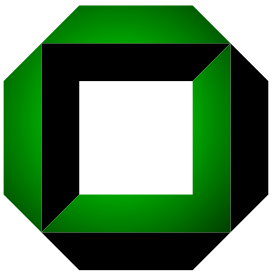


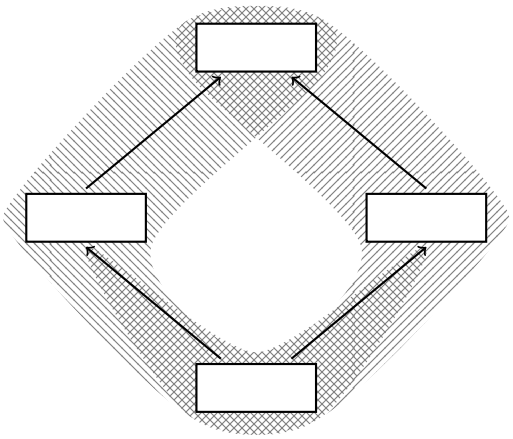
Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825

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

Befehlsauswahl auf expliziten Abhängigkeitsgraphen

Diplomarbeit von Sebastian Buchwald und Andreas Zwinkau

22. Dezember 2008



Betreuer:
Dipl.-Inf. Michael Beck

Verantwortlicher Betreuer:
Prof. em. Dr. Dr. h.c. Gerhard Goos

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

Ort, Datum	Andreas Zwinkau	Sebastian Buchwald

Inhaltsverzeichnis

Symbolverzeichnis

$\mathcal{A}_v$	Menge von Alternativen eines Knotens v.
$\mathcal{M}$	Menge von Mustergraphen.
$\mathcal{U}_G$	Menge von Überdeckungen eines Graphen G.
$\mathrm{cod}(\iota)$	Kodomäne eines Graphmorphismus ι.
deg	Abbildung $V_G \to \mathbb{N}_0$ ordnet jedem Knoten seinen Ausgangsgrad zu.
deg	Abbildung $\Sigma_{EAG} \to \mathbb{N}_0$ ordnet jedem Typ seinen Ausgangsgrad zu.
ι	Einbettung $M \hookrightarrow G$ eines Mustergraphen in einen Graph.
$\pi_{\mathcal{M}}$	Bildet einen Programmgraphen G auf eine PBQP-Instanz bzgl. $\mathcal{M}$ ab.
pos	Abbildung $E_G \to \mathbb{N}_0$ ordnet jeder Kante ihre Position zu.
Ψ	Abbildung $\mathcal{U}_G \to L(P)$ ordnet jeder Überdeckung eine Lösung zu.
$\mathrm{rt}(G)$	Wurzel des Graphen G.
src	Abbildung $E_G \to V_G$ ordnet jeder Kante ihren Quellknoten zu.
tgt	Abbildung $E_G \to V_G$ ordnet jeder Kante ihren Zielknoten zu.
typ	Partielle Abbildung $V_G \to \Sigma_{EAG}$ ordnet Knoten einen Typ zu.
$\vec{a}$	Ein Vektor a.
A_v	Eine Alternative eine PBQP-Knotens v.
$adj(v)$	Menge der Nachbarknoten eines Knoten v.
E_G	Menge der Kanten eines Graphen G.
G	Ein Graph.
$L(P)$	Menge der Lösungen einer PBQP-Instanz P.
M	Ein Mustergraph.
$min(v)$	Minimum der Skalareinträge eines Vektors v.

P	Eine PBQP-Instanz.
p	Ein Pfad eines Graphen.
$u, v, \ldots$	Knoten eines Graphen.
U_G	Überdeckung eines Graphen G.
U_v	Überdeckung eines Knotens v.
V_G	Menge der Knoten eines Graphen G.

1 Einleitung

Die Aufgabe eines Übersetzers ist die Transformation einer menschenlesbaren Quellsprache in eine maschinenlesbare Zielsprache[1]. Man gliedert diese Transformation meist in drei Phasen: Analyse, Optimierung und Synthese. In der Analysephase wird ein Programm der Quellsprache eingelesen und zur internen Verarbeitung in eine Zwischensprache umgewandelt. Diese Zwischensprache ist unabhängig von Quell- und Zielsprache und kann bei einer Erweiterung des Übersetzers um weitere Quell- oder Zielsprachen wiederverwendet werden. Auf der Zwischensprache werden üblicherweise eine Vielzahl verschiedener Optimierungen ausgeführt, die ebenfalls unabhängig von der Zielsprache sind. Die Synthesephase, auch Codegenerierung genannt, wird spezifisch für eine Maschinenarchitektur entwickelt und transformiert die Zwischensprache in eine Zielsprache, die aus maschinenlesbaren Befehlen besteht.

Das Portieren eines Übersetzers auf eine neue Architektur ist ein aufwendiges Projekt. Um den Prozess zu beschleunigen und Fehler zu vermeiden, werden Generatoren benutzt, die aus einer Spezifikation einen entsprechenden Codegenerator erzeugen, so dass dieser nicht manuell programmiert werden muss. Dieses Verfahren wird primär im Bereich eingebetteter Systeme benutzt, in dem eine kurze Entwicklungszeit des Übersetzers aus wirtschaftlichen Gründen höhere Priorität hat. Bei der Softwareentwicklung für Schreibtischrechner ändert sich die Zielarchitektur selten und die Synthesephase der verbreitetsten Übersetzer ist handgeschrieben, denn im Leistungsvergleich schneiden generierte Codegeneratoren bisher schlechter ab. Dies spiegelt sich besonders in der Befehlsauswahl wider, deren Aufgabe die Auswahl geeigneter Maschinenbefehle für die Operationen der Zwischensprache ist.

Bekannte Verfahren zur automatischen Generierung einer Befehlsauswahl arbeiten mit Baumstrukturen. Moderne Übersetzer benutzen allerdings eine Zwischensprache, die Graphstruktur besitzt, da diese Vorteile in der Optimierungsphase bietet. Ebenso verfügen Maschinenarchitekturen mit komplexen Befehlssätzen[2] über Befehle, deren Muster in der Zwischensprache Graphstruktur haben. Eine Befehlsauswahl, die nur Befehle in Baumstruktur verarbeitet, kann also nicht das volle Leistungspotential einer solchen Zielarchitektur nutzen. Dies ist einer der Hauptgründe, warum generative Verfahren sich bisher nicht durchsetzen konnten, obwohl die Entwicklung mit Hilfe eines Generators schneller und mit weniger Fehlern möglich wäre.

[1] Eine Übersetzung von Hochsprache zu Hochsprache werden wir in dieser Arbeit ignorieren.
[2] Maschinen im Complex Instruction Set Computing (CISC) Design

1.1 Problemstellung und Kriterien

Ziel dieser Diplomarbeit ist die Konzeption und Implementierung eines graphbasierten Befehlsauswahlgenerators. Als Grundlage benutzen wir die Zwischensprache Firm[TLB99] in ihrer Implementierung libFirm. Die Zielarchitektur soll im Bereich der Schreibtischrechner verbreitet sein und komplexe Befehle anbieten. Wir benutzen daher IA-32[3], welche seit vielen Jahren die dominante Architektur in diesem Bereich ist.

Kernfunktionalität dieser Befehlsauswahl ist die Transformation von Teilgraphen der Zwischensprache in Befehle der Zielarchitektur. Zur Transformation selbst bietet sich die Verwendung eines Graphersetzungssystems wie GrGen[GBG+06] an, mit dem die Anwendung von deklarativ spezifizierten Ersetzungsregeln gesteuert werden kann. Batz[Bat05] integrierte GrGen in libFirm, um Optimierungen durch Graphtransformationen zu implementieren. Wir wollen diesen Ansatz nun zur Transformation der Zwischensprache in Maschinenbefehle nutzen. Eine solche Befehlsauswahl sollte den folgenden zwei Kriterien genügen:

1. Die Befehlsauswahl eines Übersetzers darf nur Algorithmen enthalten, deren Laufzeit- und Speicherverhalten skaliert.
2. Die Ersetzungsregeln sollen in einer abstrakten Sprache deklariert werden können, damit deren Spezifikation ohne tieferes Wissen über den Übersetzer erstellt werden kann.

1.2 Lösungsansatz

Der Lösungsansatz zu den obigen beiden Kriterien ist folgender:

1. Die Ersetzungsregeln der Befehlsauswahl werden als GrGen-Regeln formuliert. GrGen ist eines der schnellsten[GTB+08] Werkzeuge zur Graphtransformation und bietet skalierende Such- und Ersetzungsfunktionen. Zur Auswahl der Ersetzungsalternativen verwenden wir das abstrakte Optimierungsproblem PBQP[4], für das eine geeignete lineare Heuristik existiert.

   Es ist zu zeigen, dass diese Heuristik für alle Eingaben eine korrekte Auswahl trifft. Dazu entwickeln wir ein mathematisches Modell, das Überdeckungen eines Graphen mit Mustergraphen beschreibt und formalisieren die Auswahl

[3] Intel Architecture 32Bit
[4] Partitioned Boolean Quadratic Problem; dt.: partitioniertes, boolesches, quadratisches Problem; Eine formale Definition befindet sich in Abschnitt 2.4.

einer konfliktfreien Überdeckung. Es sollen hinreichende Voraussetzungen identifiziert werden, die eine korrekte Befehlsauswahl garantieren.

Schließlich implementieren wir eine Befehlsauswahl und zeigen anhand von Testprogrammen und Messungen die Skalierbarkeit des Ansatzes und die Qualität des erzeugten Maschinencodes.

2. Zur Generierung der Ersetzungsregeln entwickeln wir einen Prototypen, der aus einer Architekturspezifikation Ersetzungsregeln für GrGen generiert. Die exploratorisch gewonnenen Erkenntnisse über notwendige Mechanismen und Spezialfälle von Ersetzungsregeln dienen zur Konzeption einer deklarativen Beschreibungssprache. Für eine gegebene Spezifikation in einer solchen Sprache lassen sich die hinreichenden Voraussetzungen für eine korrekte Befehlsauswahl statisch überprüfen, so dass ein Entwickler durch aussagekräftige Fehlermeldungen über fehlende und fehlerhafte Muster unterstützt werden kann.

1.3 Gliederung

In Kapitel 2 wird die Zwischensprache Firm, der Begriff eines Codegenerator-Generators und das Optimierungsproblem PBQP vorgestellt, sowie die Befehlsauswahlphase in die Codegenerierung eingeordnet. Kapitel 3 beschreibt bisherige Ansätze von Codegenerator-Generatoren und insbesondere die PBQP-basierte Befehlsauswahl. Weiter werden zwei Arbeiten vorgestellt, welche das Graphersetzungssystem GrGen im Übersetzerbau eingesetzt haben.

Kapitel 4 beschäftigt sich mit der Korrektheit einer PBQP-basierten Befehlsauswahl, wozu zuerst eine Formalisierung von graphbasierter Befehlsauswahl vorgestellt wird, die einen formalen Beweis unseres Verfahrens ermöglicht. Anschließend zeigen wir, dass durch eine Erweiterung des Algorithmus eine gültige Lösung garantiert werden kann und wie die Voraussetzungen dazu automatisch an einer Maschinenspezifikation überprüft werden können.

In Kapitel 5 findet sich eine Beschreibung unserer Implementierung einer PBQP-basierten Befehlsauswahl und unseres Prototyps einer Spezifikationssprache. Kapitel 6 dokumentiert die Mechanismen, die wir in diesem Prototyp benutzt haben und dessen Grenzen, so dass wir eine abstrakte Sprache skizzieren können, die diese Mechanismen anbietet und die Spezifikation einer Befehlsauswahl im Vergleich zu herkömmlichen Ansätzen vereinfacht. In Kapitel 7 bewerten wir die Leistungsfähigkeit unserer Implementierung und vergleichen sie mit einer handprogrammierten Variante.

Abschließend geben wir in Kapitel 8 einen Ausblick auf mögliche Erweiterungen unserer Arbeit.

2 Grundlagen

Dieses Kapitel erklärt die grundlegenden Begriffe, wie sie in dieser Diplomarbeit verwendet werden.

2.1 Programmdarstellung

Die Art der Zwischensprache ist entscheidend für die Modellierung einer Befehlsauswahl, da im allgemeinen keine Transformation der Programmdarstellung zwischen Optimierungs- und Synthesephase durchgeführt und damit die Befehlsauswahl auf der Zwischensprache getroffen wird. In diesem Abschnitt beschreiben wir die von uns verwendete Zwischensprache.

2.1.1 SSA-Form

Die statische Einmalzuweisung[BCHS98] (SSA[1]) als Konzept für die Zwischensprache ist mittlerweile in praktisch allen modernen Übersetzern etabliert. Die Besonderheit dieser Form ist, dass jede Variable genau einmal einen Wert zugewiesen bekommt. Die Intention besteht darin, prozedurglobale Optimierungen zu erleichtern, indem die Relation zwischen Zuweisung und Benutzung einer Variablen eindeutig ist.

Falls in einer Befehlssequenz der Quellsprache eine Variable a mehrmals Werte zugewiesen werden, wird für jede Zuweisung eine neue Variable (a_1, a_2, ...) erzeugt und deren Benutzer entsprechend angepasst. Durch Schleifen und Verzweigungen im Kontrollfluss kann ein Grundblock mehrere Steuerflussvorgänger haben und der Wert einer Variablen a_3 davon abhängig sein, über welchen Vorgänger der Grundblock erreicht wird, allerdings darf a_3 nur genau einmal ein Wert zugewiesen werden. Falls beispielsweise zwei Vorgänger existieren, könnte $a_3 = a_1$ oder $a_3 = a_2$ sein. Für diesen Fall werden Φ-Knoten eingeführt, die am Beginn eines Grundblocks je nach Vorgänger einen entsprechenden Wert auswählen, die Zuweisung unseres Beispiels wird also $a_3 = \Phi(a_1, a_2)$.

[1] engl.: Single Static Assignment

Der Vorteil dieser Form der Programmdarstellung ist die explizite und eindeutige Verbindung von Zuweisung und Verwendung einer Variablen. Das macht Optimierungen, die Informationen über den Datenfluss benötigen, einfacher und effizienter.

2.1.2 Explizite Abhängigkeitsgraphen

Eine genaue Definition einer Zwischensprache, die nach dem Konzept der SSA-Form modelliert ist, liefert Trapp[Tra01] durch zulässige explizite Abhängigkeitsgraphen. Jeder Knoten im Programmgraph ist mit einer Signatur markiert, die angibt, welche Art von Operation durch ihn repräsentiert wird. Eine Liste aller möglichen Operationen ist in Tabelle 2.1 gegeben. Die genaue Definition ist von Trapp[Tra01] übernommen.

Definition 1 (Expliziter Abhängigkeitsgraph, EAG)**.** Ein expliziter Abhängigkeitsgraph ist ein gerichteter, markierter Graph. Die Knoten sind mit Funktionssymbolen aus Σ_{EAG} markiert. Knoten besitzen geordnete Eingänge und Ausgänge. Die Anzahl der Ein- und Ausgänge der Knoten und ihre Ordnung ist identisch mit der der Parameter und Ergebnisse des zugehörigen Terms aus Σ_{EAG}. Die Ein- und Ausgänge sind mit Typen aus T_{EAG} markiert. Kanten verbinden Ausgänge mit Eingängen desselben Typs. Typen T_{EAG} und Signatur Σ_{EAG} sind in Tabelle 2.1 dargestellt.

Jede Operation[2] verweist durch Kanten auf ihre Operanden und repräsentiert so Datenabhängigkeiten. Es ist eine Besonderheit von EAGs, dass im Gegensatz zu anderen Zwischensprachen Datenabhängigkeiten statt dem dazu inversen Datenfluss abgebildet werden. Genauso wird der Steuerfluss im Programmgraphen durch Abhängigkeitskanten dargestellt. Außerdem existieren noch Speicherabhängigkeitskanten, so dass die Reihenfolge von Speicherzugriffen modelliert werden kann. Speicherabhängigkeiten werden in keiner anderen uns bekannten Zwischensprache explizit modelliert.

Die Typen in T_{EAG} sollten folgendermaßen interpretiert werden:

B Block	**M** Speicher	**I** Ganzzahlwert	**b** Boolscher Wert
X Steuerfluss	**P** Adresse	**F** Fließkommawert	

Weiter definiert Trapp den Begriff eines *zulässigen EAG*. Ein zulässiger EAG ist durch neun Eigenschaften gekennzeichnet, die wir hier nicht weiter ausführen. Diese Eigenschaften beinhalten unter anderem das Konzept der SSA-Form.

[2] Operation und „als Operation markierter Knoten“ verwenden wir synonym

$T_{\text{EAG}} : \{B, X, M, P, I, F, b\}$

Σ_{EAG}:

Block	$: X^n$	$\rightarrow B$
Start	$: B$	$\rightarrow X \times M \times t_1 \times \cdots \times t_n$
End	$: B$	$\rightarrow$
Jmp	$: B$	$\rightarrow X$
Cond	$: B \times b$	$\rightarrow X \times X$
Return	$: B \times M \times t_1 \times \cdots \times t_n$	$\rightarrow X$
Raise	$: B \times M \times P$	$\rightarrow X \times M$
Const	$: B$	$\rightarrow t$
SymConst	$: B$	$\rightarrow I$
Phi	$: B \times s^n$	$\rightarrow s$
CallB	$: B \times M \times P \times t_1 \times \cdots \times t_n$	$\rightarrow X$
CallE	$: B \times X$	$\rightarrow M \times t_{n+1} \times \cdots \times t_{n+m}$
FIn	$: B \times s$	$\rightarrow s$
FOut	$: B \times s$	$\rightarrow s$
Alloc	$: B \times M \times I$	$\rightarrow M \times P$
Load	$: B \times M \times P$	$\rightarrow M \times t$
Store	$: B \times M \times P \times t$	$\rightarrow M$
Sync	$: B \times M^n$	$\rightarrow M$
Sel	$: B \times M \times P \times I$	$\rightarrow P$
Conv	$: B \times t_1$	$\rightarrow t_2$
Add	$: B \times t \times t$	$\rightarrow t$
Sub	$: B \times t \times t$	$\rightarrow t$
Mul	$: B \times t \times t$	$\rightarrow t$
Quot	$: B \times M \times F \times F$	$\rightarrow M \times F$
Div	$: B \times M \times I \times I$	$\rightarrow M \times I$
Mod	$: B \times M \times t \times t$	$\rightarrow M \times I$
Abs	$: B \times t$	$\rightarrow t$
And	$: B \times I \times I$	$\rightarrow I$
Or	$: B \times I \times I$	$\rightarrow I$
Eor	$: B \times I \times I$	$\rightarrow I$
Cmp	$: B \times t \times t$	$\rightarrow b^{16}$
Shl	$: B \times I \times I$	$\rightarrow I$
Shr	$: B \times I \times I$	$\rightarrow I$

Tabelle 2.1: Typen T_{EAG} und Signatur Σ_{EAG}. $t, t_i \in \{P, I, F\}, s \in \{P, I, F, M\}$.

2.1.3 Firm

Eine Instanz eines zulässigen EAG ist die Zwischensprache Firm[TLB99]. Programmgraphen in Firm beginnen mit einem Start-Knoten und hören mit einem End-Knoten auf. Blattknoten können beispielsweise Start-, Const- und SymConst-Knoten sein.

