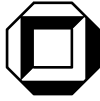


Universität Karlsruhe (TH)

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Dr. Goos

Graphersetzung für Optimierungen in der Codeerzeugung

Diplomarbeit

Sebastian Hack

Dezember 2003

Betreuer:
Dipl.-Inform. Rubino Geiß
Dr. Sabine Glesner

Verantwortlicher Betreuer:
Prof. Dr. Gerhard Goos

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Datum, Ort Unterschrift

Kurzfassung

Diese Diplomarbeit beschreibt die theoretischen und praktischen Eigenschaften eines Graphersetzungssystems zur Manipulation von graphbasierten Zwischendarstellungen in Übersetzern. Das Werkzeug und der damit verbundene Formalismus wurden besonders für den Einsatz bei der Codeerzeugung entwickelt bzw. angepasst. Ein Schwerpunkt der Entwicklung wurde auf eine modulare Architektur und genaue Trennung der einzelnen Komponenten eines Graphersetzungssystems gelegt um das System flexibel erweiterbar bzw. anpassbar zu halten. Das Problem des Findens eines homomorphen Teilgraphen wurde mittels Ausdrücken der relationalen Algebra formuliert. Die eigentliche Graphersetzung beruht auf dem etablierten SPO-Ansatz (Single-Pushout-Approach).

Inhaltsverzeichnis

1 Einleitung 1

1.1 Anforderungen 1
1.2 Lösungsansatz 2
1.3 Überblick 2

2 Grundlagen 3

2.1 Graphen 3
2.2 Graphersetzung 5
2.3 Algebraische Graphersetzung 6
2.3.1 Einleitende Überlegungen 7
2.3.2 Der Double-Pushout-Ansatz 8
2.3.3 Der Single-Pushout-Ansatz 12
2.3.4 Zusammenfassung 13

3 Verwandte Arbeiten 15

3.1 PGRS/Progres 15
3.1.1 Das Graphmodell 15
3.1.2 Ersetzung und Matchen 16
3.1.3 Besonderheiten 16
3.2 Optimix 17
3.2.1 Das Graphmodell 17
3.2.2 Ersetzung und Matchen 17
3.2.3 Besonderheiten 18
3.3 Agg 18

4 Unser Ansatz 19

4.1 Programme als Graphen 19
4.1.1 Die Zwischendarstellung Firm 19
4.2 Das Graphmodell 20
4.3 Der Ersetzungsschritt 23
4.4 Das Finden von Graphmustern 24
4.4.1 Das Finden von Graphmustern mit relationaler Algebra . 26
4.5 Übersicht und Vergleich 32

5 GRGEN 33

5.1 Architektur der LibGr 33
5.2 GrGen 34
5.2.1 Die Spezifikationssprache von GrGen 34
5.3 Vergleich zu den Implementierungen der anderen Ansätze . . . 40

6 Erste Experimente 43

7 Zusammenfassung und Ausblick 47

A Grundbegriffe der Kategorientheorie 49

A.1 Kategorien und Diagramme 49
A.2 Universelle Konstruktionen 50

B Grundbegriffe der Relationalen Algebra 53

B.1 Definitionen 53
B.2 Operationen auf Relationen 53
B.3 Terme 56

C Die Spezifikationssprache von GrGen 57

1 Einleitung

Codeerzeugung für Prozessoren mit herkömmlichen RISC-Befehlssätzen wird seit langem mit Werkzeugeinsatz betrieben. Moderne Prozessorarchitekturen verfügen zusätzlich über Komplexbefehle. Diese Befehle, die zur Beschleunigung von Multimedia-Anwendungen (Video-Kodierung/-Dekodierung) eingeführt wurden, haben sich mittlerweile zu vollständigen Vektorbefehlssätzen entwickelt, die zur Zeit Register von einer Breite bis 128 Bit (sowohl Ganzzahl- als auch Gleitkomma-Arithmetik) verarbeiten können. Beispiele hierfür sind Intel's MMX und SSE, sowie Motorola's AltiVec. Diese Befehle können strukturbedingt mit gängigen Verfahren der Befehlsauswahl nicht erzeugt werden. Deshalb muss eine vorgeschaltete Optimierung die Komplexbefehle identifizieren und für klassische Codeerzeuger handhabbar machen.

Moderne Übersetzer verwenden meist graphbasierten Zwischendarstellungen. Das Eingabeprogramm wird dabei durch Daten- und Steuerflussgraphen repräsentiert. Optimierungen stellen sich auf dieser Art von Zwischendarstellung als Transformation von Graphen dar. Diese Transformationen werden momentan meist ausprogrammiert, was fehleranfällig und nur für Transformationen auf kleinen Bereichen der Graphen überschaubar ist. Gerade beim Versuch größere Muster (10-50) in einem Graphen zu finden, wie sie bei Komplexbefehlen auftreten, zeigt sich, dass der Einsatz eines Werkzeugs unabdingbar ist, da die Mustersuche für jedes zu findende Muster immer neu geschrieben werden muss. Um die bei Komplexbefehlen auftretenden Muster elegant aufzuspüren und umzuwandeln ist der Einsatz eines Graphersetzungssystems (GES) notwendig. Ziel dieser Arbeit ist die Entwicklung eines Werkzeugs, mit dem Graphersetzungssysteme spezifiziert, generiert und so in einen Übersetzer integriert werden können.

1.1 Anforderungen

Hierbei werden durch das skizzierte Einsatzszenarium folgende Anforderungen an das zu entwickelnde Werkzeug gestellt:

1. Eine Sprache zur einfachen Spezifikation von Graphersetzungsregeln. Eine Regel setzt sich hierbei aus einem zu findenden Teilgraphen eines attributierten Zwischendarstellungs-Graphen und einem Graphen, in den der gefundene Teilgraph überführt wird, zusammen.

2. Generierung des Graphersetzungssystems aus der Spezifikation zur Integration in eine Übersetzerinfrastruktur.

3. Modularität im Hinblick auf die interne Repräsentation der Graphen und des Matchers, da hier Optimierungspotenzial zu erwarten ist.
4. Selektive Aktivierbarkeit der Regeln aus dem Übersetzer.
5. Theoretische Fundierung des Graphmodells, des Ersetzungsmechanismus und des Matchers zur Berücksichtigung von Verifikationsaspekten.

1.2 Lösungsansatz

Die Modularität wird durch strikte, von der Theorie suggerierte Trennung des Ersetzens und des Matchens erreicht; so können die Implementierungen der Graphen und Matcher flexibel ausgetauscht werden, um die Anpassung an verschiedene Systeme zu erreichen und Leistungssteigerungen zu erzielen, ohne die formale Beschreibung der Graphersetzung zu durchbrechen. In dieser Arbeit wird das Matchen mittels Zurückführung auf einen Ausdruck der relationalen Algebra an ein relationales Datenbanksystem delegiert, was zusätzlich eine effiziente Haltung der Graphen ermöglicht. Die formale Grundlage für die eigentliche Graphersetzung bietet der etablierte SPO-Ansatz, der alle für diese Arbeit relevanten Ersetzungen beschreibt.

Die in Abschnitt 5.2.1 vorgestellte Sprache ermöglicht ein einfaches Spezifizieren von Graphersetzungsregeln. Das Werkzeug GrGen (siehe Abschnitt 5.2) setzt diese Spezifikation dann in ausführbaren C-Code um, der als Modul im Zusammenhang mit der Bibliothek LibGr (siehe Abschnitt 5.1) verwendet werden kann. Diese Bibliothek bietet eine abstrakte Schnittstelle zur gesteuerten Anwendung von Graphersetzungsregeln und ist von der konkreten Implementierung der Graphen, des Matchers und der Ersetzung unabhängig. Somit können diese Teile flexibel ausgetauscht werden, ohne das Programm, das die Graphersetzung verwendet, modifizieren zu müssen.

1.3 Überblick

Zunächst werden im nächsten Kapitel die für diese Arbeit nötigen Grundlagen der Graphentheorie und der algebraischen Graphersetzung vorgestellt. Im zweiten Teil des nächsten Kapitels werden elementare Konstrukte der Kategorientheorie verwendet, die im Anhang A erläutert werden. In Kapitel 3 werden drei existierende Werkzeuge aus dem Bereich der Graphersetzung und deren theoretische Konzepte vorgestellt. Kapitel 4 stellt die Anforderungen, die eine graphbasierte Zwischendarstellung an das formale Modell eines Graphen stellt, vor. Desweiteren wird beschrieben, wie man mittels SPO-Ansatz Graphen dieses Modells transformiert. Darauf folgend wird eine Methode vorgestellt, wie das Finden von Graphmustern (das Matchen) durch Formeln der relationalen Algebra ausgedrückt werden kann. Die dazu nötigen grundlegenden Begriffe der relationalen Algebra werden im Anhang B besprochen. Zuletzt wird der vorgestellte Ansatz zu den in Kapitel 3 diskutierten Verfahren in Beziehung gesetzt. Kapitel 5 gibt einen Überblick über das in dieser Arbeit entwickelte Werkzeug GrGen, das die in Kapitel 4 beschriebenen Konzepte praktisch umsetzt. Kapitel 7 enthält eine Zusammenfassung und einen Ausblick.

2 Grundlagen

Graphersetzung ist ein theoretisch anspruchsvolles Gebiet. Im Gegensatz zu den formalen Sprachen und der Termersetzung, die beide auf entwickelten und etablierten Theorien beruhen, finden sich bei der Graphersetzung mehrere theoretische Konzepte, die unterschiedliche Ersetzungsmechanismen für verschiedene Graphmodelle beschreiben. Von einem Standard-Graphersetzungsverfahren kann somit nicht die Rede sein.

Wir wollen zunächst die grundlegenden, in dieser Arbeit benötigten Definitionen aus der Graphentheorie geben und die grundlegenden algebraischen Graphersetzungsansätze vorstellen.

2.1 Graphen

Je nach Anwendung betrachtet man verschiedene Modelle von Graphen mit unterschiedlichen mathematischen Darstellungen. So kann man Ecken und Kanten typisieren und attributieren, den Kanten eine Richtung geben, oder bei den sogenannten *Multigraphen* mehrere Kanten zwischen zwei Ecken zulassen. Eine Einführung in die Graphentheorie findet man z.B. in [Hal80]

Die einfachste Darstellung ist sicherlich die relationale. Man betrachtet einen Graphen G als Paar (E, K), wobei E die Ecken und K die Kanten des Graphen darstellen. K ist hierbei eine Relation auf E, also $K = E \times E$. Ist in einem Graphen eine Ecke a mit einer Ecke b verbunden, so schreibt man $(a, b) \in K$. Bei ungerichteten Graphen gilt ferner: $(a, b) \in K \iff (b, a) \in K$. Alle Graphen sollen im folgenden gerichtet sein.

Diese einfache Darstellung ist schon für Multigraphen nicht ausreichend. Bei Multigraphen können zwei Ecken durch mehrere Kanten verbunden sein, was mit der Modellierung von Kanten als zweistellige Relation nicht darstellbar ist. Natürlich kann man mehrstellige Relationen verwenden, in der Literatur hingegen ist folgende Darstellung gebräuchlicher.

Definition 2.1 (Gerichteter Multigraph) Ein Multigraph $M = (E, K, \mathrm{src}, \mathrm{tgt})$ ist ein Quadrupel, wobei E die Menge der Ecken und K die Menge der Kanten darstellen. Die Abbildungen

$$\mathrm{src} : K \to E, \quad \mathrm{tgt} : K \to E$$

einer Kante Quell- und Zielecken zu. Meinen wir die Untermenge der Ecken (bzw. Kanten) eines Multigraphen M, so schreiben wir E_M bzw. K_M. Sei

e eine Ecke eines Multigraphen M. Die Menge der Kanten, gegeben durch $\{k \mid \operatorname{src} k = e\}$ bezeichnen wir mit $e^\bullet$. Die Mächtigkeit $|e^\bullet|$ von $e^\bullet$ heißt *Ausgangsgrad* der Ecke e. Analog definieren wir ${}^\bullet e = \{k \mid \operatorname{tgt} k = e\}$ und bezeichnen $|{}^\bullet e|$ als *Eingangsgrad*.

Wir nennen eine Kante k zu einer Ecke e inzident, wenn $\operatorname{src} k = e \vee \operatorname{tgt} k = e$ und definieren hierfür das Prädikat:

$$\begin{aligned} \operatorname{inc} : K \times E &\rightarrow \mathbb{B} \\ (k, e) &\mapsto k \in {}^\bullet e \cup e^\bullet \end{aligned}$$

Zuletzt sei mit

$$\begin{aligned} \gamma : E &\rightarrow \mathbb{N} \\ e &\mapsto |{}^\bullet e| + |e^\bullet| \end{aligned}$$

der *Grad* einer Ecke definiert.

Ist in einem Graphen eine Ecke a mit einer Ecke b über eine Kante k verbunden, so ist $\operatorname{tgt} k = b$ und $\operatorname{src} k = a$. Eine Kante k heißt *Schlinge*, wenn $\operatorname{src} k = \operatorname{tgt} k$.

Definition 2.2 (Teilgraph) Ein Graph $G' = (E', K', \operatorname{src}', \operatorname{tgt}')$ heißt Teilgraph eines Graphen $G = (E, K, \operatorname{src}, \operatorname{tgt})$, wenn $E' \subseteq E$ und $K' \subseteq K$. src' und tgt' sind auf K' eingeschränkt: $\operatorname{src}' = \operatorname{src}|_{K'}$, $\operatorname{tgt}' = \operatorname{tgt}|_{K'}$. Man schreibt dann auch $G' \subseteq G$.

Definition 2.3 (Untergraph) G' heißt Untergraph von G, wenn G' ein Teilgraph von G ist, und alle Kanten aus G, die Ecken aus G' verbinden auch in G' vorhanden sind:

$$\forall k \in K_G : (\operatorname{src} k \in E_{G'} \wedge \operatorname{tgt} k \in E_{G'}) \implies k \in K_{G'}$$

Somit sind Untergraphen durch die Menge der Ecken eindeutig bestimmt.

Oft ist man an Graphen interessiert, bei denen Ecken und Kanten markiert werden können. Man spricht dann von einem markierten Graphen und es existieren dann zwei Abbildungen, die jeder Ecke und jeder Kante ein Element einer Menge zuordnen.

Definition 2.4 (Markierter Graph) Ein gerichteter Graph ist ein Tupel $G = (E, K, \operatorname{src}, \operatorname{tgt}, \Sigma_E, \Sigma_K, \operatorname{lab}_E, \operatorname{lab}_K)$, mit

$$\operatorname{lab}_E : E \rightarrow \Sigma_E, \quad \operatorname{lab}_K : K \rightarrow \Sigma_K$$

Desweiteren ist man noch an Abbildungen von Graphen auf Graphen interessiert, welche die „Graphstruktur“ erhalten.

Definition 2.5 (Graph-Homomorphismus) Seien $G_1 = (E_1, K_1, \operatorname{src}_1, \operatorname{tgt}_1)$ und $G_2 = (E_2, K_2, \operatorname{src}_2, \operatorname{tgt}_2)$ Graphen wie in Definition 2.1. Ein Graph-Homomorphismus ϕ ist ein Paar $(\phi_E : E_1 \rightarrow E_2,\ \phi_K : K_1 \rightarrow K_2)$ für den für alle $k \in K_1$ gilt:

$$\operatorname{src}_2 \circ \phi_K = \phi_E \circ \operatorname{src}_1$$

und

$$\operatorname{tgt}_2 \circ \phi_K = \phi_E \circ \operatorname{tgt}_1$$

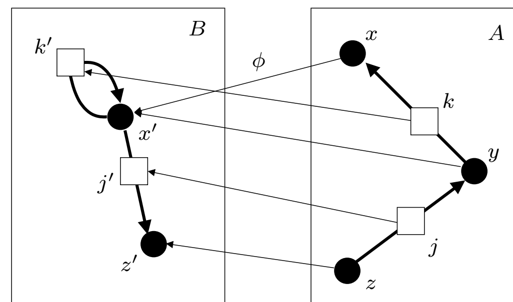
Definition 2.6 (Partieller Graph-Homomorphismus) Sei G ein Multigraph. Ein partieller Graph-Homomorphismus ist ein totaler Graph-Homomorphismus eines Teilgraphen $S \subseteq G$ zu einem Graphen H.

Die Definition des Graph-Homomorphismus läßt sich auf markierte Graphen erweitern.

Definition 2.7 (Graph-Homomorphismus auf markierten Graphen) Seien G und H zwei markierte Graphen wie in Definition 2.4. Eine Abbildung ϕ ist ein Graph-Homomorphismus von G nach H, wenn ϕ ein Graph-Homomorphismus im Sinne von Definition 2.5 ist und die Markierungen erhält:

$$\forall e \in E_G : \text{lab}_{E_G}(e) = \text{lab}_{E_H}(\phi(e)), \quad \forall k \in K_G : \text{lab}_{K_G}(k) = \text{lab}_{K_H}(\phi(k))$$

Im folgenden werden zur besseren Darstellung von Graph-Homomorphismen Diagramme der folgenden Art verwendet[1]:



Die weißen Quadrate stellen in Verbindung mit den dicken schwarzen Pfeilen die Kanten, die schwarzen Kreise die Ecken eines Graphen dar. Die dünnen Pfeile repräsentieren die Abbildungsvorschrift eines Graph-Homomorphimus. In diesem Diagramm wird ein Graph-Homomorphimus $\phi : A \to B$ mit folgender Abbildungsvorschrift dargestellt:

$$\phi(x) = \phi(y) = x',\ \phi(z) = z',\ \phi(k) = k',\ \phi(j) = j'$$

2.2 Graphersetzung

Ersetzungsmechanismen auf Zeichenketten (die formalen Sprachen) und Termen zählen zu den Grundlagen der theoretischen Informatik. Eine Erweiterung auf Graphen scheint nur konsequent und wird seit den späten 1960er Jahren untersucht. Erste Ansätze zielten auf das Finden eines Pendants zu den Chomsky-2 Grammatiken ab und wurden z.B. in der Bildanalyse verwendet. Hierbei wurden aus digitalisierten Bildern (z.B. Aufnahmen von Blasenkammer-Experimenten) Graphen errechnet, die die Struktur der Bilder wiederspiegelten. Mit Hilfe von Graphgrammatiken wurde versucht, Teilmuster, an denen man interessiert war, zu identifizieren und durch „Platzhalter", die das Teilmuster dann beschrieben, zu ersetzen. Weitere Anwendungen fanden sich auch im VLSI-Bereich, vornehmlich zur Unterstützung des Schaltungsentwurfs.

[1]Diese Art der Darstellung stammt aus [Kah02]

Reizvoll ist Graphersetzung vor allem deswegen, weil sehr viele Probleme sehr gut mithilfe von Graphen spezifiziert werden können. Was läge nun näher als ein Instrument um diese „einfachen“ Strukturen systematisch ineinander umzuformen? Aber gerade diese Allgemeinheit, die die Ausdrucksstärke von Graphen begründet, erschwert das Finden von geeigneten Formalismen für solche Ersetzungsmechanismen ungemein.

Die grundlegende Eigenschaft aller GES ist das Vorhandensein von Produktionen, die im allgemeinen einen Graphen L in einen Graphen R überführen. Man schreibt für eine solche Produktion p dann auch:

$$p : L \to R$$

Eine Produktion wird durch eine *direkte Ableitung* auf einen Eingabegraphen G angewandt. G wird auch als *Muttergraph* bezeichnet. Hierzu bedarf es eines sog. *Match* m, das eine zu L passende Entsprechung in G beschreibt. Mit dem Paar (p, m) geht der Eingabegraph in den Ergebnisgraphen H über. Man schreibt dafür:

$$G \overset{p,m}{\Longrightarrow} H$$

Grundsätzlich sind im Kontext von GES drei Fragen zu beantworten:

1. Was ist ein Graph?
2. Wie findet man einen Teilgraphen G, der L entspricht? Dies entspricht der Bestimmung der oben genannten Abbildung m.
3. Wie definiert man das Ersetzen von L durch R?

