Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

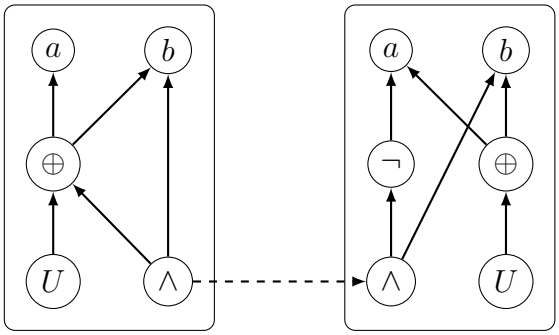
Lehrstuhl Prof. Dr.-Ing. Snelting

Bedingte Anwendung lokaler Optimierungen unter Berücksichtigung gemeinsamer Teilausdrücke

Bachelorarbeit von

Michael Hoff

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr.-Ing. Jörg Henkel
Betreuende Mitarbeiter:	Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 6. Februar 2014 – 12. Mai 2014

Zusammenfassung

Unberücksichtigte Mehrfachverwender können den positiven Effekt lokaler Optimierungen schmälern oder in Extremfällen auch Gesamtkosten erhöhen. Eine neue Optimierungsphase für lokale Optimierungen in LibFirm sammelt durch Optimierungsregeln mögliche Substitutionen und ermittelt die optimale Konfiguration über eine Reduktion auf PBQP. Bezüglich Mehrfachverwendung kritische Optimierungsregeln werden außerhalb dieser Phase deaktiviert um Erhöhungen der Kosten zu vermeiden. Es zeigt sich, dass der heuristische PBQP-Löser von LibFirm nicht für das verwendete Modellierungskonzept geeignet ist und für große Programme die hohe Kantenzahl der PBQP-Instanz eine Lösung mit dem Brute-Force-Löser stark erschwert. Die hier vorgestellte Implementierung konnte sowohl Laufzeitverschlechterungen als auch erhebliche Laufzeitverbesserungen von kompilierten Programmen erzielen.

Ignoring common subexpressions when applying local optimizations can lead to a restricted performance increase or even to a performance decrease in total. We integrate a new optimization phase for local optimizations into LibFirm. This phase uses the existing ruleset for local optimizations to gather possible substitutions and uses the optimization problem PBQP to identify the ideal configuration among those. In order to avoid destructive rule applications possibly increasing the total costs outside the phase's scope only a subset of rules are considered. It turns out that LibFirm's existing heuristic solver is not suitable for the modelation we chose. Additionally, LibFirm's brute force solver does not cope well with the large number of edges introduced due to its computational complexity. However, the presented implementation shows several performance decreases but also significant runtime benefits for compiled programs under certain circumstances.

Inhaltsverzeichnis

Symbolverzeichnis

$\perp_p$	VOID-Alternative eines PBQP-Knoten p
$\mathbb{T}$	Menge aller Knotentypen
$\mathbb{T}_{\text{loop}}$	Teilmenge der Knotentypen mit Potential zur Zyklenbildung
$\mathcal{Z}$	Zerlegung/Partitionierung einer Menge
$\bigcirc_{e \in T} f_e$	Funktionskomposition mit Selektor $e \in T$ und Funktionenfamilie f_e
ω	Wurzel eines Graphen
rl	Abbildung auf Realisierungskosten eines Knotentyps
rl_N	Abbildung auf Realisierungskosten eines einzelnen Knotens
op	Abbildung auf Menge der Operanden
op_{pos}	Abbildung auf Menge direkter Operanden mit ihren Indizes
reach_E^n	Abbildung auf alle über n Kanten erreichbare Knoten in G
$\hat{E}$	Eine Kantenmenge mit Mehrfachkanten
t	Typisierungsfunktion $\text{t} : N \to \mathbb{T}$
C_T^n	Substitutionskanten bezüglich T mit Propagationsindex n
D	Abhängigkeitskanten, engl. Dependency
G_C	Unzusammenhängender Programmgraph mit allen möglichen Konstellationen von Programmgraphen
N	Programmknoten, Operationen, Werte, engl. Node

n Im Kontext Propagation: Maximale Propagationstiefe

R Menge von lokalen Optimierungsregeln

T Kantenmenge aus Transformationskanten hergeleitet aus N und R

$V(E)_\omega$ Erreichbare Knotenmenge ausgehend von Wurzel ω über Kantenmenge E

 Substitutionskante

 Abhängigkeitskante

 Transformationskante

 Partition um einen (hier leeren) Programmknoten

 Programmknoten mit Operatorinformation (hier: Addition)

 Programmknoten mit konstantem Wert (hier: 1)

 Abstrakter Programmknoten/Subgraph mit Namen (hier: A)

SZK starke Zusammenhangskomponente

ZHK (schwache) Zusammenhangskomponente

1 Einführung

Ein Compiler übersetzt eine Quellsprache, oftmals eine höhere Programmiersprache, wie C oder Java, in eine Zielsprache. Diese Zielsprachen sind zumeist maschinenabhängig und ausführbar, oder wie im Fall von Java eine Zwischensprache, welche zur Ausführung eine spezielle Umgebung benötigt. Der Übersetzungsprozess ist in modernen Compilern in drei Teile zu gliedern. Das Frontend liest die Quellsprache und konstruiert daraus eine quellsprachen- und maschinenunabhängige Zwischenrepräsentation. Auf dieser Zwischendarstellung werden im Middleend Optimierungen, darunter die in dieser Ausarbeitung behandelten *lokalen Optimierungen*, ausgeführt. Im Backend folgen weitere zielmaschinenspezifische Optimierungen, sowie die eigentliche Codegenerierung.

Die Aufgabe der lokalen Optimierungen ist es Ausdrücke anhand lokaler Information in eine kostengünstigere Form zu transformieren. Zum Beispiel ist die Berechnung[1] $a \wedge (a \oplus b)$ durch die Vorschrift $a \wedge \neg b$ effizienter[2] durchführbar. Das Optimierungsverfahren verwendet hierfür Regeln, welche auf Basis der Operation, ihrer Parameter[3] und potentiell weiterer zur Verfügung stehender Analyseinformation eine Umstrukturierung vornehmen. Neben der Einschränkung auf diese namensgebende lokale Information sind lokale Optimierungsregeln stets so entworfen bezüglich des bekannten Kontextes niemals höhere Realsierungskosten zu erzeugen.

Bedingt durch Mehrfachverwender[4] kann, aus globaler Sicht, dennoch eine insgesamte Erhöhung der Kosten stattfinden. Wird vom obigen Ausdruck $a \wedge (a \oplus b)$ der Teilausdruck $a \oplus b$ auch an anderer Stelle verwendet so kann trotz Optimierung die Realisierung, also die Berechnungsdurchführung, dieses Wertes nicht entfallen. Die Ausführung der Optimierung resultiert somit in der zusätzlichen Realisierung der Negation und damit insgesamt in höheren Kosten.

1.1 Zielsetzung

Im Rahmen dieser Ausarbeitung soll untersucht werden, wie geeignet das Optimierungsproblem PBQP ist um die Anwendung lokaler Optimierungsregeln mit einer globaleren Sichtweise zu kontrollieren. Dabei sollen speziell die negativen Effekte von Mehrfachverwendern, aber auch die positiven Effekte durch potentielle Wiederverwender berücksichtigt werden. Im Rahmen dieser Untersuchung soll eine eigene Optimierungsphase für LibFirm erzeugt werden, welche die erarbeiteten Konzepte umsetzt.

[1] $\wedge$: AND, $\oplus$: XOR, $\neg$: NOT
[2] Annahme: Negation günstiger realisierbar als XOR
[3] auch über mehrere Ebenen
[4] ein Ausdruck verwendet durch mehr als eine Operation

1.2 Aufbau

Den inhaltliche Einstieg der Ausarbeitung bildet der Abschnitt 2. Zunächst werden in Abschnitt 2.1 aktuelle Techniken zur Programmdarstellung in Compilern und in Abschnitt 2.2 das Konzept der lokalen Optimierung eingeführt. Zur späteren formalen Betrachtung der graphbasierten Zwischensprache Firm werden in Abschnitt 2.3 einige grundlegende Konzepte und Definitionen aus der Graphentheorie dargelegt. Den Abschluss bildet Abschnitt 2.4 mit der formalen Einführung des Optimierungsproblems PBQP.

In Abschnitt 3 beginnt die Betrachtung des Problems auf theoretischer Ebene. Dazu wird das Problem in Abschnitt 3 präziser formuliert und dann in Abschnitt 3.2 eine zur theoretische Betrachtung geeignete Formalisierung von Zwischensprachen und Transformationsmöglichkeiten eingeführt. Es folgt Abschnitt 3.3 mit der schrittweisen Entwicklung eines Konzeptes zur Lösung des dargestellten Problems.

Die Umsetzung des theoretischen Konzeptes wird in Abschnitt 4 besprochen. Hier werden zunächst in Abschnitt 4.1 wichtige Details der Implementierungsumgebung LibFirm betrachtet und darauf aufbauend der Implementierungsentwurf in Abschnitt 4.2 aufgezeigt. Die im Entwurf spezifizierten Phasen der Implementierung werden dann in den folgenden Abschnitten erläutert und wichtige technische Feinheiten der jeweiligen Phase erklärt.

Die Leistungsfähigkeit des theoretischen Konzepts und der Implementierung werden in Abschnitt 5 evaluiert und kritisch untersucht. Dazu wird in Abschnitt 5.1 die Performanz zur Übersetzungszeit und in Abschnitt 5.2 die Performanz zur Laufzeit des übersetzten Resultats betrachtet und diskutiert.

Den Abschluss bildet Abschnitt 6. Hier werden die Resultate der Ausarbeitung in eigenen Worten reflektiert und auf mögliche weitere Untersuchungsmöglichkeiten hingewiesen.

2 Grundlagen

Dieser Abschnitt führt die in dieser Ausarbeitung zum Verständnis notwendigen Grundlagen ein. In Abschnitt 2.1 wird zunächst die SSA-Form und darauf aufbauend die Zwischensprache Firm eingeführt. Es folgt in Abschnitt 2.2 das Konzept der lokalen Optimierungen, deren Ausführung im Rahmen dieser Ausarbeitung näher betrachtet werden wird. Zur formalen Betrachtung von Firm wichtige Konzepte aus der Graphentheorie werden in Abschnitt 2.3 aufgegriffen und anschließend das Optimierungsproblem PBQP in Abschnitt 2.4 definiert.

2.1 Programmdarstellung

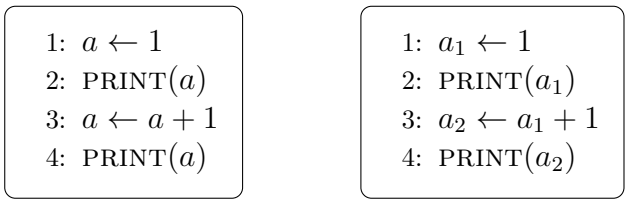
Compiler übersetzen Code von einer Quell- in eine Zielsprache. Dazu verwenden moderne Compiler eine von Quell- und Zielsprache unabhängige Zwischensprache[5] zur besseren internen Verarbeitung während des Übersetzungsvorgangs. Der Zwischencode[6] wird im quellsprachenspezifischen Frontend aus dem Quellcode gewonnen. Das entsprechende Backend für die Zielmaschine ist dann für die Synthese des Zielmaschinencodes aus dem Zwischencode zuständig. Die Zwischenrepräsentation[7] ist idealerweise zunächst vollkommen unabhängig von Quell- und Zielsprache. Dadurch ist es mit einem Middleend möglich verschiedene Quell- und Zielsprachen zu unterstützen. Jedoch gibt es auch Zwischensprachen (wie zum Beispiel Firm), welche sich im Übersetzungsvorgang der Zielsprache annähern[8] um zielmaschinenspezifische Optimierungen zusammen mit abstrakteren Optimierungen anwenden zu können.

Die im Folgeabschnitt Abschnitt 2.1.1 eingeführte SSA-Form ermöglicht eine effizente Repräsentation als Graph, welche Flussanalysen erheblich vereinfacht und dabei effizient implementierbar ist.

2.1.1 SSA

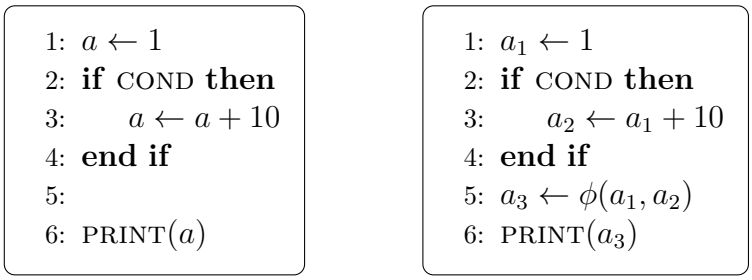
Die *statische Einmalzuweisung*[9] ist eine sehr häufig verwendete Darstellungsform für Zwischencode [1, 2]. Sie stellt sicher, dass die Relation zwischen Verwendungs- und Definitionsstelle einer Variablen immer eindeutig[10] ist. Durch diese Struktur ist es möglich

[5] engl. intermediate language
[6] engl. intermediate representation
[7] synonym zu Zwischencode
[8] die Annäherung erfolgt durch den Einbau zielsprachenspezifischer Elemente
[9] engl. static single assignment, SSA
[10] d.h. linkstotal und rechtseindeutig



(a) Eingabe **(b)** SSA-Form

Abbildung 1: Transformation in SSA-Form (Pseudocode)



(a) Eingabe **(b)** SSA-Form

Abbildung 2: Transformation in SSA-Form mit ϕ-Knoten (Pseudocode)

Zwischensprachen als Graph darzustellen, was Datenflussanalysen erheblich vereinfacht und damit auch komplexere Optimierungsverfahren in Übersetzern ermöglicht.

Bei der Transformation in SSA-Form wird für jede Zuweisung, also für jede Definitionsstelle, eine neue Variable angelegt, welche bis zur nächsten Definitionsstelle anstatt der ursprünglichen Variablen referenziert wird. Das Resultat ist, dass aus einer Variablen a eine Variablenmenge $\{ a_1, a_2, \dots \}$ entsteht, deren Mächtigkeit mindestens so groß wie die Anzahl der Zuweisungen ist.

Abbildung 1 zeigt ein einfaches Codefragment und dessen SSA-Form. Für jede Zuweisung (Zeilen 1 und 3) werden neue Variablen angelegt. Entsprechend werden die Verwendungen der jeweiligen Definition an die neuen Variablen angepasst. Für a_1 ist das der Funktionsaufruf in Zeile 2 und die Berechnung $a + 1$ in Zeile 3.

ϕ-Knoten. Eine weitere Quelle von Mehrdeutigkeiten sind kontrollflussabhängige Verwendungen wie sie in Schleifen oder bedingten Zuweisungen vorkommen. Dadurch kann eine Variable, je nach zuvor ausgeführtem Block, auf verschiedene Definitionsstellen verweisen. Zur Lösung dieses Problems wird die kontrollflussabhängige Selektion der richtigen Definitionsstelle über einen sogenannten ϕ-Knoten realisiert und das Ergebnis in einer weiteren Variablen abgelegt.

Abbildung 2 zeigt wie ϕ-Knoten eingesetzt werden um kontrollflussbedingte Mehrdeutigkeiten aufzulösen. Abhängig von der Bedingung in Zeile 2 kann sich das a in Zeile 6 auf Zeile 1 (a_1) oder Zeile 3 (a_2) beziehen. Hier wird nun ein ϕ-Knoten eingeführt, welcher je nach Kontrollflussverlauf die richtige Definition selektiert. Konkret heißt das, dass bei der Ausführung von Zeile 5 relevant ist, ob zuvor die Zeile 2 oder die Zeile 3 ausgeführt wurde. Entsprechend wird vom ϕ-Knoten a_1 beziehungsweise a_2 ausgewählt und zurückgegeben.

