Graphersetzung mit Anwendungen im Übersetzerbau

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik

der Universität Fridericiana zu Karlsruhe (TH)

genehmigte

Dissertation

von

Rubino R. Geiß

aus Pforzheim

Tag der mündlichen Prüfung: 31. Oktober 2007
Erster Gutachter: Prof. em. Dr. Dr. hc. Gerhard Goos
Zweite Gutachterin: Prof. Dr. Dorothea Wagner

VORWORT & DANKSAGUNG

In dieser Arbeit werden die theoretischen Grundlagen, der Entwurf, die Implementierung sowie einige Anwendungen von GrGen, einem Werkzeug zur Graphersetzung beschrieben. Zur genaueren Studie der Syntax und Semantik von GrGen sei hier auf das Benutzerhandbuch verwiesen [BG07, Gei08]. Im Gegensatz zu diesem vertieft der vorliegende Text die theoretischen Grundlagen, die Integration der Graphersetzung in den Übersetzerbau sowie das neue und erstmals in GrGen eingesetzte Verfahren zur Graphmustersuche, den suchplanbasierten Ansatz. Obschon dieser Arbeit eine konkrete Implementierung zugrunde liegt, können weite Teile losgelöst von jener verstanden werden und sollten, insbesondere bei der Weiter- und Neuentwicklung von Werkzeugen zur Graphersetzung, bei deren Einsatz sowie einer gleichermaßen *gründlichen* wie *schlanken* theoretischen Fundierung solcher Vorhaben eine starke Basis bilden.

Schon vor Beginn meiner Tätigkeit am Lehrstuhl Goos im Jahr 2000 war ich fasziniert von den Möglichkeiten neuerer Prozessoren; insbesondere Multimedia-Erweiterungen wie MMX, SSE, AltiVec etc. fanden meine Aufmerksamkeit. Mit diesen, wie ich sie nenne *reichhaltigen Befehlen*, kann ein Programm nicht nur, wie im Übersetzerbau üblich, um z. B. 5 bis 20 % beschleunigt werden, vielmehr ist eine Leistungssteigerung um *Faktor* 5 bis 20 möglich. Dem Ausnutzen dieses signifikanten Verbesserungspotenzials haben sich dennoch, wohl wegen der damit verbundenen (algorithmisch und komplexitätstheoretisch) sehr aufwendigen Fragestellungen, bis heute nur wenige Arbeiten systematisch angenommen. Die hierzu am IPD Goos entstandenen Arbeiten von Enno Hofmann [Hof04] und Andreas Schösser [Sch07b, SG08] sind als konkurrenzlose Pionierleistungen zu bezeichnen.

Zunächst allerdings war mein Lösungsansatz ein anderer, als der bei Hofmann und Schösser realisierte: Ich versuchte das Phasenproblem im Übersetzerbau – inklusive der geeigneten Auswahl reichhaltiger Befehle – mittels ganzzahliger linearer Optimierung[1] zu lösen. Dabei stellte sich heraus, dass es extrem schwierig ist, eine geeignete, d. h. verstehbare, korrekte und (hinreichend) effizient lösbare Formulierung zu finden. Alle Versuche auf diesem Gebiet – auch von Dritten – scheiterten aus praktischer Sicht letztlich an der nicht beherrschbaren Laufzeitkomplexität [Käs97, Men99]. Bemerkenswert dabei ist, dass Steven Bashford dieses Scheitern durch Wahl extrem kleiner zu bearbeitender Quellprogramme (oft nur idealisierte Ausschnitte aus inneren Schleifen) recht erfolgreich kaschiert [BL99, Bas00, LB00]. Die Arbeiten von Torsten Menne [Men99], einem Studenten von Bashford sowie von Daniel Kästner [Käs97] stellen die Verhältnisse deutlich realistischer dar: Diese Ansätze sind wegen ihrer prohibitiven Laufzeit *praktisch* unbrauchbar.

Unter dem Eindruck der Arbeit von Erik Eckstein, Oliver König und Bernhard Scholz [EKS03] sowie einer Erweiterung ihrer Methode am IPD Goos durch Hannes

[1] engl.: *integer linear programming* (ILP)

Jakschitsch [Jak04] wurde mir klar, dass das Problem der Mustersuche ein nicht ausreichend beachtetes und zunächst unabhängig von den anderen Herausforderungen zu lösendes Problem ist. Falls hinreichend komplexe Muster genügend schnell zu finden wären, so könnte man die Auswahl danach entweder ad hoc erledigen oder mit der Methode von Eckstein/König/Scholz auf eine solide Grundlage stellen. So begann das GrGen Projekt im Frühjahr 2003 mit einer von mir betreuten Diplomarbeit, die von Prof. Dr. Sebastian Hack [Hac03] – der damals Student und später Mitarbeiter am Lehrstuhl Goos war – bearbeitet wurde. Damals brauchten wir ein Werkzeug, um Muster in graphbasierten Zwischensprachen, wie sie im Übersetzerbau eingesetzt werden, zu finden. Wir stellten uns ein Werkzeug vor, das in der Lage sein sollte, schnell Muster finden zu können, über eine ausdrucksstarke Spezifikationssprache verfügen würde und überdies leicht in die existierende (in C implementierte) Übersetzer-Infrastruktur integriert werden könnte.

Bis dahin war Optimix [Ass00] das einzige Werkzeug, das einen Brückenschlag zwischen Graphersetzung und Übersetzerbau gewagt hatte. Jedoch war es der Ansatz von Optimix, möglichst viele Eigenschaften von Regelmengen (auch a priori) zu garantieren; darunter Terminierung, Konfluenz der Ableitungen sowie Überdeckung von Graphen. Dies wurde dadurch erreicht, dass die Mächtigkeit des Formalismus weit eingeschränkt wurde, nämlich unterhalb der Turingvollständigkeit. Erst dadurch sind solche Eigenschaften, z. T. dann sogar für alle möglichen Mengen von Regeln, a priori beweisbar. Unser – zunächst nur gedachtes – Werkzeug sollte im Gegensatz dazu turingvollständig sein. Dadurch würde es dem Benutzer obliegen, solcherlei Eigenschaften für seine Graphersetzungsanwendung zu beweisen; er hätte aber im Gegenzug die volle Mächtigkeit, d. h. er könnte tatsächlich auch alle möglichen Transformationen durchführen.

Um schnellstmöglich an einen funktionierenden Prototyp zu kommen, delegierten wir das kostspielige (und auch aufwendig zu implementierende) Finden von Passungen (Teilgraphisomorphie) an ein relationales Datenbanksystem [Hac03, Gru04]. Dieser gemeinsam mit Dr. Markus Noga ersonnene Weg ermöglichte es tatsächlich in weniger als einem Jahr ein funktionierendes Graphersetzungssystem in den Händen zu halten, dem wir den Namen GrGen (GRaphersetzungs GENerator) gaben. Obgleich wir in den folgenden Jahren die Leistungsfähigkeit der datenbankbasierten Implementierung erheblich – je nach Anwendung um Faktoren zwischen 10^7 und 10^{10} – steigern konnten, glaubten wir, dass das Ende der Fahnenstange noch nicht erreicht sei. Inspiriert durch eine Dissertation von Heiko Dörr [Dör95] überarbeiteten wir GrGen, sodass es nicht länger eine Datenbank benötigte, sondern die Mustersuche mittels eines suchplanbasierten Ansatzes [Bat05a, Bat06, BKG08] selbst erledigte. Dieser neuartige Ansatz zur Mustersuche, die Spezifikationssprache der Regeln und deren Anwendungssteuerung entwickelten sich mit der Zeit und reiften heran [Bat05b, Sza05, GBG+06]. Im Jahr 2006 schließlich begann Moritz Kroll das in C geschriebene GrGen nach .NET zu portieren, um einerseits die Produktivitätssteigerung durch C# zu nutzen und andererseits unser Werkzeug einer breiteren Nutzergemeinde (insbesondere außerhalb des Übersetzerbaus) zugänglich zu machen. So entstand das in C# geschriebene GrGen.NET [GK08, GK07, Kro07, BKG08, Jak07, Mül07, TBB+08, VAB+08, Jak08, Kro08, Buc08]. Für die ursprünglichen Aufgaben im Übersetzerbau besteht eine, in C geschriebene und in die Übersetzerbauinfrastruktur eingebettete Variante von GrGen weiter.

Obwohl wir vor fünf Jahren lediglich damit begannen einen Mustersucher für ein *spezifisches Problem* im Übersetzerbau zu entwickeln, ist GrGen.NET in der Zwischenzeit zu einem ansehnlichen *allgemeinen Werkzeug* zur Graphersetzung herangereift; dies beweisen nicht zuletzt verschiedene Arbeiten im Bereich der Softwaretechnik [GDG08, DGG08, GT07].

Ich möchte mich an dieser Stelle bei allen Mitarbeitern und besonders bei den Studenten bedanken, die während unzähliger Stunden beim Entwurf, der Implementierung und nicht zuletzt auch bei der Dokumentation Großes geleistet haben. Besonderen Dank möchte ich an Prof. Dr. Sebastian Hack, Gernot Veit Batz, Michael Beck, Tom Gelhausen, Moritz Kroll, Dr. Andreas Ludwig und Dr. Markus Noga richten – ohne jeden Einzelnen wäre diese Arbeit nicht das, was sie heute ist.

Mein Dank geht auch an alle, die diese Arbeit in verschiedenen Stadien korrekturgelesen haben sowie mir mit hilfreichen Kommentaren und erhellenden Diskussionen halfen: Jakob Blomer, Matthias Braun, Sebastian Buchwald, Karin Daiß, Oliver Denninger, Bugra Derre, Leif Geiger, Elisabeth Geiß, Rudi R. Geiß, Daniel Grund, Dr. Berthold Hoffmann, Enno Hofmann, Hannes Jakschitsch, Edgar Jakumeit, Prof. Hans-Jörg Kreowski, Christoph Mallon, Jens Müller, Christoph Schaefer, Jochen Schimmel, Andreas Schösser, Wolf-Dieter Starke, Adam Szalkowski, Prof. Dániel Varró, Prof. Dorothea Wagner, Prof. Albert Zündorf und Andreas Zwinkau. Insbesondere möchte ich mich bei Prof. Gerhard Goos für die produktive Atmosphäre und die großzügige Unterstützung bedanken, ohne die eine solche langfristig ausgerichtete Arbeit nicht möglich gewesen wäre.

Schließlich geht mein besonders herzlicher Dank an Gunther Daiß, der mir die Freiheit gab, mich auf diese Arbeit zu konzentrieren, wenn es nötig war – jedoch nicht ohne das Leben zu vergessen. Danke für Deine Liebe.

Die vorliegende Arbeit ist durch leichte Überarbeitung und Fehlerkorrektur aus meiner Dissertationsschrift hervorgegangen; die Beispiele basieren auf GrGen.NET Version 1.4 oder früher. Selbstredend sind eventuell verbliebene Unzulänglichkeiten dieser Arbeit oder in GrGen.NET nicht den zahlreichen Mitstreitern, sondern ausschließlich mir anzulasten. Sollte Ihnen in diesem Zusammenhang ein Fehler oder eine Unstimmigkeit auffallen, bitte ich um Nachricht, auf dass GrGen.NET noch ein langes, produktives Leben habe: `rubino@ipd.uni-karlsruhe.de`

Karlsruhe und Frankfurt im November 2008,
Rubino R. Geiß

INHALTSVERZEICHNIS

KAPITEL 1

EINLEITUNG

Zur Repräsentation von Daten und Metadaten aus verschiedenen Bereichen von Wissenschaft und Technik werden oft Graphen herangezogen. Operationen auf solchen Repräsentationen können – auf dieser Abstraktionsebene – als Graphersetzungen formuliert werden. In diesem Sinn ist die Graphersetzung eine uniforme Spezifikationstechnik für Operationen auf Daten in Graphdarstellung. Graphersetzungen sind deklarative Spezifikationen, die auf zwei Arten implementiert werden können: Einerseits kann ein Mensch sie von Hand in ausführbare Programme umsetzen; andererseits kann dies durch einen Generator automatisiert geschehen. Letzteres ist für viele Aufgabenstellungen ökonomischer, effektiver und eleganter.

1.1 Ziel der Arbeit

Im Laufe der letzten 30 Jahre ist die Theorie der Graphersetzung zusehends gereift. Die einsetzende Zunahme von Anwendungen, die eine direkte Behandlung von Graphersetzungen voraussetzen, benötigen für die automatisierte Verarbeitung eine *theoretisch fundierte*, *effizient* maschinell ausführbare und *einfach benutzbare* Methode.

Ziel dieser Arbeit ist es, eine Methodik zur Graphersetzung zu erarbeiten, die alle drei Anforderungen in hohem Maße erfüllt. Darüber hinaus betrachten wir die Umsetzbarkeit der Methodik für Aufgaben praktisch relevanter Größe sowie ihre effektive Benutzbarkeit nicht als optionale Eigenschaften, sondern vielmehr als die zentrale Aufgabe dieser Arbeit. Das in diesem Rahmen entstandene Programmsystem trägt den Namen GrGen (GRaphersetzungs-GENerator, `www.grgen.net`).

1.2 Hintergrund

Bevor wir unseren Lösungsansatz vorstellen, klären wir hier was Graphersetzung ist und betrachten ein Anwendungsfeld für Graphersetzungen: Den Übersetzerbau.

1.2.1 Graphersetzung

Das Gebiet der *Graphersetzung*, für das es noch weitere Bezeichnungen wie *Graphtransformation* oder *Graphgrammatiken*[1] gibt, die letztlich dasselbe aussagen, hat

[1] Def Begriff Graphgrammatiken (engl.: *graph gammar*) geht auf Arbeiten von Ehrig und Schneider [EPS73] zurück.

seine Wurzeln in Arbeiten aus den späten sechziger Jahren des letzten Jahrhunderts, wie z. B. einem Aufsatz über „web grammars“ von Pfaltz und Rosenfeld [PR69].

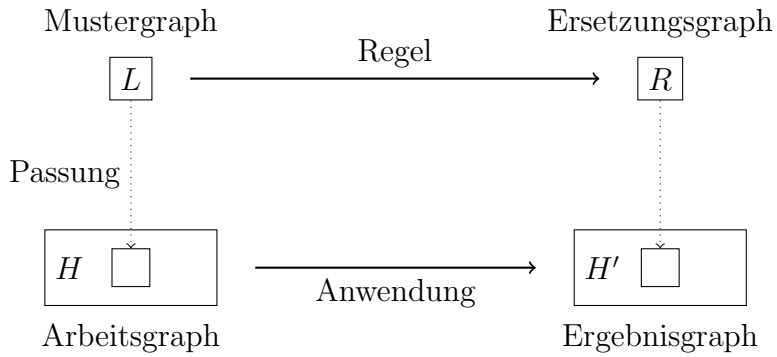


Abbildung 1.1: Vereinfachte Darstellung einer Graphersetzung

Der Begriff der Graphersetzung, wie er in dieser Arbeit verstanden wird, meint eine Methode zur deklarativen Spezifikation von „Änderungen“ an Graphen – vergleichbar etwa mit der *Termersetzung*. Normalerweise werden mehrere *Graphersetzungsregeln* benutzt, um eine gewisse Aufgabe zu lösen. Diese Regelmenge wird in der Literatur im Umfeld von Graphgrammatiken oft Graphersetzungssystem genannt. Wir hingegen benutzen diesen Begriff für ein Softwaresystem, das Graphersetzungen ausführt. Im einfachsten Fall besteht eine solche Graphersetzungsregel aus einem Tupel $(L \to R)$, wobei L linke Seite der Regel *Mustergraph*, und R rechte Seite der Regel *Ersetzungsgraph* heißt. Je nach konkretem Ersetzungsverfahren können die Regeln deutlich komplexer ausgestaltet sein, insbesondere kann der Mustergraph mit zusätzlichen Bedingungen angereichert werden.

Die *Anwendung* einer Regel auf einen *Arbeitsgraphen* H transformiert diesen. Um die Anwendung durchführen zu können, muss eine zum Mustergraphen passende Stelle im Arbeitsgraphen (eine Passung) gefunden werden. Genauer gesagt ist eine *Passung* die isomorphe Abbildung des Mustergraphen auf einen Teilgraphen des Arbeitsgraphen. Technisch gesehen ist das Finden einer Passung, umgesetzt durch eine *Graphmustersuche*, der berechnungsaufwendigste Teil der Graphersetzung. Nachdem eine Passung gefunden wurde, kann die entsprechende Stelle im Arbeitsgraphen so transformiert werden, dass dort ein isomorphes Abbild des Ersetzungsgraphen entsteht. Der daraus resultierende Graph heißt *Ergebnisgraph* H'. Natürlich müssen je nach konkretem Ersetzungsverfahren besondere Nacharbeiten ausgeführt werden, z. B. an den Berührungsstellen zwischen dem ersetzten Teilgraphen und dem von der Regel nicht erfassten Rest des Arbeitsgraphen. In Abbildung 1.1 ist die oben beschriebene Situation schematisch erfasst.

Die für unsere Arbeit relevanten Ansätze aus den Bereichen der Graphersetzung, sowie der damit eng verbundenen Graphmustersuche, stellen wir in Kapitel 3 vor. Wir geben ebenda auch einen Überblick über die gebräuchlichsten Graphersetzungssysteme.

1.2.2 Anwendungsfeld: Übersetzerbau

Die dieser Arbeit zugrunde liegende These ist, dass Graphersetzung in vielen Gebieten von Wissenschaft und Technik nutzbringend einsetzbar ist. Um die Probleme,

die sich bis heute dem breiten Einsatz der Graphersetzung entgegenstellen, greifbar zu machen, betrachten wir hier exemplarisch den *Übersetzerbau*.

Obschon Nagl bereits im Jahr 1979 der Manipulation von Programmgraphen durch entsprechende Ersetzungssysteme großes Potenzial bescheinigte [Nag79], sowie Cooper und Torczon im Jahre 2004 konstatierten, dass Optimierungen auf Zwischensprachebene intuitiv als Transformation von Graphen aufzufassen sind [CT04], wird diese Abstraktion bisher nicht auf die Ebene der Implementierung von Optimierungen heruntergebrochen. Zwar werden Optimierungen häufig mittels grafischer Veranschaulichungen entworfen, jedoch wird bei der händischen Implementierung diese Abstraktion nicht systematisch umgesetzt, was zu Ideosynkrasien führt. Dass die Theorie der *Graphersetzung*, die scheinbar eine geeignete Abstraktion für Optimierungen auf Zwischensprachen darstellt, bisher so wenig in den Übersetzerbau Einzug hielt, hat mehrere Gründe:

- Bis jetzt hat sich noch keine Variante der Graphersetzung als Standard herauskristallisiert. Die Vertreter unterscheiden sich z. T. stark in dem Verständnis von Graphen, der Methode des Findens von Passungen und dem Begriff der Ersetzung. Insofern kann eigentlich nicht von *der* Graphersetzung gesprochen werden, sondern nur von verschiedenen Verfahren, die allesamt *Graphen ersetzen*.

- Es mangelt an Generatoren, die im Stande sind, aus Graphersetzungsregeln Programme zu erzeugen, die ebendiese Ersetzungsregeln auf einen Graphen, in diesem Fall den Zwischensprachgraphen anwenden.

- Vor allem große Graphmuster werden schnell unübersichtlich und ihre Spezifikation erfordert Erfahrung. Das Ineinandergreifen von mehreren Graphtransformationen kann nur schwer mit algorithmusorientiertem Denken in Einklang gebracht werden.

- Bisherige Graphersetzungsansätze betrachten die Integration des Ersetzungsprozesses in die restliche Infrastruktur allenfalls ansatzweise. Die Frage, wie mit gefundenen Mustern außerhalb rein deklarativer Regeln umgegangen wird, bleibt ungelöst.

- Die im Allgemeinen hohe Komplexität des Findens von Passungen wird, da Teilgraphisomorphie ein NP-schweres Problem ist [GJ90], oft als Ausschlusskriterium verwandt.

1.3 Lösungsansatz

Wie wir am Beispiel des Übersetzerbaus gesehen haben, reicht es nicht aus, sich nur auf die Methode der Graphersetzung zu konzentrieren. Wir müssen fernerhin deren Effizienz und Integration in potenzielle Anwendungen im Auge behalten. Selbst eine in diesem Sinn erweiterte Graphersetzung ist nur ein Baustein in einem System, das Graphen zur Problemrepräsentation und -lösung benutzt. Je nach zu bewältigender Aufgabe sind weitere Abstraktionen (Abschnitt 7.1), die auf Graphersetzung aufbauen, zu finden und umzusetzen. Wir werden dies exemplarisch für den Übersetzerbau in Kapitel 7 zeigen, wofür wir in Kapitel 6 die nötigen Grundlagen schaffen und verwandte Arbeiten aus diesem Gebiet betrachten. Für den Übersetzerbau zeigen wir in

Abschnitt 7.4.2, wie sich mit auf Graphersetzung basierenden Methoden Fortschritte im Bereich der maschinenabhängigen Optimierung bei reichhaltigen Befehlssätzen erzielen lassen.

Das Grundgerüst unserer Methodik setzt sich aus den folgenden Elementen zusammen:

Mächtiges Graphmetamodell
: Unsere Metamodelle erlauben die Spezifikation einer Erweiterung von markierten und gerichteten Multigraphen, nämlich *attributierte, typisierte und gerichtete Multigraphen* (Abschnitt 4.1). Dies sind Graphen mit typisierten Ecken und Kanten, wobei zwischen zwei Ecken mehr als eine Kante des gleichen Typs und gleicher Richtung erlaubt ist. Das *Typsystem* bietet Mehrfachvererbung gleichermaßen für Ecken und Kanten. Darüber hinaus ist es möglich, die erlaubten Ecken- und Kantenkonfigurationen durch *Verbindungszusicherungen* einzuschränken.

Trennung von Metamodell, Ersetzungsregeln und Regelanwendung
: Die Ersetzungsregeln und das Metamodell können unabhängig voneinander spezifiziert werden. Dies ermöglicht es, verschiedene Sätze von Regeln mit denselben Metamodellen zu verwenden, und auf diese Weise Regelsätze zu modularisieren. Darüber hinaus benutzen wir eine konzise und leicht erlernbare *textuelle* Schreibweise für Metamodelle, Regeln und Regelanwendungen (Kapitel 4 und Anhang A). Damit ist, anderes als bei grafischen Werkzeugen, eine einfache Integration in automatische Prozesse möglich; ebenso sind große Graphen textuell besser handhabbar.

Theoretisch fundiertes Ersetzungsverfahren
: Unsere Methodik fußt auf einer Erweiterung des wohlbekannten *Single-Pushout Ansatzes* (SPO) der algebraischen Graphersetzung (Abschnitt 2.4). Die Erweiterung besteht aus einem mächtigeren Graphkonzept, Anwendungsbedingungen beim Finden von Passungen und der Möglichkeit, Ecken und Kanten umzutypisieren. Die nötigen mathematischen Fundamente, insbesondere die der *Kategorientheorie*, legen wir in Kapitel 2.

Ausdrucksstarke Ersetzungsregeln
: Die Menge der gültigen Passungen kann über den Mustergraphen hinaus einerseits durch *Typbedingungen*, *Attributbedingungen*, *Vorbelegungen*, *negative Anwendungsbedingungen* und *Homomorphiebedingungen* beschränkt, sowie andererseits mit *dynamischen Mustern* erweitert werden (Abschnitt 4.2). Zusätzlich ist es möglich, dass *Reattributierungsanweisungen*, *Retypisierung*, *dynamische Typisierung* und *Kopieranweisungen* mit einer Regel assoziiert werden (Abschnitt 4.3).

Behandlung von Regel- und Passungsauswahlstrategie
: Die Reihenfolge der Anwendung und Abhängigkeiten zwischen Ersetzungsregeln können elegant und konzise angegeben werden. Die *Graphersetzungssequenzen* erlauben es, die gängigen elementaren Regelauswahlstrategien auszudrücken, gehen aber in der Ausdrucksstärke darüber hinaus (Abschnitt 4.4). Graphersetzungssequenzen sind turingvollständig. Die Auswahl der Passung kann entweder durch zusätzliche Regelannotationen – also statisch – oder durch

Angaben bei der Regelanwendung – also dynamisch – spezifiziert werden (Abschnitt 4.3 und Anhang A.5).

Leicht handhabbare Schnittstellen

Wir entwerfen eine Bibliothek – die sogenannte libGr – zur Repräsentation und Manipulation von Graphen sowie zur Anwendung von Graphersetzungen, einschließlich des Anwendens von Graphersetzungssequenzen (Abschnitt 8.2). Die libGr kann in Anwendungsprogramme integriert werden. Ihre Funktionalität wird für den Nutzer auch über eine interaktive Umgebung – die GrShell – bereitgestellt. Die GrShell ermöglicht insbesondere ein schrittweises Ausführen von Graphersetzungssequenzen. Zusätzlich können die Arbeitsgraphen sowie das schrittweise Anwenden von Regeln visualisiert werden (Abschnitt 8.3).

Anwendungsspezifische Programmiersprachen

Durch das Bereitstellen einer *anwendungsspezifischen Programmiersprache* wird die Integration in konventionelle Programmiersprachen verbessert und dem Benutzer ein mächtiges Werkzeug zur Graphmanipulation an die Hand gegeben. Je nach Anwendungsdomäne muss diese anwendungsspezifische Programmiersprache den dort auftretenden spezifischen Anforderungen angepasst werden. Wir führen dies in Abschnitt 7.4.1 für das Anwendungsfeld des Übersetzerbaus durch.

Optimierung der Mustersuche

Graphersetzung erfordert das Finden des zu ersetzenden Teilgraphen. Allerdings ist diese Graphmustersuche ein NP-vollständiges Problem (vgl. Garey and Johnson, GT48 [GJ90]). Mithin ist das effiziente Beherrschen der Graphmustersuche für möglichst viele Probleminstanzen ausschlaggebend für den Erfolg der Methodik. Wir definieren hierzu eine virtuelle Maschine, deren Programme verschiedenen *Graphsuchstrategien* entsprechen. Diese Programme können dann anhand verschiedener Kostenmodelle selektiert werden. Fernerhin kann die virtuelle Maschine die Programme zur Laufzeit neu optimieren und mit Techniken der dynamischen Programmierung den Suchraum heuristisch weiter beschneiden. Die Fortschritte auf diesem Gebiet werden in Kapitel 5 vorgestellt.

TEIL I

Theorie

KAPITEL 2

GRUNDLAGEN

Die Modellierung einer Methodik zur Graphersetzung und ihre erfolgreiche Implementierung hat viele Facetten. Wir stellen in diesem Kapitel die grundlegenden theoretischen und praktischen Ansätze der verschiedenen involvierten Bereiche vor. Dabei sind insbesondere die nahe liegenden und in diesem Kontext schon zahlreich bearbeiteten Fragestellungen zu nennen:

1. Was ist ein Graph?

2. Was ist ein Mustervorkommen und wie wird es gefunden?

3. Was bedeutet das „Ersetzen“ in der Graphersetzung?

Aus komplexitätstheoretischer Sicht ist die zweite Frage die einzig wichtige. Die anderen beiden Fragen betreffen die praktische Anwendbarkeit und die theoretische Fundierung.

In Kapitel 4 wird zu sehen sein, dass die alleinige Beantwortung der obigen drei Fragen noch kein stimmiges Bild ergibt. Weitere, in der Literatur eher selten beachtete Aspekte, wie Regel- und Passungsauswahl, sind für die praktische Nutzung unverzichtbar.

2.1 Elementare mathematische Strukturen

Wir benutzen die aussagenlogischen *Junktoren* und prädikatenlogischen *Quantoren*: $\vee$ (Disjunktion), $\wedge$ (Konjunktion), $\rightarrow$ (Implikation), $\leftrightarrow$ (Äquivalenz), $\neg$ (Negation), $\exists$ (Existenzquantor) und $\forall$ (Allquantor) mit der üblichen Bedeutung. Die beiden booleschen Wahrheitswerte bezeichnen wir mit $\bot$ (falsch) und $\top$ (wahr).

Pfeile werden auch noch in weiteren Situationen benötigt: Um metatheoretische Aussagen zu kennzeichnen, benutzen wir doppelt gestrichene Pfeile, wie $\Rightarrow$ und $\Leftrightarrow$. Für Regeln und Regelanwendungen in formalen Systemen (außer der Prädikatenlogik) benutzen wir $\longrightarrow$ und $\Longrightarrow$.

Definition 2.1 (Kardinalität) Die Anzahl der Elemente einer (endlichen) Menge A, also deren *Kardinalität*, schreiben wir als $|A|$.

Definition 2.2 (Funktion) Eine *Funktion* f von Menge A nach Menge B ist eine Teilmenge des Kartesischen Produkts

$$f \subseteq A \times B$$

wobei

$$\forall x \in A : |\{y|(x,y) \in f\}| \leq 1$$

Wir schreiben dann $f : A \to B$. Für die Festlegung der *Funktionswerte* y für gewisse *Argumente* x, also konkreter Tupel der Relation

$$(x,y) \in f$$

schreiben wir:

$$f : x \mapsto y \quad \text{oder} \quad f(x) := y$$

Bei der *Anwendung* $f(x)$ einer Funktion können zwei Situationen auftreten:

Definition 2.3 (Anwendung einer Funktion) Sei f eine Funktion und x ein beliebiges Objekt aus A:

1. $\exists y \in B : (x,y) \in f$. In diesem Fall ist der Funktionswert $f(x)$ eindeutig *definiert* und $f(x) = y$.
2. $\forall y \in B : (x,y) \notin f$. In diesem Fall existiert kein Funktionswert $f(x)$, wir sagen dann $f(x)$ ist *undefiniert*. Dies notieren wir mit $f(x) = \uparrow$.

Anschaulich gesprochen wird durch eine Funktion jedem $x \in A$ *höchstens* ein $y \in B$ zugeordnet. Klassische Funktionen weisen jedem $x \in A$ *genau* ein $y \in B$ zu, wohingegen unsere Definition auch zulässt, dass f stellenweise undefiniert ist. Die Notation $f(x) = \uparrow$ für undefinierte Funktionswerte suggeriert zwar, dass $\uparrow$ ein Wert ist; dies ist allerdings nicht so: Es gilt immer $\uparrow \notin B$, mithin kann $\uparrow$ als ein *unvergleichbares* „besonderes“ Objekt angesehen werden. Funktionen werden auch *Abbildungen* genannt. Es sei noch angemerkt, dass *mehrstellige* Funktionen, also Funktionen mit mehren Argumenten, durch Currying oder Erweiterung der Tupel modelliert werden können.

Definition 2.4 (Definitionsbereich und Wertebereich) Sei $f : A \to B$ eine Funktion.

$$\begin{aligned} \text{dom}(f) &= A & \text{def}(f) &= \{x \in A | \exists y \in B : f(x) = y\} \\ \text{cod}(f) &= B & \text{ran}(f) &= \{y \in B | \exists x \in A : f(x) = y\} \end{aligned}$$

Wir nennen $\text{dom}(f)$ die *Domäne*, $\text{cod}(f)$ die *Kodomäne*, $\text{def}(f)$ den *Definitionsbereich* und $\text{ran}(f)$ den *Wertebereich* (engl.: *range*) der Funktion f.

Gemäß Definition 2.4 gilt $\mathrm{def}(f) \subseteq \mathrm{dom}(f)$ und $\mathrm{ran}(f) \subseteq \mathrm{cod}(f)$. Aus der Anmerkungen zu Definition 2.3 folgt, dass für jede Funktion f stets $\uparrow \notin \mathrm{def}(f)$, $\uparrow \notin \mathrm{dom}(f)$, $\uparrow \notin \mathrm{ran}(f)$ und $\uparrow \notin \mathrm{cod}(f)$ erfüllt ist.

Definition 2.5 (Klassifikation von Funktionen) Sei $f : A \to B$ eine Funktion. Falls

- $\mathrm{def}(f) = \mathrm{dom}(f)$ ist, dann heißt die Funktion *total*.
- $\mathrm{def}(f) \subsetneq \mathrm{dom}(f)$ ist, dann heißt die Funktion *partiell*.
- $\mathrm{ran}(f) = \mathrm{cod}(f)$ ist, dann heißt die Funktion *surjektiv*.
- $\forall a_1, a_2 \in \mathrm{def}(f) : a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$ ist, dann heißt die Funktion *injektiv*.
- die Funktion total, injektiv und surjektiv ist, dann heißt sie *bijektiv*.

Die Injektivitätsbedingung ist wegen der Unvergleichbarkeit von $\uparrow$ (selbst bei partiellen Funktionen) mit der Aussage $\forall a_1, a_2 \in \mathrm{dom}(f) : a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$ äquivalent. Ferner ist bei endlichem Definitionsbereich eine Funktion genau dann injektiv wenn $|\,\mathrm{def}(f)| = |\,\mathrm{ran}(f)|$; im unendlichen Fall impliziert die Injektivität die *Gleichmächtigkeit* von Definitions- und Wertebereich, aber nicht umgekehrt. Falls die Funktion f bijektiv ist, dann existiert die *Umkehrfunktion* f^{-1} und ist eindeutig definiert. Es gilt $\forall x \in \mathrm{def}(f) = \mathrm{dom}(f) : f^{-1}(f(x)) = x$ und $\forall y \in \mathrm{ran}(f) = \mathrm{cod}(f) : f(f^{-1}(y)) = y$. Mithin ist f^{-1} ebenfalls bijektiv.

Definition 2.6 (Erweiterung einer Funktion) Eine Funktion g *erweitert* eine Funktion f genau dann, wenn

$$\mathrm{def}(f) \subseteq \mathrm{def}(g) \quad \wedge \quad \forall x \in \mathrm{def}(f) : f(x) = g(x)\ .$$

Wir schreiben dann $f \trianglelefteq g$. Falls sogar $\mathrm{def}(f) \subsetneq \mathrm{def}(g)$ gilt, dann sprechen wir von einer *echten Erweiterung* und schreiben $f \triangleleft g$.

Offenbar ist $\trianglelefteq$ eine *Halbordnung* und $\triangleleft$ eine *Striktordnung*. Ein kleinstes Element in der Klasse aller einstelligen Funktionen ist die *überall undefinierte Funktion* $\uparrow\uparrow$ (auch *leere Funktion* genannt), wobei für jedes beliebige x gilt, dass $\uparrow\uparrow(x) = \uparrow$.

Gleichwie (partielle) Funktionen erweitert werden können, so können totale (aber auch partielle) Funktionen eingeschränkt werden. Dabei wird der Definitionsbereich der Funktion gewissermaßen reduziert.

Definition 2.7 (Einschränkung einer Funktion) Die Funktion g ist die Einschränkung einer Funktion f auf die Menge A genau dann, wenn

$$g \trianglelefteq f \ \wedge\ \mathrm{def}(g) = \mathrm{def}(f) \cap A$$

Wir schreiben dann

$$g = f\,|_A$$

Die Einschränkung g einer Funktion f ist durch die Menge A eindeutig bestimmt; dies rechtfertigt auch die obige Schreibweise.

2.2 Graphen

Es gibt viele Möglichkeiten eine Struktur zu definieren, die man als *Graph* bezeichnen kann. Wir wählen hier endliche Trägermengen für Ecken und Kanten sowie Funktionen, die Kanten ihren Quell- und Zielecken zuordnen. Diese zunächst kompliziert anmutende Formalisierung ist, wie wir in Abschnitt 4.1 sehen werden, besonders günstig, um die dann benötigten *Multigraphen* zu erfassen.

Definition 2.8 (Graph) Ein (gerichteter) *Graph* ist ein 4-Tupel

$$G = (E, K, \mathrm{src}, \mathrm{tgt})$$

mit E und K endlichen disjunkten Mengen sowie zwei totalen Abbildungen

$$\mathrm{src} : K \to E \qquad \text{und} \qquad \mathrm{tgt} : K \to E\ .$$

Die Elemente von E nennen wir *Ecken* und die von K heißen *Kanten*[1]. Die Abbildungen src und tgt weisen den Kanten die Quelle und das Ziel zu. Die Kante beginnt an der Quelle und endet an ihrem Ziel. In diesem Sinn sind unsere Graphen gerichtet. Unsere Graphdefinition 2.8 schließt Strukturen, die gemeinhin als Multigraphen bezeichnet werden, mit ein. Wenn wir von Graphen sprechen, meinen wir in dieser Arbeit immer auch Multigraphen. Die „normalen“ einfachen Graphen definieren wir nun als Spezialfall.

Definition 2.9 (Einfacher Graph) Ein Graph heißt *einfacher Graph* genau dann, wenn

$$\forall k, k' \in K : [\mathrm{src}(k) = \mathrm{src}(k') \wedge \mathrm{tgt}(k) = \mathrm{tgt}(k')] \Rightarrow k = k'$$

Bei unseren Graphen sind auch immer *Schlingen*, also *reflexive Kanten* k mit $\mathrm{src}(k) = \mathrm{tgt}(k)$, zugelassen. Im Folgenden definieren wir einige hilfreiche Begriffe in Bezug auf Graphzusammenhang und Nachbarschaft in Graphen.

Definition 2.10 (Inzidente Ecken und Kanten) $\mathrm{inc}_E(e)$ bezeichnet die zu Ecke $e \in E$ *inzidenten Kanten*

$$\mathrm{inc}_E(e) := \{k \in K | \, \mathrm{src}(k) = e \vee \mathrm{tgt}(k) = e\}$$

$\mathrm{inc}_K(k)$ bezeichnet die zu Kante k *inzidenten Ecken*

$$\mathrm{inc}_K(k) := \{\mathrm{src}(k), \mathrm{tgt}(k)\}$$

[1]Man beachte hier die fast schon babylonische Begriffsverwirrung: Ecken heißen auch oft *Knoten* – im englischen *nodes* oder *vertices*. Kanten heißen manchmal auch *Linien*, *Verbindungen* oder – vor allem bei gerichteten Graphen – *Pfeile*, im englischen *edges* oder *lines* sowie im gerichteten Fall auch *arrows*, wobei *vertices* und *lines* jeweils lateinischen Ursprungs sind. Phonetisch bedingt werden besonders gerne Ecken und edges verwechselt.

Der *Grad* einer Ecke e ist $|\operatorname{inc}_E(e)|$. Entsprechend sind auch der *Ausgangsgrad* und der *Eingangsgrad*, als Anzahl der aus- oder eingehenden Kanten an einer Ecke, definiert. Für einfache gerichtete Graphen ist der Grad beschränkt und höchstens $2|E|-1$, da jede Ecke mit jeder anderen Ecke höchstens einmal in beide Richtungen verbunden sein kann. Für die Kardinalität der Kanten- und Eckenmenge einfacher Graphen gilt $|K| \leq |E|^2$. Hat der einfache Graph überdies keine Schlingen, gilt sogar: $|K| \leq |E|(|E|-1)$. Hingegen existiert für Multigraphen keine solche Schranke.

Definition 2.11 (Teilgraph) Graph $G' = (E', K', \operatorname{src}', \operatorname{tgt}')$ heißt genau dann *Teilgraph* von $G = (E, K, \operatorname{src}, \operatorname{tgt})$ wenn

$$E' \subseteq E \qquad \operatorname{src}' := \operatorname{src}|_{K' \to E'}$$
$$K' \subseteq K \qquad \operatorname{tgt}' := \operatorname{tgt}|_{K' \to E'}$$

Wir schreiben dann $G' \subseteq G$.

Ein Teilgraph ist durch die Wahl einer Ecken- und Kantenmenge E', K' eindeutig festgelegt. Bei der Wahl von E', K' ist man allerdings nicht vollkommen frei, denn die Festlegungen der Kantenmenge müssen *konsistent*[2] mit der Eckenmenge und der Struktur des zugrunde liegenden Graphen sein.

Definition 2.12 (Untergraph) $G' = (E', K', \operatorname{src}', \operatorname{tgt}')$ heißt genau dann *Untergraph* von $G = (E, K, \operatorname{src}, \operatorname{tgt})$ wenn $G' \subseteq G$ ist und ferner

$$\forall k \in K : [\operatorname{src}(k) \in E' \wedge \operatorname{tgt}(k) \in E'] \Rightarrow k \in K' \tag{2.1}$$

gilt.

Untergraphen heißen auch *induzierte Teilgraphen*. Anschaulich gesprochen wird der Teilgraph durch die Wahl einer Eckenmenge E' induziert, d. h. die Kantenmenge K' ist nicht frei wählbar, sondern muss gemäß Formel (2.1) bestimmt werden. Bei der Wahl von E' ist man hingegen vollkommen frei; jede Eckenmenge E' kann eindeutig zu einem Untergraphen erweitert werden.

Definition 2.13 (Gemeinsame Trägermengen) Seien A, B zwei Graphen mit den Eckenmengen E_A, E_B, welche Teilmengen von ein und derselben Menge $\mathcal{E}$ sind, also es gelte $E_A, E_B \subseteq \mathcal{E}$. Analoges gelte für die Kantenmengen. Dann sagt man: Die beiden Graphen haben *gemeinsame Trägermengen* (für Ecken und Kanten).

Gemeinsame Trägermengen sind eine Voraussetzung dafür, dass zwischen zwei Graphen überhaupt die Teil- oder Untergraphbeziehung bestehen kann. Offensichtlich implizieren sie allerdings keine dieser Beziehungen.

[2] Die Mengen E', K' sind genau dann konsistent gewählt, wenn $(E', K', \operatorname{src}', \operatorname{tgt}')$ ein Graph ist.

2.2.1 Morphismen

Definition 2.14 (Graphhomomorphismus)
Seien zwei Graphen $A = (E_A, K_A, \mathrm{src}_A, \mathrm{tgt}_A)$ und $B = (E_B, K_B, \mathrm{src}_B, \mathrm{tgt}_B)$ gegeben. Ein *Graphhomomorphismus* von Graph A nach Graph B ist ein Paar von Abbildungen $m = (m_E, m_K)$ wobei

$$m_E : E_A \to E_B$$
$$m_K : K_A \to K_B$$

mit

$$\forall k \in \mathrm{def}(m_K) : m_E(\mathrm{src}_A(k)) = \mathrm{src}_B(m_K(k))$$
$$\forall k \in \mathrm{def}(m_K) : m_E(\mathrm{tgt}_A(k)) = \mathrm{tgt}_B(m_K(k))$$

Wir schreiben dann $m : A \to B$.

Graphhomomorphismen sind also *strukturerhaltende „Abbildungen“* zwischen zwei Graphen.

Definition 2.15 (Definitions- und Wertebereich von Graphhomomorphismen)
Sei $m = (m_E, m_K) : A \to B$ ein Graphhomomorphismus. Dann definieren wir

$$\mathrm{dom}(m) = A \qquad \mathrm{def}(m) = (D_E, D_K, \mathrm{src}|_{D_K \to D_E}, \mathrm{tgt}|_{D_K \to D_E})$$
$$\mathrm{cod}(m) = B \qquad \mathrm{ran}(m) = (R_E, R_K, \mathrm{src}|_{R_K \to R_E}, \mathrm{tgt}|_{R_K \to R_E})$$

wobei $D_E := \mathrm{def}(m_E), D_K := \mathrm{def}(m_K), R_E := \mathrm{ran}(m_E)$ und $R_K := \mathrm{ran}(m_K)$ ist.

Da die Definitionen von $\mathrm{def}(m)$ und $\mathrm{ran}(m)$ die Eigenschaften der strukturerhaltenden Abbildung m weitertragen, sind die Definitions- und Wertebereiche von Graphhomomorphismen immer wohldefinierte Graphen. Ferner gelten folgende Teilgraphbeziehungen: $\mathrm{def}(m) \subseteq \mathrm{dom}(m)$ und $\mathrm{ran}(m) \subseteq \mathrm{cod}(m)$.

Definition 2.16 (Erweiterung eines Graphhomomorphismus)
Ein Graphhomomorphismus m' erweitert einen Graphhomomorphismus m genau dann, wenn

$$\mathrm{def}(m) \subseteq \mathrm{def}(m') \quad \wedge \quad \forall e \in \mathrm{def}(m_E) : m_E(e) = m'_E(e)$$
$$\wedge \quad \forall k \in \mathrm{def}(m_K) : m_K(k) = m'_K(k)$$

Analog zu Definition 2.6 benutzen wir das Symbol $\trianglelefteq$ für Erweiterungen und $\triangleleft$ für echte Erweiterungen.

Definition 2.17 (Klassifikation von Graphhomomorphismen) Für je zwei Graphen A und B definieren wir, mittels Eigenschaften für die Ecken- und Kantenabbildungen, verschiedene Klassen von Homomorphismen $m = (m_E, m_K)$ von A nach B, wobei $\mathbb{T}$ für totale und $\mathbb{P}$ für partielle Graphhomomorphismen steht und $X \in \{\mathbb{T}, \mathbb{P}\}$ sei:

$$(m_E, m_K) \in \begin{cases} \mathbb{T}(A,B) & :\Longleftrightarrow \quad m_E, m_K \text{ total} \\ \mathbb{P}(A,B) & :\Longleftrightarrow \quad m_E, m_K \text{ partiell} \\ X_{\mathbb{INJ}}(A,B) & :\Longleftrightarrow \quad m \in X \wedge m_E, m_K \text{ injektiv} \\ X_{\mathbb{SUR}}(A,B) & :\Longleftrightarrow \quad m \in X \wedge m_E, m_K \text{ surjektiv} \\ \mathbb{T}_{\mathbb{ISO}}(A,B) & :\Longleftrightarrow \quad m \in \mathbb{T}(A,B) \wedge m_E, m_K \text{ bijektiv} \end{cases}$$

Für die obigen Mengen von Morphismen zwischen zwei Graphen haben sich auch spezielle Begriffe herausgebildet. Allgemeine Morphismen ($\mathbb{P}(A,B)$) heißen auch *Graphhomomorphismen*. Die Elemente von $\mathbb{T}_{\mathbb{INJ}}(A,B)$ und $\mathbb{P}_{\mathbb{INJ}}(A,B)$ werden *Graphmonomorphismen* genannt und $\mathbb{T}_{\mathbb{ISO}}(A,B)$ ist die Klasse der *Graphisomorphismen* von A nach B. Wenn der Kontext klar ist, kann der Präfix „Graph“ auch weggelassen werden.

Definition 2.18 (Graphisomorphieproblem) Gegeben sind zwei Graphen G_1, G_2. Die Frage ist, ob es Isomorphismen zwischen diesen Graphen gibt, also ob $\mathbb{T}_{\mathbb{ISO}}(G_1, G_2) \neq \emptyset$ ist. Man sagt dann, G_1 ist *isomorph* zu G_2, oder in Zeichen: $G_1 \approx G_2$.

Definition 2.19 (Teilgraphisomorphieproblem) Gegeben sind zwei Graphen G_1, G_2. Die Frage ist, ob G_1 isomorph zu einem Teilgraphen von G_2 ist, also

$$\exists G_2' \subseteq G_2 : G_1 \approx G_2'$$

zutrifft. Wir schreiben dann $G_1 \lessapprox G_2$.

Es ist bekannt, dass das Teilgraphisomorphieproblem NP-vollständig ist. Vom spezielleren Graphisomorphieproblem ist nur bewiesen, dass es in NP liegt. Ein Beweis für die NP-Vollständigkeit des Graphisomorphieproblems ist nicht zu erwarten, da sonst die polynomielle Hierarchie zusammenbrechen würde, was gemeinhin als unwahrscheinlich gilt [GJ90].

2.2.2 Klassifikation

Die qualitative Unterteilung von Graphen in dünnbesetzte, dichtbesetzte und normalbesetzte Graphen ist in der Literatur nicht ganz eindeutig, darum legen wir hier die Begriffe fest.

Definition 2.20 (Dichte eines Graphen) Sei $G = (E, K, \mathrm{src}, \mathrm{tgt})$ ein nicht leerer Graph. Wir definieren die *Dichte* von G als

$$D_G = \frac{|K|}{|E|^2}$$

Für einfache Graphen ist $0 \leq D_G \leq 1$. Falls G ein Multigraph ist, existiert keine allgemeine obere Schranke für D_G.

Definition 2.21 (Dünn- und dichtbesetzte Graphen) Eine Familie von Graphen mit zunehmender Kardinalität der Eckenmenge heißt dünnbesetzt falls $D_G \in O\left(\frac{1}{|E|}\right)$. Falls $D_G \in \Omega(1)$ heißt sie dichtbesetzt.

Wir benutzen in obiger Definition die üblichen Landau-Symbole, um das asymptotische Verhalten kompakt anzugeben.

Definition 2.22 (Gradbeschränkter Graph) Ein Graph $G = (E, K, \mathrm{src}, \mathrm{tgt})$ heisst *gradbeschränkt* genau dann, wenn

$$\forall e \in E : |inc_E(e)| \leq \delta$$

wobei δ der maximale Grad ist.

Eine Familie gradbeschränkter Graphen (mit Grad δ) ist dünnbesetzt, da $|K| \leq \delta E$ und damit $D_G = \frac{|K|}{|E|^2} \leq \frac{\delta}{|E|}$. Mit obiger Folgerung wird klar, dass *Binärbäume* dünnbesetzte (weil Grad-3-beschränkte) Graphen sind.

2.3 Graphmustersuche

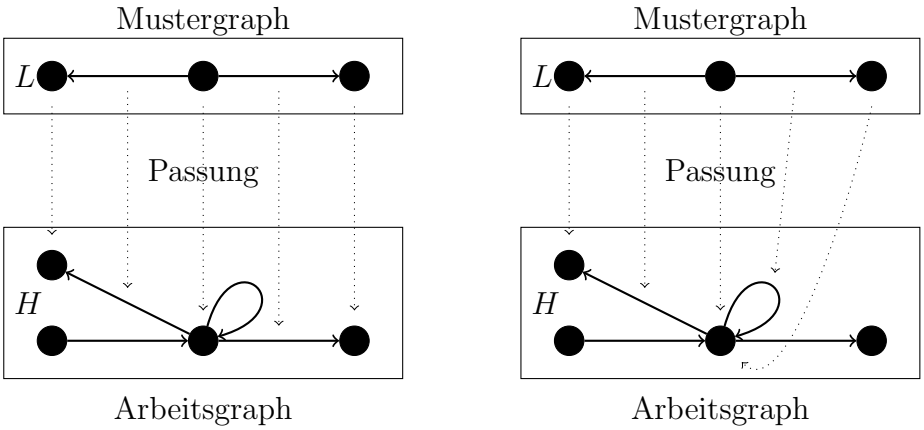
Wir betrachten hier die Graphmustersuche[3] als eigenständige Fragestellung, also zunächst vollkommen losgelöst von der Graphersetzung. Wie uns die nächsten Definitionen und Sätze zeigen, hängt Graphmustersuche eng mit dem Teilgraphisomorphieproblem zusammen. Die prinzipielle Idee der Graphmustersuche ist, dass ein *Mustergraph* L in einem *Arbeitsgraphen* H zu suchen ist, und die Elemente von L mit denen von H in Verbindung zu setzen sind.

Definition 2.23 (Passung) Seien L und H Graphen. Seien ferner m und m' gegeben durch

$$m \in \mathbb{T}_{\mathbb{INJ}}(L, H) \tag{2.2}$$
$$m' \in \mathbb{T}(L, H) \tag{2.3}$$

Dann heißt m monomorphe *Passung* (engl.: *match*) und m' (homomorphe) Passung.

[3] engl.: *subgraph matching*



(a) Monomorphe Passung (b) Homomorphe Passung

Abbildung 2.1: Passungen im Beispiel

Der Unterschied zwischen einer monomorphen und homomorphen Passung wird in Abbildung 2.1 illustriert. Der Zusatz „homomorph“ für eine Passung kann auch weggelassen, weil dies der Normalfall ist. Es sei angemerkt, dass $\mathbb{T}_{\mathbb{INJ}}(L, H) \subseteq \mathbb{T}(L, H)$ gilt, und somit jede monomorphe Passung per definitionem auch eine homomorphe Passung ist, aber nicht notwendigerweise umgekehrt. Ferner gilt für jede Passung m auch $m \in \mathbb{T}(L, H) \subseteq \mathbb{P}(L, H)$. Allerdings muss jede Passung total sein, da sie *jedes* Element des Mustergraphen mit dem Arbeitsgraphen in Beziehung zu setzen hat.

Satz 2.24 (Teilgraph und Passung) Seien L, H Graphen. Wenn $L \subseteq H$ gilt, dann existiert eine monomorphe Passung von L nach H.

Beweis: Als monomorphe Passung m von L nach H kann die (partielle) *Identität* $m = (\mathbb{1}, \mathbb{1})$ gewählt werden, da so keine Ecke oder Kante von L anders als identisch abgebildet wird, ist m tatsächlich eine monomorphe Passung. ■

Der obige Satz ist so interpretierbar, dass eine Passung die Verallgemeinerung des Teilgraphbegriffes mit nicht gemeinsamen Trägermengen ist; überdies ist selbst bei gemeinsamen Trägermengen auch ein „umnummerieren“ der Ecken und Kanten möglich. Eine Umkehrung von Satz 2.24 ist nicht möglich, da zum Beispiel die Graphen G und G', gegeben durch,

$$G = (\{e_1, e_2\}, \{k_1\}, \mathrm{src} : \{k_1 \mapsto e_1\}, \mathrm{tgt} : \{k_1 \mapsto e_2\})$$

$$G' = (\{e_1, e_2\}, \{k_1\}, \mathrm{src}' : \{k_1 \mapsto e_2\}, \mathrm{tgt}' : \{k_1 \mapsto e_1\})$$

wegen $\forall k \in K : \mathrm{src}(k) \neq \mathrm{src}'(k)$ und $\forall k \in K : \mathrm{tgt}(k) \neq \mathrm{tgt}'(k)$, nicht in Teilgraphbeziehung zueinander stehen. Jedoch existiert eine monomorphe Passung zwischen G und G', nämlich $m : (\{e_1 \mapsto e_2, e_2 \mapsto e_1\}, \{k_1 \mapsto k_1\})$. Die Graphen sind sogar isomorph, es gilt nämlich $m \in \mathbb{T}_{\mathbb{ISO}}(G, G')$.

Satz 2.25 (Teilgraphisomorphieproblem und Passung) $L \precsim H$ gilt genau dann, wenn eine monomorphe Passung m von L nach H existiert.

Beweis: Der Satz folgt in der einen Richtung unmittelbar aus den entsprechenden Definitionen 2.19, 2.18 und 2.23:

$$\begin{aligned} L \precsim H &:\Longleftrightarrow \exists H' \subseteq H : L \approx H' \\ &:\Longleftrightarrow \exists H' \subseteq H : \mathbb{T}_{\mathbb{ISO}}(L, H') \neq \emptyset \\ &\Longleftrightarrow \exists H' \subseteq H : \exists m \in \mathbb{T}_{\mathbb{ISO}}(L, H') \\ &\Longrightarrow \exists m \in \mathbb{T}_{\mathbb{INJ}}(L, H) \end{aligned}$$

Die Rückrichtung, also $\exists m \in \mathbb{T}_{\mathbb{INJ}}(L, H) \Longrightarrow \exists H' \subseteq H : \exists m \in \mathbb{T}_{\mathbb{ISO}}(L, H')$ gilt, da jeder Morphismus auf seinem Bildbereich surjektiv ist und somit $\mathrm{ran}(m)$ als H' gewählt werden kann. ∎

Dieser Satz ist eine oft stillschweigend vorausgesetzte Grundlage der Graphersetzung, stellt er doch die Verbindung zwischen Teilgraphisomorphieproblem und Passung her.

Komplexität

Für unsere Zwecke – das Finden von Passungen – sind wir an Teilgraphisomorphie bzw. Teilgraphhomomorphie interessiert. Diese Probleme sind, wie fast alle anderen Suchprobleme, in allgemeinen Graphen NP-vollständig (vgl. Abschnitt 3.1.1). Entsprechend ist die Laufzeit aller bekannten Algorithmen zur Berechnung einer Passung in $O(|L| \cdot |H|^{|L|})$, wobei $|\cdot|$ die Summe der Anzahl der Ecken und Kanten eines Graphen ist. Das Teilgraphisomorphieproblem ist *nur* dann NP-vollständig, wenn *die Mustergraphen unbeschränkte Größe* haben. Diese Aussage ist insbesondere an obiger Formel leicht abzulesen.

Hingegen existieren für Mustergraphen konstanter Größe polynomielle Algorithmen, die allerdings im Allgemeinen große Polynomgrade aufweisen. Dies scheint zunächst eine gute Ausgangslage zu sein, da in der Praxis oft die Muster festliegen und nur die Arbeitsgraphen variieren. Jedoch ist selbst für z. B. 10 Mustergraphelemente und recht kleine Arbeitsgraphen eine algorithmische Mustersuche bei einer Laufzeit von $O(|H|^{10})$ nicht durchführbar. Im Allgemeinen führt das dazu, dass nur sehr kleine Mustergraphen und kleine Arbeitsgraphen bearbeitbar sind. Wir können diese Situation nicht grundlegend ändern[4]. Dennoch können wir – wie bei jedem NP-Problem – Problemklassen finden, die effizient lösbar sind. In Kapitel 5 diskutieren wir, wie dies geschehen kann, und weisen nach, dass die so erfasste Teilmenge der Instanzen von Teilgraphisomorphieproblemen von großer praktischer Relevanz ist.

2.4 Kategorientheorie

Wir geben hier eine kurze Einführung in die für uns wichtigen Aspekte der Kategorientheorie. Die Kategorientheorie ist in der ersten Hälfte des 20. Jahrhunderts

[4]Zumindest solange niemand P ≠ NP widerlegt hat.

als Verallgemeinerung aus der algebraischen Topologie und der universellen Algebra hervorgegangen. Als Pionierarbeiten[5] sind hierbei besonders die Aufsätze von Eilenberg und Mac Lane zu nennen [EM42, EM45]. Wir benutzen hier einige grundlegende Definitionen und Sätze aus dem Buch von Pierce [Pie91]. Weitergehende Formulierungen zur kanonischen Konstruktion des *Pushouts* orientieren sich an dem vielversprechenden, aber noch unveröffentlichten Buchmanuskript „Graph Transformations – An Introduction to the Categorical Approach“ von Schneider [Sch07a].

Definition 2.26 (Kategorie) Eine Kategorie $\mathbf{C} = (O, P, \circ)$ ist gegeben durch:

(C1) Eine Ansammlung von *Objekten*.

(C2) Eine Ansammlung von *Pfeilen* (oft Morphismen genannt).

(C3) Zwei *Operationen* dom und cod, welche jedem Pfeil f je ein Objekt $\mathrm{dom}(f)$ und $\mathrm{cod}(f)$ zuordnen.

(C4) Einen assoziativen *Verknüpfungsoperator* $\circ$, der für jedes Paar von Pfeilen f, g mit $\mathrm{cod}(f) = \mathrm{dom}(g)$ einen verknüpften Pfeil $\mathrm{dom}(f) \xrightarrow{g \circ f} \mathrm{cod}(g)$ erzeugt.

(C5) Es gibt *Identitätspfeile* $A \xrightarrow{\mathrm{id}_A} A$ für jedes Objekt A, bezogen auf den Verknüpfungsoperator.

Obwohl wir für dom und cod schon in Definition 2.4 eine Bedeutung verabredet haben, verwenden wir die Operationen für die Kategorientheorie in einem neuen, wenn auch verwandten, Sinn. Zu beachten ist, dass obwohl diese Definition sehr stark an normale Mengen und Abbildungen auf diesen Mengen erinnert, dies eine abstraktere Struktur ist, die aber sehr wohl als spezielle Instanz auch den Fall der „normalen mathematischen Abbildung auf Mengen“ beinhaltet. Es sei auch daran erinnert, dass die Ansammlungen von Objekten nicht nur Mengen, sondern auch *echte Klassen* oder andere mathematische Strukturen sein können. Ferner bleiben noch folgende Anmerkungen: Für einen Pfeil f heißt $\mathrm{dom}(f)$ die *Domäne*, während $\mathrm{cod}(f)$ die sogenannte *Kodomäne* ist. Wir schreiben dann $A \xrightarrow{f} B$ mit $A = \mathrm{dom}(f)$ und $B = \mathrm{cod}(f)$. Der Kompositionsoperator ist *assoziativ* genau dann, wenn für jede drei Pfeile $A \xrightarrow{f} B$, $B \xrightarrow{g} C$ und $C \xrightarrow{h} D$ gilt[6], dass

$$h \circ (g \circ f) = (h \circ g) \circ f \ .$$

Mit Identitätspfeilen gemäß des Verknüpfungsoperators ist gemeint, dass für jeden Pfeil $A \xrightarrow{f} B$ auch zwei Identitätspfeile id_A und id_B existieren, für die gilt, dass

$$f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f$$

ist.

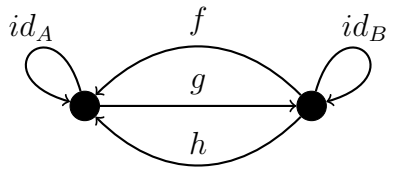
Wir geben hier nun die Definition der Kategorie **Graph**, der für unsere Arbeit wichtigsten Kategorie, nämlich der Kategorie der Graphen und Homomorphismen auf Graphen, nach Rydeheard an [Ryd85].

[5] Dem interessierten Leser sei hingegen das Buch von Pierce [Pie91] als Einstig empfohlen.
[6] A, B, C und D müssen nicht notwendigerweise verschieden sein.

Definition 2.27 (Kategorie **Graph**) Die Kategorie **Graph** hat als Ansammlung von Objekten die Menge aller Graphen, als Ansammlung von Pfeilen die Menge aller Homomorphismen auf diesen Graphen.

Diese Kategorie verbindet offenbar alle Graphen durch Pfeile zwischen denen (partielle) Homomorphismen bestehen. In einem gewissen Sinn sind also in der Kategorie **Graph** alle Passungen zwischen allen Graphen als Pfeile eingezeichnet. Überdies sind aber auch alle unvollständigen Passungen, also partielle Homomorphismen, als Pfeil repräsentiert. Nicht nur die strukturerhaltenden Abbildungen zwischen Graphen induzieren eine Kategorie, sondern jeder Graph selbst bildet seine eigene kleine Kategorie: Die Objekte sind die Ecken des Graphen und die Pfeile sind die *Pfade*[7] innerhalb des Graphen. Der Verknüpfungsoperator entspricht demnach dem Aneinanderhängen von Pfaden. Diese Kategorie heißt die *freie Kategorie* über dem entsprechenden Graphen.

Man könnte nun glauben, dass auch die scheinbar direkteste Umsetzung eines Graphen in die Welt der Kategorien eine wohldefinierte Kategorie ergibt. Seien also die Objekte die Ecken und die Pfeile die Kanten eines Graphen, der Verknüpfungsoperator bestimme die transitive Kante zweier Kanten. Dies ist allerdings mitnichten eine Kategorie, wie uns folgendes Beispiel zeigt. Betrachten wir also den nachstehenden Graphen



so ergibt sich, dass zwar die Identitätspfeile existieren

$$g \circ \mathrm{id}_A = g = \mathrm{id}_B \circ g,\ h \circ \mathrm{id}_B = h = \mathrm{id}_A \circ h,\ f \circ \mathrm{id}_B = f = \mathrm{id}_A \circ f$$

aber:

$$(h \circ g) \circ f = \mathrm{id}_A \circ f = f \neq h = h \circ \mathrm{id}_B = h \circ (g \circ f)\ .$$

Damit ist der Verknüpfungsoperator nicht assoziativ und es liegt keine Kategorie vor.

Nun steigen wir tiefer in die Mechanik der Kategorientheorie ein und schaffen das nötige Rüstzeug, um beweisen zu können, dass das sogenannte *Pushout* (siehe Definition 2.30) auf Graphen immer existiert und kanonisch konstruiert werden kann. Letzteres bedeutet, dass diese Aufgabe algorithmisch lösbar ist. Hierzu wollen wir *Koprodukt*[8], *Koegalisator*[9] und Pushout betrachten.

[7] Pfade sind (durch Kanten geeigneter Richtung) zusammenhängende Folgen von Knoten in gerichteten Graphen.
[8] engl.: *coproduct*
[9] engl.: *coequalizer*

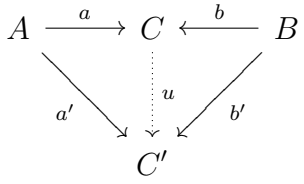


Abbildung 2.2: Bedingungen des Koprodukts als Diagramm

Definition 2.28 (Koprodukt) Das *Koprodukt* eines Paares von Objekten (A, B) ist ein Tripel (a, b, C) wobei C ein Objekt ist und $A \xrightarrow{a} C, B \xrightarrow{b} C$ Morphismen sind, sodass

$$\forall C' \, \forall (A \xrightarrow{a'} C') \, \forall (B \xrightarrow{b'} C') \, \exists_1 (C \xrightarrow{u} C') : \left[a' = u \circ a \wedge b' = u \circ b\right].$$

Die Bedingungen des Koprodukts sind im Diagramm, das in Abbildung 2.2 dargestellt wird, veranschaulicht. Um der Vorstellungskraft nachzuhelfen, geben wir zwei Beispiele für konkrete Instanzen des Koprodukts an. Es entspricht auf Mengen der *disjunkten Vereinigung* und auf *abelschen Gruppen* der *direkten Summe*. Man sagt die Kategorie $\mathbf{C}$ hat Koprodukte, wenn es für jedes Paar von Objekten von $\mathbf{C}$ ein Koprodukt gibt. Mit $\exists_1 C \xrightarrow{u} C'$ ist gemeint, dass bis auf Isomorphie *genau ein* solches u existiert. Diese Eindeutigkeit sollte nicht mit der Eindeutigkeit von C oder a und b verwechselt werden – diese ist nämlich im Allgemeinen gerade nicht gegeben.

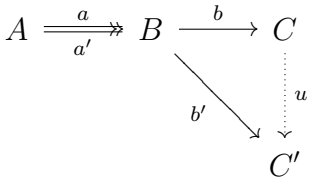


Abbildung 2.3: Bedingungen des Koegalisators als Diagramm

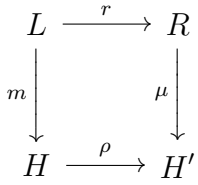
Definition 2.29 (Koegalisator) Ein Pfeil $B \xrightarrow{b} C$ wird *Koegalisator* eines Paares paralleler Pfeile a und a' genannt, genau dann, wenn gilt:

1. $b \circ a = b \circ a'$
2. $\forall C' \, \forall (B \xrightarrow{b'} C') : \left[b' \circ a = b' \circ a' \Rightarrow \exists_1 (C \xrightarrow{u} C') : b' = u \circ b\right]$

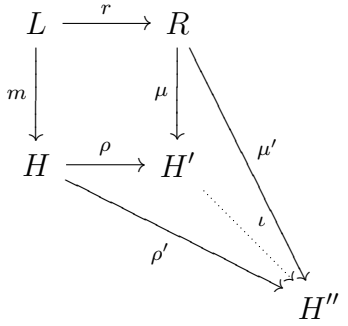
Abbildung 2.3 stellt die Bedingungen an den Koegalisator als Diagramm dar. Man kann zeigen, dass der Koegalisator in jeder Kategorie ein Epimorphismus[10] ist. Man

[10] Nicht jeder Epimorphismus in der Kategorientheorie ist auch ein Epimorphismus in der Algebra, also eine surjektiver Homomorphismus; die Umkehrung ist allerdings wahr. Darum schlägt auch Mac Lane vor, die Epimorphismen in der Kategorientheorie Epic zu nennen.

sagt die Kategorie **C** hat Koegalisatoren, wenn es für jedes Paar von parallelen Morphismen von **C** einen Koegalisator gibt. Wie auch das Koprodukt, so ist auch der Koegalisator b bis auf Isomorphie eindeutig bestimmt.



(a) Kommutierendes Diagramm: Existenz H'



(b) Eindeutigkeit von ι: Universalität H'

Abbildung 2.4: Definition des Pushouts (H', ρ, μ) durch zwei Eigenschaften

Definition 2.30 (Pushout) Seien m, r Pfeile und L ein Objekt einer Kategorie. Ein *Pushout* von (L, m, r) ist definiert als Tripel (H', ρ, μ) für welches das Diagramm 2.4(a) kommutiert und H' *universell* ist.

Die beiden charakterisierenden Eigenschaften eines Pushouts, Kommutativität und Universalität, bedeuten im Einzelnen:

Definition 2.31 (Kommutativität) $\rho \circ m = \mu \circ r$

Definition 2.32 (Universalität) $\forall \mu' : R \to H', \forall \rho' : H \to H'$ mit $\rho' \circ m = \mu' \circ r$ gibt es genau einen Morphismus $\iota : H \to H'$, so dass $\mu' = \iota \circ \mu$ und $\rho' = \iota \circ \rho$ ist.

Mit universell ist also gemeint, dass in Abbildung 2.4(b) für jedes andere Tripel (H'', ρ', μ') der Morphismus ι existiert und eindeutig ist. Im Allgemeinen kann man sich ein Pushout von (L, m, r) als eine Art disjunkter Vereinigung von H und R mit sukzessivem Verkleben an den Punkten, die durch m und r vorgegeben werden, vorstellen. Die Universalität (siehe Abbildung 2.4(b)) des Pushout-Objekts H' stellt bezogen auf Graphen zweierlei sicher: Erstens ist H' bezogen auf H und R gewissermaßen „komplett“. Dies bedeutet, dass es alle Entsprechungen von Elementen aus H und R enthält, bis auf die Änderungen, die von $r \in \mathbb{P}(L, R)$ gefordert werden. Zweitens ist H' auch „minimal“, sodass es keine im vorherigen Sinne überflüssigen Elemente enthält.

Im Folgenden soll geklärt werden, warum das so ist. Nehmen wir an, H' würde zu mindestens *einem* Element, das R enthält, keine Entsprechung enthalten. Dann würde es einen Homomorphismus μ' nach H'' geben, der dieses Element abbildet.

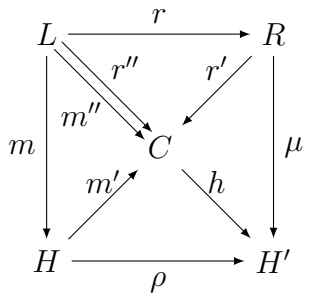


Abbildung 2.5: Kanonische Konstruktion des Pushouts

Damit würde es aber unter Umständen mehrere nicht isomorphe Morphismen von H' nach H'' geben, die das Pushout-Diagramm kommutieren lassen (H'' enthält mehr Elemente als H', sodass ι verschiedene Bildbereiche hat, also mal auf das eine Element, mal auf ein anderes abbildet). Würde hingegen H' zu viele Elemente enthalten, also zum Beispiel solche, die nicht aus H oder R induziert sind, so würde es kleinere H'' geben. Dann würde es unter Umständen mehrere nicht isomorphe ι geben, die unterschiedliche Definitionsbereiche in H' haben, aber jedes Mal nach H'' abbilden. Damit wird erzwungen, dass das Pushout-Objekt H' und mithin die Morphismen μ, ρ an der Grenze von „Allgemeinheit“ und „Spezifität“ liegen. Es wird also nichts Wichtiges weggelassen, aber auch nichts Überflüssiges hinzugefügt.

Nachdem wir der Intuition die Steigbügel gehalten haben, folgt nun ein Satz, der zeigt, dass dieses Pushout-Objekt für Graphen nicht nur einfach existiert, sondern auch immer nach dem gleichen Verfahren konstruiert werden kann.

Satz 2.33 (Kanonische Konstruktion) Eine Kategorie mit Koprodukten und Koegalisatoren hat auch Pushouts.

Beweis: Sei uns ein *Objekt* L und zwei *Pfeile* $L \xrightarrow{r} R$, $L \xrightarrow{m} H$ gegeben. Dann konstruieren wir durch folgende Schritte ein Pushout-Diagramm wie es in Abbildung 2.5 zu sehen ist.

1. Wir konstruieren das Koprodukt von R und H, also das Tripel $(R \xrightarrow{r'} C, H \xrightarrow{m'} C, C)$. Dieses Koprodukt existiert nach Voraussetzung.
2. Wir setzen $r'' := r' \circ r$ und $m'' := m' \circ m$.
3. Wir konstruieren den Koegalisator des Paares paralleler Pfeile (r'', m''): Dies ist der Pfeil $C \xrightarrow{h} H$, der ebenfalls nach Voraussetzung existiert.
4. Schließlich setzen wir $\rho := h \circ m'$ und $\mu := h \circ r$.

Per definitionem kommutiert das äußere Diagramm, also gilt $\rho \circ m = \mu \circ r$. Somit bleibt noch zu zeigen, dass H' universell ist. Dies ist möglich, indem wir zunächst ein beliebiges kommutierendes Diagramm $\rho' \circ m = \mu' \circ r$ annehmen und dann zeigen, dass sich ρ', μ' durch ein eindeutiges ι faktorisieren lassen, sodass $\rho' = \iota \circ \rho$ und $\mu' = \iota \circ \mu$ gilt. Diesen Beweisrest lassen wir hier offen, da

er nicht zu der von uns gewünschten konstruktiven Herstellung des Pushout-Objekts H' beiträgt. Es sei noch angemerkt, dass wir den eher technischen Rest des Beweises mittels der Koprodukt-Eigenschaften von C und den entsprechenden Koegalisator-Eigenschaften vervollständigen können (vgl. etwa [Sch07a]). ■

Wenn eine Kategorie unter dem *Koprodukt* seiner Objekte und dem *Koegalisator* seiner Pfeile geschlossen ist, dann existiert das Pushout immer und kann kanonisch berechnet werden. Die Kategorie **Graph** erfüllt die vorgenannten Anforderungen. Also können wir das kategorientheoretisch charakterisierte Pushout-Objekt H' für alle Graphen einheitlich bestimmen.

2.5 Graphersetzung

Es gibt viele Ansätze, die Bedeutung von *Graphersetzung* entweder formal oder ad hoc zu fassen. Es ist offenkundig nicht objektiv entscheidbar, welcher dieser Ansätze für ein allgemeines Graphersetzungssystem der geeignetste ist, insbesondere da diese Entscheidung bis zu einem gewissen Grad eine Geschmacksfrage ist. In der Literatur wird das Thema Graphersetzung – im Gegensatz beispielsweise zur Graphmustersuche – intensiver als alle anderen hier relevanten Grundlagen behandelt. Nichtsdestotrotz ist es keine komplexitätstheoretisch anspruchsvolle Fragestellung: Fast alle bekannten Ersetzungsformalismen sind linear in der Zahl der Graphelemente, die von der Passung bzw. Änderungen erfasst sind. Davon abgesehen sind die Algorithmen, die die eigentliche Ersetzung durchführen, eher trivial[11].

Wenn wir uns vor Augen führen, dass wir eine möglichst breite und damit auch ausdrucksstarke Methodik entwickeln wollen, können wir allerdings einige Anforderungen an den einzusetzenden Formalismus der Graphersetzung stellen: Erstens sollen Ecken auffindbar sein, ohne notwendigerweise ihren vollständigen Kontext angeben zu müssen. Zweitens soll – ähnlich der erstgenannten Situation – eine gefundene Ecke auch dann löschbar sein, wenn nicht alle ihre inzidenten Kanten vom Muster erfasst wurden. In diesem Fall soll das Löschen der baumelnden Kanten priorisiert werden. Drittens soll die *Anwendung* einer Regelmenge, die solange iteriert wird, bis keine Regel mehr anwendbar ist, einen turingvollständigen Formalismus darstellen. Schließlich sollen Typwechsel, Attributberechnungen und reichhaltige strukturelle Anwendungsbedingungen formulierbar sein.

Diese Anforderungen schließen viele Ansätze aus, lassen allerdings immer noch eine gewisse Wahl. Wir haben uns für den algebraischen *Single-Pushout Ansatz* (SPO) als Grundlage unserer Methodik entschieden[12]. Ein Vorteil der algebraischen Ansätze ist, dass dieselbe elegante Theorie benutzt werden kann, um von den Regeln bis hin zu deren Anwendung alles zu spezifizieren. Aufgrund des Rückgriffs auf grundlegende Definitionen der Kategorientheorie benötigen wir nichts als (partielle) Graphhomomorphismen, um das Finden von Graphmustern wie auch die Graphersetzung zu formalisieren. Im Folgenden geben wir einen Überblick über den SPO-Ansatz.

[11] Zum Beispiel ist Edmonds Algorithmus [Edm67] zur Berechnung des minimalen spannenden Arboreszenten, den wir bei der Mustersuche verwenden (siehe Abschnitt 5.3) und der dort nur einen kleinen Teilschritt bei der Vorverarbeitung der Mustersuche darstellt, viel komplizierter als die gesamte Implementierung der SPO- oder DPO-Ersetzungsformalismen.

[12] SPO steht für Single-Pushout Ansatz, da er, wie in Abbildung 2.6 zu sehen ist, mittels *eines* Pushouts definiert ist; im Gegensatz zum *Double-Pushout Ansatz* (DPO), der *zwei* Pushouts benötigt um das Transformationsergebnis zu charakterisieren.

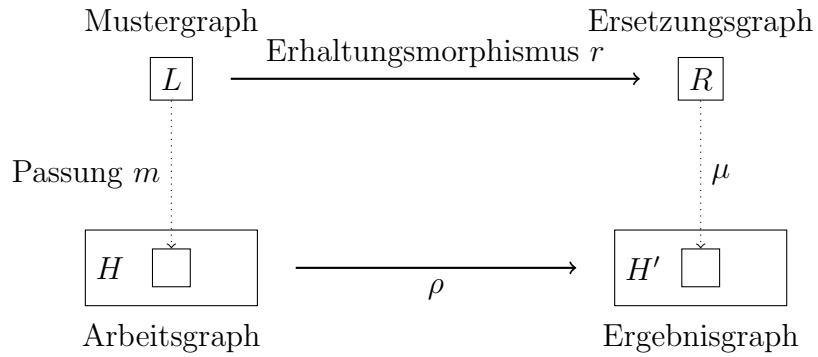


Abbildung 2.6: Darstellung einer SPO-Graphersetzungsregel und deren Anwendung

SPO-Ansatz

Beim SPO-Ansatz besteht eine Graphersetzungsregel aus einem Graphen auf der linken Seite L, einem Graphen auf der rechten Seite R und einem partiellen Graphmonomorphismus[13] r zwischen L und R. Eine solche Regel p wird geschrieben als:

$$p : L \xrightarrow{r} R$$

Eine *Anwendung* der Regel p auf den Arbeitsgraphen H nennen wir *direkte Ableitung*. Ebendiese Anwendung benötigt einen totalen Graphhomomorphismus m zwischen L und H, den sogenannten *Passungsmorphismus*. Die direkte Ableitung mündet im *Ergebnisgraphen* H', siehe das Diagramm in Abbildung 2.6. Der Passungsmorphismus ordnet jeder Ecke und Kante x in L eine Entsprechung $m(x)$ in H zu. Es ist zu beachten, dass hier die Injektivität von m nicht gefordert wird.

Der *Erhaltungsmorphismus* r bestimmt, was mit den $m(x)$ geschieht: Die Bilder von m bezüglich der Elemente in L, die keine Bilder unter r besitzen, werden gelöscht. Hingegen werden Elemente $m(x)$, die Bilder $r(x) \in R$ besitzen, erhalten. Die Entsprechungen der Elemente in R, die von r nicht *überstrichen*[14] werden, sind in den Ergebnisgraphen H' als neue Graphelemente einzufügen. Alle anderen Elemente von H – bis auf gegebenenfalls baumelnde Kanten (die zu löschen sind) – werden unverändert nach H' überführt. Bemerkenswert ist, dass im Allgemeinen ρ weder surjektiv noch total sein muss: Der Morphismus ρ ist nicht total, da Elemente von H möglicherweise in H' gelöscht sind, also nicht durch ρ abgebildet werden. ρ kann nicht surjektiv sein, falls in H' neue Elemente hinzugefügt werden; diese Elemente werden dann nicht von ρ sondern von μ abgebildet. Es gibt keine Elemente in H', die nicht von μ oder ρ überstrichen werden.

Der SPO-Ansatz ist nicht konstruktiv in dem Sinn, dass er direkt z. B. oben stehende algorithmische Beschreibung der Ersetzung vorschreiben würde. Er *charakterisiert* H' vielmehr innerhalb der Klasse aller Graphen, indem die kanonische Pushout Konstruktion in der Kategorie **Graph** verwendet wird. Die formale Definition des SPO-Ansatzes ist an sich sehr simpel:

[13] Man beachte den Unterschied zwischen einer partiellen und einer nicht surjektiven Abbildung.
[14] Die Elemente im Bildbereich einer Funktion oder eines Morphismus werden von ihr oder ihm *überstrichen*.

Definition 2.34 (SPO) Seien die Graphen L, R, H und zwei Graphhomomorphismen $m \in \mathbb{P}(L, H), r \in \mathbb{P}(L, R)$ gegeben, dann ist H' charakterisiert als Pushout-Objekt des Tripels (L, m, r).

Mit dem Diagramm aus Abbildung 2.6 folgt, dass das Pushout des Tripels (L, m, r) das Tripel (H', ρ, μ) ist. Aus den elementaren Eigenschaften der Kategorientheorie und insbesondere der universellen Konstruktion des Pushouts in der Kategorie **Graph** (siehe Satz 2.33), folgt damit, dass für jede Graphersetzungsregel $p : L \xrightarrow{r} R$ und jede Passung $m \in \mathbb{T}(L, H)$ bezüglich eines Arbeitsgraphen H eine eindeutig definierte direkte Ableitung H' existiert.

Eine detaillierte Einführung in den SPO-Ansatz ist in [EHK+99] zu finden. Die praktischen Aspekte des Ersetzungsschritts beleuchten wir in Abschnitt 4.3 sowie in Kapitel 8.

KAPITEL 3

VERWANDTE ARBEITEN

Die ersten Ansätze zur Graphersetzung reichen bis ins Jahr 1969 zurück: Pfaltz legte damals mit dem Aufsatz über „Web Grammars“ den Grundstein dieses Teilgebiets der Mathematik und Informatik [PR69]. In den achtziger Jahren des letzten Jahrhunderts stellte sich die Graphersetzung als ein sehr schönes theoretisches Konzept dar, das sich aber nur als abstraktes Beschreibungsinstrument eines ansonsten von Hand implementierten Verfahrens eignete. Auch das IPSEN-Projekt [Nag90, Nag96] von Nagel hatte zunächst diese Inklination; allerdings brachte es am Ende PROGRES – das erste weitgehend automatisches Graphersetzungswerkzeug – hervor (siehe auch Abschnitt 3.2.2).

Aus den vielfältigen, seit damals entstandenen Veröffentlichungen stellen wir in diesem Kapitel die zu unserer Arbeit verwandten zusammen, wobei wir aus dem zahlreichen, vor allem theoretisch geprägten, Material jeweils die für unsere Arbeit relevanten Hauptströmungen berücksichtigen. Wir beginnen mit der Graphmustersuche in Abschnitt 3.1. Schließlich folgt in Abschnitt 3.2 eine Übersicht der prominentesten Graphersetzungswerkzeuge. Die folgenden Abschnitte stützen sich in Teilen auf Recherchen und Analysen der Verfahren, die von Batz, Müller und Denninger erarbeitet wurden [Bat05a, Bat05b, Bat06, Den07, Mül07, DGG08].

Bevor ich diesen Abschnitt beende, möchte ich noch auf zwei besonders gelungene Arbeiten verweisen. Eine schöne Übersicht zum damaligen Stand der Graphersetzung bietet der Aufsatz von Blostein, Fahmy und Grbavec [BFG95]. Die wichtigsten theoretischen und praktischen Entwicklungen sind im dreibändigen „Handbook of Graph Grammars and Computing by Graph Transformation“ erschöpfend dargestellt [Roz99].

3.1 Graphmustersuche

Die effiziente Graphmustersuche ist einer der entscheidenden Punkte aller Bestrebungen, der Graphersetzung zum Durchbruch zu verhelfen. Wegen der NP-Vollständigkeit des Problems im strengen Sinn kann es kein Verfahren geben, das immer effizient – sprich polynomiell – ist[1]. Jedes Verfahren muss also Schwachstellen haben, die Frage ist nur, wo diese zum Tragen kommen und ob dies für die jeweilige Anwendung akzeptabel ist.

[1] Zumindest unter der Annahme P $\neq$ NP.

3.1.1 Suchprobleme auf Graphen

Zunächst betrachten wir, welche Such- und Vergleichsprobleme zwischen Graphen in der Literatur üblich sind (siehe Tabelle 3.1, ebendort werden die im Folgenden verwendeten Abkürzungen eingeführt). Dies ist nötig, da häufig Verschiedenes miteinander vermischt wird und somit eine Begriffsklärung angemessen scheint. Zur Beschreibung nutzen wir die in Kapitel 2 gelegten Grundlagen. Eine gute Einführung in dieses Gebiet bietet ein Übersichtsartikel zur Graphmustersuche von Conte et al. mit dem Titel „Thirty Years of Graph Matching in Pattern Recognition“ [CFSV04a].

Es gibt eine ganze Reihe erstaunlicher komplexitätstheoretischer Erkenntnisse zu Suchproblemen auf Graphen, in Sonderheit zu GI und SGI. Unter anderem ist GI der erfolgversprechendste Kandidat für eine Komplexitätsklasse, die zwar in NP liegt, aber nicht NP-vollständig ist und trotzdem keine polynomiellen Lösungen besitzt. Trotzdem konnte Eugene M. Luks im Jahr 1981 zeigen, dass das Graphisomorphieproblem (GI) für Graphen mit beschränktem Eckengrad in polynomieller Zeit lösbar ist.