Die Antwort auf die erste Frage klärt das Modell eines Graphen. Je nach Kontext und Einsatzgebiet des GES können verschiedene Arten von Graphen vonnöten sein. Die Definitionen aus 2.1 können zum Beispiel um Markierungen für Ecken und Kanten, Relationen zwischen Markierungen, etc. erweitert werden. Oft versucht man, über die reine Graphstruktur hinaus zusätzliche, meist vom Einsatzgebiet des GES abhängige Informationen in das Graphmodell zu geben. Eine Einführung in die verschiedenen Graphersetzungsansätze bieten zum Beispiel [BFG96], [AEH+99] und [HB02]. Wir betrachten im folgenden die beiden populärsten Ansätze der algebraischen Ansatz. Zu Beginn stellen wir den Double-Pushout-Ansatz (SPO) vor, um die kategorientheoretische Formalisierung zu verdeutlichen. Darauf aufbauend diskutieren wir den Single-Pushout-Ansatz (SPO), da er für die Modifikation von Zwischendarstellungsgraphen aus Übersetzern ist.

2.3 Algebraische Graphersetzung

Die algebraische Graphersetzung wurde 1973 von Ehrig in [EPS73] mit dem Double-Pushout-Ansatz erstmals beschrieben und verwendet Konstrukte der Kategorientheorie (siehe Anhang A) zur Formalisierung der Graphersetzung. Eine Einführung in die algebraischen Ansätze zur Graphersetzung findet man in [EKL91] und [CEH+97]. Auch die einführenden Kapitel von [Kah02] sind empfehlenswert.

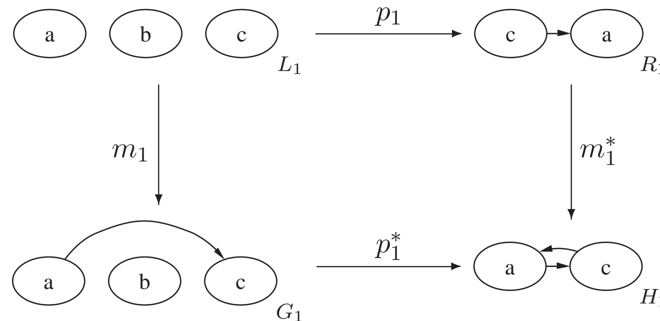


Abbildung 2.1: Eine einfache Produktion p_1 und deren Anwendung in einem Graphen G_1

2.3.1 Einleitende Überlegungen

Gewöhnlich stellt man eine *Produktion* p durch das Paar (L, R) dar. L heißt die linke Seite der Regel und R die rechte Seite. Bei der Anwendung von p auf einen Graphen G wird zunächst ein zu L homomorpher Teilgraph in G gesucht[2]. Wird dieser gefunden, so existiert ein Graph-Homomorphismus $m : L \rightarrow G$, der die Entsprechung von L in G darstellt. Elemente des Graphen (sowohl Ecken als auch Kanten), die in R und L vorhanden sind, werden durch die Anwendung von p nicht berührt. Findet sich jedoch ein Element, das in L vorhanden ist, in R aber fehlt, so wird es bei der Anwendung von p gelöscht. Ein Element, das in R vorhanden ist, aber in L fehlt, wird dem Graphen bei der Anwendung der Produktion hinzugefügt. Der resultierende Graph wird mit H bezeichnet. Abbildung 2.1 veranschaulicht dies. Dieses Verfahren bringt zwei Probleme mit sich:

1. Da m „nur“ ein Graph-Homomorphimus und nicht zwangsläufig injektiv ist, kann es sein, das zwei Ecken a, b aus L auf eine Ecke a' in G geworfen werden. Angenommen $a \in R$ und $b \notin R$. Nach obigem Schema müßte a' sowohl gelöscht werden als auch erhalten bleiben.

2. Angenommen $a \in L$ und $a \notin R$. a wäre also bei Anwendung von p aus G zu löschen. Angenommen es existieren Kanten, die von $m(a)$ zu Ecken führen, die nicht in einem Bild von L in G liegen. Diesen Kanten würde man durch die Löschung von $m(a)$ einen Endpunkt entziehen und somit die Grapheigenschaft zerstören.

Aus diesen beiden Fällen gibt es unterschiedliche Auswege. Einer wäre, bei Fall 1 das Löschen oder das Erhalten zu priorisieren. Bei Fall 2 könnte man implizit alle Kanten $k \notin m(L)$ mit $\mathrm{src}(k) = m(a) \vee \mathrm{tgt}(k) = m(a)$ löschen. Eine weitere Lösung ist, im Konfliktfall die Regelanwendung zu verbieten. Die beiden Ansätze (DPO und SPO) unterscheiden sich genau in der Handhabung dieser Fälle. Wir wollen zunächst den DPO-Ansatz vorstellen, da er eine

[2] engl.: to match a subgraph

leichtere Einführung in die Verwendung der kategorientheoretischen Konstrukte ermöglicht.

2.3.2 Der Double-Pushout-Ansatz

Der Double-Pushout-Ansatz ist der restriktivere der beiden Ansätze. Tritt bei der Anwendung einer Regel p einer der beiden oben skizzierten Problemfälle auf, so wird die Regelanwendung verboten und der Graph bleibt unmodifiziert.

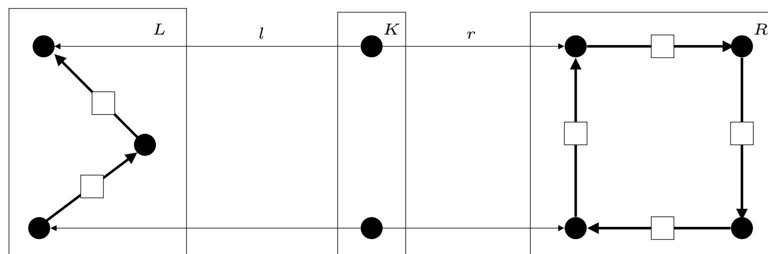
Eine Produktion p im DPO ist geben durch die beiden schon bekannten Seiten L und R, dem sogenannten *Klebegraphen*[3] K und zwei Graph-Homomorphismen $l : K \to L$, $r : K \to R$, die Ecken und Kanten aus dem Klebegraphen auf die linke bzw. rechte Seite abbilden. Man schreibt dann:

$$p = L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$$

Die Anwendung einer Produktion kann dann mithilfe zweier Pushout-Diagramme[4] (1), (2) dargestellt werden:

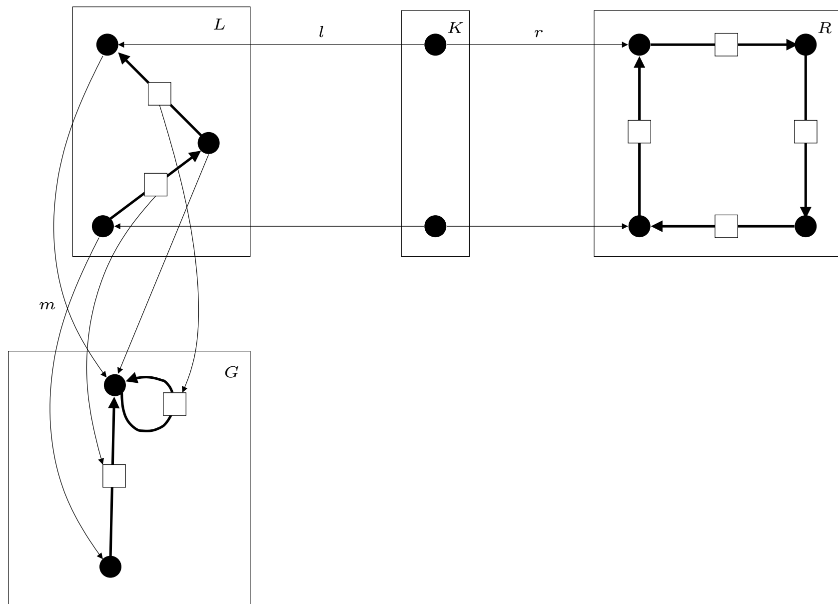
$$\begin{array}{ccccc}
L & \xleftarrow{\;l\;} & K & \xrightarrow{\;r\;} & R \\
{\scriptstyle m}\downarrow & (1) & {\scriptstyle d}\downarrow & (2) & \downarrow{\scriptstyle m^*} \\
G & \xleftarrow[\;l^*\;]{} & D & \xrightarrow[\;r^*\;]{} & H
\end{array}$$

Sowohl Diagramm (1) als auch Diagramm (2) sind Pushouts (siehe Definition A.8) in der Kategorie **Graph** (siehe Beispiel A.3), daher auch der Name *Double-Pushout*-Ansatz. D ist der Graph, der aus G hervorgeht, wenn alle Graphelemente, die in L, aber nicht in K enthalten sind, gelöscht wurden. H ist der Graph, der aus D nach dem Hinzufügen der Elemente aus R, die nicht in K sind, entsteht. Folgende Produktion p soll im Verlauf dieses Abschnitts als Beispiel dienen[5]:



Bei der Anwendung einer Produktion hält man zunächst nur die Produktion selbst, den Graphen G und den Morphismus m, der L auf einen homomorphen Subgraphen in G abbildet[6], in Händen wie folgende Abbildung zeigt:

[3] engl.: gluing graph
[4] siehe auch A.1 zur näheren Erläuterung von Diagrammen
[5] Das Beispiel stammt aus [Kah02]
[6] Man nennt m auch einen *Redex*

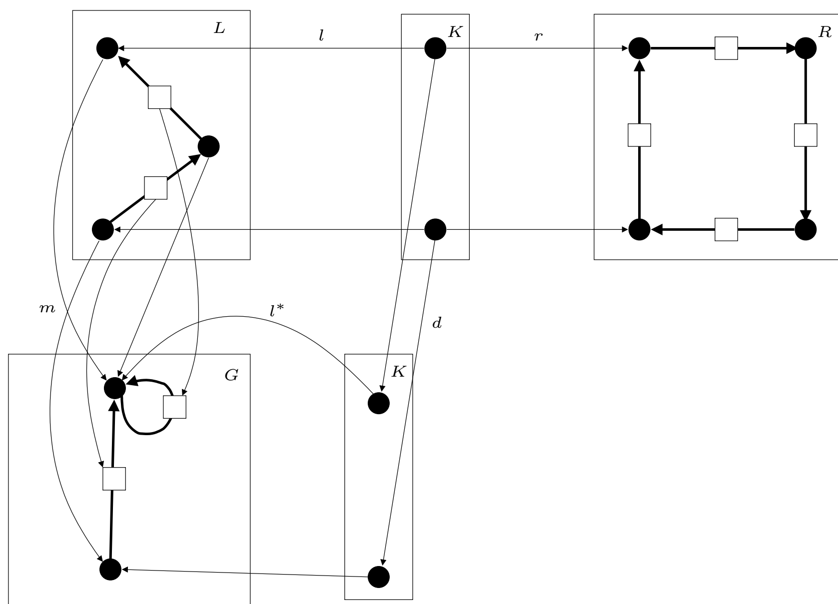


Gesucht ist nun der Graph D mit den Morphismen d und l^*. Jedoch können d und l^* nicht beliebig gewählt werden. Sie müssen bestimmte Bedingungen erfüllen, die garantieren, dass die in Abschnitt 2.3.1 beschriebenen Problemfälle nicht auftreten. Es zeigt sich, dass diese Bedingungen genau dann erfüllt sind, wenn (D, d, l^*) ein *Pushout-Komplement* ist. Damit wird G zum *Pushout-Objekt* von (l, d).

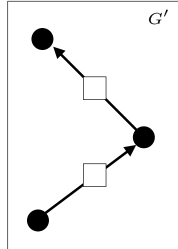
Im obigen Beispiel kann d nur die Identität von K sein, denn eine Bedingung dafür, dass (m, l^*) ein Pushout wird ist:

$$m \circ l = l^* \circ d$$

Die beiden Ecken in K können also nicht zusammenfallen. Also ergibt sich für (D, d, l^*) folgendes Bild:



Nun ist G aber kein Pushout von (l, g), da die Universalitätseigenschaft verletzt ist. Sie besagt, dass von G zu jedem anderen Pushout-Objekt G' von (l, d) genau ein Morphismus $k : G \to G'$ existieren muss. Für ein G' der Form



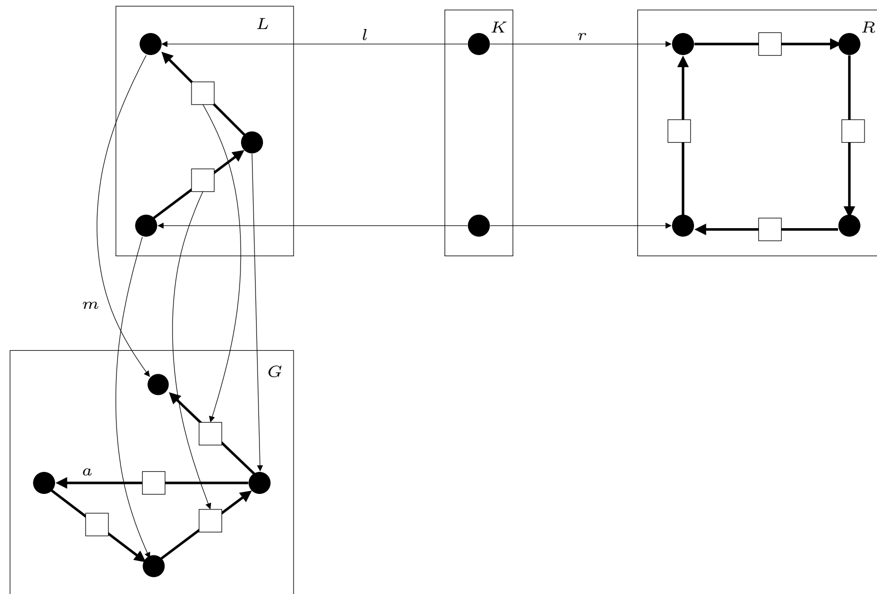
ist das nicht gegeben, da G zu G' nicht homomorph ist. Die Ableitung (p, m) ist also nicht zulässig. In der Tat ist dieses Beispiel identisch zu Problem 1 in Abschnitt 2.3.1. Eine Ecke aus $L \setminus K$ und eine Ecke aus K werden von m auf eine Ecke in G geworfen. Nun ist nicht klar, ob diese Ecke erhalten (weil in K), oder gelöscht (weil in $L \setminus K$) werden soll. Also ist folgende Bedingung für die Existenz eines Pushouts notwendig:

Definition 2.8 Zwei Morphismen $l : K \to L$ und $m : L \to G$ erfüllen die *Identifikationsbedingung*, wenn für alle Ecken $a, b \in L$ gilt:

$$a \neq b \wedge m(a) = m(b) \Rightarrow \exists a', b' \in K : a = l(a') \wedge b = l(b')$$

Das heißt, dass alle Ecken in L, die durch m auf ein und dieselbe Ecke in G abgebildet werden, Bilder von l sein müssen.

Ein weiteres Problem tritt bei folgendem Beispiel auf: Hier kann die Kante a weder von m noch von l^* rekonstruiert werden.



Somit ist G wieder kein Pushout von (l, d), da weder m noch l^* die Kante a reproduzieren können. Somit gibt es wieder ein G', zu dem es keinen Morphismus $k : G \to G'$ gibt, womit die Universalitätseigenschaft verletzt ist. Diese

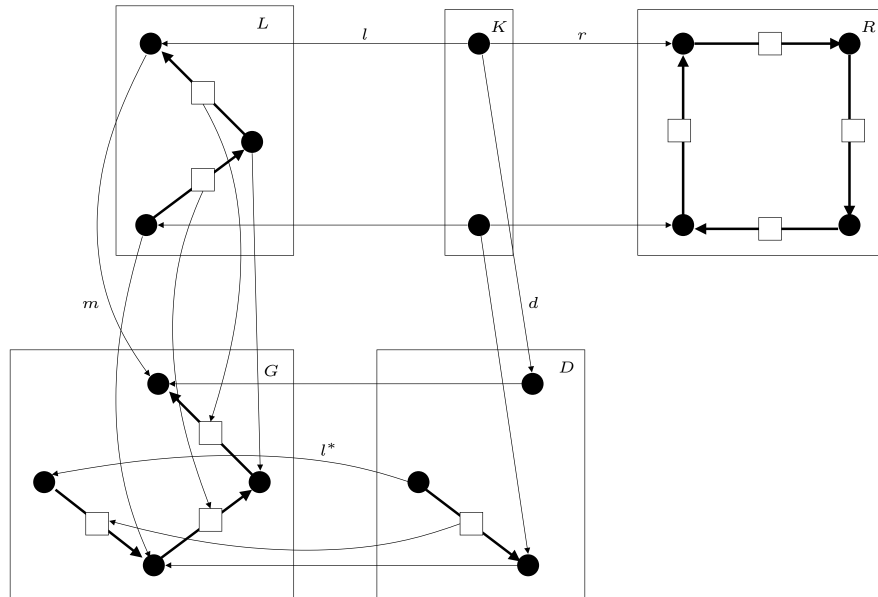
Situation ist äquivalent zu Problem 2 in Abschnitt 2.3.1. Die Kante a kommt von einer Ecke $e \in G$, die durch p gelöscht werden soll und zeigt auf eine Ecke, die *nicht* im Bild von l^* liegt, also kein Urbild in D hat. Diese Kante hätte nach Löschen von e kein Ziel mehr, wodurch die Graphstruktur zerstört würde. Man sagt auch die Kante *baumelt*[7]. Der DPO untersagt in einem solchen Fall die Regelanwendung. Folgende Bedingung ist zudem notwendig für die Existenz eines Pushouts:

Definition 2.9 Zwei Morphismen erfüllen die *Baumelbedingung*[8] genau dann, wenn:

$$\forall k \in G_K \setminus m(L) : \forall x \in m_E(L \setminus l(K)) : \neg \operatorname{inc}(x, k)$$

$G_K \setminus m(L)$ ist die Menge aller Kanten, die im Eingabegraphen, aber nicht in der linken Seite der Regel vorkommen. $m_E(L \setminus l(K))$ bezeichnet alle Ecken, die zwar in L sind, aber keine Bilder von l sind. (Sie sind also Elemente der linken Regelseite, aber *nicht* des Klebegraphen.)

Folgende Abbildung zeigt einen gültigen Muttergraphen G, auf den die Regel angewendet werden kann. Hier fehlt die baumelnde Kante a.



Man kann zeigen, dass für die Existenz eines Pushout-Komplements bezüglich (D, d, l^*) folgende Bedingung hinreichend ist:

Definition 2.10 (Klebebedingung) Die *Klebebedingung* ist erfüllt, wenn die Identifikationsbedingung und die Baumelbedingung erfüllt ist.

[7] engl.: dangling edge
[8] engl.: dangling condition

Diskussion

Die Baumelbedingung offenbart die eigentliche Schwäche des DPO. Das Löschen einer Ecke ist nicht ohne weiteres möglich. Die Baumelbedingung schreibt vor, dass alle Kanten, die zu einer zu löschenden Ecke inzident sind, im Klebegraph zu finden sein müssen. Das bedeutet, dass der Kontext der zu löschenden Ecke spezifiziert werden muss; ein implizites Löschen aller inzidenten Kanten ist nicht möglich. Andererseits bietet der DPO einen klaren und eleganten Formalismus zur Beschreibung der Graphersetzung, indem er sich auf grundlegende Konstrukte der Kategorientheorie beschränkt.

Das Problem 1 aus Abschnitt 2.3.1 ist (zumindest im Zusammenhang dieser Arbeit) weniger tragisch, da man meist an isomorphen Entsprechungen von L in G interessiert sein wird.

2.3.3 Der Single-Pushout-Ansatz

Der *Single-Pushout-Ansatz* (SPO) verwendet einen anderen Ansatz zur Modellierung der zu löschenden bzw. zu addierenden Ecken und Kanten. Die Produktionen besitzen keinen Klebegraphen K mehr, sondern werden mithilfe eines partiellen Graph-Homomorphismus ϕ dargestellt:

$$p = L \xrightarrow{\phi} R$$

im Gegensatz zu

$$p = L \xleftarrow{l} K \xrightarrow{r} R$$

beim DPO. Die Partialität von ϕ wird dazu verwendet, die zu löschenden bzw. hinzuzufügenden Elemente auszuzeichnen. Sei S_L der Teilgraph von L, der durch ϕ auf $S_R \subseteq R$ abgebildet wird. Ist ein Element in $L \setminus S_L$, so wird es bei der Regelanwendung gelöscht. Ist ein Element in $R \setminus S_R$, so wird es hinzugefügt. ϕ stellt die Beziehung der Elemente aus S_L mit denen aus S_R her; sie bleiben bei der Regelanwendung erhalten.

