Eine ökonomische statische Analysemethode zur Berechnung von Relational Attributes mittels regulärer Pfadbedingungen und ihre Anwendung auf Zeigeranalyse

Ahmed Mian Syed

Dissertation

zur Erlangung des Doktorgrades
der Naturwissenschaften
an der Fakultät für Mathematik und Informatik
der Universität Passau

Passau

28. Dezember 2003

Eine Dissertation zu schreiben bedeutet sich sowohl an den Grenze des wissenschaftlich Bekannten als auch seinen persönlichen Grenzen zu bewegen. Auf dieser interessanten und lehrreichen, aber auch oft enttäuschenden und beschwerlichen Reise war mir meine Frau Sandra eine sehr wichtige Begleiterin, die mir stets Rückhalt gegeben hat, eine unendliche Menge an Verständnis aufgebracht hat und auch für viele fachliche Diskussionen zur Verfügung gestanden ist. Ihr möchte ich an dieser Stelle sehr herzlich dafür danken.

Herrn Winfried Hahn möchte ich dafür meinen Dank aussprechen, daß er lange hinter mir gestanden hat, auch in den Zeiten als sich die ersten Ideen als gleichzeitig wissenschaftlich zu wenig ergiebig und trotzdem sehr aufwändig herausgestellt haben. Ebenso hat mir stets genügend zeitlichen und thematischen Freiraum gelassen, so daß ich letztendlich meine eigenen wissenschaftlichen Ideen verwirklichen konnte.

Eine weitere wichtige Rolle haben die "Dortmunder" Jens Knoop, Oliver Rüthing und Markus Müller-Olm gespielt, die mir sehr gute und wichtige Ratschläge zu Präsentation und Strategie meiner Arbeit gegeben haben und die mir gutes Feedback zu meiner Ausarbeitung gegeben haben. Auch ihnen möchte ich an dieser Stelle ganz herzlich danken.

Zusammenfassung

Mit steigender Abstraktionsebene von Systembeschreibungen wachsen die Anforderungen an Programmanalysetechniken, die im Idealfall herausfinden sollten, "was sich der Entwerfer dabei gedacht hat", um auf der Basis dieses Wissens die bestmögliche Implementierungsvariante zu finden. Speziell im Bereich des Hardware/Software Codesign, wo eine abstrakte Systemspezifikation durch Hardware– und Softwarekomponenten implementiert werden soll, ermöglichen genaue Analysetechniken erst die Implementierung von Teilen des Entwurfes in Hardware, bzw. besitzen ein großes Potential zur Einsparung von Hardwarekosten.

Dabei stoßen bereits bei der häufig benötigten Fragestellung der Zeigeranalyse alle bekannten Verfahren schnell an ihre Genauigkeitsgrenzen. Interessiert man sich für Ziele von Zeigervariablen, die tatsächlich in realen Programmläufen auftreten können, und läßt dabei Zeiger auf Zeiger zu, so findet man in der Literatur nur die Aussage, daß diese Fragestellung PSPACE–vollständig und damit eigentlich nicht praktisch beantwortbar ist. Eine Betrachtung der bekannten realisierten und prinzipiell denkbaren Zeigeranalysetechniken zeigt eine Gemeinsamkeit: alle diese Verfahren können kein ökonomisches Berechnungsprinzip realisieren, d.h. ein Berechnungsprinzip, bei dem nur soviel Aufwand investiert werden muß, wie minimal dazu benötigt wird. Vergleiche mit anderen Analysefragestellungen legen jedoch den Verdacht nahe, daß ein Verfahren mit der ambitionierten Zielsetzung von Exaktheit für seine praktische Einsetzbarkeit ökonomisch sein muß.

In dieser Arbeit wird daher eine Analysetechnik für Zeigeranalyse entwickelt, die exakt und ökonomisch ist. Zu diesem Zweck wird eine neue Art von Analyseprinzip eingeführt, und eine Theorie auf der Basis einer variablen Anzahl von parallel betriebenen endlichen Automaten vorgestellt, mit der man für die betrachtete Programmklasse exakte Analysen beschreiben und realisieren kann. Dabei werden potentiell unendliche Mengen von Programminstanzen als Bestandteil der Analyse verwendet, die jeweils durch von endlichen Automaten akzeptierte Sprachen dargestellt werden. Im Ausblick werden mögliche Erweiterungen des Verfahrens auf beliebige Eingabeprogramme diskutiert.

Als praktische Realisierung dieses theoretischen Prinzips wird eine spezielle Form von Erreichbarkeitsanalyse auf Graphen präsentiert, bei der im Graphen Bedingungen angetroffen werden, die die weitere Suche im Graphen einschränken. Die Graphen selbst werden durch eine neue Programmrepräsentation der zu analysierenden Eingabeprogramme gebildet.

Neben dem Nachweis der Ökonomie des Verfahrens wird anhand von experimentellen Messungen nachgewiesen, daß das Verfahren für die untersuchten realen Eingabeprogramme tatsächlich praktisch einsetzbar ist. Zusätzlich werden Gutartigkeitseigenschaften von Eingabeprogrammen vorgestellt. Die untersuchten realen Eingabeprogramme besitzen diese Gutartigkeitseigenschaften. Eine diesbezügliche Untersuchung der worst–case Eingabeprogramme aus der Literatur ergibt dagegen, daß diese die Gutartigkeitseigenschaften nicht besitzen. Interpretiert man die Gutartigkeitseigenschaften als charakteristische Eigenschaften von realen Eingabeprogrammen, was sich sowohl argumentativ als auch anhand der Untersuchungen belegen läßt, dann muß man für zu erwartende reale Analyseaufgaben nicht mit dem Eintreten des worst–case rechnen.

Für weniger gutartige Programme oder noch anspruchsvollere Fragestellungen, die im Ausblick vorgestellt werden, wird eine approximative Abwandlung des Analyseverfahrens vorgestellt, die im Gegensatz zu den Arbeiten aus der Literatur immer noch ein frei wählbares Maß an Mindestgenauigkeit der Ergebnisse garantieren kann.

Überblick

Das Ziel dieser Arbeit ist es, eine neue Programmanalysetechnik für Zeigeranalyse zu entwickeln. Diese soll exakt in dem Sinne sein, daß sie nur Ergebnisse berechnet, die tatsächlich in realen Programmläufen vorkommen können. Ebenso soll diese Analysetechnik ökonomisch sein, d.h. nur den minimal für eine exakte Lösung benötigten Berechnungsaufwand investieren müssen.

In Kapitel 1 wird ein Szenario aus dem Gebiet des Hardware/Software Codesign vorgestellt, in dem durch solche exakten Analysetechniken erhebliche Kosteneinsparungen bei der Hardwareimplementierung von Programmen, die Zeigervariablen verwenden, erzielt werden können.

Kapitel 2 gibt einen Überblick über den State of the Art. Dabei werden zwei unterschiedlich komplexe Analysezielsetzungen vorgestellt, von denen nur die aufwändigere in der Lage ist, exakte Lösungen im obigen Sinne zu berechnen. Die Verfahren aus der Literatur werden nach diesen beiden Zielsetzungen kategorisiert. Diejenigen aus der Literatur bekannten oder prinzipiell denkbaren Verfahren, die die aufwändigere Zielsetzung verfolgen, werden in eine Taxonomie eingeordnet. Dabei werden diese Verfahren daraufhin untersucht, ob sie ein ökonomisches Berechnungsprinzip erreichen können, d.h. ein Berechnungsprinzip, bei dem nur der minimal für eine exakte Lösung benötigte Berechnungsaufwand investiert werden muß.

Kapitel 3 stellt die genaue Zielsetzung dieser Arbeit vor, und legt die Klasse von zugelassenenen Eingabeprogrammen fest. Weiter wird ein Überblick über die Grundideen der Arbeit gegeben. Das vorgestellte Verfahren wird aus einer theoretischen und einer praktischen Sichtweise betrachtet werden. Zur Veranschaulichung des Verfahrens wird ein Running Example eingeführt.

Kapitel 4 beschreibt formal Eingabeprogramme aus der zugelassenen Programmklasse, und definiert deren Semantik.

Kapitel 5 beschreibt die theoretische Sichtweise des Verfahrens. Dazu wird eine Methode vorgestellt, wie man durch endliche Automaten Mengen von Programminstanzen zusammenfassend beschreiben kann, die eine bestimmte Anweisung des Eingabeprogrammes erreichen. Diese Beschreibungsform wird dazu verwendet, potentiell unendliche Mengen von Programminstanzen anzugeben, in denen jeweils eine bestimmte Programmeigenschaft wie z.B. "x zeigt auf y" gilt. Basierend auf mehreren solchen Kombinationen von Programminstanzenmengen und Programmeigenschaften wird eine Methode vorgestellt, wie man exakte Lösungen von Analysen mit der aufwändigeren der beiden oben vorgestellten möglichen Zielsetzungen erreichen kann. Von dieser Vorgehensweise wird auch eine approximative Variante vorgestellt.

Kapitel 6 stellt eine in dieser Arbeit verwendete neue Programmrepräsentation vor. Auf der Basis einer Transformationsfunktion von Eingabeprogrammen in der Darstellungsweise aus Kapitel 4 in die neue Programmrepräsentation, und einer erweiterten Semantikfunktion der neuen Programmrepräsentation, wird die Äquivalenz der beiden Darstellungsformen bewiesen. Für die neue Programmrepräsentation wird eine Interpretation als Analysegraphen vorgestellt.

Das Verfahren aus praktischer Sichtweise wird in Kapitel 7 vorgestellt. Dabei werden Queries auf dem Analysegraphen aus Kapitel 6 verarbeitet. Die wichtigsten Aspekte der Queryverarbeitung werden zunächst informell vorgestellt, und anschließend die Queryverarbeitung durch rekursive Queryfunktionen formal definiert. Auf der Basis dieser Queryfunktionen wird der Analysealgorithmus aus praktischer Sichtweise eingeführt.

Kapitel 8 befasst sich mit Korrektheitsüberlegungen für das vorgestellte Verfahren. Dabei wird bewiesen, daß in den von der Queryverarbeitung berechneten Mengen von Programminstanzen je-

weils die untersuchten Programmeigenschaften gelten. Des weiteren wird die Äquivalenz von theoretischer und praktischer Sichtweise des Verfahrens gezeigt, weshalb auch die praktische Sichtweise eine exakte Lösung für die angestrebte Zielsetzung liefert.

In Kapitel 9 wird bewiesen, daß das Verfahren maximal ökonomisch ist. Des weiteren werden Aussagen über maximale Analysekosten auf der Basis von theoretischen Obergrenzen und einer optimistischen Kostenschätzung unter Annahme von Gutartigkeitseigenschaften gemacht. Die aus der Literatur bekannten worst–case Eingabeprogramme werden kurz vorgestellt und ebenfalls auf diese Gutartigkeitseigenschaften untersucht.

Einige Aspekte der praktischen Realisierung des Verfahrens werden in Kapitel 10 diskutiert.

Kapitel 11 untersucht reale und konstruierte Eingabeprogramme, um Aussagen über das Auftreten von Gutartigkeitseigenschaften, und die praktische Anwendbarkeit des Verfahrens auf der Basis von Laufzeitabschätzungen zu machen.

Kapitel 12 fasst die Ergebnisse der Arbeit zusammen.

In Kapitel 13 wird ein Ausblick auf Erweiterungen des Verfahrens auf eine allgemeinere Klasse von Eingabeprogrammen, sowie auf weitere Analysefragestellungen gegeben.

Inhaltsverzeichnis

Kapitel 1

Motivation

Auf dem Anwendungsgebiet des Entwurfs und der Realisierung eingebetteter Systeme läßt sich in letzter Zeit ein klarer Trend zur Verwendung von High–Level Spezifikations– und Implementierungssprachen erkennen [dM99] [SN98]. Mit der damit einhergehenden zusätzlichen Ausdrucksmächtigkeit, die einem Entwerfer zur Verfügung steht, wachsen allerdings auch die Anforderungen an die Analysetechniken, mit denen man die technische Realisierung von solchen Beschreibungen optimieren oder überhaupt erst möglich machen kann.

Ein Forschungsgebiet, in dem diese Anforderungen besonders deutlich werden, ist das relativ junge Gebiet Hardware/Software Codesign [GM93] [CGJ+94] [DS95], bei dem ausgehend von einer abstrakten High–Level Spezifikation anhand von Analysen eine kostenbasierte Partitionierung des spezifizierten Systems in Hardware– und Softwarekomponenten geleistet werden soll. Dabei soll einerseits die Mächtigkeit der Spezifikationssprache gegenüber einer reinen Software–Beschreibungsform nicht zu sehr eingeschränkt werden, andererseits muß eine Realisierbarkeit zumindest eines Teils des Systems in Hardware stets gewährleistet bleiben.

Ein kritischer Teil der Analyse ist dabei die Untersuchung von potentiellen Zeigerzielen. Dabei ist eine Variable y Ziel der Zeigervariable x, wenn "x auf y zeigt", i.Z. $x \rightarrow y$. Da in verschiedenen Programmläufen eine Zeigervariable verschiedene Ziele besitzen kann, wird üblicherweise während der Analyse zu jeder Zeigervariable an jeder Anweisung die Menge von möglichen Zielen bestimmt. Diese Menge gilt dabei als sicheres Ergebnis, wenn bei Verwendung dieser Menge als Grundlage von z.B. einer Implementierung, deren Korrektheit in jedem Fall garantiert werden kann. Für die gängigen Anwendungen ist diese Menge dann ein sicheres Ergebnis, wenn sie eine Obermenge derjenigen Variablen ist, die in einem möglichen Programmlauf tatsächlich als Ziele der Zeigervariable auftreten können. Analysetechniken, die unsichere Ergebnisse produzieren könnten, werden prinzipiell nicht betrachtet.

Die Genauigkeit einer Zeigeranalysetechnik bei einem gegebenen Eingabeprogramm definiert sich dabei über die Anzahl der von der Analyse berechneten "ungenauen" potentiellen Zeigerziele für Zeigervariablen, die von der Analyse berechnet werden, die aber in keiner realen Programmausführung von diesen Zeigervariablen angenommen werden können. Beinhaltet die Ergebnismenge ausschließlich "genaue" Zeigerziele, so stellt dies ein exaktes Ergebnis dar. Ansonsten handelt es sich um approximative Ergebnisse.

Ein Beispiel zur Analyse von möglichen Zeigerzielen ist in Abbildung 1.1 dargestellt. Das Programmbeispiel links in der Abbildung weist in den Anweisungen vor Label L1 den Zeigervariablen x, y und z verschiedene Werte zu. Die Bedingung der darauffolgenden If–Anweisung wird durch Striche symbolisiert. Dies ist die in der Literatur übliche Notation, um auszudrücken, daß die Bedingungen von Verzweigungen nicht interpretiert werden, um das Problem von Nicht–Berechenbarkeit zu umgehen. Dadurch werden alle Pfade durch ein Programm, im Sinne einer Sequenz von ausgeführten Anweisungen bzw. Programmblöcken, als ausführbar angenommen. Für das Beispiel bedeutet dies also, daß eine Programminstanz, also ein möglicher Ablauf des Programmes, den Rumpf der If–Anweisung sowohl betreten als auch umgehen kann, ohne daß man hierfür Bedingungen angeben könnte. Im Beispiel gibt es demnach zwei mögliche Programmpfade vom Anfang des Programmes

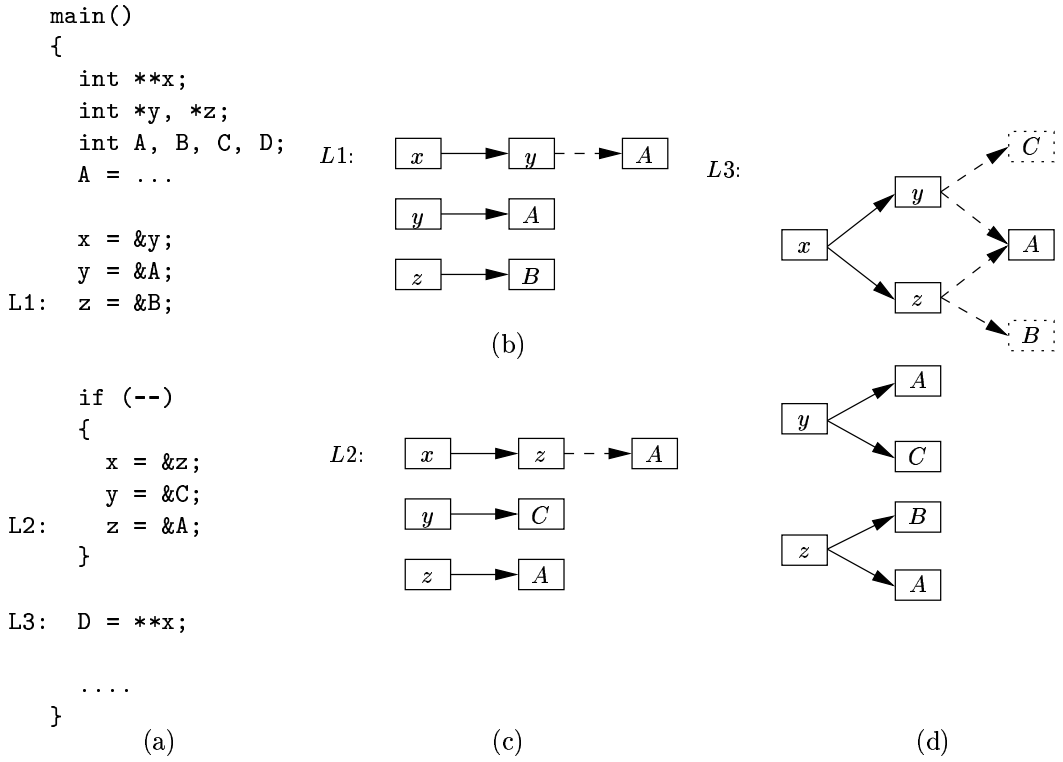


Abbildung 1.1: Motivationsbeispiel: Programmbeispiel (a) und Analyseinformation über mögliche Zeigerziele an Anweisungen L1 (b), L2 (c), sowie L3 (d).

bis zur Anweisung mit dem Label L3.

In Abbildung 1.1 (b) bis (d) sind die Ergebnisse einer gängigen Zeigeranalysemethode angegeben, wie sie z.B. in [ASU88] beschrieben wird. Mit dem Zeigerziel–Diagramm in Abb. 1.1(b) wird ausgesagt, daß bei Anweisung L1 die Zeigervariablen x, y und z auf die Variablen y, A bzw. B zeigen. Gestrichelt eingezeichnet ist das Ergebnis der Kombination der ersten beiden Analysefakten (also $x \rightarrow y$ und $y \rightarrow A$): die Variable, auf die diejenige Variable zeigt, auf die x zeigt, ist nach der Anweisung mit dem Label L1 stets die Variable A. Diese Art von Zugriff auf x stellt eine zweimalige Dereferenzierung dar und entspricht damit der Art des Zugriffes, wie er in der Zeile mit dem Label L3 geschieht (bei der Dereferenzierung einer Zeigervariable — hier mit dem $*$–Operator — wird auf das Ziel der Zeigervariable zugegriffen). Da dabei Zeigervariablen verschiedener sogenannter Referenzierungsstufen, in diesem Fall Zeiger auf Variablen vom Basistyp `int` und Zeiger auf Zeiger auf Variablen vom Basistyp `int`, vorkommen, bezeichnet man diese Art von Analyse auch als Multi–Level Zeigeranalyse.

Wenn nun der Rumpf der If–Anweisung nicht durchlaufen wird, so erreichen diese Zeigerinhalte unverändert die Anweisung L3. Daher wird dort aufgrund dieser Zeigervariablenbelegung der Variablen D der Inhalt der Variable A zugewiesen. Die Situation, die sich bei Durchlaufen des Rumpfes der If–Anweisung bei der Anweisung mit dem Label L2 ergibt, ist in Abbildung 1.1(c) darunter dargestellt. Obwohl die einzelnen Zeigervariablen dort unterschiedlichen Inhalt bezogen auf die Situation bei Anweisung L1 besitzen, ergibt sich als Ergebnis der zweifachen Dereferenzierung der Variablen x wieder die Variable A, was ebenfalls gestrichelt eingezeichnet ist. Demnach wird auch im Falle des Durchlaufens des Rumpfes der If–Anweisung bei Ausführung der Anweisung mit Label L3 der Variablen D der Inhalt der Variable A zugewiesen. Aufgrund dieser Beobachtung könnte man also die Anweisung mit Label L3 durch die Anweisung `D=A` ersetzen, da auf jedem möglichen Pfad durch das Programm die zweimalige Dereferenzierung von x die Variable A ergeben muß.

Die bisherige Argumentation betrachtete für jeden möglichen Programmpfad, welchen Inhalt die

Variablen nach dessen Durchlaufen besitzen, und was dies für die zweimalige Dereferenzierung der Variablen x bei der Anweisung mit dem Label L3 für Auswirkungen hat. Die Menge aller Pfade ist i.A. aber unendlich groß, sobald Schleifen im Programm vorkommen. Damit kann man algorithmisch diese Betrachtung jedes einzelnen Pfades nicht mehr realisieren. Stattdessen werden in den Analysetechniken aus der Literatur Zusammenfassungen von Analysefakten gebildet, bei denen die exakte Zuordnung von Pfaden zu den auf diesen Pfaden berechneten Ergebnissen verloren gehen.

In Abbildung 1.1(d) ist das Ergebnis der oben angedeuteten gängigen Zeigeranalysemethode dargestellt, die solche Zusammenfassungen bildet. Die beiden möglichen Pfade durch das Programm fließen bei der Zeile mit dem Label L3 wieder zusammen, indem sowohl eine Programminstanz, die den then–Zweig betritt, als auch eine Programminstanz, die diesen umgeht, an dieser Stelle ihre Ausführung fortsetzen. Dieses Zusammenfließen von Pfaden wird in der Literatur als Kontrollfluß–Join bezeichnet.

Dabei werden für jede Variable die Ergebnisse der Analyse, die auf diesen beiden Pfaden erzielt wurden, vereinigt. So kann z.B. x nach der If–Anweisung auf y oder auf z zeigen, y auf A oder C und z auf A oder B. Kombiniert man anhand dieses (exakten) Ergebnisses nun wieder die verschiedenen Fakten, um das Ziel der zweimaligen Dereferenzierung von x zu berechnen, so ergibt sich hier ein zur obigen Betrachtung unterschiedliches Ergebnis, wiederum gestrichelt eingezeichnet. Die Menge der möglichen Ziele der zweimaligen Dereferenzierung von x wird hier als die Menge der Variablen A, B und C berechnet. Gemäß obiger Argumentation ist dies immer noch ein sicheres Ergebnis, aber es ist nicht mehr exakt, da in keinem der beiden möglichen Programmabläufe (mit Betreten bzw. Umgehen des Rumpfes der If–Anweisung) die zweimalige Dereferenzierung der Variablen x die Variablen B oder C ergeben kann. Offensichtlich führen nicht alle Kombinationen von Analysefakten bei Anweisung L3 zu "genauen" Ergebnissen. An dieser Stelle soll zunächst auf die Auswirkungen dieser Ungenauigkeit im skizzierten Szenario eingegangen werden. Die theoretischen Hintergründe davon werden im nächsten Kapitel diskutiert.

Während bei der Zeigeranalyse im Rahmen von reinen Software–Compilern mehr das Problem des Aliasing im Vordergrund steht, bei dem manche Optimierungen z.B. nur dann vorgenommen werden können, wenn sicher ist, daß unterschiedliche Anweisungen stets verschiedene Daten beeinflussen, hat eine Zeigeranalyse im Anwendungsgebiet des Hardware/Software Codesign die Aufgabe, zur Laufzeit durch Zeigerzugriffe dynamisch entstehende Zugriffe auf Daten in eine statische Hardware–Struktur abzubilden, die für jeden möglichen Zugriff auf Variablen durch Zeiger fest verdrahtete Kommunikationsmöglichkeiten bereitstellt.

Abbildung 1.2 (a) zeigt nun die verschiedenen Hardware–Implementierungen des Beispielprogrammes gemäß des Verfahrens aus [SDM98], zunächst anhand des sicheren Ergebnisses der Analyse, daß die zweimalige Dereferenzierung der Variablen x die Variablen A, B oder C ergeben kann. Dabei wurde hier nur die Anweisung mit dem Label L3 umgesetzt und dadurch auf die Verwendung eines Kontrollautomaten verzichtet.

Je nach Belegung der Zeigervariablen muß dazu durch die Funktion f der Steuerungseingang des Multiplexers (MUX) so geschaltet werden, daß der Inhalt derjenigen Variablen, die der zweimaligen Dereferenzierung von x entspricht, am Ausgang des Multiplexers zur Zuweisung an die Variable D zur Verfügung steht. Eine entsprechende Realisierung solch einer Funktion f ist ebenfalls in der Abbildung angegeben.

In Abbildung 1.2 (b) ist die Hardware–Realisierung anhand des exakten Ergebnisses gezeigt, das durch Betrachtung der Ergebnisse auf jedem einzelnen Pfad durch das Programm gefunden wurde. Dabei entfällt für die Zuweisung der Multiplexer und die dazugehörige Ansteuerungslogik, was eine erhebliche Kostenreduktion im Sinne von Chipfläche bedeutet. Abbildung 1.2 (c) stellt die Situation dar, in der im weiteren Verlauf des Programmes (im Beispielprogramm mit ... angedeutet) die Variable D nur ein Alias für A darstellt, und daher die Hardware–Realisierung durch ein einziges Register mit Ausgangsleitungen für die entsprechenden Rollen von A geleistet werden kann.

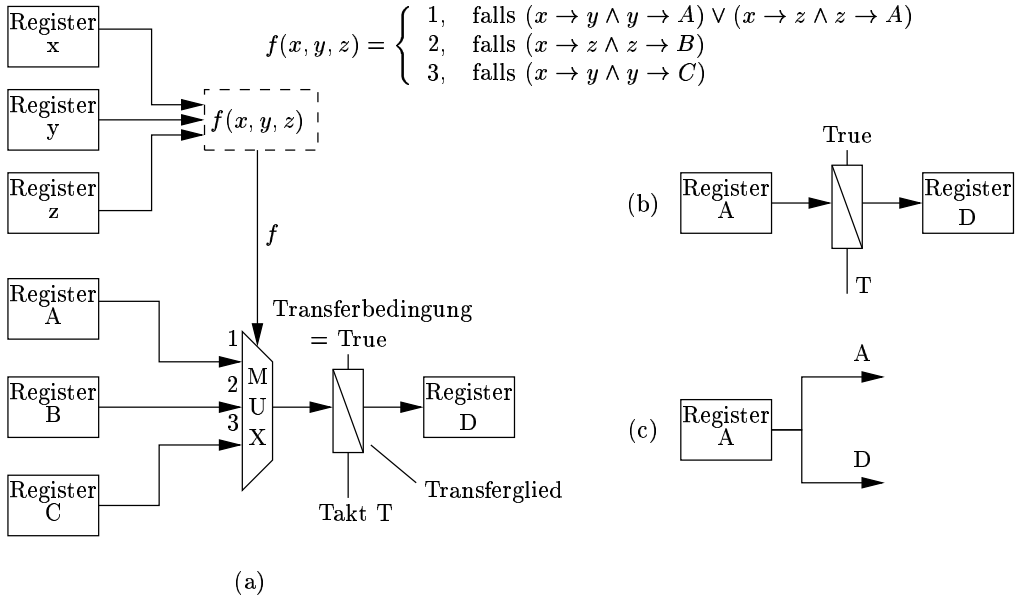


Abbildung 1.2: Hardware–Realisierungen des Motivations–Beispiels anhand der verschiedenen Analyseergebnisse.

Kapitel 2

State of the Art

Prinzipiell unterscheidet man zwei Arten von Programmanalyse: statische und dynamische Programmanalyse. Dabei unterscheiden sich die beiden Kategorien dadurch, daß bei statischer Analyse der Quellcode eines zu analysierenden Programmes betrachtet wird, während bei dynamischer Analyse Laufzeitbeobachtungen verwendet werden, um Informationen über mögliches Laufzeitverhalten von Programmen zu gewinnen. Da man aus Laufzeitinformationen aber keine allgemeinen Schlüsse über das Verhalten des Programmes in den nicht beobachteten Fällen ziehen kann, liefert die Klasse der statischen Analysetechniken i.A. die genaueren Ergebnisse über das prinzipiell mögliche Laufzeitverhalten eines Programmes. Die Analysetechniken, die in dieser Arbeit betrachtet werden, entstammen alle aus dem Gebiet der statischen Programmanalyse.

Obwohl im Programmbeispiel in der Motivation die durch die angegebene Analysetechnik berechneten möglichen Zeigerziele für x, y und z bei Anweisung L3 exakte Ergebnisse in dem Sinne sind, daß es mögliche Programmläufe gibt, in denen die Zeigervariablen diese Ziele besitzen, gilt dies offensichtlich nicht für alle Kombinationen von Zeigerzielen. Dies liegt anschaulich darin begründet, daß es keinen möglichen Pfad durch das Programm gibt, bei dessen Durchlaufen die von der Analyse kombinierten Fakten, also hier die Zeigerzielinformationen, tatsächlich gemeinsam auftreten können. Im Beispiel zeigt z.B. x nur bei Umgehen des Rumpfes der If–Anweisung auf y, während nur bei Durchlaufen des Rumpfes y auf C zeigen wird. Da eine Programminstanz (also ein möglicher Ablauf des Programmes) nur einen dieser beiden Pfade durchlaufen kann, werden diese beiden Zeigerziel–Eigenschaften nie gemeinsam auftreten. Daher ist die Kombination der Analysefakten $x \to y$ und $y \to C$ in Abbildung 1.1 eine *ungültige Kombination* von Analysefakten.

2.1 Analysefragestellungen

Dieses grundsätzliche Problem wird von Jones und Muchnick in [MJ81a] am Beispiel eines Type–Checking Algorithmus beschrieben. Sie führen daraufhin zwei prinzipielle Klassen von Analysefragestellungen ein, die Analyse von *independent attributes* und die Analyse von *relational attributes*. Diese Unterscheidung wird von Muth und Debray [MD00] erneut zum Gegenstand wissenschaftlicher Untersuchungen gemacht. Dabei vereinheitlichen sie verschiedene, sehr unterschiedliche Komplexitätsergebnisse aus der Literatur über verschiedene Analyseproblemstellungen in der Form, daß sie die dabei jeweils untersuchten Problemstellungen in die von Jones und Muchnick vorgestellten Klassen einteilen.

2.1.1 Independent Attributes

Bei der Analyse von *independent attributes* werden einzelne Analysefakten, dies wären für Zeigeranalyse z.B. mögliche Zeigerziele von einzelnen Variablen, unabhängig voneinander berechnet, ohne daß Aussagen über deren Zusammenhang, wie z.B. gemeinsames Auftreten auf Pfaden, getroffen werden. Dies entspricht der Berechnungsmethode des Beispiels, in dem z.B. die Analysefakten $x \to y$ und

$y \to C$ als voneinander unabhängig, und daher auch sämtliche Kombinationen dieser Fakten als gültig angesehen werden.

Der Großteil der in der Literatur vorgestellten Zeigeranalysetechniken interpretiert verschiedene potentielle Zeigerziele als *independent attributes*, weswegen keines von diesen Verfahren für das Beispiel ein genaueres als das vorgestellte approximative Ergebnis liefern kann. Die Schwerpunkte dieser Verfahren liegen in ihrer Behandlung von interprozeduralen Pfaden, d.h. Pfaden, die durch Aufrufe von Funktionen und die Rückkehr des Kontrollflusses zu den entsprechenden Aufrufstellen entstehen. Diese werden dabei zur Vermeidung des *invalid path problems* [LR91] benötigt, bei dem Analysefakten fälschlicherweise entlang von Programmpfaden propagiert werden, bei denen der Rücksprung nach Abarbeitung einer Funktion nicht zur Aufrufstelle erfolgt. Dazu gehören die Verfahren [Wei80] [LH88] [Gua88] [RM88] [HPR89] [CWZ90] [Deu90] [LR91] [LR92] [Lan92b] [BC92] [Deu92] [HHN92] [Ema93] [CBC93] [LRZ93] [And94] [Deu94] [EGH94] [BCCH94] [Ruf95] [PR95] [WL95] [GH96a] [GH96b] [AG98] [Ste96a] [SH97] [GH98] [RS98] [SRW98] [HH98] [YHR99] [FFA00] [FRD00]. Spezielle anforderungsgetriebene Berechnungsmethoden von *independent attributes* werden in [Rep94] [DGS95] [HRS95] vorgestellt.

Alle diese Arbeiten sind in Bezug auf die in der Motivation vorgestellte Zielsetzung, ungültige Kombinationen von Analysefakten zu vermeiden, nicht exakt, und damit für das Erreichen dieses Ziels nicht geeignet.

2.1.2 Relational Attributes

Bei der Analyse von *relational attributes* werden dagegen nicht nur einzelne Analysefakten, sondern auch deren Beziehungen untereinander betrachtet. Im Beispiel entspräche dies z.B. Analysefakten der Form $x \to y \wedge y \to A$. Solche Fakten werden im weiteren als kombinierte Analysefakten oder Kombinationen von Analysefakten bezeichnet werden. Wie im nächsten Abschnitt gezeigt werden wird, fällt diese Art von Analyse in eine deutlich schwierigere Problemklasse.

Zur Vermeidung von ungültigen Kombinationen müssen implizit oder explizit auch intraprozedurale Pfade berücksichtigt werden, wie bereits im Motivationsbeispiel gezeigt wurde. Während interprozedurale Aspekte der Analyse bereits von vielen Arbeiten betrachtet werden, stellt die intraprozedurale Analyse einen wesentlichen Unterschied zu den meisten in der Literatur vorgestellten Verfahren, und daher den Hauptaspekt von Arbeiten mit dem Ziel der Berechnung von *relational attributes* dar. Konkret ist dies für Konstantenanalyse die Arbeit von Steffen und Knoop [SK91], sowie für Multi–Level Zeigeranalyse die Arbeit von Burke et al. [BCCH97]. Prinzipiell denkbar wäre auch die Anwendung einer Methode, die auf Steffen [Ste96b] zurückgeht. Nach der Einordnung von Muth und Debray [MD00] ist bei diesen beiden Analysearten für eine exakte Lösung eine Analyse von *relational attributes* notwendig. All diesen Arbeiten ist gemeinsam, daß sie für deren Berechnung in Vereinfachung des Problems keine interprozeduralen Pfade für den Test auf die Kombinierbarkeit von Fakten betrachten. Während in [SK91] und [Ste96b] von einer Eingabesprache ohne Funktionsaufrufe ausgegangen wird, wird in [BCCH97] eine approximative Betrachtung von intraprozeduralen Pfaden zu einer traditionellen interprozeduralen Analyse von *independent attributes* hinzugefügt. Zu einer Beschränkung der Kombinationen von Analysefakten wird dort aber nur die Approximation der intraprozeduralen Pfade herangezogen, eine Test auf gemeinsame interprozedurale Pfade fehlt hierbei jedoch.

Eine genaue Einordnung dieser Algorithmen in eine Klassifizierung nach ihren Berechnungsprinzipien wird in Abschnitt 2.5 angegeben werden.

2.2 Komplexität der verschiedenen Analysefragestellungen

2.2.1 Berechnung einer Analyse von Independent Attributes

Für die konkrete Analyse von Zeigerzielen für Programme mit Single–Level–Zeigern, d.h. Programmen, in denen nur Zeiger auf Variablen von einem skalaren Basistyp vorkommen dürfen, hat Landi in [Lan92a] gezeigt, daß eine exakte Lösung in polynomieller Zeit möglich ist. Dieses Resultat wird

von Muth und Debray in [MD00] dahingehend eingeordnet, daß unter diesen Voraussetzungen eine Analyse von *independent attributes* für ein exaktes Ergebnis ausreicht. Allgemein liefern unter dieser Voraussetzung die bekannten Iterationsalgorithmen aus der Theorie der klassischen Datenflußanalyse [Kil73] [KU77] ein exaktes Ergebnis in polynomieller Zeit. Neben der Single–Level–Zeigeranalyse erfüllen auch noch andere Analysetechniken wie "Lebendige Variablen" oder "Verfügbare Ausdrücke" diese Voraussetzung.

Benötigt eine Analyseart allerdings grundsätzlich *relational attributes* zur Exaktheit, so sind diese Iterationsverfahren zwar prinzipiell auch anwendbar, sie liefern aber nur approximative, wenngleich sichere Ergebnisse. Die Berechnung aus dem Motivationsbeispiel, bei der eine zu große Ergebnismenge berechnet wird, ist ein Beispiel für die Anwendung solch eines Verfahrens auf das Problem der Multi–Level–Zeigeranalyse, für dessen exakte Lösung eine Analyse von *relational attributes* notwendig ist.

2.2.2 Berechnung einer Analyse von Relational Attributes

Muth und Debray zeigen in [MD00] ebenfalls, daß die Analyse von *relational attributes* allgemein PSPACE–vollständig ist, selbst wenn Verzweigungsbedingungen nicht interpretiert werden, und daher alle Pfade als ausführbar angenommen werden, und man keine Funktionsaufrufe oder (Zeiger–) Arithmetik zuläßt. Sie argumentieren anhand eines zum Motivationsbeispiel ähnlichen Programmfragmentes, daß die konkrete Analyse von Zeigerzielen, wenn man wie im Beispiel Multi–Level Zeiger zuläßt, eine Analyse von *relational attributes* benötigt. Das gleiche wird für die Analyseart der Analyse von *may–constants* (ohne arithmetische Operationen) gezeigt, bei der ermittelt werden soll, ob eine Variablen in einer möglichen Programminstanz einen bestimmten Wert annehmen kann. Damit fällt die Problemstellung, allgemein wie im Beispiel vorgestellte *ungültige Kombinationen* von Analysefakten zu vermeiden, in die Kategorie der PSPACE–vollständigen Probleme.

2.3 Allgemeines Framework für Analysefaktenmodellierung

Die in dieser Arbeit vorgestellten verschiedenen Methoden, um Analysen von *relational attributes* zu berechnen, sollen in ein allgemeines Framework eingeordnet werden, bei dem die Modellierung der Analysefakten betrachtet wird. Das Ziel ist dabei nicht eine absolute Vergleichbarkeit der verschiedenen Verfahren. Es wird sich zeigen, daß mit dem Analyseprinzip und der Art der Problemmodellierung durch Analysefakten für jedes konkrete Verfahren eine Menge von Parametern aus diesem Framework derart unterschiedlich festgelegt werden, daß keine absoluten Vergleiche möglich sind. Stattdessen ist das Framework als Mittel zum qualitativen Vergleich der Verfahren, und zu deren einheitlicher Beschreibung gedacht. Damit wird es ermöglicht, verfahrensübergreifend Aussagen über die minimale Menge von für eine exakte Lösung benötigten Analysefakten zu treffen. Die eigentlichen Analyseverfahren werden in das vorgestellten Framework nicht explizit mit einbezogen. Da die meisten Verfahren aus der Literatur keine Analysen von *relational attributes* zum Ziel haben, und damit auch nicht in das Framework einordenbar sind, könnten ohnehin nur wenige Verfahren, die sich zudem als nicht praxisrelevant herausstellen werden, anhand des Frameworks beschrieben werden.

2.3.1 Definition

Die folgende Definition legt das allgemeine Framework zur Einordnung von Analyseverfahren für *relational attributes* anhand der von ihnen verwendeten Analysefaktenmodellierung fest.

Definition 2.1 (Framework für endliche Analysefaktenmodellierung von *rel. attr.*)
Ein allgemeines Framework für endliche Analysefaktenmodellierung von relational attributes *sei definiert als ein Tupel* $(\mathbf{G}, \mathbf{F}, \mathbf{K}, \otimes, \Delta, \Theta)$ *mit*

$\mathbf{G}$	*einer Menge von Anweisungen oder sonstigen Programmbestandteilen, an denen ein Analysefaktum jeweils Gültigkeit hat,*
$\mathbf{F}$	*einer endlichen Menge von Analysefakten, z.B. von der Form* $x \to y$,
$\mathbf{K}$	*einer endlichen Menge von Kombinationen von Analysefakten mit* $\mathbf{F} \subseteq \mathbf{K}$, *z.B. von der Form* $x \to y \wedge y \to A$,
$\otimes : \mathbf{F} \times \mathbf{K} \to \mathbf{K}$	*einer Kombinationsabbildung von Analysefakten,*
$\Delta \subseteq \mathbf{G} \times \mathcal{P}(\mathbf{K})$	*der bezüglich Mengeninklusion minimalen Menge von Kombinationen von Analysefakten, die für eine exakte Lösung einer Analyse von* relational attributes *benötigt werden,*
$\Theta \subseteq \mathbf{G} \times \mathcal{P}(\mathbf{K})$	*mit* $\Delta \subseteq \Theta$ *der Menge der während einer Analyse entstehenden Kombinationen von Analysefakten, wenn keine Beschränkung auf die Menge* Δ *vorgenommen wird.*

Nimmt man die Menge $\mathbf{G}$ als eine Menge von Anweisungen eines Programmes an, so besitzt jedes einzelne Analysefaktum $f \in \mathbf{F}$ bzw. jede Kombination von Analysefakten $k \in \mathbf{K}$ an einer bestimmten Anweisung aus $\mathbf{G}$ Gültigkeit. Um die Darstellung nicht zu verkomplizieren, wird diese Zuordnung meist implizit geschehen. Allgemein ließe sich ein Analysefaktum als partielle Abbildung von der Menge $\mathbf{G}$ in die Menge $\mathbf{F}$ bzw. in die Menge $\mathbf{K}$ interpretieren. Die Menge aller Analysefakten bzw. deren Kombinationen, die damit an einer Anweisung gelten, wäre entsprechend als eine Abbildung von der Menge $\mathbf{G}$ in die Menge $\mathcal{P}(\mathbf{F})$ bzw. $\mathcal{P}(\mathbf{K})$ zu interpretieren, wobei $\mathcal{P}$ die Potenzmenge der jeweiligen Menge in Klammern bezeichnen soll.

2.3.2 Einordnung des Motivationsbeispiels in das Framework

Im Motivationsbeispiel wurde implizit ein Analyseverfahren vorgestellt, bei dem Mengen von Zeigerzielen als Analysefakten verwendet wurden, um ein exaktes Ergebnis der Analyse von *relational attributes* zu berechnen. Dieses Verfahren soll in das Framework aus Definition 2.1 eingeordnet werden, um dieses beispielhaft zu verdeutlichen, und um im Folgenden die Einführung eines Bewertungskriteriums für Verfahren aus der Literatur zu motivieren. Diese Einordnung nimmt damit die Details aus Abschnitt 2.5.1.1 der Taxonomie von Verfahren vorweg.

Im Folgenden bezeichne A die Menge der Anweisungen aus einem Eingabeprogramm, und V die Menge der Variablen dieses Programmes.

2.3.2.1 Gültigkeitsbereich von Analysefakten

Die Menge $\mathbf{G}$, die die Programmbestandteile beschreibt, an denen einzelne Analysefakten gültig sein können, entspricht im in der Motivation vorgestellten Analyseverfahren der Menge A von Anweisungen. Für jede Anweisung wird dabei eine Menge von Zeigerzielen berechnet, die an dieser Anweisung (möglicherweise) angenommen werden.

2.3.2.2 Analysefakten

Für das vorgestellte Analyseverfahren entspricht ein einzelnes Analysefaktum, das an einer bestimmten Anweisung $a \in \mathbf{G} = A$ Gültigkeit besitzt, einer Aussage der Form $x \to y$ mit $x, y \in V$, d.h. "x zeigt auf y". Damit kann man für das vorgestellte Verfahren festlegen:

$$\mathbf{F} = \{x \to y \mid x, y \in V\}$$

Diese Menge von Analysefakten ist aufgrund der endlichen Anzahl von Variablen in einem Eingabeprogramm stets endlich.

2.3.2.3 Kombinationen von Analysefakten

Eine Kombination von Analysefakten entspricht einer konjunktiven Verknüpfung von einzelnen Analysefakten, d.h. man kann definieren:

$$\mathbf{K} = \{x_1 \to y_1 \wedge \ldots \wedge x_n \to y_n \mid n \in \mathbb{N}, x_1, \ldots, x_n, y_1, \ldots, y_n \in V\}$$

Auch diese Menge von Kombinationen von Analysefakten ist stets endlich.

2.3.2.4 Kombinationsabbildung von Analysefakten

Die Kombinationsabbildung $\otimes : \mathbf{F} \times \mathbf{K} \rightarrow \mathbf{K}$ fügt einzelne Fakten zu Konjunktionen von Analysefakten hinzu, d.h. es gilt z.B. $x \rightarrow y \otimes y \rightarrow A \wedge A \rightarrow C = x \rightarrow y \wedge y \rightarrow A \wedge A \rightarrow C$.

2.3.2.5 Menge Δ von Kombinationen

Die Menge Δ wurde in Definition 2.1 als die minimale Menge von Kombinationen von Analysefakten definiert, die zu einer exakten Lösung einer Analyse von *relational attributes* benötigt werden. Im Motivationsbeispiel müssen dafür an jeder Anweisung, an der eine Zeigervariable (mehrfach) dereferenziert wird, die Ziele aller Dereferenzierungen in der Kombination enthalten sein. Betrachtet man z.B. die Anweisung `D=**x` im Motivationsbeispiel, die im Folgenden nur mit $a \in A$ bezeichnet werden soll, so muß in jeder Kombination von Analysefakten das Zeigerziel von `x`, sowie derjenigen Variablen, die sich als dieses Zeigerziel ergibt, enthalten sein. Nach der Berechnung aus dem Motivationsbeispiel ergibt sich damit Δ wie folgt:

$$\Delta = \{(a, \{x \rightarrow y \wedge y \rightarrow A, x \rightarrow z \wedge z \rightarrow A\})\}$$

2.3.2.6 Menge Θ von Kombinationen

Analog bestimmt man die Menge Θ von sämtlichen Kombinationen von Analysefakten, die bei einer Analyse berechnet werden, bei der keine Beschränkung auf die Menge der tatsächlich benötigten Kombinationen Δ vorgenommen wird. Nach der Berechnung aus dem Motivationsbeispiel ergibt sich damit Θ wie folgt:

$$\Theta = \{(a, \{x \rightarrow y \wedge y \rightarrow A \wedge z \rightarrow B, x \rightarrow z \wedge y \rightarrow C \wedge z \rightarrow A\})\}$$

Im Motivationsbeispiel besteht kein Unterschied zwischen den Größen der Mengen Δ und Θ, beide enthalten jeweils zwei Kombinationen von Analysefakten. In einem größeren Programmbeispiel wären aber auch die folgenden Kombinationen Θ' von Analysefakten als Ergebnis der Analyse denkbar. In diesem Fall wäre ein signifikanter Größenunterschied zu bemerken zwischen der Menge Δ, die auch in diesem Fall identisch zur oben angegebenen Menge wäre, und der Menge Θ'.

$$\begin{aligned}\Theta' = \{(a, \{&x \rightarrow y \wedge y \rightarrow A \wedge z \rightarrow A,\\ &x \rightarrow y \wedge y \rightarrow A \wedge z \rightarrow B,\\ &x \rightarrow y \wedge y \rightarrow A \wedge z \rightarrow C,\\ &x \rightarrow z \wedge y \rightarrow A \wedge z \rightarrow A,\\ &x \rightarrow z \wedge y \rightarrow B \wedge z \rightarrow A,\\ &x \rightarrow z \wedge y \rightarrow C \wedge z \rightarrow A\ \})\}\end{aligned}$$

2.4 Bewertungskriterium für Verfahren

Die Verfahren aus der Literatur, sowie das in dieser Arbeit vorgestellte neue Verfahren, sollen anhand eines einheitlichen Bewertungskriteriums untersucht werden.

2.4.1 Ökonomie von Verfahren

Dazu werden diese Verfahren, sofern sie in der Lage sind, ein exaktes Ergebnis zu berechnen, in das allgemeine Framework aus Definition 2.1 eingeordnet. Damit kann für jedes solche Verfahren die minimale Menge Δ von benötigten Kombinationen von Analysefakten betrachtet werden. Die Bewertung der verschiedenen Verfahren wird dann dahingehen erfolgen, daß untersucht wird, ob es möglich ist, tatsächlich nur diese Menge Δ von Kombinationen von Analysefakten, anstatt stets die Menge Θ zu berechnen.

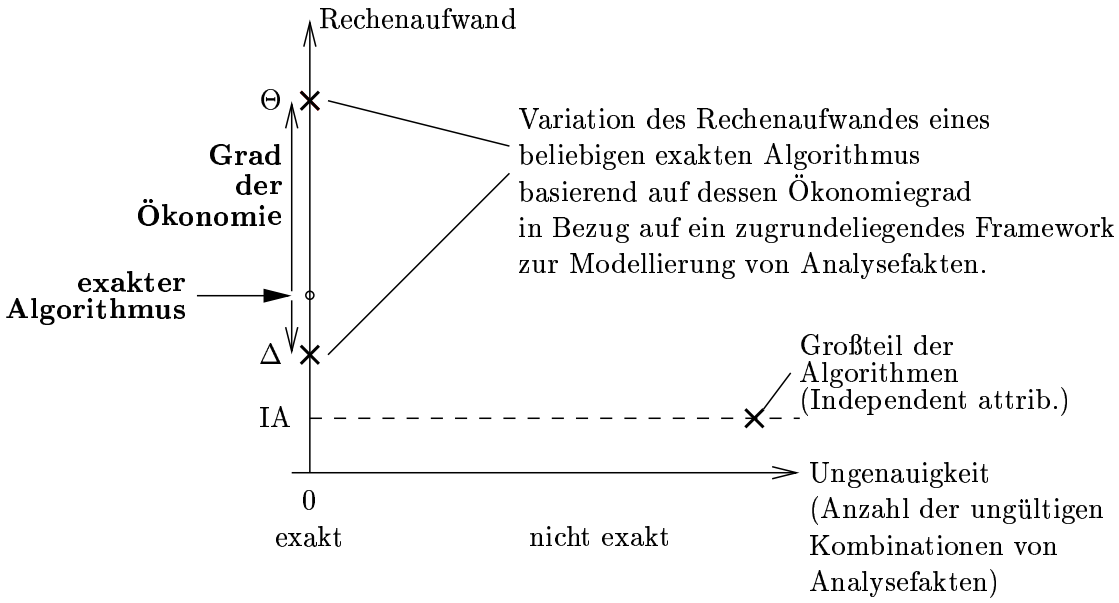


Abbildung 2.1: Mögliche Einordnungen von Algorithmen nach Rechenaufwand und Exaktheit.

Definition 2.2 (Ökonomie)
Ein Analyseverfahren, das in das Framework aus Definition 2.1 eingeordnet wurde, werde als ökonomisch *bezeichnet, wenn es nur die minimale Menge Δ von Kombinationen von Analysefakten berechnen muß, die zu einer exakten Lösung notwendig sind.*

Mit der Bezeichnung aus dieser Definition werden die Verfahren also daraufhin untersucht, ob sie ökonomisch sind. Dabei ist zu bemerken, daß ein absoluter Vergleich der Mengen Δ und Θ von verschiedenen Verfahren i.A. nicht möglich ist, da die Analysefaktenmodellierung von verschiedenen Analyseverfahren normalerweise zu unterschiedlich sind um eine gemeinsame Modellierungssweise erkennen zu lassen anhand derer man Vergleiche anstellen könnte. Das Prinzip der Ökonomie ist daher eine rein lokale Eigenschaft eines einzelnen Verfahrens. Daß dieser Begriff dennoch Aussagekraft über Verfahren besitzt, erkennt man an den Abschätzungen über die Mengen Δ und Θ in späteren Kapiteln, sowie an den offensichtlich nicht praktisch realisierbaren möglichen nicht–ökonomischen exakten Verfahren die in diesem Kapitel vorgestellt werden.

2.4.2 Mögliche Einordnungen von Verfahren

In Abbildung 2.1 ist eine Einordnung von Algorithmen nach Rechenaufwand und Exaktheit skizziert. Dabei wird in horizontaler Richtung die Exaktheit einer Analysetechnik in Bezug auf die Betrachtung von *relational attributes*, gemessen durch die Größe der von der Analyse berechneten Menge von ungültigen Kombinationen von Analysefakten, aufgetragen. Die eingezeichnete y–Achse entspricht dabei einem exakten Ergebnis, während die Genauigkeit einer Analyse mit zunehmendem Abstand von der y–Achse abnimmt. In vertikaler Richtung ist der zu erwartende Rechenaufwand eines Algorithmus dargestellt. Dies ist keine allgemeine Einordnung bezüglich der Komplexitätshierarchie, sondern nur eine relative Aussage für Verfahren die auf dem gleichen Framework zur Analysefaktenmodellierung basieren. Je weniger zusätzlichen Rechenaufwand gegenüber der Berechnung von Δ ein Verfahren dabei investieren muß, desto geringer ist der zu erwartende Rechenaufwand. Dabei

wird davon ausgegangen, daß der Rechenaufwand mit der Größe der zu berechnenden Menge von Kombinationen von Analysefakten ansteigt.

Die x–Achse entspricht dabei dem minimal möglichen Rechenaufwand aller möglichen Verfahren, man kann in konstanter Zeit die Aussage treffen, daß alle Variablen auf alle anderen Variablen zeigen können, was aber sicherlich keine besonders sinnvolle Analysemethode darstellen würde. Als nächstes ist die Kategorie der Algorithmen (mit garantiert polynomiellem Rechenaufwand) eingezeichnet, die das Problem, ggfs. vereinfachend, als Analyse von *independent attributes* (im Diagramm mit IA bezeichnet) betrachten. Da hierbei keine Kombinationen von Analysefakten gebildet werden müssen, ist der zu erwartende Rechenaufwand mit Sicherheit geringer als der von Verfahren die demgegenüber erhöhte Genauigkeit erzielen wollen. Als maximaler Rechenaufwand ist die Berechnung der Menge Θ eingezeichnet. Hierfür lassen sich keine allgemeinen Aussagen für alle möglichen Eingabeprogramme machen, da der entsprechende Berechnungsaufwand für einzelne Eingabeprogramme zwischen voraussichtlich exponentiell (die Analysefragestellung ist allgemein in der Klasse der PSPACE–vollständigen Probleme einzuordnen) und polynomiell mit der Programmgröße ansteigendem Berechnungsaufwand variieren kann. Klar ist jedoch, daß ein nicht–ökonomisches Verfahren auf der Basis der gleichen Modellierung von Analyesefakten mehr Rechenaufwand investieren muß als ein ökonomisches.

Ebenfalls angetragen ist der Rechenaufwand zur Berechnung der Menge Δ von Kombinationen, die für ein exaktes Ergebnis unbedingt berechnet werden müssen. Der dazu benötigte Rechenaufwand ist nach den Voraussetzungen aus Definition 2.1 geringer oder gleich dem zur Berechnung von Θ. Damit wird dieser auf der y–Achse unterhalb desjenigen zur Berechnung von Θ angetragen. Auch für Δ läßt sich keine allgemeine Einordnung in der üblichen Komplexitätshierarchie angeben. Da die Menge von Kombinationen von Analysefakten die Menge der einzelnen Analysefakten größenmäßig übersteigt, kann man den Berechnungsaufwand zur Berechnung von Δ als größer oder gleich dem zur Berechnung einer Analyse von *independent attributes* einordnen. Damit ist ersterer oberhalb letzterem im Diagramm eingezeichnet.

Ein exaktes Verfahren muß damit auf der y–Achse zwischen den Punkten Δ und Θ einzuordnen sein, wobei der Grad der Ökonomie ansteigt, je näher der investierte Rechenaufwand am Minimalaufwand Δ liegt.

Der Großteil der vorgestellten Verfahren ist an der Markierung rechts im Diagramm einzuordnen. Diese fallen in die Kategorie einer Analyse von *independent attributes*, und sind daher in Bezug auf das exakte Ergebnis von Analysen von *relational attributes* am rechten Rand der Skala der Genauigkeit einzuordnen. Der dabei aufzubringende polynomielle worst–case Berechnungsaufwand der verschiedenen Arbeiten wird in dieser Einordnung als ungefähr gleich gewertet. Eine Einordnung der verbleibenden Algorithmen wird im nachfolgenden Abschnitt vorgenommen werden.

2.5 Eine Taxonomie von Berechnungsverfahren von Relational Attributes

Diejenigen in der Literatur vorgestellten Verfahren, die gegenüber der Berechnung aller Kombinationen von *independent attributes* eine erhöhte Genauigkeit erzielen wollen, lassen sich in die in Abbildung 2.2 gezeigte Taxonomie einordnen. Dabei bezeichnen die fett gedruckten Kategorien diejenigen Vorgehensweisen, die bereits in der Literatur behandelt wurden. Die anderen sind entweder theoretisch denkbar oder wurden in Arbeiten als Ausblick gegeben.

2.5.1 Mengenwertige Berechnung

Eine Möglichkeit, den Zusammenhang zwischen Analysefakten nicht zu verlieren, besteht darin, anstatt von einzelnen Analysefakten Mengen von Analysefakten zu betrachten und diese stets gemeinsam zu verarbeiten. Je nach Analyserichtung ergeben sich dabei verschiedene mögliche Vorgehensweisen.

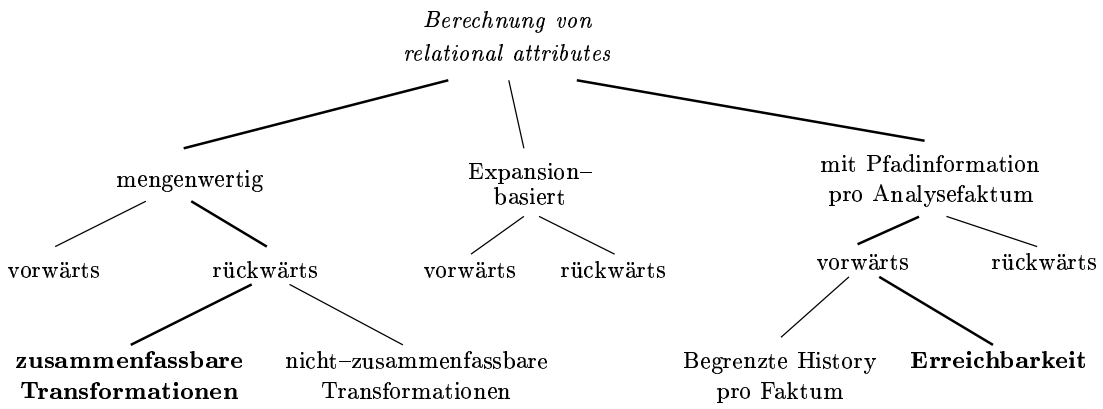


Abbildung 2.2: Taxonomie Berechnungsverfahren *relational attributes*

2.5.1.1 Mengenwertig — Vorwärts

Bei einer Analyse in Vorwärtsrichtung werden erzeugte Kombinationen von Analysefakten in Ausführungsrichtung des Programmes propagiert. Damit läßt sich eine mengenbasierte Berechnung in der Art realisieren, daß eine Kombination von Analysefakten ein *environment* von Variablen, also eine Menge von (Zeiger–)Variablenbelegungen ist. Diese Berechnungsmethode entspricht derjenigen, die bereits im Motivationsbeispiel eingeführt, und in Abschnitt 2.3.2 in das allgemeine Framework aus Definition 2.1 eingeordnet wurde. Wie man dieses Analyseverfahren auf das Motivationsbeispiel anwenden würde, wurde ebenfalls bereits vorgestellt.

In einem *environment* wird damit der Effekt sämtlicher Anweisungen entlang eines Pfades vom Anfang eines Programmes zusammengefasst. Diese Zusammenfassung läßt sich in Vorwärtsrichtung durch eine konkrete Variablenbelegung ausdrücken.

2.5.1.2 Mengenwertig — Rückwärts

Eine mengenwertige Berechnungsmethode in Rückwärtsrichtung besitzt zum Zeitpunkt der Analyse des Effektes einer Anweisung noch keine Informationen über die Daten, die diese in Vorwärtsrichtung erreichen können. Daher basiert diese Variante nicht auf Mengen von Variablenbelegungen, sondern fasst die in Rückwärtsrichtung angetroffenen Transformationen, im Sinne von Auswirkungen einer Anweisung auf eine Variablenbelegung, zusammen. Für jede in Rückwärtsrichtung angetroffene Anweisung läßt sich damit durch diese Zusammenfassung ausdrücken, welchen Effekt alle Anweisungen zwischen dieser Anweisung und derjenigen Anweisung, von der ab die Zusammenfassung generiert wurde, besitzen. Das Zusammenfassen von Anweisungen kann dann beendet werden, sobald die Zusammenfassung stets den gleichen Effekt beschreibt, der unabhängig von weiteren, noch davor angetroffenen Anweisungen ist.

Auch in Rückwärtsrichtung dürfen keine Analysefakten, hier also Transformationen, kombiniert werden, die nicht auf einem gemeinsamen (rückwärts interpretierten) Programmpfad vorkommen können. Entsprechend muß eine Analyse mit Exaktheitsanspruch Mengen von Transformationen berechnen, wobei i.A. für jeden möglichen Programmpfad vom Ausgangspunkt der Analyse rückwärts bis zu einer Stelle, an der eine definitive Aussage über das Analyseergebnis gemacht werden kann, eine eigene Zusammenfassungstransformation berechnet werden muß. Ein wichtiges Unterscheidungsmerkmal für Analysearten ist dabei, ob die von der Analyse angetroffenen Transformationen zusammenfassbar sind, oder nur aufgesammelt werden können, und ob für jede Anweisung bestimmbar ist, ob sie in die Zusammenfassungstransformation mit einbezogen werden muß.

2.5.1.3 Mengenwertig — Rückwärts — mit zusammenfassbaren Transformationen

In diesem ersten Fall sollen für die Art der Analyse zwei Eigenschaften angenommen werden:

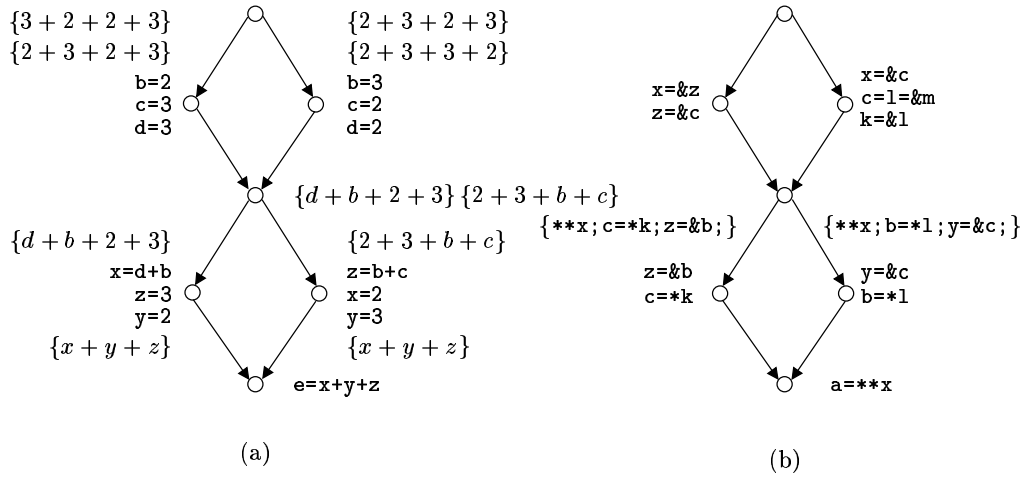


Abbildung 2.3: Zusammenfassbare (a) und nicht–zusammenfassbare Transformationen (b).

1. Die Transformationen sind zusammenfassbar.

2. Für jede Anweisung ist bestimmbar, ob sie eine Auswirkung auf das untersuchte Analysefaktum hat.

Ein Vertreter für eine Art von Analyse, bei der die aus den angetroffenen Anweisungen abgeleiteten Transformationen diese Eigenschaft besitzen, ist Konstantenanalyse. Abbildung 2.3(a) zeigt das Analyseprinzip, das Steffen und Knoop in [SK91] zur Berechnung von *finite constants* als einzige Vertreter der Kategorie der mengenwertigen, rückwärtsgerichteten Verfahren zur Berechnung von *relational attributes* anwenden. Als Darstellungsform wurden hier traditionelle Programmgraphen gewählt, bei der Knoten Anweisungen (hier vereinfachend Sequenzen von Anweisungen) und Kanten den möglichen Kontrollfluß zwischen diesen Anweisungen repräsentieren. Die gezeigte Struktur entspricht dabei zwei aufeinanderfolgenden If–Anweisungen mit jeweils einem then– und einem else–Zweig.

Für die Fragestellung, ob nach Ausführung der unten im Graphen stehenden Anweisung `e=x+y+z` die Variable `e` stets den gleichen konstanten Wert hat, kann man diesen zugewiesenen Term rückwärts im Programmgraphen propagieren und den Effekt der Anweisungen auf diesen Term zusammenfassen, bis dieser keine Variablen mehr enthält und damit einen konkreten Wert darstellt. Die Zusammenfassung geschieht dabei durch Substitution von Variablen durch die ihnen auf dem Weg zugewiesenen Terme. So wird zum Beispiel der Term $\{x + y + z\}$ durch die Zuweisung `y=2` in den Term $\{x + 2 + z\}$ umgewandelt, etc. Als Ergebnis erhält man, wenn man die oben in geschweiften Klammern stehenden Ergebnisterme normalisiert, die Aussage der Konstanz von `e` nach Ausführung der Anweisung `e=x+y+z`. Die Normalisierung der Terme ist ein i.A. nicht–triviales da in der allgemeinsten Form nicht–entscheidbares Problem. Interessanterweise haben neue Arbeiten gezeigt, daß dieses Problem für Terme, die aus Zuweisungen der Form $x := a_1 \cdot x_1 + \ldots + a_n \cdot x_n$ entstehen, entscheidbar ist [MOR01]. In [MOS02] wird dieses Ergebnis auf alle Terme, die sich aus Zuweisungen in der Form von Polynomauswertungen ergeben, ausgeweitet.

Die beiden oben angegebenen Voraussetzungen für die Zusammenfassbarkeit von Transformationen sind dabei erfüllt, da zum einen das Ergebnis der Ersetzung eines Vorkommens einer Variable in einem Term durch einen anderen Term stets wieder ein Term ist, und zum anderen eine Anweisung `x=t` genau dann eine Auswirkung auf den von der Analyse propagierten Term hat, wenn dieser die Variable `x` enthält.

In dieser Interpretationsweise entsprechen im Framework aus Definition 2.1 einzelne Analysefakten den angetroffenen Zuweisungen. Kombinationen von Analysefakten sind Terme, in denen die angetroffenen Zuweisungen wie vorgestellt substituiert wurden. Durch eine approximative Verein-

fachung des Problems gegenüber Konstantenanalyse in der allgemeinsten Form ist die Menge von verwendeten (Kombinationen von) Analysefakten endlich, und damit in das Framework einordenbar. Die Kombinationsabbildung $\otimes$ entspricht der Substitution von Teiltermen. Die Menge Δ entspricht Termen, bei denen alle relevanten Zuweisungen durch Substitution kombiniert wurden. Die Menge Θ entspräche einem einfachen Aufsammeln aller Zuweisungen, ungeachtet deren Relevanz für das betrachtete Analysefaktum.

Obwohl der Berechnungsaufwand im worst–case exponentiell mit der Programmgröße anwächst, beschreiben die Autoren ein in praktischen Tests beobachtbares gutartiges Verhalten der Analyse.

2.5.1.4 Mengenwertig — Rückwärts — mit nicht–zusammenfassbaren Transformationen

Das Gegenstück zur oben angegebenen Konstantenanalyse stellt Zeigerzielanalyse, also genau der Schwerpunkt dieser Arbeit, dar. Abbildung 2.3(b) zeigt einen entsprechenden Programmgraphen. Interessiert man sich für die möglichen Ziele der zweifachen Dereferenzierung der Zeigervariable `x` unten im Graphen, so kann man auch hier die Information `**x` rückwärts propagieren. Allerdings sind hier beide Voraussetzungen des letzten Falles nicht erfüllt. So lassen sich z.B. die beiden Zuweisungen im linken unteren Zweig `z=&b` und `c=*k` nicht zusammenfassen, ohne die entsprechenden Zeigerziele, die aber nur in Vorwärtsanalyserichtung bekannt sind, zu kennen. Insofern bleibt nur eine Generierung von Transformationen im Sinne des Aufsammelns von Sequenzen von Anweisungen, die bei jeder im weiteren Verlauf angetroffenen Anweisung aufwändig auf gemeinsame Auswertbarkeit überprüft werden müssen. Ob eine Anweisung die propagierten Informationen beeinflusst, ist ebenfalls nicht erkennbar. So sind z.B. die Zuweisungen im linken unteren Zweig relevant, während es diejenigen im rechten unteren Zweig nicht sind.

In diesem zweiten Fall von mengenwertiger Berechnung in Rückwärtsrichtung entspricht in der Interpretation des Frameworks aus Definition 2.1 ein einzelnes Analysefakten einer angetroffenen Zuweisung. Kombinationen von Analysefakten sind Mengen von aufgesammelten Zuweisungen. Um Endlichkeit zu garantieren müsste man analog zum Verfahren der Finite Constants bei Schleifen eine Beschränkung auf eine endliche Anzahl von aufgesammelten Zuweisungen vornehmen. Die Kombinationsabbildung $\otimes$ entspricht der Konkatenation von Sequenzen von aufgesammelten Anweisungen. Die Menge Δ entspräche Sequenzen von aufgesammelten Anweisungen, von denen für jede einzelne bekannt ist, daß sie einen Einfluß auf das gesuchte Ergebnis besitzt. Die Menge Θ entspricht dem Ergebnis des Aufsammelns aller Zuweisungen, ungeachtet deren Relevanz für das betrachtete Analysefaktum.

2.5.1.5 Diskussion mengenwertiger Berechnung

Die mengenwertige Berechnung von *relational attributes* in Vorwärtsrichtung erlaubt eine einfache Beschreibung des Ergebnisses einer Analyse von *relational attributes*. In reiner Vorwärtsrichtung gibt es aber keine Möglichkeit, Aussagen über die spätere Verwendung von Analysefakten zu treffen, so daß bei dieser Methode sämtliche auftretenden einzelnen Analysefakten zu Kombinationen zusammengefasst werden müssen, was daher der Menge Θ entspricht. Diese Methode, von der in der Literatur keine Realisierung bekannt ist, ist daher als nicht–ökonomisch einzustufen.

Für die mengenwertige Analyse in Rückwärtsrichtung hängt die Realisierbarkeit eines ökonomischen Prinzips von der Zusammenfassbarkeit von Transformationen für die spezielle Analyseart ab. Im Fall von Konstantenanalyse ist dies, wie bereits argumentiert wurde, der Fall, so daß die Analysefakten, in diesem Fall Terme als Transformationen, nur dann mit anderen Analysefakten, den Termen aus Zuweisungsanweisungen, durch Substitution kombiniert werden müssen, wenn die Variable, der in der Zuweisung der Term zugewiesen wurde, im aktuell propagierten Term vorkommt. Somit kann diese Berechnungsmethode für den Fall von zusammenfassbaren Transformationen als ökonomisch gewertet werden.

Im anderen Fall, also im Fall von nicht–zusammenfassbaren Transformationen, wie er bei Zeigeranalyse auftritt, kann für eine Anweisung in Rückwärtsrichtung nicht bestimmt werden, ob sie

einen Einfluß auf ein konkretes Analysefaktum hat. Daher müssen alle während der Analyse angetroffenen Anweisungen kombiniert werden, was aufgrund der Nicht–Zusammenfassbarkeit nur durch Aufsammeln sämtlicher angetroffener Anweisungen möglich ist. Dabei kann durch eine notwendige Beschränkung auf eine endliche Menge von Kombinationen von Analysefakten bereits die Exaktheit des Verfahrens verloren gehen. Darüberhinaus ist diese Vorgehensweise offensichtlich kein ökonomisches Berechnungsprinzip. Wiederum entsteht die Menge Θ an maximal möglichen Kombinationen von Analysefakten (wobei wie beschrieben die Art der Analysefakten in Rückwärtsrichtung unterschiedlich von den Analysefakten in Vorwärtsrichtung ist).

Um eine ökonomische mengenwertige Berechnungsmethode im Fall von Zeigeranalyse zu realisieren, wäre offensichtlich eine Kombination der Analyserichtungen nötig. Durch die Schwierigkeiten der Analyse in Rückwärtsrichtung kann man dies jedoch nicht durch ein intuitiv naheliegendes Verfahren leisten, das weiterhin garantiert exakte Ergebnisse liefert. In der Literatur sind keine Verfahren bekannt, die eine solche Art der Berechnung zum Ziel hätten.

2.5.2 Expansion–Basierte Verfahren

Ein Verfahren aus der Literatur, das ebenfalls unter Inkaufnahme von exponentiellem worst–case Berechnungsaufwand, das Ziel verfolgt, pfadabhängige Analyseergebnisse zu erhalten, ist *property oriented expansion* [Ste96b]. Diesem liegt die Beobachtung zugrunde, daß an Kontrollfluß–Joins, also Punkten im Programmgraph, an denen in Analyserichtung mehrere Pfade zusammentreffen, die Zusammenfassung der Ergebnisse der Einzelpfade zu Informationsverlust betreffend die Pfade, über die diese Informationen propagiert wurden, führt. Daher werden bei *property oriented expansion* Programmteile während der Analyse derart vervielfacht, daß nie zwei verschiedene Analysefakten an einem solchen Programmpunkt ankommen können. In einer späteren Analysephase können auf diese Weise getrennte Programmteilkopien anhand von passenden Kriterien wieder verschmolzen werden.

2.5.2.1 Expansion — Vorwärts

Bei der Anwendung dieses Verfahrens, zunächst in Vorwärtsrichtung betrachtet, auf die exakte Berechnung von Konstanten– oder Zeigeranalyse durch Betrachtung von *relational attributes* würde man als kombinierte Analysefakten wohl Kombinationen von Variablenbelegungen, also wie im mengenwertigen Fall *environments* von Variablen verwenden. Dabei ist an Kontrollfluß–Joins das Zusammentreffen verschiedener Analysefakten, also *environments* in denen mindestens eine Variable einen unterschiedlichen Wert besitzt, Anlaß zur Aufspaltung des Programmgraphen.

Die Einordnung dieses Verfahrens in das Framework kann analog zum mengenwertigen Fall in Vorwärtsrichtung vornehmen.

2.5.2.2 Expansion — Rückwärts

In Rückwärtsrichtung müsste analog zum mengenwertigen Fall die Expansion des Graphen anhand von Analysefakten erfolgen, die Transformationen repräsentieren, also ebenfalls den Effekt der von der Analyse untersuchten Anweisungen zusammenfassen. Daher ist auch die Einordnung in das Framework analog zu diesem Fall.

2.5.2.3 Diskussion Expansion–basierter Verfahren

Expansion–basierte Verfahren können in Vorwärtsrichtung genau wie die mengenbasierten Verfahren nicht bestimmen, welche einzelnen Analysefakten kombiniert werden müssen. Folglich müssen für ein exaktes Ergebnis mehr Programmteilkopien erzeugt werden, als man mit dem Wissen, welche Kombinationen relevant sind, benötigen würde. Die mögliche anschließende Wiederverschmelzung von Programmteilen würde an diesem zunächst aufzuwendenden Berechnungsaufwand nichts ändern. Daher ist dieses Verfahren in Vorwärtsrichtung als nicht–ökonomisch anzusehen.

Obwohl in Rückwärtsrichtung für den Fall von zusammenfassbaren Transformationen ein Expansion–basiertes Verfahren möglicherweise ähnlich praktikabel wie das Verfahren der *finite constants*

sein könnte, sind bei der Analyse von nicht–zusammenfassbaren Transformationen, konkret Zeigeranalyse als Anwendungsgebiet dieser Arbeit, die gleichen prinzipiellen Schwierigkeiten wie bei der mengenbasierten Auswertung in Rückwärtsrichtung zu erwarten. Die Expansion in Rückwärtsrichtung müsste daher anhand von potentiell unbeschränkten Sequenzen von bei der Analyse durchlaufenen Anweisungen erfolgen, von denen nicht beurteilt werden kann, ob sie für ein exaktes Ergebnis benötigt werden. Dies ist daher nicht als ökonomisch zu werten.

2.5.3 Assoziation von Pfadinformationen mit jedem einzelnen Analysefaktum

Als grundlegende Alternative zur Berechnung von Mengen von zusammengehörigen Analysefakten und den Expansion–basierten Verfahren ist die Assoziation von jedem Analysefaktum mit Information über die von diesem Faktum bei der Analyse durchlaufenen Pfade möglich. Damit ist bei jeder Kombination von Fakten ein Vergleich auf eine "gemeinsame Vergangenheit" in Analyserichtung der Fakten möglich, so daß die mitgeführte Pfadinformation idealerweise genügen müsste, um ungültige Kombinationen zu vermeiden. Dadurch ist zumindest prinzipiell die Möglichkeit von *delayed evaluation* gegeben, d.h. daß erst bei einem Zusammentreffen von Fakten auf die Gültigkeit ihrer Kombination getestet werden muß, und daher erst zu diesem Zeitpunkt Rechenaufwand investiert werden muß, anstatt bereits vorher Kombinationen aller möglicherweise zu kombinierenden Fakten zu bilden.

2.5.3.1 Pfadinformation — Vorwärts

In Vorwärtsrichtung entspräche diese Methode im Idealfall der Protokollierung des von einem Analysefaktums ab dem Zeitpunkt seiner Generierung bei der Analyse durchlaufenen Programmpfades.

Als Ausgangspunkt in der Literatur könnte man für diese theoretisch denkbare Algorithmenklasse den *call–string approach* von Sharir und Pnueli [SP81] ansehen, bei dem (allerdings nur für Funktionsaufrufe und nicht für durch If– oder While–Anweisungen bedingte intraprozedurale Pfade) jedes Analysefaktum mit einer sogenannten *propagation history* versehen wird, die die ggfs. unendliche Menge an zuletzt durchlaufenen Programmblöcken oder Anweisungen protokolliert.

Ein einzelnes Analysefaktum entspricht in dieser Sichtweise einem Paar aus einer Aussage, z.B. der Form $x \rightarrow y$, und einem damit assoziierten Pfad π, also z.B. $f = (x \rightarrow y, \pi) \in \mathbf{F}$. Kombinationen von Analysefakten sind damit z.B. von der Form $(x \rightarrow y \wedge y \rightarrow A, \pi')$, wobei sich der Pfad π' als der längste Pfad der an der Kombination beteiligten Fakten ergibt. Als Voraussetzung dürfen überhaupt nur Fakten kombiniert werden, deren damit assoziierte Pfade verschieden lange Endungen des gleichen Programmpfades sind. Diese Überprüfung ist damit Teil der Kombinationsabbildung $\otimes$. Die Mengen Δ und Θ ergeben sich als die Mengen aller solcher Kombinationen aus Konjunktionen von Aussagen und Pfaden, die im weiteren Verlauf der Analyse noch benötigt werden, bzw. die bei der Analyse ohne Rücksicht hierauf berechnet werden.

Ein prinzipielles Problem dabei ist es, daß weder die Menge $\mathbf{F}$ von Fakten, noch die Menge $\mathbf{K}$ von Kombinationen von Fakten endlich sind. Im Fall von Schleifen können die Histories, also die mit den Aussagen assoziierten Pfade, unendlich groß werden. Um überhaupt ein Verfahren mit diesem Prinzip in Betracht ziehen zu können, muß daher die Größe der mitgeführten Pfadinformation begrenzt werden.

2.5.3.2 Pfadinformation — Vorwärts — mit begrenzter History

Die Begrenzung der Größe der Histories auf einen konstanten maximalen Wert k, und damit auf die zuletzt durchlaufenen k Programmblöcke, resultiert i.A. in einem Verlust der Exaktheit des Ergebnisses. Diese Vorgehensweise wird in der Literatur als *k–Limiting* [MJ81b] bezeichnet (wobei sich der Ausdruck bei verschiedenen Autoren sowohl auf das Abschneiden von interprozeduralen Histories, als auch auf die Begrenzung von sogenannten *access paths* der Form `x->y->z` bezieht).

In der Literatur gibt es keine Ergebnisse über die Anwendung dieses intraprozeduralen Algorithmus.

2.5.3.3 Pfadinformation — Vorwärts — durch Erreichbarkeitstest angenähert

Eine Approximation von Histories wird in [BCCH97] durch Erreichbarkeitstests geleistet. Dabei wird jedes Analysefaktum mit seiner sogenannten *birth site* versehen, die z.B. die Entstehung des Faktums $x \rightarrow y$ auf eine konkrete Zeile der Form `x = &y` zurückführt. Soll nun solch ein Faktum mit einem zweiten kombiniert werden, so kann man dies als *ungültige Kombination* erkennen, falls nicht die *birth site* des einen Faktums von der *birth site* des anderen Faktums aus im Programmgraphen erreichbar ist. Andernfalls gelten die Fakten in approximativer Vereinfachung des Problems als kombinierbar. Fakten, die bereits als Kombinationen von anderen Fakten erzeugt wurden, erhalten als *source alias sets* die Menge der an der Kombination beteiligten *birth sites* als Information. Um die Kosten für den Erreichbarkeitstest zu beschränken, wird diese Menge auf eine feste Maximalgröße limitiert.

Der Erreichbarkeitstest ist bereits bei Schleifen und Alternativen sehr ungenau — für das Motivationsbeispiel ermittelt der Algorithmus das gleiche approximative Ergebnis wie die angegebene Analyse von *independent attributes*, da die Zuweisungen im then–Zweig von den Zuweisungen vor der Schleife aus erreichbar sind.

Die Analysemethode ist in Teilen prinzipiell auch für den interprozeduralen Fall konzipiert. Allerdings wird dabei jeder Funktionsrumpf als von jedem anderen Funktionsrumpf erreichbar interpretiert, weshalb bei der Überprüfung auf die Gültigkeit der Kombination von Analysefakten interprozedurale Pfade keine Rolle spielen. Daher werden bei dieser Analysetechnik zur Vermeidung von ungültigen Kombinationen von Analysefakten, genau wie bei der vorgestellten Arbeit auf dem Gebiet der Konstantenanalyse, ausschließlich intraprozedurale Pfade berücksichtigt.

2.5.3.4 Pfadinformation — Rückwärts

Eine Kombination von rückwärtsgerichteten Analysefakten mit Pfadinformation existiert bisher noch nicht. In diese Kategorie fällt das in dieser Arbeit vorgestellte Verfahren. Die prinzipiellen Schwierigkeiten von Zeigeranalyse in Rückwärtsrichtung werden dabei dadurch umgangen werden, daß Fakten mit endlichen Beschreibungen von ggfs. unendlichen Mengen von Programminstanzen assoziiert werden, in denen diese Fakten gelten. Auf dies wird noch genauer eingegangen werden.

2.5.3.5 Diskussion der Assoziation von Pfadinformationen mit Analysefakten

Durch die Nicht–Exaktheit der beiden vorgestellten Vertreter dieser Klasse von Algorithmen, dem Mitführen von begrenzte Histories und der Approximation von Histories durch Erreichbarkeitstests, fällt diese Art von Analysetechnik nicht in die gleiche Klasse wie das in dieser Arbeit vorgestellte exakte Verfahren, weswegen ein direkter Vergleich nicht möglich ist.

Die Tatsache, daß der erste der beiden Ansätze trotz seines einfachen Prinzips bisher in der Literatur nicht praktisch realisiert wurde, und der zweite Ansatz bereits im Motivationsbeispiel die gleichen ungültigen Kombinationen wie die Analyse von *independent attributes* berechnet, spricht jedoch nicht für ihre Einsetzbarkeit mit dem Anspruch auf exakte, aber trotzdem praktische Realisierbarkeit im Anwendungsgebiet dieser Arbeit.

2.6 Zusammenfassung

Es gibt in der Literatur nur wenige Arbeiten, die eine Erweiterung der Analyse von *independent attributes* in Richtung von *relational attributes* zum Ziel haben. Diese befassen sich entweder mit dem von seinen Eigenschaften bzgl. Zusammenfassbarkeit besser handhabbaren Problem der Konstantenanalyse, oder leisten im Fall von Zeigeranalyse eine approximative Lösung, die beim Auftreten von Schleifen oder Alternativen nur in einem kleinen Teil der möglichen Fälle eine Verbesserung gegenüber einer Analyse von *independent attributes* bietet. Eine exakte Lösung für Multi–Level–Zeigeranalyse wurde bisher nicht vorgestellt.

In Tabelle 2.1 ist eine Übersicht über die vorgestellten Verfahrensprinzipien aus der Taxonomie, und die Verfahren aus der Literatur, die diese Prinzipien ggfs. verwenden gezeigt. Die vorgestellten prinzipiell denkbaren, intuitiv naheliegenden Verfahren wurden bisher nicht realisiert. Alle diese

Verfahrenskategorie	Richtung	Analyseart	Prinzip	Verfahren	Exakt	Ökonomisch
Mengenwertig	Vorwärts	Zeigeranalyse	Environments	—	√	
	Rückwärts	Konstantenanalyse	Termsubstitution	[SK91]	(√)	√
		Zeigeranalyse	Anweisungsaggregation	—	√	
Expansionbasiert	Vorwärts	Zeigeranalyse	Environments	—	√	
	Rückwärts	Zeigeranalyse	Anweisungsaggregation	—	√	
Pfadinformation	Vorwärts	Zeigeranalyse	Histories	—		
	Vorwärts	Zeigeranalyse	Erreichbarkeit	[BCCH97]		
	Rückwärts	Zeigeranalyse	—	—		

Tabelle 2.1: Überblick über realisierte und denkbare Analyseverfahren von *relational attributes*.

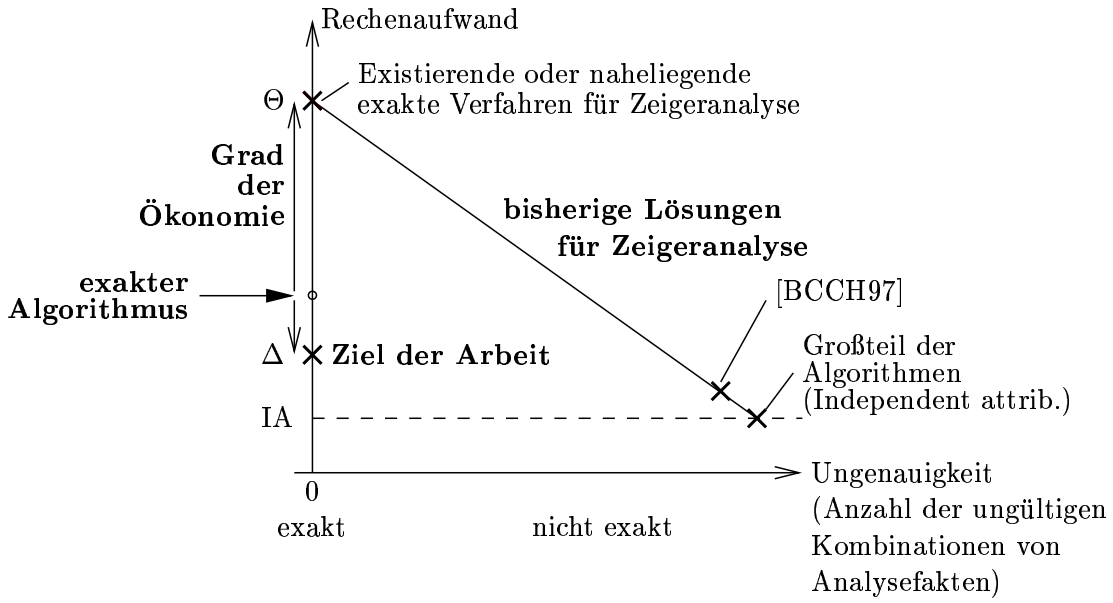


Abbildung 2.4: Einordnung aller Algorithmen nach Rechenaufwand und Exaktheit.

Verfahren besitzen die gemeinsame Eigenschaft, daß sie nicht ökonomisch realisierbar sind. Daher erscheint es plausibel, daß sie ggfs. bei genauerer Betrachtung als nicht praktisch realisierbar eingeordnet wurden. Das einzige Verfahren für eine Analyse von *relational attributes*, das tatsächlich realisiert wurde, und das laut den Autoren praktisch einsetzbar ist, verfolgt ein ökonomisches Berechnungsprinzip (die Nicht–Exaktheit des Verfahrens resultiert aus der schwierigeren Problemklasse von Konstantenanalyse mit arithmetischen Operationen, bei der eine unendliche Menge von Termen bei der Analyse erzeugt werden kann). Damit scheint Ökonomie eine Schlüsseleigenschaft für die praktische Einsetzbarkeit von Analyseverfahren von *relational attributes* zu sein.

Eine Einordnung der Verfahren für Multi–Level–Zeigeranalyse aus der Taxonomie in das vorgestellte Diagramm aus Abbildung 2.1 findet sich in Abbildung 2.4. Sämtliche bisher vorgestellten oder vorstellbaren Verfahren befinden sich im Diagramm auf der eingezeichneten Geraden. Zusätzlich läßt sich nun das vorgestellte Verfahren von Burke, Choi, Carini und Hind von 1997 in etwa beim Großteil der Verfahren rechts im Diagramm einordnen. Die gewonnene Genauigkeit in Bezug auf eine Analyse von *relational attributes* übersteigt die einer Analyse von *independent attributes* nicht signifikant, der dafür investierte Rechenaufwand ist aufgrund der vorgestellten weiteren Vereinfachungen ebenfalls nicht bedeutend größer. Alle vorgestellten denkbaren exakten Verfahren müssen die Menge Θ von Kombinationen berechnen, und sind daher nicht ökonomisch.

Kapitel 3

Zielsetzung

Das Ziel der Arbeit ist es, eine neue statische Analysetechnik mit maximalem Grad an Ökonomie gemäß Abbildung 2.4 zur exakten Bestimmung von Zeigerzielen zu entwickeln. Die in der Literatur vorgestellten Verfahren können ein solches ökonomisches Verfahren, wie in Kapitel 2 bereits diskutiert wurde, für die nachfolgend angeführte Klasse von Eingabeprogrammen nicht leisten, da sie nicht bestimmen können, welche minimale Menge an Informationen für das exakte Ergebnis benötigt wird, und daher eine Obermenge dieser Informationen berechnen müssen. Dabei sind, wie bereits gezeigt, die für diese Verfahren problematischen Programme solche, bei denen Multi–Level–Zeiger, d.h. Zeiger verschiedener Referenzierungsstufe wie z.B. Variablen vom Typ "Zeiger auf Basistyp" und vom Typ "Zeiger auf Zeiger vom Basistyp", zusammen vorkommen können.

Das vorgestellte Verfahren beschränkt dabei die Klasse von Eingabeprogrammen derart, daß eine exakte Lösung von Multi–Level–Zeigeranalyse mittels regulärer Sprachen (deren exakte Art der Verwendung später vorgestellt werden wird) möglich ist. Daraus ergibt sich als erste Auswirkung, daß im Rahmen dieser Arbeit keine Funktionsaufrufe zugelassen werden. Diese Einschränkung findet man ebenso bei allen anderen Arbeiten mit der Zielsetzung einer erhöhten Genauigkeit gegenüber Analysen von *independent attributes*. Für das vorgestellte Verfahren ist dies keine prinzipielle Einschränkung, eine entsprechende mögliche Erweiterung, die dann kontextfreie Sprachen zur Exaktheit benötigt, findet sich im Ausblick in Abschnitt 13.1.

Als weitere Einschränkung ergibt sich der Verzicht auf strukturierte Datentypen in der zugelassenen Programmklasse. Tatsächlich stellt man fest, wenn man die vorgestellte Analysetechnik auf Programme mit strukturierten Datentypen erweitert, daß für eine exakte Berechnung Stacks bei der Analyse mitgeführt werden müssen, was ebenfalls nicht mehr durch reguläre Sprachen beschreibbar ist. Entsprechende Erweiterungen des Verfahrens werden im Ausblick in Abschnitt 13.2 beschrieben werden.

3.1 Betrachtete Programmklasse

Als zugelassene Programmklasse soll eine Sprache mit den folgenden Bestandteilen dienen:

- Variablen von einem skalaren Basistyp.
- Variablen vom Typ Zeiger auf (Zeiger auf ...) Variablen vom Basistyp.
- Generieren der Adresse $\&v$ einer Variable v durch den Adress–Operator &.
- Dereferenzieren einer Zeigervariable w durch den Dereferenzierungsoperator $*$.
- Anweisungen, die Zuweisungen, If– oder While–Anweisungen sein können.
- Zuweisungen der Form $l = r$, wobei l und r jeweils Variablen oder beliebig häufige Dereferenzierungen von Variablen sein können, und r zusätzlich die Adresse einer Variablen sein darf.

- Programmblöcke als Sequenzen von Anweisungen.
- If–Anweisungen mit uninterpretierter Bedingung und je einem ggfs. leeren Programmblock als then– und else–Zweig.
- While–Anweisungen mit uninterpretierter Bedingung und einem Programmblock als Schleifenrumpf.

Damit entspricht ein Programm aus dieser Programmklasse einem Programmblock, dessen Anweisungen geschachtelte If– und While–Anweisungen sein können. Wie man das vorgestellte Analyseverfahren auf beliebigen Kontrollfluß mit `goto`–Anweisungen erweitern kann, wird in Abschnitt 9.2.5.2.4 anhand eines analysierten Beispielprogrammes vorgestellt.

Um die Beschreibung nicht unnötig zu verkomplizieren, wird von fehlerfreien Eingabeprogrammen ausgegangen, in denen keine uninitialisierten Variablen ausgelesen werden. Dies ist ebenfalls keine prinzipielle Einschränkung, da man die Analyse um ein Zeigerziel "undefiniert" erweitern kann, das auf eine intuitiv naheliegende, jedoch viele Fallunterscheidungen erfordernde Weise in die Berechnungsvorschrift integriert werden kann

3.2 Problemstellung

Für ein gegebenes Eingabeprogramm aus obiger Programmklasse soll für jede im Programm vorkommende Dereferenzierung einer Zeigervariable die exakte Menge ihrer Zeigerziele berechnet werden. Dazu müssen anders als dies beim Großteil der in der Literatur vorgestellten Verfahren geschieht, nicht nur die möglichen Ziele von einzelnen Zeigervariablen betrachtet werden, sondern darüber hinaus untersucht werden, welche Zeigerziele von verschiedenen Variablen in einem konkreten möglichen Ablauf eines Programmes gemeinsam auftreten können.

Wie in Kapitel 2 angesprochen wurde, besteht bei Multi–Level–Zeigeranalyse die Notwendigkeit, verschiedene Analysefakten in der Form von Zeigerzielen einzelner Variablen zu kombinieren, um weitere Analysefakten zu erzeugen. Im Motivationsbeispiel ist z.B. das Ergebnis der zweifachen Dereferenzierung der Variable `x` vom Inhalt dieser Variable, und gleichzeitig vom Inhalt derjenigen Variable abhängig, auf die `x` zeigt. Daher kann das Ergebnis der Analyse unexakt werden, wenn dabei Analysefakten kombiniert werden, die tatsächlich nie gemeinsam in einer Programminstanz auftreten können. Es muß also eine Möglichkeit gefunden werden, zu verschiedenen Analysefakten Aussagen treffen zu können, ob sie gemeinsam in Programminstanzen auftreten können. Wie bereits in Kapitel 2 ausführlich beschrieben wurde, wird diese Art von Analyse in der Literatur als Analyse von *relational attributes* bezeichnet.

3.3 Lösungsansatz

Im Allgemeinen hat statische Programmanalyse das Ziel, zusammenfassende Aussagen über die i.A. unendliche Menge aller Möglichkeiten, wie ein Eingabeprogramm ablaufen kann, zu machen. Jeder einzelne mögliche Ablauf, der unter den für statische Analyse üblichen Annahmen durch die durchlaufenen Programmpfade eindeutig charakterisiert wird, wird als eine Programminstanz bezeichnet. Üblicherweise werden nur Programminstanzen betrachtet, die eine bestimmte Anweisung erreichen, an der die Gültigkeit einer bestimmten Eigenschaft überprüft werden soll.

Im Fall von Zeigeranalyse interessiert man sich dabei für eine Menge von Variablen, die jeweils in mindestens einer möglichen Programminstanz Ziel einer untersuchten Zeigervariable an einer bestimmten Stelle im Programm sein können. Sobald unterschiedliche mögliche Ziele einer Zeigervariable zu einer Menge von Ergebnissen zusammengefasst werden, ist jedoch nicht mehr nachvollziehbar, in welchen konkreten Programminstanzen jedes einzelne Ergebnis vorkommen kann. Dies ist z.B. im Motivationsbeispiel in Abb. 1.1(d) der Fall, wo keine Zuordnung der möglichen Ziele der Zeigervariablen x, y und z zu den Programminstanzen existiert, in denen diese Ziele angenommen werden.

3.3.1 Grundideen

Die zentrale Idee dieser Arbeit ist es, im Gegensatz dazu den Zusammenhang zwischen einem Zeigerziel und der i.A. unendlichen Menge von Programminstanzen, in denen dieses Zeigerziel angenommen wird, in die Analyse zu integrieren.

3.3.1.1 Assoziation von Analysefakten mit Mengen von Programminstanzen

Nimmt man an, zu jedem Ziel einer Zeigervariable an einer bestimmten Stelle im Programm die Information zur Verfügung zu haben, in welcher Menge von Programminstanzen die Zeigervariable dieses Ziel annimmt, so kann man diese Information verwenden, um ungültige Kombinationen von Zeigerzielen zu vermeiden. Ist der Schnitt der mit verschiedenen Zeigerzielen assoziierten Mengen von Programminstanzen nicht–leer, so existiert mindestens eine Programminstanz, in der die an der Kombination beteiligten Zeigervariablen diese Ziele gemeinsam annehmen, weshalb die Kombination in diesem Fall zulässig ist. Diese Schnittmenge beschreibt auch gleichzeitig die Menge aller Programminstanzen, in denen dieses kombinierte Faktum auftritt, welche wiederum bei weiteren möglichen Kombinationen zum Testen derer Zulässigkeit verwendet werden kann.

3.3.1.2 Charakterisierung von Programminstanzenmengen durch Sprachen

Die Menge aller Programminstanzen stellt an sich eine algorithmisch schwer handhabbare Berechnungsgrundlage dar. Eine Programminstanz eines Programmes aus der in dieser Arbeit zugelassenen Programmklasse ist aber durch die von ihr bei der Ausführung durchlaufenen Programmpfade eindeutig charakterisiert, da entlang eines bestimmten Programmpfades die Anweisungen stets die gleichen Auswirkungen haben müssen, und daher keine zwei unterschiedliche Programminstanzen mit gleichen durchlaufenen Pfaden existieren können. Ein Pfad bezeichnet dabei üblicherweise die während eines konkreten Programmlaufes ausgeführten Anweisungen vom Anfang des Programmes bis zu seiner Terminierung, oder bis zum Erreichen einer bestimmten Anweisung. Wie in der Literatur üblich, werden dabei alle möglichen Pfade als ausführbar angenommen, d.h. die Verzweigungsbedingungen werden nicht interpretiert. Damit umgeht man das Problem, daß i.A. nicht berechenbar ist, welche Pfade ein Programm bei seiner Ausführung durchlaufen wird. Mit dieser Annahme ist z.B. zu jedem Zeitpunkt sowohl der then– als auch der else–Zweig einer If–Anweisung ausführbar.

Durch diese Korrespondenz von Programminstanzen und durchlaufenen Pfaden kann die Beschreibung eines Programmpfades vom Anfang eines Programmes bis zu seiner Terminierung eine Programminstanz, und darauf aufbauend die Beschreibung einer Menge von Pfaden eine Menge von Programminstanzen repräsentieren.

Drückt man jede mögliche Wahl einer Verzweigung in einem Programm durch ein Symbol aus, so kann man eine Programminstanz durch ein Wort aus solchen Symbolen repräsentieren. Eine Menge von Programminstanzen ist damit durch eine Menge von Worten darstellbar. Solche Mengen von Worten werden i.A. als Sprachen bezeichnet. Die Beschreibung von Pfadmengen wird in dieser Arbeit durch die von endlichen Automaten akzeptierten Sprachen geleistet, worauf in Kapitel 5 im Detail eingegangen wird.

Durch die Sprache der Automaten ist eine Beschreibungsmöglichkeit gegeben, um einem konkreten Ziel einer Zeigervariable an einer Anweisung, im folgenden allgemein als *Instanzenmengeneigenschaft* bezeichnet, eine Menge von Programminstanzen zuzuordnen. Die Zeigervariable besitzt genau dann in einer Programminstanz dieses Ziel, wenn diese Instanz in der durch die Sprache des Automaten ausgedrückten Menge von Programminstanzen enthalten ist. Da diese Instanzenmenge durch eine Beschreibung einer Menge von Pfaden spezifiziert wird, wird die Pfadmengenbeschreibung im folgenden als *Pfadbedingung* für diese *Instanzenmengeneigenschaft* bezeichnet. Eine genaue Definition von Pfadbedingungen und Instanzenmengeneigenschaften wird in Abschnitt 5.6 gegeben werden.

Im Rahmen dieser Arbeit wird im weiteren als Programminstanz nur eine Ausführung des Programmes bis zu derjenigen Anweisung verstanden, an der eine Instanzenmengeneigenschaft untersucht wird. Die entsprechenden Pfadbeschreibungen enden daher alle bei dieser Anweisung. Dies entspricht der in der Literatur üblichen Vorgehensweise.

3.3.1.3 Ermöglichen von praktischer Realisierbarkeit und Endlichkeit der Analyse

Ein prinzipielles Problem der Beschreibung von bei der Analyse berücksichtigten Programmpfaden liegt in der Nicht–Beschränktheit der Menge der möglichen Pfade, sobald Schleifen im zu analysierenden Programm vorkommen. Die Menge aller Programminstanzen, die zu einer bestimmten Instanzenmengeneigenschaft im obigen Sinne gehören, ist i.A. unendlich. Um trotzdem mit diesen Mengen arbeiten zu können, sind die folgenden Schritte notwendig:

3.3.1.3.1 Erkennung der Struktur von Pfaden

Eine unendliche Menge von Programminstanzen, beschrieben durch eine Menge von Pfaden, hat im Rahmen dieser Arbeit stets reguläre Struktur, d.h. durch eine Analyse dieser Pfade ist es möglich, eine unendliche Menge von Pfaden durch endliche Pfadabschnitte und die Anwendung der Prinzipien der Konkatenation, Wiederholung und Alternative auf diese Abschnitte zu beschreiben. Damit kann eine unendliche Menge von Programminstanzen durch eine endliche Beschreibung zusammengefasst werden. Diese Beschreibung dient dann als Grundlage für ein exaktes Analyseverfahren.

3.3.1.3.2 Klassenbildung von Pfadmengen

Ein effizientes Verfahren, um die Menge *aller* Pfade zwischen zwei vorab festgelegten Anweisungen in traditionellen Programmgraphen durch einen regulären Ausdruck darzustellen, wurde bereits in [Tar81] vorgestellt. Im Gegensatz dazu sollen in dieser Arbeit zum einen nur Teilmengen aller Pfade zwischen bestimmten Anweisungen betrachtet werden, auf denen man die gleichen semantischen Eigenschaften der auf diesen Pfaden durchlaufenen Anweisungen beobachten kann. Zum anderen soll die Anweisung, ab der beginnend Programmpfade beschrieben werden, nicht als eine feste Anweisung oder der Start des Programmes festgelegt werden, sondern variabel gehalten werden, um eine möglichst minimale und platzsparende Beschreibungsform zu ermöglichen.

Für eine Instanzenmengeneigenschaft ist i.A. nicht der gesamte durchlaufene Programmpfad vom Beginn des Programmes bis zur Anweisung, an der die Instanzenmengeneigenschaft gelten soll, relevant. Betrachtet man z.B. Ziele von Zeigervariablen, so wird an einer bestimmten Stelle im Programm die Adresse einer Variable v generiert, und diese Adresse beim weiteren Ablauf des Programmes möglicherweise in verschiedene andere Zeigervariablen umkopiert, bevor sie dann einer zu dereferenzierenden Zeigervariable zugewiesen wird. Bei deren Dereferenzierung an einer betrachteten Anweisung ergibt diese dann die Variable v als Zeigerziel. Da eine Variable zu einem Zeitpunkt in einer Programminstanz nur einen Wert annehmen kann, muß für jede Programminstanz, die vor Erreichen dieser Dereferenzierung den gleichen Endabschnitt des Pfades wie die gerade beschriebene Programminstanz beschreitet, die gleiche Instanzenmengeneigenschaft gelten, d.h. die Dereferenzierung ebenfalls die Variable v ergeben. Daher kann dieser letzte relevante Pfadabschnitt vor Erreichen der Anweisung, an der die Variable dereferenziert wird, verwendet werden, um die Klasse aller Pfade, die die gleiche Endung besitzen, zu bilden. Für alle Pfade, die in dieser Klasse liegen, kann man dann die Gültigkeit der Instanzenmengeneigenschaft folgern. Im Gegensatz zur Zielsetzung in [Tar81] werden dadurch nur Teilmengen von Pfaden, die eine bestimmte Anweisung erreichen, beschrieben.

Betrachtet man darüberhinaus zusätzlich die Menge von Programmpfaden, die die Anweisung mit der Dereferenzierung der Zeigervariablen von einer anderen Adresszuweisung an eine Variable ausgehend erreichen, so erkennt man auch die Notwendigkeit, verschiedene Anweisungen zuzulassen, von denen ab Pfadbeschreibungen gebildet werden.

Eine weitere Art der Klassenbildung besteht im "Auslassen" von für eine Instanzenmengeneigenschaft nicht relevanten Symbolen in diesen Endabschnitten von Pfaden. Auf dies wird in Abschnitt 5.1.2 informell, und in Abschnitt 5.5.2 detailliert eingegangen werden.

3.3.2 Überblick über das Analyseverfahren

Das Ziel des in dieser Arbeit vorgestellten Analyseverfahrens ist das Bestimmen von Pfadbedingungen zu Instanzenmengeneigenschaften. Wenn für eine untersuchte Instanzenmengeneigenschaft die

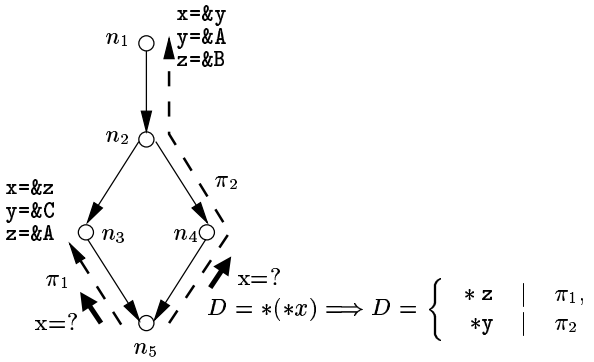


Abbildung 3.1: Transformation von Single–Level Zeigeranalyse in ein Erreichbarkeitsproblem.

durch die Pfadbedingung beschriebene Menge von Programminstanzen nicht–leer ist, dann existieren Programminstanzen, in denen diese Instanzenmengeneigenschaft auftritt.

3.3.2.1 Praktische Sichtweise des Analyseverfahrens

Anhand des Motivationsbeispiels soll im Folgenden ein Überblick aus praktischer Sichtweise über das Prinzip des Analyseverfahrens gegeben werden. Dieses läßt sich auf das Verfahrensprinzip von Erreichbarkeitsproblemen, bzw. einer Erweiterung davon, zurückführen. Um dies zu veranschaulichen, wird im Folgenden eine vereinfachte Darstellung des Analyseverfahrens vorgestellt. Die tatsächliche Vorgehensweise im Detail wird sich jedoch davon unterscheiden. Darauf wird in Abschnitt 3.3.3 genauer eingegangen werden.

3.3.2.1.1 Transformation in ein Erreichbarkeitsproblem

Das Grundprinzip des vorgestellten Verfahrens basiert auf der Transformation des Analyseproblems in ein Erreichbarkeitsproblem auf einer Graphrepräsentation des zu analysierenden Programmes, entgegen dessen Ausführungsrichtung durchgeführt. Ein Erreichbarkeitsproblem befaßt sich dabei i.A. mit der Aufgabe, zu einem gegebenen Graph mit Knoten und (gerichteten) Kanten, die von einem bestimmten Knoten aus über Kanten erreichbaren anderen Knoten zu bestimmen. Konkret wird bei der hier beschriebenen Analyse zu jeder Dereferenzierung einer Zeigervariable eine Anfrage (im folgenden als *Query* bezeichnet) erzeugt, die rückwärts in einer Graphdarstellung des Programmgraphen sucht, bis der Zeigervariablen an einem Graphknoten (einer Anweisung) die Adresse einer Variablen v zugewiesen wird. Damit ergibt die Dereferenzierung der Zeigervariable diese Variable v. Wird dabei während der Suche der Query der Zeigervariablen der Inhalt einer anderen Variable zugewiesen, so sucht die Query entsprechend nach der letzten Zuweisung zu dieser Variablen, etc., bis auch hier ein Ergebnis in Form der Zuweisung der Adresse einer Variablen gefunden wird. Damit muß die Suche i.A. nur einen Teil des Programmes untersuchen, da bei dieser Zuweisung bereits eine definitive Aussage über das Ergebnis der Dereferenzierung getroffen werden kann, ohne daß der Programmabschnitt davor eine Rolle spielen würde. Dies entspricht dem Prinzip der Klassenbildung von Programminstanzenmengen, das in Abschnitt 3.3.1 über die Grundideen des Verfahrens vorgestellt wurde.

Während der Bearbeitung der Query wird der dabei durchlaufene Programmpfad (entsprechend in Rückwärtsrichtung beschrieben) protokolliert, und als Pfadbedingung zum gefundenen Ergebnis, bzw. der Instanzenmengeneigenschaft, gespeichert.

Dieses Prinzip soll durch die folgende Betrachtung des Motivationsbeispiels veranschaulicht werden. In Abbildung 3.1 ist der traditionelle Programmgraph für das Motivationsbeispiel dargestellt. Der Knoten n_4 entspricht dabei dem leeren else–Zweig. Um die möglichen Belegungen der Variable `x` am unteren Knoten n_5 (und damit die Ziele der Dereferenzierung `*x` dieser Variablen) herauszufin-

den, wird eine Query mit der Aufgabe, rückwärts im Programmgraphen nach der letzten Zuweisung zur Variablen `x` zu suchen, gestartet. Diese Query ist in der Abbildung durch den Pfeil mit der Beschriftung "x=?" dargestellt. Vom Knoten n_5 aus gibt es zwei Möglichkeiten, im Programmgraph rückwärts zu suchen. Je nachdem, ob von einer Programminstanz der then– oder der else–Zweig durchlaufen wird, besitzt `x` den Inhalt, der der Variablen zuletzt im then– bzw. else–Zweig zugewiesen worden ist. Um sämtliche möglichen Zeigerziele herauszufinden, muß die Query in beiden Zweigen rückwärts suchen, und dabei den jeweils gewählten Pfad protokollieren.

Bei Durchlaufen des Analysepfades π_1 in Rückwärtsrichtung wird im then–Zweig eine Zuweisung der Adresse der Variablen `z` an die Variable `x` gefunden. Damit muß `x` bei Durchlaufen des (vorwärts interpretierten) Pfades π_1 in Ausführungsrichtung am Knoten n_5 auf die Variable `z` zeigen. Der Pfad π_1, auf dem in Rückwärtsrichtung ein Ergebnis gefunden wurde, entspricht damit anschaulich in Vorwärtsrichtung einer Pfadbedingung im wörtlichen Sinne für die Instanzenmengeneigenschaft $x \rightarrow z$: Wann immer der Pfad π_1 in Vorwärtsrichtung durchlaufen wird, muß die Dereferenzierung der Variablen `x` am Knoten n_5 die Variable `z` ergeben. Damit besteht die dadurch beschriebene Menge von Programminstanzen aus der einen Instanz, die bei der einzigen Verzweigungsmöglichkeit den then–Zweig beschreitet. Als zweite Instanzenmengeneigenschaft mit Pfadbedingung im Beispiel wird von der Analyse die Zuweisung der Adresse von `y` an die Variable `x` bei Durchlaufen des Pfades π_2 gefunden. Dieser repräsentiert die Programminstanz, die den else–Zweig ausführt, und damit die einzige andere mögliche Programminstanz. Die Ergebnisse, die von den einzelnen Queries gefunden werden, werden zusammen mit den dabei durchlaufenen Pfaden an denjenigen Knoten zurückgesandt, der die Queries initiiert hat. Dies entspricht damit einer vorwärtsgerichteten Komponente der Analyse. In Abbildung 3.1 wird dies dadurch symbolisch dargestellt, daß durch das Ergebnis der Analyse die Anweisung am Knoten n_5 verändert wird. Die Anweisung nach dem Folgepfeil besagt, daß sich die zweifache Dereferenzierung der Variablen `x` am Knoten n_5 aus der Dereferenzierung der Variablen `z` ergibt, falls zuvor der Pfad π_1 durchlaufen wurde, bzw. aus der Dereferenzierung der Variablen `y`, falls zuvor der Pfad π_2 durchlaufen wurde. Dies entspricht damit Instanzenmengeneigenschaften mit zugehörigen Pfadbedingungen.

3.3.2.1.2 Verwendung von bedingten Erreichbarkeitsproblemen zur Berücksichtigung von Abhängigkeiten

Die Pfadbedingungen π_1 und π_2 für die Ziele der Dereferenzierung der Variablen `x` haben sich direkt aus einem Erreichbarkeitstest ergeben, ohne daß hierfür irgendwelche Beschränkungen der durchlaufenen Teile des Programmgraphen zu berücksichtigen waren. Anschaulich betrachtet liegt dies daran, daß diese Zeigerziele nicht von anderen Analyseergebnissen abhängen.

Das Gegenstück dazu liegt dann vor, wenn man zur weiteren Queryberechnung von der Gültigkeit einer anderen Instanzenmengeneigenschaft ausgehen muß. Diese Situation ergibt sich z.B. im Motivationsbeispiel, wenn man die Ziele der zweifachen Dereferenzierung von `x` ermitteln möchte. Diese ergeben sich entweder aus der einfachen Dereferenzierung von `y` oder von `z`. Eine Query, die von der betrachteten Zuweisung `D = **x` aus rückwärts im Programmgraphen nach dem Ziel der zweifachen Dereferenzierung von `x` suchen soll, muß damit entweder nach der letzten Zuweisung zu `y` oder zu `z` suchen. Dabei besteht hier aber wieder die Gefahr der Bildung von ungültigen Kombinationen. Die letzte Zuweisung zur Variablen `y` ist genau dann relevant, wenn die durch die Pfadbedingung π_2 beschriebene Programminstanz vorliegt. Andernfalls würde man mit $x \rightarrow y$ und $y \rightarrow C$ solche Ziele der Zeigervariablen `x` und `y` kombinieren, die nur in den verschiedenen Programminstanzen, die durch π_1 und π_2 als Pfadbedingungen beschrieben werden, vorkommen.

Eine Lösung zur Vermeidung ungültiger Kombinationen, und aus praktischer Sichtweise eines der zentralen Prinzipien der Analysetechnik, liegt darin, bei der Annahme der Gültigkeit von anderen Instanzenmengeneigenschaften deren Pfadbedingungen zur Beschränkung des weiteren Queryflusses zu verwenden. Im Beispiel ist das Ziel der Zeigervariablen `y` nur dann relevant, wenn man annimmt, daß die Zeigervariable `x` diese als Ziel besitzt. Von dieser Instanzenmengeneigenschaft $x \rightarrow y$ ist aufgrund der bisherigen Analyse bekannt, welche Pfadbedingungen (π_2) sie besitzt. Geht man nun als Annahme davon aus, daß die Variable `x` auf die Variable `y` zeigt, so kann man die Query nach

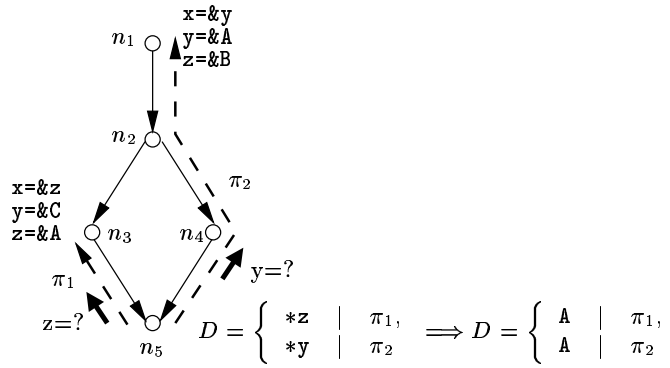


Abbildung 3.2: Transformation von Multi–Level Zeigeranalyse in ein bedingtes Erreichbarkeitsproblem.

den Zielen der zweifachen Dereferenzierung von `x` durch eine Query nach der letzten Zuweisung zu `y` realisieren, und den Fluß dieser Query auf diejenigen Programmbereiche beschränken, die durch die Pfadbedingung π_2 vorgegeben werden. Konkret bedeutet dies im Beispiel, daß die Query nach der letzten Zuweisung zu `y` nur entlang des Pfades π_2, und diejenige nach der letzten Zuweisung zu `z` nur entlang des Pfades π_1 geschickt wird.

Unter einem bedingten Erreichbarkeitsproblem wird damit in dieser Arbeit eine Erweiterung des Erreichbarkeitsproblems verstanden, bei dem beim Durchlaufen des Graphen Bedingungen angetroffen werden, die die Menge der im weiteren Verlauf der Erreichbarkeitsanalyse durchlaufbaren Kanten einschränken. Diese Bedingungen ergeben sich aus den Pfadbedingungen von Instanzenmengeneigenschaften.

In Abbildung 3.2 ist diese Beschränkung der Suche nach Instanzenmengeneigenschaften auf diejenigen Bereiche des Programmes dargestellt, die durch die entsprechenden Pfadbedingungen beschrieben werden. Dies entspricht dabei der Suche nach einer Zuweisung zur Variable z auf dem Pfad π_1 und nach einer Zuweisung zu y auf dem Pfad π_2, in der Abbildung durch die Pfeile mit der Beschriftung z=? und y=? angedeutet. Entlang des Pfades π_1 wird eine Zuweisung `z=&A` gefunden. Also läßt sich daraus schließen, daß bei Durchlaufen des Pfades π_1 die Dereferenzierung der Variable `z` die Variable `A` ergibt, und damit insgesamt auf diesem Pfad die zweimalige Dereferenzierung der Variable `x` am Knoten n_5 die Variable `A` ergeben muß. Ebenso wird entlang des Pfades π_2 eine Zuweisung `y=&A` gefunden. Also ergibt sich auch bei Durchlaufen des Pfades π_2 das gleiche Ergebnis, das rechts im Bild als symbolisiertes Ergebnis der Querybearbeitung eingezeichnet ist. Da die beiden Bedingungen π_1 und π_2 zusammen sämtliche möglichen Pfade durch das Motivationsbeispiel ergeben, kann bei Bedarf zusätzlich daraus gefolgert werden, daß die Variable `D` am Knoten n_5 in jedem Fall den Inhalt der Variablen `A` zugewiesen bekommt, was dem bereits gefundenen exakten Ergebnis entspricht.

3.3.2.2 Theoretische Sichtweise des Analyseverfahrens

Die im vorhergehenden Abschnitt beschriebene vereinfachte Einführung in das Prinzip der Analyse erweckt den Eindruck eines einfachen Algorithmus mit, wie man dies bisher von Erreichbarkeitsproblemen kennt, niedriggradig polynomiellem Berechnungsaufwand. Tatsächlich wurde jedoch bereits ein Ergebnis aus der Literatur zitiert, das besagt, daß das von diesem bedingten Erreichbarkeitsproblem exakt gelöste Problem in die Kategorie der PSPACE–vollständigen informatischen Probleme fällt. Die Darstellung des Algorithmus als bedingtes Erreichbarkeitsproblem ist offensichtlich nicht aussagekräftig und anschaulich genug, um die tatsächliche Schwierigkeit des Problems erfassen zu können. Darüberhinaus erschweren mögliche beliebig häufige Wiederholungen von Symbolen in den als Pfadbedingungen verwendeten Automaten eine klare und anschauliche Definition des Problems der bedingten Grapherreichbarkeit.

Ein weiteres Problem bei der Darstellung des Verfahrens als bedingtes Erreichbarkeitsproblem liegt in der Beweisbarkeit der Korrektheit des Verfahrens. Offensichtlich läßt sich die beschriebene Vorgehensweise nicht in die standardmäßig verwendeten Datenfluß–Frameworks [Kil73][KU77] einordnen, die in der Form von MOP– und MFP–Lösungen von Datenflußgleichungen auf Verbänden ein umfangreiches theoretisches Fundament für verschiedenste Arten von Datenflußanalysen zur Verfügung stellen. Damit muß der Beweis der Korrektheit des Verfahrens ohne Verwendung von bereits bestehenden theoretischen Erkenntnissen geleistet werden. Hierzu ist die direkte Repräsentation des Problems als bedingtes Erreichbarkeitsproblem aber ebenfalls nicht gut geeignet, da diese zunächst keine Aussagen darüber zulässt, wieso eine Beschränkung einer Erreichbarkeitsanalyse auf bestimmte Bereiche eines Graphen im Sinne der Analyse korrekt sein soll.

Als Lösung für diese Probleme wird in dieser Arbeit eine theoretische Fundierung des vorgestellten Verfahrens auf der Basis einer Abstraktion des Erreichbarkeitsproblems vorgestellt. Das Problem der exakten Zeigeranalyse wird durch eine spezielle Art des Schnittes von Sprachen von endlichen Automaten beschrieben. Diese endlichen Automaten besitzen wie bisher beschrieben Eingabesymbole für die Beschreibung von Verzweigungen, sowie zusätzliche Symbole für die Annahme der Gültigkeit einer Instanzenmengeneigenschaft. Mittels einer neuartigen Vorgehensweise, eine variierende Anzahl von Automaten gleichzeitig mit einem gemeinsamen Eingabewort zu betreiben, läßt sich das Zeigeranalyseproblem in der Welt der endlichen Automaten beschreiben und exakt lösen. Aus dieser abstrakteren Perspektive ist es möglich, Aussagen über Kosten der Analyse, gemessen in Zustandsmengengrößen, sowie über worst–case Fälle der Analyse zu treffen. Die Einführung dieser abstrakten Sichtweise wird in Kapitel 5 vorgestellt werden.

Insgesamt werden in dieser Arbeit sowohl die praktische, als auch die theoretische Sichtweise der Analyse verwendet werden. Daß beide Perspektiven ein identisches Verfahren beschreiben, wird in Kapitel 8 bewiesen werden.

3.3.3 Realisierung des Analyseverfahrens

Die folgenden Erläuterungen beziehen sich wieder auf die praktische Sichtweise des Verfahrens, und stellen weitere Details des Verfahrens vor, die bei der Beschreibung des reinen Analyseprinzips in Abschnitt 3.3.2.1 zunächst wegabstrahiert wurden.

3.3.3.1 Eine neue Programmrepräsentation

Um den Weg der Erreichbarkeitsqueries durch den Programmgraphen einfach bestimmen zu können, wird in dieser Arbeit eine neue Programmrepräsentation vorgestellt, die speziell auf die Bearbeitung dieser Queries zugeschnitten ist, und die Erzeugung eines speziellen Analysegraphen als Grundlage für das Erreichbarkeitsproblem ermöglicht. Das Prinzip dieser Programmrepräsentation basiert darauf, komplexe Anweisungen (in Abschnitt 3.1 als Zuweisungen bezeichnet) in sogenannte Grundoperationen zu zerlegen, die jeweils nur eine der folgenden Funktionen erfüllen.

- Auslesen des Inhaltes einer Variablen.
- Verändern des Inhaltes einer Variablen.
- Dereferenzieren eines Zeigers (der Adresse einer Variablen).
- Generieren der Adresse einer Variablen.

Durch diese Zerlegung kann man zunächst Sequenzen von Anweisungen durch Sequenzen von solchen Grundoperationen ausdrücken. Da dabei i.A. Anweisungen in mehrere Grundoperationen aufgespalten werden, muß in dieser neuen Darstellungsweise Information zwischen diesen Grundoperationen transferiert werden, die ansonsten nur implizit bei der Verarbeitung der ursprünglichen Anweisung verwendet werden würde. Beispielsweise wird eine Zuweisung der Form `x=y` durch zwei getrennte Operationen dargestellt, die das Auslesen der Variable `y`, bzw. das anschließende Verändern des Inhaltes der Variablen `x` ausdrücken. Zwischen diesen beiden Operationen muß damit der konkrete Wert, der von der ersten Operation ausgelesen, und von der zweiten Operation als neuem Wert der

Variablen `x` gesetzt werden soll, transferiert werden. Ebenso wird in der neuen Programmdarstellung die Dereferenzierung eines Zeigers von der Verwendung des Ziels der Dereferenzierung getrennt. Dadurch muß auch die Information, welche Variable sich als Ziel der Dereferenzierung ergeben hat, von der Operation, die die Dereferenzierung durchführt, zu derjenigen Operation, die das Ergebnis verwendet, übermittelt werden. Wie dieser Informationstransfer durchgeführt wird, wird in Kapitel 6 detailliert vorgestellt werden. Eine weitere Grundlage für die neue Programmrepräsentation stellt das Wissen dar, welche anderen Operationen vor oder nach einer betrachteten Operation auf eine gemeinsam von diesen Operationen verwendete Variable zugreifen. Aus diesen drei Arten von Abhängigkeiten von Operationen untereinander lassen sich Graphkanten zwischen Graphknoten, die Operationen repräsentieren, erzeugen, die später für die Analyse verwendet werden.

Kontrollflußstrukturen werden durch weitere Operationen beschrieben, auf die in ebenfalls in Kapitel 6 genauer eingegangen wird. Damit läßt sich das gesamte Eingabeprogramm durch Operationen darstellen, zwischen denen Abhängigkeiten bekannt sind, und die dadurch als ein Analysegraph interpretiert und verwendet werden können.

3.3.3.2 Queries

Die Art des Informationstransfers zwischen Operationen liefert eine Grundlage für die Queryverarbeitung. Wird eine Query an einer bestimmten Operation bearbeitet, so ergeben sich aus der Art der betrachteten Operation und deren Abhängigkeiten von anderen Operationen eindeutig die Aktionen aus der folgenden Menge von möglichen Aktionen, die bei der weiteren Bearbeitung der Query durchgeführt werden müssen:

- Erfolgreiche Terminierung der Query mit Rückmeldung an den die Query initiierenden Knoten.
- Weiterleitung der Query an eine oder mehrere andere Operationen, von denen das Ergebnis der betrachteten Operation abhängt.
- Pfadprotokollierung durch dynamische Erzeugung desjenigen Automaten, der später bei Terminierung der Query die Pfadbedingungen beschreibt, während des Flusses der Query.
- Speicherung, welche Queries mit welchen Bedingungen bereits an welchen Operationen bearbeitet wurden.
- Erkennen, ob die aktuell bearbeitete Query eine Wiederholung des Ergebnisses einer bereits bearbeiteten Query ergeben würde, um Schleifen oder Alternativen in den von der Query aufgebauten Automaten einzufügen.
- Hinzufügen eines Automaten zur Query, der das bereits berechnete Ergebnis einer anderen Query beschreibt, von dem die aktuelle Query abhängt, als Bedingung zur Beschränkung des weiteren Queryflusses.
- Betreiben der als Bedingungen mitgeführten Automaten zur Überprüfung, ob ein Pfad in der durch die Automaten beschriebenen Pfadmenge enthalten ist.

Durch die Beschreibung eines Eingabeprogrammes mittels der neuen Programmrepräsentation ist die Auswahl einer oder mehrerer dieser Aktionen zur Ausführung bei der Querybearbeitung nur vom Typ der von der Query aktuell betrachteten Operation abhängig. Damit läßt sich die Analyse in praktischer Sichtweise direkt als bedingtes Erreichbarkeitsproblem beschreiben. Bei der Verwendung von traditionellen Programmgraphen, bei denen bei komplexen Anweisungen wie z.B. `**x = &y` oder `*x = **z` die Bestimmung ihrer Auswirkung auf die Erreichbarkeitsqueries durch Analyse der Anweisung und komplexe Fallunterscheidungen geleistet werden muß, ist dies hingegen nicht der Fall. Dies würde die Beschreibung der Analyse und den Nachweis der Korrektheit deutlich verkomplizieren.

Von jeder Dereferenzierungsoperation aus wird während der Analyse eine Query gestartet, die zunächst einen "leeren" Automaten als Pfadprotokollierung mitführt. Die Reihenfolge, in der die Queries gestartet werden, ergibt sich dabei aus dem Typ der möglichen Zielvariablen. Als erstes

```
main()
{ // b0
  int **x;
  int *y, *z;
  int A, B, C, D;
  A = ...

  x = &y;
  y = &A;
  z = &B;


  while(--)
  { // b1
    if (--)
    {  // b2
       x = &z;
       y = &C;
       z = &A;
    }
    else
    {  // b3
       // leer
    }
  }

  D = **x;

}
```

Abbildung 3.3: Running Example: Um Schleife erweitertes Motivationsbeispiel.

werden Queries von den Dereferenzierungsoperationen aus gestartet, die als Zielvariable Variablen vom Typ "Zeiger auf ... auf Zeiger vom Basistyp" besitzen und als letztes von denjenigen Dereferenzierungsoperationen aus, deren Zielvariablen vom Basistyp sind.

