Generierung von Hyperkantenersetzungsgrammatiken zur Heapabstraktion

Christina Jansen[1] und Jonathan Heinen[2]

Lehrstuhl für Informatik 2, RWTH Aachen

Zusammenfassung. Bei der Verifizierung von heapbasierten, dynamischen Datenstrukturen stellen unendliche Zustandsräume ein Problem dar. Die potentiell unendliche Menge von Heapzuständen läßt sich durch Abstraktion endlich repräsentieren und mithilfe dieser Repräsentation durch Techniken wie Model Checking verifizieren. Grundlage dieser Abstraktion sind Hyperkantenersetzungsgrammatiken, wobei die Form einer geeigneten Grammatik stark von der zugrundeliegenden Datenstruktur und der Anwendung abhängt. Erfüllt sie dabei alle notwendigen Eigenschaften bezeichnet man sie als Heapabstraktionsgrammatik. Wir möchten vorstellen, wie die Konstruktion einer solchen Heapabstraktionsgrammatik abläuft und wie das Lernen von Produktionsregeln während der Programmausführung aussehen kann.

1 Einleitung

Heapbasierte Datenstrukturen spielen in den heutigen Programmiersprachen eine wichtige Rolle. Dadurch, dass zur Laufzeit Objekte erstellt oder entfernt werden können, und damit die Menge der Heapzustände im Allgemeinen unendlich und zur Kompilierzeit unabsehbar ist, stellen sie für die meistgenutzten Verifikationstechniken ein Problem dar. Man ist daher bemüht potentiell unendliche Strukturen durch endliche Abstraktionen zu repräsentieren ohne relevante Informationen zu verlieren.

Eine Möglichkeit zur Abstraktion von Heapzuständen bietet das Konzept der Hyperkantenersetzungsgrammatiken (HRGs) [1]. Dabei wird ein Heapzustand durch einen Hypergraphen repräsentiert, die Produktionsregeln der Grammatik werden zum Abstrahieren und Konkretisieren der Hypergraphen angewandt. Die grundlegende Idee besteht darin, die Rückwärtsanwendung von Produktionsregeln zuzulassen, diesen Vorgang bezeichnen wir als Abstraktion. Falls notwendig können abstrahierte Teilgraphen durch normale Regelanwendung wieder konkretisiert werden. Dies hat zur Folge, dass keine Semantik für die abstrakten Teile der Graphen gebraucht wird, da Operationen nur auf konkreten Teilgraphen ausgeführt werden müssen. [2]

Zur korrekten Abstraktion von Heapzuständen muss eine HRG einige notwendige Eigenschaften erfüllen, damit z.B. gewährleistet wird, dass keine relevanten

Informationen verloren gehen etc. Die rückwärtige Nutzung von Produktionsregeln führt zusätzlich zu weiteren Anforderungen an die Grammatik, damit Dinge wie Terminierung oder die Existenz entsprechender Konkretisierungsregeln garantiert werden können. Für die Abstraktion geeignete HRGs nennen wir Heapabstraktionsgrammatiken. Die Konstruktion einer solchen zu einer gegebenen Grammatik läßt sich automatisch durchführen, die Idee dabei ist es, die HRG in eine spezielle Normalform zu bringen aus der sich die gewünschten Eigenschaften durch einfache Schritte ableiten lassen.

Eine Frage bei diesem Ansatz der Heapabstraktion ist weiterhin die Wahl einer geeigneten Eingabegrammatik. Denn es existiert keine universelle Heapabstraktionsgrammatik, vielmehr hängt diese stark von dem betrachteten Problem ab und muss individuell für dieses erstellt werden. Man hat demnach verschiedene Möglichkeiten eine HRG zu bestimmen: manuell durch genaue Betrachtung der Datenstruktur und der zu verifizierenden Anwendung, Inferenz einer Grammatik durch Eingabe einer Beispielmenge von Graphen und Lernen von Grammatikregeln während der Programmausführung.

Sowohl die Konstruktion einer Heapabstraktionsgrammatik anhand einer speziellen Normalform als auch den Ansatz des Lernens einer HRG durch Kombination der drei oben genannten Möglichkeiten, möchten wir hier vorstellen.

2 Hypergraphen & HRGs

Bevor wir zu Themen wie Heapabstraktion und dem Lernen von HRGs kommen, betrachten wir vorerst die Grundlagen von Hypergraphen und der Kantenersetzung.

