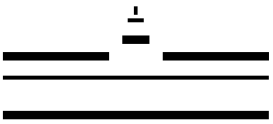


Diplomarbeit

Optimale Analyse gewichteter dynamischer Push-Down Netzwerke

Westfälische Wilhelms-Universität Münster
Fachbereich Mathematik und Informatik
Institut für Informatik

Thema gestellt von
Prof. Dr. Müller-Olm

vorgelegt von
Alexander Wenner
24. November 2009

Inhaltsverzeichnis

1 Einleitung **5**

2 Grundlagen **9**

2.1 Datenflussanalyse 9

2.2 Gewichtete Push-Down Systeme 18

2.3 Dynamische Push-Down Netzwerke 33

3 Analyse gewichteter dynamischer Pushdown-Netzwerke **41**

3.1 Ansatz 41

3.2 Charakterisierung der Pfade 70

3.3 Abstrakte Interpretation 83

4 Anwendungsbeispiele **97**

4.1 Bitvektor-Probleme 97

4.2 Kürzeste Pfade 103

5 Zusammenfassung und Ausblick **109**

Literaturverzeichnis **113**

Abbildungsverzeichnis

2.1 Beispiel für einen intraprozeduralen Flussgraphen 11
2.2 Formulierung des Ungleichungssystems einer Analyse für die erreichenden Pfade 16
2.3 Zurordnung der Transferfunktionen und Formulierung, sowie Lösung, des Ungleichungssystems einer Analyse für die Verfügbarkeit des Ausdrucks $x-1$ 17
2.4 Umwandlung eines interprozeduralen Flussgraphen $\mathcal{G}$ in ein PDS $\mathcal{P}$ 20
2.5 $\mathcal{P}$-Automat für die Menge der Konfigurationen $pe_{rec}(N_{\mathcal{G}})^*$ des in Abbildung 2.4 dargestellten PDS $\mathcal{P}$ zum Flussgraphen $\mathcal{G}$ 23
2.6 Beispiele für Ableitungsbäume des Symbols $PopSeq_{(p,e_{main},s)}$ aus der Grammatik zum PDS $\mathcal{PA}$ und die Teilpfade des ursprünglichen PDS, der zu den Ableitungsbäumen gehörigen Popsequenzen für e_{main} von p nach s . . 26
2.7 $\mathcal{P}^*$-Automat, der durch Saturierung des in Abbildung 2.5 betrachteten $\mathcal{P}$-Automaten entsteht 29
2.8 Annotierter $\mathcal{P}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, des in Abbildung 2.5 betrachteten $\mathcal{P}$-Automaten $\mathcal{A}$ entsteht 34
2.9 Umwandlung eines parallelen interprozeduralen Flussgraphen $\mathcal{G}$ in ein DPN $\mathcal{M}$ 36
2.10 $\mathcal{M}$-Automat für die Menge der Konfigurationen $(pN_{\mathcal{G}}^*\#)^*pe_{rec}N_{\mathcal{G}}^*(\#pN_{\mathcal{G}}^*)^*$ des in Abbildung 2.9 dargestellten DPN $\mathcal{M}$ zum Flussgraphen $\mathcal{G}$ 38
2.11 Automat, der durch Saturierung des in Abbildung 2.10 betrachteten Automaten entsteht 39

3.1 Beispiele für zwei Pfade von $p\gamma_1\gamma_2$ nach $\overline{v}\overline{c}$, die die gleichen Popsequenzen ausführen, und die zugehörige Darstellung als Baum. 44
3.2 Aufteilung der Ausführung eines Baumes in Pop- und Pushphase 60
3.3 Aufteilung der Ausführung einer Popphase in Popoperationen 63

4.1 Annotierter $\mathcal{M}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, des in Abbildung 2.10 betrachteten $\mathcal{M}$-Automaten entsteht 105
4.2 Annotierter $\mathcal{M}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, aus dem abgebildeten $\mathcal{M}$-Automaten entsteht 106
4.3 Annotierter $\mathcal{M}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, des in Abbildung 2.10 betrachteten $\mathcal{M}$-Automaten $\mathcal{A}$ entsteht 107

5.1 Beispiel für Pfade die nacheinander Konfigurationsmengen durchlaufen und Gewichtung die die Berechnung einer Rückwärtsanalyse an einem Programmpunkt ermöglichen 111

KAPITEL 1

Einleitung

Die Programmierung paralleler Programme hat in den letzten Jahren einen immer höheren Stellenwert in der Softwareentwicklung eingenommen. Gefördert wird dies durch die breite Verfügbarkeit paralleler Prozessoren, sowie die Integration paralleler Programmkonstrukte durch moderne Programmiersprachen wie zum Beispiel *Java*. Parallele Programmierung hat jedoch den großen Nachteil, dass durch die nebenläufige Ausführung mehrerer Kontrollflüsse schnell die Komplexität und damit die Anfälligkeit für Fehler steigt. Insbesondere tritt dies bei Verwendung von parallelen Konstrukten auf, in denen sich verschiedene Prozesse einen gemeinsamen Speicher teilen, auf, da sich die Prozesse nicht nur durch expliziten Nachrichtenaustausch, sondern auch durch eine einfache Variablenzuweisung, beeinflussen können. Das kann sich darin äußern, dass Threads, parallel ablaufende Programme die sich einen Adressraum teilen, sich unbeabsichtigt gegenseitig beeinflussen und dadurch Fehler verursachen. Ein Problem ist es also Verfahren zu entwickeln, die helfen solche Fehler zu finden und den Entwickler unterstützen fehlerfreie Programme zu entwerfen. Ein weiteres Problem ist die Optimierung der Ausführung paralleler Programme. Compiler für sequentielle Programme sind darauf ausgelegt, den Code in möglichst optimierter Form zu übersetzen, um eine schnelle Ausführung zu ermöglichen. Dabei werden Techniken angewendet, die nicht ohne weiteres auf parallele Programme übertragen werden können. So kann in einem sequentiellen Kontext eine Variablenzuweisung, die im folgenden Verlauf des Programms nicht benötigt wird, einfach entfernt werden. Auch können voneinander unabhängige Anweisungen vertauscht werden, um den Ablauf des Programms zu beschleunigen. Wenn nun in einem parallelen Umfeld ein Thread einer Variablen einen Wert zuweist, diese danach aber nicht weiter verwendet, kann diese Zuweisung im allgemeinen nicht gelöscht werden. Die Variable wird vielleicht von einem parallel laufenden Thread benötigt. Auch das Vertauschen

von Anweisungen kann dazu führen, dass ein parallel laufender Thread gestört wird, da ein benötigter Wert erst zu einem späteren Zeitpunkt zur Verfügung steht.

Um diese Probleme zu behandeln, beschäftigt man sich vermehrt mit der Analyse paralleler Programme. Insbesondere im Bereich der *Datenflussanalyse* beschäftigen sich viele Arbeiten mit der Analyse von Programmen mit parallelen Prozeduraufrufen [SS00, MO04, MO06, LMO07, EK99, EP00] oder auch mit der Analyse von Programmen mit dynamischer Threaderzeugung [LMO07]. Datenflussanalyse beschäftigt sich mit der Gewinnung von Laufzeitinformationen an bestimmten Stellen des Programms, ohne das Programm jedoch auszuführen. Die meisten dieser Verfahren sind entweder Fixpunkt-basiert oder Automaten-basiert. Fixpunkt-basierte Verfahren [SS00, MO04, MO06, LMO07] definieren eine Funktion, die die gewünschten Informationen als ihren größten Fixpunkt beschreibt. Dieser kann dann durch bekannte Techniken berechnet werden. Automaten-basierte Techniken [EK99, EP00, BMOT05] greifen auf Techniken aus dem *Model-Checking* zurück, welche sich mit der Erreichbarkeit von Mengen von Konfigurationen gewisser Modelle beschäftigen. Da dabei in der Regel die Datenflussfakten in die Elemente der Konfigurationen kodiert werden, können dabei nur *Bitvektor-Probleme*, eine bestimmte Klasse von Datenflussanalysen, die nur auf den Werten wahr oder falsch arbeiten, oder gewisse andere Analysen auf einem endlichen Wertebereich abgebildet werden.

Da Programmanalyse Eigenschaften von Programmen berechnet, ist grundsätzlich zu beachten, dass nach dem *Satz von Rice* alle nichttrivialen semantischen Eigenschaften von Programmen einer berechnungsuniversellen Programmiersprache unentscheidbar sind. Als Konsequenz daraus sind also automatische Analysen semantischer Eigenschaften für berechnungsuniverselle Programmiersprachen, die korrekt und vollständig sind, nicht möglich. In der Regel wird daher auf die Vollständigkeit der Analyse verzichtet und eine Approximation berechnet.

Die Überlegungen dieser Arbeit basieren auf dem in [RSJM05] untersuchten Ansatz für die Analyse von sequentiellen Programmen mit Prozeduren. Dort wird ein Verfahren vorgestellt, dass Automaten-basierte Techniken aus dem Bereich des Model-Checking mit Fixpunkt-basierten Methoden der Datenflussanalyse kombiniert und daher neben den üblichen Bitvektor-Problemen auch ein breites Spektrum anderer Datenflussanalysen ermöglicht. Dabei werden *gewichtete Push-Down Systeme* (WPDS) zu Grunde gelegt, in denen jeder Transitionsregel eines *Push-Down Systems* (PDS) ein Gewicht zugeordnet ist. Ein Gewicht einer Transition entspricht dabei im konkreten Fall den untersuchten Datenflussinformationen, die bei der Anwendung dieser Transition beachtet werden müssen. Auf Basis dieser Gewichte werden dann *"Meet Over all Path"-Lösungen* (MOP-Lösungen) für spezielle Mengen von Ausführungspfaden des PDS gebildet. Es werden also Pfade, Folgen von Transitionen, des PDS betrachtet und dabei die Gewichte der einzelnen Transitionen verknüpft. PDS können durch ihren Stack auf natürliche Weise Programme mit Prozeduren, unter Berücksichtigung des korrekten Aufrufkontextes, abbilden und erlauben zudem die Modellierung unbedingter Verzweigung. Dies macht sie zur passenden Grundlage für die Untersuchung sequentieller Programme. Der Verzicht auf bedingte Verzweigung ist dabei eine gängige Approximation in

der Datenflussanalyse, da sonst viele Probleme unberechenbar werden. Die korrekte Abbildung von Prozeduraufrufen ermöglicht eine präzise kontextsensitive Analyse, die in diesem Bereich das Verhalten eines Programms exakt nachbildet. Es wird dann gezeigt, dass mit Hilfe dieses Ansatzes klassische Datenflussanalysen durchgeführt werden können, bei denen für jeden Programmpunkt die MOP-Lösung aller eingehenden oder ausgehenden Pfade des Programmpunktes berechnet wird. Desweiteren bietet die Modellierung als WPDS auch die Möglichkeit, die Anfragen nicht nur auf einen Programmpunkt zu beziehen, sondern auch den restlichen Aufrufkontext, also welche Prozeduraufrufe bis zu diesem Punkt durchgeführt wurden, festzulegen. Dadurch lassen sich spezifischere Anfragen stellen und das Verfahren ist eine echte Erweiterung der klassischen Techniken zur Datenflussanalyse.

Es existieren bereits Ansätze die Ergebnisse für WPDS auch für parallele Programme mit mehreren Threads auszunutzen. So wird zum Beispiel in [LTKR08] jeder Thread durch ein WPDS dargestellt. In dieser Modellierung kann immer nur ein WPDS Transitionen ausführen, dann erfolgt ein Kontextwechsel und ein anderer Thread erhält diese Möglichkeit. Diese einzelnen Phasen der Ausführungen werden dabei nun durch WPDS modelliert und die Ergebnisse dann verknüpft. Dabei ist jedoch nur eine Betrachtung von endlich vielen Kontextwechseln möglich.

In dieser Arbeit verfolgen wir nun den Ansatz, das bei WPDS zu Grunde liegende Modell durch ein anderes Modell, das zusätzlich die Modellierung von parallel laufenden Threads ermöglicht, auszutauschen. Dazu untersuchen wir die Möglichkeit die in [LTKR08] untersuchten Techniken für gewichteten Push-Down Systeme auf die in [BMOT05] eingeführten *dynamischen Push-Down Netzwerke* (DPN) zu übertragen und *gewichtete dynamische Push-Down Netzwerke* (WDPN) zu erhalten. Dynamische Push-Down Netzwerke sind eine Erweiterung von PDS. Hierbei wird nicht mehr nur ein einzelner Stack betrachtet, sondern ein ganzes Netzwerk von Stacks. Jeder der dabei betrachteten Stacks kann, unabhängig von den anderen, Transitionen des DPN ausführen und auch neue Stacks erzeugen. Wenn wir nun im oben kurz beschriebenen Verfahren das zu Grunde liegende Modell eines PDS durch das Modell eines DPN austauschen, erhalten wir eine Methode zur Analyse von Programmen mit Prozeduren und dynamischer Threaderzeugung, da diese vom Modell der DPN in natürlicher Weise abgebildet werden können. Eine Modellierung von Synchronisation zwischen den Abläufen auf den einzelnen Stacks ist nicht möglich. Da jedoch bekannte Arbeiten zeigen, dass eine Analyse die Prozeduren und Synchronisation beachtet im Allgemeinen nicht berechenbar ist [Ram00], ist es auch hier eine übliche Vorgehensweise, auf die Modellierung von Synchronisation zu verzichten. Wir werden dann zeigen, dass sich gewisse Datenflussanalysen auf dieses neue Modell abbilden lassen und auch hier die Berechnung der klassischen MOP-Lösung an Programmpunkten möglich ist. Dabei sind auf Grund bekannter Ergebnisse jedoch bestimmte Analysen, wie Konstantenpropagation [MO04], schon durch den Übergang zu parallelen Programmen unberechenbar und scheiden für die weitere Betrachtung aus. Genau wie im Modell der WPDS erhalten wir dann für WDPN die Möglichkeit, Anfragen nicht nur auf einen Programmpunkt zu beziehen sondern auch hier den Kontext mit einzubinden. Dabei bezieht sich der Kontext hier nicht nur auf die Prozeduraufrufe des untersuchten

Threads, sondern auch auf die Zustände aller parallel laufenden Threads. Das Verfahren stellt damit eine echte Erweiterung des klassischen Verfahrens da.

Die weitere Arbeit gliedert sich in vier weitere Kapitel. In Kapitel 2 werden die bisher bekannten Ergebnisse zu DPN und WPDS als Grundlagen für die weiteren Betrachtungen zusammengefasst. Desweiteren wird kurz auf die klassische Datenflussanalyse und die dabei verwendete Technik der abstrakten Interpretation eingegangen. Im darauf folgenden Kapitel 3 betrachten wir dann die Erweiterung der WPDS Techniken auf DPN und erhalten das Modell der WDPN. Diese Ergebnisse werden dann in Kapitel 4 in zwei Beispielen angewendet. Dabei betrachten wir zum einem die schon angesprochenen Bitvektor-Probleme und zum anderen die Berechnung kürzester Pfade in einem DPN. In Kapitel 5 folgt eine abschließende Betrachtung der Ergebnisse, sowie Überlegungen für mögliche Erweiterungen des Modells.

KAPITEL 2

Grundlagen

In diesem Kapitel fassen wir die Grundlagen und in der Literatur bekannten Ergebnisse zu Datenflussanalyse, WPDS und DPN zusammen, die für die weiteren Betrachtungen in Kapitel 3 benötigt werden. Im ersten Abschnitt 2.1 erfolgt eine kurze Einführung in das Gebiet der Datenflussanalyse und die Technik der *abstrakten Interpretation*, die später von uns verwendet werden wird. Im zweiten Abschnitt 2.2 werden dann die bisherigen Ergebnisse der Untersuchung von WPDS dargelegt. Dabei wird vor allem der Ansatz betrachtet, da dieser später auch für unsere Untersuchungen relevant ist. Im letzten Abschnitt 2.3 folgt dann noch eine kurze Vorstellung der bisherigen Resultate zur Untersuchung von DPN, die eine mögliche Übertragung der Techniken der Untersuchung von WPDS auf DPN nahelegen.

2.1 Datenflussanalyse

Die folgenden Definitionen und Aussagen orientieren sich an den Ausführungen in [NNH99]. Beweise der angesprochenen Sätze können dort, in teilweise anderer Notation, nachgelesen werden.

In der Datenflussanalyse ist es üblich, Programme als gerichtete Flussgraphen zu betrachten, die den Kontrollfluss innerhalb des Programms beschreiben. Dabei repräsentieren die Knoten des Graphen Programmpunkte, die während der Ausführung erreicht werden können, und die Kanten des Graphen Anweisungen des Programms, die zum Übergang von einem Programmpunkt zum nächsten führen. Verzweigung des Kontrollflusses wird durch Verzweigungen im Graphen dargestellt, wobei die Kanten des Graphen als Anweisung eine Bedingung enthalten, die beim Durchlaufen der Kante erfüllt sein muss. Bei Vorhandensein

von Prozeduren und Parallelität, in der Form von parallelen Prozeduraufrufen oder dynamischer Threaderzeugung, wird das Programm dann durch ein System von Graphen, einen Graphen für jede Prozedur, dargestellt. Threads können dabei als Prozeduren betrachtet werden, die ab dem Aufrufzeitpunkt parallel abgearbeitet werden. Ein Graph ist dabei als das Hauptprogramm $main$, das den Einstiegspunkt beim Start des Programms darstellt, ausgezeichnet. Für den Fall von Prozeduraufrufen kann man sich dann eine Pseudokante vorstellen, die vom Anfangspunkt der Aufrufkante zum Einstiegspunkt der aufgerufenen Prozedur zeigt, und eine weitere Kante, die vom Endpunkt der Prozedur zum Endpunkt der Aufrufkante zeigt. Ein Abarbeiten der Aufrufkante entspricht dann einem vollständigen Durchlauf der Pseudokanten und des Graphen der aufgerufenen Prozedur. Im Falle von Threadaufrufen existiert nur eine Pseudokante vom Anfangspunkt der Aufrufkante zum Einstiegspunkt des Threads. Ein Durchlaufen der Kante führt dann zu einem Durchlauf der Pseudokante durch einen parallel gestarteten Kontrollfluss und einem direkten Übergang zum Endpunkt der Kante durch den aktuellen Kontrollfluss. Formal definieren wir dann ein System von Flussgraphen $(Proc, (G_\pi)_{\pi \in Proc})$.

Definition 2.1. Sei $Proc$ eine endliche Menge von Prozedurnamen mit $main \in Proc$. Die Menge möglicher Anweisungen $Stmt$ besteht aus einer Menge BA von Basisanweisungen, z.B. Variablenzuweisungen bestehend aus arithmetischen Ausdrücken oder boolesche Ausdrücke für die bedingte Verzweigung, sowie einer Menge $CA = \{\text{call } \pi \mid \pi \in Proc\}$ von Anweisungen für Prozeduraufrufe und einer Menge $SA = \{\text{spawn } \pi \mid \pi \in Proc\}$ von Anweisungen für Threadaufrufe. Der *Flussgraph* einer Prozedur $\pi \in Proc$ ist dann ein Viertupel $G_\pi = (N_\pi, E_\pi, e_\pi, r_\pi)$ bestehend aus einer endlichen Menge von Programmpunkten N_π, einer Menge von mit Anweisungen benannten Kanten $E_\pi \subseteq N_\pi \times Stmt \times N_\pi$, einem Startpunkt e_π und einem Endpunkt r_π. Ein *System von Flussgraphen* $\mathcal{G} = (Proc, (G_\pi)_{\pi \in Proc})$ zur Beschreibung eines parallelen Programms mit Prozeduren enthält dann für jede Prozedur $\pi \in Proc$ einen Flussgraphen G_π. Dabei seien die Mengen von Programmpunkten zweier verschiedener Prozeduren disjunkt, also $N_\pi \cap N_{\tilde{\pi}} = \emptyset$ für $\pi \neq \tilde{\pi}$. $N_\mathcal{G} = \bigcup_{\pi \in Proc} N_\pi$ sei die Menge aller Programmpunkte und $E_\mathcal{G} = \bigcup_{\pi \in Proc} E_\pi$ die Menge aller Kanten des Systems von Flussgraphen.

Da wir im folgenden nur grundlegend die Vorgehensweisen und Techniken der Datenflussanalyse vorstellen wollen, beschränken wir uns auf die Betrachtung der einfachsten Form von Programmen. Für das untersuchte System von Flussgraphen $(Proc, (G_\pi)_{\pi \in Proc})$ gelte daher $Proc = \{main\}$, das Programm besteht also nur aus einer Prozedur. Desweiteren gelte für jede Kante $(n, s, \tilde{n}) \in E_{main}$, dass $s \in BA$, es gibt also keine Prozedur- oder Threadaufrufe. Diese Art der Analyse heißt *intraprozedural*, da man sich nur innerhalb einer Prozedur bewegt. Jedoch ist eine Ausweitung der im folgenden dargestellten Ergebnisse auch auf interprozedurale Analysen, Analysen mit Prozeduraufrufen, und parallele Analysen, Analysen mit Betrachtung paralleler Kontrollflüsse, möglich. Ansätze dazu kann man zum Beispiel in [NNH99, SS00, MO04, MO06, LMO07] nachlesen. Da das System von Flussgraphen demnach nur aus einem Flussgraphen besteht, sei dieser im folgenden mit $G = (N, E, e_{main}, r_{main})$ bezeichnet. Abbildung 2.1 zeigt ein Beispiel für einen intraprozeduralen Flussgraphen.

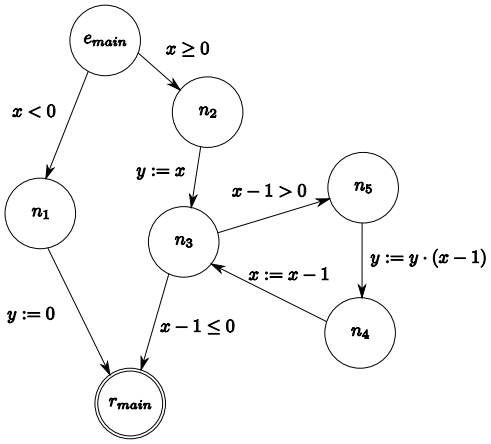


Abbildung 2.1: Beispiel für einen intraprozeduralen Flussgraphen

Datenflussanalyse bietet nun eine generische Methode um intraprozedurale Fixpunkt-basierte Programmanalysen zu konstruieren. Man betrachtet Pfade im Flussgraphen, die einer Ausführung des dargestellten Programms entsprechen, und untersucht wie sich dabei gewisse Informationen entlang dieser Pfade verhalten. Da die Pfade die Ausführung des Programms darstellen, werden diese Informationen auch mit Laufzeitinformationen bezeichnet. Datenflussanalyse berechnet diese nun ohne das Programm wirklich auszuführen.

Definition 2.2. Ein *Pfad* ist eine alternierende Folge

$$\nu = [n_1, s_1, n_2, \ldots, s_{n-1}, n_m]$$

von Zuständen $n_i \in N$ für $i \in \{1, \ldots, m\}$ und Anweisungen $s_i \in Stmt$ mit $(n_i, s_i, n_{i+1}) \in E$ für alle $i \in \{1, \ldots, m-1\}$, für $m \in \mathbb{N}$[1]. Es existiert kein allgemeiner leerer Pfad, für jeden Knoten n existiert jedoch ein Pfad $[n]$ der an diesem Knoten verweilt. Zusätzlich sein dann mit $\nu^+ = n_1$ der Anfangszustand und mit $\nu^- = n_m$ der Endzustand eines Pfades bezeichnet. $Paths_G$ sei die Menge aller Pfade des Flussgraphen.

Wir definieren einen Konkatenationsoperator ; , der zu zwei beliebigen Pfaden $\nu_1 = [\![n_1, s_1, \ldots, n_m]\!], \nu_2 = [\![n_m, s_n \ldots, n_k]\!] \in Paths_G$ mit $\nu_1^- = \nu_2^+$ einen zusammengesetzten Pfad

$$\nu_1; \nu_2 = [\![s_1, \alpha_1 \ldots, s_n, \alpha_n, \ldots, s_m]\!]$$

liefert. Für Mengen $M_1, M_2 \subseteq Paths_G$ von Pfaden mit $\nu_1^- = \nu_2^+$ für alle $\nu_1 \in M_1$, $\nu_2 \in M_2$ gelte $M_1; M_2 = \{\nu_1; \nu_2 \mid \nu_1 \in M_1, \nu_2 \in M_2\}$.

Die in der Einleitung angesprochene notwendige approximative Natur der Analyse spiegelt sich hier in der Behandlung von bedingter Verzweigung wieder. Bedingte Verzweigungen im Flussgraphen werden unbedingt betrachtet und es wird nichtdeterministisch jeder Zweig verfolgt. Es werden also auch Pfade betrachtet, die eigentlich nicht durchlaufen werden

[1] $\mathbb{N} = \{1, 2, 3, \ldots\}$

können. Dies wäre zum Beispiel der Fall, wenn sich zwei auf einem Pfad einander folgende Bedingungen gegenseitig ausschließen.

Ziel der Datenflussanalyse ist es, zu Programmpunkten gewisse Informationen zu berechnen. Dabei unterscheidet man in der Regel zwei Arten von Analysen, vorwärts und rückwärts. Vorwärtsanalyse berechnen dabei Informationen über den Zustand oder das Verhalten des Programms an einem Programmpunkt n, die sich aus dem Ablauf des Programms vom Start bis zum Erreichen des Programmpunktes ergeben. Dies erfordert die Betrachtung der Menge der Pfade ν vom Einstiegspunkt des Programms, also $\nu^+ = e_{main}$, bis zum gewünschten Programmpunkt, also $\nu^- = n$. Dabei wird in der Regel eine am Start vorliegende initiale Information durch die Ausführung von Anweisungen auf einem Pfad transformiert, woraus sich die Daten am untersuchten Programmpunkt ergeben. Dadurch erklärt sich auch die Benennung als Vorwärtsanalyse, da die Daten vorwärts durch das Programm propagiert werden. Rückwärtsanalyse betrachten hingegen Informationen die sich aus den möglichen Ausführungen vom betrachteten Programmpunkt n bis zum Endpunkt des Programms ergeben. Man betrachtet also die Menge der Pfade ν vom betrachteten Programmpunkt, also $\nu^+ = n$, bis zum Endpunkt, also $\nu^- = r_{main}$. Dabei wird in der Regel eine initiale Information, die am Ende des Programms gilt, rückwärts entlang der Ausführungen des Programms bis zum betrachteten Punkt propagiert, um die gewünschten Daten zu berechnen. Man betrachtet also die Anweisungen eines Pfades in umgekehrter Reihenfolge.

Zur Darstellung von Daten oder Informationen und deren Transformation verwendet man vollständige Verbände und monotone Funktionen darauf.

Definition 2.3. 1. Ein *vollständiger Verband* $(D, \oplus)$[2] ist ein Tupel bestehend aus einer Menge D, sowie einem idempotenten, kommutativen, assoziativen Schnittoperator $\oplus : D \times D \to D$ auf D. Dieser impliziert eine partielle Ordnung

$$d_1 \sqsubseteq d_2 \Leftrightarrow d_1 \oplus d_2 = d_1.$$

Bezüglich dieser Ordnung muss für alle $X \subseteq D$ die größte untere Schranke $\bigoplus X$ von X existieren.

2. Eine Funktion $f : D \to D$ auf einem vollständigen Verband $(D, \oplus)$ heißt *monoton*, wenn für $x, y \in D$ gilt $x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

Eine *Datenflussanalyse* $((D, \oplus), F)$ wird nun durch einen vollständigen Verband $(D, \oplus)$ und die Menge monotoner Funktionen F auf $(D, \oplus)$ beschrieben. Man kann sich auch auf die Betrachtung einer Teilmenge der monotonen Funktionen beschränken, die die Identität enthält und unter der Verknüpfung von Funktionen abgeschlossen ist.

Die Elemente des Verbandes beschreiben dabei die untersuchten Eigenschaften des Programms und werden *Datenflussfakten* genannt. Die auf dem Verband definierte Ordnung stellt dabei eine Bewertung der Informationen dar. So gilt $d_1 \sqsubseteq d_2$ dann, wenn d_2 eine präzisere

[2] Wir verwenden in Anlehnung an [RSJM05] hier $\oplus$ als Notation für den Schnittoperator, statt dem sonst üblichen $\sqcap$.

Information als d_1 ist, d_1 jedoch eine korrekte Approximation von d_2 ist. Da $(D, \oplus)$ ein vollständiger Verband ist, existiert demnach für jede Menge von Datenflussfakten $X \subseteq D$ eine präziseste korrekte Approximation $\bigoplus X$.

Die Funktionen in F werden *Transferfunktionen* genannt und beschreiben, wie sich die Datenflussfakten beim Übergang von einem Programmpunkt zu einem nachfolgende Programmpunkt verhalten. Die geforderte Monotonie der Funktionen lässt sich im Hinblick auf die Ordnung der Datenflussfakten dann so interpretieren, dass präzisere Informationen am Ausgangspunkt auch zu präziseren Informationen am Zielpunkt führen müssen.

Im Fall intraprozeduraler Analyse ordnet eine *Instanz* einer Datenflussanalyse $((D, \oplus), F)$ nun jeder Kante $e \in E$ des Flussgraphen der betrachteten Prozedur eine Transferfunktion $f_e \in F$ zu. Zudem ist ein initialer Datenflussfakt $init$ gegeben, der, je nach betrachteter Analyse, am Anfang oder am Ende des Programms gilt. Die Informationen werden dann, beginnend mit der initialen Information am Anfang oder Ende, durch die Transferfunktionen der Kanten entlang der Pfade transformiert und man kann für einen Programmpunkt den Schnitt, also die bestmögliche Information die alle ankommenden Informationen approximiert, berechnen. Dazu betrachtet man lediglich die Transformationen durch eine Kante von einem Programmpunkt zum nächsten. Gelten am Anfang der Kante gewisse Informationen, so gelten am Ende der Kante mindestens die durch die Transferfunktion der Kante transformierten Informationen. Um nun diesen Zusammenhang der an den verschiedenen Programmpunkte gültigen Fakten darzustellen, definiert man ein Ungleichungssystem über $(D, \oplus)$, dass für jeden Programmpunkt $n \in N$ einen Datenflussfakt $L[n] \in D$ an diesem Programmpunkt charakterisiert.

Definition 2.4. Zu einer Datenflussanalyse $((D, \oplus), F)$ und einer Instanz, gegeben durch einen Flussgraphen G, eine Funktion $f_e \in F$ zu jeder Kante $e \in E$ und einen initialen Datenflussfakt $init \in D$, definiert man für Vorwärtsanalysen das Ungleichungssystem:

$$\begin{aligned} L[e_{main}] &\sqsubseteq init \\ L[n] &\sqsubseteq f_e(L[\tilde{n}]) \text{ falls } e = (\tilde{n}, s, n) \in E \text{ für ein } s \in Stmt \end{aligned}$$

Für Rückwärtsanalysen wird lediglich e_{main} durch r_{main} ersetzt und die Kantenrichtung umgedreht.

Für jeden Knoten $n \in N$ existiert dann eine Funktion $f_n : D^{|N|} \to D$ mit

$$f_n((L[n])_{n\in N}) = \begin{cases} \bigoplus_{e=(\tilde{n},s,n)\in E \text{ für ein } s\in Stmt, \tilde{n}\in N} f_e(L[\tilde{n}]) \oplus init & \text{falls } n = e_{main} \\ \bigoplus_{e=(\tilde{n},s,n)\in E \text{ für ein } s\in Stmt, \tilde{n}\in N} f_e(L[\tilde{n}]) & \text{sonst} \end{cases}.$$

Alternativ kann man dann dass Ungleichungssystem mit Hilfe der Funktion $f : D^{|N|} \to D^{|N|}$ mit $f = (f_n)_{n\in N}$ als

$$(L[n])_{n\in N} \sqsubseteq f((L[n])_{n\in N})$$

ausdrücken. Dabei gilt für Tupel $(d_n^1)_{n\in N}, (d_n^2)_{n\in N} \in D^{|N|}$ die Relation

$$(d_n^1)_{n\in N} \sqsubseteq (d_n^2)_{n\in N} \Leftrightarrow d_n^1 \sqsubseteq d_n^2 \text{ für alle } n \in N.$$

Eine Lösung des Ungleichungssystem ist eine Familie $(\overline{L}[n])_{n \in N}$ mit $\overline{L}[n] \in D$ für $n \in N$, die alle Ungleichungen erfüllt, das heißt es gilt $(\overline{L}[n])_{n \in N} \sqsubseteq f((\overline{L}[n])_{n \in N})$. Für eine Lösung kann man intuitiv die Korrektheit der Formulierung, also die Tatsache, dass $\overline{L}[n]$ die gewünschten Datenflussinformationen an Programmpunkt n korrekt approximiert, erkennen. Die Informationen am Anfangspunkt müssen die initialen Fakten korrekt approximieren, da diese dort auf jeden Fall gültig sind. Zudem muss jeder Programmpunkt die durch die zugehörigen Anweisungen transformierten Informationen aller vorhergehenden Punkte korrekt approximieren, da diese Pfade durch eine Ausführung des Programms durchlaufen werden können. Man erhält also durch Lösen des Ungleichungssystems eine zumindest intuitiv korrekte Approximation der gewünschten Datenflussinformationen. Da man die präziseste Approximation, im besten Fall die genaue Beschreibung, der Fakten sucht, betrachtet man die größte aller möglichen Lösungen. Im folgenden sprechen wir von einer korrekten und präzisen Analyse, wenn die Lösung des Ungleichungssystems eine genaue Beschreibung der gesuchten Informationen ist.

Da die einzelnen Funktionen $f_e \in F$ für $e \in E$ monoton sind, sind auch die Komponentenfunktionen f_n für $n \in N$ und damit auch f monoton. Dann existiert nach dem *Fixpunkttheorem von Knaster-Tarski*, ein größter Fixpunkt von f auf dem vollständigen Verband $(D, \oplus)$. Man kann dann zeigen, dass dieser mit der größten Lösung des Ungleichungssystems übereinstimmt. Zur Berechnung des größten Fixpunktes einer Funktion existieren verschiedene Techniken, zum Beispiel *chaotische Iteration*. Auf gewissen Verbänden, die eine Terminierung garantieren, kann so eine Lösung des Ungleichungssystems berechnet werden. Für eine Übersicht sei hier auf [NNH99] verwiesen.

Um formal die Korrektheit und Präzision der berechneten Lösung zu untersuchen, existiert neben einem direkten Beweis, durch Betrachtung aller möglichen Pfade, die Methode der abstrakten Interpretation [NNH99, CC77]. Hierbei wird ausgehend von einer Datenflussanalyse, die als korrekt und präzise bekannt ist, die Korrektheit einer weiteren Datenflussanalyse gefolgert. Die Idee dabei ist, dass man auf der einen Seite eine starke Datenflussanalyse, die eine Vielzahl von Informationen enthält und für die die Korrektheit und Präzision gezeigt wurde betrachtet. Auf der anderen Seite betrachtet man eine neue Datenflussanalyse, deren Informationen sich aus den schon bekannten Informationen der starken Analyse ableiten lassen, indem man die neuen Informationen aus den schon bekannten starken Informationen extrahiert. Dann liefert die abstrakte Interpretation die Korrektheit und Präzision der zweiten Analyse. Dazu definiert man eine Abstraktionsfunktion, welche die neuen Informationen aus der bekannten Analyse extrahiert.

Definition 2.5. Gegeben seien zwei vollständige Verbände $(D, \oplus)$ und $(D^{\#}, \oplus^{\#})$, dann ist $\alpha : D \to D^{\#}$ eine Abstraktionsfunktion, wenn α universell distributiv ist, das heißt für alle $X \subseteq D$ gilt $\alpha(\bigoplus X) = \bigoplus^{\#} \{\alpha(x) \mid x \in X\}$.

Die universelle Distributivität garantiert dabei eine enge Verknüpfung der beiden Verbände durch α, näheres dazu findet man in [NNH99, CC77]. Mit Hilfe dieser Abstraktionsfunktion

kann man dann zu jedem Datenflussfakt der ursprünglichen Analyse den zugehörigen Datenflussfakt der neuen Analyse konstruieren. Für die Korrektheit und Präzision der neuen Analyse ergibt sich dann folgende Aussage.

Satz 2.6. *Gegeben seien zwei Instanzen zweier Datenflussanalysen über den Verbänden* $(D, \oplus)$ *und* $(D^\#, \oplus^\#)$*, beschrieben durch zwei Ungleichungssysteme*

$$\begin{array}{rcl} (L[n])_{n\in N} & \sqsubseteq & f((L[n])_{n\in N}) \text{ mit } L[n] \in D \text{ für alle } n \in N \\ (L^\#[n])_{n\in N} & \sqsubseteq^\# & f^\#((L^\#[n])_{n\in N}) \text{ mit } L^\#[n] \in D^\# \text{ für alle } n \in N, \end{array}$$

mit $f = (f_n)_{n\in N}$ *und* $f^\# = (f_n^\#)_{n\in N}$*. Seien* $(\overline{L}[n])_{n\in N}$ *und* $(\overline{L}^\#[n])_{n\in N}$ *die größten Lösungen dieser Ungleichungssysteme über* $(D, \oplus)$ *und* $(D^\#, \oplus^\#)$ *und* $\alpha : D \to D^\#$ *eine Abstraktionsfunktion. Dann gilt:*

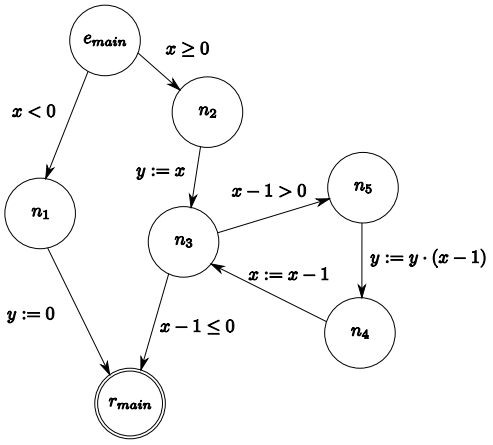
$$\alpha \circ f_n = f_n^\# \circ \alpha \text{ für alle } n \in N \Rightarrow \alpha(\overline{L}[n]) = \overline{L}^\#[n] \text{ für alle } n \in N$$

Die Aussage dieses auch als *Transferlemma* bekannten Satzes kann man dann im Rahmen der Datenflussanalyse interpretieren. Für eine Analyse, die sich als Abstraktion einer korrekten und präzisen Analyse schreiben lässt, liefert das sie beschreibende Ungleichungssystem, eine korrekte und präzise Beschreibung dieser Abstraktion, wenn sich das Ungleichungssystem mit der Abstraktion und dem Ungleichungssystem der ursprünglichen Analyse verträgt. Die genauen technischen Details können in [NNH99, CC77] nachgelesen werden.

Im folgenden betrachten wir zur Veranschaulichung dieses Ergebnisses ein Beispiel. Dabei legen wir als Ausgangsanalyse die Vorwärtsanalyse der erreichenden Pfade eines Programmpunktes zu Grunde. Für Rückwärtsanalysen kann man analog die verlassenden Pfade eines Programmpunktes betrachten. Der zu Grunde liegende Verband ist dann $(\mathcal{P}(Paths), \cup)$. Zu beachten ist hierbei, dass für $M_1, M_2 \in \mathcal{P}(Paths)$ gilt $M_1 \sqsubseteq M_2 \Leftrightarrow M_1 \supseteq M_2$, also die Ordnung auf dem Verband genau der umgekehrten Teilmengenbeziehung der Pfadmengen entspricht. Die Transferfunktionen F sind dann die monotonen Funktionen auf $(\mathcal{P}(Paths), \cup)$. Da es sich bei dem vom Übergang von einem Programmpunkt zum nächsten zu beschreibenden Effekt jedoch nur um eine Verlängerung der erreichenden Pfade des ersten Programmpunktes um die durchlaufene Kante handelt, finden nur Transferfunktionen der Form $f_e(M) = M;[n_m, s_m, n_{m+1}]$ für $e = (n_m, s_m, n_{m+1}) \in E$ Verwendung. In Abbildung 2.2 ist das Ungleichungssystem einer solchen Analyse für den schon vorher betrachteten Flussgraphen dargestellt.

Man kann nun mit Hilfe einer einfache Induktion direkt zeigen, dass die größte Lösung, also die Lösung mit den kleinsten Pfadmengen, für jeden Programmpunkt genau die erreichenden Pfade beschreibt. Dies ist auch intuitiv ersichtlich, da als Anfangsbedingung der leere Pfad den Startknoten erreichen muss und dann jeder weitere Knoten als erreichende Pfade die erreichenden Pfade eines Vorgängerknotens, verlängert um die sie verbindende Kante, enthalten muss.

Wir haben also ein Ungleichungssystem, dass zu jedem Programmpunkt die erreichenden Pfade im Flussgraphen beschreibt. Eine Berechnung der Lösung gestaltet sich auf Grund der



$$
\begin{array}{rcl}
L[e_{main}] & \supseteq & \{[e_{main}]\} \\
L[n_1] & \supseteq & L[e_{main}]; [e_{main}, x <= 0, n_1] \\
L[n_2] & \supseteq & L[e_{main}]; [e_{main}, x > 0, n_2] \\
L[n_3] & \supseteq & L[n_2]; [n_2, y = x, n_3] \\
L[n_3] & \supseteq & L[n_5]; [n_5, x = x - 1, n_3] \\
L[n_4] & \supseteq & L[n_3]; [n_3, x - 1 > 0, n_4] \\
L[n_5] & \supseteq & L[n_4]; [n_4, y = y * (x - 1), n_5] \\
L[r_{main}] & \supseteq & L[n_3]; [n_3, x - 1 <= 0, r_{main}] \\
L[r_{main}] & \supseteq & L[n_1]; [n_1, y = 0, r_{main}]
\end{array}
$$

Abbildung 2.2: Formulierung des Ungleichungssystems einer Analyse für die erreichenden Pfade

Struktur des Verbandes jedoch schwierig. Der Verband enthält unendlich aufsteigende Folgen von Mengen. Dies führt dazu, dass Verfahren, wie die chaotische Iteration, zur Berechnung der Lösung nicht terminieren.

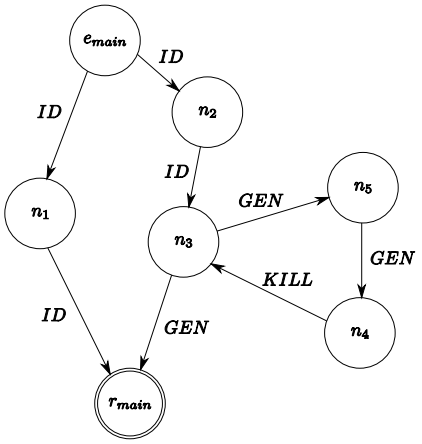
Eine Berechnung der Lösung auf dem Verband $(\mathcal{P}(Paths), \cup)$ ist jedoch gar nicht notwendig, da die Menge der erreichenden Pfade für die meisten Analysen zu viele Informationen enthält. Es werden alle Informationen über alle möglichen Ausführungen des Programms bis zu einem bestimmten Programmpunkt mitgeführt. Da wir jedoch wissen, dass das von uns konstruierte Ungleichungssystem eine korrekte und präzise Beschreibung der erreichenden Pfade ist, können wir mit Hilfe der abstrakten Interpretation daraus neue korrekte und präzise Datenflussanalysen ableiten, die sich effektiv berechnen lassen.

Ein Beispiel für eine Klasse von abgeleiteten Analysen sind Bitvektor-Probleme. Dabei wird, wie der Name schon sagt, ein Verband $(\{0,1\}^n, \oplus)$ zu Grunde gelegt. Jedes Bit des Vektors steht dabei für eine Eigenschaft, die an einem Programmpunkt entweder erfüllt oder nicht erfüllt, repräsentiert durch die Werte 1 und 0, ist. Die betrachteten Eigenschaften sind dabei unabhängig. Wir betrachten im folgenden also nur einen einstelligen Vektor, die Ergebnissen gelten analog aber auch für mehrstellige Vektoren. Bei der Definition des Schnittoperators $\oplus$ kann man zwei Arten von Problemen unterscheiden, da zwei Möglichkeiten existieren 0 und 1 anzuordnen. Zum einen betrachtet man $0 \sqsubseteq 1$. Im Kontext der Datenflussanalyse bedeutet dies dann, dass bei der Kombination zweier Beiträge zu einem Datenflussfakt die Nichterfüllung der Eigenschaft dominiert. Bei der Betrachtung von Pfaden muss also für eine Erfüllung der Eigenschaft an einem Programmpunkt, die Eigenschaft am Ende jedes Pfades der den Programmpunkt erreicht gültig sein. Diese Analysen nennen wir deshalb *sichere Bitvektor-Probleme*. Umgekehrt genügt für $1 \sqsubseteq 0$ die Gültigkeit der Eigenschaft am Ende eines erreichenden Pfades um diese am Programmpunkt zu erfüllen. Diese Analysen bezeichnen wir als *mögliche Bitvektor-Probleme*. Der Raum der monotonen Transferfunk-

tionen beschränkt sich auf Grund des einfachen Definitionsbereichs auf drei Funktionen $F = \{KILL : x \mapsto 0, ID : x \mapsto x, GEN : x \mapsto 1\}$.

Als konkretes Beispiel betrachten wir nun die Analyse der Verfügbarkeit des Ausdrucks $x - 1$ im schon betrachteten Beispielgraphen. Ein Ausdruck ist an einem Programmpunkt *verfügbar*, wenn er auf jedem erreichenden Pfad ausgewertet wurde und keiner seiner Komponenten danach ein neuer Wert zugewiesen wurde. Diese Information ist nützlich um festzustellen, ob ein Ausdruck an einem Punkt neu berechnet werden muss, oder durch Zwischenspeicherung eines vorherigen Ergebnisses eine erneute Auswertung umgangen werden kann. Dabei handelt es sich offensichtlich um ein sicheres Vorwärts-Bitvektor-Problem und demnach gilt $0 \sqsubseteq 1$. Zu einem Flussgraphen erhält man dann eine Instanz der Analyse, indem jeder Kante, deren Anweisung eine Auswertung des Ausdrucks enthält und keine Zuweisung an eine der Komponenten des Ausdrucks, die Funktion GEN zugeordnet wird. Hier wird der Ausdruck ausgewertet und nicht wieder modifiziert, also ist der Ausdruck nach Durchlaufen der Kante verfügbar. Ein Kante, deren Anweisung einer Komponente des Ausdrucks einen neuen Wert zuweist, erhält die Funktion $KILL$ zugeordnet. Hier wird ein Teil des Ausdrucks verändert also sind alle bisherigen Auswertungen des Ausdrucks nun ungültig und er ist nicht mehr verfügbar. Alle anderen Kanten erhalten die Funktion ID zugeordnet, da sie keinen Einfluss auf die Verfügbarkeit des Ausdruckes haben. Am Anfang des Programms ist kein Ausdruck verfügbar, die initiale Information ist also 0.

Abbildung 2.3 zeigt die Zuordnung von Transferfunktionen zu den Kanten des schon vorher betrachteten Flussgraphen und das daraus resultierende Ungleichungssystem, sowie die Lösung, der Analyse für die Verfügbarkeit des Ausdrucks $x - 1$.



$$
\begin{aligned}
L[e_{main}] &\sqsubseteq 0 \\
L[n_1] &\sqsubseteq ID(L[e_{main}]) \\
L[n_2] &\sqsubseteq ID(L[e_{main}]) \\
L[n_3] &\sqsubseteq ID(L[n_2]) \\
L[n_3] &\sqsubseteq KILL(L[n_5]) \\
L[n_4] &\sqsubseteq GEN(L[n_3]) \\
L[n_5] &\sqsubseteq GEN(L[n_4]) \\
L[r_{main}] &\sqsubseteq GEN(L[n_3]) \\
L[r_{main}] &\sqsubseteq ID(L[n_1])
\end{aligned}
$$

$n \in N$	e_{main}	n_1	n_2	n_3	n_4	n_5	r_{main}
$L[n] \in \{0,1\}$	0	0	0	0	1	1	1

Abbildung 2.3: Zurordnung der Transferfunktionen und Formulierung, sowie Lösung, des Ungleichungssystems einer Analyse für die Verfügbarkeit des Ausdrucks $x - 1$

Betrachtet man dann die Funktion $\alpha : \mathcal{P}(Paths) \to \{0, 1\}$ mit

$$\alpha(M) = \bigoplus\{(f_{(n_{m-1},s_{m-1},n_m)} \circ \ldots \circ f_{(n_1,s_1,n_2)})(0) \mid [n_1, s_1, \ldots, n_m] \in M\}$$

für $M \in \mathcal{P}(Paths)$, dann ist α universell distributiv und damit eine Abstraktionsfunktion. Zudem kann man zeigen, dass gilt:

$$\begin{aligned} \alpha(M; [n_i, s_i, n_{i+1}]) &= f_{(n_i,s_i,n_{i+1})}(\alpha(M)) \text{ mit } f_{(n_i,s_i,n_{i+1})} \in \{KILL, ID, GEN\} \\ \alpha(\{[e_{main}]\}) &= 0 \end{aligned}$$

Damit sind alle Bedingungen des Transferlemmas erfüllt und man sieht, dass die größte Lösung des Ungleichungssystems für die Analyse der Verfügbarkeit des Ausdrucks $x - 1$ eine korrekte und präzise Approximation der Abstraktion der Lösung der erreichenden Pfade ist. Da wir wissen, dass die Analyse der erreichenden Pfade korrekt und präzise ist und damit alle erreichenden Pfade liefert, enthält die Abstraktion dieser Analyse genau die Information, die wir berechnen wollen. Für jeden erreichenden Pfad wird die Eigenschaft, ob der Ausdruck ausgewertet und danach nicht wieder verändert wird, betrachtet. Wir haben also gezeigt, dass die Lösung des Ungleichungssystems genau die von uns gewünschten Informationen berechnet.

Diese Techniken kann man nun auch für Programme mit Prozeduren und parallelen Kontrollflüssen erweitern. Dies führt in der Regel zu komplexeren Ungleichungssystemen, da weitere Eigenschaften wie zum Beispiel die korrekte Handhabung von Aufrufkontexten bei Prozeduren beachtet werden müssen. Es sollten nur Pfade betrachtet werden, die nach einem Prozeduraufruf die Ausführung am zum Aufrufpunkt gehörenden Rücksprungpunkt fortsetzen. Desweiteren sollten bei paralleler Ausführung alle Pfade betrachtet werden, die durch Interleaving der Ausführungen der einzelnen Threads entstehen können, um alle möglichen gegenseitigen Beeinflussungen der Threads zu berücksichtigen. Da für uns im Hinblick auf nachfolgende Kapitel nur die allgemeinen Informationen über Datenflussanalyse, sowie die Technik der abstrakten Interpretation interessant sind, sei für nähere Informationen in dieser Richtung auf die weiteren Erläuterungen in [NNH99, LMO07] hingewiesen.

Im nächsten Abschnitt betrachten wir nun einen anderen Ansatz zur Datenflussanalyse. Dabei wird von der Betrachtung eines Flussgraphen zur Betrachtung eines Push-Down Systems zur Modellierung des Programmablaufs übergegangen.

2.2 Gewichtete Push-Down Systeme

In [RSJM05] wird ein alternativer Ansatz zur klassischen Fixpunkt basierten Datenflussanalyse von Programmen mit Prozeduren untersucht. Dazu wird eine Verbindung von Datenflussanalyse zum Model-Checking betrachtet. Model-Checking beschäftigt sich damit einfachere Modelle von Programmen auf bestimmte Eigenschaften zu untersuchen. Die folgenden Darstellungen basieren auf den Überlegungen in [RSJM05]. Beweise der angesprochenen Sätze können dort, in teilweise anderer Notation oder Form, nachgelesen werden.

Als grundlegendes Modell für die Darstellung von Programmen mit Prozeduren wird auf Push-Down Systeme (PDS) zurückgegriffen. Dabei wird folgende Definitionen eines PDS benutzt:

Definition 2.7. Ein Push-Down System (PDS) ist ein Tripel $\mathcal{P} = (P, \Gamma, \Delta)$. Dabei ist P eine endliche Menge von Kontrollzuständen und Γ eine endliche Menge von Stacksymbolen. Es gilt $P \cap \Gamma = \emptyset$. Δ ist eine endliche Menge mit Transitionsregeln der Form:

1. $r = p\gamma \hookrightarrow \tilde{p}$ mit $p, \tilde{p} \in P$, $\gamma \in \Gamma$
2. $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}$ mit $p, \tilde{p} \in P$, $\gamma, \tilde{\gamma} \in \Gamma$
3. $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2$ mit $p, \tilde{p} \in P$, $\gamma, \tilde{\gamma}_1, \tilde{\gamma}_2 \in \Gamma$

Eine *Konfiguration* ist ein Wort aus der Menge $P\Gamma^*$. $Conf_{\mathcal{P}}$ sei die Menge aller Konfigurationen.

Ein *Pfad* ist eine Folge

$$\rho = [r_1, \ldots, r_n] \text{ mit } r_i \in \Delta, i \in \{1, \ldots, n\}, n \in \mathbb{N}_0$$

von Transitionsregeln. [] sei der leere Pfad und $Paths_{\mathcal{P}}$ sei die Menge aller Pfade.

Wir definieren einen Konkatenationsoperator ; , der zu zwei beliebigen Pfaden $\rho_1, \rho_2 \in Paths_{\mathcal{P}}$ einen zusammengesetzten Pfad

$$\rho_1; \rho_2 = \begin{cases} \rho_1 & \text{für } \rho_2 = [] \\ \rho_2 & \text{für } \rho_1 = [] \\ [r_1, \ldots, r_n, r_{n+1}, \ldots, r_m] & \text{für } \rho_1 = [r_1, \ldots, r_n], \rho_2 = [r_{n+1}, \ldots, r_m] \end{cases}$$

liefert. Für Mengen $M_1, M_2 \subseteq Paths_{\mathcal{P}}$ von Pfaden gelte $M_1; M_2 = \{\rho_1; \rho_2 \mid \rho_1 \in M_1, \rho_2 \in M_2\}$.

Für zwei Konfigurationen $c, \bar{c} \in Conf_{\mathcal{P}}$ und einen Pfad $\rho \in Paths_{\mathcal{P}}$ gelte dann die Übergangsrelation $c \xrightarrow{\rho}_{\mathcal{P}} \bar{c}$:

1. für $\rho = []$, wenn $\bar{c} = c$.
2. für $\rho = [r]; \tilde{\rho}$ mit $r = p\gamma \hookrightarrow \tilde{p}\tilde{w} \in \Delta$, $\tilde{\rho} \in Paths_{\mathcal{P}}$, wenn $w \in \Gamma^*$ mit $c = p\gamma w$ und $\tilde{p}\tilde{w}w \xrightarrow{\tilde{\rho}}_{\mathcal{P}} \bar{c}$ existiert.

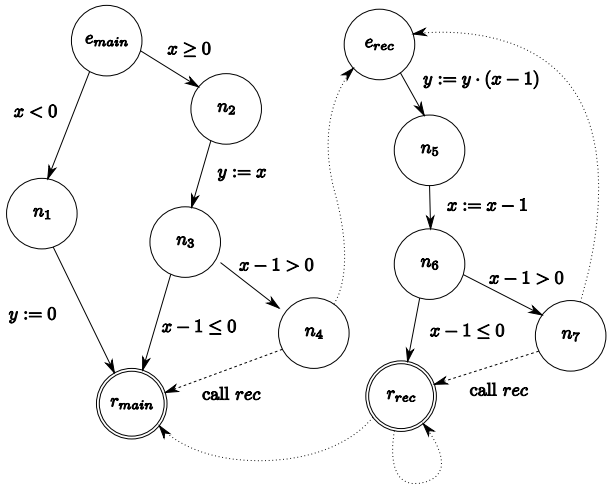
Wenn aus dem Kontext klar ist, auf welches PDS $\mathcal{P}$ Bezug genommen wird, wird im folgenden auf die explizite Nennung im Index verzichtet.

Ein PDS kann damit auf natürliche Art und Weise die Ausführung eines Programms mit Prozeduren modellieren. Dabei dient der Stack dazu die Rücksprungadressen der aufgerufenen Prozeduren zu speichern und somit eine Beachtung des Aufrufkontextes zu gewährleisten. Insbesondere kann man die im vorherigen Abschnitt 2.1 durch interprozedurale Flussgraphen beschriebenen Programme untersuchen.