3.4 Einordnung des vorgestellten Verfahrens in die Taxonomie

Das Berechnungsprinzip des vorgestellten Verfahrens läßt sich damit in die Kategorie Pfadinformation—Rückwärts einordnen. Dabei ist durch die Speicherung des in Rückwärtsrichtung gefundenen Ergebnisses am Ausgangsknoten der Query, im obigen Beispiel also am Knoten n_5, eine vorwärtsgerichtete Komponente der Analyse integriert. Über den Grad der Ökonomie des Verfahrens werden in Kapitel 9 ausführliche Untersuchungen durchgeführt werden.

3.5 Running Example

In Abbildung 3.3 ist eine Erweiterung des Motivationsbeispiels zu einem Running Example dargestellt, das für eine exakte Lösung der Analyse von *relational attributes* die Möglichkeit der Behandlung von unendlichen Pfaden benötigt. Das konkret zu berechnende Ergebnis einer Pfadbedingung für $x \rightarrow y$ bzw. $x \rightarrow z$ läßt sich informell wie folgt beschreiben. Die einmalige Dereferenzierung

der Variable `x` bei der Anweisung `D =**x` ergibt die Variable `y`, wenn zuvor entweder die Schleife komplett umgangen wurde oder beliebig häufig wiederholt wurde, wobei aber jeweils nur der else–Zweig der If–Anweisung durchlaufen wurde. Alternativ dazu zeigt `x` auf `z`, wenn im Inneren der Schleife einmal der then–Zweig betreten wurde. Eine anschliessende beliebig häufige Wiederholung der Schleife mit jeweiligem Betreten des else–Zweiges verändert dieses zweite Ergebnis nicht.

3.6 Zusammenfassung

Die zugelassene Programmklasse von Eingabeprogrammen wird so gewählt, daß man exakte Lösungen durch reguläre Sprachen beschreiben kann. Daraus ergibt sich eine Programmklasse ohne Funktionsaufrufe und strukturierte Datentypen, für die eine exakte Lösung von Zeigeranalyse aber immer noch in die Klasse der PSPACE–vollständigen Fragestellungen fällt. Mögliche, zumindest teilweise nicht–reguläre Erweiterungen des Verfahrens sind im Anhang angegeben.

Die Grundidee des Verfahrens basiert auf dem Assoziieren von Analysefakten mit potentiell unendlichen Mengen von Programminstanzen, die durch von endlichen Automaten akzeptierte Sprachen beschrieben werden. Diese Beschreibungen stellen dabei abkürzende Zusammenfassungen von Klassen von Mengen von Programminstanzen dar.

Das Verfahren selbst wird in zwei Sichtweisen vorgestellt werden. Eine theoretische Sichtweise auf der Basis einer variablen Anzahl von parallel betriebenen endlichen Automaten, und eine praktische Sichtweise als bedingtes Erreichbarkeitsproblem auf Graphen. Für letztere wird eine neue Programmrepräsentation eingeführt werden, und das Verfahren in praktischer Sichtweise als Verarbeitung von Queries auf diesen Graphen dargestellt werden.

Kapitel 4

Beschreibung von Eingabeprogrammen

Ein Eingabeprogramm, das den Forderungen aus Abschnitt 3.1 genügt, besteht aus einer Menge von Anweisungen. Die genaue Form dieser Anweisungen, sowie die Beschreibung der Auswirkung dieser Anweisungen durch eine Semantikfunktion soll in diesem Kapitel vorgestellt werden.

4.1 Struktur

4.1.1 Grundlegende Definitionen

Definition 4.1 (Natürliche Zahlen)

Im Rahmen dieser Arbeit seien die Mengen $\mathbb{N}$ *bzw.* $\mathbb{N}_0$ *von natürlichen Zahlen wie folgt definiert:*

$$\begin{aligned} \mathbb{N} &:= \{1, 2, 3, \ldots\} \\ \mathbb{N}_0 &:= \{0, 1, 2, 3, \ldots\} \end{aligned}$$

Definition 4.2 (Variablen und Domäne)

Sei $n \in \mathbb{N}$ *und bezeichne für* $0 \leq i \leq n$ *die Menge* V_i *die Menge aller Variablen mit Referenzierungsstufe* i*, d.h. vom Typ Zeiger auf ... Zeiger auf (*i*–mal) Basistyp. Die Menge* V *bezeichne die Menge aller Variablen und die Menge* D *die Domäne, d.h. die Menge der möglichen Belegungen, aller Variablen* V*.*

4.1.2 Anweisungen und Programmblöcke

Definition 4.3 (Menge aller Anweisungen)
Die Menge $\mathcal{A}$ *sei definiert als die Menge aller Anweisungen gemäß Definitionen 4.5 bis 4.7.*

Definition 4.4 (Programmblock)
Für $n \in \mathbb{N}_0$ *und eine Menge* $a_1, \ldots, a_n$ *mit*

$$\forall 1 \leq i \leq n \; : \; a_i \in \mathcal{A}$$

werde $b = (a_1, \ldots, a_n)$ *(bzw.* $b = \emptyset$ *für* $n = 0$*) als* Programmblock *bezeichnet. Die Menge aller Programmblöcke werde mit* $\mathcal{B}$ *bezeichnet.*

Definition 4.5 (Zuweisung)
Eine Anweisung $a \in \mathcal{A}$ *der Form*

$$a \equiv l = r$$

mit

$$l \equiv \underbrace{*\ldots*}_{n-mal} v$$

und

$$r \equiv \underbrace{*\ldots*}_{m-mal} w$$

mit $n, m \in \mathbb{N}_0$, *bzw.*

$$r \equiv \&w$$

mit jeweils $v, w \in V$, *werde als* Zuweisung *bezeichnet. Dabei wird das Symbol* $\equiv$ *verwendet, um die Gleichheit von Anweisungen und Teilen von Anweisungen vom Zuweisungssymbol der Eingabesprache zu unterscheiden.*
Die Menge $\mathcal{A}_z \subseteq \mathcal{A}$ *mit*

$$\forall a \in \mathcal{A}_z \ : \ a \text{ ist Zuweisung}$$

bezeichne die Menge aller derjenigen Anweisungen, die Zuweisungen nach dieser Definition sind.

Definition 4.6 (If–Anweisung)
Eine Anweisung $a \in \mathcal{A}$ *der Form*

$$a \equiv \texttt{if}_{b_1, b_2}$$

mit $b_1, b_2 \in \mathcal{B}$ *werde als* If–Anweisung *bezeichnet.*

Definition 4.7 (While–Anweisung)
Eine Anweisung $a \in \mathcal{A}$ *der Form*

$$a \equiv \texttt{while}_b$$

mit $b \in \mathcal{B}$ *werde als* While–Anweisung *bezeichnet.*

4.1.3 Programme

Definition 4.8 (Programm)
Ein Programm P *sei definiert als ein Tupel*

$$P = (A, B, V, D, b_0)$$

mit

$A \subseteq \mathcal{A}$	*einer Menge von Anweisungen,*
$B \subseteq \mathcal{B}$	*einer Menge von Programmblöcken,*
V	*einer Menge von Variablen mit o.B.d.A.* $\lvert V \rvert \geq 1$,
D	*der Domäne dieser Variablen,*
$b_0 \in B$	*einem Programmblock.*

4.1.4 Ordnungen auf Mengen von Blöcken und Anweisungen

Für die im nächsten Kapitel vorgestellte Beschreibung von Mengen von Programminstanzen durch endliche Automaten werden folgende Definition von Ordnungen (analog z.B. zur Relation $\leq$ auf $\mathbb{N}$) auf Anweisungen und Programmblöcken benötigt, um die Klassenbildung von Programminstanzenmengen formal zu fundieren.

Im Folgenden wird von einem gegebenen Programm $P = (A, B, V, D, b_0)$ gemäß Definition 4.8 ausgegangen.

Definition 4.9 (Notationsvereinbarung)
Für einen Programmblock $b \in B$ *und eine Anweisung* $a \in A$ *bezeichne*

$$a \in b :\Longleftrightarrow \exists n \in \mathbb{N} : b = (a_1, \ldots, a_n) \wedge \exists i \in \{1, \ldots, n\} : a = a_i$$

das Enthaltensein von Anweisung a *im Programmblock* b.

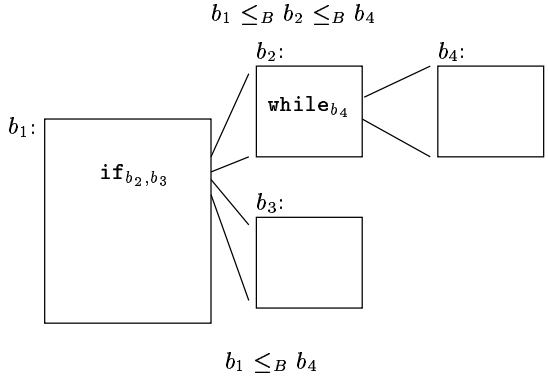


Abbildung 4.1: Ordnung zwischen Programmblöcken.

Definition 4.10 (Unterblock)
Die Relation $\sqsubset_B \subseteq B \times B$ *werde wie folgt definiert. Dabei seien* $b, b' \in B$.

$$\begin{aligned} b \sqsubset_B b' \quad :\Longleftrightarrow \quad & \exists \texttt{if}_{b,b''} \in b' \text{ mit } b'' \in B \\ & \vee \exists \texttt{if}_{b'',b} \in b' \text{ mit } b'' \in B \\ & \vee \exists \texttt{while}_b \in b' \end{aligned}$$

Ein Programmblock $b \in B$, *der mit einem Programmblock* $b' \in B$ *durch* $b \sqsubset_B b'$ *in Relation steht, werde als* Unterblock *des Programmblockes* b' *bezeichnet.*

Definition 4.11 (Definition einer Relation $\leq_B$ zwischen Programmblöcken)
Die Relation $\leq_B \subseteq B \times B$ *werde wie folgt definiert. Dabei seien* $b, b' \in B$.

$$b \leq_B b' \quad :\Longleftrightarrow \quad (b = b') \vee (\exists b_1, \ldots, b_n \text{ mit } n \in \mathbb{N} \text{ und } b' \sqsubset_B b_1 \sqsubset_B \ldots \sqsubset_B b_n \sqsubset_B b)$$

Abbildung 4.1 veranschaulicht diese Definition der Relation $\leq_B$. Dabei sind Programmblöcke jeweils als Rechtecke dargestellt, in denen die darin enthaltenen Anweisungen in der Reihenfolge ihres Auftretens im Programmblock von oben nach unten eingezeichnet sind. Im Programmblock b_1 befindet sich eine If–Anweisung $\texttt{if}_{b_2,b_3}$. Die Programmblöcke b_2 und b_3 sind damit Unterblöcke des Programmblockes b_1 im Sinne von Definition 4.10 und werden als solche dargestellt. Zusätzlich ist ein weiterer Unterblock von Block b_2 eingezeichnet. Nach Definition 4.11 gilt für diese beispielhafte Situation sowohl $b_1 \leq_B b_2 \leq_B b_4$ als auch $b_1 \leq_B b_3$. Die Relation $\leq_B$ auf Programmblöcken (für die im nachfolgenden Satz bewiesen werden wird, daß sie eine Ordnung ist) interpretiert offensichtlich die äußeren Programmblöcke als die kleinsten Elemente. Die Programmblöcke b_3 und b_4 sind bzgl. $\leq_B$ nicht vergleichbar, d.h. es gilt nach Definition 4.11 weder $b_3 \leq_B b_4$ noch $b_4 \leq_B b_3$.

Satz 4.1 ($\leq_B$ ist eine Ordnung auf B)
Die Relation $\leq_B$ *ist eine Ordnung auf* B, *d.h. eine reflexive, transitive und antisymmetrische Relation.*

Beweis:

Reflexivität Z.z.: $\forall b \in B : b \leq_B b$.

Diese Aussage ist aufgrund des ersten Teils der Bedingung in Definition 4.11 erfüllt.

Transitivität Z.z.: $\forall b, b', b'' \in B : b \leq_B b' \wedge b' \leq_B b'' \Longrightarrow b \leq_B b''$.

Seien $b, b', b'' \in B$ beliebig mit $b \leq_B b' \wedge b' \leq_B b''$. Dann gilt entweder $b = b' = b''$ und damit die Behauptung, oder es existieren $b_1, \ldots, b_n \in B$ und/oder $b'_1, \ldots, b'_{n'} \in B$ mit $n, n' \in \mathbb{N}$ und es gilt $b'' \sqsubset_B b'_1 \sqsubset_B \ldots \sqsubset_B b'_{n'} \sqsubset_B b' \sqsubset_B b_1 \sqsubset_B \ldots \sqsubset_B b_n \sqsubset_B b$, bzw. $b'' = b' \sqsubset_B b_1 \sqsubset_B$

$\ldots \sqsubset_B b_n \sqsubset_B b$ oder $b'' \sqsubset_B b_1' \sqsubset_B \ldots \sqsubset_B b_{n'}' \sqsubset_B b' = b$, was in jedem Fall $b \leq_B b''$ zur Folge hat.

Antisymmetrie Z.z.: $\forall b, b' \in B : b \leq_B b' \wedge b' \leq_B b \Longrightarrow b = b'$.

Seien $b, b' \in B$ beliebig mit $b \leq_B b' \wedge b' \leq_B b$. Da aufgrund der Definition 4.10 von $\sqsubset_B$ immer nur $b \sqsubset_B b'$ oder $b' \sqsubset_B b$ gelten kann, muß in Definition 4.11 jeweils $b = b'$ gelten.

Damit gilt die Behauptung. □

Definition 4.12 (Definition einer Relation $\leq_A$ zwischen Anweisungen)
Für je zwei Anweisungen $a, a' \in A$ werde eine Relation $\leq_A \subseteq A \times A$ wie folgt definiert:

$$a \leq_A a' \quad :\Longleftrightarrow \quad \begin{cases} i \leq j, & \textit{falls } \exists b \in B : b = (a_1, \ldots, a_n), n \in \mathbb{N} \\ & \textit{und } \exists i, j \in \{1, \ldots, n\} : a = a_i \wedge a' = a_j \\ k < l, & \textit{falls } \exists b \in B : b = (a_1, \ldots, a_n), n \in \mathbb{N} \textit{ mit } \exists k \in \{1, \ldots, n\} : a_k = a \\ & \textit{und } \exists b_1 \ldots, b_m \in B,\ m \in \mathbb{N}\ \exists l \in \{1, \ldots, n\} : \\ & (\texttt{while}_{b_1} = a_l \vee \texttt{if}_{b_1, \ldots} = a_l \vee \texttt{if}_{\ldots, b_1} = a_l) \\ & \wedge b_m \sqsubset_B \ldots \sqsubset_B b_1 \wedge a' \in b_m \\ l < k, & \textit{falls } \exists b \in B : b = (a_1, \ldots, a_n), n \in \mathbb{N} \textit{ mit } \exists k \in \{1, \ldots, n\} : a_k = a' \\ & \textit{und } \exists b_1 \ldots, b_m \in B,\ m \in \mathbb{N}\ \exists l \in \{1, \ldots, n\} : \\ & (\texttt{while}_{b_1} = a_l \vee \texttt{if}_{b_1, \ldots} = a_l \vee \texttt{if}_{\ldots, b_1} = a_l) \\ & \wedge b_m \sqsubset_B \ldots \sqsubset_B b_1 \wedge a \in b_m \\ i < j, & \textit{falls } \exists b \in B : b = (a_1, \ldots, a_n), n \in \mathbb{N} \\ & \textit{und } \exists b_1' \ldots, b_m', b_1'' \ldots, b_p'' \in B,\ m, p \in \mathbb{N}\ \exists i, j \in \{1, \ldots, n\} : \\ & (\texttt{while}_{b_1'} = a_i \vee \texttt{if}_{b_1', \ldots} = a_i \vee \texttt{if}_{\ldots, b_1'} = a_i) \\ & \wedge b_m' \sqsubset_B \ldots \sqsubset_B b_1' \wedge a \in b_m' \\ & \wedge (\texttt{while}_{b_1''} = a_j \vee \texttt{if}_{b_1'', \ldots} = a_j \vee \texttt{if}_{\ldots, b_1''} = a_j) \\ & \wedge b_p'' \sqsubset_B \ldots \sqsubset_B b_1'' \wedge a' \in b_p'' \end{cases}$$

Die verschiedenen Fälle bedeuten dabei, daß beide Anweisungen a und a' im gleichen Programmblock zu finden sind (erster Fall), eine Anweisung in einem Programmblock zu finden ist, von dem ausgehend ein Unterblock die andere Anweisung enthält (Fälle zwei und drei), bzw. beide Anweisungen in verschiedenen, von verschiedenen `if` bzw. `while`–Anweisungen ausgehenden Unterblöcken enthalten sind, die einen gemeinsamen Oberblock b besitzen (Fall vier). Die vier Fälle aus dieser Definition sind in Abbildung 4.2(a) bis (d) veranschaulicht. In Teilabbildung (e) ist eine Situation gezeigt, in der die beiden Anweisungen a und a' bzgl. $\leq_A$ nicht vergleichbar sind.

Satz 4.2 ($\leq_A$ ist eine Ordnung auf A)
Die Relation $\leq_A$ ist eine Ordnung auf A, d.h. eine reflexive, transitive und antisymmetrische Relation.

Beweis: Da die nachfolgend betrachteten Anweisungen nur in nicht–leeren Programmblöcken vorkommen können, wird im Folgenden von Programmblöcken $b \in B$ der Form $b = (a_1, \ldots, a_n)$ mit $n \in \mathbb{N}$ (anstatt $\mathbb{N}_0$) ausgegangen.

Reflexivität Z.z.: $\forall a \in A : a \leq_A a$.

Sei $a \in A$ beliebig. Dann $\exists b \in B$ mit $b = (a_1, \ldots, a_n)$, $n \in \mathbb{N}$, sowie $\exists i \in \{1, \ldots, n\}$ mit $a = a_i$. Mit $j := i$ gilt dann $a = a_i = a_j$ mit $i \leq j$, also gilt $a \leq_A a$.

Transitivität Z.z.: $\forall a, a', a'' \in A : a \leq_A a' \wedge a' \leq_A a'' \Longrightarrow a \leq_A a''$.

Seien $a, a', a'' \in A$ beliebig und gelte $a \leq_A a' \wedge a' \leq_A a''$.

Fall 1: $a \leq_A a'$ nach Fall 1 der Definition 4.12, d.h. $\exists b = (a_1, \ldots, a_n) \in B$, $n \in \mathbb{N}$, und $\exists i, j \in \{1, \ldots, n\}$ mit $a = a_i$, $a' = a_j$ und $i \leq j$.

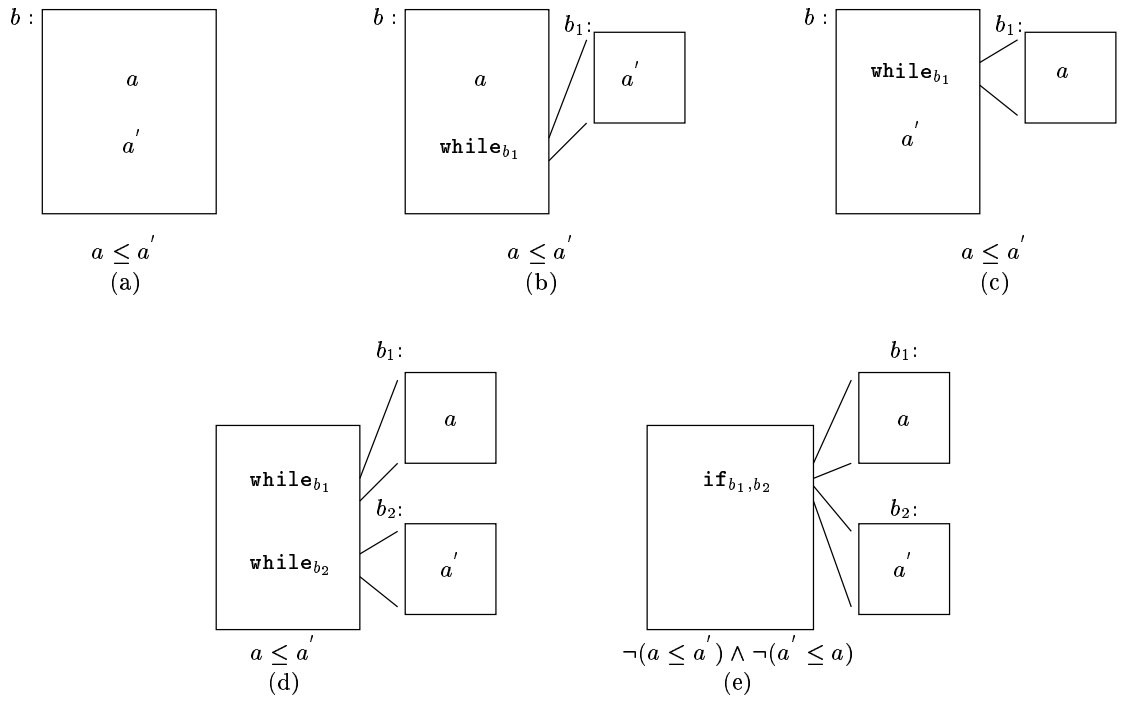


Abbildung 4.2: Ordnung zwischen Anweisungen.

Fall a: $a' \leq_A a''$ nach Fall 1 der Definition 4.12, d.h. $\exists b' = (a'_1, \ldots, a'_{n'}) \in B$, $n' \in \mathbb{N}$, und $\exists k, l \in \{1, \ldots, n'\}$ mit $a' = a'_k$, $a'' = a'_l$ und $k \leq l$.
Da die Anweisung a' nur in einem Programmblock vorkommen kann, gilt $b = b'$, d.h. $a \in b$, $a' \in b$ und $a'' \in b$ mit $a_i = a$, $a_j = a_k = a'$ und $a_l = a''$. Aus $i \leq k \leq l$ folgt $i \leq l$ und damit gilt nach Fall 1 von Definition 4.12 $a \leq_A a''$.

Fall b: $a' \leq_A a''$ nach Fall 2 der Definition 4.12. Da a' wiederum nur in einem Programmblock liegen kann, bedeutet dies, daß $a_j = a_k$ und damit $j = k$, und $\exists b_1 \ldots, b_m \in B\ \exists m \in \mathbb{N}\ \exists l \in \{1, \ldots, n\} : (\texttt{while}_{b_1} = a_l \vee \texttt{if}_{b_1,\ldots} = a_l \vee \texttt{if}_{\ldots,b_1} = a_l) \wedge b_m \sqsubset_B \ldots \sqsubset_B b_1 \wedge a' \in b_m$ mit $j < l$. Damit gilt auch $i < l$ und deswegen $a \leq_A a''$ nach Fall 2 der Definition 4.12.

Fall c: $a' \leq_A a''$ nach Fall 3 der Definition 4.12. Dann $\exists b' = (a'_1, \ldots, a'_{n'}) \in B$, $n' \in \mathbb{N}$ und $\exists k \in \{1, \ldots, n'\}$ mit $a'' = a'_k$, sowie $\exists b_1 \ldots, b_m \in B$, $m \in \mathbb{N}\ \exists l \in \{1, \ldots, n'\} : (\texttt{while}_{b_1} = a'_l \vee \texttt{if}_{b_1,\ldots} = a'_l \vee \texttt{if}_{\ldots,b_1} = a'_l) \wedge b_m \sqsubset_B \ldots \sqsubset_B b_1 \wedge a' \in b_m$ mit $l < k$.
Da nach Voraussetzung $a \leq_A a'$ aufgrund von Fall 1 der Definition gilt, müssen a und a' im gleichen Programmblock $b = b_m$ liegen. Damit gilt die Aussage für a' auch für a, d.h. $a \leq_A a''$.

Fall d: $a' \leq_A a''$ nach Fall 4 der Definition 4.12. Analog zu Fall 1c) gilt hier ebenfalls, daß a und a' im gleichen Programmblock zu finden sind, weshalb sich auch hier die Aussage $a' \leq_A a''$ auf $a \leq_A a''$ übertragen läßt.

Fall 2: $a \leq_A a'$ nach Fall 2 der Definition 4.12, d.h. $\exists b \in B : b = (a_1, \ldots, a_n), n \in \mathbb{N}$ mit $\exists k \in \{1, \ldots, n\} : a_k = a$ und $\exists b_1 \ldots, b_m \in B$, $m \in \mathbb{N}\ \exists l \in \{1, \ldots, n\} : (\texttt{while}_{b_1} = a_l \vee \texttt{if}_{b_1,\ldots} = a_l \vee \texttt{if}_{\ldots,b_1} = a_l) \wedge b_m \sqsubset_B \ldots \sqsubset_B b_1 \wedge a' \in b_m$, sowie $k < l$.

Fall a: $a' \leq_A a''$ nach Fall 1 der Definition 4.12.
Damit sind a' und a'' im gleichen Programmblock zu finden und es gilt analog zu Fall 1c) $a \leq_A a''$.

Fall b: $a' \leq_A a''$ nach Fall 2 der Definition 4.12. Damit gilt $\exists b' = (a'_1, \ldots, a'_{n'}) \in B$, $n' \in \mathbb{N}$ und $\exists k' \in \{1, \ldots, n'\}$ mit $a' = a'_{k'}$, sowie $\exists b'_1 \ldots, b'_{m'} \in B$, $m' \in \mathbb{N}$ $\exists l' \in \{1, \ldots, n'\} : (\texttt{while}_{b'_1} = a'_{l'} \vee \texttt{if}_{b'_1,\ldots} = a'_{l'} \vee \texttt{if}_{\ldots,b'_1} = a'_{l'}) \wedge b'_{m'} \sqsubset_B \ldots \sqsubset_B b'_1 \wedge a'' \in b'_{m'}$ mit $k' < l'$.

Da $a' \in b_m$ und $a' \in b'$ gilt, muß $b_m = b'$ gelten. Daher kann man folgern, daß $b'_{m'} \sqsubset_B \ldots \sqsubset_B b'_1 \sqsubset_B b' = b_m \sqsubset_B \ldots \sqsubset_B b_1$ mit $a'' \in b'_{m'}$ gilt, weswegen mit $k < l$ die Voraussetzungen für Fall 2 von Definition 4.12 erfüllt sind, d.h. es gilt $a \leq_A a''$.

Fall c: $a' \leq_A a''$ nach Fall 3 der Definition 4.12. Dann $\exists b' = (a'_1, \ldots, a'_{n'}) \in B$, $n' \in \mathbb{N}$ und $\exists k' \in \{1, \ldots, n'\}$ mit $a'' = a'_{k'}$, sowie $\exists b'_1 \ldots, b'_{m'} \in B$, $m' \in \mathbb{N}$ $\exists l' \in \{1, \ldots, n'\} : (\texttt{while}_{b'_1} = a'_{l'} \vee \texttt{if}_{b'_1,\ldots} = a'_{l'} \vee \texttt{if}_{\ldots,b'_1} = a'_{l'}) \wedge b'_{m'} \sqsubset_B \ldots \sqsubset_B b'_1 \wedge a' \in b'_{m'}$ mit $l' < k'$.

Insgesamt gilt damit $a' \in b'_{m'}$ und $a' \in b_m$, d.h. $b'_{m'} = b_m$. Da jeder Programmblock laut Definition nur direkter Unterblock eines einzigen anderen Programmblockes sein kann, müssen daher die $b'_{i'}$ und b_i für $i' \in \{1, \ldots, m'\}$ und $i \in \{1, \ldots, m\}$ zumindest teilweise übereinstimmen, d.h. es gilt $b'_{m'} = b_m, b'_{m'-1} = b_{m-1}, \ldots$ und damit einer der folgenden Fälle.

Fall i: $b = b'$ (d.h. $m = m'$). Dann gilt $a_k = a$ und $a_{k'} = a''$ mit $k < l = l' < k'$, also gilt $a \leq_A a''$ nach Fall 1 der Definition 4.12.

Fall ii: $b' \sqsubset_B \ldots \sqsubset_B b$, d.h. $\exists i \in \{1, \ldots, m\}$ mit $b_i = b'$ (und damit $m' < m$). Damit ist dann aufgrund von $k < l$ die Voraussetzung für Fall 2 der Definition 4.12 erfüllt, d.h. $a \leq_A a''$.

Fall iii: $b \sqsubset_B \ldots \sqsubset_B b'$, d.h. $\exists i' \in \{1, \ldots, m'\}$ mit $b'_{i'} = b$ (und damit $m < m'$). Analog zu Fall ii) ist aufgrund von $k < l'$ die Voraussetzung für Fall 3 der Definition 4.12 erfüllt, d.h. $a \leq_A a''$.

Fall d: Analog zu Fall 2c).

Fall 3: $a \leq_A a'$ nach Fall 3 der Definition 4.12.

Aufgrund der Ähnlichkeit der Fälle lassen sich diese analog aus den Fällen 2a) bis 2d) ableiten.

Fall 4: $a \leq_A a'$ nach Fall 4 der Definition 4.12.

Die Fälle 4a) bis 4c) lassen sich analog zu den Fällen 1d), 2d) und 3d) zeigen. Der Fall 4d) ergibt sich aus einer zu Fall 2c) ähnlichen Argumentation.

Antisymmetrie Z.z.: $\forall a, a' \in A : a \leq_A a' \wedge a' \leq_A a \Longrightarrow a = a'$.

Seien $a, a' \in A$ und gelte $a \leq_A a' \wedge a' \leq_A a$. Gilt für zwei Anweisungen $a_1, a_2 \in A$, daß nach Fall 2 oder 3 der Definition 4.12 $a_1 \leq_A a_2$ ist, so kann für diese Anweisungen aufgrund der eindeutigen Zuordnung in den beiden Fällen, welche Anweisung in einem Unterblock des Programmblockes liegt, in dem die andere Anweisung zu finden ist, nicht ebenfalls $a_2 \leq_A a_1$ aufgrund von Fall 2 oder 3 gelten. Damit kann also unter der Voraussetzung, daß $a_1 \leq_A a_2$ und $a_2 \leq_A a_1$ gilt, ausgeschlossen werden, daß beide Eigenschaften aufgrund von Fall 2/3 von Definition 4.12 gelten. Durch die unterschiedliche vorausgesetzte Blockstruktur der Fälle 2 und 3 im Gegensatz zu den Fällen 1 und 4 in Definition 4.12 lassen sich diese nicht kombinieren, um gleichzeitige Gültigkeit von $a_1 \leq_A a_2$ und $a_2 \leq_A a_1$ nach jeweils Fall 1/4 bzw. 2/3 der Definition zu erreichen. Daher kann auch nicht eine der Eigenschaften aufgrund von Fall 2/3 und die andere Eigenschaft aufgrund von Fall 1/4 gelten. Damit bleiben nur noch die Möglichkeit, daß die Eigenschaften jeweils aufgrund von Fall 1/4 der Definition gelten. Eine Kombination von Fall 1 mit Fall 4 scheidet aufgrund der Tatsache, daß nur in einem Fall Unterblöcke vorkommen, wegen der damit unterschiedlichen Voraussetzungen über die Ausgangssituation der Blockstruktur ebenfalls aus.

Soll für zwei Anweisungen $a_1, a_2 \in A$ jeweils nach Fall 4 gleichzeitig $a_1 \leq_A a_2$ und $a_2 \leq_A a_1$ gelten, so müssen die beiden Anweisungen aus Symmetriegründen in verschiedenen Unterblöcken

einer gemeinsamen `if`–Anweisung zu finden sein. In diesem Fall müsste in Definition 4.12, Fall 4 aber $i = j$ gelten, was in der Definition ausgeschlossen wird. Damit kann die Voraussetzung auch dann nicht gelten, wenn beidesmal Fall 4 von Definition 4.12 angenommen wird.

Also kann im Fall der gleichzeitigen Gültigkeit von $a \leq_A a'$ und $a' \leq_A a$ nur beidesmal Fall 1 der Definition 4.12 zutreffen. Damit kann man dann $i \leq j$ und $j \leq i$ schließen, weshalb $i = j$ und $a = a_i = a_j = a'$ und damit die Behauptung gelten muß.

□

4.2 Semantik

4.2.1 Prinzip von traditionellen Semantikfunktionen

Die Auswirkung von Anweisungen wird i.A. durch Definition einer Semantikfunktion angegeben. Die folgende Darstellungsweise entspricht dabei einer häufig in der Literatur verwendeten Beschreibungsform.

Definition 4.13 (Variablenbelegung)
Eine Funktion $\sigma : V \to D$ werde als Variablenbelegung bezeichnet. Die Menge Σ bezeichne die Menge aller Variablenbelegungen.

Definition 4.14 (Semantikfunktion)
Eine Funktion $[\![\ \ ]\!]$ mit

$$[\![\ \ ]\!] : \mathcal{A}_z \to (\Sigma \to \Sigma)$$

werde als Semantikfunktion bezeichnet. Die Anwendung $[\![\ \ ]\!](a)$ der Semantikfunktion auf eine Anweisung $a \in A$ wird im Folgenden stets durch $[\![\ a\ ]\!]$ ausgedrückt werden.

Für die Teilmenge von Anweisungen $a \in \mathcal{A}_z$ eines Eingabeprogrammes, die Zuweisungen repräsentieren, beschreibt $[\![\ a\ ]\!](\sigma) = \sigma'$ die Anwendung der Semantikfunktion auf eine Anweisung a, deren Ausführung die Variablenbelegung σ in eine Variablenbelegung σ' verändert. In der Terminologie aus der Literatur bezeichnet eine solche Semantikfunktion eine lokale Semantik.

Definition 4.15 (Notationsvereinbarung)
Für eine Variablenbelegung $\sigma \in \Sigma$ und $x, y \in V$, sowie $s \in D$ bezeichne $\sigma[x \to s]$ abkürzend eine Variablenbelegung σ' mit der Eigenschaft

$$\sigma'(y) = \sigma[x \to s](y) := \begin{cases} s, & \textit{falls } x = y \\ \sigma(y), & \textit{sonst} \end{cases},$$

d.h. eine Variablenbelegung bei der der Variablen `x` *der Wert s zugewiesen wurde.*

4.2.2 Konkrete Semantikfunktion für Eingabeprogramme

Definition 4.16 (Dereferenzierung)
Die Funktion dereference *sei wie folgt definiert*

$$\text{dereference} : \begin{cases} V \times \mathbb{N}_0 \times \Sigma & \to & D \\ (v, n, \sigma) & \mapsto & \begin{cases} v, & \textit{falls } n = 0 \\ \sigma(\text{dereference}(v, n-1, \sigma)), & \textit{falls } n > 0 \end{cases} \end{cases}$$

Die Funktion dereference bewirkt also eine ggfs. mehrfache Anwendung der Variablenbelegung σ zur Berechnung des Ergebnisses der n–fachen Dereferenzierung von Zeigervariablen.

Definition 4.17 (Semantikfunktion für Zuweisungen)
Für eine Zuweisung $a \in \mathcal{A}_z$ gemäß Definition 4.5 werde eine Semantikfunktion wie folgt definiert.

Fall 1: $a \equiv \underbrace{*\ldots*}_{n-mal} v = \underbrace{*\ldots*}_{m-mal} w$ *mit* $v, w \in V$ *und* $m, n \in \mathbb{N}_0$

$$[\![\ a\ ]\!](\sigma) \quad := \quad \sigma[\text{dereference}(v, n, \sigma) \rightarrow \sigma(\text{dereference}(w, m, \sigma)]$$

Fall 2: $a \equiv \underbrace{*\ldots*}_{n-mal} v = \&w$ *mit* $v, w \in V$ *und* $n \in \mathbb{N}_0$

$$[\![\ a\ ]\!](\sigma) \quad := \quad \sigma[\text{dereference}(v, n, \sigma) \rightarrow w]$$

Diese Semantikfunktion wird in Kapitel 6 dazu verwendet werden, um die Äquivalenz der neuen Programmrepräsentation mit der hier vorgestellten traditionellen Programmrepräsentation zu zeigen.

4.3 Zusammenfassung

Für die in Abschnitt 3.1 informell eingeführte zugelassene Klasse von Eingabeprogrammen wurde eine formale Definition von Struktur und Semantik angegeben. Zusätzlich wurden Ordnungen auf den Mengen von Anweisungen und Programmblöcken von Eingabeprogrammen definiert, die für die Beschreibung von Klassen von Programminstanzenmengen in den nachfolgenden Kapiteln benötigt werden.

Kapitel 5

Programminstanzenbeschreibung durch Automaten

Dieses Kapitel befasst sich mit der Darstellung von Mengen von Programminstanzen. Diese Darstellungen und ihre Verwendungen stellen einen zentralen Aspekt dieser Arbeit und eine neue Sichtweise von statischer Programmanalyse dar. Wie bereits in Kapitel 3 erwähnt wurde, ist die Betrachtungsweise von Programminstanzen in dieser Arbeit entgegen der Ausführungsrichtung gerichtet. In der Terminologie aus Kapitel 2 könnte man diese Art von Programminstanzenbeschreibungen als *reverse histories* bezeichnen.

5.1 Überblick

Jede von einer Programminstanz gewählte Verzweigung wird durch ein bestimmtes Symbol ausgedrückt. Weitere Symbole beschreiben die Gültigkeit von bestimmten Instanzenmengeneigenschaften eines Programmes an Anweisungen. Damit wird erreicht, daß ein Pfad durch ein Programm, auf dem die Gültigkeit von bestimmten Instanzenmengeneigenschaften angenommen wird, durch eine Sequenz von Symbolen repräsentiert werden kann. Als zentrales Prinzip dieser Arbeit werden diese Sequenzen von Symbolen durch Sprachen dargestellt, die durch endliche Automaten beschrieben werden. In welcher Form diese Beschreibung geschieht, wird in Definition 5.11 formal eingeführt werden.

5.1.1 Blockbasierte Betrachtungsweise

Im Gegensatz zu vielen Ansätzen aus der Literatur sollen Programmpfade hier nicht als Sequenz von durchlaufenen einzelnen Anweisungen, sondern durch eine Protokollierung der durchlaufenen Programmblöcke, und damit von Gruppen von Anweisungen beschrieben werden.

Damit wird das Betreten und Verlassen von Programmblöcken zum zentralen Element von Pfadbeschreibungen. Die Alternative, die ausgeführten Anweisungen zur Pfadbeschreibung zu verwenden, wie dies in der Literatur oft bei der theoretischen Beschreibung von Analysen getan wird, würde bei einer expliziten Pfadprotokollierung dagegen zu Problemen führen, indem verschiedene Teilanalysen, die aufgrund von Optimierungen bei der Implementierung nicht die gleiche Menge von Anweisungen entlang des gleichen Pfades betrachten müssen, anscheinend verschiedene Pfade durchlaufen würden. Darüberhinaus wäre der Speicher– und Verwaltungsaufwand für eine solche Pfadprotokollierung deutlich höher, als bei dieser blockbasierten Darstellung. Durch letztere wird hingegen die Einheitlichkeit der Pfadbeschreibungen für verschiedene Unteranalysen ermöglicht, da das Betreten eines Blockes unabhängig von den konkreten darin betrachteten Anweisungen ist, jede Anweisung aber eindeutig einem Block zuzuordnen ist, in dem sie enthalten ist.

5.1.2 Klassenbildung von Programminstanzenmengen

Ein Wort aus der von einem Automaten beschriebenen Sprache beschreibt einen Programmpfad in Rückwärtsrichtung, ausgehend von einer betrachteten Anweisung. Mit dem letzten Symbol des Wortes wird aber i.A. nicht der Anfang des beschriebenen Programmes erreicht werden. Vielmehr ist i.A. die Gültigkeit einer Eigenschaft wie $x \to y$ nur vom zuletzt vor Erreichen der betrachteten Anweisung durchlaufenen Programmpfadabschnitt abhängig. Dem trägt die Interpretation einer durch einen Automaten spezifizierten Menge von Programmpfaden als eine Klasse von Programminstanzen Rechnung. Dabei werden sämtliche Programminstanzen, die einen bestimmten Pfadabschnitt vor Erreichen einer untersuchten Anweisung durchlaufen, als bezüglich der untersuchten Eigenschaft äquivalent angesehen. Konkret bedeutet dies, daß durch die Beschreibung eines Endabschnittes von Programmpfaden vor Erreichen einer Anweisung sämtliche Programminstanzen zusammenfassend beschrieben werden, die einen beliebigen Verlauf ab dem Programmstart haben, dann aber vor Erreichen der untersuchten Anweisung genau einen solchen Endabschnitt von Pfaden durchlaufen.

Eine weitere Form von Klassen von Programminstanzen wird aufgrund der Beobachtung verwendet, daß manche Eigenschaften unabhängig von Zwischenabschnitten von Programmpfaden sind. Zum Beispiel kann es vorkommen, daß eine Variable in einer If–Anweisung weder im then– noch im else–Zweig verwendet wird. Die Dereferenzierung des Inhaltes dieser Variablen ist daher unabhängig davon, ob zuvor der then– oder der else–Zweig ausgeführt wurde. Im Rahmen dieser Arbeit wird dies dadurch berücksichtigt, daß ein Wort, das eine Programminstanz beschreibt, in der diese Dereferenzierung ein bestimmtes Ziel ergibt, kein Symbol für diese Verzweigung enthält. Damit repräsentiert ein solches Wort auch eine Menge von Programminstanzen, in denen "dazwischen" beliebige Pfade durchlaufen werden können.

Die formale Fundierung dieser Vorgehensweise wird in Abschnitt 5.5.2 geleistet werden. Dabei werden Voraussetzungen angegeben werden, die Teilfolgen von Symbolen eines Wortes erfüllen müssen, um korrekte Beschreibungen von solchen Klassen zu sein.

5.2 Grundlegende Definitionen

Das in dieser Arbeit vorgestellte Verfahren verwendet Aussagen über Mengen von Programminstanzen, um zwischen gültigen und nicht–gültigen Kombinationen von Analysefakten zu unterscheiden. Um dies realisieren zu können, werden im folgenden Abschnitt einige Begriffe definiert, die zu einer Beschreibung von Programminstanzen durch Sprachen im Weiteren benötigt werden.

Im Folgenden wird stets von einem gegebenen Eingabeprogramm $P = (A, B, V, D, b_0)$ gemäß Definition 4.8 ausgegangen.

5.2.1 Programmeigenschaften

Prinzipiell entsprechen Analysefakten dem Wissen, daß an einer Anweisung eine bestimmte Eigenschaft gilt, bzw. gelten kann. Die formale Beschreibung von Programminstanzen soll die Gültigkeit von solchen Eigenschaften ausdrücken können, und erfordert daher eine Definition von solchen Eigenschaften, die als Programmeigenschaften bezeichnet werden.

Definition 5.1 (Programmeigenschaft)
Für eine Anweisung $a \in A$ bezeichnet ρ_a eine Programmeigenschaft, *die an Anweisung a gilt. Die Menge aller solchen Eigenschaften, die an Anweisungen $a \in A$ betrachtet werden können, werde mit $\mathcal{H}_A$ bezeichnet.*

Eine konkrete Programmeigenschaft ρ_a könnte zum Beispiel das Ziel der Zeigervariablen x bei Anweisung a sein. Um die Beschreibung nicht zu verkomplizieren, wird im Folgenden vereinbart:

Definition 5.2 (Vereinbarung)
O.B.d.A. soll an einer Anweisung a immer nur maximal eine Programmeigenschaft ρ_a gelten. Für komplizierte Zuweisungen, an denen mehrere Zeigerziele relevant sind, können diese Eigenschaften

auf zusätzliche Anweisungen, z.B. der Form $v = v$ mit $v \notin V$ (d.h. einer neuen Variablen, die bisher nicht in der Menge der Variablen des Eingabeprogrammes vorkommt) verteilt werden.

Mit der Einführung der neuen Programmrepräsentation in Kapitel 6, und der Beschreibung von Programmeigenschaften auf der Basis dieser Darstellungsweise, wird diese Vereinbarung per Konstruktion erfüllt sein. Daher wird hier nicht explizit auf die Menge aller für die Analyse relevanten Programmeigenschaften bei komplexen Anweisungen eingegangen. Stattdessen wird zunächst allgemein betrachtet, welche Zusammenhänge zwischen gegebenen Programmeigenschaften und Programmpfaden bestehen können, und die exakte Menge von für die Analyse relevanten Programmeigenschaften später definiert werden.

Definition 5.3 (Möglichkeiten für eine Programmeigenschaft)
Eine Abbildung poss : $\mathcal{H}_A \to \mathcal{P}(D)$ *werde als* Möglichkeitenfunktion *einer Programmeigenschaft bezeichnet. Für die in dieser Arbeit betrachteten Programmeigenschaften $\rho_a \in \mathcal{H}_A$ soll die Menge* $\text{poss}(\rho_a)$ *stets endlich sein.*

Für obige Programmeigenschaft ρ_a, die das Ziel der Zeigervariablen x bei Anweisung a beschreibt, könnte z.B. $\text{poss}(\rho_a) = \{y, z\}$ gelten, d.h. die möglichen Ziele dieser Zeigervariablen wären damit die Variablen $y, z \in D$.

Definition 5.4 (Annehmen einer Möglichkeit einer Programmeigenschaft)
Für eine Programmeigenschaft $\rho_a \in \mathcal{H}_A$ und $y \in \text{poss}(\rho_a)$ bezeichnet $\rho_a(y)$ das Annehmen der Möglichkeit y von Eigenschaft ρ_a bei Anweisung $a \in A$.

Definition 5.5 (Schreibweise für Zeigerziele)
Für Programmeigenschaften $\rho_a \in \mathcal{H}$, die das Annehmen eines Zieles von einer Zeigervariable $x \in V$ (oder Dereferenzierungen davon) an einer Anweisung $a \in A$ ausdrücken, werde $\rho_a(y)$ durch $(x \to y)_a$ beschrieben. Wenn die Anweisung a aus dem Zusammenhang ersichtlich ist, kann $\rho_a(y)$ auch als $x \to y$ geschrieben werden.

Definition 5.6 (Mögliche geltende Eigenschaften an einer Anweisung)
Die Abbildung $\text{props} : A \to (\mathcal{H}_A \cup \emptyset)$ *sei definiert durch*

$$\text{props}(a) := \begin{cases} \rho_a, & \textit{falls } \exists \rho_a \in \mathcal{H}_A, \\ \emptyset, & \textit{sonst} \end{cases}$$

Damit beschreibt die Abbildung props zu einer gegebenen Anweisung $a \in A$, welche Eigenschaft(en) an dieser Anweisung gelten können.

5.2.2 Programmpfade

Obwohl das Durchlaufen bestimmter Programmpfade im weitesten Sinne ebenfalls als Programmeigenschaft anzusehen ist, wird im weiteren Verlauf eine unterschiedliche Verwendung von Programmeigenschaften und Programmpfaden offensichtlich werden, so daß das Durchlaufen von Programmpfaden hier eine eigene Beschreibungsform erhält.

Definition 5.7 (Mögliches Betreten von Programmblöcken)
Bezeichnet die Menge

$$\overline{B} := B \cup \{\overline{b} \mid b \in B\}$$

eine Erweiterung der Menge B um zusätzliche Elemente, so kann man die Abbildung branch : $A \to \mathcal{P}(\overline{B})$ *wie folgt definieren:*

$$\text{branch}(a) := \begin{cases} \{b_1, b_2\}, & \textit{falls } a \equiv \texttt{if}_{b_1,b_2} \textit{ für } b_1, b_2 \in B, \\ \{b, \overline{b}\}, & \textit{falls } a \equiv \texttt{while}_b \textit{ für } b \in B, \\ \emptyset, & \textit{sonst} \end{cases}$$

Die Vereinigung dieser Elemente über alle Anweisungen aus A werde abkürzend wie folgt bezeichnet:

$$\text{branch}(A) := \bigcup_{a \in A} \text{branch}(a)$$

Die Abbildung branch beschreibt damit für eine Anweisung $a \in A$ die Menge der an dieser Anweisung möglicherweise zu betretenden Programmblöcke, bzw. im Fall eines Symbols der Form $\overline{b}$ das explizite Nicht–Betreten eines Schleifenrumpfes (in Rückwärtsbetrachtungsrichtung).

5.2.3 Programminstanzen

Basierend auf den Definitionen von Programmeigenschaften und Programmpfaden kann man nun Programminstanzen wie folgt definieren.

Definition 5.8 (Programminstanz und Programminstanzbeschreibung)
Eine Ausführung eines Eingabeprogrammes P bis zum Erreichen einer Anweisung $a \in A$, bei der die übliche Funktionalität von `if`*– und* `while`*–Anweisungen zugrundegelegt wird, bei denen die Verzweigungsbedingungen wie in der Literatur üblich nicht interpretiert werden, werde als Programminstanz bezeichnet. Eine Beschreibung des von einer Programminstanz durchlaufenen Programmpfades mittels der Elemente der Menge aus Definition 5.7, sowie der möglicherweise in diesen Programminstanzen geltenden Programmeigenschaften nach Definition 5.4, werde als* Programminstanzenbeschreibung *bezeichnet.*
In einer solchen Beschreibung wird neben den von einer Programminstanz durchlaufenen Pfaden das Annehmen von Möglichkeiten von Programmeigenschaften ausgedrückt. Wenn diese in der Programminstanz auf der Basis der Semantikfunktion tatsächlich angenommen werden, dann werde eine solche Darstellung als widerspruchsfreie Programminstanzenbeschreibung *bezeichnet.*
Wenn die genaue Bedeutung aus dem Kontext hervorgeht, wird im Folgenden eine Programminstanzenbeschreibung auch als Programminstanz bezeichnet werden.

Eine Programminstanzenbeschreibung nach Definition 5.8 ist eine eindeutige Kennzeichnung einer Programminstanz in dem Sinne, daß die Beschreibung der durchlaufenen Programmpfade diese eindeutig identifizieren. Die Beschreibung von möglicherweise in diesen Programminstanzen gültigen Programmeigenschaften ist hingegen nach dieser Definition nicht notwendigerweise eindeutig. Da es für Zeigeranalyse ein PSPACE–vollständiges Problem ist, sämtliche in den Programminstanzen angenommenen Möglichkeiten von Programmeigenschaften in der Form von Zeigerzielen herauszufinden, lockert diese Definition die Forderung der Exaktheit an eine Programminstanzenbeschreibung, was die Beschreibung des Annehmens von Möglichkeiten von Programmeigenschaften anbelangt. Damit muß in einer solchen Darstellungsform nicht jedes beschriebene Annehmen einer Möglichkeit einer Programmeigenschaft bedeuten, daß in der durch die Programmpfade charakterisierten Programminstanz diese Programmeigenschaft auch tatsächlich gilt. Erst durch die Beschränkung auf diejenigen Programmeigenschaften, die für eine konkrete Analysebetrachtung relevant sind, wird es möglich werden, widerspruchsfreie Programminstanzenbeschreibungen für praktische Zwecke zu betrachten.

5.2.4 Endliche deterministische Automaten

Um potentiell unendliche Pfade und auf diesen Pfaden möglicherweise vorkommende Programmeigenschaften beschreiben zu können, bedarf es einer Möglichkeit, diese durch eine endliche Beschreibung darstellen zu können. Zu diesem Zweck bieten sich endliche Automaten aus dem Gebiet der theoretischen Informatik (vgl. z.B. [HU79]) an.

Definition 5.9 (Deterministischer endlicher Automat)
Ein deterministischer endlicher Automat (DEA) M sei definiert als ein Tupel

$$M = (Q, E, \delta, q_0, F, \text{result}),$$

mit

Q	*einer endlichen Menge von Zuständen,*
E	*einer Menge von Eingabesymbolen,*
$\delta : Q \times E \to Q$	*einer (partiellen) Zustandsübergangsfunktion,*
$q_0 \in Q$	*einem Startzustand,*
$F \subseteq Q$	*einer Menge von Endzuständen (die üblicherweise mit $q_{f_1}, q_{f_2}, \ldots$ bezeichnet werden),*
result : $F \to V$	*einer Abbildung von der Menge der Endzustände in die Menge V der Variablen.*

DEAs werden im folgenden oft nur als Automaten bezeichnet werden.

Automaten nach dieser Definition besitzen keine Ausgabefunktion. Zusätzlich zu den üblicherweise mit einem Automaten assoziierten Bestandteilen besitzen Automaten in dieser Arbeit die Funktion result, auf die in späteren Kapiteln genauer eingegangen wird.

Üblicherweise erweitert man die Zustandsübergangsfunktion δ derart, daß man sie auch auf Wörter, d.h. Sequenzen $e_1 \ldots e_n \in E*$ von Symbolen aus dem Eingabealphabet mit $n \in \mathbb{N}$ und $\forall 1 \leq i \leq n : e_i \in E$, anwenden kann. Dies geschieht durch die folgende Definition.

Definition 5.10 (Zustandsübergangsfunktion für Eingabewörter)
Für eine Menge Q von Zuständen, einem Eingabealphabet E, dem speziellen Symbol ϵ für eine leere Eingabe und einer Zustandsübergangsfunktion δ gemäß Definition 5.9 werde eine Funktion $\tilde{\delta}$ wie folgt definiert. Dabei bezeichne die Menge E^* die Menge aller beliebig langen Sequenzen von Symbolen aus E.

$$\tilde{\delta} \quad : \quad \begin{cases} Q \times E^* & \to & Q \\ (q, \epsilon) & \mapsto & q, \\ (q, e_1 e_2 \ldots e_n) & \mapsto & \delta(\tilde{\delta}(q, e_1 \ldots e_{n-1}), e_n) \end{cases}$$

Das Eingabewort $e_1 e_2 \ldots e_n$ mit $n = 0$ werde als das leere Wort ϵ interpretiert. Wie in der Literatur üblich wird im Folgenden nicht mehr explizit zwischen δ und $\tilde{\delta}$ unterschieden.

Definition 5.11 (Von einem Automat akzeptierte Sprache)
Für einen DEA $M = (Q, E, \delta, q_0, F, \text{result})$ sei die von diesem Automat akzeptierte Sprache $\mathcal{L}(M) \subseteq E^*$ definiert als

$$\mathcal{L}(M) = \{w \in E^* \mid \delta(q_0, w) \in F\}$$

Ein Wort $w \in \mathcal{L}(M)$ nennt man auch ein vom Automaten M akzeptiertes Wort.

Diese Art von Automaten beschreibt eine Sprache $\mathcal{L}(M)$, indem der Automat durch seine Zustandsübergangsfunktion bei Eingabe eines Wortes $w \in E^*$ vom Startzustand in einen Endzustand übergeht, falls das Wort in der Sprache ist. Ein Wort ist damit dann nicht in der Sprache, wenn der Automat nach Eingabe des Wortes (noch) nicht in einem Endzustand angelangt ist, oder für ein Symbol des Eingabewortes vom aktuellen Zustand aus kein Zustandsübergang mit diesem Symbol als Eingabesymbol existiert, und damit die Anwendung von δ auf dieses Eingabewort undefiniert ist.

Definition 5.12 (Betriebener Automat)
Ein Paar (M, q) mit $M = (Q, E, \delta, q_0, F, \text{result})$ einem DEA gemäß Definition 5.9 und $q \in Q$ werde als betriebener Automat *bezeichnet. Die Menge aller möglichen betriebenen Automaten werde mit $\mathcal{M}$ bezeichnet.*

Definition 5.13 (Notationsvereinbarung)
Für eine Zustandsübergangsfunktion δ wie in Definition 5.9 beschrieben bezeichne die Abkürzung $\delta[q' \xrightarrow{e} q'']$ eine Zustandsübergangsfunktion δ' mit der folgenden Eigenschaft:

$$\delta'(q, e') = \delta[q' \xrightarrow{e} q''](q, e') := \begin{cases} q'', & \text{falls } q = q' \text{ und } e = e', \\ \delta(q, e'), & \text{sonst} \end{cases}$$

Die Schreibweise in eckigen Klammern zur Beschreibung der Modifikation einer Funktion sei im Folgenden auf beliebige Funktionen anwendbar.

5.3 Konstruktion von Automaten zur Instanzenmengenbeschreibung

In den nachfolgenden Abschnitten soll für ein gegebenes Eingabeprogramm gemäß Definition 4.8 die Menge aller möglichen Programminstanzen, die eine Anweisung $a \in A$ erreichen, durch die von einem endlichen Automaten akzeptierte Sprache beschrieben werden.

5.3.1 Bestandteile

Dazu wird im Folgenden zu einem Eingabeprogramm $P = (A, B, V, D, b_0)$ gemäß Definition 4.8 eine Zustandsmenge, eine Eingabesymbolmenge, eine Zustandsübergangsfunktion, ein Startzustand und eine Menge von Endzuständen angegeben werden, die zusammen einen endlichen Automaten bilden, der die Menge aller möglichen Programminstanzen des Eingabeprogrammes (in Rückwärtsbeschreibungsrichtung) beschreibt, die eine Anweisung $a \in A$ erreichen.

5.3.1.1 Zustandsmenge

Definition 5.14 (Kontrollflußzustände eines Programms)
Für ein Programm $P = (A, B, V, D, b_0)$ gemäß Definition 4.8 bezeichne die Menge

$$\begin{aligned} Q_P &= \{q_a^{\text{in}} \mid a \in A\} \\ &\cup \{q_a^{\text{out}} \mid a \in A\} \\ &\cup \{q_b^{\text{in}} \mid b \in B\} \\ &\cup \{q_b^{\text{out}} \mid b \in B\} \end{aligned}$$

die Menge aller Kontrollflußzustände *Q_P eines Programms P.*

Anschaulich beschreiben diese Zustände jeweils mögliche Kontrollflußzustände eines Programmes, d.h. den Zeitpunkt vor bzw. nach Ausführung einer bestimmten Anweisung, bzw. eines Programmblocks. Dabei müssen verschiedene Zustände nicht notwendigerweise verschiedene real auftretende Kontrollflußzustände darstellen. Während der Kontrollflußzustand nach dem Ausführen einer Anweisung mit dem Zustand vor der Ausführung der darauffolgenden Anweisung für alle praktischen Verwendungen identisch ist, werden diese beiden Zustände in der hier vorgestellten Darstellungsform zum Zweck einer einfacheren Beschreibbarkeit der Vorgehensweise unterschieden.

5.3.1.2 Eingabesymbole

Eingabesymbole der Automaten bilden die Bestandteile der Wörter aus der Sprache, die von diesen Automaten akzeptiert werden. Wenn diese Wörter Programminstanzen repräsentieren sollen, dann müssen die Symbole gemäß den in Abschnitt 5.2 vorgestellten Bestandteilen zur Programminstanzbeschreibung sowohl Programmeigenschaften, als auch durchlaufene Programmabschnitte ausdrücken können.

5.3.1.2.1 Eingabesymbole zur Beschreibung von Programmeigenschaften

Die Eingabesymbole an Automaten, mit denen die Gültigkeit von Programmeigenschaften beschrieben werden kann, werden in der nachfolgenden Definition auf der Basis der Definitionen 5.1 bis 5.4 eingeführt:

Definition 5.15 (Eingabesymbol zu einer Programmeigenschaft)
Die Menge

$$E_{\mathcal{H}} := \{[\rho_a(y)] \mid \rho_a \in \mathcal{H}_A, a \in A, y \in \text{poss}(\rho_a)\}$$

bezeichne die Menge aller Eingabesymbole an Automaten, die jeweils die Gültigkeit einer Programmeigenschaft ρ_a mit gewählter Möglichkeit y an einer Anweisung a ausdrücken sollen.

Mit der in Definition 5.5 vereinbarten Schreibweise ist damit z.B. auch das Eingabesymbol $[x \to y]$ Element der Menge $E_{\mathcal{H}}$.

5.3.1.2.2 Eingabesymbole zur Beschreibung von Programmpfaden

Analog beschreibt man die Menge von Eingabesymbolen, die die durchlaufenen Pfade darstellen können, auf der Basis von Definition 5.7.

Definition 5.16 (Eingabesymbol zu einer gewählten Verzweigung)
Die Menge E_B aller Eingabesymbole, die eine von einer Programminstanz gewählte Verzweigung charakterisieren, sei wie folgt definiert:

$$E_B \quad := \quad \{\text{exit}(b) \mid b \in \text{branch}(A)\} \cup \{\overline{\text{exit}}(b) \mid \overline{b} \in \text{branch}(A)\}$$

mit branch(A) *wie in Definition 5.7 definiert.*

5.3.1.2.3 Menge aller Eingabesymbole an Automaten

Aus diesen beiden Eingabesymbolmengen setzt sich die Menge aller Eingabesymbole zusammen.

Definition 5.17 (Menge aller Eingabesymbole)
Die Menge E aller Eingabesymbole sei definiert als

$$E := E_{\mathcal{H}} \cup E_B \cup \{\eta\}$$

mit $E_{\mathcal{H}}$ wie in Definition 5.15 und E_B wie in Definition 5.16 angegeben, sowie $\eta \notin (E_{\mathcal{H}} \cup E_B)$ einem speziellen Symbol das anschaulich gesehen ein leeres Eingabesymbol darstellen soll.

5.3.1.3 Zustandsübergangsfunktion

Die Zustandsübergangsfunktion δ beschreibt in der Automatensichtweise den möglichen Kontrollfluß in einem Eingabeprogramm, und wird wie folgt definiert.

5.3.1.3.1 Anweisungen

An bestimmten Anweisungen können Programmeigenschaften gelten. Dies zu beschreiben ist die Aufgabe der nachfolgenden Definition.

Definition 5.18 (Zustandsübergangsfunktion an Anweisungen)
Für eine Anweisung $a \in A$ werde die Zustandsübergangsfunktion δ wie folgt definiert:

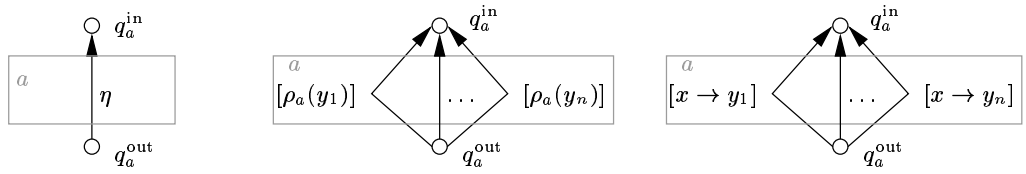
Fall 1: $\text{props}(a) = \emptyset$

$$\delta(q_a^{\text{out}}, \eta) = q_a^{\text{in}}$$

Fall 2: $\text{props}(a) = \rho_a \in \mathcal{H}$ *und* $\text{poss}(\rho_a) = \{y_1, \ldots, y_n\}$ *für ein* $n \in \mathbb{N}$*:*

$$\begin{aligned} \delta(q_a^{\text{out}}, [\rho_a(y_1)]) &= q_a^{\text{in}} \\ &\ldots \\ \delta(q_a^{\text{out}}, [\rho_a(y_n)]) &= q_a^{\text{in}} \end{aligned}$$

Abbildung 5.1 veranschaulicht diese Fälle. Das Symbol η wird später aus den Eingabewörtern entfernt werden und kann daher als ein leeres Eingabewort interpretiert werden (vergleichbar ϵ–Übergängen, die aber i.A. verwendet werden um Nichtdeterminismus darzustellen). In Teilabbildung (a) ist der erste Fall dargestellt, in dem die Menge $\text{props}(a)$ leer ist, d.h. an dieser Anweisung keine



(a) (b) (c)

Abbildung 5.1: Kontrollfluß–Zustandsübergänge an Anweisungen bei Betrachtung entgegen der Ausführungsrichtung.

relevanten Programmeigenschaften betrachtet werden. In diesem Fall wird der Zustandsübergang vom Zustand q_a^{out} zum Zustand q_a^{in} durch einen η–Zustandsübergang definiert. In Abbildung 5.1(b) ist der zweite Fall der obigen Definition dargestellt. Zwischen den beiden Zuständen q_a^{out} und q_a^{in} existieren mehrere Zustandsübergänge, die jeweils eine Möglichkeit der Programmeigenschaft gemäß Definition 5.4 beschreiben. In Abbildung 5.1(c) sind diese Zustandsübergänge für den Fall dargestellt, daß es sich bei der betrachteten Programmeigenschaft um die Ziele einer Zeigervariable gemäß Definition 5.5 handelt.

5.3.1.3.2 Programmblöcke

Programmblöcke beschreiben Sequenzen von Anweisungen. Durch die folgende Definition der Zustandsübergangsfunktion für Programmblöcke wird die sequentielle Ausführung dieser Anweisungen in der Automatensichtweise zum Ausdruck gebracht.

Definition 5.19 (Zustandsübergangsfunktion für Programmblöcke)
Für alle Anweisungen aus einem Programmblock, d.h. für ein $b \in B$ mit $n_b \in \mathbb{N}$ und $b = (a_1, \ldots, a_{n_b})$ mit den Anweisungen $a_i \in A$ für $1 \leq i \leq n_b$ wird die Zustandsübergangsfunktion δ wie folgt definiert:

$$\begin{aligned}
\forall i \in \{1, \ldots, n_b - 1\} : \delta(q_{a_{i+1}}^{\text{in}}, \eta) &= q_{a_i}^{\text{out}} \\
\delta(q_{a_1}^{\text{in}}, \eta) &= q_b^{\text{in}}, \textit{ falls } n_b \geq 1 \\
\delta(q_b^{\text{out}}, \eta) &= q_{a_{n_b}}^{\text{out}}, \textit{ falls } n_b \geq 1 \\
\delta(q_b^{\text{out}}, \eta) &= q_b^{\text{in}}, \textit{ falls } n_b = 0
\end{aligned}$$

Diese Zustandsübergänge sind in Abbildung 5.2 eingezeichnet.

5.3.1.3.3 If–Anweisungen

An If–Anweisungen werden durch die Zustandsübergangsfunktion die möglichen Verzweigungen in den then– und else–Zweig beschrieben.

Definition 5.20 (Zustandsübergangsfunktion an If–Anweisungen)
Für If–Anweisungen $a \equiv \texttt{if}_{b_1,b_2} \in A$ wird δ wie folgt definiert:

$$\begin{aligned}
\delta(q_{\texttt{if}_{b_1,b_2}}^{\text{out}}, \text{exit}(b_1)) &= q_{b_1}^{\text{out}} \\
\delta(q_{\texttt{if}_{b_1,b_2}}^{\text{out}}, \text{exit}(b_2)) &= q_{b_2}^{\text{out}} \\
\delta(q_{b_1}^{\text{in}}, \eta) &= q_{\texttt{if}_{b_1,b_2}}^{\text{in}} \\
\delta(q_{b_2}^{\text{in}}, \eta) &= q_{\texttt{if}_{b_1,b_2}}^{\text{in}}
\end{aligned}$$

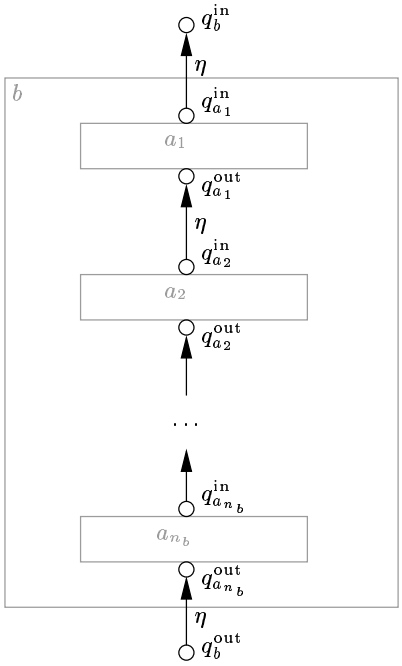


Abbildung 5.2: Kontrollfluß–Zustandsübergänge an Anweisungen innerhalb von Programmblöcken bei Betrachtung entgegen der Ausführungsrichtung.

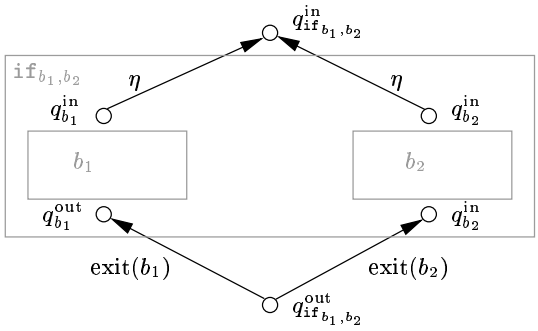


Abbildung 5.3: Kontrollfluß–Zustandsübergänge an `if`–Anweisungen bei Betrachtung entgegen der Ausführungsrichtung

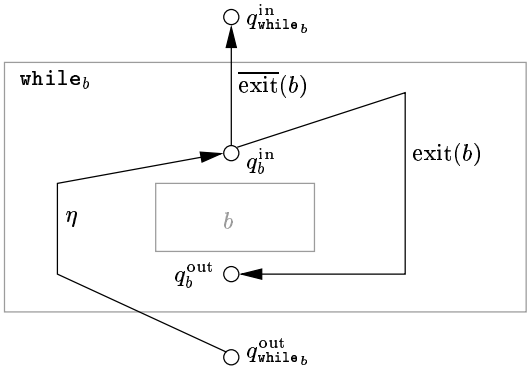


Abbildung 5.4: Kontrollfluß–Zustandsübergänge an `while`–Anweisungen bei Betrachtung entgegen der Ausführungsrichtung.

Die Zustandsübergänge an If–Anweisungen sind in Abbildung 5.3 abgebildet. Man erkennt, daß in Rückwärtsrichtung das Betreten der Programmblöcke b_1 bzw. b_2 der If–Anweisung durch die Eingabesymbole exit(b_1) bzw. exit(b_2) gekennzeichnet wird.

5.3.1.3.4 While–Anweisungen

An While–Anweisungen kann sowohl das Betreten, als auch das Umgehen des Schleifenrumpfes durch die entsprechende Definition der Zustandsübergangsfunktion ausgedrückt werden.

Definition 5.21 (Zustandsübergangsfunktion an While–Anweisungen)
Für While–Anweisungen $a \equiv \texttt{while}_b \in A$ wird δ wie folgt definiert:

$$\begin{aligned} \delta(q^{\text{out}}_{\texttt{while}_b}, \eta) &= q^{\text{in}}_b \\ \delta(q^{\text{in}}_b, \overline{\text{exit}}(b)) &= q^{\text{in}}_{\texttt{while}_b} \\ \delta(q^{\text{in}}_b, \text{exit}(b)) &= q^{\text{out}}_b \end{aligned}$$

In Abbildung 5.4 ist auch für diese Zustandsübergänge eine Veranschaulichung gezeigt. Die Verzweigungsmöglichkeit besteht hier im Zustand q^{in}_b, der den Kontrollflußzustand vor Betreten des Schleifenrumpfes b in Vorwärtsrichtung beschreibt. Durch Eingabe des Symbols $\overline{\text{exit}}(b)$ wird der Programmblock b (in Rückwärtsrichtung) nicht betreten und in den Zustand $q^{\text{in}}_{\texttt{while}_b}$ übergegangen. Das Symbol exit(b) beschreibt dagegen ein Betreten des Schleifenrumpfes in Rückwärtsrichtung. Nach der Rückkehr zum Zustand q^{in}_b bieten sich erneut diese beiden Möglichkeiten. Daher beschreibt diese Definition der Zustandsübergangsfunktion eine beliebig häufige Wiederholung des Schleifenrumpfes in Rückwärtsbetrachtungsrichtung.

5.3.1.4 Startzustand

Als Startzustand des Automaten, der die Menge aller Programminstanzen beschreibt, die eine Anweisung $a \in A$ erreichen, wird der Zustand $q^{\text{in}}_a \in Q_P$ gewählt.

5.3.1.5 Endzustandsmenge

Die Menge F der Endzustände der Automaten wird zu $F := \{q^{\text{in}}_{b_0}\}$ definiert. Dies ist derjenige Zustand, der mit dem Kontrollflußzustand vor Ausführung des Programmblockes b_0 assoziiert wird.

5.3.1.6 Zusätzliche Komponenten des Automaten

Die weiteren Bestandteile des Automaten, also die Funktion result, wird an dieser Stelle noch nicht benötigt, man kann diese also als undef setzen.

5.3.2 Definition

Mit diesen Bestandteilen kann man nun einen Automaten definieren, der die Menge aller Programminstanzen beschreibt, die eine Anweisung $a \in A$ erreichen.

Definition 5.22 (Menge aller Programminstanzen, die eine Anweisung $a \in A$ erreichen)

Für eine Anweisung $a \in A$ bezeichne

$$M_a = (Q_P, E, \delta, q_a^{\text{in}}, \{q_{b_0}^{\text{in}}\}, \text{undef}),$$

mit Q_P wie in Definition 5.14, E wie in Definition 5.17, δ wie in Definitionen 5.18 bis 5.21, und den restlichen Bestandteilen gemäß der Argumentation aus den Abschnitten 5.3.1.4 bis 5.3.1.6, denjenigen endlichen Automaten, dessen akzeptierte Sprache $\mathcal{L}(M_a)$ die Menge aller Programminstanzen beschreibt, die die Anweisung a erreichen.

Mit dieser Definition entspricht ein Wort aus der Sprache $\mathcal{L}(M_a)$ des Automaten M_a einer Programminstanzbeschreibung nach Definition 5.8 einer Programminstanz, die Anweisung a erreicht. In diesem Wort beschreibt jedes einzelne Symbol aus E die Wahl einer Programmverzweigung in Rückwärtsrichtung, bzw. die Gültigkeit einer der verschiedenen Möglichkeiten einer Programmeigenschaft. Dies wird in folgender Definition zum Ausdruck gebracht.

Definition 5.23 (Pfad)
Ein Wort $\pi \in \mathcal{L}(M_a)$ mit M_a wie in Definition 5.22 werde als Pfad *(in Rückwärtsbetrachtungsrichtung) bezeichnet.*

In dieser Definition wird die Gültigkeit von Programmeigenschaften im Sinne von *independent attributes* betrachtet, d.h. es wird angenommen, daß alle Programmeigenschaften unabhängig voneinander sind, und damit jede mögliche Kombination von Programmeigenschaften auf jedem möglichen Pfad vorkommen kann. Damit sind diese Programminstanzenbeschreibungen i.A. nicht widerspruchsfrei nach Definition 5.8. Eine Erweiterung dieser Beschreibung auf Widerspruchsfreiheit und damit auf eine Betrachtung von *relational attributes* wird später vorgestellt werden.

Da die Symbole η keine relevante Information beinhalten, werden diese im Folgenden ignoriert:

Definition 5.24 (Konvention)
Im Folgenden werden Eingabesymbole η ignoriert, d.h. Eingabeworte an Automaten werden nur unter Verwendung von Symbolen aus $E_{\mathcal{H}} \cup E_B$ dargestellt. Daß dies zu keinem Informationsverlust führt wird im nachfolgenden Satz 5.1 gezeigt werden.

5.3.3 Korrektheit

Für die hier gegebene Definition von M_a muß gewährleistet werden, daß die Menge der Wörter aus der Sprache des Automaten mit der Menge der möglichen Programminstanzenbeschreibungen direkt korrespondiert.

Satz 5.1 (Korrespondenz von M_a mit der Menge von Programminstanzen, die a err.)
Jedes Wort aus der Sprache von M_a ist in Rückwärtssichtweise eine mögliche Programminstanzbeschreibung einer Programminstanz, die nach Definition 5.8 die Anweisung a erreicht, und für jede mögliche Programminstanz, die ebenfalls nach der gleichen Definition die Anweisung a erreicht, existiert ein Wort aus der Sprache von M_a, das für diese eine Programminstanzbeschreibung ist.

Beweis: Die Korrespondenz ergibt sich offensichtlich aus der Definition von M_a. Ein Eingabesymbol $[\rho_{a'}(y)]$ kann per Definition die Gültigkeit jeder Möglichkeit für eine Programmeigenschaft an einer Anweisung $a' \in A$ ausdrücken. Für eine If–Anweisung gibt es zwei mögliche Ausführungspfade, die im Eingabewort durch entsprechende Symbole dargestellt werden können. Die mögliche Ausführung einer While–Anweisung wird ebenfalls durch Symbole für das Durchlaufen bzw. das Nicht–Durchlaufen des Schleifenrumpfes dargestellt.

Das Zusammensetzen dieser einzelnen Symbole zu einem Wort, das die Reihenfolge der Gültigkeit von Eigenschaften, bzw. von gewählten Verzweigungen widerspiegelt, erfolgt durch die Definition der Zustandsübergänge für Anweisungen innerhalb von Programmblöcken, sowie Zustandsübergängen von und zu den Zuständen vor und nach Programmblöcken. Dabei entspricht die Reihenfolge, in denen Eingabesymbole zu einem Wort zusammengefügt werden können, offensichtlich einer korrekten Reihenfolge eines möglichen Ablaufes einer Programminstanz.

Die Wahl des Startzustandes läßt die Beschreibung in Rückwärtsrichtung bei Anweisung $a \in A$ beginnen, weshalb alle durch diesen Automaten beschriebenen Programminstanzen bei Vorwärtsbetrachtung genau diese Anweisung erreichen.

Zustandsübergänge mit dem speziellen Eingabesymbol η liefern keine zusätzliche Information in dem Sinne, daß für die Betrachtung von Programminstanzen in dieser Arbeit nur Beschreibungen verwendet werden sollen, die eindeutig aber unter Vermeidung von Beschreibungsoverhead die möglichen Abläufe von Programminstanzen widerspiegeln sollen. In diese Beschreibung müssen Programmverzweigungen und die Gültigkeit von Programmeigenschaften aufgenommen werden. Die eher technisch bedingte Einführung von η–Übergängen ist hingegen für dieses Ziel nicht relevant.

Daß dadurch das Ergebnis nicht verfälscht wird, kann man aus folgender Argumentation erkennen. Gemäß den vorhergehenden Definitionen kann von jedem Zustand aus nur entweder genau ein Zustandsübergang mit dem Eingabesymbol η oder (potentiell mehrere) mit anderen Eingabesymbolen ausgehen. Damit kann zwischen je zwei Eingabesymbolen die unterschiedlich zu η sind und die bis auf η–Übergänge nacheinander in einem Eingabewort vorkommen nur eine beliebige Anzahl von η–Symbolen vorkommen. Durch das Fehlen von Verzweigungsmöglichkeiten müssen die zwei Symbole in jedem Fall in jedem Eingabewort aufeinanderfolgen. Daher wird durch das Entfernen dieser Eingabesymbole kein Nichtdeterminismus eingeführt. Mit der oben angesprochenen Zielsetzung sind diese Symbole für die Programminstanzenbeschreibung nicht notwendig und können daher bei der weiteren Betrachtung weggelassen werden. □

5.4 Menge aller Programminstanzenbeschreibungen für das Running Example

In Abbildung 5.5(a) ist der Automat M_a gemäß Definition 5.22 dargestellt, der die Menge aller Programminstanzenbeschreibungen derjenigen Programminstanzen repräsentiert, die im Running Example die Anweisung `D=**x` erreichen. Kanten ohne Beschriftung entsprechen dabei η–Übergängen. Die anderen Kanten werden zur Hervorhebung durch dickere Pfeile dargestellt. In Abbildung 5.5(b) ist eine Vereinfachung des Automaten aus Teilabbildung (a) dargestellt, in dem die η–Übergänge entfernt wurden.

Worte aus der Sprache des Automaten aus Abb. 5.5(b) sind z.B. das aus einem einzelnen Symbol bestehende Wort $\overline{\text{exit}}(b_1)$, sowie das Wort $\text{exit}(b_1)\ \text{exit}(b_3)\ \text{exit}(b_1)\ \text{exit}(b_3)\ \overline{\text{exit}}(b_1)$, da der Automat bei Eingabe dieser Wörter von seinem Startzustand in einen Endzustand übergeht. Diese beiden Worte entsprechen Pfaden, die ein Umgehen des Rumpfes der While–Schleife bzw. ein Betreten und zweimaliges Wiederholen des Rumpfes mit jeweiligem Betreten des else–Zweiges und anschließendem Verlassen der Schleife beschreiben. Als Beispiel für ein nicht akzeptiertes Wort kann man $\text{exit}(b_1) \notin \mathcal{L}(M_a)$ betrachten. Nach Eingabe des Symbols ist der Automat nicht in einem Endzustand angelangt.

5.5 Klassen von Instanzenmengen

Die nachfolgenden Abschnitte beschreiben formal, wie einzelne Worte verwendet werden können, um Klassen von Programminstanzen zu beschreiben. Diese Vorgehensweise kann man dann auf die Verwendung von Sprachen anstatt von einzelnen Worten erweitern. Zentrales Element der Definition von gültigen Beschreibungen von solchen Klassen werden dabei die Ordnungen $\leq_A$ und $\leq_B$ aus dem vorhergehenden Kapitel, bzw. darauf aufbauende Definitionen sein.

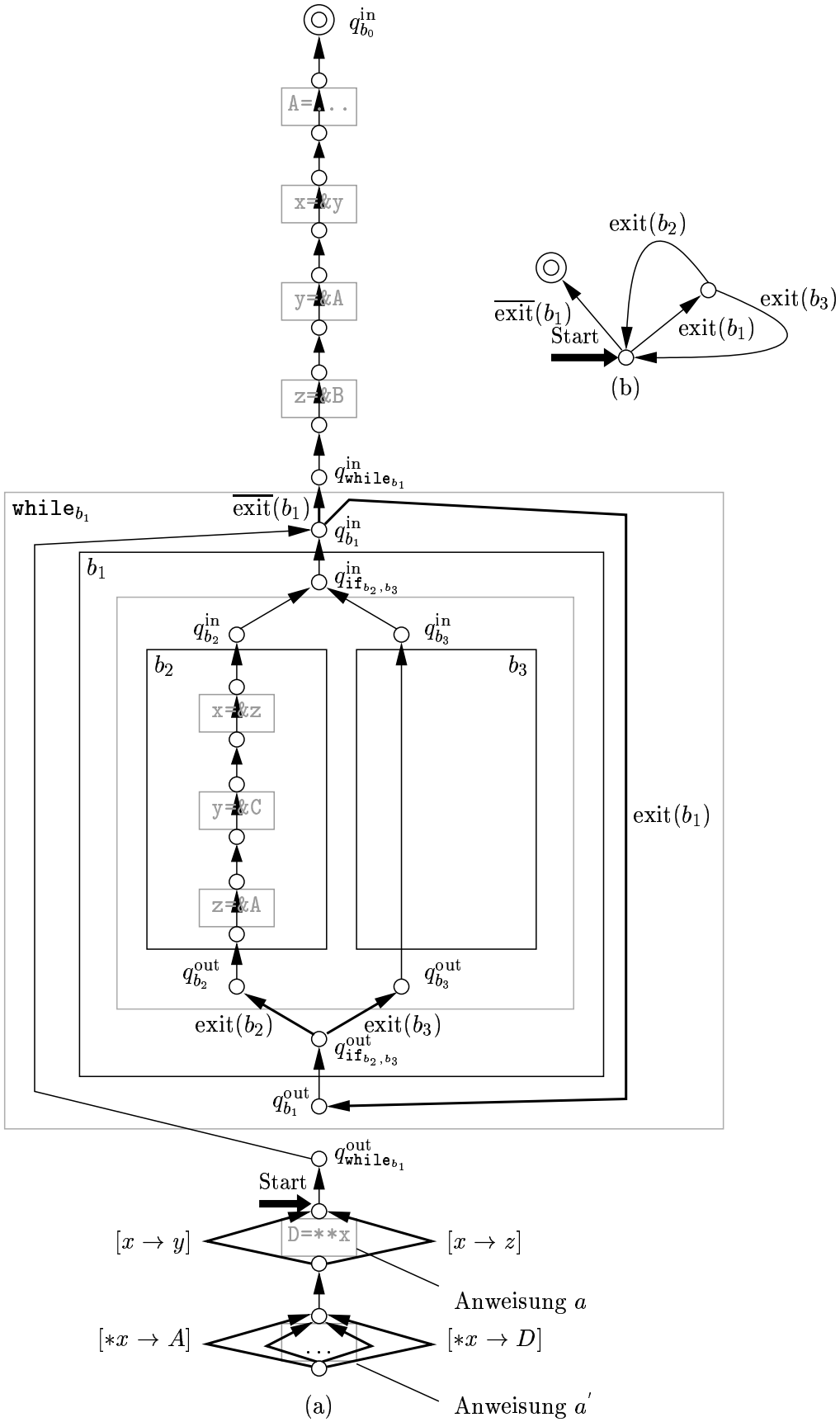


Abbildung 5.5: Automat M_a, der alle Programminstanzen beschreibt, die die Anweisung $a \equiv$ `D = * * x` erreichen (a) und der durch den Übergang zur η-Hülle entstehende Automat (b).

5.5.1 Transformation der Ordnung auf Anweisungen auf Eingabesymbole

Ziel der nachfolgenden Betrachtung ist es, eine Ordnung auf Eingabesymbolen zu definieren, anhand derer Vergleiche von Eingabesymbolen Aussagen darüber ermöglichen, welches von verschiedenen möglichen Eingabesymbolen als nächstes in einer Programminstanzbeschreibung auftreten kann. Dabei werden die Eingabesymbole in Klassen eingeordnet, wobei alle Elemente einer Klasse an der gleichen Anweisung des Eingabeprogrammes "eine Bedeutung haben".

Für die in Definition 5.17 eingeführte Menge aller Eingabesymbole wird dazu zunächst durch folgende Definition ein Bezug zwischen Eingabesymbolen und den Anweisungen, an denen diese Symbole eine Bedeutung haben, hergestellt.

Definition 5.25 (Origin–Abbildung)
Die Abbildung origin : $E \to A$ *werde wie folgt definiert. Dabei sei* $b \in B$, $a \in A$ *und* $\rho_a \in \mathcal{H}_A$.

$$\begin{aligned}
\text{origin}([\rho_a]) &:= a \\
\text{origin}(\text{exit}(b)) &:= \begin{cases} \texttt{while}_b, & \textit{falls } \exists \texttt{while}_b \in A \\ \texttt{if}_{b,\ldots}, & \textit{falls } \exists \texttt{if}_{b,\ldots} \in A \\ \texttt{if}_{\ldots,b}, & \textit{falls } \exists \texttt{if}_{\ldots,b} \in A \end{cases} \\
\text{origin}(\overline{\text{exit}}(b)) &:= \begin{cases} \texttt{while}_b, & \textit{falls } \exists \texttt{while}_b \in A \end{cases}
\end{aligned}$$

Die Abbildung origin ergibt zu einem Eingabesymbol diejenige Anweisung, an der das Eingabesymbol eine mögliche Programmeigenschaft oder die Wahl einer Verzweigung beschreibt.

Definition 5.26 (Menge aller Symbole einer Anweisung)
Für ein Eingabesymbol $e \in E$ *bezeichne*

$$\langle e \rangle := \{e' \in E \mid \text{origin}(e') = \text{origin}(e)\}$$

die Menge aller Eingabesymbole, die von der origin*–Abbildung auf die gleiche Anweisung wie das Symbol* e *abgebildet werden.*

Die Menge $\langle e \rangle$ beschreibt also anschaulich die Menge von Symbolen, die an der gleichen Anweisung wie das Symbol e eine Bedeutung haben.

Definition 5.27 (Menge von Symbolmengen)
Die Menge

$$\widehat{E} := \{\langle e \rangle \mid e \in E\}$$

bezeichne die Menge aller Mengen von Eingabesymbolen, die jeweils von der origin*–Funktion auf die gleiche Anweisung abgebildet werden. Für ein* $\langle e \rangle \in \widehat{E}$ *bezeichne* $e \in E$ *einen beliebigen Repräsentanten aus* $\langle e \rangle$.

Definition 5.28 ($\leq^A_{\widehat{E}}$–Relation)
Die Relation $\leq^A_{\widehat{E}} \subseteq \widehat{E} \times \widehat{E}$ *sei definiert durch*

$$\langle e_1 \rangle \leq^A_{\widehat{E}} \langle e_2 \rangle :\Longleftrightarrow \text{origin}(e_1) \leq_A \text{origin}(e_2)$$

für $\langle e_1 \rangle, \langle e_2 \rangle \in \widehat{E}$.

Satz 5.2 ($\leq^A_{\widehat{E}}$ ist eine Ordnung auf $\widehat{E}$)
Die Relation $\leq^A_{\widehat{E}}$ *ist eine Ordnung auf der Menge* $\widehat{E}$.

Beweis: Die Elemente von $\widehat{E}$ und A korrespondieren direkt über die origin–Funktion. Durch die Definition von $\leq^A_{\widehat{E}}$ überträgt sich daher die Ordnungseigenschaft von $\leq_A$ direkt auf die Menge $\widehat{E}$. □

Definition 5.29 ($\leq^B_{\widehat{E}}$–Relation)
Die Relation $\leq^B_{\widehat{E}} \subseteq \widehat{E} \times \widehat{E}$ sei definiert durch

$$\langle e_1\rangle \leq^B_{\widehat{E}} \langle e_2\rangle :\Longleftrightarrow b_1 \leq_B b_2 \text{ mit } \mathrm{origin}(e_1) \in b_1, \mathrm{origin}(e_2) \in b_2$$

für $\langle e_1\rangle, \langle e_2\rangle \in \widehat{E}$.

Die Relation $\leq^B_{\widehat{E}}$ ist an sich keine Ordnung, wird aber in der folgenden Definition verwendet.

Definition 5.30 ($\leq_{\widehat{E}}$–Relation)
Die Relation $\leq_{\widehat{E}} \subseteq \widehat{E} \times \widehat{E}$ sei definiert durch

$$\langle e_1\rangle \leq_{\widehat{E}} \langle e_2\rangle :\Longleftrightarrow \begin{cases} \langle e_1\rangle \leq^A_{\widehat{E}} \langle e_2\rangle, & \textit{falls } \langle e_1\rangle \textit{ und } \langle e_2\rangle \textit{ bzgl. } \leq^A_{\widehat{E}} \textit{ vergleichbar sind,} \\ \langle e_1\rangle \leq^B_{\widehat{E}} \langle e_2\rangle, & \textit{sonst} \end{cases}$$

Satz 5.3 ($\leq_{\widehat{E}}$ ist eine Ordnung auf $\widehat{E}$)
Die Relation $\leq_{\widehat{E}}$ ist eine Ordnung auf $\widehat{E}$, d.h. eine reflexive, transitive und antisymmetrische Relation.

Beweis:

Reflexivität Z.z.: $\forall \langle e\rangle \in \widehat{E} : \langle e\rangle \leq_{\widehat{E}} \langle e\rangle$.

Sei $\langle e\rangle \in \widehat{E}$ beliebig. Da $\leq^A_{\widehat{E}}$ eine Ordnung auf $\widehat{E}$ ist, ist $\langle e\rangle$ mit sich selbst bzgl. $\leq^A_{\widehat{E}}$ vergleichbar und es gilt aufgrund des ersten Falles der Definition 5.30: $\langle e\rangle \leq_{\widehat{E}} \langle e\rangle$.

Transitivität Z.z.: $\forall \langle e\rangle, \langle e'\rangle, \langle e''\rangle \in \widehat{E} : \langle e\rangle \leq_{\widehat{E}} \langle e'\rangle \wedge \langle e'\rangle \leq_{\widehat{E}} \langle e''\rangle \Longrightarrow \langle e\rangle \leq_{\widehat{E}} \langle e''\rangle$.

Seien $\langle e\rangle, \langle e'\rangle, \langle e''\rangle \in \widehat{E}$ mit $\langle e\rangle \leq_{\widehat{E}} \langle e'\rangle \wedge \langle e'\rangle \leq_{\widehat{E}} \langle e''\rangle$.

Fall 1: $\langle e\rangle \leq_{\widehat{E}} \langle e'\rangle$ gilt aufgrund von Fall 1 der Definition 5.30.

Fall a: $\langle e'\rangle \leq_{\widehat{E}} \langle e''\rangle$ gilt ebenfalls aufgrund von Fall 1 der Definition 5.30.
Da $\leq^A_{\widehat{E}}$ eine Ordnung ist, folgt aus der Transitivitätseigenschaft von $\leq^A_{\widehat{E}}$: $\langle e\rangle \leq^A_{\widehat{E}} \langle e''\rangle$. Damit sind aber auch $\langle e\rangle$ und $\langle e''\rangle$ bzgl. $\leq^A_{\widehat{E}}$ vergleichbar, weshalb $\langle e\rangle \leq_{\widehat{E}} \langle e''\rangle$ ebenfalls erfüllt ist.

Fall b: $\langle e'\rangle \leq_{\widehat{E}} \langle e''\rangle$ gilt aufgrund von Fall 2 der Definition 5.30, d.h. $\exists b', b'' \in B$ mit $\mathrm{origin}(e') \in b'$ und $\mathrm{origin}(e'') \in b''$, sowie $b' \leq_B b''$, d.h. $(b' = b'') \vee \exists b'_1, \ldots, b'_n$ mit $n \in \mathbb{N}$ und $b'' \sqsubset_B b'_1 \sqsubset_B \ldots \sqsubset_B b'_n \sqsubset_B b'$

Fall i: $\langle e\rangle \leq^A_{\widehat{E}} \langle e'\rangle$ gilt aufgrund von Fall 1 aus Definition 4.12, d.h. $\exists b \in B : b = (a_1, \ldots, a_n), n \in \mathbb{N}$ und $\exists i, j \in \{1, \ldots, n\} : \mathrm{origin}(e) = a_i \wedge \mathrm{origin}(e') = a_j$. Da $\langle e'\rangle$ und $\langle e''\rangle$ bzgl. $\leq^A_{\widehat{E}}$ unvergleichbar sein müssen, können $\mathrm{origin}(e')$ und $\mathrm{origin}(e'')$ nicht den gleichen Block b ergeben, da ansonsten $\langle e\rangle$ und $\langle e''\rangle$ aufgrund von Fall 1 der Definition von 4.12 ebenfalls vergleichbar wären. Also ist $\mathrm{origin}(e'')$ in einem Unterblock von b enthalten, weshalb $\langle e\rangle \leq^A_{\widehat{E}} \langle e''\rangle$ aufgrund von Fall 2 von Definition 4.12 gilt, und damit auch $\langle e\rangle$ und $\langle e''\rangle$ bzgl. $\leq^A_{\widehat{E}}$ vergleichbar sind. Daher gilt $\langle e\rangle \leq_{\widehat{E}} \langle e''\rangle$.

Fall ii: $\langle e\rangle \leq^A_{\widehat{E}} \langle e'\rangle$ gilt aufgrund von Fall 2 aus Definition 4.12, d.h. $\exists b \in B : b = (a_1, \ldots, a_n), n \in \mathbb{N}$ mit $\exists k \in \{1, \ldots, n\} : a_k = \mathrm{origin}(e)$ und $\exists b_1 \ldots, b_m \in B$, $m \in \mathbb{N}$ $\exists l \in \{1, \ldots, n\} : (\texttt{while}_{b_1} = a_l \vee \texttt{if}_{b_1,\ldots} = a_l \vee \texttt{if}_{\ldots,b_1} = a_l) \wedge b_m \sqsubset_B \ldots \sqsubset_B b_1 \wedge \mathrm{origin}(e') \in b_m$ mit $k < l$, wobei $b_m = b'$ sein muß.

Falls in obigem Fall b) $b' \leq_B b''$ gilt, weil $b' = b''$ ist, so ist auch origin$(e'') \in b_m$ und damit gilt aufgrund von Fall 2 aus Definition 4.12 ebenfalls $\langle e \rangle \leq_{\widehat{E}}^{A} \langle e'' \rangle$ und damit $\langle e \rangle \leq_{\widehat{E}} \langle e'' \rangle$.

Andernfalls gilt $b'' \sqsubset_B b_1' \sqsubset_B \ldots \sqsubset_B b_n' \sqsubset_B b' = b_m \sqsubset_B \ldots \sqsubset_B b_1$ und origin$(e'') \in b''$, weswegen Fall 2 aus Definition 4.12 erfüllt ist und ebenfalls $\langle e \rangle \leq_{\widehat{E}}^{A} \langle e'' \rangle$ und damit $\langle e \rangle \leq_{\widehat{E}} \langle e'' \rangle$ folgt.

Fall iii: $\langle e \rangle \leq_{\widehat{E}}^{A} \langle e' \rangle$ gilt aufgrund von Fall 3 aus Definition 4.12.

Der Beweis erfolgt analog zu Fall ii), wobei sich die Voraussetzung auf Fall 4 der Definition 4.12 zurückführen läßt.

Fall iv: $\langle e \rangle \leq_{\widehat{E}}^{A} \langle e' \rangle$ gilt aufgrund von Fall 4 aus Definition 4.12.

Beweis analog zu Fall iii).

Fall 2: $\langle e \rangle \leq_{\widehat{E}} \langle e' \rangle$ gilt aufgrund von Fall 2 der Definition 5.30.

Fall a: $\langle e' \rangle \leq_{\widehat{E}} \langle e'' \rangle$ gilt aufgrund von Fall 1 der Definition 5.30.

Beweis analog zu Fall 1b).

Fall b: $\langle e' \rangle \leq_{\widehat{E}} \langle e'' \rangle$ gilt aufgrund von Fall 2 der Definition 5.30.

Mit $b = \text{origin}(e)$, $b' = \text{origin}(e')$ und $b'' = \text{origin}(e'')$ folgt aus $b \leq_B b'$ und $b' \leq_B b''$ aufgrund der Transitivität von $\leq_B$, daß $b \leq_B b''$ gelten muß, also origin$(e) \leq_{\widehat{E}}^{B}$ origin(e'') gilt. Zu zeigen bleibt, daß $\langle e \rangle$ und $\langle e'' \rangle$ ebenfalls bzgl. $\leq_{\widehat{E}}^{A}$ unvergleichbar sind.

In beiden Fällen muß origin$(e) \neq$ origin(e') und origin$(e') \neq$ origin(e'') gelten, da die Elemente $\langle e \rangle$ und $\langle e' \rangle$ bzw. $\langle e' \rangle$ und $\langle e'' \rangle$ sonst im Widerspruch zur Annahme von Fall 2b) bzgl. $\leq_{\widehat{E}}^{A}$ vergleichbar wären. Damit bilden die Blöcke b, b' und b'' eine echte "Kette" von Unterblöcken, weswegen sie bzgl. $\leq_{\widehat{E}}^{A}$ nicht vergleichbar sind.

Also gilt auch hier $\langle e \rangle \leq_{\widehat{E}} \langle e'' \rangle$.

Insgesamt gilt also die Behauptung der Transitivität.

Antisymmetrie Z.z.: $\forall \langle e \rangle, \langle e' \rangle \in \widehat{E} : \langle e \rangle \leq_{\widehat{E}} \langle e' \rangle \wedge \langle e' \rangle \leq_{\widehat{E}} \langle e \rangle \Longrightarrow \langle e \rangle = \langle e' \rangle$.

Seien $\langle e \rangle, \langle e' \rangle \in \widehat{E}$ und gelte $\langle e \rangle \leq_{\widehat{E}} \langle e' \rangle \wedge \langle e' \rangle \leq_{\widehat{E}} \langle e \rangle$. Da Vergleichbarkeit bzgl. $\leq_{\widehat{E}}^{A}$ in beiden Richtungen gelten muß, kann die Voraussetzung nur jeweils aufgrund von Fall 1 oder Fall 2 in Definition 5.30 gelten.

Fall 1: Es gilt $\langle e \rangle \leq_{\widehat{E}}^{A} \langle e' \rangle \wedge \langle e' \rangle \leq_{\widehat{E}}^{A} \langle e \rangle$.

Da $\leq_{\widehat{E}}^{A}$ eine Ordnung ist, folgt aus deren Antisymmetrieeigenschaft $\langle e \rangle = \langle e' \rangle$.

Fall 2: $\langle e \rangle$ und $\langle e' \rangle$ sind bzgl. $\leq_{\widehat{E}}^{A}$ nicht vergleichbar. Da $\leq_B$ eine Ordnung ist, folgt origin$(e) =$ origin(e'). In diesem Fall wären die Elemente $\langle e \rangle$ und $\langle e' \rangle$ durch die Ordnung $\leq_{\widehat{E}}^{A}$ im Widerspruch zur Annahme vergleichbar gewesen. Also kann Fall 2 nicht eintreten und die Behauptung folgt aus Fall 1.

□

In Abbildung 5.6 ist die durch $\leq_{\widehat{E}}$ beschriebene Ordnung auf den Eingabesymbolen des Running Example dargestellt. Die Kästchen beschreiben jeweils Elemente $\langle e \rangle \in \widehat{E}$. Darin sind die jeweiligen $e \in E$ dargestellt, die in $\langle e \rangle$ enthalten sind. Die Pfeile entsprechen einer $\leq_{\widehat{E}}$–Beziehung zwischen den Mengen von Eingabesymbolen. Das bzgl. $\leq_{\widehat{E}}$ kleinste Element befindet sich zuoberst in der Darstellung.

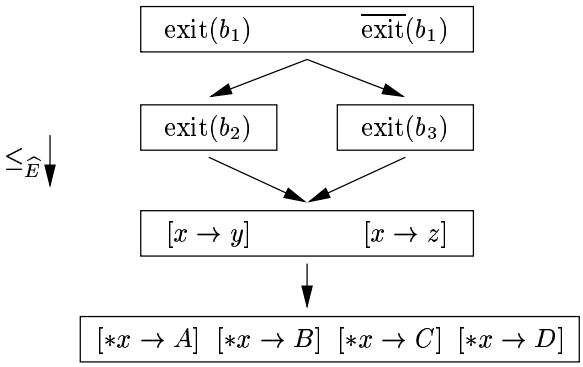


Abbildung 5.6: Ordnung $\leq_{\widehat{E}}$ auf den Eingabesymbolen des Running Example.

5.5.2 Beschreibung von Wörtern durch Teilfolgen

Wie bereits in Kapitel 3 angedeutet wurde, werden Mengen von Programminstanzen durch eine Beschreibungsform dargestellt, in der nur relevante Endabschnitte von Programmpfaden angegeben werden, die dann jeweils die Menge sämtlicher Programmpfade vom Anfang des Programmes aus mit der gleichen Endung repräsentieren. Zusätzlich soll die Möglichkeit gegeben sein, für eine Analysefragestellung unrelevante Verzweigungen und Programmeigenschaften aus der Beschreibung weglassen zu können. Im Folgenden soll eine formale Beschreibung dieses Prinzips vorgenommen werden. Dabei werden zunächst auf der Basis der oben definierten Ordnung $\leq_{\widehat{E}}$ Anforderungen vorgestellt, welche Eingabesymbole aus einer Programminstanzenbeschreibung weggelassen werden dürfen, ohne daß die eindeutige Zuordnung der Symbole des Teilwortes zu den entsprechenden Symbolen des Ausgangswortes verloren geht.

Durch das Weglassen von Eingabesymbolen aus einem Wort der Sprache des Automaten M_a kann es mehrere Worte aus dieser Sprache geben, für die das dabei entstehende Teilwort ein gemäß obigen Anforderungen zulässiges Teilwort ist. Daraus ergibt sich bereits der zentrale Verwendungszweck von solchen Teilworten. Man kann diese Teilworte anschaulich gesehen als eine Forderung an eine Menge von Programminstanzenbeschreibungen interpretieren. Jedes Wort aus der Sprache des Automaten M_a, für das das Teilwort die Anforderungen erfüllt, ist eine Programminstanzbeschreibung, die die Forderung des Teilwortes erfüllt. Damit stellen Teilworte eine Möglichkeit dar, Klassen von Programminstanzen zu bilden.

Die nachfolgenden Definitionen und Beweise haben das Ziel, die Klassenbildung von Programminstanzen formal einzuführen. Dabei werden zunächst einzelne Worte und ihre Teilfolgen betrachtet, und anschließend im nächsten Abschnitt die Betrachtung auf die Sprache des Automaten M_a und einen weiteren Automaten, dessen akzeptierte Sprache eine Menge von Teilworten und damit Forderungen an Programminstanzen darstellt, erweitert.

Als zentrale Operationen auf Worten und Teilworten werden jeweils Schnittabbildungen definiert. Diese ermöglichen es, verschiedene Teilworte eines nicht notwendigerweise bekannten Ausgangswortes nur anhand der Ordnung $\leq_{\widehat{E}}$ in einem Reißverschlussverfahren zu einem neuen Teilwort zusammenzufügen, das wiederum die Anforderungen an zulässige Teilworte erfüllt. Damit wird es ermöglicht, daß die Teilworte zum zentralen Element der Theorie und des praktischen Verfahrens werden, da sich damit alle notwendigen Berechnungen rein auf der Basis dieser Teilworte und der Ordnung $\leq_{\widehat{E}}$ durchführen lassen, und die Sprache des Automaten M_a nur im Hintergrund für die Korrektheit betrachtet wird, aber nie explizit berechnet werden muß.

Definition 5.31 (Korrekt geschachteltes Teilwort)
Sei M_a ein endlicher Automat, der gemäß Definition 5.22 die Menge aller Programmpfade beschreibt, die eine Anweisung $a \in A$ erreichen.
Für $\pi = e_1 \dots e_n \in \mathcal{L}(M_a)$ mit $n \in \mathbb{N}$ werde ein $\pi' = e_{i_1} \dots e_{i_m} \in E^*$, mit $m \in \mathbb{N}$ und $(i_k)_{k \in \{1,\dots,m\}}$

einer streng monotone Folge in $\{1,\ldots,n\}$ *und* $i_0 := 0$, *mit den Eigenschaften*

1. $\forall k \in \{1,\ldots,m\}\ \forall j \in \{i_{k-1}+1,\ldots,i_k-1\}\ :\ \text{origin}(e_j) \neq \text{origin}(e_{i_k})$
2. $\forall k \in \{1,\ldots,m\}\ \forall j \in \{i_{k-1}+1,\ldots,i_k-1\}\ :\ \langle e_{i_k}\rangle \leq_{\widehat{E}} \langle e_j\rangle$

als korrekt geschachteltes Teilwort *bezeichnet. Dieser Zusammenhang zwischen den Wörtern* π *und* π' *werde durch das Zeichen* $\pi \triangleright \pi'$ *ausgedrückt.*

Damit beschreibt ein korrekt geschachteltes Teilwort π' eine Teilfolge von Symbolen eines Wortes π aus der Sprache des Automaten M_a, für das laut der ersten Bedingung das jeweils nächste Symbol, das in π' vorkommt, auch dem nächsten Auftreten eines Symbols mit gleicher Ursprungsanweisung (durch die origin–Abbildung spezifiziert) in π entspricht. Die zweite Bedingung fordert, daß die in π' im Vergleich zu π "weggelassenen" Symbole jeweils mit dem nächsten Symbol aus π' vergleichbar und entsprechend "größer" als dieses gewesen wären.

Anschaulich betrachtet ermöglicht die Definition eines korrekt geschachtelten Teilwortes die eindeutige Zuordnung einer oder mehrerer Teilfolgen zu ihren jeweiligen Positionen im ursprünglichen Wort anhand der Ordnung $\leq_{\widehat{E}}$. Durch die erste Forderung bezieht sich das nächste Symbol aus dem Teilwort immer auf das gleiche nächste Symbol im Ursprungswort. Andernfalls könnte man beliebig lange Zwischenabschnitte im Teilwort weglassen, wodurch es verschiedene Möglichkeiten der Zuordnung des Teilwortes zum Ursprungswort geben würde.

Die zweite Forderung ermöglicht es anschaulich, rein anhand der Betrachtung des nächsten Symbols herauszufinden, welches im Ursprungswort als nächstes auftreten wird. Während dies beim Vergleich des Teilwortes mit dem Ursprungswort bereits anhand von Forderung 1 aus obiger Definition machbar ist, ist diese Eigenschaft wichtig, wenn man verschiedene Teilworte in Bezug auf ein Ursprungswort einordnen möchte, das nicht bekannt ist. Da das Prinzip der Analyse rein auf der Verwendung von korrekt geschachtelten Teilworten basieren wird, ohne daß die zugehörigen Ursprungsworte bekannt sind, ermöglichen es die Eigenschaften von korrekt geschachtelten Teilworten, mehrere solche Teilworte sozusagen im Reißverschlussverfahren zu einem einzelnen Wort zusammenzufügen, das dann auch wieder ein korrekt geschachteltes Teilwort darstellt. Dieses Reißverschlussverfahren wird in Definition 5.33 eingeführt.

Definition 5.32 (Konkatenation von Wörtern)
Für zwei Wörter $\pi = e_1 \ldots e_n, \pi' = e'_1 \ldots e'_{n'} \in E^*$ *mit* $n, n' \in \mathbb{N}_0$ *bezeichne*

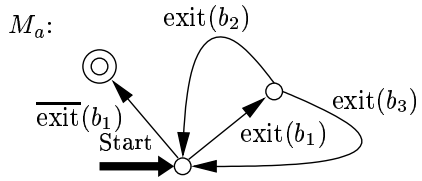
$$\pi;\pi' := e_1 \ldots e_n e'_1 \ldots e'_{n'}$$

die Konkatenation der Wörter π *und* π'.

Definition 5.33 (Schnitt von Teilfolgen von Wörtern)
Für zwei Wörter $\pi = e_1 \ldots e_n, \pi' = e'_1 \ldots e'_{n'} \in E^*$ *mit* $n, n' \in \mathbb{N}_0$ *bezeichne*

$$\pi \cap_{E^*} \pi' = e_1 \ldots e_n \cap_{E^*} e'_1 \ldots e'_{n'} := \begin{cases} e_1 \ldots e_n, & \text{falls } n' = 0, \\ e'_1 \ldots e'_{n'}, & \text{falls } n = 0, \\ e_1;(e_2 \ldots e_n \cap_{E^*} e'_1 \ldots e'_{n'}), & \text{falls } \langle e_1\rangle = \max_{\widehat{E}}(\langle e_1\rangle, \langle e'_1\rangle) \neq \langle e'_1\rangle, \\ e'_1;(e_1 \ldots e_n \cap_{E^*} e'_2 \ldots e'_{n'}), & \text{falls } \langle e'_1\rangle = \max_{\widehat{E}}(\langle e_1\rangle, \langle e'_1\rangle) \neq \langle e_1\rangle, \\ e_1;(e_2 \ldots e_n \cap_{E^*} e'_2 \ldots e'_{n'}), & \text{falls } \langle e'_1\rangle = \max_{\widehat{E}}(\langle e_1\rangle, \langle e'_1\rangle) = \langle e_1\rangle \\ & \text{und } e_1 = e'_1, \\ \text{undef}, & \text{falls } \langle e'_1\rangle = \max_{\widehat{E}}(\langle e_1\rangle, \langle e'_1\rangle) = \langle e_1\rangle \\ & \text{und } e_1 \neq e'_1 \\ \text{undef}, & \text{falls } \neg(\exists \max_{\widehat{E}}(\langle e_1\rangle, \langle e'_1\rangle)) \end{cases}$$

den Schnitt der Teilfolgen von Wörtern π *und* π', *sofern dieser definiert ist. Dabei bezeichnet* $\max_{\widehat{E}}$ *das Maximum von zwei Elementen bzgl. der Ordnung* $\leq_{\widehat{E}}$.



$$
\begin{array}{rllllll}
\mathcal{L}(M_a) \ni \pi = & \text{exit}(b_1) & \text{exit}(b_2) & \text{exit}(b_1) & \text{exit}(b_3) & \overline{\text{exit}}(b_1) & \\
\pi' = & \text{exit}(b_1) & \leq_{\widehat{E}} & \text{exit}(b_1) & \leq_{\widehat{E}} & \overline{\text{exit}}(b_1) & \Longrightarrow \pi \triangleright \pi' \\
\pi'' = & \text{exit}(b_1) & \text{exit}(b_2) & & & & \Longrightarrow \pi \triangleright \pi'' \\
\pi' \cap_{E^*} \pi'' = & \text{exit}(b_1) & \text{exit}(b_2) & \text{exit}(b_1) & & \overline{\text{exit}}(b_1) & \\
\pi''' = & \text{exit}(b_1) & & & & \overline{\text{exit}}(b_1) & \Longrightarrow \neg(\pi \triangleright \pi''') \text{ wg. Def. 5.31(1)} \\
\pi'''' = & \text{exit}(b_1) & & \leq_{\widehat{E}} \text{exit}(b_3) & & & \Longrightarrow \neg(\pi \triangleright \pi'''') \text{ wg. Def. 5.31(2)}
\end{array}
$$

Abbildung 5.7: Beispiel für Definitionen 5.31 und 5.33.

Beispiele für das Ergebnis der Anwendung von Definitionen 5.31 und 5.33 sind in Abbildung 5.7 dargestellt. Der Automat M_a, der bereits in Abbildung 5.5 vorgestellt wurde, beschreibt die Menge aller Programminstanzen, die die Anweisung `D=**x` im Running Example erreichen.

Unter diesem Automaten ist ein Wort π angegeben, das aus der Sprache des Automaten M_a ist. Dies beschreibt in Rückwärtsrichtung ein zweimaliges Durchlaufen des Schleifenrumpfes, wobei beim ersten Durchlauf der then–Zweig b_2 und beim zweiten Durchlauf der else–Zweig b_3 der If–Anweisung gewählt wird.

Ein in Bezug auf dieses Wort π korrekt geschachteltes Teilwort π' gemäß Definition 5.31 ist darunter angegeben. Das Wort π' ist offensichtlich eine Teilfolge von π. Für die ausgelassenen Symbole gilt die eingezeichnete Ordnung $\leq_{\widehat{E}}$ (wie in Abbildung 5.6 veranschaulicht), also Eigenschaft 2 aus Definition 5.31. Ebenso gilt Eigenschaft 1 aus Definition 5.31 (was hier nicht explizit eingezeichnet ist) — die enthaltenen Symbole stellen das jeweils nächste Vorkommen eines Symbols mit gleicher Ursprungsanweisung (origin) dar. Das Teilwort π' verkörpert durch die darin enthaltenen Symbole offensichtlich nur die Forderung, daß die Schleife mit Schleifenrumpf b_1 genau zweimal durchlaufen und dann verlassen werden soll. Welche Pfade im Schleifenrumpf durchlaufen werden sollen, wird durch dieses Teilwort nicht festgelegt. Konsequenterweise gibt es damit auch noch andere Worte aus der Sprache $\mathcal{L}(M_a)$, für die π' ein korrekt geschachteltes Teilwort ist. Die formale Fundierung der Vorgehensweise, nicht nur jeweils einzelne Wörter π und π', sondern jeweils Mengen von Wörtern aus Sprachen von Automaten mittels der Relation $\triangleright$ zu betrachten wird allerdings erst im nächsten Abschnitt eingeführt werden.

Darunter ist ein Wort π'' angegeben, das ebenfalls ein in Bezug auf π korrekt geschachteltes Teilwort darstellt. Man erkennt anhand von π'', daß ein korrekt geschachteltes Teilwort auch einen beliebigen Endabschnitt eines Wortes π weglassen kann. Die geforderten Eigenschaften aus Definition 5.31 beziehen sich nur auf "zwischendrin" ausgelassene Symbole. Dadurch wird es möglich, Mengen von Programminstanzenbeschreibungen durch Präfixe von Worten (in Rückwärtsinterpretation) zu beschreiben, wie dies in Abschnitt 3.3.1.3.2 bereits angesprochen wurde. Auch hier kann man, ebenfalls in einem Vorgriff auf den nachfolgenden Abschnitt, erkennen, daß die Menge von Worten aus $\mathcal{L}(M_a)$, für die dieses Teilwort π'' ein korrekt geschachteltes Teilwort ist, sämtliche Worte umfasst, die mit exit(b_1) exit(b_2) beginnen und danach beliebig fortgesetzt werden können.

In der nächsten Zeile in Abbildung 5.7 ist das Ergebnis der Anwendung von $\cap_{E^*}$ gemäß Definition 5.33 angegeben. Man erkennt im Beispiel, daß das Ergebnis eine Verschmelzung der beiden Teilfolgen π' und π'' zu einer gemeinsamen Teilfolge von π ist. Betrachtet man die Wörter π' und π'' als "Forderungen" an eine Menge von Programminstanzen, also hier daß die Schleife zweimal

durchlaufen und dann verlassen wird ($\pi^{'}$), und daß zunächst (in Ausführungsrichtung zuletzt) der Schleifenrumpf und darin der then–Zweig b_2 durchlaufen wird ($\pi^{''}$), so beschreibt $\pi' \cap_{E^*} \pi^{''}$ eine "gemeinsame Forderung". Damit ist ein ebenfalls korrekt geschachteltes Teilwort von π (dies zu zeigen ist Aufgabe des nachfolgenden Satzes) gemeint, das die Forderungen beider Teilworte in sich vereint.

Weiter sind in Abbildung 5.7 zwei Teilfolgen $\pi^{'''}$ und $\pi^{''''}$ von π angegeben, die keine korrekt geschachtelten Teilwörter von π darstellen. Beim Teilwort $\pi^{'''}$ widerspricht das Weglassen des Symbols $\text{exit}(b_1)$ in der Mitte Forderung 1 aus Definition 5.31. Anschaulich ermöglicht es diese Forderung, die Symbole aus π und aus $\pi^{'}$ bzw. $\pi^{''}$ eindeutig einander zuzuordnen.

Das Teilwort $\pi^{''''}$ verstößt gegen Forderung 2 der gleichen Definition. Für das weggelassene Symbol $\text{exit}(b_1)$ gilt nicht die Forderung, daß $\text{exit}(b_3) \leq_{\widehat{E}} \text{exit}(b_1)$ sein soll. Forderung 2 bewirkt damit, daß beim Betreten eines Programmblockes, hier also b_3 zuvor auch alle weiteren Programmblöcke betreten werden, die diesem übergeordnet sind, hier im Beispiel also b_1.

Die nachfolgenden drei Sätze haben zur Aufgabe, allgemein den Zusammenhang zwischen den im Beispiel mit $\pi^{'}$ und $\pi^{''}$ benannten Wörtern und dem Ergebnis von $\pi' \cap_{E^*} \pi^{''}$ zu zeigen. Zunächst wird, wie bereits erwähnt, bewiesen, daß das Ergebnis der Operation $\cap_{E^*}$ wiederum ein korrekt geschachteltes Teilwort von π ist. Der anschließend folgende Satz zeigt die Umkehrung dieser Schlußfolgerung, also die Aussage, daß dann, wenn $\pi^{'} \cap_{E^*} \pi^{''}$ ein korrekt geschachteltes Teilwort eines Wortes π ergibt, auch bereits die beiden Wörter $\pi^{'}$ und $\pi^{''}$ korrekt geschachtelte Teilwörter gewesen sein müssen.

Diese beiden Aussagen zusammen kann man dann derart interpretieren, daß anschaulich gesehen durch die Operation $\cap_{E^*}$ tatsächlich die Forderungen der beteiligten Teilwörter vereinigt werden.

Satz 5.4 (Ergebnis des Schnittes von Teilfolgen von Wörtern)
Sei $\pi = e_1 \dots e_n \in \mathcal{L}(M_a)$ mit $n \in \mathbb{N}$ und M_a einem Automaten gemäß Definition 5.22 für eine Anweisung $a \in A$. Seien $\pi_1, \pi_2 \in E^*$ nicht–leere Wörter mit $\pi \triangleright \pi_1$ und $\pi \triangleright \pi_2$. Dann gilt auch $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$.

Beweis: Gelten die Voraussetzungen aus dem Satz und bezeichne $\pi^{'} = (\pi_1 \cap_{E^*} \pi_2)$. Aus $\pi \triangleright \pi_1$ und $\pi \triangleright \pi_2$ folgt die Existenz von streng monotonen Folgen $(i_k)_{k \in \{1,\dots,n_1\}}$ und $(j_l)_{l \in \{1,\dots,n_2\}}$ in $\{1,\dots,n\}$ mit $n_1, n_2 \in \mathbb{N}$ und $i_0 = j_0 = 0$, so daß $\pi_1 = e_{i_1} \dots e_{i_{n_1}}$ und $\pi_2 = e_{j_1} \dots e_{j_{n_2}}$. Weiter folgt aus $\pi \triangleright \pi_1$ und $\pi \triangleright \pi_2$:

$$
\begin{aligned}
&\forall k \in \{1,\dots,n_1\} \forall j \in \{i_{k-1}+1,\dots,i_k-1\} &:\ & \text{origin}(e_j) \neq \text{origin}(e_{i_k}) \\
&\forall k \in \{1,\dots,n_2\} \forall l \in \{j_{k-1}+1,\dots,j_k-1\} &:\ & \text{origin}(e_l) \neq \text{origin}(e_{j_k}) \\
&\forall k \in \{1,\dots,n_1\} \forall j \in \{i_{k-1}+1,\dots,i_k-1\} &:\ & \langle e_{i_k} \rangle \leq_{\widehat{E}} \langle e_j \rangle \\
&\forall k \in \{1,\dots,n_2\} \forall l \in \{j_{k-1}+1,\dots,j_k-1\} &:\ & \langle e_{j_k} \rangle \leq_{\widehat{E}} \langle e_l \rangle
\end{aligned}
$$

Die Behauptung des Satzes soll mittels folgender Teilaussagen bewiesen werden. Diese beschreiben für ein $n_3 \in \mathbb{N}$ Folgen $(r_s)_{s \in \{1,\dots,n_3\}}$, $(u_s)_{s \in \{1,\dots,n_3\}}$ und $(v_s)_{s \in \{1,\dots,n_3\}}$ jeweils aus der Menge $\{1,\dots,n\}$, wobei wiederum $r_0 := 0$ definiert sein soll. Diese Folgen werden im Beweis induktiv in einer Weise definiert, daß sie die symbolweise Abarbeitung der Eingabewörter π_1 und π_2 durch die Operation $\cap_{E^*}$ beschreiben. Dabei wird die Folge (r_s) die Teilfolge $\pi^{'}$ in Bezug auf π definieren, d.h. die Position des jeweils aktuell zu $\pi^{'}$ hinzugefügten Symbols aus π_1 oder π_2 in Bezug auf das Wort π protokollieren. Die Folgen (u_s) und (v_s) beschreiben den "Abarbeitungsstatus" der beiden Eingabefolgen (i_k) und (j_l) im Sinne der symbolweisen Verarbeitung der Eingabeworte in Definition 5.33.

Damit wird es ermöglicht, basierend auf der Konstruktionsvorschrift von $\cap_{E^*}$ in Definition 5.33, Aussagen über das Wort $\pi^{'}$ als Ergebnis dieser Operation zu treffen. Die erste der folgenden Behauptungen beweist, daß die Folge (r_s) tatsächlich das Wort $\pi^{'}$ darstellt. Die zweite Aussage besagt, daß die Symbole aus $\pi^{'}$ tatsächlich eine Teilfolge von π darstellen. Die Hilfsaussagen 3 und 4 besagen informell betrachtet, dass sämtliche Symbole zwischen $e_{r_{s-1}}$ und e_{r_s} auch zwischen dem letzten und dem aktuellen verarbeiteten Symbol $e_{i_{(u_s)-1}}$ und $e_{i_{u_s}}$ bzw. $e_{j_{(v_s)-1}}$ und $e_{j_{v_s}}$ von jeweils π_1 und π_2 zu finden sind, woraus man die "korrekt geschachteltes Teilwort"–Eigenschaft aus der Voraussetzung für π_1 und π_2 in den Aussagen 5 und 6 auf die Teilfolge $\pi^{'}$ in Bezug auf π übertragen kann.

1. $\forall s \in \{1,\ldots,n_3\} : \pi' = e_{r_1} \ldots e_{r_s} \ldots$, d.h. die ersten s Symbole von π' werden durch die Folge (r_s) vorgegeben.
2. r_s ist streng monoton steigend, d.h. $\forall \in \{1,\ldots,n_3\} : r_{s+1} > r_s$.
3. $\forall s \in \{1,\ldots,n_3\} : \{r_{s-1},\ldots,r_s\} \subseteq \{i_{(u_s)-1},\ldots,i_{u_s}\}$
4. $\forall s \in \{1,\ldots,n_3\} : \{r_{s-1},\ldots,r_s\} \subseteq \{j_{(v_s)-1},\ldots,j_{v_s}\}$
5. $\forall s \in \{1,\ldots,n_3\} \forall t \in \{r_{s-1}+1,\ldots,r_s-1\} : \text{origin}(e_t) \neq \text{origin}(r_s)$.
6. $\forall s \in \{1,\ldots,n_3\} \forall t \in \{r_{s-1}+1,\ldots,r_s-1\} : \langle e_{r_s}\rangle \leq_{\widehat{E}} \langle e_t\rangle$.

Diese 6 Aussagen werden durch Induktion über s bewiesen. Hierbei wird jeweils das in den verschiedenen Fällen aus Definition 5.33 nächste zu verarbeitende Symbol ausgewählt, und dafür die Gültigkeit der Behauptungen 1 bis 6 gezeigt.

$s = 1$: Definiere $u_1 := v_1 := 1$ (d.h. am Anfang werden die Symbole $e_{i_{u_1}} = e_{i_1}$ und $e_{j_{v_1}} = e_{v_1}$ von der Definition von $\cap_{E^*}$ verglichen).

Fall 1: $i_1 = j_1$, d.h. das erste Symbol von π_1 und π_2 entspricht dem gleichen Symbol an der gleichen Position von π. Damit gilt $e_{i_1} = e_{j_1}$ und dadurch $\max_{\widehat{E}}(\langle e_{i_1}\rangle, \langle e_{j_1}\rangle) = \langle e_{i_1}\rangle = \langle e_{j_1}\rangle$. Nach Definition 5.33 ergibt sich das erste Symbol von π' daher als e_{i_1}. Mit $r_1 := i_1$ ergibt sich damit Behauptung 1.

Aufgrund der Monotonie von (i_k) und (j_l) müssen für i_2 und j_2 die Aussagen $i_2 > i_1$ und $j_2 > j_1$ gelten. Da das nächste Symbol für π' aus e_{i_2} und e_{j_2} ausgewählt wird (Fall 5 in Definition 5.33), muß $r_2 > r_1$ gelten, also folgt Behauptung 2 (da $r_0 = 0$ und $r_1 \geq 1$ ist, gilt die Monotonie auch ab dem Index 0 beginnend).

Mit $r_1 = i_1 = j_1$ folgt $\{r_0,\ldots,r_1\} = \{i_0,\ldots,i_1\}$ und $\{r_0,\ldots,r_1\} = \{j_0,\ldots,j_1\}$ und daraus speziell auch die Behauptungen 3 und 4.

Behauptungen 5 und 6 folgen sofort aus den entsprechenden Eigenschaften von (i_k) und (j_l) für $t \in \{1,\ldots,r_1-1\}$.

Im Fall 1 werden $u_2 := v_2 := 2$ gesetzt (was sich aus Fall 5 der der Definition 5.33 dadurch ergibt, daß beim Betrachten des Schnittes der "Restwörter" in der rekursiven Definition jeweils das erste Symbol von π_1 und π_2 im Weiteren nicht mehr betrachtet wird).

Fall 2: $i_1 > j_1$, d.h. das erste Symbol von π_1 ist "später" im Wort π einzuordnen als das erste Symbol von π_2.

Aufgrund der Voraussetzung gilt $\forall k \in \{1,\ldots,i_1-1\} : \langle e_{i_1}\rangle \leq_{\widehat{E}} \langle e_k\rangle$ und damit speziell $\langle e_{i_1}\rangle \leq_{\widehat{E}} \langle e_{j_1}\rangle$. Deswegen sind $\langle e_{i_1}\rangle$ und $\langle e_{j_1}\rangle$ bzgl. $\leq_{\widehat{E}}$ vergleichbar und es gilt $\max_{\widehat{E}}(\langle e_{i_1}\rangle, \langle e_{j_1}\rangle) = \langle e_{j_1}\rangle$. Weiter gilt nach Voraussetzung $\forall k \in \{1,\ldots,i_1-1\} : \text{origin}(e_{i_1}) \neq \text{origin}(e_k)$ und damit speziell $\text{origin}(e_{i_1}) \neq \text{origin}(e_{j_1})$, woraus $e_{i_1} \neq e_{j_1}$ folgt. Damit ergibt sich aus Fall 4 der Definition 5.33 das erste Symbol von π' als e_{j_1}, was mit $r_1 := j_1$ Behauptung 1 erfüllt. Weiter gilt, daß das nächste Symbol von π' ebenfalls nach Definition 5.33 aus e_{i_1} und e_{j_2} ausgewählt wird, wobei nach Voraussetzung im aktuellen Fall $i_1 > j_1 = r_1$ und nach Voraussetzung für die Folge (j_l) die Aussage $j_2 > j_1 = r_1$ gilt, daher folgt Behauptung 2 (da $r_0 = 0$ und $r_1 \geq 1$ ist, gilt die Monotonie auch in diesem Fall ab dem Index 0 beginnend).

Mit $r_1 = j_1$ folgt $\{r_0,\ldots,r_1\} = \{j_0,\ldots,j_1\}$ und wegen $i_1 > j_1$ $\{r_0,\ldots,r_1\} \subseteq \{i_0,\ldots,i_1\}$ und daraus die Behauptungen 3 und 4.

Da aus der Voraussetzung $\forall k \in \{1,\ldots,j_1-1\} : \text{origin}(e_{j_1}) \neq \text{origin}(e_k)$ und $\forall k \in \{1,\ldots,j_1-1\} : \langle e_{j_1}\rangle \leq_{\widehat{E}} \langle e_k\rangle$ folgt, gilt dies auch für $r_1 = j_1$, weshalb Behauptungen 5 und 6 folgen.