2.1 Hypergraphen

Hypergraphen stellen eine Generalisierung der weit verbreiteten, "normalen" Graphen dar. Formal lassen sie sich durch ein 5-Tupel (V, E, lab, att, ext) beschreiben, wobei sie wie gewohnt Knoten V und Kanten E besitzen. Letzteren wird durch die Funktion $lab : E \to \Sigma$ eine Bezeichnung aus dem Alphabet Σ zugewiesen. Kanten in Hypergraphen dürfen eine beliebige Anzahl von anliegenden Knoten besitzen, die Attraktor-Funktion $att : E \to V^*$ ordnet dabei jeder Kante die zugehörigen Knoten zu. Zuletzt dürfen Knoten als extern gekennzeichnet werden, sie werden durch die ext-Menge beschrieben und ihre Anzahl bestimmt den Rang des Hypergraphen.

Eine graphische Darstellung eines Hypergraphen findet sich in Abb. 1. Er besitzt drei Knoten, wobei einer als extern markiert ist. Außerdem enthält er zwei Hyperkanten, eine Terminalkante (hier als übliche gerichtete Kante dargestellt) mit der Beschriftung n und eine Nichtterminalkante, die mit X bezeichnet ist.

2.2 HRGs

Eine Hyperkantenersetzungsgrammatik (engl. hyperedge replacement grammar) ermöglicht das Ableiten von Terminalgraphen durch gezielte Ersetzung von Hy-

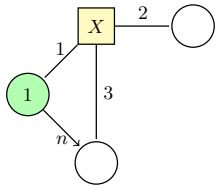

Abb. 1. Beispiel-Hypergraph

perkanten. Sie wird analog zu String-Grammatiken über ein 4-Tupel $G = (N, T, P, S)$ formalisiert, dessen Nichtterminalsymbole N und Terminalsymbole T das Alphabet $\Sigma = N \cup T$ bilden mit $T \cap N = \emptyset$. Die Produktionsregeln $(X, H) \in P$ sind auf der linken Produktionsregelseite durch Nichtterminale $X \in N$ beschriftet und enthalten einen Hypergraphen H auf der rechten Regelseite. Sie werden angewandt indem eine mit X beschriftete Hyperkante durch H ersetzt wird. Dabei werden die externen Knoten von H auf die an der Hyperkante anliegenden Knoten abgebildet. Einen solchen Ableitungsschritt nennen wir Kantenersetzung.

Abbildung 2 zeigt eine HRG für einfach verkettete Listen, denn ihre Sprache – die Menge der ableitbaren Terminalgraphen – enthält alle Listen, in denen jedes Objekt einen Zeiger auf seinen Nachfolger besitzt. Eine beispielhafte Ableitungsfolge, die eine solche Liste zum Ergebnis hat, ist in Abb. 3 zu finden.

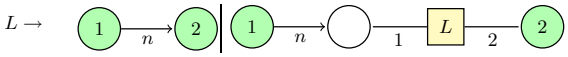

Abb. 2. HRG: einfach verkettete Liste

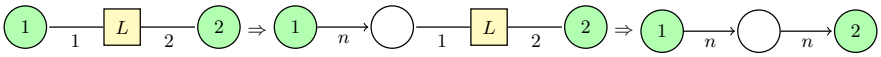

Abb. 3. Kantenersetzung: einfach verkettete Liste

3 Heapabstraktionsgrammatiken

Wie bereits erwähnt lassen sich HRGs als Abstraktionsmechanismus für eine Menge von Heapzuständen nutzen. Betrachten wir genauer wie Heapzustände als Graph dargestellt und durch die Produktionsregeln der HRG abstrahiert werden.

3.1 Heaprepräsentation

Wir repräsentieren Heapzustände durch Hypergraphen, indem wir Variablen und Zeiger durch entsprechende Kanten darstellen. Ein solcher Hypergraph wird dann Heapkonfiguration genannt. In ihm stellen Terminalkanten vom Rang 2 Zeiger dar und Terminalkanten vom Rang 1, die mit Variablen beschriftet sind, entsprechen den Variablen im Heap. Eine Heapkonfiguration mit einer Variablen x und Selektoren n ist in Abb. 4 dargestellt. Programmanweisungen, die den Heap manipulieren, werden analog im Hypergraphen durchgeführt. Eine Anweisung zur Löschung eines Objektes ist dabei nicht vorgesehen, dies geschieht stattdessen durch eine Zuweisung von nil. Abbildung 5 zeigt die Umsetzung der wichtigsten Programmanweisungen im Hypergraphen.

