

MASTERARBEIT

TRANSFORMATION VON
SEPARATION-LOGIC-PRÄDIKATEN DURCH
HYPERKANTENERSETZUNGSGRAMMATIKEN

REARRANGEMENT AND ABSTRACTION
OF SEPARATION LOGIC PREDICATES USING
HYPEREDGE REPLACEMENT GRAMMARS

*Florian Göbe
RWTH Aachen*

*Gutachter:
apl. Prof. Dr. Thomas Noll
Prof. Dr. Ir. Joost-Pieter Katoen*

MASTERARBEIT

TRANSFORMATION VON SEPARATION-LOGIC-PRÄDIKATEN DURCH HYPERKANTENERSETZUNGSGRAMMATIKEN

REARRANGEMENT AND ABSTRACTION OF SEPARATION LOGIC PREDICATES USING
HYPEREDGE REPLACEMENT GRAMMARS

Florian Göbe,
B. Sc. RWTH
RWTH Aachen
florian.goebe@rwth-aachen.de

Gutachter:
apl. Prof. Dr. Thomas Noll
Prof. Dr. Ir. Joost-Pieter Katoen

DANKSAGUNG

An dieser Stelle möchte ich mich bei allen Menschen bedanken, die mich während der Fertigstellung dieser Arbeit unterstützt haben.

Ich bedanke mich bei meinen Gutachtern apl. Prof. Dr. Thomas Noll und Prof. Dr. Ir. Joost-Pieter Katoen für die Annahme und Betreuung meiner Abschlussarbeit.

Mein besonderer Dank gilt meiner direkten Betreuerin Dipl.-Inform. Christina Jansen, die immer ein offenes Ohr für meine Fragen hatte und mir bei schwierigen Problemen stets beratend zur Seite stand.

Auch möchte ich mich bei Dipl.-Inform. Jonathan Heinen für viele interessante und anregende Gespräche bedanken.

Zu guter Letzt danke ich meinen Eltern für die Unterstützung bei der Themenwahl und den Druck der Ausarbeitung.

ERKLÄRUNG

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Sämtliche Zitate wurden eindeutig als solche gekennzeichnet.

Aachen, 28. September 2012

(Florian Göbe)

INHALTSVERZEICHNIS

1. EINLEITUNG	1
2. EINFÜHRUNG IN DATENSTRUKTURGRAMMATIKEN	5
2.1. Hypergraphen und Heapkonfigurationen	6
2.2. Hyperkantenersetzungs- und Datenstrukturgrammatiken	9
2.3. Abstraktion und Konkretisierung	12
3. EINFÜHRUNG IN SEPARATION LOGIC	21
3.1. Intuitive Erläuterung der Separation Logic	22
3.2. SL-Fragment zur Heapabstraktion	24
4. ÜBERSETZUNG ZWISCHEN HRGEN UND SLF	31
4.1. Untersuchung beider Ansätze	31
4.2. Synthese von HRGs aus SLF-Formeln	39
4.3. Korrektheit der Synthesefunktion	53
4.4. Generierung von SLF-Formeln aus HRGs	64
4.5. Korrektheit der Übersetzungsfunktion	71
5. HAG-EIGENSCHAFTEN FÜR SL-FORMELN	79
5.1. Produktivität	79
5.2. Getyptheit	81
5.3. Wachstum	84
5.4. Lokale Konkretisierbarkeit	86
5.5. HAG-äquivalente Formeln	91
6. BEWEISREGELN FÜR DIE SL-VERIFIKATION	93
6.1. Rearrangement- und Abstraktionsregeln für Prädikate	93
6.2. Erzeugung von Beweisregeln aus HAGs	97
6.3. Verifikation eines Beispielprogramms	102
7. ZUSAMMENFASSUNG UND AUSBLICK	107
7.1. Zusammenfassung	107
7.2. Ausblick	109

Inhaltsverzeichnis

A. NOTATION	111
A.1. Verwendete Formelzeichen	111
A.2. Funktionen	112
LITERATURVERZEICHNIS	113

KAPITEL 1

EINLEITUNG

Dynamische Datenstrukturen haben in den vergangenen Jahrzehnten innerhalb der Softwareentwicklung stetig an Bedeutung gewonnen [Bal04]. Diese bestehen aus selbständigen Objekten, die auf dem Heap in zuvor allozierten Speicherbereichen abgelegt sind. Im Gegensatz zu statischen Datenstrukturen, wo der Compiler die Adressen aller Strukturen und Variablen kennt, speichern die Elemente dynamischer Datenstrukturen ihre Adressen gegenseitig zur Laufzeit, so dass eine Vernetzung zwischen ihnen entsteht. In einer verketteten Liste beispielsweise hält jedes Listenelement die Adresse seines Nachfolgers in Form eines Zeigers oder einer Referenz. Durch sukzessives Dereferenzieren dieser Zeiger kann die gesamte Liste traversiert und jedes Element betrachtet werden.

Neben Listen können auch eine Vielzahl anderer strukturellen Paradigmen, wie Bäume, Grids, etc. realisiert werden, je nachdem, welche und wie viele Adressen anderer Elemente in jedem Objekt hinterlegt werden. Der Vorteil dynamischer Datenstrukturen liegt in ihrer variablen Größe. So können ihnen in der Regel Elemente hinzugefügt oder aus ihnen entfernt werden, ohne, dass dafür die gesamte Datenmenge verschoben oder gar der gesamte Speicher neu alloziert werden muss. Dynamische Datenstrukturen werden in unzähligen Bereichen eingesetzt und sind aus modernen Programmiersprachen nicht mehr wegzudenken.

Software wird heute in nahezu allen Bereichen der Wirtschaft, Industrie und Wissenschaft eingesetzt. In vielen Fällen übernimmt sie dabei inzwischen auch sicherheitskritische Aufgaben, für die früher reine Hardwareschaltungen oder Personal verantwortlich waren. Andere, zum Teil sicherheitskritische, Systeme wurden aufgrund ihrer Komplexität aber auch erst durch den Einsatz von Software wirtschaftlich realisierbar, man denke beispielsweise an komplexe Steuerungsaufgaben in der Raumfahrt oder regelungsüberwachende Prozesse in der Medizintechnik.

In diesen oder ähnlichen Bereichen werden besonders hohe Ansprüche an die Zuverlässigkeit und insbesondere das fehlerfreie Verhalten der eingesetzten Systeme gestellt. Seit den späten Sechzigerjahren werden deshalb Methoden entwickelt, die die Korrektheit von Software in Bezug auf bestimmte Anforderungen mathematisch beweisen und damit über die Möglichkeiten einfacher Tests weit hinausgehen. Diese sogenannte *Softwareverifikation* basiert vielfach auf diversen Methoden der Modellierung im Sinne von Zuständen, in denen sich ein Programm befinden kann. Dynamische Datenstrukturen spannen jedoch aufgrund ihrer nicht-fixen Größe in den meisten Fällen einen unendlichen Zustandsraum auf, bei dem die gängigen Verifikationstechniken an ihre Grenzen stoßen. Das Gleiche gilt für zahlreiche stati-

Kapitel 1. Einleitung

sche Analyseverfahren, die bisher nur auf fest-adressierten Teilen der Programme eingesetzt werden können. Die Untersuchung verschiedener Möglichkeiten, diesen Zustandsraum durch eine geeignete Form der Abstraktion auf endliche und damit verifizierbare Größe zu bringen ist daher Gegenstand der aktuellen Forschung. Der Kerngedanke ist dabei, den vorgegebenen Aufbau der verwendeten Datenstruktur auszunutzen. So sind zum Beispiel alle Elemente in einer Liste gleich aufgebaut und werden aus diesem Grund von der Software entsprechend gleich behandelt. Die Frage, ob darin gerade das dritte, fünfte oder zehntausendste Objekt betrachtet wird, ist daher in den meisten Fällen für die Verifikation irrelevant. Diese soll zeigen, dass das untersuchte Programm *alle* Elemente korrekt behandelt. Im Beispiel der Traversierung von Listen gibt es daher im Wesentlichen drei abstrakte Zustandsklassen: Das Programm befindet sich am Anfang der Liste, irgendwo in der Mitte oder am Ende. Basierend auf diesen drei *symbolischen Zuständen* kann schließlich die Verifikation erfolgen.

In dieser Arbeit werden zwei Konzepte vorgestellt, die sich mit der Modellierung und Abstraktion dynamischer Datenstrukturen beziehungsweise des Heaps beschäftigen. Der erste betrachtete Ansatz entstammt dem Juggernaut-Projekt [HJKN12] in dessen Rahmen diese Arbeit entstand. Dabei wird die Vernetzung von Objekten innerhalb einer Datenstruktur als *gerichteter Graph* modelliert, dessen Knoten die einzelnen Objekte und dessen Kanten die Zeiger zwischen ihnen repräsentieren. Der konzeptionelle Aufbau von Datenstrukturen wird dabei durch kontextfreie Grammatiken über Graphen, sogenannte *Graphgrammatiken* definiert. Diese haben eine ähnliche Gestalt wie kontextfreie Stringgrammatiken, also solche, die eine Sprache von Wörtern definieren. Anstelle von Zeichen verwenden sie jedoch Knoten und Kanten, sodass ihre Sprache keine Wörter sondern stattdessen Graphen enthält. Nichtterminale dabei durch sogenannte *Nichtterminalkanten* repräsentiert, die – analog zu Stringgrammatiken – durch die Anwendung von Produktionsregeln substituiert werden. Man bezeichnet diese Art von Grammatik daher auch als *Hyperkantenersetzungsgrammatik*.

Heinen et al. definieren spezielle Anforderungen an die Produktionsregeln dieser Grammatiken, die es erlauben, Teile von Graphen mittels dieser zu abstrahieren und die Graphen so auf eine kompaktere Form zu bringen. Werden diese Abstraktionen kaskadiert angewandt, ermöglichen sie es in vielen Fällen, unendlich viele Instanzen einer Datenstruktur in einer endlichen Abstraktion darzustellen und sie so für die Verifikation nutzbar zu machen.

Der zweite vorgestellte Ansatz ist die von Reynolds und O’Hearn entwickelte und 2002 erstmalig vorgestellte *Separation Logic* („Abspaltungslogik“, kurz SL). Ebenso wie die lange etablierte Hoare-Logik, aus der sie hervorging und deren Erweiterung sie darstellt, spezifiziert sie einen Programmzustand mittels sogenannter *Assertionen* (engl. *Assertions*). Dabei handelt es sich um prädikatenlogische Formeln, die den Speicherinhalt charakterisieren. Im Gegensatz zur Hoare-Logik, die sich dabei auf Aussagen über Variablen beschränkt, wird in der Separation Logic auch der Heap betrachtet. In Bezug auf die Assertionen selbst ist sie jedoch bis auf wenige Ausnahmen zur Hoare-Logik abwärtskompatibel. Der Name *Separation Logic* wurde in Anlehnung an ihren wichtigsten neu eingeführten Operator, der *separierenden Konjunktion* (engl. *separating conjunction*) gewählt. Diese wird dazu verwendet, die Spezifikationen zweier disjunkter Heapteile miteinander zu verknüpfen. Auf die Art lassen sich Aussagen über den gesamten Speicher in einer Assertion zusammenfassen.

Dynamische Datenstrukturen werden in der Separation Logic durch parametrisierte *Prädikate* beschrieben. Diese definieren, wie in der Prädikatenlogik üblich, eine Relation über ihre

Argumente und ordnen ihnen innerhalb einer Assertion einen Wahrheitswert zu. Dieser gibt an, ob die Argumente die betrachtete Datenstruktur formen oder nicht. Verifikationsbeweise werden gemäß dem Vorbild der Hoare-Logik als Beweisbäume über eine Programmspezifikation geführt. Der Beweiser versucht dabei, die Spezifikation auf Axiome zurückzuführen, aus denen die Behauptung nach endlich vielen Beweisschritten folgt. Diese Schritte werden durch Anwendung von festen Beweisregeln durchgeführt, die aus einer oder mehreren *Prämissen* eine Schlussfolgerung, die sogenannte *Konklusion* ziehen. Die Gültigkeit der Regeln selbst wurde zuvor separat und allgemeingültig bewiesen.

Wie die Assertionen selbst sind auch Prädikate üblicherweise als Separation-Logic-Formeln über einer Sequenz von Parametern definiert. Um die Spezifikation eines Programmes, die üblicherweise unter Verwendung von Prädikaten gegeben ist, auf Axiome zurückzuführen, die ihrerseits Aussagen über konkrete Speicherinhalte fordern, werden spezielle Regeln benötigt. Diese *konkretisieren* ein Prädikat unter Betrachtung aller möglichen Ausprägungen und erlauben so die Rückführung auf eine axiomatische Basis. Wie auch bei der Hoare-Logik treten in SL-Beweisen häufig sogenannte Entailments, also Schlussfolgerungen, in den Prämissen auf, die manuell zu beweisen sind. Ebenso müssen die Invarianten zur Verifikation von Schleifen in den aktuellen Ansätzen noch von Hand angegeben werden. Insgesamt kann die Separation Logic auf dem derzeitigen Entwicklungsstand als teilautomatisierter beziehungsweise teilautomatisierbarer Beweiskalkül verstanden werden.

INHALT UND AUFBAU DER ARBEIT

Die beiden Modellierungsansätze der Hyperkantenersetzungsgrammatiken und der Separation Logic werden in dieser Arbeit eingehend untersucht und einander gegenübergestellt. Dazu werden sie in Kapitel 2 und 3 zunächst eingeführt und auf ein formales Fundament gestellt. Es werden außerdem Anforderungen an Struktur und Sprache der Grammatiken formalisiert, die ihre Eignung als Abstraktionswerkzeug, sogenannte Heapabstraktionsgrammatiken, sicherstellen.

Um die prinzipiell mächtigere Separation Logic an die Ausdrucksstärke kontextfreier Graphgrammatiken anzupassen, wird im Rahmen dieser Arbeit größtenteils ein syntaktisch und semantisch eingeschränktes Fragment der Logik verwendet. Außerdem wird auf die eingebettete Definition von Prädikaten nach dem Vorbild von Dodds & Plump [DP08b] eingegangen. Kapitel 4 befasst sich mit dem konzeptionellen Vergleich der Ansätze. Dazu werden die Analogien und Differenzen zwischen beiden Methoden herausgearbeitet und verschiedene Lösungen diskutiert, letztere zu überbrücken. Im Anschluss daran wird eine Übersetzung von SL-Formeln in Graphgrammatiken und die Gegenrichtung nach dem Vorbild von Mike Dodds' Dissertation [Dod08], angepasst an die Grammatikmodelle des Juggernaut-Projektes, vorgestellt und deren Korrektheit gezeigt. Eine wichtige Erweiterung zu Dodds ist dabei die Einführung von Programmvariablen, die als Zeiger auf Objekte im Speicher verweisen können. Es dürfen dabei beliebig viele Variablen dasselbe Objekt referenzieren, was die Einführung von Gleichheitsaussagen innerhalb von SL-Assertionen erfordert. Während das Objektmodell von Dodds genau zwei Zeiger als Felder in jedem Objekt vorsieht, wird hier eine beliebige, wenn auch konstante, Anzahl an Feldern erlaubt. Entscheidend ist dabei die Möglichkeit, die Feldinhalte desselben Objektes über verschiedene Prädikate definieren zu

Kapitel 1. Einleitung

dürfen, was über das sogenannte *Field Splitting* realisiert wird.

In Kapitel 5 werden die Anforderungen, die an Grammatiken als Abstraktionswerkzeug gestellt werden, auf Separation-Logic-Prädikate übertragen und gezeigt, dass diese unter der in Kapitel 4 vorgestellten Übersetzung erhalten bleiben. Für die vollständige und korrekte Abstraktion und Konkretisierung von Prädikaten sind diese größtenteils ebenso notwendig, wie für Grammatiken.

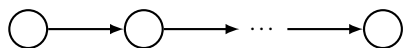
Schließlich geht Kapitel 6 auf die Verifikation von Programmen mit Hilfe von SL-Beweisbäumen ein. Es wird ein Verfahren vorgestellt, mit dem die benötigten Beweisregeln zur Abstraktion und Konkretisierung von Prädikaten aus den dazugehörigen Graphgrammatiken gewonnen werden können. Insbesondere für die Konkretisierung der Vorbedingungen von Spezifikationen reicht die Zugehörigkeit zur Klasse der Heapabstraktionsgrammatiken dabei nicht aus. Um endliche und dennoch vollständige Beweisbäume ableiten zu können, wird auf die lokale Greibach-Normalform [JHKN11] von Jansen zurückgegriffen mit deren Hilfe variablen-spezifisch verkürzte Prädikatdefinitionen erzeugt werden können. Diese erlauben die Ableitung von Beweisregeln zur Konkretisierung einzelner Parameter, sind jedoch zu den ursprünglichen Prädikaten äquivalent, die Beweisbäume also vollständig. Abschließend wird die Verifikation an einem einfachen Beispielprogramm Schritt für Schritt illustriert.

KAPITEL 2

EINFÜHRUNG IN DATENSTRUKTURGRAMMATIKEN

Die Modellierung oder Illustration von Datenstrukturen als gerichtete Graphen ist wahrscheinlich so alt, wie die Idee verketteter Listen selbst. Ein Graph beschreibt dabei stets den Zustand des Speichers beziehungsweise der darin abgelegten Datenstruktur zu einem gewissen Zeitpunkt. Er hat eine definierte Anzahl an Knoten und Kanten und die durch ihn dargestellte Datenstruktur damit eine feste Größe. Betrachtet man dynamische Datenstrukturen wie Listen oder Bäume, ist diese jedoch nicht vorgegeben und in vielen Fällen auch nicht von Interesse. Bei der Modellierung durch Graphen wird dies oft durch „...“ verdeutlicht, wie in der folgenden Darstellung zu sehen:

Beispiel 2.1.



△

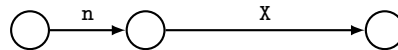
Wie der durch „...“ dargestellte Teil des Graphen genau auszusehen hat, ist dabei nicht formal definiert, sondern für gewöhnlich der Intuition des Lesers überlassen. Eine ähnliche Notation wird auch für Strings verwendet, z.B. `abab...ab`.

Um eine Klasse von Strings variabler Länge kompakt und formel darstellen zu können, gibt es verschiedene Ansätze, wie Wildcards, reguläre Ausdrücke oder kontextfreie und -sensitive Grammatiken, deren Mächtigkeit in dieser Reihenfolge zunimmt. Alle diese Werkzeuge stellen eine Möglichkeit der Spezifikation von Wortmengen dar, die bestimmten Mustern genügen [Woo87].

Graphgrammatiken sind ein zu Stringgrammatiken analoges Konzept für Graphsprachen. Sie erlauben – genau wie jene – die endliche Darstellung einer möglicherweise unendlichen Menge von wiederum endlichen Graphen.

Zur sinnvollen Darstellung einer Datenstruktur als Graph erhält jede Kante ein sogenanntes Label, das ihren Bezug zur Datenstruktur repräsentiert. Dabei handelt es sich gewöhnlicherweise um den Namen eines Zeigers, der von einem Objekt auf das nächste verweist. Um die kompakte Darstellung von Graphteilen auf ein formales Fundament zu stellen, führen wir außerdem eine Klasse von Platzhalter-Labels ein, die wir später als Nichtterminale bezeichnen werden. Im Gegensatz zu „...“ repräsentiert eine Kante, die mit einem solchen Platzhalter versehen ist, einen formal spezifizierten Graphenteil. Der Graph aus Beispiel 2.1 sieht unter Verwendung der Labels n für den Zeiger auf das jeweils nächste Element und X für den Platzhalter wie folgt aus:

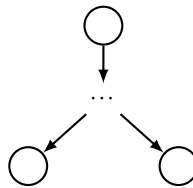
Beispiel 2.2.



△

Der fundamentale Unterschied zwischen Strings und Graphen ist die fehlende Linearität. Während in einem String jedes Zeichen maximal einen Vorgänger und einen Nachfolger hat und es genau ein Zeichen ohne Vorgänger und eines ohne Nachfolger gibt, mag dies für Listen zwar noch gelten, für komplexere Datenstrukturen, z.B. Bäume, jedoch nicht mehr.

Beispiel 2.3.



△

Um einen Teilgraphen, der über mehr als zwei Kanten mit seiner Umgebung verbunden ist (wie „ \dots “ im obigen Bild), ebenfalls durch einen Platzhalter darstellen zu können, gibt es im Wesentlichen zwei Möglichkeiten. Die erste besteht in der Verwendung von Knoten anstelle von Kanten als Platzhalter [Eng97]. Ein solcher Knoten könnte mit beliebig vielen Kanten des umliegenden Graphen verbunden sein. Die zweite Methode führt Kanten höheren Grades (sog. Hyperkanten) ein, die mehr als zwei Knoten miteinander verbinden können. In dieser Arbeit wird ausschließlich der Hyperkanten-Ansatz betrachtet.

2.1. HYPERGRAPHEN UND HEAPKONFIGURATIONEN

Im Gegensatz zu einer (einfachen) Kante, ist eine Hyperkante kein Tupel zweier Knoten, sondern ein eigenständiges Objekt, dessen Inzidenzen über eine Funktion definiert werden. Ein Graph, der Hyperkanten enthält, wird Hypergraph genannt. Die hier verwendeten Hypergraphen verfügen ausschließlich über Hyperkanten. „Normale“ Kanten werden daher als Hyperkanten von Grad zwei (also mit zwei inzidenten Knoten) dargestellt.

Diese Arbeit basiert auf den Forschungsergebnissen von Heinen et al., insbesondere auf [HJKN12].

DEFINITION 2.1 (Hypergraph) Sei Σ_N ein endliches Alphabet und $\text{rk} : \Sigma_N \rightarrow \mathbb{N}$ eine Funktion, die jedem Symbol $Y \in \Sigma_N$ einen Rang $\text{rk}(Y)$ zuweist. Ein *Hypergraph* H über Σ_N ist ein Quintupel

$$H = (V, E, \text{att}, \text{lab}, \text{ext})$$

mit einer endlichen Knotenmenge V , einer endlichen Menge an Hyperkanten E , einer Inzidenzfunktion $\text{att} : E \rightarrow V^*$, die jeder Hyperkante eine endliche Sequenz inzidenter Knoten zuweist, einer Beschriftungsfunktion $\text{lab} : E \rightarrow \Sigma_N$, die jeder Hyperkante ein Label zuweist, sowie einer Sequenz paarweise verschiedener externer Knoten $\text{ext} \in V^*$. Für jede Hyperkante $e \in E$ fordern wir $|\text{att}(e)| = \text{rk}(\text{lab}(e))$. Desweiteren schreiben wir abkürzend $\text{rk}(e)$ für $\text{rk}(\text{lab}(e))$. rk und lab seien darüber hinaus auch elementweise für Mengen von Knoten

2.1. Hypergraphen und Heapkonfigurationen

bzw. Symbolen definiert. Mit $V_H, E_H, \text{att}_H, \text{lab}_H$ und ext_H notieren wir abkürzend die entsprechenden Elemente des Hypergraphen H . HG_{Σ_N} stellt die Menge aller Hypergraphen über Σ_N dar.

Wir bezeichnen zwei Hypergraphen als isomorph, wenn sie bis auf Umbenennungen von Kanten und Knoten identisch sind. Da in dieser Arbeit ausschließlich Hypergraphen nach der obigen Definition betrachtet werden, werden die Begriffe *Graph* und *Hypergraph* synonym verwendet. Analoges gilt für *Kanten* und *Hyperkanten*.

Die folgende Definition stellt Hilfsfunktionen für die Ordnung und den Zugriff von Mengen und Sequenzen bereit.

DEFINITION 2.2 ($[\cdot], \langle \cdot \rangle, \text{cn}$) Sei D eine beliebige, abzählbare Domäne.

Die *Mengenfunktion* $[\cdot]$ ist definiert durch

$$[\cdot] : D^* \rightarrow 2^D, \quad [d_1 \dots d_n] := \{d_1, \dots, d_n\}$$

Die *Ordnungsfunktion* $\langle \cdot \rangle$ ist definiert durch

$$\langle \cdot \rangle : 2^D \rightarrow D^*, \quad \langle \{d_1, \dots, d_n\} \rangle := d_{i_1} \dots d_{i_n}$$

wobei i_j den Index des j -ten Elements der Menge $\{d_1, \dots, d_n\}$ nach einer beliebigen, aber festen kanonischen Ordnungsrelation darstellt.

Sei $M \subseteq D$. Die Funktion $\text{cn} : M \rightarrow \mathbb{N}^0$ gibt den nullbasierten Index jedes Elements $m_j \in M$ zurück:

$$\text{cn}(m_j) := i_j - 1$$

Auf das j -te Element einer Sequenz $\bar{d} = d_1 d_2 \dots d_n \in D^*$ greifen wir mit (\cdot) zu:

$$\bar{d}(j) := d_j$$

Ferner gilt: $[\varepsilon] = \emptyset$ und $\langle \emptyset \rangle = \varepsilon$.

Analog zur Unterscheidung zwischen Start- und Endknoten für normale, gerichtete Kanten, werden auch die an eine Hyperkante angeschlossenen Knoten unterschieden. So ist es bei einer verketteten Liste zum Beispiel wichtig, welcher Knoten der Vorgänger und welcher der Nachfolger eines durch eine Nichtterminalkante dargestellten Listensegments ist. Da die jeweilige Rolle der Knoten direkt vom Label der Kante abhängt (Vorgänger und Nachfolger beispielsweise haben für alle Kanten, die eine Liste repräsentieren, die gleiche Bedeutung), ist es sinnvoll, die Unterscheidung zwischen den Rollen des ersten, zweiten, ... angeschlossenen Knotens einer Kante direkt für das Label, anstatt für jede Kante einzeln, zu definieren.

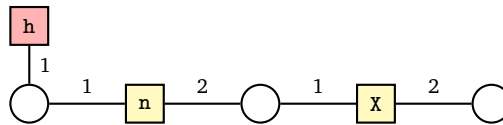
DEFINITION 2.3 (Tentakel) Seien $X \in \Sigma_N$ und $1 \leq i \leq \text{rk}(X)$. Das Paar (X, i) ist ein *Tentakel*.

Der Begriff des Tentakels erlaubt uns, Aussagen wie *Der Knoten am ersten Tentakel von X repräsentiert den Vorgänger, der am Zweiten den Nachfolger*, zu treffen. Die Funktion att bildet Kanten auf Sequenzen anstatt auf Mengen von Knoten ab und ordnet auf diese Weise die inzidenten Knoten der Kante den Tentakeln ihres Labels zu. Vor allem bei der im folgenden Abschnitt eingeführten Hyperkantenersetzung spielen Tentakel in Verbindung mit externen Knoten eine zentrale Rolle.

Kapitel 2. Einführung in Datenstrukturgrammatiken

Um Datenstrukturen zu modellieren, unterteilen wir das Label-Alphabet Σ_N zunächst in eine Menge von *Terminalsymbolen* Σ und eine Menge von *Nichtterminalsymbolen* N , so dass $\Sigma_N = \Sigma \uplus N$. Die Terminalsymbole gliedern sich dabei weiter in *Selektoren* sel_Σ und *Variablen* var_Σ : $\Sigma = \text{sel}_\Sigma \uplus \text{var}_\Sigma$. Intuitiv repräsentiert ein Selektor ein Feld in einem Objekt. Im Beispiel der Liste ist ein Selektor n für den Zeiger „next“ naheliegend. Der Nachfolger eines Knotens würde dann durch eine Kante mit dem Label n auf den entsprechend nächsten Knoten dargestellt. Kanten mit Labels aus sel_Σ werden auch als *Selektorkanten* bezeichnet. Variablen werden als Labels von Rang-1-Kanten (also Kanten $e \in E$ mit $\text{rk}(e) = 1$) verwendet, um einen Knoten global zu markieren. Bei der Analyse eines Programms werden diese Kanten außerdem verwendet, um die Programmvariablen, also die Stellen der Datenstruktur, auf die das Programm direkt zugreifen kann, zu modellieren. Wir bezeichnen sie im Folgenden als *Variablenkanten*.

Beispiel 2.4. Für die Liste ist eine Variable h für „head“ als Label einer Kante, die am ersten Knoten dieser Liste anliegt, sinnvoll. Durch die Einführung von Hyperkanten ergibt sich aus Beispiel 2.2 der Hypergraph:

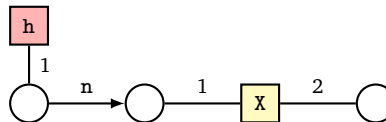


△

Die beschrifteten Quadrate visualisieren dabei die Hyperkanten mit ihren Labels und eine mit j beschriftete Linie an einer Kante e den j -ten Eintrag in $\text{att}(e)$.

Der Übersicht halber werden Selektorkanten üblicherweise weiterhin als Pfeil dargestellt (siehe Beispiel 2.5), der bei $\text{att}(e)(1)$ beginnt und auf $\text{att}(e)(2)$ zeigt. Ersteren Knoten bezeichnen wir dabei als *Quell-* und letzteren als *Zielknoten*. Wir nennen die Kante e aus ihrem Quellknoten *ausgehend* und an ihrem Zielknoten *eingehend*.

Beispiel 2.5.



△

Externe Knoten sind in den Abbildungen durch eine Zahl innerhalb des Knotens gekennzeichnet, die seine Position (bei 1 beginnend) innerhalb der Sequenz ext widerspiegelt. In Beispiel 2.6 ist ein externer Knoten zu sehen.

DEFINITION 2.4 (Ausgehende Kanten) Seien $H \in \text{HC}_\Sigma$ und $v \in V_H$. Die Menge *ausgehender Kanten* am Knoten v ist definiert als $\text{out}(v) := \{e \in E_H \mid \text{att}(e)(1) = v\}$.

Wir schränken nun die Menge der Hypergraphen auf diejenigen ein, die eine sinnvolle Datenstruktur repräsentieren und bezeichnen sie als *Heapkonfigurationen*. Dabei handelt es sich um ein Hypergraph ohne externe Knoten, bei dem alle mit einer Variable gekennzeichneten Kanten den Rang 1 und die mit einem Selektor gekennzeichneten den Rang 2 haben, es jede Variable höchstens einmal gibt und jeder Selektor jeden Knoten höchstens einmal verlässt.

2.2. Hyperkanteneretzungs- und Datenstrukturgrammatiken

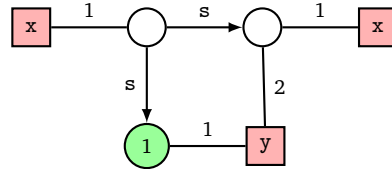
DEFINITION 2.5 (Heapkonfiguration) Eine *Heapkonfiguration* ist ein Hypergraph $H = (V, E, \text{att}, \text{lab}, \text{ext}) \in \text{HG}_{\Sigma_N}$ für den gilt

- $\text{rk}(\text{lab}(E) \cap \text{var}_{\Sigma}) = \{1\}$ und $\text{rk}(\text{lab}(E) \cap \text{sel}_{\Sigma}) = \{2\}$,
- für alle $x \in \text{var}_{\Sigma}$ gilt $|\{e \in E \mid \text{lab}(e) = x\}| \leq 1$
- für alle $v \in V$, $s \in \text{sel}_{\Sigma}$ gilt $|\{e \in E \mid \text{lab}(e) = s, \text{att}(e)(1) = v\}| \leq 1$ und
- $\text{ext} = \varepsilon$

Wenn $\text{lab}(E) \cap N = \emptyset$ gilt, bezeichnen wir H als *konkret*, andernfalls als *abstrakt*. Die Menge aller Heapkonfigurationen wird mit HC_{Σ_N} , die aller konkreten Heapkonfigurationen entsprechend mit HC_{Σ} bezeichnet.

Ein Hypergraph $H \in \text{HG}_{\Sigma_N} \setminus \text{HC}_{\Sigma_N}$ kann offensichtlich keine existierende Datenstruktur-Instanz beschreiben. Das mehrfache Auftreten derselben Variable ist dabei ebensowenig sinnvoll wie das Vorhandensein zweier Felder mit demselben Namen innerhalb eines Objekts. Die Rang-Beschränkungen ergeben sich aus der Tatsache, dass sowohl Variablen als auch Selektoren stets auf genau ein Objekt verweisen und Letztere auch Teil genau eines Objektes sind. Externe Knoten sind lediglich für die Hyperkanteneretzung erforderlich und haben für die Instanzen einer Datenstruktur keine Bedeutung.

Beispiel 2.6. Seien $x, y \in \text{var}_{\Sigma}$ und $s \in \text{sel}_{\Sigma}$. Der folgende Graph ist keine HC, da er mehrere Eigenschaften verletzt: Die Variable x existiert doppelt, die Variable y hat den Rang 2, der Selektor s geht aus dem linken, oberen Knoten doppelt aus und es gibt einen externen Knoten (links unten).



△

2.2. HYPERKANTENERSETZUNGS- UND DATENSTRUKTURGRAMMATIKEN

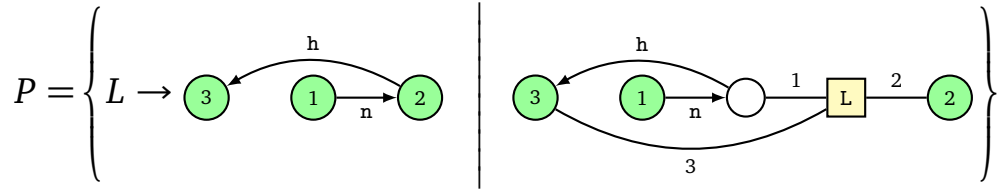
Um Nichtterminalen und damit auch Nichtterminalkanten eine Bedeutung zuzuweisen, bedarf es Ableitungsregeln.

DEFINITION 2.6 (Regelsatz) Ein *Regelsatz* ist eine Menge P von Produktionsregeln $\pi \in N \times \text{HG}_{\Sigma_N}$. Einzelne Regeln $(X, H) \in N \times \text{HG}_{\Sigma_N}$ notieren wir mit $X \rightarrow H$, wobei stets $|\text{ext}_H| = \text{rk}(X)$ gelten muss. Die Menge aller Regelsätze über Σ_N bezeichnen wir mit HRR_{Σ_N} (*Hyperedge Replacement Rules*).

Beispiel 2.7. Ein Regelsatz für eine verkettete Liste, in der jedes (außer dem ersten) Element

Kapitel 2. Einführung in Datenstrukturgrammatiken

einen Zeiger h auf den Kopf der Liste hat ($s, h \in \text{sel}_\Sigma$, $L \in N$, $\text{rk}(L) = 3$):



Δ

Eine Produktionsregel $X \rightarrow K$ transformiert $H \in \text{HG}_{\Sigma_N}$ in einen neuen Hypergraphen H' , indem sie eine Nichtterminalkante $e \in E_H$ mit $\text{lab}(e) = X$ durch den Graphen K ersetzt. Die externen Knoten von K werden dabei mit den zu e inzidenten Knoten verschmolzen (für $1 \leq j \leq \text{rk}(X)$ jeweils $\text{ext}_K(j)$ mit $\text{att}_H(e)(j)$). Die folgende Definition formalisiert diesen Sachverhalt.

DEFINITION 2.7 (Hyperkantenersetzung) Seien $H, K \in \text{HG}_{\Sigma_N}$ und $e \in E_H$, so dass $\text{rk}(e) = |\text{ext}_K|$. Wir nehmen o. B. d. A. an, dass $V_H \cap V_K = E_H \cap E_K = \emptyset$. Die Ersetzung von e durch K , notiert durch $H[K/e]$, ist der Hypergraph $H' \in \text{HG}_{\Sigma_N}$ mit:

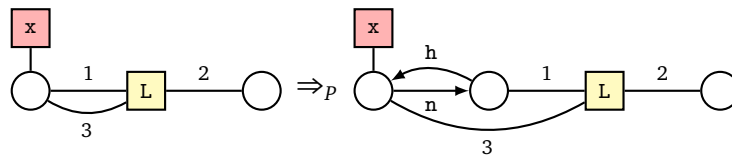
$$\begin{aligned} V_{H'} &= V_H \cup (V_K \setminus [\text{ext}_K]) & E_{H'} &= (E_H \setminus \{e\}) \cup E_K \\ \text{lab}_{H'} &= (\text{lab}_H \upharpoonright (E_H \setminus \{e\})) \cup \text{lab}_K & \text{ext}_{H'} &= \text{ext}_H \\ \text{att}_{H'} &= \text{att}_H \upharpoonright (E_H \setminus \{e\}) \cup (\text{mod} \circ \text{att}_K) \end{aligned}$$

und $\text{mod} = \text{id}_{V_{H'}} \cup \{[\text{ext}_K(1) \mapsto \text{att}_H(e)(1), \dots, \text{ext}_K(\text{rk}(e)) \mapsto \text{att}_H(e)(\text{rk}(e))]\}$.

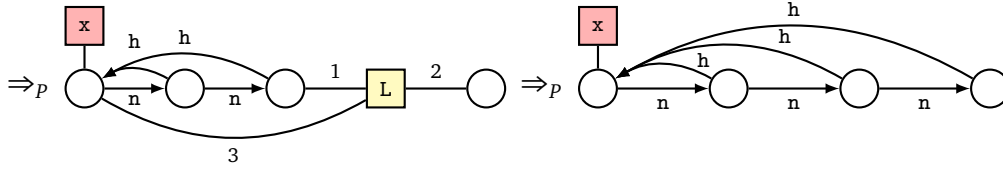
Der Graph wird also durch die Knotenmenge V_K unter Ausschluss der externen Knoten ergänzt und e wird gelöscht. Die Funktion mod ersetzt die externen Knoten von K durch die inzidenten Knoten von e . Schließlich werden die Inzidenzfunktionen att_H und att_K zusammengeführt. Die folgende Definition führt den Begriff der Ableitung unter Verwendung der Hyperkantenersetzung ein.

DEFINITION 2.8 (Ableitung) Seien $P \in \text{HRR}_{\Sigma_N}$, $H, H' \in \text{HG}_{\Sigma_N}$, $\pi = X \rightarrow K \in P$ und $e \in E_H$, so dass $\text{lab}_H(e) = X$. H' wird von H durch π *abgeleitet*, wenn H' zu $H[K/e]$ isomorph ist. Wir notieren dies mit $H \Rightarrow_{e,p} H'$. Ferner gilt $H \Rightarrow_p H'$, wenn es $e \in E_H$ und $\pi \in P$ gibt, so dass $H \Rightarrow_{e,p} H'$ gilt. Wenn P klar aus dem Kontext hervorgeht, schreiben wir auch vereinfacht $H \Rightarrow H'$. Die Relation \Rightarrow^* beschreibt die reflexive und transitive, abgeschlossene Hülle von \Rightarrow .

Beispiel 2.8. Mögliche Ableitung ausgehend von einem gegebenen Graphen unter P (Beispiel 2.7):



2.2. Hyperkantenersetzungs- und Datenstrukturgrammatiken



Zunächst wird zweimal die zweite Produktionsregel angewandt und anschließend einmal die erste. Der letzte Graph ist konkret, es können also keine weiteren Ableitungsschritte mehr erfolgen. \triangle

Heinen et al. führen an dieser Stelle den Terminus der Sprache als Abbildung von Graphen auf eine Menge konkreter Graphen ein. Dieser passt allerdings nur bedingt zu dem Verständnis einer Sprache, das die Zuordnung einer ableitbaren konkreten Graphmenge zu einer Grammatik beschreibt und das wir in dieser Arbeit zugrunde legen. Tatsächlich sind beide Auffassungen sehr eng miteinander verwandt, schließlich unterscheiden sie sich lediglich in der freien oder durch die Grammatik vorgegebenen Wahl der Ausgangsgraphen, aus denen mit Hilfe eines Regelsatzes abgeleitet wird. Um die eingeführten Begriffe eindeutig und konsistent zu verwenden und um Verwirrung zu vermeiden, sprechen wir im ersteren Falle von der *konkreten Hülle* eines Graphen. Auf die Sprache einer Grammatik wird in Definition 2.11 eingegangen.

DEFINITION 2.9 (konkrete Hülle) Sei $P \in \text{HRR}_{\Sigma_N}$. Die *konkrete Hülle* eines Hypergraphen $H \in \text{HG}_{\Sigma_N}$ unter P ist definiert als

$$L_P(H) = \{H' \in \text{HG}_{\Sigma} \mid H \Rightarrow^* H'\}$$

Die konkrete Hülle eines Graphen ist die Menge aller aus ihm ableitbaren konkreten Graphen. Damit eine Grammatik eine eindeutige Graphsprache beschreibt, ist zusätzlich zum Regelsatz noch die Definition der Ausgangspunkte für die Ableitungen notwendig. Während String-Grammatiken üblicherweise ein Start-(Nichtterminal)-Symbol festlegen, gehen wir hier von einer Menge vorgegebener Startgraphen aus.

DEFINITION 2.10 (Hyperkantenersetzungsgrammatik) Eine *Hyperkantenersetzungsgrammatik* ist ein Paar $G = \langle \mathfrak{J}, P \rangle$ aus einer *Startgraphmenge* $\mathfrak{J} \subseteq \text{HG}_{\Sigma_N}$ und einem Regelsatz $P \in \text{HRR}_{\Sigma_N}$.

Die Menge aller Hyperkantenersetzungsgrammatiken über Σ_N bezeichnen wir mit HRG_{Σ_N} .

Die Mengen sel_{Σ} , var_{Σ} und damit auch Σ werden stets als von außen vorgegeben betrachtet. Dies ist vor allem bei der Grammatiksynthese, wie sie in Abschnitt 4.2 beschrieben wird, von großer Bedeutung, da diese im vorgestellten Ansatz auf einem festen Alphabet operiert. Da die Mächtigkeit der Nichtterminalmenge N für die Synthese und Formelgenerierung nach oben nicht beschränkt sein muss, fassen wir aus Konsistenzgründen beide Mengen, Σ und N als global definiert auf. Sie sind insbesondere weder Teil eines Hypergraphen noch einer Grammatik. Die Teilmenge der Nichtterminale, die auf den linken Regelseiten einer Grammatik $G = \langle \mathfrak{J}, P \rangle$ auftreten, notieren wir mit $N_G := \{X \in N \mid \exists X \rightarrow H \in P\}$. Basierend auf den Definitionen der konkreten Hülle sowie der HRGen führen wir den Begriff der Sprache einer Grammatik ein.

DEFINITION 2.11 (Sprache) Die Sprache einer Grammatik $G = \langle \mathfrak{J}, P \rangle \in \text{HRG}_{\Sigma_N}$ ist definiert als die konkrete Hülle aller ihrer Startgraphen $H \in \mathfrak{J}$ unter ihrem Regelsatz P .

$$\mathcal{L}(G) := \bigcup_{H \in \mathfrak{J}} L_P(H).$$

Um Aussagen über den Regelsatz einer Grammatik G zu treffen, wird zumeist die konkrete Hülle der linken Regelseiten, also der Nichtterminale N_G , verwendet. Bei Stringgrammatiken stellt ein einzelnes Nichtterminal bereits ein Wort da, vom dem weitere (Terminal-)Wörter abgeleitet werden können. Im Kontext von Graphgrammatiken sind Nichtterminale jedoch zunächst keine Graphen. Aus syntaktischen Gründen führen wir daher das *Handle*, einen Graphen, der ausschließlich aus einer Nichtterminalkante und externen Knoten besteht, ein. Das Handle einer linken Regelseite ist dann ein vollwertiger Graph, so dass die konkrete Hülle darauf definiert ist.

DEFINITION 2.12 (Handle) Für $X \in N$ mit $\text{rk}(X) = n$ ist ein X -Handle definiert durch

$$X^\bullet := \left(\underbrace{\{v_1, \dots, v_n\}}_V, \underbrace{\{e\}}_E, \underbrace{[e \mapsto v_1 \dots v_n]}_{\text{att}}, \underbrace{[e \mapsto X]}_{\text{lab}}, \underbrace{v_1 \dots v_n}_{\text{ext}} \right) \in \text{HG}_{\Sigma_N}$$

Für die Modellierung von Datenstrukturen sind insbesondere solche Grammatiken interessant, deren Sprache ausschließlich Heapkonfigurationen, also den Speicher sinnvoll abbildende Graphen, enthalten. Darüber hinaus sollen auch die Graphen in der konkreten Hüllen solcher Nichtterminale, die nicht in Startgraphen verwendet werden, HCs sein.

DEFINITION 2.13 (Datenstrukturgrammatik) Eine Grammatik $G = \langle \mathfrak{J}, P \rangle \in \text{HRG}_{\Sigma_N}$ heißt *Datenstrukturgrammatik*, wenn $\mathcal{L}(G) \subseteq \text{HC}_{\Sigma_N}$ und für alle $X \in N$ gilt, dass $L_P(X^\bullet) \subseteq \text{HC}_{\Sigma}$, also die Sprache und die konkreten Hüllen aller Nichtterminale ausschließlich Heapkonfigurationen enthalten.

Die Menge aller Datenstrukturgrammatiken über Σ_N wird mit DSG_{Σ_N} bezeichnet. Die Menge der Regelsätze, auf die dies zutrifft bezeichnen wir mit DSR_{Σ_N} .

SATZ 2.1 Die DSG-Eigenschaft ist entscheidbar.

Beweis. Die Zugehörigkeit von \mathfrak{J} zu HC_{Σ} kann durch Iteration über alle Knoten und Kanten entschieden werden. Die Zugehörigkeit von P zu DSR_{Σ_N} ist entscheidbar nach [HJKN12], Anhang A. □

2.3. ABSTRAKTION UND KONKRETISIERUNG

Hyperkantenersetzungsgrammatiken ermöglichen eine Vielzahl verschiedener Anwendungen. Die wohl Elementarste ist die Beschreibung einer Klasse von Hypergraphen, die mittels \mathcal{L} definiert ist. Dazu bedarf es beider Elemente der Grammatik – der Startgraphen und des Regelsatzes. Die Sprache einer gegebenen Grammatik ist fest definiert. Der Einsatz von Grammatiken zur Beschreibung von Graphklassen ist also in gewisser Weise eine statische Anwendung. Es geht dabei hauptsächlich um die Ableitbarkeit eines Graphen und weniger um die Ableitungen selbst.

Eine andere Verwendung von Graphgrammatiken findet sich im Juggernaut-Projekt von Heinen et al. [HJKN12]. Hier werden DSGen nicht zur statischen Beschreibung einer Graphmenge sondern als Werkzeug zur Transformation, genauer gesagt zur Abstraktion und Konkretisierung von Hypergraphen verwendet.

2.3.1. ABSTRAKTION UND KONKRETISIERUNG DURCH GRAMMATIKEN

Wie schon zu Beginn dieses Kapitels erläutert, können Hypergraphen beziehungsweise Heapkonfigurationen den Zustand einer verketteten Datenstruktur modellieren. Die Knoten des Graphen repräsentieren dabei die einzelnen Objekte, während Selektorkanten die Zeiger innerhalb der Objekte verkörpern. Eine dynamische Datenstruktur kann im Allgemeinen beliebige Größe erreichen. Ein Programm, das eine bestimmte Datenstruktur, beispielsweise einen binären Baum, verwendet, hat daher einen potenziell unendlichen Zustandsraum. Dies stellt für gängige formale Verifikationsmethoden wie das Model Checking ein großes Problem dar, weil das Programm durch kein endliches Transitionssystem beschrieben werden kann [BK08].

In viele Fällen ist jedoch nicht die gesamte Datenstruktur sondern nur eine endliche Menge von Knoten interessant. Werden diese Knoten beliebig gewählt, lässt sich daraus auf die Korrektheit des Programms für die gesamte Datenstruktur schließen.

Beispiel 2.9. Als Beispiel sei hier ein Programm zum Invertieren einer Liste genannt. Wenn für jede zwei beliebig ausgewählten Objekte a_1, a_2 vor der Ausführung $a_1.n = a_2$ und anschließend $a_2.n = a_1$ gilt, ist das Programm für die gesamte Liste korrekt [HJKN12]. Δ

Zur Markierung dieser Objekte werden die oben eingeführten Variablenkanten verwendet. Betrachtet man eine atomare Programmanweisung, so stellt man fest, dass ihre Ausführung den Heap nur in einem relativ kleinen Bereich ausliest oder ändert. Die genaue Beschaffenheit des restlichen Heaps hingegen spielt keine Rolle. Der unendliche Zustandsraum lässt sich daher in eine endliche Menge an Äquivalenzklassen gliedern, so dass die Heaps, die sich nur an den für die Anweisungen irrelevanten Stellen unterscheiden, in jeweils derselben Äquivalenzklasse liegen. Übertragen auf Graphen beziehungsweise Heapkonfigurationen bedeutet dies, dass diejenigen Teile der HC, die von der aktuellen Anweisung nicht berührt werden, durch Nichtterminalkanten abstrakt dargestellt werden können. Der unendliche Zustandsraum kollabiert dadurch zu einem endlichen, durch deren Hilfe alle für die Anweisung äquivalenten Zustände des Graphen abgedeckt werden.

Ein Programm arbeitet gewöhnlicherweise nicht nur auf einem eingeschränkten Teil sondern auf der gesamten Datenstruktur. Daher reicht die abstrakte Modellierung eines Heapteils alleine nicht aus. Stattdessen werden DSGen verwendet, um die durch Nichtterminalkanten abstrakt dargestellten Teile der HC durch Anwendung ihrer Produktionsregeln sukzessive abzuleiten (Konkretisierung) und konkrete Heapteile auch wieder durch Nichtterminalkanten zu substituieren (Abstraktion), damit der Zustandsraum endlich bleibt. Anstelle von konkreten HGen werden also abstrakte HGen als Programmzustände betrachtet. Bei einer Traversierung der Datenstruktur durch eine Zeigervariable wird die Heapkonfiguration bei jedem Schritt unmittelbar vor dem Zeiger konkretisiert und dahinter wieder abstrahiert. Konkretisieren bedeutet dabei die Eliminierung einer Nichtterminalkante durch Ableitung. Bei der Abstraktion werden Subgraphen, die einer rechten Regelseite entsprechen, durch

eine Nichtterminalkante mit dem Label auf der linken Regelseite ersetzt.