Die Information, welche Definition in welcher Situation gewählt werden soll wird hier nicht explizit dargestellt. Hierfür ist die Realisierung der Zwischensprache zuständig. Firm beispielsweise ordnet jeder Alternative eines ϕ-Knoten eine Kontrollflussabhängigkeit des zugehörigen Grundblocks zu.

2.1.2 Firm

Firm [3] ist eine graphbasierte Zwischensprache in SSA-Form. Die Kanten[11] werden hier eingesetzt um residual zum Fluss die Abhängigkeiten zwischen Knoten anzugeben. Ein Knoten repräsentiert immer das Resultat seiner inhärenten Operation angewandt auf seine Operanden. Bedingt durch die SSA-Form lassen sich lokale Variablen über Kanten und ϕ-Knoten modellieren. Nur im Speicher allokierte Variablen werden zusätzlich über monadische Lade- und Speicheroperationen modelliert, deren Ausführung durch den Speicherfluss einer Halbordnung unterworfen wird. Firm ist die Instanziierung eines expliziten Abhängigkeitsgraphs.

LibFirm. LibFirm ist die Implementierung von Firm in Form einer C-Bibiliothek. Zur Zeit wird diese vom IPD Snelting am KIT weiterentwickelt. Darin enthalten sind neben allgemeineren und Firm-spezifischen Datenstrukturen und Analyseverfahren auch eine Reihe von fertig implementierten Optimierungsverfahren. Darunter befindet sich ein großer handgeschriebener Regelsatz von lokalen Optimierungsregeln. Im Rahmen ihrer Diplomarbeit haben Buchwald and Zwinkau [4] dort auch eine Implementierung zweier PBQP-Löser eingepflegt.

Die Bibiliothek ist ein funktionsfähiges Middle- und Backend und bildet zusammen mit dem Frontend cparser[12] einen kompletten Compiler für C-Code. Zum Zeitpunkt dieser Ausarbeitung waren Backends für IA32, Sparc, ARM und AMD64 enthalten.

[11] Kontroll-, Speicher-, Datenkanten
[12] `http://pp.ipd.kit.edu/git/cparser.git`

2.2 Lokale Optimierung

Optimierungsverfahren lassen sich hinsichtlich ihrer Lokalität in zwei Klassen untergliedern. Während *globale* Optimierungen das Programm als Ganzes, oder zumindest große Teile davon, prozessieren, greifen die *lokalen* Optimierungen [5] namensgebend nur in einem sehr kleinen Kontext. Das heißt eine Operation und Teile ihrer transitiven Operanden werden betrachtet um *lokal* gesehen eine bessere[13] Realisierung zu finden.

Optimierungsregeln kapseln die Logik zur Analyse der vorliegenden Situation und zur Errechnung der, lokal gesehen, geschickteren Darstellung. Dabei können Optimierungsregeln bei ihrer Ausführung durchaus auf Informationen zurückgreifen, welche durch globale Analysemethoden erarbeitet wurden. Einige wichtige und in dieser Ausarbeitung relevante Ausprägungen von lokalen Optimierungen werden im Folgenden vorgestellt.

Konstantenfaltung. In der Regel können Operationen, welche ausschließlich auf Konstanten zugreifen schon zur Übersetzungszeit ausgewertet werden. Die Ersetzung einer solchen Operation durch den resultierenden konstanten Wert nennt sich *Konstantenfaltung*. Oft ermöglicht die Anwendung dieser Optimierungsform die Anwendung weiterer Optimierungsregeln. In dem Ausdruck $3 + 4$ besitzt die Addition ausschließlich konstante Operanden. Dies ermöglicht die Berechnung zur Übersetzungszeit und führt zur Ersetzung des kompletten Ausdrucks durch das Resultat $7 = 3 + 4$.

Operatorvereinfachung. Der funktionelle Zusammenhang zwischen Operationen lässt sich in bestimmten Situationen ausnutzen um Operationen durch günstigere Varianten zu ersetzen. Die Multiplikation $x * y$ beispielsweise beruht auf der x-maligen Addition von y zu 0. Das heißt der Ausdruck $2 * x$ ist äquivalent zu dem effizienter umsetzbaren Ausdruck $x + x$, welcher nun statt der ursprünglichen Version verwendet werden kann.

Algebraische Vereinfachungen. Algebraische Umformungen können Ausdrücke in vereinzelten Situationen in eine kostengünstigere Form bringen. Zum Beispiel sind binäre Operationen, welche ihr neutrales Element verwenden, immer durch den entsprechenden anderen Operanden ersetzbar. Konkret ist der Ausdruck $x + 0$ äquivalent zu x, weil 0 das neutrale Element der Addition darstellt.

[13] bezüglich eines Kostenmodells (z.B. Taktzyklen oder Ausführungsdauer)

2.3 Graphen

Dieser Unterabschnitt enthält einige grundlegende Definitionen aus den Bereichen Graphentheorie und Mengenlehre. Die Betrachtung graphbasierter Zwischensprachen benötigt diese für eine sinnvolle theoretische Behandlung. Um verschiedenartige Kantenmengen zur gleichen Knotenmenge und Verbindungen dieser geeignet formulieren zu können, wird weiterhin das Konzept der Kantenenmenge verallgemeinert und erweitert.

Definition 2.1 (Gerichtete Kantenmenge (mit Mehrfachkanten))**.** Zu einer Menge V heißt eine Menge $E \subseteq V \times V$ *gerichtete Kantenmenge* und ein 3-Tupel $(E, \mathrm{src}, \mathrm{trg})$ *gerichtete Kantenmenge mit Mehrfachkanten*, wobei die Abbildungen $\mathrm{src} : E \to V$ und $\mathrm{trg} : E \to V$ den Kanten respektiv ihre Quelle und ihr Ziel zuordnen.

Bemerkung 2.1. Eine gerichtete Kantenmenge $E \subseteq V \times V$ ohne Mehrfachkanten lässt sich kanonisch in eine gerichtete Kantenmenge mit Mehrfachkanten $\hat{E} = (E, \mathrm{src}, \mathrm{trg})$ übertragen:

$$\begin{aligned} \mathrm{src} : E &\to V \\ (v, w) &\mapsto v \\ \mathrm{trg} : E &\to V \\ (v, w) &\mapsto w \end{aligned}$$

Bemerkung 2.2. Die Operatoren $(\cup)$ und $(\in)$ lassen sich wie folgt auf gerichtete Kantenmengen mit Mehrfachkanten $\hat{E}_1 = (E_1, \mathrm{src}_1, \mathrm{trg}_1)$ und $\hat{E}_2 = (E_2, \mathrm{src}_2, \mathrm{trg}_2)$ übertragen:

$$e \in \hat{E}_1 :\Leftrightarrow e \in E_1$$

$$\hat{E}_1 \cup \hat{E}_2 \coloneqq (E_1 \cup E_2, \mathrm{src}, \mathrm{trg})$$

$$\mathrm{src} : e \mapsto \begin{cases} \mathrm{src}_1(e) & e \in E_1 \\ \mathrm{src}_2(e) & e \in E_2 \end{cases}$$

$$\mathrm{trg} : e \mapsto \begin{cases} \mathrm{trg}_1(e) & e \in E_1 \\ \mathrm{trg}_2(e) & e \in E_2 \end{cases}$$

Definition 2.2 (Gerichteter Graph)**.** Ein *gerichteter Graph* ist ein 2-Tupel $G = (V, E)$ mit der *Knotenmenge* V und der dazugehörigen Kantenmenge E. Ist E eine gerichtete Kantemenge mit Mehrfachkanten, so heißt G ein *gerichteter Graph mit Mehrfachkanten*.

Im Rahmen dieser Ausarbeitung ist mit einem gerichteten Graphen, sofern nicht explizit anderweitig ausgezeichnet, stets ein gerichteter Graph mit Mehrfachkanten gemeint.

Die folgenden Definitionen und Bemerkungen beziehen sich auf gerichtete Kantenmengen ohne Mehrfachkanten, sind aber analog für gerichtete Kantenmengen mit Mehrfachkanten zu verstehen. Als übertragender Formalismus kann die in Bemerkung 2.1 eingeführte Transformation verwendet werden.

Definition 2.3 (Pfad)**.** Sei E eine gerichtete Kantemenge zur Menge V. Für ein $n \in \mathbb{N}_+$ heißt die Kantenfolge $(e_1, \ldots, e_n) \in E^n$ *Pfad* der Länge n genau dann, wenn

$$\forall i \in \{1, \ldots, n-1\} : \mathrm{trg}(e_i) = \mathrm{src}(e_{i+1})$$

Für einen Graph (V, E) ist die Darstellung als Knotenfolge $p = (v_1, \ldots, v_{n+1}) \in V^{n+1}$ äquivalent. Der *Quellknoten* des Pfades sei mit $\mathrm{src}(p) = v_1$ und der *Zielknoten* sei mit $\mathrm{trg}(p) = v_{n+1}$ bezeichnet.

Definition 2.4 (Zyklus)**.** Sei E eine gerichtete Kantemenge. Ein Pfad $p \in E^n$ heißt *Zyklus* genau dann, wenn $\mathrm{src}(p) = \mathrm{trg}(p)$.

Definition 2.5 (Erreichbarkeit)**.** Sei E eine gerichtete Kantenmenge zur Menge V. Die folgende Funktion beschreibt alle von einem gegebenen Startelement durch Pfade mit maximaler Länge n erreichbare Knoten.

$$\begin{aligned}\mathrm{reach}_E^n : V &\to \mathcal{P}(V) \\ v &\mapsto \{\, w \in V \mid \exists m \leq n \colon \exists \text{ Pfad } p \in E^m \colon \mathrm{src}(p) = v \wedge \mathrm{trg}(p) = w \,\}\end{aligned}$$

Die von der Pfadlänge unabhängige Erreichbarkeit sei durch die Funktion $\mathrm{reach}_E \coloneqq \mathrm{reach}_E^\infty$ beschrieben. Für zwei Knoten $v, w \in V$ heißt w *erreichbar* von v genau dann, wenn $w \in \mathrm{reach}_G(v)$.

Definition 2.6 (Gewurzelter Graph)**.** Sei $G = (V, \hat{E})$ ein gerichteter Graph. Ein $\omega \in V$ heißt *Wurzel* von G genau dann, wenn $\mathrm{reach}_{\hat{E}}(\omega) \cup \{\, v \,\} = V$. Existiert für G eine Wurzel, so ist diese eindeutig und G heißt *gewurzelter* Graph.

Bemerkung 2.3. In dieser Definition der Wurzel wird nicht gefordert, dass sie von keinem anderen Knoten erreicht werden kann. Damit kann eine Wurzel auch Teil eines Zyklus oder einer starken Zusammenhangskomponente sein.

Bemerkung 2.4. Ein gewurzelter Graph $G = (V, \hat{E})$ ist *schwach zusammenhängend*. Nach Definition ist jedes $v \in V$ von der Wurzel $\omega \in V$ aus erreichbar. Damit lässt sich im zu G gehörigen ungerichteten Graphen G' ausgehend von v jedes $v' \in V$ erreichen indem die, nach der Wurzel-Eigenschaft existenten, Pfade $(v, \ldots, \omega)$ und $(\omega, \ldots, v')$ zu einem Pfad $(v, \ldots, \omega, \ldots, v')$ verknüpft werden.

Definition 2.7 (Teilgraph)**.** Sei (V, E) ein gerichteter Graph und $\omega \in V$ ein Knoten. Dann ist $V(E)_\omega \coloneqq \mathrm{reach}_E^\infty$ die durch ω induzierte Knotenmenge mit $V(E)_\omega \subseteq V$. Weiterhin bildet $(V(E)_\omega, E)$ einen Teilgraph von (V, E).

Definition 2.8 (Partitionierung)**.** Gegeben sei eine Menge M. Ein Mengensystem $\mathcal{Z} \subset \mathcal{P}(M)$ heißt *Partitionierung* von M genau dann, wenn $\mathcal{Z}$ nicht die leere Menge $\emptyset$ enthält, die Elemente von $\mathcal{Z}$ paarweise disjunkt sind und $\bigcup_{A \in \mathcal{Z}} A = M$ gilt. Zu einem Element $m \in M$ bezeichnet $\mathcal{Z}(m) \in \mathcal{Z}$ diejenige Partition, welche m enthält.

2.4 PBQP

Das *Partitioned Boolean Quadratic Problem* (kurz PBQP) ist ein spezielles *Quadratic Assignment Problem* [6]. Hier gilt es eine Reihe von untereinander abhängigen Entscheidungen so zu fällen, dass als Resultat die sinnvollste Gesamtentscheidung gefällt wird. Um die Sinnhaftigkeit einer Entscheidung und ihrer Konsequenzen zu quantifizieren wird ein Kostenmodell herangezogen. Dazu wird in diesem Abschnitt zunächst das verwendete Kostenmodell und anschließend das Problem formal eingeführt. Kostenmodell und Problemdefinition orientieren sich an Buchwald and Zwinkau [4].

Kostenmodell. Das PBQP ist ein *abstraktes* Optimierungsproblem und benötigt somit keine Information weder über die konkrete Gestalt von Entscheidungsmöglichkeiten noch über Ursache und Ausprägung von Abhängigkeiten zwischen Entscheidungen. Es genügt Entscheidungsmöglichkeiten hinsichtlich ihres Effekts auf das Gesamtsystem und die Konsequenzen ihrer Wahl bewerten zu können. Dazu wird ein Kostenmodell herangezogen, welches jeder Entscheidungsmöglichkeit und jeder Konsequenz *reellwertige* Kosten zuordnet. Um einzelne Entscheidungskombinationen verbieten zu können findet das Konzept *unendlicher* Kosten Anwendung. Als algebraische Struktur für das verwendete Kostenmodell wird der kommutative Monoid $(\mathbb{C}, +)$ auf $\mathbb{C} = \mathbb{R} \cup \{\infty\}$ definiert, wobei die kommutative Operation $(+)$ auf folgende Weise erweitert wird:

$$x + y = \begin{cases} x + y & x, y \in \mathbb{R} \\ \infty & \text{sonst} \end{cases}$$

Definition als Graph. Eine PBQP-Instanz I ist ein 4-Tupel $(V, E \subset V^2, \mathcal{A}, c)$ mit einem gerichteten Graphen (V, E), einer Abbildung[14] $\mathcal{A} = \{(p, A_p) \mid p \in V\}$ von V auf beliebige Mengen und einer Kostenfunktion c. Die Elemente von V modellieren die zu fällenden *Entscheidungen*. Diese werden im Folgenden auch als PBQP-*Knoten* bezeichnet. Jede Entscheidung $p \in V$ hat eine zugehörige Menge von *Alternativen* $\mathcal{A}(p) = A_p$. Diese modellieren die möglichen Resultate einer gegebenen Entscheidung. Zusätzlich werden jeder Alternative $a \in A_p$ Kosten $c(a) \in \mathbb{C}$ zugeordnet. Die Kanten $(p, q) \in E$ modellieren die Abhängigkeit zwischen zwei Entscheidungen p und q. Hierfür werden für jede Kante jeder Alternativkombination $(a_p, a_q) \in A_p \times A_q$ der Entscheidungen p und q mit Kosten $c((a_p, a_q)) \in \mathbb{C}$ versehen. Formal kann die hier generisch eingesetzte Kostenfunktion c als Funktionenfamilie aufgefasst werden:

$$\begin{aligned} c(a) &= c_p(a) & \forall p \in V : \forall a \in A_p \\ c((a_p, a_q)) &= c_{(p,q)}(a_p, a_q) & \forall (p, q) \in E : \forall (a_p, a_q) \in A_p \times A_q \end{aligned}$$

[14] mengentheoretische Schreibweise

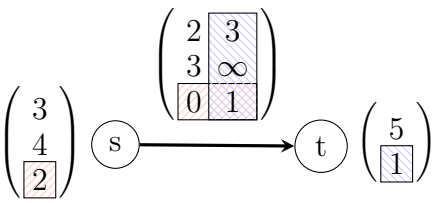


Abbildung 3: Eine einfache PBQP-Instanz mit optimaler Lösung

Eine *Auswahl* auf einer PBQP-Instanz $I = (V, E, \mathcal{A}, c)$ ist eine Funktion $\alpha : p \in V \mapsto a \in A_p$, welche jeder Entscheidung eine Alternative zuordnet. Dabei ergeben sich für die Auswahl α Gesamtkosten $c(\alpha)$ in Abhängigkeit der vorgenommenen Zuordnung und der Kostenfunktion c:

$$c(\alpha) \coloneqq \sum_{A \in V} c(\alpha(A)) + \sum_{(A,A') \in E} c((\alpha(A), \alpha(A')))$$

Eine Auswahl α auf I ist genau dann eine *Lösung* von I, wenn $c(\alpha) \in \mathbb{R} = \mathbb{C} \setminus \{ \infty \}$. Eine Lösung α ist genau dann *optimal* bezüglich I, wenn keine andere Lösung α' auf I mit $c(\alpha') < c(\alpha)$ existiert.