Am wichtigsten für unsere Arbeit ist das SGI und SGH, also das Teilgraph*iso*morphieproblem und das Teilgraph*homo*morphieproblem. Diese Probleme entsprechen direkt der Aufgabe, einen Mustergraphen in einem Arbeitsgraphen zu suchen. Für die CSI- und Max-CSI-Probleme sind ganz andere Algorithmen zu bevorzugen, als dies für das SGI-Problem der Fall ist. Denn bei CSI, Max-CSI, FSD und Fuzzy-SGI hat man in der Regel[2] eine recht große Menge an Arbeitsgraphen, die unveränderlich und gegen eine kleine, sich ändernde Menge von Mustergraphen zu prüfen sind. In dieser Situation lohnt es sich, z. T. aufwendige (d. h. exponentielle) Vorverarbeitung der Arbeitsgraphen durchzuführen, um dann den eigentlichen Graphvergleich für einen gewissen Mustergraphen schneller zu beantworten. Mithin sind viele der Verfahren zur Lösung von Suchproblemen eben nicht zur Graphmustersuche in unserem Sinn anwendbar. Wir können höchstens gewisse Anleihen aus diesen Verfahren machen.

3.1.2 Verfahren ohne Suchplanung mit Rücksetzen

3.1.2.1 Ullmann

Einer der ersten Algorithmen zur Graphmustersuche aus dem Jahr 1976 stammt von Ullmann [Ull76]. Er benutzt einen Algorithmus mit Rücksetzen[3], der die möglichen Eckenabbildungen aufzählt und dabei die benötigten Kanten im Arbeitsgraphen prüft. Dabei wird der Suchraum beschnitten, um die Laufzeit der Suche (hoffentlich) erheblich zu verringern. Ein erstes Kriterium zur Beschneidung ist Ecken mit zu geringem Grad auszuschließen. Dann wird bei jedem Durchlauf eine sogenannte Verfeinerungsprozedur ausgeführt, die dem Zusammenhang des Arbeitsgraphen folgend weitere Eckenabbildungen entfernt. Diese Verfeinerungsprozedur wird so lange ausgeführt, bis nichts mehr entfernt werden kann. Das Ergebnis sind dann alle Passungen des Mustergraphen im Arbeitsgraphen.

[2] Anwendungen wie z. B. in der Chemie verlangen der Vergleich neuer Wirkstoffe mit einer großen Datenbank existierender Substanzen. Darum sind diese Algorithmen normalerweise für diesen Fall getrimmt.

[3] engl.: *backtracking*; das Prinzip wird in der griechischen Mythologie (Ariadnefaden) erstmals erwähnt [Homhr].

Tabelle 3.1: Nomenklatur von Suchproblemen auf Graphen

Teilgraph, *sub graph* (SG)	
FORMELL:	$G' \subseteq G$
PROBLEM:	Ist G' Teilgraph von G?
KOMPLEXITÄT:	Polynomiell.

Untergraph, *induced sub graph* (ISG)	
FORMELL:	$\forall k \in K : [\mathrm{src}(k) \in E' \wedge \mathrm{tgt}(k) \in E'] \Rightarrow k \in K'$
PROBLEM:	Ist G' Untergraph von G?
KOMPLEXITÄT:	Polynomiell.

Graphisomorphie, *graph isomorphism* (GI)	
FORMELL:	$G' \approx G$
PROBLEM:	Ist G' isomorph zu G? [McK81]
KOMPLEXITÄT:	In NP, unklar ob NP-vollständig (siehe Abschnitt 2.3).

Teilgraphisomorphie, *sub graph isomorphism* (SGI)	
FORMELL:	$G' \precsim G$
PROBLEM:	Ist G' ein isomorpher Teilgraph von G?
KOMPLEXITÄT:	NP-vollständig [GJ90], in planaren Graphen polynomiell [Epp95].

Teilgraphhomomorphie, *sub graph homomorphism* (SGH)	
FORMELL:	$\mathbb{T}(G', G) \neq \emptyset$
PROBLEM:	Ist G' ein homomorpher Teilgraph von G?
KOMPLEXITÄT:	NP-vollständig, da SGI damit lösbar.

Automorphismus-Gruppe, *automorphism* (AUT)	
FORMELL:	$\left\{G' \middle| G' \approx G\right\}$
PROBLEM:	Bestimme *alle* zu G isomorphen Graphen!
KOMPLEXITÄT:	In NP.

Gemeinsame Teilgraphisomorphie, *common sub graph isomorphism* (CSI)	
FORMELL:	$\left\{G' \middle| \forall G_i \in M : G' \precsim G_i\right\}$
PROBLEM:	Bestimme alle Graphen, die zu jedem Graph einer Menge M SGI sind!
KOMPLEXITÄT:	NP-hart.

Maximale Teilgraphisomorphie, *maximum common sub graph isomorphism* (Max-CSI)	
FORMELL:	$\arg\max_{G' : \forall G_i \in M : G' \precsim G_i} (\lvert G' \rvert)$
PROBLEM:	Bestimme den/die größten Graphen einer CSI Menge!
KOMPLEXITÄT:	NP-hart.

Quantil-Maximale Teilgraphisomorphie, *frequent sub graph discovery* (FSD)	
FORMELL:	$\arg\max_{\arg\max_{G' : \forall G_i \in M : G' \precsim G_i} (\lvert G' \rvert) =: \tilde{M} \subseteq M \wedge \frac{\lvert \tilde{M} \rvert}{\lvert M \rvert} > p} \left(\lvert \tilde{M} \rvert \right)$
PROBLEM:	Bestimme eine Teilmenge $\tilde{M}$ einer Menge M von Graphen, die mindestens p-Prozent der Ursprungsmenge enthält, so dass die Graphen, die isomorph zur gewählten Teilmenge sind, maximal groß sind!
KOMPLEXITÄT:	NP-hart in der Graphgröße, linear in der Zahl der Graphen [KK04].

Unscharfe Teilgraphisomorphie, *fuzzy sub graph isomorphism* (Fuzzy-SGI)	
FORMELL:	$\exists G'' : \lvert\lvert G'', G' \rvert\rvert < k \wedge G'' \precsim G$
PROBLEM:	Ist G' (oder ein „leicht veränderter“ Graph G'') ein isomorpher Teilgraph von G? Dabei kommen verschiedene Maße für die Ähnlichkeit von Graphen zum Einsatz (z. B. eine Levenshtein-Abstand für Graphen [WH04, MWH00, Lev66]).
KOMPLEXITÄT:	NP-hart.

In seinen Experimenten behandelt Ullmann die Ecken im Mustergraphen mit hohem Grad zuerst. Dies soll dazu führen, dass das Fehlen von Kanten im Arbeitsgraphen frühzeitig entdeckt wird und so stärker zur Reduktion des Suchraums beiträgt. Ullmann schlägt fernerhin vor, dass die Bearbeitungsreihenfolge der Musterecken vom durch die Anwendung gegebenen Kontext abhängen soll. Dies ist von der Idee her auch Grundlage der neueren suchplanbasierten Verfahren (vgl. Abschnitt 3.1.3.3 sowie Kapitel 5).

Ullmanns Algorithmus benötigt sehr viel Speicher, um alle Passungen direkt zu repräsentieren, und berechnet überdies immer *alle* Passungen, selbst wenn man nur eine benötigt. Der Algorithmus kann nicht direkt mit Multigraphen umgehen, offensichtliche Erweiterungen brauchen noch mehr Speicher. Eine Berücksichtigung der Typen von Ecken und Kanten ist nicht vorgesehen. Die initiale Beschneidung des Suchraums wie auch die Verfeinerungsprozedur sind problematisch, falls die Eckengrade im einem Mustergraphen gering sind; die Mustersuche kann dann zum vollständigen Aufzählen entarten.

3.1.2.2 Eppstein

Eppstein [Epp94, Epp95] stellt einen Algorithmus vor, der in planaren Graphen die Zahl der Vorkommen eines festen Mustergraphen in linearer Zeit bezüglich der Größe des Arbeitsgraphen feststellen kann. Der Algorithmus basiert darauf, dass für einen planaren Graphen effizient sogenannte Baumzerlegungen berechenbar sind. Aufbauend darauf kann der Algorithmus die Mustergraphen jeweils lokal in den einzelnen Baumzerlegungen suchen. Dabei ist sichergestellt, dass eine Ecke für eine Passung höchstens in drei der Teilgraphen sein kann. Da die Größe der Zerlegungen des Arbeitsgraphen alle linear von der Größe des Mustergraphen abhängen, ergibt sich so die lineare Komplexität.

Jedoch funktioniert der Algorithmus nur bei planaren Graphen. Eine Erweiterung auf erheblich größere Klassen von Graphen ist bisher nicht gelungen. Das von Eppstein vorgestellte Verfahren beschränkt sich zunächst auf zusammenhängende Mustergraphen. Eine Erweiterung auf unzusammenhängende Muster gelingt nur für das Zählen *aller* Mustervorkommen, *das Finden* derselben in Linearzeit ist vom Autor nicht erreicht worden. Typen oder Multigraphen sollten durch offensichtliche Erweiterungen zu berücksichtigen sein.

3.1.2.3 VF2

Cordella et al. schlagen mit dem VF2-Algorithmus ebenfalls ein auf Rücksetzen und Beschneiden des Suchraums basierendes Verfahren vor [CFSV04b]. VF2 ist eine Erweiterung des ullmannschen Algorithmus. Als Erstes werden Eckenabbildungen, die nicht dem lokalen Zusammenhang des Mustergraphen entsprechen, ausgeschlossen. Dann wird eine Art lokal beschränkte Breitensuche durchgeführt. Genauer gesagt wird geprüft, ob die Nachbarschaft einer potenziellen Erweiterung der Eckenabbildung auch genug Ecken enthält, die mit den schon geprüften Ecken passend verbunden sind. Schließlich wird noch geprüft, ob die Ecken und Kanten der potenziellen Erweiterung passende Markierungen tragen.

Der VF2-Algorithmus weist, bezogen auf die Zahl der Ecken im Arbeitsgraphen, eine bessere Speicherkomplexität als der ullmannsche Algorithmus auf (linear statt

kubisch). Die Laufzeit von VF2 ist um einen konstanten Faktor besser als der ullmannsche Algorithmus. Die (komplexen) Typen von Ecken und Kanten werden offenbar bei der Suche erst spät berücksichtigt.

3.1.3 Verfahren mit Suchplanung

3.1.3.1 Progres

Das Verfahren von Zündorf zur Graphmustersuche wird im dem Graphersetzungswerkzeug Progres eingesetzt [Zün96]. Seine Herangehensweise, die ebenfalls eine gewisse Ähnlichkeit mit unserem suchplanbasierten Ansatz hat, benutzt zweierlei *elementare Suchbefehle*: das Erweitern einer Passung, dem Graphzusammenhang folgend, und das Suchen einer passenden Aufsatzecke, von der ausgehend in nachfolgenden Schritten die Passung erweitert werden kann. Darüber hinaus können Pfade unbeschränkter Länge gesucht werden, Ecken können mit Passungen vorbelegt werden und Attribute von Ecken können indiziert werden, um sie schneller wiederzufinden. Um diese Suchbefehle bewerten zu können, beschreibt Zündorf ein recht ausgeklügeltes Kostenmodell. Hingegen erfolgt die Planung der Suchstrategie durch eine Best-First-Suche, die abgesehen von den potenziell teuren *Aufpunkt*[4]-Befehlen gierig arbeitet. Die Kosten der elementaren Suchbefehle werden lediglich aus statischem Wissen, aus dem Metamodell und generellen Annahmen gewonnen. Dieses statische Wissen wird im Metamodell in Form von Verbindungskardinalitäten im Sinn von UML angegeben.

Typen, Multigraphen und weitere elaborierte Eigenschaften werden von diesem Verfahren ebenfalls abgedeckt. Obgleich Zündorf ein sehr ausgeklügeltes Kostenmodell angibt, werden diese Kosten, wenn es zum eigentlichen Planen kommt, in fünf grobe Kategorien eingeteilt, was zwar für die Best-First-Suche dienlich ist, aber die Genauigkeit des Modells über Gebühr verschlechtert.

3.1.3.2 Dörr

Dörr identifiziert Büschel von isomorphen Kanten[5] als Wurzel der exponentiellen Auffächerung bei der Graphmustersuche [Dör95]. Er prägt für solche Situationen den Begriff der (starken) *V-Struktur*. V-Strukturen können möglicherweise umgangen werden, indem man bei der Mustersuche nicht vom Büschel (also der Spitze des V) her dem Graphzusammenhang folgt, sondern die entgegengesetzte Richtung wählt. Falls eine solche V-Struktur nicht umgangen werden kann, weil sich z. B. an die eine V-Struktur eine andere V-Struktur ungünstig anschließt, dann ist das Auffächern nicht zu verhindern.

Jedoch hat der dörrsche Ansatz kein Kostenmodell. Der Arbeitsgraph wird in einer polynomiellen Vorverarbeitung[6] auf unumgehbare V-Strukturen – bezüglich eines Mustergraphen – hin überprüft. Falls solche auftreten, scheitert das Verfahren; es wird also keine Passung berechnet. Ist der Arbeitsgraph frei von unumgehbaren V-Strukturen, wird in linearer Zeit eine Passung berechnet; dabei werden – beim

[4] engl.: *lookup*

[5] Kanten sind isomorph, wenn ihre Typen kompatibel sind und sie von Ecken mit jeweils kompatiblem Typ ausgehen sowie an Ecken mit jeweils kompatiblem Typ münden.

[6] Implementierungen, deren Laufzeit linear von der Größe des Arbeitsgraphen abhängt, sind möglich.

schrittweisen Erweitern der potenziellen Passungen – alle V-Strukturen umgangen. Dörr gibt noch weitere Optimierungstechniken, wie die *rahmenbasierte Speicherung*[7] der Graphen an. Es ist uns keine Implementierung seiner Techniken bekannt. Unsere Experimente haben gezeigt, dass alle Techniken, die Dörr vorschlägt, bis auf die Umgehung der V-Strukturen, praktisch keinen Vorteil bringen [Bat05a]. Wir erweitern in Kapitel 5 das dörrsche Verfahren zur V-Strukturumgehung um eine kostenbasierte heuristische Suchplanung, die in allen Arbeitsgraphen entsprechende Mustergraphen suchen kann, allerdings – falls V-Strukturen anwesend sind – nicht immer in linearer Zeit.

3.1.3.3 Suchplanung nach Varró

Varró et al. schlagen ein Verfahren zur Mustersuche vor, das ähnlich wie das zündorfsche Verfahren (vgl. Abschnitt 3.1.3.1) auf einer Suchplanung auf Basis kostenbewerteter elementarer Suchbefehle fußt [VVF05]. Im Unterschied zu dem von Zündorf beschriebenen Ansatz benutzt Varró allerdings eine viel profundere Methode, um aus den Kosten eine Reihenfolge der elementaren Suchbefehle abzuleiten. Zunächst legt Varró Kosten für die elementaren Suchbefehle fest. Er bildet einen *Planungsgraphen*, der dem Mustergraphen verwandt ist, und trägt dort die Suchbefehle ein, die der Suche der jeweiligen Graphelemente entsprechen. Auf diesem Planungsgraphen wird dann der minimale spannende Arboreszent (MSA) mittels des Edmonds/Chu-Liu-Algorithmus [Edm67, CL65] bestimmt. Hernach wird die Reihenfolge der Befehle durch eine best-first Strategie beim Ablaufen des MSA bestimmt.

Die Suchplanung nach Varró ist sehr ähnlich zu dem von uns im gleichen Jahr unabhängig von Varró entwickelten Verfahren (vgl. Kapitel 5 und besonders Abschnitt 5.3). Allerdings wird das Nachschlagen von Kanten vom varróschen Ansatz nicht unterstützt. Die Kosten werden anders gebildet und auch die Herkunft des Verfahrens aus den dörrschen Grundlagen wurde von Varró nicht erkannt [BKG08]. Varró hat keine Implementierung als Referenz angegeben, mithin gibt es auch keine Laufzeitmessungen.

3.1.4 Reduktionistische Verfahren

Die Abbildung von Graphmustersuche auf CSP[8] ist von mehreren Autoren versucht worden. Rudolf [Rud98] wie auch Larrosa und Valiente [LV02] befleißigen sich dieser Techniken (Abschnitt 3.1.4.1). Die Hoffnung hinter solch einer Reduktion eines NP-schweren Problems auf ein zweites, gut verstandenes Problem ist, dass von effizienten Lösungsverfahren des zweiten Problems profitiert werden kann. Oft gibt es für solche Probleme auch eine große Zahl von effizienten Werkzeugen; man denke dabei z. B. an relationale Algebra oder lineares Programmieren. Dieser Idee folgt auch eine Arbeit von Varró (Abschnitt 3.1.4.2) und einer unserer Ansätze (Abschnitt 5.1), wobei beide Arbeiten nicht auf CSP, sondern auf Anfragen in relationaler Algebra abbilden.

[7] engl.: *frame-based layout*

[8] engl.: *constraint satisfaction problem*; für eine Einführung siehe z. B. die Monografie „Constraint Processing“ von Dechter [Dec03]

3.1.4.1 CSP

Um die Mustersuche zu beschleunigen, versucht Rudolf dem CSP-Löser die Informationen, die in der Graphstruktur stecken, zugänglich zu machen [Rud98]. Es werden Bedingungen quasi in Graphform als Anfragen an den CSP-Löser weitergegeben. Diese Graphen erinnern stark an den von uns in Abschnitt 5.3 beschriebenen Planungsgraphen. Es liegt also nahe, eine Verwandtschaft zwischen diesen Anfragen und den von uns entwickelten elementaren Suchbefehlen anzunehmen. Ebenso behandelt die *Variablenanordnung* beim CSP dasselbe Problem wie unsere Suchplanung, nämlich die Festlegung einer günstigen Reihenfolge der betrachteten Passungsteile.

Larrosa und Valiente betrachten fünf verschiedene Formulierungen der Mustersuche innerhalb eines CSP-Rahmenwerks [LV02]. Dabei ist der fünfte ein von ihnen entwickelter neuer Ansatz, den sie nRF+ nennen, und der in den meisten Fällen die anderen Ansätze im Bezug auf die Geschwindigkeit überflügelt. Larrosa und Valiente behaupten, dass das eigene der fünf Verfahren, die sie betrachten, dem Ullmann-Algorithmus entspricht, und dass nRF+ im wesentlichen eine stärkere Beschneidung des Suchraums beinhaltet.

Ein Vorgriff zu Abschnitt 9.1 und insbesondere auf die Ergebnisse des Varró-Benchmarks sei hier gestattet: wie gesagt, ähnelt das rudolfsche Verfahren sehr stark unserer suchplanbasierten Mustersuche. Umso erstaunlicher ist es, dass AGG, welches auf dem rudolfschen Verfahren aufbaut, mit Abstand das langsamste Graphersetzungswerkzeug ist, während unser System GRGEN ebenfalls mit beachtlichem Abstand das schnellste System darstellt. Larrosa und Valiente argumentieren nicht, warum eine Reformulierung des ullmannschen Algorithmus im CSP-Kontext überhaupt aussagekräftig ist. Letztlich bleibt das Problem aller reduktionistischen Verfahren, dass die eigentliche Charakteristik des Ursprungsproblems weitgehend verloren geht und so, unter Umständen, eine eigentlich gar nicht schwierige Aufgabe nur durch die Reformulierung komplex wird.

3.1.4.2 SQL nach Varró

In dieser Arbeit bilden Varró et al. das Problem der Graphmustersuche auf Anfragen in relationaler Algebra ab und erreichen so, dass eine generische Lösung mit relationalen Datenbanksystemen (RDBMSs[9]) möglich ist [VFV05]. Varró definiert hierzu für jede Regel eine Sicht[10] im Datenbanksystem. Die Anfragen, die diese Sichten definieren, werden mittels Verbundoperationen[11] formuliert. Allerdings gibt es keine nennenswerte Planung der Anordnung der Join-Operationen, was zunächst wie eine gute Idee erscheint, da somit der RDBMSs volle Freiheit bei der Optimierung besitzt, sich aber – wie Abschnitt 5.1 zeigt – in der Praxis leider eher negativ auf die Laufzeit der Suche auswirkt.

[9] engl.: *relational database management system*

[10] engl.: *view*. Der Datenbankbenutzer erstellt eine Sicht – als virtuelle Tabelle – mittels einer Abfrage. Diese kann er dann wie eine normale Tabelle abfragen. Wann immer eine Abfrage diese Sicht benutzt, wird diese zuvor durch das RDBMS berechnet – sie könnte aber auch inkrementell nachgeführt werden, was ggf. zu Geschwindigkeitssteigerungen gegenüber der ständigen Neuberechnung führen kann. Eine View stellt also im Wesentlichen einen Alias für eine Abfrage dar.

[11] engl.: *join*

3.1.5 Andere Verfahren

3.1.5.1 Möller

Möller hat ein System zur Ausführung von turingvollständigen Graphprogrammen entwickelt, das auch Mustersuche betreibt um Regelprimitive anzuwenden [Möl03]. Er spricht in diesem Zusammenhang von einer *virtuellen Maschine* (VM), die diese Programme ausführt, wobei die VM zwar Befehle für die Ablaufsteuerung, nicht aber für die Mustersuche besitzt. Da wir in Abschnitt 5.2 auch eine virtuelle Maschine für die Graphmustersuche einführen, möchten wir hier betonen, dass diese beiden außer dem Begriff selbst nichts gemeinsam haben.

Die Mustersuche, die nichts mit der virtuellen Maschine selbst zu tun hat, benutzt einfaches Rücksetzen ohne Planung. Möller erlaubt nur injektive Passungen. Formal basiert der Ersetzungsformalismus auf dem DPO-Ansatz. Die Graphprogramme sind mit den von uns erfundenen erweitern Graphersetzungssequenzen (siehe Abschnitt 4.4) oder auch mit der Ablaufsteuerung in PROGRES vergleichbar.

3.1.5.2 Nauty

NAUTY von McKay [McK81, McK90] gilt als das leistungsstärkste Verfahren für Isomorphietests. NAUTY führt einen i. A. exponentielle Vorverarbeitung der Muster- und Arbeitsgraphen durch und kann dann für einen Mustergraphen in polynomieller Zeit entschieden, ob dieser isomorph zu einem der Arbeitsgraphen ist. Dies ist besonders dann hilfreich, wenn ein oder mehre Mustergraphen gegen eine (ebenso große wie auch unveränderliche) Sammlung von Arbeitsgraphen auf Isomorphie geprüft werden sollen, da so die kostspielige (exponentielle) Vortransformation nur einmal anfällt und damit jedes Paar effizient (polynomiell) auf Isomorphie geprüft werden kann.

Offenbar ist dieses Vorgehen für ein System, das Graphersetzung durchführen soll, nicht sonderlich brauchbar, da hierbei ein einziger, sich aber mit jedem Transformationsschritt ändernder Arbeitsgraph, vorliegt. Dies würde kostspielige Vorverarbeitungen des jeweils aktuellen Arbeitsgraphen nach sich ziehen, also in Summe für jeden Transformationsschritt einen i. A. exponentiellen Aufwand bedeuten. Überdies sind zwar einfache Markierungen, nicht aber komplexe Typen für Ecken und auf keinen Fall für Kanten machbar. Multigraphen sind nicht einfach integrierbar.

3.1.5.3 Messmer und Bunke

Messmer und Bunke schlagen einen Verfahren vor [MB00b, BGT90], das dem RETE-Ansatz [For79, For82] aus der KI-Mustersuche entstammt. Dieses Verfahren wurde entwickelt, um gleichzeitig mehrere Mustergraphen in einem einzigen Arbeitsgraphen zu finden. Mittels einer Vorverarbeitung, die unabhängig vom Arbeitsgraphen geschehen kann, werden die Mustergraphen in eine spezielle Darstellung überführt. Dabei werden insbesondere gemeinsame Teile verschiedener Mustergraphen zusammengeführt, die so nur einmal zu suchen sind.

Dieser Algorithmus ist für die Graphersetzung nicht unbedingt geeignet, da in der Regel nur ein Mustergraph gesucht, dann angewendet und hernach wieder ein neuer Mustergraph in dem durch die vorangehende Anwendung veränderten Arbeitsgraphen zu suchen ist. Obschon es Situationen gibt, in denen wir nur an Passungen verschiedener Muster auf ein und denselben Arbeitsgraphen interessiert sind

– z. B. bei der simultanen Ersetzung unabhängiger Muster – ist gerade in diesem Fall das Messmer-Bunke-Verfahren nicht hilfreich: wenn nämlich n recht ähnliche Mustergraphen gleichzeitig gesucht werden, dann kann sich zwar eine Beschleunigung um den Faktor n ergeben – allerdings ist dann auch zu erwarten, dass die gefundenen Mustervorkommen nicht unabhängig sind, also die Mustergraphen nicht konfliktfrei anwendbar sind. Jedoch wenn es um n vollständig unterschiedliche – also insbesondere höchstwahrscheinlich unabhängig, sprich konfliktfrei, anwendbare – Mustergraphen handelt, ist keine Beschleunigung zu erwarten.

3.2 Graphersetzungswerkzeuge

Zunächst formulieren wir wünschenswerte Kriterien für ein Graphersetzungswerkzeug. Anschließend analysieren wir verschiedene Graphersetzungswerkzeuge bezüglich dieser Kriterien und ziehen ein Fazit.

3.2.1 Kriterien

Die folgenden Kriterien ergeben sich aus der Forderung eines möglichst allgemeinen, leistungsstarken und mit leicht einsehbarer Spezifikation versehenen Graphersetzungswerkzeugs, das überdies implementiert ist und in einem Übersetzer eingesetzt werden kann.

Um die Bemühungen nach formal korrekten Übersetzern zu unterstützen, ist eine formale Definition der Semantik des Graphersetzungswerkzeugs wünschenswert. Diese formale Semantik kann insbesondere dazu verwendet werden, die bei ad hoc Verfahren häufig anzutreffenden Ideosynkrasien zu vermeiden. Die Ausdrucksstärke und intuitive Niederschrift der Transformationen sind eine Grundvoraussetzung für ein erfolgreiches Benutzen des Systems. Desweiteren möchten wir einen möglichst einfachen Übergang zwischen unserem „Problemraum“ und der Darstellung im Graphersetzungswerkzeug erreichen. Hierzu ist es notwendig, dass der Formalismus zur Beschreibung von Graphersetzung facettenreich ausgestaltet ist. Es reicht insbesondere nicht aus, nur Ecken und Kanten zu haben, obgleich man darauf auch beliebige komplexere Strukturen abbilden kann. Es muss möglich sein, Ecken und auch Kanten mittels Graphersetzungsregeln ersatzlos zu streichen bzw. vollständig zu kopieren. Für Anwendungen im Übersetzerbau wie zum Beispiel das *Entfernen toten Codes* bzw. dem *Ausrollen von Schleifen* ist dies wünschenswert.

3.2.2 Progres

Schürr et al. haben Progres seit Mitte der neunziger Jahre des letzten Jahrhunderts [SWZ99] zu einem System mit über 250.000 Zeilen Code und einer zugehörigen formalen Theorie anwachsen lassen. Progres ist das erste allgemeine Graphersetzungswerkzeug, das implementiert wurde. Es basiert auf dem IPSEN-Projekt von Nagl [Nag96], das seinen Ursprung in den grafischen Programmiersystemen hat. Progres hat eine reichhaltige Spezifikationssprache, jedoch ist die Formalisierung nicht vollständig und folgt keinen einheitlichen Prinzipien. Ferner ist die Ablaufsteuerung mit den Graphersetzungsregeln verwoben; Vertreter dieser Bauart nennt man *programmierte Graphersetzer*. Die Anwendungsreihenfolge von Regeln und Re-

gelgruppen mittels BCF[12] erfolgt nichtdeterministisch und zeigt somit ein nicht wiederholbares Verhalten, was insbesondere im Übersetzerbau unerwünscht sein kann. Die Implementierung ist leider nur in binärer Form verfügbar und daher an Umgebungen gebunden, die sich heutzutage kaum reproduzieren lassen. Eine leichte Integration scheint ausgeschlossen.

3.2.3 AGG

Das AGG-Werkzeug von Tänzer et al. ist das erste System [ERT99], das sich strikt an die SPO-Semantik hält. Die Spezifikation von Regeln und Graphmodellen erfolgt ausschließlich grafisch, was die Integration in einen automatisierten Prozess erschwert. Die Spezifikationssprache ist nicht sehr reichhaltig, enthält aber mit negativen Anwendungsbedingungen ein innovatives Konzept. Abgesehen von der puren Graphersetzung ist es möglich, Paare von Regeln auf Konfliktfreiheit zu prüfen. AGG besitzt nur eine rudimentäre Steuerung der Regelanwendung; jede komplexere Anwendungsreihenfolge muss über eine Java-Schnittstelle ausprogrammiert werden.

3.2.4 OPTIMIX

Das OPTIMIX-System von Assmann ist das einzige uns bekannte Werkzeug, das unmittelbar im Kontext des Übersetzerbaus entstanden ist [Ass00]. Es hat zwei Besonderheiten: Erstens ist der Regelsatz, ebenso wie die Anwendung dieser Regeln, sehr eingeschränkt. Zweitens ist es dadurch möglich, für alle spezifizierbaren Regelsätze die Terminierung und die Konfluenz zu zeigen. Dies hat zur Folge, dass es nicht möglich ist, selbst einfache Optimierungen, wie das Entfernen toten Codes, direkt auszudrücken – Assmann selbst konstatiert, dass es nicht gelingt, das Entfernen partieller Redundanzen[13] zu formulieren.

3.2.5 FUJABA

Das FUJABA-Werkzeug von Zündorf ist ein Nachfolger von PROGRES, der allerdings vollständig neu implementiert wurde. Das intendierte Einsatzgebiet von FUJABA ist spezifikationsgetriebene Programmierung. Die Semantik von FUJABA ist durch Rückgriff auf die Semantik von PROGRES definiert. In FUJABA werden Regeln ausschließlich grafisch angegeben. Die Anwendungssteuerung kann auch grafisch angegeben oder in Java ausprogrammiert werden. FUJABA bildet die Graphmodelle, die als UML-Klassendiagramme vorliegen, und die Graphersetzungsregeln unmittelbar auf die Eigenschaften von Java ab – dadurch ist der Übergang zwischen Programmierung und Graphersetzung fließend.

FUJABA ist in einem gewissen Sinn kein automatisches Graphersetzungswerkzeug, da aus einer deklarativen Spezifikation nicht direkt eine Implementierung dieser Regel erzeugt werden kann. Der Benutzer muss Aufpunkte für die Mustersuche manuell festlegen, was, falls dies optimal geschehen soll, erstens nicht trivial und zweitens auch vom jeweiligen Arbeitsgraphen abhängig ist.

[12] engl.: *basic control flow operators*

[13] Das Entfernen partieller Redundanzen ist eine Optimierung aus dem Übersetzerbau.

3.2.6 Fazit

Die Ausdrucksstärke und intuitive Notation von PROGRES ist – unter den besprochenen Systemen – dem Ideal am nächsten. Die Notation von OPTIMIX ist für unseren Zweck, nämlich dem Hinzufügen und Löschen von Ecken, recht unpraktisch, da es keine direkte Spezifikation dieser Operationen erlaubt. AGG hat nicht den Fokus, ein allgemeines Werkzeug zu sein, auch wenn es inzwischen so auf der Webseite beworben wird. Es wurde ausschließlich als visuelle Programmiersprache entwickelt und führt auf seine Weise den Ansatz von PROGRES zu Ende.

Im Übersetzerbau ist Graphersetzung zum direkten Spezifizieren von Optimierungen auf Zwischendarstellungen ein noch kaum beackertes Einsatzfeld. Die einzigen in der Literatur bekannten Aktivitäten stammen von Assmann. Anwendungen von PROGRES auf diesem Gebiet sind nicht bekannt. Obschon Aufsätze mit Titeln wie *„Graph Transformation for Specification and Programming“* anderes vermuten lassen, behandeln diese Arbeiten (siehe [AEH+99] und [And96]) nur die so genannte visuelle Programmierung — das Spezifizieren von Programmen mittels Graphen und deren Ersetzung.

PROGRES ist das System, das unsere Anforderungen am ehesten erfüllt. Allerdings ist die Implementierung aus zweierlei Sicht mangelhaft: Erstens sind die Quellen „Closed Source“ und zweitens ist die Geschwindigkeit der Graphersetzung gering. Darum entwickeln wir in Kapitel 4 eine eigene Methode zur Graphersetzung und setzen sie wie in Kapitel 8 beschrieben um. Die Leistungsfähigkeit unserer Implementierung übersteigt die eines jeden hier vorgestellten Werkzeugs (siehe Kapitel 9).

KAPITEL 4

GRAPHERSETZUNG

In diesem Kapitel beschreiben wir die Theorie der von uns entwickelten Methodik der Graphersetzung. Am Ende jedes Abschnitts gehen wir, um die Verbindung zwischen Theorie und Praxis herzustellen, auf die Schreibweise der theoretischen Konzepte in der Spezifikationssprache des von uns entwickelten Graphersetzungswerkzeug GrGen ein.

Unsere Theorie ruht auf drei Pfeilern: Auf der *Kategorientheorie*, um den Kern des Ersetzungsvorgangs zu beschreiben, auf *denotationellen Auswertungs-* und *Ausführungsfunktion*, welche die über den reinen SPO-Ansatz hinausgehenden Elemente fassen, und schließlich auf den *Graphhomomorphismen*, die als Bindeglied zwischen den anderen beiden fungieren. Dieser neuartige Ansatz, Graphersetzung durch eine Kombination von, für das jeweilige Teilgebiet angemessenen, Beschreibungssystemen darzustellen, führt zu einer kompakten und handhabbaren Theorie; ganz im Gegensatz zum barocken Theoriegebäude, das bei dem Versuch entsteht, sämtliche Eigenschaften erweiterter Graphersetzung auf die Kategorientheorie abzubilden [EEPT06].

Graphersetzer sind in drei Hauptkomponenten zerlegbar, deren saubere Trennung nicht nur beim Entwurf die Freiheitsgrade offen legt, sondern auch zu größerer Flexibilität der Implementierung führt: Erstens die Modellierung der betrachteten Graphen sowie gegebenenfalls eine Einschränkung auf gewisse Graphfamilien. Zweitens die Art und Weise, mit der die Muster auf Passung geprüft werden. Drittens der eigentliche „Ersetzer“, der passende Stellen im Graphen gemäß dem jeweiligen Ersetzungsformalismus transformiert.

Um aus diesen Teilen eine wirkungsvoll einsetzbare Methodik zusammenzufügen, müssen ferner eine geeignete Regel- und Passungsauswahl sowie die anwendungsspezifische Integration der deklarativen Regeln in konventionelle Hochsprachen hinzukommen. Diese Sechsteilung in *Graphen* (Abschnitt 4.1), *Graphmuster* (Abschnitt 4.2), *Ersetzungsformalismus* (Abschnitt 4.3), *Regelauswahl* (Abschnitt 4.4), *Passungsauswahl* (Abschnitt 4.2.9) und *anwendungsspezifische Sprache für den Übersetzerbau* (Abschnitt 7.4.1) ist im Folgenden unser Leitfaden.

Insbesondere die Regel- und Passungsauswahl wird von den meisten anderen Ansätzen nur stiefmütterlich behandelt, ist aber für den praktischen Einsatz unerlässlich. Die zentrale, in der praktischen Anwendung der Methodik auftretende Schwierigkeit, nämlich die Geschwindigkeit der Mustersuche, wird in Kapitel 5 mit zwei neuen Ansätzen zur schnellen Graphmustersuche angegangen. Die beachtlichen Erfolge, die dabei erzielt werden konnten, sind in Kapitel 9 dargestellt. Die Spezifikationen, bestehend aus deklarativen Graphersetzungsregeln zusammen mit der

Regelauswahl, können mit dem in Kapitel 8 dargestellten Graphersetzungswerkzeug GrGen ausführbar gemacht werden.

4.1 Graphen

In diesem Abschnitt betrachten wir zunächst die Frage, welche Graphen unsere Methodik beherrschen soll, was uns zu *Graphmodellen* führt (siehe Abschnitt 4.1.1). In der Stringenz der Darstellung ist unsere Formulierung einzigartig, da normalerweise die Formalismen an einer oder gar mehreren Stellen die Ecken und Kanten unterschiedlich behandeln. Oft wird die Typisierung bei anderen Verfahren nicht mit Mehrfachvererbung realisiert, was die Spezifikationen aber unnötig schlecht strukturiert oder kompliziert werden lässt. Ebenso ist die Attributierung der Kanten oft nicht wie bei unserem Verfahren analog zu den Ecken möglich. Abschnitt 4.1.2 führt ein Beispiel für solche Graphmodelle vor. Die Frage, ob ein Graph aus Anwendersicht valide ist und wie dies festzustellen ist, beschäftigt uns in Abschnitt 4.1.3. Schließlich geben wir in Abschnitt 4.1.4 einen Einblick in die textuelle Notation der Graphmodelle.

4.1.1 Graphmodelle

Wir definieren im Folgenden eine mächtige und allgemeine Familie von *gerichteten, typisierten und attributierten Multigraphen*, die für zahlreiche Anwendungen ausreichend ist. Im Abschnitt 7.3.4 werden wir sehen, dass sich damit insbesondere auch unsere Übersetzerzwischendarstellung elegant repräsentieren lässt. Wir betrachten zunächst ein Typ- und Attributierungsmodell für Ecken und Kanten. Daraus leiten wir den Begriff des *Graphmodells*[1] ab, der analog zur Typisierung bei Hochsprachen angibt, welche Graphen konkret auftreten können, und welche Bedeutung wir ihnen beimessen.

Da wir Ecken und Kanten gleich behandeln wollen, abstrahieren wir zu einem allgemeinen Modell M_X, wobei wir später X durch endliche *Trägermengen* der Ecken E bzw. Kanten K ersetzen werden. Sei T_X eine endliche Menge der erlaubten Typen über X, dann ist

$$\mathrm{typ}_X : X \to T_X \tag{4.1}$$

eine Abbildung, die den Ecken bzw. Kanten eines Graphen einen Typ zuordnet. Im Folgenden sagen wir kurz *Graphelement*, wenn Ecken oder Kanten gemeint sind. Um eine Typhierarchie aufbauen zu können, führen wir eine Halbordnung $\leq_X$, also eine reflexive, transitive und antisymmetrische Relation auf T_X ein, die analog zu der Vererbungsbeziehung in einer modernen objektorientierten Sprache ausgestaltet wird. Wir verlangen hier, dass $\leq_X$ reflexiv ist, da wir auf diese Weise später leichter definieren können, was es bedeutet, wenn zwei Typen passen. Es bedeutet $t_1 \leq_X t_2$ mit $t_1, t_2 \in T_X$, dass t_1 ein *Obertyp* von t_2 ist. Insbesondere sagt man: t_2 *ist ein* t_1. Wir wählen hier eine auf den ersten Blick ungewohnte Richtung des Relationssymbols, jedoch ist sie so analog zur symbolischen Schreibweise der Untertypbeziehung in UML und auch konform zur Teilmengenrelation auf den Attributmengen, was wir

[1] Wir wollen unter Graphmodell die mathematische Repräsentation eines Graphen verstehen. So ist ein gerichteter Multigraph oder ein markierter gerichteter Multigraph mit einer gewissen Markierungsmenge ein Modell eines Graphen. Das Wort Modell hat in diesem Zusammenhang nichts mit dem Modellbegriff aus der Logik zu tun.

in Formel 4.4 sehen können. Wir erweitern T_X um ein kleinstes Element $\bot_X$, sodass gilt:

$$\forall t \in T_X : \bot_X \leq_X t$$

Nun versehen wir die Graphelemente noch mit Attributen. Sei A_X eine endliche Menge von Attributen. Die Graphelemente können beliebig viele – insbesondere auch keine – Attribute besitzen, also

$$\text{attr}_X : T_X \to \wp(A_X)$$

Jedem Attribut $a \in A_X$ ist ein Attributtyp $\tau \in S_X$ via der Abbildung attrtyp_X zuordnet. Über diesen ist die Wertemenge V_τ des jeweiligen Attributs, dessen konkreter Wert mit val_X ermittelbar ist, festgelegt. In Formeln:

$$\text{attrtyp}_X : A_X \to S_X \tag{4.2}$$

$$\text{val}_X : X \times A_X \to \bigcup_{\tau \in S_X} V_\tau \tag{4.3}$$

Wir stellen nun den Zusammenhang zwischen Vererbung und Attributen her. Da $t_1 \leq t_2$ bedeutet, dass t_2 ein t_1 *ist*, fordern wir, dass

$$\forall t_1, t_2 \in T_X : t_1 \leq t_2 \implies \text{attr}_X(t_1) \subseteq \text{attr}_X(t_2) \tag{4.4}$$

und

$$\text{attr}_X(\bot_X) = \emptyset$$

gilt. Dadurch sind Graphelemente – wie gewünscht – abhängig von ihrem Typ attributiert.

Definition 4.1 (Typ- und Attributierungsmodell für X) Seien die Mengen T_X, A_X, S_X mit $\bot_X \in T_X$ und die Funktionen $\text{attr}_X, \text{attrtyp}_X$ sowie die Relation $\leq_X$ gegeben wie oben beschrieben, dann heißt

$$M_X = (T_X, \leq_X, A_X, S_X, \text{attr}_X, \text{attrtyp}_X)$$

Typ- und Attributierungsmodell für X.

Mit Hilfe dieser Teilmodelle definieren wir nun den Begriff Graphmodell. Gerichtete Multigraphen mit genau solchen Graphmodellen kann unser GrGen-System verarbeiten, darum nennen wir sie im Folgenden kurz Gr-Graphen.

Definition 4.2 (Gerichteter typisierter attributierter Multigraph) Seien E und K endliche Mengen. $e \in E$ nennen wir dann eine Ecke und $k \in K$ eine Kante. Die Funktionen typ_X und val_X seien wie in Formel 4.1 und 4.3 definiert. Ferner seien src und tgt zwei Funktionen mit $\text{src} : E \to K$ sowie $\text{tgt} : E \to K$ und M_E bzw. M_K Typ- und Attributierungsmodelle für E bzw. K. Dann heißt

$$G = (E, K, \text{src}, \text{tgt}, \text{typ}_E, \text{typ}_K, \text{val}_E, \text{val}_K, M_E, M_K)$$

gerichteter typisierter attributierter Multigraph mit dem *Graphmodell* $M = (M_E, M_K)$ oder kurz Gr-Graph.

Es sei hier angemerkt, dass Graphmodelle unabhängig von konkreten Graphelementen sind; sie bilden das *Metamodell*. Die Graphmodelle beschreiben eine Menge von Graphen, genauer legen sie fest, welche Typen und Attribute die einzelnen konkreten Graphelemente haben *können*, ohne allerdings dies für eine tatsächliche Ecke und Kante zu tun. Die Abbildungen typ_X und val_X hingegen sind von Graph zu Graph anders definiert, drücken sie doch aus, welchen Typ einzelne konkrete Graphelemente haben und welchen Wert ihre Attribute tragen. Die Graphen, die zu einem Graphmodell gehören, nennen wir auch *Instanzen* desselben.[2]

Definition 4.3 (Instanzen eines Graphmodells) Sei M ein Graphmodell, dann ist $\mathcal{I}_M$ die Menge aller Instanzen dieses Graphmodells.

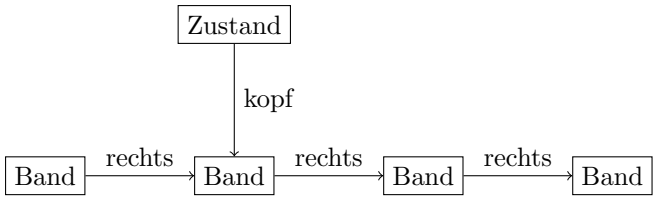
4.1.2 Beispiel

Wir geben nun ein Beispiel für ein Graphmodell an, welches uns – vervollständigt um weitere Aspekte – in den folgenden Abschnitten immer wieder begegnen wird. Wie wir noch detaillierter erörtern werden, entsprechen die Instanzen dieses Graphmodells einer grafischen Darstellung der Konfiguration einer Turingmaschine; darum nennen wir das Graphmodell M_{TM}.

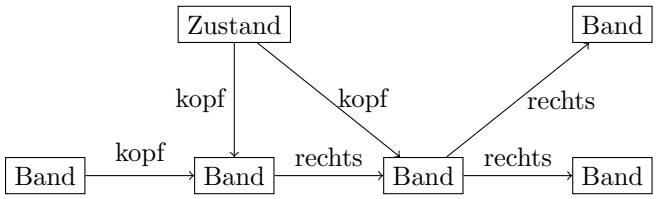
$$
\begin{aligned}
T_E &= \{\bot_E, \text{Band}, \text{Zustand}, \text{Stopp}\} \\
T_K &= \{\bot_K, \text{kopf}, \text{rechts}\} \\
\leq_E &= \{(\bot_E, \text{Band}), (\bot_E, \text{Zustand}), (\text{Zustand}, \text{Stopp})\} \\
\leq_K &= \{(\bot_K, \text{kopf}), (\bot_K, \text{rechts})\} \\
A_E &= \{\text{bu}, \text{q}\} \\
A_K &= \{\} \\
S_E &= \{\text{string}, \text{int}\} \\
S_K &= \{\} \\
\text{attr}_E(\text{Band}) &= \{\text{bu}\} \\
\text{attr}_E(\text{Zustand}) &= \{\text{q}\} \\
\text{attrtyp}_E(\text{bu}) &= \text{string} \\
\text{attrtyp}_E(\text{q}) &= \text{int}
\end{aligned}
$$

In Abbildung 4.1 zeigen wir zwei Instanzen des obigen Graphmodells. Die Abbildung ist als informelle Darstellung des formal definierten Graphen zu sehen. Die Graphelemente haben als Markierung den jeweiligen Typ, nicht etwa ein Element aus E oder K, wie das bei normalen Graphen üblich ist (vgl. etwa Definition 2.8). Weiterhin lässt die hier gewählte Darstellungsweise z. B. die Attribute außen vor.

[2] Wenn wir einen Vergleich mit objektorientierten Programmierung anstellen wollen, so entsprechen Graphmodelle den Klassen(-diagrammen). Die Instanzen eines Graphmodell, also die Graphen, entsprechen in der objektorientierten Welt den Objekten bzw. Objektgraphen.



(a) Graph $G_{TM}^{(1)}$ *mit* Entsprechung



(b) Graph $G_{TM}^{(2)}$ *ohne* Entsprechung

Abbildung 4.1: Zwei Abbildungen von Instanzen des Graphmodells M_{TM} mit und ohne entsprechender Turingmaschinenkonfiguration.

Der Graph $G_{TM}^{(1)}$ aus Abbildung 4.1(a) ist formal gegeben durch:

$$
\begin{aligned}
G_{TM}^{(1)} &= (E, K, \mathrm{src}, \mathrm{tgt}, \mathrm{typ}_E, \mathrm{typ}_K, \mathrm{val}_E, \mathrm{val}_K, M_E, M_K) \\
E &= \{e_1, e_2, e_3, e_4, e_5\} \\
K &= \{k_1, k_2, k_3, k_4\} \\
\mathrm{src} &: k_1 \mapsto e_1, k_2 \mapsto e_2, k_3 \mapsto e_3, k_4 \mapsto e_4 \\
\mathrm{tgt} &: k_1 \mapsto e_3, k_2 \mapsto e_3, k_3 \mapsto e_4, k_4 \mapsto e_5 \\
\mathrm{typ}_E &: e_1 \mapsto \text{Zustand}, \{e_2, e_3, e_4, e_5\} \mapsto \text{Band} \\
\mathrm{typ}_K &: k_1 \mapsto \text{kopf}, \{k_2, k_3, k_4\} \mapsto \text{rechts} \\
\mathrm{val}_E &: (e_1, \mathrm{q}) \mapsto 0, (e_2, \mathrm{bu}) \mapsto \text{"\#"}, (e_3, \mathrm{bu}) \mapsto \text{"a"}, \\
&\quad (e_4, \mathrm{bu}) \mapsto \text{"b"}, (e_5, \mathrm{bu}) \mapsto \text{"\#"} \\
\mathrm{val}_K &: \uparrow\uparrow
\end{aligned}
$$

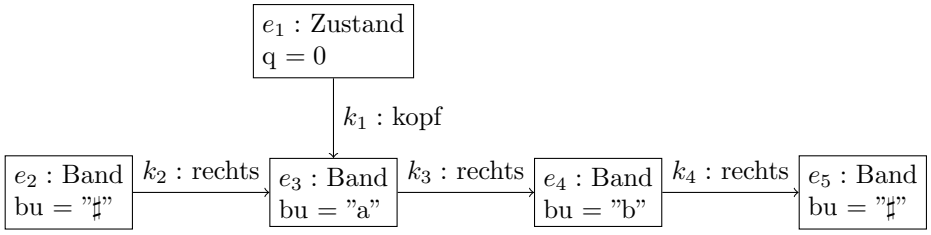


Abbildung 4.2: $G_{TM}^{(1)}$ dargestellt als benannter Graph

Wird ein Graph formal angegeben, ist dies offenbar bei weitem unanschaulicher als eine entsprechende Abbildung. Darum werden wir die Graphen im Folgenden meist in Form von Abbildungen angeben, und uns der *benannten Graphen* bedienen, wenn wir detaillierte Informationen repräsentieren wollen. Die *Markierungen* der Graphelemente benannter Graphen sind dabei wie folgt zusammengesetzt: Zuerst steht der Ecken- bzw. Kantenidentifikator, eine textuelle Darstellung des mathematischen Objektes $x \in E \cup K$. Durch einen Doppelpunkt davon abgetrennt steht der

Typ und ggf. die Attributnamen und -werte in jeweils neuen Zeilen. Abbildung 4.2 zeigt den Graphen $G^{(1)}_{TM}$ dargestellt als benannten Graphen.

4.1.3 Validieren von Graphen

Wenn wir Abbildung 4.1 betrachten, fällt auf, dass wir eigentlich nur Graphen, wie die in Abbildung 4.1(a) als Darstellung einer Turingmaschine ansehen können. Die Abbildung 4.1(b) zeigt eine „Turingmaschine“ mit verzweigendem Band, das an einer Stelle mit einer Kante des falschen Types verkettet ist und einem Kopf, der auf zwei Bandpositionen gleichzeitig steht.

Um solche, meist aus der Anwendungsdomäne heraus, als falsch zu betrachtende Graphen auszuschließen, führen wir sogenannte *Verbindungszusicherungen* ein. Diese Zusicherungen bereichern unsere Graphmodelle um Nebenbedingungen, die gewisse unerwünschte Kanten ausschließen und andere notwendige Kanten erzwingen. Das Prüfen dieser Zusicherungen nennen wir *Validieren* des Graphen mittels Verbindungszusicherungen. Diese Prüfung ist optional; sie kann entweder bei jeder Änderung des Graphen oder zu gewissen Zeitpunkten oder auch gar nicht stattfinden. Wann und ob diese Prüfung stattfinden soll, ist dem Anwender respektive den Anwendungsprogrammen überlassen.

Andere Formalismen aus der einschlägigen Literatur kennen auch ähnliche Beschränkungen der zulässigen Graphen, aber keiner ist so flexibel, dass Graphen diese Zusicherungen auch (zumindest temporär) verletzen können. Gerade diese Fähigkeit ist entscheidend, wenn man komplexe Transformationen durch mehrere hintereinander angewendete Graphersetzungsregeln durchführt. Hat man in einem solchen Fall nicht die Möglichkeit, zumindest temporär auf die Erfüllung der Verbindungszusicherungen zu verzichten, muss man entweder die Regel künstlich komplex gestalten oder die Verbindungszusicherungen selbst lockern. Letzteres führt dazu, dass die Verbindungszusicherungen auch dann noch weniger streng sind, wenn das eigentlich gar nicht mehr nötig wäre; somit wird der validierende Charakter von Verbindungszusicherungen pervertiert. Über die Möglichkeiten der Verbindungszusicherung hinausgehend ist das Konzept der *Validierungssequenz* (siehe Definition 4.5), das eine turingvollständige Validierung erlaubt; es ist bei keinem anderen uns bekannten Ansatz vorhanden.

Definition 4.4 (Verbindungszusicherungen) Eine Menge von 7-Tupeln

$$Z \subset T_K \times T_E \times \mathbb{N} \times \mathbb{N} \times T_E \times \mathbb{N} \times \mathbb{N}$$

heißt *Verbindungszusicherungen*, wobei die Zusicherung z

$$z = (\tau, \tau_s, n_s^\perp, n_s^\top, \tau_t, n_t^\perp, n_t^\top) \in Z$$

genau dann erfüllt ist, wenn

$$\begin{aligned} \forall x \in E:\ n_s^\perp \leq \big|\{k \in K | x = \mathrm{src}(k) \wedge \mathrm{typ}_E(x) \geq_E \tau_s \wedge \\ E \ni y := \mathrm{tgt}(k) \wedge \mathrm{typ}_E(y) \geq_E \tau_t \wedge \\ \tau =_K \mathrm{typ}_K(k)\}\big| \leq n_s^\top \end{aligned} \tag{4.5}$$

und

$$\begin{aligned} \forall y \in E:\ n_t^\perp \leq \big|\{k \in K | E \ni x := \mathrm{src}(k) \wedge \mathrm{typ}_E(x) \geq_E \tau_s \wedge \\ y = \mathrm{tgt}(k) \wedge \mathrm{typ}_E(y) \geq_E \tau_t \wedge \\ \tau =_K \mathrm{typ}_K(k)\}\big| \leq n_t^\top \end{aligned} \tag{4.6}$$

erfüllt sind, wobei die Ecken- bzw. Kantentypgleichheit $=_X$ als $u =_X v :\Leftrightarrow u \leq_X v \wedge v \leq_X u$ vereinbart sei. Die Menge Z heißt *erfüllt*, genau dann, wenn jedes $z \in Z$ erfüllt ist; der Graph ist dann valide bezüglich der Verbindungszusicherungen Z.

Mit Verbindungszusicherungen Z kann somit ein Graphmodell $M = (M_E, M_K)$ erweitert werden zu $M' = (M_E, M_K, Z)$.

Verbindungszusicherungen sind intuitiv gesehen genau dann erfüllt, wenn Kanten eines Typs τ an gewissen Ecken in der „richtigen“ Anzahl auftreten. Bedingung 4.5 betrachtet ausgehende Kanten k an Ecken x. Der Typ dieser Ecken muss ein Untertyp[3] von τ_s sein und außerdem muss k an Ecken y, deren Typ ein Untertyp von τ_t ist, enden. Von solchen Kanten k darf es pro Ecke x zwischen $n_s^\perp$ und $n_s^\top$ Stück geben. Bedingung 4.6 fordert analoges für eingehende Kanten. Eine Verbindungszusicherung $(\tau, \tau_s, \cdot, \cdot, \tau_t, \cdot, \cdot)$ legt mögliche Kanten(-grade) für genau einen Kantentyp τ fest; Untertypen von τ werden davon nicht berührt. Um in GrGen.NET auch Verbindungszusicherungen entlang der Vererbung der Kantentypen zu erlauben, gibt es das Schlüsselwort `inherit`, das die Zusicherungen von den unmittelbaren Obertypen übernimmt. Im Gegensatz dazu wird zur Prüfung der inzidenten Eckentypen der Kante (also τ_s, τ_t) die Vererbungsbeziehung mit betrachtet.

Betrachten wir die Definition der Erfüllbarkeit von Verbindungszusicherungen genauer, so fällt auf, dass für Kantentypen, die weder selbst, noch vertreten durch einen ihrer Obertypen in einer Zusicherung auftreten, nichts ausgesagt ist; solche Kanten dürfen also an beliebiger Stelle auftauchen. In der Regel wollen wir das umgekehrte Verhalten – nur explizit erlaubte Kanten sollen zulässig sein. Dazu benutzen wir die *strikten Verbindungszusicherungen*, bei der die Menge von Zusicherungen nur dann erfüllt ist, wenn zusätzlich alle Kanten durch mindestens eine Zusicherung abgedeckt wird. Es sei noch angemerkt, dass sich umgekehrt auch zwei oder mehr Zusicherungen, die für ein und dieselbe Kante passen, formulieren lassen sind. Solcherlei mehrfache Abdeckung von Kanten kann dazu führen, dass die Menge der Zusicherungen nie (sprich von keinem Graphen) erfüllbar ist.

Betrachten wir noch einmal unser Beispielgraphmodell M_{TM}. Wir wünschen uns eine Kette aus über `rechts`-Kanten verbundenen `Band`-Ecken. Mit dem `kopf` soll nur auf eine `Band`-Ecke gezeigt werden; `kopf`- und `rechts`-Kanten sollen an keinen weiteren Stellen auftreten. Das Gewünschte erreichen wir durch folgende Verbindungszusicherungen:

$$Z_{TM} = \{(\text{kopf}, \text{Zustand}, 1, 1, \text{Band}, 0, 1), (\text{rechts}, \text{Band}, 0, 1, \text{Band}, 0, 1)\}$$

Somit haben wir nun ein abgeändertes Graphmodell $M'_{TM} = (M_E, M_K, Z_{TM})$ für unsere Turingmaschine, wobei M_E und M_K die unveränderten Typ- und Attributierungsmodelle aus Abschnitt 4.1.2 sind. Hierdurch werden Graphen wie in Abbildung 4.1(b) wunschgemäß ausgeschlossen. Allerdings ist klar, dass nicht alle bezüglich der jeweiligen Anwendung unsinnigen Graphen dadurch verhindert werden können. Zum Beispiel verletzt der Graph in Abbildung 4.3 die Verbindungszusicherungen nicht, kann jedoch genauso wenig als sinnvolle Turingmaschine interpretiert werden. Verbindungszusicherungen sind also nur notwendige Bedingungen an (aus Sicht der Anwendung) korrekte Graphen. Um beliebige Bedingungen in Bezug auf

[3]Der Typ selbst ist in unserer Sprechweise immer auch sein eigener Untertyp.

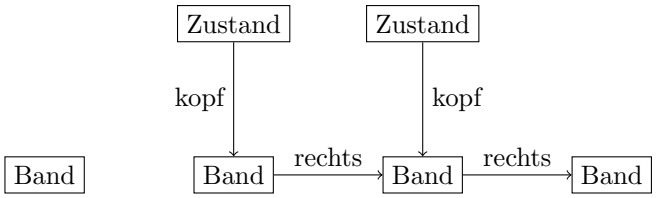


Abbildung 4.3: Unsinnige Instanz des Graphmodells M_{TM} und M'_{TM}.

die Struktur der Graphen zu prüfen, benötigt man im Allgemeinen einen turingvollständigen Formalismus[4].

Einen ebensolchen turingvollständigen Formalismus stellen wir mit einer auf erweiterten Graphersetzungssequenzen[5] basierenden Validierungsmethode zur Verfügung. Diese beruht auf der Idee, dass wir den jeweiligen zu validieren Graphen von einer Graphersetzungssequenz prüfen lassen, die wir dann *Validierungssequenz* nennen. Wir wollen den Graph genau dann als valide ansehen, wenn die Graphersetzungssequenz erfolgreich beendet wurde. Bei der Abarbeitung der Sequenz darf der zu validierende Graph beliebig verändert werden, da zuvor ein Schnappschuss erstellt wurde, der nach der Validierung wiederhergestellt wird. Auf diese Weise kann man unter anderem auch eine auf *Graphgrammatiken* (genauer den *Graphparsern*) basierende Validierung benutzen.

Definition 4.5 (Validierungssequenz) Sei eine Graphersetzungssequenz s nach Definition 4.22 gegeben. Dann ist der Arbeitsgraph H genau dann gemäß der *Validierungssequenz* s valide, wenn

$$[\![\mathtt{s}]\!](H, \Uparrow\uparrow, \top) =: (H', \mathrm{val}, z)$$

mit $z = \top$ ausgewertet wird.

4.1.4 Notation

Der Graphersetzungsgenerator GrGen nimmt die Graphmodelle als textuelle Spezifikationen entgegen. Diese enthält alle Informationen, die in Definition 4.2 für Graphmodelle gefordert sind. In Programm 4.1 ist das Beispielgraphmodell M_{TM} in GrGen-Schreibweise angegeben. Die komplette Grammatik der Spezifikationssprache findet sich in Anhang A. Verbindungszusicherungen werden mit dem Schlüsselwort `connect` eingeleitet und sind der jeweiligen Kante zugeordnet. Die Syntax der Sprache orientiert sich an Java und Pascal. Die GrGen-Schreibweise liegt nahe bei den jeweiligen mathematischen Konstrukten. Wir verweisen darum hier nur auf die Kurzreferenz in Anhang A.2 und eine detaillierte Beschreibung im Benutzerhandbuch [BG07] von GrGen.NET.

[4]Ein Vergleich mit konventionellen Programmiersprachen: Ein korrekt typisiertes Programm kann dennoch (aus Sicht der Anwendung) fehlerhaft sein; genauso verhält es sich mit Verbindungszusicherungen.

[5]Graphersetzungssequenzen sind Sequenzen von Regelanwendungen, die vermittels einer auf booleschen Ausdrücken und Iteration basierenden Spezifikationssprache angegeben werden (vgl. Abschnitt 4.4).

Programm 4.1: Graphmodell der Turingmaschine M'_{TM}

```
1  node class Band {
2    bu : string;
3  }
4
5  node class Zustand {
6    q : int;
7  }
8
9  node class Stopp extends Zustand;
10
11 edge class kopf
12             connect Zustand[1] -> Band[0:1];
13
14 edge class rechts
15             connect Band[0:1] -> Band[0:1];
```

4.2 Graphmuster

Der aus Laufzeitgesichtspunkten wichtigste Teil der Graphersetzung ist das Mustersuchen. Wie wir in Abschnitt 3.2 gesehen haben, befassen sich die meisten Veröffentlichungen leider nicht mit diesem Thema. Bei praktischen Arbeiten wird die Skalierbarkeit und Größe der bearbeitbaren Muster und Graphen selten überhaupt erwähnt, geschweige denn eingehend behandelt. Darum beschäftigen wir uns im nächsten Kapitel ausführlich mit der Effizienz des Mustersuchers. Zuerst definieren wir hier, wann in einem GR-Graphen ein Muster als gefunden gelten soll.

Wir erweitern hierzu die Definition 2.23 der Passung so, dass *typisierte Graphen* (Abschnitt 4.2.1) mit *dynamischen Typbedingungen* (Abschnitt 4.2.3), *Attributbedingungen* (Abschnitt 4.2.2), *Vorbelegungen* (Abschnitt 4.2.4), *negative Anwendungsbedingungen* (NAC, Abschnitt 4.2.5) und *Homomorphiebedingungen* (Abschnitt 4.2.7), sowie *dynamische Muster* (Abschnitt 4.2.8) berücksichtigt werden. Negative Anwendungsbedingungen werden benötigt, um die Zulässigkeit von Ersetzungen im Graphen vom Nichtvorhandensein bestimmter Aspekte in ebendiesem Graphen abhängig zu machen. Attributbedingungen erlauben, das Passen eines Musters zusätzlich, also über die reine Graphstruktur hinaus, von Attributwerten abhängig zu machen. Mittels Vorbelegen einer Passung kann die Mustersuche, falls dieses Vorwissen vorhanden ist, auf gewisse Stellen des Graphen eingeschränkt werden. Ecken oder Kanten können mittels der Homomorphiebedingungen entweder injektiv oder *nicht* (notwendigerweise) injektiv abgebildet werden. Schließlich sind dynamische Muster eine mächtige Erweiterung des traditionellen Begriffs des Graphmusters hin zu dynamischen Strukturen.

Die Neuerungen bei den Mustern liegen auf zweierlei Ebenen: Zum einen haben wir ein neues einheitliches Beschreibungsrahmenwerk basierend auf denotationeller Semantik und Graphhomomorphismen eingeführt, das die verschiedenen Aspekte eines angereicherten Musters uniform spezifizierbar macht. Zum anderen sind einige der Aspekte des angereicherten Musters selbst neu, so z. B. die für einzelne Ecken *und vor allem* Kanten vereinbarten Homomorphiebedingungen, die nun erstmals

kaskadierbaren negativen Anwendungsbedingungen. Neu sind auch die dynamischen Typbedingungen, die komplexe Beziehungen zwischen den Typen verschiedener gepasster Arbeitsgraphelemente ausdrückbar machen. Von zentraler Bedeutung für die Formalisierung von negativen Anwendungsbedingungen – die eleganter als bisher bekannt vonstattengeht – aber auch für die bessere Integration von einzelnen deklarativen Regelanwendungen in imperative Anwendungen ist der *Vorbelegungsmorphismus* zusammen mit dem Begriff des *Arguments* eines Graphmusters.

4.2.1 Mustergraphen und Graphhomomorphismen

Die zentrale Rolle bei der Mustersuche kommt dem Passungsmorphismus m zu. Wir werden die klassische Definition in mehreren Schritten auf die von uns benötigte Ausdrucksstärke heben. Um die Menge aller Passungsmorphismen zwischen einem Mustergraphen und einem Arbeitsgraphen zu charakterisieren, definieren wir mehrere jeweils aufeinander aufbauende *match-Funktoren*.

Mit Definition 4.2 haben wir den konventionellen Begriff des Graphen aus Definition 2.8 hin zu gerichteten, typisierten und attributierten Multigraphen erweitert. Nunmehr sind Graphen nicht länger 4-Tupel wie

$$(E, K, \mathrm{src}, \mathrm{tgt})$$

sondern 10-Tupel

$$(E, K, \mathrm{src}, \mathrm{tgt}, \mathrm{typ}_E, \mathrm{typ}_K, \mathrm{val}_E, \mathrm{val}_K, M_E, M_K)$$

wobei $M = (M_E, M_K)$ für das Graphmodell steht, welches wiederum jeweils für Ecken und Kanten ein 6-Tupel ist, also

$$M_X = (T_X, \leq_X, A_X, S_X, \mathrm{attr}_X, \mathrm{attrtyp}_X)$$

Bei aller Analogie der beiden Definitionen, sei hier angemerkt, dass wir in Definition 4.2 nicht wie in Definition 2.8 gefordert haben, dass src und tgt total sein müssen. Damit ist es möglich auch losgelöste oder einseitig mit einer Ecke verbundene Kanten darzustellen. Allerdings wird dies nur in Mustern oder von Passungen überdeckten Teilen des Arbeitsgraphen benötigt; der Arbeitsgraph selbst soll immer totale src- und tgt-Funktionen haben. Ferner fordern wir aus Gründen der Annehmlichkeit, dass stets $E \cap K = \emptyset$ sei. Auf diese Weise sind Elemente aus $E \cup K$ eindeutig als Kante oder Ecke identifizierbar.

Allerdings ist die Definition 2.14 des Graphhomomorphismus noch nicht entsprechend nachgezogen. Wann wollen wir jedoch in diesem neuen Szenario überhaupt sagen, dass zwischen zwei Graphen L und H ein Homomorphismus besteht? Zunächst fordern wir, dass das Graphmodell von L und H dasselbe[6] ist. Wenn wir die Attributierung und Typisierung außen vor lassen, dann soll natürlich der Homomorphiebegriff unverändert bleiben. Die Typisierung schränkt die zunächst freie Abbildbarkeit von Graphelementen ein: Es sollen Elemente in L nur auf entsprechende Elemente in H abgebildet werden können, wenn diese denselben oder einen

[6]Zwar könnte man auch Homomorphie zwischen unterschiedlichen Graphmodellen eine gewisse Sinnhaftigkeit zusprechen, aber wir wollen uns nicht auf diese Meta-Ebene begeben. Schließlich kann man zwei Graphmodelle immer vereinigen, wobei man ggf. Umbenennungen vornehmen muss, und so zu einem gemeinsamen Modell kommen.

Untertyp des Urelements in L aufweisen. Die Attributierung wird anders gehandhabt, da wir komplexe Attributberechnungen zulassen wollen, sodass insbesondere mehrere Attribute verschiedener Graphelemente miteinbezogen werden können (siehe Abschnitt 4.2.2). Somit nehmen wir die Attribute vom zentralen Homomorphiebegriff aus und werden hinterher die Attributbedingungen prüfen. Formal führt das zu folgender Form für gerichtete, typisierte und attributierte Muster- und Arbeitsmultigraphen, kurz GR-Graphen:

$$L = (E_L, K_L, \mathrm{src}_L, \mathrm{tgt}_L, \mathrm{typ}_{E_L}, \mathrm{typ}_{K_L}, \Uparrow, \Uparrow, M_E, M_K) \tag{4.7}$$
$$H = (E_H, K_H, \mathrm{src}_H, \mathrm{tgt}_H, \mathrm{typ}_{E_H}, \mathrm{typ}_{K_H}, \mathrm{val}_{E_H}, \mathrm{val}_{K_H}, M_E, M_K) \tag{4.8}$$

Da Mustergraphen – aus den oben genannten Gründen – keine Attributwerte tragen, ist ihre Attributierungsfunktion immer die total undefinierte Funktion. Wir betrachten im Folgenden einen Homomorphismus $m = (m_E, m_K)$ mit m_E der Eckenabbildung und m_K der Abbildung für die Kanten. Die Homomorphiebedingungen sind mit den obigen Überlegungen gegeben durch

$$\forall e \in \mathrm{def}(m_E) : m_K(\mathrm{src}_A(e)) = \mathrm{src}_B(m_E(e)) \tag{4.9}$$
$$\forall e \in \mathrm{def}(m_E) : m_K(\mathrm{tgt}_A(e)) = \mathrm{tgt}_B(m_E(e)) \tag{4.10}$$
$$\forall e \in \mathrm{def}(m_E) : \mathrm{typ}_{E_L}(e) \leq_E \mathrm{typ}_{E_H}(m_E(e)) \tag{4.11}$$
$$\forall k \in \mathrm{def}(m_K) : \mathrm{typ}_{K_L}(k) \leq_K \mathrm{typ}_{K_H}(m_K(k)) \tag{4.12}$$

wobei die ersten beiden Formeln die konventionelle Strukturgleichheit fordern, während die letzten beiden Formeln das Passen der Typen sicherstellen. Mit diesem Homomorphiebegriff seien nun die Mengen $\mathbb{T}, \mathbb{P}$ sowie ihre Verfeinerungen analog auf GR-Graphen erweitert (vgl. Definition 2.17).

Im einfachsten Fall hat der match-Funktor folgende Signatur, die wir in den folgenden Abschnitten um weitere Bedingungen anreichern:

$$\mathrm{match} : \mathcal{I}_M \times \mathcal{I}_M \to \mathbb{T}(\mathcal{I}_M, \mathcal{I}_M)$$

Der Anschein, dass der Bildbereich zu allgemein gewählt ist, da er alle Homomorphismen zwischen zwei Graphen mit Graphmodell M einschließt, trügt, denn wir werden zwar zunächst nur einschränkende Bedingungen an den match-Funktor definieren, aber die dynamischen Muster (Abschnitt 4.2.8) erweitern den Urbildbereich der Morphismen quasi über L hinaus.

Definition 4.6 (match-Funktor) Seien $L, H \in \mathcal{I}_M$ dann ist

$$\mathrm{match}(L, H) \mapsto \mathbb{T}(L, H)$$

Der elementare match-Funktor ist vollständig durch den Homomorphiebegriff bestimmt, enthält er doch genau die totalen (die Typisierung respektierenden, vgl. Formeln 4.11 und 4.12) Homomorphismen zwischen L und H.

4.2.2 Attributbedingungen

Die Attributbedingungen bilden ausgehend von Attributen boolesche Ausdrücke, die in Syntax und Semantik denen von Java gleichkommen. Allerdings sind keine Zuweisungen oder andere nichtfunktionale Konstrukte erlaubt. Darüber hinaus

bilden neben den üblichen booleschen, ganzzahligen, Gleitkomma- und textwertigen Konstanten (wie `true, false, 0, 42, 8.15, "Ich bin ein Berliner"`) nur Attributzugriffe von Graphelementen (wie `ecke.attribut`) die Blätter des Ausdrucksbaumes. Wir wollen hier keiner überschießenden Formalisierung Vorschub leisten, indem wir hier selbst die Ausdrucksauswertung formalisieren, und verweisen darum auf den Sprachbericht von Java [GJSB05], und setzen eine der üblichen Formalisierungen der Ausdrücke in denotationeller Semantik[7] voraus. Da wir keinerlei Rekursion, Schleifen sowie andere mittels der einfachen und eleganten denotationellen Mechanismen schwieriger fassbaren Konstrukte verwenden, reicht dieser Formalismus vollkommen aus.

Formal betrachtet wird eine Attributbedingung c durch eine Auswertungsfunktion

$$\text{eval}[\![\mathsf{c}]\!] : \mathbb{T}(\mathcal{I}_M, \mathcal{I}_M) \to \mathbb{B}$$

ausgewertet, die einen Homomorphismus $m \in \mathbb{T}(L, H)$ in den Bereich der Wahrheitswerte abbildet. Die denotationelle Auswertungsfunktion $\text{eval}[\![\cdot]\!]$ muss, über die erwähnten Java-ähnlichen Ausdrücke hinaus, noch für Ecken- und Kantenattributzugriffe definiert werden:

$$\text{eval}[\![\mathtt{x.a}]\!]_m = \begin{cases} \text{val}_E(m(x), a) & \text{wenn } x \in E \\ \text{val}_K(m(x), a) & \text{wenn } x \in K \end{cases},$$

wobei $x \in X$ mit $X = E$ oder $X = K$ und $a \in \text{attr}_X(\text{typ}_X(x))$ gelte. Hierfür werden die Attribute mittels val_X ausgelesen, wobei zuerst jenes, dem Mustergraphelement x durch den Homomorphismus m zugeordnetes tatsächliches Arbeitsgraphelement $m(x)$ ermittelt werden muss. Wenn das Muster L im Graphen H vermittels des Morphismus m gefunden wird, soll diese Fundstelle ungültig sein, falls die Attributbedingung c diese ablehnt, also falls $\text{eval}[\![\mathsf{c}]\!]_m = \bot$ ist.

Mit diesem Wissen können wir nun den match-Funktor um Attributbedingung erweitern:

Definition 4.7 (match-Funktor mit Attributbedingung) Seien $L, H \in \mathcal{I}_M$ und c eine Attributbedingung, dann ist

$$\text{match}(L, H, c) \mapsto \{m \in \text{match}(L, H) \mid \text{eval}[\![\mathsf{c}]\!]_m = \top\}$$

Wenn wir an eine Passung mehrere Attributbedingungen stellen wollen, können diese einfach konjunktiv zu einer Bedingung verknüpft werden. Somit stellt es keine Einschränkung dar, dass wir nur eine Attributbedingung pro Passung erlauben.

4.2.3 Dynamische Typbedingungen

Eine zur Vereinfachung von Regeln besonders nützliche Eigenschaft sind *dynamische Typbedingungen*. Damit ist es möglich den zulässigen Typ eines Musterelements vom Typ des Arbeitsgraphelements, mit dem ein anderes Musterelement zur Passung gebracht wurde, abhängig zu machen. Der Wunsch nach einer solchen Einschränkung

[7] Denotationelle Semantik wurde in den Siebzigern von Scott und Strachey in Oxford entwickelt und erstmals durch die Monografie von Stoy [Sto77] sowie dem Artikel von Tennent [Ten76] einer breiteren Öffentlichkeit dargelegt.

wird durch eine dynamische Typbedingung, die den `typeof`-Operator verwendet, ausgedrückt. Sie ist immer dann nötig, wenn nicht alle Untertypen eines Typs aus Sicht des Anwenders für eine Passung geeignet sind, insbesondere dann, wenn noch Beziehungen zwischen den Typen der gepassten Arbeitsgraphelemente erfüllt sein müssen. Technisch geschieht dies als Attributbedingung, wobei eben keine Attribute ausgewertet werden, sondern der Typ der gepassten Arbeitsgraphelemente. Wir müssen also Definition 4.7 nicht abändern, sondern lediglich die Auswertungsfunktion erweitern.

Dynamische Typbedingungen werden mit der denotationellen Auswertungsfunktion $\text{eval}[\![\cdot]\!]$ in Wahrheitswerte überführt. Im Einzelnen gilt

$$\text{eval}[\![\texttt{typeof(x) <= typeof(y)}]\!]_m = \begin{cases} \text{typ}_E(m(x)) \leq \text{typ}_E(m(y)) & \text{wenn } x \in E \wedge y \in E \\ \text{typ}_K(m(x)) \leq \text{typ}_K(m(y)) & \text{wenn } x \in K \wedge y \in K \\ \bot & \text{sonst} \end{cases}$$

Analoges gilt für die restlichen Vergleichsoperatoren. Es gibt noch die Möglichkeit direkt mit einem Typen zu vergleichen, wobei natürlich wieder alle üblichen Vergleichsoperatoren erlaubt sind, also:

$$\text{eval}[\![\texttt{typeof(x) <= t}]\!]_m = \begin{cases} \text{typ}_E(m(x)) \leq t & \text{wenn } x \in E \wedge t \in T_E \\ \text{typ}_K(m(x)) \leq t & \text{wenn } x \in K \wedge t \in T_K \\ \bot & \text{sonst} \end{cases}$$

4.2.4 Morphismen mit Vorbelegungen

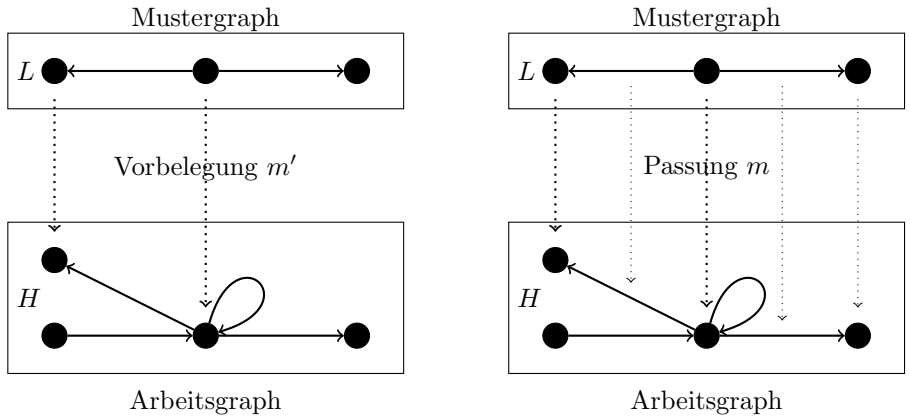
Normalerweise wird sich die Lösung eines Problems nicht in einer einzigen Anwendung einer einzigen Regel fassen lassen, sondern es werden (mehrere) Regeln hintereinander Anwendung finden. Oft werden solche Regeln nicht losgelöst voneinander eingesetzt, sondern die vorangehende Regelanwendung wird *Aufpunkte* für die nachfolgenden finden oder gar schaffen. Wenn wir Regeln auf diese Weise anwenden wollen, dann kann es von Vorteil sein – im imperativen Sinne – gewisse Graphelemente als Parameter an diese Regeln zu übergeben. Die Graphmustersuche muss dann diese Graphelemente berücksichtigen. Dadurch kann es möglich sein, auf sogenannte Schiffchen zur Ablaufsteuerung zu verzichten oder zumindest unter Laufzeitgesichtspunkten positive Effekte zu erzielen.

Definition 4.8 (match-Funktor mit Vorbelegung) Seien $L, H \in \mathcal{I}_M$, c eine Attributbedingung und $m' \in \mathbb{P}(L, H)$, dann ist

$$\text{match}(L, H, c, m') \mapsto \{m \in \text{match}(L, H, c) \mid m' \trianglelefteq m\}$$

mit m' als *Vorbelegung*.

Ein *Vorbelegungsmorphismus* m' legt also immer eine gewisse Anzahl Elemente (genauer gesagt $|\text{def}(m')|$ Stück) des Mustergraphen schon im vorhinein fest. Abbildung 4.4 stellt einen Vorbelegungsmorphismus und eine daraus erweiterbare Passung dar, wobei auf Typen und andere Einschränkungen nicht eingegangen wird.



(a) Vorbelegungsmorphismus m'

(b) Erweiterung eines Vorbelegungsmorphismus m' zu einer Passung m

Abbildung 4.4: Passung $m \in \text{match}(L, H, \cdot, m')$ mit Vorbelegung m'

4.2.5 Negative Anwendungsbedingungen

Mit *negativen Anwendungsbedingungen*[8] wollen wir beim Finden von L gewisse Muster N_i nicht in H finden. Überdies soll, um die Anknüpfung mit dem eigentlich zu suchenden Muster L herzustellen, ein partieller injektiver Homomorphismus $l_i \in \mathbb{P}_{\mathbb{INJ}}(L, N_i)$ zwischen dem zu-findenden und dem nicht-zu-findenden Muster vorhanden sein. Auf diese Weise ist es möglich, negative Kontexte zu formulieren, denn oft ist es erheblich leichter zu sagen, dass es gewisse Ecken bzw. Kanten nicht geben darf, als alle gültigen Kontexte aufzuzählen. Auch das Finden einer NAC kann mit Attributbedingungen c_i eingeschränkt werden.

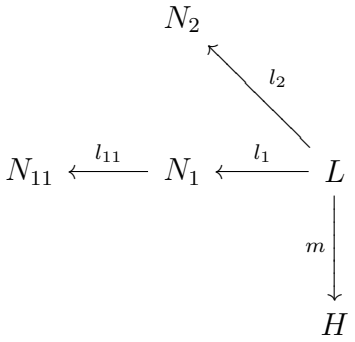


Abbildung 4.5: Morphismusdiagramm mit geschachtelten NACs

Wir haben eine Situation bei mehreren (und *geschachtelten*) negativen Anwendungsbedingungen in Abbildung 4.5 aufgetragen. Die Tatsache, dass die Einbettung der NACs jeweils nur über eine Stufe reicht, ist, wie wir in Abschnitt 4.2.10 sehen werden, keine Einschränkung. Es ist also nicht nötig, einen weiteren Morphismus von L direkt nach N_{11} gehen zu lassen, da die indirekte Einbettung über l_1 und l_{11} zusammen mit geeigneten Homomorphiebedingungen (siehe Abschnitt 4.2.7) ausreichen, dies zu simulieren.

[8] engl.: *negative application condition* (NAC)

Definition 4.9 (match-Funktor mit NACs) Seien $L, H \in \mathcal{I}_M$ Graphen, c eine Attributbedingung und $m' \in \mathbb{P}(L, H)$ ein Vorbelegungsmorphismus. Weiterhin seien $N_i \in \mathcal{I}_M$ Graphen, $l_i \in \mathbb{P}_{\mathbb{INJ}}(L, N_i)$ partielle injektive Homomorphismen, c_i Attributbedingungen mit Attributen aus N_i mit $i \in \{1, \ldots, k\}$, dann nennen wir

$$\mathcal{N} = \{(N_1, l_1, c_1, \mathcal{N}_1), \ldots, (N_k, l_k, c_k, \mathcal{N}_k)\}$$

negative Anwendungsbedingungen, wobei das $\mathcal{N}_i$ seinerseits wieder negative Anwendungsbedingungen enthält. Der Definitionsbereich des Morphismus l' in $(N', l', c', \mathcal{N}') \in \mathcal{N}_i$ ist natürlich entsprechend anzupassen, also $l' \in \mathbb{P}_{\mathbb{INJ}}(N', H)$ und c' ist nun eine Attributbedingung mit Attributen aus N'. Für tiefer verschachtelte NACs gilt Entsprechendes. Dann ist der match-Funktor mit negativen Anwendungsbedingungen wie folgt definiert:

$$\text{match}(L, H, c, m', \mathcal{N}) \mapsto$$
$$\left\{ m \in \text{match}(L, H, c, m') \;\middle|\; \left[\bigcup_{\substack{(N_i, l_i, c_i, \mathcal{N}_i) \in \mathcal{N} \\ \eta_i \in \mathbb{T}(\text{ran}(l_i), H) \\ \eta_i \circ l_i \trianglelefteq m}} \text{match}(N_i, H, c_i, \eta_i, \mathcal{N}_i) \right] = \emptyset \right\}$$

Die negativen Anwendungsbedingungen müssen natürlich nicht immer über mehrere Ebenen geschachtelt auftreten. Falls ein $\mathcal{N}$ die leere Menge ist, dann entspricht $\text{match}(L, H, c, m', \mathcal{N})$ dem match-Funktor ohne NAC, also $\text{match}(L, H, c, m', \emptyset) = \text{match}(L, H, c, m')$; dies ist keine Zusatzbedingung, sondern folgt unmittelbar aus der Eigenschaft der Vereinigung über eine leere Indexmenge.

Diese Formulierung der negativen Anwendungsbedingungen zeichnet sich durch eine elegante Reduktion des Problems auf Vorbelegungen und Erweiterung von Morphismen aus. Ohne NACs gilt die herkömmliche Semantik. Sobald ein NAC, also $(N_i, l_i, c_i, \mathcal{N}_i) \in \mathcal{N}$ vorliegt, muss zusätzlich geprüft werden, ob nicht auch das negative Muster N_i auf H passt. Allerdings kann N_i im Allgemeinen nicht beliebig auf H abgebildet werden: Es wird versucht eine Passung $n_i \in \text{match}(N_i, H, c_i, \eta_i, \mathcal{N}_i)$ unter Berücksichtigung der Attributbedingung c_i und von speziell gewählten Vorbelegungsmorphismen η_i zu finden (siehe Abbildung 4.6).