Man modelliert den Ersetzungsschritt als Pushout in der Kategorie **GraphP** (der Kategorie der Graphen und der partiellen Graph-Homomorphismen). Vom Morphismus $m : L \to G$, der L auf die Entsprechung von L in G abbildet, wird jedoch Totalität gefordert, da eine partielle Entsprechung wenig Sinn ergibt. Der Ersetzungsschritt kann dann wieder als kommutierendes Diagramm dargestellt werden:

$$\begin{array}{ccc} L & \xrightarrow{\phi} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow{\scriptstyle m^*} \\ G & \xrightarrow[\phi^*]{} & H \end{array}$$

Abbildung 2.2 zeigt die beiden Fälle aus 2.3.1. In Fall 1 identifiziert der Entsprechungs-Morphismus m zwei Ecken aus L in G. Eine Ecke der beiden soll aber gelöscht

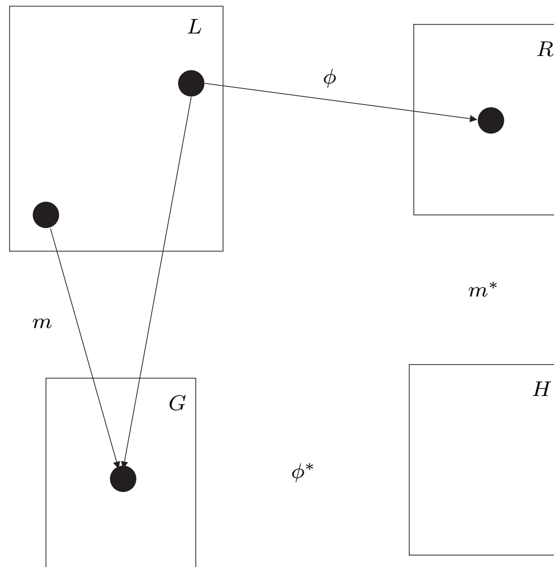
werden, da der partielle Graph-Homomorphismus l nur diese nach R abbildet. Da (H, ϕ^*, m^*) ein Pushout von (L, ϕ, m) sein muss, kann es keine Ecke in H mehr geben. Somit wird über die Pushout-Eigenschaft das Löschen der Ecke bevorzugt.

In Fall 2 wird die einzige Ecke in L gelöscht, da der partielle Homomorphimus ϕ leer ist. Nun wird diese Ecke durch m aber auf eine Ecke geworfen, die mit einer anderen Ecke in $G \setminus m(L)$ verbunden ist. Diese Situation verhinderte im DPO die Anwendung der Regel. Auch hier ist mit der Pushout-Bedingung das Löschen der baumelnden Kante verbunden. Würde die Kante nicht gelöscht werden, so müßte die Ecke aus $m(L)$ auch in H zu finden sein, was die Kommutativitätseigenschaft des Pushouts verletzen würde.

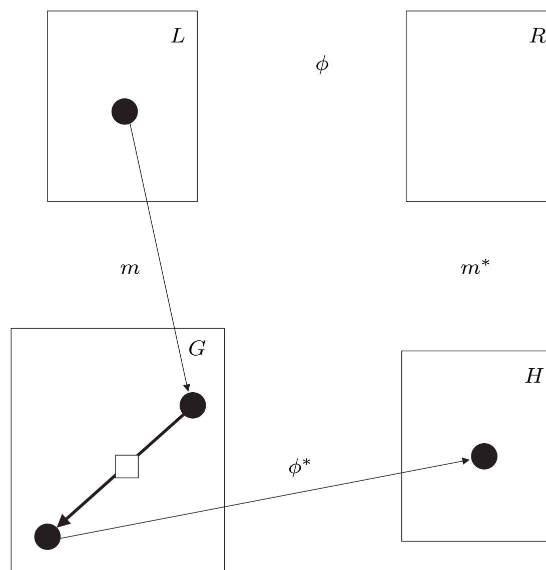
2.3.4 Zusammenfassung

Der DPO bietet eine einfache Theorie, die das mathematische Argumentieren über den Ersetzungsmechanismus erheblich erleichtert. Im Gegensatz dazu stehen die doch beschränkten Möglichkeiten, die von der Klebebedingung her rühren. Der SPO kann diese Nachteile wettmachen, indem er die totalen Graph-Homomorphismen und den Klebegraphen durch partielle Graph-Homomorphismen ersetzt. Beiden Ansätzen fehlt aber die Möglichkeit, *Graphvariablen* zu spezifizieren, die ganze Teilgraphen in linken Seiten repräsentieren. Dies hätte den Vorteil, dass man gesamte Teilgraphen vervielfältigen kann. Diese Eigenschaften erfordern kompliziertere mathematische Modelle oder sind in ihrer Modellierung wesentlich unintuitiver als die beiden hier vorgestellten Ansätze. Der interessierte Leser sei hier auf den *Pullback-Ansatz* von Bauderon [Bau95] und die Habilitation von Kahl [Kah02] verwiesen, der die Verwendung von Homomorphismen in der algebraischen Graphersetzung auf Relationen ausdehnt.

Es wurden natürlich nicht nur die Ersetzungsschritte im einzelnen untersucht, sondern auch Mengen von Regeln, die dann eine *Graphgrammatik* bilden. So existieren vielfältige Aussagen über die parallele Anwendbarkeit der Regeln. Siehe hierzu auch [CEH+97].



(a) Fall 1



(b) Fall 2

Abbildung 2.2: Die Fälle 1 und 2 aus Abschnitt 2.3.1

3 Verwandte Arbeiten

Dieses Kapitel beschreibt bekannte Ansätze der Graphersetzung. Hier sollen die verschiedenen Formalismen der einzelnen Ansätze beschrieben und verglichen werden. Ein Vergleich der Werkzeuge, welche die betrachteten Ansätze implementieren findet sich in Abschnitt 5.3.

3.1 PGRS/Progres

PGRS ist der Formalismus, der PROGRES [SWZ97] innewohnt. PROGRES, hauptsächlich entwickelt von ANDY SCHÜRR, ist wohl das mächtigste Werkzeug im Graphersetzungsbereich. Sein Hauptschwerpunkt ist die visuelle Spezifikation von großen Softwaresystemen und Datenbanken. PGRS verbindet die reine Graphersetzung mit einer Art Programmiersprache zur Steuerung der Ersetzung, die in Anlehnung an Datenbanksysteme transaktionsbasiert vonstatten geht.

3.1.1 Das Graphmodell

PGRS [Sch97] verwendet einen anderen Ansatz als die meisten GES. Graphen werden weder als relationale noch als algebraische Konstrukte, wie in Definition 2.1, sondern als logische Konstrukte betrachtet. Wir wollen hier kurz auf die Modellierung von Graphen in PGRS eingehen.

Ein $\Sigma = (\mathcal{A}_F, \mathcal{A}_P, \mathcal{V}, \mathcal{W}, \mathcal{X})$ heißt Signatur, wobei $\mathcal{A}_F$ Funktionssymbole, $\mathcal{A}_P$ Prädikatssymbole enthalten. $\mathcal{V}$ ist ein Alphabet von Objektbezeichnern, $\mathcal{W}$ ein Alphabet von Mengenbezeichnern, und $\mathcal{X}$ beinhaltet Variablen für prädikatenlogische Formeln. $\mathcal{A}_P$ enthält beispielsweise die Prädikate node(*Bezeichner*, *Typ*) und edge(*Quelle*, *Typ*, *Ziel*), die nötig sind, um einen Graphen zu modellieren.

$\mathcal{A}(\Sigma)$ ist die Menge der atomaren Formeln über Σ-Termen. Atomare Formeln beinhalten Prädikate und $=$, um die Gleichheit zweier Σ-Terme auszudrücken. $\mathcal{F}(\Sigma)$ ist die Menge aller geschlossenen Formeln der Prädikatenlogik erster Stufe die aus den atomaren Formeln $\mathcal{A}(\Sigma)$, den Junktoren $\wedge, \vee, \neg$ und den Quantoren $\forall, \exists$ besteht. Eine Teilmenge $F \subseteq \mathcal{A}(\Sigma)$ heißt Σ-Struktur (im Sinne von PGRS), wenn keine Formel in F die Gleichheit $=$ enthält. Betrachten wir folgende Σ-Struktur:

$$F = \{node(a, t_1), node(b, t_2), edge(a, t_3, b)\}\,; \quad a, b \in \mathcal{V},\ t_1, t_2, t_3 \in \mathcal{A}_F$$

F hat sicher viele Modelle (im logischen Sinn), sicherlich auch welche, die zusätzliche Ecken und Kanten beinhalten. Gemeint ist durch F aber ein Graph, der

zwei Ecken mit Typen t_1 und t_2 und eine Kante vom Typ t_3 beinhaltet. Um diese „unerwünschten" Modelle zu beseitigen, benötigt man einen sog. Vervollständigungsoperator $\mathcal{C}$. Eine Menge von Formeln Φ, die keine Elemente aus $\mathcal{V}$ und $\mathcal{W}$ enthält, kombiniert mit einen Vervollständigungsoperator $\mathcal{C}$, wird Σ-*Strukturschema* genannt. Ein Strukturschema drückt den *Typ* (wie in Abschnitt 4.2 definiert) eines Graphen mithilfe prädikatenlogischer Formeln aus. Graphen, die einem Σ-Strukturschema genügen, sind sog. *schemakonsistente Strukturen.* Vererbungshierarchien lassen sich so mit prädikatenlogischen Formeln beschreiben:

$$\forall x : node(x, sokrates) \implies node(x, mensch)$$

Es lassen sich zusätzlich Einschränkungen an einen Graphen formulieren

$$\forall x, y, z : edge(x, sohn, y) \wedge edge(z, sohn, y) \implies x = z$$

oder auch komplexere Prädikate definieren:

$$\begin{aligned}\forall x, y : brother(x, y) \iff \exists z :& edge(z, kind, x) \wedge edge(z, kind, y) \\ & \wedge node(y, mann) \wedge \neg(x = y)\end{aligned}$$

Hierdurch können zum Beispiel Pfadausdrücke für linke Seiten einer Regel implementiert werden. Kanten als eigene Objekte zu betrachten, sie insbesondere mit Attributen auszuzeichnen, ist mit PGRS im Gegensatz zum DPO- bzw. SPO-Ansatz nicht möglich.

3.1.2 Ersetzung und Matchen

Mithilfe der Definition der Σ-Strukturen lassen sich auch Teilstrukturen definieren, die Teilgraphen entsprechen. Die einer linken Seite entsprechenden Teilgraphen lassen sich aber nicht, wie bei der algebraischen Graphersetzung mittels Abbildungen (Graph-Homomorphismen) mit einer linken Seite in Beziehung setzen, da linke Seiten Mengenbezeichner enthalten können, denen mehrere Ecken in einem Muttergraphen entsprechen können. Deshalb wird der Zusammenhang zwischen linker Regelseite und Match in einem Muttergraphen mittels Relationen hergestellt. Interessanterweise ist die Frage, wie man zu einem Beweis kommt, der zeigt, dass alle Strukturen, die eine Regelmenge erzeugt, schemakonsistent sind, ungeklärt.

Der Matcher [Zün96] ähnelt dem OPTIMIX-Ansatz. Der Matcher besitzt zusätzlich einen Planer, der mittels einer Kostenfunktion durch das Strukturschema gegebene Einschränkungen versucht günstige Eckenbesuchs-Sequenzen zu finden.

3.1.3 Besonderheiten

PGRS besitzt durch Pfadausdrücke und die durch die Verwendung von Σ-Strukturen möglichen Einschränkungen an mögliche Graphen Fähigkeiten, die es von den anderen betrachteten Ansätzen abhebt. Die Verwendung von prädikatenlogischen Formeln zur Formalisierung beschränkt die Definition der Ersetzung nicht auf Graphen, sondern im allgemeinen auf Σ-Strukturen. Mit BCF

(Basic Control Flow) ist die Anwendungssteuerung Teil der Formalisierung des Graphersetzungs-Systems.

3.2 Optimix

Optimix [Ass95] [Ass00] entstand aus der Dissertation von Uwe Assmann am IPD Lehrstuhl Goos der Universität Karlsruhe. Optimix ist ein Generator für Optimierer in Übersetzern, die mit Graphersetzungssystemen spezifiziert werden. Der Schwerpunkt von Optimix liegt eher im Herstellen von Beziehungen zwischen einzelnen Elementen der Zwischendarstellung als in deren Umwandlung.

3.2.1 Das Graphmodell

Optimix verwendet im Gegensatz zu unserem Ansatz keine Multigraphen. Kanten werden in Optimix als Elemente einer Relation $E \times E$ betrachtet. Ecken und Kanten sind markiert und können mit Vererbung ausgestattet sein, jedoch wird dies sowie auch die Attributierung von Ecken und Kanten in [Ass00] nicht betrachtet. Im Gegensatz zu [Ass95] dürfen in [Ass00] Kanten eines Typs immer nur zwischen zwei Ecken mit festgelegten Typen existieren. Kanten der sog. Σ-Graphen sind eine Familie von zweistelligen Relationen

$$K = \{K_l\}_{l \in \Sigma_K}, K_l \subseteq E_{l_1} \times E_{l_2}, l_1, l_2 \in \Sigma_E$$

wobei Σ_E und Σ_K die möglichen Markierungen der Ecken bzw. Kanten enthalten. Desweiteren existieren $\Sigma^\neg$-Graphen, die zusätzlich eine Abbildung

$$negated : K \rightarrow \mathbb{B}$$

mitbringen. $\Sigma^\neg$-Graphen werden verwendet, um negative Kantenbedingungen auszudrücken (siehe Abschnitt 4.4.1) und dürfen folglich nur zur Formulierung von linken Seiten von Regeln verwendet werden.

3.2.2 Ersetzung und Matchen

Optimix hat einen eigens definierten Ersetzungsschritt, der dem SPO-Ansatz ähnelt. Baumelnde Kanten von zu löschenden Ecken werden ebenfalls gelöscht. Die Hauptunterschiede sind folgende:

1. Optimix verzichtet auf eine Formalisierung des Ersetzungsschrittes mittels Graph-Homomorphismen. Dies liegt daran, dass in [Ass95] (nicht jedoch in [Ass00]) zusätzlich zur Abbildung *negated* noch eine Abbildung $forall : E \rightarrow \mathbb{B}$ erwähnt ist. Diese Abbildung drückt aus, dass eine Ecke e einer linken Seite mit $forall(e) = true$ einer *Menge* von Ecken im Muttergraphen entspricht. Dies ist durch Graph-Homomorphismen nicht ausdrückbar.

2. Es werden keine nicht-injektiven Matches unterstützt, so dass das Problem, dass zwei Ecken aus einer linken Seite auf ein einzige Ecke im Muttergraphen abgebildet werden, nicht auftritt.

3. Eine Regel wird nur dann angewandt, wenn noch nicht alle Kanten, die sie hinzufügen würde, im Muttergraphen vorhanden sind.

OPTIMIX kann wie auch unser Ansatz Teilgraphen finden und ist nicht auf Untergraphen beschränkt. Der Matcher arbeitet ähnlich dem Matchen mittels relationaler Algebra; er führt im wesentlichen einen Verbund über alle sich im Muttergraphen befindlichen Ecken aus und testet, mittels vorher berechneter Pfade, ob die entsprechenden Kanten vorhanden und die Ecken gleich sind, wenn zwei (oder mehrere) Kanten sich treffen.

3.2.3 Besonderheiten

OPTIMIX legt besonderen Wert auf die Betrachtung der Regelmenge im Ganzen und beschränkt die Gestalt der Regelmenge derart, dass bestimmte Eigenschaften erfüllt werden. Von OPTIMIX werden momentan zwei Graphersetzungs-Systeme unterstützt:

XGRS (Exhaustive Graph Rewrite System)
: In jedem XGRS gibt es Teilmengen der $\Theta_E^+ \subset \Sigma_E$ und $\Theta_K^+ \subset \Sigma_K$, die Ecken- bzw. Kantenterminierungsmarkierungen genannt werden. Alle Regeln eines XGRS addieren (oder subtrahieren) jeweils eine Kante, die mit einem Element aus Θ_K^+ markiert ist, zu Ecken, die mit Elementen aus Θ_E^+ markiert sind. Diese Terminierungskanten (und Ecken) dürfen von keiner Regel des Systems gelöscht (oder hinzugefügt) werden. Somit ist die Terminierung des Systems sichergestellt.

EARS (Edge Addition Rewrite System)
: EARS sind eine Teilmenge der XGRS und werden aufgrund ihrer Verwandtschaft zu DATALOG [CGT89] gesondert behandelt. Eine Regel muss eine oder mehrere Kanten zum Graphen addieren, keine Ecke oder Kante löschen und keine Ecke hinzufügen. Da OPTIMIX nicht mit Multigraphen arbeitet, ist der Graph irgendwann voll besetzt und das Ersetzungssystem terminiert.

3.3 Agg

AGG [ERT97] stellt SPO-Graphersetzung für JAVA-Programme zur Verfügung. Ecken und Kanten können mit JAVA-Klassen attributiert werden. Ferner können durch JAVA-Ausdrücke Attribut-Tests und Attribut-Neuberechnungen implementiert werden.

AGG verfügt über sogenannte *Negative Application Conditions* (NACs). Mit NACs kann spezifiziert werden, welche Ecken und Kanten bei einem Match *nicht* vorgefunden werden dürfen. Dieser Ansatz geht über die von uns verwendeten negativen Kantenbedingungen hinaus, da er erlaubt, ganze Graphen als Negativbedingung zu verwenden. Das Graphmodell und der Ersetzungschritt entsprechen weitestgehend denen in Kapitel 4 vorgestellten Konzepten.

4 Unser Ansatz

Das Ziel dieser Arbeit ist die Entwicklung eines Graphersetzungswerkzeugs, das in einen Übersetzer eingebaut werden kann und von diesem aus steuerbar ist. Die Graphersetzung wird in diesem Kontext eher von der softwaretechnischen Seite betrachtet: Als Hilfsmittel, das dem Entwickler das Ausprogrammieren von Teilgraphtests und deren Ersetzungen abnimmt und ihm dafür eine Spezifikationssprache an die Hand gibt, in der diese leicht beschrieben werden können.

4.1 Programme als Graphen

Wie bereits in der Einleitung erwähnt, verfügen moderne Übersetzer oft über graphbasierte Zwischendarstellungen. Wir wollen hier die Zwischendarstellung Firm [TLB99] kurz vorstellen und auf die Besonderheiten in Bezug auf die durch die Struktur von Firm auftretenden Graphen eingehen.

Da wir vor allem an Graphersetzung im Kontext von Firm oder einer Firm-ähnlichen Zwischendarstellung interessiert sind, muss ein GES bestimmte Bedingungen erfüllen. Diese sollen hier genauer untersucht werden.

4.1.1 Die Zwischendarstellung Firm

Die Zwischendarstellung Firm wird am IPD Lehrstuhl Goos entwickelt und ist eine moderne SSA-basierte Zwischendarstellung, die das Eingabeprogramm als Steuerfluss- und Datenabhängigkeitsgraphen darstellt.

Firm repräsentiert Operationen (wie Addition, Laden aus dem Speicher oder auch einen Funktionsaufruf) als Ecken. Datenfluss wird über Kanten zwischen diesen Ecken dargestellt. Es existieren auch Ecken und Kanten, die den Steuerfluss zum Ausdruck bringen: So existiert pro Grundblock eine Ecke. Jede Datenflussecke ist 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.

Desweiteren werden alle Typen und deren Instanzen im Firm-Graphen dargestellt. Bestimmte Datenflussecken nehmen über spezielle Kanten Bezug auf die vorhandene Typinformation. Eine genauere Darstellung und Definition von Firm findet man in [TLB99]. Abbildung 4.1 zeigt den Firm-Graph zu folgendem C-Programm, das den ggT zweier Zahlen berechnet:

```
int gcd(int a, int b) {
  while(a != b) {
      if(a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
  return a;
}
```

Die Ecken mit Namen wie *Start*, *CmpT*, *PhiI*, ... stellen die Datenflussecken dar. Sie sind in größere Blöcke eingeschlossen (mit den Nummern 287, 285, 267, 271, 280, 269, 263, 265). Diese Blöcke stellen die Grundblöcke dar. Durch die *Cond*- und *Jmp*-Ecken wird der Steuerfluss verzweigt.