In Fall 2 wird $u_2 := 1$ und $v_2 := 2$ gesetzt (was sich aus der Definition 5.33 dadurch ergibt, daß beim Betrachten des Schnittes der "Restwörter" nur das erste Symbol von π_2 ab nun nicht mehr betrachtet wird).

Fall 3: $i_1 < j_1$.

Analog zu Fall 2. Hierbei wird $r_1 := i_1$, sowie $u_2 := 2$ und $v_2 := 1$ gesetzt.

$s \to s+1$ Gelte die Behauptung für s, d.h. seien speziell die ersten s Symbole von π' durch $e_{r_1} \dots e_{r_s}$ ausgedrückt und seien u_{s+1} und v_{s+1} wie jeweils bei Induktionsanfang und Induktionsschritt beschrieben gegeben. Weiter gelte als Induktionsvoraussetzung $\{r_{s-1}, \dots, r_s\} \subseteq \{i_{(u_s)-1}, \dots, i_{u_s}\}$ und $\{r_{s-1}, \dots, r_s\} \subseteq \{j_{(v_s)-1}, \dots, j_{v_s}\}$

Fall 1: $i_{u_{s+1}} = j_{v_{s+1}}$. Damit gilt $e_{i_{u_{s+1}}} = e_{j_{v_{s+1}}}$ und dadurch $\max_{\widehat{E}}(\langle e_{i_{u_{s+1}}}\rangle, \langle e_{j_{v_{s+1}}}\rangle) = \langle e_{i_{u_{s+1}}}\rangle = \langle e_{j_{v_{s+1}}}\rangle$. Nach Fall 5 in Definition 5.33 ergibt sich das nächste Symbol von π' daher als $e_{i_{u_{s+1}}}$. Mit $r_{s+1} := i_{u_{s+1}}$ ergibt sich damit Behauptung 1.

Die Begründung der Monotonie (Behauptung 2) erfolgt analog zur Argumentation in Fall 1 im Induktionsanfang.

Je nachdem, ob das Folgenelement r_s aus der Folge (i_k) oder (j_l) entstammt, ergeben sich für den Beweis von Behauptung 3 zwei Fälle:

Fall a: $r_s = i_{u_{s+1}-1}$ (d.h. das Element der Folge (i_k), das dort vor dem aktuell betrachteten Element $i_{u_{s+1}}$ vorkommt, wurde im Schritt s "konsumiert" und damit in r_s übernommen).
In diesem Fall kann man direkt folgern, daß $\{r_s, \dots, r_{s+1}\} = \{i_{u_{s+1}-1}, \dots, i_{u_{s+1}}\}$ (aufgrund der Gleichheit der Intervallgrenzen).

Fall b: $r_s \neq i_{u_{s+1}-1}$. Das Folgenelement u_{s+1} beschreibt das in Schritt $s+1$ zu verarbeitende Symbol von π_1, das Folgenelement u_s das in Schritt s verarbeitete Symbol. In Definition 5.33 wird nur dann (in der Ausdrucksweise dieses Beweises) $u_{s+1} = u_s + 1$ gesetzt, wenn im Schritt s das Eingabesymbol $e_{i_{u_s}}$ "konsumiert" wurde, d.h. wenn $r_s = i_{u_s}$ und wegen $u_{s+1} = u_s + 1$ auch $r_s = i_{u_{s+1}-1}$ gilt. Da das nach Voraussetzung des aktuellen Falles nicht gilt, muß entsprechend $u_{s+1} = u_s$ als einzige andere Möglichkeit gelten (d.h. in Schritt s wurde das Symbol $e_{i_{u_s}}$ noch nicht verarbeitet). Daher folgt aus der Induktionsvoraussetzung $\{r_{s-1}, \dots, r_s\} \subseteq \{i_{(u_s)-1}, \dots, i_{u_s}\}$, daß $\{r_{s-1}, \dots, r_s\} \subseteq \{i_{(u_{s+1})-1}, \dots, i_{u_{s+1}}\}$. Aus der obigen Definition $r_{s+1} = i_{u_{s+1}}$ folgt damit, daß auch $\{r_{s-1}, \dots, r_s, \dots, r_{s+1}\} \subseteq \{i_{(u_{s+1})-1}, \dots, i_{u_{s+1}}\}$ und damit speziell auch $\{r_s, \dots, r_{s+1}\} \subseteq \{i_{(u_{s+1})-1}, \dots, i_{u_{s+1}}\}$ gelten muß, weswegen Behauptung 3 (für $s+1$) gezeigt ist.

Behauptung 4 kann analog zu Behauptung 3 gezeigt werden.

Da $r_{s+1} = i_{u_{s+1}}$ definiert wurde und die Teilmengenbeziehung aus Behauptung 3 gilt, kann man aus der Gültigkeit der Eigenschaften aus Definition 5.31 für die Folge (i_k) auch auf die Gültigkeit dieser Eigenschaften für die Folge (r_s) gemäß Behauptung 5 und 6 schließen: die Elemente zwischen r_s und r_{s+1} liegen auch zwischen Elementen i_{p-1} und i_p für $p = u_{s+1}$.

In Fall 1 werden $u_{s+2} := u_{s+1} + 1$ und $v_{s+2} := v_{s+1} + 1$ gesetzt.

Fall 2: $i_{u_{s+1}} > j_{v_{s+1}}$

Setze $r_{s+1} := j_{v_{s+1}}$. Zunächst soll Behauptung 3 gezeigt werden.

Fall a: $r_s = i_{u_{s+1}-1}$.
Dann gilt $\{r_s, \dots, r_{s+1}\} = \{i_{u_{s+1}-1}, \dots, j_{v_{s+1}}\}$ und damit auch $\{r_s, \dots, r_{s+1}\} \subseteq \{i_{u_{s+1}-1}, \dots, i_{u_{s+1}}\}$. Also gilt für diesen Fall die Behauptung 3 (für $s+1$).

Fall b: $r_s \neq i_{u_{s+1}-1}$. Analog zur Argumentation in Fall 1b) muß hier ebenfalls $u_{s+1} = u_s$ gelten. Daher folgt aus der Induktionsvoraussetzung $\{r_{s-1}, \dots, r_s\} \subseteq \{i_{(u_s)-1}, \dots, i_{u_s}\}$, daß $\{r_{s-1}, \dots, r_s\} \subseteq \{i_{(u_{s+1})-1}, \dots, i_{u_{s+1}}\}$. Mit $r_{s+1} = j_{v_{s+1}}$ und $i_{u_{s+1}} > j_{v_{s+1}}$ gilt dann weiter $\{r_{s-1}, \dots, r_s, \dots, r_{s+1}\} \subseteq \{i_{(u_{s+1})-1}, \dots, i_{u_{s+1}}\}$ und damit $\{r_s, \dots, r_{s+1}\} \subseteq \{i_{(u_{s+1})-1}, \dots, i_{u_{s+1}}\}$, also Behauptung 3.

Für Behauptung 4 ergibt sich aufgrund der Tatsache, daß $r_{s+1} = j_{v_{s+1}}$ ist die vollkommen analoge Situation zu Fall 1.

Da $r_{s+1} = j_{v_{s+1}}$ definiert wurde und die Teilmengenbeziehung aus Behauptung 4 gilt, kann man aus der Gültigkeit der Eigenschaften aus Definition 5.31 für die Folge (l_j) auch auf die Gültigkeit dieser Eigenschaften für die Folge (r_s) gemäß Behauptung 5 und 6 schließen.

Es bleibt zu zeigen, daß die obige Wahl von r_{s+1} mit der durch Definition 5.33 vorgegebenen Vorgehensweise übereinstimmt. Für r_s kann aufgrund der Monotonie von (r_s) wie bisher gelten: $r_s = i_{u_{s+1}-1} \geq j_{v_{s+1}-1}$ oder $r_s = j_{v_{s+1}-1} \geq i_{u_{s+1}-1}$. Weiter gilt $r_s < i_{u_{s+1}}$ und $r_s < j_{v_{s+1}}$ (die beiden Symbole $e_{i_{u_{s+1}}}$ und $e_{j_{v_{s+1}}}$ wurden in Schritt s noch nicht abgearbeitet). In beiden Fällen gilt damit wegen der Monotonie der Folgen und der Voraussetzung für Fall 2: $i_{u_{s+1}-1} \leq j_{v_{s+1}} \leq i_{u_{s+1}}$, wodurch die Symbole $\langle e_{i_{u_{s+1}}} \rangle$ und $\langle e_{j_{v_{s+1}}} \rangle$ nach Voraussetzung für die Folge (i_k) vergleichbar mit Ergebnis $\langle e_{i_{u_{s+1}}} \rangle \leq_{\widehat{E}} \langle e_{j_{v_{s+1}}} \rangle$ sind. Damit stimmt die Wahl von $r_{s+1} = j_{v_{s+1}}$ mit der Definition 5.33, speziell der Wahl des maximalen Symbols, überein.

In Fall 2 werden $u_{(s+1)+1} := u_{s+1}$ und $v_{(s+1)+1} := v_{s+1} + 1$ gesetzt.

Fall 3: $i_{u_{s+1}} < j_{v_{s+1}}$.

Analog zu Fall 2 mit der Definition von $u_{(s+1)+1} := u_{s+1} + 1$ und $v_{(s+1)+1} := v_{s+1}$.

Zusätzlich kann man durch diesen konstruktiven Beweis auch folgern, daß diejenigen Fälle aus Definition 5.33, die ein undefiniertes Ergebnis liefern würden, nicht eintreten können. Also ist $(\pi_1 \cap_{E^*} \pi_2)$ unter den gegebenen Voraussetzungen stets definiert. Die Teilbehauptungen 1, 2, 5 und 6 entsprechen dann der Behauptung $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$.

□

Satz 5.5 (Aus dem Schnitt von Teilfolgen von Wörtern folgende Eigenschaften)
Sei $\pi = e_1 \dots e_n \in \mathcal{L}(M_a)$ mit $n \in \mathbb{N}$ und M_a einem Automaten gemäß Definition 5.22 für eine Anweisung $a \in A$. Seien $\pi_1, \pi_2 \in E^*$ nicht–leere Wörter, die jeweils eine Teilfolge der Symbole aus π darstellen. Gelte weiter $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$. Dann gilt auch $\pi \triangleright \pi_1$ und $\pi \triangleright \pi_2$.

Beweis: Die zu zeigende Aussage

$$\pi \triangleright (\pi_1 \cap_{E^*} \pi_2) \Longrightarrow \pi \triangleright \pi_1 \wedge \pi \triangleright \pi_2$$

ist äquivalent zur Aussage

$$\neg(\pi \triangleright \pi_1 \wedge \pi \triangleright \pi_2) \Longrightarrow \neg(\pi \triangleright (\pi_1 \cap_{E^*} \pi_2))$$

bzw.

$$\neg(\pi \triangleright \pi_1) \vee \neg(\pi \triangleright \pi_2) \Longrightarrow \neg(\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)),$$

welche hier gezeigt werden soll.

Dieser Beweis verwendet explizit die Teilfolgeneigenschaft der vorkommenden Wörter in Bezug auf π. Obwohl durch geeignete Konstruktion von π_1 und π_2 und Verwendung der Definition von $\cap_{E^*}$ Symbolfolgen entstehen können, die zunächst symbolweise gleich mit tatsächlichen Teilfolgen von π sein können, ist dies kein Gegenbeispiel für diesen Satz. Eine Betrachtung der von $\cap_{E^*}$ erzeugten Symbolfolge in Bezug auf das Wort π, bei der die Position der einzelnen verarbeiteten Symbole aus π_1 und π_2 im Wort π mit einbezogen wird, würde ergeben, daß diese keine echte Teilfolge von π darstellt.

Gelte nun o.B.d.A. $\neg(\pi \triangleright \pi_1)$, d.h. gemäß Definition 5.31 mit $\pi_1 = (e_{i_k})_{k \in \{1,\dots,m\}}$ für ein passendes $m \in \mathbb{N}$:

$$\exists k \in \{1, \dots, m\}\ \exists j \in \{i_{k-1} + 1, \dots, i_k - 1\}\ :\ \text{origin}(e_j) = \text{origin}(e_{i_k}) \tag{5.1}$$

oder

$$\exists k \in \{1, \dots, m\}\ \exists j \in \{i_{k-1} + 1, \dots, i_k - 1\}\ :\ \neg(\langle e_{i_k} \rangle \leq_{\widehat{E}} \langle e_j \rangle) \tag{5.2}$$

Gelte zunächst Gleichung 5.1. Aus der Voraussetzung über die Folge $(i_k)_{k\in\{1,\ldots,m\}}$ folgt, daß das Symbol e_j nicht in π_1 enthalten ist (e_j bezeichnet ein Symbol, das in π an der Position j steht).

Fall 1: π_2 enthält das Symbol e_j ebenfalls nicht.

Dann enthält gemäß Definition 5.33 das Wort $\pi_1 \cap_{E^*} \pi_2$ das Symbol e_j ebenfalls nicht zwischen den Symbolen $e_{i_{k-1}+1}$ und e_{i_k-1}, da von $\cap_{E^*}$ nur jeweils das nächste zu verarbeitende Symbol aus π_1 und π_2 an die nächste Position des Ergebniswortes übernommen werden kann. Dieses Nicht–Enthaltensein von e_j im Ergebnis von $\cap_{E^*}$ steht im Widerspruch zu Eigenschaft 1 in Definition 5.31. Damit kann auch nicht $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$ gelten: das Symbol e_j ist nicht Element dieser Teilfolge $(\pi_1 \cap_{E^*} \pi_2)$, obwohl es "benachbarte" Elemente der Teilfolge gibt, für die die origin–Funktion das gleiche Ergebnis hat.

Fall 2: π_2 enthält das Symbol e_j.

Dann wird durch die Definition 5.33 das Symbol e_j mit dem Symbol e_{i_k} verglichen, da $\langle e_j \rangle = \langle e_{i_k} \rangle$ gilt. Dies hat im einen Fall zur Folge, daß das Ergebnis von $\cap_{E^*}$ undefiniert wird, falls $e_j \neq e_{i_k}$ ist, d.h. daß nicht $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$ gelten kann, was der Behauptung entspricht.

Ist dieses Ergebnis nicht undefiniert, so wird das Symbol e_j in das Ergebnis von $\cap_{E^*}$ aufgenommen. Da aber nach Voraussetzung $j \in \{i_{k-1}+1, \ldots, i_k-1\}$ und damit speziell $j \neq i_k$ gilt, ist damit aber das Ergebnis von $\pi_1 \cap_{E^*} \pi_2$ gemäß Definition 5.33 keine Teilfolge von π mehr, weswegen ebenfalls nicht $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$ gelten kann.

Also kann in beiden Fällen nicht $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$ gelten, was zu zeigen war.

Es bleibt zu zeigen, daß aus der Annahme von Gleichung 5.2 die gleiche Aussage gefolgert werden kann. Gelte nun also Gleichung 5.2. Die Aussage $\neg(\langle e_{i_k} \rangle \leq_{\widehat{E}} \langle e_j \rangle)$ bedeutet, daß die Elemente $\langle e_{i_k} \rangle$ und $\langle e_j \rangle$ von $\widehat{E}$ nicht vergleichbar sind oder $\langle e_j \rangle \leq_{\widehat{E}} \langle e_{i_k} \rangle$ gilt. Es müssen die gleichen Fälle wie unter der Annahme von Gleichung 5.1 unterschieden werden.

Fall 1: π_2 enthält das Symbol e_j nicht.

Analog zur obigen Argumentation wird das Fehlen von Eigenschaft 2 aus Definition 5.31 auf das Ergebniswort $\pi_1 \cap_{E^*} \pi_2$ übertragen, weswegen nicht $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$ gelten kann.

Fall 2: π_2 enthält das Symbol e_j.

Durch Definition 5.33 werden die Symbole $\langle e_j \rangle$ und $\langle e_{i_k} \rangle$ miteinander verglichen.

Fall a: Die Symbole $\langle e_j \rangle$ und $\langle e_{i_k} \rangle$ sind nicht vergleichbar. Dann ist das Ergebnis von $\pi_1 \cap_{E^*} \pi_2$ undefiniert und es gilt die Behauptung.

Fall b: Es gilt $\langle e_j \rangle \leq_{\widehat{E}} \langle e_{i_k} \rangle$. Durch Definition 5.33 wird dann als nächstes Symbol e_{i_k} in das Ergebnis von $\pi_1 \cap_{E^*} \pi_2$ aufgenommen. Da aber wiederum aufgrund der entsprechenden Intervallgrenzen $j < i_k$ gilt, ist das Ergebnis ebenfalls keine Teilfolge von π mehr, was die Behauptung ist.

Damit gilt auch unter der Annahme von Gleichung 5.2, daß nicht $\pi \triangleright (\pi_1 \cap_{E^*} \pi_2)$ gelten kann.

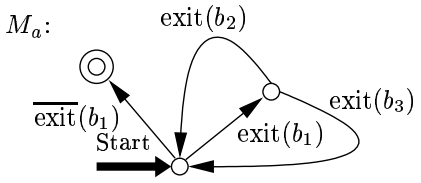
Damit ist die Aussage bewiesen. □

Satz 5.6 (Folgerung aus den Sätzen 5.4 und 5.5)
Sei $\pi = e_1 \ldots e_n \in \mathcal{L}(M_a)$ mit $n \in \mathbb{N}$ und M_a einem Automaten gemäß Definition 5.22 für eine Anweisung $a \in A$. Seien $\pi_1, \pi_2 \in E^*$ nicht–leere Wörter, die jeweils eine Teilfolge der Symbole aus π darstellen. Dann gilt

$$\pi \triangleright (\pi_1 \cap_{E^*} \pi_2) \Longleftrightarrow \pi \triangleright \pi_1 \wedge \pi \triangleright \pi_2$$

Beweis: Die Behauptung "$\Longleftarrow$" folgt aus Satz 5.4 und die Behauptung "$\Longrightarrow$" aus Satz 5.5. □

Die Aussage aus Satz 5.6 entspricht der bisher vorgestellten anschaulichen Interpretation, daß die Operation $\cap_{E^*}$ die "Forderungen" der beteiligten Eingabewörter vereinigt. Die Äquivalenzaussage bedeutet dabei, daß ein Wort π genau dann die Forderung von $\pi_1 \cap_{E^*} \pi_2$ erfüllt, wenn es zugleich die Forderungen π_1 und π_2 unabhängig voneinander erfüllt.



$$
\begin{array}{rllllll}
\pi = & \text{exit}(b_1) & & \text{exit}(b_1) & & \overline{\text{exit}}(b_1) \\
M_a \triangleright \pi = & \{\text{exit}(b_1) & \text{exit}(b_2) & \text{exit}(b_1) & \text{exit}(b_2) & \overline{\text{exit}}(b_1), \\
& \text{exit}(b_1) & \text{exit}(b_2) & \text{exit}(b_1) & \text{exit}(b_3) & \overline{\text{exit}}(b_1), \\
& \text{exit}(b_1) & \text{exit}(b_3) & \text{exit}(b_1) & \text{exit}(b_2) & \overline{\text{exit}}(b_1), \\
& \text{exit}(b_1) & \text{exit}(b_3) & \text{exit}(b_1) & \text{exit}(b_3) & \overline{\text{exit}}(b_1)\}
\end{array}
$$

Abbildung 5.8: Beispiel für Definition 5.34.

5.5.3 Erweiterung der Teilfolgenkonstruktion auf Automaten

Wie bereits im vorigen Abschnitt angedeutet wurde, soll die Betrachtung von Forderungen an Mengen von Programminstanzen in der Form von Teilworten auf die Verwendung von Automaten erweitert werden. Zunächst wird eine Menge von Worten aus der Sprache des Automaten M_a ausgewählt, die die Forderungen eines einzelnen Teilwortes erfüllen. Anschließend wird die vorgestellte Vorgehensweise dahingehend erweitert, daß auch Mengen von Teilworten durch Automaten dargestellt werden, und man diejenige Menge von Programminstanzen betrachtet, für die jeweils ein Teilwort aus dieser Menge die Anforderungen an zulässige Teilworte erfüllt, d.h. ein korrekt geschachteltes Teilwort ist. Analog zur Schnittabbildung auf Worten wird ein Automat konstruiert werden, der ebenfalls ein Reißverschlussverfahren auf mehreren Automaten durchführt, die jeweils Teilworte beschreiben. Daß diese Automaten tatsächlich die Forderungen der einzelnen Automaten in einem gemeinsamen Automaten vereinen, wird die Aussage der anschließenden Beweise sein.

Definition 5.34 (Von einem Wort induzierte Wortteilmenge d. Sprache eines Autom.)

Für ein Wort $\pi \in E^*$, eine Anweisung $a \in A$ und einen Automaten M_a gemäß Definition 5.22 bezeichne

$$M_a \triangleright \pi := \{\pi' \in \mathcal{L}(M_a) \mid \pi' \triangleright \pi\} \subseteq \mathcal{L}(M_a)$$

die vom Teilwort π induzierte Wortteilmenge aus der Sprache des Automaten M_a. *Da man die Worte aus der Sprache des Automaten M_a auch als Menge von Programminstanzen(–beschreibungen) nach Definition 5.8 interpretieren kann, wird $M_a \triangleright \pi$ auch als die* vom Teilwort π induzierte Menge von Programminstanzen *bezeichnet.*

Abbildung 5.8 zeigt ein Beispiel für die Verwendung von Definition 5.34. Wie in Abbildung 5.7 ist hier der gleiche Automat M_a und das gleiche Wort π (dort mit π' bezeichnet) dargestellt. Darunter ist das Ergebnis von $M_a \triangleright \pi$ angegeben. Dies sind alle diejenigen Worte aus der Sprache von M_a, für die π ein korrekt geschachteltes Teilwort darstellt. Wie bereits erwähnt, bezieht sich die Forderung des Wortes π nur auf die Anzahl der Schleifendurchläufe. Damit sind alle Worte aus $\mathcal{L}(M_a)$ Elemente der Menge $M_a \triangleright \pi$, die Programminstanzen beschreiben, die den Schleifenrumpf zweimal betreten und anschließend verlassen, unabhängig von den jeweils im Schleifenrumpf gewählten Verzweigungen.

Mit den bisherigen Definitionen ist es noch nicht möglich, eine Menge von Worten aus $\mathcal{L}(M_a)$ zu beschreiben, die z.B. den Schleifenrumpf b_1 beliebig häufig durchlaufen können, und dabei jeweils den then–Zweig b_2 betreten. Durch ein einziges (endliches) Wort π kann nur entweder eine genau bestimmte Anzahl von Schleifendurchläufen beschrieben werden, oder aber nach Abarbeitung des Wortes π durch die Operation $\cap_{E^*}$ kein Einfluß mehr auf die weiteren Symbole genommen werden.

Das Wort $\pi = \text{exit}(b_1)\ \text{exit}(b_2)$ beschreibt z.B. alle Programminstanzen, die zumindest einmal den Schleifenrumpf und darin den then–Zweig b_2 betreten. Für die weiteren Schleifendurchläufe lassen sich aber keinerlei Einschränkungen mehr definieren, da das Wort π nach dem ersten Schleifendurchlauf bereits abgearbeitet ist (Fälle 1 und 2 in Definition 5.33). Die nachfolgende Definition ermöglicht es nun, anstatt nur eines einzelnen Wortes π ebenfalls eine von einem Automaten akzeptierte Sprache zu verwenden, und für jedes Wort π aus dieser Sprache $M_a \triangleright \pi$ zu bilden.

Definition 5.35 (Von einem Automaten induzierte Wortteilmenge)
Sei M_a ein Automat gemäß Definition 5.22 für eine Anweisung $a \in A$ und sei $M' = (Q', E, \delta', q_0', F',$ result$')$ ein endlicher Automat, für den gilt

$$\forall q \in Q'\ \forall e_1, e_2 \in E\ :\ (\exists q', q'' \in Q'\ :\ \delta'(q, e_1) = q' \wedge \delta'(q, e_2) = q'' \Longrightarrow \langle e_1 \rangle = \langle e_2 \rangle), \tag{5.3}$$

sowie

$$\forall q \in F'\ \forall e \in E\ :\ \delta'(q, e) = q \tag{5.4}$$

Dann bezeichne

$$M_a \triangleright M' := \bigcup_{\pi \in \mathcal{L}(M')} M_a \triangleright \pi \subseteq \mathcal{L}(M_a)$$

die vom Automaten M' induzierte Wortteilmenge der Sprache des Automaten M_a *bzw. die* vom Automaten M' induzierte Menge von Programminstanzen.

Im Folgenden wird bei der Verwendung dieser Definition stets implizit von der Gültigkeit von Gleichung (5.3) ausgegangen. Die in Gleichung (5.4) beschriebene Eigenschaft äussert sich in Zustandsübergängen von Endzuständen zu sich selbst, die bei beliebigen Eingabesymbolen durchgeführt werden können. Auf eine explizite Darstellung dieser Zustandsübergänge wird im Weiteren meist verzichtet werden.

Eine Anwendung der nachfolgenden Definition wird es sein, die Menge $M_a \triangleright M'$ aus Definition 5.35 explizit durch die Sprache eines Automaten auszudrücken, anstatt diese nur implizit durch eine Vereinigung über i.A. unendlich große Wortmengen anzugeben. Allgemein betrachtet ergibt Definition 5.36 eine Methode, um zwei "Forderungen" an Worte aus $\mathcal{L}(M_a)$, die nun in der Form von endlichen Automaten M^1 und M^2 vorliegen, in einem gemeinsamen Automaten zu vereinen (analog zur Definition von $\cap_{E^*}$ auf einzelnen Wörtern).

Definition 5.36 (Schnitt der Sprachen von Wortteilmengen induzierenden Automaten)

Sei für ein $a \in A$ der Automat M_a wie in Definition 5.22 beschrieben und seien

$$M^1 = (Q^1, E, \delta^1, q_0^1, F^1, \text{result}^1)$$

und

$$M^2 = (Q^2, E, \delta^2, q_0^2, F^2, \text{result}^2)$$

Wortteilmengen induzierende Automaten für die $M_a \triangleright M^1$ und $M_a \triangleright M^2$ definiert sind. Abkürzend bezeichne

$$\delta(q, \langle e \rangle) :\Longleftrightarrow \exists e' \in \langle e \rangle\ :\ \delta(q, e')\ \textit{ist definiert}$$

das Vorhandensein eines Zustandsüberganges der Zustandsübergangsfunktion δ aus einem Zustand q mit einem Eingabesymbol aus der Menge $\langle e \rangle \in \widehat{E}$. Die Abbildung $M^1 \cap_{M_a} M^2 := M'$ bezeichne einen Automaten M', der wie folgt definiert ist:

$$M' := (Q^1 \times Q^2, \delta', (q_0^1, q_0^2), F^1 \times F^2, \text{undef})$$

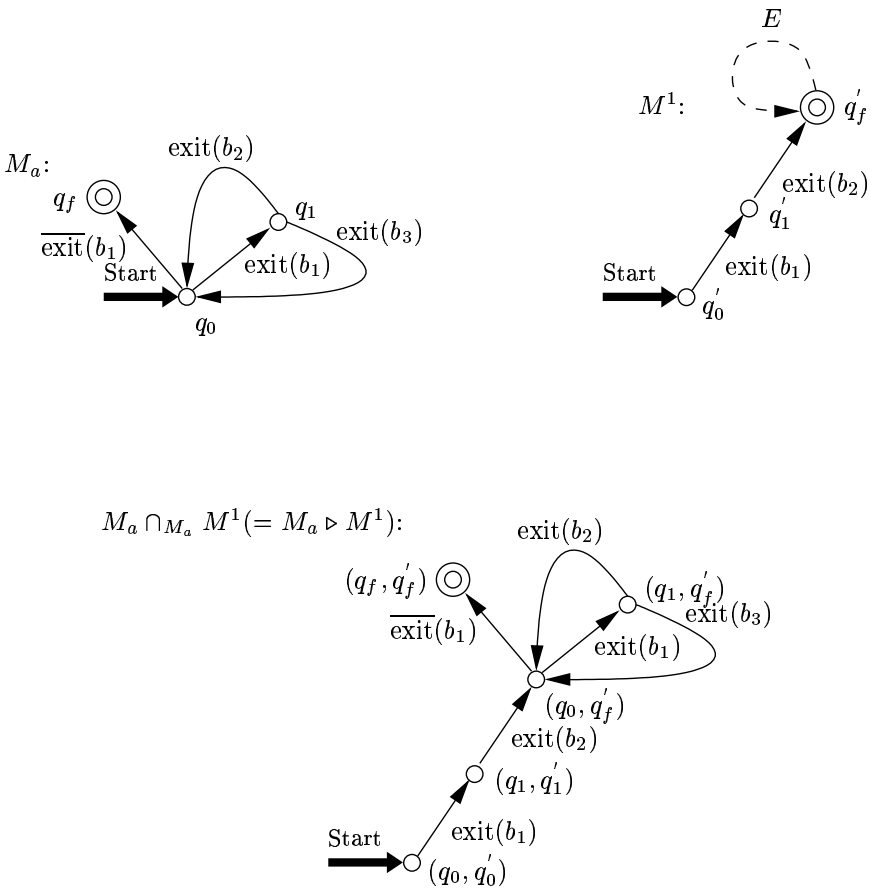


Abbildung 5.9: Beispiel für die Berechnung von $\cap_{M_a}$ gemäß Definition 5.36.

mit

$$\delta'((q_1,q_2),e) = \begin{cases} (\delta^1(q_1,e),\delta^2(q_2,e)), & \textit{falls } \delta^1(q_1,\langle e\rangle) \textit{ und } \delta^2(q_2,\langle e\rangle) \\ & \textit{und } \delta^1(q_1,e),\delta^2(q_2,e) \textit{ sind definiert,} \\ (\delta^1(q_1,e),q_2), & \textit{falls } \exists\langle e_1\rangle \neq \langle e_2\rangle \in \widehat{E} \textit{ mit } e \in \langle e_1\rangle \\ & \textit{und } \delta^1(q_1,\langle e_1\rangle),\delta^2(q_2,\langle e_2\rangle) \textit{ mit } \langle e_2\rangle \leq_{\widehat{E}} \langle e_1\rangle \\ & \textit{und } \delta^1(q_1,e) \textit{ ist definiert,} \\ (q_1,\delta^2(q_2,e)), & \textit{falls } \exists\langle e_1\rangle \neq \langle e_2\rangle \in \widehat{E} \textit{ mit } e \in \langle e_2\rangle \\ & \textit{und } \delta^1(q_1,\langle e_1\rangle),\delta^2(q_2,\langle e_2\rangle) \textit{ mit } \langle e_1\rangle \leq_{\widehat{E}} \langle e_2\rangle \\ & \textit{und } \delta^2(q_2,e) \textit{ ist definiert,} \\ \text{undef}, & \textit{sonst.} \end{cases}$$

In Abbildung 5.9 ist ein Beispiel zur Verwendung von Definition 5.36 dargestellt. Der Automat M^1 besitzt die Eigenschaft, daß $M_a \triangleright M^1$ definiert ist, da er die Bedingungen aus Definition 5.35 erfüllt. Die Zustandsübergänge, die sich aus Gleichung (5.4) ergeben, sind gestrichelt angedeutet. Da diese bei allen Automaten M', für die $M_a \triangleright M'$ gilt, identisch vorkommen, werden diese im Folgenden nicht mehr explizit dargestellt werden. Der Automat M^1 beschreibt offenbar eine Menge von Programminstanzen, bei denen vor Erreichen der Anweisung a der Schleifenrumpf b_1 und darin der then–Zweig b_2 durchlaufen wurde. In der Abbildung wird für diesen Automaten M^1 und den bereits mehrmals verwendeten Automaten M_a das Ergebnis der Berechnung von $M_a \cap_{M_a} M^1$ gezeigt. Die Zustände dieses Automaten entsprechen Paaren von Zuständen, die jeweils aus der Zustandsmenge von M_a bzw. M^1 stammen. Dargestellt sind nur solche Kombinationen von Zuständen, die vom

Startzustand aus erreichbar sind. Wie man sieht, durchläuft jede der dadurch beschriebenen Programminstanzen zuletzt den oben beschriebenen Pfadabschnitt. Im Gegensatz zur Menge aller Programminstanzen, die durch $\mathcal{L}(M_a)$ beschrieben wurden, sind in diesem Ergebnis diejenigen Programminstanzen, die direkt vor Erreichen von a den Schleifenrumpf umgangen haben, und diejenigen, die zuletzt im Schleifenrumpf den else–Zweig b_3 betreten haben, ausgeschlossen worden.

In Satz 5.8 wird bewiesen werden, daß das Ergebnis von $M_a \cap_{M_a} M^1$ einen Automaten ergibt, der die Menge $M_a \triangleright M^1$ von Wörtern beschreibt. Der dargestellte Automat stellt also eine explizite Berechnung der Menge von Wörtern $M_a \triangleright M^1$ in Form der Sprache eines Automaten dar. In Satz 5.7 wird zuvor bewiesen werden, daß die Operation $\cap_{M_a}$ für zwei Automaten, analog zur Vorgehensweise für Wörter im vorhergehenden Abschnitt, einen Automaten erzeugt, der die "Forderungen" der zwei Automaten vereinigt.

Anschaulich betrachtet verhält sich die hier im Beispiel verwendete Menge aller Wörter aus der Sprache des Automaten M_a neutral bei der Berechnung von $M_a \cap_{M_a} M^1$. Durch sie werden keine neuen Anforderungen an die von M_a und M^1 induzierten Pfadmengen eingeführt, sondern direkt diejenigen Programminstanzen aus der Sprache von M_a ausgewählt, die mit den Vorgaben aus M^1 verträglich sind.

Satz 5.7 (Ergebnis des Schnittes von Automaten)
Für eine Anweisung $a \in A$ und den Automaten M_a wie in Definition 5.22 beschrieben, sowie zwei Wortteilmengen induzierende Automaten M^1 und M^2, für die $M_a \triangleright M^1$ und $M_a \triangleright M^2$ definiert sind, bezeichne $M' = M^1 \cap_{M_a} M^2$ denjenigen Automaten, der aus Anwendung von Definition 5.36 entsteht. Dann gilt: $M_a \triangleright M' = (M_a \triangleright M^1) \cap (M_a \triangleright M^2)$.

Beweis: "$\subseteq$": Sei $\pi \in M_a \triangleright M'$. Dann ist $\pi \in \mathcal{L}(M_a)$ und es existiert ein $\pi' \in \mathcal{L}(M^1 \cap_{M_a} M^2)$ mit $\pi \triangleright \pi'$. Aus der Definition 5.36 ist ersichtlich, daß durch die von δ' ausgeführten Zustandsübergänge auf den Zustandsmengen Q^1 und Q^2 jeweils ein Wort $\pi_1 \in \mathcal{L}(M^1)$ und $\pi_2 \in \mathcal{L}(M^2)$ eindeutig beschrieben werden. Aufgrund der Ähnlichkeit der Definition 5.36 mit 5.33, speziell der durch die Ordnung $\leq_{\widehat{E}}$ festgelegten Reihenfolge der Verarbeitung der Eingabesymbole, kann man folgern, daß $\pi' = \pi_1 \cap_{E^*} \pi_2$ gelten muß. Aus Satz 5.6 kann man damit aus $\pi \triangleright \pi'$ folgern, daß $\pi \triangleright \pi_1$ und $\pi \triangleright \pi_2$ gelten muß. Damit ist $\pi \in (M_a \triangleright M^1)$ und $\pi \in (M_a \triangleright M^2)$, also $\pi \in ((M_a \triangleright M^1) \cap (M_a \triangleright M^2))$.

"$\supseteq$": Sei $\pi \in (M_a \triangleright M^1) \cap (M_a \triangleright M^2)$. Dann existieren $\pi_1 \in \mathcal{L}(M^1)$ und $\pi_2 \in \mathcal{L}(M^2)$ mit $\pi \triangleright \pi_1$ und $\pi \triangleright \pi_2$. Definiert man $\pi' = \pi_1 \cap_{E^*} \pi_2$, so gilt nach Satz 5.6 auch $\pi \triangleright \pi'$. Kann man zeigen, daß $\pi' \in \mathcal{L}(M')$ gilt, so kann man daraus die Behauptung folgern. Aufgrund der zur Definition 5.33 von $\cap_{E^*}$ ähnlichen Definition von $\cap_{M_a}$ ist ersichtlich, daß die Auswahl der jeweils zu vergleichenden Symbole bzw. Zustandsübergängen von $\cap_{M_a}$ und $\cap_{E^*}$ identisch vorgenommen wird. Aufgrund von Gleichung (5.3) in Definition 5.35, sowie der Art der Definition der Zustandsübergangsfunktion δ für den Automaten M_a in Definition 5.22, gehen von jedem Zustand der am Schnitt $\cap_{M_a}$ beteiligten Automaten nur jeweils Zustandsübergänge aus, deren Eingabesymbole alle gleiche origin–Anweisungen besitzen. Damit läßt sich die Vorgehensweise, einzelne Symbole von zwei Eingabeworten mittels der Abbildung $\cap_{E^*}$ zu bearbeiten, direkt auf die verschiedenen von einem Zustand ausgehenden Zustandsübergänge bei der Operation $\cap_{M_a}$ übertragen. Dies wird in Definition 5.36 durch die Berücksichtigung verschiedener Eingabesymbole e bereits geleistet. Durch die Voraussetzung in Gleichung (5.4) in Definition 5.35 ergibt sich auch bei Erreichen eines Endzustandes eines der beiden Automaten eine zu Fall 1 und 2 in Definition 5.33 identische weitere Vorgehensweise.

Damit führt ein Wort π', für das $\pi' = \pi_1 \cap_{E^*} \pi_2$ gilt, auch den Automaten M' in einen Endzustand aus $F^1 \times F^2$ über. Daher ist $\pi' \in \mathcal{L}(M')$, also $\pi \in (M_a \triangleright M')$ und es gilt die Behauptung "$\supseteq$", womit der Satz bewiesen ist. □

Als Ergebnis von Satz 5.7 ergibt sich ein durch Definition 5.36 ausgedrücktes Verfahren, wie man aus zwei Automaten M^1 und M^2, die jeweils eine Teilmenge von Wörtern eines Automaten M_a gemäß Definition 5.22 induzieren, einen Automaten $M' = M^1 \cap_{M_a} M^2$ ohne Verwendung des Automaten M_a selbst, sondern nur anhand der Ordnung $\leq_{\widehat{E}}$ auf Symbolen berechnen kann. Dieser Automat M' vereint die Eigenschaften von M^1 und M^2 derart, daß die von M' induzierte Teilmenge von Wörtern des Automaten M_a genau diejenigen (Programminstanzen beschreibenden) Wörter

enthält, die sowohl die von M^1 als auch die von M^2 vorgegebenen Anforderungen an durchlaufene Pfade und geltende Eigenschaften erfüllen.

Diese Vorgehensweise wird die Grundlage der Behandlung von von Queries als Bedingungen mitgeführten betriebenen Automaten bilden. Dabei wird der Automat M_a ebenfalls nicht benötigt, und daher auch nicht explizit berechnet. Dies erspart die Konstruktion des eher unhandlichen Automaten M_a bei der Analyse und damit Berechnungsaufwand.

Satz 5.8 (Berechnung von $M_a \triangleright M$)
Seien M_a und M^2 wie in Satz 5.7 definiert. Dann gilt:

$$\mathcal{L}(M_a \cap_{M_a} M^2) = M_a \triangleright M^2$$

Beweis: Zunächst läßt sich beobachten, daß für ein $\pi \in \mathcal{L}(M_a)$ stets $\pi \triangleright \pi$ gilt, da π offensichtlich eine Teilfolge von sich selbst ist und aufgrund der Tatsache, daß keine Symbole ausgelassen wurden, sind die Eigenschaften 1 und 2 aus Definition 5.31 trivialerweise erfüllt. Damit ergibt sich aus Definition 5.35 die Aussage, daß $M_a \triangleright M_a = \mathcal{L}(M_a)$ sein muß, da für jedes Wort π aus $\mathcal{L}(M_a)$ das identische Wort auf der rechten Seite von $\triangleright$ zur Verfügung steht, für das dann $\pi \triangleright \pi$ gilt. Weiter kann man erkennen, daß $M_a \cap_{M_a} M^2 \subseteq \mathcal{L}(M_a)$ gilt: da für jedes Wort aus $\mathcal{L}(M_a)$ keine Symbole ausgelassen wurden, muß jedes Eingabesymbol an den Automaten, der aus $M_a \cap_{M_a} M^2$ resultiert, einen Zustandsübergang von M_a zur Folge haben (jedes nächste Symbol aus einem Wort aus $\mathcal{L}(M^2)$ ist bzgl. $\leq_{\widehat{E}}$ per Definition kleiner oder gleich dem nächsten Symbol aus dem Wort aus $\mathcal{L}(M_a)$). Also kann man für jedes Eingabewort aus der Sprache von $M_a \cap_{M_a} M^2$ auch folgern, daß es den Automaten M_a "alleine" in einen Endzustand überführen würde. Damit ist dieses Eingabewort auch bereits in $\mathcal{L}(M_a)$.

Betrachtet man nun Satz 5.6, indem man $M^1 = M_a$ setzt, so ergibt sich:

$$\begin{aligned}
M_a \triangleright (M_a \cap_{M_a} M^2) &= \underbrace{(M_a \triangleright M_a)}_{=\mathcal{L}(M_a)} \cap (M_a \triangleright M^2) \\
\Longrightarrow M_a \triangleright (M_a \cap_{M_a} M^2) &= \mathcal{L}(M_a) \cap \underbrace{(M_a \triangleright M^2)}_{\subseteq \mathcal{L}(M_a)} \\
\Longrightarrow M_a \triangleright \underbrace{(M_a \cap_{M_a} M^2)}_{\subseteq \mathcal{L}(M_a)} &= (M_a \triangleright M^2) \\
\Longrightarrow \mathcal{L}(M_a \cap_{M_a} M^2) &= (M_a \triangleright M^2)
\end{aligned}$$

Der letzte Schritt folgt aus der Beobachtung, daß $M_a \triangleright M'$ für einen Automaten M', für den $\mathcal{L}(M') \subseteq \mathcal{L}(M_a)$ gilt, sich als die Menge derjenigen Wörter aus $\mathcal{L}(M_a)$ ergibt, die ebenfalls in $\mathcal{L}(M')$ enthalten sind. Daher ist $M_a \triangleright (M_a \cap_{M_a} M^2) = \mathcal{L}(M_a) \cap \mathcal{L}(M_a \cap_{M_a} M^2) = \mathcal{L}(M_a \cap_{M_a} M^2)$. □

5.6 Pfadbedingungen

Bisher wurden Automaten vorgestellt, die die Menge aller Programminstanzen beschreiben, die eine Anweisung $a \in A$ erreichen. Weiter wurde eine Möglichkeit eingeführt, Teilmengen dieser Programminstanzenmenge durch weitere Automaten zu beschreiben, die das Auftreten einer Teilfolge von Symbolen in Worten als Auswahlkriterium verwenden.

Im Folgenden soll ein Zusammenhang zwischen Programmeigenschaften gemäß Abschnitt 5.2.1 und solchen (Teil–)Mengen von Programminstanzen(–beschreibungen) hergestellt werden.

Definition 5.37 (Pfadbedingung)
Für eine Anweisung $a \in A$, den Automaten M_a gemäß Definition 5.22 und eine Programmeigenschaft $\rho_a \in \mathcal{H}_A$ gemäß Definition 5.1, sowie ein $y \in \text{poss}(\rho_a)$ werde ein Automat M, für den

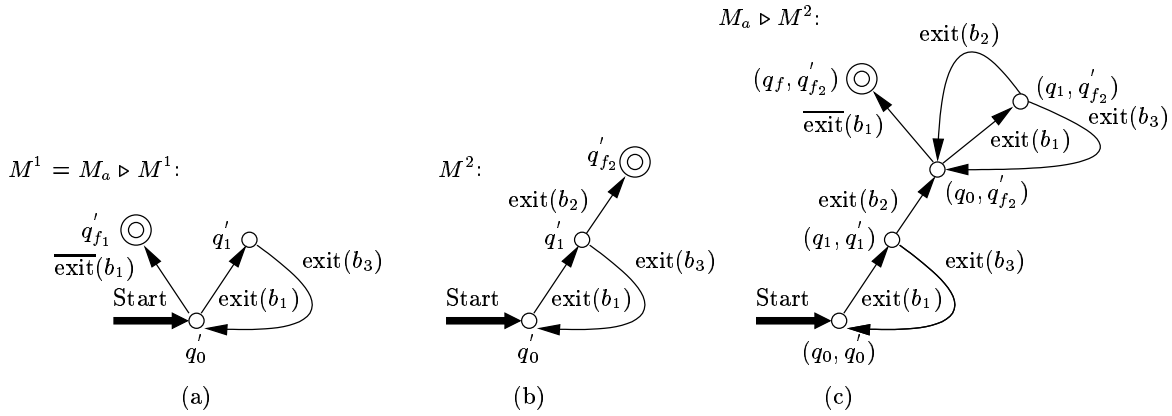


Abbildung 5.10: Beispiel für Pfadbedingungen im Running Example.

$M_a \triangleright M$ *definiert ist, als* Pfadbedingung zur Programmeigenschaft mit angenommener Möglichkeit $\rho_a(y)$ *bezeichnet und mit*

$$M \Longrightarrow \rho_a(y)$$

ausgedrückt, falls in jeder Programminstanz aus $M_a \triangleright M$ *die Programmeigenschaft* ρ_a *die Möglichkeit* y *annimmt. Die Programmeigenschaft mit angenommener Möglichkeit* $\rho_a(y)$ *werde als* Instanzenmengeneigenschaft *der Menge* $M_a \triangleright M$ *von Programminstanzen bezeichnet.*

Beispiele für solche Pfadbedingungen finden sich in Abbildung 5.10. In Teilabbildung (a) ist ein Automat M^1 dargestellt, für den bereits von seiner Struktur $M^1 = M_a \triangleright M^1 (= M_a \cap_{M_a} M^1)$ gilt. Dieser Automat beschreibt eine Menge von Programminstanzen, in denen beliebig häufig die Schleife wiederholt werden kann, in jedem Schleifenrumpf aber der else–Zweig durchlaufen werden muß. Anschließend wird die Schleife verlassen (durch das Symbol $\overline{\text{exit}}(b_1)$ ausgedrückt). Durch Betrachtung des Running Example erkennt man, daß in jeder dieser Programminstanzen die Variable x an der Anweisung $a \equiv$ `D = **x` auf die Variable y zeigen muß. Bei Ausführung des Programmes, in Vorwärtsrichtung betrachtet, wird vor der Schleife der Variablen x die Adresse von y zugewiesen und beim Durchlaufen der Schleife und des else–Zweiges diese Belegung nicht verändert. Offensichtlich gilt also

$$M^1 \Longrightarrow (x \to y)_a$$

Der Automat M^2 aus Teilabbildung (b), für den in Teilabbildung (c) das Ergebnis von $M_a \cap_{M_a} M^2$ dargestellt ist, beschreibt ebenfalls eine Menge von Programminstanzen. In jeder dieser Programminstanzen kann vor Erreichen der Anweisung a ebenfalls beliebig häufig der else–Zweig durchlaufen werden. Davor muß aber einmal im Schleifenrumpf der then–Zweig betreten worden sein. Darin wird der Variablen x die Adresse der Variablen z zugewiesen. Unabhängig vom davor durchlaufenen Pfad (der gemäß Teilabbildung (c) verlaufen kann) wird in jeder dieser Programminstanzen daher die Variable x bei Anweisung a stets die Variable z als Zeigerziel besitzen, es gilt also

$$M^2 \Longrightarrow (x \to z)_a$$

Bei näherer Betrachtung erkennt man, daß die Automaten M^1 und M^2 in Abbildung 5.10 (a) und (b) die beliebige Wiederholung des else–Zweiges als gemeinsame Struktur besitzen. Tatsächlich wird in diesem keines der beiden Ergebnisse verändert, unabhängig davon ob x nun y oder z als Zeigerziel hat. Aufgrund dieser Beobachtung ist es möglich, daß man die Automaten M^1 und M^2, bzw. allgemein die Automaten, die verschiedene Möglichkeiten der gleichen Programmeigenschaft beschreiben, zu einem einzigen Automaten zusammenfasst. Die verschiedenen Möglichkeiten der Programmeigenschaft ergeben sich dann ausschließlich durch die verschiedenen Endzustände, hier im Beispiel also q'_{f_1} und q'_{f_2}. Bei Erreichen des Endzustandes q'_{f_1} beschreibt das dabei an den Automaten

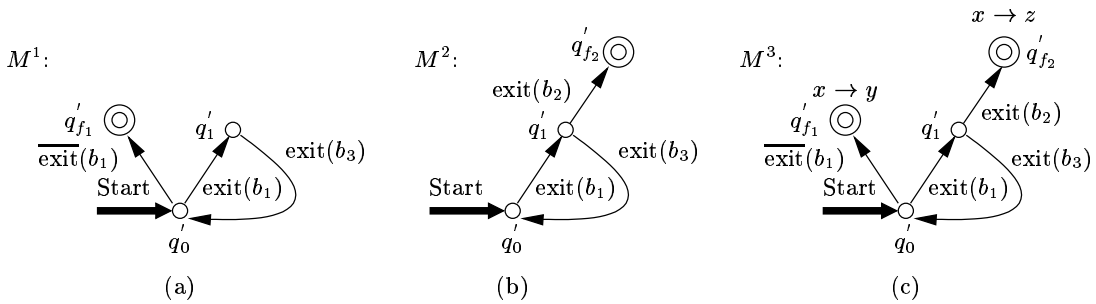


Abbildung 5.11: Beispiel für das Zusammenfassen der Pfadbedingungen im Running Example in einen einzigen Automaten.

eingegebene Wort eine Programminstanz, in der x bei Anweisung a auf y zeigt, bei q'_{f_2} entsprechend auf z.

Um dies zu beschreiben wird die in Definition 5.9 eingeführte Funktion result : $F \to V$ verwendet, die jedem Endzustand des Automaten aus F eine Variable $v \in V$ zuordnet. Im Beispiel wäre damit also $\text{result}(q'_{f_1}) = x$ und $\text{result}(q'_{f_2}) = z$. Diese Vorgehensweise wird im Folgenden formal definiert.

Definition 5.38 (Schreibweise)
Für einen Automaten $M = (Q, E, \delta, q_0, F, \text{result})$ *mit*

$$\forall q \in F \ : \ \text{result}(q) \in V$$

bezeichne für ein $y \in \text{result}(F)$ *(also aus dem Bild der Menge* F *unter der Abbildung* result*)*

$$M_y := (Q, E, \delta, q_0, \text{result}^{-1}(y), \text{result})$$

denjenigen Automaten, dessen Endzustandsmenge gegenüber M *auf nur diejenigen Endzustände* q_f *reduziert wurde, für die* $\text{result}(q_f) = y$ *gilt.*

Definition 5.39 (Zusammenfassen von Pfadbedingungen)
Für einen Automaten $M = (Q, E, \delta, q_0, F, \text{result})$ *mit*

$$\text{result}(F) = \{v_1, \ldots, v_n\} \subseteq V \text{ für ein } n \in \mathbb{N}$$

und eine Programmeigenschaft $\rho_a \in \mathcal{H}_A$ *bezeichne*

$$M \Longrightarrow \rho_a$$

(bzw.

$$M \Longrightarrow (x \to \ldots)_a$$

für Zeigerziele) abkürzend die folgenden Eigenschaften

$$\begin{aligned} M_{v_1} &\Longrightarrow \rho_a(v_1) \\ &\ldots \\ M_{v_n} &\Longrightarrow \rho_a(v_n) \end{aligned}$$

In Abbildung 5.11 sind in Teilabbildungen (a) und (b) nochmals die beiden Pfadbedingungen M^1 und M^2 für das Running Example dargestellt. In Teilabbildung (c) ist derjenige Automat M^3 dargestellt, der die Zusammenfassung der Automaten M^1 und M^2 bildet. Die Zuordnung der Endzustände zu Variablen mittels der result–Funktion wird durch eine entsprechende Annotierung der Endzustände in der Darstellung ausgedrückt.

5.7 Bedingungsautomaten

Die bisher eingeführten Operationen auf Automaten, die Wortteilmengen von Sprachen induzieren, gingen stets von zwei Automaten M^1 und M^2 aus, die beide in Bezug auf die Menge aller Programminstanzen, die jeweils die gleiche Anweisung $a \in A$ erreichen (im Sinne der Sprache des Automaten M_a), zu interpretieren waren. Die Operation $M^1 \cap_{M_a} M^2$ (aus Definition 5.36) berechnet nach Satz 5.7 als Ergebnis für diese beiden Automaten einen weiteren Automaten, der die Wortteilmenge aller derjenigen Programminstanzen induziert, in denen sowohl die Beschränkungen des Automaten M^1 als auch des Automaten M^2 gleichzeitig gelten. Durch diese Berechnung kann man also für zwei Programmeigenschaften $\rho_a, \rho_a^{'} \in \mathcal{H}_A$ mit gegebenen Pfadbedingungen der Form

$$\begin{aligned} M^1 &\Longrightarrow \rho_a(y_1) \\ M^2 &\Longrightarrow \rho_a^{'}(y_2) \end{aligned}$$

einen Automaten $M^{'} = M^1 \cap_{M_a} M^2$ berechnen, der dann durch $M_a \triangleright M^{'}$ die Menge von Programminstanzen beschreibt, in denen beide Programmeigenschaften gleichzeitig gelten.

Dieses Prinzip bildet die Grundlage für die Berechnung einer Analyse von *relational attributes*. Sind zu verschiedenen Programmeigenschaften jeweils Pfadbedingungen bekannt, so kann man berechnen, ob und in welchen Programminstanzen Kombinationen von Programmeigenschaften vorkommen können. Aus der oben angesprochenen Voraussetzung, daß alle Pfadbedingungen bezüglich der Menge aller Programminstanzen, die genau eine bestimmte Anweisung erreichen, zu interpretieren sind, ergibt sich bisher die Einschränkung, daß nur für Programmeigenschaften, die an der gleichen Anweisung gelten sollen, deren Kombinierbarkeit getestet werden kann.

Im Folgenden soll eine Verallgemeinerung der bisherigen Berechnungsweise eingeführt werden, die auch für an verschiedenen Anweisungen geltende Programmeigenschaften deren Kombinierbarkeit untersuchen kann.

Diese Verallgemeinerung wird durch die Einführung einer neuen Form von Automaten geschehen, die Bedingungsautomaten genannt werden. Während bei der Definition von $\cap_{M_a}$ von genau zwei zu Beginn gegebenen Automaten ausgegangen wird, werden bei Bedingungsautomaten weitere Automaten dann zur Verarbeitung analog zu $\cap_{M_a}$ hinzugefügt, wenn ein Eingabesymbol eingelesen wird, das eine Programmeigenschaft und damit eine zu erfüllende Bedingung beschreibt. Bedingungsautomaten betreiben also eine Menge von Automaten gleichzeitig. Dies kann man damit als eine Verallgemeinerung der Definition von $\cap_{M_a}$ sehen.

Eine Programminstanz, in der z.B. $(x \to y)_a$ gilt, muß vor Erreichen der Anweisung a einen Programmpfad durchlaufen haben, auf dem garantiert werden kann, daß $(x \to y)_a$ gilt. Dabei geht man von einer bekannten Pfadbedingung der Form $M^1 \Longrightarrow (x \to y)_a$ aus. Bei Eingabe des Symbols $[(x \to y)_a]$ an den Automaten wird dann der Automat M^1 zur Verarbeitung hinzugefügt. Eine Programminstanzbeschreibung, die bei weiterem Betreiben des Bedingungsautomaten analog zur Definition von $\cap_{M_a}$ alle gleichzeitig betriebenen Automaten, und damit speziell auch den Automaten M^1 in einen Endzustand überführt, beschreibt damit eine Programminstanz, in der die Programmeigenschaft $(x \to y)_a$ garantiert gilt.

Definition 5.40 (Bedingungsautomat)
Sei $a \in A$ eine Anweisung und $\rho_a \in \mathcal{H}_A$ eine Programmeigenschaft. Sei weiter $y \in \text{poss}(\rho_a)$ eine Möglichkeit für die Programmeigenschaft ρ_a. Sei $M^c = (Q^c, E, \delta^c, q_0^c, F^c, \text{undef})$ ein DEA, der eine Pfadbedingung für die Programmeigenschaft ρ_a mit Möglichkeit y ist, d.h. gelte $M^c \Longrightarrow \rho_a(y)$. Seien weiter $M^1 = (Q^1, E, \delta^1, q_0^1, F^1, \text{undef}), \ldots, M^n = (Q^n, E, \delta^n, q_0^n, F^n, \text{undef})$ DEAs gemäß Definition 5.9 für ein $n \in N$. Ein Tupel

$$C = (\{M^1, \ldots, M^n\}, E, \delta, (M^c, q_0^c), \emptyset)$$

mit nachfolgend definierter Zustandsübergangsfunktion $\delta : \mathcal{P}(\mathcal{M}) \times E \to \mathcal{P}(\mathcal{M})$ werde als Bedingungsautomat *bezeichnet. Zur Definition von δ sei im Folgenden stets $\{(M^{(1)}, q^{(1)}), \ldots, (M^{(m)}, q^{(m)})\}$ für ein $m \in \mathbb{N}$ eine Menge von betriebenen Automaten, so daß $\forall 1 \leq i \leq m : M^{(i)} \in \{M^1, \ldots, M^n\}$ und $\forall 1 \leq i \leq m : q^{(i)} \in Q^{(i)}$ gilt, wobei $Q^{(i)}$ die Zustandsmenge des Automaten $M^{(i)}$ bezeichnen*

soll. Für ein $e \in \langle e' \rangle$ *mit* $\langle e' \rangle := \max_{\widehat{E}\, i \in \{1,\ldots,m\}} \langle e_i \rangle$, *wobei* $\delta^{(1)}(q^{(1)}, \langle e_1 \rangle) \wedge \ldots \wedge \delta^{(m)}(q^{(m)}, \langle e_m \rangle)$ *für geeignete* $\langle e_i \rangle \in \widehat{E}$ *und die Zustandsübergangsfunktionen* $\delta^{(i)}$ *der Automaten* $M^{(i)}$ *gelten soll, sei* δ *definiert als*

$$\delta(\{(M^{(1)}, q^{(1)}), \ldots, (M^{(m)}, q^{(m)})\}, e) = \begin{cases} \bigcup\limits_{j \in \{1,\ldots,m\}} \overline{\delta}((M^{(i)}, q^{(i)}), e), & \textit{falls alle } \overline{\delta}((M^{(i)}, q^{(i)}), e) \\ & \textit{definiert sind,} \\ \text{undef}, & \textit{sonst} \end{cases}$$

Dabei sei $\overline{\delta} : \mathcal{M} \times E \to \mathcal{P}(\mathcal{M})$ *wie folgt definiert:*

$$\overline{\delta}((M^{(i)}, q^{(i)}), e) = \overline{\delta_a}((M^{(i)}, q^{(i)}), e) \cup \overline{\delta_b}((M^{(i)}, q^{(i)}), e)$$

mit $F^{(i)}$ *der Endzustandsmenge des Automaten* $M^{(i)}$ *und*

$$\overline{\delta_a}((M^{(i)}, q^{(i)}), e) = \begin{cases} \emptyset, & \textit{falls } \delta^{(i)}(q^{(i)}, e) \in F^{(i)}, \\ \text{undef}, & \textit{falls } \delta^{(i)}(q^{(i)}, \langle e \rangle), \textit{ aber } \delta^{(i)}(q^{(i)}, e) \textit{ ist nicht definiert,} \\ \{(M^{(i)}, q^{(i)})\}, & \textit{falls nicht } \delta(q^{(i)}, \langle e \rangle) \textit{ gilt,} \\ \{(M^{(i)}, q^{(i)'})\}, & \textit{falls } \delta^{(i)}(q^{(i)}, e) = q^{(i)'} \in Q^{(i)} \\ & \textit{und ein Endzustand aus } F^{(i)} \textit{ ist von } q^{(i)'} \textit{ noch erreichbar,} \\ \text{undef}, & \textit{sonst.} \end{cases}$$

und mit

$$\overline{\delta_b}((M^{(i)}, q^{(i)}), e) = \begin{cases} \{(M, q_0)\}, & \textit{falls } \overline{\delta_a}((M^{(i)}, q^{(i)}), e) = \{(M^{(i)}, q^{(i)'})\} \textit{ oder } \overline{\delta_a}((M^{(i)}, q^{(i)}), e) = \emptyset \\ & \textit{und } e = [\rho_a(y)] \textit{ für ein } a \in A \textit{ und ein } \rho_a \in \mathcal{H}_A \textit{ und } y \in \text{poss}(\rho_a) \\ & \textit{und einem Automaten } M \in \{M^1, \ldots, M^n\} \textit{ mit } M \Longrightarrow \rho_a(y) \\ & \textit{und Startzustand } q_0, \\ \emptyset, & \textit{sonst} \end{cases}$$

Der betriebene Automat (M^c, q_0^c) *werde als* Startzustand des Bedingungsautomaten C *bezeichnet. Die betriebenen Automaten, die bei Eingabe von entsprechenden Symbolen der Menge von betriebenen Automaten hinzugefügt werden, werden als* Bedingungen *bezeichnet. Die Zustandsübergangsfunktion* δ *soll in zu Definition 5.10 analoger Weise rekursiv auf die Eingabe von Wörtern aus* E^* *erweitert werden.*

Abbildung 5.12 zeigt beispielhaft das Prinzip des Betreibens von Bedingungsautomaten. Dabei wird eine Menge von Automaten gleichzeitig betrieben. Die in der Abbildung zuoberst gezeigten Automaten sollen im Beispiel diese Menge von betriebenen Automaten zu einem bestimmten Zeitpunkt in ihren jeweiligen aktuellen Zuständen darstellen. Zu Beginn des Betreibens eines Bedingungsautomaten besteht die Menge der gleichzeitig betriebenen Automaten nach der Definition zunächst nur aus seinem Startzustand (M^c, q_0^c).

Als erster Schritt der Berechnung der Zustandsübergangsfunktion $\overline{\delta}$ wird eine Klasse von Eingabesymbolen anhand der Ordnung $\leq_{\widehat{E}}$ ausgewählt, mit deren darin enthaltenen Symbolen der Bedingungsautomat als nächstes einen Zustandsübergang durchführen kann. Als nächstes wird für einen Teil der Automaten, die "etwas mit diesen Symbolen anfangen können", ein Zustandsübergang mit einem Symbol aus dieser Klasse von Eingabesymbolen durchgeführt. Im Beispiel ist das nur für das Symbol $[x \to y]$ für alle Automaten gemeinsam möglich, allgemein können aber vom aktuellen Zustand des Bedingungsautomaten, der durch die Menge der aktuellen Zustände seiner betriebenen Automaten charakterisiert wird, verschiedene Zustandsübergänge mit verschiedenen Eingabesymbolen durchgeführt werden. Sofern das Eingabesymbol das Annehmen einer Möglichkeit einer Programmeigenschaft beschreibt, wie das im Beispiel der Fall ist, wird eine Pfadbedingung für diese Programmeigenschaft zur Menge der gleichzeitig betriebenen Automaten hinzugefügt. Insgesamt wird somit ein neuer Gesamtzustand des Bedingungsautomaten erreicht, von dem aus erneut nach dem gleichen Schema Zustandsübergänge durchgeführt werden können.

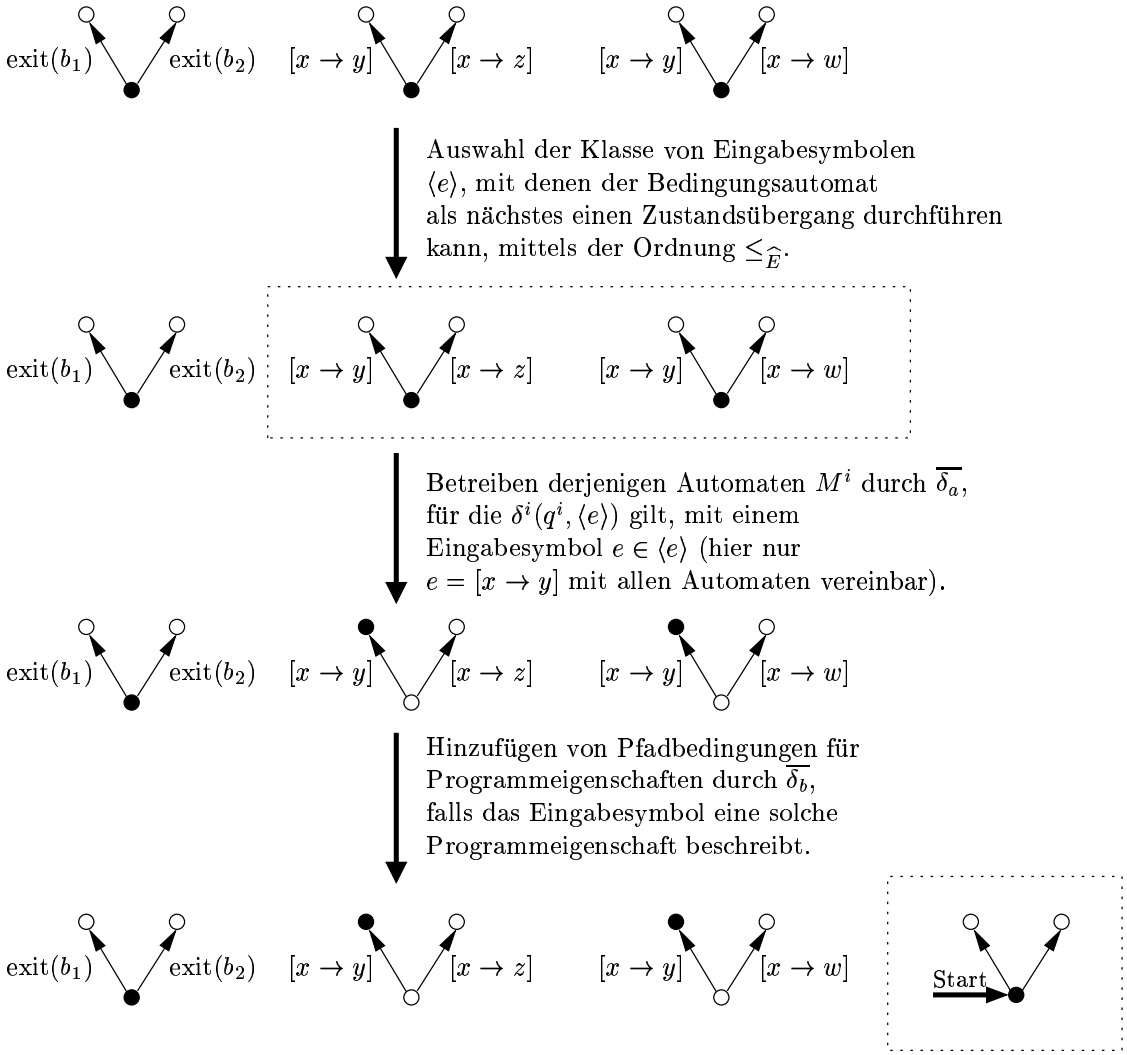


Abbildung 5.12: Prinzip des Betreibens von Bedingungsautomaten.

Erreichen einzelne Automaten ihre Endzustände, so werden sie aus der Menge der betriebenen Automaten entfernt. Wenn die Menge der betriebenen Automaten leer ist, dann hat der gesamte Bedingungsautomat anschaulich gesehen einen Endzustand erreicht. Dies geht in der nachfolgenden Definition in die Beschreibung der Sprache eines Bedingungsautomaten mit ein.

Definition 5.41 (Sprache eines Bedingungsautomaten)
Für einen Automaten $M^c = (Q^c, E, \delta^c, q_0^c, F^c, \text{undef})$ *und einen Bedingungsautomaten* C *wie in Definition 5.40 beschrieben, bezeichne*

$$\mathcal{L}(C) := \{\pi \in E^* \mid \delta(\{(M^c, q_0^c)\}, \pi) = \emptyset\}$$

die Menge der Wörter aus der Sprache des Bedingungsautomaten C.

Definition 5.42 (Von Bedingungsautomaten induzierte Programminstanzenmenge)
Sei M_a *ein Automat gemäß Definition 5.22 und sei* C *ein Bedingungsautomat wie in Definition 5.40 beschrieben. Dann bezeichne analog zu Definition 5.35*

$$M_a \triangleright C := \bigcup_{\pi \in \mathcal{L}(C)} M_a \triangleright \pi$$

die von einem Bedingungsautomaten induzierte Wortteilmenge der Sprache des Automaten M_a *bzw. die* von einem Bedingungsautomaten induzierte Menge von Programminstanzen.

Definition 5.43 (Darstellung eines Bedingungsautomaten durch einen DEA)
Sei C *ein Bedingungsautomat wie in Definition 5.40 beschrieben. Identifiziert man die unterschiedlichen Mengen von betriebenen Automaten mit ihren jeweiligen aktuellen Zuständen als eine neue Zustandsmenge, und setzt man den Startzustand des Bedingungsautomaten als Startzustand, sowie alle Konstellationen des Bedingungsautomaten, in denen dieser nur noch die leere Menge von betriebenen Automaten mitführt, als Endzustände, so erhält man eine Repräsentation des Bedingungsautomaten in der Form eines einzelnen DEA nach Definition 5.9. Dieser soll als* DEA–Repräsentation des Bedingungsautomaten C *bezeichnet werden.*

Diese DEA–Repräsentation eines Bedingungsautomaten kann z.B. dazu verwendet werden, um die Menge $M_a \triangleright C$ explizit in der Form der Sprache eines Automaten zu berechnen. Des weiteren wird die Zustandsmengengröße dieser DEA–Repräsentation später als Grundlage von Kostenabschätzungen verwendet werden.

Für einen Automaten M^c, in dessen Sprache keine Symbole vorkommen, die Programmeigenschaften beschreiben, ist die Definition der Sprache $\mathcal{L}(C)$ des Bedingungsautomaten mit Startzustand (M^c, q_0^c) offensichtlich äquivalent zu $\mathcal{L}(M^c)$. Sind in der Sprache hingegen Symbole, die Programmeigenschaften beschreiben, vorhanden, so wird bei Eingabe eines solchen Symbols an den Bedingungsautomaten ein weiterer Automat zur Verarbeitung hinzugefügt, der eine Pfadbedingungung für diese Programmeigenschaft darstellt. Von Bedingungsautomaten akzeptierte Wörter müssen sämtliche mitgeführten Automaten in ihre Endzustände überführen. Damit kann garantiert werden, daß in der Programminstanz, die durch das an den Automaten eingegebene Wort beschrieben wird, sämtliche geforderten Programmeigenschaften auch tatsächlich gelten. Diese Eigenschaft wird durch die folgende Definition formal eingeführt.

Definition 5.44 (Widerspruchsfreie Pfadbedingung)
Ein Automat M *heiße* widerspruchsfreie Pfadbedingung *für eine Programmeigenschaft* $\rho_a(y)$ *mit* $\rho_a \in \mathcal{H}_A$ *und* $y \in \text{poss}(\rho_a)$, *mit der Notation* $M \overset{r.a.}{\Longrightarrow} \rho_a(y)$, *falls die folgenden Bedingungen erfüllt sind:*

1. *Es gilt* $M \Longrightarrow \rho_a(y)$, *d.h. in jeder Programminstanz aus* $M_a \triangleright M$ *gilt die Programmeigenschaft* $\rho_a(y)$ *mit Möglichkeit* y.
2. *Jede der Programmeigenschaften, die durch Eingabe eines Symbols* $[\rho_{a'}'(y')]$ *an den Automaten* M *beschrieben wird, ist ebenfalls in der Menge* $M_a \triangleright M$ *von Programminstanzen erfüllt.*

Satz 5.9 (Bedingungsautomaten sind widerspruchsfreie Pfadbedingungen)
Seien für alle in einem Programm vorkommenden Programmeigenschaften $\rho_{a'} \in \mathcal{H}_A$ für alle $a' \in A$ und alle Möglichkeiten $y' \in \text{poss}(\rho_{a'})$ jeweils Pfadbedingungen der Form $M^{a'} \Longrightarrow \rho_{a'}(y')$ bekannt. Sei $a \in A$ eine Anweisung und $\rho_a \in \mathcal{H}_A$ eine Programmeigenschaft mit Möglichkeit $y \in \text{poss}(\rho_a)$. Der Automat M^c mit Startzustand q_0^c bezeichne eine Pfadbedingung für diese Programmeigenschaft, d.h. es gelte $M^c \Longrightarrow \rho_a(y)$. Dann ist der Bedingungsautomat C mit Startzustand (M^c, q_0^c) nach Definition 5.40 eine widerspruchsfreie Pfadbedingung gemäß Definition 5.44 für die Programmeigenschaft $\rho_a(y)$.

Beweis: Die Art der gemeinsamen Verarbeitung von mehreren Automaten anhand der Ordnung $\leq_{\widehat{E}}$ ist offensichtlich eine Verallgemeinerung der Operation $\cap_{M_a}$. Damit stellt der Bedingungsautomat offensichtlich eine gültige Beschreibungsform einer Menge von induzierten Programminstanzen dar.

Jeder vom Bedingungsautomaten mitgeführte betriebene Automat schränkt die Sprache des Automaten M^c und damit die Menge der induzierten Programminstanzen ein. Wenn der Automat M^c von einem Zustand aus ein Eingabesymbol akzeptieren würde, für das einer der betriebenen Automaten nach Fall 2 aus der Definition von $\overline{\delta_a}$ in Definition 5.40 ein undefiniertes Ergebnis zur Folge hätte, wird dieses Eingabesymbol vom Bedingungsautomaten C insgesamt nicht akzeptiert. Falls der Automat M^c im Vergleich zum Automaten M_a ein Eingabesymbol nach Definition 5.31 ausläßt, das einer der betriebenen Automaten als nächstes akzeptieren kann, so wird der Bedingungsautomat insgesamt dieses Eingabesymbol akzeptieren. Während für das ausgelassene Eingabesymbol des Automaten M^c bei der Betrachtung der Menge der von M^c induzierten Programminstanzen beliebige Möglichkeiten zulässig sind, stellt das Integrieren dieses Eingabesymbol in den Bedingungsautomaten durch die damit verbundene Festlegung i.A. ebenfalls eine Einschränkung der Menge der induzierten Programminstanzen dar.

Damit ist $M_a \triangleright C \subseteq M_a \triangleright M^c$. Da bereits in allen Programminstanzen $M_a \triangleright M^c$ die Programmeigenschaft $\rho_a(y)$ gilt, gilt diese speziell auch in allen Programminstanzen $M_a \triangleright C$, weshalb Eigenschaft 1 aus Definition 5.44 erfüllt ist.

Die als Voraussetzung gegebenen Pfadbedingungen $M^{a'} \Longrightarrow \rho_{a'}(y')$ für Anweisungen $a' \in A$ haben die Eigenschaft, daß sie eine Aussage darüber treffen, für welche Mengen von Programminstanzen $M_{a'} \triangleright M^{a'}$ die Gültigkeit von Eigenschaft $\rho_{a'}(y')$ gefolgert werden kann. Bei Eingabe eines Symbols der Form $[\rho_{a'}(y')]$ an den Bedingungsautomaten wird diese Pfadbedingung zur Menge der vom Bedingungsautomaten gleichzeitig betriebenen Automaten hinzugefügt. Der Teil des Eingabewortes, der ab diesem Zeitpunkt an den Bedingungsautomaten eingegeben wird, muß nach Definition 5.40 auch diesen mitgeführten Automaten in einen Endzustand überführen. Da dieser nach Voraussetzung eine Pfadbedingung ist, gilt die Programmeigenschaft in allen von diesem Teil des Eingabewortes induzierten Menge von Programminstanzen, die die Anweisung a' erreichen.

Damit akzeptiert der Bedingungsautomat nur Eingabewörter, bei denen für sämtliche darin vorkommenden Programmeigenschaften beschreibenden Symbole auch deren Gültigkeit in der Menge von induzierten Programminstanzen garantiert werden kann. Dies ist Eigenschaft 2 aus Definition 5.44, weswegen insgesamt die Behauptung gilt. □

In Abbildung 5.13 ist ein Beispiel für einen Bedingungsautomaten dargestellt. Teilabbildungen (a) und (b) zeigen nochmals die bereits vorgestellten Automaten M_y^1 und M_z^1, die Pfadbedingungen dafür sind, daß $(x \to y)_a$ bzw. $(x \to z)_a$ gilt. In Teilabbildung (c) ist eine Pfadbedingung für die Programmeigenschaft $(*x \to \ldots)_{a'}$ vorgegeben. Diese beschreibt die Menge aller Programminstanzen, in denen bei Anweisung a' (wie in Abbildung 5.5 definiert) die zweifache Dereferenzierung der Variablen x bei Anweisung a' die Ziele A, B oder C besitzen kann. Auf die konkrete Berechnung von solchen Pfadbedingungen soll erst später eingegangen werden. Offensichtlich beschreibt der Automat M^2 eine Menge von Programminstanzen, die Anweisung a' erreichen. Unter der durch das Eingabesymbol $[x \to y]$ ausgedrückten Annahme, daß bei Anweisung a die Variable x das Zeigerziel y besitzt, ergibt sich die linke Hälfte des Automaten M^2 mit seinen Ergebniszuständen aus der Betrachtung der möglichen Inhalte der Variablen y, was analog zur bisherigen Betrachtung der Inhalte der Variablen x geleistet werden kann. Unter der anderen Annahme, daß x als Zeigerziel die Variable z besitzt, ergibt sich analog die rechte Hälfte des Automaten M^2.

Der Automat M^2 beschreibt bisher die Ergebnisse einer Analyse von *independent attributes*.

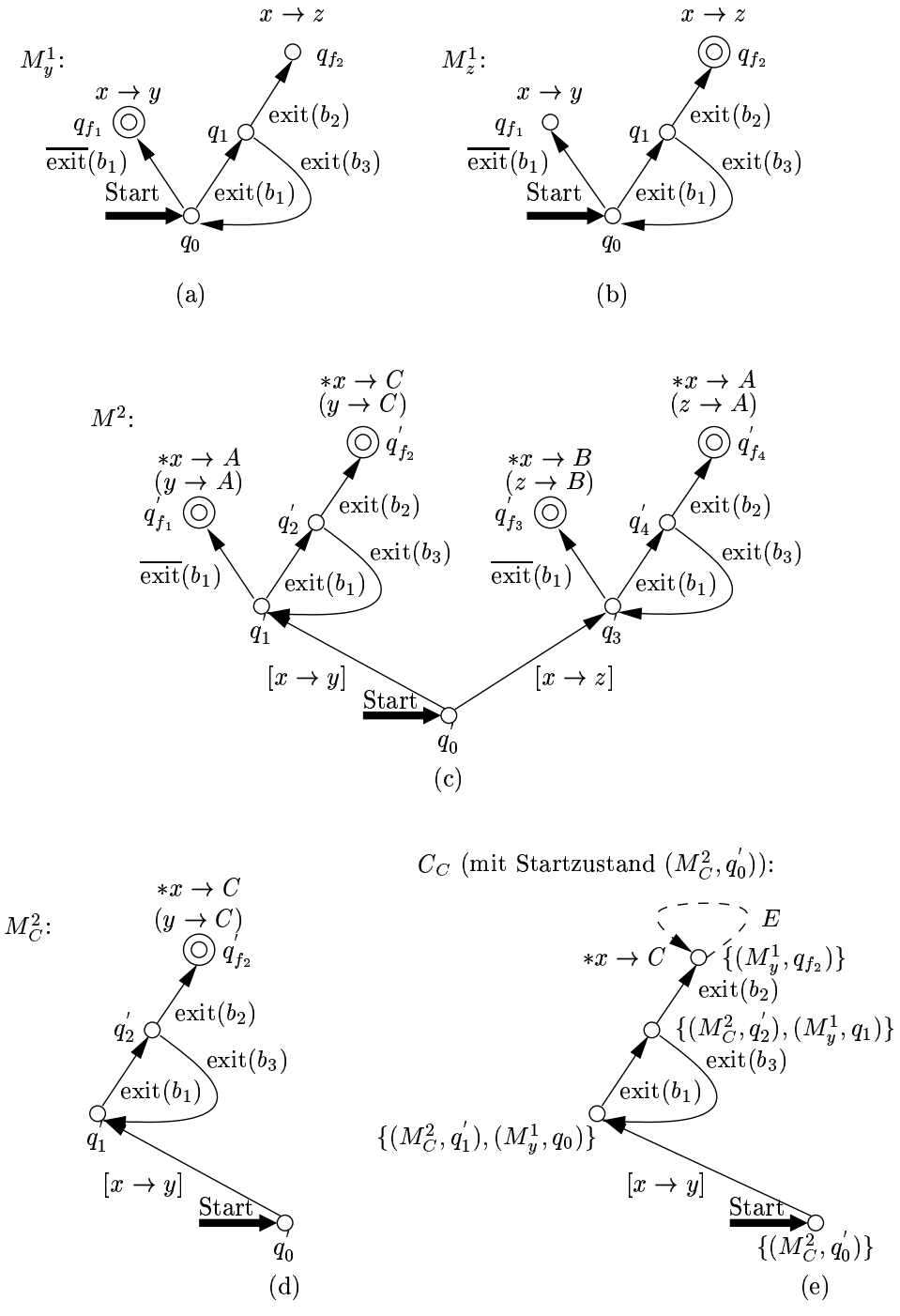


Abbildung 5.13: Beispiel für einen Bedingungsautomaten.

Dies kann man daran erkennen, daß nur durch Betrachtung des Automaten M^2 die Variablen A, B und C als Ziele der zweifachen Dereferenzierung der Variablen x bei Anweisung a (bzw. $a^{'}$ gemäß Vereinbarung 5.2) in Frage kommen, was im Kapitel "Motivation" bereits als nicht–exaktes Ergebnis erkannt wurde.

Den Übergang zu einer Analyse von *relational attributes* leistet nun Definition 5.40 eines Bedingungsautomaten. Dabei wird zunächst vom Automaten M_C^2 (gemäß Definition 5.38) ausgegangen, der in Abbildung 5.13(d) dargestellt ist. Für die Sprache des Automaten M_C^2 sind nur diejenigen Zustände relevant, von denen aus der Endzustand $q_{f_2}^{'}$ noch erreichbar ist. Eine entsprechende Auswahl dieser Zustände wurde in Abbildung 5.13(d) bereits vorgenommen. Durch diesen Automaten als Startzustand (d.h. den betriebenen Automaten $\{(M_C^2, q_0^{'})\}$) sei der Bedingungsautomat C_C definiert. In Abbildung 5.13(e) ist das Ergebnis dieser Konstruktion dargestellt.

Die Zustände des Bedingungsautomaten entsprechen Mengen von betriebenen Automaten, die in der Abbildung eingezeichnet sind. Bei Eingabe des Bedingungssymbols $[x \rightarrow y]$ vom Startzustand $\{(M_C^2, q_0^{'})\}$ aus, wird gemäß Definition 5.40 der Automat M_y^1 in seinem Startzustand der Menge der vom Bedingungsautomaten C_C betriebenen Automaten hinzugefügt.

Laut Definition 5.41 sind nur solche Worte in der Sprache $\mathcal{L}(C_C)$ des Bedingungsautomaten, deren Eingabe an die Zustandsübergangsfunktion δ aus Definition 5.40 vom Startzustand $\{(M_C^2, q_0^{'})\}$ aus die leere Menge von betriebenen Automaten ergibt. Bei Erreichen des Endzustandes $q_{f_2}^{'}$ von M_C^2 hat der mitgeführte Automat M_y^1 allerdings keinen Endzustand, sondern den Zustand q_{f_2} erreicht, von dem aus auch kein Endzustand mehr erreicht werden kann: gemäß der Forderung nach Gültigkeit von Gleichung (5.4) in Definition 5.35 führen Zustandsübergänge aus q_{f_2} nur wieder nach q_{f_2}.

Die Sprache des Bedingungsautomaten C_C ist also leer, da es kein Eingabewort gibt, das alle betriebenen Automaten in Endzustände überführen würde. Damit existiert keine Programminstanz, in der die zweifache Dereferenzierung von x bei Anweisung a (bzw. $a^{'}$) die Variable C ergeben kann. Dieses Ergebnis deckt sich mit dem Ergebnis aus dem Motivationsbeispiel, das sich auf das demgegenüber nur geringfügig veränderte Running Example übertragen läßt. Die Berechnung des Bedingungsautomaten C_C entspricht also offensichtlich einem Teil einer Analyse von *relational attributes* für die Ziele der zweifachen Dereferenzierung von x bei Anweisung a (bzw. $a^{'}$).

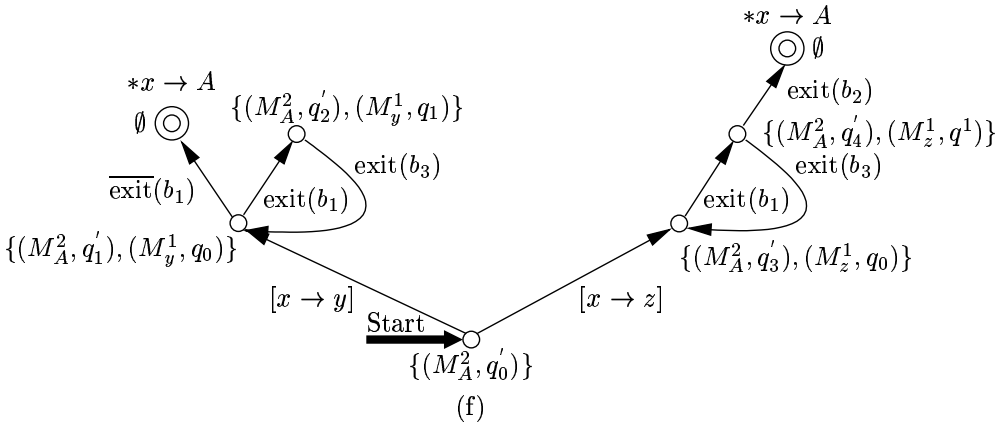
In Abbildung 5.13(f) und (g) sind die Bedingungsautomaten C_A und C_B dargestellt, die genau wie C_C konstruiert werden. Offensichtlich ist der Automat C_A der einzige von den drei konstruierten Bedingungsautomaten, dessen Sprache nicht–leer ist. Dies deckt sich mit dem Ergebnis der Betrachtung aus der Motivation, daß die Variable `A` das einzige mögliche Zeigerziel für die zweifache Dereferenzierung der Variablen `x` am Ende des Motivationsbeispiels (bzw. des Running Examples) ist. Damit sind die vorgestellten Bedingungsautomaten für das Running Example in der Lage gewesen, eine Analyse von *relational attributes* zu leisten. Daß dieses Problem allgemein durch Bedingungsautomaten gelöst werden kann, war die Aussage von Satz 5.9.

In Abbildung 5.13(h) ist schließlich noch eine Zusammenfassung der Bedingungsautomaten C_A, C_B und C_C in einen einzigen Bedingungsautomaten dargestellt. Wie dies bei den bisher verwendeten Pfadbedingungen auch der Fall war, besitzen die verschiedenen Bedingungsautomaten gemeinsame identische Teile. Die grau eingezeichneten Kanten beschreiben Zustandsübergänge, die bei der Betrachtung von Bedingungsautomaten gar nicht erst berücksichtigt werden müssen, da bereits vor ihrer Durchführung erkennbar ist, daß nach dem Zustandsübergang nicht mehr alle betriebenen Automaten ihre jeweiligen Endzustände erreichen können.

Die theoretische Beschreibung von Bedingungsautomaten geht von der in Abbildung 5.13(e) bis (g) gezeigten Vorgehensweise aus, für jedes mögliche Ergebnis des Automaten M^2 einen Bedingungsautomaten zu konstruieren, und jeweils dessen akzeptierte Sprache zu betrachten. In der praktischen Realisierung des Verfahrens wird dagegen eine gemeinsame Berechnung sämtlicher möglichen Ergebnisse des Bedingungsautomaten stattfinden, wie dies in Abbildung 5.13(h) angedeutet ist. Da beide Vorgehensweisen aber offensichtlich sehr eng miteinander verwandt sind, und eine Umwandlung von der einen Beschreibungsform in die andere trivial ist, wird auf diesen Unterschied im Weiteren nicht mehr eingegangen werden.

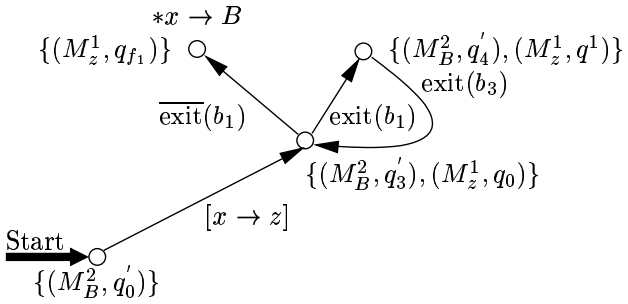
Wie sich das Prinzip der Konstruktion von Bedingungsautomaten in einer praktisch anwendbaren Analyse realisieren läßt, werden die weiteren Kapitel dieser Arbeit zeigen.

C_A (mit Startzustand $(M_A^2, q_0^{'})$):

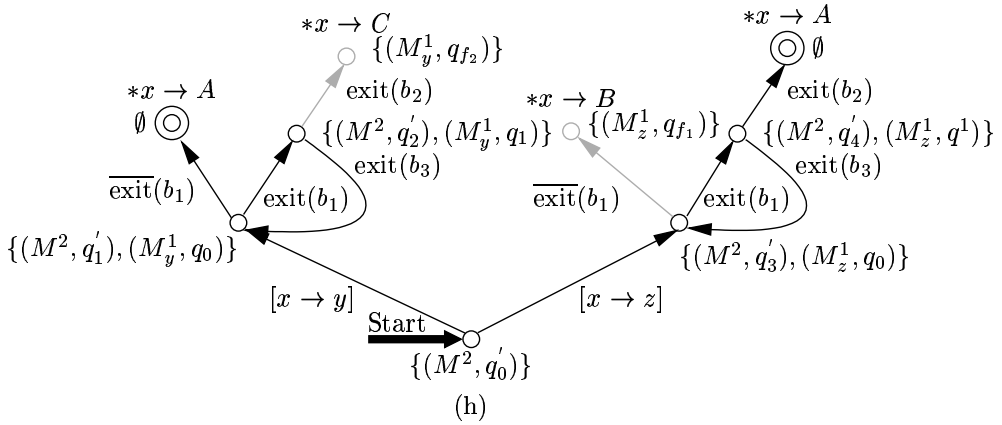


(f)

C_B (mit Startzustand $(M_B^2, q_0^{'})$):



(g)



(h)

Abbildung 5.14: Beispiel für Bedingungsautomaten: C_A und C_B, sowie eine Kombination der Bedingungsautomaten C_A, C_B und C_C in einen einzigen Bedingungsautomaten.

5.8 Approximative Berechnung von Bedingungsautomaten

In späteren Kapiteln werden Aussagen über maximale Analysekosten gemacht werden. Aus diesen läßt sich eine Notwendigkeit erkennen, in bestimmten Situationen die Exaktheit der Lösung einer Analyse von *relational attributes* mittels Bedingungsautomaten aufzugeben, um noch eine praktische Realisierbarkeit des Verfahrens garantieren zu können.

Ohne an dieser Stelle bereits auf die Kostenbetrachtung einzugehen, kann man eine intuitiv naheliegende Vereinfachung der Berechnung von Bedingungsautomaten angeben. Dies soll im Folgenden definiert werden. In welchem Zusammenhang diese vereinfachte Berechnungsweise eingesetzt werden soll, wird in Kapitel 11 diskutiert werden.

Definition 5.45 (Sichere Approximation von widerspruchsfreien Pfadbedingungen)
Sei $a \in A$ eine Anweisung, und der Automat M_a wie in Definition 5.22 angegeben. Sei weiter $\rho_a \in \mathcal{H}_A$ eine Programmeigenschaft mit Möglichkeit $y \in \text{poss}(\rho_a)$, und sei $M = (Q, E, \delta, q_0, F, \text{undef})$ eine widerspruchsfreie Pfadbedingung für diese Programmeigenschaft, d.h. daß nach Definition 5.44 gilt: $M \overset{r.a.}{\Longrightarrow} \rho_a(y)$. Ein Automat $M' = (Q', E, \delta', q_0', F', \text{undef})$ werde als sichere Approximation der widerspruchsfreien Pfadbedingung *M bezeichnet, wenn die folgenden Eigenschaften gelten:*

1. *In allen Programminstanzen $M_a \triangleright M'$ gilt ebenfalls die Programmeigenschaft $\rho_a(y)$.*
2. *$M_a \triangleright M' \supseteq M_a \triangleright M$, d.h. die vom Automaten M' induzierte Menge von Programminstanzen ist eine Obermenge derjenigen Programminstanzen, die von der widerspruchsfreien Pfadbedingung M induziert werden.*

Die Menge von Programminstanzen $M_a \triangleright M'$ soll laut dieser Definition eine Obermenge der Menge $M_a \triangleright M$ von Programminstanzen sein, in denen garantiert die Programmeigenschaft $\rho_a(y)$ zusammen mit allen durch Eingabesymbole beschriebenen weiteren Programmeigenschaften gilt. Trotzdem soll nach Eigenschaft 1 aus obiger Definition in allen diesen Programminstanzen noch die Programmeigenschaft $\rho_a(y)$ gelten. Damit lockert die Definition einer sicheren Approximation einer widerspruchsfreien Pfadbedingung die Forderung der Widerspruchsfreiheit. In einer Programminstanz, die in der Mengendifferenz zwischen der von der widerspruchsfreien Pfadbedingung und von ihrer sicheren Approximation induzierten Menge von Programminstanzen enthalten ist, können nicht alle von Eingabesymbolen geforderten Programmeigenschaften gleichzeitig gültig sein.

Eine vereinfachte Berechnung von Bedingungsautomaten, die ein sicheres approximatives Ergebnis liefert, kann man aufgrund dieser Beobachtung dadurch realisieren, daß man nicht für jedes Eingabesymbol der Form $[\rho_a(y)]$ einen Automaten zur Menge der vom Bedingungsautomaten betriebenen Automaten hinzufügt. Im Vergleich zu einer exakten Lösung einer Analyse von *relational attributes* mittels Bedingungsautomaten wird damit nur eine Teilmenge der dafür benötigten betriebenen Automaten verwendet. Daß diese Vorgehensweise tatsächlich ein sicheres Ergebnis liefert, wird im folgenden Satz ausgesagt.

Satz 5.10 (Approximative Vereinfachung von Bedingungsautomaten)
Modifiziert man die Ausführungsvorschrift eines Bedingungsautomaten nach Definition 5.40 derart, daß nur bei einem Teil der Eingabesymbole der Form $[\rho_a(y)]$ die entsprechende Pfadbedingung zur Menge der vom Bedingungsautomaten gleichzeitig betriebenen Automaten hinzugefügt wird, so beschreibt dieser modifizierte Bedingungsautomat eine sichere Approximation des unmodifizierten Bedingungsautomaten nach Definition 5.45.

Beweis: Dadurch, daß der modifizierte Bedingungsautomat den gleichen Startzustand wie der unmodifizierte Bedingungsautomat besitzt, und dieser eine Pfadbedingung für die betrachtete Programmeigenschaft darstellt, kann man auch in allen vom modifizierten Bedingungsautomaten induzierten Programminstanzen die Gültigkeit der betrachteten Programmeigenschaft folgern. Dies erfüllt Eigenschaft 1 aus Definition 5.45.

Wie bereits im Beweis von Satz 5.9 beschrieben wurde, stellt jeder betriebene Automat eine Einschränkung der Menge der induzierten Programminstanzen dar. Das Weglassen eines solchen betriebenen Automaten verursacht daher eine Vergrößerung der Menge der induzierten Programminstanzen. Dies erfüllt damit Eigenschaft 2 aus Definition 5.45. □

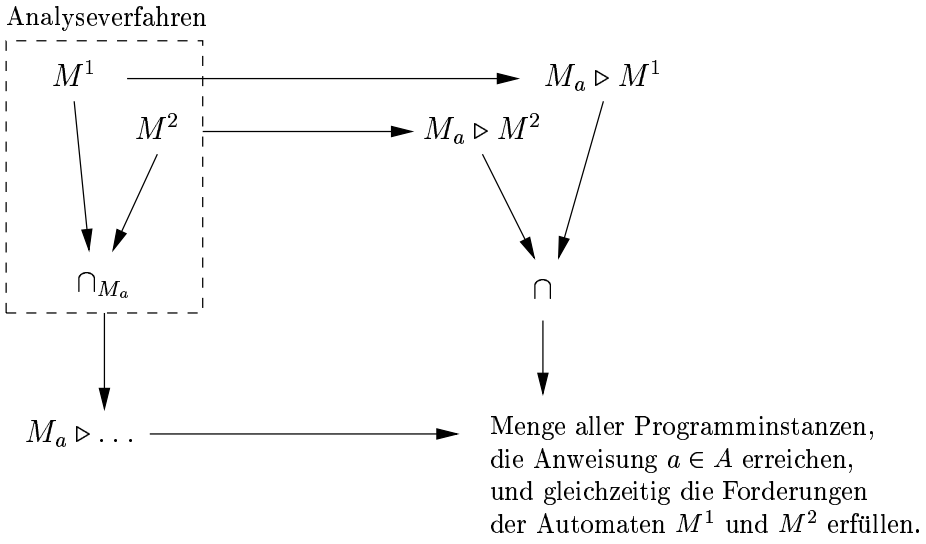


Abbildung 5.15: Veranschaulichung des Prinzips der von Automaten induzierten Mengen von Programminstanzen und der Operation $\cap_{M_a}$ auf diesen Automaten.

Die sichere Approximation von widerspruchsfreien Pfadbedingungen stellt anschaulich gesehen eine Auswahl von Programmeigenschaften dar, die gleichzeitig in einer Programminstanz gelten sollen. Dadurch gewinnt man selbst bei Verwendung der approximativen Variante des Verfahrens aus Satz 5.10 gegenüber traditionellen Analysemethoden den Vorteil, daß man eine frei wählbare Mindestgenauigkeit erreichen kann, die sich durch die Anzahl der gleichzeitig geltenden Programmeigenschaften definiert.

Traditionelle Analysemethoden von *independent attributes*, die keinerlei weitere Programmeigenschaften in die Betrachtung mit einbeziehen, kann man als Bedingungsautomaten interpretieren, bei denen keinerlei betriebene Automaten mitgeführt werden. Damit stellen die Ergebnisse dieser Analysen auch in der Interpretation von Definition 5.45 sichere Approximationen von Analysen von *relational attributes* dar.

5.9 Zusammenfassung

Für ein gegebenes Eingabeprogramm wurde ein Automat M_a definiert, der die Menge aller Programminstanzen(–beschreibungen), die eine bestimmte Anweisung $a \in A$ erreichen, durch die von ihm akzeptierte Sprache darstellt. Weiter wurde eine Vorgehensweise vorgestellt, wie man durch weitere Automaten M eine Teilmenge der von M_a beschriebenen Menge von Programminstanzen zusammenfassend beschreiben kann. Diese Teilmenge $M_a \triangleright M$ wurde als die von einem Automaten induzierte Wortteilmenge der Sprache des Automaten M_a bezeichnet. Die durch diese Wortteilmenge beschriebene Menge von Programminstanzen wurde als die vom Automaten M induzierte Menge von Programminstanzen bezeichnet. Für mehrere solche Automaten wurde eine Schnittoperation $\cap_{M_a}$ angegeben, die anschaulich betrachtet die durch diese Automaten zum Ausdruck gebrachten Forderungen an die jeweils induzierte Menge von Programminstanzen in einer gemeinsamen Beschreibung vereinigt. In Abbildung 5.15 ist dieses Prinzip im Überblick dargestellt. Die Automaten M^1 und M^2 sollen beide Wortteilmengen induzierende Automaten sein, die Forderungen an die Menge aller Programminstanzen darstellen, die die Anweisung $a \in A$ erreichen, und damit jeweils eine Teilmenge dieser Menge von Programminstanzen eindeutig charakterisieren. Entlang der oberen Pfeile ist die naheliegende Vorgehensweise dargestellt, wie man die Menge aller Programminstanzen erhalten kann, die Anweisung $a \in A$ erreichen und die Forderungen der beiden Automaten M^1 und

M^2 gleichzeitig erfüllen: man berechnet jeweils die von den beiden Automaten induzierten Wortteilmengen der Sprache des Automaten M_a, und bildet die Schnittmenge aller dieser Worte. Für diese Vorgehensweise muß man den Automaten M_a explizit berechnen, und die Schnittmenge über potentiell unendliche Mengen von Worten bilden. Die Alternative dazu besteht in der Verwendung der Operation $\cap_{M_a}$. Entlang der linken Pfeile ist dargestellt, wie zuerst aus den Automaten M^1 und M^2 ein Automat $M^1 \cap_{M_a} M^2$ berechnet wird, der dann diejenige Wortteilmenge induziert, in denen ebenfalls die Forderungen von M^1 und M^2 gleichzeitig erfüllt sind. Daß beide Vorgehensweisen die gleiche Menge von Programminstanzen beschreiben, ist die Aussage von Satz 5.7. Damit kann man ohne die Berechnung von M_a und die Bildung von Schnittmengen von unendlichen Sprachen alleine durch die Betrachtung von Wortteilmengen induzierenden Automaten Mengen von Programminstanzen charakterisieren und in einer handhabbaren Art und Weise darstellen und verarbeiten. Dies ermöglicht die Beschränkung der Verantwortlichkeit des Analyseverfahrens auf die Behandlung von solchen Automaten, wie dies in der Abbildung durch die gestrichelte Umrandung symbolisiert wird.

Die Verwendung dieser Form der Behandlung von Mengen von Programminstanzen auf das Problem der Programmanalyse geschieht durch die Definition von Pfadbedingungen. Diese sind Wortteilmengen induzierende Automaten, für die man die Aussage treffen kann, daß in allen von diesen Automaten induzierten Programminstanzen eine bestimmte Programmeigenschaft wie z.B. $x \rightarrow y$ gelten muß. Durch die Verknüpfung von mehreren solchen Pfadbedingungen durch die Operation $\cap_{M_a}$ kann man Automaten berechnen, die Mengen von Programminstanzen induzieren, in denen mehrere solche Programmeigenschaften gelten müssen. Dies ist das Prinzip des Übergangs auf eine Analyse von *relational attributes*. Eine Lösung einer Analyse von *relational attributes* wird in der vorgestellten Sichtweise so definiert, daß man die Menge aller Programminstanzen betrachtet, in denen sämtliche durch Symbole der Form $[x \rightarrow y]$ beschriebenen Programmeigenschaften auch tatsächlich gelten. Ein Automat, der eine solche Menge von Programminstanzen induziert, wird als widerspruchsfreie Pfadbedingung bezeichnet.

In der Form von Bedingungsautomaten wird eine Vorgehensweise vorgestellt, wie man mehrere Pfadbedingungen gemeinsam betreiben kann, um damit widerspruchsfreie Pfadbedingungen zu erhalten. Diese entspricht einer Verallgemeinerung der Operation $\cap_{M_a}$.

Schließlich wurde noch eine Definition von sicheren Approximationen von widerspruchsfreien Pfadbedingungen gegeben, und gezeigt, daß eine intuitiv naheliegende Weise, Bedingungsautomaten unter Einsparung von Berechnungsaufwand zu betreiben, eine solche sichere Approximation darstellt.

Kapitel 6

Programmrepräsentationen

Um die Analyse von *relational attributes* in ein Erreichbarkeitsproblem zu transformieren und die Beschreibung der Queryverarbeitung, wie sie in Abschnitt 3.3.3.2 bereits informell vorgestellt wurde, zu erleichtern, wird in dieser Arbeit eine neue Programmrepräsentation vorgestellt. Dieser liegt die Beobachtung zugrunde, daß bei der Ausführung einer komplexen Anweisung, in Abschnitt 3.1 als Zuweisung bezeichnet, stets mehrere der folgenden Aktionen ausgeführt werden:

- Lesen des Wertes einer Variable
- Verändern des Wertes einer Variable
- Dereferenzieren eines Zeigers (der Adresse einer Variablen)
- Generieren der Adresse einer Variable

Dabei bedarf es einer Analyse der Zuweisung, um deren genauen Effekt herauszufinden. Betrachtet man z.B. die Zuweisung `**x = *y`, so wird bei deren Ausführung der Inhalt der Variable `y` ausgelesen, dereferenziert und der Inhalt derjenigen Variablen, die aus der Dereferenzierung resultiert, wiederum ausgelesen. Die Variable `x` wird ebenfalls ausgelesen und dereferenziert, dann der Wert der Variablen, die das Ergebnis der Dereferenzierung ist, ausgelesen und wiederum dereferenziert. Die Variable, die dabei resultiert, wird nun aber nicht ausgelesen, sondern ihr wird der Wert der vorher berechneten rechten Seite der Zuweisung zugewiesen. Aufgrund all dieser verschiedenen Vorgänge bei der Ausführung einer einzelnen komplexen Zuweisung, bei denen die genaue Verwendung einer dereferenzierten Variablen zusätzlich von der Art ihres Auftretens in der Zuweisung abhängt, wird zur Beantwortung der Frage, welchen Einfluß eine solche Zuweisung auf eine Erreichbarkeitsquery hat, eine Analyse der Zuweisung mit komplexen Fallunterscheidungen notwendig.

Das Ziel der neuen Programmrepräsentation ist es daher, einen Analysegraphen zu erzeugen, der die Analyse von Zeigerzielen in ein bedingtes Erreichbarkeitsproblem (wie bereits in Abschnitt 3.3.2.1.2 informell eingeführt) transformierbar macht, wodurch einerseits während der Analyse komplexe Untersuchungen der Auswirkungen von Anweisungen vermieden werden können, und andererseits die Korrektheit des Verfahrens anhand des direkten und einfachen Zusammenhangs zwischen der Graphstruktur des Analysegraphen und der Semantik der neuen Programmrepräsentation bewiesen werden kann.

6.1 Überblick

In dieser Arbeit werden drei verschiedene Arten von Programmdarstellungen verwendet.

- Am Anfang der Analyse steht die textuelle Beschreibung eines zu analysierenden Programmes, das der in Abschnitt 3.1 geforderten Form genügt. Die Auswirkung von Anweisungen in dieser ursprünglichen Darstellungsform kann durch eine traditionelle Semantikfunktion, wie in Abschnitt 4.2 angegeben, definiert werden.

```
if(...)
        v=1;
else
        v=2;
... = v;
```

```
if(...)
        v_1 ← 1;
else
        v_2 ← 2;
v_3 ← φ(v_1, v_2);
... ← v_3;
```

Abbildung 6.1: Beispiel für die Static Single Assignment–Form von Programmen: Programmcode (links) und SSA–Form (rechts)

- Als erste Ausprägung der neuen Programmrepräsentation wird eine ebenfalls textbasierte Darstellungsform eingeführt werden, deren Bestandteile Operationen genannt werden. Komplexe Zuweisungen im Eingabeprogramm werden in Sequenzen von einfacheren Grundoperationen, die eine Teilmenge aller Operationen darstellen, zerlegt. Diese Grundoperationen stellen die vier Bestandteile von Zuweisungen, wie sie in obiger Aufzählung beschrieben wurden, dar.

  Für diese Grundoperationen wird eine Semantikdefinition eingeführt, erweiterte Semantik genannt, anhand derer die Äquivalenz des ursprünglichen Programmes mit dieser neuen textuellen Darstellung gezeigt werden kann. Für Kontrollflußanweisungen werden weitere Operationen eingeführt, die aber aufgrund der Tatsache, daß ihre Verzweigungsbedingungen nicht interpretiert werden, keine Semantik besitzen, sondern nur den möglichen Kontrollfluß durch ein Programm beschreiben.

  Diese textuelle Darstellung mit ihrer erweiterten Semantikfunktion für Grundoperationen bildet die formale Basis dieser Arbeit. Die Beschreibung der Queryverarbeitung benutzt diese textuelle Darstellung als Grundlage. Anhand der erweiterten Semantikfunktion wird die Korrektheit der Queryberechnung bewiesen werden.

- Die dritte Form von Programmrepräsentation ist schließlich eine Transformation der textbasierten neuen Programmrepräsentation in einen Graphen. Jede Operation wird in diesem Graphen durch einen Knoten dargestellt. Zwischen den Operationen bestehen drei verschiedene Arten von Abhängigkeiten der Ergebnisse ihrer Ausführung untereinander. Diese Abhängigkeiten werden durch Kanten in diesem Graphen dargestellt.

  Als Ergebnis erhält man eine graphbasierte Darstellung der neuen Programmrepräsentation, die sozusagen als Nebeneffekt eine Veranschaulichung der textuellen Form der Programmrepräsentation darstellt, deren Hauptzweck aber in der Verwendung als Analysegraph liegt. Die später eingeführten Queries lassen sich damit als bedingtes Erreichbarkeitsproblem auf diesen Graphen realisieren.

6.2 Ähnliche Prinzipien aus der Literatur

Die im Weiteren vorgestellte neue Programmrepräsentation basiert im Prinzip auf zwei aus der Literatur bekannten Konzepten, die aber für sich jeweils nicht ausreichen, um die in dieser Arbeit verfolgte Zielsetzung zu erreichen.

6.2.1 Static Single Assignment

Static Single Assignment (SSA) [CFR+91] transformiert Eingabeprogramme in eine textuelle Darstellung, in der jeder Variablen im Programm nur an einer Stelle im Programm ein Wert zugewiesen wird. Dies wird durch eine Umbenennung der im Programm vorkommenden Variablen in ggfs. mehrere unterschiedliche Variablennamen erreicht. Ein Beispiel für die Transformation in diese Darstellungsform findet sich in Abbildung 6.1, in der das Programm links in der Abbildung durch Umbenennung der Variablen v in die Variablennamen v_1, v_2 und v_3 derart transformiert wird,

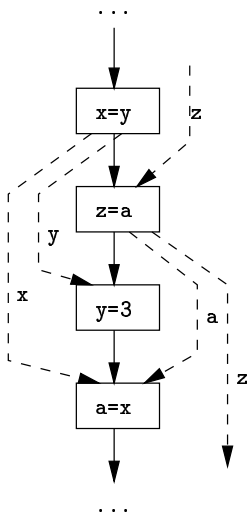


Abbildung 6.2: Prinzip von Use–Use–Chains.

daß jeder dieser neuen Variablen nur an einer Stelle im Programm ein Wert zugewiesen wird. An Kontrollfluß–Joins wählen sogenannte ϕ–Operatoren je nach durchlaufenem Programmpfad eine der mehreren Umbenennungen der Variablen aus, um den Wert der Variablen nach dem Kontrollfluß–Join zu bestimmen. Im Beispiel ergibt sich die Belegung von v_3 aus der Belegung von v_1, falls der then–Zweig ausgeführt wurde, und aus der Belegung von v_2, wenn der else–Zweig ausgeführt wurde.

Die Gemeinsamkeit zwischen SSA und der in dieser Arbeit vorgestellten Programmrepräsentation liegt in der Verwendung von "Fallunterscheidungskonstrukten" analog zu ϕ–Operatoren an Kontrollfluß–Joins.

SSA läßt jedoch keine weiteren Aussagen darüber zu, ob Belegungen von Variablen oder allgemein Analysefakten auf gemeinsamen Pfaden entstanden sein können. Im Prinzip werden die Fallunterscheidungen nur dazu verwendet, um die verschiedenen Möglichkeiten für die Belegung von Variablen in einer Zusammenfassung darzustellen. Eine Zuordnung von Variablenbelegungen zu konkreten Programmpfaden ist nicht möglich, ebenso können mehrere solche Fallunterscheidungskonstrukte nicht auf das Vorkommen ihrer jeweiligen Fälle auf gemeinsamen Pfaden untersucht werden. Stattdessen werden hier analog zu einer Analyse von *independent attributes* wieder sämtliche Kombinationen als gültig angesehen. Selbst bei Annotation der ϕ–Operatoren mit den jeweiligen Kontrollfluß–Joins, an denen sie auftreten, können keine weiteren Aussagen darüber getroffen werden, wie ϕ–Operatoren, die an verschiedenen Kontrollfluß–Joins auftreten, zu vereinheitlichen wären.

Von diesem Blickwinkel betrachtet könnte man die Programmdarstellung in dieser Arbeit als eine Verallgemeinerung der SSA–Programmrepräsentation, und die bei der Analyse erzeugten Automaten als Gegenstück zu den entsprechenden Zusammenfassungen von ϕ–Konstrukten interpretieren.

6.2.2 Use–Use–Chains

Da eine Aufgabe der Queries darin besteht, letzte Zuweisungen zu einer Variablen zu finden, bietet sich hierfür ein bekanntes Konzept aus der Compilertechnik an, die sog. Use–Use–Chains (z.B. [All70] [ASU88] [HS94]). Damit bezeichnet man bei traditionellen Programmgraphen die Verkettung von Anweisungen untereinander, die auf die gleiche Variable zugreifen. In Abbildung 6.2 ist ein einfaches Beispiel für Use–Use–Chains in traditionellen Programmgraphen dargestellt. I.A. verwendet nicht jede Anweisung die gleiche Menge von Variablen. Für jeweils eine Variable werden nun sämtliche Anweisungen, die diese Variable verwenden, verkettet, so daß ein effizienter Zugriff auf die nächste oder letzte Anweisung, die diese Variable verwendet oder verändert, möglich ist.

Traditionelle Use–Use–Chains eignen sich jedoch nicht, wenn zusätzlich zur letzten Verwendung

einer Variablen eine genaue Protokollierung des intraprozeduralen Pfades zwischen den Anweisungen benötigt wird, wie das in dieser Arbeit der Fall ist. Daher wird hier das Prinzip von Use–Use–Chains integriert und um zusätzliche Elemente erweitert, die eine solche Pfadprotokollierung ermöglichen. Die (mögliche) aufeinanderfolgende Verwendung der gleichen Variablen von zwei Operationen beschreibt dabei eine der Abhängigkeiten, die in der Graphdarstellung der neuen Programmrepräsentation als Grundlage zur Erzeugung von Kanten verwendet werden.

6.3 Textuelle neue Programmrepräsentation

Als Grundlage für die formale Beschreibung der neuen Programmrepräsentation und für die Definition der Queryberechnung dient in dieser Arbeit die im Folgenden eingeführte Beschreibung eines Eingabeprogrammes durch eine Menge von *Operationen*. Anschaulich betrachtet beschreiben *Grundoperationen* einfachere Bausteine von komplexen Zuweisungen, während *Kontrollflußoperationen* den möglichen Kontrollfluß durch das Eingabeprogramm spezifizieren.

6.3.1 Struktur

6.3.1.1 Grundoperationen

Eine Sequenz von Zuweisungen wird, wie bereits erwähnt, in eine Sequenz von Grundoperationen transformiert, die bei Ausführung die gleiche Auswirkung (bzgl. der nachfolgend definierten Semantik) wie die ursprünglichen Zuweisungen haben soll. Für die vorgestellte Programmklasse benötigt man dazu folgende Grundoperationen. Diese werden zunächst auf syntaktischer Ebene unter informeller Angabe ihres späteren Verwendungszweckes vorgestellt.

Definition 6.1 (Zeichenketten)
Die Menge P sei definiert als Menge aller Zeichenketten.

Definition 6.2 (Grundoperationen)
Die Menge Ω_0 von Grundoperationen sei wie folgt definiert. Dabei bezeichne V wie in Definition 4.2 die Menge aller Variablen und P wie in Definition 6.1 die Menge aller Zeichenketten. Die tiefergestellten Anteile der eingeführten Operationen entsprechen dabei Parametern.

- *Auslesen des Wertes einer Variable (Get–Operation): für $v \in V$ und für $\omega' \in \Omega_0$ bezeichnen* $\mathrm{get}_v \in \Omega_0$ *bzw.* $\mathrm{get}_{\omega'} \in \Omega_0$ *Get–Operationen.*
- *Verändern des Wertes einer Variable (Set–Operation): für $v \in V$, $\omega' \in \Omega_0$, $\omega'' \in \Omega_0$ und $p \in P$ eine Zeichenkette bezeichnen* $\mathrm{set}_{v,\omega',p} \in \Omega_0$ *bzw.* $\mathrm{set}_{\omega'',\omega',p} \in \Omega_0$ *Set–Operationen.*
- *Generieren der Adresse einer Variable (Adress–Operation): für $v \in V$ bezeichnet* $\mathrm{addr}_v \in \Omega_0$ *eine Adress–Operation.*
- *Dereferenzieren einer Variable (Dereferenzierungs–Operation): für $\omega' \in \Omega_0$ und $p \in P$ eine Zeichenkette bezeichne* $\mathrm{deref}_{\omega',p} \in \Omega_0$ *eine Dereferenzierungs–Operation.*

Diese Menge von Grundoperationen wird im nächsten Abschnitt auf die Menge Ω aller Operationen, die in dieser Arbeit verwendet werden, erweitert.

6.3.1.2 Kontrollflußoperationen

Die bisherige Umsetzung von Zuweisungen in Grundoperationen beschäftigte sich mit unverzweigten Programmabschnitten. Als nächster Schritt muß nun eine Möglichkeit geschaffen werden, um die durch If- und While–Anweisungen bedingten Verzweigungen in Programmen darzustellen. Wie bereits aus der Definition der zugelassenen Programmklasse hervorgeht, ist die Struktur von Eingabeprogrammen in dieser Arbeit hierarchisch gegliedert: ein Programm ist ein Programmblock, Programmblöcke bestehen aus Sequenzen von Anweisungen, von denen die If– und While–Anweisungen

selbst wieder Programmblöcke als untergeordnete Blöcke für den then– und else–Zweig bzw. Schleifenrumpf enthalten, etc. Diese Strukturierung soll bei der Transformation in die neue Programmrepräsentation erhalten bleiben.

Da die formale Definition eines Operationsblockes die Definition einer allgemeinen Operation voraussetzt, die spezielle subblock–Operation (bzw. die subblocks–Operation), die hier definiert wird, aber ihrerseits die Definition eines Operationsblockes benötigt, kann die folgende Definition nur implizit rekursiv gegeben werden.

Definition 6.3 (Menge aller Operationen)
Die Menge Ω aller möglichen Operationen, die die Menge aller Grund– und Kontrollflußoperationen umfasst, sei definiert als die kleinste Menge, die folgenden Forderungen genügt:

- $\Omega_0 \subseteq \Omega$, *d.h. die Menge aller Grundoperationen ist in Ω enthalten.*
- *Für die Menge Ω von möglichen Operationen bezeichnet $\Lambda = \{\beta \mid \beta = (\omega_1, \ldots, \omega_n), n \in \mathbb{N}, \forall 1 \leq i \leq n : \omega_i \in \Omega\}$ die Menge aller möglichen* Operationsblöcke. *Dabei seien Operationen, die in verschiedenen Operationsblöcken oder im gleichen Operationsblock an verschiedenen Positionen vorkommen stets unterscheidbar verschieden, selbst wenn sie vom gleichen Typ sind. (Damit gilt z.B. für den Operationsblock $\beta = (\omega_1, \omega_2)$ mit $\omega_1 = \mathrm{get}_v$ und $\omega_2 = \mathrm{get}_v$, daß $\omega_1 \neq \omega_2$ ist). Im folgenden wird die folgende Abkürzung verwendet:*

$$\omega \in \beta \quad :\Longleftrightarrow \quad \exists n \in \mathbb{N} : \beta = (\omega_1, \ldots, \omega_n) \wedge \exists i \in \{1, \ldots, n\} : \omega = \omega_i$$

- *Für alle Variablen $v \in V$ seien* $\mathrm{var}_v \in \Omega$ *und* $\mathrm{rav}_v \in \Omega$.
- *Für einen Operationsblock $\beta \in \Lambda$ sei* $\mathrm{subblock}_\beta \in \Omega$.
- *Für zwei Operationsblöcke $\beta_1, \beta_2 \in \Lambda$ sei* $\mathrm{subblocks}_{\beta_1,\beta_2} \in \Omega$.
- *Für Operationsblöcke $\beta_1, \beta_2 \in \Lambda$ und eine Variable $v \in V$ sei* $\mathrm{split}^{\mathtt{if}}_{v,\beta_1,\beta_2} \in \Omega$. *Die Operation werde als Split–Operation für die Variable v einer If–Anweisung bezeichnet.*
- *Für Operationsblöcke $\beta_1, \beta_2 \in \Lambda$ und eine Variable $v \in V$ sei* $\mathrm{join}^{\mathtt{if}}_{v,\beta_1,\beta_2} \in \Omega$. *Die Operation werde als Join–Operation für die Variable v einer If–Anweisung bezeichnet.*
- *Für einen Operationsblock $\beta \in \Lambda$ und eine Variable $v \in V$ sei* $\mathrm{split}^{\mathtt{while}}_{v,\beta} \in \Omega$. *Die Operation werde als Split–Operation für die Variable v einer While–Anweisung bezeichnet.*
- *Für einen Operationsblock $\beta \in \Lambda$ und eine Variable $v \in V$ sei* $\mathrm{join}^{\mathtt{while}}_{v,\beta} \in \Omega$. *Die Operation werde als Join–Operation für die Variable v einer While–Anweisung bezeichnet.*

In dieser Definition bilden die Operationsblöcke die Gegenstücke in der neuen Programmrepräsentation zu Programmblöcken in der Spezifikation der Eingabeprogramme. Die Operationen subblock und subblocks ermöglichen die Integration von anderen untergeordneten Operationsblöcken in die Darstellung. Die If– und While–Anweisungen werden nicht direkt in diese Darstellungsform übersetzt werden, sondern deren enthaltene Programmblöcke in Operationsblöcke transformiert und mittels der subblock(s)–Operation in die Darstellung integriert. Die eigentliche Funktion der If– und While–Anweisungen, mögliche Kontrollflußpfade zu definieren, wird dabei von den zugehörigen split– und join–Operationen realisiert. Durch das "Wissen" der split– und join–Operationen, in welcher Reihenfolge die integrierten Subblöcke ausgeführt werden können, stellen sie auch eine Beschreibung der Kontrollflußpfade der ursprünglichen If– und While–Anweisungen dar. Da die Menge der Variablen, die in Kontrollflußanweisungen untergeordneten Programmblöcken vorkommen, nicht immer gleich der Menge aller außerhalb sichtbaren Variablen ist, werden separate Split– und Join–Operationen für jede dieser Variablen erzeugt, um nach außen zu signalisieren, welche Variablen in diesen Anweisungen vorkommen. Damit kann der spätere Query–Fluss auf diejenigen Programmbereiche beschränkt werden, in denen eine bestimmte Variable eine Veränderung erfahren kann. Die Split– und

Join–Operationen stellen für jede Variable in einem der untergeordneten Programm– bzw. Operationsblöcke eine Verbindung zwischen den Verwendungen der Variablen innerhalb und außerhalb der (Darstellungen der) If– bzw. While–Anweisung her.

Die Operationen var und rav besitzen keine eigentliche Funktionalität, sondern werden zur Vereinfachung der Beschreibung von gegenseitigen Beziehungen von Operationen in verschiedenen Operationsblöcken verwendet. Dazu werden diese per Konstruktion die erste bzw. letzte Verwendung einer Variablen in einem Operationsblock darstellen. Damit besitzt jede Operation in einem Operationsblock außer den var– und rav–Operationen einen Vorgänger und einen Nachfolger im gleichen Operationsblock, so daß hier Fallunterscheidungen vermieden werden können.

Definition 6.4 (Sequenzen von (Grund–)Operationen)
Die Mengen

$$\Omega^* := \{(\omega_1, \ldots, \omega_n) \mid n \in \mathbb{N} \wedge \forall 1 \leq i \leq n \ : \ \omega_i \in \Omega\}$$

und

$$\Omega_0^* := \{(\omega_1, \ldots, \omega_n) \mid n \in \mathbb{N} \wedge \forall 1 \leq i \leq n \ : \ \omega_i \in \Omega_0\}$$

bezeichnen die Menge aller endlichen Sequenzen von (Grund–)Operationen.

6.3.2 Semantik

Zur Beschreibung der Ausführung einer Operation wird in dieser Arbeit eine neue Form von Semantikfunktion eingeführt, die als *erweiterte Semantik* bezeichnet wird. Diese wird im nachfolgenden Abschnitt zunächst informell vorgestellt und anschließend formal definiert werden.

6.3.2.1 Grundlegende Definitionen

Um eine Aufteilung von Zuweisungen in mehrere Grundoperationen zu ermöglichen, muß zwischen diesen Grundoperationen eine Möglichkeit geschaffen werden, Informationen zu transferieren, die bei der Abarbeitung der ursprünglichen Zuweisung nur zuweisungsintern entstehen und verwendet werden. Dies spiegelt sich in einer erweiterten Form einer Semantikfunktion wieder, die im nächsten Abschnitt vorgestellt werden wird, und die als Effekt der Ausführung einer Grundoperation zusätzlich zu einer Auswirkung auf die Variablenbelegung eine Menge von weiteren Informationen zur Verfügung stellt oder verwendet. Diese Informationen haben die folgende Form, wobei wie im letzten Abschnitt mit V die Menge aller Variablen und mit D die Wertemenge dieser Variablen bezeichnet werden soll.

Definition 6.5 (Benannter Wert)
Ein Tupel (ω, p, s) wie nachfolgend spezifiziert werde als benannter Wert *bezeichnet. Dabei sei $\omega \in \Omega_0$ eine Grundoperation, die diesen Wert produziert, p eine Zeichenkette, die einem Namen des benannten Wertes entspricht, und $s \in D$ ein Wert aus D, im Folgenden auch als Inhalt des benannten Wertes bezeichnet. Die Menge aller möglichen benannten Werte werde mit $\mathcal{R}$ bezeichnet.*

Die während der Ausführung einer Zuweisung zur Verfügung stehenden Informationen in Form von konkreten Werten, z.B. aus dem Inhalt von Variablen resultierend, werden, sofern sie nach der Transformation in die neue Programmrepräsentation von verschiedenen Grundoperationen erzeugt und verwendet werden, als solche benannte Werte zwischen diesen Grundoperationen transferiert.

Definition 6.6 (Variablenselektionsfunktion)
Eine partielle Abbildung $\Psi : \Omega \to V$ werde als Variablenselektionsfunktion *bezeichnet. Die Menge aller Variablenselektionsfunktionen werde mit $\mathcal{S}$ bezeichnet.*

Um zwischen zwei Grundoperationen das Ergebnis der Dereferenzierung einer Zeigervariablen transferieren zu können, wird eine Variablenselektionsfunktion als Teil der erweiterten Semantikfunktion eingeführt. Eine Grundoperation $\omega \in \Omega_0$, die die Dereferenzierung einer Zeigervariable durchführt und als Ergebnis die Variable $v \in V$ erhält, modifiziert diese Funktion bei ihrer Ausführung. Für die

dabei entstehende Variablenselektionsfunktion Ψ' gilt, daß $\Psi'(\omega) = v$ ist. Eine andere Grundoperation, die dieses Ergebnis der Operation ω verwendet, kann danach anstatt auf eine feste Variable $v' \in V$ auf die sich aus $\Psi'(\omega)$ ergebende Variable zugreifen, in diesem Fall also auf v. Damit wird es ermöglicht, daß die Dereferenzierung von einer Grundoperation durchgeführt wird, und die Verwendung der dereferenzierten Variablen durch eine andere Grundoperation beschrieben wird. Die Variablenselektionsfunktion dient dabei zur Informationsübertragung zwischen diesen Operationen, indem die dereferenzierende Operation in der Variablenselektionsfunktion das Ergebnis der Dereferenzierung codiert, und die das Ergebnis verwendende Operation auf diese Information zugreift.

6.3.2.2 Einführung einer erweiterten Semantikdefinition

Für die oben vorgestellten Grundoperationen können nun die entsprechenden erweiterten Semantikfunktionen definiert werden.