Die Morphismen n_i sind also Passungsmorphismen des negativen Mustergraphen N_i. Die Bedingung $\eta_i \in \mathbb{T}(\text{ran}(l_i), H)$ fordert, dass alle Elemente, die durch l_i überstrichen werden, auch vorbelegt werden. Durch $\eta_i \circ l_i \trianglelefteq m$ wird erreicht, dass alle Graphelemente aus N_i, die η_i festlegt sowie die entsprechenden Graphelemente aus L gleichermaßen – und zwar so wie durch m vorgegeben – abgebildet werden (der kleine Kreis mit dem starken Rand, der dreimal in Abbildung 4.6 vorkommt). Für jedes m kann somit eindeutig ein η_i bestimmt werden. Es ist wichtig zu erkennen, dass Graphelemente, die von l_i nicht erfasst werden, von m und n_i beliebig abgebildet werden können, da η_i für diese keine Vorgaben macht. Insbesondere können solche Graphelemente aus L und N_i durch m und N_i auch auf dieselben Graphelemente in H abgebildet werden (siehe nochmals Abbildung 4.6: Der überlagerte Bereich des gepunkteten und des gestrichelten Kreis in der Ellipse von H). Der Morphismus l_i, der den Anschluss des negativen Mustergraphen N_i an das positive Muster herstellt, ist

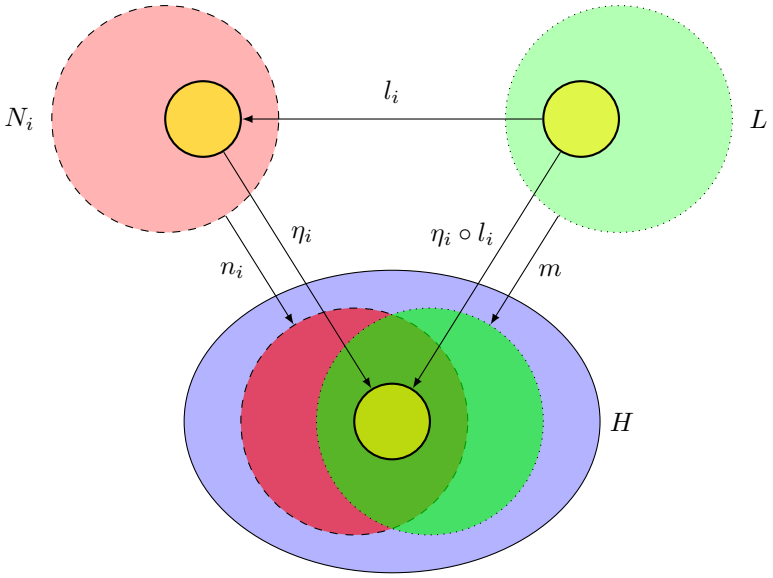


Abbildung 4.6: Mengentheoretische Darstellung der Bild- und Urbildbereiche der Morphismen rund um η_i und seiner definierenden Bedingungen $\eta_i \in \mathbb{T}(\mathrm{ran}(l_i), H)$ und $\eta_i \circ l_i \trianglelefteq m$. In diesem Bild wird offenbar ein n_i gefunden, mithin ist die Menge $\mathrm{match}(N_i, H, c_i, \eta_i, \mathcal{N}_i)$ nicht leer und die Passung m ist zu verwerfen.

injektiv[9], um die Zusammengehörigkeit von Elementen aus L und N_i auszudrücken. Somit wird sichergestellt, dass jene Elemente gleichermaßen von m und n_i in den Arbeitsgraphen abgebildet werden.

Die zentrale Erkenntnis ist, dass negative Anwendungsbedingungen nichts anderes sind als (normale, positive) Muster, die unter Berücksichtigung spezieller Vorbelegungsmorphismen η_i zu finden sind. Diese Darstellung ist deutlich einfacher und allgemeiner als die in der Literatur übliche Darstellung der negativen Anwendungsbedingungen [EHK+99, ERT99, Gru04].

4.2.6 Beispiel

Nach all diesen theoretischen Überlegungen noch zwei konkrete Beispiele mit NACs: Als Erstes betrachten wir, wie das Finden einer Passung im konkreten Fall in der Anwesenheit von zwei NACs aussieht. Als zweites Beispiel untersuchen wir anhand einer gegebenen Aufgabenstellung, auf welche Weise die Ausdrucksstärke der Muster durch NACs zunimmt.

In Abbildung 4.7 sehen wir einen (positiven) Mustergraphen L, bestehend aus drei Ecken und zwei Kanten, für den schon eine Passung m im Arbeitsgraphen H gefunden wurde. Man beachte, dass es möglich ist, dass zwei der Ecken im Mustergraphen nicht injektiv auf ein Element im Arbeitsgraphen abgebildet werden. Die Morphismen l_1 und l_2 stellen für jeweils das gleiche Mustergraphelement eine Verbindung zu den jeweiligen negativen Mustergraphen her – dies muss im Allgemeinen natürlich nicht so sein. Für das linke NAC – gegeben durch N_1 und l_1 – ist schon die (partielle) negative Passung n_1 eingezeichnet. Allerdings kann diese nicht zu einer

[9] Der Morphismus l_i bildet keine zwei Graphelemente aus L auf ein und dasselbe Graphelement in N_i ab.

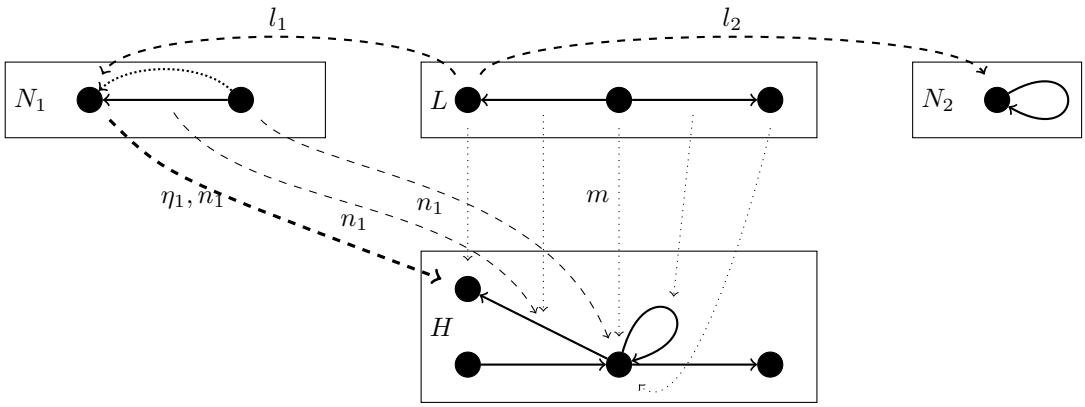


Abbildung 4.7: Passung mit NAC

vollständigen Passung erweitert werden, da die gepunktet gezeichnete Kante von N_1 keine Entsprechung in H hat. Auch wenn man die Passung n_1 verwirft und versucht, eine andere zu konstruieren, gelingt das nicht, da die eine Ecke von N_1 veranlasst durch l_1 immer auf dasselbe Element in H abgebildet werden muss. Genau jene Abbildung, nämlich der Vorbelegungsmorphismus η_1 des NAC, ist durch den dickeren gestrichelten Pfeil markiert. Damit kann das linke NAC nicht zur Ablehnung der Passung m führen.

Für das zweite NAC existiert ebenfalls ein Vorbelegungsmorphismus η_2, der allerdings der Übersichtlichkeit halber nicht eingezeichnet ist. Er muss die einzige Ecke in N_2 auf dasselbe Element abbilden, wie dies auch η_1 getan hat. Es sei hier nochmals betont, dass dies in diesem Beispiel zwar so ist, aber nicht so sein muss, wenn der Urbildbereich von l_1 und l_2 nicht identisch sind. Da aber jene Ecke in H keine Schleife besitzt, kann man das n_2 auch nicht zu einer negativen Passung vervollständigen. Insgesamt führt keines der beiden NACs zur Ablehnung der Passung m.

Wie müssten wir die NACs abändern, wenn wir die Passung ablehnen wollen? Nehmen wir zum Beispiel an, dass l_2 der leere Morphismus sei, dann wäre die Ecke mit Schleife beliebig abzubilden – insbesondere auch auf die zentrale Ecke in H. Damit hätten wir eine negative Passung und die (positive) Passung m wäre abzulehnen.

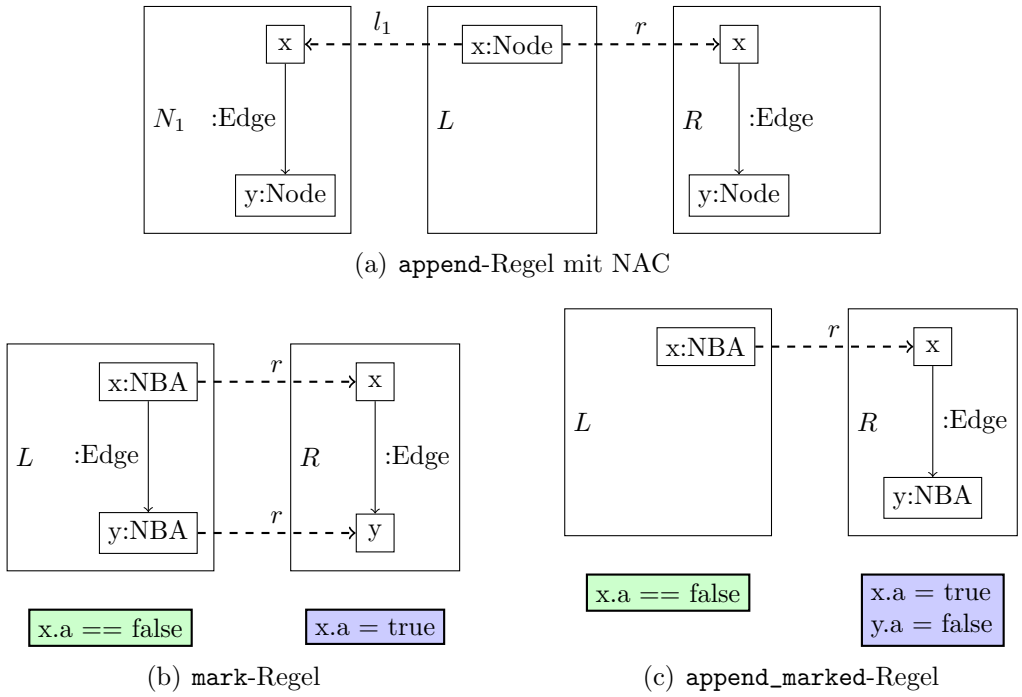
In folgendem Beispiel zeigen wir, wie die NAC die Ausdrucksstärke der Mustersuche erhöhen. Obwohl wir dabei einer Definition von Graphersetzungsregeln und ihrer Anwendung vorgreifen, sollte das Beispiel intuitiv verständlich sein. Wir wollen an das letzte Glied einer perlenschnurartigen Kette ein weiteres Glied (in Form einer neuen mit dem Ende verbundenen Ecke) anfügen. Aus

$$\mathtt{n}_1 \rightarrow \mathtt{n}_2 \rightarrow \mathtt{n}_3$$

soll also

$$\mathtt{n}_1 \rightarrow \mathtt{n}_2 \rightarrow \mathtt{n}_3 \rightarrow \mathtt{n}_4$$

werden. Mit NAC ist das einfach, die `append`-Regel in Abbildung 4.8(a) bewirkt, dass die letzte Ecke in der Kette, und nur diese, gefunden und entsprechend eine Ecke angefügt wird. Ohne NAC ist das nicht in einer einzigen Regelanwendung zu schaffen. Man benötigt dann beispielsweise ein Hilfsattribut, das für jede Ecke angibt, ob es nicht die Letzte ist. Dieses Attribut sei mit `false` initialisiert. Zunächst



(a) `append`-Regel mit NAC

(b) `mark`-Regel

(c) `append_marked`-Regel

Abbildung 4.8: Ausdrucksstärke von negativen Anwendungsbedingungen am Beispiel der Konstruktion einer Perlenschur

wird dieses Attribut mithilfe der Regel `mark`-Regel auf `true` gesetzt, es sei denn, es handelt sich um die letzte Ecke (siehe Abbildung 4.8(b)). Diese Regel muss so oft angewendet werden, bis sie nicht mehr anwendbar ist. Als Nächstes erweitert eine einmalige Anwendung der `append_marked`-Regel aus Abbildung 4.8(c) den Graphen wie gewünscht, denn die Attributbedingung `x.a == false` erlaubt nur am Ende der Kette eine Regelanwendung. Mit negativen Anwendungsbedingungen gewinnt die Spezifikationssprache für die Mustersuche also erheblich an Ausdrucksstärke.

4.2.7 Homomorphiebedingungen

Unser match-Funktor liefert zunächst totale Homomorphismen, also sind insbesondere auch *nicht injektive* Morphismen inbegriffen. Dies ist zwar manchmal gewünscht, aber führt doch zu schwerer verständlichen Regeln. Aus Sicht des Spezifikateurs ist dies fehlerträchtiger, da er ggf. nicht immer bedacht hat, dass irgendetwas nicht injektiv gepasst wird und somit eine Regel fälschlicherweise angewendet wird. Also kehren wir den Spieß um und verlangen normalerweise injektive Abbildungen mit der Möglichkeit, diese für einzelne Graphelemente wieder aufzuheben.

Auf diese Weise gewinnen wir noch an Ausdrucksstärke, da wir nun gezielt einzelne Graphelemente homomorph passen können, dies aber für andere Graphelemente ausschließen und eine monomorphe Passung verlangen. Diese *Homomorphiebedingungen* erlauben also für jeweils eine gewisse Menge von Graphelementen eine homomorphe Passung. Wir verlangen von der Relation die diese Mengen induziert, dass sie *reflexiv* und *symmetrisch* ist, nicht aber *transitiv* zu sein braucht. Damit werden die Homomorphiebedingungen offenbar von einer Verträglichkeitsrelation[10]

[10] engl.: *dependency relation*

und nicht von einer Äquivalenzrelation[11] induziert. Weil wir die „Klassenbildung“[12] durch eine Verträglichkeitsrelation allgemeiner gefasst haben, ist es möglich, dass ein Graphelement in zwei „Klassen“ vorkommt, ohne dass die „Klassen“ zusammenfallen.

Der praktische Vorteil gewisse Mustergraphelemente homomorph andere aber isomorph abbilden zu können, liegt in der Möglichkeit mehrere Situationen in einer Regel formalisieren zu können. So können mehrere fast identische Regeln zu einer Regel zusammengelegt werden.

Definition 4.10 (Homomorphiebedingung) Sei $\sim$ eine beliebige Verträglichkeitsrelation für den Graphen L dann ist

$$\begin{aligned}\hom(m) = \forall x, y \in E : \neg(x \sim y) \Rightarrow m(x) \neq m(y) \wedge \\ \forall a, b \in K : \neg(x \sim y) \Rightarrow m(x) \neq m(y)\end{aligned} \tag{4.13}$$

die *Homomorphiebedingung* des Graphen.

Kurz gesagt gibt die Verträglichkeitsrelation also an, welches Graphelement auf dasselbe Graphelement abgebildet werden darf. Natürlich kann die Verträglichkeitsrelation immer nur für Elemente ein und desselben Graphen definiert werden, somit sind übergreifende Konstruktionen zwischen NAC und positiven Mustergraphen nicht möglich. Allerdings können negative Graphen eine eigene Verträglichkeitsrelation besitzen und so ihre Homomorphiebedingungen bilden. Bei einer Vorbelegung ist die Homomorphiebedingung natürlich zu prüfen: Zwei vorbelegte Mustergraphelemente, die auf ein und dasselbe Element des Arbeitsgraphen abgebildet werden, aber die nicht in Verträglichkeitsrelation miteinander stehen, führen dazu, dass die Passung nie gültig erweitert werden kann, da die Homomorphiebedingung immer unerfüllt ist.

Definition 4.11 (match-Funktor mit Homomorphiebedingung) Seien $L, H \in \mathcal{I}_M$ Graphen, c eine Attributbedingung, $m' \in \mathbb{P}(L, H)$ ein Vorbelegungsmorphismus und $(N_i, l_i, c_i, \mathcal{N}_i)$ negative Anwendungsbedingungen. Seien weiterhin $\hom, \hom_i$ Homomorphiebedingungen, dann sind

$$\mathcal{N} = \{(N_1, l_1, c_1, \mathcal{N}_1, \hom_1), \ldots, (N_k, l_k, c_k, \mathcal{N}_k, \hom_k)\}$$

negative Anwendungsbedingungen mit Homomorphiebedingungen und

$$\text{match}(L, H, c, m', \mathcal{N}, \hom) \mapsto \{m \in \text{match}(L, H, c, m', \mathcal{N}) \mid \hom(m) = \top\}$$

ist der match-Funktor mit *Homomorphiebedingung*.

4.2.8 Dynamische Mustergraphen

Bisher haben wir nur *statische Muster* betrachtet. Insbesondere kennt auch die klassische Graphersetzung (siehe zum Beispiel Definition 2.34 oder Abschnitt 2.5) nur

[11] Eine Äquivalenzrelation ist reflexiv, transitiv und symmetrisch.

[12] Der Begriff Klasse ist hier eigentlich fehl am Platz da es eine Partitionierung, der Element in Klassen impliziert, was hier gerade nicht der Fall ist.

Programm 4.2: Beispiel eines dynamischen Musters

```
1  pattern Chain(from:Node, to:Node) {
2     alternative {
3        Base {
4           from --> to;
5        }
6        Recursive {
7           from --> intermediate:Node;
8           rest:Chain(intermediate, to);
9        }
10    }
11 }
12
13 rule DeleteLink {
14    x:Node --> x;
15    y:Node --> y;
16
17    ch:Chain(x, y);
18
19    replace {
20       x; y;
21    }
22 }
```

Muster fester Größe; es stehen also vor dem Finden der Passung die Graphelemente des Mustergraphen, die es zu finden gilt, fest. Nun erweitern wir die Möglichkeiten der Mustersuche hin zu *dynamischen Mustergraphen*. Bemerkenswert ist, dass wir den bisherigen Formalismus des Mustersuchens, nämlich den der Graphhomomorphismen, nicht verlassen: Einzig und allein wird der Mustergraph nicht mehr vor Beginn der Mustersuche feststehen, sondern vielmehr schrittweise nach gewissen Vorgaben konstruiert, bis, sofern dies möglich ist, ein Passungsmorphismus zwischen dem (konstruierten) Mustergraphen und dem Arbeitsgraphen existiert.

Die dynamischen Muster werden rekursiv deklariert und benötigen, zur Terminierung der Rekursion, Musteralternativen. Die eigentliche Passung versucht die Mustergraphen stückweise zu erweitern, bis ein terminierender Fall eintritt. In Programm 4.2 ist ein solches Muster zu sehen. Es besteht aus zwei *Alternativen* (Schlüsselwort `alternative`) namens `Base` und `Recursive`. Die erste von diesen ist ein terminierender Fall. Die zweite Alternative enthält die auf Rekursion basierende Mustererweiterung. In Zeile 1 wird das für dynamische Muster typische Schlüsselwort `pattern` verwendet, dass es ermöglicht Muster außerhalb einer Regel zu spezifizieren. Insgesamt spezifiziert das Muster `Chain` eine beliebig lange Kette aus elementaren Ecken- und Kantentypen zwischen zwei vorgegebenen Ecken. In Zeile 17 wird dieses rekursive Muster in einer Regel verwendet, die die gefundene Verbindung von `x` und `y`, sowie die Schleifen an ihnen, löscht. Die dynamischen Muster (also die Rekursion) werden nicht in den normalen Musterbegriff integriert, sondern vorab ausgefaltet; hernach ist die normale Mustersuche möglich.

Diese Erweiterungen wurden im Rahmen der Studienarbeit von Jakumeit erarbeitet [Jak07] und in seiner Diplomarbeit implementiert [Jak08].

4.2.9 GR-Muster

Die von uns im Verlauf dieses Kapitels angereicherten neuartigen Mustergraphen nennen wir nun GR-Muster, um zu zeigen, dass sie erstens vom GRGEN-Werkzeug verarbeitet werden können und um zweitens ihre Abkehr vom reinen Graphmuster deutlich zu machen. Die Frage, wann die linke Seite einer Graphersetzungsregeln – also ein GR-Muster – auf einen Arbeitsgraphen passt, wird durch die Definition 4.13 geklärt.

Normalerweise wollen wir bei GR-Mustern ohne Kenntnis der inneren Struktur Vorbelegungen durchführen können. Ähnlich wie eine Schnittstellenbeschreibung in höheren Programmiersprachen, wollen wir nicht den gesamten Mustergraphen L, also die Implementierung, kennen müssen, um den Vorbelegungsmorphismus m' bestimmen zu können. Es soll uns analog zur *Methodensignatur* in höheren Programmiersprachen reichen, zu wissen, welche Elemente vorbelegt werden können. Die Information, welche dies sind, soll dann ausschließlich in der Reihenfolge der Argumente enthalten sein.

Darum definieren wir eine Argumentfunktion, die unabhängig von aktuellen Arbeitsgraphen den Zusammenhang zwischen der Position des Argumentes in der Folge der Argumente und einem Element des Mustergraphen L herstellt:

$$\arg : \mathbb{N} \to L$$

Die Argumente selbst müssen natürlich dem jeweiligen Arbeitsgraphen H entstammen, also ist eine Argumentfolge als

$$(h_1, \ldots, h_k) \qquad \text{mit } h_i \in H$$

definiert. Der Zusammenhang zwischen einer Argumentfunktion, einer Argumentfolge und einem Vorbelegungsmorphismus wird durch folgende Definition gegeben:

Definition 4.12 (Argumente und Vorbelegungsmorphismus) Sei $L, H \in \mathcal{I}_M$ Graphen, arg eine Argumentfunktion für L, $(h_1, \ldots, h_k)$ eine Argumentfolge aus H. Ferner sei $\{1, \ldots, k\} \subseteq \text{def}(\arg)$, dann ist der *zugehörige Vorbelegungsmorphismus* $m' \in \mathbb{P}(L, H)$ gegeben durch

$$m' : \arg(i) \mapsto h_i \qquad \text{mit } i \in \{1, \ldots, k\}$$

Mit diesen Vorbereitungen können wir nun sagen, was es bedeutet ein GR-Muster zu sein und wann dieses passt.

Definition 4.13 (GR-Muster und Passungen) Seien $L, H \in \mathcal{I}_M$ Graphen, c eine Attributbedingung, $\mathcal{N}$ negative Anwendungsbedingungen, hom eine Homomorphiebedingung, arg ein Argumentfunktion für L und $(h_0, \ldots, h_k)$ eine Folge von Argumenten aus H, dann hat das GR-Muster $G = (L, c, \arg, \mathcal{N}, \text{hom})$ mit den Argumenten $G(h_0, \ldots, h_k)$ genau dann auf dem Arbeitsgraphen H *Passungen*, wenn

$$\text{match}(L, H, c, m', \mathcal{N}, \text{hom}) \neq \emptyset$$

mit dem nach Definition 4.12 zu den Argumenten gehörigem Vorbelegungsmorphismus m'.

Die Menge $\text{match}(L, H, c, m', \mathcal{N}, \text{hom})$ ist die *Menge der Passungen*. Für diese Menge der Passungen gilt, dass sie eine Teilmenge der elementaren (sprich nicht angereicherten) Passungen des Graphmusters L sind, also gilt:

$$\text{match}(L, H, c, m', \mathcal{N}, \text{hom}) \subseteq \mathbb{T}(L, H)$$

Wie auch bei elementaren Passungen kann es keine, eine oder mehrere Passungen geben. Falls es keine oder eine Passung gibt, ist keine Auswahlmöglichkeit gegeben. Im Falle mehrerer Passungen hingegen ist aus Sicht des Graphmusters keine der Passungen zu bevorzugen; es muss also von außen eine Entscheidung getroffen werden. Man kann *keine* wählen, man kann eine zufällig oder implementierungsabhängig wählen, oder gar alle Passungen weiterverwenden. Diese Entscheidung obliegt allerdings nicht dem Graphersetzungsverfahren, sondern muss offenbar vom Benutzer (sei das ein Programm oder ein Mensch) getroffen werden. Denn könnte man diese Entscheidung mit Graphersetzung elegant ausdrücken, hätte der Benutzer die Regeln entsprechend eindeutig formulieren können. Mehr zu praktischen Aspekten der *Passungsauswahl* in Anhang A.5. Eine damit eng verwandte Fragestellung ist die *Regelauswahl*. Diese wird in Abschnitt 4.3.7 von ihren theoretischen und philosophischen Seiten beleuchtet, wohingegen Abschnitt 4.4 eine praktische Antwort gibt, nämlich die *erweiterten Graphersetzungssequenzen*.

```
rule append {
  pattern {
    x:Node;
    negative {
      x --> y:Node;
    }
  }
  replace {
    x --> y:Node;
  }
}
```

(a) `append`-Regel mit NAC

```
rule mark {
  pattern {
    x:NBA --> y:NBA;
    if {
      x.a == false;
    }
  }
  replace {
    x --> y;
  }
  eval {
    x.a = true;
  }
}
```

(b) `mark`-Regel

```
rule append_marked {
  pattern {
    x:NBA;
    if {
      x.a == false;
    }
  }
  replace {
    x --> y:NBA;
  }
  eval {
    x.a = true;
    y.a = false;
  }
}
```

(c) `append_marked`-Regel

Abbildung 4.9: Regel zur Erweiterung einer Perlenschur mit und ohne NAC, analog zur grafischen Darstellung in Abbildung 4.8

4.2.10 Notation

In diesem Abschnitt geben wir Beispiele für die textuelle Notation der Graphmuster; die gesamten Graphersetzungsregeln betrachten wir erst am Ende des nächsten Hauptabschnitts. Normalerweise stehen die Muster[13] nicht allein, sondern sind in

[13] engl.: *pattern*

Graphersetzungsregeln eingebettet, darum sind hier auch ganze Regeln angegeben. Abbildung 4.9 zeigt die Regeln in textueller Form des Beispieles (vgl. Abbildung 4.8) aus Abschnitt 4.2.6.

Mit dem Schlüsselwort `pattern` wird ein Mustergraph eingeleitet, negative Mustergraphen, also NACs, werden mit dem Schlüsselwort `negative` angegeben. Die Notation der Graphen erfolgt analog zu den *benannten Graphen*, die in Abschnitt 4.1.2 diskutiert wurden. Der Erhaltungsmorphismus wird implizit durch Namensgleichheit von Graphelementen in Mustergraph und Ersetzungsgraph definiert. Der gleiche implizite Mechanismus wird bei den Einbettungen der negativen Mustergraphen benutzt. So werden zum Beispiel bei der `append`-Regel in Abbildung 4.9 die Ecke des Mustergraphen, des Ersetzungsgraphen und des negativen Mustergraphen, die jeweils den gleichen Namen x haben, mit r und l_1 entsprechend aufeinander abgebildet. Der Name x wurde im Kontext des Mustergraphen definiert und festgelegt, dass er an Ecken vom Typ `Node` $= \bot$ binden darf. Die Benutzungen im Ersetzungsgraphen und NAC tragen diese Typisierung ebenfalls mit sich. Die Gültigkeitsregel von Definitionen folgt der Blockschachtelung, mit der Ausnahme, dass alle Definitionen des Mustergraphen auch im Ersetzungsgraphen gültig sind.

In Erweiterung der Schreibweise für benannte Graphelemente `x : T` können wir auch Typeinschränkungen (sogar dynamische) direkt angeben. Also sind z. B. folgende Definitionen von benannten Graphelementen erlaubt: `x : T \ T'`, `y : T \ (T' + T'')` und `z : typeof(x)`. Im Einzelnen können diese drei Definitionen von Graphelementen wie folgt auf bereits Bekanntes zurückgeführt werden:

$$
\begin{aligned}
\texttt{x : T} \setminus \texttt{T}' &\implies \texttt{x : T; if\{!(T}' \texttt{<= typeof(x)); \}} \\
\texttt{y : T} \setminus (\texttt{T}' + \texttt{T}'') &\implies \texttt{x : T; if\{!(T}' \texttt{<= typeof(y)) \& !(T}'' \texttt{<= typeof(y)); \}} \\
\texttt{z : typeof(x)} &\implies \texttt{x : } \bot\texttt{; if\{typeof(x) <= typeof(z); \}}
\end{aligned}
$$

Wenn wir in einem NAC auf ein in der Umgebung definiertes Graphelement zugreifen wollen, dann muss dieses Element auch in der lokalen Umgebung definiert sein. Dies ist jedoch nicht immer der Fall, wie das Programm 4.3 zeigt. Allerdings kann durch eine Vorverarbeitung eine Situation herbeigeführt werden, sodass wir unsere NAC-Theorie darauf wieder anwenden können: Alle Graphelemente die in einem tieferen Gültigkeitsbereich benötigt werden, aber eigentlich nicht lokal vorkommen dürfen (da sonst die Semantik eine andere wäre) werden dennoch auf jede Ebene bis zur Verwendung eingefügt. Schließlich muss noch die Wirkung davon wieder „neutralisiert“ werden, was durch Konstruktionen von Homomorphiebedingung mit allen Ecken bzw. Kanten eines Gültigkeitsbereich erreicht werden kann. Durch letztere Maßnahme ist es möglich die Elemente des NAC wieder frei abzubilden, so als wären die störenden Graphelemente tatsächlich nicht eingefügt worden. Führen wir diese Maßnahmen für Programm 4.3 durch, führt dies zu der vorverarbeiteten Programm 4.4.

Des Weiteren sei hier wieder auf die Kurzreferenz im Anhang A.3 und eine detaillierte Beschreibung im Benutzerhandbuch von GrGen.NET verwiesen [BG07]. Die Randfälle von Definition 4.13 und einige ausgesuchte Beispiele betrachten wir im Anhang B.1.

Programm 4.3: Verschachteltes NAC

```
...
pattern {
  x:T; y:A --> x;
  z:B -e:Edge-> z;
  negative {
    z --> :C;



    if{ e.v == 42; }
    negative {
      x -:F-> x;

      if{ y.a != 23; }
    }
  }
}
...
```

Programm 4.4: Vorverarbeitetes Muster

```
...
pattern {
  x:T; y:A -$1:Edge-> x;
  z:B -e:Edge-> z;
  negative {
    z -$2:Edge-> $3:C;
    -e:Edge->; hom(e, $2);
    x; hom(x, z); hom(x, $3); hom(x, y);
    y; hom(y, z); hom(y, $3);
    if{ e.v == 42; }
    negative {
      x -$4:F-> x;
      y; hom(x, y);
      if{ y.a != 23; }
    }
  }
}
...
```

4.3 Erweiterter Single-Pushout Ansatz (XSPO)

In diesem Kapitel haben wir bisher intuitiv von Graphersetzung und Graphersetzungsregeln gesprochen. In diesem Abschnitt wollen wir diese Begriffe konkretisieren und auf die formalen Fundamente aus Abschnitt 2.5 stellen. Graphersetzungsformalismen gibt es, wie wir in Abschnitt 2.5 gesehen haben, einige, jedoch sind nur wenige für unsere Belange brauchbar. Aus diesen Varianten wählen wir den SPO-Ansatz als Grundlage.

Wir erweitern den SPO-basierten Ersetzungsformalismus (vgl. Definition 2.34) um *Reattributierungsanweisungen* (Abschnitt 4.3.1), *Retypisierung* (Abschnitt 4.3.2), *dynamische Typisierung* (Abschnitt 4.3.3), *Kopieranweisungen* (Abschnitt 4.3.4) sowie die *Rückgabe von Arbeitsgraphelementen* (Abschnitt 4.3.5). Der Graphersetzungsschritt läuft bei *XSPO* zweigeteilt ab, zunächst geschieht die ganz normale SPO-Ersetzung, dann wird dieser Ergebnisgraph durch eine spezielle denotationelle *Ausführungsfunktion* weiter verändert. In Reminiszenz an den zugrunde liegenden Formalismus nennen wir diesen von uns erheblich erweiterten Ersetzungsformalismus XSPO[14]. Die Formalisierung basiert, wie auch bei den Graphmustern, auf denotationellen Auswertungsfunktionen und Graphhomomorphismen; für den Ersetzungsformalismus kommt noch die Kategorientheorie hinzu.

4.3.1 Reattributierungsanweisungen

Genau, wie wir die Mustersuche, via Attributbedingungen, von Attributen abhängig gemacht haben, so wollen wir auch beim Ersetzungsschritt die Attribute miteinbeziehen. Wir schaffen durch *Reattributierungsanweisungen* die Möglichkeit, die Attribute des gerade ersetzten Teiles des Graphen in Abhängigkeit von den Attributen, die den

[14] extended single pushout

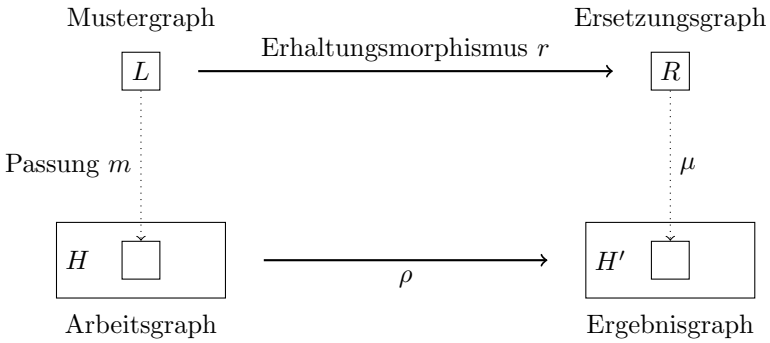


Abbildung 4.10: Darstellung einer SPO-Graphersetzungsregel und deren Anwendung

Graphelementen im Muster entsprechen, zu aktualisieren. Ein Beispiel hierfür haben wir schon in der Regel `mark` in Abbildung 4.8 und 4.9 vorweggenommen.

Die Attribute, die von den Reattributierungsanweisungen einer Regel erfasst werden können, sind auf das Bild des Ersetzungsgraphen im Arbeitsgraphen $\mu(R)$ beschränkt (siehe Abbildung 4.10). Dies geschieht aus zweierlei Gründen: Übersichtlichkeit und Geschwindigkeit. Für viele Anwendungen ist eine automatische Fortschaltung der Attributwerte über den gesamten Arbeitsgraphen hinweg nicht nötig, sondern dies macht sogar die Spezifikationen schwerer verständlich und führt – aufgrund der nichtlokalen Wirkung – zu Fehlern[15]. Es ist allerdings möglich durch weitere Regeln, die nur der Attributfortschaltung dienen, die Änderung der Attribute beliebig im Graphen zu transportieren. Auf diese Weise ist auch Fixpunktiteration o. ä. realisierbar. Jedoch sind hierzu eben mehrere Regeln und wiederholte Regelanwendung nötig.

Es sind nur Reattributierungsanweisungen der Form $x.a = e$ erlaubt, wobei, wie bei Attributbedingungen, $x \in X$ mit $X = E$ oder $X = K$ und $a \in \mathrm{attr}_X(\mathrm{typ}_X(x))$. Ferner sei e ein Java-ähnlicher Ausdruck, der, anders als bei Attributbedingungen, über die booleschen Werte hinaus auch zu ganzen Zahlen, Gleitkommazahlen oder Texten ausgewertet werden darf. Natürlich müssen der Typ des Ausdrucks e und der Typ des Attributs $x.a$, also $\mathrm{attrtyp}_X(a)$, kompatibel sein. Die in Abschnitt 4.2.2 definierte Auswertungsfunktion $\mathrm{eval}[\![\cdot]\!]$ wird nun für die neuen Anforderungen erweitert, mithin gilt nun

$$\mathrm{eval}[\![\cdot]\!] : \mathbb{T}(\mathcal{I}_M, \mathcal{I}_M) \times \mathbb{T}(\mathcal{I}_M, \mathcal{I}_M) \to \bigcup_{\tau \in S_E \cup S_K} V_\tau$$

wobei S_E bzw. S_K die Menge aller Ecken- und Kanten-Attributtypen von M ist. Bei der Bestimmung von Attributwerten werden die Attribute des Ergebnisgraphs bevorzugt:

$$\mathrm{eval}[\![\texttt{x.a}]\!]_{m,\mu} = \begin{cases} \mathrm{eval}[\![\texttt{x.a}]\!]_\mu & \text{wenn } x \in \mathrm{def}(\mu) \\ \mathrm{eval}[\![\texttt{x.a}]\!]_m & \text{sonst} \end{cases},$$

Die Semantik einer Anweisungen `x.a = e` definieren wir mithilfe der denotationellen *Ausführungsfunktion*

$$\mathrm{exec}[\![\cdot]\!] : \mathcal{I}_M \times \mathbb{T}(\mathcal{I}_M, \mathcal{I}_M) \times \mathbb{T}(\mathcal{I}_M, \mathcal{I}_M) \to \mathcal{I}_M$$

[15] Man denke insbesondere an die Erfahrungen, die man bei attributierten Grammatiken [WG84] im Übersetzerbau macht, wenn man dieses Konzept Anfängern nahe bringen will. Die dort inhärenten nichtlokalen Wirkungen benötigen sehr viel Erfahrung des Spezifikateurs, um fehlerarm beherrscht zu werden.

die wiederum auf der Auswertungsfunktion für Ausdrücke basiert. Die Auswertungsfunktionen benötigen nun auch zwei Morphismen, nämlich m und μ, für den Zugriff auf „alte“ sowie neue Graphelemente.

Definition 4.14 Wir setzen die korrekte Typisierung der Anweisung $x.a = e$ voraus, dann ist die *Ausführungsfunktion* für *Reattributierungsanweisungen* bestimmt durch

$$\text{exec}[\![\texttt{x.a = e}]\!]_{m,\mu} H = \tilde{H}, \text{ wobei } \tilde{H} := H \text{ bis auf:}$$

$$\left\{ \begin{array}{lcll} \text{val}_E(m(x), a) & := & \text{eval}[\![\texttt{e}]\!]_{m,\mu} & \text{wenn } x \in E \wedge x \notin \text{def}(\mu) \\ \text{val}_E(\mu(x), a) & := & \text{eval}[\![\texttt{e}]\!]_{m,\mu} & \text{wenn } x \in E \wedge x \in \text{def}(\mu) \\ \text{val}_K(m(x), a) & := & \text{eval}[\![\texttt{e}]\!]_{m,\mu} & \text{wenn } x \in K \wedge x \notin \text{def}(\mu) \\ \text{val}_K(\mu(x), a) & := & \text{eval}[\![\texttt{e}]\!]_{m,\mu} & \text{wenn } x \in K \wedge x \in \text{def}(\mu) \end{array} \right\}$$

4.3.2 Retypisierung

Die Retypisierung erlaubt, den Typ von beliebigen Graphelementen zu wechseln – und zwar ohne Einschränkungen. Es ist also erlaubt ein Graphelement des Typs s zu einem Graphelement des Typs t zu retypisieren, obwohl die Typen in keinerlei Untertypbeziehung stehen[16]. Der Sinn von Retypisierung ist, dass man, ohne den Kontext eines Graphelements zu verlieren (bei Ecken sind dies insbesondere die i. A. unbeschränkt vielen inzidenten Kanten) es quasi durch ein neues (also eines mit einem neuen Typ) ersetzen kann.

Damit eine Retypisierung `x : t<y>` des Graphelements y zu einem Graphelement x mit dem Typen t wohlgeformt ist, genügt zweierlei: Erstens y muss ein Graphelement des Ersetzungsgraphen R sein und x darf nicht anderweitig definiert sein. Zweitens, wenn y eine Kante ist, dann ist auch t ein Kantentyp, wenn hingegen y eine Ecke ist, dann ist auch t ein Eckentyp.

Wenn eine Retypisierung `x : t<y>` in einer rechten Seite einer Regel verlangt wird, dann wird der Ersetzungsgraph wie folgt abgeändert: Falls x und also auch y eine Kante ist, wird eine neue Kante x in R eingefügt, die zu y parallel ist und den Typ t hat. Falls x eine Ecke ist, verhält sich die Retypisierung zunächst wie eine normale Eckendefinition in R. Es wird also eine neue Ecke x in R eingefügt, die den Typ t hat und vorerst keine inzidenten Kanten besitzt. Allerdings können in R auch Verbindungen zu x spezifiziert sein. Auf jeden Fall ist der Erhaltungsmorphismus so zu wählen, dass er y überstreicht, also y im Bild des Erhaltungsmorphismus r enthalten ist (vgl. Abbildung 4.10).

Definition 4.15 Wir setzen die Wohlgeformtheit von `x : t<y>` voraus, dann ist

[16] Die Retypisierung ist gewisserweise eine Verallgemeinerung des *cast*-Operators in herkömmlichen Programmiersprachen, die sich über Typschranken hinwegsetzt.

die *Ausführungsfunktion* für die *Retypisierung* bestimmt durch

$\text{exec}[\![\texttt{x : t<y>}]\!]_{m,\mu} H = \tilde{H}$, wobei $\tilde{H} := H$ bis auf:

$$\left\{\begin{array}{lcll} \text{val}_E(\mu(x), a) & := & \text{val}_E(\mu(y), a) & \text{wenn } x \in E \wedge a \in A_E \\ \text{val}_K(\mu(x), a) & := & \text{val}_K(\mu(y), a) & \text{wenn } x \in K \wedge a \in A_K \\ \text{src}(k) & := & x & \text{wenn } x \in E \wedge k \in K_H : \text{src}(k) = y \\ \text{tgt}(k) & := & x & \text{wenn } x \in E \wedge k \in K_H : \text{tgt}(k) = y \\ E_{\tilde{H}} & := & E_{\tilde{H}} \setminus \{y\} & \text{wenn } x \in E \\ K_{\tilde{H}} & := & K_{\tilde{H}} \setminus \{y\} & \text{wenn } x \in K \end{array}\right\}$$

mit $A_E := \text{attr}_E(\text{typ}_E(x)) \cap \text{attr}_E(\text{typ}_E(y))$,
$A_K := \text{attr}_K(\text{typ}_K(x)) \cap \text{attr}_K(\text{typ}_K(y))$

4.3.3 Dynamische Typisierung

So wie die dynamischen Typbedingungen aus Abschnitt 4.2.3, die Typen der einen Mustergraphelemente von den Typen der gepassten Arbeitsgraphelemente anderer Mustergraphelemente abhängig machen, so erlaubt die dynamische Typisierung das Erzeugen von Graphelementen im Ersetzungsgraphen mit Typen, die durch Mustergraph- und Arbeitsgraphelemente vorgegeben werden. Diese Eigenschaft ist immer dann nötig, wenn man ein neues Graphelement erzeugen will, aber dessen Typ vom tatsächlichen, nicht dem spezifizierten Typ eines Musterelements abhängt. Durch die Verwendung dynamischer Typisierung können gegebenenfalls mehrere Regeln zu einer zusammenfasst werden.

Falls ein Element x des Ersetzungsgraphen dynamisch typisiert werden soll, also `x:typeof(y)`, dann ist es zunächst normal als Element mit minimalem Typ in R aufzunehmen, d. h. $x \in X$ mit $\text{typ}_X(x) = \bot$ sowie $X = E$ oder $X = K$, je nachdem ob x eine Ecke oder Kante ist. Also wird beim SPO-Ersetzungsschritt mit $\mu(x)$ ein korrespondierendes Element im Ergebnisgraphen erzeugt. Damit die dynamische Typisierung wohlgeformt ist, muss nur y existieren und x natürlich nicht anderweitig in R definiert sein. Nun muss noch der Typ des Graphelements aus dem Ersetzungsgraphen angepasst werden, dies geschieht wiederum durch das Erweitern der Ausführungsfunktion exec:

Definition 4.16 Wir setzen die Wohlgeformtheit von `x : typeof(y)` voraus, dann ist die *Ausführungsfunktion* für die *dynamischen Retypisierung* bestimmt durch

$\text{exec}[\![\texttt{x : typeof(y)}]\!]_{m,\mu} H = \tilde{H}$, wobei $\tilde{H} := H$ bis auf:

$$\left\{\begin{array}{lcll} \text{typ}_E(\mu(x)) & := & \text{typ}_E(\mu(y)) & \text{wenn } y \in E \wedge y \in \text{def}(\mu) \\ \text{typ}_E(\mu(x)) & := & \text{typ}_E(m(y)) & \text{wenn } y \in E \wedge y \notin \text{def}(\mu) \\ \text{typ}_K(\mu(x)) & := & \text{typ}_K(\mu(y)) & \text{wenn } y \in K \wedge y \in \text{def}(\mu) \\ \text{typ}_K(\mu(x)) & := & \text{typ}_K(m(y)) & \text{wenn } y \in K \wedge y \notin \text{def}(\mu) \end{array}\right\}$$

4.3.4 Kopieranweisungen

Normalerweise erfasst eine Graphersetzungsregel mit der linken oder rechten Seite immer eine vorher definierte Menge an Elementen im Arbeitsgraphen. Es ist nicht möglich, gewisse, erfasste Teilstrukturen zu verdoppeln; ein mehrfaches Benutzen eines Graphelements in der textuellen Spezifikation des Ersetzungsgraphen referenziert immer wieder dasselbe Element des Ersetzungsgraphen und erzeugt keine Kopien (siehe auch Abschnitt 4.3.6).

Jedoch ist es wünschenswert bei Bedarf gewisse Graphelemente kopieren zu können (genauer gesagt zu duplizieren), da so das umständliche und langsame manuelle Kopieren mit eigens dafür spezifizierten *Schiffchen* und Graphersetzungsregeln entfallen kann. Genau dies leistet eine Kopieranweisung `x : copy(y)`, die in Ersetzungsgraphen auftreten darf. Diese legt von der gepassten Entsprechung von Graphelement y eine Kopie mit Namen x an, wobei der gesamte Kontext – dazu zählen auch die Attribute – erhalten bleibt.

Der Ersetzungsgraphen wird anlog zur Retypisierung (vgl. Abschnitt 4.3.2) vorbereitet: Wenn eine Kopieranweisungen `x : copy(y)` in einer rechten Seite einer Regel verlangt wird, dann wird der Ersetzungsgraph wie folgt abgeändert: Falls x und also auch y eine Kante ist, wird eine neue Kante x in R eingefügt, die zu y parallel ist und den Typ $\bot$ hat. Falls x eine Ecke ist, verhält sich die Retypisierung zunächst wie eine normale Eckendefinition in R. Es wird also eine neue Ecke x in R eingefügt, die den Typ $\bot$ hat und vorerst keine inzidenten Kanten besitzt. Allerdings können in R auch Verbindungen zu x spezifiziert sein. Auf jeden Fall ist der Erhaltungsmorphismus so zu wählen, dass er y überstreicht, also y im Bild des Erhaltungsmorphismus r enthalten ist (vgl. Abbildung 4.10).

Definition 4.17 Wir setzen die Wohlgeformtheit von `x : copy(y)` voraus, dann ist die *Ausführungsfunktion* für die *Kopieranweisungen* bestimmt durch

$$\text{exec}[\![\texttt{x : copy(y)}]\!]_{m,\mu} H = \tilde{H}, \text{ wobei } \tilde{H} := H \text{ bis auf:}$$

- $\text{typ}_X(\mu(x)) := \text{typ}_X(\mu(y))$
- $\text{val}_X(\mu(x), a) := \text{val}_X(\mu(y), a) \qquad \forall a \in \text{attr}_X(\text{typ}_X(\mu(y)))$
- Falls $y \in E$ gilt, dann sei $\forall k \in K_H$:
  - $\text{src}(k') := \mu(x), \text{tgt}(k') := \mu(x) \qquad$ falls $\mu(y) = \text{src}(k) = \text{tgt}(k)$
  - $\text{src}(k') := \mu(x), \text{tgt}(k') := \text{tgt}(k) \qquad$ falls $\mu(y) = \text{src}(k) \neq \text{tgt}(k)$
  - $\text{src}(k') := \text{src}(k), \text{tgt}(k') := \mu(x) \qquad$ falls $\mu(y) = \text{tgt}(k) \neq \text{src}(k)$

  wobei k' jeweils eine neue Kante $K_{\tilde{H}} := K_{\tilde{H}} \cup \{k'\}$ ist

mit $X = E$ falls $x \in E$ und $X = K$ falls $x \in K$.

4.3.5 Anwendung und XSPO-Ersetzungsschritt

Wir erweitern die SPO-Graphersetzung aus Abschnitt 2.5, bei der eine Graphersetzungsregel die einfache Form

$$p : L \xrightarrow{r} R$$

hat, um *typisierte Graphen* (inklusive *dynamischer Typbedingungen*), *Attributbedingungen*, *Vorbelegungen*, *negative Anwendungsbedingungen*, *Homomorphiebedingungen*, *dynamische Muster*, *Reattributierungsanweisungen*, *Retypisierung*, *dynamische Typisierung*, *Kopieranweisungen* und die *Rückgabe von Arbeitsgraphelementen*. Die Definitionen dieses Abschnitts spiegeln jene Erweiterung wider. Es sei hier zusätzlich angemerkt, dass keiner der anderen bekannten Graphersetzungsformalismen derart reichhaltig ausgestaltete, aber dennoch auf formalen Grundlagen stehende Graphersetzungsregeln aufweist.

Wir wollen zur Vereinfachung der Integration der Regelanwendung in imperative Umgebungen analog zu den Argumenten einer Regel (vgl. Abschnitt 4.2.9) auch die Rückgabe von Arbeitsgraphelementen ermöglichen, die per Passung im Ergebnisgraphen identifizierbar sind. Dazu definieren wir eine *Rückgabefunktion* für jede Regel, die einer natürlichen Zahl ein Element des Ersetzungsgraphen zuordnet.

$$\text{ret} : \mathbb{N} \to R$$

Definition 4.18 (Graphersetzungsregel) Seien $L, R, H \in \mathcal{I}_M$ Graphen, ein GR-Muster sei gegeben als $G = (L, c, \text{arg}, \mathcal{N}, \text{hom})$, e eine Folge von Anweisung für die Ausführungsfunktion exec und ret eine Rückgabefunktion, dann heißt

$$p : (L, c, \text{arg}, \mathcal{N}, \text{hom}) \xrightarrow{r} (R, e, \text{ret})$$

eine *XSPO-Graphersetzungsregel*. Die Regel p ist mit den Argumenten $(h_0, \ldots, h_k)$ genau dann auf den Arbeitsgraphen H *anwendbar*, wenn das zugehörige GR-Muster $G(h_0, \ldots, h_k)$ Passungen hat.

Eine Graphersetzungsregel spezifiziert durch die linke Seite $(L, c, \text{arg}, \mathcal{N}, \text{hom})$ was gefunden werden soll und mit der rechten Seite (R, e, ret) wird angegeben, wie die Fundstelle zu transformieren ist und was zurückgegeben wird. Die Frage, wie diese Transformation geschrieben wird, und wie ihre Semantik ist, klären die beiden folgenden Definitionen.

Definition 4.19 (XSPO-Ersetzungsschritt) Seien eine XSPO-Graphersetzungsregel

$$p : (L, c, \text{arg}, \mathcal{N}, \text{hom}) \xrightarrow{r} (R, e, \text{ret})$$

mit den Argumenten $(h_0, \ldots, h_k)$ gegeben. Sei außerdem m eine Passung des GR-Musters $(L, c, \text{arg}, \mathcal{N}, \text{hom})$ im Arbeitsgraphen H. Dann ist

$$H' = [\![e]\!]_{m,\mu}\tilde{H}$$

der Ergebnisgraph des *XSPO-Ersetzungsschrittes*, wobei der Morphismus μ und der Graph $\tilde{H}$ (als Pushout) durch folgende Anwendung einer SPO-Graphersetzungsregel $p' : L \xrightarrow{r} R$ gegeben ist:

$$H \overset{p',m}{\Longrightarrow} \tilde{H}$$

Da der Passungsmorphismus nicht notwendigerweise eindeutig ist, muss er – gleichwie die Argumente – immer mit angegeben werden, um die Anwendung einer Graphersetzungsregel vollständig zu spezifizieren.

Definition 4.20 (Anwendung) Falls die Graphersetzungsregel p mit den Argumenten $(h_0, \ldots, h_k)$ auf dem Graphen H eine Passung m besitzt, dann wird die *Anwendung* auf ebendiesen Graphen H mit dem Ergebnis des zugehörigen *XSPO-Ersetzungsschrittes* H' als

$$H \overset{p(h_0,\ldots,h_k),m}{\Longrightarrow} \big(H', (\mu(\mathrm{ret}(1)), \ldots, \mu(\mathrm{ret}(k))\big)$$

geschrieben.

Man beachte, dass es möglich ist, dass eine Regel bei gegebenem Graphen und Argumenten nicht passen kann, also kein geeignetes m existiert. In diesem Fall kann man gar keine Anwendung von p durchführen, da ohne m keine Ersetzung stattfinden kann. Wegen des Rückgriffs auf die grundlegenden SPO-Definitionen ist die Anwendung immer wohldefiniert, d. h. es können nie Ergebnisse auftreten, die nicht eindeutig charakterisiert oder keine Graphen sind; mehr noch, der XSPO-Ersetzungsschritt ist immer algorithmisch konstruierbar.

4.3.6 Abgrenzung

Es gibt in der Fachliteratur zahlreiche Ersetzungs-Verfahren, die, obgleich sie nicht unbedingt Graphersetzung betreiben, das Bild des Ersetzungsbegriffs geprägt haben. Es folgen hier einige Beispiele, die zeigen sollen, wie wir uns mit der XSPO-Graphersetzung von diesen Verfahren abheben.

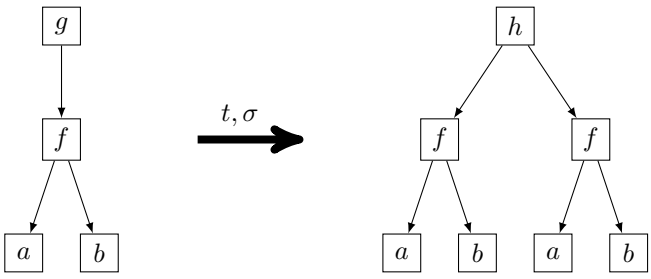


Abbildung 4.11: Anwendung einer Termersetzungsregel

4.3.6.1 Termersetzung mit Variablen

Wir betrachten zunächst eine Termersetzung t

$$t : g(x) \to h(x, x),$$

wobei die Substitution der Variable x durch beliebige Teilterme erlaubt sein soll. Wir wenden nun die obige Termersetzung t auf den Term $g(f(a, b))$ an:

$$g(f(a, b)) \overset{t}{\Longrightarrow} h(f(a, b), f(a, b))$$

Offenbar dupliziert hierbei die Termersetzungssubstitution $\sigma = [(f(a,b)/x]$ den Teilterm $f(a,b)$. Wenn wir das Ganze als Baumstruktur visualisieren, ergibt sich eine Struktur wie in Abbildung 4.11 zu sehen.

Dieses Verhalten, nämlich das Kopieren von Teiltermen, ist aus vielerlei Hinsicht ungünstig. Denn es vergrößert die Darstellung, erschwert durch viele ähnliche Teile die Mustersuche, benötigt mehr Speicher um die gleichen Informationen darzustellen und macht den Ersetzungsschritt komplexer.

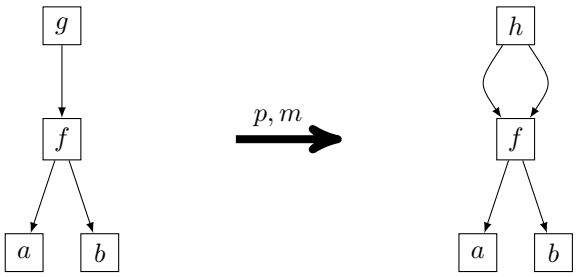


Abbildung 4.12: Anwendung einer Graphersetzungsregel

Die Termersetzungsregel $t : g(x) \to h(x,x)$ ist rein syntaktisch sehr ähnlich der XSPO-Graphersetzungsregel in Programm 4.5. Genau dann, wenn die Termersetzungsregel t auf einem Term anwendbar ist, dann ist auch die Graphersetzungsregel p auf dem zu diesem Term äquivalenten Graphen anwendbar. Wir verwenden hier die GrGen-Syntax, die kompakter und lesbarer ist als die entspreche formale Notation. Den Erhaltungsmorphismus erhalten wir durch Namensgleichheit, also der gleiche Name x in L und R signalisieren Erhaltung der entsprechenden Ecke. Die Typen der Ecken g und h entsprechen den jeweiligen Funktionstypen und x hat den allgemeinsten Eckentyp, damit x mit jeder Ecke zur Passung gebracht werden kann.

Programm 4.5: Graphersetzungsregel p in GrGen-Syntax

```
1 rule p {
2   pattern { g:G --> x:Node; }
3   replace { x <-- h:H --> x; }
4 }
```

Allerdings hat die Graphersetzung ein anderes Ergebnis, als die entsprechende Termersetzung, obwohl R analog zur rechten Seite der Termersetzungsregel konstruiert wurde. Die beiden Ecken x werden nämlich nicht vermittels der Substitution σ durch einen ganzen Subgraphen ersetzt, sondern sie werden nur durch eine Ecke $\mu(x)$ repräsentiert, wobei μ durch das kommutierende Diagramm (siehe Abbildung 4.10) und speziell den Passungsmorphismus m, den Erhaltungsmorphismus r und die Graphen L, R bestimmt ist. Darum werden, wie in Abbildung 4.12 zu sehen ist, die beiden Argumente von h, also $x \leftarrow h \rightarrow x$, nur durch zwei Kanten von der Ecke $\mu(x)$ zur Ecke $\mu(h)$ repräsentiert.

4.3.6.2 Termgraphen

Auch bei Termgraphen herrscht die Sichtweise vor, dass gleiche Teilterme bei der Transformation nicht kopiert, sondern durch einen Verweis, sprich eine Kante, repräsentiert werden [Plu99]. Diese Sichtweise wird von Plum durch die größere Effizienz dieses Vorgehens gerechtfertigt. Wir haben noch weitere Gründe hierfür: Ein

solcher Ersetzungsbegriff impliziert z. B. bei Programmgraphen im Übersetzerbau (vgl. Abschnitt 6.3), dass Werte mehrfach verwendet werden, statt ihre Berechnungen zu duplizieren. Dies entspricht einer impliziten *Eliminierung gemeinsamer Teilausdrücke*[17], einer Standardoptimierung [WG84]. Außerdem kann der SPO-Ansatz, der auf der Existenz der Pushout-Morphismen beruht, nur von m bzw. r erfasste Teile des Quell-Graphen H ändern; somit müssten, um das Kopieren von Teiltermen zu realisieren, die Morphismen m bzw. r dynamisch, d. h. für jeden Graphen H anders, erweitert werden. Im Gegensatz zu unserem Ansatz können Termgraphen nur azyklische Graphen darstellen. Wir gehen in Abschnitt 4.3.4 genauer darauf ein, wie wir in unserem Kontext das Kopieren von Graphen lösen, das aus dem SPO-Ansatz selbst heraus nicht möglich ist.

4.3.7 Anwendungsmodell

Nachdem wir nun festgelegt haben, was unter der Anwendung *einer* Graphersetzungsregel verstehen, wenden wir uns der Frage zu, wie mehrere Regeln anzuwenden sind. Hierbei sind mehrere Ansätze möglich:

1. Einzelne Regeln werden
   (a) explizit (und interaktiv) vom Benutzer ausgewählt.
   (b) vermittels einer Zugriffsschicht durch ein Programm, das normalerweise in einer konventionellen Programmiersprache implementiert ist, zur Anwendung gebracht.
2. Alle Regeln werden
   (a) manuell angeordnet und immer in dieser Reihenfolge angewandt (vgl. z. B. AGG [ERT99]).
   (b) automatisch nach bestimmten Gesichtspunkten angeordnet und immer in dieser Reihenfolge angewandt (vgl. etwa OPTIMIX [Ass00]).
   (c) zielgesteuert, wie bei *Bottom-Up-Zerteilern*, ausgewählt.
   (d) wie bei Chomskygrammatiken üblich wahllos angewandt.

Die Anwendungsmodelle sind so geordnet, dass jene mit größerer imperativer, d. h. algorithmischer oder manueller Kontrolle oben stehen. Das letzte Modell entspricht der Sichtweise der Graphgrammatiken, in denen keinerlei externe algorithmische Steuerung vorhanden ist. Nun darf man nicht vergessen, dass gewisse Fragestellungen, wie die Terminierung und Determiniertheit nur bei grammatikähnlichen Modellen, also 2b bis 2d, sinnvoll sind. Denn eine einzige XSPO-Graphersetzung, wie in Fall 1a terminiert immer und führt mit einem gegebenen Passungsmorphismus immer zu einem eineindeutigen Ergebnis!

Wir unterstützen alle Anwendungsmodelle – zumindest indirekt – obgleich wir auf drei davon besonderes Augenmerk legen. Der Fall 1a wird als eine interaktive Kommandoumgebung zur Verfügung gestellt, in der man Schritt für Schritt einzelne Graphersetzungen anwenden, den Graphen inspizieren sowie direkte Manipulationen am Graphen vornehmen kann. Dies hat sich bei der Entwicklung und prototypischen

[17] engl.: *common subexpression elimination* (CSE)

Implementierung neuer Graphersetzungsanwendungen als besonders hilfreich erwiesen. Wir verwenden Ansatz 1b in mehreren Ausprägungen. Hier unterscheidet sich vor allem die Art des steuernden Programms; im einfachsten Fall kann man eine zuvor manuell erprobte Folge von Graphersetzungen automatisch ablaufen lassen, was wiederum Fall 2a entspricht. Mehr dazu in Abschnitt 4.4.

Was ist das richtige Anwendungsmodell? Die Antwort auf diese Frage hängt von mehreren Faktoren ab: Der Mächtigkeit des Ersetzungsformalismusses, der typischen Gestalt der Graphen und Regeln sowie der Art und Weise, wie die Regeln zum Gesamtergebnis beitragen sollen. Betrachten wir nun die Zusammenhänge im Einzelnen. Wenn ein einfacher Ersetzungsformalismus vorliegt, dann ist eine vollautomatische Regelanwendung sinnvoll, insbesondere wenn die Regeln *per se* konfluent bzw. deterministisch sind. Dieser Fall liegt zum Beispiel bei der Syntaxanalyse mittels deterministisch kontextfreier Sprachen vor. Ebenso kann bei klassischen Codegeneratoren die Anwendung von Überdeckungsregeln generisch gesteuert werden, dabei werden etwaige Indeterminismen durch eine Kostensteuerung aufgelöst. Sind allerdings die spezifizierbaren Regeln so mächtig, dass Konfluenz im Allgemeinen nicht mehr gewährleistet werden kann, wie zum Beispiel bei kontextsensitiven Sprachen oder auch bei XSPO-Regeln, muss der Benutzer gewisse Anwendungsreihenfolgen vorgeben können. Auf diese Weise ist es möglich zumindest ein deterministisches Ergebnis zu produzieren oder, durch partielle Vorgaben der Anwendungsreihenfolge, sogar die Konfluenz der dann noch erlaubten Anwendungspfade zu erreichen. Letztlich kann eine solche Anwendungsreihenfolge durch sogenannte Schiffchen erzwungen werden (vergleiche zum Beispiel das Kapitel über Semi-Thue-Systeme in [GZ06]). Dies ist unserer Meinung nach jedoch ein Missbrauch von Graphersetzung, da solcherlei Transformationen normalerweise besser algorithmisch formulierbar sind.

Alle diese Überlegungen gelten freilich nur, wenn das gewünschte Gesamtergebnis ausschließlich durch den Ersetzungsformalismus erzielt werden soll. Wir wünschen uns allerdings auch eine Integration der Graphersetzung in eine konventionelle Hochsprache, mit dem Ziel, nur gewisse, mit Graphersetzungsregeln besonders gut spezifizierbare Teile eines Algorithmus damit zu realisieren. Hierbei wird klar, dass nur einzelne Ersetzungen gezielt angewendet werden sollen. Eine wahlloses Anwenden im Sinne von Graphgrammatiken [EKL91, ERT99] kann hier wegen des daraus resultierenden Mangels an Durchsichtigkeit und Vorhersagbarkeit nicht zum Ziel führen. Insbesondere ist es hierbei möglich, falls mehrere Passungsmorphismen existieren, einen basierend auf Informationen jenseits der Graphersetzungsregeln auszuwählen. Besser noch – es kann a priori festgestellt werden, in welchen Bereichen des Graphen die Passungsmorphismen zu suchen sind. Die Terminierung von solchen Algorithmen mit Unterstützung eines Graphersetzers ist offenbar weniger eine Frage der einzelnen Ersetzungsschritte, als der umgebenden, in einer konventionellen Hochsprache geschriebenen Steuerung.

Dass beim wahllosen Anwenden von Regeln die Ausdrucksstärke eng mit der Konfluenz bzw. dem Erhalt eines deterministischen Transformationsergebnisses sowie der Terminierung verknüpft ist, scheint zwar intuitiv klar zu sein, ist aber dennoch einer genaueren Betrachtung wert. Ersetzungsformalismen, bei denen die fortgesetzte wahllose Regelanwendung terminieren soll, müssen unbeschränktes Löschen und Hinzufügen von Ecken und Kanten sowie Multigraphen verbieten. Eine Möglichkeit die Terminierung der Anwendung einer Menge von Regeln zu garantieren, ist ausschließlich das Hinzufügen von Kanten zu erlauben (Multigraphen seien ver-

boten). In diesem Fall terminiert die Anwendung offenbar immer, wenn der Graph vollständig verbunden ist, wie Assmann mit seinen EARS[18] zeigt [Ass00]. Dadurch ist es aber zum Beispiel nicht möglich, eine Regelmenge anzugeben, die Standardaufgaben des Übersetzerbaus, wie die *Konstantenfaltung* und die *Eliminierung toten Codes*[19] auf einer graphbasierte Zwischensprache durchführt. Weitere Kombinationen von Einschränkungen erlauben jeweils das Eine oder das Andere zu formulieren, jedoch muss immer zwischen verschiedenen Formalismen gewechselt werden – manches bleibt nicht ausdrückbar.

Solcherlei Einschränkungen machen die Arbeit mit jenen Ersetzungsformalismen unnötig kompliziert, da ein Großteil der Schwierigkeit darin besteht, den Einschränkungen des Formalismus zu entgehen. Darum haben wir den XSPO-Ersetzungsformalismus so ausdrucksstark wie möglich gemacht und dadurch bewusst auf Eigenschaften wie Terminierung oder Konfluenz bei wahllosem Anwenden von Regeln einer Regelmenge verzichtet. Es sei hier noch angemerkt, dass die automatisch durch sogenannte Stratifikation angeordneten Regeln in Assmans Optimix [Ass00] durch XSPO-Graphersetzung leicht simulierbar sind. Natürlich sind auch den *Graphgrammatiken* ähnliche, von manchen auch *parallele Graphersetzungen* genannte, Verfahren auf Basis unseres Ansatzes implementierbar; man muss eben nur die entsprechende Anwendungssteuerung implementieren.

Programm 4.6: Graphmodell für Bauer und Ameisenhaufen

```
1  enum Country { NL=31, DE=49, NZ, GB }
2  node class Apple { weight : double; origin : Country; worms : int; }
3  abstract node class Red;
4  node class RedApple extends Apple, Red { tasty : boolean; }
5  node class ToffeeApple extends Apple;
6  node class AntHill { name : string; size : int; }
7  node class Farmer { angry : boolean; }
8  edge class AntTrail connect AntHill[2:5] -> Apple[*] { distance : float; }
9  edge class Sees connect Farmer[*] -> Apple[*],
10     Farmer[*] -> AntHill[*], Farmer[*] -> Farmer[*];
11 edge class Fancies extends Sees;
```

4.3.8 Notation

In diesem Abschnitt betrachten wir ein etwas längeres Beispiel, das wir einem unveröffentlichten Übersichtspapier von Kroll und Geiß zu GrGen.NET entnommen haben. Es zeigt an einem künstlichen Beispiel fast alle Möglichkeiten, die GrGen bei der Spezifikation von Graphmodellen und Graphersetzungsregeln erlaubt. Besonders bemerkenswert ist eine Variante der Schreibweise des Ersetzungsgraphen: Es müssen nur die Änderungen angegeben werden, der Rest wird automatisch übernommen (Schlüsselwort `modify`). Dies ändert nichts an den zugrunde liegenden Formalismen, sondern ist reiner *syntaktischer Zucker*. Des Weiteren sei hier wieder auf die Kurzreferenz im Anhang A.3 und eine detaillierte Beschreibung im Benutzerhandbuch von GrGen.NET verweisen [BG07].

[18] engl.: *edge addition rewrite system*
[19] engl.: *dead code elimination*

Programm 4.7: Regel mit `replace`

```
using AgriModel;
rule repRule(f:Farmer) : (Apple) {
 pattern {
  f -sa:Sees-> a:Apple\ToffeeApple
   <-:AntTrail- h:AntHill <-s:Sees- f;
  if { typeof(a) > Apple;
       f.angry == true; }
  negative { hom(f,o);
   h <-:Sees- o:Farmer -:Sees-> f; }
 }
 replace {
  a <-fan- f -s-> h;
  -fan:Fancies<sa>->;
  n:typeof(a) <-:Sees- f;
  eval { f.angry = false;
   n.worms = a.worms + h.size / 13;
  }
  return (a);
 }
}
```

Programm 4.8: Regel mit `modify`

```
using AgriModel;
rule modRule(f:Farmer) : (Apple) {
 pattern {
  f -sa:Sees-> a:Apple\ToffeeApple
   <-t:AntTrail- h:AntHill <-:Sees- f;

  // Zeilen 6-9 wie in repRule


 }
 modify {
  delete(t);
  -:Fancies<sa>->;


  // Zeilen 14-18 wie in repRule


 }
}
```

Beispiel: Bauer und Ameisenhaufen

Betrachten wir nun folgende Kurzgeschichte, die das zugegebenermaßen surreale Beispiel illustrieren soll. Das Graphmodell hierfür ist in Programm 4.6 zu sehen, die zugehörigen Graphersetzungsregeln werden in Programm 4.7 bzw. 4.8 gezeigt. Die eine Regel, die in zwei Varianten `repRule` und `modRule` angegeben ist, stellt die ganze Geschichte in sich dar.

> Ein uns bekannter Bauer (`repRule(f:Farmer)`) sieht (`f -sa:Sees-> a:Apple`) einen besonderen Apfel (`typeof(a) > Apple`), der allerdings kein karamellisierter Apfel ist (`a:Apfel\ToffeeApple`). Der Bauer (`f`) sieht ebenfalls einen Ameisenhaufen (`h:AntHill <-s:Sees- f`) und stellt fest, dass es eine Ameisenstraße (`<-:AntTrail-`) zwischen dem Haufen und jenem Apfel gibt (`a:Apfel\ToffeeApple <-:AntTrail- h`). Der Bauer schaut sich zunächst um (`negative`), ob da noch ein anderer Bauer ist, der ebenfalls den Ameisenhaufen sehen kann (`h <-:Sees- o:Farmer -:Sees-> f;`) – er könnte auch sich selbst sehen, wenn da z. B. ein Spiegel wäre, was wir nicht ausschließen wollen (`hom(f,o);`); er erblickt niemanden. Da der Bauer schon schlecht gelaunt ist (`f.angry == true;`), und er sich nicht beobachtet fühlt, zerstört er die Ameisenstraße (`delete(t);` bei `modRule`, `t` bleibt unerwähnt bei `repRule`). Jetzt ist der Bauer wieder fröhlich (`f.angry = false`), ihm gefällt der Apfel ausnehmend gut (`-fan:Fancies<sa>->`) und er will ihn sich für Weiteres merken (`return (a)`). Aufgrund plötzlicher massiver Quantenfluktuationen[20] entsteht aus dem Nichts ein neuer Apfel (`n`), von derselben Sorte wie jener Apfel (`n:typeof(a)`), aber mit mehr Würmern darin

[20] Wir benutzen hier dieses ungewöhnliche physikalische Phänomen, um zu erklären, dass bei der

(`n.worms = a.worms + h.size / 13`). Der Bauer sieht diesen ebenfalls (`n:typeof(a) <-:Sees- f`).

4.4 Graphersetzungssequenzen

Eine einfache und für viele Fälle ausreichende Technik zur Spezifikation einer Serie von einzelnen Anwendungen bestimmter Graphersetzungsregeln sind die von uns entwickelten *erweiterten Graphersetzungssequenzen*. Diese legen in einer, den booleschen und regulären Ausdrücken entlehnten Schreibweise fest, in welcher Reihenfolge und wie oft gewisse Graphersetzungsregeln angewandt werden sollen. Grob gesagt ist es mit erweiterten Graphersetzungssequenzen möglich, die Anwendungsmodelle 1a bis 2a aus Abschnitt 4.3.7 zu realisieren.

Die Graphersetzungssequenzen sind aus Sicht des Anwenders unserer Methodik gewissermaßen ein Ersatz für das Schreiben eines Programms in einer Hochsprache, das über die libGr einzelne Ersetzungen durchführt und den Ablauf steuert (siehe Anhang A.5 für ein Beispiel einer Regelanwendung über die libGr), also festlegt, welche Regel wie oft anzuwenden versucht wird. Die Vorteile, eine Graphersetzungssequenz für jene Aufgabe zu verwenden, sind mannigfaltig: Der Theoretiker kann die recht kompakte Semantik der Graphersetzungssequenzen benutzen, um Beweise zu führen, die ohne dieses Hilfsmittel, also direkt auf Basis eines konventionellen Programms, kaum möglich gewesen wären. Die Sequenzen sind recht kompakt und die Syntax leicht zu erlernen, sodass Anwendungen, die Graphersetzung benutzen, schneller und übersichtlicher zu gestalten sind. Von großem praktischem Wert ist die Integration der Graphersetzungssequenzen in die interaktive Umgebung GrShell, da so die Entwicklung von Anwendungen mithilfe eines Debuggers für Graphersetzungen möglich ist. Dieser Debugger erlaubt die Inspektion der Zwischenschritte der Graphersetzung, also: Passung finden, neue Graphelemente einfügen, alte Graphelemente löschen, direkt anhand einer Visualisierung des Arbeitsgraphen. Eine konventionelle Programmiersprache könnte eine solche Abstraktion nicht bieten.

Gegenüber der sonst üblichen Methoden der Regelanwendung (falls diese überhaupt Teil der Methodik sind), sind die von uns entwickelten Graphersetzungssequenzen sehr viel ausdrucksstärker und haben unter anderem mit der *Transaktionsklammer* (s. u.) sonst nicht zur Verfügung stehende mächtige Operatoren. Besonders schlagkräftig ist die Benutzung einer Kombination von Operatoren, die der *Aussagenlogik* entlehnt wurden und über den Erfolg einzelner Regel- und Sequenzanwendungen wachen, sowie den Iterationsoperatoren, die wir den *regulären Mengen* entlehnt haben.

Wichtig zum Verständnis ist hierzu, dass, falls die Regel an mehreren Stellen anwendbar ist, keinerlei Einfluss auf die Auswahl der tatsächlich zu ersetzenden Stelle im Graphen genommen wird. Es wird also irgendeine Passung (der möglichen) ausgewählt und das Bild dieser dann ersetzt. Dies ist implementierungsabhängig zulässig und geschieht nicht etwa zufällig. Es wird also weder garantiert, dass bei zwei gleichen Ausgangssituationen dieselben Passungen gewählt werden, noch dass sie jedes Mal verschieden sind. Der Benutzer hat also die Verantwortung, entweder konfluente Regeln zu schreiben, oder die Sequenzen imperativ so zu wählen, dass ohne Konfluenz das vom ihm gewünschte Ergebnis entsteht; was durch zusätzliche

Graphersetzung plötzlich Dinge aus dem Nichts entstehen. Dies hat offenbar in der realen Welt keine triviale Entsprechung.

Regeln, Bedingungen in den Regeln oder den Sequenzen immer zu erreichen ist. Die Frage der Nichtdeterminiertheit der Passungsauswahl wird auch in Abschnitt 4.2.9 behandelt.

Die Regelauswahl wird generell durch die Sequenzen gesteuert, wobei diese von links nach rechts unter Beachtung der üblichen booleschen Operatorpräzedenzen abgearbeitet werden. Durch den $-Operatorpräfix kann allerdings diese Reihenfolge für jeden einzelnen Operator aufgehoben werden. Es ist dann erlaubt, die Operanden in beliebiger Reihenfolge abzuarbeiten. Allerdings wird nicht vorgeschrieben, nach welchen Regeln die Operanden dann auszuwählen sind; dies kann durch die Implementierung geeignet gewählt werden. So kann zum Beispiel für eine spezielle Anwendung eine geeignete Verteilung gewählt, oder im Kontext der Transaktionsklammer (s. u.) beim Zurücksetzen jede mögliche Reihenfolge ausprobiert werden.

4.4.1 Syntax

Als Erstes definieren wir die Syntax der erweiterten Graphersetzungssequenzen. Im nächsten Abschnitt wird die Semantik zuerst informell und dann formal angegeben. Der Name *erweitere Graphersetzungssequenz* selbst ist historisch bedingt: Wir hatten zunächst deutlich eingeschränktere Operatoren, die im Gegensatz zu den jetzigen erweiterten Graphersetzungssequenzen außerdem nicht vollständig orthogonal waren. Es gibt drei Hauptgruppen von Operatoren: Sequenzoperatoren, welche die Regelanwendungen steuern, Iterationsoperatoren, welche für die Wiederholung von Sequenzen sorgen, und Transaktionsklammern, die sicherstellen, dass die Wirkung von Sequenzen nur bei vollem Erfolg festgeschrieben wird. Die binären Operatoren sind jeweils rechtsassoziativ.

Definition 4.21 (Syntax erweiterter Graphersetzungssequenzen) Sei $\mathcal{P}$ eine Menge von Graphersetzungsregeln sowie $M = (M_E, M_K)$ das gemeinsame Graphmodell der Regeln und bearbeiteten Arbeitsgraphen. Sei ferner $\mathcal{V}$ eine Menge von Variablen, denen Ecken, Kanten oder Teilgraphen des Arbeitsgraphen zugeordnet werden können.

$$
\begin{aligned}
p \in \mathcal{P} &\Rightarrow \mathtt{p} \in \mathcal{S} \\
p \in \mathcal{P} \wedge a_1, \ldots, a_n \in \mathcal{V} &\Rightarrow \mathtt{p}(\mathtt{a_1}, \ldots, \mathtt{a_n}) \in \mathcal{S} \\
p \in \mathcal{P} \wedge b_1, \ldots, b_m \in \mathcal{V} &\Rightarrow (\mathtt{b_1}, \ldots, \mathtt{b_m}) = \mathtt{p} \in \mathcal{S} \\
p \in \mathcal{P} \wedge a_1, \ldots, a_n, b_1, \ldots, b_m \in \mathcal{V} &\Rightarrow (\mathtt{b_1}, \ldots, \mathtt{b_m}) = \mathtt{p}(\mathtt{a_1}, \ldots, \mathtt{a_n}) \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} &\Rightarrow < \mathtt{s} > \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} &\Rightarrow (\mathtt{s}) \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} \wedge b \in \mathcal{V} &\Rightarrow \mathtt{b} = (\mathtt{s}) \in \mathcal{S} \\
b \in \mathcal{V} \wedge \mathrm{dom}(\mathrm{val}(b)) \in \mathbb{B} &\Rightarrow \mathtt{b} \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} &\Rightarrow \mathtt{s}[*] \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} &\Rightarrow \mathtt{s}[+] \in \mathcal{S}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{s} \in \mathcal{S} \wedge n, m \in \mathbb{N} &\Rightarrow \mathtt{s}[\mathtt{n} : \mathtt{m}] \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} \wedge n \in \mathbb{N} &\Rightarrow \mathtt{s}[\mathtt{n}] \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} \wedge n \in \mathbb{N} &\Rightarrow \mathtt{s}[\mathtt{n} : *] \in \mathcal{S} \\
\mathtt{s} \in \mathcal{S} &\Rightarrow \mathtt{!s} \in \mathcal{S} \\
\mathtt{s}_1, \mathtt{s}_2 \in \mathcal{S}, op \in \{\ \&,\ |,\ \hat{},\ \&\&,\ ||\ \} &\Rightarrow \mathtt{s}_1\ \mathtt{op}\ \mathtt{s}_2 \in \mathcal{S} \\
\mathtt{a}_1, \ldots, \mathtt{a}_n \in \mathcal{V} &\Rightarrow \mathtt{def}(\mathtt{a}_1, \ldots, \mathtt{a}_n) \in \mathcal{S} \\
&\quad \top, \bot \in \mathcal{S}
\end{aligned}
$$

Der $-*Operatorpräfix* kann mit allen binären und unären Operatoren, die eine Sequenz als Argument haben kombiniert werden (mit Ausnahme von $(\cdot)$ und $< \cdot >$). Das so induktive definierte $\mathcal{S}$ ist die Menge der *erweiterten Graphersetzungssequenzen* über der Regelmenge $\mathcal{P}$. Ein Element $s \in \mathcal{S}$ heißt erweiterte Graphersetzungssequenz (*XGRS*).

4.4.2 Semantik

Bevor wir im Folgenden eine exakte Definition für die XGRS-Semantik geben, wollen wir hier zunächst die Hintergründe ein wenig erhellen. Die Semantik ist ebenso wie die Syntax den booleschen und regulären Ausdrücken entlehnt. Allerdings sollte man nie aus dem Auge verlieren, dass Graphersetzungssequenzen und reguläre Ausdrücke wie auch boolesche Ausdrücke letztlich verschiedene Dinge sind.

Die XGRS-Sequenzen haben drei „Wirkungen“: Erstens verändert ihre Anwendung auf einen Arbeitsgraphen ebendiesen. Zweitens führen die Operatoren Buch darüber, ob ihre Operanden und auch sie selbst *erfolgreich* waren, wobei sie das Ergebnis der Operanden gemäß ihrer booleschen Semantik verknüpfen. Bei Iterationsoperatoren und anderem gilt Gesondertes. Der Erfolg einer Operation hängt also von ihrer Art ab: Die unbeschränkte Iteration $\mathtt{s}[*]$ ist immer erfolgreich, der boolesche &-Operator hingegen nur dann, wenn auch seine Operanden erfolgreich waren – was immer auch das für diese Operanden heißen mag. Fundiert wird der Erfolg durch die erfolgreiche oder *nicht* erfolgreiche Anwendung von Regeln sowie boolesche Variablen und Konstanten. Drittens werden Variablen durch die Zuweisungen verändert, wobei die Werte als Rückgabe von Regeln stammen können oder aber der Erfolgsinformation einer Sequenzen entsprechen.

Regelanwendung: $(\mathtt{b}_1, \ldots, \mathtt{b}_m) = \mathtt{p}(\mathtt{a}_1, \ldots, \mathtt{a}_n),\ \ldots$

Die elementarste Operation ist die Anwendung einer einzelnen Graphersetzungsregel $p \in \mathcal{P}$. Wenn die Regel Parameter hat, dann stehen die Variablen $(\mathtt{a}_1, \ldots, \mathtt{a}_n)$ für Ecken, Kanten und Teilgraphen des Arbeitsgraphen, die als Vorbelegungen dienen. Die ggf. vorhandenen Rückgabewerte der Regel werden in die Variablen $(\mathtt{b}_1, \ldots, \mathtt{b}_m)$ zurückgeschrieben; dies muss aber nicht geschehen. Variablen, die als Argumente auftreten aber nicht definiert sind, belassen das entsprechende Element im Mustergraphen ohne Vorbelegung.

Transaktionsklammer: $< \mathtt{s} >$

Die Wirkung der Sequenz $\mathtt{s}$ wird nur dann materialisiert, wenn sie insgesamt erfolgreich ausgeführt wurde.

Iteration: `s`[∗], `s`[`n`], ...

Der nachgestellte *Iterationsoperator* kann entweder die Sequenz `s` keinmal, einmal oder *unbeschränkt* oft ausführen. Im *beschränkten* Fall wird verlangt, dass die Sequenz eine gewisse Anzahl oft erfolgreich ausgeführt werden konnte, um selbst erfolgreich zu sein.

Der [∗]-Operator wiederholt die Anwendung der XGRS `s` solange, bis `s` nicht mehr erfolgreich anwendbar ist. Wir bezeichnen dies als unbeschränkte Iteration, da a priori nicht klar ist wann und ob überhaupt die Anwendung einer Sequenz `s`[∗] terminiert. Betrachten wir zum Beispiel eine Regel $\mathtt{T} \in \mathcal{P}$, die genau eine Ecke durch eine in allen Belangen identische ersetzt; eine solche Regel wird den Graphen unverändert lassen. Damit folgt, dass `T`[∗] nie terminiert, da `T` immer anwendbar ist.

Die beschränkte Iteration `s`[`n`] wiederholt die Anwendung der erweiterten Graphersetzungssequenz `s` genau n-mal. Die Anwendung ist insgesamt erfolgreich, falls die Sequenz `s` jedes Mal erfolgreich anwendbar war. Es gibt auch Mischformen die eine untere Schranke haben, aber keine obere.

Strikte Sequenzoperatoren: !, &, |, ^

Die *strikten* Sequenzoperatoren führen die Sequenz immer von links nach rechts komplett aus. Sie sind genau dann erfolgreich, wenn die ihre Operanden mit der entsprechenden booleschen Verknüpfung erfolgreich sind.

Faule Sequenzoperatoren: &&, ||

Die *faulen* Sequenzoperatoren führen zunächst nur den ersten Operanden aus. Der zweite Operand wird nur ausgeführt, falls noch nicht feststeht, ob die Operation insgesamt erfolgreich ist oder nicht.

