Praktische Probleme bei der Konstruktion von Software zur automatisierten Terminierungsanalyse

Dissertation
zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik
von

Jan Schultze

Dortmund
2014

Tag der mündlichen Prüfung:	10. September 2014
Dekan/Dekanin:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter/Gutachterinnen:	Prof. Dr. Ernst-Erich Doberkat
	Prof. Dr. Peter Padawitz

Danksagung

Die vorliegende Arbeit ist im Rahmen meiner Tätigkeit am Lehrstuhl für Software-Technologie der Technischen Universität Dortmund entstanden. Ich möchte mich bei folgenden Mitgliedern der Fakultät für Informatik besonders herzlich bedanken.

Prof. Dr. Ernst-Erich Doberkat danke ich für seine Betreuung und Unterstützung, auf die ich mich stets verlassen konnte, sowie für die weitreichenden Freiheiten und vor allem die guten Fragen.

Bei Prof. Dr. Peter Padawitz bedanke ich mich für die Begutachtung und die Gespräche über die funktionale Programmierung im Rahmen des Haskell-Stammtischs. Prof. Dr. Jakob Rehof und Dr. Doris Schmedding danke ich für die Teilnahme an der Prüfungskommission, die schnellen Zusagen und die angenehme Prüfungsatmosphäre.

Für die offenen Türen, die Ratschläge, die kollegiale Atmosphäre und den des Öfteren wehenden Rückenwind danke ich allen Menschen, mit denen ich am Lehrstuhl zusammen arbeiten durfte und von denen ich viel gelernt habe.

Inhaltsverzeichnis

1 Einleitung **5**

I Terminierungsanalyse imperativer Sprachen **9**

2 Überblick über Teil I **11**

3 Termersetzungssysteme **13**

3.1 Definition 13

3.1.1 Terme über Signaturen 13

3.1.2 Kontexte 15

3.1.3 Substitutionen 17

3.2 Ersetzungsregeln 18

3.3 Terminierung 20

3.3.1 Abstrakte Reduktionssysteme und einige ihrer Eigenschaften . 20

3.3.2 Beweistechniken zum Nachweis der Terminierungseigenschaft von Termersetzungssystemen 21

3.3.3 Modularität 22

4 Das Dependency Pair Framework **29**

4.1 Grundbegriffe 29

4.2 Terminierungskriterien 31

4.3 Berechnung des Abhängigkeitsgraphen 36

4.4 Vom Ansatz zum Framework 39

5 Automatisierte Terminierungsanalyse von Haskell-Programmen **49**

5.1 Die Sprache Haskell 50

5.1.1 Klassifikation 50

5.1.2 Funktionen und Funktionsauswertung 52

5.1.3 Formale Beschreibung der Semantik von Haskell 54

5.2 Terminierungsgraphen für Haskell-Programme 60

5.3 Erzeugung eines $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Problem aus dem Terminierungsgraphen . 66

6 Automatisierte Terminierungsanalyse imperativer Sprachen 73

6.1 Kombinatoren . . . 74

6.1.1 Bedeutung von Kombinatoren . . . 75

6.1.2 Ein einfaches, terminierendes Programm . . . 75

6.1.3 Probleme bei der Terminierungsanalyse . . . 78

6.1.4 Problemvermeidung durch Programmtransformation . . . 80

6.2 Verwaltung und Darstellung des Programmzustands . . . 84

6.2.1 Verwaltung des Zustands mit Transitionsmonaden . . . 84

6.2.2 Darstellung der Variablenbelegung . . . 85

6.3 Übersetzung auf Basis der SSA-CPS-Korrespondenz . . . 89

6.3.1 Modellierung des Zustandsraums . . . 90

6.3.2 Realisierung der Zustandsübergänge . . . 99

6.3.3 Einbindung der Zustandsübergänge . . . 101

6.3.4 Die Heap-Komponente . . . 102

7 Fazit zu Teil I 105

7.1 Verwandte Arbeiten . . . 106

II Terminierungsanalyse in der Gegenwart von Ein- und Ausgabe 109

8 Überblick über Teil II 111

9 Eine Sprache mit Ein- und Ausgabe 113

9.1 Die Grammatik von **IMP** . . . 113

9.1.1 Die Grammatik der Urfassung . . . 113

9.1.2 Die Grammatik der erweiterten Fassung . . . 114

9.1.3 Syntaktische Definitionen . . . 116

9.2 Die Semantik von **IMP** . . . 118

9.2.1 Der Zustand in **IMP** . . . 119

9.2.2 Die Semantik der Urfassung von **IMP** . . . 121

9.2.3 Die Semantik der Spracherweiterungen . . . 124

9.3 Semantische Äquivalenz . . . 126

10 Transformation von Sequenzen atomarer Anweisungen **129**
10.1 Eliminierung einzelner Anweisungen 129
10.2 Sequenzen atomarer Anweisungen der Länge zwei 131
10.3 Permutationen von beliebig langen Sequenzen atomarer Anweisungen 134
10.3.1 Abhängigkeiten zwischen Anweisungen 136
10.3.2 Kopplung κ und Abhängigkeit $\lhd_S$ 139
10.3.3 Ein Zusammenhang zwischen $\lhd_S$ und semantischer Äquivalenz 142

11 Transformation von Sequenzen terminierender Anweisungen **149**
11.1 Die Kopplung beliebiger, terminierender Anweisungen 149
11.2 Permutationen von beliebig langen Sequenzen terminierender Anweisungen 158
11.3 Approximationen der Kopplungsrelation 163

12 Fazit zu Teil II **173**
12.1 Verwandte Arbeiten 174

A Mathematische Grundbegriffe **177**

Quelltextverzeichnis **181**

Abbildungsverzeichnis **183**

Literaturverzeichnis **185**

1 Einleitung

Die Konstruktion korrekter Software ist das zentrale Anliegen der Software-Technologie. Die Verifikation von Software, also das Entscheiden, ob ein Programm einer gegebenen Spezifikation genügt, ist dabei das zentrale Problem. Nach dem Satz von Rice [Ric53] kann dieses Problem im Allgemeinen für nicht triviale Funktionen nicht gelöst werden. Ursächlich ist das ebenfalls unentscheidbare Halteproblem [Tur36]. Ohne die Berücksichtigung des Terminierungsverhalten können nur partielle Korrektheitsergebnisse ermittelt werden. In der Einleitung zu seinem bereits 1976 erschienenen Buch „A discipline of programming“ [Dij76] schreibt Dijkstra, dass er den starken Hang zur partiellen Korrektheit bedaure, der immer noch so üblich sei[1]. Die Situation hat sich seitdem nicht verbessert. So stellen Byron Cook et al. in ihrem Artikel [CPR11] aus dem Jahr 2011 fest, dass viele glauben, dass es für alle Programme unmöglich sei, Terminierungseigenschaften zu beweisen. Dies ist sicherlich eine Übertreibung, aber es lässt sich nicht abstreiten, dass in den meisten Software-Projekten keine Zeit für Terminierungsbeweise aufgewendet wird.

Einen Ausweg könnten automatisierte Verfahren zur Terminierungsanalyse bieten. Aufgrund der Unentscheidbarkeit des Halteproblems betrachten derartige Verfahren stets das relaxierte Halteproblem. Dabei darf die Prüfung eines Programms nicht nur mit den Ergebnissen „terminiert“ oder „terminiert nicht“, sondern auch mit einem „unentschieden“ enden. Während aus einem unentschiedenen Ergebnis weder Terminierung noch das Gegenteil folgt, so sind die ersten beiden Resultate zuverlässig in dem Sinne, dass die Terminierung bewiesen oder widerlegt werden konnte. Gemäß dieser Semantik können sich verschiedene Verfahren in ihren Antworten unterscheiden, jedoch nie widersprechen. Die Güte eines Verfahrens ergibt sich daraus, für wie viele Programme eindeutige Resultate ermittelt werden können.

Viele bestehende Terminierungsanalyseverfahren beschäftigen sich mit Termersetzungssystemen. Hingegen sind Verfahren für imperative Sprachen selten anzutreffen. Intuitiv liegen deklarative Sprachen wie Prolog oder Haskell zwischen diesen beiden Welten. Auch für sie existieren automatisierte Verfahren zur Terminierungsanaly-

[1] Im englischen Original heißt es „... I came to regret the strong bias towards partial correctness that is still so common.“

se. Dabei nutzen die Erfolgreichsten die Nähe der deklarativen Sprachen zu den Termersetzungssystemen aus. Um von den Ergebnissen für Termersetzungssysteme auch für imperative Sprachen zu profitieren, müssen Wege gefunden werden, die existierenden Methoden auf imperative Programme anzuwenden. Da die Turing-Vollständigkeit von Termersetzungssystemen garantiert, dass jedes imperative Programm $\mathcal{P}$ durch ein geeignetes Termersetzungssystem $\mathcal{R}_\mathcal{P}$ simuliert werden kann, können die bekannten Verfahren auf $\mathcal{R}_\mathcal{P}$ angewendet werden. Jedoch ist dieser Ansatz nur dann von Erfolg gekrönt, wenn bei der Transformation des Programms $\mathcal{P}$ kein Termersetzungssystem entsteht, das die bestehenden Methoden überfordert.

Eine Alternative zur Transformation imperativer Programme in Termersetzungssysteme besteht darin, imperative Programme in deklarative Programme zu übersetzen und diese mit den bekannten Methoden zu analysieren. Dieser Ansatz wird in [AAC+08] und [SMP10] verfolgt. In beiden Arbeiten wird die logische, regelbasierte Programmierung als deklarative Form gewählt. In der vorliegenden Arbeit werden die Probleme betrachtet, die sich ergeben, wenn die rein funktionale Sprache Haskell als Zwischenstufe eingesetzt wird, also imperative Programme in rein funktionale Programme übersetzt werden. Die Software AProVE [GTSKF04] wird eingesetzt, um das Terminierungsverhalten der erzeugten Haskell-Programme automatisiert zu analysieren. AProVE transformiert die Haskell-Eingabe wiederum in eine Eingabe für das Dependency Pair Framework, welches die erfolgreichste Methode zur automatisierten Analyse von Termersetzungssystemen darstellt. Aufgrund der mathematischen Natur und der Unterstützung für den monadischen Programmieransatz erweist sich Haskell als eine Sprache, die besonders gut geeignet ist, imperative Sprachen zu simulieren und sich somit als Zielsprache für Übersetzer anbietet. Ferner erleichtert die Zustandslosigkeit die Verifikation von Haskell-Programmen. Es scheint, als wäre Haskell ein günstiger Zwischenschritt. In Teil I wird gezeigt, warum diese Vermutung nicht zutrifft.

Die meisten imperativen Sprachen erlauben einen flexiblen Umgang mit dem zur Verfügung stehenden Speicher. Die abgelegten Informationen können beliebig überschrieben werden. Deklarative Sprachen wie Haskell bieten keinen vergleichbar direkten Speicherzugriff. Zur Übersetzung der imperativen Sprachen müssen diese destruktiven Variablenveränderungen daher auf andere Weise nachgeahmt werden. In Teil I werden Probleme beschrieben, die bei der Terminierungsanalyse derartiger simulierter Speichermodelle auftreten.

Diese Überlegungen können unabhängig von der konkreten imperativen Sprache geführt werden. Objektorientierte Sprachen stellen zur Zeit die am weitesten verbreitete Form von imperativen Sprachen dar. Am Beispiel der konkreten Sprache

Jinja werden in diesem Teil abschließend die Probleme betrachtet, die sich in objektorientierten Sprachen ergeben.

Die Erkenntnis über die begrenzten Erfolge bei der automatisierten Terminierungsanalyse führen zu den Überlegungen im zweiten Teil der Arbeit, indem ein Beitrag zur Zerlegung der zu analysierenden Programme geleistet wird. Die Komplexität der in der Industrie erstellten Software ist zu hoch, als dass heute bestehende Terminierungsanalyseverfahren vollständige Programme analysieren könnten. Des Weiteren gibt es eine Reihe von Programmen, wie zum Beispiel Betriebssysteme oder Webserver, bei denen gewünscht ist, dass sie im Regelfall nicht terminieren. Der Nutzen dieser Systeme hängt davon ab, dass sie in der Lage sind, auf Eingaben, die erst im Laufe der Ausführung gemacht werden, in endlicher Zeit mit entsprechenden Ausgaben zu reagieren. Dabei sind die später folgenden Eingaben möglicherweise abhängig von den bisherigen Ausgaben. Die Software ist in diesem Sinne interaktiv. Folglich können Ein- und Ausgabe nicht ignoriert werden.

Solange Ein- und Ausgabe in der Terminierungsanalyse nicht berücksichtigt werden, lassen sich lediglich die Teilsequenzen zwischen Ein- und Ausgabeoperationen analysieren. Es ist allerdings zu erwarten, dass Algorithmen von Ein- und Ausgabeoperationen durchsetzt sind. Dabei stellt eine Ausgabe in eine Datei, formal betrachtet, nicht nur eine Ausgabe, sondern auch eine Eingabe dar, da Fehlerzustände des Dateisystems auch gemeldet werden. Es ergeben sich daher unter Umständen nur sehr kleine Teilprogramme, die überprüft werden können. Die Aussagekraft einer Analyse von Teilprogrammen ist jedoch eingeschränkt und zwar in folgender Weise. Kommt eine Terminierungsanalyse zu dem Ergebnis, dass ein Teilprogramm nicht terminiert, so muss es zumindest einen Zustand geben, in dem das Teilprogramm nicht terminiert. Jedoch wird die Zahl der möglichen Zustände, in denen das Teilprogramm ausgeführt wird, durch die im Programm vorhergehenden Operationen eingeschränkt. Gehört der von der Terminierungsanalyse ermittelte Zustand, für den das Teilprogramm nicht terminiert, nicht zu den tatsächlich möglichen Zuständen, so war das Ergebnis der Analyse irrelevant. Dieses Problem ist im Allgemeinen nicht feststellbar, da es aufgrund des Halteproblems nicht möglich ist, die Menge der möglichen Zustände vollständig zu bestimmen. Es kann daher wünschenswert sein, ein Eingabeprogramm vor der Zerlegung zu transformieren, sodass für die Analyse problematische Ein- und Ausgabeoperationen verschoben werden und ein möglichst langes, analysierbares Teilprogramm entsteht.

Welche Zerlegungen in Teilprogramme für das einzusetzende automatisierte Verfahren geeignet sind, lässt sich nur mit Kenntnis des Verfahrens entscheiden. Der zweite Teil der Arbeit befasst sich unabhängig von einem konkreten Verfahren da-

mit, Programme unter Erhaltung der Semantik zu transformieren, sodass günstigere Zerlegungen gefunden werden können. Es wird ein Kriterium entwickelt, mit dem effizient nachgewiesen werden kann, dass zwei Programmsequenzen semantisch äquivalent sind. Eine möglicherweise bestehende Abhängigkeit der Ein- und Ausgabeoperationen untereinander wird dabei berücksichtigt. Damit ist es nun möglich, ein Eingabeprogramm in eine für das eingesetzte Verfahren geeignete Form zu transformieren. Die semantische Äquivalenz der Transformation kann mit dem vorgestellten Kriterium leicht überprüft werden.

Während sich Teil I mit imperativen Programmiersprachen im Allgemeinen (und einer konkreten objektorientierten Sprache im Besonderen) beschäftigt, werden in Teil II imperative Programme einer konkreten Sprache betrachtet. Die Ergebnisse sind unabhängig voneinander, die beiden Teile nehmen daher keinen direkten Bezug aufeinander. Eine kombinierte Anwendung der Resultate ist allerdings möglich. Imperative Programme können vor einer Analyse nach dem in Teil I dargelegten Verfahren zerlegt werden und die Zerlegung mit den Methoden aus Teil II auf Korrektheit überprüft werden. Es sei jedoch darauf hingewiesen, dass die in den jeweiligen Kapiteln verwendeten Sprachen aus den geschilderten technischen Gründen disjunkt sind.

Teil I

Terminierungsanalyse imperativer Sprachen

2 Überblick über Teil I

Grundlegend für den ersten Teil der vorliegenden Arbeit sind zwei Beobachtungen über die rein funktionale Sprache Haskell. Zum einen gestaltet sich die Verifikation von Haskell-Programmen aufgrund der referentiellen Transparenz für gewöhnlich einfacher als die ihrer imperativen Gegenstücke. Insbesondere existiert mit APROVE (siehe [GTSKF04]) bereits ein fortschrittliches, automatisiertes Verfahren zur Terminierungsanalyse für Haskell-Programme. Dieses ist laut [GRSK+11] für weite Teile der Haskell-Standardbibliothek in der Lage, die Terminierungseigenschaften korrekt zu bestimmen. Es beruht auf dem Dependency Pair Framework, einer Technik zur Terminierungsanalyse für Termersetzungssysteme. Die zweite Beobachtung über Haskell ist, dass es sich hervorragend zur Simulation von imperativen Programmiersprachen eignet. Dies resultiert aus Haskells Unterstützung des monadischen Programmieransatzes, mit dem sich nicht nur leicht Parser, sondern auch Ausführungsumgebungen konstruieren lassen, die die Zustandswechsel imperativer Programme imitieren. Haskell ist damit nicht nur zur Konstruktion von Übersetzern, sondern auch als Zielplattform eine geeignete Wahl.

Aus den beiden Beobachtungen ergibt sich, dass die Kombination aus einem Übersetzer für eine imperative Sprache mit Haskell als Zielsprache und dem bestehenden Verfahren zur Terminierungsanalyse ein Verfahren für eine imperative Sprache bildet, welches sich verhältnismäßig leicht konstruieren lässt. A priori ist jedoch nicht klar, wie die Güte eines solchen Verfahrens ausfällt.

Kapitel 3 führt Termersetzungssysteme ein und definiert den Terminierungsbegriff anhand abstrakter Reduktionssysteme. Dieses Kapitel stellt die Grundlage für die Einführung des in APROVE eingesetzten Dependency Pair Frameworks dar. In Kapitel 4 wird dieses Framework selbst eingeführt. Das Dependency Pair Framework erlaubt es, verschiedenste Methoden zur Terminierungsanalyse von Termersetzungssystemen zusammenzuführen, indem aus einem Termersetzungssystem mehrere, getrennt voneinander zu lösende Probleme erzeugt werden. In Kapitel 5 wird ein für die Sprache Haskell geeigneter Terminierungsbegriff eingeführt und gezeigt, wie APROVE aus einem Haskell-Programm Eingaben für das Dependency Pair Framework erzeugt. Zentral ist hierfür der Terminierungsgraph, der einer symbolischen Auswer-

tung eines Haskellausdrucks entspricht.

Zuletzt werden in Kapitel 6 zwei Ansätze diskutiert, um eine imperative Sprache in ein semantisch äquivalentes Haskell-Programm zu übersetzen. Der erste Ansatz beruht auf Zustandstransitionsmonaden. Ist die Zustandsübergangsrelation einer Sprache, also deren operationelle Semantik, bekannt, so können die Übergänge monadisch aufgefasst und somit jegliches Programm dieser Sprache simuliert werden. Es werden zwei Probleme beschrieben, die bei der Terminierungsanalyse auftreten können. Zum einen kann die Verwendung von Kombinatoren dazu führen, dass der erzeugte Terminierungsgraph nicht genügend Informationen trägt. Es wird eine Programmtransformation eingeführt, die die Erzeugung eines günstigeren Terminierungsgraphen erzwingt. Ein zweites Problem ist die Simulation des Variablenkonzepts, die sich für die Terminierungsanalyse als problematisch erweist.

Der zweite Ansatz zur Übersetzung nutzt die Korrespondenz zwischen der statischen Einmal-Zuweisungsform und dem Continuation-Passing-Style aus. Auf diese Weise kann ein imperatives Programm in ein Haskell-Programm überführt werden, das Variablen nicht simuliert, sondern als Parameter darstellt. Am Beispiel einer Java-ähnlichen Sprache wird demonstriert, dass jedoch Elemente verbleiben, die simuliert werden müssen. Für diese gilt aus ähnlichen Überlegungen, dass damit eine Terminierungsanalyse nach dem beschriebenen Verfahren nicht möglich ist.

3 Termersetzungssysteme

Ein Termersetzungssystem ist eine Konstruktion, in der Terme nach bestimmten Regeln verändert werden. Eine Veränderung ist dabei immer eine Ersetzung eines Teils eines Terms durch einen anderen Term, der so zum neuen Teilterm wird. Im Folgenden werden diese Begriffe definiert. Im Wesentlichen werden die Notationen und die Definitionen aus [BK03] und [KdV03a] verwendet.

3.1 Definition

3.1.1 Terme über Signaturen

Definition 3.1. *Eine* Signatur Σ *ist eine Menge von Funktionssymbolen* $f, g, \ldots$. *Für jedes Funktionssymbol gibt es eine feste Anzahl an Argumenten, die sogenannte* Stelligkeit *oder* Arität. *Die Funktion* $\mathrm{ari} : \Sigma \to \mathbb{N}$ *gibt diese Stelligkeit an. Funktionssymbole mit Arität Null heißen* Konstanten *und werden mit c bezeichnet.*

Die im Folgenden betrachteten Terme sind über eine feste Signatur definiert. Neben den Funktionssymbolen enthalten sie auch Variablen, welche mit x, y, z bezeichnet werden, gegebenenfalls auch als x', y', z' oder mit natürlichen Zahlen indiziert. Die Menge der Variablen und die Signatur sind disjunkt. Terme haben eine baumartige Struktur. Jeder Knoten ist mit einem Funktionssymbol aus der Signatur oder einer Variable markiert. Jeder Knoten hat der Arität des Funktionssymbols entsprechend viele Kinder. Variablen und Konstanten bilden die Blätter des Baums. Dieses Grundkonzept lässt sich wie folgt formalisieren.

Definition 3.2. *Die* Menge aller Terme $\mathrm{Ter}(\Sigma, \mathrm{Var})$ *über einer Signatur* Σ *und die Variablenmenge* Var *wird induktiv definiert.*

(i) Alle Variablen und Konstanten sind Terme und in $\mathrm{Ter}(\Sigma, \mathrm{Var})$ *enthalten.*

(ii) Für gegebene n *Terme* $t_1, \ldots, t_n$ *über* Σ *und ein* n*-stelliges Funktionssymbol* f *aus* Σ*, ist* $f(t_1, \ldots, t_n)$ *ein Term über* Σ*.*

Die Menge $\mathrm{Ter}(\Sigma, \mathrm{Var})$ *ist die kleinste Menge, sodass alle Terme nach (i) und (ii) enthalten sind.*

In Anlehnung an die Baumstruktur wird das Symbol f im Term $t = f(t_1, \ldots, t_n)$ als Wurzelsymbol *bezeichnet und mit* $\mathrm{root}(t)$ *notiert. Ein Term* beginnt *mit seinem Wurzelsymbol. Die Terme $t_1, \ldots, t_n$ sind die* Argumente *des Terms t.*

Ist die Menge der Variablen aus dem Kontext ersichtlich, wird vereinfachend $\mathrm{Ter}(\Sigma)$ verwendet. Die syntaktische Gleichheit zweier Terme s und t wird mit $s \equiv t$ notiert. Terme, in denen jede Variable höchstens einmal verwendet wird, heißen *linear*.

Die Menge der Variablen in einem Term t wird mit $\mathrm{Var}(t)$ bezeichnet. Terme ohne Variablen heißen *geschlossene Terme*, die Menge aller solcher Terme wird mit $\mathrm{Ter}_0(\Sigma) = \mathrm{Ter}(\Sigma, \emptyset)$ notiert. Für einen Term t, für den $\mathrm{Var}(t) = \emptyset$ gilt, folgt $t \in \mathrm{Ter}_0(\Sigma)$.

Die Länge eines Terms t, notiert als $|t|$, wird induktiv definiert. Für die Länge einer Konstanten c oder einer Variablen x gilt $|c| = |x| = 1$. Die Länge aller anderen Terme ergibt sich aus der Summe der Längen aller Argumente, erhöht um 1. Somit gilt $|f(t_1, \ldots, t_2)| = 1 + |t_1| + \ldots + |t_n|$.

Gelegentlich ist zwischen k verschiedenen Termen mit einem festen Wurzelsymbol f mit Arität n zu differenzieren, die sich lediglich in den Argumenten $t_{i_1}, \ldots, t_{i_n}$ mit $1 \leq i \leq k$ unterscheiden. Es bietet sich die Vektornotation an, um die Gesamtheit der Argumente eines Operationssymbols mit einem Index versehen zu können. Daher wird $t_{1_1}, \ldots, t_{1_n}$ als $\vec{t_1}$ notiert oder auch $t_1, \ldots, t_n$ als $\vec{t}$.

Zusätzlich zu den Definitionen in [KdV03a] definieren wir noch die Tiefe eines Terms auf folgende Weise.

Definition 3.3. *Die* Tiefe *eines Terms wird als $\delta : \mathrm{Ter}(\Sigma) \to \mathbb{N}$ induktiv definiert mit:*

$$\begin{aligned} \delta(x) &:= 1 \\ \delta(c) &:= 1 \\ \delta(f(t_1, \ldots, t_n)) &:= 1 + \max(\delta(t_1), \ldots, \delta(t_n)) \end{aligned}$$

Die Menge der maximal tiefen Teilterme $M(t)$ *eines Terms t wird induktiv definiert*

als

$$M(x) := \{x\}$$
$$M(c) := \{c\}$$
$$M(f(t_1, \ldots, t_n)) := \{f(t_1, \ldots, t_n)\} \cup \bigcup_{i \in I} M(t_i)$$

wobei $I = \{i | 1 \leq i \leq n, \delta(t_i) = \max(\delta(t_1), \ldots, \delta(t_n))\}$.

3.1.2 Kontexte

Um über die Struktur eines Terms präzise Aussagen zu treffen, ist das Konstrukt eines Kontextes nützlich. Dabei handelt es sich um einen unvollständigen Term, der genau ein Loch[1] enthält. Formal wird dieses mit dem Symbol $\Box$ repräsentiert. Die folgende Definition weicht von [KdV03a] ab, wo Kontexte mit mehreren Löchern als Terme über einer erweiterten Signatur gewählt werden.

Definition 3.4. Kontexte*, notiert mit* $C[]$*, werden induktiv definiert.*

(i) $C[] \equiv \Box$ *ist ein Kontext (der* triviale Kontext*).*

(ii) $C[] \equiv f(t_1, \ldots, t_n)$ *ist genau dann ein Kontext, wenn genau ein Index* i *mit* $1 \leq i \leq n$ *existiert, sodass* t_i *ein Kontext ist und für* $j \neq i : t_j$ *ein Term über* Σ *ist.*

Mit $C[s]$ *wird der Term beschrieben, der sich aus dem Ersetzen des Lochs durch den Term* $s \in \mathrm{Ter}(\Sigma)$ *ergibt. Ist* $t = C[s]$ *so heißt* $C[]$ *Präfix von* t*, notiert als* $C[] \ggg t$*.*

Definition 3.5. *Ein Term* s *ist ein* Teilterm *eines Terms* t *genau dann, wenn es einen Kontext* $C[]$ *gibt, sodass* $t \equiv C[s]$ *gilt. Die Notation* $s \sqsubseteq t$ *wird verwendet, um* s *als Teilterm von* t *zu kennzeichnen. Aufgrund des trivialen Kontextes gilt für alle Terme* $t \sqsubseteq t$*. Die Notation* $t' \sqsubset t$ *ist äquivalent zu* $t' \sqsubseteq t \wedge t' \not\equiv t$ *und beschreibt* echte Teilterme*.*

Die Tiefe eines Kontextes $C[]$ wird analog zur Tiefe eines Terms definiert. Diese sagt jedoch wenig über die Tiefe eines Terms der Form $C[t]$ aus. Offensichtlich stellt sie nur ein Minimum für die Größe des sich aus der Einbettung in den Kontext ergebenden Terms dar. Es stellen sich die Fragen, wie tief ein Term sein muss, damit diese minimale Größe überschritten wird und um wie viel die Tiefe für einen

[1] In der Literatur werden teilweise mehrere Löcher zugelassen.

hinreichend tiefen Term zunimmt. Mit der folgenden, zusätzlich zu den Definitionen aus [KdV03a] definierten Größe lassen sich beide Fragen beantworten.

Definition 3.6. *Der* Versatz *des Kontextes $C[]$ wird definiert als*

$$\xi(C[]) = \delta(C[t]) - \delta(t)$$

für ein beliebiges $t \in \mathrm{Ter}(\Sigma)$ mit $\delta(C[t]) > \delta(C[])$.

Beispiel 3.7. *Sei $C[] \equiv f(g(g(c)), \Box)$. Es ergeben sich $\delta(C[]) = 4$ und $\xi(C[]) = 1$. Terme mit einer echt größeren Tiefe als $\delta(C[]) - \xi(C[]) = 3$ wachsen also bei Einsatz im Kontext $C[]$ um $\xi(C[])$.*

Lemma 3.8. *Der Versatz $\xi(C[])$ ist unabhängig vom gewählten t und daher wohldefiniert.*

Zum Beweis wird der folgende Hilfssatz benötigt.

Lemma 3.9. *Sei $D[]$ ein beliebiger Kontext und f ein beliebiges Funktionssymbol. Für $C[] = f(\dots, D[], \dots)$ und einen beliebigen Term $t \in \mathrm{Ter}(\Sigma)$ gilt*

$$\delta(C[t]) > \delta(C[]) \implies \delta(D[t]) > \delta(D[]).$$

Beweis. (Lemma 3.9) Durch Einsetzen der Definition der Tiefe in die Voraussetzung des Lemmas ergibt sich

$$\delta(C[t]) > \delta(C[]) \implies 1 + \max(\dots, \delta(D[t]), \dots) > 1 + \max(\dots, \delta(D[]), \dots).$$

Außerdem muss

$$\max(\dots, \delta(D[t]), \dots) = \delta(D[t])$$

gelten, denn sonst würde es ein Argument $s \not\equiv D[t]$ von f in $C[]$ geben, sodass $\delta(s) > \delta(D[t])$. Da s aber auch in $C[]$ vorhanden ist, müsste $\delta(C[t]) = \delta(C[])$ im Widerspruch zur Voraussetzung gelten. Es gilt also

$$\delta(D[t]) > \max(\dots, \delta(D[]), \dots)$$

und da $\max(\dots, \delta(D[]), \dots) \geq \delta(D[])$ folgt

$$\delta(D[t]) > \delta(D[]).$$

□

Beweis. (Lemma 3.8) Um zu zeigen, dass für einen festen Kontext $C[]$ der Versatz $\xi(C[])$ unabhängig von t ist, muss bewiesen werden, dass $\xi(C[])$ für jede Wahl von t konstant ist, also

$$\forall t \in \text{Ter}(\Sigma) : \delta(C[t]) > \delta(C[]) \implies \delta(C[t]) - \delta(t) = n_{C[]}$$

für ein konstantes $n_{C[]} \in \mathbb{N}$. Der Induktionsbeweis folgt der Struktur der Kontexte. Für $C[] \equiv \Box$ gilt $C[t] \equiv t$ und somit $\delta(C[t]) - \delta(t) = 0$ für alle Terme t. Dies dient als Induktionsbasis.

Sei nun $C[] \equiv f(\ldots, D[], \ldots)$. Die Induktionsvoraussetzung ist, dass ein $n_{D[]}$ existiert, sodass für alle $t \in \text{Ter}(\Sigma)$ die Implikation $\delta(D[t]) > \delta(D[]) \implies \delta(D[t]) - \delta(t) = n_{D[]}$ gilt. Zu zeigen ist nun, dass für alle t aus $\delta(C[t]) > \delta(C[])$ folgt, dass die Differenz $\delta(C[t]) - \delta(t)$ konstant ist. Durch Einsetzen ergibt sich

$$\delta(C[t]) - \delta(t) = \delta(f(\ldots, D[t], \ldots)) - \delta(t) = 1 + max(\ldots, \delta(D[t]), \ldots) - \delta(t).$$

Aus $C[] \equiv f(\ldots, D[], \ldots)$ und $\delta(C[t]) > \delta(C[])$ ergibt sich, dass für alle Argumente s von f in $C[]$ die Tiefe $\delta(s)$ durch $\delta(D[t])$ beschränkt ist. Wäre dies nicht der Fall, würde analog zu der Argumentation im Beweis von Lemma 3.9 $\delta(C[]) = 1 + \delta(s)$ für ein Argument s gelten und somit $\delta(C[]) = \delta(C[t])$ im Widerspruch zur Annahme $\delta(C[t]) > \delta(C[])$. Damit muss

$$max(\ldots, \delta(D[t]), \ldots) = \delta(D[t])$$

sein. Nach Lemma 3.9 gilt daher auch $\delta(D[t]) > \delta(D[])$, womit sich die Induktionsvoraussetzung einsetzen lässt und somit

$$\delta(C[t]) - \delta(t) = 1 + \delta(D[t]) - \delta(t) = 1 + n_{D[]} = n_{C[]}$$

ergibt. □

3.1.3 Substitutionen

Eine Substitution ist eine Ersetzungsvorschrift. Sie kann vollständig durch eine Funktion charakterisiert werden.

Definition 3.10. *Eine* Substitution *ist eine Funktion σ, die aus der Menge der*

Variablen in die Menge der Terme abbildet:

$$\sigma : \mathrm{Var} \to \mathrm{Ter}(\Sigma)$$

Die Funktion σ wird auf folgende Weise zu einer Funktion auf Termen erweitert, die ebenfalls als Substitution σ bezeichnet wird:

$$\sigma(F(t_1, \dots, t_n)) \equiv F(\sigma(t_1), \dots, \sigma(t_n))$$

Die Anwendung der Substitution σ auf einen Term t wird als t^σ notiert. Eine Substitution σ heißt geschlossene Substitution *genau dann, wenn für alle Variablen x entweder $\sigma(x) = x$ oder $\sigma(x) = t$ mit $t \in \mathrm{Ter}_0(\Sigma)$ gilt.*

In vielen Fällen werden Substitutionen lediglich für endlich viele Variablen x von der Identität abweichen. Die Notation $[x_1, \dots, x_n := s_1, \dots, s_n]$ kann dann verwendet werden, um die Substitution σ zu beschreiben, die für $1 \leq i \leq n$ die Variablen x_i auf s_i abbildet und sonst der Identität entspricht.

Definition 3.11. *Der Term t* subsumiert *den Term s genau dann, wenn es eine Substitution σ gibt, sodass $s \equiv t^\sigma$ gilt. Dabei heißt s* Instanz *von t. Gilt $s^\sigma \equiv t^\sigma$ für eine Substitution σ, so s und t* unifizierbar *und σ ist der* Unifikator.

3.2 Ersetzungsregeln

Ein Termersetzungssystem ersetzt Teilterme nach bestimmten Regeln. Diese Regeln werden formal wie folgt definiert.

Definition 3.12. *Eine Paar $\langle l, r\rangle$ von Termen über einer Signatur Σ heißt* Ersetzungsregel *genau dann, wenn $l \not\equiv x$ für alle $x \in \mathrm{Var}$ und $\mathrm{Var}(r) \subseteq \mathrm{Var}(l)$ gilt, also r keine Variablen außer den in l vorkommenden enthält. Der Term l heißt die* linke Seite*, der Term r die* rechte Seite *der Ersetzungsregel. Ersetzungsregeln werden in der Form $l \to r$ notiert. Ersetzungsregeln können auch benannt werden und werden dann in der Form $\rho : l \to r$ aufgeschrieben, wobei ρ der Name ist.*

Die Einschränkung des Vorkommens von Variablen auf der rechten Seite hat folgenden Hintergrund. Die Variablen in den Ersetzungsregeln sind nicht an die Variablen in den Termen gebunden, auf denen die Regeln angewendet werden. Vielmehr ist eine Variable in einer Reduktionsregel ein Platzhalter für einen beliebigen Teilterm (wohingegen die Bedeutung einer Variablen in einem Term nicht von Belang ist).

Die Variablen in den Reduktionsregeln haben somit ihren eigenen Gültigkeitsbereich. Durch den Abgleich der linken Seite der Regel mit dem tatsächlich zu verändernden Term ergibt sich die Belegung der Variablen und somit der reduzierte Term. Würde die rechte Seite neue Variablen enthalten, so ließe sich keine Belegung für sie finden. Die formale Funktionsweise der Regeln wird mit einer Substitution definiert.

Definition 3.13. *Eine Regel $\rho : l \to r$ wird durch beidseitige Anwendung einer Substitution σ* instantiiert*. Es ergibt sich ein* atomarer Ersetzungsschritt *der Form $l^\sigma \to_\rho r^\sigma$. Dabei wird l^σ als* Redex *und r^σ als* Kontraktum *bezeichnet.*

Die Verwendung einer Substitution erlaubt, dass die Variablen einer Regel nicht zu den Variablen eines Terms disjunkt sein müssen. Das folgende Beispiel illustriert dies. Gegeben seien die Regel $\rho : F(x) \to G(x, F(x))$ und der Term $t \equiv F(H(x))$. Durch die Wahl der Substitution $[x := H(x)]$ ergeben sich der Redex $F(H(x))$, welcher syntaktisch identisch zu t ist, und das Kontraktum $G(H(x), F(H(x))$. Hierbei ist zu beachten, dass die Regel $\rho' : F(y) \to G(y, F(y))$, welche sich aus ρ ergibt, indem x in y unbenannt wird, mit der Substitution $[y := H(x)]$ zu exakt demselben atomaren Ersetzungsschritt führt. Dies verdeutlicht das Konzept der verschiedenen Sichtbarkeitsbereiche.

Die Anwendung eines solchen atomaren Ersetzungsschritts soll auch dann zulässig sein, wenn der Redex nur ein Teilterm des vollständigen Terms ist. Ein solcher Schritt heißt *Ersetzungsschritt*.

Definition 3.14. *Die Anwendung einer atomaren Ersetzung gemäß einer Regel ρ in einem beliebigen Kontext $C[]$ liefert den* Ersetzungsschritt $C[l^\sigma] \to_\rho C[r^\sigma]$*. Die Relation $\to_\rho$ ist die durch ρ erzeugte Ersetzungsrelation.*

Es ergibt sich damit, dass, sollte ein Redex mehrfach als Teilterm eines Terms enthalten sein, ein Ersetzungsschritt nur einen dieser Redexe ersetzt und zwar den durch den Kontext $C[]$ spezifizierten.

Definition 3.15. *Ein Termersetzungssystem (TRS) $\mathcal{R}$ besteht aus einer Signatur Σ und einer Menge R von Ersetzungsregeln. Es ergibt sich eine Ersetzungsrelation $\to_R = \bigcup\{\to_\rho |\rho \in R\}$*

Ist das Termersetzungssystem aus dem Kontext bekannt, kann der Index $_R$ bei der Ersetzungsrelation entfallen.

3.3 Terminierung

Die notwendigen Begriffe, um die Terminierungseigenschaften von Termersetzungssystemen zu beschreiben, ergeben sich aus den abstrakten Reduktionssystemen, die im Folgenden kurz eingeführt werden. Termersetzungssysteme stellen eine Instanz der allgemeiner gefassten abstrakten Reduktionssysteme dar.

3.3.1 Abstrakte Reduktionssysteme und einige ihrer Eigenschaften

Definition 3.16. *Ein* abstraktes Reduktionssystem $\mathcal{A}$ *ist ein Paar aus einer Trägermenge* A *und einer Menge von Relationen* $R \subseteq \mathcal{P}(A \times A)$. *Die Notation ist* $\mathcal{A} = (A, R)$ *oder vereinfachend* $\mathcal{A} = (A, \rightarrow)$ *falls* $R = \{\rightarrow\}$.

Wir notieren die transitive Hülle der Relation $\rightarrow$ mit $\rightarrow^+$, die reflexiv-transitive Hülle mit $\twoheadrightarrow$ und die Äquivalenzrelation, die aus $\rightarrow$ erzeugt wird, mit $\leftrightarrow^*$. Letzteres ist eine Abweichung von der Notation in [BK03].

Definition 3.17. *Eine Reduktionsfolge ist eine möglicherweise unendliche Folge aus Elementen* $a_1, a_2, \ldots \in A$, *sodass* $a_1 \rightarrow a_2 \rightarrow \ldots$ *gilt.*

Ein Termersetzungssystem kann als abstraktes Reduktionssystem betrachtet werden, bei dem die Grundmenge als $\mathrm{Ter}(\Sigma)$ gewählt wird und die Menge der Relationen lediglich die Ersetzungsrelation $\rightarrow$ umfasst, also $(\mathrm{Ter}(\Sigma), \rightarrow)$. Im Folgenden wird nicht explizit zwischen beiden Auffassungen unterschieden. Für abstrakte Reduktionssysteme lassen sich eine Reihe von Eigenschaften formulieren, wie z.B. die Church-Rosser Eigenschaft. Von Belang sind in der vorliegenden Arbeit nur die Eigenschaften, die mit der Terminierung in Zusammenhang stehen.

Definition 3.18. *Sei* $\mathcal{A} = (A, R)$ *ein abstraktes Reduktionssystem und* $\rightarrow = \bigcup R$ *Ein Element* $a \in A$ *heißt* Normalform, *falls es kein Element* $b \in A$ *gibt, sodass* $a \rightarrow b$ *gilt,* a *also keinen Nachfolger besitzt. Ein* $a \in A$ *heißt* schwach normalisierend *genau dann, wenn es eine Normalform* b *gibt, sodass* $a \twoheadrightarrow b$ *gilt. Ein abstraktes Reduktionssystem heißt* schwach normalisierend, *wenn alle Elemente der Trägermenge schwach normalisierend sind. Ein* $a \in A$ *heißt* stark normalisierend *genau dann, wenn alle Reduktionsfolgen, die mit* a *beginnen, endlich sind. Ein abstraktes Reduktionssystem heißt* terminierend *oder* stark normalisierend, *wenn alle Elemente stark normalisierend sind.*

3.3.2 Beweistechniken zum Nachweis der Terminierungseigenschaft von Termersetzungssystemen

Termersetzungssysteme erweisen sich als Turing-vollständig, d.h. es gibt für jede Turingmaschine ein Termersetzungssystem, das selbige simuliert. Der älteste Beweis dafür stammt von Huet und Lankford [HL78]. Durch die Unentscheidbarkeit des Halteproblems für Turingmaschinen ergibt sich, dass es im Allgemeinen nicht entscheidbar ist, ob ein Termersetzungssystem terminierend ist. Es gibt allerdings eine Reihe von Beweistechniken, mit denen die Terminierungseigenschaft im Einzelfall nachgewiesen werden kann. Zugrunde liegt hierbei das folgende Lemma.

Lemma 3.19. *Ein Termersetzungssystem* (Σ, R) *mit der Ersetzungsrelation* $\rightarrow$ *ist genau dann terminierend, wenn es eine wohlfundierte Striktordnung* $>$ *gibt, sodass für alle* $t, u \in \mathrm{Ter}(\Sigma)$ *mit* $t \rightarrow u$ *gilt* $t > u$.

Die Definition einer Striktordnung kann Anhang A entnommen werden.

Beweis. In einem terminierenden Termersetzungssystem kann die transitive Hülle $\rightarrow^+$ der Ersetzungsrelation $\rightarrow$ als wohlfundierte Striktordnung gewählt werden. Die transitive Hülle $\rightarrow^+$ bildet eine Striktordnung, da sie sowohl transitiv als auch irreflexiv ist. Die Irreflexivität folgt aus der Terminierungseigenschaft. Gäbe es einen Term $t \in \mathrm{Ter}(\Sigma)$ mit $t \rightarrow t$, so wäre t nicht stark normalisierend und (Σ, R) kein terminierendes Termersetzungssystem. Ferner ist $\rightarrow^+$ wohlfundiert. Gäbe es eine unendliche $\rightarrow^+$-absteigende Folge, so gäbe es auch eine unendliche $\rightarrow$-absteigende Folge und somit wären sämtliche Glieder der Folge nicht stark normalisierend und (Σ, R) kein terminierendes Termersetzungssystem. Als Striktordnung kann daher $\rightarrow^+$ gewählt werden. Umgekehrt folgt aus der Existenz einer wohlfundierten Ordnung, die für alle $t > u$ auch $t \rightarrow u$ impliziert, dass es keine unendlichen Ersetzungssequenzen geben kann. □

Beweistechniken lassen sich danach klassifizieren, ob sie sich für eine Automatisierung anbieten oder nicht.

Zur Automatisierung wird häufig folgendes Lemma genutzt.

Definition 3.20. *Eine wohlfundierte Striktordnung* $>$ *auf* $\mathrm{Ter}(\Sigma)$ *heißt* Reduktionsordnung *genau dann, wenn* $t > u$ *für eine beliebige Substitution* σ *impliziert, dass* $t^\sigma > u^\sigma$, *und für einen beliebigen Kontext* $C[]$ *folgt, dass* $C[t] > C[u]$. *Eine Reduktionsordnung heißt* kompatibel *mit einem Termersetzungssystem* (Σ, R) *genau dann, wenn für alle Regeln* $\rho \in R : l \rightarrow r$ *gilt* $l > r$.

Lemma 3.21. *Zu jedem Termersetzungssystem* (Σ, R) *existiert genau dann eine kompatible Reduktionsordnung* $>$*, wenn es terminiert.*

Beweis. In einem terminierenden Termersetzungssystem ist die transitive Hülle der Ersetzungsrelation wohlfundiert und kann als Reduktionsordnung gewählt werden. Die Bedingungen an die Abgeschlossenheit unter Kontexteinbettung und Substitution folgen aus der Konstruktion. Umgekehrt folgt gerade aus der Abgeschlossenheit der Reduktionsordnung, dass auch bei allen Ersetzungsschritten $C[l^\sigma] \to C[r^\sigma]$ gilt, dass $C[l^\sigma] > C[r^\sigma]$. Aufgrund der Wohlfundiertheit von $>$ muss das Termersetzungssystem terminieren. □

Automatische Methoden setzen nun an, anhand der Ersetzungsregeln eine kompatible Reduktionsordnung zu berechnen. Bekannte Methoden sind die rekursive Pfadordnung (z.B. nach Dershowitz, siehe [Der82]) oder die Knuth-Bendix-Pfadordnung (siehe [KB69]). Diese Methoden orientieren sich an syntaktischen Eigenschaften und werden daher in [Zan03] als solche bezeichnet. Sie sind zu trennen von den semantischen Methoden, die eine Algebra zur Interpretation des Termersetzungssystem verwenden und auf diese Weise eine wohlfundierte Striktordnung finden, wie sie in Lemma 3.19 beschrieben wird. Eine dritte Kategorie bilden die Transformationsmethoden. Hierbei ist das Ziel, ein Termersetzungssystem in ein anderes zu überführen, welches sich für die Analyse mit syntaktischen Mitteln eignet. Dabei muss natürlich die Terminierungseigenschaft beibehalten werden, d.h. das transformierte System darf nur genau dann terminieren, wenn das Eingabesystem ebenfalls terminiert. Eine der neueren Methoden ist das Dependency-Pair-Framework (siehe [GTSK05]), das sich ebenfalls in die Reihe von Transformationsmethoden einordnet. Es ist im Folgenden von Bedeutung und wird in Kapitel 4 eingehend vorgestellt.

3.3.3 Modularität

Die Zerlegung von Problemen in besser beherrschbare Teilprobleme ist ein gängiger Ansatz in Mathematik und Informatik. Es scheint daher nahezuliegen, Termersetzungssysteme in mehrere, kleinere Termersetzungssysteme zu zerlegen und deren Terminierungseigenschaften zu analysieren, um Rückschlüsse über das ursprüngliche Termersetzungssystem zu ermöglichen. Eine mögliche Zerlegung ist die disjunkte Zerlegung.

Definition 3.22. *Eine Menge von Termersetzungssystemen* $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ *mit* $\mathcal{R}_i = (\Sigma_i, R_i)$ *ist eine* disjunkte Zerlegung *eines Termersetzungssystems* $\mathcal{R} = (\Sigma, R)$ *genau*

dann, wenn

$$\bigcup_{1 \leq i \leq n} \Sigma_i = \Sigma \land \bigcup_{1 \leq i \leq n} R_i = R$$

und

$$\forall 1 \leq i < j \leq n : \Sigma_i \cap \Sigma_j = \emptyset \land R_i \cap R_j = \emptyset,$$

gelten.

Die Termersetzungssysteme $\mathcal{R}_1, \ldots, \mathcal{R}_n$ *heißen dabei* Teiltermersetzungssysteme *einer disjunkten Zerlegung von* $\mathcal{R}$*. Zwei Termersetzungssysteme* $\mathcal{R}_1 = (\Sigma_1, R_1)$ *und* $\mathcal{R}_2 = (\Sigma_2, R_2)$ *heißen* disjunkt*, wenn* $\{\mathcal{R}_1, \mathcal{R}_2\}$ *eine disjunkte Zerlegung des Termersetzungssystems* $(\Sigma_1 \cup \Sigma_2, R_1 \cup R_2)$ *bilden. Das Termersetzungssystem* $\mathcal{R}$ *heißt* disjunkte Vereinigung *der Termersetzungssysteme* $\mathcal{R}_1, \ldots, \mathcal{R}_n$*.*

In [Der81] stellt Dershowitz die These auf, dass, falls zwei disjunkte Termersetzungssysteme terminierend sind, auch die Vereinigung terminierend ist. Das folgende Beispiel nach Toyama (siehe [Toy87]) demonstriert, dass die Terminierungseigenschaft nicht modular unter Vereinigung disjunkter Termersetzungssysteme ist und widerlegt die These von Dershowitz.

Toyamas Gegenbeispiel 3.23. *Das Gegenbeispiel besteht aus den beiden Termersetzungssystemen* $\mathcal{R}_1 = (\Sigma_1, \{\rho_1\})$ *mit*

$$\rho_1 : f(0, 1, x) \to f(x, x, x)$$

und $\mathcal{R}_2 = (\Sigma_2, \{\rho_2, \rho_3\})$ *mit*

$$\rho_2 : g(x, y) \to x$$

$$\rho_3 : g(x, y) \to y.$$

Die Signaturen Σ_1 *und* Σ_2 *werden dabei nicht vollständig definiert. Offensichtlich gilt* $\{0, 1, f\} \subseteq \Sigma_1$ *und* $\{g\} \subseteq \Sigma_2$*, sowie* $\mathrm{ari}(f) = 3$ *und* $\mathrm{ari}(g) = 2$*. Ferner wird verlangt, dass die Signaturen disjunkt sind.*

Für die den beiden Termersetzungssystemen zugrunde liegenden Signaturen Σ_1 und Σ_2 wird zusätzlich verlangt, dass sie disjunkt sind. Dies ist als weitere Vereinfachung gedacht, denn, wie Baader und Nipkow in Kapitel 9 aus [BN99] konstatieren, ist es offensichtlich, dass je weniger „Interaktion“ es zwischen zwei Systemen gibt, sie sich umso einfacher kombinieren lassen. Baader und Nipkow weisen auch darauf hin, dass die Analyse der disjunkten Vereinigung selbst für den Fall disjunkter Systeme bei weitem nicht trivial ist.

Beide Termersetzungssysteme aus Toyamas Beispiel terminieren. Toyama gibt in [Toy87] keinen Beweis dafür an. Auch in [KdV03b] findet sich lediglich die Behauptung. Mit Kenntnis entsprechender Techniken lassen sich allerdings leicht Beweise finden. So gibt es für $\mathcal{R}_1$ in [BN99] einen Hinweis auf eine geeignete Multimenge. In [Ohl02] findet sich ein Beweis, der mit den in Kapitel 4 vorgestellten Abhängigkeitspaaren geführt wird. In der vorliegenden Arbeit wird für beide Termersetzungssysteme jeweils ein Beweis angegeben, der ohne fortgeschrittene Techniken auskommt.

Es wird mit dem kürzeren Beweis für $\mathcal{R}_2$ begonnen. Sei $g \in \mathrm{Ter}(\Sigma)$ das Symbol aus Toyamas Beispiel. Wir definieren $k : \mathrm{Ter}(\Sigma) \to \mathbb{N}$ induktiv durch

$$\begin{aligned} k(x) &:= 0 \\ k(c) &:= 0 \\ k(g(t_1, t_2)) &:= 1 + k(t_1) + k(t_2) \\ k(h(t_1, \dots, t_n)) &:= \sum_{i=1}^{n} k(t_i) \text{ für alle } h \in \Sigma \text{ mit } h \not\equiv g. \end{aligned}$$

Es ergibt sich die Relation $>_g$ mit

$$s >_g t \iff k(s) > k(t)$$

Lemma 3.24. *Die Striktordnung $>_g$ ist wohlfundiert mit $s >_g t$ falls $s \to t$.*

Beweis. Wäre $>_g$ nicht wohlfundiert, gäbe es eine unendliche Kette der Form $s_1 >_g s_2 >_g \dots$ und damit auch unendlich viele Zahlen, die strikt kleiner sind als $k(s_1)$. Aus dem Widerspruch folgt die Wohlfundiertheit.

Ferner gilt $s \to t \implies s >_g t$, denn aus $s \to t$ folgt die Existenz eines Kontextes $C[]$ und einer Substitution σ, sodass $s \equiv C[g(x, y)^\sigma]$ und abhängig von der zur Geltung kommenden Regel $t \equiv C[x^\sigma]$ oder $t \equiv C[y^\sigma]$. Offensichtlich gilt $g(x, y)^\sigma >_g x^\sigma$ bzw. $g(x, y)^\sigma >_g y^\sigma$. Da $>_g$ nach Konstruktion abgeschlossen unter Kontexteinbettung ist, folgt $s >_g t$. □

Korollar 3.25. *Das Termersetzungssystem $\mathcal{R}_2$ aus Toyamas Gegenbeispiel terminiert.*

Beweis. Gemäß 3.24 gilt $s >_g t$ falls $s \to t$. Damit genügt $>_g$ der Voraussetzung aus Lemma 3.19 und aus der Wohlfundiertheit von $>_g$ folgt, dass $\mathcal{R}_2$ terminiert. □

Bemerkung 3.26. *Ein Beweis über Lemma 3.21 kann für $>_g$ nicht geführt werden, da es sich nicht um eine Reduktionsordnung handelt. Es stellt sich heraus, dass $>_g$*

nicht abgeschlossen unter Substitution ist. Während $s \equiv g(x, g(y, y)) >_g g(x, x) \equiv t$ *ist, gilt dies nicht mehr unter der Substitution* $\sigma = [x := g(y, y)]$ *, da* $s^\sigma \equiv t^\sigma \equiv g(g(y, y), g(y, y))$ *ist.*

Für $\mathcal{R}_1$ gestaltet sich die Suche nach einer geeigneten Reduktionsordnung schwieriger. Es beinhaltet lediglich die Regel $\rho_1 : f(0, 1, x) \to f(x, x, x)$. Durch Zählen der Teilterme, die das Präfix $f(0, 1, \square)$ tragen, kann eine wohlfundierte Striktordnung über $(>, \mathbb{N})$ gebildet werden. Dies führt jedoch nicht zum Ziel, da sich in einem Ersetzungsschritt die Anzahl der in Frage kommenden Teilterme durchaus erhöhen kann, wie das folgende Beispiel illustriert. Sei $t \equiv f(0, 1, (f(0, 1, y))$. Mit $\sigma = [x := f(0, 1, y)]$ ergibt sich nach der Regel $f(0, 1, x) \to f(x, x, x)$ der Redex $f(0, 1, x)^\sigma \equiv f(0, 1, (f(0, 1, y))$ und somit das Kontraktum $f(x, x, x)^\sigma \equiv f(f(0, 1, y), f(0, 1, y), f(0, 1, y))$. Lemma 3.19 kann daher nicht für diese Striktordnung angewendet werden. Für das Termersetzungssystem $\mathcal{R}_1$ kann aber mit folgenden Überlegungen eine andere Striktordnung gefunden werden.

Lemma 3.27. *Die Tiefe aller Terme in jeder Reduktionsfolge in* $\mathcal{R}_1$ *ist konstant.*

Beweis. Für einen beliebigen Reduktionsschritt $s \to_{R_1} t$ gilt

$$s \equiv C[f(0, 1, x)^\sigma] \equiv C[f(x, x, x)^\sigma] \equiv t$$

für einen geeigneten Kontext $C[]$ und eine geeignete Substitution σ. Es genügt zu zeigen, dass $\delta(f(0, 1, x)^\sigma) = \delta(f(x, x, x))$. Da $\delta(0) = \delta(1) = 1$, gilt $\delta(f(0, 1, x)^\sigma) = 1 + \delta(x^\sigma)$. Auf der anderen Seite gilt aber $\delta(f(x, x, x)^\sigma) = 1 + \max(\delta(x^\sigma), \delta(x^\sigma), \delta(x^\sigma)) = 1 + \delta(x^\sigma) = \delta(f(0, 1, x)^\sigma)$. $\square$

Jeder Term einer Reduktionsfolge kann nun als Baum und dieser wiederum ebenenweise betrachtet werden. Für jede Ebene wird gezählt, wie häufig hier ein Teilterm beginnt, der ein Redex zur Regel ρ_1 sein könnte. An der Wurzel kann dies höchstens einmal der Fall sein, in den Blättern hingegen nie. Für einen Term der Tiefe n ergibt sich ein n-Tupel über den natürlichen Zahlen. Für

$$f(0, 1, f(f(0, 1, y), f(0, 1, y), f(0, 1, y)))$$

ergibt sich zum Beispiel $(1, 0, 3, 0)$. Da die Tiefe der Terme in einer Reduktionsfolge konstant ist, haben auch die zugehörigen Tupel der gesamten Reduktionsfolge dieselbe Stelligkeit. Es lässt sich leicht eine Striktordnung auf diesen aufstellen, die mit einem Reduktionsschritt übereinstimmt. Dabei wird ausgenutzt, dass durch einen Ersetzungsschritt in der i-ten Ebene sich die i-te Stelle des Tupels um eins verringert,

während die vorhergehenden Stellen gleich bleiben. Die unter Umständen veränderten Werte der folgenden Stellen werden für die Striktordnung nicht berücksichtigt. Im Folgenden wird diese Striktordnung formal spezifiziert.

Dazu wird eine Hilfsmenge $F^n_{C[]}(t)$ definiert, die genau die Teilterme von t beinhaltet, die mit dem Präfix $C[]$ beginnen und sich durch einen Kontext mit dem Versatz n zu t ergänzen lassen. Die Kardinalitäten der Mengen ergeben die gewünschten Tupel.

Definition 3.28. *Für ein festes $n \in \mathbb{N}$ und einen festen Kontext $C[]$ über eine Signatur Σ wird $F_{C[]} : \mathrm{Ter}(\Sigma) \to \mathcal{P}(\mathrm{Ter}(\Sigma))$ definiert als*

$$F^n_{C[]}(t) = \{t' \sqsubseteq t | C[] \ggg t' \ \wedge \exists D[] : \xi(D[]) = n \wedge D[t'] \equiv t\}.$$

Für einen Term $t \in \mathrm{Ter}(\Sigma)$ der Tiefe $\delta(t) = n$ wird $m : \mathrm{Ter}(\Sigma) \to \mathbb{N}^n$ definiert als

$$m(t) = (|F^0_{f(0,1,\square)}(t)|, \ldots, |F^{n-1}_{f(0,1,\square)}(t)|).$$

Bemerkung 3.29. *Der Kontext $D[]$ in der Konstruktion von $F^n_{C[]}(t)$ ist eindeutig bestimmt für jeden Teilterm t' durch $D[t'] \equiv t$.*

Definition 3.30. *Die Striktordnung $>_n$ wird auf $\mathbb{N}^n$ definiert als*

$$(k_1, \ldots, k_n) >_n (l_1, \ldots, l_n) \iff \exists i \leq n : k_i > l_i \wedge \forall j < i : k_j = l_j.$$

Die Striktordnung $>_n$ ist total und hat 0^n als kleinstes Element. Es folgt, dass $>_n$ wohlfundiert ist.

Nun lässt sich die eingangs beschriebene Striktordnung definieren. Hierbei muss berücksichtigt werden, dass nur Terme gleicher Tiefe vergleichbar sind. Dies stellt keine Einschränkung dar, da die Tiefe aller Terme einer Reduktionsfolge in dem Termersetzungssystem $\mathcal{R}_1$ aus Toyamas Gegenbeispiel konstant ist.

Definition 3.31. *Die Striktordnung $>_f$ auf $\mathrm{Ter}\,\Sigma_1$ wird definiert durch*

$$s >_f t \iff m(s) >_n m(t) \wedge \delta(s) = \delta(t) = n$$

Lemma 3.32. *Das Termersetzungssystem $\mathcal{R}_1$ terminiert.*

Beweis. Der Beweis erfolgt über Lemma 3.19 mit der Ordnung $>_f$. Aus der Wohlfundiertheit von $>_n$ folgt die Wohlfundiertheit von $>_f$. Es bleibt zu zeigen, dass $s >_f t$ aus $s \to_{R_1} t$ folgt. Wegen $s \to_{R_1} t$ gilt $s \equiv C[f(0,1,x)^\sigma] \equiv C[f(x,x,x)^\sigma] \equiv$

t. Nach Lemma 3.27 ist $\delta(s) = \delta(t) = n$. Sei nun $(s_0, \ldots, s_{n-1}) = m(s)$ und $(t_0, \ldots, t_{n-1}) = m(t)$ und $i = \xi(C[])$. Es gilt $s_j = t_j$ für alle $j < i$. Dies liegt daran, dass an diesen Stellen nur die Anzahl derjenigen Teilterme eingeht, die sich in einer geringeren Tiefe als der zu ersetzende Redex befinden. Da sich aber nach Lemma 3.27 die Tiefe nicht ändert und das Präfix durch die Ersetzung nicht verändert wird, bleibt die Anzahl an den Stellen kleiner i konstant. Ferner muss $s_i > t_i$ gelten, denn der Redex $f(0,1,x)^\sigma$ ist Element der Menge $F^i_{f(0,1,\Box)}(t)$, das Kontraktum $f(x,x,x)^\sigma$ aber nicht. Da $s_i = |F^i_{f(0,1,\Box)}(s)|$ und $t_i = |F^i_{f(0,1,\Box)}(t)|$, gilt $s_i = t_i + 1$. Nach Definition 3.30 folgt $m(s) >_n m(t)$ und somit auch $s >_f t$. □

Lemma 3.33. *Die disjunkte Vereinigung der beiden Regeln erzeugt ein nicht terminierendes Termersetzungssystem.*

Beweis. Der Term $f(0,1,g(0,1))$ hat unter $(\Sigma, \{\rho_1\} \cup \{\rho_2, \rho_3\})$ eine unendliche Reduktion der folgenden Form.

$$\begin{aligned} f(0,1,g(0,1)) &\to_{\rho_1} f(g(0,1), g(0,1), g(0,1)) \\ &\to_{\rho_2} f(0, g(0,1), g(0,1)) \\ &\to_{\rho_3} f(0,1,g(0,1)) \\ &\to_{\rho_1} f(g(0,1), g(0,1), g(0,1)) \\ &\ldots \end{aligned}$$

□

Toyamas Gegenbeispiel demonstriert, dass Termersetzungssysteme nicht analysiert werden können, indem disjunkte Teiltermersetzungssysteme betrachtet werden. Im folgenden Kapitel wird eine von Arts und Giesl entwickelte Methode beschrieben, die eine Zerlegung in Teilprobleme möglich macht. Dabei sind die Teilprobleme jedoch nicht mehr einfache Termersetzungssysteme.