Zum Zwecke der Visualisierung und Implementierung werden die Entscheidungen einer Ordnung unterworfen und entsprechend dieser ein Kostenvektor bestehend aus den Alternativkosten aufgebaut. Analog werden die Kantenkosten über Kostenmatrizen dargestellt, wobei der Zeilenindex dem Vektorindex des Quellknotens und der Spaltenindex dem Vektorindex des Zielknotens entspricht. Auf diese Weise wird jeder Alternativkombination eine Zelle in der Matrix zugeordnet, welche die aufzuwendenden Kosten darstellt. Die Alternativen pro Knoten werden entweder nicht explizit dargestellt, oder als Knotenbeschriftung mit Kommata getrennt aufgeführt.

Abbildung 3 zeigt eine kleine PBQP-Instanz auf der eine optimale Auswahl α getroffen wurde. Für die Knoten s und t wurde jeweils die letzte Alternative ausgewählt. Das heißt es wurden respektiv die Vektorenindizes 3 und 2 selektiert, welche unter Beachtung der zugrunde liegenden Ordnung auf den Entscheidungen genutzt werden können um die selektierten Alternativen zu ermitteln. Die Kosten dieser Alternativen sind $c(\alpha(s)) = 2$ und $c(\alpha(t)) = 1$. Die Kosten der Kante folgen analog als $c((\alpha(s), \alpha(t))) = 1$. Für die Instanz ergeben sich damit die (endlichen) Kosten:

$$c(\alpha) = \underbrace{c(\alpha(s))}_{=2} + \underbrace{c(\alpha(t))}_{=1} + \underbrace{c((\alpha(s), \alpha(t)))}_{=1} = 4$$

Die Gesamtkosten $c(\alpha) = 4 \in \mathbb{R} = \mathbb{C} \setminus \{\infty\}$ der Auswahl sind endlich. Damit ist α eine Lösung auf I. Weiterhin lässt sich in dieser Instanz keine kostengünstigere Lösung α' finden, weswegen α damit ebenfalls eine optimale Lösung auf I ist.

Komplexität. Das Finden einer beliebigen Lösung ist in PBQP im Allgemeinen NP-vollständig [7]. Dabei ist vor allem die Zahl der Kanten beziehungsweise die Vernetzung des Graphen ausschlaggebend für den Lösungsaufwand.

In dieser Ausarbeitung werden zwei Lösungsverfahren behandelt, welche von Buchwald and Zwinkau [4] im Rahmen ihrer Diplomarbeit in LibFirm integriert wurden. Beide Löser arbeiten mit informationserhaltenden Reduktionsverfahren, welche bei geeigneter schwacher Vernetzung entweder einzelne Kanten oder zwei 2 aufeinanderfolgende Kanten und jeweils deren Quellknoten in die Zielknoten integrieren können. Der erste Löser verwendet eine Heuristik um den insgesamten Lösungsaufwand klein zu halten. Nachteil ist, dass wie schon beschrieben selbst das Finden einer Auswahl mit endlichen Kosten NP-vollständig ist und der heuristische Ansatz dementsprechend auch lösbare Instanzen als unlösbar erkennt. Der zweite Löser verfolgt einen Brute-Force-Ansatz und benötigt entsprechend der Komplexität von PBQP unter Umständen viel Zeit und Speicher.

3 Theoretische Betrachtung

Dieser Abschnitt betrachtet das Problem und potentielle Lösungsmöglichkeiten auf einer theoretischen Ebene. Dazu wird in Abschnitt 3.1 das behandelte Problem genauer erfasst und daraufhin verschiedene grundsätzliche Lösungsmöglichkeiten diskutiert. Um dann in Abschnitt 3.3 ein konkretes theoretisches Verfahren formulieren zu können, wird in Abschnitt 3.2 zunächst eine geeignete Formalisierung der verwendeten Konzepte vorgenommen.

3.1 Problemstellung

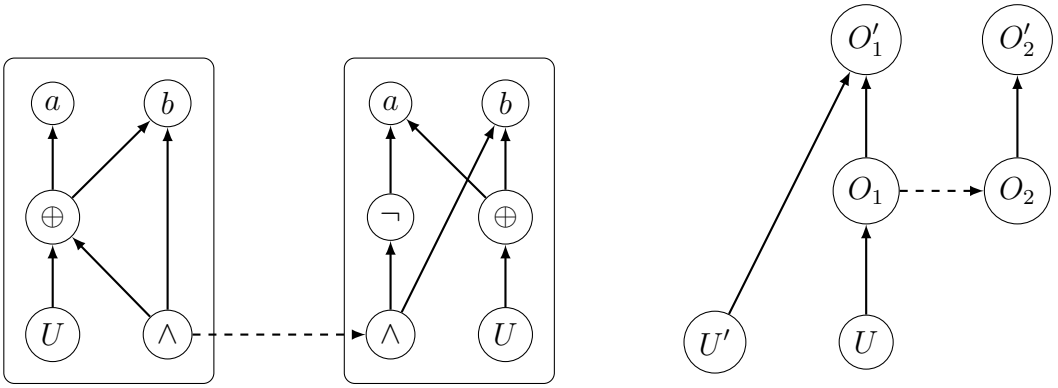
Bei der Anwendung lokaler Optimierungsregeln werden grundsätzlich nur ein Knoten und dessen Operanden betrachtet um zu entscheiden ob eine Umstrukturierung der Berechnung sinnvoll ist. Auch wenn die neu gewonnene Struktur lokal gesehen nicht teurer ist, so kann die Realisierung des kompletten Programms in Einzelfällen teurer werden.

Abbildung 4a illustriert anhand eines Beispiels wie eine Erhöhung der Gesamtkosten entstehen kann. Durch die Anwendung einer Optimierungsregel wird der linke Programmgraph in den neuen rechten Programmgraphen transformiert. Im Gegensatz zu anderen Abbildungen wird hier der Übergang zweiter kompletter Graphen gezeigt, wobei sonst im Folgenden nur mögliche Transformationen durch Kanten angedeutet werden.

Die linke Seite der Abbildung zeigt einen (vereinfachten) Programmgraphen, welcher die Berechnung $(a \oplus b) \wedge b$ enthält. Weiterhin gibt es hier eine nicht näher spezifizierte Operation U, welche ebenfalls auf das Berechnungsergebnis des Unterausdrucks $a \oplus b$ zugreift. Dieser Knoten drückt die Mehrfachverwendung des Teilausdrucks durch andere, nicht optimierbare Programmteile aus. Die Anwendung einer lokalen Optimierungsregel erlaubt die semantikerhaltende Ersetzung $(a \oplus b) \wedge b \mapsto \neg a \wedge b$. Die rechte Seite der Abbildung zeigt das Resultat der Anwendung. Man sieht leicht, dass die Gesamtkosten des Programms um die Kosten der Realisierung der Negierung gestiegen sind, ohne Kosten einer anderen Realisierung zu entfernen.

Abbildung 4b zeigt das problembehaftete Szenario in einer abstrakteren Darstellung. Der zusätzliche Verwender U' sorgt dafür, dass die Anwendung der Optimierungsregel $O_1 \mapsto O_2$ nicht die Realisierung von O_1' erspart und somit insgesamt höhere Kosten entstehen können.

Ansätze. Zur Behandlung des beschriebenen Problems ist es erforderlich verschiedene Ansätze abzuwägen. Zunächst ist es sinnvoll die Anwendung einer Optimierungsregel als



(a) Anwendungsbeispiel $(a \oplus b) \wedge b \mapsto \neg a \wedge b$ **(b)** Allgemeines Szenario

Abbildung 4: Lokale Optimierungsregeln können die Gesamtkosten erhöhen, wenn Unterausdrücke durch Mehrfachverwendung erhalten bleiben

zweischrittigen Prozess aufzufassen: Vorschlag einer Substitution durch die Regel und die Durchführung durch das aufrufende System.

Da Substitutionen neue Verwender erzeugen, aber auch alte vernichten können, ist es notwendig die Information über alle in Verbindung stehenden Substitutionsvorschläge zu aggregieren und dann erst zu evaluieren. Damit ist eine Behandlung innerhalb der Regeln durch die beschränkte Sichtweite und die konzeptionellen Schranken ausgeschlossen.

Es liegt nahe ein Verfahren zu entwickeln, welches als Ausführungsumgebung für Optimerungsregeln arbeitet und die Substitutionsdurchführungen geeignet steuert. Nun sind die Anwendungen lokaler Optimierungsregeln, neben expliziten Phasen für deren Anwendung, oft über das komplette Middleend, also über mehrere Optimierungsphasen, verteilt. In LibFirm findet sich die Anwendung lokaler Optimierungsregeln zum Beispiel schon direkt nach der Erzeugung eines jeden Knotens. Die Information über das gesamte Middleend zu sammeln gestaltet sich als sehr aufwendig, da jede Optimierungsphase je nach Programmkonstellation anders handeln könnte und so viele Konfigurationsmöglichkeiten gleichzeitig betrachtet werden müssen. Dies ist schon rein speichertechnisch sehr aufwendig und erfordert eine geeignete Darstellung und Filterung der weiterzutragenden Information.

Tate et al. [8] haben einen Ansatz entwickelt, der tatsächlich Transformationsinformation durch das komplette Middleend schleust und dabei nicht nur die Information der lokalen Optimierungen, sondern auch die anderer Effekte der Optimierungsphasen miteinbezieht. Dazu führen sie die Information in einer neuen Art von Zwischenrepräsentation zusammen, welche in der Lage ist eine Menge von möglichen Graphkonstellationen zu kodieren. Unter diesen Möglichkeiten wird am Ende der Übersetzung heuristisch eine Lösung ausgewählt und angewandt. Um die Datenkomplexität einzugrenzen, haben die Autoren eine Oberschranke eingebaut.

Ein andere Möglichkeit ist die phasenweise Auftrennung im Middleend. Es wird eine Optimierungsphase eingeführt, die speziell für die umsichtige Anwendung lokaler Optimierungsregeln verantwortlich ist. Andere Optimierungsphasen dürfen nur beschränkt Optimierungsregeln anwenden, so dass keine Konstellation gefährdet wird, die durch Mehrfachverwendung profitieren könnte. In regelmäßigen Schritten folgt die explizite Phase der lokalen Optimierungen, in welcher alle Regeln freigeschaltet und alle Substitutionsvorschläge gesammelt werden. Auf Basis dieser Information lässt sich eine Entscheidung treffen und darauf wieder mit eingeschränkten Optimierungsregeln bis zur nächsten Phase weiter verfahren.

Diese letzte Verfahrensform wird im Folgenden theoretisch hergeleitet und entwickelt. Dazu wird zunächst eine geeignete Formalisierung der verwendeten Strukturen durchgeführt.

3.2 Formalisierung

Definition 3.1 (Knotentypen)**.** Die Menge aller Knotentypen sei mit $\mathbb{T}$ und deren Untermenge mit zyklenbildenden Typen mit $\mathbb{T}_{\text{loop}}$ bezeichnet:

$$\mathbb{T} \coloneqq \{\, \texttt{Add}, \texttt{Sub}, \texttt{Phi} \ldots \,\}$$
$$\mathbb{T}_{\text{loop}} \coloneqq \{\, \texttt{Phi}, \texttt{Block} \,\} \subset \mathbb{T}$$

In Firm können Abhängigkeitszyklen nur über einzelne Knotentypen aus $\mathbb{T}_{\text{loop}}$ gebildet werden. Würde zum Beispiel ein `ADD` sich selbst verwenden, so könnte dieses nie ausführen, da der Wert seines Operanden nie bereitstehen würde.

Definition 3.2 (Realisierungskosten von Knotentypen)**.** Die Realisierungskosten von Knotentypen ist durch die Funktion $\text{rl} : \mathbb{T} \to \mathbb{N}_0$ definiert.

Definition 3.3 (Programmgraph, Abhängigkeitskante)**.** Ein *Programmgraph* G ist ein 5-Tupel $(N, \hat{D} = (D, \text{src}, \text{trg}), \text{t}, \text{pos}, \omega)$ mit einem gerichteten Graphen mit Mehrfachkanten $G_C = (N, \hat{D})$, einer Typisierungsfunktion $\text{t} : N \to \mathbb{T}$, einer Kantenordnung $\text{pos} : E \to \mathbb{N}$ und einem Knoten $\omega \in N$, welcher einen gewurzelten Teilgraphen induziert. Die Knoten N sind die *Programmknoten*, welche im Speziellen auch *Operationen* oder *Werte* genannt werden. Die Kanten $\hat{D}$ beziehungsweise D heißen *Abhängigkeitskanten*. Für alle Zyklen $(v_1, \ldots, v_n) \in N^n$ in $(N, \hat{D})$ existiert ein $i \in \{1 \ldots n\}$ mit $\text{t}(v_i) \in \mathbb{T}_{\text{loop}}$.

Programmgraphen werden in dieser Ausarbeitung eingesetzt um Firm-Graphen zu formalisieren. G_C repräsentiert hierbei alle möglichen Graphkonstellationen und die Wurzel ω selektiert einen konkreten Programmgraph $G_\omega = (N(\hat{D})_\omega, \hat{D})$ unter diesen.

Zyklen dürfen in FIRM-Graphen nur über zyklenbildende Knotentypen realisiert werden. Andere Konstruktionen sind hinsichtlich der Semantik sinnfrei und erlauben keine sinnvolle Umsetzung in funktionalen Maschinencode.

Im Rahmen dieser Ausarbeitung wird immer bezüglich eines eindeutigen und konstanten Programmgraphen G argumentiert. Demnach werden Objekte, welche von G selbst oder Komponenten von G abgeleitet werden, nicht parametriert.

Bemerkung 3.1 (Verwender, Operanden). Im Kontext von Abhängigkeitsgraphen mit einer Abhängigkeitskantenmenge D und der Knotenmenge N wird vereinfachend $\mathrm{op}(v)^n \mapsto \mathrm{reach}^n_{\hat{D}}(v)$ und $\mathrm{op}(v) \mapsto \mathrm{reach}^1_{\hat{D}}(v)$ eingesetzt. Ein $v \in N$ heißt *Verwender* von $w \in \mathrm{op}^n(v)$ und analog w *Operand* von v. Um einen direkten Zusammenhang $n = 1$ anzuzeigen kann von *direkten* Verwendern und Operanden und analog für $n > 1$ von *transitiven* Verwendern und Operanden gesprochen werden.