Hier sieht man auch die SSA-Eigenschaft von FIRM. Im Schleifenkopf (Block mit der Nummer 267) werden a und b verglichen, da dieser Block aber mehrere Steuerflussvorgänger hat (nämlich 285, 280 und 269; 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.

In FIRM stellt Datenfluss- und Steuerflusskanten aus historlischen Gründen als Datenabhängigkeits- bzw. Steuerabhängigkeitsgraphen dar. aus diesem Grund besitzen die Kanten in Abbildung 4.1 umgekehrte Richtung.

FIRM-Graphen haben einige Besonderheiten, die sie von allgemeinen Graphen unterscheiden. Zunächst sind Ecken und Kanten in FIRM-Graphen typisiert (bzw. markiert). Ferner besitzen Ecken Attribute, die das dargestellte Objekt genauer beschreiben. Beispielsweise besitzen Ecken, die Funktionstypen darstellen, bestimmte Attribute, die den Funktionstyp genauer beschreiben. Ferner 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.

Desweiteren sind folgende Eigenschaften interessant, obwohl sie sich nicht auf die Modellierung der Graphen auswirken. Als Daten- bzw. Steuerflussgraph hat ein FIRM-Graph immer zwei ausgezeichnete Ecken *Start* und *End* mit $|{}^\bullet Start| = 0$ und $|End^\bullet| = 0$. Desweiteren ist der Ausgangsgrad der Ecken bis auf wenige Ausnahmen für alle FIRM-Graphen bekannt und konstant. So hat zum Beispiel die Ecke, die eine Addition darstellt, immer zwei ausgehende Ecken, niemals eine oder drei. Allein die *Call*-, *Return*- und *BB*-Ecken besitzen einen variablen Ausgangsgrad.

4.2 Das Graphmodell

Obwohl das in dieser Diplomarbeit entwickelte Werkzeug nicht auf FIRM festgelegt ist, waren die Eigenschaften von FIRM, wie sie im vorigen Abschnitt

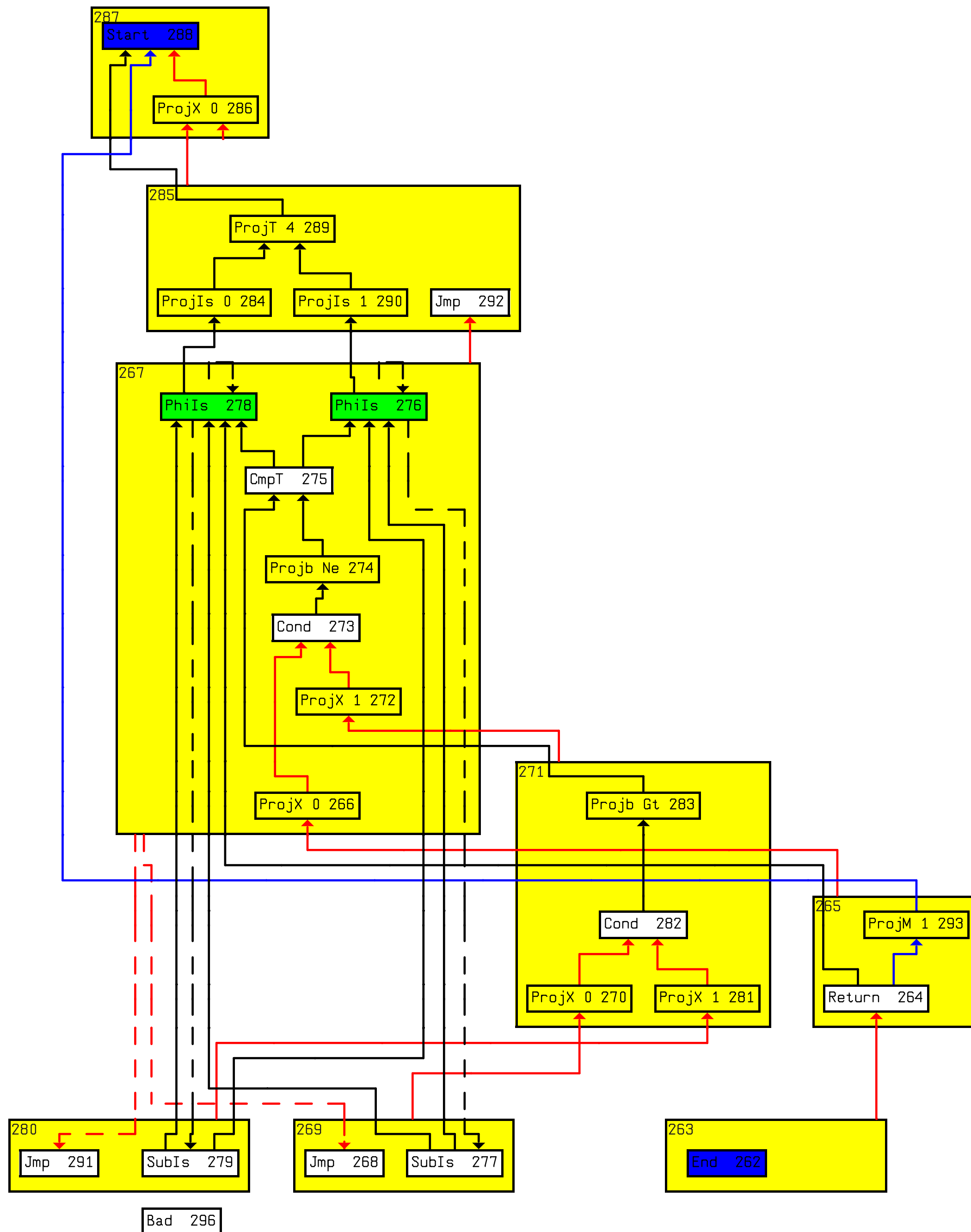


Abbildung 4.1: Das ggT-Programm in der Zwischendarstellung Firm

vorgestellt wurden, doch eine Maßgabe dessen, was unterstützt werden muss. Daher wurde in das Graphmodell alles integriert, was zur Graphersetzung auf FIRM-Graphen nötig ist[1].

Wir erweitern die Definition des Multigraphen aus Abschnitt 2.1 um Typen und Attribute für Ecken und Kanten. Hierzu fügen wir einem Multigraphen $G = (E, K, \mathrm{src}, \mathrm{tgt})$ zuerst die endlichen Mengen Σ_E und Σ_K hinzu, die die möglichen Typen der Ecken bzw. Kanten beinhalten. Ferner existieren zwei Abbildungen

$$\mathrm{typ}_E : E \to \Sigma_E, \quad \mathrm{typ}_K : K \to \Sigma_K$$

die den Ecken und Kanten eines Graphen einen Typ zuordnen. Um die Spezifikation von Regeln für den Anwender freundlicher zu gestalten, führen wir noch zwei Relationen $\leq_E$ und $\leq_K$ ein, die eine azyklische Halbordnung auf Σ_E und Σ_K herstellen[2]. $\leq_E$ und $\leq_K$ entsprechen der Halbordnung, die Vererbung in einer modernen objektorientierten Sprache induziert. $t_1 \leq_E t_2$; $t_1, t_2 \in \Sigma_E$ bedeutet, dass t_1 ein Obertyp von t_2 ist. Insbesondere sagt man: t_2 *ist* ein t_1. Wir erweitern Σ_E und Σ_K um die kleinsten Elemente $\bot_E$ und $\bot_K$, so dass gilt:

$$\forall t \in \Sigma_E : \bot_E \leq_E t \text{ und } \forall t \in \Sigma_K : \bot_K \leq_K t$$

Bis jetzt haben wir somit

$$G = (E, K, \mathrm{src}, \mathrm{tgt}, \Sigma_E, \Sigma_K, \mathrm{typ}_E, \mathrm{typ}_K, \bot_E, \bot_K, \leq_E, \leq_K)$$

Nun möchten wir die Ecken und Kanten noch mit Attributen versehen. Seien A_E und A_K zwei endliche und disjunkte Mengen. Jedem Attribut $a \in A_E \cup A_K$ ist eine Wertemenge D_a zugeordnet. Den Elementen aus Σ_E und Σ_K wird mit den Abbildungen

$$\mathrm{attr}_E : \Sigma_E \to \mathcal{P}(A_E), \quad \mathrm{attr}_K : \Sigma_K \to \mathcal{P}(A_K)$$

eine Menge von Attributen zugeordnet.

Da nun $t_1 \leq t_2$ bedeutet, dass t_2 ein t_1 *ist*, müssen wir zusätzlich fordern, dass sowohl für Ecken und Kanten (wir lassen die Indizierung weg, falls eine Aussage sowohl für E als auch für K zutrifft) gilt:

$$\forall t_1, t_2 \in \Sigma : t_1 \leq t_2 \implies \mathrm{attr}\, t_1 \subseteq \mathrm{attr}\, t_2$$

Ferner gilt:

$$\mathrm{attr}\, \bot = \emptyset$$

Dadurch sind Ecken (und Kanten) abhängig von ihrem Typ attributiert. Zuletzt sind noch zwei Abbildungen erforderlich, die eine Ecke (Kante) und ein Attribut ihres Typs mit einem Wert in Beziehung setzten:

$$\mathrm{val}_E : E \times A_E \to \bigcup_{a \in A_E} D_a, \quad \mathrm{val}_K : K \times A_K \to \bigcup_{a \in A_K} D_a$$

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

[2] $\leq_E$, $\leq_K$ sind reflexiv, transitiv und antisymmetrisch

Setzt man $T_X = (\Sigma_X, A_X, \mathrm{typ}_X, \mathrm{attr}_X, val_X, \leq_X, \perp_X)$, so erhalten wir für einen Graphen unseres Graphmodells

$$G = (E, K, \mathrm{src}, \mathrm{tgt}, T_E, T_K) \tag{4.1}$$

Wir nennen (T_E, T_K) den *Typen* des Graphen G.

Die Einführung der Vererbungsrelation macht eine Änderung der Definition des Graph-Homomorphismus für markierte Graphen (Definition 2.7) nötig, da wir beim Finden von Graphmustern von der Vererbungseigenschaft Gebrauch machen und uns nicht auf Typgleichheit beschränken wollen. Wir verstehen im Zusammenhang mit unserem Graphmodell unter einem Homomorphismus folgendes:

Definition 4.1 (Graph-Homomorphimus) Seien G und H zwei Graphen wie aus Gleichung (4.1) mit gleichem Typen (T_E, T_K). Eine Abbildung ϕ ist ein Graph-Homomorphismus von G nach H, wenn ϕ ein Graph-Homomorphismus im Sinne von Definition 2.5 ist und die Typen erhält:

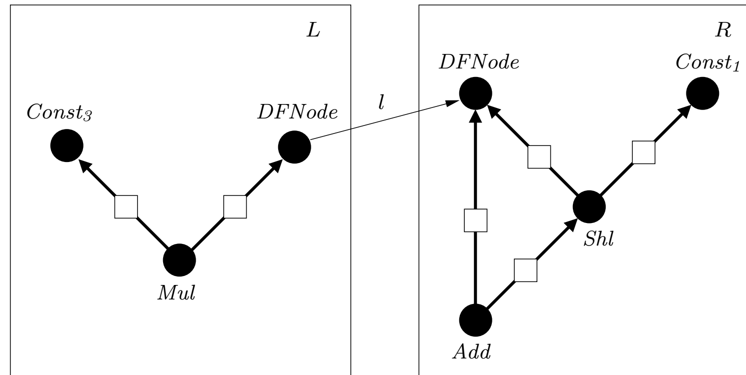
$$\forall e \in E_G : \mathrm{typ}_E(e) \leq_E \mathrm{typ}_E(\phi(e)), \quad \forall k \in K_G : \mathrm{typ}_K(k) \leq_K \mathrm{typ}_K(\phi(k))$$

4.3 Der Ersetzungsschritt

Entscheidend bei der Graphersetzung ist die Art und Weise, wie ein gefundener Teilgraph, der einer linken Seite einer Regel entspricht, durch die rechte Seite der Regel ersetzt wird. Wir haben uns aus mehreren Gründen für den SPO-Ansatz entschieden. Der SPO-Ansatz (wie auch der DPO-Ansatz) beschreibt *nur* den Ersetzungsschritt an sich, er stellt keine Bedingungen an den Matcher, denn er sieht die Existenz des Morphismus m, der die linke Seite einer Regel auf den gefundenen Teilgraphen eines Graphen G abbildet, als gegeben voraus. Dies erlaubt ein flexibles Austauschen der Matcher und läßt an dieser Stelle Raum für Optimierungen, die wohl auch nötig sind, wie der nächste Abschnitt zeigt.

Essenziell für die Ersetzung ist eine klare Formalisierung, da dieser Teil derjenige ist, der den Graphen letztlich manipuliert. (Die Theorie der formalen Sprachen gibt auch keine Auskunft darüber, wie man eine rechte Seite in einem gegebenen Wort auffindet. Sie beschreibt lediglich den Vorgang des Ersetzens bei *gefundener* rechter Seite.)

Der DPO-Ansatz (siehe Abschnitt 2.3.2) ist zu restriktiv, da die Regelanwendung bei baumelnden Kanten verhindert wird. Diese Situation ist aber gerade bei der Manipulation von Datenflussgraphen allgegenwärtig, da Datenflussecken in einem Datenflussgraphen zwar eine konstante Anzahl von ausgehenden, aber eine unbekannte Anzahl von eingehenden Kanten haben. Diese können unmöglich in jeder Regel mitspezifiziert werden, wie folgendes Beispiel zeigt, das eine Multiplikation mit der Konstanten 3 durch eine Addition und einen Bitshift ersetzt. Die eingehenden Kanten der *Mul*-Ecke können in diesen Mustern nicht mitspezifiziert werden, da schlicht nicht bekannt ist, wieviele andere Ecken mit dieser Ecke verbunden sind.



Die Markierung der Kanten wurde der Übersicht wegen weggelassen. Die Markierungen an den Ecken stellen Typen (also Elemente aus Σ_E) der Ecken dar. Eine Indizierung wie $Const_n$ soll bedeuten, dass ein Attribut dieser Ecke (in diesem Fall das Attribut, das den Wert der Konstante repräsentiert) den Wert n hat.

Beim SPO-Ansatz wird die Regelanwendung im eben skizzierten Fall nicht verhindert, jedoch werden alle baumelnden Kanten gelöscht. Die *Mul*-Ecke wird gelöscht, somit werden auch alle eingehenden Kanten gelöscht. Die Intention war aber, die *Mul*-Ecke durch die *Add*-Ecke zu ersetzen und alle eingehenden Kanten beizubehalten. Dazu müßte der partielle Homomorphismus $l : L \to R$ die *Mul*- auf die *Add*-Ecke abbilden. Dies ist jedoch nicht möglich, da l dann die Homomorphismus-Eigenschaft für markierte Graphen verletzten würde.

Das Graphmodell muss also, um Graphersetzung mit dem SPO-Ansatz zu ermöglichen, umformuliert werden. Um obiges Problem zu lösen, müssen die Markierungen von den Ecken entfernt werden, da nur so eine Homomorphie zwischen zwei Ecken hergestellt werden kann. Um dennoch mit typisierten Ecken arbeiten zu können, postuliert man, dass jede Ecke ein Schlinge besitzt, die die Typinformation der Ecke trägt. Diese Schlinge kann dann bei einem Ersetzungsschritt gelöscht und durch eine andere ersetzt werden, ohne dass die eingehenden Kanten zu einer Ecke zerstört werden. Dieses Vorgehen wurde auch bei der Implementierung des AGG-Interpreters eingesetzt [Rud97]. Diese Umformulierung ist „nach aussen hin“ völlig transparent und nur zur Formalisierung des Ersetzungsschrittes notwendig. Es ergeben sich aus ihr keinerlei Einschränkungen, so dass unser Graphmodell, wie im letzten Abschnitt besprochen, davon nicht tangiert wird.

4.4 Das Finden von Graphmustern

Das Finden eines *Teilgraphen*[3] L eines Graphen G entspricht der Bestimmung des Morphismus m, der bei der Anwendung einer SPO-Regel als gegeben angenommen wird. Tatsächlich ist die Bestimmung von m jedoch die komplexitätstheoretisch aufwendigste Operation eines Graphersetzungsschrittes, da

[3] Man beachte den feinen Unterschied zwischen Teil- und Untergraph

das Finden von homomorphen Teilgraphen NP-vollständig ist. Schon das Problem, ob L *isomorph* zu einem Teilgraphen von G ist, ist NP-vollständig (siehe auch [GJ79]).

> Sei $n \in \mathbb{N}$ beliebig aber fest und $n > 0$. Sei G ein unmarkierter Multigraph mit $|E_G| = n$. Sei nun L ein unmarkierter Multigraph mit $E_L = \{e_1, \ldots, e_n\}$ und $K_L = \{k_1, \ldots, k_n\}$. Desweiteren soll gelten:
>
> $$\operatorname{src} k_i = e_i,\ \operatorname{tgt} k_j = e_{j+1},\ \operatorname{tgt} k_n = e_1 \quad ; 1 \leq i \leq n,\ 1 \leq j < n$$
>
> L stellt also einen Kreis dar. Das Finden eines Teilgraphen von G, der zu L isomorph ist entspricht somit dem Finden eines Hamiltonschen Pfads. Dieses Problem ist aber NP-vollständig. Das Problem des homomorphen Teilgraphen ist eine Verallgemeinerung vieler NP-vollständiger Probleme aus der Graphentheorie.

Insofern ist der Entwurf eines Matchers, der mit geeigneten Heuristiken und Einschränkungen an das Graphmodell versucht, der dem Problem innewohnenden Komplexität Herr zu werden, eine umfangreiche Aufgabe. Viele vorhandenen Ansätze schränken deswegen die Klasse der möglichen m auf vielfältige Weise ein:

1. Es werden nur isomorphe m zugelassen. Der Matcher muss also niemals die Identifikation zweier Ecken berücksichtigen.
2. Man beschränkt sich auf Untergraphen anstelle von Teilgraphen. Dieses hat zum Vorteil, dass das Vorhandensein einer Kante in G, die keine Entsprechung in L hat, schon zum Scheitern der Suche führt.
3. Der Benutzer muss bestimmte Ecken oder Kanten als Parameter an den Matcher übergeben. Somit ist der Match in G lokal „begrenzt“.
4. Ecken können mit Einschränkungen an die ein- bzw. ausgehenden Kanten versehen werden. So kann der Benutzer in PROGRES spezifizieren, dass alle Ecken vom Typ A genau eine ausgehende Kante des Typs B besitzen. Dies hilft den Suchraum beim Matchen einzuschränken.

Gerade der erste Punkt stellt aber im Hinblick auf die Bearbeitung von Datenabhängigkeitsgraphen mit Graphersetzung eine unerwünschte Einschränkung dar, da die beiden Operanden eines binären Operators durchaus durch dieselbe Ecke gegeben sein könnten. Müßte man mit ausschließlich injektiven Matches leben, so müßte man die Regelmenge erheblich vergrößern, da jeder Fall (beide Operanden sind gleich oder ungleich) einzeln betrachtet werden müßte.

Die Charakterisierung der ausgehenden Kanten einer Ecke wird von PROGRES verwendet, um das Matchen zu beschleunigen. So werden Kanten, die nur einmal an einer Ecke vorkommen, von einem Planer, der die Suche plant, bevorzugt, da ein nicht Vorhandensein einer solchen Kante sofort zum Abbruch der Suche führt. Sicherlich ändert sich an der theoretischen Komplexität nichts, jedoch besteht die Hoffnung, durch diese und andere Heuristiken die praktisch relevanten Fälle zu beschleunigen [Zün96].

RUDOLF [Rud00] reduziert das Finden von Graphmustern auf das *Constraint Satisfaction Problem* (CSP) [DvB97] um von existierenden Algorithmen

zu profitieren. Leider sind keine Messungen über die Leistungsfähigkeit der Methode bekannt.

DÖRR entwickelte in seiner Dissertation [Dör95] eine Methode zum Finden von Graphmustern in linearer Zeit, indem eine Ecke des Musters im Muttergraphen vorgegeben wird. Er erreicht dies im wesentlichen dadurch, dass sog. *starke V-Strukturen* beim Matchen nach hinten verlagert werden. Starke V-Strukturen entstehen dadurch, dass eine Ecke mit zwei Ecken desselben Typs über Kanten mit gleichem Typ verbunden sind.