Definition 6.7 (Erweiterte Semantikfunktion für Grundoperationen)
Die nachfolgend definierte Funktion

$$[\![\ ]\!] : \Omega_0 \to (\Sigma \times \mathcal{P}(\mathcal{R}) \times \mathcal{S} \to \Sigma \times \mathcal{P}(\mathcal{R}) \times \mathcal{S})$$

werde als erweiterte Semantikfunktion *bezeichnet, wobei wie bisher mit $\mathcal{P}(\ldots)$ die Potenzmenge einer Menge bezeichnet sein soll und Σ wie in Definition 4.13, $\mathcal{R}$ wie in Definition 6.5, sowie $\mathcal{S}$ wie in Definition 6.6 eingeführt sein sollen. Für Parameter $\sigma \in \Sigma$, $R \in \mathcal{P}(\mathcal{R})$, $\Psi \in \mathcal{S}$ und $\omega \in \Omega_0$, sowie $p \in P$ mit $p =$ outValue sei $[\![\ ]\!]$ dann wie folgt definiert:*

$$
\begin{array}{lcrcl}
\omega = \mathrm{get}_v & : & [\![\ \mathrm{get}_v\ ]\!](\sigma, R, \Psi) & := & (\sigma, R \cup \{(\omega, \mathrm{outValue}, \sigma(v))\}, \Psi) \\
\omega = \mathrm{get}_{\omega'} & : & [\![\ \mathrm{get}_{\omega'}\ ]\!](\sigma, R, \Psi) & := & (\sigma, R \cup \{(\omega, \mathrm{outValue}, \sigma(\Psi(\omega'\,)))\}, \Psi[\omega' \to \mathrm{undef}]) \\
\omega = \mathrm{set}_{v,\omega',p} & : & [\![\ \mathrm{set}_{v,\omega',p}\ ]\!](\sigma, R, \Psi) & := & (\sigma[v \to s], R \setminus (\omega', p, s), \Psi), \\
 & & & & \textit{falls } (\omega', p, s) \in R \textit{ (ansonsten undefiniert)} \\
\omega = \mathrm{set}_{\omega'',\omega',p} & : & [\![\ \mathrm{set}_{\omega'',\omega',p}\ ]\!](\sigma, R, \Psi) & := & (\sigma[\Psi(\omega'') \to s], R \setminus (\omega', p, s), \Psi[\omega'' \to \mathrm{undef}]), \\
 & & & & \textit{falls } (\omega', p, s) \in R \textit{ (ansonsten undefiniert)} \\
\omega = \mathrm{addr}_v & : & [\![\ \mathrm{addr}_v\ ]\!](\sigma, R, \Psi) & := & (\sigma, R \cup \{(\omega, \mathrm{outValue}, v)\}, \Psi) \\
\omega = \mathrm{deref}_{\omega',p} & : & [\![\ \mathrm{deref}_{\omega',p}\ ]\!](\sigma, R, \Psi) & := & (\sigma, R \setminus (\omega', p, v), \Psi[\omega \to v]), \\
 & & & & \textit{falls } (\omega', p, v) \in R \textit{ (ansonsten undefiniert)}
\end{array}
$$

Die Anwendung $[\![\ ]\!](\omega)$ der erweiterten Semantikfunktion auf eine Operation $\omega \in \Omega_0$ werde im Folgenden durch $[\![\ \omega\ ]\!]$ ausgedrückt.

Anschaulich entspricht dies bei der Get–Operation dem Hinzufügen eines neuen benannten Wertes mit Ursprungsknoten ω, also der betrachteten Grundoperation, dem Namen outValue und der Belegung der Variablen v als Inhalt zur Menge R der benannten Werte. Die zweite Variante der Get–Operation enthält im produzierten benannten Wert den Inhalt derjenigen Variable, die sich aus der Anwendung der Variablenselektionsfunktion $\Psi(\omega')$ ergibt. Dabei wird die partielle Abbildung Ψ so verändert, daß $\Psi(\omega')$ nach Ausführung der Operation wieder undefiniert ist. Die Set–Operation modifiziert die Belegung der Variable v auf den Inhalt s eines benannten Wertes (ω', p, s), der in der Menge R enthalten sein muß, und der bei der Ausführung der Set–Operation konsumiert wird. Analog modifiziert die zweite Variante der Set–Operation diejenige Variable, die sich aus der Anwendung der Variablenselektionsfunktion $\Psi(\omega')$ ergibt, auf die gleiche Weise. Die Adress–Operation generiert einen benannten Wert mit Wert $v \in D$. Die Dereferenzierungsoperation schließlich ist die einzige Grundoperation, die die Variablenselektionsfunktion Ψ durch Erweiterung ihres Definitionsbereiches verändert. Dabei wird von der Dereferenzierungsoperation ω ein benannter Wert konsumiert und dessen Inhalt, der eine Variable $v \in D$ sein muß, als Ergebnis der Variablenselektionsfunktion Ψ, angewandt auf ω definiert.

Definition 6.8 (Erweiterte Semantik einer Sequenz von Grundoperationen)
Die erweiterte Semantik $[\![\ ]\!]^*$ einer Sequenz von Grundoperationen $(\omega_1, \ldots, \omega_n) \in \Omega_0^*$ mit $n \in \mathbb{N}$

und $\forall 1 \leq i \leq n : \omega_i \in \Omega_0$ *werde wie folgt definiert, wobei* $\sigma \in \Sigma$, $R \in \mathcal{P}(\mathcal{R})$ *und* $\Psi \in \mathcal{S}$ *sein sollen*

$$[\![(\omega_1, \ldots, \omega_n)]\!]^*(\sigma, R, \Psi) := \begin{cases} [\![\omega_1]\!](\sigma, R, \Psi), & \textit{falls } n{=}1 \\ [\![(\omega_2, \ldots, \omega_n)]\!]^*([\![\omega_1]\!](\sigma, R, \Psi)), & \textit{sonst} \end{cases}$$

Wie in der Literatur üblich wird im Folgenden nicht mehr explizit zwischen $[\![\;]\!]$ *und* $[\![\;]\!]^*$ *unterschieden.*

Definition 6.9 (Produzierte und konsumierte benannte Werte)
Für eine Grundoperation $\omega \in \Omega_0$, *für deren erweiterte Semantikfunktion* $[\![\omega]\!](\sigma, R, \Psi) = (\sigma', R', \Psi')$ *gilt, seien die folgenden Abkürzungen definiert:*

$$\begin{aligned} \text{cons}_\omega &:= R \setminus R' \\ \text{prod}_\omega &:= R' \setminus R \end{aligned}$$

Damit entspricht cons_ω der Menge der benannten Werte, die vor Ausführung der Grundoperation ω in R enthalten sind und nach deren Ausführung nicht mehr in R' enthalten sind, und damit den benannten Werten, die bei der Ausführung von ω *konsumiert* werden. Analog entspricht prod_ω den von ω *produzierten* benannten Werten. Gilt z.B. $R = \{(\omega_1, p_1, s_1), (\omega_2, p_2, s_2), (\omega_3, p_3, s_3)\}$ und $R' = \{(\omega_3, p_3, s_3), (\omega_4, p_4, s_4)\}$, so ergibt sich $\text{cons}_\omega = \{(\omega_1, p_1, s_1), (\omega_2, p_2, s_2)\}$ und $\text{prod}_\omega = \{(\omega_4, p_4, s_4)\}$.

An Definition 6.7 ist zu erkennen, daß die Mengen cons_ω und prod_ω gemäß Definition 6.9 für jede Operation $\omega \in \Omega_0$ bis auf die konkreten Inhalte der benannten Werte unabhängig von σ, R und Ψ sind. Demnach enthalten diese Mengen stets die gleichen benannten Werte, wenn man nur die ersten beiden Komponenten von benannten Werten aus $\Omega \times P$ betrachtet.

6.3.2.3 Erweiterte Semantikdefinition für Kontrollflußoperationen

Betrachtet man traditionelle Programmgraphen, so erkennt man, daß Kontrollflußanweisungen wie If oder While dort keine Anweisungen im eigentlichen Sinn sind, und entsprechend auch keine Semantik besitzen. In diesen Graphen werden nur Anweisungen, die keine Kontrollflußanweisungen sind (in Abschnitt 3.1 als Zuweisung bezeichnet) explizit dargestellt. Diese Zuweisungen werden jeweils durch einen Knoten im Programmgraphen repräsentiert und besitzen eine Semantikfunktion. Der mögliche Kontrollfluß wird implizit durch die Kanten zwischen diesen Knoten dargestellt. Damit wird es möglich, die Auswirkung einer Menge von Zuweisungen entlang eines möglichen Programmpfades zu beschreiben, indem ein (intraprozeduraler) Programmpfad durch eine Menge von Knoten beschrieben wird, die zusammenhängend durch Kanten verbunden sind. Das die Verzweigung verursachende Programmkonstrukt selbst hat aber keine direkte Repräsentation und es wird auch keine Aussage darüber gemacht, was für eine Auswirkung die Ausführung z.B. einer If–Anweisung hat. In einer konkreten Programminstanz kann stets eine der beiden Alternativen ausgeführt werden. Dies bedeutet, daß für eine Programminstanz, die bei der Ausführung diese If–Anweisung erreicht, zwei gültige Fortsetzungen existieren.

Das gleiche Prinzip verfolgen auch die Kontrollflußoperationen aus dieser Arbeit. Diese Operationen besitzen ebenfalls keine konkrete Semantik, sondern dienen nur der Beschreibung der möglichen Pfade durch ein Eingabeprogramm.

6.4 Transformation in die neue textuelle Programmrepräsentation

6.4.1 Transformationsvorschrift

6.4.1.1 Grundoperationen

Die Transformation von Eingabeprogrammen in die textuelle Darstellungsform der neuen Programmrepräsentation wird durch eine im Folgenden schrittweise definierte Transformationsabbildung $T : A \to \Omega^*$ realisiert, die Anweisungen $a \in A$ in Sequenzen von Operationen überführt. Für diese Sequenzen von Operationen gelte die folgende Notationsvereinbarung.

Definition 6.10 (Schreibweise)
Für eine Sequenz von Operationen $(\omega_1, \ldots, \omega_n) \in \Omega^*$ *mit* $n \in \mathbb{N}$ *und* $\forall 1 \leq i \leq n \ : \ \omega_i \in \Omega$ *sei die folgende zeilenweise Schreibweise eine äquivalente Beschreibungsform:*

$$\begin{array}{rl} & \omega_1 \\ & \ldots \\ \omega: & \omega_l \\ & \ldots \\ & \omega_n \end{array}$$

Die optionale Schreibweise $\omega : \omega_l$ *entspricht dabei einer Definition der Operation* ω *als Operation* ω_l*.*

Die folgende Definition beschreibt einen Teil des Transformationsschrittes.

Definition 6.11 (Transformationsabbildung $\overline{T}$)
Die Abbildung $\overline{T} : (V \times \mathbb{N}_0) \to \Omega^*$*, die eine Variable und eine natürliche Zahl auf eine Sequenz von Operationen abbildet, sei wie folgt definiert.*

$$\begin{array}{rcl} \overline{T}(v,0) & := & \mathrm{get}_v, \\ \overline{T}(v,n) & := & \begin{array}{rl} & \omega_1 \\ & \ldots \\ & \omega_m \\ \omega: & \mathrm{deref}_{\omega_m,\mathrm{outValue}} \\ & \mathrm{get}_\omega \end{array} \end{array}$$

wobei gelten soll, daß für ein $m \in \mathbb{N}$

$$(\omega_1, \ldots, \omega_m) := \overline{T}(v, n-1)$$

Mit dieser Definition kann man nun die Transformationsvorschrift T_z für Zuweisungen wie folgt definieren.

Definition 6.12 (Transformation von Zuweisungen)
Für eine Anweisung $a \in \mathcal{A}_z$ *mit* $\mathcal{A}_z$ *wie in Definition 4.5 sei die Abbildung* $T_z : \mathcal{A}_z \to \Omega^*$ *wie folgt definiert:*

$$T_z(\underbrace{*\ldots*}_{n-mal} v = \underbrace{*\ldots*}_{m-mal} w) \quad := \quad \left\{ \begin{array}{rll} & \omega_1 & \textit{falls } n = 0 \textit{ und} \\ & \ldots & \overline{T}(w,m) = (\omega_1, \ldots, \omega_l) \\ & \omega_l & \textit{für ein } l \in \mathbb{N} \\ & \mathrm{set}_{v,\omega_l,\mathrm{outValue}}, & \\ & & \\ & \omega_1 & \textit{falls } n > 0 \textit{ und} \\ & \ldots & \overline{T}(w,m) = (\omega_1, \ldots, \omega_l) \\ & \omega_l & \textit{für ein } l \in \mathbb{N} \\ & \omega_1' & \textit{und } \overline{T}(v, n-1) = (\omega_1', \ldots, \omega_k') \\ & \ldots & \textit{für ein } k \in \mathbb{N}, \\ & \omega_k' & \\ \omega: & \mathrm{deref}_{\omega_k',\mathrm{outValue}} & \\ & \mathrm{set}_{\omega,\omega_l,\mathrm{outValue}}, & \end{array} \right.$$

$$T_z(\underbrace{*\ldots*}_{n-mal} v = \&w) \quad := \quad \begin{cases} \omega: & \mathrm{addr}_w & \textit{falls } n = 0, \\ & \mathrm{set}_{v,\omega,\mathrm{outValue}}, & \\ \omega': & \mathrm{addr}_w & \textit{falls } n > 0 \textit{ und} \\ & \omega_1 & \overline{T}(v, n-1) = (\omega_1, \ldots, \omega_l) \\ & \ldots & \textit{für ein } l \in \mathbb{N} \\ & \omega_l & \\ \omega'': & \mathrm{deref}_{\omega_l,\mathrm{outValue}} & \\ & \mathrm{set}_{\omega'',\omega',\mathrm{outValue}} & \end{cases}$$

Satz 6.1 (Zusammenhang von $\overline{T}$ und dereference)
Für alle $n \in \mathbb{N}_0$ und alle $\sigma \in \Sigma$, $R \in \mathcal{P}(\mathcal{R})$ und $\Psi \in \mathcal{S}$ gilt:

$$\begin{aligned} &\overline{T}(v,n) = (\omega_1, \ldots, \omega_m) \in \Omega_0^*, m \in \mathbb{N}, \textit{ und } [\![\, \overline{T}(v,n) \,]\!](\sigma, R, \Psi) = (\sigma', R', \Psi') \\ &\quad \Longrightarrow \quad (\omega_m, \mathrm{outValue}, \sigma(\mathrm{dereference}(v,n,\sigma))) \in R' \end{aligned}$$

Beweis: Induktion über n.

$n = 0$**:** Nach Definition 6.11 gilt $\overline{T}(v,0) = \omega_1 = \mathrm{get}_v$. Seien $\sigma \in \Sigma$, $R \in \mathcal{P}(\mathcal{R})$ und $\Psi \in \mathcal{S}$ beliebig. Dann gilt nach Definition 6.7:

$$[\![\, \mathrm{get}_v \,]\!](\sigma, R, \Psi) = (\sigma, \underbrace{R \cup \{(\omega_1, \mathrm{outValue}, \sigma(v))\}}_{=:R'}, \Psi).$$

Andererseits gilt nach Definition 4.16

$$\mathrm{dereference}(v, 0, \sigma) = v$$

und damit die Behauptung

$$(\omega_1, \mathrm{outValue}, \sigma(\mathrm{dereference}(v,0,\sigma))) \in R'.$$

$n \to n+1$**:** Gelte die Behauptung für n, d.h.

$$\begin{aligned} &\overline{T}(v,n) = (\omega_1, \ldots, \omega_m) \in \Omega_0^*, m \in \mathbb{N}, \text{ und } [\![\, \overline{T}(v,n) \,]\!](\sigma, R, \Psi) = (\sigma', R', \Psi') \\ &\quad \Longrightarrow \quad (\omega_m, \mathrm{outValue}, \sigma(\mathrm{dereference}(v,n,\sigma))) \in R' \end{aligned}$$

Zu zeigen ist:

$$\begin{aligned} &\overline{T}(v,n+1) = (\omega_1, \ldots, \omega_{m'}) \in \Omega_0^*, m' \in \mathbb{N}, \text{ und } [\![\, \overline{T}(v,n+1) \,]\!](\sigma, R, \Psi) = (\sigma'', R'', \Psi'') \\ &\quad \Longrightarrow \quad (\omega_{m'}, \mathrm{outValue}, \sigma(\mathrm{dereference}(v,n+1,\sigma))) \in R'' \end{aligned}$$

Nach Definition 6.11 gilt mit $\overline{T}(v,n) = (\omega_1, \ldots, \omega_m)$ für ein $m \in \mathbb{N}$:

$$\overline{T}(v,n+1) \quad = \quad \begin{array}{ll} & \omega_1 \\ & \ldots \\ & \omega_m \\ \omega: & \mathrm{deref}_{\omega_m,\mathrm{outValue}} \\ \omega': & \mathrm{get}_\omega \end{array}$$

Nach Induktionsannahme gilt mit $(\omega_1, \ldots, \omega_m) = \overline{T}(v,n)$ gemäß Definition 6.11 und für beliebige $\sigma \in \Sigma$, $R \in \mathcal{P}(\mathcal{R})$, $\Psi \in \mathcal{S}$ mit $(\sigma', R', \Psi') := [\![\, \overline{T}(v,n) \,]\!]$ gemäß Definitionen 6.7 und 6.8:

$$(\omega_m, \mathrm{outValue}, \sigma(\mathrm{dereference}(v,n,\sigma))) \in R'$$

Nach Definition 6.8 ergibt sich

$$
\begin{aligned}
&[\![\, \overline{T}(v, n+1) \,]\!](\sigma, R, \Psi) \\
&= \quad [\![\, \mathrm{get}_{\omega} \,]\!]([\![\, \mathrm{deref}_{\omega_m, \mathrm{outValue}} \,]\!]([\![\, \overline{T}(v,n) \,]\!](\sigma, R, \Psi))) \\
&= \quad [\![\, \mathrm{get}_{\omega} \,]\!]([\![\, \mathrm{deref}_{\omega_m, \mathrm{outValue}} \,]\!](\sigma', R', \Psi')) \\
&\overset{\text{Def.6.7}}{=} \quad [\![\, \mathrm{get}_{\omega} \,]\!](\sigma', \underbrace{R' \setminus \{(\omega_m, \mathrm{outValue}, \sigma(\mathrm{dereference}(v,n,\sigma)))\}}_{=:R'''}, \\
&\qquad\qquad \underbrace{\Psi'[\omega \to \sigma(\mathrm{dereference}(v,n,\sigma))]}_{=:\Psi'''}]) \\
&= \quad (\sigma', R''' \cup \{(\omega', \mathrm{outValue}, \sigma(\Psi'''(\omega)))\}, \Psi'''[\omega \to \mathrm{undef}]) \\
&= \quad (\sigma', R''' \cup \{(\omega', \mathrm{outValue}, \sigma(\underbrace{\sigma(\mathrm{dereference}(v,n,\sigma))}_{\overset{\text{Def.4.16}}{=}\sigma(\mathrm{dereference}(v,n+1,\sigma))})\}, \Psi'''[\omega \to \mathrm{undef}]) \\
&= \quad (\underbrace{\sigma'}_{=:\sigma''}, \underbrace{R''' \cup \{(\omega', \mathrm{outValue}, \sigma(\mathrm{dereference}(v,n+1,\sigma))\}}_{=:R''}, \underbrace{\Psi'''[\omega \to \mathrm{undef}]}_{=:\Psi''})
\end{aligned}
$$

Damit gilt also $(\omega', \mathrm{outValue}, \sigma(\mathrm{dereference}(v, n+1, \sigma)) \in R''$ und damit die Behauptung mit $m' = m + 2$ und $\omega_{m'} = \omega'$.

□

Satz 6.2 (Äquivalenz der Programmdarstellungen)
Sei $\sigma \in \Sigma$, $R \in \mathcal{P}(\mathcal{R})$ und $\Psi \in \mathcal{S}$ beliebig. Dann gilt:

1. *Für eine Zuweisung $a \equiv \underbrace{*\ldots*}_{n-mal} v = \underbrace{*\ldots*}_{m-mal} w \in \mathcal{A}_z$:*

$$[\![\, a \,]\!](\sigma) = \sigma'' \wedge [\![\, T_z(a) \,]\!](\sigma, R, \Psi) = (\sigma', R', \Psi') \Longrightarrow \sigma' = \sigma''$$

2. *Für eine Zuweisung $a \equiv \underbrace{*\ldots*}_{n-mal} v = \&w \in \mathcal{A}_z$:*

$$[\![\, a \,]\!](\sigma) = \sigma'' \wedge [\![\, T_z(a) \,]\!](\sigma, R, \Psi) = (\sigma', R', \Psi') \Longrightarrow \sigma' = \sigma''$$

Beweis: Sei $\sigma \in \Sigma$, $R \in \mathcal{P}(\mathcal{R})$ und $\Psi \in \mathcal{S}$ beliebig.

Behauptung 1: Sei $a \equiv \underbrace{*\ldots*}_{n-\mathrm{mal}} v = \underbrace{*\ldots*}_{m-\mathrm{mal}} w \in \mathcal{A}_z$. Nach Definition 4.17 gilt:

$$[\![\, a \,]\!](\sigma) \quad = \quad \sigma[\mathrm{dereference}(v,n,\sigma) \to \sigma(\mathrm{dereference}(w,m,\sigma))] =: \sigma''$$

Fall 1: $n = 0$: Nach Definition 6.12 gilt mit $\overline{T}(w,m) = (\omega_1, \ldots, \omega_l)$ für ein $l \in \mathbb{N}$:

$$
\begin{aligned}
T_z(v = \underbrace{*\ldots*}_{m-\mathrm{mal}} w) \quad = \quad & \omega_1 \\
& \ldots \\
& \omega_l \\
& \mathrm{set}_{v,\omega_l,\mathrm{outValue}}
\end{aligned}
$$

Die Operationen, die in der Sequenz $\overline{T}(w,m)$ vorkommen, verändern gemäß Definitionen 6.7 und 6.11 jeweils die Variablenbelegung σ nicht. Damit ist die Operation $\mathrm{set}_{v,\omega_l,\mathrm{outValue}}$ die einzige Operation, die eine Veränderung der Variablenbelegung σ bewirken kann. Nach

Satz 6.1 gilt damit für das Tupel $(\sigma''', R''', \Psi''') := [\![\ \overline{T}(w,m)\ ]\!](\sigma, R, \Psi)$, daß $\sigma = \sigma'''$ und

$$(\omega_l, \text{outValue}, \sigma(\text{dereference}(w,m,\sigma))) \in R'''$$

ist. Damit ergibt sich nach Definition 6.7:

$$\begin{aligned}
&[\![\ \text{set}_{v,\omega_l,\text{outValue}}\ ]\!](\sigma, R''', \Psi''') \\
&= \quad (\sigma[v \to \sigma(\text{dereference}(w,m,\sigma))], \ldots, \ldots) \\
&\overset{\text{Def.4.16}}{=} \quad (\underbrace{\sigma[\text{dereference}(v,0,\sigma) \to \sigma(\text{dereference}(w,m,\sigma))]}_{=:\sigma'}, \ldots, \ldots)
\end{aligned}$$

und damit gilt die Behauptung.

Fall 2: $n > 0$: Nach Definition 6.12 gilt mit $\overline{T}(w,m) = (\omega_1, \ldots, \omega_l)$ für ein $l \in \mathbb{N}$ und $\overline{T}(v, n-1) = (\omega_1', \ldots, \omega_k')$ für ein $k \in \mathbb{N}$

$$T_z(\underbrace{*\ldots*}_{n-\text{mal}} v = \underbrace{*\ldots*}_{m-\text{mal}} w) \quad = \quad \begin{array}{r} \omega_1 \\ \ldots \\ \omega_l \\ \omega_1' \\ \ldots \\ \omega_k' \\ \omega: \quad \text{deref}_{\omega_k',\text{outValue}} \\ \text{set}_{\omega,\omega_l,\text{outValue}} \end{array}$$

Nach Satz 6.1 gilt für $(\sigma_1, R_1, \Psi_1) := [\![\ (\omega_1, \ldots, \omega_l)\ ]\!](\sigma, R, \Psi)$, daß

$$(\omega_l, \text{outValue}, \sigma(\text{dereference}(w,m,\sigma))) \in R_1$$

ist. Weiter gilt nach dem gleichen Satz, daß für $(\sigma_2, R_2, \Psi_2) := [\![\ (\omega_1', \ldots, \omega_k')\ ]\!](\sigma_1, R_1, \Psi_1)$ $(\omega_k', \text{outValue}, \sigma(\text{dereference}(v, n-1, \sigma))) \in R_2$ ist. Da die Sequenzen von Operationen $(\omega_1, \ldots, \omega_l)$ und $(\omega_1', \ldots, \omega_k')$ keine Veränderung der Variablenbelegung bewirken, und die Operationen $(\omega_1', \ldots, \omega_k')$ den von ω_l produzierten benannten Wert nicht konsumieren, gilt damit

$$[\![\ (\omega_1, \ldots, \omega_l, \omega_1', \ldots, \omega_k')\ ]\!](\sigma, R, \Psi) = (\sigma, R_2, \Psi_2)$$

mit

$$(\omega_l, \text{outValue}, \sigma(\text{dereference}(w,m,\sigma))) \in R_2$$

und

$$(\omega_k', \text{outValue}, \sigma(\text{dereference}(v, n-1, \sigma))) \in R_2$$

Damit gilt dann

$$\begin{aligned}
&[\![\ T_z(\underbrace{*\ldots*}_{n-\text{mal}} v = \underbrace{*\ldots*}_{m-\text{mal}} w)\ ]\!] \\
&\overset{\text{Def.6.8}}{=} \quad [\![\ \text{set}_{\omega,\omega_l,\text{outValue}}\ ]\!]([\![\ \text{deref}_{\omega_k',\text{outValue}}\ ]\!](\sigma, R_2, \Psi_2)) \\
&\overset{\text{Def.6.7}}{=} \quad [\![\ \text{set}_{\omega,\omega_l,\text{outValue}}\ ]\!](\sigma, \underbrace{R_2 \setminus \{(\omega_k', \text{outValue}, \sigma(\text{dereference}(v, n-1, \sigma)))\}}_{=:R_3}, \\
&\qquad\qquad \underbrace{\Psi_2[\omega \to \sigma(\text{dereference}(v, n-1, \sigma))]}_{=:\Psi_3}) \\
&\overset{\text{Def.6.7}}{=} \quad (\sigma[\Psi_3(\omega) \to \sigma(\text{dereference}(w,m,\sigma))], \ldots, \ldots) \\
&= \quad (\sigma[\underbrace{\sigma(\text{dereference}(v, n-1, \sigma))}_{=\text{dereference}(v,n,\sigma)} \to \sigma(\text{dereference}(w,m,\sigma))], \ldots, \ldots) \\
&= \quad (\underbrace{\sigma[\text{dereference}(v,n,\sigma) \to \sigma(\text{dereference}(w,m,\sigma))]}_{=:\sigma'}, \ldots, \ldots)
\end{aligned}$$

Damit gilt die Behauptung, da $\sigma = \sigma^{'}$ ist.

Behauptung 2: Sei $a \equiv \underbrace{*\ldots*}_{n-\text{mal}} v = \&w \in \mathcal{A}_z$ und sei $\sigma \in \Sigma$ beliebig. Nach Definition 4.17 gilt:

$$[\![\ a\ ]\!](\sigma) \quad = \quad \sigma[\text{dereference}(v, n, \sigma) \rightarrow w] =: \sigma^{''}$$

Fall 1: $n = 0$: Nach Definition 6.12 gilt damit:

$$T_z(\underbrace{*\ldots*}_{n-\text{mal}} v = \&w) \quad = \quad \omega : \begin{array}{c} \text{addr}_w \\ \text{set}_{v,\omega,\text{outValue}} \end{array}$$

Für beliebige $R \in \mathcal{P}(\mathcal{R})$ und $\Psi \in \mathcal{S}$ gilt dann:

$$\begin{array}{ll} & [\!|\ (\text{addr}_w, \text{set}_{v,\omega,\text{outValue}})\ |\!](\sigma, R, \Psi) \\ \overset{\text{Def.6.8}}{=} & [\!|\ \text{set}_{v,\omega,\text{outValue}}\ |\!]([\!|\ \text{addr}_w\ |\!](\sigma, R, \Psi)) \\ \overset{\text{Def.6.7}}{=} & [\!|\ \text{set}_{v,\omega,\text{outValue}}\ |\!](\sigma, R \cup \{(\omega, \text{outValue}, w)\}, \Psi) \\ = & (\sigma[v \rightarrow w], \ldots, \ldots) \\ = & (\underbrace{\sigma[\text{dereference}(v, 0, \sigma) \rightarrow w]}_{=:\sigma^{'}}, \ldots, \ldots) \end{array}$$

und damit wegen $\sigma = \sigma^{'}$ die Behauptung.

Fall 2: $n > 0$: Beweis analog zum Fall $n = 0$ und Behauptung 1.

□

Satz 6.2 beweist die Äquivalenz der Darstellung eines Eingabeprogrammes gemäß Kapitel 4 mit der zugehörigen traditionellen Semantik mit einer Darstellung durch die neue Programmrepräsentation in dieser Arbeit. Dies geschieht in dem Sinne, daß die Ausführung einer Zuweisung, ausgedrückt durch eine Semantikfunktion, auf eine Variablenbelegung die gleiche Auswirkung hat, wie die Ausführung derjenigen Operationen, durch die erweiterte Semantikfunktion definiert, in die die Zuweisung durch die Transformation T_z überführt wird.

6.4.1.2 Kontrollflußoperationen

Die Kontrollflußanweisungen `if` und `while` werden durch nachfolgend definierte Transformationsvorschrift T_k ebenfalls in Sequenzen von Operationen umgesetzt. Die Transformation T_b von Programmblöcken erfolgt durch die Transformation der in diesen Programmblöcken enthaltenen Anweisungen gemäß den Transformationen T_z bzw. T_k.

Definition 6.13 (Transformation von Programmblöcken und Kontrollflußanweisungen)

Die Transformationsfunktionen

$$T_k : (\mathcal{A} \setminus \mathcal{A}_z) \rightarrow \Omega^*$$

und

$$T_b : \mathcal{B} \rightarrow \Omega^*$$

werden wie nachfolgend angegeben definiert. Aufbauend auf diesen Definitionen bezeichne

$$T : \left\{ \begin{array}{lcll} \mathcal{A} & \rightarrow & \Omega^* & \\ a & \mapsto & T_z(a), & \textit{falls } a \in \mathcal{A}_z \\ a & \mapsto & T_k(a), & \textit{falls } a \in \mathcal{A} \setminus \mathcal{A}_z \end{array} \right.$$

die Transformation der Menge aller Anweisungen in Sequenzen von Operationen. Für die Definition von T_k und T_b bezeichnen im Folgenden $b_1, b_2 \in \mathcal{B}$ Programmblöcke und $V_1 = \{v_1, \ldots, v_j\}$ für ein $j \in$

$\mathbb{N}_0$ *bzw.* V_2 *die Menge derjenigen Variablen, die von den Anweisungen in den Programmblöcken* b_1 *bzw.* b_2 *aufgrund ihres Typs (also im Rahmen dieser Arbeit der Referenzierungsstufe) möglicherweise gelesen oder verändert werden. Weiter bezeichne* $V^{'}$ *die Menge derjenigen Variablen, die in den Anweisungen* $a_1, \ldots, a_k$ *des Programmblockes* $b = (a_1, \ldots, a_k) \in \mathcal{B}$ *für ein* $k \in \mathbb{N}_0$*, bzw. den ihnen im Fall von Kontrollflußanweisungen untergeordneten Programmblöcken (auf der Basis der gleichen Betrachtung wie oben beschrieben) möglicherweise verwendet werden. Sei* $V_1 \cup V_2 = \{v_1, \ldots, v_n\}$ *für ein* $j \leq n \in \mathbb{N}_0$ *und* $V^{'} = \{v_1^{'}, \ldots, v_m^{'}\}$ *für ein* $m \in \mathbb{N}_0$*. Seien die Operationsblöcke* $\beta_1, \beta_2 \in \Lambda$ *als das Ergebnis* $\beta_1 = T_b(b_1)$ *bzw.* $\beta_2 = T_b(b_2)$ *der Transformationsabbildung* T_b *für die den* $\texttt{if}_{b_1,b_2}$*– bzw.* $\texttt{while}_{b_1}$*–Anweisungen untergeordneten Programmblöcke* b_1 *bzw.* b_2 *definiert.*

$$
\begin{aligned}
T_k(\texttt{if}_{b_1,b_2}) \quad := \quad & \text{split}^{\texttt{if}}_{v_1,\beta_1,\beta_2} \\
& \text{split}^{\texttt{if}}_{v_2,\beta_1,\beta_2} \\
& \ldots \\
& \text{split}^{\texttt{if}}_{v_n,\beta_1,\beta_2} \\
& \text{subblocks}_{\beta_1,\beta_2} \\
& \text{join}^{\texttt{if}}_{v_1,\beta_1,\beta_2} \\
& \text{join}^{\texttt{if}}_{v_2,\beta_1,\beta_2} \\
& \ldots \\
& \text{join}^{\texttt{if}}_{v_n,\beta_1,\beta_2} \\
T_k(\texttt{while}_{b_1}) \quad := \quad & \text{split}^{\texttt{while}}_{v_1,\beta_1} \\
& \text{split}^{\texttt{while}}_{v_2,\beta_1} \\
& \ldots \\
& \text{split}^{\texttt{while}}_{v_j,\beta_1} \\
& \text{subblock}_{\beta_1} \\
& \text{join}^{\texttt{while}}_{v_1,\beta_1} \\
& \text{join}^{\texttt{while}}_{v_2,\beta_1} \\
& \ldots \\
& \text{join}^{\texttt{while}}_{v_j,\beta_1} \\
T_b(b) = T_b((a_1, \ldots, a_k)) \quad := \quad & \text{var}_{v_1^{'}} \\
& \text{var}_{v_2^{'}} \\
& \ldots \\
& \text{var}_{v_m^{'}} \\
& T(a_1) \\
& T(a_2) \\
& \ldots \\
& T(a_k) \\
& \text{rav}_{v_1^{'}} \\
& \text{rav}_{v_2^{'}} \\
& \ldots \\
& \text{rav}_{v_m^{'}}
\end{aligned}
$$

Die in der Erläuterung zu Definition 6.3 bereits angesprochene Eigenschaft von var– und rav–Operationen, per Konstruktion die erste bzw. letzte Verwendung von Variablen in einem Operationsblock darzustellen, wird von dieser Konstruktion der Transformation T_b offensichtlich erfüllt.

Definition 6.14 (Transformation eines Programmes P)
Für ein Programm $P = (A, B, V, D, b_0)$ *gemäß Definition 4.8 bezeichnet* $T_b(b_0) \in \Omega^*$ *die Darstellung*

Auswirkung (Semantik): σ	Urspr. Zuw.	$\xrightarrow{T}$	Grundop.	Auswirkung (Erweiterte Semantik): (σ, R, Ψ)
$\emptyset$				$(\emptyset, \emptyset, \emptyset)$
$\{x \to y\}$	`x=&y`		$\omega_1 : \text{addr}_y$	$(\emptyset, \{(\omega_1, \text{outValue}, y)\}, \emptyset)$
			$\omega_2 : \text{set}_{x,\omega_1,\text{outValue}}$	$(\{x \to y\}, \emptyset, \emptyset)$
$\{x \to y, y \to A\}$	`y=&A`		$\omega_3 : \text{addr}_A$	$(\{x \to y\}, \{(\omega_3, \text{outValue}, A)\}, \emptyset)$
			$\omega_4 : \text{set}_{y,\omega_3,\text{outValue}}$	$(\{x \to y, y \to A\}, \emptyset, \emptyset)$
$\{x \to y, y \to A,$ $z \to B\}$	`z=&B`		$\omega_5 : \text{addr}_B$	$(\{x \to y, y \to A\},$ $(\omega_5, \text{outValue}, B)\}, \emptyset)$
			$\omega_6 : \text{set}_{z,\omega_5,\text{outValue}}$	$(\{x \to y, y \to A, z \to B\}, \emptyset, \emptyset)$

Tabelle 6.1: Transformation der Adresszuweisungen aus dem Running Example in Grundoperationen.

des Programmes P durch eine Sequenz von Operationen aus Ω. Die Menge von Operationen, die in der Sequenz von Operationen $T_b(b_0)$ oder dieser durch subblock*(s)–Operationen untergeordneten weiteren Sequenzen von Operationen enthalten sind, werden mit Ω_P bezeichnet.*

6.4.2 Transformation des Running Example in eine textuelle Sequenz von Operationen

Diese Definitionen sollen im folgenden Abschnitt auf das Running Example angewandt und dadurch veranschaulicht werden.

6.4.2.1 Grundoperationen

Zur Veranschaulichung der Transformationsabbildung T sollen zunächst die unverzweigten Sequenzen von Zuweisungen aus dem Running Example in Abb. 3.3 in eine Sequenz von Grundoperationen übertragen werden. Anhand der Berechnung der jeweiligen Semantikfunktionen kann auch beispielhaft die Äquivalenz der beiden Darstellungsformen bzgl. der Variablenbelegungen σ gesehen werden. Diese Art der Berechnung liegt Satz 6.2 zugrunde, der die Äquivalenz der Programmdarstellungen allgemein beweist.

Zunächst soll die Menge der Adresszuweisungen vor der Schleife betrachtet werden. In Tabelle 6.1 sind in der zweiten Spalte die ursprünglichen Zuweisungen aufgelistet. In der Spalte links davon ist jeweils in der gleichen Zeile das Ergebnis der Anwendung der traditionellen Semantikfunktion auf diese Zuweisungen angegeben. Dabei werden von Variablenbelegungen nur diejenigen Variablen aufgelistet, deren Belegungen nicht "undef" sind. Eine Variablenbelegung $\sigma = \emptyset$ im Beispiel ist demnach undefiniert für alle Variablen. Analog wird die Variablenselektionsfunktion $\Psi = \emptyset$ interpretiert. In der dritten Spalte sind diejenigen Grundoperationen aufgelistet, die sich als Ergebnis der Transformation T, hier also konkret T_z, ergeben. In der vierten Spalte findet sich, ebenfalls in der jeweils gleichen Zeile, das Ergebnis der Anwendung der erweiterten Semantikfunktion auf diese Grundoperationen. Nach Ausführung der jeweils letzten Zuweisung bzw. Grundoperation haben die traditionelle und die erweiterte Semantikfunktion die gleiche Variablenbelegung als Ergebnis bzw. als erste Komponente des Ergebnisses.

Analog wird die Sequenz von Zuweisungen in der Schleife transformiert. Für die zweimalige Dereferenzierung von `x` in der letzten Zeile ergibt sich eine Darstellung mittels Grundoperationen, wie sie in Tabelle 6.2 dargestellt ist. Dabei wird hier für die Variablenbelegung vor Ausführung der Anweisung angenommen, daß die Schleife nicht durchlaufen wurde, d.h. daß die Variablenbelegung, die am Ende von Tabelle 6.1 berechnet wurde, hier zu Beginn der Ausführung dieser Zuweisung gilt. Da das Ziel dieser Arbeit Zeigeranalyse ist, wurde hier im Beispiel der konkrete Wert der Variablen `A` und damit auch der konkrete Wert der Variablen `D` nicht angegeben, sondern nur durch "..."

Auswirkung (Semantik): σ	Urspr. Zuw.	$\xrightarrow{T}$	Grundop.	Auswirkung (Erweiterte Semantik): (σ, R, Ψ)
$\{x \to y, y \to A,$ $z \to B, A \to \ldots\}$ $\{\ldots, D \to \ldots\}$	`D=**x`			$(\{x \to y, y \to A, z \to B\}, \emptyset, \emptyset)$
			$\omega_1 : \text{get}_x$	$(\ldots, \{\omega_1, \text{outValue}, y)\}, \emptyset)$
			$\omega_2 : \text{deref}_{\omega_1, \text{outValue}}$	$(\ldots, \emptyset, \{\omega_2 \to y\})$
			$\omega_3 : \text{get}_{\omega_2}$	$(\ldots, \{(\omega_3, \text{outValue}, A)\}, \emptyset)$
			$\omega_4 : \text{deref}_{\omega_3, \text{outValue}}$	$(\ldots, \emptyset, \{\omega_4 \to A\})$
			$\omega_5 : \text{get}_{\omega_4}$	$(\ldots, \{(\omega_5, \text{outValue}, \ldots)\}, \emptyset)$
			$\omega_6 : \text{set}_{D, \omega_5, \text{outValue}}$	$(\{\ldots, D \to \ldots\}, \emptyset, \emptyset)\}$

Tabelle 6.2: Transformation der zweimaligen Dereferenzierung von `x` aus dem Running Example in Grundoperationen.

angedeutet. Dies soll ausdrücken, daß die Analyse diese Werte nicht berechnen, sondern nur die Auswirkung der Zuweisung anhand der möglichen Zeigerziele ausdrücken soll.

6.4.2.2 Kontrollflußoperationen

In Abbildung 6.3 ist das gesamte Running Example, inklusive Kontrollflußanweisungen und Programmblöcken, in eine textuelle Sequenz von Operationen transformiert worden. Man erkennt den Operationsblock β_0, der das Programm selbst repräsentiert, sowie die jeweils durch die subblock bzw. subblocks–Operationen untergeordneten Operationsblöcke β_1 für den Schleifenrumpf, sowie β_2 und β_3 für then– bzw. leeren else–Zweig. Die im vorigen Abschnitt bereits in die neue Programmrepräsentation transformierten unverzweigten Sequenzen von Zuweisungen befinden sich in den Operationsblöcken β_0 und β_2.

6.4.3 Umsetzung der Instanzenmengenbeschreibung durch Automaten auf die neue Programmrepräsentation

Die in Kapitel 5 vorgestellten Prinzipien zur Beschreibung von Programminstanzenmengen durch endliche Automaten lassen sich direkt auf die neue Programmrepräsentation übertragen. Dazu werden im Folgenden die gegenüber Kapitel 5 angepassten Definitionen vorgestellt. Anstatt hier eine größtenteils zu Definition 5.22 identische Definition anzugeben, wird teils informell nur auf die notwendigen Änderungen eingegangen werden. Als Ergebnis ergibt sich für ein Eingabeprogramm, das durch die Transformation T in die neue Programmdarstellung überführt wurde, ein Automat, der analog zum Automaten aus Definition 5.22 die Menge aller Programminstanzen beschreibt, die eine bestimmte Operation erreichen.

Die Korrektheitsaspekte können identisch zur dortigen Argumentation erfolgen und werden daher hier ebenfalls nicht erneut angegeben.

6.4.3.1 Zustandsmenge

Analog zu Definition 5.14 kann, basierend auf der neuen Programmrepräsentation, eine Zustandsmenge definiert werden, die Kontrollflußzustände vor und nach Ausführung von Operationen (anstatt von Anweisungen), und Operationsblöcken (anstatt von Programmblöcken) beschreibt.

6.4.3.2 Eingabesymbole

Als Unterschied zur Programminstanzenbeschreibung aus Kapitel 5 ergibt sich eine unterschiedliche Interpretation von Programmeigenschaften, was sich in entsprechend verschiedenen Eingabesymbolen äussert. Die Eingabesymbole, die Verzweigungen charakterisieren, müssen aufgrund der Trans-

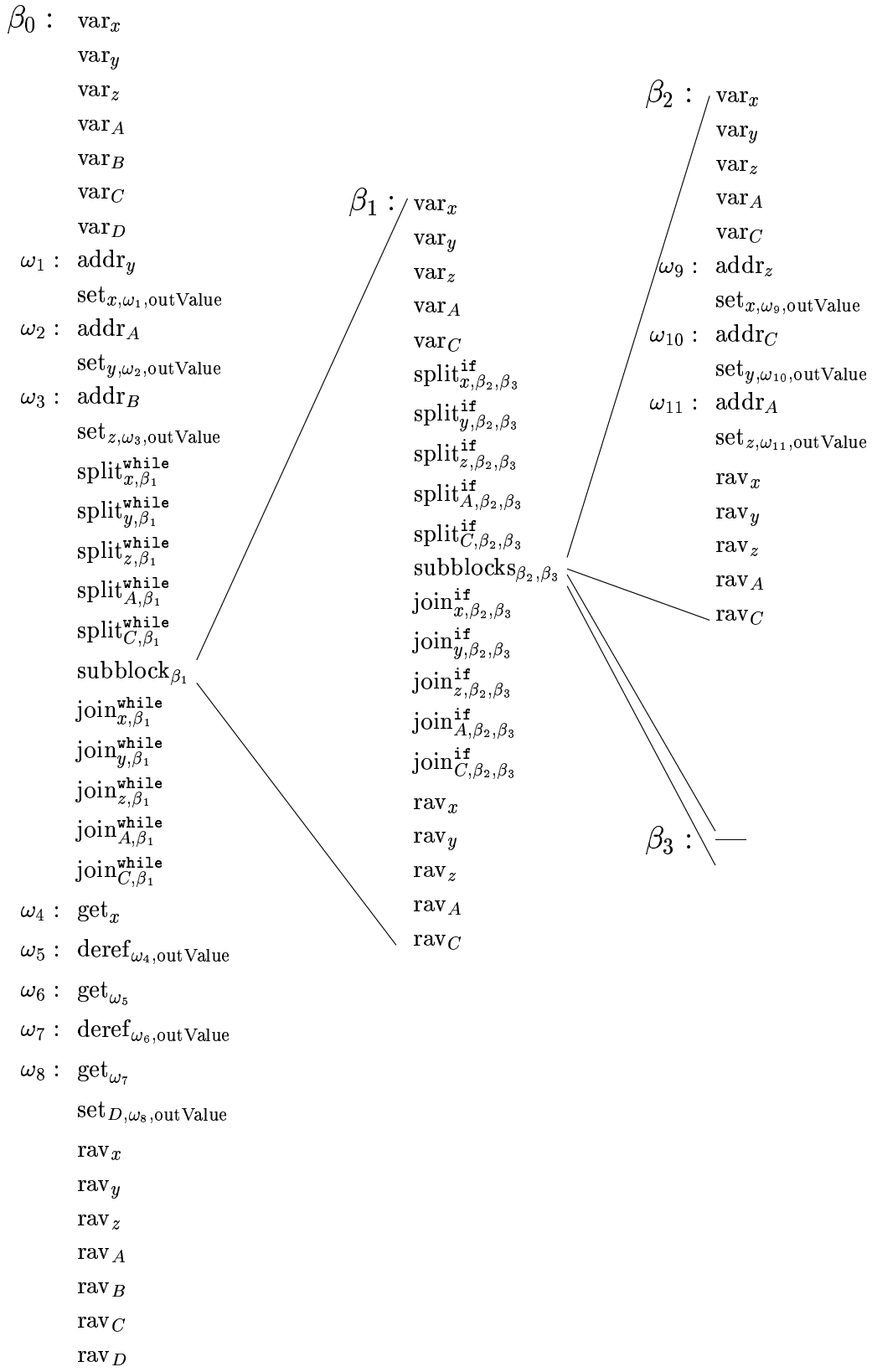


Abbildung 6.3: In eine textuelle Sequenz von Operationen transformiertes Running Example.

formation von Programmblöcken in Operationsblöcke entsprechend angepasst werden, daß sie sich an der neuen Beschreibungsform orientieren.

6.4.3.2.1 Programmeigenschaften

Die Definition von Programmeigenschaften in Bezug auf die traditionelle Programmrepräsentation in Abschnitt 5.2.1 wird wie folgt an die neue Programmrepräsentation angepasst.

Definition 6.15 (Programmeigenschaft für Operationen)
Analog zu Definition 5.1 für Anweisungen bezeichne ρ^{Ω}_{ω} *für eine Operation* $\omega \in \Omega$ *eine* Programmeigenschaft, *die an Operation* ω *gilt. Die Menge aller solchen Eigenschaften, die an Operationen* $\omega \in \Omega$ *betrachtet werden können, werde mit* $\mathcal{H}_{\Omega}$ *bezeichnet. Im Rahmen dieser Arbeit wird für jede Dereferenzierungsoperation* $\text{deref}_{\omega',p}$ *das Ergebnis der Dereferenzierung (d.h. das Setzen von* $\Psi(\text{deref}_{\omega',p})$ *durch die erweiterte Semantikfunktion) als Programmeigenschaft betrachtet. Andere Programmeigenschaften sind für das Ziel dieser Arbeit nicht relevant und werden nicht betrachtet.*

Zur Konkretisierung, welche Möglichkeiten für Programmeigenschaften in der neuen Programmrepräsentation betrachtet werden, wird zunächst definiert, auf welche Variablen eine deref–Operation aufgrund von Typbetrachtungen möglicherweise zugreifen kann.

Definition 6.16 (Referenzierungsstufe der von deref**–Operationen verw. Variablen)**
Die Funktion $\text{deref\_level} : \Omega \to \mathbb{N}_0$ *sei wie folgt definiert, wobei* V_n *wie in Definition 4.2 definiert sein soll.*

$$\begin{aligned} \text{deref\_level}(\text{get}_v) &= n, \text{ für } v \in V_n \\ \text{deref\_level}(\text{get}_{\omega'}) &= \text{deref\_level}(\omega') \\ \text{deref\_level}(\text{deref}_{\omega',p}) &= \text{deref\_level}(\omega') - 1 \end{aligned}$$

Die Funktion deref_level ordnet durch diese rekursive Definition jeder deref–Operation die Referenzierungsstufe derjenigen Variablen zu, die deren mögliche Ziele darstellen.

Definition 6.17 (Möglichkeiten für eine Programmeigenschaft für Operationen)
Analog zu Definition 5.3 werde eine Abbildung $\text{poss}_{\Omega} : \mathcal{H}_{\Omega} \to \mathcal{P}(D)$ *als* Möglichkeitenfunktion *einer Programmeigenschaft für Operationen bezeichnet. Für eine Programmeigenschaft* $\rho^{\Omega}_{\text{deref}_{\omega',p}}$ *gemäß Definition 6.15 gelte*

$$\text{poss}_{\Omega}(\rho^{\Omega}_{\text{deref}_{\omega',p}}) = \{v \mid v \in V_n \text{ mit } n = \text{deref\_level}(\text{deref}_{\omega',p})\}$$

Die Menge aller möglichen Ziele einer Dereferenzierungsoperation ergibt sich damit, wie in der Literatur vom Prinzip her üblich, als die Menge derjenigen Variablen, die passende Referenzierungsstufe, also, allgemein betrachtet, passenden Typ haben.

Definition 6.18 (Annehmen einer Möglichkeit einer Programmeigenschaft f. Operat.)
Für eine Programmeigenschaft für Operationen $\rho^{\Omega}_{\text{deref}_{\omega',p}} \in \mathcal{H}_{\Omega}$ *und* $y \in \text{poss}(\rho^{\Omega}_{\text{deref}_{\omega',p}})$ *bezeichne analog zu Definition 5.4* $\rho^{\Omega}_{\text{deref}_{\omega',p}}(y)$ *bzw. (für Zeigeranalyse im Rahmen dieser Arbeit)* $\text{deref}_{\omega',p} \to y$ *das Annehmen der Möglichkeit* y *von Eigenschaft* $\rho^{\Omega}_{\text{deref}_{\omega',p}}$ *bei Operation* $\text{deref}_{\omega',p} \in \Omega$.

Aus dieser Definition ergeben sich damit aus der sinngemäßen Umsetzung von Definition 5.18 Eingabesymbole für die Instanzenmengen beschreibenden Automaten der Form $[\text{deref}_{\omega,p} \to y]$ für alle vorkommenden deref–Operationen aus Ω_0.

6.4.3.2.2 Pfadbeschreibung

Für einen Programmblock $b \in \mathcal{B}$ und einen Operationsblock $\beta \in \Lambda$, für die $\beta = T_b(b)$ gilt, werden die Eingabesymbole $\text{exit}(b)$ und $\text{exit}(\beta)$, sowie $\overline{\text{exit}}(b)$ und $\overline{\text{exit}}(\beta)$ im Folgenden als äquivalent betrachtet. Die Zuordnung dieser Symbole zu Operationen durch eine angepasste Form der origin–Abbildung soll in einer Weise geschehen, daß diese mit den entsprechenden subblock(s)–Operationen assoziiert werden, die sich aus der Transformation T der `if`– und `while`–Anweisungen ergeben.

6.4.3.3 Zustandsübergangsfunktion

Die Zustandsübergangsfunktion δ läßt sich für Operationsblöcke analog zur Definition 5.19 für Programmblöcke beschreiben. Die Zustandsübergangsfunktion bei If– und While–Anweisungen kann auf die subblock bzw. subblocks–Operationen sinngemäß übertragen werden. Zustandsübergänge mit Symbolen, die Programmeigenschaften der Form $[\omega \rightarrow y]$ charakterisieren, werden analog zu Definition 5.18 an deref–Operationen erzeugt. Alle anderen Operationen erzeugen nur ϵ–Zustandsübergänge, wie in Abbildung 5.1(a) für Anweisungen dargestellt.

6.4.3.4 Restliche Komponenten des Automaten

Startzustand und Endzustandsmenge werden analog zur Beschreibung in den Abschnitten 5.3.1.4 und 5.3.1.5 definiert. Die Abbildung result als Bestandteil der Automaten wird auch bei der Umsetzung der Beschreibungsform auf die neue Programmrepräsentation nicht benötigt.

6.4.3.5 Definition

Die auf diese Weise konstruierten Automaten beschreiben damit alle Programminstanzen, die eine Operation ω erreichen. Entsprechend kann man diese Automaten analog zu Definition 5.22 mit M_ω für ein $\omega \in \Omega$ bezeichnen.

6.4.3.6 Anwendung auf das Running Example

In Abbildung 6.4(a) wird diese Umsetzung der Instanzenmengenbeschreibung auf die neue Programmrepräsentation anhand des Running Example veranschaulicht, das in Abbildung 6.3 in die neue Programmdarstellung transformiert wurde. Durch die Wahl eines anderen Startzustandes wird in Abbildung 6.4(a) die Menge aller Programminstanzen, die das Ende des Eingabeprogrammes erreichen, betrachtet.

Diejenigen Zustände gemäß Abschnitt 6.4.3.1 vor und nach Operationen, sowie vor und nach Programmblöcken, die nur ϵ–Übergänge besitzen, sind in grau eingezeichnet. Dem abgebildeten Automaten liegt eine bereits teilweise berechnete ϵ–Hülle zugrunde, indem mehrere ϵ–Übergänge bereits zusammengefasst wurden. In Abbildung 6.4(b) ist die vollständig berechnete ϵ–Hülle des abgebildeten Automaten dargestellt.

Die Struktur des Automaten in Bezug auf Verzweigungen charakterisierende Symbole ist mit dem Automaten aus Abbildung 5.5(b), wo der entsprechende Automat basierend auf der traditionellen Programmdarstellung berechnet wurde, identisch bis auf die Tatsache, daß den dort verwendeten Symbolen $\text{exit}(b_1), \ldots$ hier die Symbole $\text{exit}(\beta_1), \ldots$ gegenüberstehen.

Die einzigen Operationen, an denen im Running Example Programmeigenschaften betrachtet werden, sind nach Definition 6.15 die beiden deref–Operationen ω_5 und ω_7. Für diese Programmeigenschaften, die man nach Definition 6.15 mit $\rho^{\Omega}_{\omega_5}$ bzw. $\rho^{\Omega}_{\omega_7}$ bezeichnen würde, ergeben sich nach Definition 6.17 die folgenden Möglichkeiten. Für die Operation ω_5 wird zunächst $\text{deref\_level}(\omega_5)$ gemäß Definition 6.16 wie folgt berechnet:

$$\begin{aligned} \text{deref\_level}(\omega_5) &= \text{deref\_level}(\text{deref}_{\omega_4,\text{outValue}}) \\ &= \text{deref\_level}(\omega_4) - 1 \\ &= \text{deref\_level}(\text{get}_x) - 1 \\ &= 2 - 1 = 1, \text{ da } x \in V_2 \text{ (Zeiger auf Zeiger vom Basistyp)} \end{aligned} \tag{6.1}$$

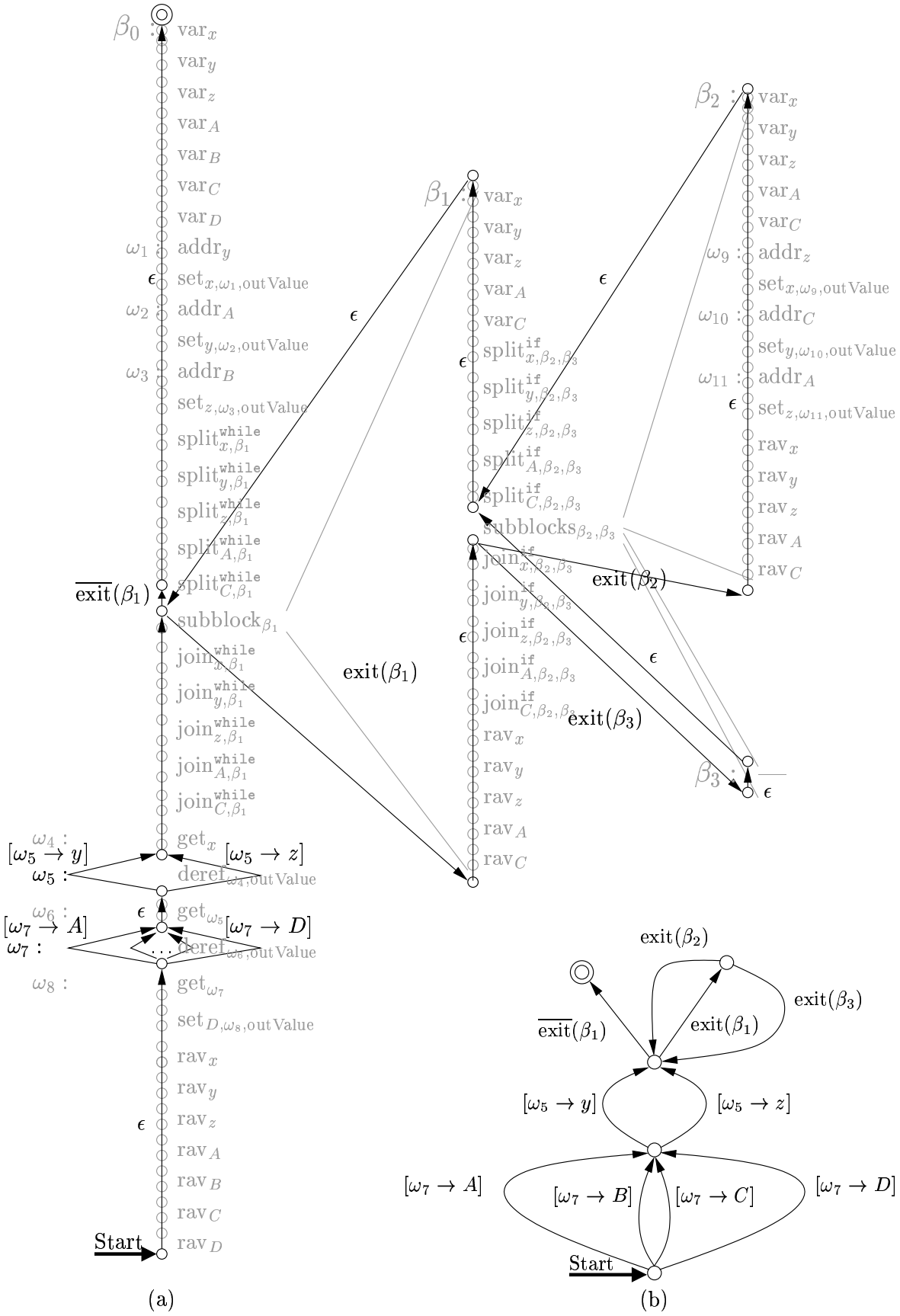


Abbildung 6.4: Anwendung der Anpassung der Definition der instanzenmengenbeschreibenden Automaten an die neue Programmrepräsentation.

Damit sind alle $v \in V_1 = \{y, z\}$ in $\text{poss}_\Omega(\rho^\Omega_{\omega_5})$ enthalten und es ergeben sich daraus die möglichen Zustandsübergänge wie in Abbildung 6.4 bei Operation ω_5 dargestellt ist.

Für ω_7 berechnet man analog

$$\begin{aligned} \text{deref\_level}(\omega_7) &= \text{deref\_level}(\text{deref}_{\omega_6,\text{outValue}}) \\ &= \text{deref\_level}(\omega_6) - 1 \\ &= \text{deref\_level}(\text{get}_{\omega_5}) - 1 \\ &= \text{deref\_level}(\omega_5) - 1 \\ &\overset{(6.1)}{=} 1 - 1 = 0 \end{aligned}$$

Damit sind alle Variablen $v \in V_0 = \{A, B, C, D\}$ Element von $\text{poss}_\Omega(\rho^\Omega_{\omega_7})$, was die Zustandsübergänge wie in Abbildung 6.4 bei Operation ω_7 eingezeichnet ermöglicht.

Vergleicht man den Automaten aus Abbildung 6.4(a) mit dem aus Abbildung 5.5(a), so erkennt man in Bezug auf die betrachteten Programmeigenschaften ebenfalls eine weitgehende Übereinstimmung. Das Zeigerziel von x entspricht dem Ziel von ω_5 und das Zeigerziel von $*x$ demjenigen von ω_7.

Während die Auswahl der betrachteten Programmeigenschaften anhand der traditionellen Programmrepräsentation allerdings intuitiv erfolgte und anschließend gemäß Vereinbarung 5.2 auf mehrere Anweisungen verteilt wurde, ergibt sich hier aus der neuen Programmrepräsentation eindeutig die Information, welche Programmeigenschaften relevant sind. Darüberhinaus wird an jeder Operation bereits per Definition nur maximal eine Programmeigenschaft betrachtet, weshalb das nach Vereinbarung 5.2 notwendig werdende Einfügen von zusätzlichen Anweisungen hier unterbleiben kann. Die neue Programmrepräsentation ergibt also u.A. ein einfaches Schema zur Bestimmung der im Rahmen der Analyse relevanten Programmeigenschaften.

6.4.4 Umsetzung der Ordnung auf Eingabesymbolen

Die grundsätzliche Struktur von Eingabeprogrammen in der traditionellen und neuen Programmrepräsentation ist identisch. Der hierarchische Aufbau der Eingabeprogramme wird unverändert in die neue Programmrepräsentation übernommen, und beide Darstellungsformen ordnen Anweisungen bzw. Operationen sequentiell in Programmblöcken bzw. Operationsblöcken an. Damit läßt sich die Ordnung auf Eingabesymbolen aus Definition 5.30, die auf der Basis der traditionellen Programmrepräsentation definiert wurde, direkt in eine analoge Beschreibung auf der Basis der neuen Programmrepräsentation umsetzen. Die Beweise, daß Definition 5.30 und die zur Definition verwendeten weiteren Relationen jeweils Ordnungen sind, ließen sich exakt analog zur dortigen Vorgehensweise auch auf der Basis der neuen Programmrepräsentation realisieren.

Im Folgenden wird davon ausgegangen, daß die Ordnung $\leq_{\widehat{E}}$ für Eingabesymbole auch eine Ordnung auf den Eingabesymbolen ist, die an den in diesem Abschnitt vorgestellten, gegenüber Kapitel 5 modifizierten Automaten zur Beschreibung von Instanzenmengen eingegeben werden können. Damit gilt z.B. für das Running Example nach Abbildung 6.4, daß

$$\langle [\omega_5 \to y] \rangle \leq_{\widehat{E}} \langle [\omega_7 \to A] \rangle$$

6.5 Integration von Use–Use–Chains in die neue Programmrepräsentation

Für die hier vorgestellten Grundoperationen der neuen Programmrepräsentation soll analog zu Use–Use–Chains eine Verkettung der Zugriffe auf eine Variable eingeführt werden.

6.5.1 Von Operationen verwendete Variablen

Dazu wird zunächst eine Funktion uses definiert, die jeder Operation $\omega \in \Omega$ und jedem Operationsblock $\beta \in \Lambda$ eine Menge von Variablen zuordnet, die von dieser Operation bzw. diesem Operations-

block (möglicherweise) gelesen oder verändert werden.

Definition 6.19 (Von Operationen verwendete Variablen)
Die Funktion uses : $\Omega \cup \Lambda \to \mathcal{P}(V)$ *sei wie folgt definiert.*

$$
\begin{aligned}
\text{uses}(\text{get}_v) &= \{v\} \\
\text{uses}(\text{get}_{\omega'}) &= \text{uses}(\omega') \\
\text{uses}(\text{set}_{v,\omega',p}) &= \{v\} \\
\text{uses}(\text{set}_{\omega'',\omega',p}) &= \text{uses}(\omega'') \\
\text{uses}(\text{addr}_v) &= \{v\} \\
\text{uses}(\text{deref}_{\omega,p}) &= \{v \mid v \in V_n \ \textit{mit}\ n = \text{deref\_level}(\text{deref}_{\omega,p})\} \\
\text{uses}(\beta) &= \bigcup_{i \in \{1,\ldots,n\}} \text{uses}(\omega_i),\ \textit{mit}\ \beta = (\omega_1, \ldots, \omega_n)\ \textit{für ein}\ n \in \mathbb{N} \\
\text{uses}(\text{split}^{\texttt{if}}_{v,\beta_1,\beta_2}) &= v \\
\text{uses}(\text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}) &= v \\
\text{uses}(\text{split}^{\texttt{while}}_{v,\beta}) &= v \\
\text{uses}(\text{join}^{\texttt{while}}_{v,\beta}) &= v \\
\text{uses}(\text{subblock}_{\beta}) &= \text{uses}(\beta) \\
\text{uses}(\text{subblocks}_{\beta_1,\beta_2}) &= \text{uses}(\beta_1) \cup \text{uses}(\beta_2) \\
\text{uses}(\text{var}_v) &= \{v\} \\
\text{uses}(\text{rav}_v) &= \{v\}
\end{aligned}
$$

Für die Definition der von einem Dereferenzierungsknoten (möglicherweise) verwendeten Variablen muß dabei zunächst eine Abschätzung verwendet werden, da die konkreten Ziele erst das Ergebnis der noch auszuführenden Analyse sein werden. Die Definition der von einem Dereferenzierungsknoten verwendeten Variablen (in Einklang mit Definition 6.17) beschreibt dabei die in der Literatur häufig getroffene Annahme, daß die Dereferenzierung einer Zeigervariable jede Variable passenden Typs ergeben kann. Für die Variablen in dieser Arbeit, die Zeiger verschiedener Referenzierungstufe auf Variablen vom Basistyp sein können, entspricht dabei die Referenzierungsstufe dem Kriterium für "passenden Typ". Beispielsweise haben Zeiger auf Zeiger auf Basistyp die Referenzierungsstufe zwei. Wird daher eine Zeigervariable der Referenzierungsstufe drei an einer Dereferenzierungsoperation dereferenziert, so sind alle Zeigervariablen mit Dereferenzierungsstufe zwei potentielle Ziele dieser Dereferenzierung. Die tatsächlichen Ziele zu bestimmen, die auch tatsächlich in einer Programminstanz vorkommen können, bleibt weiterhin Aufgabe der Analyse.

6.5.2 Letzte Verwendung von Variablen innerhalb des gleichen Operationsblockes

Damit kann nun eine Funktion definiert werden, die zu einer Variable v und einer Operation ω, die in einem Operationsblock β enthalten ist, diejenige Operation ergibt, die zuletzt (in Ausführungsrichtung) vor der Operation ω auf die Variable v zugegriffen hat. Im Folgenden bezeichne Ω_P stets die Menge von Operationen, in die ein Eingabeprogramm P durch Definition 6.14 transformiert wurde.

Zunächst läßt sich allgemein für Grundoperationen, sowie rav– und split–Operationen folgende gemeinsame Definition einer Funktion prev_use geben, die die letzte Verwendung einer Variablen im gleichen Operationsblock als Ergebnis liefert. Für var–Operationen wird dies im nächsten Abschnitt definiert werden.

Definition 6.20 (Letzte Verwendung von Variablen)
Die Funktion prev_use : $V \times \Omega_P \times \Lambda \to \Omega_P$ *sei für Operationen* $\omega \in (\Omega_0 \cap \Omega_P)$, *sowie* rav– *und*

Operation	uses	prev_use
ω_1 : var_x	x	
ω_2 : var_y	y	
ω_3 : var_z	z	
ω_4 : var_A	A	
ω_5 : var_B	B	
ω_6 : addr_y	y	ω_2
ω_7 : $\text{set}_{x,\omega_6,\text{outValue}}$	x	ω_1
ω_8 : addr_A	A	ω_4
ω_9 : $\text{set}_{y,\omega_8,\text{outValue}}$	y	ω_6
ω_{10} : addr_B	B	ω_5
ω_{11} : $\text{set}_{z,\omega_{10},\text{outValue}}$	z	ω_3
ω_{12} : rav_x	x	ω_7
ω_{13} : rav_y	y	ω_9
ω_{14} : rav_z	z	ω_{11}
ω_{15} : rav_A	A	ω_8
ω_{16} : rav_B	B	ω_{10}

Tabelle 6.3: Von dem Teil der Operationen aus Programmblock β_0 des Running Example vor der Schleife verwendete Variablen und deren jeweils letzte Verwendung.

split–*Operationen wie folgt definiert:*

$$\begin{aligned}\forall\omega \in \big((\Omega_0 \cap \Omega_P) \cup \{\text{rav}_v \mid \text{rav}_v \in \Omega_P\} \cup \{\text{split}^{\texttt{if}}_{v,\beta_1,\beta_2} \mid \text{split}^{\texttt{if}}_{v,\beta_1,\beta_2} \in \Omega_P\} \\ \cup\{\text{split}^{\texttt{while}}_{v,\beta_1} \mid \text{split}^{\texttt{while}}_{v,\beta_1} \in \Omega_P\}\big) : \\ \text{prev\_use}(v,\omega,\beta) \quad := \quad \omega_i \; \textit{mit}\; i = \max\{j \in \mathbb{N} \quad | \quad \exists n \in \mathbb{N} \; : \; \beta = (\omega_1,\ldots,\omega_n) \\ \wedge \exists k \in \{1,\ldots,n\} \; : \; \omega_k = \omega \\ \wedge 1 \leq j < k \\ \wedge v \in \text{uses}(\omega_j)\}\end{aligned}$$

Durch die Einführung der var–Operation als per Konstruktion erste Verwendung jeder Variablen in einem Operationsblock ergibt dies stets eine Operation im gleichen Operationsblock als Ergebnis.

Zur Veranschaulichung dieser Definition wird in Tabelle 6.3 der erste Teil des Operationsblockes β_0 bis zur Schleife bezüglich der Menge von Variablen untersucht, die von den in ihnen enthaltenen Operationen verwendet werden. Die Darstellungsweise inklusive der rav–Operationen entspricht dabei einem Operationsblock, der ausschließlich die angegebenen Operationen enthält. Auf die selbe Weise ist der Teil des Operationsblockes β_0 nach der Schleife in Tabelle 6.4 untersucht. Die Variablen in der zweiten Spalte dieser beiden Tabellen geben die Menge von Variablen an, die sich als Ergebnis der Funktion uses für die jeweilige Operation in der ersten Spalte ergeben. Für Dereferenzierungsoperationen bestimmt sich diese Menge gemäß Definition 6.19 wie bereits vorgestellt. In der dritten Spalte ist für jede von der Operation verwendete Variable diejenige Operation angegeben, die zuletzt (möglicherweise) auf die gleiche Variable zugegriffen hat.

6.5.3 Letzte Verwendung von Variablen in anderen Operationsblöcken

Für die von var–Operationen verwendeten Variablen läßt sich konstruktionsgemäß keine letzte Verwendung im gleichen Operationsblock finden. Ebenso kann die Operation, deren letzte Verwendung einer Variable eine rav–Operation ist, nicht im gleichen Operationsblock enthalten sein. Für diese Fälle wird die letzte Verwendung einer Variablen über die split– und join–Operationen vor und nach untergeordneten Operationsblöcken bestimmt.

Definition 6.21 (Letzte Verwendung vor var**–Operationen)**
Sei $v \in V$ *und sei die Operation* $\text{var}_v \in \Omega_P$ *im Operationsblock* $\beta \in \Lambda$ *enthalten. Die Funktion*

Operation	uses	prev_use
ω_1 : var_x	x	
ω_2 : var_y	y	
ω_3 : var_z	z	
ω_4 : var_A	A	
ω_5 : var_B	B	
ω_6 : var_C	C	
ω_7 : var_D	D	
ω_8 : get_x	x	ω_1
ω_9 : $\text{deref}_{\omega_8,\text{outValue}}$	y, z	$y : \omega_2, z : \omega_3$
ω_{10} : get_{ω_9}	y, z	$y, z : \omega_9$
ω_{11} : $\text{deref}_{\omega_{10},\text{outValue}}$	A, B, C, D	$A : \omega_4, B : \omega_5, C : \omega_6, D : \omega_7$
ω_{12} : $\text{get}_{\omega_{11}}$	A, B, C, D	$A, B, C, D : \omega_{11}$
ω_{13} : $\text{set}_{D,\omega_{12},\text{outValue}}$	D	ω_{12}
ω_{14} : rav_x	x	ω_8
ω_{15} : rav_y	y	ω_{10}
ω_{16} : rav_z	z	ω_{10}
ω_{17} : rav_A	A	ω_{11}
ω_{18} : rav_B	B	ω_{11}
ω_{19} : rav_C	C	ω_{11}
ω_{20} : rav_D	D	ω_{12}

Tabelle 6.4: Von dem Teil der Operationen aus Programmblock β_0 des Running Example nach der Schleife verwendete Variablen und deren jeweils letzte Verwendung.

prev_use : $V \times \Omega_P \times \Lambda \to \Omega_P$ *werde für diese Operation wie folgt definiert:*

$$\text{prev\_use}(v, \text{var}_v, \beta) \quad = \quad \begin{cases} \text{split}^{\texttt{if}}_{v,\beta_1,\beta_2} \in \Omega_P, & \textit{falls } \beta_1 = \beta \textit{ oder } \beta_2 = \beta \\ \text{join}^{\texttt{while}}_{v,\beta_1} \in \Omega_P, & \textit{falls } \beta_1 = \beta, \end{cases}$$

Dabei wird durch Rückführung der letzten Verwendung einer Variablen auf die join–Operation in dem Fall, daß der untergeordnete Operationsblock von $\text{split}^{\texttt{while}}_{\ldots}$ und $\text{join}^{\texttt{while}}_{\ldots}$–Operationen umschlossen wird, eine Wiederholung des gesamten Schleifenkonstruktes in Rückwärtsbetrachtung bewirkt. Dies wird im nächsten Abschnitt verdeutlicht werden.

Für die vor einer join–Operation zuletzt ausgeführte Operation gibt es allgemein mehrere Möglichkeiten. Nimmt man an, daß von einer Programminstanz in Ausführungsrichtung der untergeordnete Operationsblock β_1 ausgeführt wird, der aus dem then–Zweig einer If–Anweisung entstanden ist, dann findet sich die letzte Verwendung einer Variablen unter dieser Bedingung in β_1. Nimmt man stattdessen die Ausführung des aus dem else–Zweig entstandenen Operationsblockes β_2 an, so ist die letzte Verwendung einer Variablen in β_2 zu finden. Für die Suche nach der letzten Zuweisung zu einer Variablen gibt es an dieser Stelle also zwei Möglichkeiten. Diesem wird durch die folgende Definition einer Abbildung für bedingte letzte Verwendungen von Variablen Rechnung getragen.

Definition 6.22 (Bedingte letzte Verwendung von Variablen)
Die Funktion cond_prev_use : $V \times \Omega_P \times \Lambda \times E \to \Omega_P$ *sei wie folgt definiert. Dabei soll als Forderung an die Parameter* $\omega \in \Omega_P$ *und* $\beta \in \Lambda$ *für die betrachtete Operation* ω *gelten, daß* $\omega \in \beta$ *ist.*

$$\text{cond\_prev\_use}(v, \text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}, \beta, \text{exit}(\beta_1)) \quad = \quad \begin{cases} \text{rav}_v \in \beta_1, & \textit{falls } v \in \text{uses}(\beta_1) \\ \text{split}^{\texttt{if}}_{v,\beta_1,\beta_2} \in \beta, & \textit{sonst} \end{cases}$$

$$\text{cond\_prev\_use}(v, \text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}, \beta, \text{exit}(\beta_2)) \quad = \quad \begin{cases} \text{rav}_v \in \beta_2, & \textit{falls } v \in \text{uses}(\beta_2) \\ \text{split}^{\texttt{if}}_{v,\beta_1,\beta_2} \in \beta, & \textit{sonst} \end{cases}$$

$$\text{cond\_prev\_use}(v, \text{join}^{\texttt{while}}_{v,\beta_1}, \beta, \text{exit}(\beta_1)) \quad = \quad \begin{cases} \text{rav}_v \in \beta_1, & \textit{falls } v \in \text{uses}(\beta_1) \\ \text{join}^{\texttt{while}}_{v,\beta_1} \in \beta, & \textit{sonst} \end{cases}$$

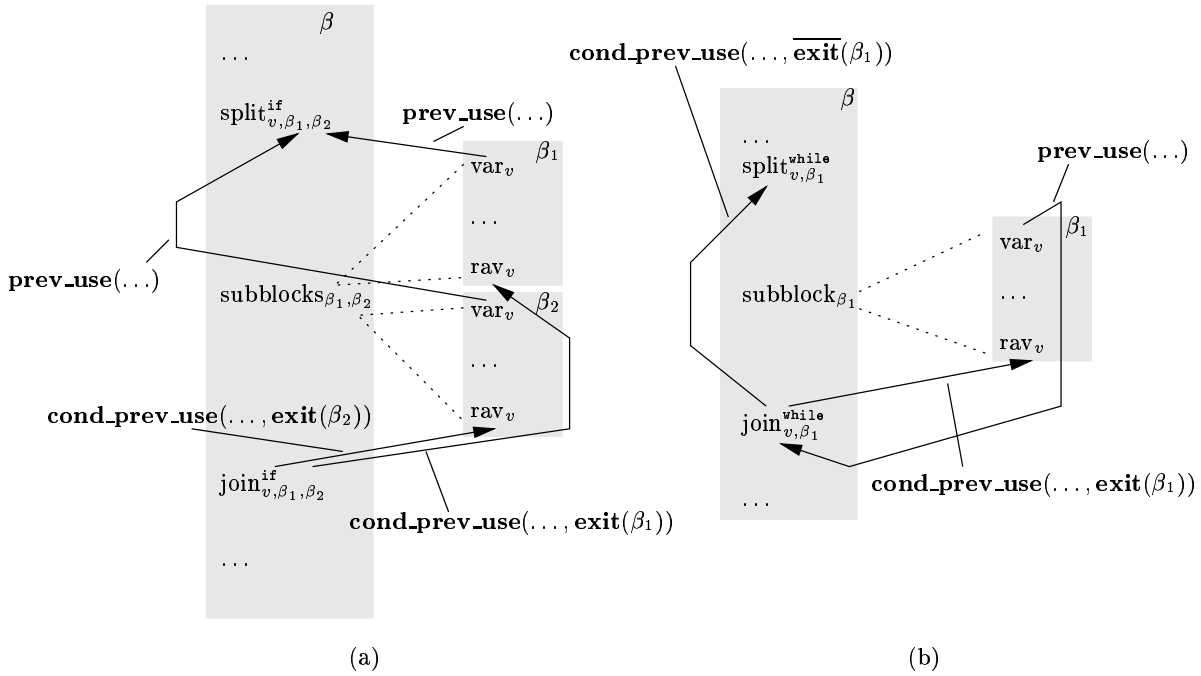


Abbildung 6.5: Operationsblockübergreifende letzte Verwendungen von Variablen.

$$\begin{aligned} \text{cond\_prev\_use}(v, \text{join}^{\texttt{while}}_{v,\beta_1}, \beta, \overline{\text{exit}}(\beta_1)) &= \text{split}^{\texttt{while}}_{v,\beta_1} \in \beta \\ \text{cond\_prev\_use}(v, \omega, \beta, e) &= \text{undef}, \ \textit{sonst} \end{aligned}$$

Damit ergibt sich als letzte Verwendung einer Variablen v vor einer $\text{join}^{\texttt{if}}_{\ldots}$–Operation die rav–Operation in demjenigen Unterblock $\beta_{\ldots}$, der durch das Symbol $\text{exit}(\beta_{\ldots})$ spezifiziert wird. Falls die Variable v in diesem Block nicht verwendet wird, ist die letzte Verwendung stattdessen die zur join–Operation zugehörige split–Operation im gleichen Operationsblock β. Für aus `while`–Anweisungen entstandene Verzweigungen ergibt sich für den Fall, daß durch das Symbol $\text{exit}(\beta_1)$ ausgedrückt wird, daß der Schleifenrumpf betreten werden soll, eine analoge Definition für cond_prev_use zu der von $\text{join}^{\texttt{if}}_{\ldots}$–Operationen, allerdings wird dort die join–Anweisung selbst als letzte Verwendung beim Betreten eines Schleifenrumpfes, der die entsprechende Variable nicht enthält, definiert. Wird stattdessen durch das Symbol $\overline{\text{exit}}(\beta_1)$ symbolisiert, daß der Schleifenrumpf umgangen werden soll, so ergibt sich als letzte Verwendung der Variablen die zugehörige split–Operation. Abbildung 6.5 zeigt zur Veranschaulichung der Funktionen prev_use und cond_prev_use die Verbindungen zwischen Operationen und den durch diese Funktionen spezifizierten letzten Operationen, die die gleiche Variable verwenden. Dabei werden in der Abbildung nur diejenigen Operationen gezeigt, deren letzte Verwendung nicht im gleichen Operationsblock zu finden ist, in dem sie sich selbst befinden. In Abbildung 6.5(a) ist ein Operationsblock β mit einigen seiner beinhalteten Operationen grau eingezeichnet, der die Umsetzung einer If–Anweisung beinhaltet. Ebenfalls grau eingezeichnet sind die Subblöcke β_1 und β_2 für den then– bzw. else–Zweig, die im Operationsblock β durch die subblocks–Operationen eingebunden sind. Von der $\text{join}^{\texttt{if}}_{\ldots}$–Operation unten im Block β ergibt die oben definierte Funktion $\text{cond\_prev\_use}(v, \text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}, \beta, e)$ die rav–Operation in Block β_1 bzw. β_2 für den Parameter $e = \text{exit}(\beta_1)$ bzw. $e = \text{exit}(\beta_2)$. Die var–Operationen sind hier direkt mit den entsprechenden split–Operationen verbunden, was sich aus der Definition von prev_use ergibt.

Der Operationsblock β in Abbildung 6.5(b) beinhaltet die aus einer While–Anweisung generierten Konstrukte. Von der $\text{join}^{\texttt{while}}_{\ldots}$–Operation ergibt sich $\text{cond\_prev\_use}(v, \text{join}^{\texttt{while}}_{v,\beta_1}, \beta, e)$ als die zugehörige split–Anweisung für $e = \overline{\text{exit}}(\beta_1)$, was ein Verlassen der Schleife in Rückwärtsrichtung signalisiert, bzw. als die rav–Operation im Schleifenrumpf β_1 für $e = \text{exit}(\beta_1)$. Von der sich dort

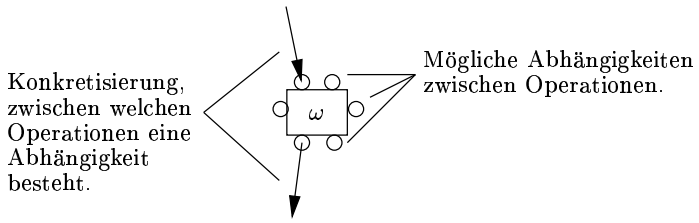


Abbildung 6.6: Grundoperationen mit ihren möglichen Abhängigkeiten als Graphknoten dargestellt.

befindenden var–Operation am Anfang des Schleifenrumpfes ist die letzte Verwendung der Variablen v als der $\text{join}^{\texttt{while}}_{\ldots}$–Knoten definiert, wodurch ein beliebig häufiges Durchlaufen des Schleifenrumpfes β_1 ermöglicht wird.

6.6 Graphbasierte neue Programmrepräsentation

Die bisher vorgestellte Definition der neuen Programmrepräsentation ist die Grundlage für die weitere formale Vorgehensweise in dieser Arbeit, speziell für die Definition der Queryverarbeitung und den Beweis der Korrektheit des Verfahrens. Zur Transformation der Analyse in ein bedingtes Erreichbarkeitsproblem wird darüber hinaus eine graphbasierte Darstellung des Programmes benötigt. Dazu transformiert man direkt die Operationen aus der textuellen Darstellung in Graphknoten und drückt die verschiedenen Abhängigkeiten zwischen diesen Operationen durch Kanten aus.

Damit erhält man einen Analysegraphen, der als Grundlage der konkreten Queryberechnung dient, und als positiven Nebeneffekt eine Veranschaulichung der langen textuellen Sequenzen von Operationen leistet. Durch den direkten Zusammenhang zwischen textueller und graphbasierter Darstellung kann auf die explizite Definition einer Semantik der Graphen verzichtet, und stattdessen auf die entsprechenden Definitionen für die textuelle Darstellung zurückgegriffen werden.

6.6.1 Grundoperationen als Graphknoten

Abbildung 6.6 zeigt dabei die allgemeine Form der Darstellung einer Operation als Graphknoten. Das Kästchen repräsentiert die Operation ω. Die Kreise entlang des Randes des Kästchens entsprechen den verschiedenen Möglichkeiten von Abhängigkeiten zwischen dieser und anderen Operationen. Durch die Position der Kreise oben und unten, bzw. links und rechts werden Art und Richtung der Abhängigkeiten ausgedrückt. Welche Abhängigkeiten dies sein können, wird im nächsten Abschnitt eingeführt. Die Kanten konkretisieren, zwischen welchen Operationen eine Abhängigkeit besteht.

6.6.2 Abhängigkeiten zwischen Grundoperationen als Graphkanten

Für Grundoperationen wurden bisher einige Eigenschaften definiert, die diese kennzeichnen. Dazu gehört zum einen die Menge von Variablen $\text{uses}(\omega)$, die von einer Grundoperation $\omega \in \Omega_0$ verwendet werden. Die zur Grundoperation gehörige erweiterte Semantikfunktion legt durch den Anteil, der eine Variablenbelegung verändert, fest, wie genau die verwendeten Variablen benutzt werden, also ob auf sie lesend oder schreibend zugegriffen wird und welcher neue Wert ggfs. der Variablen bei der Ausführung der Grundoperation zugewiesen wird. Als weitere Information über eine Grundoperation existieren die Mengen cons_ω und prod_ω von benannten Werten, die bei der Ausführung der Operation $\omega \in \Omega_0$ konsumiert bzw. produziert werden. Und als dritte Komponente der erweiterten Semantikfunktion wurde eine Variablenselektionsfunktion vorgestellt, die Dereferenzierungsergebnisse von Zeigervariablen zwischen verschiedenen Operationen transferieren kann.

Die für die verschiedenen Abhängigkeits– und damit auch Kantentypen verantwortlichen Fälle ergeben sich daraus wie folgt:

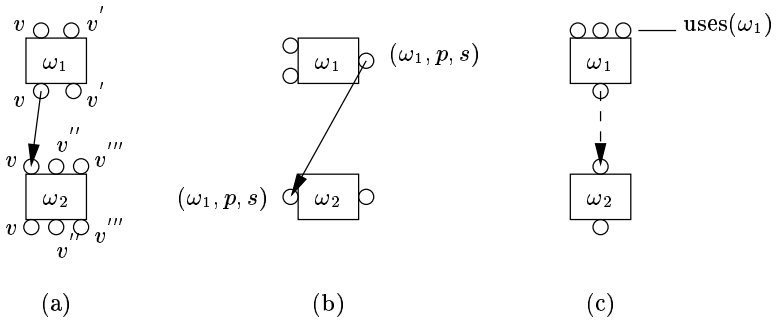


Abbildung 6.7: Verschiedene Beziehungen zwischen Grundoperationen.

- Für zwei Operationen $\omega_1, \omega_2 \in \Omega_0$, einen Operationsblock $\beta \in \Lambda$ mit $\omega_1, \omega_2 \in \beta$ und eine Variable $v \in V$ mit $v \in \text{uses}(\omega_1) \cap \text{uses}(\omega_2)$ gilt: $\text{prev\_use}(v, \omega_2, \beta) = \omega_1$. In diesem Fall stellt ω_1 die von ω_2 aus gesehen letzte Verwendung einer gemeinsamen Variablen dar. Diese Art von Abhängigkeit ist in Abbildung 6.7(a) dargestellt. Zur Kennzeichnung dieser Art von Abhängigkeit erhält jeder eine Operation $\omega \in \Omega_0$ repräsentierende Graphknoten für jede Variable $v \in \text{uses}(\omega)$ je eine "Verbindungsstelle" für Kanten oberhalb und unterhalb des Graphknotens. Die oben beschriebene Abhängigkeit zwischen ω_1 und ω_2 wird damit durch eine Kante von der v repräsentierenden Verbindungsstelle unterhalb der letzten Verwendung der Variablen, hier also Operation ω_1 zur ebenfalls v repräsentierenden Verbindungsstelle oberhalb von ω_2 dargestellt. Analog werden Kanten zwischen entsprechenden Operationen in verschiedenen Operationsblöcken eingeführt.

- Für zwei Operationen $\omega_1, \omega_2 \in \Omega_0$ existiert ein benannter Wert $r = (\omega_1, p, s)$ mit $r \in \text{prod}(\omega_1)$ und $r \in \text{cons}(\omega_2)$. Dazu wird jeder von einer Operation konsumierte benannte Wert durch eine Verbindungsstelle am linken Rand des die Operation repräsentierenden Knotens, und jeder von der Operation produzierte benannte Wert durch eine Verbindungsstelle am rechten Rand des Knotens symbolisiert. Die oben beschriebene Abhängigkeit wird nun durch eine Kante von der Verbindungsstelle von ω_1 am rechten Rand, die den produzierten benannten Wert repräsentiert, zur Verbindungsstelle am linken Rand von ω_2 dargestellt, die den gleichen, hier konsumierten benannten Wert repräsentiert, symbolisiert. Dies wird in Abbildung 6.7(b) dargestellt.

- Für zwei Operationen $\omega_1, \omega_2 \in \Omega_0$ gilt: $\omega_2 = \text{get}_{\omega_1}$ oder $\omega_2 = \text{set}_{\omega_1}$, d.h. die Variable, die von der Operation ω_2 verwendet wird, hängt vom Ergebnis der Operation ω_1 ab, die dabei nur eine Dereferenzierungsoperation sein kann. Per Definition der Abbildung uses haben beide Operationen die gleiche Menge von verwendeten Variablen, so daß eine Verbindung der Knoten wie in Abbildung 6.7(a) keine zusätzliche Information in die graphische Darstellung bringen würde. Stattdessen werden diese Knoten durch eine andere Art von Kante verbunden, die die Abhängigkeit des zweiten Knotens, der die Operation ω_2 repräsentiert, von Ergebnis des ersten Knotens, der ω_1 repräsentiert, ausdrückt. Dies ist in Abbildung 6.7(c) durch eine gestrichelte Kante dargestellt. Wenn die Art der Beziehung zwischen den die Operationen repräsentierenden Knoten aus dem Zusammenhang eindeutig ist, wird diese Kante wie die anderen Kanten ebenfalls nicht–gestrichelt dargestellt werden.

6.6.3 Aufzählung der möglichen Abhängigkeiten zwischen verschiedenen Arten von Operationen

In Abbildung 6.8 ist nun eine entsprechende Umsetzung der in Abschnitt 6.3.1.1 eingeführten Grundoperationen dargestellt. Dabei gehen die dort definierte erweiterte Semantikfunktion, sowie die in Abschnitt 6.5.1 definierte Spezifikation der Menge der von einer Grundoperation verwendeten Variablen in die spezielle Darstellungsform ein. Aus dieser Definition kann für jede Grundoperation

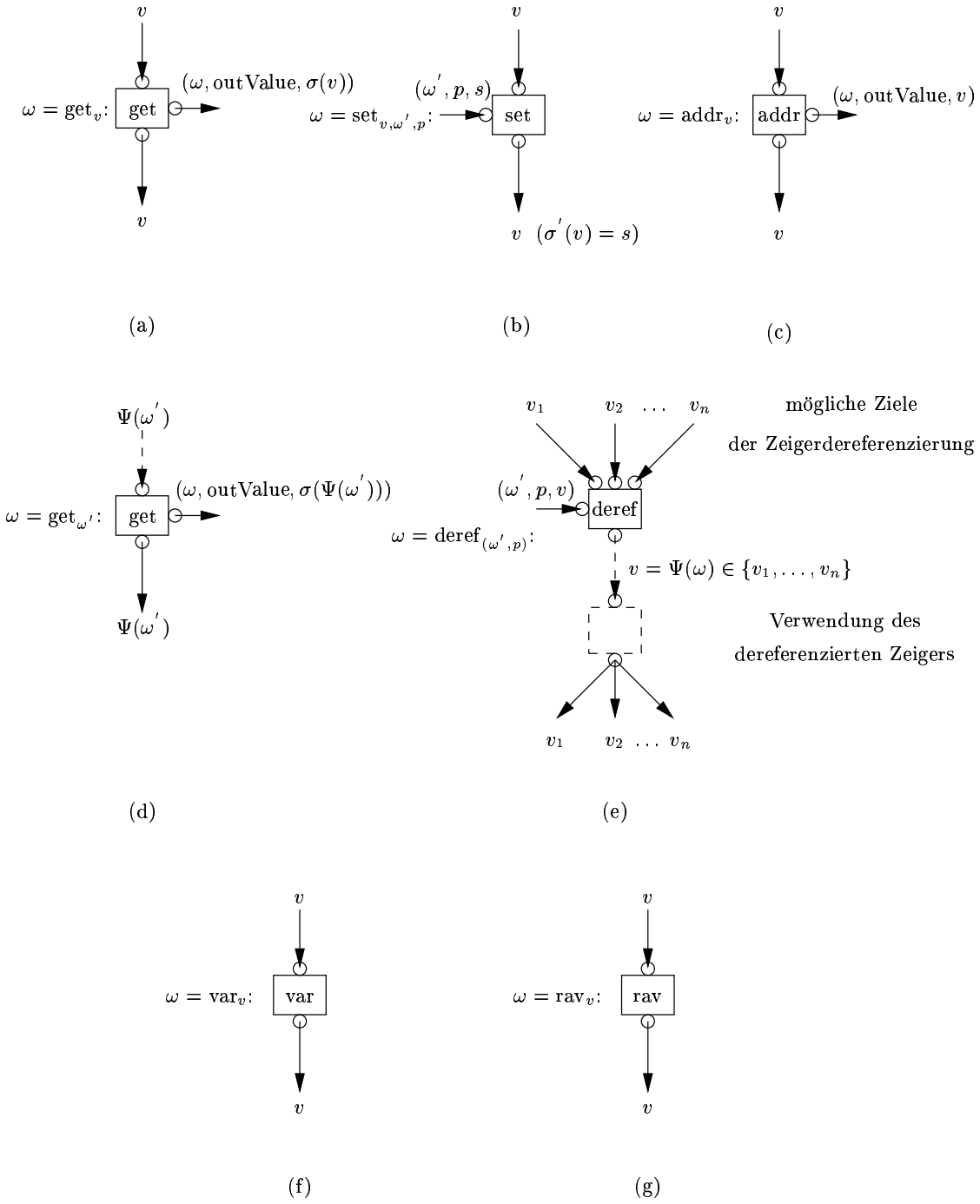


Abbildung 6.8: Darstellung aller Grundoperationen, sowie von var– und rav–Operationen.

festgestellt werden, auf welche Arten ein sie repräsentierender Graphknoten mit anderen Graphknoten in Verbindung stehen kann. In der Abbildung ist dies durch die Darstellung von entsprechenden Verbindungsstellen an den Knoten ersichtlich.

In Abb. 6.8(a) ist die Darstellung der Get–Operation $\omega = \text{get}_v$ gezeigt, die die Variable v verwendet und einen benannten Wert $(\omega, \text{outValue}, \sigma(v))$ produziert, der entsprechend der Definition der get–Operation als Inhalt die Belegung der Variable v enthält. In Teilabbildung (b) ist eine set–Operation als Knoten dargestellt, die die Variable v verwendet und einen benannten Wert konsumiert, dessen Inhalt s der Variable v zugewiesen wird. In Teilabb. (c) ist eine Adressoperation als Knoten dargestellt, die eine Variable v verwendet und als benannten Wert die Variable $v \in D$ (bzw. die Adresse dieser Variable) produziert. In Teilabb. (d) ist die zweite Variante der get–Operation, beschrieben durch $\text{get}_{\omega'}$, abgebildet. Diese greift auf diejenige Variable zu, die durch $\Psi(\omega')$ bestimmt wird. Damit hängt die von diesem Knoten verwendete Variable vom Ergebnis der Operation ω' ab, was durch eine Kante entsprechend Abb. 6.7(c) dargestellt wird. Analog existiert auch eine set–Operation bzw. ein set–Knoten mit der gleichen Art von Abhängigkeit, der hier nicht explizit dargestellt ist. In Abbildung 6.8(e) ist die Darstellung der Dereferenzierungsoperation gezeigt, die dabei dem Knoten ω_1 aus Abbildung 6.7(c) entspricht. Obwohl nicht in der Menge der Grundoperationen enthalten, sind var– und rav–Operationen als Kontrollfluß–Operationen Bestandteile von Operationsblöcken und ihre entsprechenden Graphknoten werden in den Teilabbildungen (f) und (g) an dieser Stelle zusammen mit den Grundoperationen vorgestellt. Diese Operationen sind per Definition eine Verwendung der entsprechenden Variablen, aber haben keine Auswirkung auf die Variablenbelegung, Menge der benannten Werte oder Variablenselektionsfunktion. Die Graphknoten, die die weiteren Kontrollfluß–Operationen repräsentieren, werden in Abschnitt 6.6.5 vorgestellt werden.

6.6.4 Zusammensetzen von Grundoperationen zu Graphen

Bisher wurden verschiedene Möglichkeiten vorgestellt, wie Graphknoten prinzipiell miteinander in Verbindung stehen können, sowie verschiedene Darstellungsarten, um dies graphisch zu unterscheiden. Außerdem wurde für die Menge von Grundoperationen vorgestellt, auf welche Weise sie jeweils konkret mit anderen Operationen in Verbindung stehen können, und dies durch die Darstellung von entsprechenden Verbindungsstellen an den sie repräsentierenden Knoten symbolisiert. Damit läßt sich eine Sequenz von Grundoperationen in eine Sequenz von Graphknoten transformieren, die das Verhältnis der Grundoperationen zueinander anhand von Kanten ausdrückt. Das Ergebnis dieser Vorgehensweise, angewandt auf den Teil des Running Example, der in Tabelle 6.3 als Sequenz von Grundoperationen bezüglich der von seinen Operationen verwendeten Variablen untersucht wurde, ist dabei in Abbildung 6.9(a) dargestellt.

Als nächster Schritt kann nun diese Darstellung "entfaltet" werden, indem die Knoten aus einer eindimensionalen Sequenz in eine zweidimensionale Anordnung transferiert werden, in der in der zusätzlichen Dimension die Operationen nach den von ihnen verwendeten Variablen angeordnet werden. Daraus ergibt sich für das Beispiel die Darstellung wie in Abbildung 6.9(b). Die daraus entstandene Graphstruktur spiegelt immer noch die ursprüngliche Reihenfolge der Grundoperationen eindeutig wieder. Zu einer Beschreibung der Abhängigkeiten genügt es aber, diese Darstellung vertikal zu komprimieren, indem mehrere Knoten in "Ränge" zusammengefasst werden. Dabei soll die Reihenfolge von Operationen, die auf die gleiche Variable zugreifen, weiterhin eindeutig erkennbar sein, zum anderen aber, wenn möglich, die Operationen, die aus einer gemeinsamen Anweisung des Eingabeprogrammes entstanden sind, auch in einem gemeinsamen Rang zu finden sein.

Definition 6.23 (Regeln für die Einordnung von Graphknoten in Ränge)
Für die Bestimmung von Rängen von Graphknoten sollen die folgenden Regeln verwendet werden.

- *Knoten, die* var*–Operationen entsprechen, haben Rang 0.*
- *Ein Knoten, der die gleiche Variable verwendet wie ein zuvor in der Sequenz von Operationen enthaltener anderer Knoten besitzt einen um mindestens eins größeren Rang.*

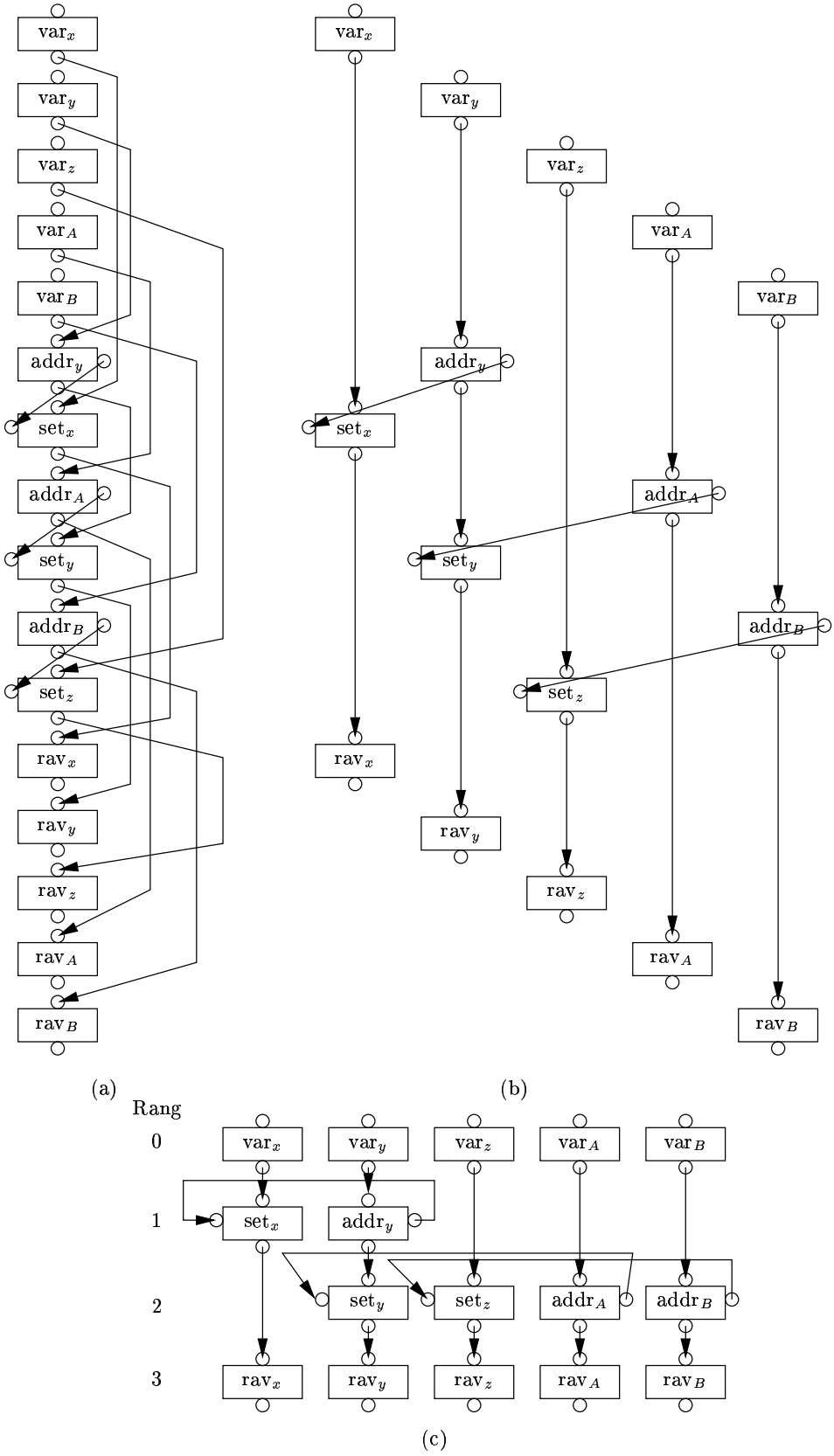


Abbildung 6.9: Erzeugen eines Analysegraphen aus einem Teil des Running Example.

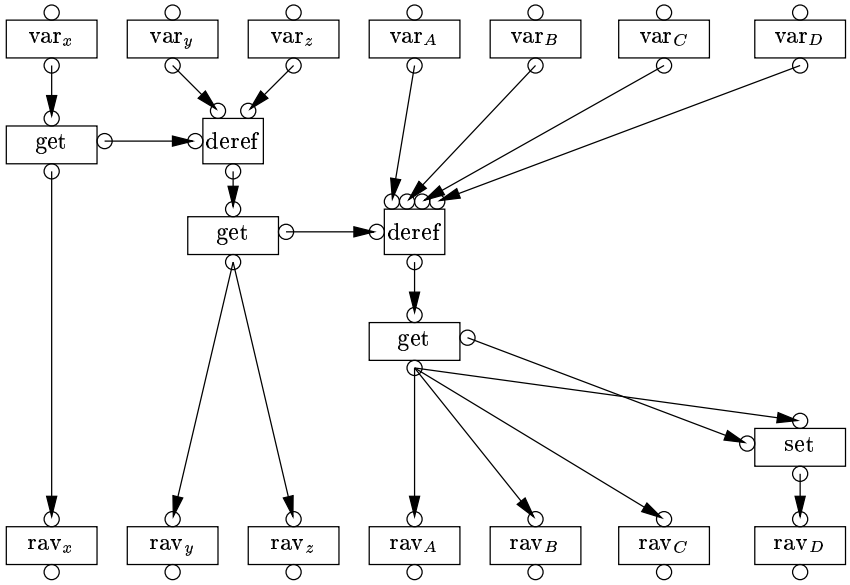


Abbildung 6.10: Erzeugen eines Analysegraphen aus der zweifachen Dereferenzierung von x im Running Example.

- *Ein Knoten, der den von einem anderen Knoten, der zuvor in der Sequenz von Operationen enthalten ist, produzierten benannten Wert konsumiert, hat mindestens den gleichen Rang wie dieser Knoten. Sofern möglich, soll der Produzent eines benannten Wertes sogar den gleichen Rang wie der Konsument besitzen.*

- *Kein Knoten hat niedrigeren Rang, als ein Knoten, der vor diesem in der Sequenz von Operationen steht.*

- *Sämtliche* rav*–Operationen befinden sich gemeinsam auf einem Rang, auf dem sich keine anderen Operationen befinden.*

Das Ergebnis dieser Einordnung in Ränge ist in Abbildung 6.9(c) dargestellt. In Abbildung 6.10 ist bereits das Ergebnis der gleichen Vorgehensweise für die zweite bisher explizit betrachtete Sequenz von Grundoperationen aus Tabelle 6.4 abgebildet. Dabei wurden bei den get– und set–Knoten die jeweils verwendeten Variablen aus der Beschriftung des Knotens weggelassen, da diese aus der Verbindung des Knotens in der Use–Use–Chain eindeutig hervorgehen.

6.6.5 Integration von Kontrollflußoperationen und untergeordneten Operationsblöcken in die Graphendarstellung

Als letzter Schritt bleibt noch, die durch die subblock(s)–Operation in andere Operationsblöcke integrierten Operationsblöcke, sowie die split– und join–Operationen in die Analysegraphdarstellung zu integrieren. Dazu transformiert man zunächst die integrierten Blöcke in graphbasierte Repräsentationen und verwendet diese in den Graphen als Meta–Knoten, die die split– und join–Operationen als Verbindungen zum Operationsblock, in den dieser integriert werden soll, beinhalten. In Abbildung 6.11 wird diese Vorgehensweise für split$^{\texttt{if}}_{\ldots}$ und join$^{\texttt{if}}_{\ldots}$–Operationen gezeigt. Die beiden untergeordneten Blöcke β_1 und β_2 werden mitsamt ihrer var– und rav–Operationen in die Darstellung integriert und durch ihre split– und join–Operationen wie eine Grundoperation mit den anderen Operationen aus dem übergeordneten Operationsblock verbunden. Das Gegenstück zur links in der Abbildung gezeigten Sequenz von Operationen ist dabei der rechts abgebildete Graph.

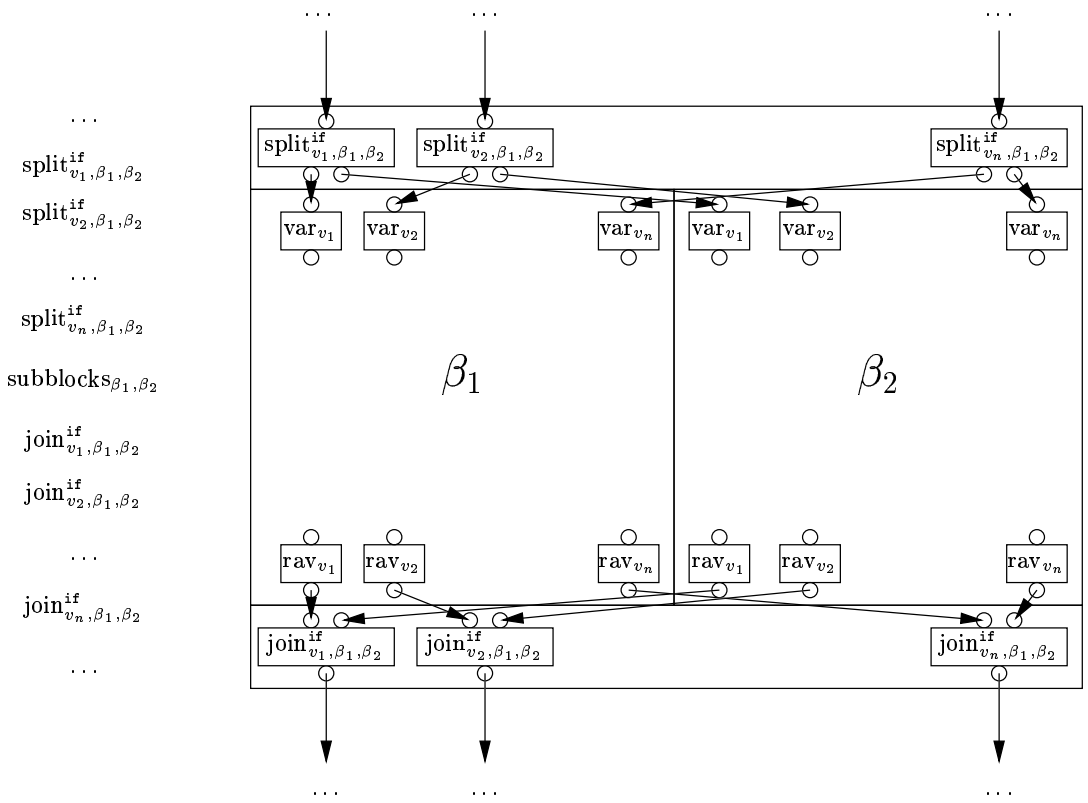


Abbildung 6.11: Integration von Unterblöcken mit split$^{\texttt{if}}_{\ldots}$ und join$^{\texttt{if}}_{\ldots}$–Operationen.

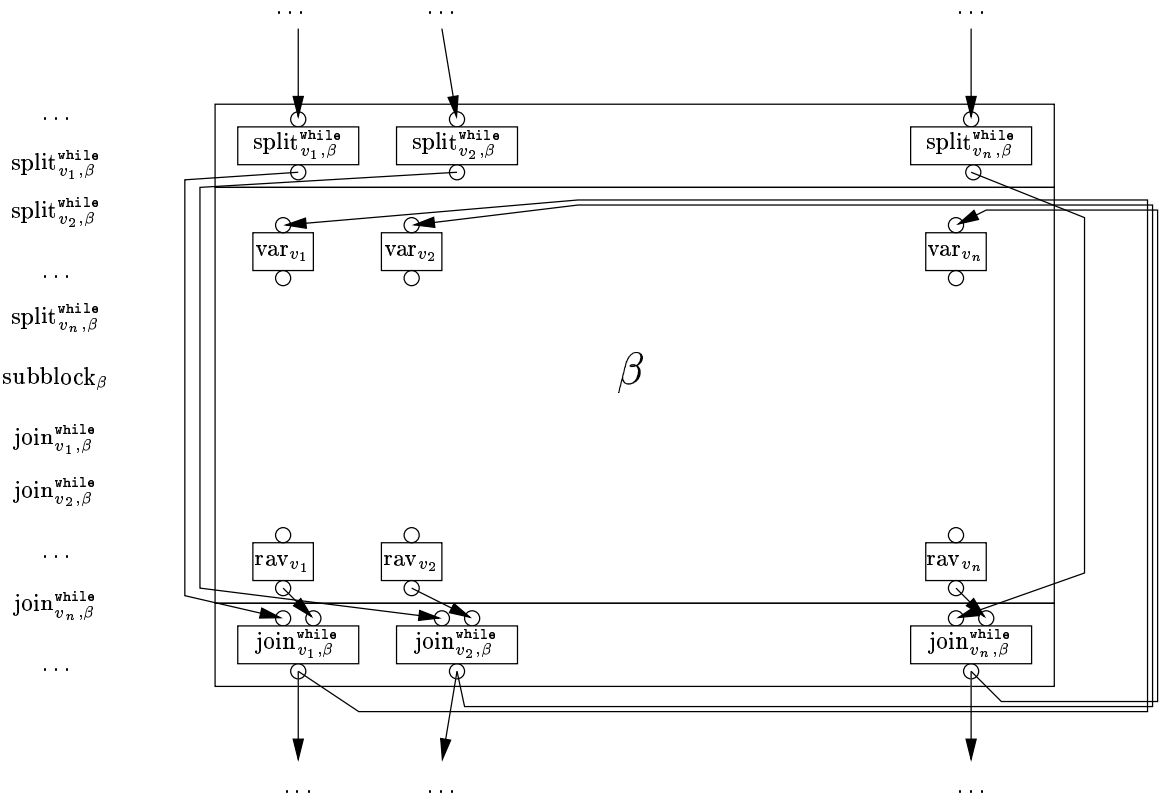


Abbildung 6.12: Integration von Unterblöcken mit split$^{while}_{...}$ und join$^{while}_{...}$–Operationen.

Die Integration eines untergeordneten Operationsblockes β mit $\text{split}^{\texttt{while}}_{\ldots}$ und $\text{join}^{\texttt{while}}_{\ldots}$–Operationen ist auf die gleiche Weise in Abbildung 6.12 dargestellt. Dabei sind die Kanten gemäß der Definition von cond_prev_use eingezeichnet. Diese Kantenverbindung ermöglicht die Beschreibung des möglichen Kontrollfluß aus der Rückwärtsbetrachtungsweise der Queries.

In Abbildung 6.13 ist die vollständige graphbasierte Repräsentation des Running Example dargestellt. Man erkennt die geschachtelten Unterblöcke für die While– und darin enthaltene If–Anweisung, sowie Sequenzen von Grundoperationen vor und nach diesen geschachtelten Unterblöcken, sowie im Block b_2, der als Unterblock von Block b_1 enthalten ist.

6.7 Zusammenfassung

Die neue Programmrepräsentation, die in dieser Arbeit verwendet wird, wurde in zwei verschiedenen Sichtweisen vorgestellt. Eine formale Definition der Darstellung von Eingabeprogrammen durch textuelle Sequenzen von Operationen bildet die Grundlage für die Beschreibung und Verwendung der neuen Programmrepräsentation. Für diese neue Programmrepräsentation wurde eine erweiterte Semantik definiert. Eingabeprogramme in der Darstellungsform aus Kapitel 4 werden durch eine Transformation T in die neue Programmdarstellung übersetzt. Anhand der traditionellen Semantikfunktion der Eingabeprogramme, und der erweiterten Semantikfunktion derer neuer Repräsentation wurde die Äquivalenz der Programmdarstellungen bewiesen.

Für die Darstellung des Analyseverfahrens in praktischer Sichtweise wurde eine Interpretation der neuen Programmrepräsentation als Analysegraphen vorgestellt. Diese Graphen dienen im nächsten Kapitel als Grundlage der Queryverarbeitung.

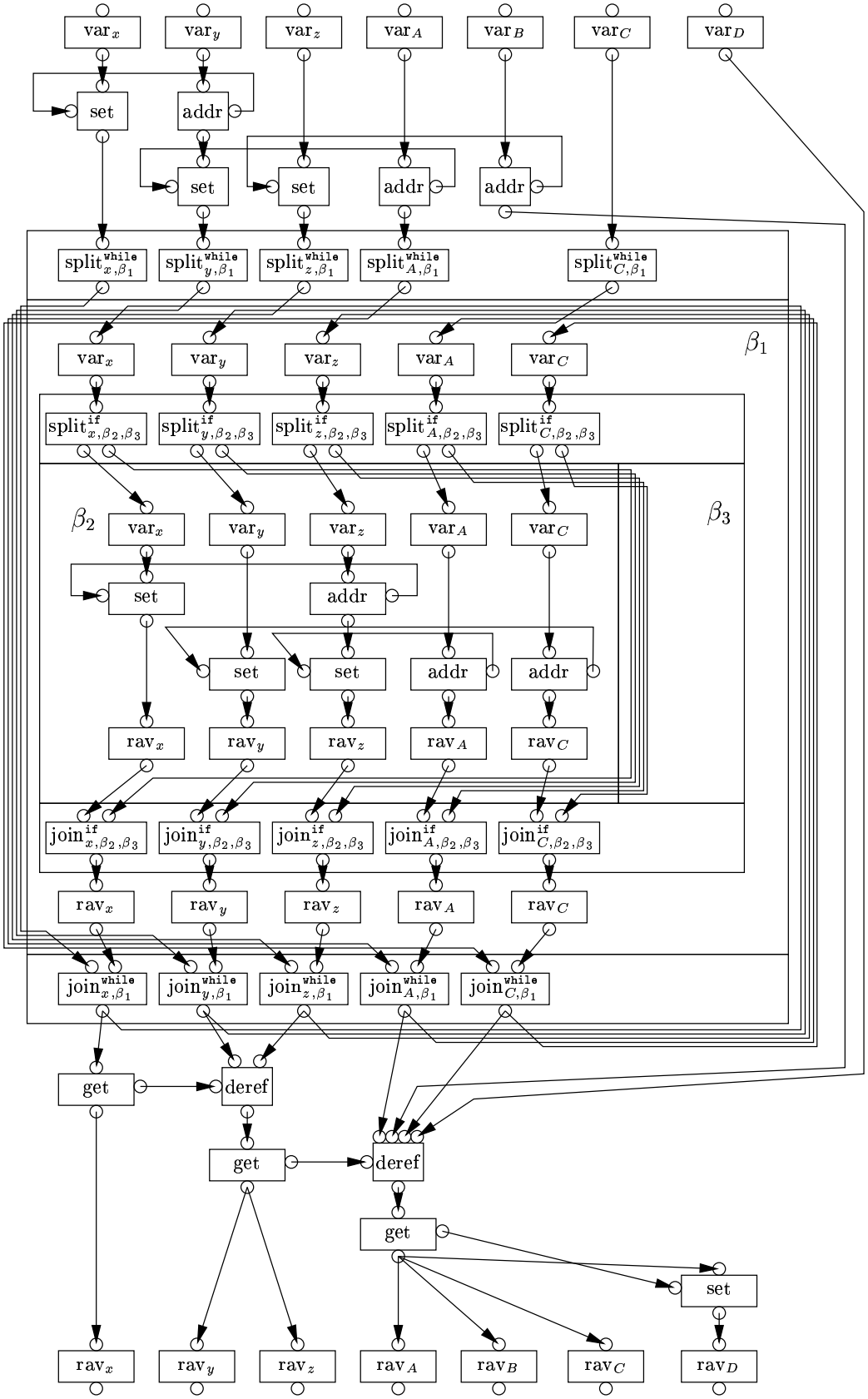


Abbildung 6.13: Darstellung des Running Example in Graphform.

Kapitel 7

Queries

Aufbauend auf der bereits vorgestellten Vorgehensweise, Mengen von Programminstanzen durch endliche Automaten zu beschreiben, und der neuen Programmrepräsentation, die zur Formulierung von Zeigeranalyse als Erreichbarkeitsproblem eingeführt wurde, kann nun die eigentliche Analyse vorgestellt werden.

7.1 Prinzip

Die Analyse wird durch Queries realisiert. Eine Query repräsentiert eine Instanz der Graphsuche im Sinne des (in Abschnitt 3.3.2.1.2 informell eingeführten) bedingten Erreichbarkeitsproblems. Queries werden bei jeder Dereferenzierungsoperation generiert, und von dort aus rückwärts durch die neue Programmrepräsentation geschickt, bis sie ein Ergebnis in der Form einer Adressoperation finden. Eine Query "betrachtet" dabei verschiedene Operationen und wird, abhängig von der Art der Operation und ihren eigenen Eigenschaften, dabei erfolgreich beendet oder setzt an einer oder mehreren andere Operationen ihre "Betrachtung" fort.

Zu den Aufgaben einer Query gehört es, den bei der Analyse durchlaufenen Programmpfad zu protokollieren. Dazu werden während der Querybearbeitung endliche Automaten (wie in Kapitel 5 eingeführt) erzeugt, die nach erfolgreicher Terminierung der Query eine Menge von Pfaden beschreiben. Das Ergebnis der Query wird vollständig von diesen Automaten beschrieben. Für jeden Endzustand $q_f \in F$, der während der Analyse erzeugt wird, besagt die Abbildung $\text{result}(q_f) = v \in V$, welche Variable v Ziel der Dereferenzierungsoperation ist, wenn eine Programminstanz einen "passenden" Pfad durchläuft. Die Menge aller solchen "passenden" Pfade sind dabei genau durch diejenigen Folgen von Eingabesymbolen charakterisiert, deren Eingabe an den Automaten diesen von seinem Startzustand in den Zustand q_f überführen. Damit sind diese Automaten Pfadbedingungen im Sinne von Definition 5.37.

Die Automaten, die für Dereferenzierungsoperationen als Ergebnis berechnet wurden, können auch zur Beschränkung des Queryflusses dienen, wenn andere Queries von diesen Ergebnissen abhängen. Damit wird die Analyse zum bereits vorgestellten bedingten Erreichbarkeitsproblem. Die Abhängigkeit von Queries von Querybedingungen, in der Form von Ergebnissen von anderen Queries gegeben, wird in den erzeugten Automaten durch Zustandsübergänge ausgedrückt, die Programmeigenschaften beschreiben.

Die Behandlung von Automaten bei der Bearbeitung der Queries wird sich im Korrektheitsteil dieses Kapitels auf die in Abschnitt 5.7 vorgestellten Bedingungsautomaten zurückführen lassen. Bis dahin soll aber zunächst eine intuitive, davon unabhängige Vorstellung der hinter der Queryverarbeitung liegenden Prinzipien gewählt werden. Zu diesem Zweck soll im Folgenden vor der formalen Definition der Queries ein informeller Überblick über einige der bei der Queryverarbeitung angewandten Prinzipien gegeben werden. Dazu gehört zunächst der Zusammenhang zwischen dem Queryfluß und den Kanten, die im Analysegraphen zwischen Operationen bzw. Knoten eingeführt wurden. Des weiteren wird ein Überblick gegeben, wie die Automaten zur Beschreibung von Pfad-

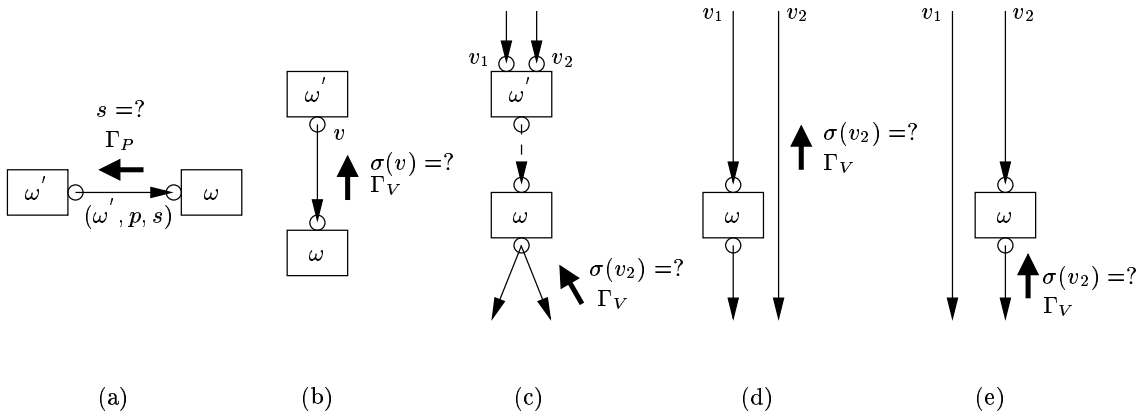


Abbildung 7.1: Zusammenhang zwischen Queries und Operationen–Abhängigkeiten.

mengen von Queries erzeugt werden, was ein Ergebnis einer Query ist, und wie diese Ergebnisse in die Automatensichtweise integriert werden. Die weiteren Abschnitte behandeln das Erkennen von Wiederholungen von Queries zum Zweck der Vermeidung von unendlichen Berechnungen und das Hinzufügen, sowie das Verarbeiten von Querybedingungen in der Form von anderen Automaten zu einer Query. Dabei wird die Queryverarbeitung an den jeweils in den Beispielen vorkommenden Operationen informell vorgestellt. Diese Beispiele bestehen der Überschaubarkeit wegen nur aus kleinen Graph– bzw. Programmfragmenten. Die Anwendung der Analyse auf das Running Example wird erst im Anschluß an die formalen Definitionen vorgestellt werden.

7.2 Zusammenhang zwischen Queries und Abhängigkeiten zwischen Operationen

Das Prinzip der Queries basiert auf den bereits vorgestellten verschiedenen möglichen Abhängigkeiten zwischen Operationen. Eine Dereferenzierungsoperation $\omega = \text{deref}_{\omega',p}$ konsumiert z.B. bei ihrer Ausführung einen benannten Wert (ω', p, s), der als Inhalt s eine Variable $v \in D$ enthält. Welche Variable genau dies ist, ist abhängig von den von der Programminstanz durchlaufenen Programmpfaden. Die Operation ω' bestimmt aber, wie der benannte Wert gebildet werden soll, d.h. ob z.B. die aktuelle Belegung einer Variablen oder (im Fall, daß ω' eine Adressoperation ist) stets eine konkrete Variable zum Inhalt s des benannten Wertes (ω', p, s) gemacht werden soll. Damit hängt das Ziel der Dereferenzierungsoperation bei ihrer Ausführung vom Ergebnis der Operation ω' ab. Dort wird der von ω konsumierte benannte Wert produziert und dort entscheidet sich auch zumindest zum Teil, welche Variable zum Inhalt des benannten Wertes gemacht wird. Interessiert man sich nun für die möglichen Ziele der Dereferenzierungsoperation, so muß man zu diesem Zweck die möglichen Inhalte s des benannten Wertes (ω', p, s) herausfinden. Dies ist die Aufgabe von Γ_P, der ersten der beiden rekursiven Queryfunktionen, die in diesem Fall die möglichen Inhalte für den von der Operation ω' produzierten benannten Wert ermitteln soll. Diese Situation ist in Abbildung 7.1(a) dargestellt. Um die Frage nach dem möglichen Inhalt des benannten Wertes (ω', p, s) beantworten zu können, wird aus der Sicht der Operation ω eine Query Γ_P zum Produzenten dieses benannten Wertes, also zur Operation ω' geschickt.

Eine zweite Art der Abhängigkeit ist diejenige, die in Abbildung 7.1(b) dargestellt ist. Dabei stehen zwei Operationen dadurch in Beziehung zueinander, daß sie zwei aufeinanderfolgende Zugriffe auf die gleiche Variable $v \in V$ darstellen. Interessiert man sich bei der abgebildeten Operation ω für die möglichen Belegungen der Variablen v, so ergibt sich diese aus der letzten Verwendung der gleichen Variablen, hier also der Operation ω'. In diesem Fall wird eine Query Γ_V, die die zweite Form der rekursiven Queryfunktionen darstellt, zu dieser letzten Verwendung der Variablen geschickt, um

die möglichen Belegungen der Variablen v zu bestimmen.

Die dritte Art der Abhängigkeiten von Operationen untereinander ist diejenige, die in Abb. 7.1(c) dargestellt wird. Dort ist die von einer Operation verwendete Variable vom Ergebnis einer Dereferenzierungsoperation abhängig. Durch diese Abhängigkeit werden bei der Analyse Querybedingungen für den Queryfluß eingeführt. In der Abbildung soll eine Query Γ_V die letzte Verwendung der Variablen v_2 auf mögliche Variablenbelegungen für diese Variable untersuchen. Welche Operation die letzte Verwendung von v_2 darstellt, hängt aber vom Ergebnis der Dereferenzierungsoperation ω' ab, die hier im Beispiel entweder die Variable v_1 oder die Variable v_2 zum Ziel hat. Für die Querybehandlung ergeben sich hier nun zwei Fälle, in Abbildung 7.1(d) und (e) schematisch dargestellt. Falls die Dereferenzierungsoperation als Ergebnis die Variable v_1 besitzt, dann ergibt sich die Situation aus Teilabbildung (d), bei der die Query die Operation ω nicht betrachten muß, da sie keinen Einfluß auf die Variable v_2 haben kann. Im anderen Fall, in dem die Dereferenzierungsoperation die Variable v_2 zum Ziel hat, ergibt sich die Situation aus Teilabbildung (e), in der die Operation ω relevant ist und untersucht werden muß. Im Rahmen der Analyse wird in solchen Situationen stets eine Fallunterscheidung nach den möglichen Fällen durchgeführt. In jedem der Fälle wird der Queryfluß anschließend durch eine Pfadbedingung für den entsprechenden Fall begrenzt, d.h. auf einen Bereich des Analysegraphen beschränkt. Auf dies wird in Abschnitt 7.3.4 noch detaillierter eingegangen werden.