4 Das Dependency Pair Framework

Das Dependency Pair Framework (siehe [GTSK05]) stellt einen Ansatz dar, mit dem beliebige Methoden zur Terminierungsanalyse von Termersetzungssystemen zusammengeführt werden. Diese Modularität, wohlgemerkt bezüglich der Methoden, macht den Ansatz besonders erfolgreich. Das Framework beruht auf dem sogenannten Dependency Pair Approach, welcher von Thomas Arts in [Art97] beschrieben und in [AG00] verfeinert wurde.

4.1 Grundbegriffe

Das Dependency Pair Framework führt zwei zentrale Begriffe ein, Abhängigkeitspaare und Ketten. Diese sollen in diesem Abschnitt definiert und motiviert werden.

Definition 4.1. *Die Wurzelsymbole der linken Seiten der Regeln eines Termersetzungssystem $\mathcal{R}$ werden als* definierte Symbole *bezeichnet. Die Menge der definierten Symbole des Termersetzungssystems $\mathcal{R}$ werden mit $D_{\mathcal{R}}$ bezeichnet. Alle anderen Funktionssymbole der Signatur des Termersetzungssystems heißen* Konstruktoren*. Die Menge der Konstruktoren wird mit $C_{\mathcal{R}}$ bezeichnet und es gilt $C_{\mathcal{R}} = \Sigma \setminus D_{\mathcal{R}}$. Ist das Termersetzungssystem aus dem Kontext ersichtlich, kann der Index entfallen. Für das Termersetzungssystem (Σ, R) mit $\Sigma = D \cup C$ wird auch (D, C, R) geschrieben.*

Beispiel 4.2. *Sei $\Sigma = \{f, g, h, c\}$ mit den Stelligkeit $\mathrm{ari}(f) = \mathrm{ari}(g) = \mathrm{ari}(h) = 1$ und $\mathrm{ari}(c) = 0$ die Signatur des Termersetzungssystem $\mathcal{R} = (\Sigma, R)$, wobei R die folgenden zwei Regeln enthält:*

$$f(f(x)) \to g(x)$$

$$g(h(c)) \to h(c)$$

Die definierten Symbole sind $D_{\mathcal{R}} = \{f, g\}$, die Konstruktoren und $C_{\mathcal{R}} = \{h, c\}$.

Der dem Dependency-Pair-Framework zugrunde liegende Ansatz betrachtet nun nur diejenigen Regeln, deren rechte Seiten wieder eine neue Ersetzungsfolge in Gang setzen können. Das ist nur dann der Fall, wenn Teilterme der rechten Seite mit definierten Symbolen beginnen, da die definierten Symbole gerade diejenigen sind, die auf der linken Seite einer Regel stehen, also einen neuen Ersetzungsschritt ermöglichen könnten. Ob tatsächlich eine weitere Ersetzung möglich ist, hängt von den Argumenten ab. Um diese zu betrachten, werden neue Symbole in die Signatur eingeführt, die das Argumenttupel festhalten. Dabei wird die Notation aus [AG00] genutzt und angenommen, dass sich die Signatur nur aus kleinbuchstabigen Funktionssymbolen zusammensetzt. Die neu eingeführten Symbole werden mit den zugehörigen Großbuchstaben gekennzeichnet (also F für f etc.). Die neuen Symbole werden als Tupelsymbole bezeichnet, da sie die Argumenttupel der definierten Symbole verfolgen.

Definition 4.3. *Für ein Termersetzungssystem $\mathcal{R} = (D_\mathcal{R}, C_\mathcal{R}, R)$ mit $g \in D$ ist*

$$\langle F(s_1, \ldots, s_n), G(t_1, \ldots, t_n)\rangle$$

ein Abhängigkeitspaar *genau dann, wenn es eine Regel $\rho \in R$ der Form*

$$\rho : f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_n)]$$

gibt, wobei C ein beliebiger Kontext ist. Die Menge aller Abhängigkeitspaare eines Termersetzungssystems $\mathcal{R}$ wird als $\mathrm{DP}(\mathcal{R})$ *notiert.*

Gemäß Definition von D folgt, dass nicht nur $g \in D$ sondern auch $f \in D$ gilt, da f das Wurzelsymbol der linken Seite der Regel $\rho \in R$ ist.

Definition 4.4. *Eine möglicherweise unendliche Folge von Abhängigkeitspaaren aus dem Termersetzungssystem $\mathcal{R}$*

$$\langle s_i, t_i\rangle, \langle s_{i+1}, t_{i+1}\rangle, \ldots$$

*ist genau dann eine $\mathcal{R}$-*Kette*, wenn es Substitutionen σ_i gibt, sodass $t_i^{\sigma_i} \twoheadrightarrow_R s_{i+1}^{\sigma_{i+1}}$ für alle unmittelbar aufeinander folgenden Paare $\langle s_i, t_i\rangle$ und $\langle s_{i+1}, t_{i+1}\rangle$ gilt.*

Wenn aus dem Kontext das Termersetzungssystem $\mathcal{R}$ bekannt ist, wird eine $\mathcal{R}$-Kette auch kurz eine Kette genannt.

Die obige Definition weicht leicht von der in [AG00] ab. Dort wird eine einzige Substitution σ für die gesamte Kette verlangt. Allerdings nehmen Arts und Giesl

implizit an, dass die Variablen zweier Vorkommen von Abhängigkeitspaaren in einer Kette disjunkt sind. Die Annahme entspricht der Vorstellung, dass die Variablen in den Termen der Ersetzungsregeln einen Gültigkeitsbereich haben, der nicht über den gerade vorgenommen Ersetzungsschritt hinausgeht. Nimmt man diese zulässige, aber wenig intuitive Annahme hinzu, lassen sich die in der vorliegenden Arbeit verwendeten, vielen Substitutionen einer Kette zu einer einzigen zusammenfassen, nämlich als $\sigma = \sigma_1 \circ \sigma_2 \circ \ldots$.

Bemerkung 4.5. *Alle Ketten haben die Form*

$$\langle F_1(\vec{u}_1), F_2(\vec{v}_2)\rangle, \langle F_2(\vec{u}_2), F_3(\vec{v}_3)\rangle, \langle F_3(\vec{u}_3), F_4(\vec{v}_4)\rangle, \ldots$$

denn es gilt $\mathrm{root}(t_i) = \mathrm{root}(s_{i+1})$. *Der Grund hierfür ist, dass Abhängigkeitspaare aus Tupelsymbolen mit Argumenten bestehen, welche von $\mathcal{R}$ nicht transformiert werden, da sie neu eingeführt wurden. Die für die Ketteneigenschaft geforderte Reduktionsfolge von $t_i^{\sigma_i} \twoheadrightarrow_R s_{i+1}^{\sigma_{i+1}}$ kann daher nur die Argumente von t_i verändern, muss allerdings* $\mathrm{root}(t_i)$ *belassen und somit gilt* $\mathrm{root}(t_i) = \mathrm{root}(s_{i+1})$.

4.2 Terminierungskriterien

Auf Basis des Kettenbegriffs lässt sich ein neues Terminierungskriterium (Theorem 4.8) definieren, dass die Grundlage für weitere Kriterien darstellt. Die folgende Definition erweist sich als nützlich für den Beweis, dass das Theorem 4.8 korrekt ist.

Definition 4.6. *Ein Term $s \equiv f(t_1, \ldots, t_n)$ heißt* minimaler, nicht terminierender Term *bezüglich eines Termersetzungssystems $\mathcal{R}$ genau dann, wenn s nicht terminiert, aber alle Teilterme terminieren.*

Korollar 4.7. *Sei $f(\vec{t})$ ein minimaler, nicht terminierender Term bezüglich eines Termersetzungssystems $\mathcal{R}$. Werden Ersetzungsschritte gemäß $\mathcal{R}$ nur in den Teiltermen $\vec{t}$ vorgenommen, sodass nach n Schritten einer solchen Reduktionsfolge die Teilterme $\vec{t_n}$ entstehen, so sind auch alle Teilterme $\vec{t_n}$ stark normalisierend. Ansonsten wäre $f(\vec{t})$ nicht minimal, da ein Teilterm in $\vec{t}$ existiert, der ebenfalls nicht terminierend ist.*

Arts und Giesl formulieren und Beweisen in [AG00] das zentrale Terminierungskriterium wie folgt.

Theorem 4.8. *Ein Termersetzungssystem $\mathcal{R}$ terminiert genau dann, wenn es keine unendlichen $\mathcal{R}$-Ketten gibt.*

Beweis. Das Kriterium ist hinreichend, denn es lässt sich für jede unendliche Reduktionsfolge eine entsprechende Kette aufstellen. Der erste Term der unendlichen Reduktionsfolge enthält einen minimalen, nicht terminierenden Term $f_1(\vec{u}_1)$. Im Rahmen der mit $f_1(\vec{u}_1)$ beginnenden, unendlichen Reduktionsfolge muss also eine Regel angewendet werden, die $f_1(\vec{u}_1)$ transformiert. Diese Regel sei $f_1(\vec{w}_1) \to r_1$. Damit diese angewendet werden kann, wird $\vec{u}_1$ in endlich vielen Reduktionsschritten zu $\vec{v}_1$ transformiert, sodass es eine Substitution σ_1 gibt, bei der $\vec{w}_1^{\sigma_1} \equiv \vec{v}_1$ gilt. Die unendliche Reduktionsfolge geht mit $r_1^{\sigma_1}$ weiter. Da alle Terme aus $\vec{v}_1$ gemäß Korollar 4.7 stark normalisierend sind, gibt es keine Variable x, sodass $\sigma_1(x)$ nicht stark normalisierend wäre. Für den notwendigerweise in $r_1^{\sigma_1}$ existierenden Teilterm $f_2(\vec{u}_2)^{\sigma_1}$, der analog zu $f_1(\vec{u}_1)$ ein minimaler, nicht terminierender Term ist, folgt, dass $f_2(\vec{u}_2) \sqsubseteq r_1$ ist. Ansonsten wären die Terme $\vec{v}_1$ nicht stark normalisierend. Damit ist $\langle F_1(\vec{w}_1), F_2(\vec{u}_2)\rangle$ ein Abhängigkeitspaar. Die weiteren Abhängigkeitspaare ergeben sich auf dieselbe Weise. Die Paare bilden nach Konstruktion eine Kette mit den jeweiligen Substitutionen σ_i.

Das Kriterium ist ferner notwendig. Für eine Kette

$$\langle F_1(\vec{u}_1), F_2(\vec{v}_2)\rangle, \langle F_2(\vec{u}_2), F_3(\vec{v}_3)\rangle, \langle F_3(\vec{u}_3), F_4(\vec{v}_4)\rangle, \ldots$$

gibt es nach Definition Substitutionen σ_i, sodass

$$F_i(\vec{v}_i)^{\sigma_{i-1}} \twoheadrightarrow F_i(\vec{u}_i)^{\sigma_i}$$

gilt und folglich auch

$$f_i(\vec{v}_i)^{\sigma_{i-1}} \twoheadrightarrow f_i(\vec{u}_i)^{\sigma_i}$$

gilt. Aus dieser Kette kann eine unendliche Reduktionsfolge auf folgende Art und Weise konstruiert werden.

Sei $l_1 \to r_1$ die Regel, aus der das Abhängigkeitspaar $\langle F_1(\vec{u}_1), F_2(\vec{v}_2)\rangle$ hervorging. Es gilt $l_1 \equiv f_1(\vec{u}_1)$ und $r_1 \equiv C_1[f_2(\vec{v}_2)]$. Der erste Term der unendlichen Reduktionsfolge ist $f_1(\vec{u}_1)^{\sigma_1}$. Gemäß der Regel $l_1 \to r_1$ gilt $f_1(\vec{u}_1)^{\sigma_1} \to C_1[f_2(\vec{v}_2)^{\sigma_1}]$. Aus der Ketten-Eigenschaft folgt nach den Vorüberlegungen, dass

$$f_2(\vec{v}_2)^{\sigma_1} \twoheadrightarrow f_2(\vec{u}_2)^{\sigma_2}$$

und somit

$$C_1[f_2(\vec{v}_2)^{\sigma_1}] \twoheadrightarrow C_1[f_2(\vec{u}_2)^{\sigma_2}].$$

Zum zweiten Abhängigkeitspaar $\langle F_2(\vec{u}_2), F_3(\vec{v}_3)\rangle$ gehört es ein Regel $l_2 \to r_2$ mit

$l_2 \equiv f_2(\vec{u}_2)$. Offensichtlich ist diese Regel anwendbar auf $C_1[f_2(\vec{u}_2)^{\sigma_2}]$. Mit $r_2 \equiv C_2[f_3(\vec{v}_3)]$ entsteht der Term $C_1[C_2[f_3(\vec{v}_3)^{\sigma_2}]]$. Der innere Term $f_3(\vec{v}_3)^{\sigma_2}$ kann nach der Vorüberlegung zu $f_3(\vec{u}_3)^{\sigma_3}$ reduziert werden. Insgesamt entsteht die unendliche Konstruktion:

$$\begin{aligned} f_1(\vec{u}_1)^{\sigma_1} &\to C_1[f_2(\vec{v}_2)^{\sigma_1}] \\ &\twoheadrightarrow C_1[f_2(\vec{u}_2)^{\sigma_2}] \\ &\to C_1[C_2[f_2(\vec{v}_3)^{\sigma_2}]] \\ &\twoheadrightarrow C_1[C_2[f_2(\vec{u}_3)^{\sigma_3}]] \\ &\to C_1[C_2[C_3[f_3(\vec{v}_4)]^{\sigma_4}]] \\ &\to \ldots \end{aligned}$$

□

Der Vorteil des Terminierungskriteriums aus Theorem 4.8 ist nicht unmittelbar einleuchtend. Statt zu beweisen, dass keine unendlichen Reduktionsfolgen existieren, muss nun gezeigt werden, dass keine unendlichen Ketten existieren. Nach wie vor muss also nachgewiesen werden, dass bestimmte unendliche Strukturen nicht existieren. Der Nachweis kann also wieder erfolgen, indem eine geeignete, wohlfundierte Relation auf den Paaren gefunden wird. Der Vorteil liegt allerdings darin, dass Ketten im Gegensatz zu beliebigen Reduktionsfolgen strukturierter sind, wodurch sich mehr Freiheit bei der Wahl der Relation ergibt. So reicht es, eine wohlfundierte Quasiordnung $\geq$ zu finden, für die

$$s_1^{\sigma_0} > t_1^{\sigma_1} \geq s_2^{\sigma_1} > t_2^{\sigma_2} \geq \ldots$$

gilt, um die Terminierung zu beweisen. Diese Methode ist ebenfalls nicht praktisch durchführbar, demonstriert aber den prinzipiellen Vorteil, den die Struktur der Ketten bietet. Die Abhängigkeitspaare stellen ferner eine Vereinfachung dar, da nicht mehr der gesamte Term betrachtet wird, sondern nur noch Teilterme.

Eine tatsächlich durchführbare Methode, die die Vorteile der Kettenstruktur ausnutzt, ist die Analyse des Abhängigkeitsgraphen.

Definition 4.9. *Der* Abhängigkeitsgraph *des TRS $\mathcal{R}$ ist der gerichtete Graph (V, E) mit den Abhängigkeitspaaren als Knotenmenge V und*

$$E = \{(\langle s,t\rangle, \langle u,v\rangle) | \exists\sigma : t^\sigma \twoheadrightarrow_R u^\sigma\}$$

als Kantenmenge. Zwischen den Knoten besteht also genau dann eine Kante, wenn die beiden Abhängigkeitspaare eine zweielementige Kette bilden.

Jede Kette korrespondiert mit einem Pfad in dem Graphen und umgekehrt. Irritierenderweise findet sich in [AG00] die Behauptung, dass die Umkehrung nicht zuträfe. Dies gilt nur vor dem Hintergrund der Definition von Ketten über eine einzelne Substitution und auch dann nur, wenn man die zusätzliche Annahme über disjunkte Variablen fallen lässt. Arts und Giesl erläutern an dieser Stelle nicht, warum sie die Annahme in ihrer Definition des Abhängigkeitsgraphen fallen lassen. Die Frage nach der Umkehrbarkeit spielt allerdings keine Rolle für die weiteren Resultate.

Der Abhängigkeitsgraph lässt sich nun ausnutzen, um die Terminierung von Termersetzungssystemen zu beweisen. Im Zentrum steht die Beobachtung, dass in einem endlichen Graphen nur dann ein unendlicher Pfad existiert, wenn der Graph Zyklen enthält. Unendliche Ketten führen also unendlich oft über einen oder mehrere der Zyklen des Abhängigkeitsgraphen. Wieder wird eine wohlfundierte Quasiordnung $\geq$ bemüht. Falls es gelingt zu zeigen, dass auf jedem der Kreise ein Schritt liegt, der dem strikten Teil $>$ der Ordnung $\geq$ genügt, so folgt, dass es in einer unendlichen Kette unendliche viele dieser strikten Schritte geben müsste, was im Widerspruch zu der Wohlfundiertheit der Quasiordnung steht. Aus diesem Argument formulieren und beweisen Arts und Giesl Theorem 4.10.

Theorem 4.10. *Ein Termersetzungssystem $\mathcal{R}$ terminiert genau dann, wenn es eine wohlfundierte, schwach monotone Quasiordnung $\geq$ gibt, deren strikter Teil $>$ und sie selber unter Substitution abgeschlossen ist und die folgenden Eigenschaften erfüllt sind:*

- $l \geq r$ *für alle Regeln* $l \to r$ *aus* R *gilt*
- $s \geq t$ *für alle Abhängigkeitspaare* $\langle s, t \rangle$ *auf Kreisen im Abhängigkeitsgraphen*
- $s > t$ *für mindestens ein Abhängigkeitspaar* $\langle s, t \rangle$ *in jedem Kreis im Abhängigkeitsgraphen*

Beweis. Der Beweis, dass die Existenz einer wie in Theorem 4.10 beschriebenen Quasiordnung hinreichend für die Terminierung von $\mathcal{R}$ ist, folgt aus den obigen Überlegungen. Die wohlfundierte Quasiordnung garantiert, dass es keine unendlich absteigenden $\mathcal{R}$-Ketten gibt, da jede dieser Ketten unendlich viele, strikt absteigende Paare enthalten müsste. Dies widerspräche allerdings der Wohlfundiertheit von $>$.

Es folgt, dass keine unendlichen $\mathcal{R}$-Ketten existieren. Nach Theorem 4.8 ergibt sich, dass $\mathcal{R}$ terminiert.

Es bleibt zu zeigen, dass jedes terminierende Termersetzungssystem notwendigerweise eine derartige Ordnung besitzt. Diese Ordnung ergibt sich aus der Reduktionsordnung $\twoheadrightarrow_{\mathcal{R}'}$ des Termersetzungssystems $\mathcal{R}'$, welches aus den Regeln von $\mathcal{R}$ und zusätzlichen Regeln $s \to t$ bestehen, die aus den Abhängigkeitspaaren $\langle s, t\rangle$ von $\mathcal{R}$ gebildet werden. Arts und Giesl zeigen, dass $\mathcal{R}'$ terminiert, wenn $\mathcal{R}$ terminiert.

Dazu wird angenommen, dass es einen Term q_1 gibt, der eine unendliche Reduktionsfolge

$$q_1 \to_{\mathcal{R}'} q_2 \to_{\mathcal{R}'} \dots$$

beginnt, aber keine unendliche Reduktionsfolge in $\mathcal{R}$ beginnt. Daher muss q_1 Tupelsymbole beinhalten. Ferner sei q_1 minimal in dem Sinne, dass alle echten Teilterme von q_1 keine unendliche Reduktionsfolge bezüglich $\mathcal{R}'$ beginnen.

Aus der Minimalität folgt, dass das Wurzelsymbol des Terms q_1 ein Tupelsymbol sein muss, wie folgende Überlegung zeigt. Sei q_1' der Term, der sich aus q_1 ergibt, indem alle Tupelsymbole durch eine Variable x ersetzt werden. In der unendlichen Reduktionsfolge können nur endlich viele Reduktionsschritte stattfinden, bei denen die Redexe nicht mit einem Tupelsymbol beginnen oder unterhalb eines Tupelsymbols liegen. Ansonsten würde auch q_1' eine unendliche Reduktionsfolge starten. Da aber q_1' nach Konstruktion keine Tupelsymbole enthält, können nur Regeln aus $\mathcal{R}$ angewendet werden. Da $\mathcal{R}$ nach Annahme terminiert, ergibt sich die Aussage, dass nach endlich vielen Schritten, alle Redexe unterhalb eines Tupelsymbols liegen oder ein Tupelsymbol als Wurzelsymbol haben.

Sei q_k der letzte Term, der aus einem Reduktionsschritt oberhalb von allen Tupelsymbolen hervorgeht. Mit q_k beginnt ebenfalls eine unendliche Reduktionsfolge bezüglich $\mathcal{R}'$. Da alle Reduktionsschritte ab q_k unter einem Tupelsymbol stattfinden, gibt es einen Teilterm s_k von q_k, der mit einem Tupelsymbol beginnt und selbst eine unendliche Reduktionsfolge beginnt. Da $\mathcal{R}$ keine Tupelsymbol einführt und die anderen Regeln in $\mathcal{R}'$ nur Tupelsymbole umwandeln, muss es einen Teilterm s_1 von q_1 geben, sodass das Wurzelsymbol von s_1 ein Tupelsymbol ist und $s_1 \twoheadrightarrow'_{\mathcal{R}} s_k$. Damit startet s_1 ebenfalls eine unendliche Reduktionsfolge und aus der geforderten Minimalität ergibt sich, dass $q_1 \equiv s_1$.

Das Wurzelsymbol von q_1 ist daher ein Tupelsymbol. Die Reduktionsfolge muss also die Gestalt

$$F_1(\vec{u}_1) \twoheadrightarrow_{\mathcal{R}} F_1(\vec{v}_1) \to_{\mathcal{R}'} F_2(\vec{u}_2) \twoheadrightarrow_{\mathcal{R}} F_2(\vec{v}_2) \to_{\mathcal{R}'} F_3(\vec{u}_3) \twoheadrightarrow_{\mathcal{R}} \dots$$

haben. Damit sind $F_i(\vec{v}_i)$ und $F_{i+1}(\vec{u}_{i+1})$ Instanzen eines Abhängigkeitspaares aus $\mathcal{R}$ für eine bestimmte Substitution σ_i und es ergibt sich eine unendliche Folge von Abhängigkeitspaaren nämlich

$$\langle F_1(\vec{s}_1), F_2(\vec{t}_2)\rangle, \langle F_2(\vec{s}_2), F_3(\vec{t}_3)\rangle, \langle F_3(\vec{s}_3), F_4(\vec{t}_4)\rangle, \ldots$$

welche mit den Substitutionen σ_i eine unendliche $\mathcal{R}$-Kette bildet. Dies steht im Widerspruch zur Terminierung von $\mathcal{R}$.

Es folgt, dass die Ersetzungsrelation $\twoheadrightarrow_{\mathcal{R}'}$ eine wohlfundierte Ordnung ist und damit auch eine wohlfundierte Quasiordnung. Nach Konstruktion als Ersetzungsrelation ist $\twoheadrightarrow_{\mathcal{R}'}$ ferner schwach monoton und abgeschlossen unter Substitution bezüglich $\mathcal{R}'$. Da $\mathcal{R}'$ die Regeln aus $\mathcal{R}$ umfasst, gelten die Eigenschaften auch für $\mathcal{R}$. □

4.3 Berechnung des Abhängigkeitsgraphen

Der Abhängigkeitsgraph ist im Allgemeinen nicht berechenbar, da für zwei Abhängigkeitspaare nicht entscheidbar ist, ob sie eine Kette bilden. Arts und Giesl bilden daher eine Approximation des Abhängigkeitsgraphen. Dabei handelt es sich um einen Graphen, der den Abhängigkeitsgraphen als Teilgraphen enthält. Dadurch lässt sich Theorem 4.10 auf den approximierten Graphen anwenden. Findet sich eine geeignete Ordnung, die den Anforderungen aus Theorem 4.10 für den approximierten Graphen genügt, so sind die Anforderungen auch für den tatsächlichen Abhängigkeitsgraphen erfüllt. Gemäß Theorem 4.10 folgt die Terminierung des Termersetzungssystems.

Die Approximation des Graphen wird über verbindbare Terme bestimmt. Der Begriff wird so definiert, dass, wenn zwei Terme t und u verbindbar sind, die Abhängigkeitspaare $\langle s, t\rangle$ und $\langle u, v\rangle$ eine Kette bilden.

Die Approximation des Abhängigkeitsgraphen wird nun berechnet, indem die verbindbaren Terme zwischen Abhängigkeitspaaren bestimmt werden und für diese Kanten erzeugt werden. Auf diese Weise enthält der approximierte Graph mindestens alle Kanten des Abhängigkeitsgraphen.

Um zu beschreiben, unter welchen Bedingung zwei Terme verbindbar sind, werden zwei Funktionen definiert. Die erste Funktion heißt REN. Sie vergibt unterschiedliche Namen für die unterschiedlichen Vorkommen möglicherweise gleicher Variablen. Um die neuen Variablen unterschiedlich zu benennen, wird ein Index aus dem freien Monoid über $\mathbb{N}$ verwendet (siehe Definition A.6).

Definition 4.11. *Für ein Termersetzungssystem* (D, C, R) *ist die Funktion* $\mathrm{REN} : \mathrm{Ter}(\Sigma, V) \times \mathbb{N}^* \to \mathrm{Ter}(\Sigma, V)$ *definiert als*

$$\begin{aligned} \mathrm{REN}(x, p) &= y_p && \textit{falls } x \in V \\ \mathrm{REN}(f(t_1, \ldots, t_n), p) &= f(\mathrm{REN}(t_1, p \cdot 1), \ldots, \mathrm{REN}(t_n, p \cdot n)) \end{aligned}$$

wobei für alle $p \in \mathbb{N}^*$ *gilt, dass* y_p *eine Variable ist, die nicht im Ausgangsterm verwendet wird.*

Die Motivation für diese Funktion lässt sich leicht aus Toyamas Gegenbeispiel herleiten. In dem Termersetzungssystem aus 3.23 findet sich lediglich ein einziges Abhängigkeitspaar nämlich $\langle F(0, 1, x), F(x, x, x)\rangle$. Da das Beispiel nicht terminiert, muss es auch eine unendliche Kette geben, die an jeder Stelle das einzige Abhängigkeitspaar enthält. Folglich gibt es im Abhängigkeitsgraphen eine Kante von $\langle F(0, 1, x), F(x, x, x)\rangle$ zu sich selbst, welche auch in der Approximation enthalten sein muss. Da diese Approximation jeweils für verbindbare Terme Kanten enthält, müssen $F(x, x, x)$ und $F(0, 1, x)$ verbindbar sein. Da es aber keine Substitution σ geben kann, sodass $F(x, x, x)^\sigma \equiv F(0, 1, x)$ gilt, ist es nötig, die drei Vorkommen der Variablen x durch unterschiedliche Variablen zu ersetzen. Die Funktion REN leistet genau dies, denn mit dem leeren Wort $\epsilon \in \mathbb{N}^*$ gilt $\mathrm{REN}(F(x, x, x), \epsilon) \equiv F(y_1, y_2, y_3)$.

Arts und Giesl definieren REN (und auch die folgende Funktion CAP) als einstellige Funktionen mit dem Hinweis, dass es sich nicht um echte Funktionen handelt, da das Resultat von der Auswertungsreihenfolge abhängt.

Sie schlagen vor, eine Liste von frischen Variablen $y_1, y_2, \ldots$ zu nutzen und einen zweiten formalen Parameter einzuführen, in dem eine frische Variable aus der Liste übergeben wird. Damit ist das Problem allerdings nicht gelöst. Für eine formale Definition müssten in der letzten Definitionsgleichung die Argumente für die rekursiven Aufrufe geschickt gewählt werden, um Kollisionen zu verhindern. Dazu müsste entweder die Struktur der Teilterme bekannt sein oder eine Auswertungsreihenfolge festgelegt werden. Beides gestaltet sich offensichtlich schwieriger als der in der vorliegenden Arbeit vorgestellte Ansatz, mit einem Index aus dem Kleeneschen Abschluss über die natürlichen Zahlen zu arbeiten.

Die zweite Funktion heißt CAP. Ihre Aufgabe besteht darin, Terme zu vereinfachen, indem Teilterme, die mit einem definierten Symbol beginnen, durch neue, unterschiedliche Variablen ersetzt werden. Die Motivation besteht darin, dass definierte Symbole in weiteren Ersetzungsschritten ausgetauscht werden können, wäh-

rend Konstruktorsymbole per Definition höchstens als Teilterm ausgetauscht werden können, ansonsten aber fest bleiben. Erneut wird $\mathbb{N}^*$ als Indexmenge gewählt, um garantiert neue und unterschiedliche Variablen einzuführen.

Definition 4.12. *Sei (D, C, R) ein Termersetzungssystem. Dann ist die Funktion* $\text{CAP} : \text{Ter}(\Sigma, V) \times \mathbb{N}^* \to \text{Ter}(\Sigma, V)$ *definiert als*

$$\begin{aligned} \text{CAP}(x, p) &= y_p && \textit{falls } x \in V \\ \text{CAP}(f(t_1, \ldots, t_n), p) &= y_p && \textit{falls } f \in D \\ \text{CAP}(f(t_1, \ldots, t_n), p) &= f(\text{CAP}(t_1, p \cdot 1), \ldots, \text{CAP}(t_n, p \cdot n)) && \textit{falls } f \notin D \end{aligned}$$

wobei y_p nicht im ursprünglichen Term verwendete Variablen sind.

Im Folgenden steht $\text{REN}(t)$ für $\text{REN}(t, \epsilon)$ und $\text{CAP}(t)$ für $\text{CAP}(t, \epsilon)$.

Definition 4.13. *Der Term s heißt* verbindbar *mit einem Term t genau dann, wenn* $\text{REN}(\text{CAP}(s))$ *unifizierbar mit t ist, also eine Substitution σ existiert, sodass* $\text{REN}(\text{CAP}(s))^\sigma \equiv t^\sigma$ *gilt.*

Lemma 4.14. *Gegeben sei ein Termersetzungssystem $\mathcal{R}$. Der Graph, der aus den Abhängigkeitspaaren von $\mathcal{R}$ als Knoten und mit Kanten zwischen $\langle s, t\rangle$ und $\langle u, v\rangle$ besteht, falls t verbindbar mit u ist, enthält den Abhängigkeitsgraphen von $\mathcal{R}$ als Teilgraphen.*

Beweis. Es ist zu zeigen, dass für jede Kette $\langle s, t\rangle, \langle u, v\rangle$ der Term t mit u verbindbar ist. Arts und Giesl zeigen mittels Induktion über die Struktur der Terme, dass $\text{REN}(\text{CAP}(t))$ einen Term w subsumiert, falls $t^\sigma \twoheadrightarrow w$ gilt. Besteht der Term t nur aus einer Variablen oder beginnt mit einem definierten Symbol, ist $\text{REN}(\text{CAP}(t))$ eine frische Variable. Trivialerweise subsumiert eine Variable alle Terme wegen der Substitution, die die Variable durch den Term ersetzt. Dies ist der Induktionsanfang. Ist das Wurzelsymbol des Terms t ein Konstruktorsymbol c, also $t \equiv c(t_1, \ldots, t_n)$, so folgt, dass der Term w ebenfalls mit c beginnt, da Konstruktorsymbole, die Symbole sind, die per Definition nicht an der Wurzel der linken Seite einer Regel stehen dürfen. Folglich ist zu zeigen, dass $\text{REN}(\text{CAP}(c(t_1, \ldots, t_n)))$ den Term $w \equiv c(w_1, \ldots, w_n)$ subsumiert. Es gilt

$$\text{REN}(\text{CAP}(c(t_1, \ldots, t_n))) = c(\text{REN}(\text{CAP}(t_1), 1), \ldots, \text{REN}(\text{CAP}(t_n), n))$$

und nach Induktionsannahme subsumieren die Terme $\text{REN}(\text{CAP}(t_i), i)$ jeweils die Terme w_i. Da die Variablenmengen von $\text{REN}(\text{CAP}(t_i), i)$ nach Konstruktion der

Funktion REN disjunkt sind, folgt, dass $\mathrm{REN}(\mathrm{CAP}(c(t_1,\ldots,t_n)))$ den Term w subsumiert.

Bekannt ist ferner, dass $t^\sigma \twoheadrightarrow u^\sigma$ gilt, da es sich bei $\langle s,t\rangle, \langle u,v\rangle$ um eine Kette handelt. Daher subsumiert $\mathrm{REN}(\mathrm{CAP}(t))$ auch u^σ, es existiert also eine Substitution τ, sodass $\mathrm{REN}(\mathrm{CAP}(t))^\tau \equiv u^\sigma$ ist. Es reicht, wenn sich die Domäne der τ-Substitution auf die frischen Variablen aus $\mathrm{REN}(\mathrm{CAP}(t))$ beschränkt, welche disjunkt von den Variablen in u sind. Analoges gilt für σ, dass sich auf Variablen in u und t beschränkt. Es folgt, dass für die Substitution $\tau \circ \sigma$ die Gleichheiten $\mathrm{REN}(\mathrm{CAP}(t))^\tau \equiv \mathrm{REN}(\mathrm{CAP}(t))^{\tau\circ\sigma}$ und $u^\sigma \equiv u^{\tau\circ\sigma}$ erfüllt sind. Damit ist aber auch

$$\mathrm{REN}(\mathrm{CAP}(t))^{\tau\circ\sigma} \equiv \mathrm{REN}(\mathrm{CAP}(t))^\tau \equiv u^\sigma \equiv u^{\tau\circ\sigma}$$

und schließlich t mit u verbindbar. □

Im Folgenden werden die Begriffe Abhängigkeitsgraph und Approximation eines Abhängigkeitsgraphen nicht mehr scharf voneinander getrennt.

4.4 Vom Ansatz zum Framework

Die bisher vorgestellten Elemente entstammen, soweit nicht anders angemerkt, der Arbeit von Arts und Giesl und bilden den Kern ihrer Arbeit [AG00]. Es handelt sich allerdings nur um einen Ausschnitt. Arts und Giesl beschreiben in [AG00] weitere Verbesserungen gegenüber dem in der vorliegenden Arbeit vorgestellten Ausschnitt sowie eine Variante des Ansatzes, die sich mit Terminierung unter Strategien beschäftigt, nämlich der Strategie der innersten Reduktion. Strategien spielen eine Rolle, falls mehrere Regeln eines Termersetzungssystems angewendet werden. Die Strategie der innersten Reduktion schreibt vor, dass nur Terme ersetzt werden können, deren echte Teilterme nicht weiter reduziert werden können. Dies ist offensichtlich eine starke Einschränkung und erlaubt eine leichtere Terminierungsanalyse. Im Allgemeinen terminiert ein Termersetzungssystem allerdings nicht zwangsläufig, wenn es unter der Strategie der innersten Reduktion terminiert. Das Kriterium der innersten Terminierung ist daher notwendig, aber nicht hinreichend.

In [GTSK05] stellen Giesl et al. eine Verallgemeinerung des bisherigen Ansatzes zu einem Framework vor, der die eingangs des Kapitels erwähnte Modularität ermöglicht. Die gewöhnliche Termersetzung und die unter der Strategie der innersten Reduktion werden dabei durch einen allgemeineren Ersetzungsbegriff, nämlich $\mathcal{Q}$-eingeschränkte Termersetzung, gleichzeitig erfasst. Dazu wird ein zweites Termer-

setzungssystem $\mathcal{Q}$ eingeführt, dessen einziger Zweck darin besteht, bestimmte Ersetzungsschritte auszuschließen und so Strategien zu emulieren. Ein Ersetzungsschritt darf nur noch durchgeführt werden, wenn der zu kontrahierenden Redex bezüglich $\mathcal{Q}$ in einer Normalform ist. Durch Wahl von $\mathcal{Q}$ als leeres Termersetzungssystem wird $\mathcal{Q}$-beschränkte Termersetzung zur gewöhnlichen Termersetzung, während die Wahl von $\mathcal{Q} = \mathcal{R}$ die Ersetzung unter der Strategie der innersten Reduktion ergibt.

Definition 4.15. *Für die Termersetzungssysteme $\mathcal{R}$ und $\mathcal{Q}$ ist die $\mathcal{Q}$*-beschränkte Termersetzungsrelation $\overset{\mathcal{Q}}{\to}_{\mathcal{R}}$ *wie folgt definiert. Es gilt $s \overset{\mathcal{Q}}{\to}_{\mathcal{R}} t$ genau dann, wenn $s \to_{\mathcal{R}} t$ gilt (also eine Regel $l \to r \in \mathcal{R}$, eine Substitution σ und ein Kontext $C[]$ existieren, sodass $s \equiv C[l^\sigma]$ und $t \equiv C[r^\sigma]$) und die echten Teilterme von l^σ in Normalform bezüglich $\mathcal{Q}$ sind. Ein Termersetzungssystem heißt $\mathcal{Q}$*-terminierend, *genau dann wenn $\overset{\mathcal{Q}}{\to}_{\mathcal{R}}$ wohlfundiert ist.*

Bezüglich des leeren Termersetzungssystems $\mathcal{Q} = \emptyset$ ist jeder Term in Normalform, da keine Regel existiert und somit auch nicht angewendet werden kann. Aus der Definition folgt, dass $\to_{\mathcal{R}} = \overset{\emptyset}{\to}_{\mathcal{R}}$ für jedes Termersetzungssystem $\mathcal{R}$ gilt. Es ergibt sich die Beobachtung, dass das Hinzufügen von Regeln zu $\mathcal{Q}$ die Relation $\overset{\mathcal{Q}}{\to}_{\mathcal{R}}$ weiter einschränkt und mehr Terme in Normalform bezüglich $\overset{\mathcal{Q}}{\to}_{\mathcal{R}}$ sind. Andererseits würde ein Hinzufügen von Regeln zu dem grundlegenden Termersetzungssystem $\mathcal{R}$ die Relation erweitern, sodass mehr Reduktionsschritte möglich sind. Ein an der Anzahl der Regeln gemessen kleineres Termersetzungssystem hingegen beschreibt eine stärker eingeschränkte Relation. Diese gegensätzlichen Rollen präzisiert das folgende Korollar.

Korollar 4.16. *Seien $\mathcal{R}, \mathcal{R}', \mathcal{Q}$ und $\mathcal{Q}'$ Termersetzungssysteme und $\mathcal{R}' \subseteq \mathcal{R}$ sowie $\mathcal{Q}' \supseteq \mathcal{Q}$, dann folgt $\overset{\mathcal{Q}'}{\to}_{\mathcal{R}'} \subseteq \overset{\mathcal{Q}}{\to}_{\mathcal{R}}$.*

Die bisherigen Überlegungen zur gewöhnlichen, nicht $\mathcal{Q}$-eingeschränkten Termersetzung stellen daher den allgemeinsten Fall dar. Das Einschränken der Ersetzungsrelation kann die Terminierungseigenschaften nur dahingehend beeinflussen, dass bezüglich der nicht eingeschränkten Ersetzungsrelation nicht terminierenden Terme bei einer Einschränkung gegebenenfalls eine Normalform erhalten. Ein Term in Normalform behält auch bei Einschränkung weiterhin seine Normalformeigenschaft. Ein Term, der eine Normalform bezüglich der uneingeschränkten Relation besitzt, kann durch Einschränkung der Relation nur eine andere Normalform erhalten.

Neben der Verallgemeinerung der klassischen Termersetzung zur $\mathcal{Q}$-beschränkten Termersetzung sind noch weitere Verallgemeinerungen bei dem Wechsel zum Framework nötig. Es sei hier betont, dass die Definition des Abhängigkeitspaares jedoch

unverändert bleibt. Die Definition ist nur von syntaktischen Eigenschaften des Termersetzungssystems $\mathcal{R}$, genauer der Regeln des Termersetzungssystems, abhängig. Da das Termersetzungssystem $\mathcal{Q}$ nur für die Auswahl der Regeln eine Rolle spielt, kann es bei der Beschreibung der Struktur der Regeln nicht sinnvoll berücksichtigt werden.

Hingegen ist $\mathcal{Q}$ für den Begriff der Kette relevant. Eine $\mathcal{R}$-Kette beschreibt Verbindungen zwischen den Abhängigkeitspaaren $\mathrm{DP}(\mathcal{R})$ durch die transitive Hülle der Termersetzungsrelation $\twoheadrightarrow_{\mathcal{R}}$. Eine naheliegende Anpassung besteht nun darin, die transitive Hülle der $\mathcal{Q}$-beschränkten Termersetzungsrelation $\overset{\mathcal{Q}}{\twoheadrightarrow}_{\mathcal{R}}$ zu verwenden. Die Verallgemeinerung geht allerdings weiter. Die bisher definierte $\mathcal{R}$-Kette basiert auf den Abhängigkeitspaaren $\mathrm{DP}(\mathcal{R})$. Diese Beziehung zwischen dem Termersetzungssystem und den Abhängigkeitspaaren wird aufgehoben. Ketten werden auf beliebigen, endlichen Mengen von Abhängigkeitspaaren definiert. Giesl et al. verwenden den Ausdruck Termersetzungssystem auch für Mengen von Abhängigkeitspaaren. Auf diese rein strukturelle Interpretation wird hier allerdings nur hingewiesen, um die Notation zu begründen. Das verallgemeinerte Kettenkonzept wird als $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette bezeichnet.

Definition 4.17. *Seien $\mathcal{Q}$ und $\mathcal{R}$ Termersetzungssysteme und $\mathcal{P}$ eine Menge von Abhängigkeitspaaren. Eine möglicherweise unendliche Folge von Abhängigkeitspaaren aus der Menge $\mathcal{P}$*

$$\langle s_i, t_i \rangle, \langle s_{i+1}, t_{i+1} \rangle, \ldots$$

ist genau dann eine $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette, *wenn es Substitutionen σ_i gibt, sodass $t_i^{\sigma_i} \overset{\mathcal{Q}}{\twoheadrightarrow}_{\mathcal{R}} s_{i+1}^{\sigma_{i+1}}$ für alle unmittelbar aufeinander folgenden Paare $\langle s_i, t_i \rangle$ und $\langle s_{i+1}, t_{i+1} \rangle$ gilt.*

Das Konzept der $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette schließt das ursprüngliche Konzept der $\mathcal{R}$-Kette mit ein. Durch die Wahl von $\mathcal{P} = \mathrm{DP}(\mathcal{R})$ und $\mathcal{Q} = \emptyset$ ergibt sich, dass jede $\mathcal{R}$-Kette auch eine verallgemeinerte $(\mathrm{DP}(\mathcal{R}), \emptyset, \mathcal{R})$-Kette ist.

Da die Definition einer $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette eine Verbindung zwischen der Menge $\mathcal{P}$ und dem Termersetzungssystem $\mathcal{R}$ nicht vorschreibt, muss es auch Ketten geben, die sich nicht zur Terminierungsanalyse eignen. Der Begriff der $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette scheint damit zu allgemein. In das Framework sollen sich allerdings auch neue, bisher unbekannte Techniken integrieren lassen. Die Definition der $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Ketten kann daher nicht weiter eingeschränkt werden, ohne die Modularität des Frameworks zu beschränken.

Dabei bilden $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Ketten die Schnittstelle zwischen dem Framework und konkreten, möglicherweise neuen Terminierungsanalysemethoden.

Die Zerlegung des Abhängigkeitsgraphen in seine Kreise in Abschnitt 4.2 illustriert ein mögliches Vorgehen entsprechend diesem Muster. Diese Methode wird im

Folgenden in das Framework integriert. Dabei wird durch die Aufteilung in Kreise nicht mehr die gesamte Menge der Abhängigkeitspaare betrachtet, sondern lediglich solche, die auf einem Kreis liegen.

Den grundlegenden Zusammenhang zwischen $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Ketten, Abhängigkeitspaaren und Terminierung stellt das folgende Theorem aus [GTSK05] auf.

Theorem 4.18. *Für ein Termersetzungssystem $\mathcal{R}$ gilt, dass es genau dann $\mathcal{Q}$-terminierend ist, wenn keine unendliche* $(\mathrm{DP}(\mathcal{R}), \mathcal{Q}, \mathcal{R})$*-Kette existiert.*

Beweis. Wir zeigen zunächst, dass die Abwesenheit einer $(\mathrm{DP}(\mathcal{R}), \mathcal{Q}, \mathcal{R})$-Kette hinreichend für die Terminierung von $\mathcal{R}$ ist. Gibt es keine unendliche $(\mathrm{DP}(\mathcal{R}), \mathcal{Q}, \mathcal{R})$-Kette, so kann es aufgrund von Korollar 4.16 auch keine unendliche $(\mathrm{DP}(\mathcal{R}), \emptyset, \mathcal{R})$-Kette geben und somit keine $\mathcal{R}$-Ketten. Es folgt aus Theorem 4.8, dass $\mathcal{R}$ terminiert.

Die Notwendigkeit ergibt sich aus folgender Überlegung. Wenn $\mathcal{R}$ terminiert, $\mathcal{Q}'$-terminiert es ebenfalls für $\mathcal{Q}' = \emptyset$. Da $\emptyset \subseteq \mathcal{Q}$ für alle Termersetzungssysteme $\mathcal{Q}$ gilt, folgt ebenfalls wegen Korollar 4.16, dass $\mathcal{R}$ auch $\mathcal{Q}$-terminiert. Jede $(\mathrm{DP}(\mathcal{R}), \mathcal{Q}, \mathcal{R})$-Kette ist auch eine $(\mathrm{DP}(\mathcal{R}), \emptyset, \mathcal{R})$-Kette. Ferner existieren nach 4.8, keine unendlichen $\mathcal{R}$-Ketten und somit auch keine $(\mathrm{DP}(\mathcal{R}), \emptyset, \mathcal{R})$-Ketten. □

Die Arbeitsweise des Frameworks besteht wie bei dem bisherigen Ansatz nach [AG00] darin, nachzuweisen, dass es keine unendlichen Ketten gibt. Der wesentliche Unterschied liegt nun darin, dass in dem Framework das ursprüngliche Problem in Teilprobleme zerlegt wird. Das Framework beschreibt dabei nicht die Zerlegung der Probleme, sondern die Art und Weise, in der verschiedene Zerlegungen zusammengeführt werden können. Ebenso sind die Methoden zur Lösung der Teilprobleme nicht Teil des Frameworks, sondern werden von dem Framework eingebunden. Tatsächlich behandelt das Framework sowohl zerlegende als auch lösenden Methoden uniform. Die Teilprobleme heißen dem Namen des Frameworks entsprechend Abhängigkeitspaarprobleme, die zusammengeführten Schritte Abhängigkeitspaarprozessoren. Grob formuliert ist das Ziel, für ein Abhängigkeitspaarproblem entweder die Existenz oder die Abwesenheit von unendlichen $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Ketten zu beweisen. Als drittes Ergebnis kann ein Prozessor aus einem gegebenen Problem auch eine Menge von einfacheren Abhängigkeitspaarproblemen erzeugen. Die Lösung dieser einfacheren Probleme kann dann mit anderen Prozessoren versucht werden. Das Framework beschreibt dabei die Bedingungen, die eine Zusammenführung der Lösung der Teilprobleme zu einer korrekten Gesamtlösung erlaubt.

Die formale Definition des Frameworks ergibt sich aus der Definition seiner Bestandteile, nämlich den Abhängigkeitspaarproblemen und Abhängigkeitspaarprozes-

soren. Zunächst muss jedoch der Begriff der minimalen $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette eingeführt werden.

Definition 4.19. *Eine $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette bestehend aus den Paaren $\langle s_i, t_i \rangle$ und den Substitutionen σ_i für $i \in I \subseteq \mathbb{N}$ heißt* minimal, *wenn keiner der echten Teilterme von $t_i^{\sigma_i}$ Anfang einer unendlichen Reduktionsfolge unter $\xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ ist.*

Diese Definition ist analog zu der Verwendung des Begriffs „minimal" im Beweis zu Theorem 4.8. Arts und Giesl nutzen in [AG00] aus, dass für $\mathcal{R}$-Ketten die Abhängigkeitspaare $\mathrm{DP}(\mathcal{R})$ die Rolle der Menge $\mathcal{P}$ eingenommen haben. Dadurch ist es möglich, aus der Nichtexistenz minimaler, unendlicher Ketten auf die Nichtexistenz jeglicher unendlicher Ketten zu schließen. Durch die Verallgemeinerung zu $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Ketten sind allerdings Verarbeitungsschritte möglich, die diesen Schluss nicht mehr zulassen und daher auch die Abwesenheit beliebiger Ketten explizit überprüft werden muss. Die Definition eines Abhängigkeitspaarproblems nach Giesl et al. trägt dem in folgender Form Rechnung.

Definition 4.20. *Ein Abhängigkeitspaarproblem ist ein Quadrupel $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ aus einer Menge von Abhängigkeitspaaren $\mathcal{P}$, zwei Termersetzungssystemen $\mathcal{Q}$ und $\mathcal{R}$ und einer Markierung $f \in \{m, a\}$.*

Die Markierung m (für minimal) zeigt an, dass für dieses Teilproblem geprüft werden soll, ob es minimale, unendliche Ketten gibt. Hingegen zeigt a (für arbitrary) an, dass die Existenz beliebiger unendlicher Ketten analysiert werden soll. Für das Ursprungsproblem reicht es aus, die Existenz minimaler unendlicher Ketten auszuschließen. Die Argumentation dafür ergibt sich direkt aus dem Beweis von Theorem 4.8. Als Ursprungsproblem wird deswegen im späteren $(\mathrm{DP}(\mathcal{R}), \mathcal{Q}, \mathcal{R}, m)$ aufgestellt. Für die Teilprobleme lassen sich nun zwei Eigenschaften formulieren.

Definition 4.21. *Ein Abhängigkeitspaarproblem $P = (\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$, ist genau dann* finit, *wenn $f = a$ und keine unendliche $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette existiert oder wenn $f = m$ und keine minimale $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette existiert.*

Definition 4.22. *Ein Abhängigkeitspaarproblem $P = (\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ ist* infinit, *wenn es nicht finit ist oder wenn $\mathcal{R}$ nicht $\mathcal{Q}$-terminiert.*

Die Definitionen sind offensichtlich nicht symmetrisch, auch wenn die Namen anderes vermuten lassen. Es lassen sich in der Tat Beispiele finden, die sowohl finit als auch infinit sind. Giesl et al. geben folgendes Beispiel an.

Beispiel 4.23. *Sei* $\mathcal{P} = \{\langle A, B\rangle\}, \mathcal{Q} = \emptyset$ *und* $\mathcal{R} = \{a \to a, a \to b, b \to c\}$*. Das Abhängigkeitspaarproblem* $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, m)$ *ist finit, da es keine unendliche* $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$*-Kette gibt, und infinit da* $\mathcal{R}$ *wegen* $a \to a$ *offensichtlich weder terminiert noch* $\mathcal{Q}$*-terminiert.*

Für das Ursprungsproblem $(\mathrm{DP}(\mathcal{R}), \mathcal{Q}, \mathcal{R}, m)$ kann dieses Problem nicht auftreten, da es im Widerspruch zu Theorem 4.18 stehen würde. Das Ursprungsproblem kann also entweder finit oder infinit, aber nicht beides sein.

Auch in der weiteren Verarbeitung stellt der Spezialfall der Probleme, die sowohl finit als auch infinit sind, keinen Widerspruch dar. Tritt ein solches Problem auf, terminiert das ursprüngliche Termersetzungssystem nicht. Dabei kann das kritische Teilproblem entweder direkt als Zeuge dienen oder es wird bei der Zerlegung in Teilprobleme sichergestellt, dass es ein weiteres Teilproblem gibt, welches infinit und nicht finit ist. Diese Zerlegung erfolgt mittels Abhängigkeitspaarprozessoren.

Definition 4.24. *Ein Abhängigkeitspaarprozessor (kurz APP) ist eine berechenbare Funktion* Proc*, welche ein Abhängigkeitspaarproblem entweder auf den Wert* no *oder auf eine Menge von weiteren Abhängigkeitspaarproblemen abbildet.*

Ein APP ist korrekt*, falls für alle Abhängigkeitspaarprobleme* d *gilt*

$$(\mathrm{Proc}(d) \neq \mathrm{no} \wedge \forall d' \in \mathrm{Proc}(d) : d' \textit{ ist finit}) \implies d \textit{ ist finit.}$$

Ein APP ist vollständig*, falls für alle Abhängigkeitspaarprobleme* d *gilt*

$$(\mathrm{Proc}(d) = \mathrm{no} \wedge (\exists d' \in \mathrm{Proc}(d) : d' \textit{ ist infinit}) \implies d \textit{ ist infinit.}$$

Der Eindruck, dass ein korrekter und vollständiger APP eine Verletzung des Halteproblems darstellt, erweist sich als falsch. Vielmehr ist durch das Halteproblem sichergestellt, dass es keinen APP Proc gibt, sodass für alle Abhängigkeitspaarprobleme d entweder $\mathrm{Proc}(d) = \emptyset$ oder $\mathrm{Proc}(d) = \mathrm{no}$ gilt.

Die grundlegende Idee des Dependency Pair Frameworks stellt sich nun wie folgt dar. Für die Termersetzungssysteme $\mathcal{R}$ und $\mathcal{Q}$ kann ein Baum, dessen Blätter entweder mit $\mathrm{yes}, \mathrm{no}$ oder einem Abhängigkeitspaarproblem beschriftet sind, nach folgender Vorschrift konstruiert werden.

Die Wurzel des Baums erhält als Beschriftung das Abhängigkeitspaarproblem $(\mathrm{DP}(\mathcal{R}), \mathcal{Q}, \mathcal{R}, m)$. Für jeden Knoten, der mit einem Abhängigkeitspaarproblem d beschriftet ist, wird ein korrekter APP Proc gewählt. Ist $\mathrm{Proc}(d)$ eine nicht-leere Menge, so erhält der Knoten mit der Beschriftung d jeweils ein Kind für jedes Element aus $\mathrm{Proc}(d)$, welcher mit dem jeweiligen Problem beschriftet wird. Gilt

$\text{Proc}(d) = \emptyset$, so wird nur ein Kind an den Knoten mit der Beschriftung d angefügt, das die Beschriftung yes trägt und selbst ein Blatt darstellt. Gilt $\text{Proc}(d) = \text{no}$, so wird ebenfalls nur ein Blatt mit der Beschriftung no angefügt.

Das Termersetzungssystem $\mathcal{R}$ $\mathcal{Q}$-terminiert, wenn alle Blätter dieses Baumes mit yes beschriftet sind. Das Termersetzungssystem $\mathcal{R}$ $\mathcal{Q}$-terminiert nicht, wenn mindestens ein Blatt dieses Baumes mit no beschriftet ist und alle Abhängigkeitspaarprozessoren auf dem Pfad von der Wurzel zu eben diesem Blatt vollständig sind.

Den Abhängigkeitsgraphen aus Abschnitt 4.2 fassen Giesl et al. nun als einen Abhängigkeitspaarprozessor auf. Auch in diesem Abschnitt diente der Abhängigkeitsgraph dazu, das Terminierungskriterium zu verfeinern, indem die Zusammenhangskomponenten betrachtet wurden. Dieses Vorgehen wird nun im Rahmen des Frameworks von Giesl et al. formalisiert.

Definition 4.25. *Der* $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Abhängigkeitsgraph *für ein Abhängigkeitspaarproblem* $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ *ist ein gerichteter Graph mit den Paaren aus* $\mathcal{P}$ *als Knoten und*

$$E = \{(\langle s,t\rangle, \langle u,v\rangle) \in \mathcal{P} \times \mathcal{P} | \exists \sigma : t^\sigma \overset{\mathcal{Q}}{\twoheadrightarrow}_R u^\sigma\}$$

als Kantenmenge.

Wieder besteht zwischen den Knoten genau dann eine Kante, wenn die beiden Abhängigkeitspaare eine zweielementige $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette bilden. Auch dieser Graph ist im Allgemeinen nicht berechenbar, da die Definition mit der des Abhängigkeitsgraphen aus 4.9 zusammenfällt, falls $(\mathcal{P}, \mathcal{Q}, \mathcal{R}) = (\text{DP}(\mathcal{R}), \emptyset, \mathcal{R})$. Es muss also wieder eine Approximation verwendet werden, die den vollständigen Abhängigkeitsgraphen enthält. Dazu können die Überlegungen aus Abschnitt 4.3 verwendet werden.

Mithilfe des $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-*Abhängigkeitsgraph* lassen sich nun Abhängigkeitspaarprozessoren definieren, die möglicherweise unterschiedliche Methoden der Approximation verwenden. Die exakte Art der Approximation ist nicht interessant, vorausgesetzt wird lediglich, dass der Abhängigkeitsgraph vollständig enthalten ist.

Von dem Graphen bzw. seiner Approximation ausgehend, werden die starken Zusammenhangskomponenten und nicht mehr die einzelnen Kreise betrachtet, wie es noch in dem ursprünglichen, integrierten Ansatz aus Abschnitt 4.2 der Fall war. Zusammenhangskomponenten sind jene Teilmengen $\mathcal{P}'$ von $\mathcal{P}$, sodass zwischen zwei beliebigen Paaren aus $\mathcal{P}'$ ein Pfad existiert, der ausschließlich aus Knoten aus $\mathcal{P}'$ besteht. Gibt es keine größere Zusammenhangskomponente in $\mathcal{P}$, die $\mathcal{P}'$ beinhaltet, so ist $\mathcal{P}'$ eine starke Zusammenhangskomponente. Starke Zusammenhangskomponenten sind damit notwendigerweise disjunkt. Kreise bilden immer Zusammenhangs-

komponenten, jedoch nicht notwendigerweise starke. Ein Beispiel dafür sind zwei Kreise, die bis auf einen Knoten disjunkt sind.

Während in dem integrierten Ansatz die Methode bekannt war, mit der sichergestellt wurde, dass die Kreise keine unendlichen Ketten zulassen, kann in dem modularen Ansatz keine Aussage getroffen werden, mit welchen weiteren Abhängigkeitspaarprozessoren die Ergebnisse des auf dem Abhängigkeitsgraphen basierenden APP weiterverarbeitet werden. Für das Beispiel der zwei in nur einem Knoten verbundenen Kreise wurde in dem Terminierungskriterium des integrierten Ansatzes gefordert, dass in jedem Kreis ein strikter Schritt bezüglich einer wohlfundierten Ordnung erfolgt. Eine unendliche Kette, die notwendigerweise wenigstens einen dieser Kreise unendlich oft durchläuft, würde auch eine unendliche Folge von strikten Schritten bezüglich der gewählten Ordnung enthalten, im Widerspruch zu der Wohlfundiertheit. Dieses Argument beruht auf der Verwendung des selben Kriteriums für alle Kreise.

Eine Aufteilung in Kreise ist nicht für einen APP geeignet, da die Teilprobleme der Zerlegung durch einen APP nicht notwendigerweise durch den selben Prozessor verarbeitet werden. Durch eine Aufteilung in starke Zusammenhangskomponenten kann allerdings ein zur Terminierungsanalyse geeigneter APP konstruiert werden. Mit dem folgenden Theorem und dem zugehörigen Beweis belegen Giesl et al. diese Aussage.

Theorem 4.26. *Sei* Proc *ein APP und* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *Mengen, die die starken Zusammenhangskomponeten des* $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$*-Abhängigkeitsgraphen eines beliebigen Problems* $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ *enthalten. Falls stets*

$$\mathrm{Proc}((\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)) = \{(\mathcal{P}_1, \mathcal{Q}, \mathcal{R}, f), \ldots, (\mathcal{P}_n, \mathcal{Q}, \mathcal{R}, f)\}$$

gilt, ist Proc *korrekt und vollständig.*

Beweis. In einer unendlichen $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette werden nach einem endlichen Präfix nur Knoten aus einer starken Zusammenhangskomponente besucht. Würden Abhängigkeitspaare aus mehreren starken Zusammenhangskomponenten unendlich oft in einer unendlichen Kette vorkommen, so müsste es Verbindungen von einer starken Zusammenhangskomponente in die Andere und zurück geben. Damit wäre die Vereinigung dieser starken Zusammenhangskomponenten allerdings wieder eine größere Zusammenhangskomponente, im Widerspruch zu der Definition starker Zusammenhangskomponenten.

