mathit{constdecl} \mid \mathit{fundecl} \\
\mathit{defid} & \rightarrow & \mathit{id} \\
\mathit{refid} & \rightarrow & \mathit{id}
\end{array}
$$

```
data Program = Program {
  types     :: FiniteMap TypeId Type,
  constants :: FiniteMap ConstantId Constant,
  functions :: FiniteMap FunctionId Function
}
emptyProgram = Program emptyFM emptyFM emptyFM
```

Programm 7.5: Maschinenprogramm

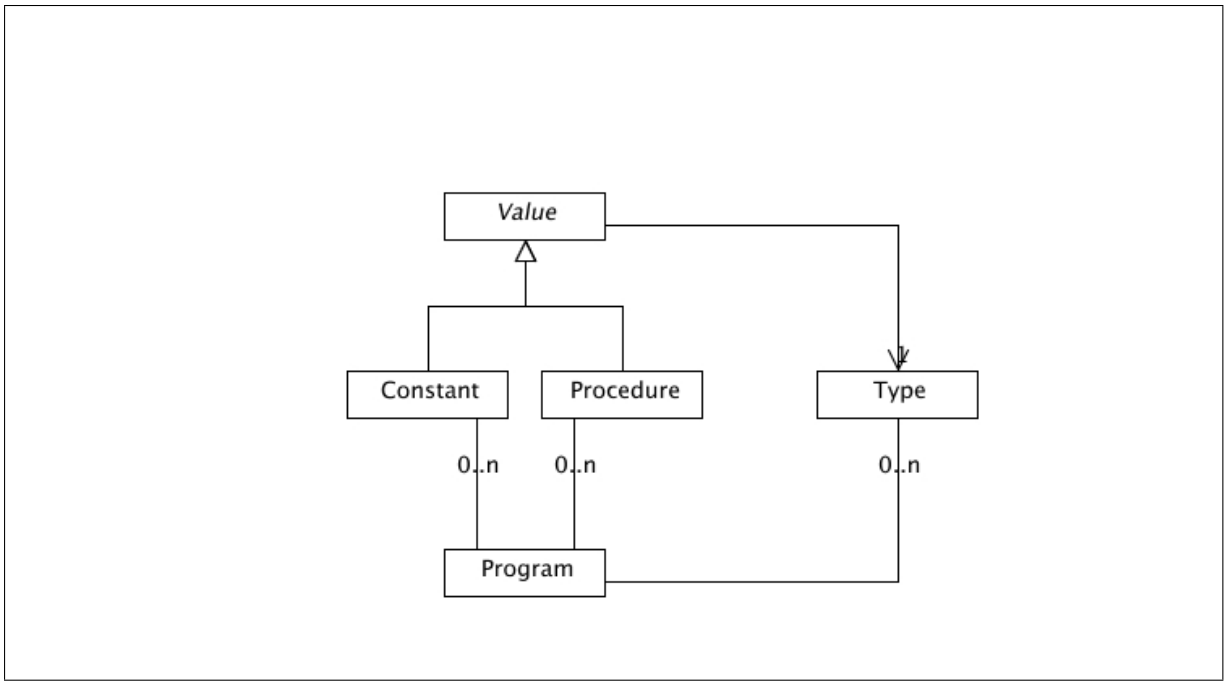


Abbildung 7.1: Programmmodell

7.2.2 Vergleich

Die größten Gemeinsamkeiten hat die hier vorgestellte Maschine mit objektorientierten Maschinen wie der Java VM[32] oder der .NET-Maschine[9].

1. Programmcode besteht aus einer Sammlung von in Prozeduren organisierten Anweisungsblöcken.

2. Die Grundoperationen zur Datenerzeugung sind die Allokation von Zellen und die Zuweisung von Variablen.

3. Lokaler Datenfluß zwischen Operationen erfolgt über einen kleinen Bereich des Stack, den *Operandenstack*.

Als wesentliche Unterschiede sind zu nennen:

1. Anstelle der typisch objektorientierten Techniken der zusätzlichen dynamischen Typisierung und des dynamischen Kontrollflußes durch virtuelle Methoden stehen Prozeduren höherer Ordnung.

2. Der intraprozedurale Kontrollfluß ist auf ein Minimum beschränkt und insbesondere zyklenfrei. Das zentrale Kontrollkonstrukt ist daher die Rekursion. Damit entfallen auch explizite Sprünge und somit die Numerierung der Anweisungen, das wesentliche Hindernis für die Programmierbarkeit von Maschinencode durch Menschen. Die Kontrollflußanalyse ist trivial.

3. Zur Unterstützung des *top-down* Vorgehens ist die Lebensdauer einer Zelle klar in Erzeugung, Initialisierung und Verwendung getrennt.

4. Anstelle spezifischer Zugriffsoperationen für verschiedene Speicherbereiche steht ein allgemeines Referenzkonzept.

7.2.2.1 Warum Operandenstack?

Bei der Reduktion einer komplexen Sprache auf ein operationales Mininum, das den Titel *abstrakte* Maschine tatsächlich verdient, bleibt als ein wesentliches Problem die Beschreibung von Datenfluß. Hier sind im wesentlichen zwei klassische Ansätze zu nennen:

1. Die *Registermaschine* verfügt über eine (im virtuellen Fall oft unbeschränkte) Anzahl von direkt adressierbaren Speicherzellen, die Register. Jede Operation greift für Ein- und Ausgaben entweder auf bestimmte Register zu oder ist mit Registeradressen statisch parametrisierbar. Datenfluß von einer Operation zur anderen findet explizit statt, indem ein Ausgaberegister der ersten zum Eingaberegister der nächsten gemacht wird.[1]

2. Die *Stackmaschine* verfügt über einen (im virtuellen Fall unbeschränkten) Stack-Speicher. Jede Operation konsumiert ihre Eingaben in festgelegter Reihenfolge vom oberen Ende des Stacks, und legt ihre Ausgaben in fester Reihenfolge auf dem Stack ab. Datenfluß findet implizit durch den Stackzustand statt.

Der bekannteste Vorteil des Stack-Ansatzes ist, daß er in natürlicher Weise der Auswertung von *Ausdrücken* entspricht. Die anonymen Zwischenwerte, welche bei der Auswertung von Teilausdrücken anfallen, werden als ebenso anonyme Bindungen in Zwischenzustanden des Stacks adäquat repräsentiert, während beim Register-Ansatz für alle Werte Adressen vergeben und Konflikte vermieden werden müssen. Ein weniger beachteter Vorteil ist, daß Stackwerte explizit konsumiert und somit bei nichtlinearer Verwendung auch explizit dupliziert werden. Zahlreiche fortgeschrittene Optimierungen, insbesondere bei funktionalen Sprachen, basieren auf der Linearitätsanalyse des Datenflusses und profitieren somit von dieser Darstellung.

Der unmittelbare Nachteil des Stack-Ansatzes ist seine Unhandlichkeit bei der Beschreibung anderer Datenflußmuster als der einfachen Termreduktion. Administrative Operationen, die lediglich den Operandenstack permutieren, werden zwischen die eigentlichen Operationen eingefügt, um den gewünschten Fluß zu erreichen. Die Auswahl und Verwendung dieser sogenannten *hygienischen* Operationen ist eine nichttriviale Kunst, wie jeder Forth-Programmierer bestätigen kann, und die eigentlichen Programmfunktionen werden schnell von der Stack-Bürokratie dominiert.

Als Konsequenz aus diesem Dilemma wird für die hier entwickelte Maschine ähnlich wie für die JVM ein hybrider Ansatz gewählt. Es steht sowohl ein Operandenstack für Reduktionen zur Verfügung, als auch eine unbeschränkte Anzahl von lokalen Variablen, die im wesentlichen Registern entsprechen, außer daß ihre Lebensdauer auf die Ausführung eines Blocks, also maximal einer Prozedur, beschränkt ist. Damit soll gewährleistet werden, daß Maschinencode auch von Menschen mit vertretbarem Aufwand geschrieben werden kann, indem fallweise die jeweils adäquatere Darstellung gewählt wird.

[1] Ohne den enthaltenen Wert zwischendurch zu überschreiben; bei beschränkter Registeranzahl ein nichttriviales Problem.

7.3 Statisches Modell

7.3.1 Module

Die Eingabesprache TASM unterstützt ein rudimentäres Modulsystem.

$$\begin{array}{lcl} \mathit{unit} & \rightarrow & \underline{\texttt{module}}\ \mathit{id}\ \mathit{decl}^* \\ \mathit{decl} & \rightarrow & \underline{\texttt{import}}\ \mathit{id} \end{array}$$

Die Definitionen importierter Module werden dem zu parsierenden Modul hinzugefügt, um zu einem vollständigen Programmmodell zu gelangen. Das definierende Modul kann bei einer Referenz explizit durch Voranstellen des Modulnamens angegeben werden.

$$\mathit{refid} \rightarrow \mathit{id}\ \underline{.}\ \mathit{refid}$$

Namenskonflikte können und müssen auf diese Weise aufgelöst werden.

7.3.2 Typen

Das Typsystem der Maschine unterteilt sich in primitive, strukturierte und Referenztypen.

7.3.2.1 Abstrakte primitive Typen

Die üblichen Verdächtigen, insbesondere die von einer konkreten ausführenden Maschine unterstützten Zahlenbereiche, können prinzipiell als primitive Typen zur Verfügung gestellt werden. Da diese nur in konkreten Implementierungen interessant sind, für das eigentliche Berechnungsmodell aber keine Rolle spielen, sind sie hier unspezifiziert.

$$\mathit{typedecl} \rightarrow \underline{\texttt{type}}\ \mathit{defid}$$

Beispiel 7.1 (Abstrakter Typ). Ein Fließkomma-Typ der konkreten Ausführungsumgebung könnte wie folgt deklariert werden:

type *real*

□

7.3.2.2 Strukturierte Typen

Konstruktoren für Produkt-, Coprodukt- und Prozedurtypen können zu beliebigen, auch zyklischen Strukturen kombiniert werden. Es existiert keine algebraische Normalform (Coprodukt von Produkten) für freie Datentypen. Generizität wird in Form freier Typvariablen und expliziter Substitutionen unterstützt. Als Abgrenzung von den weiter unten beschriebenen Referenztypen heißen strukturierte und variable Typen auch Typen erster Ordnung.

In der Eingabesprache können Typen benannt und per Namen referenziert werden.

$$\begin{array}{lcl} \mathit{typedecl} & \rightarrow & \underline{\texttt{type}}\ \mathit{defid}\ \underline{=}\ \mathit{type} \\ \mathit{type} & \rightarrow & \mathit{type}_0 \\ \mathit{type}_0 & \rightarrow & \mathit{refid} \end{array}$$

Anders als in vielen Sprachen ist der Name nicht Bestandteil der Typdefinition. Eine Typreferenz t im Kontext eines Systems $\mathbb{T}$ von Typdeklarationen ist als Coterm $\mathbb{T} \bullet t$ zu lesen.

```
type TypeId = String
type Label  = String
type Family = FiniteMap Label
type TSubst = FiniteMap TypeId Type
data Type   = TVar TypeId | TConst TypeId
            | Product (Family Type)
            | Coproduct (Family (Maybe Type))
            | Exponent {
                domain   :: [Type],
                codomain :: [Type],
                redomain :: [Type]
              }
            | Instantiation TSubst Type
```

Programm 7.6: Heap-Typsystem der Maschine

```
data RType = ValueType Type
           | RefType RType
```

Programm 7.7: Stack-Typsystem der Maschine

7.3.2.3 Referenztypen

Strukturierte Werte werden grundsätzlich per Referenz übergeben, ohne daß sich das im Typausdruck wiederspiegelt. Daneben können in einem Programm auch Referenzen zweiter und höherer Ordnung auftreten. Höhere Referenztypen sind ein rein technisches Phänomen und kein Bestandteil der Datenmodellierungssprache, daher sind sie als Belegung von Typvariablen ausgeschlossen.

$$rtype \quad \rightarrow \quad \hat{\_}^* \ type_0$$

Da in Datenzellen auf dem Heap nur Referenzen erster Ordnung zugelassen sind, und Referenzen auf den Stack immer mindestens zweiter Ordnung sind, ist kein typkorrektes Programm möglich, das einen Verweis vom Heap auf den Stack erzeugt. Diese Eigenschaft ist notwendig für die Möglichkeit automatischer Speicherverwaltung.

7.3.2.4 Statische Typisierung

Das Werteuniversum der Maschine ist *statisch streng typisiert*. Das heißt, daß jedem statischen Wertobjekt ein Typ zugeordnet ist. Als statische Wertobjekte gelten:

1. *Konstanten.* Der Typ ist explizit dem Objekt zuzuordnen beziehungsweise bei der Deklaration anzugeben.
2. *Prozeduren.* Der Typ ist explizit dem Objekt zuzuordnen beziehungsweise bei der Deklaration anzugeben.
3. *Parameter.* Der Typ ergibt sich aus dem Typ der Prozedur.
4. *Lokale Variablen.* Der Typ ist explizit dem deklarierenden Block zuzuordnen.
5. *Operanden.* Der Typ eines Wertes auf dem Operandenstack ergibt sich aus der Operation, die diesen Wert abgelegt hat, und deren Vorzustand.

7.3.2.5 Produkte

Typen können zu endlichen Produkten mit explizit benannten Komponenten zusammengefaßt werden.

$$type_0 \quad \rightarrow \quad \underline{(} \ \left(id \ \underline{:} \ type_0 \ \#\ \underline{,}\right)^* \ \underline{)}$$

Eine Instanz dieses Produkttyps (Tupel) wird als Zelle dargestellt, die pro Komponente ein Feld vom jeweiligen Komponententyp besitzt. Der Zugriff auf die einzelnen Felder eines Tupels erfolgt über Operationen der Maschine und den Komponentennamen. Ein indirekter Zugriff über eine definierte Anordnung der Felder oder dynamische Typinformation ist nicht vorgesehen. Ein Tupelwert ist eine Referenz auf eine solche Zelle.

Für den häufigen Spezialfall des *kartesischen* Produkts mit den anonymen Komponenten $(t_0, \ldots, t_{n-1})$ bietet die Eingabesprache TASM eine Kurznotation an, bei der die Komponenten in textueller Reihenfolge an die Indizes 0 bis $(n-1)$ gebunden werden.

$$type_0 \quad \rightarrow \quad \underline{(} \ \left(type_0 \ \#\ \underline{,}\right)^* \ \underline{)}$$

Beispiel 7.2 (Produkttyp). Ein Produkttyp mit expliziten Indizes sieht wie folgt aus:

type *complex* = (re : *real*, img : *real*)

□

Das einstellige anonyme Produkt (t) wird auf seine Komponente t abgebildet. Daher können Typausdrücke auch bei Bedarf geklammert werden, ohne daß Mehrdeutigkeiten entstehen:

$$type_0 \quad \rightarrow \quad \underline{(} \ type \ \underline{)}$$

7.3.2.6 Coprodukte

Coprodukte werden aus explizit benannten Varianten zusammengesetzt. Der Argumenttyp einer Variante kann weggelassen werden. Ein Coprodukt von ausschließlich argumentlosen Varianten heißt *Aufzählungstyp*.

$$\begin{array}{lcl} type_0 & \rightarrow & \underline{\{} \ \left(variant \ \#\ \underline{,}\right)^* \ \underline{\}} \\ variant & \rightarrow & id \ \left(\underline{:} \ type_0\right)^? \end{array}$$

Beispiel 7.3 (Coprodukttypen). Ein komplexer Coprodukttyp und ein Aufzählungstyp.

type *nat* = {zero, pred : *nat*}
type *bool* = {false, true}

Man beachte, daß die syntaktisch rekursive Definition bei der Abbildung in das Objektmodell der Maschine in ein zirkuläres Objekt übersetzt wird. □

Bei jeder Instanz des Coprodukttyps (Cotupel) ist genau eine der Varianten ausgeprägt. Besitzt die Variante einen Argumenttyp, wird die Instanz als Zelle darsgestellt. Diese enthält eine primitive Markierung, die die Variante auszeichnet, sowie einen *Rumpf*, einen Wert des Argumentttyps. Ein Cotupelwert ist dann eine Referenz auf diese Zelle. Für argumentlose Varianten wird die Markierung direkt als Wert verwendet.

Der Zugriff auf Markierung und Rumpf eines Cotupels erfolgt über Operationen der Maschine und den Variantennamen. Die Abbildung von Variantennamen auf Markierungswerte ist undefiniert.

```
whnf                        :: Type → Type
whnf (Instantiation s t)    = inst s t
whnf t                      = t
inst                        :: TSubst → Type → Type
inst s t                    =
  case t of
  TVar a          → lookupWithDefaultFM s t a
  TConst c        → TConst c
  Product ts      → Product (inst' s ts)
                    where inst' = mapFM ∘ const ∘ Instantiation
  Coproduct ts    → Coproduct (inst' s ts)
                    where inst' = mapFM ∘ const ∘ fmap ∘ Instantiation
  Exponent ts us vs → Exponent (inst' s ts) (inst' s us) (inst' s vs)
                    where inst' = map ∘ Instantiation
  Instantiation s' t → inst s (inst s' t)
```

Programm 7.8: Elimination von Typinstanziierungen

7.3.2.7 Prozedurtypen

Prozedurtypen spezifizieren die Schnittstelle einer Prozedur. Ein Prozedurtyp besteht aus einer endlichen Liste von Heap-Typen, von denen jeder einen Prozedurparameter spezifiziert. Die Liste ist in drei aufeinanderfolgende Bereiche aufgeteilt, die unterschiedlich auf Stack-Typen abgebildet werden:

Eingabeparameter Diese werden beim Aufruf der Prozedur mit Werten belegt und haben daher stets die spezifizierten Typen erster Ordnung.

Ausgabeparameter Diese werden beim Aufruf der Prozedur mit Referenzen auf uninitialisierte Wertvariable belegt und haben daher stets Referenztypen zweiter Ordnung.

Differenzparameter Diese werden beim Aufruf der Prozedur mit Referenzen auf uninitialisierte Referenzvariable belegt und haben daher stets Referenztypen dritter Ordnung.

Prozedurtypen dürfen wegen der Überladung des Kommas innerhalb von TASM-Typausdrücken nur geklammert vorkommen:

$$\begin{array}{lcl} \mathit{type} & \rightarrow & \mathit{types}\ \underline{\texttt{->}}\ \mathit{types}\ (\underline{\texttt{<-}}\ \mathit{types})^? \\ \mathit{types} & \rightarrow & (\mathit{type}_0\ \#\ \underline{\texttt{,}})^* \end{array}$$

Ein Prozedurwert ist eine Referenz auf eine Zelle, die eine definierte Prozedur und eine partielle Belegung von deren Eingabeparametern kapselt (*Closure*).

7.3.2.8 Typinstanziierung

Neben den freien Typkonstruktoren für Produkte, Coprodukte, und Prozedurtypen existiert noch die nicht-freie Typinstanziierung, die sich aus einem Rumpf und einer Belegung von Typvariablen zusammensetzt. An einigen Stellen dieser Spezifikation wird jedoch explizit ein freier Typkonstruktor als Kopf eines Typausdrucks erwartet. In der Sprache der Termreduktion entspricht dies einer schwachen Kopf-Normalform (WHNF). Um diese zu erhalten, müssen gegebenenfalls Instanziierungen aufgelöst werden, indem die belegten Typvariablen im Rumpf substituiert werden.

Programm 7.8 zeigt die Implementierung in Haskell. Da diese Sprache eine *lazy*-Semantik besitzt, wäre es auch möglich, anstelle des Konstruktors *Instantiation* stets nur die Funktion *inst* zu verwenden. Die Reduktion würde dann implizit stattfinden.

Es ist möglich, einen Typausdruck zu konstruieren, der sich nicht zu einer solchen Normalform reduzieren läßt. Ein solcher Typausdruck gilt als nicht wohlgeformt, und darf nicht in einem Programm vorkommen. Da die Sprache der Typausdrücke sehr einfach ist, ist Wohlgeformtheit entscheidbar: Ein Typausdruck ist genau dann wohlgeformt, wenn er keinen Zyklus enthält, in welchem jedes Mitglied eine Instanziierung ist, wobei nur der Rumpf, nicht aber die Belegungen von Typvariablen als mögliche Nachfolger betrachtet werden.

7.3.2.9 Polymorphie

In der Eingabesprache TASM werden freie Typvariablen in Deklarationen als formale Typparameter und Substitutionen als aktuelle Typparameter notiert:

$$\begin{array}{lcl} \mathit{defid} & \rightarrow & \mathit{id}\ \underline{(}\ \left(\mathit{id}\ \#\ \underline{,}\right)^*\ \underline{)} \\ \mathit{refid} & \rightarrow & \mathit{id}\ \underline{(}\ \left(\mathit{type}\ \#\ \underline{,}\right)^*\ \underline{)} \end{array}$$

Dabei gelten folgende Regeln:

1. Eine Deklaration heißt polymorph, wenn sie freie Typvariablen enthält.
2. Alle freien Typvariablen einer polymorphen Deklaration müssen als verschiedene formale Typparameter aufgeführt werden. Die Reihenfolge ist beliebig wählbar.
3. Bei der Verwendung eines polymorph deklarierten Gegenstands sind alle als Typparameter aufgeführten Typvariablen zu belegen.
4. Die Namen der formalen Typparameter dienen nur der Identifikation von Typvariablen innerhalb einer Deklaration. Sie haben außerhalb der Deklaration keine Bedeutung. Die Zuordnung aktueller Typparameter erfolgt über die Reihenfolge.
5. Bei der Abbildung auf das Objektmodell der Maschine wird pro Deklaration und Typparameter eine frische Typvariable generiert (α-Normalisierung).

Ziel dieser Konventionen ist es, partielle Instanziierungen und Namenskonflikte bei Substitution auszuschließen.

Beispiel 7.4 (Instanziierung polymorpher Typen). Der polymorphe Typ der homogenen α-Listen als rekursives System zweier, sich wechselseitig instanziierender Deklarationen:

$$\begin{array}{l} \textbf{type}\ \mathit{list}(\alpha) = \left\{\mathsf{nil},\ \mathsf{cons}:\ \mathit{cell}(\alpha)\right\} \\ \textbf{type}\ \mathit{cell}(\alpha) = \left(\mathsf{car}:\ \alpha,\ \mathsf{cdr}:\ \mathit{list}(\alpha)\right) \end{array}$$

□

```
type FunctionId = String
data Function   = Function {
  functionId    :: FunctionId,
  functionType  :: Type,
  principalBody :: Body,
  recycleBody   :: Body
}
type Body       = Maybe Block
```

Programm 7.9: Prozeduren

7.3.3 Prozeduren

Die Anweisungen eines Programms sind in Prozeduren organisiert. Eine Prozedur verfügt über eine Schnittstelle, die über den Typ der Prozedur festgelegt ist, sowie über (im Normalfall) einen Block von Anweisungen als Rumpf.

$$\begin{array}{lcl} \textit{fundecl} & \rightarrow & \underline{\texttt{fun}}\ \textit{defid}\ \underline{:}\ \textit{type}\ \underline{=}\ \textit{definition} \\ \textit{definition} & \rightarrow & \textit{block} \\ \textit{funref} & \rightarrow & \textit{refid} \end{array}$$

Der Typ einer Prozedur muß (nach Auflösung von Substitutionen) ein Prozedurtyp sein. Anzahl und Typen der Prozedurparameter werden durch die Komponenten des Prozedurtyps bestimmt. Enthält der Typ noch freie Variable, ist die Prozedur *polymorph*.

Programm 7.9 zeigt die Implementierung von Prozeduren. Zur Repräsentation des Prozedurrumpfes siehe die nächsten Abschnitte.

7.3.3.1 Abstrakte Prozeduren

Abstrakte Prozeduren werden durch das Weglassen des Rumpfes definiert. Um ein ausführbares Programm zu erhalten, müssen Implementierungen für alle abstrakten Prozeduren hinzugefügt werden. Diese werden typischerweise im Laufzeitsystem der Ausführungsumgebung in einer anderen Programmiersprache realisiert.

$$\textit{fundecl} \rightarrow \underline{\texttt{fun}}\ \textit{defid}\ \underline{:}\ \textit{type}$$

7.3.3.2 Anonyme Prozeduren

Anstelle einer Referenz auf eine Prozedur kann im Code stets auch eine anonyme Prozedur stehen. Eine solche Prozedur ist nur lokal bekannt. Sie ist nicht eigenständig polymorph, sondern erbt die Belegung von Typvariablen von ihrer Umgebung.

$$\textit{funref} \rightarrow \underline{\texttt{fun}}\ \underline{:}\ \textit{type}\ \underline{=}\ \textit{definition}$$

7.3.3.3 Zyklenbehandlung

Soll eine Prozedur Zyklenbehandlung unterstützen, wird diese durch einen zweiten Rumpf definiert, der im Fall eines erkannten Zyklus anstelle des ersten ausgeführt wird. Dieser kann auch spezielle Anweisungen enthalten, die das Kopieren von Resultaten umgebender Inkarnationen leisten. Die eigentliche Zyklenerkennung erfolgt implizit beim Prozeduraufruf. Zur technischen Umsetzung siehe Abschnitt 7.4.5.10.

$$\textit{definition} \rightarrow \textit{block}\ \underline{\texttt{recursively}}\ \textit{block}$$

```
type ConstantId = String
data Constant   = Constant {
  constantId   :: ConstantId,
  constantType :: Type,
  initializer  :: Function
}
```

Programm 7.10: Konstanten

7.3.3.4 Differenzparameter

Die Differenzparameter von Prozeduren sind eine Variante einer Technik, die in der logischen Programmierung, beispielsweise in Prolog, unter dem Schlagwort *Differenzlisten* bekannt und beliebt ist. Sie gestatten es einer Prozedur, Datenstrukturen mit Löchern zu erzeugen, die vom Aufrufer gefüllt werden können.

Logische Systeme verwenden dazu logische Variable, die nachträglich per Unifikation gebunden werden. Als Konsequenz hat die Verwendung von Differenzlisten subtile Interferenzen mit Backtracking und gekapselter Suche. In einem funktionalen System dagegen bietet es sich an, die entsprechenden Speicherstellen schlichtweg uninitialisiert zu lassen, und Referenzen darauf zurückzuliefern.

Differenzparameter dienen dazu, die Trennung von Erzeugung und Initialisierung, die üblicherweise den primitiven Konstruktoranweisungen, beispielsweise der `new`-Operation der JVM, vorbehalten ist, auf die Ebene ganzer Prozeduren zu generalisieren. Das ist für das corekursive Vorgehen bei der Erzeugung von Daten absolut notwendig, aber auch für zahlreiche andere Anwendungen nützlich. Zwar handelt es sich um eine Technik, die einem C-Virtuosen geläufig sein dürfte, aber über eine formal saubere Unterstützung ist in der einschlägigen Literatur nichts zu finden.

7.3.4 Konstanten

Ein Programm kann außer Prozeduren auch Konstanten definieren, die vor der eigentlichen Programmausführung initialisiert werden. Dazu wird jede Konstante mit einer anonymen Initialisierungsprozedur versehen. Diese bekommt als einzigen Parameter die Konstante (als Ausgabe) übergeben. Zyklische Konstanten sind möglich; hierzu kann die Zyklenbehandlung der Initialisierungsprozeduren verwendet werden. Es ist ein dynamischer Fehler, wenn eine Initialisierungsprozedur nicht terminiert.

$$\begin{array}{lcl} \textit{constdecl} & \rightarrow & \underline{\texttt{const}}\ \textit{defid}\ \underline{:}\ \textit{type}\ \underline{=}\ \textit{definition} \\ \textit{constref} & \rightarrow & \textit{refid} \end{array}$$

Wird eine Konstante vom Typ t deklariert, dann ist ihre Initialisierungsprozedur vom Typ $(\rightarrow t)$.

Die Reihenfolge der Initialisierung von Konstanten ist unspezifiziert. Daher muß der Zugriff auf die Werte von Konstanten innerhalb von Initialisierungsprozeduren anders behandelt werden. Ein solcher Zugriff kann stets durch einen Aufruf der zugehörigen Initialisierungsprozedur ersetzt werden. Ob eine Implementierung der Maschine diese Ersetzung tatsächlich ausführt, oder ob als Optimierung auf bereits initialisierte Konstanten direkt zugegriffen wird, ist unspezifiziert. Als Folge ist die Häufigkeit der Ausführung von Initialisierungsprozeduren ebenfalls unspezifiziert.

7.3.4.1 Abstrakte Konstanten

Analog zu abstrakten Prozeduren können auch abstrakte Konstanten deklariert werden, die von einem Laufzeitsystem durch Implementierung der abstrakten Initialisierungsprozedur zu belegen sind.

$$\textit{constdecl} \rightarrow \underline{\texttt{const}}\ \textit{defid}\ \underline{:}\ \textit{type}$$

7.3.4.2 Anonyme Konstanten

Analog zu den abstrakten Konstanten kann anstelle einer Referenz auf eine Konstante im Code stets auch eine anonyme Konstante stehen. Es gelten die gleichen Regeln hinsichtlich Gültigkeitsbereich und Polymorphie.

constref → `const` `:` *type* `=` *definition*

```
data Progress ρ α = Halted
                  | Returned Memory
                  | Running α

instance Functor (Progress ρ) where
  fmap _ Halted       = Halted
  fmap _ (Returned x) = Returned x
  fmap f (Running x)  = Running (f x)
instance Monad (Progress ρ) where
  Halted >>= _      = Halted
  Returned x >>= _  = Returned x
  Running x >>= f   = f x
  return x          = Running x
  fail s            = Halted
type Computation = StateT Memory Progress
```

Programm 7.11: Ablaufmonade

```
halt, return :: Computation α
halt         = StateT (const Halted)
return       = StateT Returned
```

Programm 7.12: Operationen der Ablaufmonade

7.4 Dynamisches Modell

7.4.1 Abläufe

Programm 7.11 zeigt den Monadentyp, der den Fortschritt eines Maschinenablaufs darstellt. Die möglichen Fälle sind:

1. *Halted*. Die Berechnung der Maschine wurde durch das Programm selbst abgebrochen.
2. *Returned*. Die Berechnung hat einen Prozeduraufruf erfolgreich zu Ende geführt. Es liegt ein Endzustand des Speichers vor.
3. *Running*. Die Berechnung ist im Gang. Es liegt ein Zwischenergebnis vom Typ α vor.

Dabei sind die ersten zwei Abbruchfälle, nur der dritte ist die eigentliche monadische Kapselung. Der eigentliche monadische Berechnungstyp entsteht durch die Einbettung der Fortschrittsmonade in den Zustandsmonadentransformator *StateT*. Programm 7.12 zeigt die zugehörigen Operationen für die Abbruchfälle.

7.4.2 Prozeduraufruf

Das wesentliche Strukturierungsmittel für Programme ist die Prozedur. Feinere Strukturierung ist nur rudimentär in Form von Blöcken ausgeprägt. Gröbere Strukturierung, wie etwa Module, ist zur Laufzeit nicht vorhanden.

Beim Aufruf einer Prozedur sind folgende Informationen als *Frame* auf dem Stack abgelegt:

```
data Frame    = Frame {
  frameFunction :: Function,
  params        :: [Value],
  opStack       :: [Slot],
  vars          :: [Slot]
}
initFrame f xs = Frame {
  frameFunction = f,
  params        = xs,
  opStack       = [],
  vars          = []
}
```

Programm 7.13: Stack-Frames

1. die Identität der Prozedur,

2. die Belegung der Parameter.

Darüber hinaus ist der unmittelbar umgebende Frame, von dem aus die Prozedur aufgerufen wurde, zugänglich. Im transitiven Abschluss sind alle umgebenden Frames erreichbar.

Der Block, welcher den Rumpf der Prozedur definiert, wird ausgeführt. Bei einer Prozedur mit einem zweiten Rumpf zur Zyklenbehandlung wird die Auswahl zwischen beiden in Abhängigkeit vom Erfolg einer Zyklenerkennung getroffen.

Während der Ausführung der Prozedur gehören auch folgende Informationen zu einem Frame:

1. die Belegung lokaler Variablen,

2. ein Mini-Stack von Operanden.

Programm 7.13 zeigt den Aufbau eines Frames. Zu beachten ist, daß auf Parameter stets per Wert zugegriffen wird, auf Variablen dagegen per Referenz. Operanden sind zwar ebenfalls referenzierbar, der Zugriff erfolgt aber fast immer per Wert. Siehe dazu Abschnitt 7.4.6.

7.4.3 Referenzen

Es ist ein dynamischer Kontextfehler, wenn die Lebensdauer einer Referenz die Lebensdauer der referenzierten Speicherstelle überschreitet.

7.4.4 Operandenstack

Die Operationen der Maschine tauschen Daten über den Operandenstack aus. Wir schreiben:

1. Eine Operation *konsumiert* einen Wert, indem sie ihn vom Operandenstack entfernt.

2. Eine Operation *produziert* einen Wert, indem sie ihn auf den Operandenstack legt.

Der Operandenstack ist beim Beginn der Prozedurausführung leer. Es ist ein statischer Kontextfehler, wenn für eine Operation nicht genügend Operanden auf dem Stack vorhanden sind (Unterlauf).

7.4.5 Operationen

7.4.5.1 Blöcke

Ein Block ist eine komplexe Anweisung, die eine Anzahl von lokalen Variablen und weiteren Anweisungen kapselt.

Lokale Variable werden für die Ausführung eines Blocks auf dem Stack angelegt. Dazu sind Anzahl und Typen anzugeben.

$$\begin{array}{lcl} \mathit{block} & \rightarrow & \underline{\texttt{[}}\ \mathit{vars}^?\ \mathit{stmt}^*\ \underline{\texttt{]}} \\ \mathit{vars} & \rightarrow & \underline{\texttt{vars}}\ \left(\mathit{rtype}\ \#\ \underline{\texttt{,}}\right)^* \end{array}$$

Die Lebensdauer der lokalen Variablen umfaßt exakt die Ausführung des Blocks. Zu Beginn der Ausführung sind die lokalen Variablen uninitialisiert. Lokale Variablen werden fortlaufend numeriert, wobei an die Numerierung lebender Variablen der umgebenden Blöcke lückenlos angeknüpft wird.

Die Ausführung des Blocks besteht in der sequentiellen Ausführung der enthaltenen Anweisungen. Blöcke können beliebig geschachtelt werden.

$$\mathit{stmt} \rightarrow \mathit{block} \mid \mathit{opn}$$

Es ist ein statischer Kontextfehler, wenn ein Block nicht die und nur die Operanden konsumiert, die er selbst produziert hat. Damit ist gewährleistet, daß

1. das Einschieben eines Blocks beliebigen Inhalts den Operandenstack nicht verändert,
2. die Bedeutung eines Blocks nicht vom Vorzustand des Operandenstacks abhängt.

7.4.5.2 Manipulation von Operanden

$$\mathit{opn} \rightarrow \underline{\texttt{pop}} \mid \underline{\texttt{dup}} \mid \underline{\texttt{swap}}$$

Zur Manipulation des Operandenstacks stellt die Maschine nur wenige einfache Operationen zur Verfügung. Ein komfortabler oder gar vollständiger Mechanismus, wie ihn etwa die Sprache Forth[13] anbietet, ist nicht vorgesehen. Für komplexeren Datenfluß können lokale Variable als Pseudoregister verwendet werden. Folgende Operationen manipulieren (nur) den Operandenstack:

`pop` — konsumiert das oberste Element des Operandenstacks.

`dup` — dupliziert das oberste Element des Operandenstacks.

`swap` — vertauscht die obersten beiden Elemente des Operandenstacks.

7.4.5.3 Zugriff auf den Stack

$$\mathit{opn} \rightarrow \underline{\texttt{param}}\ \mathit{nat} \mid \underline{\texttt{var}}\ \mathit{nat} \mid \underline{\texttt{insitu}}\ \mathit{rtype}^?$$

`param` i — produziert die Belegung des i-ten Parameters auf den Operandenstack. Es ist ein statischer Kontextfehler, wenn die Prozedur keinen i-ten Parameter besitzt.

`var` i — produziert eine Referenz auf die i-te lokale Variable. Es ist ein statischer Kontextfehler, wenn der Block oder ein umgebender Block keine i-te Variable besitzt.

`insitu` t — produziert eine Referenz, die auf ihren eigenen Speicherplatz verweist. Diese kann als Referenz auf eine uninitialisierte Variable des Typs t dienen. Solche Referenzen sind nützlich im Zusammenhang mit dem geschachtelten Aufruf von Prozeduren. Aufgrund der strengen Datenflußregeln läßt sich jede solche Referenz eindeutig einem konsumierenden Prozeduraufruf zuordnen. Daher ist der Typ t auch inferierbar und kann weggelassen werden.

Programm 7.17 zeigt die Zugriffsoperationen. Das Typargument von `insitu` ist für die Dynamik des Stacks irrelevant. Es dient nur der Möglichkeit der statischen Typüberprüfung.

```
data Block     = Block [RType] [Statement]
data Statement = SBlock Block
               | Pop | Dup | Swap
               | Return | Halt
               | Param Int | Var Int
               | Insitu (Maybe RType)
               | Get | Set
               | Tuple Type
               | Cotuple Label Type
               | Field Label
               | Body Label
               | Yield Constant
               | Call Function
               | Loop | Dito
               | Curry Function
               | Apply Int
               | Eval
               | Switch (FiniteMap Label Block)
```

Programm 7.14: Blöcke und Anweisungen

```
execStatement             :: Statement → Computation ()
execStatement (SBlock b) = evalBlock b
execBlock                 :: Block → Computation ()
execBlock (Block vs ss)   = do
                              let n = length vs
                              liftStateStack (replicateM_ n addVar)
                              mapM_ execStatement ss
                              liftStateStack (replicateM_ n removeVar)
```

Programm 7.15: Blockausführung

```
execStatement Pop  = do
                       consume
                       return ()
execStatement Dup  = do
                       x ← consume
                       produce x
                       produce x
execStatement Swap = do
                       x ← consume
                       y ← consume
                       produce x
                       produce y
```

Programm 7.16: Manipulation von Operanden

```
execStatement (Param i) = liftStateStack (param i) >>= produce
execStatement (Var i)   = liftStateStack (var i) >>= produce
execStatement (Insitu t) = liftStateStack insitu



param, var :: Int → State Stack Value
param i    = liftStateFrame (gets ((!! i) ∘ params))
var i      = liftStateFrame (gets (StackRef ∘ (!! i) ∘ vars))
insitu     :: State Stack ()
insitu     = do
               sl ← freshSlot
               liftStateFrame (produceSlot sl)
               setSlot sl (StackRef sl)
```

Programm 7.17: Zugriff auf den Stack

```
execStatement (Field l) = do
                            Plain c ← consume
                            UTuple m ← liftStateHeap (getCell c)
                            produce (HeapRef (lookupFM' m l))
execStatement (Body l) = do
                            Plain c ← consume
                            UCotuple l' b ← liftStateHeap (getCell c)
                            case b of Just b' | l == l' → produce (HeapRef b')
```

Programm 7.18: Zugriff auf den Heap

7.4.5.4 Zugriff auf den Heap

$$opn \quad \rightarrow \quad \underline{\texttt{field}}\ id \mid \underline{\texttt{body}}\ id$$

`field` x — konsumiert einen Tupelwert und produziert eine Referenz auf das Feld namens x des Tupels. Es ist ein statischer Typfehler, wenn der konsumierte Wert kein Tupelwert ist oder der Typ des Tupels keine Komponente namens x besitzt.

`body` x — konsumiert eine Referenz auf ein Cotupel und produziert eine Referenz auf den Rumpf des Cotupels. Es ist ein statischer Typfehler, wenn der konsumierte Wert kein Cotupelwert ist, oder der Typ des Cotupels keine oder nur eine rumpflose Variante namens x besitzt. Es ist ein dynamischer Typfehler, wenn das Cotupel nicht die Markierung der Variante x trägt.

7.4.5.5 Konstruktion

$$\begin{array}{lcl} opn & \rightarrow & \underline{\texttt{yield}}\ constref \\ opn & \rightarrow & \underline{\texttt{tuple}}\ type_0 \mid \underline{\texttt{cotuple}}\ id\ \underline{:}\ type_0 \\ opn & \rightarrow & \underline{\texttt{curry}}\ funref \mid \underline{\texttt{apply}}\ nat \end{array}$$

Für die Erzeugung neuer Werte und gegebenenfalls die Allokation zugehöriger Zellen auf dem Heap stellt die Maschine folgende Operationen zur Verfügung:

`yield` c — produziert den Wert der definierten Konstanten c.

```
execStatement (Yield c)      = gets getConst >>= produce
  where
    getConst mem = lookupFM' (memConstants mem) (constantId c)
execStatement (Tuple t)      = alloc (UTuple ts) >>= produce
  where
    Product ts = whnf t
execStatement (Cotuple l t) = alloc (UCotuple l b) >>= produce
  where
    Coproduct ts = whnf t
    b            = (lookupFM' ts l)
alloc s                      = do
                                 (c, _) <- liftStateHeap (newCell s)
                                 return (Plain c)
execStatement (Curry f)      = do
                                 c <- liftStateHeap (consCell (UClosure f []))
                                 produce (Plain c)
execStatement (Apply n)      = do
                                 Plain c <- consume
                                 UClosure f xFields <- liftStateHeap (getCell c)
                                 xCells <- liftStateHeap (mapM getField xFields)
                                 yValues <- replicateM n consume
                                 let yCells = fmap fromPlain yValues
                                 let result = UClosure f (xCells ++ yCells)
                                 c' <- liftStateHeap (consCell result)
                                 produce (Plain c')
```

Programm 7.19: Konstruktion von Daten

`tuple` t — produziert eine Referenz auf ein neues Tupel vom Typ t. Die Felder dieses Tupels sind uninitialisiert. Es ist ein statischer Typfehler, wenn t kein Produkttyp ist.

`cotuple` l**:** t — produziert entweder eine primitive Markierung, falls l eine rumpflose Variante von t ist, oder andernfalls eine Referenz auf ein neues Cotupel. Die Markierung des Cotuples ist initialisiert, der Rumpf nicht. Es ist ein statischer Typfehler, wenn t kein Coprodukttyp ist oder keine Variante namens l besitzt.

`curry` p — produziert eine Closure, welche die definierte Prozedur p bezeichnet und keine belegten Eingabeparameter besitzt.

`apply` k — konsumiert eine Closure und k Parameterwerte, und produziert eine Closure, welche die partielle Belegung der nächsten k Eingabeparameter mit den Parameterwerten beschreibt. Der Typ der produzierten Closure ergibt sich aus dem Typ der konsumierten Closure durch Streichen der belegten Parameter. Es ist ein statischer Typfehler, wenn die Closure keine ist oder die Typen der Parameterwerte nicht mit den Parametern übereinstimmen. Es ist ein statischer Kontextfehler, wenn die Prozedur keine weiteren k Eingabeparameter besitzt.

Programm 7.19 zeigt die Konstruktionsoperationen. Für Tupel und Cotupel wird die Zellenstruktur aus dem Typausdruck hergeleitet.

7.4.5.6 Dereferenzierung

$$opn \quad \rightarrow \quad \underline{\texttt{get}} \mid \underline{\texttt{set}}$$

```
execStatement Get = consume >>= getValue >>= produce
execStatement Set = do
                      v <- consume
                      r <- consume
                      setValue r v
```

Programm 7.20: Dereferenzierung

```
getValue (StackRef r)          = liftStateStack (getSlot r)
getValue (HeapRef r)           = liftStateHeap (fmap Plain (getField r))
setValue (StackRef r) v        = liftStateStack (setSlot r v)
setValue (HeapRef r) (Plain c) = liftStateHeap (setField r c)
```

Programm 7.21: Implementierte Dereferenzierung

`get` — konsumiert eine Referenz und produziert den Inhalt der referenzierten Speicherstelle. Es ist ein statischer Typfehler, wenn die Referenz nicht höherer Ordnung ist. Es ist ein dynamischer Kontextfehler, wenn die referenzierte Speicherstelle uninitialisiert ist.

`set` — konsumiert einen Wert und eine Referenz und weist der referenzierten Speicherstelle den Wert zu. Es ist ein statischer Typfehler, wenn der Typ der Referenz und des Wertes nicht kompatibel sind. Es ist ein dynamischer Kontextfehler, wenn die referenzierte Speicherstelle initialisiert ist.

Programm 7.20 zeigt die Modellierung der Dereferenzierungsoperationen, Programm 7.21 zeigt die Implementierung.

7.4.5.7 Globaler Kontrollfluß

$$opn \rightarrow \underline{\texttt{halt}}$$

`halt` — beendet die Ausführung des Programms. Folgende Anweisungen in dieser oder umgebenden Prozedurinkarnationen werden nicht ausgeführt.

7.4.5.8 Intraprozeduraler Kontrollfluß

Getreu dem funktionalen Paradigma steht hier nur ein einziges Fallunterscheidungskonstrukt zur Verfügung, das dem universellen Morphismus des Coprodukts nachempfunden ist. Alle anderen Arten von Sprüngen müssen durch Prozeduraufruf und gegebenenfalls Rekursion ausgedrückt werden.

$$\begin{aligned} opn &\rightarrow \underline{\texttt{switch}}\ case^* \\ case &\rightarrow (id\ \#\ \underline{,})^+\ \underline{:}\ block \\ case &\rightarrow \underline{\texttt{default}}\ \underline{:}\ block \end{aligned}$$

```
execStatement Halt = halt
```

Programm 7.22: Globaler Kontrollfluß

```
execStatement (Switch cases) = do
                                 Plain c ← consume
                                 UCotuple l b ← liftStateHeap (getCell c)
                                 unpack b
                                 execBlock (lookupFM′ cases l)
  where
    unpack Nothing = return ()
    unpack (Just b′) = do
                         c′ ← liftStateHeap (getField b′)
                         produce (Plain c′)
```

Programm 7.23: Verzweigung

`switch` — konsumiert einen Cotupelwert und verzweigt zu einem Block mit passender Markierung, oder andernfalls zu einem `default`-Block. Produziert vor der Verzweigung den Rumpf des Cotupels, falls vorhanden. Es ist ein statischer Typfehler, wenn der konsumierte Wert nicht von einem Coprodukttyp ist. Es ist ein dynamischer Kontextfehler, wenn kein passend markierter Block vorhanden ist. Es ist ein statischer Kontextfehler, wenn eine Blockmarkierung mehrfach auftritt, oder im Typ des konsumierten Wertes nicht vorkommt. Da der Typ des konsumierten Wertes statisch feststeht, kann der `default`-Fall nach statischer Typanalyse eliminiert und in die fehlenden Fälle überführt werden.

7.4.5.9 Interprozeduraler Kontrollfluß

$$opn \quad \rightarrow \quad \underline{\texttt{return}} \mid \underline{\texttt{eval}} \mid \underline{\texttt{call}}\ funref \mid \underline{\texttt{loop}}$$

`return` — beendet die Ausführung der Prozedur. Folgende Anweisungen in diesem oder umgebenden Blöcken werden nicht ausgeführt.

`eval` — konsumiert einen Prozedurwert und eine typabhängige Anzahl von Ausgabe- und Differenzparameterwerten. Ruft die Prozedur mit den gegebenen Parametern auf. Es ist ein statischer Typfehler, wenn die Prozedur noch Eingabeparameter erwartet, oder wenn die Typen der Parameter und der Werte nicht übereinstimmen. Es ist ein statischer Kontextfehler, wenn nicht genügend Operanden für alle Ausgabe- und Differenzparameter der Prozedur vorhanden sind.

`call` p — konsumiert eine vom Typ der definierten Prozedur p abhängige Anzahl von Eingabe-, Ausgabe- und Differenzparametern. Ruft die Prozedur mit den Parametern auf. Es ist ein statischer Typfehler, wenn die Typen der Parameter und der Werte nicht übereinstimmen. Es ist ein statischer Kontextfehler, wenn nicht genügend Operanden für alle Parameter der Prozedur vorhanden sind. Nach Abarbeitung der aufgerufenen Prozedur ist mit der Fortseztung des aktuellen Ablaufs (nach der `call`-Anweisung) fortzufahren.

`loop` — ein Spezialfall von `call`, wobei aufrufende und aufgerufene Prozedur identisch sind, gefolgt von `return`.

Programm 7.24 zeigt die Modellierung. Für die Funktionsweise der beiden Hilfsroutinen *consumeArgs* und *consumeOutArgs* sei auf Abschnitt 7.4.6 verwiesen.