Definitionstest: `def`($\mathtt{a}_1, \ldots \mathtt{a}_n$)

Der pseudo-Operator `def` ist genau dann erfolgreich, wenn alle Variablen $\mathtt{a}_1$, ..., $\mathtt{a}_n$ gültige Elemente des Arbeitsgraphen enthalten. Eine Variable a kann „Ungültiges“ enthalten, wenn entweder noch nicht an a zugewiesen wurde oder Regelanwendungen seit der Zuweisungen die zugewiesenen Graphelementen aus dem Arbeitsgraphen entfernt haben.

Steuerungsvariablen: `b` = (`s`)

Ob die Sequenz erfolgreich war oder nicht, kann in einer Steuerungsvariablen b abgespeichert werden. Diese Variable kann an anderer Stelle als Operand eingesetzt werden.

Nichtdeterminismus Operatorenmodifikator: $-*Operatorpräfix*

Ein vorangestellter $-Operatorpräfix hebt die Auswertungsreihenfolge des nachfolgenden Operators auf. Also kann z. B. die XGRS $\mathtt{s}_1\$\&\mathtt{s}_2\$\&\mathtt{s}_3$ unter anderem die Wirkung von $\mathtt{s}_1\&\mathtt{s}_2\&\mathtt{s}_3$ als auch $\mathtt{s}_3\&\mathtt{s}_1\&\mathtt{s}_2$ entfalten. Die Auswahl unter den Möglichkeiten geschieht beliebig, muss aber nicht zufällig sein. Vielmehr ist jede Reihenfolge der Operanden $\mathtt{s}_i$ zulässig. Der Sinn dieser Operation liegt im expliziten Erlauben simultaner Ersetzungen und der benutzerspezifischen Anpassung der Regelauswahl in einer einheitlichen Theorie. Anmerkung: s\$[2] ist nicht sinnlos. Sei $s = a\$\&b$ damit ist auch $b\&b\&a\&a$ zulässig.

Definition 4.22 (Semantik erweiterter Graphersetzungssequenzen) Für die Anwendung einer Graphersetzungssequenz $s \in \mathcal{S}$ auf einen Arbeitsgraphen H schreiben wir

$$H \overset{s}{\Longrightarrow} H'$$

wobei der Ergebnisgraph H' durch $H' = [\![\mathtt{s}]\!]\sigma$ mit $\sigma = (H, \uparrow\uparrow, \bot)$ bestimmt ist. Der Zustand der Auswertung $\sigma = (H, \mathrm{val}, z)$ besteht aus dem jeweiligen Arbeitsgraphen $H \in \mathcal{I}_M$, einer Variablenbelegungsfunktion $\mathrm{val} : \mathcal{V} \to E_H \cup K_H \cup \mathcal{I}_M$ und einem booleschen Wert z. Die denotationelle *Auswertungsfunktion für erweiterte Graphersetzungssequenzen* $[\![\cdot]\!] : \mathcal{I}_M \times (\mathcal{V} \to E_H \cup K_H \cup \mathcal{I}_M) \times \mathbb{B} \to \mathcal{I}_M \times (\mathcal{V} \to E_H \cup K_H \cup \mathcal{I}_M) \times \mathbb{B}$ ist wie folgt festgelegt:

Regelanwendung

$$[\![(\mathtt{b}_1, \ldots, \mathtt{b}_m) = \mathtt{p}(\mathtt{a}_1, \ldots, \mathtt{a}_n)]\!]\sigma = \begin{cases} (H', \mathrm{val}', \top) & \text{falls } H_\sigma \overset{p(\alpha_1,\ldots,\alpha_n),m}{\Longrightarrow} (H', (\beta_1, \ldots, \beta_m)), \\ (\sigma_H, \sigma_{\mathrm{val}}, \bot) & \text{sonst.} \end{cases}$$

wobei

$$m \in \mathrm{match}(L, H_\sigma, c, m', \mathcal{N}, \mathrm{hom})$$

für die Graphersetzungsregel $p : (L, c, \mathrm{arg}, \mathcal{N}, \mathrm{hom}) \xrightarrow{r} (R, e, \mathrm{ret})$ beliebig gewählt sei. Vor der Anwendung von p setzen wir

$$\alpha_i := \mathrm{val}(a_i) \qquad \text{falls } \mathrm{val}(a_i) \neq \uparrow \text{ mit } i \in \{1, \ldots, n\}$$

sodass wir mit den Argumenten den Vorbelegungsmorphismus m' bestimmen können. Nach der Anwendung von p setzen wir dann

$$\mathrm{val} : b_i \mapsto \beta_i \qquad \text{mit } i \in \{1, \ldots, m\}$$

um die Rückgabewerte in Variablen zu sichern. Diese Regelanwendung ist wohldefiniert, wenn $\{1, \ldots, n\} \subseteq \mathrm{def}(\mathrm{arg})$ und $\{1, \ldots, m\} \subseteq \mathrm{def}(\mathrm{ret})$ gilt. Für eine Regelanwendung ohne Argumente oder Rückgabe lassen wir den entsprechenden Schritt einfach weg. Falls wir keine Argumente haben, ist der Vorbelegungsmorphismus leer, also $m' = \uparrow\uparrow$.

boolesche Sequenzoperatoren

$$\begin{aligned} [\![!\mathtt{s}]\!]\sigma &= (H_{\sigma'}, \mathrm{val}_{\sigma'}, \neg z_{\sigma'}) \quad \text{mit } \sigma' = [\![\mathtt{s}]\!]\sigma \\ [\![\mathtt{s}_1 \& \mathtt{s}_2]\!]\sigma &= (H_{\sigma'}, \mathrm{val}_{\sigma'}, z_{\sigma'} \wedge z_{\sigma''}) \quad \text{mit } \sigma' = [\![\mathtt{s}_1]\!]\sigma, \sigma'' = [\![\mathtt{s}_2]\!]\sigma' \\ [\![\mathtt{s}_1 | \mathtt{s}_2]\!]\sigma &= (H_{\sigma'}, \mathrm{val}_{\sigma'}, z_{\sigma'} \vee z_{\sigma''}) \quad \text{mit } \sigma' = [\![\mathtt{s}_1]\!]\sigma, \sigma'' = [\![\mathtt{s}_2]\!]\sigma' \\ [\![\mathtt{s}_1 \hat{} \mathtt{s}_2]\!]\sigma &= [\![(\mathtt{s}_1 | \mathtt{s}_2) \& !(\mathtt{s}_1 \& \mathtt{s}_2)]\!]\sigma \\ [\![\mathtt{s}_1 \&\& \mathtt{s}_2]\!]\sigma &= \begin{cases} \sigma' & \text{falls } z_{\sigma'} = \bot \text{ mit } \sigma' = [\![\mathtt{s}_1]\!]\sigma, \\ [\![\mathtt{s}_1 \& \mathtt{s}_2]\!]\sigma & \text{sonst.} \end{cases} \\ [\![\mathtt{s}_1 || \mathtt{s}_2]\!]\sigma &= \begin{cases} \sigma' & \text{falls } z_{\sigma'} = \top \text{ mit } \sigma' = [\![\mathtt{s}_1]\!]\sigma, \\ [\![\mathtt{s}_1 | \mathtt{s}_2]\!]\sigma & \text{sonst.} \end{cases} \end{aligned}$$

Iterationsoperatoren

$$[\![\mathtt{s}[*]]\!]\sigma = \begin{cases} [\![\mathtt{s\&s}[*]]\!]\sigma & \text{falls } z_{\sigma'} = \top \text{ mit } \sigma' = [\![\mathtt{s}]\!]\sigma, \\ [\![\top]\!]\sigma & \text{sonst.} \end{cases}$$

$$[\![\mathtt{s}[+]]\!]\sigma = [\![\mathtt{s\&s}[*]]\!]\sigma$$

$$[\![\mathtt{s}[\mathtt{n}]]\!]\sigma = [\![\underbrace{\mathtt{s\&}\ldots\mathtt{\&s}}_{n\text{-mal}}]\!]\sigma$$

$$[\![\mathtt{s}[\mathtt{n}:\mathtt{m}]]\!]\sigma = [\![\mathtt{s}[\mathtt{n}] \mid \mathtt{s}[\mathtt{d}]]\!]\sigma \quad \text{mit } d = m - n$$

$$[\![\mathtt{s}[\mathtt{n}:*]]\!]\sigma = [\![\mathtt{s}[\mathtt{n}] \mid \mathtt{s}[*]]\!]\sigma$$

Klammern und Variablen

$$[\![(\mathtt{s})]\!]\sigma = [\![\mathtt{s}]\!]\sigma$$

$$[\![\mathtt{b} = (\mathtt{s})]\!]\sigma = (H_{\sigma'}, \mathrm{val}', z_{\sigma'})$$
$$\text{mit } \mathrm{val}' = \mathrm{val}_{\sigma'}, \mathrm{val}' : b \mapsto z_{\sigma'} \text{ und } \sigma' = [\![\mathtt{s}]\!]\sigma$$

$$[\![< \mathtt{s} >]\!]\sigma = \begin{cases} \sigma' & \text{falls } z_{\sigma'} = \top \text{ mit } \sigma' = [\![\mathtt{s}]\!]\sigma, \\ [\![\bot]\!]\sigma & \text{sonst.} \end{cases}$$

$$[\![\bot]\!]\sigma = (H_\sigma, \mathrm{val}_\sigma, \bot)$$

$$[\![\top]\!]\sigma = (H_\sigma, \mathrm{val}_\sigma, \top)$$

$$[\![\mathtt{b}]\!]\sigma = (H_\sigma, \mathrm{val}_\sigma, \mathrm{val}(b))$$

$$[\![\mathtt{def}(\mathtt{a}_1, \ldots \mathtt{a}_n)]\!]\sigma = \left(H_\sigma, \mathrm{val}_\sigma, \bigwedge_{i=1}^{n} \mathrm{val}(a_i) \neq \uparrow\right)$$

$-Operatorpräfix bei booleschen Sequenzoperatoren

Die Bedeutung des $-Operatorpräfix für binäre Operatoren ist wie folgt gegeben, wobei $\mathtt{op} \in \&, |, \hat{}, \&\&, ||$ sei:

$$[\![\mathtt{s}_1\ \$\mathtt{op}\ \mathtt{s}_2]\!]\sigma = \begin{cases} [\![\mathtt{s}_1\ \mathtt{op}\ \mathtt{s}_2]\!]\sigma & \text{entweder,} \\ [\![\mathtt{s}_2\ \$\mathtt{op}\ \mathtt{s}_1]\!]\sigma & \text{oder.} \end{cases}$$

$-Operatorpräfix bei Iterationsoperatoren

Bei Iterationsoperatoren wird der $-Operatorpräfix an die darunter liegenden Verküpfungen weitergereicht:

$$[\![\mathtt{s}\$[*]]\!]\sigma = \begin{cases} [\![\mathtt{s}\ \$\&\ \mathtt{s}\$[*]]\!]\sigma & \text{falls } z_{\sigma'} = \top \text{ mit } \sigma' = [\![\mathtt{s}]\!]\sigma, \\ [\![\top]\!]\sigma & \text{sonst.} \end{cases}$$

$$[\![\mathtt{s}\$[+]]\!]\sigma = [\![\mathtt{s}\ \$\&\ \mathtt{s}\$[*]]\!]\sigma$$

$$[\![\mathtt{s}\$[\mathtt{n}]]\!]\sigma = [\![\underbrace{\mathtt{s}\ \$\&\ \ldots\ \$\&\ \mathtt{s}}_{n\text{-mal}}]\!]\sigma$$

$$[\![\mathtt{s}\$[\mathtt{n}:\mathtt{m}]]\!]\sigma = [\![\mathtt{s}\$[\mathtt{n}] \mid \mathtt{s}\$[\mathtt{d}]]\!]\sigma \quad \text{mit } d = m - n$$

$$[\![\mathtt{s}\$[\mathtt{n}:*]]\!]\sigma = [\![\mathtt{s}\$[\mathtt{n}] \mid \mathtt{s}\$*]\!]\sigma$$

Obschon die Syntax und Semantik von XGRS den booleschen oder in Teilen auch den regulären Ausdrücken entlehnt ist, sollte diese Analogie nicht überstrapaziert werden: Offenbar werden bei erweiterten Graphersetzungssequenzen keine Texte erkannt[21], sondern Graphersetzungsregeln angewandt. Bei der Anwendung von Graphersetzungsregeln wird die Struktur, auf der die Passungen gesucht werden, fortlaufend geändert – bei regulären Ausdrücken bleibt der Text unverändert. Bevor wir auf die praktischen Anwendungen von XGRS eingehen, betrachten wir im Folgenden die Mächtigkeit dieses Verfahrens.

4.4.3 Turingvollständigkeit

In Abschnitt 4.3.7 haben wir kurz die Terminierung von Graphersetzungssystemen angesprochen und festgestellt, dass einzelne XSPO-Regeln immer terminieren. Erweiterte Graphersetzungssequenzen tun dies allerdings mitnichten. Wir zeigen sogar – in gewisser Weise – das Gegenteil:

Satz 4.23 (Turingvollständigkeit) Erweiterte Graphersetzungssequenzen sind turingvollständig.

Beweis: Gegeben sei eine Turingmaschine mit beidseitig unendlichem Band

$$TM = (Q, \Sigma, \Gamma, \delta, q_0, \#, F).$$

O. B. d. A. wählen wir die Zustandsmengen

$$F = \{f\} \subseteq Q \subset \mathrm{dom}(\texttt{Integer})$$

und die Alphabete

$$\Sigma \subseteq \Gamma \subset \bigcup_{s \in \mathrm{dom}(\texttt{String}) \wedge |s|=1} s$$

mit einem Buchstaben $\# \notin \Sigma$.

Wir zeigen, dass die Graphersetzungssequenz $\texttt{TM} = (\texttt{X}_\texttt{R}|\texttt{X}_\texttt{L}|\texttt{T}_\texttt{1}|\cdots|\texttt{T}_\texttt{n})[*]$, deren Regeln wir im Folgenden angeben, eine Turingmaschine simuliert. Die Startkonfiguration $q_0\omega$ mit $\omega = \omega_1 \ldots \omega_j \in \Sigma$ wird durch Graphen, wie den in Abbildung 4.13 gezeigten, repräsentiert. Wir interpretieren solcherlei Graphen wie folgt: Die Singleton-Ecke e_1 vom Typ `Zustand` beinhaltet als Attribut `q` den ganzzahligen Maschinenzustand. Mit einer Kante vom Typ `kopf` zeigt e_1 die jeweilige Position des Lesekopfes auf dem Band an. Letzteres wiederum wird durch eine perlenschnurartige Verkettung von Ecken des Typs `Band` simuliert, die den Buchstaben ω_i der jeweiligen Zelle durch den Wert eines Attributs `bu` darstellen.

Zu jedem Zustandsübergang $\delta : (q, e) \mapsto (q', a, d)$ konstruieren wir eine Graphersetzungsregel $\texttt{T}_\texttt{i}$. Diese haben pro jeweiliger Richtung $d \in \{R, L, N\}$ ein anderes Schema. In Abbildung 4.14 ist exemplarisch das Regelschema für $d = R$ zu sehen, die restlichen Schemata gestalten sich analog und sind in Anhang B.2 vollständig angegeben. Weiterhin benötigen wir zwei Regeln $\texttt{X}_\texttt{R}$ und $\texttt{X}_\texttt{L}$, die das

[21] Die regulären Ausdrücke selbst stammen aus der Statistik [Kle56], wo sie unter dem Begriff *regular events* bekannt wurden.

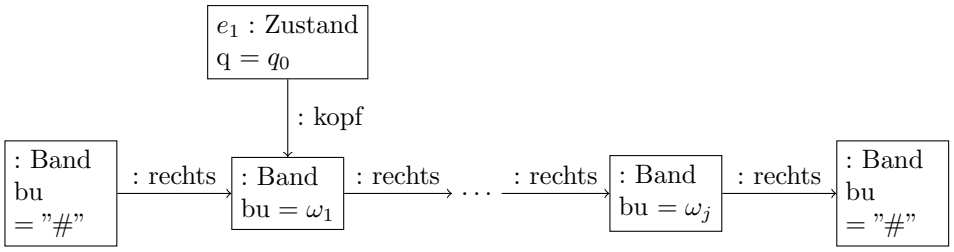


Abbildung 4.13: Die Startkonfiguration $q_0\omega$ mit $\omega = \omega_1 \dots \omega_j \in \Sigma$

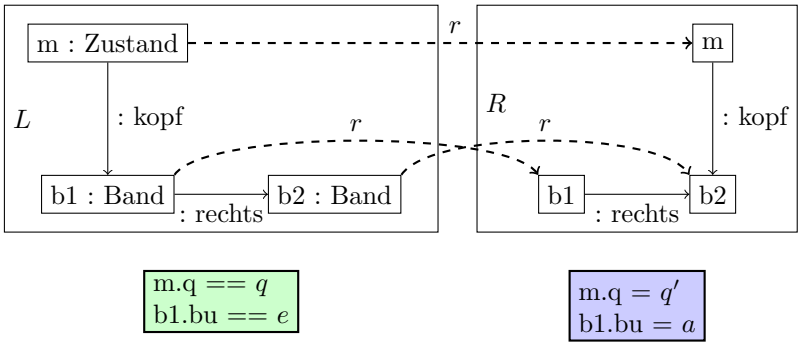


Abbildung 4.14: Schematische Graphersetzungsregel für einen Zustandsübergang $\delta : (q, e) \mapsto (q', a, R)$

beidseitig unendliche Band bei Bedarf stückweise herstellen. Denn offensichtlich kann jede Folgekonfiguration höchstens *einen* Schritt weiter auf bisher unbenutztes Band vordringen. Also genügt es vor jeder eigentlichen Ersetzung, die einer Folgekonfiguration der originalen Turingmaschine entspricht, das Band nach links und rechts zu erweitern. Die Regel $\mathtt{X_R}$ in Abbildung 4.15 bewerkstelligt ebendiese Erweiterung nach rechts. Entsprechend erweitert eine zweite Regel $\mathtt{X_L}$ nach links (siehe Anhang B.2). Das Band wird auf diese Weise zwar an beiden Enden in

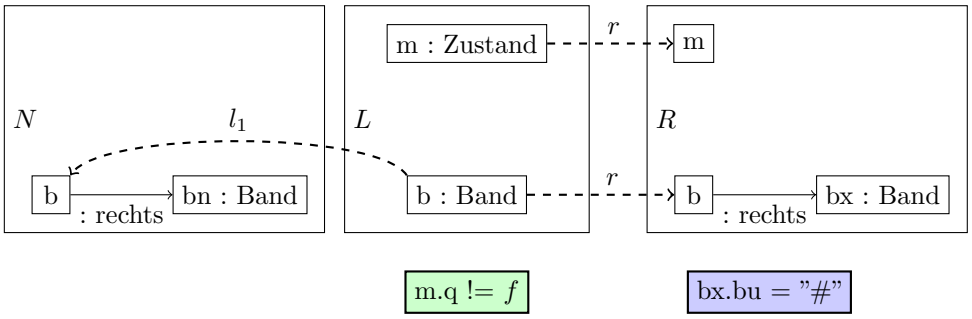


Abbildung 4.15: Die Regel $\mathtt{X_R}$ erweitert das Band nach Rechts

jedem Schritt um ein #-Zeichen erweitert, was aber höchstens unnötig, nicht jedoch schädlich sein kann. Falls die simulierte Turingmaschine den Endzustand f erreicht, endet, wegen der Bedingung $m.q \neq f$ für die Ecke m, auch die Anwendbarkeit der Regeln $\mathtt{X_R}$ und $\mathtt{X_L}$. Schließlich sind dann keine Regeln mehr anwendbar und die Anwendung der erweiterten Graphersetzungssequenz $(\mathtt{X_R}|\mathtt{X_L}|\mathtt{T_1}|\cdots|\mathtt{T_n})[*]$ terminiert.

Von der Startkonfiguration ausgehend ist jeweils nur ein einziges $\mathtt{T_i}$ anwendbar. Denn per Konstruktion muss das passende Zustands/Eingabebuchstaben-

Paar $(q, e) \in Q \times \Gamma$ vorliegen. Da δ eine Funktion ist, kann dies jeweils nur für eine Regel der Fall sein. Also ist jedes $\texttt{T}_\texttt{i}$ offenbar genau dann anwendbar, wenn die echte Turingmaschine den entsprechenden Übergang $\delta(q, e) \mapsto (q', a, d)$ vornimmt und der resultierende Graph korrespondiert überdies mit der Folgekonfiguration von (q, e). Damit liefert die Anwendung der erweiterten Graphersetzungssequenz `TM` ausgehend von der Startkonfiguration eine Folge von Graphen, die der Konfigurationsfolge der zu simulierenden Turingmaschine TM entsprechen. ■

Die Turingmaschine kann anscheinend sehr elegant mit einer XGRS simuliert werden. So ist die simulierte Turingmaschine quasi eine grafische Veranschaulichung des mathematischen Modells: Einzelne Schritte der Maschine entsprechen unmittelbar einzelnen Graphersetzungen. Im Anhang B.2 findet sich die vollständige Spezifikation der hier besprochenen Graphersetzungssequenzen sowie eine Beispielsimulation. Dies ist ein Zeichen der gewünschten Ausdrucksstärke unserer *Regelanwendungssprache*.

KAPITEL 5

EFFIZIENTE GRAPHMUSTERSUCHE

In diesem Kapitel stellen wir zwei von uns entwickelte neue Ansätze zur Graphmustersuche dar: Eine Reduktion auf relationale Algebra (Abschnitt 5.1) sowie eine direkte[1] Lösung mittels einer virtuellen Maschine und Suchprogrammen (Abschnitte 5.2 und 5.3). Abschließend gehen wir kurz auf die Leistungsfähigkeit unserer beiden Ansätze ein. Kapitel 9 beinhaltet einen Vergleich der Geschwindigkeit unserer Ansätze mit den prominentesten Graphersetzungssystemen.

Die Idee, das Finden einer Passung mittels *relationaler Algebra*[2] zu beschreiben, hat den wesentlichen Vorteil, diese komplexe Suchaufgabe durch ein *relationales Datenbankmanagementsystem*[3] erledigen zu lassen. Ein weiterer offensichtlicher Vorteil liegt in der Persistenz, die uns das Datenbanksystem zur Verfügung stellt. Ohne diese Abbildung auf relationale Algebra müsste man die von einigen Anwendungen benötigte Persistenz erst aufwendig implementieren.

Im Gegensatz zu den indirekten Ansätzen wie zum Beispiel dem *Constraint Satisfaction Programming (CSP)* und der relationalen Algebra, die die Mustersuche auf andere Probleme reduzieren, steht bei unserem direkten Verfahren die Ausnutzung der Informationen über den Muster- und Arbeitsgraphen sowie ggf. der Ersetzungssequenz im Sinne von XGRS (siehe Definition 4.22) im Vordergrund. Wenn wir einen einzelnen *Ersetzungsschritt* betrachten, können prinzipiell der Graphzusammenhang, die Typinformation und die Attribute herangezogen werden, um das Muster effizient zu suchen. Im Falle mehrerer, aufeinanderfolgender Ersetzungsschritte kann es möglich sein, aus Vorherigem auf Künftiges zu schließen.

Als Kuriosum sei hier noch angemerkt, dass Varró et al. zu den beiden von uns entwickelten Ansätzen zur Graphmustersuche, jeweils im gleichen Jahr, recht ähnliche Ansätze veröffentlichten (vgl. dazu auch Abschnitt 3.1.4.2 und 3.1.3.3).

5.1 Relationale Algebra

Für die *Reduktion* der *Graphmustersuche* auf relationale Algebra benötigen wir zweierlei: Wir müssen erstens GR-Graphen (vgl. Definition 4.2) als relationales Schema darstellen (Abschnitt 5.1.1) und zweitens die Bedingungen eines GR-Musters (vgl.

[1] Direkt bedeutet hier, dass der Problemraum – nämlich Graphen und Graphmorphismen – nicht verlassen wird und eine handgeschneiderte Lösung entworfen wird, die so besondere Gegebenheiten von Graphen ausnutzen kann.

[2] Wir gehen hier davon aus, dass der Leser mit den elementaren Begriffen der relationalen Algebra [Cod70] vertraut ist.

[3] engl.: *relational database management system* (RDBMS), kurz *Datenbanksystem*

Definition 4.13) in relationale *Abfragen* umsetzen. Für letztere Aufgabe haben wir zwei Ansätze gewählt. Nämlich die eher simple Benutzung von `WHERE`-basierten Abfragen (Abschnitt 5.1.2), die sich voll auf die automatisierte Optimierung der Abfrage durch das Datenbanksystem verlässt, sowie ein Ansatz der `JOIN`-basierte Abfragen mit erzwungener Anordnung verwendet und so die Planungshoheit dem Datenbanksystem entzieht (Abschnitt 5.1.3).

Obwohl wir uns hier nur mit der Mustersuche beschäftigen, müssen auch die Änderungen am Graphen, welche eine Ersetzung mit sich bringt oder die der Anwender direkt auslöst, durchgeführt werden; da dies jedoch weder von der Berechnungs- noch von der Implementierungskomplexität ein nennenswerter Punkt ist, sei es hier als gegeben vorausgesetzt. In der Diplomarbeit von Hack [Hac03] und der Studienarbeit von Grund [Gru04] wurden die hier dargestellten Ergebnisse ausgearbeitet und in der ersten Implementierung von GrGen umgesetzt.

5.1.1 Repräsentation der Graphen

Die Gr-Muster können auf verschiedene Weisen in relationaler Algebra dargestellt werden. Im Folgenden diskutieren wir die relevanten Möglichkeiten für die Graphen, ohne zunächst auf Attribute einzugehen:

- Pro Ecken- und Kantentyp eine eigene Relation.
- Eine Relation für alle Ecken, eine zweite für alle Kanten.
- Insgesamt nur eine Relation für Ecken und Kanten.

Für die Repräsentation von Attributen gibt es wiederum mehrere Möglichkeiten:

- Für die Attribute jedes Ecken- und Kantentyps jeweils eine eigene Relation.
- Alle Attribute von Ecken und Kanten jeweils in einer Relation.
- Attribute in der gleichen Relation wie auch ihre zugehörigen Ecken und Kanten.

Falls wir pro Ecken- und Kantentyp eine eigene Relation benutzen, können wir zwar die Attribute direkt dort mitspeichern, ohne Verschnitt zu produzieren. Durch die integrierte Attributspeicherung könnte man außerdem Geschwindigkeitsvorteile erwarten, da eine Indirektion beim Zugriff entfällt. Jedoch ist damit nicht zu erreichen, dass eine Vererbungsbeziehung auf Typen innerhalb von relationalen Abfragen ausdrückbar ist: Für jede Abfrage muss man schon vorab wissen, welche Tabelle angesprochen werden soll. Also sind entweder alle möglichen Abfragen im Vorhinein zu generieren, was zu einer exponentiellen Auffächerung in der Typhierarchie führt, oder man muss mit elementaren Abfragen arbeiten und die Teilergebnisse hinterher – außerhalb des Datenbanksystems – aggregieren, was allerdings zu einem hohen Laufzeitwasserkopf führt, da bei Datenbanksystemen jede Abfrage, unabhängig von dem eigentlichen Berechnungsaufwand, eine gewisse Zeit[4] benötigt und sich diese hier (durch die mehreren Abfragen) vervielfachen kann.

Wenn man je eine Relation für alle Ecken und Kanten hat, ist es zwar leicht möglich die Typbedingungen innerhalb des Datenbanksystems zu formulieren, aber es

[4]engl.: *round trip time*

Schema	Attribute
`nodes`	`node_id, node_type_id`
`node_attrs`	`node_id, [Attribute je nach Graphmodell]`
`node_type_rel`	`node_type_id, isa_node_type_id`
`edges`	`edge_id, edge_type_id, src_id, tgt_id`
`edge_attrs`	`edge_id, [Attribute je nach Graphmodell]`
`edge_type_rel`	`edge_type_id, isa_edge_type_id`

Tabelle 5.1: Schemata zur Darstellung eines Graphen in relationaler Algebra.

ergeben sich Probleme bei der Speicherung von Attributen. Eine integrierte Speicherung ist eher nachteilig, da hier die Tabellen, aufgrund der dann nötigen Vereinigung über alle Attribute, sehr groß werden. Also sollte man die Attribute getrennt abspeichern. Falls hier pro Graphelementtyp eine Relation für Attribute erstellt wird, resultiert dies in den gleichen Problemen wie bei der getrennten Speicherung der Graphelemente selbst. Darum kann es sinnvoll sein, den Verschnitt, der durch die Vereinigung von jeweils den Ecken- und Kantenattributen entsteht, hinzunehmen.

Die Speicherung in einer einzigen Tabelle ergibt keinen erkennbaren Vorteil, weist allerdings die Nachteile des vorherigen Ansatzes in nochmals verstärkter Form auf, da nun tatsächlich alle Attribute in einer Relation vereinigt werden.

Auf einem Schema, das je eine Relation für alle Ecken und Kanten hat, sowie die Attribute in zwei extra Tabellen speichert, bauen wir unsere weiteren Überlegungen auf. Tabelle 5.1 zeigt ein dazu kompatibles relationales Schema. Zusätzlich kann es je nach konkretem Graphmodell bei den Relationen `node_attrs` und `edge_attrs` beliebig viele weitere Stellen geben, welche die Ecken- und Kantenattribute aufnehmen und jeweils abhängig vom Graphmodell ausgestaltet werden. Wir werden später die `node_type_rel` und `edge_type_rel` nicht zum Bestimmen der Passungen heranziehen – vielmehr wird dies direkt über die Primärschlüssel ausprogrammiert. Jene zwei Relationen dienen nur dazu, Daten des Graphmodells zu speichern; sie können ggf. über die libGr abgefragt werden.

Dieses Schema ist in zweiter Normalform, da alle Attributwerte (Attribute der Relation) elementar sind, und Nichtschlüssel-Attribute von genau einem Primärschlüssel abhängig sind. Allerdings ist wegen der transitiven Abhängigkeit des Attributes `node_type_id` vom Schema `nodes` über `node_id` in Schema `node_attrs` hinzu den Graph-Attributen, die ebendort gespeichert werden, das Schema nicht in dritter Normalform. Dies nehmen wir jedoch hin, da wir so auf die Attribute uniform zugreifen können und auch beim Umtypisieren effizienter sind.

Bei der Umsetzung der Mustersuche in relationale Abfragen können mehrere Wege beschritten werden: Wenn wir den Weg der Abwälzung der Komplexität auf das RDBMS konsequent verfolgen, dann können wir dies durch eine maximal deklarative Formulierung mittels `WHERE`-basierten Abfragen erreichen. Dieser Abfragetyp impliziert keinerlei Reihenfolge, in der die Dinge zu suchen sind, was dem RDBMS zwar maximale Freiheit gibt, aber eben auch voraussetzt, dass dies erfolgreich getan werden kann (Abschnitt 5.1.2). Andererseits ist es auch möglich, diese Planung schon weitgehend selbst zu erledigen und diese als Abfragen mit explizitem *JOIN* an das RDBMS zu übergeben (Abschnitt 5.1.3). Hier kann sogar jegliche Umordnung dem RDBMS verboten werden, um die eigene Planung nicht zu zerstören.

5.1.2 `WHERE`-basierte Abfragen

Der prinzipielle Aufbau einer `WHERE`-basierten Abfrage ist in Programm 5.1 zu sehen, wobei anstelle der Kommentare – je nach Gr-Muster – Entsprechendes eingefügt werden muss. In der `SELECT`-Klausel werden die numerischen Bezeichner der Arbeitsgraphelemente ausgewählt; sie stellen die Rückgabe einer erfolgreichen Abfrage dar. Für jedes Mustergraphelement wird in der `FROM`-Klausel ein *Alias* auf die Ecken- oder der Kantentabelle des Arbeitsgraphen erstellt. Aus diesen Aliase selektieren wir in der nachfolgenden `WHERE`-Klausel die jeweils zum entsprechenden Mustergraphelement passenden Arbeitsgraphelemente. Diese Selektion enthält Entsprechungen für alle Bedingungen, mit denen ein Gr-Muster eingeschränkt werden kann: Typbedingungen, Attributbedingungen und Homomorphiebedingung – selbst Vorbelegungen können durch Bedingungen ausgedrückt werden. Natürlich wird auch der Zusammenhang des Mustergraphen durch sogenannte Inzidenzbedingungen eingefordert.

Programm 5.1: Prinzipieller Aufbau `WHERE`-basierter Abfragen

```
1  SELECT /* Ecken- und Kanten des Musters */
2  FROM   /* Ecken, Kanten, Eckenattribute, Kantenattribute */
3  WHERE  /* Inzidenz-, Typ-, Attribut-, Homomorphiebedingung, ... des Musters */
4  AND NOT EXISTS ( SELECT *
5                   FROM   /* Ecken, Kanten, Eckenattribute, Kantenattribute */
6                   WHERE  /* Inzidenz-, Typ-, Attribut-, Homomorphiebedingung, ...
7                             des 1. NAC */
8                 )
9  ...
10 AND NOT EXISTS ( SELECT *
11                  FROM   /* Ecken, Kanten, Eckenattribute, Kantenattribute */
12                  WHERE  /* Inzidenz-, Typ-, Attribut-, Homomorphiebedingung, ...
13                            des k. NAC */
14                )
```

Eine NAC ist in einer zu den (positiven) Mustern analogen Unterabfrage darzustellen. Der einzige Unterschied ist, dass es bei negativen Anwendungsbedingungen nur auf die Existenz ankommt, nicht aber auf die korrekte Auswahl der Graphelemente. Darum kann in der `SELECT`-Klausel ein Stern stehen. Den Anschluss an den Rest des Musters wird durch Blockschachtelung der Gültigkeitsbereiche in SQL und der damit verbundenen Wiederverwendung von außen definierten Aliase erreicht.

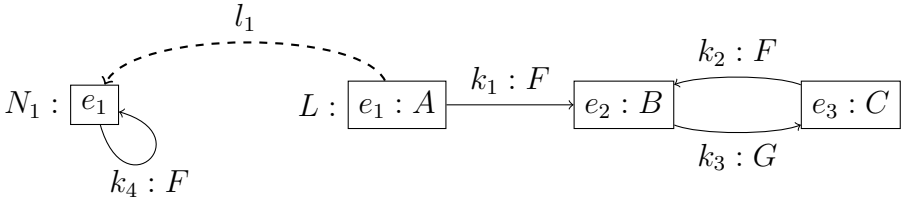


Abbildung 5.1: Mustergraph L mit einem NAC N_1

In Abbildung 5.1 ist ein Mustergraph L mit einem NAC N_1 dargestellt. Das Graphmodell sei so gewählt, dass die Typen der dort verwendeten Graphelemente jeweils die folgende Anzahl von kompatiblen (Unter-)Typen haben: A hat einen kompatiblen Typ mit `node_type_id` = 23, B hat einen mit `node_type_id` = 42,

Programm 5.2: Beispiel für `WHERE`-basierte Abfrage

```
SELECT e1.node_id, e2.node_id, e3.node_id, k1.edge_id, k2.edge_id, k3.edge_id
FROM   nodes AS e1, nodes AS e2, nodes AS e3, edges AS k1, edges AS k2, edges AS k3
WHERE  k1.src_id = e1.node_id AND k1.tgt_id = e2.node_id AND -- Inzidenzbedingungen
       k2.src_id = e3.node_id AND k2.tgt_id = e2.node_id AND
       k3.src_id = e2.node_id AND k3.tgt_id = e3.node_id AND
       e1.node_type_id = 23 AND -- Typbedingungen
       e2.node_type_id = 42 AND
       (e3.node_type_id BETWEEN 24 AND 32 OR e3.node_type_id = 66) AND
       k1.edge_type_id = 4 AND
       k2.edge_type_id = 4 AND
       k3.edge_type_id = 23 AND
       -- Homomorphiebedingungen
       e1.node_id <> e2.node_id AND e1.node_id <> e3.node_id AND
       e2.node_id <> e3.node_id AND
       k1.node_id <> k2.node_id AND k1.node_id <> k3.node_id AND
       k2.node_id <> k3.node_id AND
AND NOT EXISTS ( SELECT *
                 FROM   edge AS k4
                 WHERE  k4.src_id = k3.node_id AND k4.tgt_id = k3.node_id AND
                        k4.edge_type_id = 4 AND
               )
```

C hat mehrere kompatible Typen mit `node_type_id` = 24, ..., 32 und 66. Die beiden Kantentypen haben keine Untertypen, sie selbst seien durch eine 4 für F und 24 für G repräsentiert. Das einzige eigenständige Graphelement k_4 der negativen Anwendungsbedingungen N_1 hat ebenfalls den Typ F.

Die Abfrage für dieses Muster ist in Programm 5.2 angegeben. Es ist deutlich zu sehen, dass alle Bedingungen für einen Mustergraphen ungeordnet (da der `AND`-Operator assoziativ und kommutativ ist) in einer `WHERE`-Klausel stehen. Das NAC ist wie ein normales Muster zu finden, nur dass es anschließend auf Nichtexistenz geprüft wird. Man könnte auch eine zweite Abfrage für jedes NAC formulieren, wobei man die spezielle Vorbelegung η der NACs (siehe Definition 4.9) jeweils in die Abfrage einsetzen müsste. Allerdings ist dieses Vorgehen nicht empfehlenswert; obschon es die Konstruktion der Abfrage vereinfacht, führt es doch zu mehreren Abfragen, die je nach Anzahl der NACs zu beachtlichem Mehraufwand an Laufzeit führen können. Darum werden bei beiden hier besprochenen Ansätzen alle NACs, zusammen mit dem Muster, in einer einzigen Abfrage gestellt.

5.1.3 Abfragen mit explizitem `JOIN`

Programm 5.3 skizziert den prinzipiellen Aufbau einer Abfrage mit explizitem `JOIN`. Dieser ist (erstens) komplizierter als eine `WHERE`-basierte Abfrage und lässt (zweitens) den angeordneten Aufbau gut erkennen: In jedem JOIN wird ein neues Graphelement oder Attribut untersucht.

Die `SELECT`-Klausel sieht bei diesem Ansatz genauso aus wie beim vorherigen. Unterschiede sind allerdings schnell feststellbar: Die `FROM`-Klausel (ohne die nachfolgenden `JOIN`s) enthält nur einen Alias, nämlich den für das als Erstes zu verarbeitende Graphelement oder Attribut. Weitere Mustergraphelemente werden durch `INNER JOIN`-Klauseln als Alias hinzugefügt, und die die Mustergraphelemente be-

Programm 5.3: Prinzipieller Aufbau JOIN-basierter Abfragen

```
1  SELECT     /* Ecken- und Kanten des Musters */
2  FROM       /* 1. Ecke, Kante, Eckenattribut, Kantenattribut */
3  INNER JOIN /* 2. Ecke, Kante, Eckenattribut, Kantenattribut */
4          ON -- Inzidenz-, Typ-, Attribut-, Homomorphiebedingung
5             -- ... für 1. + 2. zu prüfendes Element des Musters
6  INNER JOIN /* 3.  Ecke, Kante, Eckenattribut, Kantenattribut */
7          ON -- Inzidenz-, Typ-, Attribut-, Homomorphiebedingung
8             -- ... für 2. + 3. zu prüfendes Element des Musters
9  ...
10 LEFT JOIN  /* 1. NAC-Ecke, -Kante, -Eckenattribut, -Kantenattribut */
11         ON -- Inzidenz-, Typ-, Attribut-, Homomorphiebedingung
12            -- ... für 1. zu prüfendes Element des 1. NAC
13 LEFT JOIN  /* 2. NAC-Ecke, -Kante, -Eckenattribut, -Kantenattribut */
14         ON -- Inzidenz-, Typ-, Attribut-, Homomorphiebedingung
15            -- ... für 2. zu prüfendes Element des 1. NAC
16 ...
17 LEFT JOIN  /* letzte NAC-Ecke, -Kante, -Eckenattribut, -Kantenattribut */
18         ON -- Inzidenz-, Typ-, Attribut-, Homomorphiebedingung
19            -- ... für letztes zu prüfendes Element des k. NAC
20 GROUP BY   /* Ecken- und Kanten des Musters */
21 HAVING     COUNT(/* letztes Element des 1. NAC */) = 0 AND
22 ...
23            COUNT(/* letztes Element des k. NAC */) = 0
```

treffenden Bedingungen in der zugehörigen ON-Klausel angegeben. Hier werden auch Attributbedingungen, die mehrere Attribute umfassen, genau dann eingefügt, wenn alle beteiligten Graphelemente schon gepasst wurden. Sobald eine Bedingung einer Stufe dazu führt, dass keine einzige Relation selektiert werden kann, bricht die INNER JOIN-Kette ab, da der NULL-Wert durch die Stufen kaskadiert, selbst wenn dort alle weiteren Bedingungen erfüllbar sind.

Die Graphelemente der NACs werden durch LEFT JOINs zur Passung gebracht. Dies geschieht, weil der LEFT JOIN die erste Tabelle immer übernimmt. Falls er in der zweiten Tabelle keine Zeile findet, die die Bedingungen erfüllt, werden die entsprechenden Spalten mit NULL aufgefüllt. Dieser NULL-Wert kaskadiert dann ebenso, aber ohne die Suche sofort abzubrechen. Die Auswertung, ob ein negatives Muster gepasst werden konnte – was zur Ablehnung der (positiven) Passung führen muss – geschieht erst am Ende durch eine Kombination aus GROUP BY und HAVING-Klauseln.

Nehmen wir erstens an, dass mehrere positive Passungen gefunden wurden, und zweitens, dass es eine Zeile gibt, in der keines der Elemente eines negativen Musters auf NULL abgebildet wurde. In dieser Situation lässt die GROUP BY-Klausel, die alle positive Graphelemente umfasst, pro positiver Passung eine Zeile entstehen – es sei denn, es können auch negative Passungen gefunden werden. Letzteres ist aber nach unseren Annahmen der Fall, also haben wir diese Situation noch festzustellen. Wir dürfen aber nicht die unvollständigen Passungen mitzählen. Darum ist ein schieres Zählen der Zeilen nicht ausreichend. Das letzte Element eines negativen Musters wird nur dann gefunden, wenn auch das ganze Muster gefunden wurde, also eine negative Passung existiert. Somit müssen wir also genau diese Elemente, mittels einer geeigneten HAVING COUNT(...) = 0-Klausel, zählen. Da NULL-Werte nicht mitgezählt werden, gibt es auch keine „Fehlalarme“, wenn das negative Muster

Programm 5.4: Beispiel für JOIN-basierte Abfrage

```
1  SELECT     e3.node_id, e2.node_id, e1.node_id, k3.edge_id, k1.edge_id, k2.edge_id
2  FROM       nodes AS e3
3  INNER JOIN edges AS k3
4          ON e3.node_id = k3.tgt_id AND k3.edge_type_id = 23 AND
5             (e3.node_type_id BETWEEN 24 AND 32 OR e3.node_type_id = 66)
6  INNER JOIN nodes AS e2
7          ON e2.node_id = k3.src_id AND e2.node_type_id = 42 AND e2.node_id <> e3.node_id
8  INNER JOIN edges AS k1
9          ON e2.node_id = k1.tgt_id AND k1.edge_type_id = 4
10 INNER JOIN nodes AS e1
11         ON e1.node_id = k1.src_id AND e1.node_type_id = 23 AND
12            e1.node_id <> e2.node_id AND e1.node_id <> e3.node_id
13 INNER JOIN edges AS k2
14         ON e2.node_id = k2.tgt_id AND e3.node_id = k2.src_id AND k2.edge_type_id = 4
15 LEFT JOIN  edges AS k4
16         ON e3.node_id = k4.src_id AND e3.node_id = k4.tgt_id AND k4.edge_type_id = 4
17 GROUP BY   e3.node_id, e2.node_id, e1.node_id, k3.edge_id, k1.edge_id, k2.edge_id
18 HAVING     COUNT(k4.edge_id) = 0
```

nicht vollständig gefunden wurde.

Wie bei den WHERE-basierten Ansätzen haben wir hier ebenfalls das Muster aus Abbildung 5.1 als SQL-Abfragen unter Verwendung von JOIN-Klauseln als Programm 5.4 umgesetzt. In dem Beispiel ist die nun aufgeprägte Ordnung beim Konstruieren der Lösung augenfällig. Die Reihenfolge lässt sich an der ersten WHERE-Klausel und den nachfolgenden JOIN-Klauseln ablesen. In dem Programm 5.4 ist die Reihenfolge der Betrachtung der Mustergraphelemente also:

$$e_3, k_3, e_2, k_1, e_1, k_2, [\text{NAC1}]\ k_4$$

wobei NAC1 andeuten soll, dass ab hier auf die Verarbeitung negativer Muster umgeschaltet wird.

5.2 Virtuelle Maschine

In diesem Abschnitt definieren wir zunächst eine virtuelle Maschine, deren Programme verschiedenen Suchstrategien entsprechen. Wir zeigen, wie man die Maschine in ihrer Effizienz noch steigern kann, indem wir Techniken des dynamischen Programmierens einsetzen. Die Programme, sprich Suchstrategien, werden dann im nächsten Abschnitt 5.3 mit Kosten versehen, sodass wir ein Optimierungsproblem formulieren können. Dieser Ansatz wurde in den Studienarbeiten von Batz [Bat05a], Szalkowski [Sza05] und Kroll [Kro07] sowei in der Diplomarbeit von Batz [Bat05b] und in weiteren Veröffentlichungen untersucht [Bat06, BKG08] und in GrGen in mehreren Varianten implementiert. Unser hier vorgestellter Ansatz – nämlich die *suchplanbasierte Mustersuche* – ist eng verwandt mit einem gleichzeitig entwickelten Ansatz von Varró et al., der unabhängig von unserer Gruppe ähnliche Prinzipien und auch Methoden der Durchführung gefunden hat (siehe auch Abschnitt 3.1.3.3).

5.2.1 Suchstrategien als Suchprogramme

Wir legen als einzige Einschränkung – die wir aus Geschwindigkeitsgründen machen – fest, dass wir den Problemraum, also Graphen und Graphmorphismen, dargestellt in einer zunächst noch offengelassenen hauptspeicherbasierten Datenstruktur, nicht verlassen möchten.

Die Frage ist nun: Welche Suchstrategien sind in diesem Szenario sinnvoll, um einen Mustergraphen L in einem Arbeitsgraphen H zu finden? Zunächst einmal: Womit starten wir unsere Suche? Auf der einen Seite können wir mit der *leeren Passung* starten und diese Schritt für Schritt – falls überhaupt möglich – zu einer *vollständigen Passung* „erweitern“, wobei wir noch offen lassen, wie dieses Erweitern im Einzelnen vonstattengeht. Andererseits kann man aus dem Kreuzprodukt der Ecken und Kanten aus L und H die Passungen „herausfiltern“, indem alle Zeilen gelöscht werden, die einem Element aus L ein (oder mehrere) unpassende Element(e) aus H zuordnen. Da das Konstruieren des Kreuzprodukts sehr speicher- und damit auch laufzeitintensiv ist, wählen wir die leere Passung als Startpunkt.

Betrachten wir hier ein GR-Muster $G = (L, c, \arg, \mathcal{N}, \hom)$ mit folgendem zugehörigen match-Funktor, das auf einem Arbeitsgraphen H mit dem selben Graphmodell zur Passung gebracht werden soll:

$$\text{match}(L, H, c, m', \mathcal{N}, \hom)$$

Von der leeren Passung ausgehend, muss zunächst eine erste Ecke oder Kante des Mustergraphen zur Passung gebracht werden. Um die Auswahl einzuschränken, können wir zum Beispiel den Ecken- oder Kantentyp heranziehen; falls es noch zusätzlich Attributbedingungen zu erfüllen gilt, reduzieren diese die Auswahl weiter. Schließlich haben wir eine Menge von partiellen Passungen Γ_1 für die gelten soll:

$$\forall m \in \text{match}(L, H, c, m', \mathcal{N}, \hom)\ \exists \gamma_1 \in \Gamma_1 : \gamma_1 \trianglelefteq m$$

Also hat jede Passung mindestens eine entsprechend erweiterbare *partielle Passung* in Γ_1. Natürlich ist die Umkehrung im Allgemeinen nicht wahr, denn Γ_1 kann partielle Passungen enthalten, die sich nicht zu Passungen erweitern lassen.

Nun stellt sich die Frage, wie die initialen partiellen Passungen $\gamma_1 \in \Gamma_1$ zu partiellen Passungen der zweiten Stufe, also $\gamma_1 \trianglelefteq \gamma_2 \in \Gamma_2$ erweitert werden. Offensichtlich ist, dass wir von dem einzigen abgebildeten Graphelement $x \in L$ mit $H \ni \mathfrak{x} = \gamma_1(x)$ ausgehen können, um dessen adjazente Ecken oder Kanten näher zu betrachten. Allerdings muss ein Folgen des Graphzusammenhangs nicht immer klug sein: Betrachten wir zum Beispiel die Situation in Abbildung 5.2, so erkennen wir, dass es vorteilhaft sein kann, das Muster nicht entlang des Graphzusammenhangs zu suchen: Der Mustergraph verlangt zwei spezielle Eckentypen S am Anfang und Ende der Kette. Allerdings ist im Arbeitsgraphen jeweils nur einer vorhanden. Die Struktur des Arbeitsgraphen würde zu einer erheblichen Auffächerung führen, wenn wir nur *ein* S nachschlagen und von dort aus zu erweitern suchen. Wir sehen also, dass wir prinzipiell zwei Wege haben, partielle Passungen γ_i schrittweise zu erweitern, wobei die Kardinalität der Mengen Γ_i für wachsende i sowohl wachsen als auch schrumpfen kann, bis schließlich

$$\Gamma_k = \text{match}(L, H, c, m', \mathcal{N}, \hom)$$

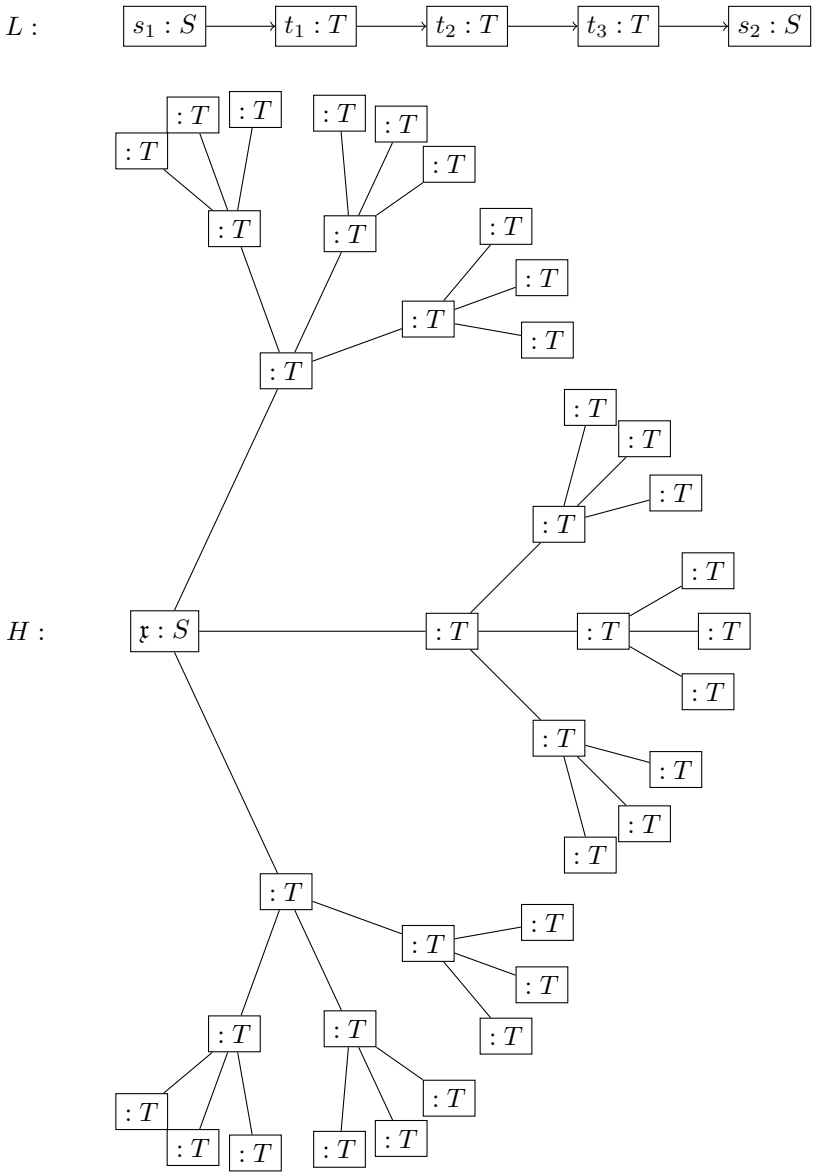


Abbildung 5.2: Ein Arbeitsgraph, bei dem Nachschlagen der beiden im Arbeitsgraphen seltenen Musterknoten s_1, s_2 vom Typ S sinnvoll ist. Die fehlenden Pfeilspitzen sollen andeuten, dass die Kanten im Arbeitsgraphen jeweils in beide Richtungen vorhanden sind.

für irgendein von der Mustergröße abhängiges k gilt. Obige Gleichung ist offenbar eine Absichtserklärung und keine Trivialität: Wir wollen die Γ_is schrittweise so konstruieren, dass sie letztendlich $\text{match}(L, H, c, m', \mathcal{N}, \text{hom})$ ergeben. Dies muss aber noch bewiesen werden, was wir mit Satz 5.5 auch tun.

Um so kleiner die Mengen Γ_i sind, desto effizienter ist die Mustersuche. Falls ein Γ_i irgendwann einmal leer wird, dann ist die Suche erfolglos vorüber; es konnte also keine Passung gefunden werden. Beim Übergang von $\gamma_i \in \Gamma_i$ nach $\gamma_i \trianglelefteq \gamma_{i+1} \in \Gamma_{i+1}$ ist jeweils zu prüfen, ob eine partielle Passung ungültig erweitert wurde, mithin ob

$$\forall m \in \text{match}(L, H, c, m', \mathcal{N}, \text{hom})\ \exists \gamma_{i+1} \in \Gamma_{i+1} : \gamma_{i+1} \trianglelefteq m$$

verletzt ist. Aus dieser Prüfung der Elemente und aus der bloßen Notwendigkeit die Elemente zu repräsentieren ergibt sich, dass die Laufzeit abhängig von der Zahl der Elemente von Γ_i ist. Es gibt im Wesentlichen zwei Einflussgrößen auf die Kardinalität von Γ_i: Zum einen die Reihenfolge[5] des Hinzunehmens der Elemente x aus L in den Definitionsbereich von γ_i und zum anderen die Auswahl der Merkmale, die wir für die Prüfung dieser x heranziehen. Welche *Merkmale* können wir also verwenden, um die Auswahl bei jedem Übergang von Γ_i nach Γ_{i+1} möglichst stark einzuschränken und damit die zu prüfenden Mengen klein zu halten? Im Prinzip bilden alle strukturbildenden Entitäten des Muster- und Arbeitsgraphen solche Merkmale:

- Ecken-/Kantentyp ($\text{typ}_X(x)$)
- Attributwert ($\text{val}_X(x, a)$)
- Eckenkardinalität ($|\text{inc}_E(e)|$)
- Richtung ($e = \text{src}(k)$ oder $e = \text{tgt}(k)$)

mit $x \in X$ und $a \in \text{attr}_X(\text{typ}_X(x))$ sowie $X = K \ni k$ oder $X = E \ni e$. Darüber hinaus können mehrere Merkmale zu lokalen Mustern kombiniert werden. Wenn wir zum Beispiel wissen, dass wir häufig eine Ecke mit Schlinge suchen, können wir alle Vorkommen von Ecken mit Schlingen im Arbeitsgraphen gesondert merken und so schnell darauf zugreifen.

Zur Repräsentation des Arbeitsgraphen im Hauptspeicher müssen wir natürlich die elementaren Merkmale wie den Eckentyp speichern. Allerdings ermöglicht uns erst die gesonderte Speicherung von Graphelementen, jeweils bezüglich gewisser Merkmale, eine schnelle Mustersuche. Dieser Ansatz benötigt jedoch einen Laufzeit-Mehraufwand beim Einfügen oder Löschen von Graphelementen sowie zusätzlichen Speicher für die gesonderte Speicherung gewisser Datenstrukturen (z. B. in einer Liste oder Streutabelle). Auf den nun auszulotenden Kompromiss zwischen Beschleunigung der Mustersuche durch gesonderte Speicherung und die dadurch gleichzeitig hervorgerufene Verlangsamung anderer Operationen wegen Laufzeit- respektive Speichermehraufwand kommen wir im Folgenden noch zurück. Als Nächstes betrachten wir die elementaren Operationen, die dem jeweiligen Finden eines Graphelements mit gewissen Merkmalen zugeordnet sind.

- *Nachschlagen* mittels Merkmalen in gesonderten Datenstrukturen mit der lkp-*Anweisung*. Dies findet einen, möglicherweise vom restlichen, schon gefundenen

[5] Die Reihenfolge des Hinzunehmens ist für die Leistungsfähigkeit der Mustersuche entscheidend aber auch komplexer, sodass wir dieser Frage Abschnitt 5.3 widmen.

Muster isolierten, *Aufpunkt* im Graphen. Falls nicht anderweitig Aufpunkte bekannt sind (zum Beispiel durch Vorbelegung), muss die Mustersuche immer mit einer lkp-Anweisung beginnen.

- *Erweitern* mittels Graphzusammenhang unter Berücksichtigung von Merkmalen wie zum Beispiel Kantenrichtung und Typen mit der ext-*Anweisung*. Auf diese Weise werden nur zusammenhängende Musterteile gefunden. Das Erweitern eines Musters erfordert – in der einfachsten Variante – keine zusätzlichen Datenstrukturen, sondern operiert ausschließlich auf der primären Graphstruktur.

Die Anweisungen lkp und ext können für die Verwendung zahlreicher Merkmale spezialisiert werden. Überdies ist eine spezielle Variante der ext-Anweisung, nämlich die chk-Anweisung, möglich, die nicht erweiternden, sondern prüfenden Charakter hat. Ein Programmsystem kann ebendiese Anweisungsfolgen schrittweise ausführen. Ein solches System kann als eine *virtuelle Maschine* gesehen werden, wobei jene Anweisungsfolgen seine Programme sind. Im nächsten Abschnitt betrachten wir die virtuelle Maschine und ihre Anweisungen im Detail. Mit diesem Rüstzeug ist es möglich, sehr viele Suchstrategien als Sequenz elementarer Suchbefehlen, einem sogenannten *Suchprogramm*, darzustellen.

5.2.2 Befehlssatz der virtuellen Maschine

Wir geben zunächst einen vollständigen Befehlssatz einer virtuellen Maschine für die Graphmustersuche an. Dieser ist minimal in dem Sinn, dass durch Weglassen oder Einschränken eines Befehls nicht mehr jeder Mustergraph gefunden werden kann. Allerdings führt dieser minimale vollständige Befehlssatz nicht zu hoher Geschwindigkeit der Mustersuche. Darum betrachten wir im Anschluss daran effizientere Befehlsvarianten beziehungsweise virtuelle Maschinen, deren Semantik wir immer wieder darauf zurückführen werden.

5.2.2.1 Ein vollständiger Befehlssatz

Ein minimaler und vollständiger Satz von Suchbefehlen besteht aus dem Nachschlagen von Graphelementen, dem Prüfen des Graphzusammenhangs und weiterer Eigenschaften des GR-Musters wie Homomorphie- und Attributbedingungen. Nachfolgend sind die Syntax sowie die Semantik der einzelnen Befehle angegeben.

Definition 5.1 (Minimaler und vollständiger Satz von Suchbefehlen) Sei H ein Arbeitsgraph und $G = (L, c, \arg, \mathcal{N}, \hom)$ ein GR-Muster mit $H, L \in \mathcal{I}_M$, dann definieren wir den folgenden Satz von *elementaren Suchbefehlen* mittels

$$\mathrm{vm}[\![\cdot]\!] : \; H \times \wp(\mathbb{P}(L,H)) \longrightarrow \wp(\mathbb{P}(L,H))$$

einer denotationellen Auswertungsfunktion:

Nachschlagen mittels Eckentyp

Syntax	$\mathrm{lkp}(a : A), \qquad a \in E_L,\ A \in T_E$
Semantik	$\mathrm{vm}[\![\mathrm{lkp}(a : A)]\!]_H \Gamma =$
	$\left\{\gamma \cup \{a \mapsto \mathfrak{x}\} \,\middle|\, \forall \gamma \in \Gamma\ \forall \mathfrak{x} \in E_H : \mathrm{typ}_{E_H}(\mathfrak{x}) \leq A\right\}$
Beispiel	Abbildung 5.3

Nachschlagen mittels Kantentyp

Syntax	$\mathrm{lkp}(b : B), \quad b \in K_L,\ B \in T_K$
Semantik	$\mathrm{vm}[\![\mathrm{lkp}(b : B)]\!]_H\Gamma =$ $\{\gamma \cup \{b \mapsto \mathfrak{x}\} \mid \forall \gamma \in \Gamma\ \forall \mathfrak{x} \in K_H : \mathrm{typ}_{K_H}(\mathfrak{x}) \leq B\}$
Beispiel	Abbildung 5.4

Prüfen einer ausgehenden Kante

Syntax	$\mathrm{chk}(a \xrightarrow{b}), \quad a \in E_L,\ b \in K_L$
Semantik	$\mathrm{vm}[\![\mathrm{chk}(a \xrightarrow{b})]\!]_H\Gamma = \{\gamma \in \Gamma \mid \mathrm{src}(\gamma(b)) = \gamma(a)\}$
Beispiel	Abbildung 5.5

Prüfen einer eingehenden Kante

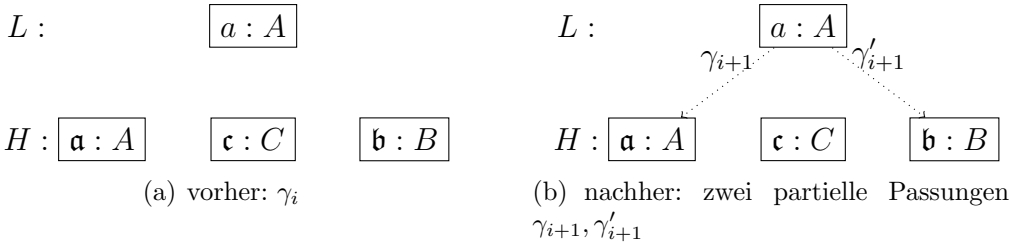
Syntax	$\mathrm{chk}(a \xleftarrow{b}), \quad a \in E_L,\ b \in K_L$
Semantik	$\mathrm{vm}[\![\mathrm{chk}(a \xleftarrow{b})]\!]_H\Gamma = \{\gamma \in \Gamma \mid \mathrm{tgt}(\gamma(b)) = \gamma(a)\}$
Beispiel	Analog zu Abbildung 5.5

Attributbedingungen

Syntax	$\mathrm{chk}(c), \quad c$ eine Attributbedingung
Semantik	$\mathrm{vm}[\![\mathrm{chk}(c)]\!]_H\Gamma = \{\gamma \in \Gamma \mid \mathrm{eval}[\![c]\!]_\gamma = \top\}$
Beispiel	$\mathrm{chk}(\texttt{a.x == 42 * b.y}), \quad a \in E_L,\ b \in K_L$

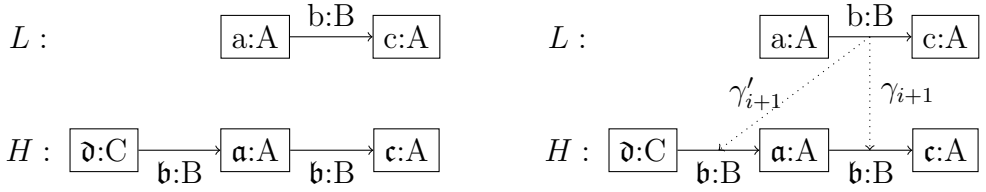
Homomorphiebedingungen

Syntax	$\mathrm{chk}(\mathrm{chk}(\mathrm{hom})),$ hom eine Homomorphiebedingung
Semantik	$\mathrm{vm}[\![\mathrm{chk}(\mathrm{hom})]\!]_H\Gamma = \{\gamma \in \Gamma \mid \mathrm{hom}(\gamma) = \top\}$



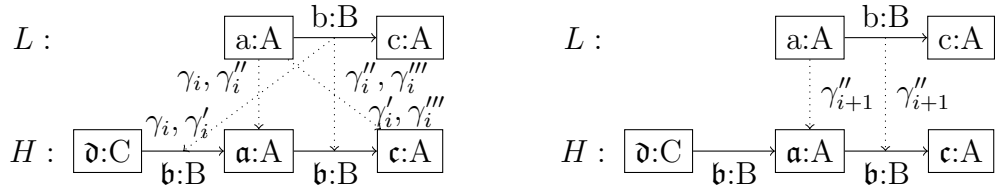
(a) vorher: γ_i (b) nachher: zwei partielle Passungen $\gamma_{i+1}, \gamma'_{i+1}$

Abbildung 5.3: Beispiel zu $\mathrm{lkp}(a : A)$. Sei B Untertyp von A; C inkompatibel zu A.



(a) vorher: γ_i (b) nachher: γ_{i+1}

Abbildung 5.4: Beispiel zu $\mathrm{lkp}(b : B)$



(a) vorher: vier partielle Passung $\Gamma_i = \{\gamma_i, \gamma_i', \gamma_i'', \gamma_i'''\}$

(b) nachher: nur γ_{i+1}'' bleibt bestehen

Abbildung 5.5: Beispiel zu chk($a \xrightarrow{b}$)

Obige Semantikdefinition ist noch lückenhaft: chk-Befehle benötigen, dass $\gamma \in \Gamma$ schon gewisse Graphelemente abbildet. Wir legen nun fest, falls dies nicht der Fall ist, dann soll $\mathrm{vm}[\![\mathrm{chk}(\cdot)]\!]_H\Gamma$ das *Fehlerelement* $\perp$ als Ergebnis haben. Weiterhin legen wir fest, dass $\perp$ für alle lkp und chk-Befehle ein Fixpunktoperand ist, also gelte: $\mathrm{vm}[\![\mathrm{chk}(\cdot)]\!]_H\perp = \perp$ und $\mathrm{vm}[\![\mathrm{lkp}(\cdot)]\!]_H\perp = \perp$. Das explizite Prüfen der Homomorphiebedingung mittels eines entsprechenden chk-Befehls ist nötig, da zwei Graphelemente a_1, a_2 des Mustergraphen mittels zweier lkp-Befehle $\mathrm{lkp}(a_1 : A), \mathrm{lkp}(a_2 : A)$ zunächst nicht notwendigerweise injektiv durch ein entsprechendes γ abgebildet werden. Da aber Mustergraphelemente immer injektiv auf den Arbeitsgraphen abgebildet werden sollen (siehe Definition 4.11) – es sei denn für sie gilt $a_1 \sim a_2$ – ist es in der Regel für typkompatible Mustergraphelemente nötig ihre injektive Abbildung sicherzustellen.

Definition 5.2 (Suchprogramm) Ein *Suchprogramm* S besteht aus einer Folge aus Suchbefehlen $(s_1, \ldots, s_q)$. Die Semantik eines Suchprogramms ist *sequenziell* und *kompositional* bestimmt durch die Semantik seiner Suchbefehle.

Konkrete Suchprogramme werden offenbar immer im Hinblick auf einen Mustergraphen aufgeschrieben, da sie aus Elementen konkreter Mustergraphen aufgebaut werden. Allerdings besteht zunächst kein Zusammenhang zwischen Suchprogramm und Mustergraph: Man könnte aus den Mustergraphelementen Programme konstruieren, die vollkommen andere Muster suchen, da nur die Elemente nicht aber der Graphzusammenhang verwendet werden. Prinzipiell können diese Suchprogramme auf alle Arbeitsgraphen angewendet werden, die sich mit dem Mustergraphen das Graphmodell teilen; falls die Graphmodelle inkompatibel sind, wird per definitionem nie eine Passung gefunden. Falls klar ist, welcher Arbeitsgraph gemeint ist, können wir statt $\mathrm{vm}[\![\cdot]\!]_H\Gamma$ auch kurz $\mathrm{vm}[\![\cdot]\!]\Gamma$ schreiben. Der Satz an elementaren Suchbefehlen aus Definition 5.1 ist offenbar minimal in dem Sinn, dass man keinen der Befehle weglassen kann; dass er auch vollständig ist, werden wir in Satz 5.5 sehen.

Zum Abarbeiten der Suchprogramme definieren wir eine virtuelle Maschine (VM) für die Graphmustersuche. Die VM berechnet ihr Ergebnis, wegen der sequenziellen Kompositionalität der Suchprogramme, durch schrittweise Auswertung von Suchbe-

fehlen, also

$$\begin{aligned}
[\![S]\!]\Gamma &= [\![(s_1, \ldots, s_q)]\!]\Gamma \\
&= [\![(s_2, \ldots, s_q)]\!][\![s_1]\!]\Gamma \\
&= [\![(s_2, \ldots, s_q)]\!]\Gamma_1 \\
&\vdots \\
&= [\![(s_q)]\!]\Gamma_{q-1} \\
&= \Gamma_q
\end{aligned}$$

Definition 5.3 (Virtuelle Maschine zur Graphmustersuche) Sei H ein Graph mit $H \in \mathcal{I}_M$, $S = (s_1, \ldots, s_q)$ ein Suchprogramm und $\Gamma \subset \mathbb{P}(\mathcal{I}_M, H)$ gegeben, dann ist mit obiger Formel $\Gamma_q = \mathrm{vm}[\![S]\!]_H\Gamma$ das Ergebnis der *virtuellen Maschine* zur Graphmustersuche.

Man beachte, dass hier kein Mustergraph vorausgesetzt ist. In der Tat kann die virtuelle Maschine partielle Passungen mit zunächst beliebigem Urbild erweitern. Die Frage, ob das Ergebnis als sinnvoll bezeichnet werden kann, wird erst im folgenden Abschnitt beantwortet.

5.2.2.2 Suchprogramme für GR-Muster

Nun ist noch zu klären, wie wir zu Suchprogrammen für einen gegebenen Mustergraphen kommen. Als Erstes definieren wir, was es bedeuten soll, ein Suchprogramm (Definition 5.2) für einen Mustergraphen zu sein. Dies geschieht durch die Betrachtung aller Ergebnisse der virtuellen Maschine bei Eingabe aller möglichen Arbeitsgraphen des Graphmodells von L. Diese mit unendlichen Mengen arbeitende Definition ist praktisch unhandlich und wird im Beweis von Satz 5.5 durch einen Algorithmus ersetzt, der die gewünschten Eigenschaften ohne Kenntnis von Arbeitsgraphen berechnet. Ferner klären wir, wann die Berechnung mit dem Fehlerelement endet.

Definition 5.4 (Suchprogramm für ein GR-Muster) Sei M ein Graphmodell und $G = (L, c, \arg, \mathcal{N}, \hom)$ ein GR-Muster mit $L \in \mathcal{I}_M$ und zugehörigem match-Funktor gegeben, so heißt S *Suchprogramm* für G, genau dann wenn

$$\forall H \in \mathcal{I}_M : \ \mathrm{vm}[\![S]\!]_H\{\uparrow\uparrow\} = \mathrm{match}(L, H, c, m', \mathcal{N}, \hom)$$

ist.

Für eine Suchprogramm S eines Musters G gilt somit für alle Arbeitsgraphen H, dass

$$[\![S]\!]_H\{\uparrow\uparrow\} = \mathrm{match}(L, H, c, \uparrow\uparrow, \mathcal{N}, \hom)$$

und ferner sogar

$$[\![S]\!]_H\{m'\} = \mathrm{match}(L, H, c, m', \mathcal{N}, \hom)$$

ist. Wenn es uns gelingt, die Suchprogramme für ein GR-Musters algorithmisch zu bestimmen, dann haben wir eine algorithmische Grundlage zur Berechnung des match-Funktors mit Attributbedingung c, Vorbelegung m' und Homomorphiebedingung hom. Da wir die negativen Anwendungsbedingungen $\mathcal{N}$, wie wir in Abschnitt 4.2.5 gesehen haben, komplett auf Mustersuche mit Vorbelegung zurückführen können, sind damit auch NACs bestimmbar.

Satz 5.5 (Statistische Eigenschaften von Suchprogrammen) Für beliebige Suchprogramme $S = (s_1, \ldots, s_q)$ ist statisch (also ohne Kenntnis des Arbeitsgraphen) entscheidbar ob sie ein Suchprogramm für ein gewisses GR-Muster $G = (L, c, \arg, \mathcal{N}, \hom)$ sind.

Beweis: Falls die Homomorphiebedingung in der Prüfung $\mathrm{chk}(\hom) \in S$ nicht identisch mit der Homomorphiebedingung $\hom_G$ des Musters G ist, dann kann S kein Suchprogramm für G sein. Ebenso ist das Suchprogramm S keines für das Muster G, falls die Attributbedingungen $\mathrm{chk}(c)$ der entsprechenden Prüfungen konjunktiv verknüpft nicht äquivalent zu c_G des Musters G sind.

Wir wählen einem beliebigen Arbeitsgraphen H mit $L, H \in \mathcal{I}_M$. Sei m' die zu G bezüglich gewisser Argumente aus dem Arbeitsgraphen H gehörende Vorbelegung, dann ist nach Voraussetzung $m' \in \mathbb{P}(L, H)$. Die Abarbeitung der VM beginnt somit bei $[\![S]\!]_H\{m'\}$.

Dann kann mittels Induktion schrittweise gezeigt werden, dass jeder chk, lkp oder ext-Befehl den Mustergraphen stückweise beschreibt. Jeweils, wenn es Befehle gäbe, die nicht einem Stück des Mustergraphen entsprechen, endet die Abarbeitung mit dem Fehlerelement. ∎

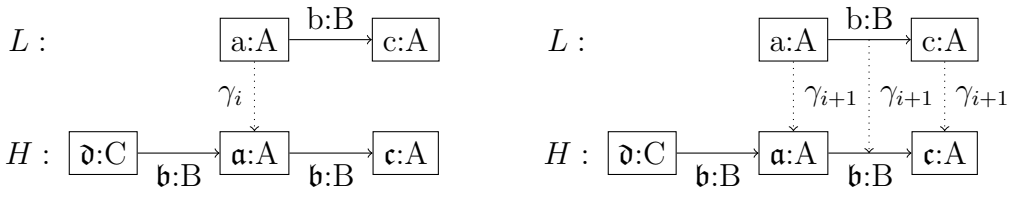
Aus dem Beweis von Satz 5.5 folgt direkt, dass für beliebige Suchprogramme statisch entscheidbar ist, ob sie ein Suchprogramm für einen Mustergraphen sind und ob ihre Berechnung mit dem Fehlerelement $\perp$ endet.

5.2.2.3 Befehlssatzvarianten und optimierende VMs

Obschon die im Vorausgehenden definierten Suchprogramme für Graphmuster die Suche eines jeden Musters algorithmisch beschreiben können, sind sie aus komplexitätstheoretischer Sicht äußerst unzulänglich. Betrachten wir zum Beispiel das Muster in Abbildung 5.7(a), so sind Kombinationen von elementaren Suchbefehlen sehr viel effizienter und eleganter. Nachstehend ist ein solcher komplexer Suchbefehl angegeben.

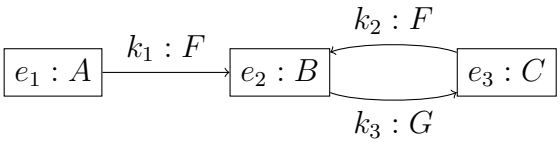
Erweitern mittels Kanten- und Eckentyp

Syntax	$\mathrm{ext}(c \xleftarrow{b:B} a : A), \qquad a, c \in E_L,\ b \in K_L,\ A \in T_E,\ B \in T_K$
Semantik	$[\![\mathrm{ext}(c \xleftarrow{b:B} a : A)]\!]\Gamma_i =$
	$[\![(\mathrm{lkp}(b : B), \mathrm{chk}(c \xleftarrow{b}), \mathrm{lkp}(a : A), \mathrm{chk}(a \xrightarrow{b}))]\!]\Gamma_i$

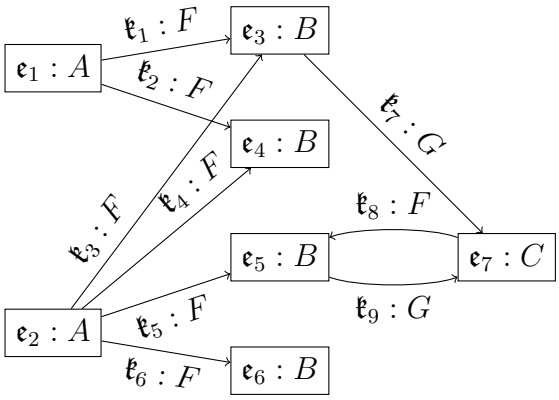


(a) vorher: γ_i (b) nachher: γ_{i+1}

Abbildung 5.6: Beispiel zu $(b, a) := \text{ext}(c \overset{B}{\leftarrow} A)$



(a) Mustergraph L



(b) Arbeitsgraph H

Abbildung 5.7: Beispiel eines Muster- und Arbeitsgraphen

5.2.3 Beispiel

Betrachten wir nun folgendes Muster in GrGen-Schreibweise:

```
pattern{
  e1:A -k1:F-> e2:B <-k2:F- e3:C;
  e2 -k3:G-> e3;
}
```

mit folgendem Graphmodell:

```
node class A;
node class B;
edge class F;
edge class G;
```

In Abbildung 5.7(a) ist eine Visualisierung des obigen Musters zu sehen. Wir geben im Folgenden vier Suchprogramme für diesen Mustergraphen an, wobei das eine aus elementaren Suchbefehlen besteht, hingegen die anderen drei optimierte

Suchbefehle nutzen.

$$
\begin{aligned}
S_1 &= \big(\ \mathrm{lkp}(e_1 : A), \mathrm{lkp}(k_1 : F), \mathrm{chk}(e_1 \xrightarrow{k_1}), \mathrm{lkp}(e_2 : B), \mathrm{chk}(e_2 \xleftarrow{k_1}), \mathrm{lkp}(k_2 : F), \\
&\qquad \mathrm{chk}(e_2 \xleftarrow{k_2}), \mathrm{lkp}(k_3 : G), \mathrm{chk}(e_2 \xrightarrow{k_3}), \mathrm{lkp}(e_3 : C), \mathrm{chk}(e_3 \xleftarrow{k_3}), \mathrm{chk}(e_3 \xrightarrow{k_2}) \ \big) \\
S_2 &= \big(\ \mathrm{lkp}(e_1 : A), \mathrm{ext}(e_1 \xrightarrow{k_1:F}), \mathrm{ext}(e_2 : B \xleftarrow{k_1}), \mathrm{ext}(e_2 \xrightarrow{k_3:G}), \mathrm{ext}(e_3 : C \xleftarrow{k_3}), \\
&\qquad \mathrm{ext}(e_2 \xleftarrow{k_2:F}), \mathrm{chk}(e_3 \xrightarrow{k_2}) \ \big) \\
S_3 &= \big(\ \mathrm{lkp}(e_3 : C), \mathrm{ext}(e_3 \xrightarrow{k_2:F}), \mathrm{ext}(e_3 \xleftarrow{k_3:G}), \mathrm{ext}(e_2 : B \xrightarrow{k_3}), \mathrm{chk}(e_2 \xleftarrow{k_2}), \\
&\qquad \mathrm{ext}(e_2 \xleftarrow{k_1:F}), \mathrm{ext}(e_1 : A \xrightarrow{k_1}) \ \big) \\
S_4 &= \big(\ \mathrm{lkp}(e_1 : A), \mathrm{ext}(e_1 \xrightarrow{k_1:F} e_2 : B), \mathrm{ext}(e_2 \xrightarrow{k_3:G} e_3 : C), \mathrm{ext}(e_2 \xleftarrow{k_2:F} e_3) \ \big)
\end{aligned}
$$

Nur das Suchprogramm S_3 ist optimal, alle anderen angegebenen Programme führen an der einen oder anderen Stelle zu Auffächerungen.

5.2.4 Suchraumfortschaltung

Die Idee der von uns entwickelten *Suchraumfortschaltung* beruht aus zwei Beobachtungen: Erstens bauen Graphersetzungssequenzen oft Graphen im Fließbandverfahren um, d. h. es werden mehrere Regeln iteriert angewandt, wobei innerhalb der iterierten Sequenz eine Regel gewisse Graphelemente manipuliert, welche die Anwendung der darauf folgenden Regel(n) erst möglich macht. Daraus kann man schließen, dass es nützlich wäre, zuerst nach einer Passung in dem Teil des Arbeitsgraphen zu suchen, der gerade eben geändert wurde. Zweitens ist es selten sinnvoll den schon durchsuchten Teil des Graphen nochmals zu durchsuchen, wenn an einer anderen Stelle eine Änderung stattgefunden hat, denn höchstwahrscheinlich ist durch diese Änderung keine neue Passung in dem schon durchsuchten und unverändert gebliebenen Bereich entstanden, wobei letzteres natürlich trotzdem nicht auszuschließen ist – insbesondere bei nicht zusammenhängenden Mustergraphen.

Wir erreichen ein Verhalten unseres Graphersetzungssystem, das obige Beobachtungen ausnutzt, indem wir die Ecken- und Kantentypen in speziellen Listen anordnen. Diese Listen sind doppelt verkettet und zyklisch[6]. Sie unterstützen das Umsetzen des Kopfes an eine gegebene Position in konstanter Zeit. Zunächst fügen wir die Graphelemente in solche Listen ein, wobei jeder Typ seine eigene Liste hat. Wenn wir eine Ersetzung durchführen, dann setzten wir die Köpfe der beteiligten Listen neu. Neue Elemente werden immer nach dem Kopf eingefügt.

Dadurch beginnt eine neue Suche immer mit den gerade eingefügten Elementen, die Elemente die schon durchsucht wurden, kommen als letztes an die Reihe. Somit ist die Suchraumfortschaltung eine Heuristik, die exakt ist, in dem Sinn, dass sie immer alle Muster liefert – so als wäre sie nicht vorhanden. Allerdings ist es, was das Laufzeitverhalten angeht, eben nur eine Heuristik: Obschon es Muster und Arbeitsgraphen gibt, bei denen, ohne die Fortschaltung der Listenköpfe, das Ergebnis schneller zu finden wäre, ist uns ein solcher Fall in der Praxis noch nicht aufgefallen.

Für einen Arbeitsgraphen H und einen Mustergraphen L ist die Beschleunigung, die im Idealfall (die beiden Annahmen vom Anfang des Abschnitts sind erfüllt) erreicht werden kann, proportional zu $|H|^{|L|}$. Sprich die Geschwindigkeitssteigerung

[6] Es handelt sich um sogenannte Ringlisten.

aufgrund von Suchraumfortschaltung kann bei variablem Mustergraphen exponentiell sein. Dies rührt daher, dass für jedes Mustergraphelement im schlechtesten Fall der ganze Arbeitsgraph durchsucht werden muss, und beim Erweitern der Passung, dies für jede der so entstandenen neuen partiellen Passungen (dies können bis zu $|H|$ Stück sein), der Vorgang wiederholt werden muss. Daraus folgt die exponentielle Auffächerung zur Basis $|H|$ und der Mustergröße als Exponent. Wir können diese außerordentlichen Eigenschaften auch tatsächlich in Benchmarks beobachten, so ist beim STSmany-Benchmark von Varró die Laufzeit für die Suche einer Passung innerhalb der Graphersetzungssequenz *konstant* und somit unabhängig von der Größe des Arbeitsgraphen (siehe Abschnitt 9.1).

5.3 Suchplanbasierte Mustersuche

Anders als beim Ansatz die Mustersuche mit relationaler Algebra zu erledigen, versucht die *suchplanbasierte Mustersuche* so viel Wissen wie möglich über den Graphen in die Optimierung der Suche zu stecken. Dabei bauen wir auf grundlegenden Überlegungen von Dörr auf (siehe Abschnitt 3.1.3.2). Unser Verfahren orientiert sich an den klassischen Aufgaben des Backends im Übersetzerbau: Es müssen Suchbefehle ausgewählt, angeordnet und ausgegeben werden. Für die Befehlsauswahl definieren wir ein Kostenmodell, das wir für die Befehlsanordnung wiederum verfeinern.

5.3.1 V-Strukturen

Die exponentielle Auffächerung an Möglichkeiten (im Bezug auf die Mustergröße) ergibt sich aus zwei Gründen:

1. Mehrere lkp-Befehle – ohne weitere Prüfungen – führen immer zu der Bildung des kartesischen Produkts der nachgeschlagenen Arbeitsgraphelemente. Diese Ausweitung an Möglichkeiten kann jedoch mit rechtzeitiger Einstreuung von chk-Befehlen entgegengewirkt werden. Folgt ein Suchprogramm dem Graphzusammenhang des Mustergraphen L und enthält der Arbeitsgraph H keine „Verzweigungen“, dann ist die Anzahl der zu überprüfenden Arbeitsgraphelemente, bei n-maligem Nachschlagen – durch den Graphzusammenhang des Musters nachbildende chk-Befehle – auf $n \times |H|$ zu begrenzen.

2. Die bösartigere Quelle der Ausweitung an Möglichkeiten liegt an diesen „Verzweigungen“ im Arbeitsgraphen. Sie können verhindern, dass chk-Befehle zum Kollabieren der Γ_i führen. Aus gleichem Grund wird bei ext-Befehlen in einem solchen Fall die Anzahl der Elemente in Γ_i im schlechtesten Fall mit dem Grad der „Verzweigungen“ multipliziert.

