

RWTH Aachen

Dekanat der Fakultät für Mathematik, Informatik und
Naturwissenschaften

Lehrstuhl für Softwaremodellierung und Verifikation,
Informatik 2

Konstruktion & Inferenz von
Heapabstraktionsgrammatiken

Diplomarbeit

Aachen, den 4. Januar 2010

Vorgelegt von: Christina Maria Jansen
geb. am: 05.01.1984
Studiengang: Informatik (Diplom)

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, 4. Januar 2010

Unterschrift

Kurzdarstellung

In der heutigen Zeit, in der der Bedarf an immer komplexer werdender Software in immer kürzeren Abständen ständig steigt und eingebettete Systeme in immer mehr Lebensbereichen Einzug halten, gewinnt die Softwareverifikation zunehmend an Bedeutung. Gleichzeitig finden heapbasierte, dynamische Datenstrukturen in den modernen Programmierkonzepten mehr und mehr Verwendung und sind aus diesen längst nicht mehr wegzudenken. Bei der Verifizierung stellen unendliche Zustandsräume, wie sie bei der Auswertung von Programmen auftreten, die mithilfe von heapbasierten, dynamischen Datenstrukturen arbeiten, ein Problem dar. Durch Abstraktionstechniken lassen sich jedoch endliche Repräsentation herstellen, auf denen dann Techniken wie Model Checking zur Verifikation angewandt werden können. Als Grundlage zur Abstraktion können Hyperkantenersetzungsgrammatiken (engl. hyperedge replacement grammar, HRG) dienen, die – um Korrektheit des Vorgehens gewährleisten zu können – einige spezielle Eigenschaften besitzen müssen. Die Form einer geeigneten Grammatik hängt dabei stark von der zugrunde liegenden Datenstruktur und dem Zeiger-Programm ab. Um den Nutzer nicht vor das Problem des Grammatikentwurfs stellen zu müssen, ist das automatische Generieren von geeigneten HRGs zu gegebenen Zeiger-Programmen sowie die Herstellung der geforderten Eigenschaften von grossem Interesse. In dieser Diplomarbeit werden die Anforderungen an HRGs zur Heapabstraktion im Detail diskutiert, um im Anschluss einen Algorithmus zur Konstruktion einer diese Eigenschaften erfüllenden, äquivalenten Grammatik angeben zu können. Mit diesen Voraussetzungen wird dann ein Lernalgorithmus erarbeitet, der als Resultat eine für Abstraktionszwecke geeignete Hyperkantenersetzungsgrammatik liefert.

Danksagung

Bei Herrn Dr. T. Noll bedanke ich mich für die Vergabe und Betreuung dieser Diplomarbeit. Ganz besonders möchte ich mich bei Jonathan Heinen bedanken, der mich durch zahlreiche Ideen und Denkanstöße immer wieder bei der Erarbeitung des Themas und durch zahlreiche Verbesserungsvorschläge bei der Erstellung der Arbeit unterstützt hat.

Des weiteren danke ich Stefan, Stefan und Robert für ihre Unterstützung und viele Stunden des Korrekturlesens.

Nicht zuletzt möchte ich meinen Eltern danken, die mir durch ihre fortwährende – nicht nur finanzielle – Unterstützung das Studium und diese Arbeit erst ermöglichten und ihr Fortschreiten mit Anteilnahme verfolgten.

Inhaltsverzeichnis

Eidesstattliche Erklärung	i
Kurzdarstellung	iii
Danksagung	v
1 Einleitung	3
2 Grundlagen	7
2.1 Hyperkanten und Hypergraphen	7
2.2 Hyperkantenersetzungsgrammatik	10
2.3 Hypergraphen und HRGs: grundlegende Eigenschaften	13
2.3.1 Inhärente Eigenschaften	14
2.3.2 Spezielle Eigenschaften	15
2.4 Graphalgorithmen für HRGs	18
3 Heapabstraktion	21
3.1 Heaprepräsentation	21
3.2 Abstraktion & Konkretisierung	24
3.3 Erweiterungen	32
4 Lokale Apex-Eigenschaft für Heapabstraktionsgrammatiken	35
4.1 Konstruktion der lokalen Apex-Eigenschaft	35
4.1.1 Konstruktion der lokalen Greibach Normalform	45
4.2 Lokale Greibach Normalform am Beispiel	59
5 Inferenz von HRGs	67
5.1 Grammatikalische Inferenz	67
5.2 Lernen von HRGs	69
5.2.1 Determinismus	72
5.2.2 Korrektheit	81
5.3 Lernen von HRGs am Beispiel	82

6 Zusammenfassung & Ausblick	87
Literaturverzeichnis	i

Kapitel 1

Einleitung

In Zeiten, wo immer komplexere Software in immer kürzeren Produktionszeiten entwickelt wird und eingebettete Systeme in immer mehr Lebensbereichen Einzug halten, in denen Fehlfunktionen ernstzunehmende Konsequenzen nach sich ziehen, gewinnt die Softwareverifikation zunehmend an Bedeutung. Gleichzeitig sind heapbasierte Datenstrukturen in den heutigen Programmierkonzepten allzeit präsent und aus diesen längst nicht mehr wegzudenken. Dadurch, dass zur Laufzeit Objekte erstellt oder entfernt werden können, und damit die Menge der möglichen Heapzustände im Allgemeinen unendlich und zur Kompilierzeit nicht bestimmbar ist, stellen sie allerdings für die meistgenutzten Verifikationstechniken ein Problem dar. Man ist daher bemüht potentiell unendliche Strukturen durch endliche Abstraktionen zu repräsentieren ohne relevante Informationen zu verlieren.

Eine Möglichkeit zur Abstraktion von Heapzuständen bietet das Konzept der Hyperkantenersetzungsgrammatiken [Hab92]. Dabei wird ein Heapzustand durch einen Hypergraphen repräsentiert, die Produktionsregeln der Grammatik werden zum Abstrahieren und Konkretisieren der Hypergraphen angewandt. Die grundlegende Idee besteht darin, die Rückwärtsanwendung von Produktionsregeln zuzulassen, diesen Vorgang bezeichnen wir als Abstraktion. Falls notwendig können abstrahierte Teilgraphen durch normale Regelanwendung wieder konkretisiert werden. Dies hat zur Folge, dass keine Semantik für die abstrakten Teile der Graphen gebraucht wird, da Operationen nur auf konkreten Teilgraphen ausgeführt werden müssen [RN08].

Zur korrekten Abstraktion von Heapzuständen muss eine HRG festgelegte notwendige Eigenschaften erfüllen, damit z.B. gewährleistet wird, dass keine relevanten Informationen, wie die Anzahl und der Inhalt von Programmvariablen, verloren gehen. Die rückwärtige Nutzung von Produktionsregeln führt zusätzlich zu weiteren Anforderungen an die Grammatik, damit die Terminierung des Abstraktionsvorgangs oder die Existenz entsprechender Konkretisierungsregeln garantiert werden können. Für die Abstraktion geeignete HRGs nennen wir Heapabstraktionsgram-

matiken. Es existiert allerdings keine universelle Heapabstraktionsgrammatik, vielmehr hängt diese stark von dem betrachteten Zeiger-Programm ab und muss individuell für dieses erstellt werden. Dabei hat man verschiedene Möglichkeiten eine HRG zu bestimmen: manuell durch genaue Betrachtung der Datenstruktur und der zu verifizierenden Anwendung, automatisch durch Inferenz einer Grammatik aus einer Beispielmenge von Graphen und Lernen von Grammatikregeln während der Programmausführung oder eine Kombination der genannten Ansätze. Bei jeder Grammatikgenerierung muss im Nachhinein sichergestellt sein, dass alle für ein korrektes Abstraktionsverfahren benötigten Anforderungen in der erstellten HRG gegeben sind. Um dem Anwender die häufig schwierige Grammatikgenerierung abzunehmen, wird in dieser Diplomarbeit ein Algorithmus zum automatischen Lernen von HRGs vorgestellt. Als Grundlage dient der Algorithmus zum Inferieren von Hyperkantenersetzungsgrammatiken aus [JK90]. Das Lernen von Grammatiken wird dabei ausgehend von einer Positivmenge von Eingabegraphen durch iterative Dekomposition von Graphen realisiert.

Der Algorithmus arbeitet mit einer Beispielmenge von ungerichteten und unbeschrifteten Eingabegraphen und weist zudem hochgradig nichtdeterministisches Verhalten auf, weshalb er in der ursprünglichen Form für hier betrachtete Zwecke nicht einsatzfähig ist. Daher wird anlehnend ein deterministischer Lernalgorithmus entwickelt, der sowohl mit Hypergraphen als Eingabe umgehen kann als auch das Lernen von weiteren Produktionsregeln während der Programmausführung unterstützt. Initiiert wird dieser durch das Auftreten eines Heapzustandes, der durch vorhandene Produktionsregeln nicht ausreichend abstrahiert werden kann. Für den Lernprozess werden fünf Basisoperationen benötigt, die als Resultat eine erweiterte Grammatik liefern, anhand derer sich der zuvor problematische Heapzustand nun abstrahieren lässt. Der Algorithmus soll zusätzlich gewährleisten, dass die für eine Heapabstraktionsgrammatik geforderten Eigenschaften gegeben sind. Im Verlauf der Arbeit hat sich die Idee, während der Grammatikkonstruktion ausschließlich Produktionsregeln zu erlernen, die geforderte Eigenschaften nicht verletzen, als problematisch herausgestellt und es wurde eine alternative Lösung gesucht.

Getrennt vom Lernvorgang wurde ein Algorithmus entworfen, der zu einer gegebenen Grammatik eine äquivalente Grammatik konstruiert, die alle Anforderungen zur Heapabstraktion erfüllt. Er beruht auf einer Umformung der Eingabegrammatik in eine Normalform, die es durch einfache Kantenersetzung erlaubt, die geforderten Eigenschaften herzustellen. In [EHL94] ist bereits eine ähnliche Konstruktion für die Greibach Normalform betrachtet worden. Da sie aber ausschließlich für HRG-Sprachen mit beschränktem Grad vorgesehen ist und damit nicht mit Datenstrukturen wie z.B. gewurzelten Bäumen arbeiten kann, wurde basierend auf der Greibach Normalform eine neue Normalform entworfen, die auch unbeschränkten Eingangsgrad zulässt. Die Umformung einer Eingabe-HRG in diese neue Normalform erfolgt analog zum Algorithmus der Greibach Normalform. Knoten mit unbeschränktem Eingangsgrad müssen jedoch getrennt betrachtet und aufgesplittet werden, um weiterhin Korrektheit des Verfahrens gewährleisten zu können. Diese vom eigentlichen Lernalgorithmus abgekoppelte Vorgehensweise hat den Vorteil, dass sie es

ermöglicht auch eine Startgrammatik – sollte der Nutzer eine vorgeben wollen – in eine geeignete Heapabstraktionsgrammatik umformen zu können.

Schlussendlich erhalten wir durch die Kombination beider Algorithmen – des Lernalgorithmus sowie des Algorithmus zur Herstellung der Normalform – eine zulässige Heapabstraktionsgrammatik, die zur Laufzeit des zu verifizierenden Programmes automatisch erlernt werden kann. Dabei steht es dem Nutzer frei eine Startgrammatik vorzuschlagen, oder diese nach Bedarf generieren zu lassen.

Aufbau der Arbeit In Kapitel 2 werden wichtige Grundlagen zu Hypergraphen und Hyperkantenersetzungsgrammatiken eingeführt. Außerdem wird in einem Unterabschnitt auf die Übertragbarkeit zweier herkömmlicher Graphalgorithmen auf HRGs betrachtet, da diese Algorithmen später im Lernprozess benötigt werden. Daran anschließend beschäftigt sich Kapitel 3 mit der Heaprepräsentation durch Hypergraphen, sowie mit ihrer Abstraktion und Konkretisierung durch HRGs. Der zweite Teil des Kapitels beschäftigt sich mit den Anforderungen an Heapabstraktionsgrammatiken, Eigenschaften wie Kontextfreiheit, Variablenfreiheit und lokal apex werden vorgestellt. Das darauffolgende Kapitel 4 widmet sich der lokalen Apex-Eigenschaft. Sie ist die komplexeste Eigenschaft, die an Heapabstraktionsgrammatiken gestellt wird. Ihre Gültigkeit wird im Lernprozess zu einer gegebenen HRG durch die Konstruktion einer äquivalenten Grammatik mit lokaler Apex-Eigenschaft sichergestellt. Diese Konstruktion wird im Verlauf des Kapitels erarbeitet. Unter Zuhilfenahme der Apex-Konstruktion wird in Kapitel 5 ein Lernalgorithmus für HRGs entwickelt, der anhand von fünf Basisoperationen eine bestehende Grammatik um weitere Produktionsregeln erweitert, sobald eine Abstraktion ein im Programmverlauf des zu verifizierenden Zeiger-Programmes auftretender Heapzustand durch bestehende Regeln nicht ausreichend möglich ist. Abschließend werden in Kapitel 6 die Ergebnisse der Arbeit zusammengefasst und für den Lernalgorithmus interessante Fragestellungen aufgeführt, deren Betrachtung für weitere Arbeit auf dem Gebiet der Heapabstraktion von Interesse ist.

Kapitel 2

Grundlagen

Hyperkantenersetzungsgrammatiken lassen sich bei der Abstraktion von auf heap-basierten, dynamischen Datenstrukturen basierenden Zeigerprogrammen als intuitive, graphische Modelliersprache einsetzen und bieten in den meisten Fällen ausreichende Ausdrucksstärke [HNR09]. In diesem Kapitel werden Grundlagen von Hypergraphen und Graphersetzungsgrammatiken sowie wichtige Begriffe eingeführt, die zum besseren Verständnis der nachfolgenden Kapitel beitragen. Die Definitionen halten sich an die Doktorarbeit [Hab92], diese haben sich als gute Grundlage im Verlauf der Arbeit herausgestellt.

2.1 Hyperkanten und Hypergraphen

Hypergraphen stellen eine Generalisierung der im Allgemeinen üblichen Graphen dar, indem sie normale Kanten durch Hyperkanten ersetzen. Eine Hyperkante erweitert das Konzept der Kanten darum, dass sie eine beliebige Anzahl von anliegenden Knoten besitzen kann, anstatt auf zwei Knoten beschränkt zu sein. Eine übliche, gerichtete Kante lässt sich dabei durch eine Hyperkante mit genau zwei anliegenden Knoten darstellen.

Definition 2.1.1 (Hyperkante) *Eine Hyperkante v ist ein beschriftetes Objekt mit einer geordneten Menge von anliegenden Tentakeln.*

Bevor wir zur der Definition von Hypergraphen kommen, betrachten wir noch einige grundlegende Annahmen und Notationen. Sei Σ ein Alphabet. Wir nehmen im Folgenden an, dass sich Σ aus den disjunkten Mengen T und N zusammensetzt, d.h. dass $\Sigma = N \cup T$ und $N \cap T = \emptyset$ gilt.

Wir bezeichnen die Menge N als Nichtterminalmenge und nutzen Grossbuchstaben, um ihre Elemente zu bezeichnen. T hingegen nennen wir Terminalmenge, ihre Elemente werden durch Kleinbuchstaben gekennzeichnet. Weiterhin definieren wir zu einer Menge V die Menge aller Zeichenketten über V mit V^* . Für ein Element $\omega \in V^*$ schreiben wir $|\omega|$ für die Länge von ω und $\omega(i)$ für das i -te Zeichen von ω . Die Identitätsfunktion auf einer Menge V wird mit id_V bezeichnet. Abschließend definieren wir die Restriktion $f|_{E'}$ einer Funktion $f : E \rightarrow V$ als $f|_{E'} : E' \rightarrow V$, $f(e) = f(e)$, $\forall e \in E'$, falls $E' \subseteq E$.

Definition 2.1.2 (Hypergraph über Σ) Ein Hypergraph H über dem Alphabet Σ ist ein 5-Tupel (V, E, lab, att, ext) mit den folgenden Komponenten:

- V - Knotenmenge
- E - Menge von Hyperkanten
- $lab : E \rightarrow \Sigma$ - Beschriftungsfunktion
- $att : E \rightarrow V^*$ - Bindungsfunktion
- $ext \in V^*$ - externe Knoten

$V_H, E_H, lab_H, att_H, ext_H$ beschreibt die einzelnen Komponenten von H . Die Menge aller Hypergraphen über Σ wird mit H_Σ bezeichnet.

Bemerkung 2.1.1 Eine Kante mit Nichtterminalbezeichner nennen wir Nichtterminalkante. Diese Konvention gilt analog für Terminalsymbole. Ein Knoten wird als Nichtterminalknoten bezeichnet, wenn an ihm mindestens eine Nichtterminalkante anliegt.

Die Menge der an einem Knoten v anliegenden Kanten wird mit $E(v)$ bezeichnet. Den an einer Kante e durch Tentakel i anliegenden Knoten v erhält man mit $v = nod(e, i) = att(e)(i)$.

Notation 2.1.1 Zur graphischen Darstellung eines Hypergraphen orientiert sich diese Arbeit an einem weitverbreiteten Standard. Knoten werden dabei als Kreise dargestellt, während Hyperkanten durch Rechtecke und von diesen ausgehenden Tentakeln als ungerichtete Kanten abgebildet werden. Sowohl die Tentakeln einer Hyperkante als auch die externen Knoten werden aufsteigend, beginnend bei 1, mit natürlichen Zahlen $i \in \mathbb{N}$ beschriftet. Terminalkanten bilden bei dieser Notation eine Ausnahme, sie werden als herkömmliche, gerichtete Kante dargestellt. Der Quellknoten ist dabei der an Tentakel 1 anliegenden Knoten.

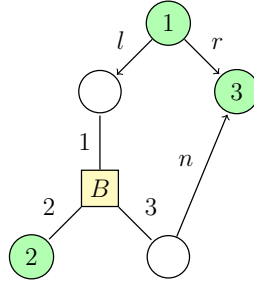


Abbildung 2.1: Graphische Darstellung eines Hypergraphen

Beispiel 2.1.1 Ein Beispiel für eine graphische Darstellung eines Hypergraphen ist in Abbildung 2.1 zu finden.

Die grün hinterlegten Knoten, die mit $i \in [1, 3]$ beschriftet sind, stellen die externen Knoten dar. Die gerichteten Kanten mit Beschriftungen l , r und n sind Terminalkanten. Das Rechteck oberhalb des zweiten externen Knotens entspricht einer Nichtterminalkante. Sie besitzt drei Tentakel, die wiederum eine Zahl $i \in [1, 3]$ tragen.

Führen wir an dieser Stelle noch einige im weiteren Verlauf nützliche Definitionen bezüglich Hypergraphen ein.

Definition 2.1.3 (Rang eines Hypergraphen/einer Hyperkante) Der Rang eines Hypergraphen $rk(H)$ entspricht der Anzahl seiner externen Knoten, d.h. $rk(H) = |ext(H)|$. Der Rang einer Hyperkante $rk(e)$ entspricht der Anzahl der ihr zugeordneten Knoten, d.h. $rk(e) = |att(e)|$.

Die Menge aller externen Knoten eines Hypergraphen H notieren wir als $[ext_H] = \{v \in V \mid v = ext(i) \text{ für ein } 1 \leq i \leq rk(H)\}$.

Definition 2.1.4 (Henkel) Sei $A \in \Sigma$. Wir nennen einen Hypergraphen $H = (V, E, lab, att, ext)$ ein A -Henkel A^\bullet , falls $|V| = |ext|$ und $E = \{e\}$ mit $\forall v \in V : v = ext(i) \wedge v = nod(e, i)$ gilt, für ein $i \in [1, |ext|]$. Die Bezeichnung $A(m)^\bullet$ wird analog verwendet, wenn $rk(e) = m$.

2.2 Hyperkantenersetzungsgrammatik

In diesem Abschnitt wird das Konzept der Hypergraphen um Grammatiken erweitert, die die gezielte Ersetzung von Hyperkanten durch komplette Hypergraphen erlauben. Diese Grammatiken werden Hyperkantenersetzungsgrammatiken genannt. Sie arbeiten im Grunde analog zu den bekannten Stringgrammatiken. Jedoch wird bei Anwendung einer Produktionsregel kein Zeichen durch eine Zeichenkette ersetzt, sondern eine Hyperkante durch einen Hypergraphen. Die linken Seiten der Produktionsregeln sind daher mit Nichtterminalen beschriftet. Bei der Ersetzung von Hyperkanten muss man jedoch auf eine Feinheit achten: Hyperkante und Hypergraph müssen den gleichen Rang haben, d.h. die Anzahl der durch die Entfernung der Hyperkante „freigegebenen“ Knoten entspricht der Anzahl der Knoten des Hypergraphen, die „angedockt“ werden sollen. Bevor wir zur Definition einer Hyperkantenersetzungsgrammatik kommen, formalisieren wir vorerst das hier intuitiv beschriebene Konzept der Kantenersetzung.

Definition 2.2.1 (Hyperkantenersetzung) *Gegeben ein Hypergraph H . Die Ersetzung der Hyperkante $e \in E_H$ durch einen Hypergraphen K induziert eine zulässige Hyperkantenersetzung $H[K/e]$, falls $rk(e) = rk(K)$ gilt. Es entsteht ein Hypergraph $H[K/e] = (V', E', lab', att', ext')$ wie folgt:*

- $V' = V_H \cup (V_K \setminus [ext_K])$
- $E' = (E_H \setminus \{e\}) \cup E_K$
- $lab' = (lab_H \upharpoonright (E_H \setminus \{e\})) \cup lab_K$
- $att' = repl \circ (att_H \upharpoonright (E_H \setminus \{e\})) \cup att_K$ mit
 $repl = id_{V_{H[K/e]}}[ext_K(1)/att_H(e)(1), \dots, ext_K(rk(e))/att_H(e)(rk(e))]$
- $ext' = ext_H$

Anhand dieser Grundlagen definieren wir nun Grammatiken, die durch Kantenersetzung auf Hypergraphen operieren.

Definition 2.2.2 (Produktionsregel) *Gegeben die Menge von Hypergraphen H_Σ sowie eine Menge von Nichtterminalen $N \in \Sigma$. Eine Produktionsregel $p = (X, H)$ ist ein geordnetes Paar mit $X \in N$ und $H \in H_\Sigma$.*

Bemerkung 2.2.1 *Im Folgenden wird eine Produktionsregel (X, H) auch als $X \rightarrow H$ geschrieben und wir bezeichnen X als die „linke Seite“ der Produktion, analog H als „rechte Seite“.*

Definition 2.2.3 (Hyperkantenersetzungsgrammatik) Sei Σ eine Alphabet. Eine kontextfreie Hyperkantenersetzungsgrammatik G ist ein 4-Tupel (N, T, P, S) mit:

- N : Menge von Nichtterminalen, wobei $N \in \Sigma$
- T : Menge von Terminalen, wobei $T = \Sigma \setminus N$
- P : Menge von Produktionsregeln über N
- S : Axiom bzw. Ausgangsgraph mit $S \in H_\Sigma$

Bemerkung 2.2.2 Die Menge aller HRGs über dem Alphabet Σ bezeichnen wir mit HRG_Σ . Die Menge der X -Grammatikregeln, d.h. alle Regeln der Form $X \rightarrow H$, einer HRG G wird im Folgenden mit G^X bezeichnet.

Nachdem definiert wurde, wie eine Hyperkantenersetzungsgrammatik aussieht, soll nun geklärt werden, was wir unter der Sprache verstehen, die diese Grammatik beschreibt. Bei Stringgrammatiken lassen sich „Sätze“ einer Sprache bilden, indem man Nichtterminale - ausgehend vom Startsymbol - sukzessive anhand der Produktionsregeln ersetzt. Übertragen auf Hypergraphgrammatiken werden neue Hypergraphen abgeleitet, indem man ausgehend vom Axiom, Hyperkanten durch Hypergraphen ersetzt.

Definition 2.2.4 (Ableitung) Seien H_1, H_2 zwei Hypergraphen über dem Alphabet Σ . H_2 lässt sich mittels einer Produktionsregel $p : (X, K)$ von H_1 ableiten, wenn in H_1 eine Hyperkante e mit $\text{lab}(e) = X$ existiert, $H_1[K/e]$ zulässige Hyperkantenersetzung ist und $H_2 = H_1[K/e]$.

Bemerkung 2.2.3 Lässt sich ein Hypergraph H_2 durch eine Produktionsregel p von einem Hypergraphen H_1 ableiten, schreiben wir verkürzt auch $H_1 \xrightarrow{p} H_2$.

Definition 2.2.5 (Sprache einer Hypergraphgrammatik) Die Sprache, die eine Hypergraphgrammatik $G = (N, T, P, S)$ beschreibt, wird mit $L(G)$ bezeichnet und besteht aus allen Hypergraphen, die ausschließlich Terminalkanten besitzen und vom Axiom S ausgehend, abgeleitet werden können, d.h.

$$L(G) = \{H \in H_T \mid S \xrightarrow{P^*} H\}.$$

Beispiel 2.2.1 Zum besseren Verständnis von Hyperkantenersetzungsgrammatiken, dem Durchführen von Kantenersetzung und zugehörigen oben eingeführten Definitionen, betrachten wir an dieser Stelle ein ausführliches Beispiel. Als Grundlage dient die Datenstruktur aller binärer Bäume mit verketteter Blattfront, die uns auch in den nachfolgenden Kapiteln immer wieder als Fortführung dieses Beispiels begleiten wird. Denn sie ist nicht unnötig komplex und bietet trotzdem eine breite Palette an Eigenschaften, die viele Problematiken bei der Erstellung einer Heapabstraktionsgrammatik aufzeigen können. Einen Beispielbaum mit verketteter Blattfront ist in Abbildung 2.2 gegeben. Eine Grammatik G , die jeden saturierten Binärbäum mit verketteter Blattfront erzeugen kann, ist in Abbildung 2.3 zu finden.

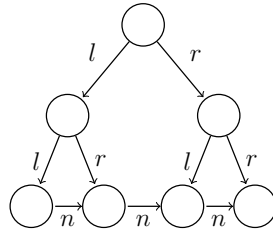


Abbildung 2.2: Beispielgraph: binärer Baum mit verketteter Blattfront

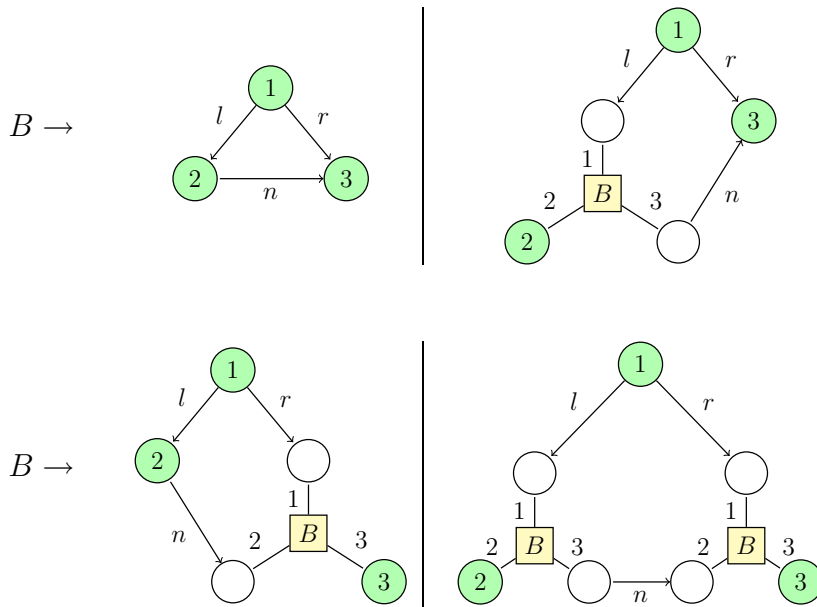


Abbildung 2.3: Grammatik: binärer Baum mit verketteter Blattfront

In Abbildung 2.3 sind vier Produktionsregeln $(B, H_1), \dots, (B, H_4)$ gegeben. Die linken Regelseiten sind ausschließlich mit dem Nichtterminal B beschriftet, daher gilt in diesen Fall $G^B = G$. Die

Hypergraphen der vier rechten Regelseiten sind alle vom Rang 3, d.h. sie besitzen drei externe Knoten. Der Rang der Hyperkanten mit Beschriftung B aus der zweiten, sowie aus allen weiteren Regeln, beträgt wiederum 3, da sie jeweils 3 anliegende Knoten besitzt.

Der Vollständigkeit halber schauen wir uns noch einen Ableitungsschritt an, um den Vorgang der Kantenersetzung am Beispiel deutlich zu machen. Betrachten wir den Hypergraphen aus der zweiten Regel (B, H_2) der Beispielgrammatik. Die vorhandene Nichtterminalkante e mit der Beschriftung B soll gemäß einer Produktionsregel $p : (B, H)$ ersetzt werden. Dies ist natürlich nur möglich, falls $H_2[H/e]$ eine zulässige Kantenersetzung induziert. Wählen wir hier die einzige Terminalregel der Grammatik – Regel 1 – gilt offensichtlich $rk(B) = rk(H_1)$ und damit bildet $H_2[H_1/e]$ eine zulässige Kantenersetzung. Abbildung 2.4 zeigt den resultierenden Hypergraphen H_2' des Ableitungsschrittes. Dieser Ableitungsschritt lässt sich damit verkürzt durch $H_2 \xrightarrow{P} H_2'$ darstellen. Da wie oben beschrieben $rk(B) = rk(H_2)$ gilt, lässt sich H_2 wiederum aus dem B -Henkel herleiten, d.h. $B(3)^\bullet \Rightarrow H_2 \Rightarrow H_2'$. Weiterhin ist anzumerken, dass H_2' ein Terminalgraph ist, d.h. $H_2' \in H_T$ und in der Sprache $L(G)$ liegt, falls $S \xrightarrow{P^*} H_2$.