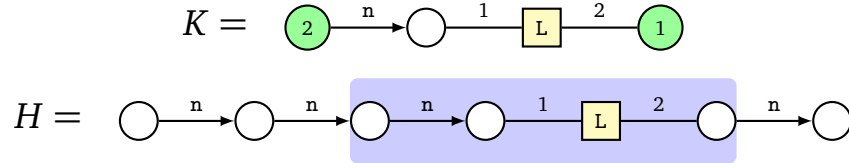
2.3.2. EINSATZ BEI DER PROGRAMMANALYSE

Zur Eingrenzung der DSGen auf HAGs bedarf es zunächst einiger weiterer Betrachtungen. Das folgende Lemma aus [HJKN12] sichert die Genauigkeitserhaltung bei der Konkretisierung.

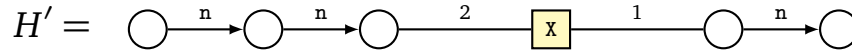
LEMMA 2.2 (Genauigkeitserhaltung der Konkretisierung) Seien $G = \langle \mathfrak{Z}, P \rangle \in \text{DSG}_{\Sigma_N}$ und $H \in \text{HC}_{\Sigma_N}$. Dann gilt für alle $H' \in \text{HC}_{\Sigma_N}$ mit $H \Rightarrow_p^* H'$, dass $L_p(H') \subseteq L_p(H)$.

Die Abstraktion verlangt die Rückwärtsanwendung einer Produktionsregel. Dazu muss zunächst eine rechte Regelseite K innerhalb eines Graphen H identifiziert werden. Man spricht hier von der Projektion [DP08b] oder der Einbettung [HJKN12] von K in H . Beide Begriffe meinen dabei im Kern denselben Sachverhalt: In H befindet sich eine Teilmenge von Knoten und Kanten, so dass diese für sich genommen zu K isomorph sind. An den Knoten des Subgraphen, die den internen Knoten von K entsprechen dürfen darüber hinaus keine weiteren Kanten anliegen. Außerdem müssen diese Knoten in H ebenfalls intern sein, damit im Subgraphen von H keine Kanten erzeugt werden können, die nicht auch in K ableitbar sind. Eine formale Definition der Einbettung findet sich in [HJKN12], Definition 13. Die Abstraktion selbst besteht in der Ersetzung der Projektion einer rechten Regelseite durch eine Hyperkante. Die Inzidenzen der Kante sind dabei die externen Knoten der verwendeten rechten Regelseite beziehungsweise deren Entsprechung in der Projektion.

Beispiel 2.10. Projektion des Graphen K im Graphen H (blau unterlegt):



Enthält die verwendete Grammatik beispielsweise eine Regel $X \rightarrow K \in P$, dann kann die Projektion von K durch Rückwärtsanwendung dieser Regel zu einer X -Kante abstrahiert werden:



△

DEFINITION 2.14 (Vollständige Abstraktion) Für $G = \langle \mathfrak{Z}, P \rangle \in \text{DSG}_{\Sigma_N}$ und $H \in \text{HC}_{\Sigma_N}$ ist die *vollständige Heapabstraktion* definiert als mengenwertige Funktion:

$$\text{fullAbstr}_G(H) = \{K \in \text{HC}_{\Sigma_N} \mid K \Rightarrow_p^* H \wedge \forall X \rightarrow K' \in P : \text{es ex. keine Proj. von } K' \text{ in } K\}$$

Je nach Beschaffenheit der Grammatik ist es durchaus möglich, dass ein Graph über mehrere vollständige Abstraktionen verfügt. Ist $\text{fullAbstr}_G(H)$ für alle $H \in \text{HC}_{\Sigma_N}$ einelementig, nennen wir G *rückwärtig konfluent*.

SATZ 2.3 Es ist entscheidbar, ob eine HRG G rückwärtig konfluent ist.

Beweis. Siehe [HJKN12], Theorem 3. □

Für die folgenden Abschnitte werden noch die Termini des *Reduktionstentakels* und der *Zulässigkeit* eingeführt.

DEFINITION 2.15 (Reduktionstentakel) Der Tentakel (X, j) heißt *Reduktionstentakel*, wenn für alle $H \in L(X^\bullet)$ gilt, dass $\text{out}(\text{ext}_H(j)) = \emptyset$.

Ein Reduktionstentakel erzeugt also ausschließlich solche Terminalkanten, die in den daran inzidenten Knoten eingehen, den durch die Nichtterminalkante eingesetzten Graphen also verlassen.

Damit der Verifikationsalgorithmus ordnungsgemäß funktioniert, muss zu jedem Zeitpunkt sichergestellt sein, dass das Programm nur auf die Bereiche des Heaps zugreift, die im Graphen gerade nicht abstrakt dargestellt sind. Die maximale Dereferenzierungstiefe der Programmiersprache wird dazu ohne Einschränkung auf 1 festgelegt. Ein Zugriff auf einen Selektor s eines in einer Variablen hinterlegten Objektes ($x.s$) ist also erlaubt, ein tieferer Zugriff, wie $x.s.s$, jedoch nicht. Auf die Mächtigkeit der Sprache hat dies keinen Einfluss, da der Zugriff $x.s.s$ durch $y := x.s; y.s$ dargestellt werden kann. Zugriffe größerer Tiefe können analog gehandhabt werden. Der Vorteil der Verteilung eines tiefen Zugriffs auf mehrere Zugriffe und Zuweisungen liegt in der Ermöglichung zwischenzeitlich ausgeführter Konkretisierungsschritte.

Für jede atomare Anweisung des Programms ist dadurch sichergestellt, dass es ausschließlich mit denjenigen Knoten arbeitet, die entweder selbst inzident zu einer Variablenkante sind, oder die eine eingehende Selektorkante von einem solchen Knoten haben. Der Algorithmus muss also dafür sorgen, dass jeder Knoten, der eine inzidente Variablenkante hat, über keine inzidente Nichtterminalkante, also benachbarte abstrakte Subgraphen, verfügt. Eine Heapkonfiguration, die diese Anforderung erfüllt, nennen wir *zulässig (admissible)*. Formal lässt sich dies über Verletzungspunkte definieren.

DEFINITION 2.16 (Verletzungspunkt, Zulässigkeit) Sei $H \in \text{HC}_{\Sigma_N}$. Das Paar $(e, j) \in E_H \times \mathbb{N}$ mit $1 \leq j \leq \text{rk}(\text{lab}(e))$ und $\text{lab}(e) \in N$ heißt *Verletzungspunkt*, wenn $(\text{lab}(e), j)$ kein Reduktionstentakel ist und es eine Variablenkante $e' \in E_H$ gibt, so dass $\text{att}(e')(1) = \text{att}(e)(1)$. Eine HC ist *zulässig*, wenn sie frei von Verletzungspunkten ist. Die Menge aller zulässigen HGen notieren wir mit AHC_{Σ_N} , die Menge aller Verletzungspunkte in H mit VP_H .

2.3.3. HEAPABSTRAKTIONSGRAMMATIKEN

Nicht jede DSG ist zur Konkretisierung und Abstraktion geeignet. Es müssen weitere Anforderungen erfüllt werden, was schließlich zur Klasse der Heapabstraktionsgrammatiken (HAG) führt. Die Startgraphen spielen dabei offensichtlich keine Rolle. Alle Einschränkungen beziehen sich daher auf die Regelsätze der Grammatiken. Tatsächlich wird die Startgraphmenge \mathfrak{J} in HAGs nicht einmal benötigt. Um die Begrifflichkeiten der gegenwärtigen Literatur beizubehalten, sprechen wir dennoch von Grammatiken anstelle von Regelsätzen als Abstraktions- und Konkretisierungswerkzeug. Zur konsistenten Verwendung des Terminus Grammatik, behalten wir die Startgraphmenge \mathfrak{J} auch für HAGen bei, setzen sie jedoch als leer voraus und ignorieren sie im weiteren Vorgehen.

Die folgenden Unterabschnitte definieren die vier Anforderungen an Grammatiken, die zur Verwendung als Heapabstraktionswerkzeug sichergestellt sein müssen: *Produktivität*, *Getyptheit*, *Wachstum* und *lokale Konkretisierbarkeit*.

PRODUKTIVITÄT

Die Produktivität ist zunächst eine Eigenschaft von Nichtterminalsymbolen. Ein Nichtterminal ist dabei produktiv unter einer Grammatik $G = \langle \mathfrak{Z}, P \rangle \in \text{DSG}_{\Sigma_N}$, wenn es möglich ist, aus ihm, beziehungsweise seinem Handle, unter P eine konkrete Heapkonfiguration abzuleiten. Eine DSG ist produktiv, wenn alle Nichtterminale unter ihr produktiv sind.

Die Produktivität ist eine essentielle Eigenschaft für die Konkretisierung. Gibt es ein nichtproduktives Nichtterminal, so kann eine dazugehörige Kante nicht konkretisiert werden. Definiert ist die Produktivität über die konkrete Hülle.

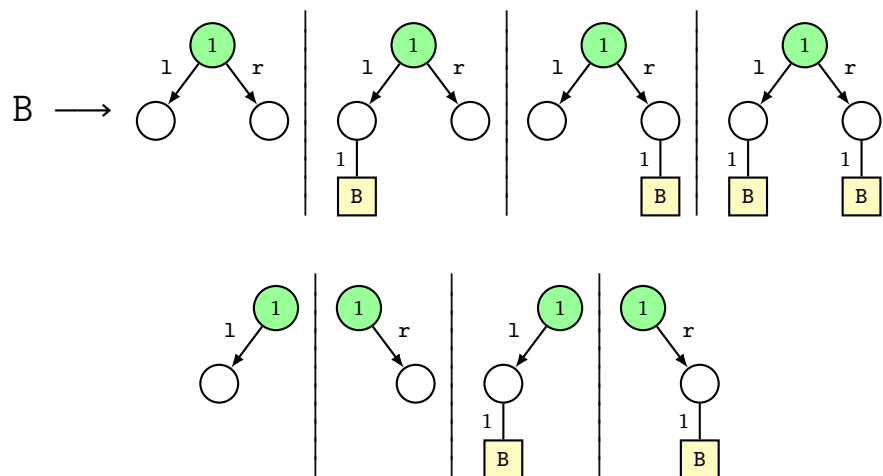
DEFINITION 2.17 (Produktivität) Seien $X \in N$ und $P \in \text{DSR}_{\Sigma_N}$. X ist *produktiv* unter P , wenn $L_P(X^*) \neq \emptyset$ gilt. Eine Grammatik $G = \langle \mathfrak{Z}, P \rangle \in \text{DSG}_{\Sigma_N}$ ist *produktiv*, wenn alle $X \in N_G$ unter P produktiv sind.

Da nichtproduktive Nichtterminale offensichtlich keinen Einfluss auf die Sprache der Grammatik oder die konkrete Hülle eines Graphen haben, gibt es zu jeder nichtproduktiven Grammatik $G = \langle \mathfrak{Z}, P \rangle$ eine produktive Grammatik $G' = \langle \mathfrak{Z}, P' \rangle$, so dass $L_P = L_{P'}$ und $\mathcal{L}(G) = \mathcal{L}(G')$. Diese erhält man durch iteratives Löschen aller Regeln, die nicht zu einem Terminalgraphen führen. Siehe dazu [HJKN12], Anhang A.

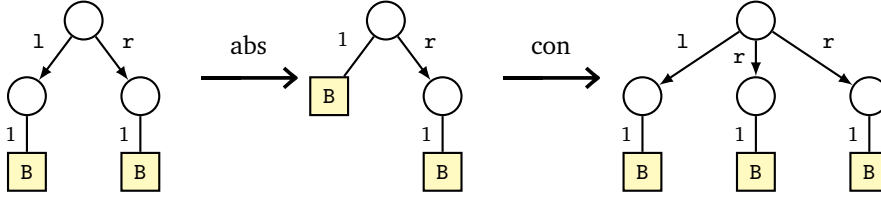
GETYPHEIT

Bei der Abstraktion von Hypergraphen durch DSGen ist der zu abstrahierende Subgraph nicht immer eindeutig gegeben. Kritisch wird dies, wenn zwei verschiedene Subgraphen, die zum Teil dieselben Knoten enthalten, zur selben Nichtterminalkante abstrahiert werden können.

Beispiel 2.11. Der folgende Regelsatz für binäre Bäume soll dies erläutern [HJKN12].



Die Regeln für B erlauben einem Knoten sowohl nur einen rechten oder linken, als auch beide Söhne zu haben. Wendet man nacheinander die Abstraktion und die Konkretisierung auf einen Baum dargestellten Baum an, so führt dies im unteren Beispiel zu einem Graphen, der keine HC ist.



Zunächst wird der linke Sohnknoten mit Hilfe der 7. Regel abstrahiert. Anschließend wird zur erneuten Konkretisierung die 4. Regel angewandt. Der Wurzelknoten besitzt nun einen doppelten Selektor r , die HC-Eigenschaften sind verletzt. Δ

Die Getyptheit unterbindet dieses Verhalten. Der Begriff Typ bezieht sich dabei auf die Tentakeln der Nichtterminale und spezifiziert, welche ausgehenden Terminalkanten, also Selektoren, durch einen Tentakel induziert werden. Die Getyptheit verlangt, dass alle Produktionsregeln für dieses Nichtterminal an den inzidenten Knoten genau diejenigen Selektoren erzeugen, die der Typ des jeweiligen Tentakels vorgibt.

DEFINITION 2.18 (Getyptheit) Eine Grammatik $G = \langle \mathfrak{J}, P \rangle \in \text{DSG}_{\Sigma_N}$ ist *getypt*, wenn für alle $X \in N_G$ und $j \in \mathbb{N}$, $1 \leq j \leq \text{rk}(X)$ eine Funktion $\text{type}(X, j) \subseteq \Sigma$ existiert, so dass für jede HC $H \in L_P(X^\bullet)$ gilt $\text{type}(X, j) = \text{lab}_H(\text{out}_H(\text{ext}_H(j)))$.

Die Existenz der Funktion type reicht aus, um die oben genannte Anforderung zu erfüllen. Für die Abstraktion und Konkretisierung wird sie selbst jedoch nicht benötigt. Bei der in Kapitel 4.4 eingeführten Übersetzung von Grammatiken in Separation-Logic-Formeln sowie der Definition der lokalen Konkretisierbarkeit greifen wir jedoch darauf zurück. Verwendet man in Beispiel 2.11 ausschließlich die Regeln der oberen Reihe, so ist die daraus resultierende Baumgrammatik *getypt* mit $\text{type}(B, 1) = \{r, 1\}$. Es werden also stets beide Selektoren erzeugt. Dies verkleinert allerdings die abgeschlossene Hülle $L_P(B)$. Bäume, die Knoten mit nur einem Selektor enthalten, werden dann nicht mehr unterstützt.

WACHSTUM

Die Wachstumseigenschaft verbietet Produktionsregeln, deren Anwendung den Hypergraphen in Bezug auf Knoten oder Kanten nicht vergrößert. Eine Regel $X \rightarrow X^\bullet$ oder ein Regelsatz, der die Regeln $X \rightarrow Y^\bullet$, $Y \rightarrow X^\bullet$ enthält, kann bei der Abstraktion zu nicht-terminierendem Verhalten führen, da stets eine weitere Projektion einer rechten Regelseite gefunden wird. Für wachsende Grammatiken ist dies ausgeschlossen, weil der Graph in jedem Abstraktionsschritt „kleiner“ wird.

DEFINITION 2.19 (Isolierter Knoten) Seien $H \in \text{HRG}_{\Sigma_N}$ und $v \in V_H$. v ist *isoliert*, wenn es keine Kante $e \in E_H$ gibt, so dass $v \in [\text{att}(e)]$.

Isolierte Knoten repräsentieren Objekte auf dem Heap, die weder über eine Variable, noch über einen Selektor zugreifbar sind (Garbage).

DEFINITION 2.20 (Wachstum) Sei $G = \langle \mathfrak{J}, P \rangle \in \text{DSG}_{\Sigma_N}$. G ist *wachsend*, wenn für alle $X \rightarrow H \in P$ gilt: H enthält keinen isolierten Knoten und

- $H \in \text{HC}_\Sigma$ ODER
- $|V_H| + |E_H| > \text{rk}(X) + 1$.

Isolierte Knoten sind bei der Modellierung von Datenstrukturen offenbar wenig sinnvoll, in einer Grammatik jedoch prinzipiell denkbar. Externe isolierte Knoten führen zu einer nicht terminierenden Abstraktion, wie das folgende Beispiel verdeutlicht.

Beispiel 2.12. Gegeben sei

$$P = \{X \rightarrow \textcircled{1}\}$$

Die rechte Regelseite ist offensichtlich konkret, enthält jedoch einen isolierten Knoten. Ihre Einbettung ist in jedem Graphen an beliebiger Stelle möglich. Bei der rückwärtigen Anwendung der Regel werden also stets weitere X -Kanten erzeugt, so dass der Abstraktionsvorgang nicht terminiert. Δ

Interne isolierte Knoten verursachen derartige Probleme zwar nicht, bieten aus Sicht der Datenstrukturen jedoch auch keinen Mehrwert, weswegen wir sie ebenfalls verbieten.

LOKALE KONKRETISIERBARKEIT

In Abschnitt 2.3.2 wurde der Einsatz der Konkretisierung bei der Programmanalyse und -verifikation beschrieben. Dabei ist es von großer Wichtigkeit, dass sich eine Nichtterminalkante, also ein abstrakter Teil des Heaps, so konkretisieren lässt, dass auf alle Knoten, zu denen durch diese Kante ein Selektor induziert wird, sofort zugegriffen werden kann. Diese Konkretisierung muss unabhängig von der Größe der Datenstruktur in endlich vielen Schritten möglich sein. Die lokale Konkretisierbarkeit fordert daher, dass jede Selektorkante in einem einzigen Ableitungsschritt erzeugbar sein muss.

DEFINITION 2.21 (Lokale Konkretisierbarkeit) Sei $G = \langle \mathfrak{J}, P \rangle \in \text{DSG}_{\Sigma_N}$ eine getypte Datenstrukturgrammatik. G ist *lokal konkretisierbar*, wenn für alle $X \in N_G$ Regelsätze $P_{(X,1)}, \dots, P_{(X, \text{rk}(X))} \subseteq P^X$ existieren, so dass für alle j mit $1 \leq j \leq \text{rk}(X)$ gilt

1. $L_{P_{(X,j)} \cup \overline{P^X}}(X^\bullet) = L_P(X^\bullet)$

2. Für alle $a \in \text{type}(X, j)$, $(X \rightarrow H) \in P_{(X,j)}$ gilt:

$$\text{Es ex. } e \in E_H \text{ mit } \text{lab}_H(e) = a \text{ und } \text{att}_H(e)(1) = \text{ext}_H(j),$$

wobei $P^X := \{X \rightarrow H \in P\}$ und $\overline{P'} := P \setminus P'$.

Es muss also zu jedem Tentakel (X, j) eine Menge an Regeln $P_{(X,j)}$ in P geben, die (1.) P vollständig repräsentiert, also dieselbe Hülle erzeugt, und die (2.) jede Selektorkante (sofort) erzeugt.

Eine Traversierung kann prinzipiell von jeder Seite aus erfolgen, die eine eingehende Kante in den zu konkretisierenden Heapteil hinein bereitstellt. Diese aus dem bereits konkreten Teil der Datenstruktur ausgehenden Kanten können bei der Ableitung nur an den Knoten erzeugt werden, an denen ein Nicht-Reduktionstentakel anliegt. Diese Idee wird durch die lokale Greibach-Normalform formalisiert, die eine hinreichende Bedingung für lokale Konkretisierbarkeit darstellt [JHKN11].

HEAPABSTRAKTIONSGRAMMATIKEN

Eine Datenstrukturgrammatik, die alle vier Anforderungen für die Abstraktion und Konkretisierung erfüllt, bezeichnen wir als *Heapabstraktionsgrammatik* [HJKN12].

DEFINITION 2.22 (Heapabstraktionsgrammatik) $G = \langle \emptyset, P \rangle \in \text{DSG}_{\Sigma_N}$ ist eine *Heapabstraktionsgrammatik* (HAG), wenn G produktiv, getypt, wachsend und lokal konkretisierbar ist. Die Menge aller HAGen über Σ_N wird mit HAG_{Σ_N} bezeichnet.

Mit Hilfe einer HAG können sowohl alle Abstraktions- als auch alle Konkretisierungsschritte adäquat durchgeführt werden. Ziel der Abstraktion ist es, alle rückwärts anwendbaren Produktionen auszuführen, bis keine Projektion einer rechten Regelseite mehr gefunden wird. Dieser Sachverhalt wurde bereits durch die Funktion fullAbstr_G formalisiert. Die Konkretisierung soll eine Heapkonfiguration in eine zulässige Heapkonfiguration transformieren. Dazu werden nacheinander alle Verletzungspunkte erkannt und die dazugehörigen Nichtterminalkanten durch Vorwärtsanwendung von Produktionsregeln eliminiert. Dieser Vorgang wiederholt sich rekursiv, bis alle Verletzungspunkte eliminiert wurden.

DEFINITION 2.23 (con und abs) Seien $G = \langle \mathfrak{J}, P \rangle \in \text{HAG}_{\Sigma_N}$ und $H \in \text{HC}_{\Sigma_N}$. Die Funktion $\text{con} : \text{HC}_{\Sigma_N} \rightarrow 2^{\text{AHC}_{\Sigma_N}}$ ist gegeben durch

$$\text{con}(H) = \begin{cases} \bigcup_{X \rightarrow K \in P_{(X,i)}} \text{con}(H[e/K]) & \text{if } \exists (e, i) \in \text{VP}_H : \text{lab}(e) = X \\ \{H\} & \text{if } H \in \text{AHC}_{\Sigma_N} \end{cases}$$

und $\text{abs} : \text{AHC}_{\Sigma_N} \rightarrow \text{AHC}_{\Sigma_N}$ ist definiert als

$$\text{abs}(H) = H'$$

mit einem beliebigen aber festen $H' \in \text{fullAbstr}_G(H)$.

Nach jeder ausgeführten Anweisung des Programms wird zunächst die Zulässigkeit der Heapkonfiguration durch Anwendung von con wiederhergestellt. Anschließend wird abs ausgeführt, um alle Subgraphen zu abstrahieren, die zum aktuellen Zeitpunkt nicht über eine Variablenkante zugreifbar sind.

$$H_{\text{neu}} = \text{abs}(\text{con}(H_{\text{alt}})).$$

KAPITEL 3

EINFÜHRUNG IN SEPARATION LOGIC

Die semantische Verifikation von Software ist ein breites Forschungsfeld innerhalb der Informatik. Der axiomatische Ansatz, wie er von C. A. Hoare in dem nach ihm benannten Kalkül beschrieben wurde [Hoa69], stellt dabei eine grundlegende Methode sowohl zur Spezifikation der Semantik von Software als auch zu ihrer Verifikation dar. Dazu werden Aussagen über Variablen und damit den Zustand des Programms in Form von sogenannten Assertionen getroffen und die logisch-kausale Abhängigkeit dieser Assertionen untereinander untersucht. Der Benutzer gibt dabei üblicherweise eine Vorbedingung (engl. *precedent*) und eine Nachbedingung (engl. *consequent*) für das Programm an. Der Beweiser versucht dann zu zeigen, dass das Gelten der Vorbedingung die Erfüllung der Nachbedingung nach Programmausführung logisch impliziert. Dazu verwendet er vorgegebene Regeln, die die Auswirkungen der einzelnen Anweisungen auf die Variablen zwar allgemein, aber dennoch möglichst genau erfassen.

Beispiel 3.1. Gegeben sei ein Programm, das die Summe aller Zahlen von 1 bis zum Wert der Variable x berechnet und in der Variable y ablegt. Hierfür sind die Vorbedingung $A = (k = x)$ und die Nachbedingung $B = \left(y = \sum_{j=1}^k j\right)$ denkbar. Der ursprüngliche Wert der Programmvariable x muss dabei in einer anderen Variable k zwischengespeichert werden, weil x seinen Wert im Laufe des Programmes ändern kann. \triangle

Die Assertionen beziehen sich stets auf die aktuellen Werte der Variablen zum jeweiligen Ausführungszeitpunkt. Variablen wie k aus dem obigen Beispiel, die vom Programm selbst nicht verwendet werden, werden als *logische Variablen* bezeichnet [Nol12]. Stellen wir das Programm durch das Symbol c dar, so notieren wir die Aussage „Wenn Bedingung A erfüllt ist, dann gilt nach Ausführung von c Bedingung B “ als sogenanntes *Hoare-Tripel* $\{A\}c\{B\}$. Die Assertionen des Hoare-Kalküls sind in der Lage, Aussagen über eine (endliche) Menge lokaler Variablen zu treffen. Für die Modellierung dynamischer Datenstrukturen reicht dies jedoch nicht aus. Die Separation Logic erweitert deshalb den Hoare-Kalkül um Aussagen über den Heap. Sie wurde von J.C. Reynolds und Peter O’Hearn entwickelt und 2002 erstmalig vorgestellt [Rey02]. Im Gegensatz zu den im vorherigen Kapitel eingeführten Hypergraphen beschreibt die Separation Logic eine Datenstruktur nicht direkt, sondern modelliert zunächst den Heap an sich.

Wie der Hoare-Kalkül erlaubt auch die Separation Logic beliebige Aussagen in FO-Logik. Darüber hinaus stellt sie das Prädikat \rightarrow zur Verfügung, das die Existenz und die Belegung

einer oder mehrerer Speicherzellen des Heaps definiert, sowie das Schlüsselwort **emp**. Die booleschen Operatoren der Hoare-Logik \wedge, \vee, \neg und \Rightarrow werden ergänzt durch die separierende Konjunktion $*$ und die separierende Implikation \ast , auch bekannt als *magic wand* (Zauberstab) [Rey08].

Die Separation Logic wird in dieser Arbeit nicht im vollen Umfang genutzt. Der Grund dafür ist ihre im Vergleich zu den vorgestellten Graphgrammatiken wesentlich größere Aussagekraft in Bezug auf die Modellierung von Datenstrukturen. Um beide Konzepte ineinander zu überführen und füreinander nutzbar zu machen, ist daher eine Angleichung der Mächtigkeit vonnöten.

Der folgende Abschnitt schafft eine Intuition für die Handhabung der Separation Logic, führt das zugrunde liegende Speichermodell ein und erläutert die Operatoren. Im darauffolgenden Abschnitt 3.2 wird das in dieser Arbeit verwendete Fragment der Logik formal syntaktisch und semantisch definiert.

3.1. INTUITIVE ERLÄUTERUNG DER SEPARATION LOGIC

Die Separation Logic (SL) stellt eine echte Erweiterung der Hoare-Logik dar und verwendet insbesondere denselben Beweiskalkül. Dieser Abschnitt wird sich mit dem Aufbau und der Bedeutung einzelner Assertionen beschäftigen. Auf die für Beweise verwendeten Hoare-Tripeln wird später in Kapitel 6 eingegangen. Eine umfangreiche Einführung in die der Separation Logic findet sich in [Rey08].

Eine Hoare-Logik-Formel spezifiziert Anforderungen an eine sogenannte Variableninterpretation. Diese, oft auch als Variablenbelegung oder Programmzustand bezeichnete Funktion bildet Variablennamen auf Werte ab. Logische Variablen, die zur Konservierung der Werte von Programmvariablen dienen, werden in dieser Arbeit nicht betrachtet. Sie können bei Bedarf jedoch ohne Weiteres eingeführt und analog zu jenen verwendet werden. Die Unterscheidung zwischen logischen und Programmvariablen spielt erst bei der eigentlichen Verifikation eine Rolle. Innerhalb der Assertionen werden sie identisch behandelt.

Auf der Menge aller Interpretationen und der Menge der Formeln gibt es eine Modell- oder Erfüllungsrelation \models , wie im folgenden Beispiel gezeigt wird.

Beispiel 3.2. Sei $i = [x \mapsto 2, y \mapsto 4]$ eine Interpretation und $\varphi = (y = 2x)$ eine Assertion. Offenbar wird φ von i erfüllt oder anders ausgedrückt: i ist ein Modell von φ , notiert als $i \models \varphi$. Die Interpretationen $i' = [x \mapsto 2, y \mapsto 3]$ und $i'' = [x \mapsto 1]$ hingegen erfüllen φ nicht: $i' \not\models \varphi, i'' \not\models \varphi$. △

Die Separation Logic trifft Aussagen über den Speicher auf dem Heap. Dieser muss daher ebenfalls Teil der Modellrelation sein. Ein Heap h bildet Speicheradressen auf Inhalte ab. Wir gehen hier in beiden Fällen von natürlichen Zahlen aus, so dass h zunächst eine partielle Abbildung von \mathbb{N} auf \mathbb{N} ist. Während eine Hoare-Formel nur die Variableninterpretation charakterisiert, beschreibt die SL außerdem den Heap. Hervorzuheben ist dabei, dass eine Formel immer den **gesamten** betrachteten Heap oder Heapteil beschreibt. Für einen Heap h , eine Interpretation i und eine Formel φ notieren wir $h, i \models \varphi$ oder $h, i \not\models \varphi$, je nachdem, ob h in Verbindung mit i die Formel erfüllt, oder nicht. Die SL erweitert die Hoare-Logik um die folgenden vier Formelelemente (mit dom notieren wir die Definitionsmenge einer Abbildung):

3.1. Intuitive Erläuterung der Separation Logic

emp: Das Literal **emp** beschreibt den leeren Heap. Es gilt also $h, i \models \mathbf{emp}$ genau dann, wenn $\text{dom}(h) = \emptyset$.

$x \mapsto y$: Diese Aussage bezeichnen wir als *Zeigerassertion*. Sie stellt zwei Anforderungen an den Heap:

1. $\text{dom}(h) = i(x)$ – Der Heap darf nur eine Zelle mit der Adresse enthalten, die dem Wert der Variablen x unter der Interpretation i entspricht.
2. $h(i(x)) = i(x)$ – Diese Zelle muss den Wert von y unter i enthalten.

$\varphi * \psi$: Der Heap lässt sich in zwei disjunkte Heapteile zerlegen, so dass der eine φ und der andere ψ erfüllt. Der Operator $*$ heißt *separierende Konjunktion*, da im Gegensatz zu einer Disjunktion beide Aussagen, φ und ψ , Gültigkeit haben müssen. Der Unterschied zur Konjunktion \wedge besteht darin, dass sich diese Gültigkeit auf jeweils einen Heapteil beschränkt anstatt von beiden gefordert zu werden.

$\varphi \multimap \psi$: Die *separierende Implikation*, auch *Zauberstab* genannt, trifft die Aussage: „Wenn der Heap um einen zu ihm disjunkten Heap, der φ erfüllt, erweitert wird, dann erfüllt er anschließend ψ .“

Beispiel 3.3. Die folgenden Beispiele illustrieren die Bedeutung dieser Elemente (Die Notation ist jeweils $h, i \models \varphi$).

$$[], [x \mapsto 4] \models \mathbf{emp} \quad (3.1)$$

$$[1 \mapsto 2], [] \not\models \mathbf{emp} \quad (3.2)$$

$$[2 \mapsto 3], [x \mapsto 2, y \mapsto 3] \models x \mapsto y \quad (3.3)$$

$$[2 \mapsto 3, 3 \mapsto 5], [x \mapsto 2, y \mapsto 3] \not\models x \mapsto y \quad (3.4)$$

$$[2 \mapsto 3, 3 \mapsto 5], [x \mapsto 2, y \mapsto 3, z \mapsto 5] \models x \mapsto y * y \mapsto z \quad (3.5)$$

$$[2 \mapsto 3, 3 \mapsto 5], [x \mapsto 2, y \mapsto 3, z \mapsto 5] \models x \mapsto y * y \mapsto z * \mathbf{emp} \quad (3.6)$$

$$[2 \mapsto 3], [x \mapsto 2, y \mapsto 3, z \mapsto 7] \models (y \mapsto z) \multimap (x \mapsto y * y \mapsto z) \quad (3.7)$$

In den Zeilen 3.1 und 3.2 lässt sich gut erkennen, dass in der Interpretation definierte Variablen, über die keine Aussagen getroffen werden, die Erfüllung der Formel nicht beeinflussen, der Definitionsbereich des Heaps jedoch exakt darauf passen muss. Zeile 3.5 illustriert die Verwendung der separierenden Konjunktion, die den Heap $[2 \mapsto 3, 3 \mapsto 5]$ in die beiden Teile $[2 \mapsto 3]$ und $[3 \mapsto 5]$ zerlegt, die ihrerseits die beiden Teilformeln erfüllen. **emp** ist dabei das neutrale Element (Zeile 3.6), da jeder Heap in sich selbst und den leeren Heap disjunkt zerlegt werden kann. \triangle

Zusätzlich zu den speziellen SL-Operatoren dürfen auch sämtliche Operatoren und Ausdrücke der Hoare-Logik, wie $\wedge, \vee, \neg, \mathbf{true}, \mathbf{false}, +, -, \dots$ verwendet werden. Dabei ist jedoch zu berücksichtigen, dass sich zwei mit diesen Operatoren verknüpfte Terme stets auf denselben Heap beziehen (es findet also keine Separation statt), was teilweise zu einer zunächst ungewohnten Semantik führt.

Beispiel 3.4. Die Formel $x \mapsto y \vee z \mapsto y$ sagt demnach aus, dass ein Heap h mit dem Definitionsbereich $\text{dom}(h) = \{i(x)\}$ mit dem Inhalt $h(i(x)) = i(y)$ die Formel genauso erfüllt, wie ein Heap h' , dessen Definitionsbereich aus $i(z)$ und demselben Inhalt $h'(i(z)) = i(y)$ besteht. Ein Heap über $\{i(x), i(z)\}$ jedoch erfüllt keinen der beiden Terme und damit auch nicht die gesamte Formel. Δ

Beispiel 3.5. Die Assertion $\varphi = (x \mapsto y \wedge a \mapsto b)$ impliziert für $h, i \models \varphi$, dass x und a in i denselben Wert haben, da h laut dem linken Term nur die Speicheradresse $i(x)$ und laut dem rechten nur die Adresse $i(a)$ enthält. φ ist also äquivalent zu $x \mapsto y \wedge a = x \wedge b = y$. Δ

Zusätzlich zu den elementaren Operatoren findet man häufig den Operator \hookrightarrow , wobei $x \hookrightarrow y$ eine abkürzende Schreibweise für $x \mapsto y * \mathbf{true}$ darstellt. Der Term \mathbf{true} erlaubt jede beliebige Interpretation und jeden Heap. $x \hookrightarrow y$ wird also von jedem Heap erfüllt, der an Adresse $i(x)$ den Wert $i(y)$ enthält, so dass von diesem der Heap $[i(x) \mapsto i(y)]$ separiert werden kann. Die Beschaffenheit des übrigen Heaps spielt wegen \mathbf{true} keine Rolle. In dieser Arbeit wird \hookrightarrow nicht weiter beachtet.

Es gibt eine Reihe spezieller Assertionsklassen, die den Gültigkeitsbereich und die Erweiterbarkeit von Formeln charakterisieren. So unterscheidet die Literatur zwischen *pure*, *strictly exact*, *precise*, *intuitionistic* und *supported Assertions* [Rey08], die nach bestimmten Kriterien miteinander kombiniert werden können. In dieser Arbeit spielen dabei ausschließlich die reinen (*pure*) Assertions eine gesonderte Rolle.

3.2. SL-FRAGMENT ZUR HEAPABSTRAKTION

Während der vorangegangene Abschnitt ein Grundverständnis der Separation Logic vermittelt und sich dabei auf die in der Standardliteratur von Reynolds und O'Hearn verwendeten Elemente bezieht, führt dieser Abschnitt eine leicht abgewandelte Form der SL ein, die in dieser Arbeit verwendet wird. Im Wesentlichen zeichnet sich dieses Fragment der SL durch folgende Besonderheiten aus

- Verbot von \wedge , \neg und \mathbf{true} und damit Einschränkung der Mächtigkeit.
- Inline-Definition von Prädikaten durch „**let**...**in**...“-Konstrukt.

Dieser Abschnitt orientiert sich am Vorgehen von Dodds & Plump [DP08b] und gliedert sich wie folgt: Zu Beginn werden das Speicher- und Heapmodell vorgestellt. Danach wird die Formelsyntax des SL-Fragments eingeführt, gefolgt von einem Abschnitt über die Definition und Bedeutung von Prädikaten. Im Anschluss wird die Semantik definiert und anhand einiger Beispiele erläutert.

HEAP UND VARIABLENINTERPRETATION

Wie schon in Abschnitt 3.1 erwähnt, ist die Modellrelation für SL-Formeln (mindestens) dreistellig, da über die Variableninterpretation hinaus noch eine Heapinterpretation, die wir im Folgenden einfach als *Heap* bezeichnen, benötigt wird. Wir definieren zunächst die Domäne, also den elementaren Datentypen, auf dem wir arbeiten.

DEFINITION 3.1 (Loc, Elem, Var) *Loc* ist eine unendliche, abzählbare Menge an Adressen, so dass $(\text{Loc}, +)$ eine abelsche Gruppe bildet. $\text{Elem} := \text{Loc} \cup \{\mathbf{nil}\}$ erweitert *Loc* um die

unallokierbare Adresse **nil**. Wir nehmen hier an, dass $\text{Loc} = \mathbb{N}$. Var ist eine unendliche Menge (ein Pool) von Variablen, so dass $\text{var}_\Sigma \subsetneq \text{Var}$.

In dieser Arbeit beschränken wir uns auf strukturelle Aussagen über Datenstrukturen. Insbesondere werden keine Nutzdaten betrachtet. Wir verwenden daher Loc als Adress- und Elem als Datenraum. Var ist die Menge der Variablen, die wir in SL-Formeln verwenden. Sie ist nicht mit der Menge der Programmvariablen var_Σ zu verwechseln, die eine echte Teilmenge von Var ist. Der Grund hierfür wird in Kapitel 4 erörtert.

DEFINITION 3.2 (Variableninterpretation) Eine *Variableninterpretation* ist eine partielle Abbildung $i : \text{Var} \rightarrow \text{Loc}$. Sie definiert eine Auswertungsfunktion $\llbracket \cdot \rrbracket i : \text{Var} \cup \{\mathbf{nil}\} \rightarrow \text{Elem}$,

$$\llbracket el \rrbracket i = \begin{cases} \mathbf{nil} & , \text{if } el = \mathbf{nil} \\ i(el) & , \text{if } el \in \text{dom}(i) \end{cases}$$

Die Menge aller Variableninterpretationen bezeichnen wir mit Int .

Eine Variable hält in unserem Modell also ein elementares Datum vom Typ Elem . Wir definieren den Heap zunächst analog dazu als einfachen Speicher, der Adressen Inhalte zuordnet.

DEFINITION 3.3 (Heap) Ein *Heap* ist eine partielle Abbildung $h : \text{Loc} \rightarrow \text{Elem}$. He bezeichnet die Menge aller Heaps.

Zur Realisierung dynamischer Datenstrukturen muss der Heap in der Lage sein, komplexe Objekte, die aus mehreren elementaren Werten gebildet werden, zu speichern. In dieser Arbeit wird dazu vereinfachend von nur einer einzigen Objektklasse ausgegangen, alle Objekte definieren also dieselben Felder. Dies ist jedoch keine funktionale Einschränkung, sondern benötigt lediglich einen größeren Speicher. Zur Definition eines Objektes verwenden wir die bereits in Kapitel 2 verwendete Selektormenge sel_Σ . Um jedem Selektor eine eindeutige relative Adresse zuordnen zu können, ordnen wir sel_Σ kanonisch zu $\langle \text{sel}_\Sigma \rangle$ (siehe Definition 2.2). Jedem Selektor $s \in \text{sel}_\Sigma$ wird der Offset $\text{cn}(s)$ zugeordnet, wobei $0 \leq \text{cn}(s) < |\text{sel}_\Sigma|$. Die Heapadresse von Selektor s einer Variablen x unter $i \in \text{Int}$ ist demnach $\llbracket x \rrbracket i + \text{cn}(s)$ und der auf dem Heap h gespeicherte Wert dieses Selektors entsprechend $h(\llbracket x \rrbracket i + \text{cn}(s))$. Eine Zeigervariable, die direkt auf einen Selektor zeigt, ist in einem Hypergraphen nicht realisierbar. Um eine analoge Ausdrucksstärke zu erhalten, werden daher nur *sichere* Variableninterpretationen betrachtet.

DEFINITION 3.4 (Sichere Interpretation) Sei sel_Σ eine Menge von Selektoren. Eine Interpretation $i \in \text{Int}$ ist *sel_Σ-sicher*, wenn für alle $x_1, x_2 \in \text{dom}(i)$ gilt:

$$x_1 = x_2 \vee |x_1 - x_2| \geq |\text{sel}_\Sigma|$$

Beispiel 3.6. Für $\text{sel}_\Sigma = \{s, u, w\}$, also $|\text{sel}_\Sigma| = 3$ ist $i := [x \mapsto 3, y \mapsto 5]$ nicht sicher, $i' := [x \mapsto 3, y \mapsto 10]$ jedoch schon. △

Für einen Heap $h \in \text{He}$ und eine sel_Σ -sichere Interpretation $i \in \text{Int}$ ist $h(\llbracket x \rrbracket i + \text{cn}(s))$ also der Inhalt von Selektor s des Objektes, auf das Variable x zeigt. Für $y \in \text{Var}$ und $u \in \text{sel}_\Sigma$ mit $\llbracket y \rrbracket i + \text{cn}(u) = \llbracket x \rrbracket i + \text{cn}(s)$ gilt $u = s$ und $y = x$.

3.2.1. SYNTAX VON SLF

In dieser Arbeit wird ein Fragment der Separation Logic betrachtet, das wir mit SLF bezeichnen. Dieses ist im Vergleich zur gesamten Separation Logic funktional eingeschränkt (mehr dazu in Abschnitt 4.1). Für die Definition der Syntax muss zunächst die Menge der Prädikate eingeführt werden.

DEFINITION 3.5 (Pred) Pred ist eine unendliche Menge von Prädikatnamen.

Für Prädikate verwenden wir üblicherweise die Metavariablen σ .

DEFINITION 3.6 (Syntax von SLF) Gegeben ist die folgende kontextfreie Grammatik mit dem Startsymbol S .

$$\begin{aligned}
 E &::= x \mid \mathbf{nil} \\
 P &::= x = y \mid P \wedge P \\
 F &::= \mathbf{emp} \mid x.s \mapsto E \mid F * F \mid \exists x.F \mid F \vee F \mid \sigma(E, \dots, E) \mid \mathbf{let} \Gamma \mathbf{in} F \\
 \Gamma &::= \sigma(x_1, \dots, x_n) = F \mid \Gamma, \Gamma \\
 S &::= \mathbf{emp} \mid x.s \mapsto E \mid S * S \mid \exists x.S \mid S \vee S \mid \sigma(E, \dots, E) \mid \mathbf{let} \Gamma \mathbf{in} S \mid P \wedge S \mid S \wedge P
 \end{aligned}$$

Dabei sind $x, y \in \text{Var}$, $s \in \text{sel}_{\Sigma}$ und $\sigma \in \text{Pred}$ Metavariablen für beliebige Variablen, Selektoren und Prädikate. Die Variablen $\{x_n \mid n \in \mathbb{N}\} \subseteq \text{Var}$ sind feste, für Parameter reservierte Variablen. Jeder Ausdruck sei implizit geklammert und stellt eine sogenannte *Teilformel* dar. $*$ und \wedge binden dabei stärker als \vee . Die Menge aller aus dieser Grammatik erzeugbaren Formeln bezeichnen wir mit SLF (Separation Logic Fragment).

Zunächst fällt auf, dass der Operator \wedge nur im Kontext von S , also auf oberster Ebene oder im **in**-Teil einer Formel, und nur zur Verknüpfung mit einem P -Element verwendet werden darf. Der Grund dafür liegt in der Semantik der Nicht-separierenden Konjunktion \wedge . Diese erlaubt es, zwei voneinander unabhängige Aussagen über denselben Heapteil zu treffen. Dieser Sachverhalt ist mit Graphgrammatiken jedoch nicht abbildbar. Das Nichtterminal P generiert jedoch nur solche Formeln, die ausschließlich Aussagen über die Variableninterpretation, nicht aber über den Heap treffen. Ebenfalls fehlen die Negation \neg und die Schlüsselwörter **true** und **false**. Abschnitt 4.1 geht auf die Gründe für das Fehlen dieser Operatoren ein.

Beispiel 3.7. Einige SLF-Formeln über $\text{sel}_{\Sigma} = \{s, u\}$:

$$(x.s \mapsto y * y.s \mapsto \mathbf{nil}) \vee \mathbf{emp} \tag{3.8}$$

$$\mathbf{let} \sigma(x_1) = (x_1 \mapsto \mathbf{nil}) \mathbf{in} ((x.s \mapsto y * \sigma(y)) \vee \mathbf{emp}) \tag{3.9}$$

$$\mathbf{let} \sigma(x_1) = (\exists r : r.u \mapsto x_1 * x_1.u \mapsto \mathbf{nil}) \mathbf{in} (\exists r : x.s \mapsto r * \sigma(x)) \tag{3.10}$$

$$\mathbf{let} \sigma_1(x_1) = (x_1 \mapsto \mathbf{nil}) \mathbf{in} (\mathbf{let} \sigma_2(x_1, x_2) = (x_2.s \mapsto x_1 * \sigma_1(x_1)) \mathbf{in} (\mathbf{emp})) \tag{3.11}$$

$$x = y \wedge z = x \wedge y.s \mapsto z \tag{3.12}$$

△

DEFINITION 3.7 (Atomare Assertionen) Seien $x \in \text{Var}$, $s \in \text{sel}_\Sigma$ und $el \in \text{Elem}$. Assertionen der Form $\mathbf{emp}, x.s \mapsto el$ und $x = y$ heißen *atomar*. Ferner bezeichnen wir $x.s \mapsto el$ als *Zeiger-* und $x = y$ als *Gleichheitsassertion*. Den Spezialfall $x.s \mapsto \mathbf{nil}$ bezeichnen wir auch als *Nullzeigerassertion*.

DEFINITION 3.8 (Freie Variable) Die Menge der *freien Variablen* $\text{FV}(\varphi)$ einer Formel $\varphi \in \text{SLF}$ ist induktiv definiert:

$$\begin{array}{ll}
 \text{FV}(\mathbf{emp}) & = \emptyset \\
 \text{FV}(x.s \mapsto \mathbf{nil}) & = \{x\} \\
 \text{FV}(x.s \mapsto y) = \text{FV}(x = y) & = \{x, y\} \\
 \text{FV}(\sigma(x_1, \dots, x_n)) & = \{x_1, \dots, x_n\} \\
 \text{FV}(\varphi * \psi) = \text{FV}(\varphi \vee \psi) = \text{FV}(\varphi \wedge \psi) & = \text{FV}(\varphi) \cup \text{FV}(\psi) \\
 \text{FV}(\exists x : \varphi) & = \text{FV}(\varphi) \setminus \{x\} \\
 \text{FV}(\sigma(x_1, \dots, x_n) = \varphi) & = \text{FV}(\varphi) \\
 \text{FV}(\Gamma_1, \Gamma_2) & = \text{FV}(\Gamma_1) \cup \text{FV}(\Gamma_2) \\
 \text{FV}(\mathbf{let} \Gamma \mathbf{in} \varphi) & = \text{FV}(\Gamma) \cup \text{FV}(\varphi)
 \end{array}$$

Beispiel 3.8. Die Variable x ist in φ frei, tritt aber ebenfalls quantifiziert auf, y ist frei und z ist nicht frei:

$$\varphi = (\exists x : x.s \mapsto y) * (\exists z : z.s \mapsto x), \quad \implies \text{FV}(\varphi) = \{x, y\}$$

△

3.2.2. PRÄDIKATE

Prädikate definieren in der Separation Logic, wie auch in vielen anderen Logiken, einen Sachverhalt über eine Sequenz von endlich vielen Parametern. Die Anzahl der Parameter ist dabei für jedes Prädikat fest. In den meisten Veröffentlichungen zur SL werden Prädikate global definiert und innerhalb der Formeln mit einer Reihe von Argumenten aufgerufen. Prominente Beispiele sind **list**, **tree** oder **dag** [Rey08]. Prädikate dienen zur kompakten Darstellung einer Eigenschaft ihrer Parameter. Dabei dürfen sie insbesondere auch rekursiv definiert sein, wie das folgende Beispiel illustriert.

Beispiel 3.9. Rekursives Prädikat zur Definition eines Listensegments mit $\text{sel}_\Sigma = \{\mathbf{n}\}$:

$$\sigma_{\mathbf{ls}}(x_1, x_2) = (x_1.\mathbf{n} \mapsto x_2) \vee (\exists r : x_1.\mathbf{n} \mapsto r * \sigma_{\mathbf{ls}}(r, x_2))$$

Eingebettet in eine SLF-Formel, die die Existenz eines Listensegments von x nach y fordert:

$$\mathbf{let} \sigma_{\mathbf{ls}}(x_1, x_2) = (x_1.\mathbf{n} \mapsto x_2) \vee (\exists r : x_1.\mathbf{n} \mapsto r * \sigma_{\mathbf{ls}}(r, x_2)) \mathbf{in} \sigma_{\mathbf{ls}}(x, y)$$

△

Die Verwendung von rekursiven Prädikaten erlaubt offensichtlich erst die Modellierung dynamischer Datenstrukturen in der Separation Logic. Bei der Übersetzung von Grammatiken

in SLF-Formeln, die ein Ziel dieser Arbeit darstellt, werden Prädikate aus Ableitungsregeln generiert. Die Prädikate hängen also direkt von der Grammatik ab. Eine globale Sichtweise auf diese ist daher ungeeignet. Daher werden ihre Definitionen mit dem „**let**...**in**...“-Konstrukt nach dem Vorbild von Dodds & Plump [DP08b] in die Formel eingebettet (*Inline-Definition*). Im **let**-Teil der Formel können beliebig viele Prädikate nach dem Muster $\sigma(x_1, \dots, x_n) = F$ definiert werden. Die Variablennamen der Parameter sind dabei fest als $\{x_n \mid n \in \mathbb{N}\}$ vorgegeben. In einer SLF-Formel können prinzipiell beliebig viele **let**-Konstrukte ineinander geschachtelt werden (siehe Formel 3.11 in Beispiel 3.7).

DEFINITION 3.9 (let-freie Formel) Eine Formel $\varphi \in \text{SLF}$ heißt *let-frei*, wenn sie das Schlüsselwort **let** nicht enthält. $\overline{\text{SLF}}$ ist die Menge aller **let**-freien Formeln.

Eine **let**-freie Formel kann Prädikate lediglich aufrufen, definieren kann sie diese jedoch nicht.