Wir entschieden uns aus mehreren Gründen für einen anderen Ansatz: Dem Finden von Graphmustern mit relationaler Algebra. Folgende Punkte standen bei dieser Entscheidung im Vordergrund:

Unabhängigkeit
: Der Matcher sollte formal nicht vom Ersetzungsschritt abhängig sein um ihn austauschbar zu halten.

Einfachheit
: Das Erstellen von Formeln der relationalen Algebra (im konkreten Fall SQL-Anweisungen) aus Graphmustern ist leicht nachzuvollziehen und somit weniger fehleranfällig.

Wiederverwendung
: Anstelle eigener aufwendiger Optimierungen wurde das Suchen der Graphmuster in die Hände eines Datenbanksystems gegeben, das mithilfe seines Anfragenoptimierers das Auffinden beschleunigen kann.

Verfügbarkeit
: Das System sollte innerhalb kurzer Zeit einsatzbereit sein, um den Einsatz von Graphersetzung im Übersetzerbau besser erforschen zu können.

Der folgende Abschnitt erläutert das Verfahren im Detail.

4.4.1 Das Finden von Graphmustern mit relationaler Algebra

Bei der Implementierung unseres Ansatzes wurde die Repräsentation der Graphen und das Auffinden von Teilgraphen zunächst mit einem Datenbanksystem umgesetzt. In diesem Abschnitt wollen wir eine Formulierung des Auffindens von Teilgraphen mittels relationaler Algebra vorstellen. Die nötige Terminologie wird in Anhang B eingeführt. Wir wollen relationale Schemata in diesem Abschnitt mit kalligraphischen und Relationen mit kleinen griechischen Buchstaben bezeichnen, um nicht mit den Bezeichnern von Mengen sowie den Ecken und Kanten eines Graphen zu kollidieren.

Für das Ablegen von Graphen in einer Datenbank sind zunächst zwei rela-

tionale Schemata nötig[4]:

$$\begin{array}{rcl}\mathcal{E} & = & \{node_id, node_type_id\} \\ \mathcal{K} & = & \{edge_id, src_id, tgt_id, edge_type_id\}\end{array}$$

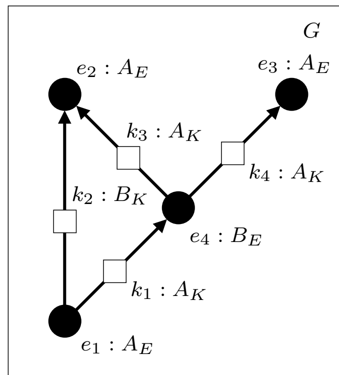
Alle Attribute in $\mathcal{E}$ und $\mathcal{K}$ haben die natürlichen Zahlen als Wertebereich. Im folgenden bezeichnen wir die Relation, die alle Ecken eines Graphen G beinhaltet, mit η_G und die Relation, die die Kanten des Graphen beinhaltet mit κ_G. Ist aus dem Kontext klar, um welchen Graphen es sich handelt, lassen wir das G im Index weg. Ferner müssen die Attribute *node_id* und *edge_id* in η_G, bzw. κ_G eindeutig sein. Es gilt $\mathcal{S}(\eta_G) = \mathcal{E}, \mathcal{S}(\kappa_G) = \mathcal{K}$. Desweiteren existieren zwei Abbildungen

$$\#_E : \Sigma_E \to \mathbb{N}, \quad \#_K : \Sigma_K \to \mathbb{N}$$

die den Typen der Ecken und Kanten eineindeutig Typnummern zuordnen. Da die Typmengen sowohl für Ecken und Kanten endlich sind, existieren solche Abbildungen beispielsweise durch Aufzählen. Desweiteren benötigen wir zwei Relationen $\sqsubseteq_E$, $\sqsubseteq_K \subset \mathbb{N} \times \mathbb{N}$, welche die Typkompatibilität auf den Typnummern nachbilden (wir lassen das E bzw. K im Index der Übersichtlichkeit halber weg):

$$a \sqsubseteq b \iff \#^{-1}(a) \leq \#^{-1}(b)$$

Betrachten wir beispielsweise folgenden Graphen G:



Die Ecken und Kanten sind mit Namen versehen, um im Text darauf Bezug nehmen zu können. $e_1 : A$ bedeutet, dass e_1 eine Ecke des Typs A ist. Die Darstellung von G durch zwei Relationen η_G, κ_G sieht wie folgt aus:

η_G:

node_id	*node_type_id*
1	1
2	1
3	1
4	2

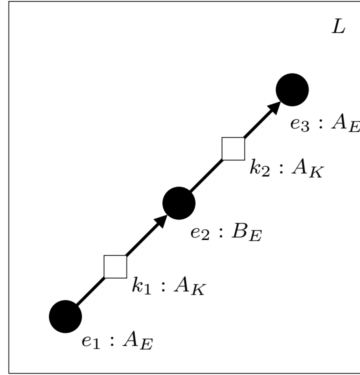
κ_G:

edge_id	*src_id*	*tgt_id*	*edge_type_id*
1	1	4	1
2	1	2	2
3	4	2	1
4	4	3	1

[4] Wir verzichten hier der Einfachheit halber auf die Betrachtung von Attributen für Ecken und Kanten. Deren Implementierung ist durch die Hinzunahmen zweier weiterer relationaler Schemata gewährleistet.

Das Suchen eines Teilgraphen L in G läßt sich durch Ausdrücke der relationalen Algebra formulieren. Sie stellen, wie in Anhang B besprochen, Abbildungen von Datenbanken auf Datenbanken dar. Somit ist der evtl. gefundene Teilgraph wieder als Relation gegeben. Wir müssen nun den zu findenden Teilgraphen L so beschreiben, dass sich aus der Beschreibung eine Formel der relationalen Algebra ableiten läßt, die die Datenbank D_G, die G beschreibt, auf eine Datenbank D_L abbildet, die einen Teilgraphen in G beschreibt, der zu L isomorph ist. Wir wollen hier zuerst das Finden von isomorphen Matches beschreiben. Die Erweiterung auf nichtisomorphe Matches folgt später.

Beginnen wir wieder mit einem Beispiel. Sei $L = (E_L, K_L, \mathrm{src}_L, \mathrm{tgt}_L, T_E, T_K)$ mit $E_L = \{e_1, e_2, e_3\}$, $K_L = \{k_1, k_2\}$, $T_E = \{A_E, B_E\}$ und $T_K = \{A_K, B_K\}$ gegeben durch:



Man kann nun für von L eine prädikatenlogische Formel erstellen, die alle Teilgraphen eines Graphen G, die zu L isomorph sind, beschreibt. [5] Im Falle von L ist diese Formel gegeben durch:

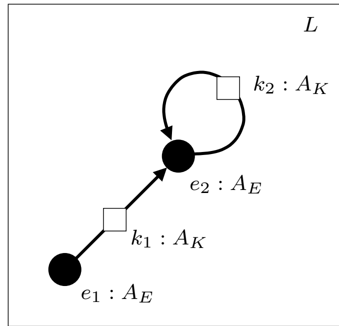
$$\begin{aligned}
F_L &= \exists e_1, e_2, e_3 \in E_G : \exists k_1, k_2 \in K_G : p_E(e_1, e_2, e_3) \wedge p_K(k_1, k_2) \\
p_E(e_1, e_2, e_3) &= \neg(e_1 = e_2) \wedge \neg(e_1 = e_3) \wedge \neg(e_2 = e_3) \\
&\quad \wedge(\mathrm{typ}_E(e_1) \leq_E A_E) \wedge (\mathrm{typ}_E(e_2) \leq_E B_E) \wedge (\mathrm{typ}(e_3) \leq_E A_E) \\
p_K(k_1, k_2) &= (\mathrm{src}_G\, k_1 = e_1) \wedge (\mathrm{tgt}_G\, k_1 = e_2) \wedge (\mathrm{src}_G\, k_2 = e_2) \wedge (\mathrm{tgt}_G\, k_2 = e_3) \\
&\quad \wedge(\mathrm{typ}_K(k_1) \leq_K A_K) \wedge (\mathrm{typ}_K(k_2) \leq_K A_K)
\end{aligned}$$

Die Formel F_L beschreibt drei paarweise disjunkte Ecken $e_1, e_2, e_3 \in G_E$ (Prädikat p_E), die durch zwei Kanten k_1, k_2 verbunden sind, wobei k_1 e_1 mit e_2 verbindet und k_2 e_2 mit e_3 (Prädikat p_K). Gibt es eine Belegung für $e_1, e_2, e_3 \in E_G, k_{1,2} \in K_G$, so dass F_L wahr wird, besitzt G einen zu L isomorphen Teilgraphen. Man sieht leicht, dass durch das Weglassen von Termen der Form $\neg(e_2 = e_3)$ in p_E auch Teilgraphen von G beschrieben werden können, die zu L nicht mehr isomorph, aber dennoch homomorph sind. Ersetzt man p_E in F_L zum Beispiel durch

$$p'_E(e_1, e_2, e_3) = \neg(e_1 = e_2) \wedge \neg(e_1 = e_3)$$

[5] Wir betrachten zuerst prädikatenlogische Formeln, da sie die wesentlichen Bestandteile zur Konstruktion von Formeln der relationalen Algebra enthalten und somit das grundlegende Konzept in bekanntem Formalismus präsentieren.

so wäre auch der folgender Teilgraph mit der modifizierten Formel beschrieben:



Diese Beschreibung von Graphen mit prädikatenlogischen Formeln läßt sich nutzen, um jene Abbildung zu formulieren, die die Datenbank D_G auf D_L abbildet. Diese Abbildung sollte eine Relation μ liefern, deren Schema $\mathcal{M}$ für jede Ecke und Kante in L ein entsprechendes Attribut besitzt. Somit kann man das Ergebnis leicht weiterverarbeiten, da jedes Tupel einem gefundenen Teilgraphen entspricht. Ist die Relation μ leer, so enthält der Graph G keinen Teilgraphen, der zu L isomorph ist. Betrachten wir zunächst das Schema $\mathcal{M}$ von μ. Es enthält für jede Ecke und Kante von L je ein Attribut:

$$\mathcal{M} = \left\{ node_id_1, \ldots, node_id_{|E|}, edge_id_1, \ldots, edge_id_{|K|} \right\}$$

Die Relation μ besteht somit aus Verbunden über der Eckenrelation η_G und der Kantenrelation κ_G. Die Bedingungen der Verbundoperatoren müssen die paarweise Disjunktivität der Ecken und die Inzidenzsituation der Kanten ausdrücken. Zunächst wollen wir noch eine Bezeichnung einführen, um die folgenden Ausdrücke lesbarer zu gestalten: Sei ρ eine Relation. $\rho_{i\leftarrow}$ ist nun die Relation, die aus ρ hervorgeht, indem jedes Attribut $a \in \mathcal{S}(\rho)$ in a_i umbenannt wird.

Die Relation μ, die die gefundenen Teilgraphen von G darstellt, läßt sich wie folgt konstruieren. Für jede Ecke $e_i \in E_L$ benötigt man $\gamma(e_i)$ Verbundoperationen um die Inzidenzsituation an e_i auszudrücken. Die Bedingungen für die Verbundoperationen entsprechen dem Prädikat p_K des letzten Absatzes. Die dadurch für jede Ecke e_i entstehende Relation soll hier mit η_i bezeichnet werden. μ ergibt sich dann aus einem Verbund aller η_i, indem alle e_i als paarweise disjunkt markiert werden. Zunächst betrachten wir die Konstruktion der η_i:

Sei e_i eine Ecke aus L. Desweiteren sei $n = |{}^\bullet e_i|$ und $m = |e_i{}^\bullet|$. P und Q stellen die Indexmengen der zu e_i eingehenden bzw. ausgehenden Kanten dar:

$$\begin{aligned} P &= \{p_1, \ldots, p_n\} = \{j \mid k_j \in {}^\bullet e_i\} \subset \mathbb{N} \\ Q &= \{q_1, \ldots, q_m\} = \{j \mid k_j \in e_i{}^\bullet\} \subset \mathbb{N} \end{aligned}$$

η_i setzt sich aus drei Komponenten zusammen: Der Eckenrelation an sich und jeweils einem Verbund über die eingehenden und die ausgehenden Kanten von e_i:

$$\eta_i = \eta_{i\leftarrow} \bowtie_A \kappa_{in} \bowtie_B \kappa_{out}$$

Wobei κ_{in} und κ_{out} gegeben sind durch:

$$\begin{aligned} \kappa_{in} &= \kappa_{p_1 \leftarrow} \bowtie_{\Theta_1} \cdots \bowtie_{\Theta_{n-1}} \kappa_{p_n \leftarrow} \\ \kappa_{out} &= \kappa_{q_1 \leftarrow} \bowtie_{I_1} \cdots \bowtie_{I_{m-1}} \kappa_{q_m \leftarrow} \end{aligned}$$

Die Prädikate Θ_j und I_j stellen die Gleichheit der Ziele bzw. Quellen der Kanten her. A verbindet die Eckenrelation mit der ersten Kantenrelation κ_{in} und B verbindet die beiden Kantenrelationen untereinander. Anschaulich gesprochen entsprechen A und B dem $\wedge$-Operator in F_L. Θ und I entsprechen den $\wedge$-Operatoren in p_K. A, B, Θ_j, I_j sind gegeben durch:

$$\begin{aligned} A &: (node_id_i = tgt_id_{p_1}) \wedge (node_type_id_i \sqsubseteq_E \#_E(e_i)) \\ B &: (tgt_id_{p_n} = src_id_{q_1}) \\ &\quad \wedge (edge_type_id_{p_n} \sqsubseteq_K \#_K(k_{p_n})) \\ &\quad \wedge (edge_type_id_{q_1} \sqsubseteq_K \#_K(k_{q_1})) \\ \Theta_j &: (tgt_id_{p_j} = tgt_id_{p_{j+1}}) \wedge (edge_type_id_{p_j} \sqsubseteq_K \#_K(k_{p_j})) \\ I_j &: (src_id_{q_j} = src_id_{q_{j+1}}) \wedge (edge_type_id_{q_{j+1}} \sqsubseteq_K \#_K(k_{q_{j+1}})) \end{aligned}$$

Verbindet man nun alle η_i, $1 \leq i \leq |E|$, so erhält man die Relation μ^*, die fast der gewünschten Relation μ entspricht. In μ soll für jede Kante in L genau ein *edge_id* existieren. Aufgrund der obigen Konstruktion sind bei einem Verbund aller η_i für jede Kante genau zwei Attribute vorhanden. Denn eine Kante ist immer Bestandteil einer κ_{in} als auch einer κ_{out}-Relation. Somit ergibt sich μ aus μ^*, indem mittels Projektion die doppelt vorhandenen Kanten verworfen werden.

$$\mu^* = \eta_1 \bowtie_{\Lambda_1} \cdots \bowtie_{\Lambda_{|E|-1}} \eta_{|E|}$$

mit

$$\Lambda_j : \neg(node_id_j = node_id_{j+1}) \wedge \cdots \wedge \neg(node_id_j = node_id_{|E|})$$

Wünscht man nicht isomorphe Matches, wie oben bereits angedeutet, so äußert sich das darin, dass in den Λ_i Terme fehlen. Im Extremfall sind die Λ_i leer und die Verbunde zwischen den η_i degenerieren zu kartesischen Produkten. Somit ergeben sich $|E| + |K| - 1$ Verbundoperationen über $|E| + |K|$ Relationen.

Erweiterungen

Wie bereits erwähnt, kann man mit obigem Verfahren die Injektivität des Morphismus m für jedes Eckenpaar gezielt verwerfen, indem man die entsprechenden Terme in den Λ_i wegläßt. Somit wird m zum Epimorphismus. Diese Eigenschaft unter Umständen kann von Nutzen sein, um die Regelmenge klein zu halten und somit eine einfachere Spezifikation der Graphersetzungsregeln zu ermöglichen.

Eine zweite wünschenswerte Eigenschaft an das Matchen ist die Möglichkeit *negative Kantenbedingungen* zu spezifizieren. Man möchte in bestimmten Situationen zum Ausdruck bringen können, dass zwei Ecken *nicht* durch eine Kante des Typs X verbunden ein dürfen. Diese Fähigkeit besitzen auch OPTIMIX und PROGRES.

In einer prädikatenlogischen Formel wie F_L, die die Existenz eines zu L isomorphen (oder auch nur homomorphen) Teilgraphen ausdrückt, müssen nun weitere Formeln vermöge eines Prädikats p_N eingeführt werden. Möchte man beispielsweise spezifizieren, dass die Ecke e_1 nicht durch eine Kante des Typs B_K mit einer anderen Ecke verbunden sein darf, so lautet das Prädikat $p_N(e_1)$:

$$p_N(e_1) = \neg\exists k \in E_K : \mathrm{inc}(e_1, k) \wedge (\mathrm{typ}_K(k) \leq_K B_K)$$

In der relationalen Algebra kann man hierzu die *linke äußere Verbindung*[6] verwenden. Die linke äußere Verbindung $\rho_1 \,{}^{\circ}\!\bowtie_\Theta \rho_1$ nimmt zusätzlich zu den Tupeln aus $\rho_1 \bowtie_\Theta \rho_2$ noch alle Tupel aus ρ_1 auf, die nicht in $\rho_1 \bowtie_\Theta \rho_2$ eingegangen sind, und ergänzt die fehlenden Werte der rechten Seite zu $\perp$.

Bildet man den Ausdruck

$$\pi_{\{node_id\}}(\eta \,{}^{\circ}\!\bowtie_A \kappa)$$

mit

$$A : (node_id = a) \wedge (edge_type_id \sqsubseteq_K \#_K(X_K)), \quad a \in \{tgt_it, src_id\}$$

so enthält die resultierende Relation genau dann $\perp$, wenn es keine eingehende bzw. ausgehende Kante eines Typs X_K zu der spezifizierten Ecke gibt.

Implementierung mit SQL

Die praktische Umsetzung des im letzten Abschnitt besprochenen Matchingansatzes implementiert das Werkzeug GrGen mittels SQL. In der jetzigen Version werden die Verbundoperationen nicht explizit ausgegeben, sondern mittels `SELECT ... FROM ... WHERE ...` umgesetzt. Dieses Vorgehen läßt dem Anfragenoptimierer genügend Freiraum um die Anfrage zu optimieren. Es ist jedoch zu untersuchen, ob man durch eine festgelegte Vorgabe der Verbundreihenfolge eine bessere Leistung erzielen kann.

Zur Implementierung der Vererbungsrelation $\sqsubseteq$ auf Ecken- und Kantentypen werden die (C-) Funktionen `node_type_is_a` und `edge_type_is_a` aus der Spezifikation erzeugt und zu einer Bibliothek weiterverarbeitet, die der SQL-Server zur Laufzeit dynamisch laden kann. Diese Fähigkeit besitzen die meisten verfügbaren SQL-Server. Aus dem obigen Beispielgraphen L wäre zum Beispiel folgender SQL-Ausdruck erzeugt worden:

```
SELECT n1.node_id, n2.node_id, n3.node_id, e1.edge_id, e2.edge_id
FROM nodes AS n1, nodes AS n2, nodes AS n3, edges AS e1, edges AS e2
WHERE node_type_is_a(n1.type_id, 1)
AND n1.node_id <> n3.node_id
AND n1.node_id <> n2.node_id
AND n1.node_id = e1.src_id
AND node_type_is_a(n2.type_id, 2)
AND n2.node_id <> n3.node_id
AND n2.node_id = e2.src_id
AND e2.src_id = e1.tgt_id
```

[6] engl.: Left Outer Join

```
AND node_type_is_a(n3.type_id, 1)
AND n3.node_id = e2.tgt_id
AND edge_type_is_a(e1.type_id, 1)
AND edge_type_is_a(e2.type_id, 1)
```

4.5 Übersicht und Vergleich

Das in Abschnitt 4.2 vorgestellte Graphmodell bietet alle nötigen Eigenschaften, um Graphen aus Übersetzerzwischensprachen zu repräsentieren. Die wichtige Ordnung der Kanten in Datenabhängigkeitsgraphen kann durch Kantenattribute implementiert werden. In OPTIMIX' Graphmodell sind Kanten implizit geordnet. In PGRS kann diese Ordnung nur durch Einführung von verschiedenen Kantentypen erreicht werden. So muss für jede Position (erstes Argument einer Operation, zweites Argument, etc.) ein eigener Kantentyp vorhanden sein. Ferner erlaubt PGRS wie auch OPTIMIX keine Multigraphen, was bei Datenabhängigkeitsgraphen eine Einschränkung darstellt. Vererbung für Ecken bieten alle betrachteten Ansätze. Für die Zusatzfähigkeiten von PGRS wie Basic Control Flow und Pfadausdrücke sehen wir momentan keinen Bedarf, zumal sie nur durch eine erheblich aufwendigere Implementierung zu erhalten sind. Desweiteren bietet der SPO-Ansatz mit Graphen und Graph-Homomorphismen im Vergleich zu PGRS' Σ-Strukturen eine für unsere Belange elegantere Formalisierung. OPTIMIX ist aufgrund der Beschränkung auf EARS und XGRS für die Ersetzungsregeln im Kontext der Erzeugung von Multimediabefehlen nicht brauchbar, da hier sehr wohl Ecken und Kanten beliebig gelöscht und hinzugefügt werden müssen.