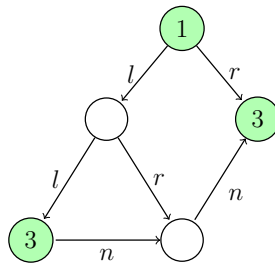


Abbildung 2.4: Kantenersetzung: binärer Baum mit verketteter Blattfront

2.3 Hypergraphen und HRGs: grundlegende Eigenschaften

Im weiteren Verlauf sind einige grundlegende Eigenschaften, die Hypergraphen oder Hyperkantenersetzungsgrammatiken von Haus aus mitbringen, sehr hilfreich. Auch ist es manchmal nötig, zusätzliche Anforderungen an Graph oder Grammatik zu stellen, um diese sinnvoll einsetzen zu können. In diesem Abschnitt sollen allgemein bekannte Eigenschaften erläutert werden.

2.3.1 Inhärente Eigenschaften

Zu den bekanntesten und wichtigsten Eigenschaften bzw. Folgerungen, die per Definition bereits von jeder HRG erfüllt sind, gehört die Kontextfreiheit. Ein formaler Beweis dieser Eigenschaft für HRGs ist in [Eng97] zu finden. Er benötigt einige grundlegende Eigenschaften der Kantenersetzung in Hypergraphen, die daher noch vor dem Kontextfreiheitslemma für HRGs betrachtet werden sollen. Da sie bereits durch andere Formalismen, die auf Ersetzung anhand von Grammatikregeln basieren, bekannt sind, folgt hier nur eine kurze Aufzählung. Für zugehörige Beweise siehe [Cou87].

Sequenzierung & Parallelisierung Sei H ein Hypergraph mit Nichtterminalkanten $e_1, \dots, e_n \in E_H$ und H_i ein Hypergraph mit $rk(H_i) = rk(e_i)$, $i \in [1, \dots, n]$. Dann gilt:

$$H[H_1/e_1, \dots, H_n/e_n] = H[H_1/e_1] \dots [H_n/e_n]$$

Konfluenz Sei H ein Hypergraph mit zwei Nichtterminalkanten $e_1, e_2 \in E_H$ und H_i ein Hypergraph mit $rk(H_i) = rk(e_i)$, $i \in \{1, 2\}$. Dann gilt:

$$H[H_1/e_1][H_2/e_2] = H[H_2/e_2][H_1/e_1]$$

Assoziativität Seien H, H_1, H_2 Hypergraphen mit Nichtterminalkanten $e_1 \in E_H$ und $e_2 \in E_{H_1}$, so dass $rk(e_1) = rk(H_1)$ und $rk(e_2) = rk(H_2)$. Dann gilt:

$$H[H_1/e_1][H_2/e_2] = H[H_1[H_2/e_2]/e_1]$$

Die Eigenschaft der Sequenzierung und Parallelisierung besagt, dass gleichzeitige Ausführung von Kantenersetzung in demselben Hypergraphen resultiert wie die Ausführung der einzelnen Kantenersetzungen hintereinander. Dies ist auch der Grund, warum es für Definition 2.2.1 ausreichend ist, die Ersetzung einer einzelnen Hyperkante zu betrachten. Konfluenz folgt eigentlich direkt aus der Sequenzierung und Parallelisierung und beschreibt die Tatsache, dass es keinen Unterschied macht, in welcher Reihenfolge die Nichtterminalkanten des Ausgangsgraphen durch Hypergraphen ersetzt werden. Die Assoziativität der Kantenersetzung besagt, dass die Kantenersetzung nur lokale Auswirkungen auf den Hypergraphen hat, in dem sie angewandt wird. Der resultierende Hypergraph bleibt derselbe, egal ob im Ausgangshypergraphen eine Kante ersetzt und in diesem neuen Teilgraphen wiederum Kantenersetzung durchführt wird, oder ob erst die Kantenersetzung in einem Hypergraphen durchführt wird, um diesen dann für eine Kante im Ausgangsgraphen austauschen.

Aus den obigen Eigenschaften lässt sich die Kontextfreiheit von Hyperkantenersetzungsgrammatiken mithilfe von Induktion direkt folgern. Für String-Grammatiken ist die Kontextfreiheit eine verbreitete Eigenschaft. Ihr Hauptmerkmal ist die implizite Aussage, dass Ableitungen in Teilableitungen zerlegt und beliebig wieder zusammengefügt werden können. Da dieses wichtige Merkmal auch für HRGs gilt, wird das folgende Lemma häufig als Kontextfreiheitslemma für HRGs bezeichnet. Detaillierte Beweise sind in [Eng97, Hab92, Lau88] zu finden. Zur Überprüfung, ob für eine Sprache eine kontextfreie HRG existiert, gibt es – völlig analog zu String-Grammatiken – ein Pumping-Lemma, welches hier nicht näher betrachtet werden soll. Details zum Pumping-Lemma sind in [Hab92] zu finden.

Lemma 2.3.1 (Kontextfreiheit von HRGs) *Gegeben eine HRG $G = (N, T, P, S)$ über dem Alphabet Σ , ein Hypergraph $H \in HRG_\Sigma$ und ein Terminalgraph $K \in HRG_T$. Seien e_1, \dots, e_n die Nichtterminalkanten von H mit $lab_H(e_i) = X_i$, $i \in [1, \dots, n]$. Dann gilt:
 $H \Rightarrow^* K$ gdw. $\exists K_1, \dots, K_n \in HRG_T$ mit $K = H[K_1/e_1] \dots [K_n/e_n]$ und $X_i(rk(e_i))^\bullet \Rightarrow^* K_i$ für alle $1 \leq i \leq n$.
 Es gilt sogar, dass die Länge der Ableitung $H \Rightarrow^* K$ gleich der Summe der Längen der Ableitungen $X_i(rk(e_i))^\bullet \Rightarrow^* K_i$ für $1 \leq i \leq n$ ist.*

An dieser Stelle sei noch erwähnt, dass einige bekannte Normalformen für String-Grammatiken in sehr ähnlicher Form auch für HRGs existieren und ihre Konstruktion fast analog verläuft. Im Kapitel 4 wird ein Vertreter – die Greibach Normalform – für HRGs ausführlich diskutiert.

Einen ähnlichen Sachverhalt beschreibt auch das Dekompositionslemma, welches sich als Folgerung aus den obigen Eigenschaften von HRGs ergibt. Für den Beweis sei an dieser Stelle lediglich auf [Hab92, Cou87, Lau88] verwiesen.

Lemma 2.3.2 (Dekompositionslemma) *Gegeben eine HRG $G = (N, T, P, S)$ und Hypergraphen $H, K \in HRG_{(N \cup T)}$. Seien e_1, \dots, e_n Nichtterminalkanten von H mit $lab_H(e_i) \in N$, $1 \leq i \leq n$. Dann gilt:
 $(H \Rightarrow^* K) \Leftrightarrow (\exists K_1, \dots, K_n \in HRG_{(N \cup T)} : K = H[K_1/e_1, \dots, K_n/e_n] \wedge X_i(rk(e_i))^\bullet \Rightarrow^* K_i, \forall 1 \leq i \leq n)$*

2.3.2 Spezielle Eigenschaften

Betrachten wir noch einige weitere Anforderungen, die des öfteren an HRGs gestellt werden, beginnend mit den von String-Grammatiken bekannten Eigenschaften der Produktivität und des Wachstums.

Definition 2.3.1 (Produktivität einer HRG) Gegeben eine HRG $G \in \text{HRG}_\Sigma$. Die Grammatik G ist produktiv, wenn sie ausschließlich Nichtterminale besitzt, aus denen Terminalgraphen (Graphen, die nur Terminalkanten enthalten) ableitbar sind, d.h. $\forall X \in N_\Sigma : L(G, X^\bullet) \neq \emptyset$.

Um die Bedeutung der wachsend-Eigenschaft für Hyperkantenersetzungsgrammatiken erfassen zu können, wird ein Maß für die Größe eines Hypergraphen benötigt, die wir durch seine Anzahl an Kanten und Knoten definieren.

Definition 2.3.2 (Größe eines Hypergraphen) Sei $H = (V, E, \text{lab}, \text{att}, \text{ext})$ ein Hypergraph. Wir bezeichnen seine Größe mit $|H|$, wobei $|H| = |V| + |E|$.

Definition 2.3.3 (Wachsende HRG) Gegeben eine HRG $G \in \text{HRG}_\Sigma$. Die Grammatik G ist wachsend, falls für jede ihrer Regeln (X, H) mit $H \notin \text{HG}_T$ die rechte Produktionsseite größer ist als die linke Produktionsseite, d.h. $\forall X \rightarrow H \in G : \text{lab}_H(E_H) \cap N_\Sigma \neq \emptyset \rightarrow |X^\bullet| < |H|$.

Eine weitere Anforderung ist die Isolationsfreiheit von Hypergraphen. Sie besagt, dass es keine alleinstehenden externen Knoten geben darf.

Definition 2.3.4 (Isolationsfreiheit) Ein Hypergraph H ist isolationsfrei, wenn alle ihm zugehörigen externen Knoten mindestens eine anliegende Kante besitzen, d.h. einen Grad ≥ 1 haben. Eine HRG G ist isolationsfrei, wenn alle rechten Produktionsregelseiten isolationsfrei sind.

Lemma 2.3.3 (Erhaltung der Isolationsfreiheit [Eng92]) Gegeben zwei isolationsfreie Hypergraphen H, K . Wenn H eine Nichtterminalkante e besitzt mit $rk(e) = rk(K)$, dann ist auch der Hypergraph $H[K/e]$ isolationsfrei.

Bisher wurden keine weiteren Anforderungen an HRGs gestellt, was die Beschriftung oder die Form der Hyperkanten betrifft. So dürfen zum Beispiel Hyperkanten mit verschiedenen Rängen dieselben Bezeichner tragen oder mehrere Tentakel einer Hyperkante an ein und demselben Knoten anliegen. Spezielle HRGs, in denen diese Fälle nicht auftreten dürfen, werden als typisiert bzw. wohlgeformt bezeichnet.

Definition 2.3.5 Eine HRG $G = (N, T, P, S)$ wird typisiert genannt, wenn eine Funktion $ltype : N \rightarrow \mathbb{N} \times \mathbb{N}$ existiert, so dass Folgendes erfüllt ist: $(\forall R \text{ mit } A \rightarrow R \in P, \forall e \in E_R \text{ mit } \text{lab}_R(e) \in N : ltype(\text{lab}_R(e)) = rk(e)) \wedge (\forall (A, R) \in P : ltype(A) = rk(R))$.

Definition 2.3.6 Ein Hypergraph H_Σ ist wiederholungsfrei, falls $\text{ext}_H(i) \neq \text{ext}_H(j)$ für alle $i, j \in [1, \text{rk}(H)]$ mit $i \neq j$. Eine Hyperkante $e \in E_H$ ist wiederholungsfrei, falls $\text{att}_H(e, i) \neq \text{att}_H(e, j)$ für alle $i, j \in [1, \text{rk}(e)]$ mit $i \neq j$. Eine HRG ist wiederholungsfrei, falls alle ihre rechten Produktionsregelseiten wiederholungsfrei sind.

Definition 2.3.7 Eine HRG $G = (N, T, P, S)$ ist wohlgeformt, falls sie wiederholungsfrei ist und alle mit Bezeichnern aus N versehenen Hyperkanten in G wiederholungsfrei sind.

Dass diese Eigenschaften keine Einschränkung für HRGs darstellen, zeigen das Typisierungstheorem und das Wohlgeformtheitstheorem aus [Hab92].

Theorem 2.3.1 (Typisierungstheorem) Zu jeder HRG lässt sich eine äquivalente, typisierte HRG konstruieren.

Theorem 2.3.2 (Wohlgeformtheitstheorem) Zu jeder HRG lässt sich eine äquivalente, wohlgeformte HRG konstruieren.

Kommen wir an dieser Stelle noch zur Apex-Eigenschaft. Anhand dieser lassen sich HRGs kategorisieren, dessen Sprache nur aus Hypergraphen mit beschränktem Grad besteht.

Definition 2.3.8 Eine HRG $G = (N, T, P, S)$ wird als apex bezeichnet, falls für jede Produktionsregel und darin an jedem externen Knoten ausschließlich Terminalkanten anliegen. Analog wird ein Knoten v als apex bezeichnet, wenn an ihm nur Terminalkanten anliegen.

Wie bereits erwähnt lässt sich zeigen, dass HRGs mit Apex-Eigenschaft genau die Sprachen beschreiben, die nur aus Hypergraphen mit beschränktem Grad bestehen. Theorem 2.3.3 formalisiert diese Eigenschaft. Für weitere Informationen und einem ausführlichen Beweis zu dieser Kategorisierung sei auf [EHL94] verwiesen.

Theorem 2.3.3 ([EHL94]) HRGs mit Apex-Eigenschaft erkennen genau die Hypergraph-Sprachen mit beschränktem Grad.

2.4 Graphalgorithmen für HRGs

Beim automatisierten Lernen von Hyperkantenersetzungsgrammatiken zur Heapabstraktion benötigen wir Algorithmen zur Lösung verbreiteter Probleme, dem Erreichbarkeitsproblem und den minimalen Schnitt für Hypergraphen. In diesem Abschnitt stellen wir die benötigten graphentheoretischen Grundlagen kurz vor und führen gängige Algorithmen ein.

Das Erreichbarkeitsproblem beschäftigt sich mit der Frage, ob im betrachteten Graphen $G = (V, E)$ ein Weg von einem Knoten v_1 zu einem weiteren Knoten v_2 existiert, $v_1, v_2 \in V$. Ist ein solcher Weg vorhanden, nennen wir v_2 von v_1 aus *erreichbar*. Dieses Problem lässt sich – auch in Hypergraphen – durch eine einfache Tiefen- oder Breitensuche lösen und stellt uns vor keine weiteren Schwierigkeiten, da die zur Heapabstraktion genutzten Hypergraphen endlich sind. Für die im Abschnitt 5 benötigte Art der Erreichbarkeit betrachten wir alle Kanten im Hypergraphen als ungerichtet.

Als weiteres Problem soll der minimale Schnitt (engl. *min-cut*) zwischen zwei Knoten eines Graphen betrachtet werden. Dabei definieren wir ihn wie folgt.

Definition 2.4.1 (Minimaler Schnitt) *Gegeben ein ungerichteter Graph $G = (V, E)$, sowie zwei Knoten $v_1, v_2 \in V$. Dann ist die Kantenmenge $E(v_1, v_2)$ ein Schnitt zwischen v_1 und v_2 , falls im Graphen $G = (V, E \setminus E(v_1, v_2))$ kein Weg von v_1 nach v_2 existiert. $E(v_1, v_2)$ ist ein minimaler Schnitt, falls gilt: $\forall E'(v_1, v_2) : |E(v_1, v_2)| \leq |E'(v_1, v_2)|$.*

Um zu einem gegebenen Graphen einen minimalen Schnitt zu bestimmen, gibt es mehrere mehr oder weniger bekannte Vorgehensweisen. Die meisten basieren auf Flussalgorithmen, die in einem Netzwerk den maximalen Fluss bestimmen um daraus einen minimalen Schnitt im Graphen ableiten zu können. Hierzu wird das Max-Flow-Min-Cut-Theorem benötigt, welches den genauen Zusammenhang von minimalem Schnitt und maximalem Fluss formuliert.

Theorem 2.4.1 (Max-Flow-Min-Cut-Theorem [EFS56]) *Der maximale Fluss hat genau den Wert des minimalen Schnitts im Netzwerk.*

Algorithmen, um den maximalen Fluss in einem Netzwerk zu berechnen, sind zahlreich. Der wohl Bekannteste, der an dieser Stelle zumindest genannt werden sollte, ist der Ford-Fulkerson-Algorithmus [FF56]. Seine grundlegende Idee besteht darin, sukzessive im Netzwerk den Fluss entlang flussvergrößernder Pfade zu erhöhen, bis dies nicht weiter möglich ist.

Zur Vermeidung der Notwendigkeit für Hypergraphen einen neuen Algorithmus entwickeln zu müssen, „übersetzen“ wir den betrachteten Hypergraphen im Vorfeld in einen gewöhnlichen Graphen. Diese Umformung muss natürlich eine solche Gestalt haben, dass nach Berechnung des minimalen Schnittes im gewöhnlichen Graphen bestimmbar ist, welche Hyperkanten entsprechend für den Hypergraphen im minimalen Schnitt enthalten sein müssen. Die „Übersetzung“ ist intuitiv: Hyperkanten werden durch den vollständigen Graphen zwischen den anliegenden Knoten ersetzt. Dieses Vorgehen garantiert, dass bei der Berechnung des Schnittes Hyperkanten mit niedrigerem denen mit höherem Rang vorgezogen werden.

Um den eigentlichen Algorithmus zur Berechnung des minimalen Schnittes anwenden zu können, muss der betrachtete Graph zuerst in ein Netzwerk konvertiert werden. Dazu wird jede ungerichtete Kante durch zwei gerichtete ersetzt und jeder Kante die Kapazität 1 zugeordnet. Gehört eine der gerichteten Kanten zum minimalen Schnitt des Netzwerkes, wird die entsprechende ungerichtete Kante im Eingabegraphen in seinen minimalen Schnitt aufgenommen.

Der – wie oben beschrieben – berechnete Schnitt wird im Anschluss wieder „rückübersetzt“. Dabei wird eine Hyperkante in den Schnitt des Hypergraphen aufgenommen, wenn mindestens eine ihrer Stellvertreterkanten in der Schnittmenge des gewöhnlichen Graphen vorhanden ist. Mithilfe dieser kleinen Umformungen lassen sich mit gängigen Algorithmen minimale Schnitte auch für Hypergraphen bestimmen.

Damit schließen wir das Kapitel über die Grundlagen ab und widmen uns der Heapabstraktion durch Hyperkantenersetzungsgrammatiken.

Kapitel 3

Heapabstraktion

Durch den immer größer werdenden Einfluß von objekt-orientierten Programmiersprachen sind heapbasierte, dynamische Datenstrukturen auch in der Softwareverifikation immer häufiger an der Tagesordnung. Objekte können zur Laufzeit - zur Kompilierzeit im Allgemeinen unabsehbar - generiert werden, wodurch die Menge der Heapzustände und damit der Zustandsraum potentiell unendlich wird. Unendliche Zustandsräume stellen aber für viele Verifikationstechniken wie z.B. das Model Checking ein Problem dar, da die Terminierung der Algorithmen nicht mehr garantiert werden kann. Als logische Konsequenz werden entweder Verifikationsalgorithmen gebraucht, die mit unendlichen Zustandsräumen umzugehen vermögen oder Methoden zur Abstraktion, um die potentiell unendliche Menge von Heapzuständen durch endliche Strukturen zu repräsentieren.

Eine Möglichkeit zur Abstraktion von Heapzuständen bieten die im vorangegangenen Kapitel eingeführten Hyperkantenersetzungsgrammatiken. Dabei wird ein Heapzustand durch einen Hypergraphen repräsentiert, der dann durch Anwendung der Regeln der Grammatik abstrahiert bzw. konkretisiert werden kann. Da der Zustand des Heaps durch Abarbeitung des zu verifizierenden Programmes verändert wird, müssen wir auch die Auswirkungen der einzelnen Anweisungen auf den Hypergraphen festlegen. Dies kann natürlich nicht geschehen ohne sich vorher auf eine grundlegende Programmiersprache geeinigt zu haben. Mit der Problematik der Heaprepräsentation durch Hypergraphen befasst sich der nächste Abschnitt.

3.1 Heaprepräsentation

Als Ausgangspunkt zur Heaprepräsentation wird eine Programmiersprache dienen, die mithilfe von Zeigern auf dem Heap operiert. Folglich muss es eine Möglichkeit geben Variablen und Zeiger direkt im Hypergraphen zu modellieren. Um dies zu ermöglichen werden zwei neue Mengen von

Terminalkanten Var_Σ und Sel_Σ eingeführt: Var_Σ enthält Kanten vom Rang 1, die mit Variablenamen beschriftet sind, wohingegen Sel_Σ Kanten vom Rang 2 enthält, die mit Selektoren beschriftet werden. Dabei bilden diese Bezeichner die gesamte Terminalmenge T des Alphabets Σ eines heaprepräsentierenden Hypergraphen, d.h. $T_\Sigma = Var_\Sigma \uplus Sel_\Sigma$. Ein Hypergraph, der mit diesen Mitteln einen Heapzustand repräsentiert, wird Heapkonfiguration genannt. Diese darf für jede Programmvariable höchstens eine Hyperkante enthalten.

Definition 3.1.1 *Gegeben ein Hypergraph H über dem Alphabet Σ . H ist eine Heapkonfiguration, falls er keine externen Knoten besitzt und es gilt:*

- $\forall x \in Var_\Sigma : |\{e \in E_H \mid lab_H(e) = x\}| \leq 1$
- $\forall v \in V_H : E_T(v) = \{e \in E_H(v) \mid lab_H(e) \in T\} \rightarrow lab_H(e_i) \neq lab_H(e_j), \forall e_i, e_j \in E_T(v), e_i \neq e_j$

Die Syntax der verwendeten Zeiger-Programme ist wie folgt definiert.

Definition 3.1.2 (Syntax der Zeiger-Programme) *Ein Zeiger-Programm besteht aus einer Folge von Anweisungen der Form:*

- $PExp := PExp$ (Zuweisung)
- $new(PExp)$ (Objekterzeugung)
- $if BExp goto n$ (bedingte Sprunganweisung)
- $goto n$ (Sprunganweisung)

Dabei seien die Zeiger- und bool'schen Ausdrücke $PExp$ und $BExp$ definiert als:

- $PExp := nil \mid x (\in Var_\Sigma) \mid x.s (x \in Var_\Sigma, s \in Sel_\Sigma)$
- $BExp := PExp = PExp \mid BExp \wedge BExp \mid \neg BExp$

An dieser Stelle fällt schnell auf, dass keine Anweisung existiert, die ein bestehendes Objekt entfernen kann. Realisiert wird das Löschen eines Objektes daher durch seine Isolierung, d.h. durch das Entfernen aller ihn involvierenden Zeiger. Ein Garbage Collector übernimmt das Auffinden und Entfernen ungenutzter Objekte, siehe [HNR09] für Details.

Bis hierher können wir nun einen Heapzustand durch einen Hypergraphen repräsentieren, wie sich allerdings Programmanweisungen in Letzterem auswirken ist noch ungeklärt. Die Definition

einer Semantik auf Hypergraphen stellt allerdings keine Probleme dar. Im Gegenteil ist sie sogar sehr intuitiv erfassbar, was im nächsten Beispiel kurz veranschaulicht werden soll.

Beispiel 3.1.1 Wir betrachten die ersten Zeilen eines Baum-Traversier-Algorithmus 3.1 und die Heapkonfiguration aus Abbildung 3.1(a). Die *if*-Anweisung in der ersten Zeile hat keine Auswirkungen auf die Heapkonfiguration, da weder Variablen zugewiesen noch Zeiger geändert werden. Sie beeinflusst allerdings den Programmverlauf: In diesem Fall wird sie zu *false* ausgewertet, weshalb ihre Anweisung *goto 15* ignoriert wird.

Algorithm 3.1 Baum-Traversier-Algorithmus

```

1: if root = nil goto 15
2: new(sen)
3: prev := sen
4: cur := root
5: next := cur.l
6: ...

```

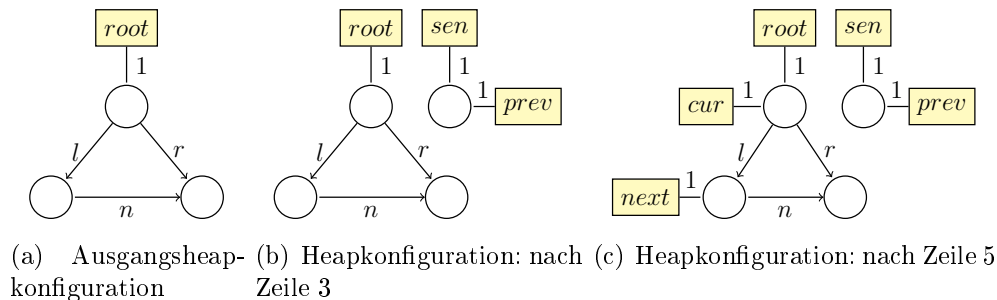


Abbildung 3.1: Beispiel: Heapkonfiguration

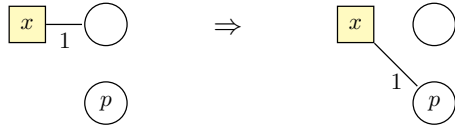
Die Objekterschaffung mit *new*(*sen*) fügt eine neue Variable hinzu und lässt diese auf einen neuen Knoten in der Heapkonfiguration zeigen. Im Anschluss wird in Zeile 3 einer neuen Variablen *prev* der Wert von *sen* zugewiesen. Die resultierende Heapkonfiguration ist in Abbildung 3.1(b) zu finden.

Anweisung 4 wiederholt das Vorgehen von 3 mit anderen Variablen. In Zeile 5 wird dann eine neue Variable *next* erstellt, die *cur.l* zugewiesen bekommt. Die Heapkonfiguration wird wiederum um eine Variable erweitert, die am Knoten anliegt, auf den der *l*-Selektor von *cur* zeigt. Es ergibt sich die Heapkonfiguration aus Abbildung 3.1(c)

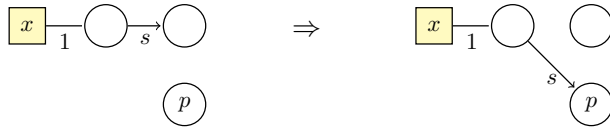
Da die Semantik der gegebenen Zeiger-Programmiersprache aus Definition 3.1.2 den üblichen Konventionen folgt, soll sie hier nur für die Programmanweisungen und ohne ausführliche Beschreibung formalisiert werden. Es sei noch erwähnt, dass bei Vorkommen einer nach der Programmiersprache unzulässigen Anweisung, eine Fehlerkonfiguration err eingeführt wird.

Definition 3.1.3 (Semantik der Programmanweisungen) Sei H eine Heapkonfiguration über Σ und $pExp \in PExp$ ein Zeigerausdruck.

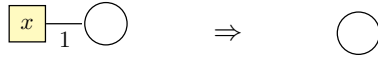
- $x := pExp$ mit $x \in Var_\Sigma$, sei außerdem p der Knoten, der sich durch den Zeigerausdruck $pExp$ ergibt



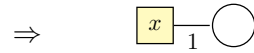
- $x.s := pExp$ mit $x \in Var_\Sigma, x \neq nil$, sei außerdem p der Knoten, der sich durch den Zeigerausdruck $pExp$ ergibt



- $x := nil$ ($x.s := nil$ analog)



- $new(x)$ mit $\nexists e \in E_H : lab(e) = x$



- $new(x.s)$, $x \neq nil$ und am mit x verbundenen Knoten existiert keine Kante mit der Beschriftung $s \in Sel_\Sigma$



- $x.s := pExp$ mit $x = nil$ und $new(x.s)$, $x = nil$ führen zu einer Fehlerkonfiguration H_{err}

3.2 Abstraktion & Konkretisierung

Zu einem dynamischen, heapbasierten Programm existiert eine potentiell unendliche Menge an Heapzuständen und damit auch Heapkonfigurationen, die bei unbeschränktem Heap ihrerseits

potentiell keine endliche Darstellung besitzen. Um zur Verifikation benötigte endliche Repräsentationen zu finden, betrachten wir die Abstraktion von Heapkonfigurationen. Die grundlegende Idee besteht darin, die vom jeweils nächsten Programmschritt unbeeinflussten Teile des Hypergraphen zu abstrahieren und erst bei Bedarf wieder zu konkretisieren. Als temporär nicht relevant gelten dabei alle Teilgraphen, in denen momentan keine Kanten e mit $lab(e) \in Var$ vorkommen und die nicht durch eine Anweisung im nächsten Schritt "betreten" werden können. Sind relevante Teilgraphen abstrahiert, ist die betrachtete Heapkonfiguration unzulässig und wird solange konkretisiert bis sie die Vorgaben erfüllt. Dadurch werden Operationen nur auf konkreten Teilgraphen ausgeführt und es wird keine Definition der Semantik für die abstrakten Teile der Graphen benötigt. Zur Hypergraphtransformation werden Regeln in Form einer Hyperkantenersetzungsgrammatik angegeben, wobei ihre übliche Anwendung der Konkretisierung dient, wohingegen ihre "Rückwärtsanwendung" Teile eines Hypergraphen abstrahiert.

Definition 3.2.1 (Zulässige Heapkonfiguration) *Gegeben eine Heapkonfiguration H über dem Alphabet $\Sigma = N \cup T$. H ist eine zulässige Heapkonfiguration, falls die Menge der Verletzungspunkte $VP(H) \subseteq E_H \times \mathbb{N}$ leer ist. Dabei ist das Paar (e, i) in $VP(H)$ enthalten, falls $lab(e) \in N_\Sigma \wedge \exists e' \in E_H, \exists v \in V_H : lab(e') \in Var_\Sigma \wedge att(e')(1) = v \wedge v = att_H(e)(i)$.*

Beispiel 3.2.1 *Machen wir uns mit den Begriffen Abstraktion, Konkretisierung und zulässige Heapkonfiguration anhand der Datenstruktur der Bäume mit verketteter Blattfront vertraut, dessen HRG bereits in Abbildung 2.3 eingeführt wurde.*

Die Heapkonfiguration aus Abbildung 3.2(a) besitzt einen Verletzungspunkt $(e, 1)$ mit $lab(e) = B$ an der Variablen $next$ und ist damit nicht zulässig. Um diesen Verletzungspunkt zu entfernen wenden wir eine Produktionsregel an, die am Knoten $v = nod(e, 1)$ Terminalkanten produziert. Wir wählen hier beispielhaft die einzige Terminalregel der Grammatik. Die entsprechende zulässige Konfiguration ist in Abbildung 3.2(b) zu sehen.