Definition 2.8. Gegeben sei ein System von Flussgraphen $\mathcal{G} = (Proc, (G_\pi)_{\pi \in Proc})$, in dem keine Kante mit einem Threadaufruf beschriftet ist. Dann sei das zugehörige PDS definiert durch:

- $P = \{p\}$
- $\Gamma = N_\mathcal{G}$
- Δ enthält für jede Kante $e = (n, s, \tilde{n}) \in E_\mathcal{G}$ in einem der Flussgraphen eine der folgenden Transitionsregeln:
  - $pn \hookrightarrow p\tilde{n} \in \Delta$, falls $s \in BA$
  - $pn \hookrightarrow pe_\pi \tilde{n} \in \Delta$, falls $s = \text{call } \pi \in CA$ für ein $\pi \in Proc$
- Δ enthält für jede Prozedur $\pi \in Proc$ eine Transitionsregel $pr_\pi \hookrightarrow p \in \Delta$

Man sieht sofort, dass sich PDS besonders gut eignen um kontextsensitive Analysen für Systeme Flussgraphen mit Prozeduraufrufen zu formulieren. Bei einem Prozeduraufruf wird der Anfangspunkt der aufgerufenen Prozedur oben auf den Stack gelegt und der Knoten, an dem die Ausführung der aufrufenden Prozedur unterbrochen wurde, darunter gespeichert. Wenn, durch einfaches entfernen des Endpunktes einer Prozedur vom Stack, eine aufgerufene Prozedur verlassen wird, steht automatisch der Punkt an dem die Ausführung unterbrochen wurde wieder an oberster Stelle und das Programm wird an vorgesehener Stelle fortgesetzt. Das PDS kann also genau die Pfade durchlaufen, die den im System von Flussgraphen zu untersuchenden Pfaden entsprechen. Abbildung 2.4 zeigt die Umwandlung eines interprozeduralen Flussgraphen $\mathcal{G}$ in ein PDS $\mathcal{P}$.



$$
\begin{array}{lcl}
r_1 & : & pe_{main} \hookrightarrow pn_2 \\
r_2 & : & pe_{main} \hookrightarrow pn_1 \\
r_3 & : & pn_1 \hookrightarrow pr_{main} \\
r_4 & : & pn_2 \hookrightarrow pn_3 \\
r_5 & : & pn_3 \hookrightarrow pn_4 \\
r_6 & : & pn_3 \hookrightarrow pr_{main} \\
r_7 & : & pn_4 \hookrightarrow pe_{rec} r_{main} \\
r_8 & : & pe_{rec} \hookrightarrow pn_5 \\
r_9 & : & pn_5 \hookrightarrow pn_6 \\
r_{10} & : & pn_6 \hookrightarrow pn_7 \\
r_{11} & : & pn_6 \hookrightarrow pr_{rec} \\
r_{12} & : & pn_7 \hookrightarrow pe_{rec} r_{rec} \\
r_{13} & : & pr_{rec} \hookrightarrow p \\
r_{14} & : & pr_{main} \hookrightarrow p
\end{array}
$$

Abbildung 2.4: Umwandlung eines interprozeduralen Flussgraphen $\mathcal{G}$ in ein PDS $\mathcal{P}$

Im folgenden sei also $\mathcal{P} = (P, \Gamma, \Delta)$ ein PDS. Die in [RSJM05] eingeführte Neuerung ist nun, das Datenflussinformationen nicht mehr nur für einzelne Programmpunkte berechnet

werden. Stattdessen wird für eine beliebige Konfiguration $c \in Conf$ die Menge der Pfade des PDS betrachtet, die diese Konfiguration so transformieren, dass die abschließende Konfiguration in einer regulären Menge $C \subseteq Conf$ liegt. Um aus der Untersuchung der Pfade Informationen gewinnen zu können, werden die einzelnen Transitionen des PDS mit Gewichten versehen. Diese stellen im konkreten Fall die untersuchten Datenflussinformationen dar, die bei der Anwendung der Transition beachtet werden müssen. Mit diesen Gewichten werden dann die Datenflussinformationen an den Enden aller untersuchten Pfade berechnet und durch einen Wert approximiert. Dazu definiert man für die Gewichte:

Definition 2.9. Ein *beschränkter idempotenter Halbring* ist ein Quintupel $\mathcal{S} = (D, \oplus, \odot, 0, 1)$. Dabei ist D eine Menge mit $0, 1 \in D$ und $\oplus, \odot$ sind binäre Operatoren auf D mit:

1. $(D, \oplus)$ ist ein kommutativer Monoid mit 0 als neutralem Element und $\oplus$ ist idempotent.
2. $(D, \odot)$ ist ein Monoid mit 1 als neutralem Element.
3. $\odot$ distributiert über $\oplus$.
4. Für alle $d \in D$ gilt $0 \odot d = 0 = d \odot 0$.
5. $(D, \oplus)$ ist ein vollständiger Verband, in dem jede absteigende Kette schließlich stabil wird.

In [RSJM05] wird lediglich gefordert, dass $(D, \oplus)$ eine partielle Ordnung mit den geforderten Eigenschaften induziert. Aus der Existenz des Nullelementes 0 als größtem Element und der Eigenschaft, dass jede absteigende Kette schließlich stabil wird, folgt jedoch schon, dass $(D, \oplus)$ ein vollständiger Verband ist. Damit existiert für alle $X \subseteq D$ auch $\bigoplus X \in D$, als Erweiterung des paarweisen Operators $\oplus$.

Die Wahl eines idempotenten Halbringes für die Darstellung von Gewichten begründet sich dabei mit den benötigten Operationen. Zu einer Konfiguration werden alle Pfade, die diese in der oben beschriebenen Form transformieren betrachtet. Um nun eine Aussage über das Gewicht aller dieser Pfade machen zu können, braucht man einen Operator $\oplus$, der die Gewichte von zwei Pfade kombiniert und korrekt und präzise durch ein einzelnes Gewicht approximiert. Die Korrektheit und Präzision der Approximation im Sinne der Datenflussanalyse folgt dabei direkt aus der Definition der Ordnung auf den Gewichten. Für die spätere Berechnung benötigt man, zur Initialisierung, die Existenz eines größten Elements. Es repräsentiert den Datenflussfakt, der keine Information beschreibt. Demzufolge ist jede andere Information eine korrekte Approximation. Desweiteren betrachtet man Pfade als Folgen von Transitionen, die jeweils mit einem Gewicht versehen sind. Um nun am Ende des Pfades eine Aussage über das Gewicht des ganzen Pfades machen zu können benötigt man einen Operator $\odot$, der die Gewichte von zwei nacheinander ausgeführten Transitionen sequentiell konkateniert und zu einem Gewicht zusammenfügt. Dabei ist es nun sinnvoll die Konkatenation als strikt im größten Element zu fordern, da dieses für die leere Information steht und die Konkatenation von leerer Information mit einem beliebigen Datenflussfakt die leere Information ergeben sollte. Die Forderung der Distributivität kann

man damit begründen, dass man Informationen von Pfaden die sich überlappen möglichst früh kombinieren möchte. So erlaubt diese Eigenschaft das Gewicht des überlappenden Teilpfades nur einmal zu berechnen und dann mit den zwei Restpfaden zu verknüpfen.

Mit dieser Definition der Gewichte definiert man dann ein gewichtetes Push-Down System (WPDS) als ein PDS in dem jeder Transition ein Gewicht eines Halbringes zugewiesen wird.

Definition 2.10. Ein gewichtetes Push-Down System (WPDS) ist ein Tripel $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$. Dabei ist $\mathcal{P} = (P, \Gamma, \Delta)$ ein Push-Down System, $\mathcal{S} = (D, \oplus, \odot, 0, 1)$ ein beschränkter idempotenter Halbring und $f : \Delta \to D$ eine Gewichtungsfunktion, die jeder Regel in Δ ein Gewicht in D zuweist.

Wie oben schon beschrieben, betrachtet man nun zu einer beliebigen Konfiguration die Gewichte der Pfade die diese in eine Konfiguration einer bestimmten Menge transformieren. Diese bezeichnet man mit erreichenden Pfaden, da sie die gegebene Menge erreichen. Man beschränkt sich dabei auf die Betrachtung von regulären Mengen von Zielkonfigurationen. Mit den Definitionen für die Gewichte in Form eines Halbringes und der Definition für WPDS lässt sich dieses Problem dann folgendermaßen formulieren:

Definition 2.11. Sei $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ ein gewichtetes Push-Down System und $C \subseteq Conf$ eine reguläre Menge von Konfigurationen.

- Die Menge der erreichenden Pfade für eine Konfiguration $c \in Conf$ ist dann

$$Paths(c, C) = \{\rho \in Paths \mid c \xrightarrow{\rho} \bar{c} \text{ für ein } \bar{c} \in C\}.$$

- Das "generalized pushdown predecessor problem" besteht dann darin, den Wert

$$\delta(c) = \bigoplus \{f(r_1) \odot \ldots \odot f(r_n) \mid [r_1, \ldots, r_n] \in Paths(c, C)\}$$

  zu berechnen.

In den folgenden Betrachtungen wird davon ausgegangen, dass die reguläre Menge der Zielkonfigurationen $C \subseteq Conf$ durch einen bestimmten Automatentyp beschrieben wird. Dieser lässt sich jedoch für jede reguläre Menge von Konfigurationen konstruieren. Wir betrachten hier eine leicht abgewandelte Definition des in [RSJM05] betrachteten Automaten. Die Beobachtungen lassen sich jedoch leicht übertragen. Die gewählte Darstellung ist an die später für DPN untersuchten Automaten angelehnt.

Definition 2.12. Ein *$\mathcal{P}^*$-Automat* $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$ ist ein spezieller endlicher Automat zur Beschreibung einer regulären Menge von Konfigurationen eines PDS $\mathcal{P} = (P, \Gamma, \Delta)$. Dabei erfüllt der Automat die folgenden Bedingungen:

1. $\Sigma = P \cup \Gamma$

2. $S = \{s^s\} \cup S_s$ mit S_s als eine endliche Menge von Zuständen mit $P \subseteq S_s$ und $s^s \notin S_s$
3. $\bar{\delta} = \bar{\delta}_{state} \cup \bar{\delta}_{stack}$ wobei
   - $\bar{\delta}_{state} = \{(s^s, p, p) \mid p \in P\}$
   - $\bar{\delta}_{stack} \subseteq (P \times \Gamma \times P) \cup (S_s \times \Gamma \times (S_s \setminus P))$
4. $F \subseteq S_s$

Ein $\mathcal{P}$-*Automat* $\mathcal{A}$ ist ein $\mathcal{P}^*$-Automat, für dessen Transitionsmenge, in diesem Fall in der Regel mit δ bezeichnet, sogar $\delta_{stack} \subseteq S_s \times \Gamma \times (S_s \setminus P)$ gilt.

Man kann sich einen $\mathcal{P}$-Automaten als eine Sammlung von Teilautomaten vorstellen, die in der Lage sind Wörter aus Stacksymbolen zu erkennen. Jeder dieser Teilautomaten hat einen eigenen Anfangszustand, der mit einem Kontrollzustand des PDS bezeichnet ist. Man kann also für jeden Kontrollzustand des PDS eine eigene Menge von erlaubten Stackwörter spezifizieren. Die restlichen Zustände, zur Erkennung des Stackinhalts, können dabei von den Teilautomaten gemeinsam genutzt werden. Die Konstruktion eines solchen Automaten ist für jede reguläre Menge von Konfigurationen möglich. In unserer Formulierung eines $\mathcal{P}$-Automaten wird durch das Einlesen des Kontrollzustandes einer Konfiguration automatisch der Startzustand des zugehörigen Teilautomaten ausgewählt. Von diesem ausgehend kann dann der Stackinhalt erkannt werden. Die ursprüngliche Definition eines $\mathcal{P}$-Automaten enthielt diesen, durch den zusätzlichen Startzustand und die Kanten in δ_{state} eingeführten, Mechanismus nicht. Der zum Kontrollzustand gehörende Startzustand musste vorher ausgewählt werden, und dann wurde nur der Stackinhalt eingelesen. Die folgenden Aussagen beziehen sich daher in der Regel auf die Transitionen in δ_{stack}, welche aus der ursprünglichen Definition übernommen wurden. Abbildung 2.5 zeigt ein Beispiel für einen $\mathcal{P}$-Automaten zur Beschreibung einer Menge von Konfigurationen des in Abbildung 2.4, aus dem Flussgraphen $\mathcal{G}$, konstruierten PDS $\mathcal{P}$.

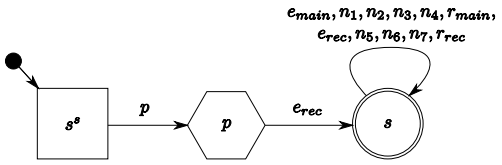


Abbildung 2.5: $\mathcal{P}$-Automat für die Menge der Konfigurationen $pe_{rec}(N_{\mathcal{G}})^*$ des in Abbildung 2.4 dargestellten PDS $\mathcal{P}$ zum Flussgraphen $\mathcal{G}$

Im folgenden wird nun eine reguläre Menge von Konfiguration $C \subseteq Conf$ eines WPDS $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, mit PDS $\mathcal{P} = (P, \Gamma, \Delta)$ und Halbring $\mathcal{S} = (D, \oplus, \odot, 0, 1)$, beschrieben durch den $\mathcal{P}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$ betrachtet.

Man betrachtet nun nicht direkt die Menge der erreichenden Pfade des PDS, sondern beschreitet einen Umweg. Dazu wird ein neues PDS $\mathcal{PA} = (S, \Gamma, \tilde{\Delta})$ definiert, dass das ursprüngliche PDS $\mathcal{P}$ erweitert um zusätzlich Teile des Automaten $\mathcal{A}$ zu simulieren. Dabei sei $\tilde{\Delta} = \Delta \cup \{s\gamma \hookrightarrow \tilde{s} \mid (s, \gamma, \tilde{s}) \in \delta_{stack}\}$. Nach Definition gilt $P \subseteq S$. $\mathcal{PA}$ kann also

sowohl Transitionen des ursprünglichen PDS ausführen, als auch das Akzeptieren eines Wortes durch den Automaten $\mathcal{A}$ simulieren, indem der vorhandenen Stack geleert wird und ein akzeptierender Kontrollzustand erreicht wird. Man definiert, dass eine Konfiguration $c \in Conf_{\mathcal{P}}$ von $\mathcal{PA}$ akzeptiert wird, wenn ein Pfad $\rho \in Paths_{\mathcal{PA}}$ und ein Zustand $s \in F$ existiert mit $c \xrightarrow{\rho}_{\mathcal{PA}} s$. Eine genauere Untersuchung ergibt dann, dass ein solcher Pfad ρ aus zwei Teilen besteht. Im ersten Teil werden nur Transition des ursprünglichen PDS $\mathcal{P}$ angewendet und der Kontrollzustand verbleibt in P. Dieser Teil des Pfades entspricht also einer Transformation der Konfiguration c durch das PDS in eine andere Konfiguration $\bar{c} \in Conf_{\mathcal{P}}$. Wird nun jedoch einmal eine Transition ausgeführt, die aus dem Automaten $\mathcal{A}$ hinzugefügt wurde, so können im folgenden nur weitere Transitionen dieses Typs ausgeführt werden. Dies liegt darin begründet, dass alle Transitionen des Automaten in einem Zustand enden, der kein Kontrollzustand des ursprünglichen PDS ist. Diese zweite Phase, in der durch Anwendung der Transitionen des Automaten nacheinander alle Symbole des Stacks gelöscht werden, um mit dem leeren Stack und in einem akzeptierenden Zustand zu enden, entspricht dem akzeptieren der Konfiguration $\bar{c}$ durch den Automaten $\mathcal{A}$. Demnach gilt $\bar{c} \in C$.

Das PDS $\mathcal{PA}$ akzeptiert also Konfigurationen $c \in Conf_{\mathcal{P}}$, die einen *Nachfolger* in der Menge C haben, durch leeren des Stacks und Übergang in einen akzeptierenden Zustand. Dabei repräsentiert der erste Teil des Pfades, der dabei durchlaufen wird, genau einen der Pfade aus $Paths(c, C)$. Der gesamte betrachtete Pfad leert den kompletten Stack. Da immer nur das oberste Stacksymbol bearbeitet werden kann, bedingt das Leeren des Stacks, dass der gesamte Pfad in mehrere Teile zerlegt werden kann, die jeweils nacheinander ein Symbol vom Stack entfernen. Demnach lässt sich auch der für die Betrachtung interessante Teil in diese Teile zerlegen. Im folgenden werden diese Teile nun näher betrachtet.

Definition 2.13. Eine *Popsequenz* für γ vom Zustand s nach $\tilde{s}$ für das PDS $\mathcal{PA}$ ist ein Pfad ρ, der in der Konfiguration $s\gamma$ startend in der Konfiguration $\tilde{s}$ endet. Es gilt also $s\gamma \xrightarrow{\rho} \tilde{s}$.

Auf Grund der Eigenschaft der Übergangsrelation der PDS gilt dann auch $s\gamma w \xrightarrow{\rho} \tilde{s}w$ für eine Popsequenz ρ für γ von s nach $\tilde{s}$ für alle $w \in \Gamma^*$. Durch Ausführung einer Popsequenz für jedes Stacksymbol einer Konfiguration kann also der Stack geleert werden. Die Popsequenzen sind also die Bausteine, aus denen sich die für die Analyse interessanten Pfade zusammensetzen lassen. Im folgenden versucht man die Mengen der Popsequenzen zu charakterisieren. Dazu wird folgende Grammatik definiert:

Definition 2.14. Definiere die folgende kontextfreie Grammatik mit den nichtterminalen Symbolen $PopSeq_{(s,\gamma,\tilde{s})}$ für $s, \tilde{s} \in S$ und $\gamma \in \Gamma$, sowie dem einzigen Terminalsymbol ϵ:

(INIT) $PopSeq_{(s,\gamma,\tilde{s})} \to \epsilon$ wenn $(s, \gamma, \tilde{s}) \in \delta_{stack}$

(RETURN) $PopSeq_{(p,\gamma,\tilde{p})} \to \epsilon$ wenn $r = p\gamma \hookrightarrow \tilde{p} \in \Delta$

(STEP) $PopSeq_{(p,\gamma,s)} \to PopSeq_{(\tilde{p},\tilde{\gamma},s)}$ wenn $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma} \in \Delta$, $s \in S$

(CALL) $PopSeq_{(p,\gamma,s)} \to PopSeq_{(\tilde{p},\tilde{\gamma}_1,s)} PopSeq_{(s,\tilde{\gamma}_2,\tilde{s})}$ wenn $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \in \Delta$, $s, \tilde{s} \in S$

Betrachtet man die einzelnen Regeln der Grammatik genauer, erkennt man, dass zu einem Nichtterminal $PopSeq_{(s,\gamma,\tilde{s})}$ genau dann ein vollständiger Ableitungsbaum T existiert, wenn es eine Popsequenz im PDS $\mathcal{PA}$ für γ von s nach $\tilde{s}$ gibt. Alle Blätter eines vollständigen Ableitungsbaums sind im Fall dieser Grammatik dabei mit ϵ beschriftet. So existiert im PDS $\mathcal{PA}$ eine Regel $s\gamma \hookrightarrow \tilde{s}$ für jede Transition $(s, \gamma, \tilde{s}) \in \delta_{stack}$ und damit eine Popsequenz für γ von s nach $\tilde{s}$. Nach den (INIT) Regeln hat aber auch das Symbol $PopSeq_{(s,\gamma,\tilde{s})}$ einen vollständigen Ableitungsbaum, da man es direkt zu ϵ ableiten kann. Die (RETURN) Regeln garantieren genauso, dass jedes Symbol $PopSeq_{(p,\gamma,\tilde{p})}$, dessen Popsequenz direkt durch eine Transition des PDS, die das Symbol entfernt, beschrieben werden kann, einen vollständigen Ableitungsbaum hat. Die (STEP) Regeln decken den Fall ab, dass man, wenn eine Popsequenz für $\tilde{\gamma}$ von $\tilde{p}$ nach s existiert und eine Transition des PDS den Stack $p\gamma$ nach $\tilde{p}\tilde{\gamma}$ transformieren kann, durch Kombination der Transition mit der vorhandenen Popsequenz eine Popsequenz für γ von p nach s erhält. Die Regeln garantieren in diesem Fall die Existenz eines vollständigen Ableitungsbaums, wenn für die benötigte folgende Popsequenz ein Ableitungsbaum existiert. Die (CALL) Regeln betrachten den Fall, dass man zwei Popsequenzen für $\tilde{\gamma}_1$ von $\tilde{p}$ nach s und für $\tilde{\gamma}_2$ von s nach $\tilde{s}$ hat. Anschaulich hat man also zwei Pfade die hintereinander ausgeführt die Konfiguration $\tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2$ in die Konfiguration $\tilde{s}$ überführen würden. Wenn man nun durch eine Transition des PDS die Konfiguration $p\gamma$ in die Konfiguration $\tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2$ überführen kann, erhält man durch Kombination dieser Transition mit den Popsequenzen eine Popsequenz für γ von p nach $\tilde{s}$. Die (CALL) Regeln garantieren in diesem Fall die Existenz eines vollständigen Ableitungsbaums. Man sieht, dass die vollständigen Ableitungsbäume sogar die einzelnen Transitionen einer Popsequenz enthalten. Jeder innere Knoten eines Ableitungsbaums, dieser entspricht einem nichtterminalen Symbol, kann mit der Produktionsregel beschriftet werden, mit deren Hilfe er in diesem konkreten Baum weiter abgeleitet wurde. Jede Regel der Grammatik steht jedoch in direkter Verbindung mit einer Regel des PDS $\mathcal{PA}$. Betrachtet man die Grammatik zum PDS $\mathcal{PA}$, kann man den folgenden Satz beweisen:

Satz 2.15. *Ein nichtterminales Symbol $PopSeq_{(s,\gamma,\tilde{s})}$ hat genau dann einen vollständigen Ableitungsbaum, wenn das PDS $\mathcal{PA}$ eine Popsequenz für γ vom Zustand s nach $\tilde{s}$ hat. Jeder vollständige Ableitungsbaum liefert durch eine "preorder" Auflistung der zu den angewendeten Produktionsregeln gehörenden Transitionsregeln des PDS $\mathcal{PA}$ einen Pfad für die Popsequenz.*

Man kann also eine Popsequenz für ein Stacksymbol von einem Zustand in einen anderen konstruieren, falls eine solche existiert, indem man einen beliebigen vollständigen Ableitungsbaum des zugehörigen nichtterminalen Symbols der Grammatik konstruiert und den zugehörigen Pfad betrachtet. Umgekehrt existiert für jede Popsequenz ein zugehöriger vollständiger Ableitungsbaum. Mit Hilfe des nichtterminalen Symbols $PopSeq_{(s,\gamma,\tilde{s})}$ kann man also die komplette Klasse aller Popsequenzen für γ von s nach $\tilde{s}$ konstruieren. Mit Hilfe dieser Konstruktion kann man einen akzeptierenden Pfad ρ im PDS $\mathcal{PA}$ für eine beliebige Konfiguration $p\gamma_1 \dots \gamma_n$ zusammensetzen. Dazu wählt man für jedes der Stacksymbole γ_i der untersuchten Konfiguration eine passende Klasse von Popsequenzen aus. Der Startzustand der ersten Klasse wird dabei durch den Kontrollzustand der Konfiguration festgelegt.

Die Anfangs- und Endzustände der folgenden Klassen müssen nur zueinander passen und der Endzustand der letzten Klasse einem Endzustand des Automaten entsprechen. Kann man nun für jede der Klassen, wie oben beschrieben, eine Popsequenz konstruieren, gilt für die Verknüpfung dieser, dass sie den kompletten Stack leeren und in einem akzeptierenden Zustand enden. Betrachtet man in diesem Pfad nun nur die Transitionen des ursprünglichen PDS $\mathcal{P}$, so erhält man, wie oben schon erklärt, einen der erreichenden Pfade. Da man jeden erreichenden Pfad in einen akzeptierenden Pfad von $\mathcal{PA}$ einbetten kann, die erreichte Konfiguration liegt in der Sprache des Automaten, und jeden dieser in Popsequenzen aufspalten kann, kann man diese Konstruktion umgekehrt auch für jeden erreichenden Pfad durchführen. Da man also mit Hilfe der Grammatik der Popsequenzen alle möglichen erreichenden Pfade beschreiben kann, kann man nun Aussage über diese mit Hilfe der Grammatik formulieren. Abbildung 2.6 zeigt beispielhaft zwei Ableitungsbäume des nichtterminalen Symbols $PopSeq_{(p,e_{main},s)}$ für die Grammatik des PDS $\mathcal{PA}$, das sich aus dem in Abbildung 2.4 betrachtete PDS $\mathcal{P}$ und dem in Abbildung 2.5 dargestellten Automaten $\mathcal{A}$ ergibt. Abgebildet sind auch die Teilpfade des ursprünglichen PDS. Da $s \in F$ sind die abgeleiteten Popsequenzen sogar akzeptierende Pfade des erweiterten PDS $\mathcal{PA}$ und die dargestellten Teilpfade damit erreichende Pfade des ursprünglichen PDS.

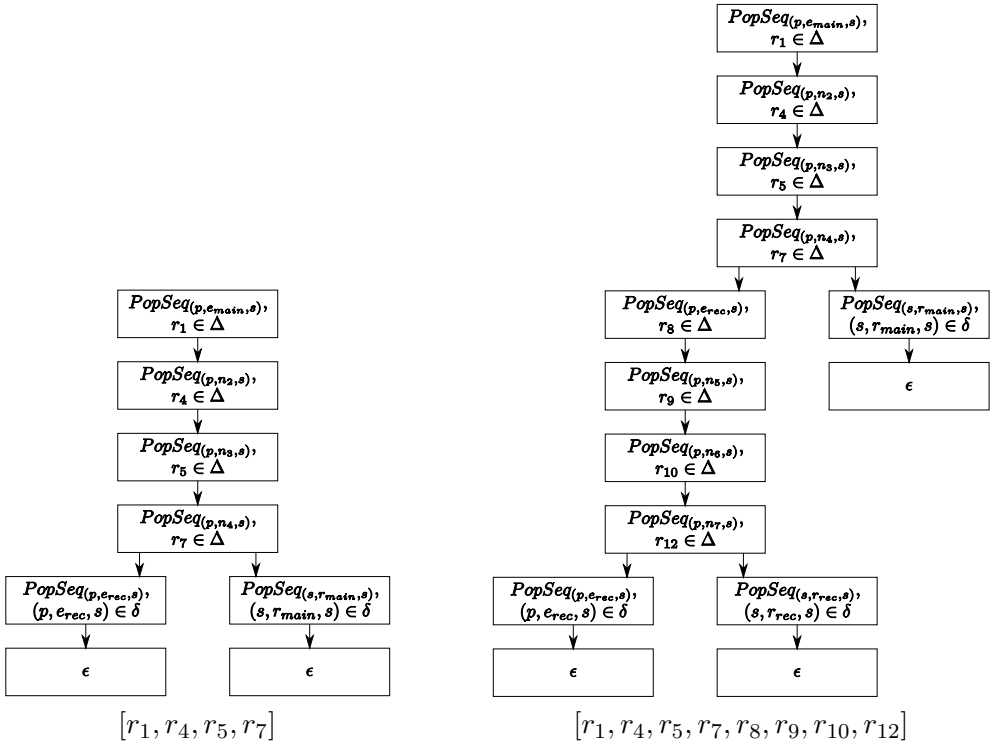


Abbildung 2.6: Beispiele für Ableitungsbäume des Symbols $PopSeq_{(p,e_{main},s)}$ aus der Grammatik zum PDS $\mathcal{PA}$ und die Teilpfade des ursprünglichen PDS, der zu den Ableitungsbäumen gehörigen Popsequenzen für e_{main} von p nach s

Um nun aus der Grammatik Informationen über die gewichteten Pfade ableiten zu können, wird jeder Produktionsregel der Grammatik ein Gewicht zugeordnet. Die Produktionsregeln vom Typ (INIT), die eine Popsequenz durch eine Kante des Automaten darstellen, erhalten dabei das Gewicht 1, da hier keine Arbeit durch das ursprüngliche PDS verrichtet wird. Alle anderen Regeln erhalten das Gewicht $f(r)$ der ihnen zugeordneten Transitionsregel des PDS. Nun wird statt einer "preorder" Auflistung, der zu den angewendeten Produktionsregeln gehörenden Transitionsregeln eines Ableitungsbaum, eine Auflistung der zu den Produktionsregeln gehörenden Gewichte betrachtet. Dabei werden die einzelnen Gewichte durch den $\odot$ Operator sequentiell miteinander verknüpft. Somit erhält man für jeden Ableitungsbaum ein Gewicht, dass dem Gewicht der in der Popsequenz ausgeführten Transitionen des ursprünglichen PDS entspricht. Nun werden die Gewichte aller Ableitungsbäume zu einem nichtterminalen Symbol mit Hilfe des $\bigoplus$ Operators kombiniert und man erhält für jedes nichtterminale Symbol der Grammatik ein Gewicht, dass die Gewichte aller Ableitungsbäume, und damit das Gewicht der Klasse der Popsequenzen die durch dieses Symbol beschrieben wird, korrekt und präzise beschreibt. Die dabei zur konkreten Berechnung angewandte Technik nennt sich abstrakte Grammatik und erlaubt eine Berechnung der Gewichte direkt anhand der Produktionsregeln der Grammatik ohne explizit die Ableitungsbäume zu betrachten. Das resultierende System von Gewichten, das jedem nichtterminalen Symbol ein Gewicht zuordnet, wird dabei als Fixpunkt eines Gleichungssystems berechnet, dass durch die Regeln der Grammatik festgelegt wird. Da wir für unsere späteren Betrachtungen auf die Technik der abstrakten Interpretation von Ungleichungssystemen zurückgreifen sei dies hier nur kurz erwähnt und für nähere Informationen auf [RSJM05] verwiesen.

Aus den Gewichten der Klassen von Popsequenzen kann man analog zur oben schon beschriebenen Konstruktion der erreichenden Pfade, und dank der Distributivität der Gewichte, auch das Gewicht über alle erreichenden Pfade zusammensetzen, indem man für die Stacksymbole der untersuchten Konfiguration alle möglichen Folgen von Klassen von Popsequenzen betrachtet deren Anfangs- und Endzustände zueinander passen und deren Endzustand der gesamten Sequenz einem Endzustand des Automaten entspricht. Die Gewichte der einzelnen Klassen werden mit dem $\odot$ Operator verknüpft und die Gewichte der einzelnen Folgen mit $\bigoplus$ kombiniert. Insgesamt erhält man das Gewicht aller erreichenden Pfade und damit den gesuchten Wert $\delta(p\gamma_1, \ldots, \gamma_n)$.

Man sieht jedoch, dass die oben beschriebene Grammatik schnell sehr groß werden kann, da bei vielen Zuständen und Stacksymbolen die Anzahl der zu betrachtenden Symbole sehr groß wird. Dabei sind viele dieser Symbole vielleicht gar nicht interessant, da die zugehörige Klasse von Popsequenzen leer ist.

Außerdem wird bei Anfragen mit verschiedenen Konfigurationen c, aber gleichem C wiederholt die gleiche Grammatik betrachtet werden. Es ist daher sinnvoll eine Möglichkeit zu finden die für die Klassen von Popsequenzen berechneten Gewichte in einer Form zu speichern, die deren Benutzung für verschiedene c erlaubt.

Im folgenden bedient man sich daher einer im Bereich des Model-Checking betrachteten Technik, die uns eine Verkleinerung der benötigten Grammatik, sowie eine geeignete Darstel-

lungsform, die eine Benutzung der bereits berechneten Gewichte für verschiedene Anfragen ermöglicht, liefert.

Wie schon erwähnt akzeptiert das PDS $\mathcal{PA}$ eine Konfigurationen c durch leeren des Stacks und Übergang in einen akzeptierenden Zustand des Automaten. Wie oben beschrieben lässt sich der dabei durchlaufene Pfad ρ in zwei Teile ρ_1 und ρ_2 aufspalten. Der erste Teil ρ_1 besteht nur aus Transitionen des ursprünglichen PDS $\mathcal{P}$ und bewirkt daher eine Transformation der Konfiguration c in eine Konfiguration $\bar{c}$. Es gilt $c \xrightarrow{\rho_1}_{\mathcal{P}} \bar{c}$. Der zweite Teil ρ_2 besteht nur aus Transitionen des Automaten $\mathcal{A}$ und simuliert damit ein Akzeptieren der Konfiguration $\bar{c}$ durch den Automaten. Damit gilt also $\bar{c} \in C$. Das PDS $\mathcal{PA}$ akzeptiert also alle Konfigurationen c, die einen Nachfolger in der Menge C haben. Im Model-Checking wird diese Menge mit $PRE^*(C)$ bezeichnet.

Definition 2.16. Zu einer Menge $C \subseteq Conf$ von Konfigurationen eines PDS $\mathcal{P}$ sei

$$PRE^*(C) = \{c \in Conf \mid c \xrightarrow{\rho} \bar{c} \text{ für ein } \bar{c} \in C \text{ und } \rho \in Paths\}$$

die Menge der vorhergehenden Konfigurationen.

Es existieren Automaten-basierte Techniken [BEM97] um diese Menge direkt zu einer gegebenen regulären Menge C zu berechnen. Insbesondere ist die konstruierte Menge dabei wieder regulär. Dies kann zum Beispiel dazu benutzt werden um die Erreichbarkeit von Fehlerzuständen zu testen. Auch kann man, wie in Kapitel 1 schon erwähnt, gewisse Datenflussanalysen direkt auf diese Konstruktion abbilden [EK99, EP00]. Wir betrachten nun das Verfahren zur Berechnung von $PRE^*(C)$ etwas näher:

Definition 2.17. Zu einer regulären Menge $C \subseteq Conf$ von Konfigurationen eines PDS $\mathcal{P}$ gegeben durch einen $\mathcal{P}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, P, F)$ definieren wir einen Saturierungsalgorithmus zur Berechnung des Automaten $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, P, F)$. Dabei ist $\bar{\delta} = \bar{\delta}_{state} \cup \bar{\delta}_{stack}$ mit $\delta_{state} = \bar{\delta}_{state}$ und $\bar{\delta}_{stack}$ ist die kleinste Menge, die die folgenden Bedingungen erfüllt:

(INIT) $\bar{\delta}_{stack} \supseteq \delta_{stack}$

(RETURN) $r = p\gamma \hookrightarrow \tilde{p} \in \Delta$, dann $t = (p, \gamma, \tilde{p}) \in \bar{\delta}_{stack}$

(STEP) $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma} \in \Delta$ und $\tilde{t} = (\tilde{p}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}$ für $\tilde{s} \in S$, dann $t = (p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$

(CALL) $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \in \Delta$ und $\tilde{t}_1 = (\tilde{p}, \tilde{\gamma}_1, \tilde{s}_1), \tilde{t}_2 = (\tilde{s}_1, \tilde{\gamma}_2, \tilde{s}_2) \in \bar{\delta}_{stack}$ für $\tilde{s}_1, \tilde{s}_2 \in S$, dann $t = (p, \gamma, \tilde{s}_2) \in \bar{\delta}_{stack}$

Die Definition enthält ein Verfahren, bei dem ausgehend von der ursprünglichen Transitionsmenge iterativ den Regeln entsprechend neue Transitionen zum Automaten hinzugefügt werden. Dieser Vorgang terminiert schließlich, ad die Menge aller möglichen Transitionen endlich ist, und man erhält die kleinste Transitionsmenge die alle Regeln erfüllt. Dass der auf diese Weise konstruierte Automat die Menge der Vorgängerkonfigurationen beschreibt ist

intuitiv durch die Regeln und die iterative Konstruktion klar. Man startet mit der ursprünglichen Transitionsmenge, also sind alle Konfigurationen die in der ursprünglichen Menge sind auch in der Menge der Vorgänger. Dies sind aber gerade die Konfigurationen, die durch den leeren Pfad erreicht werden können und damit Vorgänger ihrer selbst sind. Nun betrachtet man eine Zwischenmenge von Transitionen, in der alle bis zu diesem Punkt hinzugefügten Transitionen nur zulässige Konfigurationen zur Sprache des Automaten hinzugefügt haben. Wenn durch Anwendung der (RETURN) Regel eine Transition $(p, \gamma, \tilde{p})$ hinzugefügt wird, kann der Automat auch Konfigurationen $p\gamma w$ mit $w \in \Gamma^*$ akzeptieren, falls w von $\tilde{p}$ ausgehend akzeptiert wird. Dann akzeptiert der Automat auch die Konfiguration $\tilde{p}w$. Da aber nach Voraussetzung eine entsprechende Transition im PDS vorhanden ist, kann $p\gamma w$ im PDS zu $\tilde{p}w$ transformiert werden und ist damit Vorgängerkonfiguration einer Konfiguration die schon erkannt wird und damit Vorgänger einer Konfiguration in C. Analog gilt nach Hinzufügen einer Transition $(p, \gamma, \tilde{s})$ auf Grund der (STEP) Regel, dass Konfigurationen $p\gamma w$ akzeptiert werden können, wenn w von $\tilde{s}$ aus akzeptiert wird. Da aber nach Voraussetzung eine Transition $(\tilde{p}, \tilde{\gamma}, \tilde{s})$ existiert, kann vorher auch schon die Konfiguration $\tilde{p}\tilde{\gamma}w$ erkannt werden. Auf Grund der Existenz der passenden Regeln im PDS ist jedoch dann $p\gamma w$ eine direkte Vorgängerkonfiguration. Wiederum analog gilt die gleiche Begründung für Konfigurationen die durch, auf Grund der (CALL) Regel, hinzugefügte Transitionen erkannt werden können. Man kann formal beweisen [BEM97], dass:

Satz 2.18. *Für den zu einer regulären Menge $C \subseteq Conf$ von Konfigurationen eines PDS $\mathcal{P}$, gegeben durch einen $\mathcal{P}$-Automaten $\mathcal{A}$, durch den Saturierungsalgorithmus berechneten Automaten $\mathcal{A}^*$ gilt $\mathcal{L}(\mathcal{A}^*) = PRE^*(C)$.*

Abbildung 2.7 zeigt den $\mathcal{P}^*$-Automaten der durch Saturierung des in Abbildung 2.5 betrachteten $\mathcal{P}$-Automaten für Konfigurationen des in Abbildung 2.4 erstellten PDS $\mathcal{P}$ entsteht.

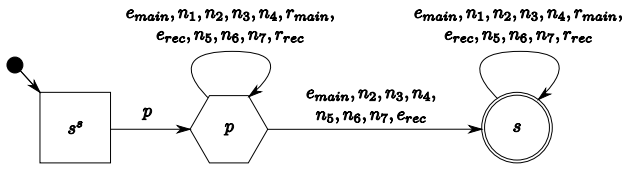


Abbildung 2.7: $\mathcal{P}^*$-Automat, der durch Saturierung des in Abbildung 2.5 betrachteten $\mathcal{P}$-Automaten entsteht

Bei genauerer Betrachtung des Regelsystems für die Konstruktion des Automaten und der Grammatik zur Beschreibung der Popsequenzen erkennt man gewisse Übereinstimmungen zwischen der Existenz von Transitionen im saturierten Automaten und der Existenz von vollständigen Ableitungsbäumen in der Grammatik. So erlauben die (INIT) Regeln der Grammatik eine Ableitung eines nichtterminalen Symbols $PopSeq_(s, \gamma, \tilde{s})$, wenn der ursprüngliche Automat eine Transition $(s, \gamma, \tilde{s})$ enthält. Die (INIT) Regel des Saturierungsalgorithmus garantiert, dass alle Transitionen des ursprünglichen Automaten auch im neuen Automaten enthalten sind. Desweiteren erlaubt zum Beispiel eine (STEP) Regel der Grammatik eine Ableitung eines nichtterminalen Symbols $PopSeq_(p, \gamma, \tilde{s})$, wenn eine passende

Regel $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}$ und eine Ableitung für das nichtterminale Symbol $PopSeq_(\tilde{p}, \tilde{\gamma}, \tilde{s})$ existieren. Analog garantiert die (STEP) Regel des Saturierungsalgorithmus die Existenz einer Transition $(p, \gamma, \tilde{s})$, wenn die gleiche Regel r und eine Transition $(\tilde{p}, \tilde{\gamma}, \tilde{s})$ existieren. Zu einem PDS $\mathcal{P}$ und einem $\mathcal{P}$-Automaten $\mathcal{A}$ kann man für den, durch den Saturierungsalgorithmus gewonnenen, Automaten $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$ und die zugehörige Grammatik zeigen, dass:

Satz 2.19. *Eine Transition $(s, \gamma, \tilde{s})$ ist genau dann in $\bar{\delta}_{stack}$ enthalten, wenn das nichtterminale Symbol $PopSeq_{(s,\gamma,\tilde{s})}$ einen vollständigen Ableitungsbaum hat, also eine Popsequenz für γ von s nach $\tilde{s}$ existiert.*

Die Transitionen des Automaten $\mathcal{A}^*$ beschreiben also alle nichtterminalen Symbole der Grammatik die für die Betrachtung des Problems interessant sind, da sie eine nichtleere Klasse von Popsequenzen charakterisieren. Durch Konstruktion des Automaten erhält man also die Teilmenge von nichtterminalen Symbole für die eine Betrachtung notwendig ist.

Wie oben schon beschrieben muss man zur Bestimmung des Gesamtgewichtes aller Pfade einer Konfiguration nun alle möglichen Kombinationen von zueinander passenden Klassen von Popsequenzen für die einzelnen Stacksymbole der Konfiguration betrachten. Durch Betrachtung des Automaten kann man die zu betrachtenden Klassen auf die nichtleeren beschränken und damit die Anzahl der Kombinationen reduzieren.

Im folgenden betrachtet man noch weitere Eigenschaften des berechneten Automaten. Dazu führen wir das Konzept eines Laufs innerhalb eines Automaten ein. Dies werden wir später bei unseren Betrachtungen verwenden und es ermöglicht auch hier schon eine einfache Darstellung der Beobachtungen aus [RSJM05]. Dabei beschreibt ein Lauf von einem Zustand s zu einem Zustand $\tilde{s}$ über ein Wort w eine Folge von Transitionen und Zuständen, die der Automat bei der Verarbeitung von w startend im Zustand s durchlaufen könnte um im Zustand $\tilde{s}$ zu enden.

Definition 2.20. Zu einem Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$ definieren wir einen *Lauf* als alternierende Folge $\phi = [\![s_1, \alpha_1, s_2, \ldots, \alpha_{n-1}, s_n]\!]$ von Zuständen $s_i \in S$ für $i \in \{1, \ldots, n\}$ und Symbolen $\alpha_i \in \Sigma$ mit $(s_i, \alpha_i, s_{i+1}) \in \delta$ für alle $i \in \{1, \ldots, n-1\}$, für $n \in \mathbb{N}$. Es existiert kein allgemeiner leerer Lauf, für jeden Zustand s existiert jedoch ein Lauf $[\![s]\!]$ der in diesem Zustand verweilt und dabei das leere Wort ϵ durchläuft. Zusätzlich seien mit $\phi^+ = s_1$ der Anfangszustand, mit $\phi^- = s_n$ der Endzustand und mit $\phi^w = \alpha_1 \ldots \alpha_{n-1}$ das Wort eines Laufs bezeichnet.

$Runs_{\mathcal{A}}$ sei die Menge aller Läufe des Automaten und $Runs_{\mathcal{A}}(s, w, \tilde{s}) = \{\phi \mid \phi^+ = s, \phi^- = \tilde{s}, \phi^w = w\}$ sei die Menge aller Läufe, die im Zustand s beginnen, in $\tilde{s}$ enden und dabei das Wort w durchlaufen. Desweiteren sei $Accept_{\mathcal{A}}(w) = \bigcup_{\tilde{s} \in F} Runs(s^s, w, \tilde{s})$ die Menge aller akzeptierenden Läufe für ein Wort w. Folglich akzeptiert der Automat ein Wort w, wenn $Accept_{\mathcal{A}}(w) \neq \emptyset$.

Wir definieren einen Konkatenationsoperator ; , der zu zwei beliebigen Läufen $\phi_1 = [\![s_1, \alpha_1, \ldots, s_n]\!], \phi_2 = [\![s_n, \alpha_n \ldots, s_m]\!] \in Runs_{\mathcal{A}}$ mit $\phi_1^- = \phi_2^+$ einen zusammengesetzten lauf

$$\phi_1; \phi_2 = [\![s_1, \alpha_1 \ldots, s_n, \alpha_n, \ldots, s_m]\!]$$

liefert. Für Mengen $M_1, M_2 \subseteq Runs_{\mathcal{A}}$ von Läufen mit $\phi_1^- = \phi_2^+$ für alle $\phi_1 \in M_1$, $\phi_2 \in M_2$ gelte $M_1; M_2 = \{\phi_1; \phi_2 \mid \phi_1 \in M_1, \phi_2 \in M_2\}$.

Betrachtet man nun einen akzeptierenden Lauf für eine Konfiguration im saturierten Automaten, so stellt man fest, dass nach durchlaufen der ersten Transition, die den Kontrollzustand erkennt, nur noch Transitionen aus $\bar{\delta}_{stack}$ auftreten. Diese Transitionen erkennen den Stackanteil der Konfiguration und enden in einem akzeptierenden Zustand. Nach vorherigen Überlegungen folgt aus der Existenz dieser Transitionen, dass die zugehörigen Klassen von Popsequenzen nichtleer sind und der Lauf liefert damit eine Kombination von nichtleeren zueinander passenden Klassen von Popsequenzen, die in einem akzeptierenden Zustand enden. Durch Kombination der Popsequenzen dieser Klassen erhält man also eine Teilmenge der akzeptierenden Pfade für die Konfiguration im PDS $\mathcal{PA}$. Umgekehrt ist jeder akzeptierende Pfad in einer Kombination von zueinander passenden nichtleeren Klassen von Popsequenzen, die in einem akzeptierenden Zustand enden, enthalten und diese entspricht einem akzeptierenden Lauf im saturierten Automaten, da nach den vorherigen Überlegungen die zugehörigen Transitionen in $\bar{\delta}_{stack}$ existieren.

Insgesamt liefert die Konstruktion des Automaten, neben der Reduzierung der Anzahl der zu betrachtenden Klassen von Popsequenzen, also alle relevanten Kombinationen dieser Klassen für eine Konfiguration c in Gestalt aller akzeptierenden Läufe der Konfiguration im Automaten.

Nun vermerkt man an jeder Transition des Automaten das Gewicht der zugehörigen Klassen von Popsequenzen und man erhält einen annotierten $\mathcal{P}^*$-Automaten $(\mathcal{A}^*, D, l)$, wobei:

Definition 2.21. Ein *annotierter $\mathcal{P}^*$-Automat* ist ein Tripel $(\mathcal{A}^*, M, l)$, wobei $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$ ein $\mathcal{P}^*$-Automat, M eine Menge von Annotationen und $l : \bar{\delta}_{stack} \to M$ eine Annotationsfunktion ist, die jeder Transition $t \in \bar{\delta}_{stack}$ einen Wert zuweist.

Zur Berechnung des Gewichtes aller Pfade zu einer beliebigen Konfiguration c muss man nun lediglich alle akzeptierenden Läufe dieser Konfiguration im Automaten betrachten. Dies sind nur endlich viele. Dann kann man aus den Gewichten der Läufe das gesuchte Gesamtgewicht kombinieren. Das Gewicht eines Laufs $[\![s^s p s_1 \gamma_1 \ldots \gamma_n s_{n+1}]\!] \in Accept(p\gamma_1 \ldots \gamma_n)$ ist dabei die Verkettung $l((s_1, \gamma_1, s_2)) \odot \ldots \odot l((s_n, \gamma_n, s_{n+1}))$ der Gewichte der dabei durchlaufenen Transitionen aus $\bar{\delta}_{stack}$. Wenn keine Transition aus $\bar{\delta}_{stack}$ durchlaufen werden, der Stack der gelesen Konfiguration also leer ist, sei das Gewicht 1. Auf einem leeren Stack können keine Transitionen ausgeführt werden, also ist der leere Pfad der einzige erreichende Pfad.

Da der Automat $\mathcal{A}^*$ die Menge $PRE^*(C)$ beschreibt, erkennt er alle Konfigurationen für die das Problem $\delta(c)$ eine nichttriviale Lösung hat. Der Automat muss inklusive der Annotationen nur einmal berechnet werden und liefert für jede dieser Konfigurationen das gesuchte

Ergebnis. Dabei kann, wie in [RSJM05] weiter ausgeführt wird, der Saturierungsalgorithmus mit der Berechnung der abstrakten Grammatik kombiniert werden, um direkt einen annotierten Automaten zu erhalten.

Um das Ergebnis einer klassischen vorwärts Datenflussanalyse mit Hilfe von WPDS zu berechnen, betrachtet man ein aus einem Flussgraphen gewonnenes PDS und die Menge der Konfigurationen in denen das PDS an einem bestimmten Programmpunkt steht. Dies drückt sich dadurch aus, dass der Programmpunkt an oberster Stelle des Stacks der Konfigurationen steht, der Rest des Stacks ist beliebig. Die Menge ist regulär und man kann das Verfahren für WPDS darauf anwenden um, für die Startkonfiguration das Gewicht über alle Pfade zu diesem Programmpunkt zu berechnen. In der Startkonfiguration liegt dabei nur der Einstiegspunkt des Programms auf dem Stack. Die Gewichte des WPDS werden entsprechend der betrachteten Datenflussanalyse $((D, \oplus), F)$ gewählt. Da für die Gewichte eine Verknüpfungsoperation gefordert wird, wählt man die Menge der monotonen Funktionen F mit dem Operator $\oplus_F$, so dass $f \sqsubseteq_F g \Leftrightarrow f(x) \sqsubseteq g(x)$ für alle $x \in D$, als vollständigen Verband und die Verknüpfung von Funktionen $\circ$ als Konkatenationsoperator $\odot$. Die konkrete Zuordnung von Gewichten zu den Transitionen des PDS ergibt sich aus der Zuordnung von Transferfunktionen zu Kanten den Flussgraphen in der Instanz der Datenflussanalyse zum gegebenen Flussgraphen. Im klassischen Fall betrachtet man alle erreichenden Pfade und die Anwendung ihrer Transferfunktionen auf die initiale Information. Dann bildet man die Kombination der Werte und erhält die MOP-Lösung. Im Fall von WPDS berechnet man nun die Kombination der Transferfunktionen aller erreichenden Pfade und wendet dann die gewonnenen Funktion auf die initiale Information an. Man kann also das Ergebnis einer Vorwärtsanalyse für eine Programmpunkt berechnen.

Zur Veranschaulichung greifen wir die in Abschnitt 2.1 eingeführte Klasse der Bitvektor-Probleme $((\{0,1\}, \oplus), \{KILL, ID, GEN\})$ noch einmal auf. Zur Verwendung mit WPDS definieren wir einen Halbring $(D, \odot, \oplus, 0, 1)$. Wie oben schon erwähnt verwendet man die Menge der Transferfunktionen als Wertebereich und die Verknüpfung von Funktionen $\circ$ zur Verknüpfung der Gewichte. Da jedoch für keine der Funktionen die Eigenschaften der 0 gelten, insbesondere $d \odot 0 = 0 \odot d = 0$ für alle $d \in D$, wird noch eine zusätzliches Element $ZERO$ eingeführt und die Definition der Verknüpfung entsprechend erweitert. Für Vorwärtsanalyse gilt dann

$$d_1 \odot d_2 = \begin{cases} d_2 \circ d_1 & \text{für } d_1, d_2 \neq ZERO \\ ZERO & \text{sonst} \end{cases}$$

und für Rückwärtsanalysen analog

$$d_1 \odot d_2 = \begin{cases} d_1 \circ d_2 & \text{für } d_1, d_2 \neq ZERO \\ ZERO & \text{sonst} \end{cases}.$$

Für den so definierten Verknüpfungsoperator gelten die geforderten Eigenschaften für $0 = ZERO$ und $1 = ID$. Für die Ordnung der Gewichte gilt nach Definition der Ordnung auf

Funktionen im Fall von sicheren Bitvektor-Problemen $KILL \sqsubseteq ID \sqsubset GEN \sqsubset ZERO$. Für mögliche Bitvektor-Probleme gilt analog $GEN \sqsubseteq ID \sqsubset KILL \sqsubset ZERO$.

Als konkretes Beispiel betrachten wir wiederum die Analyse der Verfügbarkeit des Ausdrucks $x - 1$ am Programmpunkt e_{rec} in dem in Abbildung 2.4 dargestellten Flussgraphen. Da es sich um ein sicheres Vorwärts-Bitvektor-Problem handelt gelte für den Halbring $(\{KILL, ID, GEN, ZERO\}, \oplus, \odot, ZERO, ID)$ die oben angegebene Definition. Die Zuordnung von Transferfunktionen zu Transitionen des PDS ergibt sich dabei aus der Zuordnung von Transferfunktionen zu den Kanten des Flussgraphen im Fall klassischer Datenflussanalyse.

Zur Berechnung der Datenflussinformation am Programmpunkt e_{rec} betrachtet man nun die Menge der Konfigurationen an diesem Programmpunkt $pe_{rec}(N_{\mathcal{G}})^*$ und konstruiert dazu einen $\mathcal{P}$-Automaten. Hierbei handelt es sich um den schon in Abbildung 2.5 dargestellten Automaten. Aus diesem wird ein annotierter $\mathcal{P}^*$-Automat für $PRE^*(pe_{rec}(N_{main} \cup N_{rec})^*)$ berechnet und es wird mit dem oben beschriebenen Verfahren der Wert für die Startkonfiguration pe_{main} bestimmt. Man erhält als Ergebnis die Transferfunktion GEN. Diese approximiert die Transferfunktionen aller erreichenden Pfade und durch Anwendung auf die initiale Information, der Ausdruck ist am Anfang des Programms nicht verfügbar, erhält man das Ergebnis, dass der Ausdruck verfügbar ist. Abbildung 2.8 zeigt das Ergebnis des Verfahrens konkret für die Berechnung der Verfügbarkeit des Ausdrucks $x - 1$ am Programmpunkt e_{rec}.

Es existieren weitere Techniken um dann auch Rückwärtsanalyse mit Hilfe von WPDS berechnen zu können. Mehr Informationen kann man in [RSJM05] nachlesen.

Das in diesem Kapitel vorgestellte Verfahren basiert auf dem Modell von PDS und dessen Eigenschaft sequentielle Programme mit Prozeduren darstellen zu können. Zur Lösung wurden Ergebnisse über abstrakte Grammatiken und gewisse Ergebnissen aus dem Model-Checking zur Konstruktion von Mengen von Vorgängerkonfigurationen verwendet. Unsere Idee das Verfahren zu erweitern, ist das hier zu Grunde gelegte Modell der PDS durch ein Modell zu ersetzen, dass zusätzlich parallele Ausführung von Programmen modellieren kann, aber gleichzeitig aus Sicht des Model-Checking die gleichen Voraussetzungen, also die Möglichkeit Mengen von Vorgängerkonfigurationen konstruieren zu können, mitbringt. Im nächsten Kapitel stellen wir dynamische Push-Down Netzwerke vor, welche genau diese Eigenschaften erfüllen.

2.3 Dynamische Push-Down Netzwerke

In [BMOT05] werden dynamische Push-Down Netzwerke eingeführt, die eine Erweiterung von PDS darstellen. Dabei wird nicht nur ein Stack im Sinne eines PDS betrachtet, sondern zu jedem Zeitpunkt betrachtet man eine Sammlung von Stacks. Jeder einzelne dieser Stacks kann dabei, unabhängig von den anderen, Operationen ausführen und in einen anderen Zustand übergehen. Zusätzlich erhalten die einzelnen Stacks die Möglichkeit zusätzliche

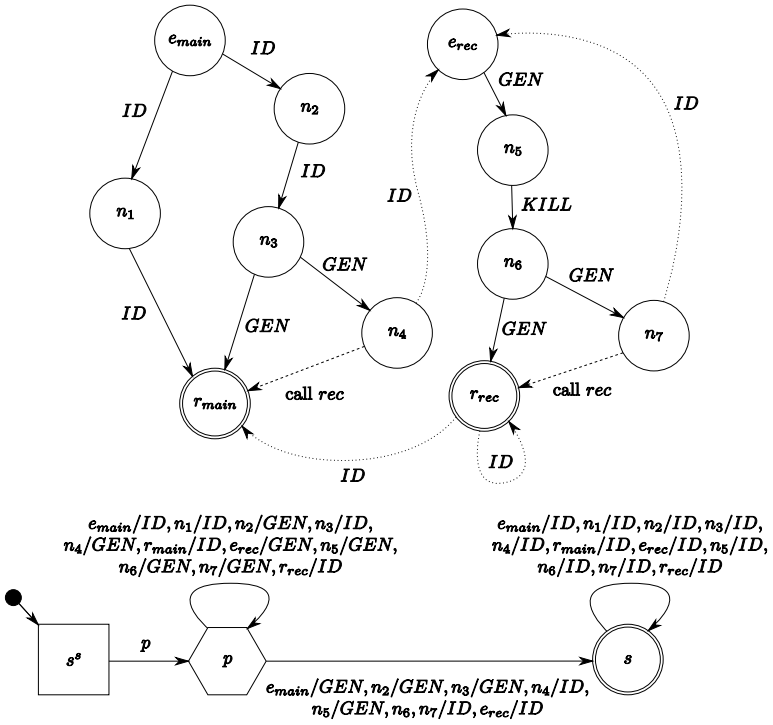


Abbildung 2.8: Annotierter $\mathcal{P}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, des in Abbildung 2.5 betrachteten $\mathcal{P}$-Automaten $\mathcal{A}$ entsteht

neue Stacks zu erzeugen. Die folgenden Darstellungen basieren auf den Überlegungen in [BMOT05]. Beweise der angesprochenen Sätze können dort, in teilweise anderer Notation oder Form, nachgelesen werden.