Bemerkung 3.2 (Geordnete Operanden). Durch die Abbildung pos unterliegen die ausgehenden Abhängigkeitskanten einer Ordnung. Mit der Abbildung $\mathrm{op}_{\mathrm{pos}} \colon \mathcal{P}(N \to \mathbb{N} \times N)$ werden alle direkten Operanden eines Knotens zusammen mit ihren Operandenindizes beschrieben:

$$\mathrm{op}_{\mathrm{pos}} \colon v \mapsto \{(\mathrm{pos}(e), \mathrm{trg}(e)) | e \in E \wedge \mathrm{src}(e) = v\}$$

Definition 3.4 (Realisierungskosten von Knoten)**.** Die *Realisierungskosten eines Knotens* sind über die Abbildungen t und rl definiert:

$$\mathrm{rl}_N : v \mapsto \mathrm{rl}(\mathrm{t}(v))$$

Die Kosten eines Knotens $\mathrm{rl}_N(v)$ entsprechen also den Kosten des Knotentyps. Damit haben die Parameter eine Operation keinen Einfluss auf der Kosten derselben. Die Kosten eines Subgraphs $\mathrm{rl}_D(v)$ ergeben sich durch die Summe aller Knotenkosten über den durch v induzierten Teilgraph.

Definition 3.5 (Knotensubstitution)**.** Eine *Knotensubstitution* bezüglich eines Programmgraphen $(N, (D, \mathrm{src}, \mathrm{trg}), \mathrm{t}, \mathrm{pos}, \omega)$ ist die Ersetzung eines Knotens $v \in N$ mit einem Knoten $v' \in N$ durch Redefinition der Zielfunktion trg:

$$\underset{v \mapsto v'}{\mathrm{subst}} : (N, (D, \mathrm{src}, \mathrm{trg}), \mathrm{t}, \mathrm{pos}, \omega) \mapsto (N, (D, \mathrm{src}, \mathrm{trg}'), \mathrm{t}, \mathrm{pos}, \omega')$$

$$\forall d \in D : \mathrm{trg}'(d) \coloneqq \begin{cases} v' & \mathrm{trg}(d) = v \\ \mathrm{trg}(d) & \text{sonst} \end{cases}$$

$$\omega' \coloneqq \begin{cases} v' & v = \omega \\ \omega & \text{sonst} \end{cases}$$

Bemerkung 3.3 (Knotensubstitution). Die Modifkation des Programmgraphens kann im Allgemeinen vermieden werden, in dem lediglich das Wurzelelement ω der Substitution entsprechend angepasst wird. Damit wird derjenige Subgraph selektiert, welcher sich gerade durch diese Substitution unterscheidet.

Definition 3.6 (Semantikerhaltende Substitution)**.** Eine Knotensubstitution ist genau dann *semantikerhaltend*, wenn die Semantik aller Graphen durch die Durchführung unverändert bleibt.

Definition 3.7 (Lokale Optimierungen)**.** Die *lokalen Optimierungsregeln* sind eine Regelmenge R mit Abbildungen $N \to N$. Für jede Regel $r \in R$ und jeden Knoten $v \in N$ muss gelten, dass für $v' = r(v)$ die Knotensubstitution von v mit v' semantikerhaltend ist und lokal gesehen die Kosten durch mit v' nicht größer werden. Eine Regel $r \in R$ kann genau dann *erfolgreich* auf $v \in N$ angewendet werden, wenn $r(v) \neq v$.

Die in Abbildung 4a angewandte Transformation kann nun als die Durchführung einer Substitution, vorgeschlagen durch eine Regel $r \in R$, aufgefasst werden. Der Wurzelknoten $v \in N$ zum Ausdruck $(a \oplus b) \wedge b$ mit $t(v) = \texttt{AND}$ soll mit dem Wurzelknoten $v' = r(v)$ des Ausdrucks $\neg a \wedge b$ mit ebenfalls $t(v') = \texttt{AND}$ ersetzt werden. Die Kosten des relevanten Subgraphen von v belaufen sich auf $\mathrm{rl}(\texttt{AND}) + \mathrm{rl}(\texttt{XOR})$ und die des Subgraphen von v' auf $\mathrm{rl}(\texttt{AND}) + \mathrm{rl}(\texttt{NEG})$. Da im Allgemeinen ein `NEG` effizienter zu realisieren ist als ein `XOR` ist mit einer Vergünstigung durch die Regelanwendung zu rechnen.

Definition 3.8 (Transformationskanten)**.** Eine Kante $(v, v') \in T \subseteq N^2$ heißt *Transformationskante*, wenn die Knotensubstitution von v mit v' semantikerhaltend ist.

Bemerkung 3.4. Die Regelmenge R induziert die Kantenmenge T aller[15] Transformationskanten durch folgende Vorschrift:

$$T \coloneqq \{(v, v') \mid v \in N, r \in R : v' = r(v) \wedge v \neq v'\}$$

3.3 Verfahrensentwicklung

In diesem Abschnitt werden Verfahren zur Herleitung einer geeigneten Reduktion auf das PBQP diskutiert. Zunächst werden die Entwurfskriterien erläutert und anschließend einige intuitive, aber naive Ansätze zum Widerspruch geführt.

Bei der Entwicklung eines Verfahrens zur Reduktion des Optimierungsproblems auf PBQP und zur anschließenden Rekonstruktion der Lösung müssen zwei wichtige Entwurfskriterien beachtet werden:

1. Güte der Lösung
2. Komplexität von PBQP

[15] aus lokalen Optimierungsregeln herleitbaren

Das erste Kriterium erfordert, dass eine Lösung der PBQP-Instanz eine möglichst optimale Lösung des Substitutionsproblems liefert. Hierfür muss hinreichend viel Information in den Aufbau der PBQP-Instanz fließen, was wiederum den Lösungsaufwand erhöht. Das zweite Kriterium erfordert die Komplexität von PBQP zu berücksichtigen und damit den Lösungsaufwand gering zu halten. Für das PBQP heißt das, dass die Kantenzahl möglichst gering gehalten werden muss. Die beiden Entwurfsziele sind konträr zueinander und müssen dementsprechend in einem geeigneten Gleichgewicht gehalten werden.

3.3.1 Reduktion auf PBQP

In diesem Abschnitt wird ein Reduktionsverfahren hergeleitet, welches es erlaubt einen Programmgraphen $(N(\hat{E})_\omega, \hat{E})$ mit einer Menge semantikerhaltender Substitutionsmöglichkeiten T so auf eine PBQP-Instanz abzubilden, dass eine Lösung der Instanz eine optimale Konfigurationsmöglichkeit des Programmgraphen bezüglich der Substitutionsmöglichkeiten impliziert.

Dazu wird der Programmgraph mit der Transformationskantenmenge T zu $(N(\hat{E} \cup \hat{T})_\omega, \hat{E})$ erweitert. Dieser Graph enthält zwei Arten von Kanten, Abhängigkeitskanten und Transformationskanten, und alle Knoten, die ausgehend von ω über diese Kanten erreichbar sind. Somit enthält er die Information des ursprünglichen Programmgraphen und alle durch Transformationskanten hergeleiteten Variationsmöglichkeiten. Ein Beispiel für solch einen Graphen ist in Abbildung 5a zu sehen.

Naiver Ansatz. Eine Kante $(v, v') \in T$ repräsentiert eine semantikerhaltende Substitutionsmöglichkeit von v mit v'. Folglich müssen v und v' beide den gleichen Wert berechnen. Dies motiviert die Auffassung von v und v' und aller anderen durch Transformationskanten erreichbaren Knoten als Äquivalenzklasse bezüglich des berechneten Wertes. Formal wird die Knotenmenge $N(\hat{E} \cup \hat{T})_\omega$ durch die ungerichtete Interpretation von T in Zusammenhangskomponenten partitioniert:

$$\mathcal{Z} \coloneqq \{Z \mid Z \text{ ZHK in } (N(\hat{E} \cup \hat{T})_\omega, \{\{v, v'\} \mid (v, v') \in T\})\}$$

Ein Beispiel für einen Graph $(N(\hat{E})_\omega, \hat{E})$ zusammen dem entsprechenden $\mathcal{Z}$ ist in Abbildung 5b zu sehen. Interpretiert man eine solche Zusammenhangskomponente nun als Menge semantisch äquivalenter aber realisierungstechnisch und kostentechnisch verschiedener Alternativen aus welcher die kosteneffizienteste zu wählen ist, eignet sich ein PBQP-Knoten zur Modellierung. Dem zu Grunde liegt die Entwurfsentscheidung, dass immer nur ein Knoten pro Partition realisiert werden muss und alle anderen Partitionselemente mit diesem einen ersetzt werden müssen. Im Folgenden wird die formale

Konstruktion der PBQP-Instanz $I = (V, E, \mathcal{A}, c)$ dargestellt:

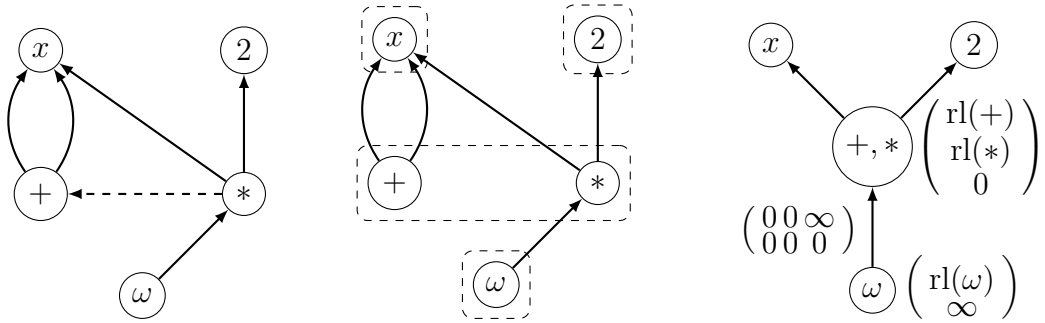
$$
\begin{aligned}
V &:= \mathcal{Z} \\
E &:= \{(\mathcal{Z}_v, \mathcal{Z}_w) \mid \mathcal{Z}_v, \mathcal{Z}_w \in \mathcal{Z} = V : w \in \mathrm{op}(v)\} \\
\mathcal{A} &:= \{(Z, A_Z := Z \cup \{\bot_Z\}) \mid Z \in V\} \\
c_Z(a) &:= \begin{cases} \begin{cases} \infty & \omega \in Z \\ 0 & \text{sonst} \end{cases} & a = \bot_Z \\ \mathrm{rl}_N(a) & \text{sonst} \end{cases} && Z \in V, a \in A_Z \\
c_{(Z,Z')}(a, a') &:= \begin{cases} \infty & a' = \bot_{Z'} \wedge a' \in \mathrm{op}(a) \\ 0 & \text{sonst} \end{cases} && (Z, Z') \in E, a \in A_Z, a' \in A_{Z'}
\end{aligned}
$$

Eine Partition $Z \in \mathcal{Z}$ wird mit allen Elementen in die Alternativen des entsprechenden PBQP-Knoten übertragen. Da im Allgemeinen einige Partitionen gar nicht realisiert werden müssen, wird zusätzlich eine VOID-Alternative $\bot_Z$ eingeführt, welche die Nicht-Realisierung modelliert. Die entsprechenden Kosten ergeben sich aus den Realisierungskosten der einzelnen Knoten außer bei der VOID-Alternative. Hier sind die Kosten 0 oder ∞. Unendliche Kosten ergeben sich für die VOID-Alternative nur in dem Sonderfall, dass ω in der entsprechenden Partition enthalten ist. Dieser Zusammenhang wird im Folgenden im Rahmen der Kantenkonstruktion erläutert.

Zwei PBQP-Knoten werden dann über eine Kante verbunden, wenn in der Quellpartition ein Knoten existiert, welcher einen Knoten der Zielpartition verwendet. Für jeden solchen Knoten aus der Quellpartition werden dann in der Kante für die VOID-Alternative der Zielpartition unendliche Kosten gesetzt. Damit wird erzielt, dass schrittweise, ausgehend von einer Quellpartition die realisiert werden muss, die Realisierung eines kompletten Subgraphen erzwungen wird. Die initial zu realisierende Quellpartition ist, wie in der Knotenkonstruktion schon angedeutet, diejenige Partition, die die Wurzel ω enthält. Hierdurch ist ebenfalls sichergestellt, dass es mindestens eine Auswahl mit endlichen Kosten gibt: Der ursprüngliche durch ω induzierte Programmgraph.

Abbildung 5 illustriert die drei Teilschritte der Konstruktion. In (a) ist der Programmgraph dargestellt, in welchem eine Optimierungsregel eine Substitutionsmöglichkeit aufdecken konnte. Die Multiplikation mit 2 lässt sich durch die kostengünstigere Addition mit sich selbst realisieren. Die Kante wird daraufhin in (b) als Verschmelzung der Partitionen aufgefasst und bildet somit eine Partition bestehend aus dem Additions- und dem Multiplikationsknoten. Alle anderen Knoten liegen nur in ihrer eigenen 1-elementigen Partition. In (c) ist die nach der formalen Konstruktion resultierende PBQP-Instanz dargestellt.

Abbildung 6 zeigt ein Problem bei dieser Interpretation der Transformationskanten. Die hier angewandte Optimierungsregel ist eine algebraische Vereinfachung, welche die Addition mit dem neutralen Element 0 erkennt und entsprechend den anderen Operanden

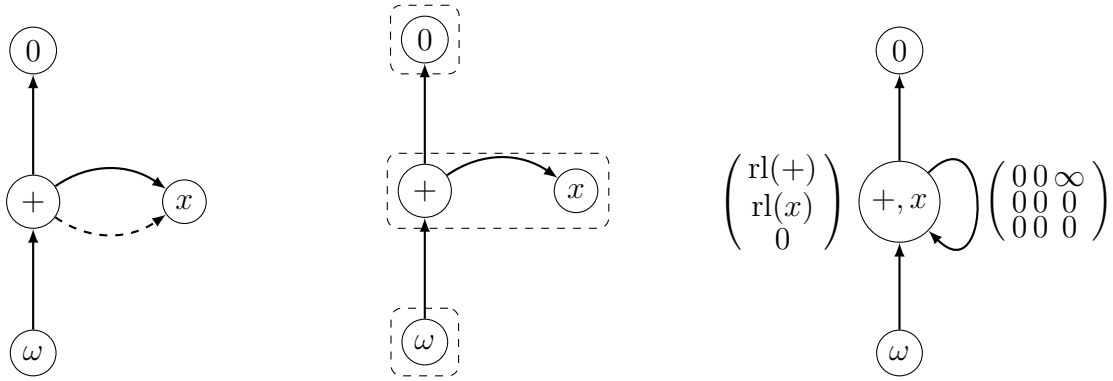


(a) Programmgraph mit Transformationskanten

(b) Programmgraph mit Partitionierung

(c) PBQP-Instanz

Abbildung 5: Schrittweise Konstruktion einer PBQP-Instanz über die ungerichtete Interpretation von Transformationskanten



(a) Programmgraph mit Transformationskanten

(b) Programmgraph mit Partitionierung

(c) PBQP-Instanz

Abbildung 6: Entstehung eines Abhängigkeitszyklus durch die ungerichtete Interpretation einer Transformationskante

zurückliefert. Die Partitionierung erzeugt, wie in (b) dargestellt, eine Partition mit einer innerhalb verlaufenden Abhängigkeitskante. Dem Modellierungskonzept zufolge resultiert die Auswahl eines Knotens in der Substitution aller anderen Knoten der selben Partition mit dem ausgewählten. Wie sich schon in (c) erkennen lässt, führt dieses Konstrukt bei der Auswahl der Addition zu einer Abhängigkeitsschleife. Da x einen beliebig großen Subgraphen repräsentiert sind dessen Kosten in der Regel deutlich größer als die einer Addition nebst einer Konstante. Dieser Sachverhalt lässt sich bei allen Abhängigkeitszyklen beobachten und führt zu dem Schluss, dass die Existenz eines potentiellen Abhängigkeitszyklus in der Regel zu dessen Realisierung führt. Abhängigkeitszyklen ohne dediziert dafür vorgesehene Knotentypen aus $\mathbb{T}_{\text{loop}}$ haben keine Semantik und sind dementsprechend als Konstrukt ungültig.