In Tabelle 2.1 sieht man, dass jeder Knoten an erster Stelle auf den Grundblock verweist, in dem er sich befindet. Der Block-Knoten ist allerdings für Befehlsauswahl nicht relevant, da der Kontrollfluss bei der Befehlsauswahl nicht verändert wird. Wenn wir über die ausgehenden Kanten reden, werden wir diese erste Grundblockkante ignorieren. Insbesondere bedeutet ein „nullstelliger“ Knotentyp eigentlich „einstellig“, wobei die erster Kante zum zugehörigen Block-Knoten führt.

Wir verwenden in unserer Arbeit die Firm Implementierung libFirm, die in ihrer aktuellen Fassung nicht mehr genau der Tabelle 2.1 entspricht. Die Menge der Typen wurde erweitert und neue Operationen eingefügt. Unter anderem sind auch die Operationen der Zielarchitektur enthalten. Bis zum Ende des Übersetzerlaufs erfüllt ein Programmgraph die Bedingungen eines zulässigen EAG. So existiert beispielsweise keine Phase in libFirm, die Phi-Knoten eliminiert, sondern diese bleiben bis zum Schluß erhalten. In diesem Sinne sind auch Programmgraphen der Synthese-Phase als Firm-Graphen anzusehen.

2.2 Codegenerator-Generatoren

Um eine Trennung der zur Codegenerierung verwendeten Algorithmen von der Spezifikation der Zielsprache zu erreichen, werden Codegenerator-Generatoren eingesetzt, in denen die Algorithmen gekapselt sind. Der Großteil der Implementierung eines generierten Codegenerators besteht deshalb in der Zielsprachen-Spezifikation, die ausführliche Informationen über die Zielmaschine bereitstellt. Dazu gehört die Spezifikation der vorhandenen Befehle mit Registerbeschränkungen sowie ein abstraktes Kostenmodell.

Generierte Codegeneratoren sind zuverlässiger und schneller zu implementieren als handgeschriebene Codegeneratoren. Allerdings erreichen handgeschriebene Codegeneratoren erfahrungsgemäß eine höhere Qualität des erzeugten Codes, da Zielmaschinen-spezifische Optimierungen an den verwendeten Algorithmen vorgenommen werden können, die sich nicht im Kostenmodell des Codegenerator-Generators widerspiegeln lassen. Die Flexibilität der Algorithmen und des Kostenmodells ist somit ausschlaggebend für die maximal erreichbare Codequalität.

Der Großteil einer Maschinenspezifikation besteht aus den Informationen, die zur Befehlsauswahl nötig sind. Für die Registerzuteilung sind Beschränkungen von Maschinenbefehlen und eine Beschreibung der vorhanden Register notwendig. Für Prozessoren, die eine explizite, parallele Ausführung mehrere Befehle erlauben, sollte die Befehlsanordnung in der Lage sein, entsprechende Gruppen zu bilden. Andere Prozessoren verlangen, dass Ergebnisse von Befehlen nach einer festen Zahl von Takten weiterverarbeitet werden. Diese zusätzlichen Informationen lassen sich als Eigenschaften der Zielbefehle modellieren. Da unsere Arbeit diese Probleme nicht behandelt, beschränken wir uns auf die zur Befehlsauswahl notwendigen Attribute.

2.3 Befehlsauswahl

Befehlsauswahl ist ein Teil der Codegenerierung im Übersetzer. Diese Phase transformiert im Programmgraph Operationen der Zwischensprache in Befehle der Zielarchitektur.

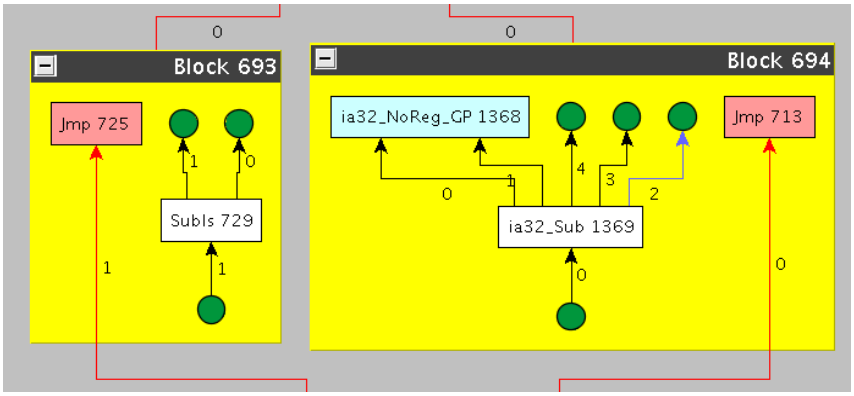


Abbildung 2.1: Beispiel von Befehlsauswahl in libFirm.

Da Firm Programme als Abhängigkeitsgraphen modelliert, kann Befehlsauswahl mittels Graphersetzung realisiert werden. Ein einfaches Beispiel einer 1:1 Ersetzung ist in Abbildung 2.1 gezeigt. Der linke Sub-Knoten der Zwischensprache wird in den rechten ia32_Sub-Knoten transformiert. Für effizienten Code sind komplexere Ersetzungen von n Zwischensprachknoten zu m Zielarchitekturnoten notwendig. Wir suchen also bestimmte Teilmuster in einem Programmabhängigkeitsgraphen und ersetzen diese durch zugehörige Graphen.

2.3.1 Einordnung in die Codegenerierung

Die Befehlsauswahl als Teil der Codegenerierung ist eingebettet in eine Reihe anderer Phasen. In libFirm werden die folgenden Phasen der Reihe nach ausgeführt:

Abbildung des ABI [3] Vorstufe zur Generierung von Funktionsköpfen, -aufrufen und -enden, die dazu führt, dass der Programmgraph bereits Knoten der Zielarchitektur enthält. Diese Knoten müssen bei der Befehlsauswahl ignoriert werden.

Befehlsauswahl Transformation der Zwischensprache in Operationen der Zielarchitektur.

Befehlsanordnung Serialisierung der Operationen. Da Firm Programmgraphen ohne Reihenfolge der Operationen repräsentiert, ist nur durch die Abhängigkeitskanten eine Halbordnung vorgegeben. Es ist aber für die Ausgabe zwingend notwendig, eine möglichst geschickte topologische Sortierung vorzunehmen.

Behandlung von Registerbeschränkungen Auflösen von offensichtlichen Registerkonflikten, die sich durch Auswahl und Anordnung der Befehle ergeben haben, durch Einfügen von Kopieroperationen. Durch diese Phase muss während der Befehlsauswahl keine Rücksicht auf Registerbeschränkungen genommen werden, da alle Konflikte auflösbar sind. Allerdings steckt in dieser Problematik natürlich Optimierungspotential.

Auslagerung Durch Einfügen von Speicher- und Ladeoperationen wird der Registerdrucks auf die Anzahl der verfügbaren Register gesenkt. Nach Hack[Hac07] ist anschließend sichergestellt, dass zu keinem Zeitpunkt *mehr* Register benötigt werden, als verfügbar sind.

Registerzuteilung Zuweisung der endgültigen Registerbelegungen.

Kopienminimierung Entfernen von unnötigen Kopieroperationen, die durch Auslagerung oder Behandlung von Registerbeschränkungen entstanden sind.

Anordnung der Grundblöcke Serialisierung der Grundblöcke ähnlich der Befehlsanordnung. Ziel ist hier primär eine Minimierung von Sprungoperationen.

Gucklochoptimierung Zusätzliche Optimierung, die speziell an die Zielarchitektur angepasst ist. Kurze Sequenzen von Befehlen werden durch günstigere Befehle ersetzt.

Codeausgabe Ausgabe als Assembler-Quelltext.

Die Reihenfolge bedeutet für die Befehlsauswahl, dass Registerbeschränkungen ignoriert werden können. Auch die Anordnung der Befehle muss in den Ersetzungsregeln nicht beachtet werden. Nur die Abhängigkeitskanten müssen entsprechend der benötigten Werte gesetzt werden.

[3] Application Binary Interface; deutsch: Binärschnittstelle

2.3.2 Rematerialisierung

Ein Knoten der Zwischensprache wird in der Befehlsauswahl entweder durch Knoten der Zielarchitektur ersetzt oder er wird mit anderen Knoten zu einem komplexeren Befehl zusammengefasst. Durch Mehrfachüberdeckung mit äquivalenten Mustern (siehe Abschnitt 4.1) kann ein Knoten auch in mehrere komplexe Befehle integriert werden, was oft als Rematerialisierung bezeichnet wird.

Wir benutzen den Begriff Rematerialisierung in dieser Arbeit für die Möglichkeit, einen Knoten durch einen expliziten Befehl zu ersetzen und gleichzeitig in einem komplexen Befehl zu integrieren.

2.4 PBQP

Das Partitioned Boolean Quadratic Problem (PBQP) ist ein abstraktes Optimierungsproblem. Aus einer Reihe von Auswahlmöglichkeiten, die voneinander abhängig sein können, soll eine kostenminimale Auswahl getroffen werden. Im Rahmen dieser Arbeit bilden wir die Befehlsauswahl auf ein solches Optimierungsproblem ab. Dieser Abschnitt definiert das Problem und beschreibt eine Lösungsstrategie.

2.4.1 Min-Plus-Algebra $\mathbb{R}_{\text{min}}$

Zur Formulierung von PBQP, wie wir es einsetzen, benötigen wir einen Zahlenbereich, der ∞ einschließt, um damit auszudrücken, dass manche Auswahlmöglichkeiten unmöglich sind. Zu diesem Zwecke benutzen wir die Min-Plus-Algebra[CQOB92], wie sie in der Beschreibung diskreter Ereignissysteme benutzt wird.

Die Min-Plus-Algebra $(\mathbb{R}_{\text{min}}, \min, +)$ ist definiert auf $\mathbb{R}_{\text{min}} = \mathbb{R} \cup \{\infty\}$ mit den beiden kommutativen Operationen $\min(x, y)$ und $x + y$. Es gelten die folgenden intuitiven Rechenregeln:

$$\min(x, y) = \begin{cases} x & x \leq y \\ y & \text{sonst} \end{cases}$$

$$x + y = \begin{cases} x + y & x \neq \infty \wedge y \neq \infty \\ \infty & \text{sonst} \end{cases}$$

Sowohl $(\mathbb{R}_{\text{min}}, \min)$ als auch $(\mathbb{R}_{\text{min}}, +)$ sind kommutative Monoide mit neutralem Element ∞ bzw. 0. Insbesondere existiert kein additiv inverses Element zu ∞. Weiter

gilt das Distributivgesetz, also $\min(a,b)+c = \min(a+c, b+c)$, woraus folgt, dass $(\mathbb{R}_{\min}, \min, +)$ einen Halbring bildet.

Zusätzlich benötigen wir eine äußere Multiplikation mit dem Restklassenkörper $\mathbb{F}_2 = \{0,1\}$. Dafür definieren wir eine kommutative Operation $\cdot : \mathbb{R}_{\min} \times \mathbb{F}_2 \to \mathbb{R}_{\min}$ mit der üblichen Kurzschreibweise:

$$xy = x \cdot y = \begin{cases} x & y = 1 \\ 0 & y = 0 \end{cases} \text{ für } x \in \mathbb{R}_{\min} \text{ und } y \in \mathbb{F}_2$$

Daraus folgt insbesondere $\infty \cdot 0 = 0$. Man beachte, dass die Multiplikation nicht distributiv bzgl. $+$ ist, denn $0 = \infty \cdot 0 = \infty \cdot (1+1) \neq \infty \cdot 1 + \infty \cdot 1 = \infty + \infty = \infty$.

Diese äußere Multiplikation kann auf Matrizen über $\mathbb{R}_{\min}$ erweitert werden. Seien dazu $\vec{x} \in \mathbb{F}_2^k$, $\vec{y} \in \mathbb{F}_2^l$ und $C \in \mathbb{R}_{\min}^{k \times l}$. Dann ist $\vec{z} = \vec{x}^\top \cdot C \in \mathbb{R}_{\min}^l$ gegeben durch

$$\vec{y}_j = \sum_{i=1}^{k} \vec{x}_i \cdot C_{ij}.$$

Analog ist $\vec{z} = C \cdot \vec{y} \in \mathbb{R}_{\min}^k$ definiert durch

$$\vec{y}_i = \sum_{j=1}^{l} C_{ij} \cdot \vec{x}_j.$$

Da die reelen Zahlen eine Teilmenge von $\mathbb{R}_{\min}$ sind, liefern uns die obigen Definitionen ebenfalls eine äußere Multiplikation von reelwertigen Matrizen mit $\mathbb{F}_2$-Vektoren.

2.4.2 Das Optimierungsproblem

Nun können wir das PBQP formal definieren. Ein ausführliches Anwendungsbeispiel des PBQP folgt in Kapitel 4. Jeder Alternative jeder Auswahlmöglichkeit i sind durch einen Vektor $\vec{c}_i$ Kosten zugeordnet. Außerdem sind jeder Kombination von Alternativen zwischen zwei Auswahlmöglichkeiten i und j durch eine Matrix C_{ij} Kosten zugeordnet. Eine Lösung gibt für jede Auswahlmöglichkeit die Alternative an. Die Gesamtkosten, die minimiert werden sollen, errechnen sich aus den Kosten der gewählten Alternativen und den Kosten der Abhängigkeiten zwischen ihnen.

Definition 2 (Partitioned Boolean Quadratic Problem)**.** Für $n \in \mathbb{N}$ und $i, j \in \{1, \ldots, n\}$ seien Vektoren $\vec{c}_i \in \mathbb{R}^{k_i}$ und für $i < j$ Matrizen $C_{ij} \in \mathbb{R}_{\min}^{k_i \times k_j}$ gegeben. Das Optimierungsproblem besteht darin, boolsche Entscheidungsvektoren $\vec{x}_i \in \mathbb{F}_2^{k_i}$ zu finden, so dass

$$\sum_{1 \leq i < j \leq n} \vec{x}_i^\top \cdot C_{ij} \cdot \vec{x}_j + \sum_{1 \leq i \leq n} \vec{x}_i^\top \cdot \vec{c}_i$$

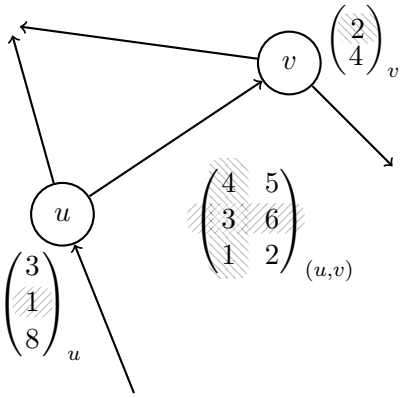


Abbildung 2.2: Das Auswahlprinzip in einem PBQP-Graphen.

unter den Nebenbedingungen

$$\vec{x}_i^{\top} \cdot \vec{1}_{\mathbb{R}} = 1, 1 \leq i \leq n \tag{2.1}$$

$$\vec{x}_i^{\top} \cdot C_{ij} \cdot \vec{x}_j < \infty, 1 \leq i < j \leq n \tag{2.2}$$

minimal wird.

Die erste Nebenbedingung erzwingt, dass genau eine Position pro Entscheidungsvektor ausgewählt wird. Die zweite Nebenbedingung fordert, dass jede Lösung des PBQP reellwertig ist. Somit können inkompatible Vektor-Einträge adjazenter Knoten durch einen ∞-Eintrag in der Kostenmatrix der entsprechenden Kante modelliert werden.

Das PBQP kann auch als Graph formuliert werden: Sei dazu $G = (V, E)$ ein gerichteter Graph mit einer total geordneten Knotenmenge V und Kantenmenge $E = \{(u, v) | u < v, u \in V, v \in V\}$. Wir ordnen jedem Knoten $u \in V$ eine Menge $\mathcal{A}_u = \{A_1, \ldots, A_{|\mathcal{A}_u|}\}$ von Alternativen zu, deren Kosten durch den Vektor $\vec{c}_u$ beschrieben werden. Jeder Kante (u, v) wird die Kostenmatrix $C_{(u,v)}$ zugeordnet. Existiert zwischen zwei Knoten keine Kante (u, v), so sei $C_{(u,v)} = 0$.

Wählt man am Quellknoten u einer Kante (u, v) eine Alternative A_l aus, so wird in der zugehörigen Kostenmatrix $C_{(u,v)}$ die l-te Zeile ausgewählt. Analog legt die Auswahl am Zielknoten v die Spalte der Kostenmatrix fest. In Abbildung 2.2 sieht man zwei Knoten, in deren zugehörigen Vektoren jeweils ein Eintrag (1 bzw. 2) ausgewählt wurde. Aus dieser Auswahl ergibt sich die Auswahl von Spalte und Zeile in der Kantenmatrix und daraus die Auswahl des Feldes mit dem Wert 3.

Für diese Darstellung des PBQP kann eine **Auswahl** als Abbildung l modelliert werden, die jedem Knoten v eine Alternative $l(v) \in \mathcal{A}_v$ zuordnet. Die Kosten einer solchen Auswahl sind

$$c(l) = \sum_{u \in V} c(l(u)) + \sum_{(u,v) \in E} c(l(u), l(v)),$$

wobei $c(A_u, A_v) = c(A_v, A_u)$ die Kosten des durch A_u und A_v gewählten Matrix-Eintrags der Kante zwischen u und v darstellen. Eine Auswahl heißt **Lösung**, wenn die Kosten der Auswahl endlich sind.

Da es beim Lösen einer PBQP-Instanz dazu kommen kann, dass Vektor-Einträge gelöscht werden, modellieren wir diese gelöschten Alternativen durch unendliche Kosten. Dadurch lässt sich die Definition der Kosten auf gelöschte Vektor-Einträge erweitern. Wir werden im Laufe der Arbeit oft auf diese Modellierung zurückgreifen, da sie eine einfache Handhabung gelöschter Einträge ermöglicht.

In der obigen Formulierung des PBQP wird die Orientierung einer Kante durch die Totalordnung der Knoten vorgegeben. Für das PBQP ist es allerdings ausreichend eine beliebige Orientierung der Kanten zu wählen, die festlegt welcher Knoten die Zeile bzw. die Spalte der zugehörigen Kostenmatrix auswählt. Fasst man die Orientierung als Eigenschaft der Kostenmatrizen auf, so kann der PBQP-Graph als ungerichtet angesehen werden. Aus diesem Grund können wir im Verlauf der Arbeit auf Begriffe der Graphentheorie zurückgreifen, die nur für ungerichtete Graphen definiert sind.

2.4.3 Lösen einer PBQP-Instanz

Wir beschreiben nun anschaulich einen PBQP-Lösungsalgorithmus. Im Allgemeinen ist das Finden einer zulässigen Lösung NP-vollständig[Jak04], weswegen in manchen Fällen eine Heuristik zur Lösungsfindung benutzt wird, um die Laufzeit linear in der Anzahl der Knoten zu halten. Falls von der Heuristik eine ungültige Auswahl getroffen wird, liefert der Algorithmus eine Auswahl mit unendlichen Kosten, obwohl die PBQP-Instanz lösbar gewesen wäre. Auf diese Problematik gehen wir in Abschnitt 3.4 genauer ein.