Definition 2.22. Ein dynamisches Push-Down Netzwerk (DPN) ist ein Tripel $\mathcal{M} = (P, \Gamma, \Delta)$. Dabei ist P eine endliche Menge von Kontrollzuständen und Γ eine endliche Menge von Stacksymbolen. Es gilt $P \cap \Gamma = \emptyset$. $\Delta = \Delta_1 \cup \Delta_2$ ist eine endliche Menge mit Transitionsregeln der Form:

1. $r = p\gamma \hookrightarrow \tilde{p} \in \Delta_1$ mit $p, \tilde{p} \in P$, $\gamma \in \Gamma$
2. $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma} \in \Delta_1$ mit $p, \tilde{p} \in P$, $\gamma, \tilde{\gamma} \in \Gamma$
3. $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \in \Delta_1$ mit $p, \tilde{p} \in P$, $\gamma, \tilde{\gamma}_1, \tilde{\gamma}_2 \in \Gamma$
4. $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta_2$ mit $p, \hat{p}, \tilde{p} \in P$, $\gamma, \hat{\gamma}, \tilde{\gamma} \in \Gamma$

Eine Konfiguration ist ein Wort aus der Menge $P\Gamma^*(\#P\Gamma^*)^*$. $Conf_{\mathcal{M}}$ sei die Menge aller Konfigurationen.

Ein Pfad ist eine Folge

$$\rho = [r_1, \ldots, r_n] \text{ mit } r_i \in \Delta, i \in \{1, \ldots, n\}, n \in \mathbb{N}_0$$

von Transitionsregeln. [] sei der leere Pfad und $Paths_{\mathcal{M}}$ sei die Menge aller Pfade.

Wir definieren einen Konkatenationsoperator ; , der zu zwei beliebigen Pfaden $\rho_1, \rho_2 \in Paths_{\mathcal{M}}$ einen zusammengesetzten Pfad

$$\rho_1;\rho_2 = \begin{cases} \rho_1 & \text{für } \rho_2 = [] \\ \rho_2 & \text{für } \rho_1 = [] \\ [r_1,\ldots,r_n,r_{n+1},\ldots,r_m] & \text{für } \rho_1 = [r_1,\ldots,r_n], \rho_2 = [r_{n+1},\ldots,r_m] \end{cases}$$

liefert. Für Mengen $M_1, M_2 \subseteq Paths_{\mathcal{M}}$ von Pfaden gelte $M_1;M_2 = \{\rho_1;\rho_2 \mid \rho_1 \in M_1, \rho_2 \in M_2\}$.

Für zwei Konfigurationen $c, \bar{c} \in Conf_{\mathcal{M}}$ und einen Pfad $\rho \in Paths_{\mathcal{M}}$ gelte dann die Übergangsrelation $c \xrightarrow{\rho}_{\mathcal{M}} \bar{c}$

1. $c \xrightarrow{\rho}_{\mathcal{M}} \bar{c}$ für $\rho = []$ und $\bar{c} = c$.
2. $c \xrightarrow{\rho}_{\mathcal{M}} \bar{c}$ für $\rho = [r];\tilde{\rho}$ mit $r = p\gamma \hookrightarrow \tilde{p}\tilde{c} \in \Delta$, $\tilde{\rho} \in Paths_{\mathcal{M}}$, wenn $v_1 \in P\Gamma^*(\#P\Gamma^*)^*\#$, $w \in \Gamma^*$, $v_2 \in \#P\Gamma^*(\#P\Gamma^*)^*$ mit $c = v_1 p\gamma w v_2$ und $v_1\tilde{c}wv_2 \xrightarrow{\tilde{\rho}}_{\mathcal{M}} \bar{c}$ existieren.

Abweichend von der Definition in [BMOT05] verzichten wir auf die Benennung von Transitionen mit Aktionen, da diese in unserem Fall keine weiteren Informationen liefern. Außerdem verwenden wir das Zeichen # zur Trennung der einzelnen Stacks einer Konfiguration, dies verbessert die Lesbarkeit und ermöglicht es in der Konstruktion der später verwendeten Automaten auf ϵ-Transitionen zu verzichten.

Wenn aus dem Kontext klar ist auf welches DPN $\mathcal{M}$ Bezug genommen wird, wird im folgenden auf die explizite Nennung im Index verzichtet.

Die Definition von DPN ermöglicht nun eine natürliche Modellierung von Programmen mit Prozeduren, da jeder einzelne Stack den Aufrufkontext eines Prozesses darstellen kann, und dynamischer Threaderzeugung, da jeder Stack einen Prozess darstellt und neue Prozesse, also Stacks, dynamisch erzeugt werden können. Insbesondere kann man die im Abschnitt 2.1 durch interprozedurale parallele Flussgraphen beschriebenen Programme untersuchen.

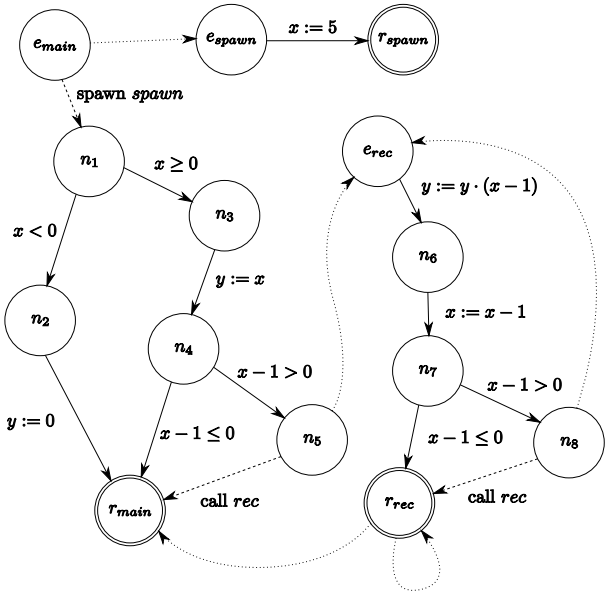
Definition 2.23. Gegeben ein System von Flussgraphen $\mathcal{G} = (Proc, (G_\pi)_{\pi \in Proc})$. Dann sei das zugehörige DPN definiert durch:

- $P = \{p\}$
- $\Gamma = N_{\mathcal{G}}$
- Δ enthält für jede Kante $e = (n, s, \tilde{n}) \in E_{\mathcal{G}}$ in einem der Flussgraphen eine der folgenden Transitionsregeln:
  - $pn \hookrightarrow p\tilde{n} \in \Delta$, falls $s \in BA$
  - $pn \hookrightarrow pe_\pi\tilde{n} \in \Delta$, falls $s = \text{call } \pi \in CA$ für ein $\pi \in Proc$

– $pn \hookrightarrow pe_\pi \# p\tilde{n} \in \Delta$, falls $s =$ spawn $\pi \in SA$ für ein $\pi \in Proc$

- Δ enthält für jede Prozedur $\pi \in Proc$ eine Transitionsregel $pr_\pi \hookrightarrow p \in \Delta$

Man sieht, dass sich DPN gut eignen kontextsensitive Analysen für Systeme von Flussgraphen mit Prozeduraufrufen und dynamischer Threaderzeugung zu formulieren. Prozeduraufrufe werden analog zu der Übersetzung in PDS für jeden einzelnen Stack einer DPN Konfiguration abgebildet und ermöglichen daher eine genaue Modellierung von Prozeduraufrufen mehrerer paralleler Threads. Die dynamische Erzeugung eines Threads wird von DPN direkt unterstützt und auf eine Regel zur Erzeugung eines neuen Stacks abgebildet. Dieser kann im folgenden unabhängig vom erzeugenden Thread operieren und parallel zu diesem Anweisungen ausführen. Das DPN kann also genau die Pfade durchlaufen, die im Flussgraphen untersucht werden sollen. Abbildung 2.9 zeigt die Umwandlung eines parallelen interprozeduralen Flussgraphen $\mathcal{G}$ in ein DPN $\mathcal{M}$.



$$
\begin{array}{lcl}
r_1 & : & pe_{main} \hookrightarrow pe_{spawn} \# pn_1 \\
r_2 & : & pe_{spawn} \hookrightarrow pr_{spawn} \\
r_3 & : & pn_1 \hookrightarrow pn_2 \\
r_4 & : & pn_2 \hookrightarrow pr_{main} \\
r_5 & : & pn_3 \hookrightarrow pn_4 \\
r_6 & : & pn_4 \hookrightarrow pn_5 \\
r_7 & : & pn_4 \hookrightarrow pr_{main} \\
r_8 & : & pn_5 \hookrightarrow pe_{rec} r_{main} \\
r_9 & : & pe_{rec} \hookrightarrow pn_6 \\
r_{10} & : & pn_6 \hookrightarrow pn_7 \\
r_{11} & : & pn_7 \hookrightarrow pn_8 \\
r_{12} & : & pn_7 \hookrightarrow pr_{rec} \\
r_{13} & : & pn_8 \hookrightarrow pe_{rec} r_{rec} \\
r_{14} & : & pr_{rec} \hookrightarrow p \\
r_{15} & : & pr_{main} \hookrightarrow p \\
r_{16} & : & pr_{spawn} \hookrightarrow p
\end{array}
$$

Abbildung 2.9: Umwandlung eines parallelen interprozeduralen Flussgraphen $\mathcal{G}$ in ein DPN $\mathcal{M}$

Für uns ist, wie schon für PDS gesehen, die Menge der Vorgängerkonfigurationen einer regulären Menge C interessant.

Definition 2.24. Zu einer Menge $C \subseteq Conf$ von Konfigurationen eines DPN $\mathcal{M}$ sei

$$PRE^*(C) = \{c \in Conf \mid c \xrightarrow{\rho} \bar{c} \text{ für ein } \bar{c} \in C \text{ und } \rho \in Paths\}$$

die Menge der vorhergehenden Konfigurationen.

Analog zu der Konstruktion in Abschnitt 2.2 wird in [BMOT05] ein Verfahren definiert, das auch für DPN eine direkte Konstruktion der Menge $PRE^*(C)$ für eine reguläre Menge

$C \subseteq Conf$ zulässt. Insbesondere ist die konstruierte Menge dabei wieder regulär. Dabei muss auch hier die Menge C von einem bestimmten Automatentypen beschrieben werden.

Definition 2.25. Ein *$\mathcal{M}^*$-Automat* $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$ ist ein spezieller endlicher Automat zur Beschreibung einer regulären Menge von Konfigurationen eines DPN $\mathcal{M} = (P, \Gamma, \Delta)$. Dabei erfüllt der Automat die folgenden Bedingungen:

1. $\Sigma = P \cup \Gamma \cup \{\#\}$
2. $S = S_c \cup S_s$ mit $S_c \cap S_s = \emptyset$
3. Für alle $s \in S_c, p \in P$ existiert ein eindeutiger Zustand $s_p \in S_s$, $S_p = \{s_p \mid s \in S_c, p \in P\}$ sei die Menge aller dieser Zustände
4. $\bar{\delta} = \bar{\delta}_{separator} \cup \bar{\delta}_{state} \cup \bar{\delta}_{stack}$ wobei
   - $\bar{\delta}_{separator} \subseteq S_s \times \{\#\} \times S_c$
   - $\bar{\delta}_{state} = \{(s, p, s_p) \mid s \in S_c, p \in P\}$
   - $\bar{\delta}_{stack} \subseteq (S_p \times \Gamma \times S_p) \cup (S_s \times \Gamma \times (S_s \setminus S_p))$
5. $s^s \in S_c$
6. $F \subseteq S_s$

Ein *$\mathcal{M}$-Automat* $\mathcal{A}$ ist ein $\mathcal{M}^*$-Automat, für dessen Transitionsmenge, in diesem Fall in der Regel mit δ bezeichnet, sogar $\delta_{stack} \subseteq S_s \times \Gamma \times (S_s \setminus S_p)$ gilt.

Man kann sich einen $\mathcal{M}$-Automaten als Sammlung von $\mathcal{P}$-Automaten vorstellen. Dabei enthält die Menge S_c die Anfangszustände dieser Automaten. In S_p ist zu jedem dieser Anfangszustände ein eindeutiger Satz von Zuständen enthalten, die den Kontrollzuständen des DPN entsprechen. Diese sind durch die mit den Kontrollzuständen beschrifteten Transitionen in δ_{state} mit ihrem jeweiligen Anfangszustand verbunden. Die Menge S_s bildet zusammen mit δ_{stack} den Teil der Automaten die den Stack erkennen. Dabei können sich in einem $\mathcal{M}$-Automat mehrere Teilautomaten diese Zustände teilen. Die einzelnen Teilautomaten sind dann noch durch die Transitionen in $\delta_{separator}$ miteinander verbunden, die jeweils zum Anfangszustand des nächsten Teilautomaten führen. Das Akzeptieren einer Konfiguration startet dann mit dem Akzeptieren des ersten Stacks durch den, mit dem Startzustand des gesamten Automaten ausgezeichneten, ersten Teilautomaten. Dann wird durch den Übergang über eine Transition aus $\delta_{separator}$ das Trennzeichen, das diesen Stack vom nächsten trennt, eingelesen und der nächste Stack kann von dem damit erreichten Teilautomaten gelesen werden. Dies setzt sich solange fort, bis alle Stacks abgearbeitet wurden. Die Konstruktion eines solchen Automaten ist für jede reguläre Menge von Konfigurationen C möglich. Abbildung 2.10 zeigt ein Beispiel für einen $\mathcal{M}$-Automaten zur Beschreibung einer Mengen von Konfigurationen des in Abbildung 2.9, aus dem Flussgraphen $\mathcal{G}$, konstruierten DPN $\mathcal{M}$.

Wenn C durch einen solchen Automaten gegeben ist, kann durch folgendes Verfahren ein Automat konstruiert werden, der $PRE^*(C)$ beschreibt.

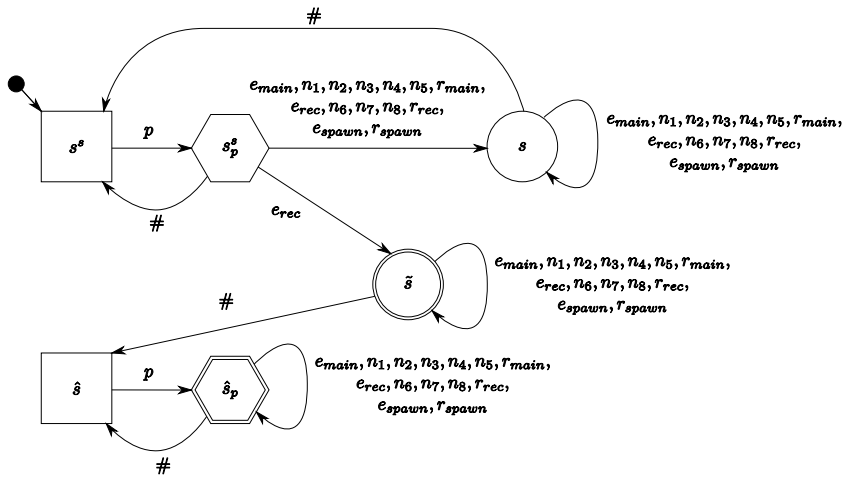


Abbildung 2.10: $\mathcal{M}$-Automat für die Menge der Konfigurationen $(pN^*_{\mathcal{G}}\#)^* pe_{rec}N^*_{\mathcal{G}}(\#pN^*_{\mathcal{G}})^*$ des in Abbildung 2.9 dargestellten DPN $\mathcal{M}$ zum Flussgraphen $\mathcal{G}$

Definition 2.26. Zu einer regulären Menge $C \subseteq Conf$ von Konfigurationen eines DPN $\mathcal{M}$ gegeben durch einen $\mathcal{M}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$ definieren wir einen Saturierungsalgorithmus zur Berechnung des Automaten $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$. Dabei ist $\bar{\delta} = \bar{\delta}_{separator} \cup \bar{\delta}_{state} \cup \bar{\delta}_{stack}$ mit $\bar{\delta}_{separator} = \delta_{separator}$, $\bar{\delta}_{state} = \delta_{state}$ und $\bar{\delta}_{stack}$ ist die kleinste Menge, die die folgenden Bedingungen erfüllt:

(INIT) $\bar{\delta}_{stack} \supseteq \delta_{stack}$

(RETURN) $r = p\gamma \hookrightarrow \tilde{p} \in \Delta$, dann $t = (s_p, \gamma, s_{\tilde{p}}) \in \bar{\delta}_{stack}$ für alle $s \in S_c$

(STEP) $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma} \in \Delta$ und $\tilde{t} = (s_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}$ für $s \in S_c$, $\tilde{s} \in S_s$, dann $t = (s_p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$

(CALL) $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \in \Delta$ und $\tilde{t}_1 = (s_{\tilde{p}}, \tilde{\gamma}_1, \tilde{s}_1), \tilde{t}_2 = (\tilde{s}_1, \tilde{\gamma}_2, \tilde{s}_2) \in \bar{\delta}_{stack}$ für $s \in S_c$, $\tilde{s}_1, \tilde{s}_2 \in S_s$, dann $t = (s_p, \gamma, \tilde{s}_2) \in \bar{\delta}_{stack}$

(SPAWN) $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta$ und $\hat{t} = (s_{\hat{p}}, \hat{\gamma}, \hat{s}), \tilde{t} = (\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}, (\hat{s}, \#, \bar{s}) \in \delta_{separator}$ für $s, \bar{s} \in S_c$, $\hat{s}, \tilde{s} \in S_s$, dann $t = (s_p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$

Analog zur Berechnung bei PDS enthält die Definition ein Verfahren, bei dem ausgehend von der ursprünglichen Transitionsmenge iterativ den Regeln entsprechend neue Transitionen zum Automaten hinzugefügt werden. Dieser Vorgang terminiert irgendwann, da die Menge aller möglichen Transitionen endlich ist, und man erhält die kleinste Transitionsmenge die alle Regeln erfüllt. Dass der auf diese Weise konstruierte Automat die Menge der Vorgängerkonfigurationen beschreibt ist intuitiv durch die Regeln klar. Die ersten vier Regeln sind dabei fast identisch mit den Regeln des Verfahrens für PDS, deshalb ist die Begründung für die Korrektheit hier die gleiche. Der einzige Unterschied besteht in der Struktur der Automaten. Konfigurationen eines PDS bestehen aus einem Stack, deshalb hat, wie oben schon beschrieben, ein $\mathcal{P}$-Automat einen Bereich, der diesen Stack einliest. Bei

der Betrachtung von Konfigurationen eines DPN muss man gegebenenfalls mehrere Stacks untersuchen. Dies spiegelt sich auch der Struktur eines $\mathcal{M}$-Automaten wieder, der deshalb, wie oben schon erwähnt, aus einer Aneinanderreihung von $\mathcal{P}$-Automaten besteht. Da die einzelnen Stacks einer Konfiguration unabhängig voneinander operieren, kann man bei der Betrachtung von Vorgängerkonfigurationen die einzelnen Stacks unabhängig voneinander betrachten. Deshalb wendet man das für PDS entwickelte Verfahren auf jeden Teilautomaten eines $\mathcal{M}$-Automaten einzeln an. Man betrachtet die Regeln also nicht nur für den Startzustand s^s, sondern für jeden Anfangszustand eines der Teilautomaten $s \in S_c$. Die einzige Ausnahme bilden Konfigurationen, die durch Erzeugung eines neuen Stacks aus einer Vorgängerkonfiguration hervorgehen. Hierbei müssen zwei benachbarte Teilautomaten betrachtet werden. Wenn durch Anwendung der (SPAWN) Regel eine Transition $(s_p, \gamma, \tilde{s})$ hinzugefügt wird, kann der Automat auch Konfigurationen $v_1 p\gamma w v_2$ akzeptieren, wobei v_1 von s^s nach s und wv_2 von $\tilde{s}$ ausgehend akzeptiert werden muss. Da nach Voraussetzung Transitionen $(s_{\hat{p}}, \hat{\gamma}, \hat{s})$, $(\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s})$ und $(\hat{s}, \#, \bar{s})$ existieren, erkennt der Automat also $\hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}$ von s nach $\tilde{s}$ und durchläuft dabei zwei benachbarte Teilautomaten. Der Automat akzeptiert insgesamt also auch $c_1\hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}c_2$. Da aber eine entsprechende Regel im PDS vorhanden ist, kann $c_1 p\gamma c_2$ zu $c_1\hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}$ transformiert werden und ist damit Vorgängerkonfiguration einer Konfiguration die schon erkannt wird. Man kann formal beweisen, dass:

Satz 2.27. *Für eine reguläre Menge $C \subseteq Conf$ von Konfiguration eines DPN $\mathcal{M}$, gegeben durch einen $\mathcal{M}$-Automaten $\mathcal{A}$, beschreibt der durch den Saturierungsalgorithmus berechnete Automat $\mathcal{A}^*$ die Menge $PRE^*(C)$, also gilt $\mathcal{L}(\mathcal{A}^*) = PRE^*(C)$.*

Abbildung 2.11 zeigt den Automaten der durch Saturierung des in Abbildung 2.10 betrachteten Automaten entsteht.

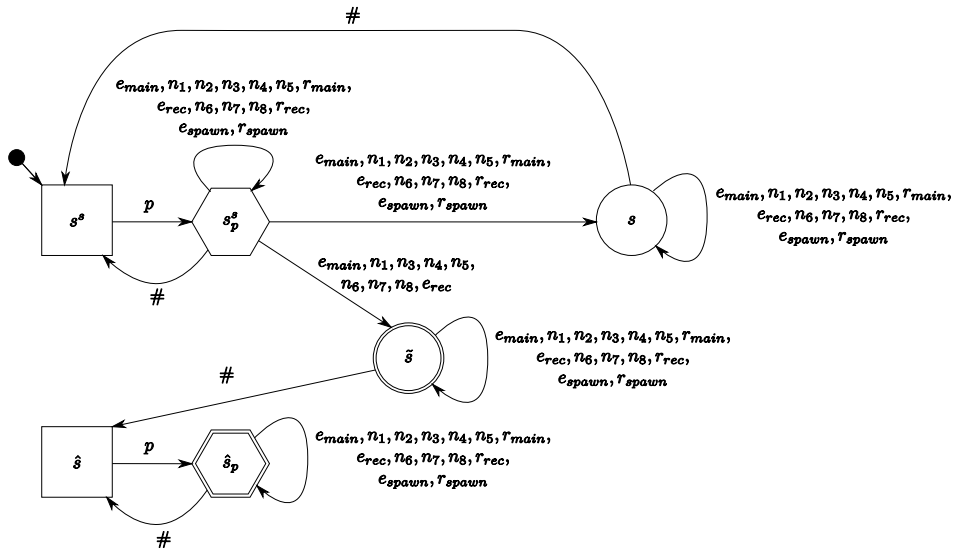


Abbildung 2.11: Automat, der durch Saturierung des in Abbildung 2.10 betrachteten Automaten entsteht

Wir haben also zumindest einen Teil der für WPDS untersuchten Ergebnisse auch für DPN. Im nun folgenden Hauptteil dieser Arbeit werden wir uns damit beschäftigen, ob auch die weiteren für WPDS untersuchten Techniken und Ergebnisse auf DPN übertragbar sind. Damit wäre eine Ausweitung der Analysefähigkeiten der WPDS auf parallele Programme möglich.

KAPITEL 3

Analyse gewichteter dynamischer Pushdown-Netzwerke

In diesem Kapitel wollen wir nun gewichtete dynamische Push-Down-Netzwerke untersuchen. Dazu bauen wir auf den in Kapitel 2 vorgestellten Grundlagen auf. Unsere anfänglichen Überlegungen basieren auf den in [RSJM05] eingeführten WPDS. In Abschnitt 3.1 übertragen wir die Problemstellung für WPDS auf DPN und untersuchen die Beobachtungen für Popsequenzen von PDS im Rahmen von DPN. Dabei führen wir eine benötigte neue Darstellung von Pfaden als Bäume ein. Im Anschluss daran charakterisieren wir in Abschnitt 3.2 die Klassen von Popsequenzen und zeigen, dass wir mit ihnen die für die Analyse interessanten Übergänge darstellen können. Anders als bei der Betrachtung von PDS greifen wir dabei nicht auf eine Grammatik zurück, sondern verwenden ein System von Ungleichungen zur Beschreibung der Mengen. In Abschnitt 3.3 wird dann, analog zum Übergang von der Grammatik zur abstrakten Grammatik für WPDS, mit Hilfe abstrakter Interpretation vom Ungleichungssystem für die Popsequenzen zu einem Ungleichungssystem der zugehörigen Gewichte übergegangen. Um der neuen Struktur der Pfade als Bäume Rechnung zu tragen, müssen dabei die verwendeten Gewichte, gegenüber den Gewichten für WPDS, erweitert werden.

3.1 Ansatz

Analog zu der Betrachtung bei WPDS wollen wir nun für DPN eine Aussage über die Pfade von einer beliebigen Konfiguration c zu einer Konfiguration in einer gegebenen regulären Menge C machen. Um auch hier bestimmte Informationen berechnen zu können, wird jede Transitionsregel des DPN mit einem Gewicht bestückt. Da die Pfade eines DPN die

gleiche sequentielle Struktur haben, wie die Pfade eines PDS, behalten wir die Struktur eines Halbrings zur Beschreibung der Gewichte bei. Man kann dann analog zur Definition bei WPDS das Gesamtgewicht eines Pfades als die sequentielle Verknüpfung der Gewichte der auf ihm durchlaufenen Transitionen definieren. Dann interessiert man sich für die Kombination der Gewichte aller Pfade die von c aus eine Konfiguration in C erreichen. Das zu lösende Problem ist dann:

Definition 3.1. Sei $\mathcal{S} = (D, \oplus, \odot, 0, 1)$ ein beschränkter idempotenter Halbring und $f : \Delta \to D$ eine Funktion die jeder Regel des Push-Down Netzwerkes ein Gewicht aus dem Halbring zuweist, sowie $C \subseteq Conf$ eine reguläre Menge von Konfigurationen.

- Die Menge der erreichenden Pfade für eine Konfiguration $c \in Conf$ ist dann

$$Paths(c, C) = \{\rho \in Paths \mid c \xrightarrow{\rho} \tilde{c} \text{ für ein } \tilde{c} \in C\}.$$

- Das Gewicht $\alpha(\rho) \in D$ eines Pfades $\rho \in Paths$, sei

$$\alpha(\rho) = \begin{cases} 1 & \text{für } \rho = [] \\ \alpha(\tilde{\rho}) \odot f(r) \text{ für } \rho = \tilde{\rho}; [r] \end{cases}.$$

  Für eine Menge $M \subseteq Paths$ von Pfaden gelte $\alpha(M) = \bigoplus\{\alpha(\rho) \mid \rho \in M\}$.

- Das "generalized pushdown predecessor problem" besteht dann darin, den Wert

$$\delta(c) = \alpha(Paths(c, C))$$

  zu berechnen.

Der Ansatz bei WPDS ist, die erreichenden Pfade, für eine Konfiguration c und eine Menge C von Zielkonfigurationen, beschrieben durch einen $\mathcal{P}$-Automaten $\mathcal{A}$, eines PDS $\mathcal{P}$, in akzeptierende Pfade der Konfiguration in einem erweiterten PDS $\mathcal{PA}$ einzubetten. Das erweiterte PDS simuliert dabei zusätzlich zum Verhalten des ursprünglichen PDS, das Verhalten des Automaten $\mathcal{A}$. Die akzeptierenden Pfade zeichnen sich durch das Leeren des Stacks der betrachteten Konfiguration aus. Daraus kann man auf eine Aufteilung der Pfade in Popsequenzen schließen, die jeweils ein Symbol vom Stack entfernen. Diese Aufteilung wirkt sich auch auf die eingebetteten erreichenden Pfade aus, und man kann durch eine Beschreibung der Popsequenzen auch die erreichenden Pfade beschreiben. Für die Popsequenzen gilt zudem eine Einteilung in Klassen, die in direktem Zusammenhang mit der Konstruktion eines saturierten Automaten, zur Beschreibung der Menge der Vorgängerkonfigurationen, von C stehen.

Eine direkte Übertragung dieses Ansatzes auf ein DPN $\mathcal{M}$ ist nun nicht möglich, da man kein DPN konstruieren kann, dass zusätzlich das Verhalten eines gegebenen $\mathcal{M}$-Automaten simuliert. Dies liegt darin begründet, dass ein $\mathcal{M}$-Automat Konfigurationen bestehend aus mehreren Stacks durch Erreichen eines Zustands akzeptieren kann. Ein DPN kann jedoch einmal erzeugte Stacks nicht wieder entfernen. Deshalb existiert keine DPN-Regel die das

Verhalten eines $\mathcal{M}$-Automaten beschreibt, wenn dieser einen Stack der Konfiguration erkannt hat und dann in einem gewissen Zustand mit dem Erkennen des nächsten Stacks beginnt.

Betrachten wir deshalb nochmal genauer ein PDS $\mathcal{P} = (P, \Gamma, \Delta)$ und eine reguläre Menge von Konfigurationen $C \subseteq Conf$, beschrieben durch den $\mathcal{P}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$.

Wie in Abschnitt 2.2 erwähnt kann man eine Popsequenz in dem erweiterten PDS $\mathcal{PA}$ für ein Symbol γ von einem Zustand s in einen Zustand $\tilde{s}$, also einen Pfad ρ mit $s\gamma \xrightarrow{\rho}_{\mathcal{PA}} \tilde{s}$, in zwei Phasen aufteilen. Die erste Phase enthält dabei nur Transitionen des ursprünglichen PDS und die zweite Phase nur Transitionen, die den Automaten simulieren. Man kann dann drei Typen von Popsequenzen unterscheiden.

Für den ersten Typ besteht die Popsequenz nur aus Transitionen des ursprünglichen PDS. Es gilt also schon $s\gamma \xrightarrow{\rho}_{\mathcal{P}} \tilde{s}$ im ursprünglichen PDS $\mathcal{P}$. Dann handelt es sich auch bei den Zuständen $s, \tilde{s}$ um Kontrollzustände von $\mathcal{P}$. Da im betrachteten $\mathcal{P}$-Automaten $\mathcal{A}$ die mit den Kontrollzuständen beschrifteten Kanten δ_{state} von Startzustand s^s zu den mit den Kontrollzuständen bezeichneten Zuständen des Automaten existieren, kann man insgesamt sagen, dass eine Popsequenz dieses Typs, die Konfiguration $s\gamma$ in eine Konfiguration überführt, die in $\mathcal{A}$ von einem Lauf von s^s nach $\tilde{s}$ erkannt wird. Dieser Lauf besteht nur aus der Kante $(s, \tilde{s}, \tilde{s}) \in \delta_{state}$. Umgekehrt kann aus jedem Paar von Pfad und Lauf, die diese Eigenschaften erfüllen, eine Popsequenz konstruiert werden.

Der zweite Typ von Popsequenzen beginnt in einem Zustand $s \in P$ und enthält dann eine, möglicherweise auch leere, Folge von Transitionen des PDS $\mathcal{P}$, und Folge von Transitionen die den Automaten $\mathcal{A}$ simulieren. Betrachtet man dann nur die Transitionen des ursprünglichen PDS, erhält man einen Pfad ρ der die Konfiguration $s\gamma$ in eine Konfiguration $p\tilde{w}$, mit $p \in P, \tilde{w} \in \Gamma^+$, überführt. Für den leeren Pfad gilt $p\tilde{w} = s\gamma$. Der zweite Teil der Popsequenz simuliert nun einen Lauf in $\mathcal{A}$, der den Stack $\tilde{w}$ der erreichten Konfiguration vom Zustand p zum Zustand $\tilde{s}$ erkennt. Da dabei mindestens eine Transition des Automaten durchlaufen wird, ist $\tilde{s}$ ein Zustand des Automaten, der nicht gleichzeitig Kontrollzustand des PDS ist, da keine Kanten in δ_{stack} existieren, die zu Zuständen in P führen. Da der Automat wiederum eine Kante $(s, p, p) \in \delta_{state}$ enthält, kann man insgesamt sagen, das die Popsequenz die Konfiguration $s\gamma$ in eine Konfiguration überführt, die in $\mathcal{A}$ von einem Lauf von s^s nach $\tilde{s}$ erkannt wird. Umgekehrt kann man aus jedem Paar von Pfad und Lauf, die diese Eigenschaften erfüllen, eine Popsequenz konstruieren.

Für die ersten beiden Type kann man also insgesamt sagen, dass eine Popsequenz für ein Symbol γ von einem Zustand $p \in P$ zu einem Zustand $\tilde{s} \in S$ genau dann existiert, wenn ein Pfad ρ im PDS $\mathcal{P}$ und eine Konfiguration $\bar{c}$ existieren, mit $p\gamma \xrightarrow{\rho} \bar{c}$ und $Runs_{\mathcal{A}}(s^s, \bar{c}, \tilde{s}) \neq \emptyset$.

Für den dritten Typ von Popsequenzen gilt $s \in S \setminus P$ und eine Sequenz besteht dann nur aus einer Transitionen, die den Automaten $\mathcal{A}$ simuliert. Der zum ursprünglichen PDS gehörende Teilpfad ist hier mit dem leeren Pfad [] beschrieben. Dieser kann jedoch, streng genommen, auf der Konfiguration $s\gamma$ nicht ausgeführt werden, da $s \notin P$. Daher müssen wir für diesen

Fall eine getrennte Beschreibung einer Popsequenz angeben. Eine Popsequenz vom dritten Typ existiert genau dann, wenn im Automaten $\mathcal{A}$ eine Kante $(s, \gamma, \tilde{s})$ existiert.

Mit dieser Beschreibung von Popsequenzen durch Pfade im PDS und Läufe im Automaten, unabhängig vom erweiterten PDS $\mathcal{PA}$, betrachten wir dann ein DPN $\mathcal{M} = (P, \Gamma, \Delta)$ und eine reguläre Menge von Konfigurationen $C \subseteq Conf$, beschrieben durch den $\mathcal{M}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$.

Sei dann ρ_1 ein Pfad des DPN, so dass für die Konfiguration $p\gamma_1$, mit $p \in P, \gamma_1 \in \Gamma$ eine Konfiguration $\bar{v}p$, mit $\bar{v} \in (P\Gamma^*\#)^*$, existiert mit $p\gamma_1 \xrightarrow{\rho_1}_{\mathcal{M}} \bar{v}p$ und es existiert ein Zustand $\tilde{s}_p \in S$, so dass ein Lauf in $\mathcal{A}$ existiert, der $\bar{v}p$ von s^s nach $\tilde{s}_p$ erkennt. Es gilt also $Runs_{\mathcal{A}}(s^s, vp, \tilde{s}_p) \neq \emptyset$. Wenn wir eine analoge Definition von Popsequenzen für DPN, wie für PDS, annehmen, handelt es sich also um eine Popsequenz vom ersten Typ. Da in einem $\mathcal{M}$-Automaten nur Kanten in δ_{state} mit Kontrollzuständen bezeichnet sind, besteht der Lauf für $\bar{v}p$ aus einem Teillauf, der $\bar{v}$ von s^s nach $\tilde{s}$ erkennt und der Kante $(\tilde{s}, p, \tilde{s}_p)$. Betrachte einen zweiten Pfad ρ_2 des DPN, so dass für die Konfiguration $p\gamma_2$, mit $\gamma_2 \in \Gamma$ eine Konfiguration c, mit $c \in P\Gamma^*(\#P\Gamma^*)^*$, existiert mit $p\gamma_2 \xrightarrow{\rho_2}_{\mathcal{M}} c$ und es existiert ein Zustand $\hat{s} \in F$, so dass ein Lauf in $\mathcal{A}$ existiert, der $\bar{c}$ von $\tilde{s}$ nach $\hat{s}$ erkennt. Es gilt also $Runs_{\mathcal{A}}(\tilde{s}, \bar{c}, \hat{s}) \neq \emptyset$. Wenn wir eine analoge Definition von Popsequenzen für DPN, wie für PDS, annehmen, handelt es sich also um eine Popsequenz vom zweiten Typ.

Verknüpft man nun die beiden Pfade, erhält man, nach Definition der Übergangsrelation für DPN, einen Pfad $\rho_1; \rho_2$ für den $p\gamma_1\gamma_2 \xrightarrow{\rho_1;\rho_2} \bar{v}\bar{c}$ gilt. Da nach Voraussetzung Läufe in $\mathcal{A}$ existieren, die $\bar{v}$ von s^s nach $\tilde{s}$ und $\bar{c}$ von $\tilde{s}$ nach $\hat{s}$ erkennen, existiert auch ein Lauf der $\bar{v}\bar{c}$ von s^s nach $\hat{s}$ erkennt. Da $\hat{s} \in F$ handelt es sich um einen akzeptierenden Lauf und es gilt $\bar{v}\bar{c} \in C$. Damit handelt es sich bei dem Pfad $\rho_1; \rho_2$ um einen erreichenden Pfad für die Konfiguration $p\gamma_1\gamma_2$. Man kann also aus den Popsequenzen einen erreichenden Pfad konstruieren. Umgekehrt kann man bei dieser Definition jedoch nicht jeden erreichenden Pfad durch solche Popsequenzen beschreiben. Betrachten wir dazu die in Abbildung 3.1 dargestellte Situation.

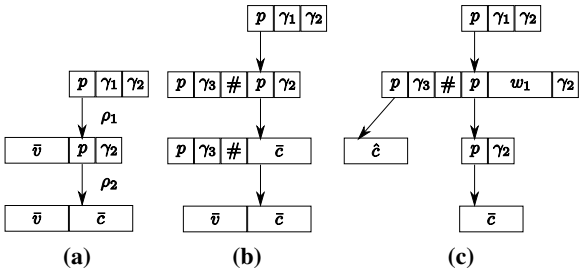


Abbildung 3.1: Beispiele für zwei Pfade von $p\gamma_1\gamma_2$ nach $\bar{v}\bar{c}$, die die gleichen Popsequenzen ausführen, und die zugehörige Darstellung als Baum.

Bei einer analogen Definition einer Popsequenz vom ersten und zweiten Typ für DPN, enthalten die Pfade neben den Anweisungen, die auf dem anfänglichen Stack ausgeführt

werden, um auf diesem ein Symbol zu bearbeiten, auch Anweisungen, die auf den dabei neu erzeugten Stacks ausgeführt werden. Da die einzelnen Stacks unabhängig voneinander Operationen ausführen können, können die parallel ausgeführten Anweisungen an beliebiger Stelle zwischen den Anweisungen auf dem anfänglichen Stack auftreten. Betrachtet man nun eine Verkettung von Popsequenzen, treten die parallel ausgeführten Anweisungen der ersten Popsequenz jedoch nur im Teil der Ausführung auf den anfänglichen Stack auf, Abbildung 3.1a, der zur ersten Popsequenz gehört. Ein erreichender Pfad, in dem diese Anweisungen erst parallel zur zweiten Popsequenz, Abbildung 3.1b, ausgeführt werden, kann so durch Verkettung der Sequenzen nicht konstruiert werden.

Die Darstellung der Popsequenzen als Pfade führt also dazu, dass wir nicht mehr alle erreichenden Pfade betrachten können. Da das Problem in dem Einfügen parallel ausgeführter Anweisung in den Programmablauf liegt, abstrahieren wir von einer Betrachtung von Pfaden in denen die Anweisungen der Stacks gemischt ausgeführt werden und betrachten Bäume, in denen jeder Knoten für eine Anweisung steht und jede Verzweigung des Baumes steht für die Aufspaltung des Kontrollflusses in zwei parallel ablaufende Flüsse. Wir ordnen also jedem Stack seinen eigenen Ausführungszweig zu, Abbildung 3.1c. Jeder Baum beschreibt dann eine Menge von Pfade, die durch eine Mischung der folgenden Anweisungen an jeder Verzweigung entstehen.

Definition 3.2. Ein *Baum* τ sei rekursiv wie folgt definiert, dabei sei $Trees_{\mathcal{M}}$ die Menge aller Bäume:

1. $\tau = []$
2. $\tau = [r](\tilde{\tau})$ mit $\tilde{\tau} \in Trees$, $r \in \Delta_1$
3. $\tau = [r](\hat{\tau}, \tilde{\tau})$ mit $\hat{\tau}, \tilde{\tau} \in Trees$ und $r \in \Delta_2$

Wir definieren rekursiv einen Konkatenationsoperator ; , der zu zwei beliebigen Bäumen $\tau_1, \tau_2 \in Trees_{\mathcal{M}}$ einen zusammengesetzten Baum $\tau_1; \tau_2$ liefert. Es gilt:

$$\tau_1; \tau_2 = \begin{cases} \tau_2 & \text{für } \tau_1 = [] \\ [r](\tilde{\tau_1}; \tau_2) & \text{für } \tau_1 = [r](\tilde{\tau_1}) \\ [r](\hat{\tau_1}, \tilde{\tau_1}; \tau_2) & \text{für } \tau_1 = [r](\hat{\tau_1}, \tilde{\tau_1}) \end{cases} .$$

Der definierte Konkatenationsoperator hängt den zu konkatenierenden Baum an den äußersten rechten Ast des bestehenden Baumes. Durch Konkatenation zweier Bäume verlängert man also die Ausführung auf dem anfänglichen Stack, die durch den äußersten rechten Ast dargestellt wird. Anweisungen für neu erzeugte Stacks liegen auf einem nach links abzweigenden Ast.

Da nun Konfigurationen eines DPN aus mehreren Stacks bestehen können ist die Darstellung einer gesamten Ausführung auf einer solchen Konfiguration problematisch. Parallel laufende Prozesse sollten ihren eigenen Ast im gesamten Baum haben. Deshalb benutzen wird das Konzept einer Hecke:

Definition 3.3. Eine *Hecke* σ ist ein Tupel $\sigma = \langle \tau_1, \ldots, \tau_n \rangle$ von Bäumen $\tau_i \in Trees$ für $i \in \{1, \ldots, n\}$ und $n \in \mathbb{N}$. Eine Hecke besteht immer mindestens aus einem Baum, da auch Konfigurationen immer mindestens aus einem Stack bestehen. Mit $Hedges_{\mathcal{M}}$ sei die Menge aller Hecken bezeichnet. Zusätzlich sei $Hedges^1_{\mathcal{M}} = \{\langle \tau \rangle \mid \tau \in Trees_{\mathcal{M}}\}$ die Menge der einstelligen Hecken, diese kann mit der Menge der Bäume identifiziert werden und spielt daher bei der Betrachtung von einfachen Stacks ein hervorgehobene Rolle. Wenn im folgenden von Bäumen die Rede ist, sei damit immer eine einstellige Hecke gemeint.

Wir definieren einen Konkatenationsoperator $;$, der zu einer beliebigen Hecke $\sigma_1 = \langle \tau_1, \ldots, \tau_n \rangle \in Hedges_{\mathcal{M}}$ beliebiger Stelligkeit und einem beliebigen Baum $\sigma_2 = \langle \tau \rangle \in Hedges^1_{\mathcal{M}}$ eine zusammengesetzte Hecke

$$\sigma_1; \sigma_2 = \langle \tau_1, \ldots, \tau_n; \tau \rangle$$

liefert. Analog definieren wir einen zweiten Konkatenationsoperator $\triangleright$, der in der gleichen Situation eine zusammengesetzte Hecke

$$\sigma_1 \triangleright \sigma_2 = \langle \tau_1, \ldots, \tau_n, \tau \rangle$$

liefert. Für Mengen $M_1 \subseteq Hedges_{\mathcal{M}}$ und $M_2 \subseteq Hedges^1_{\mathcal{M}}$ von Hecken gelte dann $M_1; M_2 = \{\sigma_1; \sigma_2 \mid \sigma_i \in M_i\}$ und $M_1 \triangleright M_2 = \{\sigma_1 \triangleright \sigma_2 \mid \sigma_i \in M_i\}$.

Für zwei Konfigurationen $c, \bar{c} \in Conf$ und eine Hecke $\sigma \in Hedges_{\mathcal{M}}$ gelte dann die Übergangsrelation $c \overset{\sigma}{\Longrightarrow}_{\mathcal{M}} \bar{c}$:

1. für $\sigma = \langle \tau_1, \ldots, \tau_n \rangle$, wenn $c = p_1 w_1 \# \ldots \# p_n w_n$ und $\bar{c} = \bar{c_1} \# \ldots \# \bar{c_n}$, für $p_i w_i \in P\Gamma^*$ und $\bar{c_i} \in P\Gamma^*(\#P\Gamma^*)^*$, mit $p_i w_i \overset{\langle \tau_i \rangle}{\Longrightarrow}_{\mathcal{M}} \bar{c_i}$ für $i \in \{1, \ldots, n\}$.
2. für $\sigma = \langle [] \rangle$, wenn $c = pw = \bar{c}$ mit $p \in P$ und $w \in \Gamma^*$.
3. für $\sigma = \langle [r](\tilde{\tau}) \rangle$ mit $r = p\gamma \hookrightarrow \tilde{p}\tilde{w} \in \Delta_1$, $\tilde{w} \in \Gamma^*, |\tilde{w}| \in \{0, 1, 2\}$[1], $\tilde{\tau} \in Trees_{\mathcal{M}}$, wenn $c = p\gamma w$, für ein $w \in \Gamma^*$, und es gilt $\tilde{p}\tilde{w}w \overset{\langle \tilde{\tau} \rangle}{\Longrightarrow}_{\mathcal{M}} \bar{c}$.
4. für $\sigma = \langle [r](\hat{\tau}, \tilde{\tau}) \rangle$ mit $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta_2$, $\hat{\tau}, \tilde{\tau} \in Trees_{\mathcal{M}}$, wenn $c = p\gamma w$, für ein $w \in \Gamma^*$, und $\bar{c} = \hat{c}\#\tilde{c}$, mit $\hat{c}, \tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$, und $\tilde{p}\tilde{\gamma}w \overset{\langle \tilde{\tau} \rangle}{\Longrightarrow}_{\mathcal{M}} \tilde{c}$, sowie $\hat{p}\hat{\gamma} \overset{\langle \hat{\tau} \rangle}{\Longrightarrow}_{\mathcal{M}} \hat{c}$.

Eine Hecke enthält dann zu jedem anfänglichen Stack einer Konfiguration einen Baum. Die Übergangsrelation für Hecken ist dementsprechend so definiert, dass jeder dieser Bäume seinen eigenen Beitrag zum erreichen der Zielkonfiguration leisten muss. Der Übergang für einzelne Stacks ist dann auf Basis der in den Knoten des Baumes vermerkten Transitionen definiert. Bei einer Verzweigung der Bäume wird für den neu erzeugten Stack ein eigener Baum ausgeführt.

[1] Die Länge eines Wortes w ist dabei definiert als $|w| = \begin{cases} 0 & \text{für } w = \epsilon \\ |\tilde{w}| + 1 & \text{mitfür } w = \tilde{w}\lambda \end{cases}$.

Der erste Konkatenationsoperator entspricht dem Konkatenationsoperator für Bäume, wobei nur der letzte Baum der Hecke beachtet wird. Der zweiten Operator fügt den angehängten Baum als neuen letzten Baum in die Hecke ein. Mit Hilfe dieser Operatoren, kann man dann eine Hecke als Konkatenation von Bäumen darstellen.

Mit dieser Definition von Hecken und Bäumen können wir nun, analog zu WPDS die Klasse der Popsequenzen, der beteiligte Pfad hat jetzt die Form eines Baumes, für ein Symbol γ von einem Zustand s zu einem Zustand $\tilde{s}$ definieren.

Definition 3.4. Für $s, \tilde{s} \in S_s$, $\gamma \in \Gamma$ sei

$$L(s,\gamma,\tilde{s}) = \begin{cases} \{\langle\tau\rangle \in Hedges^1 \mid p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}, & \text{für } s = \bar{s}_p \in S_p, \bar{s} \in S_c, p \in P \\ \text{für ein } \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*, & \\ \text{mit } Runs_{\mathcal{A}}(\bar{s},\bar{c},\tilde{s}) \neq \emptyset\} & \\ \{\langle[]\rangle\} & \text{für } s \notin S_p, (s,\gamma,\tilde{s}) \in \delta_{stack} \\ \emptyset & \text{sonst} \end{cases}$$

die Menge der Popsequenzen für γ von s nach $\tilde{s}$.

Die Popsequenzen werden also mit den in ihnen ausgeführten Bäumen identifiziert. Bei der Bezeichnung eines Baumes als Popsequenz aus einer Klasse für γ von s nach $\tilde{s}$ wird also impliziert, dass ein passender Lauf existiert, der die erreichte Konfiguration erkennt. In Abschnitt 3.2 werden wir zeigen, dass wir mit Hilfe dieser Klassen von Popsequenzen die Menge der erreichenden Hecken beschreiben können. Diese definieren wir als:

Definition 3.5. Zu einer Konfiguration $c \in Conf$ und einer Menge von Konfigurationen $C \subseteq Conf$ definieren wir die Menge der erreichenden Hecken:

$$Hedges(c,C) = \{\sigma \in Hedges \mid c \overset{\sigma}{\Longrightarrow} \bar{c} \text{ für ein } \bar{c} \in C\}.$$

Um zu zeigen, dass wir durch Untersuchung der erreichenden Hecken auch die ursprünglich zu untersuchende Menge der erreichenden Pfade betrachten können, definieren wir zu einer Hecke die Menge der Pfade die durch diese charakterisiert werden. Dazu benötigen wir eine Konstruktion, die zu zwei bestehenden Pfade die Menge der Pfade liefert, die durch eine parallel Abarbeitung, auch *Interleaving*, entstehen können.

Definition 3.6. Wir definieren einen Interleavingoperator $\|$, der zu zwei beliebigen Pfaden $\rho_1, \rho_2 \in Paths$ rekursiv die Menge der interleavten Pfade $\rho_1 \parallel \rho_2$ liefert. Es gelte:

$$\rho_1 \parallel \rho_2 = \begin{cases} \{\rho_2\} & \text{für } \rho_1 = [] \\ \{\rho_1\} & \text{für } \rho_2 = [] \\ [r_1];(\tilde{\rho_1} \parallel \rho_2) \cup [r_2];(\rho_1 \parallel \tilde{\rho_2}) & \text{für } \rho_1 = [r_1];\tilde{\rho_1}, \rho_2 = [r_2];\tilde{\rho_2} \end{cases}$$

Für Mengen $M_1, M_2 \subseteq Paths$ von Pfaden gelte $M_1 \parallel M_2 = \bigcup\{\rho_1 \parallel \rho_2 \mid \rho_1 \in M_1, \rho_2 \in M_2\}$.

Aus der Definition der Menge der interleavten Pfade ergeben sich einige nützliche Eigenschaften für die weiteren Betrachtungen.

Korollar 3.7. *Für den Operator $\|$ gilt:*

1. *$\|$ distributiert über $\cup$. Es gilt also $(M_1 \cup M_2) \parallel M_3 = (M_1 \parallel M_3) \cup (M_2 \parallel M_3)$. $\|$ ist sogar universell distributiv.*
2. *$\|$ ist kommutativ. Es gilt also $\rho_1 \parallel \rho_2 = \rho_2 \parallel \rho_1$ für $\rho_1, \rho_2 \in Paths$. Und dann auch $M_1 \parallel M_2 = M_2 \parallel M_1$ für $M_1, M_2 \subseteq Paths$.*
3. *$\|$ ist assoziativ. Es gilt also $\rho_1 \parallel (\rho_2 \parallel \rho_3) = (\rho_1 \parallel \rho_2) \parallel \rho_3$ für $\rho_1, \rho_2, \rho_3 \in Paths$. Und dann auch $M_1 \parallel (M_2 \parallel M_3) = (M_1 \parallel M_2) \parallel M_3$ für $M_1, M_2, M_3 \subseteq Paths$.*
4. *Für $\rho_1, \ldots, \rho_n \in Paths$ Pfade und $i_1, \ldots, i_m$ die Indizes der nichtleeren Pfade, das heißt $\rho_{i_j} = [r_{i_j}]; \tilde{\rho}_{i_j}$ für $r_{i_j} \in \Delta$, $\tilde{\rho}_{i_j} \in Paths$, für $j \in \{1, \ldots, m\}$, und $\rho_k = []$ für $k \notin \{i_1, \ldots, i_m\}$, gilt*

$$\rho_1 \parallel \ldots \parallel \rho_n = \rho_{i_1} \parallel \ldots \parallel \rho_{i_m} = \bigcup_{j=1}^{m} [r_{i_j}]; (\rho_{i_1} \parallel \ldots \parallel \tilde{\rho}_{i_j} \parallel \ldots \parallel \rho_{i_m}).$$

Und für Mengen $M_1, \ldots, M_n \subseteq Paths$ von Pfaden und $i_1, \ldots, i_m$ die Indizes, so dass $M_{i_j} = [r_{i_j}]; \tilde{M}_{i_j}$ für $r_{i_j} \in \Delta$, $\tilde{M}_{i_j} \subseteq Paths$, für $j \in \{1, \ldots, m\}$, und $M_k = \{[]\}$ für $k \notin \{i_1, \ldots, i_m\}$, gilt

$$M_1 \parallel \ldots \parallel M_n = M_{i_1} \parallel \ldots \parallel M_{i_m} = \bigcup_{j=1}^{m} [r_{i_j}]; (M_{i_1} \parallel \ldots \parallel \tilde{M}_{i_j} \parallel \ldots \parallel M_{i_m}).$$

Beweis. zu 1.: Folgt direkt aus der Definition von $\|$ für Mengen von Pfaden.

zu 2.: Folgt durch Induktion über $min(|\rho_1|, |\rho_2|)$ aus der Definition[2]. Die Erweiterung für Mengen folgt aus 1.

zu 3.: Folgt analog zu 2.

zu 4.: Folgt mit 2. und 3. aus Anwendung der Definition. Die Erweiterung für Mengen folgt dann aus 1.

□

Damit können wir nun zu einer Hecke die Menge der Pfade definieren, die von dieser Hecke dargestellt werden. Dabei werden die Pfade der einzelnen Bäume einer Hecke, die parallel auf ihren Stacks arbeiten, durch den Interleavingoperator miteinander kombiniert. Die Pfade eines Baumes entstehen durch eine "preorder" Auflistung der an die Knoten annotierten Regeln. Dabei wird bei einer Verzweigung statt einer Verkettung der Pfade der Äste die Menge der Interleavings dieser angehängt. Die Anweisungen eines neu erzeugten Threads

[2]Die Länge $|\rho|$ eines Pfades ρ ist dabei definiert als $|\rho| = \begin{cases} 0 & \text{für } \rho = [] \\ 1 + |\tilde{\rho}| & \text{für } \rho = [r]; \tilde{\rho} \end{cases}$.

werden also in den weiteren Programmablauf nach dem Zeitpunkt der Erzeugung beliebig eingefügt.

Definition 3.8. Wir definieren eine Funktion ψ, die zu einer beliebigen Hecke $\sigma \in Hedges$ rekursiv die Menge der zugehörigen Pfade liefert. Es gilt:

$$\psi(\sigma) = \begin{cases} \psi(\langle\tau_1\rangle) \parallel \ldots \parallel \psi(\langle\tau_n\rangle) & \text{für } \sigma = \langle\tau_1, \ldots, \tau_n\rangle \\ \{[]\} & \text{für } \sigma = \langle[]\rangle \\ [r]; \psi(\langle\tilde{\tau}\rangle) & \text{für } \sigma = \langle[r](\tilde{\tau})\rangle \\ [r]; (\psi(\langle\hat{\tau}\rangle) \parallel \psi(\langle\tilde{\tau}\rangle)) & \text{für } \sigma = \langle[r](\hat{\tau}, \tilde{\tau})\rangle \end{cases}$$

Für eine Menge $S \subseteq Hedges$ von Hecken gelte $\psi(S) = \bigcup\{\psi(\sigma) \mid \sigma \in S\}$

Der leere Baum beschreibt also nur den leeren Pfad. Der Baum bestehend aus einem Wurzelknoten und einem nachfolgenden Ast beschreibt also die Pfade, die mit der am Wurzelknoten stehenden Regel beginnen, und anschließend durch einen beliebigen, durch den Ast beschriebenen, Pfad fortgesetzt werden. Dies entspricht also einer sequentiellen Ausführung der Regeln. Der Baum bestehend aus einem Wurzelknoten und zwei nachfolgenden Ästen beschreibt die Pfade, die mit der Regel des Wurzelknoten beginnen und dann durch ein Interleaving der, durch die beiden Äste beschriebenen, Pfade fortgesetzt werden. Dies entspricht der Idee, dass die zwei Äste zwei parallele Threads beschreiben, deren Anweisungen interleavt ausgeführt werden.