HINWEIS: Gelegentlich bezeichnen wir auch Formelfragmente, beispielsweise eine Liste von Prädikatdefinitionen „ $\sigma_1(\dots) = \dots, \sigma_2(\dots) = \dots$ “, als **let**-frei, wenn sie **let** nicht enthalten. Da diese aber keine gültige Formel gemäß Definition 3.6 darstellen, sind derartige Fragmente in der Menge $\overline{\text{SLF}}$ nicht enthalten.

In einer Auslegung der SL, die Prädikate unterstützt, ist für die Definition der Semantik zusätzlich zu Heap und Variableninterpretation noch eine dritte Interpretation, die Prädikatinterpretation, nötig.

DEFINITION 3.10 (Prädikatinterpretation) Eine *Prädikatinterpretation* ist eine partielle Abbildung

$$\eta : \text{Pred} \rightarrow 2^{\text{Loc}^* \times \text{He}}$$

PI bezeichnet die Menge aller Prädikatinterpretationen.

Eine Prädikatinterpretation ordnet also einem Prädikatnamen eine Menge an Paaren von Adresssequenzen und Heaps zu. Erstere repräsentiert dabei einen Satz von Argumenten und letztere einen Heap, für den das Prädikat unter diesen Argumenten erfüllt ist. Zu einem gegebenen Prädikat σ enthält $\eta(\sigma)$ also genau die Paare, für die das Prädikat zu wahr ausgewertet.

3.2.3. SEMANTIK VON SLF

Die Semantik von SLF ist durch die Modellrelation \models gegeben. Zu deren Definition werden noch die funktionale Vereinigung und die formale Definition purer Formeln benötigt.

DEFINITION 3.11 (Disjunkte funktionale Vereinigung) Seien D_1 und D_2 zwei Domänen, so dass $D_1 \cap D_2 = \emptyset$. Seien außerdem $f_1 : D_1 \rightarrow W$ und $f_2 : D_2 \rightarrow W$ zwei (partielle) Funktionen. Die *funktionale Vereinigung* ist definiert durch:

$$f_1 \uplus f_2 : (D_1 \cup D_2) \rightarrow W, \quad (f_1 \uplus f_2)(d) = \begin{cases} f_1(d) & , \text{ if } d \in D_1 \\ f_2(d) & , \text{ otherwise} \end{cases}$$

Der Graph von $f_1 \uplus f_2$ entspricht also der Vereinigung der Graphen von f_1 und f_2 . Die Verwendung des Ausdruckes $f_1 \uplus f_2$ impliziert stets die Aussage $\text{dom}(f_1) \cap \text{dom}(f_2) = \emptyset$.

3.2. SL-Fragment zur Heapabstraktion

DEFINITION 3.12 (Pure Formel) Sei $\varphi \in \text{SLF}$ eine Formel. φ heißt *pur*, wenn für alle $i \in \text{Int}$, $\eta \in \text{PI}$, $h, h' \in \text{He}$ gilt

$$h, i, \eta \models \varphi \iff h', i, \eta \models \varphi.$$

Die Formel φ ist also pur, wenn sie keine Aussagen über h trifft. Das ist für herkömmliche SL-Formeln genau dann der Fall, wenn sie weder \mapsto oder **emp** enthalten noch den Aufruf eines Prädikates, auf dessen Definition dies zutrifft. Eingeschränkt auf SLF haben pure Formeln stets die Form $x_1 = y_1 \wedge \dots \wedge x_n = y_n$.

DEFINITION 3.13 (Semantik von SLF, \models) Die Modellrelation \models ist induktiv definiert:

$$\begin{aligned} h, i, \eta \models \mathbf{emp} &\iff \text{dom}(h) = \emptyset \\ h, i, \eta \models x.s \mapsto el &\iff \text{dom}(h) = \{\llbracket x \rrbracket i + \text{cn}(s)\} \\ &\quad \text{und } h(\llbracket x \rrbracket i + \text{cn}(s)) = \llbracket el \rrbracket i \\ h, i, \eta \models \varphi * \psi &\iff \text{es ex. } h_1, h_2 : h_1 \boxplus h_2 = h \\ &\quad \text{und } h_1, i, \eta \models \varphi \text{ und } h_2, i, \eta \models \psi \\ h, i, \eta \models \varphi \vee \psi &\iff h, i, \eta \models \varphi \text{ oder } h, i, \eta \models \psi \\ h, i, \eta \models x = y \wedge \varphi^\dagger &\iff \llbracket x \rrbracket i = \llbracket y \rrbracket i \text{ und } h, i, \eta \models \varphi^\dagger \\ h, i, \eta \models \varphi^\dagger \wedge x = y &\iff h, i, \eta \models x = y \wedge \varphi^\dagger \\ h, i, \eta \models \exists x : \varphi &\iff \text{es ex. } v \in \text{Loc}, \text{ so dass } h, i[x \mapsto v], \eta \models \varphi \\ h, i, \eta \models \sigma(x_1, \dots, x_n) &\iff ((\llbracket el_1 \rrbracket i, \dots, \llbracket el_n \rrbracket i), h) \in \eta(\sigma) \\ h, i, \eta \models \mathbf{let } \Gamma \mathbf{ in } \varphi &\iff h, i, \eta[\sigma \mapsto k_{\text{fix}}(\sigma)]_{\sigma \in \text{dom}(\Gamma)} \models \varphi \end{aligned}$$

wobei i stets sel_Σ -sicher und φ^\dagger nicht pur ist. k_{fix} sei der kleinste Fixpunkt von $f_{\eta, \Gamma}$

$$f_{\eta, \Gamma} : k \mapsto \left[\sigma \mapsto \left\{ (\vec{l}, h) \in \text{Loc}^* \times \text{He} \mid h, [\vec{x} \mapsto \vec{l}], \eta [\beta \mapsto k(\beta)]_{\beta \in \text{dom}(\Gamma)} \models \psi \right\} \right]_{\sigma \in \text{dom}(\Gamma)}$$

(wobei ψ die syntaktische Definition von σ in Γ ist)

Das Funktional $f_{\eta, \Gamma}$ nähert die aus Γ resultierende Prädikatinterpretation durch Iteration auf Basis von η an. In jeder Iteration wird dabei jedem Prädikat σ die Menge an Argument-Heap-Paaren (\vec{l}, h) zugeordnet die durch Einsatz der aktuellen Prädikatinterpretation k die Definition ψ dieses Prädikates erfüllt.

Freie Variablen werden der Variableninterpretation i nicht hinzugefügt, wie es im Falle von \exists erfolgt. Sie müssen bereits in $\text{dom}(i)$ vorhanden sein und einen Wert zugewiesen bekommen. Dadurch hat die jeweilige Variableninterpretation direkten Einfluss auf die Erfüllung der Formel. Analog verhält es sich mit Prädikaten, die nicht innerhalb der Formel definiert werden. Die für sie zulässigen Argument-Heap-Paare müssen ebenfalls bereits in η vorhanden sein. Dies führt zur Verwendung von Tripeln aus $\text{He} \times \text{Int} \times \text{PI}$ für die Modellrelation \models .

DEFINITION 3.14 (Äquivalenz von SLF-Formeln) Zwei Formeln $\varphi, \psi \in \text{SLF}$ sind *äquivalent*, notiert durch $\varphi \equiv \psi$, wenn für alle $h \in \text{He}$, $i \in \text{Int}$, $\eta \in \text{PI}$ gilt

$$h, i, \eta \models \varphi \iff h, i, \eta \models \psi$$

Innerhalb einer Formel quantifizierte (nicht-freie) Variablen lassen sich in beliebige nicht-auftretende Variablen umbenennen.

LEMMA 3.1 Sei $\psi \in \text{SLF}$ eine Formel, so dass $x \notin \text{FV}(\psi)$, und $\text{Var}(\psi)$ die Menge aller in ψ auftretenden Variablen. Dann kann x in eine beliebige Variable $y \notin \text{Var}(\psi)$ umbenannt werden, ohne die Semantik von ψ zu verändern:

$$\psi \equiv \psi[y/x] \quad \text{für beliebiges } y \in \text{Var} \setminus \text{Var}(\psi)$$

$\psi[y/x]$ bezeichnet die Ersetzung aller Vorkommen von x in ψ durch y .

Beweis. Direkt aus Definition 3.13 folgt, dass für $h, i, \eta \models \psi$ auch $h, i, \eta \models \psi[x \mapsto y]$ gilt, da $i(x)$ und $i(y)$, sofern sie definiert sind, wegen der Quantifizierung keine Auswirkung haben. \square

LEMMA 3.2 Sei $\psi \in \text{SLF}$ eine Teilformel von φ und $\psi' \equiv \psi$. Dann gilt

$$\varphi \equiv \varphi[\psi \mapsto \psi']$$

wobei $\varphi[\psi \mapsto \psi']$ die Ersetzung von ψ durch ψ' in φ darstellt.

Beweis. Folgt direkt aus Definition 3.13. \square

3.2.4. FLACHE FORMELN

Die Konstruktion der Syntax erlaubt prinzipiell eine beliebig tiefe Verschachtelung von Prädikatdefinitionen und Formeln mittels „**let**...**in**...“ ineinander. Dies erlaubt eine sehr flexible Kombination insbesondere solcher Assertionen, die ihre eigenen Prädikatdefinitionen „mitbringen“. Bei der Synthese einer HRG aus einer SLF-Formel bereitet es jedoch einige Schwierigkeiten, weswegen hierzu ausschließlich *flache* Formeln zugelassen sind.

DEFINITION 3.15 (Flache Formel) Eine Formel $\varphi = \text{let } \Gamma \text{ in } \psi \in \text{SLF}$ heißt *flach*, wenn Γ und ψ **let**-frei sind.

Dodds & Plump stellen einen Flattening-Algorithmus vor [DP08b], der für die hier definierte Form analog angewandt werden kann. Der einzige Unterschied zwischen dem dort vorgestellten und dem hier eingeführten Separation-Logic-Fragment besteht in der Zulassung von Gleichheitsassertionen, die im Algorithmus jedoch keine Rolle spielen. Davon abgesehen ist der hier vorgestellte Ansatz demjenigen von Dodds & Plump bewusst nachempfunden. Das Flattening-Verfahren erlaubt es, aus jeder SLF-Formel eine äquivalente flache SLF-Formel zu generieren, die dann für die Übersetzung verwendet werden kann. Die Beschränkung auf flache Formeln berührt die Ausdrucksstärke von SLF also nicht.

KAPITEL 4

ÜBERSETZUNG ZWISCHEN HRGEN UND SLF

Hypergraphen, wie sie in Kapitel 2 eingeführt wurden, stellen eine intuitive und anschauliche Methode zur Modellierung von Datenstrukturen da. Die Verwendung von Heapabstraktionsgrammatiken ermöglicht die abstrakte Betrachtung der Modelle. Dies lässt den per Definition unendlichen Zustandsraum dynamischer Datenstrukturen zu einer endlichen Menge an Zuständen zusammenfallen, sofern eine geeignete HAG gefunden wird.

Die Separation Logic hingegen formuliert Aussagen über die Speicherbelegung auf dem Heap durch prädikatenlogische Formeln, sogenannte Assertionen, und ermöglicht so die Verifikation von Programmen über den Hoare-Kalkül. Auch Assertionen lassen sich mittels rekursiver Prädikate abstrakt darstellen, was die kompakte Darstellung dynamisch verketteter Datenstrukturen erlaubt.

Dieses Kapitel befasst sich mit einem Ansatz, beide Abstraktionskonzepte so ineinander zu überführen, dass ihre Bedeutung in Bezug auf die Instanzen der Datenstrukturen erhalten bleibt. Ein solches Verfahren wurde von Dodds & Plump bereits auf der ICGT 2008 für eine vereinfachte Form von Graphgrammatiken vorgestellt [DP08b, Dod08]. Dieses Kapitel baut unmittelbar auf den dort erzielten Ergebnissen auf und überträgt sie auf den HRG-Ansatz von Heinen, et al. [HJKN12], der von ersterem in einigen Punkten wesentlich abweicht.

Abschnitt 4.1 beschäftigt sich daher zunächst mit den Gemeinsamkeiten und Unterschieden zwischen HRGen und SLF, stellt Verbindungen zwischen den verwendeten Konzepten und Begriffen her und erläutert die Überwindung von Inkompatibilitäten. Zum Ende des Abschnittes wird die Darstellung von Heaps durch Hypergraphen durch eine Funktion formal definiert. Die darauffolgenden Abschnitte 4.2 und 4.4 führen schließlich eine Übersetzungsfunktionen zur Grammatiksynthese aus SLF-Formeln sowie zur Formelgenerierung aus Grammatiken ein und beweisen deren Korrektheit.

4.1. UNTERSUCHUNG BEIDER ANSÄTZE

Die Darstellung einer Datenstruktur durch einen Graphen unterscheidet sich in der Art der Betrachtung grundlegend von einer Separation-Logic-Assertion. Erstere modelliert den Zustand einer Datenstruktur aus einem semantischen Blickwinkel, stellt also Beziehungen zwischen den Objekten dar, während letztere Aussagen über den Inhalt von Variablen und Heap in Form von „harten“ Speicherinhalten trifft. Während die Herangehensweise bei Graphen und Grammatiken außerdem auf den konstruktiven Aufbau einer Datenstruktur abzielt, ist sie bei SL-Assertionen eher einschränkend-spezifizierender Natur. Dennoch lassen sich

zwischen fast allen Konzepten beider Methoden Analogien herstellen, die eine Übersetzung der Spezifikationen sowohl in die eine als auch in die andere Richtung ermöglichen.

4.1.1. ANALOGIEN ZWISCHEN DEN ANSÄTZEN

Auf Seiten der HRGen gibt es die folgenden zentralen Konzepte: Graphen, Grammatiken, Produktionsregeln, Startgraphen, Nichtterminale, Tentakel, Nichtterminal-, Variablen- und Selektorkanten, sowie die Sprachrelation \mathcal{L} . Im SL-Ansatz finden wir Heaps, Variablen- und Prädikatinterpretationen, Formeln (inklusive **let**-freien und flachen Formeln), Prädikate, (Null-)Zeiger- und Gleichheitsassertionen und die Modellrelation \models . Zwischen diesen Begriffen lassen sich zahlreiche Parallelen und Entsprechungen finden, worauf im Folgenden eingegangen wird.

Um einen konkreten Zustand einer Datenstruktur darzustellen, bedienen wir uns im HRG-Ansatz eines einzelnen Terminalgraphen. Dieser modelliert sämtliche Variablen und Selektoren der Datenstruktur ohne die Verwendung einer Art von Abstraktion. Im SL-Ansatz wird ein Zustand durch einen Heap und eine Variableninterpretation dargestellt. Eine Grammatik als Werkzeug zur Definition einer Menge von Graphen ähnelt einer SL-Formel, welche ihrerseits eine Menge an Heaps beschreibt, die sie erfüllen. Die Modellrelation \models wird dabei analog zu \mathcal{L} verwendet. Es gibt also offenbar einen Zusammenhang zwischen den Aussagen

$$H \in \mathcal{L}(G) \quad \text{und} \quad h, i, \eta \models \varphi.$$

Die Grammatik $G = \langle \mathfrak{J}, P \rangle$ erzeugt die Sprache der spezifizierten Hypergraphen durch Anwendung der Produktionsregeln P auf die Menge von Startgraphen \mathfrak{J} . Hat die Formel φ die Gestalt **let** Γ **in** ψ , so ergibt sich die volle Spezifikation des Heaps durch Anwendung der in Γ definierten Prädikate auf ψ . Dank des Flattening-Algorithmus (Abschnitt 3.2.4) können wir jede Formel durch eine äquivalente flache Formel ersetzen und daher o. B. d. A. davon ausgehen, dass Γ und ψ **let**-frei sind. Der Teil Γ der Formel ist also in gewisser Hinsicht das Analogon zum Regelsatz P .

Die Entsprechung eines Graphen ist eine disjunktions- und **let**-freie SLF-Formel. Eine Disjunktion spezifiziert zwei Optionen für die zulässige Gestalt des Heaps. Ein Hypergraph ist nicht in der Lage, diesen Sachverhalt abzubilden, da er keine optionalen Knoten oder Kanten kennt. Eine Formel, die eine Disjunktion $\varphi_1 \vee \varphi_2$ enthält, kann jedoch durch zwei Graphen dargestellt werden, wobei der erste die Formel repräsentiert, in der anstelle der Disjunktion φ_1 eingesetzt wird und der zweite Entsprechendes für φ_2 tut. Eine allgemeine, **let**-freie, endliche Formel kann also durch eine endliche Graphmenge dargestellt werden.

Der Aufruf eines Prädikats gleicht dem Vorhandensein einer Nichtterminalkante. Er ist in eine Formel eingebettet, so wie die Kante in einen Graphen eingebettet ist. Seine Argumente entsprechen den an die Kante angeschlossenen Knoten. Eine Nichtterminalkante e mit $\text{lab}(e) = X$ stellt eine Menge möglicher Subgraphen $H \in L_P(X^\bullet)$ auf abstrakte Weise dar. Ähnlich dazu abstrahiert der Prädikataufruf $\sigma(\dots)$ einen Teil der Heapspezifikation. Die Definition eines Prädikates σ mit n Parametern durch eine Formel entspricht dabei dem Teilregelsatz P^X zu einem bestimmten Nichtterminal X des Ranges n . Die Parameter des Prädikates sind das Analogon zu den Tentakeln des Nichtterminals. Sie treffen allgemeine Aussagen über die eingesetzten Argumente beziehungsweise die inzidenten Knoten. Die

Menge der rechten Regelseiten von P^X entspricht in ihrer Gesamtheit der Formel ψ innerhalb der Prädikatdefinition $\sigma(x_1, \dots, x_n) = \psi$.

Variablenkanten in Hypergraphen repräsentieren Programmvariablen, die auf bestimmte Objekte in der Datenstruktur verweisen. In SLF übernehmen freie Variablen diese Rolle. Ihre Werte werden von einer Interpretation i bestimmt. In Verbindung mit einem Heap h wird einer Variable damit ebenfalls die Adresse eines konkreten Objektes zugewiesen.

Sind im Kontext von Heapabstraktionsgrammatiken nur die Regelsätze von Interesse, so können analog dazu auf Seiten der SL die reinen Prädikatdefinitionen, also der **let**-Teil der Formel betrachtet werden. Die Startgraphen und der ihnen entsprechende **in**-Teil bleiben dann außer Acht. Für derartige Untersuchungen ist die Verwendung der Modellrelation \models auf der gesamten Formel **let** Γ **in** φ nicht mehr sinnvoll. Stattdessen wird Γ in diesem Fall häufig als **Menge** von Prädikatdefinitionen aufgefasst. In Kapitel 5 werden Eigenschaften von Prädikatmengen analog zu den die Regelsätze betreffenden HAG-Eigenschaften untersucht. Die folgende Tabelle fasst die Analogien zwischen HRGen und SLF noch einmal zusammen.

HRG-Ansatz		SL-Ansatz	
konkr. Hypergraph	H	Paar: Heap & Variablenint.	(h, i)
Graphmenge	\mathfrak{H}	let -freie Formel	ψ
Startgraphmenge	\mathfrak{S}	in -Teil einer Formel (let -frei)	ψ
Knoten	v	Speicheradresse in $\text{dom}(h)$	l
Nichtterminalsymbol	X	Prädikat	σ
Selektorkante ($\text{lab}(e) \in \text{sel}_{\Sigma}$)	e	Nicht-Null-Zeigerassertion	$x.s \mapsto y$
—		Nullzeigerassertion	$x.s \mapsto \mathbf{nil}$
Variablenkante ($\text{lab}(e) \in \text{var}_{\Sigma}$)	e	Freie Variable	x
Nichtterminalkante ($\text{lab}(e) \in N$)	e	Prädikataufruf	$\sigma(x_1, \dots, x_n)$
Tentakel	(X, j)	Prädikat-Parameter	x_j
(Teil-)Regelsatz	P^X	Prädikatdefinition	$\sigma(x_1, \dots, x_n) = \psi$
Regelsatz	P	Prädikatdefinitionen	Γ
Grammatik	G	flache Formel	φ

Tabelle 4.1.: Analogien zwischen HRGen und SLF

4.1.2. KONZEPTIONELLE UNTERSCHIEDE UND KONFLIKTE

Aufgrund ihrer verschiedenen Sichtweise auf Datenstrukturen, gibt es zwischen dem SL- und dem HRG-Ansatz einige konzeptionelle Abweichungen, die im Folgenden beleuchtet werden sollen. Das von Dodds & Plump in [DP08b] verwendete HRG-Konzept verfügt über einige Besonderheiten und Einschränkungen, die die Übersetzung in SLF-Formeln und die Grammatiksynthese aus selbigen begünstigen. Hyperkanten haben dort stets Rang 3, wobei der erste inzidente Knoten als Quell- und die beiden anderen als Zielknoten betrachtet werden. Jeder Knoten verfügt über genau eine ausgehende Kante und hat demnach exakt zwei Nachfolger. Eine Ausnahme bildet ein durch eine spezielle Rang-1-Kante gekennzeichnete Knoten, der den Wert **nil** repräsentiert. Dieser besitzt logischerweise keine Nachfolger. Auf der SL-Seite sieht das Heapmodell genau zwei Felder pro Objekt vor, in denen die Adressen der beiden Objekte abgelegt werden, die im Graphen adjazent sind. Ist ein Knoten adjazent zum **nil**-Knoten, wird im entsprechenden Feld der Wert **nil** gespeichert. Variablenkanten sind

nicht vorgesehen.

Im hier verwendeten HRG-Modell von Heinen et al. ist die Menge der ausgehenden Selektoren für jeden Knoten eine beliebige Teilmenge von sel_Σ . Die Kardinalität von sel_Σ ist ebenfalls beliebig (jedoch endlich). Einen **nil**-Knoten hingegen gibt es nicht.

IMPLIZITES VS. EXPLIZITES NIL

Aus Tabelle 4.1 wird ersichtlich, dass es auf Seiten der HRGen keine Entsprechung für eine Nullzeigerassertion gibt. Für den Fall, dass ein Selektor nicht definiert ist, also nirgendwohin verweist, ist die entsprechende Selektorkante schlicht nicht vorhanden. In einer Separation-Logic-Formel werden Felder jedoch auf den Wert **nil** gesetzt, wenn sie einen Nullzeiger repräsentieren. Den Grund dafür soll ein Beispiel verdeutlichen.

Beispiel 4.1. Gegeben sei $\text{sel}_\Sigma = \{s, u\}$. Die Formeln

$$\varphi_1 = x.s \mapsto y \quad \not\equiv \quad \varphi_2 = x.s \mapsto y * x.u \mapsto \mathbf{nil}$$

sind nicht äquivalent. Für $h_1, i_1, \eta \models \varphi_1$ gilt $\llbracket x \rrbracket_{i_1} + \text{cn}(u) \notin \text{dom}(h_1)$, für $h_2, i_2, \eta \models \varphi_2$ hingegen gilt $\llbracket x \rrbracket_{i_2} + \text{cn}(u) \in \text{dom}(h_2)$ und $h_2(\llbracket x \rrbracket_{i_2} + \text{cn}(s)) = \mathbf{nil}$. In Worten ausgedrückt, bedeutet dies: φ_1 fordert, dass der Selektor u der Variablen x nicht existiert. In φ_2 gibt es ihn, er verweist jedoch auf **nil**. In Bezug auf den Speicher, dessen Spezifikation die Formel ja primär dient, ist dies ein signifikanter Unterschied. \triangle

Ein implizites **nil** durch Weglassen von Nullzeigerassertionen beziehungsweise deren Darstellung durch **emp** innerhalb von SLF-Formeln ist demnach keine adäquate Alternative. Stattdessen wird sogar davon ausgegangen, dass jedes Objekt stets vollständig spezifiziert wird. Hierauf wird in Abschnitt 4.2 genauer eingegangen.

Eine explizite Modellierung von Nullzeigern in Hypergraphen durch Kanten, die auf einen speziellen Null-Knoten verweisen, ist jedoch möglich. Insbesondere benötigen wir ihn nicht für gegebene Grammatiken, aus denen Formeln generiert werden sollen. Dies spart auf der einen Seite Aufwand beim Modellieren der Graphen und auf der anderen Seite die Notwendigkeit einer Randbedingung, die die Existenz aller Selektoren fordert. Stattdessen wird in Abschnitt 4.4 der Begriff des *Nullzeigers* formal definiert und für die Übersetzung genutzt. Die Verwendung eines dedizierten **nil**-Knotens brächte einige Vorteile, was die Handhabung von Nichtterminalkanten betrifft. So wäre es beispielsweise möglich, einen Tentakel mit **nil** zu verbinden, etwa um das Ende einer Liste zu markieren. SLF könnte dann entsprechend um die Möglichkeit erweitert werden, **nil** als Argument an Prädikate zu übergeben, oder es innerhalb von Formeln mit anderen Variablen gleichzusetzen ($x = \mathbf{nil} \wedge \varphi$). Dies bringt jedoch einige Probleme mit sich. So muss zum Beispiel verhindert werden, dass ein Prädikat (oder eine Produktionsregel) auf dem **nil**-Knoten Felder beziehungsweise ausgehende Selektorkanten definiert, es müssten also je zwei Klassen von Tentakeln und Parametern definiert werden.

In modernen Programmiersprachen werden derartige Probleme meist durch Exception-Handling also Erkennung der **nil**-Dereferenzierung zur Laufzeit gelöst.

VERBOT VON \wedge , \neg UND TRUE

Die nicht auf SLF eingeschränkte Separation Logic erlaubt die nicht-separierende Konjunktion \wedge , die Negation \neg und das Literal **true** (und damit auch **false**). Diese übersteigen jedoch die

Ausdrucksstärke von kontextfreien Hyperkantenersetzungsgrammatiken. Die Gründe dafür werden hier kurz erläutert. Formale Beweise sind in [DP08b] zu finden.

- \wedge – Die nicht-separierende Konjunktion erlaubt die Definition zweier unabhängiger Spezifikationen, die auf demselben Heapteil gelten müssen, um die Formel zu erfüllen. Betrachten wir die Formel $\varphi = \psi_1 \wedge \psi_2$, so muss für $h, i, \eta \models \varphi$ sowohl $h, i, \eta \models \psi_1$ als auch $h, i, \eta \models \psi_2$ gelten. Übertragen wir diesen Sachverhalt auf HRGs, so erhalten wir eine HRG G_{ψ_1} , die der Formel ψ_1 entspricht, und eine HRG G_{ψ_2} für ψ_2 . Der Hypergraph zu (h, i) muss demnach in $\mathcal{L}(G_{\psi_1})$ und in $\mathcal{L}(G_{\psi_2})$, also in $\mathcal{L}(G_{\psi_1}) \cap \mathcal{L}(G_{\psi_2})$ liegen. Der Schnitt zweier kontextfreier Sprachen ist für Stringgrammatiken im Allgemeinen nicht kontextfrei. Da sich Stringgrammatiken durch HRGs darstellen lassen, indem Wörter als Listen betrachtet werden, ist auch der Schnitt zweier Graphgrammatiken nicht kontextfrei.
- true** – Die Formel **true** wird von jedem Heap erfüllt, stellt also keinerlei Anforderungen an die Datenstruktur. Es gibt jedoch Datenstrukturen, die sich durch kontextfreie Grammatiken nicht darstellen lassen. Genannt sei hier beispielsweise ein zweidimensionales Grid, bei dem jeder Knoten mit seinen vier Nachbarn verbunden ist.
- \neg – Die Negation in einer Formel entspricht nach der gleichen Argumentationskette wie für \wedge dem Komplement einer Sprache $\mathcal{L}(G)$, also der Gesamtheit aller Graphen, die nicht in $\mathcal{L}(G)$ liegen. Mit Hilfe von Komplement und Vereinigung ließe sich allerdings der Schnitt zweier Sprachen darstellen, der, wie oben dargelegt, nicht kontextfrei ist.

4.1.3. MEHRFACHE VARIABLENKANTEN

Das Konzept der Variablenkanten ist für die Verifikation von Algorithmen von großer Wichtigkeit. Dabei ist es prinzipiell möglich (und durchaus üblich), dass mehrere Variablenkanten am selben Knoten anliegen. Dies entspricht in einer SLF-Formel einer Menge von Variablen, die auf dasselbe Objekt verweisen. Als Erweiterung von [DP08b] wurden daher die Gleichheitsassertionen eingeführt, die zwei Variablen als gleich spezifizieren. In der weiteren Formel können dann sämtliche Aussagen über das Objekt durch eine der beiden Variablen getroffen werden. Es dürfen zwar auch weiterhin beide verwendet werden, dabei ist jedoch darauf zu achten, dass nicht zweimal dieselbe Assertion auf verschiedenen Variablen definiert wird. Das folgende Beispiel erläutert dieses Problem.

Beispiel 4.2.

$$x = y \wedge x.s \mapsto z \quad \equiv \quad x = y \wedge y.s \mapsto z \quad \not\equiv \quad x = y \wedge x.s \mapsto z * y.s \mapsto z \quad \equiv \quad \text{false}$$

Die letzte der drei Formeln ist unerfüllbar, weil sie den Heap in zwei disjunkte Teile teilt, die jedoch dieselbe Adresse enthalten sollen. Die ersten beiden hingegen sind äquivalent. \triangle

Gleichheitsassertionen erfordern die Einführung von \wedge für pure Formeln. Dies ist insofern unkritisch, als dass diese keinerlei Aussagen über den Heap treffen. Eine Assertion der Form $(x = y) * \varphi$ wäre hingegen unbrauchbar, da $x = y$ und φ dann auf verschiedenen Heapteilen h_1, h_2 mit $h_1 \boxplus h_2 = h$ vorausgesetzt werden. h_1 dürfte dabei beliebige Gestalt haben, da die

Anforderung $h_1, i \models (x = y)$ sich nur auf die Variableninterpretation, nicht aber auf den Heap bezieht. Die Verwendung des \wedge -Operators ist in diesem Zusammenhang also unvermeidbar. Gleichheitsassertionen müssen bei der Grammatiksynthese adäquat in Variablenkanten umgewandelt werden, worauf in Abschnitt 4.2 eingegangen wird. In Prädikaten sind sie gemäß der SyntaxDefinition 3.6 nicht zulässig. Wir werden daher Variablenkanten auf rechten Produktionsregelseiten und analog dazu freie Variablen in Prädikatdefinitionen, von den Parametern x_1, x_2, \dots einmal abgesehen, vollständig verbieten. In den meisten Fällen sind diese ohnehin nur wenig sinnvoll.

4.1.4. OBJEKTMODELL

In vielen Veröffentlichungen zur Separation Logic, z.B. [BCO04, BCO05, DP08b], werden Objekte auf dem Heap atomar betrachtet. Sie können beliebige Felder definieren, haben keine feste Größe und werden unter genau einer Adresse auf dem Heap gespeichert. Dieses Modell erlaubt flexible Aussagen über Objekte und ermöglicht die parallele Verwendung verschiedener Objektklassen. Darüber hinaus werden Anforderungen wie die Einschränkung auf sel_Σ -sichere Interpretationen oder die Adressierung über eine kanonische Sortierung der Selektoren überflüssig. Das Objekt wird dann als partielle Abbildung von Selektoren auf Werte modelliert. Der Zugriff erfolgt mittels eines Funktionsaufrufes der Form $h(\llbracket x \rrbracket i)(s)$ anstelle der direkten Adressierung des Selektors über $h(\llbracket x \rrbracket i + \text{cn}(s))$. Die Assertionen für diese Art der Objektmodellierung haben dann die Form

$$x \mapsto [s_1 : el_1, \dots, s_n : el_n] \quad (4.1)$$

oder ähnlich, anstelle von

$$x.s_1 \mapsto el_1 * \dots * x.s_n \mapsto el_n. \quad (4.2)$$

Genau in der kompakten Darstellung liegt jedoch die Schwäche dieser Methode, die wir fortan als *Field Aggregating* bezeichnen. Die Aufteilung der Spezifikation in separierend konjunctierte Zeigerassertionen in Formel 4.2 erlaubt die Auslagerung eines Teils der Assertionen in Prädikate. Dazu müssen die Felder beziehungsweise Selektoren jedoch an verschiedenen Heapadressen liegen, so dass der Heap disjunkt geteilt werden kann. Man bezeichnet dieses Objektmodell als *Field Splitting*, da die einzelnen Objekte nicht unter einer Speicheradresse aggregiert, sondern auf mehrere verteilt sind [DP08a].

Beispiel 4.3. Die folgende Formel beschreibt zwei miteinander verbundene doppelt verkettete Listensegmente über der Selektormenge $\text{sel}_\Sigma = \{p, n\}$ (previous, next).

$$\begin{aligned} \text{let } \sigma_{\text{dls}}(x_1, x_2) = & (x_1.n \mapsto x_2 * x_2.p \mapsto x_1) \\ & \vee (\exists r : x_1.n \mapsto r * r.p \mapsto x_1 * \sigma_{\text{dls}}(r, x_2)) \\ & \text{in } \sigma_{\text{dls}}(x, y) * \sigma_{\text{dls}}(y, z) \end{aligned}$$

Die Felder p und n der Variable y werden von zwei verschiedenen Prädikataufrufen gesetzt. Dies wäre mit *Field Aggregating* nicht möglich. △

Hyperkantenersetzungsgrammatiken erlauben ebenfalls die Abstraktion einzelner oder mehrerer Selektoren durch Nichtterminale, was gewissermaßen dem *Field Splitting* entspricht.

Ein zum Field Aggregating analoges Verfahren müsste alle Kanten eines Knotens stets gleichzeitig, also im selben Ableitungsschritt erzeugen. Zusammenfassend ist das Field Splitting also nicht nur flexibler als das Field Aggregating, sondern darüber hinaus auch besser mit dem HRG-Ansatz vereinbar.

4.1.5. DARSTELLUNG VON HEAPS UND INTERPRETATIONEN ALS GRAPHEN

Ein konkreter Speicherzustand lässt sich als Paar $h, i \in \text{He} \times \text{Int}$ darstellen. Repräsentiert der Inhalt des betrachteten Speichers eine Datenstruktur, die von einer Formel $\varphi \in \text{SLF}$ spezifiziert wird, so gilt $h, i \models \varphi$. Im HRG-Ansatz wird die Instanz einer Datenstruktur durch einen konkreten Hypergraphen dargestellt. Dieser liegt in der Sprache derjenigen Grammatik, die die Datenstruktur beschreibt. Um eine DSG aus einer SLF-Formel zu erzeugen, muss also zunächst festgelegt werden, welcher Hypergraph einen bestimmten Speicherzustand beschreibt. Gleiches gilt für die Gegenrichtung. Wir definieren daher eine Funktion α , die einem Heap-Variableninterpretations-Paar h, i einen konkreten Hypergraphen $H = \alpha(h, i)$ zuordnet, der die Datenstruktur in h mit den durch i definierten Variablenkanten darstellt.

DEFINITION 4.1 (Graphfunktion α) Die Funktion $\alpha : \text{He} \times \text{Int} \rightarrow \text{HC}$ ist definiert wie durch

$$\alpha(h, i) = (V, E, \text{att}, \text{lab}, \text{ext}) \quad (4.3)$$

mit der Knoten- und Kantenmenge

$$V = \bigcup_{j=1}^n \{v_j\} \subseteq \text{Loc} \quad \text{wobei} \quad n = \frac{|\text{dom}(h)|}{|\text{sel}_\Sigma|} \quad (4.4)$$

$$E = \{e_{v,s} \mid v \in V, s \in \text{sel}_\Sigma, h(v + \text{cn}(s)) \neq \mathbf{nil}\} \cup \{e_x \mid x \in \text{dom}(i) \wedge i(x) \in \text{dom}(h)\} \quad (4.5)$$

$$\text{ext} = \varepsilon. \quad (4.6)$$

und den Funktionen $\text{att} : E \rightarrow V \cup V^2$ und $\text{lab} : E \rightarrow \Sigma$ so dass

$$\text{att}(e_{v,s}) = vh(v + \text{cn}(s)), \quad \text{att}(e_x) = i(x) \quad (4.7)$$

$$\text{lab}(e_{v,s}) = s, \quad \text{lab}(e_x) = x \quad (4.8)$$

Die Knoten v_1, \dots, v_n ergeben sich aus den Adressen des jeweils ersten Selektors der Objekte in h :

$$v_1 = \min(\text{dom}(h)) \quad (4.9)$$

$$v_j = \min \left(\text{dom}(h) \setminus \bigcup_{k=1}^{j-1} \bigcup_{l=0}^{|\text{sel}_\Sigma|-1} \{v_k + l\} \right) \quad (4.10)$$

Knoten sind zunächst nicht auf eine bestimmte Domäne festgelegt, sondern beliebige Objekte, die ihre Semantik innerhalb eines Graphen erst durch die Funktion att erhalten. Es ist demnach unproblematisch, V als Teilmenge von Loc anzusehen. Die Funktion α verwendet daher direkt die Objektadressen aus h als Knoten. Damit dies funktioniert, müssen sämtliche

Kapitel 4. Übersetzung zwischen HRGen und SLF

Felder aller in h definierten Objekte vorhanden sein. Dies wird jedoch durch die in Abschnitt 4.2 formulierte Anforderung **C1** an die verwendeten SLF-Formeln für alle Modelle dieser Formeln garantiert. Ein Heap h definiert dann genau n Objekte. Die Adresse v_j des j -ten Objektes ist dabei das kleinste in $\text{dom}(h)$ verbleibende Element, nachdem die Adressen aller Selektoren der bis dahin extrahierten Objekte entfernt wurden.

Die Menge E wird mit zwei Typen von Kanten gefüllt: Zu jedem Feld, das nicht den Wert **nil** enthält, wird eine Selektorkante angelegt. Außerdem wird zu jeder von i definierten Variable, deren Wert eine Adresse in h darstellt, eine Variablenkante erzeugt. Die Inzidenzfunktion att assoziiert jede Selektorkante mit ihrem Ursprungsknoten und dem durch h definierten Zielknoten und jede Variablenkante mit dem durch i festgelegten Knoten. Es verbleibt die Labellingfunktion, die jeder Selektorkante ihren Selektor und jeder Variablenkante ihre Variable zuweist. Alle Kanten sind offensichtlich Terminalkanten.

Beispiel 4.4. Die verwendete Selektormenge sei $\text{sel}_\Sigma = \{s, u\}$ und $\text{cn}(s) < \text{cn}(u)$. Wir betrachten den Heap $h = [1 \mapsto 5, 2 \mapsto \text{nil}, 5 \mapsto \text{nil}, 6 \mapsto 1]$ und die Variableninterpretation $i = [x \mapsto 5]$. Der resultierende Graph lautet

$$\alpha(h, i) = \underbrace{(\{1, 5\})}_V, \underbrace{\{e_{1s}, e_{5u}, e_x\}}_E, \underbrace{[e_{1s} \mapsto 1, 5, e_{5u} \mapsto 5, 1, e_x \mapsto 5]}_{\text{att}}, \underbrace{[e_{1s} \mapsto s, e_{5u} \mapsto u, e_x \mapsto x]}_{\text{lab}}, \underbrace{\varepsilon}_{\text{ext}}$$

△

Die Graphfunktion α erzeugt stets Heapkonfigurationen. Das folgende Lemma formatliert diesen Sachverhalt.

LEMMA 4.1 (HC-Eigenschaft von α) Für alle $h \in \text{He}$ und $i \in \text{Int}$ gilt

$$\alpha(h, i) \in \text{HC}_{\Sigma_N}$$

Beweis. Per Konstruktion haben alle Selektorkanten den Rang 2 und alle Variablenkanten den Rang 1. Jede Selektorkante $e_{v,s}$ wird an jedem Knoten v höchstens einmal erzeugt. Ebenso wird für jede Variable $x \in \text{Var}$ höchstens eine Variablenkante e_x erzeugt. Die Sequenz der externen Knoten ist stets leer. □

Durch die Verwendung von Heapadressen als Knoten, gibt es zu jeder Heapkonfiguration genau einen Heap. Dies ist insofern nicht selbstverständlich, als das die durch einen Graphen dargestellte Datenstruktur unabhängig von absoluten Adressen ist. Zwei Graphen werden daher gewöhnlicherweise nicht auf Gleichheit sondern auf Isomorphie überprüft (siehe Kapitel 2).

DEFINITION 4.2 (Heapfunktion α^{-1}) Die Funktion $\alpha^{-1} : \text{HC} \rightarrow \text{He} \times \text{Int}$ ordnet jeder Heapkonfiguration $H \in \text{HC}$ einen Heap h und eine unendliche Menge an Variableninterpretationen i zu, so dass gilt

$$\alpha^{-1}(H) = \{(h, i) \in \text{He} \times \text{Int} \mid H = \alpha(h, i) \wedge |\text{dom}(h)| = |V_H| \cdot |\text{sel}_\Sigma|\}$$

Der Grund für die Unendlichkeit von $\alpha^{-1}(H)$ liegt darin, dass $\text{dom}(i)$ bei der Semantik von SLF keine Rolle spielt. Alle Variablen, denen i Werte außerhalb von $\text{dom}(h)$ zuweist, haben auf die Modellrelation \models keinen Einfluss, weswegen sie auch von α ignoriert werden. Abbildung 4.1 illustriert die Rolle der in den folgenden Abschnitten eingeführten Grammatiksynthese und Formelgenerierung in Abhängigkeit der Graphfunktion α und ihrer Inversen. Ein Paar $(h, i) \in \text{He} \times \text{Int}$ soll demnach eine Formel φ genau dann erfüllen, wenn der dazugehörige Graph $H = \alpha(h, i)$ in der Sprache der daraus synthetisierten Grammatik liegt. Gleiches gilt für die Gegenrichtung unter Verwendung von α^{-1} .

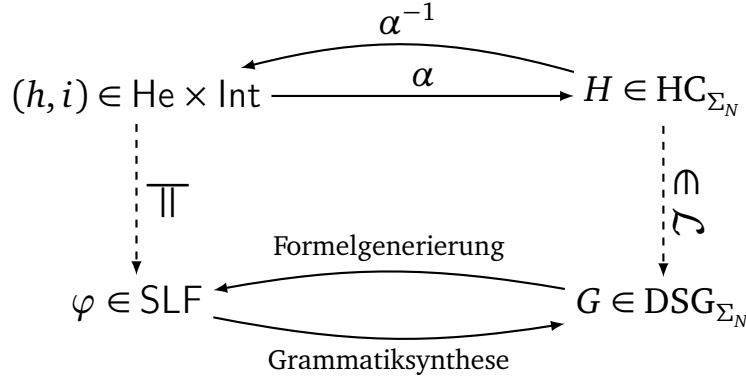


Abbildung 4.1.: Übersicht über Synthese und Formelgenerierung

4.2. SYNTHESE VON HRGs AUS SLF-FORMELN

Das vorgestellte Fragment der Separation Logic, SLF, ist in seiner Ausdrucksstärke an die Mächtigkeit kontextfreier Hyperkantenersetzungsgrammatiken angepasst. Nachdem der vorangegangene Abschnitt durch Analyse der Analogien den Weg für die Überführung beider Konzepte ineinander bereitet hat, wird im Folgenden eine konkrete Methode zur Synthese von Grammatiken aus SLF-Formeln vorgestellt und formal eingeführt. Die resultierende Grammatik soll eine zur Formel äquivalente Semantik in Bezug auf die im letzten Abschnitt eingeführte Graphfunktion α aufweisen. Sei $G \in \text{HRG}_{\Sigma_N}$ die aus einer Formel $\varphi \in \text{SLF}$ synthetisierte Grammatik. Dann soll für alle $h, i \in \text{He} \times \text{Int}$ gelten

$$h, i \models \varphi \iff \alpha(h, i) \in \mathcal{L}(G).$$

Wie schon in Abschnitt 4.1 erwähnt, basiert der hier vorgestellte Ansatz auf den Ergebnissen von Dodds & Plump [DP08b] und verwendet zur besseren Nachvollziehbarkeit teilweise dieselben Formelzeichen und Funktionsnamen.

Im Folgenden wird die Synthese von Grammatiken durch drei Funktionen $h[\cdot]$, $r[\cdot]$ und $g[\cdot]$ definiert, die wir stets mit doppelten Klammern notieren, um Verwechslungen, z.B. mit Heaps, auszuschließen. Zunächst wird die Synthese von Graphmengen aus **let**-freien Formeln durch $h[\cdot]$ definiert. Die Funktion $r[\cdot]$ übersetzt unter Verwendung von $h[\cdot]$ Prädikatdefinitionen in Regelsätze. Schließlich wird die Funktion $g[\cdot]$ definiert, die aus beliebigen flachen Formeln vollständige Grammatiken synthetisiert.

Kapitel 4. Übersetzung zwischen HRGen und SLF

Zunächst gilt es, einen Bezug zwischen den Variablen einer Formel und den Knoten eines Graphen herzustellen. In SLF wird grundsätzlich jedes Objekt durch mindestens eine Variable beschrieben. Auf Hypergraphen trifft dies nicht zu und ist im Allgemeinen auch nicht sinnvoll. Um während des Syntheseprozesses die einzelnen Knoten jedoch identifizieren zu können und außerdem die Namen der Variablen zur Erzeugung von Variablenkanten zu erhalten, führen wir *getaggte Hypergraphen* ein.

DEFINITION 4.3 (Getaggtter Hypergraph) Ein *Tagging* t zu einem Graphen $H \in \text{HG}_{\Sigma_N}$ ist eine Funktion

$$t : V_H \rightarrow 2^{\text{Var}},$$

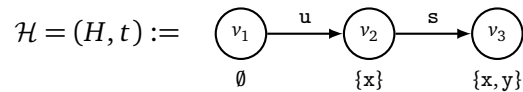
die jedem Knoten eine Menge von Variablen zuordnet.

Ein *getaggtter Graph* ist ein Paar $\mathcal{H} = (H, t)$ aus einem Hypergraphen und einem Tagging. $\text{HG}_{\Sigma_N}^T$ und $\text{HC}_{\Sigma_N}^T$ bezeichnen die Menge der getaggtten Hypergraphen bzw. Heapkonfigurationen über Σ_N .

HINWEIS: Das Symbol \mathcal{H} anstelle von H bezeichnet stets einen getaggtten Graphen. Übergeben wir einen solchen Graphen $\mathcal{H} = (H, t)$ als Argument an eine Funktion, die auf HG_{Σ_N} oder HC_{Σ_N} definiert ist, so behandeln wir dies aus Gründen der Einfachheit, als ob stattdessen der enthaltene Graph H übergeben würde und ignorieren t .

Das Tagging eines Hypergraphen wird in Abbildungen durch an den Knoten notierte Variablenmengen dargestellt.

Beispiel 4.5. Sei $H = (\{v_1, v_2, v_3\}, \{e_1, e_2\}, [e_1 \mapsto v_1 v_2, e_2 \mapsto v_2 v_3], [e_1 \mapsto u, e_2 \mapsto s], \varepsilon)$ und $t = [v_1 \mapsto \emptyset, v_2 \mapsto \{x\}, v_3 \mapsto \{x, y\}]$.



Die Beschriftungen innerhalb der Knoten dienen lediglich der Illustration. Alle drei Knoten sind intern. △

In Abschnitt 4.1 wurde der Zusammenhang von Prädikaten und Nichtterminalen erläutert. Es wird also eine injektive Abbildung von Prädikatnamen auf Nichtterminale benötigt, die diesen Zusammenhang herstellt. Da Prädikatnamen und Nichtterminale jedoch rein logische Gebilde sind, deren Semantik erst durch Prädikatinterpretationen und Produktionsregeln entsteht, können wir die jeweils unendlichen Mengen als gleich annehmen:

$$\text{Pred} = N.$$

Syntaktisch wird der erzeugten Kante also der Prädikatname (σ) als Label zugewiesen. Ferner gilt $\text{rk}(\sigma) = n$. Wir werden jedoch weiterhin die Formelzeichen $X \in N$ und $\sigma \in \text{Pred}$ verwenden, um einerseits anzuzeigen, ob das Symbol gerade als Nichtterminal oder als Prädikat betrachtet wird. Andererseits ist es für den Großteil der Aussagen über Nichtterminale und Prädikate nicht von Bedeutung, dass beide Klassen gleich sind. Wird ein Bezug zwischen Nichtterminalen und Prädikaten hergestellt verwenden wir daher die Notation X_σ , wenn σ als Nichtterminalsymbol verwendet wird, und σ_X analog.

4.2.1. GRAPHSYNTHESE

Die erste Stufe der Grammatiksynthese ist die Übersetzung einzelner, **let**-freier Assertionen in Graphmengen. Diese wird von der Funktion $h[\cdot]$ realisiert.

Für die Definition dieser Graphsynthesefunktion werden zwei neue Operatoren verwendet: der unäre Vereinfachungsoperator (*Unification*) und der binäre Verschmelzungsoperator (*Join*).

DEFINITION 4.4 (Vereinfachungsoperator \Downarrow) Sei $\mathcal{H} \in \text{HG}_{\Sigma_N}^T$ mit $\text{ext}_{\mathcal{H}} = \varepsilon$. Die Vereinfachung $\Downarrow \mathcal{H}$ ist ein getaggter Hypergraph

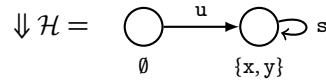
$$\Downarrow \mathcal{H} := ((V, E, \text{att}, \text{lab}, \varepsilon), t)$$

und f eine surjektive Abbildung $f : V_{\mathcal{H}} \rightarrow V$, so dass

- (1) $E = E_{\mathcal{H}}, \quad \text{lab} = \text{lab}_{\mathcal{H}}$
- (2) $\forall v \in V_{\mathcal{H}} : \quad t_{\mathcal{H}}(v) \subseteq t(f(v))$
- (3) $\forall v_1, v_2 \in V : \quad v_1 \neq v_2 \implies t(v_1) \cap t(v_2) = \emptyset$
- (4) $\forall v_1, v_2 \in V_{\mathcal{H}} : \quad t_{\mathcal{H}}(v_1) = \emptyset \wedge v_1 \neq v_2 \implies f(v_1) \neq f(v_2)$
- (5) $\forall v \in V : \quad t(v) \subseteq \bigcup_{v' \in V_{\mathcal{H}}} t(v')$
- (6) $\forall e \in E, 1 \leq j \leq \text{rk}(e) : \quad \text{att}(e)(j) = f(\text{att}_{\mathcal{H}}(e)(j))$

Der Vereinfachungsoperator transformiert einen getaggten Graphen durch Verschmelzen aller Knoten, deren Tagging nicht disjunkt ist, die also durch dieselbe Variable repräsentiert werden, so dass jede Variable anschließend höchstens einmal im Tagging erscheint. Die Funktion f bildet jeden Knoten in \mathcal{H} auf seine Entsprechung in $\Downarrow \mathcal{H}$ ab. Es handelt sich dabei gewissermaßen um eine Simulation von \mathcal{H} durch $\Downarrow \mathcal{H}$. Die Kantenmenge mitsamt ihren Labels bleibt dabei erhalten (1), alle Variablen im Tag eines Knotens bleiben erhalten (2), die neuen Tags sind paarweise disjunkt (3), anonyme Knoten (mit leerem Tag) werden nicht verschmolzen (4), es werden keine zusätzlichen Variablen in das Tagging aufgenommen (5) und jede Kante ist in $\Downarrow \mathcal{H}$ zu den Knoten inzident, die ihren inzidenten Knoten in \mathcal{H} entsprechen (6).