Obwohl die im Folgenden gegebenen Definitionen formal auf der Beschreibung der neuen Programmrepräsentation als Sequenz von Operationen basiert, wird für die Veranschaulichung im Folgenden stets die Darstellung der Programmrepräsentation als Graphen verwendet werden. Dadurch wird eine bessere Verständlichkeit der Definitionen ermöglicht, die aber trotzdem formal fundiert bleiben. Bei der Beschreibung der Besonderheiten der Analyse in den folgenden Abschnitten wird zunächst eine informelle Beschreibung der Querybearbeitung für die verschiedenen Operationstypen mit einfließen. Eine exakte Definition folgt später in diesem Kapitel.

7.3 Teilaspekte der Querybearbeitung

7.3.1 Aufbau von Protokollautomaten bei der Bearbeitung der Queries

Wie bereits geschildert, wird der von einer Query durchlaufene Pfad durch den Aufbau eines endlichen Automaten protokolliert. Dazu wird bei der Initiierung einer Query für einen Dereferenzierungsknoten ein endlicher Automat erzeugt, der nur aus seinem Startzustand besteht und weder Zustandsübergänge, noch Endzustände besitzt. Eine Query führt nun während ihrer Verarbeitung diesen Automaten als betriebenen Automaten mit, d.h. als Automat mit einem aktuellen Zustand. Dieser betriebene Automat wird auch als Protokollautomat bezeichnet. In diesen Automaten wird eine Protokollierung des von der Query durchlaufenen Weges durch den Analysegraphen integriert. In Abbildung 7.2 ist dies beispielhaft dargestellt. Die Numerierung beschreibt die Reihenfolge, in der die folgenden Schritte unternommen werden. Der in den einzelnen Schritten von der Query mitgeführte Automat ist dabei in der Abbildung jeweils den Query–Pfeilen zugeordnet. Dabei wird der aktuelle Zustand des betriebenen Automaten durch einen ausgefüllten Kreis dargestellt.

1. Für die eingezeichnete Dereferenzierungsoperation wird eine Query generiert, die einen Automaten, der nur aus dem Startzustand besteht, als betriebenen Automaten mit sich führt. Diese Query, realisiert durch die Queryfunktion Γ_P, soll die möglichen Inhalte des benannten Wertes, der von der Dereferenzierungsoperation konsumiert wird, ermitteln. Im Beispiel wird dieser benannte Wert von der eingezeichneten get_x–Operation erzeugt, die (nach der Definition der Get–Operation) die Belegung der Variablen x als Inhalt des benannten Wertes zur Verfügung stellt.

2. Damit muß für eine Suche nach den Zielen der Dereferenzierungsoperation nun die letzte Verwendung der Variablen x betrachtet werden, was durch die Queryfunktion Γ_V geleistet wird. Dieses Vorgehen beschreibt bereits informell die Vorgehensweise, eine Query an einer

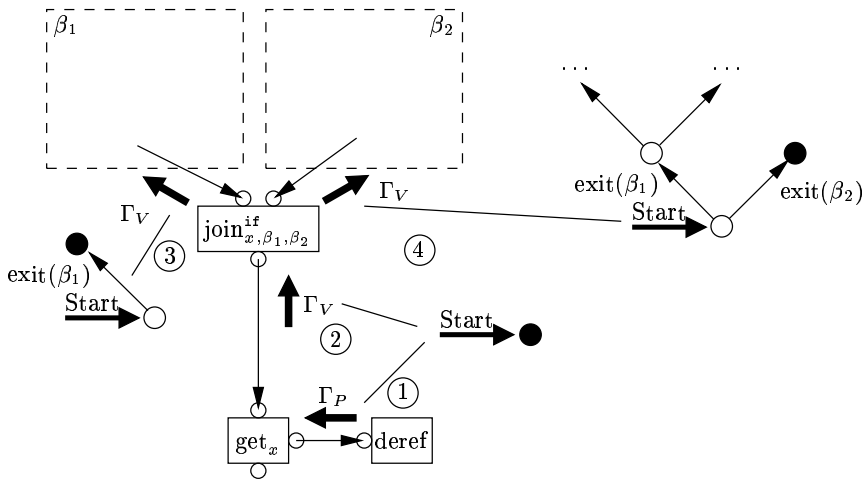


Abbildung 7.2: Erzeugung von Zustandsübergängen zur Pfadprotokollierung.

Get–Operation zu verarbeiten, die eine festgelegte Variable verwendet. Nach diesem Schritt besitzt diese Query immer noch den unveränderten betriebenen Automaten.

3. Bei möglichen Verzweigungen (Join–Operationen) wird für jeden der Zweige eine eigene Query erzeugt, die im jeweiligen Zweig weiterverarbeitet wird. Um den dabei gewählten Zweig zu protokollieren, wird für jeden der Zweige ein neuer Zustand des Automaten und jeweils ein neuer Zustandsübergang vom aktuellen Zustand des Protokollautomaten zu diesen Zuständen erzeugt. Im Beispiel wird von der Query eine $\text{join}^{\text{if}}_{\ldots}$–Operation erreicht. Dabei wird im aktuellen Schritt zunächst die Bearbeitung der Query im Operationsblock β_1 fortgesetzt. Dazu wird, wie eingezeichnet, ein neuer Zustand generiert und ein Zustandsübergang zum Protokollautomaten hinzugefügt, der bei Eingabe des entsprechenden Symbols aus E, hier also exit(β_1), in diesen neuen Zustand übergeht. Prinzipiell entspricht ein Zustand damit dem Operationsblock, in dem eine Query gerade bearbeitet wird. Mittels Tiefensuche wird nun von diesem Zustand als aktuellem Zustand des Protokollautomaten ausgehend zunächst diese Query fortgeführt, bis deren Bearbeitung komplett beendet ist.

4. Danach wird der dabei entstandene Protokollautomat, dessen Fortsetzung in der Abbildung bei Schritt 4 durch ...angedeutet ist, verwendet, um für den zweiten möglichen Zweig einen neuen Zustand und einen neuen Zustandsübergang zu erzeugen, und die Query mit diesem neuen Zustand als aktuellem Zustand im Operationsblock β_2 weiter zu bearbeiten.

Die Generierung der neuen Zustände und Zustandsübergänge, sowie das Weiterleiten der beiden Queries definiert dabei bereits informell die Bearbeitung von Queries an join–Operationen.

7.3.2 Integration von Ergebnissen in Protokollautomaten

Der Fluß von Queries wird fortgesetzt, bis entweder eine Wiederholung festgestellt werden kann (was im nächsten Abschnitt behandelt wird) oder eine Adressoperation eine definitive Aussage darüber zulässt, welche Variable das Ziel einer Dereferenzierungsoperation sein wird. Im zweiten Fall wird der aktuelle Zustand des betriebenen Automaten zu einem Endzustand gemacht. Diese Vorgehensweise wird in Abbildung 7.3 vorgestellt.

1. Wie im vorherigen Abschnitt wird hier eine Query mit einem Protokollautomaten, der nur aus seinem Startzustand besteht, generiert, und durch die get_x–Operation weitergeleitet.

2. Die Query erreicht eine join–Operation.

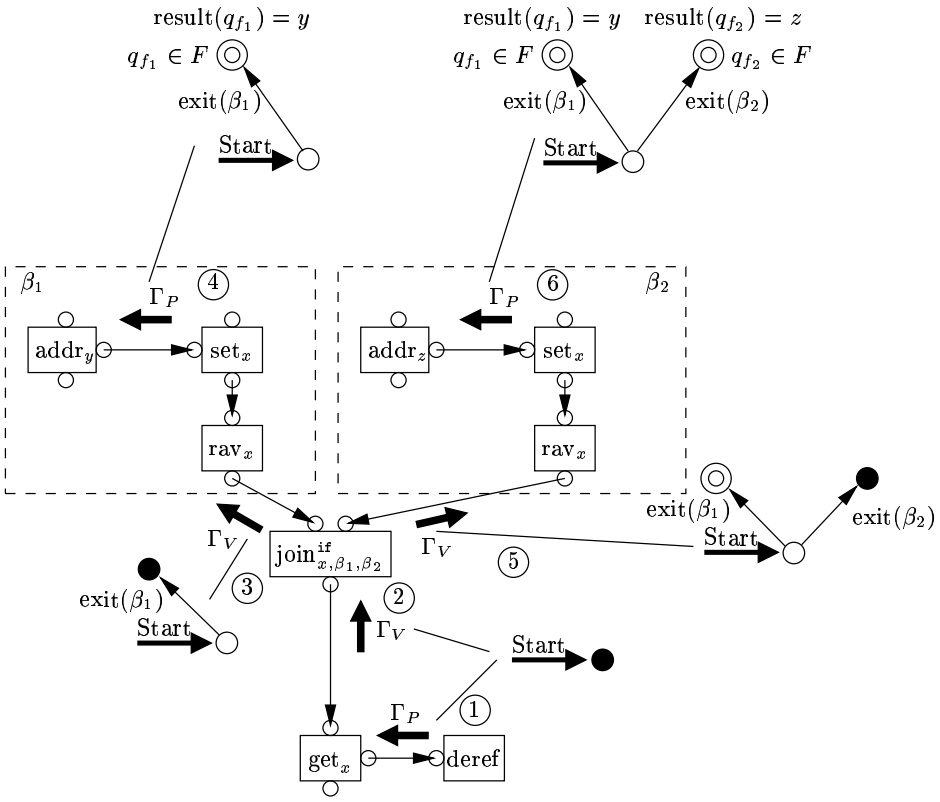


Abbildung 7.3: Erzeugung von Endzuständen zur Ergebnisbeschreibung.

3. Ebenfalls wie im letzten Abschnitt beschrieben, wird beim Eintritt der Query in den Block β_1 ein neuer Zustand und ein neuer Zustandsübergang erzeugt. Im Inneren des Blockes β_1 erreicht die Suche nach der letzten Zuweisung zur Variablen x zunächst eine rav–Operation, die keine Veränderung der Variablenbelegung bewirkt, weshalb die Query direkt weitergeleitet wird (damit ist auch die Query–Verarbeitung an rav–Operationen und analog an var–Operationen informell definiert). Nach der Weiterleitung erreicht die Query eine Set–Operation, die der Variable x einen Wert zuweist, der (nach der Definition der Set–Operation) aus einem konsumierten benannten Wert stammt. Daher wird an dieser Stelle das Ergebnis der Query Γ_V durch das Ergebnis der eingezeichneten Query Γ_P berechnet, die den konkreten Inhalt des benannten Wertes ermitteln soll. Diese Vorgehensweise definiert bereits informell die Verarbeitung einer Query an einer Set–Operation, die wie hier im Beispiel stets die gleiche Variable verwendet.

4. Die Query Γ_P trifft nun auf eine Adressoperation, die die Adresse der Variablen y in einem benannten Wert zur Verfügung stellt. Damit wird an dieser Stelle ein Ergebnis gefunden, da man für den bisherigen Weg der Query die Aussage machen kann, daß diese Adresse von y den betrachteten Dereferenzierungsknoten in Ausführungsrichtung erreichen wird, wenn der Weg der Query in Vorwärtsrichtung ausgeführt wird: die Adressoperation generiert stets einen benannten Wert mit Inhalt y, dieser wird bei der Set–Operation stets der Variablen x zugewiesen. Die rav–Operation ändert diesen Wert nicht, also besitzt die Variable x beim Erreichen der $\text{join}^{\texttt{if}}_{x,\ldots}$–Operation stets die Belegung y. An der join–Operation kann man nun die Aussage treffen, daß wann immer der Block β_1 in Vorwärtsrichtung ausgeführt wird, die Variable x danach als Belegung die Variable y besitzt. Mit dieser Information kann man nun weiter folgern, daß unter dieser Voraussetzung auch die Get–Operation stets den Wert y als von ihr produzierten benannten Wert erzeugen wird, der dann von der Dereferenzierungsoperation konsumiert wird, die als Ziel der Dereferenzierung unter der gleichen Voraussetzung, daß zuvor der Block β_1 durchlaufen wurde, stets die Variable y haben muß.

   Damit entspricht das Erreichen einer Adressoperation durch eine Γ_P–Query (also durch eine Query, die die Dereferenzierungsoperation "von rechts" erreicht) dem Finden eines Ergebnisses. Im Automaten wird dies dadurch beschrieben, daß der aktuelle Zustand des Protokollautomaten zu einem Endzustand gemacht wird, wie dies in der Abbildung in Schritt 4 gezeigt wird. Die Information, daß in diesem Zustand konkret die Adresse der Variablen y gefunden wurde, wird durch die Abbildung result : $F \to V$ beschrieben, die in dieser Arbeit jedem Endzustand eines Automaten eine Variable zuordnet. In der Abbildung wurde der Endzustand mit $q_{f_1} \in F$ bezeichnet, für den $\text{result}(q_{f_1}) = y$ gilt.

5. Bei der weiteren Verarbeitung wird in Schritt 5 eine Query in den Block β_2 geschickt, was im Protokollautomaten wie eingezeichnet vermerkt wird.

6. Diese Query findet analog zur bisherigen Schilderung ein Ergebnis in der Form einer Adressoperation addr_z. Damit wird auch dieser Zustand des betriebenen Automaten zu einem Endzustand gemacht und es entsteht der Automat, der in der Abbildung bei Schritt 6 angegeben ist, als Endergebnis der Query.

Dieses Beispiel definiert bereits informell die Queryverarbeitung an Adress–Operationen.

7.3.3 Erkennen von Wiederholungen

Das bisher beschriebene Prinzip, bei der Bearbeitung von Queries neue Zustände und Zustandsübergänge zu erzeugen, könnte an sich bei Schleifen, in denen keine Adressoperationen enthalten sind, die die Query beenden könnten, beliebig oft fortgesetzt werden. Dies entspräche damit dem klassischen Problem, das sämtliche einfachen Verfahren bei der Protokollierung von Programmpfaden haben.

Wie bereits mehrfach erwähnt, ist das in dieser Arbeit vorgestellte Verfahren in der Lage, auch diese unendlichen Pfade mit einer endlichen Beschreibung ohne Verlust von Genauigkeit zu behandeln. Dazu ist es wichtig, festzustellen, wann eine Query nur mehr das gleiche Ergebnis berechnen würde, das bereits vorher von einer Query berechnet wurde. Dazu wird gespeichert, an welcher

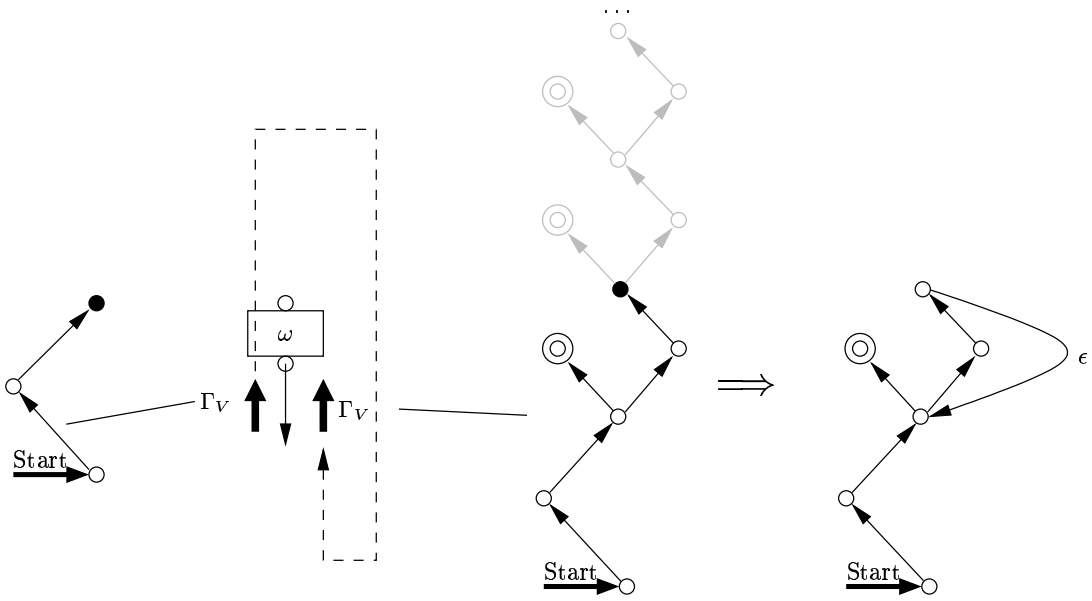
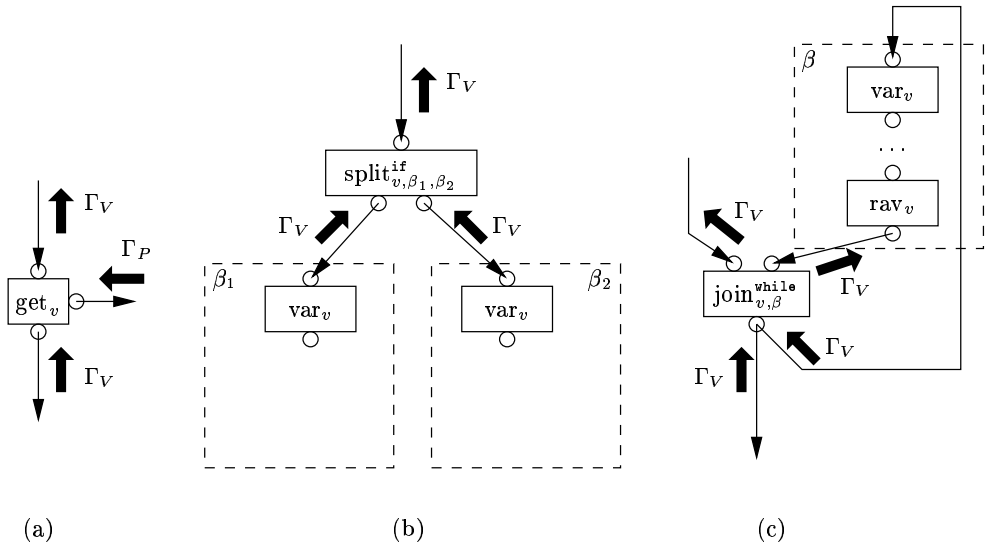


Abbildung 7.4: Erzeugung von Wiederholungen bei der Pfadprotokollierung.

Operation welche Query mit welchen Querybedingungen (im Sinne des bedingten Erreichbarkeitsproblems — auf diese wird im nächsten Abschnitt genauer eingegangen werden) bereits bearbeitet wurde, und welchen aktuellen Zustand der Protokollautomat dabei hatte. Soll nun an einer Operation erneut eine Query für die gleiche Dereferenzierungsoperation bearbeitet werden, so werden zunächst die Querybedingungen der aktuellen und der vorherigen Queries verglichen. Falls die aktuelle Query die gleichen Querybedingungen zu berücksichtigen hat, wie eine vorherige, dann würde eine Weiterberechnung der aktuellen Query nur exakt das gleiche Ergebnis liefern, das diese andere Query bereits berechnet hat. Damit reicht es aus, dieses bereits berechnete Ergebnis wiederzuverwenden. Da die Ergebnisse durch die berechneten Protokollautomaten vollständig beschrieben werden, erreicht man dies, indem man vom aktuellen Zustand des Protokollautomaten der Query einen Zustandsübergang ohne Eingabesymbol, also einen ϵ–Übergang, zu demjenigen Zustand generiert, den die vorher bearbeitete Query an dieser Stelle als aktuellen Zustand hatte. Damit ist auch vom aktuellen Zustand aus das Ergebnis der vorherigen Berechnung durch Zustandsübergänge zu erreichen. Diese Vorgehensweise wird in Abbildung 7.4 veranschaulicht. Dabei soll an einer Operation ω zunächst eine Query mit dem links davon abgebildeten betriebenen Automaten bearbeitet werden. Die Query wird danach fortgesetzt und erreicht erneut die gleiche Operation, diesmal mit dem betriebenen Automaten, der rechts abgebildet ist. Offensichtlich wurde seit dem letzten Betrachten der Query eine Verzweigung untersucht, in der ein Ergebnis gefunden wurde, was durch den Endzustand gekennzeichnet ist, und über eine andere Verzweigung die gleiche Operation erneut erreicht. Da die aktuelle und die frühere Query im hier angenommenen Beispiel die gleiche Ausgangssituation haben, d.h. keine Querybedingungen zu berücksichtigen haben, würde bei einer weiteren Berechnung der Query immer wieder das gleiche Ergebnis berechnet werden, was in der Abbildung durch die grau gezeichneten Abschnitte des Automaten ausgedrückt werden soll. Die vorher beschriebene Vorgehensweise sieht dagegen vor, wie ganz rechts in der Abbildung eingezeichnet einen ϵ–Zustandsübergang vom aktuellen Zustand des Protokollautomaten zu dem Zustand, der bei der zuletzt bearbeiteten Query der aktuelle Zustand war, zu erzeugen. Der Automat beschreibt damit die Möglichkeit der beliebig häufigen Wiederholung dieser Schleife mit einer endlichen Beschreibung.

Der Test auf Wiederholungen wird nur an denjenigen Operationen durchgeführt, an denen Queries über verschiedene Graphkanten zusammentreffen können. Für alle anderen Operationen, an denen eine Wiederholung festgestellt werden kann, muß diese Wiederholung auch bereits vorher



(a) (b) (c)

Abbildung 7.5: Operationen, an denen eine Überprüfung auf Wiederholungen von Queries notwendig ist.

erkennbar gewesen sein, wenn sämtliche Queries mit ihren Querybedingungen vorher an einer gemeinsamen anderen Operation verarbeitet wurden. In Abbildung 7.5 sind diejenigen Operationen dargestellt, an denen Wiederholungsprüfungen vorgenommen werden müssen, da sich dort die Ergebnisse von verschiedenen Queries aus der gleichen Weiterleitung der Query im Programmgraphen ergeben. In Abbildung 7.5(a) ist eine Get–Operation mit den möglichen sie erreichenden Queries dargestellt. Sowohl die Query nach der letzten Zuweisung zur Variablen v als auch die Query nach dem Inhalt des von der Get–Operation produzierten benannten Wertes werden durch die Suche nach der wiederum letzten Zuweisung an die Variable v berechnet. Daher ist die Get–Operation eine Operation, an der auf Wiederholungen getestet werden muß, da sie einen Zusammenflußpunkt von Queries über verschiedene Graphkanten darstellt und beide durch die gleiche weitergeleitete Query berechnet. Die gleiche Situation tritt bei $\text{split}^{\texttt{if}}_{\ldots}$–Operationen, in Abbildung 7.5(b) dargestellt, auf, wo Queries aus beiden Unterblöcken durch die gleiche Weiterleitung der Queries berechnet werden. Bei $\text{join}^{\texttt{while}}_{\ldots}$–Operationen, wie in Abbildung 7.5(c) dargestellt, läßt sich die gleiche Situation beim Zusammenfluß der Queries aus dem Schleifenrumpf und von außerhalb der Schleife erkennen.

7.3.4 Hinzufügen und Verarbeiten von Querybedingungen

Wenn Queries von den Ergebnissen von anderen Queries abhängen, dann kann man die Analyse, wie bereits in Abschnitt 3.3 geschildert wurde, als ein bedingtes Erreichbarkeitsproblem formulieren. Die Abhängigkeit von solchen Ergebnissen wird dabei ebenfalls im Protokollautomaten protokolliert. Die dazu verwendeten Eingabesymbole an den Automaten entstammen der Menge E, die in Abschnitt 6.4.3.2.1 definiert wurden. So soll das in E enthaltene Symbol $[\omega \rightarrow y]$ beispielsweise ausdrücken, daß die Dereferenzierungsoperation ω als Ziel der Dereferenzierung $\Psi(\omega)$ die Variable y ergibt. Aus der Automatensichtweise bedeutet also ein Zustandsübergang $\delta(q, [\omega \rightarrow y]) = q^{'}$ mit $q, q^{'} \in Q$ und $[\omega \rightarrow y] \in E$, daß bei der Bearbeitung der Query, deren Protokollautomat sich zunächst im Zustand q befindet und dann unter Eingabe dieses Symbols in den Zustand $q^{'}$ übergeht, nun als Voraussetzung davon ausgegangen wird, daß die Dereferenzierungsoperation die Variable y als Ziel besitzt. Diese Art von Eigenschaft wurde in Abschnitt 5.2.1 als Programmeigenschaft bezeichnet. Da Queries i.A. unter der Annahme der Gültigkeit verschiedener solcher Programmeigenschaften verschiedene Ergebnisse liefern, dienen Zustandsübergänge mit solchen Eingabesymbolen dazu, für die Queries, die von verschiedenen Annahmen ausgehen, separate Zustandsmengen und Ergebnisse

zu erzeugen.

Die Annahme der Gültigkeit einer solchen Programmeigenschaft hat eine Beschränkung des weiteren Queryflusses zur Folge. Geht man z.B. von der Gültigkeit der Programmeigenschaft $[\omega \rightarrow y]$ aus, und hat bereits die möglichen Ziele der Dereferenzierungsoperation ω in der Form eines Automaten $M^\omega = (Q, E, \delta, q_0, F, \text{result})$ berechnet, der eine Pfadbedingung gemäß Definition 5.37 der Form

$$M^\omega \Longrightarrow \omega \rightarrow y$$

darstellt, so beschreibt dieser Automat die Menge aller Pfade, deren Durchlaufen dazu führt, daß ω genau das Ziel y besitzt. Bezeichne im Folgenden wie in Definition 5.38

$$M_y^\omega = (Q, E, \delta, q_0, \text{result}^{-1}(y), \text{result}).$$

Um nun den weiteren Fluß der Query, bei deren Bearbeitung man von dieser Annahme ausgeht, auf diese Menge von Pfaden zu beschränken, wird von der Query für jede angenommene Programmeigenschaft ein betriebener Automat $(M_y^\omega, q_0) = ((Q, E, \delta, q_0, \text{result}^{-1}(y), \text{result}), q_0)$ mitgeführt. Der aktuelle Zustand dieses betriebenen Automaten wird dabei auf seinen Startzustand gesetzt. Ansonsten ist dieser Automat bis auf seine Endzustandsmenge identisch zum Ergebnisautomaten M^ω. Die Endzustandsmenge F wurde zu $\text{result}^{-1}(y)$ gemäß obiger Definition reduziert.

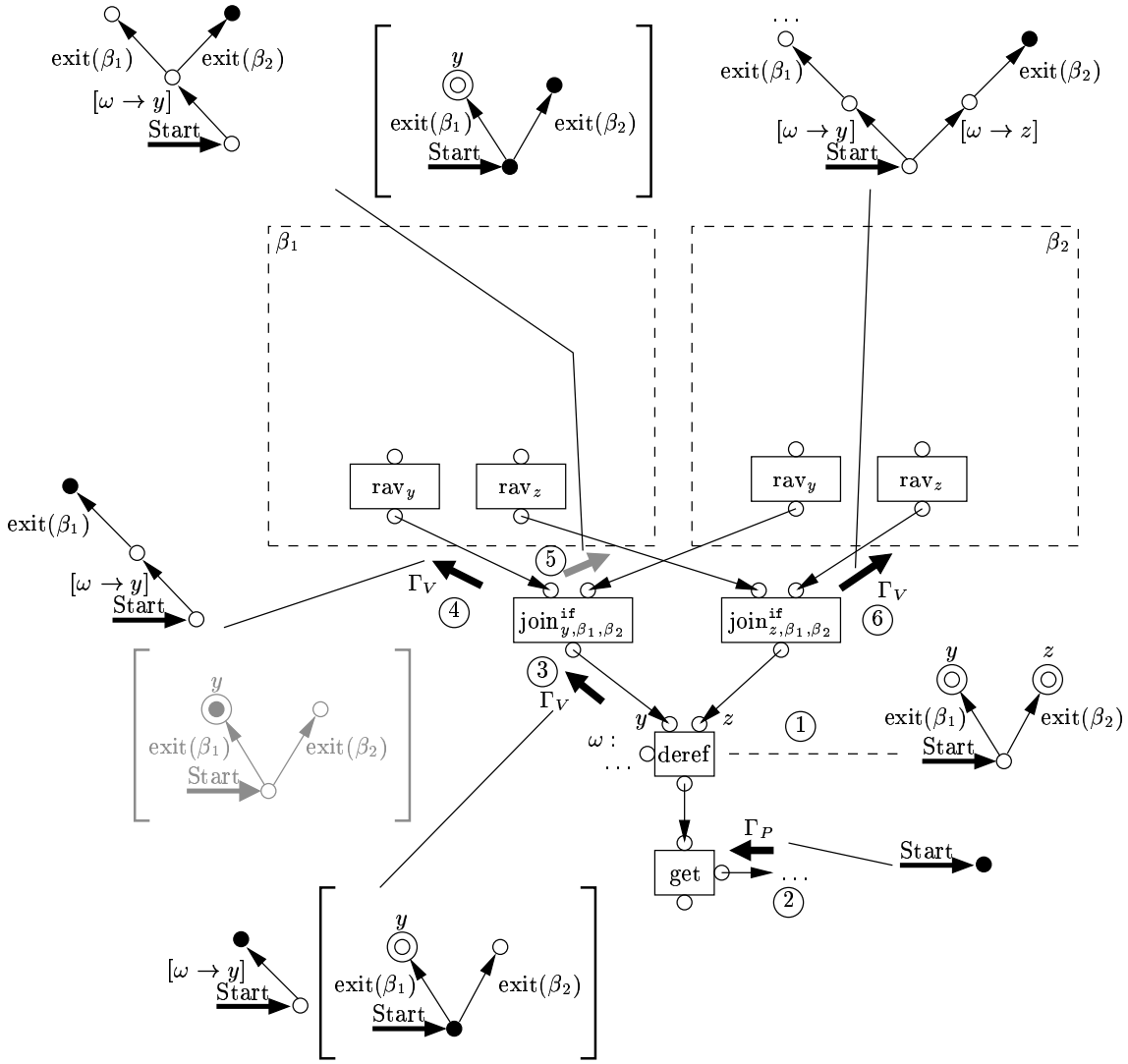
Durch das Mitführen eines oder mehrerer betriebenen Automaten, die als Querybedingungen bezeichnet werden, soll der weitere Query–Fluß in einer Weise eingeschränkt werden, daß nur solche Pfade von Queries durchlaufen werden, die den betriebenen Automaten in einen seiner Endzustände überführen. Wenn die Query unter dieser Einschränkung ein Ergebnis findet, so kann damit garantiert werden, daß dieses Ergebnis gemeinsam mit den Programmeigenschaften, von deren Gültigkeit dabei als Annahme ausgegangen wurde, auf einem möglichen Programmpfad und damit in einer Programminstanz auftreten können. Das Hinzufügen von weiteren Automaten beim Antreffen von Programmeigenschaften repräsentierenden Eingabesymbolen erfolgt offenbar identisch zu den Bedingungsautomaten aus Abschnitt 5.7.

An jeder möglichen Verzweigung für eine Query wird überprüft, ob diese Verzweigung mit sämtlichen, Querybedingungen repräsentierenden Automaten vereinbar ist. Damit dies der Fall sein kann, müssen zunächst die betriebenen Automaten, die als Querybedingungen mitgeführt werden, von ihrem aktuellen Zustand aus einen Zustandsübergang mit dem die Verzweigung charakterisierenden Symbol als Eingabe durchführen können, also z.B. mit dem Eingabesymbol $\text{exit}(\beta_1)$. Zusätzlich muß von dem dadurch erreichten Zustand noch eine Möglichkeit gegeben sein, die jeweiligen Endzustände der betriebenen Automaten zu erreichen. Ansonsten ist die Auswahl dieser Verzweigung mit den Querybedingungen nicht vereinbar und die Query muß mit diesen Querybedingungen nach dieser Verzweigung nicht weiter bearbeitet werden, da es dann keine Programminstanz geben kann, die die aktuell von der Query betrachtete Operation erreicht, in der sämtliche bisher geforderten Programmeigenschaften gleichzeitig gelten können.

In Abbildung 7.6 ist dies beispielhaft vorgestellt. Zu jedem Schritt wird der Protokollautomat und ggfs. ein als Querybedingung mitgeführter weiterer betriebener Automat, in eckigen Klammern, angegeben. Die in der Zeichnung dargestellten Schritte sind dabei wie folgt:

1. Für die Dereferenzierungsoperation ω wird angenommen, daß bereits das daneben eingezeichnete Ergebnis berechnet wurde. Dabei entspricht dieses Ergebnis demjenigen aus dem Beispiel in Abbildung 7.3. Demnach ist das Ziel der Dereferenzierungsoperation ω die Variable y, wenn von einer Programminstanz zuvor der Block β_1 durchlaufen wurde, und die Variable z, wenn zuvor der Block β_2 durchlaufen wurde.

2. Im Beispiel wird angenommen, daß eine Query Γ_P mit einem Protokollautomaten, der daneben dargestellt ist und nur aus seinem Anfangszustand besteht, die eingezeichnete Get–Operation erreicht. Diese Get–Operation liest stets die Belegung derjenigen Variablen aus, die als Ergebnis der Dereferenzierungsoperation ω durch $\Psi(\omega)$ ausgedrückt wird. Nachdem laut Schritt 1 für die Operation ω bereits bekannt ist, welche Variablen bei Durchlaufen welcher Pfade Ziel der Dereferenzierungsoperation sind, kann die Get–Operation nur den Wert der Variablen y oder



Ergebnis:

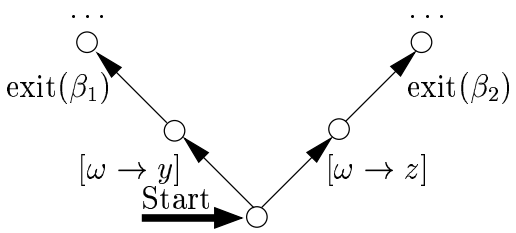


Abbildung 7.6: Hinzufügen und Bearbeiten von Querybedingungen.

z auslesen. Damit ergibt sich das Ergebnis der Query Γ_P entweder aus der letzten Zuweisung zu y oder der letzten Zuweisung zu z.

3. Für die weitere Bearbeitung der Query wird zunächst von der Annahme ausgegangen, daß die Dereferenzierungsoperation ω die Variable y zum Ziel hat. Damit muß mittels der eingezeichneten Queryfunktion Γ_V die letzte Zuweisung zur Variable y gefunden werden. Im zugehörigen Protokollautomaten wird diese Annahme dokumentiert. In der Abbildung wurde dazu vom Startzustand ein Zustandsübergang mit dem Eingabesymbol $[\omega \rightarrow y]$ hinzugefügt, der die angenommene Programmeigenschaft ausdrückt. In den eckigen Klammern ist der ab diesem Zeitpunkt als Querybedingung mitgeführte betriebene Automat dargestellt, der sich aus dem Ergebnisautomat der Dereferenzierungsoperation ω, wie unter Punkt 1 beschrieben, sowie derjenigen Teilmenge von Endzuständen, für die die Anwendung der result–Funktion die Variable y ergibt, zusammensetzt.

4. An der eingezeichneten join–Operation ergeben sich zwei Möglichkeiten für die Fortsetzung der Querybearbeitung: in den Block β_1 und in den Block β_2. Die Fortsetzung der Bearbeitung im Block β_1 ist mit der mitgeführten Querybedingung vereinbar. Bei Eintritt in den Block wird für den als Querybedingung mitgeführten betriebenen Automat ein Zustandsübergang mit dem die Verzweigung charakterisierenden Eingabesymbol exit(β_1) durchgeführt. Nach diesem Zustandsübergang ist der Endzustand immer noch erreichbar. Konkret wird dieser Endzustand bei Eingabe dieses Symbols sogar erreicht. Damit kann das Mitführen der Querybedingung (grau eingezeichnet) an dieser Stelle beendet werden, weil bereits sichergestellt ist, daß auf dem bisherigen Weg der Query in jedem Fall die Gültigkeit der Programmeigenschaft $[\omega \rightarrow y]$ garantiert werden kann.

5. Bei der Überprüfung, ob die aktuelle Query im Block β_2 fortgesetzt werden muß, läßt sich erkennen, daß dies widersprüchlich zu der mitgeführten Querybedingung ist. Durch Eingabe des Symbols exit(β_2) an diesen Automaten würde dieser in einen Zustand übergehen, von dem aus der Endzustand nicht mehr erreichbar wäre. Daher muß die Query im Block β_2 nicht fortgesetzt werden, was in der Abbildung durch einen grau eingezeichneten Query–Pfeil dargestellt ist. Konkret würde in einer Programminstanz, die den Block β_2 durchläuft, die Dereferenzierungsoperation ω als Ziel die Variable z besitzen, was zur hier zunächst angenommenen Programmeigenschaft, daß ω die Variable y zum Ziel haben soll, widersprüchlich wäre.

6. Unter der anderen Voraussetzung, daß die Dereferenzierungsoperation die Variable z zum Ziel hat, ergibt sich die analoge Situation zum letzten Schritt, wobei hier der Block β_2 von der Query betreten wird, während das Betreten von Block β_1 hier widersprüchlich zur mitgeführten Querybedingung ist.

Unterhalb der Abbildung ist der insgesamt entstandene betriebene Automat dargestellt. Vom Startzustand gehen zunächst Zustandsübergänge aus, die mit den Symbolen $[\omega \rightarrow y]$ und $[\omega \rightarrow z]$ beschriftet sind. Von jedem der dabei erreichbaren Zustände gehen danach Zustandsübergänge aus, die das Betreten der mit den Querybedingungen jeweils vereinbarbaren Blöcke bezeichnen.

7.3.5 Nicht komplett abgearbeitete Querybedingungen

Wie im vorherigen Abschnitt geschildert wurde, wird bei Abhängigkeiten von Queries von den Ergebnissen von anderen Queries ein betriebener Automat als Querybedingung zur Query hinzugefügt. Im Beispiel wurde dieser während der Bearbeitung der Query in einen Endzustand überführt, woraufhin die Querybedingung im weiteren nicht mehr betrachtet werden musste. Allgemein kann es jedoch vorkommen, daß eine Query ein Ergebnis findet und unter Generierung eines Endzustandes die Bearbeitung der Query beendet, bevor alle Querybedingungen abgearbeitet wurden. Ein Beispiel hierfür findet sich in Abbildung 7.7. Dort wird genau wie im vorherigen Beispiel in Schritt 1 von einem bekannten Ergebnis der Analyse für die Dereferenzierungsoperation ω ausgegangen, und in Schritt 2 eine Query angenommen, die von diesen Ergebnissen abhängt. In Schritt 3 schließlich,

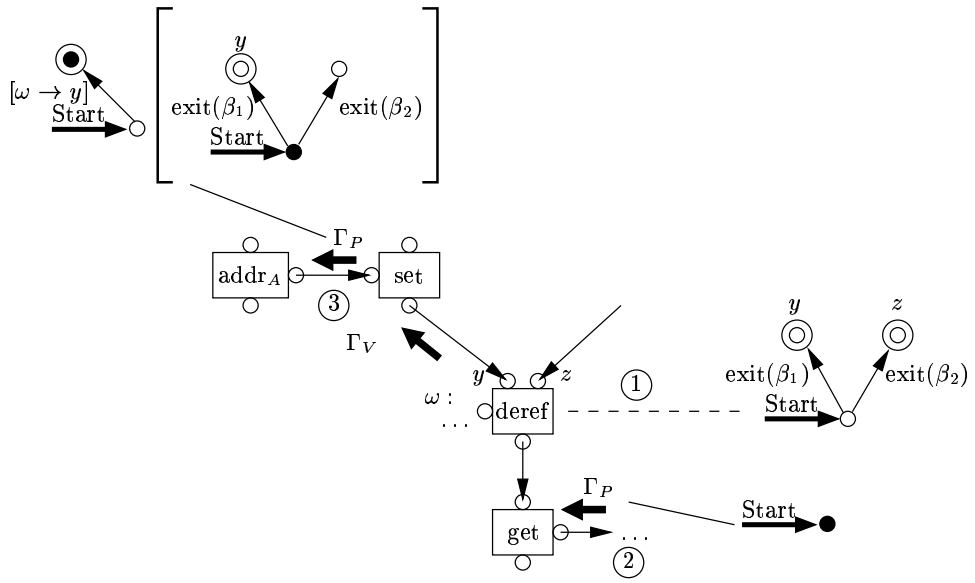


Abbildung 7.7: Bei Ergebnissen verbliebene Querybedingungen.

nachdem wie im vorherigen Beispiel die angenommene Programmeigenschaft in den Protokollautomaten integriert wurde und ein betriebener Automat als Querybedingung hinzugefügt wurde, wird vor Erreichen einer Verzweigungsmöglichkeit ein Ergebnis für die Query gefunden, da der Variablen y dort stets die Adresse von A zugewiesen wird. Damit wird wie bisher der aktuelle Zustand des betriebenen Automaten zum Endzustand gemacht, wie dies in der Abbildung oberhalb des Graphen dargestellt ist.

Die noch verbliebene Querybedingung muß aber weiterhin berücksichtigt werden. Obwohl die Variable y an dieser Stelle stets die Adresse von A als Belegung besitzt, ist die Belegung von y für die Berechnung der Query Γ_P aus Schritt 2 nur dann relevant, wenn die Dereferenzierungsoperation diese Variable y zum Ziel hat. Dies ist aber nur dann der Fall, wenn die restliche Querybedingung ebenfalls erfüllt ist, d.h. zuvor der Block β_1 durchlaufen wurde. Aus diesem Szenario lassen sich die folgenden notwendigen Vorgehensweisen erkennen:

- Bei der Terminierung einer Query kann ein Ergebnis nur dann gefunden werden, wenn die zum Zeitpunkt der Terminierung der Query verbliebenen Querybedingungen widerspruchsfrei sind, d.h. mindestens ein Pfad existiert, der alle Automaten der Querybedingungen in einen Endzustand überführt. Ansonsten könnte es keine Programminstanz geben, in der alle geforderten Programmeigenschaften gleichzeitig gültig sind. In diesem Fall würde die Query beendet werden, ohne einen Endzustand zu erzeugen.

- Die bei der Terminierung einer Query übriggebliebenen (widerspruchsfreien) Querybedingungen werden den entsprechenden Endzuständen zugeordnet und dadurch gespeichert.

- Wann immer ein als Querybedingung mitgeführter betriebener Automat einen Endzustand q_f erreicht, dem übriggebliebene betriebene Automaten zugeordnet sind, werden diese betriebenen Automaten als neue zusätzliche Querybedingungen der Query hinzugefügt. Damit werden diese übrig gebliebenen Querybedingungen von anderen Queries, die von diesen Ergebnissen abhängen, weiter verarbeitet und berücksichtigt.

7.3.6 Behandlung von unterschiedlichen Eingabesymbolen bei Querybedingungen

Bei der Erzeugung von neuen Zuständen des betriebenen Automaten an Verzweigungen werden, wie bereits geschildert, die als Querybedingungen mitgeführten betriebenen Automaten auf eine Vereinbarkeit der Verzweigung mit der Querybedingung getestet, und ggfs. auf diesen betriebenen Automaten ebenfalls Zustandsübergänge durchgeführt. Dies entspricht dem gemeinsamen Betreiben einer Menge von Automaten durch Bedingungsautomaten. Die von einer Query erreichten Verzweigungen müssen aber nicht unbedingt zu den nächstmöglichen Eingabesymbolen für die als Querybedingungen mitgeführten betriebenen Automaten passend sein. Da nicht alle Variablen in allen Unterblöcken auftreten, kann es vorkommen, daß eine Querybedingung Aussagen über mögliche Pfade in einem Programmblock macht, während die eigentliche Query diesen Block nicht untersucht, da die Variable, deren letzte Zuweisung gefunden werden soll, in diesem Block nicht vorkommt. Ebenso können die als Querybedingungen mitgeführten betriebenen Automaten Symbole der Form $[\omega \rightarrow v]$ enthalten, von denen die eigentliche Query nicht direkt abhängt. Damit können verschiedene Protokollautomaten jeweils Eingabesymbole auslassen, die für die Menge der von ihnen induzierten Programminstanzen unrelevant sind. In den Begriffen aus Kapitel 5 bedeutet dies, daß die Protokollautomaten Automaten sind, die Wortteilmengen induzieren. Anders ausgedrückt ist jedes Wort, das von einem solchen Automaten akzeptiert wird, ein korrekt geschachteltes Teilwort in Bezug auf die Menge aller Programminstanzen, die die Operation erreichen, an der die betrachtete Programmeigenschaft gilt.

Ein Beispiel hierfür ist in Abbildung 7.8 dargestellt. Dort ist eine Situation gezeigt, in der unten im Bild eine Query Γ_P mit dem dort abgebildeten Protokollautomaten bei der Dereferenzierungsoperation ω_3 den Inhalt des benannten Wertes, der sich aus dem Auslesen des Ziels der Dereferenzierung der Variablen w durch die Dereferenzierungsoperation ω_3 ergibt, ermitteln soll. Dabei sind in der Abbildung nur diejenigen Operationen dargestellt, die für dieses Ergebnis relevant sind. Eine vollständige Graphrepräsentation würde zusätzliche var– und rav–Operationen enthalten, die hier zur Verbesserung der Übersichtlichkeit weggelassen wurden.

Im Beispiel wird zur Vereinfachung der Argumentation angenommen, daß die Dereferenzierung der Belegung der Variablen w jeweils die Variable v ergibt. Ein Teil eines möglichen Automaten M^{ω_3}, der dieses Ergebnis repräsentiert, ist unter dem Programmgraphen dargestellt. Der Start einer Query Γ_P an der Dereferenzierungsoperation ω_3 würde nach den bisher vorgestellten Vorgehensweisen diesen Protokollautomaten ergeben. Darin sind diejenigen Zustandsübergänge, die aus Programmabschnitten resultieren, die vor dem dargestellten Ausschnitt angenommen werden, mit ... angedeutet. Die letzte Zuweisung an die Variable $w \in V$ kann laut diesem Ergebnisautomaten je nach durchlaufenem Programmpfad im Block β_3 oder β_4 geschehen und sich dort abhängig vom Ergebnis der Dereferenzierungsoperationen ω_1 und ω_2 jeweils aus dem Inhalt der Variablen x, y oder z ergeben. Dies wird durch die verschiedenen Verzweigungen im das Ergebnis darstellenden Automaten ausgedrückt, in dem Zustandsübergänge mit den Symbolen $\text{exit}(\beta_3)$ und $\text{exit}(\beta_4)$, sowie den verschiedenen Symbolen $[\omega_1 \rightarrow \ldots]$ bzw. $[\omega_2 \rightarrow \ldots]$ vorkommen. Die Variablen x und z kommen in den Blöcken β_1 und β_2 vor, so daß hierfür ebenfalls Zustandsübergänge im Automaten existieren. Die Variable y kommt hingegen in diesen Blöcken nicht vor, so daß der nächste Zustandsübergang in diesem Teil des Automaten andere Eingabesymbole hat (durch ... angedeutet), die im Beispiel weiter oben im Programmgraphen zu finden wären.

Die Fortsetzung der unten eingezeichneten Query Γ_P an der Get–Operation ergibt sich damit im weiteren aus der eingezeichneten Query Γ_V, die unter Mitführen des Ergebnisautomaten M^{ω_3} der Dereferenzierungsoperation ω_3 als Querybedingung in Form eines betriebenen Automaten in seinem Startzustand (wie unten in Abbildung 7.8 eingezeichnet), die letzte Zuweisung zur Variablen v sucht. Da die Variable v in den Blöcken β_3 und β_4 nicht vorkommt, wird die Query an diesen Blöcken vorbeigeleitet. An der $\text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}$–Operation muß nun der als Querybedingung mitgeführte Automat auf eine Vereinbarkeit des Betretens der Blöcke β_1 und β_2 mit der Querybedingung überprüft werden. Die nächsten Eingabesymbole dieses Automaten lassen hierüber jedoch keine Aufschlüsse zu.

Prinzipiell könnten in dieser Situation zwei Fälle eintreten. Die nächsten Symbole, die die als Querybedingung mitgeführten Automaten akzeptieren, können zu Verzweigungen gehören, die im Programm noch oberhalb der Blöcke β_1 und β_2 zu finden sind. In diesem Fall kann die Query-

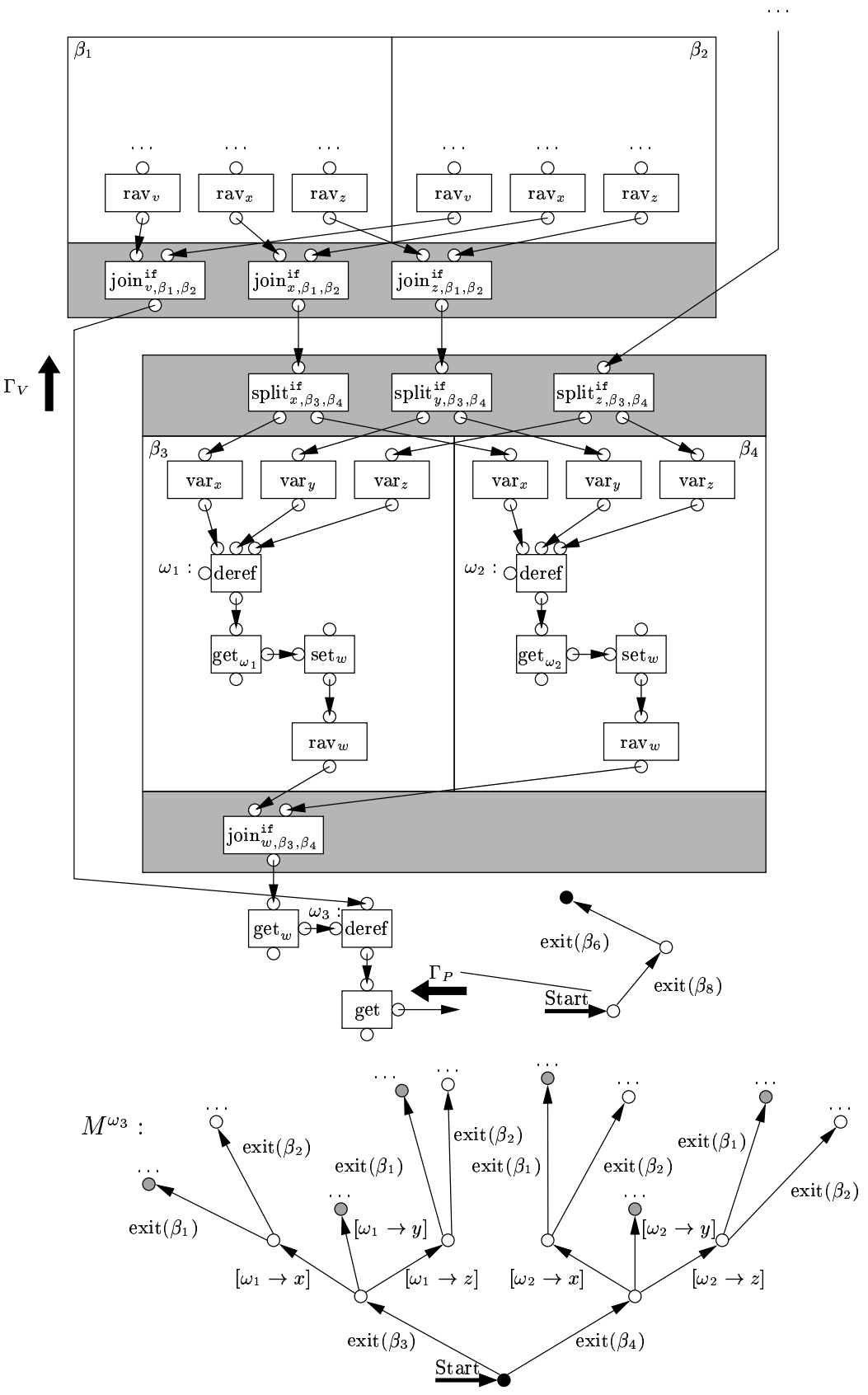


Abbildung 7.8: Verschiedene Eingabesymbole für Protokollautomat und Querybedingungen.

bedingung unverändert mitgeführt werden, um erst an dieser anderen Stelle im Programm erneut betrachtet zu werden.

Die Situation des zweiten Falles ergibt sich hier im Beispiel. Bevor die Eingabesymbole exit(β_1) bzw. exit(β_2) verarbeitet werden können, müssen erst die zusätzlichen Eingabesymbole der als Querybedingung mitgeführten Automaten berücksichtigt, und die entsprechenden Zustandsübergänge der Querybedingung(en) durchgeführt werden. Nach diesen Zustandsübergängen sind alle in Abbildung 7.8 (unten) grau eingezeichneten Zustände korrekte Möglichkeiten dafür, in welchen aktuellen Zuständen sich der als Querybedingung mitgeführte betriebene Automat befinden kann, wenn die Query z.B. den Block β_1 betritt.

Die Unterscheidung zwischen den beiden Fällen läßt sich anhand der in Abschnitt 5.5.1 eingeführten Ordnung $\leq_{\widehat{E}}$ auf Eingabesymbolen treffen. Das bezüglich dieser Ordnung maximale Eingabesymbol ist dasjenige, das als nächstes berücksichtigt werden muß. Das entspricht der Vorgehensweise in den Definitionen 5.33, 5.36 und 5.40, jeweils aus den nächstmöglichen Eingabesymbolen von zwei oder mehr Automaten ein bzgl. dieser Ordnung maximales Element als nächstes Eingabesymbol auszuwählen.

Zur anschaulichen Interpretation dieser Vorgehensweise im vorliegenden Fall ist es zunächst hilfreich, die verwendeten Automaten als Beschreibungen von Programminstanzenmengen zu betrachten. Für den Inhalt der Variablen v an der Dereferenzierungsoperation ω_3 ist es unrelevant, ob eine Programminstanz vor Erreichen der Dereferenzierungsoperation den Block β_3 oder β_4 betreten hat und welche Zeigerziele die Dereferenzierungsoperationen ω_1 und ω_2 haben. Der Automat, der bei einer reinen Betrachtung von v entstünde, besäße als nächste Eingabesymbole exit(β_1) bzw. exit(β_2) und wäre damit, wie in Abschnitt 5.1.2 informell vorgestellt, eine zusammenfassende Beschreibung einer Menge von möglichen Programminstanzen, würde also wieder einer Klasse von Programminstanzen entsprechen. Anders ausgedrückt würde der Automat eine Sprache akzeptieren, die jeweils korrekt geschachtelte Teilworte in Bezug auf die Menge aller Programminstanzen, die die Operation ω_3 erreichen, darstellen.

Durch die Abhängigkeit der Queryverarbeitung von der Dereferenzierungsoperation ω_3 haben aber i.A. nicht mehr alle Programminstanzen aus dieser Klasse die gleiche Auswirkung. Man könnte dies im Rahmen der bisher in der Arbeit vorgestellten Veranschaulichungen als zusätzliche "Forderungen" an die Menge von Programminstanzen interpretieren. Neben der reinen Betrachtung der Forderungen, die sich aus der Suche nach der letzten Zuweisung zu v ergeben, müssen auch noch die Forderungen in Bezug auf die Dereferenzierungsoperation ω_3 erfüllt werden. Entsprechend muß die Pfadbeschreibung an dieser Stelle verfeinert werden, was praktisch durch die Integration von Zustandsübergängen aus den als Querybedingungen mitgeführten Automaten in den Protokollautomaten geleistet wird. Bei Bedingungsautomaten beobachtet man diesen Effekt dann, wenn zunächst die weiteren mitgeführten Automaten Zustandsübergänge durchführen, bevor der im Startzustand verwendete Automat wieder einen Zustandsübergang durchführt. Abbildung 7.9 zeigt das Ergebnis der Integration der Symbole, mit denen die als Querybedingungen mitgeführten Automaten als nächstes einen Zustandsübergang durchführen können, in den Protokollautomaten. Führte die Query Γ_P bei Erreichen der Get–Operation unten in Abbildung 7.8 den danebenstehenden Automaten als Pfadprotokollierung mit, so wird dieser bei Erreichen der Join–Operation wie in Abbildung 7.9 gezeigt erweitert. Die grau eingezeichneten Zustände sind dabei diejenigen Zustände, von denen ab die Query bei Eintritt in den Block β_1 ihre Pfadprotokollierung fortsetzen wird. Für jeden dieser Zustände wird eine eigene Query erzeugt, die als Querybedingung den betriebenen Automaten (wie in Abbildung 7.8 unten abgebildet) mitführt, der sich im zum jeweiligen Zustand des Protokollautomat korrespondierenden aktuellen Zustand befindet. Jede solche Query wird anschließend in den Block β_1 weitergeleitet. Analog wird für den Eintritt in den Block β_2 verfahren.

Die Anwendung der Ordnung auf Eingabesymbolen, mit der die Gesamtreihenfolge der nächsten möglichen Eingabesymbole der Query und der mitgeführten Querybedingungen bestimmt wird, ist für das Beispiel in Abbildung 7.10 dargestellt. Dort ist im linken Teil der Protokollautomat der aktuellen Query Γ_V abgebildet, der als nächstes mögliches Eingabesymbol durch die Ankunft an der entsprechenden $\text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}$–Operation das Symbol exit(β_1) für den nächsten zu generierenden Zustandsübergang verwenden würde, sofern keine Querybedingungen vorhanden wären. Im rechten

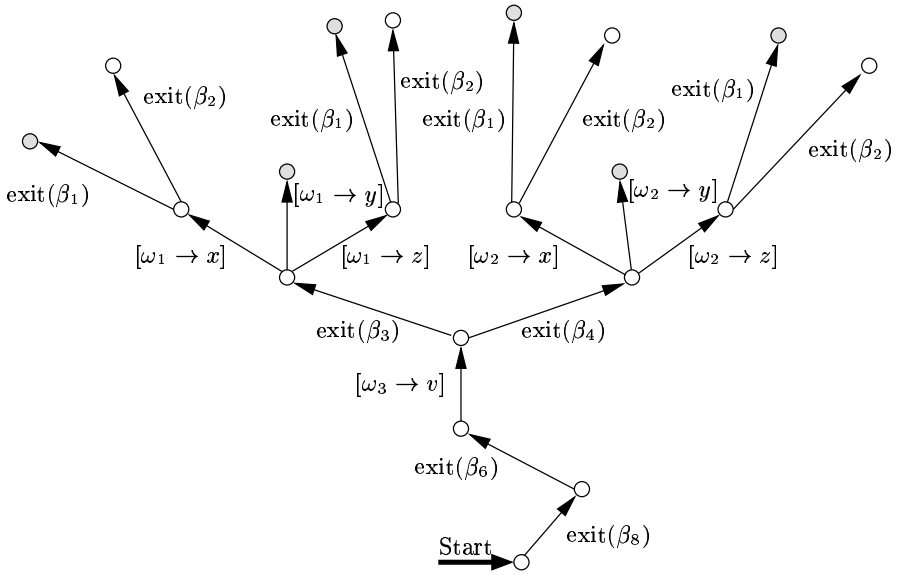


Abbildung 7.9: Ergebnis der Integration von Teilen des als Querybedingung mitgeführten betriebenen Automaten in den Automaten der Query.

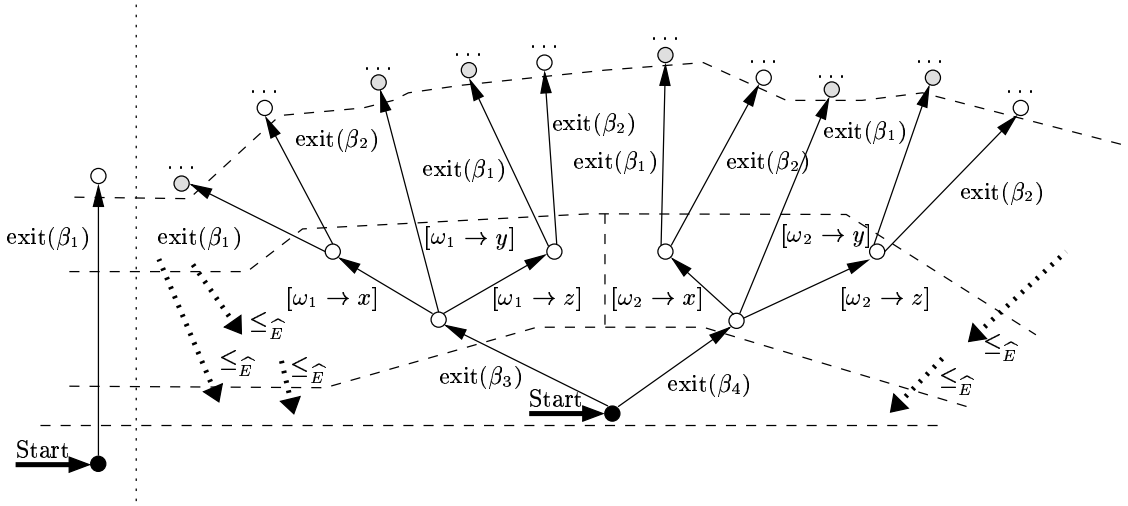


Abbildung 7.10: Anwendung der Ordnung von Eingabesymbolen zur Integration von Zustandsübergängen von Querybedingungen in den Protokollautomaten einer Query.

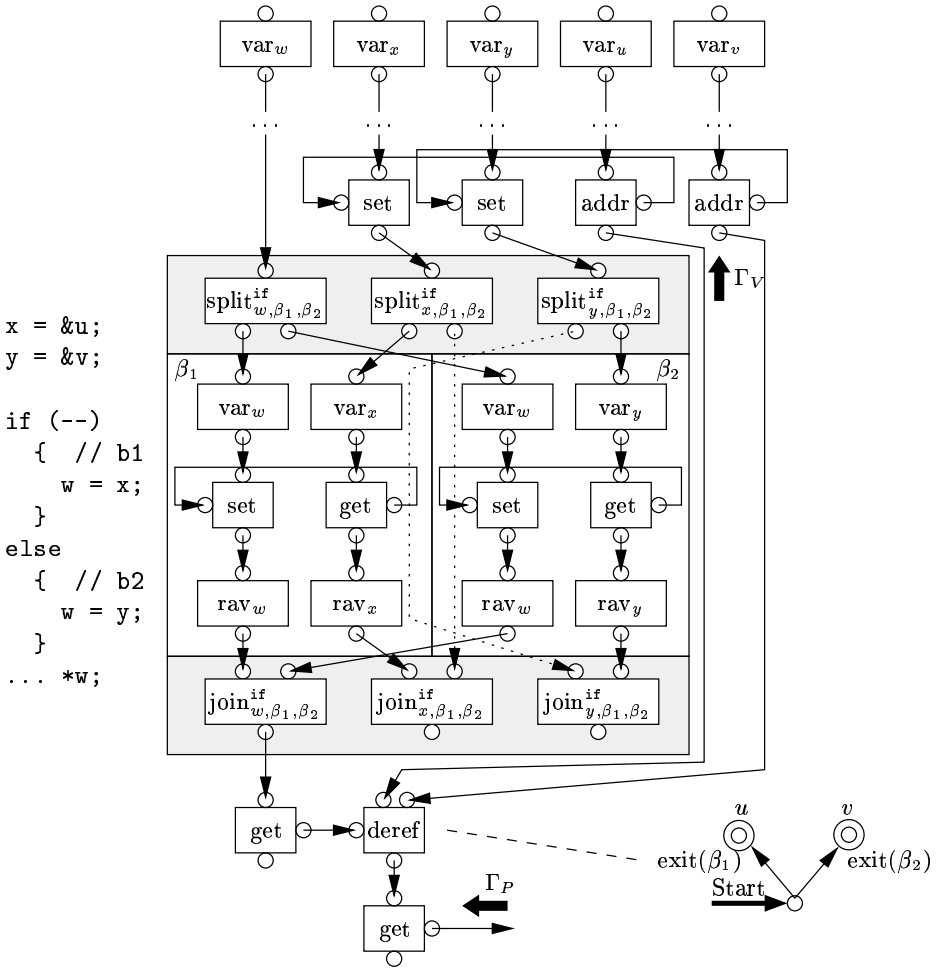


Abbildung 7.11: Integration von Pfaden in Querybedingungen in den betriebenen Automaten einer Query.

Teil der Abbildung ist der Automat aus dem Beispiel aus Abbildung 7.8 nochmals dargestellt, diesmal mit einer Einteilung der Eingabesymbole in Zonen, die jeweils alle Eingabesymbole e umfassen, die bzgl. Definition 5.26 in der gleichen Menge $\langle e \rangle$ enthalten sind. Die Ordnung, die zwischen den Symbolen in verschiedenen Zonen gilt, ist durch die gestrichelten Pfeile eingezeichnet. Die Zonen, zwischen denen keine solchen Pfeile eingezeichnet sind, sind bzgl. dieser Ordnung unvergleichbar. Die Information aus dieser Ordnung reicht aber aus, um im Beispiel zu erkennen, daß sowohl die Symbole exit(β_3) bzw. exit(β_4) als auch sämtliche Symbole der Form $[\omega_{\ldots} \to \ldots]$ vor Verarbeitung der Symbole exit(β_1) bzw. exit(β_2) in den Protokollautomaten integriert werden müssen, da erstere in dieser Ordnung jeweils größer als letztere sind.

Ein weiteres Beispiel, bei dem sichtbar wird, daß die von den Querybedingungen geforderten Pfade auch dann für die Query relevant sind, selbst wenn deren betrachtete Variable in den Blöcken entlang dieser Pfade nicht vorkommt, ist in Abbildung 7.11 dargestellt. Dabei ist das Beispiel–Programmfragment im linken Teil der Abbildung in die graphbasierte Form der neuen Programmrepräsentation transformiert worden. Wie im letzten Beispiel betritt die rechts oben eingezeichnete Query Γ_V, die sich als Weiterleitung der unten eingezeichneten Query Γ_P ergibt, die Subblöcke β_1 und β_2 nicht. In diesen Subblöcken werden aber Zuweisungen ausgeführt, die für die Annahme, daß die unten eingezeichnete Dereferenzierungsoperation das Ziel u bzw. v besitzt, notwendige Voraus-

setzungen sind. Auch in diesem Fall werden die Zustandsübergänge der Querybedingungen in den betriebenen Automaten der Query integriert. Ebenso wie im vorhergehenden Beispiel in diesem Abschnitt wird z.B. an der nächsten von der Query erreichten join–Operation festgestellt werden, daß das Eingabesymbol der Querybedingung größer als das zur dortigen Operation zugehörige Symbol ist, und entsprechende Erweiterungen am betriebenen Automaten vornehmen.

7.4 Formale Definition der Queryverarbeitung

7.4.1 Funktionalität der Queries

Die Beschreibung der Bearbeitung von Queries wird im Folgenden durch zwei rekursive Funktionen Γ_V und Γ_P geleistet werden, durch die das Ergebnis einer Query definiert wird. Dazu soll zunächst die folgende abkürzende Definition eingeführt werden.

Definition 7.1 (Queryspeicherfunktionen)
Die Menge $\mathcal{K}$ sei wie folgt als Menge von Queryspeicherfunktionen *definiert.*

$$\mathcal{K} = \{\kappa \mid \kappa : (\Omega \times \Omega_0 \times (V \cup \emptyset) \times (P \cup \emptyset) \times \mathcal{P}(\mathcal{M}) \to \mathcal{M})\}$$

Damit wird mit $\mathcal{K}$ die Menge aller Funktionen bezeichnet, die Tupel bestehend aus einer Operation $\omega \in \Omega$, einer Grundoperation $\omega \in \Omega_0$, einer Variable $v \in V$, einer Zeichenkette $p \in P$ und einer Menge von betriebenen Automaten $\{(M_1, q_1), \ldots, (M_n, q_n)\} \in \mathcal{P}(\mathcal{M})$ mit $n \in \mathbb{N}_0$ auf einen betriebenen Automaten $(M', q') \in \mathcal{M}$ abbildet. Solche Funktionen werden dazu verwendet werden, um an Operationen bereits bearbeitete Queries zu speichern, und mit dieser Information mögliche Wiederholungen in der Analyse zu erkennen und in der Automatensichtweise zu beschreiben. Damit kann man nun die rekursiven Queryfunktionen wie folgt spezifizieren, die genaue Definition wird im Folgenden für die verschiedenen Operationstypen $\omega \in \Omega$ getrennt gegeben werden.

Definition 7.2 (Queryfunktionen)
Funktionen Γ_P und Γ_V der folgenden Form

$$\Gamma_V : \left\{ \begin{array}{rcl} \Omega \times \Omega_0 \times V \times \mathcal{M} \times \mathcal{P}(\mathcal{M}) \times \mathcal{K} & \to & \mathcal{M} \times \mathcal{K} \\ (\omega, \omega_{\text{deref}}, v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa) & = & ((M', q'), \kappa') \end{array} \right.$$

sowie

$$\Gamma_P : \left\{ \begin{array}{rcl} \Omega \times \Omega_0 \times P \times \mathcal{M} \times \mathcal{P}(\mathcal{M}) \times \mathcal{K} & \to & \mathcal{M} \times \mathcal{K} \\ (\omega, \omega_{\text{deref}}, p, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa) & = & ((M', q'), \kappa') \end{array} \right.$$

seien als Queryfunktionen *definiert.*

Die beiden Funktionen Γ_V und Γ_P unterscheiden sich in ihrem Definitionsbereich nur im dritten Parameter, der bei Γ_V eine Variable und bei Γ_P eine Zeichenkette ist. Im ersten Fall werden eine Operation $\omega \in \Omega$, eine weitere Operation $\omega_{\text{deref}} \in \Omega_0$, die eine Dereferenzierungsoperation sein soll, eine Variable $v \in V$, ein betriebener Automat $(M, q) \in \mathcal{M}$ und eine Menge von betriebenen Automaten $\{(M_1, q_1), \ldots, (M_n, q_n)\} \in \mathcal{P}(\mathcal{M})$, sowie eine Funktion $\kappa \in \mathcal{K}$ auf ein Paar aus einem betriebenen Automaten $(M', q') \in \mathcal{M}$ und einer Funktion $\kappa' \in \mathcal{K}$ abgebildet. Im zweiten Fall wird anstatt der Variablen v eine Zeichenkette p übergeben. Anschaulich soll die Funktion Γ_V die bereits beschriebene Suche nach der letzten Zuweisung zu einer Variablen realisieren und die Funktion Γ_P den Inhalt s eines konsumierten benannten Wertes herausfinden. Dabei wird ein betriebener Automat (M, q) als Protokollautomat übergeben, der während der Querybearbeitung erweitert wird, um die bei der Analyse durchlaufenen Pfade zu beschreiben. Die Menge von betriebenen Automaten $\{(M_1, q_1), \ldots, (M_n, q_n)\}$ beschreibt eine Menge von Querybedingungen, die den Fluß der Query im Sinne eines bedingten Erreichbarkeitsproblemes beschränken.

Ein zusätzlicher Parameter ist dabei die Queryspeicherfunktion $\kappa \in \mathcal{K}$ gemäß Definition 7.1, die während der Bearbeitung der Query protokolliert, mit welchen als Querybedingung mitgeführten

betriebenen Automaten an einer Operation $\omega \in \Omega$ eine Query für das Ergebnis der Dereferenzierungsoperation $\omega_{\text{deref}} \in \Omega_0$ auf der Suche nach der letzten Verwendung einer Variablen $v \in V$ oder der Erzeugung eines benannten Wertes mit Namen $p \in P$ bereits bearbeitet wurde. Sofern diese Funktion für bestimmte Parameter definiert ist, ergibt sie als Ergebnis einen betriebenen Automaten (M', q'). Zum Zeitpunkt der letzten Bearbeitung einer Query befand sich deren Protokollautomat M' im Zustand q'. Falls die aktuell bearbeitete Query die gleichen Querybedingungen wie die vorher bearbeitete Query hat, würde eine Weiterführung der Query nur das zur früheren Berechnung identische Ergebnis liefern. Da dies bereits bekannt ist, kann man dieses Ergebnis in den von der aktuellen Query berechneten Teil des betriebenen Automaten (M, q) integrieren, indem man eine Transition vom Zustand q zum Zustand q' erzeugt und diese mit dem leeren Eingabesymbol ϵ versieht (wie bereits informell in Abschnitt 7.3.3 beschrieben wurde). Bei der Berechnung dieser Query–Funktionen Γ_P und Γ_V wird neben dem betriebenen Automaten $(M', q') \in \mathcal{M}$, der die Auswirkungen der Querybearbeitung auf den Protokollautomaten $(M, q) \in \mathcal{M}$ enthält, auch eine Funktion $\kappa' \in \mathcal{K}$ zurückgegeben, die zusätzlich den aktuellen Zustand und die aktuell vorhandenen Querybedingungen bei der Bearbeitung der Query an der aktuell betrachteten Operation $\omega \in \Omega$, sowie allen weiteren von diesem rekursiven Zweig der Query betrachteten Operationen registriert.

7.4.2 Definitionen zur Behandlung von Querybedingungen

In den Abschnitten 7.3.5 und 7.3.6 wurden Situationen vorgestellt, in denen betriebene Automaten bei der Terminierung von Queries verblieben sind, bzw. in denen Zustandsübergänge aus den betriebenen Automaten in den Protokollautomaten integriert werden müssen. Die Vorgehensweise in diesen Situationen wird in diesem Abschnitt formal definiert werden.

7.4.2.1 Endliche Automaten mit bedingten Endzuständen

Zur Behandlung der ersten dieser Situationen werden endliche Automaten um sogenannte bedingte Endzustände erweitert.

Definition 7.3 (Automat mit bedingen Endzuständen)
Ein Automat

$$M = (Q, E, \delta, q_0, F, \text{result}, \text{conditions})$$

mit Q, E, δ, q_0, F *und* result *wie in Definition 5.9 spezifiziert und*

$$\text{conditions} : F \to \mathcal{P}(\mathcal{M})$$

einer Abbildung von der Menge der Endzustände F *des Automaten in die Potenzmenge aller möglichen betriebenen Automaten werde als* endlicher deterministischer Automat mit bedingten Endzuständen *bezeichnet. Im Weiteren soll hierfür ebenfalls die abkürzende Bezeichnung Automat verwendet werden, wenn aus dem Kontext hervorgeht, um welche Art von Automaten es sich handelt. Ein Endzustand* $q_f \in F$ *mit* $\text{conditions}(q_f) \neq \emptyset$ *werde als* bedingter Endzustand *bezeichnet.*

Um die Sprache von Automaten mit bedingten Endzuständen zu definieren, bietet es sich an, zunächst folgende vereinfachende Definitionen zum gleichzeitigen Betreiben von mehreren Automaten mit bedingten Endzuständen einzuführen.

7.4.2.2 Ermitteln des nächsten Eingabesymbols an eine Menge von Automaten

Dazu wird zunächst für eine Menge von betriebenen Automaten mit bedingten Endzuständen eine Funktion definiert, die für diese Menge eine Klasse $\langle e \rangle$ von Eingabesymbolen liefert, deren darin enthaltenen Symbole bzgl. der vorgestellten Ordnung auf Eingabesymbolen unter denjenigen Symbolen, mit denen die betriebenen Automaten jeweils als nächstes einen Zustandsübergang durchführen können, maximal sind.

Definition 7.4 (Nächstes Eingabesymbol einer Menge von betriebenen Automaten) *Sei* $\{(M_1, q_1), \ldots, (M_n, q_n)\}$ *mit* $n \in \mathbb{N}$ *eine nicht–leere Menge von betriebenen Automaten mit bedingten Endzuständen. Die Funktion* $\text{next} : \mathcal{P}(\mathcal{M}) \to \widehat{E}$ *sei für diese wie folgt definiert:*

$$\begin{aligned} &\text{next}(\{(M_1, q_1), \ldots, (M_n, q_n)\}) \\ &= \max\{\langle e\rangle \in \widehat{E} \mid \exists i \in \{1, \ldots, n\} \ : \ \delta_i(q_i, \langle e\rangle)\}, \end{aligned}$$

wobei δ_i *die Zustandsübergangsfunktion des Automaten* M_i *bezeichnen soll.*

Daß dieses Maximum stets existiert, liegt daran, daß jeder dieser mitgeführten Automaten Worte akzeptiert, die jeweils korrekt geschachtelte Teilworte in Bezug auf die Menge aller Programminstanzen darstellen. Darauf soll im Rahmen der Korrektheitsbetrachtung der Queryverarbeitung im nächsten Kapitel noch eingegangen werden.

7.4.2.3 Gemeinsames Betreiben einer Menge von Automaten

Die folgenden Definitionen sollen das gemeinsame Betreiben von als Querybedingung mitgeführten betriebenen Automaten einführen. Dazu wird zunächst eine Funktion $\widehat{\delta}$ wie folgt definiert.

Definition 7.5 (Zustandsübergänge eines einzelnen betriebenen Automaten) *Die Funktion* $\widehat{\delta} : \mathcal{M} \times E \to \mathcal{P}(\mathcal{M})$ *sei wie folgt definiert.*

$$\widehat{\delta}(((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), e)$$

$$= \begin{cases} \{((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q')\}, & \text{falls } \delta(q, e) = q' \in Q, \text{ und ein} \\ & \text{Endzustand } q_f \in F \text{ ist von } q' \text{ erreichbar} \\ \{((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q)\}, & \text{falls } \delta(q, e) \text{ ist nicht definiert, aber } \exists e' \in E \\ & \text{mit } \delta(q, e') \in Q \text{ und } \langle e\rangle \neq \langle e'\rangle, \\ \text{conditions}(q_f), & \text{falls } \delta(q, e) = q_f \in F, \\ \text{undef}, & \text{sonst.} \end{cases}$$

Diese Funktion $\widehat{\delta}$ gibt für einen betriebenen Automaten und ein Eingabesymbol $e \in E$ eine Menge von betriebenen Automaten zurück, die entweder den betriebenen Automaten in einem neuen Zustand nach Eingabe des Symbols, den betriebenen Automaten im gleichen Zustand wie vorher, falls er nur Symbole mit anderem "origin" verarbeiten kann, die leere Menge, falls der betriebene Automat durch den Zustandsübergang in einen Endzustand $q_f \in F$ übergeht, der kein bedingter Endzustand ist, und die Menge $\text{conditions}(q_f)$ von betriebenen Automaten, falls es sich dabei um einen bedingten Endzustand handelt, beschreiben. Dabei werden die letzten beiden Fälle bzgl. bedingten Endzuständen im dritten Fall der Definition zusammengefasst, da die Funktion conditions auf einen Endzustand $q_f \in F$ angewandt die leere Menge ergibt, falls es sich nicht um einen bedingten Endzustand handelt.

Diese Definition ist weitgehend identisch zur Definition von $\overline{\delta}$ bzw. $\overline{\delta_a}$ in Definition 5.40. Als Unterschiede kann man die Behandlung von bedingten Endzuständen in Definition 7.5 erkennen. Weiter werden in Definition 7.5 keine weiteren betriebenen Automaten als Folge von bestimmten Eingabesymbolen zur weiteren Bearbeitung hinzugefügt, wie dies in Definition 5.40 durch die Funktion $\overline{\delta_b}$ der Fall ist.

Damit kann man nun eine Zustandsübergangsfunktion für eine Menge von betriebenen Automaten wie folgt definieren.

Definition 7.6 (Zustandsübergangsfunktion für eine Menge von betriebenen Automaten)

Die Funktion $\delta^* : \mathcal{P}(\mathcal{M}) \times E \to \mathcal{P}(\mathcal{M})$ *sei wie folgt definiert.*

$$\delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},e) \quad = \quad \begin{cases} \bigcup_{i\in\{1,\ldots,n\}} \widehat{\delta}((M_i,q_i),e), \\ \qquad \textit{falls alle } \widehat{\delta}((M_i,q_i),e) \textit{ für } i \in \{1,\ldots,n\} \\ \qquad \textit{definiert sind, und } n = 0 \textit{ oder} \\ \qquad e \in \text{next}(\{(M_1,q_1),\ldots,(M_n,q_n)\}) \textit{ gilt,} \\ \text{undef}, \\ \qquad \textit{sonst} \end{cases}$$

Wie üblich soll auch δ^* *analog zu Definition 5.10 auf Eingabewörter anwendbar sein.*

Diese Funktion δ^* ordnet einer Menge $\{(M_1,q_1),\ldots,(M_n,q_n)\}$ für $n \in N_0$ von betriebenen Automaten und einem Eingabesymbol $e \in E$ eine Menge von betriebenen Automaten zu, die alle diesen Zustandsübergang gemäß obiger Definition der Funktion $\widehat{\delta}$ durchgeführt haben. Für die leere Menge von betriebenen Automaten als Eingabe ergibt sich dabei für jedes Eingabesymbol auch wieder die leere Menge von betriebenen Automaten als Ergebnis von δ^*.

Mit diesen Definitionen kann man nun also für eine nicht–leere Menge von als Querybedingungen mitgeführten betriebenen Automaten $\{(M_1,q_1),\ldots,(M_n,q_n)\}$ zunächst eine Symbolmenge $\langle e\rangle = \text{next}(\{(M_1,q_1),\ \ldots,\ (M_n,q_n)\}) \in \widehat{E}$ bestimmen, die unter allen Symbolen, die von den betriebenen Automaten in ihren aktuellen Zuständen für Zustandsübergänge verwendet werden können, maximal bzgl. der Ordnung $\leq_{\widehat{E}}$ auf diesen Symbolen sind. Als nächster Schritt kann dann für die Symbole $e \in \langle e\rangle$ die Zustandsübergangsfunktion $\delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},e)$ angewendet werden, um eine Menge von betriebenen Automaten zu erhalten, die alle diesen Zustandsübergang ausgeführt haben, sofern dies von ihrem aktuellen Zustand aus möglich war.

7.4.2.4 Sprache eines Automaten mit bedingten Endzuständen

Die Sprache eines Automaten mit bedingten Endzuständen kann man nun wie folgt definieren.

Definition 7.7 (Sprache eines Automaten mit bedingten Endzuständen)
Für einen Automat $M = (Q, E, \delta, q_0, F, \text{result}, \text{conditions})$ *mit bedingten Endzuständen gemäß Definition 7.3 sei die von ihm akzeptierte Sprache* $\mathcal{L}(M)$ *wie folgt definiert:*

$$\mathcal{L}(M) := \{\pi \in E^* \mid \pi = \pi_1;\pi_2 \wedge \delta(q_0,\pi_1) = q_f \in F \wedge \delta^*(\text{conditions}(q_f),\pi_2) = \emptyset\}$$

Bei Verwendung von Automaten mit bedingten Endzuständen in den Definitionen aus Kapitel 5 soll diese Definition der Sprache eines Automaten anstatt der dort in Definition 5.11 angegebenen verwendet werden.

Ein solches Wort aus der Sprache eines Automaten mit bedingten Endzuständen besteht aus zwei Teilworten. Das erste Teilwort π_1 überführt den Automaten M in einen Endzustand. Das zweite Teilwort (das auch das leere Wort sein kann) überführt die Menge von betriebenen Automaten, die diesem Endzustand durch die Funktion conditions zugeordnet wurden, ebenfalls in ihre Endzustände—in Definition 7.6 der Zustandsübergangsfunktion δ^* werden Automaten, die ihre (nicht–bedingten) Endzustände erreichen, aus der Menge der betriebenen Automaten entfernt. Diese Vorgehensweise ist offensichtlich eine Mischform aus dem Betreiben eines einzelnen Automaten gemäß Definition 5.9 bis zum Erreichen eines (bedingten) Endzustandes, und dem gemeinsamen Betreiben mehrerer Automaten wie bei Bedingungsautomaten gemäß Definition 5.40 nach diesem Zeitpunkt.

Bedingte Endzustände treten in dieser Form in den Bedingungsautomaten aus Abschnitt 5.7 selbst nicht auf. Bedingungsautomaten werden weiter betrieben, selbst wenn der als Startzustand verwendete erste betriebene Automat in einen Endzustand übergeht. Die Erzeugung eines bedingten Endzustandes entspricht offensichtlich genau diesem Abarbeitungszeitpunkt des Bedingungsautomaten. Die "Restwörter" π_2 aus obiger Aufzählung entsprechen daher Eingabewörtern an den Bedingungsautomaten nach diesem Zeitpunkt.

7.4.2.5 Algorithmus zur Integration von Eingabesymbolen aus betriebenen Automaten in den Protokollautomaten

Die Integration von Eingabesymbolen aus Querybedingungen erfolgt durch den folgenden Algorithmus.

Definition 7.8 (Integration von Eingabesymbolen von Querybedingungen)

Eingabe:

- *Eine Operation* $\omega \in \Omega$*, an der die nächste Veränderung des Protokollautomaten der Query vorgenommen werden soll bzw. kann.*
- *Der Protokollautomat* $(M, q) = (Q, E, \delta, q_0, F, \text{result}, \text{conditions}) \in \mathcal{M}$ *der Query.*
- *Eine Menge von als Querybedingungen mitgeführten betriebenen Automaten* $\{(M_1, q_1), \ldots, (M_n, q_n)\}$ *mit* $n \in \mathbb{N}_0$.
- *Ein boolescher Parameter* incCurrent*, der angibt, ob auch die mit der aktuellen Operation* ω *assoziierten Eingabesymbole in den Protokollautomaten integriert werden sollen.*

Ausgabe:

- *Ein Automat* $M' = (Q', E, \delta', q_0', F', \text{result}', \text{conditions}')$*, der aus dem Protokollautomaten* M *durch die Integration der Eingabesymbole aus den Querybedingungen entstanden ist.*
- *Eine Menge* ConditionStates $\subseteq \mathcal{P}(Q' \times \mathcal{P}(\mathcal{M}))$*, die eine Menge von Tupeln aus Zuständen des Automaten* M' *und einer Menge von betriebenen Automaten, die Querybedingungen repräsentieren, besteht. Mit diesem Ergebnis kann die Query mit jedem Zustand dieser Menge als aktuellem Zustand des Protokollautomaten, und mit der jeweiligen Menge von betriebenen Automaten als Querybedingungen an der Operation* ω *fortgesetzt werden.*

Algorithmus:

```
function IntegrateConditions(ω, ((Q, E, δ, q_0, F, result, conditions), q),
                                {(M_1, q_1), ..., (M_n, q_n)}, incCurrent)
begin
    ConditionStates := {(q, {(M_1, q_1), ..., (M_n, q_n)})};
    ProcessedStates := ∅;
    Quit := False;
    while ((Quit == False)
        ∧(∃l ∈ ℕ ∃(q', {(M'_1, q'_1), ..., (M'_l, q'_l)}) ∈ ConditionStates : {(M'_1, q'_1), ..., (M'_l, q'_l)} ≠ ∅))
    begin
        ⟨e⟩ := max_{(q',{(M'_1,q'_1),...,(M'_l,q'_l)})∈ConditionStates, l∈ℕ} next({(M'_1, q'_1), ..., (M'_l, q'_l)}) ;
        if ((incCurrent == False ∧ origin(⟨e⟩) > ω) ∨ (incCurrent == True ∧ origin(⟨e⟩) ≥ ω))
        then
        begin
            Choose (q', {(M'_1, q'_1), ..., (M'_l, q'_l)}) ∈ ConditionStates, l ∈ ℕ
                with next((q', {(M'_1, q'_1), ..., (M'_l, q'_l)})) = ⟨e⟩;
            ConditionStates := ConditionStates \ {(q', {(M'_1, q'_1), ..., (M'_l, q'_l)})};
            ProcessedStates := ProcessedStates ∪ {(q', {(M'_1, q'_1), ..., (M'_l, q'_l)})};
            for e' ∈ ⟨e⟩ do
            begin
                if (δ*({(M'_1, q'_1), ..., (M'_l, q'_l)}, e') is defined)
                then
```

```
                begin
                    {(M''_1, q''_1), ..., (M''_m, q''_m)} := δ*({(M'_1, q'_1), ..., (M'_l, q'_l)}, e'), m ∈ ℕ_0;
                    if (∃(q''', {(M'''_1, q'''_1), ..., (M'''_o, q'''_o)}) ∈ ProcessedStates, o ∈ ℕ_0
                        with {(M'''_1, q'''_1), ..., (M'''_o, q'''_o)} = {(M''_1, q''_1), ..., (M''_m, q''_m)})
                    then
                    begin
                        δ := δ[q --e'--> q'''];
                    end
                    else
                    begin
                        Let q_new ∉ Q be a new state;
                        Q := Q ∪ q_new;
                        δ := δ[q --e'--> q_new];
                        ConditionStates := ConditionStates ∪ {(q_new, {(M''_1, q''_1), ..., (M''_m, q''_m)})};
                    end
                end
            end
        end
        else
            Quit := True;
    end
    result := ((Q, E, δ, q_0, F, result, conditions), ConditionStates);
end
```

Dieser Algorithmus wird in der nachfolgenden Definition der Queryfunktionalität an allen Operationen eingesetzt, bei deren Erreichen durch eine Query der betriebene Automat verändert werden kann. Damit kann gewährleistet werden, daß das Einfügen von solchen Zustandsübergängen in den Protokollautomaten in einer korrekten Reihenfolge von den richtigen Zuständen im Protokollautomaten aus erfolgt. In denjenigen Fällen, in denen keine Zustandsübergänge in den Protokollautomaten zu integrieren sind, reduziert sich der Algorithmus auf einen einzigen Vergleich von Eingabesymbolen durch die Ordnung $\leq_{\widehat{E}}$. Ebenso besteht der Algorithmus aus nur einem Vergleich, wenn keine betriebenen Automaten mitgeführt werden.

Die Definitionen ähneln im Prinzip denen von Bedingungsautomaten in Abschnitt 5.7. Auf Gemeinsamkeiten und Unterschiede der Definitionen wird beim Beweis der Korrektheit der Queryberechnung in Kapitel 8 eingegangen werden, wo die in diesem Abschnitt vorgestellte Vorgehenweise auf Bedingungsautomaten zurückgeführt werden wird.

7.4.3 Der Analyse–Algorithmus

Mit diesen Queryfunktionen, die erst noch im Detail für die verschiedenen Operationen definiert werden müssen, läßt sich der prinzipielle Analysealgorithmus für Programme aus der betrachteten Programmklasse beschreiben.

Definition 7.9 (Menge v. Dereferenzierungsoperationen mit Referenzierungsstufe i)
Für $i \in \mathbb{N}_0$ bezeichne $\Omega^{(i)}_{\text{deref}} \subseteq \Omega$ mit

$$\Omega^{(i)}_{\text{deref}} := \{\text{deref}_{\omega,p} \in \Omega_P \mid \text{deref\_level}(\text{deref}_{\omega,p}) = i\}$$

die Menge aller Dereferenzierungsoperationen mit Referenzierungsstufe i gemäß Definition 6.16, d.h. der Dereferenzierungsoperationen, die Variablen $v \in V_i$ als (mögliche) Zeigerziele besitzen. Die Menge aller Dereferenzierungsoperationen sei definiert als

$$\Omega_{\text{deref}} := \bigcup_{i \in \mathbb{N}_0} \Omega^{(i)}_{\text{deref}}$$

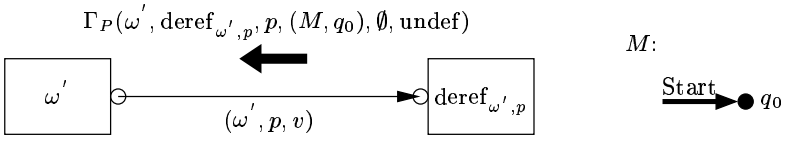


Abbildung 7.12: Initiierung des Query–Algorithmus an Dereferenzierungsoperationen.

Definition 7.10 (Query–Algorithmus)
Der grundlegende Queryalgorithmus sei wie folgt definiert.

Eingabe: *Ein Eingabeprogramm, dargestellt durch die neue Programmrepräsentation in dieser Arbeit, als Sequenz von Operationen.*

Ausgabe: *Ein Automat mit bedingten Endzuständen* $M^\omega = (Q, E, \delta, q_0, F, \text{result}, \text{conditions})$ *zu jeder Dereferenzierungsoperation* $\omega = \text{deref}_{\omega',p}$, *mit der Eigenschaft* $M^\omega \overset{r.a.}{\Longrightarrow} (\omega \rightarrow \ldots)_\omega$, *d.h. daß die von Queries berechneten Automaten widerspruchsfreie Pfadbedingungen für das Ziel der Dereferenzierungsoperation* ω *darstellen.*

Algorithmus:

```
N := max{j ∈ ℕ0 | Ω(j)deref ≠ ∅}
for i := N to 0 do
begin
    for all ω = deref_{ω',p} ∈ Ω(i)deref do
        begin
            M := ({q0}, E, undef, q0, ∅, undef, undef)
            ((M', q'), κ) := Γ_P(ω', ω, p, (M, q0), ∅, undef)
            M^ω := M'
        end
end
```

Daß dieser Algorithmus tatsächlich widerspruchsfreie Pfadbedingungen berechnet, wird in Kapitel 8 bewiesen werden. Der Algorithmus besteht im Prinzip aus folgenden Schritten, die für jede im Eingabeprogramm vorkommende Dereferenzierungsoperation durchzuführen sind (innerste Schleife): Erzeugen eines neuen Automaten mit nur einem Anfangszustand, keinem Endzustand und Funktionen δ, result und conditions, die zunächst undefiniert sind. Anschließend wird durch die Query–Funktion Γ_P wie angegeben ein Tupel $((M', q'), \kappa)$ berechnet und der darin enthaltene Automat M' als Ergebnis M^ω wie oben beschrieben zugewiesen. Die Menge der so bearbeiteten Dereferenzierungsoperationen wird dabei in der umgekehrten Reihenfolge der Referenzierungsstufen ihrer Variablen bearbeitet. Das exakte Ergebnis der Dereferenzierung einer Zeigervariable kann nur vom Ergebnis von anderen Zeigervariablen abhängen, die grössere Referenzierungsstufe besitzen, da nur deren Dereferenzierung die Zeigervariable ergeben und diese dadurch beeinflussen können. Durch die angegebene Reihenfolge der Betrachtung von Dereferenzierungsoperationen durch den Algorithmus kann damit gewährleistet werden, daß bei der Analyse der Zeigerziele einer Variablen sämtliche andere Queries, von denen diese Analyse abhängen kann, bereits berechnet wurden, und diese ggfs. von der aktuellen Query verwendet werden können. Das Initiieren einer Query für eine konkrete Dereferenzierungsoperation ist in Abbildung 7.12 dargestellt. Jede Dereferenzierungsoperation konsumiert bei ihrer Ausführung einen benannten Wert, der die Adresse einer Variable beinhaltet. Damit ist für das Ziel der Query entscheidend, welcher Inhalt dabei im konsumierten benannten Wert vorkommt. Um dies herauszufinden, wird eine Query Γ_P generiert, die anschaulich gesehen in entgegengesetzter Richtung entlang der Kante vom Produzenten des benannten Wertes, in der Abbildung der Operation ω', zur Dereferenzierungsoperation geschickt wird. Dies wird in der Abbildung durch den fett gedruckten Pfeil entgegen der Kantenrichtung dargestellt. Die Parameter der Query sind entsprechend mit eingezeichnet. Der daneben abgebildete betriebene Automat (M, q), der der Query als

Parameter mitgegeben wird, ist dabei durch die Initialisierung im Query–Algorithmus gleich dem dargestellten Automat, der nur aus seinem Grundzustand besteht.

7.4.4 Definition der Query–Funktion für Operationen

Für die verschiedenen Grundoperationen sollen nun die Query–Funktionen Γ_V und Γ_P angegeben werden. Aufgrund der vielen zusammenhängenden Definitionen wird dies nicht in der Form vieler einzelner numerierter Definitionen, sondern als eine große zusammenhängende Definition mit Gleichungsnummern zur Kennzeichnung der einzelnen Bestandteile der Definition geschehen.

7.4.4.1 Get–Operation mit fester auszulesender Variable

Für eine Get–Operation $\text{get}_v \in \Omega$ mit $\text{get}_v \in \beta$ für ein $\beta \in \Lambda$ werden Γ_P und Γ_V wie folgt definiert. Dazu werden zunächst Funktionen $\widehat{\Gamma_V}$ und $\widehat{\Gamma_P}$ definiert, die die Auswirkung einer Get–Operation auf eine konkrete Query beschreiben. Die eigentlichen Funktionen Γ_V und Γ_P werden dann aus der Anwendung dieser Funktionen $\widehat{\Gamma_V}$ und $\widehat{\Gamma_P}$ auf die Menge von Kombinationen aus Zuständen und Querybedingungen angewandt, die sich als Ergebnis der Funktion IntegrateConditions ergeben, die ggfs. Zustandsübergänge der als Querybedingungen mitgeführten betriebenen Automaten in den Protokollautomaten der Query integriert.

$$\widehat{\Gamma_V}(\text{get}_v, \omega_{\text{deref}}, v, (\underbrace{(Q, E, \delta, q_0, F, \text{result}, \text{conditions})}_{=:M}, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$

$$:= \begin{cases} (((Q, E, \delta[q \xrightarrow{\epsilon} q'], q_0, F, \text{result}, \text{conditions}), q), \kappa), \\ \qquad \text{falls } \kappa(\text{get}_v, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) = (M', q') \\ \Gamma_V(\text{prev\_use}(v, \text{get}_v, \beta), \omega_{\text{deref}}, v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \\ \quad \kappa[(\text{get}_v, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) \rightarrow (M, q)]), \\ \qquad \text{sonst} \end{cases} \tag{7.1}$$

$$\widehat{\Gamma_P}(\text{get}_v, \omega_{\text{deref}}, p, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$

$$:= \begin{cases} (((Q, E, \delta[q \xrightarrow{\epsilon} q'], q_0, F, \text{result}, \text{conditions}), q), \kappa), \\ \qquad \text{falls } \kappa(\text{get}_v, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) = (M', q') \\ \Gamma_V(\text{prev\_use}(v, \text{get}_v, \beta), \omega_{\text{deref}}, v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \\ \quad \kappa[(\text{get}_v, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) \rightarrow (M, q)]), \\ \qquad \text{sonst} \end{cases} \tag{7.2}$$

Damit definiert man nun mit

$$\begin{aligned} &(M', \{(q^{(1)}, \{(M_1^{(1)}, q_1^{(1)}), \ldots, (M_{l_1}^{(1)}, q_{l_1}^{(1)})\}), \ldots, (q^{(m)}, \{(M_1^{(m)}, q_1^{(m)}), \ldots, (M_{l_m}^{(m)}, q_{l_m}^{(m)})\}) \\ &\quad := \text{IntegrateConditions}(\text{get}_v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \text{False}), \end{aligned}$$

für $m \in \mathbb{N}$ und $\forall 1 \leq i \leq m : l_i \in \mathbb{N}_0$ die eigentliche Queryfunktion wie folgt:

$$\begin{aligned} &\Gamma_V(\text{get}_v, \omega_{\text{deref}}, v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}), \kappa) \\ &\quad := \quad ((M^m, q^m), \kappa^m), \end{aligned} \tag{7.3}$$

mit

$$\begin{aligned} ((M^1, q^1), \kappa^1) &:= \widehat{\Gamma_V}(\text{get}_v, \omega_{\text{deref}}, v, (M', q^{(1)}), \{(M_1^{(1)}, q_1^{(1)}), \ldots, (M_{l_1}^{(1)}, q_{l_1}^{(1)})\}, \kappa) \\ ((M^2, q^2), \kappa^2) &:= \widehat{\Gamma_V}(\text{get}_v, \omega_{\text{deref}}, v, (M^1, q^{(2)}), \{(M_1^{(2)}, q_1^{(2)}), \ldots, (M_{l_2}^{(2)}, q_{l_2}^{(2)})\}, \kappa^1) \\ &\ldots \\ ((M^m, q^m), \kappa^m) &:= \widehat{\Gamma_V}(\text{get}_v, \omega_{\text{deref}}, v, (M^{m-1}, q^{(m)}), \{(M_1^{(m)}, q_1^{(m)}), \ldots, (M_{l_m}^{(m)}, q_{l_m}^{(m)})\}, \kappa^{m-1}) \end{aligned}$$

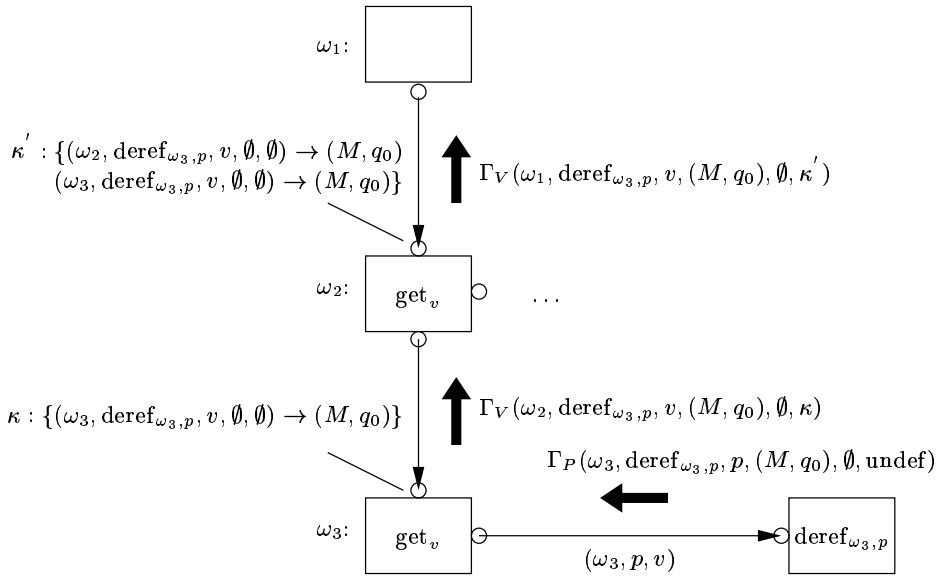


Abbildung 7.13: Query–Verarbeitung an Get–Operationen.

bzw.

$$\begin{aligned}&\Gamma_P(\text{get}_v, \omega_{\text{deref}}, p, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}), \kappa)\\ &\quad := \quad ((M^m, q^m), \kappa^m),\end{aligned} \tag{7.4}$$

mit

$$\begin{aligned}((M^1, q^1), \kappa^1) &:= \widehat{\Gamma_P}(\text{get}_v, \omega_{\text{deref}}, p, \{(M_1^{(1)}, q_1^{(1)}), \ldots, (M_{l_1}^{(1)}, q_{l_1}^{(1)})\}, (M', q^{(1)}), \kappa)\\ ((M^2, q^2), \kappa^2) &:= \widehat{\Gamma_P}(\text{get}_v, \omega_{\text{deref}}, p, \{(M_1^{(2)}, q_1^{(2)}), \ldots, (M_{l_2}^{(2)}, q_{l_2}^{(2)})\}, (M^1, q^{(2)}), \kappa^1)\\ &\ldots\\ ((M^m, q^m), \kappa^m) &:= \widehat{\Gamma_P}(\text{get}_v, \omega_{\text{deref}}, p, \{(M_1^{(m)}, q_1^{(m)}), \ldots, (M_{l_m}^{(m)}, q_{l_m}^{(m)})\}, (M^{m-1}, q^{(m)}), \kappa^{m-1}),\end{aligned}$$

Die Bezeichnungen mit hochgestellter Numerierung stellen dabei die Zwischenergebnisse dar, die nach sequentieller Anwendung der Queryfunktion auf die verschiedenen Ergebnisse der Funktion IntegrateConditions (mit hochgestellter Numerierung in Klammern) entstehen.

In Abbildung 7.13 ist diese Definition anschaulich an einer einfachen Beispielsequenz von Operationen dargestellt. Dort soll die Dereferenzierungsoperation den von der Get–Operation ω_3 produzierten benannten Wert konsumieren. Nach der Definition 7.10 des Query–Algorithmus wird damit bei der Analyse die in der Zeichnung analog zu Abbildung 7.12 initiierte Ausführung der Query–Operation Γ_P veranlasst. Da bislang keine Querybedingungen mitgeführt werden, ist IntegrateConditions die identische Abbildung, und die Berechnung von $\Gamma_P(\ldots)$ ergibt sich direkt aus der Berechnung von $\widehat{\Gamma_P}(\ldots)$. Dafür kann man die obige Definition in Gleichung (7.2) verwenden, nach der sich diese aus einer Anwendung von Γ_V ergibt, wobei u.A. ein Parameter verwendet wird, der sich aus der letzten Verwendung $\text{prev\_use}(\omega_3, v, \beta)$ für ein geeignetes $\beta \in \Lambda$ ergibt. Da die Funktion $\kappa \in \mathcal{K}$ bislang undefiniert ist, kann der in dieser Gleichung zuerst beschriebene Fall nicht eintreten. Für die Auswertung von Γ_V kann man nun die Gleichungen (7.3) und (7.1) verwenden, nach der sich $\Gamma_V(\omega_2, \ldots)$ aus $\widehat{\Gamma_V}(\omega_2, \ldots)$, und dieses wiederum aus $\Gamma_V(\omega_1, \ldots)$ berechnen läßt. Dies ist im Beispiel durch einen weiteren fett gedruckten Pfeil dargestellt. In der Abbildung ebenfalls gezeigt sind die Funktionen $\kappa \in \mathcal{K}$ bzw. $\kappa' \in \mathcal{K}$, die während des Flusses der Query die Information aufsammeln, an welchen Operationen mit welchen Querybedingungen bereits vorher Queries bearbeitet wurden. Im Beispiel besteht der der Query zugeordnete betriebene Automat stets nur aus seinem Anfangszustand

und es werden keine betriebenen Automaten als Querybedingungen verwendet. Daher assoziiert die Funktion κ' die Operationen ω_2 und ω_3 zusammen mit jeweils einer leeren Menge von betriebenen Automaten als Querybedingungen mit dem betriebenen Automaten M im Zustand q_0.

7.4.4.2 Get–Operation mit variabler auszulesender Variable

Für eine Get–Operation $\omega = \text{get}_{\omega'}$ mit $\omega \in \beta$ für ein $\beta \in \Lambda$, die von einer Dereferenzierungsoperation $\omega' \in \Omega_0$ abhängt, werden die Query–Funktionen wie folgt definiert. Dabei wird davon ausgegangen, daß für die Query–Operation ω' bereits ein Automat $M^{\omega'} = (Q^{\omega'}, E, \delta^{\omega'}, q_0^{\omega'}, F^{\omega'}, \text{result}^{\omega'}, \text{conditions}^{\omega'})$ existiert, der das Ergebnis der Querybearbeitung im Rahmen des Algorithmus aus Abschnitt 7.4.3 für diese Dereferenzierungsoperation beschreibt. Wie bei der obigen Definition der Querybearbeitung an einer Get–Operation, die nicht von einer Dereferenzierungsoperation abhängt, kann bei der Queryverarbeitung in diesem Fall ebenfalls der Protokollautomat der Query verändert werden. Daher wird auch dieser Definition eine Überprüfung vorausgeschickt, ob Zustandsübergänge aus den als Querybedingungen mitgeführten betriebenen Automaten in den betriebenen Automaten der Query integriert werden müssen, bevor die Queryverarbeitung fortgesetzt wird. In der folgenden Definition ist wie in der vorherigen die Definition der Queryfunktionen $\widehat{\Gamma_V}$ und $\widehat{\Gamma_P}$ angegeben, die die Verarbeitung einer Query in einem aktuellen Zustand des Protokollautomaten mit einer konkreten Menge von als Querybedingungen mitgeführten betriebenen Automaten beschreibt. Diese Funktionen werden dann wie nachfolgend beschrieben analog zu den Gleichungen (7.3) und (7.4) sequentiell auf die ggfs. mehreren Elemente der Ergebnismenge der Funktion IntegrateConditions angewandt.

$$\widehat{\Gamma_V}(\text{get}_{\omega'}, \omega_{\text{deref}}, v, (\underbrace{(Q, E, \delta, q_0, F, \text{result}, \text{conditions})}_{=:M}, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$

$$:= \begin{cases} (((Q, E, \delta[q \xrightarrow{\epsilon} q'], q_0, F, \text{result}, \text{conditions}), q), \kappa), \\ \qquad \text{falls } \kappa(\text{get}_{\omega'}, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) = (M', q') \\ \\ \Gamma_V(\text{prev\_use}(v, \omega', \beta), \omega_{\text{deref}}, v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \\ \quad \kappa[(\text{get}_{\omega'}, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) \rightarrow (M, q)]), \\ \qquad \text{sonst} \end{cases}$$

In dieser Definition wird ausgesagt, daß eine Get–Operation für die Frage der letzten Zuweisung zu einer Variablen keine Auswirkung hat. Entsprechend wird die Query Γ_V an der von der Dereferenzierungsoperation ω' aus gesehen letzten Verwendung der Variablen fortgesetzt, sofern bei der Bearbeitung dieser Query keine Wiederholung festgestellt werden kann. Andernfalls wird ein entsprechender ϵ–Zustandsübergang erzeugt und die Querybearbeitung damit beendet.

Die Integration von Zustandsübergängen aus den als Querybedingungen mitgeführten Automaten geschieht an dieser Stelle unterschiedlich von den anderen Definitionen. Obwohl die Get–Operation in keinem Fall eine Auswirkung auf die Belegung der Variablen v hat, und es daher unrelevant ist, welche Zeigerziele die Dereferenzierungsoperation ω' besitzt, kann es vorkommen, daß als Querybedingungen mitgeführte Automaten Eingabesymbole der Form $[\omega' \rightarrow \ldots]$ besitzen. Diese müssen dann bei der Querybearbeitung an der Get–Operation berücksichtigt werden. Daher definiert man im vorliegenden Fall (mit verändertem letzten Parameter der Funktion IntegrateConditions) mit

$$\begin{aligned} &(M', \{(q^{(1)}, \{(M_1^{(1)}, q_1^{(1)}), \ldots, (M_{l_1}^{(1)}, q_{l_1}^{(1)})\}), \ldots, (q^{(m)}, \{(M_1^{(m)}, q_1^{(m)}), \ldots, (M_{l_m}^{(m)}, q_{l_m}^{(m)})\}) \\ &\quad := \text{IntegrateConditions}(\text{get}_v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \text{True}), \end{aligned}$$

für $m \in \mathbb{N}$ und $\forall 1 \leq i \leq m : l_i \in \mathbb{N}_0$ die Queryfunktion Γ_V auf der Basis der Queryfunktion $\widehat{\Gamma_V}$ wie folgt:

$$\begin{aligned} &\Gamma_V(\text{get}_{\omega'}, \omega_{\text{deref}}, v, (M, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}), \kappa) \\ &\quad := \quad ((M^m, q^m), \kappa^m), \end{aligned} \tag{7.5}$$

mit

$$\begin{aligned}
((M^1,q^1),\kappa^1) &:= \widehat{\Gamma_V}(\mathrm{get}_{\omega'},\omega_{\mathrm{deref}},v,(M^{'},q^{(1)}),\{(M_1^{(1)},q_1^{(1)}),\ldots,(M_{l_1}^{(1)},q_{l_1}^{(1)})\},\kappa)\\
((M^2,q^2),\kappa^2) &:= \widehat{\Gamma_V}(\mathrm{get}_{\omega'},\omega_{\mathrm{deref}},v,(M^1,q^{(2)}),\{(M_1^{(2)},q_1^{(2)}),\ldots,(M_{l_2}^{(2)},q_{l_2}^{(2)})\},\kappa^1)\\
&\ldots\\
((M^m,q^m),\kappa^m) &:= \widehat{\Gamma_V}(\mathrm{get}_{\omega'},\omega_{\mathrm{deref}},v,(M^{m-1},q^{(m)}),\{(M_1^{(m)},q_1^{(m)}),\ldots,(M_{l_m}^{(m)},q_{l_m}^{(m)})\},\kappa^{m-1})
\end{aligned}$$

Für die nachfolgende Definition sei mit $V^{\omega^{'}} := \{v_1,\ldots,v_m\} := \mathrm{result}^{\omega^{'}}(F^{\omega^{'}})$ für ein $m \in \mathbb{N}$ (dem Bild der Menge $F^{\omega^{'}}$ unter der Abbildung $\mathrm{result}^{\omega^{'}}$) die Menge aller Variablen bezeichnet, auf die Endzustände des Automaten $M^{\omega^{'}}$ durch die $\mathrm{result}^{\omega^{'}}$–Funktion abgebildet werden. Damit beschreibt $V^{\omega^{'}}$ die Menge aller Variablen, die tatsächlich Ziel der Dereferenzierungsoperation $\omega^{'}$ sein können. Wiederum bezeichne nach Definition 5.38 für $x \in V^{\omega^{'}}$

$$F_x^{\omega^{'}} = \{q \in F^{\omega^{'}} \mid \mathrm{result}^{\omega^{'}}(q) = x\} \subseteq F^{\omega^{'}}$$

die Menge derjenigen Endzustände, die von der $\mathrm{result}^{\omega^{'}}$–Abbildung auf genau die Variable $x \in V$ abgebildet werden. Zusätzlich sollen $q^1,\ldots,q^n$ neue Zustände bezeichnen, die in keiner der Zustandsmengen $Q,Q_1,\ldots,Q_n$ der Automaten aus der folgenden Definition enthalten sein sollen. Wiederum soll die eigentliche Funktion Γ_P analog zu Gleichung (7.4) definiert sein.

$$\begin{aligned}
&\widehat{\Gamma_P}(\mathrm{get}_{\omega'},\omega_{\mathrm{deref}},p,((Q,E,\delta,q_0,F,\mathrm{result},\mathrm{conditions}),q),\{(M_1,q_1),\ldots,(M_n,q_n)\},\kappa)\\
&\quad := (((Q^{(m)},E,\delta^{(m)},q_0^{(m)},F^{(m)},\mathrm{result}^{(m)},\mathrm{conditions}^{(m)}),q^{(m)}),\kappa^{(m)}), \qquad (7.6)
\end{aligned}$$

wobei mit q als dem entsprechenden Eingabeparameter aus Gleichung (7.6)

$$\begin{aligned}
&(((Q^{(1)},E,\delta^{(1)},q_0^{(1)},F^{(1)},\mathrm{result}^{(1)},\mathrm{conditions}^{(1)}),q^{(1)}),\kappa^{(1)})\\
&:= \begin{cases}
(((Q,E,\delta,q_0,F,\mathrm{result},\mathrm{conditions}),q),\kappa),\\
\qquad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega^{'}\to v_1]) \text{ ist nicht definiert,}\\
\Gamma_V(\mathrm{prev_use}(v_1,\omega^{'},\beta),\omega_{\mathrm{deref}},v_1,\\
\quad \underbrace{((Q\cup\{q^1\},E,\delta[q^{[\omega^{'}\to v_1]}q^1],q_0,F,\mathrm{result},\mathrm{conditions})}_{=:M^1},q^1),\\
\quad \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega^{'}\to v_1])\\
\quad \cup\{(\underbrace{(Q^{\omega^{'}},E,\delta^{\omega^{'}},q_0^{\omega^{'}},F_{v_1}^{\omega^{'}},\mathrm{result}^{\omega^{'}},\mathrm{conditions}^{\omega^{'}})}_{=:M_{v_1}},q_0^{\omega^{'}})\},\\
\quad \kappa[(\mathrm{get}_{\omega'},\omega_{\mathrm{deref}},v_1,\emptyset,\delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega^{'}\to v_1])\cup\{(M_{v_1},q_0^{\omega^{'}})\})\to(M^1,q^1)]),\\
\qquad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega^{'}\to v_1]) \text{ ist definiert,}\\
\qquad \text{und } \kappa(\mathrm{get}_{\omega'},\omega_{\mathrm{deref}},v_1,\emptyset,\delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega^{'}\to v_1])\cup\{(M_{v_1},q_0^{\omega^{'}})\})\\
\qquad \text{ist nicht definiert,}\\
(((Q,E,\delta[q^{[\omega^{'}\to v_1]}q^{'}],q_0,F,\mathrm{result},\mathrm{conditions}),q),\kappa),\\
\qquad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega^{'}\to v_1]) \text{ ist definiert,}\\
\qquad \text{und } \kappa(\mathrm{get}_{\omega'},\omega_{\mathrm{deref}},v_1,\emptyset,\delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega^{'}\to v_1])\cup\{(M_{v_1},q_0^{\omega^{'}})\})\\
\qquad = (M^{'},q^{'})
\end{cases}\\
&(((Q^{(2)},E,\delta^{(2)},q_0^{(2)},F^{(2)},\mathrm{result}^{(2)},\mathrm{conditions}^{(2)}),q^{(2)}),\kappa^{(2)})
\end{aligned}$$

$$
:= \begin{cases}
(((Q^{(1)}, E, \delta^{(1)}, q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q), \kappa^{(1)}), \\
\qquad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_2]) \text{ ist nicht definiert,} \\
\\
\Gamma_V(\text{prev\_use}(v_2, \omega', \beta), \omega_{\text{deref}}, v_2, \\
\quad \underbrace{((Q^{(1)} \cup \{q^2\}, E, \delta^{(1)}[q \overset{[\omega' \to v_2]}{\to} q^2], q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)})}_{=:M^2}, q^2), \\
\quad \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_2]) \\
\quad \cup \{(\underbrace{(Q^{\omega'}, E, \delta^{\omega'}, q_0^{\omega'}, F_{v_2}^{\omega'}, \text{result}^{\omega'}, \text{conditions}^{\omega'})}_{=:M_{v_2}}, q_0^{\omega'})\}, \\
\quad \kappa^{(1)}[(\text{get}_{\omega'}, \omega_{\text{deref}}, v_2, \emptyset, \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_2]) \cup \{(M_{v_2}, q_0^{\omega'})\}) \to (M^2, q^2)]), \\
\qquad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_2]) \text{ ist definiert,} \\
\qquad \text{und } \kappa^{(1)}(\text{get}_{\omega'}, \omega_{\text{deref}}, v_2, \emptyset, \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_2]) \cup \{(M_{v_2}, q_0^{\omega'})\}) \\
\qquad \text{ist nicht definiert,} \\
\\
(((Q^{(1)}, E, \delta^{(1)}[q \overset{[\omega' \to v_2]}{\to} q'], q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q), \kappa^{(1)}), \\
\qquad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_2]) \text{ ist definiert,} \\
\qquad \text{und } \kappa^{(1)}(\text{get}_{\omega'}, \omega_{\text{deref}}, v_2, \emptyset, \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_2]) \cup \{(M_{v_2}, q_0^{\omega'})\}) \\
\qquad = (M', q')
\end{cases}
$$

...

$$
(((Q^{(m)}, E, \delta^{(m)}, q_0^{(m)}, F^{(m)}, \text{result}^{(m)}, \text{conditions}^{(m)}), q^{(m)}), \kappa^{(m)})
$$

$$
:= \begin{cases}
(((Q^{(m-1)}, E, \delta^{(m-1)}, q_0^{(m-1)}, F^{(m-1)}, \text{result}^{(m-1)}, \text{conditions}^{(m-1)}), q), \kappa^{(m-1)}), \\
\qquad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_m]) \text{ ist nicht definiert,} \\
\\
\Gamma_V(\text{prev\_use}(v_m, \omega', \beta), \omega_{\text{deref}}, v_m, \\
\quad \underbrace{((Q^{(m-1)} \cup \{q^m\}, E, \delta^{(m-1)}[q \overset{[\omega' \to v_m]}{\to} q^m], q_0^{(m-1)}, F^{(m-1)}, \text{result}^{(m-1)}, \text{conditions}^{(m-1)})}_{=:M^m}, q^m), \\
\quad \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_m]) \\
\quad \cup \{(\underbrace{(Q^{\omega'}, E, \delta^{\omega'}, q_0^{\omega'}, F_{v_m}^{\omega'}, \text{result}^{\omega'}, \text{conditions}^{\omega'})}_{=:M_{v_m}}, q_0^{\omega'})\}, \\
\quad \kappa^{(m-1)}[(\text{get}_{\omega'}, \omega_{\text{deref}}, v_m, \emptyset, \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_m]) \cup \{(M_{v_m}, q_0^{\omega'})\}) \to (M^m, q^m)]), \\
\qquad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_m]) \text{ ist definiert,} \\
\qquad \text{und } \kappa^{(m-1)}(\text{get}_{\omega'}, \omega_{\text{deref}}, v_m, \emptyset, \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_m]) \cup \{(M_{v_m}, q_0^{\omega'})\}) \\
\qquad \text{ist nicht definiert,} \\
\\
(((Q^{(m-1)}, E, \delta^{(m-1)}[q \overset{[\omega' \to v_m]}{\to} q'], q_0^{(m-1)}, F^{(m-1)}, \text{result}^{(m-1)}, \text{conditions}^{(m-1)}), q), \kappa^{(m-1)}), \\
\qquad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_m]) \text{ ist definiert,} \\
\qquad \text{und } \kappa^{(m-1)}(\text{get}_{\omega'}, \omega_{\text{deref}}, v_m, \emptyset, \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, [\omega' \to v_m]) \cup \{(M_{v_m}, q_0^{\omega'})\}) \\
\qquad = (M', q')
\end{cases}
$$

Die verschiedenen $(((Q^{(i)}, \ldots), q^{(i)}), \kappa^{(i)})$ für $1 \leq i \leq m$ bezeichnen dabei Zwischenergebnisse der Berechnung, die jeweils unter der Annahme, daß eine der Variablen v_1 bis v_m Ziel der Dereferenzierungsoperation ist, die Querybearbeitung fortsetzt. Allgemein stellen in der Definition Terme mit oberem Index in Klammern solche Zwischenergebnisse dar. Terme mit oberem Index ohne Klammern sind die neu eingeführten Zustände bzw. diejenigen Automaten, die durch Einfügen dieser neuen Zustände entstehen. Betriebene Automaten mit unterem ungeklammertem Index sind diejenigen betriebenen Automaten, die bereits als Querybedingungen mit der Query mitgeführt werden.

Der jeweils erste Fall in den geschweiften Klammern stellt diejenige Situation dar, in der die Annahme, daß die Dereferenzierungsoperation ω' das entsprechende Zeigerziel v_i (mit i wie oben beschrieben) besitzt, nicht mit den mitgeführten Querybedingungen vereinbar ist. In diesem Fall wird die Queryverarbeitung unter dieser Annahme nicht fortgesetzt, und der Protokollautomat nicht verändert. Der dritte Fall in den geschweiften Klammern stellt jeweils den Fall dar, in dem eine Wiederholung festgestellt werden kann. Dazu wird zur Berechnung des $i+1$–ten Zwischenergebnisses für $0 \leq i \leq m-1$ überprüft, ob die Funktion $\kappa^{(i)}$, die sich aus dem i–ten Zwischenschritt ergeben hat, bzw. κ für $i = 0$, aussagt, daß bereits eine Query mit denjenigen Querybedingungen bearbeitet wurde, die durch die Annahme, daß die Dereferenzierungsoperation ω' die Variable v_i zum Ziel hat, entstehen würden. Dies umfaßt sowohl das Hinzufügen desjenigen betriebenen Automaten, der eine Pfadbedingung für diese Annahme darstellt, als auch das Ergebnis des Zustandsüberganges der bereits als Querybedingungen mitgeführten betriebenen Automaten mit dem entsprechenden Symbol.

Die anderen Fälle basieren auf dem folgenden Prinzip. Das erste Tupel $(((Q^{(1)}, \ldots), q^{(1)}), \kappa^{(1)})$ wird in der Art berechnet, daß das Ergebnis der Query Γ_P auf eine Query Γ_V zurückgeführt wird, die an der letzten Verwendung der Variablen v_1 vor der deref–Operation nach deren möglichen Belegungen suchen soll. Der Protokollautomat der Query wird dabei um einen Zustand $q^{(1)}$ erweitert, zu dem von dessen bisherigem aktuellen Zustand q ein Zustandsübergang mit dem Eingabesymbol $[\omega' \rightarrow v_1]$ führt. Dieser Zustand $q^{(1)}$ wird auch zum aktuellen Zustand des Protokollautomaten gemacht. Dies entspricht der Vorgehensweise, die in Abbildung 7.6 dargestellt wurde. Zusätzlich wird zur Menge der bereits als Querybedingungen mitgeführten betriebenen Automaten $\{(M_1, q_1), \ldots, (M_n, q_n)\}$, auf die die Zustandsübergangsfunktion δ^* mit dem Eingabesymbol $[\omega' \rightarrow v_1]$ angewandt wurde, ein zusätzlicher betriebener Automat hinzugefügt, der die Ergebnisse der Analyse für die Dereferenzierungsoperation ω' darstellt. Dieser wird gegenüber der Endzustandsmenge $F^{\omega'}$ eingeschränkt auf diejenigen Endzustände $F^{\omega'}_{v_1}$, denen von der Abbildung result$^{\omega'}$ die Variable v_1 zugeordnet wird. Dies entspricht damit der Vorgehensweise, die ebenfalls im Beispiel in Abbildung 7.6 dargestellt wurde. Als Ergebnis dieser Query–Funktion Γ_V entstehen ein Protokollautomat $((Q^{(1)}, \ldots), q^{(1)})$ und eine Abbildung $\kappa^{(1)}$. In ersterem sind die Ergebnisse dieser Query in der Form von neuen Zuständen, Endzuständen und Zustandsübergängen gespeichert. Die Funktion $\kappa^{(1)}$ vermerkt, an welchen Operationen diese Query mit welchen Querybedingungen bereits bearbeitet wurde. In der zweiten Gleichung, in der $(((Q^{(2)}, \ldots), q^{(2)}), \kappa^{(2)})$ berechnet wird, wird diese Information als Eingabe an die dort verwendete Queryfunktion Γ_V verwendet. Damit dient derjenige Protokollautomat, der bereits die Ergebnisse der ersten Teilberechnung enthält, als Grundlage für die zweite Teilberechnung. Ebenso ist in der zweiten Teilberechnung durch die Funktion $\kappa^{(1)}$ bereits bekannt, an welchen Operationen entsprechende Queries bereits bearbeitet wurden. Damit können auch Wiederholungen der Querybearbeitung festgestellt werden, die an der aktuell untersuchten Operation get$_{\omega'}$ von verschiedenen Querybedingungen ausgehend entstehen können. Jede der m Teilberechnungen geht dabei von der Annahme aus, daß die Dereferenzierungsoperation ω' als Ziel eine der Variablen $v_1, \ldots, v_m$ besitzt, die sich aus der result$^{\omega'}$–Abbildung des Automaten $M^{\omega'}$ ergeben, der der Dereferenzierungsoperation ω' durch den Queryalgorithmus zuvor bereits zugeordnet wurde. Diese Annahmen werden jeweils durch Eingabesymbole in den Protokollautomaten integriert, und ein entsprechender betriebener Automat als Querybedingung der Query hinzugefügt. Nach der Berechnung aller m Zwischenergebnisse wird dann das Ergebnis der $\widehat{\Gamma_P}$–Query in Gleichung (7.6) als das letzte Zwischenergebnis $(((Q^{(m)}, \ldots), q^{(m)}), \kappa^{(m)})$ definiert.

In Abbildung 7.14 ist diese Vorgehensweise nochmal im Überblick dargestellt. Wie bisher soll angenommen werden, daß die Funktion IntegrateConditions im Beispiel die identische Abbildung ist. Die Queryfunktion Γ_P, die die Get–Operation erreicht, wird durch mehrere Queries Γ_V berechnet. Für jede dieser Queries wird dabei im Protokollautomaten ein Zustandsübergang mit einem Symbol, das die entsprechend dabei angenommene Programmeigenschaft der Form $[\omega' \rightarrow v_i]$ mit $i \in \{1, \ldots, m\}$ ausdrückt, erzeugt. Diese sind in der Abbildung neben den jeweiligen Queries Γ_V dargestellt. Bei den Protokollautomaten ist auch jeweils beispielhaft ein betriebener Automat dargestellt, der fortan als (zusätzliche) Querybedingung mitgeführt wird. Dieser entspricht jeweils dem Ergebnis der Analyse für die Dereferenzierungsoperation ω' mit seinem Startzustand als aktuellem

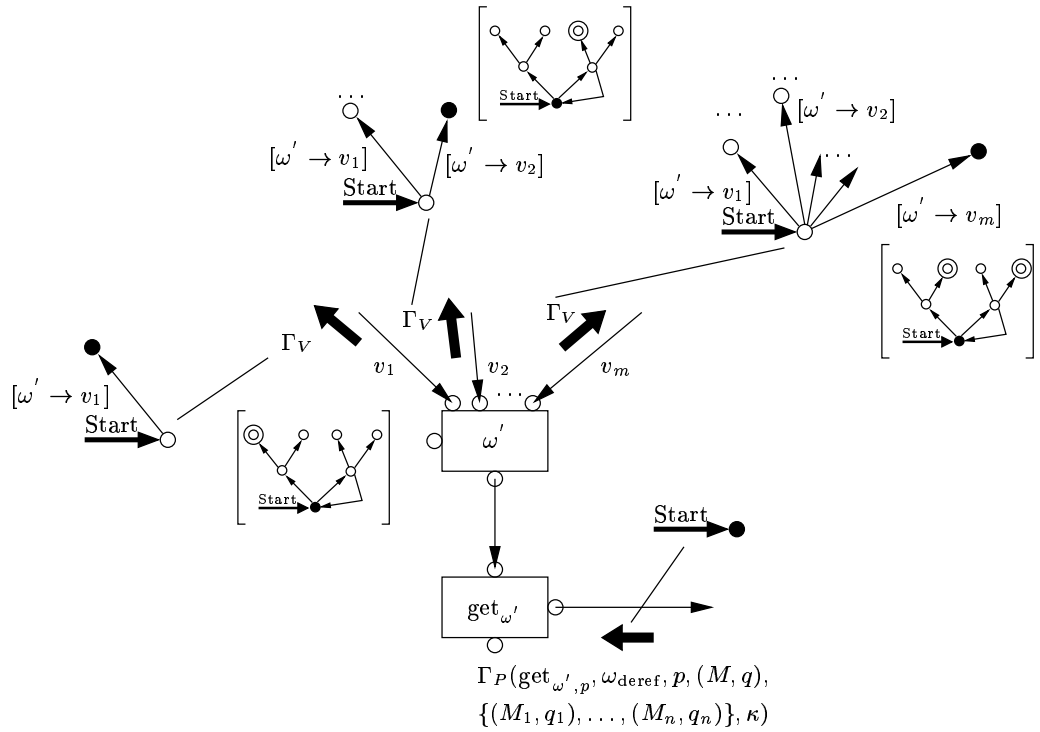


Abbildung 7.14: Query–Verarbeitung an Get–Operationen, die von Dereferenzierungsoperationen abhängen.