Um den Übergang zur Betrachtung von Hecken nun zu rechtfertigen, zeigen wir:

Satz 3.9. *Für eine Konfiguration $c \in Conf$ und eine Menge von Konfigurationen $C \subseteq Conf$ gilt $\psi(Hedges(c, C)) = Paths(c, C)$.*

Für den Beweis betrachten wir zunächst einige Hilfsmittel. Bei der Betrachtung von Hecken werden die Anweisungsfolgen eines jeden Stacks in einer Konfiguration getrennt betrachtet. Um nun zu einem Pfad im DPN eine zugehörige Hecke zu erstellen, müssen wir die einzelnen Anweisungen des Pfades den Teilen der Konfiguration zuordnen können, auf denen sie ausgeführt werden. Dabei sollten für die Teilkonfigurationen Teilpfade entstehen, deren Interleaving den ursprünglichen Pfad enthält. Man kann zeigen, dass:

Lemma 3.10. *Wenn $c_1\# \ldots \#c_n \xrightarrow{\rho} \bar{c}$ mit $c_1, \ldots, c_n, \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, dann existieren $\bar{c}_1, \ldots, \bar{c}_n \in P\Gamma^*(\#P\Gamma^*)^*$ und $\rho_1, \ldots, \rho_n \in Paths$ mit $\bar{c} = \bar{c}_1\# \ldots \#\bar{c}_n$ und $c_i \xrightarrow{\rho_i} \bar{c}_i$ für $i \in \{1, \ldots, n\}$ und $\rho \in \rho_1 \parallel \ldots \parallel \rho_n$.*

Beweis. Induktion über die Länge von ρ:

1. $|\rho| = 0$

   Dann handelt es sich bei dem betrachteten Pfad um den leeren Pfad $\rho = []$. Wähle $\bar{c}_i = c_i$ und $\rho_i = []$ für $i \in \{1 \ldots n\}$, dann folgt die Behauptung aus der Definition des Übergangs durch den leeren Pfad und der Tatsache, dass das Interleaving von leeren Pfaden nur den leeren Pfad enthält.

2. $|\rho| > 0$

Dann enthält der betrachtete Pfad mindestens eine Transition und hat also die Form $\rho = [r];\tilde{\rho}$ für ein $r \in \Delta$ und $\tilde{\rho} \in Paths$. Nach Definition der Übergänge in einem DPN gilt $c_1\#\ldots\#c_n \xrightarrow{[r];\tilde{\rho}} \bar{c}$ für $r = p\gamma \hookrightarrow v \in \Delta$, $v \in P\Gamma\#P\Gamma \cup P \cup P\Gamma \cup P\Gamma\Gamma$, dann, wenn $c_i = v_1 p\gamma v_2$, mit $v_1 \in (P\Gamma^*\#)^*$, $v_2 \in (\#P\Gamma^*)^*$, für ein $i \in \{1,\ldots,n\}$ und $c_1\#\ldots\#c_{i-1}\#v_1vv_2\#c_{i+1}\#\ldots\#c_n \xrightarrow{\tilde{\rho}} \bar{c}$. Dann existieren nach Induktionsvoraussetzung für den echt kürzeren Pfad $\tilde{\rho}$ die Teilkonfigurationen $\bar{c}_1,\ldots,\bar{c}_n \in P\Gamma^*(\#P\Gamma^*)^*$ und Teilpfade $\tilde{\rho}_1,\ldots,\tilde{\rho}_n \in Paths$ mit $\bar{c} = \bar{c}_1\#\ldots\#\bar{c}_n$ und $c_j \xrightarrow{\tilde{\rho}_j} \bar{c}_j$, für $j \in \{1,\ldots,n\}, j \neq i$, und $v_1vv_2 \xrightarrow{\tilde{\rho}_i} \bar{c}_i$, sowie $\tilde{\rho} \in \tilde{\rho}_1 \parallel \ldots \parallel \tilde{\rho}_n$. Wählt man nun die gleichen Teilkonfigurationen $\bar{c}_1,\ldots,\bar{c}_n$ und dazu die Teilpfade $\rho_j = \tilde{\rho}_j$ für $j \in \{1,\ldots,n\}, j \neq i$ für die Teile die von der Regel r nicht betroffen sind, sowie den um diese Regel verlängerten Teilpfad $\rho_i = [r];\tilde{\rho}_i$ für den betroffenen Teil. Dann gilt die Behauptung auch für den Pfad ρ, denn $c_j \xrightarrow{\rho_j} \bar{c}_j$ für $j \in \{1,\ldots,n\}, j \neq i$ nach Induktionsvoraussetzung und $c_i \xrightarrow{\rho_i} \bar{c}_i$ nach Definition des Übergangs in DPN und Induktionvoraussetzung. Desweiteren gilt nach Definition der interleavten Pfade und Induktionsvoraussetzung $\rho = [r];\tilde{\rho} \in [r];(\tilde{\rho}_1 \parallel \ldots \parallel \tilde{\rho}_n) \subseteq \rho_1 \parallel \ldots \parallel \rho_n$.

□

Umgekehrt können wir auch zeigen, dass sogar jeder Pfad, der durch das Interleaving von Teilpfaden, die jeweils eine Teilkonfiguration transformieren, entsteht, die gesamte Konfiguration transformiert.

Lemma 3.11. *Wenn $c_i \xrightarrow{\rho_i} \bar{c}_i$ für $i \in \{1,\ldots,n\}$ mit $\rho_i \in Paths$ und $c_1,\ldots,c_n,\bar{c}_1,\ldots,\bar{c}_n \in P\Gamma^*(\#P\Gamma^*)^*$, dann gilt $c_1\#\ldots\#c_n \xrightarrow{\rho} \bar{c}_1\#\ldots\#\bar{c}_n$ für alle $\rho \in \rho_1 \parallel \ldots \parallel \rho_n$.*

Beweis. Induktion über die Summe der Längen der ρ_i:

1. $\sum_{i=1}^{n} |\rho_i| = 0$

Bei allen betrachteten Pfade handelt es sich um den leeren Pfad. Nach Definition gilt $c_i \xrightarrow{\rho_i} \bar{c}_i$ dann, wenn $\bar{c}_i = c_i$ für $i \in \{1,\ldots n\}$. Für die Menge der interleavten Pfade gilt $\rho_1 \parallel \ldots \parallel \rho_n = \{[]\}$ und damit die Behauptung, denn nach Definition des Übergangs in DPN und der Voraussetzung gilt $c_1\#\ldots\#c_n \xrightarrow{[]} \bar{c}_1\#\ldots\#\bar{c}_n$.

2. $\sum_{i=1}^{n} |\rho_i| > 0$

Es existiert also mindestens ein Pfad, der nicht leer ist und dessen erste Transition auf einem interleavten Pfad als erstes ausgeführt werden kann. Seien dann $i_1,\ldots,i_m$ die Indizes der nichtleeren Pfade. Dann gilt, nach Korollar 3.7, $\rho_1 \parallel \ldots \parallel \rho_n = \rho_{i_1} \parallel \ldots \parallel \rho_{i_m} = \bigcup_{k=1}^{m} [r_{i_k}];(\rho_{i_1} \parallel \ldots \parallel \tilde{\rho_{i_k}} \parallel \ldots \parallel \rho_{i_m})$. Dann gilt für einen beliebigen Pfad $\rho \in \rho_1 \parallel \ldots \parallel \rho_n$ aus der Menge der Interleavings, dass $\rho = [r_{i_k}];\tilde{\rho}$ für ein $k \in \{1,\ldots,m\}$ und $\tilde{\rho} \in \rho_{i_1} \parallel \ldots \parallel \tilde{\rho_{i_k}} \parallel \ldots \parallel \rho_{i_m} = \rho_1 \parallel \ldots \parallel \tilde{\rho_{i_k}} \parallel \ldots \parallel \rho_n$. Der Pfad hat als erste Transition also die erste Transition eines der nichtleeren Pfade und als Rest ein Interleaving

aus den anderen Pfade sowie dem Rest des Pfades, dem die Transition entnommen wurde. Über den Pfad, aus dem die erste Transition entnommen wird, ist bekannt, dass er auf der Teilkonfiguration c_{i_k} arbeitet. Sei also $r_{i_k} = p\gamma \hookrightarrow v \in \Delta, v \in P\Gamma\#P\Gamma \cup P \cup P\Gamma \cup P\Gamma\Gamma$, dann gilt $c_{i_k} \xrightarrow{\rho_{i_k}} \bar{c}_{i_k}$, wenn $c_{i_k} = v_1 p\gamma v_2$, mit $v_1 \in (P\Gamma^*\#)^*, v_2 \in (\#P\Gamma^*)^*$, und $v_1 v v_2 \xrightarrow{\tilde{\rho_{i_k}}} \bar{c}_{i_k}$. Wenn man nun statt c_{i_k} die Teilkonfiguration $v_1 v v_2$ und statt ρ_{i_k} den Pfad $\tilde{\rho_{i_k}}$ betrachtet, gilt nach Induktionsvoraussetzung für das Interleaving $\tilde{\rho}$, aus Pfaden die in der Summe kürzer sind, dass $c_1\# \ldots \#v_1 v v_2\# \ldots \#c_n \xrightarrow{\tilde{\rho}} \bar{c}_1\# \ldots \#\bar{c}_n$. Nach Definition des Übergangs in DPN und der Voraussetzung gilt dann auch $c_1\# \ldots \#c_n \xrightarrow{\rho} \bar{c}_1\# \ldots \#\bar{c}_n$.

□

Diese Ergebnisse ermöglichen es nun in einem ersten Schritt zu zeigen, dass alle Pfade, die durch einen Baum beschrieben werden, auf einem Stack die gleiche Transformation ausführen wie der Baum selbst.

Lemma 3.12. *Sei $\langle\tau\rangle \in Hedges^1$ eine Hecke mit $c \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$, für $c, \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, dann gilt für alle Pfade $\rho \in \psi(\langle\tau\rangle)$, dass $c \xrightarrow{\rho} \bar{c}$.*

Beweis. Strukturelle Induktion über den Aufbau von τ:

1. $\tau = []$

   Nach Definition gilt $c \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ dann, wenn $c = pw = \bar{c}$, für $p \in P, w \in \Gamma^*$. Außerdem gilt $\psi(\langle\tau\rangle) = \{[]\}$ und damit die Behauptung, denn nach Definition gilt $pw \xrightarrow{[]} pw$.
2. $\tau = [r](\tilde{\tau})$

   Nach Definition gilt $c \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ für $r = p\gamma \hookrightarrow \tilde{p}\tilde{w} \in \Delta_1$, $\tilde{w} \in \Gamma^*, |\tilde{w}| \in \{0, 1, 2\}$, und $\tilde{\tau} \in Trees$ dann, wenn $c = p\gamma w$, mit $w \in \Gamma^*$, und $\tilde{p}\tilde{w}w \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}$. Nach Induktionsvoraussetzung gilt dann $\tilde{p}\tilde{w}w \xrightarrow{\tilde{\rho}} \bar{c}$ für alle Pfade $\tilde{\rho} \in \psi(\langle\tilde{\tau}\rangle)$ die sich aus dem restlichen Baum ergeben. Nach Definition des Übergangs in DPN und der Voraussetzung gilt damit auch $c \xrightarrow{[r];\tilde{\rho}} \bar{c}$. Da zudem $\psi(\langle\tau\rangle) = \psi(\langle[r](\tilde{\tau})\rangle) = [r]; \psi(\langle\tilde{\tau}\rangle)$ gilt, folgt die Behauptung.
3. $\tau = [r](\hat{\tau}, \tilde{\tau})$

   Nach Definition gilt $c \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ für $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta_2$ und $\hat{\tau}, \tilde{\tau} \in Trees$ dann, wenn $c = p\gamma w$, mit $w \in \Gamma^*$, und $\bar{c} = \hat{c}\#\tilde{c}$, für $\hat{c}, \tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$, mit $\tilde{p}\tilde{\gamma}w \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$ und $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$. Nach Induktionsvoraussetzung gilt dann $\tilde{p}\tilde{\gamma}w \xrightarrow{\tilde{\rho}} \tilde{c}$ für alle Pfade $\tilde{\rho} \in \psi(\langle\tilde{\tau}\rangle)$, die sich als Interleaving des rechten Teilbaums ergeben. Und $\hat{p}\hat{\gamma} \xrightarrow{\hat{\rho}} \hat{c}$ für alle Pfade $\hat{\rho} \in \psi(\langle\hat{\tau}\rangle)$, die sich als Interleaving des linken Teilbaums ergeben. Nach Lemma 3.11 gilt dann $\hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}w \xrightarrow{\rho} \bar{c}$ für alle $\bar{\rho} \in \psi(\langle\hat{\tau}\rangle) \parallel \psi(\langle\tilde{\tau}\rangle)$. Nach Definition des Übergangs in DPN und der Voraussetzung gilt damit auch $c \xrightarrow{[r];\bar{\rho}} \bar{c}$. Da zudem $\psi(\langle\tau\rangle) = \psi(\langle[r](\hat{\tau}, \tilde{\tau})\rangle) = [r]; (\psi(\langle\hat{\tau}\rangle) \parallel \psi(\langle\tilde{\tau}\rangle))$ gilt, folgt die Behauptung.

□

Im nächsten Schritt erweitern wir dies auf Hecken die auf einer Konfiguration mit beliebig vielen Stacks arbeitet.

Lemma 3.13. *Sei $\sigma = \langle \tau_1, \ldots, \tau_n \rangle \in Hedges$ eine Hecke mit $c \overset{\sigma}{\Longrightarrow} \bar{c}$, für $c, \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, dann gilt für alle Pfade $\rho \in \psi(\sigma)$, dass $c \xrightarrow{\rho} \bar{c}$.*

Beweis. Nach Definition gilt $c \overset{\sigma}{\Longrightarrow} \bar{c}$, wenn $c = p_1 w_1 \# \ldots \# p_n w_n$, mit $p_i w_i \in P\Gamma^*$, und $\bar{c} = \bar{c}_1 \# \ldots \# \bar{c}_n$, für $\bar{c}_i \in P\Gamma^*(\#P\Gamma^*)^*$, mit $p_i w_i \overset{\langle \tau_i \rangle}{\Longrightarrow} \bar{c}_i$. Nach Lemma 3.12 folgt dann aber schon $p_i w_i \xrightarrow{\rho_i} \bar{c}_i$ für alle $\rho_i \in \psi(\langle \tau_i \rangle)$ und somit nach Lemma 3.11 auch $c \xrightarrow{\rho} \bar{c}$ für alle $\rho \in \psi(\langle \tau_1 \rangle) \parallel \ldots \parallel \psi(\langle \tau_n \rangle) = \psi(\sigma)$. □

Wir stellen also fest, dass jeder Pfad, der durch eine erreichenden Hecke beschrieben wird, ein erreichender Pfad ist. Damit wäre eine Inklusionsrichtung des Beweises von Satz 3.9 bewiesen. Für die andere Richtung betrachten wir nun im ersten Schritt einen Pfad auf einem Stack und konstruieren dazu einen Baum der die gleiche Transformation durchführt und zusätzlich den Pfad beschreibt.

Lemma 3.14. *Sei $\rho \in Paths$ ein Pfad mit $pw \xrightarrow{\rho} \bar{c}$, für $pw \in P\Gamma^*, \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, dann existiert ein Baum $\langle \tau \rangle \in Hedges^1$ mit $\rho \in \psi(\langle \tau \rangle)$ und $pw \overset{\langle \tau \rangle}{\Longrightarrow} \bar{c}$.*

Beweis. Induktion über die Länge von ρ:

1. $|\rho| = 0$

   Der betrachtete Pfad ist also der leere Pfad $\rho = []$. Nach Definition gilt $pw \xrightarrow{\rho} \bar{c}$ dann, wenn $\bar{c} = pw$. Wähle $\langle \tau \rangle = \langle [] \rangle$, dann gilt $pw \overset{\langle \tau \rangle}{\Longrightarrow} \bar{c}$ und $\rho \in \psi(\langle \tau \rangle)$.

2. $|\rho| > 0$

   Dann enthält der betrachtete Pfad mindestens eine Transition und lässt sich als $\rho = [r]; \tilde{\rho}$, mit $r \in \Delta, \tilde{\rho} \in Paths$, schreiben. Unterscheide dann zwei Fälle:

   1. Fall: $r \in \Delta_1$

      Nach Definition gilt $pw \xrightarrow{\rho} \bar{c}$ für $r = p\gamma \hookrightarrow \tilde{p}\tilde{w}$, $\tilde{w} \in \Gamma^*, |\tilde{w}| \in \{0, 1, 2\}$, wenn $w = \gamma\bar{w}$, für $\bar{w} \in \Gamma^*$ und $\tilde{p}\tilde{w}\bar{w} \xrightarrow{\tilde{\rho}} \bar{c}$. Dann existiert nach Induktionsvoraussetzung für den kürzeren Pfad $\tilde{\rho}$ ein Baum $\langle \tilde{\tau} \rangle \in Hedges^1$ mit $\tilde{p}\tilde{w}\bar{w} \overset{\langle \tilde{\tau} \rangle}{\Longrightarrow} \bar{c}$ und $\tilde{\rho} \in \psi(\langle \tilde{\tau} \rangle)$. Wähle dann $\langle \tau \rangle = \langle [r](\tilde{\tau}) \rangle$. Nach Definition des Übergangs für Hecken und der Voraussetzung gilt dann $pw \overset{\langle \tau \rangle}{\Longrightarrow} \bar{c}$ und $\rho \in \psi(\langle \tau \rangle) = [r]; \psi(\langle \tilde{\tau} \rangle)$.

   2. Fall: $r \in \Delta_2$

      Nach Definition gilt $pw \xrightarrow{\rho} \bar{c}$ für $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}$, wenn $w = \gamma\bar{w}$, für $\bar{w} \in \Gamma^*$, und $\hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}\bar{w} \xrightarrow{\tilde{\rho}} \bar{c}$. Nach Lemma 3.10 existieren dann Pfade $\bar{\rho}, \hat{\rho} \in Paths$ und Teilkonfigurationen $\hat{c}, \tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$ mit $\tilde{p}\tilde{\gamma}\bar{w} \xrightarrow{\bar{\rho}} \tilde{c}$ und $\hat{p}\hat{\gamma} \xrightarrow{\hat{\rho}} \hat{c}$. Dabei gilt $\tilde{\rho} \in \hat{\rho} \parallel \bar{\rho}$

und $c = \hat{c}\#\tilde{c}$. Dann existieren nach Induktionsvoraussetzung für diese Teilpfade Bäume $\langle\hat{\tau}\rangle, \langle\tilde{\tau}\rangle \in Hedges^1$ mit $\tilde{p}\tilde{\gamma}\bar{w} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$ und $\bar{\rho} \in \psi(\langle\tilde{\tau}\rangle)$, sowie $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$ und $\hat{\rho} \in \psi(\langle\hat{\tau}\rangle)$. Wähle dann $\langle\tau\rangle = \langle[r](\hat{\tau}, \tilde{\tau})\rangle$. Nach Definition des Übergangs für Hecken und der Voraussetzung gilt dann $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ und $\rho \in \psi(\langle\tau\rangle) = [r]; (\psi(\langle\hat{\tau}\rangle) \parallel \psi(\langle\tilde{\tau}\rangle))$.

□

Auch hier erweitern wir dieses Ergebnis nun auf die Konstruktion einer Hecke, die einen Pfad auf einer Konfiguration mit beliebig vielen Stacks nachahmt und diesen zusätzlich beschreibt.

Lemma 3.15. *Sei $\rho \in Paths$ ein Pfad mit $c \xrightarrow{\rho} \bar{c}$, $c = p_1w_1\# \ldots \#p_nw_n, p_iw_i \in P\Gamma^*, \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, dann existiert eine Hecke $\sigma \in Hedges$ mit $\rho \in \psi(\sigma)$ und $c \overset{\sigma}{\Longrightarrow} \bar{c}$*

Beweis. Nach Lemma 3.10 existieren Pfade $\rho_1, \ldots, \rho_n \in Paths$ und $\bar{c}_1, \ldots, \bar{c}_n \in P\Gamma^*(\#P\Gamma^*)^*$ mit $p_iw_i \xrightarrow{\rho_i} \bar{c}_i$ und $\bar{c} = \bar{c}_i\# \ldots \#\bar{c}_n$, sowie $\rho \in \rho_1 \parallel \ldots \parallel \rho_n$. Nach Lemma 3.14 existieren dann aber Bäume $\langle\tau_1\rangle, \ldots, \langle\tau_n\rangle \in Hedges^1$ für die einzelnen Pfade mit $p_iw_i \overset{\langle\tau_i\rangle}{\Longrightarrow} \bar{c}_i$ und $\rho_i \in \psi(\langle\tau_i\rangle)$. Dann gilt für $\sigma = \langle\tau_1, \ldots, \tau_n\rangle$ nach Definition des Übergangs für Hecken auch $c \overset{\sigma}{\Longrightarrow} \bar{c}$ und $\rho \in \psi(\sigma) = \psi(\langle\tau_1\rangle) \parallel \ldots \parallel \psi(\langle\tau_n\rangle)$. □

Wir haben also gezeigt, dass man für jeden erreichenden Pfad eine erreichende Hecke konstruieren kann, die diesen Pfad beschreibt. Der Beweis von Satz 3.9 gestaltet sich dann wie folgt:

Beweis. zu Satz 3.9. Zeige $\subseteq$ und $\supseteq$ getrennt:

(i) $\subseteq$

Sei $\sigma \in Hedges(c, C)$ und $\rho \in \psi(\sigma)$, dann existiert $\bar{c} \in C$ mit $c \overset{\sigma}{\Longrightarrow} \bar{c}$. Nach Lemma 3.13 folgt $c \xrightarrow{\rho} \bar{c}$ und damit $\rho \in Paths(c, C)$.

(ii) $\supseteq$

Sei $\rho \in Paths(c, C)$ dann existiert $\bar{c} \in C$ mit $c \xrightarrow{\rho} \bar{c}$. Nach Lemma 3.15 existiert dann σ mit $c \overset{\sigma}{\Longrightarrow} \bar{c}$, also $\sigma \in Hedges(c, C)$, und $\rho \in \psi(\sigma)$. Damit gilt dann aber auch $\rho \in \psi(Hedges(c, C))$.

□

Man kann also durch die Menge der erreichenden Hecken zu einer Konfiguration c und einer Menge von Konfigurationen C auch die Menge der erreichenden Pfade beschreiben. Daher können wir uns nun der Beschreibung der erreichenden Hecken durch die Klassen von Popsequenzen zuwenden. Wir beweisen dazu in einem ersten Schritt, dass zwischen den Klassen der Popsequenzen und den Transitionen des, durch den Algorithmus in Definition 2.26 aus dem Automaten $\mathcal{A}$ berechneten, Automaten $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$ ein analoger Zusammenhang wie bei WPDS besteht.

Lemma 3.16. *Seien $s, \tilde{s} \in S_s$, $\gamma \in \Gamma$, dann gilt $(s, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$ genau dann, wenn $L(s, \gamma, \tilde{s}) \neq \emptyset$.*

Die Menge der Transitionen im saturierten Automaten beschreibt genau die Menge der nicht leeren Klassen von Popsequenzen. Dieses Ergebnis erlaubt es dann, sich bei den im nächsten Abschnitt folgenden Aussagen, auf diese Klassen von Popsequenzen zu konzentrieren.

Für den Beweis betrachtet man zunächst eine Reihe von weiteren Hilfslemmata. Als erstes stellt man fest, dass man bei einem gegebenen Baum, der auf einem Stack eine gewisse Transformation durchführt, diesen Stack nach unten beliebig vergrößern kann und diese neuen Stackelemente durch den Baum nicht verändert werden. Dies ermöglicht es Transformationen mit bestimmten Eigenschaften, die ein bestimmtes Aussehen der obersten Stackelemente voraussetzen, auf beliebige Stacks anzuwenden, wobei lediglich die obersten Stackelemente die Voraussetzung erfüllen müssen.

Lemma 3.17. *Sei $pw \in P\Gamma^*, \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$ und $\langle\tau\rangle \in Hedges^1$, dann gilt $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ genau dann, wenn $pw\bar{w} \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}\bar{w}$ für alle $\bar{w} \in \Gamma^*$*

Beweis. Zeige $\Leftarrow$ und $\Rightarrow$ getrennt:

(i) $\Leftarrow$

Für $\bar{w} = \epsilon$ folgt die Behauptung.

(ii) $\Rightarrow$

Strukturelle Induktion über den Aufbau von τ:

1. $\tau = []$

   Nach Definition gilt $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\bar{c} = pw$. Dann gilt aber auch schon die Behauptung, denn dann gilt auch $\bar{c}\bar{w} = pw\bar{w}$.
2. $\tau = [r](\tilde{\tau})$

   Nach Definition gilt $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ für $r = p\gamma \hookrightarrow \tilde{p}\tilde{w} \in \Delta_1$, $\tilde{w} \in \Gamma^*, |\tilde{w}| \in \{0, 1, 2\}$, dann, wenn $w = \gamma\hat{w}$, für $\hat{w} \in \Gamma^*$, und $\tilde{p}\tilde{w}\hat{w} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}$. Nach Induktionsvoraussetzung gilt dann $\tilde{p}\tilde{w}\hat{w}\bar{w} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}\bar{w}$ für alle $\bar{w} \in \Gamma^*$. Nach Definition des Übergangs durch Bäume gilt damit auch $pw\bar{w} \overset{\langle[r](\tilde{\tau})\rangle}{\Longrightarrow} \bar{c}\bar{w}$, denn $pw\bar{w} = p\gamma\hat{w}\bar{w}$.
3. $\tau = [r](\hat{\tau}, \tilde{\tau})$

   Nach Definition gilt $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ für $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta_2$ dann, wenn $w = \gamma\hat{w}$, mit $\hat{w} \in \Gamma^*$, und $\bar{c} = \hat{c}\#\tilde{c}$, für $\hat{c}, \tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$, mit $\tilde{p}\tilde{\gamma}\hat{w} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$, sowie $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$. Nach Induktionsvoraussetzung gilt dann für den rechten Teilbaum $\tilde{p}\tilde{\gamma}\hat{w}\bar{w} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}\bar{w}$ für alle $\bar{w} \in \Gamma^*$. Zusammen mit dem Übergang durch den linken Teilbaum gilt dann nach Definition des Übergangs durch Bäume auch $pw\bar{w} \overset{\langle[r](\hat{\tau},\tilde{\tau})\rangle}{\Longrightarrow} \bar{c}\bar{w}$, denn $pw\bar{w} = \gamma\hat{w}\bar{w}$.

□

Als zweites betrachten wir die Auswirkungen der Konkatenation zweier Bäume die auf einem Stack arbeiten. Der erste Baum bearbeitet dabei den Stack zuerst und erzeugt gegebenenfalls neue Stacks und führt auf diesen Transformationen aus. Auch der Hauptstack wird in einen neuen Zustand transformiert. Der zweite Baum arbeitet nur auf diesem Hauptstack weiter, die Konkatenation verlängert den äußersten rechten Ast, und erzeugt gegebenfalls wieder neue Stacks. Der Baum der aus dem Zusammenfügen der beiden Teile entsteht transformiert dann also den anfänglichen Stack in eine Konfiguration in der zuerst die erzeugten Stacks des ersten Baums, dann die erzeugten Stacks der zweiten Baums und dann der, erst vom ersten und dann vom zweiten Baum transformierte, Hauptstack stehen. Wenn also die Transformationen zweier Bäume bekannt sind und sie so zusammenpassen, dass der Hauptstack nach Bearbeitung der ersten Hecke die passende Gestalt für den zweiten Baum hat, können wir die Auswirkungen des zusammengefügten Baums beschreiben.

Lemma 3.18. *Wenn* $pw \overset{\langle\tau_1\rangle}{\Longrightarrow} \bar{v}\tilde{p}\tilde{w}$*, für* $pw \in P\Gamma^*, \bar{v} \in (P\Gamma^*\#)^*, \tilde{p}\tilde{w} \in P\Gamma^*, \langle\tau_1\rangle \in Hedges^1$*, und* $\tilde{p}\tilde{w} \overset{\langle\tau_2\rangle}{\Longrightarrow} \bar{c}$*, für* $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*, \langle\tau_2\rangle \in Hedges^1$*, dann gilt* $pw \overset{\langle\tau_1\rangle;\langle\tau_2\rangle}{\Longrightarrow} \bar{v}\bar{c}$*.*

Beweis. Strukturelle Induktion über den Aufbau von τ_1:

1. $\tau_1 = []$

   Nach Definition gilt $pw \overset{\langle\tau_1\rangle}{\Longrightarrow} \bar{v}\tilde{p}\tilde{w}$ dann, wenn $\bar{v} = \epsilon$ und $\tilde{p}\tilde{w} = pw$. Außerdem gilt $\langle\tau_1\rangle;\langle\tau_2\rangle = \langle\tau_1;\tau_2\rangle = \langle\tau_2\rangle$ und damit $pw \overset{\langle\tau_1\rangle;\langle\tau_2\rangle}{\Longrightarrow} \bar{v}\bar{c}$.
2. $\tau_1 = [r](\tilde{\tau}_1)$

   Nach Definition gilt $pw \overset{\langle\tau_1\rangle}{\Longrightarrow} \bar{v}\tilde{p}\tilde{w}$ dann für $r = p\gamma \hookrightarrow \bar{p}\bar{w} \in \Delta_1$, $\bar{w} \in \Gamma^*, |\bar{w}| \in \{0,1,2\}$, wenn $w = \gamma\check{w}$, für $\check{w} \in \Gamma^*$, und $\bar{p}\bar{w}\check{w} \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \bar{v}\tilde{p}\tilde{w}$. Nach Induktionsvoraussetzung gilt dann für den Teilbaum $\bar{p}\bar{w}\check{w} \overset{\langle\tilde{\tau}_1\rangle;\langle\tau_2\rangle}{\Longrightarrow} \bar{v}\bar{c}$. Nach Voraussetzung und Definition des Übergangs durch Bäume gilt damit auch $pw \overset{\langle[r](\tilde{\tau}_1;\tau_2)\rangle}{\Longrightarrow} \bar{v}\bar{c}$. Dies ist die Behauptung, da $\langle[r](\tilde{\tau}_1;\tau_2)\rangle = \langle[r](\tilde{\tau}_1)\rangle;\langle\tau_2\rangle = \langle\tau_1\rangle;\langle\tau_2\rangle$.
3. $\tau_1 = [r](\hat{\tau}_1, \tilde{\tau}_1)$

   Nach Definition gilt $pw \overset{\langle\tau_1\rangle}{\Longrightarrow} \bar{v}\tilde{p}\tilde{w}$ dann für $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta_2$, wenn $w = \gamma\bar{w}$, mit $\bar{w} \in \Gamma^*$, und $\tilde{p}\tilde{\gamma}\bar{w} \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \tilde{v}\tilde{p}\tilde{w}$, mit $\tilde{v} \in (P\Gamma^*\#)^*$, sowie $\hat{p}\hat{w} \overset{\langle\hat{\tau}_1\rangle}{\Longrightarrow} \hat{c}$, für $\hat{c} \in P\Gamma^*(\#P\Gamma^*)^*$, mit $\bar{v} = \hat{c}\#\tilde{v}$. Nach Induktionsvoraussetzung gilt dann für den rechten Teilbaum $\tilde{p}\tilde{\gamma}\bar{w} \overset{\langle\tilde{\tau}_1\rangle;\langle\tau_2\rangle}{\Longrightarrow} \tilde{v}\bar{c}$. Zusammen mit dem Übergang durch den linken Teilbaum folgt dann nach Definition insgesamt auch $pw \overset{\langle[r](\hat{\tau}_1,\tilde{\tau}_1;\tau_2)\rangle}{\Longrightarrow} \bar{v}\bar{c}$, da $\bar{v} = \hat{c}\#\tilde{v}$. Dies ist die Behauptung, da $\langle[r](\hat{\tau}_1,\tilde{\tau}_1;\tau_2)\rangle = \langle[r](\tilde{\tau}_1,\hat{\tau}_1)\rangle;\langle\tau_2\rangle = \langle\tau_1\rangle;\langle\tau_2\rangle$.

□

Als drittes betrachten wir verschiedene Phasen in der Ausführung eines Baums der einen Stack mit mindestens einen Symbol in eine Konfiguration überführt, in der noch mindestens

ein Symbol auf dem Hauptstack liegt. Man die Ausführung dann in verschiedenen Phase aufteilen. In einer erste Phase werden nacheinander Symbole vom Hauptstack entfernt. Während dieser Phase kann der Stack zwischenzeitlich auch wachsen, am Ende liegen jedoch weniger Symbole auf dem Stack. Dazu werden gegebenfalls auch neue Threads erzeugt. Wenn dann irgendwann ein tiefster Punkt auf dem Stack erreicht ist, werden in der nächsten Phase wieder Symbole auf den Stack gelegt. Dabei kann auch hier der Stack zwischendrin wieder verkleinert werden, jedoch nie tiefer als der schon erreichte Tiefpunkt. Man kann den anfänglichen Stack also in verschiedene Bereiche einteilen, die von den jeweiligen Phasen betroffen sind. Der erste Teil wird von der Popphase komplett entfernt. Der zweite Teil besteht aus einem einzelnen Symbol, das zum Tiefpunkt oben auf dem Stack liegt, und dann in der Pushphase in eine größere Konfiguration transformiert wird. Alle darunter liegenden Symbole werden in keiner Phase der Ausführung verändert. Diese Aufteilung liefert das folgende Lemma:

Lemma 3.19. *Es gilt* $p\gamma w \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$*, mit* $p \in P, \gamma, \bar{\gamma} \in \Gamma, w \in \Gamma^*, \langle\tau\rangle \in Hedges^1, \bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$*, genau dann, wenn es existieren* $\langle\tau_1\rangle, \langle\tau_2\rangle \in Hedges^1$*, mit* $\langle\tau\rangle = \langle\tau_1\rangle;\langle\tau_2\rangle$*, und* $\check{v} \in (P\Gamma^*\#)^*, \check{c} \in P\Gamma^*(\#P\Gamma^*)^*, \check{\gamma} \in \Gamma$*,* $w_1, w_2 \in \Gamma^*$*,* $\gamma_1 \in \Gamma$*, mit* $\gamma w = w_1\gamma_1 w_2$ *und* $\bar{c}\bar{\gamma} = \check{v}\check{c}\check{\gamma}w_2$*, sowie* $p_1 \in P$*, so dass* $pw_1 \overset{\langle\tau_1\rangle}{\Longrightarrow} \check{v}p_1$ *und* $p_1\gamma_1 \overset{\langle\tau_2\rangle}{\Longrightarrow} \check{c}\check{\gamma}$*.*

Beweis. Zeige $\Leftarrow$ und $\Rightarrow$ getrennt:

(i) $\Rightarrow$

Folgt direkt aus Lemma 3.18.

(ii) $\Leftarrow$

Zeige die Aussage durch Strukturelle Induktion über den Aufbau von τ. Abbildung 3.2 enthält für die einzelnen während der Induktion betrachten Fälle eine veranschaulichende Darstellung. Dabei ist auf der linken Seite jeweils die Ausgangssituation nach Anwendung der Induktionsvoraussetzung dargestellt und auf der rechten Seite die daraus konstruierte Einteilung der Phasen. Die einzelnen Bereiche der Konfiguration für die Phasen des Baumes, sind durch Linien getrennt.

1. $\tau = []$

   Dann gilt $p\gamma w \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$, wenn $\bar{c}\bar{\gamma} = p\gamma w$. Es finden keine Übergänge statt, demnach wird kein Stacksymbol entfernt oder modifziert. Es werden auch keine neuen Stacks erzeugt. Die Situation ist in Abbildung 3.2a dargestellt. Wähle $\langle\tau_1\rangle = \langle\tau_2\rangle = \langle[]\rangle$, $\check{v} = \epsilon, \check{c} = p$, $\check{\gamma} = \gamma$, $w_1 = \epsilon, w_2 = w$, $p_1 = p$, $\gamma_1 = \gamma$, dann gilt die Behauptung.
2. $\tau = [r](\tilde{\tau})$

   Unterscheide drei Fälle für $r \in \Delta$:

   1. Fall: $r = p\gamma \hookrightarrow \tilde{p}$

      Es gilt $p\gamma w \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$, wenn $\tilde{p}w \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$. Die erste Transition entfernt also das oberste Stackelement. Die Situation ist in Abbildung 3.2b dargestellt. Da in der Zielkonfiguration

noch mindestens ein Stacksymbol $\bar{\gamma}$ existiert, gilt auch $|w| > 0$. Nach Induktionsvoraussetzung existieren dann für den restlichen Teilbaum eine Pop- und eine Pushphase $\langle\tilde{\tau}_1\rangle, \langle\tilde{\tau}_2\rangle \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_1\rangle;\langle\tilde{\tau}_2\rangle$ und $\breve{v} \in (P\Gamma^*\#)^*$, $\breve{c} \in P\Gamma^*(\#P\Gamma^*)^*$, $\breve{\gamma} \in \Gamma$, $\tilde{w}_1, \tilde{w}_2 \in \Gamma^*$, $\tilde{p}_1 \in P$, $\tilde{\gamma}_1 \in \Gamma$ mit $w = \tilde{w}_1\tilde{\gamma}_1\tilde{w}_2$ und $\bar{c}\bar{\gamma} = \breve{v}\breve{c}\breve{\gamma}\tilde{w}_2$ und $\tilde{p}\tilde{w}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \breve{v}\tilde{p}_1$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_2\rangle}{\Longrightarrow} \breve{c}\breve{\gamma}$. Durch Erweiterung der Popphase des restlichen Teilbaumes um die ausgeführte Transition, sowie hinzufügen des durch die Transition entfernten Symbols zum Wort der entfernten Symbole erhält man die Aufteilung des gesamten Baumes. Wähle dazu $\langle\tau_1\rangle = \langle[r](\tilde{\tau}_1)\rangle$, $\langle\tau_2\rangle = \langle\tilde{\tau}_2\rangle$, $\breve{v} = \breve{v}, \breve{c} = \breve{c}$, $\breve{\gamma} = \breve{\gamma}$, $w_1 = \gamma\tilde{w}_1, w_2 = \tilde{w}_2, p_1 = \tilde{p}_1, \gamma_1 = \tilde{\gamma}_1$, dann gilt die Behauptung.

2. Fall: $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}$

Es gilt $p\gamma w \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$, wenn $\tilde{p}\tilde{\gamma}w \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$. Die erste Transition transformiert also nur das oberste Stacksymbol. Nach Induktionsvoraussetzung existieren dann für den restlichen Teilbaum eine Pop- und eine Pushphase $\langle\tilde{\tau}_1\rangle, \langle\tilde{\tau}_2\rangle \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_1\rangle;\langle\tilde{\tau}_2\rangle$ und $\breve{v} \in (P\Gamma^*\#)^*$, $\breve{c} \in P\Gamma^*(\#P\Gamma^*)^*$, $\breve{\gamma} \in \Gamma$, $\tilde{w}_1, \tilde{w}_2 \in \Gamma^*$, $\tilde{p}_1 \in P$, $\tilde{\gamma}_1 \in \Gamma$ mit $\tilde{\gamma}w = \tilde{w}_1\tilde{\gamma}_1\tilde{w}_2$ und $\bar{c}\bar{\gamma} = \breve{v}\breve{c}\breve{\gamma}\tilde{w}_2$ und $\tilde{p}\tilde{w}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \breve{v}\tilde{p}_1$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_2\rangle}{\Longrightarrow} \breve{c}\breve{\gamma}$. Zur Bestimmung der Pop- und Pushphasen für den gesamten Baum muss man dann zwei Fälle unterscheiden:

i. Fall: $|\tilde{w}_1| = 0$

Es gilt $\tilde{w}_1 = \epsilon$ und die Popphase ist der leere Baum, $\langle\tilde{\tau}_1\rangle = \langle[]\rangle$, da auf dem leeren Stack $\tilde{p}\tilde{w}_1$ keine Transition ausgeführt werden können. Damit gilt dann auch $\breve{v} = \epsilon$, da in der Popphase keine neuen Stacks erzeugt werden können. Da die Pushphase also direkt auf dem aktuellen Stack beginnt gilt $\tilde{w}_2 = w$, $\tilde{p}_1 = \tilde{p}$, $\tilde{\gamma}_1 = \tilde{\gamma}$. Die Situation ist in Abbildung 3.2c dargestellt. Da die erste Transition des gesamten Baumes das oberste Stacksymbol nur verändert und im folgenden keine Popphase stattfindet, stellt diese zusammen mit der direkt folgenden Pushphase des Teilbaumes, die Pushphase des gesamten Baumes dar. Die Popphase des gesamten Baumes bleibt der leere Baum. Wähle also $\langle\tau_1\rangle = \langle[]\rangle, \langle\tau_2\rangle = \langle[r](\tilde{\tau}_2)\rangle$, $\breve{v} = \breve{v} = \epsilon, \breve{c} = \breve{c}$, da durch die Transition in der verlängerten Pushphase keine zusätzlichen Stacks erzeugt werden, $\breve{\gamma} = \breve{\gamma}$, $w_1 = \tilde{w}_1 = \epsilon, w_2 = \tilde{w}_2$. Zudem muss der Startpunkt der Pushphase $p_1 = p, \gamma_1 = \gamma$ auf die Startbedingungen der ersten Transition geändert werden. Dann gilt die Behauptung.

ii. Fall: $|\tilde{w}_1| > 0$

Es gilt also $\tilde{w}_1 = \tilde{\gamma}\tilde{w}$, mit $\tilde{w} \in \Gamma^*$, und demnach wird in der Popphase des folgenden Teilbaums das gerade transformierte oberste Stacksymbol irgendwann entfernt. Die Situation ist in Abbildung 3.2d dargestellt. Die Popphase des gesamten Baumes ergibt sich dann durch Kombination der, zur Transformation angewandten ersten Transition, und der Popphase des Teilbaumes. Die Pushphase des gesamten Baumes entspricht der Pushphase des Teilbaums. Wähle also $\langle\tau_1\rangle = \langle[r](\tilde{\tau}_1)\rangle, \langle\tau_2\rangle = \langle\tilde{\tau}_2\rangle$, $\breve{v} = \breve{v}$, da durch die zusätzliche Transition in der Popphase keine Stacks erzeugt werden, $\breve{c} = \breve{c}$, $\breve{\gamma} = \breve{\gamma}$, $w_1 = \gamma\tilde{w}$, da die Popphase nun vor der ersten Transition, und damit auf dem untransformierten obersten Stacksymbol, beginnt, $w_2 = \tilde{w}_2$, $p_1 = \tilde{p}_1$, $\gamma_1 = \tilde{\gamma}_1$. Dann gilt die Behauptung.

3. Fall: $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}\hat{\gamma}$

Es gilt $p\gamma w \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$, wenn $\tilde{p}\tilde{\gamma}\hat{\gamma}w \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$. Die erste Transition transformiert das aktuelle oberste Stackelement und legt ein zusätzliches Symbol auf den Stack. Nach Induktionsvoraussetzung existieren dann für den restlichen Teilbaum, der jetzt auf dem vergrößerten Stack operiert, eine Pop- und eine Pushphase $\langle\tilde{\tau}_1\rangle, \langle\tilde{\tau}_2\rangle \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_1\rangle;\langle\tilde{\tau}_2\rangle$ und $\breve{v} \in (P\Gamma^*\#)^*$, $\breve{c} \in P\Gamma^*(\#P\Gamma^*)^*$, $\breve{\gamma} \in \Gamma$, $\tilde{w}_1, \tilde{w}_2 \in \Gamma^*$, $\tilde{p}_1 \in P$, $\tilde{\gamma}_1 \in \Gamma$ mit $\tilde{\gamma}\hat{\gamma}w = \tilde{w}_1\tilde{\gamma}_1\tilde{w}_2$ und $\bar{c}\bar{\gamma} = \breve{v}\breve{c}\breve{\gamma}\tilde{w}_2$ und $\tilde{p}\tilde{w}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \breve{v}\tilde{p}_1$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_2\rangle}{\Longrightarrow} \breve{c}\breve{\gamma}$. Zur Bestimmung der Pop- und Pushphasen für den gesamten Baum muss man dann drei Fälle unterscheiden:

i. Fall: $|\tilde{w}_1| = 0$

Es folgt $\tilde{w}_1 = \epsilon$ und damit $\tilde{\tau}_1 = []$, da auf einem leeren Stack keine Transitionen ausgeführt werden können. Es findet also keine Popphase statt und es startet direkt die Pushphase. Desweiteren gilt dann auch $\breve{v} = \epsilon$, da in der Popphase keine Stacks erzeugt werden, $\tilde{w}_2 = \hat{\gamma}w$, da von der Pushphase nur das oberste Stackelement verändert wird, $\tilde{p}_1 = \tilde{p}$, $\tilde{\gamma}_1 = \tilde{\gamma}$, da die Pushphase direkt in der aktuellen Konfiguration beginnt. Die Situation ist in Abbildung 3.2e dargestellt. Die Popphase des gesamten Baumes ist wiederum der leere Baum und die aktuelle Transition wird zur Pushphase gezählt. Diese startet dann direkt auf dem obersten Stacksymbol der Anfangskonfiguration und wird dann mit der Pushphase des restlichen Baumes auf dem erhöhten Stack fortgesetzt. Die von der gesamten Pushphase erzeugte Konfiguration entspricht dann der von der Pushphase des restlichen Baumes, aus dem neuen obersten Stacksymbol, erzeugten Konfiguration und dem transformierten ursprünglichen Stacksymbol. Der nicht angetastete Teil des Stacks beginnt unter diesem Symbol. Wähle also $\langle\tau_1\rangle = \langle[]\rangle, \langle\tau_2\rangle = \langle[r](\tilde{\tau}_2)\rangle$, $\breve{v} = \breve{v} = \epsilon, \breve{c} = \breve{c}\breve{\gamma}$, $\breve{\gamma} = \hat{\gamma}$, $w_1 = \tilde{w}_1 = \epsilon, w_2 = w$, $p_1 = p$, $\gamma_1 = \gamma$, dann gilt die Behauptung.

ii. Fall: $|\tilde{w}_1| = 1$

In diesem Fall gilt $\tilde{w}_1 = \tilde{\gamma}$. Nur das gerade auf den Stack gelegte neue Stacksymbol wird in der folgenden Popphase wieder entfernt. Die erste Transition bildet zusammen mit der Popphase des restlichen Baums einen Pfad der auf der gleichen Ebene des Stacks anfängt und endet. Es gilt also $\tilde{w}_2 = w$ und $\tilde{\gamma}_1 = \hat{\gamma}$. Von der Anfangskonfiguration ausgehend wird dann durch die ersten Transition, in Kombination mit der Popphase des Teilbaumes, nur das erste Symbol transformiert und es findet keine wirkliche Popphase statt. Die Situation ist in Abbildung 3.2f dargestellt. Diese Kombination gehört also zur Pushphase des kompletten Baumes und die Popphase bleibt leer. Wähle daher $\langle\tau_1\rangle = \langle[]\rangle, \langle\tau_2\rangle = \langle[r](\tilde{\tau}_1;\tilde{\tau}_2)\rangle$, $\breve{v} = \epsilon$, da in der nun leeren Popphase keine Stacks erzeugt werden, $\breve{c} = \breve{v}\breve{c}$, da die in der ursprünglichen Popphase erzeugten Stacks nun in der Pushphase erzeugt werden, $\breve{\gamma} = \breve{\gamma}$, $w_1 = \epsilon$, es gibt keine Popphase mehr, $w_2 = \tilde{w}_2$, $p_1 = p$, $\gamma_1 = \gamma$, dann gilt die Behauptung.

iii. Fall: $|\tilde{w}_1| > 1$

Es folgt $\tilde{w}_1 = \tilde{\gamma}\hat{\gamma}\tilde{w}$ mit $\tilde{w} \in \Gamma^*$. In der anschließenden Popphase des Teilbaums werden also mehr Elemente vom Stack entfernt, als durch die erste Transition hinzugefügt wurden. Die Situation ist in Abbildung 3.2g dargestellt. Insgesamt zählt die erste Transition damit zur Popphase des gesamten Baumes. Die Pushphase wird vom Teilbaum übernommen. Wähle also $\langle\tau_1\rangle = \langle[r](\tilde{\tau}_1)\rangle, \langle\tau_2 = \tilde{\tau}_2\rangle$, $\breve{v} = \breve{v}$, da keine neuen Stacks in

der verlängerten Popphase erzeugt werden, $\check{c} = \check{\check{c}}$, $\check{\gamma} = \check{\check{\gamma}}$, $w_1 = \gamma\tilde{w}, w_2 = \tilde{w}_2, p_1 = \tilde{p}_1$, $\gamma_1 = \tilde{\gamma}_1$, dann gilt die Behauptung.

3. $\tau = [r](\hat{\tau}, \tilde{\tau})$

Dann gilt $p\gamma w \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}\bar{\gamma}$ für $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}$, wenn $\tilde{p}\tilde{\gamma}w \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}\bar{\gamma}$ und $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$ mit $\bar{c}\bar{\gamma} = \hat{c}\#\tilde{c}\bar{\gamma}$. Die erste Transition erzeugt also einen neuen Stack, verhält sich aber sonst wie eine Regel die auf nur das oberste Stacksymbol austauscht. Nach Induktionsvoraussetzung existieren dann für den rechten Teilbaum eine Pop- und eine Pushphase $\langle\tilde{\tau}_1\rangle, \langle\tilde{\tau}_2\rangle \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_1\rangle;\langle\tilde{\tau}_2\rangle$ und $\check{v} \in (P\Gamma^*\#)^*$, $\check{c} \in P\Gamma^*(\#P\Gamma^*)^*$, $\check{\gamma} \in \Gamma$, $\tilde{w}_1, \tilde{w}_2 \in \Gamma^*$, $\tilde{p}_1 \in P$, $\tilde{\gamma}_1 \in \Gamma$ mit $\tilde{\gamma}w = \tilde{w}_1\tilde{\gamma}_1\tilde{w}_2$ und $\tilde{c}\bar{\gamma} = \check{v}\check{c}\check{\gamma}\tilde{w}_2$ und $\tilde{p}\tilde{w}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \check{v}\tilde{p}_1$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_2\rangle}{\Longrightarrow} \check{c}\check{\gamma}$. Zur Bestimmung der Pop- und Pushphasen für den gesamten Baum muss man dann zwei Fälle unterscheiden:

1. Fall: $|\tilde{w}_1| = 0$
   Dann folgt $\tilde{w}_1 = \epsilon$ und damit $\langle\tilde{\tau}_1\rangle = \langle[]\rangle$, da auf einem leeren Stack keine Transitionen ausgeführt werden können. Der rechte Teilbaum führt also keine Popphase aus. Weiter gilt dann $\check{v} = \epsilon$, $\tilde{w}_2 = w$, $\tilde{p}_1 = \tilde{p}$, $\tilde{\gamma}_1 = \tilde{\gamma}$. Die Situation ist in Abbildung 3.2h dargestellt. Die Popphase des gesamten Baumes ist dann auch leer und die Transition, und damit auch der linke Teilbaum, gehört zur direkt beginnenden Pushphase. Wähle $\langle\tau_1\rangle = \langle[]\rangle, \langle\tau_2\rangle = \langle[r](\hat{\tau}, \tilde{\tau}_2)\rangle$, $\check{v} = \check{\check{v}} = \epsilon, \check{c} = \hat{c}\#\check{\check{c}}$, da der durch die Transition erzeugte Thread in der Pushphase transformiert wird, $\check{\gamma} = \check{\check{\gamma}}$, $w_1 = \tilde{w}_1, w_2 = \tilde{w}_2$, $p_1 = p$, $\gamma_1 = \gamma$, dann gilt die Behauptung.
2. Fall: $|\tilde{w}_1| > 0$
   Dann folgt $\tilde{w}_1 = \tilde{\gamma}\tilde{w}$ mit $\tilde{w} \in \Gamma^*$. Das transformierte Symbol auf dem Stack wird also in der Popphase des rechten Teilbaums entfernt. Die Situation ist in Abbildung 3.2i dargestellt. Daher zählt die Transition, und damit auch der linke Teilbaum, zur Popphase des gesamten Baumes. Wähle also $\langle\tau_1\rangle = \langle[r](\hat{\tau}, \tilde{\tau}_1)\rangle, \langle\tau_2\rangle = \tilde{\tau}_2$, $\check{v} = \hat{c}\#\check{v}$, da der erzeugte Stack nun in der Popphase transformiert wird, $\check{c} = \check{\check{c}}$, $\check{\gamma} = \check{\check{\gamma}}$, $w_1 = \gamma\tilde{w}, w_2 = \tilde{w}_2$, $p_1 = \tilde{p}_1$, $\gamma_1 = \tilde{\gamma}_1$, dann gilt die Behauptung.

□

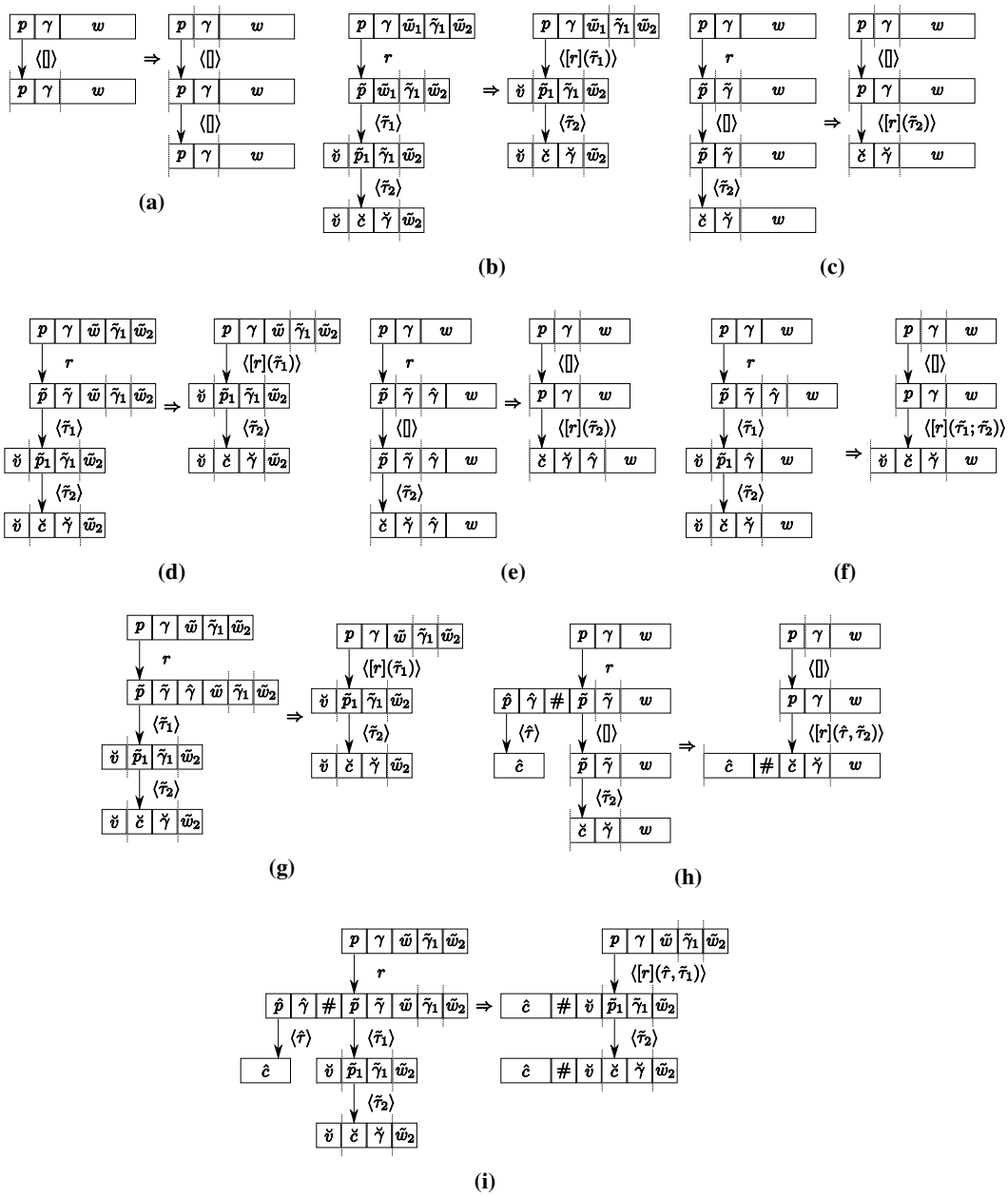


Abbildung 3.2: Aufteilung der Ausführung eines Baumes in Pop- und Pushphase

Wir können also zu einem Stack und einem Baum einen Teil des Stacks angeben, der auf jeden Fall gelöscht wird, sowie einen Teil der während der gesamten Ausführung nicht verändert wird, und ein Symbol, dass zum Zeitpunkt der Ausführung die den tiefsten Punkt markiert als oberstes auf dem Stack liegt. Weiter liefert uns das folgende Lemma nun, dass wir die Phase in der der Stack verkleinert wird noch weiter aufteilen können, indem man für

jedes einzelne Symbol, das entfernt wird, den Teil des Baumes bestimmt, der diesen Vorgang durchführt.

Lemma 3.20. *Es gilt* $p_1\gamma_1 \dots \gamma_{n-1} \overset{\langle\tau\rangle}{\Longrightarrow} vp_n$*, für* $p_1, p_n \in P, \gamma_1, \dots, \gamma_{n-1} \in \Gamma, n > 1, \langle\tau\rangle \in Hedges^1$*, genau dann, wenn es existieren* $\langle\tau_1\rangle, \dots, \langle\tau_{n-1}\rangle \in Hedges^1$*, mit* $\langle\tau\rangle = \langle\tau_1\rangle; \dots; \langle\tau_{n-1}\rangle$*, und* $v_1, \dots, v_{n-1} \in (P\Gamma^*\#)^*$*,* $p_2, \dots, p_{n-1} \in P$*, mit* $v = v_1 \dots v_{n-1}$ *und* $p_i\gamma_i \overset{\langle\tau_i\rangle}{\Longrightarrow} v_i p_{i+1}$.