Das Grundproblem liegt in der semantisch ungültigen Reversiblität mancher Transformationskanten. Die naheliegende Lösung die Partitionen an solch einer Kante zu trennen funktioniert, würde aber wichtige Information bei der Konstruktion der PBQP-Instanz ignorieren.

Fortgeschrittener Ansatz: Delegations-Alternative Kanten werden im PBQP eingesetzt um Abhängigkeitskanten im Programmgraphen zu repräsentieren. Dieses Prinzip lässt sich auf Transformationskanten übertragen. Wie eine Realisierungsalternative über eine Kante die Realisierung des Operandenknotens erzwingt, so kann eine *Delegations-Alternative* über eine Kante die Realisierung des Zielknotens erzwingen. Der einzige Unterschied liegt darin, dass Delegations-Alternativen keine Kosten haben und damit bei ihrer Auswahl einfach ihre Realisierung an einen anderen PBQP-Knoten delegieren.

Um das PBQP nicht mit zu vielen Kanten zu überlasten und die Ausdrucksfähigkeit der Knotenalternativen weiterhin auszunutzen wird ein Kompromiss zwischen Graphpartitionierung und Transformationskanten im PBQP eingesetzt. Zunächst kann eine Partitionierung anhand der starken Zusammenhangskomponenten im gerichteten Transformationsgraphen durchgeführt werden:

$$\mathcal{Z} \coloneqq \{Z \mid Z \text{ SZK in } (N(\hat{D} \cup \hat{T})_\omega, T)\}$$
$$T' \coloneqq \{(v, v') \in T \mid \mathcal{Z}_v \neq \mathcal{Z}_{v'}\}$$

Ein Beispiel für diese Partitionierungsform ist in Abbildung 7c abgebildet. Diese Partitionierung ist unproblematisch, da sie sich nur auf Invarianten stützt, die durch die lokalen Optimierungsregeln sichergestellt werden. Bei den verbleibenden Kanten ist es fraglich ob eine ungerichtete Interpretation, also die Einführung einer impliziten *Rückwärtskante*, semantisch ungültige Konstrukte begünstigt. Fortan wird angenommen, dass die Gültigkeit einer Rückwärtskante allein von der Fähigkeit abhängt Abhängigkeitszyklen auszuprägen.

Durch eine Abbildungskette lässt sich für jede einzelne fragliche Kante genau dann eine

zusätzliche Rückwärtskante einführen, wenn dadurch kein Abhängigkeitszyklus ausbildbar ist. Eine anschließende Neupartitionierung stellt sicher, dass eine jede eingeführte Rückwärtskante zu einer Verschmelzung der verbundenen Partitionen führt. Insgesamt wird damit die Kantenzahl weiter reduziert und dafür Partitionsverschmelzungen vorgenommen. Im Folgenden wird dieses Verfahren formalisiert:

$$T_{\text{rev}} \coloneqq \left(\bigcirc_{e \in T} \text{rev}_e \right) (T')$$

$$\text{rev}_{(v,v')} : T \mapsto \begin{cases} T \cup \{(v', v)\} & \forall p = (e_1, \ldots, e_n) \text{ Pfad in } \hat{D} \cup \hat{T} \text{ von } v \text{ nach } v' : \\ & \forall i \in \{1, \ldots, n\} : e_i \in T \\ T & \text{sonst} \end{cases}$$

Die Abbildungen $\text{rev}_{(v,v')}$ versuchen jeweils in das gegebene T die Gegenkante (v', v) einzufügen. Würde dabei ein potentieller Abhängigkeitszyklus entstehen wird das Einfügen unterlassen. In beiden Fällen ist das Resultat die modifizierte oder ursprüngliche Transformationskantenmenge. Über Funktionskomposition lässt sich diese Funktionenfamilie für alle Transformationskanten zu einer Funktion $\bigcirc_{e \in T} \text{rev}_e$ kombinieren.

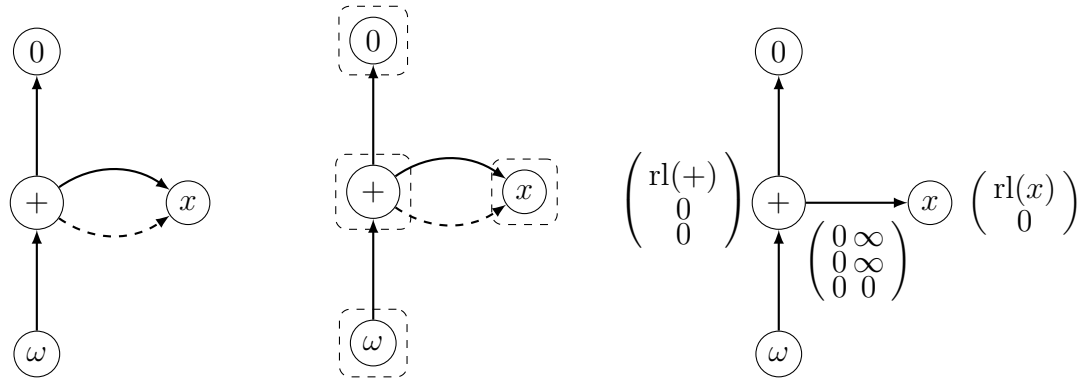
Eine Rückwärtskante kann genau dann keinen Abhängigkeitszyklus ausbilden, wenn ihre Einführung keinen Zyklus bestehend aus mindestens einer Abhängigkeitskante und beliebig vielen Transformationskanten erzeugt. Damit dies eintritt muss vorher schon ein Pfad in Richtung der ursprünglichen Transformationskante mit mindestens einer Abhängigkeitskante existiert haben. Wird so ein Pfad nicht gefunden, ist das Einfügen der Kante sicher. Die Repartitionierung und die damit verbundene Reduktion der Kantenmenge sei wie folgt bezeichnet:

$$\mathcal{Z}' \coloneqq \{Z \mid Z \text{ SZK in } (N(\hat{D} \cup \hat{T})_\omega, T \cup T_{\text{rev}})\}$$
$$T'_{\text{rev}} \coloneqq \{(v, v') \in T \mid \mathcal{Z}'_v \neq \mathcal{Z}'_{v'}\}$$

Aus $\mathcal{Z}'$, D und ω lässt sich nach bekannter Vorschrift eine PBQP-Instanz $I = (V, E, \mathcal{A}, c)$ erzeugen, welche nun im Folgenden um Delegations-Alternativen hergeleitet aus T'_{rev} zu $I' = (V, E', \mathcal{A}', c')$ erweitert wird:

$$E' \coloneqq E \cup \{(\mathcal{Z}_v, \mathcal{Z}_{v'}) \mid (v, v') \in T'_{\text{rev}}\}$$
$$\mathcal{A}' \coloneqq \{A'_Z = A_Z \cup \{(Z, Z') \in E' \setminus E\} \mid A_Z \in \mathcal{A}\}$$
$$c'_Z(a) \coloneqq \begin{cases} 0 & a \in E' \setminus E \\ c_Z(a) & \text{sonst} \end{cases} \qquad Z \in V, a \in A'_Z$$
$$c'_{(Z,Z')}(a, a') \coloneqq \begin{cases} \infty & a' = \bot_{Z'} \wedge a = (Z, Z') \\ c_{(Z,Z')}(a, a') & \text{sonst} \end{cases} \qquad (Z, Z') \in E', a \in Z, a' \in Z'$$

Die Kantenmenge des PBQP wird um die verbleibenden Transformationskanten erweitert, wobei diese nun als Transformationskanten zwischen Partitionen interpretiert werden. Die Alternativen zu einem PBQP-Knoten enthalten nun weiterhin eine Delegations-Alternative pro ausgehender Transformationskante der entsprechenden Partition. Die



(a) Programmgraph mit Transformationskanten

(b) Programmgraph mit Partitionierung und Transformationskanten

(c) PBQP-Instanz

Abbildung 7: Konstruktion einer PBQP-Instanz über gerichtete Transformationskanten

Kosten dieser Delegations-Alternativen werden immer auf 0 gesetzt und in der entsprechenden Kante setzen sie die Kosten für die VOID-Alternative des Zielknotens auf ∞ um dort eine Realisierung oder eine weitere Delegierung zu erzingen.

Abbildung 7 illustriert die Lösung der Problemsituation aus Abbildung 6. Die beiden Knoten wurden hier nicht zusammengefasst und die Information der Transformationskante zwischen ihnen wurde zusätzlich in die PBQP-Instanz (c) übertragen. Hieraus ergibt sich die 2. Alternative des Additions-Knotens, welche die Delegations-Alternative zur Transformation in x darstellt. Dazu wurden die Kosten der Kante bei Auswahl der Delegations-Alternative für die VOID-Alternative des Zielknotens auf unendlich gesetzt, um wie oben beschrieben die Realisierung beziehungsweise Redelegation dieses Knotens zu erzwingen.

Rücktransformation. Eine Lösung α zu $I' = (V, E', \mathcal{A}', c')$ kann nun verwendet werden um eine optimale Konfiguration des Programmgraphen herzuleiten. Für jede Partition $Z \in V$ kann die entsprechende Realisierung $k_Z^\alpha \in N \cup \{\bot\}$ über folgende Vorschrift hergeleitet werden:

$$k_Z^\alpha := \begin{cases} \alpha(Z) & \alpha(Z) \in Z \\ k_{Z'}^\alpha & \alpha(Z) = (Z, Z') \\ \bot & \alpha(Z) = \bot_Z \end{cases} \qquad Z \in V$$

Das Resultat $\bot$ beschreibt dabei die nicht notwendige Realisierung dieser Partition. Durch Konstruktion der PBQP-Instanz gelten zwei notwendige Konsistenzbedingungen

für eine Rekonstruktion.

$$\omega \in \mathcal{Z}_\omega \in V \Rightarrow k^\alpha_{\mathcal{Z}_\omega} \neq \bot \tag{1}$$
$$\forall Z \in V : k^\alpha_Z \neq \bot \Rightarrow \forall v \in \mathrm{op}(k^\alpha_Z) : k_{\mathcal{Z}_v} \neq \bot \tag{2}$$

Gleichung 1 gilt, weil die VOID-Alternative derjenigen Partition, die ω enthält explizit auf unendliche Kosten gesetzt wurde und Gleichung 2 gilt, weil jede Realisierungs-Alternative die VOID-Alternative des Zielknotens verbietet.

Die folgende Funktionskomposition modelliert nun die Anwendung der ausgewählten Substitutionen.

$$\mathrm{restr}(V, \alpha) \mapsto \left(\bigcirc_{v \in Z \in V} \mathrm{subst}_{\, v \mapsto k^\alpha_Z} \right)$$

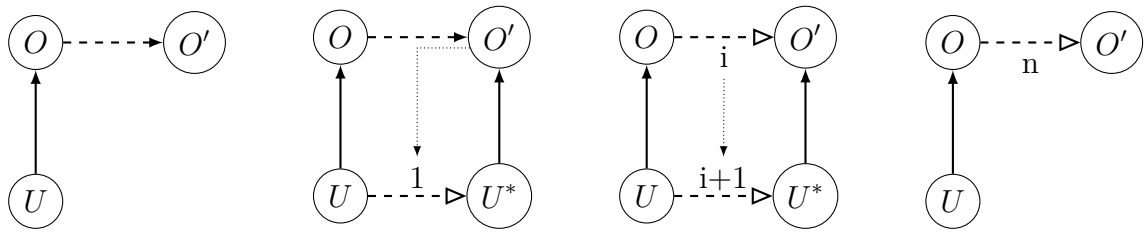
3.3.2 Ableitung von Transformationskanten

Im vorigen Abschnitt wurde ein Verfahren eingeführt, welches einen Programmgraphen und eine Transformationskantenmenge T geeignet in eine PBQP-Instanz überführt und wieder zurücktransformiert. In diesem Abschnitt wird nun die Herleitung der Transformationskantenmenge T diskutiert.

Zunächst gilt es aus dem betrachteten Programmgraphen $G_\omega = (N_\omega, \hat{D})$ möglichst viele Transformationskanten T aus der gegebenen Regelmenge R herzuleiten. Diese repräsentieren die von einander abhängigen Möglichkeiten den vorliegenden Graphen zu variieren und sollen durch das PBQP gegeneinander abgewogen werden.

Grundlegender Ansatz. Das bereits in Bemerkung 3.4 formell vorgestellte Verfahren eignet sich als grundlegender Ansatz für die Ableitung der Transformationskanten. Die Kantenmenge T leitet sich durch die Anwendung jeder Regel $r \in R$ auf jeden Knoten $v \in N$ her. Auf Anwendung der hergeleiteten Substitutionsmöglichkeiten wird bewusst verzichtet, da eine Graphveränderung ausschließlich in der Auswertungphase geschehen soll. Damit können auch explizit keine Kanten aus Regelapplikationen erhalten werden, welche erst nach Durchführung einer Operandensubstitution erfolgreich wären.

Erweiterung: Substitutionskanten. Der beschriebene grundlegende Ansatz der Kantenableitung ignoriert vollständig Regelanwendungen, welche erst nach einer Operandensubstitution erfolgreich wären. Ein klassischer Fall hierfür sind Konstantenfaltungen über mehrere Ebenen. Der Ausdruck $(1 + 2) + 3$ könnte durch die eingeschränkte Ableitung maximal zu $3 + 3$ optimiert werden, da hier eine erneute Regelanwendung nötig wäre um $3 + 3 \mapsto 6$ zu erkennen.



(a) Ausgangssituation: $O \mapsto O'$ **(b)** Initiale Substitutionskante **(c)** Propagierte Substitutionskante **(d)** Ende der Propagation

Abbildung 8: Propagierung von Substitutionskanten

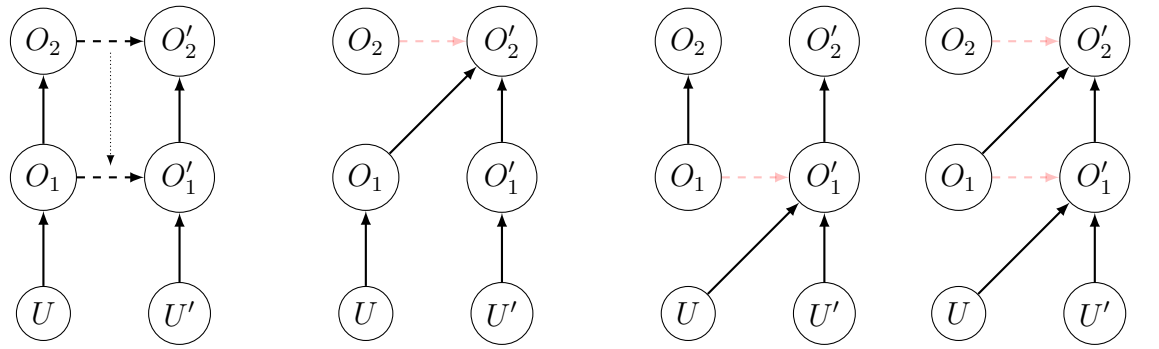
Es erweist sich als notwendig die Operandensubstitutionen in irgendeiner Form tatsächlich durchzuführen um damit weitere Optimierungsmöglichkeiten aufzudecken. Hierbei muss berücksichtigt werden, dass jede (auch transitive) Operandenkonstellation eine Regelanwendung ermöglichen könnte. Damit ist es von Nöten bis zu einer gewissen Operandentiefe n alle Operandenkonstellationen durchzuprobieren um alle Information aus den Regeln zu akquirieren. Die zu betrachtende Operandentiefe leitet sich dabei von der Optimierungsregel mit der weitestgehenden transitiven Operandenberücksichtigung ab. Im Folgenden sei n als gegebene feste natürliche Zahl relativ zur Regelmenge R angenommen.