Die Heapkonfiguration ist allerdings noch nicht vollständig abstrahiert. Der linke Teilbaum enthält keine Variable und muss daher nicht konkret erhalten bleiben. Wir wenden daher dieselbe Regel wie im vorangegangenen Schritt an, diesmal rückwärts. Der resultierende, vollständig abstrahierte Hypergraph ist in Abbildung 3.2(c) zu finden.

Die Verwendung einer Hyperkantenersetzungsgrammatik zur Abstraktion von Heapzuständen birgt einige Problematiken, da sie ursprünglich zur einseitigen Regelanwendung konzipiert wurde. Dies führt unter Umständen dazu, dass z.B. die Terminierung des Abstraktionsverfahren nicht gewährleistet werden kann. Durch das Einführen zusätzlicher Anforderungen lassen sich

diese Probleme umgehen. Da der Heapabstraktion damit aus technischer Sicht nichts mehr im Wege steht, bezeichnen wir Grammatiken mit diesen zusätzlichen Eigenschaften Heapabstraktionsgrammatiken. Bevor die formale Definition letzterer eingeführt wird, soll zuvor anhand einiger Beispiele veranschaulicht werden um welche problematischen Fälle und Auswege aus diesen es sich konkret handelt.

Beispiel 3.2.2 (Produktivität) *Unproduktive Regeln in einer HRG bereiten nicht nur Probleme bei der Heapabstraktion, sie sind vor allem einfach unerwünschte und überflüssige Regeln, da aus ihnen keine Terminalgraphen mehr ableitbar sind. Betrachten wir die Grammatik für binäre Bäume aus Abbildung 2.3, die erweitert wird durch die Regeln aus Abbildung 3.3.*

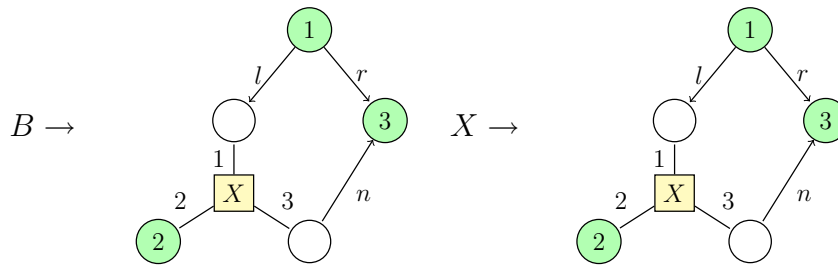


Abbildung 3.3: Bäume mit verketteter Blattfront: unproduktive Regeln

Entsteht durch Ableitung in irgendeinem Schritt eine Nichtterminalkante mit Bezeichner X im Hypergraphen, lässt sich dieser durch keinerlei Regeln in einen Terminalgraphen überführen. Unproduktive Regeln vergrößern die betrachtete Grammatik damit unnötig und führen dazu, dass abstrahierte Teilgraphen unter Umständen nicht mehr konkretisiert werden können. Sie sollen daher nicht betrachtet werden, was man durch einfaches Löschen unproduktiver Regeln erreicht.

Beispiel 3.2.3 (Wachstum) *Eine wachsende Grammatik muss gefordert werden, um die Terminierung der Abstraktion zu gewährleisten. Betrachten wir wiederum die Grammatik aus Abbildung 2.3 und erweitern diese um eine nicht wachsende Produktionsregel, siehe Abbildung 3.4.*

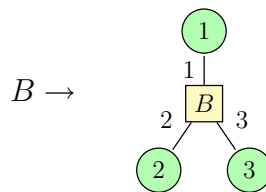


Abbildung 3.4: Bäume mit verketteter Blattfront: nicht wachsende Regel

Die neu hinzugekommene Produktionsregel lässt sich, sobald eine mit B beschriftete Nichtterminalkante vorhanden ist, sowohl beim Konkretisieren als auch beim Abstrahieren immer wieder anwenden ohne den resultierenden Hypergraphen zu verändern. Dies führt zu potentiell unendlichen Ableitungen und damit zu nicht terminierenden Abstraktionen.

Beispiel 3.2.4 (Variablenfreiheit) Die Problematik dieses Beispiels besteht darin, dass beim Abstrahieren wichtige Informationen über die Programmvariablen verloren gehen. Betrachten wir wiederum die Grammatik aus Abbildung 2.3, sowie eine zusätzliche Produktionsregel, die eine Variable enthält, zu sehen in Abbildung 3.5.

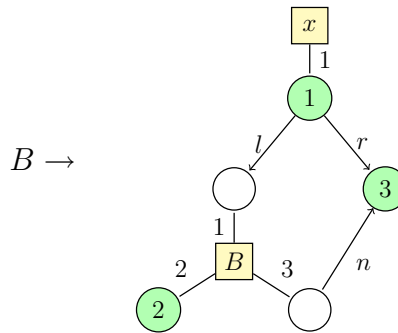


Abbildung 3.5: Bäume mit verketteter Blattfront: Regel mit Variable

Beim rückwärtigen Anwenden der neu hinzugekommenen Produktionsregel lassen sich nun Teilgraphen, in denen Variablen enthalten sind, abstrahieren. Die Informationen über die Variablen gehen vollständig verloren. Man kann im Nachhinein keine Aussage mehr über Anzahl und Art der Variablen in einem abstrahierten Teilgraphen mehr treffen. Daher verbieten wir das Vorkommen von Variablen in Produktionsregeln.

Nachdem wir die Problematik von Variablenvorkommen in Produktionsregeln in obigen Beispiel gesehen haben, formalisieren wir hier die Eigenschaft der Variablenfreiheit um Mißverständnisse zu vermeiden.

Definition 3.2.2 (Variablenfreiheit) Gegeben eine HRG $G \in HRG_{\Sigma}$. G ist variablenfrei, falls gilt: $\forall X \rightarrow H \in P_G : lab(E_H) \cap Var_{\Sigma} = \emptyset$.

Beispiel 3.2.5 (Typisierung) Bei mehrfacher Abstraktion und Konkretisierung kann es zu Füllen kommen, in denen die resultierenden Hypergraphen eine unzulässige Form annehmen. Um dies

zu veranschaulichen, betrachten wir die Datenstruktur der einfach verketteten Listen, dessen Objekte zusätzlich einen Zeiger auf ihren Vorgänger aufweisen können. Die zugehörige HRG ist in Abbildung 3.6 gegeben. Eine mögliche Abfolge aus Abstraktions- und Konkretisierungsschritt wird in Abbildung 3.7 eingeführt.

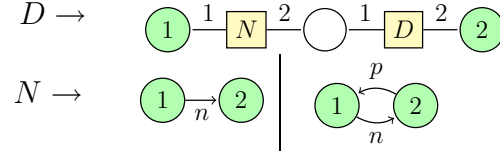


Abbildung 3.6: Liste mit pot. Rückwärtszeigern: nicht typisiert

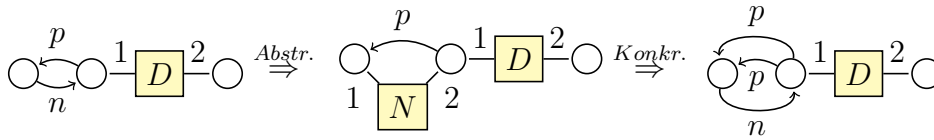


Abbildung 3.7: Liste mit pot. Rückwärtszeigern: Abstraktion & Konkretisierung

Der resultierende Hypergraph besitzt nun an einem Knoten zwei ausgehende Kanten mit derselben Beschriftung und stellt damit keine zulässige Heapkonfiguration mehr dar. Um dies zu verhindern, erweitern wir die Typisierung aus Abschnitt 2.3.2 im vorangegangenen Kapitel.

Als Typisierung bezeichnen wir die Eigenschaft, dass jedes Nichtterminal sowohl einen festen Rang als auch feste, ausgehende Terminalkanten besitzt. Letztere Eigenschaft formalisieren wir durch die Einführung einer Terminalfolge für Nichtterminale.

Definition 3.2.3 (Terminalfolge) Gegeben eine HRG $G = (N, T, P, S)$ sowie eine Produktionsregel $(X, R) \in P$. Sei G' die analoge Grammatik zu G mit R als Axiom, $G' = (N, T, P, R)$. Die Terminalfolge $terminal(X, R)$ enthält dann für jeden externen Knoten $ext(i) \in R$ mit $1 \leq i \leq rk(R)$ eine Terminalmenge T_i , wobei $T_i = \{t \in T \mid t \text{ ist ausgehende Kante am externen Knoten } ext_H(i) \text{ eines Graphen } H \in L(G')\}$.

Zuletzt passen wir die Definition der Typisierung an. Es lässt sich wieder zeigen, dass zu jeder HRG eine äquivalente, typisierte HRG existiert. Der Beweis verläuft vollständig analog zum Beweis von Theorem 2.3.1 aus Abschnitt 2.3.2.

Definition 3.2.4 Eine HRG $G = (N, T, P, S)$ wird typisiert genannt, wenn eine Funktion $ltype : N \rightarrow \mathbb{N} \times \mathbb{N}$ existiert, so dass Folgendes erfüllt ist:

- $(\forall R$ mit $A \rightarrow R \in P, \forall e \in E_R$ mit $lab_R(e) \in N : ltype(lab_R(e)) = rk(e)) \wedge (\forall (A, R) \in P : ltype(A) = rk(R)$.
- $\forall A_1 \rightarrow R_1, A_2 \rightarrow R_2 \in P : R_1 = R_2 \Rightarrow terminal(A_1, R_1) = terminal(A_2, R_2)$.

Beispiel 3.2.6 (Lokale Apex-Eigenschaft) Als wir uns zu Beginn dieses Abschnitts mit unzulässigen Konfigurationen und der Konkretisierung von diesen beschäftigt haben, sind wir stillschweigend davon ausgegangen, dass zu jeder Konfiguration Produktionsregeln existieren, die eine korrekte Konkretisierung zulassen. Dies ist jedoch nicht immer der Fall. Betrachten wir dazu wiederum die Grammatik für Bäume mit verketteter Blattfront aus Abbildung 2.3 und die Heapkonfiguration aus Abbildung 3.8.

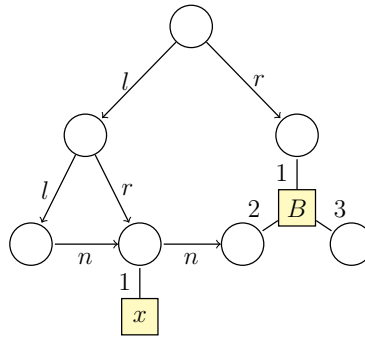


Abbildung 3.8: Bäume mit verkettete Blattfront: zul. Konfiguration

Wird im folgenden Schritt nun zum Beispiel die Anweisung $y := x.n$ ausgeführt, muss am Knoten des zweiten Tentakels der B-Kante konkretisiert werden. Zwei Regeln aus der Beispielgrammatik kommen dazu in Frage, nämlich diejenigen Regeln, die am externen Knoten 2 ausschließlich Terminalkanten besitzen. Egal welche der beiden Regeln man nun anwendet, man übergeht dabei immer die Heapkonfigurationen, in denen am betrachteten Knoten ein weiterer Teilbaum existiert. Dies darf beim Konkretisieren nicht passieren, da dabei mögliche, auftretende Heapzustände nicht korrekt wiederhergestellt werden können.

Um zu garantieren, dass zum Konkretisieren immer geeignete Produktionsregeln vorhanden sind, die es auch weiterhin erlauben die gesamte Sprache zu generieren, führen wir die lokale Apex-Eigenschaft ein. Diese geht sogar noch einen Schritt weiter und garantiert, dass beim Konkretisieren jeweils in einem Ersetzungsschritt eine zulässige Konfiguration erreicht werden kann. Um garantieren zu können, dass nach dem Konkretisieren eine äquivalente Sprache erkannt wird, ist

die lokale Apex-Eigenschaft allein jedoch nicht ausreichend. Daher fordern wir für eine Heapabstraktionsgrammatik, dass sie für jedes Nichtterminal $X \in N$ in Teilgrammatiken partitioniert werden kann, die alle X -Regeln umfassen und die zusammen mit den noch nicht zu betrachteten Nichtterminalregeln dieselbe Sprache erkennen wie die gesamte Heapabstraktionsgrammatik. Diese mit der lokalen Apex-Eigenschaft verknüpften Anforderungen finden sich im letzten Punkt der Anforderungsliste von Definition 3.2.6 wieder.

Die lokale Apex-Eigenschaft stellt eine Anpassung der Apex-Eigenschaft aus Kapitel 2 an die Heapabstraktion mithilfe von HRGs dar.

Definition 3.2.5 (Lokale Apex-Eigenschaft) Eine HRG $G = (N, T, P, S)$ wird für das Nichtterminal $X \in N$ und am externen Knoten $ext(i)$ mit $i \in [1, rk(X)]$ als lokal apex bezeichnet, falls $lab_H(E_H(ext(i))) \subseteq T_\Sigma, \forall X \rightarrow H \in G$.

Bemerkung 3.2.1 Wird darauf verwiesen, dass ein externer Knoten für eine Menge an Regeln lokale Apex-Eigenschaft besitzt, so ist analog zur obigen Definition gemeint, dass an diesem Knoten nur Terminalkanten anliegen.

Zur Sicherstellung der Korrektheit des Verfahrens muss die verwendete Hyperkantenersetzungsgrammatik die oben genannten Anforderungen erfüllen. Eine Grammatik, die die geforderten Eigenschaften besitzt, wird als Heapabstraktionsgrammatik bezeichnet. Weiterführende Details zum Ansatz der Heapabstraktion sind in [HNR09, RN08] zu finden.

Definition 3.2.6 (Heapabstraktionsgrammatik) Gegeben eine HRG $G \in HRG_\Sigma$. Weiterhin sei $\overline{G^X} := G \setminus G^X$. Man bezeichnet G als Heapabstraktionsgrammatik, falls

- G produktiv
- G wachsend
- G typisiert
- G variablenfrei
- für jedes Nichtterminal $X \in N$ mit Rang $rk(X) = k$ Teilgrammatiken $G_1^X, \dots, G_k^X \subseteq G^X$ existieren, so dass
 1. $\bigcup_{i=1}^k G_i^X = G^X$,
 2. $L(G_i^X \cup \overline{G^X}, X^\bullet) = L(G, X^\bullet), \forall 1 \leq i \leq k$,
 3. G_i^X lokal apex, $\forall 1 \leq i \leq k$.

3.3 Erweiterungen

Auf der einen Seite kann die Konstruktion einer HRG mit Apex-Eigenschaft zu einer deutlich größeren Produktionsregelmenge führen, verglichen mit einer äquivalenten Grammatik ohne Apex-Eigenschaft. Auf der anderen Seite beschränkt die lokale Apex-Eigenschaft die beschreibbaren Datenstrukturen auf diejenigen, die durch Grammatiken erkannt werden, dessen Sprache aus Graphen mit beschränktem Rang besteht. Diese beiden genannten Nachteile lassen sich durch die Lockerung der Apex-Anforderung in einem Fall ganz beseitigen oder im anderen Fall deutlich reduzieren. Wir erhalten dadurch einerseits eine kleinere Heapabstraktionsgrammatik und andererseits die Möglichkeit, Datenstrukturen mit unbeschränktem Grad abstrahieren zu können.

Die Idee besteht darin bei der Wiederherstellung einer zulässigen Konfiguration nur an Nichtterminalkanten zu konkretisieren, die durch eine Programmanweisung im nächsten Schritt durchlaufen werden können. Dies entspricht in dem in [HNR09] vorgestellten Framework genau denjenigen Nichtterminalkanten, die bei Konkretisierung zu mindestens einer ausgehenden Kante am betrachteten externen Knoten führen können, d.h. dessen Terminalfolge nicht leer ist.

Doch betrachten wir zuerst, welche Problemfälle uns hauptsächlich zur Lockerung der Apex-Eigenschaft bewogen haben.

Beispiel 3.3.1 *Betrachten wir die HRG aus Abbildung 3.9. Sie stellt die Datenstruktur von einfach verketteten Listen dar, in denen jedes Objekt zusätzlich über einen Zeiger auf den Wurzelknoten verfügt. Dass man zu dieser HRG keine äquivalente Grammatik mit Apex-Eigenschaft konstruieren kann, lässt sich leicht erkennen. Da am Wurzelknoten beliebig viele eingehende Kanten entstehen können, würden in der apex HRG ebensoviele Regeln benötigt werden um jeden Fall mit ausschließlich anliegenden Terminalkanten am betrachteten Knoten darzustellen. Dies würde zu einer unendlich grossen Anzahl von Produktionsregeln führen. Die hier beschriebenen Zusammenhänge sind in Theorem 2.3.3 nochmals konkret formuliert und in der gleichen Weise auf HRGs mit lokaler Apex-Eigenschaft übertragbar.*

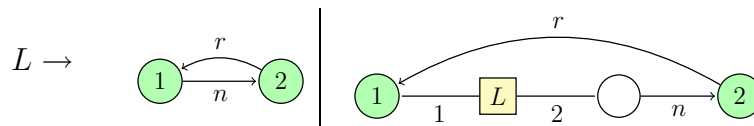


Abbildung 3.9: Einfach verkettete Liste mit Rückwärtszeigern: Grammatik

Um Datenstrukturen wie die Listen aus Abbildung 4.5, gewurzelte Bäume etc. durch eine Heapabstraktionsgrammatik darstellen zu können, muss die Apex-Eigenschaft entsprechend abgeändert

werden. Dabei ist es für unsere Zwecke ausreichend, lediglich einen unbeschränkten Eingangsgrad zu akzeptieren, da der Ausgangsgrad in jedem Fall im benutzten Framework durch die Anzahl der Terminalsymbole im Alphabet beschränkt ist, siehe auch Bemerkung 3.3.1.

Bemerkung 3.3.1 *Es wird im Allgemeinen angenommen, dass der Ausgangsgrad eines jeden Knotens in den betrachteten Hypergraphen beschränkt ist.*

Diese Annahme ist keine weitere Einschränkung der Datenstrukturen, die durch Heapabstraktion verifizierbar sind, sondern vielmehr eine Folgerung aus den Gegebenheiten. Denn jedem Objekt darf jeder Selektor höchstens einmal zugeordnet werden. Daraus folgt, dass jeder Knoten im Terminalgraphen höchstens so viele ausgehende Kanten besitzen darf wie Terminale vorhanden sind und die Menge der Terminalsymbole endlich ist.

Zur Lockerung der lokalen Apex-Eigenschaft definieren wir zunächst diejenigen Tentakel der Nichtterminalkanten genauer, die bei Überprüfung auf unzulässige Konfigurationen nicht betrachtet werden müssen, da sie nichts an der Zulässigkeit einer Konfiguration ändern. Wir bezeichnen sie als Reduktionstentakel.

Definition 3.3.1 (Reduktionstentakel) *Sei $G \in HRG_{\Sigma}$. (X, i) ist genau dann ein Reduktionstentakel mit $X \in N$, $i \in rk(X)$, wenn die Terminalmenge für den i -ten externen Knoten von X leer ist, d.h. $terminal(X)(i) = \emptyset$.*

Dementsprechend muss die Definition der lokalen Apex-Eigenschaft auf die neuen Gegebenheiten übertragen werden.

Definition 3.3.2 (Erweiterte lokale Apex-Eigenschaft) *Ein Knoten v besitzt erweiterte, lokale Apex-Eigenschaft, wenn alle an ihm anliegenden Kanten entweder ausschließlich durch Reduktionstentakel mit v verbunden oder Terminalkanten sind.*

Diese erweiterte, lokale Apex-Eigenschaft benutzen wir von hier an anstelle der lokalen Apex-Eigenschaft. In allen Zusammenhängen, in denen die lokale Apex-Eigenschaft genutzt wurde, ist sie durch ihre Erweiterung auszutauschen, auch wenn das hier nicht explizit geschieht. Wird in Zukunft der Begriff lokale Apex-Eigenschaft genutzt, ist – sofern nicht anders angegeben – ihre Erweiterung gemeint.

Bemerkung 3.3.2 *Im Folgenden wird die Menge der an einem Knoten v anliegenden Kanten $E(v)$ in Kanten, die ausschließlich über Reduktionstentakel mit v verbunden sind $E^{red}(v)$ und allen anderen Kanten aus $E(v) \setminus E^{red}(v)$ partitioniert. Analog dazu wird ein Knoten nur noch dann als Nichtterminalknoten bezeichnet, wenn er über mindestens eine nicht ausschließlich über Reduktionstentakel verbundene anliegende Nichtterminalkante verfügt.*

Kapitel 4

Lokale Apex-Eigenschaft für Heapabstraktionsgrammatiken

Ein notwendiger Schritt bei der automatischen Generierung einer geeigneten Heapabstraktionsgrammatik besteht darin, sie auf die Eigenschaften zu überprüfen, die eine solche gegenüber einer einfachen HRG auszeichnet. Denn nur wenn die generierte Grammatik alle in Kapitel 3 eingeführten Anforderungen erfüllt, kann sie fehlerfrei für die Verifikation von dynamischen Datenstrukturen eingesetzt werden. Weiterhin ist es wünschenswert ein Verfahren zur Hand zu haben, welches eine unzulässige Grammatik, die eine oder gar mehrere geforderte Eigenschaften verletzt, in eine zulässige Grammatik umformen kann.

In diesem Kapitel wird die Überprüfung auf lokale Apex-Eigenschaft betrachtet und ein Verfahren vorgestellt, dass eine beliebige HRG in eine äquivalente HRG mit dieser Eigenschaft überführt.

4.1 Konstruktion der lokalen Apex-Eigenschaft

Lokale Apex-Eigenschaft für eine beliebige HRG herzustellen ist komplex und die Überführung in eine äquivalente zulässige Grammatik bringt in vielen Fällen, wie im vorigen Kapitel erwähnt, den Nachteil mit sich, dass die Anzahl der Produktionsregeln um ein Vielfaches ansteigt [Gro09, EHL94].

In diesem Abschnitt wird gezeigt werden, dass die Existenz einer äquivalenten HRG mit lokaler Apex-Eigenschaft mit einer kleinen Einschränkung immer gewährleistet ist. Durch die Lockerung der lokalen Apex-Eigenschaft trifft dies auch auf Datenstrukturen mit unbeschränktem Ein-

gangsgrad zu. Weiterführend wird eine Methode zur Konstruktion der geforderten Grammatik betrachtet.

Für eine Hypergraph-Sprache L mit *beschränktem Knotengrad* lässt sich zeigen, dass immer eine HRG G mit Apex-Eigenschaft existiert mit $L = L(G)$ [Eng92]. Diese Aussage lässt sich auch für erweiterte, lokale Apex-Eigenschaft auf Sprachen mit beschränktem Ausgangsknotengrad übertragen. Als direkte Folgerung aus dieser Aussage werden wir sehen, dass selbst die erweiterte lokale Apex-Eigenschaft weiter gelockert werden kann ohne die Korrektheit der Heapabstraktionsgrammatik zu verlieren. Fassen wir hier kurz die Ergebnisse und Vorgehensweise aus [Eng92] zusammen, bevor wir die Modifikation zur Überführung von HRGs mit beschränktem Ausgangsknotengrad betrachten.

Grundlage des Beweises der Aussage im obigen Abschnitt bildet die Definition einer speziellen Normalform in [Eng92]. Liegt die betrachtete HRG in dieser Normalform vor, lässt sich durch die Beschränktheit des Knotengrades zeigen, dass die vorliegende HRG dann auch durch einfache Kantenersetzung in Apex-Eigenschaft überführt werden kann. Diese spezielle Normalform wird in [Eng92] als Greibach Normalform bezeichnet und ist wie folgt definiert.

Definition 4.1.1 (Greibach Normalform) *Eine HRG G ist in Greibach Normalform, wenn sie die folgenden zwei Eigenschaften besitzt:*

- G ist *isolationsfrei*.
- keine Produktionsregel $X \rightarrow H \in P$ besitzt Nichtterminalknoten vom Grad 1.

Der entsprechende Beweis, dass zu jeder HRG eine äquivalente HRG in Greibach Normalform existiert und konstruiert werden kann, ist komplex und die exakte Konstruktion der Normalform ist in [Eng92] implizit im Beweis enthalten.

Hauptidee dieser Konstruktion ist das sogenannte „Falten“ von Teilgraphen. Man konstruiert dabei neue Nichtterminale und Produktionsregeln für die Teilgraphen, die „zwischen“ den, die Greibach Normalform verletzenden, Knoten liegen, man „faltet“ diese Teilgraphen sozusagen. Somit können die Knoten, die vorher in den Konkretisierungsschritten die anliegende Nichtterminalkante wieder und wieder in eine weitere Nichtterminalkante überführt haben, bis der dazwischen liegende Teilgraph vollständig abgeleitet worden war, durch einen Konkretisierungsschritt mithilfe des neu eingeführten Nichtterminals diesen Teilgraphen in einem Schritt ableiten. Damit wird an diesen Stellen wieder Greibach Normalform gewährleistet. Den Vorgang des „Faltens“ führt man so oft hintereinander aus, wie die Greibach Eigenschaft verletzende Knoten an einer Nichtterminalkante vorhanden sind. Mit jedem dieser Schritte eliminiert man dabei einen der

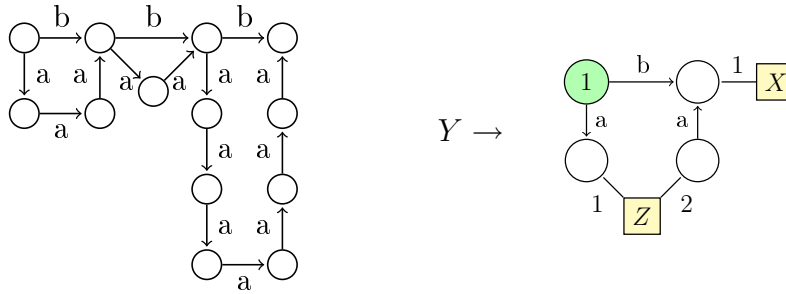


Abbildung 4.3: Datenstruktur & HRG: gefalteter Grashüpfer

Fassen wir die genaue Greibach Normalform Konstruktion zum tieferen Verständnis an dieser Stelle noch kurz zusammen.

Konstruktion: Greibach Normalform

Eingabe: HRG G mit beschränktem Knotengrad

Ausgabe: HRG G' in Greibach Normalform

Berechne die maximale Anzahl $maxp$ der an einer Nichtterminalkante anliegenden externen Knoten vom Grad 1 in allen Regeln. Bei jeder Ausführung der unterhalb aufgeführten Schritte, verringert sich $maxp$ um mindestens 1. Wiederhole Schritt 1 bis 5 solange, bis $maxp = 0$. Sei E_H^{max} die Menge der Nichtterminalkanten aus H , die $maxp$ anliegende externe Knoten vom Grad 1 besitzen. Diese Kanten werden maximal genannt.

1. Erstelle äquivalente Grammatik G_1 , die lediglich zusätzliche Informationen über die unzulässigen externen Knoten in den Nichtterminalkanten enthält. Die Produktionsregeln von G_1 werden unterteilt in vier verschiedene Klassen:

Alte Produktionsregeln/old productions: Entsprechen den Regeln, die keine Nichtterminalkante auf der rechten Regelseite E^{max} enthalten.

Einstiegsregeln/initial productions: Beinhaltet Regeln, die den Anfang einer „Knoten weiterreichenden“ Ableitungskette bilden, d.h. der Einstieg in Kantenersetzungen, die unzulässige Knoten an wiederum unzulässige Knoten weitergeben. Nichtterminalkanten aus E^{max} werden dabei umbenannt, die neuen Bezeichner enthalten Informationen zu den weitergegeben Knoten.

Mittelregeln/middle productions: Wie Einstiegsregeln, mit neuem Bezeichner auf der linken Regelseite.

Endregeln/final productions: Wie alte Produktionsregeln, mit neuem Bezeichner auf der linken Regelseite.

2. Für jeweils zwei Mittelregeln $(X_1, H), (X_2, H')$ mit $e \in E_H \wedge lab(e) = X_2$ wird ein neues

Nichtterminal $\langle X_1, X_2, \phi \rangle$ erstellt. Dieses Nichtterminal wird während der weiteren Konstruktion Regeln erhalten, die eine beliebig lange Ableitungskette von Mittelregeln – also Regeln die momentan nicht in Greibach Normalform sind – simulieren sollen. ϕ ist dabei eine Hilfsfunktion, die protokolliert, welche externen Knoten von Mittelregel X_1 an welche externen Knoten von Mittelregel X_2 weitergereicht werden.

3. Es werden zwei Arten von Hilfsregeln konstruiert:

Zwischenregeln/intermediate productions: Für jedes im vorangegangenen Schritt neu erstellte Nichtterminal $\langle X_1, X_2, \phi \rangle$, wird eine Zwischenregel $X_1 \rightarrow H$ erstellt. Sie enthält lediglich zwei Nichtterminalkanten X_2 und $\langle X_1, X_2, \phi \rangle$ und übernimmt die externen Knoten von X_1 . Diese Regeln sind Stellvertreter einer Ableitung $X_1 \rightarrow H$ und $H[X_2/e]$.