7.4.5.10 Zyklenerkennung

Programm 7.25 zeigt den Aufruf einer Prozedur. Die Inkarnation wird als Frame aufgebaut. Anschließend wird abhängig von der Existenz des zweiten Prozedurrumpfs die Zyklenerkennung ausgeführt und in den

```
execStatement Return  = return
execStatement (Call f) = consumeArgs (whnf (functionType f)) >>= callFunction f
execStatement Eval    = do
                          Plain c <- consume
                          UClosure f xCells <- liftStateHeap (getCellFields c)
                          let xValues = map Plain xCells
                          (yValues, n) <- consumeOutArgs (whnf (functionType f))
                          callFunction f (xValues ++ yValues, n)
execStatement Loop    = do
                          f <- liftStateStack currentFunction
                          execStatement (Call f) execStatement Return
```

Programm 7.24: Interprozeduraler Kontrollfluß

```
execFunction          :: Function -> [Value] -> Int -> Computation ()
execFunction f xs n   = do
                          liftStateStack (enter f xs)
                          body <- chooseBody (principalBody f) (recycleBody f)
                          transferControl (execBlock body)
                          liftStateStack (leave (take n (reverse xs)))
  where
    chooseBody (Just b) Nothing  = return b
    chooseBody (Just b) (Just b') = let cyc = liftStateStack cycleCheck
                                    in fmap (maybe b (const b')) cyc
callFunction f (xs, k) = execFunction f xs k
```

Programm 7.25: Prozeduraufruf in der Maschine

```
cycleCheck        :: State Stack (Maybe Frame)
cycleCheck        = do
                      fr_0 : frs ← gets stackFrames
                      return (find (equivEQ cycleCheckEquiv fr_0) frs)
cycleCheckEquiv :: Equiv Frame
cycleCheckEquiv = Equiv (≃)
  where
    fr ≃ fr' = let
                 f    = frameFunction fr
                 f'   = frameFunction fr'
                 n    = length (inParams f)
                 ins  = take n (params fr)
                 ins' = take n (params fr')
               in
               f == f' && ins == ins'
```

Programm 7.26: Zyklenerkennung in der Maschine

passenden Rumpf verzweigt. Abschließend wird der Frame abgebaut und etwaige insitu-Parameter für den Aufrufer zurückgelassen. Programm 7.26 zeigt die Implementierung der Zyklenerkennung, in welcher der Stack in absteigender Reihenfolge durchsucht wird, und für jeden Frame Identität der aufgerufenen Prozedur und Belegung der Eingabeparameter mit den aktuellen verglichen werden.

Dabei steht *cycleCheckEquiv* für die exakte Zyklenerkennung. Für Quasizyklen ist eine gröbere Äquivalenzrelation auf Daten, nämlich die generische Bisimilarität auf Speicherzuständen, zu verwenden.

7.4.5.11 Corekursion

$$opn \quad \rightarrow \quad \underline{\texttt{dito}}$$

Die Behandlung von Zyklen durch das Kopieren von Resultaten, wie in Abschnitt 4.3 beschrieben, wird durch eine einzige Operation realisiert. Voraussetzung ist die erfolgreiche Zyklenerkennung, wie im vorigen Abschnitt modelliert, also daß ein Frame mit passender Prozedur-Identität und Eingabeparametern gefunden wurde.

`dito` — Kopiert die Inhalte der von Ausgabeparametern referenzierten Speicherstellen vom gefundenen Frame zum aktuellen Frame. Es gelten die gleichen Fehlerbedingungen wie für eine entsprechende Folge von `set`-Operationen. Es ist ein statischer Kontextfehler, wenn die Operation im nichtzyklischen Rumpf einer Prozedur vorkommt, oder die Prozedur Differenzparameter besitzt.

Eine vergleichbare Operation ist in der einschlägigen Literatur nicht zu finden.

Programm 7.27 zeigt die Implementierung der `dito`-Operation. Anders als in Kapitel 4 können hier mehrere Ausgaben simultan produziert werden.

Beispiel 7.5 (Zyklenbehandlung). Folgende Code-Muster entsprechen den Zyklenbehandlungstechniken der Kapitel 4 und 5:

1. Corekursives Berechnen einer Funktion:

```
recursively
[
        dito
]
```

```
execStatement Dito = liftStateStack dito
dito               :: Computation ()
dito               = do
                       inner : _ ← liftStateStack (gets stackFrames)
                       Just outer ← liftStateStack cycleCheck
                       let f          = frameFunction inner
                           n          = length (inParams f)
                           0          = length (diffParams f)
                           out        = drop n ∘ params
                           copy from to = getValue from ≫= setValue to
                       zipWithM_ copy (out outer) (out inner)
```

Programm 7.27: Zyklenbehandlung in der Maschine

Die Theorie für die Klasse der primitiv rekursiven Funktionen ist in Kapitel 4 ausführlich beschrieben. Für ein darüber hinausgehendes, ebenfalls effektives Beispiel siehe Abschnitt B.3.5.

2. Rational corekursives Entscheiden eines Prädikats, kleinster Fixpunkt:

```
recursively
[
        param   n
        cotuple false: bool
        set
]
```

3. Rational corekursives Entscheiden eines Prädikats, größter Fixpunkt:

```
recursively
[
        param   n
        cotuple true: bool
        set
]
```

Andere Fälle sind denkbar, ihre Semantik muß aber individuell untersucht werden. □

7.4.6 Aufrufkonventionen

Die Konvention, Prozedurausgaben durch Ausgabeparameter anstelle von Rückgabewerten zu realisieren, hat weitreichende positive Auswirkungen:

1. Es können mehrere Ausgaben simultan produziert werden, ohne künstlich in ein Tupel eingepackt werden zu müssen.

2. Durch doppelte Anwendung der Codierung per Referenz ergeben sich natürlicherweise die Differenzparameter.

3. Die Delegation von Berechnungen in Endposition (*tail*-Aufruf), das wichtigste funktionale Ausdruckmittel für Sprünge und Schleifen, läßt sich elegant formulieren.

4. Zyklenbehandlung ist durch einfaches Kopieren von Belegungen von Ausgabeparametern möglich, wie in Algorithmus Ana_3 ausgenutzt.

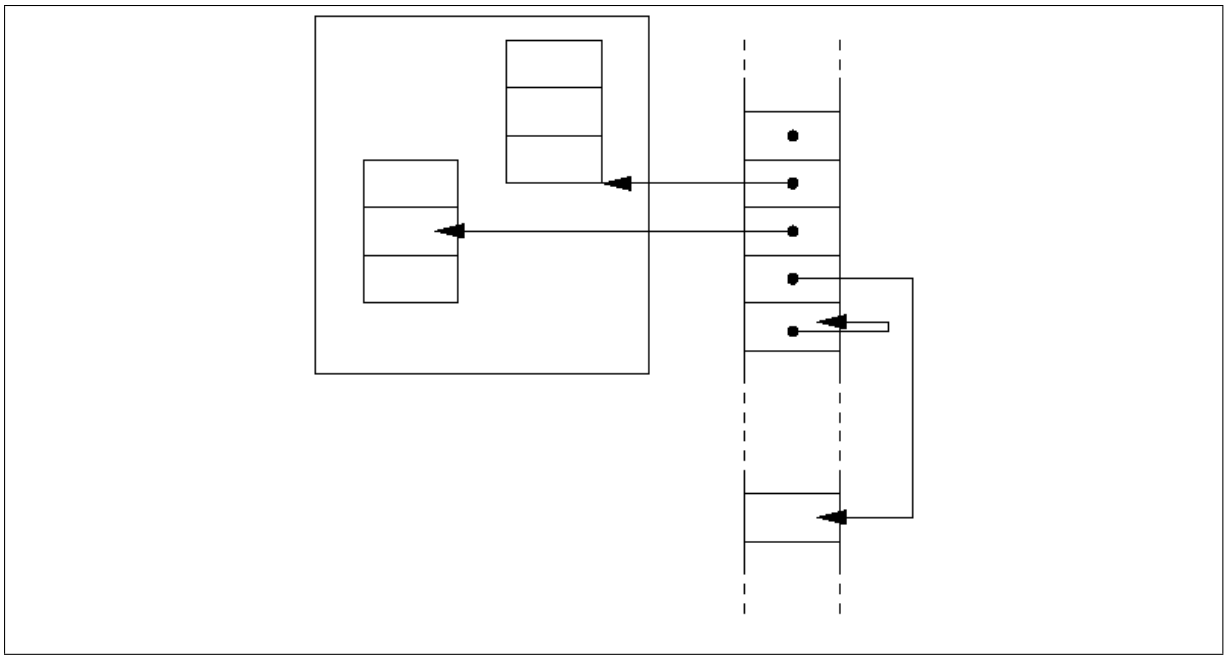


Abbildung 7.2: Aufrufkonventionen

Als gewichtiger Nachteil ist dem entgegenzuhalten, daß sich die Komposition von Prozeduraufrufen nicht ohne Bereitstellung eines Zwischenspeichers ausdrücken läßt. Hier gewinnt eindeutig die gegenteilige Konvention, bei der die Ausgaben eines Aufrufs die Eingaben auf dem Stack ersetzen.

Mit einem kleinen technischen Trick ist es möglich, die zweite Konvention in der ersten zu simulieren, um so die Vorteile beider Ansätze zu vereinen:

1. Auf der Seite des Aufrufers werden mittels der `insitu`-Operation Selbstreferenzen auf den Operandenstack gelegt.

2. Auf der Seite des Aufgerufenen können diese als Ausgabe- beziehungsweise Differenzparameter behandelt werden. Die konventionelle Zugriffsweise auf diese Parameter, Dereferenzieren gefolgt von Schreiben, führt dazu, daß die Selbstreferenz auf dem Stack durch ein Resultat ersetzt wird. Danach ist keine Dereferenzierung mehr möglich. Es ist also vorauszusetzen, daß eine Prozedur ihre Resultate nur genau einmal schreibt.

3. Zurück auf der Seite des Aufgerufenen werden die betreffenden Resultate einfach auf dem Stack gelassen, anstatt durch den Aufruf konsumiert zu werden.

Als solche insitu-Parameter kommen nur Referenzen höherer Ordnung, also Ausgabe- und Differenzparameter in Frage. Außerdem müssen sie einen zusammenhängenden Bereich auf dem Operandenstack unterhalb aller nicht-insitu-Parameter bilden. Es gilt also: wenn bei einem Prozeduraufruf $f(x_1, \ldots, x_n)$ der Parameter x_i insitu übergeben wird, dann müssen auch alle Parameter x_j mit $j > i$ insitu übergeben werden.

Abbildung 7.2 zeigt einen Ausschnitt von Heap und nach oben wachsendem Stack. Hervorgehoben ist die Inkarnation einer Prozedur mit fünf Parametern. Dabei handelt es sich in aufsteigender Reihenfolge, also von oben nach unten auf dem Stack, um zwei Eingaben, nämlich einen primitiven Wert und eine Zellenreferenz, eine Ausgabe, deren Wert auf dem Heap abgelegt werden soll, eine Ausgabe oder Differenz, deren Wert auf dem Stack abgelegt werden soll, sowie eine insitu-Ausgabe oder -Differenz.

Es ist ein statischer Kontextfehler, wenn per `insitu` erzeugte Selbstreferenzen von einer anderen Operation als `call` oder `eval` konsumiert werden. Da der Operandenstack lokal zu jedem Block ist, kann folglich die Anzahl der insitu-Parameter bei jedem Aufruf statisch bestimmt werden.

```
consumeArgs, consumeOutArgs :: Type → Computation ([Value], Int)
consumeArgs t               = do
                                xs ← replicateM (length (domain t)) consume
                                (ys, n) ← consumeOutArgs t
                                return (xs ++ ys, n)
consumeOutArgs t            = do
                                (xs, m) ← consumeOutArgsInsitu (codomain t) 0
                                (ys, n) ← consumeOutArgsInsitu (redomain t) m
                                return (xs ++ ys, n)
consumeOutArgsInsitu        :: [Type] → Int → Computation ([Value], Int)
consumeOutArgsInsitu [] n   = return ([], n)
consumeOutArgsInsitu (t : ts) k = consume >>= check
  where
    check x = case x
              of   StackRef r → do
                                  i ← liftStateStack (Stack.isInsituRef r)
                                  if i then
                                    consumeOutArgInsitu x
                                  else
                                    consumeOutArg x
                   → consumeOutArg x
    consumeOutArg x̄ | k == 0 = do
                                  (ys, n) ← consumeOutArgsInsitu ts 0
                                  return (x : ys, n)
    consumeOutArgInsitu x = do
                              (ys, n) ← consumeOutArgsInsitu ts (k + 1)
                              return (x : ys, n + 1)
```

Programm 7.28: Aufrufkonventionen der Maschine

```
enter      :: Function → [Value] → State Stack ()
enter f xs = modify (modifyFrames (initFrame f xs :))
leave      :: [Value] → State Stack ()
leave xs   = do
               modify (modifyFrames tail)
               liftStateFrame (mapM_ produceSlot (map fromStackRef xs))
```

Programm 7.29: Parameterübergabe in der Maschine

Programm 7.28 zeigt eine etwas komplexe Modellierung, bei der insitu-Parameter dynamisch erkannt werden. Ihre Anzahl wird neben den konsumierten Parameterwerten zurückgeliefert.

Programm 7.29 zeigt die Operationen zum Auf- und Abbau von Frames beim Betreten beziehungsweise Verlassen einer Prozedur. Die *leave* übergebenen Werte sind die insitu-Parameter, die in den Operandenstack des Aufrufers übernommen werden, in umgekehrter Reihenfolge.

7.4.7 Prozeduren höherer Ordnung

Das Zusammenspiel der Operationen `curry`, `apply`, `eval` und `call` läßt sich auf folgende Eigenschaften reduzieren:

1. `apply` hat Monoid-Struktur: Die Operation `apply` 0 hat keinen Effekt. Zwei aufeinanderfolgende Operationen `apply` m und `apply` n sind äquivalent zu einer einzigen Operation `apply` $(m + n)$.

2. Die Folge „`curry` p, `apply` k, `eval`“, wobei k die Anzahl der Eingabeparameter von p ist, ist äquivalent zur Operation `call` p.

Die Aufteilung von Inkarnation und Aufruf einer Prozedur kostet zwar Effizienz, bringt aber offensichtliche Vorteile:

1. Prozeduren können als Werte übergeben werden.

2. Kontextabhängige Prozeduren können als Closures dargestellt werden (*λ-lifting*).

3. Durch die verzögerte Ausführung einer vollständig inkarnierten Prozedur läßt sich explizite *lazy evaluation* realisieren. Diese Technik ist beispielsweise in der Sprache Scheme[26] unter dem Schlagwort *promise* bekannt.

7.4.8 Meta-Attribute

Anstelle von zusätzlichen Schlüsselwörtern zur Qualifikation von Programmelementen besitzt die Sprache TASM einen generischen Mechanismus zur Meta-Attributierung. Die Meta-Attribute an einem Programmelement haben die Form polymorpher Bezeichner, denen optional ein Wert der gleichen Form zugeordnet sein kann:

$$\begin{array}{lcl} \mathit{attrs} & \rightarrow & \underline{@}\ \underline{\{}\ \left(\mathit{attr}\ \#\ \underline{,}\right)^{*}\ \underline{\}} \\ \mathit{attr} & \rightarrow & \mathit{refid}\ \left(\underline{=}\ \mathit{refid}\right)^{?} \end{array}$$

In der aktuellen Version ist weder eine Deklaration von Attributtypen noch eine Parametrisierung oder Belegung der Attribute mit strukturierten Werten unterstützt. Die Bedeutung der Attribute ist prinzipiell unspezifiziert. Die von den implementierten Werkzeugen erkannten Attribute und ihre Bedeutung werden in den folgenden Abschnitten erläutert, ebenso die Plazierungsregeln. Irrelevante Attribute werden kommentarlos ignoriert.

7.4.8.1 Meta-Attribute an Prozeduren

Prozeduren können attributiert werden.

$$\begin{array}{lcl} \mathit{fundecl} & \rightarrow & \underline{\texttt{fun}}\ \mathit{defid}\ \mathit{attrs}\ \underline{:}\ \mathit{type}\ \underline{=}\ \mathit{definition} \\ \mathit{fundecl} & \rightarrow & \underline{\texttt{fun}}\ \mathit{defid}\ \mathit{attrs}\ \underline{:}\ \mathit{type} \\ \mathit{funref} & \rightarrow & \underline{\texttt{fun}}\ \mathit{attrs}\ \underline{:}\ \mathit{type}\ \underline{=}\ \mathit{definition} \end{array}$$

Folgende Attribute sind vordefiniert:

`hom` — Aktiviert die in Abschnitt 6.1.2 beschriebene Optimierung der Zyklenerkennung in Interpreter und Compiler. Es wird nicht geprüft, ob die Prozedur tatsächlich eine homomorphe Interpretation implementiert.

`quasi` — Aktiviert die in Abschnitt 6.3 beschriebene Zyklenerkennung modulo Bisimilarität in Interpreter und Compiler und für alle Eingabeparameter der Prozedur. Falls einer der Parameter von einem Funktionstyp ist, ist der Effekt unspezifiziert.

`inline` — Empfielt die Substitution des Prozedurrumpfes an allen Aufrufstellen. Hat nur im Compiler und nur bei nichtzyklischen Prozeduren einen Effekt. Außerdem sollte die Prozedur offensichtlich klein und nicht rekursiv sein.

`deprecated` — Kennzeichnet Prozeduren, die nicht oder nicht mehr verwendet werden sollen. Hat im Compiler den Effekt, daß bei jedem Zugriff per `call`/`curry` eine Warnung ausgegeben wird.

7.4.8.2 Meta-Attribute an Konstanten

Konstanten können auf die gleiche Weise wie Prozeduren attributiert werden.

$$\begin{array}{lcl} \mathit{constdecl} & \rightarrow & \underline{\texttt{const}}\ \mathit{defid}\ \mathit{attrs}\ \underline{:}\ \mathit{type}\ \underline{=}\ \mathit{definition} \\ \mathit{constdecl} & \rightarrow & \underline{\texttt{const}}\ \mathit{defid}\ \mathit{attrs}\ \underline{:}\ \mathit{type} \\ \mathit{constref} & \rightarrow & \underline{\texttt{const}}\ \mathit{attrs}\ \underline{:}\ \mathit{type}\ \underline{=}\ \mathit{definition} \end{array}$$

Attribute werden falls nötig an die Initialisierungsprozedur vererbt. Von den oben definierten Attributen sind hier nur `hom` und `deprecated` sinnvoll.

7.4.8.3 Meta-Attribute an Parametertypen

Die einzelnen Parametertypen einer Prozedurdeklaration können attributiert werden. Attribute innerhalb gekapselter Funktionstypen haben keinen Effekt.

$$\mathit{types} \rightarrow \left(\mathit{type}_0\ \mathit{attrs}^{?}\ \#\ \underline{,}\right)^{*}$$

Folgende Attribute sind vordefiniert:

`invar` — Nimmt den Eingabeparameter von der Zyklenerkennung aus. Nützlich zur effizienten Implementierung von parametrischen Familien von Interpretationen, bei denen ein oder mehrere Parameter bei Rekursion unverändert weitergegeben werden. Gilt in Interpreter und Compiler. Es wird nicht geprüft, ob die Rekurrenzrelation dieser Bedingung wirklich genügt.

`quasi` — Aktiviert die Zyklenerkennung modulo Bisimilarität nur für diesen Eingabeparameter. Gilt in Interpreter und Compiler.

7.4.8.4 Meta-Attribute an Blöcken

Prinzipiell können Blöcke attributiert werden.

$$block \quad \rightarrow \quad \underline{[} \ attrs \ vars^{?} \ stmt^{*} \ \underline{]}$$

Es sind allerdings keine Attribute vordefiniert.

7.4.8.5 Meta-Attribute an Anweisungen

Auch einzelne Anweisungen können attributiert werden.

$$stmt \quad \rightarrow \quad opn \ attrs$$

Folgende Attribute sind vordefiniert:

`inline` — empfielt die Substitution der aufgerufenen Prozedur an dieser Stelle. Wird vom Compiler in jedem Betriebsmodus an `call`-Anweisungen unterstützt. Das Attribut kann auch an `apply`- und `eval`-Anweisungen stehen, hat dann aber nur bei bestimmten Spezialisierungen, die hauptsächlich im *just-in-time*-Modus auftreten, einen Effekt.

`tag` — Aktiviert bei einer `set`-Anweisung die Markierung der geschriebenen Referenz. Kann in Interpreter und Compiler verwendet werden, um Zyklen *ad hoc* durch wechselseitige Zuweisung zu erzeugen, ohne die Markierungsinvariante für die Optimierung der Zyklenerkennung zu verletzen.

7.4.9 Referentielle Transparenz

Der Nachweis, daß ein Programm keine Fehler enthält, die in Zusammenhang mit Referenzen stehen, kann im Allgemeinen sehr komplex werden. Eine erhebliche Vereinfachung läßt sich durch ein Protokoll erreichen, welches garantiert, daß die diesbezüglichen Eigenschaften einer Prozedur aus ihrem Typ abgeleitet werden können. In diesem Fall kann die Verifikation streng modular prozedurweise erfolgen, da die Typen aufgerufener Prozeduren an der Aufrufstelle stets bekannt sind. Ein solches Protokoll heißt *prozedurweise referentielle Transparenz*. Eine Prozedur, die das Protokoll erfüllt, heißt *referentiell transparent*.

Ein Protokoll für referentielle Transparenz hat die Gestalt eines Vertrages zwischen aufrufender und aufgerufener Prozedur, also einer Zusicherung, daß die aufgerufene Prozedur gewisse Eigenschaften ihrer Ausgabe- und Differenzparameter garantiert, sofern die aufrufende Prozedur gewisse Eigenschaften der Eingabeparameter sicherstellt.

Ob ein Programm tatsächlich prozedurweise verifiziert werden kann, hängt von der Stärke des zugrundegelegten Protokolls ab. Wie bei allen statischen Einschränkungen gilt: Je strenger die Regeln, desto größer die Anzahl der nachweisbaren Eigenschaften, und desto kleiner die Menge der konformen Programme. Das im folgenden vorgestelle Protokoll kann daher keinen normativen Charakter haben, sondern lediglich die zu beachtenden Punkte in übersichtlicher Form zusammenstellen. Dabei sind zwei Prinzipien einzuhalten:

1. Einmal erreichbar gemachte, oder auch einmal initialisierte Speicherstellen ändern ihren Wert nicht mehr (*single assignment*).

2. Nur initialisierte Speicherstellen sind erreichbar. Das bedeutet, daß temporäre Referenzen, die zur Initialisierung der Zelle dienen, nicht zum Lesen des Zelleninhalts verwendet werden dürfen, und in diesem Sinne nicht zur Ereichbarkeit beitragen.

Definitionen

1. Ein Wert heißt *vollständig*, wenn alle von ihm aus erreichbaren Speicherstellen initialisiert sind. Dies entspricht dem Begriff der *Lebensfähigkeit* aus Definition 3.5.

2. Der Wert der Speicherstelle, auf die ein Ausgabe- oder Differenzparameter verweist, nach dem Aufruf heißt *Ergebnis* des Parameters.

3. Ein Wert heißt *publiziert*, wenn er vom Ergebnis eines Ausgabeparameters aus erreichbar ist. Die Speicherstellen in einer publizierten Zelle heißen ebenfalls publiziert.

4. Eine Speicherstelle heißt *delegiert*, wenn sie vom Ergebnis eines Differenzparameters referenziert wird.

5. Eine Zelle heißt *aktiv*, wenn sie von einer Konstanten oder vom Stack aus erreichbar ist. Die Speicherstellen des Stacks und von aktiven Zellen heißen ebenfalls aktiv.

Vorbedingungen

1. Alle Eingabeparameter sind vollständig.

2. Alle Ausgabe- und Differenzparameter referenzieren paarweise verschiedene, aktive und uninitialisierte Speicherstellen.

3. Falls der Aufruf direkt oder indirekt zyklisch ist und die Prozedur über eine Zyklenbehandlung mit der `dito`-Operation verfügt: Alle Ausgabeergebnisse des Aufrufers sind initialisiert.[2]

[2]Ein Verstoß gegen diese Regel bedeutet: die durch die Prozedur implementierte Funktion ist nicht in primitiv corekursiver Normalform. Einige der späteren Beispielapplikationen werden diese Regel verletzen. Potentielle Konsequenzen werden diskutiert.

Nachbedingungen für den regulären Rumpf

1. Alle Ergebnisse sind initialisiert.
2. Alle delegierten Speicherstellen sind aktiv und uninitialisiert.
3. Alle publizierten Werte werden vollständig sein, sobald alle delegierten Speicherstellen initialisiert sind. Anders ausgedrückt: Alle publizierten Speicherstellen sind entweder initialisiert oder delegiert.

Nachbedingungen für den zyklischen Rumpf

1. Alle Ergebnisse sind publiziert und initialisiert.

Ein Algorithmus, der die Einhaltung des Protokolls mittels HOARE-Logik induktiv über dem Aufbau eines Prozedurrumpfs nachweist, ist dank des primitiven Kontrollflusses denkbar, aber nicht Bestandteil dieser Spezifikation.

7.5 Einordnung des Typsystems

Das Typsystem der Maschine orientiert sich an den Typsystemen existierender Zwischensprachen für die Implementierung funktionaler Programmiersprachen, wie etwa der Zwischensprache Core des Haskell-Compilers ghc. Explizite Polymorphie in Form von Typparametern ist gut geeignet, um die üblichen Typsysteme funktionaler Hochsprachen abzubilden, siehe [48].

In einem Punkt geht das hier vorgestellte Typsystem allerdings über die Mächtigkeit herkömmlicher Systeme hinaus: Die β-Reduktion von Funktionsausdrücken wird in Form der `eval`-Operation explizit gemacht. Daher ist es möglich, den Typkonstruktor $(\rightarrow)$ ebenso wie Produkt und Coprodukt frei in beliebigen, auch zyklischen Typausdrücken zuzulassen. So ist die folgende Typgleichung eine legale Definition einer polymorphen Familie von zyklischen Typcotermen:

$$\textbf{type}\ \mathit{auto}(\alpha) = \mathit{auto}(\alpha) \rightarrow \alpha$$

Damit lassen sich Selbstapplikationen typisieren, und sogar ein effektiver Y-Kombinator konstruieren. Wir verwenden als Notation für die folgende Programmtransformation die Syntax der Sprache Haskell, auch wenn die einzelnen Ausdrücke dort nicht typisierbar sind.

1. Man beginnt mit der üblichen Definition des Y-Kombinators:

   $$y\ f\ =\ g\ g\ \textbf{where}\ g\ x\ =\ f\ (x\ x)$$

2. Durch den Prozess der *closure conversion* werden alle lokalen Funktionsausdrücke mit offenen Variablen eliminiert:

   $$\begin{array}{l} y\ f\ \ \ =\ (g\ f)\ (g\ f) \\ g\ f\ x = \ f\ (x\ x) \end{array}$$

3. Nun werden durch A-Normalisierung geschachtelte Terme aufgelöst:

   $$\begin{array}{l} y\ f\ \ \ =\ \textbf{let}\ h\ =\ g\ f\ \textbf{in}\ h\ h \\ g\ f\ x = \ \textbf{let}\ y\ =\ x\ x\ \textbf{in}\ f\ y \end{array}$$

4. Schließlich wird die Hilfsfunktion g wieder lokal gemacht:

   $$\begin{array}{l} y\ f = \ \textbf{let}\ g\ f\ x = \ \textbf{let}\ y\ =\ x\ x\ \textbf{in}\ f\ y \\ \qquad\qquad\ \ h\ \ \ \ = \ g\ f \\ \qquad\ \textbf{in}\ h\ h \end{array}$$

Damit hat die Definition eine Form, die direkt in ein Maschinenprogramm übersetzt werden kann. Bleibt noch eine geeignete Reduktionsstrategie zu wählen. Als Argumente des Y-Kombinators kommen einstellige Funktionen auf einem beliebigen Typ τ in Frage. Es ist bekannt, daß diese Funktionen in ihrem Argument nicht strikt sein dürfen, also kommt der Typ $\alpha \rightarrow \alpha$ in einem strikten Bezugssystem nicht in Betracht. Wir verwenden stattdessen vollständige Closures als Eingaben:

$$\textbf{type}\ \mathit{schema}(\alpha) = (\rightarrow \alpha) \rightarrow \alpha$$

Beispiel 7.6. Der Y-Kombinator in obiger Normalform mit anonymer Hilfsfunktion g:

$$\begin{array}{l} \textbf{fun}\ Y(\alpha) : \mathit{schema}(\alpha) \rightarrow \alpha = \\ [\end{array}$$

```
        param   1                                   // return
        param   0                                   //    f
        curry   fun : schema(α), auto(α) → α =      // let g f x =
```

```
        [
                param   2                       // return
                param   1                       //    x
                dup
                apply   1                       // let y = x x
                param   0                       //    f
                apply   1                       // in f y
                eval                            // .
        ]
        apply   1                               // let h = g f
        dup
        apply   1                               // in h h
        eval                                    // .
]
```

□

Als weiterer Beleg für die in Kapitel 1 aufgestellte Behauptung, Zyklen würden bei der semantischen Analyse leicht vernachlässigt, mag die folgende Anekdote dienen: Formuliert man das Programm in Haskell in naiver Form, dann wird es von der Typinferenz erwartungsgemäß zurückgewiesen:

$$y\ f = \mathbf{let}\ g\ =\ (\backslash\ x\ \rightarrow\ f\ (x\ x))\ \mathbf{in}\ g\ g$$

Codiert man dagegen den kritischen zyklischen Typ wie oben als freien Datentyp, dann ist das Programm laut Haskell-Spezifikation typkorrekt:

$$\begin{array}{ll}\mathbf{data}\ \mathit{Auto}\ \alpha = & \mathit{Auto}\ \{\ \mathit{applyAuto}\ ::\ \mathit{Auto}\ \alpha\ \rightarrow\ \alpha\ \} \\ y\ f & = \mathbf{let}\ g\ =\ \mathit{Auto}\ (\lambda\ x\ \rightarrow\ f\ (\mathit{applyAuto}\ x\ x)) \\ & \quad\ \mathbf{in}\ \mathit{applyAuto}\ g\ g\end{array}$$

Allerdings reagiert der populärste Haskell-Compiler ghc in den aktuellen Versionen 6.2.2 und 6.4 mit Nichttermination der semantischen Analyse. Sein Konkurrent hugs 2002 hat keine Probleme und kann auch folgende Fixpunktdefinition der Fakultätsfunktion auswerten:

$$\begin{array}{l}\mathit{fac} = \ y\ \mathit{fac}' \\ \quad \mathbf{where} \\ \quad\quad \mathit{fac}'\ f\ n = \mathbf{if}\ n\ =\ 0\ \mathbf{then}\ 1\ \mathbf{else}\ n\ *\ f\ (n-1)\end{array}$$

Beide Compiler können dagegen mit folgender punktfreier, aber schwer lesbarer Definition umgehen:

$$\begin{array}{ll}\mathit{self}\ (\mathit{Auto}\ x) = & x\ (\mathit{Auto}\ x) \\ y\ f & = (\mathit{self}\ \circ\ \mathit{Auto})\ (f\ \circ\ \mathit{self})\end{array}$$

Kapitel 8

Implementierung

In diesem Kapitel sollen die verschiedenen Komponenten, die zusammen die Implementierung des Malice-Systems bilden, beschrieben werden. Die Sichtweise ist dabei wesentlich gröber als im vorigen Kapitel. Neben einer Entwurfsbeschreibung und der Aufzählung dessen, was die Werkzeuge für den Nutzer leisten, soll auch die Effizienz des V→M-Modells diskutiert werden. Dies geschieht auf zweierlei Weisen:

1. Es wird untersucht, wie direkt sich die Operationen der Maschine auf konventionelle Maschinen oder allgemein imperativen Code abbilden lassen. Die Compilation des Maschinencodes in die imperative Zwischensprache IMPL und deren Optimierung sind konstruktive Belege für die effiziente Ausführbarkeit des Modells.

2. Die verschiedenen Ausführungsumgebungen für V→M-Programme werden anhand von Laufzeitmessungen untereinander und mit anderen Sprachimplementierungen verglichen. Damit soll weniger die Qualität der Codeerzeugung und -ausführung, als vielmehr die prinzipielle Konkurrenzfähigkeit der zyklischen Berechnungstechniken belegt werden.

8.1 Interpreter

Die Spezifikation der Maschine V→M läßt sich mit geringem Aufwand direkt implementieren. So erhält man einen Interpreter, der die Abläufe auf der Maschine präzise simuliert. Dieser kann als Experimentierumgebung für einfache zyklische Programme und als Referenz zum Validieren fortgeschrittener Ausführungsumgebungen dienen.

8.1.1 Programmmodell

Die Implementierung des Programmmodells, dessen Objekte Programme der Virtuellen Maschine und deren Bestandteile repräsentieren, in Java folgt weitestgehend den Spezifikationen in Kapitel 7. Aus praktischen Gründen wurde eine Darstellung auf zwei Abstraktionsebenen gewählt: Die Zugriffsoperationen für Typen und Code (Funktionen, Konstanten und Anweisungen) sind jeweils als abstrakter Datentyp modelliert, für den eine direkte Implementierung der Elemente in Java-Klassen zur Verfügung steht. Ebenso ist das Heap-Modell umgesetzt. Alle Werkzeuge des Malice-Systems greifen ausschließlich über die abstrakte Schnittstelle auf Programme zu und können mit konkreten Realisierungen konfiguriert werden. Der Austausch der Realisierung eröffnet unter anderem folgende Möglichkeiten:

1. Durch Vorschalten einer Proxy-Realisierung kann eine Instrumentierung bei ansonsten unveränderter Funktionalität erreicht werden. Dies dient zum Beispiel der Beobachtung von Zustandsübergängen in einer interaktiven Oberfläche oder statistischen Erhebungen.

2. Durch Realisierungen mit eigener Codierung der Elemente können Programme auf andere Datenformate abgebildet werden, etwa eine externe XML- oder Bytecode-Repräsentation. Eine extreme Anwendung dieses Prinzips ist das *reflektive* Programmmodell, welches Typen und Code als Speicherzellen auf dem Heap der Maschine, genauer gesagt auf einem konfigurierbaren Heap-Modell, codiert. Damit kann das Programm zur Laufzeit analysiert oder generiert werden. Entsprechende Typen und Hilfsfunktionen sind Bestandteil der Bibliothek, siehe Abschnitt 8.4.5.

8.1.2 Parser

Ein in der Sprache TASM geschriebenes Programm kann mittels des TASM-Parsers in ein Programmmodell übersetzt werden, das dann für Analyse und Ausführung zur Verfügung steht. Neben der Prüfung der syntaktischen Korrektheit werden dabei einige Phänomene von TASM auf primitivere Konstrukte des Programmodells abgebildet:

1. *Module und Import.* Importierte Module werden rekursiv parsiert und verwendete Definitionen in das importierende Programmmodell übernommen. Zyklische Importe werden derzeit nicht unterstützt. Die Einbindung von Modulen erfolgt auf der Quelltextebene, es wird also stets das gesamte Programm, sprich der transitive Abschluß der Importrelation, parsiert. Das modulare Hinzuladen von Programmodellen ist derzeit nicht unterstützt.

2. *Applikative Notation für Typparameter.* Die funktionsartige Notation für Typparameter in TASM wird auf freie Typvariablen und explizite Instanziierungen abgebildet.

3. *Alias-Definitionen.* Alle Referenzen per Name auf Programmelemente (Typen, Konstanten, Prozeduren) werden bei der Erzeugung des Programmmodells aufgelöst. Definitionen, die ein existentes Element einem neuen Namen zuordnen (Aliase), spielen daher keine besondere Rolle. Zyklische Aliase, die letztendlich nicht auf ein bereits bekanntes Element verweisen, werden erkannt und zurückgewiesen.

8.1.3 Ausführung

Die Ausführung eines Maschinenprogramms durch den Interpreter folgt exakt der Spezifikation in Kapitel 7, erweitert um eine Anbindung an den Kontext. Dazu gliedert sich der Interpreter in vier Bereiche:

Global Für den Aufruf des Interpreters im Kontext steht eine Konfigurationsschnittstelle für das Programmmodell, sowie globale Variablen zur Aufnahme der Ausgabe- und Differenzparameter der Einstiegsprozedur zur Verfügung. Auch die Initialisierung von Konstanten kann über diese Schnittstelle ausgelöst werden.

Interprozedural Der Prozeduraufruf wird implementiert durch Operationen zum Auf- und Abbau des Stacks, Transfer von Parameterwerten und Zyklenerkennung.

Intraprozedural Innerhalb einer Prozedur wird der Code traversiert und es finden Zustandsübergänge entsprechend der Spezifikation statt.

Primitiv Abstrakte Prozeduren, denen kein Maschinencode zugeordnet ist, können durch primitive Prozeduren direkt auf der Java-Plattform implementiert werden. Diese werden bei Bedarf dynamisch geladen. Die implementierende Klasse wird durch das Meta-Attribut `native` an der Prozedurdeklaration festgelegt. Zur vereinfachten Handhabung in der Eingabesprache TASM kann dieses Attribut auch auf Modulebene angegeben werden und wird dann an alle Prozeduren des Moduls vererbt.

8.1.3.1 Rückruf

Der Kontrollfluß kann den Interpreter in zwei Situationen verlassen: Bei der Initialisierung von Konstanten, deren Ablauf vom Programmmodell diktiert wird, und bei der Ausführung primitiver Prozeduren. Für beide steht jeweils ein Rückrufmechanismus zur Verfügung. Im Fall der Initialisierung verzweigt dieser zur Ausführung der Initialisierungsprozedur einer Konstanten zurück in den Interpreter. Im Fall der primitiven Prozedur stehen Äquivalente der Anweiungen `call`, `eval` und `yield` zur Verfügung, um auf nicht-primitive Programmteile zuzugreifen.

8.1.3.2 Nichtlokale Sprünge

Der Interpreter unterscheidet neben Fehlern drei Arten von Unterbrechungen des normalen rekursiven Ablaufs, die durch den *Exception*-Mechanismus von Java realisiert werden:

Abbruch Globaler Abbruch der Berechnung, ausgelöst durch die Ausführung einer `halt`-Anweisung oder einen erkannten Fehler. Die *Exception* wird vom aufrufenden Kontext der Berechnung gefangen.

Rücksprung Lokaler Abbruch der ausgeführten Prozedur, ausgelöst durch eine `return`-Anweisung. Die *Exception* wird auf der Ebene der Prozedurinkarnation gefangen.

Wiederholung Lokaler Abbruch und Neustart der ausgeführten Prozedur, ausgelöst durch eine `loop`-Anweisung. Die *Exception* wird in einer Schleife auf Ebene der Prozedurinkarnation gefangen. Diese Schleife wird erst durch einen anderen Abbruchfall oder das Erreichen des Endes des Prozedurrumpfs beendet. Damit wird direkte *tail*-Rekursion unbeschränkter Tiefe ermöglicht. Allerdings findet der Sprung nur dann statt, wenn zuvor das Überschreiben der Prozedurparameter für sicher befunden wurde, also:

1. die Prozedur entweder keine Zyklenbehandlung besitzt, oder
2. andernfalls als homomorph markiert ist und alle relevanten Parameter mit unmarkierten Zellen belegt sind.

In allen anderen Fällen wird `loop` wie `call` behandelt, damit die Zyklenerkennung gewährleistet bleibt.

8.1.3.3 Fehlererkennung

Der Interpreter erkennt Fehler im Programm grundsätzlich nur zur Laufzeit, unterscheidet also nicht zwischen statischen und dynamischen Fehlern. Um einen fehlerhaften Ablauf möglichst früh, sprich an der ursächlichen Fehlerstelle abbrechen zu können, werden zur Laufzeit die Typen aller lokalen Werte mitgerechnet und Überprüfungen vorgenommen, die die Erzeugung nicht typkorrekter Daten verhindern. Dazu wird bei jeder Zuweisung (`set`) und bei jeder Übergabe eines Parameterwerts (`call`, `apply`, `eval`, `loop`) der Typ von Wert und aufnehmender Variable verglichen. Auch beim Zugriff auf Cotupel (`body`) findet immer eine Überprüfung der Variante statt. Zugriffe auf uninitialisierte Speicherstellen (`get`, `dito`) werden ebenfalls abgefangen.

8.2 Compiler

Die Implementierung des Interpreters ist alles andere als eine maschinennahe Plattform: Erstens ist der Entwurf der abstrakten Maschine eher an traditionelle mathematische Berechnungsmodelle als an reale Prozessoren angelehnt. Zweitens abstrahiert Java als Implementierungsplattform bereits von vielen systemnahen Details. Drittens können einige technische Eigenheiten der zyklischen Berechnung, insbesondere der Umgang mit Referenzen, wegen der Sicherheitsrestriktionen der Java-Maschine nur sehr indirekt umgesetzt werden. Folglich hat der Interpreter höchstens Referenz- und Simulationscharakter.

Um die Frage zu klären, inwieweit sich in dieser Form repräsentierte zyklische Programme auch mit realistischer Effizienz ausführen lassen, wurde ein Compiler mit einigen Transformations- und Optimierungsphasen sowie einem dynamischen und einem statischen Codegenerator entwickelt.

Die Architektur des implementierten Compilers folgt weitgehend traditionellen Entwurfsmustern:

1. Die syntaktische Analyse entfällt, da der Compiler auf dem Programmmodell der virtuellen Maschine aufsetzt, und daher den Parser und Lader für die Eingabesprache TASM gemeinsam mit dem Interpreter benutzen kann.

2. Die statische Überprüfung der Semantik ist mit der Transformation in eine Zwischenrepräsentation IMPL verknüpft. Hier werden statische Typisierung, Kontextüberprüfungen und die Elimination von Oberflächenphänomenen durchgeführt. Fehlende Typen von insitu-Bindungen werden inferiert.

3. Ist das Programm statisch wohlgeformt, können Transformationen auf dem IMPL-Format folgen, die der Normalisierung und Optimierung dienen.

4. Abschließend wird Code für andere Ausführungsumgebungen erzeugt. Dabei stehen zwei Generatoren zur Verfügung. Der eine erzeugt ein C-Programm zur Ausführung auf konventionellen Maschinen, während der andere für den *just-in-time*-Betrieb ausgelegt ist. Der von letzterem erzeugte Code kann transparent mit interpretierten Programmteilen kombiniert werden. Außerdem steht diesem ein Rückkopplungsweg zur Verfügung, der aus einer nichtleeren Closure zur Laufzeit spezialisierten IMPL-Code erzeugt.

8.2.1 Die Zwischensprache IMPL

Da diese Sprache ausschließlich für den Gebrauch innerhalb des Compilers bestimmt ist, besitzt sie keine Eingabesyntax, sondern ist als Objektmodell spezifiziert. Es handelt sich um eine einfache imperative Sprache im Stil von C, erweitert um die spezifischen Merkmale der Zyklenerkennung. Anstelle eines Operandenstacks dienen deklarierte Variablen und explizite Zuweisungen und Übergaben der Spezifikation von Datenfluß.

Wegen der fehlenden formalen Syntax wurde in diesem Abschnitt auf den Abdruck von Beispielen verzichtet. Stattdessen wird auf das Beispielverzeichnis der Implementierung verwiesen, in dem sich zahlreiche Beispielprogramme finden, die besonders die Arbeitsweise des Compilers illustrieren. Mit Hilfe der graphischen Benutzeroberfläche können diese interaktiv nachvollzogen werden.

8.2.1.1 Programm

Ein IMPL-Programm gliedert sich ebenso wie ein Maschinenprogramm in Typ-, Konstanten- und Prozedurdefinitionen.

8.2.1.2 Typen

Das Typsystem entspricht dem der virtuellen Maschine, allerdings ohne Typvariablen und Belegungen. Diese müssen also von der Eingabetransformation eliminiert werden. Ein Programm, bei dem offene Typvariablen zurückbleiben, oder die Substitution divergiert, wird zurückgewiesen.

8.2.1.3 Operanden

Der Operandenstack wird durch eine einfache Transformation eliminiert, wie sie von *static-single-assignment*-Zwischensprachen bekannt ist. Dabei wird jede Bindung auf dem Stack durch Deklaration und initialisierende Zuweisung einer lokalen Konstanten ersetzt. Da Operanden nicht über Blockgrenzen hinweg wirken können, ist das Verfahren hier besonders einfach: Die so erzeugten Konstanten erhalten ihren Wert immer aus genau einem Zweig des Kontrollflusses. Es sind keine symbolischen Ausdrücke zur Zusammenführung (φ-Knoten) nötig. Da Tiefe und Typisierung des Stacks an jeder Stelle statisch eindeutig ist, kann jeder erzeugen Konstanten in einem linearen Durchgang ein exakter statischer Typ zugeordnet und gleichzeitig ein Unterlauf ausgeschlossen werden.

Da die Operanden auf dem Stack die einzigen nicht explizit typisierten Größen in einem Maschinenprogramm darstellen, ist ein IMPL-Programm also vollständig statisch typisiert. Damit stehen eine Reihe von Möglichkeiten zur typgesteuerten Analyse und Optimierung offen.

Die erzeugten Konstanten werden mit ihrer Position auf dem ehemaligen Operandenstack annotiert, so daß dieser bei Bedarf in der Codegenerierungsphase rekonstruiert werden kann.

8.2.1.4 Anweisungen

Der Code von IMPL-Programmen gliedert sich wie gehabt in geschachtelte Blöcke. Daneben steht eine geringe Anzahl von Operationen zur Verfügung. Operanden werden nicht implizit über einen Stack zugeordnet, sondern explizit aufgezählt.

Deklaration Jede lokale Größe, sprich Variable, Konstante oder insitu-Bindung besitzt eine Deklaration. Die Gültigkeit erstreckt sich potentiell von der Deklarationsstelle bis zum Ende des unmittelbar umgebenden Blocks.

Datenfluß Die einzigen beiden Operationen zur Bewegung von Daten sind die Zuweisung (entspricht der `set`-Operation) und die `dito`-Operation.

Kontrollfluß Hier stehen Operationen für `return` und `halt`, `call` und `eval`, sowie `switch` zur Verfügung. Letztere Operation erhält neben den Fällen einen weiteren Block, die sogenannte *Fortsetzung*. Der Zweck wird in Abschnitt 8.2.2.4 erläutert.

8.2.1.5 Ausdrücke

Der reduzierte Anweisungsvorrat wird mit einer zusätzlichen Sprachebene erkauft. Neben Bezeichnern von Speicherstellen können auch komplexere Ausdrücke als Operanden auftreten. Eine wesentliche Entwurfsentscheidung für die virtuelle Maschine, nämlich die Unabhängigkeit von Auswertungsreihenfolgen, wird dadurch jedoch nicht verletzt: Ausdrücke mit Seiteneffekt, wie zum Beispiel Konstruktoraufrufe, treten ausschließlich als Wurzel der rechten Seite einer Zuweisung auf, so daß die Reihenfolge der Effekte explizit bleibt.

Die Bedeutung von Ausdrücken wird angelehnt an die Sprache C in zwei Kontexte unterschieden. Im *Wertkontext* steht ein Ausdruck für seinen Wert (*rvalue* in C). Im *Referenzkontext* dagegen steht ein Ausdruck für seinen Speicherort (*lvalue* in C). Der Referenzkontext tritt auf der linken Seite von Zuweisungen auf, ansonsten gilt der Wertkontext. Jeder Referenzausdruck ist auch ein Wertausdruck, aber nicht umgekehrt.

Stackzugriff Nullstellige Ausdrücke zum Zugriff auf Parameter und lokale Größen. Nur Variablen und insitu-Bindungen sind Referenzausdrücke. Ein Variablenausdruck im Wertkontext impliziert also eine `get`-Operation.

Heapzugriff Einstellige Ausdrücke entsprechend `field` und `body`, jeweils für einen gegebenen Index. Zugriffe auf den Rumpf eines Cotupels existieren in zwei Varianten, mit und ohne Variantentest. Mit aktiviertem Test entspricht die Semantik der `body`-Operation. In `switch`-Zweigen oder mit

Hilfe von Datenflußanalyse kann dieser Test aber aus Effizienzgründen oft deaktiviert werden. Alle Heapzugriffe haben Referenzbedeutung.

Programmzugriff Nullstellige Ausdrücke zum Zugriff auf initiale Closures gegebener Prozeduren und den Wert gegebener Konstanten. Dies sind keine Referenzausdrücke.

Konstruktion Nullstellige Ausdrücke zur Konstruktion uninitialisierter Tupel und Cotupel, entsprechend `tuple`- und `cotuple`-Operation, sowie n-stellige Konstruktion von Closures, entsprechend `apply`. All dies sind keine Referenzausdrücke.

Referenzen Einstellige Ausdrücke zur Referenzierung und Dereferenzierung. Referenzierung ordnet einem Referenzausdruck seine Adresse als Wertausdruck zu. Dereferenzierung bildet umgekehrt einen Referenzausdruck zu der gegebenen Adresse.

Dynamik Wertausdrücke für tatsächliche Werte. Da die Maschine keine Literale kennt, können diese nur durch Konstantenauswertung zur Compilerzeit oder durch Rückkopplung im *just-in-time*-Compiler auftreten. Ausdrücke und Anweisungen mit tatsächlichen Werten als Operanden lassen sich oft vereinfachen.