Beispiel 4.6. Wir betrachten \mathcal{H} aus Beispiel 4.5.



Die Knoten v_2 und v_3 wurden wegen der gemeinsamen Variable x verschmolzen. Die beiden Kanten blieben erhalten, die Inzidenzen wurden entsprechend angepasst. \triangle

Der Vereinfachungsoperator wird in erster Linie dazu verwendet, dass eine Variable, die an verschiedenen Stellen einer Formel auftritt, stets mit demselben Knoten identifiziert wird. Die Worst-Case-Laufzeit für seine Berechnung liegt in der Komplexitätsklasse $\mathcal{O}(\max_{v \in V} |t(v)|^2)$.

$|V|^2$), da jeder Knoten mit jedem Knoten und dabei jeweils jede Variable seines Tags mit jeder des anderen Tags verglichen werden muss.

Wir führen außerdem den binären Verschmelzungsoperator als abkürzende Schreibweise für die vereinfachte Vereinigung zweier Graphen ein.

DEFINITION 4.5 (Verschmelzungsoperator \bowtie) Seien $\mathcal{H}_1, \mathcal{H}_2 \in \text{HG}_{\Sigma_N}^T$ getaggte Hypergraphen mit $\text{ext}_{\mathcal{H}_1} = \text{ext}_{\mathcal{H}_2} = \varepsilon$ und disjunkten Knoten- und Kantenmengen. Die *Verschmelzung* der Graphen ist definiert durch

$$\mathcal{H}_1 \bowtie \mathcal{H}_2 := \Downarrow ((V_{\mathcal{H}_1} \uplus V_{\mathcal{H}_2}, E_{\mathcal{H}_1} \uplus E_{\mathcal{H}_2}, \text{att}_{\mathcal{H}_1} \uplus \text{att}_{\mathcal{H}_2}, \text{lab}_{\mathcal{H}_1} \uplus \text{lab}_{\mathcal{H}_2}, \varepsilon), t_{\mathcal{H}_1} \uplus t_{\mathcal{H}_2})$$

Der Verschmelzungsoperator wird im Sinne des kartesischen Produktes auf Mengen erweitert. Seien also $\mathfrak{H}_1, \mathfrak{H}_2 \subseteq \text{HG}_{\Sigma_N}^T$ Mengen getaggtter Graphen.

$$\mathfrak{H}_1 \bowtie \mathfrak{H}_2 := \{\mathcal{H}_1 \bowtie \mathcal{H}_2 \mid \mathcal{H}_1 \in \mathfrak{H}_1, \mathcal{H}_2 \in \mathfrak{H}_2\}$$

Die eigentliche Graphsynthese ist analog zum Aufbau der Formeln induktiv definiert, basierend auf den drei Grundelementen von Formeln: **emp**, Zeigerassertionen und Prädikataufrufen. Jede dieser elementaren Assertionen wird in eine einelementige Graphmenge übersetzt, die jeweils den Graphen enthält, der die Assertion repräsentiert. Im Gegensatz zu α sind die Knoten jedoch keinesfalls an konkrete Speicheradressen gebunden, sondern – im Gegenteil – stets „frische“, bisher noch nicht verwendete Objekte. Ferner sind die Graphen in den durch $h[\cdot]$ generierten Mengen im Allgemeinen nicht konkret.

Basierend auf den drei Basisfällen wird die Übersetzung komplexer Formeln durch Komposition der aus den Teilformeln synthetisierten Graphmengen realisiert.

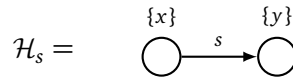
DEFINITION 4.6 (Graphsynthese $h[\cdot]$)

$$h[\cdot] : \overline{\text{SLF}} \rightarrow 2^{\text{HG}_{\Sigma_N}^T}$$

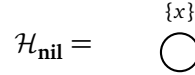
Elementare Formeln:

- $h[\mathbf{emp}] := \{\mathcal{H}_{\mathbf{emp}}\} = \{((\emptyset, \emptyset, [], [], \varepsilon), [])\}$

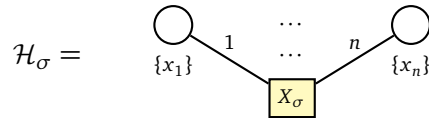
- $h[x.s \mapsto y] := \{\Downarrow \mathcal{H}_s\}$



- $h[x.s \mapsto \mathbf{nil}] := \{\Downarrow \mathcal{H}_{\mathbf{nil}}\}$



- $h[\sigma(x_1, \dots, x_n)] := \{\Downarrow \mathcal{H}_\sigma\}$



Kapitel 4. Übersetzung zwischen HRGen und SLF

Die existenzielle Quantifizierung einer Variable führt zur Anonymisierung des entsprechenden Knotens, indem sie aus dem Tagging gelöscht wird. Quantifizierte Variablen haben einen eingeschränkten Gültigkeitsbereich. Außerhalb dieses Bereiches können keine Aussagen über sie getroffen werden. Analog verhält es sich mit anonymen Knoten. Ein Knoten v , aus dessen Tag die Variable x gelöscht wurde, wird nicht mehr mit einem anderen Knoten v' mit $t(v') = \{x\}$ fusioniert. Knoten mit leerem Tag werden überhaupt nicht verschmolzen, auch nicht mit anderen anonymen Knoten.

Zur Übersetzung einer Gleichheitsassertion wird eine der beiden Variablen dem Tag desjenigen Knotens hinzugefügt, der bereits mit der anderen Variable getaggt ist. Anschließend werden die Knoten durch Anwendung des Vereinfachungsoperators fusioniert.

Beispiel 4.8.

$$h[\varphi] = \left\{ \begin{array}{c} \text{---} \xrightarrow{u} \text{---} \\ \{x\} \qquad \{y\} \end{array} \right\} \implies h[\mathbf{x} = \mathbf{y} \wedge \varphi] = \left\{ \begin{array}{c} \Downarrow \text{---} \xrightarrow{u} \text{---} \\ \{x, y\} \qquad \{y\} \end{array} \right\} = \left\{ \begin{array}{c} \text{---} \xrightarrow{u} \text{---} \\ \{x, y\} \end{array} \right\}$$

△

Die Worst-Case-Laufzeit für die Berechnung der Graphsynthese ergibt sich durch die Übersetzung von separierenden Konjunktionen. Dabei werden die Graphen zweier Mengen $\mathfrak{H}_1, \mathfrak{H}_2$ mittels \bowtie paarweise vereinigt, wobei jeweils die Vereinfachung \Downarrow ausgeführt wird. Letztere hat eine Komplexität von $\mathcal{O}(\max_{v \in V} |t(v)|^2 \cdot |V|^2)$, wodurch sich in Kombination mit dem kartesischen Produkt eine maximale Berechnungszeit von $\mathcal{O}(\zeta(\mathfrak{H}_1, \mathfrak{H}_2) \cdot |\mathfrak{H}_1| \cdot |\mathfrak{H}_2|)$ mit

$$\zeta(\mathfrak{H}_1, \mathfrak{H}_2) := \max \left\{ \max_{v \in V} (|t(v)|^2 \cdot |V|^2) \mid V = V_{H_1} \cup V_{H_2}, H_1 \in \mathfrak{H}_1, H_2 \in \mathfrak{H}_2 \right\}$$

ergibt. Sei l die Länge der Formel. Dann liegt die höchstmögliche Anzahl durchgeführter Verschmelzungen in $\mathcal{O}(l)$, was zur Komplexitätsklasse $\mathcal{O}(l \cdot \zeta(\mathfrak{H}_1, \mathfrak{H}_2) \cdot |\mathfrak{H}_1| \cdot |\mathfrak{H}_2|)$ führt. Stark vereinfacht lässt sich dies mit $\mathcal{O}(l^7)$ abschätzen, wenn man annimmt, dass jede Graphmenge höchstens l Graphen mit jeweils höchstens l Knoten enthält, die jeweils maximal l Tag-Variablen haben. Diese Abschätzung ist jedoch sehr grob und weit von realistischen Formeln entfernt.

Das folgende Beispiel einer komplexeren SLF-Formel schließt die Graphsynthese ab.

Beispiel 4.9. Sei $\varphi = \mathbf{x}.s \mapsto \mathbf{y} * (\exists z : \sigma(\mathbf{x}, z) \vee \sigma(\mathbf{z}, \mathbf{y})) \in \text{SLF}$

$$h[\varphi] = \left\{ \begin{array}{c} \text{---} \xrightarrow{s} \text{---} \\ \{x\} \qquad \{y\} \end{array} \right\} \bowtie \left\{ \begin{array}{c} \text{---} \xrightarrow{1} \boxed{X_\sigma} \xrightarrow{2} \text{---} \\ \{x\} \qquad \emptyset \end{array}, \begin{array}{c} \text{---} \xrightarrow{1} \boxed{X_\sigma} \xrightarrow{2} \text{---} \\ \emptyset \qquad \{y\} \end{array} \right\}$$

$$= \left\{ \begin{array}{c} \begin{array}{c} \{x\} \text{---} \xrightarrow{1} \boxed{X_\sigma} \xrightarrow{2} \text{---} \\ \downarrow s \\ \{y\} \end{array}, \begin{array}{c} \text{---} \xrightarrow{1} \boxed{X_\sigma} \xrightarrow{2} \text{---} \\ \uparrow s \\ \{x\} \end{array} \end{array} \right\}$$

△

4.2.2. PRODUKTIONSREGELSYNTHESE

Dieser Abschnitt widmet sich der Synthese von Produktionsregeln aus Prädikatdefinitionen. Sie wird durch die Funktion $r[\cdot]$ realisiert, die eine Liste von Prädikatdefinitionen, wie sie im **let**-Teil einer Formel auftritt, in einen Regelsatz übersetzt.

DEFINITION 4.7 (Produktionsregelsynthese $r[\cdot]$)

$$r[\cdot] : \left\{ \Gamma[\Gamma]^* \mid \Gamma \in \left\{ \sigma(x_1, \dots, x_n) = \psi \mid \sigma \in \text{Pred}, n \in \mathbb{N}^0, \psi \in \overline{\text{SLF}} \right\} \right\} \rightarrow N \times \text{HG}_{\Sigma_N}$$

$$\bullet r[\sigma(x_1, \dots, x_n) = \psi]$$

$$= \{X_\sigma \rightarrow H' \mid \exists (H, t) \in h[\psi] \wedge \text{tags}(H, t) = \{x_1, \dots, x_n\} \wedge H' = \text{expose}(H, x_1 \dots x_n)\}$$

mit

$$- \text{tags}(H, t) := \bigcup_{v \in V_H} t(v) \quad (\text{Variablen aller Tags})$$

$$- \text{expose}((H, t), \vec{x}) = ((V_H, E_H, \text{att}_H, \text{lab}_H, \vec{v}), t) \quad \text{so dass}$$

$$|\vec{v}| = |\vec{x}| \quad \text{und f\u00fcr alle } j, \quad 1 \leq j \leq |\vec{x}| \quad \text{gilt} \quad \vec{x}(j) \in t(\vec{v}(j)).$$

(Die Knoten zur Variablen-Sequenz \vec{x} werden in derselben Reihenfolge extern gemacht)

$$\bullet r[\Gamma, \Gamma] = r[\Gamma] \cup r[\Gamma]$$

Wie $h[\cdot]$ ist $r[\cdot]$ ebenfalls induktiv definiert. Hier gibt es jedoch nur einen Basisfall: die Definition eines Pr\u00e4dikates, und einen komponierten Fall: die durch Kommata getrennte Liste einzelner Definitionen. F\u00fcr eine einzelne Pr\u00e4dikatsdefinition $\sigma(x_1, \dots, x_n) = \psi$ generiert sie die Menge aller Regeln, die das Nichtterminal σ (beziehungsweise X_σ) als linke Regelseite und einen Graphen H' als rechte Regelseite verwenden, so dass (H, t) eine \u00dcbersetzung von ψ durch $h[\cdot]$ ist. Es wird also zun\u00e4chst die Menge aller rechten Regelseiten aus ψ synthetisiert. Dazu m\u00fcssen alle Knoten, die nicht mit einem der Parameter x_1, \dots, x_n getaggt sind, einen leeren Tag haben; es darf also keine weiteren freien Variablen in ψ geben. Anschließend wird H' aus H erzeugt, indem alle Knoten, deren Tagging x_1, \dots, x_n enth\u00e4lt, durch Verwendung der Funktion expose exponiert (extern gemacht) werden. Da $h[\cdot]$ selbst keine externen Knoten erzeugt, wird dabei nichts \u00fcberschrieben.

Wird eine Liste von mehr als einer Pr\u00e4dikatsdefinition \u00fcbersetzt, so werden die resultierenden Regels\u00e4tze mittels \cup vereinigt.

4.2.3. GRAMMATIKSYNTHESE

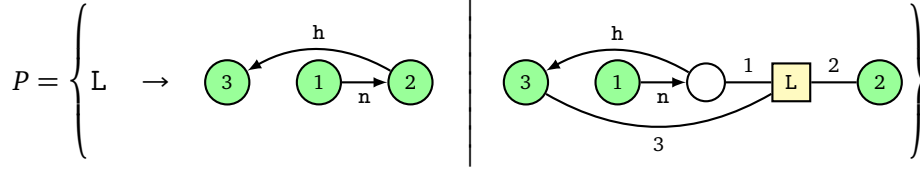
Die Synthese ganzer Grammatiken aus beliebigen flachen SLF-Formeln erfolgt durch die Funktion $g[\cdot]$ unter Verwendung von $h[\cdot]$ und $r[\cdot]$.

DEFINITION 4.8 (Grammatiksynthese $g[\cdot]$)

$$g[\cdot] : \text{SLF} \rightarrow \text{HRG}_{\Sigma_N}$$

$$g[\text{let } \Gamma \text{ in } \varphi] = \langle \exists, P \rangle$$

$r[\cdot]$ entfernt nun das Tagging, macht die mit x_1, x_2 und x_3 getaggten Knoten in der entsprechenden Reihenfolge extern und macht aus jedem Graphen der Menge eine Produktionsregel mit $L \in N$ als linker Regelseite.



Als Endresultat erhalten wir die fertig synthetisierte Grammatik $G = \langle \mathcal{G}, P \rangle \in \text{HRG}_{\Sigma_N}$. \triangle

Die Komplexität der Grammatiksynthese als ganzes wird von der Graphsynthese dominiert. Die Funktion $r[\cdot]$ kann in linearer Zeit über die Formellänge ausgewertet werden, da sie lediglich die syntaktische Bildung der Produktionsregeln auf Basis der Ergebnisse von $h[\cdot]$ definiert. Die Synthese mehrerer rechter Regelseiten bei gleicher Formellänge geht dabei zugunsten der einzelnen Graphsynthesen und erhöht die Komplexität daher nicht. Die Einführung von Variablenkanten durch $g[\cdot]$ führt wiederum $\mathcal{O}(l)$ viele Verschmelzungen durch. In diesem Fall hat eines der Elemente, \mathcal{H}_x , jedoch stets konstante Größe. Die Laufzeit ist also wiederum $\mathcal{O}(\max_{v \in V} |t(v)|^2 \cdot |V|^2)$ pro Verschmelzung, wodurch sich unter Berücksichtigung von höchstens l vielen Startgraphen die sehr grobe Abschätzung $\mathcal{O}(l^6)$ ergibt, was durch $\mathcal{O}(l^7)$ jedoch bereits abgedeckt ist. An dieser Stelle sei nochmals erwähnt, dass diese Betrachtungen sehr grob und daher extrem pessimistisch sind.

4.2.4. DSG-EIGENSCHAFTEN SYNTHETISierter GRAMMATIKEN

Nicht jede synthetisierte Grammatik ist eine DSG. Dieser Abschnitt führt ein Kriterium für SLF-Formeln an, das die DSG-Eigenschaft der daraus synthetisierten Grammatik sicherstellt. Außerdem wird ein Algorithmus vorgestellt, der diese Eigenschaft für auswertbare Formeln (Siehe Definition 5.2) prüft.

Das DSG-Kriterium für Formeln basiert auf dem Begriff der Abrollung.

DEFINITION 4.9 (Abrollung) Sei $\varphi \in \text{SLF}$. Die k -te Abrollung $\text{unroll}_k(\varphi)$ ist definiert durch

- $\text{unroll}_0(\varphi) = \varphi$
- $\text{unroll}_k(\varphi)$ ist die Formel, die aus $\text{unroll}_{k-1}(\varphi)$ durch Ersetzung jedes Prädikataufrufes durch seine Definition entsteht, wobei die Parametervariablen durch die jeweiligen Argumente ersetzt werden. Seien die Prädikate im **let**-Teil einer flachen Formel definiert durch $\sigma_j(x_1, \dots, x_{n_j}) = \psi_j$. Dann ist

$$\text{unroll}_{k+1}(\varphi) = \text{unroll}_k(\varphi)[\text{unroll}_k(\psi_1)[y_1/x_1, \dots, y_{n_1}/x_{n_1}] / \sigma_1(y_1, \dots, y_{n_1}) \\ \text{unroll}_k(\psi_m)[y_1/x_1, \dots, y_{n_m}/x_{n_m}] / \sigma_m(y_1, \dots, y_{n_m})]$$

wobei $\varphi[\psi/\psi']$ die Ersetzung aller Vorkommen von ψ' in φ durch ψ bezeichnet.

LEMMA 4.2 Für alle $\varphi \in \text{SLF}$ und alle $k \in \mathbb{N}$ gilt:

$$\varphi \equiv \text{unroll}_k(\varphi)$$

Kapitel 4. Übersetzung zwischen HRGen und SLF

Beweis. Ergibt sich aus der Semantik von SLF (Definition 3.13). \square

Ferner benötigen wir die Disjunktive Normalform, die eine Formel in disjungierte Teilformeln zerlegt, die selbst keine weiteren Disjunktionen enthalten und die wir als *Disjunkte* bezeichnen (engl. *disjuncts*).

DEFINITION 4.10 (Disjunktive Normalform) Die Funktion dnf sei definiert als Mengenwertige Funktion über **let**-freien Formeln.

$$\text{dnf} : \overline{\text{SLF}} \rightarrow 2^{\overline{\text{SLF}}}$$

$\text{dnf}(\varphi)$ ist dabei die kleinste Menge von Disjunkten, so das gilt

$$\bigvee_{\varphi' \in \text{dnf}(\varphi)} (\varphi') \equiv \varphi \quad \text{und} \quad \forall \varphi' \in \text{dnf}(\varphi) : \varphi' \text{ enthält keine Disjunktion}$$

dnf ist aufgrund der Existenz der disjunktiven Normalform für alle Formeln definiert und kann durch Anwendung der De Morgan'schen Gesetze hergestellt werden. $*$ wird dabei behandelt wie \wedge .

Darauf basierend kann das das DSG-Kriterium für SLF formuliert werden. Gemäß Tabelle 4.1 repräsentiert jedes Disjunkt in der disjunktiven Normalform einer Formel φ einen einzelnen Graphen in $h\llbracket\varphi\rrbracket$.

DEFINITION 4.11 (DSG-Kriterium für SLF-Formeln) Gegeben sei die Formel

$$\varphi = \text{let } \sigma_1(x_1, \dots, x_{n_1}) = \psi_1, \dots, \sigma_m(x_1, \dots, x_{n_m}) = \psi_m \text{ in } \varphi' \in \text{SLF}.$$

φ erfüllt das *DSG-Kriterium*, wenn für alle $j, k \in \mathbb{N}$, $1 \leq j \leq m$ die folgenden zwei Bedingungen gelten:

1. Jedes $\psi' \in \text{dnf}(\text{unroll}_k(\psi_j))$, das frei von Prädikataufrufen ist, enthält für jedes $x \in \text{Var}$ (unter Umbenennung aller quantifizierten Variablen in frische Variablen, die nicht anderweitig verwendet werden) und jeden Selektor $s \in \text{sel}_{\Sigma}$ keine zwei Assertionen der Form $x.s \mapsto y_1$ und $x.s \mapsto y_2$ für beliebige $y_1, y_2 \in \text{Var}$.
2. Die Bedingung (1) gilt ferner für alle $\varphi'' \in \text{dnf}(\text{unroll}_k(\varphi'))$, wenn darin nacheinander alle Gleichheitsassertionen $x = y$ entfernt und jeweils alle weiteren Vorkommen von x und y in φ'' durch das kanonisch kleinere Element in $\langle \{x, y\} \rangle$ substituiert werden.¹

SATZ 4.3 Sei $\varphi \in \text{SLF}$ eine Formel, die das *DSG-Kriterium* erfüllt. Dann ist $g\llbracket\varphi\rrbracket$ eine DSG.

Beweis. Eine Grammatik $G = \langle \mathfrak{Z}, P \rangle \in \text{HRG}_{\Sigma_N}$ ist DSG, wenn alle Startgraphen und sämtliche aus Nichtterminalen ableitbaren Graphen Heapkonfigurationen sind, also $\mathfrak{Z} \subseteq \text{HC}_{\Sigma_N}$ und $\bigcup_{X \in N_G} L_P(X^\bullet) \subseteq \text{HC}_{\Sigma_N}$ gelten.

Die ersten beiden HC-Bedingungen (Rang 1 für Variablen, Rang 2 für Selektoren und eindeutige Variablenkanten), sowie die vierte Bedingung ($\text{ext} = \varepsilon$) sind per Konstruktion

¹Es werden also alle gleichgesetzten Variablen in φ'' durch einen eindeutigen Repräsentanten ersetzt.

4.2. Synthese von HRGs aus SLF-Formeln

von $g[\cdot]$, $h[\cdot]$ und $r[\cdot]$ stets erfüllt: Die Funktion $h[\cdot]$ erzeugt ausschließlich Selektorkanten des Ranges 2 und $g[\cdot]$ Variablenkanten mit Rang 1. Desweiteren werden Variablen nur in Startgraphen und jede Variable höchstens einmal erzeugt. Auf rechten Regelseiten werden ausschließlich die zu Parametern gehörenden Knoten extern gemacht, in Startgraphen überhaupt keine. Alle Parameterknoten werden bei der Hyperkantenersetzung substituiert, so dass in den konkreten Hüllen der Nichtterminal-Handles $L(X^\bullet)$, $X \in N$, keine externen Knoten verbleiben.

Es bleibt zu zeigen, dass das obige Kriterium mehrfache Selektorkanten des gleichen Labels am selben Knoten in der Sprache synthetisierter Grammatiken verhindert. Dazu wird die Korrektheit der Synthesefunktionen vorausgesetzt, die in Abschnitt 4.3 gezeigt wird.

Hinreichendheit des Kriteriums:

Sei $\varphi \in \text{SLF}$ eine Formel, die die Bedingung des Satzes erfüllt und $G = \langle \mathcal{J}, P \rangle := g[\varphi]$.

Teilbedingung 1: Es gibt keinen doppelten ausgehenden Selektor in den konkreten Hüllen der Nichtterminal-Handles.

Annahme: Es gibt ein Nichtterminal $X \in N$, so dass sich daraus ein Graph mit doppelt ausgehendem Selektor ableiten lässt, also $H \in L_p(X^\bullet)$, $s \in \text{sel}_\Sigma$, $v \in V_H$, $e_1, e_2 \in E_H$, so dass $\text{lab}_H(e_1) = \text{lab}_H(e_2) = s$ und $\text{att}_H(e_1)(1) = \text{att}_H(e_2)(1) = v$.

Da das Handle X^\bullet keine Selektoren enthält, werden e_1 und e_2 durch Ableitung erzeugt. Dafür gibt es zwei Fälle.

Fall 1: e_1 und e_2 werden gleichzeitig erzeugt. Dann existiert ein produktives Nichtterminal $Y \in N$ mit einer Regel $Y \rightarrow H' \in P$, so dass $e_1, e_2 \in E_{H'}$. Dafür muss es in der DNF der Prädikatdefinition $\sigma_Y(x_1, \dots, x_n) = \psi_Y$ ein $\psi'_Y \in \text{dnf}(\psi_Y)$ geben, das zwei Assertionen $x.s \mapsto y_1$ und $x.s \mapsto y_2$ ^(*) für Variablen $x, y_1, y_2 \in \text{Var}$ enthält. Da Y produktiv ist, gibt es ein $k \in \mathbb{N}$, so dass in $\text{dnf}(\text{unroll}_k(\psi_Y))$ ein Disjunkt existiert, das die beiden Assertionen und keine weiteren Prädikataufrufe enthält. Die Bedingung ist daher verletzt. Widerspruch.

Fall 2: e_1 und e_2 werden nacheinander erzeugt. Es gibt dann Graphen H_1, H'_1, H_2, H'_2 , so dass

$$X^\bullet \Rightarrow_p^* H_1 \Rightarrow_p H'_1 \Rightarrow_p^* H_2 \Rightarrow_p H'_2 \Rightarrow_p^* H \quad (\dagger)$$

und produktive Nichtterminale $X_1 \in \text{lab}(E_{H_1})$, $X_2 \in \text{lab}(E_{H_2})$, sowie Regeln $\pi_1 = X_1 \rightarrow K_1$, $\pi_2 = X_2 \rightarrow K_2 \in P$, so dass H'_1 durch π_1 aus H_1 und H'_2 durch π_2 aus H_2 entstehen und $e_1 \in E_{K_1}, e_2 \in E_{K_2}$.

Die Regeln π_1 und π_2 sind aus Prädikatdefinitionen hervorgegangen:

$$\pi_1 \in r[\sigma_{X_1}(\dots) = \psi_1] \quad \text{und} \quad \pi_2 \in r[\sigma_{X_2}(\dots) = \psi_2]$$

Demnach sind $K_1 \in h[\psi_1]$ und $K_2 \in h[\psi_2]$. Wegen der Produktivität gibt es auch hier $k_1, k_2 \in \mathbb{N}$, so dass ein Disjunkt in $\text{dnf}(\text{unroll}_{k_1}(\psi_1))$ eine Assertion $x.s \mapsto y_1$ und eines in $\text{dnf}(\text{unroll}_{k_2}(\psi_2))$ eine Assertion $x.s \mapsto y_2$ ^(*) mit $x, y_1, y_2 \in \text{Var}$ enthält und beide frei von Prädikataufrufen sind. Sei ψ die Definition von σ_X , also $\sigma_X(\dots) = \psi$. Dann gibt es wegen (\dagger) auch ein k , so dass $\text{dnf}(\text{unroll}_k(\psi))$ ein Disjunkt mit $x.s \mapsto y_1$ und $x.s \mapsto y_2$, ^(*) jedoch ohne weitere Prädikataufrufe. Widerspruch.

Kapitel 4. Übersetzung zwischen HRGen und SLF

Teilbedingung 2: Aus keinem Startgraphen lässt sich ein doppelt ausgehender Selektor ableiten.

Annahme: Es gibt einen Startgraphen $\bar{H} \in \mathfrak{Z}$, $H \in L_P(\bar{H})$, $s \in \text{sel}_\Sigma$, $v \in V_H$, $e_1, e_2 \in E_H, \dots$ (wie oben). Der Beweis von *Teilbedingung 1* gilt analog mit dem Unterschied, dass an den mit (\ddagger) markierten Stellen statt derselben Variable x auch zwei Variablen x_1, x_2 auftreten können,

$$x_1.s \mapsto y_1 \quad \text{und} \quad x_2.s \mapsto y_2,$$

die im **in**-Teil von φ gleichgesetzt werden

$$\varphi = \mathbf{let} \dots \mathbf{in} \dots x_1 = x_2 \wedge \dots$$

und so zum Widerspruch führen. Die Argumentation bezieht sich hier nicht auf die Abrollungen einer Prädikatdefinition $\sigma_X(\dots) = \psi$ sondern auf die des **in**-Teils von φ .

Notwendigkeit des Kriteriums

Sei $G = \langle \mathfrak{Z}, P \rangle = g[\varphi] \in \text{DSG}_{\Sigma_N}$.

Dann gilt für alle $H \in \mathcal{L}(G) = \bigcup_{H' \in \mathfrak{Z}} L_P(H')$, dass H eine konkrete Heapkonfiguration ist, also $H \in \text{HC}_\Sigma$. Jeder Selektor wird pro Knoten in V_H also höchstens einmal definiert.

Selbiges gilt auch für alle $H \in \bigcup_{X \in N_G} L_P(X^*)$.

O. B. d. A. hat φ die Form

$$\mathbf{let} \sigma_1(\dots) = \psi_1, \dots, \sigma_m(\dots) = \psi_m \mathbf{in} \varphi'$$

Annahme: φ erfüllt die Bedingung des Satzes nicht. Dann sind zwei Fälle denkbar, die der Verletzung je einer Teilbedingung (siehe Beweis der Hinreichendheit) entsprechen.

Fall 1: Es existieren $j, k \in \mathbb{N}$, $1 \leq j \leq m$ und $y_1, y_2 \in \text{Var}$, so dass ein Disjunkt in $\text{dnf}(\text{unroll}_k(\psi_j))$ die Assertionen $x.s \mapsto y_1$ und $x.s \mapsto y_2$, jedoch keinen Prädikataufruf enthält. Wegen $\bigvee(\text{dnf}(\text{unroll}_k(\psi_j))) \equiv \psi_j$ gibt es eine rechte Regelseite K , so dass in P eine produktive Regel $X_{\sigma_j} \rightarrow K$ existiert. Aufgrund der Produktivität enthält die konkrete Hülle von K einen Graphen $H' \in L_P(K)$, der entsprechend den oben erwähnten Assertionen den Selektor s an einem Knoten doppelt enthält. G ist demnach keine DSG. Widerspruch.

Fall 2: Es gibt ein $k \in \mathbb{N}$, so dass $\text{dnf}(\text{unroll}_k(\varphi'))$ nach Umbenennung aller quantifizierten in frische Variablen die drei Assertionen $x_1.s \mapsto y_1$, $x_2.s \mapsto y_2$ und $x_1 = x_2$ mit $x_1, x_2, y_1, y_2 \in \text{Var}$ oder aber $x.s \mapsto y_1$ und $x.s \mapsto y_2$ mit $x, y_1, y_2 \in \text{Var}$ enthält. Trifft Ersteres zu, werden durch $h[\cdot]$ die Knoten mit den Tags x_1 und x_2 mittels \bowtie verschmolzen, so dass beide Fälle für den resultierenden Graphen äquivalent sind.

Die Argumentation ist analog zu *Fall 1*, nur dass anstelle einer rechten Regelseite ein Startgraph $K \in \mathfrak{Z}$ nachgewiesen wird, so dass dessen konkrete Hülle einen Graphen $H' \in L_P(K)$ mit doppelt ausgehendem Selektor s an einem Knoten v enthält. Die Argumentation ist dabei analog, so dass G auch in diesem Fall keine DSG ist. Widerspruch. \square

Die direkte Entscheidung des Kriteriums über die k -te Abrollung ist offenbar nicht möglich, da hierfür unendlich viele k betrachtet werden müssten. Der folgende Algorithmus prüft daher ein auf Formeln definiertes, hinreichendes Kriterium für die DSG-Eigenschaft. Dabei

werden Mengen von Mengen von Variablen-Selektor-Paaren als Analyseinformationen (AI) verwendet: $AI \subseteq 2^{\text{Var} \times \text{sel}_\Sigma}$. Jedes Element in einem solchen AI repräsentiert eine mögliche Konfiguration definierter Selektoren. Die Analyse wird im Syntaxbaum der Formel als Bottom-Up-Technik durchgeführt. Die Disjunktion \vee definiert dabei zwei optionale Konfigurationen, beziehungsweise Teilgraphen. Die Mengen werden also vereinigt. Die separierende Konjunktion hingegen erweitert eine oder mehrere Konfigurationen um neue Variablen-Selektor-Paare, es werden daher die Elemente der Analyseinformationen vereinigt. Bevor solch ein Paar hinzugefügt wird, muss stets geprüft werden, ob es bereits in einem Element von AI vorhanden ist. Ist dies der Fall, dann erlaubt die entsprechende Grammatik die Ableitung zweier Selektoren am selben Knoten und der Algorithmus bricht mit dem Ergebnis „Fail!“ ab. Das Verfahren ist in zwei Teile gegliedert. Zunächst wird durch eine Fixpunktiteration auf den Prädikatdefinitionen die Analyseinformation für jeden Prädikataufruf (\tilde{AI}_σ) berechnet. Dabei werden aus Effizienzgründen nur Parameter betrachtet und noch keine Prüfungen vorgenommen. Wenn sich die \tilde{AI}_σ -Mengen für die einzelnen Prädikate σ nicht mehr ändern, terminiert die Fixpunktiteration und der eigentliche Algorithmus (Teil 2) wird auf Basis ihrer Ergebnisse ausgeführt. Da im **in**-Teil von Formeln Gleichheitsassertionen erlaubt sind, werden in Teil 2 innerhalb der Formeln alle Variablen durch Variablenmengen repräsentiert. Für jede Gleichheitsaussage werden die Mengen innerhalb der Teilformel, auf der diese gilt, durch deren Vereinigung ersetzt.

ALGORITHMUS 4.1 (DSG-Kriterium für SLF-Formeln) Gegeben sei die Formel

$$\text{let } \sigma_1(x_1, \dots, x_{n_1}) = \psi_1, \dots, \sigma_m(x_1, \dots, x_{n_m}) = \psi_m \text{ in } \varphi \in \text{SLF}$$

Teil 1: Fixpunktiteration über alle Prädikate.

Es gilt herauszufinden, welche Parameter-Selektor-Paare von Prädikaten in welcher Kombination definiert werden können. Führe für jedes Prädikat σ_j die folgenden Schritte aus. Sei $1 \leq j \leq m$ und $\sigma_j(x_1, \dots, x_{n_j}) = \psi_j$. Es werden Analyseinformationen der Form $\tilde{AI} \subseteq 2^{\text{PS}_j}$ mit $\text{PS}_j := \{x_1, \dots, x_{n_j}\} \times \text{sel}_\Sigma$ betrachtet.

1. $\tilde{AI}_{\sigma_j} = \{\emptyset\}$
2. Erzeuge den Parsingbaum² der Formel. Führe eine Bottom-Up-Analyse durch und berechne die Informationen \tilde{AI} für jeden Knoten des Baums wie folgt:

Blatt $y.s \mapsto el$:	$\tilde{AI} := \{(y, s)\} \cap \text{PS}_j$
Blatt $\sigma'_k(y_1, \dots, y_{n_k})$:	$\tilde{AI} := \{\tilde{ai}[y_1/x_1, \dots, y_{n_k}/x_{n_k}] \cap \text{PS}_j \mid \tilde{ai} \in \tilde{AI}_{\sigma'_k}\}^3$
Innerer Knoten „*“:	$\tilde{AI} := \{\tilde{ai}_l \cup \tilde{ai}_r \mid \tilde{ai}_l \in \tilde{AI}_{\text{links}}, \tilde{ai}_r \in \tilde{AI}_{\text{rechts}}\}^4$
Innerer Knoten „ \vee “:	$\tilde{AI} := \tilde{AI}_{\text{links}} \cup \tilde{AI}_{\text{rechts}}$
Innerer Knoten „ $\exists x$ “:	$\tilde{AI} := \tilde{AI}_{\text{Sohn}}$ (wird ignoriert)
Wurzel (nach *, \vee oder \exists):	$\tilde{AI}_{\sigma_j} := \tilde{AI}$

²Syntaxbaum, der Operatoren und Quantoren in den inneren Knoten und die Elementarformeln in den Blättern enthält.

³ $\tilde{ai}[y_1/x_1, \dots, y_{n_k}/x_{n_k}]$ bezeichnet \tilde{ai} , wobei jedes Paar (x_i, s) durch (y_i, s) ersetzt wird.

⁴ \tilde{AI} des rechten beziehungsweise linken Sohnknotens im Syntaxbaum.

Kapitel 4. Übersetzung zwischen HRGen und SLF

Führe Schritt 2 nacheinander für alle Prädikate aus und wiederhole dies, bis die Informationen \tilde{AI}_{σ_j} sich nicht mehr ändern.

Teil 2: Nun ist zu ermitteln, welche Konfigurationen von Variablen-Selektor- bzw. Knoten-Selektor-Paaren insgesamt definiert werden können. Dazu wird der folgende Algorithmus nacheinander auf allen Prädikatdefinitionen sowie auf dem **in**-Teil der Formel, φ , ausgeführt. Die nun betrachteten Analyseinformationen haben die Form $AI \subseteq 2^{\text{Var} \times \text{sel}_\Sigma}$.

1. Ersetze alle Variablen x innerhalb der Formel durch Variablenmengen $\{x\}$.
2. Ziehe alle Gleichheitsassertionen innerhalb der Konjunktionkette, in der sie auftreten an den Anfang, so dass die Form $M_1 = M_1' \wedge \dots \wedge M_p = M_p' \wedge \hat{\gamma}$ entsteht und $\hat{\gamma}$ kein „=" enthält.
3. Laufe von vorne durch die Formel. Für alle $M_1 = M_2 \wedge \gamma$, lösche „ $M_1 = M_2 \wedge$ “ und ersetze in γ alle Vorkommen von M_1 und M_2 durch $M_1 \cup M_2$.
4. Erzeuge den Parsingbaum der Formel. Führe eine Bottom-Up-Analyse durch und berechne die Informationen AI für jeden Knoten des Baums wie folgt:

Blatt $\{y_1, \dots, y_n\}.s \mapsto el$:	$AI := \{(y_1, s), \dots, (y_n, s)\}$
Blatt $\sigma'_k(y_1, \dots, y_{n_k})$:	$AI := \{\tilde{ai}[y_1/x_1, \dots, y_{n_k}/x_{n_k}] \mid \tilde{ai} \in \tilde{AI}_{\sigma'_k}\}$
Innerer Knoten „*“:	a. Prüfe: $\forall ai_l \in AI_{\text{links}}, ai_r \in AI_{\text{rechts}} : ai_l \cap ai_r = \emptyset$ nein \rightarrow <i>Fail!</i> (Selektor doppelt def.) b. $AI := \{ai_l \cup ai_r \mid ai_l \in AI_{\text{links}}, ai_r \in AI_{\text{rechts}}\}$
Innerer Knoten „ \vee “:	$AI := AI_{\text{links}} \cup AI_{\text{rechts}}$
Innerer Knoten „ $\exists x$ “:	$AI := \{ai \setminus (\{x\} \times \text{sel}_\Sigma) \mid ai \in AI_{\text{sohn}}\}$ (lösche x)
5. \rightarrow *Success!*

Der Algorithmus in Teil 2 hat unter Vernachlässigung der Größe $|\text{sel}_\Sigma|$ ein Laufzeitverhalten von $\mathcal{O}(2^{2l})$, wobei l der Länge der Formel entspricht. Insbesondere die elementweise Prüfung und Vereinigung der AI auf der Domäne $2^{\text{Var} \times \text{sel}_\Sigma}$ führen zu dieser Laufzeit. In Teil 1 terminiert die Fixpunktiteration über Schritt 2, da mit der leeren Konfiguration $\tilde{AI}_{\sigma_j} = \{\emptyset\}$ begonnen wird und die Analyseinformation oder deren Elemente bei jeder Iteration wachsen. Daher wird stets in endlicher Zeit ein Fixpunkt erreicht. Das Laufzeitverhalten liegt in derselben Größenordnung wie das von Teil 2. Die Anzahl der Variablen ist zwar auf den Rang der Prädikate beschränkt, dafür wird jedoch mehrmals iteriert. Geht man von l Iterationen aus, so liegt seine Laufzeit in jedem Fall in $\mathcal{O}(2^{3l})$. Es sei jedoch betont, dass diese Angaben stark vereinfachte und sehr pessimistische Abschätzungen widerspiegeln, da eine Formel der Länge l nicht l Prädikate mit je l Parametern und l Selektoren pro Variable enthalten kann.

Terminiert der Algorithmus auf einer Formel φ mit *Success!*, so ist die DSG-Eigenschaft für $g[\varphi]$ erfüllt. Für Formeln, deren Grammatik unproduktiv ist, die also Nichtterminale enthält, deren Ableitungen nie terminiert, wertet er jedoch möglicherweise zu *Fail!* aus, obwohl die zugehörige Grammatik eine DSG ist. Warum dies so ist, soll das folgende Beispiel verdeutlichen.

Beispiel 4.11. Wir betrachten die Formel

$$\varphi = \mathbf{let} \ \sigma(x_1) = x_1.s \mapsto \mathbf{nil} * \sigma(x_1) \ \mathbf{in} \ \sigma(y)$$

Die Sprache von $g[\varphi]$ und die konkrete Hülle von X_σ sind offenbar leer, da sich von X_σ^\bullet kein konkreter Graph ableiten lässt (alle ableitbaren Graphen enthalten das Nichtterminal X_σ): $\mathcal{L}(g[\varphi]) = L_p(X_\sigma^\bullet) = \emptyset$. $g[\varphi]$ ist demnach offensichtlich eine DSG. Der Algorithmus terminiert dennoch mit *Fail!*, da der Fixpunkt für AI_σ mit der *AI* des Blattes $x_1.s \mapsto \mathbf{nil}$ verglichen wird. Beide lauten $\{(x_1, s)\}$. \triangle

KOROLLAR 4.4 Algorithmus 4.1 stellt eine hinreichende Bedingung für die DSG-Eigenschaft dar, jedoch keine notwendige. Für auswertbare Formeln (Formeln, deren synthetisierte Grammatik produktiv ist, siehe Abschnitt 5.1) ist seine Auswertung zu *Success!* auch notwendige Bedingung.

4.3. KORREKTHEIT DER SYNTHESEFUNKTION

Die im vorangegangenen Abschnitt eingeführte Synthesefunktion generiert eine Startgraph-Grammatik aus einer SLF-Formel. Diese soll eine analoge „Bedeutung“ haben, wie die Ausgangsformel. In Abschnitt 4.1 wurde durch die Funktion α definiert, wie sich ein Paar aus einem Heap h und einer Variableninterpretation i eindeutig als Hypergraph darstellen lässt. Dieser soll genau dann in der Sprache einer synthetisierten Grammatik liegen, wenn h und i die zur Synthese verwendete Formel erfüllen. Um die Korrektheit von $g[\cdot]$ zu gewährleisten, sind drei Anforderungen an die verwendeten Formeln zu erfüllen, die in Abschnitt 4.3.1 formuliert und erläutert werden. Im Anschluss führt Abschnitt 4.3.2 einige Begriffe ein, die für den im darauffolgenden Abschnitt 4.3.3 durchgeführten Korrektheitsbeweis benötigt werden.

4.3.1. ANFORDERUNGEN AN SLF-FORMELN FÜR DIE GRAMMATIKSYNTHESE

Die syntaktische Einschränkung der Separation Logic auf SLF ermöglicht die Synthese von Grammatiken aus Formeln. Die im vorangegangenen Abschnitt vorgestellte Synthesefunktion ist in der Lage, jede Formel $\varphi \in \text{SLF}$ in eine Grammatik zu übersetzen. Damit diese jedoch eine äquivalente Semantik hat, muss die verwendete Formel zusätzlich zum syntaktischen Aufbau gemäß Definition 3.6 drei Anforderungen erfüllen:

- C1:** Wenn ein Objekt definiert ist, dann ist auch stets jedes Feld dieses Objektes definiert.
- C2:** Jede Variable x , die auf der rechten Seite einer Zeigerassertion oder als Argument in einem Prädikataufruf steht, repräsentiert ein existentes Objekt. Dies kann durch das Auftreten an der linken Seite einer Zeigerassertion (z.B. $x.s \mapsto y$) oder durch eine Gleichheitsassertion (z.B. $x = y$) gegeben sein. Im letzteren Fall muss y jedoch ebenfalls auf eine der beiden Weisen mit einem existenten Objekt assoziiert werden.
- C3:** Prädikatdefinitionen enthalten (von den Parametern abgesehen) keine freien Variablen.

Die erste Bedingung, **C1**, ist bei der Argumentation innerhalb des Korrektheitsbeweises notwendig. Dodds verwendet in seinem Separation-Logic-Fragment ausschließlich ternäre Zeigerassertionen der Form $x \mapsto el_1, el_2$, so dass stets alle (beiden) Felder von x definiert werden (**nil** ist dabei – auch im Sinne von **C1** – ein gültiger Wert). Ein in einer Formel φ nicht-definiertes Feld $x.s$ wird von $h[\cdot]$ ignoriert, es wird also keine Selektorkante dafür angelegt. Dies entspricht in unserem Ansatz jedoch einem impliziten Selektor auf **nil**. Ein Paar $h, i \in \text{He} \times \text{Int}$ mit $h, i \models \varphi$ sieht für $x.s$ keine Speicherzelle vor: $\llbracket x \rrbracket i + \text{cn}(s) \notin \text{dom}(h)$. Die Graphfunktion α weist dem resultierenden Hypergraphen also ebenfalls keine Selektorkante für $x.s$ zu, er liegt demnach also in der Sprache von $g[\varphi]$. Allerdings ist auch $\alpha(h', i)$ mit $\llbracket x \rrbracket i + \text{cn}(s) \in h'$ und $h'(\llbracket x \rrbracket i + \text{cn}(s)) = \mathbf{nil}$ ein gültiger Graph dieser Sprache, obwohl offensichtlich $h', i \not\models \varphi$ gilt. Somit ist $h, i \models \varphi$ in diesem Fall offenbar keine notwendige Bedingung mehr. Da nicht definierte und auf **nil** verweisende Felder offensichtlich für Graphen dieselbe Bedeutung haben, in SLF jedoch nicht, muss die Mächtigkeit von SLF durch **C1** entsprechend angepasst werden.

Im Rahmen dieser Arbeit wird ausschließlich die Struktur des Heaps betrachtet. Eine Aussage über eine auf dem Heap gespeicherte Adresse innerhalb einer Formel muss sich daher immer auf ein existierendes Objekt oder auf **nil** beziehen. Die Bedingung **C2** fordert deshalb, dass alle verwendeten Variablen für ein Objekt stehen. Dies kann durch die Forderung der Gleichheit zu einer Variable, die diese Bedingung bereits erfüllt, gewährleistet sein, oder aber durch die Definition eines der Felder des Objektes. Wegen **C1** müssen in letzterem Fall sogar alle Felder definiert werden. Die Graphsynthesefunktion $h[\cdot]$ erzeugt zu jeder Zeigerassertion $x.s \mapsto y$ zwei Knoten, nimmt also an, dass es ein Objekt zu y gibt. Für den Fall, dass y in der Formel jedoch nicht definiert ist, existiert in einem Heap, der sie erfüllt, auch keine Speicherzelle für y . Die Graphfunktion α legt in diesem Fall entsprechend keinen Knoten an. Die Selektorkante zu s wird dennoch erzeugt, ihre Inzidenzen verweisen jedoch auf einen nichtexistenten Knoten. Der Graph ist dann nicht in der Sprache der synthetisierten Grammatik, $h, i \models \varphi$ demnach keine hinreichende Bedingung mehr. Das folgende Beispiel soll dies verdeutlichen:

Beispiel 4.12. Sei $\varphi = x.s \mapsto y$. Es gilt

$$h[\varphi] = \left\langle \left\{ \begin{array}{c} \{x\} \\ \circ \end{array} \xrightarrow{s} \begin{array}{c} \{y\} \\ \circ \end{array} \right\} \right\rangle \implies g[\varphi] = \left\langle \left\{ \left[\begin{array}{c} x \\ \square \end{array} \right] \xrightarrow{1} \circ \xrightarrow{s} \circ \xrightarrow{1} \left[\begin{array}{c} y \\ \square \end{array} \right] \right\}, \emptyset \right\rangle$$

Für $i = [x \mapsto 3, y \mapsto 10], h = [3 \mapsto 10]$ gilt offenbar:

$$h, i \models \varphi$$

$\alpha(h, i)$ enthält nur einen Knoten: $V = \{3\}$, es gibt jedoch eine Kante e_{3s} mit $\text{att}(e_{3s}) = 3h(3 + 0) = 3 \cdot 10$. Da 10 jedoch kein Knoten ist, gilt offenbar $\text{att}(e_{3s}) \notin V^*$, $\alpha(h, i)$ ist also streng genommen kein valider Hypergraph. Selbst unter Vernachlässigung dieses Problems liegt $\alpha(h, i)$ offenbar nicht in $\mathcal{L}(g[\varphi])$. Für α gehört y also nicht zum Heap, für $h[\cdot]$ jedoch schon. Um **C2** zu genügen, kann $\varphi' = x.s \mapsto y * y.s \mapsto \mathbf{nil}$ anstelle von φ verwendet werden. α und $h[\cdot]$ sind dann wieder kompatibel. \triangle

Die letzte Bedingung, **C3**, schließlich rührt daher, dass bei der Graphsynthese keine Variablenkanten angelegt werden. Dies geschieht erst durch die Grammatiksynthese auf den Startgraphen. In Grammatiken sind Variablenkanten auf rechten Regelseiten unerwünscht, da sie schnell zur Verletzung der HC-Bedingung für eindeutige Variablen führen. Außerdem ist die Abstraktion von Variablen mittels HAGs nicht sinnvoll. Würde man Variablenkanten auf rechten Regelseiten erlauben, so wären auch mehrere Variablenkanten an einem Knoten denkbar. Dazu wiederum werden auf Seiten der Separation Logic Gleichheitsassertionen in Prädikatdefinitionen benötigt (Siehe Abschnitt 4.4), die SLF jedoch nicht vorsieht. Eine derartige Erweiterung macht Prädikate insgesamt sehr unhandlich und erschwert die Definition und Untersuchung der in Kapitel 5 untersuchten Eigenschaften erheblich.