Ausstiegsregeln/tail productions: Man konstruiert Ausstiegsregeln zu jeder Zwischenregel $\langle X_1, X_2, \phi \rangle$, indem man in letzterer die Kante mit dem Bezeichner X_2 durch eine Endregel ersetzt. Diese Regeln repräsentieren eine beliebige Anzahl von Ableitungsschritten anhand von Mittelproduktionen, abgeschlossen durch eine Endproduktion.

4. Nun beginnt die eigentliche Konstruktion der Grammatik in Greibach Normalform. Folgende Regeln werden zu G' hinzugefügt:

Alte Produktionsregeln/old productions: Werden übernommen aus Schritt 1.

Brückenregeln/bridge productions: Diese Regeln entstehen aus den Einstiegsregeln, in denen durch Kantenersetzung alle Nichtterminalkanten aus E^{max} durch entsprechende End- oder Ausstiegsregeln ersetzt werden. Sie repräsentieren eine Abfolge von Ableitungsschritten, in denen keine alten Produktionsregeln genutzt werden.

Faltregeln/non-final folding productions: Faltregeln entstehen indem man eine Kante in einer Mittelregel durch eine entsprechende Zwischenregel, in der wiederum eine Kante durch eine Mittelregel ersetzt wurde, austauscht. Daraufhin entfernt man alle unzulässigen externen Knoten mit anliegenden Nichtterminalkanten aus E^{max} . Die Zuweisung der externen Knoten wird entsprechend angepasst.

Endfaltregeln/final folding productions: Hier wird für eine Mittelregel $X_1 \rightarrow H$ als Regelbezeichner das Nichtterminal $\langle X_1, X_2, \phi \rangle$ gewählt, falls die rechte Regelseite eine Kante X_2 enthält. Wie für die Faltregeln werden nun auch hier unzulässige Knoten und Nichtterminalkanten entfernt. Eine weitere Endfaltregel wird zusätzlich analog zur vorigen erstellt, jedoch wird hier eine Mittelregel $X_2 \rightarrow H'$ direkt in H eingesetzt, falls H' eine Kante X_3 enthält und ein Nichtterminal $\langle X_1, X_3, \phi' \rangle$ vorhanden ist. Letzteres wird dann als Regelbezeichner eingesetzt.

5. In der so konstruierten Grammatik G' wird für jede Regel an jeder enthaltenen Nichtterminalkante jede in G' enthaltene, anwandbare Produktionsregel benutzt. Die resultierenden Regeln werden zu einer neuen Grammatik G'' hinzugefügt, die das Alphabet und das Axi-

om von G' übernimmt. G'' bildet die Eingabe für weitere Wiederholungen von Schritt 1-5, sofern sie nun nicht bereits in Greibach Normalform vorliegt.

Da für die Anwendung als Heapabstraktionsgrammatik auf der einen Seite die lokale Apex-Eigenschaft bereits ausreichend ist und auf der anderen Seite lediglich der Ausgangsknotengrad als beschränkt angenommen werden kann, benutzen wir eine angepasste Greibach Normalform, die zum einen genau die gestellten Anforderungen erfüllt und es zum anderen zulässt, die Ergebnisse aus [Eng92] auch auf HRGs mit unbeschränkten Eingangsgrad zu übertragen.

Um den Beweis des angepassten Theorems zur Apex-Eigenschaft zu vereinfachen, passen wir die Greibach Normalform zusätzlich so an, dass lokale Apex-Eigenschaft immer nur zu einem Nichtterminal an einem zugehörigen externen Knoten herstellbar sein muss. Dadurch erhalten wir direkt durch Konstruktion der Greibach Normalform für Nichtterminal X und externen Knoten $ext(i)$ die Teilgrammatik G_i^X , siehe Definition 3.2.6. Zur Vereinfachung definieren wir vorher die Menge der Knoten, die durch Ableitung ausgehend von allen Produktionsregeln eines Nichtterminals an einen festen externen Knoten „angedockt“ werden können.

Definition 4.1.2 (Erweiterte Map-Menge) Gegeben eine HRG G , ein Nichtterminal X sowie ein $i \in [1, rk(X)]$. Wir bezeichnen (A, j) mit $A \in N$ und $j \in [1, rk(A)]$ als Map-Tupel, falls ausgehend von $(X, H) \in P$ durch beliebig viele Ableitungsschritte $H \Rightarrow^* H'[H_A/e_A]$ mit $e_A \in E(ext(i))$, $lab(e_A) = A$ und $(A, H_A) \in P$, der externe Knoten $ext(j) \in E_{H_A}$ auf den externen Knoten $ext(i)$ gemappt wird. Die Menge aller Map-Tupel (A, j) bezeichnen wir als Map-Menge $map(X, i)$. Die erweiterte Map-Menge $extmap(X, i)$ enthält als Teilmenge der Map-Menge nur Map-Tupel, dessen zugehörige Ableitung potentiell ausgehende Kanten am externen Knoten $ext(i)$ induziert, und ist wie folgt definiert: $extmap(X, i) = \{(A, j) \in map(X, i) \mid \exists B \in N, e \in E(ext(j)) : lab(e) = B \wedge nod(e, k) = ext(j) \wedge terminal(B, k) \neq \emptyset\}$.

Definition 4.1.3 (Lokale Greibach Normalform) Gegeben eine HRG G , ein Nichtterminal X und ein $i \in [1, rk(X)]$. G ist für X an Knoten $ext(i)$ in lokaler Greibach Normalform, wenn es die folgenden zwei Eigenschaften besitzt:

- G ist isolationsfrei.
- keine Produktionsregel $X' \rightarrow H \in P$ mit $(X', j) \in extmap(X, i) \cup \{(X, i)\}$ besitzt am externen Knoten $ext(j) \in H$ Ausgangsgrad 1.

Auch das Theorem zur Apex-Eigenschaft aus [Eng92] muss noch für Heapabstraktionsgrammatiken mit unbeschränktem Eingangsgrad angepasst werden.

Theorem 4.1.1 *Gegeben eine Sprache $L(G)$ einer HRG G . Ist der Knotenausgangsgrad von L beschränkt, dann existiert eine HRG G mit folgenden Eigenschaften:*

Für jedes $X \in N$ mit $rk(X) = k$ und jedes $1 \leq i \leq k$ existiert eine Teilgrammatik G_i^X mit

- $\bigcup_{i=1}^k G_i^X = G^X$
- $L(G_i^X \cup \overline{G^X}, X^\bullet) = L(G, X^\bullet), \forall 1 \leq i \leq k$
- G_i^X lokal apex, $\forall 1 \leq i \leq k$

Beweisidee. Die Beweisidee hier ist sehr intuitiv. Für jedes Nichtterminal X und jeden zugehörigen externen Knoten $ext(i)$ wird eine Grammatik G_i^X in lokaler Greibach Normalform erstellt. Die Gesamtgrammatik ergibt sich dann aus der Vereinigung der Teilgrammatiken. Es bleibt dann noch zu zeigen, dass der betrachtete externe Knoten einfach in erweiterte, lokale Apex-Eigenschaft zu bringen ist. Die Idee ist nun durch eine Abfolge von Konkretisierungsschritten jede Regel $X \rightarrow H \in G$ an Knoten $ext(i)$ in eine Regel mit erweiterter, lokaler Apex-Eigenschaft umzuwandeln. Die lokale Greibach Normalform garantiert, dass dies in endlich vielen Schritten umsetzbar ist. Es muss natürlich gewährleistet sein, dass jede beliebige HRG in diese Normalform zu überführen ist. Weiterhin ist noch zu klären, wie die erweiterte Map-Menge korrekt berechnet wird.

Wir benötigen noch folgendes Theorem zur lokalen Greibach Normalform.

Theorem 4.1.2 *Zu jeder HRG kann eine äquivalente HRG in lokaler Greibach Normalform konstruiert werden.*

Der ausführliche Beweis dieser Aussage wird an dieser Stelle auf später verschoben. Für den Beweis von Theorem 4.1.1 setzen wir vorerst die Gültigkeit der obigen Aussage voraus. Nun muss weiter formalisiert werden, wie ein Konkretisierungsschritt zum Erstellen einer Teilgrammatik mit lokaler Apex-Eigenschaft aussieht. Mit Hinblick auf bessere Verwendbarkeit bei dem zu führenden Beweis definieren wir dazu das Resultat eines solchen Schrittes, welches wiederum eine Grammatik darstellt.

Definition 4.1.4 (Ersetzungsgrammatik) *Gegeben eine HRG $G = (N, T, P, S)$ in Greibach Normalform, ein Nichtterminal $X \in N$ und einen externen Knoten $e = ext(i)$ mit $1 \leq i \leq rk(X)$. Die Ersetzungsgrammatik $repl(G, X, i)$ übernimmt Nichtterminale N , Terminale T und das Axiom S von G . Außerdem enthält $repl(G, X, i)$ für jede Regel $X \rightarrow H \in P$ folgende Produktionsregeln:*

Fall (a) am externen Knoten e sind Nichtterminalkanten e_1, \dots, e_n vorhanden, die nicht ausschließlich durch Reduktions-Tentakel mit e verbunden sind:

Sei $\text{lab}_H(e_1) \rightarrow H_1, \dots, \text{lab}_H(e_n) \rightarrow H_n \in P$

$$X \rightarrow H[H_1/e_1, \dots, H_n/e_n] \in \text{repl}(G, X, i)$$

Fall (b) am externen Knoten e liegen nur Terminalkanten und Reduktions-Tentakel an:

$$X \rightarrow H \in \text{repl}(G, X, i)$$

Die hier definierte Ersetzungsgrammatik besitzt einige für den Beweis von Theorem 4.1.1 wichtige Eigenschaften, die im nächsten Lemma aufgelistet werden.

Lemma 4.1.1 Gegeben eine isolationsfreie HRG G . Dann gilt für $\text{repl}(G, X, i)$:

1. $L(\text{repl}(G, X, i)) = L(G^X)$.
2. $\text{repl}(G, X, i)$ ist isolationsfrei.
3. Wenn der Grad des externen Nichtterminalknotens $v = \text{ext}(i) \in H$ für eine beliebige Regel (X, H) grösser oder gleich k ist (mit $k \geq 2$), dann ist der Grad von v in $\text{repl}(G, X, i) \geq k+1$.

Auch für diesen Beweis wird auf später verwiesen. Für diesen Moment nehmen wir an die Eigenschaften aus Lemma 4.1.1 gelten. Die bis hierher erworbenen Kenntnisse sind als Grundlage für den Beweis von Theorem 4.1.1 ausreichend.

Beweis. Gegeben ist eine HRG $G = (N, T, P, Z)$, dessen Sprache $L(G)$ beschränkten Ausgangsknotengrad hat. Dann lässt sich G für jedes $X \in N$ und $i \in \text{rk}(X)$ in eine äquivalente Grammatik $G_{i,X} = (N', T, P', Z)$ in lokaler Greibach Normalform umformen. Sei der in $L(G) = L(G_{i,X})$ maximal vorkommende Ausgangsknotengrad $b \in \mathbb{N}$. Nun erstellt man sukzessive $b - 1$ -mal für X und den betrachteten externen Knoten $v = \text{ext}(i)$ die Ersetzungsgrammatik $G'_{i,X} = \text{repl}^{b-1}(G_{i,X}^X, X, i)$. Solange v die lokale Apex-Eigenschaft verletzt, d.h. mindestens eine nicht ausschließlich über Reduktionstentakel anliegende Nichtterminalkante besitzt, wird dabei in jedem Schritt der Ausgangsgrad erhöht. Daraus folgt direkt, dass v spätestens nach $b - 1$ Schritten lokale Apex-Eigenschaft besitzt, denn dann beträgt der Ausgangsknotengrad an v mindestens $k + (b - 1)$, wobei $k \geq 2$ wegen lokaler Greibach Normalform. Für einen Ausgangsknotengrad von $\geq b+1$ lässt sich aber direkt folgern, dass an v nun ausschließlich Terminalkanten anliegen müssen,

da nach Voraussetzung der Ausgangsknotengrad durch b beschränkt ist. Folglich besitzt die Teilgrammatik $G'_{i,X}$ am externen Knoten $ext(i)$ aus X erweiterte, lokale Apex-Eigenschaft. Die Gesamtgrammatik G' entsteht dann durch Vereinigung der Nichtterminale, Terminale und Produktionsregeln der Teilgrammatiken. Da die Sprache jeder Teilgrammatik $L(G'_{i,X}) = L(G_{i,X})$ gleich der Sprache der Ausgangsgrammatik $L(G^X)$ ist, sind die Eigenschaften $\bigcup_{i=1}^k G'_{i,X} = G^X$ und $L(G_{i,X} \cup \overline{G^X}, X^\bullet) = L(G, X^\bullet) \forall 1 \leq i \leq k$ damit trivialerweise erfüllt und es gilt $L(G') = L(G)$. \square

Im nächsten Abschnitt beschäftigen wir uns mit dem Beweis von Lemma 4.1.1.

Beweis. 1. Wir zerlegen die Aussage in Hin- und Rückrichtung und beweisen diese getrennt.

$\mathbf{L}(\mathbf{repl}(\mathbf{G}, \mathbf{X}, \mathbf{i})) \subseteq \mathbf{L}(\mathbf{G}^{\mathbf{X}})$: Unter Benutzung der grundlegenden Eigenschaft der Assoziativität der Kantenersetzung in HRGs ist diese Inklusion einfach zu zeigen. Betrachten wir eine beliebige Produktionsregel $X \rightarrow H[H_1/e_1, \dots, H_n/e_n] \in \mathbf{repl}(G, X, i)$ und einen beliebigen Hypergraphen K , der eine Nichtterminalkante enthält, die mit X beschriftet ist. Da mit dem Assoziativgesetz gilt, dass $K[H/e][H[H_1/e_1, \dots, H_n/e_n]] = K[H[H[H_1/e_1, \dots, H_n/e_n]]]$, kann die obige Produktionsregel durch eine Abfolge von Konkretisierungsschritten in G^X simuliert werden: $X \rightarrow H, \mathit{lab}_H(e_1) \rightarrow H_1, \dots, \mathit{lab}_H(e_n) \rightarrow H_n$. Da dies für jede beliebige Produktionsregeln in $\mathbf{repl}(G, X, i)$ durchführbar ist, folgt $L(\mathbf{repl}(G, X, i)) \subseteq L(G^X)$.

$\mathbf{L}(\mathbf{G}^{\mathbf{X}}) \subseteq \mathbf{L}(\mathbf{repl}(\mathbf{G}, \mathbf{X}, \mathbf{i}))$: Für den Beweis dieser Inklusion benötigt man zusätzlich zu der Assoziativität der Kantenersetzung auch das in 2.3.2 eingeführte Dekompositionslemma. Betrachten wir eine Reihe von Konkretisierungsschritten $X \Rightarrow^* F$ in G^X , oder konkreter $X \Rightarrow H \Rightarrow^* F$ für eine Produktionsregel $X \rightarrow H$. Nach Dekompositionslemma gilt: $F = H[F_1/e_1, \dots, F_n/e_n]$, wobei e_1, \dots, e_n die mit X beschrifteten Nichtterminalkanten in G^X sind mit $X \Rightarrow^* F_j$. Es existiert folglich wiederum ein Hypergraph H_j , der von X durch einen Konkretisierungsschritt abgeleitet in F_j überführt werden kann für beliebige Produktionen $X \rightarrow H_j$ in G^X . Durch weiteres Anwenden des Dekompositionslemmas ergibt sich dann: $F_j = H_j[F_{j1}/e_{j1}, \dots, F_{jm}/e_{jm}]$, wenn die e_{j1}, \dots, e_{jm} diesmal die mit X beschrifteten Nichtterminalkanten von H_j darstellen. Es gilt wieder $X \Rightarrow^* F_{jk}$ für $k \in [1, m]$. Durch Induktion folgt dann direkt $X \Rightarrow^* F_{jk}$ in $\mathbf{repl}(G, X, i)$. Nach dem Assoziativgesetz ist folglich äquivalent:

$$F = (H[H_1/e_1, \dots, H_n/e_n])[F_{11}/e_{11}, \dots, F_{1m_1}/e_{1m_1}, \dots, F_{n1}/e_{n1}, \dots, F_{nm_n}/e_{nm_n}]$$

Es folgt durch „Rückwärtsanwendung“ des Dekompositionslemmas, dass $H[H_1/e_1, \dots, H_n/e_n] \Rightarrow^* F$ in $\mathbf{repl}(G, X, i)$. Da nach Definition die Produktionsregel $X \rightarrow H[H_1/e_1, \dots, H_n/e_n]$ auch in

$repl(G, X, i)$ vorhanden ist, lässt sich $X \Rightarrow^* F$ auch in $repl(G, X, i)$ ableiten.

2. Die Isolationsfreiheit folgt trivialerweise aus Lemma 2.3.3.

3. Sei $v = ext(i)$ externer Nichtterminalknoten aus $H[H_1/e_1, \dots, H_n/e_n] \in repl(G, X, i)$. D.h. $v \in H$ besitzt mindestens eine anliegende Nichtterminalkante, die nicht ausschließlich über Reduktionstentakel mit v verbunden ist. Da v außerdem nach der Erstellung der Ersetzungsgrammatik immer noch Nichtterminalknoten ist, muss mindestens einer der Hypergraphen H_i , $1 \leq i \leq n$ wiederum eine an dem mit v korrespondierenden externen Knoten v' nicht ausschließlich durch Reduktionstentakel verbundene Nichtterminalkante besitzen, dies seien H'_1, \dots, H'_m , mit $m \geq 1$. Angenommen $v \in H$ hat Ausgangsknotengrad d und $v' \in H'_j$ Ausgangsknotengrad d_j , wobei nach Voraussetzung $d, d_j \geq k \geq 2$. D.h. für jedes $H' \in \{H'_1, \dots, H'_m\}$ wächst der Ausgangsknotengrad um mindestens $k-1 \geq 1$. Da per Definition nur diejenigen Nichtterminalkanten e_i ersetzt werden, die nicht mit v durch Reduktionstentakel verbunden sind, folglich zum Ausgangsknotengrad beitragen, kann $v \in H[H_1/e_1, \dots, H_n/e_n]$ niemals einen kleineren Ausgangsknotengrad als $v \in H$ besitzen. Daraus folgt direkt, dass der Ausgangsknotengrad d' von $v \in H[H_1/e_1, \dots, H_n/e_n]$ größer ist als $d + m * 1$. Da $m \geq 1$ und $d \geq k$ folgt weiter $d' \geq k + 1$. \square

Es lässt sich leicht erkennen, dass es für eine Heapabstraktionsgrammatik ausreichend ist, wenn sie in Greibach Normalform vorliegt, anstatt die stärkere Anforderung der lokalen Apex-Eigenschaft zu erfüllen. Die Ersetzungsgrammatik $repl$ entsteht durch nichts anderes als mehrere hintereinander ausgeführte Konkretisierungsschritte. Durch den beschränkten Ausgangsknotengrad ist außerdem sichergestellt, dass in endlich vielen Konkretisierungsschritten erweiterte, lokale Apex-Eigenschaft an einem beliebigen externen Knoten einer beliebigen Regelmenge garantiert werden kann. Folglich ist es für ein Framework, das Konkretisierungsschritte durchführen und erkennen kann, wann lokale Apex-Eigenschaft erreicht ist, ausreichend mit einer Grammatik in Greibach Normalform anstatt mit erweiterter, lokaler Apex-Eigenschaft zu arbeiten.

Nachdem nun gezeigt wurde, dass die Greibach Normalform einer HRG zur Verifikation von auf heapbasierten Datenstrukturen operierenden Programmen genutzt werden kann, wird ein Verfahren benötigt, welches eine beliebige HRG in lokale Greibach Normalform umzuformen vermag. Dieser Konstruktion der hier betrachteten NF liegt das Vorgehen des Beweises von Theorem 4.1.2 zugrunde, der in ausführlicher Form in [EHL94] zu finden ist. Da die Konstruktion der lokalen Greibach Normalform ein recht komplexer Vorgang ist, wird ihr ein eigener Unterabschnitt gewidmet.

4.1.1 Konstruktion der lokalen Greibach Normalform

Da der Beweis zu Theorem 4.1.2 die Konstruktion der lokalen Greibach Normalform beinhaltet, wird sich dieser Abschnitt im Detail mit ihm befassen. Der grundlegende Unterschied zwischen der Greibach Normalform aus [Eng92] und der hier benötigten Normalform ist die Tatsache, dass bei ersterer gefordert wird, dass der Grad an jedem Nichtterminalknoten ≥ 1 sein muss, wohingegen für zweitere der Eingangsgrad unentscheidend ist und sie einen Ausgangsgrad ≥ 1 fordert. Eine erste Idee wäre die Konstruktion der Greibach Normalform aus [Eng92] bis auf diese kleine Änderung einfach zu übernehmen. Warum das nicht funktionieren kann, wird an einem Beispiel schnell klar.

Beispiel 4.1.2 *Betrachten wir das Minimalbeispiel, das beide Arten von Problemen für die Greibach Normalform Konstruktion enthält: einen Nichtterminalknoten mit Ausgangsgrad 1 sowie einen Knoten mit unendlichem Eingangsgrad. Eine einfache Datenstruktur, die diese Anforderungen erfüllt, ist eine einfach verkettete Liste, in der jedes Element zusätzlich einen Zeiger zum Anfang der Liste besitzt, siehe Abbildung 4.4. Die Grammatik, die alle Listen dieser Art erzeugt, ist in Abbildung 4.5 zu finden.*

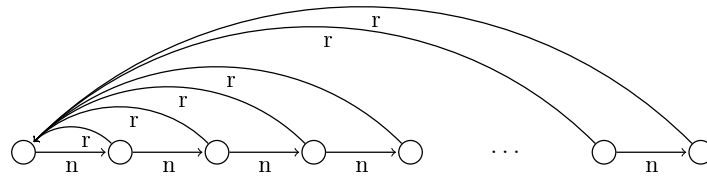


Abbildung 4.4: Beispiel: Einfach verkettete Liste mit Rückwärtszeigern

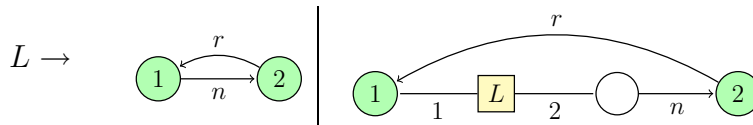


Abbildung 4.5: Grammatik: Einfach verkettete Liste mit Rückwärtszeigern

Nach [Eng92] liegt diese Grammatik bereits in Greibach Normalform vor, da jeder Knoten Grad 2 hat. Es lässt sich jedoch durch einfache Kantenersetzung keine Grammatik mit Apex-Eigenschaft konstruieren. Betrachtet man hingegen nur den Ausgangsgrad, ist der externe Knoten 1 der zweiten L-Regel Nichtterminalknoten und besitzt (Ausgangs-)Grad 1. Setzen wir also hier mit der Greibach Normalform Konstruktion an. Zuerst wird eine zur Eingabegrammatik äquivalente

Grammatik erstellt, die zusätzliche Informationen enthält (siehe Grammatik G' in [Eng92]). Das Resultat ist in Abbildung 4.6 zu sehen. Der Bezeichner L_1 steht hier für $L, \{1\}$, folgt man der Namensgebung aus [Eng92]. Er ist dabei so gewählt, dass er zusätzlich zum Nichtterminal Angaben über die externen Knoten enthält, in die, die Greibach Normalform verletzten, Knoten überführt werden.

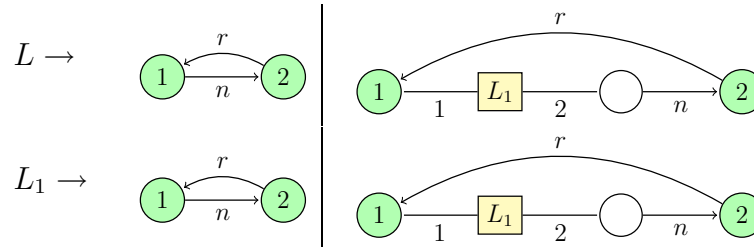


Abbildung 4.6: Erweiterte Grammatik: Einfach verkettete Liste mit Rückwärtszeigern

Aus dieser Grammatik wird dann die Grammatik G_1 in Greibach Normalform erstellt. Dazu wird ein neues Nichtterminal $D_1 = \langle L_1, \{1\}, L_1, \{1\}, \omega \rangle$ vom Rang 2 eingeführt, das das Nichtterminal L_1 an den problematischen Stellen ersetzen soll. Die resultierenden Regeln sind in Abbildung 4.7 zu finden.

Hier ist offensichtlich, wo das Problem liegt. Zur Erstellung der Faltregeln entfernt man im Prinzip den fehlerhaften Knoten, der Grad 1 besitzt und die entsprechende anliegende Nichtterminalkante. Für unsere lokale Greibach Normalform besitzt dieser Knoten nur Ausgangsgrad 1, aber zusätzlich noch eingehende Kanten. Diese Kanten darf man natürlich nicht entfernen. Um allerdings den Ausgangsgrad von 1 zu verhindern, muss die Nichtterminalkante zusammen mit dem Nichtterminalknoten vom (Ausgangs-)Grad 1 entfernt werden. Das führt dazu, dass etwaige eingehende Kanten an diesem Nichtterminalknoten nun keinen Zielknoten mehr besitzen, sozusagen „in der Luft hängen“.

Um das in Beispiel 4.1.2 illustrierte Problem beim Übertragen der Greibach Normalform Konstruktion aus [Eng92] zu umgehen, wird an dieser Stelle das Aufsplitten von externen Knoten eingeführt. Dabei ist das Vorgehen sehr intuitiv, denn es „splittet“ einen vorher festgelegten externen Knoten in zwei unterschiedliche Knoten auf, wobei einer von ihnen alle ausgehenden und der andere alle eingehenden Kanten vom ursprünglichen Knoten erhält. Um die Formalisierung des Aufsplittens eines Knotens zu vereinfachen, werden zuerst die Mengen der an einem Knoten anliegenden ein- und ausgehenden Kanten definiert.

Definition 4.1.5 (Kantenmengen eines Knotens v) Gegeben ein Hypergraphen H , ein Knoten $v \in V_H$ und seine zugehörige Kantenmenge $E(v)$. Wir partitionieren $E(v)$ in die Menge der

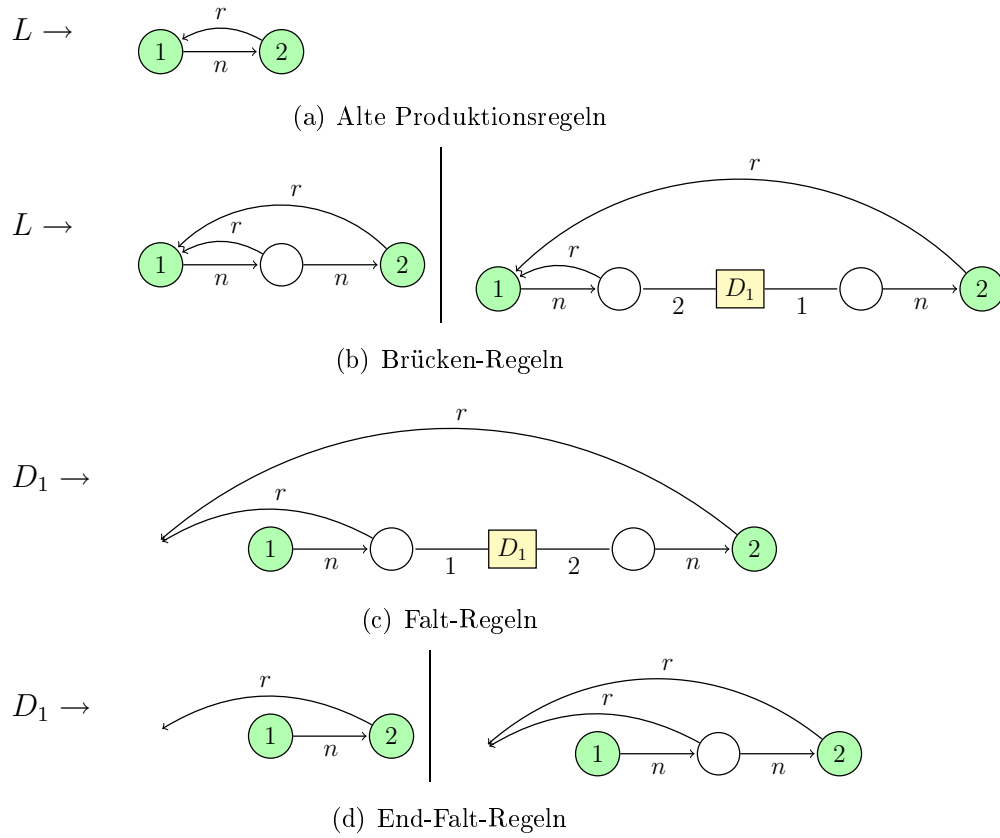


Abbildung 4.7: Grammatik in Greibach NF: Einfach verkettete Liste mit Rückwärtszeigern

eingehenden und ausgehenden Kanten $E(v) = E^{in}(v) \cup E^{out}(v)$ und diese wiederum in die Menge der eingehenden Nichtterminal- und Terminalkanten $E^{in}(v) = E^{Tin}(v) \cup E^{Nin}(v)$ sowie ausgehenden Nichtterminal- und Terminalkanten $E^{out}(v) = E^{Tout}(v) \cup E^{Nout}(v)$ mit:

- $E^{Tin}(v) = \{e \in E_H \mid lab(e) \in T_H \wedge nod(e, 2) = v\}$
- $E^{Nin}(v) = \{e \in E_H \mid \exists j \in N : lab(e) \in N_H \wedge nod(e, j) = v \wedge terminal(lab(e), j) = \emptyset\}$
- $E^{Tout}(v) = \{e \in E_H \mid lab(e) \in T_H \wedge nod(e, 1) = v\}$
- $E^{Nout}(v) = \{e \in E_H \mid \exists j \in N : lab(e) \in N_H \wedge nod(e, j) = v \wedge terminal(lab(e), j) \neq \emptyset\}$

Definition 4.1.6 (Split-Grammatik) Gegeben eine HRG G , ein Nichtterminal X , sowie ein externer Knoten $ext(i) = v$, mit $i \in [1, rk(X)]$. Dann entsteht die Split-Grammatik $G_{split}(X, i)$ aus G wie folgt:

1. Terminale T , Nichtterminale N und das Axiom S wird aus G übernommen.
2. Jede Regel $(A, H) \in P$ mit $A \neq X$ wird aus G übernommen.
3. Die Menge der Nichtterminale $N \in G_{split}(X, i)$ wird um das Nichtterminal X' mit $rk(X') = rk(X) + 1$ erweitert und eine Produktionsregel (X, M) zu $G_{split}(X, i)$ hinzugefügt, wobei $M = (V_M, E_M, lab_M, att_M, ext_M)$ folgende Gestalt hat, siehe auch Abbildung 4.8:

- $V_M = \{v_1, \dots, v_{rk(X)}\}$
- $E_M = \{e_{X'}\}$
- $lab_M(e_{X'}) = X'$
- $att_M(e_{X'}) = v_1, \dots, v_{rk(X)}, v_i$
- $ext_M = v_1, \dots, v_{rk(X)}$

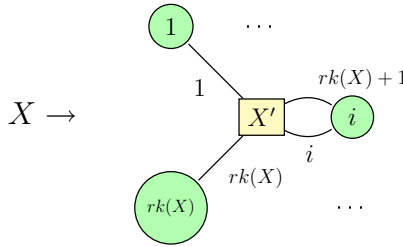


Abbildung 4.8: Schematische Darstellung: Split-Regel

Der so entstandene Hypergraph M entspricht einem modifizierten X' -Henkel $X'^{\bullet i}$, in dem zwei Tentakel an demselben externen Knoten $ext(i)$ anliegen. Wir bezeichnen die so entstandene Regel als Split-Regel.