Sonstiges Ein konstanter undefinierter Ausdruck, entsprechend der `halt`-Operation.

8.2.2 Transformationen

Alle Transformationen sind so implementiert, daß sie lokal auf einzelne Prozedurrümpfe angewendet werden. Jede Transformation besteht aus einer oder mehreren linearen Traversierungen des Codes. Findet eine Ersetzung statt, werden gegebenenfalls Folgetransformationen empfohlen. So kann eine Kombination von Transformationen einerseits als statische Sequenz mit gut abschätzbarem Aufwand realisiert werden, ohne daß die Empfehlungen beachtet werden. Andererseits kann auch eine heuristische Fixpunktsuche stattfinden, indem die Empfehlungen in eine priorisierte Warteschlange eingereiht und erschöpfend oder bis zu einer Aufwandsgrenze abgearbeitet werden. Somit läßt sich die Qualität des erzeugten Codes im Verhältnis zur Laufzeit des Compilers auch für den *just-in-time*-Betrieb ausreichend genau steuern.

Alle im folgenden beschriebenen Transformationen sind im Compiler implementiert, und stehen unter der grafischen Benutzeroberfläche zum Experimentieren zur Verfügung.

8.2.2.1 Zusammenfassung von Typen

Wie oben beschrieben, werden bei der Übersetzung nach IMPL alle Typsubstitutionen aufgelöst. Aufgrund der zyklischen Natur der Typcoterme ist es nicht wie bei azyklischen Termen möglich, bereits bei der Konstruktion vorhandene äquivalente Werte wiederzuverwenden, statt neue zu erzeugen. Das corekursive Vorgehen eignet sich nicht für die Technik des sogenannten *cons-cachings*. Redundante Strukturen können nur nachträglich erkannt und auf kanonische bisimilare Vertreter abgebildet werden. Die hier implementierte Transformation fasst alle bisimilaren Typcoterme zusammen, bildet also eine einfache Quotientencoalgebra der vorkommenden Typen. Dazu kann der Algorithmus von HOPCROFT[21] zur Minimierung von Automaten leicht auf beliebige polynomielle Signaturen verallgemeinert werden.

Im gleichen Durchlauf können logische Typvariablen eliminiert werden, die zur Typinferenz von insitu-Bindungen eingeführt wurden.

8.2.2.2 Propagation und Elimination von elementaren Bindungen

Durch Zuweisungen an lokale Konstanten werden Werte zur späteren Verwendung gebunden. Als *elementar* bezeichnen wir solche Bindungen, die keinen Seiteneffekt haben, und bei denen die Auswertungskosten der rechten Seite nicht die der linken übersteigen. Die so gebundene Konstante nennen wir eine *Kopie*. Solche Kopien treten vor allem in der Eingabetransformation auf. So erzeugt beispielsweise jede

`dup`-Operation eine, jede `swap`-Operation zwei Kopien. Eine wohlbekannte Transformation, die zahlreiche Analysen und Optimierungen begünstigt, besteht darin, an den Verwendungsstellen der Kopie das Original zu substituieren (propagieren). Die durch die Zuweisung ausgedrückte semantische Gleichung wird somit für alle nachfolgenden Transformationen explizit.

Im direkten Anschluß, aber auch als eigenständige Optimierung können elementare Bindungen, die nicht mehr verwendet werden, eliminiert werden. Damit gelingt unter anderem die vollständige Elimination der Zuweisungen zur reinen Manipulation des Operandenstacks, entsprechend `pop`, `dup` und `swap`.

Es ist aber nicht immer empfehlenswert, Kopien zu substituieren. Wird beispielsweise im generierten Code der Operandenstack tatsächlich aufgebaut, so besteht eine beliebte und elegante Technik darin, diesen beim Prozeduraufruf mit dem Frame der Inkarnation zu überlappen, um so das Kopieren der Parameterbelegungen einzusparen. In diesem Fall darf nicht substituiert werden, da die Deklaration der Kopie bereits die Parameterbelegung darstellt. Im implementierten Compiler kann die Propagation auf Parameterwerte ein- und ausgeschaltet werden. Der `C` erzeugende Codegenerator schaltet sie aus, der *just-in-time*-Generator dagegen ein.

Falls der Operandenstack tatsächlich aufgebaut werden soll, ist es auch erforderlich, nach der Erzeugung oder Elimination von lokalen Bindungen die Stacktiefe aller anderen Bindungen zu aktualisieren, um Lücken und Überlappungen zu vermeiden. Dazu sind zwei lineare Durchgänge nötig, wobei der erste die Lebensdauerintervalle aller Bindungen ermittelt und der zweite anhand deren Schachtelung einen kompakten Stack rekonstruiert.

8.2.2.3 Einschränkung des Kontrollflusses

Die Einfachheit der Kontrollstrukturen, die sich aus der funktionalen Natur des Anwendungsgebietes begründet, macht die Optimierung des Kontrollflusses innerhalb einer Prozedur recht einfach.

Falls die auszuwertende Closure einer `eval`-Anweisung zur Compilezeit bekannt ist, kann diese in eine äquivalente `call`-Anweisung, also der dynamische in einen statischen Prozeduraufruf übersetzt werden. Dadurch entfällt nicht nur der Zugriff auf die Closure-Parameter, sondern es besteht auch die Möglichkeit zur Substitution der aufgerufenen Prozedur. Darauf wird im nächsten Abschnitt näher eingegangen.

Die Schachtelung von Blöcken kann nach der semantischen Analyse eliminiert werden, um auch über Blockgrenzen hinweg Ansätze für Optimierungen finden zu können. Diese Normalisierung wird zweckmäßigerweise mit einer Abbruchanalyse kombiniert, die der Normalisierung von `return-` und `halt`-Anweisungen dient. Enthält ein Block eine solche Anweisung, so heißt er *abbrechend*. Eine Fallunterscheidung ist abbrechend, wenn es alle ihre Zweige sind. Alle folgenden Anweisungen sind unerreichbar (*dead code*) und können eliminiert werden. Anschließend werden die finalen Anweisungen der Prozedur traversiert: Ist die letzte Anweisung des Rumpfs eine Fallunterscheidung, sind rekursiv ihre Zweige zu besuchen. Ansonsten handelt es sich um eine finale Anweisung. Eine finale `return`-Anweisung ist redundant und kann eliminiert werden. Ein finaler Prozeduraufruf (*tail-call*) ist ein Kandidat für effiziente Ausführung ohne Allokation eines Frames, und wird daher markiert.

Unter Umständen können Zweige einer Fallunterscheidung beschnitten werden, nämlich wenn erstens der Bedingungsausdruck eine Konstante oder ein Konstruktor ist, oder wenn zweitens derselbe Wert auf demselben Pfad bereits getestet wurde. Bleibt genau ein Zweig übrig, kann der Test entfallen. Der erste Fall tritt typischerweise bei der Spezialisierung von Code auf und trägt wesentlich zum Spezialisierungsgewinn bei. Der zweite Fall ist bei handgeschriebenem Code eher unplausibel, kann aber durch eine weitere, spekulative Transformation entstehen:

Um die in einer Fallunterscheidung gewonnene Information im folgenden Code zu propagieren, kann dieser vervielfältigt und in alle Zweige einzeln eingefügt werden. Eine weitere Fallunterscheidung auf demselben Wert tritt dann geschachtelt in der ersten auf und kann in jedem Zweig beschnitten werden. Diese Transformation hat die unangenehme Eigenschaft, pathologischen Code exponentiell zu vergrößern, und sollte daher in sicherheitskritischen Kontexten nicht oder nur mit Ressourcenbeschränkung verwendet werden.

Die Cotupel-Analyse, die für die Beschneidung von Fallunterscheidungen verwendet wird, kann auch auf `body`-Operation angewendet werden, die normalerweise partiell definiert sind. Falls statisch gesichert

ist, daß die Operation stets wohldefiniert ist, wird sie entsprechend als sicher markiert. Im generierten Code kann dann die Überprüfung zur Laufzeit entfallen. Außerdem dient eine erfolgreich passierte `body`-Operation ebenso wie eine Fallunterscheidung als Informationsquelle für den folgenden Code. Stets undefinierte Zugriffe können dem Benutzer mit einer Warnung mitgeteilt werden.

8.2.2.4 Substitution von Prozeduren

Die Substitution von Prozedurrümpfen für Aufrufe (*inlining*) ist das zentrale Instrument der interprozeduralen Optimierung und für funktionales Programmieren, das Modularisierung und Wiederverwendung begünstigt, praktisch unabdingbar. Die Entscheidung, welche Prozeduraufrufe tatsächlich substituiert werden sollen, ist allerdings nicht leichtfertig zu treffen: Inlining wird in [50] plakativ als „*black art*“ bezeichnet, und kann allerlei Nebenwirkungen haben, von Codeexplosion bis hin zu Nichttermination des Compilers.

Der hier implementierte Ansatz ist einfach und pragmatisch: Ein spezielles Meta-Attribut `inline` an Definition oder `call`-Aufruf entscheidet über Kandidaten. Die mitgelieferten Beispielprogramme demonstrieren den gewinnbringenden Einsatz dieses Attributs durch den Programmierer. Ob und wie die Attributierung automatisiert werden kann, ist nicht Gegenstand dieser Arbeit.

Der sehr spezielle Vorrat an Kontrollstrukturen, der bisher immer als reiner Vorteil präsentiert wurde, sorgt für Komplikationen im Zusammenhang mit Codesubstitution:

1. Der Aufruf einer Prozedur mit Zyklenbehandlung kann nicht einfach durch einen Rumpf substituiert werden. Nicht nur, daß keine explizite Operation zur Zyklenerkennung vorhanden ist, diese wäre auch nicht effektiv, da mit der Elimination des Prozeduraufrufs auch dessen Inkarnations-Frame auf dem Stack veschwindet und somit nicht zur Erkennung eines Zyklus zur Verfügung steht. Da in Theorie und Praxis die zyklischen Verfahren an explizite rekrusive Aufrufe gebunden sind, lassen sich zyklische Prozeduren nicht effektiv substituieren.

   Diese Einschränkung auf den zweiten Blick wenig gravierend: Substitution ist vor allem bei nichtrekursiven Prozeduren gewinnbringend. Außerdem lassen sich, wie oben beschrieben, linear corekursive Aufrufe als Schleife zusammenfassen.

2. Mangels einer allgemeinen Sprunganweisung kann ein Rumpf, der eine `return`-Anweisung enthält, nicht bedeutungstreu substituiert werden. Da diese nach der oben beschriebenen Normalisierung nur noch am Ende von Zweigen einer Fallunterscheidung vorkommen können, läßt sich das Problem auch ohne die Einführung des gefürchteten `goto` lösen: Dazu werden alle Fallunterscheidungen gewaltsam in finale Position gebracht, indem sie einen zusätzlichen Fortsetzungsblock erhalten, der alle folgenden Anweisungen aufnimmt. Jeder Zweig wird markiert, ob er abbricht. Nur bei nichtabbrechenden Zweigen ist nach ihrem Ablauf in die Fortsetzung zu springen. Diese Lösung fügt der Kontrollflussanalyse keine nennenswerte Komplexität hinzu, gestattet aber die Elimination aller verbliebenen `return`-Anweisungen.

Wie bereits erwähnt, kann ein dynamischer `eval`-Aufruf zu einem statischen `call`-Aufruf vereinfacht werden, falls die in dem Closureausdruck gekapselte Prozedur bekannt ist. Für diesen Fall ist auch die Attributierung von `eval` mit `inline` vorgesehen. Das Attribut wird bei Vereinfachung vererbt und kann dann eine Substitution nach sich ziehen. Ist zusätzlich zur Prozedur auch mindestens ein in der Closure gekapseltes Argument bekannt, nämlich eine Konstante oder eine Prozedurreferenz, oder ein vom *just-in-time* Compiler substituierter Laufzeitwert, dann kann eine partielle β-Reduktion stattfinden. Ergebnis ist eine spezialisierte Prozedur, die mit den übrigen, unbestimmten Parameterausdrücken in einer neuen Closure gekapselt wird.

8.2.2.5 Sonstige Transformationen

Neben den bisher beschriebenen Transformationen sind noch einige sekundäre implementiert. Neben elementaren algebraischen Vereinfachungen, wie etwa der Aufhebung inverser Operatoren, ist hier noch

die Elimination von *insitu*-Variablen zu erwähnen: Falls ein *insitu*-Prozeduraufruf durch Inlining expandiert wird, und ein *insitu*-Parameter in genau einem Zweig gebunden wird, kann die an der Aufrufstelle deklarierte *insitu*-Variable durch Propagation des Parameterwerts eliminiert werden.

8.2.3 Just-in-time-Codegenerierung

Zwar sind Maschine und Compiler in Java implementiert, aber die Java-Plattform ist wegen ihrer Sicherheitsrestriktionen kein geeignetes Ziel für die direkte Codeerzeugung: Weder die effiziente Auswertung des Aufrufstacks, noch die für das corekursive Vorgehen essentielle dynamische Addressierung von einzelnen Zellenbestandteilen wird direkt unterstützt, für die Auswirkungen auf die Laufzeit siehe Abschnitt 8.5. Zur Emulation wurde ein abstrakterer Ansatz gewählt, der als *threaded code* bezeichnet wird, auch bekannt als *Kontrollflußgraph* oder *Spaghetti-Code*.

In einem solchen Graphen werden Anweisungen und Ausdrücke durch Java-Objekte repräsentiert, wobei jeder Ausdruck auf seine Argumente, und jede Anweisung auf ihren Nachfolger verweist. Die Operationalität der Bestandteile ist durch eine virtuelle Ausführungsmethode gegeben, die eine Repräsentation des Maschinenzustands übergeben bekommt und manipuliert. Dieser Zustand enthält neben der aktuellen Prozedurinkarnation auch Verweise auf die nächste Anweisung (Instruktionszeiger) und den Zustand bei Rücksprung, wodurch ein Aufrufstack aufgebaut wird. Der Übergang zur nächsten Anweisung wird durch Überschreiben des Instruktionszeigers angezeigt. Die Abarbeitung des Programms erfolgt ohne extensive Verwendung des Java-Stacks in einer Hauptschleife (Trampolin), wodurch der Ablauf leicht instrumentiert und beobachtet werden kann. Außerdem kann so jeder *tail*-Aufruf, der nicht in Konflikt mit der Zyklenerkennung steht, ohne zusätzlichen Stack-Speicher ausgeführt werden.

Der *just-in-time* compilierte Code verläßt sich ebenso wie der Interpreter auf die automatische Speicherverwaltung der gastgebenden JVM. Eine eigene explizite Speicherverwaltung wird nicht unterstützt. Die Rückrufmechanismen beider Umgebungen sind kompatibel, so daß die Implementierungen primitiver Prozeduren für den Interpreter unmittelbar wiederverwendet werden können.

8.2.3.1 Dynamische Spezialisierung

Ein *just-in-time*-Compiler arbeitet unter der äußerst günstigen Prämisse, daß ihm Informationen zur Verfügung stehen, die statisch entweder gar nicht oder nur durch aufwendige globale Analysen zu bekommen sind. Insbesondere können zur Laufzeit alle konkreten Variablenbelegungen zur Spezialisierung des Programms herangezogen werden.

Unser Compiler unterstützt diese Spezialisierung durch die `inline`-Attributierung von `apply`-Anweisungen. Dabei wird zur Laufzeit nicht einfach eine neue Closure produziert, die dieselbe Prozedur mit zusätzlichen Parametern kapselt. Stattdessen werden die akkumulierten Parameterwerte im Rumpf der gekapselten Prozedur substituiert (naive β-Reduktion). Dabei entsteht eine neue, anonyme Prozedur. Diese wird in einer Closure ohne Parameter gekapselt und auf dem Stack abgelegt. In diesem Zusammenhang ist sogar `apply` 0 sinnvoll.

Der unmittelbare Gewinn dieses Verfahrens besteht nur darin, daß die Parameterwerte nicht aus der Closure in den Aufrufstack kopiert werden müssen. Da die naive β-Reduktion als Traversierung des Prozedurcodes einen nicht zu vernachlässigenden Aufwand besitzt, amortisiert sich dieser Schritt nur bei sehr häufig aufgerufenen Closures. Der Gewinn läßt sich allerdings dramatisch steigern, wenn der β-reduzierte Prozedurrumpf einige Optimierungsphasen durchläuft. Da die substituierten Werte vollständig bekannt sind, greifen zahlreiche Vereinfachungen, die im statischen Fall nur für die relativ seltenen Konstanten und Konstruktorapplikationen gelten.

8.2.4 Generierung von C

Der C-Codegenerator für den Compiler ist ein Prototyp, der mit minimalem Aufwand belegen soll, daß zyklische Berechnungen auch unmittelbar auf realen Prozessoren ausgeführt werden können. Dabei

wurde C nicht wegen besonderer Eignung seiner Ausdrucksmittel gewählt, sondern weil die Sprache trotz Maschinennähe einfach zu generieren ist und gute, freie Compiler und Debugger existieren.

8.2.4.1 Übersetzungsschema

Die grundsätzliche Übersetzungsstrategie besteht darin, Typdefinitionen und die Stack-Frames von Prozeduren in C-Aggregattypen zu übersetzen. Prozeduren werden eins zu eins in C-Funktionen umgesetzt. Für einfache IMPL-Anweisungen und Ausdrücke wie Zuweisung und `switch` existieren direkte Entsprechungen, während komplexere Operationen wie `eval` als C-Hilfsfunktionen realisiert sind. Initialisierungsstatus und Werte von Konstanten werden in globalen Variablen gespeichert.

8.2.4.2 Aufrufkonventionen

Da die C-Spezifikation die Anordnung innerhalb eines Stack-Frames offenläßt, kann der eingebaute Parametermechanismus nicht für eine portable Implementierung von Zyklenerkennung verwendet werden. Stattdessen wird der V→M-Stack explizit als globaler Zeiger in einen eigenen Stack-Speicher modelliert. Weder für Parameter noch für lokale IMPL-Variablen wird das entsprechende C-Sprachmittel benutzt. In Prozessorstack und Registern werden also ausschließlich Rücksprungadressen und Zwischenwerte bei der Auswertung von Ausdrücken gespeichert. Dieses Vorgehen hat sowohl Vor- als auch Nachteile:

1. Alle Prozeduren werden als parameter- und ergebnislose C-Funktionen realisiert. Ein C-Compiler kann somit leicht alle *tail*-Aufrufe zu direkten Sprüngen optimieren.
2. Der V→M-Stack und damit die Schachtelung von Prozedurinkarnationen und die Wurzelmenge der erreichbaren Daten sind in portabler Weise zugreifbar. Nicht nur die Zyklenerkennung, sondern auch automatische Speicherverwaltung und Instrumentierung des Codes können so portabel gehalten werden und benötigen keine zusätzlichen Informationen oder Zusicherungen.
3. Andererseits werden Datenfluß-Analyse und Allokationsstrategien des C-Compilers praktisch außer Gefecht gesetzt. Hier ist durch die geringe Ausnutzung der Sprachmittel eine deutliche Einbuße an Effizienz zu erwarten, siehe Abschnitt 8.5.

Die Parameterübergabe beim Prozeduraufruf und die Verkettung der Frames für die Zylenerkennung werden durch zusätzlichen C-Code realisiert, der für jeden Aufruf generiert wird. Dabei wird auch die in Abschnitt 6.1.1 beschriebene Optimierung der Verkettung durchgeführt.

8.2.4.3 Zyklenerkennung

Die Erkennung von Zyklen und anschließende Verzweigung sowie die Behandlung durch die `dito`-Operation werden anders als in den Java-Implementierungen nicht durch generische Hilfsfunktionen realisiert, sondern für jede Prozedur in spezialisierten Code übersetzt.

8.2.4.4 Laufzeitsystem

Das Laufzeitsystem umfaßt nur das absolute Minimum an zusätzlichem C-Code, um einfache Programme ausführen zu können:

Speicherzellen Die Allokation von Speicherellen wird auf den üblichen C-Mechanismus (`malloc`) abgebildet. Eine automatische Speicherverwaltung und -freigabe ist nicht realisiert. Zu den dabei zu lösenden Schwierigkeiten siehe Abschnitt 6.4.

Closures Für den Umgang mit Funktionen höherer Ordnung stehen Hilfsfunktionen zum Aufbau (`apply`) und Abbau (`eval`) von Closures zur Verfügung.

Bisimilarität Der Parametervergleich modulo Bisimilarität für quasizyklische Funktionen ist nicht implementiert.

8.3 Grafische Oberfläche

Diese Arbeit stellt eine Neuformulierung etablierter theoretischer und praktischer Ansätze zur Zyklenbehandlung dar. Anstelle der klassischen relationalen Semantik und imperativen Programmiertechnik werden coalgebraische Semantik und funktionale Programmierung gesetzt. Der Grund für die Überarbeitung ist die Annahme, daß sich zyklische Algorithmen in der neuen Form eleganter darstellen lassen. Eine wesentliche Voraussetzung für die Entwicklung eleganter Algorithmen ist es aber, eine Intuition für die verwendeten Berechnungstechniken zu gewinnen:

1. Effektive Ansätze werden erst durch abstrakte Erkenntnisse über zyklische Datenstukuren ermöglicht. Dies kann an vielen, auf den ersten Blick verblüffenden Beispielen in Teil IV belegt werden, etwa der Subtraktion zweier Zahlen mit unendlich vielen Nachkommastellen oder dem Filtern einer unendlichen Liste.

2. Die Reihenfolge von Operationen spielt für die effektive Zyklenbehandlung eine wesentlich größere Rolle, als es in einem algebraischen System der Fall wäre. Um das Wirken von Zyklenerkennung und -behandlung nachvollziehen zu können, muß ein Ablauf als Folge von Zuweisungen betrachtet werden und kann nicht zu einer Reduktion abstrahiert werden. Diese Maxime, die bereits die Spezifikation der Maschine entscheidend geprägt hat, wirkt sich auch auf das empirische Verstehen von Programmen aus.

Für die weitere Erforschung der zyklischen Programmierung sind also Werkzeuge, welche Daten und Abläufe sichtbar machen, von großer Bedeutung. Folglich wurde bei der Implementierung des Malice-Systems neben den Analyse- und Ausführungswerkzeugen ebenso großer Wert auf eine Oberfläche gelegt, die das Experimentieren mit V→M-Programmen und insbesondere mit zyklischen Problemen angemessen unterstützt.

8.3.1 Editor

Der bisher einzig gangbare Weg, ein V→M-Programm zu erzeugen, besteht darin, TASM-Quelltext zu schreiben. Dafür besitzt die Oberfläche eine Komponente mit Anzeige- und Editorfunktionalität. Neben dem Laden und Speichern von Programmen und den Funktionen des in Java vordefinierten Editors leistet diese Komponente vor allem zweierlei:

1. Syntaktische Elemente von TASM werden in farblich und typografisch hervorgehoben (*highlighting*).

2. Mitgeführte Positionsinformation in anderen Aggregatzuständen des Programms ermöglich die interaktive Rückverfolgung von Verweisen, wie etwa Fehler- und Definitionsstellen.

Die jeweils nachfolgenden Verarbeitungstufen eines Programms werden per Knopfdruck erzeugt und angezeigt.

8.3.2 Visualisierung

8.3.2.1 Visualisierung von Programmen

Alle drei Formen von abstrakter Syntax, nämlich V→M-Modell, IMPL-Modell und JIT-Code, können in hierarchischen Baumansichten betrachtet werden. Für Codeblöcke kann auf eine kompaktere Textdarstellung umgeschaltet werden. Alle Programmelemente lassen sich im Quelltext zurückverfolgen.

8.3.2.2 Visualisierung von Abläufen

Die Ausführung von V→M-Code mit dem Interpreter und von JIT-Code ist an die Oberfläche angeschlossen. Die Abläufe können schritt-, block- oder Prozedurweise ausgeführt und die Maschinenzustände beobachtet werden.

8.3.2.3 Visualisierung von Daten

Die automatische Darstellung zyklischer Datenstrukturen (Coterme) ist ein nichttriviales Problem, für das auch hier keine Ideallösung gefunden werden konnte. Stattdessen wird eine Kombination von zwei komplementären Sichtweisen angeboten:

1. Eine Baumansicht stellt die nichtfundiert algebraische, potentiell unendliche Pfadstruktur dar. Verzweigungen können unbeschränkt expandiert werden. Die Zyklenerkennung ist dem Benutzer überlassen.

2. Eine Graphansicht stellt den Coterm exakt dar. Die Anordnung des Graphen in der Zeichenebene wird durch komplexe Heuristiken geregelt, die eine Spezialisierung verbreiteter Techniken auf die Besonderheiten von Cotermen darstellen. Da sich frei verfügbare Werkzeuge zur Graphdarstellung als inadäquat erwiesen haben, mußte ein eigenes entwickelt werden.

Die Graphdarstellung folgt der bewährten physikalischen Metapher, bei der Knoten mit Masse und abstoßender Ladung und Kanten als elastische Federn modelliert werden. Die Anordnung ergibt sich dann durch diskrete Simulation einer gedämpften Dynamik des Systems. Dieses Grundmodell wird in vierfacher Hinsicht auf Coterme angepaßt:

1. *Auslaufende Kanten haben eindeutige Rollen.* Das Graphmodell eines Coterms basiert auf der Erreichbarkeitsrelation zwischen Elementen. Verschiedene erreichbare Nachbarn können aber im lokalen Datensatz eines Elements unterscheidbare Rollen besitzen, beispielsweise als Felder eines Tupels. Dies wird einerseits durch gut lesbare Kantenbeschriftungen unterstützt, andererseits durch die Möglichkeit, auslaufende Kanten optisch auch in der *Knotenbeschriftung* statt am Rand zu verankern.

2. *Schleifen und Mehrfachkanten sind die Regel.* Die optische Darstellungsqualität von Kanten, die nicht zum spannenden Baum gehören, wird zu Lasten der Komplexität dadurch erheblich verbessert, daß Kanten nicht als Strecken, sondern als Kurven mit unsichtbaren Kontrollpunkten (*Splines*) dargestellt werden. Die Kontrollunkte besitzen ebenfalls Masse und Ladung, und sorgen so für ergonomische und deutliche Linienführung. Zusätzlich werden Rückwärts- und Kreuzkanten farblich hervorgehoben.

3. *Coterme besitzen eine Wurzel.* Der Wurzelnoten eines Coterms wird dadurch optisch unmißverständlich ausgezeichnet, daß der gesamte Graph, um bei der physikalischen Metapher zu bleiben, an seiner Wurzel in einem homogenen Schwerefeld aufgehängt wird.

4. *Viele Coterme teilen sich eine erzeugende Coalgebra.* Neben der Navigationsmöglichkeit, die Nachbarschaft von Knoten ein- und auszublenden, wird auch der Übergang zu einem anderen Wurzelknoten unterstützt. Dabei sind zwei Techniken zu unterscheiden:

   (a) Ein anderer Knoten kann optisch zur Wurzel gemacht werden, indem er zum Aufhängungspunkt im Schwerefeld gemacht wird. Durch Animation des Übergangs wird dieser optisch nachvollziehbar.

   (b) Dual zum bereits erwähnten *Ausblenden*, bei dem auslaufende Kanten abgeschnitten werden, ist die Technik des Einschränkens, bei dem der Graph auf die transitiv erreichbare Umgebung eines Knotens reduziert wird. Es wird also der minimale Coterm zu einer gegebenen Wurzel gebildet.

Die ersten drei Techniken sind statischer Natur und können in Abbildungen festgehalten werden. In den Abbildungen 1.1 und 1.2 sind alle beschriebenen Effekte bis auf die Kantenbeschriftung zu beobachten. Abbildung 8.1 zeigt einen großen Graphen mit Kantenbeschriftung, in dem Teile ausgeblendet wurden, um eine Baumform zu erhalten, sowie einen Graphen mit nichttrivialen Zyklen.

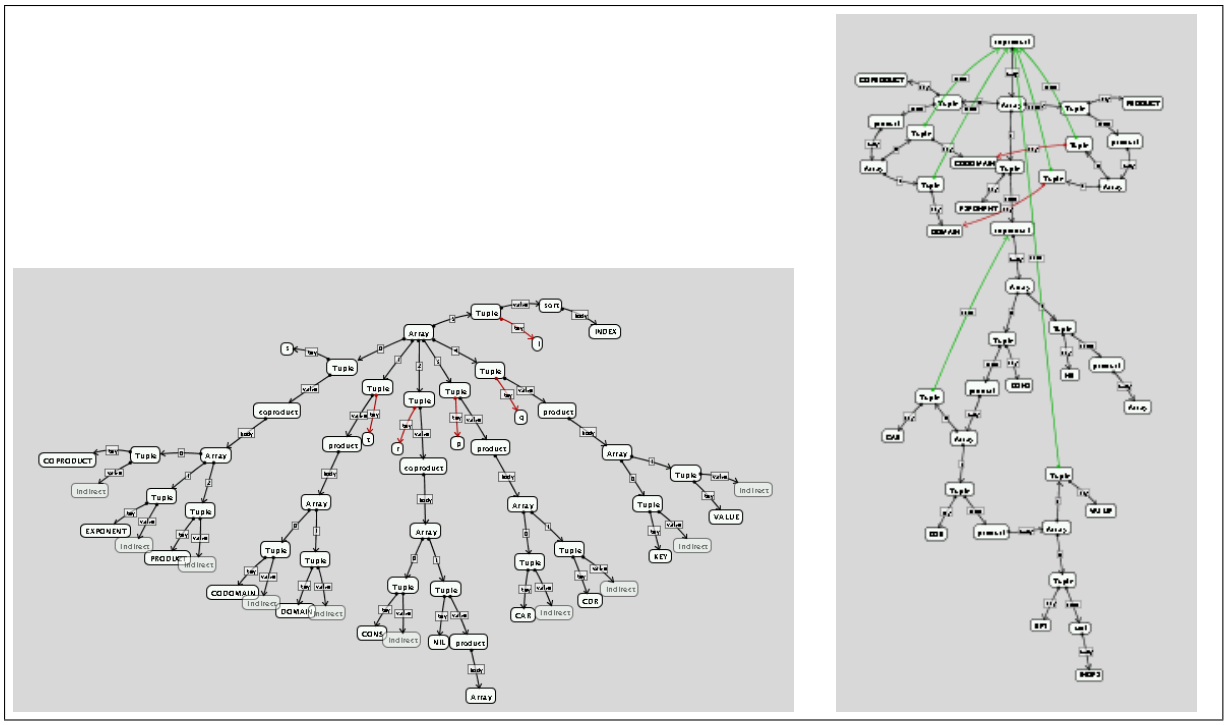


Abbildung 8.1: Komplexe Graphansicht

8.3.3 Interaktion

8.3.3.1 Auswertung

Sowohl das V→M- als auch das JIT-Modell unterstützen die Auswertung beliebiger Konstanten und Prozeduren ohne Eingabeparameter, sowohl mit als auch ohne Visualisierung des Ablaufs. Die Ergebnisse können anschließend in der Datenanzeige wie eben beschrieben betrachtet werden.

8.3.3.2 Ein-/Ausgabe

Neben der funktionalen Ein- und Ausgabe über Parameter steht auch eine Konsole für Benutzerinteraktion im Seiteneffekt zur Verfügung. Diese wird über eine primitive Implementierung abstrakter Prozeduren vom Programm aus verwendet. Für die in Anhang A und B entwickelten Algorithmen existieren einige interaktive Testprogramme mit textueller Benutzerschnittstelle.

8.3.3.3 Optimierung

Als anschauliches Werkzeug sowohl zum Test der implementierten Compilerphasen, als auch zum Verständnis von V→M-Programmen, dient ein interaktiver Optimierer, in dem Transformationen der Zwischensprache IMPL zu einer Sequenz angeordnet und abgearbeitet werden können. Der Effekt läßt sich in der textuellen Darstellung des IMPL-Programms gut beobachten. Der automatische Auswahlmechanismus des Compilers für Folgetransformationen kann ein- und ausgeschaltet und die Sequenz jederzeit revidiert werden.

8.4 Beispielbibliothek

Im Beispielverzeichnis des Malice-Systems befinden sich neben den im Anhang diskutierten Applikationen zyklischer Verfahren und Standardbeispielen der funktionalen Programmierung auch die nötige Infrastruktur für Implementierung und Test, so daß nur ein Minimum an ergänzendem Java-Code für die Ausführung nötig ist.

8.4.1 Basistypen

Verschiedene Arten von Basistypen sind samt ihrer elementaren Operationen in jeweils einem Modul vordefiniert:

1. Die abstrakten Typen `char`, `int32`, `real64` und `string`, die durch ihre Java-Pendants realisiert werden,
2. Die Aufzählungstypen `bool`, `void` und der Ergebnistyp `ord` für Vergleiche, welcher die drei Relationskonstanten $<$, $=$ und $>$ enthält.
3. Der polymorphe Produkttyp `pair` und der polymorphe Summentyp `maybe`.

8.4.2 Funktionenkalkül

Für das Arbeiten mit Funktionen (höherer Ordnung) stehen mehrere Module zur Verfügung:

`Function` definiert universelle Operationen auf Funktionen, wie Identität, Applikation und Komposition.

`Curry` definiert bespielhaft die Transformation zweistelliger Funtionen in CURRY-Form und zurück.

`Lazy` implementiert explizite *lazy*-Auswertung durch Closures. Ergebnisse werden im Seiteneffekt gespeichert, um redundante Auswertung zu verhindern.

`Upsilon` implementiert einen effektiven Y-Kombinator mit Hilfe zyklischer Typen, siehe Abschnitt 7.5.

8.4.3 Ein-/Ausgabe

Zur textuellen Interaktion stehen mit den Modulen `Parse` und `Show` Kombinatorsysteme für die Analyse und Generierung von Ausdrücken zu Verfügung. Für Basistypen und Datenstrukturen ist jeweils in einem zusätzlichen Modul eine kombinierbare Syntax definiert.

8.4.4 Datenstrukturen

An fortgeschrittenen Datenstrukturen existieren PEANO-codierte (erweiterte) natürliche Zahlen, Arrays und insbesondere polymorphe zyklische Listen, für die in Anhang B einige Algorithmen vorgestellt werden.

8.4.5 Programmmodell

Das Programmmodell der Maschine V→M wurde im Typsystem der Maschine selbst implementiert. Damit kann einerseits zur Laufzeit per *Reflection* und dynamische Codegenerierung auf das Programm zugegriffen werden, andererseits ist es auch möglich, das Programm auf dem Heap der Maschine abzulegen, beziehungsweise in einem *bootstrap*-Schritt durch ein anderes Maschinenprogramm, den Lader, erst zu erzeugen.

8.5 Evaluierung

Weder die Ergonomie der Eingabesprache TASM, noch die Performanz des Interpreters gestatten es, die Implementierung anhand einer realistisch großen Beispielanwendung zu testen und zu evaluieren. Um trotzdem einen ersten Eindruck von den Größenordungen des Laufzeitverhaltens zu gewinnen, wurde ein einfaches aber charakteristisches Referenzproblem in den verschiedenen Varianten des implementierten Systems, sowie einigen etablierten Programmiersprachen programmiert und untersucht.

8.5.1 Referenzproblem

Alle Implementierungen werden anhand der in Abschnitt B.2.1 diskutierten, wohlbekannten *map*-Funktion evaluiert. Dabei wird eine einstellige Funktion f auf eine Eingabeliste $\vec{l}$ angewendet.

1. Ist $\vec{l}$ leer, ist die Ergebnisliste ebenfalls leer.
2. Für $\vec{l} = a : \vec{r}$ ist das Ergebnis $\vec{m} = b : \vec{s}$, wobei $b = f(a)$ und $\vec{s} = map(f, \vec{r})$. Die Erzeugung der ersten Zelle von $\vec{m}$ kann vor der corekursiven Berechnung von $\vec{s}$ erfolgen; dazu dient eine Hilfsfunktion *semicons*, welche wie *cons* wirkt, aber die Restliste als Differenz- statt Eingabeparameter erwartet.

Wir verwenden $f = succ$, den freien Konstruktor der PEANO-codierten natürlichen Zahlen, sowie $\vec{l} = [1, \ldots, 1]$, wobei die Zahl 1 als Term $succ(zero)$ dargestellt ist und die Länge der Liste $\vec{l}$ als $|\vec{l}| = 2^n$ verschiedene Zweierpotenzen durchläuft. Die Erzeugung der Eingabeliste gehört dabei nicht zum beobachteten Programmablauf, sie ist daher in jeder Implementierung geeignet als Konstante zu definieren. Die Definition dieser Konstanten *many_ones* ist in den folgenden Beispielprogrammen nicht mit angegeben.

Die *map*-Funktion deckt die wichtigsten Aspekte der Ausführung funktionaler Sprachen ab: Fallunterscheidung über freien Datentypen, Erzeugung von Zellen, Rekursion und dynamischen Aufruf von Funktionsvariablen. Darüber hinaus hat eine einzige Definitionsgleichung sowohl die Form primitiver Rekursion als auch primitiver Corekursion. Folglich kann die Funktion gleichermaßen mit Auf- und Abwärtstraversierung implementiert werden. Letztere ist zusätzlich *tail*-rekursiv und kann durch Zyklenbehandlung zur Verarbeitung zyklischer Listen ergänzt werden.

8.5.2 Realisierung

Abbildung 8.2 zeigt den TASM-Code für die *map*-Funktion. Durch Weglassen der grau gefärbten Programmteile ergeben sich drei Varianten:

1. Werden die Attributierungen und der zweite Rumpf weggelassen, ergibt sich eine nur auf endlichen Listen definierte Funktion. Die Implementierung entspricht in etwa dem Schema der *tail*-Rekursion modulo Konstruktor, wie sie von optimierenden funktionalen Sprachen auf primitiv rekursive Definitionen angewandt wird. Diese Variante ist zwar weniger allgemein als solche mit Zyklenerkennung, hat dafür aber auch entsprechend weniger Zeit- und Platzbedarf zur Laufzeit. Sie wird im folgenden Abschnitt mit A bezeichnet.

2. Die Variante mit allen grauen Programmteilen umfaßt die optimierte Zyklenerkennung für homomorphe Interpretationen. Sie ist nur für passend markierte Eingaben korrekt. Sie wird mit B bezeichnet. Ihre Effizienz hängt stark davon ab, wie viele redundante Markierungen die Eingabe enthält. Im Idealfall, also einer Markierung bei zyklischen und keiner Markierung bei endlichen Listen, sollte der Aufwand kaum mehr als bei Variante A betragen.

3. Durch Weglassen der Attributierung ergibt sich die Variante mit dem maximalen Aufwand für Zyklenerkennung. Sie wird mit C bezeichnet. Sie erlaubt weder die Unterdrückung der Zyklenerkennung, noch die Optimierung von *tail*-Rekursion. Diese Variante ist daher nicht für die praktische Anwendung gedacht, sondern dient nur als Maß für die Leistung der homomorphen Optimierung (B).

```
fun map(α, β) @{hom} : (α → β) @{invar}, list(α) → list(β) =
[
        param   1                                       //    l
        switch                                          // switch l
        nil :                                           // case []
        [
                param   2                               // return...
                cotuple nil: list(β)                    // ... []
                set                                     // .
        ]
        cons :                                          // case (:)(p)
        [
                vars    α, list(α)                      //    a, r⃗
                var     1
                swap
                var     0
                swap
                call    decons(α)                       // (a, r⃗) ← p
                insitu  ^list(β)                        //    s⃗
                insitu  list(β)                         //    m⃗
                insitu  β                               //    b
                var     0
                get                                     //    a
                param   0                               //    f
                apply   1                               // f(a)...
                eval                                    // ... → b;
                call    semicons(β)                     // m⃗ ← b : s⃗;
                param   2                               // return...
                swap
                set                                     // ... m⃗;
                var     1
                get                                     //    r⃗
                param   0                               //    f
                loop                                    // s⃗ ← map(f, r⃗).
        ]
]
recursively
[
        dito
]
```

Abbildung 8.2: Die *map*-Funktion

```
fun benchmark : → =
[
        insitu  list(nat)
        yield   many_ones
        curry   succ
        call    map(nat, nat)
        pop
]
```

Abbildung 8.3: Testaufruf

Programmvariante A wird wie oben angegeben auf eine endliche Liste der Länge 2^n angewendet. Die beiden Varianten B und C werden zusätzlich auf eine Ringliste mit 2^n Elementen angewendet, was wir mit B' und C' bezeichnen. Dabei wird die für Variante B ideale Plazierung von Markierungen gewählt, die endliche Liste ist also vollständig unmarkiert, während der Verweis vom letzten auf das erste Element der Ringliste markiert ist. Die Messergebnisse für Variante B stellen also den besten Fall dar. Ob und wie häufig suboptimale Markierungen bei realen Anwendungen auftreten, ist wegen des Mangels an solchen noch nicht abzuschätzen.

Es sollen sowohl die Laufzeiten der verschiedenen Programmvarianten in einer Ausführungsumgebung, als auch die drei Umgebungen Interpreter, JIT-Compiler und C-Compiler miteinander verglichen werden.

8.5.3 Realisierungen in anderen Sprachen

Um nicht nur relative Aussagen über das Laufzeitverhalten treffen zu können, wurde Programmvariante A auch in anderen Sprachen implementiert und in professionellen, frei verfügbaren Ausführungsumgebungen getestet. Dabei wurden pro „Gewichtsklasse“ mehrere Kandidaten getestet, um neben dem Abstand unseres Prototypen auch die Varianz innerhalb der Klasse der ausgereiften Systeme zu ermitteln.

Beim Vergleich der Ergebnisse ist zu beachten, daß die funktionalen Vergleichssprachen strikte Semantik besitzen, also Berechnungstechniken verwenden, die nur den zyklenfreien Fall abdecken, während Programm A sich als Spezialfall einer allgemeineren, zyklentauglichen Berechnungstechnik ergibt. Die Effektivität des allgemeinen Verfahrens hatte bei dieser Arbeit Vorrang vor der Effizienz des speziellen. Die Vergleichsmessungen sollen klären, ob trotzdem eine akzeptable Geschwindigkeit erreicht werden kann.

8.5.3.1 Scheme

Eine der wenigen funktionalen Sprachen, die nicht nur in Spezialgebieten praktisch eingesetzt werden, ist Scheme. Als typische Implementierung verwenden wir MIT/GNU Scheme in der Version 7.7.1. Dieses System erzeugt Microcode für eine eigene virtuelle Maschine oder nativen Maschinencode. Eine *map*-Funktion ist in der Scheme-Standardbibliothek enthalten. Damit lautet das Testprogramm:

```
(define (succ x) (cons 'succ x))

(define (benchmark) (map succ many-ones))
```

Um das Optimierungspotential des Compilers auszunutzen, beginnt das Testprogramm wie empfohlen mit folgender Deklaration, die nicht zum Standardumfang von Scheme gehört:

```
(declare (usual-integrations))
```

Um zusätzliche Vergleichsmöglicheiten mit den Java-basierten Ausführungsumgebungen zu erhalten, testen wir außerdem die Scheme-Implementierung Kawa in der Version 1.8. Dieses System erzeugt Code für die Java-Maschine.

8.5.3.2 Prolog

Als logische Programmiersprache fällt Prolog zwar paradigmatisch aus dem Rahmen. Trotzdem lohnt sich ein Vergleich, da in einem logischen Formalismus mit Hilfe ungebundener Variablen die *top-down*-Erzeugung von Daten explizit ausgedrückt werden kann. Was in funktionalen Sprachen nur als Optimierung in einer für den Benutzer unsichtbaren Zwischensprache auftritt, beispielsweise die bereits erwähnte *tail*-Rekursion modulo Konstruktor, gehört in Prolog zu den Standardtechniken des Programmierens selbst.

Als Implementierung verwenden wir SWI Prolog in der Version 5.0.10. Dieses System erzeugt *byte*-Code für die WARREN-Maschine. Eine *map*-Funktion ist in der Prolog-Standardbibliothek enthalten. Damit lautet das Testprogramm:

```
succ(X, succ(X)).

benchmark :-
        many_ones(L),
        maplist(succ, L, Y).
```

8.5.3.3 ML

Als besonders weit verbreiteter und ausgereifter Vertreter der strikten funktionalen Programmiersprachen darf ML in diesem Vergleich nicht fehlen. Als Implementierung wurde Standard ML of New Jersey (SML/NJ) in der Version 110.59 gewählt. Dieses System erzeugt optimierten nativen Maschinencode. Eine *map*-Funktion ist in der ML-Standardbibliothek enthalten. Damit lautet das Testprogramm:

```
datatype nat = Zero | Succ of nat

fun benchmark() = List.map(Succ)(many_ones)
```

Da die vordefinierte Implementierung von *map* in CURRY-Form ist und auf gestaffelter Applikation beruht, wurde der Fairness halber eine zweite Version des ML-Programmes mit handgeschriebener zweistelliger *map*-Funktion erstellt:

```
fun map(f, nil) = nil
  | map(f, x :: r) = f(x) :: map(f, r)

fun benchmark() = map(Succ, many_ones)
```

Die beiden Varianten werden im folgenden Abschnitt als *ML* beziehungsweise ML' bezeichnet.

8.5.3.4 Opal

Eine weitere strikt funktionale Sprache, die sich durch hohe Effizienz bei gut vorhersagbarem Laufzeitverhalten auszeichnet, allerdings seit 1993 nicht mehr substantiell weiterentwickelt wurde, ist Opal. Als Implementierung wurde die Version 2.3h gewählt. Dieses System erzeugt C-Code. Eine *map*-Funktion ist in der Opal-Standardbibliothek enthalten. Anders als ihr ML-Pendant arbeitet diese Funktion intern mit einer zweistelligen Hilfsfunktion, so daß kein Laufzeitnachteil durch die CURRY-Form entsteht. Der vordefinierte Datentyp *nat* ist allerdings intern binär codiert, und muß daher redefiniert werden. Damit lautet das Testprogramm:

```
DATA nat == zero
            succ(pred : nat)
FUN benchmark : () -> seq[nat]
DEF benchmark() == map(succ)(many_ones)
```

8.5.4 Ergebnisse

Alle Tests wurden auf demselben PC mit der folgenden Ausstattung durchgeführt:

- Athlon XP 2500+ Prozessor, 1,83GHz/333MHz mit 512KB L2-Cache
- 512MB Speicher
- SuSE Linux 8.2

Alle Tests auf der Java-Plattform wurden mit dem Sun SDK in der Version 1.5.0_06 und der Server-Maschine, sowie 8MB Stack und 256MB Heap ausgeführt. Alle Tests auf der nativen Plattform wurden mit dem GCC in der Version 3.3.3 und den Einstellungen `-O2 -fomit-frame-pointer -mcpu=athlon` übersetzt.

Alle Diagramme sind in beiden Dimensionen logarithmisch mit den Einheiten $n = \log_2(|\vec{l}|)$ auf der x-Achse und $\log_2(t/\mathrm{ms})$ auf der y-Achse. Durch diese Darstellung lassen sich nicht nur größenordnungsmäßig verschiedene Laufzeiten in einem gemeinsamen Diagramm abbilden, sondern auch die Komplexitätsklasse einer Lösung ist gut ablesbar: Dominiert ein $\mathcal{O}(n)$-Anteil, zeigt sich der Graph asymptotisch zu einer Geraden der Steigung 1, dominiert dagegen ein $\mathcal{O}(n^2)$-Anteil, nähert sich der Graph asymptotisch der Steigung 2. Entsprechende Asymptoten sind zur Orientierung eingezeichnet.