Der im Folgenden skizzierte Algorithmus prüft **C1** und **C2**. Er arbeitet dabei sehr ähnlich zu Algorithmus 4.1, verwendet jedoch einfachere Analyseinformationen der Form $AI \subseteq \text{Var} \times \text{sel}_\Sigma$ und für die Fixpunktiteration $\tilde{AI} \subseteq \{x_1, \dots, x_n\} \times \text{sel}_\Sigma$, berücksichtigt also nicht den Kontext der definierten Selektoren, was die Laufzeit verbessert.

ALGORITHMUS 4.2 (C1 & C2) Gegeben sei die Formel

$$\text{let } \sigma_1(x_1, \dots, x_{n_1}) = \psi_1, \dots, \sigma_m(x_1, \dots, x_{n_m}) = \psi_m \text{ in } \varphi \in \text{SLF}$$

Teil 1: Fixpunktiteration über alle Prädikate.

Sei $1 \leq j \leq m$ und $\sigma_j(x_1, \dots, x_{n_j}) = \psi_j$. Es werden Analyseinformationen der Form $N \subseteq \text{PS}_j := \{x_1, \dots, x_{n_j}\} \times \text{sel}_\Sigma$ betrachtet. Ist ein Paar (x_i, s) in N_{σ_j} enthalten, bedeutet dies, dass σ_j den Selektor s des i -ten Parameters stets definiert.

1. $\tilde{AI}_{\sigma_j} = \{x_1, \dots, x_{n_j}\} \times \text{sel}_\Sigma$ (alle Selektoren werden initial erzeugt)
2. Erzeuge den Parsingbaum der Formel. Führe eine Bottom-Up-Analyse durch und berechne die Informationen \tilde{AI} für jeden Knoten des Baums wie folgt:

Blatt $y.s \mapsto el$:	$\tilde{AI} := \{(y, s)\} \cap \text{PS}_j$
Blatt $\sigma'_k(y_1, \dots, y_{n_k})$:	$\tilde{AI} := \tilde{AI}_{\sigma'_k} [y_1/x_1, \dots, y_{n_k}/x_{n_k}] \cap \text{PS}_j$
Innerer Knoten „*“:	$\tilde{AI} := \tilde{AI}_{\text{links}} \cup \tilde{AI}_{\text{rechts}}$
Innerer Knoten „∨“:	$\tilde{AI} := \tilde{AI}_{\text{links}} \cap \tilde{AI}_{\text{rechts}}$
Innerer Knoten „∃x“:	$\tilde{AI} := \tilde{AI}_{\text{Sohn}}$ (wird ignoriert)
Wurzel (nach *, ∨ oder ∃):	$\tilde{AI}_{\sigma_j} := \tilde{AI}$

Führe Schritt 2 nacheinander für alle Prädikate aus und wiederhole dies, bis die Informationen \tilde{AI}_{σ_j} sich nicht mehr ändern.

Teil 2: Hier wird geprüft, ob tatsächlich alle Variablen und Selektoren erzeugt werden. Dazu wird im Syntaxbaum bei jedem \exists -Quantor der garantierte Selektorbestand der entsprechenden Variablen geprüft. Dasselbe wird am Ende für die freien Variablen durchgeführt. Die folgenden Schritte werden nacheinander auf allen Prädikatdefinitionen sowie auf φ , dem **in**-Teil der Formel, ausgeführt. Die hier betrachteten Analyseinformationen haben die Form $AI \subseteq \text{Var} \times \text{sel}_\Sigma$.

Kapitel 4. Übersetzung zwischen HRGen und SLF

1. Ersetze alle Variablen x innerhalb der Formel durch Variablenmengen $\{x\}$.
2. Ziehe alle Gleichheitsassertionen innerhalb der Konjunktionskette, in der sie auftreten an den Anfang, so dass die Form $M_1 = M_1' \wedge \dots \wedge M_p = M_p' \wedge \hat{\gamma}$ entsteht und $\hat{\gamma}$ kein „=“ enthält.
3. Laufe von vorne durch die Formel. Für alle $M_1 = M_2 \wedge \gamma$, lösche „ $M_1 = M_2 \wedge$ “ und ersetze in γ alle Vorkommen von M_1 und M_2 durch $M_1 \cup M_2$.
4. Erzeuge den Parsingbaum der Formel. Führe eine Bottom-Up-Analyse durch und berechne die Informationen AI für jeden Knoten des Baums wie folgt:

Blatt $\{y_1, \dots, y_n\}.s \mapsto el$:	$AI := \{(y_1, s), \dots, (y_n, s)\}$
Blatt $\sigma'_k(y_1, \dots, y_{n_k})$:	$\tilde{AI} := \tilde{AI}_{\sigma'_k}[y_1/x_1, \dots, y_{n_k}/x_{n_k}]$
Innerer Knoten „*“:	$AI := AI_{\text{links}} \cup AI_{\text{rechts}}$
Innerer Knoten „ \vee “:	$AI := AI_{\text{links}} \cap AI_{\text{rechts}}$
Innerer Knoten „ $\exists x$ “:	a. Prüfe: $(\{x\} \times \text{sel}_\Sigma) \subseteq AI$ nein \rightarrow <i>Fail!</i> b. $AI := AI_{\text{sohn}} \setminus (\{x\} \times \text{sel}_\Sigma)$
5. Prüfe: $AI_{\text{Wurzel}} = \text{FV}(\varphi) \times \text{sel}_\Sigma$
ja \rightarrow *Success!*
nein \rightarrow *Fail!*

Gibt der Algorithmus an einer beliebigen Stelle *Fail!* zurück, dann existiert mindestens ein Knoten, der nicht (vollständig) definiert wurde.

Teil 2 des Algorithmus hat wegen Schritt 3 quadratische Laufzeit über die Länge der Formel l . Der Aufbau des Parsingbaumes und die Bottom-Up-Analyse liegt in $\mathcal{O}(l \log(l))$. Die Fixpunktiteration über Schritt 2 terminiert, da jeweils mit dem größten \tilde{AI}_{σ_j} begonnen wird und dieses bei jeder Iteration nur kleiner werden kann. Dies ist der Fall, da die \tilde{AI}_{σ_j} erst nach vollständiger Abarbeitung des Syntaxbaumes aktualisiert werden und es keine Negation gibt. Dementsprechend können keine (x, s) -Paare mehr in ein \tilde{AI}_{σ_j} aufgenommen werden, die zuvor bereits daraus gelöscht wurden. Es wird deshalb stets in endlicher Zeit ein Fixpunkt erreicht. Da pro Iteration mindestens ein Selektor oder eine Variable aus jedem \tilde{AI}_{σ_j} entfernt wird, und die Anzahl der Selektoren konstant ist, hat Teil 1 die Komplexität $\mathcal{O}(l \cdot \log(l) \cdot m)$, wobei m die Anzahl der Prädikate darstellt. Wegen $m < l$ ergibt sich insgesamt die Komplexitätsklasse $\mathcal{O}(l^2 \log(l))$.

Satz 4.5 (Komplexität von C1, C2 & C3) Die Eigenschaften **C1**, **C2** & **C3** sind in $\mathcal{O}(l^2 \log(l))$ überprüfbar, wobei l die Länge der Formel repräsentiert.

Beweis. **C1** & **C2**: Verwendung von Algorithmus 4.2.

C3: Zur Prüfung von **C3** laufe über jede Prädikatdefinition und speichere Analyseinformationen der Form $Q \subseteq \text{Var}$. Bei einem Quantor $\exists x$ füge x zu Q hinzu, am Ende von dessen Gültigkeitsbereich, lösche x wieder. Wird eine Variable verwendet, die weder Parameter, noch in Q ist, brich mit „Fail!“ ab. Die Laufzeit ist linear über l . \square

4.3.2. GRAPHHAUSWERTUNG UND KORRESPONDENZ

Der Korrektheitsbeweis von $g[\cdot]$ ist modular aufgebaut. Dabei wird zunächst die Korrektheit der Graphsynthese gezeigt und darauf aufbauend die der Produktionsregel- und Grammatiksynthese. Synthetisierte Graphen werden zum einen als Startgraphen, zum anderen aber auch als rechte Regelseiten eingesetzt. Letztere enthalten grundsätzlich keine Variablenkanten. Um dennoch eine Verbindung zwischen (h, i) -Paaren, $(h, i) \in \text{He} \times \text{Int}$, und synthetisierten Hypergraphen herstellen zu können, wird neben der bereits eingeführten Graphfunktion α eine weitere Funktion α_t eingeführt. Während α zu allen in i registrierten Variablen, die auf Knoten des Graphen verweisen, eine Variablenkanten anlegt, erzeugt α_t stattdessen ein zu i passendes Tagging.

DEFINITION 4.12 (α_t) Die Funktion α_t ist definiert durch

$$\begin{aligned} \alpha_t : \text{He} \times \text{Int} &\rightarrow \text{HC}_{\Sigma_N}^T \\ \alpha_t(h, i) &= (H, t) \end{aligned}$$

H entspricht dabei $\alpha(h, i)$ unter Ausschluss der Variablenkanten (vgl. Def. 4.1, Zeile 4.5):

$$\begin{aligned} H &= (V_{\alpha(h,i)}, E_H, \text{att}_{\alpha(h,i)} \upharpoonright E_H, \text{lab}_{\alpha(h,i)} \upharpoonright E_H, \varepsilon) \\ E_H &= \{e_{v,s} \mid v \in V_H, s \in \text{sel}_{\Sigma}, h(v + \text{cn}(s)) \neq \mathbf{nil}\} \end{aligned}$$

Das Tagging wird aufgrund der Nutzung von Speicheradressen als Knoten als Umkehrfunktion von i definiert, die auf die Knoten des Graphen reduziert ist.

$$t = (i^{-1} \upharpoonright V_{\alpha(h,i)})$$

Anders ausgedrückt gilt als Beziehung zwischen i und t

$$x \in t(v) \iff i(x) = v.$$

Der Beweis der Korrektheit von $h[\cdot]$ (Lemma 4.8) wird induktiv über den strukturellen Aufbau der Formel geführt. Da $h[\cdot]$ bei der Übersetzung eines Prädikataufrufes oder einer Zeigerassertion alle inzidenten Knoten erzeugt, benötigt der Beweis eine Graphfunktion, die auch die Bildmenge des Heaps ($\text{img}(h)$) berücksichtigt.

DEFINITION 4.13 ($\tilde{\alpha}_t$) Die um das Bild eines Heaps erweiterte, getaggte Graphfunktion ist gegeben durch

$$\begin{aligned} \tilde{\alpha}_t : \text{He} \times \text{Int} &\rightarrow \text{HC}_{\Sigma_N}^T \\ \tilde{\alpha}_t(h, i) &= (H, t) \end{aligned}$$

H ist analog zu α_t definiert, wobei hier $\text{img}(h)$ in die Knotenmenge mit aufgenommen wird:

$$\begin{aligned} H &= (V, E_H, \text{att}_{\alpha(h,i)} \upharpoonright E_H, \text{lab}_{\alpha(h,i)} \upharpoonright E_H, \varepsilon) \\ V &= V_{\alpha(h,i)} \cup \text{img}(h) \setminus \{\mathbf{nil}\} \\ E_H &= \{e_{v,s} \mid v \in V_H, s \in \text{sel}_{\Sigma}, h(v + \text{cn}(s)) \neq \mathbf{nil}\} \\ t &= (i^{-1} \upharpoonright V) \end{aligned}$$

Kapitel 4. Übersetzung zwischen HRGen und SLF

Damit ist $\tilde{\alpha}_t(h, i)$ für jeden Heap h ein gültiger Hypergraph und nicht ausschließlich für solche bei denen $(\text{img}(h) \setminus \{\text{nil}\}) \subseteq \text{dom}(h)$ gilt. Für Modelle von Formeln, die **C2** genügen, ist dies jedoch ohnehin gegeben, was zum folgenden Lemma führt.

LEMMA 4.6 (α_t und $\tilde{\alpha}_t$) Sei $\varphi \in \text{SLF}$ eine Formel, die **C2** genügt und $h, i \models \varphi$. Dann gilt

$$\alpha_t(h, i) = \tilde{\alpha}_t(h, i)$$

Beweis. Die Bedingung **C2** fordert, dass alle Variablen x , die in φ verwendet werden, durch eine Gleichheitsassertion oder durch Setzen eines ihrer Felder definiert werden. In beiden Fällen ist $\llbracket x \rrbracket i \in \text{dom}(h)$ als Basisadresse eines Objektes enthalten und somit auch in $V_{\alpha(h, i)}$ und $V_{\tilde{\alpha}_t(h, i)}$. Alles Weitere folgt direkt aus den Definitionen 4.12 und 4.13. \square

Weitere Werkzeuge für den Beweis sind die eng miteinander verwandten Begriffe *Graphumgebung* und *Graphauswertung*.

DEFINITION 4.14 (*Graphumgebung*) Sei N eine Menge von Nichtterminalsymbolen. Eine *Graphumgebung* $\lambda : N \rightarrow 2^{\text{HG}_{\Sigma_N}}$ ordnet jedem Nichtterminalsymbol eine Menge konkreter Hypergraphen zu.

DEFINITION 4.15 (*Graphauswertung*) Sei H ein Hypergraph und λ eine Graphumgebung. Die *Auswertung* von H unter λ , notiert mit $\kappa(\lambda, H)$, ist die Menge aller Terminalgraphen, die durch Ersetzung aller Nichtterminalkanten e in H durch je einen Hypergraphen $H' \in \lambda(\text{lab}(e))$ entsteht.

$$\kappa(\lambda, H) = \{H[H'/e] \mid \text{lab}(e) \in N \wedge H' \in \lambda(\text{lab}(e))\}$$

Die Graphauswertung entspricht in vielerlei Hinsicht der in Kapitel 2 vorgestellten konkreten Hülle L_p , die ebenfalls jedem Hypergraphen eine Menge an daraus ableitbaren Terminalgraphen zuordnet. Der Unterschied liegt im Wesentlichen in der Betrachtungsweise der Ableitung. Während L_p beliebige in Terminalgraphen endende Ableitungen durch Regeln in P zulässt, betrachtet $\kappa(\lambda, H)$ nur diejenigen Ersetzungen, die durch λ definiert sind. Diese Eigenschaft spielt in der Beweisstruktur von Dodds [Dod08], auf die hier zurückgegriffen wird, eine tragende Rolle.

HINWEIS: Für die Graphumgebung wurde bewusst das Symbol λ gewählt, um eine Verwechslung mit der konkreten Hülle L_p auszuschließen. In [Dod08] wird sie (dort: *Graph Environment*) durch das Symbol L dargestellt.

Der Bezug zum Sprachbegriff aus Kapitel 2 wird durch Lemma 4.7 formalisiert.

LEMMA 4.7 (*Sprache und Graphauswertung*) Sei $G = \langle \mathfrak{J}, P \rangle \in \text{HRG}_{\Sigma_N}$ und λ diejenige Graphumgebung sodass für alle $X \in N$ gilt $\lambda(X) = L_p(X)$. Dann gilt

$$\mathcal{L}(G) = \kappa(\lambda, \mathfrak{J})$$

Beweis. Folgt direkt aus den Definitionen 2.11 und 4.15. \square

Der im Folgenden eingeführte Begriff der Korrespondenz stellt eine Beziehung zwischen Graphumgebungen und Prädikatinterpretationen her.

DEFINITION 4.16 (Korrespondenz) Sei $H \in \text{HG}_{\Sigma_N}$ ein Hypergraph mit n externen Knoten, $h \in \text{He}$ ein Heap und $\vec{l} = l_1, \dots, l_n$ eine Sequenz von Heapadressen. H korrespondiert mit (\vec{l}, h) , wenn $(V_H, E_H, \text{att}_H, \text{lab}_H, \varepsilon) = \alpha(h, i_0)$ und für alle $j, 1 \leq j \leq n$ gilt $\text{ext}_H(j) = l_j$.

Eine Prädikatinterpretation η korrespondiert mit einer Graphumgebung λ , wenn

1. $\text{dom}(\eta) = \text{dom}(\lambda)$
2. Für alle $\sigma \in \text{dom}(\eta)$ existiert für jedes $(\vec{x}, h) \in \eta(\sigma)$ ein korrespondierender Hypergraph $H \in \lambda(X_\sigma)$
3. Für alle $X_\sigma \in \text{dom}(\lambda)$ existiert für jedes $H \in \lambda(X_\sigma)$ ein korrespondierendes Paar $(\vec{x}, h) \in \eta(\sigma)$

Im Korrektheitsbeweis werden Graphumgebung und Prädikatinterpretation für eine Formel und die daraus synthetisierte Grammatik durch eine parallel ausgeführte Fixpunktiteration ermittelt. Auf diese Weise wird die Korrespondenz zwischen beiden gewährleistet.

4.3.3. KORREKTHEITSBEWEIS

Die Semantikerhaltung von $g[\![\cdot]\!]$ definiert sich über die Relation \models auf der SLF- und \mathcal{L} auf der HRG-Seite, sowie der Funktion α , die die Modellierung von Heaps durch Graphen definiert. Es ist also zu zeigen:

$$h, i, \eta_0 \models \varphi \iff \alpha(h, i) \in \mathcal{L}(g[\![\varphi]\!])$$

Ein Heap-Interpretations-Paar soll eine Formel also genau dann erfüllen, wenn der dazugehörige Hypergraph in der Sprache der aus dieser Formel synthetisierten Grammatik liegt. η_0 repräsentiert dabei die leere Prädikatsinterpretation; wir gehen also davon aus, dass alle verwendeten Prädikate innerhalb der Formel φ definiert werden.

Bei Dodds ist die Semantikerhaltung für die dort definierte Funktion $g[\![\cdot]\!]$ in mehreren Schritten (Lemmata) bewiesen [Dod08]. Das in dieser Arbeit verwendete Graphgrammatik-Modell, das verwendete SL-Fragment SLF und dementsprechend auch die Synthesefunktion $g[\![\cdot]\!]$ weichen jedoch in einigen Punkten von den dort verwendeten Strukturen ab. Es ist dennoch nicht nötig, den gesamten Beweis neu zu formulieren, da die meisten Änderungen nur den **in**-Teil der ohne Einschränkung flachen Formeln betreffen. Die Prädikatdefinitionen (der **let**-Teil) bleiben weitestgehend unberührt. Eine Ausnahme stellt das Heapmodell dar, das formelseitig durch binäre Zeigerassertionen der Form $x \mapsto el$ anstelle der bei Dodds verwendeten ternären $x \mapsto el_1, el_2$ -Assertionen repräsentiert wird. Im ersten Teilbeweis muss daher ein Fall in einer strukturellen Induktionsverankerung ersetzt werden.

Zur besseren Nachvollziehbarkeit werden nachfolgend alle Lemmata zum Beweis der Semantikerhaltung aufgeführt.

LEMMA 4.8 Sei $\varphi \in \text{SLF}$ **let**-frei und enthalte weder „ \wedge “ noch „ $=$ “, $h \in \text{He}$ ein Heap, $i \in \text{Int}$ eine Variableninterpretation und $\eta \in \text{PI}$ eine Prädikatinterpretation. Sei λ die mit η korrespondierende Graphumgebung. Dann gilt

$$h, i, \eta \models \varphi \implies \tilde{\alpha}_t(h, i) \in \kappa(\lambda, h[\![\varphi]\!])$$

Kapitel 4. Übersetzung zwischen HRGen und SLF

Beweis. (durch strukturelle Induktion über den Aufbau von φ)

Fall: $x.s \mapsto el$

Seien h, i, η so, dass $h, i, \eta \models x.s \mapsto el$ gilt. Es folgt

$$\text{dom}(h) = (\llbracket x \rrbracket i + \text{cn}(s))$$

und

$$h(\llbracket x \rrbracket i + \text{cn}(s)) = \llbracket el \rrbracket i.$$

Der getaggte Hypergraph zu h und i ist entsprechend:

$$\tilde{\alpha}_t(h, i) = \begin{array}{ccc} \bigcirc & \xrightarrow{s} & \bigcirc \\ \{x\} & & \{el\} \end{array}$$

Ferner ist

$$\begin{aligned} \kappa(\lambda, h \llbracket x.s \mapsto el \rrbracket) &= h \llbracket x.s \mapsto el \rrbracket = \{\Downarrow(H, t)\} \\ &= \left\{ \begin{array}{ccc} \bigcirc & \xrightarrow{s} & \bigcirc \\ \{x\} & & \{el\} \end{array} \right\} \end{aligned}$$

Offenbar gilt daher

$$\tilde{\alpha}_t \in \kappa(\lambda, h \llbracket x.s \mapsto el \rrbracket).$$

Alle anderen Fälle sind analog zu Proposition 7.3 aus [Dod08], wenn darin α_t durch $\tilde{\alpha}_t$ ersetzt wird. \square

Daraus folgt direkt:

LEMMA 4.9 Sei $\varphi \in \text{SLF}$ **let**-frei, enthalte weder „ \wedge “ noch „ $=$ “ und genüge **C1** und **C2**. Seien außerdem $h \in \text{He}$ ein Heap, $i \in \text{Int}$ eine Variableninterpretation und $\eta \in \text{PI}$ eine Prädikatinterpretation. Sei λ die mit η korrespondierende Graphumgebung. Dann gilt

$$h, i, \eta \models \varphi \iff \alpha_t(h, i) \in \kappa(\lambda, h \llbracket \varphi \rrbracket)$$

Beweis. Wegen Lemma 4.6 gilt für h und i , das $\alpha_t(h, i) = \tilde{\alpha}_t(h, i)$. Nach Lemma 4.8 folgt die Behauptung für „ \implies “. Da **C1** gegeben ist, kann h keine, in φ nicht spezifizierten **nil**-Felder mehr enthalten, so dass nach Definition von α_t beziehungsweise α „ \longleftarrow “ ebenfalls gilt. \square

Dieses Lemma stellt die Basis des Beweises der Semantikerhaltung von $g \llbracket \cdot \rrbracket$ dar, indem es einen Bezug zwischen der Modell- und der Sprachrelation darstellt. Die bereits gegebene Korrespondenz zwischen der η und λ ist dabei allerdings noch Voraussetzung.

Der folgende Satz erfasst den Zusammenhang zwischen **let**-freien Formeln und der Sprache der daraus synthetisierten Grammatik.

SATZ 4.10 Sei $\varphi \in \text{SLF}$ **let**-frei, enthalte weder „ \wedge “ noch „ $=$ “ und genüge **C1** und **C2**. Sei $h \in \text{He}$ ein Heap. Dann gilt

$$h, i_0, \eta_0 \models \varphi \iff \alpha(h, i_0) \in \mathcal{L}(g \llbracket \varphi \rrbracket).$$

4.3. Korrektheit der Synthesefunktion

Beweis. Siehe Theorem 7.4 in [Dod08], Seite 153.

Hierzu sei erwähnt, dass die Heapeigenschaft von $\alpha(h, i)$ für alle $h \in \text{He}, i \in \text{Int}$ implizit gegeben ist (Lemma 4.1), eine Normierung von $g[\![\varphi]\!]$, wie sie bei Dodds durchgeführt wird, also nicht notwendig ist⁵.

Der Beweis verwendet Lemma 4.9 als Grundlage. □

Lassen wir Prädikatdefinitionen (Γ) zu, müssen wir zunächst die Korrespondenz zwischen der von Γ definierten Prädikatinterpretation und der von $r[\![\Gamma]\!]$ definierten Graphumgebung sicherstellen.

LEMMA 4.11 Sei $\text{let } \Gamma \text{ in } \varphi$ eine flache SLF-Formel, die **C1**, **C2** & **C3** genügt. Der kleinste Fixpunkt von

$$f_{\eta, \Gamma}: k \mapsto [\sigma \mapsto \{(\vec{l}, h) \mid h, [\vec{x} \mapsto \vec{l}], \eta[\beta \mapsto k(\beta)]_{\beta \in \text{dom}(\Gamma)} \models \psi\}_{\sigma \in \text{dom}(\Gamma)}]$$

(wobei ψ die syntaktische Definition von σ in Γ ist) korrespondiert mit der Graphumgebung λ , die durch den Regelsatz $r[\![\Gamma]\!]$ bestimmt ist.

Beweis. Siehe Lemma 7.5 in [Dod08], Seite 154. □

Für die allgemeine Form von SLF-Formeln, wie sie bei Dodds & Plump [DP08b] verwendet werden, bringt der folgende Satz die Ergebnisse von Lemma 4.9 und 4.11 zusammen und überträgt diese analog zu Satz 4.10 auf die Grammatiksynthesefunktion $g[\![\cdot]\!]$. Nach Lemma 4.11 korrespondiert die Prädikatinterpretation, die beim Auswerten des **let**-Teils der Formel (Γ) zunächst entsteht und anschließend im **in**-Teil angewandt wird, mit der Graphumgebung, die durch den Regelsatz definiert ist, der durch Synthese derselben Formel entsteht. Daher können wir nun auf Lemma 4.9 aufbauend, auf der linken Seite der Äquivalenzbeziehung die leere Prädikatumgebung η_0 einsetzen, die dann durch Auswertung von Γ „gefüllt“ wird, und analog dazu auf der rechten Seite die Sprachfunktion \mathcal{L} verwenden, die die Prädikatumgebung aus dem Regelsatz der synthetisierten Grammatik bildet. Der folgende Satz erfasst zunächst die Korrektheit für Formeln ohne freie Variablen und daher die leere Variableninterpretation i_0 .

SATZ 4.12 Sei $\varphi' = \text{let } \Gamma \text{ in } \varphi$ eine flache SLF-Formel gemäß **C1**, **C2** & **C3** und $h \in \text{He}$ ein Heap. φ enthalte keine freien Variablen und weder „ \wedge “ noch „ $=$ “. Es gilt

$$h, i_0, \eta_0 \models \varphi' \iff \alpha(h, i_0) \in \mathcal{L}(g[\![\varphi']\!]).$$

Beweis. Siehe Theorem 7.6 in [Dod08], Seite 155, f.

Das dort verwendete $\alpha(h)$ entspricht dabei dem hier verwendeten $\alpha(h, i_0)$, da Dodds die Synthese von freien Variablen nicht unterstützt und i daher für α irrelevant ist. □

⁵Die dort verwendeten Aussagen $\alpha(h, i) \in \mathcal{L}(g[\![\varphi]\!])$ und $\alpha(h, i) \in \mathcal{L}(\lfloor g[\![\varphi]\!] \rfloor)$ sind hier offenbar äquivalent, da $\alpha(h, i) \in \text{HC}_\sigma$ stets gilt. $\lfloor \cdot \rfloor$ ist der von Dodds verwendete Normierungsoperator, der aus einer beliebigen HRG eine DSG generiert.

Kapitel 4. Übersetzung zwischen HRGen und SLF

Das in dieser Arbeit verwendete Separation-Logic-Fragment erlaubt zusätzlich zu dem von Dodds außerdem freie Variablen, die als Rang-1-Kanten übersetzt werden, sowie Gleichheitsaussagen über beliebige Variablen. Beides ist nur im **in**-Teil der Formel zulässig, also insbesondere nicht in der Definition eines Prädikats. Das folgende Lemma gewährleistet die Korrektheit zunächst für freie Variablen.

LEMMA 4.13 Sei $\varphi = \mathbf{let} \Gamma \mathbf{in} \psi \in \text{SLF}$ flach und genüge **C1**, **C2** & **C3**. Seien außerdem $h \in \text{He}$, $i \in \text{Int}$ und ψ enthalte weder „ \wedge “ noch „ $=$ “. Es gilt

$$h, i, \eta_0 \models \varphi \iff \alpha(h, i) \in \mathcal{L}(g[\varphi]).$$

Beweis. Seien $\{y_1, \dots, y_n\}$ die freien Variablen von ψ . Wir betrachten zunächst die Formel

$$\varphi' = \mathbf{let} \Gamma \mathbf{in} \exists y_1 \dots \exists y_n. \psi$$

Offenbar enthält φ' keine freien Variablen (wegen **C3** auch nicht in Γ). Nach Satz 4.12

$$h, i_0, \eta_0 \models \varphi' \iff \alpha(h, i_0) \in \mathcal{L}(g[\varphi']).$$

Entfernen wir nun den ersten Existenzquantor und gelangen damit zu

$$\varphi'' = \mathbf{let} \Gamma \mathbf{in} \exists y_2 \dots \exists y_n. \psi,$$

so ändert dies die Startgraphen von $g[\varphi']$ dahingehend, dass die Knoten, die y_1 entsprechen, mit einer Rang-1-Kante e_{y_1} , $\text{lab}(e_{y_1}) = y_1$ versehen sind. Die Ableitungsregeln sind davon nicht betroffen, da sie ausschließlich von Γ abhängen.

φ'' verfügt über die freie Variable y_1 . Da y_1 nun nicht mehr durch seinen Existenzquantor zu i hinzugefügt wird, muss für eine Interpretation i' mit $h, i', \eta_0 \models \varphi''$ gelten, dass $\text{dom}(i') = \text{dom}(i) \cup \{y_1\}$. Offensichtlich existiert ein solches i' genau dann, wenn zuvor $h, i, \eta_0 \models \varphi'$ galt.

Gelte nun $h, i', \eta_0 \models \varphi''$. Dann enthält $\alpha(h, i')$ eine weitere Variablenkante mit dem Label y_1 , entspricht aber ansonsten $\alpha(h, i)$. Die Anwendung von \mathcal{L} auf $g[\varphi'']$ fügt Variablenkanten weder hinzu noch entfernt sie diese. Es folgt

$$\alpha(h, i') \in \mathcal{L}(g[\varphi'']).$$

Die Rückrichtung gilt analog dazu.

Wir starten mit $i = i_0$ und wiederholen dieses Vorgehen induktiv für alle Quantoren, wobei $\text{dom}(i^{(j)})$ weiter wächst. Schließlich erhalten wir die Aussage

$$h, i^{(n)}, \eta_0 \models \varphi \iff \alpha(h, i^{(n)}) \in \mathcal{L}(g[\varphi])$$

mit $\text{dom}(i^{(n)}) = \text{FV}(\varphi) = \text{FV}(\psi) = \{y_1, \dots, y_n\}$, womit die Behauptung bewiesen ist. \square

Abschließend erlauben wir noch Gleichheitsaussagen von Variablen, womit unser SL-Fragment vollständig abgedeckt ist.

SATZ 4.14 Sei $\varphi = \mathbf{let} \Gamma \mathbf{in} \psi \in \text{SLF}$ flach und genüge **C1**, **C2** & **C3**. Seien ferner $h \in \text{He}$ ein Heap und $i \in \text{Int}$ eine Variableninterpretation. Es gilt

$$h, i, \eta_0 \models \varphi \iff \alpha(h, i) \in \mathcal{L}(g[\![\varphi]\!])$$

Beweis. Annahme: $h, i, \eta_0 \models \mathbf{let} \Gamma \mathbf{in} \psi$ gilt.

Dies ist genau dann der Fall, wenn $\psi' \in \text{dnf}(\psi)$ existiert, so dass $h, i, \eta_0 \models \mathbf{let} \Gamma \mathbf{in} \psi'$. O. B. d. A. formen wir ψ' zu ψ'' um, wobei

1. Alle existenzquantifizierten Variablen in frische Variablen umbenannt werden.
2. Alle Quantoren an den Anfang gezogen werden
3. Alle Gleichheitsaussagen ebenfalls nach vorne (hinter die Quantoren) gezogen werden (Wegen der Kommutativität von \wedge und Assoziativität von $*$ und \wedge für pure SL-Formeln zulässig [Rey08]).

$$\psi'' = \exists x_1 \dots \exists x_n. (y_1 = el_1) \wedge \dots \wedge (y_m = el_m) \wedge \psi'''$$

wobei gilt, dass $\psi'' \equiv \psi'$ und dementsprechend $\mathbf{let} \Gamma \mathbf{in} \psi' \equiv \mathbf{let} \Gamma \mathbf{in} \psi''$ und ψ''' frei von $\exists, =, \wedge$ und \vee ist.

Daraus folgt, dass $h, i, \eta_0 \models \mathbf{let} \Gamma \mathbf{in} \psi''$ genau dann gilt, wenn $l_1, \dots, l_n \in \text{Loc}$ existieren und $i' := i[x_1 \mapsto l_1, \dots, x_n \mapsto l_n]$ so dass $h, i', \eta_0 \models \mathbf{let} \Gamma \mathbf{in} \psi'''$ und für alle $j, 1 \leq j \leq m$ gilt $\llbracket y_j \rrbracket i' = \llbracket el_j \rrbracket i'$.

HINWEIS: $\{x_1, \dots, x_n\}$ und $\{y_1, \dots, y_m\}$ müssen nicht disjunkt sein.

Nach Lemma 4.13 gilt

$$h, i', \eta_0 \models \mathbf{let} \Gamma \mathbf{in} \psi''' \iff \alpha(h, i') \in \mathcal{L}(g[\![\mathbf{let} \Gamma \mathbf{in} \psi''']\!]).$$

Per Konstruktion von $g[\![\cdot]\!]$ und $h[\![\cdot]\!]$ erhalten als gleich angenommene Knoten mindestens einen gemeinsamen Eintrag im Tagging und werden deshalb durch \Downarrow verschmolzen.

$\alpha(h, i')$ enthält im Vergleich zu $\alpha(h, i)$ ausschließlich weitere Rang-1-Kanten (für x_1, \dots, x_n). Aufgrund der Beschaffenheit von i' liegen die Kanten für als gleich angenommene Variablen am selben Knoten an. Es folgt

$$\alpha(h, i') \in \mathcal{L}(g[\![\mathbf{let} \Gamma \mathbf{in} \psi''']\!]) \iff \alpha(h, i) \in \mathcal{L}(g[\![\mathbf{let} \Gamma \mathbf{in} \psi'']\!]) = \mathcal{L}(g[\![\mathbf{let} \Gamma \mathbf{in} \psi']\!])$$

und somit

$$h, i, \eta_0 \models \mathbf{let} \Gamma \mathbf{in} \psi \iff \alpha(h, i) \in \mathcal{L}(g[\![\mathbf{let} \Gamma \mathbf{in} \psi]\!]).$$

□

4.4. GENERIERUNG VON SLF-FORMELN AUS HRGS

Nachdem Abschnitt 4.2 die Grammatiksynthese auf Basis der Spezifikation einer SLF-Formel eingeführt und ausführlich beleuchtet hat, widmet sich dieses Unterkapitel der Gegenrichtung, der Generierung von SLF-Formeln aus DSGs. Diese funktioniert im Großen und Ganzen analog zur Grammatiksynthese, birgt aber einige Besonderheiten, beispielsweise bei der Erzeugung von Nullzeigern. Wie schon die Elemente der Grammatiksynthese sind auch die hier verwendeten Funktionen zur Formelgenerierung eng an den von Dodds & Plump vorgestellten Ansatz [DP08b] angelehnt.

Aufgrund der fehlenden Injektivität von α ist die Formalisierung der gewünschten Semantikerhaltung bei der Formelgenerierung etwas unhandlicher, als bei der Grammatiksynthese. Sei φ die aus $G \in \text{DSG}_{\Sigma_N}$ generierte Formel. Dann soll gelten:

$$H \in \mathcal{L}(G) \implies \forall (h, i) \in \alpha^{-1}(H) : h, i, \eta_0 \models \varphi$$

$$H \in \mathcal{L}(G) \iff \exists (h, i) \in \alpha^{-1}(H) : h, i, \eta_0 \models \varphi$$

wobei η_0 wieder die leere Prädikatinterpretation repräsentiert.

Die Formelgenerierung ist ausschließlich auf Datenstrukturgrammatiken definiert. Darüber hinaus dürfen deren rechte Regelseiten keine Variablenkanten enthalten. Diese sind jedoch, ebenso wie die Verwendung von Nicht-DSGs im Kontext von SLF-Formeln, ohnehin wenig sinnvoll. Die Grammatik müsste in jedem Fall so geartet sein, dass jede Regel, die eine Variablenkante einführt, höchstens einmal ausgeführt wird, da ansonsten die HC-Eigenschaften verletzt sind. In diesem Fall kann die Variable jedoch auch direkt in die Startgraphen integriert werden. Zur Abstraktion und Konkretisierung haben Variablen auf rechten Regelseiten ebenfalls keinen Wert. Nicht-DSGs beschreiben Graphen, denen kein Heap zuzuordnen und auf denen α^{-1} daher nicht definiert ist. Sie sind daher für die meisten Anwendungen im Kontext von SLF nicht bedeutsam, wenngleich ihre Übersetzung aufgrund der recht größeren Mächtigkeit der Separation Logic prinzipiell möglich wäre. Die Korrektheit müsste dann allerdings neu definiert werden.

4.4.1. STANDARDTAGGING

In Abschnitt 4.2 wurde das Tagging als Verbindungsglied zwischen Variablen und Knoten eingeführt. Aus jeder Variable wurde ein getaggtter Knoten erzeugt, dessen Tags bei einer Quantisierung wieder entfernt wurden. Bei der Generierung von Formeln müssen hingegen Variablennamen für anonyme, also unbenannte Knoten gefunden werden. Dazu dient das Standardtagging.

DEFINITION 4.17 (Standardtagging) Sei $H \in \text{HC}_{\Sigma_N}$ eine Heapkonfiguration. Das *Standardtagging* für H ist ein eindeutig definiertes Tagging $t : V_H \rightarrow 2^{\text{Var}}$, das die folgende Bedingung erfüllt

$$t(v) = \begin{cases} \{x_j\} & , \text{ if } v = \text{ext}_H(j) \\ T \subseteq \text{var}_{\Sigma} & , \text{ if } T \neq \emptyset \wedge \forall x \in T \exists e \in E_H : \text{lab}_H(e) = x \wedge \text{att}_H(e) = v \\ \{r\} & , \text{ otherwise} \end{cases}$$

wobei $r \in \text{Var} \setminus (\{x_j \mid j \in \mathbb{N}\} \cup \{\mathbf{nil}\} \cup \text{var}_{\Sigma})$ für alle Knoten verschieden ist.

Externen Knoten wird dabei je nach Index j in der ext-Sequenz stets eine Parameter-Variable x_j zugewiesen. Verfügt ein Knoten über eine oder mehrere Variablenkanten, so wird ihm die Menge T aller zugehörigen Variablen als Tag zugewiesen. Diese dienen später zur Erzeugung von Gleichheitsassertionen. Die beiden ersten Fälle können für einen Knoten nie gleichzeitig auftreten, da Variablenkanten auf rechten Regelseiten und externe Knoten für Startgraphen verboten sind. Trifft keine der beiden Bedingungen zu, so erhält der Knoten einen Tag bestehend aus einer „frischen“ Variable r , die anderweitig noch nicht verwendet wird.

4.4.2. ERZEUGUNG VON NULLZEIGERASSERTIONEN

In Abschnitt 4.1.2 wurde die Problematik der impliziten **nil**-Darstellung in Hypergraphen im Zusammenhang mit expliziten **nil**-Assertionen erläutert.

Um aus einem Graphen eine äquivalente Formel zu generieren, müssen daher alle nicht vorhandenen Selektoren, also solche, die implizit auf **nil** zeigen, identifiziert und in entsprechende Nullzeigerassertionen übersetzt werden. Im Falle konkreter Graphen ist dies problemlos möglich. Ist der betrachtete Knoten allerdings mit einem Nicht-Reduktionstentakel einer Nichtterminalkante verbunden, so kann diese Kante weitere Selektoren erzeugen. Das Einsetzen einer Nullzeigerassertion für diesen Selektor würde die Semantik der Formel dann verfälschen oder diese sogar unerfüllbar machen.

Das Verfahren zur Formelgenerierung benötigt daher Informationen darüber, welche ausgehenden Selektoren eine Nichtterminalkante erzeugen wird. Da eine Kante aber nicht bei jeder Ableitung grundsätzlich dieselben Selektoren erzeugt, ist es notwendig, die Klasse der Datenstrukturgrammatiken für die Formelgenerierung auf getypte DSGs einzugrenzen (Siehe Abschnitt 2.3.3). Für diese ist die Menge der erzeugten, ausgehenden Selektorkanten für jeden Tentakel über alle möglichen Ableitungen konstant. So kann „außerhalb“ der Nichtterminalkante die Menge der zu erzeugenden Nullzeigerassertionen ermittelt werden. Dies wird formalisiert über den Begriff des Nullzeigers.

DEFINITION 4.18 (Nullzeiger) Sei $G \in \text{DSG}_{\Sigma_N}$ eine getypte Grammatik, $H \in \text{HG}_{\Sigma_N}$ ein Hypergraph, $v \in V_H$ ein Knoten und $s \in \text{sel}_{\Sigma}$ ein Selektor. Das Paar (v, s) heißt *Nullzeiger*, genau dann wenn alle der drei folgenden Bedingungen erfüllt sind.

- $\nexists e \in E_H$, so dass $\text{att}_H(e)(1) = v$ und $\text{lab}_H(e) = s$
(eine ausgehende s -Kante existiert nicht)
- $\nexists e \in E_H$, $X \in N$, $j \in \mathbb{N}$, $1 \leq j \leq \text{rk}(X)$,
so dass $\text{lab}_H(e) = X$, $\text{att}_H(e)(j) = v$ und $s \in \text{type}(X, j)$
(kein Tentakel erzeugt eine ausgehende s -Kante)
- $v \notin [\text{ext}_H]$
(v ist nicht extern)

NP_H bezeichnet die Menge aller Nullzeiger in H .

Der letzte Punkt ist von Bedeutung, weil über die Kanten, die an externen Knoten erzeugt werden, keine Aussagen gemacht werden können. Wird H zum Beispiel durch Hyperkantenersetzung in einen anderen Graphen H' eingebettet, dann ist nicht bekannt, welche Kanten dort an den externen Knoten von H anliegen. Dies ist jedoch unproblematisch, da die Nullzeiger

Kapitel 4. Übersetzung zwischen HRGen und SLF

in diesem Fall im Graphen H' identifiziert werden. Heapkonfigurationen enthalten keine externen Knoten, so dass alle Nullzeiger spätestens im Startgraphen erkannt werden.

LEMMA 4.15 (Entscheidbarkeit der Nullzeiger) Betrachtet wird eine getypte Grammatik. Sei $H \in \text{HG}_{\Sigma_N}$. Es ist entscheidbar, ob $(v, s) \in V_H \times \text{sel}_{\Sigma}$ ein Nullzeiger ist.

Beweis. (Skizze) Die Funktion type , die jedem Tentakel einer Nichtterminalkante die Menge der von ihm erzeugten ausgehenden Selektorkanten zuordnet, kann durch Analyse einer Produktionsregel für jedes Nichtterminal per Fixpunktiteration berechnet werden. Aufgrund der Getyptheit ist sie für alle Produktionsregeln zum selben Nichtterminal gleich.

Die Nullzeiger-Kriterien können dann in H geprüft werden. \square

Jeder Nullzeiger kann direkt in die dazugehörige Nullzeigerassertion übersetzt werden (siehe Definition 4.20).

4.4.3. ÜBERSETZUNG VON KANTEN

Wie auch die Grammatiksynthese ist die Formelgenerierung in verschiedene Teilfunktionen gegliedert, die jeweils ein Grammatik-Konzept in eine adäquate Formel übersetzen. Die Elementarformeln, die auch die Basis der induktiven Definition von $h[\cdot]$ bilden, werden durch Übersetzung von Kanten erzeugt. Dies ist sinnvoll, da eine Kante einen Bezug zwischen mehreren Knoten modelliert, ebenso wie eine Assertion es für die darin verwendeten Variablen tut. Eine Ausnahme bilden hier die Variablenkanten, die für Elementarformeln keine Rolle spielen und daher zu **emp**, dem neutralen Element der separierenden Konjunktion ($*$) übersetzt werden. Um Verwechslungen mit anderen Formelzeichen auszuschließen werden die Formelgenerierungsfunktionen ebenso wie Grammatiksynthesefunktionen stets mit Doppelklammern $\llbracket \cdot \rrbracket$ notiert.

DEFINITION 4.19 (Übersetzung von Kanten $s_e[\cdot]$) Die Funktion $s_e[\cdot]$ übersetzt beliebige Kanten e mit Hilfe eines Taggings t in **let**-freie Formeln.

$$s_e[\llbracket e, t \rrbracket] = \begin{cases} t(v)(1).s \mapsto t(\bar{v})(1) & , \text{ if } \text{lab}(e) \in \text{sel}_{\Sigma} \\ \text{mit } \text{att}(e) = v\bar{v}, s = \text{lab}(e) \\ \sigma_X(t(v_1)(1), \dots, t(v_n)(1)) & , \text{ if } \text{lab}(e) \in N \\ \text{mit } X = \text{lab}(e), \text{att}(e) = v_1 \dots v_n \\ \mathbf{emp} & , \text{ if } \text{lab}(e) \in \text{var}_{\Sigma} \end{cases}$$

Sowohl in dieser als auch in nachfolgenden Definitionen verwenden wir abkürzend die Schreibweise $t(v)(j)$ für $\langle t(v) \rangle(i)$, also das j -te Element der kanonischen Ordnung des Tags von v .

Die Formelgenerierung aus Kanten entspricht also genau der Umkehrung der ersten drei Fälle aus Definition 4.6.

Beispiel 4.13. Eine Selektorkante mit dem Label s , die aus dem mit $\{x, y\}$ getaggten Knoten ausgeht und deren Zielknoten mit $\{z\}$ getaggt ist, wird übersetzt in die Formel

$$x.s \mapsto z.$$

4.4. Generierung von SLF-Formeln aus HRGs

Eine Nichtterminalkante mit dem Label X deren inzidente Knoten die Labels $\{y_1, z_1\}, \dots, \{y_n, z_n\}$ haben, ergibt den Prädikataufruf

$$\sigma(y_1, \dots, y_n).$$

△

4.4.4. ÜBERSETZUNG VON GRAPHEN

Die Graphübersetzung wird durch die Funktion $s_h[\cdot]$ realisiert. Sie greift dabei neben $s_e[\cdot]$ auch auf die Funktion $s_{np}[\cdot]$ zurück, die die Nullzeiger des Graphen übersetzt.

DEFINITION 4.20 (Übersetzung von Nullzeigern $s_{np}[\cdot]$) Sei v ein Knoten, $s \in \text{sel}_\Sigma$ ein Selektor und t ein Tagging. Die Nullzeiger-Übersetzungsfunktion ist definiert als

$$s_{np}[\![v, s, t]\!] = t(v)(1).s \mapsto \mathbf{nil}$$

$s_h[\cdot]$ verknüpft die durch $s_e[\cdot]$ und $s_{np}[\cdot]$ erzeugten Assertionen zu komplexen Formeln und führt Gleichheitsassertionen und Quantisierungen ein.

DEFINITION 4.21 (Übersetzung von Graphen $s_h[\cdot]$) Die Übersetzung eines getaggten Graphen $\mathcal{H} = (H, t)$ ist gegeben durch $s_h[\![\cdot]\!]$:

$$s_h[\![H, t]\!] = \exists r_1 \dots \exists r_m : \bigwedge_{v \in V_H} \bigwedge_{\substack{x \in t(v) \\ x \neq t(v)(1)}} (x = t(v)(1)) \wedge \bigodot_{e \in E_H} s_e[\![e, t]\!] * \bigodot_{(v,s) \in \text{NP}_H} s_{np}[\![v, s, t]\!]$$

Mit r_1, \dots, r_m seien alle $\{r\}$ -Tags im Sinne des dritten Falles in Definition 4.17 bezeichnet. \bigodot ist der Iterator für die separierende Konjunktion. Für Iterationen über die leere Menge gilt $\bigodot_{a \in \emptyset} a := \mathbf{emp}$.

Zunächst werden alle Variablen, die durch das Standardtagging neu eingeführt wurden, also für Knoten stehen, welche weder extern sind noch über eine Variablenkante verfügen, existenzquantifiziert. Es folgt eine Sequenz von Gleichheitsassertionen. Dabei wird über alle Knoten und darin geschachtelt über alle Variablen in deren Tags iteriert, beginnend mit der zweiten. Alle gefundenen Variablen werden der ersten des jeweiligen Tags gleichgesetzt. Es folgt die durch separierende Konjunktion verknüpfte Liste der Kanten-Assertionen sowie der Nullzeigerassertionen.

4.4.5. ÜBERSETZUNG VON PRODUKTIONSREGELN

Die Übersetzung von Produktionsregeln erfordert zunächst die Übersetzung von Graphmengen. Diese wird von $s_H[\![\cdot]\!]$ durchgeführt.

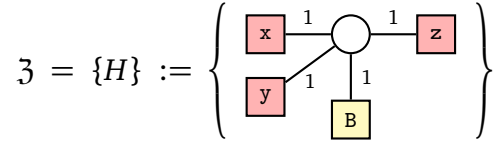
DEFINITION 4.22 (Übersetzung von Graphmengen $s_H[\![\cdot]\!]$) Seien $\mathcal{H}_1, \dots, \mathcal{H}_n \in \text{HG}_{\Sigma_N}$ getaggte Hypergraphen.

$$s_H[\![\mathcal{H}_1, \dots, \mathcal{H}_n]\!] = s_h[\![\mathcal{H}_1]\!] \vee \dots \vee s_h[\![\mathcal{H}_n]\!]$$

Eine Produktionsregel wird – analog zu $r[\![\cdot]\!]$ – in eine Prädikatdefinition $\sigma_X(x_1, \dots, x_n) = \psi$ übersetzt. Die Formel ψ ergibt sich dabei durch Anwendung von $s_H[\![\cdot]\!]$ auf die Menge aller rechten Regelseiten zum Nichtterminal X . Die Disjunktion der Teilformeln repräsentiert dabei die Wahl aus mehreren Produktionsregeln bei der Ableitung von Hypergraphen. Offensichtlich

4.4. Generierung von SLF-Formeln aus HRGs

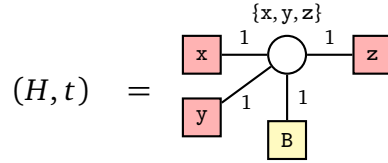
Die Startgraphmenge sei durch \mathfrak{Z} gegeben. Es gelte $\text{cn}(x) < \text{cn}(y) < \text{cn}(z)$.