4. Für jede Regel $(X, H) \in P$ wird eine neue Regel (X', H') in $G_{split}(X, i)$ eingefügt, die anstatt v die Knoten v_{in} und v_{out} enthält. An v_{in} liegen dabei alle ursprünglich an v eingehenden, an v_{out} alle ursprünglich ausgehenden Kanten an. Formal entsteht $H' = (V', E', lab', att', ext')$ aus $H = (V, E, lab, att, ext)$ wie folgt:

- $V' = V \setminus \{v\} \cup \{v_{in}, v_{out}\}$
- $E' = E_H$
- $lab_{H'}(e) = \begin{cases} X' & \text{falls } lab_H(e) = X \\ lab_H(e) & \text{sonst} \end{cases}$
- Sei $att_H(e) = v_1, \dots, v_n$ mit $1 \leq i \leq n$. Dann gilt:

$att_{H'}(e) = \left\{ \begin{array}{l} v'_1, \dots, v'_n \\ v'_1, \dots, v'_n, v_{in} \\ att_H(e) \end{array} \right.$	$v'_i = v_i$ für alle $v_i \neq v$, $v'_i = v_{out}$ für $v_i = v$ $\wedge e \in E_H^{out}(v)$, $v'_i = v_{in}$ für $v_i = v$ $\wedge e \in E_H^{in}(v)$	falls $lab_H(e) \neq X$
	$v'_i = v_i$ für alle $v_i \neq v$, $v'_i = v_{out}$ für $v_i = v$ $\wedge e \in E_H^{out}(v)$, $v'_i = v_{in}$ für $v_i = v$ $\wedge e \in E_H^{in}(v)$	falls $lab_H(e) = X$
	sonst	
- $ext_{H'}(j) = \begin{cases} ext_H(j) & \text{falls } j \leq rk(X) \wedge j \neq i \\ v_{out} & \text{falls } ext_H(j) = v \\ v_{in} & \text{falls } i = rk(X) + 1 \end{cases}$

Es muss noch gezeigt werden, dass sich die erkannte Sprache durch die Erstellung der Split-Grammatik nicht ändert. Der nächste Abschnitt widmet sich dem Beweis des nachfolgenden Lemmas.

Lemma 4.1.2 (Sprache einer Split-Grammatik) Sei $G_{split}(X, i)$ eine Split-Grammatik und die HRG G die entsprechende Grammatik aus der $G_{split}(X, i)$ entstanden ist, so sind ihre Sprachen gleich, d.h. $L(G) = L(G_{split}(X, i))$, und damit G und $G_{split}(X, i)$ äquivalent.

Beweis. $L(G) \subseteq L(G_{split}(X, i))$: Sei $S \Rightarrow^* F$ eine beliebige Ableitung mit S Axiom von G und $F \in L(G)$. Wähle beliebigen Knoten v aus F . Wir ordnen die Ableitungskette so um, dass

$S \Rightarrow H_1 \Rightarrow \dots \Rightarrow H_j \Rightarrow \dots \Rightarrow H_n$ ab H_j alle Nichtterminalkanten e ersetzt werden für die gilt: $nod(e, k) = v \rightarrow (e, k)$ Reduktionstentakel und $e \in E_S \cup E_{H_1} \cup \dots \cup E_{H_n}$ mit $k \in [1, rk(\text{lab}(e))]$. (Dekompositions-Lemma)

Wir zeigen, dass eine Ableitung $S \Rightarrow^{\leq j} H_j$ in $G_{split}(X, i)$ existiert. Seien $(N_1, A_1), \dots, (N_{j-1}, A_{j-1})$ die in G angewandten Produktionsregeln für die Ableitung $S \Rightarrow H_j$.

Fall 1: $\forall(N_k, A_k) : N_k \neq X, k \in [1, j - 1]$.

Dann existieren die gleichen Produktionsregeln (N_k, A_k) auch in $G_{split}(X, i)$ und $S \Rightarrow^{j-1} H_j$ lässt sich analog herleiten.

Fall 2: $\exists^{>1}(N_k, A_k) : N_k = X, k \in [1, j - 1]$.

Dieser Fall kann nicht auftreten, da G typisiert und wiederholungsfrei.

Fall 3: $\exists!(N_k, A_k) : N_k = X, k \in [1, j - 1]$.

Wir ordnen wiederum die Ableitungsschritte so um, dass $S \Rightarrow^m H_{help} \Rightarrow^{i-m} H_j$ bis H_{help} vollständig abgeleitet wurde, ausschließlich Produktionsregeln (N, A) mit $N \neq X$ verwandt werden. (Dekompositions-Lemma) Die Ableitungsschritte bis H_{help} können wieder analog zu G durchgeführt werden, da die notwendigen Produktionsregeln auch in $G_{split}(X, i)$ vorhanden sind.

Für die übrigen $i - m$ Schritte zeigen wir durch Induktion über die Länge der Ableitung, dass eine Ableitung $H_{help} \Rightarrow^{i-m} H_j$ in $G_{split}(X, i)$ existiert.

Ist die Ableitungskette in G lediglich einen Schritt lang und Produktionsregel (X, H) wurde angewandt, so lässt sie sich in $G_{split}(X, i)$ wie folgt simulieren: Wir wenden erst die Split-Regel $X'^{\bullet i}$ gefolgt von der (X, H) entsprechenden Regel (X', H') auf die einzige nicht durch Reduktionstentakeln mit v verbundene Nichtterminalkante $e_{X'}$ an. Sei e_X die durch (X, H) zu ersetzende Kante mit $nod(e_X, j) = v$ für ein $j \in [1, rk(X)]$. Da im Anschluß um v ausschließlich Terminalkanten anliegen (abgesehen von Reduktionstentakeln), muss dies auch für den externen Knoten $ext(j)$ in H gelten und nach Konstruktion der Split-Grammatik auch für die einzige Kante $e_{X'}$ mit $lab(e_{X'}) = X'$ mit $nod(e_{X'}, j)$ und den externen Knoten $ext(j) \in H'$. Es folgt weiter, dass $nod(e_X, j) = v \rightarrow nod(e_{X'}, j) = v \wedge nod(e_{X'}, rk(X) + 1) = v$ und da sich H und H' nur an v durch die Kanten e_X und $e_{X'}$ unterscheiden damit auch die Äquivalenz der resultierenden Hypergraphen.

Eine n Schritte lange Ableitungskette in G wird in $G_{split}(X, i)$ folgendermaßen simuliert: Wiederum beginnend mit der einzigen anwendbaren Regel – der Split-Regel $X'^{\bullet i}$ – wird zu der in G verwendeten Produktionsregel (X, H) in $G_{split}(X, i)$ die entsprechende Regel (X', H') (siehe Definition 4.1.6) angewandt. Seien $H_1 \Rightarrow \dots \Rightarrow H_n$ in G und $H_1 \Rightarrow H_{split} \Rightarrow H'_2 \Rightarrow \dots \Rightarrow H'_n$ in $G_{split}(X, i)$ die resultierenden Ableitungsketten. Es gilt für jedes j aus $[1, n]$, dass H_j zu H'_j äquivalent ist, bis auf die an v nicht durch ein Reduktionstentakel anliegende Nichtterminalkante e_X mit $lab(e_X) = X$ (alle Produktionsregeln

mit anderen Nichtterminalbezeichnern wurden wegen der Umsortierung bereits angewandt oder sind Reduktionstentakel). H'_j enthält anstelle von e_X eine Nichtterminalkante $e_{X'}$ mit $lab(e_{X'}) = X'$. X und X' unterscheiden sich dabei lediglich im Rang, d.h. $rk(X') = rk(X) + 1$ mit $nod(e_X, k) = nod(e_{X'}, k)$ für $k \in [1, rk(X)]$ und $nod(e_{X'}, rk(X) + 1) = v$. Durch die angenommene Umsortierung dürfen nach dem n -ten Ableitungsschritt ausschließlich Terminalkanten (und Reduktionstentakel) an v anliegen. Folglich wird in diesem Schritt eine Kante e_X mit $v = nod(e_X, j)$ durch eine Regel (X, H) ersetzt, die am externen Knoten $ext(j)$ ausschließlich Terminale (und Reduktionstentakel) besitzt. Sowohl Regel als auch die Nichtterminalkante existiert wiederum analog in $G_{split}(X, i)$ als (X', H') bzw. $e_{X'}$, lediglich mit gesplittetem Knoten v . Da durch die Konstruktion der Split-Grammatik gelten muss, dass $nod(e_X, j) = v \rightarrow nod(e_{X'}, j) = v \wedge nod(e_{X'}, rk(X) + 1) = v$, folgt damit die Äquivalenz von H_n und H'_n . Es existiert also eine Ableitung $H_1 \Rightarrow H_{split} \Rightarrow^n H_n$ in $G_{split}(X, i)$.

Dieses Vorgehen lässt sich nun für beliebige Knoten und die verbliebenen Ableitungsschritte $H_j \Rightarrow H_n$ wiederholen bis ausschließlich Kantenersetzung an Reduktionstentakeln (N, k) übrig bleibt. Da gilt, dass $terminal(N)(k) = \emptyset$, folgt direkt, dass $N \neq X \vee k \neq i$. Die benötigten Produktionsregeln für die verbliebenen Kantenersetzungen existieren folglich analog $G_{split}(X, i)$. Es ergibt sich damit: $S \Rightarrow^* F$ lässt sich auch in $G_{split}(X, i)$ herleiten.

$L(G_{split}(X, i)) \subseteq L(G)$: analog. □

Nachdem Beispiel 4.1.2 ein großes Problem bei der Greibach Normalform Konstruktion aus [Eng92] für die Anforderungen der Heapabstraktionsgrammatiken aufzeigte, wurde zur Lösung das Konzept der Split-Grammatik eingeführt. Diese Grammatik entsteht, indem man einen vorgegebenen externen Knoten v eines Nichtterminals in zwei Knoten aufsplittet, die jeweils nur eingehende bzw. ausgehende Kanten von v erhalten. Die erkannte Sprache wird bei diesem Vorgang nicht verändert. Für die Herstellung der erweiterten, lokalen Greibach Normalform ist es nun wichtig zu wissen, welche Knoten genau problematisch sind und gesplittet werden müssen. Bei näherer Betrachtung ist klar, dass genau für das Nichtterminal und der Knoten v , an dem die Greibach Normalform hergestellt werden soll, gesplittet werden muss. Desweiteren muss auch für jedes andere Nichtterminal und externen Knoten, der durch Ableitung an die Stelle von v treten kann, gesplittet werden. Um diese Bestimmung von Split-Knoten zu systematisieren, wird das Konzept und die Konstruktion des Nichtterminal-Ableitungsgraphen eingeführt. Dieser beinhaltet alle Paare von Nichtterminalen und externen Knoten (Map-Tupel), die bei beliebigen Ableitungsschritten ausgehend von einem Paar von Nichtterminal und externem Knoten an diesem entstehen können, ähnlich wie beim Ableitungsbaum.

Definition 4.1.7 (Nichtterminal-Ableitungsgraph) Gegeben eine HRG $G = (N, T, P, S)$. Der zugehörige Nichtterminal-Ableitungsgraph ist ein Graph, dessen Knoten mit Map-Tupeln (A, i) beschriftet sind. Dabei existiert eine Kante von (A_i, i) zu (A_j, j) , falls ein Regel $(A_i, H) \in P$ existiert, im Hypergraphen H am externen Knoten $ext(i)$ das j -te Tentakel der Nichtterminalkante A_j anliegt und $terminal(A_j)(j) \neq \emptyset$.

Zum besseren Verständnis des Nichtterminal-Ableitungsgraphen wird mit Algorithmus 4.1 ein Verfahren zur Konstruktion von diesem eingeführt. Die Terminierung des Algorithmus und damit der Konstruktion des NT-Ableitungsgraphen ist direkt ersichtlich, da in jedem Durchlauf ein Map-Tupel abgearbeitet wird. Dabei existieren nur endliche viele Kombinationsmöglichkeiten von Nichtterminal und Rangindex – diese bilden die Map-Tupel –, da nur endlich viele Nichtterminale in einer Grammatik vorhanden sind und jede Regel beschränkten Knotengrad besitzt.

Algorithm 4.1 Konstruktion: NT-Ableitungsgraph

Require: HRG G mit Axiom S

Ensure: NT-Ableitungsgraph $G' = (V, E)$

$E := \emptyset$

$V := \{(S, i)\} \forall 1 \leq i \leq rk(S)$

$q.Enqueue(V)$

while $!q.isEmpty$ **do**

$(X, i) := q.Dequeue$

$v := ext(i)$ of nonterminal N

$visited := visited \cup \{(X, i)\}$

for all productions $(X, H) \in G$ and edges $e \in E_H(v)$ **do**

if $lab(e) \in N$ **then**

$V'_H := \{(lab(e), j_1), \dots, (lab(e), j_n)\} \forall j: 1 \leq j \leq rk(e) \wedge terminal(lab(e), j) \neq \emptyset$

$V := V \cup V'_H$

$E := E \cup \{((X, i), v'_1), \dots, ((X, i), v'_n)\}$ with $E'_H(v) = \{v'_1, \dots, v'_n\}$

$q.Enqueue(V'_H \setminus visited)$

end if

end for

end while

Die Absicht hinter der Konstruktion dieses Ableitungsgraphen ist, wie oben erwähnt, herauszufinden, für welche der externen Knoten und welcher Nichtterminale die zugehörige Split-Grammatik erstellt werden muss. Für die Konstruktion der erweiterten, lokalen Greibach Normalform für ein Nichtterminal X und einen externen Knoten $ext(i)$, $1 \leq i \leq rk(X)$ sind dies genau diejenigen Paare (A, j) die ausgehend von (X, i) einschließlich im NT-Ableitungsgraphen erreichbar sind:

Alle externen Knoten in rechten Regelseiten, in die $ext(i)$ durch Ableitungsschritte überführt werden kann.

Beispiel 4.1.3 Betrachten wir wieder die Grammatik aus Abbildung 4.5. Wie in Beispiel 4.1.2 gesehen, scheiterte hier die erweiterte, lokale Greibach Normalform Konstruktion. Wir führen die gesamte Konstruktion nochmals unter Verwendung von Split-Grammatiken durch. Dafür wird als erster Schritt der NT-Ableitungsgraph der Grammatik bestimmt, der in Abbildung 4.9 zu finden ist. Soll nun wie in Beispiel 4.1.2 auch die Grammatik G_i^L für Nichtterminal L und externem Knoten $ext(i)$, $1 \leq i \leq rk(L)$ in erweiterte, lokale Greibach Normalform gebracht werden, geht aus dem NT-Ableitungsgraphen hervor, dass lediglich für alle Produktionsregeln (L, H) der externe Knoten 1 aufgesplittet werden muss. Das führt zu der Grammatik aus Abbildung 4.10.

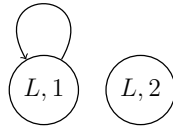


Abbildung 4.9: NT-Ableitungsgraph: Einfach verkettete Liste mit Rückwärtszeigern II

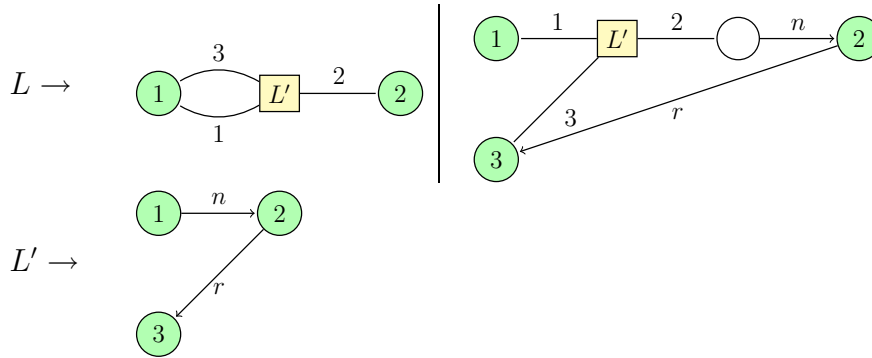


Abbildung 4.10: Split-Grammatik: Einfach verkettete Liste mit Rückwärtszeigern II

Mit dieser Split-Grammatik lässt sich nun die Greibach Normalform Konstruktion aus [Eng92] übernehmen, was hier am Beispiel demonstriert werden soll. Der erste Schritt - die Erstellung einer äquivalenten Grammatik mit zusätzlichen Informationen - ist wenig komplex, sein Resultat ist in Abbildung 4.11 zu finden. Auch hier steht das neue Nichtterminal L'_1 wieder für $L', \{1\}$ der Greibach Konstruktion.

Als letzter Schritt erfolgt nun das Falten der problematischen Regeln, um diese in lokale Greibach Normalform zu bringen. Dafür wird ein neues Nichtterminal $D_1 := \langle L'_1, L'_1, \phi \rangle$ eingeführt,

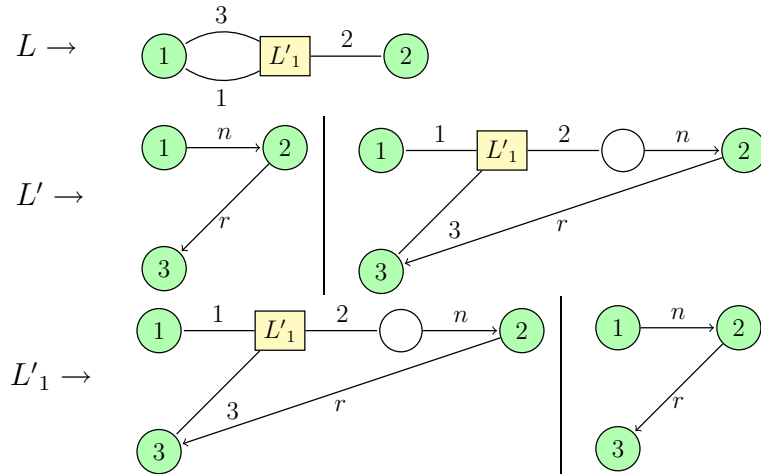
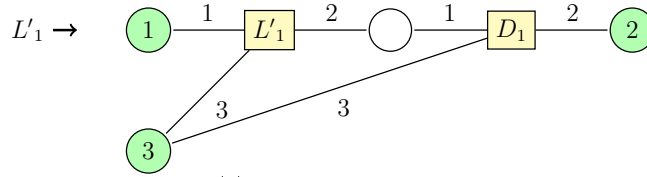


Abbildung 4.11: Erweiterte Grammatik: Einfach verkettete Liste mit Rückwärtszeigern II

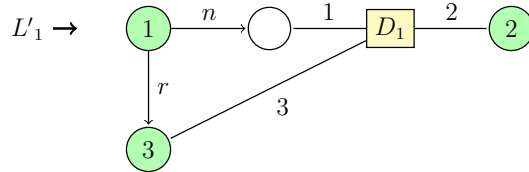
welches den ersten und letzten Schritt einer unzulässigen Ableitungsreihe simuliert, d.h. einer Ausführung mindestens zweier L'_1 -Regeln entspricht. ϕ stellt eine Funktion dar, die – intuitiv erklärt – aussagt, durch welche externen Knoten die weitergereichten externen Knoten von L'_1 in den Ableitungsschritten ersetzt werden. Für dieses Beispiel gilt also: $\phi : 1 \rightarrow 1$. Weiterhin werden die Tentakel des neuen Nichtterminals (und damit auch der Rang) definiert: $tent(D_1) = \{(3, 3), (2, \star), (\star, 2)\}$. Die Funktion $\tau : tent(D_1) \rightarrow [1, |tent(D_1)|]$ ordnet dabei jedem Tentakel bijektiv eine feste Zahl aus $rk(D_1) = |tent(D_1)|$ zu. Sei hier $\tau(\star, 2) = 1, \tau(2, \star) = 2, \tau(3, 3) = 3$. Nun werden zwei Arten von Hilfsregeln – Zwischenproduktions- und Ausstiegsregeln – erstellt, die später für die Konstruktion der Falt-Regeln genutzt werden. Die für dieses Beispiel nach der Greibach Normalform Konstruktion erstellten Hilfsregeln sind in Abbildung 4.12 zu finden.

Aus den Regeln der Grammatik aus Abbildung 4.11 werden nun die sogenannten alten Produktionsregeln und Brücken-Regeln, sowie die Falt- und Endfalt-Regeln erstellt. Die Konstruktion letzterer bezieht außerdem die Hilfsproduktionsregeln mit ein. Die vier Regeltypen sind in Abbildung 4.13 zu finden und bilden die resultierende Grammatik in erweiterter, lokaler Greibach Normalform.

Bei dem hier vorgestellten Vorgehen für die lokale Greibach Normalform kann höchstens ein verletzender Knoten – nämlich der momentan betrachtete externe Knoten – an jeder Nichtterminalkante anliegen, daher reicht für diese Anwendung immer einmaliges Falten. Eine weitere hier implizit angenommene Veränderung der Greibach NF Konstruktion aus [Eng92] resultiert direkt aus der neu eingeführten Greibach Eigenschaft, nämlich die Anpassung der Definition der Menge



(a) Zwischenproduktionsregel: D_1



(b) Ausstiegsproduktionsregel: Einfach verkettete Liste mit Rückwärtszeigern II

Abbildung 4.12: Hilfsregeln: Einfach verkettete Liste mit Rückwärtszeigern II

von NT-weiterreichenden Knoten einer Kante, $pass(e)$. In $pass(e)$ werden hier nicht alle Nichtterminalknoten vom Grad 1 aufgenommen, sondern – analog zur Definition der erweiterten, lokalen Greibach Normalform – alle Nichtterminalknoten vom Ausgangsgrad 1. Diese Änderung hat jedoch keinen Einfluß auf den Beweis des Substitutionslemmas der $pass(e)$ -Menge aus [Eng92], die Aussage kann ohne Weiteres übernommen werden.

Als weitere Variation wird hier die Greibach Konstruktion auf Grammatiken angewandt, deren Regeln Hypergraphen mit potentiell unbeschränkten Eingangsgrad erzeugen. Dies stellt jedoch kein Problem dar, da das Falten von Hypergraphen und damit die betrachtete Greibach Normalform Konstruktion für beliebige HRGs funktioniert [Eng92].

Nachdem nun die angepasste Greibach Normalform Konstruktion erläutert wurde und gezeigt wurde, dass sie hinreichende Voraussetzung für die erweiterte, lokale Apex-Eigenschaft bei Heapabstraktionsgrammatiken ist, muss als letzter und wichtiger Punkt noch gezeigt werden, dass nach der angepassten Greibach Konstruktion tatsächlich die in Definition 4.1.3 geforderten Eigenschaften erfüllt sind. Die Anforderung der Isolationsfreiheit muss nicht weiter betrachtet werden, in dieser Hinsicht ändert sich nichts an der Konstruktion der ursprünglichen Greibach Normalform. Bleibt jedoch noch zu zeigen, dass (\star) in der konstruierten Grammatik der betrachtete externen Knoten entweder einen Ausgangsgrad von ≤ 2 aufweist oder kein Nichtterminalknoten mehr ist. Betrachtet man die Entstehung der einzelnen Produktionsregeln, die nach der Greibach Konstruktion in der Grammatik vorhanden sind bzw. im Konstruktionprozess zur Herstellung der endgültigen Regeln verwandt werden, wird auch bei dieser Eigenschaft schnell klar, dass sie erfüllt sein muss. Sei die HRG G die Grammatik, für die an Nichtterminal X und externem

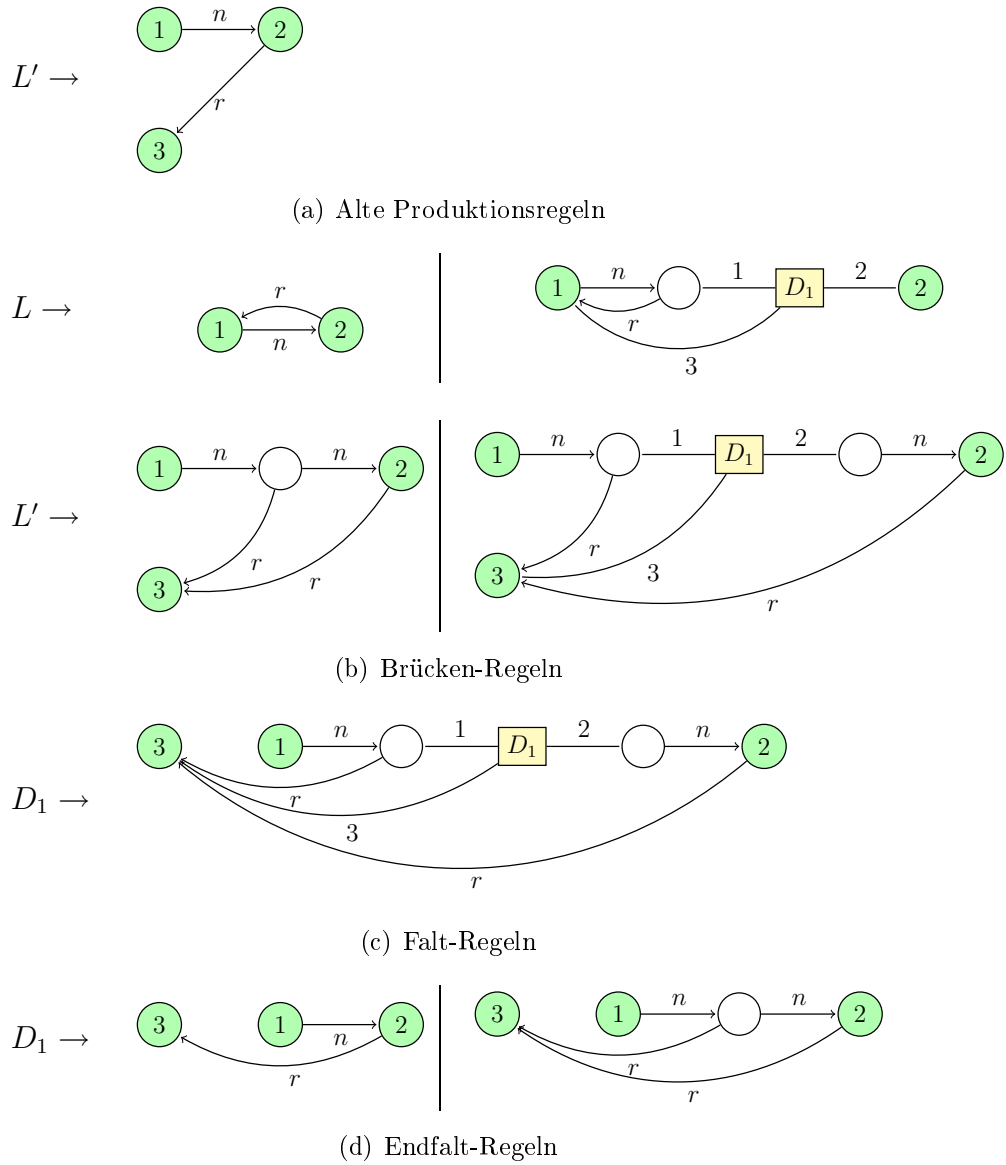


Abbildung 4.13: Lokale Greibach Normalform: Liste mit Rückwärtszeigern II

Knoten $ext(i)$, $i \in [1, rk(X)]$, eine Grammatik G' in erweiterter, lokaler Greibach Normalform konstruiert werden soll.

Betrachten wir zunächst die Entstehung der Produktionsregeln, die zur Herstellung der endgültigen Regeln gebraucht werden.

Einstiegsregel/initial production: Diese Regeln sind die Kopien der Regeln aus G , die nicht in Greibach Normalform vorliegen. Lediglich die Nichtterminalkante, die am betrachteten externen Knoten zu Ausgangsgrad 1 führt, wird in den Einstiegsregeln umbenannt.