Ist das Problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ finit, so sind auch alle Teilprobleme finit, da es kei-

ne unendlichen $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Ketten gibt und damit auch keine unendlichen Ketten in den Teilproblemen. Ist das Problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ aber nicht finit, so gibt es mindestens eine Komponente $\mathcal{P}_i$ mit mindestens einem Knoten, der unendlich oft besucht wird. Folglich gäbe es auch eine unendliche $(\mathcal{P}_i, \mathcal{Q}, \mathcal{R})$-Kette und das Teilproblem $(\mathcal{P}_i, \mathcal{Q}, \mathcal{R}, f)$ ist nicht finit. Der beschriebene APP ist somit korrekt.

Die Vollständigkeit folgt aus dem folgenden Lemma 4.27, da $\mathcal{P}_i \subseteq \mathcal{P}$ gilt. □

Das folgende Lemma aus [GTSK05] ist ein leicht stärkeres Resultat, als es für den Beweis von Theorem 4.26 nötig ist.

Lemma 4.27. *Ein APP* Proc *ist vollständig, wenn für alle Probleme* $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ *und alle Teilprobleme* $(\mathcal{P}', \mathcal{Q}', \mathcal{R}', f') \in \mathrm{Proc}((\mathcal{P}, \mathcal{Q}, \mathcal{R}, f))$ *die folgenden Teilmengenbeziehungen gelten:*

- $\mathcal{P}' \subseteq \mathcal{P}$
- $\mathcal{Q}' \supseteq \mathcal{Q}$
- $\mathcal{R}' \subseteq \mathcal{R}$

Beweis. Der Beweis erfolgt durch Widerspruch. Angenommen für ein nicht infinites Problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ gäbe es ein Teilproblem $(\mathcal{P}', \mathcal{Q}', \mathcal{R}', f') \in \mathrm{Proc}((\mathcal{P}, \mathcal{Q}, \mathcal{R}, f))$, dass infinit ist. Da $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ nicht infinit ist, muss $\mathcal{R}$ $\mathcal{Q}$-terminierend sein und $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ ist finit. Ein beliebiges Problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, m)$ ist finit, auch wenn eine nicht minimale, unendliche $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette existiert. Für das gegebene Problem $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, f)$ gibt es allerdings keine nicht minimalen, unendlichen Ketten, da $\mathcal{R}$ $\mathcal{Q}$-terminierend ist und somit alle Ketten minimal sind.

Aus der $\mathcal{Q}$-Terminierungseigenschaft von $\mathcal{R}$ mit Korollar 4.16 folgt, dass $\mathcal{R}'$ $\mathcal{Q}'$-terminierend ist, da $\mathcal{R}' \subseteq \mathcal{R}$ und $\mathcal{Q}' \supseteq \mathcal{Q}$ nach Voraussetzung von Lemma 4.27 und somit $\xrightarrow{\mathcal{Q}'}_{\mathcal{R}'} \subseteq \xrightarrow{\mathcal{Q}}_{\mathcal{R}}$. Das angenommene Widerspruchsproblem $(\mathcal{P}', \mathcal{Q}', \mathcal{R}', f')$ muss also eine unendliche $(\mathcal{P}', \mathcal{Q}', \mathcal{R}')$-Kette enthalten. Wegen $\xrightarrow{\mathcal{Q}'}_{\mathcal{R}'} \subseteq \xrightarrow{\mathcal{Q}}_{\mathcal{R}}$ muss diese aber auch eine $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette sein. Dies ist allerdings nach den obigen Überlegungen unmöglich. □

5 Automatisierte Terminierungsanalyse von Haskell-Programmen

In diesem Kapitel wird ein automatisches Verfahren zur Terminierungsanalyse von Haskell-Programmen vorgestellt, das von Giesl et al. entwickelt wurde und zuerst in [GSSKT06] beschrieben wurde. Eine Weiterentwicklung des Ansatzes findet sich in [GRSK+11].

Sowohl Haskell [PJ03] als auch Termersetzungssysteme sind Turing-vollständig. Demnach gibt es für jedes Haskell-Programm ein semantisch äquivalentes Termersetzungssystem, welches mit dem Dependency Pair Framework automatisch analysiert werden kann. Dieser naive Ansatz erweist sich jedoch laut [GRSK+11] als nicht geeignet, da die resultierenden Termersetzungssysteme mit hoher Wahrscheinlichkeit zu komplex sind, sodass mit den bekannten Methoden, einschließlich dem Dependency Pair Framework, weder die Terminierung gezeigt noch ein Gegenbeispiel ohne Normalform gefunden werden kann.

Daher wird ein Vorgehen beschrieben, welches von Giesl et al. zuerst in [GSSKT06] vorgestellt wurde und ebenfalls auf dem Dependency Pair Framework basiert. Insbesondere eignet es sich für die automatisierte Analyse. Eine Implementierung wurde in das System AProVE (kurz für „Automated Program Verification Environment“) integriert. Eine Beschreibung von AProVE befindet sich in [GTSKF04]. Die Weiterentwicklung des Ansatzes in [GRSK+11] verbessert die Methode vor allem in Bezug auf Typklassen und die Repräsentation von Ausdrücken höherer Ordnung. Dabei enthält [GRSK+11] eine vollständige Rekapitulation der Resultate aus [GSSKT06]. Die in dem vorliegenden Kapitel beschriebenen Methoden sind in beiden Artikeln beschrieben.

Im ersten Abschnitt dieses Kapitels wird die Sprache Haskell kurz vorgestellt und die Teilmenge der Sprache beschrieben, die zur Terminierungsanalyse herangezogen wird. Ferner wird der Terminierungsbegriff aus [GRSK+11] für Haskell beschrieben

und seine Eignung diskutiert. In Abschnitt 5.2 wird das Konzept des Terminierungsgraphen eingeführt und seine Eigenschaften erläutert. Im darauffolgenden Abschnitt wird ein Verfahren beschrieben, dass aus einem Terminierungsgraphen eine Reihe von Abhängigkeitspaarproblemen generiert. Auf die Betrachtung von Typklassen und Funktionen höherer Ordnung wird im Folgenden verzichtet.

Die meisten Begriffe aus den vorherigen Kapiteln, wie zum Beispiel Normalform und Abhängigkeitspaar, können unverändert übernommen werden. An die Stelle von Termen treten nun allerdings Haskell-Ausdrücke. Einige Begriffe werden daher weiter gefasst als bisher.

5.1 Die Sprache Haskell

In diesem Abschnitt werden die nötigen Begriffe und Konzepte der Sprache Haskell vorgestellt. Auf eine detaillierte Einführung wird an dieser Stelle verzichtet und stattdessen auf folgende Lehrbücher hingewiesen. In der deutschsprachigen Literatur sind vor allem [Dob12] und [Blo11] zu nennen. Besonders bekannte Einführungen in englischer Sprache sind [Lip11] und [OSG09]. Weniger bekannt und im Vergleich zu den bisher genannten Einführungen eher theoretisch ausgerichtet ist [Bir98].

Zunächst wird ein grober Überblick über die Sprache anhand der üblichen Klassifikationen vermittelt. Danach wird das grundlegende Funktions- und Auswertungskonzept in Haskell informell eingeführt und zum Abschluss die Formalisierung der Semantik von Teilen der Sprache vorgestellt, wie sie in [GSSKT06] und [GRSK+11] verwendet wird. Wir beginnen nun mit der Klassifikation von Haskell.

5.1.1 Klassifikation

Haskell fällt primär in die Rubrik der funktionalen Sprachen. Diese zeichnen sich durch die Möglichkeit der Definition von Funktionen höherer Ordnung aus. Dadurch wird im Vergleich zu nicht funktionalen Sprachen eine höhere Ausdrucksstärke erreicht. Die Flexibilität funktionaler Sprachen stellt jedoch eine besondere Herausforderung in der Terminierungsanalyse dar.

Haskell verfügt im Gegensatz zu älteren Vertretern der funktionalen Sprachen wie zum Beispiel Lisp [McC60] über ein statisches Typsystem, sprich, bereits während der Übersetzung ist für jeden Ausdruck ein Typ bekannt. Dabei fußt Haskells Typsystem auf den Arbeiten von Hindley [Hin69] und Milner [Mil78] ähnlich wie die Typsysteme der Sprachen der ML-Familie (zum Beispiel SML, siehe [MTH90]). Für diese Systeme existiert ein Algorithmus, der es erlaubt, den allgemeinsten Typ

eines Ausdrucks zu inferieren. Haskells Typsystem geht über reine Hindley-Milner-Typsysteme hinaus, sodass die automatische Bestimmung des Typs nicht für alle Ausdrücke gelingen kann. Durch die stets gegebene Möglichkeit Typen mit Typannotationen explizit zu spezifizieren, können derartige Programme dennoch übersetzt werden, da die statische Typüberprüfung möglich bleibt. In den meisten praktisch relevanten Fällen treten jedoch nur Typen auf, die automatisch inferiert werden können und so kann weitestgehend auf explizite Typannotationen verzichtet werden.

Durch die Verwendung von Typvariablen erlauben Hindley-Milner-Typsysteme und auch Haskells Typsystem, von konkreten Typen zu abstrahieren und unterstützen damit parametrische Polymorphie (siehe [Str00a]). Haskell unterstützt mit Typklassen auch Ad-hoc-Polymorphie (siehe ebenfalls [Str00a]), also das Überladen von Funktionen für unterschiedliche Typen. Im Folgenden werden Typklassen nur selten benötigt.

Haskell ist ferner eine referentiell transparente Sprache. Dies bedeutet, dass jeder Ausdruck in Haskell stets zu demselben Ergebnis ausgewertet wird, die Resultate also unabhängig vom Zeitpunkt der Evaluation sind. Haskell-Programme sind in diesem Sinne zustandslos. Es gibt keine Anweisungen, die Zustandsänderungen beschreiben, sondern nur Ausdrücke, die ausgewertet werden können. Die formale Grundlage ist das Lambda-Kalkül [Chu36]. Dieser Turing-vollständige Mechanismus kommt ebenfalls ohne einen Zustandsbegriff aus. Ein Haskell-Programm ist lediglich ein Ausdruck, dessen Auswertung der Ausführung des Programms entspricht.

Referentiell transparente, funktionale Sprachen werden auch als rein funktionale Sprachen bezeichnet. Eine Konsequenz der referentiellen Transparenz ist, das überschreibende Änderungen von Speicherinhalten nur dann vorgenommen werden, wenn die bisherigen Inhalte nicht länger benötigt werden. Dies sollte als Regel auch in imperativen Programmen befolgt werden. In rein funktionalen Sprachen stellt dies allerdings der Übersetzer sicher. Korrekte Software ist daher mit Haskell einfacher zu entwickeln, als in Sprachen, die den Speicherzugriff weniger restriktiv handhaben. Besonders effiziente Software ist mit Haskell jedoch dann schwieriger zu entwickeln, wenn die Effizienz eines Algorithmus von der Fähigkeit zur direkten Manipulation des Speichers abhängt. Haskell-Programme sind daher im Bereich der eingebetteten Systeme nicht zu erwarten.

Eine weitere Konsequenz der referentiellen Transparenz ist, dass alle Auswertungsreihenfolgen zu demselben Ergebnis führen, vorausgesetzt, dass sie tatsächlich zu einem Ergebnis führen und nicht eine unendlichen Folge von Evaluationen auslösen. Entscheidend für die Reihenfolge der Auswertung ist die Strategie, mit der der nächste Ausdruck ausgewählt wird. In imperativen Sprachen werden für die Evaluation

von Ausdrücken gewöhnlich strikte Strategien verwendet. Diese sind dadurch gekennzeichnet, dass, bevor eine Funktion ausgewertet wird, ihre Argumente evaluiert werden. Dies entspricht den innersten Reduktionen in Termersetzungssystemen.

Haskell verwendet eine nicht-strikte Strategie. Manche Autoren bezeichnen diesen Umstand als verzögerte Auswertung, da Argumente von Funktionen erst bei Bedarf ausgewertet werden. Der Vorteil dieses Ansatzes besteht darin, dass auch Ausdrücke verwendet werden können, die keine Normalform besitzen. Dies wird möglich, da nicht versucht wird diese Ausdrücke vollständig auszuwerten. Um die Strategie zu beschreiben, die ein solches Vorgehen erlaubt, muss zunächst erläutert werden, wie Funktionen in Haskell definiert werden. Dies geschieht im folgenden Abschnitt.

5.1.2 Funktionen und Funktionsauswertung

Haskell unterscheidet zwei Arten von Funktionen, definierte Funktionen und Konstruktoren. Erstere werden syntaktisch durch Bezeichner repräsentiert, die mit einem Kleinbuchstaben beginnen. Die Bezeichner der Konstruktoren beginnen hingegen stets mit einem Großbuchstaben. Im Folgenden verwenden wir den Ausdruck Funktionssymbol für den Bezeichner einer definierten Funktion und Konstruktorsymbol für den Bezeichner eines Konstruktors. Da ohne Berücksichtigung von Haskells Modulsystem und die Definition lokaler Funktionen die Symbole eindeutig sind, werden die Funktionen mit ihren Funktionssymbolen identifiziert. Gleiches gilt für die Konstruktoren. Darüber hinaus treten noch Variablen in Erscheinung. Falls nichts anderes angemerkt wird, handelt es sich bei f, g und h um Funktionssymbole, bei x, y, z um Variablen und bei C und D um Konstruktorsymbole. Wir beschränken uns analog zu [GRSK$^+$11] auf die Betrachtung korrekt getypter Ausdrücke. Es wird τ für konkrete Typen, α und β für Typvariablen verwendet. Die Funktionsanwendung wird in Haskell durch Juxtaposition notiert. Es gilt $f\ t_1\ t_2 = ((f\ t_1)\ t_2)$.

Eine definierte Funktion wird durch eine oder mehrere Funktionsdefinitionsgleichungen der Form $l = r$ beschrieben. Während auf der rechten Seite ein beliebiger Ausdruck stehen darf, so sind auf der linken Seite nur sogenannte Muster erlaubt. Ein Muster $l \equiv f\ l_1 \dots l_n$ beginnt stets mit dem Funktionssymbol gefolgt von n Argumenten. Die Muster aller Funktionsdefinitionsgleichungen einer Funktion haben stets dieselbe Anzahl der Argumente. Diese Anzahl wird als Arität der Funktion bezeichnet und wie gehabt mit $\mathrm{ari}(f)$ für eine Funktion f notiert. Als Argumente sind in einem Muster nur Konstruktorsymbole und Variablen erlaubt. Letztere dürfen in einem Muster nur einfach vorkommen.

Für die Auswertung eines Ausdrucks $t \equiv f\ t_1 \dots t_n$ wird nun nach einer Definiti-

onsgleichung $l = r$ gesucht, sodass es eine Substitution σ gibt, für die $l^\sigma \equiv t$ gilt. Dieses Verfahren heißt Musterabgleich. Ist eine entsprechende Gleichung gefunden, so wird t zu r^σ evaluiert. Um eine Substitution zu finden, kann es notwendig sein, dass der Ausdruck t, der im Gegensatz zu dem Muster l auch definierte Funktionssymbole beinhalten kann, weiter evaluiert werden muss. Dies geschieht wiederum mit dem gerade beschriebenen Verfahren, solange bis entschieden werden kann, ob es eine Substitution σ gibt.

Die Definitionsgleichungen werden beim Musterabgleich in der Reihenfolge überprüft, in der sie im Quelltext stehen. Kann keine zulässige Gleichung gefunden werden, so ist f nicht wohldefiniert und t wird als Fehlerausdruck bezeichnet. Eine vollständige Auswertung eines Ausdrucks ist nicht erforderlich, um zu entscheiden, ob es sich um einen Fehlerausdruck handelt. Seien C und D unterschiedliche Konstruktorsymbole und g und f Funktionssymbole, wobei die durch f bezeichnete Funktion die einzige Definitionsgleichung $f\ (C\ x) = x$ besitzt. Der Ausdruck $t \equiv f\ (D\ g)$ ist ein Fehlerausdruck. Da D ein Konstruktor ist, kann $D\ g$ nur weiter ausgewertet werden, indem g ausgewertet wird. Dies ist allerdings nicht zielführend. Ist das Ergebnis dieser Auswertung der Ausdruck s, so ergibt sich $D\ s$. Nach wie vor existiert keine Substitution σ, sodass $f\ (C\ x)^\sigma \equiv D\ s$ gilt.

Zusätzlich zu den Funktionsgleichungen kann der Typ einer Funktion angegeben werden. Für solche Typannotationen wird die Syntax $f :: \tau$ verwendet, um für die Funktion f den Typ τ festzulegen. Erfolgt keine Annotation, wird der allgemeinste Typ durch den Inferenzalgorithmus bestimmt.

Neben definierten Funktionen kennt Haskell noch die bereits erwähnten Konstruktoren. Diese werden im Rahmen der Definition von Datentypen eingeführt. Eine Datentypdefinition hat die Form

$$\texttt{data}\ \tau = C_1\ \tau_{1,1} \dots \tau_{1,k_1}\ \mid\ \dots\ \mid C_n\ \tau_{n,1} \dots \tau_{n,k_n}.$$

Mit obiger Deklaration wird ein Typ τ eingeführt, der n Konstruktoren $C_1\ \dots\ C_n$ besitzt. Jeder Konstruktor C_i stellt eine k_i-stellige Funktion des Typs $\tau_{i,1} \to \dots \to \tau_{i,k_i} \to \tau$ dar. Damit ist für k_i Ausdrücke $t_1, \dots, t_{k_i}$ mit den zugehörigen Typen $\tau_{i,1}, \dots, \tau_{i,k_i}$ der Ausdruck $t \equiv C_i\ t_1 \dots t_{k_i}$ vom Typ τ.

Anstelle von konkreten Typen können mit `data` auch Typkonstruktoren definiert werden. Dazu wird die Form

$$\texttt{data}\ \tau\ \alpha_1 \dots \alpha_m = C_1\ \tau_{1,1} \dots \tau_{1,k_1}\ \mid\ \dots\ \mid C_n\ \tau_{n,1} \dots \tau_{n,k_n}$$

verwendet. Die Typvariablen $\alpha_1, \ldots, \alpha_m$ können anstelle von konkreten Typen auf der rechten Seite als Konstruktorargumente verwendet werden. So definierte Konstruktoren sind polymorphe Funktionen. Ein vordefinierter, binärer Typkonstruktor in Haskell ist $\rightarrow$, mit dem Funktionstypen repräsentiert werden.

Ein Ausdruck, der mit einem Konstruktor beginnt, wird als Ausdruck in Kopfnormalform bezeichnet. Eine Erweiterung der Kopfnormalform ist die schwache Kopfnormalform. Sie umfasst neben Ausdrücken in Kopfnormalform auch Ausdrücke, die zu Funktionswerten evaluiert wurden. Funktionswerte sind dabei entweder Funktionsapplikationen auf zu wenige Argumente oder Lambda-Ausdrücke. Ausdrücke in schwacher Kopfnormalform werden in Haskell nicht weiter ausgewertet, es sei denn, dies ist für einen Musterabgleich erforderlich und möglich.

Im nächsten Abschnitt wird das Konzept der Funktionsauswertung für eine reduzierte Form der Sprache formalisiert.

5.1.3 Formale Beschreibung der Semantik von Haskell

Giesl et al. betrachten eine reduzierte Variante von Haskell, die alle bisher beschriebenen Konzepte umfasst. Konstrukte wie `type`, `newtype` oder Module werden nicht berücksichtigt. Nach [Swi05] lassen sich alle Haskell-Programme in die in [GRSK+11] verwendete, reduzierte Form übersetzen. Das Konzept der Typklassen wird in der vorliegenden Arbeit nicht betrachtet. Die durch Typklassen ermöglichte Polymorphie lässt sich bereits zur Übersetzungszeit auflösen. Es geht daher kein Verlust der Ausdrucksstärke mit der Reduzierung einher.

Sämtliche Konstrukt zur Fallunterscheidungen werden in Haskell gemäß [PJ03] in geschachtelte `case`-Ausdrücke mit jeweils zwei Zweigen überführt. In [GRSK+11] werden hingegen alle Fallunterscheidungen durch Musterabgleiche in den Funktionsdefinitionsgleichungen realisiert.

Besondere Beachtung verdienen Lambda-Ausdrücke, mit denen in Haskell anonyme Funktionen definiert werden können. Sie werden in [GRSK+11] mit dem folgenden Verfahren ersetzt. Sei $\lambda\ t_1 \ldots t_n \rightarrow t$ ein Lambda-Ausdruck mit den freien Variablen $x_1, \ldots, x_m$. Es wird eine neue Funktion f mit der Gleichung

$$f\ x_1 \ldots x_m\ t_1 \ldots t_m = t$$

definiert und der Ausdruck $\lambda\ t_1 \ldots t_n \rightarrow t$ durch $f\ x_1 \ldots x_m$ ersetzt.

Da Funktionen vollwertige Mitglieder des Typsystems sind, können Variablen in Haskell auch für Funktionen stehen. Folglich müssen Substitutionen für solche Va-

riablen beliebige Funktionen einführen können. Damit dies ohne Lambda-Ausdrücke möglich bleibt, wird erlaubt, dass Substitutionen neue Definitionsgleichungen einführen.

Im Folgenden wird gemäß [GRSK+11] die Auswertung von Ausdrücken in der so reduzierten Sprache beschrieben. Dazu dient eine Auswertungsrelation. Giesl et al. bezeichnen diese als operationelle Semantik von Haskell.

Definition 5.1. *Die* Positionen *eines Ausdrucks t sind definiert als* $\text{Pos}(t) = \{\varepsilon\}$, *falls $t \equiv f$ für ein Funktionssymbol f oder $t \equiv x$ für eine Variable x ist. Für $t \equiv t_1\ t_2$ ist* $\text{Pos}(t) = \{\varepsilon\} \cup \{1\pi | \pi \in \text{Pos}(t_1)\} \cup \{2\pi | \pi \in \text{Pos}(t_2)\}$. *Die Notation $t|_p$ für einen Ausdruck t und eine Position p wird verwendet, um den Ausdruck an der Position p zu bezeichnen. Es gilt $t|_\varepsilon = t$ und $(t_1\ t_2)|_{i\pi} = t_i|_\pi$ für $i \in \{1, 2\}$. Ein Ausdruck q heißt* Teilausdruck *von t genau dann, wenn es eine Position p gibt, sodass $t|_p = q$. Der* Kopf *eines Ausdrucks t wird mit* $\text{head}(t)$ *bezeichnet und ist definiert als* $\text{head}(t) = t_{1^n}$ *mit n als der größten Zahl, sodass* $1^n \in \text{Pos}(t)$.

Haskell gilt als sehr ausdrucksstarke Sprache. Dies kann neben den Funktionen höherer Ordnung und dem flexiblen Typsystem vor allem mit der nicht-strikten Auswertungsstrategie begründet werden. Für die Terminierungsanalyse stellt diese Strategie allerdings eine Herausforderung dar. Ohne Berücksichtigung der verzögerten Auswertung, müssten alle Auswertungsfolgen betrachtet werden. Es ist jedoch keineswegs gegeben, dass ein terminierendes Haskell-Programm unter anderen als der nicht-strikten Strategie ebenfalls terminiert. Beispiel 5.2 illustriert dies.

Beispiel 5.2. *Die rekursive Funktion* `nats` *erzeugt die unendliche Liste der natürlichen Zahlen und ist definiert als*

```
nats = 0 : map (+1) nats.
```

Die Funktion `map` *wendet die als erstes Argument übergebene Funktion elementweise auf die als zweites Argument übergebene Liste an. Die Funktion* `take n` *berechnet das Präfix der Länge n einer beliebigen Liste. Für* `take 2 nats` *ergibt sich unter der nicht-strikten Auswertungsstrategie die Liste mit den Zahlen 0 und 1, da der*

Ausdruck wie folgt evaluiert wird:

$$
\begin{aligned}
& \texttt{take 2 nats} \\
\to\ & \texttt{take 2 (0 : map (+1) nats)} \\
\to\ & \texttt{0 : take 1 (map (+1) nats)} \\
\to\ & \texttt{0 : take 1 (map (+1) (0 : map (+1) nats))} \\
\to\ & \texttt{0 : take 1 (1 : (map (+1) (map (+1) nats)))} \\
\to\ & \texttt{0 : 1 : take 0 ((map (+1) (map (+1) nats)))} \\
\to\ & \texttt{0 : 1 : []}
\end{aligned}
$$

Die Auswertung terminiert unter der nicht-strikten Strategie, da `take 0 xs` *für alle Ausdrücke* `xs` *zur leeren Liste* [] *evaluiert. Das Listenargument* `xs` *wird nicht weiter ausgewertet.*

Die strikte Auswertungsstrategie hingegen evaluiert Argumente vollständig. Dies führt für denselben Ausdruck zu der folgenden, unendlichen Auswertungsfolge:

$$
\begin{aligned}
& \texttt{take 2 nats} \\
\to\ & \texttt{take 2 (0 : map (+1) nats)} \\
\to\ & \texttt{take 2 (0 : map (+1) (0 : map (+1) nats))} \\
\to\ & \texttt{take 2 (0 : map (+1) (0 : map (+1) (0 : map (+1) nats)))} \\
\to\ & \dots
\end{aligned}
$$

Diese Folge wird unendlich fortgesetzt. Der Ausdruck `take 2 nats` *ist daher unter der strikten Auswertungsstrategie nicht terminierend.*

Um die Auswertung von Ausdrücken korrekt zu beschreiben, muss bestimmt werden, an welcher Position in einem Ausdruck als nächstes ein Ersetzungsschritt stattfindet. Im Allgemeinen ist dies die äußerste Position ε. Ausnahmen ergeben sich, wenn entweder für einen Musterabgleich zuerst ein Teilausdruck weiter ausgewertet werden muss oder für Ausdrücke in der folgenden Form. Sei f eine Funktion mit Arität n und der auszuwertende Ausdruck $f\ t_1 \dots t_n\ t_{n+1} \dots t_m$. Der nächste Auswertungsschritt kann nicht an ε erfolgen, da zunächst der Teilausdruck $f\ t_1 \dots t_n$ evaluiert werden muss. Dieser befindet sich an der Position 1^{m-n}.

Die folgenden Definitionen beziehen sich stets auf ein festes Haskellprogramm. Die erste Definition beschreibt, unter welchen Bedingungen eine gegebene Definitionsgleichung bei einem Musterabgleich mit einem Term in Frage kommt.

Definition 5.3. *Eine Gleichung $l = r$ heißt* zulässig *für einen Ausdruck t, wenn* $\mathrm{head}(l) = \mathrm{head}(t)$ *und es entweder eine Variablensubstitution σ gibt, sodass $l^\sigma \equiv t$ oder es eine Position π gibt, sodass* $\mathrm{head}(l|_\pi)$ *ein Konstruktor, aber* $\mathrm{head}(t|_\pi)$ *nicht in Kopfnormalform ist. Gibt es mehrere Positionen, die den Kriterien an π genügen, so sei π die kleinste bezüglich der lexikographischen Ordnung auf dem freien Monoid $\{1,2\}^*$. Die* Auswertungsposition bezüglich *l wird notiert mit $e_l(t)$ und ist im ersten Fall definiert als $e_l(t) = \varepsilon$ und $e_l(t) = \pi$ im zweiten Fall.*

Haskell berücksichtigt die Reihenfolge der Gleichungen einer Funktionsdefinition. Sei f ein Funktionssymbol und die zugehörige Funktion definiert über mehrere Gleichungen. Für einen Ausdruck t mit $\mathrm{head}(t) = f$ werden Teilausdrücke von t evaluiert, wenn für die erste zulässige Gleichung $l = r$ die Auswertungsposition bezüglich l nicht ε ist. Dies geschieht auch, wenn eine spätere Gleichung l' zulässig wäre und $e_{l'}(t) = \varepsilon$ gilt.

Definition 5.4. *Die* Auswertungsposition *für jeden Ausdruck t wird definiert als*

$$e(t) = \begin{cases} 1^{m-n}\pi, & \textit{falls } t = f\ t_1 \dots t_n\ t_{n+1} \dots t_m, m > \mathrm{ari}(f) \textit{ und } \pi = e(f\ t_1 \dots t_n) \\ e_l(t)\pi, & \textit{falls } t = f\ t_1 \dots t_n, n = \mathrm{ari}(f), \textit{ die erste zulässige} \\ & \quad \textit{Gleichung } l = r \textit{ ist, } e_l(t) \neq \varepsilon \textit{ und } \pi = e(t|_{e_l(t)}) \\ \varepsilon, & \textit{sonst.} \end{cases}$$

Die obige Definition gibt die Position wieder, an der der nächste Auswertungsschritt nach Haskells nicht-strikter Strategie stattfindet. Mithilfe dieser Position kann die Auswertungsrelation beschrieben werden. Giesl et al. definieren diese wie folgt.

Definition 5.5. *Die* Auswertungsrelation *wird mit $\to_H$ notiert. Es gilt $s \to_H t$ genau dann, wenn eine der folgenden Bedingungen gilt.*

1. *Die erste zulässige Gleichung ist $l = r$, es existiert eine Substitution σ mit $l^\sigma \equiv s|_{e(s)}$ und der Ausdruck t ergibt sich durch das Ersetzen von l^σ in s an der Stelle $e(s)$ durch r^σ.*

2. *Der Ausdruck s ist in der Form $s \equiv C\ s_1 \dots s_n$ und für ein i mit $1 \leq i < n$ gilt $s_i \to_H t_i$ und $t = C\ s_1 \dots s_{i-1}\ t_i\ s_{i+1} \dots s_n$.*

Die Definitionen 5.3, 5.4 und 5.5 stammen aus [GSSKT06]. In beiden Arbeiten wird nicht klar, dass die Auswertungsrelation nicht Haskells Auswertungsstrategie beschreibt. Die Unterschiede sollen im Folgenden verdeutlicht werden.

Für Ausdrücke $s = C\ s_1 \dots s_n$ sieht Haskells Auswertungsstrategie keinen weiteren Schritt vor, da es sich um einen Ausdruck in Kopfnormalform handelt. Nach Definition 5.5 Klausel 2 ist s allerdings nur dann in Normalform bezüglich $\to_H$, wenn $s_1, \dots, s_n$ ebenfalls in Normalform bezüglich $\to_H$ sind. Giesl et al. rechtfertigen die Definition mit der Darstellungsfunktion `show`, die in gängigen Interpretern wie Hugs [JP99] und GHCi verwendet wird, um das Resultat einer Auswertung darzustellen. Diese Interpreter werten bei Eingabe eines Ausdrucks t allerdings `show` t aus. Da aber `show` $C\ s_1\ \dots\ s_n$ und $C\ s_1\ \dots\ s_n$ eindeutig verschiedene Ausdrücke sind, ist diese Argumentation für den Autor der vorliegenden Arbeit nicht nachvollziehbar. Dennoch wird in der vorliegenden Arbeit Definition 5.5 unverändert übernommen. Der Grund hierfür ist der Terminierungsbegriff, der für Haskell anders definiert werden muss als bisher. Nach der Definition der sogenannten H-Terminierung wird ein Beispiel gegeben, warum auf die zweite Klausel in Definition 5.5 nicht verzichtet werden kann. Allerdings wird in der vorliegenden Arbeit aufgrund dieser Differenz darauf verzichtet, die Auswertungsrelation $\to_H$ als operationelle Semantik von Haskell zu bezeichnen, wie es in den Arbeiten [GSSKT06] und [GRSK$^+$11] der Fall ist.

Ein weiterer Unterschied zwischen der Auswertungsrelation und der Auswertungsstrategie, besteht darin, dass eine Strategie den nächsten Auswertungsschritt eindeutig beschreiben muss. Die Regel für derartige Ausdrücke beschreibt allerdings diverse mögliche Auswertungsschritte, welche gemäß der Haskell-Semantik nur vorgenommen werden, wenn der Ausdruck s ein Teilausdruck eines auszuwertenden Ausdrucks ist. Der Unterschied ergibt sich daraus, dass hier die Auswertungsrelation und nicht Haskells Auswertungsstrategie formalisiert wird. Beide induzieren unterschiedliche abstrakte Reduktionssysteme.

Während ein Termersetzungssystem $\mathcal{R}$ als terminierend bezeichnet wird, wenn alle Terme stark normalisierend bezüglich $\to_\mathcal{R}$ sind, erweist sich ein ähnlicher Begriff auf Basis von $\to_H$ als nicht ausreichend. Für Haskell führt [GRSK$^+$11] daher das Konzept der H-Terminierung ein. Dieser ist in zwei Punkten erweitert gegenüber der reinen Normalformeigenschaft von $\to_H$. Mit Haskell können in gewissem Sinne unendliche Datenstrukturen verarbeitet werden. Ausdrücke, deren Auswertungsresultat jedoch eine solche unendlichen Datenstruktur ist, sollten nicht als terminierend betrachtet werden. Der andere Unterschied ergibt sich aus Ausdrücken, die zu Funktionswerten evaluieren. Ausdrücke, die einem Funktionswert entsprechen, sollten nur dann als terminierend gelten, wenn die resultierende Funktion terminiert. Die folgende Definition nach [GRSK$^+$11] formalisiert diese Ideen.

Definition 5.6. *Die Menge der* H-terminierenden Ausdrücke ohne Variablen *ist die kleinste Menge von Ausdrücken t ohne Variablen, für die die folgenden Bedingungen gelten.*

1.) Mit t beginnt keine unendliche Auswertungsfolge $t \to_H \ldots$.

2.) Falls $t \twoheadrightarrow_H (f\ t_1 \ldots t_n)$ mit $n < \mathrm{ari}(f)$ und der Ausdruck t' H-terminiert, dann ist auch $f\ t_1 \ldots t_n\ t'$ H-terminierend.

3.) Falls $t \twoheadrightarrow_H (C\ t_1 \ldots t_n)$, so sind auch alle $t_1, \ldots, t_n$ H-terminierend.

Ein Ausdruck t ist ein H-terminierender Ausdruck *genau dann, wenn für alle Substitution σ mit H-terminierenden Ausdrücken ohne Variablen auch t^σ H-terminierend ist.*

Die dritte Klausel erscheint auf den ersten Blick überflüssig, da $\to_H$ bereits Evaluationen unterhalb eines Konstruktorsymbols zulässt. Das folgende Beispiel, für das keine Entsprechung in den Arbeiten von Giesl et al. existiert, demonstriert, dass die Klausel nicht überflüssig ist. Sei f eine nicht H-terminierende Funktion vom Typ $\alpha \to \alpha$ und C ein Datenkonstruktor für den Typ τ, der ein Argument vom Typ $(\alpha \to \alpha)$ erwartet. Damit ist $C\ f$ nach Definition 5.6 nicht H-terminierend. Allerdings ist $C\ f$ bezüglich $\to_H$ in Normalform, da C ein Konstruktor ist und f selbst in Normalform ist. Ohne die dritte Klausel wäre $C\ f$ daher fälschlich H-terminierend.

Wie angedeutet soll nun erläutert werden, warum $\to_H$ Evaluationen unterhalb der obersten Position für Ausdrücke in Kopfnormalform zulässt. Eine Erklärung hierfür findet sich weder in [GSSKT06] noch in [GRSK⁺11]. Sei $\to'_H$ definiert wie $\to_H$ jedoch ohne die zweite Klausel. Es gilt $\to'_H \subset \to_H$. Ferner sei H'-Terminierung analog zu H-Terminierung definiert, indem $\to_H$ in allen drei Klauseln durch $\to'_H$ ersetzt wird..

Es wird gezeigt, dass der Begriff der H'-Terminierung nicht wohldefiniert ist. Sei die Funktion `infinity` und der Datentyp `Nat` definiert wie im Quelltextauszug 5.1. Der Ausdruck `infinity` Z ist nach intuitivem Verständnis nicht terminierend. Da es keine Normalform bezüglich $\to_H$ für `infinity` Z gibt, ist `infinity` Z nicht H-terminierend nach Klausel 1 aus Definition 5.6. Hingegen ist für den Ausdruck `infinity` Z nicht definiert, ob er H'-terminiert. Da

$$\texttt{infinity}\ Z \to'_H \texttt{Succ}\ (\texttt{infinity}\ Z) \tag{5.1}$$

gilt und `Succ` (`infinity` Z) bezüglich $\to'_H$ in Normalform ist, ist die erste Klausel erfüllt. Die Voraussetzung der zweiten Klausel in Definition 5.6 ist nicht erfüllt, da

Quelltextauszug 5.1: Die Funktion `infinity`

```
data Nat = Zero | Succ Nat

infinity :: Nat -> Nat
infinity x = Succ (infinity x)
```

`infinity` Z eine Anwendung einer einstelligen Funktion auf ein Argument ist. Die Klausel selbst ist damit erfüllt. Es bleibt die dritte Klausel in 5.6, deren Voraussetzung erfüllt ist, da Beziehung 5.1 gilt. Es ergibt sich die Tautologie, dass `infinity` Z H'-terminiert, genau dann wenn `infinity` Z H'-terminiert. Es folgt, dass die zweite Klausel in Definition 5.5 nicht verzichtbar ist für den intendierten Terminierungsbegriff.

Im Folgenden werden nur Substitutionen betrachtet, die Variablen durch Ausdrücke ohne Variablen und in Normalform ersetzen. Derartige Substitutionen heißen Normalbasissubstitutionen. Dies stellt nach [GRSK$^+$11] keine Einschränkung dar.

5.2 Terminierungsgraphen für Haskell-Programme

Die Definitionsgleichungen eines Haskell-Programms können zum Zwecke der Terminierungsanalyse nicht einfach als Regeln eines Termersetzungssystems betrachtet werden, da dies die Auswertungsstrategie unberücksichtigt lassen würde. Stattdessen verwenden Giesl et al. in ihrem Verfahren einen sogenannten Terminierungsgraphen, aus dem sich Abhängigkeitspaarprobleme bestimmen lassen. In diesem Abschnitt soll erläutert werden, wie der Terminierungsgraph nach [GRSK$^+$11] erzeugt wird und welche Eigenschaften er hat.

Für einen initialen Ausdruck t, für den geprüft werden soll, ob er H-terminiert oder nicht, wird der Terminierungsgraph erzeugt, indem der Ausdruck gemäß der Auswertungsrelation symbolisch ausgewertet wird. Daraus entsteht zunächst eine baumartige Struktur. In späteren Schritten, den Instanziierungsschritten, werden Knoten eingefügt, die auch Rückkanten zu bestehenden Knoten haben können. Dadurch wird sichergestellt, dass es sich um eine endliche Struktur handelt. Aus dem Graphen werden dann die Abhängigkeitspaarprobleme erzeugt. Durch die Konstruktion mit Rücksicht auf Haskells Auswertungsstrategie tragen die Abhängigkeitspaarprobleme bereits die notwendigen Informationen, sodass das Dependency Pair Framework nicht an Haskells nicht-strikte Evaluation angepasst werden muss.

Wenn ein Ausdruck nicht H-terminiert, so hat der zugehörige Knoten mindes-

tens einen Nachfolger, der mit einem ebenfalls nicht H-terminierenden Ausdruck beschriftet ist.

Es wird zunächst der baumartige Teil des Graphen beschrieben. Die Konstruktion beginnt mit einem Knoten, der mit dem initialen Ausdruck t beschriftet ist. Da alle Knoten mit Ausdrücken beschriftet sind, werden die Knoten mit den Ausdrücken identifiziert. Es folgen darauf Expansionsschritte, in denen Knoten ohne Nachfolger Kinder erhalten, die wiederum mit Ausdrücken benannt sind. Die Kanten werden mit Substitutionen markiert. Bei einigen Regeln ist keine Substitution erforderlich. Es wird vereinbart, dass unbeschriftet Kanten der Identitätssubstitution entsprechen.

Abhängig von dem Ausdruck des zu erweiternden Knoten können verschiedene Expansionsregeln angewendet werden. Abgesehen von der Instanziierungsregel kommt stets nur eine Regel in Frage. Die erste Regel ist die Auswertungsregel.

Definition 5.7. *Die* Auswertungsregel *ist anwendbar auf Knoten mit der Beschriftung* $s \equiv f\ t_1 \ldots t_n$ *, wenn* f *ein Funktionssymbol,* $\mathrm{ari}(f) \leq n$ *und* $s \to_H t$ *für einen Ausdruck* t *gilt. Es wird ein Knoten mit der Beschriftung* t *und eine Kante zwischen* s *und* t *hinzugefügt. Es gilt* $\mathrm{Ausw}_G(s)$ *genau dann, wenn* s *in einem Graphen* G *durch die Auswertungsregel expandiert worden ist. Knoten* s*, für die* $\mathrm{Ausw}_G(S)$ *gilt, heißen* Auswertungsknoten*. Die erzeugte Kante erhält keine Beschriftung.*

Die nächste Regel ist die Fallunterscheidungsregel. Ist der zu expandierende Ausdruck s nicht auswertbar, weil er Variablen enthält, so wird für die am weitesten links stehende Variable eine Fallunterscheidung durchgeführt. Anhand des Typs der Variablen werden die infrage kommenden Datenkonstruktoren bestimmt. Für jeden Konstruktor werden dann neue Knoten erzeugt. Ist ein Konstruktor parametrisiert, so werden frische Variablen als Argumente verwendet. Es ergeben sich Substitutionen, die die fragliche Variablen auf die Konstruktoren abbilden. Die neuen Knoten werden mit dem Ergebnis der auf t angewandten Substitution beschriftet.

Definition 5.8. *Die* Fallunterscheidungsregel *ist anwendbar auf Knoten mit einer Beschriftung* $t \equiv f\ t_1 \ldots t_n$*, wenn* f *ein Funktionssymbol,* $\mathrm{ari}(f) \leq n$ *und* $t|_{e(t)}$ *eine Variable* x *vom Typ* τ *ist, für den es* k *Datenkonstruktoren* $C_1, \ldots, C_k$ *gibt. Sei* n_i *jeweils die Arität des Konstruktors* C_i*. Es werden* k *Nachfolgerknoten für* t *mit den Beschriftungen* t^{σ_i} *erzeugt, wobei* $\sigma_i = [x := C_i\ x_1 \ldots x_{n_i}]$ *für frische Variablen* x_j*. Die Kanten werden mit den entsprechenden Substitutionen* σ_i *beschriftet. Es gilt* $\mathrm{Fall}_G(t)$ *genau dann, wenn* t *in einem Graph* G *durch die Fallunterscheidungsregel expandiert worden ist. Knoten* t*, für die* $\mathrm{Fall}_G(S)$ *gilt, heißen* Fallunterscheidungsknoten.

Da nur Variablen ersetzt werden können, für die Datenkonstruktoren existieren, können Ausdrücke mit Funktionsvariablen nicht mit dieser Regel expandiert werden.

Die beiden ersten Regeln werden nur angewendet, wenn genügend Argumente vorhanden sind. Ansonsten greift die Variablenergänzungsregel.

Definition 5.9. *Die* Variablenergänzungsregel *oder kurz* Ergänzungsregel *ist anwendbar auf Knoten mit einer Beschriftung $t \equiv f\ t_1 \ldots t_n$, wenn f ein Funktionssymbol ist und $n < \text{ari}(f)$. Es wird ein Nachfolgerknoten erzeugt, der mit $f\ t_1 \ldots t_n\ x$ beschriftet wird, wobei x eine bisher nicht verwendete Variable ist. Die Kante wird nicht markiert. Es gilt* $\text{VarErw}_G(t)$ *genau dann, wenn t in einem Graph G durch die Ergänzungsregel expandiert worden ist. Knoten t, für die* $\text{VarErw}_G(t)$ *gilt, heißen* Variablenergänzungsknoten*.*

Abgesehen von der Instanziierungsregel bleibt nur noch die Parameterzerlegungsregel. Diese ist auf Konstruktoren anwendbar.

Definition 5.10. *Die* Parameterzerlegungsregel *ist anwendbar auf Knoten mit einer Beschriftung der Form $t \equiv c\ t_1 \ldots t_n$ wobei c ein Konstruktor oder eine Variable ist. Für jedes Argument t_i von c in dem Ausdruck t wird ein Knoten mit der Beschriftung t_i angelegt. Die Kanten werden nicht markiert und es gilt* $\text{ParZerl}_G(t)$ *genau dann, wenn t in einem Graph G durch die Parameterzerlegungsregel expandiert worden ist. Knoten t, für die* $\text{ParZerl}_G(t)$ *gilt, heißen* Parameterzerlegungsknoten*.*

Die von Giesl et al. definierte Auswertungsrelation $\rightarrow_H$ sieht für einen Ausdruck der Form $C\ t_1 \ldots t_n$ verschiedene mögliche Nachfolger vor. Daher scheint es, dass alternativ auch mit der Auswertungsregel fortgefahren werden könnte. Diese ist jedoch nur für Ausdrücke definiert, die mit einem Funktionssymbol beginnen.

Ist der durch iteratives Anwenden der Regeln konstruierte Baum endlich, so ist der initiale Ausdruck t H-terminierend[1]. Allerdings ist selbst für H-terminierende Ausdrücke nicht sichergestellt, dass ein so konstruierter Baum endlich ist. Das folgende Beispiel demonstriert diese Situation und liefert gleichzeitig die Motivation für die Instanziierungsregel. Gegeben sei das Programm aus Quelltext 5.2. Der initiale Ausdruck sei `take (Succ m)`. Die Funktion `take` berechnet das Präfix einer Liste natürlicher Zahlen mit der als erstem Argument übergebenen Länge. Überschreitet diese Länge die der Liste, so wird die Liste zurückgegeben. Da der Typ `Nat` nur endliche Längen darstellt, muss die Funktion terminieren. Die Konstruktion des Baums

[1] Auf einen Beweis wird an dieser Stelle verzichtet, er ergibt sich unmittelbar aus den weiteren Überlegungen.

Quelltextauszug 5.2: Die Funktion take

```
data Nat = Zero | Succ Nat
data NatList = Nil | Cons Nat NatList

take :: Nat -> NatList -> NatList
take m          Nil          = Nil
take Zero       xs           = Nil
take (Succ m) (Cons x xs) = Cons x (take m xs)
```

entsprechend den Expansionsregeln ergibt allerdings eine unendliche Baumstruktur, wie die folgende Überlegung zeigt.

Zunächst wird die Variablenergänzungsregel angewendet und es ergibt sich `take (Succ m) xs`. Da nicht festgestellt werden kann, welche Regel angewendet werden kann, greift die Fallunterscheidungsregel für `xs`. Der Typ von `xs` ist `NatList`, für den die beiden Konstruktoren `Nil` und `Cons` existieren. Es werden daher mit den beiden Substitutionen $\sigma_1 = [\texttt{xs} := \texttt{Nil}]$ und $\sigma_2 = [\texttt{xs} := \texttt{Cons y ys}]$ zwei Nachfolgerknoten erzeugt. Für beide wird die Auswertungsregel angewendet. Dadurch erhält `take (Succ m) Nil Nil` als Nachfolger, welcher ein Blatt darstellt. Hingegen ergibt sich für `take (Succ m) (Cons y ys)` der Nachfolger `Cons y (take m ys)`. Mit der Parameterzerlegungsregel ergeben sich dafür zwei Nachfolger: `y` und `take m ys`. Es erfolgt wieder eine Fallunterscheidung, diesmal für `m` vom Typ `Nat`. Es sind daher `Zero` und `Succ n` möglich. Erstere kann nach Definition von `take` ausgewertet werden und es ergibt sich das Blatt `Nil`. Im zweiten Fall ergibt sich `take (Succ n) ys`. Bis auf die Namen der Variablen unterscheidet sich dieser Ausdruck nicht von dem bereits betrachteten `take (Succ m) xs`. Es folgt, das alle Schritte unendlich oft wiederholt werden können. Die Instanziierungsregel erlaubt dennoch eine endliche Darstellung.

Definition 5.11. *Die* Instanziierungsregel *ist auf Knoten mit der Beschriftung $t \equiv f\ t_1 \ldots t_n$ anwendbar, wenn f ein Funktionssymbol,* $\mathrm{ari}(f) \leq n$ *und es einen Ausdruck s gibt, sodass für eine Substitution σ gilt $s^\sigma = t$ und s entweder ein Fallunterscheidungs- oder Auswertungsknoten ist. Handelt es sich bei s um einen Fallunterscheidungsknoten, so muss die zusätzliche Bedingung gelten, dass auf allen von s ausgehenden Pfaden entweder ein Blatt erreicht wird, dass mit einem Fehlerausdruck beschriftet ist, oder ein Auswertungsknoten erreicht wird, nachdem nur weitere Fallunterscheidungsknoten besucht worden sind. Durch die Regel kann ein neuer Knoten mit der Beschriftung $s \equiv x\ y$ für frische Variablen x und y erzeugt und verwendet werden. Für alle Variablen $x \in$* $\mathrm{Var}(s)$ *werden neue Nachfolgerknoten mit den Beschriftungen $\sigma(x)$ erzeugt. Von t nach s wird eine* Instanziierungskante

erzeugt, die mit σ beschriftet wird. Alle andere Kanten bleiben unbeschriftet. Es gilt $\mathrm{Ins}_G(t)$ *genau dann, wenn t in einem Graph G durch die Ergänzungsregel expandiert worden ist. Knoten t, für die* $\mathrm{Ins}_G(t)$ *gilt, heißen* Instanziierungsknoten. *Der Knoten s heißt* Instanziierungsnachfolger *von t.*

Die zusätzliche Bedingung für Fallunterscheidungsknoten stellt sicher, dass auf allen Kreisen mit Instanziierungskanten auch mindestens ein Auswertungsknoten besucht wird. Intuitiv ist dies notwendig, damit auch für Instanziierungskanten sichergestellt ist, dass die simulierte Auswertung fortschreitet.

Giesl et al. erklären in [GRSK+11] die Instanziierungsregel für einen neuen Knoten $s \equiv x\ y$ mit dem Beispiel `tma (Succ n) = tma n n`. Dabei wird auf der rechten Seite der Definitionsgleichung `tma` auf „zu viele Argumente" angewendet. Der Typ der Funktion ist `Nat` $\rightarrow$ `a` und in einem Hindley-Milner-Typsystem nicht inferierbar. Die vermutlich am stärksten verbreitete Haskell-Implementierung GHC verwendet laut [LJ03] kein reines Hindley-Milner-Typsystem und kann die Funktion übersetzen, wenn der Typ explizit annotiert wird. Für Programme, die Funktionen wie `tma` beinhalten, gelten daher zum Beispiel die Theoreme aus [Wad89] nicht. Darüber hinaus ist die Funktion nicht vollständig definiert, denn es fehlt eine Funktionsgleichung für `tma Z`. Jede Auswertung von `tma` führt allerdings zu diesem Fall und bricht daher mit einem Fehler ab. Es scheint unmöglich, eine Gleichung für den Fall `tma Z` anzugeben, sodass `tma` wenigstens für einen Wert terminiert ohne einen Fehler zu erzeugen.

Ein einfacheres Beispiel, dass in [GRSK+11] später selbst erscheint, ist die Funktion `id`. Ihr Typ ist $a \rightarrow a$. Offenbar gilt $\mathrm{ari}(\texttt{id}) = 1$. Die Anwendung von `id` auf eine ebenfalls einstellige Funktion `f` und ein weiteres Argument ergibt den validen Ausdruck `id f x`. Auch hier wird `id` auf zu viele Argumente angewendet, jedoch bleibt der Ausdruck in einem Hindley-Milner-Typsystem inferierbar und terminiert für geeignete Argumente ohne Fehler.

In dem in der vorliegenden Arbeit zur Motivation der Instanziierungsregel herangezogenen Beispiel `take (Succ m)` kann nun eine Instanziierungskante gezogen werden von `take (Succ n) ys` nach `take (Succ m) xs`. Es gilt dabei $\sigma = [\texttt{m}, \texttt{xs} := \texttt{n}, \texttt{ys}]$. Hinzu kommen zwei Nachfolgerknoten $\sigma(\texttt{m})$ und $\sigma(\texttt{xs})$. Beide sind mit Variablen beschriftet und kommen für weitere Expansionen nicht in Frage. Es gibt daher keine Knoten ohne Nachfolger, auf die eine weitere Regeln anwendbar wäre.

Mit diesen Regeln definieren Giesl et al. den Terminierungsgraphen wie folgt.

Definition 5.12. *Seien G und G′ Graphen. Es gilt $G \Rrightarrow G'$ genau dann, wenn G′ aus G durch Anwendung einer der Expansionsregeln hervorgeht. Sei t ein Ausdruck*

und G_t der Graph ohne Kanten und exakt einem Knoten beschriftet mit t. Ein Terminierungsgraph *für einen Ausdruck t ist ein Graph G, sodass $G_t \Rrightarrow^* G$ und G bezüglich $\Rrightarrow$ in Normalform ist.*

Das folgende Lemma besagt, dass die Konstruktion für alle Haskell-Programme endlich ist.

Lemma 5.13. *Sei G_t und $\Rrightarrow$ definiert wie in Definition 5.12. Für alle Ausdrücke t existiert eine Normalform für G_t bezüglich $\Rrightarrow$.*

Giesl et al. zeigen dies in [GRSK+11] auf folgende Weise.

Beweis. Unabhängig von dem initialen Ausdruck t wird zunächst ein Graph G in Normalform konstruiert. Dazu wird mit einem Graphen begonnen, der für jede Funktion f mit Arität $\text{ari}(f) = n$ einen Knoten mit der Beschriftung $f\ x_1 \dots x_n$ enthält. Die Variablen x_i sind paarweise verschieden und zwischen den Knoten existieren keine Kanten. Nach der Fallunterscheidungsregel werden die Variablen sukzessive ersetzt bis die Auswertungsregeln angewendet werden können. Es gibt damit für jede rechte Seite einer Funktionsdefinitionsgleichung einen Knoten der folgenden drei Arten.

Die erste Art sind Knoten, die nur aus einer Variablen oder einem parameterlosen Konstruktor bestehen. Auf diese sind keine Regeln mehr anwendbar. Hat der Graph nur noch Blätter dieser Art, so ist er in Normalform. Die zweite Art sind Knoten, deren Beschriftung mit einer Variablen oder einem Konstruktorsymbol beginnt. In diesen Fällen ist die Parameterzerlegungsregel anwendbar. Die entstehenden Nachfolgerknoten gehören ebenfalls jeweils zu einer der drei diskutierten Arten.

Die letzte Art sind Knoten, deren Beschriftung mit einem Funktionssymbol beginnt. Sei diese Funktion g mit Arität n. Wird g auf n Argumente $t_1, \dots, t_n$ angewendet, gibt es einen Knoten $s \equiv g\ x_1 \dots x_n$, sodass die Instanziierungsregel mit der Substitution $\sigma = [x_1, \dots, x_n := t_1, \dots, t_n]$ angewendet werden kann, da $s^\sigma = g\ t_1 \dots t_n$ gilt. Wird g auf zu wenige Argumente angewendet, also $g\ t_1 \dots t_m$ mit $m < n$, kann die Variablenergänzungsregel wiederholt angewendet werden und um Variablen $y_{m+1}, \dots, y_n$ ergänzt werden. Es entsteht ein Knoten mit der Beschriftung $g\ t_1 \dots t_m\ y_{m+1} \dots y_n$, auf den wieder die Instanziierungsregel angewendet werden kann. Der so entstandene Graph G ist in Normalform.

Kommt der Knoten t noch nicht in G vor, so wird er hinzugenommen. Er gehört ebenfalls zu einer der drei Arten. Daher kann nach demselben Verfahren t expandiert werden, bis nur noch Blätter oder Instanziierungskanten in die bestehenden Knoten

G vorkommen. Sämtliche Knoten, die nicht von t aus erreicht werden können, werden gestrichen. Der resultierende Graph ist damit in Normalform und es folgt die Behauptung. □

Um aus Terminierungsgraphen tatsächlich Rückschlüsse auf die H-Terminierung des zugrunde liegenden Programms zu ziehen, sind verschiedene Methoden denkbar. Ein Ansatz wäre die Erzeugung eines Termersetzungssystems. Laut [GRSK+11] ist dieser Ansatz jedoch nicht besonders gut geeignet. Im folgenden Abschnitt wird die in [GSSKT06] eingeführte Methode zur Erzeugung von Abhängigkeitspaarproblemen aus dem Terminierungsgraphen vorgestellt.

5.3 Erzeugung eines $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Problem aus dem Terminierungsgraphen

Das System APROVE, welches ursprünglich zur automatisierten Terminierungsanalyse von Termersetzungssystemen entwickelt wurde, erzeugt für ein Haskell-Programm zunächst einen Terminierungsgraphen und wandelt diesen in geeignete Abhängigkeitspaarprobleme um. In [GSSKT06] wurde das eingesetzte Verfahren zuerst beschrieben und wird in diesem Abschnitt rekapituliert. Die grundlegende Idee besteht darin, aus jeder starken Zusammenhangskomponente des Terminierungsgraphen ein Abhängigkeitspaarproblem $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$ zu erzeugen. Diese Probleme werden so konstruiert, dass die Abwesenheit von unendlichen $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Ketten die H-Terminierung aller Knoten implizieren.

Dieses Vorgehen erinnert an den Abhängigkeitsgraphen aus Kapitel 4. Die Bestimmung der starken Zusammenhangskomponenten bleibt allerdings die einzige Ähnlichkeit zwischen den beiden Verfahren. Die Knoten des Abhängigkeitsgraphen sind die Abhängigkeitspaare eines gegebenen Termersetzungssystems $\mathcal{R}$. Für einen Terminierungsgraphen ist allerdings weder offensichtlich, welches Termersetzungssystem $\mathcal{R}$, noch welche Abhängigkeitspaare $\mathcal{P}$ konstruiert werden sollten, um die H-Terminierung nachzuweisen. Giesl et al. geben hierfür in [GSSKT06] eine Methode an. In dieser wird $\mathcal{Q} = \emptyset$ gesetzt. Das Verfahren nutzt bestimmte Pfade in den jeweiligen starken Zusammenhangskomponenten des Terminierungsgraphen, um sowohl Regeln als auch Paare zu definieren. Giesl et al. definieren zunächst den Paarpfad wie folgt.

Definition 5.14. *Sei G' eine starke Zusammenhangskomponente des Terminierungsgraphen. Ein Pfad in G' von s nach t heißt* Paarpfad *genau dann, wenn s*

eine eingehende Instanziierungskante besitzt, t eine ausgehende Instanziierungskante besitzt und keine der Kanten des Pfads eine Instanziierungskante ist.

Da die t verlassende Instanziierungskante nicht notwendigerweise nach s führt, wird die Konvention eingeführt, dass t' der Nachfolger von t auf der Instanziierungskante ist. Dies schließt den Fall $t' = s$ nicht aus.

Für jeden Paarpfad einer starken Zusammenhangskomponente wird ein Abhängigkeitspaar erzeugt. Dabei werden die Substitutionen $\sigma_1, \ldots, \sigma_k$, die auf dem Pfad von s nach t vorkommen, berücksichtigt, indem sie auf s angewendet werden. Die linke Seite des Abhängigkeitspaares in $\mathcal{P}$ für den Paarpfad von s nach t ist s^σ mit $\sigma = \sigma_k \circ \ldots \circ \sigma_1$. Die rechte Seite der Paare aus $\mathcal{P}$ wird aus t berechnet. Dazu wird die Funktion ev verwendet, die erst in Definition 5.18 formal eingeführt wird. An dieser Stelle soll nur die Motivation verdeutlicht werden.

Prinzipiell evaluiert ev einen Ausdruck anhand des Terminierungsgraphen soweit wie möglich. Dies vereinfacht das erzeugte $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Problem in einigen Fällen. Für konkrete Beispiele sei auf [GSSKT06] verwiesen. Zusätzlich werden durch ev folgende problematische Fälle behandelt, in denen der Ausdruck t nicht direkt verwendet werden kann. Dies ist dann der Fall, wenn funktionale Parameter, das heißt Variablen, die für Funktionen stehen, vorhanden sind. Diese werden durch ev eliminiert, indem der entsprechende Teilausdruck durch eine frische Variable ersetzt wird. Das folgende Beispiel aus [GRSK+11] verdeutlicht dies.

Gegeben sei der Teilausdruck `x True`, in dem `x` für eine Funktion vom Typ `Bool` $\rightarrow$ `Bool` steht. Da der funktionale Parameter `x` nicht bestimmt ist, kann keine Ersetzungsregel erzeugt werden, die die Auswertung der übergebenen Funktion simuliert. Stattdessen wird `x True` durch die frische Variable `y` ersetzt. Damit sind alle möglichen Ergebnisse der Evaluation `x True` repräsentiert. Ohne die Ersetzung könnte durch einen solchen Teilausdruck fälschlicherweise die Terminierung attestiert werden, da für die Auswertung der Funktion `x` keine Regeln bekannt sind. In [GRSK+11] finden sich ausführliche Beispiele, die die Motivation für ev weiter verdeutlichen. Die folgenden Prädikate beschreiben Knoten, die mit derartigen Teilausdrücken beschriftet sind oder von denen es Pfade zu Knoten dieser Art gibt.

Definition 5.15. *Das Prädikat $U_G(t)$ ist für einen festen Terminierungsgraphen G über die Knoten t aus G definiert als*

$$U_G \iff \mathrm{ParZerl}_G(t) \wedge t \equiv x\ t_1 \ldots t_n \ \textit{mit } x \textit{ als Variable}$$

und das Prädikat $\mathrm{PU}_G(t)$ *wird definiert als*

$$\mathrm{PU}_G(t) \iff \exists s \in G : U_G(s) \textit{ und es gibt einen Pfad von } t \textit{ nach } s \textit{ in } G.$$

Um die Struktur des Graphen leichter zu beschreiben, werden die folgenden Definitionen eingeführt.

Definition 5.16. *Für einen beliebigen Knoten t in einem Terminierungsgraphen G gilt* $\mathrm{Blatt}_G(t)$ *genau dann, wenn t in G ein Blatt ist.*

Definition 5.17. *Für alle Knoten s und t in einem Terminierungsgraphen G gilt* $\mathrm{Kind}_G(t, s)$ *genau dann, wenn t in G ein Nachfolger von s ist.*

Die Funktion ev wird in [GRSK+11] wie folgt definiert.