Beweis. Zeige $\Leftarrow$ und $\Rightarrow$ getrennt:

(i) $\Rightarrow$

Folgt direkt aus Lemma 3.18.

(ii) $\Leftarrow$

Zeige die Aussage durch Strukturelle Induktion über den Aufbau von τ. Abbildung 3.3 enthält für die einzelnen während der Induktion betrachten Fälle eine veranschaulichende Darstellung. Dabei ist auf der linken Seite jeweils die Ausgangssituation nach Anwendung der Induktionsvoraussetzung dargestellt und auf der rechten Seite die daraus konstruierte Aufteilung der Phasen. Die einzelnen Bereiche der Konfiguration die für die jeweiligen die Phasen des Baumes relevant sind, sind durch Linien getrennt.

1. $\tau = []$

   Dieser Fall kann nicht auftreten, da $p_1\gamma_1 \dots \gamma_{n-1} \overset{\langle\tau\rangle}{\Longrightarrow} vp_n$ nur gilt, wenn $v = \epsilon$ und $n = 1$.

2. $\tau = [r](\tilde{\tau})$

   Unterscheide drei Fälle für $r \in \Delta$:

   1. Fall: $r = p_1\gamma_1 \hookrightarrow \tilde{p}_2$

      Es gilt $p_1\gamma_1 \dots \gamma_{n-1} \overset{\langle\tau\rangle}{\Longrightarrow} vp_n$, wenn $\tilde{p}_2\gamma_2 \dots \gamma_{n-1} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} vp_n$. Die erste Transition sorgt also schon für das entfernen des obersten Stacksymbols. Unterscheide dann zwei Fälle:

      i. Fall: $n = 2$

      In diesem Fall ist der Stack nach ausführen der Transition komplett geleert. Die Situation ist in Abbildung 3.3a dargestellt. Wähle $\langle\tau_1\rangle = \langle[r]([])\rangle$ und $v_1 = \epsilon$, es werden keine neuen Stacks erzeugt, dann gilt die Behauptung.

      ii. Fall: $n > 2$

      Nach entfernen des obersten Stackelementes durch die Transition, müssen noch weitere Elemente entfernt werden. Die Situation ist in Abbildung 3.3b dargestellt. Nach Induktionsvoraussetzung existieren dann für den restlichen Teilbaum Popoperationen $\tilde{\tau}_2, \dots, \tilde{\tau}_{n-1} \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_2\rangle; \dots; \langle\tilde{\tau}_{n-1}\rangle$ und $\tilde{v}_2, \dots, \tilde{v}_{n-1} \in (P\Gamma^*\#)^*$, $\tilde{p}_3, \dots, \tilde{p}_{n-1} \in P$ mit $v = \tilde{v}_2 \dots \tilde{v}_{n-1}$ und $\tilde{p}_i\gamma_i \overset{\langle\tilde{\tau}_i\rangle}{\Longrightarrow} \tilde{v}_i\tilde{p}_{i+1}$ für $i > 1$, dabei sei $\tilde{p}_n = p_n$. Insgesamt besteht dann die Popoperation für das oberste Symbol nur aus der ersten Transition und die Popoperationen für die restlichen Symbole aus den Popoperationen des Teilbaumes. Wähle also $\langle\tau_1\rangle = \langle[r]([])\rangle, \langle\tau_2\rangle = \langle\tilde{\tau}_2\rangle, \dots, \langle\tau_{n-1}\rangle = \langle\tilde{\tau}_{n-1}\rangle$

und $v_1 = \epsilon$, es werden keine neuen Stacks erzeugt, $v_2 = \tilde{v}_2, \ldots, v_{n-1} = \tilde{v}_{n-1}, p_2 = \tilde{p}_2, \ldots, p_{n-1} = \tilde{p}_{n-1}$, dann folgt die Behauptung.

2. Fall: $r = p_1\gamma_1 \hookrightarrow \tilde{p}_1\tilde{\gamma}_1$

Es gilt $p_1\gamma_1 \ldots \gamma_{n-1} \overset{\langle\tau\rangle}{\Longrightarrow} vp_n$, wenn $\tilde{p}_1\tilde{\gamma}_1\gamma_2 \ldots \gamma_{n-1} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} vp_n$. Die erste Transition verändert also das oberste Stacksymbol nur, entfernt dieses aber noch nicht. Die Situation ist in Abbildung 3.3c dargestellt. Nach Induktionsvoraussetzung existieren dann jedoch für den restlichen Teilbaum Popoperationen $\langle\tilde{\tau}_1\rangle, \ldots, \langle\tilde{\tau}_{n-1}\rangle \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_1\rangle; \ldots; \langle\tilde{\tau}_{n-1}\rangle$ und $\tilde{v}_1, \ldots, \tilde{v}_{n-1} \in (P\Gamma^*\#)^*$, $\tilde{p}_2, \ldots, \tilde{p}_{n-1} \in P$ mit $v = \tilde{v}_1 \ldots \tilde{v}_{n-1}$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \tilde{v}_1\tilde{p}_2$ und $\tilde{p}_i\gamma_i \overset{\langle\tilde{\tau}_i\rangle}{\Longrightarrow} \tilde{v}_i\tilde{p_{i+1}}$ für $i \in \{2, \ldots, n-1\}$. Man erhält nun eine Popoperationen für das oberste Stacksymbol γ_1 der ursprünglichen Startkonfiguration, indem man die Popoperation für $\tilde{\gamma}_1$ mit der ersten Transition kombiniert. Die Popoperationen für die übrigen Symbole können übernommen werden. Wähle also $\langle\tau_1\rangle = \langle[r](\tilde{\tau}_1)\rangle, \langle\tau_2\rangle = \langle\tilde{\tau}_2\rangle, \ldots, \langle\tau_{n-1}\rangle = \langle\tilde{\tau}_{n-1}\rangle$ und $v_1 = \tilde{v_1}, \ldots, v_{n-1} = \tilde{v_{n-1}}$, $p_2 = \tilde{p}_2, \ldots, p_{n-1} = \tilde{p}_{n-1}$, dann folgt die Behauptung.

3. Fall: $r = p_1\gamma_1 \hookrightarrow \tilde{p}_1\tilde{\gamma}_1\hat{\gamma}_1$

Es gilt $p_1\gamma_1 \ldots \gamma_{n-1} \overset{\langle\tau\rangle}{\Longrightarrow} vp_n$, wenn $\tilde{p}_1\tilde{\gamma}_1\hat{\gamma}_1\gamma_2 \ldots \gamma_{n-1} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} vp_n$. Durch die erste Transition wird also vorübergehend der Stack sogar noch um ein Element vergrößert. Die Situation ist in Abbildung 3.3d dargestellt. Nach Induktionsvoraussetzung existieren dann aber für den Teilbaum Popoperationen $\langle\tilde{\tau}_1\rangle, \langle\hat{\tau}_1\rangle, \langle\tilde{\tau}_2\rangle, \ldots, \langle\tilde{\tau}_{n-1}\rangle \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_1\rangle; \langle\hat{\tau}_1\rangle; \langle\tilde{\tau}_2\rangle; \ldots; \langle\tilde{\tau}_{n-1}\rangle$ und $\tilde{v}_1, \hat{v}_1, \tilde{v}_2, \ldots, \tilde{v}_{n-1} \in (P\Gamma^*\#)^*$, $\hat{p}_1, \tilde{p}_2, \ldots, \tilde{p}_{n-1} \in P$ mit $v = \tilde{v}_1\hat{v}_1\tilde{v}_2 \ldots \tilde{v}_{n-1}$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \tilde{v}_1\hat{p}_1$, sowie $\hat{p}_1\hat{\gamma}_1 \overset{\langle\hat{\tau}_1\rangle}{\Longrightarrow} \hat{v}_1\tilde{p}_2$ und $\tilde{p}_i\gamma_i \overset{\langle\tilde{\tau}_i\rangle}{\Longrightarrow} \tilde{v}_i\tilde{p}_{i+1}$ für $i \in \{2, \ldots, n-1\}$. Man erhält dann eine Popoperation für das ursprüngliche oberste Stacksymbol γ, indem man die erste Transition und dann die Popoperationen für die beiden von ihr auf den Stack gelegten Symbole nacheinander ausführt. Die Popoperationen für die übrigen Stacksymbole können übernommen werden. Wähle also $\langle\tau_1\rangle = \langle[r](\tilde{\tau}_1; \hat{\tau}_1)\rangle, \langle\tau_2\rangle = \langle\tilde{\tau}_2\rangle, \ldots, \langle\tau_{n-1}\rangle = \langle\tilde{\tau}_{n-1}\rangle$ und $v_1 = \tilde{v}_1\hat{v}_1, \ldots, v_{n-1} = \tilde{v}_{n-1}$, $p_2 = \tilde{p}_2, \ldots, p_{n-1} = \tilde{p}_{n-1}$, dann folgt die Behauptung.

3. $\tau = [r](\hat{\tau}, \tilde{\tau})$

Für $r = p_1\gamma_1 \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}_1\tilde{\gamma}_1$ gilt $p_1\gamma_1 \ldots \gamma_{n-1} \overset{\langle\tau\rangle}{\Longrightarrow} vp_n$, wenn $v = \hat{c}\#\tilde{v}$, mit $\tilde{v} \in (P\Gamma^*\#)^*, \hat{c} \in P\Gamma^*(\#P\Gamma^*)^*$ und $\tilde{p}_1\tilde{\gamma}_1\gamma_2 \ldots \gamma_{n-1} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{v}p_n$ und $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$. Auf dem ursprünglichen Stack verhält sich die Transition wie eine Transition die lediglich das oberste Stacksymbol verändert. Die Situation ist in Abbildung 3.3e dargestellt. Nach Induktionsvoraussetzung existieren dann für den rechten Teilbaum Popoperationen $\langle\tilde{\tau}_1\rangle, \ldots, \langle\tilde{\tau}_{n-1}\rangle \in Hedges^1$ mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau}_1\rangle; \ldots; \langle\tilde{\tau}_{n-1}\rangle$ und $\tilde{v}_1, \ldots, \tilde{v}_{n-1} \in (P\Gamma^*\#)^*$, $\tilde{p}_2, \ldots, \tilde{p}_{n-1} \in P$ mit $\tilde{v} = \tilde{v}_1 \ldots \tilde{v}_{n-1}$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \tilde{v}_1\tilde{p}_2$ und $\tilde{p}_i\gamma_i \overset{\langle\tilde{\tau}_i\rangle}{\Longrightarrow} \tilde{v}_i\tilde{p}_{i+1}$ für $i \in \{2, \ldots, n\}$. Man erhält dann eine Popoperation für das Stacksymbol γ_1, indem man den linken Teilbaum mit der ersten Transition und der Popoperation für $\tilde{\gamma}_1$ des rechten Teilbaums kombiniert. Die Popoperationen für die anderen Stacksymbole können übernommen werden. Wähle also $\langle\tau_1\rangle = \langle[r](\hat{\tau}, \tilde{\tau}_1)\rangle, \langle\tau_2\rangle = \langle\tilde{\tau}_2\rangle, \ldots, \langle\tau_{n-1}\rangle = \langle\tilde{\tau}_{n-1}\rangle$ und $v_1 = \hat{c}\#\tilde{v}_1, \ldots, v_{n-1} = \tilde{v}_{n-1}, p_2 = \tilde{p}_2, \ldots, p_{n-1} = \tilde{p}_{n-1}$, dann folgt die Behauptung.

□

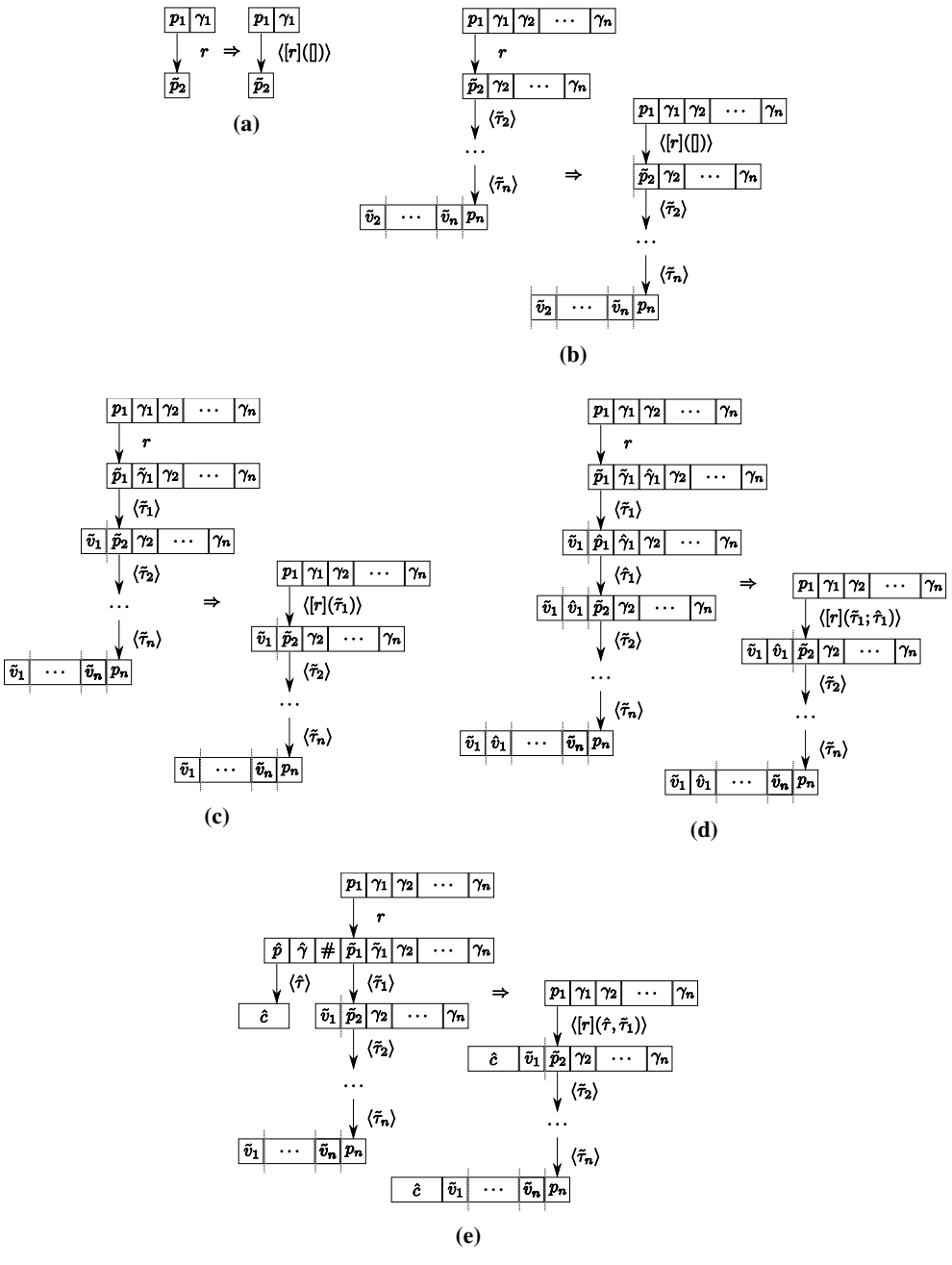


Abbildung 3.3: Aufteilung der Ausführung einer Popphase in Popoperationen

Da wir in Lemma 3.16 einen Zusammenhang zwischen der Existenz von Popsequenzen, also der Existenz von Bäumen im DPN und zugehörigen Läufen im Automaten, und der Existenz von Transitionen im saturierten Automaten herstellen wollen, betrachten wir nun noch folgendes Lemma, dass uns Informationen über den Aufbau eines Laufs in einem Automaten liefert.

Lemma 3.21. *Für einen Lauf $\phi \in Runs_{\mathcal{A}^*}$ mit $\phi^w = w\lambda$ und $w \in \Sigma^*, \lambda \in \Sigma$ gilt:*

1. *für $\lambda = p \in P$, dass $\phi = \tilde{\phi}; [\![s, p, s_p]\!]$ mit $s \in S_c, s_p \in S_p$, also $(s, p, s_p) \in \bar{\delta}_{state}$, und $\tilde{\phi} \in Runs_{\mathcal{A}^*}(\phi^+, w, s)$.*
2. *für $\lambda = \gamma \in \Gamma$, dass $\phi = \tilde{\phi}; [\![s, \gamma, \tilde{s}]\!]$ mit $s, \tilde{s} \in S_s$, also $(s, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, und $\tilde{\phi} \in Runs_{\mathcal{A}^*}(\phi^+, w, s)$. Dabei gilt für $s \in S_p$, dass $\tilde{s} \in S_s$. Für $s \notin S_p$ gilt auch $\tilde{s} \notin S_p$.*
3. *für $\lambda = \#$, dass dass $\phi = \tilde{\phi}; [\![s, \#, \tilde{s}]\!]$ mit $s \in S_s, \tilde{s} \in S_c$, also $(s, \#, \tilde{s}) \in \bar{\delta}_{separator}$, und $\tilde{\phi} \in Runs_{\mathcal{A}^*}(\phi^+, w, s)$.*
4. *für $\phi^- = \tilde{s}_p \in S_p$, dass $\phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!]$ mit $\lambda \in \{p\} \cup \Gamma, s \in S$, also $(s, \lambda, \tilde{s}) \in \bar{\delta}_{state} \cup \bar{\delta}_{stack}$, und $\tilde{\phi} \in Runs_{\mathcal{A}^*}(\phi^+, w, s)$.*
5. *für $\phi^- \in S_s \setminus S_p$, dass $\phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!]$ mit $\lambda \in \Gamma, s \in S_s$, also $(s, \lambda, \tilde{s}) \in \bar{\delta}_{stack}$, und $\tilde{\phi} \in Runs_{\mathcal{A}^*}(\phi^+, w, s)$.*
6. *für $\phi^- \in S_c$, dass $\phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!]$ mit $\lambda = \#, s \in S_s$, also $(s, \#, \tilde{s}) \in \bar{\delta}_{separator}$, und $\tilde{\phi} \in Runs_{\mathcal{A}^*}(\phi^+, w, s)$.*

Für einen Lauf $\phi \in Runs_{\mathcal{A}}$, gelten in den gleichen Situationen die gleichen Aussagen für δ. Lediglich für 2. gilt immer $\tilde{s} \notin S_p$ und für 4. immer $\lambda = p$ und damit $s = \tilde{s} \in S_c$ und $(\tilde{s}, p, \tilde{s}_p) \in \delta_{state}$.

Beweis. Sei $t \in \bar{\delta}$, die letzte von ϕ durchlaufene Kante.

zu 1.: Dann gilt $t = (s, p, \tilde{s})$ für $s, \tilde{s} \in S$. Nach Definition von $\mathcal{M}^*$-Automaten gilt dann $t \in \bar{\delta}_{state}$, und damit $s \in S_c$ und $\tilde{s} = s_p \in S_p$.

zu 2.: Dann gilt $t = (s, \gamma, \tilde{s})$ für $s, \tilde{s} \in S$. Nach Definition von $\mathcal{M}^*$-Automaten gilt dann $t \in \bar{\delta}_{stack}$ und damit $s, \tilde{s} \in S_s$. Da $\bar{\delta}_{stack} \subseteq (S_p \times \Gamma \times S_p) \cup (S_s \times \Gamma \times (S_s \setminus S_p))$, gilt für $s \in S_p$, dass $\tilde{s} \in S_p \cup (S_s \setminus S_p) = S_s$. Für $s \notin S_p$ gilt $\tilde{s} \in S_s \setminus S_p$.

Für $\mathcal{M}$-Automaten gilt $\delta_{stack} \subseteq S_s \times \Gamma \times (S_s \setminus S_p)$ und damit $\tilde{s} \notin S_p$.

zu 3.: Dann gilt $t = (s, \#, \tilde{s})$ für $s, \tilde{s} \in S$. Nach Definition von $\mathcal{M}^*$-Automaten gilt dann $t \in \bar{\delta}_{separator}$ und damit $s \in S_s$ und $\tilde{s} \in S_c$.

zu 4.: Dann gilt $t = (s, \lambda, \tilde{s}_p)$ für $s \in S, \tilde{s}_p \in S_p$. Nach Definition von $\mathcal{M}^*$-Automaten gilt dann $t \in\in \bar{\delta}_{state} \cup \bar{\delta}_{stack}$ und damit $\lambda \in \{p\} \cup \Gamma$ und $s \in S$.

Für $\mathcal{M}$-Automaten gilt $\delta_{stack} \subseteq S_s \times \Gamma \times (S_s \setminus S_p)$ und damit $t \in \delta_{state}$, $\lambda = p$ und $s = \tilde{s} \in S_c$.

zu 5.: Dann gilt $t = (s, \lambda, \tilde{s})$ für $s \in S, \tilde{s} \in S_s \setminus S_p$. Nach Definition von $\mathcal{M}^*$-Automaten gilt dann $t \in \bar{\delta}_{stack}$ und damit $\lambda \in \Gamma$ und $s \in S_s$.

zu 6.: Dann gilt $t = (s, \lambda, \tilde{s})$ für $s \in S, \tilde{s} \in S_c$. Nach Definition von $\mathcal{M}^*$-Automaten gilt dann $t \in \bar{\delta}_{separator}$ und damit $\lambda = \#$ und $s \in S_s$.

□

Mit diesen Werkzeugen betrachten wir nun den Beweis von Lemma 3.16.

Beweis. von Lemma 3.16:

Zeige $\Leftarrow$ und $\Rightarrow$ getrennt:

(i) $\Leftarrow$

Wir müssen also zeigen, dass zu einer beliebigen Popsequenz einer Klasse eine Kante im saturierten Automaten existiert, die diese beschreibt. Unterscheide zwei Fälle:

1. Fall: $s \notin S_p$

   Sei also $\langle \tau \rangle \in L(s, \gamma, \tilde{s})$. Dann gilt nach Definition schon $\langle \tau \rangle = \langle [] \rangle$ und $(s, \gamma, \tilde{s}) \in \delta_{stack}$. Nach (INIT)-Regel des Algorithmus gilt dann aber auch $(s, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$. Es existiert also eine Kante, die diese leere Psuedopopsequenz beschreibt.

2. Fall: $s \in S_p$

   Dann gilt $s = \bar{s}_p$, für $\bar{s} \in S_c, p \in P$. Sei $\langle \tau \rangle \in L(s, \gamma, \tilde{s})$, mit $p\gamma \overset{\langle \tau \rangle}{\Longrightarrow} \bar{c}$, für $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, und $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$. Zeige die Aussage durch strukturelle Induktion über den Aufbau von τ:

   1. $\tau = []$

      Nach Definition gilt $p\gamma \overset{\langle \tau \rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\bar{c} = p\gamma$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$ gilt, folgt dann $(\bar{s}_p, \gamma, \tilde{s}) \in \delta_{stack}$ und damit nach der (INIT)-Regel des Algorithmus auch $(\bar{s}_p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$. Es existiert also eine Kante, die diese leere Popsequenz beschreibt.

   2. $\tau = [r](\tilde{\tau})$

      Unterscheide drei Fälle für $r \in \Delta_1$:

      i. Fall: $r = p\gamma \hookrightarrow \tilde{p}$

      Nach Definition gilt $p\gamma \overset{\langle \tau \rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\tilde{p} \overset{\langle \tilde{\tau} \rangle}{\Longrightarrow} \bar{c}$. Daraus folgt aber schon $\langle \tilde{\tau} \rangle = \langle [] \rangle$, da auf einem leeren Stack keine Transitionen ausgeführt werden können, und damit $\bar{c} = \tilde{p}$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$ gilt, existiert die Kante $(\bar{s}, \tilde{p}, \tilde{s}) \in \delta_{state}$, und es gilt, nach Definition eines $\mathcal{M}$-Automaten, $\tilde{s} = \bar{s}_{\tilde{p}}$. Nach der (RETURN)-Regel des Algorithmus existiert dann eine Kante $(\bar{s}_p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, die diese Popsequenz, die nur aus der Anwendung der einen Transition besteht, beschreibt.

      ii. Fall: $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}$

      Nach Definition gilt $p\gamma \overset{\langle \tau \rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\tilde{p}\tilde{\gamma} \overset{\langle \tilde{\tau} \rangle}{\Longrightarrow} \bar{c}$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$ gilt, folgt nach Induktionsvoraussetzung, dass für die Popsequenz $\tilde{\tau}$ dann eine Kante $(\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}$ existiert, die diese beschreibt. Dann folgt aus der (STEP)-Regel des Algorithmus auch die Existenz einer Kante $(\bar{s}_p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, die die gesamte Popsequenz beschreibt.

iii. Fall: $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2$

Nach Definition gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}$. Unterscheide dann zwei Situationen:

a. Fall: $\bar{c} = \check{c}\check{\gamma}$ für $\check{c} \in P\Gamma^*(\#P\Gamma^*)^*, \check{\gamma} \in \Gamma$

Dann befinden wir uns in der Situation von Lemma 3.19 und können den Baum $\langle\tilde{\tau}\rangle$ entsprechend zerlegen. Es existieren also Pop- und Pushphase $\langle\tilde{\tau_1}\rangle, \langle\tilde{\tau_2}\rangle \in Hedges^1$, mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau_1}\rangle;\langle\tilde{\tau_2}\rangle$, und $\check{v} \in (P\Gamma^*\#)^*, \check{c} \in P\Gamma^*(\#P\Gamma^*)^*, \check{\gamma} \in \Gamma, w_1, w_2 \in \Gamma^*, \gamma_1 \in \Gamma$, mit $\bar{c} = \check{c}\check{\gamma} = \check{v}\check{c}\check{\gamma}w_2$ und $\tilde{\gamma}_1\tilde{\gamma}_2 = w_1\gamma_1w_2$, sowie $p_1 \in P$, so dass $\tilde{p}w_1 \overset{\langle\tilde{\tau_1}\rangle}{\Longrightarrow} \check{v}p_1$ und $p_1\gamma_1 \overset{\langle\tilde{\tau_2}\rangle}{\Longrightarrow} \check{c}\check{\gamma}$. Für γ_1 unterscheidet man dann zwei Fälle:

I. Fall: $\gamma_1 = \tilde{\gamma}_1$

Dann gilt $w_1 = \epsilon$ und damit $\langle\tilde{\tau_1}\rangle = \langle[]\rangle$, da auf einem leeren Stack keine Transition ausgeführt werden können, und $\check{v} = \epsilon$, da in der leeren Popphase keine neuen Stacks erzeugt werden. Der Baum besteht also nur aus einer Pushphase, in der nur das oberste Stacksymbol transformiert wird. Desweiteren gilt $w_2 = \tilde{\gamma}_2$ und damit $\bar{c} = \check{c}\check{\gamma} = \check{c}\check{\gamma}\tilde{\gamma}_2$. Da $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$, existiert in $\mathcal{A}$ ein Lauf, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Diesen kann in zwei Teile aufspalten. Der erste Teil erkennt $\check{c}\check{\gamma}$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21, gilt $\hat{s} \in S_s \setminus S_p$. Es gilt also $Runs_{\mathcal{A}}(\bar{s}, \check{c}\check{\gamma}, \hat{s}) \neq \emptyset$. Dann existiert nach Induktionsvoraussetzung eine Kante $(\bar{s}_{\tilde{p}}, \tilde{\gamma}_1, \hat{s}) \in \bar{\delta}_{stack}$, die die Popsequenz $\langle\tilde{\tau}_2\rangle$ beschreibt. Der zweite Teil des Laufes besteht nur aus der Kante $(\hat{s}, \tilde{\gamma}_2, \tilde{s}) \in \delta_{stack}$, die $\tilde{\gamma}_2$ von $\hat{s}$ nach $\tilde{s}$ erkennt. Nach (INIT)-Regel des Algorithmus gilt dann aber auch schon $(\hat{s}, \tilde{\gamma}_2, \tilde{s}) \in \bar{\delta}_{stack}$. Nach (CALL)-Regel des Algorithmus existiert dann auch die Kante $(\bar{s}, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, die die gesamte Popsequenz beschreibt.

II. Fall: $\gamma_1 = \tilde{\gamma}_2$

Dann gilt $w_1 = \tilde{\gamma}_1$. Der Baum besteht also aus einer Popphase, in der das oberste Stacksymbol entfernt wird, und einer Pushphase, in der das, dann oben liegende, zweite Stacksymbol transformiert wird. Es gilt $w_2 = \epsilon$ und damit $\bar{c} = \check{c}\check{\gamma} = \check{v}\check{c}\check{\gamma}$. Da $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$ existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Dieser kann in zwei Teile aufspalten werden. Der erste Teil erkennt $\check{v}$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21, gilt $\hat{s} \in S_c$. Für $\check{v} = \epsilon$ gilt $\hat{s} = \bar{s}$. Dann existiert, da für jeden Zustand in S_c die mit den Kontrollzuständen des DPN benannten Kanten δ_{state} existieren, aber auch ein Lauf in $\mathcal{A}$, der $\check{v}p_1$ von $\bar{s}$ nach $\hat{s}_{\tilde{p}}$ erkennt und damit gilt $Runs_{\mathcal{A}}(\bar{s}, \check{v}p_1, \hat{s}_{p_1}) \neq \emptyset$. Dann existiert nach Induktionsvoraussetzung für die Popsequenz $\langle\tilde{\tau}_1\rangle$ eine Kante $(\bar{s}_{\tilde{p}}, \tilde{\gamma}_1, \hat{s}_{p_1}) \in \bar{\delta}_{stack}$, die diese beschreibt. Der zweite Teil des Laufes erkennt $\check{c}\check{\gamma}$ von $\hat{s}$ nach $\tilde{s}$ und damit gilt $Runs_{\mathcal{A}}(\hat{s}, \check{c}\check{\gamma}, \tilde{s}) \neq \emptyset$. Dann existiert wiederum nach Induktionsvoraussetzung auch für die Popsequenz $\langle\tilde{\tau}_2\rangle$ eine Kante $(\hat{s}_{p_1}, \tilde{\gamma}_2, \tilde{s}) \in \bar{\delta}_{stack}$, die diese beschreibt. Nach (CALL)-Regel des Algorithmus existiert dann auch die Kante $(\bar{s}, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, die die gesamte Popsequenz beschreibt.

b. Fall: $\bar{c} = \hat{v}\hat{p}$ für $\hat{v} \in (P\Gamma^*\#)^*, \hat{p} \in P$

Dann befinden wir uns in der Situation von Lemma 3.20 und können den Baum $\langle\tilde{\tau}\rangle$ entsprechend zerlegen. Es existieren also $\langle\tilde{\tau_1}\rangle, \langle\tilde{\tau_2}\rangle \in Hedges^1$, mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau_1}\rangle;\langle\tilde{\tau_2}\rangle$,

und $\tilde{v}_1, \tilde{v}_2 \in (P\Gamma^*\#)^*$, mit $\hat{v} = \tilde{v}_1\tilde{v}_2$, sowie $\bar{p} \in P$, so dass $\tilde{p}\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \tilde{v}_1\bar{p}$ und $\bar{p}\tilde{\gamma}_2 \overset{\langle\tilde{\tau}_2\rangle}{\Longrightarrow} \tilde{v}_2\hat{p}$. Man kann also den Baum, der die beiden Stacksymbole $\tilde{\gamma}_1$ und $\tilde{\gamma}_2$ vom Stack entfernt, in zwei Teile aufteilen, von denen jeder eines der Symbole entfernt und dazwischen der Kontrollzustand $\bar{p}$ erreicht wird. Da zudem $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$, existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Da $\bar{c} = \hat{v}\hat{p} = \tilde{v}_1\tilde{v}_2\hat{p}$, kann man diesen Lauf in zwei Teile aufspalten. Der erste Teil erkennt $\tilde{v}_1$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21, gilt $\hat{s} \in S_c$. Für $\tilde{v}_1 = \epsilon$ gilt $\hat{s} = \bar{s}$. Dann gilt nach Existenz der mit Kontrollzuständen benannten Kanten δ_{state} für jeden Zustand in S_c, dass ein Lauf in $\mathcal{A}$ existiert, der $\tilde{v}_1\bar{p}$ von $\bar{s}$ nach $\hat{s}_{\bar{p}}$ erkennt. Damit gilt $Runs_{\mathcal{A}}(\bar{s}, \tilde{v}_1\bar{p}, \hat{s}_{\bar{p}}) \neq \emptyset$. Nach Induktionsvoraussetzung existiert dann eine Kante $(\bar{s}_{\tilde{p}}, \tilde{\gamma}_1, \hat{s}_{\bar{p}}) \in \bar{\delta}_{stack}$, die die Popsequenz $\langle\tilde{\tau}_1\rangle, \langle\tilde{\tau}_2\rangle$ beschreibt. Der zweite Teil des Laufes erkennt $\tilde{v}_2\hat{p}$ von $\hat{s}$ nach $\tilde{s}$. Nach Lemma 3.21 gilt dann $\tilde{s} = \check{s}_{\hat{p}}$, für ein $\check{s} \in S_c$. Zudem ist $Runs_{\mathcal{A}}(\hat{s}, \tilde{v}_2\hat{p}, \tilde{s}) \neq \emptyset$. Nach Induktionsvoraussetzung existiert dann auch eine Kante $(\hat{s}_{\bar{p}}, \tilde{\gamma}_2, \tilde{s}) \in \bar{\delta}_{stack}$, die die Popsequenzen $\langle\tilde{\tau}_2\rangle$ beschreibt. Nach (CALL)-Regel des Algorithmus existiert dann auch die Kante $(s_p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, die die gesamte Popsequenz beschreibt.

3. $\tau = [r](\hat{\tau}, \tilde{\tau})$

   Nach Definition gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ für $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}$ dann, wenn $\bar{c} = \hat{c}\#\tilde{c}$, für $\hat{c}, \tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$ und $\tilde{p}\tilde{\gamma} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$ und $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$ gilt, existiert also ein Lauf in $\mathcal{A}$, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Diesen kann man aufspalten in drei Teile. Der erste erkennt die erste Teilkonfiguration $\hat{c}$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21, gilt $\hat{s} \in S_s$. Damit gilt dann auch $Runs_{\mathcal{A}}(\bar{s}, \hat{c}, \hat{s}) \neq \emptyset$ und nach Induktionsvoraussetzung existiert eine Kante $(\bar{s}_{\hat{p}}, \hat{\gamma}, \hat{s}) \in \bar{\delta}_{stack}$, die die Popsequenz $\langle\hat{\tau}\rangle$ beschreibt. Der verbliebene Teil des Laufes erkennt dann $\#\tilde{c}$ von $\hat{s}$ nach $\tilde{s}$. Diesen kann man dann weiter aufspalten in einen Teil, der nur das Trennzeichen $\#$, von $\hat{s}$ zu einem $\check{s} \in S$ liest. Nach Lemma 3.21, gilt $\check{s} \in S_c$ und $(\hat{s}, \#, \check{s}) \in \delta_{separator}$. Nach Definition gilt dann aber auch $(\hat{s}, \#, \check{s}) \in \bar{\delta}_{separator}$. Der letzte Teil liest dann $\tilde{c}$ von $\check{s}$ nach $\tilde{s}$ und damit gilt $Runs_{\mathcal{A}}(\check{s}, \tilde{c}, \tilde{s}) \neq \emptyset$. Nach Induktionsvoraussetzung existiert dann eine Kante $(\check{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}$, die die Popsequenz $\langle\tilde{\tau}\rangle$ beschreibt. Dann liefert der Algorithmus nach (SPAWN)-Regel die Existenz der Kante $(\bar{s}_p, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, die die gesamte Popsequenz beschreibt.

(ii) $\Rightarrow$

Hier müssen wir nun zu jeder Transition des saturierten Automaten die Existenz einer Popsequenz zeigen, die in der durch die Transition beschriebenen Klasse liegt. Seien $\bar{\delta}^n_{stack}$ die Zwischenschritte des Algorithmus mit $\bar{\delta}^0_{stack} = \delta_{stack}$. Zeige durch Induktion über die Anzahl n der Schritte, dass die Aussage für alle Zwischenschritte gilt. Da der Algorithmus nach endlich vielen Zwischenschritte terminiert, gilt die Aussage dann auch für $\bar{\delta}_{stack}$.

1. $n = 0$

   Sei $t = (s, \gamma, \tilde{s}) \in \bar{\delta}^0_{stack} = \delta_{stack}$. Unterscheide zwei Fälle:

   1. Fall: $s \notin S_p$

      Dann gilt nach Definition schon $L(s, \gamma, \tilde{s}) = \{\langle[]\rangle\} \neq \emptyset$. In diesem Fall beschreibt die Kante also eine leere Psuedopopsequenz.

2. Fall: $s \in S_p$

Dann gilt $s = \bar{s}_p, \bar{s} \in S_c, p \in P$ und damit $Runs_{\mathcal{A}}(s, p\gamma, \tilde{s}) \neq \emptyset$. Da $p\gamma \xRightarrow{\langle\tau\rangle} p\gamma$ für $\langle\tau\rangle = \langle[]\rangle$, gilt nach Definition $\langle\tau\rangle \in L(s, \gamma, \tilde{s}) \neq \emptyset$. In diesem Fall wird durch die Kante also mindestens eine leere Popsequenz beschrieben.

2. $n > 0$

Sei $t = (s_p, \gamma, \tilde{s}$ die Transition, die im n-ten Schritt hinzugefügt wurde, also $\bar{\delta}^{n+1}_{stack} = \bar{\delta}^{n}_{stack} \cup \{t\}$. Nach der Initialisierung werden keine Kante mehr hinzugefügt, die von Zuständen, die nicht in S_p liegen, ausgehen, deshalb kann man sich hier auf Kanten beginnend in Zuständen in S_p beschränken. Unterscheide die vier Fälle die zum Hinzufügen von t geführt haben können:

1. Fall: (RETURN)

Dann folgt $r = p\gamma \hookrightarrow \tilde{p} \in \Delta_1$ und $\tilde{s} = s_{\tilde{p}}$. Für $\langle\tau\rangle = \langle[r]([])\rangle$, gilt dann $p\gamma \xRightarrow{\langle\tau\rangle} \tilde{p}$ und $Runs_{\mathcal{A}}(s, \tilde{p}, \tilde{s}) \neq \emptyset$. Nach Definition gilt dann $\langle\tau\rangle \in L(s, \gamma, \tilde{s}) \neq \emptyset$. Die Kante beschreibt also mindestens die Popsequenz, die nur aus der Anwendung einer Transition des DPN besteht, die das oberste Stackelement entfernt.

2. Fall: (STEP)

Dann folgt $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma} \in \Delta_1$ und es existiert eine Kante $(s_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}^{n}_{stack}$. Nach Induktionsvoraussetzung existiert dann bereits eine Popsequenz $\langle\tilde{\tau}\rangle \in L(s_{\tilde{p}}, \tilde{\gamma}, \tilde{s})$ mit $\tilde{p}\tilde{\gamma} \xRightarrow{\langle\tilde{\tau}\rangle} \bar{c}$, für $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, und $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$, die die Kante $(s_{\tilde{p}}, \tilde{\gamma}, \tilde{s})$ rechtfertigt. Für $\langle\tau\rangle = \langle[r](\tilde{\tau})\rangle$, gilt dann $p\gamma \xRightarrow{\langle\tau\rangle} \bar{c}$ und $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$. Damit gilt dann $\langle\tau\rangle \in L(s, \gamma, \tilde{s}) \neq \emptyset$. Die Kante beschreibt also mindestens die Popsequenz die durch die Kombination der Transition, die das Einfügen der neuen Kante bewirkt hat, mit einer Popsequenz, die die bereits existierende Kante rechtfertigt, entsteht.

3. Fall: (CALL)

Dann folgt $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \in \Delta_1$ und es existieren Kanten $(s_{\tilde{p}}, \tilde{\gamma}_1, \hat{s}), (\hat{s}, \gamma_1, \tilde{s}) \in \bar{\delta}^{n}_{stack}$. Unterscheide zwei Fälle:

i. Fall: $\hat{s} \in S_p$

Es gilt also $\hat{s} = \bar{s}_{\bar{p}}$ für ein $\bar{s} \in S_c, \bar{p} \in P$. Nach Induktionsvoraussetzung existieren dann bereits Popsequenzen $\langle\tilde{\tau}_1\rangle \in L(s_{\tilde{p}}, \tilde{\gamma}_1, \bar{s}_{\bar{p}}), \langle\tilde{\tau}_2\rangle \in L(\bar{s}_{\bar{p}}, \tilde{\gamma}_2, \tilde{s})$ mit $\tilde{p}\tilde{\gamma}_1 \xRightarrow{\langle\tilde{\tau}_1\rangle} \tilde{c}_1$, für $\tilde{c}_1 \in P\Gamma^*(\#P\Gamma^*)^*$, und $Runs_{\mathcal{A}}(s, \tilde{c}_1, \bar{s}_{\bar{p}}) \neq \emptyset$, sowie $\bar{p}\tilde{\gamma}_2 \xRightarrow{\langle\tilde{\tau}_2\rangle} \tilde{c}_2$, für $\tilde{c}_2 \in P\Gamma^*(\#P\Gamma^*)^*$, und $Runs_{\mathcal{A}}(\bar{s}, \tilde{c}_2, \tilde{s}) \neq \emptyset$, die die bereits vorhandenen Kanten rechtfertigen. Da $\hat{s} \in S_p$, besteht in $\mathcal{A}$ ein Lauf für $\tilde{c}_1$ von s nach $\hat{s}$, nach Lemma 3.21, aus einem Lauf von s nach $\bar{s}$ und einer Transition $(\bar{s}, \bar{p}, \bar{s}_p) \in \delta_{state}$. Es gilt also $\tilde{c}_1 = \tilde{v}\bar{p}$, für $\tilde{v} \in (P\Gamma^*\#)^*$, und $Runs_{\mathcal{A}}(s, \tilde{v}, \bar{s}) \neq \emptyset$. Man kann also mit der ersten Popsequenz das erste Stacksymbol $\tilde{\gamma}_1$ komplett entfernen und im Kontrollzustand $\hat{p}$ enden. Die dabei erzeugten neuen Stacks werden so transformiert, dass sie in $\mathcal{A}$ von einem Lauf vom Zustand s bis zum Zustand $\hat{s}$ erkannt werden. Mit der zweiten Popsequenz kann man dann ausgehend vom Kontrollzustand $\hat{p}$ und dem Stacksymbol $\tilde{\gamma}_2$ eine Konfiguration erreichen, die in $\mathcal{A}$ von einem Lauf vom Zustand $\hat{s}$ zum Zustand $\tilde{s}$ erkannt wird. Für $\langle\tau\rangle = \langle[r](\tilde{\tau}_1; \tilde{\tau}_2)\rangle$, gilt dann $p\gamma \xRightarrow{\langle\tau\rangle} \tilde{v}\tilde{c}_2$ nach Lemma 3.18 und durch Kombination der beiden Läufe, die die Teilkonfigurationen erkennen, erhält man

einen Lauf in $\mathcal{A}$, der $\bar{c} = \tilde{v}\tilde{c}_2$ von s nach $\tilde{s}$ erkennt. Es gilt also $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$ und damit $\langle\tau\rangle \in L(s, \gamma, \tilde{s})$.

ii. Fall: $\hat{s} \notin S_p$

Nach Induktionsvoraussetzung existiert dann eine Popsequenz $\langle\tilde{\tau}\rangle \in L(s_{\tilde{p}}, \tilde{\gamma}, \hat{s})$, mit $\tilde{p}\tilde{\gamma}_1 \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$ und $Runs(s, \tilde{c}, \hat{s}) \neq \emptyset$, die die erste vorhandene Kante rechtfertigt. Die zweite Kante kann nicht durch den Algorithmus hinzugefügt worden sein, da $\hat{s} \notin S_p$, und damit gilt $(\hat{s}, \tilde{\gamma}_2, \tilde{s}) \in \delta_{stack}$. Man kann mit der gegebenen Popsequenz das oberste Stacksymbol also in eine Konfiguration überführen, die in $\mathcal{A}$ durch einen Lauf vom Zustand s zum Zustand $\hat{s}$ erkannt wird. Wähle $\langle\tau\rangle = \langle[r](\tilde{\tau})\rangle$, dann gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \tilde{c}\tilde{\gamma}_2$. Verlängert man den Lauf für $\tilde{c}$ von s nach $\hat{s}$ in $\mathcal{A}$ um die Kante $(\hat{s}, \tilde{\gamma}_2, \tilde{s}) \in \delta_{stack}$, erhält man einen Lauf für $\bar{c} = \tilde{c}\tilde{\gamma}_2$ von s nach $\tilde{s}$. Es gilt $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$ und damit $\langle\tau\rangle \in L(s, \gamma, \tilde{s}) \neq \emptyset$.

4. Fall: (SPAWN)

Dann folgt $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta_2$ und des existieren Kanten $(s_{\hat{p}}, \hat{\gamma}, \hat{s}), (\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}^n_{stack}$ und $(\hat{s}, \#, \bar{s}) \in \bar{\delta}_{separator} = \delta_{separator}$. Nach Induktionsvoraussetzung existieren dann Popsequenzen $\langle\hat{\tau}\rangle \in L(s_{\hat{p}}, \hat{\gamma}, \hat{s}), \langle\tilde{\tau}\rangle \in L(\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s})$ mit $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$, für $\hat{c} \in P\Gamma^*(\#P\Gamma^*)^*$, und $Runs_{\mathcal{A}}(s, \hat{c}, \hat{s}) \neq \emptyset$, sowie $\tilde{p}\tilde{\gamma} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$, für $\tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$, und $Runs_{\mathcal{A}}(\bar{s}, \tilde{c}, \tilde{s}) \neq \emptyset$, die die bereits vorhandenen Kanten rechtfertigen. Für $\langle\tau\rangle = \langle[r](\hat{\tau}, \tilde{\tau})\rangle$, gilt dann $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \hat{c}\#\tilde{c}$. Nun existieren nach Induktionsvoraussetzung Läufe in $\mathcal{A}$, die $\hat{c}$ von s nach $\hat{s}$ und $\tilde{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennen. Diese kann man mit der nach Voraussetzung existierenden Kante $(\hat{s}, \#, \bar{s}) \in \delta_{separator}$ zu einem Lauf in $\mathcal{A}$ verbinden, der $\bar{c} = \hat{c}\#\tilde{c}$ von s nach $\tilde{s}$ erkennt. Dann gilt $Runs_{\mathcal{A}}(s, \hat{c}\#\tilde{c}, \tilde{s}) \neq \emptyset$ und damit $\langle\tau\rangle \in L(s, \gamma, \tilde{s}) \neq \emptyset$. Die Kante beschreibt also mindestens die Popsequenz die man durch Kombination der anfänglichen Transition mit einer Popsequenz für das Symbol auf den neu erzeugten Stack und einer Popsequenz für das transformierte Symbol auf dem ursprünglichen Stack erhält.

□

Damit ist also gezeigt, dass für die von uns definierten Klassen von Popsequenzen, in einem DPN, der gleiche Zusammenhang zum saturierten Automaten $\mathcal{A}^*$, wie für die Klassen von Popsequenzen in einem PDS, existiert. Im nächsten Abschnitt betrachten wir nun wie man diese Popsequenzen charakterisieren kann. Dazu werden wir, im Gegensatz zu den Betrachtungen in [RSJM05], keine Grammatik, sondern ein Ungleichungssystem zur Beschreibung der Klassen von Popsequenzen verwenden. Desweiteren werden wir dann zeigen, dass man mit Hilfe der Klassen von Popsequenzen alle erreichenden Hecken zu einer Konfiguration und einer Menge von Zielkonfigurationen beschreiben kann.

3.2 Charakterisierung der Pfade

Wir haben bis jetzt gezeigt, dass bei der Betrachtung erreichender Hecken eines DPN $\mathcal{M}$ von einer Konfiguration c zu einer regulären Konfigurationsmenge C, beschrieben durch einen $\mathcal{M}$-Automaten $\mathcal{A}$, Popsequenzen existieren, die sich analog zu den für WPDS definierten Popsequenzen verhalten. So lassen sich diese Popsequenzen in Klassen einteilen, die, analog zum Verhalten bei WPDS, in direktem Zusammenhang mit den Kanten in dem durch den Saturierungsalgorithmus, aus Definition 2.26, aus $\mathcal{A}$ berechneten $\mathcal{M}^*$-Automaten $\mathcal{A}^*$ stehen.

Im nächsten Schritt zeigen wir nun, dass man die Klassen von Popsequenzen in einer Weise beschreiben kann, die einen späteren Übergang zu einer abstrakteren Betrachtung in Form von Gewichten erlaubt. Dazu betrachten wir, im Hinblick auf eine spätere Verwendung abstrakter Interpretation, eine Darstellung durch ein Ungleichungssystem, dessen kleinste Lösung den Klassen entspricht. Wir weichen also vom Vorgehen für WPDS in [RSJM05] ab, und betrachten keine Grammatik zur Beschreibung der Klassen von Popsequenzen.

Damit wir mit Hilfe der, dann greifbar dargestellten, Klassen von Popsequenzen wirklich Aussagen über die erreichenden Hecken machen können, müssen wir, analog zu WPDS, noch zeigen, dass wir jede erreichende Hecke in eine Folge von Popsequenzen zerlegen können, und auch für jede Verknüpfung von Popsequenzen aus einer bestimmten Folge von Klassen eine erreichende Hecke erhalten. Dass dies möglich ist, wird jedoch durch die Lemmata 3.19 und 3.20 im vorherigen Abschnitt, die ein Aufteilung eines Baumes in verschiedene Teile, die jeweils ein Symbol eines Stacks betrachten, schon nahegelegt. Wir werden zudem zeigen, dass, wiederum analog zu WPDS, diese Zerlegung der Hecken in engem Zusammenhang mit den akzeptierenden Läufen für die Konfiguration c im Automaten $\mathcal{A}^*$ steht.

Betrachten wir nun zuerst die Beschreibung der Klassen von Popsequenzen durch ein Ungleichungssystem. In Lemma 3.16 wurde gezeigt, das die Klasse von Popsequenzen für ein Symbol γ von einem Zustand s zu einem Zustand $\tilde{s}$ genau dann nicht leer ist, wenn eine Transition $(s, \gamma, \tilde{s})$ im saturierten Automaten existiert. Genauer wird im Beweis der Vorwärtsinklusion demonstriert, wie man für jeden Fall, der zum Einfügen einer Transition führt, aus den Popsequenzen, aus den schon als nicht leer bekannten Klassen von Popsequenzen für bereits existierende Kanten, eine neue Popsequenz aus der Klasse der neuen Transition konstruieren kann. Da diese Konstruktion für jede Popsequenz aus den Klassen für die bereits existierenden Kanten möglich ist, kann man also folgern, dass die Klasse der Popsequenzen der neuen Transition zumindest alle Popsequenzen umfasst, die sich so konstruieren lassen. Dies lässt sich dann durch eine Ungleichung ausdrücken. Wenn nun für eine Transition die Ungleichungen zu allen Fällen, die diese dem Automaten hinzufügen würden, betrachtet werden, sollte die kleinste Lösung genau die Klasse von Popsequenzen beschreiben. Für die Konstruktion des Ungleichungssystems definieren wir Operatoren, die aus gegebenen Mengen von Popsequenzen neue Mengen von Popsequenzen konstruieren.

Definition 3.22. Wir definieren Operatoren:

1. $[r](M) = \{\langle [r](\tau)\rangle \mid \langle \tau \rangle \in M\}$ für $M \subseteq Hedges^1, r \in \Delta_1$
2. $[r](M_1, M_2) = \{\langle [r](\tau_1, \tau_2)\rangle \mid \langle \tau_1 \rangle \in M_1, \langle \tau_2 \rangle \in M_2\}$ für $M_1, M_2 \subseteq Hedges^1, r \in \Delta_2$

Der erste Operator verlängert eine Popsequenz um eine Anweisung, indem diese der Sequenz vorangestellt wird. Hierbei sind nur Regeln erlaubt die keine neuen Thread erzeugen. Die Idee dabei ist, dass man eine Popsequenz für einen Stack erhält, wenn man einen Schritt mit Hilfe der Anweisung macht und dann eine Popsequenz für den folgenden Zustand anwendet. Der zweite Operator nimmt zwei Popsequenzen und kombiniert diese mit einer Regel die einen neuen Thread erzeugt. Dabei wird die erste Sequenz dem neuen Thread zugeordnet und die zweite dem alten Thread. Die Idee hierbei ist, dass man eine Popsequenz für einen Stack erhält, wenn man mit der Anweisung einen neuen Thread erzeugt und dann sowohl den neuen Thread als auch den alten Thread mit einer Popsequenz für die jeweiligen Zustände bearbeitet.

Da wir bereits wissen, dass gerade solche Klassen von Popsequenzen nicht leer sind, die durch die Transitionen des saturierten Automaten beschrieben werden, konstruieren wir einen Automaten, der die Klassen der Popsequenzen direkt an den zugehörigen Transitionen speichert. Dazu definieren wir:

Definition 3.23. Ein annotierter $\mathcal{M}^*$-Automat ist ein Tripel $(\mathcal{A}^*, M, l)$, wobei $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$ ein $\mathcal{M}^*$-Automat, M eine Menge von Annotationen und $l : \bar{\delta}_{stack} \to M$ eine Annotationsfunktion ist, die jeder Transition $t \in \bar{\delta}_{stack}$ einen Wert zuweist.

Zur Konstruktion des Automaten erweitern wir nun den Saturierungsalgorithmus aus 2.3 um Schritte, die ein Ungleichungssystem erstellen. Da alle Ungleichungen zu einer Transition betrachtet werden müssen, werden dabei mehr Schritte gemacht, als im normalen Algorithmus. Dieser kann Fälle ignorieren, in denen schon existierende Kanten erneut eingefügt würden. Im erweiterten Algorithmus muss in diesem Fall noch die zugehörige Ungleichung hinzugefügt werden. Die Terminierung des Algorithmus ist jedoch immer noch gewährleistet, da nur endlich viele Transitionen existieren und daher nur endlich viele Ungleichungen die zugehörigen Klassen in Verbindung bringen können. Die Annotationen des berechneten Automaten ergeben sich dann als kleinste Lösung dieses Ungleichungssystems. Der berechnete Automat $\mathcal{A}^*$ ist identisch mit dem durch den normalen Algorithmus berechneten Automaten.

Definition 3.24. Zu einer regulären Menge $C \subseteq Conf$ von Konfigurationen eines DPN $\mathcal{M}$, gegeben durch einen $\mathcal{M}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$, definieren wir einen erweiterten Saturierungsalgorithmus zur Berechnung eines annotierten Automaten $(\mathcal{A}^*, \mathcal{P}(Hedges^1), l)$ mit $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$. Dabei gilt $\bar{\delta} = \delta_{separator} \cup \delta_{state} \cup \bar{\delta}_{stack}$ und $\bar{\delta}_{stack}$ ist die kleinste Menge, die die folgenden Bedingungen erfüllt, und $l(t) = \overline{L}[t]$ und $\overline{L}$ ist die kleinste Lösung des Ungleichungssystems L über $(\mathcal{P}(Hedges), \cup)$:

(INIT) $t = (s, \gamma, \tilde{s}) \in \delta_{stack}$, für $s, \tilde{s} \in S_s$, dann auch:

$$\begin{aligned} t &\in \bar{\delta}_{stack} \text{ und} \\ L[t] &\supseteq \{\langle[]\rangle\} \end{aligned}$$

(RETURN) $r = p\gamma \hookrightarrow \tilde{p} \in \Delta$, dann für alle $s \in S_c$ auch:

$$\begin{aligned} t = (s_p, \gamma, s_{\tilde{p}}) &\in \bar{\delta}_{stack} \text{ und} \\ L[t] &\supseteq [r](\{\langle[]\rangle\}) \end{aligned}$$

(STEP) $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma} \in \Delta$ und $\tilde{t} = (s_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}$ für $s \in S_c, \tilde{s} \in S_s$, dann auch:

$$\begin{aligned} t = (s_p, \gamma, \tilde{s}) &\in \bar{\delta}_{stack} \text{ und} \\ L[t] &\supseteq [r](L[\tilde{t}]) \end{aligned}$$

(CALL) $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \in \Delta$ und $\tilde{t}_1 = (s_{\tilde{p}}, \tilde{\gamma}_1, \tilde{s}_1), \tilde{t}_2 = (\tilde{s}_1, \tilde{\gamma}_2, \tilde{s}_2) \in \bar{\delta}_{stack}$ für $s \in S_c, \tilde{s}_1, \tilde{s}_2 \in S_s$, dann auch:

$$\begin{aligned} t = (s_p, \gamma, \tilde{s}_2) &\in \bar{\delta}_{stack} \text{ und} \\ L[t] &\supseteq [r](L[\tilde{t}_1]; L[\tilde{t}_2]) \end{aligned}$$

(SPAWN) $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta$ und $\hat{t} = (s_{\hat{p}}, \hat{\gamma}, \hat{s}), \tilde{t} = (\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}$, $(\hat{s}, \#, \bar{s}) \in \delta_{separator}$ für $s, \bar{s} \in S_c, \hat{s}, \tilde{s} \in S_s$, dann auch:

$$\begin{aligned} t = (s_p, \gamma, \tilde{s}) &\in \bar{\delta}_{stack} \text{ und} \\ L[t] &\supseteq [r](L[\hat{t}], L[\tilde{t}]) \end{aligned}$$

Da die oben definierten Operatoren zur Darstellung der rechten Seiten des Ungleichungssystem offensichtlich monoton sind, und $(\mathcal{P}(Hedges^1), \cup)$ ein vollständiger Verband ist, liefert das Fixpunkttheorem von Knaster-Tarski die Existenz einer kleinsten Lösung. Diese kann jedoch nicht ohne weiteres berechnet werden, da der Verband unendlich absteigende Ketten enthält und daher Verfahren wie die chaotische Iteration gegebenenfalls nicht terminieren.