Wann sind also Ecken mit hohem Aus- oder Eingangsgrad besonders bösartig? Dies ist genau dann der Fall, wenn sich die Ausweitung nicht durch lokale Bedingungen wieder eindämmen lässt. Also wenn selbst nach einem Suchbefehl, der eine Kante, deren Quelle und Ziel, die jeweiligen Typen sowie die Richtung der Kante berücksichtigt (also z. B. $\text{ext}(e \xrightarrow{k:\tau_k} t : \tau)$, eine multiplikative Ausweitung des Suchraums erfolgt. Die Ausweitung ist natürlich von den Typen der drei beteiligten Graphelemente abhängig, da diese bestimmen, wie viele der vorhandenen Kanten tatsächlich passen.

Wir nennen eine solche bösartige Stelle im Arbeitsgraphen im Einklang mit Dörr [Dör95] eine V-Struktur. Für eine Einordnung des Begriffs siehe auch Abschnitt 3.1.3.2. Der Begriff V-Struktur leitet sich aus der grafischen Veranschaulichung der Situation her, die an ein „V“ erinnert.

Definition 5.6 (V-Struktur) Sei $(\tau_e, \tau_k, \tau) \in T_E \times T_K \times T_E$ gegeben. Genau dann, wenn $s \in E : \tau_e \leq \text{typ}_E(s)$ und

$$\exists\{k_1, \ldots, k_n\} \subseteq K \Big[\forall k \in \{k_1, \ldots, k_n\} : \Big[\text{src}(k) = s \wedge \tau_k \leq \text{typ}(k) \wedge \tau \leq \text{tgt}(k)\Big] \wedge \underbrace{\Big|\{k_1, \ldots, k_n\}\Big|}_{\theta_s} > 1\Big]$$

gilt, dann heißt diese Konfiguration (bestehend aus $s, \{k_1, \ldots, k_n\}$ und $\{\text{tgt}(k_1), \ldots, \text{tgt}(k_n)\}$) *konkrete V-Struktur* mit der Mannigfaltigkeit θ_s. Für die *(abstrakte) V-Struktur* (τ_e, τ_k, τ) ist die Mannigfaltigkeit θ gegeben durch:

$$\theta := \sum_{s \in E} \theta_s$$

Eine V-Struktur ist also offenbar immer nur bezüglich eines Musters mit spezifischen Typen gefährlich, es sei denn $(\bot_E, \bot_K, \bot_E)$ ist auch eine V-Struktur. Normalerweise ist aus den Elementen des Tripel jeweils klar, ob wir eine abstrakte oder konkrete V-Struktur meinen, sodass die genauere Bezeichnung weggelassen werden kann. Um deutlicher zu machen, bezüglich welcher V-Struktur die Mannigfaltigkeit besteht, schreiben wir auch: $\theta_{(\tau_e, \tau_k, \tau)}$. Dörr zeigte: Wenn sich die V-Strukturen bei der Mustersuche umgehen lassen, dann kann das Muster mit Aufwand, der linear zu der Mustergröße ist, gefunden werden. Allerdings gibt er *kein* Verfahren an, um Muster zu suchen, wenn sich die V-Strukturen nicht umgehen lassen.

Wir erweitern im Folgenden das dörrsche 0-1-Verfahren um ein Kostenmodell sowie ein darauf aufbauendes heuristisches Verfahren, das Suchprogramme unter dem Gesichtspunkt der Vermeidung von V-Strukturen konstruiert.

5.3.2 Kostenmodell

Jedem Suchbefehl ordnen wir Kosten zu, die die zu erwartendende[7] Auffächerung widerspiegeln sollen.

Definition 5.7 (Kosten für Suchbefehle) Die Funktion c, die jedem elementaren Suchbefehl ein Element der reellen Zahlen zuordnet, heißt *Kostenfunktion für*

[7] Der Begriff soll hier nicht im mathematischen Sinn des Erwartungswerts verstanden werde, sondern vielmehr als Intension unserer Heuristik.

Suchbefehle.

$$
\begin{aligned}
\mathrm{lkp}(e:\tau) &\mapsto \left|x \in E | \tau \leq \mathrm{typ}_E(x)\right| && \text{mit } \tau \in T_E \\
\mathrm{lkp}(k:\tau) &\mapsto \left|x \in K | \tau \leq \mathrm{typ}_K(x)\right| && \text{mit } \tau \in T_K \\
\mathrm{ext}(e \xrightarrow{k:\tau_k} t:\tau) &\mapsto \begin{cases} 0 & \text{falls } c(\mathrm{lkp}(e:\mathrm{typ}_E(e))) = 0 \\ \dfrac{\theta_{(\mathrm{typ}_E(e),\tau_k,\tau)}}{c(\mathrm{lkp}(e:\mathrm{typ}_E(e)))} & \text{sonst} \end{cases} \\
\mathrm{ext}(e \xrightarrow{k:\tau_k}) &\mapsto \begin{cases} 0 & \text{falls } c(\mathrm{lkp}(e:\mathrm{typ}_E(e))) = 0 \\ \dfrac{\theta_{(\mathrm{typ}_E(e),\tau_k,\bot_E)}}{c(\mathrm{lkp}(e:\mathrm{typ}_E(e)))} & \text{sonst} \end{cases} \\
&\ldots
\end{aligned}
$$

Es sind natürlich auch andere Kostenmodelle möglich. Insbesondere kann man das geometrische Mittel anstatt des arithmetischen Mittels für die Mannigfaltigkeit nutzen. Es ist auch möglich die Normierung wegzulassen, oder die Kosten zu logarithmieren, um die Zahlenwerte kleiner zu halten. Wir können im Rahmen dieser Kostenmodelle und der weiteren Verfahrenschritte auch den dörrschen Ansatz als Spezialfall darstellen (siehe dazu Abschnitt 5.3.4). Wir haben uns, nachdem wir auch andere Formulierungen praktisch getestet haben, für die Setzung der Kosten nach Definition 5.7 entschieden.

Die Kosten für ein ganzes Suchprogramm ergeben sich nun aus der vermuteten Auffächerung, die wiederum auf den Kosten für einen elementaren Suchbefehl fußt. Da eine frühe Auffächerung – unter der Annahme, dass die Suche nicht unmittelbar danach abbricht – besonders kostspielig ist, wählen wir die Kostenfunktion in Definition 5.8, sodass die Kosten ihrer Position im Programm entsprechend häufig multiplikativ auftreten.

Definition 5.8 (Kosten für ein Suchprogramm) Sei $S = (s_1, \ldots, s_q)$ ein Suchprogramm.

$$c(S) \mapsto c(s_1) + c(s_1)c(s_2) + \ \ldots\ + c(s_1)c(s_2)\cdots c(s_q)$$

Definition 5.8 benutzt nur die Kosten der Suchbefehle und die Position der Befehle im Programm um die Kosten des Programms zu bestimmen. Man kann allerdings versuchen noch mehr Informationen über den Arbeitsgraphen einfließen zu lassen. Je nach Ausgestaltung der VM, kann es sein, dass die Kosten eine Kante mit gewissen Typ an einer bekannten Ecke zu finden proportional zu den dort vorhandenen Kanten sind. In diesem Fall könnte es günstig sein, sofern man die Wahl hat, welcher ext-Befehl als nächstes auszuführen ist, Eckentypen zuerst zu wählen, die eher geringen Grad haben. Dieser Idee entsprechend kann man folgende Definition für die Kosten eines Suchprogramms treffen:

$$c(S) \mapsto c(s_1)d(s_1) + c(s_1)c(s_2)d(s_2) + \ \ldots\ + c(s_1)c(s_2)\cdots c(s_q)d(s_q)$$

wobei $d(s_i)$ wie folgt gegeben ist:

$$
\begin{aligned}
d(\mathrm{ext}(e \xrightarrow{k:\tau_k} t:\tau)) &\mapsto c(e \xrightarrow{k:\bot} t:\tau)) \\
d(\mathrm{lkp}(\ldots)) &\mapsto 1
\end{aligned}
$$

Auch in diesem Fall haben wir Experimente mit den beiden Kostenmodellen durchgeführt. Das erweiterte Kostenmodell, das auch die Anzahl der inzidenten Kanten berücksichtigt, liefert zwar feinere Kostenstufen allerdings wird die Korrelation nicht unbedingt besser (siehe Abbildungen 9.4 und 9.5 aus Abschnitt 9.3).

Obwohl die Analyse eines Arbeitsgraphen nur lineare Zeitkomplexität hat (ein Beweis findet sich in [Bat05b]), ist es dennoch nicht immer wünschenswert den Arbeitsgraphen analysieren zu müssen. Dies ist insbesondere zutreffend, wenn die Suchprogramme unabhängig vom Arbeitsgraphen, also a priori generiert werden sollen, um den mit der Generierung verbunden Laufzeitwasserkopf zu vermeiden. Dazu leiten wir die Kosten nicht mehr wie bisher aus der Analyse eines konkreten Arbeitsgraphen ab, sondern bestimmen sie aus Domänenwissen.

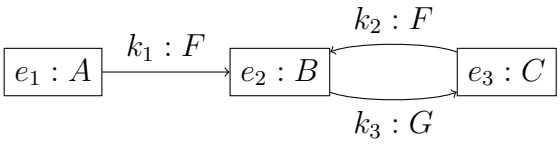
Es ist möglich, wenn man eine Anwendung für eine gewisse Domäne, zum Beispiel die Mustersuche auf Zwischendarstellungen im Übersetzerbau, beschleunigen will, die dort auftretenden Arbeitsgraphen statistisch auf V-Strukturen zu analysieren und so zu generellen Kosten zu kommen. Dies setzt natürlich zweierlei voraus, das im geringeren Maß auch für individuell bestimmte Kosten zutrifft: Erstens müssen Ecken und Kanten möglichst „aussagekräftige“ Typen haben. Aussagekräftig bedeutet hier, dass Ecken mit gleichen Typen oft in ähnlichen Kontexten auftreten. Insbesondere wäre ein zufällig erzeugter Graph mit zufällig zugewiesenen Typen nicht „aussagekräftig“ typisiert. Wir brauchen diese Struktur in Graphen, um mit der statistischen Analyse von allgemeinen Verhältnissen auf spezielle Vorkommnisse schließen zu können. Je näher ein konkreter Arbeitsgraph diesem Idealzustand kommt, desto genauer wird unsere Heuristik. Aus gleichem Grund sind Graphen ohne Typen eher ungeeignet für suchplanbasierte Verfahren.

5.3.3 Suchplangraph

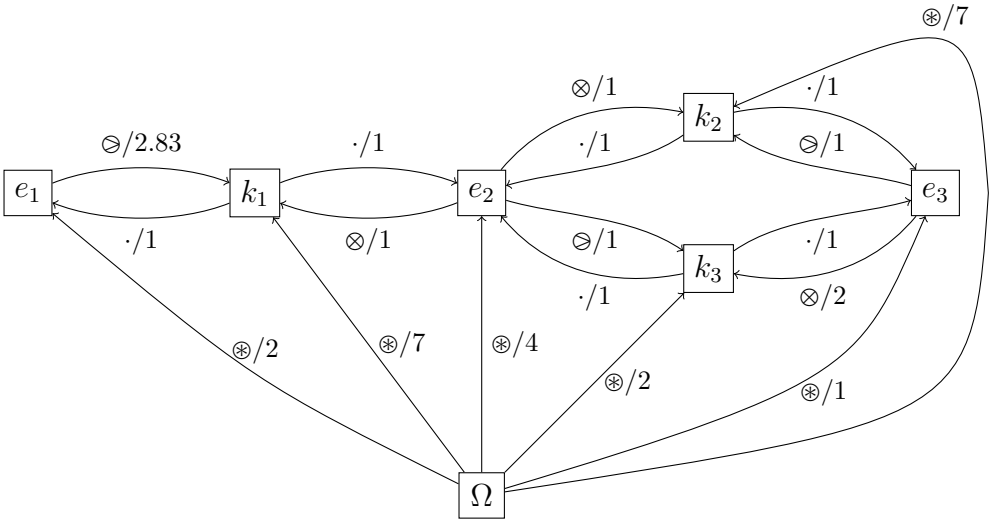
Um alle möglichen Folgen von Suchbefehlen darzustellen und auch um die Kostenfunktion heuristisch zu minimieren definieren wir für einen Mustergraphen einen zugehörigen *Suchplangraphen*.

Der Suchplangraph ist dem Mustergraphen ähnlich, besteht jedoch aus mehr Ecken und Kanten. Genauer gesagt wird für jede Ecke im Mustergraphen eine Ecke und eine Kante im Suchplangraphen erzeugt. Für jede Kante im Mustergraphen erzeugen wir eine Ecke und fünf Kanten. Die genaue Konstruktion wird in Definition 5.9 festgelegt.

Definition 5.9 (Suchplangraph eines Mustergraphen) Sei L ein Mustergraph; falls zusätzlich eine Kostenfunktion c und die Mannigfaltigkeiten θ der V-Strukturen gegeben sind, können auch die nachstehenden ξ berechnet werden, sonst sind sie nicht vorhanden. Der zu L gehörige *Suchplangraph* $\hat{L}$ ist ein



(a) Mustergraph L



(b) Suchplangraph $\hat{L}$

Abbildung 5.8: Der Suchplangraph $\hat{L}$ des Mustergraphen L den wir auch schon in Abbildung 5.7(a) verwendet haben. Die Kosten sind bezogen auf den Arbeitsgraphen aus Abbildung 5.7(b).

markierter gerichteter Graph mit

$$
\begin{aligned}
\forall e \in E_L \;:\; & E_{\hat{L}} := E_{\hat{L}} \cup \{e\},\ K_{\hat{L}} := K_{\hat{L}} \cup \{e_{\circledast/\xi_\circledast}\},\\
& \mathrm{src}_{\hat{L}}(e_{\circledast/\xi_\circledast}) := \Omega,\ \mathrm{tgt}_{\hat{L}}(e_{\circledast/\xi_\circledast}) := e,\\
& \xi_\circledast = c(\mathrm{lkp}(e : \mathrm{typ}_{E_L}(e)))\\
\forall k \in K_L \;:\; & E_{\hat{L}} := E_{\hat{L}} \cup \{k\},\ K_{\hat{L}} := K_{\hat{L}} \cup \{k_{\circledast/\xi_\circledast}, k_{\ominus/\xi_\ominus}, k_{\otimes/\xi_\otimes}, k^1_{\cdot/0}, k^2_{\cdot/0}\},\\
& \mathrm{src}_{\hat{L}}(k_{\circledast/\xi_\circledast}) := \Omega,\ \mathrm{tgt}_{\hat{L}}(k_{\circledast/\xi_\circledast}) := k,\\
& \mathrm{src}_{\hat{L}}(k_{\ominus/\xi_\ominus}) := \mathrm{src}_L(k),\ \mathrm{tgt}_{\hat{L}}(k_{\ominus/\xi_\ominus}) := k,\\
& \mathrm{src}_{\hat{L}}(k_{\otimes/\xi_\otimes}) := \mathrm{tgt}_L(k),\ \mathrm{tgt}_{\hat{L}}(k_{\otimes/\xi_\otimes}) := k,\\
& \mathrm{src}_{\hat{L}}(k^1_{\cdot/0}) := k,\ \mathrm{tgt}_{\hat{L}}(k^1_{\cdot/0}) := \mathrm{src}_L(k),\\
& \mathrm{src}_{\hat{L}}(k^2_{\cdot/0}) := k,\ \mathrm{tgt}_{\hat{L}}(k^2_{\cdot/0}) := \mathrm{tgt}_L(k),\\
& \xi_\circledast = c(\mathrm{lkp}(k : \mathrm{typ}_{K_L}(k))),\\
& \xi_\ominus = \theta_{(\mathrm{typ}_{K_L}(\mathrm{src}_{K_L}(k)), \mathrm{typ}_{K_L}(k)), \mathrm{typ}_{K_L}(\mathrm{tgt}_{K_L}(k))},\\
& \xi_\otimes = \theta_{(\mathrm{typ}_{K_L}(\mathrm{tgt}_{K_L}(k)), \mathrm{typ}_{K_L}(k)), \mathrm{typ}_{K_L}(\mathrm{src}_{K_L}(k))}
\end{aligned}
$$

wobei die Markierungen der Graphelemente von $\hat{L}$ jeweils durch den Index gegeben ist. Falls die ξ angegeben sind, dann heisst $\hat{L}$ *Suchplangraph mit Kosten*.

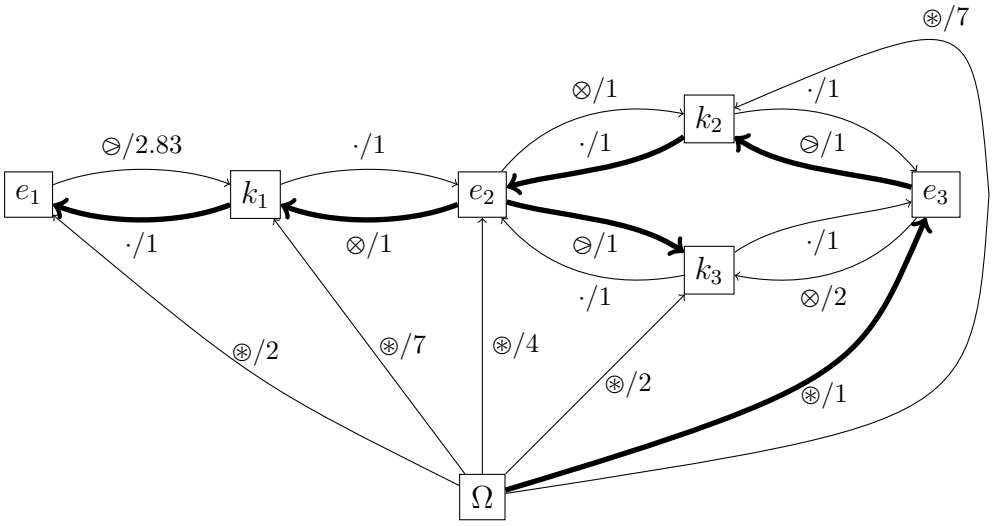


Abbildung 5.9: Suchplangraph $\hat{L}$ (aus Abbildung 5.8) mit MSA A (dicke Pfeile).

Die Symbole ⊛, ⊘ und ⊗ haben folgende Bedeutung[8]: Das Nachschlagen wird durch ⊛ symbolisiert. Das Folgen einer Kante in oder gegen ihre Richtung wird durch ⊘ und ⊗ repräsentiert. Die Zahl nach dem „/“-Zeichen gibt die jeweiligen Kosten an, die von der Heuristik vorhergesagt werden, wenn der jeweilige Befehl auf einem gegeben Arbeitsgraphen ausgeführt wird. In Abbildung 5.8 sehen wir einen Suchplangraphen $\hat{L}$ und den zugehörigen Mustergraph L.

5.3.4 Befehlsauswahl

Die Auswahl von Suchbefehlen kann nun mithilfe des Suchplangraphen beschrieben werden. Wir wollen eigentlich Suchprogramme $S = (s_1, \ldots, s_q)$ finden, welche die Kosten aus Definition 5.8 minimieren. Da aber keine analytische Lösung für dieses Problem bekannt ist, gehen wir den Weg, den signifikantesten Term, also

$$c(s_1)c(s_2)\cdots c(s_q)$$

zu minimieren.

Der Ansatz hierzu ist es, auszunutzen, dass jeder spannende Arboreszent[9] des Suchplangraph den Suchbefehlen eines Suchprogramms für den zugeordneten Mustergraphen entspricht. Da wir die Kosten schon an die einzelnen Kanten annotiert haben, brauchen wir nur noch den *minimalen spannenden Arboreszenten* (MSA) bestimmen, um die Summe

$$c(s_1) + c(s_2) + \ldots + c(s_q)$$

zu minimieren. Wir wollen zwar das Produkt $c(s_1)c(s_2)\cdots c(s_q)$ minimieren, dies ist jedoch nicht problematisch, da wir die Kosten einfach logarithmieren[10] können und so die multiplikativen Kosten additiv werden. Der MSA impliziert eine Menge

[8] Wir haben die Symbole so gewählt, dass sie jeweils ihrer Bedeutung entsprechend einem Geburtsstern, einer Pfeilspitze und einem Pfeilende ähneln.

[9] Ein Arboreszent ist die Verallgemeinerung des Spannbaumes auf gerichtete Graphen.

[10] Natürlich sind ggf. Kosten gleich (und nahe) 0 besonders zu behandeln (z. B. durch Abschneiden).

von Suchbefehlen. Wenn man jene Suchbefehle für ein Suchprogramm wählt, dann minimiert dies den signifikantesten Term der Kosten ebendieses Suchprogramms.

In Abbildung 5.9 ist ein minimaler spannender Arboreszent für den Beispiel-Mustergraphen ausgewählt. Die durch den MSA implizierten Suchbefehle sind natürlich abhängig von der Ausgestaltung der jeweiligen VM. Beherrscht zum Beispiel die VM das Nachschlagen und Erweitern von Kanten gemäß ihres Typs (ohne den Typ des anderen adjazenten Eckentyps zu berücksichtigen) dann ist folgende Menge O an Suchbefehle aus dem Suchplangraph abzulesen.

$$\begin{aligned} O \;= \{\;\; & \mathrm{lkp}(e_3 : C), \mathrm{ext}(e_3 \xrightarrow{k_2:F}), \mathrm{ext}(e_2 : B \xleftarrow{k_2}), \mathrm{ext}(e_2 \xrightarrow{k_3:G}), \\ & \mathrm{ext}(e_2 \xleftarrow{k_1:F}), \mathrm{ext}(e_1 : A \xrightarrow{k_1}) \;\;\} \end{aligned}$$

Es wird also für jede Kante im MSA ein Suchbefehl generiert.

Ist hingegen die VM in der Lage auch zusätzlich den Typ des anderen adjazenten Eckentyps zu berücksichtigen, dann folgen die Suchbefehle (mit Ausnahme des Nachschlagens) aus jeweils einer Ecke x, die eine Kanten des Mustergraphen repräsentiert und den beiden adjazenten Kanten von x, sofern sie im MSA auftreten. Kommt eine der Kanten nicht im MSA vor (in unserem Beispiel ist dies die Kante von k_3 nach e_3), so wurde das adjazente Graphelemente schon zur Passung gebracht, muss damit also nicht noch einmal gepasst werden. Es reicht dann, einen „kleineren“ Suchbefehl zu benutzen. Allerdings sind in solchen Fällen chk-Operationen nötig, die wir in Abschnitt 5.3.5 in die angeordneten Sequenzen von Suchbefehlen einfügen, um diese zu vollständigen Suchprogrammen zu machen. Für eine solche VM ergibt sich O' als Menge an Suchbefehlen.

$$O' \;= \{\;\; \mathrm{lkp}(e_3 : C), \mathrm{ext}(e_3 \xrightarrow{k_2:F} e_2 : B), \mathrm{ext}(e_2 \xrightarrow{k_3:G}), \mathrm{ext}(e_2 \xleftarrow{k_1:F} e_1 : A) \;\;\}$$

5.3.5 Befehlsanordnung

Schließlich müssen wir die Suchbefehle aus einer Befehlsauswahl O noch anordnen und zu einem Suchprogramm für einen Mustergraphen vervollständigen. Der MSA entspricht also nicht direkt einem Suchprogramm, da er eine gewisse Wahlfreiheit bei der Anordnung der durch den MSA gegeben Suchbefehle lässt. Im Speziellen ist jede *topologische Sortierung* der ausgewählten Befehle bezüglich des MSA ein Suchprogramm.

Somit können wir unter anderem aus der Befehlsauswahl O des letzten Abschnitts folgende Suchprogramme generieren:

$$\begin{aligned} S_1 \;= (\;\; & \mathrm{lkp}(e_3 : C), \mathrm{ext}(e_3 \xrightarrow{k_2:F}), \mathrm{ext}(e_2 : B \xleftarrow{k_2}), \mathrm{ext}(e_2 \xrightarrow{k_3:G}), \mathrm{chk}(e_3 \xleftarrow{k_3}), \\ & \mathrm{ext}(e_2 \xleftarrow{k_1:F}), \mathrm{ext}(e_1 : A \xrightarrow{k_1}) \;\;) \\ S_2 \;= (\;\; & \mathrm{lkp}(e_3 : C), \mathrm{ext}(e_3 \xrightarrow{k_2:F}), \mathrm{ext}(e_2 : B \xleftarrow{k_2}), \mathrm{ext}(e_2 \xleftarrow{k_1:F}), \mathrm{ext}(e_1 : A \xrightarrow{k_1}), \\ & \mathrm{ext}(e_2 \xrightarrow{k_3:G}), \mathrm{chk}(e_3 \xleftarrow{k_3}) \;\;) \end{aligned}$$

Jede Kante x im MSA, die mit $\cdot$ markiert ist, kann dabei je nach Kontext entweder zum ext-Befehl werden oder zu einem chk-Befehl. Ersteres entspricht dem Passen einer zu der Kante x adjazenten Ecke. Der ext-Befehl degeneriert zum chk-Befehl genau dann, wenn ebendiese adjazente Ecke durch andere Befehle schon zur Passung

gebracht wurde. In diesem Fall ist eben nur noch zu prüfen ob der Graphzusammenhang des Arbeitsgraphen, dem des Mustergraphen entspricht, daher der chk-Befehl.

Wenn unsere VM wiederum reichhaltigere Befehle aufweist, dann können wir die Befehlsauswahl O' aus dem vorherigen Abschnitt zu folgendem Suchprogramm komplettieren:

$$S_1' = \big(\quad \mathrm{lkp}(e_3 : C), \mathrm{ext}(e_3 \xrightarrow{k_2:F} e_2 : B), \mathrm{ext}(e_2 \xrightarrow{k_3:G}), \mathrm{chk}(e_3 \xleftarrow{k_3}), \\ \mathrm{ext}(e_2 \xleftarrow{k_1:F} e_1 : A) \quad \big)$$

Die beiden ext- und chk-Befehle für die Kante k_3 können, falls die VM einen solchen Befehl auch unterstützt, zusammengezogen werden. Mit dieser Optimierung ergibt sich das Suchprogramm

$$S_1'' = \big(\quad \mathrm{lkp}(e_3 : C), \mathrm{ext}(e_3 \xrightarrow{k_2:F} e_2 : B), \mathrm{ext}(e_2 \xrightarrow{k_3:G} e_3), \mathrm{ext}(e_2 \xleftarrow{k_1:F} e_1 : A) \quad \big)$$

Nun ist es an der Zeit noch einmal auf den Ansatz von Dörr einzugehen: Das dörrsche Verfahren (siehe Abschnitt 3.1.3.2) kann nämlich mittels speziell gewählter Kosten beschrieben werden. Die Kosten werden genau dann unendlich gesetzt, wenn einer der ext-Befehle auf eine entsprechende V-Struktur im Arbeitsgraphen trifft, kurz irgendein $\theta > 0$ ist. Sonst sind die Kosten für die Suchbefehle auf 0 zu setzen. Ist es nun möglich einen MSA zu finden (der dann offenbar 0-Kosten hat), können also alle V-Strukturen im Arbeitsgraphen *sicher* umgangen werden. Ein aus dem MSA bestimmtes Suchprogramm entspricht den von Dörr propagierten Suchplänen. Der dörrsche Ansatz ist also als Spezialfall unseres Ansatz modellierbar.

5.4 Zusammenfassung

Unsere Erfahrung hat gezeigt, dass die indirekte Lösung zwar schnell zu einer brauchbaren Implementierung führen kann, allerdings ist ihre Leistungsfähigkeit, wie wir in Kapitel 9 sehen werden, beschränkt. Ausschließlich Verfahren, die direkte Ansätze verfolgten, haben in diesen Tests akzeptable Leistungen erreicht. Für die Entwicklung unsers GrGen-Systems war es eine effektive Vorgehensweise zuerst relationale Algebra zu benutzen und diese dann durch die viel elaboriertere suchplanbasierte Technik zu ersetzen, da wir so zu einem frühen Zeitpunkt zu einem funktionsfähigen System kamen.

Die Beschleunigung der Graphmustersuche unseres suchplanbasierten Verfahrens hat zwei Wurzeln: Zum einen die Suchplanung, die für ein einzelnes Muster in Bezug auf einen Arbeitsgraphen oder mittels Domänenwissen auch für eine ganze Klasse von Arbeitsgraphen die Suchprogramme heuristisch so bestimmt, dass möglichst keine Auffächerung des Suchraumes stattfindet. Zum zweiten trägt die Suchraumfortschaltung dazu bei, dass für Sequenzen von Regelanwendungen die Anzahl der Elemente des Arbeitsgraphen, die untersucht werden müssen, klein bleibt.

TEIL II

Anwendung

KAPITEL 6

GRUNDLAGEN UND PROBLEME AUS DEM ÜBERSETZERBAU

Die ersten Übersetzer stammen von Rutishauser von der ETH Zürich [Rut52] und Hopper, die damals für die Sperry Cooperation das A-0 Programmiersystem baute [Hop52]. Seit dieser Zeit hat sich der Forschungsstand erheblich weiterentwickelt und ist inzwischen fast unüberschaubar geworden. Wir geben in diesem Kapitel eine Einführung in den zeitgenössischen Übersetzerbau, wobei wir besonderen Wert auf die Punkte legen, die uns später bei der Integration der Graphersetzung helfen werden. Der nächste Abschnitt stellt eine kurze Einleitung dar, beleuchtet einige aktuelle und prinzipielle Probleme, woraus wir unsere Zielbestimmung im Bereich Übersetzerbau ableiten. Die in der Einleitung (Abschnitt 6.1) skizzierte Übersetzerarchitektur wird in Abschnitt 6.2 ausgearbeitet. In den beiden darauffolgenden Abschnitten werden die Zwischendarstellungen (Abschnitt 6.3) und Rechnerarchitekturen (Abschnitt 6.4) genauer beleuchtet. Abschließend stellen wir verwandte Arbeiten vor (Abschnitt 6.5).

6.1 Einleitung

Übersetzer sind Programmsysteme, die Texte, die in einer Programmiersprache geschrieben sind, automatisch in eine Maschinensprache umwandeln. Pragmatisch gesehen hat der Erbauer eines Übersetzers dazu drei Aufgaben zu lösen: Erstens die vollständige und korrekte Erfassung einer Programmiersprache und die Umwandlung derer Programme in äquivalente Maschinenprogramme. Zweitens sollen die Programme möglichst effizient ablaufen. Dazu sind die Gegebenheiten der Zielmaschinenarchitektur zu berücksichtigen. Drittens soll der Bau eines Übersetzers mit geringem Aufwand möglich sein. In diesem Teil der Arbeit widmen wir uns neuen Techniken, welche die Unvereinbarkeit der drei Punkte vermindern. Die bessere Nutzung von Maschineneigenschaften bei gleichzeitig personaleffizienter Konstruktion von Übersetzern ist dabei unser Ziel.

6.1.1 Ausgangslage und Probleme

Heutzutage sind Übersetzer modular aufgebaut und in Phasen eingeteilt. Zentraler Bestandteil des Übersetzers ist die sogenannte *Zwischendarstellung*[1]. Sie stellt das Eingabeprogramm (im Weiteren *Quellprogramm* genannt) so weit wie möglich

[1] engl.: *intermediate representation* (IR)

unabhängig von der Quellsprache einerseits und der Architektur der Zielmaschine andererseits dar. Die meisten Analysen und Optimierungen des Übersetzers arbeiten auf dieser Zwischendarstellung.

Das *Frontend* hat die Aufgabe, ein Quellprogramm auf *Wohlgeformtheit* der Syntax und der statischen Semantik im Sinne der Definition der Quellsprache zu überprüfen und in die Zwischendarstellung zu überführen. Das *Backend* erzeugt aus dem in der Zwischendarstellung repräsentierten Programm Befehle für eine bestimmte Architektur, das so genannte *Zielprogramm*.

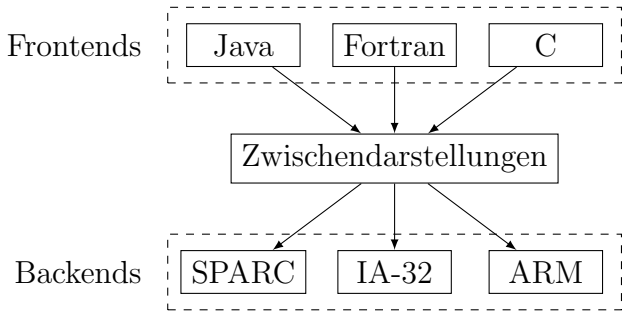


Abbildung 6.1: Schematischer Aufbau eines Übersetzers

Diese in Abbildung 6.1 schematisch dargestellte *Übersetzer-Architektur* hat den Vorteil, dass man nicht für jede Kombination aus Quellsprachen und Zielmaschinenarchitektur einen eigenen Übersetzer schreiben muss. Man konzipiert als Hauptbestandteil eine Zwischendarstellung, an die „nur noch“ Frontends und Backends anzuschließen sind.

Wie in allen Bereichen der Softwarekonstruktion, so ist auch im Übersetzerbau seit Langem ein Trend zur steigenden Abstraktion bei den Konstruktionsverfahren zu erkennen. Zum Beispiel wird die lexikalische bzw. syntaktische Analyse durch reguläre bzw. kontextfreie Sprachen spezifiziert und die Implementierung mit Hilfe von speziellen Werkzeugen generiert [Les75, Joh79, Vie88]. Ebenso wurde in den Achtzigern des letzten Jahrhunderts begonnen, ebenfalls die Befehlsauswahl nicht mehr direkt zu implementieren, sondern durch LR-Grammatiken [GG78] oder zehn Jahre später mit Baumautomaten [PLG88, ESL89] auszudrücken und daraus Übersetzerbestandteile zu generieren. Der Einsatz von Generatoren ermöglicht offenbar ein Entwerfen und Implementieren von Teilen des Übersetzers auf einem Niveau, das oberhalb dessen von Programmiersprachen liegt, was dreierlei Vorteile bietet: Die Spezifikationen werden *kürzer*, sie sind nahe am intuitiven Verständnis z. B. der Grammatik einer Programmiersprache bzw. der Befehlsauswahl und darum ist ihre Erstellung *leichter* und *fehlerärmer* möglich [Slo95, Kle05].

Diese Erfahrungen legen es nahe, dass – aus Gründen der Wirtschaftlichkeit und Korrektheit – auch andere Teile des Übersetzers so weit wie möglich automatisch, d. h. mit Hilfe von Generatoren aus abstrakten Spezifikationen, gebaut werden sollten. Auf Ebene der Zwischendarstellung wurde in den vergangenen Jahrzehnten einiger Fortschritt erzielt. Heute kennen wir *graphbasierte Zwischensprachen*, deren gemeinsames Entwurfsziel die Abstraktion von unwichtigen Artefakten der Quellsprache, sowie die effiziente Darstellung der für die Optimierungen wichtigen Zusammenhänge ist. Die *Bedeutung* von Programmen kann so mithilfe zweier Konzepte festgehalten werden: Erstens dem *Datenfluss*, der den Zusammenhang zwischen Berechnungen, also von Werten, herstellt, und zweitens dem *Steuerfluss*, der angibt in

welcher Reihenfolge bestimmte Berechnungen ausgeführt werden. Diese Zwischensprachen stellen Programme als Überlagerung von solchem Daten- und Steuerfluss in Form eines *Programmgraphen* dar.

Reichhaltige Befehlssätze, wie zum Beispiel SSE von Intel [Int07a] oder AltiVec von IBM/Motorola [Mot98], zeichnen sich vor allem dadurch aus, dass ein großer Teil der Befehle keine Entsprechung in üblichen Programmiersprachen hat, sondern diese eher Idiomen oder Mustern[2] gleichen, wie sie bei gewissen Klassen von Algorithmen anzutreffen sind. Diese Befehle arbeiten entweder nach dem *SIMD*-Prinzip[3] auf *gepackten Registern* oder führen mehrere Programmablaufänderungen und Berechnungen gleichzeitig aus (siehe auch Abschnitt 6.4). Charakteristisch ist auch das Verwenden von speziellen Registern, die zum Teil breiter als Maschinenworte sind, sowie dedizierter Lade- und Speicherbefehle. In mehreren Arbeiten wurde nachgewiesen, dass es insbesondere bei gewissen Algorithmenklassen durch den Einsatz reichhaltiger Befehlssätze zu großen Beschleunigungen kommt [NJ99, SB01, BGG05]. Strey und Bange berichten in diesem Zusammenhang von beinahe Faktor 10 bei der Optimierung von neuronalen Netzwerken [SB01] durch den manuellen Einsatz von Intels MMX-Befehlen.

Die in gegenwärtigen Zwischensprachen auftretenden Operationen sind meist an einem RISC-Befehlssatz angelehnt. Diese minimalistischen Sprachschätze lassen allerdings nicht offenbar werden, an welchen Stellen Befehle aus reichhaltigen Befehlssätzen verwendet werden sollten, um gewisse Eigenschaften des Zielprogramms zu verbessern. Ebenso wenig sind die bekannten Backends in der Lage, die oft einige duzend Zwischensprachoperationen großen Teilgraphen zu identifizieren, die durch einen einzigen reichhaltigen Befehl ersetzt werden könnten. Also besteht eine Kluft zwischen herkömmlichen Zwischensprachen mitsamt der darauf basierenden Verfahren zur Erstellung des Zielprogramms und den reichhaltigen Befehlssätzen. Wir werden in diesem Teil der Arbeit Methoden präsentieren, die diese Lücke füllen.

6.1.2 Lösungsansatz

Wir verwenden Graphersetzung und deklarative Graphmustersuche zur Manipulation und Suche von Instanzen einer graphbasierten Zwischensprache und arbeiten somit auf einer natürlichen Abstraktionsebene. Allerdings sind wir nicht per se an Graphersetzung interessiert, sondern vielmehr an einem Verfahren, das die bestmöglichste Nutzung von Maschineneigenschaften, u. a. reichhaltiger Befehlssätze, bei gleichzeitig personaleffizienter Konstruktion von Übersetzern gewährleistet. Unsere Lösung muss hier also breiter ansetzen, als *nur* ein Graphersetzungssystem für den Übersetzerbau vorzuschlagen. Wir erweitern die klassische Konstruktion von Übersetzern durch folgende Punkte:

- Eine konzise und ausdrucksstarke Spezifikationstechnik für die Wirkung von komplexen Maschinenbefehlen.
- Einen Generator, der aus solchen Spezifikationen Graphersetzungsregeln herstellt.
- Eine Technik zur Reduktion von nicht essenziellen Programmabhängigkeiten, sowie normalisierenden Darstellungen von Zwischensprachengraphen.

[2] engl.: *pattern*
[3] engl.: *single instruction multiple data*

- Eine anwendungsspezifische Sprache zur direkten Integration von *Graphoperationen* in den Konstruktionsprozess von Optimierungen.

Im folgenden Kapitel ist eine komplette Übersetzerarchitektur dargestellt, in die unsere Methoden eingebettet sind. In Kapitel 7 diskutieren wir besonders die Auswirkungen auf weitere Aspekte des Übersetzerbaus, wie der Befehlsauswahl und Konstruktion allgemeiner Optimierungen.

6.1.3 Ökonomie des Übersetzerbaus

Die Ökonomie des Übersetzerbaus wird leider viel zu selten thematisiert. So kommt Robison [Rob01] auch zu der Überzeugung, dass Optimierungen einerseits nur ein kleiner Teil der „realen Übersetzer“ sind, aber auch, dass die Optimierungen mehr in die Hand der Benutzer gelegt werden sollten. Unsere Optimierung für reichhaltige Befehlssätze kommt beiden Forderungen nach (siehe Abschnitt 7.4.2). Denn mit ihr kann der Benutzer eines Übersetzers erstmalig ohne Kenntnis der Interna des Übersetzers diesen um neue Teiloptimierung erweitern.

6.1.3.1 Optimierungen

Zunächst soll hier der Begriff Optimierung[4] im Bezug auf den Übersetzerbau betrachtet werden. In der Umgangssprache, wie auch in der Mathematik, versteht man unter Optimieren einen Vorgang, bei dem so lange nach Alternativen gesucht wird, bis eine[5] beste Lösung für ein Problem gefunden wird. Voraussetzung hierfür ist es eine Zielfunktion, deren Werte maximiert oder minimiert werden sollen, sowie den zulässigen Bereich der Lösungen zu besitzen.

Leider ist gerade im Übersetzerbau beides nur in wenigen Fällen wirklich der Fall. Nehmen wir an, wir hätten eine Zielfunktion für Aufgaben[6] wie Registerzuteilung, Befehlsauswahl oder Alias-Analyse gegeben und verlangen nun, dass diese Aufgaben wirklich optimal gelöst werden – es also keine bessere zulässige Lösung gibt. In diesem Fall sind diese Aufgaben nicht berechenbar, also i. A. nicht algorithmisch lösbar. Warum ist das so? Betrachten wir den Fall der Registerzuteilung. Nehmen wir an, die zum minimierende Zielfunktion sei die statische Anzahl der Auslagerungsoperationen in den Speicher. Es könnte nun eine Belegung der Register mit Werten so gewählt werden, dass z. B. in einem Programmpfad der zwar statisch gesehen ausführbar ist, aber tatsächlich nie genommen wird einfach alle Auslagerungsbefehle weggelassen werden. Dies ist korrekt, falls man beweisen kann, dass der nämliche Pfad nie genommen wird, was leider der Lösung des Halteproblems entspricht. Die anderen Aufgaben können mit analogen Argumenten als nicht berechenbar bewiesen werden.

Allerdings ist noch nicht einmal die Zielfunktion überhaupt bekannt. Zumindest nicht in dem Sinn, dass das Ergebnis einer verwendeten Zielfunktion für eine

[4] lat. optimum: „das Beste“

[5] Wir sprechen hier wegen der ggf. nicht eindeutigen besten Lösung von einer besten Lösung und nicht von *der* besten Lösung.

[6] Die Registerzuteilung ist die Aufgabe im Backend, die die unbeschränkt vielen Werte auf endlich viele Register aufteilt. Die Befehlsauswahl bestimmt für ein Programm in einer Zwischendarstellung eine – bezüglich gewisser Kriterien – möglichst gute Auswahl an Maschinenbefehlen. Die Alias-Analyse soll herausfinden, ob zwei Namen dasselbe Objekt meinen.

Optimierung tatsächlich mit der durch sie modellierten Größe in der Realität übereinstimmt. Nehmen wir z. B. das statische Berechnen der Laufzeit eines Programms: Das komplexe Zusammenspiel von Cache-Hierarchie, Multiskalarität, Mehrfädigkeit[7] etc. ist weder dokumentiert noch kann es empirisch genau bestimmt werden. Die Zeiten, in denen die Hardwarehersteller in den Handbüchern „Takte pro Befehl“ angegeben haben, sind schon mehrere Jahrzehnte vorbei. Man gibt sich bei fast allen modernen Prozessoren mit unteren und oberen Schranken zufrieden.

Somit definiert man sich im Übersetzerbau immer ein Modell des zu optimierenden Problems und löst die Aufgaben optimal bezüglich dieses Modells; im schlimmsten Fall haben die so erhaltenen „optimalen Lösungen“, die ja auf realer Prozessoren ausgeführt werden, nichts mit dem echten „Optimum“ zu tun! Das Finden und die Wahl eines adäquaten Modells ist somit ein nicht zu unterschätzender Teil des Übersetzerbaus. Allerdings bleibt es in fast allen Fällen eben nur eine Approximation an die Wirklichkeit.

Abgesehen von diesen unüberwindbaren Hindernissen, ist selbst dann – bei den in der Regel recht einfachen gewählten Modellen – die Komplexität der Aufgaben immer noch mindestens NP-hart, sodass man sich wiederum mit Approximation der NP-Probleme begnügt; ebenfalls zulasten des Realitätsbezugs.

Insofern ist der Begriff Optimierung, wie er im Übersetzerbau gemeinhin verwendet wird, eher als „Verfahren zur Erzielung einer *erhofften* Verbesserung bezüglich gewisser Modelle“ zu verstehen. Einfachheitshalber nennen wir dies aber weiterhin optimieren, ohne allerdings die abschwächende Konnotation zu vergessen.

6.1.3.2 Proebsting's Law

Im vorangehenden Abschnitt betrachten wir den Begriff der Optimierung selbst, in diesem Abschnitt geht es um den Effekt aller Optimierungen in Gesamtheit. Proebsting untersuchte 1998 die durch Optimierungen erzielte Beschleunigung und kam kurz gesagt zu enttäuschenden Ergebnissen: Er konnte eine Verdopplung alle 18 Jahre nachweisen [Pro98]. Auf der einen Seite ist die Leistungssteigerung der Programme durch Fortschritte im Übersetzerbau – gemessen über 35 Jahre – also recht gering gewesen. Andererseits hat Proebsting die ökonomischen Überlegungen weitgehend ausgeklammert: Im Verhältnis zu einer neuen Chipgeneration ist ein neuer Übersetzer sehr preisgünstig. Außerdem konnten gewisse wissenschaftlich und auch wirtschaftlich besonders interessante Anwendungen gerade im numerischen Bereich um mehr als den von Proebsting bestimmten Faktor vier pro 35 Jahren beschleunigt werden [Sco01].

Schließlich ist die größte Leistungssteigerung im Übersetzerbau nicht die, dass die erstellten Programme schneller ablaufen, sondern, dass die Erstellung *selbst* effizienter vonstattengeht. Moderne zeitgenössische Programmiersprachen (z. B. Java und C#), Programmierkonzepte (z. B. OO, MDA und Programmsanierung[8]) und Programmierumgebungen (z. B. Eclipse, Visual Studio) haben die Programmerstellung schneller, weniger fehleranfällig und letztlich auch ingenieurmäßiger werden lassen. Alle diese positiven Wirkungen basieren auf der zunehmenden Abstraktion, weg von der Hardware, hin zu einer adäquaten Problembeschreibung. Ohne Unterstützung durch den Übersetzerbau wäre diese Entwicklung nicht möglich.

[7] engl.: *multithreading*
[8] engl.: *refcatoring*

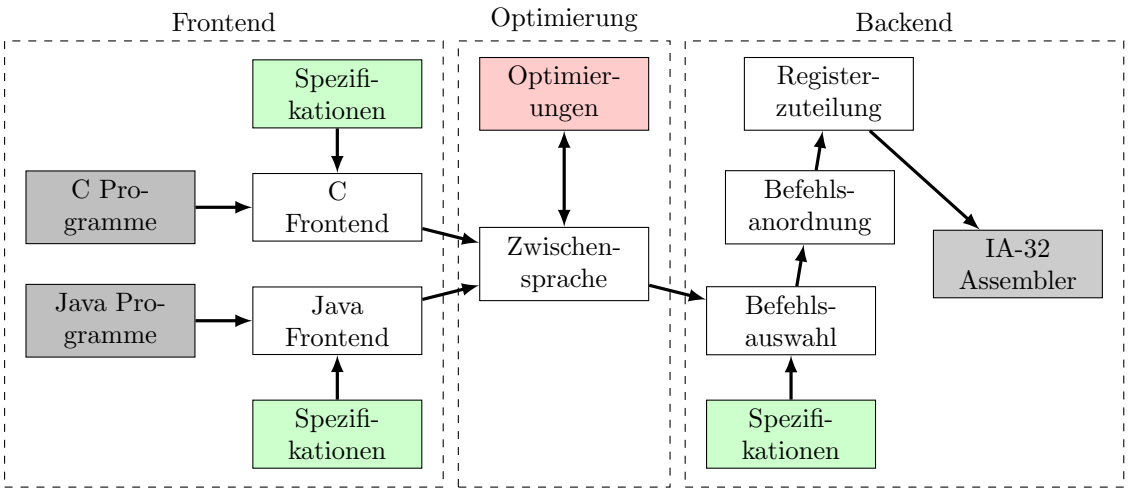


Abbildung 6.2: Beispielhafter Übersetzer

6.2 Architektur von Übersetzern

Wir verfeinern hier die Darstellung von Abbildung 6.1 und geben einen Einblick in die Arbeitsweisen heutiger Übersetzer. Das übliche Architekturmuster eines Übersetzers ist die Fließbandverarbeitung, wobei – grob gesagt – die Stufen in Einlesen der Quelle, syntaktische Analyse, semantische Analyse, Transformation in die Zwischensprache, Optimierung, Befehlsauswahl, Befehlsanordnung und Registerzuteilung sowie Ausgabe des Maschinenprogramms zu unterteilen sind. In Abbildung 6.2 sehen wir eine beispielhafte Übersetzerarchitektur. Teile dieser Einführung in den zeitgenössischen Übersetzerbau folgen dem internen Bericht „Übersetzerbau - Ein kleiner Überblick“ von Sebastian Hack und Rubino Geiß [GH03].

Die *Frontends* sind in der Regel aus Spezifikationen generiert, die aus einer Kombination von regulären und kontextfreien Grammatiken bestehen und mit syntaktischen Anknüpfungen die Eingabeprogramme in eine interne Darstellung überführen. Als (erste) interne Darstellung wird oft der *abstrakte Strukturbaum*[9] verwendet, aus der dann die eigentliche Darstellung in Zwischensprache erzeugt wird. Bei hinreichend einfachen Sprachen ist es allerdings auch möglich, die Zwischensprache direkt aufzubauen. Die Spezifikationen des Frontends bestehen also aus einer Mischung aus deklarativen Regeln einer Grammatik und den imperativ formulierten Anknüpfungen, die ausgeführt werden, sobald eine gewisse Regel zur Konstruktion des Ableitungsbaumes (konkreter Strukturbaum) ausgewählt wurde.

Die *Optimierungsphase* arbeitet auf der Zwischendarstellung und besteht aus bis zu einigen duzend verschiedenen *Optimierungen*. Jede dieser Optimierungen verändert die Zwischendarstellung im Hinblick auf ein gewisses Ziel. Es ist dabei keineswegs klar, welche Auswirkung die Reihenfolge der Anwendung diese Optimierungen hat, noch ob und wenn ja wie sich diese positiv oder negativ beeinflussen. Diese Fragen sind nach wie vor offene Probleme, deren Lösung nicht nur umfangreiche empirische Studien erfordern würden, sondern selbst dann noch inhärente Ungewissheiten aufweisen – denn der Einfluss von Programmierern, Programmierstilen, Programmiersprachen und Zielarchitekturen lässt sich nicht auslöschen. Auch der Begriff Optimierung selbst kann – wie in Abschnitt 6.1.3.1 dargelegt – angezwei-

[9] engl.: *abstract syntax tree* (AST)

felt werden. Viele der Optimierungen (z. B. Alias-Analyse, Erzeugung minimaler arithmetischer Ausdrücke, kontextsensitive Wertanalyse) sind selbst unter vereinfachenden Annahmen NP-schwer und somit allenfalls approximativ lösbar.

Die *Backends* generieren schließlich aus der Zwischendarstellung (meist symbolische) Maschinenprogramme der Zielarchitekturen. Hier gibt es wiederum schwerwiegende Aufgaben zu lösen: Die Befehlsauswahl, die Befehlsanordnung und die Registerzuteilung. Alle diese Probleme sind in den normalerweise verwendeten Modellen NP-schwer. Unangenehmerweise existiert eine Rückkopplung der drei Probleme untereinander. Dies ist das sogenannte *Phasenkopplungsproblem* des Backends. Im Jahr 2006 konnten Hack, Grund und Goos zwar zeigen, dass sich das Phasenkopplungsproblem und auch die NP-Vollständigkeit im Bezug auf die Registerzuteilung anders darstellt, als der bis dahin existierende Stand der Forschung es nahelegten [HGG06], aber dennoch bleiben drei NP-schwere Probleme übrig; sie heißen jetzt nur Befehlsauswahl, Befehlsanordnung und Kopienminimierung und sind nach wie vor gekoppelt. Für die Befehlsauswahl stehen verschiedene baummusterbasierte Verfahren zur Verfügung (siehe zum Beispiel [ESL89]). Befehlsanordnung und Registerzuteilung geschieht meist durch geeignete Parametrisierung eines allgemeinen Algorithmus, das das jeweilige Problem für eine breite Klasse von Zielarchitekturen lösen kann [AK02, Muc97]. Neuere Ansätze versuchen die graphbasierte Zwischensprache bis zur Assemblerausgabe beizubehalten. Meilensteilen auf diesem Gebiet sind die Arbeiten von Eckstein et al., die das ersten *wirklich* auf Graphen arbeitenden Verfahren zur Befehlsauswahl entwickelt haben [EKS03, ES03, SE02]; alle bisherigen Ansätze, die versuchten die Befehlsauswahl auf Graphen zu bewerkstelligen, zerschnitten letztlich die Graphen an arbiträren Stellen zu Bäumen [Boe05].

6.3 Zwischendarstellungen

Für Zwischendarstellungen gibt es keinen „Standard“ wie z. B. die LALR(1)-Grammatiken für die syntaktische Analyse. Obwohl die Zwischendarstellung von Übersetzer zu Übersetzer unterschiedlich ausgeführt ist, gibt es aber gewisse Verfahren und Optimierungen, die allgemein anerkannt sind und in vielen Übersetzern Verwendung finden.

6.3.1 Datenfluss und Steuerfluss

Die Bedeutung eines Programms einer imperativen Hochsprache kann im Wesentlichen durch Daten- und Steuerfluss ausgedrückt werden.

Datenfluss

Der Datenfluss ist der „Fluss“ der Werte (egal ob durch Variablen explizit gemacht oder implizit durch Verwendung von Zwischenergebnissen), wobei Berechnungen Zusammenflüsse im Datenfluss darstellen. Konstanten und parameterlose Funktionen sind Quellen des Datenflusses. Funktionsrücksprünge (z. B. das `return` in C, C++ und Java), Funktionen ohne Rückgabewert und das Schreiben in den Speicher sind i. A. dessen Senken. Man kann den Datenfluss als gerichteten Graphen darstellen, wobei die Werte auf den Kanten fließen, und die Berechnungen die Ecken des Datenflussgraphen[10] sind.

[10] engl.: *data flow graph* (DFG)

Beispiel 6.1 Ein + führt zwei Datenflüsse zusammen, indem es aus zwei Werten einen neuen berechnet. Abbildung 6.3 zeigt ein Beispiel für einen Datenflussgraphen.

Wenn man alle Kanten in die entgegengesetzte Richtung zeigen lässt, so stellen die Kanten nun die Datenabhängigkeiten dar. Dieser Datenabhängigkeitsgraph[11] drückt aus, welche Werte vor anderen berechnet werden müssen.

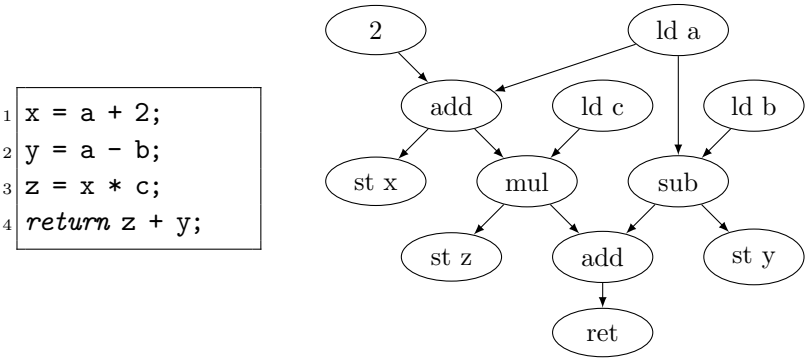


Abbildung 6.3: Programmfragment mit zugehörigem Datenflussgraphen

Steuerfluss

Der Steuerfluss ist letztlich eine Abstraktion des Befehlszeigers einer von Neumann-Maschine. Er abstrahiert den Inhalt des Befehlszeigers und damit die Abarbeitungsreihenfolge der Berechnungen. Auch der Steuerfluss ist als gerichteter Graph darstellbar. Man spricht dann vom Steuerflussgraphen[12]. Die Ecken des CFG sind die Berechnungen, die Kanten drücken die Abarbeitungsreihenfolge der Berechnungen aus. Der CFG hat zwei ausgezeichnete Ecken *start* (Quelle) und *end* (Senke), wobei die Abarbeitung des Programms (der Prozedur) bei *start* beginnt und bei *end* endet.

Man fasst zusammenhängende Ecken des CFGs, die jeweils nur einen Nachfolger und einen Vorgänger haben, zu einem *Grundblock* zusammen. Da in einem Grundblock per definitionem keine Berechnung über das Ausführen einer anderen entscheidet, wird die Berechnungsreihenfolge innerhalb des Grundblocks nur durch die Datenabhängigkeiten bestimmt.

6.3.2 Darstellung

Essenziell für die Effizienz der Zwischendarstellung ist ihre Darstellung, also die Art und Weise, wie das Quellprogramm im Übersetzer dargestellt wird. Dies ist eine Gratwanderung, denn die Zwischendarstellung sollte einerseits nicht zu viele Informationen „verschenken“, da sie bei der Optimierung benötigt werden könnten, andererseits sollte sie das Quellprogramm in eine Form bringen, dass die Erzeugung effizienter Maschinenprogramme möglich macht. Folgendes Beispiel mag diesen Konflikt verdeutlichen:

Beispiel 6.2 Fast jede imperative Programmiersprache besitzt Reihungen[13] in ihren Sprachumfang. Angenommen, in einem Programm werden Reihungen für die

[11] engl.: *data dependence graph* (DDG)
[12] engl.: *control flow graph* (CFG)
[13] engl.: *arrays*

Implementierung von Vektoren benutzt und einige Methoden für das Rechnen mit Vektoren implementiert. Etwa eine Methode, die zwei Vektoren addiert:

```
1 void add(float x[], float y[], float z[])
2 {
3   int i;
4   for(i = 0; i < length(x); i++)
5     z[i] = x[i] + y[i];
6 }
```

Moderne (RISC) Mikroprozessoren besitzen aber keinen Befehl, der einen Reihungszugriff implementiert. Insbesondere dann nicht, wenn Reihungungselemente in der Programmiersprache als Klassen dargestellt sind (wie z. B. in Java). Der Reihungszugriff muss also in einen Speicherzugriff umgesetzt werden, dessen Adresse vorher berechnet werden muss (`data` stellt die *Relativadresse*[14] der eigentlichen Reihung in der Reihungsklasse dar, `x` ist ein Zeiger auf ein Objekt der Reihungsklasse):

```
1 add  t1 = x, data
2 shl  t2 = i, 2
3 add  t3 = t1, t2
4 ld   t1 = [t3]
```

Das ist aber nicht effizient, da der erste Additionsbefehl von der Schleifenvariable `i` völlig unabhängig ist und in jedem Schleifendurchlauf berechnet wird.

In obigem Beispiel kam vielleicht eine Zwischensprache zum Einsatz, die die Adressarithmetik, die das Laden von `x[i]` steuert, nicht in die Optimierungen miteinbezogen hat. Es wäre nötig gewesen, den Reihungszugriff frühzeitig in die Adressarithmetik umzusetzen und einige Optimierungen zu starten, welche die oben beschriebene Berechnung des Versatzes[15] der eigentlichen Reihung im Objekt `x` vor die Schleife zieht.

Andererseits kann man argumentieren, dass es vielleicht eine Optimierung gibt, die Reihungszugriffe optimieren soll, zum Beispiel so, dass Zugriffe auf aufeinander folgende Reihungselemente in SIMD Konstrukte umsetzt werden. Dabei ist es sehr hilfreich (je nach Verfahren sogar unerlässlich), dass der Reihungszugriff noch nicht in Adressarithmetik umgesetzt ist.

Man sieht, dass unterschiedliche Aspekte der Optimierung unterschiedliche Darstellungen erfordern. Deswegen lässt man die Zwischendarstellung oft unterschiedliche Phasen durchlaufen, in denen die Operationen immer mehr den Befehlen der Zielarchitektur entsprechen. Neuste Übersetzer betreiben dieses Herunterbrechen der Operationen sogar soweit, dass die (graphbasierte) Zwischensprache noch nach der Befehlsauswahl erhalten bleibt.

6.3.2.1 Lineare Darstellungen

In einfacheren Übersetzern geschieht die Befehlserzeugung oft direkt aus dem AST[16], indem ein Postfix-Lauf über den AST gestartet wird und bei jeder Ecke einige Befehle erzeugt werden. Das ist natürlich völlig ineffizient (im Sinn der Codequalität),

[14] engl.: *offset*
[15] engl.: *offset*
[16] Prominente Vertreter sind: Java, MSIL von .NET und Pascal-P-Code

da man immer nur die Ecke im Baum betrachten kann, an der man sich momentan befindet. Allerdings wird dieser Code in der Regel später von *Laufzeitübersetzern*[17] optimiert und so effizient ausgeführt. In einem gewissen Sinn ist in einem solchen Fall der erste Übersetzer nur das Frontend, die eigentliche Arbeit, nämlich die Optimierung, wird erst vom Laufzeitübersetzer gemacht.

Etwas aufwendigere Übersetzer erzeugen in diesem AST-Lauf nicht direkt Maschinensprachbefehle, sondern eine Art architekturunabhängige Maschinensprache (den sogenannten Tripelcode), der das Programm dann sehr maschinennah repräsentiert. Alle Hochsprachenelemente, wie zum Beispiel Zugriff auf Verbundfelder, Reihungszugriffe und das polymorphe Aufrufen von Methoden sind schon komplett in Adressarithmetik umgesetzt. So würde z. B. ein Zugriff auf ein Feld in einem C-Struct so dargestellt:

Beispiel 6.3 Fragment eines C-Programms mit zugehöriger linearer Zwischendarstellung.

```
1 struct { int x, y; } a;
2 int i, j;
3 i = (a.x + a.y) / 2;
4 j = i + 1;
```

```
1  mov  t1 = a
2  ld   t2 = [t1]
3  add  t3 = t1, 4
4  ld   t4 = [t3]
5  add  t5 = t4, t2
6  div  t6 = t5, 2
7  st   i = t6
8  ld   t7 = i
9  add  t8 = t7, 1
10 st   j = t8
```

Diese Darstellung ist wenig geeignet um effiziente Befehlssequenzen zu erzeugen, da die Befehle in einer gewissen Weise schon angeordnet sind. Fließbandeffekte und Registerzuteilung sind daher schlecht zu berücksichtigen, da Datenabhängigkeiten oder vielmehr unabhängige Operationen erst aufwendig als solche erkannt werden müssen.

6.3.2.2 Baumdarstellungen

Eine Verfeinerung der obigen Tripelcodedarstellung sind die Baumdarstellungen. Sie benutzen keine Pseudo-Maschinensprachbefehle, die linear angeordnet sind, sondern stellen das Programm als Sequenz von Ausdrucksbäumen dar. Dies stellt schon eine Verbesserung gegenüber den linearen Darstellungen dar, da zumindest die Ausdrücke nicht als Sequenz hingeschrieben sind, sondern ihre Abhängigkeiten direkt als Baum dargestellt sind. Jedoch sind die einzelnen Ausdrücke immer noch sequenziell aneinandergereiht. Abbildung 6.4 zeigt obiges Beispiel in einer Baumdarstellung im Gegensatz zu Abbildung 6.3 ist hier die Adressarithmetik explizit.

6.3.2.3 Graphbasierte Zwischendarstellungen

Es hat sich gezeigt, dass man für besonders erfolgreiches Optimieren sehr viele Informationen des Quellprogramms beibehalten muss. Zum Beispiel sind Typinformationen von entscheidender Bedeutung, wenn es um die Optimierung objektorientierter

[17] engl.: *just in time compiler* (JIT)

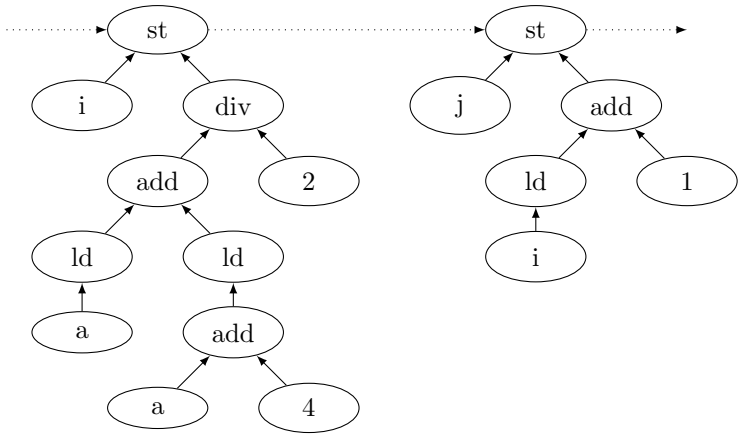


Abbildung 6.4: Baumdarstellung

Programme geht. Solche Informationen sind in einer Tripelcodedarstellung natürlich nicht mehr vorhanden, da z. B. die Felder eines Verbundes schon in *Relativadressen* umgerechnet sind. In Verbunden nicht benutzte Felder kann man dann nicht mehr „wegoptimieren“, da gerade die Information, dass es sich um einen Verbund gehandelt hat, nicht mehr explizit verfügbar ist.

Darüber hinaus ist die Tripelcodedarstellung zu „eindimensional“, da diese Form der Zwischenrepräsentation schon eine gewisse Auswahl und Anordnung der letztlich erzeugten Maschinensprachbefehle suggeriert. Man geht deswegen heute dazu über, in der Zwischendarstellung nur noch Datenabhängigkeiten darzustellen. Zwangsläufig endet man dann bei einer Zwischendarstellung, die man als Graph interpretieren kann. Der Graph stellt dann die „Wert-hängt-von-einem-anderen-Wert-ab“-Relation dar. Diese Darstellung lässt die Anordnung der später erzeugten Befehle völlig offen und ist wie in Abbildung 6.3 ersichtlich, nicht mehr intuitiv mit einer Maschinensprache vergleichbar. Neben den vielen Vorteilen einer solchen Darstellung (wie z. B. das erheblich einfachere Implementieren vieler Optimierungen) ist sie auch ästhetisch der Tripelcodedarstellung vorzuziehen, weil sie auf die für die richtige Übersetzung des Programms notwendigen Informationen fokussiert.

Der Steuerfluss wird in solchen Darstellungen auch als Graph dargestellt. Die gebräuchlichste Form ist jeder Datenflussecke einen Grundblock zuzuordnen und die Grundblöcke untereinander durch Steuerflusskanten zu verbinden.

6.3.2.4 SSA

Eine Eigenschaft für Zwischendarstellungen ist in den letzten 15 Jahren sehr populär geworden: SSA[18] [CFR+91]. SSA ist *keine* Zwischendarstellung, sondern eine Eigenschaft, dem eine Zwischendarstellung genügen kann. SSA bedeutet, dass jede Variable im ganzen Programm nur eine Zuweisung besitzt. Das hat den großen Vorteil, dass man an jeder Stelle klar weiß, mit welchem Wert die Variable belegt ist. Ohne SSA ist diese Information nur implizit durch den Steuerfluss gegeben. Man denke zum Beispiel an eine Variable, der in einem `then`- und einem `else`-Zweig einer `if`-Anweisung ein unterschiedlicher Wert zugewiesen wird, wie das Beispiel weiter unten zeigt.

[18] engl.: *static single assignment*

Natürlich kann nicht jedes Programm die SSA-Bedingung erfüllen[19]. Folgendes Beispiel demonstriert dies:

```
1 ...
2 if(a > 0)
3   x = a;
4 else
5   x = 0;
6
7 y = f(x);
```

`x` wird einmal im `if`-Teil und einmal im `else`-Teil ein anderer Wert zugewiesen, also besitzt `x` zwei Zuweisungen, und das Programm ist nicht in SSA-Form. Es ist auch nicht klar, welchen Wert x nach dem `if`-Block hat. Hat es den Wert `a`, oder den Wert `0`?

Ein erster Schritt in Richtung SSA-Form ist es, bei jeder Zuweisung an eine Variable, eine Neue einzuführen, um sicher zu stellen, dass jede Variable nur eine Zuweisung besitzt. Dann sähe das obige Beispielprogramm so aus:

```
1 ...
2 if(a > 0)
3   x1 = a;
4 else
5   x2 = 0;
6
7 y = f(x);   // x ist jetzt undefiniert!
```

Jetzt ist noch offen, wie die Zeile `y = f(x)` hingeschrieben werden kann. Denn es gibt zwei Variablen für `x`: `x1` und `x2`. Dafür wird in der SSA-Darstellung die ϕ-Funktion eingeführt. Die ϕ-Funktion ist zunächst ein ganz normaler Operator, dessen *Stelligkeit* der Anzahl der Steuerflussvorgänger des Blockes entspricht, in der er sich befindet. Der Wert der ϕ-Funktion hängt von der Steuerflussvorgänger ab, über die der Grundblock erreicht wurde: Wurde der Grundblock über die n-te Kante betreten, so ist der Wert der ϕ-Funktion der des n-ten Arguments. Somit kann man obiges Beispiel vollständig in SSA-Form schreiben:

```
1 ...
2 if(a > 0)
3   x1 = a;
4 else
5   x2 = 0;
6
7 x3 = phi(x1, x2);
8 y = f(x3);
```

6.3.3 Zusammenfassung

Unterschiedliche Arten und Eigenschaften von Zwischendarstellungen wurden vorgestellt. Da ein Bild oft mehr sagt als tausend Worte, wollen wir an dieser Stelle ein C-Programm bringen, das von einem Frontend in eine moderne, graphbasierte Zwischendarstellung und in eine lineare Darstellung transformiert wurde.

[19] Allerdings kann jedes Programm in SSA-Form gebracht werden.

Wie man sieht berechnet folgendes Programm den ggT zweier Zahlen.

```
1  int gcd(int a, int b) {
2    while(a != b) {
3        if(a > b) {
4          a = a - b;
5        } else {
6          b = b - a;
7        }
8      }
9    return a;
10 }
```

Ein Übersetzer mit linearer Zwischendarstellung hätte wohl etwas folgender Art erstellt:

```
1      arg    t0 = 0
2      arg    t1 = 1
3  L1: sub    t2 = t0, t1
4      be     t2, L2
5      cmp    t3 = t0, t1
6      ble    t3, L3
7      sub    t0 = t0, t1
8      jmp    L4
9  L3: sub    t1 = t1, t0
10     jmp    L4
11 L4: jmp    L1
12 L2: ret    t0
```

Hier sieht man ganz deutlich, wie der Steuerfluss in Sprungmarken und Sprunganweisungen ausgedrückt ist. Diese Darstellung ist sehr umständlich, da man die Grundblöcke erst noch finden muss. Eine Information, die im AST implizit enthalten ist, geht hier verloren.

Des Weiteren hat man den Eindruck, dass schon Maschinensprache entsteht. In Wahrheit imitiert die Zwischensprache des Übersetzers nur eine bevorzugte Zielarchitektur. Das Backend wird später viel Aufwand treiben müssen, um die durch den AST suggerierte lineare Anordnung der Befehle in der Zwischendarstellung aufzulösen, und eine eigene effiziente Anordnung zu finden. Es ist eigentlich auch nicht einzusehen, die Information über die Grundblöcke, die im AST noch vorhanden war, zu verwerfen und sie später wieder aufzubauen.

Abbildung 6.5 zeigt einen kombinierten Steuerfluss und Datenabhängigkeitsgraphen, wie ihn die Bibliothek Firm [TLB99] erzeugt. Die „kleinen“ Ecken mit Namen wie etwa: `Start`, `Cmp`, `PhiIs`, ... stellen die Datenflussecken dar. Sie sind in große gelbe „Rechtecke“ eingeschlossen (mit den Nummern 26, 7, 11, 9, 20, 5, 3). Diese „Rechtecke“ sind die Grundblöcke. Durch die `Cond`- und `Jmp`-Ecken wird der Steuerfluss verzweigt. Deutlich zu sehen sind die ausgezeichneten Ecken des CFG (*Start* und *End*), die blau gefärbt sind und die Nummern 18 und 16 tragen. Die roten Kanten sind die Steuerflusskanten (sie zeigen hier – aufgrund der Konsistenz mit den Datenabhängigkeiten – in die umgekehrte Richtung). Die schwarzen Kanten sind Datenabhängigkeitskanten. Die blau gepunkteten Kanten sind Speicherabhängigkeitskanten. Wenn eine Kante gestrichelt (rot) ist, so stellt sie eine Steuerflusskante

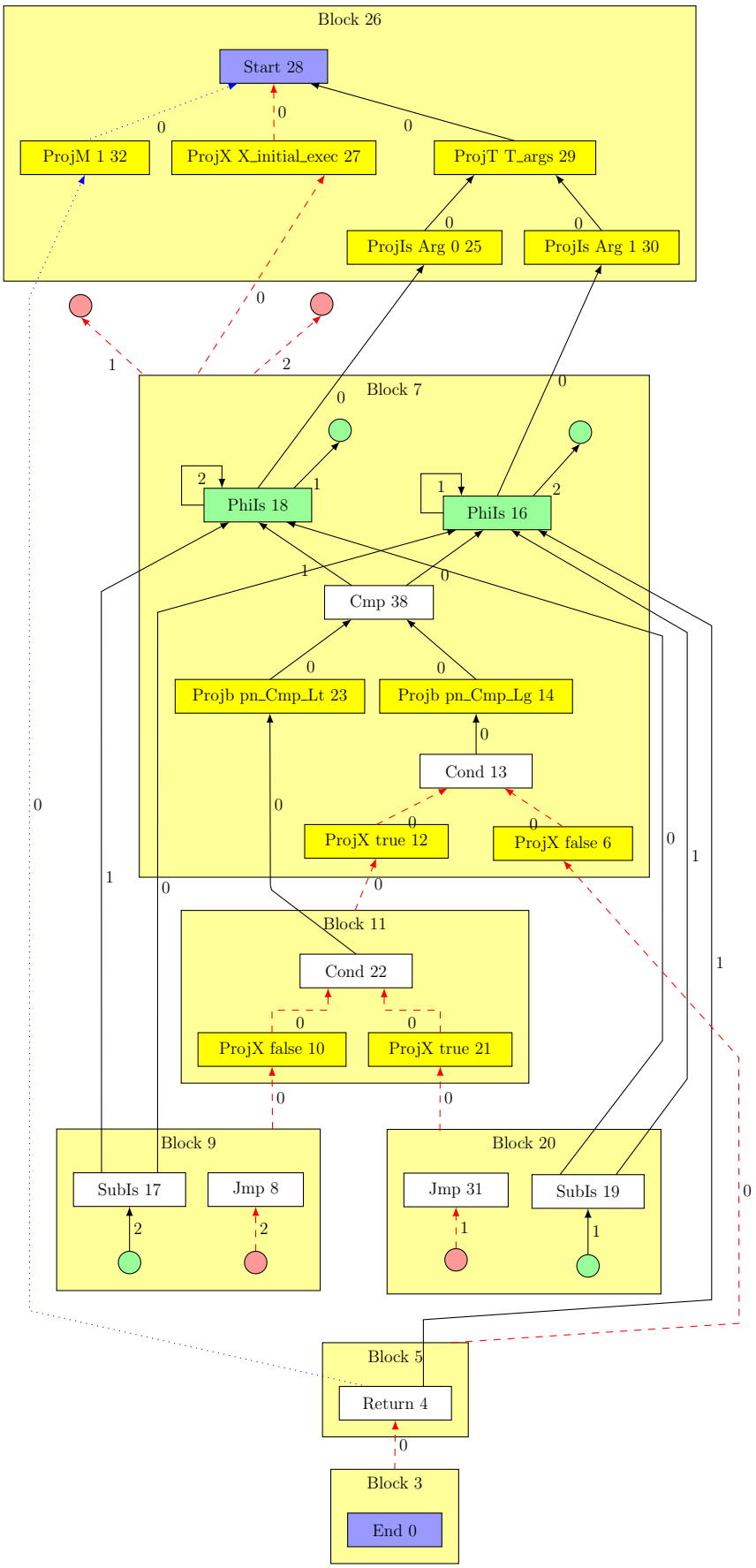


Abbildung 6.5: Das ggT-Programm in der Zwischendarstellung Firm

dar. Die Rückwärtskanten sind nicht eingezeichnet, sondern – der besseren Übersicht halber – durch kleine runde Ecken abstrahiert.

Schön sieht man auch die SSA-Eigenschaft von Firm: Im Schleifenkopf (Block mit der Nummer 7), werden `a` und `b` verglichen, da dieser Block aber mehrere Steuerflussvorgänger hat (nämlich 26, 9 und 20; sie entsprechen dem Block vor dem Schleifenkopf, dem `then`- und `else`-Zweig), und `a` und `b` in diesen Blöcken beschrieben werden, werden hier ϕ-Ecken eingesetzt.

6.3.4 Firm

Die graphbasierte Zwischendarstellung Firm repräsentiert Operationen (wie Addition, Laden aus dem Speicher, Sprung an eine Marke oder auch einen Funktionsaufruf) als Ecken. Daten- und Steuerflussabhängigkeiten werden durch Kanten dargestellt. Auch strukturbildende Elemente wie Grundblöcke werden als Ecken dargestellt. So ist jede Datenflussabhängigkeitsecke durch eine Grundblockkante an einen Grundblock gebunden. Sprungecken verbinden einen Grundblock mit einem anderen durch eine Steuerflusskante. Ferner existiert ein Eckentyp, der Datenfluss in Steuerfluss umsetzt und somit bedingte Verzweigung implementiert. Der Datenfluss, der über den Hauptspeicher stattfindet, wird durch eine Speicherkante modelliert, die vergleichbar zu einem „gewöhnlichen Wert“ den gesamten Hauptspeicherinhalt modelliert. Im Gegensatz zu dem eng verwandten „sea of nodes“ Ansatz von Click [CC95, CP95] gruppiert Firm alle Ecken in Grundblöcken.

Des Weiteren werden alle Typen und deren Instanzen in Firm dargestellt. Bestimmte Datenflussecken nehmen über spezielle Kanten Bezug auf die vorhandene Typinformation. Eine genauere Darstellung und Definition von Firm ist in [TLB99] zu finden.

Firm-Graphen haben einige Besonderheiten, die sie von allgemeinen Graphen unterscheiden. Zunächst sind Ecken und Kanten in Firm-Graphen typisiert (oder *markiert*). Ferner besitzen Ecken Attribute, die das dargestellte Objekt genauer beschreiben. Beispielsweise besitzen Ecken, die Funktionstypen darstellen, bestimmte Attribute, die den Funktionstyp genauer beschreiben. Überdies ist die Reihenfolge der Ausgangskanten einer Ecke bedeutungstragend, da nicht alle Ecken kommutative Operationen repräsentieren. Diese Eigenschaft impliziert Kantenattribute, wenn wir Firm-Graphen als markierte Multigraphen betrachten wollen (bei markierten Multigraphen können zwei Kanten desselben Typs die gleichen Ecken verbinden). So erhält jede Kante ein Attribut mit dem Wertebereich $\mathbb{N}$, das dann eine Totalordnung der Ausgangskanten für eine Ecke induziert.

Des Weiteren sind folgende Eigenschaften interessant: Als Daten- bzw. Steuerflussgraph hat ein Firm-Graph immer zwei ausgezeichnete Ecken *Start* und *End* mit $|inedges Start| = 0$ und $|outedges End| = 0$. Der Ausgangsgrad der Ecken ist bis auf wenige Ausnahmen für alle Firm-Graphen bekannt und konstant. So hat zum Beispiel die Ecke, die eine Addition darstellt, immer zwei ausgehende Kanten, niemals eine oder drei. Allein die *Call*-, *Return*- und ϕ-Ecken sowie Grundblöcke besitzen einen variablen Ausgangsgrad.

Wir haben nun gesehen, wie sich Übersetzer aufbauen und wie ihre zentrale Komponente, die Zwischendarstellung, beschaffen sein kann. Im folgenden Abschnitt gehen wir grob auf die Hardware ein, für die ein Übersetzer ggf. Zielprogramme erstellen muss.

6.4 Rechnerarchitekturen

Nachdem wir zunächst die klassische Rechnerklassifikation nach Flynn kritisch würdigen [Fly72], argumentieren wir, dass heutige Prozessoren (genauer ihre Befehle) einer neuen Klasse zugeordnet werden sollten – den *reichhaltigen Befehlen*.

6.4.1 Klassifikation nach Flynn

SISD (engl.: single instruction stream, single data stream)
: In diese Klasse fallen die traditionellen Einprozessor-Rechner: Sie führen zu einem gegebenen Zeitpunkt genau einen Befehl auf einem zu verarbeitenden Datum aus. Diese Klasse lässt jedoch überlappende Ausführung, wie sie beispielsweise durch Fließbandverarbeitung entstehen, ebenso außer Acht, wie Parallelität auf Ebene der Rechenwerke wie sie durch Hyper-Threading-Rechner erschlossen wird.

SIMD (engl.: single instruction stream, multiple data stream)
: In diese Klasse fallen Feldrechner mit ihren mehreren Recheneinheiten, die parallel auf verschiedenen Daten die gleiche Operation ausführen, sowie Vektorrechner, die quasi-parallel mehrere Daten durch von dem Befehlssatz steuerbare Fließbandverarbeitung bearbeiten. Vektorrechner und insbesondere Feldrechner können recht hohe Grade an durch die Hardware verfügbare Parallelität aufweisen, jedoch muss auch die Anwendung dieses hohe Maß an Parallelität hergeben; ansonsten liegen größte Teile der parallelen Einheiten brach.

MISD (engl.: multiple instruction stream, single data stream)
: Dies ist eine randständige Klasse, die von vielen als leere angesehen wird und heutzutage allenfalls von einiger Spezialhardware bevölkert wird.

MIMD (engl.: multiple instruction stream, multiple data stream)
: Dies ist die Klasse der „echten“ Parallelrechner, die mit mehreren eigenständigen Rechenwerken ausgestattet sind, die allerdings immer noch (wie auch immer) koordinierbar sind – mithin gemeinsam an einer Aufgaben arbeiten.

Das Flynnsche Schema stößt bereits bei der Klassifikation der seit über 20 Jahren verfügbaren multiskalaren Prozessoren [Int91], die mehrere Befehle gleichzeitig an verschiedene Funktionseinheiten zuordnen können, an seine Grenzen. Die Funktionseinheiten können der Klasse SISD zugeordnet werden, jedoch ist eine eindeutige Klassifikation des Gesamtprozessors nicht mehr unstrittig möglich.

In den klassischen Lehrbüchern (siehe u. a. [Wol95]) werden unter SIMD-Architekturen hauptsächlich feldrechnerartige Systeme verstanden. Die in den letzten Jahren immer häufiger anzutreffende Ausprägung der SIMD-Rechner als Einprozessorsystem (die eine vordefinierte zum Teil recht komplexe Folge von Befehlen gleichzeitig auf mehreren Daten ausführt) wird kaum beachtet. Genau dafür definieren wir die Klasse der Prozessoren mit *reichhaltigen Befehlen*, die eine Verallgemeinerung der SIMD-Klasse ist.

6.4.2 Reichhaltige Befehle (RIMD)

Klassische SIMD Prozessoren wenden einen Befehl auf mehrere Daten an. Diese Befehle sind in der Regel primitive Befehle im Sinn von *RISC* [HJBG81, HP06].

Heutzutage haben allerdings Prozessoren spezielle Befehle (SSE von Intel [Int07a] oder AltiVec von IBM/Motorola [Mot98] usw.), die eine kleines Programm, das auch Verzweigungen enthalten kann, auf mehrere Daten anwenden können. Zum Beispiel führt der `PSADBW`-Befehl von Intels SSE im Wesentlichen folgendes Programm auf bis zu 16 Daten aus:

```
1  int sad(int[] a, int[] b) {
2    int sum = 0;
3    int i;
4
5    for(i=0; i<16; i++) {
6      int d = a[i]-b[i];
7      if(d >= 0)
8        sum = sum + d;
9      else
10       sum = sum - d;
11   }
12   return sum;
13 }
```

Solcherlei „reichhaltige“ Befehle sind unserer Meinung nach nicht mehr als SIMD, sondern als neue Klasse[20] zu verstehen: RIMD (engl.: *rich instruction, multiple data*).

Befehlssätze moderner Prozessoren weisen eine Vielzahl an RIMD Operationen auf, die – häufig in Multimedia- oder wissenschaftlichen Anwendungen auftretende Algorithmen – beschleunigen können. Neben der Fähigkeit auf mehreren Werten gleichzeitig arbeiten zu können, implementieren diese Befehle z. B. *saturierte Arithmetik* [OK97, Ame95], Minimum oder Maximum zweier Zahlen, sodass diese Befehle nicht nur in Situationen interessant sind, in denen parallelisiert werden kann, sondern auch wenn sie nur auf einfachen Daten arbeiten. Da diese Befehle weder eine Entsprechung in üblichen Hochsprachen aufweisen, noch klassische Übersetzer diese Befehle effektiv verwenden können, liegt dieses beachtliche Potenzial zur Leistungssteigerung meist brach. Lediglich der Einsatz von vorgefertigten Bibliotheken bei der Programmerstellung oder die direkte Programmierung in einer Maschinensprache ermöglichen bis dato die umfassende Nutzung dieser reichhaltigen Befehle.

Um reichhaltige Befehle zu nutzen, ist es aus unserer Sicht nötig, Graphmuster in der Zwischendarstellung[21] zu finden, und das so gefundene Muster in einen anderen Graphen zu überführen. Ebendies beherrschen zeitgenössische Übersetzer, mit ihren Baumtransformationen und Pseudo-Graphersetzungstechniken, nicht auf eine systematische Weise. Der Stand der Kunst ist es, einige wenige Situationen durch – von Hand ausprogrammierte – ad hoc Optimierungen zu erkennen und geeignet umzusetzen. In Kapitel 7 und besonders in Abschnitt 7.4.2 werden wir uns dieser Problematik annehmen.