Mittelregel/middle production: Mittelregeln entstehen wiederum durch Kopie der „unzulässigen“ Regeln aus G . Auch hier wird wie bei den Einstiegsregeln die maximale Nichtterminalkante umbenannt, zusätzlich wird der Bezeichner der linken Regelseite abgeändert.

Endregel/final production: Die Entstehung erfolgt analog zu den Mittelregeln, jedoch für Regeln aus G , die bereits in Greibach Normalform vorliegen. Da keine unzulässige Nichtterminalkante vorliegt, wird nur der Bezeichner der linken Regelseite angepasst.

Zwischenregel/intermediate production: Diese Regeln entstehen indem für jeweils zwei Mittelregeln X_1, X_2 eine Zwischenregel $X_1 \rightarrow H$ generiert wird, dessen rechte Regelseite genau zwei Nichtterminalkanten enthält und als Bezeichner auf der linken Regelseite X_1 erhält. Für die Beschriftung einer der Nichtterminalkanten wird ein neues Nichtterminal $\langle X_1, X_2, \phi \rangle$ eingeführt. Die Beschriftung der zweiten Kante entspricht der linken Regelseite der zweiten Mittelregel, also X_2 . Die Konstruktion der rechten Regelseite sieht vor, dass die externen Knoten der ersten Mittelregel übernommen werden. Am externen Knoten, an dem lokale Apex-Eigenschaft konstruiert werden soll, liegt als einzige Kante diejenige mit dem Bezeichner X_2 an. Diese Konstruktion führt folglich zu einer Regel, die nicht in Greibach Normalform vorliegt.

Ausstiegsregel/tail production: Ausstiegsregeln entstehen, indem man in jeder Zwischenregel die Nichtterminalkante am externen Knoten, für den lokale Apex-Eigenschaft konstruiert werden soll, durch jede vorhanden Endregel ersetzt. Da alle Endregeln bereits in lokaler Greibach Normalform sind, gilt dies auch für die so entstandene Ausstiegsregel.

An dieser Stelle werden nun alle Regeln analysiert, die tatsächlich in die konstruierte Grammatik G' aufgenommen werden.

Alte Produktionsregeln: Das sind diejenigen Produktionsregeln aus G , die bereits vor der Greibach Konstruktion die Eigenschaften der erweiterten, lokalen Greibach Normalform erfüllen. Diese Regeln werden ohne Veränderung nach G' übertragen und erfüllen trivialerweise (\star).

Brückenregel/bridge production: Diese Regeln entstehen, indem man in jeder Einstiegsregel

die einzige unzulässige Nichtterminalkante am betrachteten externen Knoten v durch jede vorhandene End- oder Ausstiegsregel ersetzt. Da sowohl End- als auch Ausstiegsregeln bereits in erweiterter, lokaler Greibach Normalform sind, besitzt v nach dem Ersetzungsschritt erweiterte, lokale Apex-Eigenschaft.

Faltregel/non-final folding production: Diese Regeln brauchen in die Betrachtung nicht eingeschlossen werden, da sie als linke Regelseite ein neues Nichtterminal erhalten. Dieses Nichtterminal entspricht demjenigen, das auch in den Zwischenregeln als neues Nichtterminal verwandt wird. Da dieses nie mit dem betrachteten externen Knoten verbunden wird (auch nicht durch Ableitungsschritte), sind Faltregeln trivialerweise in Greibach Normalform.

Endfaltregel/final folding production: siehe Faltregeln.

Damit besitzen alle Regeln, die in die Grammatik G' aufgenommen werden können erweiterte, lokale Greibach Normalform für das Nichtterminal X am externen Knoten $ext(i)$. Folglich liegt G' dann auch in dieser vor.

Nachdem hier alle von der ursprünglichen Greibach Normalform abweichenden Aspekte der erweiterten, lokalen Greibach Normalform nach und nach diskutiert wurden, fassen wir abschließend die gesamte Konstruktion kurz zusammen.

Konstruktion: Erweiterte, lokale Greibach Normalform

Eingabe: HRG G , Nichtterminal X sowie externer Knoten $v = ext(i)$ mit $i \in [1, rk(X)]$ mit beschränktem Ausgangsknotengrad

Ausgabe: HRG $G_{i,X}$ in Greibach Normalform

Die maximale Anzahl $maxp$ der an einer Nichtterminalkante anliegenden externen Knoten vom Grad 1 kann für diese Anwendung nicht ≥ 1 sein. Daher reicht hier eine einmalige Ausführung der Schritte 1 bis 4.

1. Erstelle zur Grammatik G den Ableitungsgraphen $G' = (V, E)$. Sei $reach(X, i)$ die Menge der Knoten aus V , die von (X, i) aus erreichbar sind.
2. Erstelle sukzessive – jeweils ausgehend von der im vorangegangenen Schritt erstellten Grammatik – die Split-Grammatik $G_{split}(Y, j)$ für alle Knoten $(Y, j) \in reach(X, i)$.
3. Erstelle die erweiterte, lokale Greibach Normalform $G_{i,X}$, wobei ein externer Nichtterminalknoten $ext(j)$ eines Nichtterminals Y als unzulässig gilt, falls $(Y, j) \in reach(X, i)$ und $|E^{out}(ext(j))| = |E^{Nout}(ext(j))| = 1$. Sei e die einzige Kante aus $E^{Nout}(ext(j))$. Dann ist $pass(e) = ext(j)$ und e damit maximal. Für die genaue Vorgehensweise in diesem Schritt siehe die Zusammenfassung der Konstruktion der Greibach Normalform aus [Eng92].
4. Als letzten Schritt wird zu der vorliegenden Grammatik $G_{i,X}$ die Ersetzungsgammatik

$\text{repl}(G_{i,X}, X, i)$ konstruiert. $\text{repl}(G_{i,X}, X, i)$ liegt dann in erweiterter, lokaler Greibach Normalform vor.

Während der Ausführung dieses Algorithmus betrachten und modifizieren wir die Split-Regel, die unter Umständen in Schritt 3 erstellt wurde, unter keinen Umständen. Sie stellt eine kleine Ausnahme dar, da ihre externen Knoten nicht paarweise disjunkt sind. Da wir wissen, dass sie immer nach spätestens zweimaliger Kantenersetzung lokale Apex-Eigenschaft besitzt, führt ihr Ausnahmestatus zu keinen Problemen und bedarf keiner weiteren Behandlung.

Komplexität Nachdem die Konstruktion der lokalen Greibach Normalform erläutert wurde, stellt sich gerade in Hinsicht des komplexen Vorgehens die Frage, mit welchem Zuwachs von Produktionsregeln gerechnet werden muss. Der Einfachheit halber nehmen wir an, dass alle externen Knoten ausser die der Split-Regel, paarweise disjunkt sind. Dies ist nach Theorem 2.3.2 keine Einschränkung, denn diese Forderung lässt sich für jede beliebige HRG herstellen. Sei $G = (N, T, P, S)$ die zu normalisierende Grammatik, $|P|$ entsprechend die Anzahl ihrer Produktionsregeln und $|N|$ die Anzahl der Nichtterminale. Sei maxRank ausserdem der maximal auftretende Rank aller Nichtterminale. Dann liegt der Zuwachs an Produktionsregeln jeder Teilgrammatik $G_{i,X}$ in Greibach Normalform bei $\mathcal{O}(|P|^3 * |N|^2 * \text{maxRank}^2)$.

Betrachten wir abschließend als ausführliches Beispiel die erweiterte, lokale Greibach Normalform Konstruktion für die Datenstruktur der Bäume mit verbundener Blattfront, die bereits in Beispiel 2.2.1 eingeführt wurden.

4.2 Lokale Greibach Normalform am Beispiel

In diesem Abschnitt soll die Konstruktion der lokalen Greibach Normalform anhand eines komplexeren Beispiels veranschaulicht werden. Als Datenstruktur wählen wir binäre Bäume mit verbundener Blattfront und führen damit Beispiel 3.2.6 aus Kapitel 3 weiter. Dort wurde aufgezeigt, dass die Grammatik für binäre Bäume mit verketteter Blattfront – wie sie in Beispiel 2.2.1 eingeführt wurde – keine zulässige Heapabstraktionsgrammatik darstellt, da sie die lokale Apex-Eigenschaft verletzt. Wir wenden also den Algorithmus zur Konstruktion der lokalen Greibach Normalform, der in diesem Kapitel vorgestellt wurde, an.

Betrachten wir zunächst, welche Regeln an welchen externen Knoten der Eingabegrammatik G nicht in lokaler Greibach Normalform vorliegen. Nach Definition ist ausschließlich der externe

Knoten 2 externer Nichtterminalknoten und es existieren zwei Produktionsregeln an dem er Ausgangsgrad 1 besitzt, diese sind in Abbildung 4.14 zu finden.

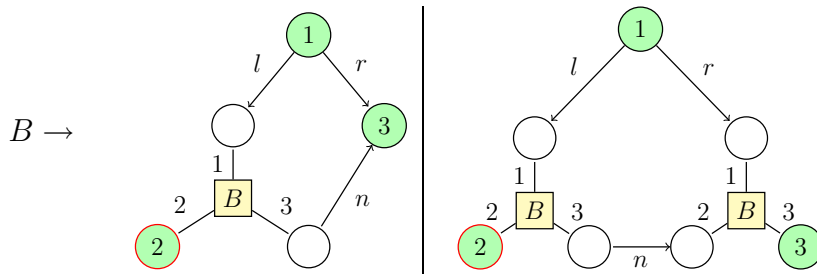


Abbildung 4.14: Lokale Apex-Eigenschaft verletzende Produktionsregeln

Wir wissen nun, dass die Greibach Konstruktion lediglich für diese B -Regeln und ausschließlich für den externen Knoten 2 durchgeführt werden muss. Der erste Schritt besteht darin, den Ableitungsgraphen und damit die Menge $reach(B, 2)$ zu berechnen. Betrachtet man die Regeln der Grammatik jedoch genauer, wird klar, dass durch beliebige Regelanwendungen immer der externe Knoten 2 auf sich selbst abgebildet wird. Der Ableitungsgraph besteht daher nur aus einer Schleife am einzigen Knoten $(B, 2)$ und wird aus diesem Grund hier nicht näher betrachtet. In Schritt zwei folgt die Konstruktion der Split-Grammatik $G_{split}(B, 2)$. Da diese aber in dem Fall der Bäume mit verketteter Blattfront nicht nötig ist, da kein Problemfall wie aus Beispiel 4.1.2 auftreten kann und die Graphen unnötig kompliziert, überspringen wir diesen Schritt. Hier sehen wir einen ersten Kritikpunkt des hier vorgestellten Algorithmus: Die Konstruktion der Split-Grammatik würde zwar keine Probleme mit sich bringen, aber sie ist in diesem Fall nicht nötig und damit Aufwand, der vermieden werden könnte. Betrachtet man allerdings die Eingabegrammatik, ist dies auf den ersten Blick nicht erkennbar. Hier wären Methoden zur Analyse hilfreich, um solche Optimierungen anwenden zu können. An dieser Stelle soll darauf aber nicht weiter eingegangen werden, sondern wir beginnen mit der eigentlichen Herstellung der Normalform. Dazu wird zuerst die erweiterte Grammatik G_1 erstellt, die lediglich zusätzliche Informationen über die unzulässigen externen Knoten in ihren Regeln enthält. G_1 besteht aus alten Produktionsregeln, Einstiegsregeln, Mittelregeln und Endregeln, die in Abbildung 4.15 zu finden sind.

Nun beginnt die Vorarbeit für die Konstruktion der Produktionsregeln, die die unzulässigen Regeln später ersetzen sollen. Für jeweils zwei Mittelregeln $(X_1, H), (X_2, H')$ mit $e \in E_H \wedge lab(e) = X_2$ wird ein neues Nichtterminal $\langle X_1, X_2, \phi \rangle$ erstellt. An dieser Stelle ergibt sich lediglich ein neues Nichtterminal $D_1 := \langle B_1, B_1, \phi \rangle$ mit $\phi : 2 \rightarrow 2$ durch die Kombination der zwei Mittelregeln aus der erweiterten Grammatik vom vorangegangenen Schritt. Seine „Tentakel-Menge“ – diejenige Menge, die später die Knoten der erstellten Regeln vorgibt und dessen Mächtigkeit den Rang diktiert – sieht wie folgt aus: $tent(D_1) = \{(1, *), (3, *), (*, 1), (*, 3)\}$. Durch eine feste, aber beliebig wählbare injektive Funktion τ wird eine Sortierung der Elemente der Tentakel-Menge

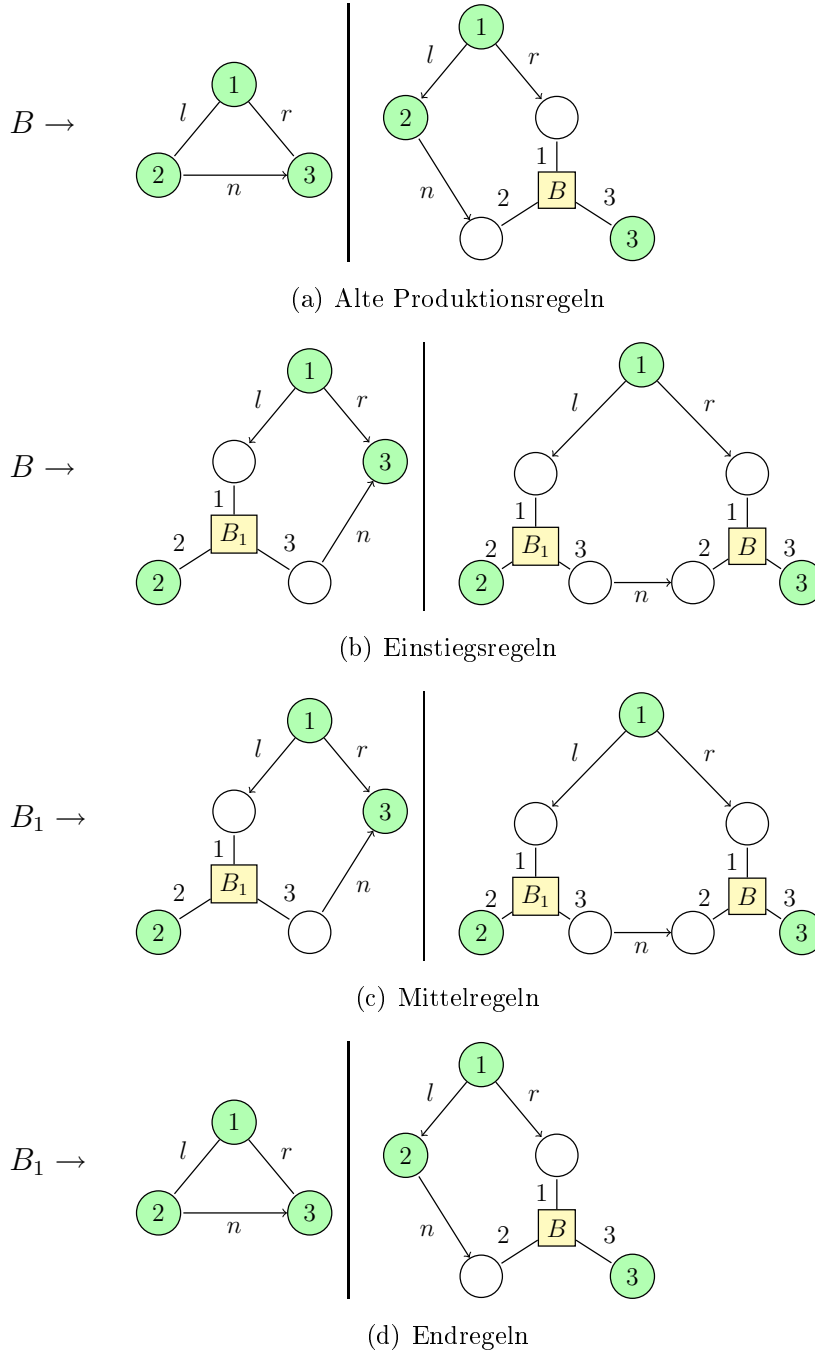


Abbildung 4.15: Erweiterte Grammatik: Baum mit verketteter Blattfront

zung in die Regel aufgenommen werden. Die resultierenden Brücken, Falt- und Endfaltregeln sind in Abbildung 4.2 und 4.19 zu finden.

Die resultierende Grammatik liegt nun in Greibach Normalform vor. In diesem speziellen Fall ist auch keine weitere Kantenersetzung nötig, um lokale Apex-Eigenschaft zu erhalten, da ein externer Nichtterminalknoten in keiner der Produktionsregeln mehr zu finden ist.

Ein ausführliches Beispiel zur händischen Erstellung einer Heapabstraktionsgrammatik für Bäume mit verketteter Blattfront ist in [Gro09] zu finden. Vergleicht man dort die resultierenden B -Regeln mit dem hier konstruierten Regelsatz, findet man viele der dort erstellten Regeln hier wieder. Es fällt allerdings auf, dass einige der hier konstruierten Regeln größer sind, insgesamt jedoch die gleiche Sprache erkannt wird. Die Konstruktion der Greibach Normalform ist folglich nicht optimal, da nicht alle hier konstruierten Regeln „minimal“ sind.

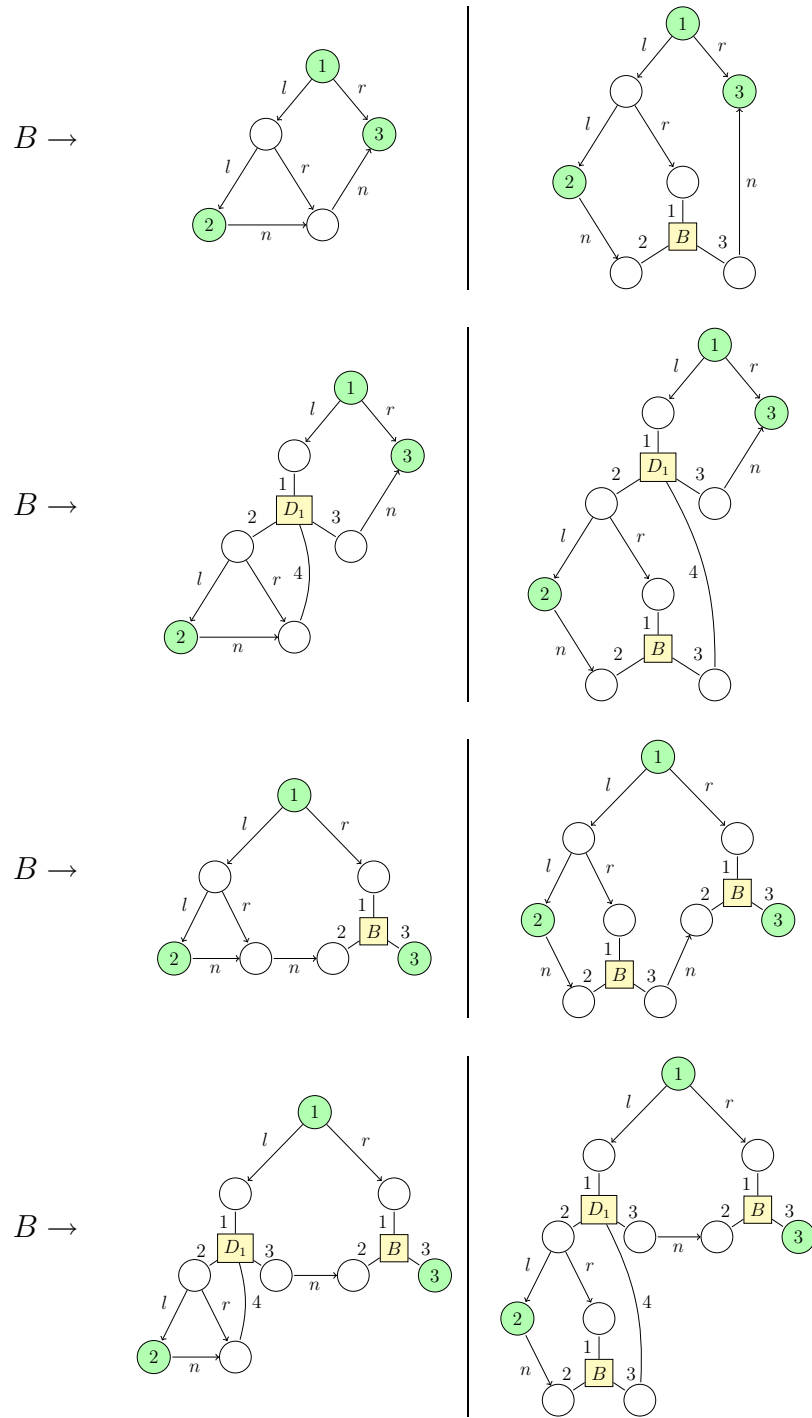
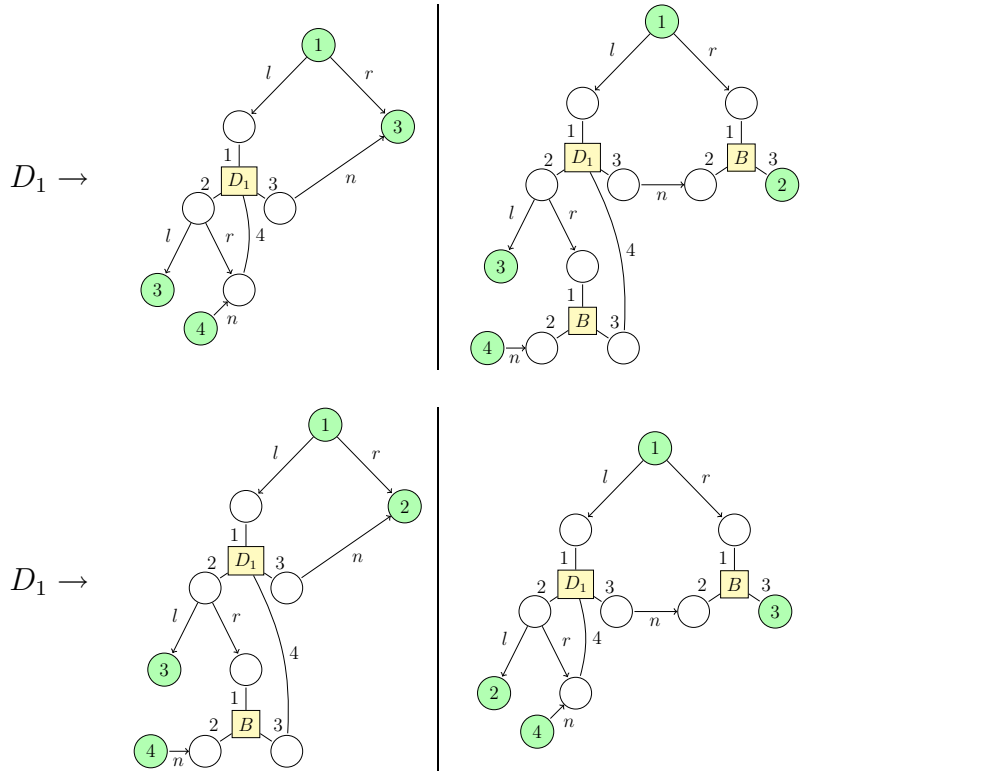
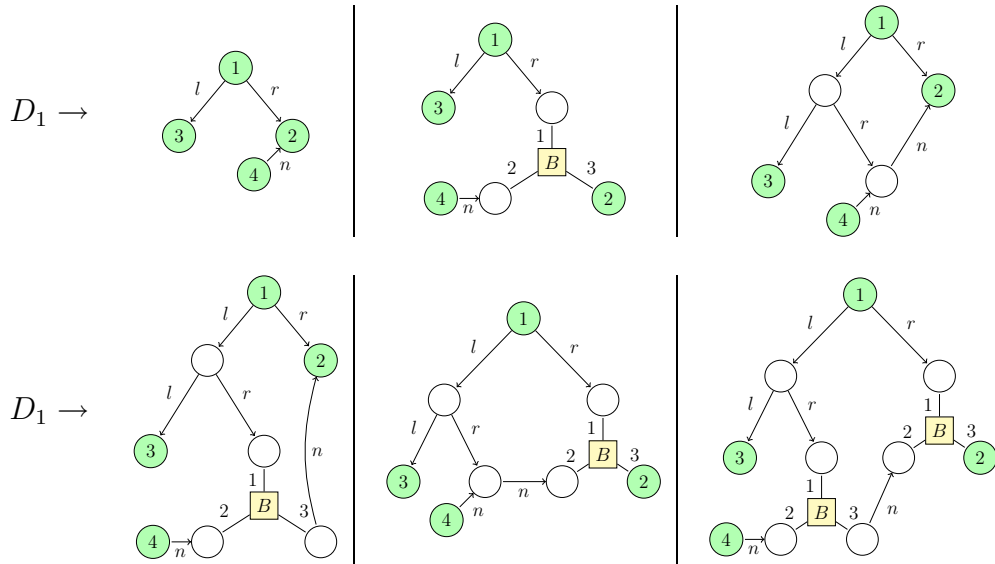


Abbildung 4.18: Greibach NF: Brückenregeln



(a) Faltregeln



(b) Endfaltregeln

Abbildung 4.19: Greibach NF: Faltregeln

Kapitel 5

Inferenz von HRGs

Nachdem in den vorangegangenen Kapiteln Heapabstraktion durch Hyperkantenersetzungsgrammatiken im Detail diskutiert wurde, stellt sich im Nachhinein die Frage, wie eine geeignete Ausgangsgrammatik, die zu einer zulässigen Heapabstraktionsgrammatik umgeformt werden kann, entsteht. Da keine universelle HRG existiert, sondern diese vielmehr stark von der betrachteten Datenstruktur und dem Zeiger-Programm abhängt, ist es wünschenswert, Leitlinien oder noch besser einen Algorithmus angeben zu können, der automatisiert eine Grammatik erstellt anstatt den Nutzer vor das Problem des Grammatikentwurfs zu stellen.

In diesem Kapitel werden wir den Ansatz der Inferenz von HRGs anhand von Beispielmengen von Graphen betrachten und ihn durch Anpassungen zu einer geeigneten Basis für einen Lernalgorithmus für Heapabstraktionsgrammatiken umzuformen.

5.1 Grammatikalische Inferenz

Als Grundlage zur Inferenz von HRGs dient der Ansatz aus [JK90], der zum besseren Verständnis der nachfolgenden Abschnitte hier eingeführt werden soll. Die Grundidee besteht darin, aus einer endlichen Menge von Beispielgraphen eine HRG abzuleiten, deren Sprache mindestens alle Graphen der Beispielmenge enthält. Letztere werden dafür iterativ in Teilgraphen zerlegt, anhand derer die Produktionsregeln der Grammatik schließlich generiert werden.

Der Algorithmus beruht im Wesentlichen auf vier Basisoperationen – INIT, DECOMPOSE, RENAME und REDUCE –. Die Operation INIT nimmt dabei eine Sonderstellung ein, denn sie generiert aus einer Beispielmenge von Graphen eine Initialgrammatik und wird daher auch

nur ein einziges Mal zu Beginn angewandt. Die weiteren drei Operationen werden nichtdeterministisch hintereinander und beliebig oft ausgeführt, um die endgültige HRG zu erhalten. Dabei lässt sich für jeden Schritt zeigen, dass die erkannte Sprache vor und nach der Anwendung einer Basisoperation entweder unverändert geblieben ist oder eine Obermenge der Eingabesprache darstellt. Die hier beschriebenen Sachverhalte formalisieren wir im Folgenden genauer.

INIT: Akzeptiert als Eingabe eine Positivmenge S^+ von ungerichteten, unbeschrifteten Beispielfraphen und generiert daraus eine HRG $G_{INIT} = (N, P, S)$ wie folgt:

$$G_{INIT} = INIT(S^+) = (S, (S, H) \mid H \in S^+, S^\bullet) \text{ mit } rk(S) = 0$$

DECOMPOSE: Dieser Schritt stellt den wichtigsten und aufwendigsten der vier Basisoperationen dar. Er akzeptiert als Eingabe eine HRG G und liefert eine modifizierte HRG G_{DEC} als Ausgabe. Die Hauptidee der DECOMPOSE-Operation besteht darin, die Produktionsregeln der Eingabegrammatik in disjunkte Teilgraphen zu zerlegen. Die Eingabegrammatik wird um Regeln, die aus den Teilgraphen entstehen, so erweitert, dass die Eingaberegeln aus den neuen Regeln ableitbar sind. Da die Eingaberegeln durch die neuen Regeln redundant sind, werden sie aus der Grammatik entfernt. $G_{DEC} = (\bar{N}, \bar{P}, S)$ entsteht aus $G = (N, P, S)$ für eine Produktionsregel $(A, R) \in P$ wie folgt:

- $\bar{N} = N \cup N_{new}$, wobei N_{new} eine Menge von Bezeichnern ist mit $N \cap N_{new} = \emptyset$.
- $\bar{P} = (P \setminus \{(A, R)\}) \cup \{(A, R_{DEC})\} \cup \{(lab(e_i), H_i) \mid 1 \leq i \leq n\}$ wobei
 - $R_{DEC} \in HRG_{\bar{N}}$
 - seien e_1, \dots, e_n die Hyperkanten in R_{DEC} mit $lab(e_i) \in N_{new}$, $1 \leq i \leq n$
 - seien H_1, \dots, H_n Hypergraphen, so dass $R_{DEC}[H_1/e_1, \dots, H_n/e_n] = R$

RENAME: Als Eingabe wird wiederum eine HRG $G = (N, P, S)$ gefordert, die Ausgabe besteht aus einer modifizierten Grammatik G_{RENAME} . Diese übernimmt alle Elemente aus G , abgesehen von der Menge der Nichtterminale und die Kantenbeschriftungen der rechten Produktionsregelseiten, benennt also lediglich Nichtterminale um. Das Resultat der Operation $RENAME(G) = G_{RENAME}$ mit $G_{RENAME} = (N', P', S')$ entsteht dann wie folgt:

- N' – Menge von Nichtterminalen
- sei $\psi : N \rightarrow N'$ die Funktion, die Nichtterminale aus G auf Nichtterminale aus N' abbildet
- sei $H \in HG_N$ mit $H = (V_H, E_H, att_H, lab_H, ext_H)$, dann entsteht $rename(H) = H'$ indem $H' = (V_H, E_H, att_H, lab', ext_H)$ mit $lab'(e) = \psi(lab_H(e))$ für alle $e \in E_H$, $lab_H(e) \in N$
- $P' = \{(\psi(A), rename(R)) \mid (A, R) \in P\}$
- $S' = rename(S)$

REDUCE: Diese Operation entfernt redundante, d.h. durch eine Kombination von anderen Regeln herleitbare, Produktionsregeln. Für eine HRG $G = (N, P, S)$ entsteht $REDUCE(G) = G_{RED}$ mit $G_{RED} = (N, P', S)$ wie folgt:

- $P' = P \setminus \{(A, R)\}$, falls eine Produktionsregel $(A, R) \in P$ existiert so, dass gilt:
 $(A, rk(R))^\bullet \xrightarrow{P'}^* R$

Es fällt auf, dass wir an dieser Stelle HRGs ohne Terminalsymbolmenge betrachten. Dies ist in diesem Zusammenhang möglich, da die Lernmenge ausschließlich unbeschriftete Graphen enthält und unbeschriftete Kanten als Terminalkanten betrachtet werden.