Nun müssen wir noch zeigen, dass die kleinste Lösung des Systems tatsächlich eine korrekte und präzise Beschreibung der Klassen von Popsequenzen zu jeder Transition des Automaten liefert. Sei dazu im folgenden $C \subseteq Conf$ eine reguläre Menge von Konfigurationen eines DPN $\mathcal{M} = (P, \Gamma, \Delta)$, gegeben durch einen $\mathcal{M}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$, und sei $(\mathcal{A}^*, \mathcal{P}(Hedges^1), l)$, mit $\mathcal{A}^* = (S, \Sigma, \bar{\delta}, s^s, F)$, der durch den erweiterten Saturierungsalgorithmus berechnete annotierte Automat. Betrachte dann folgendes Lemma:

Lemma 3.25. *Es gilt $l(t) = L(s, \gamma, \tilde{s})$ für alle $t = (s, \gamma, \tilde{s}) \in \bar{\delta}_{stack}$, mit $s, \tilde{s} \in S_s, \gamma \in \Gamma$.*

Beweis. Zeige $\supseteq$ und $\subseteq$ getrennt:

(i) $\supseteq$

Wir müssen also zeigen, dass die durch das Ungleichungssystem definierten Mengen an den Kanten mindestens die Popsequenzen der jeweiligen durch die Kanten beschriebenen Klassen enthalten. Unterscheide zwei Fälle:

1. Fall: $s \notin S_p$

Sei also $\langle\tau\rangle \in L(s,\gamma,\tilde{s})$. Dann gilt nach Definition schon $\langle\tau\rangle = \langle[]\rangle$ und $(s,\gamma,\tilde{s}) \in \delta_{stack}$, da der Algorithmus keine neuen Kanten einfügt, die nicht in Zuständen in S_p starten. Nach der zur (INIT)-Regel gehörenden Ungleichung gilt dann aber auch schon $\langle[]\rangle \in l((s,\gamma,\tilde{s}))$.

2. Fall: $s \in S_p$

Dann gilt $s = \bar{s}_p$, für $\bar{s} \in S_c, p \in P$. Sei $\langle\tau\rangle \in L(s,\gamma,\tilde{s})$, mit $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$, für $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, und $Runs_{\mathcal{A}}(\bar{s},\bar{c},\tilde{s}) \neq \emptyset$. Zeige dann $\langle\tau\rangle \in l((s,\gamma,\tilde{s}))$ durch strukturelle Induktion über den Aufbau von τ:

1. $\tau = []$

Nach Definition gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\bar{c} = p\gamma$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s},\bar{c},\tilde{s}) \neq \emptyset$ gilt, folgt dann $(\bar{s}_p,\gamma,\tilde{s}) \in \delta_{stack}$ und damit nach der zur (INIT)-Regel gehörenden Ungleichung dann auch $\langle\tau\rangle = \langle[]\rangle \in l((s,\gamma,\tilde{s}))$.

2. $\tau = [r](\tilde{\tau})$

Unterscheide drei Fälle für $r \in \Delta_1$:

i. Fall: $r = p\gamma \hookrightarrow \tilde{p}$

Nach Definition gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\tilde{p} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}$. Daraus folgt aber schon $\langle\tilde{\tau}\rangle = \langle[]\rangle$, da auf einem leeren Stack keine Transitionen ausgeführt werden können, und damit $\bar{c} = \tilde{p}$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s},\bar{c},\tilde{s}) \neq \emptyset$ gilt, existiert die Kante $(\bar{s},\tilde{p},\tilde{s}) \in \delta_{state}$, und es gilt, nach Definition eines $\mathcal{M}$-Automaten, $\tilde{s} = \bar{s}_{\tilde{p}}$. Nach der zur (RETURN)-Regel des Algorithmus gehörenden Ungleichung gilt dann $\langle\tau\rangle = \langle[r]([])\rangle \in [r](\{\langle[]\rangle\}) \subseteq l((s,\gamma,\tilde{s}))$.

ii. Fall: $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}$

Nach Definition gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\tilde{p}\tilde{\gamma} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s},\bar{c},\tilde{s}) \neq \emptyset$ gilt, existiert nach Lemma 3.16 die Kante $\tilde{t} = (\bar{s}_{\tilde{p}},\tilde{\gamma},\tilde{s}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung folgt, dass die Popsequenz $\tilde{\tau} \in l(\tilde{t})$. Dann folgt aus der zur (STEP)-Regel gehörenden Ungleichung auch $\langle\tau\rangle = \langle[r](\tilde{\tau})\rangle \in [r](l(\tilde{t})) \subseteq l((s,\gamma,\tilde{s}))$.

iii. Fall: $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2$

Nach Definition gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ dann, wenn $\tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \bar{c}$. Unterscheide dann zwei Situationen:

a. Fall: $\bar{c} = \check{c}\check{\gamma}$ für $\check{c} \in P\Gamma^*(\#P\Gamma^*)^*, \check{\gamma} \in \Gamma$

Dann befinden wir uns in der Situation von Lemma 3.19 und können den Baum $\langle\tilde{\tau}\rangle$ entsprechend zerlegen. Es existieren also Pop- und Pushphase $\langle\tilde{\tau_1}\rangle, \langle\tilde{\tau_2}\rangle \in Hedges^1$, mit $\langle\tilde{\tau}\rangle = \langle\tilde{\tau_1}\rangle;\langle\tilde{\tau_2}\rangle$, und $\check{v} \in (P\Gamma^*\#)^*, \check{c} \in P\Gamma^*(\#P\Gamma^*)^*, \check{\gamma} \in \Gamma, w_1, w_2 \in \Gamma^*, \gamma_1 \in \Gamma$, mit $\bar{c} = \check{c}\check{\gamma} = \check{v}\check{c}\check{\gamma}w_2$ und $\tilde{\gamma}_1\tilde{\gamma}_2 = w_1\gamma_1 w_2$, sowie $p_1 \in P$, so dass $\tilde{p}w_1 \overset{\langle\tilde{\tau_1}\rangle}{\Longrightarrow} \check{v}p_1$ und $p_1\gamma_1 \overset{\langle\tilde{\tau_2}\rangle}{\Longrightarrow} \check{c}\check{\gamma}$. Für γ_1 unterscheidet man dann zwei Fälle:

I. Fall: $\gamma_1 = \tilde{\gamma}_1$

Dann gilt $w_1 = \epsilon$ und damit $\langle \tilde{\tau}_1 \rangle = \langle [] \rangle$, da auf einem leeren Stack keine Transition ausgeführt werden können, und $\check{v} = \epsilon$, da in der leeren Popphase keine neuen Stacks erzeugt werden. Der Baum besteht also nur aus einer Pushphase, in der nur das oberste Stacksymbol transformiert wird. Desweiteren gilt $w_2 = \tilde{\gamma}_2$ und damit $\bar{c} = \check{c}\check{\gamma} = \check{c}\check{\gamma}\tilde{\gamma}_2$. Da $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$, existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Diesen kann in zwei Teile aufspalten. Der erste Teil erkennt $\check{c}\check{\gamma}$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21, gilt $\hat{s} \in S_s \setminus S_p$. Es gilt also $Runs_{\mathcal{A}}(\bar{s}, \check{c}\check{\gamma}, \hat{s}) \neq \emptyset$. Dann existiert nach Lemma 3.16 eine Kante $\tilde{t}_1 = (\bar{s}_{\tilde{p}}, \tilde{\gamma}_1, \hat{s}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung gilt für die Popsequenz $\langle \tilde{\tau}_2 \rangle \in l(\tilde{t}_1)$. Der zweite Teil des Laufes besteht nur aus der Kante $\tilde{t}_2 = (\hat{s}, \tilde{\gamma}_2, \tilde{s}) \in \delta_{stack}$, die $\tilde{\gamma}_2$ von $\hat{s}$ nach $\tilde{s}$ erkennt. Nach (INIT)-Regel des Algorithmus und zugehöriger Ungleichung gilt dann aber auch schon $\tilde{t}_2 \in \bar{\delta}_{stack}$ und $\langle [] \rangle \in l(\tilde{t}_2)$. Nach der zur (CALL)-Regel des Algorithmus gehörenden Ungleichung gilt dann $\langle \tau \rangle = \langle [r](\tilde{\tau}_2; []) \rangle \in [r](l(\tilde{t}_1); l(\tilde{t}_2)) \subseteq l((s, \gamma, \tilde{s}))$.

II. Fall: $\gamma_1 = \tilde{\gamma}_2$

Dann gilt $w_1 = \tilde{\gamma}_1$. Der Baum besteht also aus einer Popphase, in der das oberste Stacksymbol entfernt wird, und einer Pushphase, in der das, dann oben liegende, zweite Stacksymbol transformiert wird. Es gilt $w_2 = \epsilon$ und damit $\bar{c} = \check{c}\check{\gamma} = \check{v}\check{c}\check{\gamma}$. Da $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$ existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Dieser kann in zwei Teile aufspalten werden. Der erste Teil erkennt $\check{v}$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21, gilt $\hat{s} \in S_c$. Für $\check{v} = \epsilon$ gilt $\hat{s} = \bar{s}$. Dann existiert, da für jeden Zustand in S_c die mit den Kontrollzuständen des DPN benannten Kanten δ_{state} existieren, aber auch ein Lauf in $\mathcal{A}$, der $\check{v}p_1$ von $\bar{s}$ nach $\hat{s}_{\check{p}}$ erkennt und damit gilt $Runs_{\mathcal{A}}(\bar{s}, \check{v}p_1, \hat{s}_{p_1}) \neq \emptyset$. Dann existiert nach Lemma 3.16 eine Kante $\tilde{t}_1 = (\bar{s}_{\tilde{p}}, \tilde{\gamma}_1, \hat{s}_{p_1}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung gilt für die Popsequenz $\langle \tilde{\tau}_1 \rangle \in l(\tilde{t}_1)$. Der zweite Teil des Laufes erkennt $\check{c}\check{\gamma}$ von $\hat{s}$ nach $\tilde{s}$ und damit gilt $Runs_{\mathcal{A}}(\hat{s}, \check{c}\check{\gamma}, \tilde{s}) \neq \emptyset$. Dann existiert wiederum nach Lemma 3.16 eine Kante $\tilde{t}_2 = (\hat{s}_{p_1}, \tilde{\gamma}_2, \tilde{s}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung gilt für die Popsequenz $\langle \tilde{\tau}_2 \rangle \in l(\tilde{t}_2)$. Nach der zur (CALL)-Regel des Algorithmus gehörenden Ungleichung gilt dann $\langle \tau \rangle = \langle [r](\tilde{\tau}_1; \tilde{\tau}_2) \rangle \in [r](l(\tilde{t}_1); l(\tilde{t}_2)) \subseteq l((s, \gamma, \tilde{s}))$.

b. Fall: $\bar{c} = \hat{v}\hat{p}$ für $\hat{v} \in (P\Gamma^*\#)^*, \hat{p} \in P$

Dann befinden wir uns in der Situation von Lemma 3.20 und können den Baum $\langle \tilde{\tau} \rangle$ entsprechend zerlegen. Es existieren also $\langle \tilde{\tau}_1 \rangle, \langle \tilde{\tau}_2 \rangle \in Hedges^1$, mit $\langle \tilde{\tau} \rangle = \langle \tilde{\tau}_1 \rangle; \langle \tilde{\tau}_2 \rangle$, und $\tilde{v}_1, \tilde{v}_2 \in (P\Gamma^*\#)^*$, mit $\hat{v} = \tilde{v}_1\tilde{v}_2$, sowie $\bar{p} \in P$, so dass $\tilde{p}\tilde{\gamma}_1 \overset{\langle \tilde{\tau}_1 \rangle}{\Longrightarrow} \tilde{v}_1\bar{p}$ und $\bar{p}\tilde{\gamma}_2 \overset{\langle \tilde{\tau}_2 \rangle}{\Longrightarrow} \tilde{v}_2\hat{p}$. Man kann also den Baum, der die beiden Stacksymbole $\tilde{\gamma}_1$ und $\tilde{\gamma}_2$ vom Stack entfernt, in zwei Teile aufteilen, von denen jeder eines der Symbole entfernt. Da zudem $Runs_{\mathcal{A}}(\bar{s}, \bar{c}, \tilde{s}) \neq \emptyset$ existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Da $\bar{c} = \hat{v}\hat{p} = \tilde{v}_1\tilde{v}_2\hat{p}$ kann man diesen Lauf in zwei Teile aufspalten. Der erste Teil erkennt $\tilde{v}_1$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21, gilt $\hat{s} \in S_c$. Für $\tilde{v}_1 = \epsilon$ gilt $\hat{s} = \bar{s}$. Dann gilt, nach Existenz der mit Kontrollzuständen benannten Kanten δ_{state} für jeden Zustand in S_c, dass ein Lauf in $\mathcal{A}$ existiert, der $\tilde{v}_1\bar{p}$ von $\bar{s}$ nach $\hat{s}_{\bar{p}}$ erkennt. Damit gilt $Runs_{\mathcal{A}}(\bar{s}, \tilde{v}_1\bar{p}, \hat{s}_{\bar{p}}) \neq \emptyset$. Nach Lemma 3.16 existiert dann eine Kante $\tilde{t}_1 = (\bar{s}_{\tilde{p}}, \tilde{\gamma}_1, \hat{s}_{\bar{p}}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung gilt $\langle \tilde{\tau}_1 \rangle \in l(\tilde{t}_1)$.

Der zweite Teil des Laufes erkennt $\tilde{v}_2$ von $\hat{s}$ nach $\tilde{s}$. Nach Lemma 3.21 gilt dann $\tilde{s} = \check{s}_{\hat{p}}$, für ein $\check{s} \in S_c$. Zudem gilt $Runs_{\mathcal{A}}(\hat{s}, \tilde{v}_2\hat{p}, \tilde{s}) \neq \emptyset$. Nach Lemma 3.16 existiert auch eine Kante $\tilde{t}_2 = (\hat{s}_{\bar{p}}, \tilde{\gamma}_2, \tilde{s}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung gilt $\langle\tilde{\tau}_2\rangle \in l(\tilde{t}_2)$. Nach der zur (CALL)-Regel des Algorithmus gehörenden Ungleichung gilt dann $\langle\tau\rangle = \langle[r](\tilde{\tau}_1;\tilde{\tau}_2)\rangle \in [r](l(\tilde{t}_1);l(\tilde{t}_2)) \subseteq l((s,\gamma,\tilde{s}))$.

3. $\tau = [r](\hat{\tau},\tilde{\tau})$

Nach Definition gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ für $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma}$ dann, wenn $\bar{c} = \hat{c}\#\tilde{c}$, für $\hat{c},\tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$ und $\tilde{p}\tilde{\gamma} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$ und $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$. Da zusätzlich $Runs_{\mathcal{A}}(\bar{s},\bar{c},\tilde{s}) \neq \emptyset$ gilt, existiert also ein Lauf in $\mathcal{A}$, der $\bar{c}$ von $\bar{s}$ nach $\tilde{s}$ erkennt. Diesen kann man aufspalten in drei Teile. Der erste erkennt die erste Teilkonfiguration $\hat{c}$ von $\bar{s}$ zu einem $\hat{s} \in S$. Nach Lemma 3.21 gilt $\hat{s} \in S_s$. Damit gilt dann auch $Runs_{\mathcal{A}}(\bar{s},\hat{c},\hat{s}) \neq \emptyset$. Nach Lemma 3.16 existiert eine Kante $\hat{t} = (\bar{s}_{\hat{p}},\hat{\gamma},\hat{s}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung gilt $\hat{\tau} \in l(\hat{t})$. Der verbliebene Teil des Laufes erkennt dann $\#\tilde{c}$ von $\hat{s}$ nach $\tilde{s}$. Diesen kann man dann weiter aufspalten in einen Teil, der nur das Trennzeichen $\#$ von $\hat{s}$ zu einem $\check{s} \in S$ liest. Nach Lemma 3.21, gilt $\check{s} \in S_c$ und $(\hat{s},\#,\check{s}) \in \delta_{separator}$. Dann gilt nach Definition aber auch $(\hat{s},\#,\check{s}) \in \bar{\delta}_{separator}$. Der letzte Teil liest dann $\tilde{c}$ von $\check{s}$ nach $\tilde{s}$ und damit gilt $Runs_{\mathcal{A}}(\check{s},\tilde{c},\tilde{s}) \neq \emptyset$. Nach Lemma 3.16 existiert dann auch eine Kante $\tilde{t} = (\check{s}_{\tilde{p}},\tilde{\gamma},\tilde{s}) \in \bar{\delta}_{stack}$ und nach Induktionsvoraussetzung gilt $\tilde{\tau} \in l(\tilde{t})$. Dann liefert die Ungleichung der (SPAWN)-Regel, dass $\langle\tau\rangle = [r](\hat{\tau},\tilde{\tau}) \in [r](l(\hat{t}),l(\tilde{t})) \subseteq l((s,\gamma,\tilde{s}))$.

(ii) $\Rightarrow$

Zeige dazu, dass $L(s,\gamma,\tilde{s})$ eine Lösung des Ungleichungssystems $L[t]$ für alle $t = (s,\gamma,\tilde{s}) \in \bar{\delta}_{stack}$ ist. Dann gilt für die kleinste Lösung $l(t) \subseteq L(s,\gamma,\tilde{s})$. Untersuche die fünf Typen von Ungleichungen:

1. Fall: $L(s,\gamma,\tilde{s}) \supseteq \{\langle[]\rangle\}$ für $(s,\gamma,\tilde{s}) \in \delta_{stack}$

Unterscheide zwei Fälle:

i. Fall: $s \in S_p$, d.h. $s = \bar{s}_p$ für ein $\bar{s} \in S_c, p \in P$

Es gilt $p\gamma \overset{\langle[]\rangle}{\Longrightarrow} p\gamma$ und $Runs_{\mathcal{A}}(\bar{s},p\gamma,\tilde{s}) \neq \emptyset$, da $(\bar{s}_p,\gamma,\tilde{s}) \in \delta_{stack}$. Dann gilt $\langle[]\rangle \in L(s,\gamma,\tilde{s})$ und die Ungleichung ist erfüllt.

ii. Fall: $s \notin S_p$

Dann folgt nach Definition $L(s,\gamma,\tilde{s}) = \{\langle[]\rangle\}$ und damit ist die Ungleichung erfüllt.

2. Fall: $L(s_p,\gamma,\tilde{s}) \supseteq [r](\{\langle[]\rangle\})$ für $(s_p,\gamma,s_{\tilde{p}}) \in \bar{\delta}_{stack}$, $r = p\gamma \hookrightarrow \tilde{p} \in \Delta$, $s \in S_c$

Es gilt $p\gamma \overset{\langle[r]([])\rangle}{\Longrightarrow} \tilde{p}$ und $Runs_{\mathcal{A}}(s,\tilde{p},s_{\tilde{p}}) \neq \emptyset$. Dann gilt nach Definition $\langle[r]([])\rangle \in L(s_p,\gamma,s_{\tilde{p}})$ und damit ist die Ungleichung erfüllt.

3. Fall: $L(s_p,\gamma,\tilde{s}) \supseteq [r](L(s_{\tilde{p}},\tilde{\gamma},\tilde{s}))$ für $(s_p,\gamma,\tilde{s}),(s_{\tilde{p}},\tilde{\gamma},\tilde{s}) \in \bar{\delta}_{stack}$, $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma} \in \Delta$, $s \in S_c$, $\tilde{s} \in S_s$

Nach Definition gilt $\tilde{p}\tilde{\gamma} \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ und $Runs_{\mathcal{A}}(s,\bar{c},\tilde{s}) \neq \emptyset$ für beliebiges $\langle\tau\rangle \in L(s_{\tilde{p}},\tilde{\gamma},\tilde{s})$ und passendes $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$. Dann gilt auch $p\gamma \overset{\langle[r](\tau)\rangle}{\Longrightarrow} \bar{c}$ mit $Runs_{\mathcal{A}}(s,\bar{c},\tilde{s}) \neq \emptyset$ und damit $\langle[r](\tau)\rangle \in L(s_p,\gamma,\tilde{s})$. Also ist die Ungleichung erfüllt.

4. Fall: $L(s_p,\gamma,\tilde{s}_2) \supseteq [r](L(s_{\tilde{p}},\tilde{\gamma}_1,\tilde{s}_1);L(\tilde{s}_1,\tilde{\gamma}_2,\tilde{s}_2))$ für $r = p\gamma \hookrightarrow \tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \in \Delta$, $(s_p,\gamma,\tilde{s}_2),(s_{\tilde{p}},\tilde{\gamma}_1,\tilde{s}_1),(\tilde{s}_1,\tilde{\gamma}_2,\tilde{s}_2) \in \bar{\delta}_{stack}$, $s \in S_c$, $\tilde{s}_1,\tilde{s}_2 \in S_s$

Unterscheide zwei Fälle:

i. Fall: $\tilde{s}_1 \in S_p$

Dann gilt $\tilde{s}_1 = \bar{s}_{\bar{p}}$ für ein $\bar{s} \in S_c, \bar{p} \in P$. Nach Definition gilt $\tilde{p}\tilde{\gamma}_1 \overset{\langle\tau_1\rangle}{\Longrightarrow} \tilde{c}_1$, mit $Runs_{\mathcal{A}}(s, \tilde{c}_1, \tilde{s}_1) \neq \emptyset$, für beliebiges $\langle\tau_1\rangle \in L(s_{\tilde{p}}, \tilde{\gamma}_1, \tilde{s}_1)$ und passendes $\tilde{c}_1 \in P\Gamma^*(\#P\Gamma^*)^*$. Es existiert also ein Lauf in $\mathcal{A}$, der $\tilde{c}_1$ von s nach $\bar{s}_{\bar{p}}$ erkennt. Nach Lemma 3.21 gilt $\tilde{c}_1 = \tilde{v}_1\bar{p}$, für $\tilde{v}_1 \in (P\Gamma^*\#)^*$ und der erste Teil des Laufs erkennt $\tilde{v}_1$ von s nach $\bar{s}$. Da $\tilde{s}_1 = \bar{s}_{\bar{p}} \in S_p$ gilt auch $\bar{p}\tilde{\gamma}_2 \overset{\langle\tau_2\rangle}{\Longrightarrow} \tilde{c}_2$, mit $Runs_{\mathcal{A}}(\bar{s}, \tilde{c}_2, \tilde{s}_2) \neq \emptyset$ für beliebiges $\langle\tau_2\rangle \in L(\bar{s}_{\bar{p}}, \tilde{\gamma}_2, \tilde{s}_2)$ und passendes $\tilde{c}_2 \in P\Gamma^*(\#P\Gamma^*)^*$. Durch Kombination des ersten Teils des vorher betrachteten Laufs mit einem beliebigen Lauf in $\mathcal{A}$ der $\tilde{c}_2$ erkennt, erhält man einen Lauf in $\mathcal{A}$, der $\bar{c} = \tilde{v}_1\tilde{c}_2$ von s nach $\tilde{s}_2$ erkennt und damit $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}_2) \neq \emptyset$. Nach Lemma 3.18 gilt zudem $\tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \overset{\langle\tau_1\rangle;\langle\tau_2\rangle}{\Longrightarrow} \bar{c}$. Dann gilt auch $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$, für $\langle\tau\rangle = \langle[r](\tau_1;\tau_2)\rangle$, und damit $\langle\tau\rangle \in L(s_p, \gamma, \tilde{s}_2)$. Da dies für beliebige $\langle\tau_1\rangle \in L(s_{\tilde{p}}, \tilde{\gamma}_1, \tilde{s}_1), \langle\tau_2\rangle \in L(\bar{s}_{\bar{p}}, \tilde{\gamma}_2, \tilde{s}_2)$ gilt, ist die Ungleichung erfüllt.

ii. Fall: $\tilde{s}_1 \notin S_p$

Nach Definition gilt $\tilde{p}\tilde{\gamma}_1 \overset{\langle\tau_1\rangle}{\Longrightarrow} \tilde{c}_1$, mit $Runs_{\mathcal{A}}(s, \tilde{c}_1, \tilde{s}_1) \neq \emptyset$, für beliebiges $\langle\tau_1\rangle \in L(s_{\tilde{p}}, \tilde{\gamma}_1, \tilde{s}_1)$ und passendes $\tilde{c}_1 \in P\Gamma^*(\#P\Gamma^*)^*$. Es existiert also ein Lauf in $\mathcal{A}$, der $\tilde{c}_1$ von s nach $\bar{s}_{\bar{p}}$ erkennt. Da $\tilde{s}_1 \notin S_p$, gilt schon $(\tilde{s}_1, \tilde{\gamma}_2, \tilde{s}_2) \in \delta_{stack}$, denn der Algorithmus erzeugt keine neuen Kanten, die nicht von Zuständen in S_p ausgehen. Man kann den erwähnten Lauf also um diese Kante verlängern und erhält $Runs_{\mathcal{A}}(s, \tilde{c}_1\tilde{\gamma}_2, \tilde{s}_2) \neq \emptyset$. Nach Definition gilt zudem für alle $\langle\tau_2\rangle \in L(\tilde{s}_1, \tilde{\gamma}_2, \tilde{s}_2)$, dass $\langle\tau_2\rangle = \langle[]\rangle$. Nach Lemma 3.18 gilt $\tilde{p}\tilde{\gamma}_1\tilde{\gamma}_2 \overset{\langle\tau_1\rangle;\langle\tau_2\rangle}{\Longrightarrow} \tilde{c}_1\tilde{\gamma}_2$. Dann gilt auch $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \tilde{c}_1\tilde{\gamma}_2$, für $\langle\tau\rangle = \langle[r](\tau_1;\tau_2)\rangle$, und damit $\langle\tau\rangle \in L(s_p, \gamma, \tilde{s}_2)$. Da dies für beliebige $\langle\tau_1\rangle \in L(s_{\tilde{p}}, \tilde{\gamma}_1, \tilde{s}_1), \langle\tau_2\rangle \in L(\bar{s}_{\bar{p}}, \tilde{\gamma}_2, \tilde{s}_2)$ gilt, ist die Ungleichung erfüllt.

5. Fall: $L(s_p, \gamma, \tilde{s}) \supseteq [r](L(s_{\hat{p}}, \hat{\gamma}, \hat{s}), L(\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}))$ für $(s_p, \gamma, \tilde{s}), (s_{\hat{p}}, \hat{\gamma}, \hat{s}), (\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}) \in \bar{\delta}_{stack}$, $(\hat{s}, \#, \bar{s}) \in \delta_{separator}$, $s, \bar{s} \in S_c$, $\hat{s}, \tilde{s} \in S_s$, $r = p\gamma \hookrightarrow \hat{p}\hat{\gamma}\#\tilde{p}\tilde{\gamma} \in \Delta$

Nach Definition gilt $\hat{p}\hat{\gamma} \overset{\langle\hat{\tau}\rangle}{\Longrightarrow} \hat{c}$, mit $Runs_{\mathcal{A}}(s, \hat{c}, \hat{s}) \neq \emptyset$ für beliebiges $\langle\tau_2\rangle \in L(s_{\hat{p}}, \hat{\gamma}, \hat{s})$ und passendes $\hat{c} \in P\Gamma^*(\#P\Gamma^*)^*$. Außerdem gilt $\tilde{p}\tilde{\gamma} \overset{\langle\tilde{\tau}\rangle}{\Longrightarrow} \tilde{c}$, mit $Runs_{\mathcal{A}}(\bar{s}, \tilde{c}, \tilde{s}) \neq \emptyset$ für beliebiges $\langle\tau_1\rangle \in L(\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s})$ und passendes $\tilde{c} \in P\Gamma^*(\#P\Gamma^*)^*$. Dann lässt sich aus einem Lauf für $\hat{c}$ von s nach $\hat{s}$ in $\mathcal{A}$, der Transition $(\hat{s}, \#, \bar{s})$ und einem Lauf für $\tilde{c}$ von $\bar{s}$ nach $\tilde{s}$ in $\mathcal{A}$, ein Lauf für $\bar{c} = \tilde{c}\#\hat{c}$ in $\mathcal{A}$ konstruieren. Es gilt also $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$. Zudem gilt $p\gamma \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$, für $\langle\tau\rangle = \langle[r](\hat{\tau}, \tilde{\tau})\rangle$, und damit $\langle\tau\rangle \in L(s_p, \gamma, \tilde{s})$. Da dies für beliebige $\langle\tau_1\rangle \in L(\bar{s}_{\tilde{p}}, \tilde{\gamma}, \tilde{s}), \langle\tau_2\rangle \in L(s_{\hat{p}}, \hat{\gamma}, \hat{s})$ gilt, ist damit auch die Ungleichung erfüllt.

□

Wir können also theoretisch einen annotierten Automaten konstruieren, in dem zu jeder Transition die Klasse von Popsequenzen vermerkt ist, die durch diese beschrieben wird. Für WPDS wurde gezeigt, dass durch Verknüpfung der Klassen von Popsequenzen zu Transitionen entlang akzeptierender Pfade des konstruierten Automaten für eine Konfiguration c, die Menge der erreichenden Pfade konstruiert werden konnte. Dieses Ergebnis gilt auch für

DPN, nur das hier die Menge der erreichenden Hecken konstruiert wird. Um diesen Vorgang des Auslesens der Menge der erreichenden Hecken aus dem Automaten zu formalisieren, definieren wir eine Funktion, die zu einem Lauf des annotierten Automaten eine Menge von Hecken konstruiert.

Definition 3.26. Zu einem Lauf $\phi \in Runs_{\mathcal{A}^*}$ definieren wir den Wert:

$$\pi(\phi) = \begin{cases} \{\langle[]\rangle\} & \text{für } \phi = [\![s]\!], s \in S \\ \pi(\tilde{\phi}) \triangleright \langle[]\rangle & \text{für } \phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!], (s, \lambda, \tilde{s}) \in \bar{\delta}_{separator} \\ \pi(\tilde{\phi}) & \text{für } \phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!], (s, \lambda, \tilde{s}) \in \bar{\delta}_{state} \\ \pi(\tilde{\phi}); l(t) & \text{für } \phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!], (s, \lambda, \tilde{s}) \in \bar{\delta}_{stack} \end{cases}.$$

Man konstruiert also aus den zu den auf dem Lauf angetroffenen Transitionen gehörenden Klassen von Popsequenzen eine Menge von Hecken. Transition in $\bar{\delta}_{state}$ und $\bar{\delta}_{separator}$, die keine Popsequenzen beschreiben, nehmen dabei eine gesonderte Stellung ein. Eine Transition in $\bar{\delta}_{state}$ dient lediglich der Strukturierung des Automaten und hat daher keinen Einfluss auf die erreichenden Hecken der betrachteten Konfiguration. Eine Transition in $\bar{\delta}_{stack}$ signalisiert jedoch das Ende eines Stacks der Transition und den Beginn eines neuen. Wenn das Ende eines Stacks erreicht ist, heißt dies im Sinne der Betrachtung von Popsequenzen, dass jedes Element des Stacks bearbeitet wurde. Damit ist der Baum der diesen Stack bearbeitet vollständig. Für den nächsten Stack wird ein neuer Baum der Hecke hinzugefügt. Beim Durchlaufen einer Transition in $\bar{\delta}_{stack}$, die eine nicht leere Klasse von Popsequenzen beschreibt, werden die Popsequenzen an die bis zu diesem Zeitpunkt konstruierten Hecken angehängt. Für die so gewonnene Menge von Hecken gelten für bestimmte Typen von Läufen dann folgende Eigenschaften:

Korollar 3.27. *Für einen Lauf $\phi \in Runs_{\mathcal{A}^*}$ mit $\phi^w = w$ und $w \in \Sigma^+$ gilt:*

1. *für $w = p\gamma_1 \dots \gamma_n \in P\Gamma^*, n \in \mathbb{N}$, also $\phi = [\![s, p, s_1]\!]; [\![s_1, \gamma_1, s_2]\!]; \dots; [\![s_{n-1}, \gamma_n, s_n]\!]$, für $s \in S_c, s_i \in S_s$, dass*

$$\pi(\phi) = l((s_1, \gamma_1, s_2)); \dots; l((s_{n-1}, \gamma_n, s_n)).$$

2. *für $w = p_1 w_1 \# \dots \# p_n w_n$, mit $p_i w_i \in P\Gamma^*, n \in \mathbb{N}$, also $\phi = \phi_1; [\![\tilde{s}_1, \#, s_2]\!]; \dots; [\![\tilde{s}_{n-1}, \#, s_n]\!]; \phi_n$, für $s_i \in S_c, \tilde{s}_i \in S_s$ und $\phi_i \in Runs_{\mathcal{A}^*}(s_i, p_i w_i, \tilde{s}_i)$, dass*

$$\pi(\phi) = (\dots(\pi(\phi_1) \triangleright \pi(\phi_2)) \triangleright \dots) \triangleright \pi(\phi_n).$$

Beweis. zu 1.: Folgt durch Induktion nach n aus Lemma 3.21 und der Definition von π.

zu 2.: Folgt durch Induktion nach n mit 1. aus Lemma 3.21 und der Definition von π, sowie der Tatsache, dass $(M_1 \triangleright M_2); M_3 = M_1 \triangleright (M_2; M_3)$, für $M_1 \subseteq Hedges, M_2, M_3 \subseteq Hedges^1$.

□

Für WPDS wurde gezeigt, dass die Menge der erreichenden Pfade für eine Konfiguration c genau der Menge der Pfade entspricht, die man durch Verknüpfung der Klassen von Popsequenzen entlang der akzeptierenden Läufe für die Konfiguration im saturierten Automaten erhält. Analog definieren wir mit der Hilfe der Funktion π, die zu einem Lauf die Menge der Hecken liefert die entlang des Laufes konstruiert werden können, die Menge der Hecken für eine Konfiguration c im Automaten $\mathcal{A}^*$.

Definition 3.28. Wir definieren zu einer Konfiguration $c \in Conf$ den Wert $\pi(c) \subseteq Hedges$ als:

$$\pi(c) = \bigcup\{\pi(\phi) \mid \phi \in Accept_{\mathcal{A}^*}(c)\}.$$

Wir wollen nun zeigen dass dieser Wert der Menge der erreichenden Hecken für die Konfiguration c entspricht.

Satz 3.29. *Für eine beliebige Konfiguration $c \in Conf$ gilt:*

$$\pi(c) = Hedges(c, C).$$

In einem ersten Schritt betrachten wir dazu die Menge der Bäume die man durch einen Lauf gewinnen kann, der eine einfache Konfiguration pw in einem der Teilautomaten von $\mathcal{A}^*$ von einem $s \in S_c$ zu einem $\tilde{s} \in S_s$ einliest. Bei der betrachteten Menge handelt es um eine Menge von Bäumen, da nur ein Stack durch den Lauf erkannt wird. Die Existenz eines solchen Laufes impliziert die Existenz nichtleerer Klassen von Popsequenzen für alle durchlaufenen Transitionen, und für die Menge der Bäume sollte gelten, dass diese die Konfiguration pw in eine Konfiguration transformiert, die der ursprüngliche Automat erkennt. Man kann sogar zeigen, dass der ursprüngliche Automat die erreichte Konfiguration von s nach $\tilde{s}$ erkennt. Umgekehrt kann man für jeden Baum der die Konfiguration pw in eine Konfiguration transformiert, die dann im ursprünglichen Automaten von s nach $\tilde{s}$ erkannt wird, einen Lauf im saturierten Automaten von s nach $\tilde{s}$ finden, der pw erkennt und den Baum in der Menge der Bäume des Laufes enthalten ist.

Lemma 3.30. *Für eine Hecke $\langle\tau\rangle \in Hedges^1$ und $s \in S_c, \tilde{s} \in S_s, pw \in P\Gamma^*$ gilt, dass ein Lauf $\phi \in Runs_{\mathcal{A}^*}(s, pw, \tilde{s})$ mit $\langle\tau\rangle \in \pi(\phi)$ genau dann existiert, wenn ein $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$ existiert mit $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ und $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$.*

Beweis. Zeige $\Leftarrow$ und $\Rightarrow$ getrennt:

(i) $\Leftarrow$

Wir müssen zeigen, dass zu einem beliebigen Baum, der eine einfache Konfiguration pw in eine Konfiguration $\bar{c}$ überführt, welche dann in $\mathcal{A}$ von einem Zustand s zu einem Zustand $\tilde{s}$ erkannt wird, ein Lauf in $\mathcal{A}^*$ existiert der die Ausgangskonfiguration von s nach $\tilde{s}$ erkennt und den Baum in der von ihm beschriebenen Menge enthält. Unterscheide zwei Fälle.

1. Fall: $|w| = 0$

Dann gilt $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$ nur für $\bar{c} = p$ und $\langle\tau\rangle = \langle[]\rangle$, da auf einem leeren Stack keine Transitionen ausgeführt werden können. Dann gilt $\tilde{s} = s_p$ und $(s, p, \tilde{s}) \in \delta_{state} = \bar{\delta}_{state}$ und damit $Runs_{\mathcal{A}^*}(s, pw, \tilde{s}) = \{[\![s, p, \tilde{s}]\!]\}$. Nach Definition gilt für $\phi = \{[\![s, p, \tilde{s}]\!]\}$, dass $\pi(\phi) = \{\langle[]\rangle\}$ und damit die Behauptung.

2. Fall: $|w| > 0$

Sei dann $w = \gamma_1 \dots \gamma_n$, mit $\gamma_i \in \Gamma, n \in \mathbb{N}$, und $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, mit $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$. Unterscheide wiederum zwei Fälle:

i. Fall: $\bar{c} = \check{c}\check{\gamma}$, für $\check{c} \in P\Gamma^*(\#P\Gamma^*)^*, \check{\gamma} \in \Gamma$

Dann befindet man sich in der Situation von Lemma 3.19 und kann den Baum entsprechend aufteilen. Es existieren also Pop- und Pushphase $\langle\tau_1\rangle, \langle\tau_2\rangle \in Hedges^1$, mit $\langle\tau\rangle = \langle\tau_1\rangle; \langle\tau_2\rangle$, und $\check{v} \in (P\Gamma^*\#)^*, \check{c} \in P\Gamma^*(\#P\Gamma^*)^*, \check{\gamma} \in \Gamma, \tilde{w}_1, \tilde{w}_2 \in \Gamma^*, \tilde{\gamma}_1 \in \Gamma$, mit $\bar{c} = \check{c}\check{\gamma} = \check{v}\check{c}\check{\gamma}\tilde{w}_2$ und $w = \tilde{w}_1\tilde{\gamma}_1\tilde{w}_2$, sowie $\tilde{p}_1 \in P$, so dass $p\tilde{w}_1 \overset{\langle\tilde{\tau}_1\rangle}{\Longrightarrow} \check{v}\tilde{p}_1$ und $\tilde{p}_1\tilde{\gamma}_1 \overset{\langle\tilde{\tau}_2\rangle}{\Longrightarrow} \check{c}\check{\gamma}$. Unterscheide zwei weitere Fälle:

a. Fall: $|\tilde{w}_1| = 0$

Es gilt $\tilde{w}_1 = \epsilon$, und damit $\langle\tau_1\rangle = \langle[]\rangle$, also insgesamt $\langle\tau\rangle = \langle\tau_2\rangle$, und $\check{v} = \epsilon$, sowie $\tilde{\gamma}_1 = \gamma_1, \tilde{w}_2 = \gamma_2 \dots \gamma_n$. In diesem Fall findet keine Popphase statt, sondern nur eine Pushphase. Dabei bleibt der Stack unterhalb des obersten Stacksymbols unangetastet. Dann gilt $\bar{c} = \check{c}\check{\gamma} = \check{c}\check{\gamma}\gamma_2 \dots \gamma_n$. Da $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$, existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von s nach $\tilde{s}$ erkennt. Diesen kann man dann in n Teile aufspalten. Der erste Teil erkennt $\check{c}\check{\gamma}$ von s zu einem $s_1 \in S$. Nach Lemma 3.21 gilt $s_1 \in S_s \setminus S_p$. Es gilt also $Runs_{\mathcal{A}}(s, \check{c}\check{\gamma}, s_1) \neq \emptyset$ und damit $\langle\tau\rangle \in L(s_p, \gamma_1, s_1)$. Nach Lemma 3.16 existiert dann eine Kante $t_1 = (s_p, \gamma_1, s_1) \in \bar{\delta}_{stack}$ und nach Lemma 3.25 gilt $\langle\tau\rangle \in l(t_1)$. Die restlichen $n-1$ Teile des Laufes erkennen jeweils ein γ_i von s_{i-1} nach s_i, mit $s_i \in S$, für $i > 1$. Dabei ist $s_n = \tilde{s}$. Nach 3.21 gilt $s_i \in S_s$ und, da $s_1 \notin S_p$, zudem $s_i \notin S_p$, für $i > 1$. Für die dabei durchlaufenen Kanten $t_i = (s_{i-1}, \gamma_i, s_i) \in \delta_{stack} \subseteq \bar{\delta}_{stack}$ gilt dann $l(t_i) = L(s_{i-1}, \gamma_i, s_i) = \{\langle[]\rangle\}$. Für den Lauf $\phi = [\![s, p, s_p]\!]; [\![s_p, \gamma_1, s_1]\!]; \dots; [\![s_{n-1}, \gamma_n, s_n]\!] \in Runs_{\mathcal{A}^*}(s, pw, \tilde{s})$ in $\mathcal{A}^*$ gilt dann, nach Lemma 3.27, dass $\langle\tau\rangle = \langle\tau\rangle; \langle[]\rangle; \dots; \langle[]\rangle \in l(t_1); l(t_2); \dots; l(t_3) = \pi(\phi)$.

b. Fall: $|\tilde{w}_1| > 0$

Sei dann $\tilde{w}_1 = \gamma_1 \dots \gamma_m, m \in \mathbb{N}, m < n$. Es findet also eine Popphase statt, die die ersten m Elemente vom Stack entfernt. Nach Lemma 3.20 kann diese noch weiter aufgeteilt werden kann. Es existieren Popsequenzen $\langle\tilde{\tau}_1\rangle, \dots, \langle\tilde{\tau}_m\rangle \in Hedges^1$, mit $\langle\tau_1\rangle = \langle\tilde{\tau}_1\rangle; \dots; \langle\tilde{\tau}_m\rangle$ und $\check{v}_1, \dots, \check{v}_m \in (P\Gamma^*\#)^*$, $p_2, \dots, p_m \in P$, mit $\check{v} = \check{v}_1 \dots \check{v}_m$ und $p_i\gamma_i \overset{\langle\tau_i\rangle}{\Longrightarrow} \check{v}_i p_{i+1}$. Dabei sei $p_1 = p$ und $p_{m+1} = \tilde{p}_1$. Dann gilt $\bar{c} = \check{c}\check{\gamma} = \check{v}_1 \dots \check{v}_m\check{c}\check{\gamma}\tilde{w}_2$. Da $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$ existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von s nach $\tilde{s}$ erkennt. Diesen kann man nun in n Teile aufspalten. Die ersten m Teile erkennen jeweils ein $\check{v}_i$ von s^i nach s^{i+1}, für $i \leq m, s^j \in S, j \leq m+1$. Dabei ist $s^1 = s$. Nach Lemma 3.21 gilt dann $s^i \in S_c$, dabei gilt für $\check{v}_i = \epsilon$, dass $s^{i+1} = s^i$. Da für jeden Zustand $s^{i+1} \in S_c$ nach Definition eine Kante $(s^{i+1}, p_{i+1}, s^{i+1}_{p_{i+1}}) \in \delta_{state}$ existiert, kann man die oben genannten Teilläufe verlängern und es gilt $Runs_{\mathcal{A}}(s^i, \check{v}_i p_{i+1}, s^{i+1}_{p_{i+1}}) \neq \emptyset$. Nach Lemma 3.16 exis-

tieren dann Kanten $t_i = (s^i_{p_i}, \gamma_i, s^{i+1}_{p_{i+1}}) \in \bar{\delta}_{stack}$ und nach Lemma 3.25 gilt $\langle\tilde{\tau}_i\rangle \in l(t_i)$. Der $m+1$ Teil des Laufes für $\bar{c}$ erkennt $\check{c}\check{\gamma}$ von s^{m+1} nach s^{m+2}, für $s^{m+2} \in S$. Nach Lemma 3.21 gilt $s^{m+2} \in S_s \setminus S_p$. Es gilt also $Runs_{\mathcal{A}}(s^{m+1}, \check{c}\check{\gamma}, s^{m+2}) \neq \emptyset$ und damit existiert nach Lemma 3.16 eine Kante $t_{m+1} = (s^{m+1}_{p_{m+1}}, \gamma_{m+1}, s^{m+2}) \in \bar{\delta}_{stack}$ und nach Lemma 3.25 gilt $\langle\tau_2\rangle \in l(t_{m+1})$. Die restlichen $n-m-1$ Teile des Laufs erkennen jeweils eines der verbliebenen Stacksymbole γ_i von s^i nach s^{i+1}, für $s^i \in S, i > m+1$. Dabei ist $s^{n+1} = \tilde{s}$. Nach Lemma 3.21 gilt $s^i \in S_s \setminus S_p$, da $s^{m+2} \notin S_p$. Es existieren also Kanten $t_i = (s^i, \gamma_i, s^{i+1}) \in \delta_{stack} \subseteq \bar{\delta}_{stack}$, für $i > m+1$. Und nach Definition von $L(s^i, \gamma_i, s^{i+1})$ und Lemma 3.16 gilt $l(t_i) = \{\langle[]\rangle\}$. Für den Lauf $\phi = [\![s, p, s_p]\!]; [\![s_p, \gamma_1, s^2_{p_2}]\!]; \ldots; [\![s^{m+1}_{\tilde{p}_1}, \gamma_{m+1}, s^{m+2}]\!]; \ldots; [\![s^n, \gamma_n, \tilde{s}]\!] \in Runs_{\mathcal{A}^*}(s, pw, \tilde{s})$ in $\mathcal{A}^*$, gilt dann, nach Lemma 3.27, dass $\langle\tau\rangle = \langle\tau_1\rangle; \langle\tau_2\rangle = \langle\tilde{\tau}_1\rangle; \ldots; \langle\tilde{\tau}_m\rangle; \langle\tau_2\rangle; \langle[]\rangle; \ldots; \langle[]\rangle \in l(t_1); \ldots; l(t_n) = \pi(\phi)$.

ii. Fall: $\bar{c} = \hat{v}\hat{p}$, für $\hat{v} \in (P\Gamma^*\#)^*, \hat{p} \in P$

Dann befindet man sich in der Situation von Lemma 3.20 und kann den Baum entsprechend aufteilen. Es existieren also Popsequenzen $\langle\tau_1\rangle, \ldots, \langle\tau_n\rangle \in Hedges^1$ mit $\langle\tau\rangle = \langle\tau_1\rangle; \ldots; \langle\tau_n\rangle$ und $\hat{v}_1, \ldots, \hat{v}_n \in (P\Gamma^*\#)^*$, $p_2, \ldots, p_n \in P$ mit $\hat{v} = \hat{v}_1 \ldots \hat{v}_n$ und $p_i\gamma_i \overset{\langle\tau_i\rangle}{\Longrightarrow} \hat{v}_i p_{i+1}$, dabei sei $p_1 = p, p_{n+1} = \hat{p}$. Dann gilt $\bar{c} = \hat{v}_1 \ldots \hat{v}_n\hat{p}$. Da $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$ existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von s nach $\tilde{s}$ erkennt. Diesen kann man nun in n Teile aufspalten. Die ersten $n-1$ Teile erkennen jeweils ein $\hat{v}_i$ von s^i nach s^{i+1}, für $i < n, s^j \in S, j \leq n$. Dabei ist $s^1 = s$. Nach Lemma 3.21 gilt dann $s^i \in S_c$, dabei gilt für $\hat{v}_i = \epsilon$, dass $s^{i+1} = s^i$. Da für jeden Zustand $s^{i+1} \in S_c$ zusätzlich eine Kante $(s^{i+1}, p_{i+1}, s^{i+1}_{p_{i+1}}) \in \delta_{state}$ existiert, kann man die oben genannten Teilläufe verlängern und es gilt $Runs_{\mathcal{A}}(s^i, \hat{v}_i p_{i+1}, s^{i+1}_{p_{i+1}}) \neq \emptyset$. Nach Lemma 3.16 existieren dann Kanten $t_i = (s^i_{p_i}, \gamma_i, s^{i+1}_{p_{i+1}}) \in \bar{\delta}_{stack}$ und nach Lemma 3.25 gilt $\langle\tau_i\rangle \in l(t_i)$. Analog existiert auch für den letzten Teil des Laufes, der $\hat{v}_n\hat{p}$ von s^n nach $\tilde{s}$ erkennt, eine Kante $t_n = (s^n_{p_n}, \gamma_n, \tilde{s}) \in \bar{\delta}_{stack}$ mit $\langle\tau_n\rangle \in l(t_n)$. Für den Lauf $\phi = [\![s, p, s_p]\!]; [\![s_p, \gamma_1, s^2_{p_2}]\!]; \ldots; [\![s^n_{p_n}, \gamma_n, \tilde{s}]\!] \in Runs_{\mathcal{A}^*}(s, pw, \tilde{s})$ in $\mathcal{A}^*$ gilt dann, nach Lemma 3.27, dass $\langle\tau\rangle = \langle\tau_1\rangle; \ldots; \langle\tau_n\rangle \in l(t_1); \ldots; l(t_n) = \pi(\phi)$.

(ii) $\Rightarrow$

Zu einem Lauf in $\mathcal{A}^*$, der eine Konfiguration pw von einem Zustand s zu einem Zustand $\tilde{s}$ erkennt, müssen wir zeigen, dass ein beliebiger Baum, der in der durch den Lauf beschriebenen Menge liegt, die Anfangskonfiguration in eine eine Konfiguration $\bar{c}$ überführt, welche dann in $\mathcal{A}$ vom Zustand s zum Zustand $\tilde{s}$ erkannt wird. Unterscheide zwei Fälle:

1. Fall: $|w| = 0$

Dann gilt $\tilde{s} = s_p$ und $(s, p, \tilde{s}) \in \bar{\delta}_{state} = \delta_{state}$ und damit $Runs_{\mathcal{A}^*}(s, pw, \tilde{s}) = \{[\![s, p, \tilde{s}]\!]\} = Runs_{\mathcal{A}}(s, pw, \tilde{s})$. Für $\phi = [\![s, p, \tilde{s}]\!]$ gilt zudem $\pi(\phi) = \{\langle[]\rangle\}$ und damit $\langle\tau\rangle = \langle[]\rangle$ für $\langle\tau\rangle \in \pi(\phi)$. Damit folgt dann $pw \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$, für $\bar{c} = pw$. Da, wie oben schon angemerkt, $Runs_{\mathcal{A}}(s, pw, \tilde{s}) \neq \emptyset$, gilt also die Behauptung.

2. Fall: $|w| > 0$

Sei $w = \gamma_1 \ldots \gamma_n$, und $\phi \in Runs_{\mathcal{A}^*}(s, pw, \tilde{s})$ ein Lauf in $\mathcal{A}^*$ mit $\langle\tau\rangle \in \pi(\phi)$. Man kann ϕ in $n+1$ Teile aufspalten, die jeweils ein Symbol der Konfiguration erkennen. Der erste

Teillauf erkennt den Kontrollzustand von s zu einem $s_1 \in S$. Nach Lemma 3.21 gilt $s_1 = s_p \in S_p$. Die folgenden Teile erkenne jeweils ein γ_i von einem Zustand s_i zu einem Zustand s_{i+1}, mit $s_i \in S$. Dabei gilt $s_{n+1} = \tilde{s}$. Nach Lemma 3.21 gilt $s_i \in S_s$ und es existiert ein $k \in \mathbb{N}$, mit $s_i \in S_p$ für alle $i \leq k$ und $s_i \notin S_p$ für alle $i > k$. Es gilt also $\phi = [\![s, p, s_1]\!]; [\![s_1, \gamma_1, s_2]\!]; \ldots; [\![s_{n-1}, \gamma_n, s_n]\!]$. Nach Korollar 3.27 und Lemma 3.25 gilt dann $\pi(\phi) = l((s_1, \gamma_1, s_2)); \ldots; l((s_{n-1}, \gamma_n, s_n)) = L(s_1, \gamma_1, s_2); \ldots; L(s_{n-1}, \gamma_n, s_n)$ und damit $\langle\tau\rangle = \langle\tau_1\rangle; \ldots; \langle\tau_n\rangle$, mit $\langle\tau_i\rangle \in L(s_i, \gamma_i, s_{i+1})$, für $\langle\tau\rangle \in \pi(\phi)$. Da $s_i = \bar{s}^i_{p_i} \in S_p$, mit $\bar{s}^i \in S_c, p_i \in P$, für $i \leq k$, existieren, nach Definition von $L(\bar{s}^i_{p_i}, \gamma_i, s_{i+1})$, dann $\bar{c}_i \in P\Gamma^*(\#P\Gamma^*)^*$, mit $p_i\gamma_i \overset{\langle\tau_i\rangle}{\Longrightarrow} \bar{c}_i$ und $Runs_{\mathcal{A}}(\bar{s}^i, \bar{c}_i, s_{i+1} \neq \emptyset$, für $i \leq k$. Da für $i < k$ auch $s_{i+1} = \bar{s}^{i+1}_{p_{i+1}}$, gilt nach Lemma 3.21, dass $\bar{c}_i = \bar{v}_i p_{i+1}$, für $\bar{v}_i \in (P\Gamma^*\#)^*$, und dass ein Lauf in $\mathcal{A}$ für $\bar{c}_i$ von $\bar{s}^i$ nach $\bar{s}^{i+1}_{p_{i+1}}$ aus einem Lauf für $\bar{v}_i$ von $\bar{s}^i$ nach $\bar{s}^{i+1}$ und einer Kante $(\bar{s}^{i+1}, p_{i+1}, \bar{s}^{i+1}_{p_{i+1}}) \in \delta_{state}$ besteht. Damit gilt $Runs_{\mathcal{A}}(\bar{s}^i, \bar{v}_i, \bar{s}^{i+1}) \neq \emptyset$. Dann gilt nach Lemma 3.18, dass $p\gamma_1 \ldots \gamma_k \overset{\langle\tau_1\rangle;\ldots;\langle\tau_k\rangle}{\Longrightarrow} \bar{v}_1 \ldots \bar{v}_{k-1}\bar{c}_i$. Desweiteren kann man die Läufe, die $\bar{v}_i$ von $\bar{s}^i$ nach $\bar{s}^{i+1}$ erkennen und den Lauf, der $\bar{c}_k$ von $\bar{s}^k$ nach s_{k+1} erkennt kombinieren und erhält $Runs_{\mathcal{A}}(s, \bar{v}_1 \ldots \bar{v}_{k-1}\bar{c}_k, s_{k+1}) \neq \emptyset$. Für $i > k$ gilt $s_i \notin S_p$ und damit direkt $(s_i, \gamma_i, s_{i+1}) \in \delta_{stack}$, da durch den Algorithmus keine Kanten, die nicht von Zuständen in S_p ausgehen, hinzugefügt werden. Dann gilt nach Definition von $L(s_i, \gamma_i, s_{i+1})$ aber schon $\langle\tau_i\rangle = \langle[]\rangle$ und damit insgesamt $p\gamma_1 \ldots \gamma_n \overset{\langle\tau\rangle}{\Longrightarrow} \bar{c}$, für $\langle\tau\rangle = \langle\tau_1\rangle; \ldots; \langle\tau_n\rangle$ und $\bar{c} = \bar{v}_1 \ldots \bar{v}_{k-1}c_k\gamma_{k+1} \ldots \gamma_n$. Für den ersten Teil von $\bar{c}$ existiert wie oben gezeigt ein Lauf in $\mathcal{A}$, der diesen von s nach s_{k+1} erkennt. Diesen kann man durch die Kanten $(s_i, \gamma_i, s_{i+1})\delta_{stack}$, für $i > k$, zu einem Lauf in $\mathcal{A}$ verlängern, der die komplette Konfiguration $\bar{c}$ von s nach $\tilde{s}$ erkennt und erhält $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$.

□

Das Ergebnis, dass Teilläufe, die einzelne Stacks erkennen, genau die Menge von Bäumen beschreiben, die diese Stacks in eine durch den ursprünglichen Automaten erkannte Teilkonfiguration transformieren, kann man nun dazu verwenden, eine ähnliche Aussage für Läufe zu machen die ganze Konfigurationen, bestehend aus mehreren Stacks, erkennen und demnach Hecken, bestehend aus mehreren Bäumen, beschreiben.