Das Lösen einer PBQP-Instanz geschieht in drei Phasen:

1. Der PBQP-Graph wird reduziert, bis keine Kanten mehr vorhanden sind.
2. Ohne Kanten kann an jedem Knoten lokal das Minimum ausgewählt werden.
3. Durch Rückwärtspropagierung werden die im ersten Schritt entfernten Knoten und Kanten wieder eingefügt und Alternativen ausgewählt.

Wünschenswert sind Vereinfachungen, bei denen aus einer optimalen Lösung der vereinfachten Instanz auch eine optimale Lösung der ursprünglichen Instanz gewonnen werden kann. Einige solcher informationserhaltenden Vereinfachungen möchten wir im Folgenden vorstellen. Eine exakte Beschreibung des Algorithmus ist in Abschnitt 4.4.

Unabhängige Kanten

Unabhängige Kanten sind Kanten mit einer Kostenmatrix der Form

$$C_U = \begin{pmatrix} u_1 + v_1 & \cdots & u_1 + v_m \\ \vdots & \ddots & \vdots \\ u_n + v_1 & \cdots & u_n + v_m \end{pmatrix}$$

wobei $u_i, v_j \in \mathbb{R}_{\min}$. Wir bezeichnen solche Kostenmatrizen ebenfalls als unabhängig. Wie der Begriff schon vermuten lässt, stellen unabhängige Kanten keinerlei Beziehungen zwischen den adjazenten Knoten dar. Die zugehörige Kostenmatrix lässt sich vollständig in die Nullmatrix überführen, indem der Vektor $\vec{u} = (u_1, \ldots, u_n)^\top$ zum Kostenvektor des Quellknotens und der Vektor $\vec{v} = (v_1, \ldots, v_m)^\top$ zum Kostenvektor des Zielknotens addiert wird. Anschließend kann die Kante entfernt werden. Für einige der später folgenden Reduktionen ist es erforderlich einen der beiden Vektoren $\vec{u}$ und $\vec{v}$ bzgl. des Kostenmaßes

$$c(\vec{u}) = \sum_{i=1}^{n} u_i, \qquad \vec{u} \in \mathbb{R}^n_{\min}$$

zu maximieren. Im Fall der Maximierung von $\vec{u}$ sprechen wir von einer Normalisierung bezüglich des Quellknotens, bei $\vec{v}$ von einer Normalisierung bezüglich des Zielknotens.

Ein Beispiel für die Reduktion einer unabhängigen Kante ist in Abbildung 2.3 gezeigt. Besitzen $\vec{u}$ oder $\vec{v}$ einen ∞-Eintrag, so muss dieser gesondert behandelt werden: Nach dem Addieren werden die ∞-Einträge des Kostenvektors und die zugehörigen Zeilen bzw. Spalten der Kostenmatrizen der inzidenten Kanten entfernt. Ist der resultierende Kostenvektor null-dimensional, existiert keine Lösung der PBQP-Instanz.

Wir nennen eine Matrix C_N in Normalform, wenn jede Zeile und jede Spalte mindestens einen Null-Eintrag enthält. Ist eine Matrix C nicht in Normalform, so gibt es eine Zerlegung

$$C = C_N + C_U,$$

wobei C_N in Normalform und C_U eine unabhängige Matrix ist. Wie bereits gezeigt, können die Kosten von C_U auf die Vektoren der inzidenten Knoten übertragen werden. Die resultierende Matrix C'_N liegt in Normalform vor und ist eindeutig bestimmt.

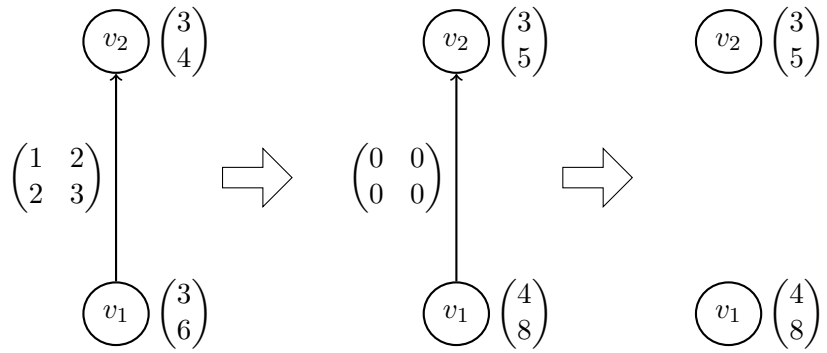


Abbildung 2.3: Die Kante zwischen v_1 und v_2 ist unabhängig und kann nach einer Normalisierung entfernt werden.

Besitzt ein Knoten nur eine Alternative, so sind alle inzidenten Kanten unabhängig und können reduziert werden. Diese Eigenschaft werden wir bei der später vorgestellten Heuristik intensiv nutzen.

Knoten vom Grad Eins

Knoten mit genau einer inzidenten Kante können entfernt werden, nachdem deren Kosten in den Kostenvektor des adjazenten Knotens u verschoben wurden:

$$c'(A_u) = c(A_u) + \min_{A_v \in \mathcal{A}_v} (c(A_v) + c(A_u, A_v)).$$

Wir zeigen zunächst, dass sich die Kosten einer Auswahl durch diese sogenannte $R1$-Reduktion nicht verschlechtern können.

Lemma 1. *Sei l eine Auswahl einer PBQP-Instanz P und P' die durch Anwendung einer $R1$-Reduktion am Knoten v entstandene PBQP-Instanz. Dann ist $l' = l|_{V \setminus \{v\}}$ eine Lösung von P' mit Kosten $c'(l') \leq c(l)$.*

Beweis: Sei u der zu v adjazente Knoten. Es gilt

$$\begin{aligned}
c(l) - c'(l') &= \sum_{u \in V_P} c(l(u)) + \sum_{(u,v) \in E_P} c(l(u), l(v)) - \\
&\quad \sum_{u \in V_{P'}} c'(l'(u)) - \sum_{(u,v) \in E_{P'}} c'(l'(u), l'(v)) \\
&= c(l(u)) + c(l(v)) + c(l(u), l(v)) - c'(l'(u)) \\
&= c(l(v)) + c(l(u), l(v)) - \min_{A_v \in \mathcal{A}_v} (c(A_v) + c(l(u), A_v)) \\
&\geq 0.
\end{aligned}$$

□

Umgekehrt kann im obigen Lemma eine Auswahl von P' stets durch lokale Minimumsbildung zu einer gleichwertigen Auswahl von P fortgesetzt werden.

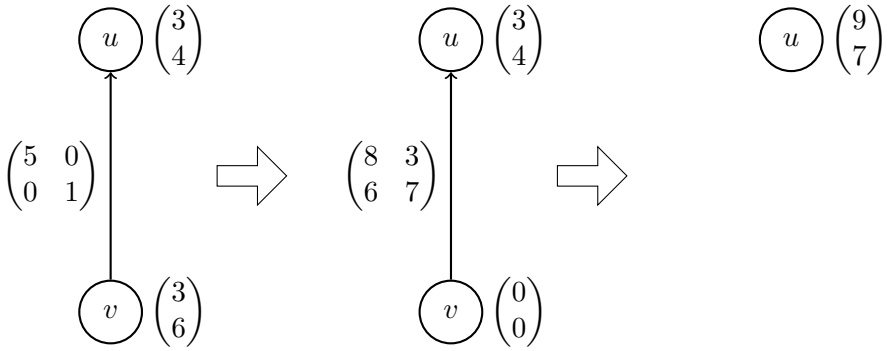


Abbildung 2.4: Knoten mit nur einer inzidenten Kante können entfernt werden.

In Abbildung 2.4 wird eine $R1$-Reduktion des Knotens v gezeigt. Dabei werden die Kosten von v zunächst auf die Spalten der Kostenmatrix addiert

$$\begin{pmatrix} 5 & 0 \\ 0 & 1 \end{pmatrix} \oplus \begin{pmatrix} 3 \\ 6 \end{pmatrix} = \begin{pmatrix} 8 & 3 \\ 6 & 7 \end{pmatrix},$$

um die Kostenmatrix anschließend bezüglich u zu normalisieren

$$\begin{pmatrix} 3 \\ 4 \end{pmatrix} + \begin{pmatrix} \min(8,6) \\ \min(3,7) \end{pmatrix} = \begin{pmatrix} 9 \\ 7 \end{pmatrix}.$$

Durch diese Vorgehensweise wird sichergestellt, dass zu jeder Alternative von u eine Alternative von v existiert, die keine weiteren Kosten birgt. Im Beispiel wird die reduzierte Instanz durch Auswahl der zweiten Alternative optimal gelöst. Fügt man nun den Knoten v wieder hinzu, so zeigt sich, dass die Auswahl der ersten Alternative von v die bisherige Lösung optimal fortsetzt.

Knoten vom Grad Zwei

Ein Knoten v mit genau zwei inzidenten Kanten kann zu einer Kostenmatrix für die adjazenten Knoten u und w reduziert werden, wobei die Matrix-Einträge wie folgt berechnet werden:

$$c'(A_u, A_w) = \min_{A_v \in \mathcal{A}_v} (c(A_u, A_v) + c(A_v) + c(A_v, A_w)).$$

Die so entstandene Kostenmatrix wird auf die Kostenmatrix der gegebenenfalls neu erzeugten Kante zwischen den Knoten u und w addiert. Wir wollen zunächst zeigen, dass sich auch bei „$R2$-Reduktionen“ die Kosten einer Auswahl nicht verschlechtern können.

Lemma 2. *Sei l eine Auswahl einer PBQP-Instanz P und P' die durch Anwendung einer R2-Reduktion am Knoten v entstandene PBQP-Instanz. Dann ist $l' = l|_{V\setminus\{v\}}$ eine Lösung von P' mit Kosten $c'(l') \leq c(l)$.*

Beweis: Seien u und w die adjazenten Knoten von v. Es gilt

$$\begin{aligned} c(l) - c'(l') &= \sum_{u \in V_P} c(l(u)) + \sum_{(u,v) \in E_P} c(l(u), l(v)) - \\ &\quad \sum_{u \in V_{P'}} c'(l'(u)) - \sum_{(u,v) \in E_{P'}} c'(l'(u), l'(v)) \\ &= c(l(u)) + c(l(v)) + c(l(w)) + c(l(u), l(v)) + c(l(v), l(w)) + \\ &\quad c(l(u), l(w)) - c'(l'(u)) - c'(l'(w)) - c'(l'(u), l'(w)) \\ &= c(l(u), l(v)) + c(l(v)) + c(l(v), l(w)) - \\ &\quad \min_{A_v \in \mathcal{A}_v} (c(l(u), A_v) + c(A_v) + c(A_v, l(w))) \\ &\geq 0. \end{aligned}$$

□

Analog zur $R1$-Reduktion kann auch bei der $R2$-Reduktion eine Lösung der reduzierten PBQP-Instanz P' zu einer gleichwertigen Lösung der orginalen PBQP-Instanz P erweitern werden.

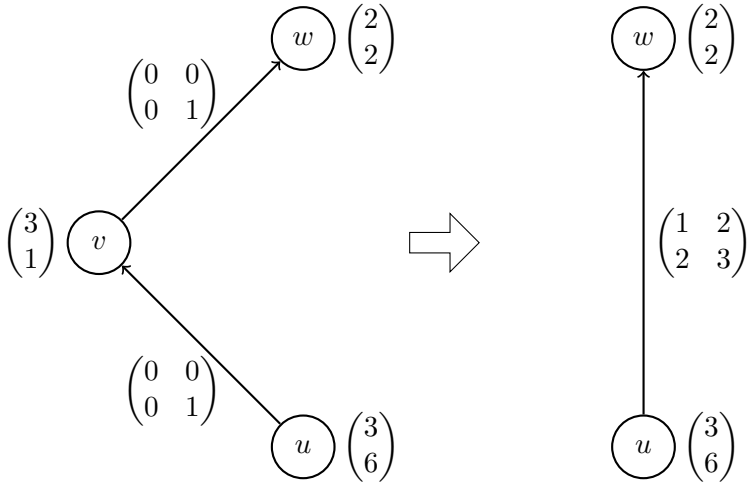


Abbildung 2.5: Knoten mit zwei inzidenten Kanten können zu einer Kante zwischen den adjazenten Knoten transformiert werden.

Für unser Beispiel aus Abbildung 2.5 ist die berechnete Kostenmatrix

$$\begin{pmatrix} \min(0+3+0, 0+1+0) & \min(0+3+0, 0+1+1) \\ \min(0+3+0, 1+1+0) & \min(0+3+0, 1+1+1) \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}$$

sogar unabhängig und die enstandene Kante kann durch Normalisierung entfernt werden. Nach einer Normalisierung bezüglich u erhalten wir Kostenvektoren $\vec{u} = (4, 8)^\top$ und $\vec{w} = (2, 3)^\top$. Anschließend wählen an beiden Knoten lokal die günstigste Alternative aus und erhalten dadurch Gesamtkosten von 6. Durch Auswahl der zweiten Alternative von v können wir diese Lösung zu einer gleichwertigen Lösung der orginalen PBQP-Instanz erweitern.

Heuristik

Erst wenn keine dieser informationserhaltenden Transformationen mehr anwendbar ist, werden heuristische Entscheidungen getroffen, die dazu führen können, dass die gefundene Lösung nicht mehr optimal ist. Die Heuristik wählt an einem Knoten v mit maximalem Knotengrad lokal, d.h. unter Berücksichtigung der adjazenten Knoten, die minimale Alternative aus. Formal wird die Alternative $A_v \in \mathcal{A}_v$ ausgewählt, welche die Kostenfunktion

$$\sum_{u \in adj(v)} \min_{A_u \in \mathcal{A}_u} (c(A_u) + c(A_u, A_v))$$

minimiert. Dabei bedeutet die Auswahl der Alternative A_v, dass alle anderen Alternativen aus dem Kostenvektor entfernt werden.

Da der Knoten nach der Heuristik nur noch eine Alternative besitzt, können alle inzidenten Kanten durch eine Normalisierung entfernt werden. In Abbildung 2.6 muss die Heuristik angewendet werden, weil alle Knoten mindestens drei inzidente Kanten besitzen. Da alle Knoten vom selben Grad sind, können wir frei entscheiden, an welchem Knoten wir die Heuristik anwenden, in unserem Fall ist es Knoten v_2. Wählen wir die erste Alternative, erhalten wir die Kosten $1_{(v_1)} + 1_{(v_1,v_2)} + 3_{(v_2)} + 3_{(v_3)} + 1_{(v_4)} = 9$, bei Wahl der zweiten Alternative ergeben sich Kosten $1_{(v_1)} + 4_{(v_2)} + 2_{(v_3)} + 6_{(v_4)} = 13$. Somit wählen wir an v_2 die erste Alternative aus.

Durch das Dekrementieren der Knotengrade aller zu v_2 adjazenten Knoten sind wieder informationserhaltende Reduktionen auf die entstandene PBQP-Instanz anwendbar. In unserem Beispiel kann jedoch keine Lösung der reduzierten PBQP-Instanz gefunden werden. Eine Auswahl der zweiten Alternative am Knoten v_2 hätte eine Lösung der ursprünglichen PBQP-Instanz ermöglicht. Die Anwendung der Heuristik kann also dazu führen, dass für eine ehemals lösbare PBQP-Instanz keine Lösung gefunden wird.

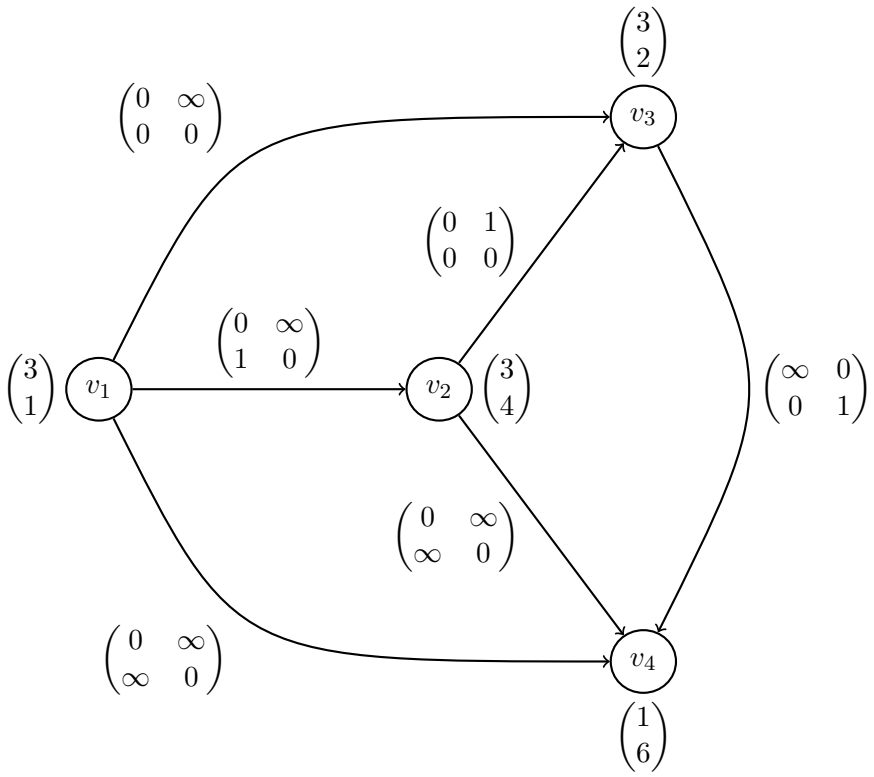


Abbildung 2.6: Knoten mit drei oder mehr inzidenten Kanten werden mittels einer linearen Heuristik entfernt.

In Abschnitt 4.5 zeigen wir, wie sich dieses Problem im Falle der Befehlsauswahl vermeiden lässt. Vereinfacht gesagt garantiert die Transformation von Befehlsauswahl auf das PBQP die Existenz einer Lösung und unser erweiterter Algorithmus findet dann auch immer eine Lösung.

3 Verwandte Arbeiten

Dieses Kapitel stellt grundlegende und verwandte Arbeiten vor. Dazu stellen wir die historische Entwicklung von Codegenerator-Generatoren dar, die zu unserem PBQP-basierten Verfahren geführt hat. Diese Arbeiten betrachten wir vor allem in Hinblick auf verwendete Algorithmen und das Kostenmodell, da diese ausschlaggebend für die maximal erreichbare Codequalität der generierten Befehlsauswahl sind. Weiter werden zwei Arbeiten vorgestellt, die das Graphersetzungssystem GrGen im Übersetzerbau eingesetzt haben.

3.1 Klassische Ansätze zur Befehlsauswahl

Bevor wir auf Algorithmen eingehen, die allgemeine Graphstrukturen verarbeiten, wollen wir in diesem Abschnitt einige Verfahren vorstellen, die sich dadurch auszeichnen, dass sie Baumstrukturen verwenden.

3.1.1 Makrosubstitution

Das erste und einfachste Konzept zur Befehlsauswahl ist die Makrosubstitution, bei der jede Operation der Zwischensprache durch eine Befehlssequenz der Zielarchitektur ersetzt wird. Da kein Kontext zur Verfügung steht, ist die Qualität des erzeugten Codes allerdings recht gering und der Entwurfsprozess entspricht dem naiven, händischen ausprogrammieren der Codegenerierung. In [GFH82] ist ein Überblick der Codeerzeugungstechniken bis Anfang der achtziger Jahre dokumentiert.