Für die obigen Basisoperationen lässt sich nun zeigen, dass die Grammatik vor der Anwendung immer eine Teilmenge der resultierenden Grammatik darstellt. Aufgeschlüsselt nach Operation ergibt sich:

- $L(INIT(S^+)) = S^+$
- $L(HRG) = L(DECOMPOSE(HRG))$
- $L(HRG) \subseteq L(RENAME(HRG))$
- $L(HRG) = L(REDUCE(HRG))$

Für Beweise zu den obigen Aussagen wird an dieser Stelle auf [JK90] verwiesen. Als direkte Schlußfolgerung sind wir in der Lage einen Algorithmus zum Inferieren von HRGs anzugeben.

Korollar 5.1.1 (Grammatikalische Inferenz [JK90]) *Gegeben eine endliche Menge S^+ von ungerichteten, unbeschrifteten Graphen. Sei G eine HRG, initialisiert durch $INIT(S^+)$ und modifiziert durch eine in beliebiger Reihenfolge und Häufigkeit ausgeführte Abfolge der Basisoperationen $DECOMPOSE$, $RENAME$ und $REDUCE$. Dann gilt: $S^+ \subseteq L(G)$.*

5.2 Lernen von HRGs

Der im vorangegangenen Abschnitt vorgestellte Inferenzalgorithmus aus [JK90] erfüllt in mehreren Hinsichten nicht die Anforderungen zum Inferieren von Heapabstraktionsgrammatiken. Er lässt sich allerdings durch Erweiterungen und Heuristiken soweit anpassen, dass er sich zum automatischen Lernen von HRGs eignet, die dann durch die in Kapitel 4 vorgestellte Vorgehensweise in zulässige Heapabstraktionsgrammatiken überführt werden können.

In diesem Abschnitt sollen die dem Inferenzalgorithmus fehlenden Komponenten aufgezeigt und entsprechende Modifikationen entwickelt werden, die die Nutzung zum Lernen von Heapabstraktionsgrammatiken erlauben.

Positivmenge von Hypergraphen Die Eingabemenge des Inferenzalgorithmus besteht aus ungerichteten und unbeschrifteten Graphen. Für unsere Zwecke benötigen wir einen Algorithmus, der mit (gerichteten) Terminalgraphen und Hypergraphen als Positivmenge zurechtkommt. Da sich gerichtete Terminalgraphen als Hypergraphen darstellen lassen, betrachten wir der Einfachheit halber ausschließlich letztere als gültige Beispielgraphen.

Die Anpassung des Inferenzalgorithmus aus [JK90] an Positivmengen von Hypergraphen ist glücklicherweise nicht schwer. Die Basisoperationen DECOMPOSE, RENAME und REDUCE arbeiten bereits auf HRGs und bedürfen daher nur einer kleinen Änderung: als Eingabegrammatik muss nun ein 4-Tupel (N, T, P, S) akzeptiert werden, damit mit Terminalen beschriftete Hyperkanten berücksichtigt werden können. Da die Terminalmenge T nicht von Umformungen durch diese Operationen betroffen ist, wird sie lediglich unverändert an die Ausgabegrammatik weitergegeben.

Init Die INIT-Operation generiert die Terminalmenge bei Aufruf, indem sie alle in den Beispielgraphen auftretenden Terminalsymbole in einer Menge zusammenfasst. Ansonsten arbeitet auch sie wie gewohnt, mit dem kleinen Unterschied, dass die Produktionsregeln der entstehenden Grammatik nun Heapkonfigurationen auf der rechten Regelseite enthalten statt ungerichteter und unbeschrifteter Graphen. Hier sei noch anzumerken, dass nach Aufruf der INIT-Operation die Eigenschaft der Variablenfreiheit für die Initialgrammatik nicht gewährleistet werden kann. Daher fordern wir den anschließenden Aufruf der DECOMPOSE-Operation, der Variablenfreiheit implizit herstellt, um eine gültige Heapabstraktionsgrammatik zu erhalten.

Beispiel 5.2.1 *Wir wählen als Beispieldatenstruktur wiederum binäre Bäume mit verketteter Blattfront, wie bereits in Beispiel 2.2.1 vorgestellt. Unsere Eingabemenge bestehe aus zwei Heapkonfigurationen H_1 und H_2 die in Abbildung 5.1 zu sehen sind.*

Durch den Aufruf von $INIT(\{H_1, H_2\})$ wird eine HRG $G = (N, T, P, S^\bullet)$ erzeugt, wobei S das Axiom der Grammatik darstellt. Damit erhalten wir aus der Konstruktion von INIT direkt $N = \{S\}$. Die Terminalmenge T enthält alle in den Heapkonfigurationen vorkommenden Terminalbeschriftungen, also $T = \{l, n, r\}$. Für die Produktionsregeln werden die Eingabeheapkonfigurationen als rechte Regelseiten übernommen und das Startsymbol S als Regelbeschrifter genutzt. Dies führt zur Produktionsregelmeng $P = \{(S, H_1), (S, H_2)\}$.

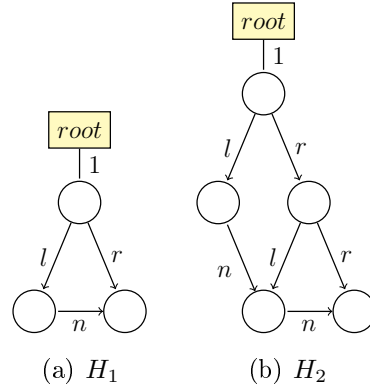


Abbildung 5.1: INIT: Positivmenge von Heapkonfigurationen

Lernen während der Programmausführung Wie bereits im vorangegangenen Abschnitt kurz angesprochen, können im Programmverlauf Heapkonfigurationen entstehen, die mit den bisher aus der Positivmenge erlernten Regeln nicht vollständig oder sogar überhaupt nicht abstrahiert werden können. Es ist erforderlich zusätzliche Produktionsregeln in die Grammatik aufzunehmen, die eine Abstraktion erlauben. Dazu erweitern wir die Positivmenge S^+ durch die problematische Heapkonfiguration und starten unseren Inferenzalgorithmus von Neuem. Um die bis zu diesem Zeitpunkt bereits erlernte Grammatik weiterverwenden zu können, definieren wir eine weitere Basisoperation EXTEND.

Extend Die Funktionsweise von EXTEND entspricht der der INIT-Operation, mit dem Unterschied, dass sie keine neue Grammatik konstruiert, sondern eine bereits Bestehende erweitert. Daher erwartet EXTEND sowohl eine HRG $G = (N, T, P, S)$, als auch eine Heapkonfiguration H als Eingabe. Die Ausgabegrammatik $G_{EXT} = (N, T', P', S)$ entsteht durch $EXTEND(G, H)$ wie folgt:

- $S^+ = S^+ \cup \{H\}$
- $T' = T \cup \{lab(e) \mid e \in E_H \wedge lab(e) \notin N\}$
- $P' = P \cup \{(S, H)\}$

Durch das Hinzufügen der unmodifizierten Heapkonfiguration kann es nun vorkommen, dass die entstandene HRG, wie auch im Fall der INIT-Operation, nicht mehr variablenfrei ist. Dieses Problem wird durch die Anpassung der DECOMPOSE-Operation behoben, die in Abschnitt 5.2.1 im Detail diskutiert werden. Wir müssen daher für jede Anwendung der EXTEND-Operation fordern, dass im Anschluss mindestens einmal ein DECOMPOSE-Schritt ausgeführt wird.

Um die Erweiterung weiterhin konsistent mit Korollar 5.1.1 und damit dem Inferenzalgorithmus aus [JK90] zu halten, müssen wir die Gültigkeit der Aussage $S^+ \subseteq L(EXTEND(HRG, HG))$ zeigen.

Bemerkung 5.2.1 *Gegeben eine Heapkonfiguration H sowie eine HRG G , die durch grammatische Inferenz aus der Positivmenge $S^+ \setminus \{H\}$ entstanden ist. Dann gilt:*

$$L(EXTEND(G, H)) \supseteq S^+$$

Beweis. Die Aussage gilt nach 5.1.1 bereits für alle $S \in (S^+ \setminus \{H\})$. Weiterhin existiert für H eine Ableitung $S^\bullet \Rightarrow H$ in $EXTEND(G, H)$ (durch Konstruktion der Basisoperation $EXTEND$). Damit folgt direkt, dass $S^+ \subseteq L(EXTEND(G, H))$. \square

Heapabstraktionsgrammatik Bis hierher berücksichtigt der Inferenzalgorithmus die Anforderungen an Heapabstraktionsgrammatiken, wie sie in Kapitel 3 aufgezeigt wurden, nicht. Da bereits in Kapitel 4 ein Verfahren zur Herstellung der komplexesten Anforderung – der Apex-Eigenschaft – vorgestellt wurde, liegt es nahe dieses Vorgehen in den Prozess des Lernens einzugliedern. Wir nutzen es, um nach jedem Schritt, in dem neue Produktionsregeln zur Grammatik hinzugefügt wurden, die resultierende HRG in eine zulässige Heapabstraktionsgrammatik zu überführen. Da dabei die erkannte Sprache unverändert bleibt, gilt weiterhin die Aussage aus Korollar 5.1.1.

Genauer formuliert wird aus der HRG eine Heapabstraktionsgrammatik hergestellt, indem nach jedem $DECOMPOSE$ -Schritt die Erfüllung der Anforderungen überprüft und gegebenenfalls hergestellt werden. Die durch $EXTEND$ -Operationen hinzugefügten Produktionsregeln brauchen nicht betrachtet werden, da sie im geforderten, anschließenden $DECOMPOSE$ -Schritt wieder aus der Grammatik entfernt werden.

5.2.1 Determinismus

Ein weiteres, gravierendes Problem des Inferenzalgorithmus aus [JK90] ist sein hochgradiger Nichtdeterminismus. Dieser macht ihn für eine Anwendung zum Lernen von HRGs während der Programmausführung sehr unattraktiv oder sogar unbrauchbar. Durch die Anwendung von Heuristiken lassen sich die Basisoperationen deterministisch durchführen. Da $INIT$, $EXTEND$ und $REDUCE$ bereits deterministisch arbeiten und es zum Lernen von HRGs ausreichend ist,

INIT und REDUCE einmal im Lernprozess und EXTEND so häufig wie nicht abstrahierbare Heapkonfigurationen auftreten zu verwenden, müssen Heuristiken für die Operationen DECOMPOSE und RENAME angewandt werden. Mit diesen setzen wir uns im nachfolgenden Abschnitt genauer auseinander.

Decompose Um für die DECOMPOSE-Operation deterministisches Verhalten zu erhalten und dabei möglichst wenige Produktionsregeln zu übergehen, die zur Heapabstraktion geeignet wären, entscheiden wir uns für eine iterative Vorgehensweise. Dabei wird die DECOMPOSE-Funktion für die Regeln der Eingabe-HRG aufgerufen, bis diese nicht weiter sinnvoll zerlegt werden können. In jedem Schritt wird dabei die Größe der zerlegten Teilgraphen und damit die der Produktionsregeln verkleinert. Zur einfacheren Umsetzung akzeptiert die neue Operation als Eingabe eine HRG G sowie eine Produktionsregel aus G , die dann weiter zerlegt wird. Die Ausgabe besteht aus einer modifizierten HRG, die stellvertretend für die Eingaberegelle in diesem Schritt neu erstellten Regeln enthält. Die genaue Vorgehensweise der angepassten DECOMPOSE-Operation ist in Algorithmus 5.1 zu finden.

Innerhalb der DECOMPOSE-Operation wird der minimale Schnitt, wie in Kapitel 2, Abschnitt 2.4 vorgestellt, berechnet und als Hilfestellung zur Bestimmung einer Zerlegung der Eingabe in disjunkte Teilgraphen eingesetzt. Warum dies nötig ist, erläutern wir anhand eines kurzen Beispiels, bevor wir zur detaillierten Beschreibung der DECOMPOSE-Operation kommen.

Beispiel 5.2.2 *Betrachten wir noch einmal die Positivmenge von Heapkonfigurationen aus Abbildung 5.1. Beide Elemente der Positivmenge enthalten einen Variablenknoten $root$, der zusammen mit der jeweiligen Heapkonfiguration an die DECOMPOSE-Funktion übergeben wird. Es beginnt die Zerlegung in Teilgraphen, ausgehend vom einzigen Variablenknoten $root$. H_1 ist nicht weiter sinnvoll zerlegbar, die Zerlegung von H_2 ohne Einbeziehung des minimalen Schnitts und die daraus resultierenden Produktionsregeln sind jedoch in Abbildung 5.2 zu finden.*

Für dieses Beispiel ist anhand der Art der generierten Regeln direkt ersichtlich, warum die Berechnung des minimalen Schnitts und eine diesen berücksichtigende Zerlegung notwendig ist: Solange die direkten Nachfolger des Knotens, von dem ausgehend Produktionsregeln erstellt werden sollen, nach der Zerlegung in demselben Teilgraphen liegen, sind alle Terminalgraphen aus der Sprache der resultierenden Grammatik in der Anzahl der n -Kanten beschränkt. Mit dieser Methode lässt sich daher z.B. für die Datenstruktur der Bäume mit verketteter Blattfront nie eine Grammatik für beliebig lange Blattfronten generieren. Durch Zerlegung mithilfe des minimalen Schnittes kann man das Lernen von solchen Grammatiken verwirklichen.

Betrachten wir nun die Methodik der DECOMPOSE-Operation genauer: Anfangs wird geprüft,

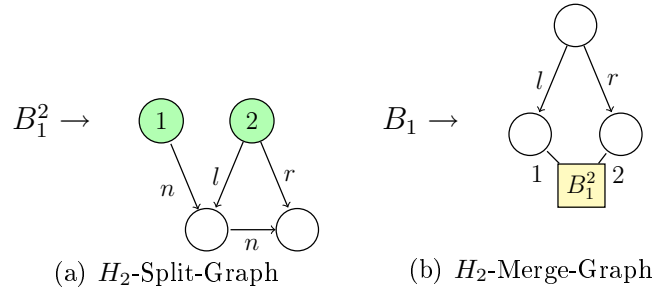


Abbildung 5.2: DECOMPOSE ohne minimalen Schnitt

ob die rechte Regelseite der Eingaberegeln überhaupt weiter sinnvoll zerlegt werden kann. Ist dies der Fall, überprüfen wir, ob die Eingabeproduktionsregel das Resultat einer INIT oder EXTEND-Operation ist (sie enthält Variablen) oder in einem vorangegangenen DECOMPOSE-Schritt erlernt worden ist. Abhängig davon werden an die SPLIT-Funktion zusätzlich zum Hypergraphen ein festgelegter externer Knoten oder die Variablenknoten als Knotenmenge *root* übergeben. Diese Knoten dürfen beim erneuten Zerlegen des Hypergraphen in keinem der zerlegten Teilgraphen enthalten sein. Wir stellen damit sicher, dass in jedem Schritt ausschließlich Subgraphen des Eingabegraphen erlernt werden, d.h. die resultierenden rechten Produktionsregelseiten immer kleiner werden.

Die SPLIT-Funktion erstellt zwei Kopien des Eingabegraphen, um diese dann zu modifizieren. Aus einer Kopie R_{split} werden die Eingabeknotenmenge *root* und alle an den Elementen aus *root* anliegenden Kanten entfernt. Gegebenenfalls werden weiterhin die Kanten eines minimalen Schnitts zwischen zwei in R_{split} verbliebenen Knoten entfernt. Der so entstandene Hypergraph enthält nun mindestens eine, häufig mehrere Zusammenhangskomponenten. Die Idee für die zweite Kopie R_{merge} des Eingabehypergraphen ist nun, sie so zu modifizieren, dass sie alle Kanten und Knoten enthält, die nicht in R_{split} vorhanden sind, sowie zusätzlich alle Knoten, die notwendig sind, um eine Hyperkante als Stellvertreter einer Zusammenhangskomponente aus R_{split} einsetzen zu können. Im Anschluss werden alle Variablenkanten aus R_{merge} entfernt, um die Eigenschaft der Variablenfreiheit nicht zu verletzen. Sowohl R_{split} als auch R_{merge} werden zur Weiterverarbeitung an die DECOMPOSE-Operationen zurückgegeben.

Die DECOMPOSE-Funktion beginnt nun für jede Zusammenhangskomponente aus R_{split} eine Produktionsregel zur Grammatik hinzuzufügen (sofern diese wachsend ist) und das Alphabet durch ein neues Nichtterminal zu erweitern. Die linke Regelseite erhält diesen neuen Nichtterminalbezeichner. Für jede so hinzugefügte Regel (A', R') wird in R_{merge} an der Stelle, wo die rechte Regelseite als Teilgraph „entnommen“ wurde, eine Hyperkante mit der Beschriftung A' und dem Rang $rk(A')$ eingefügt. Nachdem im Anschluss die Eingaberegeln (A, R) der DECOMPOSE-

Algorithm 5.1 Basisoperation: DECOMPOSE

Require: HRG $G = (N, T, P, S)$, Produktionsregel $(A, R) \in P$ **Ensure:** $G_{DEC} = DECOMPOSE(G, (A, R))$ mit $G_{DEC} = (N', T, P', S)$

```

if  $R.decomposable$  then
  if  $\exists var \in Var_G : lab_R(e) = var$  für ein  $e \in E_R$  then
     $root := \{v \in V_R \mid \exists e \in E_R, i \in rk(e) : nod(e, i) = v \wedge lab(e) \in Var_G\}$ 
     $R_{merge}, R_{split} := SPLIT(R, root)$ 
  else
     $root := \{ext_R(1) \text{ falls vorhanden, andernfalls } v_1 \in V_R\}$ 
     $R_{merge}, R_{split} := SPLIT(R, root)$ 
  end if

  for all  $R'_{split}$  mit  $R'_{split}$  Zusammenhangskomponente in  $R_{split}$  do
     $nonterminal = A_i^{|ext_{R'_{split}}|}$ , falls  $\{A_1^{|ext_{R'_{split}}|}, \dots, A_{i-1}^{|ext_{R'_{split}}|}\} \in N \wedge A_i^{|ext_{R'_{split}}|} \notin N$ 
    if  $(nonterminal, R'_{split})$  wachsend then
       $addRule(G, nonterminal, R'_{split})$ , falls  $(nonterminal, R'_{split})$  wachsend
       $addEdge(R_{merge}, nonterminal, ext_{R'_{split}})$ 
    else
       $addGraph(R_{merge}, R'_{split})$ 
    end if
  end for

   $remove(G, (A, R))$ 
   $addRule(G, A, R_{merge})$ 
else
  if  $\exists var \in Var_G : lab_R(e) = var$  für ein  $e \in E_R$  then
    for all  $e \in E_{R_{merge}} : lab(e) \in Var_G$  do
       $E_R := E_R \setminus \{e\}$ 
    end for
  end if
end if

```

Algorithm 5.2 Hilfsoperation: SPLIT

Require: Hypergraph R , Knotenmenge $root \subseteq E_R$

Ensure: Hypergraph R_{merge} & R_{split}

```

 $R_{merge} := R; R_{split} := R$ 
for all  $v \in root$  do
     $succHelp := \{v' \in V_{R_{merge}} \mid v \text{ besitzt Kante } e \text{ von } v \text{ nach } v' \wedge (nod(e, i) = v \wedge lab(e) \in N) \rightarrow terminal(lab(e), i) \neq \emptyset\}$ 
     $predeccHelp := \{v' \in V_{R_{merge}} \mid v' \text{ besitzt Kante } e \text{ von } v\} \setminus succHelp$ 
     $succ := succ \cup succHelp$ 
     $predecc := predecc \cup predeccHelp$ 
     $cutEdges := cutEdges \cup E_{R_{merge}}(v)$ 
end for

 $V_{R_{split}} := V_{R_{split}} \setminus root$ 
 $E_{R_{split}} := E_{R_{split}} \setminus cutEdges$ 
 $ext_{R_{split}} := succ \cup predecc$ 

for all  $v_1, v_2 \in succ, v_1 \neq v_2$  do
    if  $v_2 \in reach(v_1)$  then
         $minCutEdges := minCut(v_1, v_2)$ 
         $minCutVertices := \{v \in V_{R_{split}} \mid \exists e \in minCutEdges : e \in E(v)\}$ 
         $E_{R_{split}} := E_{R_{split}} \setminus minCutEdges$ 
         $ext_{R_{split}} := ext_{R_{split}} \cup minCutVertices$ 
    end if
end for

 $V_{R_{merge}} := ext_{R_{merge}} \cup root \cup succ \cup minCutVertices$ 
 $E_{R_{merge}} := cutEdges \cup minCutEdges$ 
for all  $e \in E_{R_{merge}} : lab(e) \in Var_G$  do
     $E_{R_{merge}} := E_{R_{merge}} \setminus \{e\}$ 
     $E_{R_{split}} := E_{R_{split}} \setminus \{e\}$ 
end for

return  $R_{merge}, R_{split}$ 

```

Operationen aus der Grammatik entfernt wurde, wird stellvertretend die Regel (A, R_{merge}) hinzugefügt. Sollte die Eingaberegeln nicht weiter zerlegbar sein, aber Variablen enthalten, wird sie wieder aus der Grammatik entfernt. Dieses Vorgehen ist notwendig um die Eigenschaft der Variablenfreiheit später gewährleisten zu können.

Beispiel 5.2.3 Betrachten wir die Vorgehensweise vom DECOMPOSE an zwei Beispielen. In einem Fall wird als Eingabe eine Regel (R_1, H_1) mit der Heapkonfiguration aus Abbildung 5.1(b) als rechte Regelseite verwandt, im anderen eine Regel (R_2, H_2) mit einem Hypergraph als rechter Regelseite, der weiter zerlegt werden soll. Dieser ist in Abbildung 5.3 zu sehen.

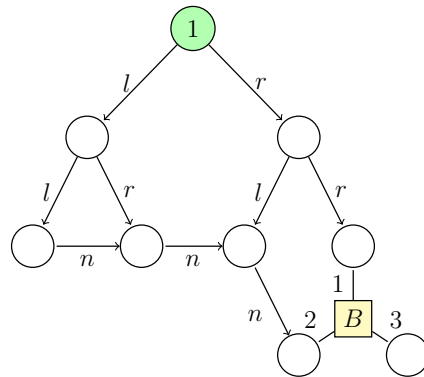


Abbildung 5.3: DECOMPOSE: Eingabehypergraph

Da beide Eingabegraphen H_1, H_2 weiter zerlegbar sind, wird die SPLIT-Funktion mit dem einzigen Variablenknoten oder dem externen Knoten 1 aufgerufen, um H_{split}^i und H_{merge}^i , $i = 1, 2$ zu berechnen. In H_{split}^i markieren wir dazu die Nachfolger- und Vorgängerknoten der Eingabeknoten und überprüfen, ob sie miteinander verbunden sind. Da das in beiden Fällen zutrifft, wird der minimale Schnitt zwischen ihnen berechnet. Die resultierenden Graphen H_{split}^i sind in Abbildung 5.4 zu sehen.

Als nächstes werden die Knoten- und Kantenmengen der beiden Graphen neu berechnet, dabei werden auch alle Kanten mit Variablenbezeichnungen entfernt. Abbildung 5.5 zeigt die nach dem SPLIT-Algorithmus modifizierten Graphen, die in diesem Zustand auch an die DECOMPOSE-Funktion zurückgegeben werden.

Anhand der zurückgelieferten Split-Graphen erstellt die DECOMPOSE-Funktion nun eine neue Produktionsregel für jede enthaltene Zusammenhangskomponente (sofern die resultierende Produktionsregel wachsend ist). Die linken Produktionsregelseiten erhalten dabei ein neues Nichtterminal, das zum Alphabet der Eingabegrammatik hinzugefügt wird. Die Merge-Graphen erhalten für jede

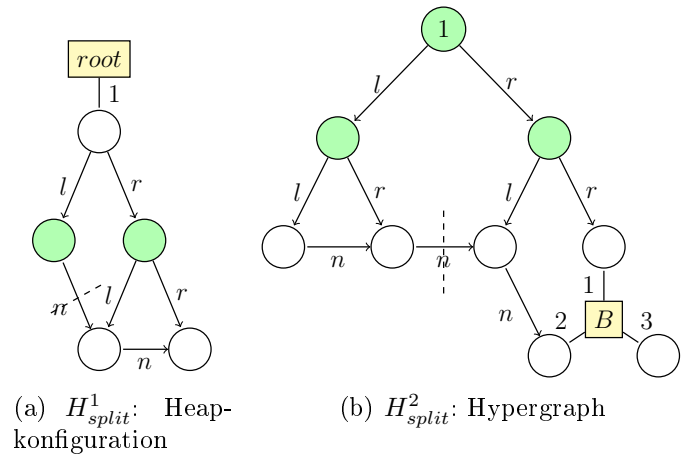


Abbildung 5.4: SPLIT: Zwischenresultat Split-Graphen

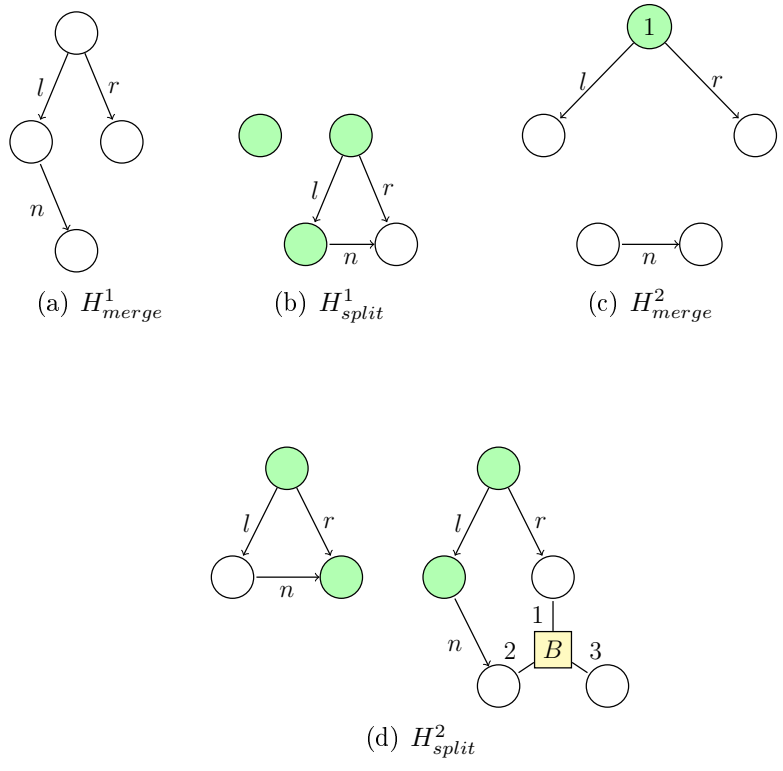


Abbildung 5.5: SPLIT: Rückgabegraphen

so abgearbeitete Zusammenhangskomponente stellvertretend für diese eine zusätzliche Kante, die mit dem neuen Nichtterminal beschriftet wird (oder den Teilgraphen, der aufgrund der Wachsend-Eigenschaft nicht erlernt wurde). In der Grammatik wird als letzter Schritt dann die Eingaberegeln durch die entsprechende Merge-Regel (R_1, H_{merge}^1) bzw. (R_2, H_{merge}^2) ersetzt. Die resultierenden Regeln sind in Abbildung 5.6 zu finden.