Zustand. Die Menge der jeweiligen Endzustände ist jedoch verschieden, wie bereits erläutert wurde.

7.4.4.3 Set–Operation mit fester auszulesender Variable

Für eine Set–Operation $\text{set}_{v,\omega',p} \in \Omega$ mit $\text{set}_v \in \beta$ für ein $\beta \in \Lambda$ werde Γ_V wie folgt definiert. Da in diesem Fall der Protokollautomat nicht verändert wird, muß an dieser Stelle (noch) nicht auf eine notwendige Integration von Zustandsübergängen von als Querybedingungen mitgeführten betriebenen Automaten getestet werden. Daher ergibt sich die Definition für Γ_V direkt wie folgt:

$$\begin{aligned}&\Gamma_V(\text{set}_{v,\omega',p}, \omega_{\text{deref}}, v, ((Q,E,\delta,q_0,F,\text{result},\text{conditions}),q), \{(M_1,q_1),\ldots,(M_n,q_n)\},\kappa)\\ &:= \quad \Gamma_P(\omega', \omega_{\text{deref}}, p, ((Q,E,\delta,q_0,F,\text{result},\text{conditions}),q), \{(M_1,q_1),\ldots,(M_n,q_n)\},\kappa) \quad (7.7)\end{aligned}$$

Da eine Set–Operation keinen benannten Wert produziert, können keine Queries Γ_P eine Set–Operation erreichen. In Abbildung 7.15 ist die Vorgehensweise gemäß Definition (7.7) veranschaulicht. Die abgebildete Set–Operation stellt die letzte Verwendung einer Variablen v dar. Der Wert, der dieser dabei zugewiesen wird, ergibt sich aus dem dabei konsumierten benannten Wert (ω', p, s). Entsprechend muß zur Ermittlung der möglichen Belegungen von v eine Query Γ_P zur Ermittlung der möglichen benannten Werte verwendet werden. An set–Operationen muß keine Wiederholungserkennung durchgeführt werden. Damit wird die Queryspeicherfunktion κ nicht verändert.

7.4.4.4 Set–Operation mit variabler auszulesender Variable

Für eine Set–Operation $\text{set}_{\omega'',\omega',p} \in \Omega$ mit $\text{set}_{\omega'',\omega',p} \in \beta$ für ein $\beta \in \Lambda$ wird die Queryfunktion Γ_V wie folgt definiert. Dabei sei wiederum mit $M^{\omega''} = (Q^{\omega''}, E, \delta^{\omega''}, q_0^{\omega''}, F^{\omega''}, \text{result}^{\omega''}, \text{conditions}^{\omega''})$ das Queryergebnis der Dereferenzierungsoperation ω'' bekannt, und mit $V^{\omega''} := \{v_1,\ldots,v_m\} :=$

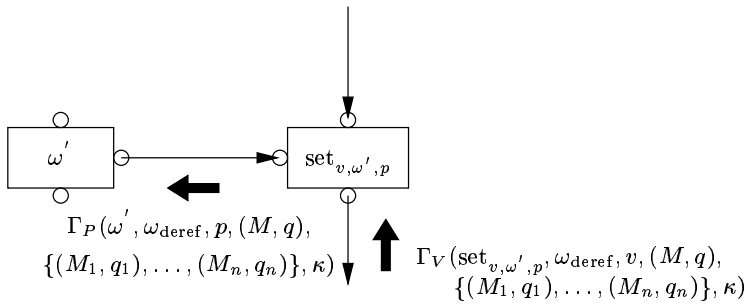


Abbildung 7.15: Queryverarbeitung an Set–Operationen.

$\text{result}^{\omega''}(F^{\omega''})$ für ein $m \in \mathbb{N}$ (dem Bild der Menge $F^{\omega''}$ unter der Abbildung $\text{result}^{\omega''}$) die Menge aller Variablen bezeichnet, für die es Endzustände des Automaten $M^{\omega''}$ gibt, für die die $\text{result}^{\omega''}$–Abbildung diese Variablen ergibt. Damit beschreibt $V^{\omega''}$ die Menge aller Variablen, die tatsächlich Ziel der Dereferenzierungsoperation ω'' sein können. O.B.d.A. sei diese Menge von Variablen so numeriert, daß die Variable v, die Parameter der Γ_V–Funktion ist, gleich der Variablen v_1 ist. Wiederum bezeichne nach Definition 5.38 für $x \in V^{\omega''}$

$$F_x^{\omega''} = \{q \in F^{\omega''} \mid \text{result}^{\omega''}(q) = x\} \subseteq F^{\omega''}$$

die Menge derjenigen Endzustände, die von der $\text{result}^{\omega''}$–Abbildung auf eine Variable $x \in V$ abgebildet werden. Zusätzlich sollen $q^1, \ldots, q^m$ hier ebenfalls neue Zustände bezeichnen, die in keiner der Zustandsmengen $Q, Q_1, \ldots, Q_n$ der Automaten aus der folgenden Definition enthalten sein sollen. Damit kann man nun $\widehat{\Gamma_V}$ für eine solche Set–Operation definieren. Analog zu Gleichung (7.3) wird dann daraus die Funktion Γ_V definiert.

$$\begin{aligned}&\widehat{\Gamma_V}(\text{set}_{\omega'',\omega',p}, \omega_{\text{deref}}, v, ((Q,E,\delta,q_0,F,\text{result},\text{conditions}),q),\{(M_1,q_1),\ldots,(M_n,q_n)\},\kappa)\\ &\quad := \quad (((Q^{(m)},E,\delta^{(m)},q_0^{(m)},F^{(m)},\text{result}^{(m)},\text{conditions}^{(m)}),q^{(m)}),\kappa^{(m)}),\end{aligned} \tag{7.8}$$

wobei

$$(((Q^{(1)},E,\delta^{(1)},q_0^{(1)},F^{(1)},\text{result}^{(1)},\text{conditions}^{(1)}),q^{(1)}),\kappa^{(1)})$$

$$:= \begin{cases} (Q,E,\delta,q_0,F,\text{result},\text{conditions}), \\ \quad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega'' \to v]) \text{ nicht definiert ist,} \\ \\ \Gamma_P(\omega', \omega_{\text{deref}}, p, (Q \cup \{q^1\}, E, \delta[q \overset{[\omega'' \to v]}{\to} q^1], q_0, F, \text{result}, \text{conditions}), q^1), \\ \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega'' \to v]) \\ \cup\{((Q^{\omega''},E,\delta^{\omega''},q_0^{\omega''},F_v^{\omega''},\text{result}^{\omega''},\text{conditions}^{\omega''}),q_0^{\omega''})\},\kappa), \\ \quad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega'' \to v]) \text{ definiert ist.} \end{cases}$$

$$(((Q^{(2)},E,\delta^{(2)},q_0^{(2)},F^{(2)},\text{result}^{(2)},\text{conditions}^{(2)}),q^{(2)}),\kappa^{(2)})$$

$$:= \begin{cases} (Q^{(1)},E,\delta^{(1)},q_0^{(1)},F^{(1)},\text{result}^{(1)},\text{conditions}^{(1)}), \\ \quad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega'' \to v_2]) \text{ nicht definiert ist,} \\ \\ \Gamma_V(\text{prev\_use}(v,\omega'',\beta), \omega_{\text{deref}}, v, \\ (Q^{(1)} \cup \{q^2\}, E, \delta^{(1)}[q \overset{[\omega'' \to v_2]}{\to} q^2], q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^2), \\ \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega'' \to v_2]) \\ \cup\{((Q^{\omega''},E,\delta^{\omega''},q_0^{\omega''},F_{v_2}^{\omega''},\text{result}^{\omega''},\text{conditions}^{\omega''}),q_0^{\omega''})\},\kappa^{(1)}), \\ \quad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\},[\omega'' \to v_2]) \text{ definiert ist.} \end{cases}$$

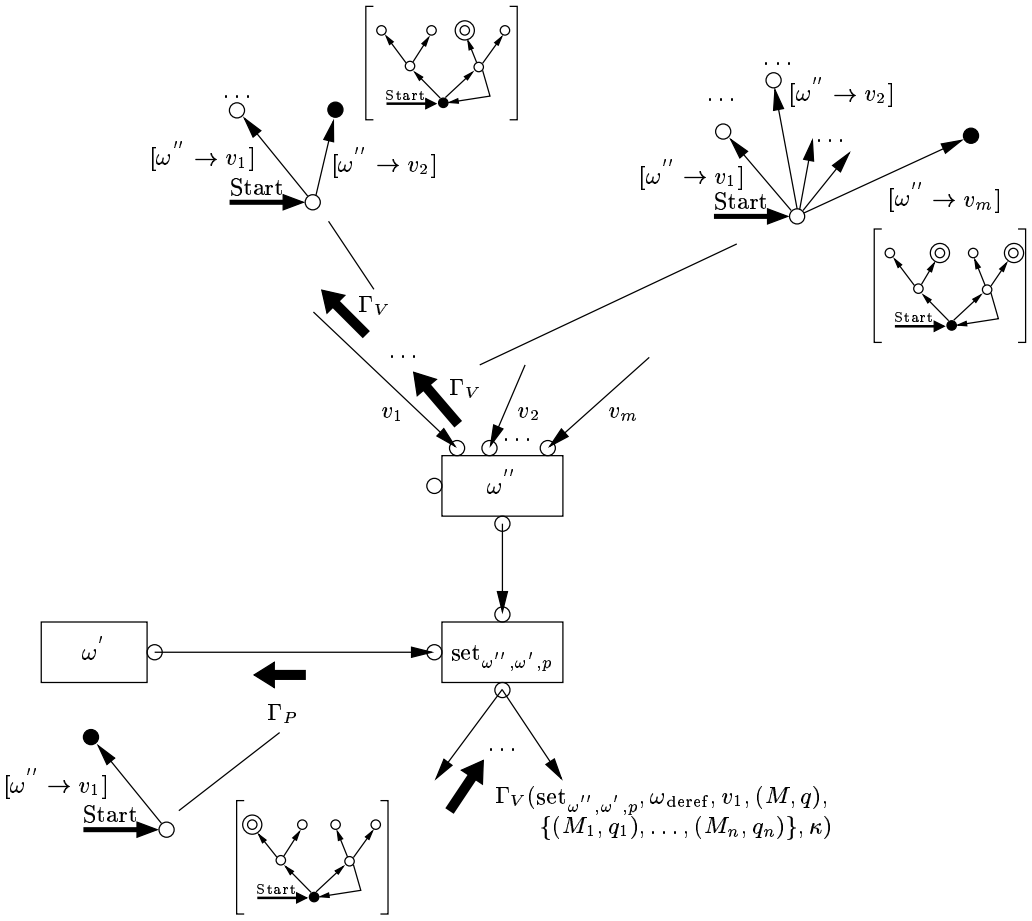


Abbildung 7.16: Query–Verarbeitung an Set–Operationen, die von Dereferenzierungsoperationen abhängen.

...

$$(((Q^{(m)}, E, \delta^{(m)}, q_0^{(m)}, F^{(m)}, \text{result}^{(m)}, \text{conditions}^{(m)}), q^{(m)}), \kappa^{(m)})$$

$$:= \begin{cases} (Q^{(m-1)}, E, \delta^{(m-1)}, q_0^{(m-1)}, F^{(m-1)}, \text{result}^{(m-1)}, \text{conditions}^{(m-1)}), \\ \quad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\}, [\omega'' \to v_m]) \text{ nicht definiert ist}, \\ \\ \Gamma_V(\text{prev\_use}(v, \omega'', \beta), \omega_{\text{deref}}, v, \\ (Q^{(m-1)} \cup \{q^m\}, E, \delta^{(m-1)}[q \overset{[\omega'' \to v_m]}{\to} q^m], q_0^{(m-1)}, F^{(m-1)}, \text{result}^{(m-1)}, \text{conditions}^{(m-1)}), q^m), \\ \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\}, [\omega'' \to v_m]) \\ \cup\{((Q^{\omega''}, E, \delta^{\omega''}, q_0^{\omega''}, F_{v_m}^{\omega''}, \text{result}^{\omega''}, \text{conditions}^{\omega''}), q_0^{\omega''})\}, \kappa^{(m-1)}), \\ \quad \text{falls } \delta^*(\{(M_1,q_1),\ldots,(M_n,q_n)\}, [\omega'' \to v_m]) \text{ definiert ist}. \end{cases}$$

Das Prinzip der Definition ist dabei das gleiche wie bei der Definition der Querybehandlung bei Get–Operationen, die von einer Dereferenzierungsoperation abhängen. Bei der hier gegebenen Definition wird ebenfalls eine Menge von weiteren Queries mit neuen Zuständen in ihren Protokollautomaten und mit zusätzlichen betriebenen Automaten als Querybedingungen generiert und aus deren Ergebnissen das Ergebnis der Queryfunktion Γ_V definiert. Bei der Set–Operation ergeben sich allerdings dazu unterschiedliche Interpretationen der Auswirkung einer Set–Operation in den Fällen, daß die Dereferenzierungsoperation diejenige Variable, nach deren letzter Zuweisung gesucht

wird, als Ziel besitzt bzw. nicht zum Ziel besitzt. Falls die Dereferenzierungsoperation diese Variable zum Ziel hat, so wird dieser Variablen von der Set–Operation ein Wert zugewiesen. In diesem Fall sind die möglichen Belegungen der Variablen also mittels einer Γ_P–Query zu ermitteln. Dies ist in Abbildung 7.16 dargestellt. Unter der angenommenen Voraussetzung, daß die Dereferenzierungsoperation ω'' die Variable v_1, für deren letzte Zuweisung sich die Query Γ_V interessiert, als Ziel besitzt, wird die Menge der möglichen Inhalte von v_1 durch die eingezeichnete Query Γ_P berechnet. In den anderen Fällen hat die Set–Operation keinen Einfluß auf die Variable, da die Dereferenzierungsoperation explizit eine andere Variable als Ziel besitzt. In diesen Fällen wird analog zum Vorgehen bei der entsprechenden Get–Operation die Berechnung der Query Γ_V durch weitere Γ_V–Queries mit den entsprechenden Querybedingungen, allerdings jeweils entlang der Use–Use–Chain von v_1, geleistet. Wie in Abbildung 7.14 sind die betriebenen Automaten, die als Querybedingungen hinzugefügt werden, in eckigen Klammern beispielhaft angedeutet.

7.4.4.5 Adress–Operation

Für eine Adress–Operation $\text{addr}_v \in \Omega$ mit $\text{addr}_v \in \beta$ für ein $\beta \in \Lambda$ werden die Queryfunktionen Γ_V und Γ_P wie folgt definiert:

$$\begin{aligned}&\Gamma_V(\text{addr}_v, \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)\\ &:= \quad \Gamma_V(\text{prev\_use}(v, \text{addr}_v, \beta), \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q),\\ &\qquad \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)\end{aligned} \tag{7.9}$$

Damit wird eine solche Query Γ_V direkt zur letzten Verwendung der Variablen v weitergeleitet.

Zur Definition der Funktion Γ_P wird wiederum eine Funktion $\widehat{\Gamma_P}$ definiert, aus der analog zu Gleichung (7.4) dann die Funktion Γ_P definiert wird.

$$\begin{aligned}&\widehat{\Gamma_P}(\text{addr}_v, \omega_{\text{deref}}, p, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)\\ &:= \begin{cases} (((Q, E, \delta, q_0, F \cup \{q\}, \text{result}[q \to v], \text{conditions}[q \to \{(M_1, q_1), \ldots, (M_n, q_n)\}]), q), \kappa), \\ \qquad \text{falls } \exists \pi \in E* \,:\, \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, \pi) = \emptyset \\ ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \kappa), \\ \qquad \text{sonst} \end{cases}\end{aligned} \tag{7.10}$$

Eine Adress–Operation stellt damit, wie bereits informell vorgestellt wurde, eine Terminierung der Querybearbeitung dar. Je nachdem, ob die bei der Query verbliebenen Querybedingungen $\{(M_1, q_1), \ldots, (M_n, q_n)\}$ durch einen Pfad $\pi \in E*$ erfüllt, d.h. gemeinsam in einen Endzustand überführt werden können, wird der aktuelle Zustand der Query zu einem Endzustand mit entsprechender Definition der result–Funktion gemacht und die restlichen Querybedingungen durch die conditions–Abbildung ausgedrückt, oder andernfalls die Queryverarbeitung ohne Generierung eines Endzustandes beendet.

7.4.4.6 Dereferenzierungs–Operation

Für Dereferenzierungsoperationen müssen die Queryfunktionen Γ_P und Γ_V nicht definiert werden, da die Ergebnisse der Dereferenzierungsoperation in die Behandlung derjenigen Operationen einbezogen werden, die die Verwendungen derjenigen Variablen darstellen, die Ziel der Dereferenzierungsoperationen sind. Diese Verwendungen stellen auch diejenigen Operationen dar, die von Queries erreicht werden können. Da diese die Queries nicht an die Dereferenzierungsoperationen, sondern z.B. an die letzte Verwendung einer Variablen vor der Dereferenzierungsoperation weiterleiten, kann eine Dereferenzierungsoperation in keinem Fall von einer Query "betrachtet" werden müssen.

7.4.4.7 Var– und Rav–Operationen

Die Querybearbeitung für Operationen $\text{var}_v \in \Omega$– und $\text{rav}_v \in \Omega$ mit $\text{var}_v \in \beta$ bzw. $\text{rav}_v \in \beta$ für ein $\beta \in \Lambda$ und $v \in V$ läßt sich als direkte Weiterleitung der entsprechenden Queries beschreiben:

$$\Gamma_V(\text{var}_v, \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$

$$:= \Gamma_V(\text{prev\_use}(v, \text{var}_v, \beta), \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa) \quad (7.11)$$

$$\Gamma_V(\text{rav}_v, \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$
$$:= \Gamma_V(\text{prev\_use}(v, \text{rav}_v, \beta), \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa) \quad (7.12)$$

7.4.4.8 Split–Operationen

Analog werden die Queryfunktionen für Split–Operationen einfach an die letzten Verwendungen der entsprechenden Variablen weitergeleitet. Bei $\text{split}^{\texttt{if}}_{\ldots}$–Operationen besteht die Möglichkeit, daß verschiedene Queries über verschiedene Analysegraphkanten an der split–Operation zusammentreffen. Deshalb muß an diesen Operationen eine Überprüfung auf Wiederholungen integriert werden. Wie für einige Operationen bisher, wird für eine $\text{split}^{\ldots}_{v,\beta_1,\beta_2}$–Operation anstatt der Funktion Γ_V eine Funktion $\widehat{\Gamma_V}$ definiert, aus der dann analog zu Gleichung (7.3) die Funktion Γ_V definiert werden kann. Für Operationen $\text{split}^{\texttt{if}}_{v,\beta_1,\beta_2} \in \Omega$ und $\text{split}^{\texttt{while}}_{v,\beta} \in \Omega$ mit $\text{split}^{\texttt{if}}_{v,\beta_1,\beta_2} \in \beta$ bzw. $\text{split}^{\texttt{while}}_{v,\beta} \in \beta$ für ein $\beta \in \Lambda$ wird also definiert:

$$\widehat{\Gamma_V}(\text{split}^{\texttt{if}}_{v,\beta_1,\beta_2}, \omega_{\text{deref}}, v, (\underbrace{(Q, E, \delta, q_0, F, \text{result}, \text{conditions})}_{=:M}, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$

$$:= \begin{cases} ((Q, E, \delta[q \xrightarrow{\epsilon} q'], q_0, F, \text{result}, \text{conditions}), \kappa), \\ \qquad \text{falls } \kappa(\text{split}^{\texttt{if}}_{v,\beta_1,\beta_2}, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) = (M', q') \\ \Gamma_V(\text{prev\_use}(v, \text{split}^{\texttt{if}}_{v,\beta_1,\beta_2}, \beta), \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \\ \quad \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa[(\text{split}^{\texttt{if}}_{v,\beta_1,\beta_2}, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) \to (M, q)]), \\ \qquad \text{sonst} \end{cases} \quad (7.13)$$

$$\Gamma_V(\text{split}^{\texttt{while}}_{v,\beta}, \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$
$$:= \Gamma_V(\text{prev\_use}(v, \text{split}^{\texttt{while}}_{v,\beta}, \beta), \omega_{\text{deref}}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa) \quad (7.14)$$

7.4.4.9 Join–Operationen

Join–Operationen stellen, wie bereits in der informellen Vorstellung dargestellt, diejenigen Operationen dar, die für die Protokollierung der von einer Query durchlaufenen Pfade zuständig sind. Wiederum werden im Folgenden Funktionen $\widehat{\Gamma_V}$ definiert, aus denen analog zu Gleichung (7.3) die Funktion Γ_V abgeleitet wird. Für Operationen $\text{join}^{\texttt{if}}_{v,\beta_1,\beta_2} \in \Omega$ wird mit zwei neuen Zuständen q^1 und q^2, die in Q nicht vorkommen sollen, definiert:

$$\widehat{\Gamma_V}(\text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}, \omega_{\text{deref}}, v, (\underbrace{(Q, E, \delta, q_0, F, \text{result}, \text{conditions})}_{=:M}, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa)$$
$$:= (((Q^{(2)}, E, \delta^{(2)}, q_0^{(2)}, F^{(2)}, \text{result}^{(2)}, \text{conditions}^{(2)}), q^{(2)}), \kappa^{(2)}), \quad (7.15)$$

wobei

$$(((Q^{(1)}, E, \delta^{(1)}, q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^{(1)}), \kappa^{(1)})$$
$$:= \begin{cases} \Gamma_V(\text{cond\_prev\_use}(v, \text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}, \beta_1, \text{exit}(\beta_1)), \omega_{\text{deref}}, v, \\ (Q \cup \{q^1\}, E, \delta[q \xrightarrow{\text{exit}(\beta_1)} q^1], q_0, F, \text{result}, \text{conditions}), q^1), \\ \{(M_1', q_1'), \ldots, (M_m', q_m')\}, \kappa), \\ \qquad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, \text{exit}(\beta_1)) = \{(M_1', q_1'), \ldots, (M_m', q_m')\} \\ \qquad \text{für ein } m \in \mathbb{N}_0 \\ (((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \kappa) \\ \qquad \text{sonst} \end{cases} \quad (7.16)$$

$$
\begin{aligned}
&(((Q^{(2)}, E, \delta^{(2)}, q_0^{(2)}, F^{(2)}, \text{result}^{(2)}, \text{conditions}^{(2)}), q^{(2)}), \kappa^{(2)}) \\
&:= \begin{cases}
\Gamma_V(\text{cond\_prev\_use}(v, \text{join}^{\texttt{if}}_{v,\beta_1,\beta_2}, \beta, \text{exit}(\beta_2)), \omega_{\text{deref}}, v, \\
(Q^{(1)} \cup \{q^2\}, E, \delta^{(1)}[q \overset{\text{exit}(\beta_2)}{\rightarrow} q^2], q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^2), \\
\{(M_1^{''}, q_1^{''}), \ldots, (M_l^{''}, q_l^{''})\}, \kappa^{(1)}), \\
\quad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, \text{exit}(\beta_2)) = \{(M_1^{''}, q_1^{''}), \ldots, (M_l^{''}, q_l^{''})\} \\
\quad \text{für ein } l \in \mathbb{N}_0 \\
(((Q^{(1)}, E, \delta^{(1)}, q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^{(1)}), \kappa^{(1)}) \\
\quad \text{sonst}
\end{cases}
\end{aligned} \tag{7.17}
$$

Analog wird für Operationen $\text{join}^{\texttt{while}}_{v,\beta_1} \in \Omega$ mit $\text{join}^{\texttt{while}}_{v,\beta_1} \in \beta$ für ein $\beta \in \Lambda$ und zwei neuen Zuständen q^1 und q^2, die in Q nicht vorkommen sollen, und im Gegensatz zur $\text{join}^{\texttt{if}}_{\ldots}$–Operation mit zusätzlicher Wiederholungserkennung definiert:

$$
\begin{aligned}
&\widehat{\Gamma_V}(\text{join}^{\texttt{while}}_{v,\beta_1}, \omega_{\text{deref}}, v, (\underbrace{(Q, E, \delta, q_0, F, \text{result}, \text{conditions})}_{=:M}, q), \{(M_1, q_1), \ldots, (M_n, q_n)\}, \kappa) \\
&:= \begin{cases}
(((Q, E, \delta[q \overset{\epsilon}{\rightarrow} q^{'}], q_0, F, \text{result}, \text{conditions}), q), \kappa), \\
\quad \text{falls } \kappa(\text{join}^{\texttt{while}}_{v,\beta_1}, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) = (M^{'}, q^{'}) \\
(((Q^{(2)}, E, \delta^{(2)}, q_0^{(2)}, F^{(2)}, \text{result}^{(2)}, \text{conditions}^{(2)}), q^{(2)}), \kappa^{(2)}), \\
\quad \text{sonst}
\end{cases}
\end{aligned} \tag{7.18}
$$

wobei

$$
\begin{aligned}
&(((Q^{(1)}, E, \delta^{(1)}, q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^{(1)}), \kappa^{(1)}) \\
&:= \begin{cases}
\Gamma_V(\text{cond\_prev\_use}(v, \text{join}^{\texttt{while}}_{v,\beta_1}, \beta, \text{exit}(\beta_1)), \omega_{\text{deref}}, v, \\
((Q \cup \{q^1\}, E, \delta[q \overset{\text{exit}(\beta_1)}{\rightarrow} q^1], q_0, F, \text{result}, \text{conditions}), q^1), \\
\{(M_1^{'}, q_1^{'}), \ldots, (M_m^{'}, q_m^{'})\}, \\
\kappa[(\text{join}^{\texttt{while}}_{v,\beta_1}, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) \rightarrow (M, q)]), \\
\quad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, \text{exit}(\beta_1)) = \{(M_1^{'}, q_1^{'}), \ldots, (M_m^{'}, q_m^{'})\} \\
\quad \text{für ein } m \in \mathbb{N}_0 \\
(((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \\
\kappa[(\text{join}^{\texttt{while}}_{v,\beta_1}, \omega_{\text{deref}}, v, \emptyset, \{(M_1, q_1), \ldots, (M_n, q_n)\}) \rightarrow (M, q)]) \\
\quad \text{sonst}
\end{cases} \\
&(((Q^{(2)}, E, \delta^{(2)}, q_0^{(2)}, F^{(2)}, \text{result}^{(2)}, \text{conditions}^{(2)}), q^{(2)}), \kappa^{(2)}) \\
&:= \begin{cases}
\Gamma_V(\text{cond\_prev\_use}(v, \text{join}^{\texttt{while}}_{v,\beta_1}, \beta, \overline{\text{exit}}(\beta_1)), \omega_{\text{deref}}, v, \\
(Q^{(1)} \cup \{q^2\}, E, \delta^{(1)}[q \overset{\overline{\text{exit}}(\beta_1)}{\rightarrow} q^2], q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^2), \\
\{(M_1^{''}, q_1^{''}), \ldots, (M_l^{''}, q_l^{''})\}, \kappa^{(1)}), \\
\quad \text{falls } \delta^*(\{(M_1, q_1), \ldots, (M_n, q_n)\}, \overline{\text{exit}}(\beta_1)) = \{(M_1^{''}, q_1^{''}), \ldots, (M_l^{''}, q_l^{''})\} \\
\quad \text{für ein } l \in \mathbb{N}_0 \\
(((Q^{(1)}, E, \delta^{(1)}, q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^{(1)}), \kappa^{(1)}) \\
\quad \text{sonst}
\end{cases}
\end{aligned} \tag{7.19, 7.20}
$$

In beiden Definitionen wird überprüft, ob alle als Querybedingungen mitgeführten betriebenen Automaten einen Zustandsübergang mit dem Symbol durchführen können, das die entsprechende Verzweigung charakterisiert. Die Protokollierung in der Funktion κ, daß diese Query mit den angetroffenen Querybedingungen hier bereits bearbeitet wurde, geschieht in dieser Definition vor der Bearbeitung der ersten weitergeleiteten Query. Dadurch kann eine Wiederholung nicht erst nach kompletter Abarbeitung der Queryverarbeitung erkannt werden.

7.5 Beispiel für die Queryverarbeitung

Die bisher vorgestellte Queryverarbeitung soll nun auf das Running Example angewandt werden. Dazu wird in Abbildung 7.17 die Analysegraphdarstellung der neuen Programmrepräsentation verwendet, um die Queryverarbeitung anhand der Definitionen für die rekursiven Funktionen Γ_V und Γ_P zu veranschaulichen.

7.5.1 Informelle Beschreibung

An den numerierten Positionen im Graphen in Abbildung 7.17 werden die folgenden Berechnungsschritte bei der Querybearbeitung durchgeführt:

1. Der prinzipielle Query–Algorithmus besagt, daß an jeder Dereferenzierungsoperation, die in absteigender Reihenfolge der Referenzierungsstufe der möglichen Zielvariablen verarbeitet werden, eine Query mit einem Automaten, der nur aus seinem Startzustand besteht, initiiert wird. Entsprechend beginnt im Beispiel die Queryverarbeitung mit der Dereferenzierungs–Operation deref_1, von der aus eine Query, mit Γ_P bezeichnet, entlang der Kante zur Get–Operation links davon geschickt wird. Der dabei zur Protokollierung mitgeführte Automat ist in Abbildung 7.18(a) dargestellt.

2. Die Query Γ_P erreicht eine Get–Operation, d.h. daß der gesuchte Wert aus der Variablen x ausgelesen wird. Entsprechend muß die Fortsetzung der Querybearbeitung nun die letzte Zuweisung zur Variablen x finden, was die Aufgabe der eingezeichneten Query Γ_V ist.

3. Die Query Γ_V erreicht die Join–Operation $\text{join}^{\texttt{while}}_{x,\beta_1}$, d.h. je nachdem, ob eine Programminstanz den Schleifenrumpf umgangen oder betreten hat, muss entlang von verschiedenen Abschnitten der Use–Use–Chain nach der letzten Zuweisung zu x gesucht werden. Daher wird je eine Query Γ_V entlang der verschiedenen möglichen Kanten geschickt, und dabei im Protokollautomaten vermerkt, welche Verzweigung die jeweilige Query gewählt hat. Als erstes wird die Möglichkeit untersucht, daß der Schleifenrumpf betreten wurde. Entsprechend wird in den Protokollautomat das Eingabesymbol $\text{exit}(\beta_1)$ integriert. Das Ergebnis dieser Aktion ist in Abbildung 7.18(b) gezeigt. Der aktuelle Zustand des Protokollautomaten wird dabei auf den neu hinzugenommenen Zustand q_1 gesetzt.

4. Eine Rav–Operation besitzt keine Funktionalität, also leitet sie die Query nur weiter an die im Graphen darüberliegende Join–Operation $\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}$.

5. An dieser Join–Operation wird wie oben beschrieben verfahren. Zunächst wird eine Query in den Block β_2 geschickt und dies ebenfalls im Automaten protokolliert, was in Abbildung 7.18(c) dargestellt ist.

6. Das Erreichen einer Set–Operation bedeutet das Erreichen der letzten Zuweisung zu x.

7. Interessant ist nun für die Query, was für einen Inhalt der benannte Wert hat, der der Variablen x dort zugewiesen wird. Der bei der Ausführung der Set–Operation konsumierte benannte Wert wurde von derjenigen Operation produziert, die über die entsprechende Kante mit der Set–Operation verbunden ist. Damit läßt er sich durch eine Query Γ_P entlang dieser Kante ermitteln.

8. Beim Erreichen einer Adressoperation "von rechts" wird ein Ergebnis gefunden. Der Wert, der hier produziert wird, entspricht immer (der Adresse) der Variablen z. Der Rückwärtsfluß der Query beschreibt bei Interpretation in Vorwärtsrichtung eine "Bahn", in der dieser Wert bei Durchlaufen des zugehörigen Pfades stets die Dereferenzierungsoperation erreicht, und diese damit die gefundene Variable z als Ziel ihrer Dereferenzierung besitzt.

   Dieses Ergebnis wird im Automaten durch Erzeugen eines Endzustandes vermerkt, dem durch die result–Abbildung die Variable z zugewiesen wird. Dieser Automat ist in Abbildung 7.18(d) zu sehen.

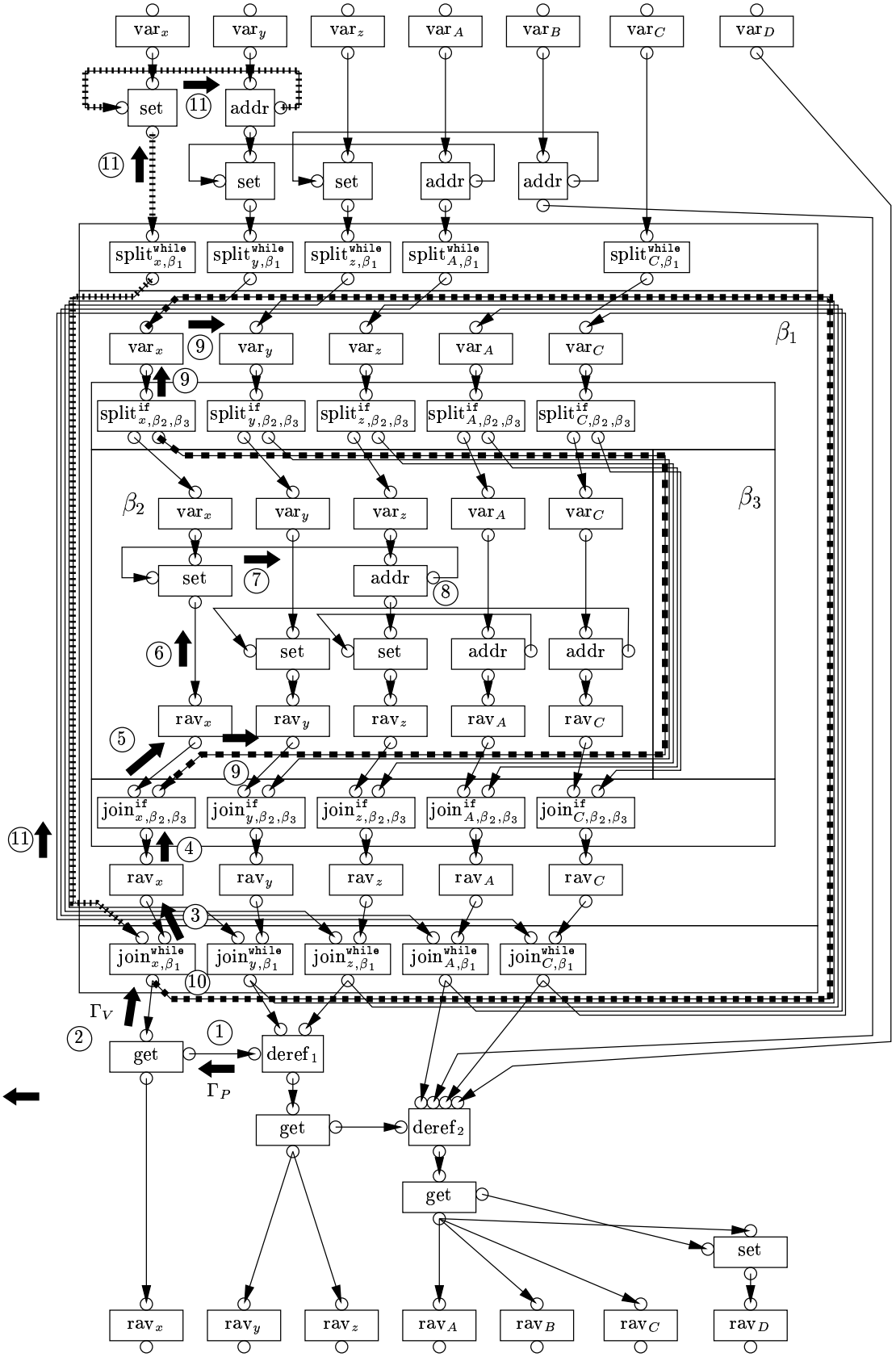


Abbildung 7.17: Queryverarbeitung im Running Example.

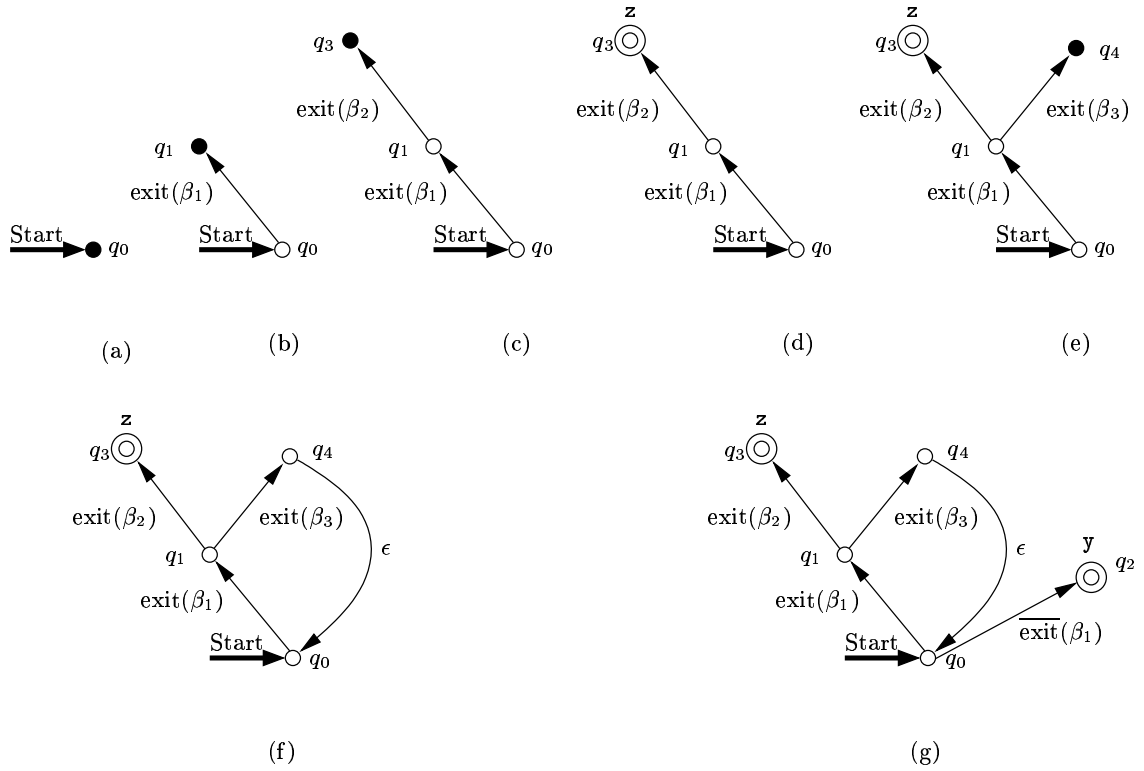


Abbildung 7.18: Queryverarbeitung im Running Example: Aufbau des Protokollautomaten.

9. Bisher wurde in Schritt 5 erst die erste Verzweigungsmöglichkeit untersucht. Nach Abschluß der entsprechenden Querybehandlung ist nun die zweite Alternative, der Eintritt der Query in den leeren else–Zweig, Ziel der Untersuchung. Die entsprechende Protokollierung im Automaten ist in Abbildung 7.18(e) dargestellt.

10. Auf dem gestrichelt eingezeichneten Weg der Query erreicht diese erneut die Join–Operation, an der die Query bereits in Schritt 2 bzw. 3 bearbeitet wurde. Als Teil des Algorithmus wird wie vorgestellt an bestimmten Operationen protokolliert, welche Queries für welche Dereferenzierungsoperationen mit welchen mitgeführten Querybedingungen in welchem aktuellen Zustand des Protokollautomaten bereits bearbeitet wurden. Im aktuellen Schritt werden keine Querybedingungen mitgeführt, ebenso waren zum Zeitpunkt der ersten Bearbeitung der Query keine Querybedingungen zu berücksichtigen. Entsprechend kann der gestrichelte Pfadabschnitt beliebig häufig wiederholt werden, ohne daß sich dabei das Ergebnis der Queryberechnung verändern würde. Dies wird im Automaten durch Generierung eines ϵ–Zustandsübergangs zu demjenigen Zustand, in dem sich der Protokollautomat der Query bei der ersten Bearbeitung befunden hat, beschrieben. Der sich dabei ergebende Automat ist in Abbildung 7.18(f) dargestellt.

11. Die in Schritt 3 begonnene Verarbeitung der Query an der dortigen Join–Operation wird fortgesetzt und erreicht entlang der zweiten gestrichelten Linie analog zu der bisher vorgestellten Vorgehensweise ebenfalls ein Ergebnis. Diesmal wird allerdings die Adresse der Variablen y mit dem Ergebniszustand assoziiert, was in Abbildung 7.18(g) dargestellt ist.

Die bisherige Analyse entsprach einer Analyse von Single–Level–Zeigern, da für die untersuchte Variable keine Abhängigkeiten von anderen Dereferenzierungsoperationen bestanden haben. Im weiteren wird bei der Querybearbeitung, ausgehend von der zweiten Dereferenzierungsoperation, eine solche Abhängigkeit gefunden werden, und die Analyse wird als Analyse von Multi–Level–Zeigern fortgeführt. Dieser zweite Teil der Analyse ist in Abbildung 7.19 dargestellt. Dabei werden die folgenden Schritte zur Queryverarbeitung unternommen.

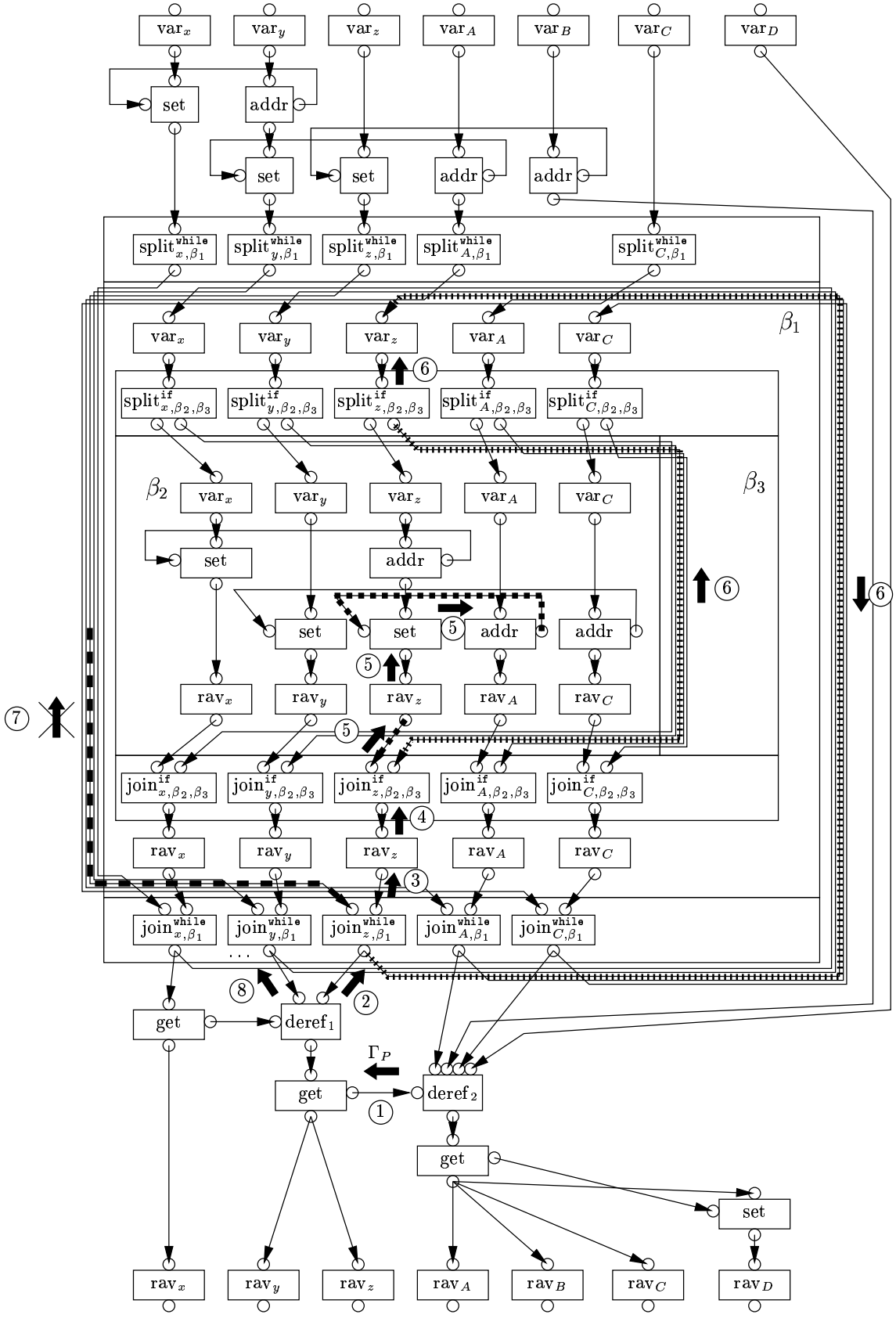


Abbildung 7.19: Queryverarbeitung mit Abhängigkeiten im Running Example.

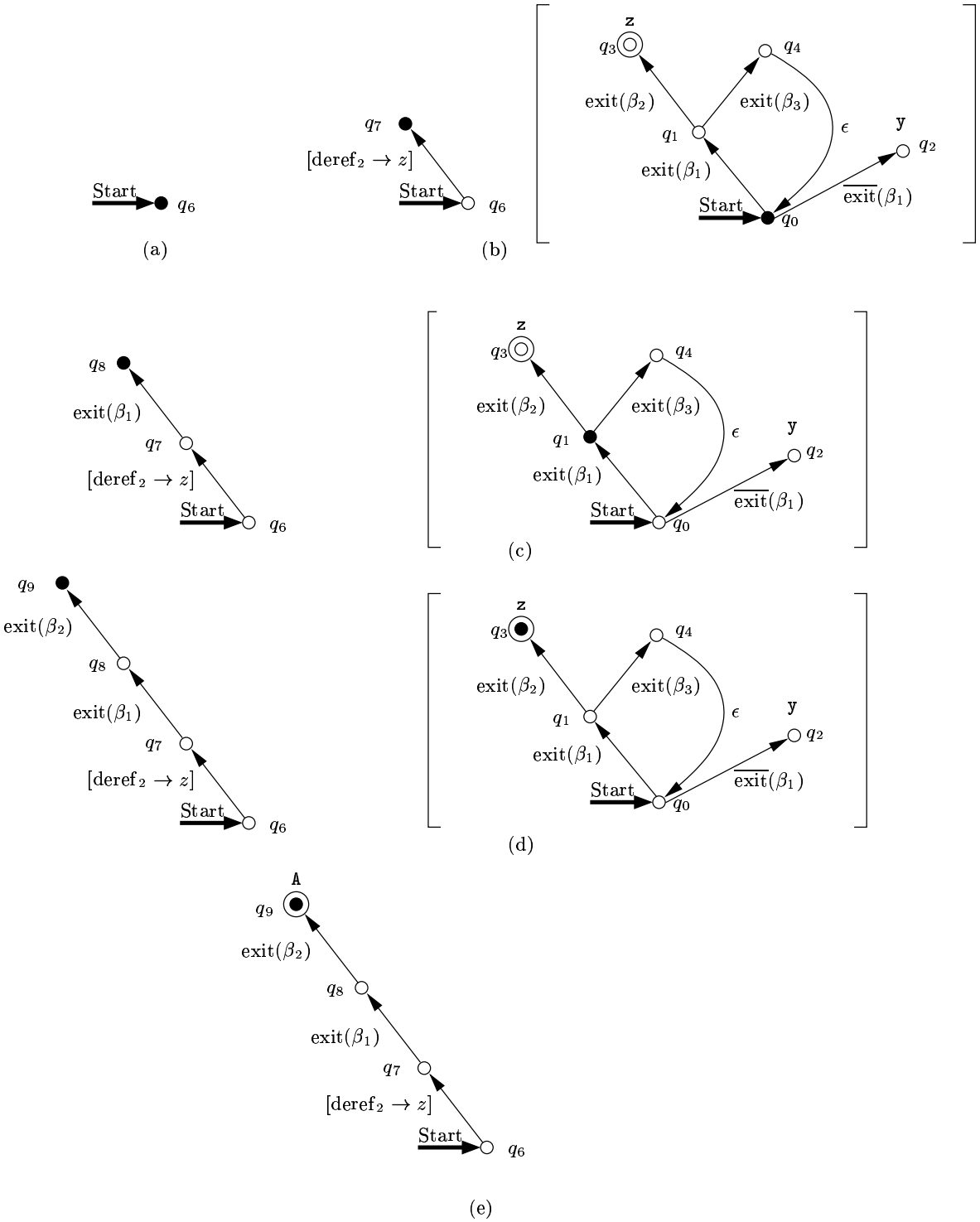


Abbildung 7.20: Queryverarbeitung im Running Example mit Abhängigkeiten: Aufbau des Protokollautomaten und betriebene Automaten.

1. Von der zweiten Dereferenzierungsoperation deref_2 wird nach Beenden der Queryverarbeitung für die erste Dereferenzierungsoperation deref_1 ebenfalls eine Query initiiert, deren Protokollautomat nur aus seinem Startzustand besteht, in Abbildung 7.20(a) dargestellt. Zur besseren Unterscheidung werden im Beispiel die Zustände beispielweit fortlaufend numeriert, d.h. der Startzustand des abgebildeten Automaten wird hier mit q_6 bezeichnet.

   Die Query erreicht eine Get–Operation. Anders als vorher liest diese aber nicht immer den Wert einer bestimmten Variablen aus, sondern derjenigen, die das Ergebnis der Dereferenzierungsoperation deref_1 ist. Die möglichen Ziele dieser Dereferenzierungsoperation sind aber bereits bekannt, ebenso der Protokollautomat, der eine Pfadbedingung für diese Ergebnisse beschreibt.

2. In der Beispielverarbeitung wird als erstes von der Annahme ausgegangen, daß die Dereferenzierungsoperation die Variable z als Ziel besitzt. Damit liest die get–Operation unter dieser Annahme den Inhalt der Variablen z aus, und es muß zur Weiterberechnung der Query die letzte Zuweisung zur Variablen z mittels einer Query Γ_V gesucht werden. Diese Annahme wird durch Integration eines entsprechenden Eingabesymbols $[\text{deref}_1 \rightarrow z]$ protokolliert (Abb. 7.20(b) links).

   Als weitere Konsequenz aus der Annahme wird der Query der Protokollautomat der bereits abgeschlossenen Query für die Dereferenzierungsoperation deref_1 zur Beschränkung des weiteren Queryflusses hinzugefügt. Als Modifikation gegenüber dem ursprünglichen Ergebnis enthält dieser als Querybedingung mitgeführte Automat aber nur diejenigen Endzustände, deren result–Abbildung die Variable z ergibt.

   Dieser Automat, der in Abbildung 7.20(b) in eckigen Klammern dargestellt ist, wird ab diesem Zeitpunkt von der Query bei der Verarbeitung mitgeführt und dient der Beschränkung des Queryflusses auf eine Teilmenge der möglichen Pfade im Sinne eines bedingten Erreichbarkeitsproblems.

3. Bei Erreichen der Join–Operation $\text{join}^{\texttt{while}}_{z,\beta_1}$ wird die Query zunächst in den Schleifenrumpf gesendet. Dies wird im Protokollautomaten vermerkt (Abb. 7.20(c)). Ebenso wird für alle als Querybedingungen mitgeführten betriebenen Automaten (in eckigen Klammern) ein Zustandsübergang mit dem gleichen Symbol $\text{exit}(\beta_1)$ durchgeführt. Dies ist mit diesem Symbol in der aktuellen Situation möglich. Auf den anderen Fall wird weiter unten noch eingegangen werden.

   Die zwischenzeitlich erreichte Rav–Operation leitet die Query auch im Fall von mitgeführten Querybedingungen nur weiter.

4. Von der zweiten Join–Operation $\text{join}^{\texttt{if}}_{z,\beta_2,\beta_3}$ aus wird zunächst eine Query in den then–Zweig, also den Block β_2 geschickt. Auch dies wird wieder im Protokollautomaten vermerkt (Abb. 7.20(d)). Ebenso führt der als Querybedingung mitgeführte betriebene Automat in eckigen Klammern einen Zustandsübergang mit diesem Symbol aus. Dabei erreicht dieser einen Endzustand. Dies bedeutet, daß die Querybedingung durch den bisher von der Query durchlaufenen Pfadabschnitt bereits erfüllt ist, im Beispiel also die Dereferenzierungsoperation deref_1 bereits garantiert das Ziel z besitzt.

   Ein als Querybedingung mitgeführter Automat, der einen Endzustand erreicht, muß bei der weiteren Queryverarbeitung nicht mehr mitgeführt werden, er kann entfernt werden.

5. Bei der Fortsetzung der Queryverarbeitung wird wie im Fall von Single–Level–Zeigeranalyse eine Adress–Operation erreicht und damit ein Ergebnis gefunden, das durch einen Endzustand beschrieben wird (Abb. 7.20(e)). Da im letzten Schritt der als Querybedingung mitgeführte betriebene Automat entfernt wurde, besitzt die Query zum Zeitpunkt der Endzustandserzeugung keine Querybedingungen mehr. Damit ist dieses Ergebnis ohne weitere Untersuchungen korrekt. Andernfalls würde von der Queryverarbeitung ein *bedingter Endzustand* erzeugt werden,

bei dem die weiter mitgeführten Querybedingungen zunächst auf Vereinbarkeit untersucht werden müssten. Dies wurde in Abschnitt 7.3.5 bereits informell vorgestellt und bei der formalen Definition der Queryverarbeitung genau eingeführt.

6. Bei der Bearbeitung der zweiten Verzweigungsmöglichkeit an der Join–Operation aus Schritt 4 wird eine Query in den leeren else–Zweig geschickt und diese Verzweigung, wie in Abbildung 7.20(f) gezeigt, in den Protokollautomaten integriert. Der als Querybedingung mitgeführte Automat (an der Verzweigung hatte dieser noch nicht seinen Endzustand erreicht) führt ebenfalls einen Zustandsübergang mit dem Eingabesymbol exit(β_3) durch, was durch den anschließend möglichen ϵ–Zustandsübergang ein Erreichen des Zustandes q_0 bewirkt.

   Die gestrichelte Linie zeigt den weiteren Fluß der Query. Dabei wird eine Join–Operation erreicht, die bereits in Schritt 2 von der Query betrachtet wurde. Wie im Fall von Single–Level–Zeigern könnte man in dieser Situation eine Wiederholung feststellen und durch einen ϵ–Zustandsübergang im Protokollautomaten beschreiben. Im Fall von mitgeführten Querybedingungen ist es aber zusätzlich notwendig, daß sich auch sämtliche mitgeführten Querybedingungen im gleichen Zustand wie bei der ersten Bearbeitung der Query befinden. Dies ist aber im Beispiel erfüllt, da sich der als Querybedingung mitgeführte Automat beidesmal im Zustand q_0 befindet bzw. befunden hat. Damit kann ein entsprechender ϵ–Zustandsübergang erzeugt werden und es ergibt sich der Automat aus Abbildung 7.20(g).

7. Als letzte übriggebliebene Verzweigung besteht für die Query noch die Möglichkeit, von der in Schritt 2 erreichten Join–Operation aus entlang der dort eingezeichneten gestrichelten Linie den Schleifenrumpf zu umgehen. Dies würde den Protokollautomat und den als Querybedingung mitgeführten Automaten wie in Abbildung 7.20(h) gezeigt verändern. Durch den Zustandsübergang des als Querybedingung mitgeführten Automaten erreicht dieser allerdings den Zustand q_2, von dem aus der Endzustand nicht mehr erreichbar ist. Damit ist entlang dieses Pfades die Annahme, daß die Dereferenzierungsoperation deref_1 die Variable z als Ziel besitzen soll, nicht mehr erfüllbar. Entsprechend muß die Queryberechnung an dieser Stelle nicht mehr fortgesetzt werden, was durch den durchgestrichenen Query–Pfeil symbolisiert wird. Bei Fortsetzung der Query hätte man als Ergebnis oberhalb der Schleife die Adresse von B gefunden, was nach der Argumentation in der Motivation eine ungültige Kombination von Fakten dargestellt hätte.

8. Analog zur bisherigen Vorgehensweise wird die Queryverarbeitung auch unter der Annahme, daß die Dereferenzierungsoperation deref_1 die Variable y als Ziel besitzt, fortgeführt. Das komplette Ergebnis nach Abarbeitung dieser Möglichkeit ist in Abbildung 7.20(i) dargestellt. Dabei wurde nur A als mögliches Ziel von deref_2, also der zweifachen Dereferenzierung von x gefunden, was für das Beispiel ein exaktes Ergebnis darstellt.

Das Ergebnis der Queryberechnung in Abbildung 7.20(i) ist strukturell identisch zu dem Bedingungsautomaten, der in Abbildung 5.14(h) dargestellt wurde. Offensichtlich sind die Queryfunktionen in der Lage, zum einen Pfadbedingungen für Programmeigenschaften zu berechnen, und zum anderen diese berechneten Pfadbedingungen bereits bei ihrer Entstehung im Sinne eines Bedingungsautomaten zu kombinieren. Auf diese Eigenschaft der Queryberechnung wird in Kapitel 8 genauer eingegangen werden.

7.5.2 Berechnung anhand der formalen Darstellung der Query–Funktionen

Die einzelnen Schritte der Queryberechnung anhand der formalen Darstellung der Queryfunktionen sind wie bei der informellen Beschreibung in Abbildung 7.17 numeriert und ergeben sich wie folgt. Da die einzelnen Operationen aus der Darstellung des Running Example als Analysegraph nicht weiter benannt sind, ergibt sich die Zuordnung, welche Operation z.B. durch get_x gemeint ist, sinngemäß aus der Darstellung des Analysegraphen, und der informellen Vorstellung der Queryverarbeitung aus dem vorhergehenden Abschnitt.

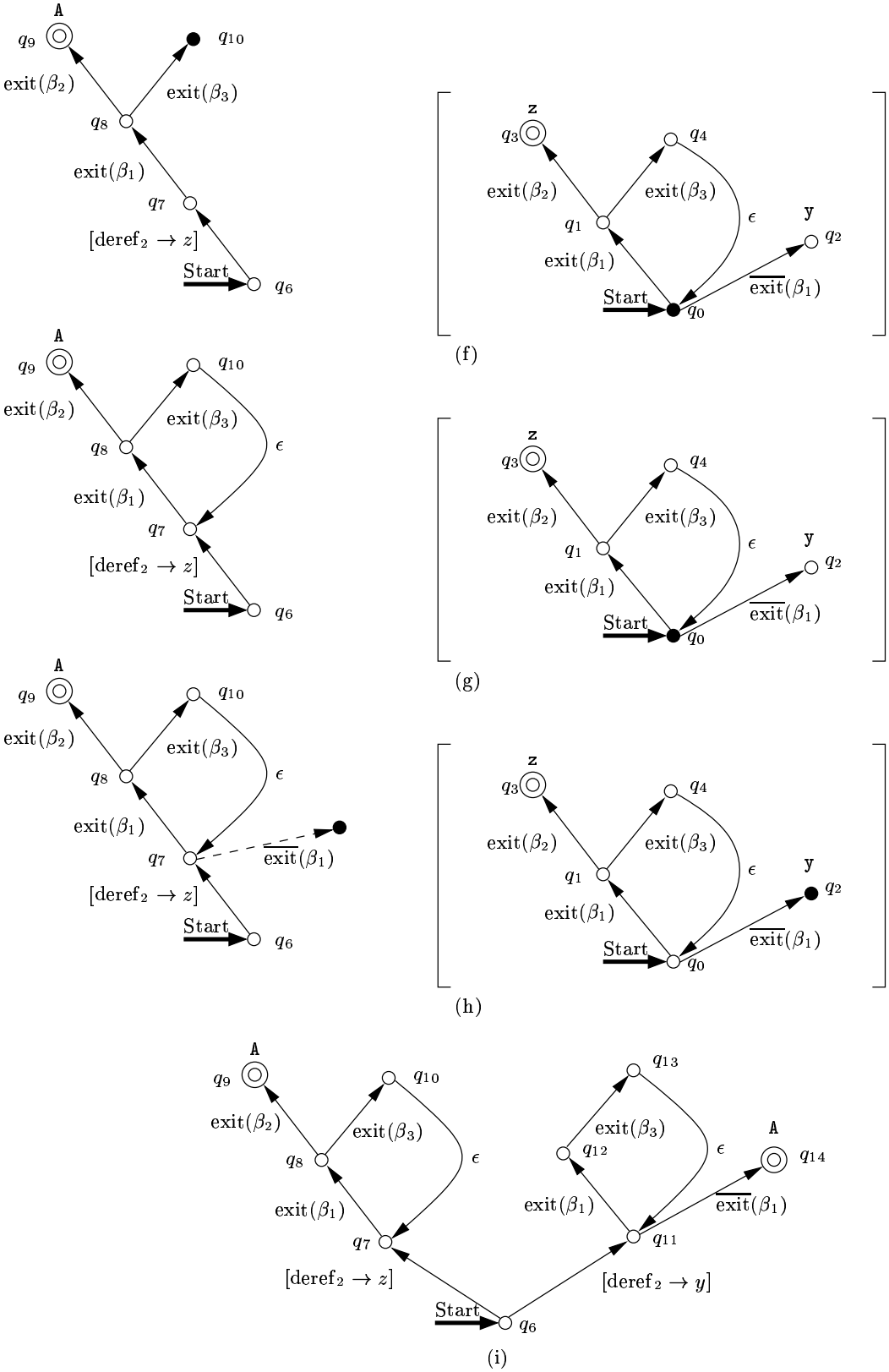


Abbildung 7.21: Queryverarbeitung im Running Example mit Abhängigkeiten: Betriebene Automaten (Teil 2).

1. Laut der Definition des Query–Algorithmus in Abschnitt 7.4.3 werden für jede Dereferenzierungsoperation, in absteigender Reihenfolge der Referenzierungsstufe der möglichen Zielvariablen, die folgenden Aktionen ausgeführt. Zunächst wird ein Automat

$$M = (\{q_0\}, E, \text{undef}, q_0, \emptyset, \text{undef}, \text{undef}),$$

der nur aus seinem Startzustand besteht, erzeugt. Dieser Automat, der in Abbildung 7.18(a) dargestellt ist, und mit seinem Startzustand q_0 als aktuellem Zustand (ausgefüllter Kreis) einen betriebenen Automaten darstellt, wird als Parameter eines Aufrufs der Queryfunktion Γ_P in folgender Zuweisung verwendet:

$$((M', q'), \kappa) = \Gamma_P(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, \text{outValue}, (M, q), \emptyset, \text{undef}) \tag{7.21}$$

Der sich dabei als Ergebnis ergebende Automat M' wird danach als das Ergebnis der Query definiert.

2. Gleichung (7.4) beschreibt die weitere Vorgehensweise, um in obiger Gleichung (7.21) $\Gamma_P(\ldots)$ zu berechnen. Dabei gilt

$$\begin{aligned} &\Gamma_P(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, \text{outValue}, (M, q_0), \emptyset, \text{undef}) \\ &\overset{(7.4)}{=} \quad ((M^1, q^1, \kappa^1), \end{aligned}$$

mit

$$(M^1, q^1, \kappa^1) = \widehat{\Gamma_P}(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, \text{outValue}, (M, q_0), \emptyset, \text{undef}),$$

da

$$\text{IntegrateConditions}(\text{get}_x, (M, q_0), \emptyset, \text{False}) = (M, \{(q, \emptyset)\}),$$

denn aufgrund der leeren Menge von als Querybedingung mitgeführten Automaten wird die Schleife im Algorithmus IntegrateConditions nicht betreten. Weiter gilt

$$\begin{aligned} &\widehat{\Gamma_P}(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, \text{outValue}, (M, q_0), \emptyset, \text{undef}) \\ &\overset{(7.1)}{=} \quad \Gamma_V(\text{prev\_use}(x, \text{get}_x, \beta_0), \text{deref}_{\text{get}_x,\text{outValue}}, x, (M, q_0), \emptyset, \\ &\qquad \text{undef}[(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)] \\ &= \quad \Gamma_V(\text{prev\_use}(x, \text{get}_x, \beta_0), \text{deref}_{\text{get}_x,\text{outValue}}, x, (M, q_0), \emptyset \\ &\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\} \end{aligned}$$

Anschaulich betrachtet erreicht die Query eine Get–Operation get_x. Der dort produzierte Wert ergibt sich aus dem aktuellen Inhalt der Variablen x. Entsprechend muß an dieser Stelle die letzte Zuweisung zur Variablen x gefunden werden, was die Aufgabe der Query(funktion) Γ_V ist. Da keine Querybedingungen mitgeführt werden und daher die Funktion IntegrateConditions keine Auswirkungen hat, sowie die Funktion κ bisher undefiniert ist und daher keine Wiederholung erkannt werden kann, ergibt sich das Ergebnis der Query Γ_P direkt wie beschrieben aus der Query Γ_V, wobei zusätzlich die Funktion κ (bisher undef) wie angegeben aktualisiert wird, um eine Protokollierung der bereits an Operationen bearbeiteten Queries zu leisten.

3. Die Auswertung der Funktion prev_use aus Abschnitt 6.5.2 ergibt die folgende weitere Berechnung:

$$\begin{aligned} &\Gamma_V(\text{prev\_use}(x, \text{get}_x, \beta_0), \text{deref}_{\text{get}_x,\text{outValue}}, x, (M, q_0), \emptyset, \\ &\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\} \\ &\overset{6.5.2}{=} \quad \Gamma_V(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, (M, q_0), \emptyset, \\ &\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\} \\ &= \quad \widehat{\Gamma_V}(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, (M, q_0), \emptyset, \\ &\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\}, \end{aligned}$$

da wie bisher die Funktion IntegrateConditions keine Auswirkungen hat und sich daher hier wieder $\Gamma_V(\text{join}^{\texttt{while}}_{x,\beta_1},\ldots)$ direkt aus $\widehat{\Gamma_V}(\text{join}^{\texttt{while}}_{x,\beta_1},\ldots)$ ergibt. Zur Berechnung davon werden die in Abschnitt 7.4.4.9 geforderten Zustände, die nicht in Q enthalten sein sollen und hier konkret mit q_1 und q_2 (anstatt q^1 und q^2 in der Definition) benannt werden, eingeführt. Damit gilt nun

$$\begin{aligned}
&\widehat{\Gamma_V}(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, (M, q_0), \emptyset,\\
&\qquad \underbrace{\{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\}}_{=:\kappa})\\
&\overset{(7.18)}{=} (((Q^{(2)}, E, \delta^{(2)}, F^{(2)}, \text{result}^{(2)}, \text{conditions}^{(2)}), \kappa^{(2)}),
\end{aligned}$$

mit noch zu berechnenden $Q^{(2)},\ldots$, da $\kappa(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset)$ undefiniert ist. Das weitere Vorgehen ergibt sich wie folgt. Dabei wird der erste Fall der Definition aus Gleichung (7.19) verwendet, da aktuell die leere Menge von Automaten als Querybedingungen mitgeführt wird, die durch die Zustandsübergangsfunktion δ^* auf sich selbst abgebildet wird, weshalb der dort spezifizierte erste Fall für $m = 0$ erfüllt ist.

$$\begin{aligned}
&((((Q^{(1)}, E, \delta^{(1)}, q_0^{(1)}, F^{(1)}, \text{result}^{(1)}, \text{conditions}^{(1)}), q^{(1)})\kappa^{(1)})\\
&= \Gamma_V(\text{cond_prev_use}(x, \text{join}^{\texttt{while}}_{x,\beta_1}, \beta_0, \text{exit}(\beta_1)), \text{deref}_{\text{get}_x,\text{outValue}}, x,\\
&\quad ((Q \cup \{q_1\}, E, \text{undef}[q_0 \overset{\text{exit}(\beta_1)}{\to} q_1], q_0, \emptyset, \text{undef}, \text{undef}), q_1), \emptyset,\\
&\quad \kappa[(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)])\\
&= \Gamma_V(\text{cond_prev_use}(x, \text{join}^{\texttt{while}}_{x,\beta_1}, \beta_0, \text{exit}(\beta_1)), \text{deref}_{\text{get}_x,\text{outValue}}, x,\\
&\quad (\{q_0, q_1\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_1), \emptyset,\\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0),\\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\})
\end{aligned}$$

Der Automat, der nun in diesem Aufruf von Γ_V als Parameter übergeben wird, entspricht damit dem Automaten aus Abbildung 7.18(b), der zusätzlich zum initialen Automaten den Zustand q_1 besitzt, der vom Anfangszustand über einen Zustandsübergang bei Eingabe des Symbols $\text{exit}(\beta_1)$ erreichbar ist. Dieser Zustand q_1 ist auch aktueller Zustand (ausgefüllter Kreis) des betriebenen Automaten.

4. Einsetzen der Definition 6.22 von cond_prev_use aus Abschnitt 6.5.3 ergibt:

$$\begin{aligned}
&\Gamma_V(\text{cond_prev_use}(x, \text{join}^{\texttt{while}}_{x,\beta_1}, \beta_0, \text{exit}(\beta_1)), \text{deref}_{\text{get}_x,\text{outValue}}, x,\\
&\qquad (\{q_0, q_1\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_1),\\
&\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0),\\
&\qquad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\})\\
&\overset{6.5.3}{=} \Gamma_V(\text{rav}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x,\\
&\qquad (\{q_0, q_1\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_1),\\
&\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0),\\
&\qquad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\})\\
&\overset{(7.12)}{=} \Gamma_V(\text{prev_use}(x, \text{rav}_x, \beta_1), \text{deref}_{\text{get}_x,\text{outValue}}, x,\\
&\qquad (\{q_0, q_1\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_1),\\
&\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0),\\
&\qquad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0)\})\\
&\overset{6.5.2}{=} \Gamma_V(\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x,
\end{aligned}$$

$$(((\{q_0, q_1\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_1),$$
$$\{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0)\})$$

5. Zur Berechnung der Queryverarbeitung an dieser Join–Operation $\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}$ nach Gleichung (7.15) werden wiederum zwei neue Zustände q_3 und q_4 erzeugt und damit wie folgt vorgegangen. Wie oben werden zunächst die folgenden Teilberechnungen durchgeführt, die zur Unterscheidung mit obiger Berechnung einen anderen oberen Index als in Gleichung (7.16) erhalten:

$$\begin{aligned}
&((((Q^{(3)}, E, \delta^{(3)}, q_0^{(3)}, F^{(3)}, \text{result}^{(3)}, \text{conditions}^{(3)}), q^{(3)})\kappa^{(3)}) \\
&= \Gamma_V(\text{cond_prev_use}(x, \text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \beta_2, \text{exit}(\beta_2)), \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\quad (((\{q_0, q_1\} \cup \{q_3\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1)\}[q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3], q_0, \emptyset, \text{undef}, \text{undef}), q_3), \emptyset, \\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0)\} \\
&\quad [(\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)]) \\
&= \Gamma_V(\text{cond_prev_use}(x, \text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \beta_2, \text{exit}(\beta_2)), \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\quad (((\{q_0, q_1, q_3\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_3), \emptyset, \\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\}) \\
&\overset{6.5.3}{=} \Gamma_V(\text{rav}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\quad (((\{q_0, q_1, q_3\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_3), \emptyset, \\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\}) \\
&\overset{(7.12)}{=} \Gamma_V(\text{prev_use}(\text{rav}_x, x, \beta_2), \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\quad (((\{q_0, q_1, q_3\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_3), \emptyset, \\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})
\end{aligned}$$

Dabei ergibt sich ein Automat als Parameter der Query, wie er in Abbildung 7.18(c) abgebildet ist.

6. Aus der Anwendung der Funktion prev_use ergibt sich daraus:

$$\begin{aligned}
&\Gamma_V(\text{prev_use}(\text{rav}_x, x, \beta_2), \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\quad (((\{q_0, q_1, q_3\}, E, \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \text{undef}, \text{undef}), q_3), \emptyset, \\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\quad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})
\end{aligned}$$

$$\overset{6.5.2}{=} \Gamma_V(\mathrm{set}_{x,\mathrm{addr}_z,\mathrm{outValue}}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x,$$
$$((\{q_0,q_1,q_3\}, E, \{(q_0 \overset{\mathrm{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\mathrm{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \mathrm{undef}, \mathrm{undef}), q_3), \emptyset,$$
$$\{(\mathrm{get}_x, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{while}}_{x,\beta_1}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{if}}_{x,\beta_2,\beta_3}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})$$

7. Mit der Definition der Query–Funktionalität an Set–Operationen gilt dann:

$$\Gamma_V(\mathrm{set}_{x,\mathrm{addr}_z,\mathrm{outValue}}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x,$$
$$((\{q_0,q_1,q_3\}, E, \{(q_0 \overset{\mathrm{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\mathrm{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \mathrm{undef}, \mathrm{undef}), q_3),$$
$$\{(\mathrm{get}_x, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{while}}_{x,\beta_1}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{if}}_{x,\beta_2,\beta_3}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})$$
$$\overset{(7.7)}{=} \Gamma_P(\mathrm{addr}_z, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, \mathrm{outValue},$$
$$((\{q_0,q_1,q_3\}, E, \{(q_0 \overset{\mathrm{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\mathrm{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \mathrm{undef}, \mathrm{undef}), q_3),$$
$$\{(\mathrm{get}_x, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{while}}_{x,\beta_1}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{if}}_{x,\beta_2,\beta_3}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})$$

8. Da hierbei bisher keine Automaten als Querybedingungen mitgeführt werden, ergibt sich die Queryfunktion $\Gamma_P(\mathrm{addr}_z, \ldots)$ wiederum direkt aus $\widehat{\Gamma_P}(\mathrm{addr}_z, \ldots)$. Aus dem gleichen Grund ist dort der erste Fall der Definition zutreffend.

$$\Gamma_P(\mathrm{addr}_z, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, \mathrm{outValue},$$
$$((\{q_0,q_1,q_3\}, E, \{(q_0 \overset{\mathrm{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\mathrm{exit}(\beta_2)}{\rightarrow} q_3)\}, q_0, \emptyset, \mathrm{undef}, \mathrm{undef}), q_3),$$
$$\{(\mathrm{get}_x, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{while}}_{x,\beta_1}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{if}}_{x,\beta_2,\beta_3}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})$$
$$\overset{(7.10)}{=} (((\{q_0,q_1,q_3\}, E, \{(q_0 \overset{\mathrm{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\mathrm{exit}(\beta_2)}{\rightarrow} q_3)\},$$
$$q_0, \emptyset \cup \{q_3\}, \mathrm{undef}[q_3 \rightarrow z], \mathrm{undef}[q_3 \rightarrow \emptyset]), q_3),$$
$$\{(\mathrm{get}_x, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{while}}_{x,\beta_1}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{if}}_{x,\beta_2,\beta_3}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})$$
$$= (((\{q_0,q_1,q_3\}, E, \{(q_0 \overset{\mathrm{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\mathrm{exit}(\beta_2)}{\rightarrow} q_3)\},$$
$$q_0, \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_3),$$
$$\{(\mathrm{get}_x, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{while}}_{x,\beta_1}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\mathrm{join}^{\mathtt{if}}_{x,\beta_2,\beta_3}, \mathrm{deref}_{\mathrm{get}_x,\mathrm{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\})$$

Dies ergibt einen Automaten, der den Endzustand q_3 besitzt, dem durch die result–Abbildung die Variable z zugeordnet wird, und der durch die conditions–Abbildung auf die leere Menge

von betriebenen Automaten als mitgeführte Querybedingungen abgebildet wird. Dieser Automat ist in Abbildung 7.18(d) dargestellt.

9. Das Ergebnis aus Schritt 8 ist nun dasjenige Ergebnis, das in Schritt 5 dem Tupel

$$((((Q^{(3)}, E, \delta^{(3)}, q_0^{(3)}, F^{(3)}, \text{result}^{(3)}, \text{conditions}^{(3)}), q^{(3)})\kappa^{(3)})$$

zugewiesen wird. Entsprechend wird dieses Ergebnis nun verwendet, um die in Schritt 5 begonnene Berechnung fortzusetzen. Dabei ergibt sich der Automat aus Abbildung 7.18(e):

$$\begin{aligned}
&((((Q^{(4)}, E, \delta^{(4)}, q_0^{(4)}, F^{(4)}, \text{result}^{(4)}, \text{conditions}^{(4)}), q^{(4)})\kappa^{(4)}) \\
&\quad = \quad \Gamma_V(\text{cond_prev_use}(x, \text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \beta_3, \text{exit}(\beta_3)), \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\qquad (\{q_0, q_1, q_3\} \cup \{q_4\}, E, \\
&\qquad \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3)\}[q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4], \\
&\qquad q_0, \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset, \\
&\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\}) \\
&\overset{6.5.3}{=} \quad \Gamma_V(\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\qquad (\{q_0, q_1, q_3, q_4\}, E, \\
&\qquad \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4)\}, \\
&\qquad q_0, \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset, \\
&\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\}) \\
&\overset{(7.13)}{=} \quad \Gamma_V(\text{prev_use}(x, \text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \beta_1), \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\qquad (\{q_0, q_1, q_3, q_4\}, E, \\
&\qquad \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4)\}, \\
&\qquad q_0, \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset, \\
&\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1)\} \\
&\qquad [(\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)]) \\
&\overset{6.5.2}{=} \quad \Gamma_V(\text{var}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\qquad (\{q_0, q_1, q_3, q_4\}, E, \\
&\qquad \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4)\}, \\
&\qquad q_0, \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset, \\
&\qquad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
&\qquad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1), \\
&\qquad (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)\}) \\
&\overset{(7.11),6.5.2}{=} \quad \Gamma_V(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
&\qquad (\{q_0, q_1, q_3, q_4\}, E,
\end{aligned}$$

$$\{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4)\},$$
$$q_0, \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset,$$
$$\{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),$$
$$(\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1),$$
$$(\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)\})$$

10. An dieser Stelle der Berechnung tritt die Wiederholungserkennung in Aktion. Die Queryverarbeitung betrachtet die Operation $\text{join}^{\texttt{while}}_{x,\beta_1}$ mit einer leeren Menge von als Querybedingung mitgeführten betriebenen Automaten. Aus der Funktion κ geht hervor, daß eine Query für die gleiche Dereferenzierungs–Operation bereits mit einer leeren Menge von als Querybedingung mitgeführten betriebenen Automaten im Zustand q_0 bearbeitet wurde. Entsprechend berechnet sich $\Gamma_V(\text{join}^{\texttt{while}}_{x,\beta_1}, \ldots)$ gemäß des ersten Falls aus Gleichung (7.18).

$$\begin{aligned}
&\Gamma_V(\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x,\\
&\quad \{q_0, q_1, q_3, q_4\}, E,\\
&\quad \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4)\},\\
&\quad q_0, \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset,\\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),\\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),\\
&\quad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1),\\
&\quad (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)\})\\
\overset{(7.18)}{=}\ &(((\{q_0, q_1, q_3, q_4\}, E,\\
&\quad \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4)\}[q_4 \overset{\epsilon}{\rightarrow} q_0], q_0,\\
&\quad \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset,\\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),\\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),\\
&\quad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1),\\
&\quad (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)\})\\
=\ &(((\{q_0, q_1, q_3, q_4\}, E,\\
&\quad \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4), (q_4 \overset{\epsilon}{\rightarrow} q_0)\}, q_0,\\
&\quad \{q_3\}, \{(q_3 \rightarrow z)\}, \{(q_3 \rightarrow \emptyset)\}), q_4), \emptyset,\\
&\quad \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),\\
&\quad (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0),\\
&\quad (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1),\\
&\quad (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)\})
\end{aligned}$$

Dieses Ergebnis entspricht dem Automaten aus Abbildung 7.18(f).

11. Damit kann man nun die Berechnung aus Schritt 3 fortsetzen, indem dieses Ergebnis dort verwendet wird, um $(Q^{(2)}, \ldots)$ und damit das Endergebnis zu berechnen.

$$\begin{aligned}
&((((Q^{(2)}, E, \delta^{(2)}, q_0^{(2)}, F^{(2)}, \text{result}^{(2)}, \text{conditions}^{(2)}), q^{(2)})\kappa^{(2)})\\
&\quad = \quad \Gamma_V(\text{cond_prev_use}(x, \text{join}^{\texttt{while}}_{x,\beta_1}, \beta_0, \overline{\text{exit}}(\beta_1)), \text{deref}_{\text{get}_x,\text{outValue}}, x,\\
&\quad\quad (((\{q_0, q_1, q_3, q_4\} \cup \{q_2\}, E,
\end{aligned}$$

$$
\begin{aligned}
& \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\to} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\to} q_4), (q_4 \overset{\epsilon}{\to} q_0)\}[q_0 \overset{\overline{\text{exit}}(\beta_1)}{\to} q_2], \\
& q_0, \{q_3\}, \{(q_3 \to z)\}, \{(q_3 \to \emptyset)\}), q_2), \emptyset, \\
& \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_1), \\
& (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_4)\} \\
= \quad & \Gamma_V(\text{cond\_prev\_use}(x, \text{join}^{\texttt{while}}_{x,\beta_1}, \beta_0, \overline{\text{exit}}(\beta_1)), \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
& ((\{q_0, q_1, q_2, q_3, q_4\}, E, \\
& \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\to} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\to} q_4), (q_4 \overset{\epsilon}{\to} q_0), (q_0 \overset{\overline{\text{exit}}(\beta_1)}{\to} q_2)\}, \\
& q_0, \{q_3\}, \{(q_3 \to z)\}, \{(q_3 \to \emptyset)\}), q_2), \emptyset, \\
& \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_1), \\
& (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_4)\} \\
\overset{6.5.3}{=} \quad & \Gamma_V(\text{split}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
& ((\{q_0, q_1, q_2, q_3, q_4\}, E, \\
& \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\to} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\to} q_4), (q_4 \overset{\epsilon}{\to} q_0), (q_0 \overset{\overline{\text{exit}}(\beta_1)}{\to} q_2)\}, \\
& q_0, \{q_3\}, \{(q_3 \to z)\}, \{(q_3 \to \emptyset)\}), q_2), \emptyset, \\
& \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_1), \\
& (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_4)\} \\
\overset{(7.14),6.5.2}{=} \quad & \Gamma_V(\text{set}_{x,\text{addr}_y,\text{outValue}}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \\
& ((\{q_0, q_1, q_2, q_3, q_4\}, E, \\
& \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\to} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\to} q_4), (q_4 \overset{\epsilon}{\to} q_0), (q_0 \overset{\overline{\text{exit}}(\beta_1)}{\to} q_2)\}, \\
& q_0, \{q_3\}, \{(q_3 \to z)\}, \{(q_3 \to \emptyset)\}), q_2), \emptyset, \\
& \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_1), \\
& (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_4)\} \\
\overset{(7.7)}{=} \quad & \Gamma_P(\text{addr}_y, \text{deref}_{\text{get}_x,\text{outValue}}, \text{outValue}, \\
& ((\{q_0, q_1, q_2, q_3, q_4\}, E, \\
& \{(q_0 \overset{\text{exit}(\beta_1)}{\to} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\to} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\to} q_4), (q_4 \overset{\epsilon}{\to} q_0), (q_0 \overset{\overline{\text{exit}}(\beta_1)}{\to} q_2)\}, \\
& q_0, \{q_3\}, \{(q_3 \to z)\}, \{(q_3 \to \emptyset)\}), q_2), \emptyset, \\
& \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_0), \\
& (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_1), \\
& (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \to (M, q_4)\} \\
\overset{(7.10)}{=} \quad & (((\{q_0, q_1, q_2, q_3, q_4\}, E,
\end{aligned}
$$

$$
\begin{aligned}
& \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4), (q_4 \overset{\epsilon}{\rightarrow} q_0), (q_0 \overset{\overline{\text{exit}}(\beta_1)}{\rightarrow} q_2)\}, \\
& q_0, \{q_3\} \cup \{q_2\}, \{(q_3 \rightarrow z)\}[(q_2 \rightarrow y)], \{(q_3 \rightarrow \emptyset)\}[(q_2 \rightarrow \emptyset)]), q_2), \\
& \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
& (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
& (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1), \\
& (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)\}) \\
= \quad & (((\{q_0, q_1, q_2, q_3, q_4\}, E, \\
& \{(q_0 \overset{\text{exit}(\beta_1)}{\rightarrow} q_1), (q_1 \overset{\text{exit}(\beta_2)}{\rightarrow} q_3), (q_1 \overset{\text{exit}(\beta_3)}{\rightarrow} q_4), (q_4 \overset{\epsilon}{\rightarrow} q_0), (q_0 \overset{\overline{\text{exit}}(\beta_1)}{\rightarrow} q_2)\}, \\
& q_0, \{q_2, q_3\}, \{(q_3 \rightarrow z), (q_2 \rightarrow y)\}, \{(q_3 \rightarrow \emptyset), (q_2 \rightarrow \emptyset)\}), q_2), \\
& \{(\text{get}_x, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
& (\text{join}^{\texttt{while}}_{x,\beta_1}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_0), \\
& (\text{join}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_1), \\
& (\text{split}^{\texttt{if}}_{x,\beta_2,\beta_3}, \text{deref}_{\text{get}_x,\text{outValue}}, x, \emptyset, \emptyset) \rightarrow (M, q_4)\})
\end{aligned}
$$

Dieses Endergebnis entspricht dem Automaten aus Abbildung 7.18(g).

Die Berechnung der Queryverarbeitung für die zweite Dereferenzierungsoperation würde vom Prinzip analog verlaufen. Als Unterschied würde dort ab Schritt 2 in Abbildung 7.19 der gerade berechnete Automat als Querybedingung mitgeführt werden und bei den join–Operationen anhand der Zustandsübergangsfunktion δ^* betrieben werden, um damit eine Verträglichkeit des Weges der Query mit der mitgeführten Querybedingung sicherzustellen.

7.6 Zusammenfassung

Die Queryverarbeitung, die in praktischer Sichtweise das Analyseverfahren darstellt, wurde zusammen mit den dazu benötigten weiteren Definitionen und Algorithmen als rekursive Queryfunktionen Γ_P und Γ_V definiert. Auf der Basis dieser Funktionen wurde der Queryalgorithmus für Eingabeprogramme, die in der neuen Programmrepräsentation vorliegen, definiert.

Zuvor wurden die relevanten Teilaspekte der Queryverarbeitung jeweils informell vorgestellt. Sowohl die formale als auch die informelle Beschreibungsform wurden dann dazu verwendet, die Queryverarbeitung auf das Running Example anzuwenden. Das dabei berechnete Ergebnis stimmte mit dem anhand von Bedingungsautomaten in Kapitel 5 gefundenen überein.

Kapitel 8

Korrektheit

Im folgenden Kapitel soll die Korrektheit der Queryverarbeitung und damit die des gesamten vorgestellten Analyseverfahrens gezeigt werden. Dies umfasst zum einen die Korrespondenz der von Queries erzeugten Protokollautomaten mit Mengen von Programminstanzen, die bestimmte Operationen erreichen. Diese Korrespondenz wird sich auf die Eigenschaft "korrekt geschachteltes Teilwort" aus Definition 5.31 zurückführen lassen. Als nächstes Ziel wird gezeigt werden, daß eine Queryberechnung, bei der keine betriebenen Automaten als Querybedingungen mitgeführt werden, Pfadbedingungen im Sinne einer Analyse von *independent attributes* erzeugt. Schließlich wird die Verarbeitung von als Querybedingungen mitgeführten betriebenen Automaten auf das Prinzip der Bedingungsautomaten zurückgeführt werden, woraus sich die Korrektheit der Analyse im Sinne einer Berechnung von *relational attributes* ergeben wird.

8.1 Eigenschaften der bei der Queryverarbeitung erzeugten Protokollautomaten

Die Queryverarbeitung ist im vorhergehenden Kapitel durch eine rekursive Funktionsdefinition angegeben worden, die anhand des ersten Parameters jeweils als das Bearbeiten einer Query an einer bestimmten Operation interpretiert werden kann. Die Querybearbeitung wird dabei gezielt bei Operationen fortgesetzt, die die letzte Verwendung einer gemeinsam verwendeten Variablen, oder den Produzenten eines bestimmten benannten Wertes darstellen. Dadurch werden i.A. Operationen umgangen, die in Sequenzen von Operationen zwischen diesen beiden Operationen liegen. Zustandsübergänge mit denjenigen Eingabesymbolen, die bei der Bearbeitung der Query an diesen Operationen in den Protokollautomaten integriert worden wären, fehlen entsprechend im Protokollautomaten.

Ziel der nachfolgenden Beweise ist es zunächst, zu zeigen, daß die von der Queryverarbeitung in den Protokollautomaten integrierten Symbole mit denjenigen Symbolen, die laut Abschnitt 6.4.3 Programmeigenschaften und Programmpfade, damit also Programminstanzen, charakterisieren, korrespondieren. Dazu gehört zunächst eine Übereinstimmung der jeweiligen Symbole und der Zuordnung dieser Symbole zu Operationen, an denen diese bei der Queryverarbeitung und der Programminstanzenbeschreibung jeweils eine Bedeutung haben. Des weiteren muß eine Übereinstimmung der Reihenfolge, in denen diese Symbole in Protokollautomaten, bzw. instanzenmengenbeschreibenden Automaten vorkommen, gezeigt werden. Schließlich bleibt die Aussage zu beweisen, daß die von der Queryverarbeitung ausgelassenen Symbole die Eigenschaft "korrekt geschachteltes Teilwort" erfüllen.

Im Folgenden wird die Behandlung von als Querybedingungen mitgeführten Automaten bei der Queryverarbeitung zunächst explizit von der Betrachtung ausgenommen. Damit entfällt das Hinzufügen von solchen Automaten bei Operationen, an denen die Gültigkeit von Programmeigenschaften angenommen wird, das gemeinsame Betreiben dieser Automaten durch die Funktion δ^*, und das Integrieren von Eingabesymbolen bzw. Zustandsübergängen aus diesen Automaten in den Protokollautomaten gemäß Definition 7.8. Daß diese Teilaspekte der Queryverarbeitung korrekt sind,

wird im Rahmen der nachfolgenden Betrachtungen der Analyse von *relational attributes* gesondert nachgewiesen werden.

Satz 8.1 (Korrespondenz von Queryverarbeitung und Programmeigenschaften)
Die bei der Queryverarbeitung in den Protokollautomaten integrierten Symbole entsprechen der Beschreibung von Programmeigenschaften gemäß Abschnitt 6.4.3 in dem Sinne, daß eine eindeutige Zuordnung von Operationen, an denen die Querybehandlung ein solches Symbol integriert, zu Operationen, an denen die durch diese Symbole beschriebenen Programmeigenschaften gelten, existiert.

Beweis: In den Definitionen der Queryverarbeitung aus Abschnitt 7.4 werden die Protokollautomaten in den Fällen aus Gleichungen (7.6), (7.8), (7.15) und (7.18) durch Einfügen eines neuen Zustandsüberganges (der keine ϵ–Transition zur Wiederholungsbehandlung darstellt) verändert. Dies entspricht einer Queryverarbeitung an $\text{get}_{\omega'}$, $\text{set}_{\omega',\ldots}$, $\text{join}^{}_{\ldots}$ und $\text{join}^{\texttt{while}}_{\ldots}$–Operationen, die im Folgenden jeweils betrachtet werden soll.

Bei Operationen $\text{get}_{\omega'}$ werden Symbole der Form $[\omega' \rightarrow \ldots]$ in den Protokollautomaten der Query integriert. Diese stammen offensichtlich aus der Menge der Beschreibungen von Programmeigenschaften, die an der Operation ω' (die nur eine deref–Operation sein kann) gelten können. Da die Menge von Programmeigenschaften, die an der $\text{get}_{\omega'}$–Operation selbst gelten können, leer ist, und die $\text{get}_{\omega'}$ und die deref–Operation ω' von den Transformationen $\overline{T}$ (Definition 6.11) und T_z (Definition 6.12) stets als aufeinanderfolgende Operationen in der erzeugten Sequenz von Operationen erzeugt werden, kann man die in den Protokollautomaten integrierten Symbole auch der deref–Operation ω' zuordnen, wie das in Abschnitt 6.4.3 geschieht. Damit korrespondieren die bei der Queryverarbeitung an $\text{get}_{\omega'}$–Operationen in den Protokollautomaten integrierten Symbole direkt mit der Beschreibung von Programmeigenschaften gemäß Definition 6.4.3. Analog lassen sich bei der Queryverarbeitung an $\text{set}_{\omega'}$–Operationen in den Protokollautomaten integrierte Symbole der ebenfalls direkt davor in der Sequenz von Operationen stehenden deref–Operation ω' zuordnen. Die an join–Operationen integrierten Symbole korrespondieren mit den laut Abschnitt 6.4.3 den subblock(s)–Operationen zugeordneten Symbolen. Da eine Query Γ_V nur jeweils auf der Suche nach der letzten Zuweisung zu einer einzigen Variablen sein kann, und per Konstruktion zu jeder Variablen und jeder subblock(s)–Operation nur eine join–Operation direkt nach der subblock(s)–Operation existiert, kann man auch dieses Symbol der subblock(s)–Operation zuordnen und erhält eine zu Abschnitt 6.4.3 analoge Situation. □

Satz 8.2 (Reihenfolge der Integration von Symbolen)
Die Reihenfolge, in der von Queries Symbole in den Protokollautomaten integriert werden, entspricht der Reihenfolge der zugehörigen Zustandsübergänge in Abschnitt 6.4.3.

Beweis: Die Queryverarbeitung wird durch die rekursive Definition jeweils bei der letzten Verwendung einer Variablen oder beim Produzenten eines benannten Wertes fortgesetzt. In unverzweigten Sequenzen von Operationen befinden sich diese Zustandsübergänge jeweils weiter "vorne" in den Sequenzen von Operationen, was mit der Zustandsübergangsfunktion für Operationsblöcke aus Abschnitt 6.4.3 übereinstimmt. Im Falle von Schleifen oder Alternativen ergeben sich aus der Verwendung der Funktion cond_prev_use in der Definition der Queryverarbeitung analoge Zustandsübergänge zu denen aus Abschnitt 6.4.3. □

Satz 8.3 (Queries und korrekt geschachtelte Teilworte)
Die von Queries erzeugten Protokollautomaten beschreiben korrekt geschachtelte Teilworte in Bezug auf die Menge von Programminstanzen, die jeweils die aktuell von der Query betrachtete Operation erreichen.

Beweis: Nach Satz 8.1 und 8.2 beschreiben die Protokollautomaten bereits Teilfolgen der Worte, die die Menge aller Programminstanzen, die jeweils die aktuell von der Query betrachtete Operation erreichen, beschreiben. Zu zeigen bleibt, daß für die von der Queryverarbeitung nicht integrierten Symbole die Eigenschaften 1 und 2 der Definition 5.31 von korrekt geschachtelten Teilworten gilt.

Eigenschaft 1 ergibt sich aus der Feststellung, daß von einer Query eine Operation nur dann ein zweitesmal erreicht oder umgangen wird, wenn diese in einem Schleifenrumpf enthalten ist, der erneut betreten wird. Daher wird von der zugehörigen join–Operation ein Symbol exit(...) in den Protokollautomaten integriert. Selbst wenn beim ersten oder zweiten Passieren der Operation kein Eingabesymbol integriert wird, befinden sich die beiden potentiellen Vorkommen des Symbols durch das Symbol exit(...) getrennt in verschiedenen "Intervallen" des Teilwortes, so daß damit Eigenschaft 1 erfüllt ist.

Eigenschaft 2 ergibt sich aus einer Betrachtung, welche Operationen von Queries umgangen werden können. Hierfür existieren drei verschiedene Fälle. In Fall 1 befinden sich die umgangenen Operationen und die als nächstes von der Query betrachtete Operation, bei der ein Symbol in den Protokollautomaten integriert wird, in einem gemeinsamen Operationsblock. Damit verbleiben die Fälle, in denen die angesprochenen Operationen in verschiedenen Operationsblöcken liegen. In Fall 2 befinden sich die ausgelassenen Operationen in einem Schleifenrumpf, während die nächste betrachtete Operation mit Protokollautomatenmodifikation bei der Queryverarbeitung außerhalb des Schleifenrumpfes oder im gleichen Schleifenrumpf in einer weiteren Iteration liegt. Fall 3 bedeutet schließlich, daß die ausgelassenen Operationen in einem Operationsblock liegen, vor dem in einem hierarchisch übergeordneten Operationsblock die nächste Operation liegt, an der die Queryverarbeitung ein Symbol in den Protokollautomaten integriert. Da beim Eintritt in weitere Operationsblöcke zuvor an einer entsprechenden join–Operation der Protokollautomat verändert wird, und damit diese join–Operation bereits diejenige Operation darstellt, an der der Protokollautomat das nächste mal modifiziert wird, beschreibt der dritte Fall sämtliche restlichen Möglichkeiten, die nicht von den Fällen 1 und 2 abgedeckt werden.

Im ersten Fall folgt aus den Definitionen 4.12 und 5.30, daß die ausgelassenen Symbole jeweils größer als das nächste zu integrierende Symbol sind, da die umgangenen Operationen in der Sequenz von Operationen unterhalb derjenigen Operation sind, an denen der Protokollautomat als nächstes modifiziert wird. In Fall 2 wird vor der Bearbeitung eines Symbols, das an einer Operation außerhalb des Schleifenrumpfes oder in einer weiteren Iteration im gleichen Schleifenrumpf eine Bedeutung hat, an der entsprechenden join–Operation der Schleife ein Symbol exit(...) oder $\overline{\text{exit}}(...)$ integriert. Da eine Query Γ_V nur auf der Suche nach der letzten Zuweisung zu einer Variablen sein kann, die im aktuellen Operationsblock vorkommt, muss diese join–Operation existieren. Daher ist in dem Fall, daß die Query den Schleifenrumpf verläßt oder erneut betritt die nächste Operation, an der ein Symbol in den Protokollautomaten integriert wird, automatisch diese join–Operation. Aufgrund der Definition 5.30 von $\leq_{\widehat{E}}$ gilt für alle Symbole, die an Operationen im Schleifenrumpf eine Bedeutung haben, daß diese bzgl. der Ordnung $\leq_{\widehat{E}}$ größer sind als die entsprechenden exit(...) bzw. $\overline{\text{exit}}(...)$–Symbole. Damit kann Eigenschaft 2 aus Definition 5.31 auch in Fall 2 nachgewiesen werden.

Auch im dritten Fall kann man die Gültigkeit von Eigenschaft 2 aus der Definition 5.30 von $\leq_{\widehat{E}}$, speziell Fall 2 von Definition 4.12, ableiten.

Also gilt die Behauptung, d.h. daß diejenigen Worte, die von von Queries erzeugten Protokollautomaten akzeptiert werden, in Bezug auf Worte, die vollständig Programminstanzen beschreiben, korrekt geschachtelte Teilworte darstellen. □

Aus der Aussage dieses Satzes kann man folgern, daß die in Kapitel 5 eingeführten Operationen $\cap_{E^*}$ und $\cap_{M_a}$ auch auf von Protokollautomaten akzeptierte Worte bzw. Protokollautomaten selbst angewendet werden können, um die Forderungen von mehreren Worten oder Automaten zu vereinen. Damit lassen sich die Definitionen und Sätze aus Kapitel 5 auf Protokollautomaten anwenden, wodurch dieses theoretische Fundament für die Betrachtung der Queryverarbeitung zur Verfügung steht.

8.2 Protokollautomaten und Pfadbedingungen

Die Aufgabe des folgenden Abschnittes wird es sein, zu zeigen, daß von Queries erzeugte Protokollautomaten Pfadbedingungen im Sinne einer Analyse von *independent attributes* darstellen. Dazu wird zunächst eine implizite Definition der Queryverarbeitung ohne das Mitführen von betriebenen Automaten zur Querybeschränkung eingeführt.

Definition 8.1 (Vereinfachte Queryverarbeitung)
Die Funktionen $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ bezeichnen Queryfunktionen wie in Abschnitt 7.4.4 durch Γ_P und Γ_V definiert, die derart modifiziert wurden, daß sie keine weiteren Automaten bei der Queryverarbeitung hinzufügen oder betreiben.

Satz 8.4 (Terminierung von $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$)
Die rekursiven Queryfunktionen $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ terminieren in jedem Fall.

Beweis: Da in der vereinfachten Queryverarbeitung keine weiteren Automaten als Querybedingungen mitgeführt werden, kann an jeder Operation, an der eine Query für eine deref–Operation bereits vorher bearbeitet wurde, eine Wiederholung festgestellt und die Queryverarbeitung unter Generieren eines ϵ–Zustandsüberganges beendet werden. Da sowohl die Menge von Operationen, als auch die Anzahl von Variablen, nach deren letzter Zuweisung Queries $\overline{\Gamma_V}$ suchen können, endlich sind, gibt es nur endlich viele Möglichkeiten für jede Query, bevor ihre Verarbeitung beendet werden kann. □

Definition 8.2 (Queryberechnung mit maximaler Aufruftiefe)
Die Menge G_n bezeichne für ein $n \in \mathbb{N}$ die Menge von Berechnungen von $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ mit maximaler rekursiver Aufruftiefe n.

Satz 8.5 (Zusammenhang zwischen Queries und "globaler erweiterter Semantik")
Für alle $n \in \mathbb{N}$ gilt die folgende Aussage. Dabei sei mit $\text{deref}_{\ldots}$ *eine eindeutig bestimmte, aber nicht näher spezifizierte* deref*–Operation benannt. Bezeichnet für ein $\omega \in \Omega$ der Automat M_ω die Menge aller Programminstanzen, die gemäß Abschnitt 6.4.3 die Anweisung ω erreichen, und beschreibt weiter für einen Automaten $(Q, E, \delta, q_0, F, \text{result}, \text{conditions})$ mit bedingten Endzuständen, einen Zustand $q \in Q$ mit $q \notin F$ und eine Abbildung $\kappa \in \mathcal{K}$ (Fall 1)*

$$\begin{aligned}&\overline{\Gamma_P}(\omega, \text{deref}_{\ldots}, \text{outValue}, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa)\\ &= \ \ldots = \ (((Q', E, \delta', q_0, F', \text{result}', \text{conditions}'), q'), \kappa')\end{aligned}$$

oder alternativ für zusätzlich ein $v \in V$ (Fall 2)

$$\begin{aligned}&\overline{\Gamma_V}(\omega, \text{deref}_{\ldots}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa)\\ &= \ \ldots = (((Q', E, \delta', q_0, F', \text{result}', \text{conditions}'), q'), \kappa')\end{aligned}$$

jeweils eine Berechnung von $\overline{\Gamma_P}$ bzw. $\overline{\Gamma_V}$ aus der Menge G_n gemäß Definition 8.2, so gilt für alle $y \in \text{result}'(F' \backslash F)$ und alle $\pi \in E^*$ mit $\delta'(q, \pi) \in \left(\text{result}'\right)^{-1}(y) \backslash F$, daß in der durch $M_\omega \triangleright \pi$ beschriebenen Menge von Programminstanzen mit dabei durchlaufenen Grundoperationen $(\omega_1, \ldots, \omega_m) \in \Omega_0^*$ für ein $m \in \mathbb{N}$ jeweils gilt (Fall 1):

$$[\![(\omega_1, \ldots, \omega_m, \omega)]\!](\emptyset, \emptyset, \emptyset) = (\sigma, R, \Psi) \Longrightarrow (\omega, \text{outValue}, y) \in R$$

bzw. alternativ (Fall 2)

$$[\![(\omega_1, \ldots, \omega_m, \omega)]\!](\emptyset, \emptyset, \emptyset) = (\sigma, R, \Psi) \Longrightarrow \sigma(v) = y$$

Beweis: Beweis durch Induktion über n.
Induktionsanfang: $n = 1$.
Als einzige Situation, in der eine Queryfunktion mit nur einem Aufruf von $\overline{\Gamma_P}$ bzw. $\overline{\Gamma_V}$ ein Ergebnis liefert, ergibt sich die Bearbeitung einer Query $\overline{\Gamma_P}$ an einer addr_y–Operation für eine Variable $y \in V$. Dabei gilt:

$$\begin{aligned}&\overline{\Gamma_P}(\underbrace{\text{addr}_y}_{=:\omega}, \text{deref}_{\ldots}, \text{outValue}, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa)\\ &\overset{(7.10)}{=} (((Q, E, \delta, q_0, \underbrace{F \cup \{q\}}_{=:F'}, \underbrace{\text{result}[q \to y]}_{=:\text{result}'}, \text{conditions}[q \to \emptyset]), q), \kappa)\end{aligned}$$

Die betrachtete Ergebnismenge ergibt sich dabei als $\text{result}'(F' \setminus F) = \text{result}'(\{q\}) = y$. Die Menge von Worten $\pi \in E^*$, für die $\delta'(q,\pi) \in \left(\text{result}'\right)^{-1}(y) \setminus F$, also (da nach Voraussetzung $q \notin F$ gilt) $\delta'(q,\pi) = q$ ist, ergibt sich dabei als nur das leere Wort $\pi = \epsilon$.

Dadurch beschreibt $M_\omega \triangleright \pi$ die Menge M_ω selbst, d.h. sämtliche Programminstanzen, die ω erreichen. Für jede dieser Programminstanzen, bei deren Ablauf die Grundoperationen $(\omega_1, \ldots, \omega_m) \in \Omega_0^*$ für ein $m \in \mathbb{N}$ ausgeführt werden, kann man wie folgt anhand der erweiterten Semantik der addr–Operation folgern:

$$\begin{aligned}
&[\![(\omega_1, \ldots, \omega_m, \omega)]\!](\emptyset,\emptyset,\emptyset) \\
\overset{\text{Def.6.8}}{=}\ & [\![\omega]\!](\underbrace{[\![(\omega_1, \ldots, \omega_m)]\!](\emptyset,\emptyset,\emptyset)}_{=:(\sigma',R',\Psi')}) \\
=\ & [\![\text{addr}_y]\!](\sigma', R', \Psi') \\
\overset{(6.7)}{=}\ & (\sigma', R' \cup \{(\omega, \text{outValue}, y)\}, \Psi')) =: (\sigma, R, \Psi)
\end{aligned}$$

Damit gilt in allen diesen Programminstanzen, daß $(\omega, \text{outValue}, y) \in R$ ist und damit die Behauptung.

Induktionsanfang: $n = 2$

Um ein Ergebnis für eine Queryberechnung durch $\overline{\Gamma_V}$ zu erhalten, benötigt man mindestens zwei Operationen und damit einen zweiten rekursiven Aufruf. Damit ist die erste Berechnung, bei der man die Aussage für $\overline{\Gamma_V}$ zeigen kann, Element der Menge G_2. Hierbei müssen die Operationen von der Form

$$\begin{aligned}
\omega' :&\quad \text{addr}_y \\
\omega :&\quad \text{set}_{v,\omega',\text{outValue}}
\end{aligned}$$

sein. Analog zur obigen Argumentation im Fall $n = 1$ beschreibt der von der Queryauswertung erzeugte Automat wieder die Menge M_ω. Ebenso kann man aus der erweiterten Semantik der set–Operation folgern, daß $\sigma(v) = y$ sein muß, wenn wie oben

$$(\sigma, R, \Psi) := [\![(\omega_1, \ldots, \omega_m, \omega)]\!](\emptyset,\emptyset,\emptyset)$$

definiert wird. Daraus folgt die Behauptung. In beiden Fällen des Induktionsanfangs wird an den entsprechenden Operationen keine Wiederholungserkennung durchgeführt, weshalb diese hier auch nicht berücksichtigt werden musste.

Induktionsschritt: $n \to n+1$

Gelte die Behauptung für n. Bezeichne zunächst für ein $\omega \in \Omega$, einen Automaten $(Q, E, \delta, q_0, F, \text{result}, \text{conditions})$, einen Zustand $q \in Q$ mit $q \notin F$ und eine Abbildung $\kappa \in \mathcal{K}$

$$\begin{aligned}
&\overline{\Gamma_P}(\omega, \text{deref}_\ldots, \text{outValue}, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa) \\
&\quad = \quad \ldots = (((Q', E, \delta', q_0, F', \text{result}', \text{conditions}'), q'), \kappa')
\end{aligned}$$

bzw. für ein $v \in V$

$$\begin{aligned}
&\overline{\Gamma_V}(\omega, \text{deref}_\ldots, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa) \\
&\quad = \quad \ldots = (((Q', E, \delta', q_0, F', \text{result}', \text{conditions}'), q'), \kappa')
\end{aligned}$$

eine Berechnung aus G_{n+1}.

Der Induktionsschritt wird über eine Fallunterscheidung über die möglichen Typen von Operationen für ω geführt und exemplarisch für get_z und $\text{get}_{\omega'}$–Operationen ausführlich gezeigt. Für alle anderen Operationentypen ergeben sich keine von dieser Beweisführung verschiedenen Aspekte, so daß sie hier nur informell vorgeführt werden. In keinem der Fälle, in denen die Induktionsvoraussetzung verwendet wird, wird der aktuelle Zustand q des Protokollautomaten bei der Berechnung der Queryfunktion an der Operation ω zur Menge der Endzustände F hinzugefügt. Daher gilt für jeden aktuellen Zustand q des Protokollautomaten bei Verwendung der Induktionsvoraussetzung ebenfalls $q \notin F$, wie in der Induktionsaussage gefordert.

Fall 1: $\omega = \text{get}_v$ für eine Variable $v \in V$.

Aussage für $\overline{\Gamma_P}$:

Gemäß Gleichung (7.6) in Abschnitt 7.4.4 berechnet sich die Queryfunktion $\overline{\Gamma_P}$ wie folgt. Da von der Funktion $\overline{\Gamma_P}$ keine betriebenen Automaten mitgeführt werden, ergibt sich zunächst:

$$\begin{aligned} &\text{IntegrateConditions}(\text{get}_v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \text{False}) \\ &= \quad ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), \{(q, \emptyset)\}) \end{aligned}$$

Daher ergibt sich die Berechnung von $\overline{\Gamma_P}$:

$$\begin{aligned} &\overline{\Gamma_P}(\text{get}_v, \text{deref}_{\ldots}, \text{outValue}, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa) \\ &= \quad \widehat{\Gamma_P}(\text{get}_v, \text{deref}_{\ldots}, \text{outValue}, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa) \end{aligned}$$

Fall a: Zunächst soll Fall 2 in Gleichung (7.6) angenommen werden, d.h. es gilt nicht

$$\kappa(\text{get}_v, \text{deref}_{\ldots}, v, \emptyset, \emptyset) = (M'', q'')$$

für einen Automaten M'' und einen Zustand q'' dieses Automaten.

Dann berechnet sich $\widehat{\Gamma_P}(\ldots)$ wie folgt, wobei $\beta \in \Lambda$ denjenigen Operationsblock beschreiben soll, für den $\text{get}_v \in \beta$ ist.

$$\begin{aligned} &\widehat{\Gamma_P}(\text{get}_v, \text{deref}_{\ldots}, \text{outValue}, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa) \\ &= \quad \Gamma_V(\underbrace{\text{prev\_use}(v, \text{get}_v, \beta)}_{=:\omega'}, \text{deref}_{\ldots}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa[\ldots]) \end{aligned}$$

Dies ergibt einen rekursiven Aufruf mit maximaler Tiefe n, d.h. daß sich aus der Induktionsvoraussetzung ergibt:

$$\begin{aligned} &\Gamma_V(\omega', \text{deref}_{\ldots}, v, ((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \emptyset, \kappa[\ldots]) \\ &= \quad \ldots \quad = (((Q', E, \delta', q_0, F', \text{result}', \text{conditions}'), q'), \kappa') \end{aligned}$$

mit den folgenden Eigenschaften. Für alle $y \in \text{result}'(F' \setminus F)$ und alle $\pi \in E^*$ mit $\delta'(q, \pi) \in \left(\text{result}'\right)^{-1}(y) \setminus F$ gilt, daß in allen Programminstanzen aus der Menge $M_{\omega'} \triangleright \pi$, die die Operation ω' erreichen, für die Menge von darin ausgeführten Grundoperationen $(\omega_1, \ldots, \omega_m) \in \Omega_0^*$ mit $m \in \mathbb{N}$ die folgende Aussage gelten muß:

$$[\![(\omega_1, \ldots, \omega_m, \omega')]\!](\emptyset, \emptyset, \emptyset) = (\sigma, R, \Psi) \Longrightarrow \sigma(v) = y$$

Bezeichnen $\omega_1', \ldots, \omega_l'$ mit $l \in \mathbb{N}$ die Operationen, die in einer beliebigen Programminstanz zwischen ω' und ω ausgeführt werden, so kann man für diese folgern, daß sie keine Verwendung der Variablen v darstellen (sonst würde prev_use anstatt von ω' eine dieser Operationen als Ergebnis liefern). Damit kann man nun weiter folgern:

$$\begin{aligned} &[\![(\omega_1, \ldots, \omega_m, \omega', \omega_1', \ldots, \omega_l', \omega)]\!](\emptyset, \emptyset, \emptyset) \\ &= \quad [\![\text{get}_v]\!]([\![\underbrace{(\omega_1, \ldots, \omega_m, \omega', \omega_1', \ldots, \omega_l')}_{=:(\sigma'', R'', \Psi'') \text{ mit } \sigma''(v) = y}]\!](\emptyset, \emptyset, \emptyset)) \\ &= \quad (\sigma'', R'' \cup \{(\omega, \text{outValue}, y)\}, \Psi'') =: (\sigma, R, \Psi). \end{aligned}$$

Dieses Ergebnis gilt also offensichtlich in allen Programminstanzen, die die Operation ω erreichen und dabei vom Anfang des Programmes bis zur Operation ω' den Forderungen der möglichen Worte π (wie oben angegeben) genügen, sowie danach bis zum Erreichen der Operation ω

beliebige Programmeigenschaften annehmen bzw. beliebige Pfade durchlaufen können. Die Eingabesymbole, die Programmeigenschaften oder Verzweigungen an den Operationen $\omega_1', \ldots, \omega_l'$ beschreiben, werden von der Query nicht in den Protokollautomaten integriert. Nach Satz 8.3 beschreibt der von der Query erzeugte Protokollautomat korrekt geschachtelte Teilwörter in Bezug auf die Menge aller Programminstanzen, die die Operation ω erreichen. Damit sind die nicht integrierten Eingabesymbole alle vergleichbar mit und größer gleich dem letzten Symbol des Wortes π, weshalb $M_\omega \triangleright \pi$ mit $\delta'(q,\pi) \in \left(\text{result}'\right)^{-1}(y) \setminus F$ genau die oben informell beschriebene Menge aller Programminstanzen beschreibt. Damit wurde die Induktionsvoraussetzung auf die Menge aller Programminstanzen, die die Operation ω erreichen, übertragen und die Behauptung gezeigt.

Fall b: Es gilt $\kappa(\text{get}_v, \text{deref}_\ldots, v, \emptyset, \emptyset) = (M'', q'')$ für einen Automaten M'' und einen Zustand q'' dieses Automaten. Das bedeutet, daß an der get_v–Operation bereits eine Query (ausgehend von der gleichen $\text{deref}_\ldots$–Operation) verarbeitet wurde. Der Protokollautomat der Query befand sich zum damaligen Zeitpunkt im aktuellen Zustand q''.

Ohne die Wiederholungserkennung könnte man für beliebige endliche Aufruftiefen die Korrektheit der dabei produzierten Automaten per Induktion beweisen. Dabei würde der Teil des Protokollautomaten, der vom Zustand q'' ausgeht, durch die Weiterberechnung der Query von Zustand q aus identisch neu erzeugt werden. Damit würde der Protokollautomat die Menge von Worten beschreiben, die ein Teilwort $\pi' \in E^*$ mit $\delta(q'', \pi') = q$ beliebig häufig wiederholt enthalten können. Durch die Erzeugung der ϵ–Transition beschreibt der Protokollautomat genau diese Menge von Worten, ohne dabei einen Abschnitt des Protokollautomaten wiederholt berechnen zu müssen.

Aussage für $\overline{\Gamma_V}$:

Die Behandlung von Queries $\overline{\Gamma_V}$ an get_v–Operationen erfolgt analog zu der von Queries $\overline{\Gamma_P}$. Dabei wird ebenfalls eine Wiederholungserkennung vorgenommen, deren Korrektheit mit der gleichen Argumentation bewiesen werden kann. Die erweiterte Semantik der get_v–Operation verändert keine Variablenwerte, so daß die Weiterberechnung der Query $\overline{\Gamma_V}$ an der letzten Verwendung der Variablen v offensichtlich korrekt ist. Die Argumentation über die Menge von Programminstanzen, in denen die entsprechenden Aussagen gelten, kann analog zur Behandlung von $\overline{\Gamma_P}$–Queries erfolgen, da hier ebenfalls der Protokollautomat der Query nicht verändert wird.

Fall 2: $\omega = \text{get}_{\omega'}$ für eine (deref–)Operation $\omega' \in \Omega$.

Aussage für $\overline{\Gamma_P}$:

Nach Gleichung (7.6) in Abschnitt 7.4.4, sinngemäß angepasst an Definition 8.1, wird die Queryfunktion im Fall 2 durch die Hintereinanderausführung der Queries $\overline{\Gamma_V}(\ldots, \ldots, v_1, \ldots), \ldots, \overline{\Gamma_V}(\ldots \ldots, v_n, \ldots)$ definiert, die jeweils die letzte Zuweisung zu jeder der Variablen $v_1, \ldots, v_n$, die Ziel der Dereferenzierungsoperation ω' sein können, suchen. Der von jeder dieser Queries erzeugte Protokollautomat dient wiederum als Eingabe für die nächste Query. Das Ergebnis der sequentiellen Ausführung all dieser Queries wird als das Ergebnis der Queryfunktion $\overline{\Gamma_P}$ definiert.

Diese Queries haben nach Voraussetzung für die Berechnung von $\overline{\Gamma_P}$ jeweils maximale Aufruftiefe n, weshalb die Induktionsvoraussetzung für alle diese Queryberechnungen gilt. Daher beschreiben (in der Notation aus Gleichung (7.6)) für alle $y_1 \in \text{result}^{(1)}(F^{(1)} \setminus F)$, $y_2 \in \text{result}^{(2)}(F^{(2)} \setminus F^{(1)}), \ldots, y_n \in \text{result}^{(n)}(F^{(n)} \setminus F^{(n-1)})$ die Worte $\pi_1, \ldots, \pi_n \in E^*$ mit $\delta^{(1)}(q^1, \pi_1) \in \left(\text{result}^{(1)}\right)^{-1}(y_1) \setminus F, \ldots, \delta^{(n)}(q^n, \pi_n) \in \left(\text{result}^{(n)}\right)^{-1}(y_n) \setminus F^{(n-1)}$ jeweils Mengen von Programminstanzen $M_{\omega'} \triangleright \pi_1, \ldots, M_{\omega'} \triangleright \pi_n$, in denen bei der Operation ω' für jede mögliche in diesen Programminstanzen durchlaufene Sequenz von Grundoperationen $(\omega_1^{(1)}, \ldots, \omega_{m^{(1)}}^{(1)}), \ldots, (\omega_1^{(n)}, \ldots, \omega_{m^{(n)}}^{(n)}) \in \Omega_0^*$ für geeignete $m^{(1)}, \ldots, m^{(n)} \in \mathbb{N}$ jeweils aus $[\![(\omega_1^{(1)}, \ldots, \omega_{m^{(1)}}^{(1)})]\!](\emptyset, \emptyset, \emptyset) = (\sigma^{(1)}, R^{(1)}, \Psi^{(1)}), \ldots, [\![(\omega_1^{(n)}, \ldots, \omega_{m^{(n)}}^{(n)})]\!](\emptyset, \emptyset, \emptyset) = (\sigma^{(n)}, R^{(n)}, \Psi^{(n)})$ folgt, daß $\sigma^{(1)}(v_1) =$

$y_1, \ldots, \sigma^{(n)}(v_n) = y_n$ gelten muss. Da die Funktion result durch die Queryfunktion nur in ihrem Wertebereich erweitert, aber in ihrer bereits definierten Abbildungsvorschrift nicht verändert wird, genügt es, im Weiteren ausschließlich die Funktion result$^{(n)}$ zu betrachten.

Sei im Folgenden abkürzend $F^{(0)} := F$ definiert. Die Menge der Endzustände, die bei der Queryverarbeitung der ursprünglichen Query $\overline{\Gamma_P}$ nach der Berechnung von Gleichung (7.6) hinzugekommen sind, ergibt sich aufgrund der jeweiligen Teilmengenbeziehung $F^{(i)} \supseteq F^{(i-1)}$ für $1 \leq i \leq n$ als disjunkte Vereinigung der Endzustände, die in den einzelnen Queryberechnungen $\overline{\Gamma_V}$ erzeugt werden:

$$F^{(n)} \setminus F = (F^{(n)} \setminus F^{(n-1)}) \cup (F^{(n-1)} \setminus F^{(n-2)}) \ldots \cup (F^{(1)} \setminus F)$$

Für jeden Endzustand $q_f \in F^{(n)} \setminus F$ kann man daher ein $1 \leq i \leq n$ angeben, so daß $q_f \in F^{(i)} \setminus F^{(i-1)}$ gilt. Für ein $y \in \text{result}^{(n)}(F^{(n)} \setminus F)$ kann man damit i.A. eine Menge von Endzuständen $q_{f_1}, \ldots, q_{f_k} \in F^{(n)} \setminus F$ für ein $k \in \mathbb{N}$ angeben, von denen jeder für ein $1 \leq i \leq n$ einer solchen Menge $F^{(i)} \setminus F^{(i-1)}$ zuzuordnen ist, und für die jeweils für $1 \leq l \leq k$ gilt, daß $\text{result}^{(n)}(q_{f_l}) = y$.

Sei nun $y \in \text{result}^{(n)}(F^{(n)} \setminus F)$ beliebig und sei $q_f \in F^{(n)} \setminus F$ beliebig gewählt, mit $\text{result}^{(n)}(q_f) = y$. Damit existiert ein $i \in \{1, \ldots, n\}$ mit $q_f \in F^{(i)} \setminus F^{(i-1)}$ und damit gelten die oben aus der Induktionsannahme gefolgerten Aussagen für $y_i := y$.

Der Protokollautomat, der sich nach Gleichung (7.6) als Ergebnis von $\overline{\Gamma_P}$ aus den oben angegebenen Teilergebnissen zusammensetzt, verbindet den aktuellen Zustand q der Query $\overline{\Gamma_P}$ durch Zustandsübergänge mit den Eingabesymbolen $[\omega' \rightarrow v_1], \ldots, [\omega' \rightarrow v_n]$ mit den bei der Queryverarbeitung neu hinzugefügten Zuständen $q^1, \ldots, q^n$. Die Worte $\pi \in E^*$, für die $\delta(q, \pi) \in F^{(n)} \setminus F$ gilt, sind nun alle von der Form $[\omega' \rightarrow v_1]; \pi_1, \ldots, [\omega' \rightarrow v_n]; \pi_n$ für Worte $\pi_1, \ldots, \pi_n$ wie oben definiert.

In jeder Programminstanz, in der $\text{get}_{\omega'}$ ausgeführt wird, wird aufgrund der Form der Transformationsfunktion T_z direkt zuvor die deref–Operation ω' ausgeführt. Aufgrund von Eigenschaft 1 in Definition 5.31 und der Tatsache, daß nach Satz 8.3 von Protokollautomaten akzeptierte Automaten korrekt geschachtelte Teilwörter sind, kann man folgern, daß das Symbol $[\omega' \rightarrow \ldots]$ das Ziel der deref–Operation ω' bei genau dieser letzten Ausführung beschreibt. Daher gilt in jeder der Programminstanzen $M_\omega \triangleright \pi$, die durch ein Wort $\pi \in E^*$ mit $\pi = [\omega' \rightarrow v_i]; \pi_i$ mit π_i wie oben definiert beschrieben wird, daß die Dereferenzierungsoperation das Ziel v_i besitzt. Für jede Sequenz von Grundoperationen $(\omega_1, \ldots, \omega_l, \omega', \omega) \in \Omega_0^*$ für ein $l \in \mathbb{N}$, die in einer dieser Programminstanzen ausgeführt werden, kann man dann folgern, daß für

$$(\sigma', R', \Psi') := [\![(\omega_1, \ldots, \omega_l, \omega')]\!](\emptyset, \emptyset, \emptyset)$$

gilt, daß $\Psi'(\omega') = v_i$ und aufgrund der obigen Betrachtung der Induktionsvoraussetzung $\sigma(v_i) = y_i = y$ ist. Betrachtet man nun

$$(\sigma, R, \Psi) := [\![(\omega_1, \ldots, \omega_l, \omega', \omega)]\!](\emptyset, \emptyset, \emptyset),$$

so kann man aus der erweiterten Semantik der $\text{get}_{\omega'}$–Operation und Definition 6.8 folgern, daß

$$\begin{aligned}
(\sigma, R, \Psi) &= [\![(\omega_1, \ldots, \omega_l, \omega', \omega)]\!](\emptyset, \emptyset, \emptyset) \\
&= [\![\text{get}_{\omega'}]\!]([\![(\omega_1, \ldots, \omega_l, \omega')]\!](\emptyset, \emptyset, \emptyset)) \\
&= [\![\text{get}_{\omega'}]\!](\sigma', R', \Psi') \\
&= (\sigma', R' \cup \{(\omega, \text{outValue}, \sigma(\Psi'(\omega')))\}, \ldots) \\
&= (\sigma', R' \cup \{(\omega, \text{outValue}, \sigma(v_i))\}, \ldots) \\
&= (\sigma', R' \cup \{(\omega, \text{outValue}, y)\}, \ldots)
\end{aligned}$$

Damit gilt die Behauptung auch im Fall 2. Die Korrektheit der Wiederholungserkennung im Fall 2 kann analog zum Fall 1 bewiesen werden.

Aussage für $\overline{\Gamma_V}$: Da die erweiterte Semantik der $\text{get}_{\omega'}$–Operation Variablenbelegungen unverändert läßt und die Queryverarbeitung in Gleichung 7.5 analog zu Gleichung 7.1 erfolgt, ist der Beweis analog zum Beweis von $\overline{\Gamma_V}$ in Fall 1.

Fall 3: $\omega = \text{set}_{v,\omega',p}$.

Aussage für $\overline{\Gamma_P}$: Da die set–Operation keinen benannten Wert produziert, kann sie nicht von einer Query $\overline{\Gamma_P}$ erreicht werden.

Aussage für $\overline{\Gamma_V}$: Der Beweis erfolgt analog zu Fall 1. Dabei kann man anhand der erweiterten Semantikfunktion der $\text{set}_{v,\omega',p}$–Operation aus der Induktionsvoraussetzung für Queries $\overline{\Gamma_P}$ auf die Gültigkeit der Induktionsbehauptung für Queries $\overline{\Gamma_V}$ schließen. Die Argumentation über die Menge von Programminstanzen, die von den bei diesem Querybearbeitungsschritt nicht veränderten Protokollautomaten beschrieben werden, erfolgt ebenfalls analog zu Fall 1.

Fall 4: $\omega = \text{set}_{\omega'',\omega',p}$.

Aussage für $\overline{\Gamma_P}$: Da diese set–Operation ebenfalls keinen benannten Wert produziert, kann sie nicht von einer Query $\overline{\Gamma_P}$ erreicht werden.

Aussage für $\overline{\Gamma_V}$: Dieser Beweis erfolgt analog zu Fall 2. Als einziger Unterschied ergibt sich eine Fallunterscheidung danach, ob bei Annahme der Programmeigenschaft $\omega'' \rightarrow v_i$ die Query $\overline{\Gamma_V}$ auf der Suche nach der letzten Zuweisung zur Variablen v_i oder einer anderen Variablen war. Entsprechend läßt sich die Induktionsbehauptung auf die Induktionsvoraussetzung für $\overline{\Gamma_P}$ zurückführen, wenn durch die set–Operation der Variablen v_i in der betrachteten Menge von Programminstanzen der Inhalt des entsprechenden benannten Wertes zugewiesen wurde, bzw. auf die Induktionsvoraussetzung für $\overline{\Gamma_V}$, falls in dieser betrachteten Menge von Programminstanzen einer anderen Variablen von der set–Operation ein Wert zugewiesen wurde, und die Operation ω daher die von der Query untersuchte Variable effektiv nicht verwendet.

Fall 5: $\omega = \text{addr}_v$ für ein $v \in V$.

Aussage für $\overline{\Gamma_P}$: Da die Queryverarbeitung von Queries $\overline{\Gamma_P}$ stets an addr–Operationen terminiert, kann es keine Berechnung mit weiterer Berechnungstiefe $n > 1$ geben. Daher ist dieser Teil von Fall 5 bereits durch den Induktionsanfang abgedeckt.