Formal wird nun der Programmgraph $(N(\hat{D})_\omega, \hat{D})$ mit den regulär extrahierten Transformationskanten T für ein $n \in \mathbb{N}_{\geq 0}$ um eine weitere Kantenmenge $C_T^n \subseteq N^2$ erweitert:

$$
\begin{aligned}
C_T^0 &\coloneqq T \\
C_T^n &\coloneqq \left\{ (v,w) \;\middle|\; \begin{array}{l} \exists v, w \in N, (o,o') \in C_T^{n-1}, i \in \mathbb{N} : t(v) = t(w) \\ \wedge (i,o) \in \mathrm{op}_{\mathrm{pos}}(v) \wedge (i,o') \in \mathrm{op}_{\mathrm{pos}}(w) \\ \wedge \mathrm{op}_{\mathrm{pos}}(v) \setminus \{(i,o)\} = \mathrm{op}_{\mathrm{pos}}(w) \setminus \{(i,o')\} \end{array} \right\} \\
C_T^{n+} &\coloneqq \bigcup_{i \in \{1 \ldots n\}} C_T^i \\
C_T^{n*} &\coloneqq C_T^{n+} \cup T = \bigcup_{i \in \{0 \ldots n\}} C_T^i
\end{aligned}
$$

Eine Kante in $(v,w) \in C_T^n$ modelliert das eine Transformationskante in T existiert, deren Substitution v in w überführen würde. Dabei ist der zu ersetzende Operand von v genau n Operanden entfernt, also für $n = 1$ ein direkter Operand. Der Parameter n wird fortan auch als *Propagationsindex* bezeichnet. Dafür müssen beide Knoten den gleichen Typ und bis auf den substituierten Parameter die gleichen Operanden haben. Weiterhin müssen die Operanden bei beiden Operationen der gleichen Ordnung unterliegen. Im Folgenden werden diese Kanten *Substitutionskanten* genannt.

Abbildung 8 illustriert das Konzept der Substitutionskanten. Aus einer Transformati-



(a) Ausgangssituation **(b)** Auswahl $O_2 \mapsto O'_2$ **(c)** Auswahl $O_1 \mapsto O'_1$ **(d)** Auswahl beide

Abbildung 9: Partiell redundante Auswahlmöglichkeiten durch Hinzunahme von Substitutionskanten

on $O \mapsto O'$, dargestellt in (a), lässt sich eine Operandensubstition an die[16] Verwender weiterleiten. Dies wird in (b) dargestellt. Die Substitutionskante wurde mit dem Tiefenindex 1 parametriert, weil sie sich aus einer einem Operanden mit Entfernung 1 herleitet. Damit ist sie ein Element von C_T^1. Die weiterführende Propagation, ausgehend von Substitutionskanten, wird in in (c) gezeigt. Bei jeder Verwenderebene wächst der Propagationsindex um 1. Hier zeigt sich auch die rekursive Mengenbeziehung: Ein Element aus C_T^{i+1} leitet sich immer aus einem Element aus C_T^i her und mündet schließlich bei der entsprechenden initialen Transformationskante aus T. Das Ende der Propagation wird in (d) dargestellt. Hier wurde der maximale Propagationsindex n erreicht, wie er durch die Kantenmenge C_T^{n+} vorgegeben wurde.

Die Knotenmenge $N(\hat{D} \cup \hat{T} \cup \hat{C}_T^{n+})_\omega = N(\hat{D} \cup \hat{C}_T^{n*})_\omega$ repräsentiert alle durch lokale Optimierungsregeln herleitbare Modifikationen der initialen Knotenmenge $N(\hat{D})_\omega$. Mit einer geeigneten Interpretation der Substitutionskanten bei der Konstruktion der PBQP-Instanz ist die optimale Lösung der Instanz äquivalent zu einer global optimalen[17] Konfiguration des Programmgraphen.

Interpretation: Substitutionskanten als Transformationskanten. Aufgrund des semantischen Zusammenhangs zwischen Transformations- und Substitutionskante liegt es nahe im PBQP die Substitutionskanten ebenfalls als Transformationskanten aufzufassen. Demnach genügt es die zur Konstruktion verwendete Transformationskantenmenge zu $T \subseteq C_T^*$ zu erweitern. Dieser intuitive aber naive Interpretationsansatz führt zu subtilen Problemen.

Abbildung 9 zeigt wie die Interpretation einer Substitutionskante als Transformations-

[16] hier nur einer, im Allgemeinen mehrere
[17] relativ zu Programmgraph und Optimierungsregeln

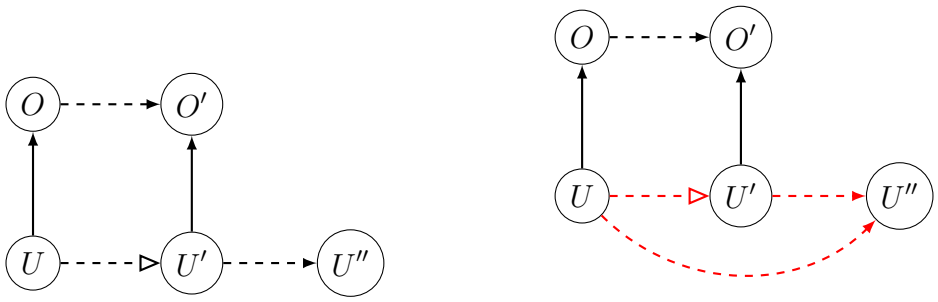
kante aus einer Entscheidungsmöglichkeit vier partiell redundante Entscheidungsmöglichkeiten generiert. In (a) ist die Ausgangssituation dargestellt. Die Transformationskante (O_2, O_2') wurde an den Verwender O_1 propagiert und resultiert dort in einer, hier als Transformationskante interpretierten, Substitutionskante (O_1, O_1'). Die Knoten O_1 und O_2 haben in der korrespondierenden PBQP-Instanz jeweils, neben der VOID-Alternative, eine Realisierungs-Alternative und eine Delegations-Alternative. Jede Auswahlmöglichkeit, bei der mindestens eine der beiden Delegations-Alternativen ausgewählt wird, resultiert in dem selben Effekt für U: U verwendet O_1' oder den nach der Ersetzung zu O_1' äquivalenten Knoten O_1. Die drei Konfigurationsmöglichkeiten werden in (b),(c) und (d) visualisiert. Für den Lösungsprozess[18] bedeutet dies, dass je nach Szenario bis zu drei äquivalente Konfigurationsmöglichkeiten evaluiert werden müssen um die optimale Auswahl festzustellen.

Neben der szenarioabhängigen Redundanz wird durch diesen Interpretationsansatz eine sehr große Anzahl[19] von Kanten in die PBQP-Instanz eingeführt. Insgesamt führt dies zu einem sehr hohen Lösungsaufwand. Es sei angemerkt, dass an dieser Stelle eine Abänderung des Modellierungskonzeptes möglich ist, um mit adjazenten Abhängigkeitskanten eine Verschaltung der beiden Delegations-Alternativen zu erzwingen und so die Redundanzen zu verhindern. Dieser Ansatz wird in dieser Ausarbeitung nicht weiter besprochen.

Weiterhin ist ein zusätzliches Problem in (b) ersichtlich. Nach Definition trennt eine Substitutionskante zwei Knoten, welche durch eine entsprechende Operandenersetzung äquivalent wären. Folglich führt die Ersetzung von O_2 mit O_2' zur Äquivalenz von O_1 und O_1'. Diese Redundanz lässt sich durch CSE[20] lösen und führt zur Identifikation der beiden Knoten miteinander. Das Problem hierbei ist, dass diese Äquivalenz bei Fehlen der Substitutionskante für das PBQP nicht ersichtlich ist, obwohl die Operandenersetzung durchaus modelliert ist. Dieser Sachverhalt kommt im Folgenden zum Tragen, wenn auf explizite Modellierung der Substitutionskanten im PBQP verzichtet wird.

Interpretation: Substitutionskanten als Vermittler. Der oben beschriebene Ansatz zur Kantenextraktion resultiert in einer massiven Erhöhung der Kantenzahl, womit die Komplexität des Lösungsprozesses erheblich erschwert wird. Im Folgenden wird eine Abwandlung dieses Ansatzes beschrieben bei dem Substitutionskanten lediglich als Werkzeug zum Aufspannen transitiver Transformationskanten eingesetzt werden. Dieser Ansatz folgt der Grundidee, dass eine Operandenersetzung nur dann zusätzliche (Transformations-)Information liefert, wenn diese neue Transformationskanten erreichbar macht.

[18] Annahme: Brute-Force
[19] abhängig von n und Verwenderkonstellationen
[20] Common Subexpression Elimination



(a) Ausgangssituation **(b)** Aufspannen der Transformationskante

Abbildung 10: Erzeugung transitiver Transformationskanten

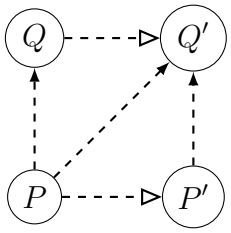


Abbildung 11: Redundanz durch transitive Transformationskanten

Abbildung 10 zeigt die Systematik des abgeänderten Ansatzes. In (a) wird die Ausgangssituation dargestellt. Die Transformationskante (O, O') wurde als Substitutionskante an den Verwender U propagiert. Weiterhin existiert eine Möglichkeit U' in U'' zu transformieren. In (b) wird das Aufspannen der transitiven Kante (U, U'') gezeigt. Die Transformationsmöglichkeit (U', U'') wird über die anliegende Substitutionskante (U, U') an U vermittelt und erzeugt eine weitere (transitive) Transformationskante von U zu U''. Dies erlaubt es, die Transformationsinformation die Substitutionskanten liefern, weiterhin in das PBQP einzupflegen, aber vermeidet dabei das Einbinden einer hohen Zahl von wenig nutzbringenden Transformationskanten.

Formal kann die Menge der transitiven Transformationskanten $\widetilde{T}$ rekursiv über die Menge der klassisch hergeleiteten Transformationskanten T und der Substitutionskanten C definiert werden.

$$\widetilde{T} \coloneqq \{(v, v'') \in N \times N \mid \exists (v, v') \in C^{n+}_{\widetilde{T} \cup \hat{T}}, (v', v'') \in \widetilde{T} \cup \hat{T}\}$$

Es ist zu beachten, dass es die rekursive Definition transitiven Transformationskanten ihrerseits erlaubt Substitutionskanten zu propagieren und als Vorlage für weitere transitive Transformationskanten im Sinne ihrer eigenen Definition zu dienen.

Abbildung 11 illustriert ein Szenario, welches auch bei Verwendung transitiver Transformationskanten redundante Wege zulässt. Die transitive Transformationskante (P, Q') wurde durch den Pfad über P' eingeführt. Der Pfad über Q ist allerdings bereits in der Lage, den Übergang von P nach Q' zu modellieren. Wird die Ersetzung von P mit Q und

die Operandenersetzung notwendig für den Übergang Q nach Q' durchgeführt, so wird aus P ein zu Q' äquivalenter Knoten. Die zusätzliche transitive Transformationskante ist also bezüglich der Überführungsmöglichkeit redundant.

Einschränkung: Bedingte Vermittlung. Die Entstehung partieller Redundanzen lässt sich durch eine geeignete Subgraphüberprüfung verhindern. Wenn nicht jeder Pfad von v nach v'' aus mindestens einer Substitutionskante und einer darauf folgenden Transformationskante besteht, so kann v durch eine geeignete Auswahl schon zu einem zu v'' äquivalenten Knoten werden. Das heißt, existiert kein solcher Pfad, kann die transitive Transformationskante (v, v'') gespannt werden, ohne redundante Information in die PBQP-Instanz einzubringen.

4 Implementierung

Dieser Abschnitt behandelt die Implementierung eines lokalen Optimierungsverfahren, welches im Folgenden als AdvLocal bezeichnet wird, basierend auf dem in Abschnitt 3 hergeleiteten Konzept. Als Implementierungsumgebung kommt die in Abschnitt 2.1.2 vorgestellte Bibliothek LibFirm zum Einsatz.

In Abschnitt 4.1 werden zunächst Details der Implementierungsumgebung erläutert, welche für die Realisierung des Verfahrens relevant sind. Im Speziellen wird hier das in LibFirm verwendete System zur Durchführung lokaler Optimierungen erläutert und damit verbundene Implementierungsherausforderungen aufgezeigt. Es folgen der Überblick über den Implementierungsentwurf in Abschnitt 4.2 und die Folgeabschnitte, welche die Teilimplementierungen des Entwurfs behandeln.

4.1 Implementierungsumgebung LibFirm

LibFirm beinhaltet eine große Menge handgeschriebener lokaler Optimierungsregeln. Zur effizienteren Verwaltung sind diese in drei Gruppen unterteilt.

1. Konstantenfaltung: Auswertung zu einer Konstante

2. Vereinfachung: Rückgabe von Operanden (vgl. algebraische Vereinfachungen)

3. Transformationen: Erzeugung neuer semantisch äquivalenter Knoten

Zweck dieser Klassifizierung ist eine geeignete Priorisierung bei der sukzessiven Regelanwendung. Konstantenfaltungen werten einen gegebenen Ausdruck zu einer Konstante aus und machen damit jegliche weitere Regelanwendung obsolet. Vereinfachungen ihrerseits sind günstig, weil sie auf einen Knotenaufbau verzichten und in der Regel[21] bereits optimierte Knoten zurückliefern.

Es sei angemerkt, dass die obige Klassifizerung in LibFirm nicht vollständig umgesetzt wurde. Im Verlauf dieser Implementierung sind diverse Transformationsregeln aufgefallen, welche zumindest in manchen Einzelfällen strukturell lediglich Operanden zurückliefern. Weiterhin gibt es auch Regeln, welche bei Ausführung den Ausgangsknoten verändern. Dies widerspricht der angedachten seiteneffektfreien Semantik der lokalen Optimierungsregeln in LibFirm. Diese beiden Aspekte müssen bei der Verwendung der entsprechenden Regeln berücksichtigt werden.

[21] Prinzip von `optimize_graph_df`

Die Gruppen sind innerhalb nochmals operationsspezifisch unterteilt. Jeder Teilbereich wird über eine Zugangsfunktion zusammengefasst und als entsprechende Optimierungsfunktion für die jeweilige Operation registriert.

Realisierung als Fixpunktiteration. Die Anwendung der drei Gruppen wird nun von LibFirm als Fixpunktiteration in einer Funktion `transform_node` realisiert. Diese wendet entsprechend der obigen Priorisierung sukzessive die Zugangsfunktionen der Regelgruppen und beginnt jedesmal von Neuem wenn sich eine Änderung ergeben hat. Wurden alle drei Funktionen nacheinander angewandt ohne eine Änderung zu erreichen ist der Fixpunkt und damit das gewünschte Ergebnis erreicht.

Prozeduroptimierung. Weiterhin gibt es die Funktion `optimize_graph_df`, welche einen kompletten Prozedurgraphen traversiert und versucht jeden Knoten mit den lokalen Optimierungen zu optimieren. Dabei wird generell so vorgegangen, dass vor einer Operation des Operanden optimiert werden, um insgesamt den Optimierungsaufwand klein zu halten.

Verwendung der lokalen Optimierungen. Die öffentlichen zugänglichen Methoden, wie `optimize_graph_df` und `optimize_node`, werden von der expliziten `locals`-Phase, aber auch von anderen Programmteilen verwendet. Bei der Generierung neuer Knoten wird beispielsweise generell `optimize_node` aufgerufen, welches wiederum die Fixpunktiteration von `transform_node` aufruft.

4.2 Entwurf

Dieser Unterabschnitt beschreibt die Implementierungsstruktur von AdvLocal. Zunächst werden in Abschnitt 4.2.1 notwendige Modifikationen an LibFirm vorgenommen und AdvLocal als Optimierungsphase eingeordnet. In Abschnitt 4.2.2 wird die Gliederung von AdvLocal in die verschiedenen Teile der Implementierung aufgezeigt und in den Folgeabschnitten konkret die einzelnen Teile beschrieben und erläutert.