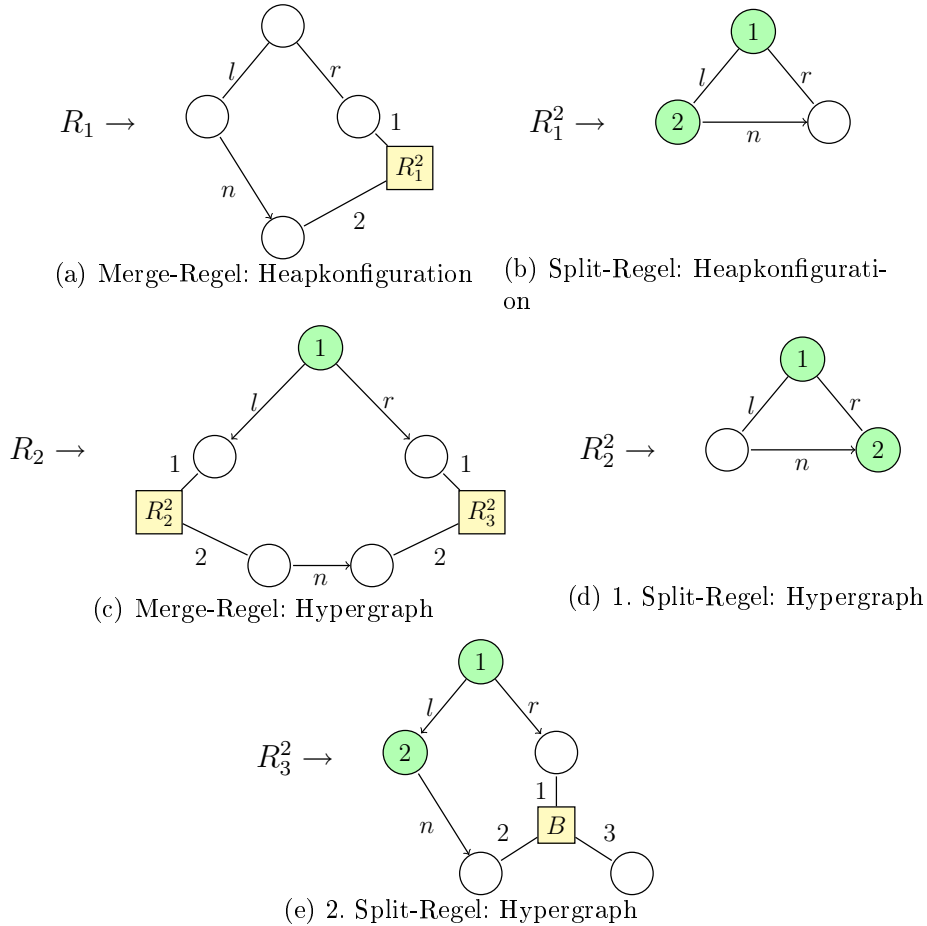


Abbildung 5.6: DECOMPOSE: erlernte Regeln

Für alle Regeln der Eingabegrammatik wird die DECOMPOSE-Operation nun so oft ausgeführt, bis keine der Regeln weiter zerlegbar ist. Zwischen jedem Schritt wird dabei überprüft, ob die lokale Apex-Eigenschaft weiterhin erfüllt ist und gegebenenfalls wiederhergestellt. Dieses Vorgehen stellt sicher, dass nach einem SPLIT-Schritt ausschließlich Terminalkanten und Reduktionstakel im Merge-Graphen verbleiben.

Die hier vorgestellte DECOMPOSE-Operationen stellt einen Spezialfall der gleichnamigen Operation aus [JK90] dar, da sie auf spezielle Weise und nur eine Produktionsregel pro Anwendung statt aller in der Grammatik enthaltenen zerlegt. An der Umsetzung der eigentlichen Dekomposition – die Eingaberegeln werden durch mehrere Regeln ersetzt aus denen erstere durch einfache Kantenersetzung wieder abzuleiten ist – finden ausser der Vorschrift, wie zerlegt wird, keine Änderungen statt. Daher gilt bei Nichtbeachtung der u.U. aus INIT/EXTEND vorhandenen Variablenkanten weiterhin: $L(HRG) = L(DECOMPOSE(HRG))$.

Rename Die RENAME-Operationen ist wichtig, um die Zahl der neu gelernten Nichtterminale in Grenzen zu halten und eine Grammatik zu erhalten, die nicht ausschließlich die Graphen der Positivmenge erkennt. Aber wie schon im vorangegangenen Abschnitt für DECOMPOSE diskutiert, möchten wir auch hier deterministisches Verhalten für RENAME erhalten. Dies wird durch Angabe einer Heuristik erreicht, die genau bestimmt, welche Nichtterminale zusammengefasst werden. Dabei wählen wir die stärkste mögliche Zusammenfassung, die eine zulässige Heapabstraktion gewährleistet, d.h. in einer typisierten Grammatik resultiert: alle Nichtterminale mit gleicher Anzahl von externen Knoten werden zusammengefasst, sofern ihre Terminalfolgen (ggf. nach Umsortierung) gleich sind. Dies lässt sich erreichen, indem die Funktion $\psi : N \rightarrow N'$ genauer spezifiziert wird. Das Resultat wird in der folgenden Redefinition der RENAME-Operation formalisiert:

Zu einer HRG G entsteht $RENAME(G) = G_{RENAME}$ mit $G_{RENAME} = (N', T, P', S')$ wie folgt:

- N' – Menge von Nichtterminalen
- sei $\psi : N \rightarrow N'$ eine Funktion, die Nichtterminale aus G auf Nichtterminale aus N' unter folgender Voraussetzung abbildet:
 - seien $N_1, N_2 \in N$, seien weiterhin $terminalset(N_i) = \{T_j \mid T_j \in terminal(N_i), 1 \leq j \leq rk(N_i)\}$ Multimengen für $i = 1, 2$: falls $rk(N_1) = rk(N_2) \wedge terminalset(N_1) = terminalset(N_2)$ gilt, dann auch $\psi(N_1) = \psi(N_2)$.
- sei $H \in HG_N$ mit $H = (V_H, E_H, att_H, lab_H, ext_H)$, dann entsteht $rename(H) = H'$ indem $H' = (V_H, E_H, att_H, lab', ext_H)$ mit $lab'(e) = \psi(lab_H(e))$ für alle $e \in E_H, lab_H(e) \in N$
- $P' = \{(\psi(A), rename(R)) \mid (A, R) \in P\}$
- $S' = rename(S)$

Für zukünftige Arbeiten auf diesem Gebiet ist es sicherlich von Interesse weitere, weniger starke Heuristiken zum Zusammenfassen der Nichtterminale zu untersuchen.

Beispiel 5.2.4 Um uns die Vorgehensweise bei der *RENAME*-Operation am Beispiel zu vergegenwärtigen, betrachten wir nochmals die Regeln aus Abbildung 5.6. Als Kandidaten zum Zusammenfassen kommen hier die drei Regeln mit den Bezeichnern R_1^2 , R_2^2 und R_3^2 in Frage, da sie denselben Rang besitzen. Weder R_1 noch R_2 kommen zum Zusammenfassen in Betracht, da keine weitere Regel mit Rang 0 bzw. 1 vorhanden ist. Aber auch R_i^2 , $i = 1, 2, 3$ lassen sich nicht einfach zusammenfassen, denn dazu muss eine weitere Voraussetzung erfüllt sein. Ihre Terminalfolgen müssen – ggf. nach Umsortierung der enthaltenen Mengen – identisch sein. Dies gilt lediglich für die Regeln mit den Bezeichnern R_1^2 und R_3^2 , weil $\text{terminal}(R_1^2) = \{l, r\}\{n\} = \text{terminal}(R_3^2)$, aber $\text{terminal}(R_2^2) = \{l, r\}\emptyset$. Eine gültige Funktion $\psi : N \rightarrow N'$ wäre daher für dieses Beispiel und unveränderte Nichtterminalmenge $N = N'$:

$$\psi(R_1) = R_1, \psi(R_2) = R_2, \psi(R_1^2) = R_1^2, \psi(R_2^2) = R_2^2, \psi(R_3^2) = R_1^2$$

RENAME verhält sich gegenüber der ursprünglichen Operation aus [JK90] analog zur *DECOMPOSE*-Operation. Die vorgestellte Modifikation benutzt eine vorgegebene Funktion ψ zur Umbenennung der Nichtterminale und ist damit auch als Spezialfall der allgemeinen *RENAME*-Funktion aus [JK90] zu sehen. Da der Beweis zur Aussage $L(\text{HRG}) \subseteq L(\text{RENAME}(\text{HRG}))$ dort eine beliebige Funktionen ψ annimmt, behält sie auch hier ihre Gültigkeit.

5.2.2 Korrektheit

Der Korrektheitsbegriff im Rahmen der Grammatikinferenz zur Heapabstraktion lässt sich schnell erschließen. Die erlernte Grammatik sollte alle vorkommenden Hypergraphen ableiten können und die vier Eigenschaften erfüllen, die an Heapabstraktionsgrammatiken gestellt werden.

Der betrachtete Lernalgorithmus für HRGs ist korrekt in dem Sinne, dass jeder Hypergraph aus der Positivmenge – nach Entfernen der Variablenkanten – auch tatsächlich durch die Grammatik ableitbar ist. Es muss noch sichergestellt werden, dass die gelernte HRG auch tatsächlich eine zulässige Heapabstraktionsgrammatik darstellt. Dies geschieht durch das Ausführen des Vorgehens aus Kapitel 4 nach jedem Schritt, in dem neue Produktionsregeln gelernt wurden. Es stellt lokale Apex-Eigenschaft und implizit auch Wachstum der so gelernten HRG sicher. Variablenfreiheit wird implizit durch die vorgestellte Konstruktion gewährleistet, da jede Heapkonfiguration mit Variablen aus der HRG entfernt wird und alle Kanten mit Variablenbeschriftungen aus den zu lernenden Hypergraphen entfernt werden. Die Eigenschaft der Typisierung kann nur in der *Rename*-Operation verletzt werden, indem Nichtterminale zusammengefasst werden, die unterschiedlichen Rang oder unterschiedliche Terminalfolgen besitzen. Dies wird durch die hier benutzte Heuristik beim *Rename* allerdings unterbunden, womit auch Typisierung der Grammatik gewährleistet

werden kann. Auf die Sicherstellung der Produktivität ist bisher nicht näher eingegangen worden. Da diese durch einfaches Löschen der unproduktiven Produktionsregeln hergestellt werden kann, werden wir auf das Vorgehen nicht näher eingehen sondern es als gegeben annehmen. Das hier vorgestellte Verfahren zum Lernen von HRGs während der Programmausführung ist damit unter den oben aufgezählten Gesichtspunkten korrekt.

5.3 Lernen von HRGs am Beispiel

Wie bereits für die Konstruktion einer Grammatik in Greibach Normalform im vorangegangenen Kapitel geschehen, soll auch an dieser Stelle der in diesem Kapitel vorgestellte Lernalgorithmus an einem durchgehenden Beispiel veranschaulicht werden. Als Grundlage dient die Heapabstraktionsgrammatik der binären Bäume mit verketteter Blattfront, die in Abschnitt 4.2 konstruiert wurde. Dabei nehmen wir an, dass die Ausgangsgrammatik aus Abbildung 2.2.1 entweder durch den Nutzer vorgegeben oder bereits inferiert wurde und dann in eine Heapabstraktionsgrammatik umgeformt wurde. Mithilfe dieser Grammatik lässt sich jeder saturierte Binärbaum konstruieren, allerdings ausschließlich top-down. Um beliebige Teilgraphen einer Heapkonfiguration – z.B. innerhalb des Baumes – für diese Datenstruktur abstrahieren zu können, werden daher weitere Produktionsregeln benötigt.

Wir nehmen an, dass im Programmverlauf die Heapkonfiguration H aus Abbildung 5.7 auftritt. Obwohl sie nicht vollständig abstrahiert ist, lässt sie sich durch die vorhandenen Produktionsregeln nicht weiter abstrahieren. Hier setzt der Lernalgorithmus an, der für diesen Fall geeignete Produktionsregeln generieren soll. Wir starten den Lernprozess, indem wir die Positivmenge um die problematische Heapkonfiguration erweitern. Die EXTEND-Operation fügt dann die Produktionsregel (S, H) zur bestehenden Grammatik hinzu, im Anschluss daran muss, wie im vorangegangenen Abschnitt beschrieben, zwingenderweise DECOMPOSE aufgerufen werden um weiterhin Variablenfreiheit der Grammatik gewährleisten zu können. Nachdem im DECOMPOSE-Schritt der Eingabegraph als Heapkonfiguration erkannt wurde, weil er Variablen enthält, wird die *root*-Menge als Menge der Variablenknoten definiert und als Eingabe an die SPLIT-Operation übergeben. In dieser Funktion werden nun die Nachfolger- und Vorgängermengen aller Knoten aus *root* berechnet. Außerdem muss für drei Paare (v, v') aus der Nachfolgermenge der minimale Schnitt berechnet werden, da v und v' verbunden sind. Abbildung 5.8 zeigt die Eingabeheapkonfiguration, in der die Knoten der Nachfolgermenge und zugehörige Kanten rot und die der Vorgängermenge blau eingefärbt sind, weiterhin werden die berechneten minimalen Schnitte durch gestrichelte Kanten angedeutet.

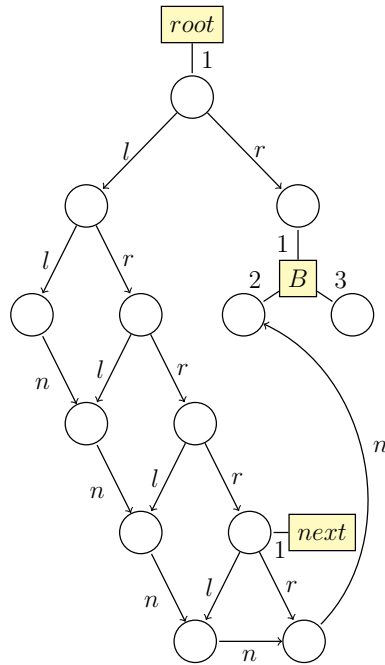


Abbildung 5.7: Lernalgorithmus: Eingabeheapkonfiguration

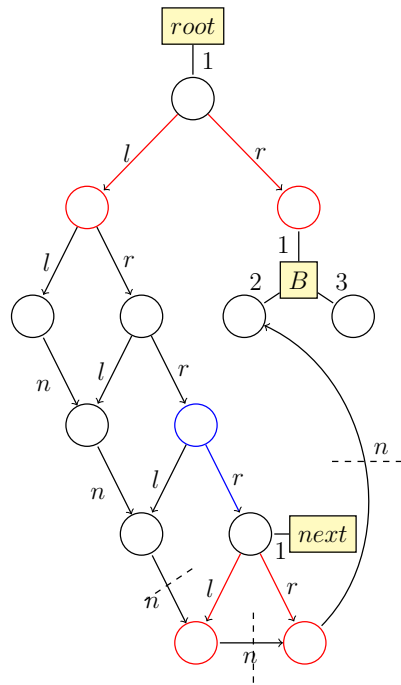


Abbildung 5.8: Lernalgorithmus: Nachfolger und Vorgänger, minimale Schnitte

Anhand der Hilfsmengen, deren Knoten bzw. Kanten in Abbildung 5.8 farbig kenntlich gemacht wurden, lassen sich die zwei Hypergraphen R_{split} und R_{merge} generieren, die zur weiteren Bearbeitung an die DECOMPOSE-Funktion zurückgegeben werden. R_{split} und R_{merge} sind in Abbildung 5.9 zu finden, wobei in R_{split} isolierte Knoten aus Gründen der Übersichtlichkeit nicht abgebildet sind. Diese isolierten Knoten haben auf die weitere Bearbeitung keinerlei Auswirkung und werden einfach ignoriert.

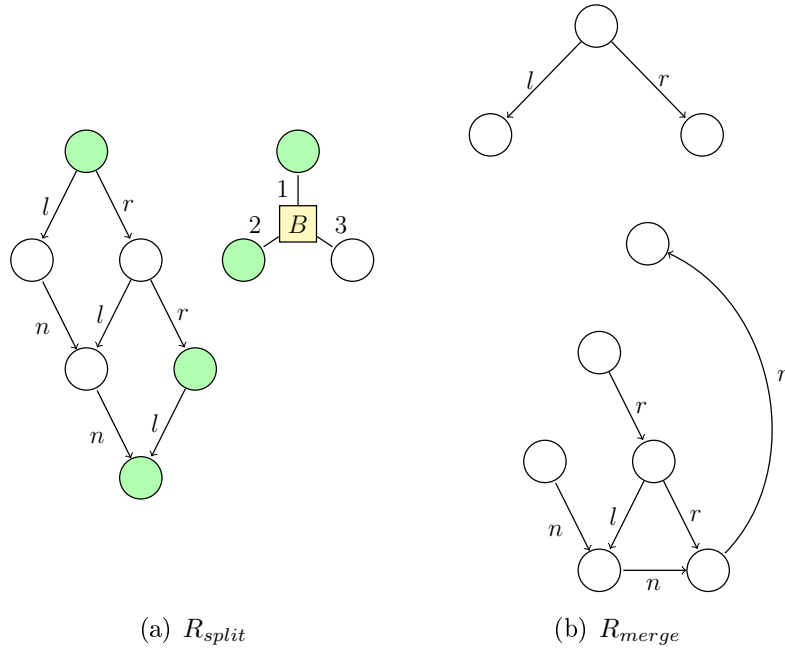


Abbildung 5.9: Lernalgorithmus: R_{split} und R_{merge}

Zurück in der DECOMPOSE-Funktion, wird für jede Zusammenhangskomponente von R_{split} eine neue Produktionsregel erstellt, sofern diese wachsend ist. In diesem Fall existiert nur eine solche Zusammenhangskomponente, da die Regel (X_1^2, B^\bullet) die Wachstumseigenschaft verletzen würde. Durch die drei in der verbliebenen Zusammenhangskomponente als extern markierten Knoten, wird als Regelbezeichner ein neues Nichtterminal X_1^3 eingesetzt. In R_{merge} wird für die neu erstellte Regel eine Nichtterminalkante mit demselben Bezeichner eingefügt. Außerdem nehmen wir die B -Kante in R_{merge} auf, da sie keine wachsende Regel induziert hätte. Schließlich wird die Eingaberegeln (S, H) durch (S, R_{merge}) ersetzt. Die resultierenden Regeln aus diesem DECOMPOSE-Schritt sind in Abbildung 5.10 zu finden.

Im nächsten Schritt muss nun für die neu hinzugekommenen Regeln wieder lokale Apex-Eigenschaft sichergestellt werden. Die hier generierten Regeln aus Abbildung 5.10 sind bereits apex. Daher

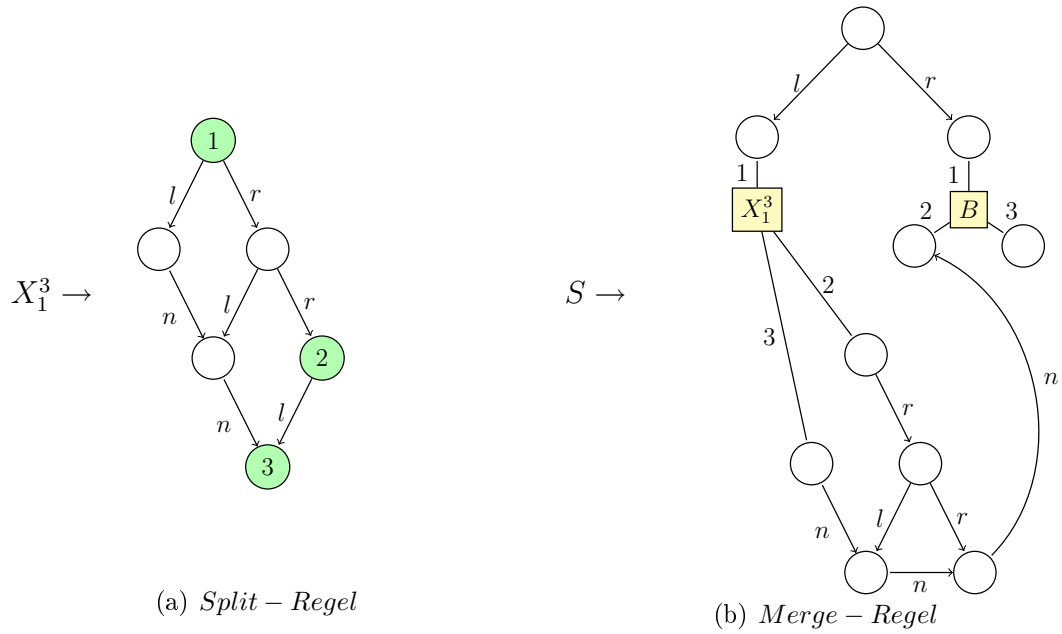


Abbildung 5.10: Lernalgorithmus: resultierende Merge- und Splitregeln

kann direkt der nächste DECOMPOSE-Schritt ausgeführt werden, den wir hier aber nicht weiter betrachten werden. Dieser gesamte Vorgang wird solange wiederholt bis keine der Grammatikregeln weiter zerlegbar ist.

Kapitel 6

Zusammenfassung & Ausblick

In der modernen Softwareentwicklung kommt der Verifikation eine immer größere Bedeutung zu, da zuverlässige Software in immer mehr Einsatzbereichen und in immer kürzerer Zeit benötigt wird. Programmierkonzepte, die mithilfe von dynamischen, heapbasierten Datenstrukturen arbeiten und heutzutage in einer Vielzahl von Programmen zu finden sind, induzieren jedoch eine unendliche Menge an möglichen Heapzuständen und lassen sich aus diesem Grund nicht durch verbreitete Verifikationstechniken evaluieren. Abhilfe schafft das Konzept der Heapabstraktion durch Hyperkantenersetzungsgrammatiken, wobei eine geeignete HRG allerdings stark vom Zeiger-Program abhängt und dem Anwender meist garnicht oder nur sehr schwer ersichtlich ist. Um das Problem des Grammatikentwurfs zu umgehen, wurde in dieser Diplomarbeit ein Algorithmus entwickelt, der zu einem gegebenen Programm eine Hyperkantenersetzungsgrammatik generiert, die alle zur Korrektheit des Abstraktionsverfahrens benötigten Eigenschaften besitzt.

Beginnend mit den Grundlagen der Heapabstraktion durch Hyperkantenersetzungsgrammatiken, wurden zunächst die in [RN08, HNR09] gestellten Anforderungen an eine Heapabstraktionsgrammatik genauer untersucht und dahingehend angepasst, dass sie als Erweiterung die Abstraktion von Hypergraphen mit unbeschränktem Eingangsgrad erlauben und zudem zu einer gegebenen HRG die Konstruktion einer äquivalenten HRG mit geforderten Eigenschaften ermöglichen. Weiterhin wurden die Anforderungen an Heapabstraktionsgrammatiken um die Typisierungseigenschaft ergänzt, die zur Korrektheit der Abstraktion und Konkretisierung von Heapkonfigurationen nötig ist. Unter den so gegebenen Voraussetzungen wurde für die komplexeste Anforderung – die lokale Apex-Eigenschaft - ein Verfahren entwickelt, welches als Resultat eine diese Eigenschaft erfüllende, äquivalente Grammatik liefert. Die lokale Apex-Eigenschaft ist für die Korrektheit der Heapabstraktion von entscheidender Bedeutung, da sie sicherstellt, dass beim der Konkretisierung alle relevanten Heapkonfigurationen wiederhergestellt werden können. Ohne sie würden unter Umständen aufgetretene Heapzustände im Verifikationsprozess nicht betrachtet und jegliche Aussagekraft des Resultats ginge verloren. Das Verfahren zur Sicherstellung der lokalen

Apex-Eigenschaft beruht auf der Herstellung einer Normalform für die Eingabegrammatik. Die Beschaffenheit der Normalform ist dabei so gewählt, dass ausgehend von einer normalisierten Grammatik lokale Apex-Eigenschaft durch einfache Kantenersetzung erreicht werden kann. Es wurde gezeigt, dass zu jeder HRG mit beschränktem Ausgangsgrad eine Grammatik in dieser Normalform existiert. Mit dem vorliegenden Algorithmus kann sichergestellt werden, dass eine beliebige HRG in lokaler Apex-Eigenschaft vorliegt – solange ihre Sprache beschränkten Ausgangsgrad besitzt – und wachsend ist. Die Eigenschaft der Produktivität ist durch einfaches Entfernen von unproduktiven Regeln möglich.

Als nächster Schritt wurde ein Algorithmus erarbeitet, welche das automatisierte Lernen von weiteren Produktionsregeln zu einer gegebenen HRG realisiert. Die Startgrammatik kann dabei durch den Nutzer vorgegeben oder anhand von einer Positivmenge von Heapkonfigurationen erstellt werden. Solange die Startgrammatik in variablenfreier Form vorliegt – und dies lässt sich leicht überprüfen – garantiert der Lernalgorithmus Variablenfreiheit für alle weiteren Produktionsregeln, die zur Grammatik hinzugefügt wurden. Der Lernalgorithmus beruht auf der schrittweisen Dekomposition von Heapkonfigurationen der Eingabemenge bzw. von vorhandenen Regeln. Dabei wird nach jedem Dekompositionsschritt die Konstruktion der Apex-Eigenschaft ausgeführt. Auf diese Weise lässt sich garantieren, dass die resultierende Grammatik eine zulässige Heapabstraktionsgrammatik darstellt, mit allen damit verbundenen Eigenschaften.

Zusammenfassend lässt sich sagen, dass in dieser Arbeit die Grundlagen der Heapabstraktion durch Hyperkantenersetzungsgrammatiken aus [HNR09] dargelegt und erweitert und ein Verfahren vorgestellt wurde, welches zu einer gegebenen HRG eine Grammatik mit lokaler Apex-Eigenschaft konstruiert. Mithilfe dieses Verfahrens wurde dann ein Algorithmus entwickelt, um zu einem Zeiger-Programm eine geeignete und zulässige Heapabstraktionsgrammatik zu generieren.

Am Beispiel von binären Bäumen mit verbundener Blattfront aus Abschnitt 4.2 haben wir gesehen, dass das Vorgehen zur Sicherstellung der lokalen Apex-Eigenschaft nicht optimal ist. Aus diesem Grund ist eine interessante Fragestellung für weitere Arbeiten auf diesem Gebiet, ob sich die konstruierte Grammatik in Greibach Normalform durch Optimierungen verkleinern lässt und der Algorithmus durch weiterführende Analyse der Eingabegrammatik beschleunigt werden kann. Auch für den im Kapitel 5 vorgestellten Lernalgorithmus für HRGs sind während der Erstellung der Diplomarbeit weitere Aspekte und Fragen aufgekommen, dessen genauere Untersuchung von weiterem Interesse ist. Hierzu gehört die Frage nach der Terminierung des Lernalgorithmus. Lässt sich etwa zeigen, dass wenn eine Heapabstraktionsgrammatik zum betrachteten Zeigerprogramm existiert, diese dann vom Lernalgorithmus tatsächlich gefunden wird? Bisher wurde der Lernalgorithmus immer auf einem beliebigen, aber festem externen Knoten gestartet. Es wäre nützlich zu klären, ob sich die von der erstellten Heapabstraktionsgrammatik erkannte Sprache ändert, wenn man einen anderen externen Knoten auswählt. Ein weiterer interessanter Ansatzpunkt betrifft

das Zusammenfassen der Nichtterminale im Lernalgorithmus. Hier ließe sich z.B. untersuchen, ob durch eine Kategorisierung der Nichtterminale zusätzlich zum Rang und der Terminalfolge die Sicherstellung von Eigenschaften wie Baumeigenschaft bei der Verifikation vereinfacht bzw. erst ermöglicht werden kann.

Literaturverzeichnis

- [Cou87] COURCELLE, BRUNO: *An axiomatic definition of context-free rewriting and its application to NLC graph grammars*. Theoretical Computer Science, 5:141–181, 1987.
- [EFS56] ELIAS, P., A. FEINSTEIN und C. SHANNON: *A note on the maximum flow through a network*. Information Theory, IRE Transactions on, 2(4):117–119, 1956.
- [EH92] ENGELFRIET, JOOST und LINDA HEYKER: *Context-free hypergraph grammars have the same term-generating power as attribute grammars*. Acta Inf., 29(2):161–210, 1992.
- [EHL94] ENGELFRIET, JOOST, LINDA HEYKER und GEORGE LEIH: *Context-Free Graph Languages of Bounded Degree are Generated by Apex Graph Grammars*. Acta Inf., 31(4):341–378, 1994.
- [Eng92] ENGELFRIET, JOOST: *A Greibach Normal Form for Context-free Graph Grammars*. In: *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, Seiten 138–149, London, UK, 1992. Springer-Verlag.
- [Eng97] ENGELFRIET, JOOST: *Context-free graph grammars*. Seiten 125–213, 1997.
- [FF56] FORD JR, L. R. und D. R. FULKERSON: *Maximal Flow Through a Network*. Canadian Journal of Mathematics, 8(3):399–404, Juni 1956.
- [Gro09] GROSSMANN, RALF: *Heapabstraktion durch partielle Graphreduktion mittels Graphgrammatiken*. Diplomarbeit, RWTH Aachen, April 2009.
- [Hab92] HABEL, A.: *Hyperedge Replacement: Grammars and Languages*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.
- [HNR09] HEINEN, JONATHAN, THOMAS NOLL und STEFAN RIEGER: *Juggernaut: Graph Grammar Abstraction for Unbounded Heap Structures*. In: *TTSS '09: 3rd Int. Workshop on Harnessing Theories for Tool Support in Software*, 2009.
- [JK90] JELTSCH, ERIC und HANS-JÖRG KREOWSKI: *Grammatical Inference Based on Hyperedge Replacement*. In: *Graph-Grammars and Their Application to Computer Science*, Seiten 461–474, 1990.
- [Lau88] LAUTEMANN, CLEMENS: *Efficient Algorithms on Context-Free Graph Grammars*. In: *ICALP '88: Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, Seiten 362–378, London, UK, 1988. Springer-Verlag.

- [RN08] RIEGER, STEFAN und THOMAS NOLL: *Abstracting Complex Data Structures by Hyperedge Replacement*. In: *ICGT '08: Proceedings of the 4th international conference on Graph Transformations*, Seiten 69–83, Berlin, Heidelberg, 2008. Springer-Verlag.