Aussage für $\overline{\Gamma_V}$: Analog zu einer get_v–Operation aus Fall 1 hat eine addr–Operation keine Auswirkung auf Variablenbelegungen. Daher verläuft der Beweis analog zu dem aus Fall 1.

Fall 6: $\omega = \text{deref}_{\omega',\text{outValue}}$ für ein $\omega' \in \Omega$.

Dieser Fall ist nur der Vollständigkeit der betrachteten Operationen halber aufgelistet. Da die Queries $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ stets an denjenigen Operationen verarbeitet werden, die eine Verwendung des Ergebnisses der Dereferenzierung darstellen, wird eine deref–Operation nie von einer Query erreicht.

Fall 7: $\omega = \text{var}_v$ oder $\omega = \text{rav}_v$ für ein $v \in V$.

Aussage für $\overline{\Gamma_P}$: Da var– oder rav–Operationen keinen benannten Wert produzieren, können sie auch nicht von Queries $\overline{\Gamma_P}$ erreicht werden.

Aussage für $\overline{\Gamma_V}$: Analog zu get_v–Operationen in Fall 1 verändern var– und rav–Operationen Variablenbelegungen nicht. Der Beweis der Induktionsbehauptung ergibt sich daher analog zu Fall 1.

Fall 8: $\omega = \text{join}^{\texttt{if}}_{\ldots}$ oder $\omega = \text{join}^{\texttt{while}}_{\ldots}$.

Zum Beweis der Korrektheit der Aussage an $\text{join}^{\texttt{if}}_{\ldots}$ und $\text{join}^{\texttt{while}}_{\ldots}$–Operationen kann im Gegensatz zu den anderen Fällen, in denen Grundoperationen betrachtet werden, nicht die erweitere Semantikfunktion herangezogen werden. Wie in Abschnitt 6.3.2.3 bereits diskutiert wurde,

besitzen Kontrollflußoperationen keine erweiterte Semantik. Daher tragen diese nur implizit dazu bei, die Korrespondenz von Programminstanzen und den dabei durchlaufenen Sequenzen von Grundoperationen zu beschreiben.

Aussage für $\overline{\Gamma_P}$: Da $\text{join}^{\texttt{if}}_{\ldots}$– oder $\text{join}^{\texttt{while}}_{\ldots}$–Operationen keinen benannten Wert produzieren, können sie auch nicht von Queries $\overline{\Gamma_P}$ erreicht werden.

Aussage für $\overline{\Gamma_V}$: Gemäß Gleichungen 7.15 und 7.18 wird die Verarbeitung von Queries $\overline{\Gamma_V}$ an join–Operationen durch zwei hintereinander berechnete Queries definiert, die jeweils im then– und else–Zweig, bzw. im und vor dem Schleifenrumpf nach einer letzten Zuweisung zur von der Query betrachteten Variablen suchen. Diese beiden Queries haben jeweils wieder maximale Aufruftiefe n, weshalb die Induktionsvoraussetzung für sie gilt. Der Protokollautomat wird dabei analog zu Fall 2 um die entsprechenden exit(...) bzw. $\overline{\text{exit}}$(...)–Symbole erweitert. Damit legen für ein $y \in \text{result}^{(2)}(F^{(2)} \setminus F)$ Worte $\pi \in E^*$ mit $\delta(q,\pi) \in \left(\text{result}^{(2)}\right)^{-1}(y) \setminus F$ zunächst fest, welcher Operationsblock von der damit beschriebenen Programminstanz ausgeführt wurde, und beschreiben dann weiter das Ergebnis, das von einer der beiden Queries sozusagen unter der Annahme dieser gewählten Verzweigung gefunden wurde.

Fall 9: $\omega = \text{split}^{\texttt{if}}_{\ldots}$ oder $\omega = \text{split}^{\texttt{while}}_{\ldots}$.

Aussage für $\overline{\Gamma_P}$: Da $\text{split}^{\texttt{if}}_{\ldots}$– oder $\text{split}^{\texttt{while}}_{\ldots}$–Operationen keinen benannten Wert produzieren, können sie auch nicht von Queries $\overline{\Gamma_P}$ erreicht werden.

Aussage für $\overline{\Gamma_V}$: Die $\text{split}^{\texttt{if}}_{\ldots}$– oder $\text{split}^{\texttt{while}}_{\ldots}$–Operationen verändern keine Variablenbelegungen und können daher analog zu Fall 1 behandelt werden. Dies schließt auch die Wiederholungserkennung an den $\text{split}^{\texttt{if}}_{\ldots}$–Operationen mit ein.

□

Aus diesem Satz kann man nun als Folgerung den Zusammenhang zwischen Protokollautomaten und Pfadbedingungen ableiten.

Satz 8.6 (Folgerung: Protokollautomaten sind Pfadbedingungen)
Modifiziert man den Query–Algorithmus aus Abschnitt 7.4.3 derart, daß er Queries $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ ohne mitgeführte Querybedingungen anstelle von Queries Γ_P und Γ_V verwendet, dann beschreiben die dabei berechneten Protokollautomaten $M = (Q, E, \delta, q_0, F, \text{result}, \text{conditions})$ Pfadbedingungen für Operationen $\omega = \text{deref}_{\omega', \text{outValue}}$, von denen aus diese Queries gestartet wurden, im Sinne einer Analyse von independent attributes, *d.h. es gilt*

$$M \Longrightarrow (\omega \to \ldots)_\omega$$

gemäß Definition 5.39.

Beweis: Obige Schreibweise für Pfadbedingungen umfasst folgende Aussagen, wobei für ein $y \in \text{result}(F)$ der Automat M_y wie in Definition 5.38 beschrieben definiert sein soll. Dabei bezeichne $\{v_1, \ldots, v_n\} = \text{result}(F)$ für ein $n \in \mathbb{N}$ die Menge der möglichen Variablen, die Endzuständen aus F bei der Queryverarbeitung zugeordnet werden.

$$\begin{aligned} M_{v_1} &\Longrightarrow (\omega \to v_1)_\omega \\ &\ldots \\ M_{v_n} &\Longrightarrow (\omega \to v_n)_\omega \end{aligned}$$

Nach Satz 8.5 gilt, daß für jede Queryberechnung für eine deref–Operation $\omega = \text{deref}_{\omega', \text{outValue}}$

$$\begin{aligned} &\overline{\Gamma_P}(\omega', \text{deref}_{\omega', \text{outValue}}, \text{outValue}, ((Q', E, \delta', q_0', F', \text{result}', \text{conditions}'), q'), \emptyset, \kappa) \\ &= \ldots = (((Q'', E, \delta'', q_0', F'', \text{result}'', \text{conditions}''), q''), \kappa'') \end{aligned}$$

gefolgert werden kann, daß für alle $y \in \text{result}'(F'' \setminus F')$ und alle $\pi \in E^*$ mit $\delta''(q', \pi) \in \left(\text{result}''\right)^{-1}(y) \setminus F'$ gilt, daß in der durch $M_\omega \triangleright \pi$ beschriebenen Menge von Programminstanzen mit dabei durchlaufenen Grundoperationen $(\omega_1, \ldots, \omega_m) \in \Omega_0^*$ für ein $m \in \mathbb{N}$ jeweils gilt:

$$[\![(\omega_1, \ldots, \omega_m, \omega)]\!](\emptyset, \emptyset, \emptyset) = (\sigma, R, \Psi) \Longrightarrow (\omega, \text{outValue}, y) \in R \tag{8.1}$$

Die Berechnung, die vom Queryalgorithmus gestartet wird, und dann das oben mit M bezeichnete Ergebnis besitzt, wird mit einem Automaten, der nur aus seinem Startzustand, der ebenfalls aktueller Zustand des betriebenen Automaten ist, als Parameter initiiert. Damit gilt

$$\begin{aligned}&\overline{\Gamma_P}(\omega', \text{deref}_{\omega', \text{outValue}}, \text{outValue}, ((\{q_0\}, E, \text{undef}, q_0, \emptyset, \text{undef}, \text{undef}), q_0), \emptyset, \text{undef})\\ &\quad = \quad \ldots = \quad (((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \kappa)\end{aligned}$$

Setzt man die in dieser Gleichung verwendeten Automatenbestandteile als die ein– und zweigestrichenen Bestandteile der obigen Aussage von Satz 8.5 ein, so erhält man folgende Aussage: für alle $y \in \text{result}(F \setminus \emptyset)$ und alle $\pi \in E^*$ mit $\delta(q_0, \pi) \in (\text{result})^{-1}(y) \setminus \emptyset$ gilt, daß in der durch $M_\omega \triangleright \pi$ beschriebenen Menge von Programminstanzen mit dabei durchlaufenen Grundoperationen $(\omega_1, \ldots, \omega_m) \in \Omega_0^*$ für ein $m \in \mathbb{N}$ jeweils die Aussage aus Gleichung (8.1) gefolgert werden kann. Die Menge dieser Wörter π beschreibt dabei genau die Sprache $\mathcal{L}(M_y)$ gemäß Definition 5.11. Aus der erweiterten Semantik der $\text{deref}_{\omega', \text{outValue}}$–Operation aus Definition 6.7 folgt, daß diese die Variable y als Ziel $\Psi(\text{deref}_{\omega', \text{outValue}})$ besitzt, wenn sie einen benannten Wert $(\omega', \text{outValue}, y)$ bei der Ausführung konsumiert. Dies kann aufgrund von Gleichung (8.1) für jede Programminstanz aus $M_\omega \triangleright \pi$ und, da π beliebig aus M_y gewählt war, auch für jede Programminstanz aus $M_\omega \triangleright M_y$ gefolgert werden, weshalb damit nach Definition 5.37 jeder der Automaten M_y für alle $y \in \text{result}(F)$ eine Pfadbedingung für die Programmeigenschaft $(\omega \rightarrow y)_\omega$ darstellt. Damit gilt die Behauptung.

Da durch die vereinfachte Queryberechnung anhand der Funktionen $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ keine Überprüfung auf mögliches gemeinsames Vorkommen der durch Eingabesymbole beschriebenen Programmeigenschaften in Programminstanzen vorgenommen wird, handelt es sich dabei offensichtlich um eine Analyse von *independent attributes*. □

Die Aussage darüber, welche Programmeigenschaften in welchen Mengen von Programminstanzen gelten, die vom Anfang des Programmes eine bestimmte Operation erreichen, ist das Gegenstück in dieser Arbeit zum Übergang von einer lokalen zu einer globalen Semantik in traditionellen Programmanalysemethoden. Im Gegensatz zu diesen ist die Aussage der hier vorgestellten Analysemethode eine Partitionierung der Menge von Programminstanzen nach darin geltenden Programmeigenschaften, anstatt der Berechnung der (verbands–)maximal ableitbaren Information, die in allen Programminstanzen gilt.

8.3 Realisierung einer Analyse von *relational attributes*

Bisher wurde für eine vereinfachte Version der Queryverarbeitung gezeigt, daß sie Pfadbedingungen im Sinne einer Analyse von *independent attributes* berechnet. Nach Satz 5.9 kann man diese Pfadbedingungen dazu verwenden, um durch gemeinsames Betreiben einer dynamischen Menge von solchen Pfadbedingungen in einem gemeinsamen Konstrukt, das als Bedingungsautomat bezeichnet wurde, eine Lösung für eine Analyse von *relational attributes* zu erhalten. Der Beweis der Korrektheit der Analyse von *relational attributes* durch die Queryverarbeitung wird daher darauf abzielen, daß das Mitführen von weiteren betriebenen Automaten beim Fluß der Query durch den Analysegraphen dem Betreiben eines Bedingungsautomaten entspricht.

Satz 8.7 (Queryberechnung entspricht Betreiben von Bedingungsautomaten)
Bezeichne gemäß Definition 7.9

$$N = \max\{j \in \mathbb{N} \mid \Omega^{(j)}_{\text{deref}} \neq \emptyset\}$$

die maximale Referenzierungsstufe der in der Darstellung eines Eingabeprogrammes in der neuen Programmrepräsentation vorkommenden Dereferenzierungsoperationen. Seien weiter für alle Operationen

$$\omega_i = \text{deref}_{\omega_i', \text{outValue}} \in \bigcup_{j=0}^{N} \Omega^{(j)}_{\text{deref}}$$

für $i \in \{1, \ldots, m\}$ für ein geeignetes $m \in \mathbb{N}$ die Automaten $\left(\overline{M_i}\right)_{i \in \{1,\ldots,m\}}$ wie folgt definiert:

$$((\overline{M_i}, \ldots), \ldots) \quad := \quad \overline{\Gamma_P}(\omega_i', \omega_i, \text{outValue}, ((\{\overline{q_0^i}\}, E, \text{undef}, \overline{q_0^i}, \emptyset, \text{undef}, \text{undef}), \overline{q_0^i}), \emptyset, \text{undef})$$

mit $\{\overline{q_0^i} \mid i \in \{1, \ldots, m\}\}$ einer Menge von Zuständen. Die Automaten $\left(\overline{M_i}\right)_{i \in \{1,\ldots,m\}}$ beschreiben nach Satz 8.6 Pfadbedingungen für die Ziele der Dereferenzierungsoperationen ω_i, $i \in \{1, \ldots, m\}$ im Sinne einer Analyse von independent attributes.

Seien weiter die Automaten $(M_i)_{i \in \{1,\ldots,m\}}$, in der durch den Analysealgorithmus aus Abschnitt 7.4.3 festgelegten Reihenfolge berechnet, wie folgt definiert:

$$((M_i, \ldots), \ldots) \quad := \quad \Gamma_P(\omega_i', \omega_i, \text{outValue}, ((\{q_0^i\}, E, \text{undef}, q_0^i, \emptyset, \text{undef}, \text{undef}), q_0^i), \emptyset, \text{undef})$$

mit $\{q_0^i \mid i \in \{1, \ldots, m\}\}$ ebenfalls einer Menge von Zuständen.

Definiere weiter für alle $i \in \{1, \ldots, m\}$ gemäß Definition 5.40 C_i einen Bedingungsautomaten mit Startzustand $(\overline{M_i}, \overline{q_0^i})$, der bei Antreffen von Symbolen, die Programmeigenschaften beschreiben, Automaten aus der Menge $\left(\overline{M_j}\right)_{j \in \{1,\ldots,m\}}$ zur Menge der betriebenen Automaten hinzufügt.

Dann gilt für alle $n \in \{0, \ldots, N\}$ und alle $i \in \{1, \ldots, m\}$ mit $\omega_i = \text{deref}_{\omega', \text{outValue}} \in \Omega^{(n)}_{\text{deref}}$, daß die Automaten M^i und C_i strukturell identisch sind, und deswegen

$$\mathcal{L}(M^i) = \mathcal{L}(C_i)$$

ist.

Beweis: Anstatt hier einen länglichen Isomorphiebeweis der Automaten aus der Behauptung anzuführen, sollen im Folgenden teilweise informell die prinzipiellen Unterschiede in der Verarbeitung von Queries Γ_P (die als Ergebnis die Automaten M_i ergeben) und dem Betreiben von Bedingungsautomaten (auf der Grundlage der Automaten $\left(\overline{M_j}\right)_{j \in \{1,\ldots,m\}}$) aufgeführt werden. Anhand dieser kann dann argumentiert werden, daß die dabei jeweils resultierende Struktur und damit Sprache der Automaten gleich sein muß.

Von der grundsätzlichen Struktur ist die Argumentation ein Induktionsbeweis über die Referenzierungsstufe $n \in \{0, \ldots, N\}$ der Dereferenzierungsoperationen ω_i mit Queryergebnis M_i.

Induktionsanfang: $n = N$. Da der Fluß der Query Γ_P, die als Ergebnis den Automaten M_i zur Dereferenzierungsoperation ω_i mit $\omega_i \in \Omega^{(n)}_{\text{deref}}$ für ein $i \in \{1, \ldots, m\}$ ergibt, nur von Dereferenzierungsoperationen mit Referenzierungsstufe $N + 1$ abhängen kann, diese nach Voraussetzung aber nicht existieren, treffen die Queries Γ_P und $\overline{\Gamma_P}$ für diese Dereferenzierungsoperation ω_i beide keine Operationen an, an denen sie Symbole für Programmeigenschaften in den Protokollautomaten aufnehmen und ggfs. weitere Automaten zur Bearbeitung hinzufügen müssten. Damit verläuft die Berechnung dieser beiden Automaten absolut gleich, weshalb sie identische Struktur besitzen und damit auch die von ihnen akzeptierte Sprache gleich ist. Damit ist der Induktionsanfang gezeigt.

Induktionsschritt: $n \to n - 1$: Gelte die Behauptung für n, d.h. daß die Berechnung aller Queryfunktionen beginnend bei Dereferenzierungsoperationen $\omega_i \in \Omega^{(n)}_{\text{deref}}$ mit Referenzierungsstufe n jeweils Automaten M_i ergibt, deren Struktur und Sprache gleich der von entsprechenden Bedingungsautomaten mit Startzustand $(\overline{M_i}, \overline{q_0^i})$ ist. Zu zeigen ist die Aussage für $n - 1$. Da der Fluß der Queries Γ_P bzw. $\overline{\Gamma_P}$ beginnend an Dereferenzierungsoperationen mit Referenzierungsstufe $n - 1$ jeweils nur von anderen Dereferenzierungsoperationen mit Referenzierungsstufe n abhängen kann, sind nach Induktionsvoraussetzung alle von der Query Γ_P zur Verarbeitung hinzugenommene Automaten bereits von ihrer akzeptierten Sprache äquivalent zu entsprechenden Bedingungsautomaten. Damit beschreibt jeder während der Queryverarbeitung hinzugenommene betriebene Automat M^i

mit $i \in \{1, \ldots, m\}$ zusammenfassend das gemeinsame Betreiben einer Menge von Automaten aus $\left(\overline{M_j}\right)_{j \in \{1,\ldots,m\}}$ gemäß der Definition 5.40 eines Bedingungsautomaten.

Die relevanten Unterschiede bzw. Gemeinsamkeiten zwischen den beiden Arten, die Automaten M_i und $\overline{M_i}$ zu berechnen sind nun wie folgt.

1. Die Auswahl des nächsten möglichen Eingabesymbols erfolgt bei beiden Automaten gleich, da aufgrund von Satz 8.3 die einzelnen von Bedingungsautomaten mitgeführten Automaten jeweils korrekt geschachtelte Teilworte als Sprache akzeptieren, und damit gemäß der Argumentation aus Kapitel 5 sämtliche als nächstes mögliche Eingabesymbole bzgl. der Ordnung $\leq_{\widehat{E}}$ vergleichbar sind. Damit sind auch die nächstmöglichen Eingabesymbole der bereits zusammengefassten Bedingungsautomaten vergleichbar und ergeben das gleiche maximale Symbol.

2. Bei der Berechnung des Automaten M_i durch eine Query Γ_P existiert ein ausgezeichneter "Hauptautomat", der aktuell von der Query erzeugte Protokollautomat, der weitere betriebene "Nebenautomaten" mitführt. Während beim Betreiben eines Bedingungsautomaten alle Automaten gleichberechtigt sind, und dabei für jedes Eingabesymbol, das eine Programmeigenschaft beschreibt, der entsprechende Pfadbedingungsautomat zur Menge der vom Bedingungsautomaten betriebenen Automaten hinzugefügt wird, werden von den Queries $\Gamma_{P/V}$ nur für den Fluss der Query und damit den Protokollautomaten direkt relevante Eingabesymbole durch das Hinzufügen von weiteren Automaten berücksichtigt. In den Zustandsmengen dieser Automaten sind nach Induktionsvoraussetzung aber bereits die bei deren Berechnung zur Verarbeitung hinzugefügten weiteren Automaten kodiert. Daher ergeben sich aus diesem Unterschied in der Berechnungsmethode zwischen den Automaten $(M_i)_{i \in \{1,\ldots,m\}}$ und $\left(\overline{M_i}\right)_{i \in \{1,\ldots,m\}}$ keine Unterschiede in der Sprache der Automaten.

3. Die Ausgangssituation, also der Startzustand, ist für die Automaten (M_i, q_0^i) und $(\overline{M_i}, \overline{q_0^i})$ identisch, da bei beiden zunächst keine weiteren betriebenen Automaten zu berücksichtigen sind, und deshalb das Ergebnis der Queryfunktionen Γ_P und $\overline{\Gamma_P}$ anfangs gleich ist.

4. Durch die in Punkt 2 bereits erwähnten unterschiedlichen Rollen von Protokollautomaten und mitgeführten betriebenen Automaten ergibt sich auch eine unterschiedliche Behandlung von Endzuständen. Während bei Bedingungsautomaten das Erreichen eines Endzustandes des letzten betriebenen Automaten das Akzeptieren eines Eingabewortes signalisiert, ist das Erreichen eines Endzustandes des Protokollautomaten i.A. nur ein vorläufiges Ergebnis. Durch das Speichern der zu diesem Zeitpunkt verbliebenen betriebenen Automaten in der Abbildung conditions, den Test auf Erfüllbarkeit dieser restlichen Querybedingungen gemäß Gleichung 7.10 in Abschnitt 7.4.4.5, und das Weiterbetreiben dieser Automaten beim Erreichen eines Endzustandes der von Queries $\Gamma_{P/V}$ mitgeführten Automaten gemäß den Funktionen $\overline{\delta}$ bzw. δ^* aus den Definitionen 7.5 und 7.6 wird die Auswirkung dieser funktionalen Trennung von Protokollautomat und mitgeführten betriebenen Automaten gegenüber Bedingungsautomaten kompensiert.

5. Wiederholungen werden von Queries $\Gamma_{P/V}$ nur dann erzeugt, wenn eine bereits zuvor bearbeitete Query mit der exakt gleichen Menge von betriebenen Automaten in den jeweils gleichen aktuellen Zuständen bereits berechnet wurde. Durch die dabei eingeführte ϵ–Transition kehrt auch der Protokollautomat beim Akzeptieren eines Eingabewortes an dieser Stelle wieder in den Zustand zurück, in dem er sich zum Zeitpunkt der ersten Queryverarbeitung befunden hatte. Dies entspricht damit in der Sicht eines Bedingungsautomaten einem wiederholten Erreichen eines Gesamtzustandes der Menge von betriebenen Automaten in ihren jeweiligen aktuellen Zuständen. Daher ist auch das Ergebnis der Erkennung von Wiederholungen bei der Queryverarbeitung vollkommen analog zum Betreiben eines Bedingungsautomaten.

6. Aus der funktionalen Trennung von Protokollautomat und mitgeführten betriebenen Automaten ergibt sich noch eine zusätzliche notwendige Anpassung der Queryberechnung, die in den Definitionen aus Abschnitt 7.4 bereits eingeführt wurde. Der Fluß einer Query an einer

Operation, an der ein Programmpfad gewählt oder die Gültigkeit einer Programmeigenschaft angenommen wird, entspricht einem Zustandsübergang des Protokollautomaten. In den Fällen, in denen anhand der Ordnung $\leq_{\widehat{E}}$ erkennbar ist, daß nicht der Protokollautomat, sondern nur einer oder mehrere der betriebenen Automaten einen Zustandsübergang durchführen kann bzw. können, werden diese Zustandsübergänge durch die Funktion IntegrateConditions in den Protokollautomaten aufgenommen. Die Vorgehensweise, solange die Zustandsübergänge der mitgeführten betriebenen Automaten zu berücksichtigen, wie deren zugehörige Eingabesymbole größer als das nächste mögliche Eingabesymbol an den Protokollautomaten alleine ist, entspricht dabei wiederum genau dem Betreiben eines Bedingungsautomaten. Durch die in den Algorithmus IntegrateConditions aus Definition 7.8 integrierte Wiederholungserkennung anhand der Menge ProcessedStates werden Wiederholungen des Gesamtzustandes des Bedingungsautomaten auch dann erkannt, wenn die Query selbst aktuell keinen Zustandsübergang des Protokollautomaten veranlasst.

Damit wurden für den Induktionsschritt die wesentlichen Unterschiede in der Berechnung von M_i und $\overline{M_i}$ vorgestellt, und auf deren Auswirkung auf die Sprachen der jeweiligen Automaten eingegangen. Da dabei jeweils die durch die Definition der Queryverarbeitung anhand der Funktionen Γ_P bzw. Γ_V festgelegte Berechnung des Automaten M_i, und das durch die Definition 5.40 beschriebene Betreibens eines Bedingungsautomaten mit Startzustand $(\overline{M_i}, \overline{q_0^i})$ den exakt gleichen Effekt hat, sowie die Startsituationen gemäß Punkt 3 identisch sind, kann man die Gültigkeit der Aussage für $n-1$ und damit insgesamt für alle $n \in \{0, \ldots, N\}$ folgern, weshalb der Satz bewiesen ist. □

Satz 8.8 (Folgerung: Terminierung der Queryberechnung)
Die Berechnung der Queryfunktionen Γ_P bzw. Γ_V terminiert.

Beweis: Nach Satz 8.7 ist die Sprache der Automaten, die von Queries Γ_P bzw. Γ_V berechnet werden, gleich der Sprache eines Bedingungsautomaten wie dort spezifiziert. Im Beweis des Satzes läßt sich darüberhinaus eine strukturelle Gleichheit der Automaten erkennen.

Als Voraussetzungen in Kapitel 5 sind in dieser Arbeit die Menge von betrachteten Programmeigenschaften und die Menge der Möglichkeiten dieser Programmeigenschaften endlich. Ebenso folgt aus der Argumentation aus Satz 8.4, daß jeder Automat, der sich als Pfadbedingung (im Sinne einer Analyse von *independent attributes*) für diese Programmeigenschaften ergibt, eine endliche Menge von Zuständen besitzt. Bedingungsautomaten können daher maximal so viele Zustände besitzen, wie es verschiedene Kombinationen von Zuständen der einzelnen Automaten gibt. Dies ergibt aufgrund der obigen Endlichkeitsargumentation wieder eine endliche Menge an Zuständen. Da bei der Queryverarbeitung nur Zustände zum Protokollautomat hinzugefügt werden und es keine unendlichen Wege von Queries gibt, auf denen keine neuen Zustände dem Protokollautomaten hinzugefügt werden, sowie die Berechnung nach dem Beweis zum Satz 8.7 genau die Bedingungsautomaten mit endlicher Zustandsmenge ergibt, muß die Berechnung terminieren. □
Der folgende Satz stellt die zentrale Aussage des Korrektheitsbeweises der Queryverarbeitung in dieser Arbeit dar.

Satz 8.9 (Queries berechnen Analyse von *relational attributes*)
Die von Queries gemäß des Algorithmus aus Abschnitt 7.4.3 berechneten Protokollautomaten sind eine Lösung einer Analyse von relational attributes *im Sinne von widerspruchsfreien Pfadbedingungen nach Definition 5.44, d.h. mit den Bezeichnungen aus Satz 8.7 gilt*

$$\forall i \in \{1, \ldots, m\} : M_i \stackrel{r.a.}{\Longrightarrow} (\omega_i \rightarrow \ldots)_{\omega_i}$$

Beweis: Die Berechnung der Queryverarbeitung gemäß des Algorithmus aus Abschnitt 7.4.3 beschreibt nach Satz 8.7 jeweils Automaten, die von ihrer akzeptierten Sprache identisch zu Bedingungsautomaten sind. Die Automaten, die von diesen Bedingungsautomaten gleichzeitig betrieben werden, sind nach Satz 8.1 jeweils Pfadbedingungen für Programmeigenschaften im Sinne einer Analyse von *independent attributes*. Nach Satz 5.9 setzen diese Bedingungsautomaten diese Pfadbedingungen in einer Weise zusammen, daß sie eine exakte Lösung einer Analyse von *relational*

attributes im Sinne von widerspruchsfreien Pfadbedingungen nach Definition 5.44 beschreiben. Da die Queryverarbeitung ein dazu äquivalentes Ergebnis berechnet, ist auch für das Ergebnis der Queryverarbeitung gezeigt, daß es eine exakte Lösung einer Analyse von *relational attributes* darstellt. Jeder der Automaten M_i mit $i \in \{1, \ldots, m\}$ ist daher eine widerspruchsfreie Pfadbedingung für die Programmeigenschaft $(\omega_i \rightarrow \ldots)_{\omega_i}$. □

8.4 Zusammenfassung

Für die von Queries berechneten Protokollautomaten wurde gezeigt, daß sie Sprachen akzeptieren, deren Worte in Bezug auf die Worte aus der Sprache $\mathcal{L}(M_\omega)$, die die Menge von Programminstanzen repräsentieren, die bestimmte Operationen ω erreichen, korrekt geschachtelte Teilworte sind. Damit läßt sich die in Kapitel 5 vorgestellte Theorie auch auf Protokollautomaten anwenden.

Weiter wurde für eine vereinfachte Version der Queryberechnung bewiesen, daß die dabei berechneten Protokollautomaten Pfadbedingungen für die von der Analyse untersuchten Programmeigenschaften darstellen. Dadurch wurde die Behandlung von Mengen von Programminstanzen mit Programmeigenschaften und damit semantischen Eigenschaften des Eingabeprogrammes in Verbindung gebracht.

Schließlich wurde für die Queryverarbeitung bewiesen, daß diese widerspruchsfreie Pfadbedingungen berechnet, und damit eine exakte Lösung einer Analyse von *relational attributes* darstellt.

Kapitel 9

Ökonomie und maximale Analysekosten

Um die praktische Einsetzbarkeit des vorgestellten Verfahrens einschätzen zu können, sind verschiedene Fragestellungen zu untersuchen. Zum einen muß für das Verfahren geklärt werden, inwiefern es ein ökonomisches Berechnungsprinzip realisiert. Zusätzlich ist es notwendig, den Analyseaufwand für die Berechnung von Δ näher einzugrenzen, da ohne weitere Erkenntnisse i.A. selbst der Aufwand zur Berechnung von Δ jenseits der Grenzen der praktischen Realisierbarkeit liegen könnte.

9.1 Ökonomie

9.1.1 Gewählte Darstellungsform des Verfahrens

Der Übergang vom Ergebnis einer Analyse von *independent attributes* zu einer Analyse von *relational attributes* wird von dieser Arbeit auf theoretischer Ebene durch die Einführung von Bedingungsautomaten nach Definition 5.40 geleistet. Daß diese tatsächlich eine Analyse von *relational attributes* realisieren, wurde in Satz 5.9 bewiesen. Damit stellen die Bedingungsautomaten eine theoretische Grundlage für das Verfahren in dieser Arbeit dar. Die in Kapitel 7 vorgestellte Queryberechnung entspricht einer Realisierung dieser theoretischen Grundlage. Der Zusammenhang zwischen den Ergebnissen der Queryauswertung und Bedingungsautomaten wurde in Satz 8.7 gezeigt. Daher kann im Folgenden die Argumentation über Analysekosten und Ökonomie des vorgestellten Verfahrens auf Bedingungsautomaten basieren, wodurch auf die Berücksichtigung einiger Details der konkreten Queryberechnung verzichtet werden kann.

9.1.2 Einordnung des Verfahrens in das Framework

Das genaue Aussehen und die Berechnungskosten der Ergebnisse von Analysen von *independent attributes* und *relational attributes* hängen allgemein von der Art der Analyse und damit von der Art der dabei betrachteten Analysefakten ab. Die folgenden Definitionen legen die Interpretation von Analysefakten, wie sie in dieser Arbeit verwendet werden, anhand einer Einordnung in das Framework aus Definition 2.1 fest.

9.1.2.1 Gültigkeitsbereich von Analysefakten

Die Menge $\mathbf{G}$, die die Programmstellen beschreibt, an denen die einzelnen Analysefakten gültig sein können, entspricht im vorgestellten Analyseverfahren der Menge Ω_P von Operationen, in die nach Definition 6.14 ein Eingabeprogramm gemäß Definition 4.8 transferiert wird.

9.1.2.2 Analysefakten

Ein einzelnes Analysefaktum entspricht dem Ergebnis einer Berechnung der Queryfunktionen $\overline{\Gamma_P}$ bzw. $\overline{\Gamma_V}$ nach Definition 8.1. Dieses beschreibt nach Satz 8.6 eine Pfadbedingung für eine Programmeigenschaft der Form $(\omega \rightarrow \ldots)_\omega$ an einer Operation $\omega \in \mathbf{G} = \Omega_P$. Dabei werden diese Automaten in der Interpretation als Analysefaktum als betriebene Automaten nach Definition 5.12 in ihrem jeweiligen Grundzustand betrachtet, sowie mit einer Menge von Endzuständen versehen, die einem Ergebnis aus der Menge result(F) zugeordnet sind. Die Menge $\mathbf{F}$ von Analysefakten kann man damit wie folgt angeben.

$$\begin{aligned}\mathbf{F} \;=\; \{((Q, E, \delta, q_0, F_y, \text{result}), q_0) \;\mid\; & (Q, E, \delta, q_0, F, \text{result}) \text{ DEA nach Def. 5.9} \\ & \wedge\, \exists y \in \text{result}(F)\ \exists \omega \in \mathbf{G}\ : \\ & \quad (Q, E, \delta, q_0, F_y, \text{result}) \Longrightarrow (\omega \rightarrow y)_\omega\}\end{aligned}$$

9.1.2.3 Kombinationen von Analysefakten

Eine Kombination von Analysefakten entspricht im vorgestellten Verfahren mehreren gleichzeitig mit einem gemeinsamen Eingabewort betriebenen Automaten, die alle in ihrem jeweiligen Startzustand einzelne Analysefakten nach obiger Definition darstellen:

$$\begin{aligned}\mathbf{K} \;=\; & \{\{((Q^1, E, \delta^1, q_0^1, F_{y^1}^1, \text{result}^1), q^1), \ldots, ((Q^n, E, \delta^n, q_0^n, F_{y^n}^n, \text{result}^n), q^n)\} \\ & \mid n \in \mathbb{N} \wedge \forall 1 \leq i \leq n\ :\ q^i \in Q^i \\ & \wedge\, \forall 1 \leq i \leq n\ :\ ((Q^i, E, \delta^i, q_0^i, F_{y^i}^i, \text{result}^i), q_0^i) \in \mathbf{F}\}\end{aligned}$$

Dadurch ist auch die Forderung $\mathbf{F} \subseteq \mathbf{K}$ aus Definition 2.1 erfüllt.

9.1.2.4 Kombinationsabbildung von Analysefakten

Die Kombinationsabbildung $\otimes : \mathbf{F} \times \mathbf{K} \rightarrow \mathbf{K}$ fügt einer Menge von betriebenen Automaten einen weiteren betrieben Automaten $f \in \mathbf{F}$ hinzu, der sich in seinem Startzustand befindet.

9.1.2.5 Menge Δ von Kombinationen

Die Menge Δ wurde in Definition 2.1 als die minimale Menge von Kombinationen von Analysefakten definiert, die zu einer exakten Lösung einer Analyse von *relational attributes* benötigt werden. Daraus leiten sich zwei konkrete Forderungen an die Menge Δ ab:

1. Mittels der in Δ enthaltenen Kombinationen von Analysefakten läßt sich eine exakte Lösung einer Analyse von *relational attributes* beschreiben.
2. Die Menge von Analysefakten in Δ ist minimal, d.h. jedes darin enthaltene Analysefaktum ist zwingend zu einer exakten Lösung notwendig.

Wie diese Menge Δ im angegeben Framework konkret aussieht, wird im folgenden Satz ausgesagt.

Satz 9.1 (Bedingungsautomaten entsprechen Δ)
Die durch die Definition 5.40 eines Bedingungsautomaten spezifizierte Vorgehensweise, eine Menge von Automaten gleichzeitig zu betreiben, resultiert in einer exakten Lösung einer Analyse von relational attributes. *Diese Lösung ist gleichzeitig eine minimale Kombination von Analysefakten nach der Definition aus Abschnitt 9.1.2.3. Damit entsprechen Bedingungsautomaten in der Interpretationsweise dieser Definition der Menge Δ.*

Beweis: Bedingungsautomaten stellen nach Satz 5.9 eine exakte Lösung einer Analyse von *relational attributes* dar. Damit ist der erste Teil des Satzes gezeigt.

Ein Bedingungsautomat entspricht weiter nach der Definition in Abschnitt 9.1.2.3 einer Kombination von Analysefakten, da er mehrere Automaten gleichzeitig betreibt. Um zu beweisen, daß diese Kombination von Analysefakten minimal ist, muß gezeigt werden, daß jedes in dieser Kombination

enthaltene Analysefaktum nach der Definition in Abschnitt 9.1.2.2 für eine exakte Lösung unbedingt notwendig ist.

Der Startzustand des Bedingungsautomaten entspricht einer Pfadbedingung im Sinne einer Analyse von *independent attributes* für die untersuchte Programmeigenschaft. Daher ist dieser betriebene Automat auf jeden Fall für eine exakte Lösung relevant. Weiter dürfen nur solche betriebenen Automaten zur Verarbeitung hinzugefügt werden, die für eine exakte Lösung benötigt werden. Das Hinzufügen von Automaten zur Menge der von diesem Bedingungsautomaten betriebenen Automaten geschieht nach Definition 5.40 genau dann, wenn der Bedingungsautomat ein Eingabesymbol der Form $[\omega \rightarrow y]$ für ein $y \in V$ liest. Betrachtet man die Menge von Programminstanzen, die nach Definition 5.35 von diesem Bedingungsautomaten induziert wird, so muß die Programmeigenschaft $[\omega \rightarrow y]$ aufgrund der Definition von widerspruchsfreien Pfadbedingungen, die Bedingungsautomaten nach Satz 5.9 darstellen, in jeder dieser Programminstanzen gelten. Für das weitere Betreiben des Bedingungsautomaten muß daher gewährleistet werden, daß die vom Bedingungsautomaten ab diesem Symbol akzeptierten Eingabesymbole auch nur eine Menge von Programminstanzen induzieren, in denen diese Eigenschaft auch garantiert gilt. Um dies zu erreichen, muß daher in jedem Fall fortan die Pfadbedingung für diese Programmeigenschaft als betriebener Automat mitgeführt, und von den nachfolgenden Symbolen aus dem Eingabewort in einen Endzustand überführt werden. Die Definition 5.40 eines Bedingungsautomaten sieht genau diese Vorgehensweise bei Eingabe eines solchen Eingabesymbols vor. Damit ist jedes einzelne Analysefaktum, das beim Betreiben eines Bedingungsautomaten zur Menge der betriebenen Automaten hinzugefügt wird, für eine exakte Lösung notwendig. Beim Erreichen eines Endzustandes durch einen der betriebenen Automaten kann nach dem Beweis zu Satz 5.9 gefolgert werden, daß in der vom Bedingungsautomaten induzierten Menge von Programminstanzen die mit dem Automaten assoziierte Programmeigenschaft gilt. Damit ist der betriebene Automat ab diesem Zeitpunkt für eine exakte Lösung nicht mehr relevant. Durch die Definition 5.40 eines Bedingungsautomaten wird dieser dann aus der Kombination von Analysefakten entfernt, weshalb auch dadurch die Minimalität der Menge von betriebenen Automaten gewährleistet werden kann.

Um die Sprache des Bedingungsautomaten bestimmen zu können, müssen die betriebenen Automaten gemeinsam Zustandsübergänge durchführen. Da aufgrund der Definition der Zustandsübergangsfunktion δ von Bedingungsautomaten nach jedem durchgeführten Zustandsübergang noch alle betriebenen Automaten ihre jeweiligen Endzustände erreichen können, müssen sämtliche durch Zustandsübergänge des Bedingungsautomaten entstehenden Kombinationen von Analysefakten zur Bestimmung der Sprache des Bedingungsautomaten auch erzeugt, und weiter betrachtet werden.

Insgesamt entspricht damit die Anzahl der mitgeführten betriebenen Automaten eines Bedingungsautomaten genau der minimalen Anzahl, die für eine exakte Lösung unbedingt benötigt werden. Genau dies wird in Definition 2.1 für die Menge Δ gefordert. Damit ist die Behauptung bewiesen. □

9.1.2.6 Menge Θ von Kombinationen

Die Menge Θ von Kombinationen von Analysefakten ist im Gegensatz zur Menge Δ definiert als die Menge sämtlicher Kombinationen von Analysefakten, die bei einer Analyse berechnet werden, bei der keine Beschränkung auf die Menge Δ der tatsächlich benötigten Kombinationen von Analysefakten vorgenommen wird.

Das in der Form von Bedingungsautomaten vorgestellte Analyseverfahren berechnet nach Satz 9.1 nur die Menge Δ. Aus diesem Blickwinkel ist es zunächst nicht offensichtlich, wie die Menge Θ zu interpretieren ist. Um diese angeben zu können, muß man anscheinend eine Verschlechterung des Verfahrens einführen, die explizit mehr weitere Analysefakten (betriebene Automaten) zur Kombination von Analysefakten (Menge der vom Bedingungsautomaten betriebenen Automaten) hinzufügt, als zu einer exakten Lösung notwendig wäre. Nach der Definition von Θ in Definition 2.1 soll dieses Hinzufügen von Fakten unabhängig davon geschehen, ob das Faktum für eine exakte Lösung benötigt wird. Dies erreicht man, indem man beim Betreiben von Bedingungsautomaten für jedes Eingabesymbol der Form $[\omega \rightarrow y]$ mit $y \in V$, das eine einzelne Programmeigenschaft beschreibt, nicht nur die zugehörige Pfadbedingung zur Menge der gleichzeitig verarbeiteten betriebenen Automaten hin-

zufügt, sondern stattdessen von der gleichzeitigen Gültigkeit von mehreren Programmeigenschaften ausgeht, und daher mehrere betriebene Automaten gleichzeitig zu dieser Menge hinzufügt. Damit würden effektiv Eingabesymbole der Form $[(\omega_1 \rightarrow y_1) \wedge \ldots \wedge (\omega_n \rightarrow y_n)]$ verwendet, wobei für ein $n \in \mathbb{N}$ die Menge $\{\omega_1, \ldots, \omega_n\} = \Omega_{\text{deref}}$ alle im Eingabeprogramm vorkommenden Dereferenzierungsoperationen bezeichnen soll.

9.1.3 Ökonomiebetrachtung

Aus der Einordnung des vorgestellten Verfahrens in das Framework aus Definition 2.1, sowie der Betrachtung der Menge der vom vorgestellten Verfahren berechneten Menge von Kombinationen von Analysefakten, ergibt sich folgender Satz über den Grad der Ökonomie des Verfahrens.

Satz 9.2 (Folgerung: Ökonomiebetrachtung)
Das vorgestellte Verfahren zur Berechnung einer Analyse von relational attributes *ist maximal ökonomisch.*

Beweis: Die Aussage ist eine direkte Konsequenz aus Satz 9.1 und der Definition des Begriffes Ökonomie aus Definition 2.2. □

9.2 Maximale Analysekosten

Basierend auf der Interpretation von Automaten als Analysefakten bzw. Kombinationen von Analysefakten kann man eine Abschätzung der Menge von Kombinationen von Analysefakten finden, die bei ökonomischen bzw. nicht-ökonomischen Verfahren berechnet werden. Diese Abschätzungen lassen sich anhand einer theoretischen worst–case Betrachtung herleiten. Für reale Eingabeprogramme an die Analyse würde man im Gegensatz zum theoretischen worst–case von bestimmten gutartigen Eigenschaften ausgehen. Welcher Art diese Eigenschaften sind, und welche Auswirkung diese auf die Analysekosten haben, wird im darauffolgenden Abschnitt diskutiert werden. Schließlich werden noch die konstruierten Eingabeprogramme aus der Literatur, mit denen gezeigt wurde, daß Zeigeranalyse NP–hart bzw. PSPACE–vollständig ist, unter dem Aspekt dieser Gutartigkeitseigenschaften betrachtet.

9.2.1 Grundlagen

9.2.1.1 Kostenmaße

Die Lösung einer Analyse von *independent attributes* (Berechnung der Queryfunktionen $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ aus Definition 8.1) bzw. *relational attributes* (Betreiben mehrerer solcher Automaten in der Form eines Bedingungsautomaten) wird in dieser Arbeit jeweils durch einen endlichen Automaten dargestellt, der sich als Ergebnis der Queryverarbeitung in der Form eines Protokollautomaten ergibt, bzw. durch die Identifikation der Menge aller aktuellen Zustände der von einem Bedingungsautomaten betriebenen Automaten mit einer einzigen Zustandsmenge nach Definition 5.43.

Für das Ergebnis einer Analyse von *independent attributes* wird in der Interpretation aus dem vorhergehenden Abschnitt genau ein einzelnes Analysefaktum berechnet. Um darüber hinaus Aussagen über den Rechenaufwand treffen zu können, der zur Berechnung dieses Analysefaktums investiert werden muß, bietet es sich an, ein Kostenmaß für einzelne Analysefakten einzuführen. Als solches eignet sich die Größe der Zustandsmenge des berechneten Automaten. Bei der Queryverarbeitung werden fortlaufend neue Zustände in den Protokollautomaten integriert. Daher ergibt eine zeitlich aufwändigere Analyse auch einen Protokollautomaten mit mehr Zuständen, als dies bei einer weniger aufwändigen Analyse der Fall wäre.

Im Gegensatz dazu ergibt sich die Kostenbetrachtung einer Analyse von *relational attributes* direkt aus der Anzahl der Kombinationen von Analysefakten, die benötigt bzw. berechnet werden. Jede einzelne solche Kombination von Analysefakten entspricht einem Gesamtzustand eines Bedingungsautomaten. Die Menge aller möglichen solchen Gesamtzustände, die ein Bedingungsautomat

annehmen kann, entspricht daher der Menge aller Kombinationen von Analysefakten, die im Rahmen einer Analyse berechnet werden.

Obwohl die Interpretation der Kostenmaße für Analysen von *independent attributes* und *relational attributes* unterschiedlich ist, ergibt sich doch in beiden Fällen die Anzahl von Zuständen, die ein (Bedingungs)–Automat besitzt, als einheitliche Maßgröße für Analyselösungen.

9.2.1.2 Annahmen

Die folgende Annahme über Eingabeprogramme betrifft die maximale Referenzierungsstufe von Variablen in Eingabeprogrammen.

Definition 9.1 (Annahme über die maximale Referenzierungsstufe von Variablen)
Die maximale Referenzierungsstufe von Dereferenzierungsoperationen in der Darstellungsform eines Eingabeprogrammes durch Sequenzen von Operationen (und damit von Variablen in einem Programm) werde als unabhängig von der Programmgröße angesehen, und mit $\mathrm{ref}_{\max}$ *bezeichnet.*

Diese Annahme kann man durch die praktische Erfahrung plausibel machen, daß die maximale Referenzierungsstufe von Variablen in einem Programm primär von persönlichem Programmierstil und der Art der Zielsetzung beeinflusst wird.

9.2.1.3 Größe der Darstellung eines Eingabeprogrammes in der neuen Programmrepräsentation

Für die weiteren Aussagen über Analysekosten benötigt man zunächst die folgende Aussage über die Größe der Menge von Operationen, in die ein Eingabeprogramm aus der traditionellen Darstellungsform aus Kapitel 4 durch die Transformation aus Definition 6.14 überführt wird.

Satz 9.3 (Größe der Repräsentation eines Eingabeprogrammes durch Operationen)
Sei $P = (A, B, V, D, b_0)$ *ein Programm gemäß Definition 4.8 und bezeichne* Ω_P *diejenige Menge von Operationen, die sich als Ergebnis* $T_b(b_0)$ *der Transformation dieses Programmes in die neue Programmrepräsentation gemäß Definition 6.14 ergibt. Dann gilt für die Anzahl der Operationen aus* Ω_P

$$|\Omega_P| \leq (c + |V|) \cdot |A| \leq c' \cdot |A| \cdot |V|,$$

wobei $c, c' \in \mathbb{N}$ *Konstanten sind, die nur von der maximalen Referenzierungsstufe* $\mathrm{ref}_{\max}$ *von Variablen in einem Programm abhängen.*

Beweis: Von der Transformation T_b aus Definition 6.14 wird jede Zuweisung in eine Menge von Operationen transformiert, deren Anzahl nur von der Anzahl der Dereferenzierungen der beteiligten Variablen auf beiden Seiten der Zuweisung abhängt. Dies können höchstens so viele sein, wie durch die maximale Referenzierungsstufe von Variablen im Eingabeprogramm vorgegeben wird. Nach der Annahme aus Definition 9.1 kann man diese maximale Referenzierungsstufe als einen festen Wert $\mathrm{ref}_{\max}$ unabhängig von der Größe des Eingabeprogrammes annehmen. Damit kann man auch die Anzahl von Operationen, die insgesamt aus der Umsetzung von solchen Zuweisungen entstehen, mit einer Konstante $c'' \in \mathbb{N}$ nach oben durch $c'' \cdot |A_z| \leq c'' \cdot |A|$, mit A_z wie in Definition 4.5 angegeben, abschätzen.

Für jede `if`– und `while`–Anweisung wird jeweils genau eine subblock(s)–Operation, sowie eine Menge von split– und join–Operationen erzeugt. Da dabei für jede Variable, die in einem der den `if`– oder `while`–Anweisung untergeordneten Programmblöcken vorkommt, je eine eigene split– und join–Operation erzeugt wird, ergibt sich als Umsetzung aller `if`– und `while`–Anweisungen des Eingabeprogrammes eine maximale Anzahl von Operationen, die kleiner oder gleich $(1+|V|)\cdot|A \setminus A_z| \leq (1 + |V|) \cdot |A|$ ist. Insgesamt gilt daher

$$|\Omega_P| \leq (\underbrace{c'' + 1}_{=:c} + |V|) \cdot |A| = (c + |V|) \cdot |A|.$$

Weiter gilt wegen $|V| \geq 1$ (nach Definition 4.8)

$$\begin{aligned}(c+|V|)\cdot|A| &= c\cdot|A| + |A|\cdot|V| \leq c\cdot|A|\cdot|V| + |A|\cdot|V| \\ &= \underbrace{(c+1)}_{=:c'}\cdot|A|\cdot|V| = c'\cdot|A|\cdot|V|\end{aligned}$$

und damit die Behauptung. □

9.2.2 Theoretische Obergrenzen

Mit der obigen Abschätzung für die Anzahl von Operationen, durch die ein Eingabeprogramm repräsentiert wird, kann man Aussagen über die maximalen Kosten von Analysen von *independent attributes* bzw. *relational attributes* machen.

9.2.2.1 Analyse von *independent attributes*

Bei einer Analyse von *independent attributes* werden einzelne Analysefakten im Sinne der Definition aus Abschnitt 9.1.2.2 berechnet. In Abschnitt 8.2 wurde eine Methode vorgestellt, wie mit einer abgewandelten Variante des Verfahrens aus dieser Arbeit eine Analyse von *independent attributes* durch die Queryfunktionen $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ realisiert werden kann. Die maximalen Kosten dieses Verfahrens stellen damit auch eine Obergrenze für die Kosten zur Berechnung einer Analyse von *independent attributes* dar. Diese sollen im Folgenden ermittelt werden.

Satz 9.4 (Zustandsmengengröße bei einer Analyse von *independent attributes*)
Sei ein Programm P und seine Darstellung in der neuen Programmrepräsentation durch Operationen Ω_P, wie in Satz 9.3 definiert, gegeben, und sei $\text{deref}_{\omega',\text{outValue}} \in \Omega_{\text{deref}} \subseteq \Omega_P$ *eine Dereferenzierungsoperation aus dieser Darstellung, sowie q_0 ein Zustand. Bezeichne weiter der Automat*

$$\begin{aligned}&(((Q, E, \delta, q_0, F, \text{result}, \text{conditions}), q), \kappa) \\ &= \overline{\Gamma_P}(\omega', \text{deref}_{\omega',\text{outValue}}, \text{outValue}, (((\{q_0\}, E, \text{undef}, q_0, \emptyset, \text{undef}, \text{undef}), q_0), \emptyset, \text{undef})\end{aligned}$$

das Ergebnis der Queryberechnung der vereinfachten Queryfunktionen $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ gemäß Definition 8.1. Dann gilt für die Größe $|Q|$ der Zustandsmenge Q die folgende Aussage.

$$|Q| \leq c\cdot(|V|\cdot|A|)$$

für eine Konstante $c \in \mathbb{N}$, die nur von der maximalen Referenzierungsstufe von Variablen in einem Programm abhängt.
Beweis: Die Queries $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ berechnen in praktischer Sichtweise offensichtlich das Ergebnis eines Erreichbarkeitsproblems auf dem Analysegraphen, wie er durch Ω_P vorgegeben wird. Bei jedem zweiten Erreichen einer Operation durch eine Query wird die Queryverarbeitung beendet, und an jeder Operation wird maximal ein neuer Zustand des Protokollautomaten erzeugt. Damit ist die Anzahl der Zustände des Protokollautomaten nach oben beschränkt durch die Anzahl der Operationen aus Ω_P. Nach Satz 9.3 ist diese wiederum beschränkt durch $c\cdot|V|\cdot|A|$ mit $c \in \mathbb{N}$ wie dort vorgegeben. Damit gilt die Behauptung. □

Satz 9.5 (Folgerung: Gesamte Zustandsmengengröße)
Bezeichne $N \in \mathbb{N}$ die Größe eines Eingabeprogrammes (z.B. gemessen durch die Anzahl von Zeilen Programmcode) und seien weiter $Q_1, \ldots, Q_n$ für ein $n \in \mathbb{N}$ die Zustandsmengen der Automaten, die sich jeweils wie in Satz 9.4 beschrieben als Ergebnis der Queries $\overline{\Gamma_P}$ für alle Dereferenzierungsoperationen $\omega_1, \ldots, \omega_n$ eines Eingabeprogrammes ergeben. Für das gesamte Ergebnis einer Analyse von independent attributes *der Ziele aller dieser Dereferenzierungsoperationen kann man folgern, daß*

$$\sum_{i=1}^{n} |Q_i| \leq c\cdot N^3$$

für eine Konstante $c \in \mathbb{N}$, die nur von der maximalen Referenzierungsstufe von Variablen in einem Programm abhängt.

Beweis: Nach Satz 9.4 gilt für eine Konstante $c' \in \mathbb{N}$, die nur von der maximalen Referenzierungsstufe von Variablen in einem Programm abhängt:

$$\forall i \in \{1, \ldots, n\} \; : \; |Q_i| \leq c' \cdot |V| \cdot |A|$$

Für die Zustandsmengen, die sich aus der Queryberechnung für sämtliche Dereferenzierungsoperationen ergeben, folgt daraus

$$\sum_{i=1}^{n} |Q_i| \leq c' \cdot |\Omega_{\text{deref}}| \cdot |V| \cdot |A|$$

Da sowohl die Anzahl der Dereferenzierungsoperationen, als auch die Anzahl von Anweisungen und Variablen eines Eingabeprogrammes jeweils aus $\mathcal{O}(N)$ sind, kann man daraus die Behauptung für ein passendes $c \in \mathbb{N}$ folgern. □

Dieses Ergebnis zeigt, daß das vorgestellte Analyseverfahren trotz der gegenüber traditionellen Verfahren insgesamt aufwändigeren Zielsetzung und dem damit verbundenen Aufwand für die Pfadprotokollierung eine Analyse von *independent attributes* mit garantiert polynomiellem Berechnungsaufwand leisten kann. Für Programmeigenschaften, die von keinen anderen Programmeigenschaften abhängen, ist dieses Ergebnis auch eine korrekte Lösung einer Analyse von *relational attributes*.

9.2.2.2 Analyse von *relational attributes*

Für eine Kostenabschätzung einer Analyse von *relational attributes* kann man sowohl das in dieser Arbeit vorgestellte Verfahren, als auch die in Abschnitt 9.1.2.6 eingeführte explizite Verschlechterung dieses Verfahrens mit dem Ziel der Berechnung von Θ betrachten.

9.2.2.2.1 Berechnung von Δ

Nach Satz 9.1 entspricht die Berechnung einer exakten Lösung einer Analyse von *relational attributes* durch Bedingungsautomaten der minimalen Kombination Δ von Analysefakten. Bestimmt man die maximalen Kosten dieser Analyse, gemessen durch die Größe der Menge von Zuständen des Bedingungsautomaten, so stellen diese auch eine Obergrenze für die Berechnung der Menge Δ von Kombinationen im Allgemeinen dar. Im folgenden Satz wird eine Abschätzung für die maximalen Kosten einer solchen Analyse von *relational attributes* gegeben werden.

Satz 9.6 (Maximale Zustandsmengengröße bei einer Berechnung von Δ)
Sei $M = (Q, E, \delta, q_0, F, \text{result})$ ein Automat, der eine Pfadbedingung im Sinne einer Analyse von independent attributes *für eine Dereferenzierungsoperation ω darstellt, d.h. für den $M \Longrightarrow (\omega \rightarrow \ldots)_\omega$ gilt. Sei weiter für ein $n \in \mathbb{N}_0$ eine minimale Menge von Dereferenzierungsoperationen $\{\omega_1, \ldots, \omega_n\}$ mit zugehörigen Pfadbedingungen im Sinne einer Analyse von* independent attributes *der Form $M_1 = (Q_1, E, \delta_1, q_0^1, F_1, \text{result}_1), \ldots, M_n = (Q_n, E, \delta_n, q_0^n, F_n, \text{result}_n)$ mit $M_1 \Longrightarrow (\omega_1 \rightarrow \ldots)_{\omega_1}, \ldots, M_n \Longrightarrow (\omega_n \rightarrow \ldots)_{\omega_n}$ gegeben, die die folgende Eigenschaft erfüllt: für jedes Eingabesymbol der Form $[\omega' \rightarrow \ldots]$, für das einer der Automaten $M, M_1, \ldots, M_n$ einen Zustandsübergang besitzt, gilt $\omega' \in \{\omega_1, \ldots, \omega_n\}$. Bezeichne weiter für $1 \leq i \leq n$ jeweils $r_i := |\text{result}_i(F_i)|$ die Anzahl der verschiedenen Variablen, die den Endzuständen des Automaten M_i zugeordnet sind. Sei weiter C ein Bedingungsautomat mit Startzustand $\{(M, q_0)\}$, und bezeichne Q_C die Zustandsmenge dieses Bedingungsautomaten in dem Sinne, daß gemäß Definition 5.43 alle vorkommenden Zustandskombinationen der gemeinsam betriebenen Automaten jeweils mit einem Zustand aus Q_C identifiziert werden.*

Für die Größe dieser Zustandsmenge gilt dann

$$|Q_C| \leq |Q| \cdot r_1 \cdot 2^{|Q_1|} \cdot r_2 \cdot 2^{|Q_2|} \cdot \ldots \cdot r_n \cdot 2^{|Q_n|}$$

Beweis: Beim Betreiben des Bedingungsautomaten C werden aufgrund der Voraussetzung genau die Automaten $M_1, \ldots, M_n$ zur Menge der betriebenen Automaten hinzugefügt. Der Automat M kann aufgrund von Typbetrachtungen (analog zum Beweis von Satz 8.7) nicht von seiner eigenen Pfadbedingung abhängen. Daher wird er außer als Startzustand nicht erneut als weiterer betriebener Automat zur Menge der vom Bedingungsautomaten gleichzeitig betriebenen Automaten hinzugefügt.

Jeder der anderen Automaten $M_1, \ldots, M_n$ kann im Gegensatz dazu mehrmals in verschiedenen aktuellen Zuständen mitgeführt werden, wenn das Eingabewort an den Bedingungsautomaten das zugehörige Symbol mehrmals enthält. Die Menge aller Zustände eines einzelnen Automaten Q_i für $1 \leq i \leq n$, die damit gleichzeitig in betriebenen Automaten mitgeführt werden können, ergibt sich als die Größe der Potenzmenge von Q_i und damit als $2^{|Q_i|}$. Die verschiedenen möglichen Endzustandsmengen der betriebenen Automaten M_i, die in Definition 5.38 in den Automaten M_y für $y \in V$ verwendet werden, kann man vereinfacht als jeweils r_i verschiedene Automaten interpretieren. Daraus ergeben sich für den Automaten mit der Zustandsmenge Q_i insgesamt $r_i \cdot 2^{|Q_i|}$ Möglichkeiten, als betriebener Automat mitgeführt zu werden.

Die Kombination aller möglichen mitgeführten Automaten, zusammen mit den möglichen Zuständen des Automaten M, ergibt daher die Menge aller Möglichkeiten für Kombinationen von betriebenen Automaten, und damit der maximalen Größe der Zustandsmenge $|Q_C|$, wie dies im Satz beschrieben ist. □

Diese Anzahl $|Q_C|$ von Zuständen ist nach der Einordnung des Verfahrens in das Framework in Abschnitt 9.1.2 auch gleichzeitig die Anzahl von Kombinationen von Analysefakten, und damit gleich $|\Delta|$.

9.2.2.2.2 Berechnung von Θ

Analog zur Betrachtung der Kosten zur Berechnung von Δ im vorigen Abschnitt kann man die Kosten für die Berechnung von Θ abschätzen.

Satz 9.7 (Maximale Zustandsmengengröße bei einer Berechnung von Θ)
Sei $M = (Q, E, \delta, q_0, F, \text{result})$ ein Automat, der eine Pfadbedingung im Sinne einer Analyse von independent attributes *für eine Dereferenzierungsoperation ω darstellt, d.h. für den $M \Longrightarrow (\omega \rightarrow \ldots)_\omega$ gilt. Sei weiter für ein $n \in \mathbb{N}_0$ eine minimale Menge von Dereferenzierungsoperationen $\{\omega_1, \ldots, \omega_n\}$ mit zugehörigen Pfadbedingungen im Sinne einer Analyse von* independent attributes *der Form $M_1 = (Q_1, E, \delta_1, q_0^1, F_1, \text{result}_1), \ldots, M_n = (Q_n, E, \delta_n, q_0^n, F_n, \text{result}_n)$ mit $M_1 \Longrightarrow (\omega_1 \rightarrow \ldots)_{\omega_1}, \ldots, M_n \Longrightarrow (\omega_n \rightarrow \ldots)_{\omega_n}$ gegeben, die die folgende Eigenschaft erfüllt: für jedes Eingabesymbol der Form $[\omega' \rightarrow \ldots]$, für das einer der Automaten $M, M_1, \ldots, M_n$ einen Zustandsübergang besitzt, gilt $\omega' \in \{\omega_1, \ldots, \omega_n\}$. Sei weiter $m \in \mathbb{N}$ mit $m \geq n$ und $\omega_{n+1}, \ldots, \omega_m$ so gewählt, daß die Menge $\{\omega_1, \ldots, \omega_n, \omega_{n+1}, \ldots, \omega_m\}$ alle Dereferenzierungsoperationen des Eingabeprogrammes darstellen. Die Automaten $M_{n+1} = (Q_{n+1}, E, \delta_{n+1}, q_0^{n+1}, F_{n+1}, \text{result}_{n+1}), \ldots, M_m = (Q_m, E, \delta_m, q_0^m, F_m, \text{result}_m)$ seien Pfadbedingungen für die Programmeigenschaften $(\omega_{n+1} \rightarrow \ldots)_{\omega_{n+1}}, \ldots, (\omega_m \rightarrow \ldots)_{\omega_m}$, d.h. es gelte $M_{n+1} \Longrightarrow (\omega_{n+1} \rightarrow \ldots)_{\omega_{n+1}}, \ldots, M_m \Longrightarrow (\omega_m \rightarrow \ldots)_{\omega_m}$.*

Bezeichne weiter für $1 \leq i \leq m$ jeweils $r_i := |\text{result}_i(F_i)|$ die Größe der Ergebnismenge des Automaten M_i. Sei C_Θ ein Bedingungsautomat mit Startzustand (M, q_0), der nach der Argumentation aus Abschnitt 9.1.2.6 zur Beschreibung von Programmeigenschaften nur Eingabesymbole der Form $[(\omega_1 \rightarrow y_1) \wedge \ldots \wedge (\omega_m \rightarrow y_m)]$ akzeptiert, und der dabei sämtliche zu diesen Programmeigenschaften gehörenden Pfadbedingungen zur Menge der betriebenen Automaten hinzufügt. Weiter sei Q_{C_Θ} die Zustandsmenge dieses Bedingungsautomaten in dem Sinne, daß alle vorkommenden Zustandskombinationen der gemeinsam betriebenen Automaten gemäß Definition 5.43 jeweils mit einem Zustand aus Q_{C_Θ} identifiziert werden.

Für die Größe dieser Zustandsmenge gilt dann

$$|Q_{C_\Theta}| \leq |Q| \cdot r_1 \cdot 2^{|Q_1|} \cdot r_2 \cdot 2^{|Q_2|} \cdot \ldots \cdot r_n \cdot 2^{|Q_n|} \cdot r_{n+1} \cdot 2^{|Q_{n+1}|} \cdot \ldots \cdot r_m \cdot 2^{|Q_m|}$$

Beweis: Beim Betreiben des wie im Satz angegeben modifizierten Bedingungsautomaten C_Θ werden aufgrund der Voraussetzung bei jedem Eingabesymbol, das die Gültigkeit einer Programmeigenschaft beschreibt, genau die Automaten $M_1, \ldots, M_m$ zur Menge der betriebenen Automaten hinzugefügt.

Dabei kann wie im Beweis zu Satz 9.6 wieder jeder dieser Automaten mehrmals in verschiedenen aktuellen Zuständen mitgeführt werden. Die Menge aller Zustände eines einzelnen Automaten Q_i für $1 \leq i \leq m$, die damit gleichzeitig in betriebenen Automaten mitgeführt werden können, ergibt sich vollkommen analog zu Satz 9.6 als die Größe der Potenzmenge von Q_i und damit als $2^{|Q_i|}$. Die verschiedenen mit den betriebenen Automaten (M_i, q) assoziierten Endzustandsmengen kann man hier ebenfalls vereinfacht als jeweils r_i verschiedene Automaten interpretieren, woraus sich für den Automaten mit der Zustandsmenge Q_i insgesamt $r_i \cdot 2^{|Q_i|}$ Möglichkeiten ergeben, als betriebener Automat mitgeführt zu werden. Die Kombination aller dieser Automaten, zusammen mit den möglichen Zuständen des Automaten M ergibt daher die Menge aller Möglichkeiten für Kombinationen von mitgeführten Automaten, und damit die maximale Größe der Zustandsmenge $|Q_{C_\Theta}|$, wie dies im Satz beschrieben ist. □

9.2.2.3 Vergleich von Δ und Θ

Der Unterschied zwischen den theoretischen Obergrenzen für Ergebnisse von Analyseverfahren, die die Mengen Δ bzw. Θ berechnen, hängt nur vom Verhältnis der Variablen n zu m in den jeweiligen Sätzen ab. Die Variable n trifft eine Aussage darüber, von wievielen anderen Programmeigenschaften eine konkrete Programmeigenschaft tatsächlich abhängt. Die Variable m beschreibt hingegen die Anzahl aller anderen Programmeigenschaften der Form $(\omega \to \ldots)_\omega$, von denen die betrachtete Programmeigenschaft abhängen könnte. Für die vorgestellte Anwendung der Theorie von Bedingungsautomaten auf Zeigeranalyse ist m gleich der Anzahl der Dereferenzierungen von Variablen im Eingabeprogramm, also anschaulich gesehen gleich der Anzahl von $*$–Operatoren in Anweisungen. Damit ist die Menge Θ sicher nicht für praktisch realisierbare Analysen verwendbar.

Für reale Eingabeprogramme würde man erfahrungsgemäß eine relativ geringe Anzahl von gegenseitigen Abhängigkeiten n von Programmeigenschaften erwarten. Eine genauere Betrachtung dieser Größe, die diese Erwartung bestätigen wird, wird in Abschnitt 9.2.3.3, sowie in Kapitel 11 erfolgen. Daher kann man i.A. davon ausgehen, daß die Menge Δ für die vorgestellte Interpretation von Automaten als Analysefakten signifikant kleiner als die Menge Θ ist.

Geht man für die verschiedenen in das allgemeine Framework aus Definition 2.1 eingeordneten Analyseverfahren von *relational attributes* aus Kapitel 2 von einem in etwa vergleichbaren Aufwand zur Berechnung der Menge Θ von Kombinationen aus, so bedeutet dies, daß das vorgestellte Verfahren im Vergleich zu allen nicht–ökonomischen Verfahren einen signifikant geringeren Berechnungsaufwand für eine exakte Lösung erfordert. Dies ist ein Indiz für die mögliche praktische Anwendbarkeit des Verfahrens. Da zwischen den verschiedenen Verfahren aufgrund ihrer sehr unterschiedlichen Herangehensweise an die Problemstellung aber kein absoluter direkter Vergleich des erforderlichen Berechnungsaufwandes möglich ist, muß die praktische Anwendbarkeit zusätzlich durch andere Nachweise belegt werden. Dies wird in den folgenden Abschnitten diskutiert werden, und in Kapitel 11 durch Messungen an realen Eingabeprogrammen belegt werden.

9.2.3 Gutartigkeit von Eingabeprogrammen

Trotz der Tatsache, daß durch die Berechnung von Δ anstatt von Θ im vorgestellten Verfahren eine deutlich geringere Anzahl von Kombinationen von Analysefakten berechnet werden muß, wächst die theoretische Obergrenze für den Berechnungsaufwand immer noch exponentiell mit der Eingabeprogrammgröße. Da das von Bedingungsautomaten gelöste Problem PSPACE–vollständig ist, kann diese Obergrenze für den Rechenaufwand (zumindest nach aktuellem Kenntnisstand der Informatik) ohnehin kein besseres Ergebnis für den worst–case versprechen.

Im Gegensatz dazu besitzen reale Programme im Vergleich zum worst–case bestimmte Gutartigkeitseigenschaften. Im folgenden Abschnitt sollen diese Eigenschaften bestimmt, und auf ihre Auswirkungen auf die Berechnungskostenobergrenzen untersucht werden. Daß diese Eigenschaften bei realen Programmen auch tatsächlich beobachtet werden können, wird durch experimentelle Ergebnisse in Kapitel 11 belegt werden. In der nachfolgenden Auflistung soll dagegen zunächst nur

argumentativ auf die erwarteten Eigenschaften an sich, sowie deren mögliche Ursachen eingegangen werden.

9.2.3.1 Lokalität

Eine wichtige Einflußgröße auf die Größe der Menge $|Q_C|$ in Satz 9.6 sind die Zustandsmengengrößen der einzelnen Automaten Q und $Q_1, \ldots, Q_n$. Nach Satz 9.4 und Folgerung 9.5 können diese mit der dritten Potenz der Eingabeprogrammgröße anwachsen. Um diesen worst–case tatsächlich zu erreichen, müsste eine konkrete Belegung einer Zeigervariablen, die von einer Dereferenzierungsoperation zu deren Zielauswahl verwendet wird, im Laufe einer Programminstanz als Belegung jeder vorkommenden Variablen an jeder Operation einmal auftreten. Anschaulich beschrieben würde diese konkrete Variablenbelegung den gesamten Analysegraphen durchwandern. Die Anzahl von Zuständen dieser Automaten kann man damit als ein Maß für die Lokalität des Einflußbereiches eines Programmes auf eine betrachtete Programmeigenschaft interpretieren. Als einfache Abschätzung entspricht jeder Zustand eines solchen Automaten der Relevanz des Inhaltes einer Variablen in einem Programmblock für das betrachtete Ergebnis.

Mit etwas Aufwand ließe sich ein Eingabeprogramm künstlich generieren, das den kubischen worst–case der Nicht–Lokalität darstellen würde. Es wäre aber keinesfalls repräsentativ für reale Eingabeprogramme. Bei diesen würde man eher von einer starken lokalen Beschränktheit des Einflussbereiches für eine Programmeigenschaft ausgehen.

Ein Teil dieser Annahme gründet sich in dem Wissen um eine i.A. kurze Lebenszeit von Variablen. Diese Zeit beschreibt die Anzahl von Anweisungen oder Programmblöcken, in denen eine Variable nach einer Zuweisung einen für das restliche Programm relevanten Inhalt besitzt, bevor sie erneut einen anderen Wert zugewiesen bekommt, oder der Inhalt der Variablen nicht mehr ausgelesen wird. Für eine Programmeigenschaft ist jeweils nur eine solche Lebensspanne relevant. Damit kann man erwarten, daß eine Query $\overline{\Gamma_P}$ bzw. $\overline{\Gamma_V}$ nur wenige Programmblöcke durchläuft und hierfür jeweils Zustände in den Protokollautomaten integriert, bevor ein Ergebnis gefunden und die Queryverarbeitung beendet werden kann.

Eine weitere Beobachtung, die man bei realen Eingabeprogrammen machen kann, und die sich ebenfalls in kleinen Zustandsmengen Q bzw. $Q_1, \ldots, Q_n$ bemerkbar macht, betrifft die Anzahl von Programmblöcken, in denen eine Variable sichtbar ist und verwendet wird. Diese Anzahl würde man erfahrungsgemäß als einen eher kleinen Anteil der Menge aller Programmblöcke einschätzen. Sucht eine Query $\overline{\Gamma_V}$ nach der letzten Zuweisung zu einer bestimmten Variablen, so werden nur solche Programmblöcke betreten und dafür auch entsprechende Zustände im Protokollautomaten erzeugt, in denen die Variable verwendet wird. Andernfalls ermöglicht es das Prinzip der korrekt geschachtelten Teilworte aus Definition 5.31, keine Eingabesymbole und damit auch keine zusätzlichen Zustände für die nicht–betretenen Programmblöcke im Protokollautomaten zu integrieren, ohne die Exaktheit des Ergebnisses zu verlieren.

Als Auswirkung des Prinzips der Lokalität, wird man für reale Programme von relativ kleinen Zustandsmengengrößen für die Mengen Q und $Q_1, \ldots, Q_n$ in Satz 9.6 ausgehen, was sich positiv auf die Kosten der Analyse, ausgedrückt durch $|Q_C|$, auswirkt. Diesbezügliche Messungen anhand von realen Programmen sind in Kapitel 11 angegeben.

9.2.3.2 Kleine Ergebnismengen

Eine weitere Einflußgröße in der Abschätzung aus Satz 9.6 sind die Ergebnismengengrößen r_i der einzelnen Automaten für $1 \leq i \leq n$. In der Literatur wurden in [LR92] und [Ruf95] bereits diesbezügliche Messungen vorgenommen. Dabei ergab sich eine durchschnittliche Anzahl von 1.2 Variablen, auf die bei Anweisungen über Zeigervariablen zugegriffen wird. Dieser Wert entspricht damit direkt einem Durchschnittswert für r_i für reale Programme. Diese zu erwartenden kleinen Werte für die r_i haben ebenfalls einen positiven Einfluß auf die maximalen Analysekosten in Satz 9.6.

9.2.3.3 Unabhängigkeit von Programmeigenschaften untereinander

Der Unterschied zwischen den Abschätzungen der maximalen Größe der Mengen Δ und Θ im vorigen Abschnitt war nur abhängig von der Anzahl der weiteren Programmeigenschaften, von denen eine betrachtete Programmeigenschaft abhängt. Eine Programmeigenschaft in der Form des Ziels der Dereferenzierung des Inhaltes einer Zeigervariablen hängt dann von einer anderen solchen Programmeigenschaft ab, wenn der Zeigervariablen als möglichem Ziel einer anderen Zeigervariablen an einer Anweisung ein Wert zugewiesen werden konnte. Ein hoher Grad an Abhängigkeit einer Programmeigenschaft von anderen Programmeigenschaften bedeutet damit, daß viele Dereferenzierungen von verschiedenen Zeigervariablen einen Einfluß auf diese Programmeigenschaft haben können.

Für reale Programme würde man allerdings davon ausgehen, daß man Variablen in funktionale Gruppen unterteilen kann, in denen jeweils wenige Variablen Teil der Realisierung einer gemeinsamen Aufgabe sind. Für Variablen aus unterschiedlichen solchen Funktionsgruppen würde man komplette Unabhängigkeit voneinander erwarten können. Mit dieser Interpretation läßt sich die Anzahl n von Programmeigenschaften, von denen eine betrachtete Programmeigenschaft abhängt, als relativ klein annehmen. Dies ist ein zentraler Faktor, um die Abschätzung der Größe der Menge Q_C in Satz 9.6 für reale Programme relativieren und verkleinern zu können. Eine experimentelle Bestätigung dieser Annahme findet sich in Kapitel 11.

9.2.3.4 Zustandskorrespondenz

Die Teilterme $2^{|Q_i|}$ sind deshalb Teil der Abschätzung in Satz 9.6, da dort davon ausgegangen wird, daß jeder Automat M_i in allen Kombinationen seiner Zustände Q_i von einem Bedingungsautomaten C mitgeführt werden kann. Diese Abschätzung berücksichtigt den theoretischen worst–case, ist aber für reale Eingabeprogramme deutlich zu pessimistisch. Die verschiedenen Automaten mit Zustandsmengen Q_i sind jeweils das Ergebnis von Queryberechnungen. Ein Zustand $q \in Q_i$ eines Automaten repräsentiert i.A. das Bearbeiten einer Query in einem bestimmten Programmblock. Die Zustände verschiedener solcher Automaten, die den gleichen Programmblock vertreten, sind daher auch über einen Zustandsübergang mit dem gleichen Eingabesymbol zu erreichen. Betreibt ein Bedingungsautomat mehrere solche Automaten gemeinsam, dann gehen diese alle durch das gleiche Eingabesymbol in ihren jeweiligen, diesen Programmblock repräsentierenden, Zustand über. Diese Eigenschaft von mehreren Automaten soll als Zustandskorrespondenz bezeichnet werden. Dadurch treten die aktuellen Zustände von betriebenen Automaten in Bedingungsautomaten nicht in beliebigen Kombinationen auf, sondern in Gruppen von jeweils korrespondierenden Zuständen.

Als Voraussetzung für eine Zustandskorrespondenz, bei der jeder Zustand des Automaten M jeweils genau einen korrespondierenden Zustand in den Automaten $M_1, \ldots, M_n$ besitzt, muß der Automat M für jede der korrespondierenden Zustandsgruppen der anderen Automaten unterschiedliche eigene Zustände besitzen. Die Menge der Programmblöcke, die von einer Query in den Automaten M durch das Generieren von Zuständen eingearbeitet wurden, müssen also eine Obermenge der Programmblöcke bilden, die von den Queries, die die Automaten $M_1, \ldots, M_n$ berechnet haben, betreten wurden. Andernfalls kann man den Automaten M anhand der Ordnung $\leq_{\widehat{E}}$ auf Eingabesymbolen um die entsprechenden Zustände erweitern, und erhält einen Automaten $M^{'}$, für dessen Zustandsgröße $|Q^{'}|$ gilt, daß $|Q^{'}| \leq |Q| + \sum_{i=1}^{n} |Q_i|$ ist. Dieser Automat $M^{'}$ mit der zugehörigen Zustandsmengengrößenabschätzung wird als Grundlage der nachfolgenden Abschätzung verwendet werden.

Die Gutartigkeitsannahme der Zustandskorrespondenz reduziert die maximalen Analysekosten erheblich, da anstatt der $2^{|Q_i|}$ Möglichkeiten, einen betriebenen Automaten M_i mitzuführen, dadurch pro Zustand des Automaten $M^{'}$ nur genau jeweils ein korrespondierender Zustand aller Automaten M_i für $1 \leq i \leq n$ existiert, der mit diesem zusammen in Kombinationen von aktuellen Zuständen von betriebenen Automaten vorkommt.

9.2.4 Auswirkung der Gutartigkeitsannahmen auf max. Analysekosten

Auf der Basis der Annahme der Zustandskorrespondenz aus dem vorhergehenden Abschnitt soll im Folgenden eine optimistische Kostenschätzung hergeleitet werden. Daß die Annahmen, die darin eingehen, für reale Eingabeprogramme auch tatsächlich beobachtet werden können, wird in Kapitel 11 belegt werden. Dort werden für Eingabeprogramme mit bestimmten charakteristischen Parametern die Zustandsmengengrößen der berechneten Bedingungsautomaten gemessen, und mit der optimistischen Obergrenze verglichen.

9.2.4.1 Charakteristische Parameter

Zentrale Parameter für die nachfolgende Abschätzung sind die Zustandsmengengrößen $|Q|$, $|Q_1|$, ..., $|Q_n|$. Ebenso werden die Ergebnismengengrößen r_i darin verwendet. Als weiterer zentraler Parameter tritt darin die Anzahl von maximal gleichzeitig durch einen Bedingungsautomaten betriebenen Automaten auf. Dieser Parameter vereint die Aspekte der Lokalität und der Unabhängigkeit von Programmeigenschaften untereinander. Je unabhängiger eine Programmeigenschaft von anderen Eigenschaften ist, desto seltener wird ein weiterer betriebener Automat zur Menge der vom Bedingungsautomaten betriebenen Automaten hinzugefügt. Damit sinkt die Wahrscheinlichkeit, viele Automaten gleichzeitig zu betreiben. Je größer die Lokalität der verwendeten Programmeigenschaften ist, desto früher erreichen die zugehörigen Pfadbedingungen einen Endzustand und werden aus der Menge der betriebenen Automaten wieder entfernt. Beide Aspekte zusammen bewirken damit, daß die Anzahl der gleichzeitig mitgeführten betriebenen Automaten eines Bedingungsautomaten begrenzt wird

9.2.4.2 Optimistische Kostenschätzung

Unter der Annahme von Zustandskorrespondenz ergibt sich die folgende Abschätzung für die Menge Q_C.

Satz 9.8 (Abschätzung von Q_C unter Annahme von Zustandskorrespondenz)
Seien die Voraussetzungen aus Satz 9.6 gegeben. Bezeichne M' einen Automaten, der eine Erweiterung des Automaten M wie in Abschnitt 9.2.3.4 geschildert, darstellt. Sei weiter von den Automaten $M', M_1, \ldots, M_n$ bekannt, daß sie eine Zustandskorrespondenz besitzen, d.h. daß zu jedem Zustand des Automaten M' genau ein Zustand von jedem anderen Automaten $M_1, \ldots, M_n$ existiert, so daß diese gemeinsam die einzigen Kombinationsmöglichkeiten von aktuellen Zuständen in Bedingungsautomaten darstellen. Zusätzlich bezeichne $k \in \mathbb{N}$ die Menge der maximal gleichzeitig vom Bedingungsautomaten C betriebenen Automaten (ausgenommen den betriebenen Automaten aus dem Startzustand des Bedingungsautomaten). Sei $r := \max_{1 \leq i \leq n} r_i$ das Maximum der Ergebnismengengrößen der Automaten $M_1, \ldots, M_n$.

Dann gilt für die maximale Zustandsmenge $|Q_C|$ des Bedingungsautomaten C:

$$|Q_C| \leq |Q'| \cdot (r+1)^k$$

Beweis: Durch die Zustandskorrespondenz entfallen die verschiedenen Möglichkeiten, in welchen Zuständen sich die vom Bedingungsautomaten C betriebenen Automaten $M_1, \ldots, M_n$ befinden können, wenn sich der betriebene Automat M' in einem bestimmten Zustand befindet. Die Anzahl der möglichen Zustände des Bedingungsautomaten hängt daher nur von der Anzahl der verschiedenen Konstellationen ab, in denen die betriebenen Automaten $M_1, \ldots, M_n$ mitgeführt werden können. Jeder dieser Automaten kann in einem konkreten Gesamtzustand des Bedingungsautomaten entweder nicht mitgeführt werden, oder aber mitgeführt werden, und dabei mit einer der r_i verschiedenen Endzustandsteilmengen betrachtet werden. Damit ergeben sich für jeden einzelnen Automaten M_i maximal $r_i + 1$ Möglichkeiten, als betriebener Automat vom Bedingungsautomaten C mitgeführt zu werden. Insgesamt werden nach Voraussetzung maximal k solche Automaten gleichzeitig betrieben. Durch die Definition von r als das Maximum der r_i gibt es für jeden der gleichzeitig betriebenen Automaten maximal $r + 1$ mögliche Konstellationen. Für maximal k gleichzeitig betriebene Automaten ergeben sich daraus höchstens $(r+1)^k$ verschiedene Konstellationen. Daraus ergibt sich die

Behauptung. □

Für die Zahl k, die man als den zentralen Parameter der optimistischen Kostenschätzung in Satz 9.8 ansehen kann, werden in Kapitel 11 Aussagen anhand von realen Programmen gemacht werden.

9.2.5 Einordnung des worst–case anhand der Gutartigkeitsannahmen

In der Literatur existieren zwei Beweise, die Aussagen über die Schwierigkeit von Multi–Level–Zeigeranalyse machen, indem sie bekannte informatische Probleme auf Zeigeranalyse zurückführen. Mit einer exakten Lösung für Zeigeranalyse kann man damit auch das informatische Problem lösen. Damit wird gezeigt, daß Zeigeranalyse mindestens so schwierig wie das Problem ist, das man auf Zeigeranalyse zurückgeführt hat. In beiden Arbeiten werden Programme konstruiert, die man als Eingabeprogramm für eine Zeigeranalyse verwendet. Diese Programme stellen aufgrund der schwierigen informatischen Problemen, die darin codiert sind, worst–case Eingabeprogramme dar. In beiden Fällen steigen die Analysekosten exponentiell mit der Problemgröße, und damit der Eingabeprogrammgröße an.

Die optimistische Abschätzung der Analysekosten aus dem vorhergehenden Abschnitt, zusammen mit einer experimentellen Bestimmung der zentralen Parameter in Kapitel 11, soll dagegen belegen, daß das exponentielle Ansteigen der Analysekosten mit zunehmender Programmgröße für reale Programme nicht zu erwarten ist.

Um diesen scheinbaren Widerspruch aufzuklären, werden im Folgenden die beiden Beweise aus der Literatur kurz vorgestellt, und die darin konstruierten Eingabeprogramme einer Gutartigkeitsanalyse unterzogen. Daraus lassen sich dann Schlüsse ableiten, ob diese Eingabeprogramme Repräsentanten von Problemen sind, die auch in Analysen von realen Programmen auftreten können, oder ob diese in Bezug auf die Gutartigkeitsannahmen unrealistisch sind.

9.2.5.1 Zeigeranalyse ist NP–hart

Der erste Beweis aus der Literatur zeigt, daß Zeigeranalyse mindestens so schwierig ist, wie die Probleme aus der Klasse der NP–vollständigen Probleme. Diese Aussage wird auch als "Zeigeranalyse ist NP–hart" formuliert.

9.2.5.1.1 Ursprung in der Literatur

Landi und Ryder stellen in [LR91] einen Beweis vor, in dem sie ein konkretes Problem aus der Klasse der NP–vollständigen Probleme durch Zeigeranalyse mit Multi–Level–Zeigern lösen, und damit zeigen, daß diese Form von Zeigeranalyse NP–hart ist.

9.2.5.1.2 Zugrundeliegendes theoretisches Problem

Das Problem aus der Klasse der NP–vollständigen Probleme, das sie dabei lösen ist das sogenannte 3Sat–Problem [GJ79]. Das 3Sat–Problem betrachtet als Eingabe einen Booleschen Term, der eine Konjunktion von Klauseln ist, die jede eine Disjunktion von drei jeweils negierten oder nicht–negierten Variablen aus einer Menge $v_1, \ldots, v_n$ ($n \in \mathbb{N}$) von Variablen darstellt. Die Fragestellung besteht darin, herauszufinden, ob es eine Belegung für die vorkommenden Variablen gibt, so daß die Auswertung des Terms "wahr" ergibt. Aus der theoretischen Informatik ist bekannt, daß dieses Problem NP–vollständig ist.

Nach dem aktuellen Stand der Wissenschaft kann man dieses Problem nur lösen, indem man sämtliche Kombinationen von Eingabevariablen ausprobiert. Da diese Menge von Kombinationen exponentiell mit der Problemgröße n ansteigt, muß man für eine Lösung des 3Sat–Problems exponentiell in der Problemgröße ansteigenden Rechenaufwand in Kauf nehmen.

9.2.5.1.3 Art der Reduktion

Eine konkrete Instanz des 3Sat–Problems wird in ein Eingabeprogramm für Zeigeranalyse transformiert. Dazu werden die Variablen $v_1, \ldots, v_n$ durch Zeigervariablen `v1`, `v1q`, ..., `vn`, `vnq` vom Typ `int **` dargestellt. Die Belegung der Variablen wird anhand von zwei weiteren Zeigervariablen `T` und `F` (für *true* und *false*) vom Typ `int *` modelliert. Die Variable v_1 besitzt z.B. dann den Wert *true*, wenn für die Variablen im Programm gilt, daß `v1` $\rightarrow$ `T` $\wedge$ `v1q` $\rightarrow$ `F` gilt. Umgekehrt repräsentiert die Variablenbelegung `v1` $\rightarrow$ `F` $\wedge$ `v1q` $\rightarrow$ `T`, daß die Variable v_1 die Belegung *false* besitzt.

Ein Beispiel für ein solches Programm, das aus dem konkreten Term $(v_1 \vee \neg v_2 \vee v_3) \wedge (v_2 \vee v_1 \vee \neg v_3) \wedge (v_1 \vee v_3 \vee \neg v_2)$ mit drei Variablen und drei Klauseln erzeugt wurde, ist in Abbildung 9.1 dargestellt.

In der ersten Sequenz von If–Anweisungen wird eine konkrete Variablenbelegung durch den durchlaufenen Pfad festgelegt. Als nächstes wird der Variablen `F` die Adresse von `no` zugewiesen. Der darauffolgende Block von If–Anweisungen repräsentiert die Klauseln. Als letzte Anweisung wird die Variable `F` dereferenziert und das Ergebnis der Variablen `result` zugewiesen.

Eine Lösung des 3Sat–Problems erhält man dann, wenn in der letzten Zeile die Variable `F` noch auf die Variable `no` zeigen kann. In diesem Fall repräsentiert ein Pfad durch das Programm diese Lösung. Aus dem ersten Teil des Programmes geht die Variablenbelegung hervor, während aus dem Pfad durch den zweiten Teil des Programmes diejenigen Teile der Klauseln ersichtlich werden, die durch diese Variablenbelegung erfüllt werden.

9.2.5.1.4 Analyse und Ergebnisse

Die grundsätzliche Struktur des Analysegraphen, der aus diesen Eingabeprogrammen erzeugt wird, ist in Abbildung 9.2 dargestellt. Aus Platzgründen sind in der Abbildung nur zwei Variablen und zwei Klauseln gezeigt, die jeweils nur aus zwei statt drei negierten bzw. nicht–negierten Variablen bestehen. Der Graph repräsentiert damit ein "2Sat–Problem" für den Term $(v_1 \vee \neg v_2) \wedge (v_2 \vee v_1)$. Die Erweiterung auf eine Darstellung des eigentlichen Problems ist aber offensichtlich. Zur Verbesserung der Übersichtlichkeit sind in den die Klauseln repräsentierenden Operationsblöcken split– und join–Operationen für alle vorkommenden Variablen dargestellt. Ebenfalls aus Platzgründen wurden die var– und rav–Operationen nicht eingezeichnet. Kanten, die direkt split– und join–Operationen verbinden würden, sind ebenfalls nicht dargestellt.

Die vorgestellte Analysetechnik berechnet zunächst für jede Dereferenzierungsoperation deref_1 bis deref_4 (bzw. deref_1 bis deref_9 im angegebenen 3Sat–Beispiel) die möglichen Ziele. In Abbildung 9.3 ist das Ergebnis der Query abgebildet, die von der Dereferenzierungsoperation deref_1 gestartet wird. Anschließend werden die Ziele der Dereferenzierungsoperation deref_5 (bzw. deref_{10} im angegebenen 3Sat–Beispiel) berechnet. Dazu werden im Klausel–Teil des Programmes je nach von den Queries durchlaufenen Pfaden so viele betriebene Automaten aufgesammelt, wie Klauseln vorhanden sind. Bei der Adressoperation oberhalb des Klauselteils werden bedingte Endzustände für diese Queries erzeugt. Wenn die aufgesammelten betriebenen Automaten einen widerspruchsfreien Pfad durch den Variablenbelegungsteil beschreiben, sind diese bedingten Endzustände echte Endzustände und stellen eine Lösung des 3Sat–Problems dar.

9.2.5.1.5 Einordnung bzgl. Gutartigkeit

Die Faktoren, die in Abschnitt 9.2.3 als Gutartigkeit angeführt wurden, sind Lokalität, kleine Ergebnismengen, Unabhängigkeit von Programmeigenschaften untereinander, sowie Zustandskorrespondenz. Auf diese sollen die konstruierten Eingabeprogramme im Folgenden untersucht werden.

Lokalität Die Automaten, die sich als Ergebnis der Queryverarbeitung für die Dereferenzierungsoperationen deref_1 bis deref_9 ergeben, bestehen jeweils nur aus drei Zuständen. Diese erfüllen damit die Lokalitätseigenschaft. Das Queryergebnis der Dereferenzierungsoperation deref_{10} beinhaltet hingegen Zustände für das Betreten sämtlicher Operationsblöcke des Analysegraphen. Damit besitzt dieser Protokollautomat die Lokalitätseigenschaft nicht. Trotzdem ist dies

```
int main (int argc, char *argv[])
{
  int * * v1, * * v1q;
  int * * v2, * * v2q;
  int * * v3, * * v3q;
  int yes, no;
  int * T, * F;
  int result;
  if (--)
    { v1 = &T; v1q = &F; }
  else
    { v1 = &F; v1q = &T; }
  if (--)
    { v2 = &T; v2q = &F; }
  else
    { v2 = &F; v2q = &T; }
  if (--)
    { v3 = &T; v3q = &F; }
  else
    { v3 = &F; v3q = &T; }

  F = &no;

  if (--)
    { *v1 = &yes; }          // deref1
  else if (--)
    { *v2q = &yes; }         // deref2
  else
    { *v3 = &yes; }          // deref3

  if (--)
    { *v2 = &yes; }          // deref4
  else if (--)
    { *v1 = &yes; }          // deref5
  else
    { *v3q = &yes; }         // deref6

  if (--)
    { *v1 = &yes; }          // deref7
  else if (--)
    { *v3 = &yes; }          // deref8
  else
    { *v2q = &yes; }         // deref9

  result = *F;               // deref10
}
```

Abbildung 9.1: Beispiel für eine Transformation des 3Sat–Problems in Multi–Level–Zeigeranalyse.

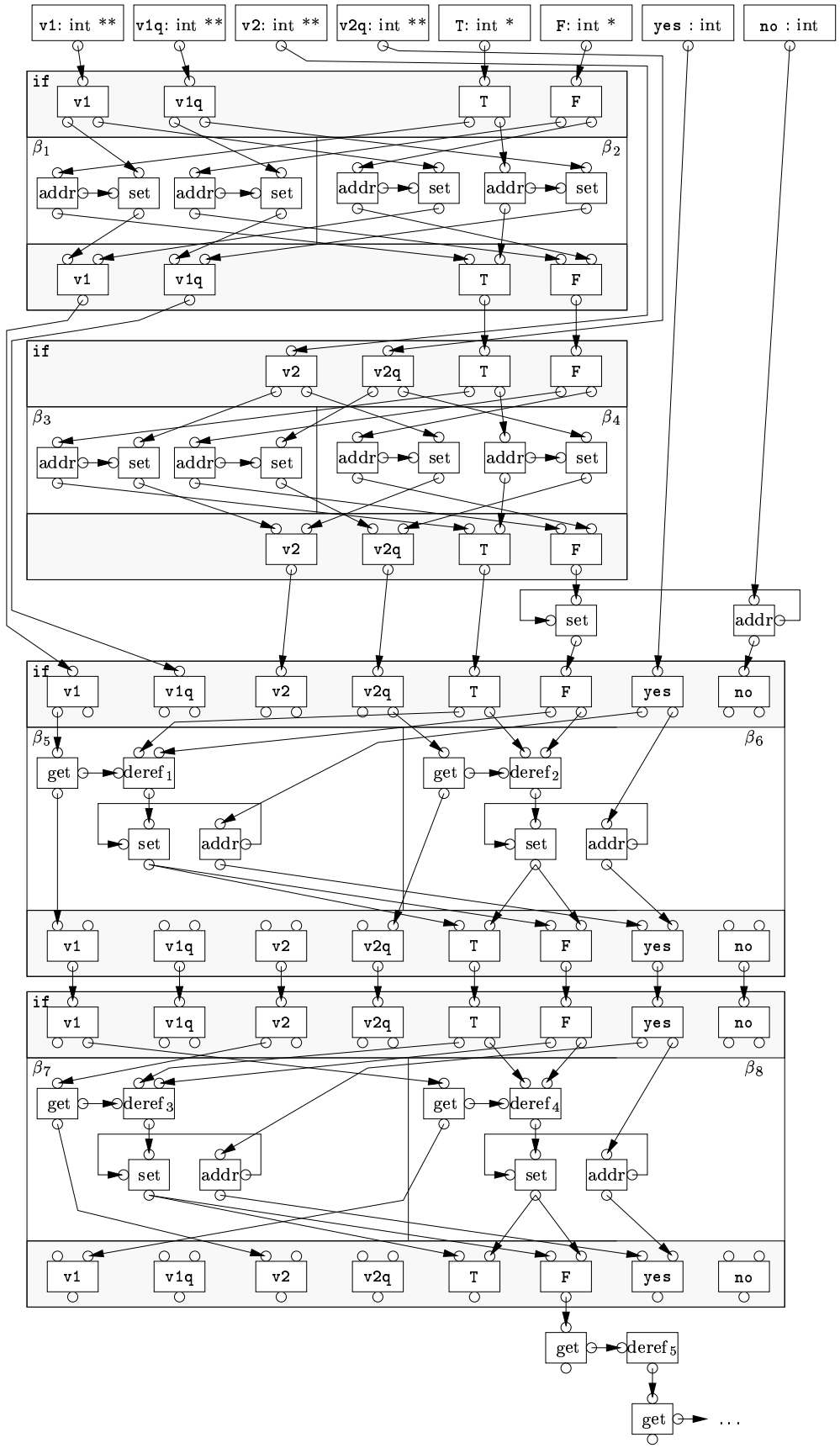


Abbildung 9.2: Vereinfachte Darstellung der Struktur von Analysegraphen für das 3Sat–Problem.

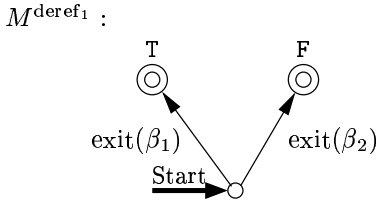


Abbildung 9.3: Teilergebnis der Analyse des 3Sat–Problems.

nicht der ausschlaggebende Faktor dafür, daß der Analyseaufwand exponentiell mit der Problemgröße ansteigt. Die Nicht–Lokalität dieses speziellen Automaten spiegelt sich in der Größe $|Q|$ bzw. $|Q^{'}|$ in den Abschätzungen 9.6 und 9.8 wieder, wodurch aber noch kein exponentielles Wachstum des Berechnungsaufwandes mit zunehmender Problemgröße zu erklären ist.

Kleine Ergebnismengen Die Ergebnismengen der Automaten, die für das 3Sat–Beispiel berechnet werden, bestehen jeweils aus genau zwei verschiedenen Ergebnissen, d.h. es gilt $r_i = 2$ für das Analyseergebnis aller Dereferenzierungsoperationen deref_i im Eingabeprogramm. Damit erfüllen die Eingabeprogramme diese Eigenschaft.

Unabhängigkeit von Programmeigenschaften untereinander Für eine exakte Lösung des 3Sat–Problems müssen sämtliche Programmeigenschaften im Analyseprogramm gemeinsam betrachtet werden. Dies widerspricht der Unabhängigkeitseigenschaft. Damit ist die Anzahl der gleichzeitig betriebenen Automaten der Faktor, der den Analyseaufwand mit zunehmender Problemgröße exponentiell ansteigen läßt. Bei der Queryberechnung für die Dereferenzierungsoperation deref_{10} werden gleichzeitig maximal soviele betriebene Automaten mitgeführt, wie Klauseln im Eingabeprogramm existieren. Damit entspricht die Anzahl der Klauseln im Problem der Größe k in Abschätzung 9.8.

Zustandskorrespondenz Bei Betrachtung der Automaten, die sich aus der Analyse ergeben, erkennt man das Erfülltsein der Zustandskorrespondenzeigenschaft.

Die Eingabeprogramme, die sich als Darstellungen von Instanzen des 3Sat–Problems ergeben, erfüllen alle Gutartigkeitseigenschaften, bis auf die Unabhängigkeit von Programmeigenschaften untereinander. Daher eignen sich diese Eingabeprogramme sehr gut für Messungen der Größe $|Q_C|$ in Abhängigkeit vom Parameter k in Abschätzung 9.8. Solche Messungen werden in Kapitel 11 vorgenommen werden.

Aus Messungen des Parameters k anhand von realen Programmen wird sich zeigen, daß das beliebige Anwachsen des Parameters k mit zunehmender Programmgröße als eher unrealistisch einzuschätzen ist. Damit muß auch für die Analyse von realen Programme kein exponentiell mit der Programmgröße anwachsender Rechenaufwand befürchtet werden.

9.2.5.2 Analyse von *relational attributes* ist PSPACE–vollständig

Im zweiten hier vorgestellten Beweis aus der Literatur wird auf die Schwierigkeit einer Analyse von *relational attributes* im Allgemeinen eingegangen. Von dieser Analysefragestellung wird gezeigt, daß sie in die informatische Komplexitätsklasse der PSPACE–vollständigen Probleme fällt. Von Problemen aus dieser Klasse wird allgemein angenommen, daß sie schwieriger zu lösen sind als NP–vollständige Probleme.

9.2.5.2.1 Ursprung in der Literatur

Daß Multi–Level–Zeigeranalyse PSPACE–hart ist, wurde zuerst von Landi in [Lan92b] unter Verwendung von Zeigervariablen mit Referenzierungsstufe vier gezeigt. Die Veröffentlichung von Muth

und Debray [MD00] beweist PSPACE–Vollständigkeit allgemeiner für alle Analysefragestellungen, die *relational attributes* für eine exakte Lösung benötigen. Darunter fällt, wie in Kapitel 2 bereits beschrieben, auch Multi–Level–Zeigeranalyse. Sie konstruieren dazu ein Eingabeprogramm für eine Konstantenanalyse von *relational attributes*. Anhand dieses Eingabeprogrammes beweisen sie, daß Analysen von *relational attributes* allgemein PSPACE–hart sind. Sie geben informell ein Schema an, um das gleiche Ergebnis auf Zeigeranalyse zu übertragen. Ebenso beweisen sie in der gleichen Veröffentlichung, daß Analysen von *relational attributes* nicht nur gleich schwierig oder schwieriger als die Probleme aus der Klasse der PSPACE–vollständigen Probleme sind (PSPACE–hart), sondern genau in diese Klasse einzuordnen sind (PSPACE–vollständig).

Da das Ergebnis von Muth und Debray allgemeiner als das von Landi ist, und zudem das in ihrem Beweis konstruierte Eingabeprogramm direkt einer Analyse von *relational attributes* mit einer leicht modifizierten Variante des vorgestellten Verfahrens unterzogen werden kann, eignet es sich besser zur Betrachtung der Auswirkungen des theoretischen worst-case auf das vorgestellte Verfahren, als das Ergebnis von Landi. Es soll daher im Folgenden kurz vorgestellt, und anschließend eine auf Konstantenanalyse modifizierte Variante der vorgestellten Analysetechnik darauf angewandt werden. Auf die mögliche Darstellung als Zeigeranalyseproblem anstatt eines Konstantenanalyseproblems wird anschließend kurz eingegangen werden.

Das in der Veröffentlichung von Muth und Debray konstruierte Eingabeprogramm zum Beweis "PSPACE–hart" stellt den worst–case dar, mit dem reale Eingabeprogramme auf der Basis der vorgestellten Gutartigkeitsannahmen verglichen werden sollen. Aus dem Beweis der PSPACE–Vollständigkeit kann man folgern, daß Analysen von *relational attributes* keiner noch schwierigeren Problemklasse angehören, und damit das konstruierte Eingabeprogramm auch tatsächlich den schlimmstmöglichen Fall darstellt.

Die Darstellungsform der nachfolgenden Skizze des Beweises orientiert sich von der Terminologie direkt an [MD00].

9.2.5.2.2 Zugrundeliegendes theoretisches Problem

Das zugrundeliegende Problem, auf das die Analyse von *relational attributes* zurückgeführt wird, ist das Halteproblem für eine polynomiell bandbeschränkte Turingmaschine $TM_p = (Q, \Sigma, \Gamma_{TM}, \delta, q_0, F)$ für eine konkrete Eingabe $x \in \Sigma^*$ (vgl. z.B. [HU79, Kap. 13]), wobei $p \in \mathbb{N}$, Q eine Menge von Zuständen, Σ ein Eingabealphabet, und $\Gamma_{TM} = \{0, 1, \ldots, ns\}$ mit $ns \in \mathbb{N}$ ein Bandalphabet sein sollen, bei dem 0 für das leere Symbol steht. Weiter bezeichnet $\delta \in Q \times \Gamma_{TM} \rightarrow Q \times \Gamma_{TM} \times \{L, R\}$ eine Zustandsübergangsfunktion. Die Symbole L und R bezeichnen dabei die Anweisungen, den Lesekopf der Turingmaschine um ein Symbol nach links bzw. rechts zu bewegen. Der Zustand $q_0 \in Q$ bezeichnet den Startzustand der Turingmaschine, und die Menge F, die o.B.d.A. zu $F = \{q_1\}$ festgelegt wird, bezeichnet die Menge der Endzustände. Die Turingmaschine TM_p wird als polynomiell bandbeschränkt bezeichnet, wenn sie höchstens $|x|^p$ Plätze auf dem Band verwendet, wobei $|x|$ die Länge des Eingabewortes x bezeichnen soll.

Das Halteproblem besteht nun darin, herauszufinden, ob die Turingmaschine TM_p ihre Eingabe x akzeptiert, d.h. durch die Zustandsübergangsfunktion δ unter Eingabe von x von ihrem Startzustand in einen Endzustand übergeht. Dazu muß die Turingmaschine anschaulich gesehen emuliert werden.

9.2.5.2.3 Art der Reduktion

Für ihren Beweis gehen Muth und Debray von Annahmen aus, die o.B.d.A. für die Turingmaschine TM_p gelten sollen. Zum einen soll die Turingmaschine ein zyklisches Band besitzen, dessen Anfang und Ende zusammentreffen. Zum anderen soll die Turingmaschine TM_p den Inhalt des Bandes löschen, bevor sie anhält.

Um die aktuelle Position des Eingabekopfes nicht darstellen zu müssen, modellieren sie eine Kopfbewegung der Turingmaschine durch eine Bewegung des zyklischen Bandes in die Gegenrichtung. Damit liest und schreibt die Turingmaschine stets an der Bandposition 0.

Für eine gegebene Turingmaschine TM_p und ein Eingabewort x konstruieren sie ein Eingabeprogramm $P_{TM_p,x}$, wie in Abbildung 9.4 gezeigt. Dieses Programm enthält drei verschiedene Gruppen von Boole'schen Variablen:

1. Die Variablen $Q_0, \ldots, Q_{nq}$, wobei $nq = |Q| - 1$ die Größe der Zustandsmenge der Turingmaschine bezeichnen soll. Diese Variablen repräsentieren den aktuellen Zustand der Turingmaschine. Befindet sich eine Turingmaschine im aktuellen Zustand $q_i \in Q$, wird dies durch durch die Variablenbelegung
$$Q_k = \begin{cases} 1, & \text{falls } k = i, \\ 0, & \text{sonst} \end{cases}$$
dargestellt.

2. Die Variablen $T_{0,0}, \ldots, T_{nt,ns}$, wobei $nt = |x|^p - 1$ die maximal benötigte Bandlänge, und $ns = |\Gamma_{TM}| - 1$ die Größe des Bandalphabets bezeichnen soll. Diese Variablen beschreiben den aktuellen Bandzustand der Turingmaschine. Die Gruppen von Variablen $T_{i,0}, \ldots, T_{i,ns}$ beschreibt den aktuellen Inhalt des i–ten Bandelementes. Enthält dieses das Bandsymbol j, so wird dies wie folgt durch diese Variablen ausgedrückt:
$$T_{i,k} = \begin{cases} 1, & \text{falls } k = j, \\ 0, & \text{sonst} \end{cases}$$

3. Die Variablen $X_0, \ldots, X_{ns}$ sind temporäre Variablen für das Kopieren des Bandinhaltes bei dessen Rotation.

Im Programm $P_{TM_p,x}$ in Abbildung 9.4 werden die Zustandsübergänge $\delta(q_i, s_j)$ durch die folgende, durch $MOV_{i,j}$ abgekürzte Sequenz von Anweisungen repräsentiert:

$\delta(q_i, s_j) = (q_k, s_m, L)$	$\delta(q_i, s_j) = (q_k, s_m, R)$
Q_i = Q_k;	Q_i = Q_k;
Q_k = 1;	Q_k = 1;
$T_{0,j}$=$T_{0,m}$;	$T_{0,j}$=$T_{0,m}$;
$T_{0,m}$ = 1;	$T_{0,m}$ = 1;
goto copy_left;	goto copy_right;

Dabei aktualisieren die ersten beiden Anweisungen jeweils die Zustandsvariablen, und die nächsten beiden Anweisungen den Inhalt des aktuell betrachteten Bandelementes. Die letzte Zeile initiiert die Rotation des Bandinhaltes durch Umkopieren der entsprechenden Variableninhalte.

Bei einer Ausführung des Programmes aus Abbildung 9.4 werden zunächst die Variablen für den aktuellen Zustand und die Eingabe x auf dem Band initialisiert. Danach wird in einer Schleife, beginnend ab dem Label `Start`, ein Zustandsübergang und ein zugehöriges Eingabesymbol auf dem Band an Position 0 geraten, und entsprechend Zustand und Bandinhalt der Turingmaschine aktualisiert. Wenn der Rateversuch für Zustand oder Bandinhalt falsch ist, werden durch die Konstruktion mehrere Variablen für aktuelle Zustände oder Bandinhalte gleichzeitig auf 1 gesetzt, die Emulation geht in einen inkonsistenten Zustand über, den sie aufgrund der Konstruktion der Anweisungen in $MOV_{i,j}$ nicht mehr verläßt. Eine gültige Konfiguration, in der das Eingabewort akzeptiert worden ist, erkennt man demnach daran, daß nur die Variable Q_1 den Wert 1 besitzt (also der Endzustand erreicht worden ist), und das Band jeweils nur eindeutig leere Symbole enthält.

Die Frage, ob diese konkrete Variablenbelegung am Ende des Programmes angenommen werden kann, entspricht der Fragestellung einer Analyse von *relational attributes* für die entsprechenden Variablenbelegungen. Durch Beantworten dieser Frage kann man durch die angegebene Konstruktion auch das Halteproblem für die polynomiell bandbeschränkte Turingmaschine TM_p lösen. Damit ist eine Analyse von *relational attributes* schwieriger als das Halteproblem, wodurch die Einordnung von ersterer als PSPACE–hartes Problem bewiesen ist.

Für die Umsetzung auf Zeigeranalyse verwenden die Autoren analog zum vorherigen Beweis, daß Zeigeranalyse NP–hart ist, Zeigervariablen, die auf die Variablen *Zero* und *One* (dort T und F) zeigen können. Dadurch erreichen sie eine Übertragbarkeit des vorgestellten Problems mit Boole'schen Variablen auf Zeigeranalyse.

```
/* Program P_{TM_p,x} to emulate a given polynomial space--bounded
   Turing Machine TM_p on input x */
   int Q_0, ..., Q_nq;
   int T_{0,0}, ..., T_{nt,ns};
   int X_0, ..., X_ns;
{
   T_{0,0} = ...; ...; T_{nt,ns} = ...;        /* initialize T_{i,j} based on input string x */
   Q_0 = 1; Q_1 = 0; ...; Q_nq = 0;            /* initial state */
Start:                                         /* emulation loop */
   X_0 = 0; ...; X_ns = 0;                     /* clear temps */
Dispatch:                    /* transitions based on current state and tape symbol */
   if ( - )
   { /* Q_0 == 1? */
      if( - )          { /* T_{0,0} == 1 */  MOV_{0,0}; }                  // jump_{0,0}
      ...
      else if ( - )    { /* T_{0,i} == 1 */  MOV_{0,i}; }                  // jump_{0,i}
      ...
      else if ( - )    { /* T_{0,ns} == 1 */ MOV_{0,ns}; }                 // jump_{0,ns}
   }
   else if ( - ) goto Done;       /* Q_1 == 1?: q_1 = final state */        // jump_{1,0}
   else if ( - )
   { /* Q_2 == 1? */
      ...
   }
   ...
   else if ( - )
   { /* Q_nq == 1? */
      if( - )          { /* T_{0,0} == 1 */  MOV_{nq,0}; }                // jump_{nq,0}
      ...
      else if ( - )    { /* T_{0,i} == 1 */  MOV_{nq,i}; }                // jump_{nq,i}
      ...
      else if ( - )    { /* T_{0,ns} == 1 */ MOV_{nq,ns}; }               // jump_{nq,ns}
   }
   /* copy tape left or right */
copy_right:
   X_0 = T_{0,0}; ...; X_ns = T_{0,ns};
   T_{0,0} = T_{1,0}; ...; T_{0,ns} = T_{1,ns};
   ...
   T_{nt,0} = X_0; ...; T_{nt,ns} = X_ns;
   goto Start;                                                             // jump_{nq+1,0}
copy_left:
   X_0 = T_{nt,0}; ...; X_ns = T_{nt,ns};
   T_{nt,0} = T_{nt-1,0}; ...; T_{nt,ns} = T_{nt-1,ns};
   ...
   T_{0,0} = X_0; ...; T_{0,ns} = X_ns;
   goto Start;                                                             // jump_{nq+2,0}
Done:
   X_0 = 0; ...; X_ns = 0;
End:
}
```

Abbildung 9.4: Reduktion von Zeigeranalyse auf das Halteproblem einer polynomiell bandbeschränkten Turingmaschine.

9.2.5.2.4 Analyse und Ergebnisse

Um das so konstruierte Problem der Konstantenanalyse durch das in dieser Arbeit vorgestellte Verfahren zu lösen, muß man dieses zunächst sinngemäß auf Konstantenanalyse übertragen. Da es in der betrachteten Klasse von Eingabeprogrammen keine arithmetischen Operationen gibt, läßt sich dies realisieren. Die Protokollautomaten beschreiben dabei wie bisher Programmpfade in Rückwärtsrichtung, die von Queries durchlaufen werden. Diese werden an der Stelle im Programm mit dem Label `End` jeweils als Queries Γ_V für jede Variable erzeugt, deren aktuelle Belegung für die zu lösende Fragestellung ermittelt werden soll. Das Beenden der Bearbeitung einer Query wird bei Zuweisungen mit dem Wert 0 oder 1 veranlasst. Dieser zugewiesene Wert wird, analog zur bisherigen Vorgehensweise für Zeigeranalyse, von der result–Funktion einem Endzustand des Protokollautomaten zugeordnet.

Eine Betrachtung des konstruierten Eingabeprogrammes aus dem Blickwinkel der Bedingungsautomaten macht deutlich, daß die Belegungen der einzelnen Variablen aus dem konstruierten Programm komplett unabhängig voneinander sind. In der Ausdrucksweise der Bedingungsautomaten bedeutet dies, daß in den Protokollautomaten der oben beschriebenen Queries keinerlei Symbole der Form $[\rho_\omega^\Omega]$ mit $\rho_\omega^\Omega \in \mathcal{H}_\Omega$ vorkommen. Dies trifft sowohl für das Eingabeprogramm mit Konstantenanalyse, als auch für dessen Übertragung auf Zeigeranalyse nach dem oben angegebenen Schema zu. Selbst eine Analyse von *relational attributes* durch die Queries Γ_V und Γ_P anstatt der Queries $\overline{\Gamma_P}$ und $\overline{\Gamma_V}$ würde daher im Sinne der Aufgabenstellung dieser Arbeit nur jeweils Analysen von *independent attributes* durchführen. Die gesamten Kosten für diese würden sich nach Satz 9.5 nur in der Größenordnung von $\mathcal{O}(N^3)$ bewegen, wenn N die Programmgröße von $P_{TM_p,x}$ bezeichnet.

Die Schwierigkeit der Analyse des Eingabeprogrammes entstammt daher einzig der von außen herangetragenen Fragestellung nach der Belegung sämtlicher Variablen eines Programmes an einer Anweisung. Diese Fragestellung könnte, z.B. durch einen interaktiven Benutzer, auch durch Bedingungsautomaten ausgedrückt werden. In diesem Fall müsste man einen Bedingungsautomaten betreiben, der die Menge sämtlicher einzelner Pfadbedingungen für die korrekte Belegung jeder einzelnen Variable am Ende des Programmes als Startzustand besitzt.

Um sich das Ergebnis einer Analyse des Programmes vorstellen zu können, ist es weiterhin nötig, informell die Erweiterung der bisher vorgestellten Pfadprotokollierungsmethode um Sprungbefehle einzuführen. Dies geschieht an dieser Stelle nur im Sinne eines Ausblicks. Die erwarteten Ergebnisse der Analyse entstammen daher auch keinen realen Analyseberechnungen, sondern sind ohne Unterstützung durch den Implementierungsprototypen manuell berechnet.

Im Programm existieren zwei Verzweigungen, an denen mit Zustandsübergängen im Protokollautomaten eine Pfadprotokollierung des Queryflusses vorgenommen werden muß. An jedem möglichen Sprungziel von `goto`–Anweisungen existieren in Rückwärtsbetrachtungsrichtung Verzweigungen zu allen Stellen im Programm, von denen aus dieses Sprungziel erreicht werden kann. Im konstruierten Programm ist dies an den Labeln `Start` und `Done`, sowie `copy_left` bzw. `copy_right` der Fall. Die Sprungbefehle im Programm in Abbildung 9.4 werden in den Kommentaren am Ende der Zeile in der Form $\texttt{jump}_{i,j}$ für geeignete $i, j \in \mathbb{N}_0$ benannt.

Die Anweisung mit dem Label `Start` wird vom Anfang des Programmes, sowie von den beiden Sprüngen $\texttt{jump}_{nq+1,0}$ und $\texttt{jump}_{nq+2,0}$ erreicht. Das Ende des Programmes mit dem Label `Done` wird nur von dem Sprungbefehl $\texttt{jump}_{1,0}$ angesprungen. Die Labels `copy_left` bzw. `copy_right` werden jeweils aus den Fällen der kaskadierten If–Anweisungen erreicht, in denen der darin repräsentierte Zustandsübergang eine Kopfbewegung nach links bzw. rechts bewirkt.

Die prinzipielle Struktur, die jeder einzelne der Protokollautomaten besitzt, der die Belegung einer einzelnen Variablen am Ende des Programmes beschreibt, ist in Abbildung 9.5 dargestellt. Dabei beschreiben die bisher nicht eingeführten Eingabesymbole $\text{come\_from}(\texttt{jump}_{i,j})$ das Erreichen eines Labels durch einen Sprung $\texttt{jump}_{i,j}$. Das Eingabesymbol no_jump hingegen beschreibt das Erreichen eines Labels durch die normale sequentielle Abarbeitung von Befehlen bzw. Operationen. Vom Startzustand q_0 des Automaten führt ein Zustandsübergang mit dem Eingabesymbol $\text{come\_from}(\texttt{jump}_{1,0})$ zum Zustand q_1. Dieser Zustandsübergang repräsentiert das Erreichen des Labels `Done` durch den mit $\texttt{jump}_{1,0}$ bezeichneten `goto`–Befehl. Die nächsten Zustandsübergänge beschreiben das Erreichen des Labels `Start` von den `goto`–Befehlen $\texttt{jump}_{nq+1,0}$ bzw. $\texttt{jump}_{nq+2,0}$, sowie durch das Symbol no_jump

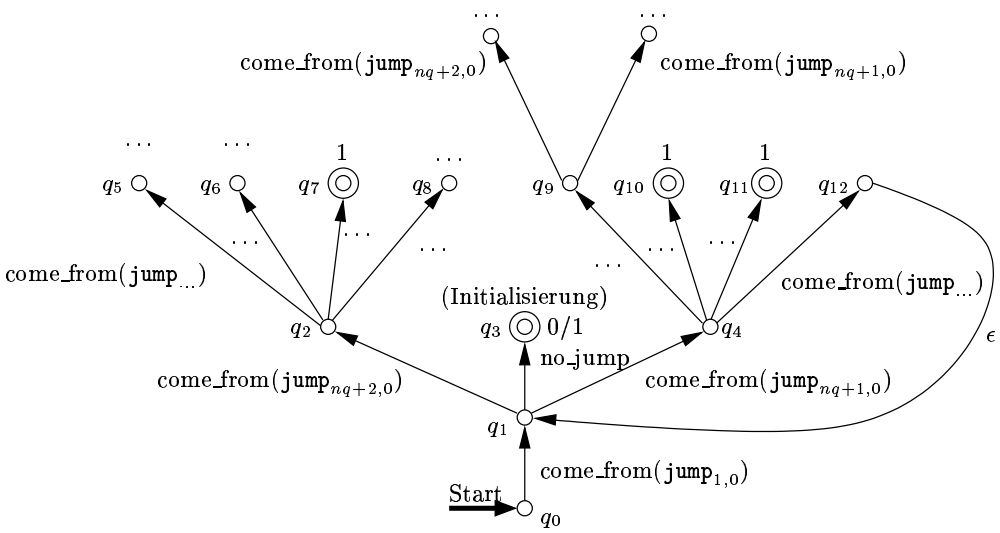


Abbildung 9.5: Struktur eines Automaten, der die Belegung einer einzelnen Variablen am Ende des Programmes $P_{TM_p,x}$ beschreibt.

ausgedrückt, vom Anfang des Programmes. Dort wird die Initialisierung der Variablen mit der Kodierung des Startzustandes, bzw. des Bandzustandes, der das Eingabewort enthält, vorgenommen. Entsprechend ergibt sich der Zustand q_3 als Endzustand mit einem Wert result$(q_3) \in \{0,1\}$ je nach Initialisierung der aktuell von der Query betrachteten Variablen. Die Eingabesymbole $\texttt{jump}_{nq+1,0}$ und $\texttt{jump}_{nq+2,0}$ beschreiben das Ausführen eines Zustandsüberganges, bei dem der Lesekopf der Turingmaschine nach rechts bzw. links bewegt worden ist. Das nächste Eingabesymbol, ausgehend vom Zustand q_2 bzw. q_4, spezifiziert dann, um welchen konkreten Zustandsübergang es sich gehandelt hat. Wird in dem betretenen Programmblock der von der Query betrachteten Variablen der Wert 1 zugewiesen, so wird ein entsprechender Endzustand erzeugt, wie dies in Abbildung 9.5 beispielhaft für die Zustände q_7, q_{10} und q_{11} dargestellt ist. Ansonsten wird die Query auf der Suche nach der letzten Zuweisung zur gleichen Variablen, bzw. ggfs. durch die Zuweisungen Q_i=Q_k bzw. $T_{0,j}$=$T_{0,m}$ in $MOV_{i,j}$ verursacht, auf der Suche nach der letzten Zuweisung zu einer anderen Variablen Q_k bzw. $T_{0,m}$, fortgesetzt. Entsprechend wird der Automat wie vom Zustand q_9 aus beispielhaft gezeigt fortgesetzt. Beim Wiedererreichen des Labels `Start` auf der Suche nach der letzten Zuweisung zu einer Variablen, die von der gleichen Query bereits betrachtet wurde, wird, wie vom Zustand q_{12} aus beispielhaft eingezeichnet, eine Rückwärtskante von der Wiederholungserkennung der Queries erzeugt.

Die Anzahl der Zustände, die ein solcher Protokollautomat besitzen kann, ergibt sich daher als die doppelte Anzahl der Variablen des Programmes $P_{TM_p,x}$, plus den Startzustand q_0.

Interessanterweise ergibt eine Analyse des konstruierten Eingabeprogrammes zunächst eine Menge von Protokollautomaten, die jeweils für eine einzelne Variable eine eindeutige Beschreibung der von der Turingmaschine ausgeführten Zustandsübergänge, sowie der daraus folgerbaren Belegung dieser Variablen ergibt. Ein Kombinieren aller dieser Protokollautomaten durch einen Bedingungsautomaten ergibt dann ein Ergebnis einer Analyse von *relational attributes*. Als Nebeneffekt der Analysetechnik könnte man damit das Halteproblem für polynomiell bandbeschränkte Turingmaschinen für eine konkrete Eingabe rückwärts berechnen.

9.2.5.2.5 Einordnung bzgl. Gutartigkeit

Trotz der Tatsache, daß die gegebene Fragestellung künstlich vorgegeben ist, und damit eigentlich bereits nicht repräsentativ für Fragestellungen bei der Analyse von realen Programmen ist, soll im

Folgenden eine Einordnung des konstruierten Programmes $P_{TM_p,x}$ bezüglich der Gutartigkeitseigenschaften vorgenommen werden.

Lokalität Durch das Umkopieren des Bandinhaltes bei den Labeln `copy_left` und `copy_right` kann sich die aktuelle Belegung einer Variablen aus der jeder anderen Variablen ergeben. Damit stellt das konstruierte Eingabeprogramm den worst–case bzgl. Lokalität dar.

Kleine Ergebnismengen Die Gutartigkeitseigenschaft der kleinen Ergebnismengen ist auch vom konstruierten Eingabeprogramm erfüllt.

Unabhängigkeit von Programmeigenschaften untereinander Die Programmeigenschaften sind nach obiger Argumentation zwar eigentlich alle unabhängig voneinander, durch die von außen herangetragene Fragestellung wird allerdings künstlich eine Abhängigkeit aller Programmeigenschaften untereinander herbeigeführt. Damit ist das konstruierte Eingabeprogramm mit der künstlichen Fragestellung ebenfalls ein worst–case bzgl. der Unabhängigkeit von Programmeigenschaften untereinander.

Die Anzahl k von Automaten, die gleichzeitig vom oben angegebenen Bedingungsautomaten mit dem entsprechend modifizierten Startzustand betrieben werden, ergibt sich dabei als die Zahl der Variablen des Programmes, und damit zu

$$k = |Q| + |x|^p \cdot |\Gamma_{TM}| + |\Gamma_{TM}|$$

Zustandskorrespondenz Auch bezüglich Zustandskorrespondenz stellt das konstruierte Programm den worst–case dar. Das konstruierte Eingabeprogramm besitzt effektiv nur eine einzige charakteristische Verzweigung, die einen von der Turingmaschine ausgeführten Zustandsübergang repräsentiert. Dadurch wird mit jedem Eingabesymbol an den Bedingungsautomaten stets für alle betriebenen Automaten ein Zustandsübergang durchgeführt. Durch das Umkopieren des Bandinhaltes ergeben sich für die Protokollautomaten der Queries viele Möglichkeiten, Rückwärtskanten durch Wiederholungserkennung zu erzeugen. Die Periode, mit der in einem solchen Automaten ein Zustand wiederholt erreicht werden kann, variiert dabei von einem Eingabesymbol bis zu einer Sequenz von Eingabesymbolen, die so lange ist, wie die Anzahl der Variablen, die das Programm besitzt. Durch die unterschiedlichen Perioden der einzelnen Automaten kann man erwarten, daß sehr viele unterschiedliche Kombinationen von aktuellen Zuständen der betriebenen Automaten auftreten, was dem Prinzip der Zustandskorrespondenz widerspricht.

Das konstruierte Eingabeprogramm zum Beweis der PSPACE–Vollständigkeit erfüllt bis auf die Eigenschaft der kleinen Ergebnismengen keine der Gutartigkeitseigenschaften. Im nachfolgenden Kapitel über Ergebnisse wird die Gültigkeit der Gutartigkeitseigenschaften für die dort untersuchten realen Eingabeprogramme nachgewiesen werden. Dadurch ist das konstruierte Eingabeprogramm bezogen auf die Gutartigkeitseigenschaften als unrealistisch im Vergleich zu realen Programmen einzustufen. Speziell die Abhängigkeit von sehr vielen Programmeigenschaften voneinander, die zudem nicht aus der Fragestellung selbst hervorgeht, sondern eine von außen herangetragene künstliche Erschwerung des Problems darstellt, trägt maßgeblich zur Schwierigkeit des Analyseproblems bei. Damit ist für das in dieser Arbeit vorgestellte Verfahren der theoretische worst–case keine Einschränkung der erwarteten praktischen Realisierbarkeit der Analysemethode.

9.3 Zusammenfassung

Für die vorgestellte Analysemethode wurde eine Einordnung in das Framework aus Definition 2.1 vorgenommen. Auf der Basis dieser Einordnung wurde bewiesen, daß das Verfahren maximal ökonomisch ist. Damit erfüllt es eine wichtige Voraussetzung für die praktische Einsetzbarkeit. Weiter wurden theoretische Abschätzungen für maximale Analysekosten für das Verfahren und eine konstruierte, nicht–ökonomische Variante davon hergeleitet. Aus diesen geht hervor, daß die nicht–ökonomische Variante in keinem Fall praktisch einsetzbar sein wird, und daß die maximalen Analysekosten

für das eigentliche Verfahren deutlich geringer einzuschätzen sind. Geht man von einem vergleichbaren Rechenaufwand für alle nicht–ökonomischen Verfahren aus, so bedeutet dies eine deutliche Verbesserung gegenüber den bisher realisierten oder angedeuteten Verfahren aus der Literatur.

Darüberhinaus wurde eine optimistische Kostenschätzung angegeben, die die zu erwartenden maximalen Analysekosten unter der Voraussetzung von Gutartigkeitsannahmen gegenüber der theoretischen Obergrenze nochmals deutlich reduzieren. Die Gutartigkeitseigenschaften wurden vorgestellt, und ihr wahrscheinliches Auftreten plausibel begründet.