Definition 5.18. *Die Funktion* ev *ist für Knoten aus einem Terminierungsgraphen G definiert als*

$$\mathrm{ev}(t) = \begin{cases} x, & \textit{für eine frische Variable } x\textit{, falls } U_G(t) \\ t, & \textit{falls } \mathrm{Blatt}_G(t) \vee \mathrm{Fall}_G(t) \vee \mathrm{VarErw}_G(t) \\ \mathrm{ev}(t'), & \textit{falls } \mathrm{Ausw}_G(t) \wedge \mathrm{Kind}(t', t) \\ C\ h(t_1) \dots h(t_n), & \textit{falls } \mathrm{ParZerl}_G(t) \wedge t \equiv C\ t_1 \dots t_n \\ \mathrm{ev}_{\mathrm{Ins}}(t), & \textit{falls } \mathrm{Ins}_G(t) \end{cases}$$

mit der Hilfsfunktion h definiert als

$$h(s) = \begin{cases} y, & \textit{falls } \mathrm{PU}_G(s) \textit{ wobei } y \textit{ eine frische Variable ist} \\ \mathrm{ev}(s), & \textit{sonst.} \end{cases}$$

und der Hilfsfunktion $\mathrm{ev}_{\mathrm{Ins}}(t)$*, welche für einen Instanziierungsknoten t mit den Kindern $t_1, \dots, t_n, t'$ und der Instanziierungskante von t nach t', sodass $t \equiv t'^{\sigma}$ für $\sigma = [x_1, \dots, x_n := t_1, \dots, t_n]$, definiert wird als*

$$\mathrm{ev}_{\mathrm{Ins}}(t) = t'[x_1, \dots, x_n := h(t_1), \dots, h(t_n)].$$

Die Funktion ev ersetzt Knoten mit einer Beschriftung t, für die $U_G(t)$ gilt, durch eine frische Variable. Blätter, Fallunterscheidungsknoten oder Variablenerweiterungsknoten werden nicht weiter transformiert. Es verbleiben nur noch Instanziierungs-, Auswertungs- oder Parameterzerlegungsknoten, wenn deren Wurzelsymbol ein Konstruktor ist. In diesen Fällen wird ev rekursiv auf die Nachfolgerknoten

angewendet, wobei mit der Funktion h Teilausdrücke, die zu Knoten s für die $U_G(s)$ gilt, ebenfalls durch frische Variablen ersetzt werden.

Mit ev können nun die Abhängigkeitspaare $\mathcal{P}$ des Abhängigkeitspaarproblems $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$ beschrieben werden. Für einen Paarpfad von s nach t mit den Substitutionen $\sigma_1, \ldots, \sigma_k$ wird das Abhängigkeitspaar $\langle s^\sigma, \mathrm{ev}(t)\rangle$ erzeugt.

Wie bereits erwähnt, bleibt das Termersetzungssystem $\mathcal{Q}$ in dem Ansatz von Giesl et al. leer. Das Termersetzungssystems $\mathcal{R}$ wird hingegen benötigt. Die folgende Überlegung beschreibt die Motivation. In einer $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette $\langle s_1, \mathrm{ev}(t_1)\rangle, \langle s_2, \mathrm{ev}(t_2)\rangle$ gilt, dass es Substitution σ_1 und σ_2 gibt mit $\mathrm{ev}(t_1)^{\sigma_1} \twoheadrightarrow_{\mathcal{R}} s_2^{\sigma_2}$. Aus der Definition von ev folgt, dass $\mathrm{ev}(t_1)$ ein Term sein kann, der keine Entsprechung als Knoten in dem Terminierungsgraphen besitzt, während s_2 aus dem Graphen stammt. Intuitiv liegt hierin die Bedeutung des Termersetzungssystems $\mathcal{R}$. Es wird verwendet, um die weitere Evaluation zu simulieren. Die Regeln in $\mathcal{R}$ werden analog zu dem Paarpfad aus einem Regelpfad abgeleitet, der in [GRSK+11] wie folgt definiert wird.

Definition 5.19. *Ein Pfad von s nach t in einem Terminierungsgraphen G heißt* Regelpfad*, wenn t weder ein Auswertungsknoten noch ein Fallunterscheidungsknoten ist, aber s und alle Knoten auf dem Pfad außer t Auswertungs- oder Fallunterscheidungsknoten sind.*

Um das weitere Vorgehen zu motivieren, betrachten wir folgendes Beispiel. Sei s ein Knoten von dem ein Regelpfad zu dem Parameterzerlegungsknoten t führt. Der Knoten t kann mit ev weiter zu $\mathrm{ev}(t)$ ausgewertet werden. Wiederum gilt, dass $\mathrm{ev}(t)$ ein Term ist, der keine Entsprechung als Knoten in dem Terminierungsgraphen haben muss. Um die weitere Auswertung zu simulieren, werden von t aus sämtliche Knoten bestimmt, nach denen weitere Auswertungen stattfinden können, die nicht bereits durch ev abgedeckt werden. Dies sind Fallunterscheidungsknoten und die Instanziierungsnachfolger von Instanziierungsknoten. Die Funktion con beschreibt dies formal.

Definition 5.20. *Für einen Knoten t sei die Menge* $\mathrm{con}(t)$ *definiert als*

$$\mathrm{con}(t) = \begin{cases} \emptyset, & \textit{falls } \mathrm{Blatt}(t) \vee \mathrm{VarErw}(t) \vee \mathrm{PU}_G(t) \\ \{t\}, & \textit{falls } \mathrm{Fall}(t) \\ \{t'\} \cup \mathrm{con}(t_1) \cup \ldots \cup \mathrm{con}(t_n), & \textit{falls } \mathrm{Ins}(t) \\ \bigcup_{\mathrm{Kind}(t',t)} \mathrm{con}(t') & \textit{sonst.} \end{cases}$$

Dabei gilt falls $\mathrm{Ins}(t)$*, dass $t_1, \ldots, t_n, t'$ die Kinder von t sind und die Instanziierungskante von t nach t' führt.*

Für einen einzelnen Knoten s kann es mehrere Regelpfade geben. Um für einen Knoten alle Regeln zu bestimmen, wird die Funktion rl definiert. Die Menge $\mathrm{rl}(s)$ umfasst alle Regeln, die sich für einen Knoten s ergeben. Sie wird von Giesl et al. rekursiv definiert, sodass auch alle folgenden Berechnungen erfasst werden.

Definition 5.21. *Für einen Knoten s ist die Funktion $\mathrm{rl}(s)$ definiert als die kleinste Menge von Regeln, sodass*

$$\begin{aligned} (s^{\sigma_k \circ \ldots \circ \sigma_1} \to \mathrm{ev}(t)) &\in \mathrm{rl}(s) \text{ und} \\ \mathrm{rl}(q) &\subseteq \mathrm{rl}(s) \end{aligned}$$

wenn es einen Regelpfad von s nach t mit den Substitutionen $\sigma_1, \ldots, \sigma_k$ gibt und $q \in \mathrm{con}(t)$.

Lemma 5.22. *Sei G ein Terminierungsgraph. Für jeden Knoten s ist die Menge $\mathrm{rl}(s)$ wohldefiniert.*

Weder in [GSSKT06] noch in [GRSK⁺11] findet sich ein Beweis für die Existenz der Menge $\mathrm{rl}(s)$. In der vorliegenden Arbeit wird Lemma 5.22 wie folgt bewiesen.

Beweis. Die Existenz folgt aus der Endlichkeit des Graphen G und der endlichen Anzahl an Regelpfaden in G. Die Endlichkeit des Graphen G ergibt sich aus Lemma 5.13. Des Weiteren gilt, dass nach Definition 5.19 lediglich der letzte Knoten eines Regelpfads ein Instanziierungsknoten sein kann. Ein Regelpfad enthält daher keine Instanziierungskanten. Damit sind alle Regelpfade in G auch in dem Graphen G' enthalten, der sich aus G durch Entfernen sämtlicher Instanziierungskanten ergibt. Nach Definition des Terminierungsgraphen ist G' ein Baum und damit azyklisch. Es folgt, dass es nur endlich viele Regelpfade gibt. □

Mit der Funktion rl beschreiben Giesl et al. das vollständige Abhängigkeitspaarproblem, das sich aus einer starken Zusammenhangskomponente ergibt.

Definition 5.23. *Sei G' eine starke Zusammenhangskomponente des Terminierungsgraphen G. Das Abhängigkeitspaarproblem $\mathrm{dp}(G')$ wird definiert als $(\mathcal{P}, \emptyset, \mathcal{R})$ wobei $\mathcal{P}$ und $\mathcal{R}$ die kleinsten Mengen sind, sodass*

$$\begin{aligned} \langle s^{\sigma_k \circ \ldots \circ \sigma_1}, \mathrm{ev}(t) \rangle &\in \mathcal{P} \text{ und} \\ \mathrm{rl}(q) &\subseteq \mathcal{R} \end{aligned}$$

für alle Paarpfade in G' von s nach t mit den Substitutionen $\sigma_1, \ldots, \sigma_k$ enthält, von $t = t'[x_1, \ldots, x_n := t_1, \ldots, t_n]$ eine Instanziierungskante nach t' führt und $q \in$ $\text{con}(t_1) \cup \ldots \cup \text{con}(t_n)$.

Es sei darauf hingewiesen, dass q nicht aus der Menge $\text{con}(t')$ stammt. Dies ist nicht nötig, da t' entweder Ausgangspunkt eines neuen Paarpfades ist oder die starke Zusammenhangskomponente verlässt. Die Korrektheit des Ansatzes beschreibt folgendes Theorem aus [GSSKT06].

Theorem 5.24. *Sei G ein Terminierungsgraph. Wenn die Abhängigkeitspaarprobleme* $\text{dp}(G')$ *für alle starken Zusammenhangskomponenten G' von G finit sind, so sind alle Knoten in G H-terminierend.*

Beweis. Der vollständige Beweis findet sich in [GSSKT06]. An dieser Stelle soll nur die Idee vermittelt werden. Es reicht zu zeigen, dass aus der Existenz eines Knoten in G', der mit einem nicht H-terminierenden Ausdruck beschriftet ist, folgt, dass es für eine starke Zusammenhangskomponente G' mit $\text{dp}(G') = (\mathcal{P}, \mathcal{Q}, \mathcal{R})$ eine unendliche $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette gibt. Dies kann bewiesen werden, indem für jede Reduktionsfolge bezüglich $\to_H$ gezeigt wird, dass es eine korrespondierende $(\mathcal{P}, \mathcal{Q}, \mathcal{R})$-Kette gibt. Für eine unendliche Reduktionsfolge ergibt sich somit eine unendliche Kette.

Analog zu der Argumentation im Beweis von Theorem 4.26 wird ein möglicherweise existierendes endliches Präfix der Reduktionsfolge außer Acht gelassen, in dem Ausdrücke auftreten, die nicht mit Knoten aus G' korrespondieren. Giesl et al. zeigen, dass es für jeden Knoten s in G, der für eine Substitution μ_1 eine unendliche Reduktionsfolge auslöst, einen Pfad zu einen Instanziierungsknoten t in G gibt. Dieser Pfad ist ein Paarpfad mit den Substitutionen $\rho_1, \ldots, \rho_n$. Zu dem Paarpfad gehört daher das Abhängigkeitspaar $\langle s^\rho, \text{ev}(t) \rangle$, wobei $\rho = \rho_n \circ \ldots \circ \rho_1$. Sei dies das erste Abhängigkeitspaar in der unendlichen Kette. Die zugehörige Substitution σ_1 ergibt sich nach $\mu_1 = \sigma_1 \circ \rho$. Ferner wird gezeigt, dass $\text{ev}(t)^{\sigma_1} \twoheadrightarrow_{\mathcal{R}} t'^{\mu_2}$ gilt und t'^{μ_2} ein Ausdruck ist, der analog zu s^{μ_1} eine unendliche Reduktionsfolge beginnt. Die Konstruktion kann daher unendlich oft fortgesetzt werden, sodass eine unendliche Kette entsteht. □

Das in diesem Kapitel beschriebene Verfahren von Giesl et al. wird im nächsten Kapitel genutzt, um für imperative Programme ein automatisiertes Verfahren zur Terminierungsanalyse zu entwickeln, indem imperative Programme in Haskell-Programme übersetzt werden.

6 Automatisierte Terminierungsanalyse imperativer Sprachen

In diesem Kapitel wird betrachtet, wie sich das Dependency Pair Framework einsetzen lässt, um Terminierungseigenschaften imperativer Programme automatisiert zu ermitteln. Leider ist nicht offensichtlich, wie ein Programm in ein Abhängigkeitspaarproblem überführt werden kann. Um ein solches Verfahren zu konstruieren, genügen nicht nur Kenntnisse der zu analysierenden Sprache. Vielmehr wird auch eine gute Kenntnis des Dependency Pair Frameworks benötigt. Imperative Programme in Haskell-Programme zu übersetzen, fällt aufgrund der mathematischen Natur Haskells vergleichsweise leicht. Es erscheint daher einfacher, einen solchen Übersetzer zu konstruieren und das übersetzte Programm mit dem Verfahren aus Kapitel 5 zu analysieren. Da das übersetzte Programm semantisch äquivalent zu dem Eingabeprogramm ist, folgt auch die Übertragbarkeit des Ergebnisses.

Simon Peyton Jones schreibt am Ende der Einleitung zu [PJ03], dass nach seiner Überzeugung Haskell die beste imperative Programmiersprache der Welt sei. Peyton Jones begründet dies damit, dass der monadische Programmieransatz erlaubt, imperative Anweisungen als Werte zu betrachten (sogenannte Aktionen) und wie gewöhnliche Daten zu verarbeiten. Während in imperativen Sprachen mit vordefinierten Konstrukten zur Steuerung des Kontrollflusses gearbeitet werden muss, können in Haskell eigene Kombinatoren für Aktionen definiert werden, die an das konkret zu lösende Problem angepasst sind. Aus demselben Grund schreiben O'Sullivan et al. in [OSG09], dass Monaden eine Art „programmierbares Semikolon" darstellen.

Die Zustandstransitionsmonade[1] ist die naheliegende Wahl, um imperative Sprachen zu simulieren. Imperative Sprachen sind dadurch gekennzeichnet, dass Anweisungen aneinandergereiht werden. Jede dieser Anweisungen beeinflusst dabei einen globalen Zustand. Eine Anweisung ist daher eine Zustandstransition und ein impe-

[1] Häufig findet sich in der Literatur auch die Bezeichnung Zustandsmonade. In der vorliegenden Arbeit wird der Begriff Zustandstransitionsmonade oder kurz Transitionsmonade vorgezogen.

ratives Programm nichts anderes als ein Folge solcher Transitionen. Ein Beispiel für dieses Vorgehen zur Übersetzung der Sprache SETL [SDSD86] findet sich in [Sch08]. Grundsätzlich ist es dabei notwendig, den Zustandsraum der Sprache und die primitiven Kontrollflusskonstruktionen zu modellieren. Dieses generische Vorgehen lässt sich auf alle imperativen Sprachen anwenden. Kombiniert mit den gewöhnlichen Techniken des Übersetzerbaus (siehe zum Beispiel [ASU88] oder [App97]) kann leicht ein Übersetzer konstruiert werden, der Haskell als Zielsprache hat.

In Verbindung mit einem Verfahren zur Terminierungsanalyse für Haskell, wie zum Beispiel AProVE, ergibt sich so ein Werkzeug zur Terminierungsanalyse von imperativen Programmen. Intuitiv erwartet man jedoch, dass ein solches kombiniertes Werkzeug nicht für viele Programme zu einem definierten Ergebnis führt. Dieses Kapitel bestätigt diese Einschätzung und zeigt, wodurch diese Probleme entstehen.

Im ersten Abschnitt dieses Kapitels wird ein Problem bei der Terminierungsanalyse betrachtet, das sich durch die Verwendung von Kombinatoren ergibt. Es wird eine Programmtransformation eingeführt, die dieses Problem löst. In diesem Abschnitt wird noch ein sehr einfacher Zustandsraum verwendet. Komplexere Zustandsräume erfordern jedoch ausgefeiltere Methoden zur Verwaltung der Zustände, wie zum Beispiel Transitionsmonaden. Als Problem wird sich hierbei die veränderliche Belegung von Variablen erweisen. Im folgenden Abschnitt werden die sich damit ergebenden Schwierigkeiten für die automatisierte Analyse erläutert. Im letzten Abschnitt wird eine alternative Form der Übersetzung vorgestellt, die auf der SSA-CPS-Korrespondenz beruht. Dies geschieht anhand der konkreten Sprache Jinja [KN06]. Durch die Nutzung der SSA-CPS-Korrespondenz kann die Variablenbelegung so ausgedrückt werden, dass die Terminierungsanalyse wiederum gelingt. Es verbleibt jedoch ein Problem, dass sich mit der Darstellung des Laufzeit-Heaps ergibt.

6.1 Kombinatoren

Ein erster Ansatz zur Übersetzung einer imperativen Sprache beruht auf der Idee, die Zustandsübergangsrelation eines Programms, wie sie die operationelle Semantik der Quellsprache beschreibt, in einem Haskell-Programm abzubilden. Häufig lassen sich alle Regeln, die zur Beschreibung der Semantik eines Sprachkonstrukts nötig sind, mithilfe eines Kombinators in Haskell formalisieren.

6.1.1 Bedeutung von Kombinatoren

Der Kern einer imperativen Sprache ist aus Sicht der operationellen Semantik die Menge der Regeln, die die möglichen Zustandsübergänge beschreiben. Für ein konkretes Programm und einen festen Startzustand ergibt sich damit eine Folge von Zuständen, die dem Ablauf des Programms entsprechen. Für Turing-vollständige Sprachen ist diese Folge möglicherweise unendlich. Das Ziel des Übersetzers ist nun, ein Programm zu konstruieren, welches diese Folge von Zuständen berechnet. Daher ist es notwendig, einen Datentyp für den Programmzustand zu definieren. Aufgrund der Turing-Vollständigkeit Haskells gelingt dies stets. Allerdings gestaltet es sich schwierig, eine für die Terminierungsanalyse geeignete Darstellung des Zustandsraums zu finden. Dies wird in Abschnitt 6.2 gezeigt. In diesem Abschnitt nehmen wir an, dass eine solche Repräsentation existiert. Insbesondere werden wir im folgenden Beispiel den Zustandsraum so wählen, dass wir sicher sind, dass das gezeigte Problem nicht durch den Zustandsraum verursacht wird. Der Fokus des Abschnitts liegt auf den Problemen, die sich aus den Kombinatoren ergeben.

Die operationelle Semantik der Quellsprache beschreibt die möglichen Übergänge mit Hilfe von sprachspezifischen Regeln. Die in diesem Abschnitt betrachtete Technik zur Übersetzung imperativer Programme definiert Kombinatoren, die die Regeln, die ein Sprachkonstrukt beschreiben, zusammenfasst und so ihre Semantik in Haskell abbildet. Kombinatoren sind also Funktionen, die gemäß der Semantik der Quellsprache und anhand des Zustands entscheiden, welcher Zustandsübergang erfolgt. Ein Beispiel sind bedingte Anweisungen, mit denen die Ausführung einer Anweisung an eine Bedingung geknüpft wird, sodass die Anweisung nur ausgeführt wird, falls die Bedingung erfüllt ist. In der operationellen Semantik finden sich hierfür zwei Regeln, die die beiden Fälle beschreiben. Beide Regeln lassen sich mit einem dreistelligen Kombinator zusammenfassen. Das erste Argument stellt den Zustand dar, das zweite den Ausdruck und das dritte die Anweisung. Das Resultat ist der Folgezustand. Mit `State` als Zustandstyp ergibt sich eine Definition wie aus Quelltext 6.1. Im Folgenden wird ein Beispiel gezeigt, welches diesen Ansatz und die resultierenden Probleme bei der Terminierungsanalyse illustriert.

6.1.2 Ein einfaches, terminierendes Programm

Grundlage für das Beispiel ist das Programm aus Quelltext 6.2. Das Programm ist in der Sprache C (siehe [Ker88]) verfasst. Die Variable `x` ist vom Typ `unsigned int` und repräsentiert damit eine positive ganze Zahl. Das Programm terminiert offensichtlich, da `x` so lange dekrementiert wird, bis es den Wert 0 trägt.

Quelltextauszug 6.1: Der Konditional-Kombinator

```
cond :: State -> (State -> Bool) -> (State -> State) -> State
cond state predicate transition
    = if (predicate state)
          then (transition state)
          else state
```

Quelltextauszug 6.2: Das Countdown-Programm in C

```
unsigned int x = ...;
while (x != 0) {
    x--;
}
```

Die in Quelltext 6.3 gezeigte Funktion `countdown` stellt eine einfache Haskell-Übersetzung des C-Programms aus 6.2 dar, bei der ein Kombinator für `while` eingesetzt wurde. Die Funktion `select` stellt einen bedingten Ausdruck dar. Haskell bietet dafür die Konstruktion `if then else`. Diese gehört jedoch, wie in Abschnitt 5.1.3 dargelegt, nicht zu der reduzierten Haskell-Variante, sondern wird durch eine Konstruktion wie `select` ersetzt.

Der Typ `UnsignedInt` repräsentiert streng genommen die natürlichen Zahlen, während der Typ `unsigned int` in C in der Größe beschränkt ist. Dies stellt jedoch kein Problem dar, da nach Definition der H-Terminierung 5.6 nur endliche Wert vom Typ `UnsignedInt` betrachtet werden. Der Zustandsraum wird durch `UnsignedInt` vollständig repräsentiert. Dies gelingt, da es lediglich eine Variable in dem Ausgangsprogramm gibt.

Die Typsynonyme `Predicate` und `Transition` sind der Übersichtlichkeit halber vorhanden, obwohl sie nicht explizit Teil der reduzierten Fassung sind. Die Konstruktion lässt sich durch die Ersetzung der jeweiligen Typen leicht vermeiden. Da der Zustandsraum auf eine Variable verengt ist, können boolesche Ausdrücke und Zustandsübergänge jeweils vollständig mit `Predicate` und `Transition` dargestellt werden.

Die Funktion `dec` entspricht dem Dekrementierungsoperator −−, mit dem Unterschied, dass `dec Zero` keinen Überlauf verursacht. Da das Programm `countdown` jedoch `dec Zero` nicht auswertet, ergibt sich kein Unterschied. Die Funktion `notZero` entspricht dem booleschen Ausdruck `x != 0` des C-Programms.

Der einzige Kombinator, der in dem Programm definiert wird, ist `while`. Ab-

Quelltextauszug 6.3: Das Countdown-Programm in Haskell

```
select :: Bool -> a -> a -> a
select True  x y = x
select False x y = y

data UnsignedInt = Zero | Successor UnsignedInt

type Predicate = UnsignedInt -> Bool

type Transition = UnsignedInt -> UnsignedInt

dec :: Transition
dec Zero          = Zero
dec (Successor x) = x

notZero :: Predicate
notZero (Successor x) = True
notZero Zero          = False

while :: Predicate -> Transition -> Transition
while predicate trans state
    = select
          (pred state)
          (while pred trans (trans state))
          state

countdown :: Transition
countdown = while notZero dec
```

hängig vom Ergebnis der Auswertung des Prädikats, wird die Auswertung für den durch die angegebene Transition transformierten Zustand wiederholt oder beendet. Der Ausdruck `while` ist für sich genommen nicht H-terminierend. In der Anwendung auf `notZero` und `dec` ergibt sich jedoch ein H-terminierender Ausdruck, da die resultierende Transition für keinen H-terminierenden Ausdruck ohne Variablen eine unendliche Auswertungsfolge beginnt. Zu diesem Ergebnis gelangt die automatisierte Terminierungsanalyse jedoch nicht, wie im Anschluss demonstriert wird.

6.1.3 Probleme bei der Terminierungsanalyse

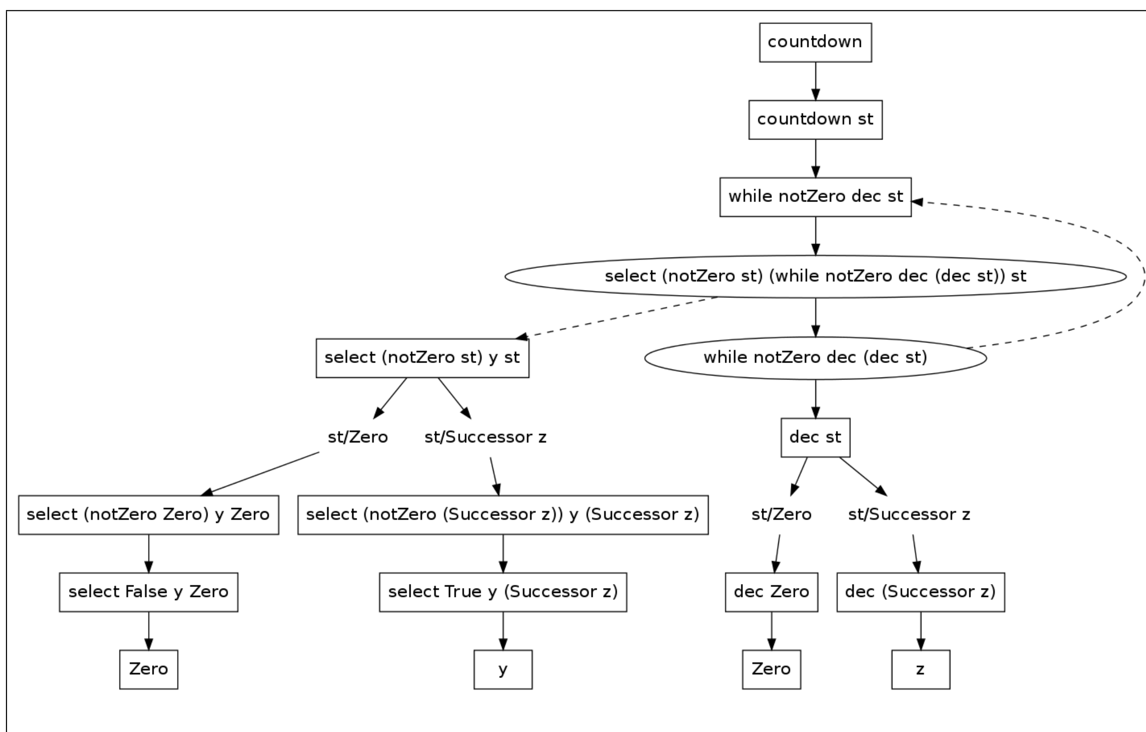

Abbildung 6.1: Der Terminierungsgraph der Funktion `countdown`

Die automatisierte Terminierungsanalyse beginnt mit der Überführung des Programms in einen Terminierungsgraphen. Das System AProVE erzeugt für die Funktion `countdown` den Terminierungsgraphen in Abbildung 6.1. Ovale Knoten sind Instanziierungsknoten, deren Instanziierungskante gestrichelt dargestellt wird.

Der initiale Knoten des Graphen ist mit `countdown` beschriftet. Nach der Variablenergänzungsregel entsteht der Knoten `countdown st`, welcher ein Auswertungsknoten ist und zu `while notZero dec st` ausgewertet wird. Nach Definition von `while` ergibt sich der Knoten

`select (notZero st) (while notZero dec (dec st)) st.`

Dies stellt den längsten Ausdruck in dem Graphen dar. Mithilfe der Instanziierungsregel kann dieser Term als Instanziierung des Ausdrucks `select (notZero st) st' st` betrachtet werden, indem die Substitution [`st'` := `while notZero dec (dec st)`] gewählt wird. Entsprechend wird neben dem Instanziierungsnachfolger ein weiterer Nachfolgerknoten mit der Beschriftung `while notZero dec (dec st)` erzeugt. In Abbildung 6.1 findet sich der Instanziierungsnachfolger auf der linken Seite. Auf dieser Seite wird durch die Fallunterscheidungsregel die Variable `st` nach dem Instanziierungsnachfolger durch `Zero` und `Successor n` belegt. In beiden Fällen wird die Auswertungsregel zweifach angewendet und es entstehen die Blätter `Zero` und `st'`.

Auf der rechten Seite von Abbildung 6.1 wird `while notZero dec (dec st)` als Instanziierung des bereits existierenden Knoten `while notZero dec st` betrachtet. Dazu ist die Substitution [`st` := `dec st`] notwendig. Es ergibt sich der Nachfolger `dec st`, welcher durch Fallunterscheidung für `st` in `Zero` und `Successor n` und der Auswertungsregel zu den Blättern `Zero` und `n` führt.

Der Graph besitzt eine starke Zusammenhangskomponente, die die beiden Knoten `while notZero dec st` und `while notZero dec (dec st)` beinhaltet, sowie den dazwischenliegenden Knoten, der mit `select` beginnt. Es ergibt sich lediglich ein Paarpfad, auf dem alle Knoten der Zusammenhangskomponente liegen und der von `while notZero dec st` nach `while notZero dec (dec st)` führt. Damit gilt, dass $\mathcal{P} = \{\langle \texttt{while}(x), \texttt{while}(\texttt{dec}(x))\rangle\}$. Ferner ergibt sich, dass $\mathcal{R} = \{\texttt{dec}(\texttt{Successor}(x)) \to \texttt{Successor}(x), \texttt{dec}(\texttt{Zero}) \to \texttt{Zero}\}$.

Lemma 6.1. *Das Problem* $(\mathcal{P}, \emptyset, \mathcal{R})$ *ist nicht finit.*

Beweis. Es wird eine unendliche $(\mathcal{P}, \emptyset, \mathcal{R})$-Kette konstruiert. Da es nur ein Abhängigkeitspaar gibt, muss die Kette die Form

$$\langle \texttt{while}(x), \texttt{while}(\texttt{dec}(x))\rangle, \langle \texttt{while}(x), \texttt{while}(\texttt{dec}(x))\rangle, \dots$$

haben. Es wird gezeigt, dass dies tatsächlich eine Kette ist, also Substitutionen σ_i existieren, für die

$$\texttt{while}(\texttt{dec}(x))^{\sigma_i} \twoheadrightarrow_{\mathcal{R}} \texttt{while}(x)^{\sigma_{i+1}}$$

stets gilt. Da $\twoheadrightarrow_{\mathcal{R}}$ die reflexive Hülle umfasst, reicht es zu zeigen, dass

$$\texttt{while}(\texttt{dec}(x))^{\sigma_i} \equiv \texttt{while}(x)^{\sigma_{i+1}}$$

gilt. Sei σ_1 die Identität und $\sigma_i = [x := \texttt{dec}(\sigma_{i-1}(x))]$ für $i > 1$. Es folgt, dass

$$\texttt{while}(x)^{\sigma_{i+1}} = \texttt{while}(x^{\sigma_{i+1}}) = \texttt{while}(\texttt{dec}(x^{\sigma_i})) = \texttt{while}(\texttt{dec}(x))^{\sigma_i}.$$

und damit die Behauptung. □

Aus dem Beweis wird auch klar, dass die Wahl von `dec Zero = Zero` unproblematisch war. Aus dieser Gleichung hat sich lediglich eine Regel in $\mathcal{R}$ ergeben. Der vorangegangene Beweis kann jedoch auch geführt werden, wenn $\mathcal{R} = \emptyset$ gilt, da keine Regel aus $\mathcal{R}$ angewendet wird.

Mit Theorem 5.24 kann daher die Terminierung nicht bewiesen werden. Umgekehrt kann allerdings auch nicht die Nicht-Terminierung gezeigt werden. Das System AProVE antwortet dementsprechend mit der undefinierten Antwort „Maybe“.

6.1.4 Problemvermeidung durch Programmtransformation

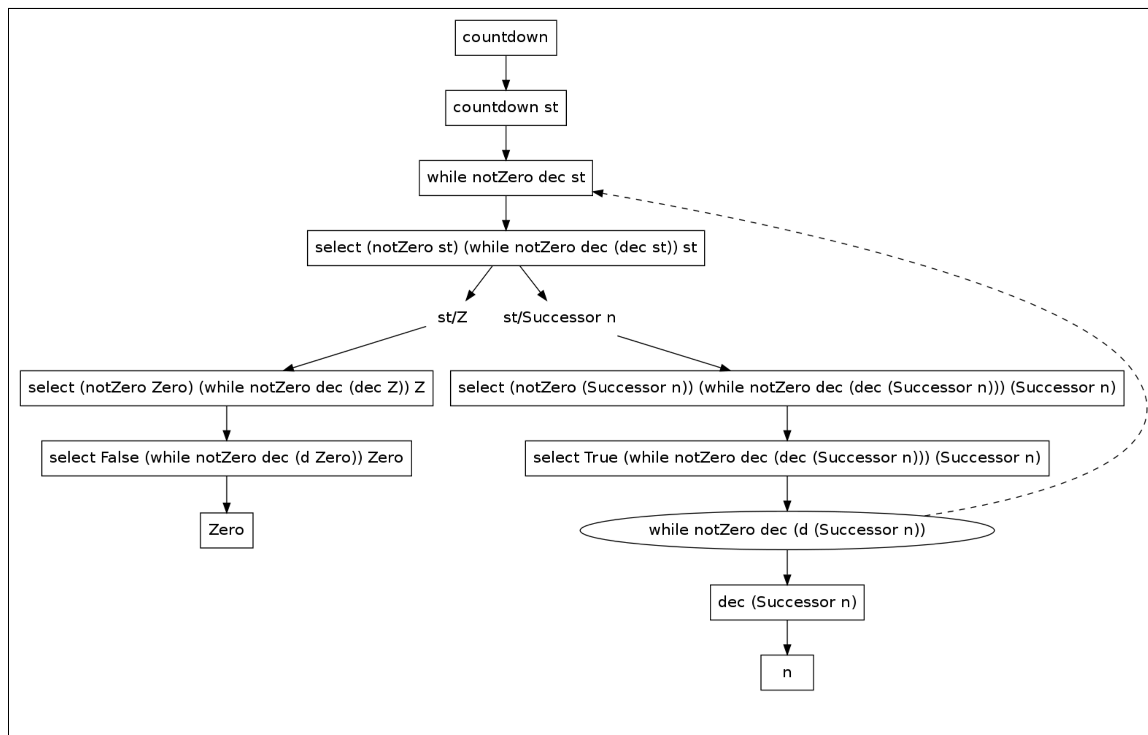

Abbildung 6.2: Alternativer Terminierungsgraph für `countdown`

Die Ursache für das unbefriedigende Ergebnis ist in dem Terminierungsgraphen zu suchen. Der durch AProVE erzeugte Graph aus Abbildung 6.1 ist insofern problematisch, als dass die Zusammenhangskomponente zu klein ist, beziehungsweise der entstandene Paarpfad zu kurz ist und daher nicht genügend Information trägt.

Die für die Terminierung entscheidende Dekrementierung ist auf dem Paarpfad nicht erfasst. Wird nur die Zusammenhangskomponente betrachtet, so scheint die Berechnung unendlich fortzuschreiten, da immer neue Auswertungen von `dec` notwendig werden. Tatsächlich können jedoch nur endlich viele `dec`-Auswertungen stattfinden, nur findet sich diese Information nicht in der Zusammenhangskomponente und damit nicht auf dem Paarpfad.

Der Terminierungsgraph aus Abbildung 6.1 ist nicht der einzig mögliche Graph. Es handelt sich lediglich um den von APROVE erzeugten. Der Terminierungsgraph aus Abbildung 6.2 ist ebenfalls ein valider Terminierungsgraph für die Funktion `countdown`, der allerdings zu einem finiten Abhängigkeitspaarproblem führt. Es existiert wiederum nur eine starke Zusammenhangskomponente. Diese enthält jedoch einen längeren Paarpfad. Es entsteht $\mathcal{P}$ mit dem einzigen Abhängigkeitspaar

$$\langle \texttt{while}(\texttt{Successor}(x)), \texttt{while}(\texttt{dec}(x)) \rangle$$

und $\texttt{dec}(\texttt{Successor}(x)) \to x$ als einzige Regel in $\mathcal{R}$. Dieses Problem ist offensichtlich finit und automatisierte Verfahren sind in der Lage, dies zu beweisen.

Die folgende Programmtransformation erzeugt ein Programm, für das APROVE automatisch einen geeigneten Terminierungsgraphen erzeugt. Um den Berechnungsschritt in die Zusammenhangskomponente zu ziehen, muss verhindert werden, dass zu früh eine Instanziierungskante gezogen werden kann. Dies kann erreicht werden, indem die rekursive Funktion `while` durch zwei Funktionen `while0` und `while1` ersetzt wird, welche sich gegenseitig rekursiv aufrufen. In Quelltext 6.4 ist dies durchgeführt worden.

Die beschriebene Programmtransformation lässt sich allgemein formulieren.

Definition 6.2. *Sei f eine n-stellige Funktion, die durch k Gleichungen definiert ist. Sei $f\ s_{i,1}\ \ldots\ s_{i,n} = t_i$ die i-te Gleichung der Definition und $P_f(t_i)$ die Menge der Positionen, an denen in t_i ein Teilausdruck mit f als Wurzelsymbol steht. Die Funktion $r_f(t_i, j)$ ersetzt an allen Stellen $P_f(t_i)$ das Symbol f in t_i durch f_j. Die Transformation führt zwei neue Funktionen f_0 und f_1 mit den gleichen Stelligkeiten und derselben Anzahl an Funktionsgleichungen. Die Definitionsgleichungen der Funktion f_0 sind $f_0\ s_{i,1}\ \ldots\ s_{i,n} = r_f(t_i, 1)$ und die der Funktion f_1 sind $f_1\ s_{i,1}\ \ldots\ s_{i,n} = r_f(t_i, 0)$. Die Anwendung der Transformation wird als* Ausrollen *bezeichnet. Die Funktionen f_0 und f_1 heißen* ausgerollte Funktionen von f.

Die Transformation verändert die Semantik des Programms nicht, wie das folgende Lemma zeigt.

Quelltextauszug 6.4: Das Countdown-Programm in Haskell

```
while0, while1 :: Predicate -> Transition -> Transition
while0 predicate trans state
     = select
          (pred state)
          (while1 pred trans (trans state))
          state

while1 predicate trans state
     = select
          (pred state)
          (while0 pred trans (trans state))
          state

countdown :: Transition
countdown = while0 notZero dec
```

Lemma 6.3. *Seien f_0 und f_1 die ausgerollten Funktionen von f. Die Funktionen f_0, f_1 und f sind semantisch äquivalent.*

Beweis. Die Behauptung folgt offensichtlich, wenn f nicht rekursiv ist, das heißt für alle Gleichungen i gilt, $P_f(t_i) = \emptyset$. Der Beweis erfolgt per Induktion über die Tiefe d der Rekursion eines beliebigen Ausdrucks s mit f als Wurzelsymbol. Für Tiefe $d = 0$ folgt die Behauptung. Wenn s ein Fehlerausdruck für f ist, so ist s auch ein Fehlerausdruck für f_0 und f_1, da die Argumente der linken Seiten aller drei Funktionsgleichungen identisch sind. Sei i die Nummer der Gleichung, die bei der Auswertung von s genutzt wird und m die Rekursionstiefe von s. Es folgt, dass die Rekursionstiefe von t_i genau $m-1$ ist. Nach Induktionsannahme sind die Funktionen f_0, f_1 und f semantisch äquivalent für die Tiefe $m-1$. Da in der rechten Seite der i-ten Definitionsgleichung von f_0 an jeder Position $P_f(t-i)$ die Funktion f_1 steht, folgt, dass f_0 semantisch äquivalent zu f ist. Analoges gilt für f_1. □

Der automatisch erzeugte Graph für das Countdown-Programm mit ausgerollten `while`-Kombinator wird in Abbildung 6.3 dargestellt. Aus Gründen der Übersichtlichkeit wurden die Namen der Funktionen gekürzt. Auch für diesen Graph ergibt sich dasselbe Abhängigkeitspaarproblem wie für den Graphen aus Abbildung 6.2.

Nach der Betrachtung der Kombinatoren wenden wir uns im nächsten Abschnitt einer eleganteren Methode zur Übersetzung zu und betrachten vor allem die Modellierung des Zustandsraums.

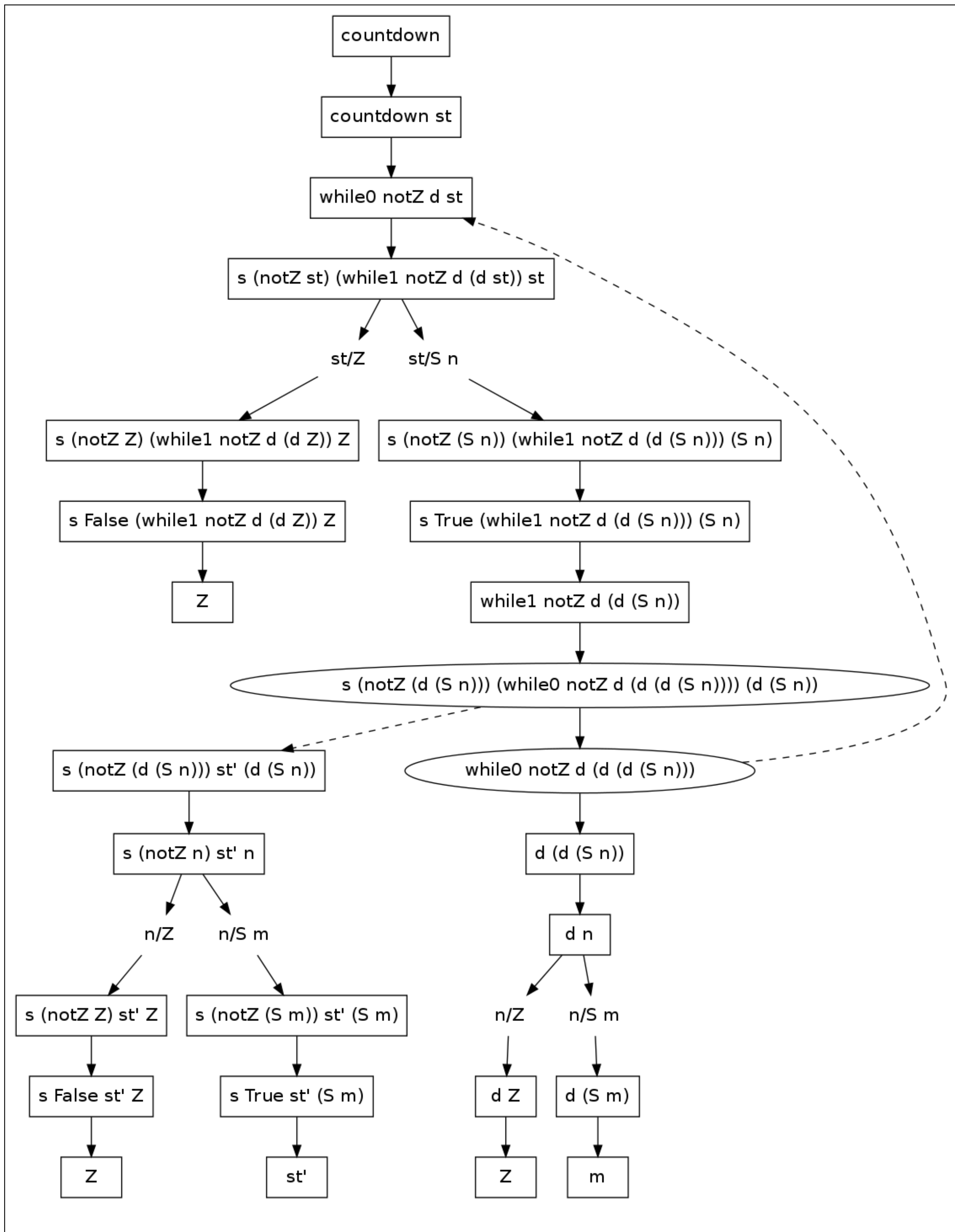

Abbildung 6.3: Automatisch erzeugter Terminierungsgraph für `countdown` mit ausgerolltem `while`-Kombinator.

6.2 Verwaltung und Darstellung des Programmzustands

Variablen in imperativen Programmen sind im Allgemeinen veränderlich, ihr Wert ist vom aktuellen Programmzustand abhängig. Dies steht im Gegensatz zu der referentiellen Transparenz rein funktionaler Sprachen wie Haskell. Um imperative Programme in Haskell zu simulieren, ist es daher notwendig, den veränderlichen Zustand durch unveränderliche Datenstrukturen zu simulieren. Jede Zustandsänderung erzeugt einen neuen Zustand. Es ist erforderlich, den jeweils aktuellen Zustand zu kennen und die Übergänge korrekt zu simulieren. Transitionsmonaden erweisen sich dabei als äußerst nützlich.

6.2.1 Verwaltung des Zustands mit Transitionsmonaden

Eine Realisierung der Transitionsmonade in Haskell wird in Quelltext 6.5 dargestellt. Der Typ `StateTransition state a` repräsentiert Zustandsübergänge in einem Zustandstyp `state`, die neben dem Folgezustand noch ein Ergebnis vom Typ `a` ergeben. Die Funktion `runWith` dient lediglich der Anwendung der Zustandstransition auf einen konkreten Zustand. Der Typ `StateTransition state` kann damit Mitglied der Typklasse `Monad` werden, die die Funktionen `return` und >>= (gesprochen bind) vorschreibt. Für die Transitionsmonade erzeugt `return` einen identitären Zustandsübergang, dessen Ergebnis das Argument von `return` ist. Die Komposition >>= dient dazu, zwei Zustandsübergänge aneinander zu binden. Dabei kann der zweite

Quelltextauszug 6.5: Implementierung der Transitionsmonade

```haskell
data StateTransition state a
   = Transition (state -> (a, state))

runWith :: StateTransition state a -> state -> (a, state)
runWith (Transition transition) = transition

instance Monad (StateTransition state) where
  return v = Transition (\state -> (v,state))
  t >>= f = Transition (\state ->
                      let (v, state') = runWith t state
                      in runWith (f v) state')
```

Übergang abhängig vom ersten sein. Dies drückt sich in dem Typ

$$(>>=) :: \texttt{ma} \rightarrow (\texttt{a} \rightarrow \texttt{mb}) \rightarrow \texttt{mb}$$

aus. Die Semantik der Transition $t >>= f$ ist, dass zuerst t im Eingabezustand ausgewertet wird. Das Resultat wird mit f auf die Folgetransition abgebildet und diese mit dem Folgezustand ausgeführt.

Die Instanziierung der Typklasse `Monad` durch den Typ `StateTransition` stellt eine Monade dar. Zum Nachweis sind die sogenannten Monadengesetze (Links- und Rechtsidentität von `return` bezüglich >>=, so wie die Assoziativität von >>=) zu zeigen. Da im Folgenden die Monadeneigenschaften nicht weiter von Interesse sind, wird für den Nachweis auf die Literatur verwiesen.

Durch eine geeignete Wahl des Zustandstyps kann die allgemeine Zustandstransitionsmonade genutzt werden, um den Ablauf imperativer Programme in Haskell zu simulieren. Der Wechsel zwischen den Zuständen ist dabei in der Transitionsmonade gekapselt. Innerhalb der zustandsbehafteten Simulation ist ein Zugriff auf den gesamten Zustand nicht erforderlich. Vielmehr genügt es, mit geeigneten Funktionen spezifische Informationen aus dem Zustand abzufragen oder in definierter Art und Weise zu manipulieren. Einen wesentlichen Teil des Zustands stellt die veränderliche Variablenbelegung dar. Mit der Repräsentation dieser Komponente beschäftigt sich der folgenden Unterabschnitt.

6.2.2 Darstellung der Variablenbelegung

Für unsere Betrachtungen verzichten wir darauf, unterschiedliche Typen zu simulieren. Es sollen lediglich natürliche Zahlen repräsentiert werden und dazu wird der Datentyp `Nat` genutzt. Sei `State` der Name des Zustandsdatentyps. Die Bezeichner von Variablen werden durch den Typ `Ident` repräsentiert. Der Zugriff auf diese Komponente erfolgt mit zwei speziellen Zustandsübergängen `get` und `set`.

Die Funktion `get` vom Typ `Ident` → `StateTransition State Nat` bildet von einem Variablenbezeichner auf den aktuellen Wert ab. Formal handelt es sich um einen Zustandsübergang, bei dem sich der Zustand nicht ändert, sondern lediglich ein Wert aus dem Zustand abgefragt wird. Das Gegenstück stellt `set` vom Typ `Ident` → `Nat` → `StateTransition State` () dar. Hier wird der Zustand verändert, indem der Wert der durch den übergebenen Bezeichner indizierten Variablen überschrieben wird. Das formale Resultat dieser Operation ist nicht relevant, häufig wird der Typ () (gesprochen Unit) und sein einziger, gleichnamiger Wert () verwendet.

Mit diesen beiden Funktionen kann die tatsächliche, technische Repräsentation der Variablen im Zustandsdatentyp `State` verborgen werden. Es ist daher aus softwaretechnischer Sicht wünschenswert, auf diese Weise vorzugehen. Für die Terminierungsanalyse ist die Art der Repräsentation allerdings relevant. Im Folgenden werden zwei Möglichkeiten zur Realisierung der Abbildung von Bezeichnern auf Werte betrachtet. Bereits für das bekannte Countdown-Programm ergeben sich hier neue Schwierigkeiten.

Die erste Variante ist die Realisierung der Variablenbelegung als Funktion. Aus Sicht der Terminierungsanalyse bietet sich dieses Vorgehen allerdings nicht an. In Kapitel 5 wurde das Prädikat U eingeführt, um Ausdrücke zu beschreiben, deren Wurzelsymbol eine Variable ist, die eine Funktion repräsentiert. Diese Ausdrücke wurden in der Auswertung durch frische Variablen ersetzt, um alle möglichen Resultate der Funktionsauswertung zu repräsentieren. Die Behandlung von Funktionen als Daten ist daher prinzipiell möglich, jedoch wird durch die in dem verwendeten Verfahren durchgeführte Ersetzung jede Information, die in den Funktionen enthalten ist, entfernt. Eine Terminierungsanalyse kann dann nur noch gelingen, wenn die Belegung der Variablen vollständig irrelevant ist.

Wir betrachten daher die Modellierung der Belegung als Tabelle. Bei dieser Form werden Paare aus Variablenbezeichnern und Werten in einer Liste abgelegt. Dieses Verfahren besticht nicht durch Effizienz, jedoch ist das Ziel der Übersetzung auch kein effizientes, sondern ein möglichst einfaches Programm, um die Terminierungsanalyse nicht zu erschweren.

Mit demselben Argument verzichten wir darauf, das Countdown-Programm tatsächlich in monadischer Form zu notieren. Stattdessen analysieren wir ein einfacheres Programm, das dieselben primitiven Funktionen einsetzt. Zusätzlich motiviert wird dieser Schritt durch das Problem, dass die Transitionsmonade Funktionen als Daten beinhaltet. Wie bereits bei der Repräsentation der Variablenbelegung mit Funktionen erklärt wurde, sind hierdurch Schwierigkeiten zu erwarten. Die im Folgenden gezeigten Programme verwenden jedoch die primitiven Funktionen, mit denen eine monadische Variante konstruiert werden müsste. Daher sind die gewonnen Erkenntnisse auch für die monadische Form relevant.

Im Quelltext 6.6 wird eine Implementierung der Repräsentation als Tabelle gezeigt. Der Vollständigkeit halber enthält der Quelltext auch Definitionen für den Listentyp `List` und den Variablentyp `Pair`, sowie eine Definition der Gleichheit auf den natürlichen Zahlen. Als Variablenbezeichner werden die natürlichen Zahlen verwendet. Die Funktion `get` realisiert eine Suche in der Variablentabelle. Hierbei kann es zu dem Fehlerausdruck `get Nil` kommen. Auf eine Fehlerbehandlung wird jedoch

Quelltextauszug 6.6: Repräsentation der Variablenbelegung als Tabelle

```
data Nat = Succ Nat | Zero

equal :: Nat -> Nat -> Bool
equal Zero           Zero = True
equal (Succ x)       Zero = False
equal Zero      (Succ x) = False
equal (Succ x) (Succ y) = equal x y

type Ident = Nat

data Pair = Pair Ident Nat

data List a = Cons a (List a) | Nil

type State = List Pair

get :: Ident -> State -> Nat
get x (Cons (Pair y z) rest)
    = select (equal x y) z (get x rest)

set :: Ident -> Nat -> State -> State
set x val Nil = Cons (Pair x val) Nil
set x val (Cons (Pair y v) rest)
    = select (equal x y)
          (Cons (Pair x val) rest)
          (Cons (Pair y v) (set x val rest))

select :: Bool -> a -> a -> a
select True a b = a
select False a b = b
```

Quelltextauszug 6.7: Countdown als Variablenproblem

```
notZero (Succ x) = True
notZero Zero     = False

dec Zero     = Zero
dec (Succ x) = x

while0 pred trans state
     = select (pred state)
           (while1 pred trans (trans state))
           state
while1 pred trans state
     = select (pred state)
           (while0 pred trans (trans state))
           state

notZeroS :: Ident -> State -> Bool
notZeroS x state = notZero (get x state)

decS :: Ident -> State -> State
decS x state = set x (dec (get x state) ) state

countdown x state = while0 (notZeroS x) (decS x) state
```

zugunsten der Einfachheit verzichtet. In `set` wurde die bereits aus Abschnitt 6.1 bekannte Funktion `select` verwendet.

Auf den Grundlagen dieser Typen und Funktionen kann nun das Countdown-Programm wie in Quelltext 6.7 definiert werden. Auf die Angabe der nur leicht veränderten Typen der Kombinatoren und Transitionen wurde verzichtet. Der Kombinator `while` wurde bereits ausgerollt.[2] Die Funktionen `decS` und `notZeroS` sind die zustandsbasierten Gegenstücke zu `dec` und `notZero`. Ihnen kann ein Variablenbezeichner übergeben werden, die zugehörige Funktion wird dann auf der entsprechenden Variablen ausgeführt.

Für dieses Programm ergibt sich ein wesentlich größerer Terminierungsgraph, bestehend aus 140 Knoten und 160 Kanten, die sich in fünf starke Zusammenhangskomponenten gliedern. Aus einer der Zusammenhangskomponenten entsteht ein Abhängigkeitspaarproblem, das nicht finit ist. Damit kann APROVE die Terminierung nach Theorem 5.24 nicht beweisen.

[2] Wird darauf verzichtet, den Kombinator `while` auszurollen, entsteht ein Graph, aus dem ein nicht finites Problem hervorgeht, welches dem nicht finiten Problem aus Abschnitt 6.1 ähnelt.

Ein Terminierungsargument für jedes Countdown-Programm muss zwangsläufig berücksichtigen, dass die dekrementierte Variable auch die Variable aus dem Vergleich ist. Um diesen Zusammenhang in der vorgestellten Implementierung zu identifizieren, reicht es nicht, zu erkennen, dass `notZeroS` und `decS` dasselbe Argument erhalten. Es ist auch erforderlich, festzustellen, dass sich der Teil des Zustands, der die Variablenbezeichner beinhaltet, unveränderlich ist, während der zweite Teil veränderlich ist. Aufgrund dieser Änderung ergibt sich formal ein neuer Zustand. Die beschriebene Methode betreibt jedoch keine Analyse der Datenstrukturen und kann daher das Terminierungsargument nicht hervorbringen.

Das Verfahren ist auf die Analyse von Programmen ausgelegt, die auf algebraischen Datenstrukturen arbeiten. Während das vorgestellte Programm formal nur algebraische Datenstrukturen beinhaltet, ist das Terminierungsargument nur identifizierbar in der Interpretation der Datenstruktur als Variablenbelegung. Dieses Problem müssen alle Programme teilen, die Variablenbelegungen in algebraischen Datenstrukturen repräsentieren. Im folgenden Abschnitt wird eine Methode vorgestellt, die Variablen als Funktionsparameter und die Belegung als Funktionsargumente darstellt.

6.3 Übersetzung auf Basis der SSA-CPS-Korrespondenz

Ein alternative Möglichkeit, imperative Sprachen nach Haskell zu übersetzen, besteht in der Ausnutzung der Korrespondenz der statischen Einmalzuweisungsform (kurz SSA-Form, siehe [AWZ88, RWZ88]) und dem sogenannten Continuation Passing Style (kurz CPS, siehe [SW00]). Auf diese Korrespondenz hat zuerst Richard Kelsey in [Kel95] hingewiesen. Beide Formen werden in Übersetzern als Zwischenstufen in der Programmoptimierung genutzt. Während die SSA-Form wie im vorhergehenden Kapitel beschrieben für optimierende Übersetzer imperativer Sprachen interessant ist, so ist die CPS-Form für optimierende Übersetzer funktionaler Sprachen interessant.

Kelsey hat gezeigt, wie sich Programme aus der einen Form in die andere Form überführen lassen. Die Transformation wird auf syntaktischer Ebene definiert – eine aufwendige Datenflussanalyse ist nicht erforderlich. Genau hierin bestand Kelseys Motivation. Durch die Transformation eines funktionalen Programms in die SSA-Form lassen sich laut Kelsey viele Optimierungen durchführen, ohne aufwendige Datenflussanalysen vornehmen zu müssen. Durch die Rückübersetzung des optimierten

SSA-Programms in ein CPS-Programm ist diese Optimierungsmethode modular in bestehende Übersetzer auf Basis der CPS-Form integrierbar.

Während sich alle Programme in SSA-Form in die CPS-Form überführen lassen, gilt die Umkehrung nicht. Es gibt Programme in CPS-Form die nicht in die SSA-Form überführt werden können. Die beiden Formen sind also nicht vollständig äquivalent. Kelsey stellt allerdings fest, dass derartige Programme durch die üblichen Übersetzer funktionaler Sprachen nicht erzeugt werden. Im Folgenden wird nun lediglich der Rückübersetzungsschritt durchgeführt und so aus dem imperativen SSA Programm ein Haskell-Programm erzeugt. Die Asymmetrie der Umwandelbarkeit stellt daher keine Einschränkung dar.

Das Verfahren wird in diesem Abschnitt anhand der Sprache Jinja konkretisiert. Die Sprache Jinja wurde von Klein und Nipkow in [KN06] eingeführt. Jinja ist bewusst an Java [GJS+13] angelehnt. Es wurden verschiedene Bemühungen zur Formalisierung von Teilaspekten von Java-ähnlichen Sprachen zusammengeführt. Um sicherzustellen, dass durch die Zusammenführung keine unbemerkten Widersprüche eingeführt werden, wurde die operationelle Semantik von Jinja mit dem automatisierten Theorembeweiser Isabelle/HOL [NPW02] formalisiert. Auf eine detailliertere Einführung wird aufgrund der Nähe von Jinja zu Java verzichtet.

Im Folgenden werden mögliche Modellierungen von Jinjas Objektsystem in Haskell besprochen. Im Rahmen dieser Erläuterungen werden auf Unterschiede zwischen den Sprachen hingewiesen, wenn diese von Interesse sind. Es sei vorab darauf hingewiesen, dass es in Jinja keinen parametrischen Polymorphismus gibt, wie ihn Java mit Generics [NW06] bietet. Analog zu Java existieren Mechanismen zur Ausnahmebehandlung in Jinja. Diese werden in der vorliegenden Arbeit jedoch nicht betrachtet. Im Anschluss an die Modellierung des Objektsystems wird die SSA-Form und ihre Übersetzung in ein rein funktionales Programm beschrieben.

6.3.1 Modellierung des Zustandsraums

Bevor die konkrete Methode der Übersetzung beschrieben wird, wird die Modellierung von Jinjas Zustandsraum in Haskell beschrieben. In Jinja besteht der Zustandsraum aus zwei Komponenten: dem Heap und der Menge der lokalen Variablen. Beide werden in [KN06] als partielle Funktionen aufgefasst. Die Heap-Funktion bildet Referenz-Werte auf Objekte ab, während eine weitere Funktion (in [KN06] „store“ genannt) die Bezeichner für lokale Variablen auf Werte abbildet. Für beide gilt, dass sie für nicht vorhandene Referenzen oder Bezeichner nicht definiert sind. In der Praxis spielt die partielle Natur der Funktion der lokalen Variable keine Rolle, da

der Übersetzer undefinierte Variablen bemängelt und kein Programm erzeugt. Die partielle Definition ist eher den Limitierungen des Isabelle/HOL System geschuldet. Obwohl Jinja keine Möglichkeit vorsieht, Referenzwerte zu manipulieren, wie es zum Beispiel die Zeigerarithmetik in C oder C++ zulässt, kann es bei der Auflösung von Referenzen dennoch zu undefinierten Situationen kommen und zwar, wenn es sich um die `null`-Referenz handelt. Für die Realisierung in Haskell stellen partielle Funktionen keine Schwierigkeit dar. Ähnlich wie in Isabelle/HOL werden sie mit einem Optionstyp realisiert. In Haskell handelt es sich dabei um `Maybe`.

Um den Zustandsraum vollständig zu modellieren, müssen die Datentypen repräsentiert werden. Die ganzen Zahlen und booleschen Werte stellen die einzigen primitiven Datentypen `int` und `boolean` dar. Hinzukommen zwei ausgezeichnete Werte, die `null`-Referenz und der Einheitswert Unit. Letzterer stellt das formale Ergebnis von Methoden des Rückgabetyps `void` dar. Ihre Repräsentation bereitet in Haskell keine Schwierigkeiten, im Gegensatz zur Wahl einer geeigneten Darstellung der Klassen. Wie in [KN06] beschrieben, ist ein Jinja-Programm eine Sammlung von Klassen, die in einer Subtypenbeziehung zueinander stehen. Naheliegend scheint die Definition eines eigenen Haskell-Datentyp für jede Jinja-Klasse. Jedoch lässt sich die Subtypenbeziehung auf diese Weise nicht ausdrücken.

Im Folgenden werden drei verschiedene Modellierungen vorgestellt. In den ersten beiden Ansätzen wird versucht, Jinja-Klassen mit den in Haskell definierten Konzepten eines Datentyps beziehungsweise einer Typklasse zu modellieren. Beide Ansätze zielen darauf ab, die statischen Typinformationen aus Jinja in Haskell als statische Typen darzustellen. Die Alternative besteht darin, die Typinformationen zu reifizieren, und sie so zur Laufzeit verfügbar zu machen. Damit ergibt sich allerdings auch, dass diese bei der Terminierungsanalyse als Daten auftreten und berücksichtigt werden müssen. Es wird sich herausstellen, dass die Typsysteme der beiden Sprachen zu unterschiedlich sind und eine Reifikation der Typen in Haskell unumgänglich ist. Im letzten Ansatz werden daher alle Klassen durch denselben Datentyp repräsentiert.