Wir übersetzen G mittels $s[\cdot]$ in eine SLF-Formel.

$$\varphi = s[G] = \mathbf{let} \ s_{\mathcal{P}}[P] \ \mathbf{in} \ s_H[\mathfrak{Z}']$$

Dabei ist $s_H[\mathfrak{Z}'] = s_h[(H, t)]$, da \mathfrak{Z} nur einen Graphen enthält. Andernfalls würden die mit $s_h[\cdot]$ generierten Formeln miteinander disjunct. t ist das Standardtagging zu H :

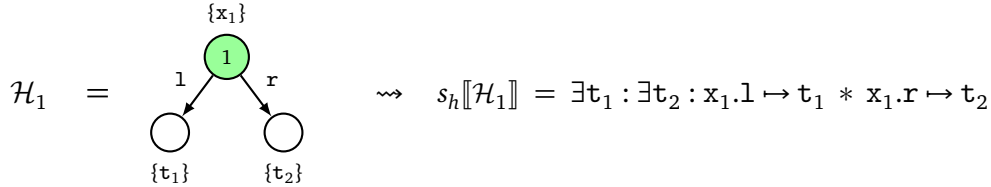


$$\rightsquigarrow s_H[\mathfrak{Z}'] = s_h[(H, t)] = (x = y) \wedge (x = z) \wedge \sigma_B(x)$$

Aus dem Regelsatz P ergibt sich die Definition des einzigen Prädikates σ_B .

$$\begin{aligned} s_{\mathcal{P}}[P] &= (\sigma_B(x_1) = s_H[\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3, \mathcal{H}_4]) \\ &= (\sigma_B(x_1) = s_h[\mathcal{H}_1] \vee s_h[\mathcal{H}_2] \vee s_h[\mathcal{H}_3] \vee s_h[\mathcal{H}_4]) \end{aligned}$$

wobei $\mathcal{H}_1, \dots, \mathcal{H}_4$ die vier rechten Regelseiten zu B , jeweils mit ihrem Standardtagging sind. In \mathcal{H}_1 beispielsweise wird der externe Knoten gemäß Definition 4.17 mit x_1 und die beiden Verbleibenden mit frischen Variablen t_1 und t_2 getaggt. Dementsprechend werden sie durch $s_h[\cdot]$ existenzquantifiziert (analog für $\mathcal{H}_2, \mathcal{H}_3, \mathcal{H}_4$).



Für die gesamte Grammatik ergibt sich schließlich die Formel

$$\begin{aligned} s[G] = \mathbf{let} \ \sigma_B(x_1) = & \exists t_1 : \exists t_2 : x_1.l \mapsto t_1 * x_1.r \mapsto t_2 \\ & \vee \exists t_3 : \exists t_4 : x_1.l \mapsto t_3 * x_1.r \mapsto t_4 * \sigma_B(t_3) \\ & \vee \exists t_5 : \exists t_6 : x_1.l \mapsto t_5 * x_1.r \mapsto t_6 * \sigma_B(t_6) \\ & \vee \exists t_7 : \exists t_8 : x_1.l \mapsto t_7 * x_1.r \mapsto t_8 * \sigma_B(t_7) * \sigma_B(t_8) \\ & \mathbf{in} \ (x = y) \wedge (x = z) \wedge \sigma_B(x) \end{aligned}$$

△

4.4.7. LAUFZEITBETRACHTUNG

Wie schon bei der Grammatiksynthese, macht auch hier die Übersetzung von Graphen den wichtigsten Teil der Laufzeitabschätzung aus. Diese wurde in Definition 4.21 mittels der Funktion $s_h[\cdot]$ definiert:

$$s_h[(H, t)] = \exists r_1 \dots \exists r_m : \bigwedge_{v \in V_H} \bigwedge_{\substack{x \in t(v) \\ x \neq t(v)(1)}} (x = t(v)(1)) \wedge \bigodot_{e \in E_H} s_e[e, t] * \bigodot_{(v,s) \in NP_H} s_{np}[v, s, t]$$

Die Quantifizierung $\exists r_1 \dots \exists r_m$ liegt offensichtlich in $\mathcal{O}(|V|)$. Die anschließende Gleichsetzung von Variablen hat die Komplexität $\mathcal{O}(|\text{var}_\Sigma|)$, da jede Variable höchstens einmal im Graphen vorkommen kann. Wir gehen dabei zunächst von einem vorgegebenen Tagging aus.

Für die Kantenübersetzung $\bigodot_{e \in E_H} s_e[e, t]$ ergibt sich die Laufzeitabschätzung

$$\mathcal{O} \left(|V| \cdot |\text{sel}_\Sigma| + |\text{var}_\Sigma| + |N_G| \cdot |V| \cdot \max_{X \in N_G} \text{rk}(X)^2 \right).$$

Es gibt höchstens $|N_G| \cdot |V| \cdot \max_{X \in N_G} \text{rk}(X)$ viele Nichtterminalkanten. Ihre Übersetzung hat jeweils die Worst-Case-Laufzeit

$$\mathcal{O} \left(\max_{X \in N_G} \text{rk}(X) \right),$$

da alle Argumente, deren Anzahl vom Rang des Nichtterminals abhängt, in die Formel aufgenommen werden müssen. Deshalb wird der Term $\max_{X \in N_G} \text{rk}(X)$ in der Laufzeitbetrachtung quadriert. Die Übersetzung aller Terminalkanten liegt in $\mathcal{O}(1)$. Ihre Anzahl ist durch $\mathcal{O}(|\text{var}_\Sigma| + |V| \cdot |\text{sel}_\Sigma|)$ beschränkt.

Die Ermittlung sämtlicher Nullzeiger kann in $\mathcal{O}(|V| \cdot |N_G|)$ ausgeführt werden. Dabei wird davon ausgegangen, dass an jedem Knoten v jedes Nichtterminal X mit jedem Tentakel j höchstens einmal anliegt. Ist dies nicht der Fall, darf die Kante entweder keinen ausgehenden Selektor s am entsprechenden Tentakel erzeugen, (v, s) wäre dann sofort ein Nullzeiger, oder s wäre aufgrund der Getyptheit stets doppelt vorhanden und die Grammatik dann keine DSG mehr. Aus der zuvor ermittelten Funktion type kann mittels $\text{type}(X, j)$ ermittelt werden, ob s darin enthalten ist, oder nicht. Dies geschieht in $\mathcal{O}(\log |\text{sel}_\Sigma|)$. Das Einsetzen der Nullzeigerassertionen in die Formel hat als schlechtestes Zeitverhalten $\mathcal{O}(|V| \cdot |\text{sel}_\Sigma|)$, für den Fall, dass alle Selektoren aller Knoten Nullzeiger sind. Insgesamt ergibt sich für $s_h[\cdot]$ eine Laufzeitkomplexität von

$$\mathcal{O} \left(|\text{var}_\Sigma| + |V| \cdot |\text{sel}_\Sigma| \cdot |N_G| + |V| \cdot |N_G| \cdot \max_{X \in N_G} \text{rk}(X)^2 \right)$$

Gehen wir realistischerweise von einer konstanten Anzahl Selektoren und einem vernachlässigbaren höchsten Nichtterminal-Rang aus, so ergibt sich

$$\mathcal{O}(|\text{var}_\Sigma| + |V| \cdot |N_G|).$$

Die Berechnung der type -Funktion kann zuvor global für alle Graphen ausgeführt werden und hat die Komplexität

$$\mathcal{O} \left(|N_G| \cdot |\text{sel}_\Sigma| \cdot \max_{X \in N_G} \text{rk}(X) \right)$$

beziehungsweise $\mathcal{O}(|N_G|)$ unter Vernachlässigung von $|\text{sel}_\Sigma|$ und dem Nichtterminal-Rang. Für die Produktionsregelsynthese und schließlich die Grammatiksynthese addiert sich noch die Berechnung des Standardtaggings hinzu. Dieses hat pro Graph eine Komplexität von $\mathcal{O}(|V| + |\text{var}_\Sigma|)$ da $|V|$ Knoten getaggt werden müssen, wobei insgesamt höchstens $|\text{var}_\Sigma|$ viele freie Variablen anfallen. Die x_j und r -Tags hingegen sind stets einelementig. Für die Übersetzung einer Grammatik mit m rechten Regelseiten und k Startgraphen ergibt sich unter Annahme einer konstant-großen Selektorenmenge und unter beschränktem Nichtterminal-Rang also

$$\mathcal{O}\left((m+k) \cdot \max_{H \in \mathfrak{S}} |V_H| \cdot |N_G| + |\text{var}_\Sigma|\right)$$

wobei \mathfrak{S} die Menge aller rechten Regelseiten und Startgraphen darstellt.

4.5. KORREKTHEIT DER ÜBERSETZUNGSFUNKTION

In diesem Abschnitt wird gezeigt, dass die Übersetzungsfunktion $s[\![\cdot]\!]$ semantikerhaltend ist. Die zu beweisende Kernaussage (Satz 4.23) ist dabei analog zur Semantikerhaltung von $g[\![\cdot]\!]$ in Abschnitt 4.3 ($G \in \text{DSG}_{\Sigma_N}$, $H \in \text{HC}_{\Sigma_N}$, $h \in \text{He}$, $i \in \text{Int}$):

$$H \in \mathcal{L}(G) \implies \forall (h, i) \in \alpha^{-1}(H) : h, i, \eta_0 \models s[\![G]\!]$$

$$H \in \mathcal{L}(G) \iff \exists (h, i) \in \alpha^{-1}(H) : h, i, \eta_0 \models s[\![G]\!]$$

Die Quantisierung von (h, i) ist notwendig, da die Funktion α nicht injektiv ist. Es gibt zwar aufgrund der direkten Darstellung von Knoten durch Heapadressen für jeden aus α berechneten Graphen nur einen möglichen Heap, der als Argument von α zu diesem Graphen führt, für die Variableninterpretation trifft dies jedoch nicht zu. Tatsächlich gibt es zu jedem Tupel $(h, i) \in \text{He} \times \text{Int}$ unendlich viele $i' \in \text{Int}$, so dass $\alpha(h, i) = \alpha(h, i')$ gilt. Dies liegt daran, dass i nicht auf diejenigen Variablen beschränkt ist, deren Wert einem Knoten innerhalb des Graphen entspricht. Die Semantik von SLF (Definition 3.13) stellt an die Größe des Definitionsbereiches von i grundsätzlich keine Anforderungen. Die beiden oben aufgeführten rechten Seiten sind dennoch äquivalent, wie das folgende Lemma zeigt:

LEMMA 4.17 Sei $G \in \text{DSG}_{\Sigma_N}$ getypt, $H \in \text{HC}_{\Sigma_N}$, $(h_1, i_1), (h_2, i_2) \in \alpha^{-1}(H)$ und η eine Prädikatdefinition. Es gilt

$$h_1, i_1, \eta \models s[\![G]\!] \iff h_2, i_2, \eta \models s[\![G]\!].$$

Beweis. Annahme: $h_1, i_1, \eta \models s[\![G]\!]$. Nach Definition von α^{-1} gilt offensichtlich $\alpha(h_1, i_1) = H = \alpha(h_2, i_2)$. Gemäß der Definition von α folgt daraus wiederum $h_1 = h_2$. Ferner gilt aufgrund der Definitionen von E , att und lab bei α

$$\text{dom}(i_1) \cap \text{lab}(H) = \text{dom}(i_2) \cap \text{lab}(H)$$

und außerdem

$$\forall x \in \text{lab}(H) \cap \text{var}_\Sigma : i_1(x) = i_2(x).$$

Kapitel 4. Übersetzung zwischen HRGen und SLF

Aufgrund der Konstruktion von $s[\cdot]$ sind alle in $s[G]$ auftretenden Variablen y mit $y \notin \text{lab}(H) \cap \text{var}_\Sigma$ existenzquantifiziert. Es folgt

$$h_2, i_2, \eta \models s[H]$$

und daraus (analog in Gegenrichtung) die Behauptung. \square

Anders ausgedrückt unterscheiden sich i und i' aus $(h, i), (h, i') \in \alpha^{-1}(H)$ also stets nur in den Variablen, für die weder i noch i' auf den Heap h verweist. In Bezug auf h und H sind beide folglich äquivalent. Wir definieren deshalb eine Funktion β und analog dazu β_t , die jeweils die kleinste gültige Variableninterpretation zurückgibt.

DEFINITION 4.25 (β, β_t) β ist definiert als Umkehrfunktion zu α mit einelementigen Funktionswerten. Es gelte $\beta(H) = (h, i)$, wenn

- $(h, i) \in \alpha^{-1}(H)$ und
- $\text{dom}(i) = \text{lab}(E_H) \cap \text{var}_\Sigma$ („kleinstes“ i)

Analog dazu definieren wir β_t als inverse Funktion zu α_t . Es gelte $\beta_t(H, t) = (h, i)$, wenn

- $(h, i) \in \alpha_t^{-1}(H, t)$ und
- $\text{dom}(i) = \text{img}(t)$ („kleinstes“ i)

Unter Verwendung von β und β_t als Umkehrfunktionen von α und α_t können wir letztere als pseudo-bijektiv auffassen. Dodds setzt die exklusive Betrachtung der jeweils kleinsten Variableninterpretation in α^{-1} implizit voraus und bezeichnet α und α_t direkt als bijektiv. Der folgende Beweis der semantischen Korrektheit von $s[\cdot]$ orientiert sich stark an [Dod08] und verwendet teilweise dessen Elemente und Lemmata. Wir zeigen die Semantikerhaltung zunächst für die Übersetzung einzelner getaggtter Hypergraphen und übernehmen dann die Resultate von Dodds für Graphmengen und schließlich ganze Grammatiken, deren rechte Regelseiten und Startgraphen ungetypt und letztere gegebenenfalls mit Variablenkanten versehen sind. Da die Getyptheit der DSGen Voraussetzung für $s[\cdot]$ ist, betrachten wir im Folgenden ausschließlich getypte Grammatiken.

Der fundamentale Unterschied der hier vorgestellten Graph-Übersetzung zu der von Dodds verwendeten liegt einerseits in der Modellierung einzelner Selektoren als binäre Hyperkanten und andererseits in der impliziten Modellierung von Nullzeigern. Letztere müssen natürlich ebenfalls auf korrekte Zeigerassertionen abgebildet werden. Das folgende Lemma zeigt die Korrektheit von $s_h[\cdot]$. Dazu wird ein getaggtter Graph \mathcal{H}' in eine Formel übersetzt und gezeigt, dass jeder konkrete Graph \mathcal{H} , der in dessen Graphauswertung $\kappa(\lambda, \mathcal{H}')$ liegt, zu einem (h, i) -Paar gehört, das diese Formel erfüllt. Die Graphfunktionen α/α_t und deren Inverse β/β_t kennen jedoch keine externen Knoten. Diese haben allerdings nur insofern Auswirkungen auf die Erfüllbarkeit der generierten Formel, als dass externen Knoten keine Nullpointer erlaubt sind. Deshalb werden auf der linken Seite alle Knoten, die in \mathcal{H}' , also dem Graphen, aus dem die Formel generiert wird, extern sind, in \mathcal{H} ebenfalls exponiert. Im aus \mathcal{H} erzeugten Heap h werden analog dazu die **nil**-Felder von Objekten, deren Knoten in \mathcal{H}' extern sind, gelöscht,

4.5. Korrektheit der Übersetzungsfunktion

was zu \bar{h} führt. Einfach ausgedrückt werden also links Knoten extern gemacht und parallel rechts die entsprechenden **nil**-Felder gelöscht, um zu \mathcal{H}' kompatibel zu werden.

Die für expose benötigten Variablennamen der zu exponierenden Knoten werden vom Tagging t' des Graphen \mathcal{H}' geliefert.

LEMMA 4.18 (Übersetzung von Graphen) Seien $\mathcal{H}, \mathcal{H}' \in \text{HG}_{\Sigma, N}^T$ getaggte Graphen mit je n externen Knoten und keinen Variablenkanten, $n \geq 0$. Sei λ eine Graphumgebung und η eine Prädikatumgebung, die mit λ korrespondiert. Dann gilt

$$\text{expose}(\mathcal{H}, t'(\text{ext}_{\mathcal{H}'})) \in \kappa(\lambda, \mathcal{H}') \iff \bar{h}, i, \eta \models s_h \llbracket \mathcal{H}' \rrbracket$$

wobei $(h, i) = \beta_t(\mathcal{H})$ und \bar{h} definiert ist wie folgt

$$\text{dom}(\bar{h}) = \underbrace{\{l \in \text{dom}(h) \mid h(l) \neq \text{nil} \vee \nexists v \in [\text{ext}_{\mathcal{H}'}], s \in \text{sel}_{\Sigma} : \llbracket t(v) \rrbracket i + \text{cn}(s) = l\}}_{(l \text{ ist nicht Adresse eines externen Nullzeigers)}}$$

$$\bar{h}(l) := h(l)$$

$t'(\text{ext}_{\mathcal{H}'})$ sei die Sequenz $t_{\mathcal{H}'}(\text{ext}_{\mathcal{H}'}(1))(1) \dots t_{\mathcal{H}'}(\text{ext}_{\mathcal{H}'}(n))(1)$, also die Folge der jeweils ersten Variablen des Tags der externen Knoten von \mathcal{H}' .

Beweis. Per Induktion über die Anzahl der Kanten in \mathcal{H}' . Nehmen wir zunächst an, alle Knoten in \mathcal{H}' seien extern. Demnach erzeugt $s_h \llbracket \cdot \rrbracket$ keine Quantoren und Nullzeiger.

INDUKTIONSVERANKERUNG. \mathcal{H}' enthalte genau eine Kante: $E_{\mathcal{H}'} = \{e\}$

Fall 1: $\text{lab}(e) \in \text{sel}_{\Sigma}$

Die Kante ist terminal. $\kappa(\lambda, \cdot)$ verändert den Graphen also nicht:

$$\mathcal{H} \in \kappa(\lambda, \mathcal{H}') \iff \mathcal{H} = \mathcal{H}'.$$

$s_h \llbracket \mathcal{H}' \rrbracket$ ist von der Form $x_1.s \mapsto x_2$ mit einem Selektor s . Nach Definition von α_t bzw. β_t gilt für \mathcal{H} mit $(h, i) = \beta_t(\mathcal{H})$ demnach

$$\bar{h}, i, \eta \models s_h \llbracket \mathcal{H}' \rrbracket \iff \mathcal{H} = \mathcal{H}'$$

Da alle Knoten extern sind, gilt $\mathcal{H} = \text{expose}(\mathcal{H}, \vec{x})$ für jede beliebige Variablenfolge $\vec{x} \in \text{Var}^*$, woraus die Behauptung folgt.

Fall 2: $\text{lab}(e) \in N$

$s_h \llbracket \mathcal{H}' \rrbracket$ hat die Form $\sigma(x_1, \dots, x_n)$.

Da η und λ korrespondieren, ersetzt $\kappa(\lambda, \cdot)$ e durch einen Teilgraphen, der seinerseits mit einem Heap in $\eta(\sigma)$ korrespondiert. Es gilt also

$$\bar{h}, i, \eta \models s_h \llbracket \mathcal{H}' \rrbracket \iff \mathcal{H} \in \kappa(\lambda, \mathcal{H}').$$

und wegen $\mathcal{H} = \text{expose}(\mathcal{H}, \vec{x})$ für alle Folgen $\vec{x} \in \text{Var}^*$ die Behauptung.

Kapitel 4. Übersetzung zwischen HRGen und SLF

Hinweis: Aufgrund der Definition von \mathcal{H} sind auch hier alle Knoten extern.

INDUKTIONSSCHRITT. \mathcal{H}' habe nun n Kanten. Wir können \mathcal{H}' in \mathcal{H}'_1 und \mathcal{H}'_2 zerlegen, so dass $\mathcal{H}' = \mathcal{H}'_1 \bowtie \mathcal{H}'_2$ gilt. Da \mathcal{H}'_1 und \mathcal{H}'_2 echt weniger Kanten als \mathcal{H}' beinhalten, gilt nach der Induktionshypothese, dass es auch \mathcal{H}_1 und \mathcal{H}_2 gibt, so dass $\mathcal{H} = \mathcal{H}_1 \bowtie \mathcal{H}_2$ gilt und außerdem

$$\begin{aligned} \mathcal{H}_1 \in \kappa(\lambda, \mathcal{H}'_1) &\iff \bar{h}_1, i, \eta \models s_h \llbracket \mathcal{H}'_1 \rrbracket \\ \mathcal{H}_2 \in \kappa(\lambda, \mathcal{H}'_2) &\iff \bar{h}_1, i, \eta \models s_h \llbracket \mathcal{H}'_2 \rrbracket \end{aligned}$$

Nach [Dod08], Lemma 7.1 gilt

$$\kappa(\lambda, \mathcal{H}'_1 \bowtie \mathcal{H}'_2) = \kappa(\lambda, \mathcal{H}'_1) \bowtie \kappa(\lambda, \mathcal{H}'_2)$$

Da keine Kante sowohl in \mathcal{H}'_1 und \mathcal{H}'_2 erscheinen darf, gilt für die Adressen der in den Graphen vorhandenen Selektoren, dass diese paarweise verschieden und die erfüllenden Heaps damit disjunkt sind: $\text{dom}(\bar{h}_1) \cap \text{dom}(\bar{h}_2) = \emptyset$.

Aufgrund der Kommutativität und Assoziativität von $*$, sowie der Tatsache, dass $s_h \llbracket \mathcal{H}' \rrbracket$ quantoren- und \mathcal{H}' nullzeigerfrei ist, gilt

$$s_h \llbracket \mathcal{H}' \rrbracket \equiv s_h \llbracket \mathcal{H}'_1 \rrbracket * s_h \llbracket \mathcal{H}'_2 \rrbracket$$

Entsprechend der Semantik von $*$ gilt wegen $\bar{h} = \bar{h}_1 \boxplus \bar{h}_2$:

$$\bar{h}, i, \eta \models s_h \llbracket \mathcal{H}' \rrbracket$$

i ist dabei definiert durch $\text{graph}(i) = \text{graph}(i_1) \cup \text{graph}(i_2)$, da hier keine Anforderungen an die Disjunktheit gestellt sind.

Um dieses Ergebnis auch auf interne Knoten auszuweiten, entfernen wir nun induktiv eine beliebige Anzahl von Knoten aus der ext-Sequenz von \mathcal{H}' . Die INDUKTIONSVERANKERUNG für $[\text{ext}_{\mathcal{H}}] = V_{\mathcal{H}}$, also ausschließlich externe Knoten, ist bereits gezeigt.

INDUKTIONSSCHRITT. Sei v der nächste Knoten, der internalisiert wird.

$$s_h \llbracket \mathcal{H}'_{\text{neu}} \rrbracket \equiv \exists [t'(v)(1)] . s_h \llbracket \mathcal{H}'_{\text{alt}} \rrbracket * \bigcirc_{\substack{s \in \text{sel}_{\Sigma} \text{ mit} \\ (v,s) \in \text{NP}_{\mathcal{H}'_{\text{neu}}}}} s_{\text{np}} \llbracket v, s, t' \rrbracket$$

Die resultierende Formel wird dabei nach der Definition von $s_h \llbracket \cdot \rrbracket$ um die existentielle Quantifizierung des Tags von v , sowie um die Nullzeigerassertionen der an v anliegenden Nullzeiger erweitert.

Da α_t keine externen Knoten erzeugt und β_t entsprechend nur auf Graphen mit ausschließlich internen Knoten definiert ist, gilt $\text{ext}_{\mathcal{H}} = \varepsilon$. Folglich ist v in $\text{expose}(\mathcal{H}_{\text{neu}}, t'(\text{ext}_{\mathcal{H}'_{\text{neu}}}))$ nun ebenfalls intern. $\text{expose}(\mathcal{H}_{\text{neu}}, t'(\text{ext}_{\mathcal{H}'_{\text{neu}}})) \in \kappa(\lambda, \mathcal{H}'_{\text{neu}})$ ist also weiterhin gegeben.

Für den neuen Heap gilt nun offenbar, dass $\tilde{h}_{\text{neu}}, i, \eta \models s_h \llbracket \mathcal{H}'_{\text{neu}} \rrbracket$ genau dann erfüllt ist, wenn $\tilde{h}_{\text{neu}} = h_{\text{alt}} \left[\llbracket t(v) \rrbracket i + \text{cn}(s) \mapsto \mathbf{nil} \mid \forall s. (v, s) \in \text{NP}_{\mathcal{H}'_{\text{neu}}} \right]$ gilt.

4.5. Korrektheit der Übersetzungsfunktion

Per Definition von \bar{h} werden die **nil**-Einträge für die Nullzeiger-Adressen zu v nun nicht mehr aus h entfernt. Offenbar gilt also $\bar{h}_{\text{neu}} = \tilde{h}_{\text{neu}}$ (oben genanntes Kriterium) und somit wiederum

$$\bar{h}_{\text{neu}}, i, \eta \models s_h \llbracket \mathcal{H}'_{\text{neu}} \rrbracket \iff \mathcal{H}_{\text{neu}} \in \kappa(\lambda, \mathcal{H}'_{\text{neu}}).$$

Auf diese Weise lässt sich jeder Variablenkanten-freie Graph konstruieren. \square

Wir weiten dieses Ergebnis auf allgemeine Startgraphen aus. Diese dürfen zwar keine externen Knoten, dafür aber Variablenkanten enthalten.

LEMMA 4.19 Seien $\mathcal{H}, \mathcal{H}' \in \text{HG}_{\Sigma_N}^T$ getaggte Graphen ohne externe Knoten, λ eine Graphumgebung und η die dazu korrespondierende Prädikatinterpretation. Dann gilt

$$\mathcal{H} \in \kappa(\lambda, \mathcal{H}') \iff \beta_t(\mathcal{H}), \eta \models s_h \llbracket \mathcal{H}' \rrbracket$$

Beweis. (per Induktion über die Anzahl der Variablenkanten)

INDUKTIONSVERANKERUNG. \mathcal{H}' habe keine Variablenkanten. $\mathcal{H} \in \kappa(\lambda, \mathcal{H}')$ setzt voraus, dass \mathcal{H} ebenfalls keine Variablenkanten enthält. Nach Lemma 4.18 gilt die Behauptung.

INDUKTIONSSCHRITT. \mathcal{H}'_n enthalte n Variablenkanten. Wir fügen nun die $n + 1$ -te Variable e mit $\text{lab}(e) = x \in \text{var}_{\Sigma}$ am Knoten v' hinzu und erhalten \mathcal{H}'_{n+1} . Wir betrachten nun $\mathcal{H}_{n+1} \in \kappa(\lambda, \mathcal{H}'_{n+1})$. Dann enthält \mathcal{H}_{n+1} dieselbe zusätzliche Variablenkante an dem v' entsprechenden Knoten v .

Fall 1: In \mathcal{H}_n und \mathcal{H}'_n gab es an v und v' keine Variablenkante. Es gilt daher $t_{n+1}(v) = t'_{n+1}(v') = \{x\}$. Sei $(h, i) := \beta_t(\mathcal{H}_{n+1})$. Es gilt $x \in \text{dom}(i)$ und $i(x) = v$. In $s_h \llbracket \mathcal{H}'_{n+1} \rrbracket$ ist im Vergleich zu $s_h \llbracket \mathcal{H}'_n \rrbracket$ die anonyme Variable $t'_n(v)$ durch x ersetzt und der Quantor entfällt. Offenbar gilt

$$\beta_t(\mathcal{H}_{n+1}), \eta \models s_h \llbracket \mathcal{H}'_{n+1} \rrbracket.$$

Fall 2: v und v' hatten bereits Variablenkanten e_1, \dots, e_m .

O. B. d. A.⁶ nehmen wir an, dass $\text{cn}(x) > \text{cn}(\text{lab}(e_m)) > \dots > \text{cn}(\text{lab}(e_1))$. Für die neuen Taggings gilt dann $t_{n+1}(v) = t'_{n+1}(v') = \{x\} \cup t_{n+1}(v) = \{x, \text{lab}(e_1), \dots, \text{lab}(e_m)\}$.

Für $(h, i) := \beta_t(\mathcal{H}_{n+1})$ gilt $x \in \text{dom}(i)$ und $i(x) = v$. Sei $y := \text{lab}(e_1)$. Die neue Formel $s_h \llbracket \mathcal{H}'_{n+1} \rrbracket$ unterscheidet sich von $s_h \llbracket \mathcal{H}'_n \rrbracket$ durch Aufnahme der Gleichheitsassertion $(y = x)$ in den nicht-separierenden Konjunktionsteil.

$$s_h \llbracket \mathcal{H}'_{n+1} \rrbracket \equiv s_h \llbracket \mathcal{H}'_n \rrbracket \wedge (y = x).$$

Da $i(y) = v$ gilt, folgt

$$h, i, \eta \models s_h \llbracket \mathcal{H}'_{n+1} \rrbracket$$

und damit schließlich

$$\beta_t(\mathcal{H}_{n+1}), \eta \models s_h \llbracket \mathcal{H}'_{n+1} \rrbracket$$

\square

⁶Die Reihenfolge in der wir die Variablen zu \mathcal{H}' hinzufügen, kann entsprechend gewählt werden.

Kapitel 4. Übersetzung zwischen HRGen und SLF

Darauf aufbauend betrachten wir nun die Korrektheit von $s_H[\cdot]$ für Mengen von Graphen.

LEMMA 4.20 Sei $\mathfrak{H} \subseteq \text{HC}_{\Sigma_N}$ eine Menge von Heapkonfigurationen, \mathcal{H} ein getaggtter Graph, L_p eine Graphumgebung und η die damit korrespondierende Prädikatinterpretation. Es gilt

$$\mathcal{H} \in \kappa(\lambda, \mathfrak{H}) \iff \beta_t(\mathcal{H}), \eta \models s_H[\mathfrak{H}]$$

Beweis. Siehe Lemma 7.8 in [Dod08], Seite 161f.

Hinweis: Der Beweis baut auf der Bijektivität von α_t auf. Wir ersetzen daher α_t^{-1} im Beweis durch β_t . \square

Eine Grammatik setzt sich einerseits aus einer (Start-)Graphmenge \mathfrak{Z} und andererseits einem Regelsatz P zusammen. Das folgende Lemma aus [Dod08] befasst sich mit der Korrespondenz von P und Γ .

LEMMA 4.21 Sei $G = \langle \mathfrak{Z}, P \rangle \in \text{DSG}_{\Sigma_N}$ eine getypte Datenstrukturgrammatik und $\varphi = s[\langle \mathfrak{Z}, P \rangle] = \text{let } \Gamma \text{ in } \psi$ die daraus generierte SL-Formel. Dann korrespondiert die Graphumgebung λ zu P mit der Prädikatinterpretation, die sich durch die Semantik von \models durch Auswertung von φ auf Basis der leeren Interpretation η_0 ergibt.

Beweis. Siehe Lemma 7.9 in [Dod08], Seite 162. \square

Die Ergebnisse der vorangegangenen Lemmata werden vom folgenden Lemma für allgemeine Datenstrukturgrammatiken zusammengefasst.

LEMMA 4.22 Sei $G = \langle \mathfrak{Z}, P \rangle \in \text{DSG}_{\Sigma_N}$ getypt und $\mathcal{H} \in \text{HC}_{\Sigma_N}$ eine getaggte Heapkonfiguration. Es gilt

$$\mathcal{H} \in \mathcal{L}(G) \iff \beta_t(\mathcal{H}), \eta_0 \models s[\langle \mathfrak{Z}, P \rangle].$$

Beweis. Siehe Satz 7.10⁷ in [Dod08], Seite 162f. \square

Abschließend bringen wir diese Aussage in die zu Beginn dieses Abschnittes geforderte Form, indem wir statt β_t nun α^{-1} verwenden.

SATZ 4.23 Sei $G = \langle \mathfrak{Z}, P \rangle \in \text{DSG}_{\Sigma_N}$ getypt und $H \in \text{HC}_{\Sigma_N}$ eine Heapkonfiguration. Es gilt

$$H \in \mathcal{L}(G) \implies \forall (h, i) \in \alpha^{-1}(H) : h, i, \eta_0 \models s[G]$$

$$H \in \mathcal{L}(G) \iff \exists (h, i) \in \alpha^{-1}(H) : h, i, \eta_0 \models s[G]$$

Beweis. Aus der Definition von α und den von $s[\cdot]$ geforderten Eigenschaften des Standardtaggings (siehe Definition 4.17), ergibt sich direkt, dass die Zuordnung von Variablen und Knoten bei beiden Konzepten identisch ist: Ein Tagging t weist einem Knoten mit n Variablenkanten genau die in deren Labels enthaltenen n Variablen zu. Analog dazu generiert α zu jeder Variable $x \in \text{dom}(i)$ mit $i(x) \in \text{dom}(h)$ eine Kante. In Bezug auf freie Variablen und die dazugehörigen Kanten verhalten sich t und α^{-1} bzw. β also gleich. Eine Interpretation i , die von β_t erzeugt wird, enthält demnach im Vergleich zu derjenigen aus β lediglich die

⁷Hinweis: In [Dod08] ist hier von α die Rede, obwohl offenbar α_t gemeint ist. Im Beweis selbst wird jedoch auch dort alpha_t verwendet.

4.5. Korrektheit der Übersetzungsfunktion

zusätzlichen Speicheradressen, die sich aus t für externe Knoten und solche ohne Variablenkanten ergeben. Diese sind für die Modellrelation \models jedoch irrelevant. Das führt uns von Lemma 4.22 zum Zwischenergebnis

$$H \in \mathcal{L}(G) \iff \beta(H), \eta_0 \models s[\langle \exists, P \rangle].$$

Nach Lemma 4.17 folgt daraus direkt die Behauptung. □

Die Korrektheit von $s[\cdot]$ ist damit gezeigt.

KAPITEL 5

HAG-EIGENSCHAFTEN FÜR SL-FORMELN

Damit eine Grammatik zur Abstraktion und Konkretisierung, zum Beispiel innerhalb eines Verifikations- oder Analyseverfahrens, eingesetzt werden kann, muss sie der Klasse der Heapabstraktionsgrammatiken (HAG) angehören. Eine HAG ist nach Definition 2.22 eine produktive, wachsende, getypte und lokal konkretisierbare Datenstrukturgrammatik. Ist eine dieser vier Eigenschaften nicht gegeben, kann es zu unerwünschten Effekten, wie beispielsweise nicht-terminierenden Abstraktionsvorgängen oder ungültigen Zwischenergebnissen, also Graphen, die keine Heapkonfigurationen sind, kommen.

Je nach Anwendung kann es also von großer Wichtigkeit sein, ob eine synthetisierte Grammatik eine HAG ist, oder zumindest einige der Anforderungen erfüllt. In Abschnitt 4.2.4 wurde bereits ein notwendiges und hinreichendes Kriterium für Formeln angegeben, das die DSG-Eigenschaften synthetisierter Grammatiken sicherstellt.

Dieses Kapitel überträgt die vier HAG-Anforderungen Produktivität, Getyptheit, Wachstum und lokale Konkretisierbarkeit auf Formeln und gibt entsprechende Kriterien an. Diese garantieren einerseits die jeweilige Eigenschaft für synthetisierte Grammatiken und sind andererseits bei der Generierung einer Formel aus einer DSG mit dieser Eigenschaft selbst stets erfüllt.

Die folgenden Abschnitte widmen sich je einem HAG-Aspekt, formulieren sein Analogon für Formeln und zeigen, dass das Kriterium unter $g[\cdot]$ und $s[\cdot]$ erhalten bleibt.

5.1. PRODUKTIVITÄT

Die erste betrachtete Eigenschaft von Heapabstraktionsgrammatiken ist die Produktivität. Zur Erinnerung: Eine DSG ist genau dann produktiv, wenn jedes Nichtterminal produktiv ist, also einen konkreten Graphen erzeugen kann. Es muss also

$$\forall X \in N. \quad L(X^\bullet) \neq \emptyset$$

gelten. Die Menge der produktiven Nichtterminale ist induktiv definiert: $X \in N$ ist produktiv, wenn eine Regel $X \rightarrow H$ existiert, so dass H konkret ist, oder alle vorkommenden Nichtterminale produktiv sind.

Um eine analoge Eigenschaft für Formeln zu definieren, muss untersucht werden, welches Prädikat einem nichtproduktiven Nichtterminal entspricht. Für letzteres existieren ausschließlich Regeln die weitere nichtproduktive Nichtterminale enthalten. Es gibt also keine Ableitung, die in einem konkreten Graphen endet. Übersetzen wir diese Nichtterminale in Prädikate,

Kapitel 5. HAG-Eigenschaften für SL-Formeln

so ergeben sich Prädikatdefinition, die immer weitere Prädikate aufrufen ohne dabei jemals zu einer direkt auswertbaren Formel, also einer Formel, die frei von Prädikatsaufrufen ist, zu gelangen. Da dies für alle Regeln gilt, besteht die resultierende Prädikatdefinition aus einer Disjunktion von ausschließlich „unendlich rekursiven“ Termen. Eine solche Formel ist offenbar nicht auswertbar. Für die formale Definition auswertbarer Prädikate benötigen wir zunächst den Begriff der primitiven Formeln.

DEFINITION 5.1 (Primitive Formel) Eine Formel $\varphi \in \text{SLF}$ ist *primitiv* genau dann, wenn sie **let**-frei ist und keine Prädikatsaufrufe enthält.

Darauf aufbauend definieren wir auswertbare Prädikate wie folgt.

DEFINITION 5.2 (Auswertbares Prädikat) Die Menge von *auswertbaren Prädikaten* Π ist definiert als die kleinste Menge, so dass:

- $\sigma(x_1, \dots, x_n) = \psi$ und es existiert $\psi' \in \text{dnf}(\psi)$, so dass ψ' primitiv ist $\implies \sigma \in \Pi$
- $\sigma(x_1, \dots, x_n) = \psi$ und es existiert $\psi' \in \text{dnf}(\psi)$, so dass für alle Prädikate σ' , die in ψ' aufgerufen werden, gilt $\sigma' \in \Pi \implies \sigma \in \Pi$.

Die Menge der Auswertbarkeit ist äquivalent zur Menge produktiver Nichtterminale. Das folgende Lemma stellt den Bezug zwischen produktiven Grammatiken und auswertbaren Prädikaten her.

LEMMA 5.1

1. Für jede Formel, die kein Prädikat $\sigma \notin \Pi$ enthält, gilt:

$$g[\varphi] \text{ ist produktiv}$$

2. Für jede Grammatik $G \in \text{DSG}_{\Sigma_N}$, die produktiv ist, gilt:

$$s[G] \text{ enthält kein Prädikat } \sigma \notin \Pi.$$

Beweis. (1.) – durch strukturelle Induktion. Sei $\langle \exists, P \rangle = g[\varphi]$.

INDUKTIONSVERANKERUNG. Sei $\sigma \in \Pi$, $\sigma(x_1, \dots, x_n) = \psi$, so dass ein primitives $\psi' \in \text{dnf}(\psi)$ existiert. Dann gibt es eine Regel $X_\sigma \rightarrow h[\psi] \in P$, so dass $\text{lab}(E_{h[\psi]}) \cap N = \emptyset$ ($h[\psi]$ enthält keine Nichtterminalkante). X_σ ist also produktiv.

INDUKTIONSHYPOTHESE. Die auswertbaren Prädikate $\sigma_1, \dots, \sigma_m \in \Pi$ generieren produktive Nichtterminale $X_{\sigma_1}, \dots, X_{\sigma_m}$.

INDUKTIONSSCHRITT. Sei $\sigma(x_1, \dots, x_n) = \psi$ ein auswertbares Prädikat, so dass ein Disjunkt $\psi' \in \text{dnf}(\psi)$ existiert, das die Prädikate $\sigma_1, \dots, \sigma_m$ (und sonst keine) aufruft. Dann gibt es eine Regel $X_\sigma \rightarrow h[\psi] \in P$, so dass $\text{lab}(E_{h[\psi]}) = \{X_{\sigma_1}, \dots, X_{\sigma_m}\}$. Demnach ist X_σ produktiv. Da der Beweis der selben Struktur folgt, wie die induktive Definition von Π , ist er für alle $\sigma \in \Pi$ gültig. Es können also immer Prädikate $\sigma_1, \dots, \sigma_m$ gefunden werden, für die die I.H. gilt, so dass schließlich alle Prädikate in Π abgedeckt werden.

(2.) – strukturelle Induktion analog zu (1.)

Sei $G = (\mathcal{J}, P)$ produktiv. Dann ist jedes Nichtterminal X , für das eine Regel $X \rightarrow H \in P$ existiert, produktiv.

INDUKTIONSV ERANKERUNG. Sei X ein Nichtterminal, $H \in \text{DSG}_{\Sigma_N}$ konkret und $X \rightarrow H \in P$. $s_P[\cdot]$ übersetzt X zu $\sigma_X(x_1, \dots, x_n) = s_H[\text{RHS}_P(X)]$, wobei $\text{RHS}_P := \{H \mid X \rightarrow H \in P\}$. Wegen $H \in \text{RHS}_P(X)$ enthält $s_H[\text{RHS}_P(X)] = \psi_1 \vee \dots \vee \psi_m$, ein Disjunkt $\psi_j = s_H[H]$, das primitiv ist. Daher ist σ_X auswertbar.

INDUKTIONSHYPOTHESE. Seien X_1, \dots, X_m produktive Nichtterminale, so dass $\sigma_{X_1}, \dots, \sigma_{X_m}$ auswertbar sind.

INDUKTIONSSCHRITT. Sei X ein produktives Nichtterminal und $X \rightarrow H \in P$, so dass $\text{lab}(E_H) = \{X_1, \dots, X_m\}$. Die Prädikatdefinition von σ_X ist gegeben durch $\sigma_X(x_1, \dots, x_n) = s_H[\text{RHS}_P(X)]$ und es gilt wiederum $H \in \text{RHS}_P(X)$. Diese enthält das Disjunkt $s_h[H]$, wobei $s_h[\cdot]$ alle Nichtterminale in Aufrufe der entsprechenden Prädikate verwandelt. $s_h[H]$ enthält demnach ausschließlich Aufrufe von $\sigma_{X_1}, \dots, \sigma_{X_m}$, die nach I.H. auswertbar sind. Da die Existenz eines einzigen Disjunks für die Definition genügt, ist σ_X ebenfalls auswertbar.

Weil jede Ableitung eines konkreten Graphen irgendwann Nichtterminale erreichen muss, wie sie in I.V. beschrieben sind, gilt der Beweis für alle produktiven Nichtterminale. \square

5.2. GETYPTHEIT

Im Gegensatz zur Produktivität handelt es sich bei der Getyptheit um eine Eigenschaft, die für die Übersetzung selbst essenziell ist. Ohne die Typinformationen von Nichtterminalen ist unklar, welche Selektoren ein Tentakel erzeugt. Die Übersetzungsfunktion $s_h[\cdot]$ weiß in diesem Fall nicht, welche Selektoren sie auf **nil** setzen darf und welche Selektoren von aufgerufenen Prädikaten realisiert werden.

Formalisiert wurde dies durch: Wenn G getypt ist, gibt es eine Funktion $\text{type}(X, j) \subseteq \Sigma$, $X \in N$, $1 \leq j \leq \text{rk}(X)$, so dass $\forall H \in L(X^\bullet)$. $\text{type}(X, j) = \text{lab}_H(\text{out}_H(\text{ext}_H(j)))$.

Erzeugen wir jedoch eine Grammatik aus einer Formel, so ist die Getyptheit im Allgemeinfall keinesfalls gewährleistet. Wir müssen die Menge der SLF-Formeln dazu zunächst – wie schon durch die Auswertbarkeit – einschränken und schließlich beweisen, dass die entsprechenden Teilmengen von Formeln und Graphen äquivalent sind.

DEFINITION 5.3 (Getyptheit von Formeln) Eine Formel $\varphi \in \text{SLF}$ ist *getypt* genau dann, wenn für alle definierten und aufgerufenen Prädikate $\sigma(x_1, \dots, x_n) = \psi$ und $h, h' \in \text{He}$, $i, i' \in \text{Int}$, mit $h, i \models \psi$ und $h', i' \models \psi$, so dass für alle $1 \leq j \leq n, s \in \text{sel}_\Sigma$, gilt:

$$\llbracket x_j \rrbracket i + \text{cn}(s) \in \text{dom}(h) \iff \llbracket x_j \rrbracket i' + \text{cn}(s) \in \text{dom}(h') \quad (5.1)$$

$$\text{und } h(\llbracket x_j \rrbracket i + \text{cn}(s)) \neq \mathbf{nil} \iff h(\llbracket x_j \rrbracket i' + \text{cn}(s)) \neq \mathbf{nil} \quad (5.2)$$

Anschaulich: Die Formel ist genau dann getypt, wenn für jedes Prädikat, jeden Parameter und für alle Heaps, die die Prädikatdefinition erfüllen, und für jeden Selektor gilt, dass dieser Selektor entweder auf all diesen Heaps definiert ist, oder auf keinem (5.1). Außerdem verweist er entweder überall auf **nil** oder nirgendwo (5.2).

Das nachfolgende Lemma zeigt die Äquivalenz beider Getyptheits-Definitionen.

LEMMA 5.2 (Äquivalenz der Getyptheit)

1. Für jede getypte Formel $\varphi \in \text{SLF}$ gilt

$$g[\![\varphi]\!] \text{ ist getypt.}$$

2. Für jede getypte Grammatik $G \in \text{DSG}$ gilt

$$s[\![G]\!] \text{ ist getypt.}$$

Beweis.

(1.) Sei $\varphi \in \text{SLF}$ getypt.

Annahme: $G := g[\![\varphi]\!]$ ist nicht getypt.

Dann existieren $X \in N_G$, $j \in \mathbb{N}$ mit $1 \leq j \leq \text{rk}(X)$, $H_1, H_2 \in L(X^\bullet)$, so dass

$$\text{lab}_{H_1}(\text{out}_{H_1}(\text{ext}_{H_1}(j))) \neq \text{lab}_{H_2}(\text{out}_{H_2}(\text{ext}_{H_2}(j)))$$

oder genauer (o. B. d. A.): Es existiert $s \in \text{sel}_\Sigma$, so dass

$$s \in \text{lab}_{H_1}(\text{out}_{H_1}(\text{ext}_{H_1}(j))), \quad s \notin \text{lab}_{H_2}(\text{out}_{H_2}(\text{ext}_{H_2}(j)))$$

Es gibt also eine Kante $e \in E_{H_1}$, $\text{lab}_{H_1}(e) = s$, $\text{att}_{H_1}(e)(1) = \text{ext}_{H_1}(j)$, aber keine Kante $e' \in E_{H_2}$, so dass $\text{att}_{H_2}(e')(1) = \text{ext}_{H_2}(j)$ und $\text{lab}_{H_2}(e') = s$ gelten. (†)

Wegen $H_1, H_2 \in L(X^\bullet)$ gibt es ein Prädikat σ_X , aus dem das Nichtterminal X und seine Ableitungsregeln generiert wurden. In der Definition von σ_X , $\sigma_X(x_1, \dots, x_{\text{rk}(X)}) = \psi$ existieren also zwei Disjunkte $\psi_1, \psi_2 \in \text{dnf}(\psi)$, für die gilt

$$H_1 \in L(h[\![\psi_1]\!]), \quad H_2 \in L(h[\![\psi_2]\!]).$$

Da e, e' aus (†) Terminalkanten sind (wegen $s \in \Sigma$), können sie ausschließlich von $h[\![\cdot]\!]$ bei der Übersetzung einer Zeigerassertion entstanden sein (siehe Definition 4.6). Daher gilt für zwei Heaps $h_1, h_2 \in \text{He}$ und Interpretationen $i_1, i_2 \in \text{Int}$ mit $h_1, i_1 \models \psi_1$ und $h_2, i_2 \models \psi_2$ entweder (j bezeichnet noch immer denselben Index wie oben)

$$[\![x_j]\!]i_1 + \text{cn}(s) \in \text{dom}(h_1) \quad \text{und} \quad [\![x_j]\!]i_2 + \text{cn}(s) \notin \text{dom}(h_2)$$

oder

$$h_1([\![x_j]\!]i_1) + \text{cn}(s) \neq \mathbf{nil} \quad \text{und} \quad h_2([\![x_j]\!]i_2) + \text{cn}(s) = \mathbf{nil}.$$

Widerspruch in beiden Fällen.

(2.) Sei $G = \langle \mathfrak{J}, P \rangle \in \text{DSG}_{\Sigma_N}$ getypt.

Annahme: $\varphi = s[\![G]\!]$ ist nicht getypt.

Dann existiert ein in φ definiertes Prädikat σ

$$\sigma(x_1, \dots, x_n) = \psi,$$

das bei der Übersetzung eines Nichtterminals $X_\sigma \in N_G$ generiert wurde.

ψ ist per Konstruktion in disjunktiver Normalform (siehe Definition 4.21), also

$$\psi \doteq \bigvee_{\psi' \in \text{dnf}(\psi)} \psi' \quad ^1$$

und jedes Disjunkt $\psi' \in \text{dnf}(\psi)$ ist die Übersetzung eines Graphen H von einer Ableitungsregel $X_\sigma \rightarrow H \in P$:

$$\psi' = g \llbracket H \rrbracket$$

Da $g \llbracket \cdot \rrbracket$ keine **nil**-Zeigerassertionen der Form $t(v).s \mapsto \mathbf{nil}$ für externe Knoten $v \in \text{ext}$ erstellt (siehe Definition 4.18) und somit auch keine Assertion $x.s \mapsto \mathbf{nil}$ für unquantifizierte $x \in \text{Var}$, existieren zur Gewährleistung der Annahme j , $1 \leq j \leq \text{rk}(X_\sigma)$ und $s \in \text{sel}_\Sigma$, so dass für den Parameter x_j gilt:

$$\exists h, h' \in \text{He}, i, i' \in \text{Int}. \quad h, i \models \psi, \quad h', i' \models \psi \quad ^2 \quad \text{und}$$

$$\llbracket x_j \rrbracket i + \text{cn}(s) \in \text{dom}(h) \quad \wedge \quad \llbracket x_j \rrbracket i' + \text{cn}(s) \notin \text{dom}(h')$$

Die erste Zeile impliziert dabei

$$\exists \psi', \psi'' \in \text{dnf}(\psi), \quad h, i \models \psi' \quad \wedge \quad h', i' \models \psi''$$

Damit die zweite Zeile gilt, muss ψ' oder eines der darin aufgerufenen Prädikate die Assertion $x_j.s \mapsto y$ für ein $y \in \text{Var}$ enthalten (ein Heap wird durch eine SLF-Formel stets *vollständig* spezifiziert), wohingegen ψ'' und seine Prädikate diese nicht enthalten dürfen. Da jedes Disjunkt von ψ durch den Graphen einer rechten Regelseite generiert wird, existieren $H', H'' \in \text{HC}_{\Sigma_N}$ mit $X_\sigma \rightarrow H', X_\sigma \rightarrow H'' \in P$, so dass

$$\psi' = s_h \llbracket H' \rrbracket \quad \text{und} \quad \psi'' = s_h \llbracket H'' \rrbracket$$

Zeigerassertionen, die nicht auf **nil** verweisen, werden im Rahmen von $s \llbracket \cdot \rrbracket$ ausschließlich von $s_e \llbracket \cdot \rrbracket$ bei der Übersetzung von Selektoren erstellt. Deshalb gilt für alle $\tilde{H}' \in L(H')$, $\tilde{H}'' \in L(H'')$

$$\exists e' \in E_{\tilde{H}'}, \text{lab}_{\tilde{H}'}(e') = s, \text{att}_{\tilde{H}'}(e')(1) = \text{ext}_{\tilde{H}'}(j)$$

$$\nexists e'' \in E_{\tilde{H}''}, \text{lab}_{\tilde{H}''}(e'') = s, \text{att}_{\tilde{H}''}(e'')(1) = \text{ext}_{\tilde{H}''}(j)$$

Es kann demnach keine Funktion type , wie sie in Definition 2.18 beschrieben ist, geben, da $\tilde{H}' \in L(H')$ und $\tilde{H}'' \in L(H'')$ existieren, so dass

$$\text{lab}_{\tilde{H}'}(\text{out}_{\tilde{H}'}(\text{ext}_{\tilde{H}'}(j))) \neq \text{lab}_{\tilde{H}''}(\text{out}_{\tilde{H}''}(\text{ext}_{\tilde{H}''}(j)))$$

obwohl $\tilde{H}' \in L(X_\sigma)$ und $\tilde{H}'' \in L(X_\sigma)$ und $X_\sigma \in \text{lab}(E_G)$.