3.1.2 LR-Zerteilung

Der erste Codegenerator-Generator wurde von Graham und Glanville[GG78] entwickelt. Die verwendete Zwischensprache liess sich durch eine kontextfreie Grammatik beschreiben und so bot es sich an, die Mustererkennung wurde durch einen LR-Zerteiler zu implementieren. Shift-Reduce- und Reduce-Reduce-Konflikte werden bei diesem Verfahren automatisch aufgelöst, was auch der Nachteil ist. Die Eindeu-

tigkeit des LR-Formalismus ist sehr einschränkend und deswegen nicht geeignet für die inhärent mehrdeutige Abbildung von Zwischensprache auf Zielarchitektur.

3.1.3 Termersetzungssysteme

Ein Termersetzungssystem (TES) besteht aus Ersetzungsregeln für Terme also für Baumstrukturen, so dass insbesondere eine Transformation von Ausdrucksbäumen der Zwischensprachen in Ausdrücke der Zielarchitekturen beschrieben werden kann. Falls die Zwischensprache nicht schon als Liste von Ausdrucksbäumen, sondern als Liste von DAGs modelliert ist, müssen diese zu Bäumen „aufgebrochen“ werden. Eine vereinfachte Form von TES ist ein Grundtermersetzungssystem (GTES), das insofern eingeschränkt ist, dass Terme keine Variablen enthalten.

Dieses GTES Konzept wurde von Hoffmann und O'Donnell[HO82] genutzt als sie das Bottom-Up Pattern Matching (BUPM) entwickelten. Die Muster in einem GTES lassen sich durch eine reguläre Baumgrammatik beschreiben. Die Mustersuche ist also vergleichbar mit dem Zerteilen anhand einer kontextfreien Grammatik und lässt sich ebenso effizient implementieren. Jeder Ersetzungsregel sind Kosten zugeordnet und die Aufgabe der Befehlsauswahl besteht darin, die günstigste Auswahl an Muster zu ersetzen, die den gesamten Baum transformiert. Diese Kostenanalyse kann statisch bei der Generierung der Befehlsauswahl oder dynamisch zur Übersetzerlaufzeit vorgenommen werden. Diese Kostenanalyse ist komplex und zeitaufwendig und verlangsamt je nachdem den Generator (statisch) oder den Übersetzer (dynamisch).

Das Bottom-up Rewrite System (BURS) von Pelegrí-Llopart und Graham[PLG88] unterstützt ein TES zur Spezifikation und ist damit eine Verallgemeinerung von BUPM und als Beschreibungssprache mächtiger. Die Theorie hinter dem BURS ist jedoch komplexer als beim BUPM und eine Implementierung dadurch schwieriger. Ein großer Vorteil ist die kompaktere Regelmenge, da beispielsweise die Kommutativität der Addition durch die zusätzliche Regel `+(X,Y) => +(Y,X)` ausgedrückt werden kann, wobei im Fall von BUPM jede Regel die eine Addition benutzt in beiden Varianten formuliert werden muss.

Ein TES in ein GTES umzuwandeln, welches dieselbe Sprache akzeptiert, ist ein semi-entscheidbares Problem, d.h. es existiert ein Algorithmus, der terminiert falls ein solches GTES existiert. In der Praxis lässt sich ein zur Befehlsauswahl verwendetes TES allerdings immer in ein GTES überführen, weshalb oft die kompaktere TES-Beschreibung bevorzugt wird.

Da ein BURS einfacher zu spezifizieren ist, aber BUPM einfacher zu implementieren und zu verstehen, benutzen praktische Implementierung oft Elemente aus beiden Theorien und lassen sich nicht klar zuordnen. Als Beispiel sei der Back-

End-Generator (BEG) angeführt, der Firm als Zwischensprache und BURS zur Befehlsauswahl benutzt, aber ein GTES als Eingabe nutzt. Die Spezifikation wird als TES verfasst und vor der Generierung in ein GTES übersetzt. BEG wurde von Emmelmann[ESL89] entwickelt und kommerziell vermarktet. Ein weiteres System mit ähnlicher Funktionalität ist Burg[FHP92]. Eine detailiertere Beschreibung dieser Entwicklung wurde von Nymeyer und Katoen[NK97] verfasst.

3.2 Befehlsauswahl auf DAGs

Ein Problem aller erwähnten Systeme ist es, dass Programme als Bäume repräsentiert sind. Eine Eigenschaft, die durch Optimierungen, wie der Eliminierung gemeinsamer Teilausdrücke[1], verloren geht. Zwar erweitert Ertl[Ert99] mit DBurg das Burg-System mit der Fähigkeit DAGs optimal mit Mustern zu überdecken, allerdings sind die Muster selbst nur Bäume. Außerdem arbeiten diese Verfahren nur innerhalb von Grundblöcken, während in Firm eine Funktion als Ganzes verarbeitet wird.

3.2.1 Lineare Programmierung

Wilson[WGHB95] untersuchte den Ansatz, die Codegenerierung auf das gut erforschte Problem der Linearen Programmierung (ILP[2]) abzubilden, für das eine Vielzahl von Lösungsalgorithmen verfügbar sind. Allerdings zeigen Implementierungen dieses Ansatzes, dass die Übersetzerlaufzeit nicht skaliert und deshalb nur sehr kleine Programme verarbeitet werden können.

3.2.2 CGGG

Mit CGGG stellt Boesler[Boe98] eine Erweiterung von BURS auf DAGs vor, die eine A^*-Suche zur Auswahl der Ersetzung verwendet. Durch die A^*-Suche konnte allerdings nicht der gewünschte Laufzeitgewinn erzielt werden, so dass dieses Verfahren für große Graphen nicht geeignet ist.

[1] engl.: common subexpression elimination
[2] engl.: Integer Lineare Programming

3.3 Erweiterung der Befehlsauswahl für SSA-Graphen

Für SSA-Graphen lässt sich DAG-basierten Befehlsauswahl nicht nur auf Grundblöcken sondern auf ganzen Funktionen durchführen. Dabei wird ausgenutzt, dass jeder Zyklus eines SSA-Graphen mindestens einen ϕ-Knoten enthält. Diese ϕ-Knoten werden von der Befehlsauswahl nicht verändert, da eine direkte Verarbeitung von ϕ-Knoten durch Maschinenbefehle nicht unterstützt wird. Spaltet man jeden ϕ-Knoten in zwei Knoten auf, wobei ein Knoten alle Vorgänger und der andere Knoten alle Nachfolger übernimmt, so erhält man einen DAG, auf dem die in Abschnitt 3.2 vorgestellten Verfahren zur Befehlsauswahl anwendbar sind. Nach der Befehlsauswahl werden die gespaltenen Knoten wieder verschmolzen.

In Abbildung 3.1 sieht man im ersten Schritt einen Graphen mit einem Zyklus zwischen Add- und ϕ-Knoten. Dieser Zyklus wird durch die ϕ-Spaltung aufgetrennt und der Programmgraph wird dadurch zu einem DAG, auf dem Befehlsauswahl durchgeführt werden kann. Dabei wird der Add-Knoten durch einen ia32_Lea-Knoten ersetzt, dessen Muster sich über beide Grundblöcke erstreckt. Am Ende wird der ϕ-Knoten wieder zusammengefügt, wodurch der ursprüngliche Zyklus wieder hergestellt ist.

3.4 Abbildung von Befehlsauswahl auf PBQP

Die Idee, Befehlsauswahl durch Lösen eines PBQP durchzuführen, wurde erstmals von Scholz et al.[EKS03] vorgestellt. Dort wird vorausgesetzt, dass der zu transformierende Graph in SSA-Form vorliegt und alle Muster Bäume sind. Die Muster werden durch eine Baumgrammatik beschrieben. Eine Schwäche im PBQP-Lösungsalgorithmus, wie er von Scholz implementiert wurde, führt dazu, dass die Heuristik durch ungeschickte Auswahl das Lösen einer ehemals lösbaren PBQP-Instanz unmöglich machen kann.

In Abbildung 3.2 ist ein Szenario zu sehen, in dem der Lösungsalgorithmus durch ein entsprechendes Kostenmodell dazu gebracht werden kann, die falsche Entscheidung zu treffen. Die Heuristik wählt für den markierte Add-Knoten im ersten Schritt aus, diesen durch ein ia32_Add-Knoten zu ersetzen, der alle Konstanten, insbesondere den Scale-Knoten, integriert. An den anderen Knoten müssen für eine konsistente Auswahl nun entsprechende Alternativen ausgewählt werden. Durch zwei weitere einfache Reduktionen können anschließend auch der Load- und der zweite Add-Knoten entfernt werden. Nun ist zum zweiten Mal eine heuristische Reduktion notwendig und zwar am Scale-Knoten. Es kann die Alternative ausgewählt werden, die Konstante durch einen Befehl zu ersetzen und nicht in einen komplexen Befehl zu integrieren, was inkonsistent zur ersten Auswahl ist, aber der anliegende Shl-Knoten,

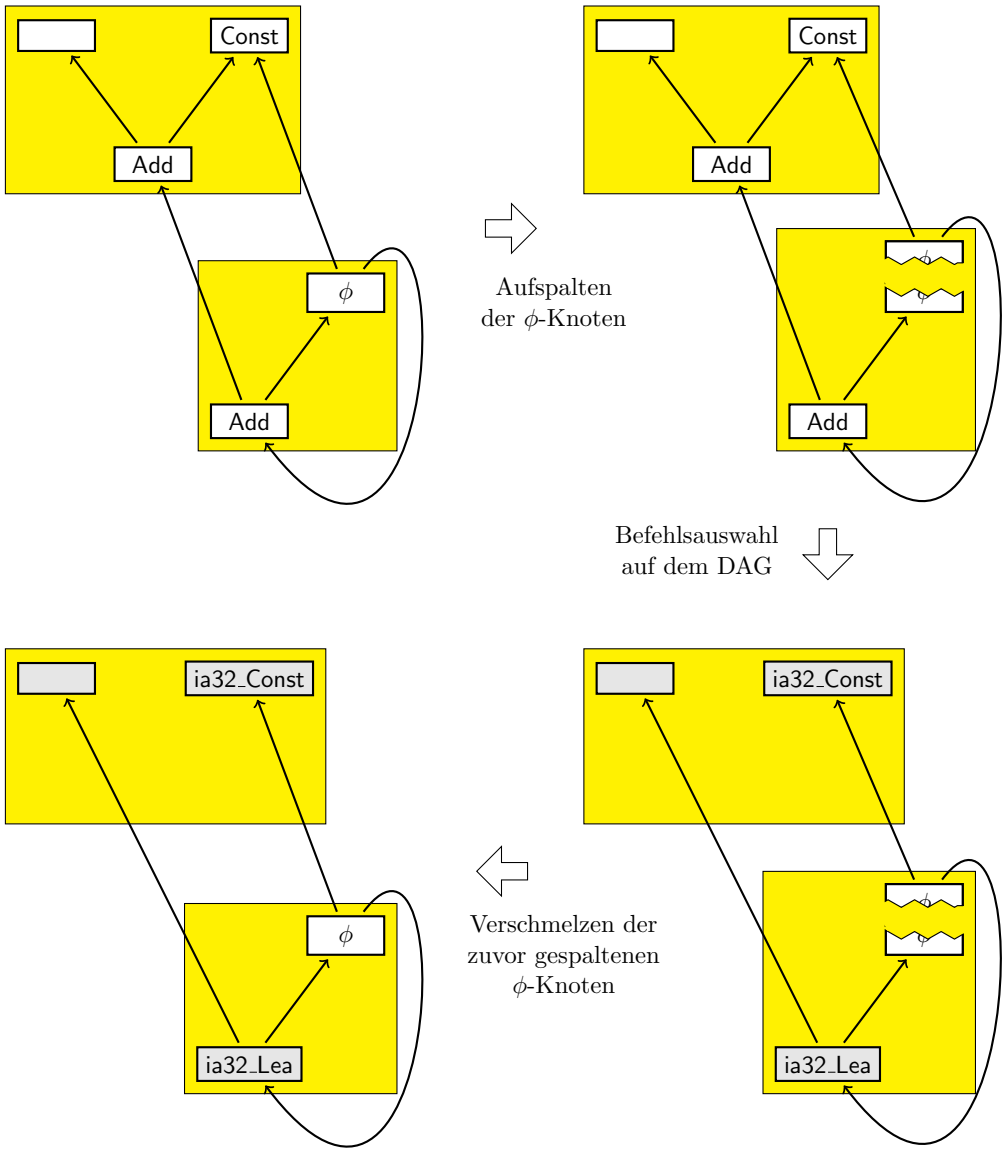


Abbildung 3.1: Durch Aufspalten der ϕ-Knoten kann ein SSA-Graph als DAG aufgefasst werden. Dadurch lässt sich der Kontext einer DAG-basierten Befehlsauswahl von Grundblöcken auf Funktionen erweitern.

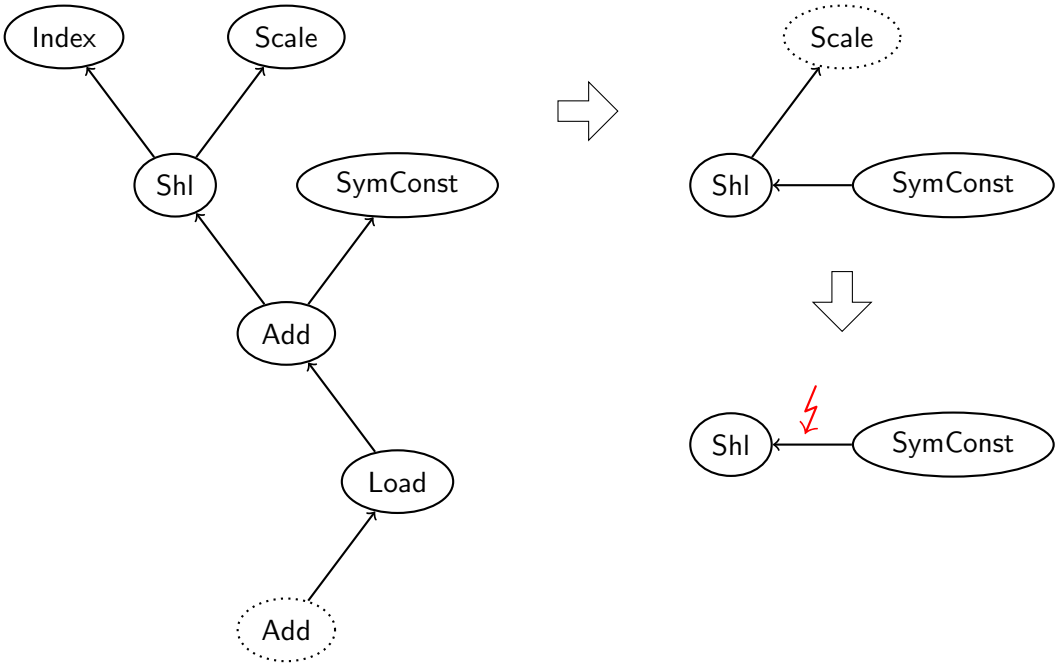


Abbildung 3.2: Situation, in der die Heuristik versagt.

der von der Heuristik betrachtet wird, zeigt nicht die Unmöglichkeit dieser Auswahl. Durch diese ungeschickte Auswahl sind nun im dritten Schritt die gewählten Alternativen an den Shl- und SymConst-Knoten inkompatibel und für die Kante zwischen den beiden Knoten muss ein Matrix-Eintrag mit unendlichen Kosten ausgewählt werden. Somit konnte keine Lösung der PBQP-Instanz gefunden werden.

Damit die Heuristik im zweiten Schritt keine inkompatible Auswahl treffen kann, müssen im Shl-Kostenvektor die entsprechenden Positionen als ungültig gekennzeichnet sein. Nach Anwendung einer heuristischen Reduktion enthält der Kostenvektor ∞-Einträge, die in die Kostenmatrix und rekursiv in anliegende Kostenvektoren übertragen werden müssen. Durch dieses erweiterte Verfahren hätte der Shl-Knoten einen Kostenvektor erhalten, der eine inkompatible Entscheidung am Scale-Knoten verhindert. Im Abschnitt 4.5 zeigen wir, dass mit unserem Algorithmus (siehe Abschnitt 4.4) kein derartiges Problem auftreten kann.

3.5 Erweiterung des Ansatzes

Der Ansatz von Scholz wurde von Jakschitsch[Jak04] um verschiedene Aspekte erweitert. Er stellt eine Modellierung von Rematerialisierung vor und zeigt eine Möglich-

keit zur Reduzierung der Alternativen. Außerdem erweitert Jakschitsch die Baummuster zu DAGs und betrachtet Knoten mit mehreren Ergebnissen.

Rematerialisierung Einen Knoten zu rematerialisieren bedeutet für den Aufbau des PBQP, dass mehrere Alternativen gleichzeitig ausgewählt werden können. Dies wird durch Potenzmengenbildung der ursprünglichen Alternativen erreicht. Somit ist für alle Kombinationen der ursprünglichen Alternativen eine neue Alternative vorhanden. Da dieses Vorgehen zu einer exponentiellen Anzahl an Alternativen führt, greifen wir das Thema bei der Komplexitätsbetrachtung in Abschnitt 4.7 nochmal auf.

Jakschitsch weißt darauf hin, dass bestimmte Alternativen von der Rematerialisierung ausgeschlossen werden müssen, da sich sonst die Semantik des Programms verändert. Diese Alternativen müssen identifiziert und bei der Potenzmengenbildung ausgelassen werden. Zusätzlich können einige ungültige Ersetzungen bereits durch Negativbedingungen bei der Mustersuche vermieden werden. So darf ein volatiler Load-Knoten in einem Adressierungspfad keinen zweiten Verwender besitzen. Hätte der Knoten einen zweiten Verwender, müsste für diesen die Lade-Operation rematerialisiert, also wiederholt, werden. Diese Wiederholung widerspricht allerdings der Semantik einer volatilen Lade-Operation.

Reduktion der Alternativen Verschiedene Ersetzungsregeln suchen nach gleichen Teilmustern (oft Adressberechnung). Jakschitsch identifiziert diese Gleichheit bei der Regelgenerierung und erzeugt eine Tabelle, um diese Information zur Laufzeit des Übersetzers abrufen zu können. So kann festgestellt werden, dass selbst unterschiedliche Operationen, also unterschiedliche Mustergraphen, gleiche Untergraphen besitzen. Für die Mustersuche kann somit für die gemeinsamen Untergraphen eine gemeinsame Suchroutine genutzt werden, was eine mögliche Optimierung für die Mustersuche darstellt. Für die entstehenden Alternativen an Ersetzungen können diese äquivalenten Teilgraphen zusammengefasst werden und die Zahl der Alternativen schrumpft. Bei der Abbildung auf PBQP entstehen so kleinere Kostenvektoren, was der Laufzeit zugute kommt.