Klassen als algebraische Datentypen Es wird mit der Modellierung von Klassen als einfachen Haskell-Datentypen begonnen. Das folgende Beispiel soll die Problematik der Subtypenbeziehung verdeutlichen. Gegeben sei ein Jinja-Programm bestehend aus zwei Klassen A und B mit A der Oberklasse von B. Die Klasse A beinhalte ferner ein Attribut vom Typ `int` und B zusätzlich ein Attribut vom Typ `String`. Die einfachste Modellierung dieser einfachen Typen findet sich in Quelltext 6.8.

Der Typ B ist ein Produkttyp aus dem nur in B definierten Attribut vom Typ `Bool` und einem Wert vom Typ A. Auf diese Weise wird deutlich, dass die Struktur

Quelltextauszug 6.8: Algebraische Datentypen für jede Klasse

```
data A = A Int
data B = B A Bool
```

Quelltextauszug 6.9: Die Methode m als Funktion in Haskell

```
m :: A -> A -> Int
m (A x) (A y) = x + y
```

des Typen B von der Struktur des Typen A abhängt. Sei nun m eine Methode, die ein Argument vom Typ A erwartet.

Gemäß der Subtypenbeziehung kann die Methode m in einem Jinja-Programm mit einer Instanz b der Klasse B aufgerufen werden. Dies gilt nicht für die Funktion m, die in Quelltext 6.9 definiert worden ist. Eine Realisierung der Klassen mit disjunkten Haskell-Typen ist daher so nicht möglich.

In dem gegebenen Beispiel müsste für jede Typkombination eine eigene Funktion definiert werden. Dies führt im Allgemeinen zu einem exponentiellen Anstieg der Funktionsdefinitionen. Verfügt eine Klasse X über eine Methode mit n Parametern vom Typ X und hat ferner k echte Unterklassen, so ergeben sich $(k+1)^{(n+1)}$ Funktionsdefinitionen. Dabei wird berücksichtigt, dass nicht nur echte Unterklassen von X sondern auch Instanzen von X selbst als Argument in Frage kommen und das Objekt, auf dem die Methode aufgerufen wird, als weiteres Argument gezählt werden muss. Da Haskell keine Überladung für Funktionen vorsieht, müssten die Typinformation in die Namensgebung einfließen.

Beim Aufruf einer Methode müsste nun jeweils die richtige Funktion ausgewählt werden. Dies kann allerdings nicht statisch zur Zeit der Übersetzung geschehen, da Methoden Instanzen von Unterklassen zurückgeben dürfen, also der Laufzeittyp nicht anhand der statischen Deklaration ersichtlich ist. Es wäre also eine Reifikation des Typs und eine Auswahl zur Laufzeit notwendig. Dies erscheint wenig elegant, da die Modellierung darauf abzielte, Typinformationen aus Jinja nicht zu reifizieren, sondern in Haskell auszudrücken.

Klassen als Typklassen Neben den algebraischen Datentypen bietet Haskell Typklassen, die sich in einer Hierarchie anordnen lassen. In diesem Ansatz wird für jede Jinja-Klasse eine Haskell-Typklasse und einen algebraischen Datentyp definiert. Der Name der Jinja-Klasse dient als Name der Typklasse, während der Name des Datentyps zusätzlich um das Suffix `Impl` ergänzt wird. Der algebraische Datentyp wird

Quelltextauszug 6.10: Der Typklassen-Ansatz

```
class A a where
    readX' :: a -> Int
    writeX' :: a -> Int -> a
    m :: (A a') => a -> a' -> Int

class A b => B b where
    readT' :: b -> Bool
    writeT' :: b -> Bool -> b

data AImpl = AImpl Int

instance A AImpl where
    readX' (AImpl x) = x
    writeX' _ newX = AImpl newX
    m (AImpl x) a = x + readX' a

data BImpl = BImpl AImpl Bool

instance A BImpl where
    readX' (BImpl super t) = readX' super
    writeX' (BImpl super t) x = BImpl (writeX' super x) t
    m (BImpl super b) = m super

instance B BImpl where
    readT' (BImpl _ t) = t
    writeT' (BImpl super _) newT = BImpl super newT
```

dann Mitglied der Typklasse. Durch die hierarchische Ordnung der Typklassen muss der Datentyp auch Mitglied der zu den Oberklassen gehörigen Typklassen sein.

Da Haskells Typklassen lediglich Funktionen und keine Daten beinhalten, muss der Zugriff auf die Attribute eines Objekts ebenfalls über geeignete Zugriffsfunktionen realisiert werden. Diese entsprechen den mit Hilfe der Record-Syntax in Haskell erstellten Funktionen, welche allerdings nur auf Datentypen und nicht in Klassen automatisch definiert werden. Die Bezeichner der Funktionen werden mit dem Präfix `read` (bzw. `write` für den schreibenden Zugriff) und dem Suffix ' zu dem Attributnamen gewählt.[3] Das Attribut vom Typ `int` in A soll den Namen x tragen. In der Klasse B sei das `boolean`-Attribut unter dem Namen t bekannt.

[3] Das Zeichen ' darf in Java nicht als Teil eines Bezeichners verwendet werden und soll intuitiv vermitteln, dass es sich hierbei nicht um eine Methode handelt. Da keine Beschränkungen für die Bezeichner in Jinja definiert sind, ist dieser Schritt kein Schutz vor Namenskollisionen.

Quelltextauszug 6.11: Eine nicht vollständige polymorphe Implementierung

```
class A a where
    m :: a -> Int
    n :: a -> Int

class A b => B b where
    -- empty

data AImpl = AImpl Int

instance A AImpl where
    m this = n this
    n this = 42

data BImpl = BImpl AImpl Bool

instance A BImpl where
    m (BImpl super cs) = m super
    n this = 23

instance B BImpl where
    -- empty
```

In Quelltextauszug 6.10 wird gezeigt, wie sich die durch Vererbung entstehende Wiederverwendung von Programmteilen nach Haskell übertragen lässt. Die Funktion m in der Instanziierung der Typklasse A für den Typ `BImpl` verwendet direkt die Implementierung des eingebetteten Superelements. Haskells Typsystem löst hier in die richtige Typklasseninstanz auf. Dies ist jedoch nur möglich, da bisher das polymorphe Verhalten von Jinja nicht relevant war. Ein Problem tritt allerdings auf, falls die Methode m eine weitere Methode n aufruft und diese in B im Gegensatz zu m überschrieben wird. Die Jinja-Spezifikation verlangt, dass ein Aufruf von m auf einer Instanz von B zu einem Aufruf der in B definierten Variante von n führt. Das Beispiel in Quelltext 6.11 zeigt das Problem. Aus Gründen der Übersichtlichkeit wurden die Attributzugriffsfunktionen nicht aufgelistet und die Methoden m und n um die Parameter gekürzt. Beides ist zur Demonstration des Effekts nicht notwendig.

Wie bereits in Quelltext 6.10 wurde m definiert, indem auf die Implementierung von m in der Typklasseninstanz von `AImpl` zurückgegriffen wurde. Da m für `AImpl` allerdings n aus `AImpl` ausführt, wurde die Polymorphie nicht korrekt realisiert.

Quelltextauszug 6.12: Korrekte Polymorphieimplementierung

```
class A a where
    m :: a -> Int
    n :: a -> Int

class A b => B b where
    -- empty

data AImpl = AImpl Int

instance A AImpl where
    m this = n this
    n this = 42

data BImpl = BImpl AImpl Bool

instance A BImpl where
    m this = n this
    n this = 23
```

Die Lösung besteht darin, in der Instanziierung der Typklasse *A* durch `BImpl` alle Methoden exakt so zu implementieren, wie es für `AImpl` erfolgt ist. In Quelltext 6.12 ist die korrekte Implementierung wiedergegeben. Erneut ist eine Duplizierung von Quelltext erforderlich, wenn auch nur noch einmal pro Methode pro erbende Klasse.

Bisher wurden in den Beispielen ausschließlich primitive Typen als Rückgabewerte in den Methoden gewählt. Im Gegensatz zum auf algebraischen Datentypen aufbauenden Ansatz können durch Typklassen Parameter- und Rückgabetypen polymorph gestaltet werden. Als Rückgabetyp kann eine Typvariable mit entsprechender Einschränkung eingesetzt werden. Quelltext 6.13 demonstriert dies für eine Methode *k* in der Klasse *A*, die eine Instanz der Klasse *A* liefert. Polymorphe Parametertypen können auf analoge Weise realisiert werden. Allerdings ist es notwendig, für jeden Parameter und den Rückgabetyp eine eigene Typvariable zu verwenden, wie es in Quelltext 6.14 zu sehen ist.

Diese Form der Modellierung erweist sich in einem Punkt als problematisch. Eine Funktion ist nun zwar in der gewünschten Form auch mit Werten von Subtypen auswertbar. Jedoch ermöglicht Jinja das Überschreiben von Methoden und zwar in einer sehr rudimentären Weise. Während Java lediglich geringe Veränderungen beim Überschreiben einer Methode zulässt (wie z.B. kovariante Rückgabetypen bzw. Einschränkungen der Sichtbarkeit) erlaubt Jinja jegliche Modifikation und verlangt

Quelltextauszug 6.13: Rückgabewerte im Typklassenansatz

```
class A a where
    m :: (A a') a -> a'

class A b => B b where
    -- empty

data AImpl = AImpl Int

instance A AImpl where
    m (AImpl x) = AImpl x

data BImpl = BImpl AImpl Bool

instance A BImpl where
    m (BImpl super t) = BImpl super t

instance B BImpl where
    -- empty
```

Quelltextauszug 6.14: Rückgabewerte kombiniert mit polymorphen Parametern

```
class A a where
    m :: (A a1, A a2) a -> a1 -> a2
```

lediglich den gleichen Bezeichner.

Eine Methode der Superklasse kann daher auch verdeckt werden, wenn eine Methode gleichen Namens mit einer unterschiedlichen Anzahl von Argumenten oder unterschiedlichen Typs definiert wird. Gleiches gilt für den Rückgabetyp. Bei der Übersetzung in ein Haskell-Programm muss hier für jede Überschreibung, die nicht mit dem Rückgabe- und allen Parametertypen konform ist, ein anderer Name gewählt werden. Ferner kann erst zur Laufzeit die tatsächlich auszuführende Methode identifiziert werden. Dies erfordert abermals eine Reifikation des Typs einer Instanz.

Uniforme Objekt-Modellierung In den vorangegangen Abschnitten ist klar geworden, dass Informationen über den Typ von Objekten zur Laufzeit verfügbar sein müssen. Ferner haben die Ansätze, die eine Korrespondenz zwischen Jinja-Klassen und Haskell-Typen oder Haskell-Typklassen herzustellen versuchten, eine Duplizierung von Quelltext notwendig gemacht. Duplizierter Quelltext ist als Indikator für schlecht wartbare Software bekannt, siehe zum Beispiel [Fow99]. Die Argumentation geht dahin, dass Änderungen am Quelltext nun an zwei Stellen parallel vorgenommen werden müssen. Damit betrifft dies allerdings nur Eingabequelltext der durch Menschen manipuliert wird, was für generierten Quelltext nicht zutrifft. Dennoch stellt duplizierter Quelltext auch hier ein Problem dar. Dies liegt darin, dass die erhöhte Komplexität auch die Terminierungsanalyse erschwert. Der folgende Ansatz vermeidet Duplizierung und nutzt dazu die Reifikation von Typen aus.

In der uniformen Objekt-Modellierung gibt es nur einen Haskell-Typ für Objekte namens `Object`. Alle Objektinstanzen sind von diesem Typ, unabhängig von ihrem Jinja-Typ. Der Datentyp `Object` beinhaltet Informationen über den Jinja-Typ und Informationen über die Zuordnung von Attributnamen zu Attributwerten. Haskells Typprüfung kann daher fehlerhafte Programme nicht mehr identifizieren. Allerdings kann durch eine statische Typprüfung des Jinja-Übersetzers die Typkorrektheit sichergestellt werden. Laufzeittypprüfungen können damit zu großen Teilen entfallen und stellen somit keinen Ballast für die Terminierungsanalyse dar. Die einzige Ausnahme bilden die expliziten Typkonvertierungen. Quelltext 6.15 enthält die nötigen Datenstruktur.

Für Bezeichner wird das Typsynonym `Identifier` verwendet. Objekte werden durch `Object` repräsentiert. Sie kennen ihren Typ sowie eine Liste von Attributen, die lediglich Tripel aus einem Bezeichner, einem Typ und einem Wert dieses Typs sind. In Jinja sind Vorgabewerte (`0`, `false` und `null`) für nicht initialisierte Attribute definiert. Es ist daher nicht notwendig, undefinierte Werte zu repräsentieren. Der Typ eines Objekts wird durch `ClassType` beschrieben, das aus dem Bezeich-

Quelltextauszug 6.15: Uniforme Objektmodellierung

```
type Identifier = String

data Object = Object ClassType [Attribute]

data Attribute = Attribute Identifier Type Value

data ClassType = ClassType Identifier [Method] SuperClass

data SuperClass = Extends ClassType | Root

data Method = Method Identifier Type [Parameter] Transition

data Heap = [(Int, Object)]

type Transition = [Argument] -> Heap -> (Value, Heap)

data Parameter = Parameter Identifier Type

data Argument = Argument Identifier Value

data Type
    = JBool
    | JInt
    | JVoid
    | JNull
    | JRef ClassType

data Value
   = VInt Int
   | VBool Bool
   | VRef Int
   | VUnit
   | VNull
```

ner der repräsentierten Jinja-Klasse, einer Liste von Methoden und einer möglichen Superklasse besteht. Dabei zeigt der Wert `Root` an, dass die Spitze der Vererbungshierarchie erreicht ist. Die Methoden tragen Informationen über ihren Namen, ihren Rückgabewert, ihre formalen Parameter und eine Funktion, die den Zustandsübergang der Methodenausführung beschreibt. Für diesen Zustandsübergang wird das Typsynonym `Transition` verwendet. Funktionen vom Typ `Transition` überführen einen Heap und eventuell vorhandene Argumente in einen modifizierten Heap sowie das Ergebnis der Methode. Bei Methoden ohne Rückgabe wird der spezielle Wert `VUnit` als formales Ergebnis verwendet.

6.3.2 Realisierung der Zustandsübergänge

Mit der uniformen Objektmodellierung ist die Repräsentation des Zustandsraums vollständig bis auf die Realisierung der konkreten Zustandsübergänge, die die Methoden der Objekte beschreiben. Jinja ist eine Ausdruck-basierte Sprache, Anweisungen sind ihr fremd. Die Methoden der einzelnen Klassen bestehen aus einem Ausdruck. Dies gelingt, da zwei Ausdrücke mit dem Sequenzierungsoperator ; zusammengeführt werden können. Das formale Resultat eines solchen Sequenzierungsausdrucks ist der Wert des Einheitstypen.

Um die Methoden zu beschreiben kann prinzipiell der monadische Ansatz aus Abschnitt 6.2 verwendet werden, jedoch führt dies zu den in demselben Abschnitt aufgeführten Problemen. Stattdessen soll hier eine alternative Methode vorgestellt werden, die die SSA-CPS Korrespondenz geschickt ausnutzt. Die grundsätzliche Machbarkeit ergibt sich bereits aus der Arbeit [Kel95] von Kelsey. Appel demonstriert in [App98], dass strukturierte Programme in SSA-Form ohne Sprünge, nach demselben Schema in funktionale Programme umgewandelt werden können. Da keine Sprünge vorhanden sind, kann auf die explizite Übergabe von Continuations verzichtet werden.

Wir beginnen mit der Darstellung der SSA-Form. In einem Programm in SSA-Form findet sich für jede Variable nur eine Zuweisung. Diese Zuweisung darf allerdings mehrfach ausgeführt werden, zum Beispiel wenn die Zuweisung im Körper einer Schleife steht. Cytron et al. beschreiben in [CFR+91] einen Algorithmus, mit dem imperative Programme in die SSA-Form überführt werden können. Der Algorithmus arbeitet auf dem Kontrollflussgraphen [All70] des imperativen Programms. Für Programme, deren Kontrollflussgraph keine Verzweigung aufweist, ist intuitiv klar, dass eine solche Überführung möglich ist. Anstelle einer zweiten Zuweisung an die Variable x_i wird eine neue Variable x_{i+1} eingeführt und alle folgenden Zugriffe auf

Quelltextauszug 6.16: Countdown im Continuation Passing Stil

```
countdown :: UnsignedInt -> UnsignedInt
countdown x = select (notZero x) (decAndRepeat x) x

decAndRepeat :: UnsignedInt -> UnsignedInt
decAndRepeat x = countdown (dec x)
```

die originale Variable durch Zugriffe auf x_{i+1} ersetzt. Verzweigung im Kontrollflussgraphen werden problematisch, wenn ein Knoten A mehrere Eingangskanten besitzt und die Variable x in den verschiedenen Vorgängerknoten B und C durch verschiedene Variablen x_B und x_C ersetzt wurde. Formal beginnt Knoten A daher mit einer Zuweisung der Form $x_A \leftarrow \phi(x_B, x_C)$. Der Ausdruck $\phi(x_1, \ldots, x_n)$ in einem Knoten j ergibt dabei x_i genau dann, wenn der Knoten j über die Kante vom Vorgängerknoten i erreicht wurde. Nicht in allen Knoten muss jedoch ein solcher ϕ-Ausdruck platziert werden. Der Algorithmus von Cytron et al. produziert die minimale Anzahl von ϕ-Ausdrücken.

Das folgende Vorgehen entspricht dem in [App98] skizzierten Verfahren. Den Ausgangspunkt für diese Übersetzung bildet der Kontrollflussgraph der SSA-Form des Eingabeprogramms. Für jeden Knoten im Kontrollflussgraphen wird eine Funktion erzeugt, die für alle Variablen, deren Wert durch ϕ-Ausdrücke bestimmt wird, einen Parameter erhält. Für das Beispiel mit den Knoten A, B und C ergibt sich eine Funktion für Knoten A, die einen Parameter x_A erhält. Die Knoten B und C, werden so transformiert, dass sie die zu Knoten A gehörige Funktion aufrufen und dabei die Variable x_B oder x_C übergeben. Auf diese Weise werden die ϕ-Funktionen aufgelöst. Da in der SSA-Form keine destruktiven Zuweisungen mehr existieren, lassen sich die Berechnungen leicht in Haskell zum Beispiel mit `let`-Ausdrücken beschreiben.

Als Beispiel soll das Countdown-Programm dienen. Der Kontrollflussgraph enthält die drei Knoten. Knoten A repräsentiert den Kopf der Schleife, Knoten B den Körper. Knoten C ist der Ausgangsknoten. Während von B nur eine Kante zu A führt, führt von A eine Kante zu B und eine zu C. Ist der Wert von x bei 0 angekommen, so wird die Kante zu C gewählt. Ansonsten führt A wieder zu B, indem x dekrementiert wird. Es ergibt sich das Programm aus Quelltext 6.16.

Im Folgenden wird behandelt, wie sich die so übersetzten Funktionen in die Modellierung einbinden lassen.

Quelltextauszug 6.17: Strukturierung durch lokale Funktionen

```
classA_methodM x y = f0 x y
   where f0 x0 y0 = ...
         f1 x1 = ...
         f2 y2 = ...
```

6.3.3 Einbindung der Zustandsübergänge

Wie im vorigen Abschnitt beschrieben, wird der zu einer Methode im Jinja-Eingabeprogramm gehörige Zustandsübergang durch mehrere Funktionen im Haskell-Programm implementiert. Diese lassen sich durch eine Funktion verbergen, die die Schnittstelle für das restliche Programm darstellt. Diese Funktionen heißen im Folgenden Rumpffunktionen. Sie tragen denselben Namen wie die Eingaberoutinen. Quelltext 6.17 zeigt eine Rumpffunktion, die einer Methode `methodM` aus einer Klasse `ClassA` entspricht. Die Knotenfunktion sind als lokale Funktionen `f0`, `f1` und `f2` angedeutet. Aus der Kombination von Klassen- und Methodenname ergeben sich mehr oder weniger natürliche Präfixe.

Bei den Knoten repräsentierenden Funktionen (im Folgenden werden sie als Knotenfunktionen bezeichnet) handelt es sich um Hilfsfunktionen, welche mit den einzelnen Knoten des Kontrollflussgraphen korrespondieren. Es ergibt sich keine Notwendigkeit, Knotenfunktionen außerhalb der zu übersetzenden Routine namentlich zu referenzieren. Dies lässt sich in Haskell ausdrücken, indem diese Hilfsfunktionen als lokale Funktionen definiert werden. Auf diese Weise werden Kollisionen im Namensraum des Haskell-Moduls vermieden. Im Namensraum der Rumpffunktion müssen die einzelnen Knotenfunktionen unterschiedlich benannt werden. Da den Knoten des Kontrollflussgraphen jedoch keine Bezeichner aus dem Eingabeprogramm zugeordnet werden können, muss eine mehr oder weniger künstliche Benennung verwendet werden. Es bieten sich entweder Zeilennummern aus dem Eingabeprogramm oder sich aus Breiten- bzw. Tiefensuche im Kontrollflussgraphen ergebende Nummerierungen an. Für die Terminierungsanalyse mit AProVE bietet die Definition der Knotenfunktionen als lokale Funktionen weder einen Vorteil noch einen Nachteil, da das AProVE-Werkzeug, wie bereits beschrieben, das Eingabeprogramm zunächst in eine Teilmenge von Haskell überführt, in der es keine lokalen Funktionen mehr gibt.

Die Rumpffunktion erhält neben den Parametern, die auch die Methode erhält, noch zwei zusätzliche Parameter. Diese repräsentieren den Heap und die Selbstreferenz `this`. Quelltext 6.18 zeigt die Deklaration der Rumpffunktion für eine einstellige

Quelltextauszug 6.18: Rumpffunktion mit Heap

```
methodM :: Heap -> Ref -> Ref -> (Int, Heap)
methodM heap this x = f0
```

Methode. Bei dem ersten Referenzparameter handelt es sich um die Selbstreferenz, beim zweiten um den Parameter. Die Bezeichner auf der linken Seite der Funktionsgleichung deuten dies an. Der Ergebnistyp der Funktion ist ebenfalls erweitert. Es reicht nicht aus, das berechnete Resultat zurückzugeben, sondern es muss auch noch ein Heap zurückgegeben werden, der eventuelle Veränderungen in dem dynamischen Speicherbereich repräsentiert. Die Referenzparameter sind nicht getypt. Gleiches würde auch für Rückgabetypen gelten, wenn es sich nicht um `int`, `boolean` oder `void` handelt.

Die Rumpffunktion ruft lediglich die erste Knotenfunktion auf. Dabei werden nur änderbare Parameter weitergegeben. Zum Beispiel lässt sich die Selbstreferenz nicht ändern und wird daher nicht übergeben. Die Parameter dieser Knotenfunktionen enthalten die lokalen Variablen des Eingabeprogramms. Die Auswertung einer solchen Funktion entspricht dem Wechsel von einem Knoten im Kontrollflussgraphen zum nächsten und gleichzeitig der Auswertung der ϕ-Ausdrücke in diesem.

6.3.4 Die Heap-Komponente

In der bisherigen Beschreibung wurde die Heap-Komponente übergangen. Der Heap wird in [KN06] als eine partielle Funktion von Referenzen zu Objekten realisiert. Mittels Referenzen können Objekte auf dem Heap in beliebigen Graphen angeordnet werden. Im Vergleich dazu erzeugen Haskells algebraische Datentypen gerichtete azyklische Graphen. Jede Haskell-Übersetzung eines Jinja-Programms muss daher potentiell zyklische Graphen darstellen und kann dazu nur baumartige Strukturen verwenden.

Klassische Graphdarstellungen erfolgen mit Adjazenzmatrizen oder Adjazenzlisten, ein neuerer Ansatz in der funktionalen Programmierung sind induktive Graphen [Erw01]. Alle Darstellungen verwenden eine symbolische Repräsentation der Knoten, um die Kanten eines Graphen zu beschreiben. Eine direkte Darstellung von Knoten (und nicht Repräsentanten der Knoten) führt zu unendlichen Datenstrukturen. Quelltext 6.19 demonstriert dies am Beispiel der Adjazenzlistendarstellung für einen Graphen, der lediglich aus einem Knoten und einer reflexiven Kante besteht. Für grundlegende Graphalgorithmen wichtige Operationen wie die Gleichheit auf

Quelltextauszug 6.19: Graph mit reflexiver Kante in Adjazenzlistendarstellung

```
data Node = Node [Node]

a = Node [a]
```

Knoten können daher nicht terminierend definiert werden.

Für die Darstellung des Objektgraphen stellen die Referenzen eine natürliche Repräsentation der Objekte dar. Die Zuordnung der Referenzen zu den Objekten stellt ein ähnliches Problem dar, wie die Zuordnung von Variablen zu ihren Werten. Analog zu den Werten der Variablen sind auch die Objekte hinter den Referenzen veränderlich. In Abschnitt 6.2 wurde bereits erläutert, warum dies ein schwieriges Problem für die Terminierungsanalyse darstellt. Es wurde in diesem Abschnitt gelöst, indem die Bezeichner der Variablen zu Funktionsparametern und die Werte der Variablen zu Funktionsargumenten wurden. Dieser Ansatz kann jedoch nicht auf die Referenz-Objekt-Zuordnung übertragen werden. Der Unterschied zwischen den beiden Problemen liegt darin, dass die Anzahl der Variablen in einem Programm beziehungsweise der Parameter einer Funktion fest ist, während die Anzahl der Objekte für nicht terminierende Programme theoretisch nicht begrenzt ist.

7 Fazit zu Teil I

In diesem Teil der Arbeit wurde gezeigt, warum sich rein funktionale Sprachen wie Haskell in der Terminierungsanalyse nicht als Eingabesprache für imperative Programme eignen, auch wenn fortgeschrittene Verfahren zur Analyse rein funktionaler Sprachen existieren. Eine aufwendige statische Programmanalyse imperativer Programme kann daher nicht vermieden werden, indem imperative Programme in rein funktionale Sprachen übersetzt werden. Unter der Annahme, dass sich Haskell als Zielsprache für Übersetzer von imperativen Programmen eignet, wurde eine Reihe von Problemen aufgezeigt, die sich ergeben, wenn die Terminierungsanalyse für das übersetzte Programm mit dem auf dem Terminierungsgraphen basierenden Verfahren durchgeführt wird.

Im Einzelnen konnten wir zeigen, wie die Verwendung von Kombinatoren zu Terminierungsgraphen führt, in denen die Paarpfade so klein werden, dass keine finiten Abhängigkeitspaarprobleme mehr entstehen. Die eingeführte Transformation des Ausrollens von rekursiven Funktionen löst dieses Problem, indem es die symbolische Auswertung zwingt, die Berechnung soweit zu führen, dass sich ein Terminierungsargument ergibt.

Ein weiteres demonstriertes Problem stellen die veränderlichen Variablen imperativer Programme dar. Diese können in Haskell lediglich simuliert werden. Die naheliegende Repräsentation in einer tabellenartigen algebraischen Datenstruktur erweist sich als nicht geeignet. Durch die strukturelle Zerlegung der Variablen in Bezeichner und Wert, wird der Zusammenhang zwischen den Informationen für die Terminierungsanalyse unsichtbar. Auf diese Weise gehen die notwendigen Informationen für ein Terminierungsargument verloren und die Analyse schlägt fehl. Durch Ausnutzung der SSA-CPS Äquivalenz gelingt es jedoch, ein imperatives Programm in eine rekursive Form zu überführen, in der Variablen nicht in einer algebraischen Datenstruktur abgelegt sind, sondern durch Funktionsparameter beschrieben werden.

Anhand der konkreten Sprache Jinja wurden verschiedene Modellierungen diskutiert, mit denen eine objektorientierte Sprache in ein Haskell-Programm übersetzt werden kann. Dabei kann der SSA-CPS Ansatz zur Übersetzung genutzt werden.

Jedoch bleibt mit der Heap-Komponente eine Struktur übrig, die, repräsentiert als algebraische Datenstruktur, Probleme analog zu der Belegung der Variablen verursacht. Die Probleme sind in der Heap-Variante allerdings dadurch verschärft, dass die Anzahl der Elemente auf dem Heap nicht zur Übersetzungszeit feststeht und für nicht-terminierende Programme zumindest potentiell unbegrenzt sein kann. Eine Lösung dieses Problems durch eine Darstellung als Funktionsparameter gelingt daher nicht mehr.

Die Übersetzung eines imperativen Programms in ein Haskell-Programm, sei es mittels einer Transitionsmonade oder mittels Continuations, führt zu einem Programm, in dem auch die Probleme, die sich bei der Terminierungsanalyse imperativer Sprache ergeben, simuliert werden. Insbesondere die Heap-Komponente objektorientierter Sprachen erweist sich als Problem. Zyklische oder mehrfach referenzierte Datenstrukturen auf dem Heap sind in nahezu allen objektorientierten Programmen anzutreffen. Rein funktionale Sprachen hingegen bieten keine Referenzen und können derartige Datenstrukturen nur wenig elegant simulieren. Um die entstehenden Probleme zu lösen, sind aufwendige statische Programmanalysen notwendig. Die bestehende Verfahren zur Terminierungsanalyse von Haskell-Programmen bieten dafür nicht die notwendigen Mechanismen. Eine Terminierungsanalyse, die gut genug ist, Programme zu analysieren, die andere Programme simulieren, würde einen Durchbruch darstellen.

7.1 Verwandte Arbeiten

Techniken zur Konstruktion von Terminierungsbeweisen finden sich vor allem im Bereich der Termersetzung. Dies gilt umso mehr für automatisierte Verfahren zur Terminierungsanalyse. Das fortgeschrittenste Werkzeug in dieser Kategorie ist APROVE [GTSKF04]. Die grundlegende Technik ist das Dependency Pair Framework [AG00, GTSK05].

Die folgenden Arbeiten sind mit der vorliegenden Arbeit insofern verwandt, als dass sie sich mit der automatisierten Terminierungsanalyse von imperativen Programmen beschäftigen.

Das Werkzeug TERMINATOR wird von Cook et al. in [CPR06] beschrieben und basiert auf der Methode des Model-Checkings. Es zielt auf Gerätetreiber ab und verarbeitet dementsprechend in den Sprachen C [Ker88] bzw. C++ [Str00b] verfasste Programme. Die Arbeit [BCDO06] von Berdine et al. beschreibt eine entscheidende Weiterentwicklung für das in TERMINATOR eingesetzte Verfahren, mit der Terminie-

rungsargumente, die die Heap-Struktur berücksichtigen, gefunden werden können. Dazu werden Ergebnisse aus der Separationslogik [Rey02] eingesetzt. Spoto et al. kritisieren diesen Ansatz in [SMP10], da lediglich die Verwendbarkeit für verkettete Listen nachgewiesen ist.

In [AAC+08] stellen Albert et al. ein Verfahren zur Terminierungsanalyse von Java-Bytecode vor. Dabei beschränken sich die Autoren auf eine Teilmenge der in der JVM [LY99] verfügbaren Befehle. Der Ansatz greift jedoch nicht auf die virtuelle Maschine für Jinja aus [KN06] zurück. Das Verfahren geht vom Kontrollflussgraphen aus und erzeugt ein regelbasiertes Programm, dessen Terminierungsverhalten mit bestehenden Verfahren analysiert wird. Es handelt sich also wie bei dem in der vorliegenden Arbeit beschriebenen Ansatz um einen transformierendes Verfahren. Allerdings sind sowohl Quell- als auch Zielsprache von den in der vorliegenden Arbeit verwendeten Sprachen unterschiedlich. Anstatt einer Hochsprache gehen Albert et al. von der virtuellem Maschinensprache Bytecode aus und anstelle der funktionalen Programmiersprache Haskell wird die logische Programmierung als Zwischenstufe gewählt. Es ist zu beachten, dass Java-Bytecode eine objektorientierte Maschinensprache darstellt und Anweisungen wie `invokevirtual` unterstützt, die Ad-hoc-Polymorphie ermöglichen. Die Arbeit beschränkt sich allerdings auf arithmetische Programme. Ist die Terminierung des Programms abhängig von Strukturveränderungen des Heaps, so schlägt der Ansatz fehl. Das Verfahren wurde in dem System COSTA realisiert.

Der Ansatz, Java-Bytecode zu analysieren, entspringt dem Sicherheitsbedürfnis mobiler Plattformen wie Android oder Java ME, die Bytecode ausführen. Dementsprechend finden sich weitere Arbeiten, die sich mit automatisierter Terminierungsanalyse von Bytecode beschäftigen. Spoto, Mesnard und Mayet beschreiben in [SMP10] einen Ansatz, der Strukturen auf dem Heap in die Terminierungsanalyse einbezieht. Der verwendete Bytecode basiert auf dem Jinja-Bytecode, der von Klein und Nipkow in [KN06] neben der Hochsprache ebenfalls definiert wurde, und entspricht damit nur einer Teilmenge der Fähigkeiten der JVM. Dabei wird wie bei Albert et al. ebenfalls eine logisches Programm erzeugt, welches mit bekannten Methoden analysiert wird. Um die Heap-Struktur berücksichtigen zu können, finden vor der Transformation eine Reihe von statischen Analysen statt, unter anderem die von Spoto et al. entwickelte Pfadlängen-Analyse (ebenfalls in [SMP10]). Der Ansatz geht damit deutlich über eine reine Transformation hinaus, was sich auch in der Güte des Verfahrens positiv auswirkt. Das Verfahren wurde in dem System JULIA realisiert.

Der Vorteil von Jinja-Bytecode liegt in der formal definierten Semantik. Arbeiten,

die die Formalisierung der Jinja-Bytecode Semantik in dem Theorembeweiser Isabelle/HOL nutzen, sind dem Autor der vorliegenden Arbeit jedoch nicht bekannt. Jedoch begründen Brockschmidt et al. in [BOvEG10] ihre Wahl von Jinja mit dem Argument der formalen Semantik. Die Arbeit, deren Ergebnisse in das AProVE-System einflossen, verfährt analog zu dem Vorgehen, das in Kapitel 5 vorgestellt wurde. Anhand des Bytecode-Programms wird ein Terminierungsgraph erzeugt, der einer symbolischen Auswertung des Programms entspricht. Der Terminierungsgraph enthält teilweise mehrere Knoten, um die Ausführung derselben Instruktion in verschiedenen Heap-Zuständen zu repräsentieren. Aus diesem Terminierungsgraph werden Termersetzungssysteme erzeugt, die mit dem Dependency Pair Framework gelöst werden.

Teil II

Terminierungsanalyse in der Gegenwart von Ein- und Ausgabe

8 Überblick über Teil II

Unterschiedliche Belange zu trennen ist ein zentrales Prinzip der Softwareentwicklung, siehe zum Beispiel Ghezzi et al. ([GJM03]). Drei Belange, auf die diese Trennung angewendet werden sollte, sind die Eingabe, Verarbeitung und Ausgabe von Daten. Diese Trennung wird in einigen Sprachen forciert. Zum Beispiel ist das Ein- und Ausgabesystem in Haskell als Monade realisiert. Theoretische Grundlagen finden sich in [Mog91], die Umsetzung in Haskell wird zum Beispiel in [PJ02] beschrieben. In vielen anderen, insbesondere imperativen Sprachen wird eine solche Trennung nicht erzwungen. Bei der Analyse von in derartigen Sprachen verfassten Programmen muss das Vorkommen von Ein- und Ausgabeoperationen an beliebigen Stellen berücksichtigt werden.

Bei der Betrachtung der Ausführung eines Programms als Prozess in einem Betriebssystem ist eine softwaretechnische Trennung von Ein- und Ausgabe, so sie denn vorhanden ist, nicht mehr beobachtbar. Seit der Einführung von Time-Sharing in Betriebssystemen (siehe zum Beispiel [SGG00]) ist eine aus Benutzersicht kontinuierliche Steuerung von laufenden Programmen möglich. Diese Eingabe beeinflusst den internen Programmzustand. Außerhalb des Programms ist dieser in weiten Teilen nicht einsehbar. Erst wenn ein Programm Daten ausgibt, kann die Umgebung auf diese reagieren. Auf die Ausgabe folgen so möglicherweise wiederum neue Eingaben für das laufende Programm und zwar in Abhängigkeit von der bisher getätigten Ausgabe. Ein- und Ausgabe sind also keine vollständig getrennten, sondern voneinander abhängige Vorgänge.

Über die Art der Abhängigkeit lassen sich keine Aussagen treffen. Sie ist vollständig der Umgebung unterworfen, in der das Programm tatsächlich ausgeführt wird und damit zum Zeitpunkt der Entwicklung nicht bekannt. Ein automatisiertes Verfahren zur Terminierungsanalyse kann daher keine Annahmen über die Abhängigkeit zwischen Ein- und Ausgabe treffen, sondern muss stets vom ungünstigsten Fall ausgehen. Damit soll nicht unterstellt werden, dass bei der Entwicklung keine Vorstellung existiert, in welcher Umgebung ein Programm eingesetzt werden soll. Eine solche Vorstellung entzieht sich allerdings der automatisierten Analyse. Erst eine explizite Spezifikation der angenommen Abhängigkeiten zwischen Ein- und Ausgabe

könnte in neuartigen Analyseverfahren berücksichtigt werden. Die Erstellung einer solchen Spezifikation bedeutet allerdings zusätzlichen Aufwand, der in der vorliegenden Arbeit gerade durch den Einsatz eines automatisierten Verfahrens vermieden werden soll.

Stattdessen soll daher die folgende Alternative betrachtet werden. Aufgrund der unklaren Abhängigkeit zwischen Ein- und Ausgabe lassen sich mit den gängigen Methoden lediglich die Programmteile analysieren, die frei von Ein- und Ausgabe sind. Dabei ist das Ergebnis einer Analyse umso aussagekräftiger, je größer der analysierte Programmteil ist. Daher ist eine Programmtransformation wünschenswert, die möglichst große Programmteile hervorbringt, die frei von Ein- und Ausgabe sind.

Dieser Teil der Arbeit beschäftigt sich daher mit folgenden zwei Aspekten. Zuerst wird die Semantik einer Sprache, die Ein- und Ausgabe unterstützt, formalisiert, ohne dabei die Abhängigkeit zwischen Ein- und Ausgabe einzuschränken. Im Anschluss wird auf Grundlage dieser Formalisierung ein Verfahren beschrieben, um die semantische Äquivalenz eines transformierten Programms bezüglich des ursprünglichen Programms zu überprüfen.

Dabei wird in den folgenden Kapiteln eine Sprache namens **IMP** vorgestellt. Diese wurde von Winskel in seiner Einführung in die formale Semantik von Programmiersprachen ([Win93]) definiert. Es handelt sich um eine einfache, imperative Sprache, die von Winskel ohne Ein- und Ausgabemechanismen definiert wurde. Sie entspricht in dieser Fassung einer WHILE-Sprache, wie sie zum Beispiel auch in [NNH99] beschrieben wird. Sie wird in dem folgenden Kapitel vorgestellt und um Anweisungen zur Ein- und Ausgabe erweitert.

Die grundlegende Idee wurde bereits in [Sch10] vorgestellt. In der vorliegenden Fassung wird allerdings eine andere Modellierung des Zustands gewählt, die die Abhängigkeit zwischen Ein- und Ausgabe formal geschickter ausdrückt. Ferner wird das Konzept der Kopplung zweier Anweisungen eingeführt. Es handelt sich dabei um eine statische Eigenschaft, die für beliebige Anweisungen unabhängig von einem Eingabeprogramm definiert werden kann. Die Analyse der Abhängigkeiten zwischen zwei Anweisungen wird durch das Konzept der Kopplung vereinfacht.

Im Folgenden wird das Kürzel EA als Sub- oder Superskript verwendet, um Konstrukte zu kennzeichnen, die in direkter Beziehung zu Ein- und Ausgabeoperationen stehen.

9 Eine Sprache mit Ein- und Ausgabe

In diesem Abschnitt soll die verwendete Sprache **IMP** beschrieben werden, sowohl in ihrer Urfassung aus [Win93] als auch die in der vorliegenden Arbeit verwendeten Erweiterungen. Die Grammatik der Sprache wird im folgenden Abschnitt vorgestellt und dabei die grundlegenden Konzepte informell eingeführt. Im Anschluss wird die Semantik der Sprache formal definiert, wiederum in der Ur- als auch in der erweiterten Fassung.

9.1 Die Grammatik von IMP

Die Grammatik von **IMP** ist kontextfrei und wird im Folgenden in Backus-Naur-Form beschrieben. Die Urfassung entstammt direkt [Win93], die Syntax der Primitive für Ein- und Ausgabe ist angelehnt an die Sprache Pascal, siehe [Wir71]. In der vorliegenden Arbeit kommt noch eine Änderung hinzu, nämlich die Einführung einer bedingten Anweisung ohne Alternativzweig.

9.1.1 Die Grammatik der Urfassung

Die Sprache **IMP** wird von Winskel in [Win93] eingeführt, um Konzepte der operationellen Semantik zu demonstrieren. Dementsprechend handelt es sich um eine absichtlich einfach gestaltete, imperative Sprache. Alle Variablen sind vom selben Datentyp, der die ganzen Zahlen repräsentiert. Die Menge aller Variablen wird mit $\mathbb{L}$ bezeichnet. Werte aus den ganzen Zahlen $\mathbb{Z}$ lassen sich mit Addition, Subtraktion und Multiplikation berechnen. Ferner können Werte aus $\mathbb{Z}$ verglichen werden. Auf diese Weise ergibt sich ein zweiter Datentyp, die booleschen Werte $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$. Dieser wird lediglich implizit zur Definition von bedingten Anweisungen und Schleifen verwendet. Bedingte Anweisungen und Schleifen stellen neben der Sequenzierung zweier Anweisungen die einzigen zusammengesetzten Anweisungen der Sprache dar. Es verbleiben zwei atomare Anweisungen, `skip`, welche den Zustand nicht ändert,

und die Zuweisung eines Werts an eine Variable. Die Grammatik der Sprache **IMP**, wie Winskel sie definiert, kann damit in knapper Form vollständig angegeben werden.

Definition 9.1. *Die Menge aller arithmetischen Ausdrücke wird mit $\mathbb{E}_A$, die Menge der booleschen Ausdrücke mit $\mathbb{E}_B$ und die Menge aller Anweisungen mit $\mathbb{C}$ notiert. In diesem Kapitel bezeichnen Variablen a, a_1, a_2 etc. arithmetische Ausdrücke, wenn nicht auf anderes hingewiesen wird. Analog zu der a-Konvention, werden die Variablen b, b_1, b_2 etc. in diesem Kapitel für boolesche Ausdrücke und Variablen c, c_1, c_2 für Anweisungen verwendet. Die Grammatik von* **IMP** *wird in Backus-Naur-Form definiert:*
Arithmetische Ausdrücke*:*

$$a ::= n \mid X \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \text{ mit } n \in \mathbb{Z} \text{ und } X \in \mathbb{L}$$

Boolesche Ausdrücke*:*

$$b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2$$

Anweisungen*:*

$$c ::= c_{atomar} \mid c_1; c_2 \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c$$

Atomare Anweisungen*:*

$$c_{atomar} ::= \texttt{skip} \mid X := a$$

*Sind zwei **IMP**-Konstrukte x und y syntaktisch gleich, so wird die Notation $x \equiv y$ verwendet. Ein arithmetischer Ausdruck $a \in \mathbb{E}_A$ heißt* Literal *falls $a \equiv n$ gilt. Ein boolescher Ausdruck $b \in \mathbb{E}_B$ heißt* Literal *falls $b \equiv \texttt{true}$ oder $b \equiv \texttt{false}$ gilt.*

9.1.2 Die Grammatik der erweiterten Fassung

Die Sprache **IMP** soll vor allem um die Fähigkeit zur Ein- und Ausgabe von Daten erweitert werden. Es wird dabei keine Annahme darüber gemacht, in welcher Form Ein- und Ausgabe erfolgen. Die am leichtesten vorstellbare Form wäre sicherlich die Verwendung gewöhnlicher Dateien auf einem Ausgabemedium. Die für die vorgestellte Erweiterung gewählten Anweisungen erinnern an die entsprechenden Anweisungen aus der Sprache Pascal (siehe zum Beispiel [Wir71]). Getreu der Philosophie der Betriebssysteme der Unix-Familie, wonach alles eine Datei ist (siehe zum

Beispiel [Tho78]), sollte allerdings beachtet werden, dass die hier modellierte Ein- und Ausgabe auch einen Zugriff auf Netzwerke, andere Prozesse oder klassische Ein- und Ausgabegeräte wie Tastatur und Bildschirm darstellen könnte. Es sollte daher davon ausgegangen werden, dass Ein- und Ausgabe in irgendeiner Weise miteinander verbunden sein könnten, sei es durch den Zustand eines externen Prozesses oder durch einen Benutzer, dessen Tastatureingaben abhängig von der Bildschirmausgabe sind. Insbesondere darf nicht angenommen werden, dass die Eingabe im Voraus feststeht.

Die Erweiterung der Grammatik von **IMP** um Ein- und Ausgabe wird auf syntaktischer Ebene durch die Ergänzung der Grammatik um zwei atomare Anweisungen erreicht.

Definition 9.2. *Die* atomaren Anweisungen *der erweiterten Fassung von* ***IMP*** *sind definiert durch*

$$c_{atomar} ::= \texttt{skip} \mid X := a \mid \texttt{read}(X) \mid \texttt{write}(a).$$

Dabei ist $X \in \mathbb{L}$ *und* $a \in \mathbb{E}_A$.

Die Syntax ist, wie bereits erwähnt, angelehnt an die Syntax von Pascal. Die Wahl von $\texttt{read}(X)$ als Eingabeoperation bedarf allerdings einer ausführlicheren Erklärung. Informell soll die Anweisung bewirken, dass der eingegebene Wert in der Variablen X gespeichert wird. Es handelt sich daher bei X in der Tat um eine Variable und nicht um einen Ausdruck.

Eine Alternative hierzu findet sich in der Einführung in die denotationelle Semantik von Allison (siehe [All86]). Dort wird eine imperative Sprache zwar nur um eine Ausgabeoperation erweitert, während auf die Modellierung der Eingabe verzichtet wird. Jedoch weist Allison auf zwei syntaktische Möglichkeiten hin, wie eine solche Erweiterung zu bewerkstelligen sei. Zum einen könnte eine Anweisung gewählt werden, wie es in der vorliegenden Arbeit der Fall ist. Alternativ könnte ein Ausdruck definiert werden. Das Resultat einer Eingabeoperation könnte dann mit der Zuweisungsoperation an eine Variable gebunden werden.

Letzteres ist syntaktisch eleganter, da der scheinbare Ausdruck X als Argument von `read` vermieden wird. Allerdings bietet die Verwendung einer Anweisung den Vorteil, dass Ausdrücke frei von Seiteneffekten bleiben. Damit kann auf eine Definition der Auswertungsreihenfolge verzichtet werden und die formale Semantik lässt sich leichter und kompakter beschreiben.

Neben der Erweiterung der Grammatik um Ein- und Ausgabe wird eine zusätzli-

che Form der bedingten Anweisung eingeführt, die keinen Alternativzweig beinhaltet. Dies dient im Folgenden lediglich der Vereinfachung durch eine Vorverarbeitung.

Definition 9.3. *Die* zusammengesetzten Anweisungen *der erweiterten Fassung von* **IMP** *sind definiert durch*

$$c ::= c_{atomar} \mid c_1; c_2 \mid \texttt{if } b \texttt{ then } c \ \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } b \texttt{ do } c$$

Dabei ist $b \in \mathbb{E}_B$ *und* $c, c_1, c_2 \in \mathbb{C}$.

Die Grammatik der erweiterten Fassung von **IMP** unterscheidet sich von Definition 9.1 also nur in den Regeln für Anweisungen. Im Folgenden wird mit **IMP** die erweiterte Variante der Sprache bezeichnet.

9.1.3 Syntaktische Definitionen

Auf der Basis dieser Grammatik werden in der vorliegenden Arbeit folgende, nicht auf Winskel zurückgehenden Hilfskonstruktionen definiert. Es wird mit den arithmetischen und booleschen Ausdrücken begonnen.

Definition 9.4. *Die Menge der* direkten, arithmetischen Teilausdrücke *eines arithmetischen Ausdrucks ist definiert als*

$$\mathrm{Subexp}_A(a) = \begin{cases} \{a_1, a_2\} & \textit{falls } a \equiv a_1 + a_2, \\ \{a_1, a_2\} & \textit{falls } a \equiv a_1 - a_2, \\ \{a_1, a_2\} & \textit{falls } a \equiv a_1 \times a_2, \\ \emptyset & \textit{sonst.} \end{cases}$$

Es gilt $\mathrm{Subexp}_A(a) \subseteq \mathbb{E}_A$.

Definition 9.5. *Die Menge der* direkten Teilausdrücke *eines booleschen Ausdrucks ist definiert als*

$$\mathrm{Subexp}_B(b) = \begin{cases} \{a_1, a_2\} & \textit{falls } e \equiv a_1 = a_2, \\ \{a_1, a_2\} & \textit{falls } e \equiv a_1 \leq a_2, \\ \{b\} & \textit{falls } e \equiv \neg b, \\ \{b_1, b_2\} & \textit{falls } e \equiv b_1 \vee b_2, \\ \{b_1, b_2\} & \textit{falls } e \equiv b_1 \wedge b_2, \\ \emptyset & \textit{sonst.} \end{cases}$$

Es gilt $\text{Subexp}_B(b) \subseteq \mathbb{E}_A \cup \mathbb{E}_B$.

Definition 9.6. *Die Menge der* direkten Teilausdrücke *eines Ausdrucks* $e \in \mathbb{E}_A \cup \mathbb{E}_B$ *wird definiert als*

$$\text{Subexp}(e) = \begin{cases} \text{Subexp}_A(e) & \textit{falls } e \in \mathbb{E}_A, \\ \text{Subexp}_B(e) & \textit{falls } e \in \mathbb{E}_B. \end{cases}$$

Es gilt $\text{Subexp}(e) \subseteq \mathbb{E}_A \cup \mathbb{E}_B$.

Definition 9.7. *Die Menge der Variablen, die in einem beliebigen Ausdruck* $e \in \mathbb{E}_A \cup \mathbb{E}_B$ *verwendet werden, wird bezeichnet als* $\text{Vars}(e)$ *und ist definiert als*

$$\text{Vars}(e) = \begin{cases} \emptyset & \textit{falls } e \textit{ ein Literal ist,} \\ \{X\} & \textit{falls } e \equiv X, \\ \bigcup\limits_{e' \in \text{Subexp}(e)} \text{Vars}(e') & \textit{sonst.} \end{cases}$$

Im Folgenden werden Beweise über Ausdrücke geführt. Diese lassen sich am einfachsten gestalten, indem Induktion über die Tiefe eines Ausdrucks verwendet wird. Die folgende Definition ist für diese Fälle hilfreich.

Definition 9.8. *Die Tiefe eines Ausdrucks* $e \in \mathbb{E}_A \cup \mathbb{E}_B$ *ist definiert als*

$$D(e) = \begin{cases} 1 + D(e_1) & \textit{falls } \text{Subexp}(e) = \{e_1\}, \\ 1 + \max(D(e_1), D(e_2)) & \textit{falls } \text{Subexp}(e) = \{e_1, e_2\}, \\ 0 & \textit{sonst.} \end{cases}$$

Um die Tiefe boolescher Ausdrücke zu beschreiben, muss aufgrund des unären booleschen Negationsoperator $\neg$ die Fallunterscheidung $\text{Subexp}(e) = \{e_1\}$ eingeführt werden.

Definition 9.9. *Die Menge der* direkten Teilanweisungen *einer Anweisung* $c \in \mathbb{C}$ *wird definiert als*

$$\text{Sub}_\text{d}(c) = \begin{cases} \emptyset, & \textit{falls } c \textit{ eine atomare Anweisung ist,} \\ \{c_1\} & \textit{falls } c \equiv \texttt{while } b \texttt{ do } c_1, \\ \{c_1\} & \textit{falls } c \equiv \texttt{if } b \texttt{ then } c_1, \\ \{c_1, c_2\} & \textit{falls } c \equiv \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \\ \{c_1, c_2\} & \textit{falls } c \equiv c_1; c_2. \end{cases}$$

Definition 9.10. *Die Menge der* atomaren Teilanweisungen *einer Anweisung* $c \in \mathbb{C}$ *wird definiert als*

$$\mathrm{Atom}(c) = \begin{cases} \{c\}, & \textit{falls } c \textit{ eine atomare Anweisung ist,} \\ \bigcup \mathrm{Atom}(c') & \textit{mit } c' \in \mathrm{Sub_d}(c) \textit{ sonst.} \end{cases}$$

Definition 9.11. *Die Menge der* Teilanweisungen *einer Anweisung* $c \in \mathbb{C}$ *wird definiert als*

$$\mathrm{Sub}(c) = \bigcup_{d \in \mathrm{Sub_d}(c)} \mathrm{Sub}(d) \cup \mathrm{Sub_d}(c).$$

9.2 Die Semantik von IMP

Winskel definiert in [Win93] nicht nur die Grammatik, sondern auch die Semantik der Sprache **IMP**. Dies geschieht mit den Mitteln der von Plotkin in [Plo81] eingeführten strukturierten operationellen Semantik. Diese Definition ist in hohem Maße abhängig von dem verwendeten Zustandsraum. Es folgt daher, dass die Erweiterung einer Sprache um neue Konstrukte wie Datentypen, Kontrollstrukturen oder neuer atomarer Anweisungen, umso komplexer wird, je stärker der Zustandsraum verändert wird. Für die in der vorliegenden Arbeit vorgestellten Erweiterung von **IMP** um Ein- und Ausgabeoperationen ist eine Änderung des Zustandsraum unumgänglich.

Wie bereits erwähnt, findet sich eine ähnliche Erweiterung in der Einführung in die denotationelle Semantik von Allison (siehe [All86]). Dort wird mithilfe der denotationellen Semantik eine imperative Sprache beschrieben. Allison bemerkt in seinem 1986 veröffentlichten Werk, dass ein Abbild des Speicherinhalts schon seit längerer Zeit kein übliches Mittel zur Ermittlung des Resultats einer Programmausführung ist. Mit dieser Begründung erweitert er den Zustandsraum um eine Folge von Ausgaben. Allison demonstriert damit allerdings lediglich die prinzipiell mögliche Integration von Ausgaben in eine formale Semantik. Die Eingabe wird hingegen nicht formalisiert. Allison vermerkt lediglich, dass dies „leicht“ möglich sei, und zwar, wie bereits diskutiert, entweder durch die Integration eines weiteren Ausdrucks oder einer atomaren Anweisung. Dadurch wird der Eindruck erweckt, dass beide Elemente getrennt voneinander integriert werden können. Dies ist allerdings nur zulässig, wenn davon ausgegangen wird, dass die gesamte Eingabe bereits vor der Programmausführung feststeht oder zumindest unabhängig von der Ausgabe ist.

Die vorliegende Arbeit geht über das Resultat von Allison hinaus, nicht nur weil die Eingabeoperation explizit mit den Mitteln der strukturierten operationellen Se-

mantik beschrieben wird, sondern vor allem, da die Abhängigkeiten zwischen Ein- und Ausgabe nicht künstlich eingeschränkt werden.

9.2.1 Der Zustand in IMP

Um die operationelle Semantik einer Sprache zu formalisieren, muss der Zustand formalisiert werden. In [Win93] definiert Winskel den Zustand eines **IMP**-Programms als Funktion $\sigma : \mathbb{L} \rightarrow \mathbb{Z}$. Auf diese Weise lässt sich die Belegung der Variablen beschreiben. In der vorliegenden Arbeit wird das kartesische Produkt aus Eingabe-, Ausgabe- und Variablenzustands gebildet. Dies ist analog zu Allisons Vorgehen, bei dem ein Tupel aus dem Zustand ohne Ausgabe und dem Ausgabezustand gebildet wurde.

Es scheint naheliegend, den Eingabezustand als unendliche Folge von ganzen Zahlen in der Form

$$\sigma_{in} : T_{in} \text{ wobei } T_{in} = (\mathbb{Z}, T_{in})$$

zu modellieren. Diese Modellierung als Eingabestrom birgt allerdings Probleme, wie das folgende Beispiel illustriert.

Sei $\sigma_{in} = (n_1, (n_2, (n_3, \ldots)))$ mit $n_1, n_2, n_3 \in \mathbb{Z}$. Durch die Ausführung einer Leseanweisung verändert sich der Strom zu $(n_2, (n_3, \ldots)))$. Dies entspricht dem Lesen aus einer Datei, deren Inhalt sich nicht verändert. Allerdings soll gerade eine möglicherweise miteinander verbundene Ein- und Ausgabe modelliert werden. Die Eingabe ist also von der Ausgabe abhängig. Die oben beschriebene Modellierung nimmt darauf allerdings keine Rücksicht. Nach einer Schreiboperation könnten gänzlich andere Eingaben auftreten. Dementsprechend müsste der gesamte Eingabestrom ersetzt werden.

In der vorliegenden Arbeit wird die Eingabe daher als Funktion modelliert, die die bisherige Ausgabe auf den nächsten Wert abbildet, der eingegeben wird. Für eine formale Definition muss daher zunächst die Ausgabe beschrieben werden. Dazu wird ein Ausgabestrom verwendet. Dabei handelt es sich analog zu dem angesprochenen Eingabestrom um eine möglicherweise unendliche Folge, allerdings über einem Ausgabealphabet Σ_{out} bestehend aus den ganzen Zahlen $\mathbb{Z}$ und einer sogenannten Lesemarke R. Die Lesemarken im Ausgabestrom markieren Lesevorgänge. Ihr Auftreten in der Ausgabe entspricht der Vorstellung, dass das Einlesen eines Wertes ein von außen beobachtbarer Vorgang ist, der folglich Auswirkungen auf die Eingabe haben könnte.

Definition 9.12. *Ein* Ausgabestrom *ist eine möglicherweise unendliche Folge über*

dem Ausgabealphabet $\Sigma_{out} = \mathbb{Z} \cup \{R\}$. *Dabei heißt R* Lesemarke *und es gilt* $R \notin \mathbb{Z}$. *Die leere Ausgabe wird mit* ε *bezeichnet.*

Der Ausgabestrom ist für terminierende Programme zwangsläufig endlich und kann insbesondere auch leer sein. Bei nicht terminierenden Programmen kann es zu einem unendlichen Ausgabestrom kommen, wenn in den sich stets wiederholenden Anweisungen des Programms eine Ausgabe erfolgt. Ist dies nicht der Fall, so ist auch der Ausgabestrom nicht terminierender Programme endlich.

Die Lesemarken unterscheiden den Ausgabestrom in der vorliegenden Arbeit von der Modellierung in [All86]. Ferner haben die Lesemarken in der vorliegenden Variante eine andere Wirkung als in einer Modellierung mit einem Eingabestrom. Der Unterschied wird nach der Definition der Eingabefunktion beschrieben.

Definition 9.13. *Die* Eingabefunktion *modelliert die Abhängigkeit zwischen den Eingaben, die ein Programm während seiner Ausführung in einer bestimmten Umgebung liest, und den Ausgaben, die es in dieser Umgebung vornimmt. Die Eingabefunktion wird notiert als*

$$\sigma_{in} : \Sigma_{out}^* \to \mathbb{Z}.$$

Auf die Modellierung eines Endes der Eingabe wird zu Gunsten der Einfachheit der `read`-Anweisung verzichtet. Ansonsten müsste eine Fehlerbehandlung ergänzt werden oder ein weiterer Datentyp eingeführt werden.

Da σ_{in} eine wohldefinierte Funktion ist, wird ein fester Ausgabestrom $\sigma_{out} \in \Sigma_{out}^*$ stets auf den selben Wert abgebildet. Die Erwartung an eine Leseoperation ist allerdings, dass auch ohne neue Ausgaben neue Werte eingelesen werden können. Die Lesemarken sorgen in der vorliegenden Arbeit dafür, dass sich der Ausgabestrom mit jeder Leseoperation ändert und somit wiederholte Leseoperationen nicht dasselbe Ergebnis liefern müssen. Die Ergebnisse aller Leseoperationen sind bei einer geeigneten Eingabefunktion unterschiedlich. Die Ausführungsumgebung kann daher in der Kombination mit einem geeigneten Ausgabestrom als mathematische Funktion modelliert werden.

Die Alternative zur Modellierung als Funktion ist die Modellierung als Strom. Dadurch ergibt sich auch ohne die Lesemarken das gewünschte Verhalten der Leseoperation. Am Ende des Abschnitts 9.3 wird näher auf die Unterschiede zwischen den beiden Modellierungen eingegangen.

Es ergibt sich folgender Zustandsraum.

Definition 9.14. *Der* Zustand *eines **IMP**-Programms wird beschrieben durch ein Tripel*

$$\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$$

mit

$$\sigma_V : \mathbb{L} \to \mathbb{Z},$$

$$\sigma_{in} : \Sigma_{out}^* \to \mathbb{Z} \text{ und}$$

$$\sigma_{out} \in \Sigma_{out}.$$

Die Komponente σ_V wird als Variablenbelegung, *σ_{in} als* Eingabefunktion *und σ_{out} als* Ausgabestrom *bezeichnet. Ferner heißt R* Lesemarke. *Der leere Ausgabestrom wird mit ε notiert. Die Menge aller möglichen Zustände heißt Σ. Für einen Zustand $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$ dient $\sigma(X)$ als Kurzschreibweise für $\sigma_V(X)$.*

Folgende Definition beschreibt die Veränderung der Variablenbelegung und entstammt bis auf die Komponenten σ_{in} und σ_{out} der Arbeit von [Win93].