G ist also nicht getypt. Widerspruch.

¹ $\alpha \doteq \beta$, bedeutet hier, dass α und β bis auf die Anordnung kommutativer Operanden gleich sind.

²o. B. d. A. nehmen wir an, dass alle in φ definierten Prädikate verfügbar sind.

5.3. WACHSTUM

Die Wachstumseigenschaft stellt sicher, dass Prädikate weder sich, noch andere Prädikate aufrufen, ohne die Menge der erfüllenden Heaps und Interpretationen weiter einzuzugrenzen. Dies soll unendliche Rekursion verhindern. Dass die Auswertbarkeit der Formel dazu alleine nicht reicht, soll das folgende Beispiel verdeutlichen

$$\sigma(x_1, x_2) = x_1.s \mapsto x_2 \vee \sigma(x_2, x_1)$$

Dieses Prädikat ist offensichtlich auswertbar, da $x_1.s \mapsto x_2$ primitiv ist. Die Wachstumseigenschaft ist jedoch nicht erfüllt, da σ erneut aufgerufen wird, ohne dass der Heap im entsprechenden Disjunkt näher spezifiziert wird, z.B. durch andere Prädikataufrufe oder Zeigerassertionen. Wir formalisieren diese Eigenschaft in der folgenden Definition

DEFINITION 5.4 (Wachstum) Sei $\varphi \in \text{SLF}$. φ erfüllt die *Wachstumseigenschaft*, wenn für jedes Prädikat σ mit $\sigma(x_1, \dots, x_n) = \psi$ und jede Variable $x \in \text{FV}(\psi)$ und $h, i, \eta \models \psi$

- $\llbracket x \rrbracket i \in \text{img}(h)$ ODER
- es ex. $s \in \text{sel}_\Sigma$ so dass $h(\llbracket x \rrbracket i + \text{cn}(s)) \neq \text{nil}$

und für jedes $\psi' \in \text{dnf}(\psi)$

- ψ' ist primitiv ODER
- Die Summe der Anzahlen der Prädikataufrufe, Zeigerassertionen und der verwendeten Variablen ist (echt) größer als $n + 1$.

gilt.

Als „verwendete Variablen“ bezeichnen wir hier Variablen, über die Aussagen in Form von Assertionen oder Prädikataufrufen getroffen werden. Eine Variable, die ausschließlich quantifiziert, jedoch danach in der Formel nicht mehr erwähnt wird, zählt nicht dazu. Die Variable r in $\exists r : \sigma(x)$ würde also nicht in die Summe eingehen, da über sie keine Aussage getroffen wird und sie daher keinen Knoten repräsentiert.

LEMMA 5.3 (Äquivalenz der Wachstumseigenschaft)

1. Für jede wachsende Formel $\varphi \in \text{SLF}$ gilt:

$$g[\varphi] \text{ ist wachsend.}$$

2. Für jede wachsende Grammatik $G \in \text{DSG}_{\Sigma_N}$ gilt:

$$s[G] \text{ ist wachsend.}$$

Beweis. (1.) Sei $\varphi \in \text{SLF}$ wachsend.

Annahme: $g[\varphi]$ ist nicht wachsend.

Fall 1: Es gibt eine Regel zu $X_\sigma \in N$ in $g[[\varphi]]$, deren rechte Regelseite einen isolierten Knoten enthält. Damit bei der Synthese keine Kante zu diesem Knoten erzeugt wird, muss es bei $\sigma(x_1, \dots, x_n) = \psi$ ein Disjunkt geben, das einer Variablen x in jedem Feld **nil** zuweist, sie nicht als Argument an ein Prädikat übergibt und auch nicht als Ziel eines Zeigers verwendet. Dann existieren h, i, η so dass

$$h, i, \eta \models \psi \quad \text{und} \quad \forall s \in \text{sel}_\Sigma : h(\llbracket x \rrbracket i + \text{cn}(s)) = \mathbf{nil} \quad \text{und} \quad x \notin \text{img}(h)$$

Widerspruch.

Fall 2: Es gibt in $g[[\varphi]]$ eine Regel $X \rightarrow H$ wobei H weder konkret ist noch $|E_H| + |V_H| > \text{rk}(H) + 1$ gilt. Da primitive Formeln bei der Synthese per Konstruktion in Nichtterminalfreie, also konkrete Hypergraphen übersetzt werden, ist φ nicht primitiv.

Jede Kante in H ist aus einem Prädikataufruf oder einer Zeigerassertion $x.s \mapsto y$ mit $y \in \text{Var}$ synthetisiert worden. Knoten werden bei der Synthese von Kanten stets automatisch erzeugt und anschließend, dem Tagging entsprechend, verschmolzen. Da es keine Gleichheitsassertionen in Prädikatdefinitionen gibt, wird aus jeder Variable, die als Argument übergeben, oder in einer Zeigerassertion verwendet wird, genau ein Knoten generiert.

Der Rang eines Nichtterminals entspricht der Anzahl der Parameter des Prädikats, aus dem es erzeugt wurde. Wegen $|E_H| + |V_H| \leq \text{rk}(H) + 1$ ist also die Summe der verwendeten Variablen und der Nicht-Nullzeigerassertion sowie der Prädikataufrufe kleiner oder gleich der Anzahl der Parameter des betrachteten Prädikats plus 1. φ ist also nicht wachsend. Widerspruch.

(2.) Sei $G \in \text{DSG}_{\Sigma_N}$ wachsend.

Isolierte Knoten sind in wachsenden DSGen verboten. Jeder Knoten v besitzt also mindestens eine eingehende oder ausgehende Kante e .

Fall 1: Eine eingehende Kante wird in eine Assertion $x.s \mapsto t(v)(1)$ für ein $x \in \text{Var}, s \in \text{sel}_\Sigma$ übersetzt. Demnach ist für $h, i, \eta \models \psi$ die Bedingung $\llbracket t(v)(1) \rrbracket i \in \text{img}(h)$ erfüllt.

Fall 2: Aus einer ausgehenden Kante generiert $s[\cdot]$ die Assertion $t(v)(1).s \mapsto y$, $y \in \text{Var}$ mit $s = \text{lab}(e)$. Nach Definition von \models ist $h(\llbracket t(v)(1) \rrbracket i + \text{cn}(s)) \neq \mathbf{nil}$ also erfüllt.

Da sämtliche Variablen $x \in \text{FV}(\psi)$ aus Taggings von Knoten entstehen, ist stets eine der beiden ersten Bedingungen von Definition 5.4 erfüllt. Wegen der Wachstumseigenschaft von $G = \langle \mathfrak{Z}, P \rangle$ gilt für alle Regeln $X \rightarrow H \in P$, dass H konkret ist oder $|V_H| + |E_H| > \text{rk}(X) + 1$ gilt.

Fall 1: H ist konkret. Dann ist $s_h[[H]]$ primitiv und $X \rightarrow H$ erzeugt ein primitives Disjunkt $\psi' \in \text{dnf}(\psi)$.

Fall 2: $|V_H| + |E_H| > \text{rk}(X) + 1$. X erzeugt das Prädikat σ_X , das $\text{rk}(X)$ Parameter hat. Es gilt also $n = \text{rk}(X)$. Aus jedem Knoten entsteht genau eine Variable, da Gleichheitsassertionen in Prädikatdefinitionen nicht auftreten. Jede Kante erzeugt entweder eine

Zeigerassertion oder einen Prädikataufruf. Damit ist die Summe der Anzahlen der Variablen, Zeigerassertionen und Prädikataufrufe genau $|V_H| + |E_H|$ und daher ebenfalls größer als $n + 1$.

Da stets mindestens einer der beiden Fälle zutrifft, gilt die Behauptung für alle $\psi' \in \text{dnf}(\psi)$. $s[[G]]$ ist also ebenfalls wachsend. \square

5.4. LOKALE KONKRETISIERBARKEIT

Eine wichtige Eigenschaft für HRGen ist die *lokale Konkretisierbarkeit*, die die Möglichkeit der Konkretisierung jedes Nichtterminals jeweils von allen Seiten durch nur einen Ableitungsschritt sicherstellt. Dieselbe Fähigkeit ist auch für die Beweisregeln der Separation Logic von essentieller Bedeutung, um endliche Beweisbäume zu gewährleisten. Kapitel 6 wird auf diese Regeln detailliert eingehen.

Konkretisieren bedeutet aus Sicht von Prädikaten, einen Prädikataufruf durch eine Formel zu ersetzen, die aus der Definition des Prädikats impliziert wird. Während bei Hypergraphen die lokale Konkretisierung, also eine Ableitung, die an einem bestimmten Knoten ausschließlich Terminalkanten erzeugt, betrachtet wird, sind für Prädikate unmittelbare Aussagen für bestimmte Parameter interessant. Die Konkretisierung ist hier also weniger ortsbezogen sondern vielmehr variablenspezifisch. Intuitiv ist ein Prädikat variablenspezifisch konkretisierbar, wenn sich für jeden Parameter x eine Teilformel der Definition finden lässt, so dass diese einerseits (unter Berücksichtigung der Semantik der anderen Prädikate) äquivalent zur gesamten Definition ist und andererseits alle Zeiger, die das Prädikat auf x definiert, ohne weitere Prädikataufrufe (also mit primitiven Formelteilen) erzeugt werden. Die folgende Definition formalisiert diese Aussage. Die dabei verwendete Prädikatinterpretation η_{emp} sieht jedes Prädikat mit allen möglichen Argumenten genau von h_0 , also dem leeren Heap als erfüllt an. Dies ist nicht zu verwwechseln mit η_0 , dessen Definitionsbereich leer ist.

DEFINITION 5.5 (Variablenspezifische Konkretisierbarkeit) Sei $\varphi = \text{let } \Gamma \text{ in } \varphi'$ und η die durch Γ definierte Prädikatinterpretation. Ferner bezeichne η_{emp} die Prädikatinterpretation, für die gilt: $\text{dom}(\eta_{\text{emp}}) = \text{dom}(\eta)$ und $\forall \sigma \in \text{dom}(\eta_{\text{emp}}) : \eta_{\text{emp}}(\sigma) = \text{Loc}^{\text{rk}(\sigma)} \times \{h_0\}$. φ ist *variablenspezifisch konkretisierbar*, wenn es für jedes darin definierte Prädikat $\sigma(x_1, \dots, x_n) = \psi$ und alle $j \in \mathbb{N}$, $1 \leq j \leq n$ eine Menge $\Psi_j \subseteq \text{dnf}(\psi)$ gibt, so dass

1. Für alle $h \in \text{He}$, $i \in \text{Int}$

$$h, i, \tilde{\eta}_j \models \sigma(x_1, \dots, x_n) \iff h, i, \eta \models \sigma(x_1, \dots, x_n)$$

Wobei $\tilde{\eta}_j$ die Prädikatinterpretation beschreibt, die entsteht, wenn man in Γ die Definition $\sigma(x_1, \dots, x_n) = \psi$ durch $\sigma(x_1, \dots, x_n) = \bigvee \Psi_j$ ersetzt.

2. Für alle $\psi' \in \Psi_j$, $h, h' \in \text{He}$, $i \in \text{Int}$ gilt

$$\begin{aligned} h, i, \eta \models \psi \wedge h', i, \eta_{\text{emp}} \models \psi' \\ \implies \forall l \in \text{dom}(h) \cap L_j \text{ mit } h(l) \neq \mathbf{nil} : l \in \text{dom}(h') \wedge h'(l) \neq \mathbf{nil}. \end{aligned}$$

wobei $L_j = \{l \in \text{Loc} \mid \exists s \in \text{sel}_\Sigma, l = \llbracket x_j \rrbracket i + \text{cn}(s)\}$.

ERLÄUTERUNG:

1. σ wird unter η , also der „echten“ Prädikatinterpretation zu Γ von den gleichen Heaps beziehungsweise Variableninterpretationen erfüllt, wie unter $\tilde{\eta}_j$, für die σ nur aus den zu j gehörigen Disjunkten besteht. Die Kernaussage ist also, dass die Disjunktion aller Formeln in Ψ_j das Prädikat vollständig repräsentiert. Äquivalent (\equiv) sind ψ und $\bigvee \Psi_j$ dennoch nicht zwangsweise, da alle anderen Prädikate in η durch $\tilde{\eta}_j$ nicht verändert werden und die identische Auswertung von σ auf beiden Seiten nach wie vor von diesen abhängen kann.
2. L_j beschreibt die Menge der Heapadressen, die Selektoren von x_j repräsentieren. h erfüllt die gesamte Prädikatdefinition. h' hingegen erfüllt das Disjunkt ψ' , das der für j ausgewählten Menge Ψ_j entstammt. Der Einsatz von η_{emp} anstelle von η sorgt dafür, dass h' nur diejenigen Heapzellen enthält, die von ψ' direkt und nicht etwa durch darin geschachtelte Prädikataufrufe definiert werden. Jedes in ψ' aufgerufene Prädikat beschreibt dadurch lediglich den leeren Heap h_0 . Die Formel $\sigma(y, x) * x.s \mapsto y$ zum Beispiel entspricht unter η_{emp} also $\mathbf{emp} * x.s \mapsto y$, was zu $x.s \mapsto y$ äquivalent ist.

Die Implikation sagt nun aus, dass sämtliche zu x_j gehörende Selektoren, die in h definiert sind und nicht auf \mathbf{nil} zeigen (also eine ausgehende Kante repräsentieren) auch in h' vorhanden sind und ebenfalls nicht auf \mathbf{nil} zeigen. Die Verwendung von η_{emp} entspricht dem Verbot weiterer Ableitungen. h' enthält also nur diejenigen Selektoren, deren zugehörige Selektorkanten in einem Ableitungsschritt erzeugt werden können.

Wir zeigen nun die Äquivalenz von lokaler und variablenspezifischer Konkretisierbarkeit.

LEMMA 5.4 (Äquivalenz der lokalen und der variablenspezifischen Konkretisierbarkeit)

1. Sei $\varphi \in \text{SLF}$ getypt und variablenspezifisch konkretisierbar. Dann gilt

$g\llbracket\varphi\rrbracket$ ist lokal konkretisierbar.

2. Sei $G \in \text{DSG}_{\Sigma_N}$ getypt und lokal konkretisierbar. Dann gilt

$s\llbracket G\rrbracket$ ist variablenspezifisch konkretisierbar.

Beweis. (1.)

Sei $\varphi \in \text{SLF}$ variablenspezifisch konkretisierbar und $G = \langle \mathfrak{J}, P \rangle = g\llbracket\varphi\rrbracket$. Wir zeigen, dass es für alle Nichtterminale $X \in N_G$ Regelmengen $P_{(X,1)}, \dots, P_{(X,\text{rk}(X))} \subseteq P^X$ gibt, so dass 1. und 2. aus Definition 2.21 gelten.

Nichtterminale in Grammatiken entsprechen Prädikaten in Formeln. Sei $\sigma(x_1, \dots, x_n) = \psi$ ein beliebiges Prädikat aus Γ . $r\llbracket\cdot\rrbracket$ übersetzt $\sigma(x_1, \dots, x_n) = \psi$ unter Verwendung von $h\llbracket\cdot\rrbracket$ in eine Menge von Produktionsregeln P^{X_σ} (siehe Definition 4.7). Die rechten Seiten dieser Regeln sind dabei die Graphen der Menge $h\llbracket\psi\rrbracket$, wobei die Knoten zu den Parametern x_1, \dots, x_n exponiert werden. Jedes $\psi' \in \text{dnf}(\psi)$ bildet dabei einen solchen Graphen.

Kapitel 5. HAG-Eigenschaften für SL-Formeln

Als Mengen $P_{(X,1)}, \dots, P_{(X, \text{rk}(X))}$ verwenden wir die Regelsätze, die aus $h[\bigvee \Psi_1], \dots, h[\bigvee \Psi_n]$ entstehen, wobei offensichtlich $h[\bigvee \Psi_j] = \bigcup_{\psi' \in \Psi_j} h[\psi']$ gilt (*). Die Regelsätze sind also

$$P_{(X_\sigma, j)} = \{X_\sigma \rightarrow h[\psi'] \mid \psi' \in \Psi_j\}.$$

Im Folgenden bezeichnet $\langle H \rangle$ den getaggten Hypergraphen (H, t) , wobei t das Standardtagging für H ist. Die Unterscheidung zwischen getaggten und einfachen Hypergraphen erfolgt in diesem Beweis nur aus syntaktisch-formalen Gründen. Für das Verständnis ist sie hier weitestgehend irrelevant, da überall das Standardtagging verwendet wird, im Tagging selbst also keine weiteren Informationen enthalten sind. Daher schreiben wir in diesem Beweis abkürzend $\mathcal{H}, \overline{\mathcal{H}}, \tilde{\mathcal{H}}, \mathcal{K}$ usw. für $\langle H \rangle, \langle \overline{H} \rangle, \langle \tilde{H} \rangle, \langle K \rangle$. Wir zeigen nun für ein beliebiges $j \in \mathbb{N}$, $1 \leq j \leq n$, die Gültigkeit von 1. und 2. aus Definition 2.21.

1. – Zu zeigen: $L_{P_{(X,j)} \cup \overline{P^X}}(X_\sigma^\bullet) = L_P(X_\sigma^\bullet)$

Es gilt $L_P(X_\sigma^\bullet) = \kappa(\lambda_P, \langle X_\sigma^\bullet \rangle)$, wobei λ_P die Graphumgebung zu P ist. Wir betrachten nun ein beliebiges $\mathcal{H} \in \kappa(\lambda_P, \langle X_\sigma^\bullet \rangle)$ und $(h, i) = \beta_t(\mathcal{H})$.

Die externen Knoten von X_σ^\bullet , v_1, \dots, v_n , sind in \mathcal{H} mit x_1, \dots, x_n getaggt. Ihre Adressen in h sind entsprechend $[[x_1]]i, \dots, [[x_n]]i$. Nach Definition von $h[\cdot]$ wird ausschließlich ein Aufruf $\sigma(y_1, \dots, y_n)$ zu einer Nichtterminalkante mit dem Label X_σ übersetzt. Es folgt

$$h[\sigma(x_1, \dots, x_n)] = X_\sigma^\bullet.$$

Für eine mit λ_P korrespondierende Prädikatinterpretation η gilt nach Lemma 4.9:

$$\mathcal{H} \in \kappa(\lambda_P, \langle X_\sigma^\bullet \rangle) \iff \beta_t(\mathcal{H}, \eta) \models \sigma(x_1, \dots, x_n).$$

Nach Konstruktion korrespondiert $\tilde{\eta}_j$ offensichtlich mit $\lambda_{P_{(X,j)} \cup \overline{P^X}}$, der Graphumgebung zu $P_{(X,j)} \cup \overline{P^X}$. Nach Lemma 4.9 folgt wiederum

$$\mathcal{H} \in \kappa(\lambda_{P_{(X,j)} \cup \overline{P^X}}, \langle X_\sigma^\bullet \rangle) = L_{P_{(X,j)} \cup \overline{P^X}}(X_\sigma^\bullet).$$

Die Gegenrichtung gilt analog. Es folgt die Behauptung.

2. – Zu zeigen: Für alle $a \in \text{type}(X_\sigma, j)$, $(X_j \rightarrow K) \in P_{(X_\sigma, j)}$ existiert $e \in E_K$, so dass $\text{lab}_K(e) = a$ und $\text{att}_K(e)(1) = \text{ext}_K(j)$.

Nach Lemma 5.2 ist $g[\varphi]$ getypt. Seien $a \in \text{type}(X_\sigma, j)$ und $(X_\sigma \rightarrow K) \in P_{(X_\sigma, j)}$ beliebig gewählt. Gemäß (*) gibt es $\psi' \in \Psi_j$, so dass $K' = h[\psi']$ und $K = \text{expose}(x_1 \dots x_n, K')$. (†)

Für alle Hypergraphen $H \in L(X_\sigma^\bullet)$ gilt wegen der Getyptheit

$$a \in \text{lab}_H(\text{out}_H(\text{ext}_H(j)))$$

(a ist Label einer ausgehenden Kante des j -ten externen Knotens).

Seien nun $h \in \text{He}$, $i \in \text{Int}$, $\overline{H} \in L(X_\sigma^\bullet)$ und $\overline{\mathcal{H}} = (\overline{H}, t)$ mit Standardtagging t , so dass $(h, i) = \beta_t(\overline{\mathcal{H}})$, bzw. $\alpha_t(h, i) = \overline{h}$. Nach Definition von α_t gilt dann $[[x_j]]i + \text{cn}(a) \in \text{dom}(h)$

und $h(\llbracket x_j \rrbracket i + \text{cn}(a)) = l \neq \mathbf{nil}$. Nach Lemma 4.9 gilt wegen $\bar{H} = \alpha(h, i)$ und $\bar{H} \in L(X_\sigma^\bullet)$: $h, i, \eta \models \psi$.

Sei nun $h' \in \text{He}$ ein Heap und η_{emp} gemäß Definition 5.5, so dass $h', i, \eta_{\text{emp}} \models \psi'$ gilt. Wegen der variablenspezifischen Konkretisierbarkeit von φ folgt aus $h, i, \eta \models \psi$ und $h', i, \eta_{\text{emp}} \models \psi'$

$$\llbracket x_j \rrbracket i + \text{cn}(a) \in \text{dom}(h') \quad \text{und} \quad h'(\llbracket x_j \rrbracket i + \text{cn}(a)) = l \neq \mathbf{nil}. \quad (\ddagger)$$

Sei P_{emp} der Regelsatz, der jedem Nichtterminal einen kantenlosen Hypergraphen mit ausschließlich externen Knoten zuweist und λ_{emp} die zugehörige Graphumgebung. Offensichtlich korrespondiert η_{emp} mit λ_{emp} , da η_{emp} ausschließlich leere Heaps enthält.

Nach Lemma 4.9 folgt aus $h', i, \eta_{\text{emp}} \models \psi'$ nun $\alpha(h', i) \in \kappa(\lambda_{\text{emp}}, \underbrace{h(\llbracket \psi' \rrbracket)}_{=\mathcal{K}'})$. Machen wir nun

x_1, \dots, x_n extern, so erhalten wir nach (\dagger) die Aussage

$$\underbrace{\text{expose}(x_1 \dots x_n, \alpha(h', i))}_{=: \mathcal{H}_{h'}} \in \kappa(\lambda_{\text{emp}}, \mathcal{K})$$

Für $H_{h'}$, den Graphen in $\mathcal{H}_{h'}$ gilt nun, dass $\llbracket x_j \rrbracket i$ der j -te externe Knoten ist, also $\text{ext}_{H_{h'}}(j) = \llbracket x_j \rrbracket i$. Wegen der Definition von α und wegen (\ddagger) gibt es in $H_{h'}$ also eine Kante e mit $\text{lab}_{H_{h'}}(e) = a$ und $\text{att}_{H_{h'}}(e)(1) = \text{ext}_{H_{h'}}(j)$. Für $\kappa(\lambda_{\text{emp}}, K)$ gilt $\kappa(\lambda_{\text{emp}}, K) = \{\hat{H}\}$, wobei \hat{H} der Hypergraph ist, der entsteht, wenn aus K alle Nichtterminalkanten gelöscht werden.

Demnach enthält \hat{H} die Kante e (so wie oben beschrieben) ebenfalls und somit aufgrund der Verwendung von λ_{emp} als Graphumgebung auch K , womit die Behauptung gezeigt ist.

(2.)

Sei $G = \langle \exists, P \rangle \in \text{DSG}$ getypt und lokal konkretisierbar und **let** Γ **in** $\varphi = s[\llbracket G \rrbracket]$.

Wir zeigen, dass es für jedes $\sigma(x_1, \dots, x_n) = \psi$ in Γ und jedes j , $1 \leq j \leq n$ eine Menge $\Psi_j \subseteq \text{dnf}(\psi)$ gibt, so dass 1. und 2. aus Definition 5.5 gelten.

Sei $X \in N_G$ ein beliebiges Nichtterminalsymbol. Die Produktionsregeln $P^X \subseteq P$ werden von $s_P[\llbracket \cdot \rrbracket]$ in ein Prädikat $\sigma_X(x_1, \dots, x_n) = \psi_1 \vee \dots \vee \psi_m$ mit $m = |P^X|$ übersetzt, so dass für $(X \rightarrow K_k) \in P^X$, $1 \leq k \leq m$ gilt: $\psi_k = s_h[\llbracket K_k \rrbracket]$.

Als Mengen Ψ_1, \dots, Ψ_n für die n Parameter von σ_X verwenden wir die Formeln, die sich bei der Übersetzung von $P_{(X,1)}, \dots, P_{(X,n)}$ ergeben. Wegen $P_{(X,1)}, \dots, P_{(X,n)} \subseteq P^X$ gilt offenbar auch $\Psi_{(X,1)}, \dots, \Psi_{(X,n)} \subseteq \{\psi_1, \dots, \psi_m\}$.

Wir wählen nun ein beliebiges $j \in \mathbb{N}$, $1 \leq j \leq n$ und zeigen die Existenz von Ψ_j , so dass 1. und 2. aus Definition 5.5 gelten.

1. – „ \Rightarrow “: Seien $h \in \text{He}$, $i \in \text{Int}$ und es gelte

$$h, i, \tilde{\eta}_j \models \sigma_X(x_1, \dots, x_n)$$

Die Formel $\sigma_X(x_1, \dots, x_n)$ wird von $s[\llbracket \cdot \rrbracket]$ ausschließlich bei der Übersetzung einer Nichtterminalkante mit dem Label X durch $s_e[\llbracket \cdot \rrbracket]$ erzeugt. Um die sie aus einem Hypergraphen zu erzeugen, darf dieser ausschließlich über externe Knoten v_1, \dots, v_n verfügen (nach Definition des Standardtaggings sind die Variablen x_k , $k \in \mathbb{N}$ für externe Knoten reserviert). Der gesuchte Graph ist genau X^\bullet : $s_h[\llbracket X^\bullet \rrbracket] = \sigma_X(x_1, \dots, x_n)$

Kapitel 5. HAG-Eigenschaften für SL-Formeln

Es gilt also

$$h, i, \tilde{\eta}_j \models s_h \llbracket \langle X^\bullet \rangle \rrbracket$$

Nach Lemma 4.18 ist dies äquivalent zu

$$\overbrace{\text{expose}(\alpha_t(h, i), \text{ext}_{\langle X^\bullet \rangle})}^{\mathcal{H} :=} \in \kappa(\lambda_{P_{(X,j)} \cup \overline{P^X}}, \langle X^\bullet \rangle)$$

$= x_1 \dots x_n$

Wegen

$$L_{P_{(X,j)} \cup \overline{P^X}}(X^\bullet) = \kappa \left(\lambda_{P_{(X,j)} \cup \overline{P^X}}, \langle X^\bullet \rangle \right)$$

gilt nun wiederum $\mathcal{H} \in L_{P_{(X,j)} \cup \overline{P^X}}(X^\bullet)$ und aufgrund der lokalen Konkretisierbarkeit von G

$$\mathcal{H} \in L_P(X^\bullet)$$

Nach erneuter Anwendung von Lemma 4.18 unter der Verwendung der zu P gehörenden Graphumgebung λ_P und der damit korrespondierenden Prädikatumgebung η ergibt sich

$$h, i, \eta \models s_h \llbracket \langle X^\bullet \rangle \rrbracket$$

und schließlich

$$h, i, \eta \models \sigma_X(x_1, \dots, x_n).$$

„ \Leftarrow “: Da P und $(P_{(X,h)} \cup \overline{P^X})$ in „ \Rightarrow “ beliebig vertauschbar sind, gilt die Gegenrichtung analog.

2. – Seien $1 \leq j \leq n$, $h, h' \in \text{He}$ und $i \in \text{Int}$. Sei ferner $\psi' \in \Psi_j$ beliebig gewählt, so dass gilt

$$h, i, \eta \models \psi \quad \wedge \quad h', i, \eta_{\text{emp}} \models \psi'.$$

Aufgrund der Konstruktion von Ψ_j gibt es $(X \rightarrow K) \in P_{(X,j)}$, so dass $\psi' = s_h \llbracket \mathcal{K} \rrbracket$. (†)

Nach Lemma 4.18 gilt

$$\underbrace{\text{expose}(\alpha_t(h', i), \text{ext}_{\mathcal{K}})}_{=: \mathcal{H}} \in \kappa(\lambda_{\text{emp}}, \mathcal{K})$$

Es folgt $\mathcal{H} \in L_{P_{\text{emp}}}(\mathcal{K})$. \mathcal{H} ist also genau der Hypergraph, der durch Löschen aller Nicht-terminalkanten aus \mathcal{K} entsteht. Für ψ gilt $\psi = \bigvee \{s \llbracket \tilde{\mathcal{K}} \rrbracket \mid (X \rightarrow \tilde{K}) \in P^X\}$. Die Beziehung $h, i, \eta \models \psi$ gilt deshalb genau dann, wenn ein $\tilde{\psi} \in \{s \llbracket \tilde{\mathcal{K}} \rrbracket \mid (X \rightarrow \tilde{K}) \in P^X\}$ existiert, so dass $h, i, \eta \models \tilde{\psi}$ erfüllt ist.

Nach Annahme gibt es ein derartiges $\tilde{\psi}$ und eine dazugehörige Regel $(X \rightarrow \tilde{K}) \in P^X$. Wir betrachten nun eine beliebige Heapadresse $l \in \text{dom}(h) \cap L_j$. Wegen $h, i, \eta \models \tilde{\psi}$ gibt es ein $s \in \text{sel}_\Sigma$, so dass $l = \llbracket x_j \rrbracket i + \text{cn}(s)$. x_j repräsentiert in \tilde{K} den externen Knoten $\text{ext}_{\tilde{K}}(j)$. Folglich gilt wegen der Getyptheit $s \in \text{type}(X, j)$. Aufgrund der lokalen Konkretisierbarkeit von G folgt nun (nach Definition 2.21, (2.)), dass es für alle Regeln $(X \rightarrow \hat{K}) \in P_{(X,j)}$ eine Kante $e \in E_{\hat{K}}$ mit $\text{lab}_{\hat{K}}(e) = s$ und $\text{att}_{\hat{K}}(e)(1) = \text{ext}_{\hat{K}}(j)$ gibt, also auch für die oben (†) verwendete Regel $(X \rightarrow K)$.

Da e keine Nichtterminalkante ist, gibt es sie auch in \mathcal{H} . Wegen $\mathcal{H} = \text{expose}(\alpha_t(h', i), \text{ext}_K)$ folgt

$$l \in \text{dom}(h') \cap L_j \quad \text{und} \quad h'(l) \neq \mathbf{nil}.$$

Wegen der willkürlichen Wahl von l aus $\text{dom}(h) \cap L_j$ gilt die Behauptung.

Dies schließt den Beweis des Lemmas ab. \square

5.5. HAG-ÄQUIVALENTE FORMELN

Nachdem das DSG-Kriterium und die vier HAG-Kriterien definiert wurden, lässt sich auf deren Basis die HAG-Zugehörigkeit garantieren.

SATZ 5.5

1. Sei $\varphi \in \text{SLF}$ eine flache Formel, die *getypt*, *wachsend* und *variablenspezifisch konkretisierbar* ist, ausschließlich *auswertbare* Prädikate enthält und außerdem das DSG-Kriterium nach Definition 4.11 erfüllt. Dann ist $g[\![\varphi]\!]$ eine HAG.
2. Sei $G \in \text{HAG}_{\Sigma_N}$. Dann ist $s[\![G]\!]$ *getypt*, *wachsend*, *variablenspezifisch konkretisierbar* und jedes darin definierte Prädikat *auswertbar*.

Beweis. **(1.)** Nach Satz 4.3, und den Lemmata 5.1, 5.2, 5.3 und 5.4 folgt, dass $g[\![\varphi]\!]$ eine produktive, getypte, wachsende und lokal konkretisierbare DSG und damit nach Definition 2.22 eine HAG ist.

(2.) Folgt aus denselben Lemmata. \square

KAPITEL 6

BEWEISREGELN FÜR DIE SL-VERIFIKATION

Heapabstraktionsgrammatiken und Separation Logic wurden unter anderem zu dem Zweck entwickelt, Software zu verifizieren, die den Heap, beziehungsweise dynamische Datenstrukturen verwendet. Der Grundgedanke zur Behandlung des unendlichen Zustandsraumes, der von diesen Datenstrukturen definiert wird, ist in beiden Fällen ähnlich: Der Heap wird stets abstrakt betrachtet und lediglich zur Anwendung von Speicherabfragen oder -manipulationen lokal konkretisiert. Für die Verifikation vieler Aussagen spielt es dabei keine Rolle, an welcher genauen Stelle innerhalb einer Datenstruktur eine Operation gerade durchgeführt wird. Beim Invertieren einer Liste mittels einer Schleife beispielsweise werden alle Elemente der Liste gleich behandelt. Zur allgemengültigen Verifikation des Schleifenrumpfes ist es daher nicht von Bedeutung, welches Element im Einzelnen betrachtet wird.

Die Separation Logic verifiziert Programm(-Fragmente) axiomatisch, also unter Verwendung einer Vor- und einer Nachbedingung, analog zur Hoare-Logik (siehe Kapitel 3). Dazu wird ein Beweisbaum abgeleitet, der die zu verifizierende Aussage mit Hilfe von gültigen Beweis- oder Ableitungsregeln auf ein oder mehrere Axiome zurückführt. Diese Regeln werden zuvor separat für den Allgemeinfall bewiesen.

In HRGs werden Datenstrukturen durch Produktionsregeln definiert, in der SL hingegen durch Prädikate. Während in den Grammatiken die Produktionsregeln die nötigen Werkzeuge für die Abstraktion und Konkretisierung repräsentieren, geschieht dies in der SL über spezielle Beweisregeln, die sogenannten *Rearrangement*- und *Abstraction*-Regeln.

Abschnitt 6.1 beschreibt die Anforderungen an Prädikate zur Erzeugung aller benötigten Beweisregeln. Im Anschluss wird in Abschnitt 6.2 eine Methode beschrieben, um diese Beweisregeln aus HAGs zu erzeugen. Schließlich wird in Abschnitt 6.3 beispielhaft die Verifikation eines einfachen Programms durchgeführt.

6.1. REARRANGEMENT- UND ABSTRAKTIONSREGELN FÜR PRÄDIKATE

Zur Verifikation von Programmen mittels Separation Logic wird ein Beweisbaum erzeugt, der die Gültigkeit eines SL-Tripels (Analog zum Hoare-Tripel: Vorbedingung, Anweisung(en), Nachbedingung) durch die Rückführung auf Axiome zeigt [Rey08]. Durch die Verwendung eines vorgegebenen Satzes gültiger Regeln lässt sich die Ableitung solcher Beweisbäume zumindest teilweise automatisieren. Eine Ableitungsregel hat die Form

$$\frac{P_1 \quad \dots \quad P_n}{C}$$

Kapitel 6. Beweisregeln für die SL-Verifikation

wobei P_1, \dots, P_n Prämissen und C Konklusion genannt werden. Die logische Aussage einer solchen Beweisregel ist

Wenn P_1 und \dots und P_n korrekt sind, dann gilt auch C .

Beweisbäume werden daher von unten nach oben konstruiert, so dass alle Zweige schließlich in allgemeingültigen Axiomen enden. Diese notieren wir als prämissenlose Regel:

$$\frac{}{\text{(Axiom)}}$$

Prämissen und Konklusionen können einerseits aus Tripeln und andererseits aus sogenannten Entailments bestehen. Ein Entailment, notiert durch $A \vdash B$ ist zunächst eine Implikation. Im Gegensatz zu $A \Rightarrow B$, das eine abkürzende Schreibweise für $\neg A \vee B$ darstellt, ist ein Entailment jedoch keine einfache Assertion sondern Teil der Beweisstruktur. $\varphi \vdash \psi$ ist gültig, wenn für alle $h \in \text{He}$, $i \in \text{Int}$ mit $h, i \models \varphi$ auch $h, i \models \psi$ gilt.

Stellen wir die zu verifizierende Spezifikation einer Programmanweisung c , die auf einer Datenstruktur operiert, als SL-Tripel dar, dann treffen die Vor- und Nachbedingungen üblicherweise Aussagen über diese Datenstruktur. Dies geschieht über die Verwendung des entsprechenden Prädikats:

$$\{\sigma(\dots) * \dots\} c \{\sigma(\dots) * \dots\}$$

Um diese Spezifikation mit Hilfe von Beweisregeln auf Axiome zurückführen zu können, ist in den meisten Fällen eine Konkretisierung der abstrakten Darstellung $\sigma(\dots)$ nötig, da sich Axiome auf konkrete, also atomare Assertionen (Definition 3.7) beziehen. Dies gilt sowohl für die Vor- als auch für die Nachbedingung. Die Konkretisierung und Abstraktion findet in Beweisbäumen also eher in „vertikaler“ als in „horizontaler“ Richtung statt.

Die Literatur stellt oft die Begriffe *Rearrangement* für konkretisierende und *Abstraction* für abstrahierende Beweisregeln gegenüber [DOY06]. Erstere werden häufig auch anschaulich als *Unroll*- und letztere als *Collapse*-Regeln bezeichnet. Aufgrund der Beweisstruktur sind die Vorgänge des Abstrahierens und Konkretisierens im Gegensatz zum HRG-Ansatz nicht immer klar trennbar. Eine genaue Definition beziehungsweise einheitliche Abgrenzung der Begriffe gegeneinander existiert daher in der gegenwärtigen Literatur noch nicht. So findet man in vielen Veröffentlichungen je nach thematischem Fokus auch Beweisregeln mit Namen wie UNROLLCOLLAPSE [BCO04].

In dieser Arbeit verwenden wir den Begriff *Rearrangement-Regel* für eine Regel, die ein Tripel mit abstrakter Vorbedingung unter der Prämisse verifiziert, dass es für alle konkretisierten Ausprägungen der Vorbedingung ebenfalls gilt. Den Begriff *Abstraktionsregel* verwenden wir, um aus einem geltenden Tripel mit konkreter Nachbedingung das entsprechende Tripel mit abstrakter Nachbedingung zu folgern.

Rearrangement:

$$\frac{\{\text{konkr. Vorb. } 1\} c \{\text{Nachb.}\} \quad \dots \quad \{\text{konkr. Vorb. } n\} c \{\text{Nachb.}\}}{\{\text{abstr. Vorb.}\} c \{\text{Nachb.}\}}$$

6.1. Rearrangement- und Abstraktionsregeln für Prädikate

Abstraktion:

$$\frac{\{\text{Vorb.}\} c \{\text{konkr. Nachb.}\}}{\{\text{Vorb.}\} c \{\text{abstr. Nachb.}\}}$$

Die Beweisregeln zu einer Datenstruktur lassen sich aus der disjunktiven Normalform der zugehörigen Prädikatdefinition erzeugen. Es ist jedoch nicht jede Prädikatdefinition zur Abstraktion und Konkretisierung geeignet, ebenso, wie nicht jede HRG zur Heapabstraktion verwendet werden kann. Beispielsweise müssen die vier in Kapitel 5 hergeleiteten Eigenschaften *Auswertbarkeit*, *Wachstum*, *Getyptheit* und *variablenspezifische Konkretisierbarkeit* für die Herleitung allgemeiner Beweisregeln gewährleistet sein. Andernfalls können unerwünschte Effekte, wie beispielsweise unendliche Beweisbäume auftreten.

Heapabstraktionsgrammatiken bieten im Vergleich zu SL-Prädikaten einige Vorteile. Die Modellierung, auch von komplexeren Datenstrukturen, ist verhältnismäßig systematisch und intuitiv, während die entsprechenden Prädikate schnell unübersichtlich werden. Durch die berechenbare Greibach-Normalform für Hyperkanteneretzungsgrammatiken lässt sich automatisiert eine lokal konkretisierbare Grammatik erzeugen [JHKN11], während sich der Nachweis der variablenspezifischen Konkretisierbarkeit von SLF-Prädikaten aufgrund ihrer Semantik-basierten Definition als relativ schwierig herausstellen dürfte. Diese ist jedoch zwingend erforderlich, wenn das betrachtete Programm sämtliche Zeiger der Datenstruktur dereferenzieren dürfen soll. Der Grund liegt darin, dass die *Lookup*-Axiome der SL konkrete Zeigerassertionen als Vor- und Nachbedingung benötigen [Rey08]. Diese müssen entsprechend von den Beweisregeln für jeden Zeiger, den das Prädikat definiert, abgeleitet werden können. Für eine Grammatik $G \in \text{HAG}_{\Sigma_N}$ gilt nach Satz 5.5, dass $\varphi = s[G]$ wachsend, getypt, variablenspezifisch konkretisierbar und jedes Prädikat darin auswertbar ist. Per Konstruktion von $s_h[\cdot]$, $s_H[\cdot]$ und $s_P[\cdot]$ sind außerdem alle Prädikatdefinitionen in φ in disjunktiver Normalform. Die folgende Definition gibt die Rearrangement-Regel für beliebige Prädikate an.

DEFINITION 6.1 ((REARR σ)) Sei $\sigma = (x_1, \dots, x_n) = \psi$ eine Prädikatdefinition und $\text{dnf}(\psi) = \{\psi_1, \dots, \psi_m\}$. Die Rearrangement-Regel für σ lautet.

$$\frac{\{\psi_1 * \varphi\} c \{\varphi'\} \quad \dots \quad \{\psi_m * \varphi\} c \{\varphi'\}}{\{\sigma(y_1, \dots, y_n) * \varphi\} c \{\varphi'\}} \text{ (REARR}\sigma\text{)}$$

LEMMA 6.1 (REARR σ) ist korrekt.

Beweis.

$$\frac{\frac{\{\psi_1 * \varphi\} c \{\varphi'\} \quad \dots \quad \{\psi_m * \varphi\} c \{\varphi'\}}{\{(\psi_1 \vee \dots \vee \psi_m) * \varphi\} c \{\varphi'\}} \text{ (DISJ)}}{\{\sigma(y_1, \dots, y_n) * \varphi\} c \{\varphi'\}} \text{ (SP)}$$

(DISJ) und (SP) – „Disjunction“ und „Strengthening Precedent“ – gemäß [Rey08]. \square

Die Abstraktionsregeln für σ sind entsprechend definiert.

Kapitel 6. Beweisregeln für die SL-Verifikation

DEFINITION 6.2 ((ABSTR σ)) Sei $\sigma = (x_1, \dots, x_n) = \psi$ eine Prädikatdefinition und $\text{dnf}(\psi) = \{\psi_1, \dots, \psi_m\}$. Jede der folgenden Beweisregeln bezeichnen wir mit (ABSTR σ).

$$\frac{\{\varphi\} c \{\psi_1 * \varphi'\}}{\{\varphi\} c \{\sigma(y_1, \dots, y_n) * \varphi'\}} \text{ (ABSTR}\sigma\text{)}$$

$$\vdots$$

$$\frac{\{\varphi\} c \{\psi_m * \varphi'\}}{\{\varphi\} c \{\sigma(y_1, \dots, y_n) * \varphi'\}} \text{ (ABSTR}\sigma\text{)}$$

LEMMA 6.2 Alle Regeln (ABSTR σ) sind korrekt.

Beweis. Sei $1 \leq j \leq m$

$$\frac{\{\varphi\} c \{\psi_j * \varphi'\}}{\{\varphi\} c \{(\psi_1 \vee \dots \vee \psi_m) * \varphi'\}} \text{ (WC)}$$

$$\frac{\{\varphi\} c \{(\psi_1 \vee \dots \vee \psi_m) * \varphi'\}}{\{\varphi\} c \{\sigma(y_1, \dots, y_n) * \varphi'\}} \text{ (WC)}$$

(WC) – „Weakening Consequent“ – gemäß [Rey08]. □

Während die Abstraktionsregeln relativ einfach nachzuvollziehen sind (aus ψ_j folgt offenbar jede Formel $\psi_j \vee \psi'$) erscheint die Forderung der Regel (REARR σ), dass das Tripel für alle Disjunkte gelten muss, zunächst kontraintuitiv.

Der Grund, warum dies notwendig ist, liegt in der Natur von Vorbedingungen an sich. Ist eine Vorbedingung nicht erfüllt, so ist das dazugehörige Tripel automatisch gültig. Würde man nur eine einzelne Prämisse $\{\psi_j * \varphi\} c \{\varphi'\}$ voraussetzen, so wäre die Nachbedingung φ' nur dann gefordert, wenn $\psi_j * \varphi$ wahr ist. Die Regel soll φ' aber als Nachbedingung für *alle* möglichen Konkretisierungen von σ sicherstellen. Anders ausgedrückt, dürfen Vorbedingungen in Beweisrichtung (von oben nach unten) ausschließlich verstärkt werden oder äquivalent bleiben. Das folgende Beispiel soll dies anhand einfacher Hoare-Tripeln ohne Heap-Bezug verdeutlichen.

Beispiel 6.1. Sei $\sigma(x) = (x = 2) \vee (x = 4) \vee (x = 6)$. Offenbar sind die Entailments $(y = 2) \vdash \sigma(y)$, $(y = 2) \wedge (z = 4) \vdash \sigma(y)$ und sogar $(y = 2) \wedge (y = 4) \vdash \sigma(y)$ korrekt. Das letzte Entailment gilt, weil seine Vorbedingung unerfüllbar ist. Die Regel

$$\frac{\{y = 2\} y := y + 1 \{y \leq 5\}}{\{\sigma(y)\} y := y + 1 \{y \leq 5\}}$$

ist aber offenbar falsch, da die Fälle, in denen y nicht den Wert 2 hat, nicht berücksichtigt werden. Verwenden wir stattdessen die Regel

$$\frac{\{y = 2\} y := y + 1 \{y \leq 5\} \quad \{y = 4\} y := y + 1 \{y \leq 5\} \quad \{y = 6\} y := y + 1 \{y \leq 5\}}{\{\sigma(y)\} y := y + 1 \{y \leq 5\}}$$

so ist diese korrekt. Die dritte Prämisse ist offenbar ungültig, da sie fordert „Wenn y den Wert 6 hat und um 1 erhöht wird, muss es danach kleiner oder gleich 5 sein“. Dies macht die gesamte Regel jedoch gültig, wenn auch wertlos. Es gibt selbstverständlich keine gültige Regel, mit der $\{\sigma(y)\} y := y + 1 \{y \leq 5\}$ bewiesen werden kann, da dieses Tripel für die gegebene Definition von σ offensichtlich falsch ist. Δ

6.2. ERZEUGUNG VON BEWEISREGELN AUS HAGS

Damit eine DSG zur korrekten Abstraktion und Konkretisierung verwendet werden kann, muss sie die vier HAG-Eigenschaften erfüllen und darüber hinaus rückwärtig konfluent sein [HJKN12]. Entsprechendes gilt auch für Prädikate, die zur Verifikation von Programmen eingesetzt werden sollen. In Kapitel 5 wurden analoge Eigenschaften für Formeln definiert und die Erhaltung der Eigenschaften bei der Formelgenerierung und Grammatiksynthese gezeigt. Aus denselben beziehungsweise analogen Gründen, die Nicht-HAGs für die Abstraktion und Konkretisierung allgemein unbrauchbar machen, müssen auch Formeln diese Eigenschaften erfüllen. Da die in die Formel eingebettete Definition der Prädikate im Sinne des **let...in...**-Konstruktes und insbesondere der **in**-Teil der Formel für die Erzeugung von Beweisregeln nicht von Belang sind, sprechen wir in diesem Zusammenhang meistens direkt über die generierten Prädikate und ihre Definitionen. Die syntaktische Form der Formel und die Semantik des **in**-Teiles spielen dabei keine Rolle (ebenso wie die Startgraphen in HAGs).

Getyptheit – Die Getyptheit wird bereits für die Generierung einer Formel aus einer Grammatik vorausgesetzt (vgl. Abschnitt 4.4), um explizite Nullzeigerassertionen zu erstellen. Dadurch, dass jedes Feld eines Objektes auf ein anderes Objekt oder **nil** verweist, ist eine implizite Typisierung gegeben. Ohne diese könnten valide Beweisbäume für eigentlich unerfüllbare Nachbedingungen hergeleitet werden. Wird ein Prädikat manuell definiert, also ohne zugehörige Grammatik, und dabei die Verwendung expliziter Nullzeiger konsequent angewandt, ist dessen Getyptheit im Sinne von Abschnitt 5.2 nicht erforderlich.

Auswertbarkeit – Nicht auswertbare Prädikate definieren – wie auch unproduktive Nichtterminale – keine sinnvollen Datenstrukturen. Wären sie in Vor- oder Nachbedingungen zugelassen, könnte dies – je nach Algorithmus des Beweisers – zu unendlichen (sich wiederholenden) Beweisbäumen führen.