Wir fassen hier die Hauptmerkmale der betrachteten Graphersetzungsansätze zusammen. AGG ist aufgrund der Ähnlichkeit zu unserem Ansatz hinsichtlich des formalen Modells nicht explizit aufgeführt.

Eigenschaft	PGRS	OPTIMIX	Unser Ansatz
Vererbung Ecken	✓	?	✓
Vererbung Kanten	–	–	✓
Attribute Ecken	✓	✓	✓
Attribute Kanten	–	–	✓
Geordnete Kanten	–	✓	–
Pfadasudrücke	✓	–	–
Abgeleitete Attribute	✓	–	–
Matching (Teilgraph/ Untergraph)	Teilgraph	Teilgraph	Teilgraph
Matching Morphismus (isomorph/homomorph)	homomorph	isomorph	homomorph
Behandlung baumelnder Kanten	Löschen	Löschen	Löschen
Multigraphen	–	–	✓

Ein Vergleich der Implementierungen findet sich in Abschnitt 5.3.

5 GRGEN

Im Rahmen dieser Arbeit wurde das Werkzeug GrGen entwickelt, das die Integration von Graphersetzungsmechanismen in andere Programme (insbesondere Übersetzer) ermöglicht. Bei der Entwicklung war das Hauptaugenmerk auf eine möglichst modulare Bauweise gelegt, so dass das Werkzeug auf die Bedürfnisse des Einsatzszenariums zugeschnitten werden kann. Die GrGen-Umgebung besteht aus zwei Teilen:

LibGr *(Library for Graph Replacement)*
: ist eine C-Bibliothek, die einen einheitlichen Zugriff auf Graphen, die dem im letzten Kapitel beschriebenen Graphmodell entsprechen, sowie auf Ersetzungsregeln bietet. Die Art der Speicherung der Graphen und Umsetzung der Graphersetzung wird vor dem Programmierer verborgen, so dass das Programm, das mithilfe der LibGr Graphersetzung einsetzt, von der Implementierung der Graphmodelle und Ersetzungsregeln völlig unabhängig ist.

GrGen *(Graph Replacement Generator)*
: ist ein Werkzeug, das aus einer einfachen Sprache zur Spezifikation von Graphmodellen und Ersetzungsregeln Graphersetzer erzeugt. Diese können dann insbesondere durch die LibGr verwendet werden.

5.1 Architektur der LibGr

Die LibGr gliedert sich in die drei Hauptkomponenten:

Graph
: stellt Methoden zur Manipulation eines Graphen auf unterster Ebene zur Verfügung: Hinzufügen und Entfernen von Ecken und Kanten. Lesen und Schreiben der Typen und Attribute von Ecken und Kanten.

: Desweiteren kann der Typ des Graphen (siehe Abschnitt 4.2) inspiziert werden: Welche Ecken- bzw. Kantentypen sind vorhanden? Ist A ein Untertyp von B? Welche Attribute hat ein Typ?

Actions
: ist die Schnittstelle zum Ausführen von Ersetzungsregeln auf einem Graphen. Mustersuche und Ersetzungsschritt sind einzeln ausführbar, so dass die Anwendung entscheiden kann, welches gefundene Muster ersetzt werden soll.

Backend
: gewährleistet die Initialisierung und Verwaltung der von *Graph* und *Actions* benötigten Infrastruktur.

Zu jeder dieser drei Komponenten existiert ein sogenanntes *Model*. Diese *Models* implementieren die jeweilige Schnittstelle und können dynamisch geladen werden. Dabei hängt ein *GraphModel* von einem *BackendModel* ab, da es auf die von diesem verwalteten Betriebsmittel zugreift. Das *ActionsModel* hängt von einem *GraphModel* ab, da es die deklarierten Typen und Attribute der Ecken und Kanten und eventuell auf die interne Repräsentation des Graphen zugreifen muss. Letztlich besteht die LibGr nur aus Umwicklermethoden; die eigentliche „Arbeit“ erledigen die *Models*. Eine bequeme Möglichkeit, *Models* zu testen bietet die in der LibGr enthaltene GrShell, mit der man interaktiv Graphen aufbauen, manipulieren und die in einem *ActionModel* implementierten Graphersetzungen ausführen kann.

5.2 GrGen

GrGen ist ein Generatorwerkzeug, das die in einer eigens entwickelten Sprache formulierten Graphersetzungsregeln und Graphtypen in ausführbaren Code umsetzen kann. Es kann unter anderem die von der LibGr ladbaren *Models* erzeugen. GrGen ist in der für einen Übersetzer üblichen Schichtenarchitektur aufgebaut. Es besitzt ein *Frontend*, das die Spezifikationssprache syntaktisch und semantisch analysiert und daraus eine *Zwischendarstellung* aufbaut, die bequemen Zugriff auf die einzelnen Elemente der Spezifikation ermöglicht. Daran können beliebige *Backends*[1] angeschlossen werden, die aus der Zwischendarstellung *etwas* erstellen. In dieser Diplomarbeit wurde ein *Backend* erzeugt, das *Models* für die LibGr erzeugt und dabei das Finden von Graphmustern mit relationaler Algebra implementiert[2].

5.2.1 Die Spezifikationssprache von GrGen

Mit der Spezifikationssprache von GrGen können Graphersetzungsregeln und Typen von Graphen, gemäß dem in Abschnitt 4.2 vorgestellten Graphmodell beschrieben werden. Zunächst betrachten wir die Beschreibung der Graphtypen.

Typdeklarationen

Die beiden Typen $\perp_E$ und $\perp_K$ heißen `Node` und `Edge`, besitzen keine Attribute und sind immer definiert. Neue Ecken- und Kantentypen können mit einer Klassendeklaration erzeugt werden:

```
node class Operator {
}
```

[1] Nicht zu verwechseln mit den *Backends* der LibGr

[2] Es sei bemerkt, dass GrGen keineswegs von LibGr abhängig ist, da aus der Zwischendarstellug beliebiges erzeugt werden kann.

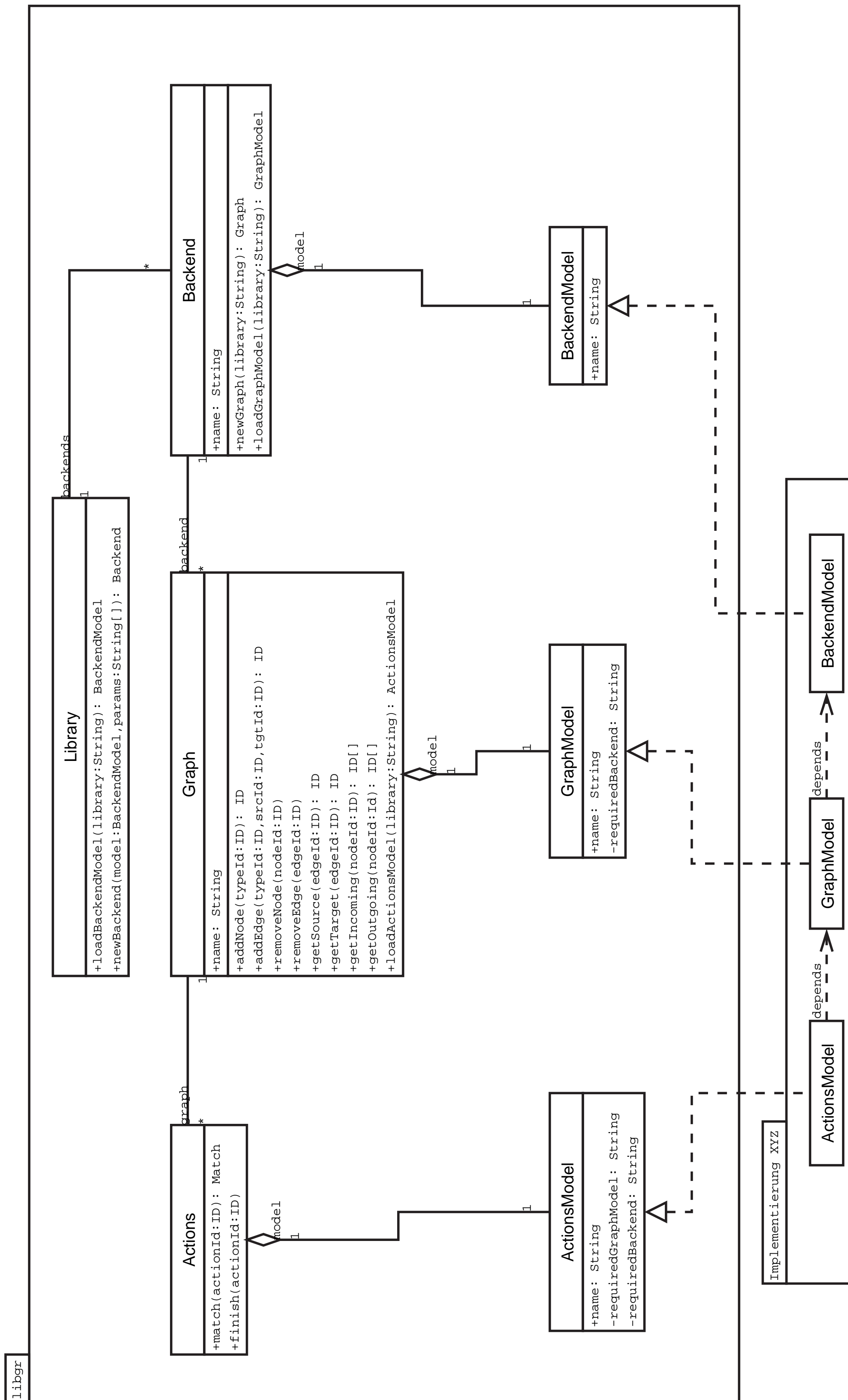


Abbildung 5.1: UML-Klassendiagramm der Hauptkomponenten der LibGr

```
edge class Operand {
}
```

`A` bzw. `B` erben implizit von `Node` bzw. `Edge`. Will man Eckenklassen (Kantenklassen) von anderen erben lassen, so versieht man die Deklaration mit einem Doppelpunkt und führt die ererbten Klassen danach auf:

```
node class Constant: Operator, Evaluateable {
}
```

Neben Ecken- und Kantenklassen kann man noch Aufzählungstypen deklarieren, die den `enums` aus C ähneln.

```
enum Mode {
  Integer, Float, Boolean
}
```

Einzelne Aufzählungselemente können auch mit Ausdrücken initialisiert werden. Diese müssen jedoch konstant sein:

```
enum Rights {
  execute = 1, write, read = 1 << 2, readwrite = write | read
}
```

Hat ein Aufzählungselement keinen Initialisierer, so bekommt es den Wert des vorigen +1. Das erste Element erhält (sofern es keinen Initialisierer besitzt) den Wert 1. Ein Aufzählungselement kann auch mithilfe eines vorigen initialisiert werden wie `readwrite` im letzten Beispiel. Im Gegensatz zu C gehen die Aufzählungselemente nicht in den globalen Namensraum über, sondern müssen mit dem Aufzählungsnamen qualifiziert werden.

Die Typen der Attribute von Ecken- bzw. Kantenklassen können entweder Aufzählungstypen oder einer der primitiven Typen `int`, `boolean` und `string` sein. Sie werden wie in Pascal mit deklariert:

```
node class Operator {
  name:string;
  unsigned:boolean;
  mode:Mode;
  bitWidth:int;
}

edge class Operand {
  position:int;
}
```

Deklaration von Graph-Aktionen

GrGen kennt zwei Arten von „Aktionen“, die auf Graphen angewandt werden können: Tests und Regeln. Mit einem Test kann man einen Subgraph in einem Graphen suchen. Bei Regeln kann zusätzlich noch die Ersetzung des gefundenen Subgraphs durch einen anderen Graphen gemäß dem SPO-Ansatz spezifiziert werden. Betrachten wir zunächst folgendes Beispiel eines Tests.

```
test HasConstOperand {
  pattern {
    op:Operator --> c:Constant;
  }
}
```

Dieser Test beschreibt einen zu suchenden Teilgraphen, indem eine Ecke des Typs `Operator` mit einer Ecke des Typs `Constant` verbunden ist. Die beiden Ecken werden desweiteren mit den Bezeichnern `op` und `c` belegt, da man eventuell später auf sie zurückgreifen möchte. So könnte man zum Beispiel einen Test angeben, der auf einen Graphen passt, in dem zwei Ecken vom Typ `Operator` mit der selben Ecke vom Typ `Operator` verbunden sind (die Einrückungen und Ausrichtungen dienen nur der besseren Lesbarkeit):

```
test BinaryOp {
  pattern {
    a:Operator --> op0:Operator;
    a          --> op1:Operator;
  }
}
```

Die Kanten sind in diesem Beispiel vom Typ `Edge`. Natürlich kann man den Kantentyp auch näher spezifizieren:

```
test BinaryOp {
  pattern {
    a:Operator -arg0:Operand-> op0:Operator;
    a          -arg1:Operand-> op1:Operator;
  }
}
```

Die Kanten müssen jetzt vom Kantentyp `Operand` sein, wenn der Test erfolgreich sein soll. An die Attribute der Ecken und Kanten können zusätzlich Bedingungen gestellt werden:

```
test BinaryOp {
  pattern {
    a:Operator -arg0:Operand-> op0:Operator;
    a          -arg1:Operand-> op1:Operator;
  }
  cond {
    arg0.position == 0;
    a.mode == Mode.Integer;
  }
}
```

Im `cond`-Teil können mehrere Bedingungen angegeben werden. Der Ergebnistyp des Ausdrucks muss `boolean` sein. Die Gesamtbedingung ist dann wahr, wenn jede einzelne wahr ist. Soll auf eine Kante später nicht mehr verwiesen werden, so kann der Bezeichner vor dem Doppelpunkt entfallen:

```
test BinaryOp {
  pattern {
    a:Operator -arg0:Operand-> op0:Operator;
    a          -:Operand->     op1:Operator;
  }
  cond {
    arg0.position == 0;
    a.mode == Mode.Integer;
  }
}
```

Verwendet man viele Ecken desselben Typs oder eine Ecke häufiger in einem Muster, so kann man die Ecken zu Beginn des `pattern`s deklarieren:

```
test BinaryOp {
  pattern {
    node a:Operator, op0:Operator, op1:Operator;

    a -arg0:Operand-> op0;
    a -:Operand->     op1;
  }
  ...
}
```

Zeile 3 kann noch einfacher geschrieben werden:

```
    node (a, op0, op1):Operator;
```

Es sei noch darauf hingewiesen, dass sich

```
pattern {                    pattern{
  op:Operator;       und       node op:Operator;
}                            }
```

voneinander unterscheiden. Beide Muster deklarieren eine Ecke mit dem Typ `Operator` und mit Namen `op`. Das erste Muster stellt den Teilgraphen dar, der genau diese eine Ecke enthält, das zweite jedoch den leeren Teilgraphen. Alle Ecken, die auf die bereits beschriebene Weise in einem `pattern`-Teil deklariert werden, werden in dem vom Matcher berechneten Morphismus injektiv abgebildet. Das bedeutet, dass die in einem Graphen G gefundenen Bilder der beiden Ecken `op0` und `op1` niemals die gleiche Ecke in G sind; der Morphismus m ist ein Isomorphismus. In manchen Situationen, wie zum Beispiel in der obigen, möchte man spezifizieren können, dass zwei Ecken aus einem `pattern` auf *eine* Ecke abgebildet werden können. Somit wird m zum Epimorphismus. Hierzu verwendet man in der Eckendeklaration eine Tilde (`~`) anstelle eines Kommas (`,`). Möchte man also zulassen, dass `op0` und `op1` auf dieselbe Ecke abgebildet werden können, so schreibt man;

```
  pattern {
    node a:Operator, (op0 ~ op1):Operator;
    ...
  }
```

Es sei noch bemerkt, dass Kanten auch in anderer Richtung geschrieben werden können, was zu einer erhöhten Lesbarkeit der Spezifikation beitragen kann. Anstelle von

```
a -> b;
a -> c;
```

kann man auch

```
b <- a -> c;
```

schreiben. Es ist ferner möglich auszudrücken, dass zwei Ecken *nicht* durch eine Kante eines Typs verbunden sein sollen. Hierzu stellt man dem `-` oder dem `<-` ein `!` voraus. Will man z.B. in obigem Beispiel verhindern, dass `op0` und `op1` durch eine Kante verbunden sind, so schreibt man:

```
test BinaryOp {
  pattern {
    node (a, op0, op1):Operator;

    op1 <-:Operand- a -arg0:Operand-> op0 !-> op1;
  }
  ...
}
```

Mit Regeln kann man, zusätzlich zu reinen Teilgraphtests, Graphersetzungen angeben:

```
rule TransformMul {
  pattern {
    node m:MulOp, c:Constant, op:Operator;

    op <-e1:Operand- m -e0:Operand-> c;
  }
  cond {
    c.value == 2;
  }
  replace {
    node a:AddOp;

    op <-:Operand- a -e1-> op;
  }
}
```

Angenommen, in einem Graphen G existiert ein zu dem im `pattern`-Teil angegebenen Graphen homomorpher Teilgraph. Durch obige Regel werden alle Ecken, die in `pattern` vorhanden sind, aber nicht in `replace` (inklusive der eingehenden Kanten) gelöscht. Alle Ecken, die in `replace` vorhanden sind, aber in `pattern` fehlen, werden dem G hinzugefügt. Dasselbe gilt für die Kanten. Anonyme Kanten (Kanten der Bauart `-:Typ->`) werden erhalten, wenn sie in

`pattern` und `replace` dieselben Ecken verbinden. So werden in obigem Beispiel die Ecken `c`, `m` und die Kante `e0` gelöscht. Zusätzlich werden alle Kanten, die in G zu den gelöschten Ecken ein- bzw. ausgehend sind, mitgelöscht. Dies schreibt der SPO-Ansatz vor. Nun möchte man im obigen Beispiel die eingehenden Kanten zu `m` nicht verlieren, sondern vielmehr `m` zu `a` umwandeln. In Abschnitt 4.3 wurde diese Problematik theoretisch diskutiert. In GrGen kann man den Typ einer Ecke ändern, was dem Löschen einer Schlinge und dem Einfügen einer neuen Schlinge aus Abschnitt 4.3 entspricht. Hierzu verwendet man den Bezeichner der Ecke weiter und fügt den neuen Typ mit zwei Doppelpunkten (`::`) an:

```
rule TransformMul {
  pattern {
    node m:MulOp, c:Constant, op:Operator;

    op <-e1:Operand- m -e0:Operand-> c;
  }
  cond {
    c.value == 2;
  }
  replace {
    op <-:Operand- m::AddOp -e1-> op;
  }
}
```

Änderungen an Attributen werden im `eval`-Teil spezifiziert:

```
rule FoldConstantAdd {
  pattern {
    node a:AddOp, (c0, c1):Constant;

    c0 <-- a --> c1;
  }
  replace {
    a::Constant;
  }
  eval {
    a.value = c0.value + c1.value;
  }
}
```

5.3 Vergleich zu den Implementierungen der anderen Ansätze

GrGen zeigt am meisten Ähnlichkeiten zum Agg-System (siehe Abschnitt 3.3). Auch Agg verwendet den SPO-Ansatz zur Formalisierung des Ersetzungsschritts und Graphmodelle mit attributierten Ecken und Kanten. Jedoch steht

bei AGG die Verwendung als interaktive Umgebung zur Entwicklung von Graphersetzungssystemen im Vordergrund. Es ist wie auch PROGRES eher als Entwicklungsumgebung gedacht, wohingegen bei GRGEN der Aspekt der Generierung und Verwendung der Graphersetzer in anderen Programmen (Übersetzern) im Vordergrund steht. Mit GRGEN kann man keine nicht-injektiven Matchings auf Kanten beschreiben, obwohl das im SPO-Ansatz zulässig ist. Desweiteren müssen die Morphismen von einer linken Seite zu einer rechten Seite immer injektiv sein. Somit können keine Ecken aus der linken Seite zu einer Ecke aus der rechten Seite „verschmolzen“ werden, was in AGG möglich ist. Diese Fähigkeiten schienen uns für unser Einsatzszenarium nicht von Bedeutung und wurden somit nicht implementiert. Dies ermöglichte uns ferner die Spezifikationssprache einfacher zu gestalten.