Definition 9.15. *Der sich durch die Belegung einer Variablen X mit dem Wert $n \in \mathbb{Z}$ ergebende Zustand im Zustand $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$ wird notiert als $\sigma[n/X] = (\sigma_V[n/X], \sigma_{in}, \sigma_{out})$ mit*

$$\sigma_V[n/X] : \mathbb{L} \to \mathbb{Z}$$

$$\sigma_V[n/X](Y) = \begin{cases} n & \text{falls } Y = X, \\ \sigma_V(Y) & \text{falls } Y \neq X. \end{cases}$$

9.2.2 Die Semantik der Urfassung von IMP

Mit der Definition des Zustands lässt sich nun die Semantik der Auswertung von Ausdrücken und der Ausführung von Anweisungen beschreiben. Die operationelle Semantik wird dabei definiert durch eine Menge von Regeln der Form $\langle x, \sigma \rangle \to y$. Dabei ist die Notation überladen, sodass x ein arithmetischer oder boolescher Ausdruck sein kann. Ist x ein Ausdruck, so beschreibt $\langle x, \sigma \rangle \to y$, dass x im Zustand σ zu y ausgewertet wird. Im Fall $x \in \mathbb{E}_A$ ist y ein Wert aus $\mathbb{Z}$, im Fall $x \in \mathbb{E}_B$ gilt $y \in \mathbb{B}$.

Die folgenden Regeln entsprechen bis auf 9.24 und 9.25 den Regeln aus [Win93]. Teilweise sind die Regeln sogar syntaktisch gleich. Dies ist möglich, da mit der Kurzschreibweise $\sigma(X)$ anstelle von $\sigma_V(X)$ für $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$ die Erweiterung des Zustands verborgen bleibt. Es bleibt allerdings bei dem Unterschied, dass Winskels Regeln auf dem Zustandsraum $\Sigma = \mathbb{L} \to \mathbb{Z}$ arbeiten während die hier angegeben Regeln sich auf den Zustandsraum $\Sigma = (\mathbb{L} \to \mathbb{Z}) \times (\Sigma_{out} \to \mathbb{Z}) \times \Sigma_{out}$ beziehen.

Definition 9.16. *Die Auswertung arithmetischer Ausdrücke in **IMP** wird beschrieben durch die folgenden Regeln, bei denen $a_1, a_2 \in \mathbb{E}_A$, $X \in \mathbb{L}$ und $n, n_1, n_2 \in \mathbb{Z}$ gilt.*

1. Auswertung von Literalen:

$$\frac{}{\langle n, \sigma \rangle \rightarrow n}$$

2. Auswertung von Variablen:

$$\frac{}{\langle X, \sigma \rangle \rightarrow \sigma(X)}$$

3. Auswertung der Addition:

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 + a_2, \sigma \rangle \rightarrow n} \; \textit{mit } n = n_1 + n_2$$

4. Auswertung der Subtraktion

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 - a_2, \sigma \rangle \rightarrow n} \; \textit{mit } n = n_1 - n_2$$

5. Auswertung der Multiplikation

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 \times a_2, \sigma \rangle \rightarrow n} \; \textit{mit } n = n_1 * n_2$$

Definition 9.17. *Die Auswertung boolescher Ausdrücke in* ***IMP*** *wird beschrieben durch die folgenden Regeln, bei denen* $a_1, a_2 \in \mathbb{E}_A$*,* $b, b_1, b_2 \in \mathbb{E}_B$*,* $X \in \mathbb{L}$ *und* $n, n_1, n_2 \in \mathbb{Z}$ *gilt.*

1. Auswertung von Literalen:

$$\langle \texttt{true}, \sigma \rangle \rightarrow \texttt{true}$$

$$\langle \texttt{false}, \sigma \rangle \rightarrow \texttt{false}$$

2. Auswertung von Gleichheit und Ungleichheit:

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 = a_2, \sigma \rangle \rightarrow t} \; \textit{mit } t = \begin{cases} \texttt{true} & \textit{falls } n_1 = n_2 \\ \texttt{false} & \textit{sonst} \end{cases}$$

$$\frac{\langle a_1, \sigma \rangle \rightarrow n_1 \quad \langle a_2, \sigma \rangle \rightarrow n_2}{\langle a_1 \leq a_2, \sigma \rangle \rightarrow t} \; \textit{mit } t = \begin{cases} \texttt{true} & \textit{falls } n_1 \leq n_2 \\ \texttt{false} & \textit{sonst} \end{cases}$$

3. Auswertung der Negation*:*

$$\frac{\langle b, \sigma \rangle \to \texttt{true}}{\langle \neg b, \sigma \rangle \to \texttt{false}} \qquad \frac{\langle b, \sigma \rangle \to \texttt{false}}{\langle \neg b, \sigma \rangle \to \texttt{true}}$$

4. Auswertung von Konjunktion und Disjunktion*:*

$$\frac{\langle b_1, \sigma \rangle \to t_1 \quad \langle b_2, \sigma \rangle \to t_2}{\langle b_1 \wedge b_2, \sigma \rangle \to t_c} \; mit\ t_c = \begin{cases} \texttt{true} & falls\ t_1 = t_2 = \texttt{true} \\ \texttt{false} & sonst \end{cases}$$

$$\frac{\langle b_1, \sigma \rangle \to t_1 \quad \langle b_2, \sigma \rangle \to t_2}{\langle b_1 \vee b_2, \sigma \rangle \to t_d} \; mit\ t_d = \begin{cases} \texttt{false} & falls\ t_1 = t_2 = \texttt{false} \\ \texttt{true} & sonst \end{cases}$$

Die Notation wird weiter überladen, sodass auch die Ausführung einer Anweisung $c \in \mathbb{L}$ im Zustand $\sigma \in \Sigma$ durch $\langle c, \sigma \rangle \to \sigma'$ beschrieben wird. Der Zustand $\sigma' \in \Sigma$ wird als Folgezustand von σ für die Anweisung c bezeichnet.

Definition 9.18. *Die Anweisung* `skip` *verändert den Zustand nicht, es gilt daher*

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \to \sigma.}$$

Definition 9.19. *Eine* Zuweisung $X := a$ *wertet den Ausdruck* $a \in \mathbb{E}_A$ *aus und weist den Wert der Variablen* $X \in \mathbb{L}$ *zu. Es gilt die Regel*

$$\frac{\langle a, (\sigma_V, \sigma_{in}, \sigma_{out}) \rangle \to n}{\langle X := a, (\sigma_V, \sigma_{in}, \sigma_{out}) \rangle \to (\sigma_V[n/X], \sigma_{in}, \sigma_{out}).}$$

Um zu betonen, dass die Zuweisung die Ein- und Ausgabeströme nicht verändert, wurde hier der Zustand als Tupel aus Variablenbelegung, Eingabestrom und Ausgabestrom beschrieben.

Es verbleiben lediglich die drei zusammengesetzten Anweisungen, namentlich die Sequenz, die bedingte Anweisung und die Schleife. Sie werden in dieser Reihenfolge eingeführt.

Definition 9.20. *Zwei Anweisungen* $c_1, c_2 \in \mathbb{C}$ *werden nacheinander ausgeführt, wenn sie mit dem Semikolon verbunden werden. Es gilt die Regel*

$$\frac{\langle c_1, \sigma \rangle \to \sigma'' \quad \langle c_2, \sigma'' \rangle \to \sigma'}{\langle c_1; c_2, \sigma \rangle \to \sigma'.}$$

Die beiden Anweisungen $c_1; c_2$ *bilden eine* Sequenz*. Der Zustand* σ' *wird als* Zwi-

schenzustand *bezeichnet.*

Definition 9.21. *Für einen booleschen Ausdruck* $b \in \mathbb{E}_B$ *und die Anweisungen* $c_1, c_2 \in \mathbb{C}$ *lauten die Regeln für die bedingte Anweisung*

$$\frac{\langle b, \sigma\rangle \to \texttt{true} \quad \langle c_1, \sigma\rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \sigma\rangle \to \sigma'}$$

und

$$\frac{\langle b, \sigma\rangle \to \texttt{false} \quad \langle c_2, \sigma\rangle \to \sigma''}{\langle \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \sigma\rangle \to \sigma''.}$$

Definition 9.22. *Für einen booleschen Ausdruck* $b \in \mathbb{E}_B$ *und eine Anweisung* $c \in \mathbb{C}$ *lauten die Regeln für die Schleifen-Anweisung*

$$\frac{\langle b, \sigma\rangle \to \texttt{false}}{\langle \texttt{while } b \texttt{ do } c, \sigma\rangle \to \sigma}$$

und

$$\frac{\langle b, \sigma\rangle \to \texttt{true} \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle \texttt{while } b \texttt{ do } c, \sigma''\rangle \to \sigma'}{\langle \texttt{while } b \texttt{ do } c, \sigma\rangle \to \sigma'.}$$

9.2.3 Die Semantik der Spracherweiterungen

Die Semantik der erweiterten Fassung von **IMP** besteht aus den Regeln des vorherigen Abschnitts, welche unverändert weiterhin gelten. Die folgenden drei Regeln beschreiben die Semantik der zusätzlichen Anweisungen.

Definition 9.23. *Für einen booleschen Ausdruck* $b \in \mathbb{E}_B$ *und die Anweisungen* $c \in \mathbb{C}$ *lautet die Regeln für die bedingte Einzelanweisung*

$$\frac{\langle b, \sigma\rangle \to \texttt{true} \quad \langle c, \sigma\rangle \to \sigma'}{\langle \texttt{if } b \texttt{ then } c, \sigma\rangle \to \sigma'}$$

und

$$\frac{\langle b, \sigma\rangle \to \texttt{false}}{\langle \texttt{if } b \texttt{ then } c, \sigma\rangle \to \sigma.}$$

Definition 9.24. *Für die* Ausgabeanweisung `write` *und einen arithmetischen Ausdruck* $a \in \mathbb{E}_A$ *gilt die Regel*

$$\frac{\langle a, (\sigma_V, \sigma_{in}, \sigma_{out})\rangle \to m}{\langle \texttt{write}(a), (\sigma_V, \sigma_{in}, \sigma_{out})\rangle \to (\sigma_V, \sigma_{in}, m \cdot \sigma_{out})}.$$

Definition 9.25. *Für die* Eingabeanweisung `read` *und eine Variable* $a \in \mathbb{L}$ *gilt die Regel*

$$\frac{}{\langle \textbf{\textit{read}}(X), (\sigma_V, \sigma_{in}, \sigma_{out})\rangle \to (\sigma_V[n/X], \sigma_{in}, R \cdot \sigma_{out})} \text{ mit } n = \sigma_{in}(\sigma_{out}).$$

Im Gegensatz zu einem Eingabestrom ist das Element σ_{in} in dieser Arbeit unveränderlich. Dies spiegelt die Tatsache wider, dass ein Programm keine direkte Kontrolle über die Eingabe besitzt, was allerdings nicht bedeutet, dass es keinen Zusammenhang zwischen der Ein- und Ausgabe gibt. Die Art und Weise der Abhängigkeit wird allerdings nicht modelliert. Dies ist auch nicht mehr Teil des Programms, sondern Teil der Ausführungsumgebung, in der das Programm ausgeführt wird und damit möglicherweise in jeder Ausführung unterschiedlich.

Die Modellierung zeigt, dass die Ausführung eines Programms mit Ein- und Ausgabe von drei scharf getrennten Elementen abhängt. Erstens gibt es einen internen, von außen nicht beobachtbaren Zustand. Im Fall von **IMP** ist dies die Variablenbelegung. Zweitens existiert eine äußere Umgebung, die aus der Sicht des Programms unveränderlich ist. Diese kann oder kann nicht von der Ausgabe abhängen. Das letzte Element bildet der Ausgabestrom, welcher den beobachtbaren Signalen des Programms entspricht, auf die die Umgebung reagieren kann.

Definition 9.26. *Für jeden Zustand* $\sigma \in \Sigma$ *bildet die Funktion* $t_\sigma : \mathbb{C} \to \mathbb{C}^*$ *eine Anweisung* c *auf die im Zuge der Ausführung* c *im Zustand* σ ausgeführten, atomaren Anweisungen *ab. Die Funktion ist definiert als*

$$\begin{aligned}
t_\sigma(c) &= \langle c\rangle \text{ falls } c \text{ eine atomare Anweisung ist} \\
t_\sigma(c; d) &= t_\sigma(c) \cdot t_{\sigma'}(d) \\
t_\sigma(\texttt{if } a \texttt{ then } c) &= \begin{cases} t_\sigma(c) & \text{falls } \langle a, \sigma\rangle \to \texttt{true} \\ \epsilon & \text{sonst} \end{cases} \\
t_\sigma(\texttt{if } a \texttt{ then } c \texttt{ else } d) &= \begin{cases} t_\sigma(c) & \text{falls } \langle a, \sigma\rangle \to \texttt{true} \\ t_\sigma(d) & \text{sonst} \end{cases} \\
t_\sigma(\texttt{while } a \texttt{ do } c) &= \begin{cases} t_\sigma(c) \cdot t_{\sigma'}(\texttt{while } a \texttt{ do } c) & \text{falls } \langle a, \sigma\rangle \to \texttt{true} \\ \epsilon & \text{sonst} \end{cases}
\end{aligned}$$

wobei stets $\langle c, \sigma\rangle \to \sigma'$.

9.3 Semantische Äquivalenz

Der Begriff der semantischen Äquivalenz ist im Folgenden von Bedeutung und soll hier formal definiert werden. Es wird die Definition aus [Win93] verwendet. Winskel nennt diese die „natürliche semantische Äquivalenzrelation“.

Definition 9.27. *Die* semantische Äquivalenzrelation $\sim \subseteq \mathbb{C} \times \mathbb{C}$ *ist definiert durch* $c \sim c'$ *genau dann, wenn*

$$\forall \sigma, \sigma' \in \Sigma : (\langle c, \sigma \rangle \rightarrow \sigma' \iff \langle c', \sigma \rangle \rightarrow \sigma').$$

Diese Definition wird für die vorliegende Arbeit mit dem erweiterten Zustandsbegriff übernommen. Sie umschließt damit auch Ein- und Ausgabeanweisungen, die zusammengesetzten Anweisungen und insbesondere Sequenzen von Ein- und Ausgabeanweisungen.

Es handelt sich bei der semantischen Äquivalenzrelation $\sim$ um eine Äquivalenzrelation, da sie offensichtlich reflexiv, symmetrisch und transitiv ist.

Lemma 9.28. *Zwei semantisch äquivalente Anweisungen lassen sich gegeneinander in beliebigen Sequenzen austauschen, ohne die Semantik der gesamten Sequenz zu verändern. Formal gilt für alle* $c_0, c_1, c_1', c_2 \in \mathbb{C}$*, dass*

$$c_1 \sim c_1' \implies c_0; c_1; c_2 \sim c_0; c_1'; c_2.$$

Beweis. Es ist zu zeigen, dass für semantische äquivalente Anweisungen c_1 und c_1' die beiden Sequenzen $c_0; c_1; c_2$ und $c_0; c_1'; c_2$ semantisch äquivalent sind, dass also bei Ausführung beider Sequenzen im selben Startzustand der gleiche Endzustand erreicht wird. Sei $\sigma \in \Sigma$ ein beliebiger Zustand. Es folgt für $c_0; c_1; c_2$ aus Definition 9.20, dass

$$\frac{\frac{\langle c_0,\sigma\rangle \rightarrow \sigma_1 \quad \langle c_1,\sigma_1\rangle \rightarrow \sigma_2}{\langle c_0;c_1,\sigma\rangle \rightarrow \sigma_2} \quad \langle c_2, \sigma_2 \rangle \rightarrow \sigma_3}{\langle c_0; c_1; c_2, \sigma \rangle \rightarrow \sigma_3}.$$

Analog ergibt sich für $c_0; c_1'; c_2$, dass

$$\frac{\frac{\langle c_0,\sigma\rangle \rightarrow \sigma_1 \quad \langle c_1',\sigma_1\rangle \rightarrow \sigma_2'}{\langle c_0;c_1',\sigma\rangle \rightarrow \sigma_2'} \quad \langle c_2, \sigma_2' \rangle \rightarrow \sigma_3'}{\langle c_0; c_1'; c_2, \sigma \rangle \rightarrow \sigma_3'}.$$

Aus $c_1 \sim c_1'$ folgt, dass σ_2 und σ_2' wegen $\langle c_1, \sigma_1 \rangle \rightarrow \sigma_2$ und $\langle c_1', \sigma_1 \rangle \rightarrow \sigma_2'$ identisch sein müssen. Damit wird die letzte Anweisung c_2 in beiden Sequenzen im gleichen Zustand ausgeführt und auch σ_3 und σ_3' müssen identisch sein. $\square$

Die Intention hinter der Einführung der Lesemarken ist, sicherzustellen, dass bestimmte Sequenzen von Anweisungen semantisch nicht äquivalent sind und zwar insbesondere solche, die Ein- und Ausgabeanweisungen enthalten. Durch die Modellierung der Eingabe als Funktion, ergibt sich in einigen Fällen eine solche Nicht-Äquivalenz bereits ohne Lesemarken, wie das folgende Beispiel verdeutlicht.

Sei $c_1 \equiv \texttt{read}(X); \texttt{write}(1)$ und $c_2 \equiv \texttt{write}(1); \texttt{read}(X)$. Ohne Lesemarken ergäbe sich, dass $c_1 \not\sim c_2$. Sei $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$. Durch Ausführung von c_1 und c_2 im Zustand σ würde jeweils der Ausgabestrom $1 \cdot \sigma_{out}$ entstehen. Allerdings ergäbe sich möglicherweise eine andere Variablenbelegung. Nach c_1 würde die Variable X den Wert $\sigma_{in}(\sigma_{out})$ haben. Während der Ausführung von c_2 würde allerdings der Ausgabestrom vor der `read`-Anweisung verändert, sodass X den Wert $\sigma_{in}(1 \cdot \sigma_{out})$ erhält, welcher für ein geeignetes σ_{in} von $\sigma_{in}(\sigma_{out})$ abweichen kann.

Dies ist ein Vorteil der Modellierung als Eingabefunktion gegenüber der als Eingabestrom. Mit einem Eingabestrom wären ohne Lesemarken die Anweisungssequenzen c_1 und c_2 aus dem obigen Beispiel semantisch äquivalent, solange nicht zusätzlich angenommen wird, dass der Eingabestrom nach einer Ausgabe ersetzt wird.

Die Lesemarken sind allerdings auch bei einer Modellierung mit Eingabefunktion unverzichtbar. Sie bewirken, dass die Anweisungen $c_1 \equiv \texttt{read}(X); \texttt{read}(Y)$ und ihre Vertauschung $c_2 \equiv \texttt{read}(Y); \texttt{read}(X)$ nicht semantisch äquivalent sind. Sei $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$ ein Zustand und σ_1 der Zustand, der sich aus der Ausführung von c_1 in σ ergibt, also $\langle \sigma, c_1 \rangle \to \sigma_1$. Analog sei σ_2 der zu c_2 gehörige Endzustand nach $\langle \sigma, c_2 \rangle \to \sigma_2$. Ohne Lesemarken würde gelten, dass die Ausgabekomponente von σ_1 gleich der von σ_2 und gleich σ_{out} ist. Folglich wäre auch $\sigma_1(X) = \sigma_1(Y) = \sigma_2(X) = \sigma_2(Y) = \sigma_{in}(\sigma_{out})$. Erst durch die Lesemarken ergibt sich, dass zwar $\sigma_1(X) = \sigma_2(Y) = \sigma_{in}(\sigma_{out})$ aber $\sigma_1(Y) = \sigma_2(X) = \sigma_{in}(R \cdot \sigma_{out})$ und damit, falls $\sigma_{in}(\sigma_{out}) \neq \sigma_{in}(R \cdot \sigma_{out})$ auch $\sigma_1(X) \neq \sigma_1(Y)$ beziehungsweise $\sigma_2(X) \neq \sigma_2(Y)$.

10 Transformation von Sequenzen atomarer Anweisungen

In diesem Kapitel werden Sequenzen atomarer Anweisungen und Transformationen dieser Sequenzen betrachtet, die die Semantik der Sequenz erhalten. Dabei interessieren, abgesehen von einem Vorverarbeitungsschritt, ausschließlich Permutationen der Sequenzen. Dabei handelt es sich um Sequenzen, die lediglich durch die Veränderung der Position der bestehenden Anweisungen entstehen. Es werden keine neuen Anweisungen hinzugefügt und auch keine weiteren eliminiert. Das Ziel dieses Kapitels ist die Ermittlung eines leicht überprüfbaren Kriteriums, mit dem die semantische Äquivalenz einer Permutation einer Sequenz atomarer Anweisungen nachgewiesen werden kann.

Dies geschieht in drei Schritten. Zunächst wird diskutiert, welche Anweisungen vorab eliminiert werden können. Dies dient der Vereinfachung der folgenden Betrachtung. Im Anschluss werden Sequenzen der Länge zwei betrachtet. Für diese wird diskutiert, unter welchen Umständen sich die beiden Anweisungen vertauschen lassen, ohne die Semantik der Sequenz zu ändern. Im letzten Schritt wird eine Relation definiert und gezeigt, dass alle Permutationen einer Eingabesequenz atomarer Anweisungen, die bezüglich der besagten Ordnung eine topologische Sortierung darstellen, die Semantik nicht verändern. Damit ist ein Kriterium gefunden, nämlich die Prüfung einer topologischen Sortierung.

10.1 Eliminierung einzelner Anweisungen

In diesem Abschnitt wird diskutiert, welche Anweisungen in einem Vorverarbeitungsschritt entfernt werden können.

Da durch die `skip`-Anweisung der Zustand nicht verändert wird, kann sie offensichtlich an beliebigen Stellen in ein Programm eingefügt werden. An vielen Stellen lässt sich die `skip`-Anweisung auch entfernen. Um im Folgenden die Überlegung nicht durch Einbeziehen der `skip`-Anweisung als weiteren Fall zu verkomplizieren,

ist es wünschenswert, alle `skip`-Anweisungen zu eliminieren. Aus Sequenzen ist dies möglich, indem sowohl `skip`; c als auch c; `skip` durch c ersetzt werden.

Es verbleiben die Fälle der beiden Formen der bedingten Anweisung und der Schleife. Eine bedingte Einzelanweisung ohne Alternativzweig der Form

$$\texttt{if}\ b\ \texttt{then skip}$$

ist, da die Auswertung von Ausdrücken seiteneffektfrei ist, äquivalent zur `skip`-Anweisung und kann durch diese ersetzt werden. Eine bedingte Anweisung mit Alternativzweig der Form

$$\texttt{if}\ b\ \texttt{then}\ c\ \texttt{else skip}$$

kann durch die Form ohne Alternativzweig ersetzt werden, also `if` b `then` c. Befindet sich das `skip` nicht im Alternativzweig, liegt also eine Anweisung der Form

$$\texttt{if}\ b\ \texttt{then skip else}\ c$$

vor, so kann dies mit der Negation der Bedingung b ebenfalls in die Form ohne Alternativzweig überführt werden und zwar zu

$$\texttt{if}\ \neg b\ \texttt{then}\ c.$$

Der letzte Fall, der der Schleife `while` b `do skip`, gestaltet sich am schwierigsten. Der Grund hierfür liegt in der Wiederholungssemantik. Wird der Ausdruck b initial zu `false` ausgewertet, ist die Ausführung der Schleife beendet. Die Anweisung ist damit effektiv äquivalent zur `skip`-Anweisung. Ist das Ergebnis der Auswertung hingegen `true`, so wird die Schleife fortgesetzt. Da aber weder die `skip`-Anweisung noch die Auswertung der Bedingung b den Zustand verändern, ergibt sich, dass die Schleife endlos fortgesetzt wird. Damit ist die Schleife nicht mehr äquivalent zu `skip`, sondern zu einem nicht-terminierenden Programm. Aufgrund der Unentscheidbarkeit des Halteproblems ist es allerdings nicht möglich, vorab zu entscheiden, zu welchen Wert die Bedingung b ausgewertet wird. Eine Ersetzung kann daher nicht abhängig von der Bedingung erfolgen. Allerdings kann die `skip`-Anweisung in einer Schleife durch die Anweisung $X := X$ ersetzt werden, da diese ebenfalls den Zustand nicht verändert (es gilt `skip` $\sim X := X$). Freilich liegt es damit nahe, jede `skip`-Anweisung durch $X := X$ zu ersetzen. Am Ende des Kapitels wird demonstriert werden, warum dies keine adäquate Alternative ist.

Mit den vorgestellten Überlegungen können in einem Vorverarbeitungsschritt alle

`skip`-Anweisungen aus einem Programm entfernt werden ohne die Semantik des Programms zu verändern. Die `skip`-Anweisung geht daher nicht in die folgenden Überlegungen mit ein, d.h. es wird davon ausgegangen, dass alle `skip`-Anweisungen eliminiert worden sind.

10.2 Sequenzen atomarer Anweisungen der Länge zwei

In diesem Abschnitt werden bestimmte Sequenzen aus zwei atomaren Anweisungen betrachtet. Dabei soll insbesondere betrachtet werden, unter welchen Bedingungen garantiert werden kann, dass die beiden Permutationen einer solchen Sequenz semantisch äquivalent sind. Daraus ergibt sich mit Lemma 9.28, dass zwei in einer beliebigen Sequenz direkt aufeinanderfolgenden Anweisungen immer vertauscht werden können, ohne die Semantik zu verändern. Die beiden folgenden Lemmata werden sich dabei als nützlich erweisen.

Lemma 10.1. *Sei $X \in \mathbb{L}$ eine Variable und $a \in \mathbb{E}_A$ ein Ausdruck, sodass X nicht in a vorkommt, also $X \notin \text{Vars}(a)$ gilt. Für jede beliebige ganze Zahl n und die beiden Zustände σ und $\sigma[n/X]$ folgt aus*

$$\langle a, \sigma \rangle \to m$$

$$\langle a, \sigma[n/X] \rangle \to m',$$

dass $m = m'$.

Beweis. Der Beweis erfolgt per Induktion über die Tiefe $D(a)$ des Ausdrucks a. Für die Tiefe $D(a) = 0$ muss a entweder ein Literal sein oder eine Variable Y. Falls es sich um eine Variable handelt, so gilt $X \neq Y$, da $X \notin \text{Vars}(a)$. Gemäß der Auswertungsregel 9.16 für arithmetische Ausdrücke gilt Lemma 10.1 für Ausdrücke der Tiefe 0.

Sei nun $D(a) = n > 0$. Wenn Lemma 10.1 für alle Ausdrücke a' mit einer Tiefe $D(a') < n$ gilt, folgt, dass die direkten Teilausdrücke von a in beiden Zuständen zu denselben Werten evaluieren. Nach den Auswertungsregeln für zusammengesetzte arithmetische Ausdrücke in 9.16 ist damit auch das Resultat von a in den Zuständen σ und $\sigma[n/X]$ gleich. □

Das zweite Lemma betrifft die Reihenfolge, in der die Werte von Variablen geändert werden.

Lemma 10.2. *Seien $X, Y \in \mathbb{L}$ unterschiedliche Variablen, also $X \neq Y$. Für beliebige ganze Zahlen $m, n \in \mathbb{Z}$ und einen beliebigen Zustand $\sigma \in \Sigma$ gilt*

$$\sigma[m/X][n/Y] = \sigma[n/Y][m/X].$$

Beweis. Aus der Definition der Variablensubstitution 9.15 folgt

$$\sigma[m/X][n/Y](V) = \begin{cases} n & \text{falls } V = Y \\ \sigma[m/X](V) & \text{falls } V \neq Y \end{cases}$$

$$\sigma[m/X](V) = \begin{cases} m & \text{falls } V = X \\ \sigma(V) & \text{falls } V \neq X. \end{cases}$$

Da $X \neq Y$ kann dies in einer Funktionsdefinition zusammengefasst werden

$$\sigma[m/X][n/Y](V) = \begin{cases} n & \text{falls } V = Y \\ m & \text{falls } V = X \\ \sigma(V) & \text{sonst.} \end{cases}$$

Analog ergibt sich für $\sigma[n/Y][m/X]$, dass

$$\sigma[n/Y][m/X](V) = \begin{cases} n & \text{falls } V = Y \\ m & \text{falls } V = X \\ \sigma(V) & \text{sonst.} \end{cases}$$

Beide Variablenbelegungen und damit beide Zustände sind daher identisch. □

Im Folgenden werden nun drei Sequenzen aus je zwei atomaren Anweisungen c_1 und c_2 betrachtet. Die Sequenzen sind dementsprechend $c_1; c_2$ bzw. die Vertauschung $c_2; c_1$. Beide Sequenzen werden in demselben Startzustand σ ausgeführt. Es ergeben sich zwei Zwischen- und Endzustände. Auch in dem Fall, dass $c_1; c_2 \sim c_2; c_1$ sind die beiden Zwischenzustände nicht notwendigerweise identisch. Es gilt für $\sigma_1; \sigma_2$, dass

$$\frac{\langle c_1, \sigma\rangle \to \sigma_1 \quad \langle c_2, \sigma_1\rangle \to \sigma_{1,2}}{\langle c_1; c_2, \sigma\rangle \to \sigma_{1,2}}$$

und für $\sigma_2; \sigma_1$, dass

$$\frac{\langle c_2, \sigma\rangle \to \sigma_2 \quad \langle c_1, \sigma_2\rangle \to \sigma_{2,1}}{\langle c_2; c_1, \sigma\rangle \to \sigma_{2,1}}.$$

Gilt für alle Zustände σ, dass die beiden Endzustände $\sigma_{1,2}$ und $\sigma_{2,1}$ gleich sind, so sind die beiden Sequenzen nach Definition 9.27 semantisch äquivalent.

Die drei nun behandelten Fälle ergeben sich jeweils aus einer Zuweisung $c_1 \equiv X := a$ sowie einmal einer weiteren Zuweisung, einer `write`-Anweisung und einer `read`-Anweisung für c_2. Es wird untersucht, unter welchen Umständen die beiden Permutationen semantisch äquivalent sind.

In allen Fällen heißt der Anfangszustand $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$. Wir beginnen mit $c_2 \equiv \texttt{write}(b)$.

Lemma 10.3. *Sei $c_1 \equiv X := a$ und $c_2 \equiv \texttt{write}(b)$. Falls die Variable X nicht in* $\mathrm{Vars}(b)$ *enthalten ist, gilt $c_1; c_2 \sim c_2; c_1$.*

Beweis. Dieser und die weiteren beiden Beweise ergeben sich aus der Betrachtung der Zwischen- und Endzustände beider Sequenzen. Wenn im Startzustand σ die Ausdrücke a und b gemäß

$$\langle a, \sigma \rangle \to m,$$
$$\langle b, \sigma \rangle \to n$$

zu m und n ausgewertet werden, dann folgt für die Zwischenzustände σ_1 und σ_2, dass

$$\sigma_1 = (\sigma_V[m/X], \sigma_{in}, \sigma_{out}),$$
$$\sigma_2 = (\sigma_V, \sigma_{in}, n \cdot \sigma_{out}).$$

Wegen der Voraussetzung $X \notin \mathrm{Vars}(b)$ folgt nach Lemma 10.1, dass $\langle b, \sigma_V[m/X] \rangle \to n$. Somit gilt

$$\sigma_{1,2} = (\sigma_V[m/X], \sigma_{in}, n \cdot \sigma_{out}) = \sigma_{2,1}.$$

□

Es folgt die Betrachtung einer Zuweisung und einer `read`-Anweisung.

Lemma 10.4. *Sei $c_1 \equiv X := a$ und $c_2 \equiv \texttt{read}(Y)$. Falls die Variablen X und Y unterschiedlich sind und zusätzlich $Y \notin$* $\mathrm{Vars}(a)$*, so gilt $c_1; c_2 \sim c_2; c_1$.*

Beweis. Es wird angenommen, dass

$$\langle a, \sigma \rangle \to m,$$
$$\sigma_{in}(\sigma_{out}) = n.$$

Als Zwischenzustände ergeben sich

$$\sigma_1 = (\sigma_V[m/X], \sigma_{in}, \sigma_{out}),$$
$$\sigma_2 = (\sigma_V[n/Y], \sigma_{in}, R \cdot \sigma_{out}).$$

Nach Voraussetzung gilt $Y \notin \mathrm{Vars}(a)$ und somit nach Lemma 10.1 $\langle a, \sigma_V[n/Y]\rangle \to m$ auch

$$\sigma_{1,2} = (\sigma_V[m/X][n/Y], \sigma'_{in}, R \cdot \sigma_{out}),$$
$$\sigma_{2,1} = (\sigma_V[n/Y][m/X], \sigma'_{in}, R \cdot \sigma_{out}).$$

Ebenfalls nach Voraussetzung gilt $Y \neq X$ und somit folgt aus Lemma 10.2, dass $\sigma_V[m/X][n/Y] = \sigma_V[n/Y][m/X]$. Es folgt $\sigma_{1,2} = \sigma_{2,1}$. □

Den letzten Fall stellen zwei Zuweisungen dar.

Lemma 10.5. *Sei $c_1 \equiv X := a$ und $c_2 \equiv Y := b$. Falls die Variablen X und Y unterschiedlich sind, $Y \notin \mathrm{Vars}(a)$ und $X \notin \mathrm{Vars}(b)$, so gilt $c_1; c_2 \sim c_2; c_1$.*

Beweis. Es wird angenommen, dass im Startzustand σ die beiden Ausdrücke a und b gemäß

$$\langle a, \sigma\rangle \to m \text{ und}$$
$$\langle b, \sigma\rangle \to n$$

ausgewertet werden. Somit ergeben sich die Zwischenzustände als

$$\sigma_1 = (\sigma_V[m/X], \sigma_{in}, \sigma_{out}) \text{ und}$$
$$\sigma_2 = (\sigma_V[n/Y], \sigma_{in}, \sigma_{out}).$$

Da $X \notin \mathrm{Var}(b)$ und $Y \notin \mathrm{Var}(a)$, folgt aus Lemma 10.1, dass $\langle b, \sigma_V[m/X]\rangle \to n$ und $\langle a, \sigma_V[n/Y]\rangle \to m$. Somit ist

$$\sigma_{1,2} = (\sigma_V[m/X][n/Y], \sigma_{in}, \sigma_{out}) \text{ und}$$
$$\sigma_{2,1} = (\sigma_V[n/Y][m/X], \sigma_{in}, \sigma_{out}).$$

Nach Lemma 10.2 ist damit $\sigma_{1,2} = \sigma_{2,1}$, da $X \neq Y$. □

10.3 Permutationen von beliebig langen Sequenzen atomarer Anweisungen

Alle weiteren Überlegungen sind unabhängig von der operationellen Semantik und folgen aus den Eigenschaften der semantischen Äquivalenzrelation sowie den konkreten Äquivalenzen aus den Lemmata 10.3, 10.4 und 10.5.

Die naheliegende Erweiterung der Ergebnisse über die Vertauschung zweier Anweisungen ist die Betrachtung mehrerer Anweisungen. In diesem Abschnitt sei daher $S \equiv c_1; c_2 \ldots c_n$ eine beliebige aber feste Sequenz von atomaren Anweisungen, also

$c_i \in \mathbb{C}_{atomar}$ für $1 \leq i \leq n$. Im Folgenden werden benachbarte Anweisungen in dieser Sequenz ausgetauscht. Gelten die Vorbedingungen aus den Lemmata 10.3, 10.4 und 10.5 für die vertauschten Anweisungen, so verändert sich nach Lemma 9.28 die Semantik der gesamten Sequenz nicht. Sei S' die Sequenz, die aus S entsteht, wenn die Anweisung an einer Stelle i mit ihrem Nachfolger getauscht wird, also $S' \equiv c_1 \dots c_{i+1}; c_i \dots c_n$ für $1 \leq i < n$. Formal gilt, dass $S \sim S'$, falls für c_i und c_{i+1} gezeigt werden kann, dass $c_i; c_{i+1} \sim c_{i+1}; c_i$. Dieses Argument lässt sich auch auf eine Sequenz S'', die durch die Vertauschung zweier benachbarter Anweisungen in S' entsteht, anwenden. Aufgrund der Transitivität von $\sim$ gilt dann nicht nur $S' \sim S''$ sondern wegen $S \sim S'$ auch $S \sim S''$.

Das Ziel der Programmtransformation ist, eine für die Terminierungsanalyse vorteilhafte Variation eines gegebenen Programms zu erstellen. Die Überlegungen in Abschnitt 10.2 erlauben es, semantisch äquivalente Variationen zu erzeugen. In der Menge dieser Variationen kann nach den Varianten gesucht werden, die die längsten Teilsequenzen von Anweisungen ohne Ein- oder Ausgabeanweisungen enthalten. Dieser Ansatz birgt allerdings folgendes Problem. Ließen sich in einer Sequenz S der Länge n alle Anweisungen miteinander vertauschen, so ergäben sich $n!$ Permutationen. Die Menge der Variationen ist daher möglicherweise zu groß für eine effiziente Terminierungsanalyse. Im Folgenden wird daher ein zweiter Ansatz beschrieben. Bisher wurden semantisch äquivalente Permutationen erzeugt und auf bestimmte, für die Terminierungsanalyse vorteilhafte Kriterien überprüft. Alternativ ist denkbar, Permutationen mit den gewünschten Eigenschaften bezüglich der Terminierungsanalyse ohne Rücksicht auf die semantische Äquivalenz zu erzeugen, und dann mit einem effizienten Verfahren auf semantische Äquivalenz zu prüfen.

Es soll also für eine feste Sequenz S ein Kriterium aufgestellt werden, welches erlaubt effizient zu überprüfen, ob eine Permutation S' semantisch äquivalent zu S ist. Dazu wird in diesem Abschnitt eine Relation $\lhd_S$ bestimmt, sodass jede Sequenz der Anweisungen $c_1, \dots, c_n$, die ein topologische Sortierung bezüglich $\lhd_S$ darstellt, semantisch äquivalent zu S ist.

Definition 10.6. *Sei S eine Sequenz der Form $c_1; c_2 \dots c_n$ mit $c_i \in \mathbb{C}_{atomar}$ für alle $1 \leq i \leq n$. Die* Folge der Anweisungen aus S *wird mit $\vec{S}$ bezeichnet, die* Menge der Anweisungen in S *als $\mathbb{S}$ und es gelten*

$$\vec{S} = \langle c_1, c_2 \dots c_n \rangle \textit{ sowie } \mathbb{S} = \{c_1, c_2 \dots c_n\}.$$

Die atomaren Anweisungen der Sequenz S werden als Vorkommen von Anweisungen betrachtet. Die Anweisungen sind daher paarweise unterschiedlich und es gilt

$c_i \neq c_j$ für $i \neq j$. Für eine Sequenz der Länge n zählt die Menge $\mathbb{S}$ dementsprechend n Elemente. Es ist jedoch nicht ausgeschlossen, dass syntaktische Gleichheiten der Form $c_i \equiv c_j$ für unterschiedliche Indizes i und j bestehen.

10.3.1 Abhängigkeiten zwischen Anweisungen

In diesem Abschnitt werden die Abhängigkeiten zwischen den Anweisungen aus einer Sequenz als Mengen beschrieben. Aufbauend auf den Konzepten der gelesenen und der veränderten Variablen, lässt sich beschreiben, welche Anweisungen vor einer Anweisung ausgeführt werden müssen. Mit den folgenden Definitionen lässt sich $\lhd_S$ bereits definieren. In Abschnitt 10.3.2 wird jedoch eine alternative Definition gegeben und gezeigt, dass beide äquivalent sind.

Definition 10.7. *Mit $\preceq_S$ wird die totale Halbordnung auf den atomaren Anweisungen aus $\mathbb{S}$*

$$c_i \preceq_S c_j \iff i \leq j$$

notiert. Mit $\prec_S$ wird die totale Striktordnung

$$c_i \prec c_j \iff i < j$$

notiert. Es gilt $\preceq_S, \prec_S \subseteq \mathbb{S} \times \mathbb{S}$.

Die Ordnungseigenschaften von $\preceq_S$ und $\prec_S$ folgen aus den Ordnungseigenschaften von $\leq$ und $<$ auf den natürlichen Zahlen.

Definition 10.8. *Die Menge der* gelesenen Variablen *in einer atomaren Anweisung c ist definiert als*

$$\text{Used}(c) = \begin{cases} \text{Vars}(a), & \textit{falls } c \equiv X := a, \\ \text{Vars}(a), & \textit{falls } c \equiv \texttt{write}(a), \\ \emptyset, & \textit{falls } c \equiv \texttt{read}(X). \end{cases}$$

Definition 10.9. *Die Menge der* veränderten Variablen *in einer atomaren Anweisung c ist definiert als*

$$\text{Modified}(c) = \begin{cases} \{X\}, & \textit{falls } c \equiv X := a, \\ \{X\}, & \textit{falls } c \equiv \texttt{read}(X), \\ \emptyset, & \textit{falls } c \equiv \texttt{write}(a). \end{cases}$$

Definition 10.10. *Das Prädikat* EA(c) *beschreibt Ein- und Ausgabeanweisungen formal und wird definiert als*

$$\mathrm{EA}(c) \iff (c \equiv \texttt{write}(a)) \vee (c \equiv \texttt{read}(a)).$$

Alle folgenden Definitionen beziehen sich implizit auf die Sequenz S.

Definition 10.11. *Die Menge der* vorhergehenden Anweisungen *für eine Anweisung c in der Sequenz S ist wird bezeichnet mit* Prev(c) *und ist definiert als*

$$\mathrm{Prev}(c) = \{c' \in \mathbb{S} : c' \prec c\}.$$

Die Menge der vorhergehenden Anweisungen wird nun eingeschränkt. Die folgenden Definitionen beschreiben Mengen von Anweisungen bezüglich einer Anweisung c, die sich in der Sequenz S jeweils vor c befinden und bestimmte Eigenschaften aufweisen. Es wird begonnen mit den vorhergehenden Ein- und Ausgabeanweisungen.

Definition 10.12. *Die Menge der* vorhergehenden Ein- und Ausgabeanweisungen *für eine Anweisung c ist die Teilmenge der c vorhergehenden Ein- und Ausgabeanweisungen. Diese Menge wird mit* $\mathrm{Prev}^{\mathrm{EA}}(c)$ *bezeichnet und ist definiert als*

$$\mathrm{Prev}^{\mathrm{EA}}(c) = \{c' \in \mathrm{Prev}(c) : \mathrm{EA}(c')\}.$$

Für $\mathrm{Prev}^{\mathrm{EA}}(c)$ ist lediglich die Position von c in S, aber keine andere Eigenschaft relevant. In den Definitionen 10.13, 10.14 und 10.15 ist dies nicht mehr der Fall, sondern es werden tatsächliche oder mögliche Interaktionen untersucht.

Definition 10.13. *Die Menge der* definierenden Anweisungen *für eine Anweisung c, ist die Teilmenge der vorhergehenden Anweisungen, die einer Variablen einen Wert zuweisen, die in c gelesen wird. Die Menge wird mit* Def(c) *bezeichnet und ist definiert als*

$$\mathrm{Def}(c) = \{c' \in \mathrm{Prev}(c) : \mathrm{Used}(c) \cap \mathrm{Modified}(c') \neq \emptyset\}.$$

Die Anweisungen in Def(c) sind daher für die Anweisung c möglicherweise relevant, die Anweisungen aus Def(c) sollten also vor c ausgeführt werden. Die nächsten beiden Definitionen erfassen umgekehrt Anweisungen, welche nicht nach c ausgeführt werden sollten.

Definition 10.14. *Die Menge der* zuweisenden Anweisungen *für eine Anweisung c, ist die Teilmenge der vorhergehenden Anweisungen, die einer Variablen einen Wert*

zuweisen, der in c ebenfalls ein Wert zugewiesen wird. Die Menge wird mit Assign(c) *bezeichnet und ist definiert als*

$$Assign(c) = \{c' \in \mathrm{Prev}(c) : \mathrm{Modified}(c) \cap \mathrm{Modified}(c') \neq \emptyset\}.$$

Definition 10.15. *Die Menge der* verfälschten Anweisungen *für eine Anweisung c, ist die Teilmenge der vorhergehenden Anweisungen, die eine Variablen verwenden, der in c ein Wert zugewiesen wird. Die Menge wird mit* Anti(c) *bezeichnet und ist definiert als*

$$\mathrm{Anti}(c) = \{c' \in \mathrm{Prev}(c) : \mathrm{Modified}(c) \cap \mathrm{Used}(c') \neq \emptyset\}.$$

Bei der Betrachtung der Definitionen von Def, Assign und Anti fällt auf, dass keine Menge für c definiert wird, die der Form

$$\{c' \in \mathrm{Prev}(c) : \mathrm{Used}(c) \cap \mathrm{Used}(c') \neq \emptyset\}$$

entspricht. Tatsächlich ähneln die Definitionen in der vorliegenden Arbeit den sogenannten Abhängigkeitsrelationen, die von Kuck et al. in [KKP+81] definiert werden. Dort findet sich auch eine Relation δ^I, für die es in der vorliegenden Arbeit aber keine Entsprechung gibt. Umgekehrt, da Kuck et al. Ein- und Ausgabe in ihrer Arbeit nicht berücksichtigen, findet sich dort kein Gegenstück zu $\mathrm{Prev}^{\mathrm{EA}}$.

Jede der Teilmengen aus den Definitionen 10.12, 10.13, 10.14 und 10.15 beschreiben Mengen von Anweisungen in Abhängigkeit von einer Anweisung c, die in S vor c stehen. Ferner ist es leicht, Beispiele für S so zu finden, dass, wenn eine Anweisung c' aus einer dieser Mengen in einer Permutation von S nach der Anweisung c steht, die Permutation nicht mehr semantisch äquivalent zu S ist. Die Teilmengen werden daher zusammengefasst.

Definition 10.16. *Die Menge der* vorausgesetzten Anweisungen *für eine Anweisung c wird bezeichnet mit* Req(c) *und ist definiert als*

$$\mathrm{Req}(c) = \begin{cases} \mathrm{Def}(c) \cup \mathrm{Assign}(c) \cup \mathrm{Anti}(c) \cup \mathrm{Prev}^{\mathrm{EA}}(c), & \textit{falls } \mathrm{EA}(c) \\ \mathrm{Def}(c) \cup \mathrm{Assign}(c) \cup \mathrm{Anti}(c), & \textit{falls } \neg\,\mathrm{EA}(c). \end{cases}$$

Eine Anweisung c' heißt von c vorausgesetzte Anweisung *genau dann, wenn* $c' \in \mathrm{Req}(c)$.

Auf Basis der vorausgesetzten Anweisungen lässt sich $\lhd_S$ beschreiben als

$$c_i \lhd_S c_j \iff c_i \in \mathrm{Req}_S(c_j).$$

Da für alle c gilt, das $\mathrm{Req}_S(c) \subseteq \mathrm{Prev}(c)$, ist $\lhd_S$ zyklenfrei. Wie noch zu zeigen ist, ist $\lhd_S$ geeignet, die Prüfung der semantischen Äquivalenz auf die Prüfung der topologischen Sortierung zu reduzieren.

10.3.2 Kopplung κ und Abhängigkeit $\lhd_S$

In diesem Abschnitt wird die Relation $\lhd_S$ auf eine andere Weise definiert. Die neue Definition ist äquivalent zu der vorhergehenden, bietet allerdings den Vorteil, dass sich einige Beweise leichter formulieren lassen. Dabei wird $\lhd_S$ definiert als Schnitt aus $\prec_S$ und einer weiteren Relation κ, also $\lhd_S = \prec_S \cap \kappa$. Dabei ist κ unabhängig von der Sequenz S und es gilt dementsprechend $\kappa \subseteq \mathbb{C}_{atomar} \times \mathbb{C}_{atomar}$.

Ähnlich wie zuvor die Menge der vorausgesetzten Anweisungen aus Teilmengen aufgebaut wurde, wird die Relation κ aus Teilrelationen aufgebaut. Es wird wieder mit der Ein- und Ausgabe begonnen.

Definition 10.17. *Die Relation $\kappa_\mathrm{E} \subseteq \mathbb{C}_{atomar} \times \mathbb{C}_{atomar}$ heißt* EA-Kopplung *und ist definiert als*

$$c\,\kappa_\mathrm{E}\,d \iff \mathrm{EA}(c) \wedge \mathrm{EA}(d).$$

Als Kurznotation für $(c,d) \notin \kappa_\mathrm{E}$ wird $c\,\hat{\kappa}_\mathrm{E}\,d$ geschrieben.

Die Relation κ_E ist eine Entsprechung der Menge $\mathrm{Prev}_S^{\mathrm{EA}}(c)$ aus Definition 10.12 in dem Sinne, dass gilt

$$\mathrm{Prev}_S^{\mathrm{EA}}(c) = \{c' \in \mathbb{S} : c' \prec c \wedge c'\,\kappa_\mathrm{E}\,c\}.$$

Definition 10.18. *Die Relation $\kappa_\mathrm{M} \subseteq \mathbb{C}_{atomar} \times \mathbb{C}_{atomar}$ heißt* Modifikationskopplung *und ist definiert als*

$$c\,\kappa_\mathrm{M}\,d \iff \mathrm{Modified}(c) \cap \mathrm{Modified}(d) \neq \emptyset.$$

Als Kurznotation für $(c,d) \notin \kappa_\mathrm{M}$ wird $c\,\hat{\kappa}_\mathrm{M}\,d$ geschrieben.

Die Relation κ_M ist eine Entsprechung der Menge $\mathrm{Assign}_S(c)$ aus Definition 10.14 in dem Sinne, dass gilt

$$\mathrm{Assign}_S(c) = \{c' \in \mathbb{S} : c' \prec c \wedge c'\,\kappa_\mathrm{M}\,c\}.$$

Definition 10.19. *Die Relation* $\kappa_{\mathrm{V}} \subseteq \mathbb{C}_{atomar} \times \mathbb{C}_{atomar}$ *heißt* Verarbeitungskopplung *und ist definiert als*

$$c\,\kappa_{\mathrm{V}}\,d \iff \mathrm{Used}(c) \cap \mathrm{Modified}(d) \neq \emptyset \vee \mathrm{Modified}(c) \cap \mathrm{Used}(d) \neq \emptyset.$$

Als Kurznotation für $(c, d) \notin \kappa_{\mathrm{V}}$ *wird* $c\,\hat{\kappa}_{\mathrm{V}}\,d$ *geschrieben.*

Die Relation κ_{V} ist eine Entsprechung der Mengen $\mathrm{Def}_S(c)$ und $\mathrm{Anti}_S(c)$ in dem Sinne, dass gilt

$$\mathrm{Def}_S(c) \cup \mathrm{Anti}_S(c) = \{c' \in \mathbb{S} : c' \prec c \wedge c'\,\kappa_{\mathrm{V}}\,c\}$$

Mit den drei Relationen κ_{E}, κ_{M} und κ_{V} lässt sich die Kopplung definieren.

Definition 10.20. *Die Relation* $\kappa \subseteq \mathbb{C}_{atomar} \times \mathbb{C}_{atomar}$ *heißt* Kopplung *und ist definiert als*

$$\kappa = \kappa_{\mathrm{E}} \cup \kappa_{\mathrm{M}} \cup \kappa_{\mathrm{V}}\,.$$

Die Anweisungen c und d werden als gekoppelte Anweisungen *bezeichnet. Als Kurznotation für* $(c, d) \notin \kappa$ *wird* $c\,\hat{\kappa}\,d$ *geschrieben.*

Die Relation κ ist, wie bereits erwähnt, nicht nur auf $\mathbb{S}$ definiert und damit unabhängig von einem konkreten Eingabeprogramm. Ferner ist κ symmetrisch, da $\cap$ und $\wedge$ kommutativ sind. Diese Eigenschaften unterscheiden die Kopplung von den asymmetrischen Abhängigkeitsrelationen δ in [KKP+81]. Bereits der Begriff Abhängigkeit ist für κ nicht angemessen, da κ keine Reihenfolge festlegt. Die asymmetrische Abhängigkeit ist in [KKP+81] ein untrennbares Geflecht aus der symmetrischen Kopplung mit der Ordnung der Anweisungen in der Sequenz S. In der vorliegenden Arbeit werden diese Elemente erst später verbunden. Zunächst soll eine andere Eigenschaft von κ beschrieben werden.

Lemma 10.21. *Die Relation κ ist nicht transitiv.*

Beweis. Sei $c_1 \equiv A := B$, $c_2 \equiv B := C$ und $c_3 \equiv C := D$. Es gilt $c_1\,\kappa\,c_2$, da $c_1\,\kappa_{\mathrm{V}}\,c_2$, denn $B \in \mathrm{Modified}(c_2) \cap \mathrm{Used}(c_1)$. Analog gilt $c_2\,\kappa\,c_3$, da $c_2\,\kappa_{\mathrm{V}}\,c_3$. Allerdings gilt $c_1\,\hat{\kappa}\,c_3$, denn die Mengen $\mathrm{Used}(c_1) = \{B\}, \mathrm{Modified}(c_1) = \{A\}, \mathrm{Used}(c_3) = \{D\}, \mathrm{Modified}(c_3) = \{C\}$ sind paarweise disjunkt und somit $c_1\,\hat{\kappa}\,c_3$. □

Das Gegenbeispiel aus dem Beweis von Lemma 10.21 kommt mit Zuweisungen aus. Es demonstriert daher auch, dass die Transitivität nicht erst durch die Einführung

von Ein- und Ausgabeoperationen in die Sprache **IMP** gebrochen wurde. Die Zuweisungen sind in dem Beispiel indiziert, es ist jedoch nicht notwendig anzunehmen, dass $S \equiv c_1; c_2; c_3$ gilt, da die Kopplung unabhängig von konkreten Programmen definiert ist. Nachdem die Eigenschaften von κ erörtert wurden, kann der Begriff der Abhängigkeit eingeführt werden.

Definition 10.22. *Die Relation $\lhd_S$ heißt* Abhängigkeitsrelation *der Sequenz S und wird definiert als*

$$\lhd_S = \prec_S \cap \, \kappa \, .$$

Es gilt $\lhd_S \subseteq \mathbb{S} \times \mathbb{S}$. Ist die Sequenz S aus dem Kontext ersichtlich, kann der Index entfallen.

Durch die Definition von $\lhd_S$ als Schnitt von $\prec_S$ und κ, geht die Eigenschaft der Symmetrie von κ in $\lhd_S$ verloren, da $\prec_S$ nicht symmetrisch ist. Dies gilt ebenso für die Transitivität von $\prec_S$. Dennoch bietet diese Definition den Vorteil, dass für zwei Anweisungen getrennt betrachtet werden kann, in welcher Reihenfolge sie stehen und ob sie gekoppelt sind. In den getrennten Betrachtungen kann jeweils mit Symmetrie oder Transitivität argumentiert werden. Ferner ist dieses Vorgehen von Vorteil, um für zwei Anweisungen nachzuweisen, dass sie nicht voneinander abhängig sind, da es ausreicht zu zeigen, dass sie entweder nicht gekoppelt sind oder in umgekehrter Reihenfolge vorkommen.

Bemerkung 10.23. *Die Relation $\lhd_S$ ist nicht transitiv.*

Beweis. Sei $S \equiv c_1; c_2; c_3$ und $c_1 \, \kappa \, c_2$, $c_2 \, \kappa \, c_3$ und $c_1 \, \hat{\kappa} \, c_3$. Die Existenz solcher bezüglich κ nicht transitiver Anweisungstripel folgt aus Lemma 10.21. Wegen $c_1 \prec_S c_2$ gilt $c_1 \lhd_S c_2$ und wegen $c_2 \prec_S c_3$ auch $c_2 \lhd_S c_3$. Da $c_1 \, \hat{\kappa} \, c_3$ gilt aber $c_1 \not\lhd_S c_3$. □

Es bleibt zu zeigen, dass die neue Definition von $\lhd$ in 10.22 äquivalent zu der vorhergehenden aus Abschnitt 10.3.1 ist.

Bemerkung 10.24. *Diese Definition ist äquivalent zu der Definition aus Abschnitt 10.3.1. Dort wurde $\lhd_S$ über die Menge* Req_S *aus Definition 10.16 definiert als*

$$c \lhd_S d \iff c \in \mathrm{Req}_S(d)$$

für zwei Anweisungen c und d aus $\mathbb{S}$.

Beweis. Zu zeigen ist, dass $c \lhd_S d \iff c \in \mathrm{Req}_S(d)$. Wir beginnen mit der Implikation $c \lhd_S d \implies c \in \mathrm{Req}_S(d)$. Aus $c \lhd_S d$ folgt $c \prec d$ und somit $c \in$

$\mathrm{Prev}_S(d)$. Ferner gilt $c\,\kappa\,d$ wegen $c \lhd_S d$. Damit muss entweder $c\,\kappa_\mathrm{E}\,d$, $c\,\kappa_\mathrm{M}\,d$ oder $c\,\kappa_\mathrm{V}\,d$ gelten.

Im Fall einer EA-Kopplung $c\,\kappa_\mathrm{E}\,d$ gilt $\mathrm{EA}(c)$ und somit $c \in \mathrm{Prev}_S^{\mathrm{EA}}(d)$. Außerdem gilt $\mathrm{EA}(d)$ und somit $\mathrm{Req}_S(d) \supseteq \mathrm{Prev}_S^{\mathrm{EA}}(c)$ nach Definition von Req_S. Folglich ist $c \in \mathrm{Req}_S(d)$.

Im Fall einer Modifikationskopplung $c\,\kappa_\mathrm{M}\,d$ gilt $\mathrm{Modified}(c) \cap \mathrm{Modified}(d) \neq \emptyset$. Damit ist $c \in \mathrm{Assign}_S(d)$ und auch $c \in \mathrm{Req}_S(d)$.

Im Fall einer Verarbeitungskopplung $c\,\kappa_\mathrm{V}\,d$ ist der Schnitt $\mathrm{Used}(c) \cap \mathrm{Modified}(d)$ oder der Schnitt $\mathrm{Modified}(c) \cap \mathrm{Used}(d)$ nicht leer. Ersteres führt zu $c \in \mathrm{Def}_S(d)$, letzteres zu $c \in \mathrm{Anti}_S(d)$ oder beidem. Es folgt, dass $c \in \mathrm{Req}_S(d)$.

Es verbleibt die Richtigkeit der Gegenrichtung zu zeigen. Aus $c \in \mathrm{Req}_S(d)$ ergibt sich notwendigerweise $c \prec_S d$. Ansonsten wäre $c \notin \mathrm{Prev}(d)$ und somit auch nicht in $\mathrm{Req}_S(d)$ enthalten. Damit muss nur noch $c\,\kappa\,d$ belegt werden. Dazu reicht es zu zeigen, dass $c\,\kappa_\mathrm{E}\,d$, $c\,\kappa_\mathrm{M}\,d$ oder $c\,\kappa_\mathrm{V}\,d$ gilt.

Falls d eine Ein- oder Ausgabeanweisung ist, kann c in $\mathrm{Prev}_S^{\mathrm{EA}}(d)$ liegen. Aus $c \in \mathrm{Prev}_S^{\mathrm{EA}}(c)$ folgt $\mathrm{EA}(c)$ und somit gilt $c\,\kappa_\mathrm{E}\,d$.

Unabhängig von $\mathrm{EA}(d)$ kann c auch in $\mathrm{Def}_S(d)$, $\mathrm{Assign}_S(d)$ oder $\mathrm{Anti}_S(d)$ liegen. Dabei impliziert $c \in \mathrm{Assign}_S(d)$, dass $c\,\kappa_\mathrm{M}\,d$, da $\mathrm{Modified}(c) \cap \mathrm{Modified}(d) \neq \emptyset$ gilt. Hingegen folgt aus $c \in \mathrm{Def}_S(d)$ oder $c \in \mathrm{Anti}_S(d)$, dass $c\,\kappa_\mathrm{V}\,d$ gilt, da nun $\mathrm{Used}(d) \cap \mathrm{Modified}(c) \neq \emptyset$ oder $\mathrm{Modified}(d) \cap \mathrm{Used}(c) \neq \emptyset$ gelten muss. □

Es ist also gelungen, die Relation $\lhd$ als Schnitt zweier Relationen zu beschreiben, wobei beide Relationen jeweils eine Eigenschaft besitzen, nämlich Symmetrie von κ und Transitivität von $\prec$, die sich in den folgenden Überlegung als vorteilhaft erweist.

10.3.3 Ein Zusammenhang zwischen $\lhd_S$ und semantischer Äquivalenz

Die Relation $\lhd_S$ soll zur Prüfung der semantischen Äquivalenz einer Permutation S' bezüglich der Sequenz S dienen. Als Kriterium wird herangezogen, ob S' eine topologische Sortierung bezüglich $\lhd_S$ darstellt.

Im Allgemeinen ist die Existenz einer topologischen Sortierung bezüglich einer Relation R nicht gesichert ist, falls R zyklisch ist. Zwar hat sich bereits herausgestellt, dass $\lhd_S$ weder symmetrisch noch transitiv ist, aber es kann gezeigt werden, dass $\lhd_S$ azyklisch ist.

Lemma 10.25. *Die Striktordnung $\prec$ ist azyklisch.*

Beweis. Wäre $\prec$ zyklisch, so gäbe es eine Folge von Indices $i_1, i_2, \ldots, i_n$, sodass $i_1 = i_n$, aber wegen der Transitivität von $<$ auch $i_1 < i_n$. □

Lemma 10.26. *Die Relation $\lhd$ ist azyklisch.*

Beweis. Da $\lhd \subseteq \prec$ folgt die Behauptung unmittelbar aus A.13. □

Ferner existiert mindestens eine topologische Sortierung bezüglich $\lhd$, nämlich die sich aus S ergebende.