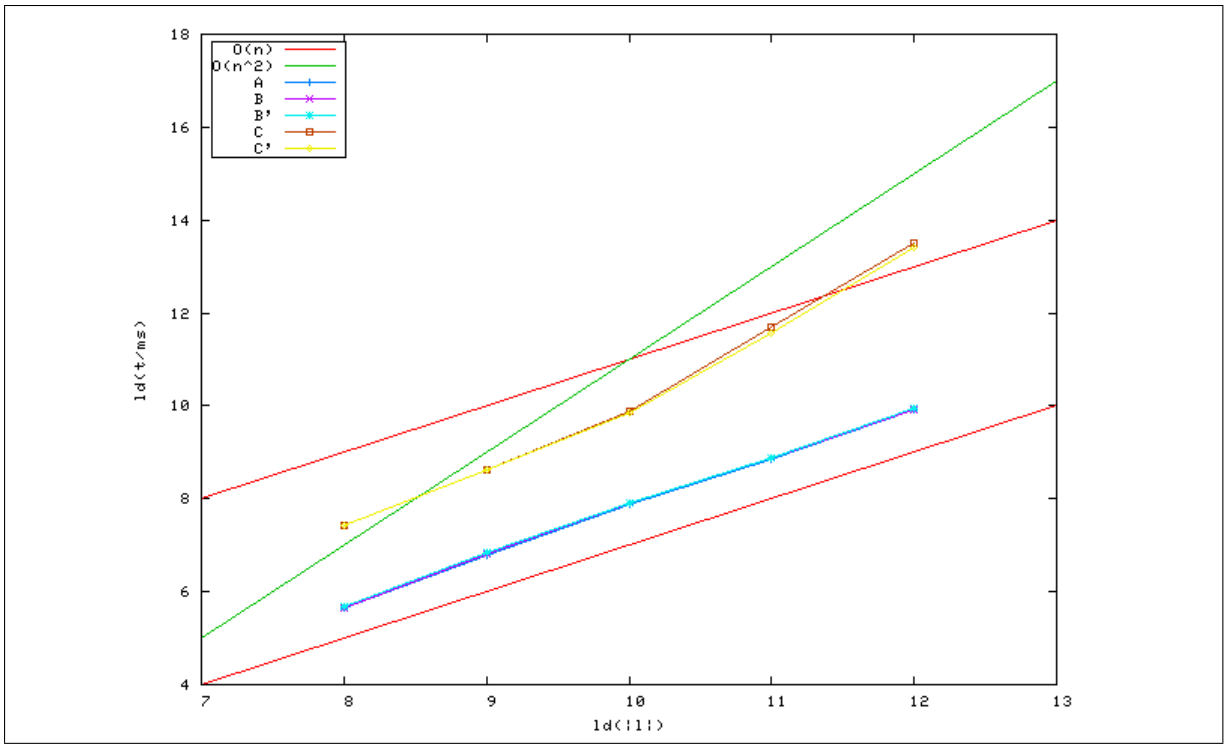


Abbildung 8.4: Interpreter

8.5.4.1 Interpreter

Abbildung 8.4 zeigt die Messergebnisse für alle fünf Varianten des Testprogramms, ausgeführt auf dem interaktiven Interpreter. Für die nichtinteraktive Basisversion sind im Vergleich vernachlässigbar geringere Laufzeiten zu erwarten. Die dargestellten Punkte stellen jeweils den Mittelwert von 10 unabhängigen Messungen eines einzelnen Laufes dar. Gemessen wurde mit den von Java standardmäßig zur Verfügung gestellten Mitteln (`System.currentTimeMillis`) die Systemzeit mit einer Genauigkeit von 1ms.

Da der Interpreter als Simulations- und Referenzwerkzeug gedacht ist, sind die absoluten Laufzeiten von geringem Interesse. Die wesentlichen qualitativen Vermutungen werden aber von den Daten bestätigt:

1. Bei der naiven Zyklenerkennung (C, C') dominiert ab einer gewissen, realistischen Eingabegröße der quadratische Aufwand der Suche auf dem Stack. Der Übergang von eher linearem zu eher quadratischem Verhalten ist im Bild deutlich bei $n = 10$ erkennbar.

2. Ohne Zyklenerkennung (A) ergibt sich nicht nur gleichmäßig linearer Aufwand, sondern auch ein proportionaler Geschwindigkeitsvorteil, der auf die Möglichkeit optimierter *tail*-Rekursion zurückzuführen ist.

3. Homomorphe Zyklenerkennung mit sehr dünner Verteilung markierter Kanten (B, B') bringt demgegenüber keinen größenordnungsmäßig relevanten Mehraufwand mit sich. Im Diagramm sind die Kurven optisch nicht zu unterscheiden. Daraus schließen wir, daß homomorphe Funktionen bedenkenlos auch für zyklenfreie Daten eingesetzt werden können. Zumindest der homomorphe Spezialfall der primitiven Corekursion stellt damit eine attraktive Alternative zur primitiven Rekursion dar, da er einerseits für alle zyklischen Daten effektiv ist, andererseits aber für zyklenarme Daten die Effizienz klassischer Verfahren in guter Näherung erreicht.

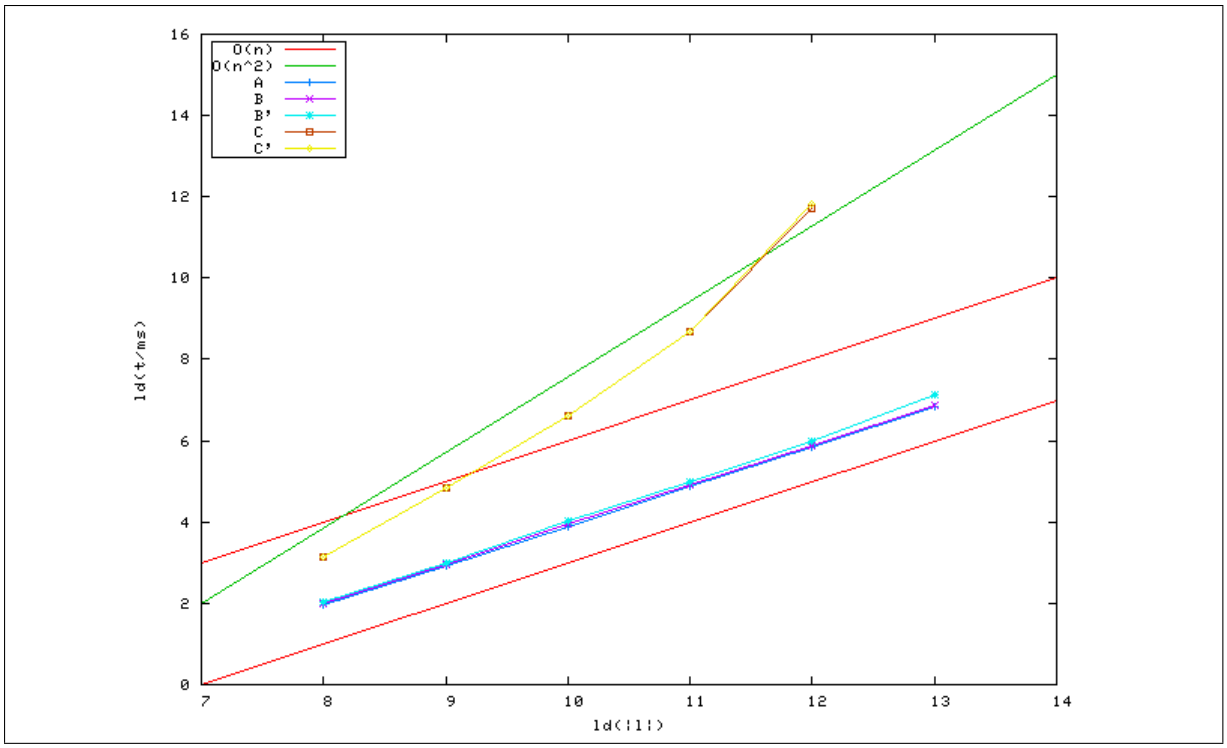


Abbildung 8.5: JIT-Compiler

8.5.4.2 JIT-Compiler

Abbildung 8.5 zeigt die Messergebnisse für alle fünf Varianten des Testprogramms, ausgeführt in der interaktiven Laufzeitumgebung des JIT-Compilers. Wie beim Interpreter sind für die nichtinteraktive Basisversion im Vergleich vernachlässigbar geringere Laufzeiten zu erwarten. Die dargestellten Punkte stellen jeweils den Mittelwert von 10 unabhängigen Messungen dar. Gemessen wurde mit den von Java standardmäßig zur Verfügung gestellten Mitteln die Systemzeit mit einer Genauigkeit von 1ms. Zur Erhöhung der Genauigkeit wurden pro Messung 10 Läufe in einer Schleife abgearbeitet, und die Gesamtzeit durch 10 geteilt.

Im Wesentlichen wiederholen sich die qualitativen Beobachtungen, die bereits beim Interpreter gemacht wurden. Da besonders der lineare Anteil der Berechnung beschleunigt wurde, liegen alle Meßwerte der naiven Verfahren (C, C') im quadratisch dominierten Bereich. Jenseits von $n = 11$ explodiert deren Laufzeit, was vermutlich auf Skalierungsprobleme der Java-Speicherverwaltung zurückzuführen ist.

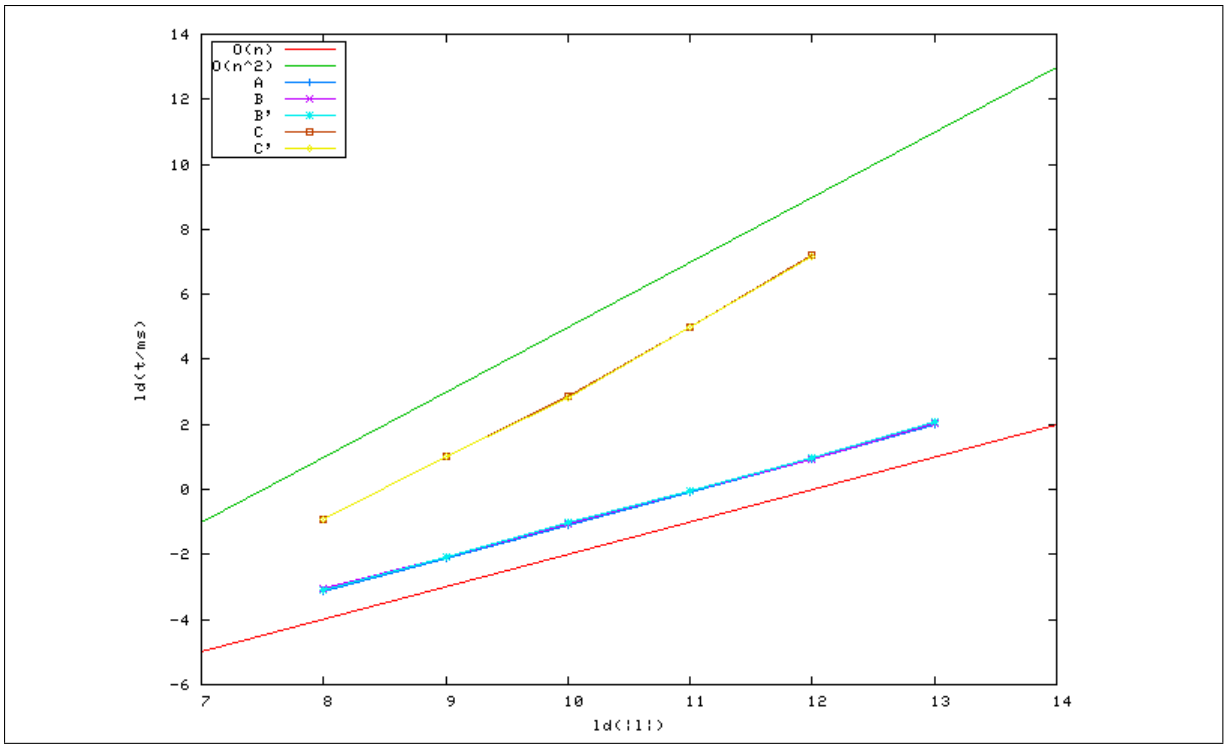


Abbildung 8.6: C-Compiler

8.5.4.3 C-Compiler

Abbildung 8.6 zeigt die Messergebnisse für alle fünf Varianten des Testprogramms, ausgeführt als nativer Maschinencode, erzeugt aus der Ausgabe des C-Compilers. Die dargestellten Punkte stellen jeweils den Mittelwert von 10 unabhängigen Messungen eines einzelnen Laufes dar. Gemessen wurde mit den von POSIX vorgesehen Mitteln (`times`) die Prozessorzeit mit einer Genauigkeit von 1μs.

Im Wesentlichen gelten die bereits gemachten qualitativen Beobachtungen. Die absoluten Laufzeiten sind allerdings nicht direkt auf realistische Anwendungen übertragbar, da das Laufzeitsystem nicht über automatische Speicherverwaltung verfügt.

Abbildung 8.7 zeigt noch einmal alle drei Ausführungsumgebungen im direkten Vergleich. Auf unterschiedliche Darstellung der drei Meßreihen wurde verzichtet, da die Trennung optisch eindeutig ist.

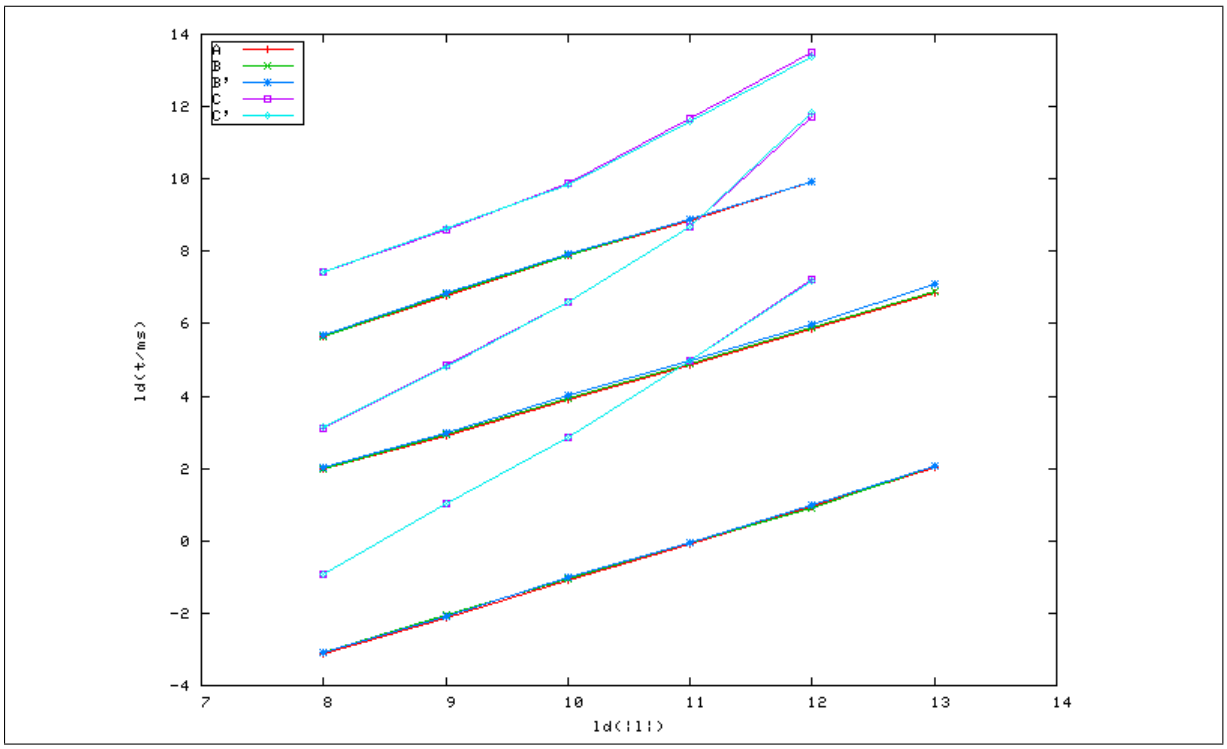


Abbildung 8.7: Übersicht

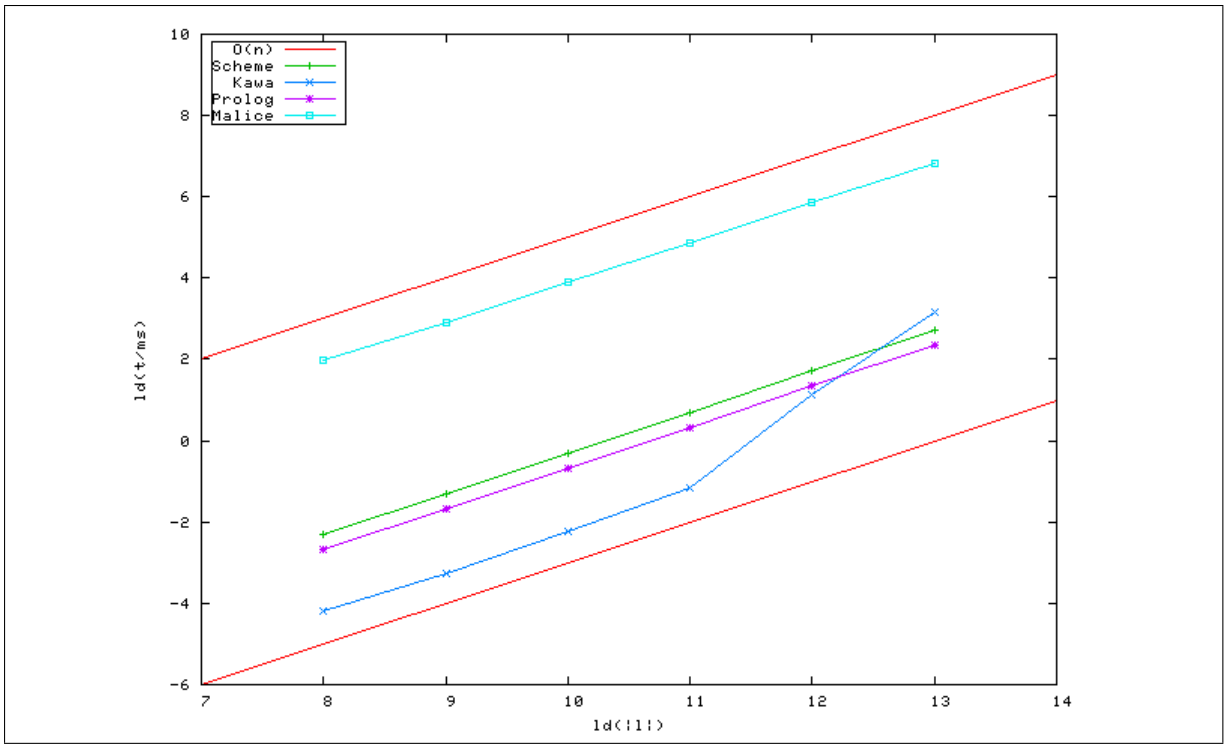


Abbildung 8.8: Plattformen mit virtuellem Maschinencode

8.5.4.4 Vergleich: Plattformen mit virtuellem Maschinencode

Abbildung 8.8 zeigt die Messergebnisse für alle getesteten Systeme, die keinen nativen Maschinencode, sondern Zwischencode einer virtuellen Maschine ausführen.

Der sich aus dem Vergleich des IMPL-JIT-Compilers und anderen virtuellen Maschinen ergebende Verlangsamungsfaktor ist auf den ersten Blick enttäuschend. Dabei ist allerdings zu bedenken, daß alle Vergleichsmaschinen ihrerseits in nativem Code implementiert sind, während die Abarbeitung des IMPL-JIT-Codes in einer Laufzeitschicht oberhalb der Java-Plattform erfolgt. Bei der Emulation typisch funktionaler Berechnungstechniken in Java haben andere Forscher[57, 64] Verlangsamungen um Faktor 3–30 beziehungsweise 10–40 festgestellt. In diesem Licht können die hier vorliegenden Ergebnisse mit einem Faktor von 15–20 als durchaus befriedigend gelten.

Im Gegensatz dazu erzeugt die Kawa-Implementierung von Scheme direkt Code für die Java-Maschine. Damit kann der überwiegende Anteil eines Scheme-Programms ohne Emulationsschicht ausgeführt und auch vom Java-JIT-Compiler in nativen Code übersetzt werden. Der Geschwindigkeitsvorteil ist für kleine n sehr groß, so daß sich diese Form der Codegenerierung möglicherweise auch als alternatives Ziel für unsere Zwischensprache IMPL anbieten würde. Allerdings zeigen sich bei steigendem n dieselben Skalierungsprobleme, wie wir sie schon beim IMPL-JIT-Compiler in den Programmversionen C und C' beobachtet haben, und die vermutlich mit der automatischen Speicherverwaltung zusammenhängen.

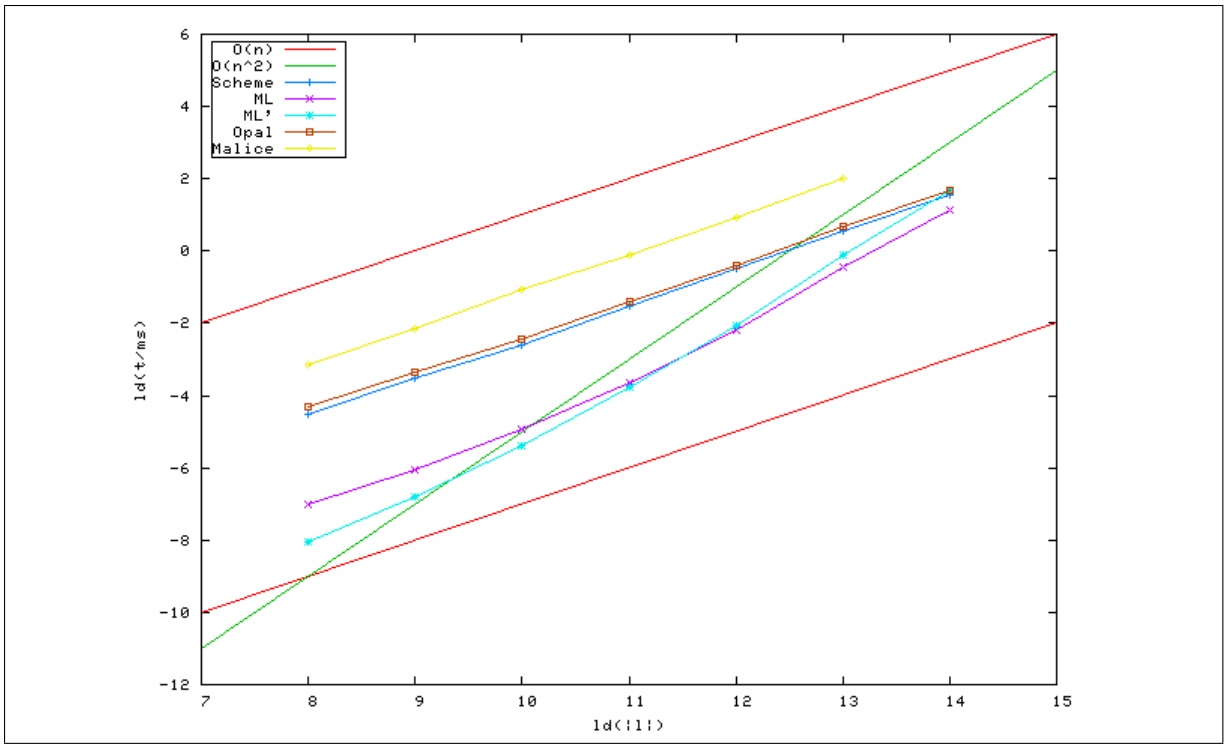


Abbildung 8.9: Plattformen mit nativem Maschinencode

8.5.4.5 Vergleich: Plattformen mit nativem Maschinencode

Abbildung 8.9 zeigt die Messergebnisse für alle getesteten Systeme, die nativen Maschinencode erzeugen, entweder direkt oder auf dem Umweg über C.

Dabei zeigt sich der IMPL-Compiler hinreichend nah an der Performanz von Scheme und Opal. Dabei sind allerdings zwei verzerrende Effekte zu beachten:

1. Einerseits verfügt das Laufzeitsystem des IMPL-C-Compilers noch nicht über eine vollwertige Speicherverwaltung. Die gemessenen Laufzeiten beinhalten daher zwar die Allokation (per `malloc`), nicht aber die automatische Deallokation von Zellen.

2. Andererseits ist der Codegenerator unseres Compilers ein minimaler Prototyp von etwa 2000 Zeilen. Eine weitere Optmierung findet bei der Übersetzung nicht statt, stattdessen wird das Verhalten des IMPL-Codes in Bezug auf lokale Variablen exakt und explizit nachgebildet. Damit ist zwar die Neutralität des erzeugten Codes gegenüber Speicherverwaltung und Instrumentierung gewährleistet, aber auch die Optimierungen des C-Compilers werden größtenteils verhindert.

Deutlich anders als die bisher betrachteten Kandidaten verhalten sich die ML-Programme. Für kleine n konkurrenzlos effizient, zeigen beide ein massives Skalierungsproblem im Form eines quadratischen Anteils, der ab $n = 11$ zu dominieren beginnt. Dieser Effekt betrifft die selbst definierte *map*-Funktion besonders stark, bei der vordefinierten ist er um den Preis eines konstanten Faktors etwas gedäpmft. Die Ursache für dieses Verhalten wurde nicht weiter untersucht.

Kapitel 9

Zusammenfassung und Ausblick

9.1 Zusammenfassung

Diese Arbeit hat sich mit dem Problem zyklischer Daten, ihrer Berechnung und Traversierung auf allen Ebenen von der semantischen Modellierung bis hin zur Implementierung eines Programmiersystems beschäftigt. Dabei ist es gelungen, das Phänomen der Zyklen von dem der divergent unendlichen Daten zu trennen, mit dem es in deklarativen Sichtweisen immer verknüpft war. Dadurch wird es möglich, Berechnungen auf zyklischen Daten strikt auszuwerten und somit die Daten von den Berechnungen streng zu trennen. Damit rücken deklarative und imperative Programmierung im Bereich der zyklischen Daten dichter zusammen, als dies mit der *lazy*-Auswertung möglich wäre, die gemeinhin mit funktionalen zyklischen Daten assoziiert wird. Darüberhinaus sind auf endlich zyklischen Daten globale Traversierungen möglich und damit Prädikate entscheidbar, an denen *lazy* Sprachen wohlbekanntermaßen scheitern.

Inwieweit diese Arbeit die am Anfang gesteckten Ziele erreicht hat, soll die folgende Rekapitulation von Abschnitt 1.3.2 klären:

1. *Deutung und begriffliche Erweiterung der kategoriellen universellen Coalgebra als Theorie der zyklischen Daten.* Dazu wurde in Kapitel 2 die Bedeutung der Wahl der zugrundeliegenden Kategorie untersucht. Es hat sich gezeigt, daß in der Kategorie **Set** der Mengen und Abbildungen, die die reinen Daten adäquat modelliert, initiale Algebren und finale Coalgebren unterschiedlich reicher Struktur existieren, die gemeinhin als die Modelle endlicher beziehungsweise unendlicher Daten bezeichnet werden. Mit Hilfe des kausalen Erreichbarkeitsbegriffs konnte diese Unterscheidung formalisiert, und ein dazwischenliegendes rationales Modell der endlich zyklischen Daten ausgezeichnet werden.

   Mit konstruktiven Begriffen wie der Individualtransformation und der Bisimulationsoperation wurde der Grundstein für die formale Semantik der späteren Algorithmen gelegt. Mit der Untersuchung von Cotermen und Pattern Matching wurde die syntaktische Deutung, die für initiale Algebren der dominierende Aspekt ist, auf rationale Coalgebren fortgesetzt. Dabei zeigte sich insbesondere, daß Pattern Matching eine genuin coalgebraische Operation darstellt, die in der initialen Algebra nur gelingt, weil diese Teil der finalen Coalgebra ist.

2. *Fortsetzung des coalgebraischen Paradigmas von der rein semantischen Ebene von Daten auf die tatsächliche Repräsentation und die Dynamik von Daten im Speicher einer abstrakten oder konkreten Maschine.* Dazu wurde in Kapitel 3 das Grundprinzip der Assoziation von Adressen und Inhalten im Speicher als Coalgebra gedeutet, was eine neue, sehr geschlossene Sichtweise auf den Begriff der *Pointer Algebra* ermöglicht. Diese coalgebraische Sichtweise definiert unter anderem eine kanonische finale Semantik, die dem Konzept der cofreien Datentypen entspricht. Die zusätzliche Verfeinerung der indirekten Adressierung gestattet die formale Beschreibung von partieller und in-

krementeller Initialisierung von Zellen, wie sie für die strikte Berechnung corekursiver Funktionen notwendig ist, innerhalb dieser Sichtweise.

Auf Basis der coalgebraischen Sicht wurden die üblichen Grundtransitionen zwischen Speicherzuständen formalisiert, und dabei auch der recht überladene Begriff der referentiellen Transparenz präzisiert.

3. *Identifikation der Bedingungen für die Möglichkeit zyklischer Berechnung.* Es wurde gezeigt, daß sich klassische Techniken zur strikten Berechnung direkt auf zyklische Strukturen generalisieren lassen, sofern der Anwendungsbereich höchstens rational ist, und ein Aufrufstack mit den abzuarbeitenden Inkarnationen zur Zyklenerkennung zur Verfügung steht. In Kapitel 4 wurde zusätzlich auf die Zyklenbehandlung bei der Berechnung primitiv corekursiver Funktionen eingegeangen, die im Vergleich zum algebraischen Vorgehen von unten nach oben auf dem Kopf zu stehen scheint, und die nur unter inkrementeller Initialisierung von Zellen realisiert werden kann. In Kapitel 5 wurde auf die inhärente Mehrdeutigkeit zyklischer Prädikate eingegangen und eine kompositionale Methode zur Auszeichnung intendierter Modelle durch Erwartungen vorgestellt, die sich als Spezialfall der Zyklenbehandlung erwiesen hat.

4. *Methodische Untersuchung der Voraussetzungen für zyklische Berechnung.* Es wurden unter anderem drei Prinzipien, die zum wohlbekannten Handwerkszeug der Informatik gehören, im Rahmen einer geschlossenen Theorie neu interpretiert, um so strikte zyklische Berechnungen zu ermöglichen.

   Das Prinzip der Zyklenerkennung durch Markierung besuchter Objekte ist bereits in den frühesten Graphalgorithmen erkennbar. Die Tatsache, daß es sich dabei für rekursive Algorithmen nur um eine Verdopplung des Aufrufstacks handelt, wurde entweder nicht beobachtet oder nicht weiter verfolgt.

   Das Prinzip der Zyklenbehandlung beim Entscheiden des kleinsten oder größten Fixpunktes ist ebenfalls als Abbruchkriterium von Suchverfahren auf Graphen wohlbekannt. Wir haben über den Stand der Kunst hinaus eine Formalisierung vorgenommen, die die Korrektheit des technischen Tricks in allgemeinerer Form nachweist. In dieser Darstellungsform ist es neben den beiden Extremen gestattet, Fixpunkte in den Bestandteilen eines komplexen Kalküls jeweils frei zu wählen.

   Das Prinzip, neue Zellen ohne Inhalt zu erzeugen und deren Initialisierung per Referenz an andere Ablaufstellen zu delegieren, findet sich gleich in mehreren Gebieten der Sprachimplementierung als altbekanntes Verfahren. Die Realisierung von Konstruktoren in objektorientierten Sprachen geschieht meist auf diese Weise. Auch die Optimierung einer linear rekursiven Funktion zu *tail*-Rekursion modulo Konstruktor basiert auf der kausalen Entkopplung des Konstruktors von seinen Argumenten, so daß die Berechnung des rekursiven Arguments in Endposition gebracht werden kann. Für das Dilemma zwischen Elimination von *tail*-Rekursion und Zyklenerkennung wurde in Kapitel 6 eine effektive und im azyklischen Fall optimale dynamische Lösung vorgestellt.

5. *Abgrenzung der Dualität Algebra-Coalgebra in mengentheoretisch fundierten Ansätzen von der Symmetrie nichtfundierter Ansätze.* Die theoretischen Untersuchungen in Kapitel 4 und 5 haben gezeigt, daß in einem fundierten Ansatz mit strikter Auswertung der Bereich der effektiv lösbaren Probleme gegenüber den nichtstrikten *lazy*-Sprachen verschoben ist. Einerseits wird auf die Berechnung (endlicher Präfizes) echt unendlicher Daten verzichtet. Andererseits erschließt sich damit die Möglichkeit, Prädikate mit Fixpunktsemantik auf dem gesamten Datenraum zu entscheiden. Viele Algorithmen lassen sich nur mit dieser Kombination corekursiver Funktionen und zyklischer Prädikate vom endlichen auf den zyklischen Fall generalisieren, wie in Teil IV gezeigt wird.

   Als technische Konsequenz der Dualität von Algebra und Coalgebra hat sich gezeigt, daß dem atomaren Konstruktor als Grundoperation algebraischer Funktionsberechnung die inkrementell initialisierende Zuweisung als coalgebraisch duale Operation gegenübersteht. Damit nähert sich die operationale Semantik eines coalgebraischen Datenmodells eher dem imperativen als dem klassisch

algebraisch funktionalen Paradigma an. Dem wurde im Entwurf der abstrakten Maschine Rechnung getragen, die sich stärker an die Prinzipien objektorientierter Maschinen als an das in der funktionalen Welt dominierende Graphreduktionsverfahren anlehnt. Implementierungen zyklischer Algorithmen auf dieser Maschine lassen sich daher leicht in einen imperativen Kontext übertragen, sofern dort Mittel zur Zyklenerkennung zur Verfügung stehen.

6. *Semantische Integration von Theorie und Praxis der zyklischen Berechnung.* Die coalgebraische Sichtweise ermöglicht die gleichzeitige Modellierung konkreter Repräsentationen von Daten in nichtfinalen Coalgebren und deren kanonischer Semantik in der zugehörigen finalen Coalgebra. Die Abstraktionsbeziehung zwischen beiden ist durch den universellen Morphismus kanonisch festgelegt und entspricht der Quotientenbildung modulo Bisimilarität. In Kapitel 4 wurde diese Dualität genutzt, um die Freiheitsgrade in der Spezifikation der primitiv corekursiven Funktionsberechnung formal zu erfassen. Die simultane Modellierung von Semantik und Implementierung ist auch von großer praktischer Relevanz für die Zyklenerkennung, da im Allgemeinen nur die Zyklen der konkreten Ebene mit vertretbarem Aufwand erkannt werden können. Wie in Kapitel 6 erläutert, ist dieses Kriterium in den meisten Fällen auch ausreichend.

   Den mathematischen Spezifikationen der theoretischen universellen Verfahren und der abstrakten Maschine zu ihrer instanzweisen Realisierung sind ausführbare Haskell-Programme zur Seite gestellt worden, die der Simulation von Vorgängen und prinzipiell auch der mechanischen Verifikation von Eigenschaften dienen können. Auf die Ausführung des letzteren Vorhabens innerhalb dieser Arbeit wurde nicht zuletzt wegen der komplexen Semantik von Haskell verzichtet. Stattdessen sei als anschaulicher Beleg für die Konsistenz von Theorie und Praxis auf die implementierten Beispielalgorithmen verwiesen.

7. *Nachweis der Praktikabilität der operationalen Semantik.* In Kapitel 7 wurde der Entwurf einer abstrakten Maschine beschrieben und aus den theoretischen Anforderungen begründet. Diese gestattet es, die Operationalisierung der theoretischen Verfahren soweit zu konkretisieren, daß spezifische Anwendungen der universellen Verfahren implementiert, und über die Effizienz ihrer Ausführung argumentiert werden kann. Die lauffähige Implementierung dieser Maschine umfaßt alle wichtigen Komponenten einer Programmierumgebung, insbesondere Parser, Interpreter und Compiler, sowie ein graphische Oberfläche, mit deren Hilfe Transformation und Ablauf von Programmen im interaktiven Experiment nachvollzogen werden können.

   Diese Maschine bietet nicht nur eingebaute Unterstützung für die Erkennung und Behandlung von Zyklen, sondern auch eine weitgehende Umsetzung der in Kapitel 6 vorgestellten Optimierungen, die den Aufwand für zyklische Daten stark einschränken und für nichtzyklische Daten beinahe auf das Maß einer klassischen algebraischen Ausführungstechnik reduzieren.

8. *Nachweis der praktischen Relevanz des ganzen Ansatzes.* In Anhang A wird der Archetyp der zyklischen Datenstrukturen, nämlich die Stellenwertdarstellung rationaler Zahlen behandelt. Dabei steht das Typische der zyklischen Probleme im Vordergrund, sowie der Nachweis der Tatsache, daß die rationale Semantik die natürliche Modellierung von konkreten Problemen darstellt, die traditionell mit expliziten Zeigerstrukturen beschrieben werden.

   Anhang B befaßt sich dagegen mit dem Verhältnis von initialer und rationaler Semantik von Datentypen, insbesondere mit dem möglichst nahtlosen Übergang von Spezialfall der zylenfreien zum allgemeinen Fall der beliebig zyklischen Daten. Dazu werden Standardalgorithmen der funktionalen Listenprogrammierung in eine Form gebracht, die auch für zyklische Listen effektiv ist, und der durch die Verallgemeinerung gewonnene Mehrwert diskutiert.

   Anhang C schließlich beschreibt das bekannte Problem der durch Signaturen induzierten Subtypbeziehung mit Hilfe einer neuartigen, hochgradig zyklischen Datenstruktur. Diese ähnelt von der Idee den *vtables* der Objektorientierung, kann aber mit Hilfe zyklischer Berechnungen auch ad-hoc und insbesondere zur Laufzeit verarbeitet werden.

9.2 Verwandte Arbeiten

9.2.1 Relationale Modellierung von Strukturen im Speicher

Als besonders weitreichende und charakteristische Arbeit zur Modellierung von Referenzstrukturen im Speicher ist die von MÖLLER zu nennen. Auch wenn der Autor den Begriff *Pointer Algebra* geprägt hat, so handelt es sich in allen wesentlichen Punkten um eine der coalgebraischen äquivalente Sichtweise. Man könnte den Ansatz als implizite Coalgebra bezeichnen, wie anhand von [41] belegt werden kann:

1. Zur Modellierung eines Speicherzustands wird eine Abbildung von Adressen auf Inhalte verwendet. Diese entspricht der Operation einer Speichercoalgebra.

2. Semantische Schlüsse werden anhand von Erreichbarkeitsmengen gezogen. Diese entsprechen Untercoalgebren und minimalen Cotermen.

3. Als Semantik einer Referenzstruktur wird eine Abstraktionsfunktion angegeben. Der Wertebereich dieser Abstraktionsfunktion wird zwar nicht generisch spezifiziert, der Autor räumt aber ein, daß im Falle zyklischer Listenstrukturen und nichtstrikter Semantik bei naiver Interpretation unendliche Listen als Bilder entstehen. Die finale Coalgebra stellt also einen geeigneten Wertebereich dar.

4. Die Anforderungen an Abstraktionsfunktionen entsprechen der coalgebraischen Homomorphismuseigenschaft. Zusammen mit dem vorigen Punkt ergibt sich, daß Anamorphismen die kanonischen Abstraktionsfunktionen sind.

5. Für die gesonderte Behandlung zyklischer Listen und ihre Unterscheidung von der unendlichen Variante gibt MÖLLER eine verfeinerte Abstraktionsfunktion an. Diese leistet Zyklenerkennung auf die gleiche Weise wie die in Teil II entwickelten Algorithmen. Die Abstraktion wird aber zur Spezifikation imperativer Verfahren verwendet, daher werden keine Schlüsse über die funktionalen Eigenschaften von Zyklenbehandlungsstrategien gezogen.

9.2.2 Corekursion und Berechenbarkeit

Im Zusammenhang mit unendlichen Datenstrukturen versagt der klassische Berechenbarkeitsbegriff, da die Extension einer unendlichen Struktur offensichtlich nicht in endlicher Zeit gebildet werden kann. Bei der nichtstrikten Auswertung solcher Strukturen ergibt sich der duale Begriff der *Produktivität*, welcher besagt, daß die direkten Nachfolger eines Datenelements in endlicher Zeit bestimmt werden können. In einer grundsätzlich nichtstrikten Sprache ist die Grenze zwischen Produktivität und Nichtproduktivität sehr unklar, analog zu dem unentscheidbaren Problem, terminierende von nichtterminierenden rekursiven Funktionen zu unterscheiden.

Etwas deutlicher wird das Wesen der Produktivität in speziellen Kontexten, wie zum Beispiel einer stark normalisierenden Sprache mit zusätzlichen unendlichen Strömen, wie sie in [60] beschrieben wird. Diese Arbeit illustriert den semantischen Umgang mit Corekursion besonders deutlich und hat die hier vorliegende Untersuchung der rationalen Coalgebra, in der produktive und terminierende Verfahren zusammenfallen, entscheidend inspiriert.

9.2.3 Coalgebraische Strukturen

9.2.3.1 Zahlentheorie

Der Unterschied zwischen initialen und finalen Daten in fundierter Mengentheorie tritt an Zahlenräumen besonders prägnant in Erscheinung. In [45] wird, ausgehend von der Behauptung, coalgebraische Zahlentheorie sei im Wesentlichen mit Grenzwerten gleichzusetzen, eine finale Theorie der reellen Zahlen entwickelt. Ebenfalls mit reellen Zahlen, allerdings aus der operationalen Sicht corekursiver Dezimalentwicklung, befaßt sich eine nicht ganz ernst gemeinte Arbeit von KARCZMARCZUK[25]. Aus dem Vergleich

dieser Arbeiten hat sich der Ansatz zu Anhang A ergeben, und damit das treibende Beispiel für diese Arbeit insgesamt.

9.2.3.2 Bisimilarität

Die Konstruktion der Bisimulationsoperation υ entspricht der in einer Arbeit von LENISA[31], wo Bisimulation mit den Mitteln logischer Sequenzkalküle untersucht wird. Dort wird allerdings in umgekehrter Vorgehensweise zunächst die Intuition der Termkonstruktoren zur Rechtfertigung der Bisimulationsregeln herangezogen, und erst nachträglich auf die kategorielle Definition zurückgeführt. ABITEBOUL und VAN DEN BUSSCHE setzen in [1] aus der Sicht objektorientierter Datenbanken ebenfalls denotationelle und operationale Zugänge zur Bisimilarität, die sie *tiefe Gleichheit* nennen, in Beziehung.

9.2.3.3 Coterme

Zur coalgebraischen Syntax existieren diverse Ansätze. Als besonders nützliche Quelle für die hier vorliegende Arbeit hat sich der von GHANI und anderen [15] erwiesen. Dort werden Monaden als kategorielle Darstellung von Termen und Cotermen verwendet. Dadurch wird die Unabhängigkeit von konkreten Trägerkategorien und Signaturfunktoren erreicht. Auch der Begriff der Rationalität wird in diesem allgemeineren Rahmen untersucht. Der monadische Ansatz dient der Modellierung von Termstrukturen mit frei adjungierter Variablenmenge X, dargestellt durch die Erweiterung der Signatur um $\_ + \mathcal{C}_X$, und der Substitution. Im Gegensatz dazu haben wir Grundcoterme einerseits und algebraische Pattern mit Variablen andererseits eingeführt. Diese Unterscheidung begründet sich in der Entwurfsentscheidung, Zyklenerkennung ausschließlich als impliziten Schritt bei der Funktionsinkarnation anzusehen. Damit entfällt die Notwendigkeit, explizit mit Termgleichungen umzugehen.

9.2.4 Fixpunktsemantik

9.2.4.1 Zyklisches Berechnen

Eine aufschlußreiche, eng verwandte Arbeit auf dem Gebiet der Fixpunktsemantik von Funktionen ist in [44] beschrieben. Dort wird der von MANNA und SHAMIR in [34] eingeführte optimale Fixpunkt als Bedeutung einer rekursiven Funktion propagiert. Wärend der kleinste Fixpunkt an allen Stellen undefiniert ist, für die die Definition dies gestattet, ist der optimale Fixpunkt an allen Stellen definiert, für die die Definition genau ein definiertes Ergebnis liefert.

In [44] wird nun das intuitionistische Beweisen für die Existenz des optimalen Fixpunkts, also das punktweise Beweisen der Eindeutigkeit definierter Funktionsergebnisse, per CURRY-HOWARD-Isomorphie als Programmiersprache interpretiert. Die Autoren argumentieren ganz in unserem Sinne, daß der optimale Fixpunkt eine Erweiterung des kleinsten Fixpunktes, und damit eine effektivere Nutzung der Information in einer rekursiven Definition darstellt.

9.2.4.2 Zyklisches Schließen

Die Rehabilitation logischer Zyklen ist wesentlich den Arbeiten von NEBEL zu verdanken. In [42] wird ihre Semantik und ihre pragmatische Bedeutung für die Wissensrepräsentation umfassend untersucht. NEBEL untersucht auch die Mehrdeutigkeit der Fixpunktsemantik zyklischer Definitionen und erwähnt den kleinsten und größten Fixpunkt als kanonische, aber nicht einzige sinnvolle Lösungen. Der von ihm vorgeschlagene dritte Weg der sogenannten *deskriptiven* Semantik wird dadurch motiviert, daß unter dem größten Modell Namen Schall und Rauch sind, und alle strukturell äquivalenten (bisimilaren) Objekte zusammenfallen. Ob dieser Effekt gewünscht ist, hängt vom Gebrauch von Namen in der jeweiligen Anwendung ab. Das Problem kann also als eine Konfusion nichtfinaler und finaler Coalgebra angesehen werden. Weitere, gemischte Fixpunktlösungen wie in Kapitel 5 werden von NEBEL nicht betrachtet.

9.2.5 Coalgebraische Programmierung

9.2.5.1 Charity

Die fundamentale Arbeit auf dem Gebiet der corekursiven Programmierung ist die experimentelle Sprache Charity[10]. Primitive Rekursion und Corekursion in ihrer kategoriellen Darstellung sind die zentralen Mittel dieser Sprache, die durch eine Reihe schöner theoretischer Eigenschaften besticht. So sind die beiden dualen Konzepte in der Sprache völlig symmetrisch abgebildet, so daß die wohlunterschiedene, aber integrierte Arbeit mit initialen und finalen Datentypen möglich wird. Außerdem sind Charity-Programme stark normalisierend hinsichtlich der operationellen Semantik der Reduktion kategorieller Kombinatoren. Dadurch ergibt sich eine neue Sichtweise auf die Dialektik strikter und nichtstrikter Sprachen: Obwohl Charity in seinen kategoriellen Grundoperationen strikt ausgewertet wird, lassen sich damit unendliche Strukturen beschreiben, da die Reduktionsregeln für die als primitiv angesehenen Kata- und Anamorphismen implizit einer *lazy* Auswertung entsprechen.

Trotz aller theoretischen Eleganz hat Charity keine weite Verbreitung gefunden. Der plausibelste Grund dafür ist der für den Programmierer äußerst gewöhnungsbedürftige Aufbau von Programmen aus Kata- und Anamorphismen. Zahlreiche wohlbekannte Probleme sind zwar semantisch gesehen primitv (co)rekursiv, bedürfen aber einer nichttrivialen Umcodierung, um der syntaktischen Normalform der Morphismen zu genügen. Als minimales Beispiel mag die Fakultätsfunktion dienen, die unter anderem als Komposition eines Ana- und eines Katamorphismus dargestellt werden kann. Zahlreiche neue Klassen von Morphismen, darunter *Para-* und *Hylomorphismen*, wurden erfunden, um solche Funktionen zu taxieren.

Die hier vorliegende Arbeit versteht sich als diametraler Gegenentwurf zu Charity in dem Sinne, daß die kategorielle Darstellung zwar auf der semantischen Ebene als Metatheorie herangezogen wird. Auf der Implementierungsebene dagegen lehnt sich die entworfene virtuelle Maschine maximal an gewohnte Begriffe der imperativen Programmierung an, die schließlich das vorherrschende Paradigma im Umgang mit zyklischen Daten ist. Somit soll diese Arbeit auch dazu beitragen, Corekursion auch im imperativen Kontext praktikabel zu machen. Aus dieser Opposition zu Charity ergibt sich der Name Malice für das implementierte System.

9.2.5.2 Totalisierung partiell rekursiver Funktionen

In [19] wird eine andere Anwendung für die Einbettung der initialen Algebra in die finale Coalgebra beschrieben: Durch Erweiterung des Signaturfunktors um sogenannte unbeobachtbare Schritte und anschließende Abstraktion von denselben wird die Totalisierung partiell rekursiver Funktionen in totaler Coalgebra modelliert. Der Berechenbarkeitsbegriff bleibt dabei aber strikt algebraisch: Unendliche Elemente werden als undefinierte Ergebnisse betrachtet, und per Bisimulation zur Äquivalenzklasse des $\perp$-Elements zusammengefaßt.

9.2.6 Deklarative Verfahren auf Graphen

Die Tatsache, daß zyklische Daten meist mit Hilfe der Graphmetapher beschrieben werden, leg die Vermutung nahe, daß zyklische Algorithmen im Wesentlichen Graphalgorithmen sind. Folglich besteht eine enge Beziehung zwischen der deklarativen Lösung zyklischer Probleme und deklarativen Techniken zur Graphprogrammierung. In diesem Zusammenhang sind zwei Ergebnisse von besonderer Bedeutung.

Die Arbeiten von Launchbury [29] und King [27] beschäftigen sich damit, die für eine Graphtraversierung notwendige Zustandsinformation in einen Monadentyp zu kapseln. Dadurch wird eine Trennung der Ebenen von Traversierung und lokaler Berechnung erreicht, um den Preis der bekannten rigiden Sequentialisierung durch monadische Operationen, die eigentlich dem deklarativen Geist zuwiderläuft. Erwig verfolgt in [12] einen anderen Ansatz, nämlich die induktive Dekomposition eines Graphen in besuchten und zu besuchenden Anteil. Dadurch ergibt sich ein sehr pur funktionaler Stil, der dem didaktischen Anspruch des Ansatzes gerecht wird.