Im Vergleich zu PROGRES ist GRGEN wesentlich schlanker und modularer konzipiert. PROGRES ist eher als visuelle Programmiersprache oder integrierte Entwicklungsumbgebung für visuelle Datenbanken gedacht. Insbesondere ist PROGRES auf die ihm unterliegende Datenbank GRAS zugeschnitten, was ein flexibles Tauschen des Matchers und der Speicherung der Graphen fast unmöglich macht.

OPTIMIX ist wie GRGEN auch ein Generatorwerkzeug, keine Entwicklungsumgebung. OPTIMIX ist speziell für die Erzeugung von Optimierern in Übersetzern entworfen worden, aber aufgrund seiner Fixierung auf DATALOG zur Graphersetzung nur für einen bestimmten Kreis von Optimierungen verwendbar. Abschnitt 3.2 bietet eine genauere Darstellung von OPTIMIX.

Folgende Tabelle fasst einige Unterschiede der betrachteten Werkzeuge zusammen. Einen Vergleich der formalen Modelle bietet Abschnitt 4.5.

Eigenschaft	PROGRES	OPTIMIX	AGG	GRGEN
Textuell/Graphisch	beides	textuell	graphisch	textuell
Generierung	Modula-2/C	JAVA/C	–	C+SQL
Regelanwendungsbedingungen				
positiv	✓	✓	✓	✓
negativ	Ecken/Pfade	Kanten	Graphen	Kanten
Attribute	✓	✓	✓	✓

6 Erste Experimente

Wie bereits in der Einleitung dieser Arbeit erwähnt, liegt das erste Einsatzgebiet von GrGen in der Verarbeitung von Zwischendarstellungsgraphen wie sie in Übersetzern verwendet werden. Hierbei sollen Graphmuster, die den Einsatz von Komplexbefehlen nahelegen und von traditionellen Codeerzeugern mit Termersetzungsverfahren (vgl. hierzu [NKWA96] und [PLG88]) nicht behandelbar sind, umgeformt werden.

Moderne Übersetzer erzeugen Vektorbefehle meist aus Schleifen, die vektorisierbar sind. Bereits ausgerollte Schleifen oder Code, der nicht aus Schleifen stammt, werden nicht vektorisiert. Abbildung 6.1 zeigt den Zwischendarstellungsgraphen folgender Beispielroutine, die Möglichkeiten zur Vektorisierung bietet:

```
void vec2_add(float *a, float *b, float *res) {
  res[0] = a[0] + b[0];
  res[1] = a[1] + b[1];
}
```

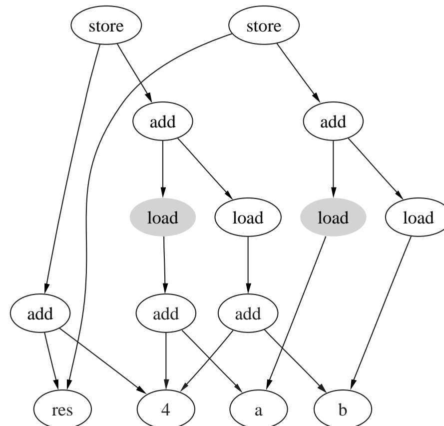
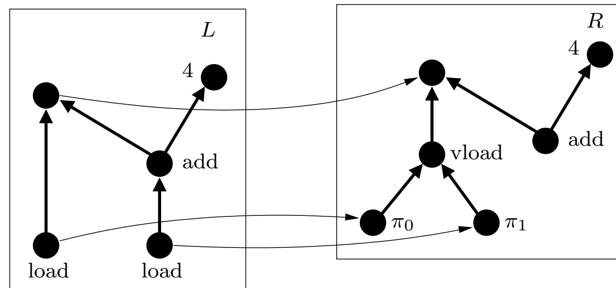


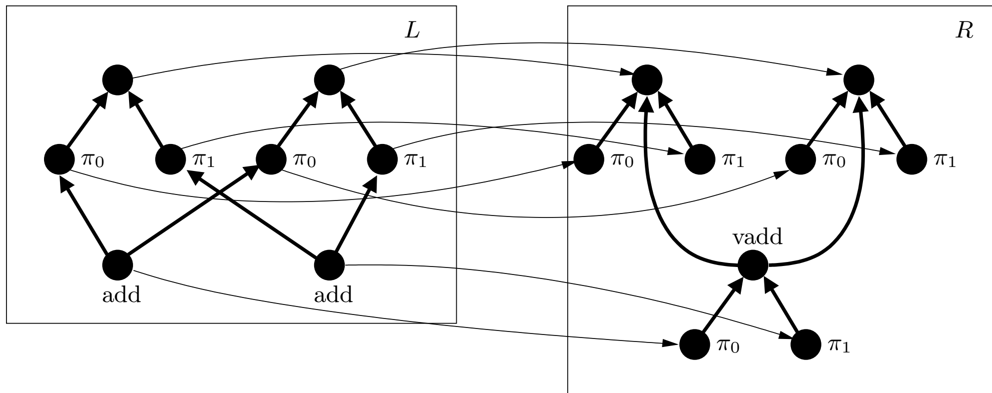
Abbildung 6.1: Der Datenabhängigkeitsgraph des Beispielprogramms

Es ist ersichtlich, dass die beiden Ladeinstruktionen, die `a[0]` und `a[1]` laden (in Abbildung 6.1 grau gefärbt), zu einer Vektor-Ladeinstruktion zusammengefasst werden können. Selbiges gilt natürlich für die Ladeinstruktionen des Vektors `b`. Nun können die beiden Ladeinstruktionen im Graphen nicht

durch eine einzige (Vektor-) Ladeinstruktion ersetzt werden, da andere Kanten zu den beiden Ecken inzident sind. Diese dürfen nicht einfach auf eine einzige Vektor-Ladeinstruktion zeigen, da die eine auf den unteren, und die andere den oberen Teil des Vektorwertes verweist. Somit müssen die beiden Ladeinstruktionen durch zwei Projektionsecken ersetzt werden, die aus dem Vektorwert den unteren bzw. den oberen Teil extrahieren. Können alle Folgeinstruktionen, die einen Teil des Vektorwerts verwenden, auch vektorisiert werden, so verwaisen die Projektionsecken und können gelöscht werden. Eine Regel, die die beiden Ladeinstruktionen vektorisiert sieht beispielsweise so aus:



Hier sieht man deutlich, dass diese Umformung *keine* Termumformung ist. Folgende Regel verarbeitet die Additionsinstruktionen zu einer Vektoraddition, wenn beide Operanden vektorisiert sind.



Es fehlen natürlich weitere Regeln, die vektorisierte Werte speichern, also Store-Instruktionen behandeln. Desweiteren werden Regeln zur Umordnung der einzelnen Elemente eines Vektorwertes benötigt. Wie bereits erwähnt, soll auf die Steuerung der Regelanwendung in dieser Arbeit nicht eingegangen werden. GrGen ist jedoch durch die einzelne Aktivierbarkeit der Regeln in der Lage nahezu beliebige Anwendungsparadigmen zu ermöglichen.

Zuletzt wollen wir für die beiden oben vorgestellten Regeln eine GrGen-Spezifikation angeben. Der Übersicht halber verzichten wir hier auf die Deklaration der Ecken- und Kantenklassen.

```
rule V2Load {
  pattern {
    node (load0, load1):Load;
    node n:Dataflow, c:Const4, add:Add;
```

```
      load0 --> n;
      load1 --> add (--> n, --> c);
    }
    replace {
      load0::Pi0 --> v:VectorLoad <-- load1::Pi1;
      v --> node;
      add (--> node, --> c);
    }
  }

  rule V2Add {
    pattern {
      node (leftp0, rightp0):Pi0;
      node (leftp1, rightp1):Pi1;
      node (v0, v1):VectorInstruction;
      node (a0, a1):Add;

      leftp0 --> v0 <-- leftp1;
      rightp0 --> v1 <-- rightp1;
      leftp0 <-- a0 --> rightp0;
      leftp1 <-- a1 --> rightp1;
    }
    replace {
      leftp0 --> v0 <-- leftp1;
      rightp0 --> v1 <-- rightp1;
      v1 <-- vadd:VectorAdd --> v0;
      a0::Pi0 --> vadd <-- a1::Pi1;
    }
  }
```

Der Musterteil der Regel `V2Load` wird von GrGen zu folgender SQL-Anweisung übersetzt:

```
SELECT n244.node_id, n221.node_id, n230.node_id, n234.node_id, n225.node_id,
  e242.edge_id, e223.edge_id, e232.edge_id, e239.edge_id
FROM nodes AS n244, nodes AS n221, nodes AS n230, nodes AS n234,
 nodes AS n225, edges AS e242, edges AS e223, edges AS e232, edges AS e239
WHERE node_type_is_a(n244.type_id, 3)
AND n244.node_id <> n221.node_id
AND n244.node_id <> n230.node_id
AND n244.node_id <> n234.node_id
AND n244.node_id <> n225.node_id
AND n244.node_id = e242.tgt_id
AND node_type_is_a(n221.type_id, 2)
AND n221.node_id <> n230.node_id
AND n221.node_id <> n234.node_id
AND n221.node_id <> n225.node_id
AND n221.node_id = e223.src_id
AND node_type_is_a(n230.type_id, 2)
AND n230.node_id <> n234.node_id
AND n230.node_id <> n225.node_id
AND n230.node_id = e232.src_id
AND node_type_is_a(n234.type_id, 6)
AND n234.node_id <> n225.node_id
```

```
AND n234.node_id = e242.src_id
AND e242.src_id = e239.src_id
AND e239.src_id = e232.tgt_id
AND node_type_is_a(n225.type_id, 1)
AND n225.node_id = e223.tgt_id
AND e223.tgt_id = e239.tgt_id
AND edge_type_is_a(e242.type_id, 0)
AND edge_type_is_a(e223.type_id, 0)
AND edge_type_is_a(e232.type_id, 0)
AND edge_type_is_a(e239.type_id, 0)
```

Im Laufe dieser Arbeit wurden diese Regeln erfolgreich auf Graphen der Zwischendarstellung FIRM (siehe Abschnitt 4.1.1) angewendet. Dabei wurden GRGEN-Backends zur Ansteuerung der Datenbanksysteme MYSQL und POSTGRESQL implementiert. MYSQL war nicht brauchbar, da der Anfrageoptimierer bei Anfragen von obiger Bauart (Verbunde über mehr als acht Relationen) nach 90 Minuten noch nicht zu einem Ende fand. POSTGRESQL konnte diese Anfragen dank genetischer Anfrageoptimierung in erträglicher Zeit optimieren (wenige Sekunden pro Anfrage mit mehr als 15 Verbunden). Durch diese randomisierte Anfrageoptimierung wurden nicht immer effiziente Anfragen erzeugt. Gelang die Optimierung der Anfrage, so dauerte eine Anfrage in obiger Gestalt ca. 200ms bei einer Problemgröße von ca. 300 Ecken. Schlecht optimierte Anfragen dauerten Minuten. Aus Zeitgründen konnten weitere Optimierungen in Form von Anfrageumstrukturierung oder Anpassen der Datenbankparameter nicht durchgeführt werden.

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Graphersetzungswerkzeug zur Manipulation von graphbasierten Zwischendarstellungen in Übersetzern samt zugehörigem Formalismus vorgestellt. Die Möglichkeit, komplexe Graphmuster und deren Ersetzung in einer mächtigen Sprache zu spezifizieren, erleichtert die Erstellung von Optimierungen auf einer graphbasierten Zwischendarstellung erheblich, wie bereits durchgeführte Experimente zeigen.

Das Graphmodell genügt allen Erfordernissen, die durch eine graphbasierte Zwischendarstellung wie FIRM gestellt werden. Es ist darüber hinaus auch für andere Einsatzbereiche ausserhalb des Übersetzerbaus verwendbar.

Graphersetzung ist auch für die Befehlsauswahl interessant. Hier existieren bereits leistungsfähige Verfahren (siehe [PLG88] und [NKWA96]), die mittels Termersetzung (also Baumersetzung) Zwischendarstellung in Maschinensprache umsetzen. Eine Erweiterung auf Graphersetzungsverfahren wäre nur konsequent.

Die Ersetzung mittels SPO-Ansatz bietet ein theoretisch fundiertes Modell der Überführung eines Graphen in einen anderen und ist vom eingesetzten Verfahren zur Auswahl des zu ersetzenden Teilgraphen völlig unabhängig. Diese Eigenschaft erlaubt das Austauschen und Optimieren des Matchers, ohne die theoretische Fundierung der Ersetzung ändern zu müssen. Dies ist eminent wichtig, da der Matcher die Stelle eines GES ist, die für die Gesamtleistung des Systems ausschlaggebend ist. Diese durchgängige formale Beschreibung erleichtert auch die Verifikation der spezifizierten Optimierungen. In Kombination mit dem Relationalen-Algebra-Matcher kann das Transaktionsprotokoll der Datenbank als Zertifikatmenge für einen Programmprüfer angesehen werden.

Durch das Matchen mit relationaler Algebra wurde ein Verfahren entwickelt, das schnell implementiert werden konnte, Graphmuster ohne Einschränkung an das Graphmodell finden kann und leicht nachvollziehbare Matcher erzeugt. Dadurch konnte das implementierte System schnell an einen Übersetzer angeschlossen und erste Versuche durchgeführt werden.

A Grundbegriffe der Kategorientheorie

Die Kategorientheorie stammt aus der algebraischen Topologie und ist ein vergleichsweise junges Teilgebiet der reinen Mathematik. Sie nimmt eine immer bedeutendere Stellung in der theoretischen Informatik ein, insbesondere bei der Formalisierung von funktionalen Programmiersprachen. Gute Einführungen in die Kategorientheorie, speziell im Bezug auf die Informatik, sind [EMC$^+$01], [Pie91] und [Fok92]. Im Folgenden sollen nur die für die algebraische Graphersetzung wichtigen Konzepte kurz vorgestellt werden.

A.1 Kategorien und Diagramme

Definition A.1 (Kategorie) Eine Kategorie $\mathbf{C}$ besteht aus:

1. Eine Klasse von Objekten (hier mit $A, B, C, \ldots$ benannt).
2. Eine Klasse von *Morphismen* (hier mit $f, g, h, \ldots$ bezeichnet).
3. Operationen, die jedem Morphismus f ein Objekt $\operatorname{dom} f = A$, seine Quelle[1] und $\operatorname{cod} f = B$, sein Ziel[2] zuweisen. Man schreibt auch: $f : A \to B$ oder $A \xrightarrow{f} B$.
4. Ein Verkettungs-Operator $\circ$, der jedem Paar von Morphismen f und g mit $\operatorname{cod} f = \operatorname{dom} g$ einen Pfeil $g \circ f : \operatorname{dom} f \to \operatorname{cod} g$ zuweist und assoziativ ist, also
$$h \circ (g \circ f) = (h \circ g) \circ f$$
Für $g \circ f$ findet man oft auch die Notation $f; g$.
5. Für jedes Objekt A gibt es einen Morphismus $\mathrm{id}_A : A \to A$, der für jeden Morphismus $f : A \to B$ die Neutralitäts-Eigenschaft
$$\mathrm{id}_B \circ f = f \text{ und } f \circ \mathrm{id}_A = f$$
erfüllt.

Da die Beziehung von Verkettungen verschiedener Morphismen oft eine wichtige Rolle spielt, stellt man diese in Diagrammen dar. Ein Pfad $A_0 \xrightarrow{f_1} A_1 \xrightarrow{f_2} \cdots \xrightarrow{f_n} A_n$ in einem Diagramm entspricht dem Morphismus $f : A_0 \to A_n$, der durch $f = f_n \circ (\cdots (f_2 \circ f_1) \cdots)$ gegeben ist.

[1] engl.: Domain
[2] engl.: Codomain

Beispiel A.2 Gilt zum Beispiel in einer Kategorie $\mathbf{C}$

$$g \circ f = k \circ h$$

so sagt man: Das Diagramm

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle g} \\ C & \xrightarrow[k]{} & D \end{array}$$

kommutiert.

Beispiel A.3 Die Graphen und deren Homomorphismen bilden die Kategorie **Graph**. Sei $G = (E, K, \mathrm{src}, \mathrm{tgt})$ ein Graph wie in Definition 2.1. Seien $f = (f_E, f_K)$, $g = (g_E, g_K)$ Graph-Homomorphismen wie in Definition 2.5. Die Verkettung von f und g ist definiert durch:

$$g \circ f = (g_E \circ f_E, g_K \circ f_K)$$

Durch die Assoziativität der Graph-Homomorphismen ist die Assoziativität der Morphismen in **Graph** gewährleistet. Die Neutralität resultiert aus der Existenz des Identitäts-Homomorphismus für jeden Graphen.

A.2 Universelle Konstruktionen

Definition A.4 (Initiale und finale Objekte) Ein Objekt $\mathbf{0}$ heißt *initiales* Objekt, wenn für jedes Objekt A ein Morphismus $f : \mathbf{0} \to A$ existiert. Ein Objekt $\mathbf{1}$ heißt *finales* Objekt, wenn für jedes Objekt A ein Morphismus $f : A \to \mathbf{1}$ existiert.

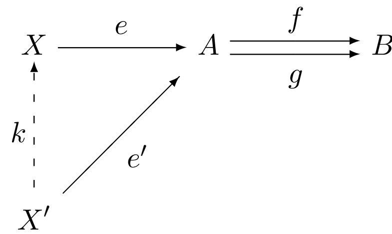
Beispiel A.5 Der leere Graph ist initial, der Graph mit einer Ecke und einer Schlinge ist final in **Graph**.

Definition A.6 (Universelle Konstruktion) Eine universelle Konstruktion ist gegeben durch eine Klasse von Objekten und zugehörigen Morphismen, die, gesehen als Objekte einer Kategorie, final sind.

Beispiel A.7 (Egalisatoren) Ein Morphismus $e : X \to A$ heißt *Egalisator* von zwei Morphismen $f : A \to B$, $g : A \to B$, wenn

1. $f \circ e = g \circ e$
2. Für alle $e' : X' \to A$, für die $f \circ e' = g \circ e'$ gilt existiert genau ein Morphismus $k : X' \to X$ mit $e \circ k = e'$

oder folgendes Diagramm kommutiert:



Der gestrichelte Pfeil symbolisiert, dass es genau einen Morphismus k gibt.

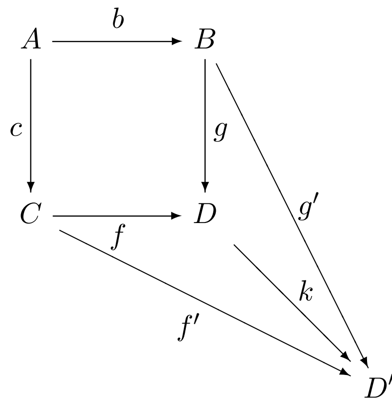
Betrachtet man die Kategorie, in der alle Egalisatoren von f und g Objekte und die Morphismen k die Morphismen sind, so ist e in dieser Kategorie final, da nach obiger Definition von allen Egalisator-Objekten e' genau ein Morphismus k zu e existiert. e ist somit ein *optimaler* Repräsentant für alle Egalisatoren von f und g. Die Bedingung 2 wird auch als Universalitätseigenschaft bezeichnet.

Definition A.8 (Pushout und Pushout-Komplement) Sei $\mathbf{C}$ eine Kategorie. Gegeben sind zwei Morphismen $b : A \to B$, $c : A \to C$. Das Tripel $(D, g : B \to D, f : C \to D)$ heißt *Pushout* von (b, c), wenn folgende Eigenschaften erfüllt sind:

Kommutativität
: $g \circ b = f \circ c$

Universalität
: Für alle $g' : B \to D'$ und $f' : C \to D'$ mit $g' \circ b = f' \circ c$ gibt es genau einen Morphismus $k : D \to D'$, so dass $g' = k \circ g$ und $f' = k \circ f$.