Korollar 10.27. *Die zur Sequenz S gehörige Folge $\vec{S} = \langle c_1, c_2, \ldots, c_n \rangle$ ist eine topologische Sortierung bezüglich $\lhd$.*

Beweis. Zu zeigen ist, dass kein Paar von Anweisungen und c_i und c_j mit $1 \leq i < j \leq n$ in S vorkommt, sodass $c_j \lhd c_i$. Aus der Irreflexivität und Transitivität von $<$ folgt wegen $i < j$, dass $j \not< i$ und somit auch $c_j \not\prec c_i$. Wegen $\lhd = \prec \cap \kappa$, muss $c_j \ntriangleleft c_i$. □

Der Rest dieses Abschnitts zeigt, dass die Überprüfung der topologischen Sortierung bezüglich $\lhd_S$ tatsächlich als Kriterium für die semantische Äquivalenz S geeignet ist. Dazu muss gezeigt werden, dass jede topologische Sortierung bezüglich $\lhd_S$ semantisch äquivalent zu S ist. Dies gliedert sich in zwei Schritte. Zunächst wird gezeigt, dass durch die iterierte Vertauschung benachbarter Elemente in S, die nicht gekoppelt sind, eine semantisch äquivalente Sequenz zu S entsteht. Danach wird gezeigt, dass jede Permutation von S, die eine topologische Sortierung bezüglich $\lhd_S$ darstellt, durch iteriertes Vertauschen benachbarter, nicht gekoppelter Anweisungen konstruiert werden kann.

Lemma 10.28. *Die Sequenzen $c_i; c_j$ und $c_j; c_i$ sind für unterschiedliche atomare Anweisungen aus S semantisch äquivalent, wenn $c_i \, \hat{\kappa} \, c_j$ gilt.*

Beweis. Es ist zu zeigen, dass

$$\forall c_i, c_j \in \mathbb{S} \text{ mit } i \neq j : c_i \, \hat{\kappa} \, c_j \implies c_i; c_j \sim c_j; c_i.$$

Dies fordert offensichtlich eine Fallunterscheidung über die verschiedenen Formen atomarer Anweisungen. Durch eine Reihe von Vorüberlegungen können diverse Paare allerdings ausgeschlossen werden. Da sowohl $\sim$ als auch $\hat{\kappa}$ symmetrisch sind, ist es nicht notwendig zu jedem Fall auch den symmetrischen Fall zu betrachten. Ferner sind durch die Vorüberlegungen keine `skip`-Anweisungen zu betrachten. Es verbleiben daher nur drei Formen von atomaren Anweisungen: `write`, `read` und die

Zuweisung. Es ergeben sich somit sechs unterschiedliche Kombination von je zwei atomaren Anweisungen:

1. Zwei `read`-Anweisungen
2. Zwei `write`-Anweisungen
3. Zwei Zuweisungen
4. Eine `read`-Anweisung und `write`-Anweisung
5. Eine Zuweisung und ein `write`-Anweisungen
6. Eine Zuweisung und ein `read`-Anweisungen

Die Fälle 1., 2. *und* 4.*:* Die Voraussetzung $c_i \,\hat{\kappa}\, c_j$ ist in diesen Fällen nicht erfüllt. Es handelt sich um jeweils zwei Ein- und/oder Ausgabeanweisungen womit $\mathrm{EA}(c_i)$ als auch $\mathrm{EA}(c_j)$ gilt. Es folgt $c_i \,\kappa_{\mathrm{E}}\, c_j$ und somit $c_i \,\kappa\, c_j$.

Die verbleibenden drei Fälle lassen sich mit Hilfe der Lemma 10.3, 10.4 und 10.5 zeigen.

Der Fall 3.*:* Um die Vertauschbarkeit zweier Zuweisungen zu zeigen, wird Lemma 10.5 verwendet. Es bleibt zu zeigen, dass die Voraussetzungen des Lemmas erfüllt sind, falls $c_i \,\hat{\kappa}\, c_j$. Im einzelnen heißt das für die Anweisungen $c_i \equiv X := a$ und $c_j \equiv Y := b$, dass $X \neq Y$, $X \neq \mathrm{Vars}(b)$ und $Y \neq \mathrm{Vars}(a)$ gilt.

Falls $X = Y$, dann wäre $\mathrm{Modified}(c_i) = \{X\} = \{Y\} = \mathrm{Modified}(c_j)$. Damit wäre $\mathrm{Modified}(c_i) \cap \mathrm{Modified}(c_j) = \{X\} \neq \emptyset$ und somit auch $c_i \,\kappa_{\mathrm{M}}\, c_j$ im Widerspruch zu $c_i \,\hat{\kappa}\, c_j$. Es folgt $X \neq Y$ falls $c_i \,\hat{\kappa}\, c_j$.

Sei zum Zwecke des Widerspruchs $X \in \mathrm{Vars}(b)$. Damit wäre $X \in \mathrm{Used}(c_j)$ und somit $\mathrm{Modified}(c_i) \cap \mathrm{Used}(c_j) = \{X\} \neq \emptyset$. Es folgt $c_i \,\kappa_{\mathrm{V}}\, c_j$ und somit $c_i \,\kappa\, c_j$. Daher muss $X \neq \mathrm{Vars}(b)$. Analog folgt, dass die letzte Voraussetzung $Y \notin \mathrm{Vars}(a)$ für Lemma 10.5 erfüllt ist.

Der Fall 5.*:* In diesem Fall sei $c_i \equiv X := a$ und $c_j \equiv \mathtt{write}(b)$. Aus $c_i \,\hat{\kappa}\, c_j$ folgt $c_i \,\hat{\kappa}_{\mathrm{V}}\, c_j$ und somit $\mathrm{Modified}(c_i) \cap \mathrm{Used}(c_j) = \emptyset$. Es gilt nach Definition $\mathrm{Modified}(c_i) = \{X\}$ und $\mathrm{Used}(c_j) = \mathrm{Vars}(b)$. Folglich muss $\{X\} \cap \mathrm{Vars}(b) = \emptyset$ und damit $X \notin \mathrm{Vars}(b)$ gelten. Gemäß Lemma 10.3 gilt damit $c_i; c_j \sim c_j; c_i$.

Der Fall 6.*:* In diesem Fall sei $c_i \equiv X := a$ und $c_j \equiv \mathtt{read}(Y)$. Es gilt also $\mathrm{Used}(c_i) = \mathrm{Vars}(a)$, $\mathrm{Modified}(c_i) = \{X\}$ und $\mathrm{Modified}(c_j) = \{Y\}$. Aus $c_i \,\hat{\kappa}\, c_j$ folgen $c_i \,\hat{\kappa}_{\mathrm{V}}\, c_j$ und $c_i \,\hat{\kappa}_{\mathrm{M}}\, c_j$. Wegen $c_i \,\hat{\kappa}_{\mathrm{V}}\, c_j$ ist $\mathrm{Used}(c_i) \cap \mathrm{Modified}(c_j) = \emptyset$ und somit muss $Y \notin \mathrm{Vars}(a)$ gelten. Ferner folgt aus $c_i \,\hat{\kappa}_{\mathrm{M}}\, c_j$, dass $\mathrm{Modified}(c_i) \cap \mathrm{Modified}(c_j) = \emptyset$, also $X \neq Y$. Da diese beiden Voraussetzung erfüllt sind, lässt sich Lemma 10.4 auf c_i und c_j anwenden und es folgt $c_i; c_j \sim c_j; c_i$. $\square$

Das folgenden Lemma zeigt einen Zusammenhang zwischen der Abhängigkeits- und der Kopplungsrelation.

Lemma 10.29. *Seien c_i und c_j atomare Anweisungen aus der Sequenz S. Es gilt die folgende Äquivalenz:*

$$c_i \not\lhd_S c_j \wedge c_j \not\lhd_S c_i \iff c_i \,\hat{\kappa}\, c_j \wedge c_j \,\hat{\kappa}\, c_i$$

Beweis. Da $\hat{\kappa}$ symmetrisch ist, reicht es zu zeigen, dass

$$c_i \not\lhd_S c_j \wedge c_j \not\lhd_S c_i \iff c_i \,\hat{\kappa}\, c_j.$$

Gemäß der Definition von $\lhd_S = \prec_S \cap \kappa$ folgt aus $c_i \,\hat{\kappa}\, c_j$, dass $c_i \not\lhd_S c_j$. Wegen $c_j \,\hat{\kappa}\, c_i$ gilt gleichzeitig $c_j \not\lhd_S c_i$.

Für die $\Longrightarrow$-Richtung der Äquivalenz wird ausgenutzt, dass $\prec_S$ total ist. Damit gilt entweder $c_i \prec_S c_j$ oder $c_j \prec_S c_i$. Sei ohne Beschränkung der Allgemeinheit $c_i \prec c_j$. Dann folgt aus $c_i \not\lhd c_j$, dass $c_i \,\hat{\kappa}\, c_j$ und mit der Symmetrie auch $c_j \,\hat{\kappa}\, c_i$. □

Aus Lemma 10.28 und Lemma 9.28 folgt direkt, dass die Vertauschung zweier benachbarter Anweisungen c_i und c_{i+1} mit $c_i \,\hat{\kappa}\, c_{i+1}$ eine neue Sequenz S' ergibt, die semantisch äquivalent zu S ist. Das folgende Lemma beschreibt den Zusammenhang zwischen der Relation $\lhd_S$ und der Relation $\lhd_{S'}$.

Lemma 10.30. *Für eine Sequenz atomarer Anweisungen $S \equiv c_1; c_2 \ldots c_n$ sei $\lhd_S$ die Abhängigkeitsrelation, die sich nach Definition 10.22 ergibt. Ferner sei i ein Index mit $1 \leq i < n$, sodass $c_i \not\lhd_S c_{i+1}$ gilt. Für die Sequenz $S' \equiv c_1; c_2 \ldots c_{i+1}; c_i \ldots c_n$ ist die Abhängigkeitsrelation $\lhd_{S'}$ identisch zu der Relation $\lhd_S$.*

Beweis. Es ist zu zeigen, dass $\lhd_S = \lhd_{S'}$ mit $S' \equiv c_1; c_2 \ldots c_{i+1}; c_i \ldots c_n$ für einen bestimmten Index i ist. Dies ist nach Definition von $\lhd$ äquivalent zu

$$\lhd_S = \prec_S \cap \kappa \;=\prec_{S'} \cap \kappa \;= \lhd_{S'} \;.$$

Da κ unabhängig von S und S' ist, können sich Unterschiede zwischen $\lhd_S$ und $\lhd_{S'}$ nur aus Unterschieden zwischen $\prec_S$ und $\prec_{S'}$ ergeben. Nach Konstruktion von S' aus S durch Vertauschung von c_i und c_{i+1} ist offensichtlich $\prec_S \neq \prec_{S'}$, da $c_i \prec_S c_{i+1}$ aber $c_i \not\prec_{S'} c_{+1}$. Umgekehrt gilt ebenfalls $c_{i+1} \prec_{S'} c_i$ aber $c_{i+1} \not\prec_S c_i$. Es wird gezeigt, dass dies die einzigen Unterschiede sind und dann, dass diese Unterschiede durch den Schnitt mit κ in $\lhd_S$ und $\lhd_{S'}$ wegfallen.

Für alle Paare $c_j, c_k \in \mathbb{S} \times \mathbb{S}$ mit $j, k \notin \{i, i+1\}$ gilt offensichtlich $c_j \prec_S c_k \iff c_j \prec_{S'} c_k$. Ferner gilt für alle Indices $k < i$ und $l \in \{i, i+1\}$, dass sowohl $c_k \prec_S c_l$ als auch $c_k \prec_{S'} c_l$. Für $\prec_S$ folgt dies aus $k < i < i+1$. Dasselbe Argument gilt auch für $\prec_{S'}$, denn der Index von c_i in S' ist $i+1$ und der von c_{i+1} ist i. Wieder ist $k < i < i+1$ und somit folgt die Behauptung. Ein analoges Argument belegt, dass für alle Indices $j > i+1$ und $l \in \{i, i+1\}$, dass sowohl $c_l \prec_S c_j$ als auch $c_l \prec_{S'} c_j$.

Damit sind (c_i, c_{i+1}) und (c_{i+1}, c_i) die einzigen Paare, für die sich $\prec_S$ und $\prec'_S$ unterscheiden. Nach Voraussetzung des Lemmas gilt allerdings $c_i \not\lhd_S c_{i+1}$. Da aber $c_i \prec_S c_{i+1}$ gilt, muss $(c_i, c_{i+1}) \notin \kappa$ gelten. Aus der Symmetrie von κ folgt auch $(c_{i+1}, c_i) \notin \kappa$. □

Bisher wurde stets nur eine einzige Vertauschung vorgenommen. Im Folgenden werden allerdings weitere Vertauschungen im Anschluss vorgenommen. Um dies kompakt zu beschreiben, werden zwei Funktionen mit dem überladenen Namen Swap definiert.

Definition 10.31. *Zum Vertauschen zweier benachbarter Elemente dient die Funktion* $\text{Swap} : \mathbb{C}^*_{atomar} \times \mathbb{N} \to \mathbb{C}^*_{atomar}$, *welche als*

$$\text{Swap}(\langle c_1, \ldots, c_n \rangle, i) = \begin{cases} \langle c_1, \ldots, c_{i+1}, c_i, \ldots, c_n \rangle & \text{falls } 1 \leq i < n \\ \langle c_1, \ldots, c_n \rangle & \text{sonst} \end{cases}$$

definiert ist. Das konsekutive Vertauschen *zweier benachbarter Elemente an den Stellen $i_1, \ldots, i_m$ wird definiert durch die Funktion* $\text{Swap} : \mathbb{C}^*_{atomar} \times \mathbb{N}^* \to \mathbb{C}^*_{atomar}$, *welche als*

$$\text{Swap}(\vec{S}, I) = \begin{cases} \text{Swap}(\text{Swap}(\vec{S}, i), I') & \text{falls } I = i \cdot I' \\ \vec{S} & \text{falls } I = \varepsilon \end{cases}$$

definiert ist.

Lemma 10.32. *Sei S eine Sequenz atomarer Anweisungen der Länge n und I eine Folge der Länge m auf den natürlichen Zahlen $I = \langle i_1, \ldots, i_m \rangle$. Sei $\vec{S}_0 = \vec{S}$ und für k mit $1 \leq k \leq m$ seien die Folgen $\vec{S}_k$ definiert durch $\vec{S}_k = \text{Swap}(\vec{S}, \langle i_1, \ldots, i_k \rangle)$. Falls für alle k mit $1 \leq k \leq m$ gilt, dass $\vec{S}_{k-1}(i_k) \not\lhd_S \vec{S}_{k-1}(i_k + 1)$, dann gilt für die zugehörigen Sequenzen $S_k = \vec{S}_k(1); \ldots; \vec{S}_k(n)$, dass S semantisch äquivalent zu S_k und insbesondere $S \sim S_m$.*

Beweis. Der Beweis erfolgt über Induktion nach der Länge m der Indexfolge I. Zusätzlich zu $S \sim S_m$ wird gezeigt, dass $\lhd_S = \lhd_{S_m}$ gilt.

Sei $m = 1$. Nach Definition von $\vec{S}_k$ gilt $\vec{S}_1 = \text{Swap}(\vec{S}, \langle i_1 \rangle)$ und nach Definition 10.31 dass $\text{Swap}(\vec{S}, \langle i_1 \rangle) = \text{Swap}(\vec{S}, i_1)$. Die Voraussetzung des zu beweisenden Lemmas liefert, dass $\vec{S}_0(i_1) \not\lhd_S \vec{S}_0(i_1 + 1)$ gilt. Da $S_0 = S$, ist Lemma 10.28 anwendbar und es folgt $S \sim S_1$. Ferner gilt Lemma 10.30 und damit $\lhd_S = \lhd_{S_1}$.

Sei nun $m = j+1$ und die Induktionsannahme für Indexfolgen der Länge j erfüllt. Es ist nun zu zeigen, dass $S \sim S_{j+1}$ und $\lhd_S = \lhd_{S_{j+1}}$. Dabei ist nach Definition $\vec{S}_{j+1} = \text{Swap}(S, \langle i_1, \ldots, i_{j+1} \rangle)$. Nach Definition von Swap ist damit $\vec{S}_{j+1} = \text{Swap}(S_j, i_{j+1})$. Nach Voraussetzung von Lemma 10.32 gilt $\vec{S}_j(i_{j+1}) \not\lhd_S \vec{S}_j(i_{j+1} + 1)$. Nach Induktionsvoraussetzung gilt $\lhd_S = \lhd_{S_j}$. Somit ist $\vec{S}_j(i_{j+1}) \not\lhd_{S_j} \vec{S}_j(i_{j+1} + 1)$ gegeben und Lemma 10.28, folgt $S_{j+1} \sim S_j$ und wegen der Induktionsvoraussetzung $S_j \sim S$ auch $S_{j+1} \sim S$. Wiederum mit 10.30 folgt, dass $\lhd_{S_{j+1}} = \lhd_{S_j} = \lhd_S$. □

Der letzte Schritt belegt die Konstruierbarkeit aller topologischen Sortierungen durch konsekutives Vertauschen benachbarter, nicht gekoppelter Anweisungen.

Lemma 10.33. *Sei S eine Sequenz der Länge n von atomaren Anweisungen und π eine Permutation, sodass $S' = \pi(S)$ eine topologische Sortierung bezüglich $\lhd_S$ darstellt. Dann kann S' durch konsekutives Vertauschen benachbarter Anweisungen in S konstruiert werden, sodass nur nicht durch $\lhd_S$ verbundene Anweisungen vertauscht werden.*

Beweis. Der Beweis erfolgt durch Induktion über die Länge der Sequenz S. Für Sequenzen der Länge 1 ist die Aussage trivialerweise wahr. Sei also die Länge von S und S' jeweils $k + 1$. Es werden zwei Fälle unterschieden.

Erstens: an den jeweils ersten Stellen befindet sich dieselbe Anweisung, also $\vec{S}(1) = \vec{S}'(1) = c_1$. Dann gibt es für die Sequenzen $\vec{S}$ und $\vec{S}'$ jeweils zwei Sequenzen der Länge k, sodass $\vec{S} = c_1 \cdot \vec{S}_t$ und $\vec{S}' = c \cdot \vec{S}'_t$ gilt. Da $|S_t| = |S'_t| = k$, folgt aus der Induktionsvoraussetzung, dass die Behauptung erfüllt ist.

Zweitens: an den jeweils ersten Stellen befinden sich unterschiedliche Anweisungen. Sei $\vec{S}(1) = c_1$ und $\vec{S}'(1) = c'_1$ mit $c_1 \neq c'_1$. Dann gibt es einen Index $i > 1$, sodass $c'_1 = S(j) = c_j$. Es wird nun gezeigt, dass vor dem Index j nur Anweisungen c_i stehen können, sodass $c_i \not\lhd_S c_j$ und $c_j \not\lhd_S c_i$ gilt. Da $\vec{S}$ nach Korollar 10.27 eine topologische Sortierung bezüglich $\lhd_S$ ist, gilt $c_j \not\lhd_S c_i$. In S' steht c_j vor c_i und $\vec{S}'$ ist nach Voraussetzung eine topologische Sortierung bezüglich $\lhd_S$. Folglich muss auch $c_i \not\lhd_S c_j$ erfüllt sein. Daher kann c_j an die erste Stelle getauscht werden, wodurch sich S'' ergibt. Damit ist $\vec{S}''(1) = \vec{S}'(1)$ und somit kann nach dem ersten Teil S'' zu S' überführt werden.

□

Theorem 10.34. *Sei S eine Sequenz atomarer Anweisungen und $\lhd_S$ die Relation aus Definition 10.22. Jede topologische Sortierung von S bezüglich $\lhd_S$ ist semantisch äquivalent zu S.*

Beweis. Der Satz gilt, da nach Lemma 10.33 jede Permutation, die eine topologische Sortierung bezüglich $\lhd_S$ ist, durch Schritte konstruiert werden kann, die nach Lemma 10.32 nur semantisch äquivalente Sequenzen erzeugen. □

11 Transformation von Sequenzen terminierender Anweisungen

Der bisherige Ansatz ist beschränkt auf das Umsortieren atomarer Anweisungen. Dies ist darin begründet, dass die Kopplungsrelation κ nur für atomare Anweisungen definiert ist. Der Beweis von Theorem 10.34 beruht nicht auf der Atomarität der Anweisungen direkt, sondern darauf, dass nicht gekoppelte Anweisungen getauscht werden können. Es scheint daher naheliegend, die Kopplung für beliebige Anweisungen zu definieren. Wenn es dabei ferner gelingt zu zeigen, dass zwei möglicherweise zusammengesetzte, aber nicht gekoppelte Anweisungen getauscht werden können, so folgt mit derselben Argumentation eine analoge Aussage über beliebige Anweisungen.

Der erste Schritt besteht darin, die in Definition 10.20 verwendeten Mengen Used(c) und Modified(c) auch für zusammengesetzte Anweisungen zu definieren. Im Folgenden wird das Subskript g verwendet, um jeweils die analogen Konstruktionen für beliebige Anweisungen zu kennzeichnen.

11.1 Die Kopplung beliebiger, terminierender Anweisungen

Die Mengen Used(c) und Modified(c) wurden über syntaktische Eigenschaften definiert. Dies erscheint für zusammengesetzte Anweisungen wie die bedingte Anweisung oder die `while`-Schleife weniger angemessen. Syntaktische Definitionen müssten jedoch bei Erweiterungen der Sprache um zum Beispiel Prozeduren oder einen Mechanismus zur Ausnahmebehandlung ebenfalls angepasst werden. Für die Definition von $\text{Used}_g(c)$ und $\text{Modified}_g(c)$ wird die Wirkung der Anweisung c auf den Zustand als Kriterium herangezogen. Dabei ist die folgende Definition hilfreich.

Definition 11.1. *Sei c eine terminierende Anweisung. Die Funktion $\delta_c : \Sigma \to \Sigma$ heißt* Zustandstransitionsfunktion *und bildet einen Zustand σ auf den Folgezustand*

ab, der sich durch die Ausführung von c in σ ergibt. Formal gilt

$$\delta_c(\sigma) = \sigma' \iff \langle \sigma, c \rangle \to \sigma'.$$

Für die Komponenten des Folgezustands $\delta_c(\sigma) = (\sigma'_V, \sigma'_{in}, \sigma'_{out})$ werden die Kurzschreibweisen

$$\delta_c(\sigma)_V = \sigma'_V,$$
$$\delta_c(\sigma)_{in} = \sigma'_{in},$$
$$\delta_c(\sigma)_{out} = \sigma'_{out}$$

verwendet. Sei $\delta_c(\sigma) = (\sigma'_V, \sigma'_{in}, \sigma'_{out})$, so wird $\delta_c(\sigma)(X)$ für $\sigma'_V(X)$ notiert.

Die Definition beschränkt sich auf terminierende Anweisungen $c \in \mathbb{C}$. Damit ergibt sich, dass δ_c eine totale Funktion ist. Diese Einschränkung wird durch die Betrachtung von Ein- und Ausgabeanweisungen notwendig. Sei c_1 eine Anweisung, die in keinem Zustand terminiert und c_2 eine Ausgabeanweisung. Offensichtlich terminieren weder $c_1; c_2$ noch $c_2; c_1$. Dennoch ergibt sich, dass $c_2; c_1$ eine Ausgabe erzeugt, $c_1; c_2$ jedoch nicht. Eine formale Semantik sollte diesen Umstand ausdrücken. Leider gelingt dies in der operationellen Semantik nicht. Die Beschränkung auf terminierende Anweisungen betrifft damit auch sämtliche weitere Überlegungen dieses Abschnitts.

Die Definition von $\text{Modified}_g(c)$ erweist sich als wesentlich einfacher und wird daher vorgezogen. In der Variante für atomare Anweisungen war höchstens eine Variable in $\text{Modified}(c)$ enthalten und zwar eine, deren Wert sich unter Umständen geändert hat. Diese Überlegung soll nun verallgemeinert werden.

Definition 11.2. *Die Menge der* veränderten Variablen *einer terminierenden Anweisung $c \in \mathbb{C}$ umfasst sämtliche Variablen, deren Wert sich bei Ausführung von c in zumindest einem Zustand σ ändert. Es wird daher definiert, dass*

$$\text{Modified}_g(c) = \{X \in \mathbb{L} : \exists \sigma \in \Sigma \textit{ mit } \sigma(X) \neq \delta_c(\sigma)(X)\}.$$

Korollar 11.3. *Sei $c \in \mathbb{C}$ eine terminierende Anweisung. Dann folgt aus $X \notin \text{Modified}_g(c)$, dass sich der Wert von X in keinem Zustand σ durch die Ausführung von c ändert. Formal gilt*

$$X \notin \text{Modified}_g(c) \implies \forall \sigma \in \Sigma : \sigma(X) = \delta_c(\sigma)(X).$$

Beweis. Sei σ ein Zustand, sodass $\sigma(X) \neq \delta_c(\sigma)(X)$. Dann wäre nach Definition 11.2 eben $X \in \text{Modified}_g(c)$ im Widerspruch zu $X \notin \text{Modified}_g(c)$. □

Bemerkung 11.4. *Die Definition 11.2 ist nicht für alle atomaren Anweisungen identisch zu der Definition 10.9.*

Beweis. Die Anweisung $c \equiv X := X$ ist ein Gegenbeispiel. Es gilt $\mathrm{Modified}(c) = \{X\}$ aber $\mathrm{Modified}_g(c) = \emptyset$. □

Die Definition der Menge der gelesenen Variablen gestaltet sich schwieriger. Zentral ist dabei das Problem, dass die Änderung einer Variable kein geeignetes Kriterium zur Definition liefert. Sei X eine möglicherweise gelesene Variable in einer Anweisung c. Offensichtlich kann nicht wie bei $\mathrm{Modified}_g(c)$ davon ausgegangen werden, dass sich der Wert von X in einem bestimmten Zustand ändert. Auch die Änderung des Wertes einer weiteren Variable Y auf den Wert von X eignet sich nicht. Der neue Wert von Y kann unabhängig von X denselben Wert annehmen, wie zum Beispiel in $X := 1; Y := 1$. Ferner kann sich der Wert von Y in Abhängigkeit von X ändern, ohne den Wert von X anzunehmen. Ein Beispiel ist $c \equiv Y := X + 1$. Wird die Belegung von Y mit dem Wert von X als Kriterium für die gelesenen Variablen herangezogen ergibt sich für dieses Beispiel fälschlicherweise, dass X keine in c gelesene Variable ist. Zusätzlich sind auch Konsequenzen denkbar, die sich durch die Verwendung von X in dem Bedingungsausdruck einer bedingten Anweisung oder Schleife ergeben. Sei $c \equiv \texttt{if } X = 0 \texttt{ then } Y := 1 \texttt{ else } Y := 0$. Der Wert von Y ändert sich in Abhängigkeit vom Wert von X, jedoch nie auf den Wert von X.

In der vorliegenden Arbeit wird daher der folgende Ansatz gewählt. Anstatt nur einen Zustand σ und den Folgezustand $\delta_c(\sigma)$ zu berücksichtigen, werden zusätzlich noch der manipulierte Zustand $\sigma[n/X]$ und der entsprechende Folgezustand $\delta_c(\sigma[n/X])$ betrachtet. Gibt es einen Wert n, sodass sich die beiden Folgezustände unterscheiden, so spielt der Wert der Variablen X in c eine Rolle. Wiederum resultieren die Schwierigkeiten nicht aus der Einführung von Ein- und Ausgabe, sondern sind bereits für die Urfassung von **IMP** gegeben.

Die folgenden Definitionen werden für die Definition von $\mathrm{Used}_g(c)$ benötigt.

Definition 11.5. *Das Prädikat* $\mathrm{C}_{c,\sigma}(X)$ *wird definiert als*

$$\mathrm{C}_{c,\sigma}(X) \iff \forall n \in \mathbb{Z} : \delta_c(\sigma[n/X])(X) = const.$$

Für den Zustand σ *heißt eine Anweisung* c *eine* konstante Zuweisung *bezüglich einer Variablen* X *genau dann, wenn* $\mathrm{C}_{c,\sigma}(X)$.

Die obige Definition ist vom Zustand abhängig. Zur Illustration werden zwei Beispiele angegeben. Einmal sei $c \equiv X := 1$. Damit ist für alle Zustände $\sigma \in \Sigma$

das Prädikat $\mathrm{C}_{c,\sigma}$ erfüllt. Das zweite Beispiel ist eine bedingte Anweisung. Sei $c \equiv \texttt{if}\ b\ \texttt{then}\ X := 1$ mit b einem beliebigen booleschen Ausdruck. Dann gilt $C_{c,\sigma}(X)$ nur, falls $\langle\sigma, b\rangle \to \texttt{true}$ gilt, also die Bedingung erfüllt ist. Es gilt jedoch folgender Zusammenhang.

Lemma 11.6. *Für alle Anweisungen c, alle Zustände $\sigma \in \Sigma$ und alle $n \in \mathbb{Z}$ gilt*

$$C_{c,\sigma}(X) \iff \mathrm{C}_{c,\sigma[n/X]}(X).$$

Beweis. Aus Definition 11.5 folgt, dass $\delta_c(\sigma) = \delta_c(\sigma[n/X])$. □

Definition 11.7. *Das Prädikat* $\mathrm{Id}_{c,\sigma}(X)$ *wird definiert als*

$$\mathrm{Id}_{c,\sigma}(X) \iff \forall n \in \mathbb{Z} : \delta_c(\sigma[n/X])(X) = n.$$

Für den Zustand σ heißt eine Anweisung identitär *bezüglich einer Variablen X genau dann, wenn* $\mathrm{Id}_{c,\sigma}(X)$.

Definition 11.8. *Zwei Zustände heißen* Ausgabe-ähnlich *genau dann, wenn*

$$\sigma \sim_{EA} \sigma' \iff \sigma_{out} = \sigma'_{out} \wedge \sigma_{in} = \sigma'_{in}$$

für $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$ und $\sigma' = (\sigma'_V, \sigma'_{in}, \sigma'_{out})$.

Die Gleichheit der Eingabekomponenten σ_{in} und σ'_{in} muss interpretiert werden als Ausführung des Programms in zwei ununterscheidbaren Umgebungen.

Definition 11.9. *Die Menge der* gelesenen Variablen *einer terminierenden Anweisung $c \in \mathbb{C}$ umfasst sämtliche Variablen, deren Wert die Ausführung von c beeinflusst. Formal gilt:*

$$\begin{aligned}\mathrm{Used}_g(c) = \{X \in \mathbb{L} : \exists \sigma \in \Sigma : &(\neg\, \mathrm{Id}_{c,\sigma}(X) \wedge \neg\, \mathrm{C}_{c,\sigma}(X)) \\ &\vee (\exists n \in \mathbb{Z} : \delta_c(\sigma) \not\sim_{EA} \delta_c(\sigma[n/X]) \\ &\vee (\exists n \in \mathbb{Z}, Y \in \mathrm{Modified}_g(c), Y \neq X : \delta_c(\sigma)(Y) \neq \delta_c(\sigma[n/X])(Y))\}\end{aligned}$$

Bemerkung 11.10. *Die Mengen* $\mathrm{Used}_g(c)$ *und* $\mathrm{Used}(c)$ *aus Definition 10.8 sind nicht für alle atomaren Anweisungen c identisch.*

Beweis. Für die Anweisung $c \equiv X := X$ gilt $\mathrm{Used}(c) = \{X\}$. Für alle Zustände $\sigma \in \Sigma$ gilt $\mathrm{Id}_{c,\sigma}$. Ebenfalls für alle Zustände $\sigma \in \Sigma$, alle Zahlen $n \in \mathbb{Z}$ und alle Variablen $Y \in Loc$ gilt und $\delta_c(\sigma) \sim_{EA} \delta_c(\sigma[n/Z'])$, da keine Lese- oder Schreiboperation

vorliegt. Ferner ist $\mathrm{Modified}_g(c) = \emptyset$. Daher muss $\mathrm{Used}_g(c) = \emptyset$ und damit $\mathrm{Used}(c) \neq \mathrm{Used}_g(c)$. □

Die im Vergleich zur Definition der veränderten Variablen verhältnismäßig komplexe Definition 11.9 ist notwendig, um alle Variablen zu erfassen, die die Ausführung einer Anweisung c beeinflussen. Dabei ist vor allem die Klausel $\neg\,\mathrm{Id}_{c,\sigma}(X) \wedge \neg\,\mathrm{C}_{c,\sigma}(X)$ relevant, wie folgendes Beispiel demonstriert.

Beispiel 11.11. *Sei* $c \equiv$ `if` $\mathrm{Z1} = 1$ `then` $\mathrm{Z2} := \mathrm{Z2}$ `else` $\mathrm{Z2} := 0$. *Für einen Zustand* σ *mit* $\sigma(\mathrm{Z1}) = \sigma(\mathrm{Z2}) = 2$ *gilt* $\sigma(\mathrm{Z2}) \neq \delta_c(\sigma)(\mathrm{Z2}) = 0$. *Es gilt daher* $\mathrm{Z2} \in \mathrm{Modified}_g(c)$. *Da* Z2 *die einzige Variable ist, der in* c *ein Wert zugewiesen wird, gilt sogar* $\mathrm{Modified}_g(c) = \{\mathrm{Z2}\}$. *Des Weiteren muss* $\mathrm{Z1} \in \mathrm{Used}_g(c)$, *da für* $n = 1$, $Y = \mathrm{Z2}$ *und den bereits definierten Zustand* σ *die Bedingung* $\delta_c(\sigma)(Y) = 0 \neq 2\delta_c(\sigma[n/\mathrm{Z1}])(Y))$.

Die folgende Überlegung zeigt, dass Z1 *die einzige Variable in* $\mathrm{Used}_g(c)$ *ist. Da* c *keine Ein- oder Ausgabeoperationen enthält, gilt stets* $\delta_c(\sigma) \sim_{EA} \delta_c(\sigma[n/X])$ *für beliebige* $n \in \mathbb{Z}$ *und* $X \in \mathbb{L}$. *Für weitere Variablen in* $\mathrm{Used}_g(c)$ *müsste daher also entweder die erste oder die letzte Klausel der Disjunktion in 11.9 gelten. Für alle Variablen außer* Z2 *gilt* $\mathrm{Id}_{c,\sigma}$ *in jedem Zustand* σ. *Für* Z2 *gilt* $\mathrm{Id}_{c,\sigma}(\mathrm{Z2})$ *nur, falls* $\sigma(\mathrm{Z1}) = 1$. *Ist allerdings* $\sigma(\mathrm{Z1}) \neq 1$, *so gilt* $\mathrm{C}_{c,\sigma}(\mathrm{Z1})$. *Die erste Klausel ist somit für keine weiteren Variablen erfüllt.*

Da Z2 *die einzige Variable in* $\mathrm{Modified}_g(c)$ *ist, müsste zur Erfüllung der letzten Klausel eine Variable existieren, deren Wert Einfluss auf den von* Z2 *hat. Außer für* Z1 *ist dies nur für* Z2 *selbst der Fall. Damit wäre jedoch* $X = Y$ *und Definition 11.9 schließt diesen Fall aus. Es folgt* $\mathrm{Used}_g(c) = \{\mathrm{Z1}\}$.

Eine alternative Definition von $\mathrm{Used}_g(c)$ *ohne die Identitätsklausel und die Bedingung* $X \neq Y$ *in der Form*

$$\begin{aligned}\mathrm{Used}'_g(c) = \{X \in \mathbb{L} : \vee(\exists n \in \mathbb{Z} : \delta_c(\sigma) \not\sim_{EA} \delta_c(\sigma[n/X]) \\ \vee (\exists n \in \mathbb{Z}, Y \in \mathrm{Modified}_g(c) : \delta_c(\sigma)(Y) \neq \delta_c(\sigma[n/X])(Y))\},\end{aligned}$$

wäre zu weit gefasst, da $\mathrm{Used}'_g(c) = \{\mathrm{Z1}, \mathrm{Z2}\}$. *Wird ferner die Bedingung* $X \neq Y$ *nicht gestrichen, also*

$$\begin{aligned}\mathrm{Used}''_g(c) = \{X \in \mathbb{L} : \vee(\exists n \in \mathbb{Z} : \delta_c(\sigma) \not\sim_{EA} \delta_c(\sigma[n/X]) \\ \vee (\exists n \in \mathbb{Z}, Y \in \mathrm{Modified}_g(c), X \neq Y : \delta_c(\sigma)(Y) \neq \delta_c(\sigma[n/X])(Y))\},\end{aligned}$$

so würde für $d \equiv Z := Z + 1$ *die Variable* Z *nicht zu den gelesenen Variablen gehören.*

Der Zweck der Definition ist am besten aus der Umkehrung erkennbar. Eine Variable X, die nicht in $\mathrm{Used}_g(c)$ enthalten ist, kann vor der Ausführung von c verändert werden, ohne dass sich der Ausgabezustand oder die Variablenbelegung ändert, abgesehen von X selbst.

Korollar 11.12. *Sei $c \in \mathbb{C}$ eine terminierende Anweisung und $X \in \mathbb{L}$ keine in c gelesene Variable. Dann impliziert $X \notin \mathrm{Used}_g(c)$, dass*

$$\forall \sigma \in \Sigma \ : \ \mathrm{Id}_{c,\sigma}(X) \vee \mathrm{C}_{c,\sigma}(X), \tag{11.1}$$

$$\forall \sigma \in \Sigma, n \in \mathbb{Z} \ : \ \delta_c(\sigma) \sim_{EA} \delta_c(\sigma[n/X]) \text{ und} \tag{11.2}$$

$$\forall \sigma \in \Sigma, n \in \mathbb{Z}, Y \in \mathrm{Modified}_g(c), X \neq Y \ : \ \delta_c(\sigma)(Y) = \delta_c(\sigma[n/X])(Y). \tag{11.3}$$

Beweis. Der Beweis erfolgt für alle drei Implikationen durch Widerspruch zur Voraussetzung $X \notin \mathrm{Used}_g(c)$. Existiert ein Zustand $\sigma \in \Sigma$, sodass weder $\mathrm{Id}_{c,\sigma}(X)$ noch $\mathrm{C}_{c,\sigma}(X)$ gelten, so ist die erste Klausel der Disjunktion aus Definition 11.9 erfüllt, und es gilt $X \in \mathrm{Used}_g(c)$ im Widerspruch zur Voraussetzung.

Existiert ein Zustand $\sigma \in \Sigma$ und eine Zahl $n \in \mathbb{Z}$, sodass $\delta_c(\sigma) \not\sim_{EA} \delta_c(\sigma[n/X])$, so ist die mittlere Disjunktionsklausel aus Definition 11.9 erfüllt, und es gilt $X \in \mathrm{Used}_g(c)$ im Widerspruch zur Voraussetzung.

Existiert ein Zustand $\sigma \in \Sigma$, eine Zahl $n \in \mathbb{Z}$ und eine Variable $Y \in \mathrm{Modified}_g(c)$ mit $X \neq Y$, sodass $\delta_c(\sigma)(Y) \neq \delta_c(\sigma[n/X])(Y)$, so ist die letzte Klausel aus Definition 11.9 erfüllt, und es gilt $X \in \mathrm{Used}_g(c)$ im Widerspruch zur Voraussetzung. □

Durch die Betrachtung zusammengesetzter Anweisungen werden eventuell mehrere Variablen verändert. Um dies zu beschreiben, ist die folgende Definition nützlich.

Definition 11.13. *Sei σ_V eine Variablenbelegung, $S \subseteq \mathbb{L}$ eine Menge von Variablen und $f : \mathbb{L} \to \mathbb{Z}$ eine Funktion. Die Variablenbelegung, die sich durch die Änderung sämtlicher Variablen in S gemäß f ergibt, wird notiert als $\sigma_V[f/S]$ und ist definiert als*

$$\sigma_V[f/S](X) = \begin{cases} f(X) & \text{if } X \in S, \\ \sigma_V(X) & \text{sonst.} \end{cases}$$

Für den Zustand $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$ wird die Notation $\sigma[f/S] = (\sigma_V[f/S], \sigma_{in}, \sigma_{out})$ vereinbart.

Das folgende Lemma verallgemeinert Korollar 11.12 auf mehrere Variablen.

Lemma 11.14. *Sei $\sigma \in \Sigma$ ein Zustand, $c \in \mathbb{C}$ eine terminierende Anweisung, $f : \mathbb{L} \to \mathbb{Z}$ eine Funktion und $S \subseteq \mathbb{L}$ eine Menge von Variablen mit $S \cap \mathrm{Used}_g(c) = \emptyset$.*

Es gilt, dass

$$\forall Y \in \mathrm{Modified}_g(c), Y \notin S : \delta_c(\sigma)(Y) = \delta_c(\sigma[f/S])(Y)$$

und

$$\delta_c(\sigma) \sim_{EA} \delta_c(\sigma[f/S]).$$

Beweis. Sei $|S| = n$ und die n unterschiedlichen Variablen gegeben durch $S = \{X_1, \ldots X_n\}$. Aus $S \cap \mathrm{Used}_g(c) = \emptyset$, folgt für X_1, dass $X_1 \notin \mathrm{Used}_g(c)$. Damit gilt nach Korollar 11.12, dass

$$\forall Y \in \mathrm{Modified}_g(c), Y \neq X_1 : \delta_c(\sigma)(Y) = \delta_c(\sigma[f/\{X_1\}])(Y)$$

und

$$\delta_c(\sigma) \sim_{EA} \delta_c(\sigma[f/\{X_1\}]),$$

da $\sigma[f/\{X_1\}] = \sigma[f(X_1)/X_1]$. Durch Anwendung des Korollars 11.12 auf $\sigma[f/\{X_1\}]$ und X_2 ergibt sich

$$\forall Y \in \mathrm{Modified}_g(c), Y \notin \{X_1, X_2\} : \delta_c(\sigma[f/\{X_1\}])(Y) = \delta_c(\sigma[f/\{X_1, X_2\}])(Y)$$

und

$$\delta_c(\sigma[f/\{X_1\}]) \sim_{EA} \delta_c(\sigma[f/\{X_1, X_2\}]).$$

Nach n derartigen Anwendungen ergibt sich

$$\forall Y \in \mathrm{Modified}_g(c), Y \notin S : \delta_c(\sigma[f/S - \{X_n\}])(Y) = \delta_c(\sigma[f/S])(Y)$$

und

$$\delta_c(\sigma[f/S - \{X_n\}]) \sim_{EA} \delta_c(\sigma[f/S]).$$

Aus der Transitivität der Gleichheit und von $\sim_{EA}$ folgt, dass

$$\forall Y \in \mathrm{Modified}_g(c), Y \notin S : \delta_c(\sigma)(Y) = \delta_c(\sigma[f/S])(Y)$$

und $\delta_c(\sigma) \sim_{EA} \delta_c(\sigma[f/S])$ gilt. □

Das Prädikat $\mathrm{EA}_g(c)$ wird nun für allgemeine Anweisungen definiert.

Definition 11.15. *Eine terminierende Anweisung $c \in \mathbb{C}$ ist eine Ein- oder Ausgabeanweisung, wenn sich die Ausgabekomponente vor und nach der Ausführung in*

mindestens einem Zustand unterscheidet.

$$\mathrm{EA}_g(c) \iff \exists \sigma = (\sigma_V, \sigma_{in}, \sigma_{out}) \in \Sigma : \sigma_{out} \neq \delta_c(\sigma)_{out}$$

Offensichtlich ist es nicht notwendig, σ_{in} in die Definition des Prädikats einzubeziehen, da alle Evaluationsregeln die Eingabefunktion unverändert lassen. Selbst wenn die Eingabekomponente als Strom modelliert wird, der bei Ausführung einer `read`-Anweisung zerlegt wird, würde die obige Definition ausreichen, wenn zusätzlich zu der Zerlegung des Eingabestroms der Ausgabestrom um eine Lesemarke ergänzt wird, wie es in Definition 9.25 der Fall ist.

Korollar 11.16. *Sei c eine nicht terminierende Anweisungen, die keine Ein- oder Ausgabeanweisung ist, also $\neg\,\mathrm{EA}_g(c)$. Die sich durch Ausführung von c ergebende Variablenbelegung ist dann unabhängig von der bisherigen Ausgabe und der Eingabefunktion der Ausführungsumgebung.*

$$\neg\,\mathrm{EA}_g(c) \implies \forall \sigma = (\sigma_V, \sigma_{in}, \sigma_{out}), \sigma'_{in}, \sigma'_{out} : \delta_c(\sigma)_V = \delta_c((\sigma_V, \sigma'_{in}, \sigma'_{out}))_V$$

Beweis. Aus $\neg\,\mathrm{EA}_g(c)$ folgt, dass $\sigma_{out} = \delta_c(\sigma)_{out}$ und $\sigma'_{out} = \delta_c((\sigma_V, \sigma'_{in}, \sigma'_{out}))_{out}$. Folglich wurde im Zuge der Ausführung von c keine `read`-Anweisung ausgeführt. Damit hängt die Variablenbelegung $\delta_c(\sigma)_V$ und $\delta_c((\sigma_V, \sigma'_{in}, \sigma'_{out}))_V$ lediglich von der ursprünglichen Belegung ab. Da dieses in σ und in $(\sigma_V, \sigma'_{in}, \sigma'_{out})$ jeweils σ_V ist, folgt die Behauptung, dass $\delta_c(\sigma)_V = \delta_c((\sigma_V, \sigma'_{in}, \sigma'_{out}))_V$. □

Im Folgenden sind die Variablen von Interesse, die in zwei Zuständen unterschiedlich belegt sind.

Definition 11.17. *Seien σ und σ' zwei Zustände. Die Menge der in σ und σ'* unterschiedlich belegten Variablen *wird mit $\mathrm{Diff}(\sigma, \sigma')$ notiert und ist definiert als*

$$\mathrm{Diff}(\sigma, \sigma') = \{X \in \mathbb{L} : \sigma(X) \neq \sigma'(X)\}.$$

Lemma 11.18. *Sei $c \in \mathbb{C}$ eine terminierende Anweisung und σ sowie σ' zwei unterschiedliche Zustände mit $\sigma \sim_{EA} \sigma'$. Falls $\mathrm{Diff}(\sigma, \sigma') \cap (\mathrm{Used}_g(c) \cup \mathrm{Modified}_g(c)) = \emptyset$ gilt, so sind nach der Ausführung von c sämtliche veränderte Variablen sowohl bei Ausführung in σ als auch in σ' gleich belegt.*

$$\sigma \sim_{EA} \sigma' \wedge \mathrm{Diff}(\sigma, \sigma') \cap (\mathrm{Used}_g(c) \cup \mathrm{Modified}_g(c)) = \emptyset$$
$$\implies \forall X \in \mathrm{Modified}_g(c) : \delta_c(\sigma)(X) = \delta_c(\sigma')(X)$$

Beweis. Sei $\sigma = (\sigma_v, \sigma_{in}, \sigma_{out})$. Wegen $\sigma \sim_{EA} \sigma'$ folgt, dass $\sigma' = (\sigma'_V, \sigma_{in}, \sigma_{out})$. Werden nun alle Variablen in σ, die sich von denen aus σ' unterschieden, mit dem Wert aus σ' belegt, so sind beide Zustände identisch. Es gilt also $\sigma' = \sigma[\sigma'_V / \operatorname{Diff}(\sigma, \sigma')]$. Durch Einsetzen in die Implikation ergibt sich, dass

$$\begin{aligned}\sigma \sim_{EA} \sigma' \wedge \operatorname{Diff}(\sigma,\sigma') \cap (\operatorname{Used}_g(c) \cup \operatorname{Modified}_g(c)) = \emptyset \\ \Longrightarrow \forall X \in \operatorname{Modified}_g(c) : \delta_c(\sigma)(X) = \delta_c(\sigma[\sigma'_V / \operatorname{Diff}(\sigma,\sigma')])(X)\end{aligned}$$

zu zeigen ist. Nach Lemma 11.14 ist

$$\forall X \in \operatorname{Modified}_g(c), X \notin S : \delta_c(\sigma)(X) = \delta_c(\sigma[f/S])(X).$$

Da $X \notin \operatorname{Diff}(\sigma, \sigma')$ wegen der Voraussetzung

$$\operatorname{Diff}(\sigma,\sigma') \cap (\operatorname{Used}_g(c) \cup \operatorname{Modified}_g(c)) = \emptyset$$

für alle $X \in \operatorname{Modified}_g(c)$ erfüllt ist, kann $S = \operatorname{Diff}(\sigma, \sigma')$ und $f = \sigma'_V$ gewählt werden. Es ergibt sich die zu zeigende Aussage

$$\forall X \in \operatorname{Modified}_g(c) : \delta_c(\sigma)(X) = \delta_c(\sigma[\sigma'_V / \operatorname{Diff}(\sigma,\sigma')])(X).$$

□

Damit kann die Kopplung zwischen beliebigen Anweisungen definiert werden. Dies erfolgt völlig analog zu der Kopplung zwischen atomaren Anweisungen. Wir fassen die einzelnen Definitionen unter Definition 11.19 zusammen und verzichten auf das Subskript g.

Definition 11.19. *Seien c und d terminierende Anweisungen aus $\mathbb{C}$. Die Relationen κ_{E}, κ_{M}, κ_{V} sind wie definiert als*

$$\begin{aligned} c\,\kappa_{\mathrm{E}}\,d &\iff \mathrm{EA}_g(c) \wedge \mathrm{EA}_g(d) \\ c\,\kappa_{\mathrm{M}}\,d &\iff \operatorname{Modified}_g(c) \cap \operatorname{Modified}_g(d) \neq \emptyset \\ c\,\kappa_{\mathrm{V}}\,d &\iff \operatorname{Used}_g(c) \cap \operatorname{Modified}_g(d) \neq \emptyset \vee \operatorname{Modified}_g(c) \cap \operatorname{Used}_g(d) \neq \emptyset \end{aligned}$$

und die Kopplungsrelation $\kappa \subseteq \mathbb{C} \times \mathbb{C}$ als

$$\kappa = \kappa_{\mathrm{E}} \cup \kappa_{\mathrm{M}} \cup \kappa_{\mathrm{V}}\,.$$

Die Kopplung κ ist eine symmetrische Relation, da $\wedge$ und $\cap$ kommutativ sind.

11.2 Permutationen von beliebig langen Sequenzen terminierender Anweisungen

Nachdem im vorangegangenen Abschnitt die Kopplung für beliebige, terminierende Anweisungen definiert wurde, wird damit in diesem Abschnitt eine analoge Aussage zu Theorem 10.34 auf beliebigen, terminierenden Anweisungen gezeigt. Zentral ist dabei Lemma 11.21, welches ein Gegenstück zu Lemma 10.28 bildet. Dazu wird das folgende Lemma benötigt. Es beschreibt die Variablenbelegung nach Ausführung zweier nicht gekoppelter Anweisungen, von denen mindestens eine keine Ein- oder Ausgabeanweisung ist.

Lemma 11.20. *Für zwei terminierende Anweisungen c und d mit $c \hat{\kappa} d$ und $\neg \operatorname{EA}(d)$, gilt*

$$\forall \sigma \in \Sigma, X \in \operatorname{Modified}(c) : \delta_d(\delta_c(\sigma))(X) = \delta_c(\delta_d(\sigma))(X)$$

Beweis. Sei $\sigma_c = \delta_c(\sigma)$, $\sigma_d = \delta_d(\sigma)$, $\sigma_{c,d} = \delta_d(\delta_c(\sigma))$ und $\sigma_{d,c} = \delta_c(\delta_d(\sigma))$. Damit ist zu zeigen, dass

$$\forall X \in \operatorname{Modified}(c) : \sigma_{c,d}(X) = \sigma_{d,c}(X).$$

Aus der Voraussetzung $c \hat{\kappa} d$ ergibt sich, dass

$$\operatorname{Used}(c) \cap \operatorname{Modified}(d) = \emptyset \tag{11.4}$$

$$\operatorname{Modified}(c) \cap \operatorname{Used}(d) = \emptyset \tag{11.5}$$

$$\operatorname{Modified}(c) \cap \operatorname{Modified}(d) = \emptyset \tag{11.6}$$

Aus $X \in \operatorname{Modified}(c)$ folgt wegen Gleichung (11.6), dass $X \notin \operatorname{Modified}(d)$. Damit ist nach Korollar 11.3 sichergestellt, dass

$$\forall X \in \operatorname{Modified}(c) : \sigma_c(X) = \sigma_{c,d}(X). \tag{11.7}$$

Aus den Gleichungen (11.4) und (11.6) folgt

$$\operatorname{Modified}(d) \cap (\operatorname{Used}(c) \cup \operatorname{Modified}(c)) = \emptyset.$$

Da $\sigma_d = \delta_d(\sigma)$ muss gemäß der Definition der veränderten Variablen Modified die Menge $\operatorname{Diff}(\sigma, \sigma_d)$ eine Teilmenge von $\operatorname{Modified}(d)$ sein. Daher gilt also

$$\operatorname{Diff}(\sigma, \sigma_d) \cap (\operatorname{Used}(c) \cup \operatorname{Modified}(c)) = \emptyset.$$

Aus $\neg\,\mathrm{EA}(d)$ folgt $\sigma \sim_{EA} \sigma_d$. Damit lässt sich Lemma 11.18 auf c, σ und σ_d anwenden, sodass folgt

$$\forall X \in \mathrm{Modified}(c) : \delta_c(\sigma)(X) = \delta_c(\sigma_d)(X).$$

Da $\delta_c(\sigma) = \sigma_c$ und $\delta_c(\sigma_d) = \sigma_{d,c}$ ist also

$$\forall X \in \mathrm{Modified}(c) : \sigma_c(X) = \sigma_{d,c}(X).$$

Mit Gleichung (11.7) folgt, dass

$$\forall X \in \mathrm{Modified}(c) : \sigma_{c,d}(X) = \sigma_{d,c}(X).$$

□

Das folgende Lemma entspricht Lemma 10.28.

Lemma 11.21. *Die Sequenzen $c; d$ und $d; c$ zweier terminierender Anweisungen sind semantisch äquivalent, wenn $c \,\hat{\kappa}\, d$ gilt.*

Beweis. Es ist zu zeigen, dass

$$c \,\hat{\kappa}\, d \implies c; d \sim d; c.$$

Die Zustände, die sich aus der Ausführung der beiden Sequenzen ergeben in einem beliebigen Startzustand σ, werden wie folgt bezeichnet. Für $c; d$ gilt gemäß Definition 9.20

$$\frac{\langle c, \sigma\rangle \to \sigma_c \quad \langle d, \sigma_c\rangle \to \sigma_{c,d}}{\langle c; d, \sigma\rangle \to \sigma_{c,d}}.$$

Analog gilt für $d; c$

$$\frac{\langle d, \sigma\rangle \to \sigma_d \quad \langle c, \sigma_d\rangle \to \sigma_{d,c}}{\langle c; d, \sigma\rangle \to \sigma_{d,c}}.$$

Aus der Voraussetzung $c \,\hat{\kappa}\, d$ ergibt sich, dass

$$\mathrm{Used}(c) \cap \mathrm{Modified}(d) = \emptyset, \tag{11.8}$$

$$\mathrm{Modified}(c) \cap \mathrm{Used}(d) = \emptyset, \tag{11.9}$$

$$\mathrm{Modified}(c) \cap \mathrm{Modified}(d) = \emptyset \text{ und} \tag{11.10}$$

$$\neg(\mathrm{EA}(c) \wedge \mathrm{EA}(d)). \tag{11.11}$$

Fall 1. Sei $\neg\,\mathrm{EA}(c)$ und $\neg\,\mathrm{EA}(d)$. Es folgt $\sigma_{c,d} \sim_{EA} \sigma_{d,c}$. In diesem Fall ist also

lediglich zu zeigen, dass $\sigma_{c,d\,V} = \sigma_{d,c\,V}$ ist, also

$$\forall X \in \mathbb{L} : \sigma_{c,d}(X) = \sigma_{d,c}(X) \tag{11.12}$$

Aus Gleichung (11.10) folgt, dass jedes $X \in \mathbb{L}$ entweder Element von Modified(c), Modified(d) oder keiner der beiden Mengen ist, aber nicht in beiden Mengen liegt. Ist $X \notin \text{Modified}(c) \cup \text{Modified}(d)$, so gilt nach Korollar 11.3, dass

$$\forall X \notin \text{Modified}(c) \cup \text{Modified}(d) : \sigma(X) = \sigma_c(X) = \sigma_d(X) = \sigma_{c,d}(X) = \sigma_{d,c}(X).$$

Ist hingegen $X \in \text{Modified}(c)$, dann folgt mit Lemma 11.20, dass

$$\forall X \in \text{Modified}(c) : \sigma_{c,d}(X) = \sigma_{d,c}(X)$$

da $\neg\,\text{EA}(d)$ und analog für $X \in \text{Modified}(d)$

$$\forall X \in \text{Modified}(d) : \sigma_{c,d}(X) = \sigma_{d,c}(X)$$

da ebenfalls $\neg\,\text{EA}(c)$. Somit gilt die Behauptung für alle $X \in \mathbb{L}$. Es ist hervorzuheben, dass die Anwendbarkeit von Lemma 11.20 nicht von der Annahme $\neg\,\text{EA}(c) \wedge \neg\,\text{EA}(d)$ abhängt. Stattdessen wird in beiden Anwendungsfällen des Lemmas jeweils nur $\neg\,\text{EA}(c)$ oder $\neg\,\text{EA}(d)$ verwendet.

Fall 2. Sei ohne Beschränkung der Allgemeinheit EA(c) und $\neg\,\text{EA}(d)$. Zunächst wird wie in Fall 1 gezeigt, dass $\sigma_{c,d\,V} = \sigma_{d,c\,V}$. Es müssen wieder drei Möglichkeiten betrachtet werden. Nach Gleichung 11.10 ist entweder X weder in Modified(c) noch in Modified(d) enthalten oder X in exakt einer der beiden Mengen enthalten.

Die ersten beiden Möglichkeiten sind leicht zu zeigen. Analog zu Fall 1 gilt

$$\forall X \notin \text{Modified}(c) \cup \text{Modified}(d) : \sigma_{c,d}(X) = \sigma_{d,c}(X) \tag{11.13}$$

nach Korollar 11.3. Da $\neg\,\text{EA}(d)$ folgt für $X \in \text{Modified}(c)$ nach Lemma 11.20, dass

$$\forall X \in \text{Modified}(c) : \sigma_{c,d}(X) = \sigma_{d,c}(X). \tag{11.14}$$

Es verbleibt zu zeigen, dass $\sigma_{c,d}(X) = \sigma_{d,c}(X)$ für alle $X \in \text{Modified}(d)$ gilt. Allerdings ist wegen EA(c) Lemma 11.20 nicht anwendbar. Da aber nach Gleichung 11.10 aus $X \in \text{Modified}(d)$ folgt, dass $X \notin \text{Modified}(c)$, muss nach Korollar 11.3

$$\forall X \in \text{Modified}(d) : \sigma_d(X) = \sigma_{d,c}(X) \tag{11.15}$$

gelten.

Für $\sigma = (\sigma_V, \sigma_{in}, \sigma_{out})$ sei nun $\sigma'_c = (\sigma_{cV}, \sigma_{in}, \sigma_{out})$. Nach Konstruktion gilt $\sigma \sim_{EA} \sigma'_c$. Aus den Gleichungen (11.8) und (11.10) folgt $\text{Modified}(c) \cap (\text{Used}(d) \cup \text{Modified}(d)) = \emptyset$. Da $\text{Diff}(\sigma, \sigma'_c) = \text{Diff}(\sigma, \sigma_c) \subseteq \text{Modified}(c)$, gilt auch $\text{Diff}(\sigma, \sigma'_c) \cap (\text{Used}(d) \cup \text{Modified}(d)) = \emptyset$. Damit ist Lemma 11.18 anwendbar. Es folgt, dass

$$\forall X \in \text{Modified}(d) : \delta_d(\sigma)(X) = \delta_d(\sigma'_c)(X).$$

Nach Korollar 11.16 gilt auch $\delta_d(\sigma_c)_V = \delta_d(\sigma'_c)_V$ und daher

$$\forall X \in \text{Modified}(d) : \sigma_d(X) = \delta_d(\sigma)(X) = \delta_d(\sigma_c)(X) = \sigma_{c,d}(X).$$

Mit Gleichung (11.15) folgt

$$\forall X \in \text{Modified}(d) : \sigma_{c,d}(X) = \sigma_{d,c}(X). \tag{11.16}$$

Aus den Gleichungen (11.13), (11.14) und (11.16) folgt, dass

$$\forall X \in \mathbb{L} : \sigma_{c,d}(X) = \sigma_{d,c}(X).$$

Es bleibt zu zeigen, dass $\sigma_{c,d} \sim_{EA} \sigma_{d,c}$. Da $\neg\,\text{EA}(d)$ gilt $\sigma_c \sim_{EA} \sigma_{c,d}$ und $\sigma \sim_{EA} \sigma_d$. Ferner gilt $\text{Diff}(\sigma, \sigma_d) \subseteq \text{Modified}(d)$ und nach Gleichung (11.8) auch $\text{Diff}(\sigma, \sigma_d)$ disjunkt zu $\text{Used}(c)$. Damit folgt nach Lemma 11.14, dass

$$\delta_c(\sigma) \sim_{EA} \delta_c(\sigma[\sigma_d / \text{Diff}(\sigma, \sigma_d)])$$

Wegen $\sigma \sim_{EA} \sigma_d$ gilt, dass $\sigma_d = \sigma[\sigma_d / \text{Diff}(\sigma, \sigma_d)]$ und somit $\delta_c(\sigma) \sim_{EA} \delta_c(\sigma_d)$ also $\sigma_c \sim_{EA} \sigma_{d,c}$. Zusammen mit $\sigma_c \sim_{EA} \sigma_{c,d}$ ergibt sich $\sigma_{c,d} \sim_{EA} \sigma_{d,c}$. □

Mit Theorem 10.34 wurde bewiesen, dass eine Permutation π einer Sequenz S atomarer Anweisungen semantisch äquivalent zu S ist, wenn $\pi(S)$ eine topologische Sortierung bezüglich $\lhd_S$ darstellt. Der Beweis beruht darauf, dass jede Permutation durch wiederholtes Vertauschen benachbarter Anweisungen erzeugt werden kann und bei jeder Vertauschung eine semantisch äquivalente Sequenz erzeugt wird. Dieses Ergebnis soll nun auf beliebige, terminierende Anweisungen ausgeweitet werden.