Mehrfachergebnisse und DAGs treten auf, wenn die Zielarchitektur Befehle bereitstellt, die mehrere Ergebnisse auf einmal erzeugen. Das Muster eines solchen Befehles hat so viele Wurzeln, wie der Befehl Ergebnisse. Das Problem von Mehrfachergebnissen ist, dass bei der Abbildung auf PBQP die Konsistenz durch entsprechende Einträge in den Kostenmatrizen sichergestellt werden muss. Bei der Lösung des PBQP darf es nicht möglich sein, an einer Wurzel eine Alternative auszuwählen, die mit der Auswahl an einer anderen Wurzel inkompatibel ist.

Ein besonderes Problem, das bei Jakschitsch offen bleibt, kommt bei der in Abbildung 3.3 dargestellten Ersetzung durch einen ia32_Add-Knoten mit Adressberechnung vor. Der zugehörige Befehl lädt einen Wert aus dem Speicher und

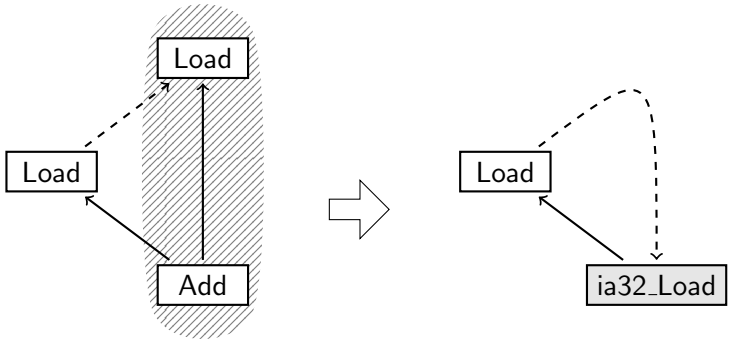


Abbildung 3.3: Gefahr der Zyklenbildung in DAGs.

hat als Ergebnis sowohl den geladenen Wert addiert mit einem Register, als auch den modifizierten Speicherzustand. Es kann durch Speicherabhängigkeiten zu Zyklen kommen, die keine gültige Befehlsanordnung mehr zulassen. Diese Problematik ist nicht innerhalb einer PBQP-Instanz formulierbar und Jakschitsch schlägt eine zusätzliche Validierungskomponente vor. Unsere Implementierung integriert diese Validierung in die Mustersuche wie in Abschnitt 5.2 näher beschrieben.

3.6 Befehle mit mehreren Ergebnissen

Für DAG-Muster mit mehreren Wurzeln entwickelten Wiedermann et al.[Wie08, EBS+08] einen erweiterten Algorithmus zum Aufbau des PBQP. Durch die Unterstützung mehrerer Wurzeln kann es zu unerwünschter Zyklenbildung kommen.

Dieses Problem ist einfach zu verdeutlichen, wenn man Speicheroperationen mit Postinkrement bedenkt, die beispielsweise vom ARM[3]-Befehlssatz angeboten werden. In Abbildung 3.4 ist die Problematik dargestellt. Im oberen Teil ist die Ersetzungsregel angegeben und darunter ein Programmgraph, an dem die Ersetzungsregel zweimal angewendet wird. Durch die zweite Ersetzung bildet sich eine zyklische Abhängigkeit, die es unmöglich macht, ein gültiges Programm zu erzeugen, obwohl jede Ersetzung für sich korrekt ist.

[3] Acorn Risc Architecture, inzwischen nicht mehr Eigentum von Acorn, sondern von der gleichnamigen Firma ARM (Advanced RISC Machines Ltd.). Verbreitet im Bereich eingebetteter Systeme.

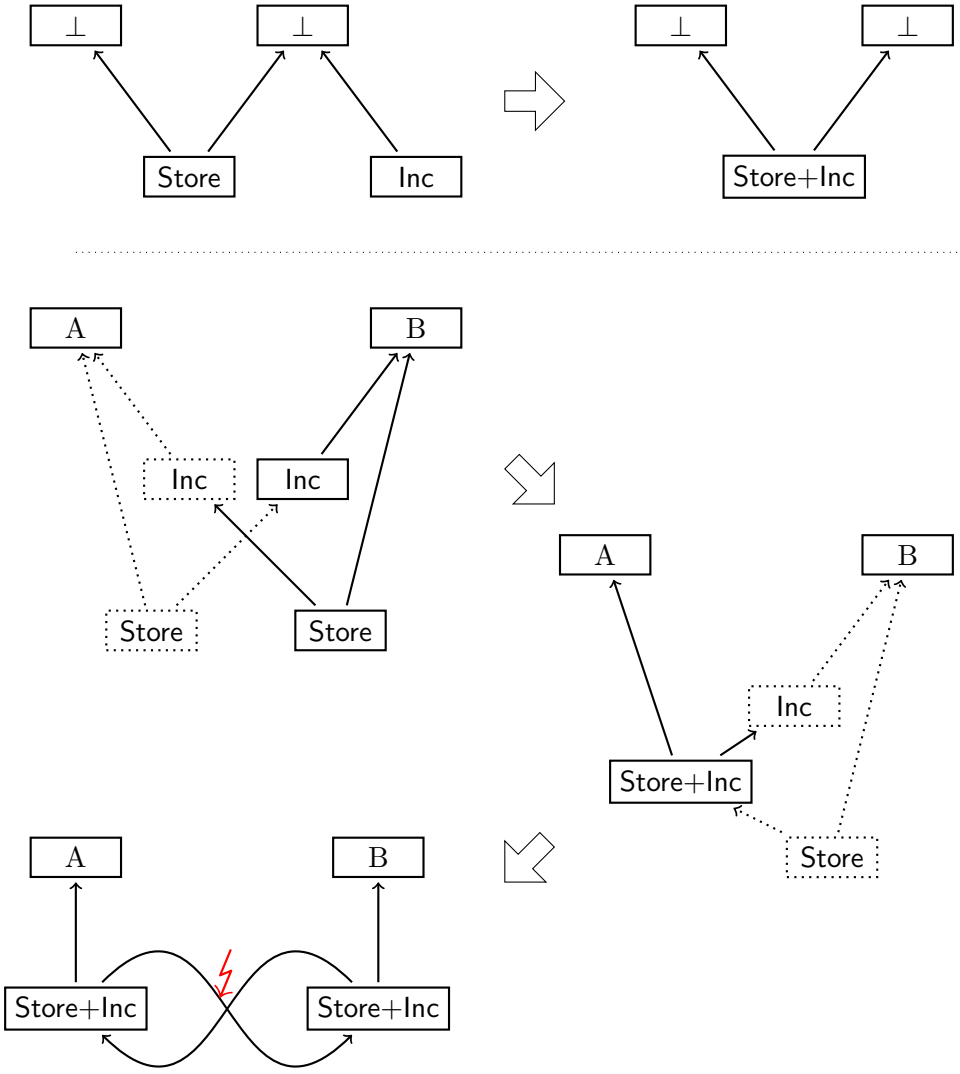


Abbildung 3.4: Problematik bei Speicheroperationen mit Postinkrement.

Wiedermann et al. lösen dieses Problem, indem sie weitere Knoten für Muster mit mehreren Wurzeln einfügen. Durch ∞-Einträge zwischen diesen neuen Knoten wird verhindert, dass eine solche ungültigen Auswahl wie im obigen Beispiel ausgewählt wird. Es ist noch offen, ob der erweiterte Konstruktionsalgorithmus von Wiedermann eine gültige Lösung garantieren kann.

3.7 Graphersetzung im Übersetzerbau

Ein Graphersetzungssystem im Übersetzerbau anzuwenden wurde von Batz[Bat05] untersucht und mit libFirm und GrGen implementiert. Die Motivation bestand darin, Transformationen der Optimierungsphase des Übersetzers durch deklarative Graphersetzung auszudrücken.

Zur Mustersuche wird ein Suchplan erzeugt, der den Suchalgorithmus steuert. Dieser Suchplan wird durch ein Kostenmodell und das Einbeziehen von Informationen über den Arbeitsgraphen optimiert, wodurch eine gute Skalierbarkeit der Mustersuche erreicht wird.

Wir benutzen in unserer Implementierung diese Integration des Graphersetzungswerkzeugs GrGen in den Übersetzer libFirm und erhalten dadurch praktisch lineare Laufzeit der Mustersuche und -ersetzung. Das rührt daher, dass Knoten in Firm-Programmgraphen einen recht geringen durchschnittlichen Knotengrad besitzen. Auch die vollständige Typisierung des Programmgraphen vereinfacht die Suche, da potentielle Fundorte schnell eingrenzbar sind.

3.8 Automatische Regelgewinnung

Schösser[Sch07] zeigt beispielhaft, dass Befehlsauswahl mittels PBQP auch reichhaltige Befehle unterstützt, wie sie beispielsweise im SSE-Befehlssatz vorhanden sind. Er gewinnt die Ersetzungsregeln aus einer Spezifikation, die C-Syntax benutzt um die Muster darzustellen. Dieser Quellcode wird von einem C-Übersetzern verarbeitet, aus dessen Zwischensprache an der Stelle die Befehlsauswahl die Muster und Ersetzungen extrahiert werden. Da die extrahierten Muster Hunderte von Knoten enthalten können, wäre ein erheblicher Aufwand, diese manuell zu spezifizieren. Da die Graphmuster nicht direkt angegeben, sondern vom Übersetzer aus entsprechendem C-Code generiert werden, bietet dieses Verfahren eine sehr einfache Möglichkeit an SIMD-Befehle zu spezifizieren.

Da die Mustergraphen oftmals mehrere Wurzeln besitzen, ist die in Abschnitt 3.6 angesprochene Erweiterung der PBQP-Befehlsauswahl eine notwendige Voraussetzung für die Unterstützung reichhaltiger Befehle. Zusätzlich wird ein ausgefeiltes Kostenmodell benötigt, um ggf. benötigte Kopieroperationen durch Kosten in den Matrix-Einträgen zu modellieren. Aufgrund dieser umfangreichen Erweiterungen wurde die Unterstützung von reichhaltigen Befehlen in dieser Arbeit ausgelassen.

4 Theoretische Betrachtung

In diesem Kapitel stellen wir eine Formalisierung vor, die Überdeckungen von Programmgraphen durch Mustergraphen modelliert. Diese Überdeckungen werden mit Hilfe eines Kostenmodells auf PBQP abgebildet. Durch Lösen des PBQP wird eine Überdeckung gefunden, welche gleichzeitig eine Auswahl von Ersetzungsregeln darstellt, die Zwischensprachoperationen in Befehle der Zielarchitektur transformieren. Weiter zeigen wir, welche Voraussetzungen bei Aufbau und Lösung des PBQP hinreichend sind, um eine korrekte Lösung zu garantieren. Schlussendlich betrachten wir die Laufzeiten verschiedener Varianten des PBQP-Aufbaus.

4.1 Formalisierung

Wie wir in Kapitel 3 beschrieben haben, arbeiten klassische Verfahren der Befehlsauswahl mit Baummustern. Für Graphmuster ist bisher kein formales Modell zur Überdeckung von Programmgraphen bekannt, welches für einen Beweis der Lösungsgarantie verwendbar wäre. Als Vorbereitung auf die folgenden Abschnitte führen wir deshalb eine Kategorie $\underline{\text{Graph}}$ ein, die sowohl die expliziten Abhängigkeitsgraphen als auch die Mustergraphen umfasst, wobei in Mustergraphen Knoten ohne Typ zulässig sind. Dies ermöglicht es uns, den Aufbau der PBQP-Instanzen auf eine Mustergraph-orientierte Art zu formalisieren und somit den Grundstein für den oben angesprochenen Beweis zu legen.

Definition 3 (Graph)**.** Ein Graph ist ein 6-Tupel $G = (V_G, E_G, \text{src}, \text{tgt}, \text{pos}, \text{typ})$, dabei ist

- V_G eine endliche Menge von Knoten
- E_G eine endliche Menge von Kanten
- $\text{src} : E_G \to V_G$ eine Abbildung, die jeder Kante ihren Quellknoten zuordnet
- $\text{tgt} : E_G \to V_G$ eine Abbildung, die jeder Kante ihren Zielknoten zuordnet
- $\text{pos} : E_G \to \mathbb{N}_0$ eine Abbildung, die jeder Kante ihre Position zuordnet
- $\text{typ} : V_G \to \Sigma_{EAG}$ eine partielle Typisierung der Knoten

Für einen untypisierten Knoten v schreiben wir $\text{typ}(v) = \perp$. Jeder Typ $t \in \Sigma_{EAG}$ besitzt einen festen Ausgangsgrad $\deg(t) \in \mathbb{N}_+$. Ebenso ordnen wir jedem Knoten v

einen Ausgangsgrad $\deg(v) = |\{e \in E_G \mid \mathrm{src}(e) = v\}|$ zu, der mit dem Ausgangsgrad des zugehörigen Typs übereinstimmen muss:

$$\mathrm{typ}(v) \neq \perp \Rightarrow \deg(v) = \deg(\mathrm{typ}(v)).$$

Weiter stellen wir zwei zusätzliche Forderungen an die Position einer Kante $e \in E_G$:

$$0 \leq \mathrm{pos}(e) < \deg(\mathrm{src}(e))$$
$$\forall e' \in E_G : \mathrm{src}(e) = \mathrm{src}(e') \wedge \mathrm{pos}(e) = \mathrm{pos}(e') \Rightarrow e = e'.$$

Definition 4 (Morphismen von Graphen)**.** Ein **Morphismus** $f : G \to G'$ ist ein Paar (f_V, f_E) von Abbildungen $f_V : V_G \to V_{G'}$ und $f_E : E_G \to E_{G'}$ mit

$$\begin{aligned} f_V(\mathrm{src}(e)) &= \mathrm{src}(f_E(e)) \\ f_V(\mathrm{tgt}(e)) &= \mathrm{tgt}(f_E(e)) \\ \deg(v) &= \deg(f_V(v)) \\ \mathrm{pos}(e) &= \mathrm{pos}(f_E(e)) \\ \mathrm{typ}(v) \neq \perp &\Rightarrow \mathrm{typ}(v) = \mathrm{typ}(f_V(v)). \end{aligned}$$

Lemma 3. *Die Klasse aller Graphen bildet zusammen mit den Morphismen von Graphen eine Kategorie* Graph*.*

Beweis: Seien $f : G \to G'$ und $g : G' \to G''$ zwei Morphismen. Wir wollen zeigen, dass auch die Komposition $g \circ f = (g_V \circ f_V, g_E \circ f_E)$ ein Morphismus ist. Es gilt

$$\begin{aligned} g_V \circ f_V(\mathrm{src}(e)) &= g_V(f_V(\mathrm{src}(e))) \\ &= g_V(\mathrm{src}(f_E(e)) \\ &= \mathrm{src}(g_E(f_E(e))) \\ &= \mathrm{src}(g_E \circ f_E(e)) \end{aligned}$$

und analog $g_V \circ f_V(\mathrm{tgt}(e)) = \mathrm{tgt}(g_E \circ f_E(e))$. Weiter ist

$$\begin{aligned} \deg(v) &= \deg(f_V(v)) \\ &= \deg(g_V(f_V(v))) \\ &= \deg(g_V \circ f_V(v)) \end{aligned}$$

und auf analoge Weise folgen auch die verbleibenden Bedingungen an einen Morphismus. Die Assoziativität der Komposition von Morphismen folgt direkt aus der Assoziativität von Abbildungen. Sei h ein Morphismus mit $\mathrm{dom}(h) = G''$, dann gilt

$$\begin{aligned} (h \circ g) \circ f &= ((h_V \circ g_V) \circ f_V, (h_E \circ g_E) \circ f_E) \\ &= (h_V \circ (g_V \circ f_V), h_E \circ (g_E \circ f_E)) \\ &= h \circ (g \circ f). \end{aligned}$$

Abschließend zeigen wir die Existenz eines Identitätsmorphismus für einen beliebigen Graph G. Seien f, g zwei Morphismen mit $\text{cod}(f) = G$ und $\text{dom}(g) = G$, dann erfüllt $\text{id}_G = (\text{id}_V, \text{id}_E)$ die Bedingungen eines Identitätsmorphismus:

$$\begin{aligned} \text{id}_{G'} \circ f &= (\text{id}_V \circ f_V, \text{id}_E \circ f_E) \\ &= (f_V, f_E) \\ &= f \end{aligned}$$

$$\begin{aligned} g \circ \text{id}_{G'} &= (g_V \circ \text{id}_V, g_E \circ \text{id}_E) \\ &= (g_V, g_E) \\ &= g. \end{aligned}$$

□

Mit diesem Wissen können wir nun Isomorphie von Graphen definieren, welche wir unter anderem in Definition 15 nutzen werden.

Definition 5 (Isomorphismen von Graphen)**.** Ein Morphismus $f : G \to G'$ heißt **Isomorphismus**, wenn ein Morphismus $g : G' \to G$ existiert, für den gilt:

$$\begin{aligned} f \circ g &= \text{id}_{G'} \\ g \circ f &= \text{id}_G. \end{aligned}$$

Definition 6 (Isomorphie von Graphen)**.** Zwei Graphen G und G' sind **isomorph** $G \cong G'$, wenn ein Isomorphismus $f : G \to G'$ existiert.

Wie bereits erwähnt, vereinigt die Kategorie Graph die gemeinsamen Eigenschaften von expliziten Abhängigkeitsgraphen und Mustergraphen. Wir wollen nun die spezifischen Eigenschaften der beiden Graphklassen herausstellen.

Definition 7 (vollständig typisierter Graph)**.** Ein Graph G heißt **vollständig typisiert**, wenn für jeden Knoten $v \in V_G$ gilt:

$$\text{typ}(v) \neq \bot \, .$$

Definition 8 (Pfad)**.** Sei G ein Graph. Ein **Pfad** in G ist eine Folge $p = (e_1, \ldots, e_n)$ von Kanten $e_i \in E_G$ mit $\text{tgt}(e_i) = \text{src}(e_{i+1})$. Dabei bezeichnet $|p| = n$ die **Länge** des Pfades p. Weiter seien $\text{src}(p) = \text{src}(e_1)$ und $\text{tgt}(p) = \text{tgt}(e_n)$ die **Quell-** bzw. **Zielknoten** von p. Besitzt p die Länge 0, so sei $\text{src}(p) = \text{tgt}(p) \in V_G$.

Definition 9 (Azyklischer Graph)**.** Ein Graph G heißt **azyklisch**, wenn jeder Pfad von $v \in V_G$ nach v die Länge 0 besitzt.

Definition 10 (Programmgraph)**.** Ein **Programmgraph** G ist ein vollständig typisierter, azyklischer Graph.

Die meisten Firm-Programmgraphen sind zyklisch, allerdings kann im Rahmen der Befehlsauswahl von einem azyklischen Programmgraphen ausgegangen werden, da Phi-Knoten entsprechend Abbildung 3.1 aufgespalten werden. Durch diese Betrachtungsweise sind Firm-Graphen azyklisch und damit Programmgraphen im Sinne der Definition.

Definition 11 (Gewurzelter Graph)**.** Ein Graph G heißt **gewurzelt**, wenn ein Knoten $v \in V_G$ existiert, so dass es für alle Knoten $v' \in V_G$ einen Pfad von v nach v' gibt.