[20] In der Literatur hat sich SIMD- aber auch Vektor-Befehle als Bezeichnung dieser Klasse von Befehlen eingebürgert, wir glauben aber, dass dies ein Missbrauch der jeweiligen Begriffe ist, und schlagen darum RIMD vor.

[21] Im nachfolgenden Abschnitt 6.5 werden wir zwar auch sehen, dass es denkbar ist, die reichhaltigen Befehle auf dem AST oder gar auf Ebene der Quellsprache direkt zu suchen, allerdings sind diese Ansätze allesamt nur mäßig erfolgreich. Wir erörtern ebendort auch die Gründe dafür.

Eigenschaft	RIMD-Prozessor	Vektorrechner
Vektorlänge	4 ... 16	64 ... 1024
Registerklassen	normale oder spezielle	immer spezielle
Vektoreinheit verfügbar	z. T. explizite Umschaltung	immer
vektorisierbare int Typen	8, 16 Bit; selten 32	alle
vektorisierbare float Typen	32 Bit; selten 64	alle
Issue-Granularität	atomar, über Fließband	chaining
Sprachen	C/C++, Java, C#	Fortran, modifiziertes C
Feld- und Schleifengrenzen	dynamisch & fest	fest (F77, [Ame78])
Programmdomäne	Multimedia, Spiele, etc.	Numerik
Vektoren speichern/laden	normale MMU/LS-Einheit	spezielle Behandlung

Tabelle 6.1: Vergleich von RIMD-Prozessoren und Vektorrechnern

6.5 Verwandte Arbeiten

Die Literatur zu Ansätzen im Bereich klassischer SIMD- und Vektor-Optimierungen ist sehr zahlreich und geht bis auf die Anfänge der Hochsprachen, nämlich den ersten Fortran-Übersetzer aus dem Jahr 1957 zurück [BBB+57]. Im Folgenden besprechen wir die wichtigsten Arbeiten in diesen Bereichen sowie erste neuere Arbeiten für reichhaltige Befehle.

6.5.1 Vektorisierertechnologie und RIMD-Prozessoren

Die vektorisierenden Übersetzer sind seit über 30 Jahren ein aktives Forschungsthema. Sie implementieren die klassischen SIMD-Optimierungen. Eine der herausragendsten Monografien des Übersetzerbaus: „Optimizing Compiler for Modern Architectures—A Dependence-Based Approach“ von Allen und Kennedy behandelt diese Fragestellungen [AK02]. Ungeachtet des dort erreichten hohen Standes der Forschung sind die Arbeiten zumindest nicht unmittelbar auf unser Problem (RIMD-Befehle) anwendbar. Tabelle 6.1 stellt die unterschiedlichen Charakteristika gegenüber.

Schon sprachlich wird der Hauptunterschied klar: man spricht von Vektorrechnern aber von RIMD-Prozessoren. Dies liegt daran, dass Vektorprozessoren ausschließlich in Vektorrechnern Verwendung finden, da sie ausschließlich in besonderen Rechnern funktionieren. Zum Beispiel muss das gesamte Speichersubsystem die Fließbandverarbeitung von Speicherzugriffen mit automatischer Adressfortschaltung hardwareseitig unterstützen, sonst können nicht in jedem Takt eine oder gar mehrere fortlaufende Speicherstellen geladen und geschrieben werden. RIMD-Prozessoren sind in der Regel normale Prozessoren, die einen erweiterten Befehlssatz haben. Die Verarbeitung jener Befehle geschieht komplett im Prozessor; besondere Anforderungen an den Rechner werden nicht gestellt.

Aus Tabelle 6.1 ergeben sich folgende Punkte, die beim Bau von Übersetzern für automatische RIMD-isierung im Gegensatz zu vektorisierenden Übersetzern beachten sind.

- Wenn Codestücke überhaupt zu vektorisieren ist, dann lohnt es sich auf jeden Fall. Das ist bei RIMD-Befehlen nicht notwendigerweise so. Insbesondere wenn

wir unvollständig ausgenutzte RIMD-Befehle betrachten, z. B. also nur Vektoren der Länge zwei oder drei anstatt der möglichen vier verarbeitbaren. Um diese Situation zu lösen, muss man ein geeignetes Kostenmodell verwenden. Das Gleiche gilt beim Umschalten zwischen Registersätzen oder Befehlsarten des Prozessors.

- *Bitbreitenanalyse* und *Typeinschränkung* gewinnen erheblich an Bedeutung, da durch diese Analysen die Parallelität unter Umständen erheblich gesteigert werden kann. In Abschnitt 7.2.2 gehen wir auf entsprechende Optimierungen ein.

- In Fortran sind Schleifen- und Feldgrenzen traditionell oft fest gewählt, obwohl das in modernem Fortran eigentlich nicht mehr nötig ist. Durch einen solchen Programmierstil und der Zeigerlosigkeit von Fortran sind viele Analysen einfacher oder überhaupt nicht nötig. Insbesondere sind Optimierungen, die quasi auf dem Quellprogramm arbeiten, nur bei Fortran oder ähnlich strukturierten Sprachen von Erfolg gekrönt. Im Falle von C ist die Darstellung der Quelle viel zu wenig normalisiert um irgendetwas erkennen zu können, Optimierungen sollten hier auf der Zwischendarstellung aufsetzen.

- Falls die Werte nicht in der richtigen Reihenfolge im Speicher liegen, oder falls sie in skalaren Registersätzen vorliegen, müssen die Werte möglicherweise umkopiert werden, um die Daten mit reichhaltigen Befehlen zu verarbeiten. Wenn die Verarbeitung abgeschlossen ist, müssen ggf. die Daten wieder zurückkopiert werden. Diese Vorgänge reduzieren aber den Nutzwert des reichhaltigen Befehles, sodass auch hier eine Kostensteuerung nötig ist.

- Der Speicher, der von RIMD-Befehlen verarbeitet wird, muss fast immer *ausgerichtet* sein. Sonst ist entweder der Befehl unzulässig oder wird sehr viel langsamer ausgeführt. Dieser Zustand kann auf folgende Weise erreicht werden: Es wird zunächst überprüft, welche Speicherbereiche auf jeden Fall ausgerichtet sind (z. B. lokale Variablen), wenn der Zugriff auf diesen Speicher ebenfalls ausgerichtet stattfindet, muss nichts getan werden. Falls wir Speicher unbekannter – oder wegen der Alias-Problematik zweifelhafter – Herkunft finden, dann können wir für jeden dieser Bereiche zwei Codevarianten erzeugen, die eine enthält den optimierten Code, die andere behandelt alles wie gehabt. Dies setzt voraus, dass der optimierte Code auch tatsächlich einen erheblichen Vorteil bietet, weil dieser z. B. in einer Schleife ausgeführt wird. Zur Laufzeit wird außerhalb der berechnungsintensiven Schleife durch eine einfache Modulo-Operation entschieden, ob der eine oder andere Code ausgeführt wird. Weitere Ansätze sind zum Beispiel in der Arbeit von Eichenberger et al. zu finden [EWO04].

6.5.2 Ausgewählte Arbeiten

Nach dieser einleitenden Übersicht zum Stand der Forschung betrachten wir einzelne ausgewählte Arbeiten, die sich mit reichhaltigen Befehlen beschäftigen. Quantitativ gesehen handelt es sich bei den meisten Arbeiten um exemplarische Analysen der Leistungsfähigkeit von reichhaltigen Befehlen beziehungsweise um die Beschleunigung einzelner Algorithmen durch manuelle Verfahren. Erst in den letzten fünf Jahren gibt es mehr Arbeiten zu automatischen Verfahren. Insgesamt sind nur

zwei Arbeiten bekannt, die mit deklarativer Mustersuche an das Problem herangehen [MKC00, MW00]. Der Ansatz von Metzger und Wen [MW00] ist dabei unserem Ansatz am ähnlichsten (vgl. dazu Abschnitt 7.4.2).

6.5.2.1 Verfügbare Übersetzer

Die praktisch verfügbaren Übersetzer beherrschen das automatische Ausnutzen von reichhaltigen Befehlen – wenn überhaupt – nur fragmentarisch. Der einzige Übersetzer, der in nennenswertem Umfang reichhaltige Befehle selbsttätig in den Code einbaut ist der ICC von Intel [Int07b]. Allerdings veröffentlicht Intel keine Verfahren und auch nicht die Implementierung. Eine Analyse des Verhaltens und nicht zuletzt der von Intel verwendete Name *auto-vectorization* lässt darauf schließen, dass dieser Übersetzer auf Vektorisierertechnologie aus der Fortran-Ära aufbaut. Diese sehr am AST orientierten Verfahren können in der Regel ausschließlich Schleifen optimieren und versagen schon bei geringsten Abweichungen, wie zum Beispiel dem Umdrehen einer Schleifenbedingung.

6.5.2.2 Manuelle Beschleunigung eines Algorithmus

Der Aufsatz *„Reducing 3D Fast Wavelet Transform Execution Time Using Blocking and the Streaming SIMD Extensions“* von Bernabé et al. ist exemplarisch für Arbeiten, die einzelne Algorithmen mithilfe von reichhaltigen Befehlen beschleunigen. Diesen Arbeiten ist gemeinsam, dass zentrale datenparallele Aufgaben auf die eine oder andere Weise durch RIMD-Befehle erledigt werden. Je nach konkreter Aufgabe wird dazu der Algorithmus umgeschrieben, sodass zum Beispiel die Laufreihenfolge durch die Felder oder die Datenanordnung eine andere ist. Manchmal werden auch noch weitere Optimierungen durchgeführt, die wie zum Beispiel bei Bernabé die Leistung des *Caches* verbessern sollen und gleichzeitig der *Ausrichtung*[22] der Daten im Speicher für die reichhaltigen Befehle dienen. Üblicherweise werden von diesen Arbeiten Leistungssteigerungen um den Faktor 5 bis 6 erreicht, einzelne Arbeiten berichten auch von Faktoren zwischen 1,2 und über 10.

Die Arbeit von Bernabé et al. ist aus einem weiteren Grund bemerkenswert: Die Autoren erkennen, dass ein leichterer Ansatz für den Einsatz von RIMD-Befehlen nicht das Vektorisieren, sondern vielmehr das Ausrollen von Schleifen ist. Dadurch liegen die Befehle direkt vor und können so viel leichter gefunden werden – ganz ohne klassische Vektorisierertechnologie. Bernabé zeigt in diesem Zusammenhang, dass eben der Intel-C++-Übersetzer dies nicht beherrscht: Lässt man ihn den voroptimierten Code direkt übersetzen, ist dies langsamer, als wenn gar keine Optimierung stattfindet!

6.5.2.3 Manuell implementierte Mustersuche

PDG basierte Idiomerkennung

Die Optimierung von Pinter und Pinter aus dem Jahr 1994 ist eine Pionierarbeit, da sie reichhaltige Befehle ins Spiel bringt, noch bevor die Hardwarehersteller diese breitflächig eingesetzt haben [PP94]. Ebenfalls beachtlich ist die Idee, den PDG, eine Art Abhängigkeitsgraph, zu verwenden, um, wie Pinter es nennt, *Idiomerkennung* zu

[22] engl.: *alignment*

betreiben. Bis dahin waren vor allem Arbeiten auf nicht graphbasierten Zwischendarstellungen (vor allem aus dem Fortran-Umfeld) gemacht worden. Andererseits lösen sie sich nicht von Fortran als Quellsprache, mit der damit verbundenen Vereinfachung der eigentlichen Aufgabe. Sie schlagen eine neuartige Darstellung, nämlich Berechnungsgraphen, vor, mit denen sie dann innere Schleifen von Programmen darstellen. Diese Berechnungsgraphen enthalten insbesondere alle schleifengetragenen Abhängigkeiten und erlauben so die klassische Vektorisierertechnologie anzuwenden, aber eben mit dem Vorteil, dass die Darstellung frei von Anweisungsreihenfolge und Ähnlichem der sonst üblichen, an den AST angelehnten Darstellungen ist. Allerdings wird nicht die gesamte Prozedur in einer graphbasierten Zwischendarstellung repräsentiert, sodass übergreifende Muster, wie zum Beispiel der PSADBW-Befehl von Intels SSE, nicht handhabbar sind.

Befehlsauswahl und reichhaltige Befehle

Leupers und Bashford stellen die Auswahl von Befehlen, die wir als reichhaltige Befehle ansehen, als Problem der Befehlsauswahl dar [BL99, Bas00, LB00]. Ihrer Meinung nach ist es besonders bei eingebetteten Systemen statthaft, einen langwierigen Übersetzungsvorgang zu haben, wenn es so möglich ist, bessere Codequalität zu erzielen. Sie bilden das Befehlsauswahlproblem in Datenflussgraphen auf ein Constraint Satisfaction Programming (CSP) ab, wobei der Problemraum auf einzelne Grundblöcke beschränkt bleibt. Dadurch können die meisten echten reichhaltigen Befehle sowieso nicht behandelt werden. Allerdings gelingt es Leupers und Bashford durch die Reduktion auf CSP optimale Befehlsauswahl für Grundblöcke auf DAGs, d. h. unter Anwesenheit von Mehrfachnutzung von Werten zu betreiben.

Jedoch ist der Preis, den sie dafür zahlen extrem hoch. Das größte Beispiel, das sie betrachten, ist 95 DFG Ecken groß und benötigt 26,5 Sekunden, um gelöst zu werden. Darüberhinaus weisen Arbeiten von Kästner und Menne [Käs97, Men99] auf prohibitiven Laufzeiten dieser Verfahren bei selbst kleinen, realitätsbezogenen Aufgaben hin, die solche Ansätze als praktisch nicht brauchbar erscheinen lassen.

Allem Anschein nach wächst die Komplexität der Lösungsfindung exponentiell mit der Anzahl der Ecken im Grundblock, sodass realistischer Code (vor allem falls dieser ausgerollt ist oder offener Einbau[23] vorliegt) kaum zu verarbeiten ist. Die Arbeit von Eckstein et al. [EKS03, ES03, SE02], die das erste *wirklich* auf Graphen (nicht auf Grundblock DAGs) arbeitende Verfahren zur Befehlsauswahl darstellt, stellt auch die Arbeit von Leupers und Bashford in den Schatten. Die PBQP-Lösung ist approximativ oder exakt bestimmbar, selbst wenn man die exakte Lösung anstrebt, kann dies meist sehr effizient (also linear) geschehen. Praktische Erfahrungen mit diesem Verfahren zeigen, dass selbst in großen Graphen (> 20.000 Ecken) die Lösungen in weniger als einer Sekunde gefunden werden können, da die Heuristik fast immer exakt ist, also nur an wenigen ausgewählten Stellen enumerative Verfahren nötig sind.

Vektorisieren von Schleifen

Der Aufsatz von Sreraman und Govindarajan präsentiert einen klassischen Ansatz zum Vektorisieren auf AST-Niveau [SG00]. Ihr Ansatz benutzt den AST des Übersetzerbaukastens SUIF, und macht auf diesem die klassischen Abhängigkeitsanalysen

[23] engl.: *inlining*

für Feldzugriffe in Schleifen. Zuvor werden die Schleifen noch normalisiert und gegebenenfalls geschält und verteilt. Auf dieser Basis läuft dann eine handprogrammierte Erkennung einiger spezieller Schleifen ab. Bei der Befehlserzeugung aus dem AST heraus wird für fast alle Teile C-Code ausgegeben, nur für die erkannten reichhaltigen Befehle werden Assemblereinschübe genutzt.

Ausrollen statt Vektorisieren von Schleifen

Krall und Lelait schlagen eine sehr einfache Lösung für das gemeinhin als kompliziert erachtete Problem der Abhängigkeitsanalyse von Schleifen im Zusammenhang mit RIMD-Befehlen vor: Eine Abhängigkeitsanalyse über Schleifen sollte schlicht nicht gemacht werden, statt dessen ist der Code entsprechend dem Grad an Datenparallelität des RIMD-Befehls auszurollen. Einerseits fällt auf, dass diese Lösung genial einfach, aber dennoch bis zur Veröffentlichung jener Arbeit noch nicht ausgesprochen war. Andererseits wird klar, warum selbst heutzutage Übersetzer ebendies nicht tun: Bei Vektorrechnern geht das nicht (Vektoren der Länge 1024 oder größer) – also tun sie es auch sonst nicht. Krall und Lelait zeigen, wie man in der ausgerollten linearen Zwischendarstellung ganz einfach die Befehle durch handprogrammierte Mustersuche finden kann, und gehen überdies auf nicht angeordneten Speicher ein.

Ausrichtung von Speicherzugriffen

Die Arbeit mit dem Titel *„Vectorization for SIMD architectures with alignment constraints“* von Eichenberger et al. nimmt einen wichtigen Teilaspekt der reichhaltigen Befehle unter die Lupe: Die *Ausrichtung* der Speicherzugriffe. Die meisten Prozessoren mit RIMD-Befehlen können nämlich nur an 8 oder 16 Byte ausgerichtete Adressen effizient laden, manche Prozessoren weisen solche Speicherzugriffe sogar als ungültige Befehle zurück. Die von Eichenberger verwendete Technik propagiert die jeweilige Ausrichtung der Adressen durch die Ausdrucksbäume, sodass sie überall zu Verfügung stehen. Zusätzliche Zugriffe an den Rändern von Speicherbereichen und Korrekturen mittels Maskieren und Schieben werden so nur an den notwendigen Stellen im Code plaziert.

6.5.2.4 Verfahren mit deklarativer Mustersuche

Mustersuche auf dem AST

Manniesing, Karkowski und Corporaal beschreiben ein Verfahren, das RIMD-Befehle auf dem AST sucht. Allerdings ist dieses Verfahren nur prototypisch ausgearbeitet, da sie im Wesentlichen nur Stellen im Programm finden, die durch RIMD-Befehle optimiert werden *könnten* – ob diese Optimierungen im vorliegenden Fall aber tatsächlich legal wären und wie die tatsächliche Transformation geschehen könnte, bleibt weitgehend offen. Sie haben mit verschiedenen Problemen zu kämpfen, die durch die Entscheidung bedingt sind, auf dem AST zu arbeiten. Immerhin konnten sie so mit recht geringem Aufwand mittels eines Baummustersuchers zeigen, dass es in üblichen Leistungstests aus dem Audio- und Videobereich wahrscheinlich viele Einsatzmöglichkeiten für reichhaltige Befehle gibt.

Mustersuche auf einer baumbasierten Zwischendarstellung

Metzger und Wen stellen einen ausgearbeiteten Ansatz vor, der ganze Algorithmen – insbesondere die rechenintensiven Kerne[24] eines Programms – auffinden kann und diesen durch einen einzigen Aufruf an eine Bibliothek ersetzt, die solche Kerne in handoptimierter Fassung enthält [MW00]. Dadurch soll insbesondere das sonst sehr kostspielige Trimmen von Leistungstests durch den Hardwarehersteller leichter von statten gehen, da der Übersetzer nun leicht um weitere optimal beherrschte rechenintensive Kerne erweitert werden kann. Die Optimierung baut auf der Zwischendarstellung des *Convex Application Compiler* [BMS92] auf, die Baumdarstellung aufweist. Als Datenstrukturen werden der Steuerbaum[25] verwendet, der die Anweisungen und den Steuerfluss enthält. Die Ausdrücke selbst werden in üblichen Ausdrucksbäumen dargestellt, wobei zusätzlich der *i-val*-Baum aufgebaut wird, der die Abhängigkeiten der Induktionsvariablen enthält. Auf Anweisungsebene wird zusätzlich der Datenflussgraph berechnet. Um die Mustersuche zu beschleunigen, werden der Steuerbaum, die Ausdrucksbäume, und der *i-val*-Baum in eine kanonische Form überführt. Dabei werden besonders kommutative Operatoren normalisiert. Die Sortierung wird durch eine Nummerierung der Ecken im Steuerbaum induziert. Metzger und Wen erklären auch, wie man die Kerne aus Programmen extrahiert und wie die Auswahl der bestmöglichen Ersetzung aussehen kann.

Unser in Abschnitt 7.4.2 vorgestellter Ansatz teilt mit diesem Ansatz mehrere wesentliche Ideen: Beide verwenden eine Datenbank von a priori erzeugten Mustern, die zu suchen und ersetzen sind. Beide haben die Idee diese Muster nicht selbst anzugeben, sondern in der Programmiersprache, die optimiert werden soll, während eines speziellen Übersetzungslaufes zu spezifizieren. Dadurch ist es für den Endbenutzer (aber auch für Übersetzerbauer) einfach neue Optimierungsteile hinzuzufügen, ohne den Übersetzer selbst zu modifizieren. Beide verwenden auch *Standardoptimierungen*, um die Variationen in der Zwischendarstellung zu minimieren. Selbstverständlich versuchen beide Ansätze, auch eine möglichst gute Ersetzung unter allen möglichen Ersetzungen auszuwählen.

Allerdings gibt es auch Unterschiede: Wir verwenden eine graphbasierte Zwischendarstellung mit integriertem Steuerflussgraphen, was uns von zahlreichen Aufgaben der Konsistenzhaltung befreit. Metzger und Wen müssen, da sie auf Anweisungsebene arbeiten, auch Methoden zum Umgang mit Umordnungen in den Anweisungslisten entwickeln; bei unserer graphbasierten Zwischendarstellung erübrigt sich dies. Wir benutzen ein Graphersetzungswerkzeug, um die Optimierung auszuführen. Unser suchplanbasierter Ansatz macht dabei eine Transformation des gesamten zu optimierenden Programmes in eine kanonische Darstellung überflüssig, da die Mustersuche auch ohne eine solche schnell vonstattengeht. Schließlich haben Metzger und Wen nur den Normalisierungsprozess implementiert, die eigentliche Mustersuche und Ersetzungen fehlt, sodass insbesondere keine praktischen Ergebnisse vorliegen.

[24] engl.: *computations kernel*
[25] engl.: *control tree*

KAPITEL 7

GRAPHERSETZUNG UND ÜBERSETZERBAU

Die Umsetzung des Leitgedankens dieser Arbeit im Bezug auf den Übersetzerbau, nämlich die bestmögliche Nutzung von Maschineneigenschaften bei gleichzeitig personaleffizienter Konstruktion von Übersetzern, erfordert ein angemessenes Abstraktionsniveau beim Entwerfen und Implementieren. In Abschnitt 7.1 beschäftigen wir uns damit, wie dieses Abstraktionsniveau herzustellen ist. Die beiden nachfolgenden Abschnitte legen auf theoretischem Niveau fest, wie wir erstmalig Graphersetzung und Übersetzerbau miteinander verbinden. Abschnitt 7.2 stellt einige Voraussetzungen für den Einsatz von Graphersetzung im Übersetzerbau dar. Wie die Zwischendarstellung als Graphmodell zu repräsentieren ist, wird in Abschnitt 7.3 behandelt. Mit den von uns entwickelten *natürlichen Spezifikationen* zeigen wir schließlich in Abschnitt 7.4.2 eine neuartige Optimierung im Übersetzerbau, die ohne effiziente integrierte Graphersetzung praktisch nicht zu realisieren ist.

7.1 Abstraktionen

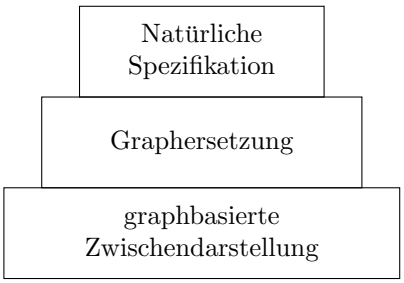


Abbildung 7.1: Abstraktionspyramide der Basisarchitektur

Als Grundlage verwenden wir eine graphbasierte Zwischendarstellung, die alle Datenabhängigkeiten und den Steuerfluss so explizit wie möglich darstellt. Somit sind Optimierungen konzise und effizient beschreibbar.

Darauf bauen wir eine zweite Abstraktionsschicht, die im Wesentlichen Graphmanipulation und insbesondere ein deklaratives Graphersetzungssystem bereitstellt. Auf dieser Grundlage erzielen wir mittels verschiedener Ansätze Verbesserungen bei der Konstruktion von Übersetzern. Zum einen kann eine Sequenz von Graphersetzungen als eigene Phase im Übersetzer verwendet werden. Zum anderen ist es darüber hinaus möglich, bei der Implementierung von Optimierungen das Finden von

Mustern nicht mehr selbst in einer konventionellen Hochsprache[1] zu implementieren, sondern dies deklarativ via Graphersetzungssystem zu erledigen. Noch weiter reicht der Ansatz, eine anwendungsspezifische Hochsprache für den Übersetzerbau zu entwickeln und auf diese Weise die Integration von deklarativer Graphersetzung, mathematischen Konstrukten und konventionellen Hochsprachenkonstrukten zu verbessern.

Eine dritte Abstraktionsschicht, welche die *natürliche Spezifikation* von reichhaltigen Befehlssätzen erlaubt, schließt unsere Architektur ab. Der Kern dieser Spezifikationstechnik besteht darin, die Wirkung von reichhaltigen Befehlen in den Termini einer konventionellen Hochsprache zu beschreiben. Die natürliche Spezifikation erlaubt uns somit den nötigen personellen Aufwand bei der Adaption an neue Zielarchitekturen zu verringern. Gleichzeitig wird durch die Verwendung einer besonders einsichtigen Spezifikationsmethode und einer anschließenden automatischen Verarbeitung dieser Spezifikationen die Fehleranfälligkeit reduziert.

7.2 Integration der Graphersetzung

Die zentrale Datenstruktur für unseren Übersetzer ist die Zwischendarstellung Firm[2] (siehe auch Abschnitt 6.3.4 und [TLB99]). Auf ihr arbeiten alle Optimierungen inklusive des Graphersetzers. Allerdings treiben wir die Beibehaltung der graphbasierten Zwischendarstellung mit SSA-Eigenschaft weiter als üblich voran: Sie bleibt selbst nach der Befehlsauswahl erhalten [Jak04]. Die Befehlsanordnung fixiert lediglich die Berechnungsecken und materialisiert somit *eine* topologische Sortierung des ursprünglichen Graphen. Die Registerzuteilung geht schließlich auch neue Wege und arbeitet auf diesen angeordneten Zwischendarstellungsgraphen [HGG06].

In diesem Abschnitt betrachten wir einige Probleme, die bei der Integration von Graphersetzern in Übersetzern auftreten, und zeigen, wie diese gelöst werden können.

7.2.1 Essenzielle Programmabhängigkeiten

> „Firm ist eine graphbasierte Zwischendarstellung mit SSA-Eigenschaft, die nur essenzielle Programmabhängigkeiten darstellt, wobei essenziell bedeutet, das mit weniger Informationen eine korrekte Übersetzung im Allgemeinen nicht mehr möglich ist“.

Die vorangehende Aussage wird von Trapp [Tra01] gemacht und scheint zunächst gerechtfertigt. Denn es wird nur der Daten- und Steuerfluss dargestellt – alle willkürlichen Anordnungen des Codes wie sie von der Quelle oder dem Übersetzungsprozess herrühren scheinen so aufgelöst. Leider stimmt diese Aussage nicht in voller Allgemeinheit: Kommen zum Beispiel zwei Ganzzahladditionen `a+b+c` in einem Ausdruck der Quelle vor, so könnte einer der folgenden Bäume `(a+b)+c`, `a+(b+c)`, `b+(a+c)`, `b+(c+a)` usw. aufgebaut werden. Allerdings „normalisiert“ eine induzierte Ordnung auf den Operanden die Ausdrucksbäume immer so, dass bei denselben Operanden

[1] Wir betrachten in diesem Zusammenhang imperative Sprachen wie C, Java, Pascal als konventionelle Hochsprachen.

[2] Wir wollen hier die Zwischendarstellung Firm und ihre Implementierung libfirm synonym benutzen.

derselbe Ausdrucksbaum entsteht. Nehmen wir also der Einfachheit halber an, dass die Ordnung lexikografisch sei, dadurch also immer die Form `(a+b)+c` entsteht. Nehmen wir ferner an, dass eine zweite Ganzzahladditionen `b+c+d` verarbeitet werden soll, dann entsteht aufgrund der gleichen induzierte Ordnung `b+(c+d)`. Damit ist Folgendes geschehen: Der gemeinsame Teilterm `b+c` wird nicht erkannt! Semantisch Gleiches wird also nicht syntaktisch gleich dargestellt. Mithin ist die Darstellung offenbar nicht nur eine Essenzielle, denn die Ausdrucksbäume der Additionsoperationen werden im Firm-Graphen in ausgeklammerter Form dargestellt, wodurch eine Anordnung der Operationen impliziert wird, jedoch ist diese für eine korrekte Übersetzung nicht nötig – Additionsoperationen[3] auf Ganzzahlen sind kommutativ und assoziativ.

Firm ist also eher eine Zwischendarstellung, die *relativ wenige* nicht essenzielle Programmabhängigkeiten beinhaltet. Beim Umgang mit Zwischendarstellungen sollte man also im Auge behalten, dass sie eben nicht immer semantische Äquivalenz syntaktisch ausdrücken können. Wir werden in Abschnitt 7.4.2.5 sehen, wie man auch obige nicht essenzielle Abhängigkeit bei der Darstellung von Ausdrucksbäumen vermeidet.

7.2.2 Bitbreitenanalyse und Typeinschränkung

Entscheidend für sehr viele Optimierungen, aber vor allem für die natürlichen Spezifikationen, die zur besseren, benutzertransparenten Ausnutzung von RIMD-Befehlen (vgl. Abschnitt 6.4.2) eingesetzt werden, ist es zu wissen, wie groß der Wert einer gewissen Variable (oder das Berechnungsergebnis einer Zwischendarstellungsecke) werden kann. Mit dieser Information ist es möglich, z. B. *Überlaufprüfungen* wegzulassen, aber insbesondere kann der Typ der Variable eingeschränkt werden: Werden von einer 32-bittigen Ganzzahlvariable nur immer höchstens 8 Bit benutzt, dann kann diese Variable vom Übersetzer intern als 8-bittige Ganzzahlvariable umdeklariert werden, wodurch diverse Optimierungen möglich werden. In Bezug auf RIMD-Befehle kann jetzt eine Befehlsvariante mit höherer Parallelität gewählt werden. War vorher nur die Verarbeitung mit 32 Bit möglich, was bei 128 Bit RIMD-Datenwortbreite eine vierfach parallele Verarbeitung bedeutet, ist jetzt 16-fache Parallelität gegeben. Diese *Bitbreitenanalysen* müssen offenbar sichere Analysen sein, da sonst bei entsprechenden Transformationen der Zwischendarstellung die Programmsemantik zerstört würde. Damit ist leider auch ein recht hoher Aufwand nötig, um diese Information zu beschaffen; mehr noch: Die exakte, d. h. nicht zu konservative und dennoch fehlerfreie, Bestimmung des Wertebereichs (oder im speziellen der Bitbreite) einer Variable ist in Programmen i. A. nicht berechenbar. Somit kann nur mit heuristischen Verfahren gearbeitet werden. Es gibt aus den vergangenen drei Jahrzehnten zahlreiche Ansätze hierzu. Wir wollen hier nur einige neuere Ansätze akzentuiert darstellen.

Natürlich kann diese Information auch explizit vom Benutzer angegeben worden sein, indem er den – auf die Bitbreite bezogenen – genau passenden Typ im Quellprogramm verwendet hat. Allerdings zeigt die Erfahrung, dass in der Regel die Programmierer eher zum generischen `int`-Typ greifen, als sich genau zu überle-

[3] Diese Aussage ist von der Semantik der Programmiersprache abhängig, gilt aber zumindest für C. Bei anderen Programmiersprachen ist die Additionsoperation möglicherweise nicht assoziativ, da so Probleme beim Überlauf vermieden werden können.

gen, welcher Typ angemessen wäre. Fernerhin kann durch geschlossene Übersetzung, offenes Einsetzen oder Variantenbildung eine Variable, die an sich eine größere Bitbreite benötigt, in diesem speziellen Übersetzungskontext mit weniger auskommen. Leider besitzt Firm, obgleich das Verfahren von Trapp [Tra01] dies eigentlich als Nebeneffekt leisten sollte[4], keine praktisch funktionierende Bitbreitenanalyse. Darum stellen wir im Rest dieses Abschnitts entsprechende Verfahren aus der Literatur vor, die in Firm integriert werden könnten. Insgesamt benötigt man zur eleganten, benutzertransparenten Ausnutzung von RIMD-Befehlen also automatische Analysen der Bitbreite. Wenn die Quellprogramme entsprechend implementiert wurden, kann allerdings darauf verzichtet werden.

Bitwise ist ein Übersetzer von Stephenson et al., der die Bitbreite, also die Zahl der Bits, die benötigt werden, um eine Ganzzahlvariable oder einen Zeiger darzustellen, minimiert [SBA00]. Dieser Übersetzer benutzt hierzu Techniken der Datenflussanalyse und der abstrakten Interpretation [NNH99]. Die Datenflussanalysen werden hier alternierend vorwärts und rückwärts betrieben, um die berechneten Schranken in beide Richtungen zu transportieren. Bitwise benutzt zusätzlich zu den normalen Datenflussanalysen eine Analyse des Schleifenbereichs, die zuerst die Schleifen klassifiziert, dann eine geschlossene Form für Induktionsvariable und die Schleifenbedingung bestimmt, und diese schließlich löst, was natürlich nicht immer gelingt. Wenn die geschlossene Form gefunden und gelöst werden konnte, wird dadurch eine Fixpunktiteration an dieser Schleife überflüssig. Falls dies nicht gelungen ist, muss die Schleife per Fixpunktiteration behandelt werden, diese wird nach einer vom Benutzer vorgegebenen Iterationsanzahl abgebrochen. Obgleich das eigentliche Ziel von Stephenson im Benutzen dieser Ergebnisse für die optimierte Synthese von (rekonfigurierbarer) Hardware liegt, lassen sich diese Ergebnisse auch in normalen Übersetzern verwenden um *Typeinschränkung* durchzuführen.

Der BitValue Übersetzer von Budiu et al. [BSWG00] erkennt unnötig berechnete und gespeicherte Anteile eines Datenworts, ebenfalls basierend auf Techniken der Datenflussanalyse und der abstrakten Interpretation. Budius Ansatz unterscheidet sich vom Bitwise-Verfahren vor allem dadurch, dass er für jedes Bit in einem Datenwort berechnet ob es je benötigt wird oder nicht, wo hingegen Bitwise nur obere und untere Schranken des Wertes berechnet. Die Schleifenanalyse ist weniger elaboriert als bei Bitwise, aber da andere Transferfunktionen und Traversierungen gewählt wurden, lässt sich ein direkter analytischer Vergleich nicht bewerkstelligen.

Rugina und Rinard bilden das Problem der Bitbreitenanalyse auf das Lösen ganzzahliger linearer Programme[5] ab [RR05]. Rugina und Rinard gehen bei der Abbildung auf ILP zweistufig vor: Zuerst wird ein polynomielles Programm aus dem Daten- und Steuerflussgraphen extrahiert, das dann im zweiten Schritt in ein ILP überführt wird. Letzteres kann dann von generischen Lösern behandelt werden. Deren Lösungen werden dann in eine Lösung des Ausgangsproblems übersetzt. Dieses Verfahren arbeitet zunächst symbolisch, allerdings ist es möglich, daraus numerische Schranken abzuleiten, sodass es vom Ergebnis her Ähnliches liefert wie das Bitwise-Verfahren.

Es ist keine vergleichende Arbeit zu diesem Thema bekannt, sodass wir hier keine Empfehlung für das eine oder andere Verfahren aussprechen wollen. Vielmehr

[4]Die trappsche Analyse konnte leider nie mit ausreichender Stabilität oder akzeptabler Geschwindigkeit implementiert werden.

[5]engl.: *integer linear programming* (ILP)

sollte hier motiviert werden, dass hinreichende Ansätze zur Lösung des Problems der Bitbreitenanalyse verfügbar sind. Darüberhinaus ist auch die *Datenausrichtung*[6] (siehe auch 6.5.2.3) ein nicht zu vernachlässigendes Problem.

7.3 Graphen

Zur Repräsentation von Programmen verwenden wir Firm, eine graphbasierte Zwischendarstellung mit SSA-Eigenschaft (siehe Abschnitt 6.3.4). Die Beschreibung der Zwischendarstellung im Handbuch [TLB99] und vor allem ihre Implementierung libfirm [Lin02] lässt die Frage offen, was genau aus graphentheoretischer Sicht ein Firm-Graph ist. In diesem Abschnitt gehen wir dieses Problem an. Schließlich klären wir, wie die Integration und Zugriffsschicht für die Graphen bzw. den Graphersetzer aussieht.

7.3.1 Anforderungen

Offensichtlich ist, dass wir zur Darstellung des Daten- und Steuerflusses von Firm am Besten gerichtete Graphen verwenden sollten. Ebenso benötigen die Ecken ihnen zugeordnete Attribute: Zum Beispiel müssen mit der *Const*-Ecke numerische Konstanten assoziierbar sein. Ferner muss die Art der Ecke festgehalten werden, allerdings hat man hierzu zwei Möglichkeiten: Einerseits ein Attribut, das den Operationstyp repräsentiert und andererseits die Benutzung von markierten bzw. typisierten Graphen. Obgleich die beiden Alternativen zunächst äquivalent in der Ausdrucksstärke sind, ist die Verwendung typisierter Graphen als elegantere Wahl vorzuziehen, denn damit ist nicht nur der Passungsmorphismus direkter formulierbar, sondern überdies ist leicht eine Vererbungshierarchie auf den Eckentypen definierbar, was unter anderem das Spezifizieren von teilweise unbestimmten Ecken deutlich erleichtert.

Ecken weisen in Firm eine innere Struktur auf. Die ausgehenden Kanten beginnen an Ports, die mit Nummern assoziiert sind und je nach Eckentyp eine besondere Bedeutung haben. Zum Beispiel entspringt am Port mit der Nummer „-1“ an fast allen Ecken die Kante, die zum Grundblock der jeweiligen Ecke zeigt. Diese Eigenschaft von Firm lässt sich leider nicht direkt auf grundlegende Konzepte der Graphentheorie abbilden. Ein Weg ist die Kanten mit Attributen zu versehen, die den Nummern der Ports entsprechen. Weil die dann auftretenden Zahlen aber nur in Verbindung mit den jeweiligen Ursprungsecken verstehbar sind, führt dies zu schlecht lesbaren Spezifikationen. Wir haben hingegen die Nummern zugunsten von Kantentypen aufgegeben, welche die Bedeutung der jeweiligen Kante klar signalisieren. Allerdings werden die Portnummern in seltenen Fällen, wie zum Beispiel bei der *Call*-Ecke zum Angeben der Position der Argumente benötigt. Dafür haben wir die Attributierung von Kanten vorgesehen. Um das Typsystem von Kanten und Ecken zu vereinheitlichen, haben auch Kantentypen eine Vererbungshierarchie. Dies ermöglicht ebenfalls das Selektieren gewisser Kanteneigenschaften in Form von Typen. Durch die Typen soll es auch möglich sein, mehrere orthogonale Aspekte auszudrücken, zum Beispiel ist die Additionsecke gleichzeitig eine arithmetische Operation, hat zwei Operanden und ist assoziativ sowie kommutativ. Eine baumartige Typhierarchie führt hier

[6] engl.: *alignment*

immer zu – durch Mehrfachvererbung[7] vermeidbaren – Idiosynkrasien.

Zusammenfassend halten wir fest, dass zur eleganten Repräsentation von Firm *gerichtete, typisierte und attributierte Multigraphen mit Mehrfachvererbung auf den Typen* nötig sind.

Die Graphinstanzen müssen aber auch als Datenstruktur vom Übersetzer erstellt und gespeichert werden. Der Ort sowie die Art und Weise der Speicherung können die praktische Benutzbarkeit mindestens so beeinflussen wie die geeignete Formalisierung der Graphen. Darum stellen wir im Folgenden die beiden von uns erprobten Integrationsgrade der Graphdatenhaltung vor, und diskutieren ihre jeweiligen Vor- und Nachteile.

7.3.2 Lose Kopplung

Die libGr-Graphbibliothek[8] wurde konstruiert um Gr-Graphen darstellen zu können und ist eine eigenständige und allgemeine Schnittstelle für Graphersetzer mit einer eigenen Zugriffsschicht für die jeweiligen Graphen (siehe Abbildung 7.2). Der Graphersetzungsgenerator GrGen steuert eine Bibliothek von einzeln ausführbaren Graphersetzungen sowie ein Typ- und Attributierungsmodell für Ecken und Kanten bei. Die Integration geschieht mittels einer Transformation `f2g` von Firm in diese eigenständige Darstellung. Eine in einer konventionellen Hochsprache implementierte Optimierung, die Graphersetzungen benutzt, kann diese durch entsprechende libGr-Funktionen anwenden. Ebenso sind Graphmanipulationen direkt auf dem libGr-Graphen möglich. Natürlich können auch mehrere Optimierungen hintereinander ausgeführt werden. Sobald die Optimierungen auf der libGr abgeschlossen sind, wird mittels `g2f` wieder nach Firm zurücktransformiert.

Dieser Ansatz hat mehrere Vorteile: Die Datenstrukturen des Graphen können auf das Finden von Graphmustern hin optimiert gestaltet werden. Über die Schnittstelle kann sichergestellt werden, dass alle bereitgestellten Manipulationsmöglichkeiten für die Aktionsplanung des Ersetzers sichtbar und verträglich sind. Darüber hinaus bietet libGr eine allgemeine, generische, graphentheoretisch fundierte und effiziente Schnittstelle zur Graphmanipulation – auch ohne den Ersetzer zu benutzen.

Jedoch ist offensichtlich, dass eine enge Verzahnung mit Firm nicht möglich ist, so kann zum Beispiel eine Analyse, die auf Firm schon vorhanden ist – wie die Dominanzanalyse – von einer libGr basierten Optimierung nicht wiederverwendet werden. Ferner tritt ein erheblicher Speicher- und Laufzeitwasserkopf auf, denn die gesamte Zwischendarstellung muss einmal hin und wieder zurückkopiert werden und ist dabei – zumindest für eine gewisse Zeit – doppelt im Speicher.

7.3.3 Direkte Kopplung

Die graphbasierte Zwischendarstellung Firm kann auch direkt zur Datenhaltung für den Graphersetzer verwendet werden. Dies führt dazu, dass, wie in Abbildung 7.3 zu sehen ist, die Zwischendarstellung nicht umkopiert werden muss, sondern direkt

[7] Da in unserem Typsystem nur Attribute und Typen selbst vorkommen, umgehen wir hier die Probleme, die in Programmiersprachen bei Mehrfachvererbung von Methoden auftreten können.

[8] Eine vollständige Erläuterung der Bibliothek und ihrer Entwurfsprinzipien findet sich in [BG07, Hac03, Gru04] sowie auf `www.grgen.net`.

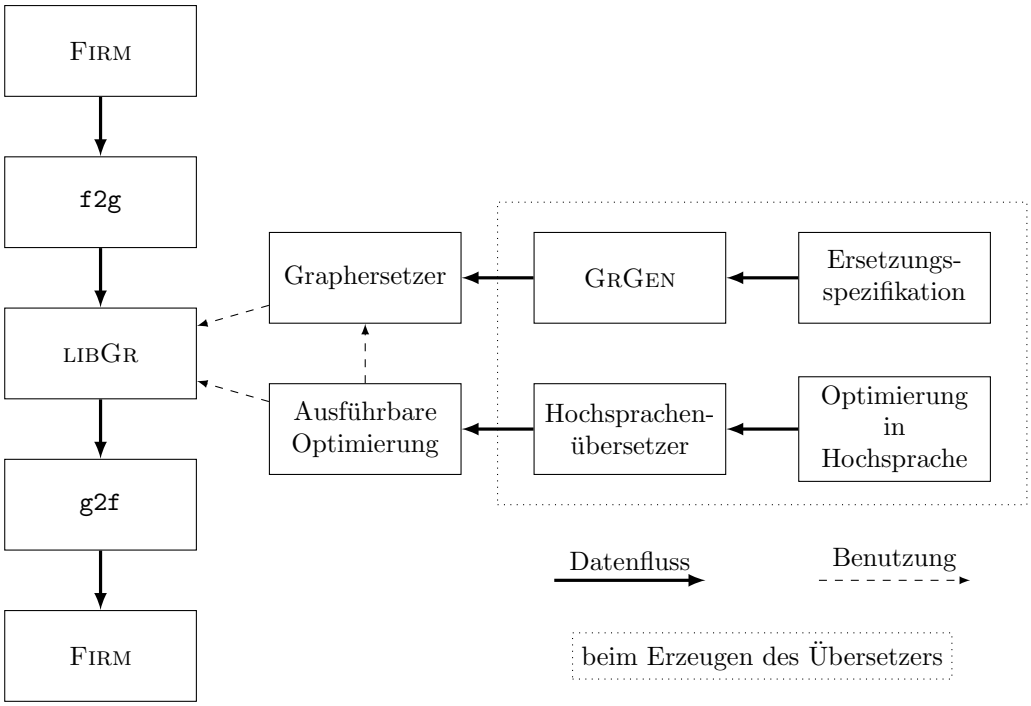


Abbildung 7.2: libGr zur Graphdatenhaltung

verwendet werden kann. Die Firm-Bibliothek ist allerdings im Gegensatz zur libGr kein allgemeines, zur Graphdatenhaltung konstruiertes System – mit folgenden Ergebnissen: Erstens sind nicht alle Graphen, die sich in dem einen System darstellen lassen, auch in dem anderen repräsentierbar. Überdies kann zweitens auch keine graphentheoretisch fundierte Schnittstelle zur Manipulation des Graphen angeboten werden, sondern vielmehr eine spezielle, aus der Welt der graphbasierten Zwischendarstellungen stammende Sicht.

Letzteres ist gleichzeitig eine der großen Stärken und Schwächen dieses Ansatzes. Einerseits wollen wir den Prozess der Graphmustersuche bzw. der deklarativen Graphersetzung eng mit den normalen Zugriffsfunktionen von Firm verzahnen, was hiermit, also der direkten Kopplung, gelungen ist. Andererseits muss das Graphersetzungssystem bei allen Zugriffen, also auch bei Zugriffen direkt auf Firm, den internen Zustand nachführen, was angesichts der mannigfaltigen und z. T. komplexen Operationen auf Firm nicht trivial ist. Wenn wir Firm als Grundlage unserer Bestrebungen verwenden wollen, wird es vorkommen können, dass wir Ecken und vielleicht sogar Kanten benötigen, die im eigentliche Firm nicht vorgesehen sind. Fernerhin müssen maschinennahe Ecken schon vor dem eigentlichen Backend im Firm eingefügt werden können. Die beiden letzteren Forderungen sind mit der aktuellen Implementierung von Firm schon so vorgesehen, da Firm neuerdings sogar zur Darstellung des Assemblercodes bis unmittelbar vor die Assemblerausgabe benutzt wird.

Da Firm sich durch eine effiziente, oft aber implizite Darstellung von Kanten und ihren Typen bzw. Attributen auszeichnet, muss GrGen spezialisierteren Code für die Graphersetzer, genauer für den Zugriff auf die Graphdatenhaltungsschicht, generieren. Dies macht die Aufgabe zur Übersetzungszeit der Ersetzungsspezifikation zwar aufwendiger, aber ermöglicht effizienter arbeitende Mustersucher, da zum Beispiel die Position der Argumente nicht wie in der libGr über Attribute oder Ty-

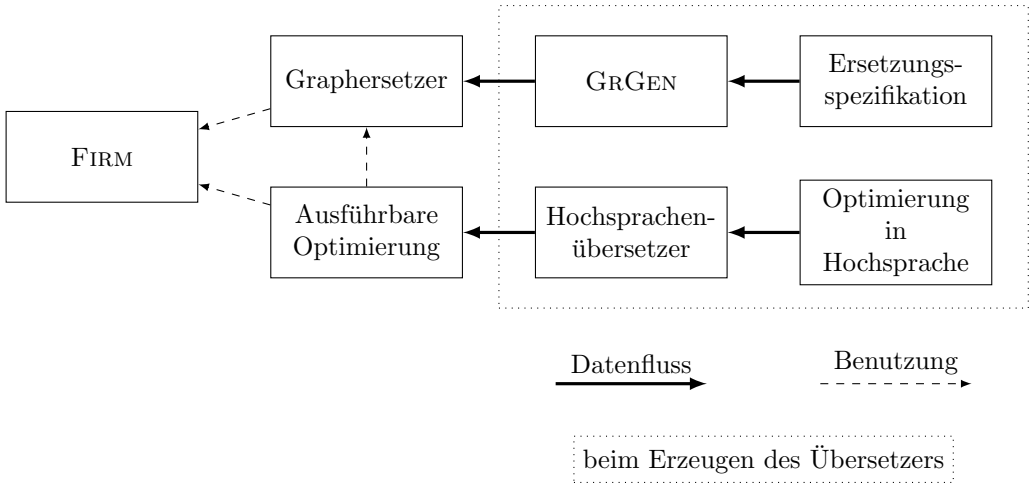


Abbildung 7.3: Firm zur Graphdatenhaltung

pen realisiert werden muss, sondern direkt als Position in einem Feld[9] ausgedrückt wird. Dies erfordert vom Graphersetzungsgenerator kompliziertere Schema zur Erzeugung der Mustersucher, da manche Attribute tatsächlich als Attribute, andere aber direkter in der Graphdatenhaltungsschicht repräsentierbar sind. Immerhin kann es eben durch diesen spezialisierteren Code zu einer Beschleunigung der Mustersuche kommen.

7.3.4 Ein Firm-Graphmodell

Wir haben in Abschnitt 4.1 eine sehr allgemeine Klasse von Graphen formalisiert, nämlich die Gr-Graphen, und anhand zweier Alternativen gesehen (siehe Abschnitt 7.3.2 und 7.3.3), auf welcher Grundlage wir diese im Rechner repräsentieren können. In diesem Abschnitt werden wir schließlich definieren, wie das Graphmodell für die Zwischendarstellung Firm aussieht. Dies scheint zunächst widersinnig, eingedenk der Tatsache, dass wir insbesondere Firm selbst zur Repräsentation von Firm verwenden, ist es jedoch nicht: Die Abbildung von Elementen der Firm-Zwischendarstellung auf die eines Graphmodells ist nicht trivial und bietet einiges an Freiheitsgraden.

Zum Beispiel hat eine `Add`-Operation – aus Sicht der Firm-Implementierung – einen Zeiger auf eine Struktur namens `Mode` mit mehreren Feldern, die Eigenschaften des Ergebnisses der Operation beschreiben. Dazu gehört insbesondere ein Feld, das angibt, ob der Ergebniswert in ein Integer-Register passt. Die Felder der `Mode`-Struktur lassen sich eindeutig aus dem Wert des Feldes `code` ableiten, sind also redundant. Nun ist die Frage, wie diese Situation auf ein Graphmodell abgebildet werden soll, nicht zwingend zu beantworten, sondern es bieten sich drei in Abbildung 7.4 skizzierte Interpretationen an:

1. Für die `Mode`-Struktur wird ein eigenständiger Eckentyp eingeführt. Den Zusammenhang mit der `Add`-Operation wird durch eine Kante speziellen Typs, der `has_mode`-Kante, hergestellt. Dieses Vorgehen führt dazu, dass die Attribute pro `Mode` nur einmal im Speicher zu halten sind. Allerdings muss die

[9] engl.: *array*

```
node class Mode {
  name       : string;
  size       : int;
  sort       : ENUM_sort;
  code       : ENUM_modecode;
  sign       : boolean;
  arithmetic : ENUM_arithmetic;
  shift      : int;
}

node class Add extends IR_node;

edge class has_mode
  connect IR_node [1] -> Mode [*];
```

(a) Mode als eigenständige Ecke verbunden mit Add-Ecke durch eine spezielle Kante.

```
node class IR_node {
  name       : string;
  size       : int;
  sort       : ENUM_sort;
  code       : ENUM_modecode;
  sign       : boolean;
  arithmetic : ENUM_arithmetic;
  shift      : int;
}

node class Add extends IR_node;
```

(b) Alle Mode-Attribute in der Oberklasse der konkreten Ecken.

```
node class IR_node {
  code        : ENUM_modecode;
}

node class Add extends IR_node;

get_mode_name : ENUM_modecode --> string
get_mode_size : ENUM_modecode --> int
...
```

(c) Eindeutiger Schlüssel für Mode-Ecken und benutzerdefinierte Funktionen für ableitbare Attribute.

Abbildung 7.4: Alternative Interpretationen des Zusammenhangs von Add-Operationen und Mode-Strukuren in Firm.

has_mode-Kante für jede Ecke, die eine Mode-Struktur hat, materialisiert werden, was natürlich auch Speicher verbraucht. Jedoch kann die Bedingung, dass zwei Ecken den gleichen Mode haben als Graphmuster ohne Bedingung an Attribute (siehe Abschnitt 4.2.2) formuliert werden. Siehe Abbildung 7.4(a).

2. Die Felder der Mode-Struktur werden dem Obertyp IR_node der konkreten Ecken zugeschlagen. Dadurch besitzt jede Add-Ecke die ihr zugeordneten Daten der Mode-Struktur. Allerdings werden so die Attribute in jeder konkreten Ecke multipel redundant gespeichert. Siehe Abbildung 7.4(b).

3. Nur das den Mode eindeutig bezeichnende Attribut code wird in den Obertyp IR_node eingefügt. Die restlichen Werte der Mode-Struktur können durch Funktionen aus code ermittelt werden. Nachteilig hierbei ist, dass diese Abbildungen nicht innerhalb des Graphmodells spezifizierbar sind. Mithin muss

GrGen um diese Funktionen erweitert werden. Siehe Abbildung 7.4(c).

Die letzten beiden Alternativen haben entweder Nachteile bei der Speichernutzung oder Implementierungskomplexität. Überdies kann bei diesen auch nicht der Test nach gleichen Modi durch ein Graphmuster ausgedrückt werden, sondern muss durch Attributbedingungen beschrieben werden. Die erste Variante hat allerdings einen erhöhten Aufwand beim Zugriff auf die Attributwerte – zumindest gegenüber der zweiten Variante, da zuerst die `has_mode` Kante verfolgt werden muss. Wir haben uns bei der Definition des Firm-Graphmodells dennoch, wegen der überwiegenden Vorteile, für die erste Variante entschieden.

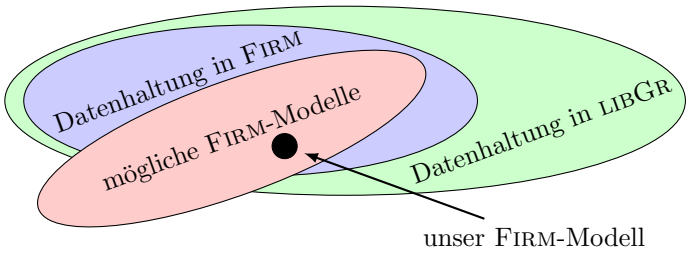


Abbildung 7.5: Vergleich der darstellbaren Graphen bei Graphdatenhaltung in Firm und libGr im Bezug auf die möglichen und tatsächlich verwendete Firm-Modell.

Anhand eines Details haben wir gesehen, welche Schwierigkeiten die Abbildung von Firm-Elementen auf die eines Graphmodells verursacht. Dabei ist immer zwischen mehreren Kriterien abzuwägen, letztlich bleibt aber ein erheblicher Spielraum, der definitorisch zu entscheiden bleibt. Wir haben dabei aber zu beachten, dass das gewählte Graphmodell in GrGen darstellbar bleibt, schärfer noch: Es muss auch darstellbar sein, wenn die Graphdatenhaltung in Firm geschieht (siehe Abschnitt 7.3.3). Man beachte, dass es auch möglich ist, Firm so zu interpretieren – z. B. als Hypergraph, wobei die Kanten die Operationen und die Ecken die Werte sind – dass die entstehende Struktur nicht mehr – zumindest nicht trivial – durch die Firm-Implementierung repräsentierbar ist. Diese Beobachtungen führen zu der in Abbildung 7.5 dargestellten Situation.

Die gesamte Umsetzung der Zwischendarstellung Firm in ein Graphmodell des Graphersetzungssystems GrGen ist in Anhang B.3 zu sehen. Wir haben bei dieser Umsetzung der Vollständigkeit halber auch die Firm-Typen und Entitäten, die nur selten von Optimierungen benutzt werden, berücksichtigt, um so keiner Optimierungsmöglichkeit im Wege zu stehen. Die Typen nebst Entitäten bilden letztlich das Typsystem sowie die vom Zielprogramm im Speicher zu haltenden Artefakte der Instanzen dieser Typen ab. Auf sie wird hier nicht weiter eingegangen, da sie im Kontext dieser Arbeit nicht benötigt werden; genaueres ist im Benutzerhandbuch von Firm zu finden [Lin02].

7.4 Spezifikationstechniken

In diesem Abschnitt betrachten wir zwei neuartige, auf Graphersetzung aufbauende Techniken zur Spezifikation von Optimierungen.

7.4.1 Anwendungsspezifische Sprache

Die anwendungsspezifische Sprache baut, wie die natürliche Spezifikation (siehe Abschnitt 7.4.2), auf der Graphersetzungsschicht auf, benutzt allerdings darüber hinaus noch weitere Algorithmen aus dem Übersetzerbau und der Graphentheorie (siehe Abbildung 7.6). Die so konstruierte Sprache soll möglichst ausdrucksstark und effektiv zur Konstruktion von Optimierungen einsetzbar sein, wir nennen sie kurz *Optimierungssprache*. Die anwendungsspezifische Sprache konnte im Rahmen dieser Arbeit nicht mehr vollständig realisiert werden, muss also hier fragmentarisch bleiben.

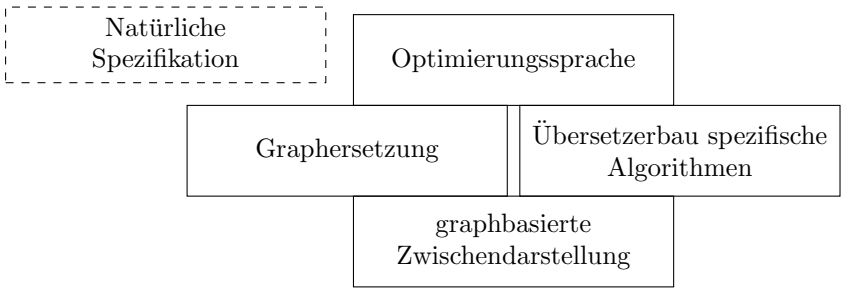


Abbildung 7.6: Optimierungssprache als anwendungsspezifische Sprache für den Übersetzerbau

Programm 7.1: Beispiel für die Optimierungssprache

```
1  rule r1 {
2    ...
3  }
4
5  rule r2 {
6    ...
7  }
8
9  void f(IR ir, IR_node x) {
10   forall pattern {
11     x <-:df- a:Add -:df-> x;
12   } in ir do {
13     if(dom(a, x))
14       Console.WriteLine("Found␣Add␣{1}␣with␣two␣identical␣operands␣that␣dom.",
15                         a.id);
16   }
17
18   bool e;
19   apply(e = (r1 & !r2) || r2*, ir);
20   if(e)
21     ...
22 }
```

Programm 7.1 stellt exemplarisch dar, wie die Sprache aussehen kann. Die Sprache integriert die Mustersuche und die Regelanwendung sowie die Definition von Graphersetzungsregeln in die imperative Hochsprache. Auf die Elemente der Gra-

phersetzungsregeln kann zum Beispiel im `forall`-Konstrukt, das über alle Passungen iteriert, direkt zugegriffen werden. Der Name `a` wird in der Schleife in jedem Durchgang auf die dann gültige Ecke des Arbeitsgraphen `ir` gesetzt. Hier sieht man den Vorteil: bei einer rein bibliotheksbasierten Integration wären hier komplizierte Zugriffsfunktionen und Casts notwendig (vgl. Anhang A.5).

Die Verarbeitung der Optimierungssprache kann durch einen Präprozessor geschehen, der die Graphersetzungsregeln und Muster extrahiert und dem eigentlichen GrGen-System zur Verarbeitung vorlegt. Die daraus von GrGen erzeugte dynamische Bibliothek wird dann von jenem Code benutzt, den der Präprozessor aus den Zugriffen auf Regel- und Mustergraphelemente und Regelanwendungen erzeugt hat. Die Frage, welche Sprachelemente nötig sind, und wie ihre Schreibweisen aussehen sollten, ist durch Fallstudien zu klären. Die hier präsentierten Fragmente der Optimierungssprache sind noch nicht durch eine solche Prüfung gegangen. Kroll hat dieses Thema in seiner Diplomarbeit [Kro08] genauer untersucht und Lösungen der Detailprobleme erarbeitet.

7.4.2 Natürliche Spezifikation

Wir wollen die automatischen, übersetzerseitige Ausnutzung reichhaltiger Befehlssätze mittels Graphersetzung vorantreiben. Dazu ist in der linken Seite einer Graphersetzungsregel ein Zwischendarstellungsgraph als Muster anzugeben, welcher der Wirkung eines reichhaltigen Befehls, dargestellt mit den „normalen“ Elementen der Zwischendarstellung, entspricht. Wenn ein solches Muster in der Zwischendarstellung eines Programms gefunden wird, kann diese durch den entsprechenden reichhaltigen Befehl ersetzt werden. Allerdings ist dieses Verfahren in mehrerlei Hinsicht verbesserungswürdig: Erstens sind die Muster sehr groß. Zweitens können bei jeder Veränderung der Definition der Zwischendarstellung oder der auf ihr arbeitenden Optimierungen die spezifizierten Muster ungültig werden. Drittens schließlich muss der Spezifikateur nicht nur detaillierte Kenntnis der jeweiligen reichhaltigen Befehlssätze besitzen, sondern auch die Zwischendarstellung erlernt haben.

Durch *natürliche Spezifikationen* gelingt es, alle diese Nachteile zu vermeiden. Die Wirkung einzelner reichhaltiger Befehle wird dazu durch Programmfragmente in einer Hochsprache beschrieben[10]. Der Übersetzer erstellt daraus eine Zwischendarstellung in Graphform. Aus dieser wird wiederum ein Graph extrahiert, der als Muster zum Finden ebendieses reichhaltigen Befehls verwendet wird. Abbildung 7.7 veranschaulicht dieses Vorgehen. Die `g2s`-Komponente bewerkstelligt die Extraktion des gewünschten Graphmusters aus der Zwischendarstellung. Anschließend kann GrGen dazu benutzt werden, die Graphersetzungsregeln in ausführbare Module zu verwandeln. Diese eigentlichen Ersetzungen werden von einem generischen Steuerungsmodul ausgelöst, welches die vorbereitenden Transformationen einleitet, eine geeignete Anwendungsreihenfolge der Ersetzungen wählt und schließlich Aufräumarbeiten erledigt.

Als Nächstes betrachten wir ein Beispiel eines reichhaltigen Befehls und seiner natürlichen Spezifikation. Im Anschluss daran beschreiben wir, aus welchen Elementen natürliche Spezifikationen bestehen. Die folgenden Abschnitte klären die genaue Funktionsweise von `g2s` und gehen auf die *algorithmische Steuerung* der erzeugten

[10] Da die Angabe der Befehlssemantik in einer Hochsprache auf besonders *natürliche* und einsichtige Weise erfolgt, nennen wir diese Methode *natürliche Spezifikation*.

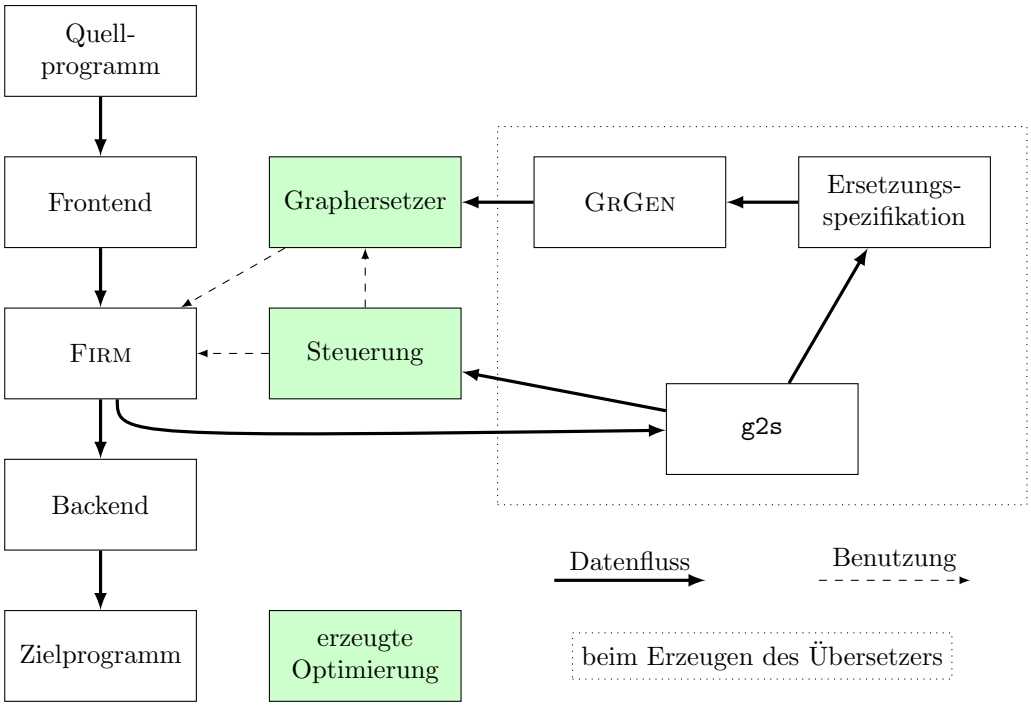


Abbildung 7.7: Natürliche Spezifikationen werden bei der Erzeugung des Übersetzers in Graphersetzungen überführt. Diese können bei weiteren Übersetzerläufen zusammen mit der notwendigen generischen Steuerung als Optimierung verwendet werden.

Graphersetzungsregeln ein. Darüber hinaus sei auf die Diplomarbeiten von Hofmann und Schösser hingewiesen [Sch07b, Hof04].

7.4.2.1 Beispiel

Der Befehl `PSADBW`[11] des IA-32/SSE Befehlssatzes hat mathematisch gesehen folgende Semantik:

$$\mathrm{sad}(a,b) = \sum_{i=0}^{15} |a_i - b_i|$$

Im Referenzhandbuch von Intel [Int07a] ist dies folgendermaßen beschrieben, wobei die textuelle Beschreibung ebendort von nachstehendem Pseudocodefragment ergänzt wird (siehe Programm 7.3):

> Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Figure 4-5 [Anm. d. Autors: hier nicht abgedruckt] shows the operation of the PSADBW instruction when using 64-bit operands.

[11] engl.: *sum of absolute differences*

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 highorder bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Programm 7.2: Pseudocodefragment für PSADBW laut Handbuch von Intel [Int07a]

```
1 TEMP0 := ABS(DEST[7:0] - SRC[7:0]);
2 /* Repeat operation for bytes 2 through 14 */
3 TEMP15 := ABS(DEST[127:120] - SRC[127:120]);
4 DEST[15:0] := SUM(TEMP0:TEMP7);
5 DEST[63:16] := 000000000000H;
6 DEST[79:64] := SUM(TEMP8:TEMP15);
7 DEST[127:80] := 000000000000H;
```

Die natürliche Spezifikation des `PSADBW`-Befehls ist in Programm 7.3 zu sehen.

Programm 7.3: Natürliche Spezifikation für `PSADBW`

```
1  #include "define_operation.h"
2
3  void psadbw()
4  {
5    unsigned char *a = Operand_0("vector", "register", "xmm"  );
6    unsigned char *b = Operand_1("vector", "register", "xmm"  );
7    unsigned int  *r =    Result("vector", "register", "in_r0");
8    Emit(". psadbw %S1, %S0\\n. pshufd \\$0xC6, %S0, %S1\\n. paddd %S1, S0");
9    Destroys("in_r0");
10   Priority(2);
11   CostSavings(40);
12
13   r[0] = ((a[0]  > b[0])  ? (a[0]  - b[0])  : (b[0]  - a[0]))  +
14          ((a[1]  > b[1])  ? (a[1]  - b[1])  : (b[1]  - a[1]))  +
15          ((a[2]  > b[2])  ? (a[2]  - b[2])  : (b[2]  - a[2]))  +
16          ((a[3]  > b[3])  ? (a[3]  - b[3])  : (b[3]  - a[3]))  +
17          ((a[4]  > b[4])  ? (a[4]  - b[4])  : (b[4]  - a[4]))  +
18          ((a[5]  > b[5])  ? (a[5]  - b[5])  : (b[5]  - a[5]))  +
19          ((a[6]  > b[6])  ? (a[6]  - b[6])  : (b[6]  - a[6]))  +
20          ((a[7]  > b[7])  ? (a[7]  - b[7])  : (b[7]  - a[7]))  +
21          ((a[8]  > b[8])  ? (a[8]  - b[8])  : (b[8]  - a[8]))  +
22          ((a[9]  > b[9])  ? (a[9]  - b[9])  : (b[9]  - a[9]))  +
23          ((a[10] > b[10]) ? (a[10] - b[10]) : (b[10] - a[10])) +
24          ((a[11] > b[11]) ? (a[11] - b[11]) : (b[11] - a[11])) +
25          ((a[12] > b[12]) ? (a[12] - b[12]) : (b[12] - a[12])) +
```

```
26          ((a[13] > b[13]) ? (a[13] - b[13]) : (b[13] - a[13])) +
27          ((a[14] > b[14]) ? (a[14] - b[14]) : (b[14] - a[14])) +
28          ((a[15] > b[15]) ? (a[15] - b[15]) : (b[15] - a[15])) ;
29  }
```

Gut zu erkennen ist die Ähnlichkeit mit dem Pseudocodefragment, das Intel verwendet hat, um den Befehl zu erläutern. In Abschnitt 9.4 wird gezeigt, wie wir durch die Benutzung der natürlichen Spezifikation des `PSADBW`-Befehls zusammen mit anderen reichhaltigen Befehlen ein Programm zur Bewegungsschätzung um den Faktor 20 beschleunigen können. Die einzelnen Elemente der natürlichen Spezifikation werden im nächsten Abschnitt erläutert.

7.4.2.2 Spezifikationssprache

Die natürliche Spezifikation eines reichhaltigen Befehls entspricht jeweils einer C-Funktion mit eindeutigem Funktionsnamen, wobei mehrere Spezifikationen möglich sind, um verschiedene idiomatische Schreibweisen zu erfassen. Der erste Teil einer Spezifikation beseht aus dem Definitionsteil, in dem Ein- und Ausgaben, aber auch die Pragmatik des Befehls angegeben wird, die sich per definitionem nicht durch den vom Übersetzer generierten Zwischendarstellungsgraphen erschließen lassen. Der zweite und letzte Teil stellt die eigentliche Verhaltensbeschreibung dar. Dieser Teil wird vom Übersetzer in einen Zwischendarstellungsgraphen verwandelt, aus dem `g2s` eine entsprechende Graphersetzungsregel ableitet.

Der Definitionsteil kann folgende Elemente enthalten, die Reihenfolge ist dabei beliebig, ebenso wie die Namen der verwendeten Variablen:

Eingabe mit ***`element_typ op_n =`***
`Operand_n(char* kind, char* loc, char* reg_class);`

Eine solche Zeile definiert eine Eingabe `op_`n, die entweder ein Register oder ein Speicherbereich ist.

`element_typ`
: Der Typ entspricht dem Programmiersprachentyp des skalaren oder vektorwertigen Wertes, der als Eingabe fungiert.

`kind`
: Die möglichen Werte `"scalar"` und `"vector"` geben an, ob sich um einen skalaren Wert oder einen vektorwertigen Wert handelt.

`loc`
: Gibt an, ob es sich der Wert in einem Register oder in einer Speicherstelle befindet (`"register"` und `"memory"`).

`reg_class`
: Falls es sich um ein Register handelt, referenziert `reg_class` einen im Backend vereinbarten Namen für die Registerklasse z. B. für IA-32 / SSE kommen `"gp"` und `"xmm"` in Frage. Beschreibt `op_`n kein Register, dann wird der Parameter ignoriert.

Ausgabe mit ***`element_typ res =`***
`Res(char* kind, char* loc, char* reg_class);`

Obige Zeile definiert eine Ausgabe `res`, wobei die selben Parameter wie bei

der Eingabe erlaubt sind. Für `reg_class` ist darüber hinaus noch `"in_r`n`"` erlaubt, wobei dies angibt, dass die Ausgabe in das entsprechende Eingaberegister zurückgeschrieben wird.

Schablone für die Assemblerausgabe ***`Emit(char* assembler_template)`***
Die Zeichenfolge `assembler_template` ist die Schablone für die Assemblerausgabe, sie bietet außer normalen Text auch `%S`n oder `%D`n. Dies sind die Platzhalter für Eingabe- oder Ausgabeoperanden, insbesondere werden dort während der Assemblierung die vom Backend gemäß der Registerklasse zugeteilten Operanden eingesetzt.

Nebeneffekte auf Register ***`Destroys(char* register_names)`***
Ein reichhaltiger Befehl kann zusätzliche Register (abgesehen von seinem Ausgaberegister) bei seiner Abarbeitung unbrauchbar machen. In `register_names` können diese „zerstörten“ Register, in der `"in_r`n`"` Schreibweise, angegeben werden.

Befehlsklasse ***`Priority(int pvalue)`***
Die Wichtigkeit der Ersetzung eines Befehls kann hiermit signalisiert werden. Dieser Parameter dient der algorithmischen Steuerung als Hinweis bei der *Regelauswahl*.

Kostenersparnis ***`CostSavings(int cvalue)`***
Dies gibt an, welche Kostenersparnis anzunehmen ist, falls der reichhaltige Befehl anstatt seiner elementaren Befehle ausgeführt wird.

Unter der Voraussetzung, dass `op_`n und `res` jeweils die Ein- und Ausgaben des reichhaltigen Befehls sind, enthält der Verhaltensbeschreibungsteil eine Spezifikation des Verhalten des reichhaltigen Befehls mit den Mitteln der entsprechenden Hochsprache, in der wir spezifizieren. Die einzelnen Anweisungen und Ausdrücke der Sprache, die dort verwendet werden, geben an, wie aus den Eingaben die Ausgaben berechnet werden können. Natürlich ist diese immer nur eine von vielen möglichen Ausdrucksweisen für das Verhalten des reichhaltigen Befehls. Wir müssen aber daraus das Muster ableiten, mit dem wir dann die Vorkommen in den zu optimierenden Programmen finden wollen. Obwohl unser Verfahren mit normalisierenden Vorverarbeitungsschritten (siehe Abschnitt 7.4.2.5) und automatischer Variantenbildung (siehe Abschnitt 7.4.2.3) arbeitet, ist es unter Umständen nicht möglich, alle Vorkommen tatsächlich auch zu finden.

Dies ist unvermeidbar, da die semantische Äquivalenz von Programmen nicht berechenbar ist. Die Äquivalenz von einzelnen Grundblöcken, die nur arithmetische Operationen enthalten, ist mindestens NP-hart. Nimmt man die Alias-Problematik über gegebenenfalls geladenen oder geschriebenen Speicher hinzu, ist das Problem sogar schon für einzelne Grundblöcke nicht mehr exakt berechenbar. Jede Optimierung im Übersetzerbau baut letztlich darauf auf, aus syntaktischen Gegebenheiten auf semantische Aussagen zu schließen. Also muss immer damit gerechnet werden, dass wir gewisse idiomatische Schreibweisen nicht erkennen, darum können mehrere Varianten ein und desselben reichhaltigen Befehls angegeben werden. Mehr noch: In Anlehnung an die Forderung von Robison [Rob01] (vgl. Abschnitt 6.1.3) ist es somit für versierte Anwender erstmals möglich, ohne detailliertes Wissen über den Übersetzerbau und insbesondere ohne den Übersetzer selbst zu kennen, die Optimierungen um neue Idiome zu erweitern.

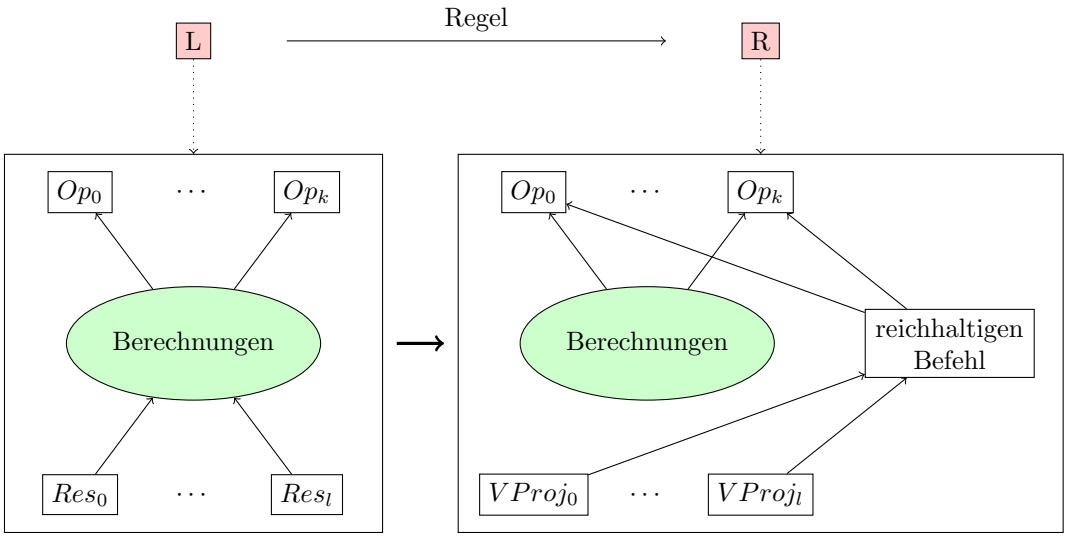


Abbildung 7.8: Die Anwendung einer Graphersetzungsregel für einen reichhaltigen Befehl.

7.4.2.3 Spezifikationsextraktion mit g2s

Der Zwischendarstellungsgraph, aus dem eine natürliche Spezifikation erzeugt wird, kann offensichtlich nicht direkt als Graphersetzungsregel dienen. Im Folgenden beschreiben wir die dafür notwendigen oder aus Gründen der Optimierung nützlichen Schritte.

Als Erstes werden unnötige Ecken und Kanten entfernt, dazu gehören zum Beispiel der Prozedurprolog und -epilog, aber auch, an gewissen Stellen, die *Speicherkante* (siehe Abschnitt 6.3.4). Die Speicherkante und die damit verbundene Serialisierung des Speichers spielt im Folgenden noch mehrfach eine Sonderrolle.

Um die Trefferwahrscheinlichkeit zu erhöhen, werden automatisch Varianten für verschiedene direkte und indirekte *Adressierungsarten* erzeugt. Auf diese Weise passt ein Speicherzugriff wie `a[0], a[1]` aus der natürlichen Spezifikation auf folgende Konstrukte im zu optimierenden Programm:

- Direkter Zugriff mit konstantem Versatz: `a[0+c], a[1+c]`, wobei `c` eine Konstante ist.
- Direkter Zugriff mit variablem Versatz (insbesondere wenn `i` ein Schleifenzähler ist): `a[0+i], a[1+i]`.
- Indirekter Zugriff über eine Verweistabelle (insbesondere einem nicht linearisierten mehrdimensionalen Feld): `a[j+b[i]], a[j+1+b[i]]`.

Die annotierte Pragmatik wird durch Inspektion der Funktionsaufrufe im Definitionsteil der natürlichen Spezifikation gewonnen. So steht fest, wie die algorithmische Steuerung zu parametrisieren ist und welche Informationen an das Backend über den reichhaltigen Befehl weiterzugeben sind. Die Verhaltensbeschreibung selbst mündet dann in den Mustergraphen.

Der Ersetzungsgraph ist nicht, wie zunächst anzunehmen ist, bis auf den neuanzulegenden reichhaltigen Befehl leer, sondern enthält zusätzlich die gesamte linke

Seite. Dies ist nötig, da der reichhaltige Befehl möglicherweise nicht alle Zwischenergebnisse produziert, die aber unter Umständen nach wie vor gebraucht werden. Darum erhält die Graphersetzungsregel alle Berechnungen und fügt den reichhaltigen Befehl zusätzlich ein. Überdies werden ausschließlich die Ecken im Arbeitsgraphen, die den produzierten Ergebnissen des reichhaltigen Befehls entsprechen, speziell behandelt. Jene Ergebnisecken werden nicht mehr benötigt, dürfen aber, um ihre Verwender nicht zu verlieren, nicht gelöscht werden. Also werden die Ergebnisecken zu sogenannten `VProj`-Ecken umtypisiert, die das ihrem vormaligen Wert entsprechende Element aus dem Ergebnisvektor des reichhaltigen Befehls projizieren. Diese Situation bei der Ersetzung ist in Abbildung 7.8 zu sehen. Falls der reichhaltige Befehl kein vektorwertiges Ergebnis hat, wird der letzte Schritt natürlich entsprechend abgeändert: Die dann eindeutig bestimmte Ergebnisecke wird selbst zum reichhaltigen Befehl umtypisiert.

Die möglicherweise überflüssig gewordenen Berechnungen können später von Standardoptimierungen des Übersetzerbaus entfernt werden, wie zum Beispiel der Eliminierung toten Codes oder div. Steuerflussoptimierungen. Letzteres ist nötig, da der reichhaltige Befehl auch mehrere Grundblöcke umfassen und somit gegebenenfalls obsolet werden lassen kann. Wie wir im nächsten Abschnitt sehen, wird entgegen des in Abbildung 7.8 vermittelten Eindrucks die Ersetzung allerdings nicht sofort durchgeführt, sobald eine Passung gefunden wurde, sondern es sind noch weitere Prüfungen der Passung außerhalb des Graphersetzungssystems nötig.

7.4.2.4 Algorithmische Steuerung

Die *algorithmische Steuerung* hat zweierlei Aufgaben: Zum Einen muss sie die Regelauswahl und zum Anderen die Passungsauswahl vornehmen. Allerdings ist es hier nicht möglich, das eine vollständig vom anderen zu trennen, da die Regelauswahl die möglichen Passungen beeinflusst und umgekehrt. Insgesamt haben wir ein Optimierungsproblem zu lösen, nämlich die Frage, welche Regelanwendung und welche Passungsauswahl die maximale Kostenersparnis nachsichzieht. Dieses Optimierungsproblem hat noch einige Randbedingungen, die im Wesentlichen daher rühren, dass wir die Muster generischer gemacht haben, als sie zur semantikerhaltenden Transformation (allein aufgrund der Graphersetzungsregeln) nötig wären. Allerdings wollen wir nicht auf den Zugewinn an Trefferwahrscheinlichkeit verzichten, weshalb der Korrektheit wegen die Passungen vor Anwendung der entsprechenden Regel nochmals geprüft werden müssen. Dazu nutzen wir eine besondere Eigenschaft von GrGen, die es erlaubt eine oder mehrere Passungen einer Regel zunächst zu finden und dann eine Passung für die Ersetzung auszuwählen.

Die Zulässigkeit im Kern ist immer dann nicht gegeben, wenn wir aus dem Datenfluss und dem Speicherzugriff durch die Ersetzung eine Schleife konstruieren, wodurch gewisse Befehle von ihren Ergebnissen – ohne dass dazwischen eine ϕ-Funktion wäre – abhängig sind. Dieser Zustand ist für SSA-Programme illegal. Analysen der Passung stellen jeweils fest, ob eine Ersetzung einen solchen Zustand herbeiführen würde, und verwerfen dann diese Passung.

Wir haben zwei Lösungswege für das Problem der Regel- und Passungsauswahl entwickelt: Einerseits kann das Problem wie auch die herkömmliche Befehlsauswahl als PBQP dargestellt und gelöst werden (siehe auch [EKS03, ES03, SE02]). Andererseits ist das Problem mit einem explorativen direkten Verfahren heuristisch lösbar. Dazu wird jede Ersetzung zum Schein insoweit durchgeführt, dass die `VProj`-Ecken

virtuell zum Arbeitsgraphen hinzugefügt werden; dadurch ist es möglich, alle darauf aufbauenden Ersetzungen auszuprobieren. Alle diese Operationen werden mitprotokolliert und so insbesondere die Kostenersparnis festgestellt. Am Schluss werden die besten Ersetzungen tatsächlich ausgeführt.

7.4.2.5 Normalisierende Vorverarbeitungsschritte

Für das Erkennen von reichhaltigen Befehlen wollen wir aus der syntaktischen Gleichheit auf semantische Äquivalenz schließen. Obschon Firm ebendiese Forderung als ein Designziel formuliert hat, reicht uns dies noch nicht weit genug. Gerade die in Abschnitt 7.2.1 angesprochenen Probleme würden dazu führen, dass zu wenig Code, der eigentlich reichhaltigen Befehlen entspricht, auch tatsächlich ersetzt würde. Wir benutzen hierzu normalisierende Vorverarbeitungsschritte, die auf den Zwischendarstellungsgraphen der Verhaltensbeschreibung des reichhaltigen Befehls, wie auch auf die Zwischendarstellungsgraphen der zu optimierenden Programme gleichermaßen einwirken.

Ein Teil dieser Vorverarbeitungsschritte sind Standardoptimierungen und Analysen, wie zum Beispiel: Das Entfernen kritischer Kanten, die Alias-Analyse mit dem Parallelisieren der Speicherkante, die If-Konversion etc. Andere Optimierungen sollten zum Zeitpunkt, an dem wir die reichhaltigen Befehle suchen, noch nicht durchgeführt worden sein, da sie Zwischendarstellungsgraphen mit vormals syntaktischgleichen Teilgraphen unterschiedlich werden lassen. Beispiel hierfür sind: Die Codeplatzierung, die PRE[12] und die Stärkenreduktion[13].