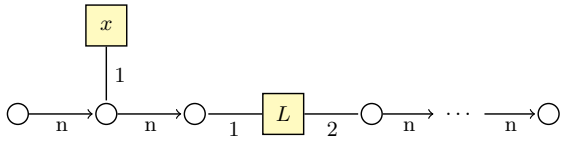

Abb. 4. Heapkonfiguration

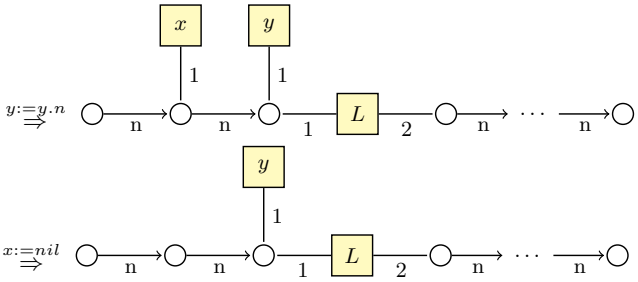

Abb. 5. Programmanweisungen

3.2 Heapabstraktion

Als weiterer Schritt soll nun eine Menge von Heapzuständen durch eine Heapkonfiguration dargestellt werden. In Abbildung 4 haben wir schon eine Heapkonfiguration, in der bereits Teile eines Heaps durch die Hyperkante mit dem Bezeichner L abstrahiert wurden, gesehen. Nichtterminalkanten repräsentieren immer einen abstrakten Teilgraphen und damit Teile eines Heaps. Ein Abstraktionsschritt entspricht dabei einer rückwärtigen Produktionsregelanwendung auf

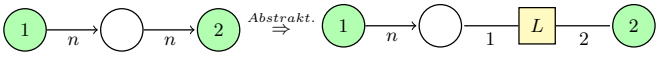

Abb. 6. Heapabstraktion

einer Heapkonfiguration. Legen wir die HRG aus Abb. 2 zugrunde, ist ein Abstraktionsschritt in Abb. 6 zu sehen.

Wie bereits erwähnt führt die Zulässigkeit der Rückwärtsanwendung von Produktionsregeln zu einigen problematischen Ableitungen, die allerdings durch zusätzliche Anforderungen an die HRG vermieden werden können. Eine zulässige Heapabstraktionsgrammatik muss die folgenden Eigenschaften besitzen:

- Produktivität
- Wachstum
- Variablenfreiheit
- Typisierung
- Lokale Apex-Eigenschaft

Produktivität wird gefordert, damit keine Hypergraphen abstrahiert werden können, die später nicht mehr zu Terminalgraphen konkretisiert werden können. Wachstum der Produktionsregeln wird benötigt um die Terminierung von Abstraktionsschritten gewährleisten zu können. Die Variablenfreiheit stellt sicher, dass keine wichtigen Informationen über Programmvariablen bei der Abstraktion verloren gehen, denn sie verbietet das Vorkommen von Variablenbezeichnern in rechten Produktionsregelseiten. Die Typisierung der Grammatik stellt sicher, dass aus mehrfachen Ableitungsschritten keine unzulässigen Hypergraphen entstehen. Sie fordert, dass jedes Nichtterminal einen festen Rang und feste ausgehende Terminalkanten an den externen Knoten besitzt. Die lokale Apex-Eigenschaft gewährleistet, dass immer, wenn ein Konkretisierungsschritt nötig ist, eine geeignete Produktionsregel existiert, die innerhalb eines Ableitungsschrittes an der gewünschten Stelle einen konkreten Teilgraphen generiert. Die Problematik, sollte dies nicht der Fall sein, läßt sich an einem kurzen Beispiel gut veranschaulichen.

Beispiel 1. Betrachten wir wiederum die in Abb. 7 diesmal leicht modifizierte HRG für einfach verkettete Listen sowie die Heapkonfigurationen aus Abb. 5. Soll nun an dem Knoten, an dem die Variablenkante y anliegt, konkretisiert werden – weil z.B. $z := y.n$ ausgeführt werden soll – ist die einzige am betreffenden Knoten konkretisierende Produktionsregel die Terminalregel der Grammatik. Wird diese jedoch angewandt, lassen sich mit der resultierenden Heapkonfiguration keine Listen beliebiger Länge mehr ableiten. Es muss daher durch eine Eigenschaft sichergestellt werden, dass immer eine geeignete Produktionsregel zur Konkretisierung vorliegt.