Firm-Programmgraphen sind zwar nicht gewurzelt, aber für die Mustergraphen wollen wir diese Eigenschaft fordern. Dies ist keine Einschränkung der Formalisierung durch Graphmorphismen, aber die betrachteten Ersetzungsregeln beinhalten nur gewurzelte Muster. Daraus folgen für Mustergraphen direkt zwei weitere Eigenschaften:

Korollar 1. *Jeder azyklische, gewurzelte Graph G besitzt eine eindeutige Wurzel* $\mathrm{rt}(G)$.

Korollar 2. *Jeder gewurzelte Graph ist schwach zusammenhängend.*

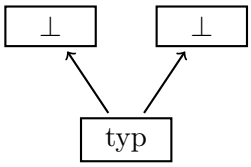
Definition 12 (Mustergraph)**.** Ein Mustergraph M ist ein gewurzelter azyklischer Graph mit

$$\mathrm{typ}(\mathrm{rt}(M)) \neq \bot$$
$$\mathrm{typ}(v) = \bot \Rightarrow \deg(v) = 0$$

für alle Knoten $v \in V_M$.

Im Gegensatz zu Programmgraphen sind Mustergraphen nicht vollständig typisiert. Ein Typ wird nur für den Wurzelknoten gefordert. Ein Knoten ohne Typ muss allerdings Knotengrad 0 besitzen, also ein Blattknoten sein.

Unsere Formalisierung von Mustergraphen umfasst somit Baummuster[EKS03], aber keine Muster mit mehreren Wurzeln[EBS+08, Jak04] und lässt sich somit zwischen den bestehenden Ansätzen einordnen. Eine Unterklasse der Mustergraphen stellen die „unzerteilbaren“ oder atomaren Mustergraphen dar.



Definition 13 (Atomare Mustergraphen)**.** Ein Mustergraph M heißt **atomar**, wenn für alle Knoten $v \in V_M$ gilt

$$v \neq \mathrm{rt}(M) \Rightarrow \mathrm{typ}(v) = \bot \ .$$

Nichtatomare, also komplexere, Mustergraphen besitzen oft gleiche Teilmuster, die sich im Programmgraphen überlappen können. Abbildung 4.1 zeigt eine Adressberechnung, die von mehreren Operationen genutzt werden kann. Für den Store- und den Load-Knoten existieren Überdeckungen die sich in dem markierten Muster überschneiden. Die gemeinsamen Teilmuster sollen als äquivalent erkannt und in *einem* Vektor-Eintrag des PBQP zusammengefasst werden. Die Wurzelknoten sind dabei explizit *nicht* äquivalent, da sie einen eigenen Vektor-Eintrag erhalten sollen.

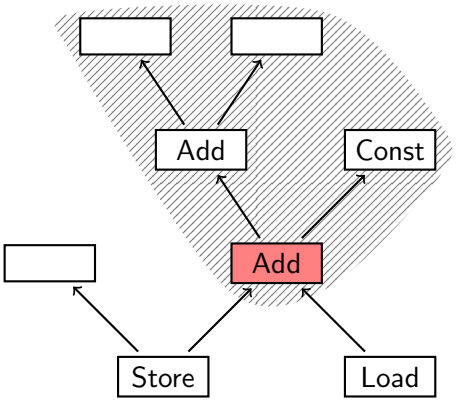


Abbildung 4.1: DAG mit markiertem Knoten und dessen aufgespanntem Graph.

Definition 14 (Aufgespannter Graph)**.** Sei G ein Graph und $v \in V_G$ ein Knoten. Sei weiter

$$V(v) = \{v' \in V_G \mid \text{es existiert ein Pfad von } v \text{ nach } v'\}$$

die Menge der von v erreichbaren Knoten. Der von v **aufgespannte Graph** $G(v)$ ist der von $V(v)$ induzierte Untergraph.

Über das Konzept der aufgespannten Graphen können wir Knoten verschiedener Muster in Relation setzen, um Überlappung von Mustern formal fassen zu können. Wurzelknoten nehmen wir explizit aus dieser Relation, abgesehen davon, dass ein Knoten äquivalent zu sich selbst ist.

Definition 15 (Äquivalenz von Musterknoten)**.** Sei $(M_i)_{i \in I}$ eine Familie von Mustergraphen mit $v_i \in V_{M_i}$ und $v_j \in V_{M_j}$. Durch

$$v_i \sim v_j \Leftrightarrow v_i = v_j \vee (G(v_i) \cong G(v_j) \wedge \mathrm{rt}(M_i) \neq v_i \wedge \mathrm{rt}(M_j) \neq v_j)$$

erhalten wir eine Äquivalenzrelation auf $\bigcup_{i \in I} V_{M_i}$.

Bisher stehen die Mustergraphen in keiner Beziehung zum Programmgraph. Diese Verbindung soll durch den Begriff der „Überdeckung“ geschaffen werden, den wir erst für einzelne Knoten und anschließend für den ganzen Programmgraph definieren. Für die Überdeckung eines Knotens ist zu beachten, dass durch Überlappung von Mustern ein Knoten von mehreren Mustern gleichzeitig überdeckt werden kann.

Definition 16 (Überdeckung eines Knotens)**.** Eine **Überdeckung** U_v eines Knotens $v \in V_G$ eines azyklischen Graphen G durch eine Menge von Mustergraphen $\mathcal{M}$ ist eine nichtleere Menge von Paaren

$$(M_i, \iota_i).$$

Dabei seien $M_i \in \mathcal{M}$ und $\iota_i : M_i \hookrightarrow G$ injektive Morphismen. Für jedes Paar fordern wir die Existenz eines Urbilds von v mit gleichem Typ:

$$\forall i : v \in \mathrm{cod}(\iota_i) \wedge \mathrm{typ}(\iota_i^{-1}(v)) = \mathrm{typ}(v), \tag{4.1}$$

wobei zwei undefinierte Typen als gleich betrachtet werden. Weiter gelte für alle Paare ι_i, ι_j mit $\iota_i(v_i) = v$ und $\iota_j(v_j) = v$:

$$v_i \sim v_j. \tag{4.2}$$

Zur Illustration betrachten wir Abbildung 4.2, in der ein Programmgraph und zwei Mustergraphen M_A und M_B gegeben sind. Im Sinne der obigen Definition wird die Überdeckung $\{(M_A, \iota_A), (M_B, \iota_B)\}$ des Shl- bzw. des Const-Knotens illustriert. Diese Knoten besitzen zwei Urbilder, da sie von zwei Mustern gleichzeitig überdeckt werden.

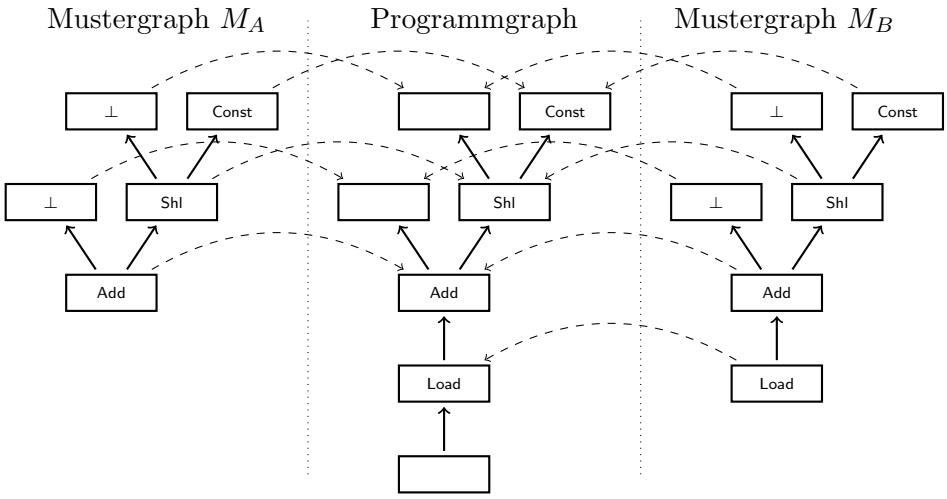


Abbildung 4.2: Überdeckung von Knoten durch Muster.

Sowohl die beiden Const-Knoten als auch die Shl-Knoten der Muster sind äquivalent, da die durch sie aufgespannten Teilmuster der beiden Mustergraphen isomorph sind. Der Add-Knoten dagegen ist in M_A ein Wurzelknoten, kann deswegen nach Definition 15 nicht äquivalent zum Add-Knoten aus M_B sein, der keine Wurzel ist, und verletzt deshalb Gleichung 4.2.

Definition 17 (Überdeckung eines Graphen)**.** Eine Menge U_G von Paaren (M_i, ι_i) heißt **Überdeckung** eines azyklischen Graphen G, wenn für jeden Knoten $v \in V_G$ die Menge

$$U_v = \{(M, \iota) \in U_G \mid v \in \text{cod}(\iota) \wedge \text{typ}(\iota^{-1}(v)) = \text{typ}(v)\}$$

eine Überdeckung von v ist. Weiter fordern wir für alle Knoten $v \in V_G$:

$$(\text{typ}(v) \neq \perp \wedge \exists i : v \in \text{cod}(\iota_i) \wedge \text{typ}(\iota_i^{-1}(v)) = \perp) \Rightarrow \exists j : \iota_j(\text{rt}(M_j)) = v. \tag{4.3}$$

Die Forderung in Gleichung 4.3 drückt aus, dass jeder Knoten, auf den ein typloser Blattknoten eines Musters abgebildet wird, auch von einem Wurzelknoten eines anderen Musters überdeckt werden muss. Umgekehr bedeutet das, wenn ein Wurzelknoten w auf einen Knoten v abgebildet wird, dann müssen andere Knoten u, die ebenfalls aus v abgebildet werden von undefiniertem Typ sein.

Als Beispiel betrachten wir noch einmal Abbildung 4.2, wo die beiden Mustergraphen M_A und M_B nicht beide Teil einer Überdeckung des Programmgraphen sein können. In Definition 17 wird für $v = \mathsf{Add}$ gefordert, dass $U_v = \{(M_A, \iota_A), (M_B, \iota_B)\}$ eine Überdeckung des Add-Knotens bildet, was wie bereits erwähnt im Widerspruch zu Gleichung 4.2 steht.

4.2 Aufbau des PBQP

In diesem Abschnitt konstruieren wir aus einem Programmgraph G und einer Mustermenge $\mathcal{M}$ eine PBQP-Instanz $P = \pi_{\mathcal{M}}(G)$. Zu unserem Programmgraphen G muss zunächst ein PBQP-Graph (V_P, E_P) konstruiert werden:

$$\begin{aligned} V_P &= V_G \\ E_P &= \{(\text{src}(e), \text{tgt}(e)) \mid e \in E_G\}. \end{aligned}$$

Da unsere Programmgraphen azyklisch sind, existiert eine Totalordnung von V_P mit $\text{src}(e) < \text{tgt}(e)$ für alle Kanten $e \in E_G$ und (V_P, E_P) erfüllt somit die Forderungen eines PBQP-Graphen. Während ein Programmgraph mehrere Kanten mit gleichem Quell- und Zielknoten enthalten kann, werden diese Kanten im PBQP-Graph zusammengefasst um den Knotengrad zu reduzieren. Knoten dagegen werden 1:1 abgebildet.

Zur Formulierung des PBQP gilt es außerdem für jeden Knoten einen Vektor an Alternativen mit verschiedenen Kosten zu erstellen, wobei äquivalente Musterteile zu jeweils einer Alternative zusammengefasst werden. Zu jeder Überdeckung $U \in \mathcal{U}_v$ eines Knotens v konstruieren wir eine Alternative

$$A_U = \bigcup_{U' \in \mathcal{U}_v} \{(M', \iota') \in U' \mid \exists (M, \iota) \in U : \iota^{-1}(v) \sim \iota'^{-1}(v)\}$$

und erhalten somit die Menge

$$\mathcal{A}_v = \bigcup_{U \in \mathcal{U}_v} \{A_U\}$$

aller Alternativen des Knotens v, die auch als Partition der Elemente aller Überdeckungen aus $\mathcal{U}_v$ aufgefasst werden kann. Jede Alternative besitzt endliche Kosten, die durch ein Kostenmodell und/oder eine Spezifikation geliefert werden. Diese Kosten werden dem Wurzelknoten des Musters zugeschrieben, da dieser nach Definition 15 zu keinem anderen Knoten äquivalent ist. Alle anderen Knoten des Musters haben Kosten 0.

Zur Definition der Kostenmatrizen fixieren wir ein Element (M_u, ι_u) der Alternative A_u des Quellknotens u und ein Element (M_v, ι_v) der Alternative A_v des Zielknotens v. Der zugehörige Eintrag der Matrix ergibt sich zu

$$c(A_u, A_v) = \begin{cases} \infty & \text{typ}(\iota_u^{-1}(v)) = \bot \wedge \iota_v^{-1}(v) \neq \text{rt}(M_v) \\ \infty & \text{typ}(\iota_u^{-1}(v)) \neq \bot \wedge \iota_u^{-1}(v) \nsim \iota_v^{-1}(v) \\ 0 & \text{sonst} \end{cases} \tag{4.4}$$

Der erste Fall behandelt Mustergrenzen. Der zweite Fall sorgt für Konsistenz innerhalb eines Musters, so dass alle oder kein Knoten eines Musters ersetzt werden. In Abbildung 4.3 ist ein grafisches Beispiel gegeben. Im oberen Teil sind drei Muster gegeben, die den Programmgraphen darunter überdecken sollen. Der Programmgraph enthält einen leeren Knoten, den wir für dieses Beispiel ignorieren werden. Das Add-Muster werde zweimal als Urbild benutzt, die Add+Const- und Const-Muster jeweils einmal. Die Überdeckung ist der Übersichtlichkeit halber als Schraffur dargestellt, statt wie in Abbildung 4.2 als Pfeile. Die Kosten der einzelnen Muster sind extern gegeben und in den Kostenvektoren eingetragen. Betrachten wir nun die Kostenmatrix. Die beiden ∞ Einträge der zweiten Spalte stammen aus dem ersten Fall, denn das Add-Muster verlangt eine Wurzelalternative an diesem Knoten. Der ∞ Eintrag in der ersten Spalte stammt aus dem zweiten Fall, denn eine Wurzelalternative ist nicht kompatibel mit dem Add+Const-Muster. Es wird sichergestellt, dass in einer gültigen PBQP-Auswahl dieses Muster entweder an beiden oder an keinem Knoten ausgewählt wird.

PBQP-Instanzen, die nach dieser Methode aus Firm-Programmgraphen konstruiert werden, bilden eine Unterklasse des PBQP, die wir als „PBQP einer Befehlsauswahl“ kurz PB bezeichnen wollen. An dieser Stelle wollen wir einige wichtige Eigenschaften der PB-Instanzen festhalten.

Lemma 4. *Sei G ein Programmgraph, $\mathcal{M}$ eine Mustermenge und $P = \pi_{\mathcal{M}}(G)$ die zugehörige PB-Instanz. Weiter sei $v \in V_P$ ein Knoten und A_v eine Alternativen von v. Für jedes Paar von Elementen $(M, \iota) \in A_v$ und $(M', \iota') \in A_v$ gilt $\iota^{-1}(v) \sim \iota'^{-1}(v)$.*

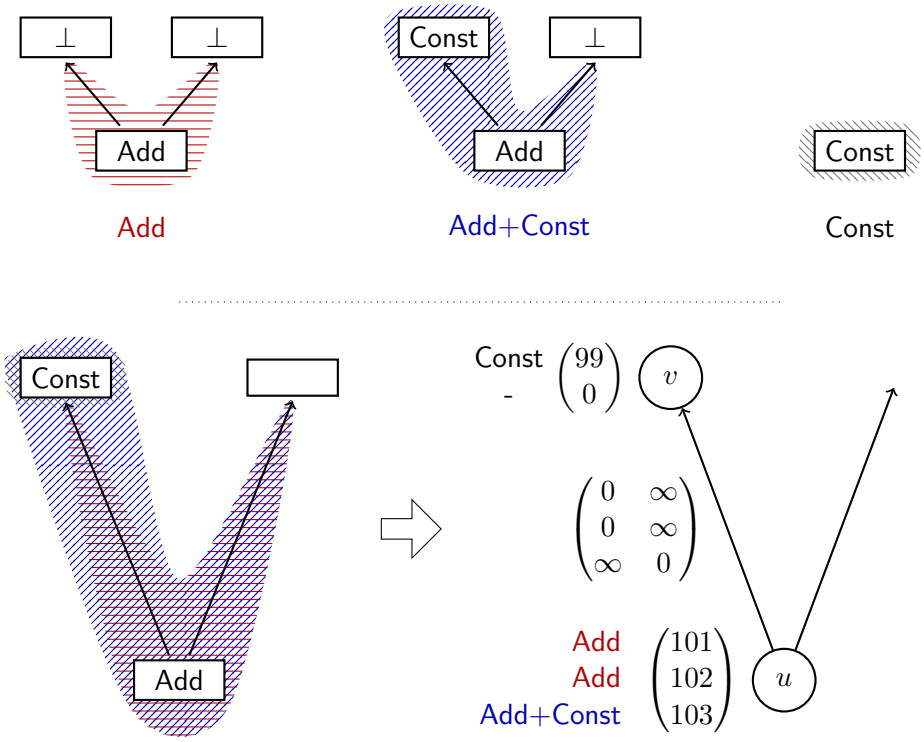


Abbildung 4.3: Abbildung einer Befehlsauswahl auf PBQP.