Beiden Ansätzen gemeinsam ist, daß sie die Kompositionalität funktionaler Programmierung höherer Ordnung ausnutzen, um von den technischen Details der Markierung und Traversierung für den Benutzer zu abstrahieren. Dies entspricht in etwa dem Übergang zu Stufe $2a$ der Algorithmen in Kapitel 4 und 5, bei dem die in Interpretationen und Kalkülen gekapselte lokale Funktionalität mit dem Traversierungszustand in Form der Liste gerade zu bearbeitender Knoten kombiniert wird.

Die coalgebraische Sichtweise geht jedoch über die graphische hinaus, insofern als jeder konkreten Graphstruktur in Form einer nichtfinalen Coalgebra eine kanonische finale Semantik zugeordnet wird, die von den Knotenidentitäten abstrahiert, die bei den meisten Graphproblemen ohnehin keine inhärente Bedeutung besitzen. Dadurch werden unscharfe Beschreibungen modulo Bisimilarität möglich, die beispielsweise die in Kapitel 6 beschriebenen Optimierungen als Äquivalenzumformungen erscheinen lassen. Eine duale Unschärfe ist von der algebraischen Sichtweise auf Daten bekannt, nämlich die Dialektik von gemeinsamen Unterbäumen (*sharing*) und Linearität.

9.2.7 Zyklenerkennung

9.2.7.1 Stackanalyse

Die Theorie der Stackanalyse zur Laufzeit ist in [14, 65] umfassend beschrieben. Die Autoren beschäftigen sich allerdings aus völlig anderen Gründen mit dem Stack als die hier vorliegende Arbeit: Flexible Sicherheitskonzepte von offenen und mobilen Plattformen wie Java oder .NET benötigen Kontextinformation über den laufenden Code, die durch Analyse des Aufrufstacks gewonnen werden soll. Im Prinzip dient die Analyse also der *Verhinderung* von Abläufen. Daß sie auch umgekehrt dazu dienen kann, die Berechnungsleistung eines Systems zu erweitern, wird dort nicht betrachtet.

Allerdings bilden die formalen Ergebnisse von [14] auch eine solide Grundlage für die Untersuchung der Wechselwirkung von Zyklenerkennung und Codetransformationen, wie sie im hier implementierten Compiler verwendet werden. Der dort ebenfalls angesprochene Konflikt von Stackanalyse und effizienter *tail*-Rekursion stellt sich bei der Nutzung der Analyse zur Zyklenerkennung anders dar: In einem kooperativen System existiert hierfür eine elegante Lösung, siehe dazu Abschnitt 6.1.2.

9.2.7.2 Schwarze Löcher

Bei der Implementierung von nichtstrikten funktionalen Sprachen durch *lazy* Graphtransformation wird eine Technik namens *black-hole*[24] hauptsächlich zur Verbesserung des Speicherverhaltens eingesetzt. Die Grundidee ist, einen Redex, der gerade reduziert wird, bereits bei Beginn der Reduktion zu überschreiben, damit eventuell seine Argumente unerreichbar werden. In [46] wird nicht nur die Implementierung in einem Laufzeitsystem für Haskell beschrieben, sondern auch festgestellt, daß als Nebeneffekt eine automatische Zyklenerkennung abfällt. Mit Ausnahme des vorzeitigen Abbruchs eines ansonsten nichtterminierenden Programms wird daraus aber kein Nutzen gezogen. Es wird sogar argumentiert, daß ein übergelaufener Stack aus diagnostischer Sicht als Zyklenerkennung vorzuziehen sei.

Einige Autoren haben weiter an der Technik gearbeitet. So werden die *black-holes* in [53] auf zahlreiche Probleme hin untersucht, die im Zusammenhang mit Nebenläufigkeit stehen. Diese werden gelöst, eine Nutzung im Sinne einer Zyklenbehandlung ist aber auch hier nicht in Sicht. Der Grund ist eine inhärente Eigenschaft des Graphreduktionsansatzes: Getreu dem algebraischen Grundparadigma werden Teilausdrücke unabhängig von ihrem Kontext, insbesondere von der Speicherstelle, welche das Ergebnis aufnehmen soll, betrachtet. Damit besteht im Allgemeinen keine Möglichkeit, erkannte Zyklen zu schließen.

9.3 Zukünftige Arbeiten

9.3.1 Offene theoretische Fragen

Die folgenden Punkte betreffen den Erkenntnisstand über zyklische Berechnung im Allgemeinen.

9.3.1.1 Verhältnis der primitiven Funktionsklassen

Es ist bekannt, daß die Klasse der primitiv rekursiven Funtionen eine echte Teilklasse der primitiv corekursiven Funktionen darstellt, wobei die ACKERMANN-Funktion die bekannteste Beispielinstanz der Differenzklasse ist. Es ist außerdem bekannt, daß gewisse Erweiterungen des strengen Begriffs der Primitivität, etwa Katamorphismen höherer Ordnung, diesen Unterschied ausgleichen können.

Offensichtlich ist auch die Klasse der rational corekursiven Funktionen eine echte Teilklasse der primitiv corekursiven, und eine Oberklasse der primitiv rekursiven. Ob allerdings diese letzte Relation eine echte Teilklassenbeziehung ist, wurde in dieser Arbeit nicht untersucht. Um diese Frage positiv zu entscheiden, müßte ein Beispiel einer rational corekursiven Funktion gefunden werden, die nicht primitiv rekursiv ist. Um stattdessen die Äquivalenz zu zeigen, könnte beispielsweise eine primitiv rekursive Simulation beliebiger rational corekursiver Funktionen in algebraischer Graphcodierung angegeben werden. Die bisherigen Erkenntnisse lassen vermuten, daß die Äquivalenz gilt.

9.3.1.2 Alternative Techniken der Zyklenbehandlung

Die in dieser Arbeit vorgestellten zyklischen Algorithmen verhalten sich sehr spezifisch hinsichtlich Zyklenbehandlung: Entweder wird das Resultat einer umgebenden identischen Inkarnation kopiert, oder es wird ein Wahrheitswert eingesetzt. Prinzipiell sind natürlich auch andere Aktionen denkbar. Die implementierten Beispielprogramme definieren beispielsweise die totale Ordnung auf einem Datentyp über eine Funktion mit der dreiwertigen Relationenalgebra $\{<, =, >\}$ als Wertebereich, wobei im zyklischen Fall stets $=$ eingesetzt wird. Wie bei den Wahrheitswerten werden hier Idempotenzeigenschaften ausgenutzt. Der Nachweis der Korrektheit eines solchen Verfahrens kann auf verschiedene Weise erfolgen, unter anderem durch Rückführung auf eine Familie von Prädikaten mit Fixpunktsemantik, oder durch eigenständige Beweisführung im Stil von Kapitel 5.

Die Zyklenbehandlung kann allerdings auch dazu „mißbraucht“ werden, die konkrete Graphstruktur einer Datenrepräsentation im Speicher offenzulegen, also im Sinn von Kapitel 4 eine nicht-abstrakte Interpretation zu implementieren. Zwar müssen nicht alle Konstrukte, die außerhalb der intendieren Semantik stehen, verboten werden. Trotzdem ergibt sich das Problem, daß die in Kapitel 6 beschriebenen Optimierungen, die für den effizienten Einsatz zyklischer Verfahren unerläßlich sind, nur unter rein abstrakten Interpretationen semantisch invariant sind, da sie die Graphstruktur von Funktionsergebnissen bisimilar verändern können.

9.3.2 Offene linguistische Fragen

Die folgenden Punkte sind noch zu klären, um in den Entwurf einer annehmbaren deklarativen Programmiersprache für zyklische Programme einzufließen.[1]

9.3.2.1 Nicht-primitive Definitionen

Ebenso wie eine universelle algebraisch funktionale Programmiersprache auch Funktionsdefinitionen in nicht-primitiv rekursiver Normalform gestattet, sollte auch in einem dieser Arbeit entsprechenden System die nicht-primitiv corekursive Funktionsdefinition erlaubt sein. In Anhang B werden solche Beispiele vorgestellt, bei denen eine Interpretation selbst rekursiv definiert und damit gegebenenfalls partiell ist.

So wie eine primitiv rekursive Funktion per Konstruktion terminiert, ist eine primitiv corekursive Funktion per dualer Konstruktion produktiv. Im Fall der rationalen Corekursion terminiert sie dann auch

[1] Teile dieses Abschnitts können zum Zeitpunkt der endgültigen Veröffentlichung der Arbeit veraltet sein.

global. Trotzdem sind Termination und Produktivität nicht völlig äquivalente Eigenschaften: Während sich nichtterminierende Rekursion üblicherweise in unbeschränkten Wachstum des Stacks und damit in der Praxis in einem Überlauf äußert, hat nichtproduktive Corekursion den Effekt, daß zum Zeitpunkt der Zyklenerkennung kein Resultat der äußeren Inkarnation vorhanden ist, das kopiert werden könnte. Mit diesem Problem kann auf verschiedene Weisen umgegangen werden:

1. Das Problem kann schlichtweg ignoriert werden. Die `dito`-Operation kopiert dann pseudozufällige Inhalte uninitialisierter Resultatvariablen. Folglich äußern sich nichtproduktive Fälle in korrupten Ergebnissen.

2. Das Problem kann zur Laufzeit erkannt werden. Bei uninitialisierten Resultaten der äußeren Inkarnation wird die Zyklenbehandlung ausgesetzt. Dann verhalten sich nichtproduktive wie nichtterminierende Fälle. Alternativ kann sofort mit einem Fehlerereignis abgebrochen werden.

3. Das Problem kann eventuell statisch durch Umstellung der Auswertungsstrategie der Funktionsimplementierung eliminiert werden. Hierfür ist nach aktuellem Kenntnisstand manuelle Programmtransformation nötig.

9.3.2.2 Verschleierung von Zyklen

Wie in Kapitel 6 bereits erläutert, ist nicht jeder Zyklus der finalen Semantik einer Berechnung auch automatisch durch einen Zyklus der corekursiven Eingaben repräsentiert. Immer dann, wenn corekursiver Abstieg nicht entlang bereits im Anfangszustand vorhandener Referenzen erfolgt, sondern auch frische Daten traversiert werden, kann ein sogenannter Quasizyklus entstehen, bei dem eine unbeschränkte Schachtelung von Inkarnationen aufgebaut wird, die lediglich modulo Bisimilarität periodisch sind. Da aber nur die nicht-finale Zellenidentität ein Kriterium von allgemein akzeptabler Effizienz für die Zyklenerkennung liefert, werden solche Spiralen normalerweise nicht erkannt.

Der Grund für quasizyklisches Verhalten kann einem Algorithmus inhärent sein; in diesem Fall empfielt sich die lokale Verwendung der aufwendigeren Zyklenerkennung modulo Bisimilarität, in der Maschinensprache ausgedrückt durch das `quasi`-Attribut. Andererseits können auch elementare Programmtransformationen, die im algebraischen Kontext harmlos sind, einen Zyklus zu einem Quasizyklus entarten lassen. An erster Stelle steht dabei die sogenannte *Produktkonversion*. Darunter verstehen wir die Transformation einer mehrstelligen Funktion in eine einstellige Funktion, die ein Tupelargument erwartet. Beim mehreren Aufrufen dieser Funktion mit komponentenweise identischen Argumenten werden im Allgemeinen verschiedene, frische Tupel übergeben. Eine Programmiersprache für zyklische Programme muß daher notwendigerweise mehrstellige Funktionsabstraktion und -applikation unterstützen, wobei die Bildung eines Argumentvektors streng verschieden von den Konstruktoren benutzerdefinierter Produkttypen ist.

9.3.2.3 Deklarative Deutung von Differenzparametern

Die Einführung der Differenzparameter in der virtuellen Maschine überträgt das Privileg der partiellen Erzeugung von Daten in wohlspezifizierter Weise von primitiven Konstruktoroperationen auf benutzerdefinierte Prozeduren. Diese Verallgemeinerung ist essentiell notwendig für die Unifikation der coalgebraischen Auswertungsstrategie mit dem Abstraktionsanspruch des funktionalen Paradigmas. Operational gesehen, gestattet die Auszeichnung eines Eingabeparameters als Differenzparameter die Ausführung einer Prozedur und die Speicherung ihrer Ausgaben vor der Berechnung dieser Eingabe.

Wenngleich dieser Effekt für die Zyklenbehandlung von großer Bedeutung ist, handelt es sich doch nur um verschiedene Auswertungsreihenfolgen. In einer deklarativen Eingabesprache könnten Differenzparameter daher prinzipiell vor dem Benutzer verborgen und als reine Implementierungstechnik verstanden werden. Andererseits ergeben sich eventuell durch eine Offenlegung zusätzliche Ausdrucksmöglichkeiten für den Programmierer, sofern eine befriedigende Syntax für Ausdrücke mit Differenzen gefunden werden kann. Erfahrungen mit dem logischen Vorbild für die funktionalen Differenzparameter in der Sprache

Prolog haben gezeigt, daß die Verwendung von Differenzen sehr kompakte und effiziente Programme ermöglicht, aber hohe Anforderungen an Dokumentation und Benutzer stellt.

9.3.2.4 Coterme

Wie in Abschnitt 2.6 ausgeführt, lassen sich rationale Daten durch sogenannte Coterme denotieren. Die ungeschachtelte Cotermnotation, bei der jeder Knoten einem expliziten symbolischen Trägerelement zugeordnet wird, kann natürlich mit der geschachtelten algebraischen Termnotation kombiniert werden, um zyklenfreie Pfade kompakt mit anonymen Zwischenknoten zu beschreiben. Die daraus entstehende Termsprache kann leicht als V→M-Programm implementiert werden, indem der endliche Träger jeder vorkommenden Coalgebra als anonymer Aufzählungstyp verstanden wird. Die Operation der Coalgebra definiert dann eine Interpretation auf diesem Typ, die mit den üblichen Techniken primitiv corekursiv ausgewertet werden kann.

Die Einbettung einer solchen Cotermsprache in TASM würde einige Vorteile bieten. Nicht nur wäre der deklarativen Typsprache eine Konstantensprache mit kompatibler Syntax zur Seite gestellt. Die mit Cotermen beschriebenen Werte sind per Konstruktion berechenbar und könnten daher auch in einem strikten System zu beliebiger Zeit, also zwischen Start der Maschine und erstem Gebrauch, berechnet werden, ohne das beobachtbare Verhalten zu verändern. Die Auswertung von Cotermen könnte sogar gefahrlos im Compiler erfolgen, um zusätzliches Optimierungspotential zu eröffnen.

9.3.3 Offene praktische Fragen

Die folgenden Punkte betreffen die zukünftige Vervollständigung der Implementierung.[2]

9.3.3.1 Speicherverwaltung

Wie in Kapitel 6 diskutiert, führt das coalgebraische Vorgehen bei der Erzeugung von Daten zu spezifischen Komplikationen für jede bekannte Technik der automatischen Speicherverwaltung. Lösungen sind theoretisch bekannt und können zu einem gewissen Preis an Mehraufwand in Speicherverwaltungssysteme integriert werden. Für eine praktische Evaluation dieser erweiterten Verfahren wäre aber nicht nur eine ausgereifte Implementierung der Speicherverwaltung selbst nötig, sondern auch Beispielapplikationen mit realistisch komplexen Speicheranforderungen. Die Codeerzeugung aus V→M-Programmen für Plattformen ohne implizite Speicherverwaltung befindet sich daher noch in einem frühen prototypischen Stadium.

9.3.3.2 Codeerzeugung

Die JVM als Ausführungsplattform für interpretierte und compilierte Programme hat einige Vorteile für einen experimentellen Prototypen, insbesondere implizite Speicherverwaltung und grafische Oberfläche. Andererseits erlaubt das Laufzeitverhalten der Programme aus technologischen Gründen keine faire Evaluation gegen andere funktionale Systeme (siehe Abschnitt 8.5), da die JVM einige Kernkonzepte der coalgebraischen Berechnung nicht direkt unterstützt, so daß diese aufwendig simuliert werden müssen. Besonders die indirekte Adressierung, also der Zugriff auf einzelne Felder einer Zelle als dynamisch adressierte Variablen, erfordert eine teure Indirektion über Proxy-Objekte. Auch für die Zyklenbehandlung kann nicht der Maschinenstack verwendet werden, da Schreibzugriffe und damit die `dito`-Operation aus Sicherheitsgründen unmöglich gemacht werden. Dieselben Einwände gelten auch für andere, vergleichbar komfortable Plattformen wie etwa .NET.

Alternative Plattformen, die einen freieren Umgang mit Referenzen und damit einen effizienteren Umgang mit Ausgabe- und Differenzparametern erlauben, setzen naturgemäß auf einem deutlich niedrigeren Abstraktionsniveau an, und müssen um elementare Laufzeitkomponenten wie Programmzustands- und

[2] Teile dieses Abschnitts können zum Zeitpunkt der endgültigen Veröffentlichung der Arbeit veraltet sein.

Speicherverwaltung ergänzt werden. Die als relativ portable maschinennahe Zielplattform bei Compilerbauern beliebte Sprache C, die auch vom hier vorgestellten Compiler verwendet wird, hat darüberhinaus die Nachteile, daß die Analyse des Stacks zur Laufzeit nicht völlig portabel ausgedrückt werden kann, und daß keine integrierte Unterstützung für *just-in-time*-Compilation oder alternative Aufrufkonventionen zur Verfügung steht. Spezifisch als Zielplattform entworfene Systeme wie etwa C--[51] könnten möglicherweise eine geeignetere Wahl sein, falls sich die Zyklenbehandlung dort implementieren läßt. Die Erzeugung von echtem Maschinencode würde zwar keine Kompromisse bei der Umsetzung der hier entwickelten Techniken erfordern, beinhaltet aber undankbare Aufgaben wie Registerallokation und Reihenfolgenplanung, die den Ansatz für einen Forschungsprototypen unattraktiv machen.

9.3.3.3 Selbstanwendungen

Anhang C kann als Beleg für die einleitend aufgestellte Behauptung gelten, typische Probleme der Interpretation und Übersetzung formaler Sprachen seien oft natürlich zyklischer Natur. Folglich sollte ein System wie das hier implementierte für solche Zwecke besonders geeignet sein, und die Selbstimplementierung (*bootstrapping*) relativ leicht vonstatten gehen. Dem steht zur Zeit das Fehlen einer deklarativen Eingabesprache hinderlich entgegen. Daher wurde bisher kein Versuch in dieser Richtung unternommen, auch wenn die Produktivität der maschinennahen Eingabesprache TASM dank statischer Typisierung und Funktionen höherer Ordnung deutlich höher als bei vergleichbaren Sprachen ausfällt.

Ein erster Schritt zur Selbstanwendung könnte die Codegenerierung zur Laufzeit sein. Diese ist durch die Abstraktion von Typ- und Codemodell und deren Abbildung auf ein Speichermodell bereits im Entwurf der Maschine vorbereitet.

Teil IV

Anhang: Applikationen

Die Konstruktion selbst ist eine Kunst,
ihre Anwendung auf die Welt ein übler Parasit.

L.E.J. Brouwer[6]

In diesem Teil der Arbeit sollen praktische Beispiele für zyklische Probleme gegeben werden. Dabei wird in erster Linie auf den semantischen Aspekt eingegangen. Die vorgestellten Datenstrukturen und Algorithmen werden in einer abstrakten Form dargestellt, die auf Teil II Bezug nimmt, also in Form von Signaturen, Interpretationen und Kalkülen. Um sauber zwischen den erzeugenden Interpretationen und Kalkülen und den erzeugten Funktionen und Prädikaten zu trennen, schreiben wir erstere so und *letztere so*:

1. Ein Funktionsausdruck $fun(x)$ bezeichnet die Berechnung $\mathsf{Ana}_3(\mathbb{C})([], x)$, wobei die fun die Operation der Coalgebra $\mathbb{C}$ definiert.

2. Ein Prädikatausdruck $pred(x)$ bezeichnet die Entscheidung $\mathsf{Res}_3(K, E)\big([], \mathsf{pred}(x)\big)$, wobei die Regelmenge des Kalküls K und die Zusammensetzung der Erwartung E sich aus dem Kontext ergeben.

Auf die technischen Details der Implementierung in der Sprache TASM wird nur am Rande eingegangen. Trotzdem bleibt die Umsetzung nicht dem Leser als Übung überlassen: Lauffähige Programme zu allen Beispielen finden sich im Beispielverzeichnis des zu dieser Arbeit gehörigen Programmsystems. An diesen kann die Arbeitsweise der Algorithmen durch visualisierte Interpretation und Compilation nachvollzogen werden. Auf den Abdruck wurde verzichtet, da der Code kaum lesbarer ist als ein beliebiges anderes Maschinenprogamm von mehreren tausend Zeilen.

Anhang A

Exakte Rationale Arithmetik

Als besonders plastisches Beispielgebiet für die Anwendung zyklischer Berechnungen kann die vermutlich älteste und bekannteste zyklische Datenstruktur der Mathematikgeschichte dienen: die rationalen Zahlen.

A.1 b-adische Brüche

Sei $b > 1$ eine natürliche Zahl. Jede reelle Zahl x mit $0 \leq x \leq 1$ ist eindeutig durch eine unendliche Folge $(a_i)_{i \in \mathcal{N}_+}$ von natürlichen Ziffern mit $0 \leq a_i < b$ charakterisiert, wobei gilt:

$$x = \sum_{i=1}^{\infty} a_i \cdot b^{-i}$$

Wir schreiben, notwendigerweise mit Pünktchen:

$$x \approx 0, a_1 a_2 \ldots_{(b)}$$

Diese charakteristische Folge besitzt genau für die rationalen Zahlen eine endliche Darstellung in Form zweier endlicher Folgen $(p_i)_{1 \in 1 \ldots m \geq 0}$ und $(q_i)_{i \in 1 \ldots n > 0}$, so daß gilt:

$$a_i = \begin{cases} p_i & i \leq m \\ q_i & m < i \leq m + n \\ a_{(i-n)} & i > m + n \end{cases}$$

Die Folge (a_i) besteht also aus der Folge (p_i) gefolgt von unendlich vielen Wiederholungen der Folge (q_i). Daher heißt (p_i) auch *Präfix* und (q_i) heißt *Periode* der Ziffernfolge (a_i). Wir schreiben also für ein rationales x exakt:

$$x = 0, p_1 \ldots p_m \overline{q_1 \ldots q_n}_{(b)}$$

Diese Darstellung ist nicht eindeutig:

1. Ein Anfangsstück der Periode kann dem Präfix zugeschlagen werden:

$$0, p_1 \ldots p_m \overline{q_1 \ldots q_n}_{(b)} = 0, p_1 \ldots p_m q_1 \overline{q_2 \ldots q_n q_1}_{(b)}$$

2. Die Periode kann vervielfacht werden:

$$0, p_1 \ldots p_m \overline{q_1 \ldots q_n}_{(b)} = 0, p_1 \ldots p_m \overline{q_1 \ldots q_n \ldots q_1 \ldots q_n}_{(b)}$$

3. Endliche Brüche haben zwei periodische Darstellungen:

$$0, p_1 \dots p_m \overline{(b-1)}_{(b)} = 0, p_1 \dots (p_m + 1)\overline{0}_{(b)}$$

Die Beachtung dieser Gleichung ist zum Beispiel Voraussetzung für das CANTORsche Diagonalargument. Als Grenzfall gilt auch:

$$1 = 0, \overline{(b-1)}_{(b)}$$

A.1.1 Ziffernfolgen als Datentyp

Der Datentyp *digits* für unendliche Folgen von Ziffern eines Aufzählungstyps *digit* kann analog zu einer Listenstruktur, aber ohne Abbruchfall definiert werden:

type *digit* $= \{\, 0, \dots, (b-1) \,\}$
type *digits* $= (\mathsf{head}: \mathit{digit}, \mathsf{tail}: \mathit{digits})$

Die zugehörige Signatur $\Sigma_b = \mathcal{C}_{0\dots(b-1)} \times \mathcal{I}$ hat, wie bereits in Beispiel 2.121 erwähnt, das reelle Intervall $[0;1]$ als finale und das rationale Intervall $[0;1]$ als rationale Coalgebra. Konkrete Daten im Speicher haben stets die Form eines linearen Präfix, der (p_i) definiert, gefolgt von einem Zyklus, der (q_i) definiert.

A.1.2 Meta-Notation

In den folgenden Abschnitten werden wir Berechnungen auf Ziffernfolgen formal als Interpretation beziehungsweise Regelsysteme definieren. Als Meta-Notation für den Datentyp der Ziffernfolgen wird dabei die Familienschreibweise beibehalten. Alle Familien seien mit natürlichen Zahlen, beginnend bei 1, indiziert. Die coinduktive Zerlegung einer Ziffernfolge (a_i) in erste Ziffer und Restfolge schreiben wir als $a_1, (a_{i+1})$, die umgekehrte Konstruktion mit $(:)$ wie bei den endlichen Listen. Es gilt also:

$$\begin{aligned}(a_i) &= a_1 : (a_{i+1}) \\ a_k &= (\pi_{\mathsf{head}} \circ \pi_{\mathsf{tail}}^{k-1})\big((a_i)\big) \\ (a_{i+k}) &= \pi_{\mathsf{tail}}^{k}\big((a_i)\big)\end{aligned}$$

A.2 Elementare Operationen auf rationalen Zahlen

A.2.1 Ordnung

Die totale Ordnung rationaler Zahlen läßt sich als lexikographische Ordnung der Ziffernfolgen über der Ordnung der Ziffern konstruieren. Zur Unterscheidung schreiben wir $<, \leq, =$ für die Relationen auf Ziffern, und $\dot{<}, \dot{\leq}, \doteq$ für deren Pendants auf Ziffernfolgen. Dann definieren wir als Regelsystem:

$$\frac{a_1 < b_1}{(a_i) \mathrel{\dot{<}} (b_i)} \qquad \frac{a_1 < b_1}{(a_i) \mathrel{\dot{\leq}} (b_i)}$$

$$\frac{a_1 = b_1 \quad (a_{i+1}) \mathrel{\dot{<}} (b_{i+1})}{(a_i) \mathrel{\dot{<}} (b_i)} \qquad \frac{a_1 = b_1 \quad (a_{i+1}) \mathrel{\dot{\leq}} (b_{i+1})}{(a_i) \mathrel{\dot{\leq}} (b_i)} \qquad \frac{a_1 = b_1 \quad (a_{i+1}) \doteq (b_{i+1})}{(a_i) \doteq (b_i)}$$

wobei $\dot{<}$ als kleinster Fixpunkt, $\dot{\leq}$ und $\doteq$ als größter Fixpunkt anzunehmen sind: Gerät man beim Entscheiden von $(a_i) \mathrel{\dot{<}} (b_i)$ in einen Zyklus, dann sind alle Paare a_i, b_i gleich, und das Ergebnis ist f; bei $(\dot{\leq})$ und $(\doteq)$ ist das Ergebnis entsprechend w. Der Kalkül hat die Prädikate $<, \leq, =$ als Importschnittstelle und ist entsprechend zu erweitern.

Alle drei Prädikate lassen sich mit den Mitteln der virtuellen Maschine zu einer einzigen Ordnungsfunktion mit dreiwertigen Resultattyp $(<, =, >)$ kombinieren. Dabei können $<$ und $>$ nur als reguläre Abbruchfälle, $=$ nur als Zyklenbehandlung auftreten.

A.2.2 Normierung

Die im vorigen Abschnitt definierte Ordnung auf Ziffernfolgen entspricht nicht exakt der Ordnung der rationalen Zahlen: Besitzt eine Zahl zwei Darstellungen, sind diese lexikographisch unterscheidbar. Um diese Mehrdeutigkeit zu vermeiden, sollte von den beiden Perioden $\overline{0}$ und $\overline{(b-1)}$ konsistent nur eine verwendet werden, wobei wir wie CANTOR der letzteren den Vorzug geben. Der Rumpf einer abstrakten Interpretation, die alle 0-terminierten Zahldarstellungen auf ihre $(b-1)$-terminierten Äquivalente abbildet, sieht wie folgt aus:

$$\mathsf{norm}_0\big((a_i)\big) = \begin{cases} (a_1 - 1) : \overline{(b-1)} & a_1 > 0 \wedge (a_{i+1}) \doteq \overline{0} \\ (a_i) & \text{sonst} \end{cases}$$

Da im zweiten Fall das Interpretationsergebnis (a_i) die Restfolge (a_{i+1}) an Rekursionsposition enthält, wird hierdurch eine corekursive Funktion generiert, die die gesamte Ziffernfolge bearbeitet.

Diese Art der Normierung sollte nicht mit der Normierung von Gleitkommazahlen verwechselt werden, die das „andere Ende“ der Ziffernfolge betrifft.

A.3 Arithmetik auf rationalen Zahlen

Von den vier Grundoperationen der Arithmetik werden im folgenden zwei behandelt: Die Division, weil sie die typischen Probleme der Corekursion besonders deutlich illustriert, und die Subtraktion als notwendige Voraussetzung. Addition und Multiplikation lassen sich darauf zurückführen. Bei den verwendeten Algorithmen handelt es sich um periodische Verallgemeinerungen der Standardverfahren für Stellenwertsysteme, wie sie zur allgemeinen Schulbildung gehören.

A.3.1 Subtraktion

Das klassische Verfahren zur Subtraktion wird durch den Datenfluß der stellenweisen Überträge in eine streng sequentielle Reihenfolge seiner Einzelschritte gebracht, beginnend bei der niederstwertigen Stelle bis hin zur höchstwertigen:

$$\begin{array}{cccccccc} & x_1 & x_2 & x_3 & x_4 & x_5 & \dots \\ - & y_1 & y_2 & y_3 & y_4 & y_5 & \dots \\ \hline c_1 & c_2 & c_3 & c_4 & c_5 & \dots & \\ = & z_1 & z_2 & z_3 & z_4 & z_5 & \dots \end{array}$$

Für natürliche Zahlen oder endliche Brüche stellt dieses Vorgehen kein Problem dar; unendliche Brüche besitzen aber keine niederstwertige Stelle. Stattdessen muß man hier bei der höchstwertigen Stelle beginnen und Überträge *rückwärts* geltend machen. Karczmarczuk verfolgt in [25] einen vom Autor selbst als eher abwegig bezeichneten Ansatz unter massiver Beanspruchung von *lazy*-Auswertung. In einem strikt corekursiven Szenario ist die Lösung einfacher.

1. Man beginnt mit einer Approximation durch stellenweise Halbsubtraktion:

$$\begin{array}{cc|c|c|c|c|c|l} & x_1 & x_2 & x_3 & x_4 & x_5 & \dots & \\ - & y_1 & y_2 & y_3 & y_4 & y_5 & \dots & \\ \hline = & z'_1 & z'_2 & z'_3 & z'_4 & z'_5 & & z'_i = (x_i - y_i) \bmod b \\ & c'_1 & c'_2 & c'_3 & c'_4 & c'_5 & & -c'_i = (x_i - y_i) \operatorname{div} b \end{array}$$

   Schließt man globalen Überlauf aus, dann gilt $c_1 = 0$.

2. Danach sind die Überträge (c_i) versetzt anzurechnen, was einfach durch Iteration des Approximationsschrittes geschieht:

$$(x_i) - (y_i) = (z'_i) - (c'_{i+1}) \tag{A}$$

   Das Paradoxon der rückwirkenden Verrechnung wird also vermieden, indem Zwischenergebnis und Übertrag simultan vollständig berechnet und anschließend gegeneinander verschoben werden.

3. Das exakte Ergebnis ist erreicht, sobald $c' \doteq \overline{0}$, also wenn keine Überträge mehr auftreten. Da jede Stelle höchstens einmal im gesamten Verfahren einen Übertrag abwerfen kann, und alle Wiederholungen der Periode simultan bearbeitet werden, ist die Anzahl der Schritte bis zur Konvergenz auf die Summe von Präfix- und Periodenlänge, also die Zellenzahl des Ergebnisses beschränkt.

Lemma A.1. *Die Gleichung* (A) *ist korrekt.*

Beweis. Anwendung der Interpretation von Ziffernfolgen und der üblichen Sätze für Reihen ergibt:

$$\begin{aligned}
\Big(\sum_{i=1}^{\infty} x_i \cdot b^{-i}\Big) - \Big(\sum_{i=1}^{\infty} y_i \cdot b^{-i}\Big) &= \sum_{i=1}^{\infty} (x_i - y_i) \cdot b^{-i} \\
&= \sum_{i=1}^{\infty} \Big(\big((x_i - y_i) \bmod b\big) + b \cdot \big((x_i - y_i) \operatorname{div} b\big)\Big) \cdot b^{-i} \\
&= \sum_{i=1}^{\infty} (z'_i - b \cdot c'_i) \cdot b^{-i} \\
&= \Big(\sum_{i=1}^{\infty} z'_i \cdot b^{-i}\Big) - b \cdot \Big(\sum_{i=1}^{\infty} c'_i \cdot b^{-i}\Big) \\
&= \Big(\sum_{i=1}^{\infty} z'_i \cdot b^{-i}\Big) - \underbrace{b \cdot c'_1}_{=0} - \Big(\sum_{i=1}^{\infty} c'_{i+1} \cdot b^{-i}\Big)
\end{aligned}$$

□

Die corekursive Berechnung von Zwischenergebnis- und Übertragsfolge kann durch zwei abstrakte Interpretationen geschehen:

$$\begin{aligned}
\mathsf{diff}_0\big((x_i),(y_i)\big) &= \big((x_1 - y_1) \bmod b\big) : \big((x_{i+1}),(y_{i+1})\big) \\
\mathsf{carry}_0\big((x_i),(y_i)\big) &= \big((x_1 - y_1) \operatorname{div} b\big) : \big((x_{i+1}),(y_{i+1})\big)
\end{aligned}$$

Da beide dieselbe Rekurrenzrelation haben, lassen sie sich als eine Prozedur *hsub* mit zwei Ausgaben implementieren.

Die eigentliche Subtraktion läßt sich dann rekursiv ohne Zyklenbehandlung definieren:

$$sub\big((x_i),(y_i)\big) = \begin{cases} (x_i) & (y_i) \doteq \overline{0} \\ sub\big((z_i),(c_{i+1})\big) & (y_i) \dot{\neq} \overline{0} \wedge hsub\big((x_i),(y_i)\big) = \big((z_i),(c_i)\big) \end{cases}$$

A.3.2 Division

Das klassische Verfahren zur Division stellt, wie bereits in Kapitel 1 erwähnt, einen idealen Prototypen der Zyklenbehandlung dar. Die technische Umsetzung ist aber nicht trivial, was in diesem Abschnitt nachvollzogen werden soll.

Anders als die Subtraktion, die als endliche Iteration eines zyklischen Schritts strukturiert ist, läßt sich die Division am besten als zyklische Iteration eines endlichen Schritts darstellen. Der Schritt besteht in der Berechnung der höchstwertigen Stelle des Quotienten und des Rests als Ziffernfolge. Dieser Schritt wird solange unter Verschiebung von Dividend oder Divisor iteriert, bis ein Rest zyklisch auftritt und damit die Periode des Quotienten bestimmt ist.

Der Einzelschritt geschieht in der Funktion *hdiv* durch gezählte Subtraktion:

$$hdiv\big((x_i),(y_i)\big) = \begin{cases} \big(0,(x_i)\big) & (x_i) \dot{<} (y_i) \\ \big(succ(z),(r_i)\big) & (x_i) \dot{\geq} (y_i) \wedge hdiv\big(sub\big((x_i),(y_i)\big),(y_i)\big) = \big(z,(r_i)\big) \end{cases}$$

Auch wenn die mathematische Notation dies nicht leicht erkennbar macht, ist *hdiv* *tail*-rekursiv bis auf den Konstruktor *succ*. In der Implementierung kann dieser sein Argument als Differenzparameter erwarten, und *hdiv* als echte Schleife ausgeführt werden. Diese Optimierung wäre nicht möglich, wenn Ziffern nicht in Unärcodierung als freier Datentyp mit Konstruktor *succ*, sondern beispielsweise in der üblichen Binärcodierung, dargestellt werden.

Damit die zyklische Iteration dieses Schritts effektiv berechnet werden kann, müssen einige Voraussetzungen geklärt werden:

1. Das Intervall $[0;1]$ ist unter Division nicht abgeschlossen. Wir betrachten hier ohne Beschränkung der Allgemeinheit nur Fälle mit Ergebnis innerhalb des Intervalls. Erweitert man die Zahldarstellung um ein Gleitkomma, dann lassen sich alle anderen Fälle darauf zurückführen.

2. Die Verschiebung von Dividend (x_i) und Divisor (y_i) gegeneinander darf nicht durch Verlängerung des Divisors erfolgen, weil sonst kein Zyklus, sondern eine unendliche divergente Folge von Resten ensteht. Stattdessen muß gewährleistet werden, daß der Dividend in jedem Schritt verkürzt werden kann. Dazu fordern wir $y_1 = 0$, was in jedem Fall durch Erweiterung beider Argumente erfüllt werden kann. Daraus folgt, daß in

$$hdiv\big((x_i),(y_i)\big) = \big(z,(r_i)\big)$$

auch $r_1 = 0$ gilt. Im nächsten Schritt kann also mit (r_{i+1}) als Dividend gerechnet werden.

Damit ergibt sich folgender Ansatz für eine abstrakte Interpretation:

$$\mathsf{div}_0\big((x_i),(y_i)\big) = z : \big((r_{i+1}),(y_i)\big) \qquad \text{wobei } hdiv\big((x_i),(y_i)\big) = \big(z,(r_i)\big)$$

Leider täuscht die mathematische Notation über ein technisches Problem hinweg: Die Hilfsfunktion *hdiv* erzeugt im Seiteneffekt eine frische Ziffernfolge, die sie als Rest (r_i) zurückliefert. Selbst wenn nach einigen Schritten ein semantisch äquivalenter Rest auftritt, wird dieser nicht ohne weiteres als Zyklus erkannt, da es sich praktisch um zwar bisimilare, aber nicht identische Zellenstrukturen handelt. Es handelt sich also um einen Quasizyklus wie in Abschnitt 6.3 beschrieben. Die virtuelle Maschine stellt für diesen Fall ein Meta-Attribut `quasi` zur Verfügung, das eine Zyklenerkennung modulo Bisimilarität bewirkt, und von Interpreter und Compiler akzeptiert wird.

A.4 Strukturanalyse rationaler Zahlen

Dieser Abschnitt soll die Annahme stützen, daß die finale Semantik, wie sie von den in dieser Arbeit verwendeten Algorithmen verlangt wird, eine adäquate Alternative zu nicht-finalen Sichtweisen darstellt, wie sie aus imperativen Sprachen mit Adressoperationen bekannt sind. Exemplarisch für die rationalen Zahlen soll gezeigt werden, daß die finale Coalgebra als semantisches Modell nicht von wesentlichen Informationen abstrahiert.

A.4.1 Bestimmung von Präfix und Periode

Es ist möglich, die Länge von Präfix und Periode einer Ziffernfolge allein aus deren finaler Semantik, ohne Beachtung der Graphstruktur im Speicher, zu bestimmen.

Der Test $\mathsf{peri}\big((a_i), m, n'\big)$, ob eine Ziffernfolge (a_i) die Präfixlänge m und die Periodenlänge $n = n' + 1$ hat, ergibt sich aus dem größten Fixpunkt des folgenden Regelsystems:

$$\frac{\mathsf{peri}\big((a_{i+1}), m, n\big)}{\mathsf{peri}\big((a_i), \mathsf{s}(m), n\big)} \qquad \frac{(a_i) \doteq (a_{i+1})}{\mathsf{peri}\big((a_i), \mathsf{z}, \mathsf{z}\big)} \qquad \frac{\mathsf{peri}\big((a_{(n+1)\cdot(i-1)+1}), \mathsf{z}, \mathsf{z}\big) \quad \mathsf{peri}\big((a_{i+1}), \mathsf{z}, \mathsf{s}(n)\big)}{\mathsf{peri}\big((a_i), \mathsf{z}, \mathsf{s}(n)\big)}$$

Hinter den Regeln stehen folgende Argumente im Klartext:

1. Eine Ziffernfolge hat einen nichtleeren Präfix der Länge $m+1$, genau dann wenn die Restfolge nach der ersten Ziffer einen Präfix der Länge m hat.

2. Eine Ziffernfolge ohne Präfix hat die Periodenlänge 1, genau dann wenn sie gleich ihrer Restfolge ist.

3. Eine Ziffernfolge ohne Präfix hat die Periodenlänge $n + 1$, genau dann wenn die Auswahl jeder $(n + 1)$-ten Ziffer, beginnend bei der ersten, die Periodenlänge 1 hat, und die Restfolge nach der ersten Ziffer ebenfalls die Periodenlänge $n + 1$ hat.

Die Regeln sind so formuliert, daß sie auch für $m = \omega$ oder $n = \omega$ effektiv gelten. Wie man die ausgedünnte Teilfolge $(a_{(n+1)\cdot(i-1)+1})$ berechnet, kann als Spezialfall aus der periodischen Listenselektion wie in Abschnitt B.2.3 abgeleitet werden.

Durch systematische Aufzählung von Kandidaten für m und n, etwa nach dem CANTORschen Paarungsverfahren, kann stets eine Lösung berechnet werden. Aus einer beliebigen Lösung kann außerdem im Umkehrschluß aus den oben genannten Freiheitsgraden die minimale Lösung abgeleitet werden:

1. Gilt $\mathsf{peri}\big((a_i), m, n-1\big)$, dann teste auch $\mathsf{peri}\big((a_i), m-j, n-1\big)$ für $1 \leq j \leq m$.

2. Gilt $\mathsf{peri}\big((a_i), m, n-1\big)$, dann teste auch $\mathsf{peri}\big((a_i), m, n/j-1\big)$ für $1 < j \leq n$.

3. Schlagen alle diese Tests fehl, ist die gegebene Lösung minimal.

A.4.2 Bruchdarstellung

Aus einer Lösung für Präfix- und Periodenlänge einer Ziffernfolge läßt sich eine Darstellung als gemeiner Bruch konstruieren:

$$\begin{aligned} x &= 0, p_1 \ldots p_m \overline{q_1 \cdots q_n}_{(b)} \\ b^m \cdot x &= p_1 \ldots p_m, \overline{q_1 \cdots q_n}_{(b)} \\ b^{m+n} \cdot x &= p_1 \ldots p_m q_1 \ldots q_n, \overline{q_1 \cdots q_n}_{(b)} \end{aligned}$$

Die Subtraktion der vorletzten von der letzten Zeile löscht die Nachkommastellen aus:

$$\begin{aligned}(b^{m+n} - b^m) \cdot x &= p_1 \dots p_m q_1 \dots q_{n(b)} - p_1 \dots p_{m(b)} \\ &= a_1 \dots a_{m+n(b)} - a_1 \dots a_{m(b)} \\ x &= \frac{\sum_{i=1}^{m+n} a_i \cdot b^{m+n-i} - \sum_{i=1}^{m} a_i \cdot b^{m-i}}{b^{m+n} - b^m}\end{aligned}$$

Der Zähler läßt sich am besten mit dem *tail*-rekursiven HORNER-Verfahren berechnen:

$$\begin{aligned}H\big((a_i), b, 0, c\big) &= c \\ H\big((a_i), b, k, c\big) &= H\big((a_{i+1}), b, k-1, a_1 + b \cdot c\big)\end{aligned}$$

Es gilt:

$$H\big((a_i), b, k, 0\big) = \sum_{i=1}^{k} a_i \cdot b^{k-i}$$

Anhang B

Unendliche Listen

Die Datenstruktur der unendlichen Listen ist als das ursprüngliche Anwendungsgebiet der *lazy*-Auswertung im Zusammenhang mit nichtstrikter Semantik bereits sehr gründlich erforscht. Die in diesem Kapitel vorgestellten Algorithmen werden daher in zwei Kategorien unterteilt:

1. Algorithmen für Probleme, die auch *lazy* lösbar sind. Naturgemäß entsprechen die strikten Algorithmen weitestgehend ihren nichtstrikten Pendants. Lediglich die Garantie der Termination wird mit der Einschränkung auf rationale Strukturen erkauft. In diese Kategorie fallen Verfahren, die als primitiv corekursive Funktionen darstellbar sind.
2. Algorithmen für Probleme, die *lazy* ohne radikale Umcodierung nicht lösbar sind. Ein Versuch, diese Algorithmen beispielsweise in Haskell direkt nachzuprogrammieren, ist also vergeblich. In diese Kategorie fallen Verfahren, die von einem als corekursiver Kalkül darstellbaren Prädikat abhängen.

Zur besseren Lesbarkeit abstrakter Interpretationen werden eingebettete corekursive Argumente unterstrichen.

B.1 Listen als Datenstruktur

Für alle folgenden Algorithmen modellieren wir Listen von Elementen eines Typs α durch folgende Typdefinition:

$$\textbf{type } \mathit{list}(\alpha) = \big\{\mathsf{nil}, \mathsf{cons} : \mathit{cell}(\alpha)\big\}$$
$$\textbf{type } \mathit{cell}(\alpha) = \big(\mathsf{car} : \alpha, \mathsf{cdr} : \mathit{list}(\alpha)\big)$$

Nimmt man an, daß sich eine wohlgeformte Liste zwar selbst als Teilliste, aber nicht als Element enthalten kann[1], dann entspricht das der folgenden Signatur:

$$\Lambda_A = \mathcal{C}_1 + A \times \mathcal{I}$$

Für die Definition von Interpretationen und Kalkülen wollen wir die bereits gewohnte Notation verwenden. Sei (C, γ) eine Λ_A-Coalgebra von Listen:

1. Für $l \in C$ mit $\gamma(l) = \iota_1(\star)$, also eine leere Liste, schreiben wir $[]$.
2. Für $l \in C$ mit $\gamma(l) = \iota_2(c)$ und $\pi_1(c) = a$ und $\pi_2(c) = l'$, also eine Liste mit Anfangselement a und Restliste l', schreiben wir $a : l'$.

Wird eine Interpretation ausschließlich mit Hilfe dieser Muster definiert, ist sie automatisch abstrakt.

[1] In bester mathematischer Tradition wäre diese Eigenschaft als *Linksfundiertheit* zu bezeichnen.

B.2 Gewöhnliche Operationen auf unendlichen Listen

B.2.1 Transformation der Elemente (Map)

Die Anwendung einer Funktion auf alle Elemente erfolgt im zyklischen exakt genauso wie im azyklischen Fall. Die einzige zusätzliche Aufgabe, den Zyklus in der Ausgabe zu schließen, wird ja durch die allgemeine Zyklenbehandlung geleistet. Man definert also für eine beliebige gegebene Funktion f eine homomorphe Interpretation:

$$\begin{aligned}\mathsf{map}_f([]) &= []\\ \mathsf{map}_f(a:l) &= f(a):\underline{l}\end{aligned}$$

Praktisch wird eine solche generische Interpretation durch eine Prozedur mit zwei Eingabeparametern realisiert. Der erste dieser Parameter wird mit der Funktion f belegt, und bleibt in allen Rekursionen konstant. Da dieser Fall häufig eintritt, sieht das Programmmodell der virtuellen Maschine und die Sprache TASM ein Attribut `invar` für solche Parameter vor. Interpreter und Compiler erkennen das Attribut können die Zyklenerkennung optimieren, da der Parameter hierfür keine Rolle spielt.

An diesem sehr kompakten Beispiel läßt sich auch demonstrieren, wie die von einer Interpretation erzeugte Funktion hergeleitet wird. Sei $\mathbb{F} = (F, \phi)$ eine finale Λ_A-Coalgebra. Sei M_f ein geeigneter Definitionsbereich, so daß $\mathbb{M}_f = \big(M_f, \mathsf{map}_f\big)$ eine Λ_A-Coalgebra ist. Für den Anamorphismus $[\![\mathbb{M}_f]\!]$ gilt:

$$\phi \circ [\![\mathbb{M}_f]\!] = \Lambda_A\big([\![\mathbb{M}_f]\!]\big) \circ \mathsf{map}_f$$

Daraus folgt die erwartete Spezifikation:

$$\begin{aligned}\phi\big([\![\mathbb{M}_f]\!]([])\big) &= \Lambda_A\big([\![\mathbb{M}_f]\!]\big)\big(\mathsf{map}_f([])\big)\\ &= \Lambda_A\big([\![\mathbb{M}_f]\!]\big)([])\\ &= []\\ \phi\big([\![\mathbb{M}_f]\!](a:l)\big) &= \Lambda_A\big([\![\mathbb{M}_f]\!]\big)\big(\mathsf{map}_f(a:l)\big)\\ &= \Lambda_A\big([\![\mathbb{M}_f]\!]\big)\big(f(a):l\big)\\ &= f(a):[\![\mathbb{M}_f]\!](l)\end{aligned}$$

B.2.2 Verkettung

Die Verkettung zweier Listen ist leicht zu verallgemeinern. Die primitiv rekursive Spezifikation taugt auch unmittelbar als primitive Corekursion, wie durch folgende Interpretation definiert:

$$\begin{aligned}\mathsf{app}([],[]) &= []\\ \mathsf{app}([],a:l) &= a:\underline{([],l)}\\ \mathsf{app}(a:l,l') &= a:\underline{(l,l')}\end{aligned}$$

Dabei fällt auf, daß die ersten beiden Zeilen bei einer konkreten Implementierung üblicherweise gemeinsam behandelt werden. In einem algebraischen Kontext würde man rekursiv definieren:

$$app([],l) = l$$

Um die Definition aber in eine corekursive Normalform zu bringen, müssen die Fälle getrennt behandelt werden. Es handelt sich dabei um eine prinzipielle Schwäche der primitiven Corekursion: Daß eine Funktion ihr Argument unverarbeitet zurückliefern soll, kann nicht direkt ausgedrückt werden. Stattdessen muß man sich mit der Herstellung einer exakten Kopie behelfen, die unter finaler Semantik zur identischen Abbildung kollabiert. In einer praktischen Implementierung ist es allerdings aus Effizienzgründen

wünschenswert, statt einer Kopie das Original als Ausgabe zu erhalten. Unsere virtuelle Maschine unterstützt das, wie die TASM-Implementierung von *app* in der Beispiel-Bibliothek zeigt. Wir schreiben für die Funktion *app* auch $+\!\!+$.