Zusätzlich haben wir spezielle normalisierende Vorverarbeitungsschritte eigens für natürliche Spezifikationen entwickelt. Wir führen beliebig stellige arithmetische und logische Operationen ein. Dies führt dazu, dass die Situation wie in Abschnitt 7.2.1 beschrieben nicht mehr auftritt, da nun eine willkürlich angeordnete Kaskade von Additions-Ecken zu einer einzigen Ecke zusammenfällt. Diese Darstellung, zumal in ausmultiplizierter Form, ist auch für andere Optimierungen nützlich, insbesondere sind für die Befehlsanordnung die Freiheitsgrade nun explizit.

[12] engl.: *partial redundancy elimination*
[13] engl.: *strength reduction*

TEIL III

Implementierung

KAPITEL 8

DER GRAPHERSETZER GRGEN

In diesem Kapitel stellen wir die Architektur und die Hauptkomponenten unseres Graphersetzungswerkzeugs GrGen und in Sonderheit seine neuste Inkarnation GrGen.NET vor [Hac03, Kro07, Jak08, Kro08]. Die Leitgedanken beim Entwurf von GrGen.NET und insbesondere von dessen Spezifikationssprachen waren folgende Punkte:

- SPO-Semantik für Ersetzung
- Keine Idiosynkrasien
- Strikte Typisierung unter Verwendung von Vererbung
- Ecken und Kanten in der Modellierung gleich behandeln
- Keine willkürlichen Einschränkungen
- Möglichst konsistente Syntax für alle Spezifikationssprachen (Graphmodell, Graphersetzungsregeln, Regelauswahl und Skriptsprache der GrShell)
- Möglichst striktes Durchhalten von Klammerung und Gültigkeitsbereichen
- Vermeidung von sprachinduzierten Fehlerquellen (Leitbild: Java / C# und nicht C / C++)

8.1 Architektur

Um die Spezifikationen eines deklarativen Graphersetzungssystems ausführbar zu machen, benutzen wir den in Abbildung 8.1 schematisch angegebenen Graphersetzungsgenerator GrGen. Dazu analysiert GrGen zunächst die Spezifikationen und repräsentiert diese intern entsprechend. Aus dieser Repräsentation werden die verschiedenen Teile eines Graphersetzers, also das reflexionunterstützende Typ- und Attributierungsmodell, der Mustersucher und der eigentliche Ersetzer, erzeugt. Es ist möglich diese drei Teile des Graphersetzers für unterschiedliche Systeme zur *Graphdatenhaltung* speziell zu generieren, was wir auch getan haben. Es gibt eine Integration der Graphdatenhaltung in unseren Firm-basierten Übersetzer, eine Anbindung der Graphdatenhaltung an die Datenbank PostgreSQL und je eine alleinstehende Implementierung in C und C#, wobei letztere wiederum zu Versuchszwecken in verschiedenen Varianten erstellt wurden.

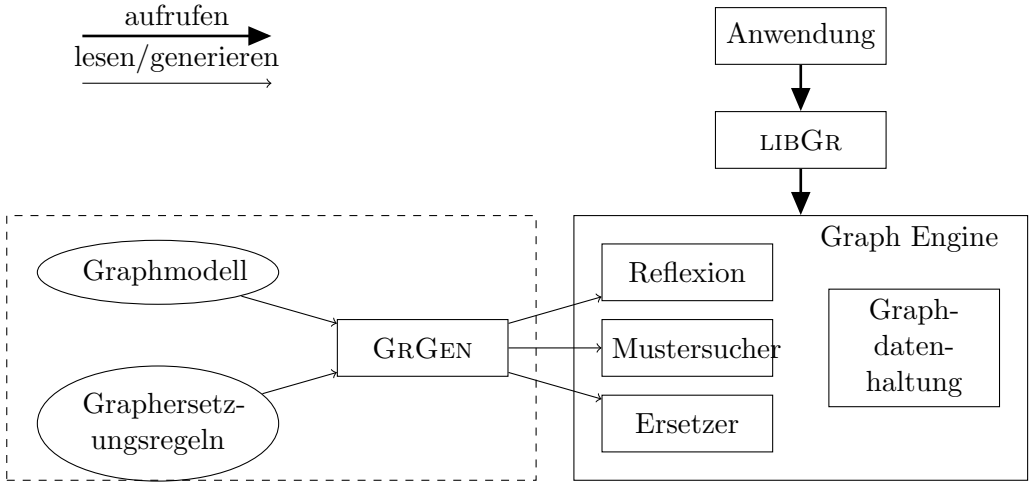


Abbildung 8.1: Übersicht über das Graphersetzungswerkzeug GrGen: Der Graphersetzungsgenerator GrGen generiert aus deklarativen Spezifikationen die drei Hauptbestandteile von Graphersetzern.

8.2 Die Komponenten

Hier geben wir einen Abriss der Funktion der Komponenten sowie Verweise auf detailliertere Ausführungen.

Ersetzer
: Der Ersetzer implementiert die XSPO-Semantik (siehe Abschnitt 4.3), d. h. er führt die eigentliche Ersetzung durch.

Mustersucher
: Wir haben zwei alternative Lösungen zur effizienten Mustersuche erarbeitet. Beide sind in Kapitel 5 dargestellt.

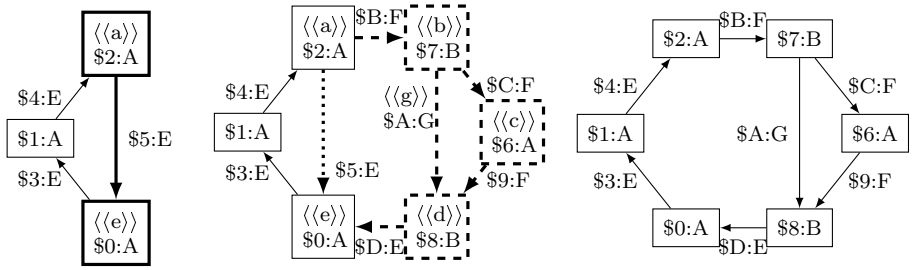
Reflexion
: Erlaubt den Zugriff auf Metainformationen des Arbeitsgraphen und der Regeln. Im Wesentlichen sind alle Informationen der entsprechenden Graphmodelle (siehe Abschnitt 4.1.1) verfügbar.

libGr
: Die libGr ist die Programmierschnittstelle unseres Graphersetzungswerkzeugs. Eine vollständige Dokumentation ist auf der Internetpräsenz von GrGen.NET zu finden: `www.grgen.net`.

Eine praktische Einführung, sowie eine Sprachreferenz der Eingabesprachen von GrGen ist im technischen Bericht von Blomer und Geiß [BG07] dargelegt; entsprechende Syntaxdiagramme finden sich in Anhang A.

8.3 GrShell

Die GrShell ist eine interaktive Umgebung zur Benutzung und Entwicklung von Anwendungen mit Graphersetzung (siehe Abbildung 8.3). Sie bietet die Möglichkeit,



(a) Die Passung (b) Die Änderungen (c) Das Ergebnis

Abbildung 8.2: Visualisierung einer Graphersetzung (abstrakte Darstellung des Debuggers der GrShell)

Programm 8.1: Eine Graphersetzungsregel zur Erstellung einer Kochkurve.

```
1 rule makeFlake1 {
2  pattern { a:Node -:E-> e:Node; }
3  replace { a -:F-> b:B -:F-> c:A -:F-> d:B -:F-> e;  b -g:G-> d; }
4 }
```

einzelne Graphersetzungsregeln oder auch ganze Graphersetzungssequenzen schrittweise ablaufen zu lassen. Dabei kann analog zu einem Debugger einer konventionellen Programmiersprache die Veränderung des Arbeitsgraphen in jedem Einzelschritt grafisch beobachtet werden; es ist insbesondere möglich, die gefundenen Passungen sowie die eigentliche Ersetzung zu visualisieren. Darüber hinaus sind die meisten Funktionen der libGr auf dieser Ebene zugreifbar.

Betrachten wir folgendes Beispiel um uns ein Bild von der Arbeitsweise der GrShell zu machen: Das Programm 8.1 erzeugt aus einem initialen Dreieck eine Kochkurve. Dabei seien `A` und `B` Eckentypen, die vom Wurzeltyp `Node` erben. Die Kantentypen `E`, `F` und `G` erben alle vom Wurzeltyp der Kanten: `Edge`. Wenn wir diese Regel auf einen Graphen, der ein geeignetes initiales Dreieck beinhaltet, anwenden und den *Debugmodus* aktivieren, dann sind die im folgenden dargestellten Schritte zu sehen, wobei die Abbildung der Graphelemente des Musters auf die des Arbeitsgraphen durch die entsprechenden Namen der Elemente in der Graphersetzungsregel (z. B. ⟨⟨a⟩⟩) angegeben wird.

1. Die Passung wird gefunden und markiert – dargestellt durch die dicken Linien in Abbildung 8.2(a).

2. Die neu zu erzeugenden und die zu löschenden Graphelemente werden hervorgehoben – dargestellt durch die gestrichelten oder gepunkteten Linien in Abbildung 8.2(b).

3. Der Ergebnisgraph wird angezeigt – siehe Abbildung 8.2(c).

Dies gibt dem Anwender tiefe Einsicht in den Vorgang der Graphersetzung und kann genutzt werden, um fehlerhafte Graphersetzungsregeln zu finden, gleich wie auch fehlerhafte Graphersetzungssequenzen getestet werden können. Darüber hinaus glauben wir, dass diese Visualisierung beim Erlernen der Graphersetzung nützlich ist.

Zum Abschluss noch ein Bildschirmfoto (siehe Abbildung 8.3), in dem ein etwas komplexeres Beispiel, nämlich das Programm 4.7 aus Abschnitt 4.3.8, gezeigt wird.

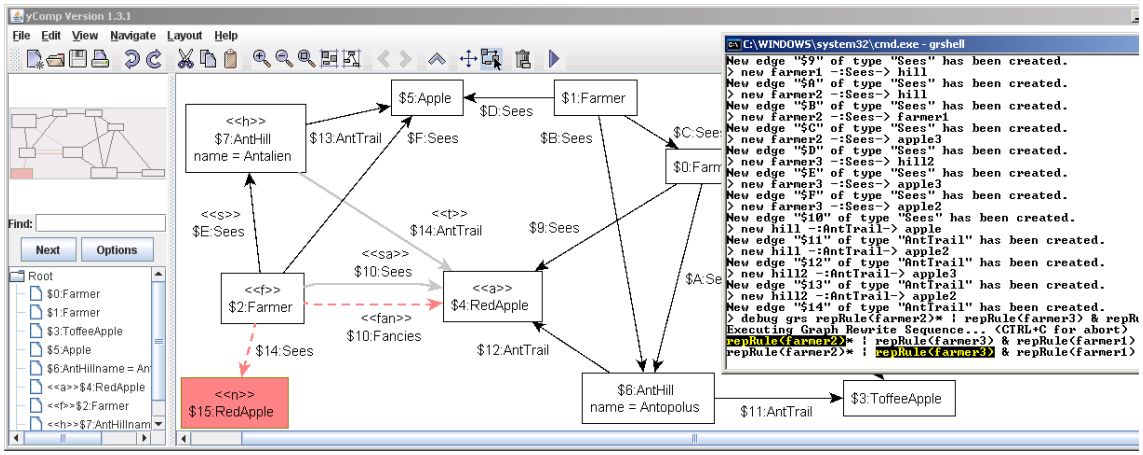


Abbildung 8.3: Bildschirmfoto der GrShell und unseres Graphanzeigesystems yComp im Debugmodus (die Farben wurden für den Druck optimiert)

KAPITEL 9

MESSUNGEN UND ERGEBNISSE

Dieses Kapitel fasst die praktischen Ergebnisse dieser Arbeit anhand von Beispielen und Messungen zusammen. Zuerst betrachten wir die Leistungsfähigkeit unseres Graphersetzers GrGen.

In Abschnitt 9.1 vergleichen wir die prominentesten Werkzeuge zur Graphersetzung mittels des von Varró entwickelten Mutex-Leistungstests. Dieser Leistungstest[1] wurde von Varró et al. im Jahre 2005 entwickelt und besteht aus einer Simulation von Teilen eines Problems mit gegenseitigem Ausschluss[2], weshalb er auch Mutex-Leistungstest heißt [VSV05a, VSV05b, VFV04]. Im Jahr 2005 war dies erstaunlicherweise der erste Leistungstest, der speziell für Graphersetzer entworfen wurde. Die ersten Veröffentlichungen von Messungen zum Mutex-Leistungstest [VSV05b, VFV04] sind leider von systematischen und zufälligen Messfehlern verfälscht; inzwischen liegen verbesserte Ergebnisse und Messmethoden vor [GK07].

Ein weiterer Abschnitt 9.2 beschäftigt sich mit einem Leistungstest, der von uns anlässlich der *AGTIVE Tool Challenge* entworfen wurde [GMK07] und der Sierpinski-Dreiecke mittels Graphersetzung generiert. Dieser Sierpinski-Leistungstest enthält noch einfachere Regeln als der Mutex-Leistungstest und testet so weniger die Fähigkeiten eines Graphersetzungswerkzeugs zur schnellen Mustersuche, als vielmehr deren Fähigkeiten im Umgang mit sehr großen Arbeitsgraphen[3].

Den ersten Teil schließt eine experimentelle Validierung unserer suchplanbasierten virtuellen Maschine (siehe Kapitel 5) zur Graphmustersuche ab. Dieser Test zeigt, dass mithilfe der von uns entwickelten Heuristik, schnelle, sehr effektiv von langsamen Suchprogrammen unterscheidbar sind.

Der zweite Teil dieses Kapitels benutzt die natürlichen Spezifikationen (vgl. Abschnitt 7.4.2) als Anwendungsfall, um die Leistungsfähigkeit von GrGen im Bezug auf den Übersetzerbau festzustellen. Hier kommen moderat große Arbeitsgraphen (bis zu einigen zehntausend Graphelementen) und extrem große Muster (bis zu mehreren hundert Graphelementen) zum Einsatz. Die Arbeitsgraphen haben alle eine spezielle Eigenschaft gemeinsam: Sie sind Programmgraphen; mehr dazu in Abschnitt 7.3. In Abschnitt 9.4 zeigen wir, wie effizient GrGen diese Aufgabe löst.

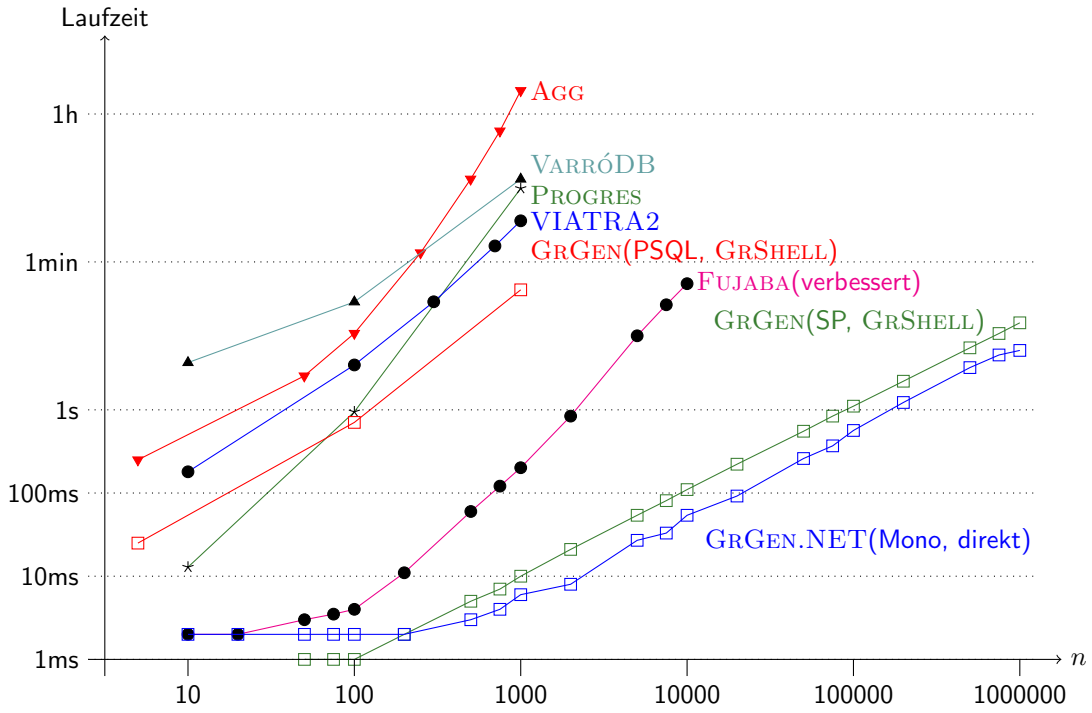


Abbildung 9.1: Laufzeiten des STSmany Leistungstest für verschiedene Graphersetzungswerkzeuge mit einer vorherigen Aufwärmphase der Größe $n = 1000$ für FUJABA und GRGEN.NET, um den Verfälschungen durch die Laufzeitübersetzung entgegenzuwirken.

Test →	STSmany			ALAP			ALAP simult.			LTS
Werkzeug ↓	10	100	1000	10	100	1000	10	100	1000	1000,1
PROGRES	12	946	459.000	21	1.267	610.600	8	471	2.361	942.100
AGG	330	8.300	6.881.000	270	8.027	13.654.000	–	–	–	$> 10^7$
FUJABA	40	305	4.927	32	203	2.821	20	69	344	3.875
VARRÓDB	4.697	19.825	593.500	893	14.088	596.800	153	537	3.130	593.200
GRGEN(PSQL)	30	760	27.715	24	1.180	406.000	–	–	–	96.486
GRGEN(SP)	< 1	1	12	< 1	1	11	< 1	< 1	9	21

Tabelle 9.1: Laufzeiten verschiedener Varianten des Varró-Leistungstests (in ms)

9.1 Mutex-Leistungstest

Der Leistungstest benutzt verschieden große Mustergraphen und mehrere Regeln, die allesamt eher kleine Mustergraphen haben. Er basiert auf einer von Varró für Zwecke der Leistungsmessung präparierten Simulation eines verteilten Systems mit gegenseitigem Ausschluss. Varró definiert mehrere Varianten des Leistungstests, die jeweils spezielle Eigenschaften der Werkzeuge testen sollen. Dabei wird besonders auf Optimierungen der Mustersuche abgezielt. Wir verwenden hier hauptsächlich die STSmany Variante. Dieser STSmany Leistungstest lässt sich mit den geeigneten Regeln als folgende Graphersetzungssequenz darstellen:

```
newRule[n-2] & mountRule & requestRule[n] &
  (takeRule & releaseRule & giveRule)[n]
```

[1]engl.: *benchmark*
[2]engl.: *mutual exclusion*
[3]Wenn es der Speicher zulässt, sprechen wir hier von zig Millionen Graphelementen.

Bei allen anderen Varianten verändert sich das im folgenden Dargestellte nur unwesentlich; die Verhältnisse zwischen den einzelnen Werkzeugen bleiben erhalten. Unsere eigenen Messungen (die für AGG, GRGEN, GRGEN.NET, FUJABA und VIATRA2) haben wir auf einem AMD Athlon XP 3000+ mit 1 GiB Hauptspeicher ausgeführt. Genauere Informationen zum verwendeten Rechner sind in Anhang B.5 zu finden.

Die Messungen, die Varró et al. ausgeführt haben, wurden auf einem Intel Pentium 4 mit 1.5 GHz sowie 768 MiB Hauptspeicher durchgeführt [Var05]. Die Wiederverwendung erspart uns die, aufgrund des Alters von PROGRES, extrem aufwendige Messung. Ferner ermöglicht es Zahlen für VARRÓDB zu präsentieren, da dieses prototypische Werkzeug nicht vollständig veröffentlicht vorliegt. Um die Messungen von Varró et al. wiederzuverwenden haben wir seine Zahlen mit 0.68 multipliziert. Dieser Faktor entspricht genau der Geschwindigkeitsdifferenz der beiden Rechner gemäß der Messungen der SPEC Organisation [Sta05]. Allerdings kommt es auf die genaue Zahl nicht an – selbst wenn wir uns um den vollkommen unwahrscheinlichen Faktor 5 geirrt haben sollten – ändert sich an den hier getroffenen Aussagen nichts (vgl. Abbildung 9.1).

Die Abbildung 9.1 zeigt die Laufzeiten von drei GRGEN Implementierungsvarianten verglichen mit den prominentesten Graphersetzungswerkzeugen, nämlich AGG [ERT99], FUJABA [Fuj05], PROGRES [Sch99], VIATRA2 [BNS+05, BV06] und einem Ansatz von Varró et al. [VFV04], den wir VARRÓDB nennen. GRGEN(SP) bezeichnet eine ältere C basierte Implementierung von GRGEN, deren Mustersuche – genauso wie die neue in C# implementierte Variante – auf Suchplanung beruht. Die GRGEN(PSQL) Variante bildet die Repräsentation der Arbeitsgraphen sowie die Mustersuche auf relationale Algebra ab. Die konkrete Implementierung verwendet das relationale Datenbankmanagementsystem POSTGRESQL[4] um die relationalen Anfragen zu bearbeiten. Schließlich ist GRGEN.NET eine Implementierung in C#, die die Fähigkeiten, welche in dieser Arbeit beschrieben werden, fast vollständig beinhaltet.

Man beachte die doppelt logarithmische Achsenteilung in Abbildung 9.1. Dies führt dazu, dass jede polynomielle Funktion asymptotisch als Gerade erscheint. Die Steigung entspricht dem dominierenden Polynomgrad, der Achsenabschnitt konstanten Faktoren. Kurven, die asymptotisch stärker wachsen als eine Gerade, entsprechen mindestens exponentielle Funktionen. Der Knick der Kurven von GRGEN.NET, GRGEN(SP) und FUJABA bei der Problemgröße $n = 100$ geht auf die dort erreichte Grenze der Auflösung der Systemuhr zusammen mit der Medianbildung aus 40 Messungen sowie dem Mehraufwand für das Starten einer Anwendung zurück.

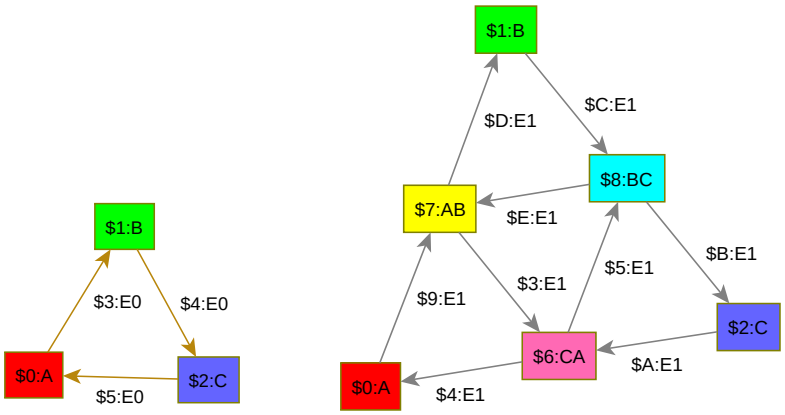
Durch die Eigenschaften der doppelt logarithmischen Achsenteilung wird augenfällig, dass GRGEN(SP) und GRGEN.NET um mindestens eine *Komplexitätsklasse* besser abschneiden, als alle anderen Werkzeuge. Eine genauere Analyse zeigt, dass das nächstschnellste Graphersetzungswerkzeug FUJABA eine quadratische Laufzeit ausweist, während GRGEN(SP) und GRGEN.NET linear sind. Da die Größe des Arbeitsgraphen genauso wie die Anzahl der Regelanwendungen linear mit der Problemgröße n wächst, ist es erstaunlich, dass die Laufzeit von GRGEN nur linear mit n wächst. Genauere Analysen zeigen hier, dass das Suchen einer einzelnen Passung, obgleich der Arbeitsgraph immer größer wird, nur konstant viel Zeit benötigt.

[4]Das relationale Datenbankmanagementsystem POSTGRESQL und das Graphersetzungswerkzeug PROGRES haben, obgleich sie zum Verwechseln ähnliche Namen haben, nichts miteinander zu tun.

Dies ist möglich, da durch die Suchraumfortschaltung (siehe Abschnitt 5.2.4) jeweils passende Arbeitsgraphelemente sofort zur Verfügung stehen – ohne den gesamten Arbeitsgraphen durchsuchen zu müssen.

Das langsamste Werkzeug AGG benötigt offenbar superpolynomielle (wahrscheinlich exponentielle) Laufzeit. Dies ist ebenfalls verwunderlich: Die Laufzeit für einen naiven Algorithmus, der alle möglichen Passungen durch vollständige Suche bestimmt, hängt, selbst im schlechtesten Fall, polynomiell von n ab, da der Mustergraph von konstanter Größe ist.

Wir sollten hier nicht aus den Augen verlieren, dass die Aufgabe der Mustersuche prinzipbedingt nicht immer so effizient geleistet werden kann. Die suchplanbasierte Mustersuche hat sich im Fall des Mutex-Leistungstests offenbar bestens bewährt. Weitere Ergebnisse auch für andere Varianten des Mutex-Leistungstests (abgesehen von STSmany) sind in Tabelle 9.1 zu sehen. Der ALAP stellt eine Variante dar, die eine lange Ersetzungssequenz mit mehr als doppelt so vielen Regeln wie STS hat. Die LTS-Variante arbeitet mit zwei Parametern, wobei nach Varró nur die Parameter 1000, 1 vermessen werden. Die Zahlen aus Tabelle 9.1 bestätigen ebenfalls, dass GRGEN.NET bzw. GRGEN(SP) für den Mutex-Leistungstest das schnellste gemessene Graphersetzungswerkzeug ist und überdies in diesem Leistungstest eine lineare Zeitkomplexität hat.



(a) Initiales Sierpinski Dreieck

(b) Erste Generation des Sierpinski Dreieck

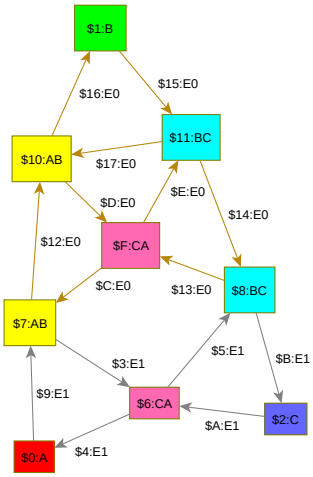
Abbildung 9.2: Initiales Dreieck und erste Generation

9.2 Sierpinski-Leistungstest

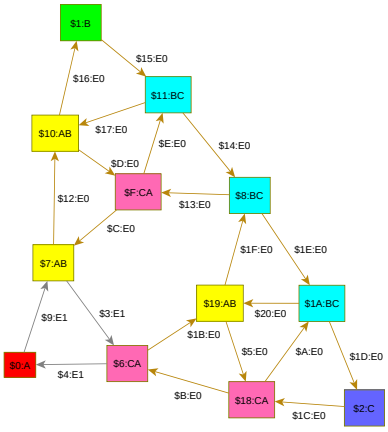
Das Ziel des sogenannten Sierpinski-Leistungstests ist es, ein Sierpinski Dreieck[5] mittels Graphersetzung zu konstruieren [GMK07, TBB+08]. Normalerweise ist ein Sierpinski Dreieck eine geometrische Figur und inhärent in die zweidimensionale Ebene eingebettet. Wir reformieren hier darum zunächst den Begriff des Sierpinski Dreiecks für abstrakte Graphen, die ohne (unmittelbare) Einbettung in die Ebene auskommen müssen.

Als Erstes müssen wir ein Dreieck als Graph repräsentieren. Dies erreichen wir dadurch, dass wir drei Ecken durch drei Kanten zu einer Struktur verbinden, die in

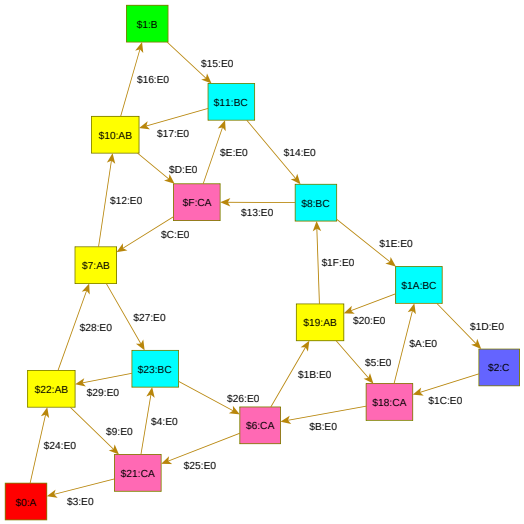
[5]Nach einer Veröffentlichung des polnischen Mathematikers Waclaw Franciszek Sierpinski [Sie15].



(a) Erster Sierpinski-Schritt



(b) Zweiter Sierpinski-Schritt



(c) Dritter Sierpinski-Schritt

Abbildung 9.3: Schrittweise Konstruktion der zweiten Generation des Sierpinski Dreiecks

Abbildung 9.2(a) zu sehen ist. Als Nächstes führen wir einen elementaren Sierpinski-Schritt durch: Dieser „verfeinert“ das eine Dreieck durch ein jeweils verschränkt eingesetztes weiteres Dreieck (Abbildung 9.2(b)). Eine Generation ist abgeschlossen, wenn alle Teildreiecke, die zu Beginn einer Generation existierten, durch einen elementaren Sierpinski-Schritt verfeinert wurden. Die Abbildung 9.3 zeigt alle Schritte für die zweite Generation.

Die Zahl der Generationen ist der laufende Parameter n des Leistungstests. In seiner Abhängigkeit vergrößert sich der Graph exponentiell (asymptotisch zur Basis von 3). Das heißt nach nur wenigen Generationen (konkret sind es $n = 12$) hat der Arbeitsgraph mehr als eine Million Elemente. Die Zahl der Ecken $|E|$ in einem Graphen der Generation n ist durch $\frac{3}{2}(1 + 3^n)$ bestimmt. In der *On-Line Encyclopedia of Integer Sequences* von N. J. A. Sloane [Slo07] hat diese Zahlenfolge die Folgennummer `A067771` [Wes07].

Tabelle 9.2 enthält die Messwerte des Sierpinski Leistungstests für GrGen.NET, wobei wir wieder unseren Leistungstest-Rechner verwendet haben (für Details sie-

Generation	$\|E\|$	$\|K\|$	Schritte	Laufzeiten [ms]	Speicher [MiB]
0	3	3	0	0	9
1	6	9	1	0	9
2	15	27	4	8	9
3	42	81	13	15	9
4	123	243	40	16	9
5	366	729	121	16	9
6	1,095	2,187	364	16	9
7	3,282	6,561	1,093	16	10
8	9,843	19,683	3,280	23	11
9	29,526	59,049	9.841	62	14
10	88,575	177,147	29,524	172	23
11	265,722	531,441	88,573	953	52
12	797,163	1,594,323	256,720	5,015	145
13	2,391,486	4,782,969	797,161	18,172	404
n	$\frac{3}{2}(1+3^n)$	$3^{(n+1)}$	$\frac{1}{2}(3^n-1)$		

Tabelle 9.2: Leistungsdaten für GrGen.NET und den Sierpinski-Leistungstest

he B.5) Wir testen hier, wie viele Graphelemente GrGen.NET repräsentieren kann und wie effizient mit sehr großen Graphen umgegangen wird. Aus diesem Grund haben wir die Anzahl der Graphelemente in Beziehung zu Speicherverbrauch und Laufzeit gesetzt, die gebraucht wurden, den jeweiligen Graphen zu erzeugen. Der Idealfall ist konstante Laufzeit und konstanter Speicherverbrauch pro Match und Graphelement. Tatsächlich erreichen wir nach einer Einschwingzeit nahezu konstante Werte und zwar 60 Byte pro Graphelement und 2 μs pro Finden einer Passung.

9.3 Suchplanung

Die experimentelle Validierung der Suchplanung steht in diesem Abschnitt im Mittelpunkt. Die in Kapitel 5 dargelegte suchplanbasierte Mustersuche ist in GrGen.NET implementiert. Wir untersuchen hier, wie die Geschwindigkeit der Mustersuche für ein Suchprogramm und die berechneten Kosten für dieses Programm zusammenhängen.

Dazu haben wir für zwei Regeln des STSmany-Leistungstests alle Suchpläne erzeugt und diese dann ausgeführt. Wir haben dabei jeweils die Laufzeit des gesamten Leistungstests (also der Graphersetzungssequenz aus Abschnitt 9.1) ermittelt, wobei natürlich alle anderen Suchprogramme identisch blieben. Die Programme der anderen Regeln in der Graphersetzungssequenz sind, die von unserem System automatisch bestimmten. Wir haben den STSmany-Leistungstest mit dem Parameter $n = 100000$ durchgeführt. Die einzelnen Messungen für ein Suchprogramm wurden jeweils bei 10 Sekunden abgebrochen.

Die Abbildungen 9.4 und 9.5 zeigen die Laufzeiten der verschiedenen Graphersetzungssequenzen, wobei jeweils alle Suchprogramme für die `takeRule` und für die `giveRule` getestet wurden. Auf der zweiten Y-Achse sind die Kosten des jeweiligen Suchprogrammes $c(P)$ und die Kosten der jeweiligen Auswahl an Suchbefehlen $c(S)$ dargestellt. Unsere Heuristik bestimmt zuerst die Auswahl an Suchbefehlen –

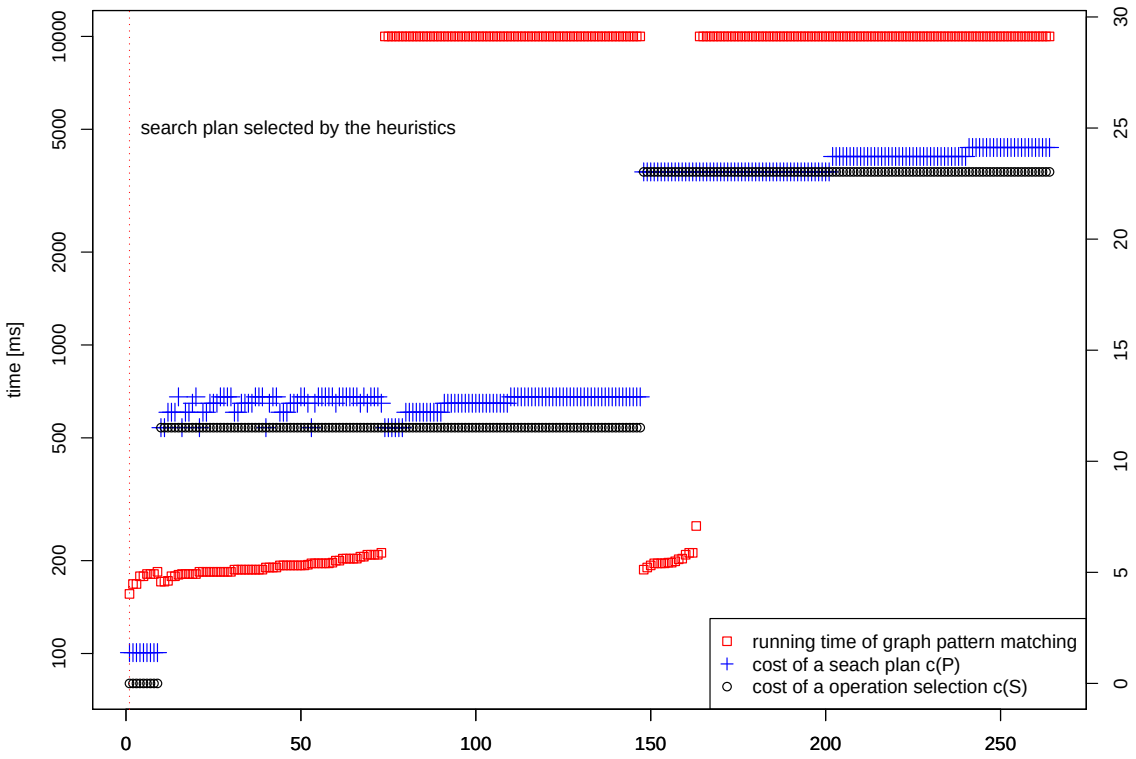


Abbildung 9.4: Laufzeit/Kosten-Diagramm für die `takeRule`.

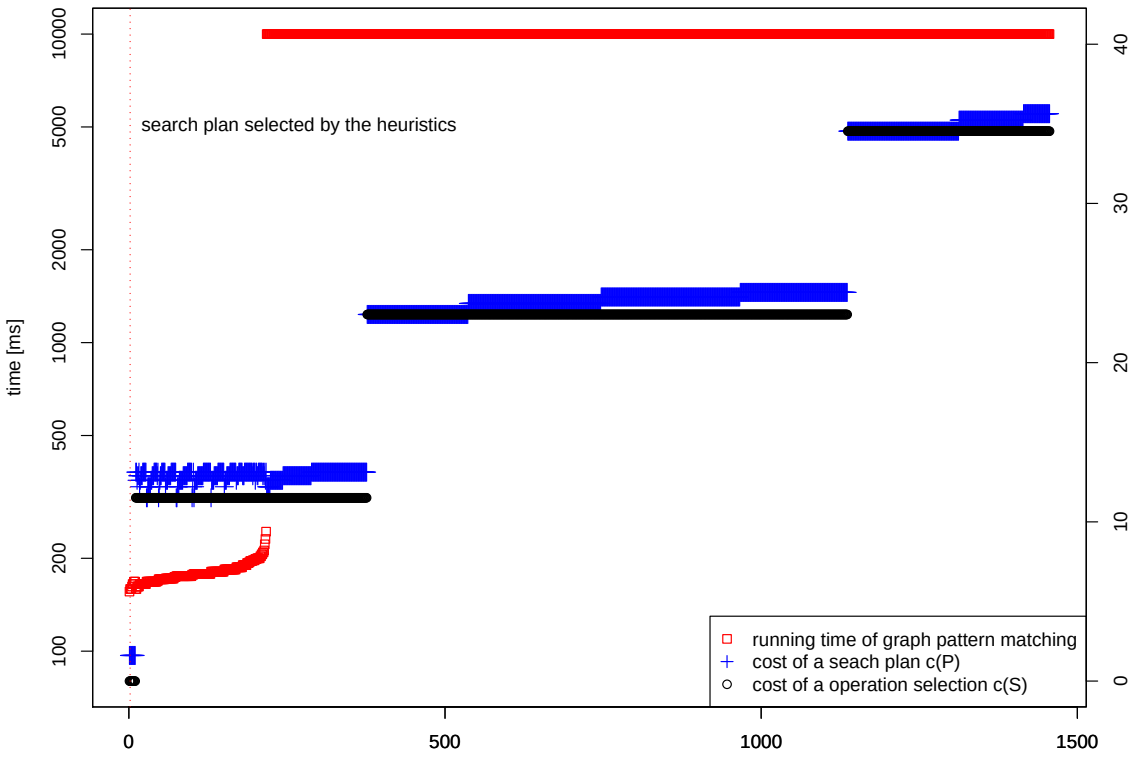


Abbildung 9.5: Laufzeit/Kosten-Diagramm für die `giveRule`.

dafür wird $c(S)$ benutzt – dann wird in einem zweiten Schritt diese Auswahl an Suchbefehlen zu einem Suchprogramm angeordnet. Wir haben die Suchprogramme, deren Nummer auf der X-Achse keinerlei Bedeutung hat, sondern nur zum Abschätzen der Anzahl der Programme dienen soll, zuerst nach den Kosten $c(P)$ des Suchprogrammes und dann nach der Laufzeit sortiert. Dadurch wird klar, aus welchen Programmen unsere Heuristik überhaupt hätte auswählen können. Es ist klar, dass die Kosten eines Suchprogramms immer größer sind, als die einer Auswahl an Suchbefehlen, da sie den größten Summanden gemeinsam haben und außerdem die Kosten nicht negativ sind. Durch eine gestrichelte vertikale Linie ist angedeutet, welches Suchprogramm von der Heuristik tatsächlich ausgewählt wurde.

Diese Experimente zeigen, dass sich unsere Heuristik beim STSmany-Leistungstest, recht eindrucksvoll schlägt. Denn obwohl die Laufzeiten der Suchprogramme großen Schwankungen unterlegen sind (die Mehrzahl benötigt mehr als 10 Sekunden), sind auf der geringsten Kostenstufe von $c(S)$ jeweils ausschließlich – im Sinne der realen Laufzeit – sehr gute Suchprogramme zu finden. Somit wählt die Heuristik immer ein gutes Programm aus. Andererseits gibt es auch schnelle Suchprogramme, die recht große Kosten haben (siehe hierzu Abbildung 9.4 und besonders die Pläne mit den Nummern von ca. 150 bis 170). Genauere Analysen haben gezeigt, dass dieser Effekt von der Suchraumfortschaltung (Abschnitt 5.2.4) hervorgerufen wird, der eigentlich lineare oder quadratische Suchprogramme mit konstanter Laufzeit ablaufen lässt, da die gesuchten Graphelemente immer als Erstes betrachtet werden.

Natürlich kann aufgrund der NP-Vollständigkeit der Graphmustersuche unsere Heuristik nicht immer so perfekt funktionieren wie im Fall des Varró Leistungstests. Jedoch hat unsere Erfahrung gezeigt, dass sehr viele praktische Probleme in der Klasse von Mustern und Arbeitsgraphen liegen, die unsere virtuelle Maschine für die Graphmustersuche sehr effizient zur Passung bringen kann.

9.4 Natürliche Spezifikation von reichhaltigen Befehlen

Der Testfall für die Optimierung, die auf der *natürlichen Spezifikation* von reichhaltigen Befehlen aufbaut (siehe Abschnitt 7.4.2), ist aus einem Videocodec, genauer der sogenannten Bewegungsschätzung[6], entnommen. In Programm 9.1 ist die Kernfunktion der Bewegungsschätzung, die Berechnung von absoluten Differenzen auf zweidimensionalen Feldern, zu sehen. Wir wenden unsere Optimierung auf dieses Programm an, wodurch es wie folgt verändert wird: Die Zwischendarstellung des unoptimierten Programms 9.1 mit 16-fach ausgerollter innerer Schleife hat 2.392 Ecken und 5.324 Kanten, wodurch sie sich hier nicht mehr sinnvoll abbilden lässt. Unsere Optimierung schafft es, in 1,2 Sekunden einen Großteil der Zwischendarstellung durch vier reichhaltige Befehle zu ersetzen. Die eigentliche Graphmustersuche geschieht in 40 ms, der Rest der Laufzeit setzt sich aus zusätzlichen Analysen des Übersetzers zusammen. Insgesamt werden 51 Passungen berechnet und analysiert. Dabei werden 4.480 Ecken und 17.893 Kanten zur Passung gebracht (hier sind natürlich Wiederholungen eingeschlossen). Nach der Optimierung hat der Graph noch 680 Ecken und 1.480 Kanten. Weitere Ecken werden von anschließenden Optimierungen als nunmehr nutzlos erkannt und entfernt.

[6]engl.: *motion estimation*

Programm 9.1: Berechnung von absoluten Differenzen

```
1  unsigned int sad(int test_blockx, int test_blocky, int *best_block_x,
2                   int *best_block_y, unsigned char frame[256][256])
3  {
4     int i, x, y, blocky, blockx;
5     unsigned tmp_diff, min_diff = 0xFFFFFFFF;
6
7     // Iterate over whole frame, x,y=coords of current block
8     for(x = 1; x < 256 - 16; x++)
9        for(y = 0; y < 256 - 16; y++)
10       {
11          tmp_diff = 0;
12
13          // Compare current Block with reference block
14          for(blocky = 0; blocky < 16; blocky++)
15          {
16             for(blockx = 0; blockx < 16; blockx++)
17                if(a[blocky][blockx] > b[blocky + y][blockx])
18                   tmp_diff += (frame[blocky][blockx] - frame[blocky + y][blockx]);
19                else
20                   tmp_diff += (frame[blocky + y][blockx] - frame[blocky][blockx]);
21          }
22
23          // Check if the current block is least different
24          if(min_diff > tmp_diff)
25          {
26             min_diff = tmp_diff;
27             *best_block_x = x;
28             *best_block_y = y;
29          }
30       }
31    }
32    return(min_diff);
33 }
```

Wir haben die unoptimierte Version mit unserem Firm-basierten Übersetzer[7] übersetzt und die Laufzeit von 100 Durchläufen des Programms 9.1 aufsummiert. Inklusive der zufälligen Initialisierung der Argumente ergeben sich so 13,550 Sekunden. Das Programm, das durch unseren Firm-basierten Übersetzer mit unserer graphersetzungsbasierten Optimierung für reichhaltige Befehle erzeugt wurde, benötigt nur 0,420 Sekunden für den gleichen Code. Dies ist eine Beschleunigung um mehr als den Faktor 32.

Damit ist einerseits eindrucksvoll gezeigt, dass die Optimierung selbst recht effizient ist, denn sie benötigt weniger als 2 Sekunden, und andererseits, dass die von ihr durchgeführte Ersetzung die Ausführung des zu optimierenden Programms erheblich beschleunigt.

[7]Die Laufzeit bleibt nahezu unverändert bei Verwendung des gcc (bis Version 4.3, Option -O3)

KAPITEL 10

ZUSAMMENFASSUNG UND AUSBLICK

Zusammenfassung

Wir haben gezeigt, dass eine theoretisch fundierte, effizient maschinell ausführbare und einfach benutzbare Methodik zur Graphersetzung realisierbar ist. Im Rahmen dieser Arbeit ist ein Graphersetzungswerkzeug entstanden, für das die zentralen Teile oft mehrfach mit verschiedenen Ansätzen umgesetzt wurden. Das Für und Wider der verschiedenen Ansätze wurde diskutiert; die jeweils besten Methoden flossen in das endgültige Werkzeug ein. Als Ergebnis ist ein besonders effizientes und leicht benutzbares Werkzeug zur Graphersetzung entstanden: GrGen.NET.

Es wurden aufeinander abgestimmte, deklarative Spezifikationssprachen für Graphmetamodelle, Graphmustersuche, Graphersetzung sowie Regelauswahl entwickelt. Über die traditionelle Ausdrucksstärke hinaus können jetzt Mustergraphen mit zusätzlichen Bedingungen (typisierte Graphen mit dynamischen Typbedingungen, Attributbedingungen, Vorbelegungen in Form von parametrisierten Mustern, negative Anwendungsbedingungen und Homomorphiebedingungen) sowie dynamischen Mustern versehen werden. Der Ersetzungsschritt ist um Reattributierungsanweisungen, Retypisierung, dynamische Typisierung, Kopieranweisungen und die Rückgabe von Arbeitsgraphelementen erweitert worden. Für die Semantik der erwähnten Sprachen existiert eine solide theoretische Fundierung. Bis auf den Kern der Semantik des Ersetzungsschritts, der auf dem wohlbekannten, kategorientheoretischen Single-Pushout Ansatz (SPO) beruht, wurden die Fundamente dieser Sprachen im Rahmen dieser Arbeit entwickelt. Eine Neuerung bei der theoretischen Fundierung ist, dass negative Anwendungsbedingungen direkt mittels elementarer Begriffe der Homomorphie gefasst werden können.

Um trotz der NP-Vollständigkeit der Mustersuche für viele praktisch relevante Fälle Passungen in polynomieller – oft sogar linearer – Zeit zu finden, konstruieren wir eine virtuelle Maschine, deren Programme verschiedenen Suchstrategien für einen Mustergraphen entsprechen. Aus der Menge dieser Programme werden anhand verschiedener Kostenmodelle „gute“ Programme selektiert, wobei die Kosten aus Domänenwissen oder aus der Analyse des vorliegenden Arbeitsgraphen stammen können. Fernerhin kann die virtuelle Maschine die Programme zur Laufzeit neu optimieren und mit Techniken der dynamischen Programmierung den Suchraum heuristisch weiter beschneiden. Damit gelingt es für Graphen, bei denen alle V-Strukturen – diese führen zur exponentiellen Ausweitung des Suchraums – bei der Mustersuche umgehbar sind, linearen Suchaufwand im Bezug auf die Größe des Mustergraphen zu garantieren. Für Graphen, die bezüglich dieses Kriteriums „komplex“ sind, gelingt es immer noch, den Suchaufwand heuristisch zu minimieren [BKG08].

Weitere Heuristiken, die spezielle Eigenschaften der Muster- und Arbeitsgraphen nutzen, sind integriert.

Die Leistungsfähigkeit unserer Graphmustersuche wird durch GrGen, des weltweit schnellsten automatischen Graphersetzungswerkzeugs, dokumentiert. Bei gleicher Ausdrucksstärke und z. T. sogar identischem, also SPO-basiertem Ansatz, ist GrGen bezüglich des von Varró entwickelten Mutex-STS-Leistungstests mindestens eine Komplexitätsklasse schneller (lineare gegenüber quadratischer Laufzeit) als das nächstschnellste Werkzeug [GBG+06].

Die Entwicklung von Graphersetzungsanwendungen wird durch eine interaktive Umgebung unterstützt, die, vergleichbar mit einem Debugger für konventionelle Programmiersprachen, das schrittweise Ausführen, Visualisieren und Komponieren von Regelsequenzen ermöglicht. Die Graphersetzungssequenzen erweisen sich dabei als einfache, aber für viele Bereiche hinreichende Spezifikationstechnik zur Regelauswahl.

Anwendungen im Übersetzerbau zeigen die Integrierbarkeit sowie die praktische Leistungsfähigkeit der hier vorgestellten Methode. Die in GrGen spezifizierten Optimierungen und die Erweiterungen für die optimierende Übersetzung von Programmen sowie die neuartigen Methoden zur Graphmustergewinnung, der sogenannten natürlichen Spezifikation, belegen, dass mit Graphersetzung bekannte Aufgaben effektiv und effizient behandelt werden können. Darüber hinaus werden neue Ansätze erst durch Graphersetzung realisierbar [SG08]. Mit natürlichen Spezifikationen ist es erstmals dem Benutzer eines Übersetzers möglich, maschinennahe Optimierungen zu spezifizieren, ohne selbst Übersetzerbauer sein zu müssen. Am Beispiel der SSE-Befehlssatzerweiterung von Intel-Prozessoren wurde gezeigt, dass dies zu erheblichen Beschleunigungen der übersetzten Programme führen kann (mehr als Faktor zehn) – ohne den Übersetzungsvorgang stark zu verlangsamen (wenige Sekunden). Neben dem Einsatz im Übersetzerbau existieren weitere Anwendungen von GrGen: Modelltransformation und Softwaretechnik (T. Gelhausen [GT07, DGG08, GDG08]), Logistik und Simulation (Prof. Kreowski und K. Hölscher) und Chemie (M. Yadav).

Insgesamt können mit der hier entwickelten Methodik (exemplifiziert durch das Graphersetzungswerkzeug GrGen) Graphersetzungsanwendungen elegant erstellt, effektiv getestet und dank effizienter Mustersuche ökonomisch sinnvoll in Anwendungen integriert werden.

Ausblick

Die offenen theoretischen Fragestellungen und die noch zu leistende Arbeit sehen wir vor allem auf drei Gebieten: Leistungstests, Suchplanung und Anwendungen.

Die *Leistungstests* sind bis zum Jahr 2005 mit dem Erscheinen des Mutex-Leistungstests von Varró ausschließlich ad hoc entstanden und nur ein einziges Mal für eine Veröffentlichung benutzt worden. Eine unabhängige Überprüfung der Ergebnisse solcher Messungen ist uns in der Literatur zur Graphersetzung nicht bekannt. Ein Vergleich verschiedener Werkzeuge aufgrund von Leistungsdaten war damit offenbar unmöglich. Diese Situation ist bedauernswerterweise mit dem Mutex-Leistungstest von Varró nur abgemildert, aber nicht behoben worden: Der Mutex-Leistungstest ist viel zu simpel. Hier sollten, ähnlich zur *Standard Performance Evaluation Corporation* (SPEC) bei Rechenanlagen oder dem *Transaction Processing Performance Council* (TPC-C/TPC-E) für Datenbanken auch Standards für Graphersetzungsleistungstests erarbeitet werden. Da jeder Leistungstest zu einem

Zerrbild der Realität ausarten kann, sollten wir gerade darum versuchen, ihn mit möglichst großem Realitätsbezug zu entwerfen. Gerüstet mit einem solchen Leistungstests sollte die Verbesserung der Werkzeuge im Bezug auf Ausdrucksstärke, Les- sowie Erweiterbarkeit der Spezifikationen und vor allem der Geschwindigkeit besser gelingen. Die Nutzer können so leichter erkennen, dass Graphersetzung im realen Einsatz ernst zu nehmende Aufgaben lösen kann, und welches der Werkzeuge sich zur Lösung eines gegebenen Problems besonders eignet.

Obgleich wir bei der *Suchplanung* erhebliche Fortschritte bei der Leistungsfähigkeit der tatsächlichen Implementierung, aber auch der theoretischen Fundierung der Heuristik gemacht haben, sind bei weitem noch nicht alle Aspekte untersucht: Es gibt keine Studie zur Frage, welche Kosten, und welche Heuristik zu deren Minimierung, für Suchprogramme angemessen sind. Wir haben bis dato nur zwei Vorschläge: den von Varró [VVF05] sowie den von Batz und Geiß [Bat06, BKG08]. Weitere sind jedoch denkbar, insbesondere im Bezug auf die Frage, ob sich Kosten finden lassen, die nicht mehr nur eine Kante und ihre inzidenten Ecken berücksichtigen. Möglicherweise kann man zu einer Sicht kommen, die den lokalen Kontext eines Graphen analysieren und klassifizieren kann, um so gewisse Teile des Graphen von der Suche auszuschließen. Die Frage, wie die Verteilung von Ecken, Kanten und Typen in Graphen die Güte der Planungsheuristik beeinflusst, ist nicht abschließend geklärt. Kann man die „gutartigen“ Graphklassen analytisch statt phänomenologisch beschreiben? Wie viele Suchprogramme gibt es bei gegebenem Muster überhaupt? Was hat darauf einen Einfluss? Wie viele der Suchprogramme sind bei welchen Arbeitsgraphen schlecht? Wie kann man die virtuelle Maschine optimieren oder die einmal gewählten Suchprogramme über das Bekannte hinaus verbessern?

Wir gehen davon aus, dass viele *Anwendungen* vom Zugewinn an Abstraktion, den die deklarative Graphersetzung bietet, profitieren können. Durch unser Werkzeug ist es nun möglich, automatisiert effiziente Implementierungen aus diesen abstrakten Spezifikationen zu erhalten. Dabei denken wir vor allem an Beispiele aus der Biologie, der Chemie, dem Verkehrswesen, den Datenbanken, der allgemeinen Programmierung mittels Graphersetzung und natürlich der Softwaretechnik. Weitere Anwendungen im Übersetzerbau halten wir für interessant, da denen unsere Vorarbeiten bei der Integration eines Graphersetzers in einen Übersetzer besonders zuträglich wären.

Zum Schluss ist es mir noch ein besonderes Anliegen, mich für die Aufnahme der Graphersetzung in den Kanon der Lehre auszusprechen. Graphersetzung ist eine Technik, die es erlaubt, viele Probleme auf einer angemessen hohen Abstraktionsebene zu beschreiben – und effizient zu lösen. Darum sollte Graphersetzung, ähnlich wie relationale Algebra oder Prädikatenlogik, im Standardrepertoire eines Informatikers nicht fehlen.

ANHANG A

DIE GRGEN-SPEZIFIKATIONSSPRACHEN

In diesem Anhang stellen wir die Spezifikationssprachen von GrGen anhand von Syntaxdiagrammen[1] kurz vor. Für eine ausführliche Beschreibung sei auf das Benutzerhandbuch [BG07] von GrGen.NET verwiesen. Syntaxdiagramme sind eine Visualisierung EBNF[2]-Grammatiken. Wenn man einem Pfad durch das Diagramm folgt, denn ergeben sich die gültigen (Teil-)Sätze der Sprache. Ellipsen stehen für Terminale und Rechtecke für Nichtterminale. Mehr zu Syntaxdiagrammen ist in einem Buch von Wirth, dem Erfinder dieser Diagramme, zu finden [MMJW91].

A.1 Gemeinsame Elemente

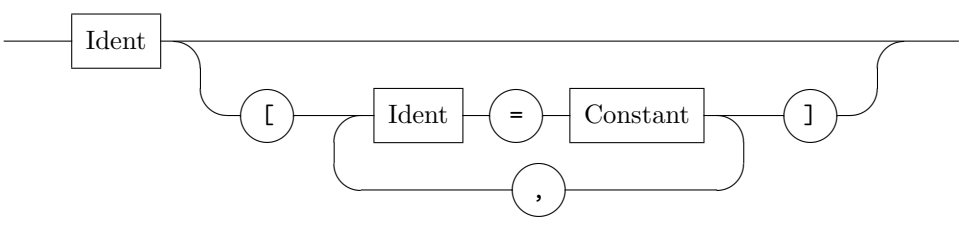
Number
: Konstanten des Wertebereichs des Typs `int`.

Constant
: Konstanten der Wertebereiche der Typen `boolean, int, string, float, double, object`.

Ident, NodeType, EdgeType, ModelIdent, ActionIdent
: Ein Bezeichner, wobei die Namen die jeweilige Herkunft oder die jeweils intendierte Verwendung angeben.

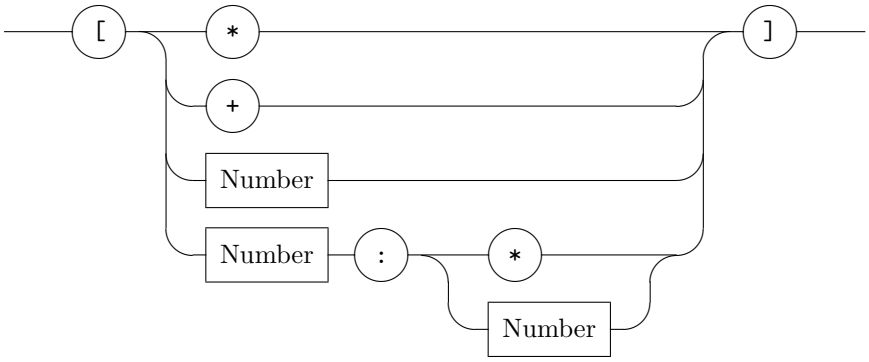
Text
: Ein Bezeichner oder ein String durch Apostrophe eingeschlossen.

IdentDecl



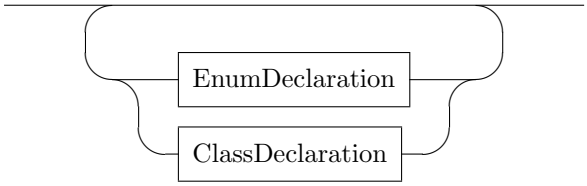
[1] engl.: *rail diagrams*
[2] engl.: *extended Backus–Naur form*

Range

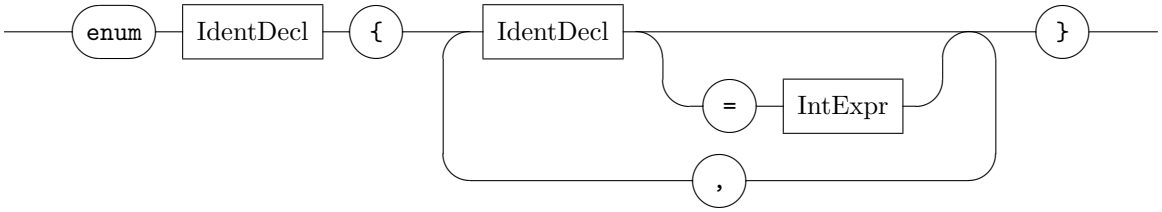


A.2 Metamodell

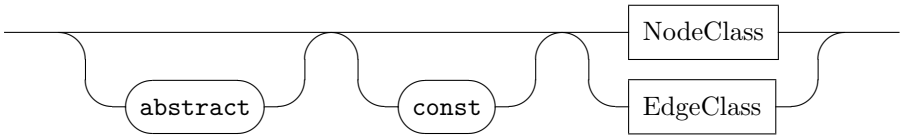
GraphModel



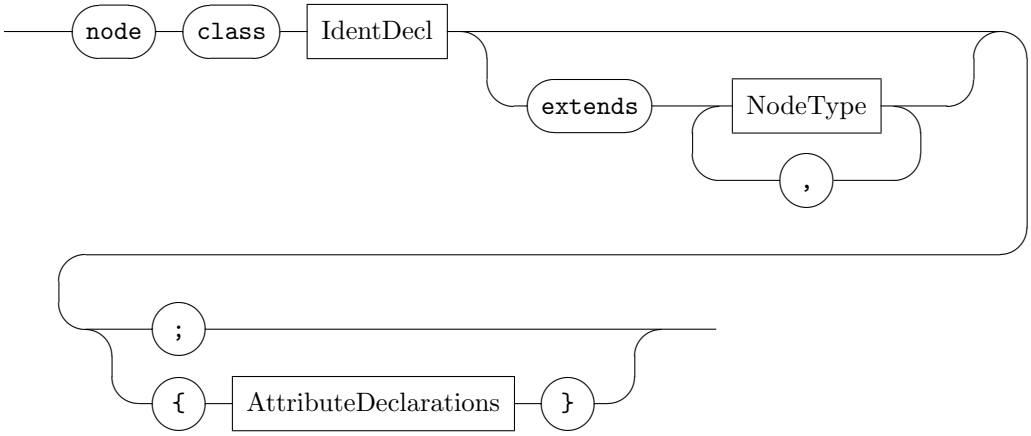
EnumDeclaration



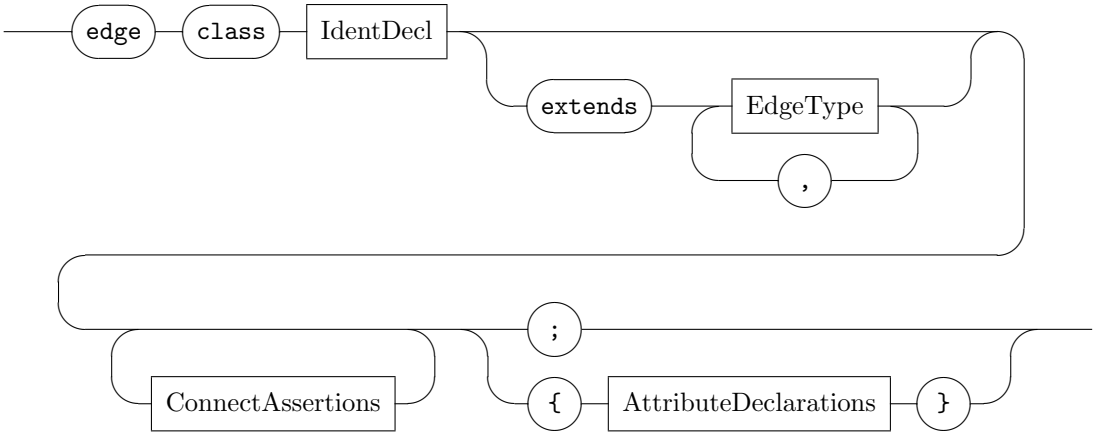
ClassDeclaration



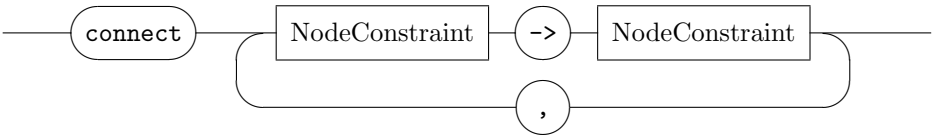
NodeClass



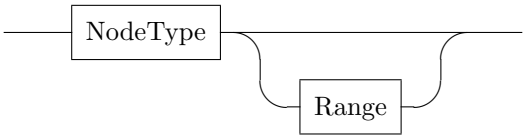
EdgeClass



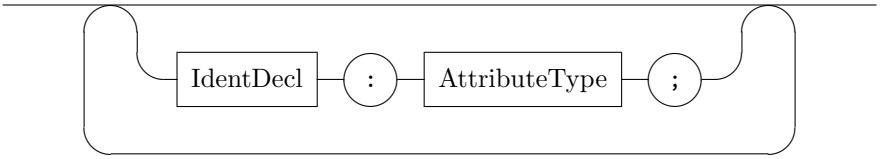
ConnectAssertions



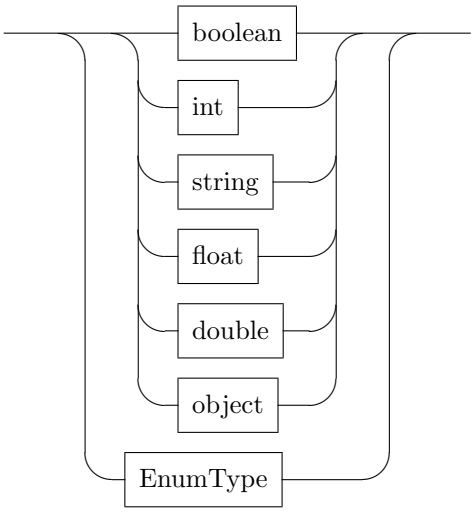
NodeConstraint



AttributeDeclarations

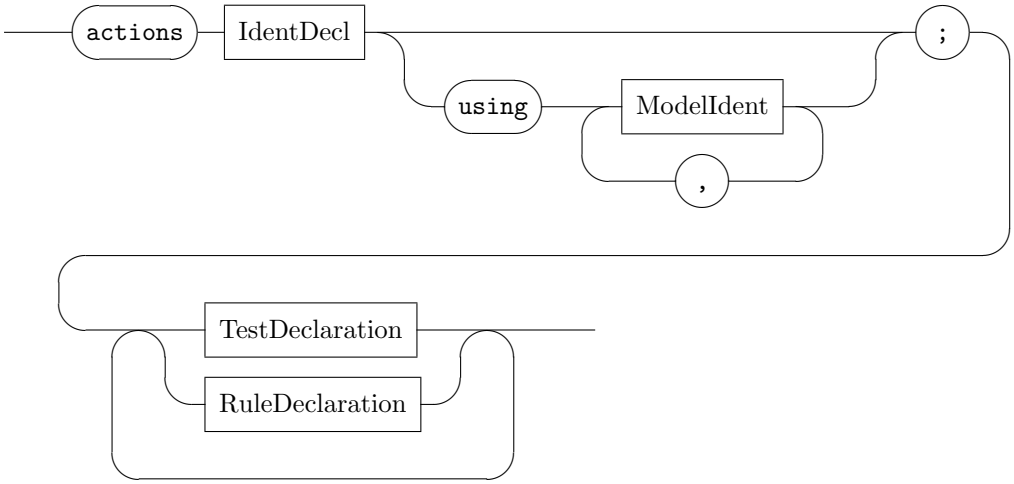


AttributeType

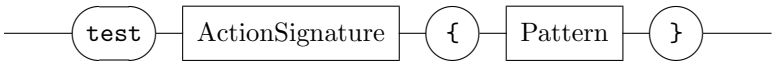


A.3 Graphersetzungsregeln

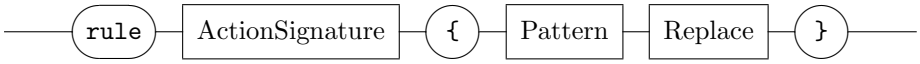
ActionSet



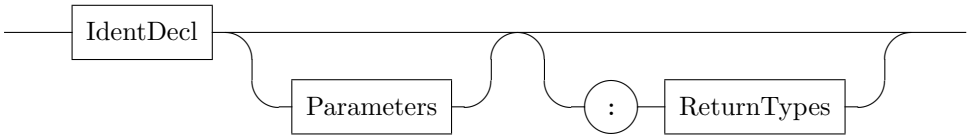
TestDeclaration



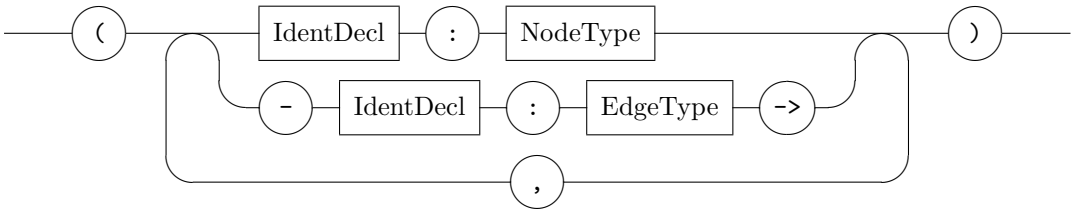
RuleDeclaration



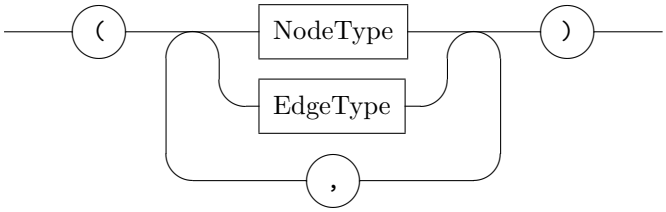
ActionSignature



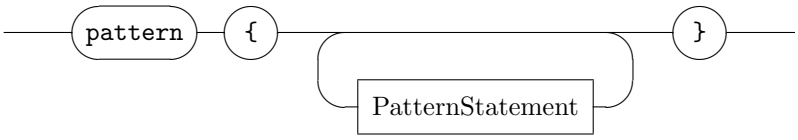
Parameters



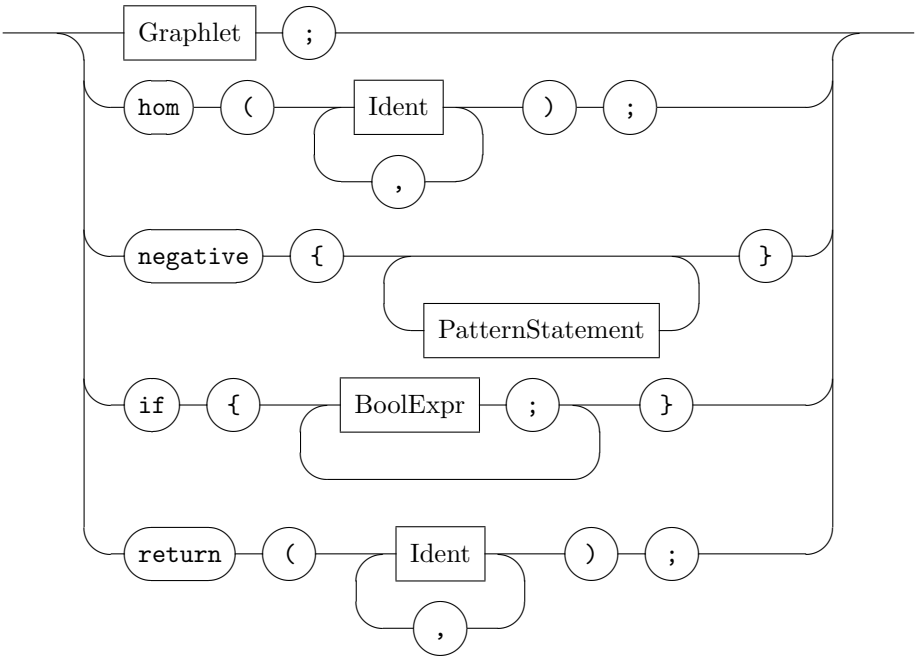
ReturnTypes



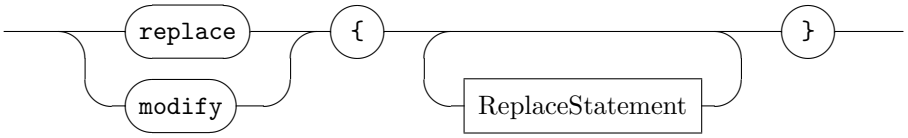
Pattern



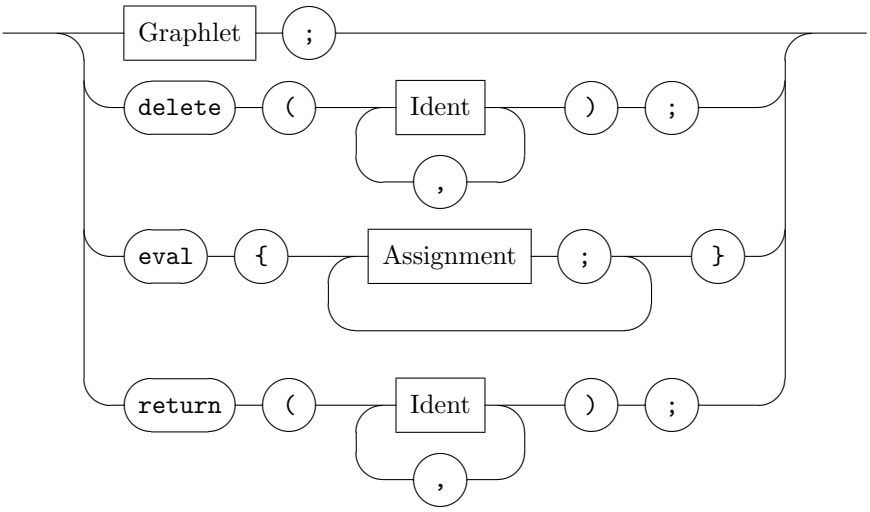
PatternStatement



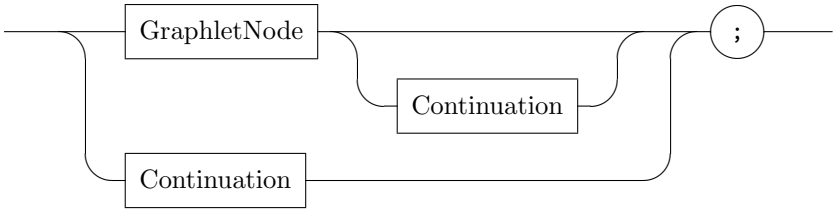
Replace



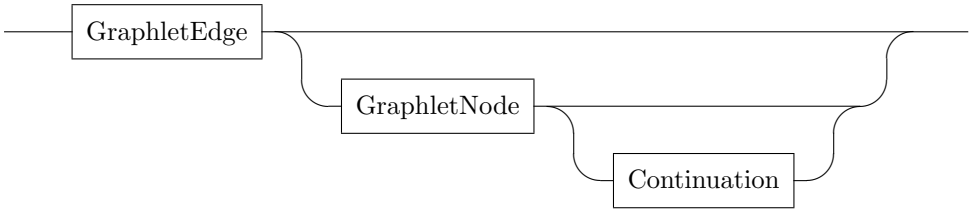
ReplaceStatement



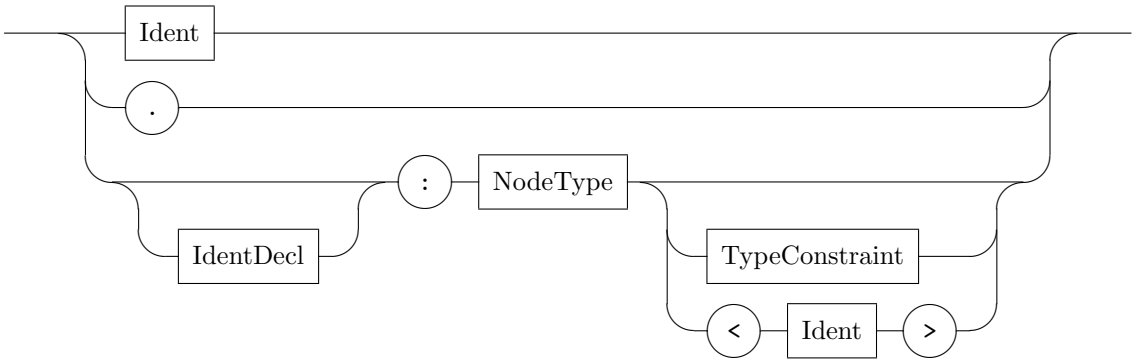
Graphlet



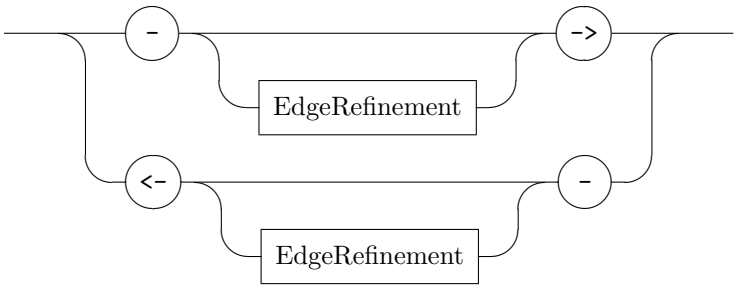
Continuation



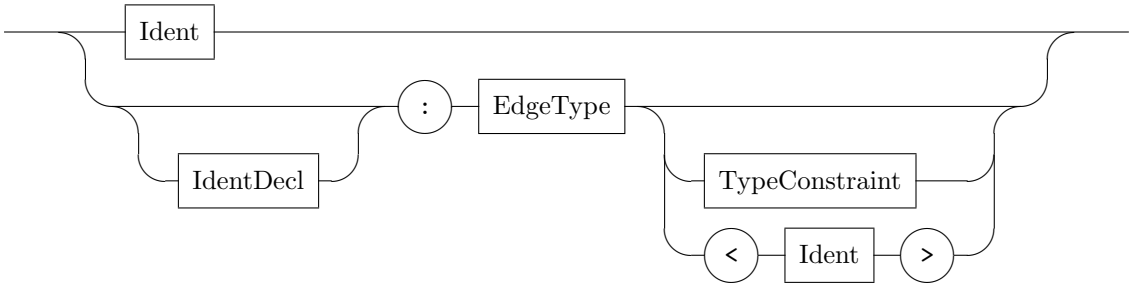
GraphletNode



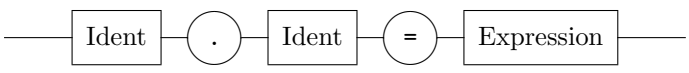
GraphletEdge



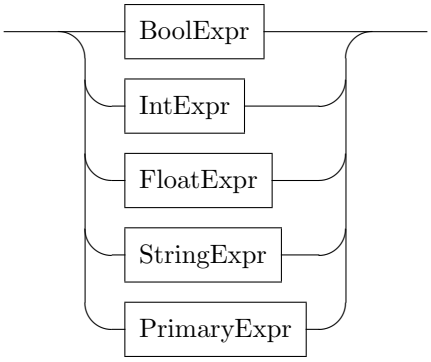
EdgeRefinement



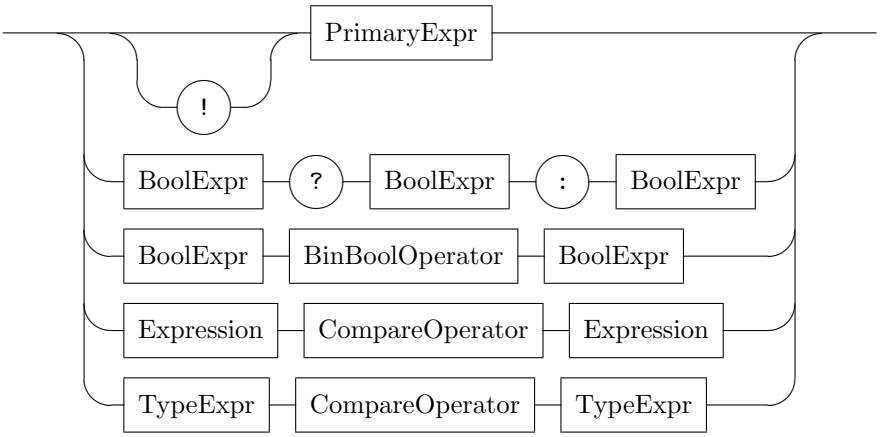
Assignment



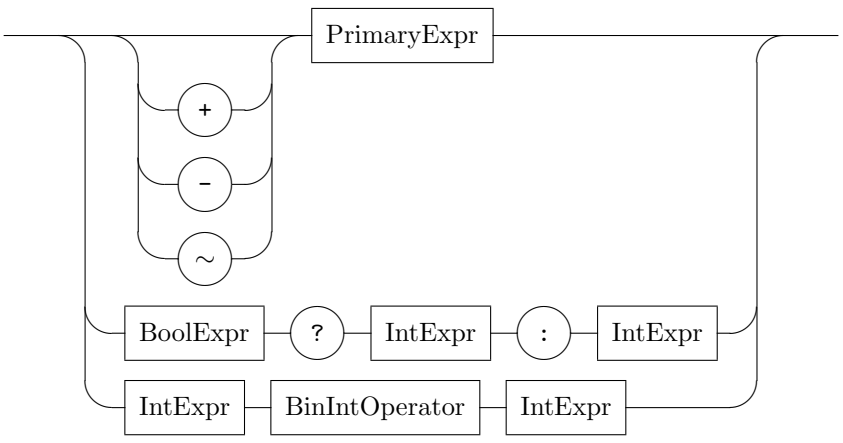
Expression



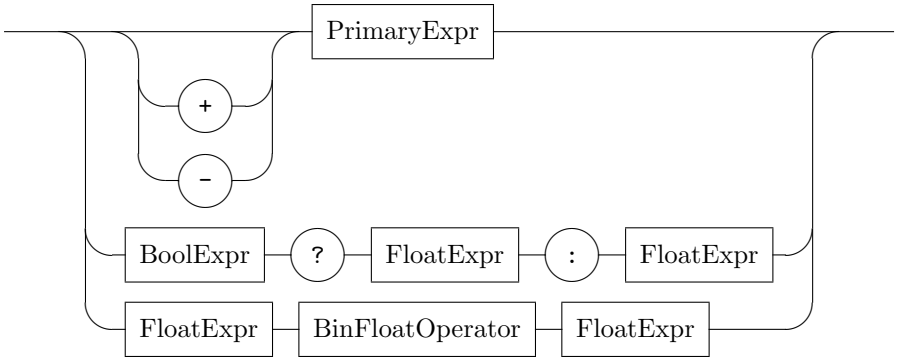
BoolExpr



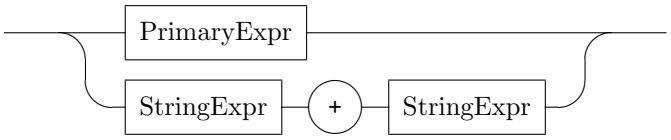
IntExpr



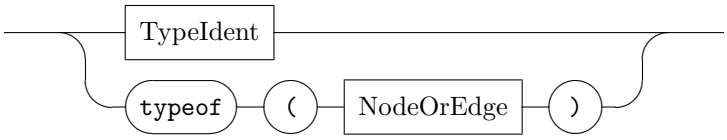
FloatExpr



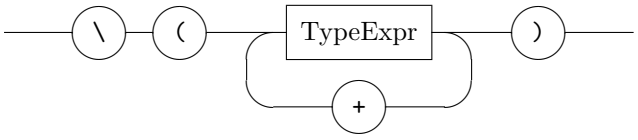
StringExpr



TypeExpr



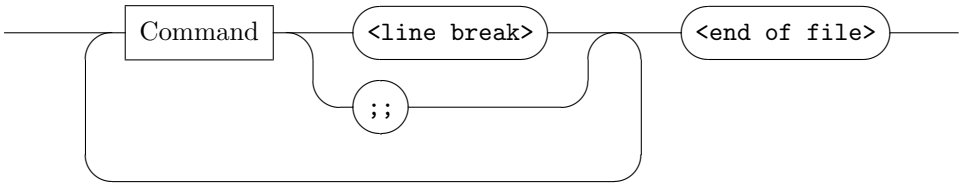
TypeConstraint



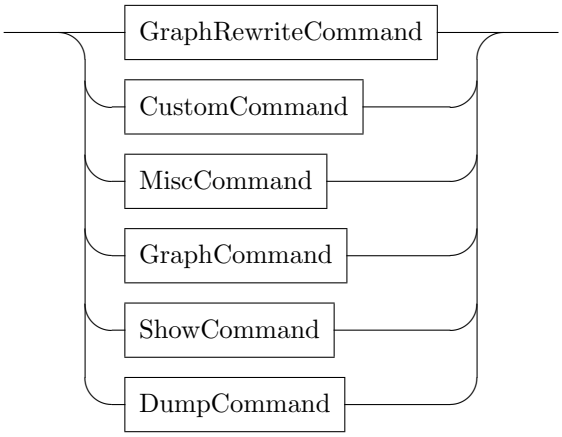
A.4 Regelauswahl

Zusätzlich zur eigentlichen Regelauswahl (mittels *GraphRewriteSequence*) geben wir hier die gesamte Syntax der GrShell an, in die die Sprache zur Regelauswahl eingebettet ist. Natürlich kann die Regelauswahl auch direkt über die libGr geschehen (siehe Abschnitt A.5 für ein Beispiel hierzu).