Beweis: Sei $U \in \mathcal{U}_v$ eine Überdeckung mit $A_U = A_v$. Wegen Gleichung 4.2 gilt die Behauptung für alle Alternativen $(M_v, \iota_v) \in U$ und $(M'_v, \iota'_v) \in U$. Nach Definition von A_U existieren Elemente $(M_v, \iota_v), (M'_v, \iota'_v) \in U$ mit $\iota^{-1}(v) \sim \iota_v^{-1}(v)$ und $\iota'^{-1}(v) \sim \iota_v'^{-1}(v)$. Aus der Transitivität von $\sim$ folgt somit $\iota^{-1}(v) \sim \iota'^{-1}(v)$. □

Lemma 5. *Sei G ein Programmgraph, $\mathcal{M}$ eine Mustermenge und $P = \pi_{\mathcal{M}}(G)$ die zugehörige PB-Instanz. Weiter sei $v \in V_P$ ein Knoten und $A_v \neq A'_v$ zwei verschiedene Alternativen von v. Für jedes Paar von Elementen $(M, \iota) \in A_v$ und $(M', \iota') \in A'_v$ der beiden Alternativen gilt $\iota^{-1}(v) \nsim \iota'^{-1}(v)$.*

Beweis: Wir nehmen das Gegenteil an, d.h. es existieren zwei Elemente $(M, \iota) \in A_v$ und $(M', \iota') \in A'_v$ mit $\iota^{-1}(v) \sim \iota'^{-1}(v)$. Mit Lemma 4 und der Transitivität von $\sim$ folgt $\iota_v^{-1}(v) \sim \iota_v'^{-1}(v)$ für alle Elemente $(M_v, \iota_v) \in A_v$ und $(M'_v, \iota'_v) \in A'_v$. Dies ist allerdings äquivalent zu $A_v = A'_v$ und steht somit im Widerspruch zur Annahme. □

4.2.1 Beispiel

Als Beispiel sei der Ausschnitt eines Programmgraphen in Abbildung 4.4 gegeben, der zwei Ladeoperationen von derselben Adresse zeigt. Die zusätzlichen Kanten oh-

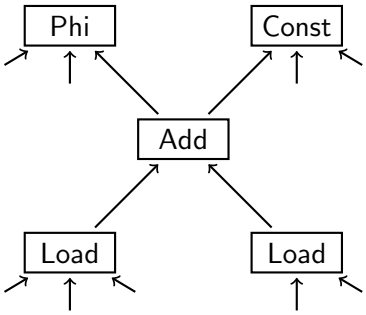


Abbildung 4.4: Ausschnitt eines Programmgraphen.

ne Quellknoten kennzeichnen Übergänge zum Rest des Programmgraphen. Für den Aufbau des PB sind diese Kanten zu ignorieren, aber wir werden diesen Beispielgraphen auch zur Demonstration des Lösungsalgorithmus verwenden und dort sind diese Kanten notwendig. In Abbildung 4.5 ist eine Mustermenge abgebildet, mit der eine Überdeckung des Programmgraphen gefunden werden soll. Die untere Zeile

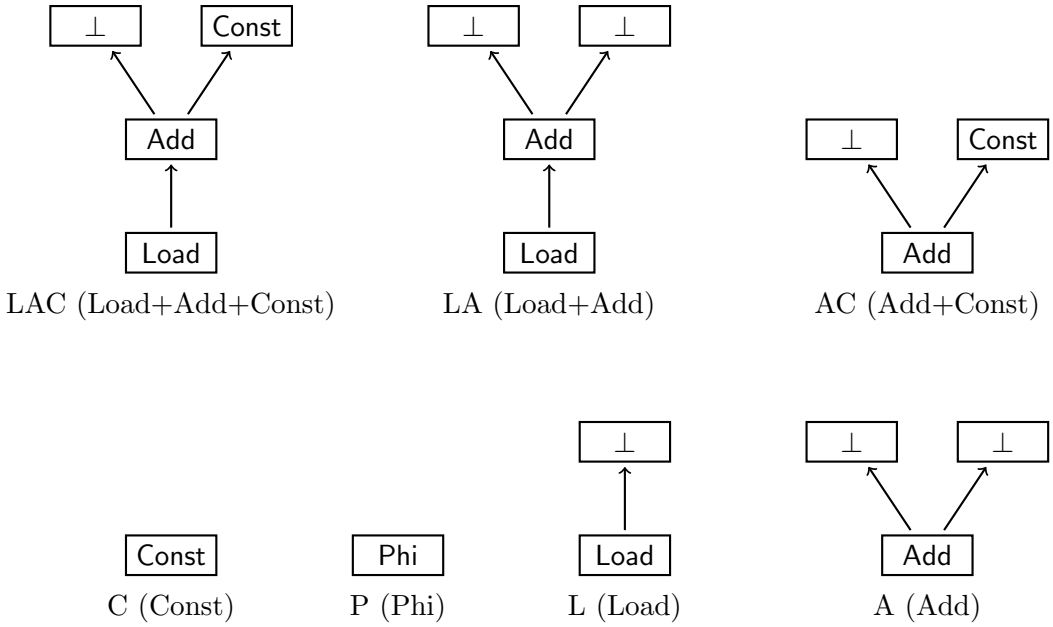


Abbildung 4.5: Mustermenge.

zeigt atomare Muster für Const-, Phi-, Load- und Add-Knoten. Darüber sind komplexere Muster für Lade- und Addieroperationen zu sehen. Alle diese Muster lassen sich in der IA-32 Architektur durch einen Maschinenbefehl ausdrücken.

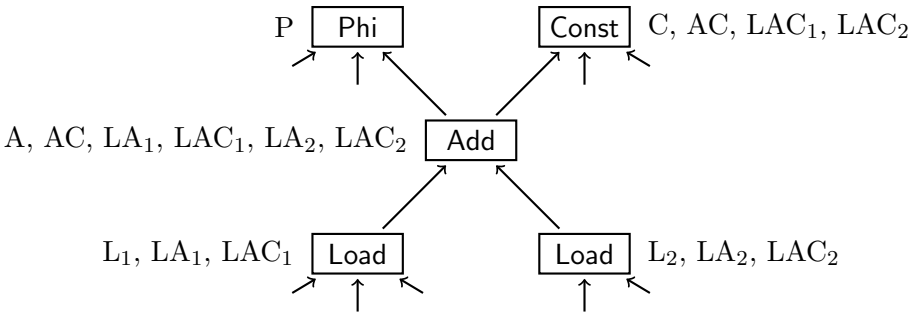


Abbildung 4.6: Programmgraph aus Abbildung 4.4 mit gefundenen Mustern.

Abbildung 4.6 zeigt, wie die Muster im Programmgraphen gefunden wurden. Neben jedem Knoten ist das Muster annotiert, als dessen Teil der Knoten identifiziert wurde. Diese Schreibweise ist nicht eindeutig, aber für dieses Beispiel ausreichend. Die Muster L, LA und LAC wurden je zweimal gefunden und deswegen mit Indizes markiert. Das atomare Muster C ist nur am Const-Knoten eingetragen, da das Muster nur aus einem Knoten besteht. Das Muster LAC umfasst drei Knoten, da untypisierte Knoten nicht mitgerechnet werden. Die drei Knoten der ersten Fundstelle dieses Musters wurden alle mit LAC_1 annotiert. Der Add-Knoten wird als Teil von sechs verschiedenen Mustern gefunden und besitzt somit die meisten vermerkten Funde.

Um nun den in Abbildung 4.7 gezeigten PB-Graphen zu konstruieren, werden die Knoten und Kanten aus dem Programmgraphen übernommen. Weiter werden die annotierten Funde zu Alternativen zusammengefasst und zu jeder Alternative existiert ein Eintrag im Kostenvektor des entsprechenden Knotens. Im Add-Knoten wurden die beiden Funde von LA und LAC jeweils zu einer Alternative zusammengefasst, da die aufgespannten Graphen isomorph sind. Genauso bilden am Const-Knoten die AC und LAC eine Alternative. Die Kostenmatrizen wurden entsprechend der Vorschrift in Gleichung 4.4 mit 0 beziehungsweise ∞ belegt. Die Kante zum Phi-Knoten kann direkt wieder entfernt werden, da sie unabhängig ist und keine Einschränkung aufweist. Da der Phi-Knoten nur eine Alternative besitzt, kann auch er sofort entfernt werden.

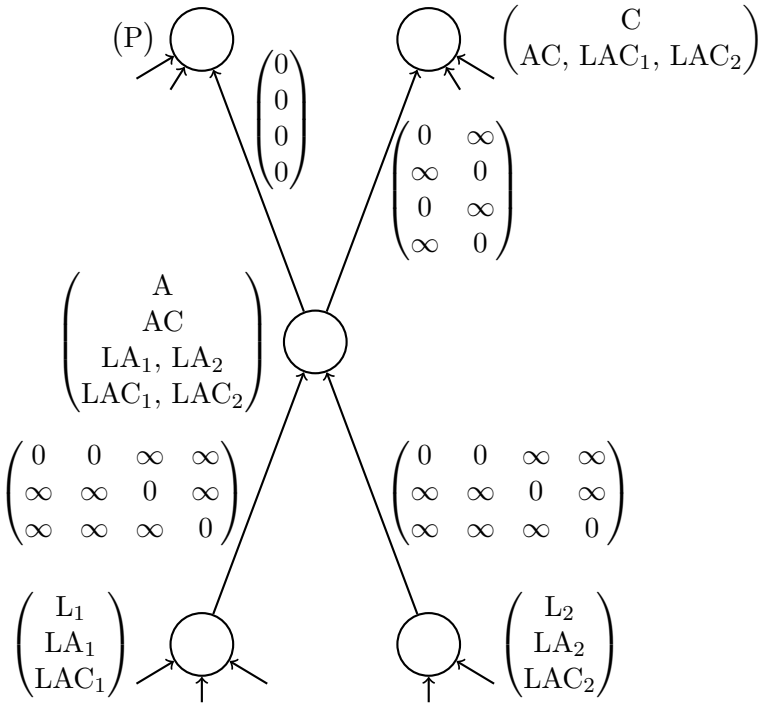


Abbildung 4.7: PB-Graph konstruiert aus Abbildung 4.6.

4.3 Kostenmodell

Das Lösen eines PBQP geschieht unter dem Gesichtspunkt der Kostenminimierung. Während die ∞-Einträge der Kostenmatrizen Korrektheit im Sinne der Befehlssemantik sicherstellen und somit festgelegt sind, können die Vektor-Einträge beliebige endliche Kosten annehmen. Dabei bietet es sich an, für jedes Muster feste Grundkosten zu vergeben, die abhängig von der Struktur des Programmgraphs angepasst werden, um eine effizientere Befehlsauswahl zu treffen.

Man bedenke folgendes Szenario: Eine Adressberechnung besitzt 31 Load-Verwender, von denen 30 in einem gemeinsamen Grundblock angeordnet sind. Es wäre nun sinnvoll, dass der einsame 31te Verwender die Adressberechnung in sich „hineinfrisst“, während für die anderen Verwender die Adresse einmal ausgerechnet und anschließend gemeinsam verwendet wird.

Es ist schwer, allgemein gültige Regeln für eine „gute“ Befehlsauswahl aufzustellen, da sich auf modernen Rechnern die Laufzeiten einzelner Befehle nicht mehr vorhersagen lassen. Auch sind die Ansichten über „gut“ nicht eindeutig. In manchen Fällen ist die minimale Codegröße wichtiger, in anderen Fällen die Ausführungsgeschwindigkeit. Mögliche Optimierungsziele sind:

Minimale Befehlsanzahl Für Zielarchitekturen, die komplexere Befehle anbieten ist es häufig sinnvoll möglichst komplexe und große Muster auszuwählen, da so viele Operationen zusammengefasst und mögliche Optimierungen des Prozessors genutzt werden können. Zum Beispiel bieten viele Architekturen eine Vielzahl an Möglichkeiten, die Adressberechnung in Operationen hineinzuziehen.

Minimale Befehlslänge Meist konkurrierend zum ersten Punkt ist die Befehlslänge. Falls eine Architektur variable Befehlslänge anbietet, kann eine Minimierung der Befehlslänge dazu führen, dass gerade kritische, innere Schleifen kürzer werden und dann beispielsweise komplett in eine Zeile des schnellen Zwischenspeichers[1] passen. Da ein Fehlzugriff in den schnellen Zwischenspeicher einen relativ langsamen Zugriff auf den Hauptspeicher nach sich zieht, kann diese Optimierung durchaus Sinn haben.

Diese Optimierung steht allerdings im Konflikt mit der vorgehenden Minimierung der Befehlszahl, denn meist sind komplexe Befehle länger als einfache. Es ist also zu entscheiden, ob mehrere kurze Befehle oder ein langer Befehl zu einem besseren Ergebnis führt.

Minimaler Registerdruck Viele Prozessoren besitzen nur wenige allgemein verwendbare Register, und eine Verringerung des Registerdrucks kann sich gerade in inneren Schleifen stark auf die Laufzeit auswirken. Eine Optimierung in diesem Sinne ist allerdings nur schwer möglich, da die Befehle vor der Registerzuteilung und insbesondere auch vor der Befehlsanordnung ausgewählt werden. Zu diesem Zeitpunkt sind noch keine konkreten Informationen vorhanden, um eine sichere Entscheidung zu treffen.

Ist ein Knoten eines Grundblocks der letzte Verwender *aller* seiner Argumente und wird von Knoten eines anderen Grundblocks konsumiert, so hat der für diesen Knoten ausgewählte Befehl Einfluss auf den Registerdruck. In diesem Fall ist eine Materialisierung, also das explizite Ersetzen des Knotens durch einen Befehl, einer Konsumtion in einen komplexeren Befehl vorzuziehen, da so nur ein Register für das Ergebnis benötigt wird.

Minimale Registereinschränkungen Eine Befehlsauswahl, die Befehle mit weniger Registereinschränkungen generiert, gibt der späteren Registerzuteilung mehr Freiraum und somit mehr Möglichkeit zur Optimierung. Für IA-32 bedeutet das zum Beispiel statt einer `add`-Operation eine `lea`-Operation zu verwenden.

Minimaler Stromverbrauch Im Bereich eingebetteter Systeme spielt der Stromverbrauch oft eine wichtige Rolle. Ein Kostenmodell kann Befehle, die für überdurchschnittlichen Stromverbrauch bekannt sind, entsprechend abwerten.

[1] engl.: cache line

4.4 Lösen eines PBQP

Dieser Abschnitt zeigt einen generellen Algorithmus zur Lösung eines PBQP. Er ist weitestgehend von Eckstein und Scholz[SE02] übernommen. Allerdings ist die Beschreibung des Algorithmus in deren Veröffentlichung abschnittsweise auf Registerzuteilung ausgelegt. Diesen Teil haben wir entfernt und außerdem die in Abschnitt 3.4 beschriebene Schwäche behoben, die dazu führte, dass eine ungültige Auswahl in einem Kostenvektor von der Heuristik nicht als solche erkannt wurde.

Das Lösen eines PBQP geschieht in drei Phasen, wie in Abschnitt 2.4.3 bereits beschrieben:

1. Der Graph wird reduziert, bis keine Kanten mehr vorhanden sind.

2. Ohne Kanten kann an jedem Knoten lokal das Minimum ausgewählt werden.

3. Durch Rückwärtspropagierung werden die im ersten Schritt entfernten Knoten und Kanten wieder eingefügt und Alternativen ausgewählt.

Der schwierige Teil ist das Finden eines guten Reduktionsalgorithmus. Das Finden einer Lösung eines PBQP ist ein NP-vollständiges Problem. Damit die Laufzeit trotzdem akzeptabel bleibt, verwenden wir an bestimmten Stellen eine Heuristik.

Für den Algorithmus sei $G(V, E)$ ein ungerichteter PBQP-Graph mit Kanten, wie in Abschnitt 2.4 beschrieben. Das Ergebnis soll ein Auswahlvektor $\vec{s}$ sein, der jedem Kostenvektor $\vec{c}_x$ die ausgewählte Alternativenposition s_x zuordnet.

In Algorithmus 1 ist unsere modifizierte Reduktion zu sehen. Der Algorithmus beginnt mit dem Aufruf von ReduceGraph. Zuerst werden alle Knoten entsprechend ihrem Knotengrad sortiert. Dann werden – solange wie möglich – einfache Reduktionen (Reduce1, Reduce2) ausgeführt. Sobald keine Kanten mehr existieren, ist die erste Phase abgeschlossen und die Reduktion beendet.

Tritt der Fall ein, dass keine einfache Reduktion mehr möglich ist, aber dennoch Kanten vorhanden sind, muss die heuristische Reduktion ReduceN verwendet werden. Diese rechnet, auf direkte Nachbarn beschränkt, die Folgekosten für jede Auswahl s_x aus und wählt das Minimum. Aufgrund dieser Auswahl werden die Kostenvektoren und -matrizen entsprechend angepasst und nun ungültige Einträge entsprechend markiert. Dieser Markierungsschritt ist unsere Erweiterung zu dem Algorithmus von Scholz, da die Relax-Prozedur ungültige Einträge im ganzen PBQP-Graphen statt nur an den anliegenden Matrizen markiert. Außerdem werden in diesem Schritt unabhängige Kanten entfernt, wodurch in manchen Fällen weitere heuristische Reduktionen vermieden werden können.

Algorithmus 1 Reduktionen

```
1:  procedure REDUCE1(x)
2:      {y} ← adj(x)
3:      for i ← 1 ... |c_y⃗| do
4:          Δ⃗(i) ← min(C_yx(i,:) + c_x⃗)
5:      c_y⃗ ← c_y⃗ + Δ⃗
6:      PushVertex(x)

7:  procedure REDUCE2(x)
8:      {y, z} ← adj(x)
9:      for i ← 1 ... |c_y⃗| do
10:         for j ← 1 ... |c_z⃗| do
11:             a⃗ ← C_yx(i,:) + C_xz(:,j)
12:             Δ(i,j) ← min(a⃗ + c_x⃗)
13:     if (y, z) ∈ E then
14:         C_yz ← C_yz + Δ
15:     else
16:         add edge (y, z)
17:         C_yz ← Δ
18:     normalize edge (y, z)
19:     PushVertex(x)

20: procedure RELAX(x)
21:     for all y ∈ adj(x) do
22:         for all i ← 0 ... |c_x⃗| do
23:             m ← min(C_xy(i,:))
24:             c_x(i) ← c_x(i) + m
25:             if m < ∞ then
26:                 a⃗ ← C_xy(i,:) − m⃗
27:                 C_xy(i,:) ← a⃗
28:     S ← {i | 0 ≤ i < |c_x⃗| ∧ c_x(i) = ∞}
29:     for all y ∈ adj(x) do
30:         for all i ∈ S do
31:             C_xy(i,:) ← ∞⃗
32:         normalize edge (x, y)
33:         if (x, y) is independent then
34:             remove edge (x, y)
35:         if C_xy changed then
36:             RELAX(y)

37: procedure REDUCEN(x)
38:     for i ← 1 ... |c_x⃗| do
39:         c⃗(i) ← 0
40:         for all y ∈ adj(x) do
41:             a⃗ ← C_xy(i,:) + c_y⃗
42:             c⃗(i) ← c⃗(i) + min(a⃗)
43:     s_x ← i_min(c⃗)
44:     for i ← 1 ... |c_x⃗| do
45:         if i ≠ s_x then
46:             c_x(i) ← ∞
47:     RELAX(x)
48:     PushVertex(x)

49: procedure REDUCEGRAPH
50:     for all x ∈ V do
51:         RELAX(x)
52:         insert x into bucket(deg(x))
53:     normalize all edges
54:     while vertices left do
55:         if ∃x ∈ bucket(1) then
56:             REDUCE1(x)
57:         else if ∃x ∈ bucket(2) then
58:             REDUCE2(x)
59:         else
60:             select vertex x
61:             REDUCEN(x)
```

Jede Reduktion endet mit der Anweisung PushVertex(x), welche dafür sorgt, dass der Knoten aus dem Graphen entfernt und in einem Reduktionskeller gespeichert wird. In der dritten Phase werden die Knoten von dort im Zuge der Rückwärtspropagation wieder in den Graphen eingefügt. Dieses Vorgehen garantiert ein Terminieren des Algorithmus, da in jedem Schritt der Graph um einen Knoten verkleinert wird.