4.2.1 Eingliederung in LibFirm

Die bestehende Optimierungsphase `locals` wird durch die neue Phase AdvLocal ersetzt. Hier wurde vorher `optimize_graph_df` aufgerufen, welches genau wie AdvLocal versucht einen kompletten Prozedurgraph zu optimieren. An dieser Stelle muss darauf hingewiesen werden, dass `optimize_graph_df` noch andere Funktionalitäten aufweißt,

die von AdvLocal nicht geleistet werden können. Darunter zählt zum Beispiel die Eliminierung von nicht ereichbarem Code.

Die Fixpunktiteration von `transform_node` wird in AdvLocal aufgespaltet so, dass nicht nur Startknoten und Zielknoten der Fixpunktiteration in das PBQP einfließen, sondern auch die Zwischenergebnisse innerhalb der Fixpunktiteration. Hier muss darauf geachtet werden, dass durch die seiteneffektbehafteten Optimierungsregeln keine ungewollte Veränderung des Eingabegraphs vorgenommen wird.

Zuletzt werden alle Optimierungsregeln in kritische und unkritische Regeln unterteilt. Kritische Optimierungsregeln, sind dadurch ausgezeichnet, dass sie potenziell durch Mehrfachverwendung auch insgesamt zu einer Kostenerhöhung führen könnten. Die Ausführung der kritischen Optimierungsregeln wird außerhalb von AdvLocal verboten, um dort destruktive Veränderungen zu vermeiden, welche von AdvLocal nicht mehr rückgängig gemacht werden könnten.

4.2.2 Gliederung

Die Implementierung von AdvLocal gliedert sich in drei Teile. Der erste Teil, *Generierung*, ist dafür zuständig aus dem gegebenen Programmgraphen möglichst viele Transformationskanten abzuleiten. Der zweite Teil, *Reduktion*, partitioniert den Graph bezüglich der Transformationsmöglichkeiten und setzt die Kantenkontraktion um. Der dritte Teil, PBQP *und Auswertung*, ist für die Konstruktion der PBQP-Instanz zuständig und für Rekonstruktion der anzuwendenden Substitutionen aus der Lösung.

4.3 Generierung

Dieser Teil bildet den Einstig in das Optimierungsverfahren. Zunächst werden verbotene kritische Regeln wieder freigeschaltet und dann in einem komplex verzahnten Prozess die relevanten Transformationskanten über Anwendung von Optimierungsregeln, Propagation und Vermittlung erzeugt.

Abbildung 12 gibt einen Überblick über die Funktionsweise des Generierungs-Algorithmus. Dieser ist die Umsetzung des maximal erweiterten Kantenableitungsverfahrens aus Abschnitt 3.3.2 der theoretischen Betrachtung, welche durch drei miteinander verknüpften geordneten Phasen realisiert wird.

1. Regelanwendung: Wendet Optimierungsregeln auf Knoten an

2. Propagation: Propagiert Transformations- und Substitutionskanten in Verwenderrichtung

3. Vermittlung: Spannt transitive Transformationskanten

Um eine rekursive Struktur zu vermeiden wird der Datenfluss zwischen den Phasen durch eine Indirektionen über Eingabe- und Ausgabepuffer entkoppelt. Die Phasenordnung prägt sich durch die Priorisierung in der Ordnung höher liegender Phasen aus. Damit wird eine Phase nur dann ausgeführt, wenn die Eingabepuffer aller höherliegenden Phasen leer sind.

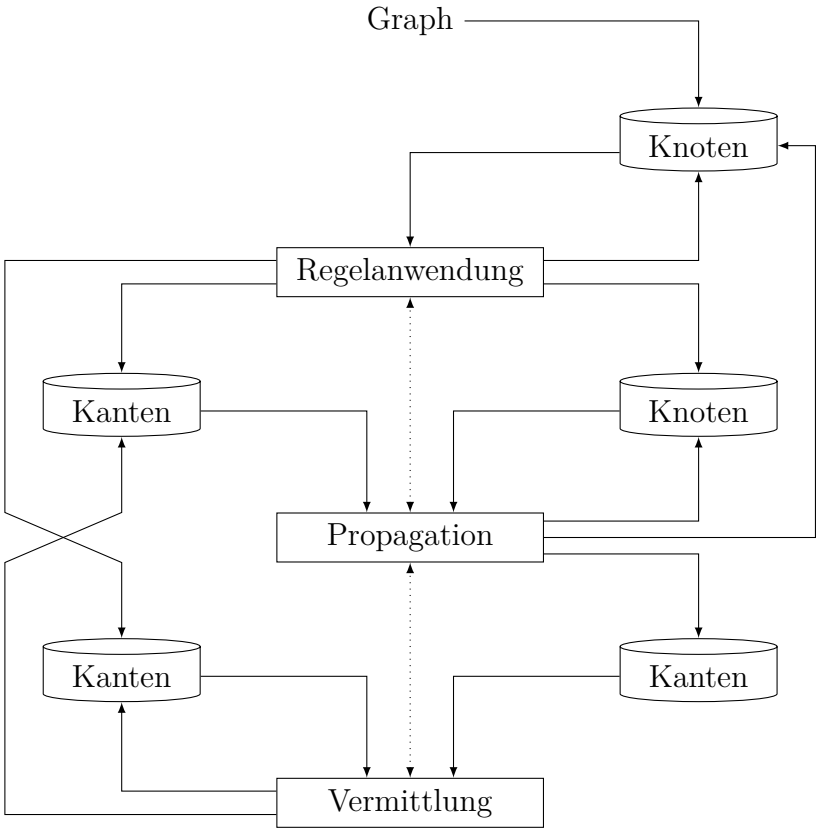


Abbildung 12: Übersichtsdiagramm: Datenfluss und verwaltende Datenstrukturen zwischen den drei Phasen

4.3.1 Regelanwendung

Die Zuständigkeit dieser Phase ist die Anwendung lokaler Optimierungsregeln auf die Eingabeknoten. Eine Markierung nach der Prozessierung stellt sicher, dass ein Knoten nicht zweimal diese Phase durchläuft. Die erfolgreiche Regelanwendung liefert pro Eingabeknoten einen transformierten Knoten und dazu die implizite Transformationskante.

Ist der Knoten noch nicht bekannt, so stellt dieser einen möglicherweise neuen Verwender für die Propagationsphase dar und wird somit in deren Eingabepuffer geschrieben. Die Transformationskante wird, sofern nicht bereits bekannt, in die Eingabepuffer der Propagations- und der Delegationsphase geschrieben. In der Propagationsphase wird die repräsentierte Operandenersetzung zu den Verwendern propagiert und die Delegationsphase verwendet sie um neue transitive Transformationskanten aufzuspannen.

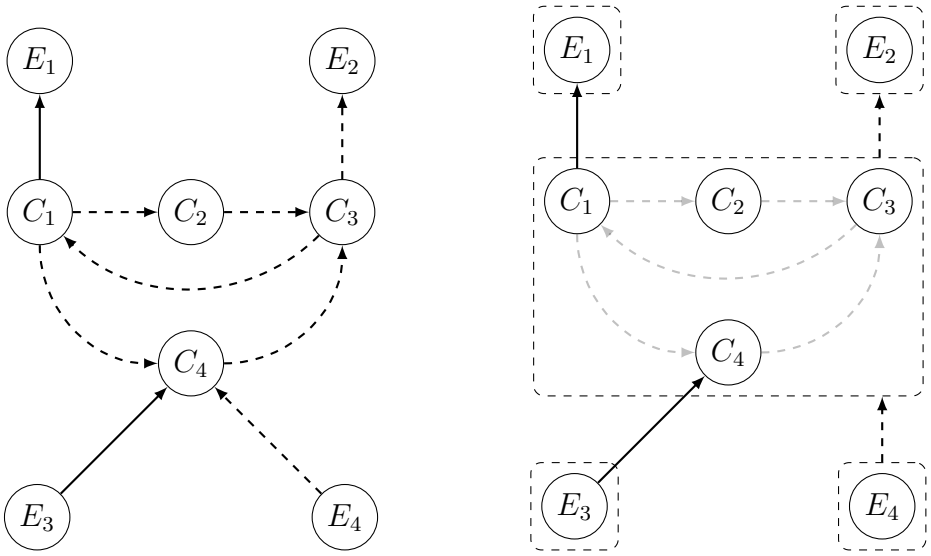
4.3.2 Propagation

Diese Phase hat die Aufgabe für jede Transformationskante Substitutionskanten in Richtung der Verwender zu propagieren. Ist das Eingabeobjekt eine Transformationskante, so werden rekursiv Substitutionskanten propagiert, bis der gesetzte maximale Propagationsindex erreicht ist. Für die Propagierung wird der jeweilige Verwender dupliziert und der entsprechende Parameter mit seiner transformierten Form ersetzt. In diesem Prozess sind zyklenbildende Knotentypen wie `Block` und `Phi` grundsätzlich ausgenommen, da deren Behandlung in nicht-terminierenden Ausführungen resultieren kann. Die Ausführung dieser Phase kann ebenfalls durch einen Knoten angestoßen werden. Dies geschieht, wenn der Knoten erst nachträglich erzeugt wurde und somit bei früheren Propagationen nicht verfügbar war. Eine Verknüpfung jeder Substitutionskante mit ihrem Propagationsindex hat hier den Vorteil, dass diese nun als Zwischenergebnis der Propagation verwendet werden können um die Propagierung an den neuen Verwendern fortzusetzen. Resultate dieser Phase sind Substitutionskanten, welche an die Delegationsphase weitergereicht werden, und potentiell neue Knoten, welche wiederum Kandidaten für die Regelanwendung aber auch wieder neue Verwender sein können.

Die maximale Propagationstiefe wurde auf konstant 2 gesetzt, da für die meisten Regeln eine Propagierung an 2 Ebenen vollkommen ausreicht. Höhere Werte haben ein deutliches Wachstum von Knoten- und Kantenzahl die Folge und müssen dementsprechend sinnvoll begründet sein. Nach der theoretischen Betrachtung müsste die maximale Tiefe auf unendlich gesetzt werden, da Optimierungsregeln existierten, welche sich innerhalb einer Schleife potentiell unendlich weit im Abhängigkeitsgraph nach oben bewegen können. Diese Umsetzung wäre nicht praktikabel, da so jede Transformationskante eine Duplikation des zum Abhängigkeitsfluss inversen Verwendersubgraphs zur Folge hätte.

4.3.3 Vermittlung

Die Phase Vermittlung ist für das Aufspannen transitiver Transformationskanten zuständig. Die Bezeichnung leitet sich von der Eigenschaft der Substitutionskanten ab, Transformationskanten an deren Quelle weiter zu vermitteln und so die besagten Kanten aufzuspannen. Eingaben sind einerseits Transformationskanten, welche über am Quell-



(a) Ausgangssituation **(b)** Partitionierung

Abbildung 13: Starke Zusammenhangskomponenten (bezüglich der Transformationskanten) werden zu Partitionen zusammengefasst

knoten eingehende Substitutionskanten vermittelt werden können und Substitutionskanten, welche durch ihr nachträgliches Enstehen an früheren Vermittlungen nicht teilhaben konnten. Erfolgreich erzeugte Transformationskanten bilden wiederum Kandidaten für die Propagationsphase.

4.4 Reduktion

In der theoretischen Betrachtung wurde formuliert, dass eine geeignete Partitionierung des Graphen notwendig für eine sinnvolle Struktur der später resultierenden PBQP-Instanz ist. Im Folgenden wird die Implementierung der bereits eingeführten Verfahren beschrieben.

Partitionierung. Der Algorithmus von Tarjan [9] ermöglicht die Bestimmung starker Zusammenhangskomponenten in $\mathcal{O}(|V| + |E|)$. Die dadurch induzierte Partitionierung erlaubt die Entfernung aller innerhalb von Zusammenhangskomponenten verlaufender Transformationskanten. Die Partition selbst modelliert fortan die Transformationsmöglichkeiten innerhalb der Partition. Weiterhin werden außerhalb liegende Transformationskanten auf Partitionsniveau gehoben um eine effiziente Partitionentraversierung über Transformationskanten zu ermöglichen. Abhängigkeitskanten werden dabei weiterhin auf Knotenniveau gehalten.

Abbildung 13 illustriert das beschriebene Verfahren. Die Knoten C_1,C_2,C_3 und C_4 sind Teil einer starken Zusammenhangskomponente, da jeder Knoten von jedem anderen Knoten aus erreichbar ist. Diese Zusammenhangskomponente resultiert nun in einer neuen Partition (siehe (b)), welche als Ziel und Quelle für die ein- und ausgehenden Kanten (siehe Kanten von C_3 nach E_2 und E_4 nach C_4) fungiert. Alle innerhalb liegenden Transformationskanten werden dadurch eliminiert, wobei die Abhängigkeitskanten erhalten bleiben. Wichtig ist weiterhin die Feststellung, dass Abhängigkeitskanten nur dann innerhalb einer Zusammenhangskomponente bestehen können, wenn zyklenbildende Knotentypen enthalten sind. Diese würden zwangsläufig zu semantisch sinnfreien Abhängigkeitsschleifen führen und wären schon vor der Zusammenführung existent gewesen.

Kantenkontraktion. In der theoretischen Betrachtung wurde bereits ein Verfahren beschrieben, welches zu Transformationskanten genau dann Gegenkanten einführt, wenn diese keinen Zyklus schließen. Zur Realisierung eignet sich eine Iteration über alle Transformationskanten, welche über eine Breitensuche ($\mathcal{O}(|V| + |E|)$) problematische Pfade aufdeckt. Eine Partitionstraversierung eignet sich hier um das Verfahren zu beschleunigen. Ist das formale Einführen der Gegenkante gültig, wird dies über eine Verschmelzung der angrenzenden Partition realisiert.

4.5 PBQP & Auswertung

Die Konstruktion der PBQP-Instanz gestaltet sich analog zum theoretischen Vorgehen aus Abschnitt 3.3.1. Mittels einer Graphtraversierung über Abhängigkeits- und Transformationskanten lassen sich alle Knoten und somit auch alle Partitionen erfassen. Jede Partition und jede traversierte Kante wird entsprechend der Konstruktionsvorschrift in die Instanz eingepflegt. Die Kostenvektoren wurden auf folgende Weise geordnet:

1. Realisierungs-Alternativen
2. Delegations-Alternativen
3. Void-Alternative

Dieses Schema ist konsistent zur Visualisierung aller PBQP-Konstruktionen in dieser Ausarbeitung. Der zur Verfügung stehende heuristisch arbeitende Löser wurde schon früh im Rahmen der Ausarbeitung gegen den Brute-Force-Löser ausgetauscht, da häufig eine PBQP-Instanzen als nicht lösbar erkannt wurden, welche 1. nach Konstruktion lösbar sein mussten und 2. durch den Brute-Force-Löser optimal gelöst werden konnten.

Rekonstruktion. Nach erfolgreichem Erhalt einer Lösung gilt es, daraus die notwendigen Veränderungen am Programmgraphen abzuleiten. Hierzu eignet sich eine Graphtraversierung über die Abhängigkeitskanten. Beim Knotenbesuch wird aus dem zur entsprechenden Partition gehörigen PBQP-Knoten die Lösung erfasst und die entsprechende Substitution durchgeführt.

- Realisierungs-Alternative: Ersetzung aller Knoten derselben Partition mit dem ausgewählten Knoten.
- Delegations-Alternative: Tiefensuche über Transformationskanten der Instanz bis realisierender PBQP-Knoten gefunden wurde und anschließende Ersetzung.
- Void-Alternative: Ungültige/unmögliche Auswahl

Insbesondere muss darauf geachtet werden, dass die Kante über welche der Knoten erreicht wurde auch entsprechend angepasst wird und keine nicht-ersetzten Knoten in einer Warteschlange verweilen. Bei der Tiefensuche über die Transformationskanten ist es sinnvoll das Ergebnis, wenn einmal gefunden, in jedem traversierten Knoten abzulegen um weitere Tiefensuchen mit demselben Ziel zu beschleunigen.