Auf kategorientheoretischer Ebene wird eine solche corekursive Berechnung mit Abbruchfall durch einen sogenannten *Apomorphismus* beschrieben, siehe [63].

B.2.2.1 Wiederholung

Ein Spezialfall von Verkettung ist die periodische Wiederholung einer gegebenen Liste l. Diese läßt sich am leichtesten über eine Familie von Hilfsinterpretationen rep'_l definieren:

$$\begin{aligned} rep(l) &= rep'_l([]) \\ \mathsf{rep}'_l([]) &= l \\ \mathsf{rep}'_l(a : l') &= a : \underline{l'} \end{aligned}$$

Daß die Definition von $\mathsf{rep}'(l)$ wohlgeformt ist, sieht man durch Aufspalten der ersten Gleichung:

$$\begin{aligned} \mathsf{rep}'_{[]}([]) &= [] \\ \mathsf{rep}'_{a:l}([]) &= a : \underline{l} \end{aligned}$$

Die Funktion *rep* läßt sich auch auf das n-fache Wiederholen *nrep* verallgemeinern. Dann gilt $rep(l) = nrep(l, \omega)$.

B.2.3 Periodisches Einfügen und Auswählen

Für diesen Abschnitt benötigen wir neben den Listen auch die natürlichen Zahlen. Eine gemeinsame Signatur für beide haben wir bereits in Kapitel 4 kennengelernt. Die Konstruktoren schreiben wir wieder als z und s. Dabei ist zu beachten, daß auch ω mit $\mathsf{s}(\omega) = \omega$ zu den natürlichen Zahlen gehört.

Um Elemente flexibel in eine Liste einzufügen zu können, definieren wir eine Interpretation, die das periodische Einfügen eines Elementes a in eine Liste l, beginnend an der i-ten Position und dann jeweils wieder nach k Originalelementen, spezifiziert:

$$\begin{aligned} \mathsf{ins}(l, \mathsf{z}, k, a) &= a : \underline{(l, k, k, a)} \\ \mathsf{ins}([], \mathsf{s}(i), k, a) &= [] \\ \mathsf{ins}(a : l, \mathsf{s}(i), k, a') &= a : \underline{(l, i, k, a')} \end{aligned}$$

Für $i = \omega$ wird die Liste kopiert. Einmaliges Einfügen an der i-ten Stelle wird durch $k = \omega$ geleistet.

Die beiden komplementären Operationen des periodischen Löschens und Auswählens lassen sich nicht so einfach in eine primitiv corekursive Normalform bringen. Als hinreichende Approximationen mögen folgende, lokal rekursive Definitionen mit den bereits bekannten Bedeutungen der Argumente l, i und k dienen:

$$\begin{aligned} \mathsf{del}([], i, k) &= [] \\ \mathsf{del}(a : l, \mathsf{z}, k) &= \mathsf{del}(l, k, k) \\ \mathsf{del}(a : l, \mathsf{s}(i), k) &= a : \underline{(l, i, k)} \end{aligned}$$

Diese Definition hat eine Lücke für unendliches l und $k = 0$. Es ist also auf diese einfache Art nicht möglich, den unendlichen Rest einer Liste vollständig zu löschen. Für Abhilfe siehe Abschnitt B.3.3.

$$\begin{aligned} \mathsf{sel}([], i, k) &= [] \\ \mathsf{sel}(a : l, \mathsf{z}, k) &= a : \underline{(l, k, k)} \\ \mathsf{sel}(a : l, \mathsf{s}(i), k) &= \mathsf{sel}(l, i, k) \end{aligned}$$

Auch diese Definition ist partiell und undefiniert für unendliches l und $i = \omega$. Da sich aber $i = \omega$ in unserem Ansatz entscheiden läßt, kann dieser Fall gesondert behandelt werden:

$$\mathsf{sel}(l, \omega, k) = []$$

B.3 Ungewöhnliche Operationen auf unendlichen Listen

Dieser Abschnitt soll sich mit Problemen unendlicher Listen beschäftigen, die gemeinhin als unlösbar gelten. Im nichtstrikten Fall ist das auch zutreffend, nicht aber im auf rationale Coalgebren eingeschränkten strikten Fall. Da die so erhaltenen Lösungen nicht immer auf den ersten Blick anschaulich erscheinen, muß bei der Beschreibung etwas weiter ausgeholt werden, als im eher konventionellen, vorhergehenden Abschnitt.

B.3.1 Quantoren

Ob alle Elemente beziehungsweise wenigstens ein Element einer Liste ein gegebenes Prädikat erfüllt, läßt sich mit dem folgenden Kalkül effektiv entscheiden:

$$\frac{}{\mathsf{all}([])} \qquad \frac{\mathsf{P}(a) \quad \mathsf{all}(l)}{\mathsf{all}(a:l)} \qquad \frac{\mathsf{P}(a)}{\mathsf{any}(a:l)} \qquad \frac{\mathsf{any}(l)}{\mathsf{any}(a:l)}$$

Der generische Kalkül hat das Prädikat P als Importschnittstelle. Durch Erweitern mit dem gewünschten Prädikat P als Bedingung wird der Kalkül spezialisiert. Dabei ist jeweils für all das größte Modell, für any das kleinste intendiert: Wird bei $\mathsf{all}(a:l)$ ein Zyklus erkannt und es gilt gleichzeitig $\mathsf{P}(a)$, dann gilt P bereits für alle Elemente des Zyklus, und das Ergebnis ist w. Wird umgekehrt bei $\mathsf{any}(a:l)$ ein Zyklus erkannt und es gilt nicht $\mathsf{P}(a)$, dann gilt P für kein Element auf dem Zyklus, und das Ergebnis ist f.

Wir schreiben $all(P)$, $any(P)$ für die Prädikate des P-bedingten Kalküls.

B.3.2 Lexikographische Ordnung und Ketten

Die im vorigen Kapitel diskutierten Ordnungsrelationen auf Ziffernfolgen sind ein Spezialfall der lexikographischen Ordnung auf Listen. Sei durch die Relationen $<$ und $=$ eine totale Ordnung des Elementtyps festgelegt. Dann definiert man:

$$\frac{a < b}{a : l \dot{<} b : m} \qquad \frac{a < b}{a : l \dot{\leq} b : m}$$

$$\frac{a = b \quad l \dot{<} m}{a : l \dot{<} b : m} \qquad \frac{a = b \quad l \dot{\leq} m}{a : l \dot{\leq} b : m} \qquad \frac{a = b \quad l \doteq m}{a : l \doteq b : m}$$

Wobei für $\dot{<}$ das kleinste, für $\dot{\leq}$ und $\doteq$ das größte Modell intendiert ist.

Eng verwandt ist das Kettenproblem, also die Frage, ob für eine gegebene Relation R und eine Liste $a_1 : a_2 : a_3 : \ldots$ die Formel $a_1\ R\ a_2 \wedge a_2\ R\ a_3 \wedge \ldots$ gilt. Diese wird auch häufig als $a_1\ R\ a_2\ R\ a_3\ R \ldots$ geschrieben. Eine generische Lösung liefert folgender Kalkül:

$$\frac{}{\mathsf{chain}([])} \qquad \frac{}{\mathsf{chain}(a:[])} \qquad \frac{a\ \mathsf{R}\ a' \quad \mathsf{chain}(a':l)}{a : a' : l}$$

Die Relation R ist die Importschnittstelle des generischen Kalküls, der durch Erweitern mit der gewünschten Relation R instanziiert wird. Wir schreiben dann $chain(R)$ für das erzeugte Prädikat. Es ist jeweils das größte Modell intendiert.

B.3.3 Filter

Das Filtern einer Liste mit einem gegebenen Prädikat zählt zu den klassischen Beispielen für potentiell unproduktive Corekursionen in nichtstrikter Algebra: Trifft das Prädikat auf das Anfangselement der Eingabeliste zu, dann steht der Anfang der Ausgabeliste fest. Andernfalls muß aber auf das Ergebnis des nächsten Filterschritts gewartet werden. Werden also unendlich viele Elemente in Folge herausgefiltert, dann kann das nächste Ausgabeelement nicht in endlicher Zeit berechnet werden.

Im Fall der rationalen zyklischen Listen existiert aber ein mehrstufiges Verfahren, das mit den Mitteln der strikten Corekursion effektiv durchführbar ist:

1. Zunächst werden auszufilternde Elemente nicht aus der Liste entfernt, sondern durch Abbildung auf einen zusätzlichen *Nullwert* $(-)$ als gestrichen markiert. Dabei handelt es sich um eine Anwendung der bereits diskutierten *map*-Operation:

$$mark(P)(a) = \begin{cases} a & P(a) \\ - & \neg P(a) \end{cases}$$

2. Anschließend werden gestrichene Elemente entfernt. Dabei kann festgestellt werden, ob die Restliste aus einem Zyklus von ausschließlich gestrichenen Elementen besteht, was der oben beschriebenen Nichttermination entspricht. In diesem Fall wird der Zyklus durch eine leere Restliste ersetzt. Andernfalls kann das nächste Element durch eine endliche lokale Rekursion gefunden werden:

$$\begin{aligned} \mathsf{sweep}([]) &= [] \\ \mathsf{sweep}(- : l) &= \begin{cases} [] & all(-)(l) \\ \mathsf{sweep}(l) & \text{sonst} \end{cases} \\ \mathsf{sweep}(a : l) &= a : l \end{aligned}$$

   In der konkreten Implementierung ist es nicht einmal nötig, auf den *all*-Quantor zurückzugreifen: Der entsprechende Fall wird auch durch die Zyklenerkennung abgedeckt. Dazu wird die *sweep*-Operation etwas anders definiert:

$$\begin{aligned} \mathsf{sweep}([]) &= [] \\ \mathsf{sweep}(- : l) &= \mathsf{sweep}\big(next(l)\big) \\ \mathsf{sweep}(a : l) &= a : l \end{aligned}$$

   Dabei liefert die Hilfsfunktion *next* eine Restliste, die mit dem nächsten ungestrichenen Element beginnt, falls ein solches existiert, und ansonsten leer ist. Diese kann nicht als primitiv corekursive Funktion durch eine Interpretation beschrieben werden, die Implementierung ist aber trivial. In diesem Fall steht also ausnahmsweise etwas anderes als eine `dito`-Operation im zweiten Rumpf.

Das Filtern mit einem Prädikat P erfolgt dann durch Komposition:

$$filter(P) = sweep \circ map\big(mark(P)\big)$$

B.3.4 Sortieren

Aufbauend auf den Definitionen der vorhergehenden Abschnitte läßt sich das bekannte *Quicksort*-Verfahren auf zyklische Listen verallgemeinern. Dabei bleibt der charakteristische Ansatz, eine nichtleere Liste anhand ihres ersten Elementes zu partitionieren und die Teile rekursiv zu sortieren, erhalten. Nur das Abbruchkriterium muß angepaßt werden: Anders als im endlichen Fall wird durch die Partitionierung nicht immer eine einelementige Liste erreicht. Im zyklischen Fall enthält eine Partition nicht unbedingt weniger Elemente als die Gesamtliste, aber weniger *verschiedene*. Es kann also abgebrochen werden, sobald eine Liste nur noch gleiche Elemente enthält. Diese ist dann, wie im endlichen Fall, trivial sortiert.

Anstelle des Tests auf Gleichheit aller Elemente verwenden wir ein allgemeineres Abbruchkriterium, nämlich ob die Liste eine aufsteigende Kette bildet. Im endlichen Fall subsumiert dies auch mehrelementige Listen, die nicht weiter sortiert werden müssen, im zyklischen Fall sind die Elemente dann alle gleich. Eine unbeschränkt aufsteigende Kette verschiedener Elemente ist nur im wirklich Unendlichen möglich. Da der Test genau wie die Partitionierung linearen Aufwand hat, bleibt die Komplexitätsklasse des Verfahrens erhalten.

Wir definieren also als rekursive Funktion:

$$qsort(l) = \begin{cases} l & chain(\leq)(l) \\ qsort'(l) & \text{sonst} \end{cases}$$

$$qsort'(a:l) = l_< +\!\!+ l_= +\!\!+ l_> \qquad \begin{pmatrix} l_< = qsort\big(filter(<a)(l)\big) \\ l_= = a : filter(=a)(l) \\ l_> = qsort\big(filter(>a)(l)\big) \end{pmatrix}$$

Andere Sortierverfahren lassen sich nicht unmittelbar auf zyklische Listen verallgemeinern. Algorithmen, die auf dem Einfügen einzelner Elemente basieren, scheiden aus offensichtlichen Gründen aus. Bei anderen Verfahren zeigen sich die Probleme erst auf den zweiten Blick. Man betrachte das *Mergesort*-Verfahren:

$$\begin{aligned} msort([]) &= [] \\ msort(a:l) &= msort'\big(odd(a:l), even(a:l)\big) \\ msort'(l_1, l_2) &= merge\big(msort(l_1), msort(l_2)\big) \end{aligned}$$

Dabei ist die Aufteilung in Teillisten durch folgende abstrakten Interpretationen definiert:

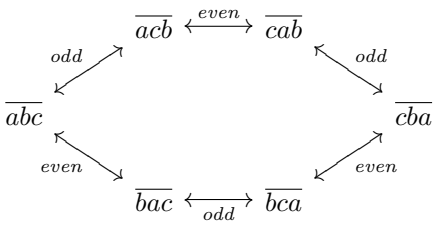
$$\begin{aligned} \mathsf{odd}([]) &= [] & \mathsf{even}([]) &= [] \\ \mathsf{odd}(a:[]) &= a : \underline{[]} & \mathsf{even}(a:[]) &= [] \\ \mathsf{odd}(a:b:l) &= a : \underline{l} & \mathsf{even}(a:b:l) &= b : \underline{l} \end{aligned}$$

Durch geschickte Ausnutzung der Aufrufkonventionen der virtuellen Maschine lassen sich beide Teillisten auch effizient simultan als $split = \langle odd, even \rangle$ berechnen. Das Zusammenfügen *merge* der sortierten Teillisten ist durch folgende abstrakte Interpretation definiert:

$$\begin{aligned} \mathsf{merge}([],[]) &= [] \\ \mathsf{merge}(a:l,[]) &= a : \underline{(l,[])} \\ \mathsf{merge}([],a:l) &= a : \underline{([],l)} \\ \mathsf{merge}(a_1:l_1, a_2:l_2) &= \begin{cases} a_1 : \underline{(l_1, a_2:l_2)} & a_1 \leq a_2 \\ a_2 : \underline{(a_1:l_1, l_2)} & a_1 > a_2 \end{cases} \end{aligned}$$

Die beiden Teiloperationen *split* und *merge* sind für sich genommen auch auf zyklischen Listen effektiv, nicht aber das rekursive Suchverfahren *msort*, da die Aufteilung zyklischer Listen im Allgemeinen

nicht zu einer Vereinfachung führt, wie an einer Ringliste von drei Elementen a, b, c gezeigt werden kann:



B.3.5 Basistransformation

Die Periodizität der *split*-Operation vereitelt zwar das von unten nach oben fortschreitende und damit inhärent nicht corekursive *Mergesort*-Verfahren, kann aber in anderen Kontexten durchaus für corekursive Berechnungen effektiv eingesetzt werden. Man betrachte dazu eine alternative Darstellungsform von Listen, egal ob initial oder final:

$$\textbf{type}\ \mathit{btree}(\alpha) = \{\mathsf{nil}, \mathsf{cons} : \mathit{bcell}(\alpha)\}$$
$$\textbf{type}\ \mathit{bcell}(\alpha) = (\mathsf{root} : \alpha, \mathsf{odd} : \mathit{btree}(\alpha), \mathsf{even} : \mathit{btree}(\alpha))$$

In dieser balancierten binären Baumdarstellung ist eine Liste entweder leer, oder sie verfügt über eine Wurzelzelle mit einem ersten Element und zwei auf die gleiche Weise repräsentierten Restlisten, auf welche die Elemente *abwechselnd* aufgeteilt sind. Es existieren effektive Übersetzungen zwischen beiden Darstellungen für rationale Listen, für die allerdings die Grenze der corekursiven Berechnung etwas über den streng primitiven Bereich hinausgeschoben werden muß.

Zur besseren Lesbarkeit der beiden Darstellungen schreiben wir $\triangle$ für den durch das Pattern nil beschriebenen leeren Baum, und $a \mathrel{.\!.} (l, r)$ für den durch das Pattern $\mathsf{cons}(\mathsf{root} : a, \mathsf{odd} : l, \mathsf{even} : r)$ beschriebenen nichtleeren Baum.

Die Übersetzung von der linearen in die baumförmige Darstellung ist durch folgende abstrakte Interpretation definiert:

$$\mathsf{balance}([]) = \triangle$$
$$\mathsf{balance}(a : l) = a \mathrel{.\!.} (\underline{\mathit{odd}(l)}, \underline{\mathit{even}(l)})$$

An den Vorkommen primitiv corekursiver Hilfsfunktionen in den unterstrichenen Argumenten kann bereits erkannt werden, daß diese Interpretation Quasizyklen erzeugt. Die Funktion *balance* kann daher nur mittels Zyklenerkennung modulo Bisimilarität (`quasi`) effektiv berechnet werden. Der Beweis der endlichen Rekursionstiefe ist etwa wie folgt:

1. Der Präfixanteil einer Liste l schrumpft beim Übergang zu $\mathit{odd}(l)$ oder $\mathit{even}(l)$. Daher terminiert das Verfahren für endliche Listen, und für zyklische Listen muß nur der Periodenanteil untersucht werden.

2. Für eine rein periodische Liste $l = \mathit{rep}(l_0)$ mit $|l_0| = n$ gilt, falls n durch 2 teilbar:

$$\mathit{odd}(\mathit{rep}(l_0)) = \mathit{rep}(\mathit{odd}(l_0))$$
$$\mathit{even}(\mathit{rep}(l_0)) = \mathit{rep}(\mathit{even}(l_0))$$

3. Im Basisfall ist n nicht durch 2 teilbar. Abstrahiere l zum zyklischen Ring $\mathcal{Z}/n\mathcal{Z}$. Dann entspricht *odd* der linearen Abbildung $(\_ \cdot 2)$ und *even* der affinen Abbildung $(\_ \cdot 2 + 1)$. Für *odd* gilt der Satz von Euler und Fermat:
$$\mathit{odd}^{\phi(n)} = \mathit{id}$$
mit der Anzahl der teilerfremden Vorgänger $\phi(n) < n$. Ein analoges, etwas komplexeres Argument gilt für *even*.

Die Rückübersetzung von der baumförmigen in die lineare Darstellung ist durch die folgende abstrakte Interpretation definiert:

$$\mathsf{serial}(\triangle) = []$$
$$\mathsf{serial}(a \mathrel{.\!.} (l, r)) = a : \underline{\mathit{bzip}(l, r)}$$

Diese stellt eine harmlose primitiv corekursive Funktion dar. Das gilt allerdings nicht für die Hilfsfunktion *bzip*, die hier daher durch syntaktische Rekursion dargestellt wird:

$$bzip(\triangle, s) = s$$
$$bzip\big(a \mathrel{\therefore} (l, r), s\big) = a : bzip\big(s, bzip(l, r)\big)$$

Diese Definition ähnelt nicht nur auf den ersten Blick der ACKERMANN-Funktion. Wie diese kann sie nicht ohne substantielle, hier nicht ausgeführte, Umcodierung in primitive Normalform gebracht werden. Da beide Gleichungen produktiv sind, kann die Funktion *bzip* dennoch mit den operationalen Mitteln der virtuellen Maschine effektiv berechnet werden, wenn auch nur mit Zyklenerkennung modulo Bisimilarität. Ferner ist *serial* redundant:

$$serial(s) = bzip(s, \triangle)$$

Anhang C

Polynomielle Subtypen

Anders als in den beiden vorhergehenden Kapiteln, die zyklische Datenstrukturen gewissermaßen axiomatisch eingeführt haben, soll hier eine zyklische Datenstruktur aus einem praktischen Problem hergeleitet werden. Die theoretische Behandlung dieses Problems wurde bereits in [61] veröffentlicht. Diese wird hier um implementierbare Spezifikationen ergänzt.

Zur besseren Lesbarkeit abstrakter Interpretationen werden eingebettete corekursive Argumente wie im letzten Kapitel unterstrichen.

C.1 Polynomielle Subtypbeziehungen

C.1.1 Typen

Das hier als Gegenstand betrachtete Typsystem ähnelt dem der Maschine V→M sehr stark. Es besteht aus

1. indizierten mehrstelligen Produkten und Coprodukten,
2. dem einfachen binären Funktionstypkonstruktor ($\_ \to \_$),
3. (Familien von) rekursiven Gleichungssystemen mit endlichen vielen Gleichungen.

Explizite Polymorphie betrachten wir der Einfachheit halber als Meta-Mechanismus, um eine parametrische Familie von Typdefinitionen simultan zu erzeugen. Typausdrücke und Typdefinitionen werden in folgender formalen Syntax denotiert:

$$\begin{array}{lcl} \textit{type-def} & \to & \textit{type-id}\ \underline{=}\ \textit{type} \\ \textit{type} & \to & \underline{(}\ \textit{type-row}\ \underline{)} \mid \underline{\{}\ \textit{type-row}\ \underline{\}} \mid \textit{type}\ \underline{->}\ \textit{type} \mid \textit{type-id} \mid \underline{(}\ \textit{type}\ \underline{)} \\ \textit{type-row} & \to & \left(\textit{index}\ \underline{:}\ \textit{type}\ \#\ \underline{,}\right)^* \end{array}$$

Dabei gelten einige Meta-Konventionen:

1. Typausdrücke werden geklammert, falls sonst Mehrdeutigkeit auftritt.
2. Alle Indizes einer Typfamilie (*type-row*) müssen verschieden sein. Die Aufzählung einer Typfamilie kann in beliebiger Reihenfolge erfolgen; Familien, die sich nur in der Permutation der Index-Typ-Paare unterscheiden, sollen als äquivalent betrachtet werden. Wir interpretieren die syntaktische Aufzählung:
$$[\mathsf{i}_1 : t_1, \ldots, \mathsf{i}_n : t_n] \longmapsto (t_i)_{i \in \{\mathsf{i}_1, \ldots, \mathsf{i}_n\}}$$

Beispiel C.1 (Listentyp). Hier der bereits hinlänglich diskutierte Listentyp als Familie rekursiver Gleichungssysteme:

$$\begin{aligned} \mathit{list}_\alpha &= \{\mathsf{nil} : (), \mathsf{cons} : \mathit{cell}_\alpha\} \\ \mathit{cell}_\alpha &= (\mathsf{car} : \alpha, \mathsf{cdr} : \mathit{list}_\alpha) \end{aligned}$$

Jede Belegung des Parameters α erzeugt ein Paar von Definitionsgleichungen. □

C.1.1.1 Namensauflösung

Bei der Definition der Typsprache fällt auf, daß Typgleichungen eigentlich nur zum Aufbau einer Coalgebra dienen. Auf die Oberflächensyntax folgt also zunächst, wie in Abschnitt 1.1.4 beschrieben, eine semi-abstrakte Welt von baumförmigen Typdefinitionstermen, die lediglich von Lexis und Klammerung abstrahieren. Im nächsten Abstraktionsschritt ergibt sich dann durch Auflösung symbolischer Namen (*type-id*) die abstrakte Welt der Typcoterme.

Praktisch wird diese Trennung in zwei Abstraktionsschritte durch zwei Typdefinitionen *texpr* und *tval* im Maschinenprogramm umgesetzt, die sich nur darin unterscheiden, daß der zweiten die Variante der symbolischen Referenz (*type* → *type-id*) fehlt:

$$\mathbf{type}\ \mathit{texpr}(\mathit{id}) = \left\{ \begin{array}{ll} \mathsf{product} & : \mathit{trow}(\mathit{texpr}(\mathit{id})), \\ \mathsf{coproduct} & : \mathit{trow}(\mathit{texpr}(\mathit{id})), \\ \mathsf{exponent} & : \mathit{tfun}(\mathit{texpr}(\mathit{id})), \\ \mathsf{indirect} & : \mathit{id} \end{array} \right\}$$

$$\textbf{type } \mathit{tval} = \left\{\begin{array}{ll} \mathsf{product} & : \mathit{trow}(\mathit{tval}), \\ \mathsf{coproduct} & : \mathit{trow}(\mathit{tval}), \\ \mathsf{exponent} & : \mathit{tfun}(\mathit{tval}) \end{array}\right\}$$

$$\textbf{type } \mathit{tfun}(t) = (\mathsf{domain} : t, \mathsf{codomain} : t)$$

Die erste, semi-abstrakte Variante ist zusätzlich polymorph über einem Namenstyp. Dabei werden Typfamilien als Assoziationslisten codiert:

$$\textbf{type } \mathit{trow}(t) = \mathit{alist}(\mathit{index}, t)$$
$$\textbf{type } \mathit{alist}(k, v) = \mathit{list}(\mathit{maplet}(k, v))$$
$$\textbf{type } \mathit{maplet}(k, v) = (\mathsf{key} : k, \mathsf{value} : v)$$

Eine solche Assoziationsliste schreiben wir kurz als $[\mathsf{i}_1 \mapsto t_1, \ldots, \mathsf{i}_n \mapsto t_n]$.

Der Typ *tval* ist eine Spezialisierung aller Instanzen von *texpr*. Würde man das hier zu entwickelnde Subtypsystem auf das implementierende System selbst anwenden, wäre es möglich ein Element von *tval* überall dort einzusetzen, wo ein Element von *texpr(id)* erwartet wird. Als vorläufige Simulation kann die corekursive Inklusionsfunktion $\mathit{tlift} : \mathit{tval} \to \mathit{texpr}(\mathit{id})$ dienen, die sich leicht als homomorphe abstrakte Interpretation implementieren läßt:

$$\begin{aligned} \mathsf{tlift}_0\big(\mathsf{product}[\mathsf{i}_1 \mapsto t_1, \ldots, \mathsf{i}_n \mapsto t_n]\big) &= \mathsf{product}[\mathsf{i}_1 \mapsto \underline{t_1}, \ldots, \mathsf{i}_n \mapsto \underline{t_n}] \\ \mathsf{tlift}_0\big(\mathsf{coproduct}[\mathsf{i}_1 \mapsto t_1, \ldots, \mathsf{i}_n \mapsto t_n]\big) &= \mathsf{coproduct}[\mathsf{i}_1 \mapsto \underline{t_1}, \ldots, \mathsf{i}_n \mapsto \underline{t_n}] \\ \mathsf{tlift}_0\big(\mathsf{exponent}(t, u)\big) &= \mathsf{exponent}(\underline{t}, \underline{u}) \end{aligned}$$

Umgekehrt kann ein semi-abstraktes Gleichungssystem, als Assoziationsliste Γ codiert, auch als zugrundeliegende Coalgebra verstanden werden, so daß jedem definierten Namen und damit jedem Typausdruck ein Coterm zugeordnet werden kann. Das leistet die inverse Funktion $\mathit{teval} : \mathit{alist}(\mathit{id}, \mathit{texpr}(\mathit{id})) \times \mathit{texpr}(\mathit{id}) \to \mathit{tval}$, die den zweiten Abstraktionsschritt implementiert, hier in CURRY-Form als Familie von abstrakten Interpretationen dargestellt:

$$\begin{aligned} \mathsf{teval}(\Gamma)_0\big(\mathsf{product}[\mathsf{i}_1 \mapsto t_1, \ldots, \mathsf{i}_n \mapsto t_n]\big) &= \mathsf{product}[\mathsf{i}_1 \mapsto \underline{t_1}, \ldots, \mathsf{i}_n \mapsto \underline{t_n}] \\ \mathsf{teval}(\Gamma)_0\big(\mathsf{coproduct}[\mathsf{i}_1 \mapsto t_1, \ldots, \mathsf{i}_n \mapsto t_n]\big) &= \mathsf{coproduct}[\mathsf{i}_1 \mapsto \underline{t_1}, \ldots, \mathsf{i}_n \mapsto \underline{t_n}] \\ \mathsf{teval}(\Gamma)_0\big(\mathsf{exponent}(t, u)\big) &= \mathsf{exponent}(\underline{t}, \underline{u}) \\ \mathsf{teval}(\Gamma)_0\big(\mathsf{indirect}(\mathsf{i}_k)\big) &= \mathsf{teval}(\Gamma)_0(t_k) \qquad \Gamma = [\mathsf{i}_1 \mapsto t_1, \ldots, \mathsf{i}_n \mapsto t_n] \end{aligned}$$

Falls die Rekursion in der letzten Zeile nicht terminiert, oder kein passendes k gefunden wird, ist das Gleichungssystem nicht wohlgeformt.

Im Folgenden verwenden wir statt der Oberflächensyntax mit den Operatoren (), {}, $\to$ und : eher die *tval*-Modellierung mit den Operatoren product, coproduct, exponent und $\mapsto$. Lediglich Elemente des Typs *tfun* werden statt $(\mathsf{domain} = t, \mathsf{codomain} = u)$ kurz (t, u) geschrieben.

C.1.2 Typschnittstellen

Um zu einer Definition der Subtyprelation zu kommen, müssen wir zunächst die beobachtbaren Eigenschaften eines Typs festlegen. Anders als etwa objektorientierte Sprachen, bei denen der Name, unter dem ein Typ definiert wird, Teil seiner Bedeutung ist, betrachten wir Typnamen als irrelevante Elemente einer Trägermenge, und beschränken uns auf deren finale Semantik. Diese wird adäquat durch die *induzierten Schnittstellen* der Typen beschrieben, wie sie in den meisten algebraisch orientierten Typsystemen Verwendung finden.

1. Ein Produkttyp $T = \mathsf{product}[\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n]$ induziert pro gültigem Index einen *Selektor*:

$$\sigma\llbracket T.\mathsf{i}_k \rrbracket : T \to T_k$$

2. Ein Coprodukttyp $T = \mathsf{coproduct}[\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n]$ induziert pro gültigem Index je einen *Diskriminator* und einen partiellen *Dekonstruktor*:

$$\tau[\![T.\mathsf{i}_k]\!] : T \to \{w, f\}$$
$$\delta[\![T.\mathsf{i}_k]\!] : T \nrightarrow T_k$$

Dabei gelten folgende Nebenbedingungen:

(a) Für jedes Element $x : T$ existiert genau ein Index i, so daß $\tau[\![T.i]\!](x) = w$. Dieses i schreiben wir als $\eta[\![T]\!](x)$.

(b) Der passende Dekonstruktionsausdruck $\delta[\![T.\eta[\![T]\!](x)]\!](x)$ ist wohldefiniert.

Die Schnittstelle für Coprodukttypen ist zwar operational vollständig, aber nicht für alle Anwendungen optimal. Insbesondere bei Fallunterscheidungen über die Varianten eines Coprodukts kann es inadäquat sein, alle Tests nacheinander auszuführen. In solchen Fällen ist eine alternative Schnittstelle vorzuziehen, nämlich die *elementare Fallunterscheidung*, die auch in der Kategorientheorie den universellen Morphismus des Coprodukts darstellt. Sei $T = \mathsf{coproduct}(\vec{r})$ der Typ des zu prüfenden Elementes und U der gemeinsame Typ aller Fälle:

$$\psi[\![T, U]\!] : \Psi[\![T, U]\!] \times T \to U$$
$$\vec{r} = [\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n] \implies \Psi[\![T, U]\!] = \mathsf{product}\big[\mathsf{i}_1 \mapsto (T_1 \to U), \ldots, \mathsf{i}_n \mapsto (T_n \to U)\big]$$

Dabei läßt sich ψ aus τ und δ oder umgekehrt über folgende Gleichung herleiten:

$$\psi[\![T, U]\!](C, x) = \sigma[\![\Psi[\![T, U]\!].i]\!](C)\big(\delta[\![T.i]\!](x)\big) \qquad \big(i = \eta[\![T]\!](x)\big)$$

Der Resultattyp U kann in einem Subtypsystem auch eine obere Schranke der Typen der Einzelfälle sein. Wo der Resultattyp irrelevant ist, wird er im Folgenden weggelassen, und $\psi[\![T]\!]$ als polymorphe Operation betrachtet.

Beispiel C.2 (Schnittstelle von Listen). Die induzierten Schnittstellen der oben definierten Listentypen:

$$\tau[\![\mathit{list}_\alpha.\mathsf{nil}]\!] : \mathit{list}_\alpha \to \{w, f\} \qquad \delta[\![\mathit{list}_\alpha.\mathsf{nil}]\!] : \mathit{list}_\alpha \nrightarrow ()$$
$$\tau[\![\mathit{list}_\alpha.\mathsf{cons}]\!] : \mathit{list}_\alpha \to \{w, f\} \quad \delta[\![\mathit{list}_\alpha.\mathsf{cons}]\!] : \mathit{list}_\alpha \nrightarrow \mathit{cell}_\alpha$$
$$\psi[\![\mathit{list}_\alpha, U]\!] : \begin{pmatrix} \mathsf{nil} & : & () \to U \\ \mathsf{cons} & : & \mathit{cell}_\alpha \to U \end{pmatrix} \to \mathit{list}_\alpha \to U$$
$$\sigma[\![\mathit{cell}_\alpha.\mathsf{car}]\!] : \mathit{cell}_\alpha \to \alpha$$
$$\sigma[\![\mathit{cell}_\alpha.\mathsf{cdr}]\!] : \mathit{cell}_\alpha \to \mathit{list}_\alpha$$

□

C.1.3 Subtypen

Mit Hilfe der induzierten Schnittstellen von Datentypen läßt sich eine strukturelle Subtyprelation definieren.

Definition C.3. Seien T, T' zwei Produkt- oder Coprodukttypen. Dann heißt T' ein *Subtyp* von T, geschrieben $T' \sqsubseteq T$, genau dann wenn die induzierte Schnittstelle von T durch die induzierte Schnittstelle von T' ohne Umbenennung simuliert werden kann. Auf die Typen der induzierten Operationen ist das Subtypkriterium corekursiv anzuwenden und der größte Fixpunkt zu bilden.

Das bedeutet im Einzelnen:

1. Man betrachte die Produkttypen $T = \mathsf{product}(\vec{r})$ und $T' = \mathsf{product}(\vec{r'})$, sowie die zugehörigen Typfamilien $(t_i)_{i\in I} = f(\vec{r})$ und $(t'_i)_{i\in I'} = f(\vec{r'})$. Dann gilt $T' \sqsubseteq T$, genau dann wenn für jeden Index $i \in I$ auch t'_i definiert ist und $t'_i \sqsubseteq t_i$ gilt. Dann wird jeder Selektor $\sigma[\![T.i]\!]$ durch sein Pendant $\sigma[\![T'.i]\!]$ simuliert.

   Um einen Produkttyp zu einem Subtyp zu spezialisieren, kann also entweder ein Komponententyp spezialisiert, oder eine weitere Komponente hinzugefügt werden.

2. Man betrachte die Coprodukttypen $T = \mathsf{coproduct}(\vec{r})$ und $T' = \mathsf{coproduct}(\vec{r'})$, sowie die zugehörigen Typfamilien $(t_i)_{i\in I} = f(\vec{r})$ und $(t'_i)_{i\in I'} = f(\vec{r'})$. Dann gilt $T' \sqsubseteq T$, falls für jedes $i \in I'$ auch t_i definiert ist und $t'_i \sqsubseteq t_i$ gilt. Dann wird jeder Test $\tau[\![T.i]\!]$ und jeder Dekonstruktor $\delta[\![T.i]\!]$ durch sein Pendant $\tau[\![T'.i]\!]$ beziehungsweise $\delta[\![T'.i]\!]$ simuliert. Für die übrigen Indizes $j \in I \setminus I'$ werden Test hinzugefügt, die stets f liefern, und Dekonstruktoren, die stets undefiniert sind.

   Um einen Coprodukttyp zu einem Subtyp zu spezialisieren, kann also entweder ein Variantentyp spezialisiert, oder eine Variante ausgeschlossen werden.

3. Die Subtyprelation wird in der üblichen Weise, links kontravariant und rechts kovariant auf Funktionstypen fortgesetzt:

$$T' \sqsubseteq T \wedge U' \sqsubseteq U \iff \mathsf{exponent}(T, U') \sqsubseteq \mathsf{exponent}(T', U)$$

   Eine Funktion von Typ $T \to U'$ kann das Schnittstellenverhalten einer Funktion vom Typ $T' \to U$ simulieren, da sie von ihrem Argument die Schnittstelle von T erwartet, die durch die Schnittstelle von T' simuliert wird, während ihr Ergebnis die Schnittstelle von U' anbietet, welche die von U simuliert.

Das Ergebnis ist tatsächlich eine partielle Ordnung. Der Nachweis hierfür ergibt sich konstruktiv aus dem weiteren Vorgehen: Den einzelnen Paaren $T' \sqsubseteq T$ der Subtyprelation werden *Beweise* zugeordnet, indem eine Konstruktion angegeben wird, die Paare von Typcotermen auf einen eindeutigen Beweiscoterm abbildet, falls die Subtypbeziehung gilt. Anschließend wird gezeigt, daß die Typen als Objekte und die Subtypbeweise als Morphismen eine Kategorie bilden. Damit ist die Reflexivität der Subtyprelation äquivalent zur Existenz des identischen Morphismus, und die Transitivität äquivalent zur Existenz der Komposition. Auch für Identität und Komposition werden Berechnungsverfahren angegeben. Damit hat die Beweisführung nicht nur theoretischen Selbstzweck, sondern ist zugleich prototypische Implementierung.

C.1.4 Das Problem

Ein Typsystem zu implementieren heißt im Wesentlichen, die induzierten Schnittstellenoperationen zu implementieren, also ein universelles Codierungsschema für alle definierbaren Typen anzugeben.

Definition C.4 (Effiziente Codierung)**.** Ein Codierungsschema für ein Typsystem heiße *effizient*, falls alle induzierten Operationen höchstens *konstanten* Zeitaufwand besitzen:

1. Selektoren,
2. Diskriminatoren, Dekonstruktoren und Fallunterscheidungen
3. Typkonversion vom Sub- zum Supertyp (*upcast*).

Gelten nur die ersten beiden Klauseln, heißt das System *lokal effizient*. □

In einem Subtypsystem ist es zulässig, eine Variable $x : T?$ an einen Wert $a : T!$ zu binden, genau dann wenn $T! \sqsubseteq T?$ gilt. Ist die Variable x λ-quantifiziert, dann kann nicht für alle Inkarnationen der Typ $T!$ statisch vorhergesagt werden. Die Abbildung der formalen Schnittstellenoperationen von x auf die aktuellen von a kann auf verschiedene Weise erfolgen:

1. Das universelle Codierungsschema wird so gewählt, daß die Schnittstellenoperationen eines Supertyps stets auch für den Subtyp gültig sind. Im Falle von Produkten hat das weitreichende Konsequenzen, da der Subtyp unbeschränkt viele Komponenten hinzufügen kann. Ein universeller Selektor ist daher zwingend auf Suche angewiesen, und hat damit bei optimaler Implementierung (Suchbäume) logarithmischen Aufwand, bei anderen typischen Implementierungen (Assoziationslisten, Hashtabellen) sogar linearen. Eine universelle Codierung kann also nicht effizient im obigen Sinn sein.

2. Es wird pro Typ ein Codierungsschema gewählt, das nur von den vorkommenden Indizes abhängt. Damit kann stets ein maschinennahes, lokal effizientes Schema gefunden werden. Die Codierung von Sub- und Supertyp sind aber im Allgemeinen inkompatibel. Dies muß zur Laufzeit kompensiert werden, indem der Bindung $x := a$ dynamische Typinformation verliehen wird, die es gestattet, Operationen von $T?$ dynamisch auf die von $T!$ abzubilden.

C.2 Lösung durch Typlogik

Typtheoretische Arbeiten, insbesondere von CARDELLI [8] haben gezeigt, daß die Subtyprelation auf ein Unifikationsproblem auf Typfamilienausdrücken (*type rows*) zurückgeführt werden kann. Man erweitert die Typlogik um Variablen und Konstruktoren für Typfamilien. Sei beispielsweise $T = (\mathsf{i}_1 : T_1, \mathsf{i}_2 : T_2)$, dann ist jeder Subtyp $T' \sqsubseteq T$ durch einen Ausdruck der Form $T' = (\mathsf{i}_1 : T_1, \mathsf{i}_2 : T_2 \mid \rho)$ darstellbar, wobei ρ eine Typfamilienvariable ist. Dies kann durch Unifikation (modulo Permutation) entschieden werden.

Implementierungen dieses Ansatzes existieren für die gängigsten funktionalen Sprachen, nämlich ML [43] und Haskell [23]. Diese unterscheiden sich erheblich in der Darstellung: erstere Lösung hat die Form einer Programmtransformation, letztere die einer Typklassenbiliothek. Das Grundprinzip ist aber in beiden Fällen der oben bereits vorweggenommene Satz: Die Anwendung subtyp-polymorpher Operationen auf statisch getypte Ausdrücke erfordert zusätzliche, dynamische Typinformation.

Die dynamische Typinformation läßt sich als konstruktiver Beweis der Subtypbeziehung betrachten: Wenn der aktuelle Typ $T!$ des Wertes a tatsächlich die Schnittstelle des formalen Typs $T?$ der Variablen x simulieren kann, dann kann bei der Bindung von x an a die Zuordnung der Schnittstellenoperationen in geeigneter Codierung mit übergeben werden.

Beispiel C.5. Sei $g : U \to V$ eine beliebige Funktion. Man definiere eine Funktion $f : T \to V$, die g auf die Komponente foo eines beliebigen passenden Tupels anwendet:

$$f(x) = g\big(\sigma[\![T.\mathsf{foo}]\!](x)\big)$$

Der allgemeinste gültige Lösung für T ist $\forall\rho.\ (\mathsf{foo} : U \mid \rho)$. Falls verschiedene Belegungen von ρ verschiedene Selektoren $\sigma[\![T.\mathsf{foo}]\!]$ ergeben, muß der aktuell gültige beim Aufruf der Funktion f übergeben werden:

$$f(s, x) = g\big(s(x)\big)$$

Durch die Wahl einer geeigneten Codierung läßt sich die Auswahl des Selektors auf einen primitiven (numerischen) Parameter reduzieren:

$$f(k, x) = g\big(\mathfrak{s}(k, x)\big)$$

wobei $\mathfrak{s}$ ein generischer Selektor ist. Eine solche Codierung wird auch Gegenstand der folgenden Abschnitte sein. □

Die Verknüpfung der dynamischen Typinformation mit der herkömmlichen Typprüfung ist semantisch elegant und hat den positiven Nebeneffekt, daß nur tatsächlich benötigte Information eingefügt wird. Die entscheidende Schwäche des Ansatzes liegt in der rekursiven Typkonvertierung. Genauer gesagt, es findet eigentlich keine Typkonvertierung statt. Man betrachte zwei verschiedene rekursive Datentypen $T \sqsubseteq T'$. Wird ein Element $a : T'$ an eine rekursive Funktion $f : T \to U$ übergeben, dann werden zwar die Schnittstellenoperationen von T auf die von T' abgebildet, aber Teilausdrücke von a sind nach wie vor vom Typ T'. Die dynamische Typinformation muß also überall dort propagiert werden, wo Teile von Subtypelementen weiterverarbeitet werden. Um ein Element wirklich von einem Sub- in einen Supertyp zu konvertieren, muß es mit der (Subtyp-polymorphen) Identitätsfunktion des Supertyps vollständig traversiert und kopiert werden. Also kann dieser Ansatz höchstens lokal effizient sein. Im folgenden Abschnitt soll eine alternative, global effiziente Technik entwickelt werden.

C.3 Lösung durch Codierung

C.3.1 Kompakte Codierung

Für die lokal effiziente Codierung beliebiger Produkte und Coprodukte muß auf die Fähigkeiten der ausführenden Maschinen Rücksicht genommen werden. Dabei unterscheiden sich die Codierungen maschinennaher Sprachen wie C oder ASN.1[3] nicht wesentlich von den oben zitierten Lösungen für funktionale Sprachen: Die Menge vorkommender Indizes wird auf einen endlichen Zahlbereich, typischerweise ein Maschinenwort, abgebildet:

1. Ein Tupel wird durch einen zusammenhängenden Speicherbereich dargestellt. Jeder Selektor ist durch den Adressabstand zwischen dem Anfang des Tupels und seiner entsprechenden Komponente festgelegt. Prozessoren unterstützen die Selektion in konstanter Zeit durch *Adressarithmetik*. Wir schreiben $\mathfrak{s}(k, a)$ für die generische Selektion der k-ten Komponente aus dem Tupel a.

2. Ein Cotupel wird durch ein Paar von numerisch codiertem Index und Rumpf dargestellt. Tests lassen sich als Zahlenvergleich implementieren, Fallunterscheidungen durch Kaskaden bedingter Sprünge oder Sprungtabellen. Wir schreiben $\mathfrak{e}(a)$ für die Indexnummer und $\mathfrak{d}(a)$ für den Rumpf eines Cotupels a.

Um die im Allgemeinen abzählbar unendliche Menge der Indizes auf einen endlichen Zahlbereich zu falten, sind verschiedene Techniken denkbar:

1. Rémy untersucht in [54] den Einsatz von Hashfunktionen. Dieser Ansatz ist praktikabel, aber notwendigerweise heuristisch: Gelingt es, eine perfekte Hashfunktion zu finden, dann sind die Schittstellen von Sub- und Supertypen unmittelbar kompatibel, und Typkonversion hat gar keinen Aufwand. Gelingt es aber nicht, muß auf suboptimale Codierungen ausgewichen werden. Durch Hinzunahme eines weiteren Typs zu einem Programm kann eine bestehende Hashfunktion allerdings unbrauchbar werden.

2. Hier soll ein alternativer Ansatz verfolgt werden, endliche Indexmengen auf geschlossene Intervalle $[0, \ldots, n-1]$ abzubilden. Dadurch wird die Repräsentation von Subtypbeweisen als Datenobjekte vereinfacht. Auch ist das Verfahren robust gegenüber zusätzlichen Typdefinitionen. Allerdings haben Sub- und Supertyp im Allgemeinen inkompatible Schnittstellen, was bei Typkonversion zur Laufzeit ausgeglichen werden muß.

C.3.1.1 Familienkalkül

Um von der mengentheoretischen Darstellung von Typfamilien zu einer praktischen zu gelangen, setzen wir eine feste, totale Ordnung $\prec$ auf den Indizes voraus. Dann lassen sich Typfamilien eindeutig als Assoziationslisten darstellen.

Definition C.6. Sei $\vec{r} = [\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n]$ die Darstellung einer Typfamilie. Dabei seien die Indizes streng aufsteigend geordnet: $\mathsf{i}_1 \prec \cdots \prec \mathsf{i}_n$. Wir schreiben:

$$\vec{r} \sharp \mathsf{i}_k = k \qquad \vec{r} @ k = \mathsf{i}_k$$
$$|\vec{r}| = n \qquad \vec{r} \bullet k = T_k$$

□

Lemma C.7. *Es gelten folgende Gesetze:*

$$\vec{r} @ \vec{r} \sharp i = i \qquad \sigma[\![\mathsf{product}(\vec{r}).i]\!] : \mathsf{product}(\vec{r}) \to \vec{r} \bullet \vec{r} \sharp i$$
$$\vec{r} \sharp \vec{r} @ k = k \qquad \delta[\![\mathsf{coproduct}(\vec{r}).i]\!] : \mathsf{coproduct}(\vec{r}) \nrightarrow \vec{r} \bullet \vec{r} \sharp i$$

C.3.1.2 Naives Codierungsschema

Mit Hilfe der Operationen auf Typfamilien läßt sich eine erste Näherung einer effizienten Codierung als Abbildung von typisierten auf generische Operationen definieren:

$$\sigma\llbracket\mathsf{product}(\vec{r}).i\rrbracket(a) \longmapsto \mathfrak{s}(\vec{r}\sharp i, a)$$
$$\psi\llbracket\mathsf{coproduct}(\vec{r})\rrbracket(C, a) \longmapsto \mathfrak{s}\big(\mathfrak{e}(a), C\big)\big(\mathfrak{d}(a)\big)$$

Die übrigen Operationen τ, δ und η lassen sich nach den bereits bekannten Gesetzen ableiten.