Lemma 3.31. *Für eine Hecke $\sigma \in Hedges$ und $s \in S_c, \tilde{s} \in S_s, c \in P\Gamma^*(\#P\Gamma^*)^*$ gilt, dass ein Lauf $\phi \in Runs_{\mathcal{A}^*}(s, c, \tilde{s})$ mit $\sigma \in \pi(\phi)$ genau dann existiert, wenn ein $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$ existiert, mit $c \overset{\sigma}{\Longrightarrow} \bar{c}$ und $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$.*

Beweis. Zeige $\Leftarrow$ und $\Rightarrow$ getrennt:

(i) $\Leftarrow$

Wir müssen zeigen, dass zu einer beliebigen Hecke, die eine Konfiguration c in eine Konfiguration $\bar{c}$ überführt, welche dann in $\mathcal{A}$ von einem Zustand s zu einem Zustand $\tilde{s}$ erkannt wird, ein Lauf in $\mathcal{A}^*$ existiert der die Ausgangskonfiguration von s nach $\tilde{s}$ erkennt und die Hecke in der von ihm beschriebenen Menge enthält.

Sei $\sigma = \langle \tau_1, \ldots, \tau_n \rangle$ und $\bar{c} \in P\Gamma^*(\#P\Gamma^*)^*$, dann gilt $c \overset{\sigma}{\Longrightarrow} \bar{c}$, wenn $c = p_1 w_1 \# \ldots \# p_n w_n$, mit $p_i w_i \in P\Gamma^*$, und $\bar{c} = \bar{c}_1 \# \ldots \# \bar{c}_n$, für $\bar{c}_i \in P\Gamma^*(\#P\Gamma^*)^*$, mit $p_i w_i \overset{\langle \tau_i \rangle}{\Longrightarrow} \bar{c}_i$. Da $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$, existiert ein Lauf in $\mathcal{A}$, der $\bar{c}$ von s nach $\tilde{s}$ erkennt. Diesen kann man in $n + (n-1)$ Teile aufteilen, die abwechselnd eine Teilkonfiguration $\bar{c}_i$ von einem Zustand s_i zu einem Zustand $\tilde{s}_i$ und ein Trennzeichen $\#$ von $\tilde{s}_i$ nach s_{i+1} erkennen. Dabei gilt $s_1 = s$ und $\tilde{s}_n = \tilde{s}$. Nach Lemma 3.21 gilt dann $s_i \in S_c$ und $\tilde{s}_i \in S_s$. Es gilt also $Runs_{\mathcal{A}}(s_i, \bar{c}_i, \tilde{s}_i) \neq \emptyset$ und damit existieren nach Lemma 3.30 Läufe $\phi_i \in Runs_{\mathcal{A}^*}(s_i, p_i w_i, \tilde{s}_i)$ mit $\langle \tau_i \rangle \in \pi(\phi_i)$. Aus diesen Läufen, und den zusätzlich existierenden Läufen für die Trennzeichen, da $\delta_{separator} = \bar{\delta}_{separator}$ existieren diese auch in $\mathcal{A}^*$, kann man nun einen Lauf $\phi = \phi_1; [\![\tilde{s}_1, \#, s_2]\!]; \ldots; [\![\tilde{s}_{n-1}, \#, s_n]\!]; \phi_n \in Runs_{\mathcal{A}^*}(s, c, \tilde{s})$ in $\mathcal{A}^*$ zusammensetzen. Nach Korollar 3.27 gilt dann $\sigma = \langle \tau_1, \ldots, \tau_n \rangle = (\ldots(\langle \tau_1 \rangle \triangleright \langle \tau_2 \rangle) \triangleright \ldots) \triangleright \langle \tau_n \rangle \in (\ldots(\pi(\phi_1) \triangleright \pi(\phi_2)) \triangleright \ldots) \triangleright \pi(\phi_n) = \pi(\phi)$, und damit die Behauptung.

(ii) $\Rightarrow$

Zu einem Lauf in $\mathcal{A}^*$, der eine Konfiguration c von einem Zustand s zu einem Zustand $\tilde{s}$ erkennt müssen wir zeigen, dass eine beliebige Hecke, die in der durch den Lauf beschriebenen Menge liegt, die Anfangskonfiguration in eine eine Konfiguration $\bar{c}$ überführt, welche dann in $\mathcal{A}$ vom Zustand s zum Zustand $\tilde{s}$ erkannt wird.

Sei $c = p_1 w_1 \# \ldots \# p_n w_n$ und $\phi \in Runs_{\mathcal{A}^*}(s, c, \tilde{s})$ ein Lauf in $\mathcal{A}^*$ mit $\sigma \in \pi(\phi)$. Man kann ϕ aufteilen in $n + (n-1)$ Teile, die abwechselnd eine Teilkonfiguration $p_i w_i$ von einem Zustand s_i zu einem Zustand $\tilde{s}_i$ und ein Trennzeichen $\#$ von $\tilde{s}_i$ nach s_{i+1} erkennen. Dabei gilt $s_1 = s$ und $\tilde{s}_n = \tilde{s}$. Nach Lemma 3.21 gilt dann $s_i \in S_c$ und $\tilde{s}_i \in S_s$. Insgesamt gilt also $\phi = \phi_1; [\![\tilde{s}_1, \#, s_2]\!]; \ldots; [\![\tilde{s}_{n-1}, \#, s_n]\!]; \phi_n$, mit den Teilläufen $\phi_i \in Runs_{\mathcal{A}^*}(s_i, p_i w_i, \tilde{s}_i)$ in $\mathcal{A}^*$, die die $p_i w_i$ erkennen. Es gilt, nach Korollar 3.27, $\pi(\phi) = (\ldots(\pi(\phi_1) \triangleright \pi(\phi_2)) \triangleright \ldots) \triangleright \pi(\phi_n)$ und damit $\sigma = \langle \tau_1, \ldots, \tau_n \rangle$, mit $\langle \tau_i \rangle \in \pi(\phi_i)$, für $\sigma \in \pi(\phi)$. Nach Lemma 3.30 existieren dann $\bar{c}_i \in P\Gamma^*(\#P\Gamma^*)^*$ mit $p_i w_i \overset{\langle \tau_i \rangle}{\Longrightarrow} \bar{c}_i$ und $Runs_{\mathcal{A}}(s_i, \bar{c}_i, \tilde{s}_i) \neq \emptyset$. Dann gilt insgesamt $c \overset{\sigma}{\Longrightarrow} \bar{c}$, für $\bar{c} = \bar{c}_1 \# \ldots \# \bar{c}_n$. Neben den Läufen in $\mathcal{A}$, die die $\bar{c}_i$ von s_i nach $\tilde{s}_i$ erkennen, existieren nach Voraussetzung auch Läufe, die ein Trennzeichen von $\tilde{s}_i$ nach s_{i+1} in $\mathcal{A}^*$ erkennen. Da $\bar{\delta}_{separator} = \delta_{separator}$, existieren diese aber auch in $\mathcal{A}$. Dann existiert auch insgesamt ein Lauf in $\mathcal{A}$, der $\bar{c}$ von s nach $\tilde{s}$ erkennt und es gilt $Runs_{\mathcal{A}}(s, \bar{c}, \tilde{s}) \neq \emptyset$.

□

Durch die gezielte Betrachtung von akzeptierenden Läufen für eine Konfiguration kann man dann Satz 3.28 beweisen. Die Hecken, die durch einen akzeptierenden Lauf beschrieben werden, transformieren die Konfiguration in eine Konfiguration, die dann auch im ursprünglichen Automaten durch einen akzeptierenden Lauf erkannt wird. Demnach ist die erreichte Konfiguration in der Zielmenge und die Hecke eine erreichende Hecke. Damit können wir nun beweisen, dass wir für eine Konfiguration c aus dem Automaten die Menge der erreichenden Hecken $Hedges(c, C)$ auslesen können.

Beweis. von Satz 3.28:

Unterscheide zwei Fälle:

1. Fall: $c \notin PRE^*(C)$

Dann existiert kein $\sigma \in Hedges$ und $\bar{c} \in C$ mit $c \stackrel{\sigma}{\Longrightarrow} \bar{c}$, also gilt $Hedges(c, C) = \emptyset$. Zudem gilt $c \notin \mathcal{L}(\mathcal{A}^*)$, dass heißt $Accept(c) = \emptyset$, und damit $\pi(c) = \emptyset$.

2. Fall: $c \in PRE^*(C)$

Zeige $\subseteq$ und $\supseteq$ getrennt:

(i) $\subseteq$

Sei $\sigma \in \pi(c)$, dann existiert ein $\phi \in Accept_{\mathcal{A}^*}(c)$ mit $\sigma \in \pi(\phi)$. Damit existiert auch $\tilde{s} \in F$ mit $\phi \in Runs_{\mathcal{A}^*}(s^s, c, \tilde{s})$. Nach Lemma 3.31 existiert dann $\bar{c}$ mit $c \stackrel{\sigma}{\Longrightarrow} \bar{c}$ und $Runs_{\mathcal{A}}(s^s, \bar{c}, \tilde{s}) \neq \emptyset$. Dann gilt $Accept_{\mathcal{A}}(\bar{c}) \neq \emptyset$ und damit $\bar{c} \in C$. Daraus folgt wiederum $\sigma \in Hedges(c, C)$.

(ii) $\supseteq$

Sei $\sigma \in Hedges(c, C)$, dann existiert ein $\bar{c} \in C$ mit $c \stackrel{\sigma}{\Longrightarrow} \bar{c}$. Da $\bar{c} \in C$, gilt $Accept_{\mathcal{A}}(c) \neq \emptyset$ und damit auch $Runs_{\mathcal{A}}(s^s, \bar{c}, s) \neq \emptyset$, für ein $\tilde{s} \in F$. Nach Lemma 3.31 existiert dann ein $\phi \in Runs_{\mathcal{A}^*}(s^s, c, s)$ mit $\sigma \in \pi(\phi)$. Da $s \in F$, gilt $Runs_{\mathcal{A}^*}(s^s, c, s) \subseteq Accept(c)$ und damit auch $\sigma \in \pi(c)$.

□

Wir haben gezeigt, dass wir die Klassen von Popsequenzen durch ein Ungleichungssystem beschreiben können. Desweiteren können wir mit diesen Klassen von Popsequenzen die erreichenden Hecken von einer Konfiguration in eine Menge von Zielkonfigurationen beschreiben. In Abschnitt 3.1 wurde bereits gezeigt, dass diese Menge eng mit der Menge der erreichenden Pfade, für deren Gewicht wir uns interessieren, verbunden ist. Da wir die Klassen von Popsequenzen jedoch nicht ohne weiteres explizit berechnen können, um dann daraus die Menge der erreichende Hecken und Pfade zu erhalten, gehen wir im nächsten Abschnitt zu einer direkten Berechnung von Gewichten für die Klassen von Popsequenzen über. Wir zeigen, dass wir diese durch ein Ungleichungssytem berechnen können. Aus den Gewichten der Popsequenzen lässt sich dann das Gewicht der erreichenden Hecken für eine Konfiguration und damit auch das Gewicht der erreichenden Pfade bestimmen.

3.3 Abstrakte Interpretation

Wir betrachten nun den Übergang von Hecken zu Gewichten. Wie schon im Fall von WPDS wollen wir jeder Transitionsregel eines DPN ein Gewicht zuweisen, um damit Aussagen über die Auswirkungen von Ausführungspfaden eines DPN zu machen. Da die Pfade eines DPN die gleiche sequentielle Struktur wie die Pfade eines PDS haben, bietet sich für die Gewichte die gleiche Struktur eines beschränkten idempotenten Halbringes an.

Im Fall von Pfaden reicht nun der eine Operator, der zwei Gewichte konkateniert, da auf einem Pfad die Transitionen nacheinander abgearbeitet werden. Wie in Abschnitt 3.1 festgestellt, müssen wir für die Beschreibung der Ausführungen eines DPN jedoch auf Hecken zurückgreifen. In einer Hecke existieren nun komplett parallel ablaufende Teile und auch nachträgliche Verzweigungen. Um dies nun für Gewichte ausdrücken zu können, brauchen wir einen zusätzlichen Operator, der die Gewichte von zwei Pfaden nimmt und diese parallel verknüpft. Wenn man aus dem Gewicht eines Astes die genaue Struktur des Astes rekonstruieren kann, kann man diesen Operator durch direkt berechnen. Die Menge der interleavten Pfade lässt sich aus den, aus den gegebenen Gewichten, rekonstruierten Ästen, erzeugen und man kann explizit die Gewichte aller Pfade untersuchen. Da in der Regel jedoch beim Übergang zu Gewichten von der zu Grunde liegende Struktur der Äste abstrahiert wird und man keine Aussage mehr über die tatsächlich auftretenden Transitionen machen kann, braucht man einen eigenen Operator, der nur auf Basis der Gewichte der Äste das Gewicht einer Verzweigung berechnen kann. Da wir die Gewichte einzelner Popsequenzen berechnen wollen, um diese dann zu Gewichten ganzer Hecken zusammenzusetzen, muss dieser Operator zudem mit der Konkatenation von Hecken kompatibel sein. Bei der Konkatenation einer Hecke und eines Baums, wird der Baum an das unterste rechte Blatt des letzten Baumes der Hecke angehängt. Deshalb muss das Gewicht der Hecke, vor der Konkatenation mit dem Baum, verknüpft mit dem Gewicht des Baumes, gleich dem Gewicht der durch die Konkatenation entstandenen Hecke sein. Ist nun das Gewicht der Hecke vor der Konkatenation durch die parallele Verknüpfung der Gewichte der einzelnen Bäume der Hecke gegeben, dann heißt dies, dass das an diese Verknüpfung angehängte Gewicht des Baumes das Gesamtgewicht genauso beeinflussen muss, wie es durch ein direktes Anhängen an das Gewicht des letzten Baumes innerhalb der parallelen Verknüpfung der Hecke passieren würde.

Um dies erreichen zu können, brauchen wir für die Berechnung, neben den Gewichten die den Transitionen zugewiesen werden, gegebenfalls Gewichte, die die Effekte von parallel laufenden Gewichten speichern und ein Anwenden auf nachträglich angehängte Gewichte ermöglichen. In Pfaden gesprochen können wir ein vorläufiges Interleaving mit dem bereits bekannten Teil des rechten Pfad bilden, müssen jedoch die Anweisungen des linken Pfades noch speichern, um bei einer Verlängerung des rechten Pfades die Menge der Interleavings zu korrigieren.

Da diese Gewichte, neben einem vorläufigen Gewicht für das Interleaving, nur zusätzliche Informationen speichern, sollte eine Projektionsfunktion existieren, die diese zusätzlichen Informationen nach Beendigung der Berechnung entfernt und ein endgültiges Gewicht ohne Nebeneffekte herstellt, das dem Gesamtgewicht der Pfade, die durch die Hecke beschrieben werden, entspricht.

Zu einem beschränkten, idempotenten Halbring, der die Gewichte für die Transitionen enthält, definieren wir für die Berechnung dann eine Erweiterung, die einen Interleavingoperator mit den geforderten Eigenschaften, gegebenenfalls zusätzliche Gewichte und eine Projektionsfunktion enthält.

Definition 3.32. Ein *erweiterer Halbring* zu einem beschränkten idempotenten Halbring $(D, \oplus, \odot, 0, 1)$ ist ein Septupel $\mathcal{E} = (E, \oplus_E, \odot_E, \otimes_E, 0, 1, \eta)$. Dabei ist E eine Menge mit $D \subseteq E$, $\eta : E \to D$ ist eine Abbildung in den Basisring und $\oplus_E, \odot_E, \otimes_E$ sind binäre Operatoren auf E. Dabei ist $(E, \oplus_E, \odot_E, 0, 1)$ ein beschränkter idempotenter Halbring und es muss gelten, dass:

1. $d_1 \oplus_E d_2 = d_1 \oplus d_2$ und $d_1 \odot_E d_2 = d_1 \odot d_2$, für $d_1, d_2 \in D$
2. $\otimes_E$ ist assoziativ.
3. $\otimes_E$ distributiert über $\oplus_E$.
4. $(d_1 \otimes_E d_2) \odot_E d_3 = d_1 \otimes_E (d_2 \odot_E d_3)$, für $d_1, d_2, d_3 \in E$
5. $\eta(d) = d$ für alle $d \in D$, also $\eta(E) = D$
6. $\eta(d_1 \oplus_E d_2) = \eta(d_1) \oplus_E \eta(d_2)$, für $d_1, d_2 \in E$
7. $\eta(d_1 \odot_E d_2) = \eta(d_1) \odot_E \eta(d_2) = d_1 \odot \eta(d_2)$ für $d_1 \in D, d_2 \in E$
8. $\eta(1 \otimes_E d) = \eta(d \otimes_E 1) = \eta(d)$, für $d \in E$
9. $\eta((d_1 \odot_E d_2) \otimes_E (d_3 \odot_E d_4)) = \eta(d_1 \odot_E (d_2 \otimes_E (d_3 \odot_E d_4)) \oplus_E d_3 \odot_E ((d_1 \odot_E d_2) \otimes_E d_4)) = (d_1 \odot \eta(d_2 \otimes_E (d_3 \odot_E d_4))) \oplus (d_3 \odot \eta((d_1 \odot_E d_2) \otimes_E d_4))$, für $d_1, d_3 \in D, d_2, d_4 \in E$
10. $\eta(d_1 \otimes_E d_2) = \eta(\eta(d_1) \otimes_E \eta(d_2))$, für $d_1, d_2 \in E$

Die Operatoren des ursprünglichen Halbringes werden auf den gegebenenfalls vergrößerten Wertebereich erweitert. Gewichte mit Nebeneffekten können aus zwei Gewichten ohne Nebeneffekt also nur durch Interleaving erzeugt werden. Die geforderte Assoziativität und Distributivität des Interleavingoperators ist in der Regel keine Einschränkung, da auch das Interleaving zweier Pfade, welches ja durch den Operator beschrieben werden soll, assoziativ und distributiv, vergleiche Korollar 3.7, ist. Kommutativität gilt im allgemeinen nicht, da dies in der Regel der geforderten Eigenschaft widerspricht, dass an ein Interleaving angehängte Gewichte sich nachträglich auf das rechte der interleavten Gewichte auswirkt. Die geforderten Eigenschaften von η garantieren, wie man später sieht, dass sich die Projektion eines Interleavings zweier Gewichte genau wie die Abstraktion des Interleavings zweier Pfade verhält, die diese Gewichte als Gesamtgewicht haben. Dies bedeutet dann, dass der Interleavingoperator, nach entfernen aller eventuell zur Erfüllung der oben genannten Eigenschaft gespeicherten Nebeneffekte, das gewünschte Gewicht liefert.

Da die Operationen $\oplus_E$ und $\odot_E$ auf $D \subseteq E$ mit den Operationen $\oplus$ und $\odot$ übereinstimmen und $\otimes_E$ nur auf E definiert ist, verzichten wir im folgenden auf die Indizierung der Operatoren.

Für den Interleavingoperator der Gewichte und die Projektionsfunktion kann man dann ähnliche Eigenschaften, wie für den Interleavingoperator für Pfade in Korollar 3.7, feststellen.

Korollar 3.33. *Für den Operator $\otimes$ und die Funktion η eines erweiterten Halbrings gilt:*

1. *$\otimes$ ist kommutativ unter η. Es gilt also $\eta(d_1 \otimes d_2) = \eta(d_2 \otimes d_1)$ für $d_1, d_2 \in E$ Elemente des Halbrings.*
2. *Für $d_1, \ldots, d_n \in E$ Elemente des Halbrings und $i_1, \ldots i_m$ Indizes, so dass $d_{i_j} = \hat{d_{i_j}} \odot \tilde{d_{i_j}}$, mit $\hat{d_{i_j}} \in D, \tilde{d_{i_j}} \in E$, für alle $j \in \{1, \ldots, m\}$, und $d_k = 1$, für alle $k \notin \{i_1, \ldots, i_m\}$, gilt*

$$\eta(d_1 \otimes \ldots \otimes d_n) = \eta(d_{i_1} \otimes \ldots \otimes d_{i_m}) = \eta(\bigoplus_{j=1}^{m} \hat{d}_{i_j} \odot (d_{i_1} \otimes \ldots \otimes \tilde{d}_{i_j} \otimes \ldots \otimes d_{i_m})).$$

Beweis. zu 1.: Kommutativität folgt aus den Punkten 7. - 9. der Definition, sowie der Eigenschaft, dass $d = 1 \odot d = d \odot 1$, für $d \in E$.

zu 2.: Folgt mit 1. aus Anwendung der Punkte 8. - 10. der Definition.

□

Mit der Definition des erweiterten Halbrings können wir dann ein gewichtetes dynamisches Push-Down-Netzwerk definieren:

Definition 3.34. Ein *gewichtetes dynamisches Pushdown-Netzwerk* (WDPN) ist ein Tripel $\mathcal{W} = (\mathcal{M}, \mathcal{E}, f)$. Dabei ist $\mathcal{M} = (P, \Gamma, \Delta)$ ein DPN, $\mathcal{E} = (E, \oplus, \odot, \otimes, 0, 1, \eta)$ ein erweiterter Halbring und $f : \Delta \to \eta(E)$ eine Gewichtungsfunktion.

Da es unser Ziel ist die Gewichte von Pfaden im DPN zu berechnen und der Umweg über die Betrachtung von Hecken nur ein Hilfsmittel ist, werden den Transition eines WDPN nur Gewichte des ursprünglichen Halbrings zugeordnet. Wir werden im folgenden nun zeigen, dass die direkte Abstraktion von Pfaden mit Hilfe dieser Gewichte das gleiche Ergebnis liefert, wie die Projektion der Abstraktion der zugehörigen Hecken durch die erweiterten Gewichte. Sei im folgenden also ein WDPN $\mathcal{W} = (\mathcal{M}, \mathcal{S}, f)$, mit DPN $\mathcal{M} = (P, \Gamma, \Delta)$ und erweiterten Halbring $\mathcal{E} = (E, \oplus, \odot, \otimes, 0, 1, \eta)$, gegeben. Damit wiederholen wir die, schon in Abschnitt 3.1 angesprochene, Definition des Gewichts einer Menge von Pfaden in einem WDPN:

Definition 3.35. Wir definieren eine Abstraktionsfunktion $\alpha : \mathcal{P}(Paths) \to \eta(E)$ durch die Abstraktion eines einzelnen Pfades $\rho \in Paths$ durch:

$$\alpha(\rho) = \begin{cases} 1 & \text{für } \rho = [] \\ \alpha(\tilde{\rho}) \odot f(r) & \text{für } \rho = \tilde{\rho}; [r] \end{cases}.$$

Für Mengen $S \subseteq Paths$ sei dann $\alpha(S) = \bigoplus\{\alpha(\rho) \mid \rho \in S\}$.

Es gilt dabei offensichtlich $\alpha(\bigcup_{i=1}^{n} S_i) = \bigoplus_{i=1}^{n} \alpha(S_i)$ und die Funktion ist universell distributiv. Auf der anderen Seite können wir mit Hilfe des erweiterten Halbrings und des Interleavingoperators auch eine Abstraktion für Hecken definieren.

Definition 3.36. Wir definieren eine Abstraktionsfunktion $\beta : \mathcal{P}(Hedges) \to E$ durch die Abstraktion einer einzelnen Hecke $\sigma \in Hedges$ durch:

$$\beta(\sigma) = \begin{cases} \beta(\langle\tau_1\rangle) \otimes \ldots \otimes \beta(\langle\tau_n\rangle) & \text{für } \sigma = \langle\tau_1, \ldots, \tau_n\rangle \\ 1 & \text{für } \sigma = \langle[]\rangle \\ f(r) \odot \beta(\langle\tau\rangle) & \text{für } \sigma = \langle[r](\tau)\rangle \\ f(r) \odot (\beta(\langle\tau_1\rangle) \otimes \beta(\langle\tau_2\rangle)) & \text{für } \sigma = \langle[r](\tau_1, \tau_2)\rangle \end{cases}$$

Für Mengen $S \subseteq Hedges$ sei dann $\beta(S) = \bigoplus\{\beta(\sigma) \mid \sigma \in S\}$.

Es gilt dabei offensichtlich $\beta(\bigcup_{i=1}^{n} S_i) = \bigoplus_{i=1}^{n} \beta(S_i)$ und die Funktion ist universell distributiv. Wir wollen nun zeigen, dass die Abstraktion einer Menge von Hecken auf dem erweiterten Halbring mit Hilfe der Funktion η zur Abstraktion der durch die Menge von Hecken beschriebenen Pfade führt. Seien im folgenden also zwei Abstraktionsfunktionen α und β, wie oben definiert, gegeben.

Betrachten wir zunächst jedoch einige Eigenschaften des erweiterten Halbrings. Nach Definition des Halbrings existiert für jede beliebige Teilmenge von Elementen eine größte untere Schranke. Man kann zeigen, dass sich diese Schranke schon als größte untere Schranke einer endlichen Teilmenge schreiben lässt.

Lemma 3.37. *Sei $M \subseteq E$ eine Menge von Gewichten, dann existiert eine endliche Menge $N \subseteq M$ mit $\bigoplus M = \bigoplus N$.*

Beweis. Wähle eine aufsteigende Folge endlicher Mengen $N_i \subseteq M$ beginnend mit $N_0 = \emptyset$. Im i-ten Schritt wähle ein $d_i \in M \setminus N_{i-1}$, mit $\bigoplus N_{i-1} \oplus d_i \neq \bigoplus N_{i-1}$ und setze $N_i = N_{i-1} \cup \{d_i\}$. Falls kein d_i existiert, das die Bedingung erfüllt, setze $N_i = N_{i-1}$. Für $i \in \mathbb{N}$ gilt dann $\bigoplus N_{i-1} \sqsupseteq \bigoplus N_i = \bigoplus N_{i-1} \oplus d_i$. Damit ist $\bigoplus N_i$ eine unendlich absteigende Kette und wird nach Voraussetzung schließlich stabil. Es existiert also ein $n \in \mathbb{N}$, mit $\bigoplus N_i = \bigoplus N_n$, für alle $i \geq n$. Nach Definition existiert dann kein $d \in M \setminus N_n$, mit $\bigoplus N_n \oplus d \neq \bigoplus N_n$. Dann gilt also $\bigoplus N_n \oplus d = \bigoplus N_n$ und damit auch $\bigoplus N_n \oplus d \sqsupseteq \bigoplus N_n$ für alle $d \in M \setminus N_n$. Daraus folgt, dass auch $\bigoplus N_n \oplus \bigoplus(M \setminus N_n) = \bigoplus M \sqsubseteq \bigoplus N_n$ und da für $N_n \subseteq M$ in jedem Fall $\bigoplus N_n \sqsupseteq \bigoplus M$, gilt insgesamt $\bigoplus N_n = \bigoplus M$. Also gilt für $N = N_n$ die Behauptung. □

Man kann die Bildung einer größten unteren Schranke einer beliebigen Menge also auf die Bildung einer größten unteren Schranke einer endliche Menge reduzieren. Dies kann man dazu benutzen, die universelle Distributivität von $\odot$, $\otimes$ und η über $\oplus$ zu zeigen, indem man sie auf die gegebene paarweise Distributivität des erweiterten Halbrings zurückführt.

Korollar 3.38. *Seien $M, M_1, M_2 \subseteq E$ Mengen von Gewichten, dann gilt:*

1. $\bigoplus M_1 \odot \bigoplus M_2 = \bigoplus\{d_1 \odot d_2 \mid d_1 \in M_1, d_2 \in M_2\}$
2. $\bigoplus M_1 \otimes \bigoplus M_2 = \bigoplus\{d_1 \otimes d_2 \mid d_1 \in M_1, d_2 \in M_2\}$

3. $\eta(\bigoplus M) = \bigoplus\{\eta(d) \mid d \in M\}$

Beweis. Durch Induktion erhält man die endliche Distributivität von $\odot$, $\otimes$ und η. Unendliche Distributivität folgt dann aus Lemma 3.37. □

Mit diesen Hilfsmitteln können wir jetzt zeigen, dass die Abstraktion der von einer Hecke beschriebenen Pfade der Projektion der Abstraktion der Hecke selbst in den ursprünglichen Halbring entspricht.

Lemma 3.39. *Für eine Hecke $\sigma \in Hedges$ gilt, dass $\alpha(\psi(\sigma)) = \eta(\beta(\sigma))$.*

Beweis. Sei $\sigma = \langle \tau_1, \ldots, \tau_n \rangle \in Hedges, n \in \mathbb{N}$. Zeige die Aussage durch Induktion über die Anzahl benannter Knoten in σ[3]:

1. $|\sigma| = 0$

   Dann ist $\tau_i = []$, für $i \in \{1, \ldots, n\}$, und damit gilt $\psi(\sigma) = \{[]\}$ und $\alpha(\psi(\sigma)) = 1$. Außerdem gilt $\beta(\sigma) = \beta(\langle\tau_1\rangle) \otimes \ldots \otimes \beta(\langle\tau_n\rangle) = 1 \otimes \ldots \otimes 1$. Nach Definition von η gilt dann $\eta(1 \otimes \ldots \otimes 1) = \eta(1) = 1$ und damit die Behauptung.

2. $|\sigma| > 0$

   Dann gilt $\tau_i = [r_i](\tilde{\tau_i})$ oder $\tau_i = [r_i](\hat{\tau}_i, \tilde{\tau}_i)$ für ein $i \in \{1, \ldots, n\}, r_i \in \Delta, \langle\tilde{\tau_i}\rangle, \langle\hat{\tau_i}\rangle \in Hedges^1$. Seien $i_1, \ldots, i_m$ die Indizes der nichtleeren Bäume, also

$$\tau_{i_j} = \begin{cases} [r_{i_j}](\tilde{\tau}_{i_j}) & \text{mit } r_{i_j} \in \Delta_1, \langle\tilde{\tau_i}\rangle \text{ oder} \\ [r_{i_j}](\hat{\tau}_{i_j}, \tilde{\tau}_{i_j}) & \text{mit } r_{i_j} \in \Delta_2, \langle\tilde{\tau}_{i_j}\rangle, \langle\hat{\tau}_{i_j}\rangle \in Hedges^1 \end{cases}$$

   für alle $j \in \{1, \ldots, m\}$, und $\tau_k = []$ für alle $k \notin \{i_1, \ldots, i_m\}$. Dann gilt $\psi(\langle\tau_{i_j}\rangle) = [r_{i_j}]; \psi(\tilde{\sigma}_{i_j})$, mit

$$\tilde{\sigma}_{i_j} = \begin{cases} \langle\tilde{\tau_{i_j}}\rangle & \text{für } \tau_{i_j} = [r_{i_j}](\tilde{\tau}_{i_j}) \\ \langle\hat{\tau}_{i_j}, \tilde{\tau}_{i_j}\rangle & \text{für } \tau_{i_j} = [r_{i_j}](\hat{\tau}_{i_j}, \tilde{\tau}_{i_j}) \end{cases},$$

   denn $\psi(\langle\hat{\tau}_{i_j}, \tilde{\tau}_{i_j}\rangle) = \psi(\langle\hat{\tau}_{i_j}\rangle) \parallel \psi(\langle\tilde{\tau}_{i_j}\rangle$. Da zusätzlich $\psi(\langle\tau_k\rangle) = \{[]\}$, für $k \notin \{i_1, \ldots, i_m\}$, folgt dann aus Korollar 3.7, dass:

$$\begin{aligned} \psi(\sigma) &= \psi(\langle\tau_1\rangle) \parallel \ldots \parallel \psi(\langle\tau_n\rangle) \\ &= \psi(\langle\tau_{i_1}\rangle) \parallel \ldots \parallel \psi(\langle\tau_{i_m}\rangle) \\ &= \bigcup_{j=1}^{m} [r_{i_j}]; (\psi(\langle\tau_{i_1}\rangle) \parallel \ldots \parallel \psi(\tilde{\sigma}_{i_j}) \parallel \ldots \parallel \psi(\langle\tau_{i_m}\rangle)) \end{aligned}$$

[3] Die Anzahl benannter Knoten einer Hecke σ ist dabei definiert als

$$|\sigma| = \begin{cases} \sum_{i=1}^{n} |\langle\tau_i\rangle| & \text{für } \sigma = \langle\tau_1, \ldots, \tau_n\rangle \\ 0 & \text{für } \sigma = \langle[]\rangle \\ 1 + |\langle\tilde{\tau}\rangle| & \text{für } \sigma = \langle[r](\tilde{\tau})\rangle \\ 1 + |\langle\hat{\tau}\rangle| + |\langle\tilde{\tau}\rangle| & \text{für } \sigma = \langle[r](\hat{\tau}, \tilde{\tau})\rangle \end{cases}$$

und entspricht der Länge eines Pfades in $\psi(\sigma)$.

Die verbliebenen Interleavings können nun wieder als Pfadmengen von Hecken dargestellt werden. Wähle dazu

$$\sigma_{i_j} = \begin{cases} \langle \tau_{i_1}, \ldots, \tilde{\tau_{i_j}}, \ldots, \tau_{i_m} \rangle & \text{für } \tau_{i_j} = [r_{i_j}](\tilde{\tau}_{i_j}) \\ \langle \tau_{i_1}, \ldots, \hat{\tau}_{i_j}, \tilde{\tau}_{i_j}, \ldots, \tau_{i_m} \rangle & \text{für } \tau_{i_j} = [r_{i_j}](\hat{\tau}_{i_j}, \tilde{\tau}_{i_j}) \end{cases}.$$

Diese Hecken haben nun einen benannten Knoten weniger, da ja eine Transition ausgegliedert wurde. Nach Induktionsvoraussetzung gilt also $\alpha(\psi(\sigma_{i_j})) = \eta(\beta(\sigma_{i_j}))$. Insgesamt gilt dann:

$$\begin{aligned}
\alpha(\psi(\sigma)) &= \alpha(\bigcup_{j=1}^{m} [r_{i_j}]; \psi(\sigma_{i_j})) \\
&= \bigoplus_{j=1}^{m} \alpha([r_{i_j}]; \psi(\sigma_{i_j})) \\
&= \bigoplus_{j=1}^{m} f(r_{i_j}) \odot \alpha(\psi(\sigma_{i_j})) \\
&\overset{\text{IV}}{=} \bigoplus_{j=1}^{m} f(r_{i_j}) \odot \eta(\beta(\sigma_{i_j})) \\
&\overset{f(r_{i_j}) \in \eta(E)}{=} \eta(\bigoplus_{j=1}^{m} f(r_{i_j}) \odot \beta(\sigma_{i_j})) \\
&= \eta(\bigoplus_{j=1}^{m} f(r_{i_j}) \odot (\beta(\langle \tau_{i_1} \rangle) \otimes \ldots \otimes \beta(\tilde{\sigma}_{i_j}) \otimes \ldots \otimes \beta(\langle \tau_{i_m} \rangle)))
\end{aligned}$$

Letzteres gilt, da $\beta(\langle \tilde{\tau}_{i_j}, \hat{\tau}_{i_j} \rangle) = \beta(\langle \tilde{\tau}_{i_j} \rangle) \otimes \beta(\langle \hat{\tau}_{i_j} \rangle)$. Damit gilt auch $\beta(\langle \tau_{i_j} \rangle) = f(r_{i_j}) \odot \beta(\tilde{\sigma}_{i_j})$. Da nun zusätzlich $\beta(\langle \tau_k \rangle) = 1$, für $k \notin \{i_1, \ldots, i_m\}$, gilt nach Korollar 3.33 insgesamt:

$$\begin{aligned}
\alpha(\psi(\sigma)) &= \eta(\bigoplus_{j=1}^{m} f(r_{i_j}) \odot (\beta(\langle \tau_{i_1} \rangle) \otimes \ldots \otimes \beta(\tilde{\sigma}_{i_j}) \otimes \ldots \otimes \beta(\langle \tau_{i_m} \rangle))) \\
&= \eta(\beta(\langle \tau_1 \rangle \otimes \ldots \otimes \beta(\langle \tau_n \rangle))) \\
&= \eta(\beta(\sigma))
\end{aligned}$$

□

Zur Berechnung des Gesamtgewichtes einer Menge von Pfaden, die durch eine Hecke beschrieben werden, brauchen wir also lediglich das Gewicht der Hecke unter der Projektionsfunktion zu betrachten. Seien im folgenden $C \subseteq Conf$ eine reguläre Menge von Konfigurationen und $c \in Conf$ eine einzelne Konfiguration. Zusammen mit der in Satz 3.9 gezeigten Aussage, dass die Menge der erreichenden Hecken von c und in die Menge C,

genau die Menge der erreichenden Pfade beschreibt, folgt dann, dass wir zur Berechnung des Gewichtes der erreichenden Pfade lediglich das Gewicht der Menge der erreichenden Hecke unter der Projektionsfunktion bestimmen müssen.

Satz 3.40. *Es gilt* $\alpha(Paths(c, C)) = \eta(\beta(Hedges(c, C)))$.

Beweis. Nach Satz 3.9 gilt $\psi(Hedges(c, C)) = Paths(c, C)$, damit folgt dann:

$$\begin{aligned}
\alpha(Paths(c,C)) &= \alpha(\psi(Hedges(c,C))) \\
&= \alpha(\bigcup\{\psi(\sigma) \mid \sigma \in Hedges(c,C)\}) \\
&= \bigoplus\{\alpha(\psi(\sigma)) \mid \sigma \in Hedges(c,C)\} \\
&\overset{\text{Lemma 3.39}}{=} \bigoplus\{\eta(\beta(\sigma)) \mid \sigma \in Hedges(c,C)\} \\
&= \eta(\bigoplus\{\beta(\sigma) \mid \sigma \in Hedges(c,C)\}) \\
&= \eta(\beta(Hedges(c,C)))
\end{aligned}$$

□

Wir haben bis jetzt also festgestellt, dass wir das Gewichts der erreichenden Pfade zu einer Konfiguration und einer Menge von Zielkonfigurationen durch Berechnung des Gewichtes der erreichenden Hecken bestimmen können. Für die Menge der erreichenden Hecken haben wir in Abschnitt 3.2 gezeigt, dass man sie aus Klassen von Popsequenzen konstruieren kann. Für diese existiert nun eine Beschreibung in Form eines Ungleichungssystems. Zur Berechnung der Abstraktion der Menge der erreichenden Hecken, berechnet man daher nun zuerst die Gewichte der einzelnen Klassen von Popsequenzen. Dazu definieren wir, analog zum Ungleichungssystem zur Beschreibung der Klassen von Popsequenzen, ein Ungleichungssystem über dem Verband der Gewichte, dass die Gewichte der einzelnen Klassen beschreibt. Wir betrachten die in Definition 3.24 beschriebene Konstruktion des ursprünglichen Ungleichungssystems und wandeln diese zur Konstruktion eines neuen Ungleichungssystems ab.

Definition 3.41. Basierend auf dem in Definition 3.24 erzeugten Ungleichungssystem L über dem Verband $(\mathcal{P}(Hedges^1), \cup)$, sei ein neues Ungleichungssystem $L^\#$ über dem Verband $(E, \oplus)$ wie folgt definiert:

- Die Ordnung $\supseteq$ wird auf die im Verband der Gewichte gegebene Ordnung $\sqsubseteq$ abgebildet.
- Die konstante Menge $\{\langle[]\rangle\}$ wird auf die Konstante 1 abgebildet.
- Der Operator $[r](\cdot)$ zur Konstruktion von Hecken denen die Transition r vorangestellt wird, wird durch die Operation $f(r) \odot \cdot$ ersetzt.
- Der Operator $[r](\cdot, \cdot)$ zur Konstruktion einer Hecke aus zwei Hecken durch Voranstellen der einen neuen Stack erzeugenden Transition r und anhängen der Hecken als Äste wird durch die Operation $f(r) \odot (\cdot \otimes \cdot)$ ersetzt.

- Der Konkatenationsoperator $\cdot;\cdot$ wird durch den Konkatenationsoperator der Gewichte $\cdot \odot \cdot$ ersetzt.

Insgesamt erhält man für die fünf verschiedenen Arten von Ungleichungen dann:

(INIT) $L[t] \supseteq \{\langle[]\rangle\}$ wird zu $L^{\#}[t] \sqsubseteq 1$

(RETURN) $L[t] \supseteq [r](\{\langle[]\rangle\})$ wird zu $L^{\#}[t] \sqsubseteq f(r)$

(STEP) $L[t] \supseteq [r](L[\tilde{t}])$ wird zu $L^{\#}[t] \sqsubseteq f(r) \odot L^{\#}[\tilde{t}]$

(CALL) $L[t] \supseteq [r](L[\tilde{t}_1]; L[\tilde{t}_2])$ wird zu $L^{\#}[t] \sqsubseteq f(r) \odot L^{\#}[\tilde{t}_1] \odot L^{\#}[\tilde{t}_2]$

(SPAWN) $L[t] \supseteq [r](L[\hat{t}], L[\tilde{t}])$ wird zu $L^{\#}[t] \sqsubseteq f(r) \odot (L^{\#}[\hat{t}] \otimes L^{\#}[\tilde{t}])$

Die Operationen auf den Popsequenzen werden durch die entsprechenden Operationen auf den Gewichten ersetzt. So wird die Erstellung eines neuen Baumes aus einer Transition und einem Baum, durch die Verknüpfung des Gewichtes der Transition mit einem Gewicht ersetzt. Die Konstruktion eines verzweigten Baumes aus einer Transition und zwei Bäumen wird durch die Verknüpfung des Gewichtes der Transition mit dem Interleaving zweier Gewichte ersetzt. Der Konkatenationsoperator zum Verbinden eines Baumes mit einer Hecke, wird durch den Verknüpfungsoperator der Gewichte ersetzt. Intuitiv wird also die Betrachtung der Bäume durch die Betrachtung der zu den Bäumen gehörenden Gewichte ersetzt.

Im folgenden sei die Menge C durch den $\mathcal{M}$-Automaten $\mathcal{A} = (S, \Sigma, \delta, s^s, F)$ beschrieben. Wir betrachten dann einen alternativen erweiterten Saturierungsalgorithmus, der, an Stelle des ursprünglichen Systems, das neue Ungleichungssystem löst und einen annotierten Automaten $(\mathcal{A}^*, E, l^{\#})$ berechnet. Das Fixpunkttheorem von Knaster-Tarski liefert die Existenz einer größten Lösung, da die auf der rechten Seite der Ungleichungen verwendeten Funktionen monoton sind. Da der Verband keine unendlich absteigenden Ketten besitzt kann die Lösung des Ungleichungssystems effektiv durch chaotische Iteration berechnet werden. Die in beiden Algorithmen konstruierten Automaten sind identisch, lediglich die berechneten Annotationen unterscheiden sich.

Um zu rechtfertigen, dass dieses Ungleichungssystem wirklich eine korrekte und präzise Beschreibung der Abstraktion der Popsequenzen liefert, verwenden wir die Technik der abstrakten Interpretation. Dazu müssen wir den in Satz 2.6 beschriebenen Zusammenhang der rechten Seiten der Ungleichungen der verschiedenen Systeme und der Abstraktionsfunktion β zeigen. Betrachten wir dazu folgendes Lemma:

Lemma 3.42. *Es gilt:*

1. $\beta(\{\langle[]\rangle\}) = 1$
2. $\beta([r](\tilde{M})) = f(r) \odot \beta(\tilde{M})$, *für* $\tilde{M} \subseteq Hedges^1$
3. $\beta(\tilde{M}_1 \triangleright \tilde{M}_2) = \beta(\tilde{M}_1) \otimes \beta(\tilde{M}_2)$, *für* $\tilde{M}_1 \subseteq Hedges, \tilde{M}_2 \subseteq Hedges^1$
4. $\beta(\sigma; \langle\tau\rangle) = \beta(\sigma) \odot \beta(\langle\tau\rangle)$ *für* $\sigma \in Hedges, \langle\tau\rangle \in Hedges^1$
5. $\beta(\tilde{M}_1; \tilde{M}_2) = \beta(\tilde{M}_1) \odot \beta(\tilde{M}_2)$, *für* $\tilde{M}_1 \subseteq Hedges, \tilde{M}_2 \subseteq Hedges^1$

6. $\beta([r](\hat{M}, \tilde{M})) = f(r) \odot (\beta(\hat{M}) \otimes \beta(\tilde{M}))$, *für* $\hat{M}, \tilde{M} \subseteq Hedges^1$

Beweis. zu 1.: Gilt nach Definition von β.

zu 2.: Es gilt:

$$\begin{aligned}
\beta([r](\tilde{M})) &= \beta(\{\langle [r](\tilde{\tau})\rangle \mid \langle\tilde{\tau}\rangle \in \tilde{M}\}) \\
&= \bigoplus\{\beta(\langle [r](\tilde{\tau})\rangle) \mid \langle\tilde{\tau}\rangle \in \tilde{M}\} \\
&= \bigoplus\{f(r) \odot \beta(\langle\tilde{\tau}\rangle) \mid \langle\tilde{\tau}\rangle \in \tilde{M}\} \\
&= f(r) \odot \bigoplus\{\beta(\langle\tilde{\tau}\rangle) \mid \langle\tilde{\tau}\rangle \in \tilde{M}\} \\
&= f(r) \odot \beta(\tilde{M})
\end{aligned}$$

zu 3.: Es gilt:

$$\begin{aligned}
\beta(\tilde{M}_1 \triangleright \tilde{M}_2) &= \beta(\{\langle\tau_1, \ldots, \tau_n, \tau\rangle \mid \langle\tau_1, \ldots, \tau_n\rangle \in \tilde{M}_1, n \in \mathbb{N}, \langle\tau\rangle \in \tilde{M}_2\}) \\
&= \bigoplus\{\beta(\langle\tau_1, \ldots, \tau_n, \tau\rangle) \mid \langle\tau_1, \ldots, \tau_n\rangle \in \tilde{M}_1, \langle\tau\rangle \in \tilde{M}_2\} \\
&= \bigoplus\{\beta(\langle\tau_1\rangle) \otimes \ldots \otimes \beta(\langle\tau_n\rangle) \otimes \beta(\langle\tau\rangle) \mid \langle\tau_1, \ldots, \tau_n\rangle \in \tilde{M}_1, \langle\tau\rangle \in \tilde{M}_2\} \\
&= \bigoplus\{\beta(\langle\tau_1, \ldots, \tau_n\rangle) \otimes \beta(\langle\tau\rangle) \mid \langle\tau_1, \ldots, \tau_n\rangle \in \tilde{M}_1, \langle\tau\rangle \in \tilde{M}_2\} \\
&= \beta(\tilde{M}_1) \otimes \beta(\tilde{M}_2)
\end{aligned}$$

zu 4.: Sei $\sigma = \langle\tau_1, \ldots, \tau_n\rangle$, dann gilt $\sigma; \langle\tau\rangle = \langle\tau_1, \ldots, \tau_n; \tau\rangle$. Damit gilt dann $\beta(\sigma; \langle\tau\rangle) = \beta(\langle\tau_1\rangle \otimes \ldots \otimes \beta(\langle\tau_{n-1}\rangle) \otimes \beta(\langle\tau_n; \tau\rangle)$. Zeige nun, dass $\beta(\langle\tau_n; \tau\rangle) = \beta(\langle\tau_n\rangle) \odot \beta(\langle\tau\rangle)$, dann gilt auch

$$\begin{aligned}
\beta(\sigma; \langle\tau\rangle) &= \beta(\langle\tau_1\rangle) \otimes \ldots \otimes \beta(\langle\tau_{n-1}\rangle) \otimes \beta(\langle\tau_n; \tau\rangle) \\
&= \beta(\langle\tau_1\rangle) \otimes \ldots \otimes \beta(\langle\tau_{n-1}\rangle) \otimes (\beta(\langle\tau_n\rangle) \odot \beta(\langle\tau\rangle)) \\
&= (\beta(\langle\tau_1\rangle) \otimes \ldots \otimes \beta(\langle\tau_{n-1}\rangle) \otimes \beta(\langle\tau_n\rangle)) \odot \beta(\langle\tau\rangle) \\
&= \beta(\sigma) \odot \beta(\langle\tau\rangle).
\end{aligned}$$

Zeige die Aussage durch Strukturelle Induktion über den Aufbau von τ_n:

1. $\tau_n = []$

   Dann gilt $\beta(\langle\tau_n; \tau\rangle) = \beta(\langle\tau\rangle)$. Außerdem gilt $\beta(\langle\tau_n\rangle) = 1$ und damit $\beta(\langle\tau_n\rangle) \odot \beta(\langle\tau\rangle) = \beta(\langle\tau\rangle)$, also gilt die Behauptung.
2. $\tau_n = [r](\tilde{\tau}_n)$

   Dann gilt $\beta(\langle\tau_n; \tau\rangle) = \beta(\langle [r](\tilde{\tau}_n; \tau)\rangle) = f(r) \odot \beta(\langle\tilde{\tau}_n; \tau\rangle)$. Nach Induktionsvoraussetzung gilt dann $\beta(\langle\tilde{\tau}_n; \tau\rangle) = \beta(\langle\tilde{\tau}_n\rangle) \odot \beta(\langle\tau\rangle)$ und damit auch $f(r) \odot \beta(\langle\tilde{\tau}_n; \tau\rangle) = f(r) \odot \beta(\langle\tilde{\tau}_n\rangle) \odot \beta(\langle\tau\rangle) = \beta(\langle\tau_n\rangle) \odot \beta(\langle\tau\rangle)$.

3. $\tau_n = [r](\hat{\tau}_n, \tilde{\tau}_n)$

Dann gilt $\beta(\langle \tau_n; \tau \rangle) = \beta(\langle [r](\hat{\tau}_n, \tilde{\tau}_n; \tau) \rangle) = f(r) \odot (\beta(\langle \hat{\tau}_n \rangle) \otimes \beta(\langle \tilde{\tau}_n; \tau \rangle))$. Nach Induktionsvoraussetzung gilt dann $\beta(\langle \tilde{\tau}_n; \tau \rangle) = \beta(\langle \tilde{\tau}_n \rangle) \odot \beta(\langle \tau \rangle)$ und damit auch $f(r) \odot (\beta(\langle \hat{\tau}_n \rangle) \otimes \beta(\langle \tilde{\tau}_n; \tau \rangle)) = f(r) \odot (\beta(\langle \hat{\tau}_n \rangle) \otimes (\beta(\langle \tilde{\tau}_n \rangle) \odot \beta(\langle \tau \rangle))) = f(r) \odot (\beta(\langle \hat{\tau}_n \rangle) \otimes \beta(\langle \tilde{\tau}_n \rangle)) \odot \beta(\langle \tau \rangle) = \beta(\langle \tau_n \rangle) \odot \beta(\langle \tau \rangle)$.

Es gilt:

$$\begin{aligned}
\beta(\tilde{M}_1; \tilde{M}_2) &= \beta(\{\langle \tau_1, \ldots, \tau_n \rangle; \langle \tau \rangle \mid \langle \tau_1, \ldots, \tau_n \rangle \in \tilde{M}_1, n \in \mathbb{N}, \langle \tau \rangle \in \tilde{M}_2\}) \\
&= \bigoplus \{\beta(\langle \tau_1, \ldots \tau_n \rangle; \langle \tau \rangle) \mid \langle \tau_1, \ldots, \tau_n \rangle \in \tilde{M}_1, n \in \mathbb{N}, \langle \tau \rangle \in \tilde{M}_2\} \\
&= \bigoplus \{\beta(\langle \tau_1, \ldots \tau_n \rangle) \odot \beta(\langle \tau \rangle) \mid \langle \tau_1, \ldots, \tau_n \rangle \in \tilde{M}_1, n \in \mathbb{N}, \langle \tau \rangle \in \tilde{M}_2\} \\
&= \bigoplus \{\beta(\langle \tau_1, \ldots, \tau_n \rangle) \mid \langle \tau_1, \ldots, \tau_n \rangle \in \tilde{M}_1\} \odot \bigoplus \{\beta(\langle \tau \rangle) \mid \langle \tau \rangle \in \tilde{M}_2\} \\
&= \beta(\tilde{M}_1) \odot \beta(\tilde{M}_2)
\end{aligned}$$

zu 5.:zu 6.: Es gilt:

$$\begin{aligned}
\beta([r](\hat{M}, \tilde{M})) &= \beta(\{\langle [r](\hat{\tau}, \tilde{\tau}) \rangle \mid \langle \hat{\tau} \rangle \in \hat{M}, \langle \tilde{\tau} \rangle \in \tilde{M}\}) \\
&= \bigoplus \{\beta(\langle [r](\hat{\tau}, \tilde{\tau}) \rangle) \mid \langle \hat{\tau} \rangle \in \hat{M}, \langle \tilde{\tau} \rangle \in \tilde{M}\} \\
&= \bigoplus \{f(r) \odot (\beta(\langle \hat{\tau} \rangle) \otimes \beta(\langle \tilde{\tau} \rangle)) \mid \langle \hat{\tau} \rangle \in \hat{M}, \langle \tilde{\tau} \rangle \in \tilde{M}\} \\
&= f(r) \odot \bigoplus \{\beta(\langle \hat{\tau} \rangle) \otimes \beta(\langle \tilde{\tau} \rangle) \mid \langle \hat{\tau} \rangle \in \hat{M}, \langle \tilde{\tau} \rangle \in \tilde{M}\} \\
&= f(r) \odot (\bigoplus \{\beta(\langle \hat{\tau} \rangle) \mid \langle \hat{\tau} \rangle \in \hat{M}\} \otimes \bigoplus \{\beta(\langle \tilde{\tau} \rangle) \mid \langle \tilde{\tau} \rangle \in \tilde{M}\}) \\
&= f(r) \odot (\beta(\hat{M}) \otimes \beta(\tilde{M}))
\end{aligned}$$

□

Aus den Aussagen des Lemmas folgt direkt die Erfüllung der Voraussetzung des Transferlemmas und man kann folgern, dass das neue Ungleichungssystem eine korrekte und präzise Beschreibung der Abstraktion der Klassen von Popsequenzen liefert. Seien also im folgenden $(\mathcal{A}^*, \mathcal{P}(Hedges^1), l)$ und $(\mathcal{A}^*, E, l^\#)$ die durch den ursprünglichen und alternativen Saturierungsalgorithmus berechneten annotierten Automaten.

Satz 3.43. *Es gilt $\beta(l(t)) = l^\#(t)$, für $t \in \bar{\delta}_{stack}$.*

Beweis. Folgt direkt aus Lemma 3.42 und dem Transferlemma 2.6 für abstrakte Interpretation. □

Wir haben also gezeigt, dass das neue Ungleichungssystem eine korrekte und präzise Beschreibung der Abstraktion der Popsequenzen berechnet. In Satz 3.28 wurde gezeigt, dass man die erreichenden Hecken aus den durch das Ungleichungssystem beschriebenen Klassen von Popsequenzen konstruieren kann, indem man die Klassen verknüpft, die zu Transitionen entlang der akzeptierenden Pfade der Konfiguration c im saturierten Automaten gehören. Basierend hierauf definieren wir nun eine Funktion θ, die analog zur Funktion π, beim Auslesen einer Konfiguration aus dem Automaten aus den Gewichten der angetroffenen Klassen von Popsequenzen ein Gesamtgewicht konstruiert.

Definition 3.44. Wir definieren wir zu einem Lauf $\phi \in Runs_{\mathcal{A}^*}$ den Wert:

$$\theta(\phi) = \begin{cases} 1 & \text{für } \phi = [\![s]\!], s \in S \\ \theta(\tilde{\phi}) \otimes 1 & \text{für } \phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!], (s, \lambda, \tilde{s}) \in \bar{\delta}_{separator} \\ \theta(\tilde{\phi}) & \text{für } \phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!], (s, \lambda, \tilde{s}) \in \bar{\delta}_{state} \\ \theta(\tilde{\phi}) \odot l^{\#}((s, \lambda, \tilde{s})) & \text{für } \phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!], (s, \lambda, \tilde{s}) \in \bar{\delta}_{stack} \end{cases}.$$

Es wird, analog zur Definition des neuen Ungleichungssystems auf Basis des ursprünglichen Systems, die Konkatenation von Popsequenzen durch die Verknüpfung der zugehörigen Gewichte ersetzt. Da der Verknüpfungsoperator sich wie der Konkatenationsoperator verhält, ist auch im Fall, dass das schon konstruierte Gewicht eine Hecke mit mehreren Bäumen beschreibt das korrekte Verhalten gewährleistet. Der Operator, der in einer Hecke einen neuen Baum leeren beginnt, wird durch den Interleavingoperator ersetzt und der leere Baum erhält das neutrale Gewicht. Ein neuer Baum in der Hecke wird parallel zu allen vorherigen Bäumen ausgeführt und daher muss auch sein Gewicht parallel zu dem vorherigen Gewicht betrachtet werden. Für diese Definition können wir nun zeigen, dass das so gewonnene Gesamtgewicht eines Laufes genau der Abstraktion der Menge von Hecken entspricht, die wir durch Auslesen des Laufes mit Hilfe der Funktion π gewinnen.

Lemma 3.45. *Für einen Lauf $\phi \in Runs_{\mathcal{A}^*}$ gilt $\theta(\phi) = \beta(\pi(\phi))$.*

Beweis. Zeige die Aussage durch Induktion über die Länge von ϕ[4]:

1. $|\phi| = 0$

   Dann gilt $\phi = [\![s]\!]$, für ein $s \in S$, und damit $\pi(\phi) = \{\langle[]\rangle\}$. Nach Definition ist dann $\beta(\pi(\phi)) = 1$, sowie $\theta(\phi) = 1$ und damit gilt die Behauptung.