Die in der Literatur vorgestellten worst–case Eingabeprogramme wurden ebenfalls in Bezug auf die Gutartigkeitseigenschaften untersucht. Dabei stellte sich heraus, daß diese die Gutartigkeitseigenschaften nur teilweise, bzw. im Fall des konstruierten Eingabeprogrammes zum Nachweis der PSPACE–Vollständigkeit überhaupt nicht erfüllen. Interpretiert man die Gutartigkeitseigenschaften als charakteristische Eigenschaften von realen Eingabeprogrammen, dann muß man für zu erwartende reale Analyseaufgaben nicht mit dem Eintreten des worst–case rechnen.

Insgesamt wurde damit eine argumentative Grundlage dafür geschaffen, warum das vorgestellte Verfahren trotz der theoretischen Schwierigkeit im worst–case sehr gute Voraussetzungen besitzt, um praktisch einsetzbar zu sein. Dadurch basiert die Glaubwürdigkeit der erzielten Ergebnisse nicht nur auf experimentellen Messungen, sondern wird zusätzlich durch eine theoretisch fundierte Begründung der erzielten Verbesserung unterstützt.

Der tatsächliche experimentelle Nachweis der Gutartigkeitseigenschaften für reale Eingabeprogramme, und eine Schätzung der daher zu erwartenden Analysekosten werden in Kapitel 11 vorgestellt werden.

Kapitel 10

Realisierung

Die bisher vorgestellte Beschreibung des Analyseverfahrens bediente sich einer formalen, formelhaften Darstellungsweise. Damit wurde eine klare und eindeutige Beschreibung des relativ komplexen Analyseverfahrens ermöglicht. Um ein Analyseverfahren wie das vorgestellte praktisch anwenden zu können, ist es darüberhinaus aber auch notwendig, daß sich effiziente Implementierungsvarianten für die Bestandteile der Analyse finden lassen. So könnte man z.B. von einer Implementierung der Queryverarbeitung, bei der Repräsentationen von endlichen Automaten vielfach kopiert und weitergeleitet werden, einen sehr großen Verwaltungsoverhead erwarten. Die tatsächliche Vorgehensweise bei der Implementierung des Verfahrens ist jedoch im Gegensatz dazu als sehr effizient einzustufen. Diese Implementierungsdetails vorzustellen, und über den geringen Verwaltungsoverhead zu argumentieren, ist das Ziel dieses Kapitels.

10.1 Überblick

Die Implementierung des Prototyps erfolgte in C++ und Tcl/Tk unter den Betriebssystemen Solaris und Linux. Als Modellierungsgrundlage wurde ein UML–Modell für das Analysetool mit über 100 Klassen erstellt. Als Grundlage für den C++–Parser wurde ein Parsertool mit Namen pccts der Purdue University, zusammen mit einer entsprechenden C++–Grammatik von John Lilley [Lil97] verwendet. Diese besteht aus insgesamt ca. 110.000 Programmzeilen C++–Code, die für die Verwendung in dieser Arbeit modifiziert werden mussten. Das eigentliche Analysetool, das eine interne Repräsentation des analysierten C++–Programmes, die Umwandlung dieser in den Analysegraphen, und die Analyse selbst umfasst, besteht aus ca. 95.000 Zeilen C++-Code.

10.2 Implementierungsdetails

10.2.1 Objektorientierte Analysegraphdarstellung

Als Grundlage für die Queryverarbeitung dient sowohl in der Theorie als auch in der Implementierung der Analysegraph. Um die zentralen Implementierungsdetails vorstellen zu können, muß daher zunächst auch die Repräsentation dieses Analysegraphen in der Implementierung vorgestellt werden. In Abbildung 10.1 ist dargestellt, wie dieser objektorientiert modelliert wird. Die einzelnen Objekte sind dabei unterscheidbare Instanzen von (Objekt–)Klassen, die jeweils Struktur und Verhalten von z.B. allen Kanten, oder allen Get– oder Set–Knoten festlegen. Auf weitere Details der Implementierung, oder Prinzipien von objektorientierter Programmierung soll hier nicht eingegangen werden. Die Veranschaulichung der Realisierung des Analysegraphen genügt bereits, um das Prinzip der Implementierung der Queryverarbeitung vorstellen zu können.

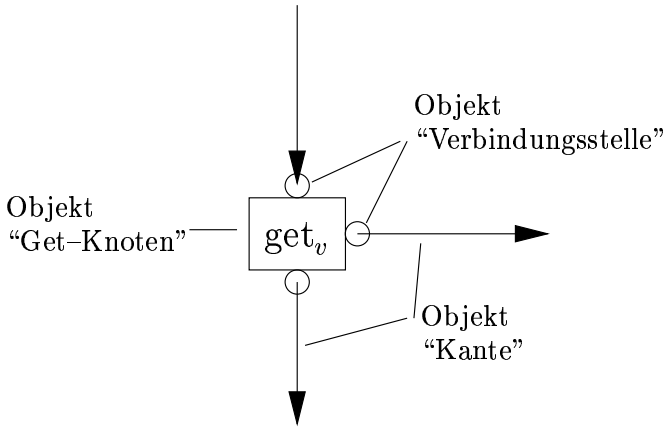


Abbildung 10.1: Objektorientierte Modellierung des Analysegraphen.

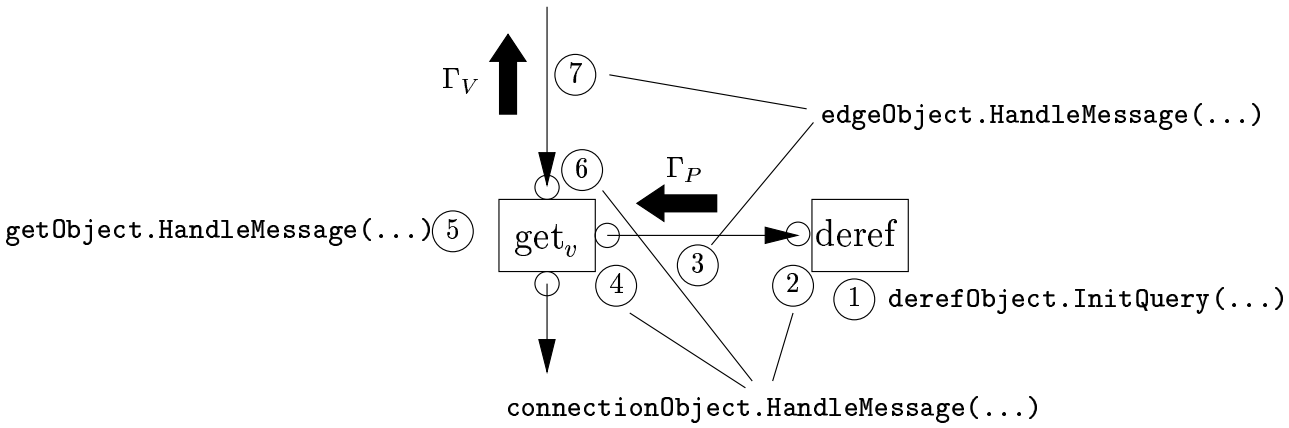


Abbildung 10.2: Implementierung der Queryverarbeitung durch Messages.

10.2.2 Queryverarbeitung durch Messages

In Abbildung 10.2 sind beispielhaft die verschiedenen Verarbeitungsschritte bei der Implementierung der Queryverarbeitung auf der Analysegraphdarstellung gezeigt. Diese Schritte sind nachfolgend erläutert.

1. Die Queryverarbeitung wird für alle Dereferenzierungsoperationen in der in Abschnitt 7.4.3 beschriebenen Reihenfolge initiiert. Dazu wird für jede Objektinstanz, die eine Dereferenzierungsoperation repräsentiert, eine Methode `InitQuery` des Objektes aufgerufen. Einen solchen Aufruf kann man auch als das Versenden einer Nachricht (Message) interpretieren. Die Parameter des Aufrufes bzw. der Message ergeben sich sinngemäß aus der Definition der Queryfunktionen Γ_P und Γ_V, und werden entsprechend hier nicht explizit angegeben. Wie die verwendeten Automaten in der Implementierung dargestellt werden, wird im nächsten Abschnitt erläutert werden.

2. Das Dereferenzierungsoperations–Objekt kennt seine Verbindungsstellen, die es mit anderen Objekten verknüpft. Um die Query Γ_P aus dem Query–Algorithmus aus Abschnitt 7.4.3 zu berechnen, wird an das eingezeichnete Verbindungsstellen–Objekt eine Message versandt. Dies entspricht dem Aufruf einer Methode `HandleMessage` dieses Objektes.

3. Von der Verbindungsstelle wird an alle eingehenden Kanten eine Message geschickt. Dies wird durch einen Methodenaufruf von `HandleMessage` aller dieser Kantenobjekte realisiert.

4. Die Methode `HandleMessage` des Kantenobjektes ruft die Methode `HandleMessage` desjenigen Verbindungsobjektes auf, von dem die Kante ausgeht.

5. Eine eingehende Message an einem Verbindungsobjekt wird an denjenigen Graphknoten weitergeleitet, an dem sich das Verbindungsobjekt befindet. Dies ist im Beispiel in der Abbildung der Get–Knoten. Dieser erhält dabei zusätzlich die Information, an welchem seiner Verbindungsobjekte die Message angekommen ist. Daraus kann die Art der Queryverarbeitung am Knotenobjekt bestimmt werden.

6. Aus der Definition der Queryfunktion Γ_P für Get–Operationen folgt, daß die Query entlang der Use–Use–Chain weitergeleitet werden muß. Dies geschieht analog zu Schritt 2 durch Senden einer Message an das Verbindungsobjekt, das oberhalb des Knotens eingezeichnet ist.

7. Analog zu Schritt 3 wird die Message an einem Kantenobjekt weitergeleitet ...

Man erkennt den direkten Zusammenhang zwischen der rekursiven Definition der Queryfunktionen Γ_P und Γ_V, und der Implementierung dieser Queryfunktionen durch rekursive Methodenaufrufe in einer den Analysegraphen darstellenden Objektstruktur.

10.2.3 Automatenrepräsentation

Bei der Queryverarbeitung werden viele Automaten als Protokollautomaten erzeugt, und später als betriebene Automaten bei der Berechnung weiterer Queries verwendet. Anstatt den rekursiven Funktionsaufrufen von Methoden jeweils eigene Kopien von den diese Automaten repräsentierenden Datenstrukturen mitzugeben, wird bei der vorgestellten Realisierung des Analyseverfahrens nur genau ein Automat pro Dereferenzierungsoperation berechnet. Bei der Verwendung eines solchen Automaten als betriebenen Automaten repräsentieren Zeigervariablen, deren Ziele Instanzen von Klassen "Zustand" sind, die betriebenen Automaten in ihren aktuellen Zuständen. Damit lassen sich sämtliche Aktionen, die mit oder auf Datenstrukturen erfolgen, die endliche Automaten repräsentieren, durch effiziente Zeigeroperationen realisieren.

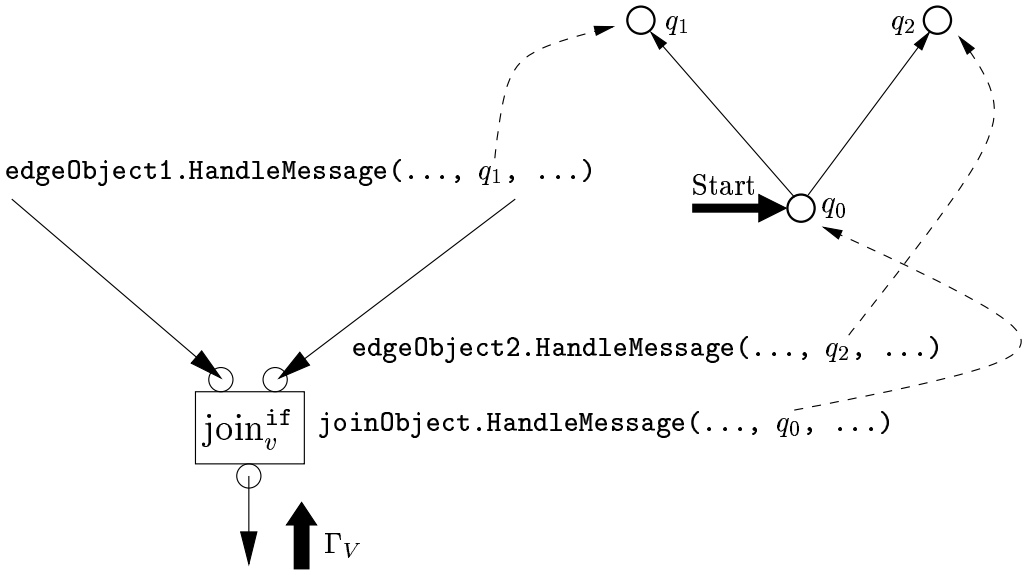


Abbildung 10.3: Gemeinsames Erzeugen eines einzelnen Protokollautomaten durch mehrere rekursive Zweige der Queryberechnung.

10.2.3.1 Aufbau von Protokollautomaten

Der Parameter (M, q) in der Definition der Queryfunktionen Γ_P bzw. Γ_V, der den Protokollautomaten in seinem aktuellen Zustand darstellt, wird durch eine Zeigervariable implementiert, die auf den aktuellen Zustand q (als Instanz das Klasse "Zustand") zeigt. Durch diese Zeigervariable ist der Protokollautomat M selbst ebenfalls bereits eindeutig bestimmt. Abbildung 10.3 zeigt diese Zusammenhang zwischen den aktuellen Zuständen des Protokollautomaten, und den Parametern der Methoden der Objekte, die für das Verarbeiten von Messages zuständig sind. Die gestrichelten Linien stellen dabei Zeigerziele dar. Vom Prinzip ist die Vorgehensweise der Erzeugung des Protokollautomaten analog zu Abbildung 7.2. In der Methode `HandleMessage` des Objektes `joinObject` werden nacheinander die neuen Zustände q_1 und q_2 zusammen mit den eingezeichneten Zustandsübergängen erzeugt, und deren Adressen als Parameter für die beiden Zweige der weiteren Queryberechnung verwendet.

10.2.3.2 Verwendung von betriebenen Automaten

In Abbildung 10.4 ist die Implementierung von betriebenen Automaten durch Zeigeroperationen dargestellt. Links im Bild ist der Aufbau des Protokollautomaten zu sehen, wie dies im letzten Abschnitt vorgestellt wurde. Im rechten Teil der Abbildung sind beispielhaft zwei Automaten M^1 und M^2 eingezeichnet, die von der aktuellen Queryberechnung in ihren jeweiligen Zuständen q_1^1 und q_0^2 mitgeführt werden sollen. Diese Automaten sind bereits vollständig berechnet und liegen als Objektstruktur im Speicher vor. Die aktuellen Zustände dieser Automaten lassen sich damit wiederum durch Zeigervariablen realisieren. Dies wird in der Abbildung durch die gestrichelten Linien dargestellt. Damit lässt sich das aufwändige Erzeugen von Kopien der Objektstrukturen vermeiden, weshalb die Implementierung in Bezug auf das Verwalten der Automatenrepräsentationen als effizient einzustufen ist.

10.2.4 Wiederholungserkennung

Durch die Realisierung der aktuellen Zustände von betriebenen Automaten durch Zeigervariablen läßt sich auch die Wiederholungserkennung der Queries effizient realisieren. Eine Menge von aktuellen

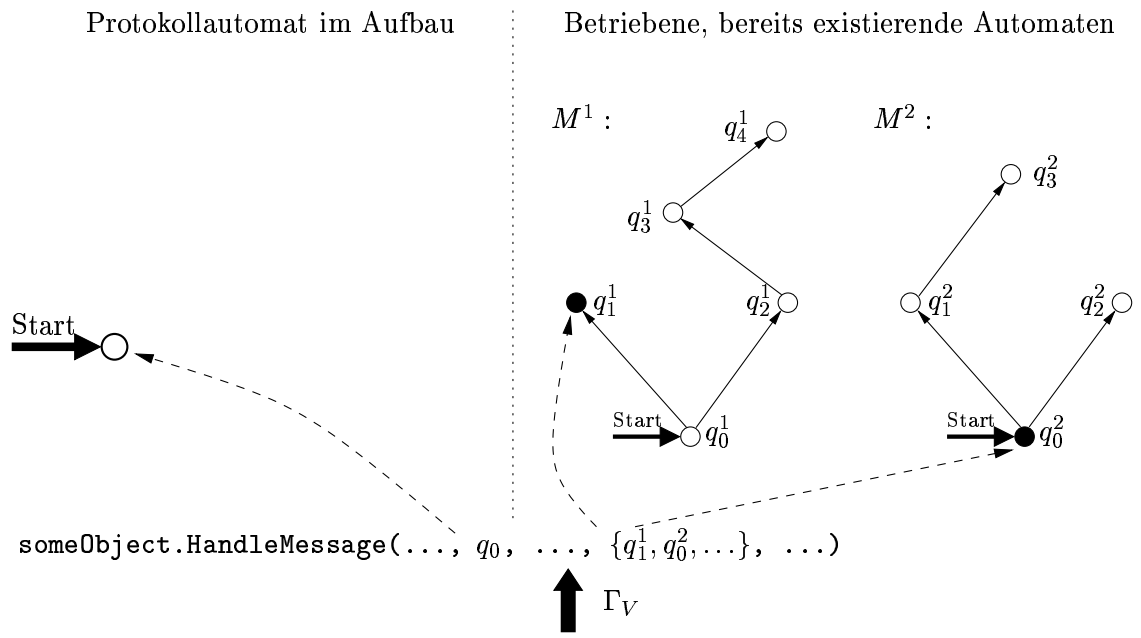


Abbildung 10.4: Implementierung von betriebenen Automaten durch Zeigeroperationen.

Zuständen der betriebenen Automaten, die beim wiederholten Bearbeiten einer Query an einer Operation (bzw. einem Knoten) zum Erkennen einer Wiederholung beidesmal gleich sein müssen, läßt sich als sortierte Liste von Zeigervariablen realisieren. An denjenigen Knoten, an denen eine Wiederholungserkennung durchgeführt werden muß, wird für jede bearbeitete Query ein Tupel, bestehend aus der QueryID (die die betrachtete Dereferenzierungsoperation kennzeichnet), einer sortierten Liste von aktuellen Zuständen der betriebenen Automaten (als Zeigervariablen), sowie dem aktuellen Zustand des Protokollautomaten gespeichert. Beim erneuten Bearbeiten einer Query mit gleicher QueryID am gleichen Knoten wird die Liste der aktuellen Zustände der betriebenen Automaten der Query mit den gespeicherten Listen verglichen, und bei Übereinstimmung ein ϵ–Zustandsübergang, wie in Abschnitt 7.3.3 beschrieben, erzeugt. Durch die Verwendung von sortierten Listen von Zeigeradressen läßt sich der Vergleichsaufwand bei der Wiederholungserkennung klein halten.

10.3 Zusammenfassung

Für das auf theoretischer Basis vorgestellte Analyseverfahren wurden einige Implementierungsdetails vorgestellt. Diese zeigen, daß sich das Analyseverfahren direkt ohne großen Verwaltungsoverhead in eine effiziente Implementierung umsetzen läßt.

Kapitel 11

Experimentelle Untersuchungen

In Kapitel 9 wurden Aussagen über Ökonomie und maximale Analysekosten des vorgestellten Verfahrens gemacht, sowie optimistische Abschätzungen für dessen maximale Analysekosten auf der Basis von Gutartigkeitsannahmen vorgestellt. Diese basierten jeweils auf kombinatorischen Betrachtungen, sowie argumentativer Belegung der Plausibilität von bestimmten Eigenschaften von realen Programmen. Das Ziel dieses Kapitels ist es zu zeigen, daß diese Aussagen und Annahmen auch durch praktische Messungen belegt werden können.

11.1 Ziele

Die Aussagen aus dem Kapitel über Ökonomie und maximale Analysekosten sollen sowohl durch Messungen an realen, als auch an konstruierten Eingabeprogrammen belegt werden. Dadurch wird es möglich sein, zum einen die Gültigkeit der Gutartigkeitseigenschaften für eine betrachtete Menge von realen Programmen nachzuweisen, und daraus allgemeine Schlüsse für die Klasse der realen Programme zu ziehen. Zum anderen ermöglichen Messungen an konstruierten Eingabeprogrammen, die in Abschnitt 9.2.5.1 als worst–case Eingabeprogramme vorgestellt wurden, Schlüsse über die Auswirkung des ökonomischen Prinzips, und die praktischen Grenzen des vorgestellten Verfahrens. Letztere kann man aufgrund der freien Variierbarkeit von zentralen Parametern in konstruierten Eingabeprogrammen gut untersuchen. Konkrete Ziele dieser praktischen Messungen sind damit:

- Belegen der Gültigkeit der Gutartigkeitsannahmen für reale Programme.
- Messung der Reduzierung der Analysezeiten durch das ökonomische Berechnungsprinzip anhand von Extremfällen für Δ.
- Messung der Abhängigkeit des Analyseaufwandes von charakteristischen Parametern aus Abschnitt 9.2.4.1.
- Abschätzung der Analysezeiten für die betrachteten realen Eingabeprogramme.

11.2 Analyse von realen Programmen

Der erste Teil der Messungen befasst sich mit realen Eingabeprogrammen.

11.2.1 Vorgehensweise

Die Eingabeprogramme, die einer Analyse gemäß des vorgestellten Verfahrens unterzogen werden, entstammen einer Menge von C–Programmen, die bereits mehrfach als Standard–Benchmarks für

Zeigeranalysealgorithmen verwendet wurden [LR92] [Ruf95]. Damit wurden diese Eingabeprogramme in diesen Arbeiten bereits als repräsentative Vertreter der Klasse der realen Programme eingeordnet. Die Programmsammlung ist öffentlich im Internet verfügbar [WWW].

Der Sprachumfang beliebiger C–Programme mit Funktionsaufrufen, Zeigerarithmetik und rekursiven Datenstrukturen übersteigt die Mächtigkeit der Eingabesprache aus Kapitel 4. Die Analyse der Eingabeprogramme erfolgt daher auf der Basis von sinngemäßen Erweiterungen des vorgestellten Verfahrens, sofern dies exakt möglich ist, bzw. durch eine vereinfachte Interpretation des Eingabeprogrammes, falls eine exakte Lösung nicht möglich ist. Die einzelnen Aspekte sind dabei nachfolgend aufgeführt.

Interprozedurale Analyse Die Erweiterung des Analyseverfahrens um interprozedurale Aspekte wird in Abschnitt 13.1 als Ausblick vorgestellt. Diese Erweiterung wird bei der Analyse der Programme sinngemäß verwendet. Da die verwendeten Eingabeprogramme keine Rekursionen in Verbindung mit Multi–Level–Zeigern verwenden, läßt sich die Analyse der Programmabschnitte, die für eine Zeigeranalyse von *relational attributes* relevant sind, mit einer endlichen Zustandsmenge, und damit mit einem exakten Ergebnis, realisieren.

Strukturierte Datentypen Eine Erweiterung des Verfahrens auf die Behandlung von strukturierten Datentypen wird im Ausblick in Abschnitt 13.2 vorgestellt. Zeigeranalyse von Programmen mit rekursiven strukturierten Datentypen ist jedoch im Allgemeinen als nicht–berechenbares Problem bekannt [Lan92a] [Ram94]. Dementsprechend kann auch die Erweiterung des vorgestellten Verfahrens hierfür keine exakte Lösung berechnen.

Sofern in den zu analysierenden Eingabeprogrammen rekursive strukturierte Datentypen vorkommen, was ohnehin nur für einen geringen Anteil der Eingabeprogramme aus der oben angegebenen Programmsammlung der Fall ist, wird daher derjenige Anteil der Queries, zu deren Bearbeitung die Behandlung von rekursiven strukturierten Datentypen notwendig wäre, nicht berechnet. Dies stellt keine Einschränkung der Allgemeingültigkeit der Messungen für Programme aus der in dieser Arbeit verwendeten Programmklasse dar.

Zeigerarithmetik Bei Verwendung von Zeigerarithmetik muß zusätzlich zur Information, welche Variable bzw. welches Feld von Variablen Ziel einer Dereferenzierungsoperation sein kann, auch die Information bereitgestellt werden, auf welchen Teil des Zielobjektes der Zugriff erfolgt. In einem Eingabeprogramm geschieht ein solcher Zugriff z.B. über einen Feldindex, der den Ausschnitt des Zielobjektes spezifiziert, der vom Zugriff betroffen ist.

Eine Erweiterung des Verfahrens auf Zeigerarithmetik könnte vom Prinzip sinngemäß wie bei [WL95] erfolgen. Dort wird die Beschreibung eines Zugriffs auf z.B. ein Feld von Variablen aufgeteilt in eine Spezifikation des Feldes selbst, und eine Beschreibung der möglichen Feldindizes, mit denen zugegriffen werden kann. Analog dazu kann man mit dem Verfahren aus dieser Arbeit die Felder als Zeigerziele beschreiben. Zusätzlich müsste man analog zur in Abschnitt 9.2.5.2 beschriebenen Vorgehensweise eine Konstantenanalyse für die möglichen Zugriffsindizes durchführen. Obwohl das in dieser Arbeit vorgestellte Verfahren dabei den exakten Zusammenhang zwischen Zeigerzielen und Indexvariablen im Sinne einer Analyse von *relational attributes* erhalten könnte, und deshalb Zeigerarithmetik prinzipiell zu integrieren wäre, ist Konstantenanalyse mit arithmetischen Operationen mit den Methoden aus dieser Arbeit nicht durch eine endliche Zustandsmenge zu lösen. Daher stellt die Erweiterung des Verfahrens aus dieser Arbeit auf Konstantenanalyse mit arithmetischen Operationen eine nicht–triviale Erweiterung dar. Viele Eingabeprogramme, die Multi–Level–Zeiger verwenden, enthalten jedoch auch Zeigerarithmetik. Dementsprechend muß bei der Analyse von realen Programmen ein Weg gefunden werden, wie man trotz der Verwendung von Zeigerarithmetik in Eingabeprogrammen eine repräsentative Aussage über reale Eingabeprogramme aus der zugelassenenen Programmklasse machen kann. Eine approximative aber sichere Erweiterung des Analysetechnik würde eine andere, schwierigere Fragestellung zu beantworten versuchen, die zunächst wenig mit der Lösung aus dieser Arbeit gemeinsam hätte.

Als Konsequenz wird bei der Analyse von Eingabeprogrammen nur der erste Teil der Fragestellung betrachtet, d.h. daß nur die Variablen bzw. Felder von Variablen, die sich als Zielgebiet von Dereferenzierungsoperationen ergeben können, ermittelt werden. Dabei wird ein Feld von Variablen vereinfachend als eine einzelne Variable interpretiert, der bei jeder Zuweisung mit einem beliebigen Index ein Wert zugewiesen wird. Obwohl dies keine sichere Approximation des schwierigeren Problems ist, wird die Aussage über die prinzipielle Struktur von Eingabeprogrammen davon nicht beeinträchtigt. Die Gutartigkeitsannahmen lassen sich auch mit dieser Vereinfachung verifizieren.

Die in realen Eingabeprogrammen verwendeten Prinzipien wie Funktionsaufrufe, strukturierte Datentypen und Zeigerarithmetik, aber auch Kontrollflußstrukturen wie `goto`–, `break`– und `case`–Anweisungen gehen weit über die in dieser Arbeit verwendete Sprachklasse hinaus. Deshalb ist der Implementierungsprototyp des Verfahrens nicht in der Lage, die Eingabeprogramme zu analysieren. Stattdessen wird die Analyse "von Hand" auf der Basis der oben aufgelisteten Vereinfachungen durchgeführt. Daher kann auch nicht direkt eine Messung der Analysezeit angegeben werden. Stattdessen können aber charakteristische Parameter der Eingabeprogramme bestimmt werden, anhand derer man auf der Basis der Analyse von künstlichen Eingabeprogrammen Rückschlüsse über die zu erwartenden Analysezeiten ziehen kann.

Zur manuellen Analyse gehört zunächst eine Auswahl von Programmabschnitten, in denen eine Verwendung von Multi–Level–Zeigern eine Analyse von *relational attributes* überhaupt erst sinnvoll macht. Reine Single–Level–Zeigeranalyse wird daher von Hand nur für solche Dereferenzierungsoperationen durchgeführt, deren Zeigerziele für eine Analyse von *relational attributes* benötigt werden. Darauf aufbauend erfolgt eine Analyse der Blockstruktur von Eingabeprogrammen. Diejenigen Anweisungen, die für eine Analyse relevant sind, werden anschließend in Analysegraphen umgesetzt. Auf diesen wird dann die Queryverarbeitung unter Berücksichtigung der obigen Erweiterungen und Vereinfachungen durchgeführt. Wenn aus der Analyse der Blockstruktur hervorgeht, daß die zu erwartenden Protokollautomaten eine handhabbare Größe übersteigen, so werden die Zustandsmengengrößen geschätzt. Dies ist entsprechend in den Ergebnistabellen angegeben. Aufgrund dieser äußerst aufwändigen Vorgehensweise ist es nicht realisierbar, eine große Menge von verschiedenen Eingabeprogrammen, oder sehr große Eingabeprogramme zu analysieren. Die Ergebnisse reichen aber aus, um den vorgestellten Analyseansatz als vielversprechend einordnen zu können.

11.2.2 Ergebnisse

In Tabelle 11.1 sind die Ergebnisse der Messungen an realen Eingabeprogrammen aufgelistet. Um den Gegensatz zwischen realen und konstruierten worst–case Eingabeprogrammen aus dem letzten Kapitel darzustellen, sind letztere ebenfalls in der Tabelle aufgeführt.

Die erste Spalte beschreibt das Eingabeprogramm, das jeweils für die Messung verwendet wurde. Dazu gehört zum einen der relative Pfad, den das Eingabeprogramm in der Programmsammlung in Bezug auf die Unterverzeichnisse im Archiv [WWW] besitzt. Zusätzlich sind diejenigen Dateien aufgelistet, in denen Multi–Level–Zeiger verwendet werden, und in denen entsprechend Analysen durchgeführt wurden. In der nächsten Spalte ist die Anzahl von Programmzeilen angegeben, die das gesamte Eingabeprogramm hat. Die mit $\Omega^{(1)}_{\text{deref}}$ überschriebene Spalte enthält die Anzahl der Dereferenzierungsoperationen mit Referenzierungsstufe 1, die bei der Analyse berücksichtigt wurden. Die darauffolgende Spalte gibt die Anzahl von berücksichtigten Dereferenzierungsoperationen mit Referenzierungsstufe 0 an. In den analysierten Programmen kamen keine Dereferenzierungsoperationen mit höheren Referenzierungsstufen vor. Die fünfte Spalte gibt die durchschnittliche Anzahl von Zuständen an, die die Protokollautomaten besitzen, die sich als Ergebnis der Queryberechnung für die Ziele der Dereferenzierungsoperationen aus $\Omega^{(1)}_{\text{deref}}$ mit Referenzierungsstufe 1 ergeben würden. Die nächste Spalte zeigt die gleiche Maßgröße, also die durchschnittliche Anzahl von Zuständen von Protokollautomaten, für die Dereferenzierungsoperationen mit Referenzierungsstufe 0. Letztere sind für die Abschätzung der Kosten einer Analyse von *relational attributes* primär von Interesse, da nur sie von anderen Programmeigenschaften abhängen können, und daher im worst–case nicht–polynomielles Wachstum mit der Eingabeprogrammgröße aufweisen können. Diese Zustandsmenge

Programm	LOC	$\Omega^{(1)}_{\text{deref}}$	$\Omega^{(0)}_{\text{deref}}$	$\overline{\lvert Q_i^{(1)}\rvert}$	$\overline{\lvert Q_i^{(0)}\rvert}$	k
compress compress.c	1497	30	4	20.4	22.0	1
ansitape ansitape.c	1744	2	8	3.0	10.8	1
unix/diff diffdir.c	1814	5	2	3.0	7.0	1
cdecl cdlex.c	3879	7	5	≤ 20	≤ 480	4
agrep parse.c follow.c	3968	21	15	7.8	9.9	1
			...			
NP [LR92]						Anz. Klauseln 3Sat–Problem
PSPACE [MD00]						$\lvert Q\rvert + \lvert\Gamma_{TM}\rvert$ $+\lvert x\rvert^p \cdot \lvert\Gamma_{TM}\rvert$

Tabelle 11.1: Ergebnis der Messungen an realen Eingabeprogrammen und Vergleich mit dem konstruierten worst-case.

stellt damit die Zustandsmenge eines Bedingungsautomaten im Sinne von Definition 5.43 dar. Die mit k übertitelte Spalte beschreibt den Parameter k aus Satz 9.8, der die maximale Anzahl von gleichzeitig betriebenen Automaten dieses Bedingungsautomaten (ausgenommen den betriebenen Automaten aus dem Startzustand des Bedingungsautomaten) angibt.

Die Analyse der Programme compress, ansitape, diff und agrep basiert auf exakten, manuell berechneten Protokollautomaten. Die Ergebnisse für das Programm cdlex sind dagegen Abschätzungen, die wie folgt gewonnen wurden.

Die Datei cdlex.c als Teil des Programmes cdecl ist vom Parsertool lex maschinengenerierter C–Code. Daraus resultiert eine relativ komplexe Programmstruktur, die aus über 50, in zwei geschachtelten Schleifen enthaltenen Programmblöcken, besteht. Eine Analyse der Blockstruktur, sowie sämtlicher Anweisungen auf ihre Relevanz für die Queryverarbeitung, hat die Werte in Tabelle 11.1 für die Anzahl von Dereferenzierungsoperationen mit Referenzierungsstufe 1 und 0, sowie die erwartete maximale Anzahl 20 von Zuständen der Protokollautomaten für Dereferenzierungsoperationen mit Referenzierungsstufe 1 ergeben. Die gleiche Maßgröße für die Protokollautomaten für Dereferenzierungsoperationen mit Referenzierungsstufe 0 ergibt sich aus der optimistischen Kostenschätzung aus Satz 9.8, in die die folgenden weiteren Parameter einfließen. Um diese Abschätzung verwenden zu können, muß zunächst die Zustandskorrespondenzeigenschaft nachgewiesen werden. Deren Gültigkeit kann man auf der Basis der Analyse der Programmstruktur, verbunden mit einer Analyse der gegenseitigen Wechselwirkungen der darin enthaltenen Anweisungen, erkennen. Damit müssen noch die weiteren Parameter der optimistischen Kostenabschätzung ermittelt werden. Den Parameter $|Q|$ aus Satz 9.8 kann man dadurch abschätzen, daß man die Anzahl von Programmblöcken betrachtet, in denen Anweisungen vorkommen, in denen Variablen von passendem Typ mit Referenzierungsstufe 0 oder 1 verwendet werden. Als Ergebnis erhält man eine maximale Zustandsmengengröße von 30 Zuständen für $|Q|$. Der Parameter $k = 4$ ergibt sich aus einer Betrachtung, welche Anweisungen überhaupt Eingabesymbole der Form $[\omega \to \ldots]$ zur Folge haben können. Dies sind genau vier Anweisungen. Daher können auf beliebigen Pfaden durch das Eingabeprogramm maximal vier betriebene Automaten von einer Query mitgeführt werden, bzw. in der theoretischen Betrachtung Teil eines Bedingungsautomaten sein. Damit ergibt sich aus Satz 9.8 mit $k = 4$, und auf der Basis der Zustandskorrespondenzeigenschaft, sowie der Schätzung der Zustandsmengengröße für die einzelnen Pfadbedingungen, eine Obergrenze von 480 Zuständen für jeden der 5 Protokollautomaten für die

Queries nach den Zielen der Dereferenzierungsoperationen aus $\Omega^{(0)}_{\text{deref}}$.

Eine Diskussion der Ergebnisse in Bezug auf die praktische Anwendbarkeit des Verfahrens wird in Abschnitt 11.5 vorgenommen werden. Bis dahin wird auch eine Grundlage für eine Analysezeitabschätzung vorgestellt worden sein. An dieser Stelle soll zunächst die Gültigkeit der Gutartigkeitseigenschaften für die Eingabeprogramme untersucht werden.

11.2.3 Einordnung bzgl. Gutartigkeitseigenschaften

Die Ergebnisse aus dem letzten Abschnitt können dazu verwendet werden, um die Gültigkeit der Gutartigkeitseigenschaften für die analysierten Eingabeprogramme zu belegen.

Lokalität Für die Programme compress, ansitape, diff und agrep sind die durchschnittlichen Zustandsmengengrößen der Protokoll– bzw. Bedingungsautomaten, die in Tabelle 11.1 in den Spalten mit den Überschriften $\overline{|Q_i^{(1)}|}$ und $\overline{|Q_i^{(0)}|}$ eingetragen sind, trotz der Programmgröße von einigen Tausend Zeilen Code auffallend klein. Diese Eingabeprogramme erfüllen damit eindeutig die Lokalitätseigenschaft. Im maschinengenerierten Teil des Programmes cdecl ist nur ungefähr die Hälfte der über 50 geschachtelten Programmblöcke für die Analyse relevant. Zusätzlich beschränkt sich der für die Analyse relevante Bereich des Eingabeprogrammes auf eine einzige Schleife, in der diese 50 Programmblöcke enthalten sind. Damit ist trotz der im Vergleich zu den ersten beiden Eingabeprogrammen größeren Zustandsmenge der Protokollautomaten auch in diesem Eingabeprogramm die Lokalitätseigenschaft erfüllt.

Kleine Ergebnismengen Die Eigenschaft der kleinen Ergebnismengen ist in jedem der Eingabeprogramme erfüllt. Konkret besitzen die Dereferenzierungsoperationen jeweils nur ein mögliches Zeigerziel, was einer minimalen Ergebnismengengröße entspricht.

Unabhängigkeit von Programmeigenschaften untereinander Die Unabhängigkeit von Programmeigenschaften untereinander ist für die Eingabeprogramme in hohem Maße erfüllt. In den Programmen compress, ansitape, diff und agrep hängen die betrachteten Programmeigenschaften maximal von einer anderen Programmeigenschaft ab. Dies ist für Dereferenzierungsoperationen mit Referenzierungsstufe 0 minimal, wenn wie in den Eingabeprogrammen nur Dereferenzierungsoperationen mit Referenzierungsstufen 0 und 1 vorkommen. Die maximale Abhängigkeit von vier anderen Programmeigenschaften im Beispiel cdecl ist ebenfalls ein sehr kleiner Wert im Vergleich zur Anzahl aller prinzipiell möglichen Abhängigkeiten von anderen Programmeigenschaften im Eingabeprogramm.

Zustandskorrespondenz Die Gutartigkeitseigenschaft der Zustandskorrespondenz wird von allen realen Eingabeprogrammen erfüllt. Bei den Programmen compress, ansitape, diff und agrep ist diese anhand der exakt von Hand berechneten Protokollautomaten offensichtlich. Beim Programm cdecl kann man abschätzen, daß alle Protokollautomaten maximal einen Zustand für jeden Programmblock besitzen, in dem die entsprechenden Variablen verwendet werden. Damit korrespondieren diese Zustände miteinander, und es können bei Bedingungsautomaten auch nur diese jeweiligen Zustände miteinander kombiniert werden. Dies entspricht der Definition der Zustandskorrespondenz.

Die Betrachtung von realen Eingabeprogrammen hat damit ergeben, daß die Gutartigkeitseigenschaften bei allen betrachteten Beispielen nachgewiesen werden konnten.

11.3 Messungen anhand von konstruierten Programmen

Die realen Eingabeprogramme aus dem letzten Abschnitt konnten nur mit großem Aufwand von Hand analysiert werden, da sie nicht in der in dieser Arbeit betrachteten Programmklassen enthalten sind. Daher lassen sich keine umfangreichen Untersuchungen über die Abhängigkeit des Analyseaufwandes von zentralen Parametern durchführen. Des weiteren erfüllen alle betrachteten Programme

sehr gut die Gutartigkeitseigenschaft der Unabhängigkeit von Programmeigenschaften untereinander. Daher kann man anhand von ihnen ebenfalls nicht untersuchen, welcher Analyseaufwand für diesbezüglich weniger gutartige Programme zu erwarten ist.

Um diese offenen Fragen zu beantworten, bieten sich konstruierte Eingabeprogramme an, die der zugelassenen Programmklasse angehören. Diese sollen nach Möglichkeit die Gutartigkeitseigenschaften erfüllen, um Schlüsse über reale Programme zuzulassen. Des weiteren sollen sie die Möglichkeit bieten, den zentralen Parameter k aus der optimistischen Kostenabschätzung in Satz 9.8 zu variieren. Die Betrachtung der konstruierten worst–case Eingabeprogramme in Abschnitt 9.2.5 hat ergeben, daß die von Landi und Ryder aus 3Sat–Probleminstanzen konstruierten Eingabeprogramme genau diese Eigenschaften erfüllen. Damit bieten sich diese Eingabeprogramme an, um anhand von ihnen praktische Messungen mit dem Implementierungsprototyp durchzuführen. Diese Messungen haben zwei verschiedene Ziele. Zum einen soll die Abhängigkeit des Analyseaufwandes von der Problemgröße für besonders kleine und besonders große Mengen Δ bestimmt werden. Damit kann man den Analyseaufwand in Extremfällen ermitteln, was einen Anhaltspunkt für das Analyseverhalten von realen Eingabeprogrammen bietet, die "dazwischen" liegen müssen. Ebenso kann man die Verringerung des Analyseaufwandes durch das ökonomische Prinzip durch geeignete Wahl von solchen Extremfällen messen. Als zweites Ziel soll im darauffolgenden Abschnitt die Abhängigkeit des Analyseaufwandes, gemessen in benötigter Analysezeit, vom zentralen Parameter k aus der optimistischen Kostenabschätzung 9.8 gemessen werden.

11.3.1 Extremfälle für Δ

Für die Menge Δ kann man zwei Extremfälle angeben. Die Menge Δ ist dann minimal, wenn für ein exaktes Analyseergebnis nur eine Analyse von *independent attributes* benötigt wird. In diesem Fall müssen keine Kombinationen von Analysefakten gebildet werden, und die Menge Δ entspricht der Menge der einzelnen, unabhängigen Analysefakten. Der zweite Extremfall tritt dann ein, wenn die Menge Δ genauso groß ist wie die Menge Θ, d.h. daß sämtliche möglichen Kombinationen von Analysefakten auch tatsächlich für ein exaktes Ergebnis benötigt werden. Um die Abhängigkeit der Analysezeiten von der Problemgröße in diesen Extremfällen messen zu können, benötigt man Eingabeprogramme, die genau diese Extremfälle verkörpern. Auch hierfür bieten sich die aus 3Sat–Instanzen generierten Eingabeprogramme an.

11.3.1.1 Minimales Δ

Betrachtet man die Darstellungsform der konstruierten Eingabeprogramme in der Veröffentlichung von Landi und Ryder [LR92], so ist die Fragestellung nach den Zeigerzielen von F am Ende des Programmes von außen vorgegeben, ohne daß eine Dereferenzierungsoperation im Eingabeprogramm dieses Ergebnis benötigen würde. In dieser Form existieren im Eingabeprogramm aber nur Dereferenzierungsoperationen mit Referenzierungsstufe 1. Also können die Ergebnisse der Queries für diese Dereferenzierungsoperationen nicht von anderen Programmeigenschaften abhängen, und es genügt eine Analyse von *independent attributes* für eine exakte Lösung. In allen hier vorgestellten Analyseframeworks nach Definition 2.1 stellt damit Δ die kleinstmögliche Menge dar. In der Interpretation von Kombinationen von Analysefakten als Environments von Variablen genügt es, Environments der Größe eins zu verwenden, die jeweils nur Aussagen über die Belegung einer einzelnen Variablen treffen. Dies entspricht in etwa der Vorgehensweise bei traditionellen Analysemethoden von *independent attributes*. In der Interpretationsweise von Analysefakten, wie sie in dieser Arbeit vorgestellt wird, wird für jede Dereferenzierungsoperation ein Protokollautomat erzeugt, der nach Satz 9.4 garantiert nur polynomiell mit der Problemgröße anwächst.

11.3.1.2 Maximales Δ

Nimmt man im konstruierten Eingabeprogramm jedoch als letzte Zeile noch die Dereferenzierung von F hinzu, wie dies in Abbildung 9.1 bereits geschehen ist, so ergibt sich der gegensätzliche Fall für Δ. Die Ziele der Dereferenzierung von F stellen eine Lösung des NP–vollständigen 3Sat–Problems

dar. Um diese zu berechnen, ist eine Analyse von *relational attributes* notwendig. Eine Betrachtung der Menge Δ für verschiedene Analyseframeworks zeigt, daß für deren Berechnung keinerlei Einschränkungen im Vergleich zur Menge Θ von Kombinationen vorgenommen werden kann. In der Interpretation von Kombinationen von Analysefakten als Environments von Variablen werden bei der Analyse im ersten Teil des Eingabeprogrammes alle möglichen Belegungen sämtlicher Variablen in Environments erzeugt, die alle für eine exakte Lösung überprüft werden müssen. Damit ist die Menge Δ hier genauso groß wie die Menge Θ, da keine der Kombinationen von Analysefakten weggelassen werden kann, ohne das exakte Ergebnis zu gefährden. In der Interpretation von Kombinationen von Analysefakten als Bedingungsautomaten ergibt sich das gleiche Ergebnis. Bei der Queryverarbeitung werden alle möglichen Kombinationen von betriebenen Automaten gebildet, die auf allen möglichen Wegen der Query für die Ziele der Dereferenzierung von F durch die die Klauseln darstellenden Programmblöcke berücksichtigt werden müssen. Auch dies stellt den Fall $\Delta = \Theta$ dar.

11.3.1.3 Vorgehensweise

Für eine vorgegebene Anzahl $n \in \mathbb{N}$ von Variablen und $m \in \mathbb{N}$ von Klauseln werden zufällige Instanzen des 3Sat–Problems erzeugt. Diese werden dann automatisch in die Programmdarstellung aus Abbildung 9.1 transformiert. Für eine Instanz des 3Sat–Problems werden zwei unterschiedliche Eingabeprogramme einer Analyse mittels des Implementierungsprototypen unterzogen. Eine Variante (mit Variante 1 bezeichnet) des ansonsten identischen Programmes enthält die letzte Zeile mit der Dereferenzierung von F, die andere Variante (mit Variante 2 bezeichnet) enthält diese Zeile nicht. In diesem Experiment werden die Größen m und n stets gleich gewählt. Dadurch ergeben sich vergleichbare Aussagen über die Mengen Δ und Θ in den verschiedenen vorgestellten Frameworks, da z.B. in der Interpretation von Analysefakten als Environments von Variablen die Anzahl n von Variablen der entscheidende Faktor für die Größe von Δ ist, während bei der Interpretation von Kombinationen von Analysefakten als Bedingungsautomaten die Menge m von Klauseln der zentrale Parameter ist.

Als Plattform für die Laufzeitmessungen wurde dabei ein PC mit 600 MHz AMD Athlon–Prozessor mit 384 Megabytes SDRAM–100 unter Linux verwendet.

11.3.1.4 Ergebnisse

Für Eingabeprogramme mit $n = m \in \{3, \ldots, 20\}$ wurden jeweils die Analyselaufzeiten für beide Varianten von Eingabeprogrammen gemessen, sofern diese eine vertretbare Zeitspanne nicht überschritten haben. Diese Zeiten sind in Tabelle 11.2 dargestellt. Die erste Spalte gibt die Problemgröße in der Form n–m an. Die zweite Spalte listet die Analysezeiten für eine exakte Lösung des 3Sat–Problems auf. In der dritten Spalte sind diejenigen Analysezeiten angegeben, die für eine reine Analyse von *independent attributes* bei Weglassen der Programmzeile mit der Dereferenzierung von F benötigt werden. Spalte vier listet die absolute Differenz der Analysezeiten zwischen den ansonsten identischen Programmen gemäß Variante 1 und 2 auf. Spalte fünf berechnet prozentual das Verhältnis der Zeiten aus den Spalten drei und vier. Abbildung 11.1 ist eine graphische Veranschaulichung des unterschiedlichen Analysezeitenanstiegs bei Varianten 1 und 2.

Als Ergebnis dieser Messung läßt sich erkennen, daß das Anwachsen der Analysezeiten abhängig von der Problemgröße für die Varianten 1 und 2 stark unterschiedliches Verhalten zeigt. Während die Analysezeiten für Variante 1 sich mit jedem Erhöhen des Parameters $n = m$ in etwa verzehnfachen, zeigen die Analysezeiten für Variante 2 ein anscheinend mit dem Parameter $m = n$ lineares Anwachsen. Aus den Messungen lassen sich die folgenden Schlüsse ziehen:

- Trotz der gegenüber herkömmlichen Analysemethoden komplizierteren Form von Analysefakten läßt sich eine Analyse von *independent attributes* mit dem Verfahren aus dieser Arbeit sehr effizient realisieren. Diejenigen Pfadbedingungen für Programmeigenschaften eines Eingabeprogrammes, die keine Abhängigkeiten von anderen Programmeigenschaften besitzen, lassen sich damit auch mit dem vorgestellten Verfahren effizient berechnen.

n–m	Variante 1	Variante 2	Diff. abs.	Verh. rel.
3–3	0.081 s	0.017 s	0.064 s	20.9 %
4–4	0.322 s	0.018 s	0.304 s	5.6 %
5–5	1.993 s	0.020 s	1.973 s	1.0 %
6–6	17.544 s	0.028 s	17.516 s	0.16 %
7–7	156.646 s	0.029 s	156.617 s	0.019 %
8–8	1535.560 s	0.035 s	1535.525 s	0.0023 %
9–9		0.036 s		
10–10		0.041 s		
11–11		0.044 s		
12–12		0.050 s		
13–13		0.051 s		
14–14		0.054 s		
15–15		0.061 s		
16–16		0.064 s		
17–17		0.068 s		
18–18		0.071 s		
19–19		0.077 s		
20–20		0.079 s		

Tabelle 11.2: Analysezeiten für Varianten 1 und 2 der Transformationen des 3Sat–Problems in Zeigeranalyseprobleme.

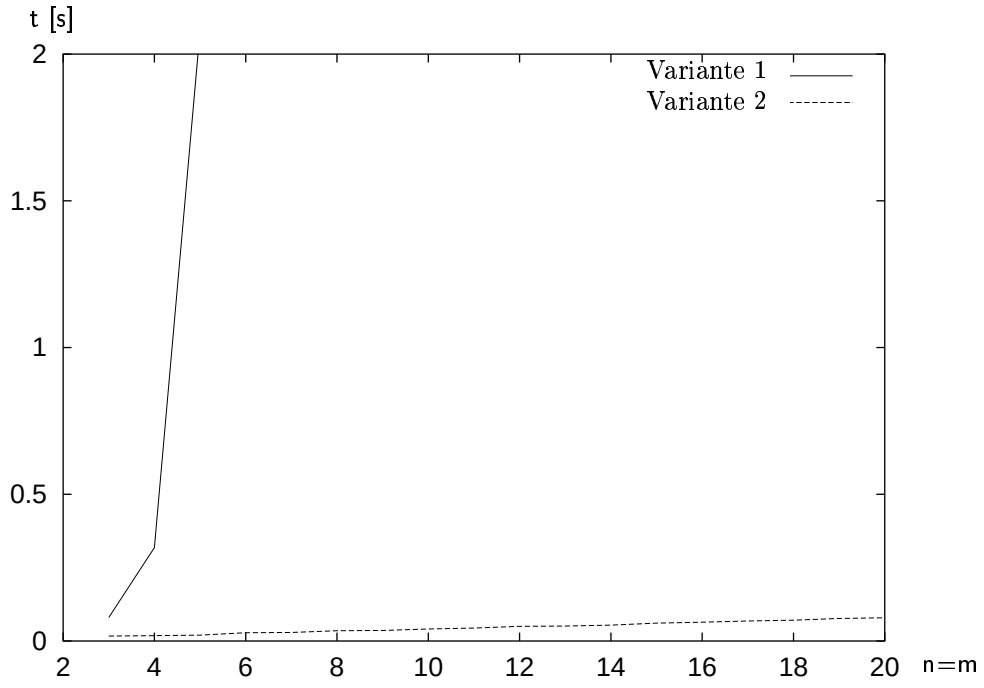


Abbildung 11.1: Laufzeitmessungen aus Tabelle 11.2 als Funktion der Problemgröße $n = m$.

- Die Schwierigkeit des Problems im worst–case zeigt sich in den mit der Problemgröße stark anwachsenden Analysezeiten für Variante 1. Der Parameter k scheint damit einen zentralen Einfluß auf die Frage der praktischen Realisierbarkeit von Analysen zu haben. Dieser wird im nachfolgenden Abschnitt näher untersucht werden.
- Nicht–ökonomische Verfahren müssen in jedem Fall zu Variante 1 vergleichbaren Analyseaufwand investieren, unabhängig davon, ob dies auch tatsächlich notwendig ist, oder ob wie bei Variante 2 die Menge Δ bereits mit deutlich weniger Aufwand zu berechnen ist. Die Differenz zwischen den Analysezeiten für die Varianten 1 und 2 kann man daher auch als Maß für die Verringerung des Berechnungsaufwandes durch das ökonomische Prinzip im Idealfall ansehen.

11.3.2 Abhängigkeit von Kernparametern

Die zweite Reihe von Laufzeitmessungen hat zum Ziel, in Abhängigkeit vom zentralen Parameter k aus Satz 9.8 die Anzahl der Zustände der berechneten Protokollautomaten, und die für die Berechnung dieser Menge von Zuständen benötigte Analysezeit zu bestimmen. Dadurch kann man Aussagen darüber treffen, welchen Grad an Abhängigkeiten von Programmeigenschaften untereinander ein Eingabeprogramm besitzen darf, um noch mit praktisch vertretbaren Berechnungszeiten einer exakten Analyse unterzogen werden zu können. Als weitere Aussagen lassen sich diese Zustandsmengengrößen mit denjenigen vergleichen, die sich aus der optimistischen Kostenschätzung in Satz 9.8 als Obergrenze ergeben. Daraus läßt sich belegen, daß diese Obergrenzen tatsächlich eingehalten werden. Ebenso ergibt sich für die verwendeten konstruierten Eingabeprogramme eine Aussage darüber, in welchem Größenverhältnis die bei deren Analyse auftretenden Zustandsmengen zu diesen Obergrenzen stehen.

11.3.2.1 Betrachtete Parameter

Im Gegensatz zum Vorgehen aus dem vorhergehenden Abschnitt werden beide Parameter m und n unabhängig voneinander variiert. Dadurch kann man zum einen die Art der Abhängigkeit des Analyseaufwandes von beiden Parametern n und m untersuchen. Zum andern kann man dadurch aus mehreren zufällig generierten Instanzen des 3Sat–Problems Mittelwerte für die gemessenen Größen bilden, die damit repräsentativer für die Menge der Eingabeprogramme mit entsprechenden Charakteristika sind, als einzelne zufällig generierte Eingabeprogramme.

Wie bereits in Abschnitt 9.2.5.1 vorgestellt wurde, entspricht die Anzahl m der Klauseln der Instanz des 3Sat–Problems der Anzahl k von gleichzeitig betriebenen Automaten des Bedingungsautomaten in Satz 9.8. Dadurch läßt sich aus den Messungen direkt die Abhängigkeit des Analyseaufwandes von diesem Parameter $k = m$ untersuchen.

11.3.2.2 Vorgehensweise

Analog zum Vorgehen im letzten Abschnitt werden aus 3Sat–Probleminstanzen generierte Eingabeprogramme vom Implementierungsprototypen analysiert. Dabei wird neben der benötigten Analysezeit auch die Größe der Zustandsmenge des einzigen auftretenden Protokollautomaten, der einen Bedingungsautomaten darstellt, gemessen.

11.3.2.3 Ergebnisse

Die Ergebnisse dieser Messungen sind in Tabelle 11.3 dargestellt. Horizontal ist die Anzahl m der Klauseln für Werte von 1 bis 8 aufgetragen. In vertikaler Richtung ist die Anzahl n von Variablen angeordnet. Diese Anzahl variiert von 3 bis 10. In der Tabelle sind jeweils die Zustandsmengengröße des für die exakte Lösung des 3Sat–Problems berechneten Protokoll– bzw. Bedingungsautomaten $|Q_C|$, sowie die zu dieser Berechnung benötigte Zeit eingetragen. In der vorletzten Zeile der Tabelle wurden spaltenweise Mittelwerte über diese Größen gebildet. Die letzte Zeile gibt zum Vergleich mit den gemessenen Zustandsmengengrößen die optimistische Obergrenze aus Satz 9.8 an. Der Parameter $|Q|$ ergibt sich dabei aus einer Betrachtung der berechneten Protokollautomaten. Der Parameter

		m=1	2	3	4	5	6	7	8
n=3	$\lvert Q_C\rvert$	26	103	326	823	2379	6790	20962	62764
	t	0.013 s	0.028 s	0.081 s	0.329 s	2.096 s	16.498 s	152.385 s	1463.77 s
4	$\lvert Q_C\rvert$	26	92	276	786	2449	7798	23240	69180
	t	0.013 s	0.027 s	0.077 s	0.322 s	1.974 s	16.231 s	141.417 s	1329.56 s
5	$\lvert Q_C\rvert$	26	83	248	742	2285	7234	21404	63105
	t	0.013 s	0.026 s	0.073 s	0.310 s	1.993 s	16.332 s	145.687 s	1373.94 s
6	$\lvert Q_C\rvert$	26	99	300	860	2570	8039	25550	75393
	t	0.013 s	0.029 s	0.080 s	0.343 s	2.083 s	17.544 s	163.526 s	1533.07 s
7	$\lvert Q_C\rvert$	26	85	261	794	2414	7647	22731	66890
	t	0.013 s	0.026 s	0.076 s	0.325 s	2.094 s	17.182 s	156.646 s	1463.48 s
8	$\lvert Q_C\rvert$	26	84	268	836	2458	8060	25536	71414
	t	0.013 s	0.026 s	0.079 s	0.346 s	2.139 s	17.815 s	164.616 s	1535.56 s
9	$\lvert Q_C\rvert$	26	85	254	788	2427	7527	23601	69322
	t	0.013 s	0.026 s	0.075 s	0.325 s	2.128 s	17.270 s	163.648 s	1512.27 s
10	$\lvert Q_C\rvert$	26	99	303	882	2737	8440	25249	76699
	t	0.013 s	0.029 s	0.080 s	0.344 s	2.292 s	18.489 s	169.993 s	1580.51 s
Durchschn.	$\lvert Q_C\rvert$	26.0	91.3	279.5	813.9	2464.9	7691.9	23534.1	69345.9
	t	0.013 s	0.027 s	0.078 s	0.331 s	2.100 s	17.170 s	157.240 s	1474.020 s
$\lvert Q\rvert \cdot (r+1)^k$		$12 \cdot 3^1$	$24 \cdot 3^2$	$36 \cdot 3^3$	$48 \cdot 3^4$	$60 \cdot 3^5$	$72 \cdot 3^6$	$84 \cdot 3^7$	$96 \cdot 3^8$
		$= 36$	$= 216$	$= 972$	$= 3888$	$= 14580$	$= 52488$	$= 183708$	$= 629856$

Tabelle 11.3: Messung von Zustandsmengengrößen und Analysezeiten abhängig von den Parametern n und $m = k$.

r besitzt den Wert 2, da jede Zeigervariable im Programm genau zwei Ziele besitzen kann. Der Parameter k ist aufgrund von obiger Argumentation gleich dem Parameter m. Anhand der Ergebnisse lassen sich die folgenden Aussagen treffen.

- Mit zunehmender Anzahl m von Klauseln, und damit der Anzahl k von maximal gleichzeitig betriebenen Automaten, steigt der Analyseaufwand deutlich an. Von der Anzahl n der Variablen des 3Sat–Problems ist der Aufwand dagegen anscheinend unabhängig. Damit spielt es keine Rolle, aus wieviel möglichen Automaten die gleichzeitig betriebenen Automaten ausgewählt werden. Der zentrale Parameter für den Analyseaufwand ist nach diesen Ergebnissen die Anzahl k von gleichzeitig betriebenen Automaten.

  Dies bestätigt die bisherigen Vermutungen auf der Basis der Annahme der Gutartigkeitseigenschaften.

- Die optimistische Obergrenze für die Zustandsmengengrößen aus Satz 9.8 wird von den berechneten Protokollautomaten eingehalten. Konkret liegen die Zustandsmengengrößen der berechneten Protokollautomaten sogar weit darunter. Dies läßt sich darauf zurückführen, daß nicht alle möglichen Kombinationen von korrespondierenden Zuständen und Ergebnismengen der betriebenen Automaten auch tatsächlich bei der Analyse vorkommen. Im untersuchten Beispiel z.B. führen die Queries bei deren Verarbeitung in den untersten, die letzte Klausel darstellenden Programmblöcken nur einen betriebenen Automaten mit, im darüberliegenden Programmblock zwei, etc. Damit der worst–case aus der optimistischen Abschätzung eintreten könnte, müssten hingegen bereits im untersten Programmblock alle möglichen betriebenen Automaten mitgeführt werden können, um die in der Abschätzung angegebene Vervielfachung der Zustandsmenge Q auch tatsächlich zu erreichen.

  Damit bestätigen diese Ergebnisse auch experimentell die Gültigkeit der optimistischen Kostenschätzung aus Satz 9.8.

- Die gemessenen Zustandsmengengrößen und Analysezeiten geben einen Anhaltspunkt, welche Eingabeprogramme mit vertretbarem praktischen Aufwand exakt analysiert werden können. Dabei erweist sich der Parameter k, der die maximale Anzahl von gleichzeitig betriebenen Automaten angibt, als Maßgröße für die Schwierigkeit der Analyse von Eingabeprogrammen. Auf dies soll in Abschnitt 11.5 genauer eingegangen werden.

11.4 Abschätzung der Analysezeiten für die untersuchten realen Programme

In Tabelle 11.4 ist eine Erweiterung von Tabelle 11.1 angegeben, in der zusätzlich die geschätzten Analysezeiten t angegeben sind. Diese lassen sich anhand der Ergebnisse aus Tabelle 11.3 ermitteln. Für das Eingabeprogramm diff z.B. sind insgesamt für 7 Dereferenzierungsoperationen Queries zu berechnen. Jede dieser Queries besitzt eine Zustandsmenge von deutlich unter 10 Zuständen. Damit läßt sich der Zeitbedarf für jede dieser Queries wie in Tabelle 11.3 für $k = m = 1$ angegeben abschätzen, da die dort berechnete Zustandsmenge ungefähr diese Größe besitzt, und dort auch der gleiche Parameter k auftritt. Daher wird jede dieser 7 Queries voraussichtlich weniger als 0.013 Sekunden benötigen, was eine gesamte Analysezeit von unter 0.1 Sekunden zur Folge hat. Analog geht man für die anderen Programme auf der Basis der dort angegebenen Anzahl von Dereferenzierungsoperationen und Zustandsmengengrößen vor. Für das Eingabeprogramm cdecl wird zur Analysezeitabschätzung auf der Basis der Zustandsmengengröße 480, was ohnehin bereits eine geschätzte obere Grenze darstellt, der Zeitbedarf für den Fall $k = 4$ in Tabelle 11.3 verwendet. Dort wurden für das konstruierte Eingabeprogramm beinahe doppelt soviele Zustände berechnet, als dies für das Programm cdecl notwendig ist. Damit stellt der sich daraus rechnerisch für die gesamte Analyse ergebende Zeitaufwand von ca. 1.75 Sekunden ebenfalls eine Obergrenze dar.

Programm	LOC	$\Omega^{(1)}_{\text{deref}}$	$\Omega^{(0)}_{\text{deref}}$	$\overline{\lvert Q_i^{(1)}\rvert}$	$\overline{\lvert Q_i^{(0)}\rvert}$	k	t ca.
compress compress.c	1497	30	4	20.4	22.0	1	$< 0.5s$
ansitape ansitape.c	1744	2	8	3.0	10.8	1	$< 0.2s$
unix/diff diffdir.c	1814	5	2	3.0	7.0	1	$< 0.1s$
cdecl cdlex.c	3879	7	5	≤ 20	≤ 480	4	$< 1.75s$
agrep parse.c follow.c	3968	21	15	7.8	9.9	1	$< 0.5s$

Tabelle 11.4: Ergebnis der Messungen an realen Eingabeprogrammen mit geschätzten Analysezeiten.

11.5 Diskussion der Ergebnisse

Die Ergebnisse aus diesem Kapitel sollen in zweierlei Hinblick diskutiert werden. Zum einen kann man aus den Ergebnissen eine Analysestrategie entwickeln, wie man mit verschieden schwierigen Eingabeprogrammen umgehen kann. Zum anderen kann man die realen Messungen verwenden, um die praktische Anwendbarkeit des Verfahrens im Allgemeinen zu untersuchen, und diese Aussagen soweit wie möglich auf Eingabeprogramme beliebiger Größe zu übertragen.

11.5.1 Vollständige und interaktive, exakte und approximative Analyse

Kategorisiert man zu analysierende Eingabeprogramme nach dem zu erwartenden Parameter k, so kann man eine Einordnung angeben, welche dieser Eingabeprogramme sich für eine exakte, aber dennoch praktisch realisierbare Analyse eignen. Für Parameter $k \in \{1, \ldots, 5\}$ kann man mit Analysezeiten bis zu ca. 2 Sekunden pro Dereferenzierungsoperation, für die Abhängigkeiten von anderen Dereferenzierungsoperationen bestehen, rechnen. Da die Anzahl von solchen Dereferenzierungsoperationen erfahrungsgemäß, und durch die Ergebnisse in Tabelle 11.1 belegt, relativ klein ist, muß man in der Summe mit keinen für die praktische Verwendung unzumutbaren Analysezeiten rechnen. Daher kann man solche Eingabeprogramme mit dem vorgestellten Verfahren noch einer vollständigen exakten Analyse für alle Dereferenzierungsoperationen unterziehen. In diese Kategorie fallen sämtliche realen Eingabeprogramme, die in Abschnitt 11.2 untersucht wurden.

Für Parameter $k \in \{6, \ldots, 8\}$ kann man interaktiv ausgewählte Anfragen noch mit praktisch vertretbarem Aufwand exakt lösen, wenn der Benutzer für die gebotene Genauigkeit bereit ist, einen Analyseaufwand zwischen 17 Sekunden für $k = 6$ und ca. 20 Minuten für $k = 8$ zu investieren.

Darüberhinaus kann man durch Beschränken der Anzahl der gleichzeitig betriebenen Automaten auf eine vorgegebene Zahl $k_{\max}$ eine sichere approximative Lösung mit wählbarer Genauigkeit berechnen. Dies wird in Satz 5.10 beschrieben. Damit ließe sich auch im Fall von Eingabeprogrammen, die eine maximale Anzahl von mehr als 8 gleichzeitig betriebenen Automaten für eine exakte Lösung benötigen, ein sinnvoller Kompromiss zwischen Genauigkeit und Analysezeit finden. Zumindest die Berücksichtigung der $k_{\max}$ betriebenen Automaten ist exakt, d.h. die betrachteten $k_{\max}$ Programmeigenschaften werden garantiert in der Menge der beschriebenen Programminstanzen gelten. Da sich $k_{\max}$ auf die Anzahl der gleichzeitig betriebenen Automaten bezieht, wird die Anzahl der Programmeigenschaften, die insgesamt garantiert gelten, im Allgemeinen sogar noch größer als $k_{\max}$ sein. Damit kann man auch im Fall von approximativen Berechnungen eine Verbesserung der Genauigkeit gegenüber traditionellen Verfahren erzielen, da bei diesen keinerlei allgemeine Aussagen über die in jedem Fall zu erzielende Genauigkeit gemacht werden können. Alle untersuchten realen Programme liegen jedoch noch weit diesseits der Grenzen der praktisch realisierbaren exakten Analyse. Daher ist

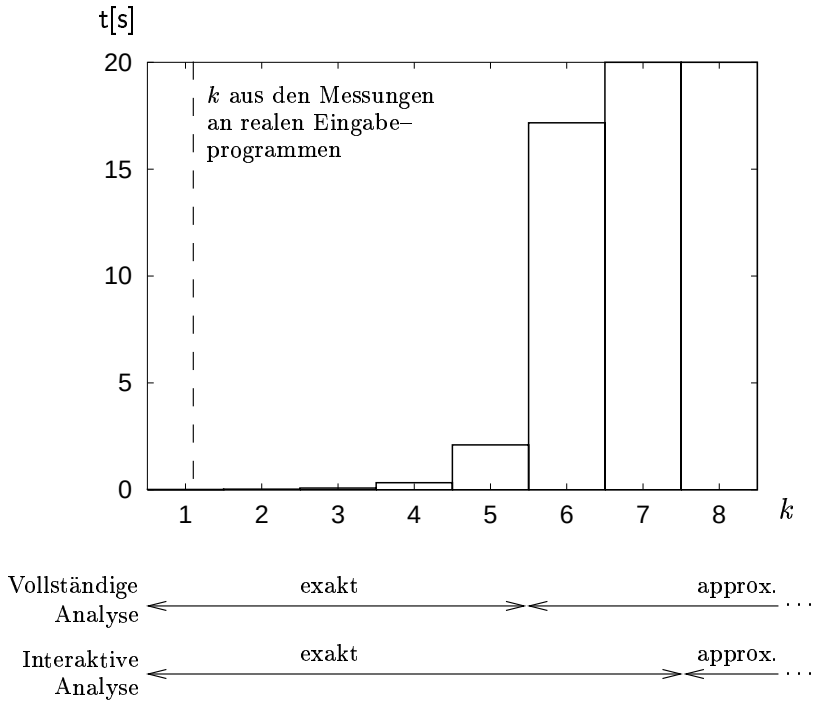


Abbildung 11.2: Analysezeiten abhängig vom Parameter k und darauf basierende Einordnung in vollständige und interaktive Analyse mit exakter bzw. approximativer Zielsetzung.

die Notwendigkeit der Verwendung von approximativen Analysetechniken auf der Basis von Bedingungsautomaten erst bei der Kombination von verschiedenen Analysefragestellungen, wie z.B. von Zeigeranalyse, Konstantenanalyse mit approximativer Berücksichtigung von Verzweigungsbedingungen (Conditional Constants), und Typanalyse zu erwarten (auf diese möglichen Erweiterungen wird im Ausblick kurz eingegangen werden).

Die Einordnung der verschiedenen möglichen Zielsetzungen der Analyse in Abhängigkeit vom Parameter k ist in Abbildung 11.2 veranschaulicht. Dabei ist im Diagramm in horizontaler Richtung der Parameter k, und in vertikaler Richtung die durchschnittliche Analysezeit in Sekunden für die konstruierten Eingabeprogramme aus Tabelle 11.3 angetragen. Darunter ist die im Text vorgestellte mögliche Einordnung in vollständige und interaktive Analyse eingezeichnet. Die gestrichelte Linie im Diagramm stellt den Durchschnitt über den Parameter k von 1.6 für alle realen Eingabeprogramme aus Tabelle 11.1 dar.

11.5.2 Praktische Realisierbarkeit

Ungeachtet des theoretischen worst–case stellen insgesamt betrachtet die realen Programme das Analyseverfahren offensichtlich vor keine unlösbaren Probleme. Eine maximale Analysezeit von unter zwei Sekunden für das exakte Ergebnis einer Analyse von *relational attributes* steht in einem extremen Gegensatz zum befürchteten worst–case. Die Eingabeprogramme sind von ihrer Größe nicht vergleichbar mit Industrieprojekten mit mehreren Millionen Zeilen Programmcode. Andererseits haben sie den Rahmen der pathologisch kleinen Eingabeprogramme sicherlich verlassen. Damit besitzen die vorgeführten Ergebnisse durchaus bereits Relevanz im Hinblick auf eine praktische Anwendbarkeit des Verfahrens.

Die Tatsache, daß die Gültigkeit der Gutartigkeitsannahmen für die betrachteten Eingabeprogramme nachgewiesen werden konnte, ist für einen Ausblick auf große Eingabeprogramme vielversprechend. Im Gegensatz zum theoretischen worst–case Analyseaufwand, der mit steigender Einga-

beprogrammgröße exponentiell anwächst, kann man die Gutartigkeitsannahmen als Aussagen über die allgemeine Struktur von Programmen ansehen, die unabhängig von der Programmgröße gelten. Damit würde bestenfalls bei großen Eingabeprogrammen primär die Anzahl der Dereferenzierungsoperationen ansteigen. Die einzelnen Zustandsmengengrößen der Automaten, und der Grad der Abhängigkeit von Programmeigenschaften untereinander, der durch den Parameter k in Satz 9.8 repräsentiert wird, würden jedoch mit den Größen aus den untersuchten Programmen vergleichbar bleiben. In diesem Fall wäre die vorgestellte Analysemethode auch sehr gut skalierbar für größere Eingabeprogramme.

Ein abschließende Aussage darüber, ob das Verfahren garantiert in jedem Fall praxistauglich ist, kann aber natürlich erst getroffen werden, wenn die theoretische Grundlage des Verfahrens um weitere Analysearten und approximative Berechnungsmethoden für die nicht exakt lösbaren Fragestellungen erweitert wurde. Erst dann kann ein Prototyp eines umfangreicheren Analyseverfahrens implementiert werden, der sämtliche Facetten der Analyse eines realen Eingabeprogrammes berücksichtigt, und der als Grundlage für weitere Messungen des Analyseaufwandes anhand von größeren Eingabeprogrammen dienen kann.

Insgesamt erscheint das ökonomische Berechnungsprinzip sehr vielversprechend. Die untersuchten realen Programme weisen keine der "bösartigen" Konstrukte aus den konstruierten worst–case Eingabeprogrammen aus der Literatur auf. Stattdessen erweist sich die Problemmodellierung aus dieser Arbeit als eine sehr praktikable Sichtweise auf ein Problem, das in der Literatur als sehr schwierig bezeichnet wird, und dessen exakte Lösung anhand dessen theoretischer worst–case Komplexität von verschiedenen Arbeiten als unrealisierbar eingestuft wird.

11.6 Zusammenfassung

Für den Nachweis der praktischen Realisierbarkeit des Verfahrens wurden sowohl reale Eingabeprogramme einer Analyse im Sinne des vorgestellten Analyseverfahrens "von Hand" unterzogen, als auch automatisch generierte Eingabeprogramme aus der zugelassenen Programmklasse mit dem Implementierungsprototypen analysiert. Die realen Eingabeprogramme entstammen einer repräsentativen Sammlung von Eingabeprogrammen, die bereits mehrmals als Testprogramme für Zeigeranalyseverfahren verwendet wurden. Bei beiden Experimenten wurden charakteristische Parameter wie Zustandsmengengrößen der Protokollautomaten und der Grad der Abhängigkeit von Programmeigenschaften untereinander ermittelt. Für die konstruierten Eingabeprogramme konnten zusätzlich Laufzeitmessungen der Analyse vorgenommen werden.

Eines der Ergebnisse der Untersuchungen war die Aussage, daß die realen Eingabeprogramme die im letzten Abschnitt vorgestellten Gutartigkeitseigenschaften erfüllen. Daraus ergibt sich eine Reduzierung der maximalen Analysekosten gemäß der optimistischen Kostenschätzung aus Kapitel 9. Darüberhinaus kann man die worst–case Eingabeprogramme aus der Literatur, die diese Eigenschaften nur teilweise bzw. gar nicht erfüllen, als unrealistisch für zu erwartende praktische Analyseaufgaben einordnen. Beide Faktoren lassen eine praktische Anwendbarkeit des Verfahrens erwarten.

Eine Untersuchung von Extremfällen für die Menge Δ von Kombinationen von Analysefakten zeigt exponentiellen Anstieg der Analysezeiten für maximales $\Delta = \Theta$ und linearen Anstieg für minimales Δ. Nicht–ökonomische Verfahren müssen in jedem Fall stets die maximale Menge Θ berechnen, weshalb diese mit hoher Wahrscheinlichkeit in jedem Fall einen exponentiell mit der Eingabeprogrammgröße anwachsenden Berechnungsaufwand investieren müssen. Im Idealfall kann durch das ökonomische Prinzip daher der zu investierende Berechnungsaufwand von exponentiellem Wachstum mit der Eingabeprogrammgröße auf lineares Wachstum reduziert werden.

Inwiefern sich diese Verbesserung im Idealfall in Analysezeiten für reale Eingabeprogramme widerspiegelt, wurde anschließend untersucht. Für die realen Eingabeprogramme, die mit mehreren Tausend Zeilen Programmcode durchaus praktische Relevanz besitzen, wurde dazu anhand der daraus bestimmten charakteristischen Parameter auf der Basis von Laufzeitmessungen der Analyse von konstruierten Eingabeprogrammen mit gleichen Parametern eine Abschätzung der zu erwartenden Analysezeiten angegeben. Diese liegen größtenteils weit unter einer Sekunde und insgesamt alle un-

ter zwei Sekunden für eine Analyse aller für eine Berechnung von *relational attributes* relevanten Protokollautomaten. Damit steht dieses Ergebnis in einem extremen Gegensatz zu dem aufgrund des theoretischen worst–case bisher zu befürchtenden Berechnungsaufwand. Das exakte Analyseverfahren scheint für die betrachtete Programmklasse direkt praktisch einsetzbar zu sein.

Interpretiert man die Gutartigkeitseigenschaften als Struktureigenschaften von Programmen, die weitgehend unabhängig von der Programmgröße sind, so kann man die praktische Anwendbarkeit auch für größere Eingabeprogramme folgern. Für den Fall daß der zu erwartende Analyseaufwand dagegen ein praktisch vertretbares Maß übersteigt, was nach den Ergebnissen allerdings erst für noch komplexere Fragestellungen wie im Anhang beschrieben zu erwarten ist, kann man eine approximative Abwandlung des Verfahrens einsetzen. Dieses erlaubt eine a priori Festlegung der Analysegenauigkeit, die sich in der gleichzeitigen Gültigkeit einer bestimmten minimalen Anzahl von Programmeigenschaften in der berechneten Menge von induzierten Programminstanzen äussert. Je nach Einsatzgebiet der Analyse, also vollständiger oder interaktiver Analyse, kann man verschiedene Parameter für eine maximale Analysegenauigkeit bei jeweils vertretbarem Rechenaufwand wählen. Die aus den realen Eingabeprogrammen gewonnenen charakteristischen Größen liegen jedoch noch weit diesseits der Grenze der praktischen Anwendbarkeit von exakter vollständiger Analyse der Eingabeprogramme.

Kapitel 12

Ergebnisse

In dieser Arbeit wurde ein neues statisches Analyseprinzip zur Berechnung von *relational attributes* entwickelt, das auf dem Prinzip der Assoziation von Analysefakten mit potentiell unendlichen Mengen von Programminstanzen basiert. Diese Mengen von Programminstanzen werden durch die von endlichen Automaten akzeptierten Sprachen dargestellt. Durch Klassenbildung von Mengen von Programminstanzen können diese Programminstanzenmengen durch handhabbare endliche Automaten beschrieben werden, die nur für relevante Programmeigenschaften oder Verzweigungen im Eingabeprogramm Zustände bzw. Zustandsübergänge enthalten. Vom vorgestellten Analyseverfahren werden nur Automaten verwendet, die solche Klassen von Programminstanzenmengen darstellen. Damit wurde eine Möglichkeit geschaffen, die an sich schwer handhabbaren Mengen von Programminstanzen in einem praktischen Verfahren zu verarbeiten. Kapitel 5 liefert die theoretische Grundlage für dieses neuartige Analyseprinzip.

Eine Darstellung von Eingabeprogrammen durch eine neue Programmrepräsentation, die formal eingeführt wurde, und deren Korrektheit bewiesen wurde, liefert die Basis für die praktische Realisierung des theoretischen Analyseprinzips. Dabei werden Queries durch Analysegraphen gesendet, die eine der möglichen Darstellungsformen der neuen Programmrepräsentation sind. Durch diese Analysegraphen ist es möglich, die Theorie der Schnittbildung von Mengen von Programminstanzen auf Automatenbasis mit der semantischen Betrachtung von Programmeigenschaften zu verknüpfen. Die von Queries berechneten Protokollautomaten stellen Pfadbedingungen für betrachtete Programmeigenschaften dar. Damit leisten die Queries auf der Grundlage der Analysegraphen eine explizite Berechnung einer zusammenfassenden Beschreibung von potentiell unendlichen Mengen von Programminstanzen mit gleichen Eigenschaften, die so bisher in der Literatur nicht erreicht wurde.

Vom vorgestellten Verfahren wurde gezeigt, daß es maximal ökonomisch ist. Von den bisher bekannten realisierten oder denkbaren Verfahren für Zeigeranalyse aus der Literatur besitzt keines diese Eigenschaft. Nach der Argumentation aus Kapitel 2 ist Ökonomie aber eine Schlüsseleigenschaft für die praktische Realisierbarkeit von Analysen von *relational attributes*.

Eine Untersuchung der maximalen Analysekosten hat eine Menge von Gutartigkeitseigenschaften von Eingabeprogrammen aufgezeigt, die in den experimentellen Untersuchungen von allen Eingabeprogrammen angenommen werden. Auf der Basis dieser Gutartigkeitseigenschaften ließ sich eine optimistische Kostenschätzung angeben, die praktische Einsetzbarkeit des Verfahrens verspricht. Darüberhinaus wurden die worst–case Eingabeprogramme aus der Literatur auf die Gutartigkeitseigenschaften untersucht. Diese besitzen die Gutartigkeitseigenschaften nur teilweise bzw. gar nicht. Damit wurde es ermöglicht, die worst–case–Ergebnisse aus der Literatur anhand von theoretischen Überlegungen als für die praktische Anwendung unrelevant einzuordnen.

Eine auf realen Laufzeitmessungen beruhende Abschätzung der für die realen Eingabeprogramme zu erwartenden Analysezeiten steht im extremen Gegensatz zu den Erwartungen, die bisher aufgrund des theoretischen worst–case in der Literatur vermittelt wurden. Das Verfahren scheint für die zugelassene Programmklasse direkt praktisch einsetzbar zu sein. Damit leistet das ökonomische Berechnungsprinzip nicht nur eine Reduktion der theoretischen Obergrenzen für Analysekosten, sondern ermöglicht auch eine experimentell nachweisbare praktische Einsetzbarkeit des Verfahrens.

Damit wurde das erste praktisch einsetzbare Analyseverfahren von *relational attributes* für Multi–Level–Zeigeranalyse erreicht.

Für weniger gutartige Eingabeprogramme oder schwierigere Analysefragestellungen, wie sie im Ausblick vorgestellt werden, wurde eine approximative Berechnungsmethode der Analyse vorgestellt. Dazu wurde eine Definition einer sicheren Approximation einer Analyse von *relational attributes* gegeben, die auch für die traditionellen Analysemethoden passend ist. Die approximative Variante des Verfahrens erlaubt aber im Gegensatz zu allen bekannten Verfahren aus der Literatur eine frei wählbare Mindestgenauigkeit, die garantiert erreicht wird. Damit stellt das Verfahren auch in dem Fall, daß eine exakte Analyse nicht mehr mit vertretbarem Aufwand realisierbar ist, eine Verbesserung gegenüber den Verfahren aus der Literatur dar. Die untersuchten Eingabeprogramme lagen jedoch alle noch weit diesseits der Grenzen der praktischen Realisierbarkeit einer vollständigen exakten Analyse.

Damit wurde mit einer exakten, aber trotzdem praktisch anwendbaren Multi–Level–Zeigeranalyse die Grundlage für Programmanalysen mit maximaler Genauigkeit in Bezug auf eine Analyse von *relational attributes* geschaffen. Mit den im Ausblick vorgestellten Erweiterungen des Analyseverfahrens auf beliebige C++– oder Java–Programme ist damit das Ziel der Verwendung von solchen Programmen als Spezifikation für Hardware/Software Codesign deutlich näher gerückt.

Eine mögliche andere Verwendung der vorgestellten Analysetechnik liegt in interaktiven Programmierunterstützungssystemen. Dabei könnte die gewonnene Genauigkeit dazu dienen, dem Benutzer nur Analyseergebnisse zu präsentieren, die tatsächlich in konkreten Programminstanzen auftreten können. Sozusagen als Nebeneffekt der Analyse könnten die berechneten Mengen von Programminstanzen, in denen eine vom Benutzer betrachtete Programmeigenschaft auftritt, als zusätzliche Information dargestellt werden.

Da Zeigeranalyse für alle Analysefragestellungen eine wichtige Voraussetzung ist, könnten auch optimierende Compiler von der vorgestellten Analysetechnik profitieren. Je genauer das Ergebnis der Zeigeranalyse ist, desto weniger ungenaue Ergebnisse der darauf aufbauenden weiteren Analysen werden berechnet.

Kapitel 13

Ausblick

Im Ausblick sollen mögliche Strategien für eine Erweiterung des Verfahrens auf interprozedurale Analyse, sowie der Behandlung von strukturierten Datentypen vorgestellt werden. Des weiteren wird die Verwendung von Bedingungsautomaten für andere Analysefragestellungen diskutiert werden.

13.1 Interprozedurale Analyse

Der Problemschwerpunkt von vielen anderen, in der Literatur vorgestellten Analysetechniken liegt in der Vermeidung der Propagation von Analyseinformationen in einer Weise, die widersprüchlich zum Prinzip des Rücksprungs bei Funktionsaufrufen nur genau zur Aufrufstelle ist (sog. *unrealizable path problem*).

Offensichtlich ist die Beschreibung von solchen Pfaden nicht mehr durch reguläre Sprachen möglich. Die komplexere Sprachklasse der kontextfreien Sprachen (z.B. [HU79, Kapitel 4]) ermöglicht es, interprozedural korrekte Pfade als Worte einer solchen Sprache darzustellen (z.B. [Kno98]).

Das Gegenstück zu endlichen Automaten für reguläre Sprachen stellen Kellerautomaten für kontextfreie Sprachen dar. Diese besitzen zusätzlich zu den Bestandteilen der in dieser Arbeit vorgestellten Automaten einen Keller bzw. Stack, dem durch Zustandsübergänge des Automaten Symbole hinzugefügt, bzw. davon entfernt werden können. Durch diese Art von Zustandsübergängen können bestimmte Zustandsübergänge des Automaten z.B. nur dann zugelassen werden, wenn ein bestimmtes Symbol oben auf dem Stack liegt.

Eine mögliche Erweiterung des Analyseverfahrens aus dieser Arbeit würde solche Stacks als Bestandteil jeder Query verwenden. Vom Prinzip wäre eine solche Erweiterung damit ähnlich den Arbeiten aus der Literatur, die *CFL–reachability* [MR00] (context–free–language reachability) verwenden, um anforderungsgetriebene Analysen von *independent attributes* [RHS95] [HRS95] [DGS97] bzw. Slicing [HRB88] [RHSR94] zu realisieren.

Im Folgenden soll eine entsprechende Erweiterung der Queryverarbeitung skizziert und auf deren Probleme, sowie mögliche Lösungen eingegangen werden.

13.1.1 Erweiterung der Analysegraphen um Funktionsaufrufe

Zur Darstellung von Funktionsaufrufen durch die neue Programmrepräsentation sind zusätzliche Operationen bzw. Graphknoten erforderlich. Diese werden als call–, caller–, sowie return–Operationen bezeichnet und in Abbildung 13.1 in ihrer Graphform vorgestellt, sowie ihre Verwendung anhand eines Beispiels erläutert.

Der Graph auf der linken Seite zeigt von oben nach unten eine var–Operation für die Variable x (mit zusätzlichen Informationen über die Art der Variable), sowie zwei call–Operationen ω_1 und ω_2, die zwei Aufrufe der gleichen Funktion, repräsentiert durch den Block β_2, darstellen. Dabei werden Parameter und Rückgabewert als benannte Werte modelliert, während die Use–Use–Chains von Variablen, die in der aufgerufenen Funktion (möglicherweise) referenziert oder modifiziert werden,

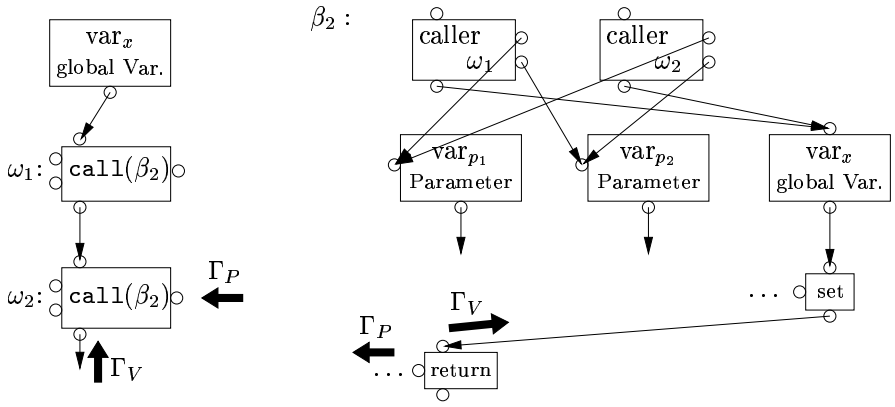


Abbildung 13.1: Graphknoten zur Repräsentation von Funktionsaufrufen

hier am Beispiel der globalen Variable x, eine Verwendung am Aufrufknoten besitzen. Analog zu der in dieser Arbeit verfolgten Vorgehensweise ist dabei eine Funktion eine mögliche Verwendung für jede Variable, die im Rumpf dieser oder einer von dieser aufgerufenen weiteren Funktion verwendet wird, bzw. die dort Ziel der Dereferenzierung einer Zeigervariable passenden Typs sein kann.

Der Funktionsrumpf β_2 ist auf der rechten Seite dargestellt, darin befinden sich in vertikaler Reihenfolge zuerst caller–Operationen als Repräsentanten der verschiedenen Aufrufstellen der Funktion. Diese produzieren einen benannten Wert für jeden Parameter "by value" und werden in die lokalen Use–Use–Chains derjenigen Variablen integriert, für die die Funktion eine Verwendung darstellt. Dies wäre im Beispiel die globale Variable x. Darunter sind Var–Operationen angeordnet. Zuletzt findet sich im Beispiel eine return–Operation.

13.1.2 Erweiterung der Queryverarbeitung um Aufruf–Stacks

Die Verarbeitungsvorschrift für die Queries läßt sich einfach erweitern, so daß auch diese zusätzlichen Operationen behandelt werden können. Das Erreichen einer call–Operation durch eine rückwärts gerichtete Query wird durch eine Fortsetzung der Queryverarbeitung an der bzw. den entsprechenden return–Operationen geleistet. In Abbildung 13.1 entsprechen die Queries Γ_V bzw. Γ_P im Block β_2 der Fortsetzung der entsprechenden Queries im linken Teil der Abbildung.

Dabei kann die Queryfunktionalität einfach dahingehend erweitert werden, nur interprozedural korrekte Pfade zu durchlaufen. Dazu wird in jede Query ein Aufrufstack integriert, der die Zulässigkeit von Rückkehr–Kanten, also von caller– zu call–Operationen, überprüfbar macht.

Als weitere Änderung gegenüber dem bisherigen Verfahren muß die Erkennung von Wiederholungen dahingehend erweitert werden, daß eine Wiederholung oder eine Alternative nur dann vorliegen kann, wenn der Aufrufstack den gleichen Inhalt besitzt wie der der vorher angetroffenen Query.

Die von Queries erzeugten Automaten beschreiben nach dieser Methode weiterhin reguläre Sprachen. Analog dazu, daß in der Vorgehensweise dieser Arbeit die Zustände der als Bedingungen mitgeführten Automaten als eine neue Zustandsmenge eines einzelnen Automaten interpretiert werden, der dann eine reguläre Beschreibung einer exakten Lösung von intraprozeduraler Multi–Level–Zeigeranalyse darstellt, müssen zur Darstellung eines exakten Ergebnisses interprozeduraler Analyse zusätzlich die Aufrufstacks der Queries mit in die neue Zustandsmenge übernommen werden. Im Fall von Programmen ohne Rekursionen ist die Größe des Aufrufstacks beschränkt und es existiert ein einzelner Automat, also eine reguläre Sprache, der bzw. die eine exakte Lösung interprozeduraler Analyse beschreibt. Man könnte die Darstellung der exakten Lösung durch eine reguläre Sprache, wofür eigentlich eine kontextfreie Sprache benötigt werden würde, als eine anforderungsgetriebene Variante der vollständigen Expansion des Aufrufgraphen mit anschließender "regulärer" Analyse interpretieren, wie dies z.B. von [EGH94] vorgeschlagen wird.

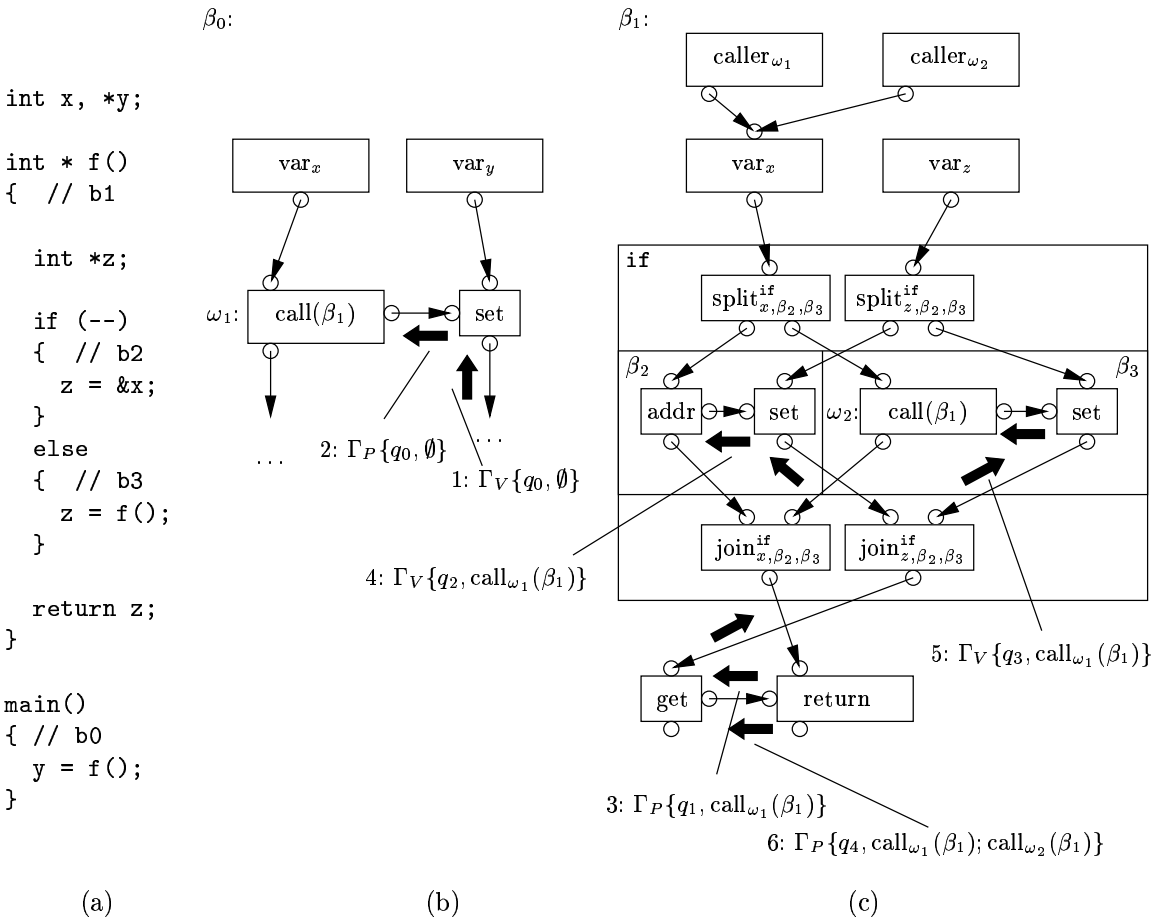


Abbildung 13.2: Beispiel für die Analyse von rekursiven Funktionen

13.1.3 Probleme

Obwohl das erwartete Ergebnis im Fall von Eingabeprogrammen ohne Rekursionen als reguläre Sprache interpretiert werden kann, müßte der Beweis der Korrektheit über kontextfreie Sprachen erfolgen, weshalb die in dieser Arbeit zentrale Vorgehensweise, den Schnitt von Sprachen endlicher Automaten als Grundlage für eine theoretische Fundierung zu verwenden, nicht mehr durchführbar wäre. Daher beschränkt sich diese Arbeit auf die Fälle, in denen die Korrektheit des Verfahrens auf der Basis regulärer Sprachen beweisbar ist.

Im Fall von Rekursionen ist der mitgeführte Aufrufstack darüberhinaus nicht größenbeschränkt. Eine exakte Lösung durch reguläre Sprachen ließe sich daher nur durch Automaten mit unendlicher Zustandsmenge beschreiben.

Abbildung 13.2 zeigt ein Beispiel für den Fall, in dem beliebig große Aufrufstacks aufgebaut werden können, ohne daß eine Wiederholung erkannt werden kann. Das einfache Beispielprogramm in Teilabbildung (a) soll dies verdeutlichen. Die Rekursion über die Funktion `f`, aufgerufen aus der Funktion `main` liefert als Rückgabewert schließlich die Adresse der Variable x. Bis dahin kann aber der rekursive Aufruf beliebig oft durchlaufen werden. In der Abbildung 13.2(b) findet sich der Analysegraph zur Funktion `main` in Form des Operationsblockes β_0. Darin wird der Variable y der Rückgabewert des Funktionsaufrufes von `f` zugewiesen. Die Funktion `f` greift dabei auf die globale Variable x zu, weswegen die Use–Use–Chain von x eine Verwendung an dieser call–Operation besitzt. Die in der Abbildung eingezeichneten Schritte der Queryverarbeitung sind dabei wie folgt:

1. Eine Query Γ_V soll mit aktuellem Zustand q_0 des Protokollautomaten und leerem Aufrufstack mögliche Belegungen der Variablen y ermitteln. Um die Darstellung übersichtlich zu halten,

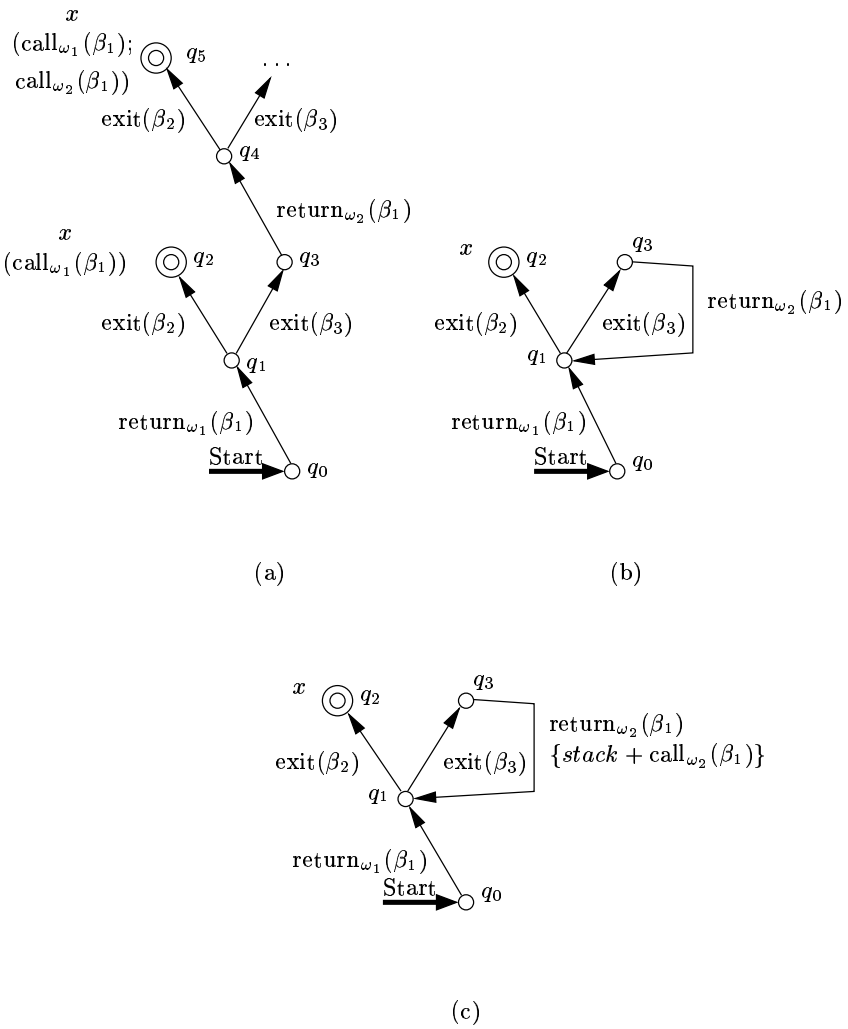


Abbildung 13.3: Verschiedene Strategien für den Umgang mit unendlichen Zustandsmengen

ist in der Abbildung zu jeder Query in geschweiften Klammern nur der aktuelle Zustand der Query, sowie der von der Query mitgeführte Aufrufstack angegeben. Die dabei je nach Strategie entstehenden Automaten, die diese Zustände enthalten, sind in Abbildung 13.3 dargestellt. In den folgenden Schritten wird zunächst der Automat aus Abbildung 13.3(a) betrachtet werden.

2. Die Query Γ_V wird wie eingezeichnet vom Set–Knoten zum Call–Knoten geleitet.

3. Von dort aus wird die Querybearbeitung im Block β_1 der Funktion `f` in Abbildung 13.2(c) an der return–Operation fortgesetzt, wobei ein Zustandsübergang zum neu erzeugten Zustand q_1 mit dem Eingabesymbol $\text{return}_{\omega_1}(\beta_1)$ und ein Ablegen des Symbols $\text{call}_{\omega_1}(\beta_1)$ auf dem Aufrufstack geschieht.

4. Die Query Γ_V, die in den Block β_2 der `if`–Anweisung gesendet wird, findet dort eine Adress–Operation für die Variable x und terminiert erfolgreich unter Generierung eines Endzustandes q_2, zu dem der Rest–Aufrufstack $\text{call}_{\omega_1}(\beta_1)$ ebenfalls gespeichert werden muß (in Abb. 13.3(a) in Klammern neben dem Ergebniszustand q_2 angedeutet).

5. Bei Betreten des anderen Zweiges β_3 wird der Zustand q_3 erzeugt.

6. Unter Generieren des Zustandes q_4 und dem Ablegen eines zusätzlichen Symbols $\text{call}_{\omega_2}(\beta_1)$ auf dem Aufrufstack veranlasst der bei Anweisung ω_2 vorgefundene Funktionsaufruf eine Fortsetzung der Query an der return–Operation, den die Query im Zustand q_1 bereits besucht hat. Allerdings ist der Aufruf–Stack der aktuellen Query $\text{call}_{\omega_1}(\beta_1);\text{call}_{\omega_2}(\beta_1)$unterschiedlich von dem der vorherigen Query, so daß von dem Ergebnis der ersten Query nicht auf das Ergebnis der zweiten Query geschlossen werden kann. Also muß die Query fortgesetzt werden und würde im weiteren Verlauf eine Automaten mit unendlicher Zustandsmenge wie in Abbildung 13.3(a) erzeugen, bei dem sich der dort abgebildete Automat in gleicher Form unendlich weiter erstrecken würde.

13.1.4 Lösungsvorschlag

Da sich auf der Schleife über den rekursiven Aufruf keine relevanten Änderungen für das schließlich gefundene Ergebnis x der Rekursion ergeben, könnte man vereinfachend eine Wiederholung im Automaten gemäß Abbildung 13.3(b) erzeugen und damit das exakte Ergebnis, das eine kontextfreie Sprache zur Beschreibung einer Pfadmengenlösung benötigt, approximativ durch eine reguläre Sprache beschreiben. Dadurch könnte dann natürlich dort Ungenauigkeit entstehen, wo eine exakte Korrespondenz der Menge der Aufrufe mit der Menge der Rücksprünge zu einem exakten Ergebnis nötig wäre.

Alternativ könnte man durch *k–Limiting* die Größe der maximal mitgeführten Aufruf–Stacks begrenzen, wie dies bei allen Call–String–Verfahren aus der Literatur der Fall ist. Dies wäre genauer als eine reine reguläre Beschreibung. Bei der Analyse wird dadurch aber bereits Genauigkeit verloren, die nicht mehr wiederherzustellen ist.

Die wohl beste Lösung bestünde daher in der Erzeugung von Meta–Informationen wie dies in Abbildung 13.3(c) dargestellt ist. In geschweiften Klammern ist an der Rückkante die Information dargestellt, daß bei jedem Durchlaufen der Schleife der Aufrufstack um das Symbol $\text{call}_{\omega_2}(\beta_1)$ erweitert wird. Damit ist dies eine exakte Beschreibung der Situation und es könnten bei Bedarf auf dieser Meta–Ebene weitere Schlüsse gezogen werden.

Ein Beispiel für einen möglichen Ausgangspunkt solch einer Analyse auf Meta–Ebene ist in Abbildung 13.4 dargestellt. Dabei ist zunächst eine beliebig häufige Wiederholung des (in Vorwärtsrichtung betrachtet) Rücksprungteils der Rekursion möglich, bei der bei jedem Durchlauf der Rückwärtskante des Automaten der Stack entsprechend modifiziert werden müsste. Im Aufrufteil der Rekursion ergeben sich nach genau zwei rekursiven Aufrufen (in Vorwärtsrichtung) für die angenommene Query der Wert x und alternativ dazu nach genau einem Aufruf der Wert y (die für ein solches Ergebnis eigentlich zusätzlich notwendige Verzweigung ist in diesem Automaten der Einfachheit halber nicht dargestellt). Bei einer Betrachtung eines solchen Automaten auf dieser Ebene läßt sich klar erkennen, wie oft die Rückwärtskante durchlaufen werden muß, um ein korrektes Ergebnis x bzw. y zu

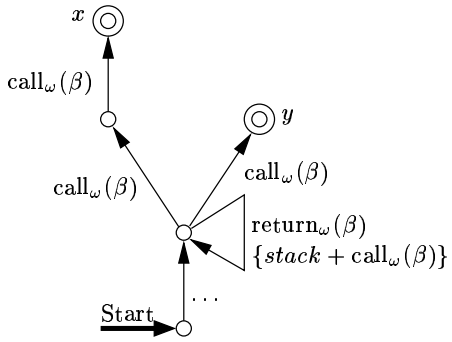


Abbildung 13.4: Analyse von Aufrufpfaden auf Meta–Ebene.

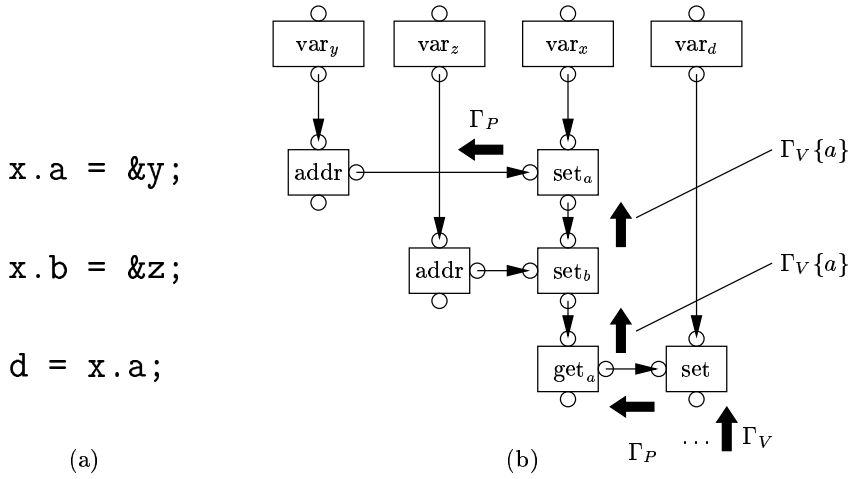


Abbildung 13.5: Member–Zugriffe in der neuen Programmrepräsentation.

erhalten. Im Fall von komplizierteren Protokollautomaten kann die Meta–Analyse an dieser Stelle immer noch die Entscheidung treffen, approximativ zu werden.

13.2 Strukturierte Datentypen

Interessanterweise läßt sich mit den Methoden dieser Arbeit die Analyse von strukturierten Datentypen ebenfalls auf kontextfreie Sprachen zurückführen. Um dies zu zeigen, wird zunächst die notwendige Erweiterung der neuen Programmrepräsentation um entsprechende Operationen vorgestellt.

13.2.1 Erweiterung der Analysegraphen um Member–Zugriffe

Für strukturierte Datentypen müssen Operationen zur Verfügung gestellt werden, die den Zugriff auf Elemente (Members) von Variablen von solchem Typ ermöglichen. Dies wird durch Set–Member und Get–Member Operationen geleistet. Diese Operationen sind in Abbildung 13.5 in einem Beispiel dargestellt. Das Programmfragment in Teilabbildung (a) wird durch die Operationen aus Teilabbildung (b) repräsentiert. Die Zuweisung zum Member a der Variable x wird durch die abgebildete set$_a$ Operation dargestellt, das Auslesen von $x.a$ durch die entsprechende get$_a$–Operation.

Zugriffe der Form `x -> a` in C–Notation, die hier im Beispiel nicht vorkommen, werden durch

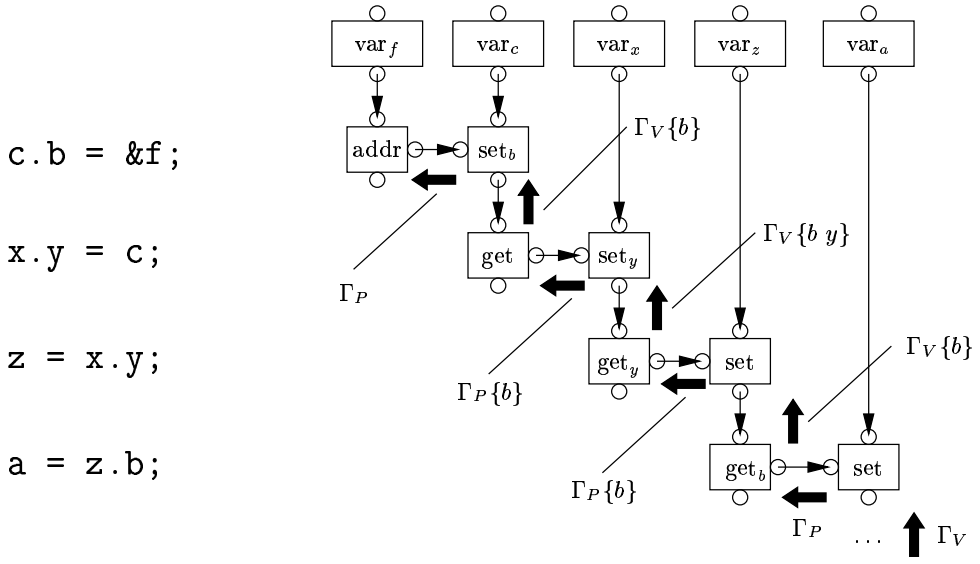


Abbildung 13.6: Notwendigkeit von Member–Stacks.

Get– bzw. Set–Member–Operationen dargestellt, deren verwendete Variable vom Ergebnis einer Dereferenzierungs–Operation abhängt.

13.2.2 Erweiterung der Queryverarbeitung um Member–Stacks

Die Erweiterung der Queryverarbeitung um diese neuen Operationen wird in Abbildung 13.5 anhand einer beispielhaften Berechnung des Queryflusses vorgestellt. Die Berechnung der unteren Query Γ_V erfolgt wie bisher durch die weitergeleitete Query Γ_P. An der get$_a$–Operation, die von dieser erreicht wird, wird aber nicht wie bisher der Wert einer Variablen ausgelesen, sondern der Inhalt des Members einer strukturierten Variablen. Deswegen muß die Query mitprotokollieren, auf welchen Member zugegriffen wurde. In der Abbildung wird dies durch den Namen dieses Members in geschweiften Klammern dargestellt. Der Sinn des Mitführens dieser Information wird bei Erreichen der set$_b$–Operation deutlich. Dort kann der Queryfluß eindeutig weiter entlang der Use–Use–Chain für x geleitet werden, bis zu einer Set–Member–Operation mit "passendem" Member–Zugriff.

Daß für diese Methode der Queryverarbeitung nicht nur der Name eines einzigen Members mitgeführt werden muß, kann man im Beispiel in Abbildung 13.6 erkennen. Dort ist analog zum ersten Beispiel das Programmfragment aus Teilabbildung (a) durch den Analysegraphen aus Teilabbildung (b) repräsentiert worden. Bei einer Queryverarbeitung analog zur bisher geschilderten Vorgehensweise ergeben sich dabei zu den Queries die jeweils in geschweiften Klammern angegebenen Member–Stacks.

13.2.3 Probleme

Für dynamische rekursive Datenstrukturen ergeben sich potentiell unendliche Member–Stacks bei der Analyse. Diese unendlichen Member–Stacks sind anschaulich gesehen die Ursache für das Problem der Nicht–Berechenbarkeit von Aliasing als Form der Zeigeranalyse mit dynamischen rekursiven Datenstrukturen. Betrachtet man die Rückführung von Post's Korrespondenzproblem auf Aliasing in [Ram94] aus der Perspektive der hier vorgestellten Member–Stacks, so finden sich im dort angegebenen Eingabeprogramm Programmpfade, auf denen beliebig große Member–Stacks aufgebaut

werden können. Eine Query aus dieser Arbeit muß für eine Lösung so lange Pfade in Rückwärtsrichtung durchlaufen, bis sie bei leerem Member–Stack eine Adress–Operation findet. Eine exakte Lösung müsste daher immer größere Member–Stacks erzeugen und diese in immer länger werdenden Pfadabschnitten wieder abbauen, ohne daß, analog zur Situation bei den Aufruf–Stacks, hierbei eine Wiederholung festgestellt werden kann.

13.2.4 Lösungsvorschlag

Analog zu den Aufruf–Stacks kann bei Eingabeprogrammen ohne dynamische rekursive Datenstrukturen eine Identifikation einer endlichen Menge von verschiedenen möglichen Member–Stacks mit Zuständen des regulären Ergebnisautomaten zu einer endlichen, exakten regulären Beschreibung einer Lösung für Zeigeranalyse führen.

Im Fall von dynamischen rekursiven Datenstrukturen bietet sich analog zum Lösungsvorschlag für rekursive Funktionen eine Beschreibung von Wiederholungen auf Meta–Ebene an. Die Interpretation der Meta–Information wäre dann eine Beschreibung, um welche Elemente ein Member–Stack bei Durchlaufen eines bestimmten Zustandsüberganges erweitert werden würde. Aus dieser Information könnte man dann nach weiterer Analyse schließen, daß jedes Durchlaufen eines solchen Zustandsüberganges ein "Pfadpräfix" an den Protokollautomaten hinzufügen würde.

Möglicherweise ließe sich auf diese Weise eine exakte Beschreibung von Zeigeranalyse mit dynamischen rekursiven Datenstrukturen auf Meta–Ebene leisten. Das Problem der Nicht–Berechenbarkeit würde dann erst beim Vergleich von zwei verschiedenen solchen Meta–Pfadmengen auftreten, also erst beim Test, ob zwei verschiedene Zeigerausdrücke die gleiche Variable zum Ziel besitzen können.

13.3 Kombination von mehreren Analysearten

Ein bisher ungelöstes Problem ist es, verschiedene Analysefragestellungen wie z.B. Zeigeranalyse und Typanalyse, d.h. eine Lösung der Fragestellung, welche konkreten Klassentypen Instanzen von Objektklassen an verschiedenen Stellen im Programm besitzen können, nahtlos und idealerweise unter Beibehaltung von Exaktheit beider Analysen zu integrieren. Bisherige Arbeiten wie die von Pande und Ryder [PR95] beschreiben Genauigkeitsverluste an der Schnittstelle zwischen den beiden abwechselnd durchgeführten Analysealgorithmen. Eine Analyse von *relational attributes*, die allgemein Programmeigenschaften aus verschiedenen Analysefragestellungen auf gemeinsames Vorkommen in Programminstanzen untersucht, ist bisher in der Literatur anscheinend nicht einmal angedacht worden.

An dieser Stelle könnte eine weitere Ausbaustufe der vorgestellten Analysetechnik ansetzen. Die vorgestellte Theorie der Bedingungsautomaten läßt sich auch auf andere Fragestellungen ausdehnen. Dies ist in Abbildung 13.7 schematisch dargestellt. Das theoretische Fundament der Bedingungsautomaten und deren Anwendung auf Multi–Level–Zeigeranalyse wurde in dieser Arbeit vorgestellt. Dies soll durch die dicke Umrandung in der Abbildung zum Ausdruck gebracht werden.

Als nächste Erweiterung, die in Teilen der Arbeit bereits angesprochen wurde, bietet sich Konstantenanalyse mit Berücksichtigung von Verzweigungsbedingungen an (in der Literatur als Analyse von *conditional constants* bezeichnet). Durch das Einführen von Programmeigenschaften, die das Ergebnis der Auswertung von Verzweigungsbedingungen zur Aussage haben, und analog zu den pfadbeschreibenden Eingabesymbolen an subblock(s)–Operationen in die Automaten M_ω integriert werden könnten, ließe sich die vorgestellte Theorie direkt auf dieses Problem anwenden. Da eine exakte Lösung für dieses Problem bereits nicht mehr berechenbar ist, müssen hierfür approximative Erweiterungen des Verfahrens gefunden werden.

Die Kombination von Multi–Level–Zeigeranalyse mit Konstantenanalyse erlaubt die Behandlung von Zeigerarithmetik, wie bereits in Abschnitt 11.2 angesprochen wurde. Analog zur Zeigeranalyse läßt auch sich Typanalyse durch Queries realisieren, indem Queries nach im Analysegraphen erreichbaren `new`–Operatoren suchen.

Damit wird als Gesamtziel eine kombinierte Analyse von *relational attributes* für alle diese Fragestellungen für z.B. beliebige C++– oder Java–Programme denkbar.

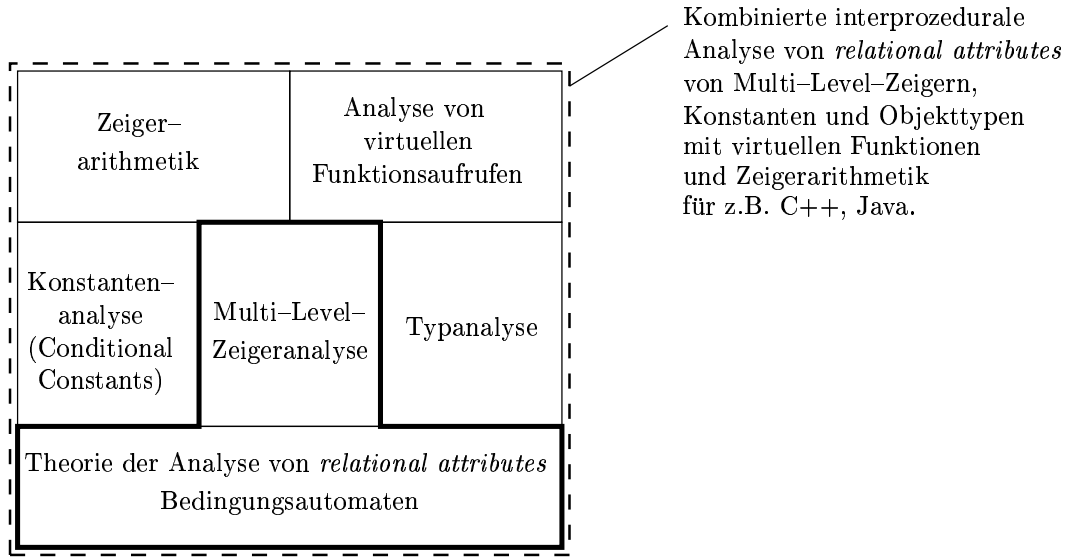


Abbildung 13.7: Integration von verschiedenen Analysearten auf der Basis der vorgestellten Theorie.

13.4 Zusammenfassung

Die Aspekte von interprozeduraler Analyse und der Behandlung von strukturierten Datentypen lassen sich in das vorgestellte Verfahren integrieren. Dabei genügen häufig reguläre Protokollautomaten für eine exakte Beschreibung von Ergebnissen. Für die in manchen Situationen zu erwartenden unendlichen Zustandsmengen wurden Lösungsvorschläge auf der Basis einer weiteren Analyseabstraktionsebene vorgeschlagen.

Die vorgestellte Theorie der Bedingungsautomaten läßt sich auch auf andere Analysefragestellungen anwenden, so daß eine kombinierte Zeiger– Typ– und Konstantenanalyse für beliebige C++– oder Java–Programme realisierbar wird.

Literaturverzeichnis

[AG98] D. Atkinson and W. Griswold. Effective whole–program analysis in the presence of pointers. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 46–55, 1998.

[All70] Frances Allen. Control flow analysis. *SIGPLAN Notes*, 5(7):1–19, July 1970.

[And94] Lars Ole Andersen. *Program Analysis and Specification for the C Programming Language.* PhD thesis, University of Copenhagen, May 1994.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison–Wesley, 1988.

[BC92] M. Burke and J.D. Choi. Precise and efficient integration of interprocedural alias information into data–flow analysis. *ACM Letters on Programming Languages and Systems*, 1(1):14–21, March 1992.

[BCCH94] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow–insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A Nicoleau, and D Padua, editors, *Lecture Notes in Computer Science*, number 892, pages 234–250. Springer–Verlag, September 1994.

[BCCH97] Michael Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Interprocedural pointer alias analysis. Research report, IBM, 1997. Computer Science/ RC 21055 (12/15/97).

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow–sensitive interprocedural computation of pointer–induced aliases and side effects. In *20th Annual ACM SIGACT–SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, 1993.

[CFR+91] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependency graph. *ACM Transactions on Programming Language and Systems*, 13(4):451–490, 1991.

[CGJ+94] Massimiliano Chiodo, Paolo Giusto, Attila Jurecska, Harry H. Hsieh, Alberto Sangiovanni-Vincentelli, and Luciano Lavagno. Hardware–software codesign of embedded systems. *IEEE Micro*, 14(4):26–36, August 1994.

[CWZ90] David R. Chase, Mark W. Wegman, and Kenneth F. Zadeck. Analysis of pointers and structures. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, number 25(6) in SIGPLAN Notices, pages 296–310, June 1990.

[Deu90] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher–order functional specifications. In *Conference record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, 1990.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right–regular equivalence relations. In *Proceedings of the IEEE 1992 Conference on Computer Languages*, pages 2–13, 1992.

[Deu94] Alain Deutsch. Interprocedural may–alias for pointers: beyond k-limiting. In *SIGPLAN '94 Conference on Programming Languages Design and Implementation*, number 29(6) in SIGPLAN notices, pages 230–241, 1994.

[DGS95] E. Duesterwald, R. Gupta, and M. L. Soffa. Demand–driven computation of interprocedural data flow. In *Conf. Rec. 22nd Symposium on Principles of Programming Languages (POPL'95)*, pages 37–48, NY, 1995. ACM.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[dM99] Giovanni de Micheli. Hardware synthesis from C/C++–models. In *Proceedings of the '99 Conference on Design, Automation and Test in Europe*, pages 382–384, Munich, Germany, 1999.

[DS95] Giovanni De Micheli and Mariagiovanni Sami, editors. *Hardware/Software Co–Design.* Kluwer, 1995.

[EGH94] M. Emami, R. Ghiya, and L.J. Hendren. Context–sensitive interprocedural points–to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montreal, Canada, 1993.

[FFA00] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow–insensitive points–to analysis for C. In *Proceedings of the International Symposium on Static Analysis*, 2000.

[FRD00] M. Fähndrich, J. Rehof, and M. Das. Scalable context–sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[GH96a] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 1996.

[GH96b] R. Ghiya and L. Hendren. Is it a tree, a dag or a cyclic graph? a shape analysis for heap–directed pointers in C. In *Conference Record of the 23rd Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[GH98] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Conference Record of the 25th Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability.* W.H. Freeman, 1979.

[GM93] Rajesh K. Gupta and Giovanni De Micheli. Hardware–software cosynthesis for digital systems. *IEEE Design and Test of Computers*, 10(3):29–41, September 1993.

[Gua88] C. A. Guarna. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International conference on Parallel Processing*, pages 212–220, 1988.

[HH98] H. Hasti and S. Horwitz. Using static single assignment form to improve flow–insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 97–105, 1998.

[HHN92] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: improving analysis and transformations of imperative languages. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, 1992.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24 of *SIGPLAN Notices*, pages 28–40, 1989.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23, pages 35–46, 1988.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Eng. (FSE'95)*, pages 104–115, 1995.

[HS94] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition–use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Addison–Wesley, 1979.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the 1st ACM Symposium on Principles of Programming Languages (POPL'73)*, pages 194–206, Boston, Massachusetts, 1973.

[Kno98] Jens Knoop. *Optimal Interprocedural Program Optimization : a new framework and its applications*, volume 1428 of *Lecture Notes in Computer Science.* Springer Verlag, 1998.

[KU77] J.B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:309–317, 1977.

[Lan92a] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[Lan92b] William Landi. *Interprocedural aliasing in the presence of function pointers.* PhD thesis, Rutgers University, New Brunswick, N.J., 1992.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, volume 23 of *SIGPLAN Notices*, pages 21–34, Atlanta, Georgia, June 1988.

[Lil97] John Lilley. John lilley's pccts–based ll(1) c++–parser, version 1.5. http://www.empathy.com/pccts/index.html, February 1997.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, 1991.

[LR92] W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Languages Design and Implementation*, pages 235–248, June 1992.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.

[MD00] Robert Muth and Saumya Debray. On the complexity of flow–sensitive dataflow analyses. In *Proceedings of the 2000 Conference on Principles of Programming Languages*, pages 67–80, Boston, Massachusetts, 2000.

[MJ81a] S.S. Muchnick and N. D. Jones. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In S.S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 12, pages 380–393. Prentice-Hall, 1981.

[MJ81b] S.S. Muchnick and N. D. Jones. Flow analysis and optimization of lisp–like structures. In S.S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[MOR01] Markus Müller-Olm and Oliver Rüthing. The complexity of constant propagation. In D. Sands, editor, *ESOP 2001 (10th European Symposium on Programming)*, volume 2028 of *Lecture Notes in Computer Science (LNCS)*, pages 190–205, Heidelberg, Germany, 2001. Springer-Verlag.

[MOS02] Markus Müller-Olm and Helmut Seidl. Polynomial constants are decidable. In *Proc.9th Int. Static Analysis Symposium (SAS'02), Madrid, Spain*, Lecture Notes in Computer Science (LNCS), Heidelberg, Germany, September 2002. Springer-Verlag. To appear.

[MR00] D. Melski and T. Reps. Interconvertibility of a class of set constraints and context-free language reachability. *Theoret. Comput. Sci.*, 248(1-2), 2000.

[PR95] Hemant Pande and Barbara G. Ryder. Static type determination and aliasing for C++. Technical Report LCSR-TR-250-A, Rutgers University, October 1995.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(6):1467–1471, November 1994.

[Rep94] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proc. 5th Int. Conf. on Compiler Construction (CC'94)*, volume 786 of *Lecture Notes in Computer Science(LNCS)*, pages 389–403, 1994.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[RHSR94] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[RM88] C. Ruggieri and T. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the 15th Annual ACM Symposium on the Principles of Programming Languages*, pages 285–293, 1988.

[RS98] John L. Ross and Mooly Sagiv. Building a bridge between pointer aliases and program dependencies. *Nordic Journal of Computing*, 5:361–386, 1998.

[Ruf95] Erik Ruf. Context–insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 13–22, 1995.

[SDM98] L. Semeria and G. De Micheli. SpC: Synthesis of pointers in C, application of pointer analysis to the behavioural synthesis from C. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 340–346, San Jose, CA, November 1998.

[SH97] M. Shapiro and S Horwitz. Fast and accurate flow–insensitive point–to analysis. In *24th Annual ACM SIGACT–SIGPLAN Symposium on the Principles of Programming Languages*, 1997.

[SK91] Bernhard Steffen and Jens Knoop. Finite constants: characterizations of a new decidable set of constants. *Theoretical Computer Science*, 80:303–318, 1991.

[SN98] Guido Schumacher and Wolfgang Nebel. Object–oriented modelling of parallel hardware systems. In *Proceedings of the '98 Conference on Design, Automation and Test in Europe*, pages 234–242, Paris, France, 1998.

[SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, 1981.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape–analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.

[Ste96a] Bjaarne Steensgard. Points–to analysis in almost linear time. In *23rd Annual ACM SIGACT–SIGPLAN Symposium on the Principles of Programming Languages*, pages 32–41, January 1996.

[Ste96b] Bernhard Steffen. Property oriented expansion. In *Proc. Int. Static Analysis Symposium (SAS'96), LNCS*, volume 1145, pages 22–41, Aachen (D), September 1996. Springer Verlag.

[Tar81] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577 – 593, 1981.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan 1980.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In Inc ACM, editor, *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implentation*, pages 1–12. Acm,Inc, June 18-21 1995.

[WWW] http://www.prolangs.rutgers.edu/ftp/ansi.pub.tar.gz.

[YHR99] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. *ACM*, May 1999. SIGPLAN'99.