Die PBQP-Instanzen sind so konstruiert, dass jede Realisierung die Realisierung der abhängigen PBQP-Knoten erzwingt. Ausgehend vom Wurzelknoten kann so also über Abhängigkeitskanten kein PBQP-Knoten erreicht werden, welcher mit der Void-Alternative gelöst wurde.

5 Evaluation

In den vorigen Abschnitten wurde die theoretische Betrachtung und Implementierung des entwickelten Verfahrens AdvLocal behandelt. In diesem Abschnitt werden die angewandten Konzepte und Resultate kritisch evaluiert und der ursprünglichen Implementierung von LibFirm gegenüber gestellt.

Eingesetzte Systeme. Zur Entwicklung der Implementierung und zur Durchführung der Benchmarks wurden zwei Systeme verwendet. Das erste System, im Folgenden als *Entwicklungssystem* bezeichnet, ist ein Lenovo T420 mit einer Intel i5-2520M CPU getaktet mit 2.50GHz bei 2 physischen Kernen und 7.6GiB Hauptspeicher. Dieses System wurde zur Entwicklung der Implementierung und zum Ausführungsvergleich der Kompilate eingesetzt. Das zweite System, im Folgenden als *Testsystem* bezeichnet, ist ein Poolrechner mit einer Intel i3-550 CPU getaktet mit 3.20GHz bei 2 physischen Kernen und 3.66GiB Hauptspeicher. uf diesem System wurden die Benchmarks SPEC2000 und SPEC2006 durchgeführt und die Kompilate für den Ausführungsvergleich erzeugt.

5.1 Performanz zur Übersetzungszeit

Im Rahmen der Implementierung wurde schon beschrieben, dass das heuristische Lösungsverfahren sehr häufig nicht in der Lage war eine Lösung zu finden, obwohl eine Lösung nach Konstruktion mit dem ursprünglichen Programmgraphen immer enthalten ist. Aus diesem Grund wurde grundsätzlich der Brute-Force-Löser mit informationserhaltenden Reduktionen eingesetzt, welcher schon von Buchwald and Zwinkau [4] beschrieben und in LibFirm implementiert wurde.

Der Brute-Force-Ansatz resultiert im Allgemeinen in einer hohen Laufzeit und großem Speicherverbrauch während des Übersetzungsprozesses. In der Regel überstieg der Speicherverbrauch dabei die Kapazitäten des ausführenden Testsystems und führte zu einem Abbruch des Vorgangs.

Die 1438 Testfälle der Firm-Testsuite[22] konnten auf dem Entwicklungssystem bis auf 36 Fehlschläge erfolgreich durchgeführt werden. Der Großteil der Fehlschläge besteht aus `SIGKILL`s[23], welche auf ein Überschreiten des Zeitfensters (maximal 60s Rechenzeit) während des PBQP-Lösungsprozesses zurückführbar sind und dementsprechend eine Auswirkung der Berechnungskomplexität darstellen. Weiterhin sind 3 Fehlschläge durch

[22] Firm-spezifische Testbibliothek `http://pp.info.uni-karlsruhe.de/git/firm-testsuite.git`
[23] forcierter Abbruch durch (Test-)System

SPEC	Suite	Benchmark	GCC	LibFirm	AdvLocal	LibFirm / AdvLocal
CPU2000	CINT	181.mcf	35.3	35.6	35.7	99.7%
		197.parser	81.5	82.9	82.9	100.0%
	CFP	183.equake	32.7	45.7	46.1	99.1%
CPU2006	CINT	429.mcf	303	319	319	100.0%
		462.libquantum	617	779	770	101.1%
	CFP	470.lbm	393	419	420	99.7%

Tabelle 1: Laufzeitvergleich der Kompilate. Messwerte in Sekunden.

Überprüfungen auf Assembler-Ebene feststellbar, welche auf vereinzelte unzureichende oder naiverweise erwartete Optimierungen zurückzuführen sind.

5.2 Performanz zur Laufzeit

Die Leistung eines Optimierungsverfahren zeigt sich letztlich in der Qualität der erzeugten Programme. Zur Analyse dieses Aspekts werden im Folgenden Benchmarks eingesetzt.

5.2.1 SPEC

Zur Evaluierung und Validierung der erarbeiteten Ergebnisse wurden zwei SPEC[24] Benchmarks, CPU2000 und CPU2006, verwendet. Diese unterteilen sich jeweils in zwei Teile, CINT[25] und CFP[26], welche jeweils 12 bis 17 konkrete Benchmarks enthalten.

Tabelle 1 vergleicht durch AdvLocal erreichbare Laufzeiten mit denen der unmodifizierten Version von LibFirm und von GCC[27]. Die Laufzeiten von GCC sind nur als Referenz angegeben und werden im Folgenden nicht näher betrachtet. Es werden hier nur Benchmarks gezeigt, welche das Testsystem mit AdvLocal erfolgreich übersetzen konnte. Die Laufzeiten wurden auf dem Testsystem gemessen und dabei über mehrere Durchführungen gemittelt. Bei CPU2000 wurden 150 Iterationen und bei CPU2006 25 Iterationen pro Benchmark eingesetzt um eine statistische Signifikanz sicherzustellen.

[24] Standard Performance Evaluation Corporation
[25] Integer Component
[26] Floating Point Component
[27] GNU Compiler Collection

Der Laufzeitvergleich zeigt neutrale, positive und negative Ausprägungen. Bei den insgesamt 6 evaluierbaren Benchmarks konnte bei 2 keine Laufzeitveränderung, bei 3 eine Verschlechterung und bei einem eine Verbesserung festgestellt werden. Dabei variiert die Laufzeitveränderung zwischen -1.1 und $+0.9$ Prozentpunkten.

Eine nähere Betrachtung von Benchmark `462.libquantum` zeigt 7 veränderte Subroutinenimplementierungen, wobei auch hier Zuwächse, aber auch Reduktionen feststellbar sind. Wie in Tabelle 1 zu sehen ist, konnte dennoch ingesamt eine deutlich Verbesserung erzielt werden.

5.2.2 Schwachstellen

Negative Laufzeitveränderungen lassen sich auf vier mögliche Ursachen zurückführen:

- Kostenfunktion
- Synergieeffekte mit anderen Optimierungsphasen
- Unzureichende Modellierung von Mehrfachverwendung bei Operandenersetzung
- Beschränkungen der Propagierung

Kostenfunktion. Bei der Konstruktion von PBQP-Instanzen werden Operationen mit ganzzahligen Kosten aus einer Kostenfunktion versehen. Zum Einsatz kam eine generische Kostenfunktionen, welche die ungefähren Verhältnisse zwischen Operationen modellieren kann, aber auf Feinheiten der Zielarchitektur nicht eingeht.

Ausbleibende Synergieeffekte. Das ursprüngliche Vorgehen von LibFirm sieht vor, jederzeit die maximal mögliche Optimierung über lokale Optimierungsregeln vorzunehmen. Die so gewonnene Struktur kann durch folgende Optimierungsphasen unter Umständen besser optimiert werden, wie die nicht-vollständig optimierte Form, welche von AdvLocal erzeugt werden würde. Weiterhin kann die Entfernung von Verwendern durch Optimierungsphasen die konstruierte Kostenoptimalität zerstören. Dadurch ist die getroffene Wahl nicht mehr notwendigerweise optimal und ein erneuter Durchlauf von AdvLocal wäre von Nöten.

Mehrfachverwendung bei Operandenersetzung. Der Ansatz Transformationskanten durch Substitutionskanten einzusparen hat den negativen Effekt, dass Mehrfachverwendung, welche nur durch eine Substitutionskante getrennt wäre von der PBQP-Instanz

nicht modelliert wird. Damit fehlt eine wichtige Information, welche potentiell deutlich effizientere Konstellationen erzeugen kann. Durch eine unglückliche Lösung des PBQP kann somit die Mehrfachverwendung an einer solchen Stelle sogar verhindert werden, obwohl sie durch die normale Anwendung lokaler Optimierungsregeln entstanden wäre.

Beschränkung der Propagierung. Die Propagierung unterliegt einer theoretischen und einer praktischen Einschränkung. In der theoretischen Betrachtung wurde die Propagierung an zyklenbildende Verwender ausgeschlossen um eine Endlosrekursion zu vermeiden. Weiterhin muss die maximale Propagierungstiefe entsprechend der Optimierungsregel mit der tiefsten Operandenbetrachtung angepasst werden. Da Optimierungsregeln mit einer theoretisch unbeschränkt tiefen Operandenbetrachtung existieren, müsste die Propagierungstiefe auf unendlich gesetzt werden. Dies würde eine Programmgraphduplikation pro Transformationskante erfordern, was so nicht praktikabel ist. Dementsprechend muss die Propagierungstiefe auf einen sinnvollen Kompromiss gesetzt werden, welcher notwendigerweise einige Optimierungsregeln nicht berücksichtigen kann.

6 Fazit

In diesem Abschnitt werden die Ergebnisse dieser Ausarbeitung zusammengefasst und in eigenen Worten beurteilt. In Abschnitt 6.1 werden offene Fragen und potentielle Entwicklungsmöglichkeiten aufgezeigt, für die im Rahmen dieser Ausarbeitung keine Zeit mehr vorhanden war.

Das Ziel dieser Ausarbeitung war die Erarbeitung einer eigenen Optimierungsphase, welche eine Ausführungsumgebung für die umsichtigere Anwendung lokaler Optimierungsregeln bereitstellt. Dabei lag der Fokus auf der Betrachtung von gemeinsamen Teilausdrücken, die bei der klassischen Anwendung lokaler Optimierungsregeln eine Einschränkung des Optimierungseffekts und in extremen Fällen eine insgesamte Erhöhung der Kosten zufolge haben können.

In Abschnitt 3 wurde zunächst ein theoretisches Verfahren entwickelt, welches zu einer gegebenen Menge lokaler Optimierungsregeln und einem Programmgraph das Problem eine optimale Anwendung der Regeln zu finden auf das Optimierungsproblem PBQP reduziert. Hierzu wurden verschiedene Ansätze diskutiert und schrittweise in ein Gesamtsystem integriert. Darunter befindet sich ein neues Konzept zur effizienten Herleitung möglichst vieler Transformationsmöglichkeiten ohne eine Graphmodifikation durchzuführen. Die Komplexität von PBQP hinsichtlich der Kantenzahl erforderte dabei einige konzeptionelle Entscheidungen, welche eine Begrenzung der modellierten Information und damit eine Einschränkung des Verbesserunspotenzials zur Folge haben.

In Abschnitt 4 wurde beschrieben, wie das theoretische Verfahren als Optimierungsphase in LibFirm integriert wurde. Hierzu wurde die vorhandene Menge lokaler Optimierungsregeln so angepasst, dass diese in vollem Funktionsumfang nur während des Ablaufs der Optimierungsphase zur Verfügung stehen. So sind unkritische lokale Optimierungen, welche selbst durch Mehrfachverwendung keine höheren Kosten erzeugen könnten, zwischen den eigenen Phasendurchläufen weiterhin möglich. Auch in diesem Abschnitt mussten Entscheidungen getroffen werden, die die potentiellen Verbesserungsmöglichkeiten weiter einschränken. Es zeigte sich schon in den frühen Entwicklungsphasen, dass die Heuristik des Lösers nicht mit dem Modellierungskonzept kompatibel ist. Bei ersten Tests während der Implementierungsphase ist ebenfalls aufgefallen, dass größere Testprogramme teilweise zu viel Zeit und Hauptspeicher in Beschlag nahmen.

Schließlich wurden in Abschnitt 3 die Ergebnisse evaluiert. Dabei zeigte sich, dass die Komplexität des eingesetzten Optimierungsproblems PBQP in Kombination mit größeren Benchmarks die Leistungsgrenze des Testsystems sehr häufig überstieg. In den wenigen Fällen, in denen der Übersetzungsprozess terminierte, waren die erzeugten Programme stets korrekt und die Übersetzungszeit lag dann auch in der selben Größenordnung wie die der unmodifizierten Implementierung. In einem Einzelfall konnte eine deutliche Verbesserung (1.1%) der Laufzeitperformanz erreicht werden, sonst war ledig-

lich diesselbe oder niedrigere Laufzeit (bis zu 0.9%) ermittelbar.

Es zeigt sich insgesamt, dass das Optimierungsproblem PBQP nur bedingt geeignet ist um ein derart vernetzungsintensives Problem effizient zu lösen. Die zum Teil nicht vorteilhaften Endresultate können insgesamt auf kantenzahlbegrenzende Entwurfsentscheidungen und Stellglieder in der Implementierung sowie an sich schwer einschätzbare Synergieeffekte mit anderen Optimierungsphasen zurückgeführt werden.

6.1 Ausblick

Im Rahmen der Ausarbeitung verblieb nicht genug Zeit das Reduktionskonzept mit der vorhandenen Heuristik abzugleichen. Möglicherweise könnten Anpassungen an Konzept oder Heuristik eine passable Lösungsqualität bei geringer Laufzeit zur Folge haben.

Um die PBQP-Instanz an sich zu vereinfachen könnten weitere Reduktionsverfahren für Transformationskanten erarbeitet oder bestehende Reduktionsverfahren verbessert werden. Hier könnte die Klassifizierungsinformation von Optimierungsregeln in LibFirm genutzt werden, um Anpassungen der PBQP-Instanz oder Vorentscheidungen vorzunehmen. Eine weitere Untersuchungsmöglichkeit sind die Transformationskanten, welche auf Grund von Redundanzenausbildung im PBQP ausgeschlossen wurden. Diese könnten geeignet eingesetzt zur Verschmelzung vereinzelter Partitionen beitragen und so die Probleminstanz weiter verkleinern.

Um die Laufzeitperformanz der Verfahrens zu verbessern kann der Propagationsalgorithmus weiter ausgebaut werden. Hier kann konzeptionell die Propagationstiefe relativ zu den Verwendern gesetzt und die Propagierung über `PHI`-Knoten untersucht werden. Allgemein könnte untersucht werden, wie hohe Performanzsteigerungen durch die Hinzunahme der hier bewusst vernachlässigten Informationen erzielt werden können.

Literatur

[1] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Software: Practice and Experience*, 28(8):859–881, July 1998. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8. URL http://dx.doi.org/10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8.

[2] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73562. URL http://doi.acm.org/10.1145/73560.73562.

[3] Martin Trapp, Götz Lindenmaier, and Boris Boesler. Documentation of the Intermediate Representation FIRM. Technical Report 1999-14, Universität Karlsruhe, Fakultät für Informatik, Dec 1999. URL http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz.

[4] Sebastian Buchwald and Andreas Zwinkau. Befehlsauswahl auf expliziten Abhängigkeitsgraphen. Master's thesis, Universität Karlsruhe (TH), December 2008. URL http://www.info.uni-karlsruhe.de/papers/da_buchwald_zwinkau.pdf.

[5] John T. Bagwell, Jr. Local optimizations. In *Proceedings of a Symposium on Compiler Optimization*, pages 52–66, New York, NY, USA, 1970. ACM. doi: 10.1145/800028.808484. URL http://doi.acm.org/10.1145/800028.808484.

[6] Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch. SSA-Based Register Allocation with PBQP. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 42–61. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19860-1. doi: 10.1007/978-3-642-19861-8_4. URL http://dx.doi.org/10.1007/978-3-642-19861-8_4.

[7] Hannes Jakschitsch. Befehlsauswahl auf SSA-Graphen. Master's thesis, IPD Goos, November 2004. URL http://www.info.uni-karlsruhe.de/papers/da_jakschitsch.pdf.

[8] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: a New Approach to Optimization. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi:

http://doi.acm.org/10.1145/1480881.1480915. URL `http://www.cs.cornell.edu/~ross/publications/eqsat/`.

[9] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

Erklärung

Hiermit erkläre ich, Michael Hoff, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum

Unterschrift