Wie auch bei den Egalisatoren (siehe Beispiel A.7) lassen sich diese Bedingungen in einem Diagramm übersichtlicher darstellen:



D heißt dann Pushout-Objekt von (b, c). Desweiteren heißt das Tripel $(C, c : A \to C, f : C \to D)$ *Pushout-Komplement* von (b, g). C heißt dann Pushout-Komplement-Objekt.

B Grundbegriffe der Relationalen Algebra

Wir wollen hier die für diese Arbeit nötigen Konzepte der relationalen Algebra vorstellen. Die relationale Algebra ist die Grundlage aller moderner Datenbanksysteme und wurde 1970 von Codd vorgestellt [Cod70]. Eine detaillierte Darstellung der relationalen Algebra und anderer Konzepte der Theorie relationaler Datenbanken findet man z.B. in [Kan90].

B.1 Definitionen

Sei $\mathcal{U}$ eine abzählbar unendliche Menge. Die Elemente von $\mathcal{U}$ heißen *Attribute* und werden im folgenden mit $a_1, a_2, \ldots, a_n$ bezeichnet. Jedem Attribut a ist eine Wertemenge $\Delta_a \cup \perp$ zugeordnet. Das Element $\perp$ ist ein spezieller Wert, der oft dazu verwendet wird, das Nichtvorhandensein einer Information auszudrücken.

Sei nun U eine endliche Teilmenge von $\mathcal{U}$. Eine Teilmenge R von U heißt *relationales Schema*. Eine endliche Menge $D = \{R_i \mid i = 1, \ldots, n\}$ relationaler Schemata mit $\bigcup_{i=1}^{n} R_i = U$ heißt *Datenbankschema* über U.

Sei D ein Datenbankschema über U und X eine beliebige Teilmenge von U. Ein X-Tupel ist eine Abbildung, die jedem Attribut $a \in X$ ein Element der Menge Δ_a zuordnet. Für ein relationales Schema R über U bezeichnen wir eine endliche Menge von R-Tupeln als *Relation* r über R. Das zu einer Relation r gehörende Schema wollen wir im folgenden mit $\mathcal{S}(r)$ bezeichnen. Im folgenden spielt die Einschränkung eines X-Tupels auf eine Teilmenge $Z \subseteq X$ eine Rolle. Sie soll mit $t[Z]$ bezeichnet werden. Zuletzt bezeichnen wir noch eine endliche Menge von Relationen, die den Schemata eines Datenbankschemas genügen als *Datenbank*.

Anschaulich entspricht ein relationales Schema der Struktur einer Tabelle in einem relationalen Datenbanksystem. Das Datenbankschema beschreibt als Menge von relationalen Schemata somit eine Menge der Tabellenstrukturen. Ein R-Tupel ist eine Zeile einer Tabelle, die dem Schema R genügt. Die Relation r entspricht als Menge von R-Tupeln dem Inhalt der Tabelle.

B.2 Operationen auf Relationen

Eine Datenbank ist natürlich nur von Nutzen, wenn man auf die darin enthaltenen Daten zugreifen kann, ohne alle Relationen in ihrer Gesamtheit betrachten zu müssen. Hierzu benötigt man sog. *Datenbank-Abfragen*[1]. Eine Abfrage

[1] engl.: Query

ist nichts anderes als eine Abbildung einer Datenbank auf eine andere. Nun ist man sicher nicht an jeder dieser Abbildungen interessiert, sondern an solchen, die sich leicht berechnen lassen, aber trotzdem den intuitiven Ansprüchen an eine Abfrage gerecht werden. Hierzu definierte CODD[2] eine Menge weniger grundlegender Operationen auf Relationen, aus denen Abfragen konstruiert werden können. Die Operationen stellen in manchen Fällen Bedingungen an die Verträglichkeit der Schemata ihrer Operanden und definieren die Gestalt des Schemas ihres Ergebnisses. Letztlich lassen sich diese Operationen aus den folgenden fünf Operationen zusammensetzen:

Projektion

Die Projektion $\pi_X(r)$ konstruiert eine neue Relation aus einer Teilmenge der Spalten einer anderen Relation r. Sei t ein X-Tupel.

Bedingung

$X \subseteq \mathcal{S}(r) \wedge \mathcal{S}(\pi_X(r)) = X$

Bedeutung

$\pi_X(r) = \{t[X] \mid t \in r\}$

Selektion

Die Selektion $\sigma_\Theta(r)$ bildet eine neue Relation, indem aus einer Relation r Tupel anhand des Prädikats $\Theta : r \to \{\mathit{true}, \mathit{false}\}$ auswählt.

Bedingung

$\mathcal{S}(\sigma_\Theta(r)) = \mathcal{S}(r)$

Bedeutung

$\sigma_\Theta(r) = \{t \mid t \in r \wedge \Theta(t)\}$

Kartesisches Produkt

Das kartesische Produkt $r_1 \times r_2$ erstellt aus zwei Relationen r_1 und r_2 eine neue, die alle Kombinationen der r_1 und r_2 beinhaltet.

Bedingung

$\mathcal{S}(r_1 \times r_2) = \mathcal{S}(r_1) \cup \mathcal{S}(r_2)$

Bedeutung

$$r_1 \times r_2 = \{t \mid t \text{ ist ein } (\mathcal{S}(r_1) \cup \mathcal{S}(r_2))\text{-Tupel mit } t[\mathcal{S}(r_1)] \in r_1 \text{ und } t[\mathcal{S}(r_2)] \in r_2\}$$

Vereinigung

Die Vereinigung $r_1 \cup r_2$ bildet eine neue Relation aus zwei Relationen r_1 und r_2, die das gleiche Schema besitzen. Die neue Relation beinhaltet alle Tupel aus r_1 und r_2.

Bedingung

$\mathcal{S}(r_1) = \mathcal{S}(r_2) = \mathcal{S}(r_1 \cup r_2)$

[2]Er nannte die aus diesen Basisoperationen konstruierbare Sprache *Relational Algebra Query Language*.

Bedeutung
$r_1 \cup r_2 = \{t \mid t \in r_1 \vee t \in r_2\}$

Differenz
Die Differenz $r_1 \setminus r_2$ enthält alle Tupel, die in r_1 enthalten sind, aber nicht in r_2, unter der Voraussetzung, dass die Schemata von r_1 und r_2 gleich sind.

Bedingung
$\mathcal{S}(r_1) = \mathcal{S}(r_2) = \mathcal{S}(r_1 \setminus r_2)$

Bedeutung
$r_1 \setminus r_2 = \{t \mid t \in r_1 \wedge t \notin r_2\}$

Entscheidend für Datenbanken (und auch für Teile dieser Arbeit) ist der sogenannte *Verbund*[3]. Der Verbund ist eine Kombination von Selektion und kartesischem Produkt und wird aufgrund seines häufigen Vorkommens als eigenständige Operation betrachtet. Er verbindet zwei Relationen vermöge eines Prädikats Θ, wie es auch bei der Selektion auftrat und verschmilzt somit zwei Relationen aufgrund von Gemeinsamkeiten der Werte bestimmter Attribute.

Gegeben sind zwei Relationen r_1 und r_2. Sei $\Theta : r_1 \times r_2 \to \{\mathit{true}, \mathit{false}\}$. Der Verbund $r_1 \bowtie_\Theta r_2$ zweier Relationen r_1 und r_2 ist definiert durch:

Bedingung
$\mathcal{S}(r_1 \bowtie_\Theta r_2) = \mathcal{S}(r_1 \times r_2)$

Bedeutung
$r_1 \bowtie_\Theta r_2 = \sigma_\Theta(r_1 \times r_2)$

Insbesondere kann man die Selektion und das kartesische Produkt als Spezialfälle des Verbundes sehen, nämlich:

$$r_1 \times r_2 = r_1 \bowtie_{\mathit{true}} r_2, \quad \sigma_\Theta(r_1) = \pi_{\mathcal{S}(r_1)}(r_1 \bowtie_\Theta \{(\bot)\})$$

Da bei einem Verbund die Schemata der zwei Relationen das gleiche Attribut aufweisen können, definiert man noch eine Umbenennungs-Operation, um in der Bedingung Θ auf die Attribute referenzieren zu können:

Umbenennung
Sei r eine Relation und $R = \mathcal{S}(r)$. Die Umbenennung $r_{x \leftarrow a_{R_i}}$ läßt die Tupel der Relation unberührt, ersetzt aber ein Attribut in R durch ein anderes, noch nicht in R vorhandenes.

Bedingung
Sei $R = (a_{R_1}, \ldots, a_{R_i}, \ldots, a_{R_n})$. Das Schema der Umbenennung ist dann: $\mathcal{S}(r_{x \leftarrow a_{R_i}}) = (a_{R_1}, \ldots, X, \ldots, a_{R_n})$ wobei $\Delta_{a_{R_i}} = \Delta_X$ gelten muss.

Bedeutung
$r_{x \leftarrow a_{R_i}} = \{t \mid t \in r\}$

[3] engl.: Join

B.3 Terme

Die Menge der Terme der relationalen Algebra ist genau wie bei arithmetischen Termen induktiv definiert. Eine Relation r ist ein Term der relationalen Algebra. Sind R, S Terme der relationalen Algebra, so sind auch

$$\pi_X(R),\ \sigma_\Theta(R),\ R \times S,\ R \cup S,\ R \setminus S,\ R \bowtie_\Theta S,\ R_{A \leftarrow B}$$

Terme der relationalen Algebra.

C Die Spezifikationssprache von GrGen

Die Spezifikationssprache von GrGen besitzt folgende Schlüsselwörter und Begrenzer:

```
unit node edge class enum test rule
pattern cond rule replace eval

!- -> !<- <- : :: ~ ! = * / % + - << >> >>>
< <= > >= == != & ^ | && || ? ( ) { } ; ,
```

Die folgende Grammatik folgt der EBNF-Spezifikation aus [GW84] und enthält folgende Konstrukte:

Konstrukt	Ersetze durch
$[\ a\ ]$	$a \mid .$
$a+$	a' mit $a' ::= a'\ a \mid a\ .$
$a*$	a' mit $a' ::= a'\ a \mid a \mid .$
$a \parallel b$	$a\ (\ b\ a\ )*$

Terminale sind in Schreibmaschinenschrift gesetzt, Nichtterminale kursiv.

text ::= `unit` ident `;` decl* .
decl ::= action-decl | class-decl | enum-decl .
action-decl ::= test-decl | rule-decl .
class-decl ::= (`node` | `edge`) `class` ident class-extends `{` class-body `}` .
enum-decl ::= `enum` ident `{` enum-body `}` .
class-extends ::= [`:` (ident ‖ `,`)] .
class-body ::= (basic-decl `;`)* .
basic-decl ::= ident `:` ident .
enum-body ::= [enum-item-decl ‖ `,`] .
enum-item-decl ::= ident [`=` expression] .
test-decl ::= `test` ident `{` pattern-part [cond-part] `}` .
rule-decl ::= `rule` ident `{` pattern-part [cond-part] replace-part [eval-part] `}` .
pattern-part ::= `pattern` `{` (pattern-stmt `;`)* `}` .
pattern-stmt ::= pattern-connections | `node` (pattern-node-decl ‖ `,`) .
pattern-connections ::= pattern-node-occ [pattern-cont] .
pattern-node-occ ::= ident | pattern-node-decl .
pattern-node-decl ::= multi-node-decl | homomorphic-node-decl .
homomorphic-node-decl ::= `(` (ident ‖ `~`) `)` `:` ident .
pattern-cont ::= pattern-pair [pattern-cont] | `(` (pattern-cont ‖ `,`) `)` .
pattern-pair ::= pattern-edge pattern-node-occ .
pattern-edge ::= pattern-positive-edge | pattern-negative-edge .

pattern-positive-edge ::= - pattern-edge-occ -> | <- pattern-edge-occ - .
pattern-edge-occ ::= [ident] : ident .
pattern-negative-edge ::= !- [: ident] -> | !<- [: ident] - .
multi-node-decl ::= (ident | ((ident ‖ ,))) : ident .
cond-part ::= cond { (expression ;)+ } .
replace-part ::= replace { (replace-stmt ;)* } .
replace-stmt ::= replace-connections | node (multi-node-decl ‖ ,) .
replace-connections ::= replace-node-occ | [replace-cont] .
replace-node-occ ::= ident | multi-node-decl | node-type-change .
node-type-change ::= ident :: ident .
replace-cont ::= replace-pair [replace-cont] | ((replace-cont ‖ ,)) .
replace-pair ::= replace-edge replace-node-occ .
replace-edge ::= (- replace-edge-occ -> | <- replace-edge-occ -) .
replace-edge-occ ::= [[ident] :] ident .
eval-part ::= eval { (assignment ;)* } .
assignment ::= qual-ident = expression .
qual-ident ::= (ident ‖ .) .
expression ::= cond-expr .
cond-expr ::= log-or-expr | log-or-expr ? expression : cond-expr .
log-or-expr ::= log-and-expr | log-or-expr || log-and-expr .
log-and-expr ::= bit-or-expr | log-and-expr && bit-or-expr .
bit-or-expr ::= bit-xor-expr | bit-or-expr | bit-xor-expr .
bit-xor-expr ::= bit-and-expr | bit-xor-expr ^ bit-and-expr .
bit-and-expr ::= eq-expr | bit-and-expr & eq-expr .
eq-expr ::= rel-expr | eq-expr [== | !=] rel-expr .
rel-expt ::= shift-expr | rel-expr [< | <= | > | >=] shift-expr .
shift-expr ::= add-expr | shift-expr [<< | >> | >>>] add-expr .
add-expr ::= mul-expr | add-expr [+ | -] mul-expr .
mul-expr ::= unary-expr | mul-expr [* | / | %] unary-expr .
unary-expr ::= [~ | ! | - | +] unary-expr | cast-expr | primary-expr .
cast-expr ::= (ident) unary-expr .
primary-expr ::= qual-ident | constant .
constant ::= true | false | number | string .

Danksagung

Ich danke allen Mitarbeitern des IPD für die angenehme Atmosphäre in den letzten zwei Jahren. Prof. Dr. Gerhard Goos danke ich für seine Unterstützung und sein Interesse am Thema, Dr. Sabine Glesner für ihre Betreuung. Insbesondere danke ich Michael Beck, Götz Lindenmaier und Markus Noga für die vielen fruchtbaren Gespräche, Diskussionen und dafür, dass ich eine Menge habe lernen dürfen.

Ganz besonders danke ich Rubino Geiß, dass er als Betreuer sowohl dieser Arbeit als auch meiner Tätigkeit als HiWi stets hinter mir stand, immer ein offenes Ohr für mich hatte und mir in zahlreichen Gesprächen neue Einsichten vermittelte.

Zuletzt danke ich Herrn Dr. phil. André Rupp für seine Freundschaft und sein Vertrauen in den letzten fünf Jahren. Meiner Familie verdanke ich mehr als ich hier zu Papier bringen kann. Last but not least danke ich Kerstin für ihre Liebe und ihre Geduld, die manchmal mit mir nötig war.

Literaturverzeichnis

[AEH⁺99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[Ass95] Uwe Assmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 1995.

[Ass00] Uwe Assmann. Graph rewrite systems for program optimization. *ACM Transaction On Programming Languages And Systems*, 22(4):583–637, 2000.

[Bau95] Michel Bauderon. A uniform approach to graph rewriting: The pullback approach. In *Workshop on Graph-Theoretic Concepts in Computer Science*, pages 101–115, 1995.

[BFG96] D. Blostein, H. Fahmy, and A. Grbavec. Practical use of graph rewriting. In J. Cuny, editor, *5th workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes In Computer Science*, pages 38–55, Berlin, 1996. Springer-Verlag.

[CEH⁺97] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, and Annika Wagner. Algebraic approaches to graph transformation - part I: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, pages 247–312. World Scientific, 1997.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications Of The ACM*, 13(6):377–387, 1970.

[Dör95] Heiko Dörr. *Efficient Graph Rewriting and its Implementation*, volume 922 of *Lecture Notes In Computer Science*. Springer-Verlag, 1995.

[DvB97] Rina Dechter and Peter van Beek. Local and global relational consistency. *Theoretical Computer Science*, 173(1):283–308, 1997.

[EKL91] Hartmut Ehrig, Martin Korff, and Michael Löwe. Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts. In G. Rozenberg, editor, *Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes In Computer Science*, pages 24–37. Springer Verlag, 1991.

[EMC+01] Hartmut Ehrig, Bernd Mahr, Felix Cornelius, Martin Große-Rhode, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik*. Springer Verlag, 2001.

[EPS73] H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: An algebraic approach. In *Proc.*, pages 167–180, 1973.

[ERT97] C. Ermel, M. Rudolf, and G. Taentzer. The AGG Approach: Language and Environment. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications*, pages 551–601. World Scientific, 1997.

[Fok92] M.M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72 of Part 1. University of Utrecht, September 1992.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[GW84] Gerhard Goos and William Waite. *Compiler Construction*. Springer Verlag, Januar 1984.

[Hal80] Rudolf Halin. *Graphentheorie I*. Wissenschaftliche Buchgesellschaft, 1980.

[HB02] Reiko Heckel and Luciano Baresi. Tutorial introduction to graph transformation: A software engineering perspective. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformation: First International Conference, ICGT 2002, Barcelona, Spain*, volume 2505 of *Lecture Notes In Computer Science*, pages 402–429. Springer-Verlag, Januar 2002.

[Kah02] Wolfram Kahl. A relation-algebraic approach to graph structure transformation. Technical Report 2002-03, Universität der Bundeswehr München, June 2002.

[Kan90] P. C. Kanellakis. Elements of relational database theory. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 1073–1156. Elsevier, Amsterdam, 1990.

[NKWA96] Albert Nymeyer, Joost-Pieter Katoen, Ymte Westra, and Henk Alblas. Code Generation = A* + BURS. In Tibor Gyimothy, editor, *Compiler Construction (CC)*, volume 1060 of *LNCS*, pages 160–176, Heidelberg, April 1996. Springer-Verlag.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[PLG88] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: an application burs theory. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 294–308. ACM Press, 1988.

[Rud97] Michael Rudolf. Konzeption und Implementierung eines Interpreters für attributierte Graphtransformation. Master's thesis, Technische Universität Berlin, 1997.

[Rud00] M. Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes In Computer Science*, Berlin, 2000. Springer-Verlag.

[Sch97] A. Schürr. Programmed graph replacement systems. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, pages 479–546. World Scientific, 1997.

[SWZ97] A. Schürr, A. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications*, pages 487–550. World Scientific, 1997.

[TLB99] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the intermediate representation firm. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999.

[Zün96] A. Zündorf. Graph pattern matching in progres. In *In Proc. 5th Intl. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073 of *Lecture Notes In Computer Science*, pages 454–468. Springer Verlag, 1996.

Index

AGG, 18
OPTIMIX, 17
PROGRES, 15
GRGEN, 33, 34

Attribute, 53
Ausgangsgrad, 4

Baumelbedingung, 11

Constraint Satisfaction Problem, 25

Datalog, 18
Datenbank, 53
Datenbank-Abfrage, 53
Datenbankschema, 53
Diagramm
  kommutiert, 50
direkte Ableitung, 6

EARS, 18
Egalisator, 50
Eingangsgrad, 4

Firm, 19

Grad, 4
Graph
  Grammatik, 13
  Homomorphismus, 4
    partieller, 5
  markiert, 4
  Multigraph, 3
  Typ, 16, 23, 33, 34

Identifikationsbedingung, 10

Kante
  baumelnd, 11
Klebebedingung, 11, 13
Klebegraph, 8
Konstruktion
  universelle, 50

Markierter Graph, 4
Match, 6
Morphismus, 49
Multigraphen, 3
Muttergraph, 6

NAC, 18

Objekt, 49
  final, 50
  initial, 50

PGRS, 15
Produktion, 7
Pullback-Ansatz, 13
Pushout, 51
Pushout-Komplement, 9, 51
Pushout-Objekt, 9

Redex, 8
Relation, 53
Relationales Schema, 53

Schlinge, 4
Signatur, 15
Single-Pushout-Ansatz, 12, 36
SQL, 31
Starke V-Struktur, 26

Teilgraph, 4

Untergraph, 4

Verbindung
  linke äußere, 31
Verbund, 55

XGRS, 18