Von diesen fünf Eigenschaften lassen sich Produktivität, Variablenfreiheit und Typisierung relativ leicht sicherstellen. Die verbliebenen zwei – Wachstum

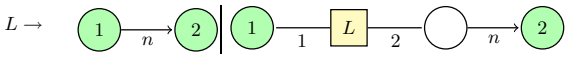

Abb. 7. Heapabstraktion

und lokale Apex-Eigenschaft – erfordern eine komplexe Konstruktion um eine äquivalente Grammatik mit diesen Eigenschaften zu erhalten. Die Idee dieser Konstruktion beruht darauf, die Eingabegrammatik in eine Normalform zu bringen, anhand derer es dann mit einfacher Kantenersetzung möglich ist eine Heapabstraktionsgrammatik abzuleiten. Einen Ansatz zur Konstruktion von äquivalenten HRGs mit Apex-Eigenschaft wird in [3] vorgestellt. Dieser fordert einerseits eine zu starke Eigenschaft, was zu unnötig grossen Regeln führt, da wir lediglich lokale Apex-Eigenschaft benötigen. Andererseits läßt er sich nur für Grammatiken anwenden, deren Sprachen beschränkt sind. Da aber Datenstrukturen wie z.B. gewurzelte Bäume oder Listen mit Zeigern auf den Wurzelknoten darstellbar sein sollen, ist diese Konstruktion für unsere Zwecke nicht ausreichend. Die Idee läßt sich allerdings auch auf für die von uns benötigte lokale Apex-Eigenschaft mit einigen Anpassungen übertragen. Sie stellt weiterhin implizit auch Wachstum der Grammatik sicher. Auch für unseren angepassten Ansatz läßt sich die Gleichheit der Sprachen von Eingabe- und resultierender Grammatik zeigen.

4 Lernen von HRGs

Wie wir bereits gesehen haben, lassen sich HRGs zur Verifizierung von dynamischen, heapbasierten Datenstrukturen einsetzen. Weiterhin ist es auch in vielen Fällen möglich zur Eingabegrammatik eine äquivalente, zulässige Heapabstraktionsgrammatik zu konstruieren. Eine Herausforderung besteht vor jedem Verifikationsprozess noch darin, eine entsprechende Eingabegrammatik zu bestimmen.

Ein Ansatz zur Inferenz einer HRG aus einer Beispielmenge von Graphen ist in [4] zu finden. Er setzt sich zusammen aus vier Basis-Funktionen – INIT, RENAME, DECOMPOSE und REDUCE –, die in beliebiger Reihenfolge und beliebig oft aufgerufen werden, um Produktionsregeln abzuleiten. Man kann zeigen, dass die so entstehenden HRGs kontextfrei sind und alle Graphen aus der Beispielmenge erkennen und damit einer Überapproximation entsprechen. Der Algorithmus ist allerdings hochgradig nichtdeterministisch und damit für unsere Anwendung nicht direkt nutzbar.

Um eine deterministische Variante zu erhalten, haben wir einige Heuristiken eingeführt, wie z.B. das Zusammenfassen von gleichartigen Nichtterminal-Bezeichnern oder die Berechnung des minimalen Schnitts als Bestimmungshilfe für die Produktionsregeln. Weiterhin erlauben wir das Lernen der HRG während der Ausführung. Sobald ein Heapzustand auftritt, der nicht vollständig abstrahiert werden kann, wird er zur Beispielmenge hinzugefügt und damit Produktionsregeln erlernt, die seine Abstraktion ermöglichen. Die Zulässigkeit der HRG wird dabei durch die im vorangegangen Abschnitt vorgestellte Konstruktion her-

gestellt. Ein solcher Ansatz bietet den Vorteil, dass zu Beginn keine Beispielmenge von Graphen als Eingabe geliefert werden muss, sondern diese während der Programmausführung automatisch entsteht. Dies führt dazu, dass alle auftretenden Heapzustände vollständig abstrahiert werden können, denn entsprechende Regeln werden bei Bedarf erlernt. Wir erlauben weiterhin die Eingabe einer HRG zu Beginn, diese ist jedoch nicht mehr zwingend notwendig.

Testläufe, die unsere Version vom Inferenzalgorithmus aus [4] benutzten, erzielten für viele gängige Datenstrukturen wie einfach und doppelt verkettete Listen, binäre Bäume etc. als Resultat eine HRG, die der manuell erstellten Grammatik sehr ähnelte oder sogar vollständig gleich war.

Literatur

1. Annegret Habel: Hyperedge Replacement: Grammars and Languages. Springer-Verlag New York, Inc., 1992
2. Stefan Rieger und Thomas Noll: Abstracting Complex Data Structures by Hyperedge Replacement. Springer-Verlag, 2008
3. Joost Engelfriet, Linda Heyker und George Leih: Context-Free Graph Languages of Bounded Degree are Generated by Apex Graph Grammars. Acta Inf., Vol. 31, 1994
4. Eric Jeltsch and Hans-Jörg Kreowski: Grammatical Inference Based on Hyperedge Replacement. Graph-Grammars and Their Application to Computer Science, 461-474, 1990