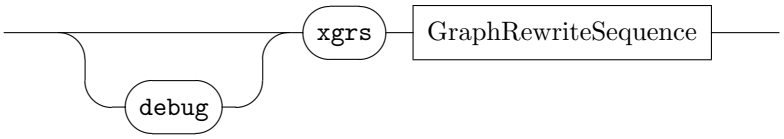
Script



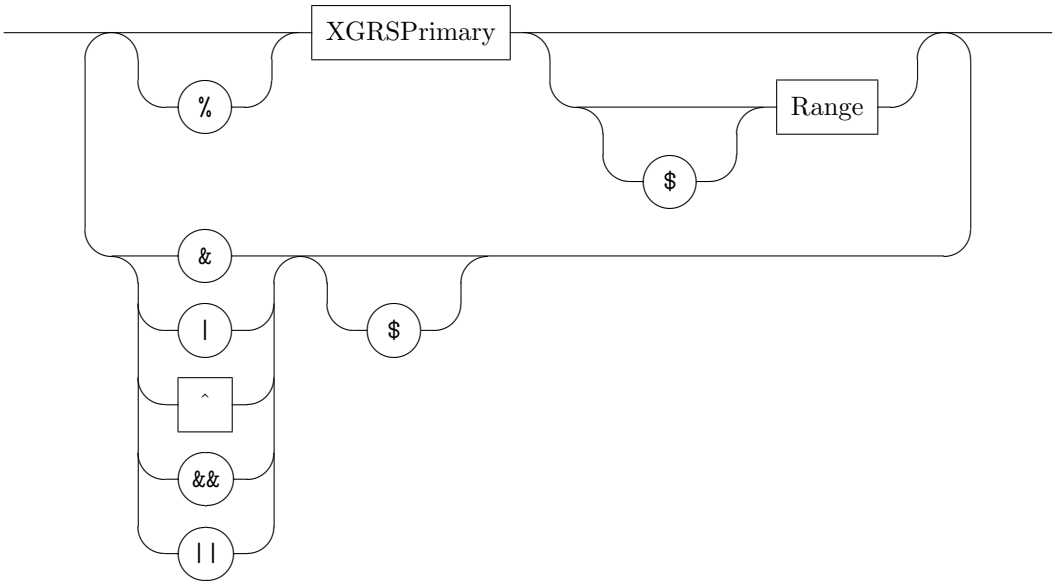
Command



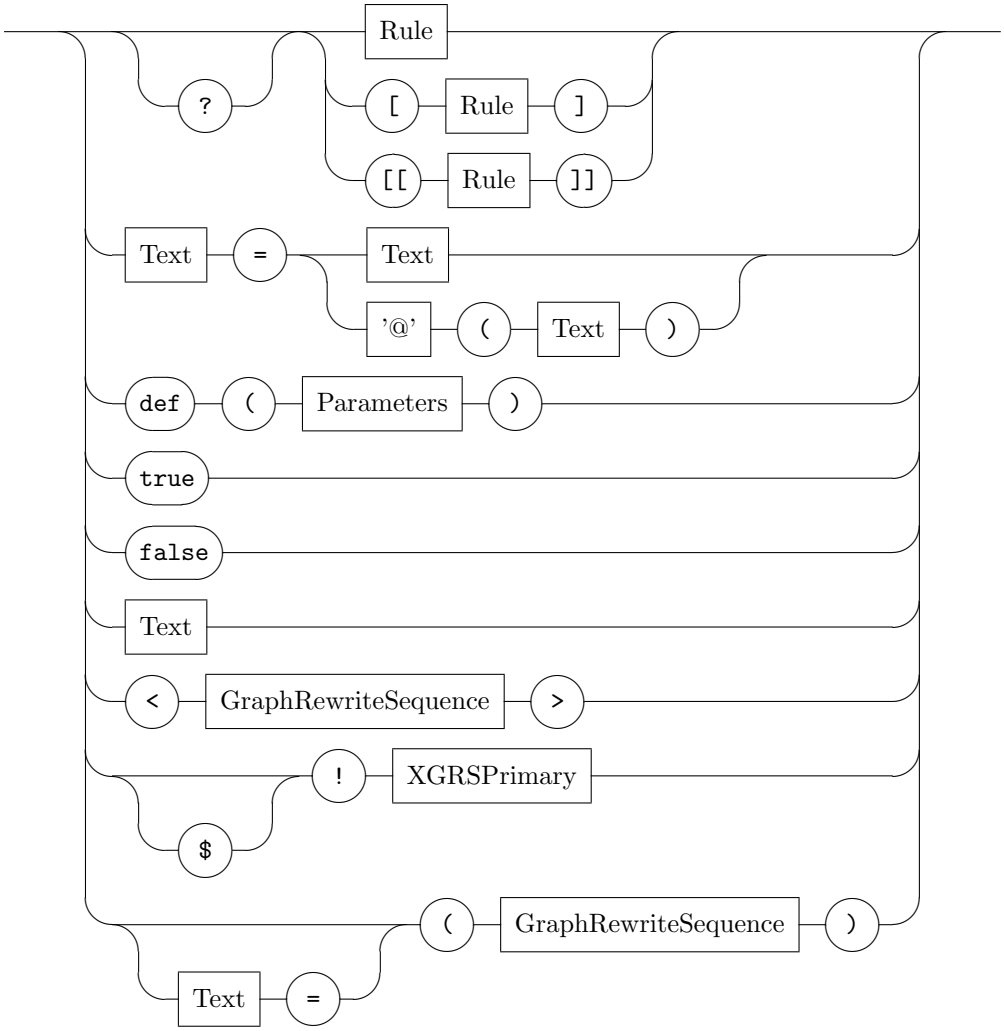
GraphRewriteCommand



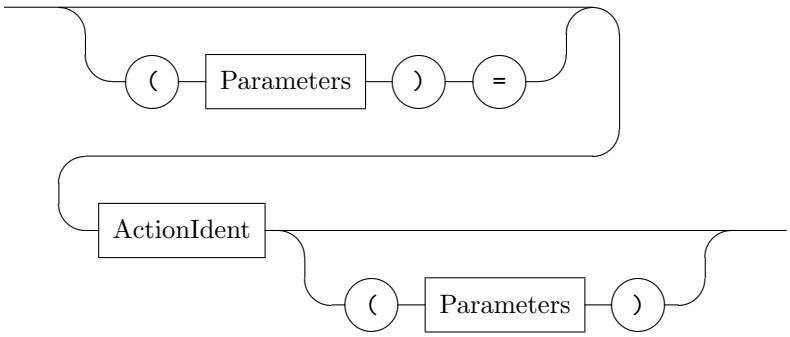
GraphRewriteSequence



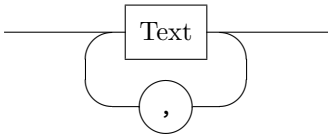
XGRSPrimary



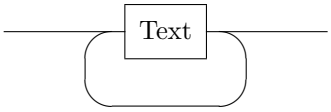
Rule



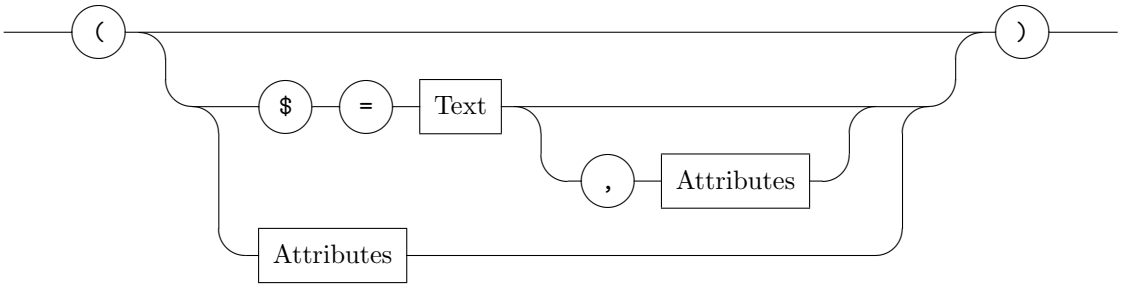
Parameters



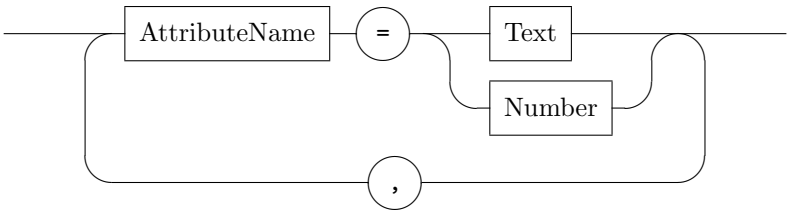
SpacedParameters



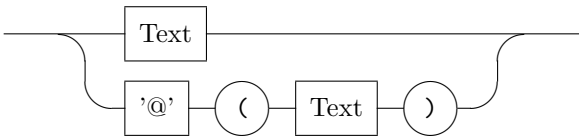
Constructor



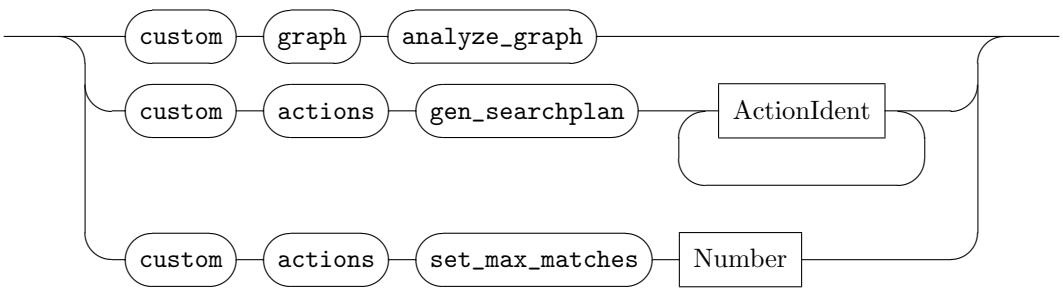
Attributes



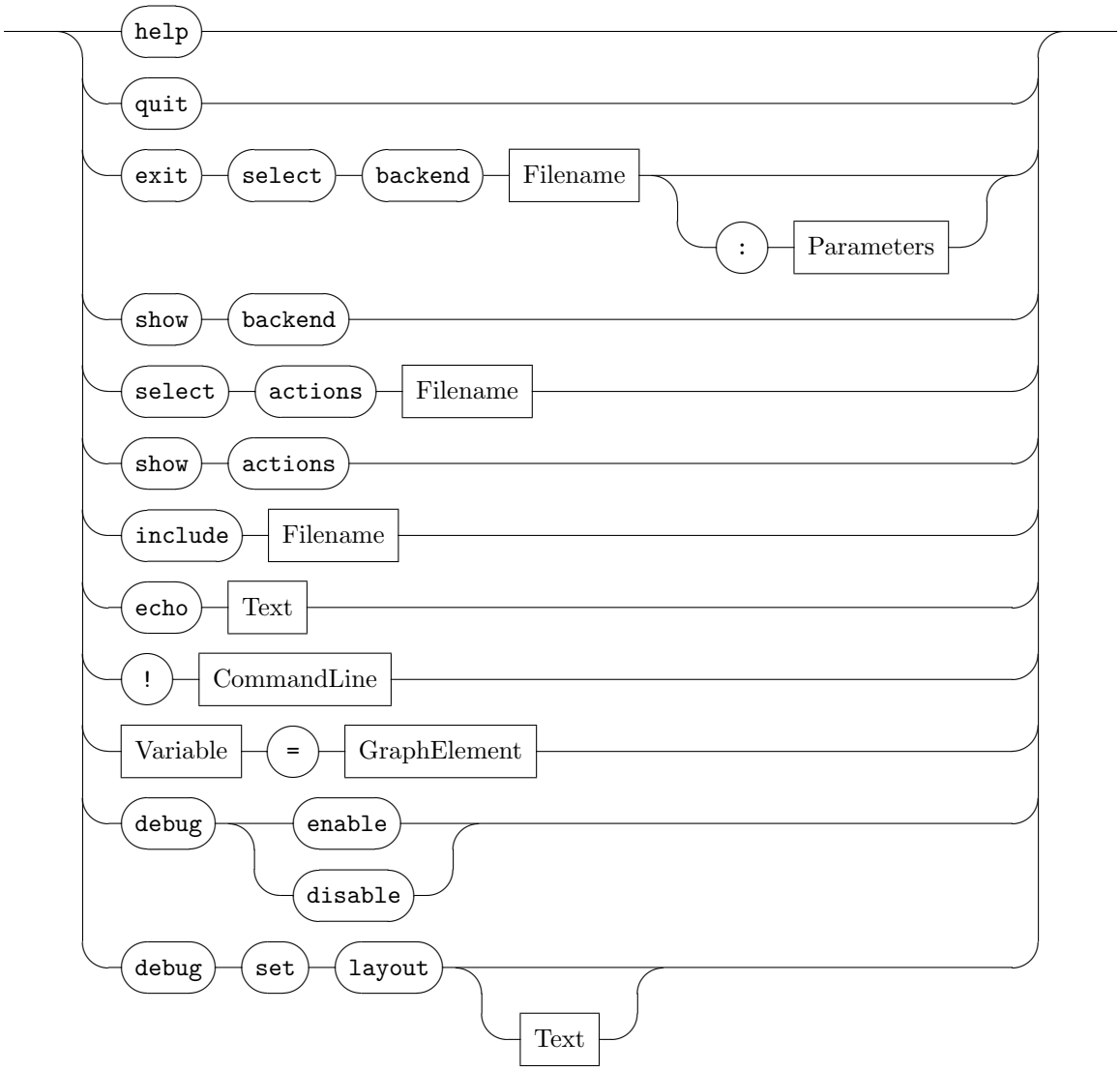
GraphElement



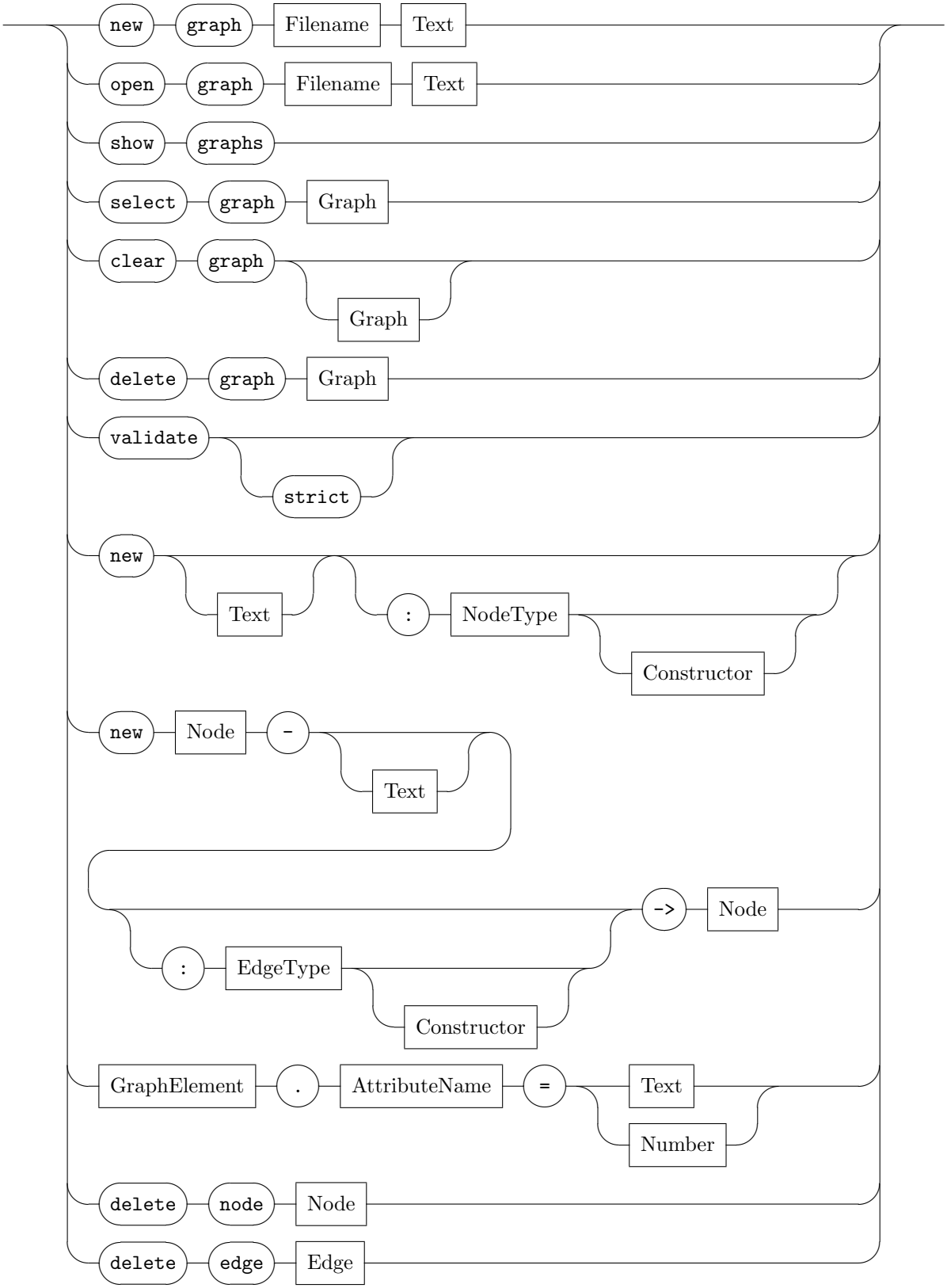
CustomCommand



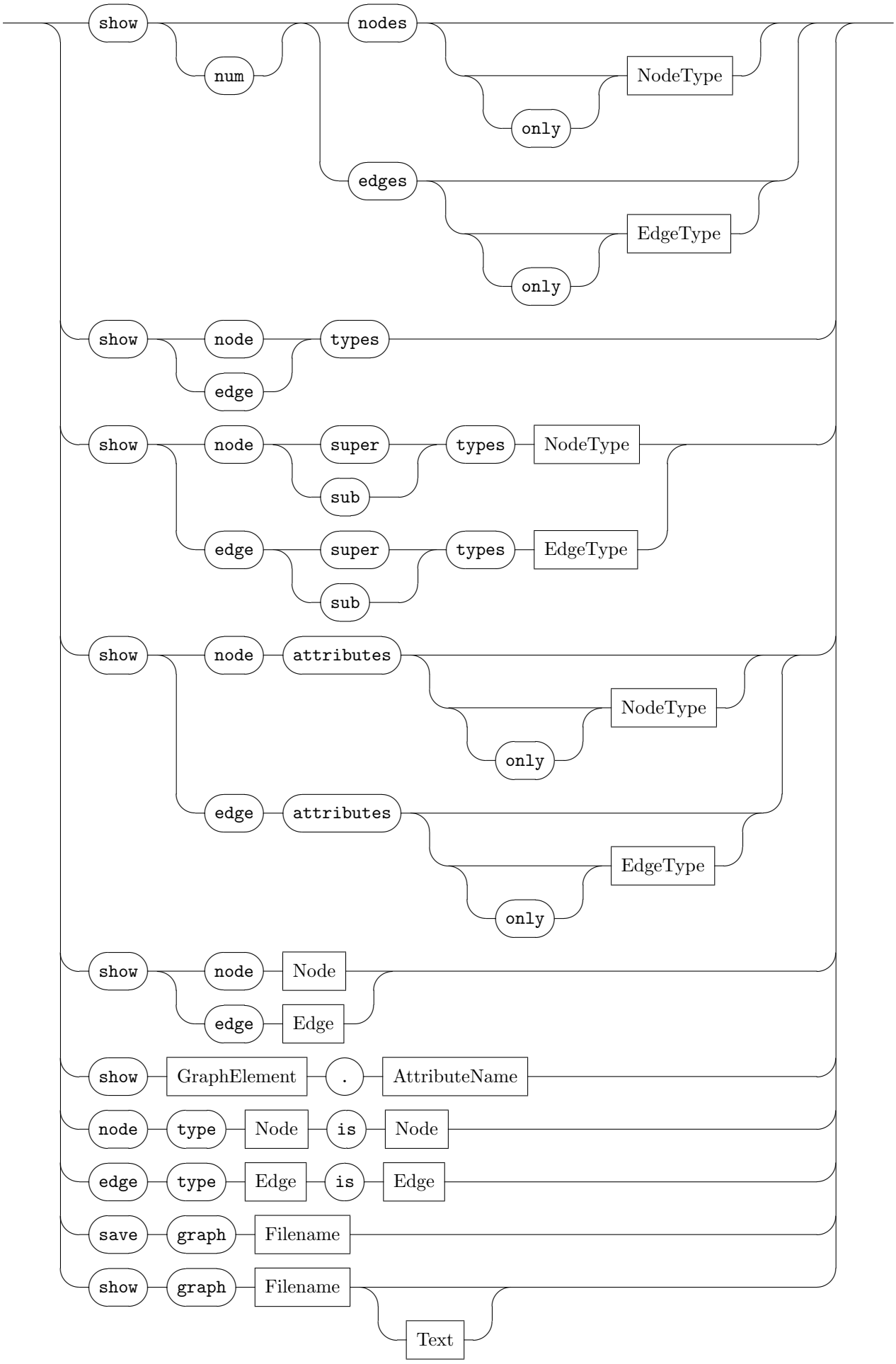
MiscCommand



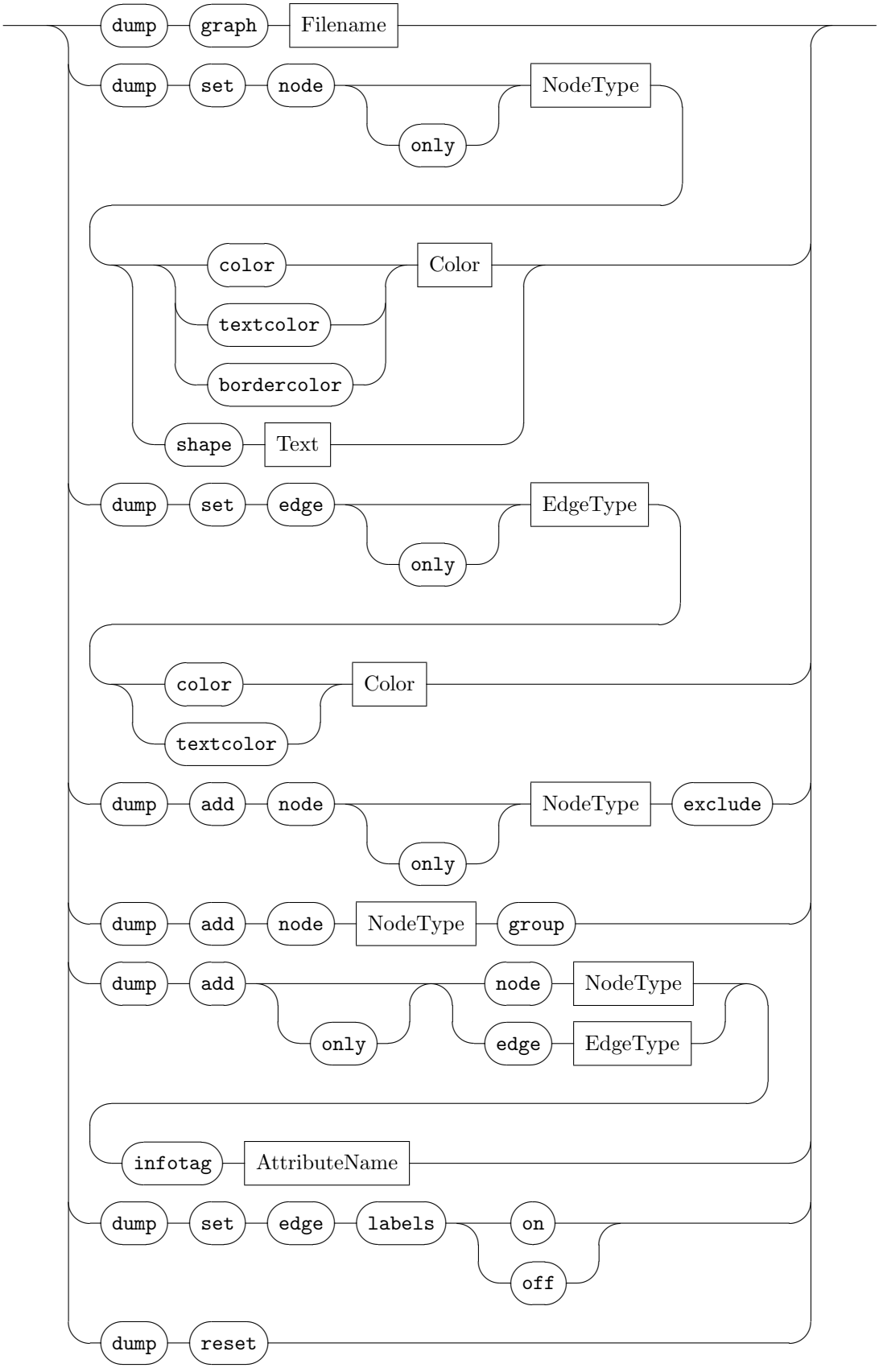
GraphCommand



ShowCommand



DumpCommand



A.5 Passungsauswahl

Die Passungsauswahl kann nur über die libGr stattfinden. Die Regelauswahl lässt durch den direkten Zugriff natürlich auch noch viel elaborierter steuern, als mit den XGRS (siehe Abschnitt 4.4). Die Bibliothek libGr stellt es dem Benutzer anheim, Mustersuche und Ersetzungsschritt voneinander zu trennen:

```
IMatches Match(IGraph graph, int maxMatches, IGraphElement[] parameters);
IGraphElement[] Modify(IGraph graph, IMatch match);
```

Im Programmfragment A.1 wird zuerst mit `myAction.Match` jede Passung bestimmt und hernach einzeln mittels `myMatches.GetMatch(i)` untersucht. Die erste Passung, die als brauchbar befunden wird, darf für den Ersetzungsschritt verwendet werden `myAction.Modify(myGraph, myMatches.GetMatch(i));`.

Programm A.1: Mustersuche und Ersetzungsschritt in libGr

```
1 IMatches myMatches = myAction.Match(myGraph, -1, null); // -1: get all the matches
2 for (int i = 0;  i < myMatches.NumMatches; i++)
3 {
4    if (inspectCarefully(myMatches.GetMatch(i))
5    {
6       myAction.Modify(myGraph, myMatches.GetMatch(i));
7       break;
8    }
9 }
```

ANHANG B

BEISPIEL UND SPEZIFIKATIONEN

B.1 Beispiele und Randfälle für Anwendbarkeit

Wir betrachten im Folgenden einige Grenzfälle von GR-Mustern. Die Frage ist jeweils, ob das GR-Muster Passungen hat und was ist ggf. das Ergebnis ist. Wir wenden die Regeln jeweils auf den leeren Graphen H_0 und den Graphen H_1 aus Abbildung B.1 an. Diese Fragen sind allesamt mithilfe von Definition 4.20 auf Seite 68 beantwortbar.

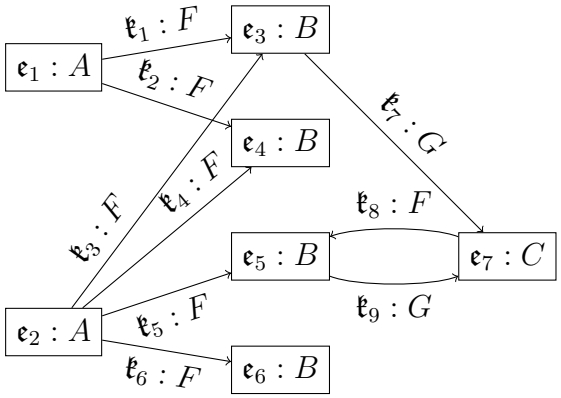


Abbildung B.1: Arbeitsgraph H_1

- Das Muster der Regel $r1$ hat genau eine Passung auf H_0 und H_1. Die Graphen sind unverändert.

```
rule r1 {
  pattern {}
  replace {}
}
```

- Das Muster der Regel $r2$ hat genau eine Passung auf H_0 und H_1. Den Graphen wird je eine neue Ecke vom Wurzeltyp hinzugefügt.

```
rule r2 {
  pattern {
    if { true; }
  }
  replace { x:Node; }
```

```
}
```

- Regel $r3$ ist nie anwendbar.

```
rule r3 {
  pattern {
    if { false; }
  }
  replace {}
}
```

- Regel $r4$ ist nie anwendbar, da das negative Muster immer passt.

```
rule r4 {
  pattern {
    negative {
    }
  }
  replace {}
}
```

- Regel $r5$ ist nie anwendbar, da das negative Muster immer passt, weil $e2$ und ex gleich abgebildet werden dürfen und so falls das positiv Muster gefunden wurde auch immer das negative gefunden wird.

```
rule r5 {
  pattern {
    e1:B --> e2:C --> e1;
    negative {
      hom(e2, ex);
      ex:B --> e2;
    }
  }
  replace {}
}
```

- Regel $r6$ ist bei H_0 nicht anwendbar. Bei H_1 wird eine Passung gefunden ($e1 \mapsto \mathfrak{e}_5, e2 \mapsto \mathfrak{e}_7$). Der Ergebnisgraph geht aus H_1 durch entfrenen von $\mathfrak{k}_5, \mathfrak{k}_7, \mathfrak{k}_8, \mathfrak{k}_9, \mathfrak{e}_5, \mathfrak{e}_7$ hervor.

```
rule r6 {
  pattern {
    e1:B --> e2:C --> e1;
    negative {
      ex:B <-- e2;
    }
  }
  replace {}
}
```

B.2 Simulation einer Turingmaschine

Ein Graphmodell für Turingmaschinen bei denen die Übergangsfunktion als Graphersetzungsregel formuliert ist und die Zustände über Attribute codiert sind:

```
node class Zustand {
  q : int;
}

node class Band {
  bu : string;
}

edge class kopf;

edge class rechts;
```

Im Folgenden sind generisch alle Graphersetzungsregeln angegeben, die für die jeweiligen Übergänge der Turingmaschine nötig sind. Zusätzlich sind zwei Regeln angegeben, die stückweise das beidseitig unendliche Band herstellen.

$\delta : (q, e) \mapsto (q', a, R)$

```
pattern {
  m:Zustand -:kopf-> b1:Band -:rechts-> b2:Band;
  if {
    m.q == q;
    b1.bu == e;
  }
}
replace {
  b1 -:rechts-> b2 <-:kopf- m;
  eval {
    m.q = q';
    b1.bu = a;
  }
}
```

$\delta : (q, e) \mapsto (q', a, L)$

```
pattern {
  b0:Band -:rechts-> b1:Band <-:kopf- m:Zustand;
  if {
    m.q == q;
    b1.bu == e;
  }
}
replace {
  m -:kopf-> b0 -:rechts-> b1;
  eval {
```

```
11      m.q = q';
12      b1.bu = a;
13    }
14  }
```

$\delta : (q, e) \mapsto (q', a, N)$

```
1  pattern {
2    m:Zustand -:kopf-> b1:Band;
3    if {
4      m.q == q;
5      b1.bu == e;
6    }
7  }
8  replace {
9    m -:kopf-> b1;
10   eval {
11     m.q = q';
12     b1.bu = a;
13   }
14 }
```

Blanksysmbol-Regel nach Links

```
1  pattern {
2    m:Zustand;
3    b:Band;
4    if {
5      m.q != f;
6    }
7    negative {
8      bn:Band -:rechts-> b;
9    }
10 }
11 replace {
12   m;
13   bn:Band -:rechts-> b;
14   eval {
15     bn.bu = "#";
16   }
17 }
```

Blanksysmbol-Regel nach Rechts

```
1  pattern {
2    m:Zustand;
3    b:Band;
4    if {
```

```
    m.q != f;
  }
  negative {
    b -:rechts-> bn:Band;
  }
}
replace {
  m;
  b -:rechts-> bn:Band;
  eval {
    bn.bu = "#";
  }
}
```

B.2.1 Fleißiger Biber

Ein fleißiger Biber (drei Zustände, zwei Bandsymbolen, vgl. [MB00a]) als Turingmaschine gemäß vorangehendem Modell (die Regeln, die stückweise das beidseitig unendliche Band herstellen, sind so abgeändert, dass nur dann das Band erweitert wird, wenn der Kopf auch tatsächlich am Rand steht):

```
using TuringZustandModel;

rule init {
   pattern {
   }
   replace {
      m : Zustand -:kopf-> b : Band;
      eval {
         m.q = 1;
         b.bu = "#";
      }
   }
}

rule XL {
   pattern {
      m : Zustand -:kopf-> b : Band;
      negative {
         b <-:rechts- bx : Band;
      }
   }
   replace {
      m -:kopf-> b <-:rechts- bn: Band;
      eval {
         bn.bu = "#";
      }
   }
}

rule XR {
```

```
31    pattern {
32       m : Zustand -:kopf-> b : Band;
33       negative {
34          b -:rechts-> bx : Band;
35       }
36    }
37    replace {
38       m -:kopf-> b -:rechts-> bn: Band;
39       eval {
40          bn.bu = "#";
41       }
42    }
43 }
44
45 rule R1 {
46    pattern {
47       m : Zustand -:kopf-> b:Band;
48       bl:Band -r:rechts-> b;
49       if { m.q == 1 &&  b.bu == "1"; }
50    }
51    replace {
52       m -:kopf-> bl;
53       bl -r-> b;
54       eval {
55          m.q = 3;
56          b.bu = "1";
57       }
58    }
59 }
60
61 rule R2 {
62    pattern {
63       m : Zustand -:kopf-> b:Band;
64       b -r:rechts-> br:Band;
65       if { m.q == 1 &&  b.bu == "#"; }
66    }
67    replace {
68       m -:kopf-> br;
69       b -r-> br;
70       eval {
71          m.q = 2;
72          b.bu = "1";
73       }
74    }
75 }
76
77 rule R3 {
78    pattern {
79       m : Zustand -:kopf-> b:Band;
80       b -r:rechts-> br:Band;
81       if { m.q == 2 &&  b.bu == "1"; }
```

```
82    }
83    replace {
84       m -:kopf-> br;
85       b -r-> br;
86       eval {
87          m.q = 2;
88          b.bu = "1";
89       }
90    }
91 }
92
93 rule R4 {
94    pattern {
95       m : Zustand -:kopf-> b:Band;
96       bl:Band -:rechts-> b;
97       if { m.q == 2 &&  b.bu == "#"; }
98    }
99    replace {
100      m -:kopf-> bl;
101      bl -:rechts-> b;
102      eval {
103         m.q = 1;
104         b.bu = "1";
105      }
106   }
107 }
108
109 rule R5 {
110   pattern {
111      m : Zustand -:kopf-> b:Band;
112      b -:rechts-> br:Band;
113      if { m.q == 3 &&  b.bu == "1"; }
114   }
115   replace {
116      m -:kopf-> br;
117      b -:rechts-> br;
118      eval {
119         m.q = 42;
120         b.bu = "1";
121      }
122   }
123 }
124
125 rule R6 {
126   pattern {
127      m : Zustand -:kopf-> b:Band;
128      bl:Band -:rechts-> b;
129      if { m.q == 3 &&  b.bu == "#"; }
130   }
131   replace {
132      m -:kopf-> bl;
```

```
133       bl -:rechts-> b;
134       eval {
135          m.q = 2;
136          b.bu = "1";
137       }
138    }
139 }
```

B.3 Graphmodell von FIRM

```
1  /*
2   * Project:     GRS
3   * File name:   Firm.gm
4   * Purpose:     A specification of Firm for use with GrGen
5   * Author:      Rubino Geiss
6   * Mowified by:
7   * Created:     10.9.2003
8   * Copyright:   (c) 2007 Universitaet Karlsruhe
9   * Licence:     GPL
10  */
11
12 model Firm;
13
14 node class FIRM_node extends Node {
15    generation: int;
16 }
17
18 edge class FIRM_edge extends Edge {
19    generation: int;
20 }
21
22 /*********************************************
23  * Modes                                     *
24  *********************************************/
25
26 enum ENUM_sort {
27    auxiliary, control_flow, memory, internal_boolean,
28    int_number, float_number, reference, character
29 }
30 enum ENUM_arithmetic_kind {
31    uninitialized, none, twos_complement, ones_complement,
32    int_BCD, ieee754, float_BCD, max, unknown
33 }
34 enum ENUM_modecode {
35    irm_BB, irm_X,   irm_F,   irm_D,  irm_E,  irm_Bs,
36    irm_Bu, irm_Hs,  irm_Hu,  irm_Is, irm_Iu, irm_Ls,
37    irm_Lu, irm_C,   irm_P,   irm_b,  irm_M,  irm_T,
38    irm_U,  irm_ANY, irm_BAD, irm_max
39 }
40
```

```
41 node class Mode extends FIRM_node {
42    name            : string;
43    size     : int;
44    sort     : ENUM_sort;
45    code     : ENUM_modecode;
46    sign     : boolean;
47    arithmetic  : ENUM_arithmetic_kind;
48    shift    : int;
49 }
50
51
52 /*********************************************
53  * Types                                     *
54  *********************************************/
55
56 enum ENUM_state       { layout_undefined, layout_fixed }
57
58 /************* Type Nodes         *************/
59
60 node class Type extends FIRM_node {
61    id    : int;
62    name  : string;
63    state : ENUM_state;
64    size  : int;
65    align : int;
66 }
67
68 node class Compound extends Type;
69 node class Class extends Compound;
70 node class Struct extends Compound;
71 node class Union extends Compound;
72 node class Method extends Type {
73    n_params : int;      // number of calling paramters
74    n_ress   : int;      // number of results
75    variadic : boolean;  // true: additional variadic parameters allowed
76 }
77 node class Array extends Type {
78    n_dimensions : int;
79 }
80 node class Enum extends Type;
81 node class Pointer extends Type;
82 node class Primitive extends Type;
83
84 /************* Type Edges         *************/
85
86 // Class (Sub) -> Class (Super)
87 edge class is_subtype_of extends FIRM_edge
88    connect Class [*] -> Class [*];  // Subclass -> Superclass
89 edge class member extends FIRM_edge
90                // Entities may or may not be an Compound member: [0:1]
91    connect  Compound [*] -> Entity [0:1];
```

```
92 edge class parameter extends FIRM_edge
93    connect  Method [*] -> Type [*] {
94    position : int;
95 }
96 edge class result extends FIRM_edge
97    connect Method [*] -> Type [*] {
98    position : int;
99 }
100 edge class element_type extends FIRM_edge
101    connect  Array [1] -> Type [*];
102 edge class element_ent extends FIRM_edge
103    connect  Array [1] -> Entity [*];
104
105
106 edge class lower extends FIRM_edge
107    connect  Array [*] -> IR_node [*] {
108    position : int;
109 }
110 edge class upper extends FIRM_edge
111    connect  Array [*] -> IR_node [*] {
112    position : int;
113 }
114
115 edge class named_value extends FIRM_edge
116    connect Enum [*] -> Tarval [*] {
117    name : string;
118 }
119 edge class has_type extends FIRM_edge
120    connect  Call     [1]   -> Type [*],
121       SymConst [0:1] -> Type [*],
122       Pointer  [1]   -> Type [*];
123
124
125 /*****************************************
126  * Tarval                                *
127  *****************************************/
128
129 node class Tarval extends FIRM_node {
130    value : string;      //  is this aprobate
131 }
132
133 edge class has_mode extends FIRM_edge
134    connect  Tarval    [1] -> Mode [*],
135       Pointer   [1] -> Mode [*],
136       Primitive [1] -> Mode [*],
137       Method    [1] -> Mode [*],
138       IR_node   [1] -> Mode [*],
139       Struct    [0:1]->Mode [*],
140                 Enum      [1] -> Mode [*];
141 edge class has_entity extends FIRM_edge
142    connect  SymConst[0:1] -> Entity [*];
```

```
143
144
145 /*********************************************
146  * Entities                                  *
147  *********************************************/
148
149 enum ENUM_allocation   { automatic, parameter, dynamic, static }
150 enum ENUM_visibility   { local, global, extern }
151 enum ENUM_variability  { uninitialized, initialized, partly_constant, constant }
152 enum ENUM_peculiarity  { description, inherited, existent }
153
154 /************* Entity Nodes       *************/
155
156 node class Entity extends FIRM_node {
157    name        : string;      // the (source) name of the entity
158    ld_name     : string;      // the linker name of the entity
159    offset      : int;
160    allocation  : ENUM_allocation;
161    visibility  : ENUM_visibility;
162    variability : ENUM_variability;
163    peculiarity : ENUM_peculiarity;
164    volatility  : boolean;
165 }
166
167 /************* Entity Edges       *************/
168
169 edge class overwrites extends FIRM_edge
170    connect Entity -> Entity;
171
172 edge class init_node extends FIRM_edge;
173
174 edge class init_entity extends FIRM_edge;
175
176 edge class graph extends FIRM_edge
177    connect Entity [0:1] -> Method_IRG [*];
178
179 edge class type extends FIRM_edge
180    connect Entity [1] -> Type [*];
181
182
183 /*********************************************
184  * Method IRG                                *
185  *********************************************/
186
187 /*
188  * Firm IRG the IR graph of a method.
189  * Pointing to Start and End nodes as well as its Entity
190  */
191 node class Method_IRG extends FIRM_node {
192    main_method : boolean;  // set, if this method is the main entry point
193 }
```

```
194
195 edge class meth_start extends FIRM_edge
196    connect Method_IRG -> Start;
197
198 edge class meth_end extends FIRM_edge
199    connect Method_IRG -> End;
200
201 edge class frame_type extends FIRM_edge
202    connect Method_IRG -> Type;
203
204 edge class belong_to extends FIRM_edge
205    connect Block -> Method_IRG;
206
207 node class IR_node extends FIRM_node {
208    index : int;//quickfix for using uprojs
209 }
210
211
212 node class Ordinary;
213 node class Special;
214 node class Arithmetic extends Ordinary;
215 node class Controlflow;
216 node class Memory;
217
218 node class Unary;
219 node class Binary;
220 node class Trinary;
221 node class Nary;
222
223 node class Commutative;
224 node class Associative;
225
226
227 /*****************************************
228  * IR Nodes                               *
229  *****************************************/
230
231 node class Block extends IR_node, Special;
232 node class Start extends IR_node, Special;
233 node class End extends IR_node, Special;
234
235 node class Jmp extends IR_node, Controlflow;
236 node class Cond extends IR_node, Controlflow, Ordinary;
237 node class Return extends IR_node, Controlflow;
238 node class Raise extends IR_node, Controlflow;
239
240 node class Const extends IR_node, Ordinary {
241    value : string;         // tarval coded as string
242 }
243 node class IntConst extends Const {
244   // ATTENTION:
```

```
245    // value inherited from Const is set to "<INTCONST>" and may not be used
246    intval : int;        // tarval coded as string
247 }
248
249 node class SymConst extends IR_node, Ordinary {
250    kind    : int;
251    ptrinfo : string;
252 }
253
254 node class Sel extends IR_node;
255 node class InstOf extends IR_node;
256 node class Call extends IR_node;
257 node class Add extends IR_node, Arithmetic, Binary, Commutative;
258 node class Sub extends IR_node, Arithmetic, Binary;
259 node class Minus extends IR_node, Arithmetic, Unary;
260 node class Mul extends IR_node, Arithmetic, Binary, Commutative;
261 node class Mulh extends IR_node, Arithmetic, Binary, Commutative;
262 node class Quot extends IR_node, Arithmetic, Binary;
263 node class DivMod extends IR_node, Arithmetic, Binary;
264 node class Div extends IR_node, Arithmetic, Binary;
265 node class Mod extends IR_node, Arithmetic, Binary;
266 node class Abs extends IR_node, Arithmetic, Unary;
267 node class And extends IR_node, Arithmetic, Binary, Commutative;
268 node class Or extends IR_node, Arithmetic, Binary, Commutative;
269 node class Eor extends IR_node, Arithmetic, Binary, Commutative;
270 node class Not extends IR_node, Arithmetic, Unary;
271 node class Cmp extends IR_node, Ordinary;
272 node class Shl extends IR_node, Arithmetic, Binary;
273 node class Shr extends IR_node, Arithmetic, Binary;
274 node class Shrs extends IR_node, Arithmetic, Binary;
275 node class Rot extends IR_node, Arithmetic, Binary;
276 node class Conv extends IR_node, Ordinary;
277 node class Cast extends IR_node;
278 node class Phi extends IR_node, Ordinary;
279 node class Mux extends IR_node, Trinary;
280
281 node class MemNode extends IR_node, Memory {
282    volatility  : boolean;
283 }
284
285 node class Load extends MemNode;
286 node class Store extends MemNode;
287
288 enum ENUM_alloc_where { stack_alloc, heap_alloc }
289
290 node class Alloc extends IR_node, Memory {
291    where : ENUM_alloc_where;
292 }
293 node class Free extends IR_node, Memory;
294 node class Sync extends IR_node, Memory {
295    arity : int;
```

```
296 }
297
298 node class Proj extends IR_node, Ordinary {
299    proj : int;
300 }
301
302
303 node class Tuple extends IR_node;
304 node class Id extends IR_node;
305 node class Bad extends IR_node;
306 node class NoMem extends IR_node;
307 node class Confirm extends IR_node;
308 node class Unknown extends IR_node;
309 node class Filter extends IR_node;
310 node class Break extends IR_node, Controlflow;
311 node class CallBegin extends IR_node;
312 node class EndReg extends IR_node;
313 node class EndExcept extends IR_node, Controlflow;
314
315 //select parts of a register (e.g. 32 bit of a 128 bit register)
316 node class VProj extends IR_node, Ordinary{
317    proj : int;
318 }
319
320 /************* IR Edges         *************/
321
322 edge class flow extends FIRM_edge
323    connect IR_node [*] -> IR_node [*] {
324    position : int;
325 }
326
327 edge class df extends flow;      // IR_node -> IR_node, data flow
328 edge class mem extends df;    // IR_node -> IR_node, memory
329 edge class tuple extends df;     // Edges of Mode Tuple
330 edge class cf extends flow    // control flow
331    connect IR_node [1] -> Block [*],
332                // We cannot distinguish ProjI etc from ProjXi: therefor 0
333                // ProjX form Start has 2 successors
334            Block   [*] -> Proj  [0:2];
```

B.4 Varró Leistungstest

Das Graphmodell:

```
1 model MutexModel;
2
3 node class Process;
4 node class Resource;
5
6 edge class next
7     connect Process [0:1] -> Process [0:1];
```

```
8
9  edge class held_by
10    connect Resource [0:1] -> Process [*];
11
12 edge class token
13    connect Resource [0:1] -> Process [*];
14
15 edge class release
16    connect Resource [0:1] -> Process [*];
17
18 edge class request
19    connect Process [*] -> Resource [*];
```

Die Regeln:

```
1  actions Mutex using MutexModel;
2
3  rule newRule {
4     pattern {
5        p1:Process -n:next-> p2:Process;
6     }
7     replace {
8        p1 -n1:next-> p:Process -n2:next-> p2;
9     }
10 }
11
12 rule mountRule {
13    pattern {
14       p:Process;
15    }
16    replace {
17       p <-t:token- r:Resource;
18    }
19 }
20
21 rule requestRule {
22    pattern {
23       p:Process;
24       r:Resource;
25       negative {
26          r -hb:held_by-> p;
27       }
28       negative {
29          p -req:request-> m:Resource;
30       }
31    }
32    replace {
33       p -req:request-> r;
34    }
35 }
36
37 rule takeRule {
```

```
38    pattern {
39       r:Resource;
40       r -t:token-> p:Process -req:request-> r;
41    }
42    replace {
43       r -hb:held_by-> p;
44    }
45 }
46
47 rule releaseRule {
48    pattern {
49       r:Resource;
50       r -hb:held_by-> p:Process;
51       negative {
52          p -req:request-> m:Resource;
53       }
54    }
55    replace {
56       r -rel:release-> p;
57    }
58 }
59
60 rule giveRule {
61    pattern {
62       r:Resource;
63       r -rel:release-> p1:Process -n:next-> p2:Process;
64    }
65    replace {
66       p1 -n-> p2 <-t:token- r;
67    }
68 }
```

B.5 Der Bechnmark Rechner

Für unsere Leistungstests verwendeten wir einen Einprozessorrechner ohne Hyper-Threading. Um die aktuellen Messungen mit älteren vergleichen zu können, haben wir immer dieselbe, schon etwas in die Jahre gekommene, Hardware benutzt. Dieser Abschnitt listet alle relevanten Soft- und Hardwarekomponenten dieses Rechners auf.

CPU

- AMD Athlon™ XP 3000+
- L1-Cache: 64 + 64 KiB (Daten + Befehle)
- L2-Cache: 512 KiB mit Kernfrequenz
- 2100 MHz
- FSB 400

Hauptspeicher

- 2 × Infineon 512 MiB
- DDR 500 CL3

Hauptplatine

- Asus A7N8X-X

Grafikkarte

- ATI/AMD Radeon 7000 OEM

Software/Linux

- Suse Linux 9.3
- Linux Kernel 2.6.11.4-21.17-default
- GNU C Library 2.3.4 (20050218)
- X Window System Version 6.8.2 (9. Feb. 2005)
- Mono 1.2.3.1
- Java 2 Runtime Environment, SE (build 1.5.0_04-b05)
- xterm 200-3
- GNOME 2.10.0
- KDE 3.4.0

Software/Windows

- Microsoft Windows XP, Version 5.1 (Build 2600.xpsp_sp2_gdr.070227-2254: Service Pack 2)
- Microsoft.NET Framework v2.0.50727
- Java™ SE Runtime Environment (build 1.6.0_01-b06)

LITERATURVERZEICHNIS

[AEH+99] Andries, Marc ; Engels, Gregor ; Habel, Annegret ; Hoffmann, Berthold ; Kreowski, Hans-Jörg ; Kuske, Sabine ; Plump, Detlef ; Schürr, Andy ; Taentzer, Gabriele: Graph Transformation for Specification and Programming. In: *Science of Computer Programming* 34 (1999), Nr. 1, S. 1–54

[AK02] Allen, Randy ; Kennedy, Ken: *Optimizing compilers for modern architectures: a dependence-based approach*. Los Altos, CA, USA : Morgan Kaufmann Publishers, 2002. – 790 S. – ISBN 1–55860–286–0

[Ame78] American National Standards Institute (Hrsg.): *ANSI Fortran X3.9–1978*. pub-ANSI:adr: American National Standards Institute, 1978. `http://www.fh-jena.de/~kleine/history/languages/ansi-x3dot9-1978-Fortran77.pdf`. – Approved April 3, 1978 (also known as Fortran 77).

[Ame95] American National Standards Institute: *American National Standard for information technology: programming language ADA: ANSI/ISO/IEC 8652-1995: Revision and redesignation of ANSI/MIL 1815A-1983*. Revised. 1430 Broadway, New York, NY 10018, USA : American National Standards Institute, 1995 (FIPS PUB 119-1)

[And96] Andries, Marc: *Graph Rewrite Systems and Visual Database Languages*, Rijksuniversiteit Leiden, Diss., 1996. `http://citeseerx.ist.psu.edu/showciting?cid=933567`

[Ass00] Assmann, Uwe: Graph rewrite systems for program optimization. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22 (2000), Nr. 4, S. 583–637. `http://dx.doi.org/10.1145/363911.363914`. – DOI 10.1145/363911.363914. – ISSN 0164–0925

[Bas00] Bashford, Steven: *Constraintbasierte Codegenerierung für eingebettete Prozessoren*, Universität Dortmund, Fachbereich Informatik, Diss., 2000

[Bat05a] Batz, Gernot V.: *Generierung von Graphersetzungen mit programmierbarem Suchalgorithmus*, Universität Karlsruhe, IPD Goos, Studienarbeit, Mai 2005. `http://www.info.uni-karlsruhe.de/papers/sa_batz.pdf`

[Bat05b] Batz, Gernot V.: *Graphersetzung für eine Zwischendarstellung im Übersetzerbau*, Universität Karlsruhe, IPD Goos, Diplomarbeit, Ok-

tober 2005. http://www.info.uni-karlsruhe.de/papers/da_batz.pdf

[Bat06] Batz, Gernot V.: An Optimization Technique for Subgraph Matching Strategies / Universität Karlsruhe, Fakultät für Informatik. Version: September 2006. http://www.info.uni-karlsruhe.de/papers/TR_2006_7.pdf. 2006 (2006-7). – Forschungsbericht. – ISSN 1432–7864

[BBB+57] Backus, J. W. ; Beeber, R. J. ; Best, S. ; Goldberg, R. ; Haibt, L. M. ; Herrick, H. L. ; Nelson, R. A. ; Sayre, D. ; Sheridan, P. B. ; Stern, H. ; Ziller, I. ; Hughes, R. A. ; Nutt, R.: The FORTRAN automatic coding system. In: *Western Joint Computer Conf. Los Angeles.*, 1957

[BFG95] Blostein, Dorothea ; Fahmy, Hoda ; Grbavec, Ann: Practical use of graph rewriting / Queen's University, Kingston, Ontario, Kanada. Version: Januar 1995. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.8841. 1995 (95-373). – Forschungsbericht

[BG07] Blomer, Jakob ; Geiss, Rubino: The GrGen.NET User Manual / Universität Karlsruhe, Fakultät für Informatik. Version: Juli 2007. http://www.info.uni-karlsruhe.de/papers/TR_2007_5.pdf. 2007 (2007-5). – Forschungsbericht. – ISSN 1432–7864

[BGG05] Bernabé, Gregorio ; García, José M. ; González, José: Reducing 3D Fast Wavelet Transform Execution Time Using Blocking and the Streaming SIMD Extensions. In: *J. VLSI Signal Process. Syst.* 41 (2005), Nr. 2, S. 209–223. http://dx.doi.org/10.1007/s11265-005-6651-6. – DOI 10.1007/s11265–005–6651–6. – ISSN 0922–5773

[BGT90] Bunke, Horst ; Glauser, Thomas ; Tran, T.-H.: An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm. In: Ehrig, Hartmut (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph-Grammars and Their Application to Computer Science* Bd. 532, Springer-Verlag, 1990 (LNCS). – ISBN 3–540–54478–X, S. 174–189

[BKG08] Batz, Gernot V. ; Kroll, Moritz ; Geiss, Rubino: A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In: Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)* Bd. 5088, Springer-Verlag, 2008 (LNCS), 233–248

[BL99] Bashford, Steven ; Leupers, Rainer: Constraint driven code selection for fixed-point DSPs. In: *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. New York, NY, USA : ACM Press, 1999. – ISBN 1–58133–109–7, S. 817–822

[BMS92] Baetke, Frank ; Metzger, Bob ; Smith, Presley: The CONVEX Application Compiler - A Major Step into the Direction of Automatic Parallelization. . In: Meuer, Hans W. (Hrsg.): *Supercomputer*, Springer-Verlag, 1992 (Informatik Aktuell). – ISBN 3–540–55709–1, 158-172

[BNS+05] Balogh, András ; Németh, Attila ; Schmidt, András ; Rath, István ; Vágó, Dávid ; Varró, Dániel ; Pataricza, András: The VIATRA2 Model Transformation Framework, 2005. – Presented at ECMDA 2005 – Tools Track

[Boe05] Boesler, Boris: *Codeerzeugung mit Graphersetzung und Lösungsgraphen*. Aachen, Germany, Universität Karlsruhe, Fakultät für Informatik, Diss., Februar 2005

[BSWG00] Budiu, Mihai ; Sakr, Majd ; Walker, Kevin ; Goldstein, Seth C.: BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In: *Proceedings of the 2000 Europar Conference* Bd. 1900. Munich, Germany : Springer-Verlag, August 2000 (LNCS). – ISSN 0302–9743, 969–979

[Buc08] Buchwald, Sebastian: *Erweiterung von* GrGen.NET *um DPO-Semantik und ungerichtete Kanten*, Universität Karlsruhe, IPD Goos, Studienarbeit, Mai 2008. `http://www.info.uni-karlsruhe.de/papers/sa_buchwald.pdf`

[BV06] Balogh, András ; Varró, Dániel: Advanced model transformation language constructs in the VIATRA2 framework. In: Haddad, Hisham (Hrsg.): *SAC*, ACM Press, 2006. – ISBN 1–59593–108–2, 1280-1287

[CC95] Click, Cliff ; Cooper, Keith D.: Combining analyses, combining optimizations. In: *ACM Trans. Program. Lang. Syst.* 17 (1995), Nr. 2, S. 181–196. `http://dx.doi.org/10.1145/201059.201061`. – DOI 10.1145/201059.201061. – ISSN 0164–0925

[CFR+91] Cytron, Ron ; Ferrante, Jeanne ; Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Efficiently computing static single assignment form and the control dependence graph. In: *ACM Trans. Program. Lang. Syst.* 13 (1991), Nr. 4, S. 451–490. `http://dx.doi.org/10.1145/115372.115320`. – DOI 10.1145/115372.115320. – ISSN 0164–0925

[CFSV04a] Conte, Donatello ; Foggia, Pasquale ; Sansone, Carlo ; Vento, Mario: Thirty years of Graph Matching in Pattern Recognition. In: *International Journal of Pattern Recognition and Artificial Intelligence* 18 (2004), Mai, Nr. 3, S. 265–298. `http://dx.doi.org/10.1142/S0218001404003228`. – DOI 10.1142/S0218001404003228

[CFSV04b] Cordella, Luigi P. ; Foggia, Pasquale ; Sansone, Carlo ; Vento, Mario: A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26 (2004), Nr. 10, S. 1367–1372. `http://dx.doi.org/10.1109/TPAMI.2004.75`. – DOI 10.1109/TPAMI.2004.75. – ISSN 0162–8828

[CL65] Chu, Y.J. ; Liu, T.H.: On the shortest arborescence of a directed graph. In: *Science Sinica* 14 (1965), S. 1396–1400

[Cod70] Codd, E. F.: A relational model of data for large shared data banks. In: *Commun. ACM* 13 (1970), Nr. 6, S. 377–387. `http://dx.doi.org/10.1145/362384.362685`. – DOI 10.1145/362384.362685. – ISSN 0001–0782

[CP95] Click, Cliff ; Paleczny, Michael: A simple graph-based intermediate representation. In: *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*. New York, NY, USA : ACM Press, 1995. – ISBN 0–89791–754–5, S. 35–49

[CT04] Cooper, Keith D. ; Torczon, Linda: *Engineering a Compiler*. Morgran Kaufmann Publishers, 2004

[Dec03] Dechter, R.: *Constraint Processing*. Morgan Kaufmann Publishers, 2003

[Den07] Denninger, Oliver: *Erweiterung des Kantenkonzepts deklarativer Graphersetzungssysteme von Einfachkanten über Hyperkanten zu „Superkanten“*, Universität Karlsruhe, IPD Tichy, Diplomarbeit, März 2007. `http://www.ipd.uka.de/Tichy/uploads/arbeiten/149/diplomarbeit.pdf`

[DGG08] Denninger, Oliver ; Gelhausen, Tom ; Geiss, Rubino: Supergraphs - Simplifying Model Transformation for Advanced UML Structures. In: Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)* Bd. 5088, Springer-Verlag, 2008 (LNCS)

[Dör95] Dörr, Heiko: *LNCS*. Bd. 922: *Efficient Graph Rewriting and its Implementation*. Springer-Verlag, 1995. – ISBN 0387600558

[Edm67] Edmonds, Jack: Optimum Branchings. In: *Journal of Research of the National Bureau of Standards* 71B (1967), S. 233–240

[EEPT06] Ehrig, H. ; Ehrig, K. ; Prange, U. ; Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006 (Monographs in Theoretical Computer Science.)

[EHK+99] Ehrig, H. ; Heckel, R. ; Korff, M. ; Löwe, M. ; Ribeiro, L. ; Wagner, A. ; Corradini, A.: Algebraic Approaches to Graph Transformation - Part II: Single Pushout A. and Comparison with Double Pushout A. In: *[Roz99]* Bd. 1. 1999, S. 247–312

[EKL91] Ehrig ; Korff ; Löwe: Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In: Ehrig, Hartmut (Hrsg.) ; Kreowski, Hans-Jörg (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph-Grammars and Their Application to Computer Science* Bd. 532, Springer-Verlag, 1991 (LNCS). – ISBN 3–540–54478–X, S. 24–37

[EKS03] Eckstein, Erik ; König, Oliver ; Scholz, Bernhard: Code Instruction Selection Based on SSA-Graphs. In: Krall, Andreas (Hrsg.): *SCOPES* Bd. 2826, Springer-Verlag, 2003 (LNCS). – ISBN 3–540–20145–9, S. 49–65

[EM42] Eilenberg, Samuel ; MacLane, Saunders: General Theory of Natural Equivalences. In: *Annals of Mathematics* 43 (1942), S. 757–831

[EM45] Eilenberg, Samuel ; MacLane, Saunders: General Theory of Natural Equivalences. In: *Transactions of American Mathematical Society* 58 (1945), Nr. 2, S. 231–294

[Epp94] Eppstein, David: Subgraph isomorphism in planar graphs and related problems / Univ. of California, Irvine, Dept. of Information and Computer Science. Version: 1994. `http://www.ics.uci.edu/~eppstein/pubs/Epp-TR-94-25.pdf`. 1994 (94-25). – Forschungsbericht

[Epp95] Eppstein, David: Subgraph isomorphism in planar graphs and related problems. In: *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1995. – ISBN 0–89871–349–8, 632–640

[EPS73] Ehrig, Hartmut ; Pfender, M. ; Schneider, H. J.: Graph Grammars: An Algebraic Approach. In: *Proc. of the 14th Annual Symposium on Switching and Automata Theory*. Iowa City, IA, USA : IEEE Computer Society Press, 1973, S. 167–180

[ERT99] Ermel, C. ; Rudolf, M. ; Taentzer, G.: The AGG Approach: Language and Environment. In: *[Roz99]* Bd. 2. 1999, S. 551–603

[ES03] Eckstein, Erik ; Scholz, Bernhard: Addressing mode selection. In: *CGO '03: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1913–X, S. 337–346

[ESL89] Emmelmann, H. ; Schröer, F.-W. ; Landwehr, L.: BEG: a generation for efficient back ends. In: *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. New York, NY, USA : ACM Press, 1989. – ISBN 0–89791–306–X, S. 227–237

[EWO04] Eichenberger, Alexandre E. ; Wu, Peng ; O'Brien, Kevin: Vectorization for SIMD architectures with alignment constraints. In: *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–807–5, S. 82–93

[Fly72] Flynn, Michael J.: Some Computer Organizations and Their Effectiveness. In: *IEEE Trans. Comput.* C-21 (1972), S. 948–960

[For79] Forgy, Charles L.: *On the efficient implementation of production systems.*, Carnegie-Mellon University, Diss., 1979

[For82] Forgy, Charles L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In: *Artificial Intelligence* 19 (1982), Nr. 1, S. 17–37

[Fuj05] Fujaba Developer Team: *Fujaba-Homepage*. `http://www.fujaba.de/`. Version: September 2005

[GBG+06] Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, Andrea (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Montanari, Ugo (Hrsg.) ; Ribeiro, Leila (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *ICGT 2006* Bd. 4178, Springer-Verlag, 2006 (LNCS). – ISBN 3–540–38870–2, S. 383–397

[GDG08] Gelhausen, Tom ; Derre, Bugra ; Geiss, Rubino: Customizing GrGen.NET for model transformation. In: *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*. New York, NY, USA : ACM Press, 2008. – ISBN 978–1–60558–033–3, 17–24

[Gei08] Geiss, Rubino: GrGen.NET. `http://www.grgen.net`. Version: Mai 2008

[GG78] Glanville, R. S. ; Graham, Susan L.: A new method for compiler code generation. In: *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA : ACM Press, 1978, S. 231–254

[GH03] Geiss, Rubino ; Hack, Sebastian: Übersetzerbau - Ein kleiner Überblick / Universität Karlsruhe, Fakultät für Informatik. Version: Oktober 2003. `http://www.info.uni-karlsruhe.de/papers/TR_2003_18.pdf`. 2003 (2003-18). – Forschungsbericht

[GJ90] Garey, Michael R. ; Johnson, David S.: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1990. – ISBN 0716710455

[GJSB05] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *Java™ Language Specification*. 3. Addison-Wesley Professional, 2005

[GK07] Geiss, Rubino ; Kroll, Moritz: On Improvements of the Varro Benchmark for Graph Transformation Tools / Universität Karlsruhe, Fakultät für Informatik. Version: Dezember 2007. `http://www.info.uni-karlsruhe.de/papers/TR_2007_7.pdf`. 2007 (2007-7). – Forschungsbericht. – ISSN 1432–7864

[GK08] Geiss, Rubino ; Kroll, Moritz: GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In: Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. 3rd Intl.*

Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07) Bd. 5088, Springer-Verlag, 2008 (LNCS), S. 568–569

[GMK07] Geiss, Rubino ; Mallon, Christoph ; Kroll, Moritz: *Sierpinski Triangle for the AGTIVE 2007 Tool Contest*. `http://www.informatik.uni-marburg.de/~swt/agtive-contest/SierpinskiTriangles.pdf`. Version: Juli 2007. – Akzeptierter Vorschlag für den AGTIVE 2007 Graph Transformation Tools Contest

[Gru04] Grund, Daniel: *Negative Anwendungsbedingungen für den Graphersetzer* GrGen *(Studienarbeit)*, Universität Karlsruhe, IPD Goos, Studienarbeit, September 2004. `http://www.info.uni-karlsruhe.de/papers/sa_grund.pdf`

[GT07] Gelhausen, Tom ; Tichy, Walter F.: Thematic Role based Generation of UML Models from Real World Requirements. In: *First IEEE International Conference on Semantic Computing (ICSC 2007)* IEEE, 2007

[GZ06] Goos, Gerhard ; Zimmermann, Wolf: *Vorlesungen über Informatik Band 1: Grundlagen und funktionales Programmieren*. 4. Springer-Verlag, 2006 `http://www.springer.com/computer/programming/book/978-3-540-24405-9`

[Hac03] Hack, Sebastian: *Graphersetzung für Optimierungen in der Codeerzeugung*, Universität Karlsruhe, IPD Goos, Diplomarbeit, Dezember 2003. `http://www.info.uni-karlsruhe.de/papers/da_hack.pdf`

[HGG06] Hack, Sebastian ; Grund, Daniel ; Goos, Gerhard: Register allocation for programs in SSA-form. In: Andreas Zeller, Alan M. (Hrsg.): *Compiler Construction 2006* Bd. 3923, Springer-Verlag, März 2006

[HJBG81] Hennessy, John L. ; Jouppi, Norman ; Baskett, Forest ; Gill, John: MIPS: a VLSI processor architecture. Stanford, CA, USA : Stanford University, 1981. – Forschungsbericht

[Hof04] Hofmann, Enno: *Regelerzeugung zur maschinenabhängigen Codeoptimierung*, Universität Karlsruhe, IPD Goos, Diplomarbeit, November 2004. `http://www.info.uni-karlsruhe.de/papers/da_hofmann.pdf`

[Homhr] Homer: *Odyssee*. spätes 8. Jahrhundert v. Chr.

[Hop52] Hopper, Grace M.: The education of a computer. In: *ACM '52: Proceedings of the 1952 ACM national meeting (Pittsburgh)*. New York, NY, USA : ACM Press, 1952, S. 243–249

[HP06] Hennessy, John L. ; Patterson, David A.: *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA : Morgan Kaufmann Publishers, 2006. – ISBN 0123704901

[Int91] Intel Corporation (Hrsg.): *i960 MC microprocessor reference manual*. Santa Clara, CA, USA: Intel Corporation, 1991

[Int07a] Intel Corporation (Hrsg.): *Intel 64 and IA-32 Architectures Software Developer's Manual – Instruction Set Reference*. Santa Clara, CA, USA: Intel Corporation, Mai 2007. http://www.intel.com/design/processor/manuals/253667.pdf

[Int07b] Intel Corporation (Hrsg.): *Intel C++ Compiler*. Santa Clara, CA, USA: Intel Corporation, August 2007. http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm

[Jak04] Jakschitsch, Hannes: *Befehlsauswahl auf SSA-Graphen*, Universität Karlsruhe, IPD Goos, Diplomarbeit, November 2004. http://www.info.uni-karlsruhe.de/papers/da_jakschitsch.pdf

[Jak07] Jakumeit, Edgar: *Vorarbeiten für die Erweiterung des Graphersetzungssystems* GrGen *um dynamisch zusammengesetzte Muster*, Universität Karlsruhe, IPD Goos, Studienarbeit, September 2007. http://www.info.uni-karlsruhe.de/papers/sa_jakumeit.pdf

[Jak08] Jakumeit, Edgar: *Mit* GrGen.NET *zu den Sternen – Erweiterung der Regeln eines Graphersetzungswerkzeugs mittels Sterngraphgrammatiken und Paargraphgrammatiken*, Universität Karlsruhe, IPD Goos, Diplomarbeit, Juli 2008. http://www.info.uni-karlsruhe.de/papers/da_jakumeit.pdf

[Joh79] Johnson, Steven C.: Yacc: Yet Another Compiler Compiler. Version: 1979. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.9394. In: *UNIX Programmer's Manual* Bd. 2. New York, NY, USA : Holt, Rinehart, and Winston, 1979, 353–387

[KK04] Kuramochi, Michihiro ; Karypis, George: An Efficient Algorithm for Discovering Frequent Subgraphs. In: *IEEE Transactions on Knowledge and Data Engineering* 16 (2004), Nr. 9, S. 1038–1051. http://dx.doi.org/10.1109/TKDE.2004.33. – DOI 10.1109/TKDE.2004.33. – ISSN 1041–4347

[Kle56] Kleene, Stephen C.: Representation of Events in Nerve Nets and Finite Automata. In: Shannon, C. E. (Hrsg.) ; McCarthy, J. (Hrsg.): *Automata Studies*. Princeton University Press, 1956, S. 3–40

[Kle05] Klein, Gerwin: *JFlex User's Manual*. http://www.jflex.de/manual.html. Version: 2005

[Kro07] Kroll, Moritz: GrGen.NET*: Portierung und Erweiterung des Graphersetzungssystems* GrGen, Universität Karlsruhe, IPD Goos, Studienarbeit, Mai 2007. http://www.info.uni-karlsruhe.de/papers/sa_kroll.pdf

[Kro08] Kroll, Moritz: *Embedding the graph rewrite system* GrGen.NET *into C#*, Universität Karlsruhe, IPD Goos, Diplomarbeit, September 2008. http://www.info.uni-karlsruhe.de/papers/da_kroll.pdf

[Käs97] Kästner, Daniel, Universität des Saarlandes, FB14 Informatik, Diplomarbeit, September 1997

[LB00] Leupers, Rainer ; Bashford, Steven: Graph-based code selection techniques for embedded processors. In: *ACM Trans. Des. Autom. Electron. Syst.* 5 (2000), Nr. 4, S. 794–814. http://dx.doi.org/10.1145/362652.362661. – DOI 10.1145/362652.362661. – ISSN 1084–4309

[Les75] Lesk, M.E.: LEX – A Lexical Aanlyzer Generator / Bell Telefone Laboratories. 1975 (39). – Forschungsbericht

[Lev66] Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. In: *Soviet Physics Dokl.* 6 (1966), S. 126–136

[Lin02] Lindenmaier, Götz: libFIRM – A Library for Compiler Optimization Research Implementing FIRM / Universität Karlsruhe, Fakultät für Informatik. Version: September 2002. http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps. 2002 (2002-5). – Forschungsbericht

[LV02] Larrosa, Javier ; Valiente, Gabriel: Constraint satisfaction algorithms for graph pattern matching. In: *Mathematical. Structures in Comp. Sci.* 12 (2002), Nr. 4, S. 403–422. http://dx.doi.org/10.1017/S0960129501003577. – DOI 10.1017/S0960129501003577. – ISSN 0960–1295

[MB00a] Marxen, H. ; Buntrock, J.: *Old list of record TMs.* http://www.drb.insel.de/~heiner/BB/index.html. Version: August 2000

[MB00b] Messmer, Bruno T. ; Bunke, Horst: Efficient Subgraph Isomorphism Detection: A Decomposition Approach. In: *IEEE Transactions on Knowledge and Data Engineering* 12 (2000), Nr. 2, S. 307–323. http://dx.doi.org/10.1109/69.842269. – DOI 10.1109/69.842269. – ISSN 1041–4347

[McK81] McKay, Brendan D.: Practical graph isomorphism. In: *Congressus Numerantium* 30 (1981), 45–87. http://cs.anu.edu.au/people/bdm/nauty/

[McK90] McKay, Brendan D.: **nauty** User's Guide (Version 1.5) / Australian National University, Department of Computer Science. 1990 (TR-CS-90-02). – Forschungsbericht

[Men99] Menne, Torsten: *Vegleich von CLP und ILP basierten Optimierungsstrategien am Beispiel der Codegenerierung für DSPs*, Universität Dortmund, Fachbereich Informatik, Diplomarbeit, 1999

[MKC00] Manniesing, Rashindra ; Karkowski, Ireneusz ; Corporaal, Henk: Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition. In: *Euro-Par 2000 - Parallel Processing: 6th*, Springer-Verlag, 2000

[MMJW91] Mickel, Andrew B. ; Miner, James F. ; Jensen, Kathleen ; Wirth, Niklaus: *Pascal user manual and report (4th ed.): ISO Pascal standard*. Springer-Verlag, 1991. – ISBN 0–387–97649–3

[Möl03] Möller, Andreas: Eine virtuelle Maschine für Graphprogramme / Universität Oldenburg. 2003 (5/03). – Forschungsbericht

[Mot98] Motorola Corporation (Hrsg.): *AltiVec Technology Programming Environments Manual*. Phoenix, AZ, USA: Motorola Corporation, November 1998. – 350 S.

[Muc97] Muchnick, Steven S.: *Advanced compiler design and implementation*. San Francisco, CA, USA : Morgan Kaufmann Publishers, 1997. – ISBN 1–55860–320–4

[Mül07] Müller, Jens: *Erweiterung des Graphersetzungswerkzeugs* GrGen.NET *um dynamische und kontextsensitive Beschleunigungstechniken*, Universität Karlsruhe, IPD Goos, Studienarbeit, Juni 2007. `http://www.info.uni-karlsruhe.de/papers/sa_mueller.pdf`

[MW00] Metzger, Robert ; Wen, Zhaofang: *Automatic algorithm recognition and replacement: a new approach to program optimization*. Cambridge, MA, USA : MIT Press, 2000. – ISBN 0–262–13368–7

[MWH00] Myers, Richard ; Wilson, Richard C. ; Hancock, Edwin R.: Bayesian Graph Edit Distance. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000), Nr. 6, S. 628–635. `http://dx.doi.org/10.1109/34.862201`. – DOI 10.1109/34.862201. – ISSN 0162–8828

[Nag79] Nagl, Manfred: *Graph - Grammatiken. Theorie, Anwendung, Implementierungen*. Vieweg, 1979

[Nag90] Nagl, Manfred: A Characterization of the IPSEN-Project. In: *Proc. Int. Conf. on System Development Environments & Factories*, Pittman, 1990, S. 141–150

[Nag96] Nagl, Manfred (Hrsg.): *Building Tightly Integrated Software Development Environments: The IPSEN Approach*. Bd. *1170*. Springer-Verlag, 1996 (LNCS). – ISBN 3–540–61985–2

[NJ99] Nguyen, Huy ; John, Lizy K.: Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. In: *ICS '99: Proceedings of the 13th international conference on Supercomputing*. New York, NY, USA : ACM Press, 1999. – ISBN 1–58113–164–X, S. 11–20

[NNH99] Nielson, Flemming ; Nielson, Hanne R. ; Hankin, Chris: *Principles of Program Analysis*. Springer-Verlag, 1999. – ISBN 3540654100

[OK97] Okayama, Sachiko ; Katsuta, Hiroshi: *US Patent #5,684,728: Data processing system having a saturation arithmetic operation function*. 1997

[Pie91] Pierce, B.C.: *Basic Category Theory for Computer Scientists.* MIT Press, 1991

[PLG88] Pelegrí-Llopart, Eduardo ; Graham, Susan L.: Optimal Code Generation for Expression Trees: An Application of BURS Theory. In: *POPL.* San Diego, CA, USA : ACM Press, Januar 1988, S. 294–308

[Plu99] Plump, D.: Term graph rewriting. In: *[Roz99]* Bd. 2, 1999, S. 1–62

[PP94] Pinter, Shlomit S. ; Pinter, Ron Y.: Program optimization and parallelization using idioms. In: *ACM Trans. Program. Lang. Syst.* 16 (1994), Nr. 3, S. 305–327. `http://dx.doi.org/10.1145/177492.177494`. – DOI 10.1145/177492.177494. – ISSN 0164–0925

[PR69] Pfaltz, John L. ; Rosenfeld, Azriel: Web Grammars. In: *Proc. of the 1st International Joint Conference on Artificial Intelligence.* Washington, DC, USA : William Kaufmann, Mai 1969. – ISBN 0–934613–21–4, S. 609–620

[Pro98] Proebsting, T. A.: Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years. (1998). `http://www.research.microsoft.com/~toddpro/papers/law.htm`

[Rob01] Robison, Arch D.: Impact of economics on compiler optimization. In: *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, ACM Press, 2001. – ISBN 1–58113–359–6, S. 1–10

[Roz99] Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific, 1999

[RR05] Rugina, Radu ; Rinard, Martin C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In: *ACM Trans. Program. Lang. Syst.* 27 (2005), Nr. 2, S. 185–235. `http://dx.doi.org/10.1145/1057387.1057388`. – DOI 10.1145/1057387.1057388. – ISSN 0164–0925

[Rud98] Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In: *TAGT'98: Selected papers from the 6th Intl. Workshop on Theory and Application of Graph Transformations* Bd. 1764, Springer-Verlag, 1998 (LNCS), S. 238–251

[Rut52] Rutishauser, Heinz: Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen. In: *Z. Angew. Math. Mech.* 32 (1952), Nr. 3, S. 312–313

[Ryd85] Rydeheard, David E.: Functors and Natural Transformations. In: Pitt, David H. (Hrsg.) ; Abramsky, Samson (Hrsg.) ; Poigné, Axel (Hrsg.) ; Rydeheard, David E. (Hrsg.): *CTCS* Bd. 240, Springer-Verlag, 1985 (LNCS). – ISBN 3–540–17162–2, S. 43–50

[SB01] Strey, Alfred ; Bange, Martin: Performance Analysis of Intel's MMX and SSE: A Case Study. In: *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. London, UK : Springer-Verlag, 2001. – ISBN 3–540–42495–4, S. 142–147

[SBA00] Stephenson, Mark ; Babb, Jonathan ; Amarasinghe, Saman: Bitwidth analysis with application to silicon compilation. In: *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. New York, NY, USA : ACM Press, 2000. – ISBN 1–58113–199–2, S. 108–120

[Sch99] Schürr, A.: The Progres Approach: Language and Environment. In: *[Roz99]* Bd. 2. 1999, S. 487–550

[Sch07a] Schneider, Hans J.: *Graph Transformations – An Introduction to the Categorical Approach*. 2007 `http://www2.informatik.uni-erlangen.de/Personen/schneide/gtbook/`. – nocht nicht veröffentlicht

[Sch07b] Schösser, Andreas: *Graphersetzungsregelgewinnung aus Hochsprachen und deren Anwendung*, Universität Karlsruhe, IPD Goos, Diplomarbeit, September 2007. `http://www.info.uni-karlsruhe.de/papers/da_schoesser.pdf`

[Sco01] Scott, Kevin: On Proebsting's Law / Department of Computer Science, University of Virginia. Version: 2001. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.434`. 2001 (CS-2001-12). – Forschungsbericht

[SE02] Scholz, Bernhard ; Eckstein, Erik: Register allocation for irregular architectures. In: *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–527–0, S. 139–148

[SG00] Sreraman, N. ; Govindarajan, R.: A vectorizing compiler for multimedia extensions. In: *Int. J. Parallel Program.* 28 (2000), Nr. 4, S. 363–400. `http://dx.doi.org/10.1023/A:1007559022013`. – DOI 10.1023/A:1007559022013. – ISSN 0885–7458

[SG08] Schösser, Andreas ; Geiss, Rubino: Graph Rewriting for Hardware Dependent Program Optimizations. In: Schürr, Andy (Hrsg.) ; Nagl, Manfred (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)* Bd. 5088, Springer-Verlag, 2008 (LNCS)

[Sie15] Sierpinski, Waclaw F.: Sur une courbe dont tout point est un point de ramification. In: *Comptes Rendus hebdomadaires des séances de l'Acadeémie des Sciences* 160 (1915), S. 302–305

[Slo95] Sloane, Anthony M.: An evaluation of an automatically generated compiler. In: *ACM Trans. Program. Lang. Syst.* 17 (1995), Nr. 5,

S. 691–703. http://dx.doi.org/10.1145/213978.213980. – DOI 10.1145/213978.213980. – ISSN 0164–0925

[Slo07] Sloane, Neil J. A.: *The On-Line Encyclopedia of Integer Sequences.* http://www.research.att.com/~njas/sequences/. Version: August 2007

[Sta05] Standard Performance Evaluation Corporation: *All SPEC CPU2000 results published by SPEC page.* http://www.spec.org/cpu2000/results/cpu2000.html. Version: September 2005

[Sto77] Stoy, Joseph E.: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* Cambridge, MA, USA : MIT Press, 1977. – ISBN 0262191474

[SWZ99] Schürr, A. ; Winter, A. ; Zündorf, A.: PROGRES: Language and environment. In: *[Roz99]* Bd. 2, 1999, S. 487–550

[Sza05] Szalkowski, Adam M.: *Negative Anwendungsbedingungen für das suchprogrammbasierte Backend von GrGen*, Universität Karlsruhe, IPD Goos, Studienarbeit, Oktober 2005. http://www.info.uni-karlsruhe.de/papers/sa_szalkowski.pdf

[TBB+08] Taentzer, Gabriele ; Biermann, Enrico ; Bisztray, Dénes ; Bohnet, Bernd ; Boneva, Iovka ; Boronat, Artur ; Geiger, Leif ; Geiss, Rubino ; Horvath Ákos ; Kniemeyer, Ole ; Mens, Tom ; Ness, Benjamin ; Plump, Detlef ; Vajk, Tamás: Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In: Schürr, A. (Hrsg.) ; Nagl, M. (Hrsg.) ; Zündorf, A. (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)* Bd. 5088, Springer-Verlag, 2008 (LNCS). – http://www.springerlink.com/content/105633/

[Ten76] Tennent, R. D.: The denotational semantics of programming languages. In: *Commun. ACM* 19 (1976), Nr. 8, S. 437–453. http://dx.doi.org/10.1145/360303.360308. – DOI 10.1145/360303.360308. – ISSN 0001–0782

[TLB99] Trapp, Martin ; Lindenmaier, Götz ; Boesler, Boris: Documentation of the Intermediate Representation FIRM / Universität Karlsruhe, Fakultät für Informatik. Version: Dezember 1999. http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz. 1999 (1999-14). – Forschungsbericht

[Tra01] Trapp, Martin: *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.*, Universität Karlsruhe, Fakultät für Informatik, Diss., Oktober 2001. http://www.amazon.de/exec/obidos/ASIN/3540423524/qid=1039790257/sr=1-1/

[Ull76] Ullmann, J. R.: An Algorithm for Subgraph Isomorphism. In: *J. ACM* 23 (1976), Nr. 1, S. 31–42. http://dx.doi.org/10.1145/321921.321925. – DOI 10.1145/321921.321925. – ISSN 0004–5411

[VAB+08] Varró, Dániel ; Asztalos, Márk ; Bisztray, Dénes ; Boronat, Artur ; Dang, Duc-Hanh ; Geiss, Rubino ; Greenyer, Joel ; Gorp, Pieter V. ; Kniemeyer, Ole ; Narayanan, Anantha ; Rencis, Edgars ; Weinell, Erhard: Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In: Schürr, A. (Hrsg.) ; Nagl, M. (Hrsg.) ; Zündorf, A. (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)* Bd. 5088, Springer-Verlag, 2008 (LNCS)

[Var05] Varró, Gergely: *Graph transformation benchmarks page*. `http://www.cs.bme.hu/~gervarro/benchmark/2.0/`. Version: August 2005

[VFV04] Varró, Gergely ; Friedl, K. ; Varró, Dániel: Graph Transformations in Relational Databases. In: *Proc. GraBaTs 2004: Intl. Workshop on Graph Based Tools*, Elsevier, 2004

[VFV05] Varró, Gergely ; Friedl, Katalin ; Varró, Dániel: Graph Transformation in Relational Databases. In: Mens, T. (Hrsg.) ; Schürr, A. (Hrsg.) ; Taentzer, G. (Hrsg.): *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)* Bd. 127, No 1, Elsevier, März 2005 (Electronic Notes in Theoretical Computer Science), 167–180

[Vie88] Vielsack, Bertram ; GMD Forschungsstelle an der Universität Karlsruhe (Hrsg.): *Grammar Tools User Manual - LALR and ELL*. Karlsruhe, Germany: GMD Forschungsstelle an der Universität Karlsruhe, April 1988

[VSV05a] Varró, Gergely ; Schürr, A. ; Varró, Dániel: Benchmarking for Graph Transformation / Department of Computer Science and Information Theory, Budapest University of Technology and Economics. 2005. – Forschungsbericht

[VSV05b] Varró, Gergely ; Schürr, Andy ; Varró, Dániel: Benchmarking for Graph Transformation. In: *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2443–5, S. 79–88

[VVF05] Varró, Gergely ; Varró, Dániel ; Friedl, Katalin: Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In: Karsai, G. (Hrsg.) ; Taentzer, G. (Hrsg.): *GraMoT 2005, International Workshop on Graph and Model Transformations*, 2005 (ENTCS)

[Wes07] Wessendorf, Martin ; Sloane, Neil J. A. (Hrsg.): *A067771 – Number of vertices in Sierpinski triangle of order n*. `http://www.research.att.com/~njas/sequences/A067771`. Version: August 2007

[WG84] Waite, William ; Goos, Gerhard: *Compiler Construction*. Springer-Verlag, 1984 `ftp://i44ftp.info.uni-karlsruhe.de/public_html/papers/ggoos/CompilerConstruction.ps.gz`

[WH04] Wilson, Richard C. ; Hancock, Edwin R.: Levenshtein Distance for Graph Spectral Features. In: *ICPR '04: Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume 2.* Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2128–2, S. 489–492

[Wol95] Wolfe, M.J.: *High-Performance Compilers for Parallel Computing.* Addison-Wesley, Reading, Massachusetts, USA, 1995

[Zün96] Zündorf, A.: Graph Pattern Matching in Progres. In: *Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science* Bd. 1073, Springer-Verlag, 1996 (LNCS), S. 454–468