2. $|\phi| > 0$

   In diesem Fall gilt $\phi = \tilde{\phi}; [\![s, \lambda, \tilde{s}]\!]$, für $\tilde{\phi} \in Runs, (s, \lambda, \tilde{s}) \in \bar{\delta}$, mit $s, \tilde{s} \in S, \lambda \in \Sigma$.

   Unterscheide drei Fälle:

[4]Die Länge eines Laufes ϕ ist dabei definiert als $|\phi| = |\phi^w|$. Da keine ϵ-Kanten existieren, entspricht dies der Anzahl der durchlaufenen Kanten.

1. Fall: $(s, \lambda, \tilde{s}) \in \delta_{separator}$
   Dann gilt $\pi(\phi) = \pi(\tilde{\phi}) \triangleright \{\langle[]\rangle\}$ und damit nach Lemma 3.42 $\beta(\pi(\phi)) = \beta(\pi(\tilde{\phi})) \otimes 1$. Nach Induktionsvoraussetzung gilt schon $\beta(\pi(\tilde{\phi})) = \theta(\tilde{\phi})$ und damit die Behauptung, denn $\theta(\phi) = \theta(\tilde{\phi}) \otimes 1$.
2. Fall: $(s, \lambda, \tilde{s}) \in \delta_{state}$
   Dann gilt $\pi(\phi) = \pi(\tilde{\phi})$ und damit $\beta(\pi(\phi)) = \beta(\pi(\tilde{\phi}))$. Nach Induktionsvoraussetzung und Definition gilt dann auch $\beta(\pi(\phi)) = \theta(\tilde{\phi}) = \theta(\phi)$.
3. Fall: $(s, \lambda, \tilde{s}) \in \bar{\delta}_{stack}$
   Sei $t = (s, \lambda, \tilde{s})$. Dann gilt $\pi(\phi) = \pi(\tilde{\phi}); l(t)$ und damit nach Lemma 3.42 und Satz 3.43 auch $\beta(\pi(\phi)) = \beta(\pi(\tilde{\phi})) \odot l^{\#}(t)$. Nach Induktionsvoraussetzung und Definition gilt dann wiederum $\beta(\pi(\phi)) = \theta(\tilde{\phi}) \odot l^{\#}(t) = \theta(\phi)$.

□

Mit Hilfe der auf Gewichten arbeitenden Auslesefunktion θ können wir dann, analog zur Auslesefunktion π, das Gesamtgewicht der erreichenden Hecken einer Konfiguration c definieren:

Definition 3.46. Wir definieren zu einer Konfiguration $c \in Conf$ den Wert:

$$\theta(c) = \bigoplus\{\theta(\phi) \mid \phi \in Accept(c)\}.$$

Für dieses Gesamtgewicht gilt dann die Behauptung, dass es genau der von uns erwarteten Abstraktion der Menge der erreichenden Hecken entspricht. Es gilt also, dass der von uns einmal berechnete Automat, analog wie im Fall der WPDS, die Abstraktion der erreichenden Hecken für beliebige Konfigurationen enthält.

Satz 3.47. *Es gilt* $\theta(c) = \beta(Hedges(c, C))$.

Beweis. Es gilt:

$$\begin{aligned}
\beta(Hedges(c, C)) &\overset{\text{Satz 3.28}}{=} \beta(\pi(c)) \\
&= \beta(\bigcup\{\pi(\phi) \mid \phi \in Accept(c)\}) \\
&= \bigoplus\{\beta(\pi(\phi)) \mid \phi \in Accept(c)\} \\
&\overset{\text{Lemma 3.45}}{=} \bigoplus\{\theta(\phi) \mid \phi \in Accept(c)\} \\
&= \theta(c)
\end{aligned}$$

□

Um nun das gewünschte Gewicht der erreichenden Pfade zu berechnen, braucht man, wie oben schon gezeigt, lediglich die Projektionsfunktion auf das durch den Automaten erhaltene Gewicht der Menge der erreichenden Hecken anwenden.

Satz 3.48. *Es gilt* $\eta(\theta(c)) = \delta(c)$.

Beweis. Es gilt:

$$\begin{aligned} \eta(\theta(c)) &\overset{\text{Satz 3.47}}{=} \eta(\beta(Hedges(c, C))) \\ &\overset{\text{Satz 3.40}}{=} \alpha(Paths(c, C)) \\ &= \delta(c) \end{aligned}$$

□

Wir können also insgesamt zu einer Menge C von Konfigurationen einen Automaten berechnen, aus dem wir durch auslesen von Konfigurationen das Gewicht der zu der Konfiguration gehörenden Menge der erreichenden Hecken in die Menge C erhalten. Dieses können wir dann durch einfache Anwendung der Projektionsfunktion in das gewünschte Gewicht der Menge der erreichenden Pfade transformieren.

KAPITEL 4

Anwendungsbeispiele

Im folgenden Kapitel werden wir zwei Anwendungen des in Kapitel 3 beschriebenen Verfahrens betrachten. Die erste Anwendung, in Abschnitt 4.1, behandelt das Feld der Bitvektor-Probleme. Wir konstruieren die Gewichte für die Lösung von Bitvektor-Probleme und demonstrieren, wie man durch richtige Verwendung des Modells klassische Datenflussinformationen gewinnen kann. Desweiteren zeigen wir, wie man durch spezifischere Anfragen diese Informationen weiter differenzieren kann. Die zweite Anwendung, in Abschnitt 4.2, betrachtet Gewichte, die die Berechnung des Gewichtes des kürzesten Pfades eines DPN ermöglichen. Dabei wird jeder Regel eines DPN eine ganzzahlige positive Länge zugewiesen. Als Ergebnis erhält man das Gewicht des kürzesten Pfades. Die Länge eines Pfades ergibt sich dabei durch Addition der Längen der einzelnen Transitionen des Pfades. Dies ist ein Beispiel dafür, dass auch Datenflussanalysen mit unendlichem Wertebereich, in diesem Fall die natürlichen Zahlen, in unserem Modell berechnet werden können.

4.1 Bitvektor-Probleme

Als erstes Beispiel einer Datenflussanalyse betrachten wir wiederum die Klasse der Bitvektor-Probleme. Basierend auf den in Abschnitt 2.2 eingeführten Halbringen, für sichere und mögliche Vorwärts- und Rückwärtsanalyse mit Hilfe von WPDS. Diese haben die Form $(\{KILL, ID, GEN, ZERO\}, \oplus, \odot, ZERO, ID)$. Wir konstruieren nun zunächst erweiterte Halbringe $(E, \oplus, \odot, \otimes, ZERO, ID, \eta)$, für sichere und mögliche Vorwärts- und Rückwärtsanalyse mit Hilfe von WDPN.

Da sich die Analysen sicherer und möglicher Eigenschaften lediglich in der Reihenfolge der Ordnung von $KILL$ und GEN unterscheiden, verwenden wir $\bot$ für das kleinere und $\top$ für das größere der beiden Elemente und behandeln diese Fälle gemeinsam. Dadurch reduzieren sich die zu betrachtenden Fälle auf Vorwärts- und Rückwärtsanalysen.

Zur Konstruktion der erweiterten Halbringe gehen wir von den bekannten Halbringen $(\{KILL, ID, GEN, ZERO\}, \oplus, \odot, ZERO, ID)$ aus und suchen einen Interleavingoperator, der die nötigen Eigenschaften erfüllt. Wenn zur Gewährleistung dieser Eigenschaften neue Elemente eingeführt werden müssen, werden die Operatoren $\oplus, \odot, \otimes$ und die Projektionsfunktion η dementsprechend erweitert.

Als Ansatzpunkt für einen Interleavingoperator verwenden wir ein in [SS00] gezeigtes Ergebnis für die Klasse der $KILL/GEN$ Analysen. In solchen Analysen gilt für die Gewichte d_1, d_2 von zwei Pfaden, dass sich das kombinierte Gewicht aller Pfade, die durch Interleaving entstehen, als $(d_1 \odot d_2) \oplus (d_2 \odot d_1)$ berechnen lässt. Da Bitvektor-Probleme ein Spezialfall der $KILL/GEN$ Analysen sind, und demnach gleiches Verhalten zeigen, können wir auf diesem Ergebnis aufbauen. Für die Definition eines Interleavingoperators können wir uns also auf die Betrachtung der Gesamtgewichte der betrachteten Pfade beschränken und brauchen keine weiteren Informationen über deren genauen Aufbau. Dies lässt sich auch direkt durch die bei Bitvektor-Problemen betrachteten Transferfunktionen begründen. Hierbei handelt es sich entweder um die Identität oder eine konstante Funktion. Das Gewicht eines Pfades wird nur durch die zuletzt ausgeführte konstante Funktion bestimmt, da sie alle, durch beliebige vorherige Funktionsaufrufe berechneten, Werte mit einer Konstanten überschreibt. Deshalb ist bei der Betrachtung des Interleavings auch nur wichtig, welcher Pfad als letztes seine bestimmende Funktion ausführt und, da alle möglichen Pfade des Interleavings betrachtet werden müssen, beide Möglichkeiten zu approximieren.

In einem ersten Schritt sei der Interleavingoperator vorläufig als $f \otimes g = (f \odot g) \oplus (g \odot f)$ definiert. Das einzige Problem ist nun die im erweiterten Halbring an den Interleavingoperator gestellte Forderung, dass das Gewicht des rechten Pfades durch Konkatenation noch nachträglich verändert werden kann. Dazu betrachten wir im folgenden Vorwärts- und Rückwärts-Bitvektor-Probleme getrennt.

4.1.1 Vorwärtsanalyse

Bei der Betrachtung von Vorwärtsanalysen gilt für $f, g \in \{\bot, ID, \top\}$, dass $f \odot g = g \circ f$. Der Konkatenationsoperator entspricht also, bei Gewichten ungleich $ZERO$, der umgekehrten Funktionenverknüpfung. Betrachte nun $f, g, h \in \{\bot, ID, \top, ZERO\}$, dann muss nach Definition des erweiterten Halbrings $(f \otimes g) \odot h = f \otimes (g \odot h)$ gelten. Untersuche die folgende Fälle:

1. Fall: $f = ZERO$ oder $g = ZERO$

   Dann gilt mit der vorläufigen Definition $(f \otimes g) \odot h = ZERO = f \otimes (g \odot h)$ und damit das geforderte. Verkürzt kann man sogar $f \otimes g = f \odot g$ schreiben.

2. Fall: $f = ID$ und $g \neq ZERO$

Das Gewicht des parallel laufenden Pfades ist die Identität. Mit der vorläufigen Definition gilt $(ID \otimes g) \odot h = ((ID \odot g) \oplus (g \odot ID)) \odot h = g \odot h = (ID \odot (g \odot h)) \oplus ((g \odot h) \odot ID) = ID \otimes (g \odot h)$ und damit das geforderte. Da die Identität neutral bezüglich der Funktionenverknüpfung ist, ist es irrelevant an welcher Stelle sie ausgeführt wird. Wiederum kann man verkürzt $ID \otimes g = ID \odot g = g$ schreiben.

3. Fall: $f = \top$ und $g \neq ZERO$

Der parallel laufende Pfad wird durch die $\top$ Funktion beschrieben. Dann gilt mit der vorläufigen Definition $(\top \otimes g) \odot h = ((\top \odot g) \oplus (g \odot \top)) \odot h = ((g \circ \top) \oplus (\top \circ g)) \odot h$. Da $\top \in \{KILL, GEN\}$ und damit eine konstante Funktion ist, gilt $\top \circ g = \top$. Daraus folgt dann $(\top \otimes g) \odot h = ((g \circ \top) \oplus \top) \odot h$. Da $(g \circ \top) \in \{\bot, ID, \top\}$, gilt nach der definierten Ordnung $(g \circ \top) \sqsubseteq \top$. Die $\top$ Funktion liefert genauere Informationen als jede andere Funktion. Bei der Betrachtung der verschiedenen Reihenfolgen für das Interleaving wird also immer der Fall, in dem die $\top$ Funktion gegebenenfalls nochmal überschrieben wird, die ungenaueren Informationen liefern und damit die Approximation festlegen. Insgesamt ist dann also $(\top \otimes g) \odot h = (\top \odot g) \odot h$. Auf der anderen Seite gilt aus den gleichen Gründen $\top \otimes (g \odot h) = (\top \odot (g \odot h)) \oplus ((g \odot h) \odot \top) = (\top \odot (g \odot h)) \oplus \top = \top \odot (g \odot h)$ und damit schon Gleichheit. Auch hier kann man verkürzt $\top \otimes g = \top \odot g$ schreiben.

4. Fall: $f = \bot$ und $g \neq ZERO$

Das Gewicht des parallel laufenden Pfades ist die $\bot$ Funktion. Es gilt, mit der vorläufigen Definition, und der gleichen Argumentation wie im vorherigen Fall $(\bot \otimes g) \odot h = ((\bot \odot g) \oplus (g \odot \bot)) \odot h = ((\bot \odot g) \oplus \bot) \odot h = \bot \odot h$. In diesem Fall liefert die $\bot$ Funktion immer die ungenaueren Informationen und legt daher, bei der Kombination mit einer beliebigen Funktion, die Approximation fest. Auf der anderen Seite ist $\bot \otimes (g \odot h) = (\bot \odot (g \odot h)) \oplus ((g \odot h) \odot \bot) = (\bot \odot (g \odot h)) \oplus \bot = \bot$. In diesem Fall muss also noch zusätzliche Arbeit geleistet werden, da die parallel ausgeführte Funktion $\bot$ das nachträglich angehängte h nicht überschreiben kann. Deswegen führen wir ein zusätzliches Gewicht $\underline{\bot}$ ein, das für ein parallel ausgeführtes $\bot$ steht, das immer als letztes ausgeführt wird. Es führt also die Auswirkungen einer parallel ausgeführten $\bot$ Funktion mit, die zu jedem Zeitpunkt das Ergebnis zerstören kann, insbesondere am Ende, wo es die größten Auswirkungen hat. Wir definieren dafür $\underline{\bot} \odot g = g \odot \underline{\bot} = \underline{\bot}$ und $ZERO \odot \underline{\bot} = \underline{\bot} \odot ZERO = ZERO$, das Gewicht kann sich also an Funktionen vorbeischieben und diese überschreiben. In der Ordnung gelte $\underline{\bot} \sqsubseteq \bot$, da $\underline{\bot}$ es zusätzlich jedes andere Gewicht überschreibt. Dann können wir das Interleaving $\bot \otimes g = \underline{\bot} \odot g = \underline{\bot}$ neu definieren. Es gelte $\underline{\bot} \otimes g = \bot \otimes g$, da sich eine in einem parallel laufenden Pfad parallel ausgeführte $\bot$ Funktion auf den aktuellen Pfad wie eine einfache parallel laufende $\bot$ Funktion auswirkt. Dann gelten für $\eta(\underline{\bot}) = \bot$ auch die Eigenschaften der Projektionsfunktion.

Insgesamt erhalten wir für Vorwärts-Bitvektor-Analysen also einen erweiterten beschränkten idempotenten Halbring $(\{ZERO, \top, ID, \bot, \underline{\bot}\}, \odot, \oplus, \otimes, 0, 1, \eta)$, mit:

$\odot$	0	$\top$	1	$\bot$	$\underline{\bot}$
0	0	0	0	0	0
$\top$	0	$\top$	$\top$	$\bot$	$\underline{\bot}$
1	0	$\top$	1	$\bot$	$\underline{\bot}$
$\bot$	0	$\top$	$\bot$	$\bot$	$\underline{\bot}$
$\underline{\bot}$	0	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$	$\underline{\bot}$

Dabei gilt $\underline{\bot} \sqsubseteq \bot \sqsubseteq ID \sqsubseteq \top \sqsubseteq ZERO$. Für das Interleaving gilt $d_1 \otimes d_2 = d_1 \odot d_2$ für $d_1 \notin \{\bot, \underline{\bot}\}$, sonst $d_1 \otimes d_2 = \underline{\bot} \odot d_2$. Zudem ist $\eta(d) = d$ für $d \neq \underline{\bot}$, sonst $\eta(\underline{\bot}) = \bot$. Durch Nachrechnen lässt sich leicht zeigen, dass der Halbring alle geforderten Eigenschaften erfüllt.

Betrachten wir als Beispiel das in Abbildung 2.9 in Abschnitt 2.3 vorgestellte System von Flussgraphen $\mathcal{G}$ und die dazu gehörende Umsetzung in ein DPN $\mathcal{M}$. Dabei untersuchen wir, wie schon in vorherigen Beispielen, die Verfügbarkeit des Ausdrucks $x - 1$ am Programmpunkt e_{rec}. Als Zielmenge betrachten wir die Menge $(pN_{\mathcal{G}}^*\#)^* pe_{rec} N_{\mathcal{G}}^* (\#pN_{\mathcal{G}}^*)^*$, die die Menge der erreichbaren Konfigurationen an Programmpunkt e_{rec} umfasst, dargestellt durch den Automaten in Abbildung 2.10. Es handelt sich um eine sichere Vorwärtsanalyse, es gilt also $\top = GEN$, $\bot = KILL$ und$\underline{\bot} = \underline{KILL}$.

Abbildung 4.1 zeigt das Ergebnis der Anwendung des Verfahrens. Der berechnete annotierte Automat liefert für die Startkonfiguration pe_{main} das Ergebnis $\underline{KILL}$. Die Anwendung der Projektionsfunktion liefert $\eta(\underline{KILL}) = KILL$. Dies ist offensichtlich korrekt, da Pfade existieren, die von e_{main} nach e_{rec} laufen, und auf jedem dieser Pfade die erste Kante genau einmal durchlaufen und damit der neue Thread erzeugt wird. Dieser kann zu jedem Zeitpunkt seine einzige Transition ausführen und damit die Verfügbarkeit des Ausdrucks zerstören.

Betrachtet man hingegen gezielt die Menge $(p(N_{\mathcal{G}} \setminus \{r_{spawn}\})^*\#)^* pe_{rec} N_{\mathcal{G}}^* (\#pN_{\mathcal{G}}^*)^*$, die die Menge der erreichbaren Konfigurationen an Programmpunkt e_{rec} umfasst, in denen der erzeugte Thread seinen Schritt noch nicht gemacht hat, erhält man das Ergebnis GEN.

4.1.2 Rückwärtsanalyse

Bei der Betrachtung von Rückwärtsanalysen gilt für $f, g \in \{\bot, ID, \top\}$, dass $f \odot g = f \circ g$. Der Konkatenationsoperator entspricht also, bei Gewichten ungleich $ZERO$, der Funktionenverknüpfung. Betrachte nun $f, g, h \in \{\bot, ID, \top, ZERO\}$, dann muss nach Definition des erweiterten Halbrings $(f \otimes g) \odot h = f \otimes (g \odot h)$ gelten. Untersuche die folgende Fälle:

1. Fall: $f = ZERO$ oder $g = ZERO$

   Dann gilt mit der vorläufigen Definition $(f \otimes g) \odot h = ZERO = f \otimes (g \odot h)$ und damit das geforderte. Verkürzt kann man sogar $f \otimes g = f \odot g$ schreiben.

2. Fall: $f = ID$ und $g \neq ZERO$

Das Gewicht des parallel laufenden Pfades ist die Identität. Mit der vorläufigen Definition gilt $(ID \otimes g) \odot h = ((ID \odot g) \oplus (g \odot ID)) \odot h = g \odot h = (ID \odot (g \odot h)) \oplus ((g \odot h) \odot ID) = ID \otimes (g \odot h)$ und damit das geforderte. Da die Identität neutral bezüglich der Funktionenverknüpfung ist, ist es irrelevant an welcher Stelle sie ausgeführt wird. Wiederum kann man verkürzt $ID \otimes g = ID \odot g = g$ schreiben.

3. Fall: $f = \bot$ und $g \neq ZERO$

Der parallel laufende Pfad wird durch die $\top$ Funktion beschrieben. Dann gilt mit der vorläufigen Definition $(\bot \otimes g) \odot h = ((\bot \odot g) \oplus (g \odot \bot)) \odot h = ((\bot \circ g) \oplus (g \circ \bot)) \odot h$. Da $\bot \in \{KILL, GEN\}$ und damit eine konstante Funktion ist, gilt $\bot \circ g = \bot$. Daraus folgt dann $(\bot \otimes g) \odot h = (\bot \oplus (g \circ \bot)) \odot h$. Da $(g \circ \bot) \in \{\bot, ID, \top\}$, gilt nach der definierten Ordnung $(g \circ \bot) \sqsupseteq \bot$. Die $\bot$ Funktion liefert ungenauere Informationen als jede andere Funktion. Bei der Betrachtung der verschiedenen Reihenfolgen für das Interleaving wird also immer der Fall, in dem die $\bot$ Funktion die gesamte Funktion bestimmt, die ungenaueren Informationen liefern und damit die Approximation festlegen. Insgesamt ist dann also $(\bot \otimes g) \odot h = \bot \odot h = \bot$. Auf der anderen Seite gilt aus den gleichen Gründen $\bot \otimes (g \odot h) = (\bot \odot (g \odot h)) \oplus ((g \odot h) \odot \bot) = \bot \oplus ((g \odot h) \odot \bot) = \bot$ und damit schon Gleichheit. Auch hier kann man verkürzt $\bot \otimes g = \bot \odot g = \bot$ schreiben.

4. Fall: $f = \top$ und $g \neq ZERO$

Das Gewicht des parallel laufenden Pfades ist die $\top$ Funktion. Es gilt, mit der vorläufigen Definition, und der gleichen Argumentation wie im vorherigen Fall $(\top \otimes g) \odot h = ((\top \odot g) \oplus (g \odot \top)) \odot h = (\top \oplus (g \odot \top)) \odot h = (g \odot \top) \odot h = g \odot \top$. In diesem Fall liefert die $\top$ Funktion immer die genaueren Informationen und daher legt, bei der Kombination mit einer beliebigen Funktion, die andere Funktion die Approximation fest. Auf der anderen Seite ist $\top \otimes (g \odot h) = (\top \odot (g \odot h)) \oplus ((g \odot h) \odot \top) = \top \oplus ((g \odot h) \odot \top) = ((g \odot h) \odot \top)$. In diesem Fall muss also noch zusätzliche Arbeit geleistet werden, da die parallel ausgeführte Funktion $\top$ das nachträglich angehängte h überschreibt. Deswegen führen wir ein zusätzliches Gewicht $\overline{\top}$ ein, das für ein parallel ausgeführtes $\top$ steht, das immer als erstes ausgeführt wird. Es führt also die Auswirkungen einer parallel ausgeführten $\top$ Funktion mit, die zu jedem Zeitpunkt das Ergebnis beeinflussen kann. Da die $\top$ Funktion die genauesten Informationen liefert, ist der schlechteste Fall also die Ausführung vor allen anderen Funktionen. Wir definieren dafür $\overline{\top} \odot g = g \odot \overline{\top} = g$ für $g \in \{\bot, \top, ZERO\}$ und $\overline{\top} \odot g = g \odot \overline{\top} = \overline{\top}$ sonst, das Gewicht kann sich also an Funktionen vorbeischieben und überschrieben werden. In der Ordnung gelte $ID \sqsubseteq \overline{\top} \sqsubseteq \top$, da $\overline{\top}$ genauere Informationen als ID liefert, aber einfacher als $\top$ überschrieben werden kann. Dann können wir das Interleaving $\top \otimes g = \overline{\top} \odot g$ neu definieren. Es gelte $\overline{\top} \otimes g = \top \otimes g$, da sich eine in einem parallel laufenden Pfad parallel ausgeführte $\top$ Funktion auf den aktuellen Pfad wie eine einfache parallel laufende $\top$ Funktion auswirkt. Dann gelten für $\eta(\overline{\top}) = \top$ auch die Eigenschaften der Projektionsfunktion.

Insgesamt erhalten wir für Rückwärts-Bitvektor-Analysen also einen erweiterten beschränkten idempotenten Halbring $(\{ZERO, \top, \overline{\top}, ID, \bot\}, \odot, \oplus, \otimes, 0, 1, \eta)$, mit:

$\odot$	0	$\top$	$\overline{\top}$	1	$\bot$
0	0	0	0	0	0
$\top$	0	$\top$	$\top$	$\top$	$\top$
$\overline{\top}$	0	$\top$	$\overline{\top}$	$\overline{\top}$	$\bot$
1	0	$\top$	$\overline{\top}$	1	$\bot$
$\bot$	0	$\bot$	$\bot$	$\bot$	$\bot$

sowie $\bot \sqsubseteq ID \sqsubseteq \overline{\top} \sqsubseteq \top \sqsubseteq ZERO$ und $d_1 \otimes d_2 = d_1 \odot d_2$ für $d_1 \notin \{\top, \overline{\top}\}$, sonst $d_1 \otimes d_2 = \overline{\top} \odot d_2$. Zudem $\eta(d) = d$ für $d \neq \overline{\top}$, sonst $\eta(\overline{\top}) = \top$. Durch Nachrechnen lässt sich leicht zeigen, dass dieser alle geforderten Eigenschaften erfüllt.

Mit Hilfe dieses Halbrings kann man nun zum Beispiel eine Analyse zur Überprüfung der Lebendigkeit einer Variablen formulieren. Eine Variable ist *lebendig*, wenn sie auf einem der ausgehenden Pfade verwendet wird, bevor sie überschrieben wird. Tote, nicht lebendige, Variablen enthalten demnach nur Werte, die nicht benutzt, sondern direkt wieder überschrieben werden. Zuweisungen an solche Variablen können also gelöscht werden. Es handelt sich hierbei um ein mögliches rückwärts Bitvektor-Problem, da ein ausgehender Pfad der die Variable benutzt ausreicht um die Lebendigkeit zu gewährleisten. Am Ende des Programms sind alle Variablen lebendig, da sie zur Ausgabe des Programms zählen. Diese Information wird rückwärts entlang der Pfade transformiert. Wenn eine Anweisung eine Variable verwendet, ist sie vor Ausführung der Anweisung lebendig. Wenn hingegen eine Anweisung eine Variable überschreibt und sie dabei nicht gleichzeitig verwendet, ist die Variable vor Ausführung der Anweisung tot, da diese den Wert in jedem Fall überschreibt, ohne die Variable zu betrachten.

Da unser Modell jedoch nur die ausgehende Pfade einer Konfiguration in eine Menge von Zielkonfigurationen beschreibt, lässt sich nicht ohne weiteres die gleiche Information, die man bei der klassischen Datenflussanalyse für einen Programmpunkt berechnet, gewinnen. Man betrachte als Zielmenge die Menge der Konfigurationen am Ende des Programms. Wenn kein wirklicher Endzustand existiert und es ausreicht Pfade zu einer beliebigen nachfolgenden Konfiguration zu betrachten, kann man auch die Menge aller Konfigurationen betrachten. Dann müsste man für jede Konfiguration, die sich am gewünschten Programmpunkt befindet und gleichzeitig von der Startkonfiguration erreichbar ist, das Gewicht bestimmen und diese dann kombinieren, um das Gewicht über alle ausgehenden Pfade des Programmpunktes zu berechnen. Da aber in der Regel die Menge der erreichbaren Konfigurationen, für eine Konfiguration, in einem DPN nicht einmal regulär ist [MO06], müssen dazu noch weitere Techniken entwickelt werden.

Als Beispiel, für die gegenwärtigen Fähigkeiten von WDPN im Bezug auf Rückwärtsanalysen, betrachten wir wiederum das in Abbildung 2.9 in Abschnitt 2.3 vorgestellte System von Flussgraphen $\mathcal{G}$ und die dazu gehörende Umsetzung in ein DPN $\mathcal{M}$. Wir untersuchen die Lebendigkeit der Variable y. Als Zielmenge betrachten wir die Menge $\{pr_{spawn}\#pr_{main}\}$, die nur aus der Konfiguration besteht, in der beide Threads ihren Endpunkt erreicht haben.

Abbildung 4.2 zeigt den Automaten zur Beschreibung von $\{pr_{spawn} \# pr_{main}\}$ und das Ergebnis der Anwendung des Verfahrens. Der berechnete annotierte Automat liefert für die Startkonfiguration pe_{main} das Ergebnis $KILL$. Dies ist offensichtlich korrekt, da Pfade existieren, die von e_{main} zum Ende des Programms laufen, und auf jedem dieser Pfade werden zuerst Kanten durchlaufen, die die Variable y initialisieren, bevor Kanten erreicht werden, die die Variable benutzen. Nun ist es wenig sinnvoll die Lebendigkeit einer Variablen am Startpunkt zu untersuchen, da keine Anweisungen davor existieren, die man eliminieren könnte. Man kann aus dieser Information jedoch die Tatsache ableiten, dass das Programm in der Hinsicht sicher ist, dass ein gegebenenfalls unsicherer initialer Wert von y keinen Einfluss auf die Ausführung des Programms hat.

4.2 Kürzeste Pfade

Ein Beispiel für eine Datenflussanalyse mit unendlichem Wertebereich ist die Bestimmung der Länge des kürzesten Pfades der einen Programmpunkt erreicht. Dabei wird jeder Regel des DPN eine nichtnegative ganzzahlige Länge zugewiesen und der kürzeste Pfad ist derjenige mit der kleinsten Summe aller Längen, der auf ihm angewendeten Regeln.

Der verwendete erweiterte Halbring $(\mathbb{N} \cup \{0, \infty\}, +, min, +, \infty, 0, id)$ basiert auf dem in [RSJM05] verwendeten *Dijkstra Halbring* $(\mathbb{N} \cup \{0, \infty\}, +, min, \infty, 0)$ zur Lösung des gleichen Problems für sequentielle Programme mit Prozeduren.

Die Erweiterung verwendet als Operator für die Berechnung des Gewichtes von zwei parallel laufenden Pfaden den gleichen Operator wie für die Konkatenation zweier nacheinander ablaufender Pfade. Diese einfache Art der Behandlung des Interleavings zweier Pfade liegt in der Struktur des Halbringes begründet. Durch die Verwendung der Addition als Konkatenationsoperator ist die eigentliche Reihenfolge der Gewichte auf einem Pfad irrelevant, da die Addition kommutativ ist. Dies ist im Allgemeinen nicht der Fall und die Ausführungsreihenfolge der Regeln hat einen großen Einfluss auf des Resultat. In diesem Fall ergibt sich durch die Kommutativität jedoch die Tatsache, dass die beim Interleaving zweier Pfade auftretenden verschiedenen Reihenfolgen ihrer Gewichte zu immer dem gleichen Ergebnis führen. Stellvertretend brauchen wir also nur den Fall zu betrachten, in dem der eine Pfad komplett vor dem anderen durchlaufen wird. Die Addition erfüllt auch alle weiteren an den Interleavingoperator gestellten Eigenschaften.

Insbesondere erfüllt die Addition auch die Eigenschaft, dass das Gewicht des rechten Pfades eines Interleavings durch weitere Konkatenation veränderbar bleiben muss. Da im Fall des Dijkstra Halbringes für Konkatenation und Interleaving die Addition verwendet wird, folgt diese Eigenschaft direkt aus der Assoziativität der Addition. Es müssen also keine zusätzlichen Gewichte eingeführt werden, die mögliche Auswirkungen des Gewichtes des linken Astes eines Baumes auf gegebenfalls noch an den rechten Ast angehängte Teile mitführen. Als Konsequenz ist auch die Projektionsfunktion durch die Forderung, dass sie Werte aus dem Basiswertebereich auf sich selbst abbildet, als die Identität id auf $\mathbb{N} \cup \{0, \infty\}$

vollständig festgelegt. Sie erfüllt auch alle weiteren an die Projektionsfunktion gestellten Eigenschaften.

Eine weitere Beobachtung ist, dass durch die Verwendung des min Operators eine totale Ordnung auf dem Wertebereich induziert wird. Auch dies ist im Allgemeinen nicht der Fall. Die Kombination zweier Gewichte kann ein neues Gewicht sein, dass die Wirkung beider Pfade nur noch approximiert. In diesem Fall führt die totale Ordnung jedoch dazu, dass ein Gewicht immer genau die Wirkung eines Pfades beschreibt, nämlich die des kürzesten.

Betrachten wir als Beispiel das in Abbildung 2.9 in Abschnitt 2.3 vorgestellte System von Flussgraphen $\mathcal{G}$ und die dazu gehörende Umsetzung in ein DPN $\mathcal{M}$. Dabei gewichten wir jede Transitionsregel welche einen neuen Thread erzeugt mit der Länge 1 und alle anderen Regeln mit der Länge 0. Das Ergebnis sollte also die minimal nötige Anzahl von erzeugten Threads widerspiegeln, die zum Erreichen einer Konfiguration der Zielmenge, ausgehend von der abgefragten Konfiguration, erzeugt werden müssen. Als Zielmenge betrachten wir die Menge $(pN_{\mathcal{G}}^*\#)^* pe_{rec} N_{\mathcal{G}}^* (\#pN_{\mathcal{G}}^*)^*$, die die Menge der tatsächlich erreichbaren Konfigurationen an Programmpunkt e_{rec} umfasst, dargestellt durch den Automaten in Abbildung 2.10.

Abbildung 4.3 zeigt das Ergebnis der Anwendung des Verfahrens. Der berechnete annotierte Automat liefert für die Startkonfiguration pe_{main} die Länge 1 als Ergebnis. Dies ist offensichtlich korrekt, da Pfade existieren, die von e_{main} nach e_{rec} laufen, und auf jedem dieser Pfade die erste Kante genau einmal durchlaufen und damit der neue Thread erzeugt wird.

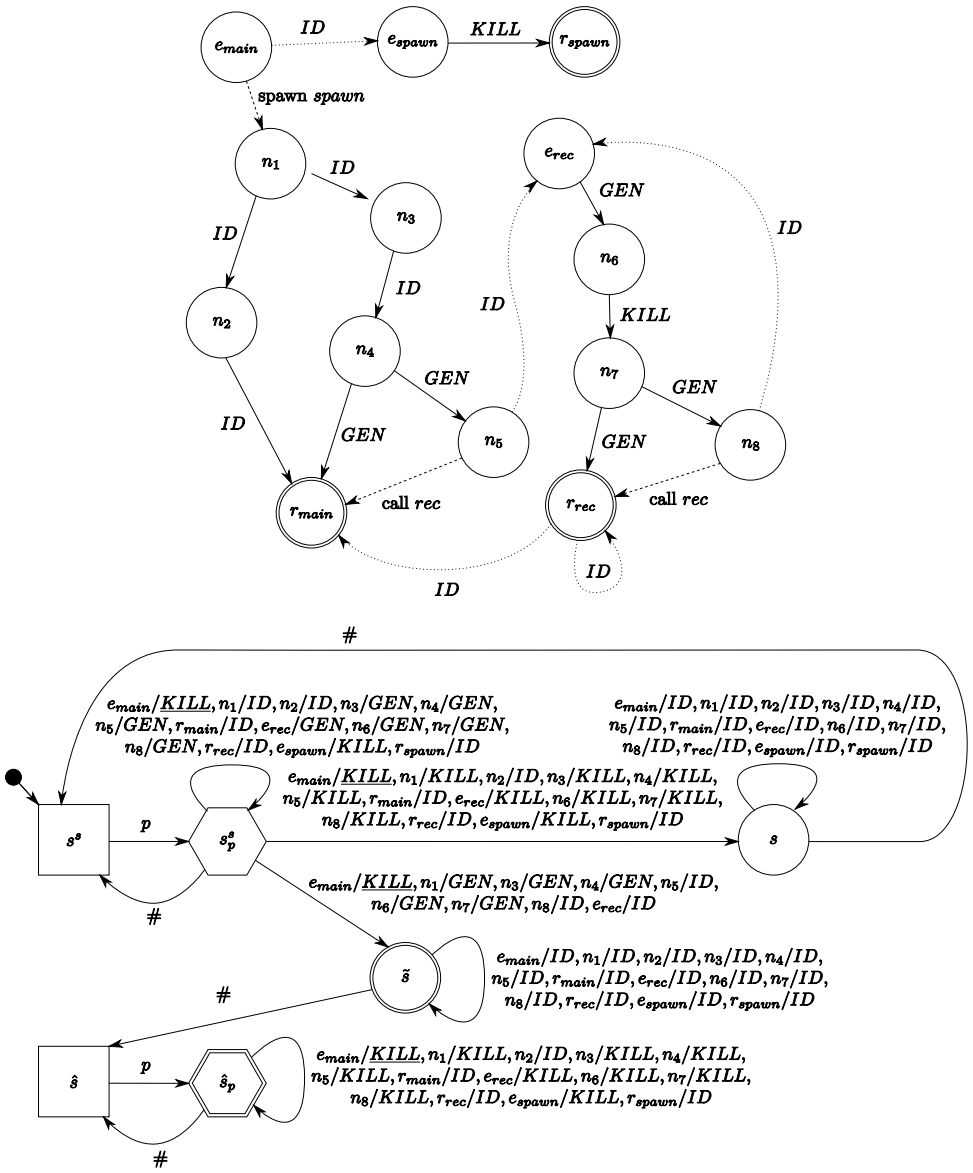


Abbildung 4.1: Annotierter $\mathcal{M}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, des in Abbildung 2.10 betrachteten $\mathcal{M}$-Automaten entsteht

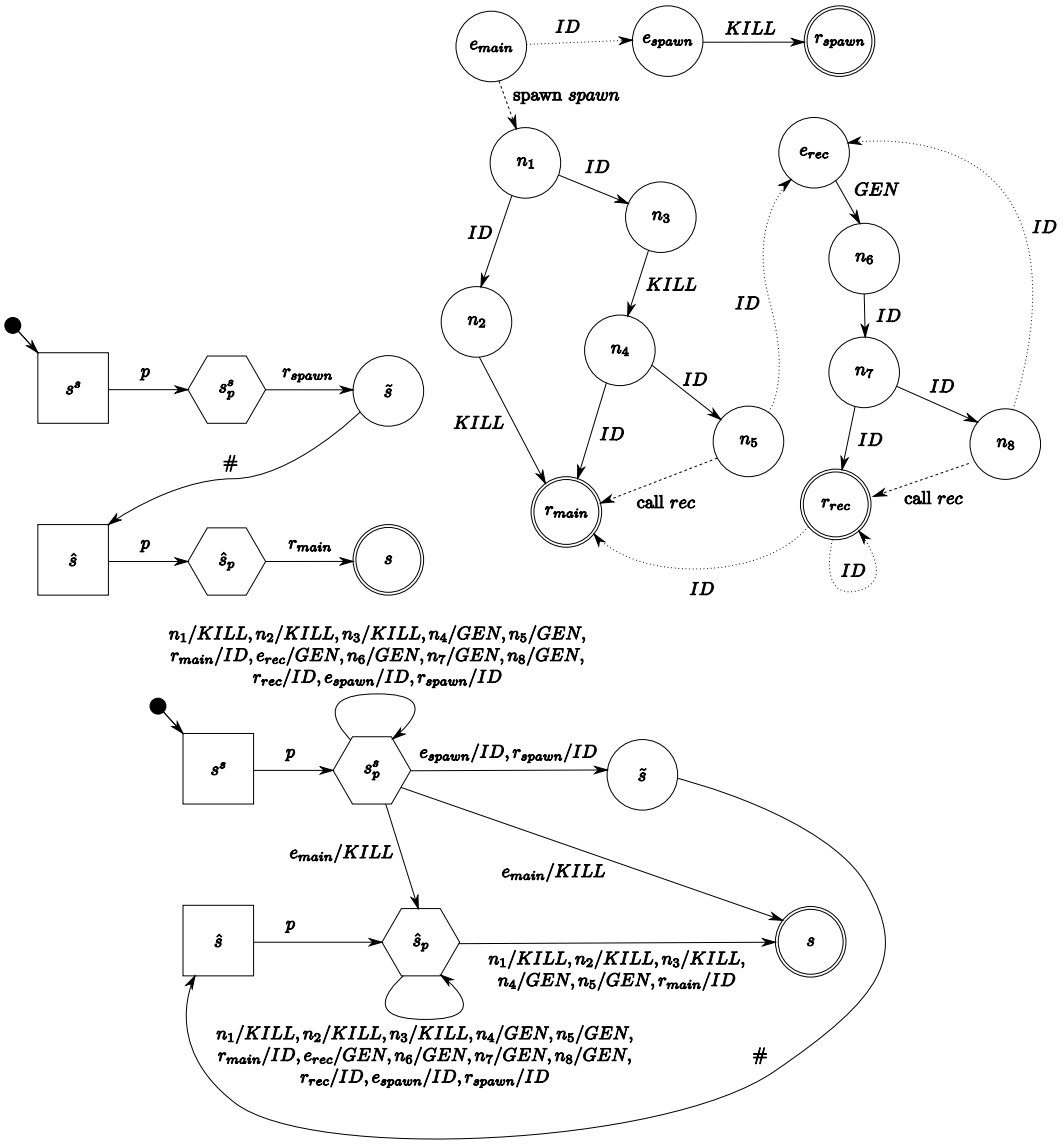


Abbildung 4.2: Annotierter $\mathcal{M}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, aus dem abgebildeten $\mathcal{M}$-Automaten entsteht

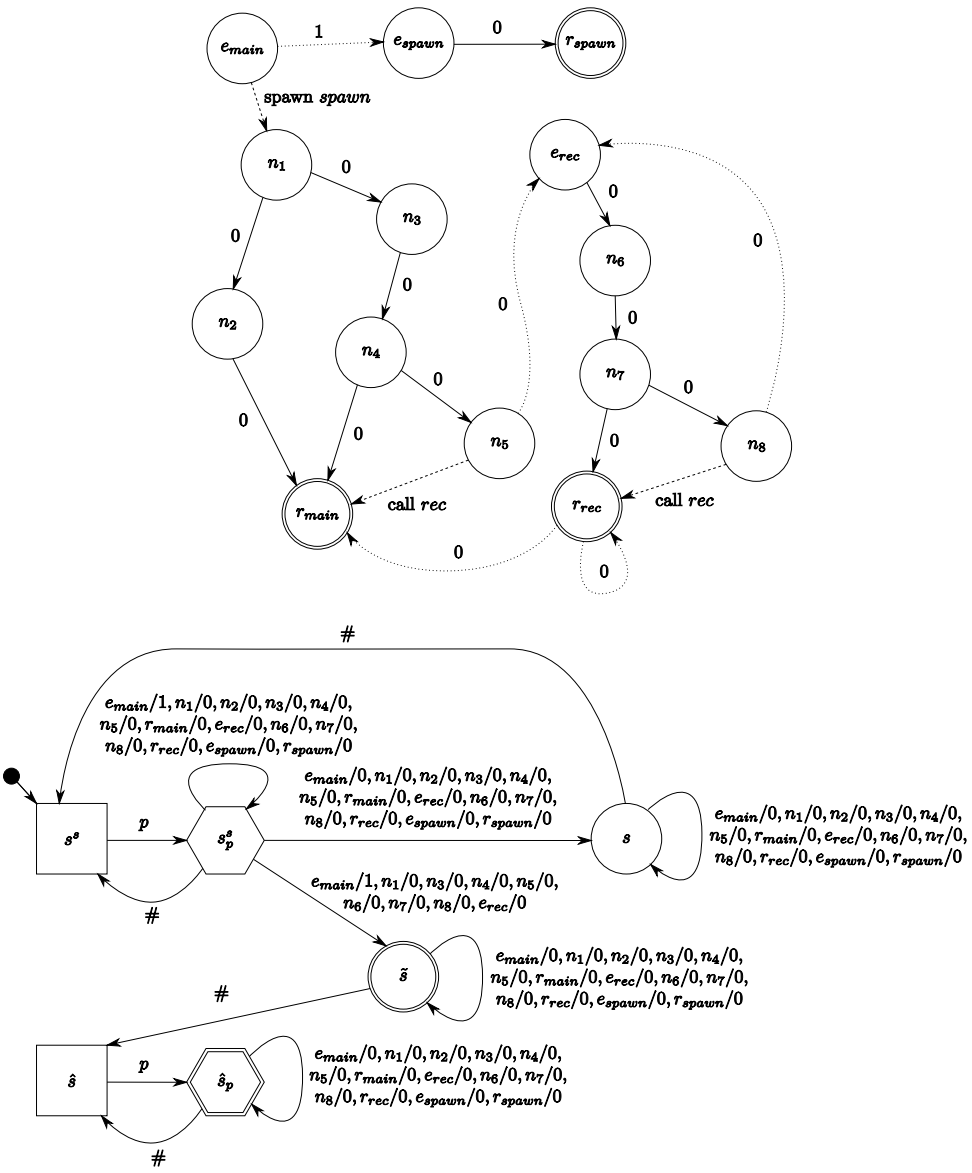


Abbildung 4.3: Annotierter $\mathcal{M}^*$-Automat, der durch Saturierung und Gewichtung, basierend auf den gegebenen Gewichten, des in Abbildung 2.10 betrachteten $\mathcal{M}$-Automaten $\mathcal{A}$ entsteht

KAPITEL 5

Zusammenfassung und Ausblick

Wir haben gezeigt, dass man den in [RSJM05] vorgestellten Ansatz der Verwendung von gewichteten Push-Down Systemen zur Datenflussanalyse auf die in [BMOT05] eingeführten dynamischen Push-Down Netzwerke erweitern kann. Die dadurch erhaltenen gewichteten dynamischen Push-Down Netzwerke erweitern die Analysemöglichkeiten von sequentiellen Programmen mit Prozeduren hin zu Programmen mit Prozeduren und dynamischer Threaderzeugung.

Nach einer notwendigen Abstraktion von Ausführungspfaden eines DPN zu Bäumen von Ausführungspfaden, in denen die Pfade der einzelnen Stacks eines DPN getrennt betrachtet werden, haben wir in Kapitel 3 die Menge der erreichenden Pfade von einer Konfiguration zu einer Menge von Zielkonfigurationen zunächst direkt durch ein Ungleichungssystem charakterisiert. Durch abstrakte Interpretation konnte von der Betrachtung der Pfadmengen zur Untersuchung einer allgemeinen Klasse von Gewichte übergegangen werden. Dabei haben wir gezeigt, dass das dazu erstellte Ungleichungssystem eine optimale und korrekte Lösung liefert, falls die Gewichte die geforderten Eigenschaften erfüllen.

In Kapitel 4 haben wir gezeigt, dass das vorgestellte Verfahren für das Berechnen von Informationen über kürzeste Ausführungspfade, sowie für das Berechnen von Lösungen von vorwärts Bitvektor-Problemen genutzt werden kann. Dabei ist das vorgestellte Verfahren eine Erweiterung des in [BMOT05] vorgestellten Verfahrens zur Lösung von Bitvektor-Problemen durch Verwendung Automaten-basierter Techniken für dynamischer Push-Down Netzwerke. Zum einen ist auch die Abbildung von gewissen Probleme, die keine Bitvektor-Problemen sind, auf unser Modell möglich, zum anderen erlaubt es durch Nutzung von Gewichten in Form von Vektoren von Bitvektor-Gewichten eine simultane Behandlung mehrerer Bitvektor-Probleme. Im Vergleich zu klassischen Verfahren wie [LMO07] besteht

aber immer noch der Nachteil, dass Anfragen für jeden Programmpunkt einzeln gestellt werden müssen. Im Gegenzug ermöglicht unser Modell jedoch diese Anfragen allgemeiner zu formulieren und neben dem aktuellen Programmpunkt auch Bedingungen an den Zustand des kompletten Stacks und aller übrigen Stacks einer Konfiguration zu stellen. Dies ermöglicht eine gezieltere Abfrage von Informationen zu gewissen Konfigurationen im Gegensatz zu vorherigen Methoden, bei denen die Informationen an den Programmpunkten über alle möglichen Konfigurationen, an denen dieser Programmpunkt aktiv, ist kombiniert werden.

Der in [LTKR08] betrachtete Ansatz einzelne Prozesse als WPDS zu modellieren und dazu explizite Kontextwechsel zu untersuchen unterliegt der Beschränkung, nur für endlich viele Kontextwechsel berechenbar zu sein. Mit unserem Modell erreichen wir eine Analyse mit beliebigen Kontextwechsel, verlieren auf der anderen Seite jedoch die Möglichkeit Synchronisation abbilden zu können.

Mögliche Erweiterungen unseres Verfahrens, die einer weiteren Untersuchung bedürfen, umfassen unter anderem die Generierung von Zeugenmengen. Dabei wird eine möglichst kleine Menge von Pfaden zu einer Lösung geliefert, deren Betrachtung ausreicht, um das Ergebnis zu rechtfertigen. Im in [RSJM05] präsentierten Algorithmus ist es durch eine einfache Modifikation möglich diese Informationen während der Berechnung mitzuführen. Auch in unserem Algorithmus wäre dies möglich, würde aber zur Generierung einer Menge von Hecken führen, da wir vom konkreten Interleaving der Anweisungen mehrerer Threads abstrahieren. Da aber in der Regel erst eine konkrete Ausführungsreihenfolge von Anweisungen des gesamten Systems eine schlüssige Rechtfertigung einer Lösung liefert, bleibt zu untersuchen, ob man diese aus den gegebenen Hecken konstruieren kann.

[LR06] untersucht effizientere Verfahren zur Lösung des Problems der abstrakten Grammatik, indem zyklische Abhängigkeiten verschiedener Regeln zuerst betrachtet werden. Die Beobachtungen könnten auf zyklische Abhängigkeiten von Ungleichungen übertragen werden und zu effizienten Verfahren zur Lösung der in dieser Arbeit betrachteten Ungleichungsysteme führen.

In [LRB05] wird das Modell der WPDS erweitert, um lokale Variablen in Prozeduren zu behandeln. Diese Erweiterung könnte man auf Umsetzbarkeit in unser Modell überprüfen. Desweiteren könnten man in diesem Kontext versuchen, das verwendete Verfahren auch zur Modellierung von lokalen Variablen in verschiedenen Threads zu verwenden.

Um in einem ersten Schritt die Berechnung von Rückwärtsanalysen zu ermöglichen, wäre es denkbar ein Verfahren zu entwickeln, das das Gewicht einer regulären Menge von Konfigurationen bestimmt. Hierzu könnte der berechnete gewichtete Automat, der die Gewichte aller Konfigurationen enthält, durch Schnittbildung auf die gewünschte Menge reduziert werden, ohne dabei die Gewichte zu verlieren. Durch einsammeln der Gewichte der eingehenden Kanten eines Zustandes im Schnittautomaten, könnte dann an den akzeptierenden Zuständen ein Gesamtgewicht abgelesen werden. Dadurch ließen sich zumindest Rückwärtsanalysen berechnen, in denen die betrachtete Menge von Konfigurationen an einem Programmpunkt regulär ist.

Desweiteren wäre die Möglichkeit zu untersuchen, das Verfahren iterativ anzuwenden. In Anlehnung an das in [BMOT05] beschriebene Verfahren zum Lösen von Bitvektor-Problemen, durch die Untersuchung einer iterativen Anwendung des PRE^*-Algorithmus unter Verwendung verschiedener Regelsysteme, wäre dies auch für unser Modell in Betracht zu ziehen. Hierbei könnte nach jeder Anwendung des Verfahrens der Ergebnisautomat, mit oben schon erwähnten Mitteln, auf eine gewünschte Menge von Konfigurationen reduziert werden. Im nächsten Schritt würde das Verfahren auf dem gewonnenen gewichteten Schnittautomaten, gegebenenfalls sogar mit veränderter Gewichtung der Transitionen oder anderem DPN-Regelsystem, angewendet. Dabei würden jedoch die in vorherigen Schritten berechneten Gewichte, die im Schnittautomaten vorhanden sind, als initiale Gewichtung in die Berechnung der neuen Gewichte mit einbezogen. Das Ergebnis einer Anfrage wäre das Gewicht der Pfade die in einer Konfiguration starten und nacheinander zu jeweils einer Konfiguration in den gewünschten Mengen führen.

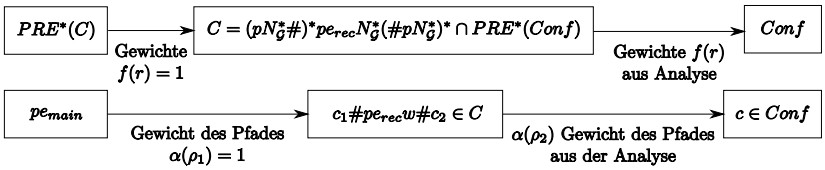


Abbildung 5.1: Beispiel für Pfade die nacheinander Konfigurationsmengen durchlaufen und Gewichtung die die Berechnung einer Rückwärtsanalyse an einem Programmpunkt ermöglichen

Hierdurch können noch spezifischere Anfragen gestellt werden. Zum einen wäre eine Berechnung von Rückwärtsanalysen möglich, da nun die Menge der Pfade, die in der Startkonfiguration beginnen, dann eine Konfiguration am gewünschten Programmpunkt und schließlich eine beliebige Endkonfiguration erreichen, genau beschrieben werden kann. Da man sich nur für die Gewichte der zweiten Abschnitte der Pfade interessiert, diese Ergeben sich aus der Rückwärtsanalyse, kann der Einfluss der ersten Abschnitte durch eine neutrale Gewichtung unterdrückt werden, wie in Abbildung 5.1 dargestellt ist. Das Gesamtgewicht der zur Startkonfiguration betrachteten Pfade entspricht dann dem Gesamtgewicht aller für die Rückwärtsanalyse relevanten Pfade vom Programmpunkt zu einer beliebigen Nachfolgerkonfiguration. Zum anderen ist auch die Abbildung von einfachen Synchronisationsoperationen zwischen Threads möglich, indem die Zwischenmengen jeweils die Zustände beschreiben in denen sich zwei Threads synchronisieren.

Im Zuge dieser Betrachtungen wären die in [LTKR08] untersuchten Techniken für WPDS zur Konstruktion eines so genannten gewichteten *Transducer* interessant. Ein Transducer ist ein endlicher Automat, der ein Wort von einer Sprache in eine andere Sprache übersetzt. Diese Konstruktion wird dazu benutzt um alle möglichen Transformationen von Konfigurationen durch Pfade in einem PDS zu beschreiben. So kann eine Konfiguration durch den Transducer in eine andere Konfiguration übersetzt werden, wenn ein Pfad im PDS existiert, der diese Transformation durchführen kann. Diese Übersetzung kann mit Gewichten versehen werden und man erhält einen Automaten, der zu jedem Paar von Konfigurationen das Gewicht

aller Pfade liefert, die diese verbinden. Mit einfachen Techniken lässt sich dann durch Vorgabe einer Menge von Konfigurationen als Ziel aus dem Transducer ein Automat wie im ursprünglichen Verfahren konstruieren. Hier wird also das Verfahren bis auf den letzten Schritt von der Angabe einer Menge von Zielkonfigurationen unabhängig gemacht. Eine direkte Anwendung auf WDPN ist hierbei nicht zu erwarten, da man aus dem Transducer auch die Menge der erreichbaren Konfigurationen konstruieren kann. Da diese für DPN im Allgemeinen nicht regulär ist, wird in der Regel kein Transducer, der die Übergänge eines DPN beschreibt, existieren. Es bleibt jedoch zu untersuchen, in welchen Fällen dies doch möglich ist, oder, ob allgemeinere Konstruktionen existieren, die eine Vorberechnung der Ergebnisse, unabhängig von der Menge der Zielkonfigurationen, für WDPN erlauben.

Der offensichtlichste Ansatzpunkt ist jedoch, neben den von uns untersuchten Gewichten zur Bestimmung der Länge kürzester Ausführungspfade und zur Lösung von Bitvektor-Problemen, für weitere Klassen von Datenflussanalysen zu untersuchen, ob diese sich durch unser Modell berechnen lassen. Interessant wäre zum Beispiel die Analyse von Abhängigkeiten. Aber auch eine Konstruktion der Mengen von Hecken selber wäre interessant. Dazu müsste jedoch die verwendete Baumstruktur etwas abgewandelt werden um eine Regularität der Mengen und damit eine endliche Darstellungsmöglichkeit zu erreichen. Im Moment werden Prozeduraufrufe und damit auch mehrfache rekursive Aufrufe sequentiell durch einen Ast des Baumes dargestellt. Die auf möglichen Ästen auftretenden Regelfolgen sind damit im Allgemeinen nicht regulär. Dies könnte jedoch zum Beispiel durch Modellierung von Prozeduraufrufe durch zusätzliche Verzweigungen erreicht werden, die dann zu Regularität der Mengen im Sinne eines Baumautomaten führen.

Literaturverzeichnis

[BEM97] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.

[BMOT05] Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *CONCUR 2005 - Concurrency Theory*, pages 473–487, London, UK, 2005. Springer-Verlag.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[EK99] Javier Esparza and Jens Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Foundations of Software Science and Computation Structure*, pages 14–30, 1999.

[EP00] Javier Esparza and Andreas Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Symposium on Principles of Programming Languages*, pages 1–11, 2000.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[LMO07] Peter Lammich and Markus Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation and procedures. In *CONCUR*, pages 287–302, 2007.

[LR06] Akash Lal and Thomas W. Reps. Improving pushdown system model checking. In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2006.

[LRB05] Akash Lal, Thomas W. Reps, and Gogul Balakrishnan. Extended weighted pushdown systems. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2005.

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2008.

[MO04] Markus Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311(1-3):325–388, 2004.

[MO06] Markus Müller-Olm. *Variations on Constants: Flow Analysis of Sequential and Parallel Programs (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[RSJM05] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[SS00] Helmut Seidl and Bernhard Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic J. of Computing*, 7(4):375–400, 2000.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Steinfurt, den 24. November 2009