Beispiel 6.2. Das folgende Prädikat ist nicht auswertbar.

$$\sigma(x_1) = \exists r : r.s \mapsto x_1 * \sigma(x_1)$$

Es beschreibt einen Heap auf dem unendlich viele Objekte existieren, deren Selektor s jeweils auf den Parameter x_1 zeigt. Dies ist offensichtlich keine realisierbare Datenstruktur. Δ

Wachstum – Diese Eigenschaft wird vor allem für die Anwendung der Abstraktionsregeln benötigt. Ist sie verletzt, so führt dies ebenfalls zu unendlichen Beweisbäumen, da bei Anwendung einer (ABSTR σ)-Regel zu einem Disjunkt ψ_j , das die Wachstumseigenschaft nicht erfüllt, keine echte Veränderung der Nachbedingung stattfindet.

Beispiel 6.3. Der folgende Beweisbaum zeigt die unendliche Abstraktionsfolge für die nicht-wachsenden Prädikate $\sigma_1(x_1) = \sigma_2(x_1)$ und $\sigma_2(x_1) = \sigma_1(x_1)$:

$$\begin{array}{c} \vdots \\ \frac{\{\varphi\} c \{\sigma_2(y)\}}{\{\varphi\} c \{\sigma_1(y)\}} \text{ (ABSTR}\sigma_1\text{)} \\ \frac{\{\varphi\} c \{\sigma_1(y)\}}{\{\varphi\} c \{\sigma_2(y)\}} \text{ (ABSTR}\sigma_2\text{)} \\ \frac{\{\varphi\} c \{\sigma_2(y)\}}{\{\varphi\} c \{\sigma_1(y)\}} \text{ (ABSTR}\sigma_1\text{)} \end{array}$$

△

Variablenspezifische Konkretisierbarkeit – Wenn das Programm sämtliche Zeiger innerhalb der Datenstruktur dereferenzieren darf – und üblicherweise darf es das – muss sichergestellt sein, dass jede Zeigerassertion, die durch ein Prädikat definiert wird, in konstant vielen Schritten in der Vorbedingung konkretisiert und aus der Nachbedingung (von oben nach unten) abstrahiert werden kann. Andernfalls ist die Ableitung eines entsprechenden Beweisbaumes nicht möglich. Die variablenspezifische Konkretisierbarkeit garantiert dies für alle Parametervariablen beziehungsweise die übergebenen Argumente und damit prinzipiell für alle Variablen, die in Vor- oder Nachbedingung auftreten.

Beispiel 6.4. Das folgende Prädikat ist nicht variablenspezifisch konkretisierbar:

$$\sigma(x_1, x_2) = x_1.n \mapsto x_2 \vee (\exists r : \sigma(x_1, r) * r.n \mapsto x_2)$$

Es repräsentiert eine einfach-verkettete Liste, die über den n -Selektor durchlaufen werden kann. Die Konkretisierung der Liste ist jedoch nur am Ende möglich, der $x_1.n$ -Selektor kann im zweiten Disjunkt nicht in konstant vielen Schritten konkretisiert werden. △

Die variablenspezifische Konkretisierbarkeit nach Definition 5.5 fordert sogar die Erzeugbarkeit jedes aus einem Parameter ausgehenden Selektors in genau einem Schritt.

Rückwärtige Konfluenz – Diese Eigenschaft garantiert, dass sämtliche auf einem Graphen ausgeführten Folgen von Abstraktionsschritten irgendwann in derselben, nicht weiter abstrahierbaren, Darstellung enden. Anders ausgedrückt bedeutet das, dass sich alle konkreten oder teilweise abstrahierten Graphen stets weiter abstrahieren lassen, bis diese Darstellung erreicht ist (siehe Definition 2.14). Analoges gilt auch für Prädikate, die aus einer rückwärtig konfluenten Grammatik erzeugt wurden.

Beispiel 6.5. Wir betrachten die folgende Prädikatdefinition:

$$\sigma(x_1, x_2) = x_1.n \mapsto x_2 \vee (\exists r : x_1.n \mapsto r * \sigma(r, x_2))$$

Darin ist σ nicht rückwärtig konfluent. So gibt es keine Möglichkeit, die Formel

$$\varphi = \exists r : \sigma(x, r) * r.n \mapsto y$$

in die kompakte Darstellung $\varphi' = \sigma(x, y)$ zu abstrahieren, obwohl $\varphi \vdash \varphi'$ gilt. △

6.2. Erzeugung von Beweisregeln aus HAGs

Wird eine Grammatik, die sämtliche dieser Eigenschaften erfüllt, also eine rückwärtig konfluente HAG ist, in Prädikatdefinitionen übersetzt, dann sind die daraus generierten Beweisregeln zwar für die Abstraktion geeignet, für das Rearrangement sind sie jedoch häufig zu allgemein, wie das folgende Beispiel zeigt.

Beispiel 6.6. Wir erweitern die Prädikatdefinition aus Beispiel 6.5, so dass es rückwärtig konfluent wird:

$$\sigma(x_1, x_2) = x_1.n \mapsto x_2 \vee (\exists r : x_1.n \mapsto r * \sigma(r, x_2)) \vee (\exists t : \sigma(x_1, t) * \sigma(t, x_2))$$

Darüber hinaus ist σ getypt und variablenspezifisch konkretisierbar. Die Formel

$$\exists r : \sigma(x, r) * r.n \mapsto y$$

kann jetzt mit Hilfe des hinzugefügten, dritten Disjunktives in zwei Schritten zu $\sigma(x, y)$ abstrahiert werden:

$$\frac{\frac{\{\varphi\} c \{\exists r : \sigma(x, r) * r.n \mapsto y\}}{\{\varphi\} c \{\sigma(x, r) * \sigma(r, y)\}} \text{ (ABSTR}\sigma\text{)}}{\{\varphi\} c \{\sigma(x, y)\}} \text{ (ABSTR}\sigma\text{)}$$

Für das Rearrangement kann σ jedoch nicht verwendet werden. Die (REARR-)Regel für σ hat gemäß Definition 6.1 die Gestalt (die Prämissen wurden aus Platzgründen übereinander notiert):

$$\frac{\frac{\frac{\{y_1.n \mapsto y_2 * \varphi\} c \{\varphi'\}}{\{\exists r : y_1.n \mapsto r * \sigma(r, y_2)\} c \{\varphi'\}}}{\{\exists t : \sigma(y_1, t) * \sigma(t, y_2) * \varphi\} c \{\varphi'\}}}{\{\sigma(y_1, y_2) * \varphi\} c \{\varphi'\}} \text{ (REARR}\sigma\text{)}$$

Die dritte, neu hinzugekommene Prämisse, konkretisiert offensichtlich keine Zeigerassertion. Der Beweisbaum zu einer Spezifikation, deren Vorbedingung σ enthält, wäre stets unendlich, da im Teilbaum über der dritten Prämisse ein erneuter Rearrangement-Schritt nötig ist, bei dem unter anderem wieder die dritte Prämisse mit ihren zwei σ -Aufrufen erzeugt wird. Auf diese Weise entsteht im Baum ein unendlicher Pfad. Δ

Zusammenfassend lässt sich feststellen, dass rückwärtige Konfluenz in vielen Fällen für die Abstraktion erforderlich ist. Auf der anderen Seite können bestimmte Regeln einer Grammatik beziehungsweise Disjunkte eines Prädikats das Rearrangement stören und unendliche Beweisbäume hervorrufen, obwohl sie die Aussage des Prädikats an sich nicht verändern. Die Lösung für dieses Problem ist die Verwendung verschiedener, in Bezug auf die Prädikate äquivalenter, Teilgrammatiken zur Erzeugung der Rearrangement- und der Abstraktionsregeln. Diese Äquivalenzeigenschaft muss jedoch zunächst formal definiert werden. Die Sprache von Grammatiken \mathcal{L} beziehungsweise die Äquivalenz von Formeln \equiv ist hierfür ungeeignet, schon deshalb, weil HAGs keine Startgraphen und die daraus generierten Formeln daher einen unerfüllbaren **in**-Teil haben. Die *Produktionsäquivalenz* beschreibt die gesuchte Eigenschaft für zwei Grammatiken.

DEFINITION 6.3 (Produktionsäquivalenz) Seien $G_1 = \langle \exists_1, P_1 \rangle$, $G_2 = \langle \exists_2, P_2 \rangle \in \text{DSG}_{\Sigma_N}$. G_1 und G_2 sind *produktionsäquivalent* ($G_1 \sim G_2$), wenn $N_{G_1} = N_{G_2}$ und für alle $X \in N_{G_1}$ gilt:

$$L_{P_1}(X^\bullet) = L_{P_2}(X^\bullet)$$

Die analoge Eigenschaft für Formeln heißt entsprechend *Prädikatäquivalenz* und wird über Prädikatinterpretationen definiert.

DEFINITION 6.4 (η_Γ) Sei $\varphi = \mathbf{let} \Gamma \mathbf{in} \varphi'$. $\eta_\Gamma \in \text{PI}$ ist die Prädikatinterpretation mit der kleinsten Definitionsmenge, so dass für beliebige $h \in \text{He}$ und $i \in \text{Int}$ gilt

$$h, i, \eta_\Gamma \models \varphi' \iff h, i, \eta_0 \models \varphi$$

η_Γ , ist also genau die Prädikatinterpretation, die durch Γ definiert wird. Dadurch enthält es sämtliche semantischen Informationen über die in Γ definierten Prädikate.

DEFINITION 6.5 (Prädikatäquivalenz) Seien $\varphi_1 = \mathbf{let} \Gamma_1 \mathbf{in} \varphi'_1$ und $\varphi_2 = \mathbf{let} \Gamma_2 \mathbf{in} \varphi'_2$.

φ_1 und φ_2 sind *prädikatäquivalent* ($\varphi_1 \sim \varphi_2$), wenn $\eta_{\Gamma_1} = \eta_{\Gamma_2}$.

Werden nur die Prädikatsequenzen betrachtet, schreiben wir auch $\Gamma_1 \sim \Gamma_2 \iff \varphi_1 \sim \varphi_2$.

Die Prädikatäquivalenz $\varphi_1 \sim \varphi_2$ ist genau dann gegeben, wenn für alle darin definierten Prädikate σ mit

$$\begin{aligned} \varphi_1 &= \mathbf{let} \dots \sigma(\dots) = \psi_1 \dots \mathbf{in} \dots \\ \varphi_2 &= \mathbf{let} \dots \sigma(\dots) = \psi_2 \dots \mathbf{in} \dots \end{aligned}$$

gilt, dass $\psi_1 \equiv \psi_2$. Letzteres ist genau dann der Fall, wenn $h, i, \eta \models \psi_1 \iff h, i, \eta \models \psi_2$, was dazu führt, dass für die entsprechenden h, i die Tupel ($\llbracket x_1 \rrbracket i \dots \llbracket x_{\text{rk}(\sigma)} \rrbracket i, h$) in beide Prädikatinterpretationen, η_1 und η_2 aufgenommen werden. Es folgt $\eta_1 = \eta_2$ und damit $\varphi_1 \sim \varphi_2$.

Der Zusammenhang zwischen Produktions- und Prädikatäquivalenz wird vom folgenden Lemma erfasst.

LEMMA 6.3 (Erhaltung der Produktionsäquivalenz) Seien $G_1 = \langle \exists_1, P_1 \rangle$, $G_2 = \langle \exists_2, P_2 \rangle \in \text{DSG}_{\Sigma_N}$. Es gilt

$$s_{\mathcal{P}} \llbracket P_1 \rrbracket \sim s_{\mathcal{P}} \llbracket P_2 \rrbracket \text{ genau dann, wenn } G_1 \sim G_2$$

Beweis. Sei $s_{\mathcal{P}} \llbracket P_1 \rrbracket \sim s_{\mathcal{P}} \llbracket P_2 \rrbracket$ gegeben. Wir definieren $\eta_1 := \eta_{s_{\mathcal{P}} \llbracket P_1 \rrbracket}$ und $\eta_2 := \eta_{s_{\mathcal{P}} \llbracket P_2 \rrbracket}$. Dann gilt nach Definition 6.5

$$\eta_1 = \eta_2$$

Dies ist gemäß Definition 3.13 genau dann der Fall, wenn für alle $h \in \text{He}$, $i \in \text{Int}$, $\sigma \in \text{dom}(\eta_1)$:

$$h, i, \eta_1 \models \sigma(x_1, \dots, x_{\text{rk}(\sigma)}) \iff h, i, \eta_2 \models \sigma(x_1, \dots, x_{\text{rk}(\sigma)})$$

Aufgrund der Korrektheit von $s_{\mathcal{P}} \llbracket \cdot \rrbracket$ gilt dies wiederum genau dann, wenn für alle $h \in \text{He}$, $i \in \text{Int}$, $\sigma \in \text{dom}(\eta_1)$:

$$\begin{aligned} & \text{expose}(x_1, \dots, x_{\text{rk}(\sigma)}, \alpha_t(h, i)) \in L_{P_1}(X_\sigma^\bullet) \\ \iff & \text{expose}(x_1, \dots, x_{\text{rk}(\sigma)}, \alpha_t(h, i)) \in L_{P_2}(X_\sigma^\bullet) \end{aligned}$$

[Die Parameter werden in $\alpha_t(h, i)$ exponiert, also die dazugehörigen Knoten zu externen Knoten gemacht. Es wird α_t anstelle von α verwendet, weil die Funktion expose Taggings anstelle von Variablenkanten erwartet. Außerdem enthalten Handles keine Variablenkanten.] Dies ist aufgrund der Definitionen von $s_P[\cdot]$ und der konkreten Hülle L_P genau dann gegeben, wenn $N_{G_1} = N_{G_2}$ und für alle $X \in N_{G_1}$

$$L_{P_1}(X^\bullet) = L_{P_2}(X^\bullet)$$

gilt, was nach Definition 6.3 genau $G_1 \sim G_2$ entspricht.

Die Rückrichtung gilt analog. □

Nach Aussage des Lemmas ist es möglich, zwei produktionsäquivalente Grammatiken in zwei Prädikate zu übersetzen, die nachweislich dieselbe Datenstruktur repräsentieren. Die für die Herleitung von Rearrangement- und Abstraktionsregeln verwendeten Prädikate müssen selbstverständlich prädikatäquivalent sein. Nach Lemma 6.3 ist dies jedoch der Fall, wenn sie aus zwei produktionsäquivalenten Grammatiken generiert wurden.

Um möglichst schlanke, aber dennoch korrekte Rearrangement-Regeln zu erhalten, generieren wir ein eigenes, spezialisiertes Prädikat für jeden zu konkretisierenden Parameter, das zur „vollständigen“ Prädikatdefinition äquivalent ist. Auf der einen Seite ist dadurch die Anwendbarkeit der Rearrangement-Regeln sichergestellt, auf der anderen Seite beschleunigt es auch den Beweis, da die resultierenden Beweisbäume nicht nur endlich sondern auch weniger stark verzweigt sind. Um diese Prädikatdefinitionen zu erhalten, kann die Greibach-Normalform (GNF) für HAGs verwendet werden [JHKN11]. Diese liefert aus einer getypten Grammatik $G \in \text{DSG}_{\Sigma_N}$ für jeden Nicht-Reduktionstentakel (X, j) genau die Teilregelsätze $P_{(X,j)}$, die X einerseits vollwertig repräsentieren (also für X dieselbe konkrete Hülle liefern) und andererseits den Selektor j erzeugen (siehe Abschnitt 2.3.3). Aus der Konstruktion der GNF geht hervor, dass die Grammatiken

$$G_{(X,j)} := \langle \emptyset, P_{(X,j)} \cup P^{\bar{X}} \rangle$$

zu G und zueinander produktionsäquivalent sind ($P^{\bar{X}} = \{X' \rightarrow K \in P \mid X' \neq X\}$).

Die einzelnen $G_{(X_{\sigma,j})}$ werden nun zunächst in Formeln $s[G_{(X_{\sigma,j})}]$ und anschließend das darin enthaltene σ in eine eigene Rearrangement-Regel ($\text{REARR}\sigma_j$) übersetzt. Diese dient ausschließlich der Konkretisierung des j -ten Parameters x_j von σ und enthält in der Regel weniger Prämissen, als die allgemeine Regel ($\text{REARR}\sigma$).

Auf Effizienzgründen ist es sinnvoll, nicht die vollständigen Grammatiken $G_{(X_{\sigma,j})}$ sondern an deren Stelle nur die Produktionsregeln für X_σ in die Prädikatdefinition von σ zu übersetzen. Da alle anderen Produktionsregeln, also $P^{\bar{X}}$ unverändert in $G_{(X_{\sigma,j})}$ übernommen werden und sich die Definition der anderen Prädikate deshalb nicht ändert, sind die generierten Rearrangement-Regeln zueinander kompatibel.

Zur Erzeugung der Abstraktionsregeln hingegen kann die gesamte Grammatik G beziehungsweise deren Übersetzung $s[G]$ verwendet werden, sofern sie rückwärtig konfluent ist. Dies ist unproblematisch, da ohnehin zu jedem Prädikat eine eigene Beweisregel mit jeweils nur einer Prämisse erzeugt wird. Ist die rückwärtige Konfluenz der Grammatik nicht gegeben, kann sie

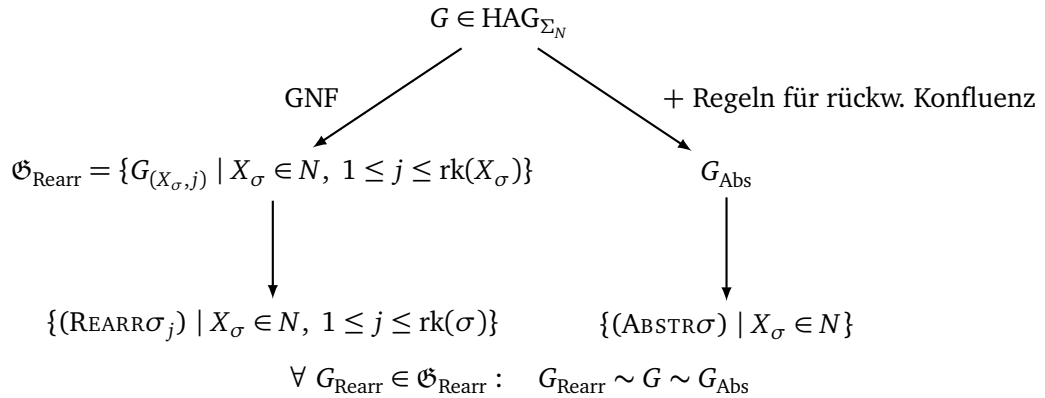


Abbildung 6.1.: Herleitung von Rearrangement- und Abstraktionsregeln

unter Umständen durch Hinzufügen weiterer Regeln hergestellt werden. Das Resultat muss jedoch zur ursprünglichen Grammatik stets produktionsäquivalent bleiben. Die rückwärtige Konfluenz einer HAG ist nach [Nel10] entscheidbar. Abbildung 6.1 fasst noch einmal die verwendeten Grammatiken und deren Beziehung zueinander zusammen.

6.3. VERIFIKATION EINES BEISPIELPROGRAMMS

Dieser Abschnitt zeigt die Verifikation eines einfachen Beispielprogramms, das eine Liste traversiert. Dazu werden aus einer HAG erzeugte Rearrangement- und Abstraktionsregeln genutzt. Alle verwendeten Beweisregeln, von (REARR) und (ABSTR) abgesehen, stammen aus [Rey08]. Die Programmiersprache wurde ebenfalls von dort übernommen und ist weitestgehend intuitiv. Eckige Klammern [] notieren darin die Dereferenzierung einer Adresse. Der Ausdruck x steht also für den Wert der Variablen ([x]i) und [x] für den Wert des Heaps an dieser Adresse, also h([x]i). Der Zugriff auf das Feld s eines auf dem Heap gespeicherten Objektes erfolgt mittels [x.s]. Die Zuweisung x := e wertet den Ausdruck e aus und speichert seinen Wert in x. Die While-Schleife while (b) do c iteriert den Ausdruck c solange, wie Bedingung b erfüllt ist. Alle Anweisungen werden mit ; abgeschlossen.

PROGRAMM

Wir betrachten das folgende Programm, das eine mindestens zweielementige Liste von x nach y mit der Variable z durchläuft.

```

z.n := x;
while (z ≠ y) do
  z := [z.n];

```

DATENSTRUKTUR

Als Datenstruktur verwenden wir eine einfach verkettete Liste, die durch die rückwärtig konfluente Grammatik G (Abbildung 6.2) definiert wird.

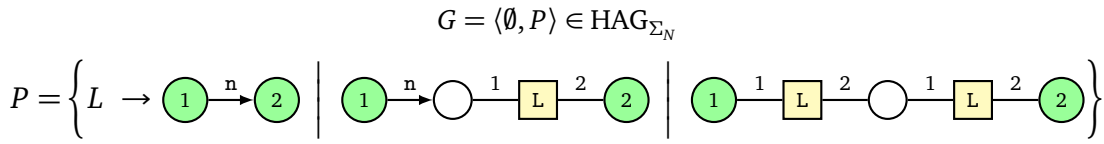


Abbildung 6.2.: HAG der verwendeten Datenstruktur

Die daraus generierte vollständige Prädikatdefinition $s_{\mathcal{P}}\llbracket P \rrbracket$ lautet

$$\sigma_L(x_1, x_2) = (x_1.n \mapsto x_2) \vee (\exists r : x_1.n \mapsto r * \sigma_L(r, x_2)) \vee (\exists r : \sigma_L(x_1, r) * \sigma_L(r, x_2)).$$

$\sigma_L(x, y)$ repräsentiert demnach eine Liste beliebiger Länge von x nach y . Da G bereits rückwärtig konfluent ist, müssen keine weiteren Regeln hinzugefügt werden, das heißt $G_{\text{Abs}} = G$.

Für das Rearrangement verwenden wir die Menge der lokal konkretisierbaren Teilgrammatiken $G_{(X,j)}$. Da es in unserem Beispiel nur ein einziges Nichtterminal gibt, das seinerseits über genau einen Nicht-Reduktionstentakel (vgl. Definition 2.15) verfügt – der zweite Tentakel erzeugt an den inzidenten Knoten ausschließlich eingehende n -Kanten und ist daher Reduktionstentakel – wird lediglich eine Rearrangement-Grammatik benötigt:

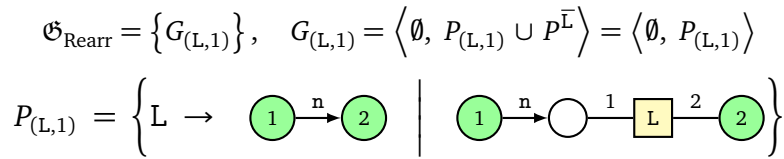


Abbildung 6.3.: Grammatik für das Rearrangement

Die dritte Produktionsregel ist nicht Teil von $P_{(L,1)}$, da sich durch sie die ausgehende n -Kante aus $\text{ext}(1)$ nicht in einem Ableitungsschritt erzeugen lässt, obwohl $n \in \text{type}(L, 1)$. Die resultierende Prädikatdefinition $s_{\mathcal{P}}\llbracket P_{(L,1)} \rrbracket$ lautet entsprechend

$$\sigma_L(x_1, x_2) = (x_1.n \mapsto x_2) \vee (\exists r : x_1.n \mapsto r * \sigma_L(r, x_2)).$$

BEWEISREGELN

Die nach Definition 6.1 und 6.2 generierten Beweisregeln für die Datenstruktur sind in Abbildung 6.4 dargestellt.

$$\begin{array}{c}
 \frac{\{y_1.n \mapsto y_2 * \varphi\} \text{ c } \{\varphi'\}}{\{\exists r : y_1.n \mapsto r * \sigma_L(r, y_2) * \varphi\} \text{ c } \{\varphi'\}} \text{ (REARR}\sigma_1\text{)} \\
 \text{(ABSTR}\sigma\text{)} \frac{\frac{\{\varphi\} \text{ c } \{y_1.n \mapsto y_2 * \varphi'\}}{\{\varphi\} \text{ c } \{\sigma_L(y_1, y_2) * \varphi'\}} \quad \frac{\{\varphi\} \text{ c } \{\exists r : y_1.n \mapsto r * \sigma_L(r, y_2) * \varphi'\}}{\{\varphi\} \text{ c } \{\sigma_L(y_1, y_2) * \varphi'\}} \text{ (ABSTR}\sigma\text{)}}{\frac{\{\varphi\} \text{ c } \{\exists r : \sigma_L(y_1, r) * \sigma_L(r, y_2) * \varphi'\}}{\{\varphi\} \text{ c } \{\sigma_L(y_1, y_2) * \varphi'\}} \text{ (ABSTR}\sigma\text{)}}
 \end{array}$$

Abbildung 6.4.: Abgeleitete Rearrangement- und Abstraktionsregeln

SPEZIFIKATION

Die Spezifikation des Programms wird als Tripel mit Vorbedingung und Nachbedingung formuliert. Als Vorbedingung legen wir fest, dass eine Liste mit mindestens zwei Elementen von x nach y existiert, dass beide Variablen verschieden sind, die Liste also keinen Zykel bildet, und der n -Selektor von y auf **nil** verweist. Die Vorbedingung lautet daher

$$x \neq y \wedge y.n \mapsto \mathbf{nil} \wedge \sigma_L(x, y).$$

Die Verwendung des Ungleich-Operators \neq ist hier problemlos möglich, da sich die betrachtete Assertion nicht in einer Prädikatdefinition befindet oder in einen Graphen übersetzt werden soll.

Nachdem das Programm terminiert, soll z am Ende der Liste angekommen sein. x und y sollen weiterhin verschieden sein, $y.n$ noch immer auf **nil** zeigen und die Liste nach wie vor existieren. Als Nachbedingung verwenden wir also

$$z = y \wedge x \neq y \wedge y.n \mapsto \mathbf{nil} \wedge \sigma_L(x, y),$$

so dass sich die Spezifikation

$$\begin{array}{l}
 \{x \neq y \wedge y.n \mapsto \mathbf{nil} \wedge \sigma_L(x, y)\} \\
 z.n := x; \\
 \mathbf{while} (z \neq y) \mathbf{do} \\
 \quad z := [z.n]; \\
 \{z = y \wedge x \neq y \wedge y.n \mapsto \mathbf{nil} \wedge \sigma_L(x, y)\}
 \end{array}$$

ergibt.

VERIFIKATION

Die Verifikation des Programmes erfolgt mittels der SL-Beweisregeln von Reynolds [Rey08]. Die Vor- und Nachbedingungen werden dazu in geschweiften Klammern $\{ \}$ zwischen den Programmzeilen notiert. Auf diese Art und Weise lässt sich ein Beweisbaum, der von der Sequenz- und Konsequenzregel abgesehen keine Verzweigungen enthält, kompakt darstellen. Folgen

Kapitel 6. Beweisregeln für die SL-Verifikation

$$\begin{array}{c}
\frac{}{\{z = x \wedge z.n \mapsto y\} z := [z.n] \{z = y \wedge x.n \mapsto y\}} \text{(LkL)} \\
\frac{}{\{z = x \wedge x.n \mapsto y\} z := [z.n] \{z = y \wedge x.n \mapsto y\}} \text{(SP)} \\
\frac{}{\{z = x \wedge x.n \mapsto y\} z := [z.n] \{z = y \wedge \sigma(x, y)\}} \text{(ABSTR}\sigma\text{)} \\
\hline
\mathbf{A} = \{z = x \wedge x.n \mapsto y\} z := [z.n] \{\sigma(x, z) * \sigma(z, y) \vee z = y \wedge \sigma(x, y)\} \text{(WC)} \\
\\
\frac{}{\{x = z \wedge z.n \mapsto r\} z := [z.n] \{r = z \wedge x.n \mapsto z\}} \text{(LkL)} \\
\frac{}{\{x = z \wedge x.n \mapsto r\} z := [z.n] \{r = z \wedge x.n \mapsto z\}} \text{(SP)} \\
\frac{}{\{x = z \wedge x.n \mapsto r * \sigma(r, y)\} z := [z.n] \{r = z \wedge x.n \mapsto z * \sigma(z, y)\}} \text{(FRAME)} \\
\frac{}{\{x = z \wedge (\exists r : x.n \mapsto r * \sigma(r, y))\} z := [z.n] \{\exists r : r = z \wedge x.n \mapsto z * \sigma(z, y)\}} (\exists) \\
\frac{}{\{x = z \wedge (\exists r : x.n \mapsto r * \sigma(r, y))\} z := [z.n] \{x.n \mapsto z * \sigma(z, y)\}} \text{(WC)} \\
\frac{}{\{x = z \wedge (\exists r : x.n \mapsto r * \sigma(r, y))\} z := [z.n] \{\sigma(x, z) * \sigma(z, y)\}} \text{(ABSTR}\sigma\text{)} \\
\hline
\mathbf{B} = \{x = z \wedge (\exists r : x.n \mapsto r * \sigma(r, y))\} z := [z.n] \{\sigma(x, z) * \sigma(z, y) \vee z = y \wedge \sigma(x, y)\} \text{(WC)} \\
\\
\frac{}{\{z.n \mapsto y \wedge r = z\} z := [z.n] \{z = y \wedge r.n \mapsto y\}} \text{(LkL)} \\
\frac{}{\{\sigma(x, r) * z.n \mapsto y \wedge r = z\} z := [z.n] \{z = y \wedge \sigma(x, r) * r.n \mapsto y\}} \text{(FRAME)} \\
\frac{}{\{\sigma(x, z) * z.n \mapsto y \wedge r = z\} z := [z.n] \{z = y \wedge \sigma(x, r) * r.n \mapsto y\}} \text{(SP)} \\
\frac{}{\{\exists r : \sigma(x, z) * z.n \mapsto y \wedge r = z\} z := [z.n] \{z = y \wedge \exists r : \sigma(x, r) * r.n \mapsto y\}} (\exists) \\
\frac{}{\{\sigma(x, z) * z.n \mapsto y\} z := [z.n] \{z = y \wedge \exists r : \sigma(x, r) * r.n \mapsto y\}} \text{(SP)} \\
\frac{}{\{\sigma(x, z) * z.n \mapsto y\} z := [z.n] \{z = y \wedge \exists r : \sigma(x, r) * \sigma(r, y)\}} \text{(ABSTR}\sigma\text{)} \\
\frac{}{\{\sigma(x, z) * z.n \mapsto y\} z := [z.n] \{z = y \wedge \sigma(x, y)\}} \text{(ABSTR}\sigma\text{)} \\
\hline
\mathbf{C} = \{\sigma(x, z) * z.n \mapsto y\} z := [z.n] \{\sigma(x, z) * \sigma(z, y) \vee z = y \wedge \sigma(x, y)\} \text{(WC)} \\
\\
\frac{}{\{t = z \wedge z.n \mapsto r\} z := [z.n] \{r = z \wedge t.n \mapsto z\}} \text{(LkL)} \\
\frac{}{\{t = z \wedge z.n \mapsto r\} z := [z.n] \{r = z \wedge t.n \mapsto r\}} \text{(WC)} \\
\frac{}{\{\exists t : t = z \wedge z.n \mapsto r\} z := [z.n] \{r = z \wedge \exists t : t.n \mapsto r\}} (\exists) \\
\frac{}{\{\exists t : t = z \wedge \sigma(x, t) * z.n \mapsto r\} z := [z.n] \{r = z \wedge \exists t : \sigma(x, t) * t.n \mapsto r\}} \text{(FRAME)} \\
\frac{}{\{\exists t : t = z \wedge \sigma(x, t) * z.n \mapsto r\} z := [z.n] \{r = z \wedge \exists t : \sigma(x, t) * \sigma(t, r)\}} \text{(ABSTR}\sigma\text{)} \\
\frac{}{\{\exists t : t = z \wedge \sigma(x, t) * z.n \mapsto r\} z := [z.n] \{r = z \wedge \sigma(x, r)\}} \text{(ABSTR}\sigma\text{)} \\
\frac{}{\{\exists t : t = z \wedge \sigma(x, t) * z.n \mapsto r * \sigma(r, y)\} z := [z.n] \{r = z \wedge \sigma(x, r) * \sigma(r, y)\}} \text{(FRAME)} \\
\frac{}{\{\exists t : t = z \wedge \sigma(x, t) * \exists r : z.n \mapsto r * \sigma(r, y)\} z := [z.n] \{\exists r : r = z \wedge \sigma(x, r) * \sigma(r, y)\}} (\exists) \\
\frac{}{\{\sigma(x, z) * \exists r : z.n \mapsto r * \sigma(r, y)\} z := [z.n] \{\sigma(x, z) * \sigma(z, y)\}} \text{(SP + WC)} \\
\hline
\mathbf{D} = \{\sigma(x, z) * \exists r : z.n \mapsto r * \sigma(r, y)\} z := [z.n] \{\sigma(x, z) * \sigma(z, y) \vee z = y \wedge \sigma(x, y)\} \text{(WC)}
\end{array}$$

Die Gültigkeit von \star ist damit gezeigt, die geforderte Spezifikation

$$\begin{array}{l}
\{x \neq y \wedge y.n \mapsto \mathbf{nil} \wedge \sigma_L(x, y)\} \\
z.n := x; \\
\mathbf{while} (z \neq y) \mathbf{do} \\
\quad z := [z.n]; \\
\{z = y \wedge x \neq y \wedge y.n \mapsto \mathbf{nil} \wedge \sigma_L(x, y)\}
\end{array}$$

folglich korrekt. △

KAPITEL 7

ZUSAMMENFASSUNG UND AUSBLICK

Die vorliegende Ausarbeitung beschäftigt sich mit zwei verschiedenen Ansätzen zur Modellierung und Abstraktion des Heaps und dynamischer Datenstrukturen. Nachfolgend werden noch einmal die dabei verwendeten Methoden, sowie die wichtigsten Ergebnisse zusammengefasst. Im Anschluss wird ein Ausblick auf offene Fragen und alternative Lösungsansätze für die vorgestellten Problemstellungen gegeben.

7.1. ZUSAMMENFASSUNG

Der Einsatz dynamischer Datenstrukturen stellt heutige Programmanalyse- und Verifikationsverfahren vor das Problem, dass sie einen meist unendlichen Zustandsraum definieren, mit dem diese nicht arbeiten können. Um dies zu umgehen, müssen geeignete Methoden zur Abstraktion der Datenstrukturen gefunden werden, die den Zustandsraum auf endliche Größe reduzieren. In dieser Arbeit wurden dazu zwei Ansätze vorgestellt, miteinander verglichen und ineinander überführt. Dies sind zum einen der von Heinen et al. entwickelte Graphgrammatik-Ansatz [HJKN12] und zum anderen die Separation Logic von Reynolds und O’Hearn [Rey02].

Zur Einführung des ersten Ansatzes wurde zunächst der Grundgedanke der Modellierung von Datenstrukturen durch Graphen vorgestellt. Um eine Klasse von Datenstrukturen, wie beispielsweise Listen oder binäre Bäume, zu definieren, wurden Hyperkantenersetzungsgrammatiken eingeführt und formal definiert. Diese substituieren – analog zu normalen Stringgrammatiken – Platzhalter innerhalb eines Graphen, sogenannte Nichtterminalkanten, durch weitere Subgraphen. Dabei wird die Nichtterminalkante aus diesem entfernt und der Subgraph an derselben Stelle eingebettet. Um dem Umstand Rechnung zu tragen, dass ein solcher Subgraph an mehr als zwei Punkten mit dem umgebenen Graphen verbunden sein kann, wurden Kanten höheren Ranges, sogenannte *Hyperkanten*, die also mehr als zwei Knoten miteinander verbinden können, eingeführt.

Die Klasse der Hyperkantenersetzungsgrammatiken, wurde weiter eingeschränkt auf die der *Datenstrukturgrammatiken* und schließlich auf *Heapabstraktionsgrammatiken*. Letztere zeichnen sich durch besondere strukturelle Eigenschaften aus, die für die Abstraktion und deren Gegenpart, die Konkretisierung, zwingend erforderlich sind. Diese Eigenschaften wurden als *Produktivität*, *Wachstum*, *Getyptheit* und *lokale Konkretisierbarkeit* formalisiert.

Der zweite vorgestellte Ansatz ist die Separation Logic, eine Erweiterung der Hoare-Logik, die Aussagen über den Heap ermöglicht. Diese werden durch *Zeigerassertionen* in Verbindung

mit der *separierenden Konjunktion* realisiert. Ihre Semantik wurde über eine Modellrelation definiert. Sie definiert, wann ein Paar, bestehend aus einem Heap und einer Variableninterpretation, zweier partieller Abbildungen, eine Formel (Assertion) erfüllt. Datenstrukturen können in der SL über *Prädikate* definiert werden. Solche Prädikate werden mit einer Liste von Argumenten aufgerufen und spezifizieren einen bestimmten Sachverhalt über diese. Dies kann beispielsweise die Existenz einer doppelt-verketteten Liste zwischen zwei Heapadressen sein. Im Gegensatz zu den meisten Veröffentlichungen zu diesem Thema, wo die Semantik der Prädikate von außen vorgegeben ist, wurde deren Definition hier analog zu [DP08b] direkt in die betrachteten Formeln eingebettet (**let**-Konstruktion). Ein Prädikat wird dabei selbst wiederum mittels einer Assertion über seine Parameter definiert.

Die Verkettung einzelner Elemente in einer Datenstruktur wird durch den rekursiven Aufruf von Prädikaten erreicht. Um dynamische Datenstrukturen zu realisieren, werden rekursive Prädikate verwendet, also solche, die sich selbst aufrufen. Ein Prädikat, das ausgehend von einem Wurzelknoten einen Baum beschreibt, spezifiziert für seinen Parameter zum Beispiel die Existenz eines rechten und linken Sohnknotens, so dass diese ebenfalls Wurzel eines Baumes sind. Dazu ruft sich das Prädikat mit den beiden Söhnen als Argumente selbst auf. Die Separation Logic verfügt über eine mächtigere Ausdrucksstärke als HRGen. Um die Mächtigkeit beider Ansätze anzugleichen, wurde ein SL-Fragment (SLF), das insbesondere die Verwendung von (nicht-separierender) Konjunktion, Negation und dem Ausdruck **true** verbietet, auf syntaktischer Basis definiert und dessen Semantik formal eingeführt.

Anschließend wurden HRGen und SLF einander gegenübergestellt wobei insbesondere die Teilkonzepte beider Ansätze miteinander verglichen wurden. Dabei wurden zahlreiche Analogien, insbesondere zwischen Parametern und Nichtterminalen beziehungsweise zwischen Parameterdefinitionen und grammatikalischen Produktionsregeln herausgearbeitet. Insbesondere wurden auch die konzeptionellen Unterschiede zwischen beiden Ansätzen ausführlich beleuchtet und mehrere Alternativen vorgestellt, diese zu überwinden. Hier seien noch einmal die Diskrepanz zwischen dem expliziten **nil** von SLF und der impliziten **nil**-Auffassung der HRGen, sowie die Verwendung von *Field Splitting* im Gegensatz zum *Field Aggregating* erwähnt. Im Anschluss daran wurde eine funktional definierte Grammatiksynthese vorgestellt, die eine SLF-Formel mitsamt ihrer eingebetteten Prädikate in eine Hyperkantenersetzungsgrammatik übersetzt. Diese ist an die von Dodds in seiner Dissertation vorgestellte Übersetzung angelehnt, wurde jedoch um die Unterstützung für implizites **nil**, *Field Splitting* mit beliebigen Selektoren und um Variablenkanten erweitert, die eine Reihe neuer Herausforderungen mit sich brachten.

Um die DSG-Eigenschaften der synthetisierten Grammatik zu sichern wurden Bedingungen für die Ausgangsformel erarbeitet und deren Äquivalenz zu den DSG-Anforderungen gezeigt. Desweiteren wurde ein Algorithmus vorgestellt, der ein hinreichendes und für auswertbare Prädikate auch notwendiges Kriterium für diese Anforderungen auf Formeln entscheidet. Zusätzlich zur syntaktischen Einschränkung auf SLF ist darüber hinaus die Erfüllung dreier semantischer Eigenschaften notwendig, um auf Basis der gegebenen Ansätze eine korrekte Grammatiksynthese durchführen zu können. Diese sind in polynomieller entscheidbar.

Für die Gegenrichtung, die Generierung von SLF-Formeln aus DSGs, wurde ebenfalls eine Übersetzung angegeben. Diese kommt – mit Ausnahme der Getyptheit, die für die Erzeugung expliziter Nullzeigerassertionen notwendig ist – ohne weitere Randbedingungen seitens

der Grammatiken aus. Die Korrektheit im Sinne der Semantikerhaltung wurde für beide Übersetzungen auf Basis einer Graphfunktion, die zu einem Heap-Variableninterpretations-Paar den äquivalenten Hypergraphen zurückgibt, gezeigt.

Um eine Datenstrukturgrammatik für die Abstraktion und Konkretisierung von Datenstrukturen einsetzen zu können, muss sie die vier Eigenschaften *Produktivität*, *Getyptheit*, *Wachstum* und *lokale Konkretisierbarkeit* haben. Grammatiken, die diese Bedingungen erfüllen, wurden als Heapabstraktionsgrammatiken bezeichnet. Im Rahmen dieser Arbeit wurden äquivalente Eigenschaften für Formeln definiert, von denen zwei ebenfalls die Namen *Getyptheit* und *Wachstum* tragen und die beiden anderen als *Auswertbarkeit* und *variablenspezifische Konkretisierbarkeit* vorgestellt wurden. Die Erhaltung dieser Eigenschaften sowohl unter der Grammatiksynthese als auch unter der Formelgenerierung wurde separat für alle vier gezeigt. Basierend auf diesen Ergebnissen wurde die Verwendbarkeit von aus Grammatiken generierten SL-Prädikaten für die Verifikation von Programmen erläutert. Dazu wurde insbesondere auf die Rolle der im vorangegangenen Kapitel hergeleiteten Eigenschaften im Kontext des Verifikationsprozesses eingegangen. Es wurden allgemeine Regeln für die Abstraktion und das Rearrangement von generierten SLF-Prädikaten innerhalb von SL-Beweisbäumen angegeben und deren Anwendbarkeit an einem abschließenden Beispielprogramm mit Heapzugriff validiert.

7.2. AUSBLICK

Bei der Überführung von DSGs in Separation-Logic-Assertionen traten einige Probleme auf, die auf die konzeptionellen Differenzen beider Ansätze zurückzuführen sind und die im Rahmen dieser Arbeit noch nicht untersucht wurden. Der wohl wichtigste Punkt ist die Einführung einer expliziten **nil**-Repräsentanz in die Domäne der Hypergraphen. Diese erhöht die Mächtigkeit von Graphgrammatiken dahingehend, als das nicht spezifiziertere oder nicht vorhandene Selektoren von solchen unterschieden werden können, die auf **nil** verweisen. In der Separation Logic ist es üblich, **nil** auch als Argumente an Prädikate übergeben zu können. Auf Seiten der DSGs wäre dazu ebenfalls ein dedizierter **nil**-Knoten notwendig, der mit dem entsprechenden Tentakeln der Nichtterminale verbunden wird. Für die Einführung eines solchen Knotens müssen jedoch weitere Anforderungen sichergestellt sein. So darf beispielsweise keine Selektorkante den **nil**-Knoten verlassen. Ebenso wenig dürfen Prädikate Felder auf Parametern definieren, an die potentiell **nil** übergeben wird. Eine Gliederung in zwei Parameterklassen, analog zu den bei HRGen verwendeten Reduktions- und Nicht-Reduktionstentakeln, wäre eine Möglichkeit, diesem Problem zu begegnen.

Es wurde gezeigt, dass die DSG-Zugehörigkeit für synthetisierte Grammatiken auf den Ausgangsformeln entscheidbar ist. Sollte die Grammatiksynthese in realen Applikationen zum Einsatz kommen, wären alternative Methoden, die die DSG-Eigenschaften in polynomieller Zeit überprüfen oder sogar herstellen, zu untersuchen. In diesem Fall sind auch eine exakte Laufzeitabschätzung der Graphsynthese selbst beziehungsweise eine effiziente Implementierung selbiger in Angriff zu nehmen.

Die ausführlichere Analyse der HAG-äquivalenten Eigenschaften bietet ebenfalls die Möglichkeit weiterer Untersuchungen. Insbesondere die Klärung der Fragen, inwieweit diese Eigenschaften für die Verifikation mit beliebigen Prädikaten ohne direkten HRG-Bezug ent-

Kapitel 7. Zusammenfassung und Ausblick

schieden oder sogar hergestellt werden können, ist erstrebenswert. Auf diese Weise könnten sich gegebenenfalls Abstraktions- und Konkretisierungsregeln auch ohne zugehörige Grammatik erzeugen lassen, was die praktische Eignung der Separation Logic für die Verifikation signifikant verbessern würde.

ANHANG

NOTATION

A.1. VERWENDETE FORMELZEICHEN

Die in der Arbeit auftretenden Formelzeichen wurden einheitlich für spezifische Konzepte und Domänen verwendet. Die folgende Tabelle listet die häufigsten von ihnen auf.

Domäne	Formelzeichen
Hypergraph	H, H', K, K', \dots
Getaggtter Hypergraph	$\mathcal{H}, \mathcal{H}', \dots, (H, t), \dots$
Graphmenge	$\mathfrak{H}, \mathfrak{H}', \dots, \mathfrak{H}$ (Startgraphmenge)
Grammatik	G, G', \dots
SL(F)-Formel	$\varphi, \varphi', \psi, \psi', \dots$
Prädikatname	σ, σ', \dots
Variable	x, y, z, \dots
konkrete Variable	$\mathbf{x}, \mathbf{y}, \mathbf{z}, \dots$
... insb. Parameter	$\mathbf{x}_1, \mathbf{x}_2, \dots$
Selektor	s, u, w, \dots
konkreter Selektor	$\mathbf{s}, \mathbf{u}, \mathbf{w}, \mathbf{n}, \mathbf{p}, \dots$
Nichtterminalsymbol	X, X', Y, Y', \dots
konkretes Nichtterminalsym.	L, X, B, \dots
Knoten	v, v', \dots
Kante	e, e', \dots
Wert (\in Elem)	el, el', \dots
Produktionsregel	π, π', \dots
Regelsatz	P, P', \dots
Heap	h, h', \dots
Variableninterpretation	i, i', \dots
Prädikatinterpretation	η, η', \dots
Prädikatdefinition(en)	Γ, Γ', \dots
Index (nat. Zahl)	j, k, \dots
Graphumgebung	λ, λ', \dots
Analyseinformation	AI, \tilde{AI}

Variablen in kursivem Satz x, y, \dots stehen für beliebige Elemente aus Var beziehungsweise var_Σ , je nach Kontext. Variablen in dicktengleichem Satz x, y, \dots repräsentieren ein konkretes Element von $\text{Var}/\text{var}_\Sigma$. Mit Ausnahme von Parametervariablen treten sie daher nur in Beispielen auf. Analoges gilt für die Selektoren.

A.2. FUNKTIONEN

Funktionen werden in dieser Arbeit folgendermaßen notiert

$$\begin{aligned} f = [] &\iff \text{dom}(f) = \emptyset \\ f = [x \mapsto y] &\iff \text{dom}(f) = x \text{ und } f(x) = y \\ f = f'[x_1 \mapsto y_1, x_2 \mapsto y_2, \dots] &\iff \text{dom}(f) = \text{dom}(f') \cup \{x_1, x_2, \dots\} \text{ und} \\ & f(x) = \begin{cases} y_1 & , x = x_1 \\ y_2 & , x = x_2 \\ \vdots & \vdots \\ f'(x) & , \text{sonst} \end{cases} \end{aligned}$$

LITERATURVERZEICHNIS

- [Bal04] BALZERT, Helmut: *Grundlagen der Informatik*. Spektrum Akademischer Verlag, 2004
- [BCO04] BERDINE, Josh ; CALCAGNO, Cristiano ; O’HEARN, Peter W.: A Decidable Fragment of Separation Logic. In: *Foundations of Software Technology and Theoretical Computer Science*, 2004
- [BCO05] BERDINE, Josh ; CALCAGNO, Cristiano ; O’HEARN, Peter W.: Symbolic Execution with Separation Logic. In: *Programming Languages and Systems*. Springer Berlin, Heidelberg, 2005
- [BK08] BAIER, C. ; KATOEN, J.P.: *Principles of Model Checking*. The MIT Press, 2008
- [Dod08] DODDS, Mike: *Graph Transformation and Pointer Structures*, The University of York, Department of Computer Science, Diss., 2008
- [DOY06] DISTEFANO, Dino ; O’HEARN, Peter W. ; YANG, Hongseok: A Local Shape Analysis Based on Separation Logic. In: *Proceedings fo the 12th International Conference of Tools and Algorithms for the Construction and Analysis of Systems*, 2006
- [DP08a] DISTEFANO, Dino ; PARKINSON, Matthew J.: jStar: towards practical verification for java. In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008
- [DP08b] DODDS, Mike ; PLUMP, Detlef: From Hyperedge Replacement to Separation Logic and Back. In: *Proceedings of the Doctoral Symposium at the International Conference on Graph Transformation*, 2008
- [Eng97] ENGELFRIET, Joost: Context-free graph grammars. In: *Handbook of formal languages*. Springer-Verlag New York, Inc., 1997, S. 125–213
- [HJKN12] HEINEN, Jonathan ; JANSEN, Christina ; KATOEN, Joost-Pieter ; NOLL, Thomas: Juggernaut: Using Graph Grammars for Abstracting Unbounded Heap Structures / Math. Struct. in Comp. Science. 2012. – Forschungsbericht
- [Hoa69] HOARE, Charles Antony R.: An axiomatic basis for computer programming. In: *Commun. ACM* 12 (1969), S. 576–580

LITERATURVERZEICHNIS

- [JHKN11] JANSEN, Christina ; HEINEN, Jonathan ; KATOEN, Joost-Pieter ; NOLL, Thomas: A Local Greibach Normal Form for Hyperedge Replacement Grammars. In: *5th international Conference on Language and Automata Theory and Applications*, 2011
- [Nel10] NELLEN, Johanna: *Konfluenzanalyse und Vervollständigung von Graphersetzungssystemen*, RWTH Aachen, Diplomarbeit, 2010
- [Nol12] NOLL, Thomas: *Semantics and Verification of Software*. Vorlesungsfolien, Wintersemester 2011/12
- [Rey02] REYNOLDS, John C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: *17th Annual IEEE Symposium on Logic in Computer Science*, 2002
- [Rey08] REYNOLDS, John C.: *An Introduction to Separation Logic*. 2008. – ITU University Copenhagen
- [Woo87] WOOD, Derick: *Theory of Computation*. John Wiley and Sons, 1987