Dieses Codierungsschema ist zwar lokal effizient, aber im Allgemeinen sind die Schnittstellen von Sub- und Supertyp nicht kompatibel, da Indizes in verschiedenen Typausdrücken auf verschiedene Zahlen abgebildet werden können. Dynamische Typinformation besteht also im Wesentlichen aus der Übersetzung zwischen diesen Numerierungen. Die folgenden Abschnitte werden sich mit der Repräsentation und Verwaltung der dynamischen Typinformation beschäftigen.

C.3.1.3 Verwaltung von Typinformation

Im Extremfall nehme man an, daß jede Variablenbindung tatsächlich mit dynamischer Typinformation ausgestattet ist. Dabei ist der exakte Werttyp $T!$ allerdings nur in wenigen Situationen statisch bekannt, zum Beispiel wenn der Wert eine Konstruktorapplikation ist. Nur dann kann dynamische Typinformation bereits statisch festgelegt werden, andernfalls ist es nötig, sie dynamisch zu berechnen. Wie dies geschieht und wie sich der Aufwand auf ein vertretbares Maß beschränken läßt, wird in den nächsten Abschnitten erläutert. Die Frage, welche Bindungen auch ohne dynamische Typinformation auskommen können, ist Gegenstand von Optimierung, und wird hier nicht behandelt.

C.3.2 Subtypbeweise

Die Struktur der Subtypbeweise folgt der Struktur der Typen selbst. Die folgende Definition postuliert, daß eine natürliche Zahl (gern als magische Zahl bezeichnet) pro Element einer Typfamilie als lokale Information genügt. Den Nachweis werden wir später antreten. Wegen der starken Analogie zu den traditionellen Implementierungstechniken für objektorientierte Subtypsysteme bezeichnen wir diese Datenstrukturen auch als *vtables*.

$$\textbf{type}\ \mathit{tinfo} = \left\{\begin{array}{ll}\mathsf{product} & : \mathit{vtab},\\ \mathsf{coproduct} & : \mathit{vtab},\\ \mathsf{exponent} & : \mathit{vfun}\end{array}\right\}$$
$$\textbf{type}\ \mathit{vfun} = (\mathsf{domain} : \mathit{tinfo}, \mathsf{codomain} : \mathit{tinfo})$$
$$\textbf{type}\ \mathit{vtab} = \mathit{alist}(\mathit{nat}, \mathit{tinfo})$$

Definition C.8. Sei $\vec{v} = [m_1 \mapsto v_1, \ldots, m_n \mapsto v_n]$ die *vtable* eines Subtypbeweises. In Analogie zu den Typfamilien schreiben wir:

$$\vec{v}\,@\,k = m_k \qquad\qquad \vec{v}\bullet k = v_k$$

Zu einer gegebenen *vtable* $\vec{v}$ definiert die Operation $\vec{v}\,@\,\_$ eine Umcodierung von Indexnummern. Im folgenden Abschnitt soll gezeigt werden, wie mit deren Hilfe die Subtypkompatibilität wiederhergestellt werden kann.

C.3.2.1 Konstruktion

Die Konstruktion von Subtypbeweisen zu gegebenen Typen ist algorithmisch beinahe trivial, da die Strukturierung implizt durch Corekursion geschieht, und lediglich die lokalen Informationen, sprich die erwähnten magischen Zahlen, explizit berechnet werden müssen. Dazu dienen die Operationen des Typfamilienkalküls. Wir definieren eine zweistellige abstrakte Interpretation und unterstreichen wieder die Rekursionsstellen:

1. Im Fall von Produkten muß die Selektion auf dem Supertyp auf die Codierung eines Tupels vom Subtyp und den Subtypbeweis der Komponententypen abgebildet werden:

$$\mathsf{tsub}_0\big(\mathsf{product}(\vec{r}), \mathsf{product}[\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n]\big) = \mathsf{product}[m_1 \mapsto \underline{x_1}, \ldots m_n \mapsto \underline{x_n}]$$
$$\text{wobei } \begin{cases} m_k = \vec{r} \sharp \mathsf{i}_k \\ x_k = (\vec{r} \bullet m_k, T_k) \end{cases}$$

2. Im Fall von Coprodukten muß umgekehrt die Codierung eines Cotupels vom Subtyp auf den beobachteten Index (Diskriminator) im Supertyp und den Subtypbeweis der Komponententypen (Dekonstruktor) abgebildet werden:

$$\mathsf{tsub}_0\big(\mathsf{coproduct}[\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n], \mathsf{coproduct}(\vec{r})\big) = \mathsf{coproduct}[m_1 \mapsto \underline{x_1}, \ldots m_n \mapsto \underline{x_n}]$$
$$\text{wobei } \begin{cases} m_k = \vec{r} \sharp \mathsf{i}_1 \\ x_k = (T_k, \vec{r} \bullet m_k) \end{cases}$$

3. Im Fall von Funktionstypen wird wie üblich in die kontravariante Argument- und die kovariante Resultatseite aufgespalten:

$$\mathsf{tsub}_0\big(\mathsf{exponent}(T, U'), \mathsf{exponent}(T', U)\big) = \mathsf{exponent}\big(\underline{(T', T)}, \underline{(U', U)}\big)$$

Streng genommen handelt es sich bei den konkreten Ergebnissen der Funktion *tsub* um Coterme für Subtypbeweise. Deren finale Semantiken bezeichnen wir als die eigentlichen Subtypbeweise. Für Subtypcoterme s aus der Bisimilaritätsklasse von $tsub(T', T)$ schreiben wir $s : T' \to T$.

Nun läßt sich auch Definition C.3 modellieren. Neben der exakteren Spezifikation des Subtypbegriffs auf der Metaebene wird damit auch ein entscheidbares Kriterium für die Definiertheit der Funktion *tsub* geschaffen:

$$\frac{\vec{r} \bullet \vec{r} \sharp \mathsf{i}_1 \sqsubseteq T_1 \quad \cdots \quad \vec{r} \bullet \vec{r} \sharp \mathsf{i}_n \sqsubseteq T_n}{\mathsf{product}(\vec{r}) \sqsubseteq \mathsf{product}[\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n]} \qquad \frac{T_1 \sqsubseteq \vec{r} \bullet \vec{r} \sharp \mathsf{i}_1 \quad \cdots \quad T_n \sqsubseteq \vec{r} \bullet \vec{r} \sharp \mathsf{i}_n}{\mathsf{coproduct}[\mathsf{i}_1 \mapsto T_1, \ldots, \mathsf{i}_n \mapsto T_n] \sqsubseteq \mathsf{coproduct}(\vec{r})}$$

$$\frac{T' \sqsubseteq T \quad U' \sqsubseteq U}{\mathsf{exponent}(T, U') \sqsubseteq \mathsf{exponent}(T', U)}$$

Die Produkt- und Coproduktregeln sollen nur dann gelten, wenn alle $\vec{r} \sharp \mathsf{i}_k$ definiert sind. Die Subtypbeziehung ist der größte Fixpunkt dieses Kalküls.

C.3.2.2 Dynamisches Codierungsschema

Ist für eine Variablenbindung die passende Subtypinformation in der eben definierten Form verfügbar, dann lassen sich die spezifischen Operationen des Variablentyps auf generische Operationen auf dem Wert abbilden, bei denen auch die Typinformation erhalten bleibt. In diesem Ansatz sind also typisierte Operationen definiert auf Paaren von Wert und Typinformation, und liefern ebensolche Paare.

$$\sigma\,[\![\mathsf{product}(\vec{r}).i]\!]\,\big(a :: \mathsf{product}(\vec{v})\big) \longmapsto \mathfrak{s}(\vec{v} \mathbin{@} k, a) :: (\vec{v} \bullet k) \qquad k = \vec{r} \sharp i$$
$$\psi\,[\![\mathsf{coproduct}(\vec{r})]\!]\,\big(a :: \mathsf{coproduct}(\vec{v})\big) \longmapsto \mathfrak{s}\big(\vec{v} \mathbin{@} k, C\big)\big(\mathfrak{d}(a)\big) :: (\vec{v} \bullet k) \qquad k = \mathfrak{e}(a)$$

Für dieses Schema läßt sich nun die Korrektheit der Subtypbeweise zeigen.

Satz C.9. *Sei $T!$ ein Produkt- oder Summentyp mit der Komponentenfamilie $\vec{r}$. Sei $T?$ ein Supertyp von $T!$. Dann liefern die induzierten Schnittstellenoperationen auf $T?$, angewandt auf ein Paar $a :: t$ aus eine, Wert a vom Typ $T!$ und einem Subtypbeweis $t : T! \to T?$, ein Paar $b :: u$, so daß gilt:*

1. *Der Ergebniswert b ist derselbe wie bei statisch typisiertem Zugriff auf a über $T!$.*
2. *Sei $U!$ der tatsächliche Ergebnistyp laut $T!$ und $U?$ der erwartete Ergebnistyp laut $T?$. Dann ist u der zugehörige Subtypbeweis $u : U! \to U?$.*

Beweis. Sei $T! = \mathsf{product}(\vec{r})$, $T? = \mathsf{product}(\vec{r}')$ und $t = tsub(T!, T?) = \mathsf{product}(\vec{v})$. Sei $\vec{r} = [\mathsf{i}_1 \mapsto U_1, \ldots, \mathsf{i}_n \mapsto U_n]$ und $\vec{r}' = [\mathsf{i}'_1 \mapsto U'_1, \ldots, \mathsf{i}'_m \mapsto U'_m]$. Seien außerdem j beliebig mit $1 \leq j \leq n$ und k passend, so daß $\mathsf{i}_j = \mathsf{i}'_k$.

$$\sigma[\![T!.\mathsf{i}_j]\!](a) = \mathfrak{s}(j, a)$$

$$\begin{aligned}
\sigma[\![T?.\mathsf{i}'_k]\!](a :: t) &= \mathfrak{s}(\vec{v} \mathbin{@} k, a) :: (\vec{v} \bullet k) \\
&= \mathfrak{s}(\vec{r} \sharp \mathsf{i}'_k, a) :: tsub(\vec{r} \bullet \vec{r} \sharp \mathsf{i}'_k, U'_k) \\
&= \mathfrak{s}(\vec{r} \sharp \mathsf{i}_j, a) :: tsub(\vec{r} \bullet \vec{r} \sharp \mathsf{i}_j, U'_k) \\
&= \mathfrak{s}(j, a) :: tsub(\vec{r} \bullet j, U'_k) \\
&= \mathfrak{s}(j, a) :: tsub(U_j, U'_k)
\end{aligned}$$

Für Coprodukttypen und ψ dual. □

C.4 Praktische Handhabung

C.4.1 Typkategorie

Es kann gezeigt werden, daß Typen und Subtypbeziehungen eine Kategorie bilden:

1. Typen spielen die Rolle von Objekten.
2. Subtypbeweise sind die Morphismen.
3. Aus der Reflexivität der Subtyprelation folgt die Existenz der Identität.
4. Aus der Transitivität der Subtyprelation folgt die Existenz der Komposition.
5. Aus der Eindeutigkeit des Morphismus zwischen zwei gegebenen Objekten folgten die Neutralität der Identität und die Assoziativität der Komposition.

Ferner erzeugt jede monotone Typkonstruktion einen Funktor auf dieser Kategorie. Da die Typkonstruktoren Produkt und Coprodukt vollständig und der Funktionspfeil rechtsseitig monoton sind, fallen darunter auch alle benutzerdefinierbaren Typkonstruktoren, die als Funktionen erster Ordnung über jenen dargestellt sind.

Wie die folgenden Abschnitte zeigen werden, sind die kategoriellen Operationen genau die in der Praxis relevanten, so daß die kategorientheoretische Darstellung hier keinen mathematischen Selbstzweck, sondern das adäquate Paradigma darstellt.

C.4.1.1 Identität

Der identische Subtypbeweis $\mathrm{id} : T \to T$ läßt sich als Speziallfall der Konstruktion *tsub* definieren. Die besondere Form dieser Morphismen erfüllt folgende Gleichungen:

1. Sei $T = c(\vec{r})$, wobei $c = \mathsf{product}, \mathsf{coproduct}$ und $n = |\vec{r}|$. Dann gilt

$$\mathrm{id}_T = c[1 \mapsto \mathrm{id}_{\vec{r}\bullet 1}, \ldots, n \mapsto \mathrm{id}_{\vec{r}\bullet n}]$$

2. Sei $T = \mathsf{exponent}(S, U)$. Dann gilt

$$\mathrm{id}_T = \mathrm{exponent}(\mathrm{id}_S, \mathrm{id}_U)$$

Für nicht-freie Typen, wie etwa primitive Zahlbereiche, die außerhalb der Subtyprelation stehen, genügt eine zusätzliche Konstante für den identischen Subtypbeweis.

C.4.1.2 Komposition

Die Komposition zweier Subtypbeweise, im Prinzip einer Matrixmultiplikation nicht unähnlich, ist durch folgende abstrakte Interpretation definiert:

$$\mathsf{comp}_0\big(\mathsf{product}[m_1 \mapsto t_1, \ldots, m_n \mapsto t_n], \mathsf{product}(\vec{r})\big) = \mathsf{product}[m'_1 \mapsto t'_1, \ldots, m'_n \mapsto t'_n]$$
$$\text{wobei } \begin{cases} m'_i = \vec{r} \mathbin{@} m_i \\ t'_i = \underline{(t_i, \vec{r} \bullet m_i)} \end{cases}$$
$$\mathsf{comp}_0\big(\mathsf{coproduct}(\vec{r}), \mathsf{coproduct}[m_1 \mapsto t_1, \ldots, m_n \mapsto t_n]\big) = \mathsf{coproduct}[m'_1 \mapsto t'_1, \ldots, m'_n \mapsto t'_n]$$
$$\text{wobei } \begin{cases} m'_i = \vec{r} \mathbin{@} m_i \\ t'_i = \underline{(\vec{r} \bullet m_i, t_i)} \end{cases}$$
$$\mathsf{comp}_0\big(\mathsf{exponent}(t, u), \mathsf{exponent}(v, w)\big) = \mathsf{exponent}\big(\underline{(v, t)}, \underline{(u, w)}\big)$$

Satz C.10. *Die finalen Semantiken von Subtypbeweis und Komposition sind verträglich:*

$$comp\big(tsub(T,U), tsub(S,T)\big) = tsub(S,U)$$

Beweis. Durch strukturelle Coinduktion, also durch Konstruktion einer Bisimulation. Dazu wird zunächst ein geeigneter Signaturfunktor Σ benötigt:

$$\Sigma = \underbrace{(\mathcal{C}_I \times \mathcal{I})^*}_{\text{product}} + \underbrace{(\mathcal{C}_I \times \mathcal{I})^*}_{\text{coproduct}} + \underbrace{\mathcal{I} \times \mathcal{I}}_{\text{exponent}}$$

Sei nun $\mathbb{F} = (F, \phi)$ eine finale Σ-Coalgebra. Dann konstruieren wir die gesuchte Bisimulation $\mathbb{Y} = \big(Y, \upsilon(\Sigma)\big)\big)$ auf $\mathbb{F}$, mit der generischen Operation υ und folgender Trägermenge:

$$Y = \Big\{ \big(comp\big(tsub(T,U), tsub(S,T)\big), tsub(S,U)\big) \;\Big|\; S \sqsubseteq T \sqsubseteq U \Big\} \subseteq F \times F$$

Nach Lemma 2.101 genügt es zu zeigen, daß $\mathbb{Y}$ eine Σ-Coalgebra ist. Das heißt:

1. $\upsilon(\Sigma)$ ist auf ganz Y definiert. Nach Satz 2.98 ist hinreichend, daß Y nur kompatible Paare enthält.
2. Alle Nachfolger liegen ebenfalls in der Trägermenge: $y \in Y \implies \mathfrak{s}(\Sigma)\big(\upsilon(\Sigma)(y)\big) \subseteq Y$.

Beide Aussagen lassen sich am ökonomischsten simultan beweisen. Seien $S, T, U \in F$ Typausdrücke, so daß gilt

$$a = comp\big(tsub(T,U), tsub(S,T)\big) \qquad\qquad b = tsub(S,U)$$

Fallunterscheidung nach Typkonstruktoren.

1. Produktfall. Wir zerlegen:

   $$S = \mathsf{product}(\vec{s}) \qquad T = \mathsf{product}(\vec{t}) \qquad U = \mathsf{product}(\vec{u})$$

   Dann gilt:

$$\begin{aligned}
a &= comp\big(tsub(T,U), tsub(S,T)\big) \\
&= comp\big(\mathsf{product}(\vec{w}), \mathsf{product}(\vec{v})\big) \\
&\qquad \text{wobei} \begin{cases} |\vec{v}| = |\vec{t}| & |\vec{w}| = |\vec{u}| \\ \vec{v} \mathbin{@} k = \vec{s} \sharp \vec{t} \mathbin{@} k & \vec{w} \mathbin{@} k = \vec{t} \sharp \vec{u} \mathbin{@} k \\ \vec{v} \bullet k = tsub(\vec{s} \bullet \vec{v} \mathbin{@} k, \vec{t} \bullet k) & \vec{w} \bullet k = tsub(\vec{t} \bullet \vec{w} \mathbin{@} k, \vec{u} \bullet k) \end{cases} \\
&= \mathsf{product}(\vec{y}) \\
&\qquad \text{wobei} \begin{cases} |\vec{y}| = |\vec{w}| \\ \vec{y} \mathbin{@} k = \vec{v} \mathbin{@} \vec{w} \mathbin{@} k \\ \qquad = \vec{s} \sharp \vec{t} \mathbin{@} \vec{w} \mathbin{@} k \\ \qquad = \vec{s} \sharp \vec{t} \mathbin{@} \vec{t} \sharp \vec{u} \mathbin{@} k \\ \qquad = \vec{s} \sharp \vec{u} \mathbin{@} k \\ \vec{y} \bullet k = comp(\vec{w} \bullet k, \vec{v} \bullet \vec{w} \mathbin{@} k) \end{cases} \\
b &= tsub(S,U) \\
&= \mathsf{product}(\vec{x}) \\
&\qquad \text{wobei} \begin{cases} |\vec{x}| = |\vec{u}| \\ \vec{x} \mathbin{@} k = \vec{s} \sharp \vec{u} \mathbin{@} k \\ \qquad = \vec{y} \mathbin{@} k \\ \vec{x} \bullet k = tsub(\vec{s} \bullet \vec{x} \mathbin{@} k, \vec{u} \bullet k) \end{cases}
\end{aligned}$$

Da $\vec{x}$ und $\vec{y}$ in Anzahl der Elemente und Indizes übereinstimmen, sind a und b kompatibel. Alle direkten Nachfolger haben die Form

$$\big(comp\big(tsub(\underbrace{\vec{t} \bullet \vec{w} \mathbin{@} k}_{T'}, \underbrace{\vec{u} \bullet k}_{U'}), tsub(\underbrace{\vec{s} \bullet \overbrace{\vec{v} \mathbin{@} \vec{w} \mathbin{@} k}^{\vec{y} @ k}}_{S'}, \underbrace{t \bullet \vec{w} \mathbin{@} k}_{T'})\big), tsub(\underbrace{\vec{s} \bullet \vec{x} \mathbin{@} k}_{S'}, \underbrace{\vec{u} \bullet k}_{U'})\big)$$

und sind damit Elemente von Y.

2. Coproduktfall analog.

3. Im Exponentenfall zerlegen wir:

$$S = \mathsf{exponent}(S', S'') \qquad T = \mathsf{exponent}(T', T'') \qquad U = \mathsf{exponent}(U', U'')$$

Dann gilt:

$$\begin{aligned} a &= comp\big(tsub(T, U), tsub(S, T)\big) \\ &= comp\big(\mathsf{exponent}(tsub(U', T'), tsub(T'', U'')), \mathsf{exponent}(tsub(T', S'), tsub(S'', T''))\big) \\ &= \mathsf{exponent}\big(comp(tsub(T', S'), tsub(U', T')), comp(tsub(T'', U''), tsub(S'', T''))\big) \\ b &= tsub(S, U) \\ &= \mathsf{exponent}\big(tsub(U', S'), tsub(S'', U'')\big) \end{aligned}$$

a und b sind kompatibel. Die beiden Nachfolger

$$\big(comp(tsub(T', S'), tsub(U', T')), tsub(U', S')\big)$$
$$\big(comp(tsub(T'', U''), tsub(S'', T'')), tsub(S'', U'')\big)$$

sind Elemente von Y. □

C.4.2 Bindung dynamisch typisierter Werte

Die Effizienz der dynamischen Codierung wie hier beschrieben beruht darauf, daß die vorhandene statische Typinformation zur Abbildung von symbolischen auf konkrete Zugriffsoperationen auch statisch genutzt werden kann. Die dynamische Typinformation spiegelt lediglich die Differenz zwischen dem erwarteten Variablentyp und dem gegebenen Werttyp wieder, so daß eine konkrete Codierung zur Laufzeit in eine andere übersetzt wird, ohne daß etwa die Auflösung symbolischer Namen erforderlich wäre.

Der unmittelbare Nachteil dieses Verfahrens zeigt sich in Situationen, in denen der erwartete Typ wechselt, zum Beispiel, weil der Wert vom Typ $T!$ einer Variablen vom Typ $T?$ mit $T! \sqsubseteq T?$ an eine andere Variable vom Typ $T??$ mit $T? \sqsubseteq T??$ gebunden wird. Obwohl der Werttyp unverändert bleibt, muß die dynamische Typinformation angepaßt werden. Da aus der dynamischen Typinformation der Typ $T!$ im Allgemeinen nicht rekonstruiert werden kann, bleibt nur die Verrechnung der beiden Differenzen, also genau die kategorielle Morphismenkomposition. Zwar steht für diese das oben definierte, effektive Verfahren zur Verfügung, aber der Aufwand ist unvernachlässigbar hoch. In der Praxis läßt sich die Effizienz erfreulicherweise deutlich verbessern, worauf in Abschnitt C.4.4 eingegangen wird.

C.4.3 Aufruf dynamisch typisierter Funktionen

Mit Hilfe der Komposition von Subtypbeweisen läßt sich auch die Applikation von Funktionen erklären, die mit dynamischer Typinformation versehen sind:

$$\big(f :: \mathsf{exponent}(q, r)\big)\big(a :: s\big) \longmapsto f\big(a :: \mathit{comp}(q, s)\big) :: r$$

Angenommen die Funktion f vom Typ $T?? \to U!$ sei an eine Variable v vom Typ $T? \to U?$ gebunden. Also gilt $q : T? \to T??$, $r : U! \to U?$. Ferner sei der Argumentwert a vom Typ $T!$ an eine Variable vom Typ $T?$ gebunden. Also gilt $s : T! \to T?$. Bei β-Reduktion wird dann a an eine λ-quantifizierte Variable vom Typ $T??$ gebunden, daher ist $\mathit{comp}(q, s) : T! \to T??$ die korrekte Typinformation. Auf der Resultatseite liefert die Reduktion einen Wert vom Typ $U!$, der im Kontext an eine Variable vom Typ $U?$ gebunden wird, daher ist hier $q : U! \to U?$ die korrekte Typinformation.

C.4.4 Optimierung

In einer naiven, unoptimierten Implementierung des vorgestellten Ansatzes würde jeder Datenfluß von einer Variable in die andere die dynamische Komposition von Typinformationen erfordern. Obwohl der Aufwand dieser Komposition in dem Sinne konstant ist, daß er nur von der Größe der beteiligten Typausdrücke und nicht von der Größe der zu konvertierenden Werte abhängt, wäre dieses Verfahren sehr ineffizient.

Wesentliches Optimierungspotential steckt in der Tatsache, daß die Typen von Quelle ($T?$) und Ziel ($T??$) eines Datenflusses statisch bekannt sind, auch wenn es der Typ des fließenden Wertes ($T!$) nicht ist. Das erlaubt die Spezialisierung der Komposition durch Einsetzen des linken Arguments $l : T? \to T??$. Dadurch kann im Produktfall die Iteration und im Coproduktfall die Suche von Elementen eliminiert werden. Außerdem ergeben sich zwei relativ häufige Spezialfälle, in denen die Komposition völlig entfällt:

1. Ist der Werttyp $T!$ ebenfalls statisch bekannt, dann kann das rechte Argument $r : T! \to T?$ ebenfalls eingesetzt und die Komposition statisch vorberechnet werden. Dies ist der Fall, wenn der Wert durch Datenflußanalyse auf einen exakt typisierten Ausdruck zurückgeführt werden kann, etwa einen Konstruktor oder die Applikation einer anderen Operation, die keine Subtypen gestattet. Auch bei Substitution von exakten Werten zur Laufzeit, sprich dynamischer Spezialisierung in einem *just-in-time*-Compiler, ergibt sich eine Fülle von Anwendungsmöglichkeiten.

2. Sind Quelle und Ziel vom gleichen Typ, dann gilt $l = \mathrm{id}_{T?}$. Das Neutralitätsaxiom gestattet es also, die Komposition zu eliminieren. Insbesondere bedeutet dies, daß Ausdrücke, die auch ohne Subtypkompatibilität wohlgetypt sind, keinerlei Berechnungsaufwand für dynamische Typinformation

haben. Wird das Instrument der Subtypen nicht genutzt, dann fallen lediglich die geringen Kosten des Mitführens der initialen dynamischen Typinformation an.

Selbst wo diese Optimierung statisch nicht greift, beispielsweise weil einer der beteiligten Typen eine polymorphe Typvariable ist, kann zur Laufzeit ein identischer Subtypbeweis erkannt werden. Damit kann immerhin noch konstanter Platzbedarf der Typinformation garantiert werden.

C.5 Anwendungen

C.5.1 Beispiele

In Selbstanwendung der Theorie betrachten wir die Subtypbeziehung zwischen den semi-abstrakten Modellen *texpr*(*id*) und dem abstrakten Modell *tval*. Hier zunächst die vollständige Definition in der Objektnotation, wobei die Typfamilien bereits sortiert wurden. Zur besseren Unterscheidung von Objekt- und Metaebene schreiben wir erstere in Großbuchstaben:

$$TEXPR_\alpha = \left\{ \begin{array}{ll} \mathsf{COPRODUCT} & : TROW_{\,TEXPR_\alpha}, \\ \mathsf{EXPONENT} & : TFUN_{\,TEXPR_\alpha}, \\ \mathsf{INDIRECT} & : \alpha, \\ \mathsf{PRODUCT} & : TROW_{\,TEXPR_\alpha} \end{array} \right\} \qquad TVAL = \left\{ \begin{array}{ll} \mathsf{COPRODUCT} & : TROW_{\,TVAL}, \\ \mathsf{EXPONENT} & : TFUN_{\,TVAL}, \\ \mathsf{PRODUCT} & : TROW_{\,TVAL} \end{array} \right\}$$

$$TFUN_\alpha = (\mathsf{CODOMAIN} : \alpha, \mathsf{DOMAIN} : \alpha) \qquad TROW_\alpha = LIST_{\,MAPLET_{\,INDEX,\alpha}}$$

$$MAPLET_{\alpha,\beta} = (\mathsf{KEY} : \alpha, \mathsf{VALUE} : \beta)$$

$$LIST_\alpha = \{\mathsf{CONS} : CELL_\alpha, \mathsf{NIL} : ()\} \qquad CELL_\alpha = (\mathsf{CAR} : \alpha, \mathsf{CDR} : LIST_\alpha)$$

Der Typ *INDEX* sei gegeben. Es existiere der identische Subtypbeweis id_{INDEX}. Die Ordnung $\prec$ sei die übliche lexikographische.

Als erstes ist ein Verfahren zu entwickeln, welches aus Instanzen dieses rekursiven Gleichungssystems eine Familie von *texpr*-Objekten erzeugt. Durch Auflösung der Referenzen ergibt sich folgendes System von flachen *texpr*-Konstruktoren, indem alle geschachtelten Typen als symbolische Referenzen dargestellt werden:

$$TEXPR(x,y,z) = \mathsf{coproduct}\left[\begin{array}{rcl} \mathsf{COPRODUCT} & \mapsto & \mathsf{indirect}(x), \\ \mathsf{EXPONENT} & \mapsto & \mathsf{indirect}(y), \\ \mathsf{INDIRECT} & \mapsto & \mathsf{indirect}(z), \\ \mathsf{PRODUCT} & \mapsto & \mathsf{indirect}(x) \end{array} \right]$$

$$TVAL(x,y) = \mathsf{coproduct}\left[\begin{array}{rcl} \mathsf{COPRODUCT} & \mapsto & \mathsf{indirect}(x), \\ \mathsf{EXPONENT} & \mapsto & \mathsf{indirect}(y), \\ \mathsf{PRODUCT} & \mapsto & \mathsf{indirect}(x) \end{array} \right]$$

$$LIST(x) = \mathsf{coproduct}\left[\begin{array}{rcl} \mathsf{CONS} & \mapsto & \mathsf{indirect}(x), \\ \mathsf{NIL} & \mapsto & \mathsf{product}[] \end{array} \right]$$

$$CELL(x,y) = \mathsf{product}\left[\begin{array}{rcl} \mathsf{CAR} & \mapsto & \mathsf{indirect}(x), \\ \mathsf{CDR} & \mapsto & \mathsf{indirect}(y) \end{array} \right]$$

$$MAPLET(x,y) = \mathsf{product}\left[\begin{array}{rcl} \mathsf{KEY} & \mapsto & \mathsf{indirect}(x), \\ \mathsf{VALUE} & \mapsto & \mathsf{indirect}(y) \end{array} \right]$$

$$TFUN(x) = \mathsf{product}\left[\begin{array}{rcl} \mathsf{CODOMAIN} & \mapsto & \mathsf{indirect}(x), \\ \mathsf{DOMAIN} & \mapsto & \mathsf{indirect}(x) \end{array} \right]$$

Eine Instanziierung $texpr_U$ für einen Trägertyp U läßt sich nun wie folgt darstellen:

$$TEXPR_U = teval(\Gamma)\big(\mathsf{indirect}(s)\big) \qquad \Gamma = \left\{ \begin{array}{rcl} s & \mapsto & TEXPR(r,t,u), \\ t & \mapsto & TFUN(s), \\ r & \mapsto & LIST(p), \\ p & \mapsto & CELL(q,r), \\ q & \mapsto & MAPLET(i,s), \\ i & \mapsto & INDEX, \\ u & \mapsto & U \end{array} \right\}$$

Dieser Ausdruck ist vom Typ $texpr_V$ mit $V = \{i, p, q, r, s, t, u\}$. Dagegen wird *tval* dargestellt als:

$$TVAL = teval(\Gamma')\big(\mathsf{indirect}(s)\big) \qquad \Gamma' = \left\{ \begin{array}{lcl} s & \mapsto & TVAL(r,t), \\ t & \mapsto & TFUN(s), \\ r & \mapsto & LIST(p), \\ p & \mapsto & CELL(q,r), \\ q & \mapsto & MAPLET(i,s) \\ i & \mapsto & INDEX \end{array} \right\}$$

Für alle Trägertypen U, U' mit $U \sqsubseteq U'$ läßt sich nun die Ungleichung $tval \sqsubseteq texpr_U \sqsubseteq texpr_{U'}$ mit Beweisobjekten belegen und implementieren. Zur besseren Lesbarkeit wird hinter jedem *vtable*-Element der zugehörige Index aufgeführt.

1. Für die Beziehung $tval \sqsubseteq texpr_U$ gilt folgender Subtypbeweiscoterm mit Wurzel s:

$$s = \mathsf{coproduct}\begin{bmatrix} 1 \mapsto r, & (\mathsf{coproduct}) \\ 2 \mapsto t, & (\mathsf{exponent}) \\ 4 \mapsto r & (\mathsf{product}) \end{bmatrix} \qquad r = \mathsf{coproduct}\begin{bmatrix} 1 \mapsto p, & (\mathsf{cons}) \\ 2 \mapsto \mathrm{id}_{()} & (\mathsf{nil}) \end{bmatrix}$$

$$p = \mathsf{product}\begin{bmatrix} 1 \mapsto q, & (\mathsf{car}) \\ 2 \mapsto r & (\mathsf{cdr}) \end{bmatrix} \qquad q = \mathsf{product}\begin{bmatrix} 1 \mapsto \mathrm{id}_{index}, & (\mathsf{key}) \\ 2 \mapsto s & (\mathsf{value}) \end{bmatrix}$$

$$t = \mathsf{product}\begin{bmatrix} 1 \mapsto s, & (\mathsf{codomain}) \\ 2 \mapsto s & (\mathsf{domain}) \end{bmatrix}$$

2. Für die Beziehung $texpr_U \sqsubseteq texpr_{U'}$ gilt ein fast identischer Subtypbeweiscoterm, lediglich die Wurzel wird ersetzt durch:

$$s = \mathsf{coproduct}\begin{bmatrix} 1 \mapsto r, & (\mathsf{coproduct}) \\ 2 \mapsto t, & (\mathsf{exponent}) \\ 3 \mapsto u, & (\mathsf{indirect}) \\ 4 \mapsto r & (\mathsf{product}) \end{bmatrix}$$

   Dabei ist $u : U \to U'$ der Beweis der Annahme $U \sqsubseteq U'$.

Die hier gezeigte Selbstanwendung ist nicht nur theoretisch relevant. Sie illustriert die Lösung eines offenen praktischen Problems: Eine komplexe Transformation zwischen Datenmodellen läßt sich, besonders in funktionaler Beschreibung, oft sinnvoll als Komposition einfacher Transformationen mit Hilfe von Zwischenmodellen beschreiben. Dabei treten häufig Normalisierungsschritte auf, deren Ergebnisraum nur ein Teil des Eingaberaums ist. Ein solcher Schritt ist also vom Typ $T \to U$ mit $U \sqsubseteq T$. Ein Subtypsystem gestattet es nun, ein normalisiertes Ergebnis vom Typ U sowohl in spezialisierten Schritten vom Typ $U \to V$, als auch in allgemeineren Schritten vom Typ $T \to W$ weiter zu verarbeiten. Handelt es sich bei T aber um einen komplexen zyklischen Typcoterm, dann versagen die herkömmlichen Subtypsysteme, insbesondere die objektorientierten. Eine Schar zyklisch rekursiver Typen läßt sich in praktisch eingesetzten Typsystemen nicht in allgemeiner Weise durch Vererbung spezialisieren[1]. Der hier vorgestellte Ansatz verfügt dagegen über die nötige Ausdrucksstärke, wobei der Aufwand für den Zugriff auf ein Datum aber dem des Aufrufs einer virtuellen Methode in einem objektorientierten System entspricht.

C.5.2 Weitere Anwendungsmöglichkeiten

Die bisher betrachteten Subtypbeweise decken nur einen kleinen Teil des Datentyps *tinfo* ab. Sie lassen sich über die in der *vtable* definierte numerische Umcodierung $\vec{v}@\_$ charakterisieren: Die Subtypbeweise

[1] auch wenn die Theorie der Vererbung dies eigentlich gestattet, siehe [11]

entsprechen genau den streng monotonen Umcodierungen. Andere Klassen von Abbildungen ergeben andere Typkompatibilitätsbegriffe, die sich ohne zusätzlichen Aufwand von derselben Implementierung unterstützt werden. Da diese aber im Allgemeinen nicht eindeutig sind, treten sie an der Oberfläche einer Sprache nicht als implizite Kompatibilität, sondern als explizite vordefinierte Konversionen auf.

C.5.2.1 Permutationen

Eine bijektive Umcodierung oder Permutation der Indexnummern entspricht einer isomorphen Umsortierung oder Umbenennung von Typkomponenten. Da Benennungsunterschiede bei ansonsten strukturgleichen Definitionen ein bekanntes und häufiges Problem der softwaretechnischen Praxis darstellen, ist dies eine durchaus relevante Anwendung.

C.5.2.2 Nichtinjektive Umcodierungen

Durch nichtinjektive Umcodierungen, bei denen verschiedene Indexnummern gleichgesetzt werden, lassen sich komplexere Typkonstruktionen beschreiben, die aus abstrakter kategorieller Sicht nichtfreien Limiten und Colimiten entsprechen.

1. Eine nichtinjektive Umcodierung zwischen Produkttypen drückt aus, daß Komponenten des Domaintyps im Codomaintyp in mehreren Instanzen gespiegelt auftreten. Anders herum ausgedrückt dient der Domaintyp als Modell für einene Teil des Codomaintyps, der durch Gleichungen zwischen den Komponenten eingeschränkt ist.

2. Eine nichtinjektive Umcodierung zwischen Coprodukttypen drückt aus, daß verschiedene Varianten des Domaintyps im Codomaintyp zusammenfallen, wie durch Gleichungen zwischen Injektionen beschrieben.

Da vergleichbare Ausdrucksmächtigkeit üblicherweise zwar in Spezifikations- aber nicht in Programmiersprachen zu finden ist, lassen sich praktische Anwendungsfälle für diese theoretische sehr reizvolle Konstruktion nur schwer finden. Sie erlaubt aber prinzipiell die Übertragung von fortgeschrittenen algebraischen oder kategoriellen Typkonstruktionen aus der Spezifikation in die Programmierung.

C.5.2.3 Berechnung mit Signaturmorphismen

Zahlreiche homomorphe Abbildungen zwischen Datentypen lassen sich als Signaturmorphismen beschreiben. So läßt sich das bereits aus Kapitel 4 bekannte Beispiel der Zählens von Listenelementen bei geeigneter Definition der natürlichen Zahlen als Morphismus zwischen den Typsignaturen beschreiben:

$$nat = \{\mathsf{zero} : (), \mathsf{pos} : pnat\} \qquad\qquad pnat = (\mathsf{pred} : nat)$$

In diesem Beispiel entspricht dem Zählen die Umcodierung

$$\mathsf{nil} \mapsto \mathsf{zero} \qquad\qquad \mathsf{cons} \mapsto \mathsf{pos} \qquad\qquad \mathsf{tail} \mapsto \mathsf{pred}$$

In solchen Fällen besteht die Berechnung des homomorphen Bildes allein in der Typkonvertierung. Diese wird aber durch das Austauschen der dynamischen Typinformation realisiert, und hat nicht den mindestens linearen Aufwand der herkömmlichen Berechnung durch Traversierung.

Abbildungsverzeichnis

1.1 Namensanalyse der rekursiven Typdefinition *Nat* . . . 19
1.2 Namensanalyse der rekursiven Funktionsdefinition *fac* . . . 19
1.3 „Zyklisches“ Scheme-Programm . . . 20
1.4 Division im Pseudocode . . . 24

6.1 Multiplikation von Zyklen . . . 137
6.2 Zyklus mit mehreren Eintrittspunkten . . . 137
6.3 Plazierung von Markierungen . . . 138

7.1 Programmmodell . . . 156
7.2 Aufrufkonventionen . . . 177

8.1 Komplexe Graphansicht . . . 200
8.2 Die *map*-Funktion . . . 203
8.3 Testaufruf . . . 203
8.4 Interpreter . . . 207
8.5 JIT-Compiler . . . 208
8.6 C-Compiler . . . 209
8.7 Übersicht . . . 210
8.8 Plattformen mit virtuellem Maschinencode . . . 211
8.9 Plattformen mit nativem Maschinencode . . . 212

Programmverzeichnis

2.1 Signaturfunktoren . . . 36
2.2 Induzierte syntaktische Gleichheit . . . 37
2.3 Dialgebren, Kata- und Anamorphismen . . . 37
2.4 Erreichbare Umgebung . . . 38
2.5 Lokaler Anteil . . . 40
2.6 Kompatibilität . . . 40
2.7 Individualtransformation . . . 41
2.8 Identische Signatur . . . 43
2.9 Konstante Signatur . . . 44
2.10 Produkt von Signaturen . . . 44
2.11 Universelle Operationen auf Produkten . . . 45
2.12 Coprodukt von Signaturen . . . 45
2.13 Universelle Operationen auf Coprodukten . . . 46
2.14 Listenabschluß von Signaturen . . . 46
2.15 Bisimulationsoperation polynomieller Signaturen . . . 50
3.1 Zellen- und Variablenadressen . . . 70
3.2 Speicherzustände . . . 71
3.3 Speicherzustand als Coalgebra . . . 71
3.4 Aufbau von Belegungen . . . 71
3.5 Zugriff auf Belegungen . . . 72
3.6 Untercoalgebraordnung von Speicherzuständen . . . 73
3.7 Erzeugen einer Zelle . . . 75
4.1 Typdefinitionen für Berechnungsalgorithmen . . . 93
4.2 Algorithmischer Grundschritt (coalgebraisch) . . . 94
4.3 Algorithmischer Grundschritt (algebraisch) . . . 94
4.4 Iterativer Berechnungsalgorithmus . . . 95
4.5 Rekursiver Berechnungsalgorithmus ohne Zyklenerkennung . . . 97
4.6 Rekursiver Berechnungsalgorithmus mit Zyklenprotokollierung . . . 99
4.7 Rekursiver Berechnungsalgorithmus mit Zyklenbehandlung . . . 100
5.1 Typdefinitionen für Entscheidungsalgorithmen . . . 115
5.2 Iterativer Entscheidungsalgorithmus . . . 116
5.3 Rekursiver Entscheidungsalgorithmus ohne Zyklenerkennung . . . 117
5.4 Rekursiver Entscheidungsalgorithmus mit Zyklenprotokollierung . . . 119
5.5 Rekursiver Entscheidungsalgorithmus mit Zyklenbehandlung . . . 120
5.6 Größtes und kleinstes Modell . . . 124
7.1 Speicher der virtuellen Maschine . . . 153
7.2 Universelle Signatur der virtuellen Maschine . . . 154
7.3 Der Stack der Maschine . . . 155
7.4 Werteuniversum der virtuellen Maschine . . . 155
7.5 Maschinenprogramm . . . 155

7.6 Heap-Typsystem der Maschine . . . 159
7.7 Stack-Typsystem der Maschine . . . 159
7.8 Elimination von Typinstanziierungen . . . 161
7.9 Prozeduren . . . 163
7.10 Konstanten . . . 164
7.11 Ablaufmonade . . . 166
7.12 Operationen der Ablaufmonade . . . 166
7.13 Stack-Frames . . . 167
7.14 Blöcke und Anweisungen . . . 169
7.15 Blockausführung . . . 169
7.16 Manipulation von Operanden . . . 169
7.17 Zugriff auf den Stack . . . 170
7.18 Zugriff auf den Heap . . . 170
7.19 Konstruktion von Daten . . . 171
7.20 Dereferenzierung . . . 172
7.21 Implementierte Dereferenzierung . . . 172
7.22 Globaler Kontrollfluß . . . 172
7.23 Verzweigung . . . 173
7.24 Interprozeduraler Kontrollfluß . . . 174
7.25 Prozeduraufruf in der Maschine . . . 174
7.26 Zyklenerkennung in der Maschine . . . 175
7.27 Zyklenbehandlung in der Maschine . . . 176
7.28 Aufrufkonventionen der Maschine . . . 178
7.29 Parameterübergabe in der Maschine . . . 179

Mathematisches Glossar

Dieses Kapitel umfaßt alle in der bisherigen Arbeit verwendeten mathematischen Begriffe, die als mathematisches Allgemeingut unhinterfragt vorausgesetzt werden, deren Modellierung und/oder Gebrauch aber nicht so standardisiert sind, daß eine explizite Darlegung nicht einen Gewinn an Klarheit bringen würde.

Abbildung Seien A, B zwei Mengen. Eine eindeutige Zuordnungsvorschrift f heißt Abbildung vom Typ $(A \to B)$, geschrieben $f : A \to B$, falls für alle x gilt:

$$x \in A \implies f(x) \in B$$

Eine Abbildung kann in dieser Deutung verschiedene Typen besitzen. Insbesondere gilt: Wenn $f : A \to B$, dann auch $f : A' \to B'$ mit $A' \subseteq A$ und $B \subseteq B'$.

Seien $f, g : A \to B$ zwei Abbildungen. Wir schreiben:

$$f =_{(A \to B)} g \iff \forall\, x \in A.\ f(x) = g(x)$$

Der Typ wird meist weggelassen, falls er sich aus dem Kontext ergibt.

Jede typisierte Abbildung $f : A \to B$ definiert eindeutig eine Relation $\operatorname{graph}_A f : A \leftrightarrow B$:

$$\operatorname{graph}_A f = \{(x, f(x)) \mid x \in A\}$$

Auch hier wird der Typ oft weggelassen.

Jede Abbildung $f : A \to B$ ist insbesondere eine partielle Abbildung $f : A \nrightarrow B$.

Zu jedem $y \in B$ existiert eine *konstante* Abbildung $const_y$, so daß für alle $x \in A$ gilt:

$$const_y(x) = y$$

Eine Abbildung $f : A \to B$ heißt

1. *injektiv*, falls gilt $\operatorname{Ker} f = \Delta_A$,
2. *surjektiv*, falls gilt $\mathcal{P}(f)(A) = B$,
3. *bijektiv*, falls sie injektiv und surjektiv ist.

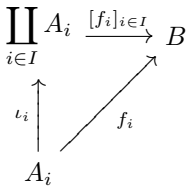
Abzählbarkeit Eine Menge A heißt abzählbar, wenn sie gleichmächtig zur Menge der natürlichen Zahlen oder einer ihrer Teilmengen ist, sonst überabzählbar. Nur für abzählbare Mengen läßt sich eine Repräsentation finden, die allen Elementen endliche Ausdrücke zuordnet.

Charakteristische Abbildung Sei A eine Menge. Jeder Teilmenge $B \subseteq A$ ist eineindeutig eine Abbildung $f : A \to \{w, f\}$ zugeordnet, so daß gilt:

$$f(x) = \begin{cases} w & x \in B \\ f & x \notin B \end{cases}$$

Ergibt sich die Obermenge A aus dem Kontext, dann können B und f identifiziert werden.

Coprodukt Sei $\mathcal{K}$ eine Kategorie und I eine Menge. Sei $(A_i)_{i\in I}$ eine I-indizierte Familie von Objekten in $\mathcal{K}$. Dann ist das Coprodukt der A_i definiert als ein Objekt $\coprod_{i\in I} A_i$ zusammen mit einer Familie von (Mono-)Morphismen $(\iota_i)_{i\in I}$, den *Injektionen*, so daß für jedes Objekt B und für jede Familie $(f_i : A_i \to B)_{i\in I}$ ein eindeutiger Morphismus $[f_i]_{i\in I} : \coprod_{i\in I} A_i \to B$ existiert, so daß $f_i = [f_i]_{i\in I} \circ \iota_i$:



Das Coprodukt ist bis auf Isomorphie eindeutig bestimmt. Für den Spezialfall $I = \{1, 2\}$ schreibt man auch $A_1 + A_2$.

Das Coprodukt läßt sich zu einem Funktor fortsetzen. Seien $(A_i), (B_i)$ zwei Familien von Objekten und $(f_i : A_i \to B_i)$ eine Familie von Morphismen. Dann definiert man:

$$\coprod_{i\in I} f_i : \coprod_{i\in I} A_i \to \coprod_{i\in I} B_i$$
$$\coprod_{i\in I} f_i = [\iota_i \circ f_i]_{i\in I}$$

In der Kategorie **Set** der Mengen und Abbildungen ist die disjunkte Vereinigung ein Coprodukt.

Disjunktheit Seien A, B zwei Mengen. A und B heißen *disjunkt*, falls sie keine gemeinsamen Elemente enthalten: $A \cap B = \emptyset$

Die *disjunkte Vereinigung* ist partiell definiert als:

$$A \uplus B = \begin{cases} A \cup B & A \cap B = \emptyset \\ \bot & \text{sonst} \end{cases}$$

Epimorphismus Sei $\mathcal{K}$ eine Kategorie. Seien A, B, C Objekte in $\mathcal{K}$. Ein Morphismus $e : A \to B$ ist epi, wenn für jedes Paar $f_1, f_2 : B \to C$ gilt: $f_1 \circ e = f_2 \circ e \implies f_1 = f_2$.

In der Kategorie **Set** der Mengen und Abbildungen sind genau die surjektiven Abbildungen epi.

Familie Eine Familie $(a_i)_{i\in I}$ von Elementen $a_i \in A$ ist eine alternative Sicht auf eine Abbildung $f : I \to A$, wobei a_i ein neuer Name für $f(i)$ ist. Die *Projektionsabbildung* π_j ist für alle Familien mit $j \in I$ definiert als:

$$\pi_j\big((a_i)_{i\in I}\big) = a_j$$

Eine Familie mit $I = \{0, \ldots, n\}$ heißt *endliche Folge*. Eine Familie mit $I = \mathcal{N}$ heißt *unendliche Folge*.