Im Folgenden wird mit S daher eine Sequenz beliebiger, terminierender Anweisungen bezeichnet. Es gelten die Notationen aus Abschnitt 10.3. Mit $\prec_S$ wird die Reihenfolge der einzelnen Anweisungen in S erfasst.

Definition 11.22. *Mit $\preceq_S$ wird die totale Halbordnung auf den beliebigen, terminierenden Anweisungen aus $\mathbb{S}$*

$$c_i \preceq_S c_j \iff i \leq j$$

notiert. Mit $\prec_S$ wird die totale Striktordnung

$$c_i \prec c_j \iff i < j$$

notiert. Es gilt $\preceq_S, \prec_S \subseteq \mathbb{S} \times \mathbb{S}$.

Damit ergibt sich direkt die Abhängigkeitsrelation für beliebige, terminierende Anweisungen.

Definition 11.23. *Die Relation $\lhd_S$ heißt* Abhängigkeitsrelation *der Sequenz S und wird definiert als*

$$\lhd_S = \prec_S \cap \kappa\,.$$

Es gilt $\lhd_S \subseteq \mathbb{S} \times \mathbb{S}$.

Lemma 11.24. *Für eine Sequenz $S \equiv c_1; c_2 \ldots c_n$ beliebiger, terminierender Anweisungen sei $\lhd_S$ die Abhängigkeitsrelation die sich nach Definition 11.23 ergibt. Ferner sei i ein Index mit $1 \leq i < n$, sodass $c_i \not\lhd_S c_{i+1}$ gilt. Für die Sequenz $S' \equiv c_1; c_2 \ldots c_{i+1}; c_i \ldots c_n$ ist die Abhängigkeitsrelation $\lhd_{S'}$ identisch zu der Relation $\lhd_S$.*

Beweis. Der Beweis erfolgt analog zu dem Beweis von Lemma 10.30. Die Striktordnung $\prec_{S'}$ unterscheidet sich von $\prec_S$ auch für beliebige Anweisungen nur in Bezug auf c_i und c_{i+1}. Aus der Voraussetzung $c_i \not\lhd_S c_{i+1}$ folgt wegen $c_i \prec_S c_{i+1}$, dass $c_i \,\hat{\kappa}\, c_{i+1}$. Da die Kopplung κ weiterhin symmetrisch ist, muss $c_{i+1} \,\hat{\kappa}\, c_i$ und somit auch $c_{i+1} \not\lhd_{S'} c_i$. Folglich gilt $\lhd_S = \lhd_{S'}$. □

Auch durch das konsekutive Vertauschen beliebiger Anweisungen bleibt die Abhängigkeitsrelation unverändert, wie das folgende Lemma beschreibt.

Lemma 11.25. *Sei S eine Sequenz beliebiger, terminierender Anweisungen der Länge n und I eine Folge der Länge m $I = \langle i_1, \ldots, i_m \rangle$ auf den natürlichen Zahlen. Sei $\vec{S}_0 = \vec{S}$ und für k mit $1 \leq k \leq m$ seien die Folgen $\vec{S}_k$ definiert durch $\vec{S}_k = \mathrm{Swap}(\vec{S}, \langle i_1, \ldots, i_k \rangle)$. Falls für alle k mit $1 \leq k \leq m$ gilt, dass $\vec{S}_{k-1}(i_k) \not\lhd_S \vec{S}_{k-1}(i_k + 1)$, dann gilt für die zugehörigen Sequenzen $S_k = \vec{S}_k(1); \ldots; \vec{S}_k(n)$, dass S semantisch äquivalent zu S_k ist und insbesondere $S \sim S_m$.*

Beweis. Der Beweis erfolgt analog zu dem Beweis von Lemma 10.32 über Induktion nach der Länge m. Dabei wird sowohl für die Induktionsannahme als auch den Induktionsschritt ausgenutzt, dass für jede einzelne Vertauschung sich nach Lemma 11.21 die Semantik und nach Lemma 11.24 die Abhängigkeitsrelation nicht ändert. □

Auch die Konstruierbarkeit jeder Permutation durch konsekutives Vertauschen gilt für beliebige Anweisungen.

Lemma 11.26. *Sei S eine Sequenz der Länge n von beliebigen, terminierenden Anweisungen und π eine Permutation, sodass $S' = \pi(S)$ eine topologische Sortierung bezüglich $\lhd_S$ darstellt. Dann kann S' durch konsekutives Vertauschen benachbarter Anweisungen in S konstruiert werden.*

Beweis. Der Beweis erfolgt analog zu dem Beweis von Lemma 10.33. Die Vertauschung eines Elements an die erste Stelle in einer Sequenz beliebiger, terminierender Anweisungen ist nach Lemma 11.25 zulässig. □

Es folgt der Hauptsatz dieses Kapitels.

Theorem 11.27. *Sei S eine Sequenz beliebiger, terminierender Anweisungen und $\lhd_S = \prec_S \cap \kappa$. Jede topologische Sortierung von S bezüglich $\lhd_S$ ist semantisch äquivalent zu S.*

Beweis. Der Satz gilt, da nach Lemma 11.26 jede Permutation, die eine topologische Sortierung bezüglich $\lhd_S$ ist, durch Schritte konstruiert werden kann, die nach Lemma 11.25 nur semantisch äquivalente Sequenzen erzeugen. □

11.3 Approximationen der Kopplungsrelation

Um das im vorigen Abschnitt bewiesene Theorem 11.27 anzuwenden, müssen die Mengen Used(c) und Modified(c) für jede Anweisung c bestimmt werden, die umsortiert werden soll. Diese Mengen sind allerdings nur für terminierende Anweisungen definiert. Nicht terminierende Anweisungen stellen für einen Algorithmus zur Berechnung der Mengen daher ungültige Eingaben dar. Über die Gültigkeit der Eingabe kann der Algorithmus jedoch nicht entscheiden, da dies im Widerspruch zum Halteproblem stünde. Für ungültige Eingaben liefert ein Algorithmus daher entweder ein Resultat in Form einer Menge von Variablen oder terminiert selber nicht.

Für terminierende Anweisungen sind die Mengen aber prinzipiell berechenbar, allerdings gelingt dies nicht effizient. Das folgende Beispiel demonstriert diese Tatsache. Wir nehmen die Existenz eines Verfahrens an, dessen Laufzeit polynomiell beschränkt ist. Es wird gezeigt, dass damit das Erfüllbarkeitsproblem SAT (siehe zum Beispiel [Weg99]) in polynomieller Zeit gelöst werden kann. Das Problem ist allerdings bekanntermaßen NP-vollständig und die Existenz eines solchen Verfahren würde damit $P = NP$ bedeuten.

Das Erfüllbarkeitsproblem besteht darin, für einen booleschen Ausdruck in konjunktiver Normalform über möglicherweise negierten Variablen zu bestimmen, ob es eine Variablenbelegung gibt, sodass der Ausdruck erfüllt ist. Sei M eine endliche Menge von Variablen, in denen die Klauseln kodiert sind und c' eine Anweisung, die berechnet, ob es eine erfüllende Belegung und das Resultat in der Variablen $X \notin M$ speichert. Dabei soll eine 1 anzeigen, dass es eine erfüllende Belegung gibt, während 0 das Gegenteil anzeigt. Wir betrachten die Anweisung $c \equiv X := 0; c'$. Aus $X \in \text{Modified}(c)$ folgt, dass es eine erfüllende Belegung gibt. Analog folgt aus $X \notin \text{Modified}(c)$, dass es nicht möglich ist, die Variablen so zu belegen, dass alle Klauseln der Eingabe erfüllt sind.

Unter der Annahme, dass $P \neq NP$, ist die Effizienz der Berechnung der Menge Modified beschränkt. Es folgt, dass die Kopplung nicht effizient berechnet werden kann. In diesem Abschnitt wird daher eine Methode vorgestellt, mit der die Relation κ approximiert werden kann. Zunächst wird der Begriff der Approximation formal definiert.

Definition 11.28. *Ein Tripel* $(\text{Used}', \text{Modified}', \text{EA}')$ *aus zwei Abbildungen* Used' *und* $\text{Modified}'$ *von* $\mathbb{C}$ *nach* $\mathcal{P}(\mathbb{L})$ *sowie einem Prädikat* EA' *über den Anweisungen* $\mathbb{C}$ *heißt* Kopplungsapproximation, *falls für alle terminierenden Anweisungen* $c \in \mathbb{C}$ *gilt* $\text{Used}(c) \subseteq \text{Used}'(c)$, $\text{Modified}(c) \subseteq \text{Modified}'(c)$ *und* $\text{EA}(c) \implies \text{EA}'(c)$. *Aus einer Kopplungsapproximation ergibt sich eine* approximierte Kopplungsrelation κ' *gemäß*

$$\kappa' = {\kappa_{\text{E}}}' \cup {\kappa_{\text{M}}}' \cup {\kappa_{\text{V}}}'$$

mit

$$\begin{aligned} c\, {\kappa_{\text{E}}}'\, d &\iff \text{EA}'(c) \wedge \text{EA}'(d) \\ c\, {\kappa_{\text{M}}}'\, d &\iff \text{Modified}'(c) \cap \text{Modified}'(d) \neq \emptyset \\ c\, {\kappa_{\text{V}}}'\, d &\iff \text{Used}'(c) \cap \text{Modified}'(d) \neq \emptyset \vee \text{Modified}'(c) \cap \text{Used}'(d) \neq \emptyset. \end{aligned}$$

Als Kurznotation für $(c, d) \notin \kappa$ *wird* $c\, \hat{\kappa}\, d$ *geschrieben.*

Die Kopplungsapproximation umfasst die Kopplungsrelation im folgenden Sinn.

Lemma 11.29. *Sei* (Used′, Modified′, EA′) *eine Kopplungsapproximation mit der approximierten Kopplungsrelation* κ'*. Dann gilt* $\kappa \subseteq \kappa'$*.*

Beweis. Zu zeigen ist, dass $c\,\kappa'\,d$ aus $c\,\kappa\,d$ folgt. Es gilt $c\,\kappa\,d$, falls $\mathrm{EA}(c) \wedge \mathrm{EA}(d)$ oder einer der Schnitte $\mathrm{Used}(c) \cap \mathrm{Modified}(d)$, $\mathrm{Modified}(c) \cap \mathrm{Used}(d)$ oder $\mathrm{Modified}(c) \cap \mathrm{Modified}(d)$ nicht leer ist. Im Fall $\mathrm{EA}(c) \wedge \mathrm{EA}(d)$ folgt wegen $\mathrm{EA}(x) \implies \mathrm{EA}'(x)$ für alle x, dass $\mathrm{EA}'(c) \wedge \mathrm{EA}'(d)$ und somit $c\,\kappa'\,d$. Da $\mathrm{Used}(x) \subseteq \mathrm{Used}'(x)$ und $\mathrm{Modified}(x) \subseteq \mathrm{Modified}'(x)$ für alle Anweisungen x gilt, folgt, dass die Schnitte der approximierten Mengen ebenfalls nicht leer sein können und daher $c\,\kappa'\,d$. □

Mit Lemma 11.29 ist es möglich, die Aussagen über die Kopplungsrelation auf eine Kopplungsapproximation zu übertragen, wie die folgenden beiden Lemmata demonstrieren.

Lemma 11.30. *Die Sequenzen* $c; d$ *und* $d; c$ *zweier terminierender Anweisungen sind semantisch äquivalent, wenn* $(c, d) \notin \kappa'$ *für eine beliebige approximierte Kopplungsrelation* κ' *gilt.*

Beweis. Wegen Lemma 11.29 folgt aus $(c, d) \notin \kappa'$, dass $(c, d) \notin \kappa$. Nach Lemma 11.21 folgt damit $c; d \sim d; c$ □

Lemma 11.31. *Sei* S *eine Sequenz von beliebigen, terminierenden Anweisungen, das Tripel* (Used′, Modified′, EA′) *eine Kopplungsapproximation mit der approximierten Kopplungsrelation* κ' *und* $\lhd'_S = \prec_S \cap\, \kappa'$*. Jede topologische Sortierung von* S *bezüglich* $\lhd'_S$ *ist semantisch äquivalent zu* S*.*

Beweis. Nach Theorem 11.27 genügt es zu zeigen, dass jede topologische Sortierung bezüglich $\lhd'_S = \prec_S \cap\, \kappa'$ auch eine topologische Sortierung bezüglich $\lhd_S = \prec_S \cap\, \kappa$ ist. Sei π eine Permutation, sodass $\pi(\vec{S})$ eine topologische Sortierung bezüglich $\lhd'_S$ aber keine bezüglich $\lhd_S$ darstellt. Es gibt daher mindestens ein Paar von Anweisungen c und d mit $c \lhd_S d$, sodass d in $\pi(\vec{S})$ vor c vorkommt. Da nach Lemma 11.29 $\kappa \subseteq \kappa'$, gilt $\lhd_S \subseteq \lhd'_S$ und aus $c \lhd_S d$ folgt somit $c \lhd'_S d$. Da d in $\pi(\vec{S})$ vor c vorkommt, ist $\pi(\vec{S})$ auch keine topologische Sortierung bezüglich $\lhd'_S$ im Widerspruch zur Annahme. □

Die Nützlichkeit von Lemma 11.31 ist gegenüber Theorem 11.27 allerdings eingeschränkt, wenn $\kappa \subset \kappa'$ gilt. Sei S eine Sequenz von Anweisungen und enthalte ein Paar die Anweisungen c und d mit $c \prec d$ und $c\,\kappa'\,d$ aber $c\,\hat{\kappa}\,d$. Die Sequenz S', die sich durch Vertauschen der beiden (möglicherweise nicht benachbarten) Anweisungen c und d ergibt, stellt eine topologische Sortierung bezüglich $\lhd_S$ dar, da $c\,\hat{\kappa}\,d$.

Mit Theorem 11.27 folgt, dass $S' \sim S$. Allerdings gilt $c \kappa d$ und wegen $c \prec_S d$ auch $c \lhd'_S d$. Da aber d in S' vor c steht, folgt, dass S' keine topologische Sortierung bezüglich $\lhd'_S$ darstellt. Lemma 11.31 ist damit nicht anwendbar und die Tatsache $S' \sim S$ kann nicht gezeigt werden.

Im Folgenden wird eine Approximation κ' beschrieben, für die $\kappa \subset \kappa'$ gilt. Nach den vorherigen Überlegung folgt, dass mit dieser Approximation nicht sämtliche semantisch äquivalenten Permutationen bestimmt werden können. Der Vorteil besteht allerdings in der effizienten Berechenbarkeit. Ähnlich wie die Mengen Used(c) und Modified(c) in Kapitel 10 auf syntaktischer Ebene für atomare Anweisungen c definiert wurden, kann eine Approximation anhand syntaktischer Eigenschaften definiert werden. Dazu werden zunächst die Komponenten dieser Approximation beschrieben.

Definition 11.32. *Das Prädikat* EA′ *wird über den Anweisungen aus* $\mathbb{C}$ *wie folgt definiert.*

$$\mathrm{EA}'(c) \iff \bigvee_{d \in \mathrm{Sub_d}(c)} \mathrm{EA}'(d) \vee c \equiv \mathtt{write}(a) \vee c \equiv \mathtt{read}(X)$$

Korollar 11.33. *Für alle Anweisungen* $c \in \mathbb{C}$ *gilt,* $\mathrm{EA}'(c)$*, falls eine atomare Anweisung* $d \in \mathrm{Atom}(c)$ *mit* $\mathrm{EA}'(d)$ *existiert.*

Beweis. Der Beweis erfolgt durch Induktion über die Struktur der Anweisungen. Für die atomaren Anweisungen folgt die Eigenschaft direkt aus der Definition von EA′. Sei c eine zusammengesetzte Anweisung und $d \in \mathrm{Atom}(c)$ mit $\mathrm{EA}'(d)$. Da $d \in \mathrm{Atom}(c)$, gibt es eine Anweisung $c' \in \mathrm{Sub_d}(c)$ mit $d \in \mathrm{Atom}(c')$. Nach Induktionsvoraussetzung gilt, dass $\mathrm{EA}'(c')$ für $c' \in \mathrm{Sub_d}(c)$, da $d \in \mathrm{Atom}(c')$. Aus $\mathrm{EA}'(c')$ für ein $c' \in \mathrm{Sub_d}(c)$ folgt aus der Definition $\mathrm{EA}'(c)$ und somit die Behauptung. □

Definition 11.34. *Die Abbildung* Modified′ *wird über den Anweisungen aus* $\mathbb{C}$ *rekursiv definiert als*

$$\mathrm{Modified}'(c) = \begin{cases} \{X\}, & \textit{falls } c \equiv X := a, \\ \{X\}, & \textit{falls } c \equiv \mathtt{read}(X), \\ \emptyset, & \textit{falls } c \equiv \mathtt{write}(a) \end{cases}$$

für atomare Anweisungen und als

$$\mathrm{Modified}'(c) = \bigcup_{d \in \mathrm{Sub_d}(c)} \mathrm{Modified}'(d)$$

für zusammengesetzte Anweisungen.

Korollar 11.35. *Sei $X \in \mathbb{L}$ eine beliebige Variable. Für alle Anweisungen $c \in \mathbb{C}$ gilt $X \in \mathrm{Modified}'(c)$, falls eine atomare Anweisung $d \in \mathrm{Atom}(c)$ existiert mit $X \in \mathrm{Modified}'(d)$.*

Beweis. Der Beweis erfolgt durch Induktion über die Struktur der Anweisungen. Für die atomaren Anweisungen gilt die Eigenschaft trivialerweise, da $c = d$ sein muss. Sei $c \in \mathbb{C}$ eine zusammengesetzte Anweisung, d eine atomare Anweisung mit $d \in \mathrm{Atom}(c)$ und $X \in \mathrm{Modified}'(d)$. Da $d \in \mathrm{Atom}(c)$, gibt es eine Anweisung $c' \in \mathrm{Sub}_\mathrm{d}(c)$ mit $d \in \mathrm{Atom}(c')$. Nach Induktionsvoraussetzung gilt für alle $c' \in \mathrm{Sub}_\mathrm{d}(c)$ mit $d \in \mathrm{Atom}(c')$, dass $X \in \mathrm{Modified}'(c')$. Nach Definition 11.34 ist damit auch $X \in \mathrm{Modified}'(c)$. □

Definition 11.36. *Die Abbildung* Used′ *wird über den Anweisungen aus $\mathbb{C}$ rekursiv definiert als*

$$\mathrm{Used}'(c) = \begin{cases} \mathrm{Vars}(a), & \textit{falls } c \equiv X := a, \\ \mathrm{Vars}(a), & \textit{falls } c \equiv \textbf{\textit{write}}(a), \\ \emptyset, & \textit{falls } c \equiv \textbf{\textit{read}}(X) \end{cases}$$

für atomare Anweisungen und als

$$\mathrm{Used}'(c) = \begin{cases} \bigcup_{d \in \mathrm{Sub}_\mathrm{d}(d)} \mathrm{Used}'(d) \cup \mathrm{Vars}(b) & \textit{falls } c \equiv \texttt{while } b \texttt{ do } c_1, \\ \mathrm{Vars}(b) \cup \mathrm{Used}'(c_1) & \textit{falls } c \equiv \texttt{if } b \texttt{ then } c_1, \\ \mathrm{Vars}(b) \cup \mathrm{Used}'(c_1) \cup \mathrm{Used}'(c_2) & \textit{falls } c \equiv \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2, \\ \mathrm{Used}'(c_1) \cup \mathrm{Used}'(c_2) & \textit{falls } c \equiv c_1; c_2 \end{cases}$$

für zusammengesetzte Anweisungen.

Die folgenden Lemmata basieren auf Used und nicht auf Used′.

Lemma 11.37. *Sei c eine beliebige Anweisung und X eine Variable mit $X \notin \mathrm{Used}(c)$.*

$$\forall \sigma \in \Sigma, n \in \mathbb{Z} : \mathrm{C}_{c,\sigma}(X) \implies \delta_c(\sigma) = \delta_c(\sigma[n/X]).$$

Beweis. Aus $X \notin \mathrm{Used}(c)$ folgt

$$\forall \sigma \in \Sigma, n \in \mathbb{Z} : \delta_c(\sigma) \sim_{EA} \delta_c(\sigma[n/X]).$$

Es verbleibt die Gleichheit der Variablenbelegungen zu zeigen.

Für alle Variablen $Y \notin \text{Modified}(c)$ mit $X \neq Y$ gilt gemäß der Definition von $\text{Modified}(c)$, dass $\sigma(Y) = \delta_c(\sigma)(Y)$ für alle Zustände σ und damit auch für $\sigma[n/X]$. Da $Y \neq X$ gilt $\sigma(Y) = \sigma[n/X](Y)$ und somit $\delta_c(\sigma)(Y) = \delta_c(\sigma[n/X])(Y)$.

Aus $X \notin \text{Used}(c)$ folgt, dass

$$\forall \sigma \in \Sigma, n \in \mathbb{Z}, Y \in \text{Modified}(c), Y \neq X : \delta_c(\sigma)(Y) = \delta_d(\sigma[n/X])(Y).$$

Zuletzt gilt für X wegen $\text{C}_{c,\sigma}(X)$, dass $\delta_c(\sigma)(X) = \delta_c(\sigma[n/X])(X)$ und somit die Behauptung. Es sei darauf hingewiesen, dass $\text{C}_{c,\sigma}(X)$ erst im letzten Schritt benötigt wurde. □

Lemma 11.38. *Sei c eine beliebige Anweisung und X eine Variable mit $X \notin \text{Used}(c)$.*

$$\forall \sigma \in \Sigma, n \in \mathbb{Z} : \text{Id}_{c,\sigma}(X) \implies \delta_c(\sigma)[n/X] = \delta_c(\sigma[n/X]).$$

Beweis. Der Beweis von Lemma 11.37 zeigt unabhängig von $\text{C}_{c,\sigma}(X)$, dass $\delta_c(\sigma) \sim_{EA} \delta_c(\sigma[n/X])$ und dass für alle Variablen Y mit $Y \neq X$ gilt $\delta_c(\sigma)(Y) = \delta_d(\sigma[n/X])(Y)$, falls $X \notin \text{Used}(c)$. Da $\delta_c(\sigma)$ und $\delta_c(\sigma)[n/X]$ sich nur im Wert von X unterscheiden, folgt

$$\forall \sigma \in \Sigma, n \in \mathbb{Z} : \delta_c(\sigma)[n/X] \sim_{EA} \delta_c(\sigma[n/X]).$$

und

$$\forall \sigma \in \Sigma, n \in \mathbb{Z}, Y \in \text{Modified}(c), Y \neq X : \delta_c(\sigma)[n/X](Y) = \delta_d(\sigma[n/X])(Y).$$

Es bleibt zu zeigen, dass

$$\forall \sigma \in \Sigma, n \in \mathbb{Z} : \delta_c(\sigma)[n/X](X) = \delta_d(\sigma[n/X])(X).$$

Nach Definition 9.15 gilt $\delta_c(\sigma)[n/X](X) = n$. Aus $\text{Id}_{c,\sigma}(X)$ folgt

$$\forall n \in \mathbb{Z} : \delta_d(\sigma[n/X])(X) = n = \delta_c(\sigma)[n/X](X).$$

□

Definition 11.39. *Das Tripel* $(\text{Used}', \text{Modified}', \text{EA}')$ *heißt* syntaktische Kopplungsapproximation.

Lemma 11.40. *Die syntaktische Kopplungsapproximation ist eine Kopplungsapproximation gemäß Definition 11.28.*

Beweis. Es ist zu zeigen, dass $\mathrm{EA}(c) \implies \mathrm{EA}'(c)$, $\mathrm{Used}(c) \implies \mathrm{Used}'(c)$ und $\mathrm{Modified}(c) \implies \mathrm{Modified}'(c)$.

Teil 1. Es wird die Implikation $\mathrm{EA}(c) \implies \mathrm{EA}'(c)$ bewiesen. Für alle atomaren Anweisungen c gilt $\mathrm{EA}(c) = \mathrm{EA}'(c)$. Nach Definition impliziert $\mathrm{EA}(c)$, dass es einen Zustand σ gibt, sodass der Ausgabestrom $\delta_c(\sigma)_{out}$ von σ_{out} abweicht. Da zusammengesetzte Anweisungen in **IMP** nur dann den Ausgabestrom verändern, wenn im Zuge der Ausführung ein atomares `read` oder `write` ausgeführt wird, gilt für zusammengesetzte Anweisungen c mit $\mathrm{EA}(c)$, dass eine atomare Anweisung $d \in \mathrm{Atom}(c)$ mit $\mathrm{EA}'(d)$ existiert. Aus Korollar 11.33 folgt $\mathrm{EA}'(c)$.

Teil 2. Es wird $\mathrm{Modified}(c) \subseteq \mathrm{Modified}'(c)$ bewiesen. Für $c \equiv \mathtt{write}(a)$ gilt $\mathrm{Modified}(c) = \emptyset \subseteq \mathrm{Modified}'(c)$. Für $c \equiv \mathtt{read}(X)$ gilt $\mathrm{Modified}(c) = \mathrm{Modified}'(c) = \{X\}$. Für $c \equiv X := a$ gilt $\mathrm{Modified}(c) \subseteq \{X\} = \mathrm{Modified}'(c)$. Es folgt, dass $\mathrm{Modified}(c) \subseteq \mathrm{Modified}'(c)$ für atomare Anweisungen c.

Sei c eine zusammengesetzte Anweisung und $X \in \mathrm{Modified}(c)$. Nach Definition von Modified gibt es einen Zustand σ mit $\delta_c(\sigma)(X) \neq \sigma(X)$. Da sich der Wert von X im Zuge der Ausführung von c geändert hat, muss eine Zuweisung oder eine Eingabeanweisung $d \in \mathrm{Atom}(c)$ mit $X \in \mathrm{Modified}(d)$ ausgeführt worden sein. Für die Anweisung d gilt $X \in \mathrm{Modified}'(d)$ und da $d \in \mathrm{Atom}(c)$ folgt nach Korollar 11.35, dass $X \in \mathrm{Modified}'(c)$. Damit ergibt sich $\mathrm{Modified}(c) \subseteq \mathrm{Modified}'(c)$.

Teil 3. Es wird $\mathrm{Used}(c) \subseteq \mathrm{Used}'(c)$ bewiesen. Für $c \equiv \mathtt{read}(X)$ gilt $\mathrm{Used}(c) = \emptyset \subseteq \mathrm{Used}'(c)$. Für $c \equiv \mathtt{write}(a)$ gilt $\mathrm{Used}(c) \subseteq \mathrm{Vars}(a) = \mathrm{Used}'(c)$, da Variablen $X \notin \mathrm{Vars}(a)$ nicht an der Evaluierung von a beteiligt sind. Das Beispiel eines Ausdrucks $a \equiv 0 \times X$, indem X mit 0 multipliziert wird, zeigt zusätzlich, dass sogar $\mathrm{Used}(c) \subset \mathrm{Vars}(a)$. Analog folgt für $c \equiv X := a$, dass $\mathrm{Used}(c) \subseteq \mathrm{Vars}(a)$. Damit ist für atomare Anweisungen c gezeigt, dass gilt $\mathrm{Used}(c) \subseteq \mathrm{Used}'(c)$.

Für zusammengesetzte Anweisungen wird die Behauptung mittels struktureller Induktion gezeigt. Die Basis liefert die Betrachtung für die atomaren Anweisungen aus dem vorherigen Absatz. Sei c eine zusammengesetzte Anweisung. Es genügt zu zeigen, dass $X \notin \mathrm{Used}'(c) \implies X \notin \mathrm{Used}(c)$. Nach Induktionsvoraussetzung gilt für alle Anweisungen $d \in \mathrm{Sub_d}(c)$, dass $X \notin \mathrm{Used}'(d) \implies X \notin \mathrm{Used}(d)$. Nach Definition von Used' gilt $\mathrm{Used}'(d) \subseteq \mathrm{Used}'(c)$. Damit gilt für alle Variablen X, die nicht in der Menge $\mathrm{Used}'(c)$ enthalten sind, $X \notin \mathrm{Used}'(d)$ und nach Induktionsvoraussetzung auch $X \notin \mathrm{Used}(d)$.

Es werden nun die verschiedenen zusammengesetzten Anweisungen betrachtet. Dabei wird ausgenutzt, dass $X \notin \mathrm{Used}(c)$ äquivalent dazu ist, dass alle drei folgenden

Aussagen erfüllt sind.

$$\forall \sigma \in \Sigma \quad : \quad \mathrm{Id}_{c,\sigma}(X) \vee \mathrm{C}_{c,\sigma}(X) \tag{11.17}$$

$$\forall \sigma \in \Sigma, n \in \mathbb{Z} \quad : \quad \delta_c(\sigma) \sim_{EA} \delta_c(\sigma[n/X]) \tag{11.18}$$

$$\forall \sigma \in \Sigma, n \in \mathbb{Z}, Y \in \mathrm{Modified}(c), Y \neq X \quad : \quad \delta_c(\sigma)(Y) = \delta_c(\sigma[n/X])(Y) \tag{11.19}$$

Sei $c \equiv \texttt{if}\ b\ \texttt{then}\ d_1\ \texttt{else}\ d_2$. Aus $X \notin \mathrm{Used}'(c)$ folgt $X \notin \mathrm{Vars}(b)$. Offenbar spielt der Wert von X bei der Auswertung von b in keinem Zustand eine Rolle. Formal gilt für alle Zustände $\sigma \in \Sigma$ und für alle $n \in \mathbb{Z}$, dass wenn $\langle b, \sigma \rangle \to m$ auch $\langle b, \sigma[n/X] \rangle \to m$. Sei $d = d_1$ falls $m = \texttt{true}$ und $d = d_2$ sonst. Es gilt $\delta_c(\sigma) = \delta_d(\sigma)$ und $\delta_c(\sigma[n/X]) = \delta_d(\sigma[n/X])$. Da die Aussagen 11.17, 11.18 und 11.19 für $d \in \{d_1, d_2\}$ nach Induktionsvoraussetzung erfüllt sind, folgen sie auch für c und somit gilt $X \notin \mathrm{Used}(c)$.

Der Fall $c \equiv \texttt{if}\ b\ \texttt{then}\ d_1$ folgt für $m = \texttt{true}$ analog und für $m = \texttt{false}$, da δ_c in diesen Zuständen die Identität ist und die Bedingungen trivialerweise erfüllt sind.

Sei $c \equiv d_1; d_2$. Es gilt $\delta_c(\sigma) = \delta_{d_2}(\delta_{d_1}(\sigma))$. Der Zwischenzustand $\delta_{d_1}(\sigma)$ wird mit σ_1 bezeichnet. Es wird gezeigt, dass in jedem Zustand σ entweder $\delta_c(\sigma) = \delta_c(\sigma[n/X])$ oder $\delta_c(\sigma)[n/X] = \delta_c(\sigma[n/X])$ für alle Zahlen $n \in \mathbb{Z}$ erfüllt ist. Nach Voraussetzung gilt in allen Zuständen σ, dass $\mathrm{Id}_{d_1,\sigma}(X) \vee \mathrm{C}_{d_1,\sigma}(X)$ und $\mathrm{Id}_{d_2,\sigma}(X) \vee \mathrm{C}_{d_2,\sigma}(X)$. Falls $\mathrm{C}_{d_1,\sigma}(X)$ gilt, so folgt nach Lemma 11.37, dass $\delta_{d_1}(\sigma[n/X]) = \delta_{d_1}(\sigma)$ und damit $\delta_c(\sigma) = \delta_c(\sigma[n/X])$. Falls $\mathrm{Id}_{d_1,\sigma}(X)$, so folgt nach Lemma 11.38, dass $\delta_{d_1}(\sigma)[n/X] = \delta_{d_1}(\sigma[n/X])$. Für $\mathrm{C}_{d_2,\sigma_1}(X)$ ergibt sich mit Lemma 11.37

$$\delta_c(\sigma[n/X]) = \delta_{d_2}(\delta_{d_1}(\sigma[n/X])) = \delta_{d_2}(\delta_{d_1}(\sigma)[n/X]) = \delta_{d_2}(\delta_{d_1}(\sigma)) = \delta_c(\sigma).$$

Für $\mathrm{Id}_{d_2,\sigma_1}(X)$ ergibt sich mit Lemma 11.38

$$\delta_{d_2}(\delta_{d_1}(\sigma[n/X])) = \delta_{d_2}(\delta_{d_1}(\sigma)[n/X]) = \delta_{d_2}(\delta_{d_1}(\sigma))[n/X] = \delta_c(\sigma)[n/X].$$

In allen Zuständen σ gilt also entweder $\delta_c(\sigma) = \delta_c(\sigma[n/X])$ oder $\delta_c(\sigma)[n/X] = \delta_c(\sigma[n/X])$ für jeweils alle $n \in \mathbb{Z}$. In beiden Fällen sind die Aussagen 11.18 und 11.19 erfüllt. Ferner gilt Aussage 11.17, denn entweder ist $\mathrm{C}_{c,\sigma}(X)$ oder $\mathrm{Id}_{c,\sigma}(X)$. Ersteres gilt falls $\delta_c(\sigma) = \delta_c(\sigma[n/X])$ für alle $n \in \mathbb{Z}$. Letzteres folgt aus $\delta_c(\sigma)[n/X] = \delta_c(\sigma[n/X])$ für alle $n \in \mathbb{Z}$. Damit folgt $X \notin \mathrm{Used}(c)$.

Sei $c \equiv \texttt{while}\ b\ \texttt{do}\ d$. Wieder gilt $\langle b, \sigma \rangle \to m \implies \langle b, \sigma[n/X] \rangle \to m$ für alle $n \in \mathbb{Z}$, da $X \notin \mathrm{Vars}(b)$ und alle Zustände $\sigma \in \Sigma$. Für $m = \texttt{false}$ wird δ_c wieder zur Identität. Ansonsten wird $d; c$ evaluiert. Da nur terminierende Anweisungen

betrachtet werden, ergibt sich eine endliche Sequenz der Form $d; \ldots ; d$. Da für d nach Voraussetzung $X \notin \mathrm{Used}(d)$ gilt, folgt $X \notin \mathrm{Used}(c)$ analog zu den Überlegungen für die Sequenz.

□

12 Fazit zu Teil II

In diesem Teil der Arbeit wurde gezeigt, wie sich Ein- und Ausgabeoperationen einer imperativen Programmiersprache mit den Mitteln der operationellen Semantik beschreiben lassen. Um dies zu realisieren, wurde die Sprache **IMP** um zwei atomare Anweisungen erweitert. Die Modellierung der Ausgabe als Strom und der Ausführungsumgebung als Eingabefunktion machte eine Erweiterung des Zustandsbegriffs notwendig. Mithilfe der operationellen Semantik wurden für grundlegende Fälle Bedingungen beschrieben, unter denen sich zwei aufeinander folgende atomare Anweisungen in beliebiger Reihenfolge, aber mit derselben Semantik ausführen lassen.

Mit dem Konzept der Kopplung gelang es, die mögliche Abhängigkeit zweier atomarer Anweisungen zu beschreiben. Die Definition erfolgte für atomare Anweisungen mit rein syntaktischen Mitteln. Die Kopplung atomarer Anweisungen ist unabhängig von konkreten Programmen definiert. Ob beide Anweisungen in einem äußeren Programm in einer bestimmten Reihenfolge auftreten oder nicht, spielt keine Rolle. Nur so kann die Kopplung eine symmetrische Relation sein, im Gegensatz zur Abhängigkeit zwischen zwei Anweisungen. Erst durch das Hinzunehmen einer Ordnung lässt sich entscheiden, welche Anweisung abhängig von der anderen ist. Die Trennung zwischen den Konzepten der Abhängigkeit und der Kopplung erleichterte die folgenden Beweise. Es konnte demonstriert werden, dass durch die Vertauschungen benachbarter, aber nicht gekoppelter atomarer Anweisungen, die Semantik nicht verändert wird. Vor allem gelang es zu zeigen, dass alle Permutationen einer Sequenz atomarer Anweisungen semantisch äquivalent sind, die eine topologische Sortierung bezüglich der Abhängigkeitsrelation bilden.

Auch für zusammengesetzte Anweisungen konnte eine geeignete Definition der Kopplung entwickelt werden. Anstelle einer syntaktischen Definition wurde eine Definition über die Veränderungen eines Zustands gewählt. Dies setzt lediglich voraus, dass ein Folgezustand erreicht wird. Die Kopplung beliebiger Anweisungen ist daher nur für terminierende Anweisungen definiert. Da dies die einzige Einschränkung ist, kann die Definition auch direkt auf neue Kontrollstrukturen wie zum Beispiel Konstrukte zur Ausnahmebehandlung angewendet werden.

Mit der Kopplung beliebiger terminierender Anweisungen konnte ebenfalls gezeigt werden, dass Permutationen, die die Abhängigkeitsrelation nicht verletzen, semantisch äquivalent sind. Die Allgemeinheit der Definition verhindert jedoch eine direkte Berechnung der Kopplung. Aus diesem Grund wurde das Konzept der approximierten Kopplung eingeführt und gezeigt, dass die darauf basierende approximierte Abhängigkeit ebenfalls ein geeignetes Kriterium zur Feststellung der semantischen Äquivalenz liefert. Für alle in **IMP** definierten Anweisungen wurde ferner gezeigt, dass eine Approximation auf syntaktischer Basis existiert.

12.1 Verwandte Arbeiten

Das hier gelöste Problem hat nach bestem Wissen des Autors nur wenige Anknüpfungspunkte zu Forschungen in der klassischen Programmanalyse (siehe z.B. [NNH99]). Der wesentliche Unterschied liegt darin, dass das hier gezeigte Kriterium entwickelt wurde, um effizient überprüfen zu können, ob eine Permutation ein semantisch äquivalentes Programm liefert. Es wird daher eine ganze Klasse von Transformationen betrachtet. Klassische Optimierungen zielen jedoch auf sehr spezifische Transformationen, wie das Verschieben der Berechnung eines invarianten Ausdrucks aus einer Schleife heraus. Diese spezifischen Transformationen stellen jedoch in den meisten Fällen keine Permutation einer Anweisungssequenz dar. Die einzige dem Autor bekannte Optimierung, bei der Permutationen von Anweisungen auftreten, ist die Registerauswahl. Das Problem ist im Allgemeinen NP-vollständig und wird laut [ASU88] in der Praxis durch heterogene Registerarten noch zusätzlich erschwert. Durch die geschickte Wahl der Auswertungsreihenfolge können Register maximal effizient ausgenutzt werden und vergleichsweise langsame Speicherzugriffe minimiert werden. Dieser Optimierungsschritt findet allerdings erst bei der Erzeugung des Maschinencodes statt und arbeitet dementsprechend notwendigerweise auf Assembler-Niveau. In der vorliegenden Arbeit werden stattdessen Transformationen auf Quelltext-Niveau betrachtet.

Grundsätzlich handelt es sich bei der eingeführten Kopplung um Datenflussanalyse. Insbesondere Def-Use-Ketten und Use-Def-Ketten (siehe ebenfalls [ASU88]) weisen Ähnlichkeiten auf. Tatsächlich kann der Schnitt $\kappa_{\mathrm{V}} \cap \prec_S$ für eine Sequenz atomarer Anweisungen S bestimmt werden, wenn sowohl die Def-Use-Ketten als auch die Use-Def-Ketten der Sequenz bekannt sind, jedoch nicht κ_{V} selbst und auch nicht κ_{M} und κ_{E}.

Für Teile der vorgestellten Arbeit finden sich allerdings ähnliche Überlegungen

in der Literatur. So beschäftigt sich Bernstein in [Ber66] mit der Frage, ob zwei Teilprogramme parallel ausgeführt werden können. Bernstein gibt Bedingungen an, unter denen eine Parallelisierung zulässig ist. Die bereits 1966 veröffentlichte Arbeit kommt dabei ohne formalen Semantikbegriff aus. Stattdessen wird über den Zugriff auf Speicherzellen argumentiert, welcher entweder lesend oder schreibend sein kann. Mit diesen Mitteln werden Mengen ähnlich zu Used und Modified definiert. Die definierten Bedingungen verlangen, dass Schnitte der Mengen leer sind. Die Ähnlichkeiten zu der vorliegenden Arbeit sind daher offensichtlich. Bernstein beschränkt sich auf ein Modell mit zwei Prozessoren, die auf einen gemeinsamen Speicher zugreifen. Folgerichtig beschränkt er sich auch darauf, zwei aufeinander folgende Teilprogramme zu parallelisieren.

Kuck et al. beschäftigen sich in [KKP+81] mit optimierenden Übersetzern. Insbesondere definieren sie einen sogenannten Abhängigkeitsgraphen, bei dem Anweisungen die Knoten bilden. Die Kanten werden durch Relationen beschrieben, die teilweise Entsprechungen in der vorliegenden Arbeit besitzen. So entspricht zum Beispiel δ^O in [KKP+81] dem Schnitt $\kappa_{\mathrm{M}} \cap \prec_S$. Auf Basis des Abhängigkeitsgraphen werden einige Programmtransformationen beschrieben, deren formale Korrektheit jedoch nicht bewiesen wird.

Keine dieser Arbeiten betrachtet die Transformation von Programmen in der Gegenwart von Ein- und Ausgabe. Die Berücksichtigung dieses Aspekts in der formalen Semantik scheint äußerst unüblich. Lediglich in [All86] finden sich die bereits in Kapitel 9 erwähnten Anmerkungen von Allison zu diesem Thema. In Arbeiten zu Optimierungen erfahren Ein- und Ausgabe aufgrund ihrer im Vergleich zu internen Berechnungen oder Hauptspeicherzugriffen extrem niedrigen Geschwindigkeit keine Beachtung.

A Mathematische Grundbegriffe

Definition A.1. *Die* Kardinalität *einer Menge A bezeichnet die Anzahl der Elemente der Menge A und wird notiert als $|A|$.*

Definition A.2. *Die Menge aller Teilmengen einer Menge A heißt* Potenzmenge *der Menge A und wird notiert als $\mathcal{P}(A)$. Es gilt $A' \subseteq A \iff A' \in \mathcal{P}(A)$.*

Definition A.3. *Eine Teilmenge* R *des kartesischen Produkts $A \times B$ zweier Mengen A und B heißt* (binäre) Relation. *Der Ausdruck $a \,\mathrm{R}\, b$ ist eine Kurzschreibweise für $(a, b) \in \mathrm{R}$. Eine Relation $R \subseteq A \times A$ heißt* Relation auf A. *Eine Relation $R' \subseteq R$ heißt* Teilrelation von R (auf A).

Definition A.4. *Eine* endliche Folge der Länge n *auf einer Menge A ist eine Abbildung von der Menge $\{1, \ldots, n\}$ auf die Menge A. Notiert wird eine endliche Folge der Länge n als $a_1, \ldots, a_n$ oder vor allem für kleine Längen als $\langle a_1, \ldots, a_n \rangle$. Das Element a_i der Folge $F = \langle a_1, \ldots, a_n \rangle$ wird als das i*-te Element *bezeichnet und mit $F(i)$ notiert. Die* Länge *F einer endlichen Folge wird notiert als $|F|$.*

Die Folge der Länge 0 wird als leere Folge *bezeichnet und mit ϵ notiert. Die Menge aller endlichen Folgen der Länge n wird mit A^n bezeichnet. Die Menge aller endlichen Folgen wird mit A^* oder als* Kleenescher Abschluss von A *bezeichnet.*

Eine (unendliche) Folge *auf einer Menge A ist eine Abbildung von der Menge der natürlichen Zahlen $\mathbb{N}$ in die Menge A. Notiert wird eine unendliche Folge als $a_1, a_2, \ldots$ oder analog zu endlichen Folgen als $\langle a_1, a_2, \ldots \rangle$.*

Definition A.5. *Für eine Menge A und eine Relation R auf A heißt eine unendliche Folge F auf A* R-aufsteigend *genau dann, wenn $\forall i \geq 1 : F(i) \,\mathrm{R}\, F(i+1)$. Gilt hingegen $\forall i \geq 1 : F(i+1) \,\mathrm{R}\, F(i)$, so heißt F* R-absteigend

Definition A.6. *Die* Konkatenation *einer endlichen Folge F auf A und einem Element $a \in A$ ist eine Funktion $\cdot : A^* \times A \to A^*$ definiert als*

$$\langle a_1, \ldots, a_n \rangle \cdot a = \langle a_1, \ldots, a_n, a \rangle.$$

Die Notation wird überladen, um zwei Folgen miteinander zu verbinden. Die Konkatenation *einer endlichen Folge F auf A und mit einer weiteren Folge G auf A ist eine Funktion $\cdot : A^* \times A^* \to A^*$ definiert als*

$$\langle a_1, \ldots, a_n \rangle \cdot \langle b_1, \ldots, b_n \rangle = \langle a_1, \ldots, a_n, b_1, \ldots, b_n \rangle.$$

Definition A.7. *Für eine Relation R auf einer beliebigen Menge A werden folgenden Eigenschaften definiert.*

Die Relation R heißt reflexiv *genau dann, wenn $\forall x \in A : x \mathrm{R} x$.*

Die Relation R heißt irreflexiv *genau dann, wenn $\nexists x \in A : x \mathrm{R} x$.*

Die Relation R heißt symmetrisch *genau dann, wenn $\forall x, y \in A : x \mathrm{R} y \iff y \mathrm{R} x$.*

Die Relation R heißt antisymmetrisch *genau dann, wenn $\forall x, y \in A : x \mathrm{R} y \wedge y \mathrm{R} x \implies x = y$.*

Die Relation R heißt transitiv *genau dann, wenn $\forall x, y, z \in A : x \mathrm{R} y \wedge y \mathrm{R} z \implies x \mathrm{R} z$.*

Die Relation R heißt Äquivalenzrelation *genau dann, wenn R reflexiv, symmetrisch und transitiv ist.*

Definition A.8. *Eine Relation R auf einer Menge A kann als Ordnung aufgefasst werden. Dabei gelten folgende Begriffe.*

Zwei Elemente $x, y \in A$ heißen vergleichbar *genau dann, wenn $x \mathrm{R} y \vee x = y \vee y \mathrm{R} x$.*

Die Relation R heißt Quasiordnung *genau dann, wenn R reflexiv und transitiv ist.*

Die Relation R heißt Halbordnung *genau dann, wenn R reflexiv, transitiv und antisymmetrisch ist.*

Die Relation R heißt Striktordnung *genau dann, wenn R irreflexiv und transitiv ist.*

Eine Halbordnung bzw. Striktordnung R heißt totale Halbordnung *bzw.* totale Striktordnung *genau dann, wenn je zwei Elemente miteinander vergleichbar sind.*

Definition A.9. *Für eine Relation R auf einer beliebigen Menge A werden folgenden Eigenschaften definiert.*

Die Relation R heißt wohlfundiert *genau dann, wenn es keine unendliche R-absteigende Folge gibt.*

Die Relation R heißt wohlfundierte Ordnung *genau dann, wenn R eine wohlfundierte Striktordnung ist.*

Die Relation R heißt Wohlordnung *genau dann, wenn R eine wohlfundierte, totale Striktordnung ist.*

Die Relation R heißt wohlfundierte Quasiordnung *genau dann, wenn R eine Quasiordnung ist, deren strikter, d.h. irreflexiver Teil, wohlfundiert ist.*

Definition A.10. *Eine Quasiordnung $\geq$ auf einer Menge von Termen* $\mathrm{Ter}(\Sigma)$ *über eine Signatur Σ heißt* schwach monoton *genau dann, wenn für alle $t, u \in \mathrm{Ter}(\Sigma)$ und $f \in \Sigma$ gilt*

$$t \geq u \implies f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n) \geq f(s_1, \ldots, s_{i-1}, u, s_{i+1}, \ldots, s_n)$$

Definition A.11. *Eine Relation R auf einer Menge A heißt* zyklisch *genau dann, wenn es eine Folge $\langle a_1, a_2, \ldots, a_n \rangle$ von Elementen aus A gibt, sodass $a_i \,\mathrm{R}\, a_{i+1}$ für alle $1 \leq i < n$ und $a_1 = a_n$. Eine solche Folge heißt* Kreis *oder* Zyklus *auf R. Eine Relation R die nicht zyklisch ist, heißt* azyklisch.

Bemerkung A.12. *Jede reflexive Relation ist zyklisch wegen der Folge $\langle a, a \rangle$.*

Bemerkung A.13. *Seien R und R' Relationen auf einer Menge A und ferner $R' \subseteq R$ sowie R azyklisch. Dann ist R' ebenfalls azyklisch.*

Beweis. Sei die Relation R' zyklisch mit einem Kreis $\langle a_1, a_2, \ldots, a_n \rangle$ auf R'. Dann muss $\langle a_1, a_2, \ldots, a_n \rangle$ aber auch ein Kreis auf R sein, da $a_i \,\mathrm{R}\, a_{i+1}$ aus $a_i \,\mathrm{R}'\, a_{i+1}$ wegen $R' \subseteq R$ folgt. Somit wäre R zyklisch im Widerspruch zur Voraussetzung. □

Definition A.14. *Sei A eine Menge, $R \subseteq A \times A$ eine Relation auf A und F eine Folge auf A der Form $\langle a_1, a_2, \ldots, a_n \rangle$, dann heißt F* topologische Sortierung *bezüglich R falls für alle Paare von Indices $1 \leq i < j \leq n$ gilt $(a_j, a_i) \notin R$.*

Definition A.15. *Für eine gegebene natürliche Zahl $n \in \mathbb{N}$ heißt die bijektive Abbildung $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$* Permutation. *Für eine Folge F über einer Menge A mit $F = \langle a_1, \ldots, a_n \rangle$ der Länge n ergibt sich die* Permutation π der Folge F *als*

$$\pi(F) = \langle a_{\pi(1)}, \ldots, a_{\pi(n)} \rangle.$$

Quelltextauszüge

5.1 Die Funktion `infinity` 60
5.2 Die Funktion take 63

6.1 Der Konditional-Kombinator 76
6.2 Das Countdown-Programm in C 76
6.3 Das Countdown-Programm in Haskell 77
6.4 Das Countdown-Programm in Haskell 82
6.5 Implementierung der Transitionsmonade 84
6.6 Repräsentation der Variablenbelegung als Tabelle 87
6.7 Countdown als Variablenproblem 88
6.8 Algebraische Datentypen für jede Klasse 92
6.9 Die Methode m als Funktion in Haskell 92
6.10 Der Typklassen-Ansatz 93
6.11 Eine nicht vollständige polymorphe Implementierung 94
6.12 Korrekte Polymorphieimplementierung 95
6.13 Rückgabewerte im Typklassenansatz 96
6.14 Rückgabewerte kombiniert mit polymorphen Parametern 96
6.15 Uniforme Objektmodellierung 98
6.16 Countdown im Continuation Passing Stil 100
6.17 Strukturierung durch lokale Funktionen 101
6.18 Rumpffunktion mit Heap 102
6.19 Graph mit reflexiver Kante in Adjazenzlistendarstellung 103

Abbildungsverzeichnis

6.1 Der Terminierungsgraph der Funktion `countdown` 78
6.2 Alternativer Terminierungsgraph für `countdown` 80
6.3 Automatisch erzeugter Terminierungsgraph für `countdown` mit ausgerolltem `while`-Kombinator. 83

Literaturverzeichnis

[AAC+08] Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla und Damiano Zanardini. Termination analysis of Java bytecode. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS '08, S. 2–18, Berlin Heidelberg, Deutschland, 2008. Springer-Verlag.

[AG00] Thomas Arts und Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[All70] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.

[All86] Lloyd Allison. *A practical introduction to denotational semantics.* Cambridge University Press, New York, New York, United States of America, 1986.

[App97] Andrew W. Appel. *Modern compiler implementation in ML: Basic techniques.* Cambridge University Press, New York, New York, United States of America, 1997.

[App98] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[Art97] Thomas Arts. *Automatically proving termination and innermost termination of term rewriting systems.* PhD thesis, Universiteit Utrecht, 1997.

[ASU88] Alfred V. Aho Aho, Ravi Sethi und Jeffrey D. Ullman. *Compilers: Principles, techniques and tools.* Addison-Wesley Publishing, Reading, Massachusets, United States of America, 1988.

[AWZ88] Bowen Alpern, Mark Wegman und Frank Kenneth Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th*

ACM SIGPLAN-SIGACT symposium on Principles of programming languages, S. 1–11, New York, New York, United States of America, 1988. ACM.

[BCDO06] Josh Berdine, Byron Cook, Dino Distefano und Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, S. 386–400, Berlin Heidelberg, Deutschland, 2006. Springer-Verlag.

[Ber66] Andrew J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[Bir98] Richard Bird. *Introduction to functional programming using Haskell*. Prentice Hall, Harlow, United Kingdom, 2. Ausgabe, 1998.

[BK03] Marc Bezem und Jan Willem Klop. Abstract reduction systems. In Marc Bezem, Jan Willem Klop und Roel de Vrijer (Hrsg.), *Term rewriting systems*. Cambridge University Press, Cambridge, United Kingdom, 2003.

[Blo11] Marco Block. *Haskell-Intensivkurs : Ein kompakter Einstieg in die funktionale Programmierung*. Xpert.press. Springer-Verlag, Berlin Heidelberg, Deutschland, 2011.

[BN99] Franz Baader und Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, New York, United States of America, 1999.

[BOvEG10] Marc Brockschmidt, Carsten Otto, Christian von Essen und Jürgen Giesl. Termination graphs for Java bytecode. In Simon Siegler und Nathan Wasser (Hrsg.), *Verification, Induction Termination Analysis*, Lecture Notes in Computer Science, S. 17–37. Springer-Verlag, Berlin Heidelberg, Deutschland, 2010.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman und F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[CPR06] Byron Cook, Andreas Podelski und Andrey Rybalchenko. Terminator: beyond safety. In Thomas Ball und RobertB. Jones (Hrsg.), *Computer Aided Verification*, Bd. 4144 *Lecture Notes in Computer Science*, S. 415–418. Springer-Verlag, Berlin Heidelberg, Deutschland, 2006.

[CPR11] Byron Cook, Andreas Podelski und Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[Der81] Nachum Dershowitz. Termination of linear rewriting systems. *Lecture Notes in Computer Science*, 115:448–458, 1981.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Dij76] Edsger W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, New Jersey, United States of America, 1976.

[Dob12] Ernst-Erich Doberkat. *Haskell - Eine Einführung für Objektorientierte.* Oldenbourg Wissenschaftsverlag, München, Deutschland, 2012.

[Erw01] Martin Erwig. Inductive Graphs and Functional Graph Algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code.* Addison-Wesley Longman Publishing, Boston, Massachusetts, United States of America, 1999.

[GJM03] Carlo Ghezzi, Mehdi Jazayeri und Dino Mandrioli. *Fundamentals of software engineering.* Prentice-Hall, Upper Saddle River, New Jersey, United States of America, 2003.

[GJS+13] James Gosling, Bill Joy, Guy Steele, Gilad Bracha und Alex Buckley. *The Java language specification, Java SE 7 Edition.* Addison-Wesley Professional Publishing, Redwood City, California, United States of America, 2013.

[GRSK+11] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski und René Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):7:1–7:39, February 2011.

[GSSKT06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp und René Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In Frank Pfenning (Hrsg.), *Term Rewriting and Applications*, Bd. 4098 *Lecture Notes in Computer Science*, S. 297–312. Springer-Verlag, Berlin Heidelberg, Deutschland, 2006.

[GTSK05] Jürgen Giesl, René Thiemann und Peter Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In Franz Baader und Andrei Voronkov (Hrsg.), *Logic for Programming, Artificial Intelligence, and Reasoning*, Bd. 3452 *Lecture Notes in Computer Science*, S. 301–331. Springer-Verlag, Berlin Heidelberg, Deutschland, 2005.

[GTSKF04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp und Stephan Falke. Automated termination proofs with AProVE. In Vincent Oostrom (Hrsg.), *Rewriting Techniques and Applications*, Bd. 3091 *Lecture Notes in Computer Science*, S. 210–220. Springer-Verlag, Berlin Heidelberg, Deutschland, 2004.

[Hin69] J. Roger Hindley. The principle type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[HL78] Gérard Huet und Dallas Lankford. On the uniform halting problem for term rewriting systems. Rapport de Recherche 283, Institut de Recherche d'Informatique et d'Automatique, Rocquencourt, Le Chesnay, France, 1978.

[JP99] Mark P. Jones und John Peterson. The Hugs 98 user manual. 1999. Verfügbar unter *http://www.haskell.org/hugs*, abgerufen am 2. Januar 2014.

[KB69] Donald Ervin Knuth und Peter B. Bendix. Simple word problems in universal algebras. In John Leech (Hrsg.), *Computational problems in abstract algebra*, S. 263–297. Pergamon Press, 1969.

[KdV03a] Jan Willem Klop und Roel de Vrijer. First-order term rewriting systems. In Marc Bezem, Jan Willem Klop und Roel de Vrijer

(Hrsg.), *Term rewriting systems.* Cambridge University Press, Cambridge, United Kingdom, 2003.

[KdV03b] Jan Willem Klop und Roel de Vrijer. Properties of rewriting: decidability and modularity. In Marc Bezem, Jan Willem Klop und Roel de Vrijer (Hrsg.), *Term rewriting systems.* Cambridge University Press, Cambridge, United Kingdom, 2003.

[Kel95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on intermediate representations*, S. 13–22, New York, New York, United States of America, 1995. ACM.

[Ker88] Brian W. Kernighan. *The C programming language.* Prentice Hall Professional Technical Reference, 2nd Ausgabe, 1988.

[KKP+81] David J. Kuck, Robert Henry Kuhn, David Padua, Bruce Leasure und Michael Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, S. 207–218, New York, New York, United States of America, 1981. ACM.

[KN06] Gerwin Klein und Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[Lip11] Miran Lipovaca. *Learn you a Haskell for great good!: A beginner's guide.* No Starch Press, San Francisco, California, United States of America, 1. Ausgabe, 2011.

[LJ03] Ralf Lämmel und Simon Peyton Jones. Scrap Your Boilerplate: A practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, S. 26–37, New York, New York, United States of America, 2003. ACM.

[LY99] Tim Lindholm und Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing, Boston, Massachusetts, United States of America, 2. Ausgabe, 1999.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4):184–195, 1960.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[MTH90] Robin Milner, Mads Tofte und Robert Harper. *Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, United States of America, 1990.

[NNH99] Flemming Nielson, Hanne Ries Nielson und Chris Hankin. *Principles of program analysis*. Springer-Verlag, Berlin Heidelberg, Deutschland, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson und Markus Wenzel. *Isabelle/HOL — A proof assistant for higher-order logic*, Bd. 2283 *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, Deutschland, 2002.

[NW06] Maurice Naftalin und Philip Wadler. *Java Generics and Collections*. O'Reilly Media, Inc., Sebastopol, California, United States of America, 2006.

[Ohl02] Enno Ohlebusch. *Advanced topics in term rewriting*. Springer Publishing Company, Incorporated, New York, New York, United States of America, 2002.

[OSG09] Bryan O'Sullivan, Donald Stewart und John Goerzen. *Real world Haskell*. O'Reilly Media, Inc., Sebastopol, California, United States of America, 2. Ausgabe, 2009.

[PJ02] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy und Ralf Steinbruggen (Hrsg.), *Engineering Theories of Software Construction*, S. 47–96. IOS Press, 2002.

[PJ03] Simon Peyton-Jones. *Haskell 98 language and libraries: The revised report*. Cambridge University Press, Cambridge, United Kingdom, 2003.

[Plo81] Gordon David Plotkin. A structural approach to operational semantics. Technical report, DAIMI FN-19, University of Aarhus, Aarhus, Danmark, 1981.

[Rey02] John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, S. 55–74, Washington, District of Columbia, United States of America, 2002. IEEE Computer Society.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):pp. 358–366, 1953.

[RWZ88] B. K. Rosen, M. N. Wegman und F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, S. 12–27, New York, NY, USA, 1988. ACM.

[Sch08] Jan Schultze. Konzeption und Realisierung eines SETL → Haskell Übersetzers. Diplomarbeit, Technische Universität Dortmund, 2008.

[Sch10] Jan Schultze. Semantically equivalent permutations of statement sequences including IO operations. Technical report, Technische Universität Dortmund, Dortmund, Deutschland, 2010.

[SDSD86] Jacob T. Schwartz, Robert B. Dewar, Edmond Schonberg und Ed Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag, New York, New York, United States of America, 1986.

[SGG00] Abraham Silberschatz, Peter Galvin und Greg Gagne. *Applied operating system concepts*. John Wiley & Sons Inc., New York, New York, United States of America, 2000.

[SMP10] Fausto Spoto, Fred Mesnard und Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3):8:1–8:70, 2010.

[Str00a] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, 2000.

[Str00b] Bjarne Stroustrup. *The C++ programming language.* Addison-Wesley Longman Publishing, Boston, Massachusetts, United States of America, 3. Ausgabe, 2000.

[SW00] Christopher Strachey und Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, 2000.

[Swi05] Stephan Swiderski. Terminierungsanalyse von Haskellprogrammen. Diplomarbeit, RWTH Aachen, 2005.

[Tho78] Ken Thompson. Unix Implementation. *Bell System Technical Journal*, 57:1931–1946, 1978.

[Toy87] Yoshihito Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, Mai 1987.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[Wad89] Philip Wadler. Theorems for Free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, S. 347–359, New York, New York, United States of America, 1989. ACM.

[Weg99] Ingo Wegener. *Theoretische Informatik - eine algorithmische Einführung.* Teubner, Stuttgart, Deutschland, 2. Ausgabe, 1999.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, Massachusetts, United States of America, 1993.

[Wir71] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[Zan03] Hans Zantema. Termination. In Marc Bezem, Jan Willem Klop und Roel de Vrijer (Hrsg.), *Term rewriting systems.* Cambridge University Press, Cambridge, United Kingdom, 2003.