Finalobjekt Sei $\mathcal{K}$ eine Kategorie. Ein Finalobjekt 1 in $\mathcal{K}$ ist ein Objekt, so daß zu jedem Objekt A ein eindeutiger Morphismus $! : A \to 1$ existiert.

Finalobjekte sind bis aus Isomorphie eindeutig bestimmt.

In der Kategorie **Set** der Mengen und Abbildungen ist eine einelementige Menge $\{\star\}$ ein Finalobjekt.

Funktor Seien $\mathcal{K}, \mathcal{L}$ zwei Kategorien. Ein Funktor $F : \mathcal{K} \to \mathcal{L}$ ist eine Abbildung von Objekten und Morphismen von $\mathcal{K}$ auf Objekte und Morphismen von $\mathcal{L}$, die Typisierung, Identität und Komposition erhält:

$$h : A \to B \implies F(h) : F(A) \to F(B) \qquad \begin{aligned} F(\mathrm{id}_A) &= \mathrm{id}_{F(A)} \\ F(h \circ g) &= F(h) \circ F(g) \end{aligned}$$

Ein gewöhnlicher Funktor heißt auch *kovariant*. Soll statt $\mathcal{L}$ die duale Kategorie $\mathcal{L}^{op}$ betrachtet werden, spricht man von einem *kontravarianten* Funktor $F' : \mathcal{K} \to \mathcal{L}^{op}$. Unter dieser Sicht gilt:

$$h : A \to B \implies F'(h) : F'(B) \to F'(A)$$

Initialobjekt Sei $\mathcal{K}$ eine Kategorie. Ein Initialobjekt 0 in $\mathcal{K}$ ist ein Objekt, so daß zu jedem Objekt A ein eindeutiger Morphismus $? : 0 \to A$ existiert.

Initialobjekte sind bis auf Isomorphie eindeutig bestimmt.

In der Kategorie **Set** der Mengen und Abbildungen ist die leere Menge ein Initialobjekt.

Inklusion Seien A, B zwei Mengen. Wenn $A \subseteq B$, dann existiert eine *Inklusionsabbildung* $\mathrm{in} : A \to B$, definiert als:

$$\mathrm{in}(x) = x$$

In der Kategorie **Set** der Mengen und Abbildungen ist die Inklusionsabbildung ein Monomorphismus.

Isomorphismus Sei $\mathcal{K}$ eine Kategorie. Seien A, B Objekte in $\mathcal{K}$. Ein Morphismus $i : A \to B$ ist iso, wenn ein inverser Morphismus $i^{-1} : B \to A$ existiert, so daß $i^{-1} \circ i = \mathrm{id}_A$ und $i \circ i^{-1} = \mathrm{id}_B$. Dieser ist eindeutig bestimmt. Die Klasse der Isomorphismen ist also der Schnitt von Rektraktionen und Coretraktionen.

In der Kategorie **Set** der Mengen und Abbildungen sind genau die bijektiven Abbildungen iso.

Kategorie Eine Kategorie $\mathcal{K} = \big(O, M, \tau, (\mathrm{id}_A), \circ\big)$ besteht aus:

1. einer Klasse O von Objekten,
2. einer Klasse M von Morphismen,
3. einer Typisierungsrelation $\tau \subseteq O \times O \times M$,
4. einer O-indizierten Familie von *identischen* Morphismen $(\mathrm{id}_A)_{A \in O}$,
5. einer partiellen zweistelligen Verknüpfung $\circ : M \times M \nrightarrow M$, der *Komposition*

Man schreibt auch $f : A \to B$ für $(A, B, f) \in \tau$. Es gelten folgende Axiome:

$$\begin{aligned} &\mathrm{id}_A : A \to A \\ &f : A \to B \wedge g : B \to C \implies g \circ f : A \to C \\ &f : A \to B \implies \mathrm{id}_B \circ f = f = f \circ \mathrm{id}_A \\ &f \circ (g \circ h) = (f \circ g) \circ h \end{aligned}$$

Kern Seien A, B Mengen und $f : A \to B$ eine Abbildung. Der Kern $\mathrm{Ker}\, f$ ist eine Äquivalenzrelation auf A, welche folgendermaßen definiert ist:

$$(x, y) \in \mathrm{Ker}\, f \iff f(x) = f(y)$$

Liste Sei A eine Menge. Die Menge A^* bezeichnet das *freie Monoid* über A, also die kleinste Menge, für die gilt:

$$[] \in A^*$$
$$x \in A \wedge \vec{w} \in A^* \implies x : \vec{w} \in A^*$$

Die Elemente von A^* heißen Listen über A. Wir schreiben auch $[x_1, \ldots, x_n]$ für $x_1 : \ldots : x_n : []$. Die eigentliche Monoidoperation, die Verkettung, ist folgendermaßen rekursiv definiert:

$$[] +\!\!+ \vec{w} = \vec{w} \qquad (x : \vec{v}) +\!\!+ \vec{w} = x : (\vec{v} +\!\!+ \vec{w})$$

Der Konstruktor $\cdot^*$ kann zu einem Funktor fortgesetzt werden, wobei gilt:

$$f^*([]) = [] \qquad f^*(x : \vec{w}) = f(x) : f^*(\vec{w})$$

Eine Liste $[x_1, \ldots, x_n] \in A^*$ kann auch als partielle Abbildung $f : \mathcal{N} \rightharpoonup A$ mit $f(i) = x_i$ aufgefaßt werden. Wir schreiben daher auch

$$\mathrm{dom}[x_1, \ldots, x_n] = \{1, \ldots, n\}$$
$$\mathrm{codom}[x_1, \ldots, x_n] = \{x_1, \ldots, x_n\}$$

Mealy-Automat Ein Mealy-Automat ist charakterisiert durch ein Tupel $(Q, A, B, \tau, \omega, s)$ mit folgenden Komponenten:

Q	Zustandsmenge
A	Eingabealphabet (endlich)
B	Ausgabealphabet (endlich)
$\tau : Q \times A \to Q$	Transitionsfunktion
$\omega : Q \times A \to B$	Ausgabefunktion
$s \in Q$	Startzustand

Ein Automat heißt endlich, wenn seine Zustandsmenge endlich ist.

Monomorphismus Sei $\mathcal{K}$ eine Kategorie. Seien A, B, C Objekte in $\mathcal{K}$. Ein Morphismus $m : B \to C$ ist mono, wenn für jedes Paar $f_1, f_2 : A \to B$ gilt: $m \circ f_1 = m \circ f_2 \implies f_1 = f_2$.

In der Kategorie **Set** der Mengen und Abbildungen sind genau die injektiven Abbildungen mono.

Multimenge Eine Multimenge verhält sich ähnlich wie eine Menge, außer daß ein Element mehrfach enthalten sein kann. Eine Multimenge M über den Elementen einer Grundmenge A läßt sich durch eine charakteristische Funktion $\#_M : A \to \mathcal{N}$ beschreiben. Daher schreiben wir auch $M : \mathcal{N}^A$.

Die Kardinalität $|M|$ einer Multimenge $M : \mathcal{N}^A$ ist definiert als

$$|M| = \sum_{a \in A} \#_M(a)$$

Falls diese Reihe eine natürliche Zahl definiert, heißt M *endlich*.

Jede Menge $A' \subseteq A$ ist auch eine spezielle Multimenge, wobei gilt:

$$\#_{A'}(x) = \begin{cases} 0 & x \notin A' \\ 1 & x \in A' \end{cases}$$

Schnitt und Vereinigung lassen sich von Mengen auf Multimengen verallgemeinern. Daneben existieren auch Summe und Differenz:

$$\#_{M \cup N}(x) = max\big(\#_M(x), \#_N(x)\big) \qquad \#_{M+N}(x) = \#_M(x) + \#_N(x)$$
$$\#_{M \cap N}(x) = min\big(\#_M(x), \#_N(x)\big) \qquad \#_{M-N}(x) = max\big(\#_M(x) - \#_N(x), 0\big)$$

Der Definitionsbereich einer Multimenge M ist definiert als:

$$\operatorname{dom} M = \{x \mid \#_M(x) > 0\}$$

Natürliche Transformation Seien $\mathcal{K}, \mathcal{L}$ Kategorien und $F, G : \mathcal{K} \to \mathcal{L}$ Funktoren. Eine natürliche Transformation $\eta : F \Rightarrow G$ ist eine Familie (η_A) von $\mathcal{L}$-Morphismen $\eta_A : F(A) \to G(A)$, so daß für alle Morphismen $h : B \to C$ gilt $\eta_C \circ F(h) = G(h) \circ \eta_B$:

$$\begin{array}{ccc} F(B) & \xrightarrow{F(h)} & F(C) \\ \Big\downarrow{\scriptstyle \eta_B} & & \Big\downarrow{\scriptstyle \eta_C} \\ G(B) & \xrightarrow[G(h)]{} & G(C) \end{array}$$

Alternativ kann η auch als ein einziger Morphismus mit vielen Typisierungen (polymorph) aufgefaßt werden.

Partielle Abbildung Seien A, B Mengen. Eine *partielle* Abbildung $f : A \rightharpoonup B$ ist eine eindeutige Zuordnung von einigen (im Allgemeinen nicht allen) Elementen von A zu Elementen von B. Sie läßt sich äquivalent als totale Abbildung $f' : A \to B \uplus \{\bot\}$ darstellen. Dabei heißt f' *Totalisierung* von f. Man schreibt:

$$\begin{array}{lll} f'(a) = b & f(a) \text{ def} & \text{(definierte Stelle)} \\ f'(a) = \bot & f(a) \text{ undef} & \text{(undefinierte Stelle)} \end{array}$$

Man definiert weiter:

$$\begin{aligned} \operatorname{dom}_{A\to B} f &= \{a \in A \mid \exists b \in B.\ f(a) = b\} \\ \operatorname{codom}_{A\to B} f &= \{b \in B \mid \exists a \in A.\ f(a) = b\} \end{aligned}$$

$$(f \oplus_{A\to B} g)(a) = \begin{cases} g(a) & a \in \operatorname{dom}_{A\to B} g \\ f(a) & a \notin \operatorname{dom}_{A\to B} g \end{cases}$$

$$(f \ominus s)(a) = \begin{cases} \bot & a \in s \\ f(a) & a \notin s \end{cases}$$

Die Typ wird weggelassen, falls er sich aus dem Kontext ergibt. Es gilt:

$$\begin{aligned} \operatorname{dom}_{A\to B} f &= \overline{\mathcal{P}}(\operatorname{graph}_A f)(B) \cap A \\ \operatorname{codom}_{A\to B} f &= \mathcal{P}(\operatorname{graph}_A f)(A) \subseteq B \end{aligned}$$

Auf den partiellen Abbildungen gleichen Typs existiert eine partielle Ordnung. Seien A, B Mengen. Eine partielle Abbildung $f : A \rightharpoonup B$ heißt *Teilabbildung* von $g : A \rightharpoonup B$ (geschrieben $f \sqsubseteq g$), falls für alle $x \in \operatorname{dom} f$ gilt:

$$f(x) = g(x)$$

Zu jedem Paar A, B von Mengen existiert die *leere* (überall undefinierte) partielle Abbildung $\emptyset : A \to B$. Diese ist stets die kleinste partielle Abbildung. Die sogenannten *Einheitsabbildung* $\{x \mapsto y\}$ ist für alle $x \in A$ und $y \in B$ definiert als die minimale partielle Abbildung $f : A \rightharpoonup B$, für die $f(x) = y$ gilt.

Seien $f, g : A \rightharpoonup B$ zwei partielle Abbildungen. Die *vereinigte Abbildung*, geschrieben $f \sqcup g$, ist die kleinste obere Schranke von f und g bezüglich $\sqsubseteq$. Die vereinigte Abbildung existiert nicht, falls sich f und g an einer gemeinsamen definierten Stelle widersprechen.

Partition Sei A eine Menge. Eine Menge $P \subseteq \mathcal{P}(A)$ heißt *Partition* von A, falls gilt:

$$\emptyset \notin P \qquad p, p' \in P \implies p = p' \vee p \cap p' = \emptyset \qquad \bigcup_{p \in P} p = A$$

Zu jeder Partition P existiert genau eine *Partitionierungsabbildung* $\mathrm{part} : A \to P$, so daß für alle $a \in A$ gilt:

$$a \in \mathrm{part}(a)$$

Potenz Sei **Rel** die Kategorie der Mengen und Relationen, **Set** die Kategorie der Mengen und Abbildungen, und $\overline{\mathbf{Set}}$ die dazu duale Kategorie. Die beiden *Potenzfunktoren* sind definiert als:

$$\mathcal{P} : \mathbf{Rel} \to \mathbf{Set} \qquad \overline{\mathcal{P}} : \mathbf{Rel} \to \overline{\mathbf{Set}}$$

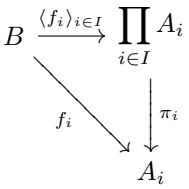
$$\mathcal{P}(A) = \{X \mid X \subseteq A\} \qquad \overline{\mathcal{P}}(A) = \{X \mid X \subseteq A\}$$

$$\mathcal{P}(R)(X) = \{y \mid \exists\, x \in X.\ x\ R\ y\} \qquad \overline{\mathcal{P}}(R)(Y) = \{x \mid \exists\, y \in Y.\ x\ R\ y\}$$

Dabei heißt $\mathcal{P}$ auch *kovariant* oder *Bildfunktor*, und $\overline{\mathcal{P}}$ auch *kontravariant* oder *Urbildfunktor*. Alle Abbildungen $\mathcal{P}(r)$ und $\overline{\mathcal{P}}(r)$ sind monoton bezüglich $\subseteq$.

Da die Potenzfunktoren häufig auf Relationen angewendet werden, die (partiellen) Abbildungen entsprechen, schreiben wir auch $\mathcal{P}(f)$ für $\mathcal{P}(\mathrm{graph}\, f)$.

Produkt Sei $\mathcal{K}$ eine Kategorie und I eine Menge. Sei $(A_i)_{i \in I}$ eine I-indizierte Familie von Objekten in $\mathcal{K}$. Dann ist das Produkt der A_i definiert als ein Objekt $\prod_{i \in I} A_i$ zusammen mit einer Familie von (Epi-)Morphismen $(\pi_i)_{i \in I}$, den *Projektionen*, so daß für jedes Objekt B und für jede Familie $(f_i : B \to A_i)_{i \in I}$ ein eindeutiger Morphismus $\langle f_i \rangle_{i \in I} : B \to \prod_{i \in I} A_i$ existiert, so daß $f_i = \pi_i \circ \langle f_i \rangle_{i \in I}$:



Das Produkt ist bis auf Isomorphie eindeutig bestimmt. Für den Spezialfall $I = \{1, 2\}$ schreibt man auch $A_1 \times A_2$.

Das Produkt läßt sich zu einem Funktor fortsetzen. Seien $(A_i), (B_i)$ zwei Familien von Objekten und $(f_i : A_i \to B_i)$ eine Familie von Morphismen. Dann definiert man:

$$\prod_{i \in I} f_i : \prod_{i \in I} A_i \to \prod_{i \in I} B_i$$

$$\prod_{i \in I} f_i = \langle f_i \circ \pi_i \rangle_{i \in I}$$

In der Kategorie **Set** der Mengen und Abbildungen ist das kartesische Produkt ein Produkt.

Relation Seien A, B zwei Mengen. Eine Menge $R \subseteq A \times B$ heißt Relation zwischen A und B, geschrieben $R : A \leftrightarrow B$. Wir schreiben:

$$a\ R\ b \iff (a, b) \in R$$

$$a_0\ R_1\ a_1\ R_2\ a_2 \cdots a_{n-1}\ R_n\ a_n \iff a_0\ R_1\ a_1 \wedge a_1\ R_2\ a_2 \wedge \cdots \wedge a_{n-1}\ R_n\ a_n$$

Zwischen beliebigen Mengen A, B existieren die *Nullrelation* $\emptyset$ und die *Allrelation* $A \times B$. Auf einer beliebigen Menge A existiert außerdem die *Diagonalrelation* $\Delta_A = \{(x, x) \mid x \in A\}$.

Zu jeder Relation $R : A \leftrightarrow B$ existiert die *inverse* Relation $R^{-1} : B \leftrightarrow A$ und die *komplementäre* Relation $\overline{R} : A \leftrightarrow B$, wobei:

$$a\ R\ b \iff b\ R^{-1}\ a \qquad\qquad a\ R\ b \iff \neg a\ \overline{R}\ b$$

Für graphische Relationssymbole ($\rightarrow$) schreiben wir R^{-1} auch als ($\leftarrow$) und $\overline{R}$ auch als ($\not\rightarrow$).

Zu jedem Paar von Relationen $R : A \leftrightarrow B$ und $Q : B \leftrightarrow C$ existiert die *Komposition* $Q \circ R : A \leftrightarrow C$, wobei:

$$a\ (Q \circ R)\ c \iff \exists b.\ a\ Q\ b\ R\ c$$

Wir schreiben auch R^n, wobei gilt:

$$R^0 = \Delta \qquad\qquad R^1 = R \qquad\qquad R^{m+n} = R^m \circ R^n$$

Eine Relation $R : A \leftrightarrow A$ heißt

1. *reflexiv*, falls $\Delta \subseteq R$,
2. *irreflexiv*, falls $R \cup \Delta = \emptyset$,
3. *symmetrisch*, falls $R = R^{-1}$,
4. *antisymmetrisch*, falls $R \cup R^{-1} = \emptyset$,
5. *transitiv*, falls $R \circ R \subseteq R$.

Eine reflexive, symmetrische, transitive Relation heißt *Äquivalenzrelation*, eine reflexive, antisymetrische, transitive Relation heißt *partielle Ordnung*. Die kleinste transitive Obermenge von R heißt *transitiver Abschluß*, geschrieben R^+. Die kleinste reflexive und transitive Obermenge von R heißt *reflexiv-transitiver* oder KLEENE-Abschluß, geschrieben R^*.

Terminationsfunktion Sei A eine Menge, wobei typischerweise nicht bekannt ist, ob A endlich oder unendlich ist. Sei R eine Relation auf A. Wenn eine streng antimonotone Abbildung $f : A \to \mathcal{N}$ existiert

$$a\ R\ b \implies f(a) > f(b)$$

dann ist jede Kette $a_1\ R\ a_2\ R \cdots$ endlich. Ist A der Zustandsraum eines Algorithmus und R eine Übergangsrelation, dann terminiert der Algorithmus.

Sei nun R^* der reflexive transitive Abschluß von R. Ist für jedes $a \in A$ die R-Umgebung $\mathcal{P}(R)(\{a\})$ endlich, dann ist auch für jedes $a \in A$ die R^*-Umgebung $\mathcal{P}(R^*)(\{a\})$ endlich. Ihre Größe kann durch $n^{f(a)}$ abgeschätzt werden, falls für alle b gilt $a\ R^*\ b \implies n \geq |\mathcal{P}(R)(\{b\})|$. Ist A der Zustandsraum eines Algorithmus und R eine endlich verzweigende Rekurrenzrelation, dann terminiert der Algorithmus.

Verband Ein Verband $(A, \sqsubseteq)$ besteht aus einer nichtleeren Menge A und einer reflexiven, transitiven Relation $\sqsubseteq : A \leftrightarrow A$, so daß zu jeder endlichen Teilmenge $X \subseteq A$ das *Supremum* $\bigsqcup X$ und das *Infimum* $\bigsqcap X$ existieren:

$$\forall x \in X.\ \bigsqcap X \sqsubseteq x \qquad\qquad \forall y.\ (\forall x \in X.\ y \sqsubseteq x) \implies y \sqsubseteq \bigsqcap X$$

$$\forall x \in X.\ x \sqsubseteq \bigsqcup X \qquad\qquad \forall y.\ (\forall x \in X.\ x \sqsubseteq y) \implies \bigsqcup X \sqsubseteq y$$

Man schreibt:

$$\top = \bigsqcap \emptyset \qquad\qquad x \sqcap y = \bigsqcap \{x, y\}$$

$$\bot = \bigsqcup \emptyset \qquad\qquad x \sqcup y = \bigsqcup \{x, y\}$$

Ein Verband heißt *vollständig*, wenn Supremum und Infimum auch für alle unendlichen Teilmengen $X \subseteq A$ existieren.

Index

Abbildung, *255*
  charakteristische, *255*
  Inklusion, 21, 237, *257*
  Kern, 30, 34, *257*
  konstante, *255*
  partielle, *259*
  Partitionierung, 21, 96, *260*
  vereinigte, 47, *259*
Abhängigkeit
  funktionale, *62*
abzählbar, 35, 40, *255*
Adressierung
  direkte, *siehe* Speicherzustand
  indirekte, *siehe* Speicherzustand
Algebra, *19*
  initiale, *19*, 25, 47, 48, 201
  Katamorphismus, *siehe* Morphismus
  Kategorie, *19*
  Konstruktor, *19*, 82
  minimale, *21*
  Morphismus, *19*
  nicht-initiale, 48
  Operation, *19*
  Quotient
    abstrakter, *22*
    eigentlicher, *22*
  Term-, 43
  Träger, *19*
  Unter-
    abstrakte, *21*
    eigentliche, *21*
all, *siehe* Liste
Ana_1, *83*
Ana_2, *85*, 122
Ana_{2a}, *87*, 207
Ana_3, *87*, 122, 215
any, *siehe* Liste
app, *siehe* Liste
`apply`, *siehe* TASM
Aufrufgraph, 122
Aufzählungstyp, *148*
Automat
  Mealy-, *258*

balance, *siehe* Liste
Bananenklammer, *siehe* Klammer
Baum, 8
  binärer, 233
  Such-, 240
  Syntax-, 6
Bild, *siehe* Funktor
Bisimilarität, *siehe* Coalgebra
Bisimulation, *siehe* Coalgebra
`body`, *siehe* TASM
Bruch, *siehe* Rationalzahl
bzip, *siehe* Liste

`call`, *siehe* TASM
chain, *siehe* Liste
Charity, 206
Chinesischer Restsatz, 124
Closure, 149, 167, 182, 184
Coalgebra, 14–15, *20*
  Anamorphismus, *siehe* Morphismus
  Bisimilarität, *30*, 74, 116–118, 205, 244
    modulo, 207, 209, 222, 234
  Bisimulation, *29*, 53, 247
  Dekonstruktor, *20*
  Element
    azyklisches, *27*
    endlich darstellbares, *46*
    irrelevantes, *45*
    ununterscheidbares, *45*
    zyklenfreies, *27*
    zyklisches, *27*
  endliche, 57
  Erreichbarkeit, 106
    Blatt, *26*
    kausale, *26*, 36
    syntaktische, *36*
    Wurzel, *26*
  finale, 15, *20*, 25, 73, 76, 201
  Kategorie, *20*
  minimale, *21*
  Morphismus, *20*

nicht-finale, 43, 56, 76
Operation, *20*
Quotient
abstrakter, *22*
eigentlicher, *22*, 45
rationale, *27*, *46*, 49
Träger, *20*
Unter-
abstrakte, *21*
eigentliche, *21*, 45, 61
zyklenfreie, *27*
Coinduktion, 21
`const`, *siehe* TASM
Coprodukt, *siehe* Typ, 238, *siehe* Funktor
von Kalkülen, *siehe* Kalkül
Corekursion
primitive, 52–54, 73, 82, 90, *siehe* Kalkül, 140, 208
Produktivität, 53
rationale, 54, 208
semantische, *50*
Coterm, 43–48, 210
äquivalenter, *44*
allgemeiner, *44*
einfacher, *45*
endlicher, *46*, 54, 90
finale Semantik, *44*
kanonischer, 45
minimaler, *45*
Morphismus, *44*
`cotuple`, *siehe* TASM
`curry`, *siehe* TASM

Datensatz, 24, 28, 74, 82
Individualtransformation, *29*
initialer, *29*
kompatibler, *28*
linearer, *37*
lokaler Anteil, 24, *28*
Prototyp, *29*
Referenz, 24
rein lokaler, *28*
`default`, *siehe* TASM
del, *siehe* Liste
`deprecated`, *siehe* Meta-Attribut
Dialgebra, 25
disjunkt, *256*
`dito`, *siehe* TASM
Division, *siehe* Rationalzahl
`dup`, *siehe* TASM

EBNF, 139
Epimorphismus, *siehe* Morphismus
Erreichbarkeit, *siehe* Coalgebra
Erwartung, *siehe* Kalkül
EULER
Satz von, 233
`eval`, *siehe* TASM
even, *siehe* Liste

Familie, *256*
FERMAT
Satz von, *siehe* EULER
`field`, *siehe* TASM
filter, *siehe* Liste
Finalobjekt, *siehe* Objekt
Fixpunkt, 6, 41, 99
gemischter, 112
größter, 100, 111, 205, 219, 223, 238, 244
kleinster, 100, 111, 205, 219
optimaler, 205
Folge
endliche, *256*
unendliche, *256*
Frame, *siehe* Speicher
`fun`, *siehe* TASM
Funktion
ACKERMANN-, 208, 234
Fakultät, 6
Funktor, 246, *257*
Bild-, *260*
Coprodukt, 31, *256*
Injektion, *256*
punktweises, 33
finitärer, *27*
Grad, *27*
identischer, *31*
KLEENE-Abschluß, 31, 40
konstante Signatur, *27*
konstanter, *31*
abzählbarer, *35*
kontravarianter, *257*
kovarianter, *257*
Liste, *258*
polynomieller, 52
abzählbarer, *35*, 40
streng, *31*
verallgemeinert, *33*
Potenz, *260*
kontravariante, *260*
kovariante, 36, *260*
Produkt, 31, *260*
Projektion, *260*
punktweises, *32*

Signatur, 18
Urbild-, 75, *260*

garbage collection, *siehe* Speicher
GCC, 194
get, *siehe* TASM
ghc, 172, 173
Gleichheit
tiefe, 205
goto, 183
Graph
Kontrollfluß-, 184
Visualisierung, 187

halt, *siehe* TASM
Hash
Funktion, 242
Tabelle, 240
Heap, *siehe* Speicher
hom, *siehe* Meta-Attribut
HORNER-Verfahren, 224
hugs, 173
Hylomorphismus, *siehe* Morphismus

import, *siehe* TASM
Individualtransformation, 82
Induktion, 19
Initialobjekt, *siehe* Objekt
inline, *siehe* Meta-Attribut
ins, *siehe* Liste
insitu, *siehe* TASM
Interpretation, 73, *74*
abstrakte, *76*, 79
homomorphe, *123*, 133, 168, 190, 226, 237, 253
Rumpf, *75*
sekundäre, *75*
invar, *siehe* Meta-Attribut
Isomorphismus, *siehe* Morphismus
Iterpretation
Rumpf, 79

Kalkül, *92*
Ableitung, *92*
Konklusion, *92*
Operator, *99*
Prämisse, *92*
abstrakter, *101*
asynchrones Produkt, 95
bedingter, *96*, 100, 113, 114
abstrakter, *101*
Bedingung, 115
Bisimilarität, *116*
Coprodukt, *94*, 100, 112, 113, 115
determinierter, *100*, 105
Erwartung, *108*
konsistente, *110*
stark konsistente, *110*, 111
triviale, 110
Modell, *98*, *siehe* Fixpunkt
primitiv corekursiver, *106*
primitiv rekursiver, *102*, 105
rational corekursiver, *106*
Regel, *92*, 107
Schnittstelle, *96*
Vereinigung, *94*
abstrakte, *101*
Kategorie, 246, *257*
KLEISLI-, 24
CPO, 49
Set, 18, 49
Träger-, 18
Kawa, 192
Kern
seeAbbildung, 257
Klammer
Banane, 20
Linse, 20
KLEISLI, *siehe* Kategorie
Kombinator
Y-, 172
kontravariant, 239, *siehe* Funktor
kovariant, 239, *siehe* Funktor

Linsenklammer, *siehe* Klammer
Liste, *258*
all, *229*
any, *229*
app, *226*
Assoziations-, 237, 240
Auswählen, 223, 227
balance, *233*
Basistransformation, 233
bzip, *234*
chain, *229*
del, *227*
Einfügen, 227
even, *231*, 233
Filter, 230
filter, *230*
ins, *227*
Kette, 229
Löschen, 227
lexikographische Ordnung, 229
map, 190, *226*, 230

mark, *230*
merge, *231*
msort, *231*
odd, *231*, 233
qsort, *231*
Quantor, 229
rep, *227*
Ring-, 232
sel, *227*
serial, *233*
Sortieren, 231
sweep, *230*
Transformation, 226
Verkettung, 226
Wiederholung, 227
Zählen, 51, 75, 76, 78, 79, 253
`loop`, *siehe* TASM

Malice, 137, 175
map, *siehe* Liste
mark, *siehe* Liste
MEALY-Automat, *siehe* Automat
merge, *siehe* Liste
Mergesort, 231
Meta-Attribut, 168–169
`deprecated`, *168*
`hom`, *168*
`inline`, *168, 169*, 183, 184
`invar`, *169*, 226
`native`, *177*
`quasi`, 129, *168, 169*, 222, 233
`tag`, *169*
MIT/GNU Scheme, 192
ML, 193
`module`, *siehe* TASM
Monoid
freies, *258*
Monomorphismus, *siehe* Morphismus
Morphismus
Ana-, *20*, 25, 52, 76, 79
Apo-, 227
Epi-, *256*
Hylo-, 206
identischer, 246, *257*
Iso-, 19, 21, *257*
Kata-, *19*, 25, 50
höherer Ordnung, 208
Komposition, 246, *257*
Mono-, *258*
Para-, 206
Signatur-, 253
msort, *siehe* Liste
Multimenge, *258*

`native`, *siehe* Meta-Attribut

Objekt
finales, *256*
initiales, *257*
odd, *siehe* Liste
Opal, 193
Ordnung
lexikographische, *siehe* Rationlzahl, *siehe* Liste
partielle, 61, 259, *261*

`param`, *siehe* TASM
Parameter
Differenz-, *152*, 164, 209
Paramorphismus, *siehe* Morphismus
Partition, *260*
Pattern, *47*, 75, 92, 107, 205
Bedeutung, *47*
geschachteltes, *48*
lineares, *47*, 101
Matching, 201
getypt, *47*
ungetypt, *47*
Pointer Algebra, 204
Polymorphie, *siehe* Typ
`pop`, *siehe* TASM
Potenz, *siehe* Funktor
Produkt, *siehe* Typ, *siehe* Funktor
asynchrones, *siehe* Kalkül
kartesisches, 148
Konversion, 209
Produktivität, 204
Prolog, 192
Prozedur
Aufruf, 185
tail-, 53, 182, 184
Substitution, 168, 169, 183

qsort, *siehe* Liste
`quasi`, *siehe* Meta-Attribut
Quasizyklus, *128*
Quicksort, 231

Rationalzahl
Bruch, 223
Division, 9, 221–222
normierte, 219
Ordnung, 219
Periode, 217, 223
Präfix, 217, 223

Subtraktion, 220–221
`ref`, *siehe* TASM
Referentielle Transparenz, *siehe* Transparenz, referentielle
Referenz, *siehe* Datensatz
markierte, *siehe* Speicherzustand
Null-, 58
Referenzzählung
zyklische, 130
Regel, *siehe* Kalkül
Rekursion, 6
primitive, 50–52, *siehe* Kalkül, 140, 208
semantische, 50
syntaktische, *50*
tail-, 128, 177, 190, 224
Termination, 52
Relation, *260*
Äquivalenz, 25, *261*
Abschluß
KLEENE-, *261*
reflexiv-transitiver, *261*
transitiver, *261*
antisymetrische, 27
antisymmetrische, *261*
inverse, *261*
komplementäre, *261*
Komposition, *261*
reflexive, 29, *261*
symmetrische, *261*
transitive, 29, *261*
rep, *siehe* Liste
Res_1, *103*
Res_2, *105*, 122
Res_{2a}, *107*, 207
Res_3, *108*, 122, 215
Resolution, 103
Restsatz
chinesischer, *siehe* Chinesischer Restsatz
`return`, *siehe* TASM

Scheme, 7, 192
Schnittstelle, *siehe* Kalkül
induzierte, *siehe* Typ
sel, *siehe* Liste
serial, *siehe* Liste
`set`, *siehe* TASM
Signatur, *siehe* Funktor
Signaturfunktor, 18
SML/NJ, 193
Speicher
Heap, 141–142, 180
Stack, 87, 141–143, 180
Frame, 142, 154
Verwaltung, 66–68, 128, 130–133, 210
Speicherverwaltung, 140
Speicherzustand, 57–60
direkt adressierter, *57*
indirekt adressierter, *57*
legaler, *57*, *58*, 74
markierter, *125*, 128, 130, 131, 133, 169
Variable
lebende, *58*, 83
Zuweisen, *64*, 82
virtueller, 79
Wurzel, *57*
Entfernen, *65*
Hinzufügen, *65*
Zelle
Erzeugen, *63*, 82
Löschen, *64*
lebende, *57*, *58*, 63, 81
lebensfähige, 61
Spezifiation
finale, *101*
Squiggol, 20
Stack, *siehe* Speicher
Operanden-, 144, 155, 182
`swap`, *siehe* TASM
sweep, *siehe* Liste
SWI Prolog, 192
`switch`, *siehe* TASM

`tag`, *siehe* Meta-Attribut
tail-Rekursion, *siehe* Rekursion
TASM
`apply`, *158*, 167, 169, 178, 181, 184
`body`, *158*, 178, 180, 182
`call`, *161*, 165, 168, 169, 177, 178, 180, 182
`const`, *152*
`cotuple`, *158*, 181
`curry`, *158*, 167, 168
`default`, *160*
`dito`, *163*, 178, 180, 209, 210, 230
`dup`, *156*, 182
`eval`, *161*, 165, 167, 169, 172, 177, 178, 180, 182
`field`, *158*, 180
`fun`, *151*
`get`, *159*, 178, 180
`halt`, *160*, 177, 180–182
`import`, *146*, 176
`insitu`, *156*, 165, 184
`loop`, *161*, 177, 178
`module`, *146*, 176

`param`, *156*
`pop`, *156*, 182
`ref`, *147*
`return`, *161*, 177, 180, 182, 183
`set`, *159*, 169, 178, 180
`swap`, *156*, 182
`switch`, *160*, 180
`tuple`, *158*, 181
`type`, *146*
`var`, *156*
`vars`, *156*
`yield`, *158*, 177
Term
Teil-, *27*, 102
Terminationsfunktion, 85, 88, 104, 109, *261*
Tiefensuche, 85
Transformation
Individual-, *siehe* Datensatz
natürliche, *259*
Transparenz
referentielle, *61*, 76, 170–171
Traversierung
bottom-up, 52
top-down, 52
Tupel
offenes, *62*
`tuple`, *siehe* TASM
Typ
Coprodukt, 148, 236, 238, 242, 244
Dekonstruktor, 238, 244
Diskriminator, *238*, 244
dynamischer, 241, 244, 249, 253
finaler, 41, 52
Funktion, 236, 239, 244, 249
initialer, 41, 51
Instanziierung, 149, 176
isomorpher, 253
polymorpher, 150, 151, 236
Produkt, 148, 236, 237, 242, 244
Prozedur, 149
rationaler, 41, 54, 90
Schnittstelle
induzierte, 237, 239
Selektor, *237*, 240, 244
Subtyp, 237, *238*
Beweis, 239, 242–245
Variable, 149–151, 176
`type`, *siehe* TASM

überabzählbar, *255*
überabzählbar, 40, 53
Unifikation, 241

Urbild, *siehe* Funktor

`var`, *siehe* TASM
Variable, *siehe* Speicherzustand
`vars`, *siehe* TASM
Verband, *261*
vollständiger, 99, *261*
Vereinigung
disjunkte, 19, *256*
von Kalkülen, *siehe* Kalkül

Wurzel, *siehe* Speicherzustand

`yield`, *siehe* TASM

Zahl
natürliche, 19
Zelle, *siehe* Speicherzustand
markierte, *siehe* Speicherzustand
Zyklus
Behandlung, 9–10, 87, 91, 108, 151, 163, 208
effiziente, 122
Erkennung, 9–10, 87, 108, 161, 176, 185, 207
logischer, 5
Quasi-, *128*, 168, 209, 222

Literaturverzeichnis

[1] Abiteboul, Serge ; Bussche, Jan van d.: Deep Equality Revisited. In: Ling, T.W. (Hrsg.) ; Mendelzon, A.O. (Hrsg.) ; Vieille, L. (Hrsg.): *Deductive and Object-Oriented Databases*, Springer, 1995 (Lecture Notes in Computer Science 1013), 213–228

[2] Aczel, Peter: *Non-Well-Founded Sets*. University of Chicago Press, 1988 (CSLI Lecture Notes)

[3] Norm ISO/IEX 8824 , 2000. *ASN.1: Abstract Syntax Notation One*

[4] Babbage, Charles: *Online Quotations*. School of Mathematics and Statistics, University of St Andrews, `http://www-groups.dcs.st-and.ac.uk/~history/Quotations/Babbage.html`

[5] Bird, Richard: Constructive Functional Programming. In: Broy, Manfred (Hrsg.): *Marktoberdorf International Summer School on Constructive Methods in Computer Science*, Springer, 1989 (NATO Advanced Science Institute Series)

[6] Brouwer, L.E.J.: *Online Quotations*. School of Mathematics and Statistics, University of St Andrews, `http://www-groups.dcs.st-and.ac.uk/~history/Quotations/Brouwer.html`

[7] Brownbridge, D. R.: Cyclic reference counting for combinator machines. In: *Functional programming languages and computer architecture*, Springer, 1985 (Lecture Notes in Computer Science 201), S. 273–288

[8] Cardelli, Luca: Extensible Records in a Pure Calculus of Subtyping. Version: 1994. `http://citeseer.ist.psu.edu/cardelli91extensible.html`. In: Gunter, C. A. (Hrsg.) ; Mitchell, J. C. (Hrsg.): *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. The MIT Press, 1994, 373–425

[9] Norm ISO/IEC 23271 , 2003. *Common Language Infrastructure*

[10] Cockett, Robin ; Fukushima, Tom: About Charity / University of Calgary. 1992 (92/480/18). – Forschungsbericht

[11] Cook, William R.: *A Denotational Semantics of Inheritance*, Brown University, Diss., 1989

[12] Erwig, Martin: Inductive Graphs and Functional Graph Algorithms. In: *Journal of Functional Programming* 11 (2001), Nr. 5, S. 467–492

[13] Norm ANSI X3.215 , 1994. *Forth*

[14] Fournet, Cédric ; Gordon, Andrew D.: Stack inspection: Theory and variants. In: *ACM Trans. Program. Lang. Syst.* 25 (2003), Nr. 3, S. 360–399. – DOI http://doi.acm.org/10.1145/641909.641912. – ISSN 0164–0925

[15] Ghani, Neil ; Lüth, Christoph ; De Marchi, Federico: Coalgebraic monads. In: *Electronic Notes in Theoretical Computer Science* 65 (2002), Nr. 1. `http://www.elsevier.nl/locate/entcs/volume65.html`

[16] *The Glasgow Haskell Compiler Homepage*. http://www.haskell.org/ghc

[17] Gibbons, Jeremy ; Hutton, Graham: *Proof methods for structured corecursive programs*. http://citeseer.ist.psu.edu/article/gibbons99proof.html. Version: 1999

[18] Gibbons, Jeremy ; Hutton, Graham ; Altenkirch, Thorsten: When is a Function a Fold or an Unfold? Version: 2001. http://citeseer.ist.psu.edu/gibbons01when.html. In: Corradini, Andrea (Hrsg.) ; Lenisa, Marina (Hrsg.) ; Montanari, Ugo (Hrsg.): *Proceedings 4th Workshop on Coalgebraic Methods in Computer Science, CMCS'01, Genova, Italy, 6–7 Apr. 2001* Bd. 44(1). Elsevier, 2001

[19] Hausmann, Daniel: *Data types and computability via final coalgebras*, Technische Universität Dresden, Diplomarbeit, 2004

[20] Hensel, Ulrich ; Huisman, Marieke ; Jacobs, Bart ; Tews, Hendrik: Reasoning about classes in object-oriented languages: Logical models and tools. In: Hankin, Ch. (Hrsg.): *European Symposium on Programming*, Springer, 1998 (Lecture Notes in Computer Science 1381), 105–121

[21] Hopcroft, John: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Kohavi, J. (Hrsg.): *Theory of Machines and Computation*, Academic Press, 1971, S. 189–196

[22] Jacobs, Bart: Objects and Classes Co-Algebraically. In: *Object-Orientation and Parallelism and Persistence*, Kluwer Academic, 1996, 83–103

[23] Jones, Mark P. ; Peyton Jones, Simon L.: Lightweight Extensible Records for Haskell. In: *Haskell Workshop*

[24] Jones, Richard E.: Tail recursion without Space Leaks. In: *Journal of Functional Programming* 2 (1992), January, Nr. 1, 73–79. http://citeseer.ist.psu.edu/jones91tail.html

[25] Karczmarczuk, Jerzy: *The Most Unreliable Technique in the World to compute π*. http://users.info.unicaen.fr/~karczma/arpap/lazypi.ps.gz. Version: 1998

[26] Kelsey, Richard ; Clinger, William ; Rees, Jonathan: Revised^5 Report on the Algorithmic Language Scheme. In: *ACM SIGPLAN Notices* 33 (1998), Nr. 9, 26–76. http://citeseer.ist.psu.edu/kelsey98revised.html

[27] King, David J.: *Functional Programming and Graph Algorithms*, University of Glasgow, Diss., 1996

[28] Kiselyov, Oleg: *Lazy evaluation and lazy recursion in flattening of a (cyclic) list*. http://okmij.org/ftp/Scheme/misc.html#lazy-flattener. – Archiv, Abruf: 2006–06–23

[29] Launchbury, John: Graph Algorithms with a Functional Flavour. In: J. Jeuring, E. M. (Hrsg.): *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Bastad, Sweden*, 1995 (Lecture Notes in Computer Science 925), S. 308–331

[30] Leibniz, Gottfried W. ; Herring, Herbert (Hrsg.): *Fünf Schriften zur Logik und Metaphysik*. Reclam, 1966 (Universal-Bibliothek 1898)

[31] Lenisa, Marina: A Complete Coinductive Logical System for Bisimulation Equivalence on Circular Objects. In: Thomas, Wolfgang (Hrsg.): *Foundations of Software Science and Computation Structure*, Springer, 1999 (Lecture Notes in Computer Science 1578), 243–257

[32] Lindholm, Tim ; Yellin, Frank: *The Java Virtual Machine Specification*. 2nd Edition. 1999

[33] Lins, Rafael D.: Cyclic Reference Counting with Lazy Mark-Scan. In: *Information Processing Letters* 44 (1992), Nr. 4, 215–220. http://citeseer.ist.psu.edu/lins90cyclic.html

[34] Manna, Zohar ; Shamir, Adi: The theoretical aspects of the optimal fixedpoint. In: *SIAM Journal of Computing* 3 (1976), Nr. 5, S. 414–426

[35] Martínez, Alejandro D. ; Wachenchauzer, Rosita ; Lins, Rafael D.: Cycle Reference Counting with Local Mark-Scan. In: *Inf. Process. Lett.* 34 (1990), Nr. 1, S. 31–35

[36] Meertens, Lambert: Algorithmics — Towards Programming as a Mathematical Activity. In: *Proceedings of the CWI Symposium on Mathematics and Computer Science*, North-Holland, 1986, S. 289–334

[37] Meijer, Erik ; Fokkinga, Maarten ; Paterson, Ross: Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In: *FPCA* Bd. 523, Springer, 1991, S. 124–144

[38] Meijer, Erik ; Hutton, Graham: Bananas in space: extending fold and unfold to exponential types. In: *FPCA*, ACM Press, 1995

[39] Object Management Group: *Meta-Object Facility (MOF), Version 1.4.* `http://www.omg.org/technology/documents/formal/mof.htm`

[40] Möller, Bernhard: Towards pointer algebra. In: *Sci. Comput. Program.* 21 (1993), Nr. 1, S. 57–90. – DOI http://dx.doi.org/10.1016/0167–6423(93)90008–D. – ISSN 0167–6423

[41] Möller, Bernhard: Linked Lists Calculated. 1997 (07). – Forschungsbericht

[42] Nebel, Bernhard: Terminological Cycles: Semantics and Computational Properties. Version: 1991. `http://citeseer.ist.psu.edu/nebel91terminological.html`. In: Sowa, J. F. (Hrsg.): *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* Morgan Kaufmann Publishers, 1991, 331–361

[43] Ohori, Atsushi: A Polymorphic Record Calculus and Its Compilation. In: *ACM Transactions on Programming Languages and Systems* 17 (1995), November, Nr. 6, 844–895. `http://citeseer.ist.psu.edu/ohori95polymorphic.html`

[44] Patterson, Anna ; Costello, Ton: *Implicit Programming and Computable Optimal Fixed Points.* `http://www-formal.standford.edu/annap/imp.ps`

[45] Pavlovic, Dusko ; Pratt, Vaughan: On coalgebra of real numbers. In: Jacobs, Bart (Hrsg.) ; Rutten, Jan (Hrsg.): *Proceedings of the Workshop on Coalgebraic Methods in Computer Science (CMCS)*, Elsevier, 2000 (Electronic Notes in Computer Science 19)

[46] Peyton Jones, Simon L.: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine. In: *Journal of Functional Programming* 2 (1992), Nr. 2, 127-202. `http://citeseer.ist.psu.edu/peytonjones92implementing.html`

[47] Peyton Jones, Simon L.: *Haskell 98 Language and Libraries.* Cambridge University Press, 2003. – ISBN 0521826144

[48] Peyton Jones, Simon L. ; Hall, Cordelia V. ; Hammond, Kevin ; Partain, Will ; Wadler, Philip: The Glasgow Haskell compiler: a technical overview. In: *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*

[49] Peyton Jones, Simon L. ; Jones, Mark P. ; Meijer, Erik: Type classes: an exploration of the design space. In: *Haskell Workshop*

[50] Peyton Jones, Simon L. ; Marlow, Simon: *Secrets of the Glasgow Haskell compiler inliner.* `http://citeseer.ist.psu.edu/jones99secrets.html`. Version: 1999

[51] Peyton Jones, Simon L. ; Nordin, Thomas ; Oliva, Dino: C--: A Portable Assembly Language. In: *Implementation of Functional Languages*, Springer, 1997 (Lecture Notes in Computer Science 1467)

[52] Reed, Lou: Vicious Circle. In: *Rock and Roll Heart*, 1976

[53] Reid, Alastair: Putting the Spine Back in the Spineless Tagless G-Machine: An Implementation of Resumable Black-Holes. In: *Implementation of Functional Languages*, Springer, 1998 (Lecture Notes in Computer Science 1595), 186–199

[54] Rémy, Didier: *Effcient representation of extensible records.* `http://citeseer.ist.psu.edu/442909.html`. Version: 1994

[55] Rutten, Jan: Universal coalgebra: a theory of systems. In: *Theoretical Computer Science* 249 (2000), Nr. 1, S. 3–80. – DOI http://dx.doi.org/10.1016/S0304–3975(00)00056–6. – ISSN 0304–3975

[56] Santayana, Jorge Augustín Nicolás Ruiz de: *The Life of Reason.* 1905/1906

[57] Schintz, Michael ; Odersky, Martin: Tail Call Elimination on the Java Virtual Machine. In: *Electronic Notes in Theoretical Computer Science* 59 (2001), Nr. 1. `http://www.elsevier.nl/locate/entcs/volume59.html`

[58] Shivers, Olin: *Control-Flow Analysis of Higher-Order Languages*, Carnegie Mellon University, Diss., 1991

[59] Smyth, Michael B. ; Plotkin, Gordon D.: The Category-Theoretic Solution of Recursive Domain Equations. In: *SIAM Journal on Computing* 11 (1982), Nr. 4, S. 761–785

[60] Telford, Alastair ; Turner, David: Ensuring Streams Flow. In: *Algebraic Methodology and Software Technology*, Springer, 1997 (Lecture Notes in Computer Science 1349), S. 509–523

[61] Trancón y Widemann, Baltasar: An Implementation Strategy for Polynomial Subtyping. In: *Proceedings of the 5th Symposium on Trends in Functional Programming (TFP)* Ludwig-Maximilians Universität München, 2004

[62] *Unified Modeling Language (UML) Resource Page.* `http://www.uml.org`

[63] Vene, Varmo ; Uustalu, Tarmo: Functional Programming with Apomorphisms (Corecursion). In: *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics* 47 (1998), Nr. 3, 147–161. `http://citeseer.ist.psu.edu/vene98functional.html`

[64] Wakeling, David: Mobile Haskell: Compiling Lazy Functional Programs for the Java Virtual Machine. In: *Proceedings of the 1998 Conference on Programming Languages, Implementations, Logics and Programs (PLILP'98)* Bd. 1490, Springer, 1998, 335–??

[65] Wallach, Dan S. ; Felten, Edward W.: *Understanding Java Stack Inspection.* `http://citeseer.ist.psu.edu/wallach98understanding.html`. Version: 1998