Lemma 6. *Sei P eine PBQP-Instanz und $v \in G_P$ ein Knoten mit Kostenvektor $\vec{v}$. Weiter sei $\vec{e}$ die durch die Alternative $A_v \in \mathcal{A}_v$ ausgewählte Zeile bzw. Spalte der Kostenmatrix einer inzidenten Kante. Dann gilt nach jeder Reduktion des Algorithmus*

$$c(A_v) = \infty \Leftrightarrow \vec{e} = \vec{\infty}.$$

Beweis: Da zu Beginn des Algorithmus für jeden Knoten die RELAX-Prozedur aufgerufen wird, gilt die Behauptung vor allen Reduktionen. $R1$- und $R2$-Reduktionen machen keine Einträge oder Kombinationen ungültig, also bleibt die Behauptung nur für RN-Reduktionen zu beweisen. Werde also REDUCEN(v) aufgerufen und ein Eintrag $c(A_v) = \infty$ gesetzt. In REDUCEN(v) wird RELAX(v) aufgerufen und dadurch gilt $\vec{e} = \vec{\infty}$. Falls in RELAX(v) $\vec{e} = \vec{\infty}$ gesetzt wurde, so war bereits $c(A_v) = \infty$ und für den anderen Endknoten v' der Kante wird RELAX(v') aufgerufen, wodurch auch $c(A_{v'}) = \infty$ gilt. Durch die rekursiven Aufrufe von RELAX gilt die Behauptung ebenfalls für v' und alle anderen Knoten. □

4.4.1 Beispiel

Wir wollen nun das Beispiel von Abbildung 4.7 weiterführen, um die Lösung des PBQP durch Algorithmus 1 beziehungsweise REDUCEN zu demonstrieren. Der Programmgraph wurde so gewählt, dass keine $R1$- oder $R2$-Reduktion möglich ist, wir müssen also heuristisch reduzieren. Der Phi-Knoten wurde entfernt, da für eine Alternative keine Auswahl nötig ist. Wir wählen den Knoten 1, da dieser den höchsten Knotengrad besitzt und maximal viele Kanten durch die Reduktion entfernt werden. Wir haben keine Kosten vorgeschrieben. Sei LAC_1 die günstigste Wahl für dieses Beispiel, also $i_{min}(\vec{c}) = 2$, wobei 2 die Position von LAC_1 ist.

Nach dieser Auswahl wird REDUCEN alle anderen Kosten-Einträge von Knoten 1 auf ∞ setzen. Unendliche Kosten für eine Alternative werden mit einem durchgestrichenen Vektor-Eintrag markiert. Die anschließend aufgerufene RELAX-Prozedur addiert die Minima der Zeilen der anliegenden Kostenmatrizen in den Kostenvektor von Knoten 1, allerdings sind in diesem Fall alle Minima Null. In einem zweiten Schritt wird für jeden Eintrag im Kostenvektor $c_x(i) = \infty$ die entsprechende Zeile der anliegenden Kostenmatrizen zu $\vec{\infty}$ gesetzt und RELAX auf dem Nachbarknoten aufgerufen, also unter anderem auch auf Knoten 3. Das resultierende PBQP ist in

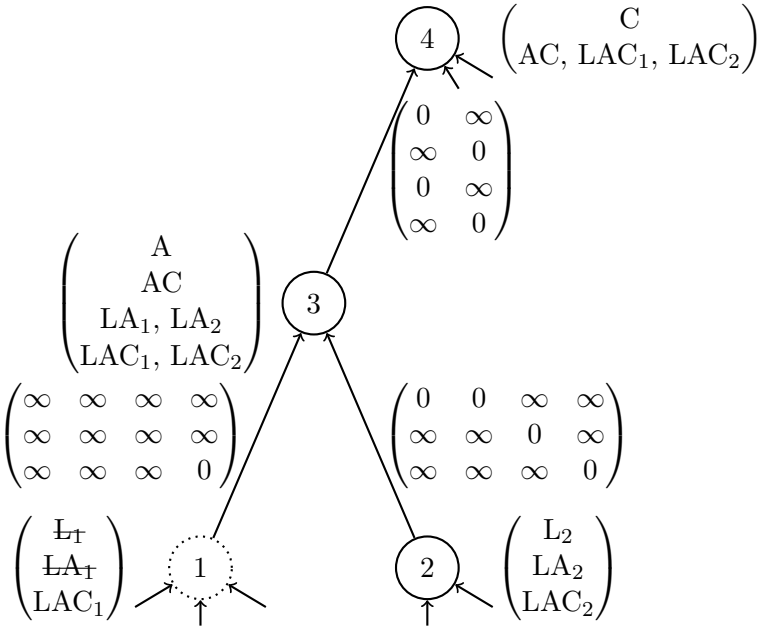


Abbildung 4.8: PB-Graph aus Abbildung 4.7 während des ersten heuristischen Reduktionsschritts nach dem ersten Durchlauf von Relax.

Abbildung 4.8 abgebildet. Der rekursive Aufruf propagiert ∞ Einträge über den Graphen. Der Relax-Aufruf am Nachbarknoten wird im ersten Schritt ∞ als Minimum feststellen und den Kostenvektor entsprechend einschränken. Anschließend werden diese Einträge auf die anliegenden Kostenmatrizen gerechnet und die Einschränkung des Lösungsraums verbreitet sich über den Graphen. Nach Ausführung von Relax wird der Knoten entfernt und der Graph sieht aus wie in Abbildung 4.9. Wie man sieht, wurden auch die Knoten 2, 3 und 4 in ihren Alternativen eingeschränkt. Für diesen Ausschnitt des Programmgraphen ist damit die Belegung festgelegt. Weitere Reduktionen werden für Knoten 2 das LAC_2 Muster auswählen, da diese als einzige Alternative noch endliche Kosten hat.

4.5 Garantie einer gültigen Lösung

Dieser Abschnitt soll zeigen, dass durch die Verwendung eines PBQP-Lösungsprogramms für gewisse Mustermengen $\mathcal{M}$ eine gültige Befehlsauswahl garantiert werden kann, selbst wenn heuristische Entscheidungen getroffen werden müssen. Dazu zeigen wir zunächst, dass aus jeder Überdeckung eines Programmgraphen G eine Lösung der zugehörigen PB-Instanz $\pi_{\mathcal{M}}(G)$ konstruiert werden kann. Zur Veranschaulichung des Beweises ist in Abbildung 4.10 ein überdeckter Programmgraph und die dazu-

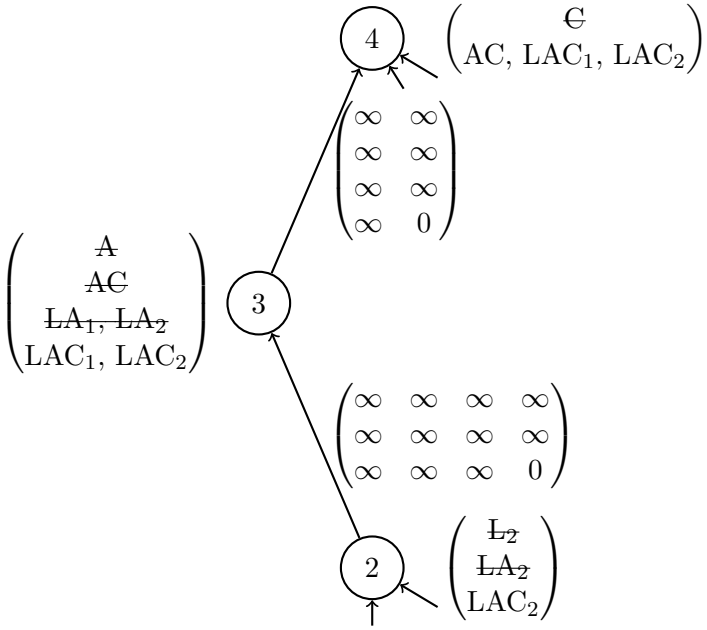


Abbildung 4.9: PB-Graph aus Abbildung 4.8 nach der Reduktion.

gehörige PBQP-Lösung dargestellt. Die Überdeckung ist durch die Muster AC_1, AC_2 und C_2 aus Abbildung 4.5 gegeben, die durch Schraffur markiert sind. Die Lösung des PBQP besteht aus den Vektor-Einträgen, die nicht durchgestrichen sind.

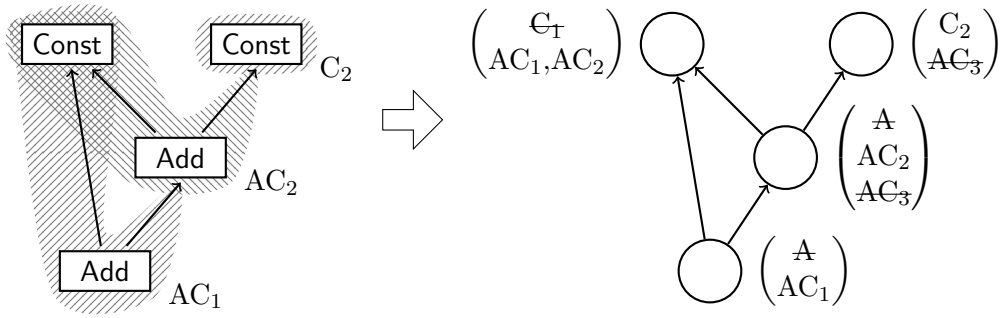


Abbildung 4.10: Von einer Überdeckung zur Lösung.

Satz 1. *Sei G ein Programmgraph und $U \in \mathcal{U}_G$ eine Überdeckung von G. Weiter sei $P = \pi(G)$ die zugehörige PB-Instanz und $L(P)$ die Menge der Lösungen von P. Dann ist Ψ, definiert durch*

$$\Psi(U)(v) \in \{A \in \mathcal{A}_v \mid \exists (M, \iota) \in U : (M, \iota) \in A\},$$

eine Abbildung $\Psi : \mathcal{U}_G \to L(P)$.

Beweis: Wegen Gleichung 4.2 und Lemma 5 ist $\Psi(U)(v)$ eindeutig bestimmt, es wird also genau eine Alternative ausgewählt. Um zu zeigen, dass $\Psi(U)$ eine Lösung der PB-Instanz ist, genügt es nachzuweisen, dass alle gewählten Matrix-Einträge endlich sind. Seien also $(M_u, \iota_u) \in A_u$ und $(M_v, \iota_v) \in A_v$ Elemente der zwei gewählten Alternativen an adjazenten Knoten u und v. Dabei sei u der Quellknoten der Kante zwischen u und v. Wir betrachten nun die beide Fälle, in denen der Matrix-Eintrag unendlich ist.

1. **Fall:** $\text{typ}(\iota_u^{-1}(v)) = \perp \wedge \iota_v^{-1}(v) \neq \text{rt}(M_v)$**.** Sei also $\text{typ}(\iota_u^{-1}(v)) = \perp$. Da G vollständig typisiert ist, gilt $\text{typ}(v) \neq \perp$. Nach Gleichung 4.3 existiert also ein $(M', \iota') \in U$ mit $\iota'(\text{rt}(M')) = v$. Es folgt $\Psi(U)(v) = \{(M', \iota')\}$, also $(M_v, \iota_v) = (M', \iota')$ und somit $\iota_v^{-1}(v) = \text{rt}(M_v)$. Dieser Fall kann demnach nicht eintreten.
2. **Fall:** $\text{typ}(\iota_u^{-1}(v)) \neq \perp \wedge \iota_u^{-1}(v) \nsim \iota_v^{-1}(v)$**.** Wir nehmen an $\text{typ}(\iota_u^{-1}(v)) \neq \perp$, außerdem wissen wir bereits, dass $\text{typ}(\iota_v^{-1}(v)) \neq \perp$. Damit folgt aber nach Gleichung 4.2, dass $\iota_u^{-1}(v) \sim \iota_v^{-1}(v)$ gilt, wodurch auch dieser Fall nicht eintreten kann.

Insgesamt ist $\Psi(U)$ also eine Lösung von P. □

Der Satz ermöglicht es uns den Großteil der Beweise auf der Ebene der Überdeckung zu führen und nur in wenigen Fällen die konstruierte PB-Instanz direkt zu betrachten.

4.5.1 Existenz einer Lösung

Um zu zeigen, dass eine Lösung gefunden werden kann, müssen wir zuerst einmal die Existenz einer Überdeckung nachweisen. Dazu müssen wir einige Forderungen an die Menge der Mustergraphen stellen.

Definition 18 (Atomar-vollständige Menge von Mustergraphen)**.** Eine Menge $\mathcal{M}$ von Mustergraphen heißt **atomar-vollständig**, wenn für jeden Typ $t \in \Sigma_{EAG}$ ein atomarer Mustergraph $M \in \mathcal{M}$ mit $\text{typ}(\text{rt}(M)) = t$ existiert.

Korollar 3 (Einbettung atomarer Muster)**.** *Sei G ein beliebiger Graph und $v \in V_G$ ein Knoten mit $\text{typ}(v) = t \in \Sigma_{EAG}$. Weiter sei M ein atomarer Mustergraph mit $\text{typ}(\text{rt}(M)) = t$. Dann existiert ein injektiver Morphismus $\iota : M \to G$ mit $\iota(\text{rt}(M)) = v$.*

Ein Beispiel einer atomaren Überdeckung ist in Abbildung 4.11 zu sehen. Durch atomare Muster aus Abbildung 4.5 kann der gesamte Programmgraph überdeckt werden. Jedes Muster überdeckt immer genau den Wurzelknoten und keinen Nachfolger.

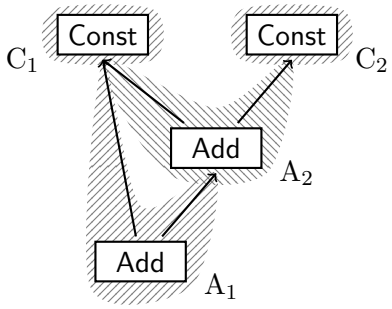


Abbildung 4.11: Atomare Überdeckung eines Programmgraphen.

Satz 2 (Existenz einer Überdeckung)**.** *Jeder Programmgraph G besitzt mindestens eine Überdeckung durch eine atomar-vollständige Menge $\mathcal{M}$ von Mustergraphen.*

Beweis: Da $\mathcal{M}$ atomar-vollständig ist, existiert eine Abbildung $\Psi : \Sigma_{EAG} \to \mathcal{M}$, die jedem Typ einen festen atomaren Mustergraphen mit $\text{typ}(\text{rt}(\Psi(t))) = t$ zuordnet. Zu jedem Knoten $v \in V_G$ sei $M_v = \Psi(\text{typ}(v))$. Nach Korollar 3 existiert eine Überdeckung $U_v = \{(M_v, \iota_v)\}$ von v mit $\iota_v(\text{rt}(M_v)) = v$. Wir wollen nun zeigen, dass

$$U_G = \bigcup_{v \in V} U_v$$

eine Überdeckung von G ist. Da G vollständig typisiert ist, besteht die in Definition 17 konstruierte Menge U_v für jeden Knoten $v \in V_G$ ausschließlich aus dem Element (M_v, ι_v), woraus direkt Gleichung 4.1 und Gleichung 4.2 folgen. Weiter existiert für jeden Knoten $v \in V_G$ eine Alternative $\{(M_v, \iota_v\}$ mit $\iota_v(\text{rt}(M_v)) = v$, weshalb auch Gleichung 4.3 erfüllt ist. □

4.5.2 Finden einer Lösung

Aus dem vorherigen Abschnitt wissen wir, dass mindestens eine Lösung für die konstruierten PB-Instanzen existiert. Allerdings kann sich die Zahl der möglichen Lösungen durch die Reduktionen des Lösungsalgorithmus vermindern. Wir untersuchen deshalb in diesem Abschnitt die Auswirkungen der Reduktionen auf die Lösbarkeit der PB-Instanzen.

Wie wir bereits in Abschnitt 2.4.3 gezeigt haben, erhalten $R1$- und $R2$-Reduktionen stets die Lösungseigenschaft einer PBQP-Auswahl und sind somit für diese Problematik unkritisch. Für die RN-Reduktionen müssen wir allerdings weitere Forderungen an die Mustermenge stellen, denn atomar-vollständig ist nicht hinreichend für eine korrekte Befehlsauswahl. In Abbildung 4.12 ist links ein problematisches Sze-

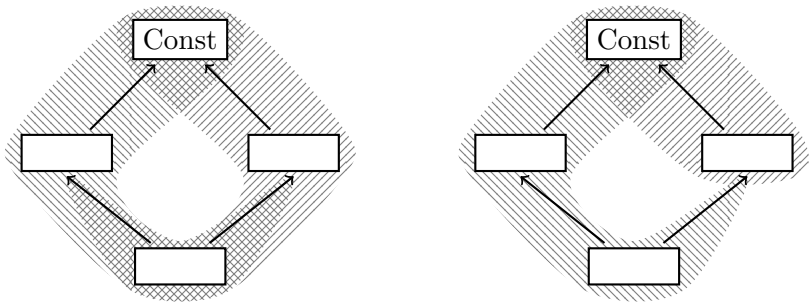


Abbildung 4.12: Kritischer Programmgraph zur Motivation der Kompositionalität.

nario abgebildet. Die beiden Muster bilden keine Überdeckung des Programmgraphen, da die Wurzel-Einträge verschiedene Alternativen sind und nicht gemeinsam ausgewählt werden dürfen. Die Auswahl einer Alternative, die diese beiden Muster enthält, ist so also ungültig. Allerdings ist jedes Muster für sich gültig, nur eben nicht beide gleichzeitig. An dem Const-Knoten ist eine Alternative auswählbar, die beide Muster enthält und eine *RN*-Reduktion könnte diese auswählen, ohne den Konflikt am Wurzelknoten zu beachten, da nur Nachbarknoten betrachtet werden. Eine atomare Überdeckung ist nach einer solchen Auswahl nicht möglich, da die Vorgänger des Const-Knotens nicht kompatibel mit der Auswahl sind. Rechts ist eine gültige Überdeckung aus zwei Mustern angegeben. Das kleinere Teilmuster ist ebenfalls Teil der Alternative und macht diese zu einer gültigen Auswahl. Würde dieses zusätzliche Muster nicht existieren, müssten an der Wurzel zwei verschiedene Befehle ausgewählt werden, was allerdings nicht möglich ist. Wenn eine Mustermenge zu jedem komplexen Muster auch die entsprechenden Teilmuster enthält, kann ein Problem solcher Art nicht auftreten. Solch eine Mustermenge bezeichnen wir als „kompositional“.

Definition 19 (Kompositionale Mustermenge)**.** Eine Mustermenge $\mathcal{M}$ heißt **kompositional**, wenn für jeden Mustergraph $M \in \mathcal{M}$ und jeden Knoten $v \in V_M$ mit $\text{typ}(v) \neq \perp$ eine Überdeckung U von M existiert, für die gilt:

$$\exists (M', \iota') \in U : \iota'(\text{rt}(M')) = v \wedge M' \cong G(v). \tag{4.5}$$

Außerdem muss für die Wurzel $\text{rt}(M)$ jedes Musters $M \in \mathcal{M}$ und für jeden Nachfolgeknoten $w \in \mathit{succ}(\text{rt}(M))$ eine Überdeckung $\{(M_w, \iota_w)\}$ von $\text{rt}(M)$ mit $M_w \in \mathcal{M}$ existieren, für die gilt:

$$\forall u \in \mathit{succ}(\text{rt}(M)) \setminus \{w\} : G(u) \cong G(\iota_w^{-1}(u)) \tag{4.6}$$

und

$$\text{typ}(\iota_w^{-1}(w)) = \perp . \tag{4.7}$$

Definition 19 lässt sich in zwei Hälften teilen. Zum einen wird in Gleichung 4.5 gefordert, dass für jeden aufgespannten Untergraph eines Musters ebenfalls ein Muster existiert. Als Beispiel ist Abbildung 4.13 gegeben, in welcher der von v aufgespannte Graph als Muster M' gefordert wird. In der zweiten Hälfte wird durch Gleichung 4.6

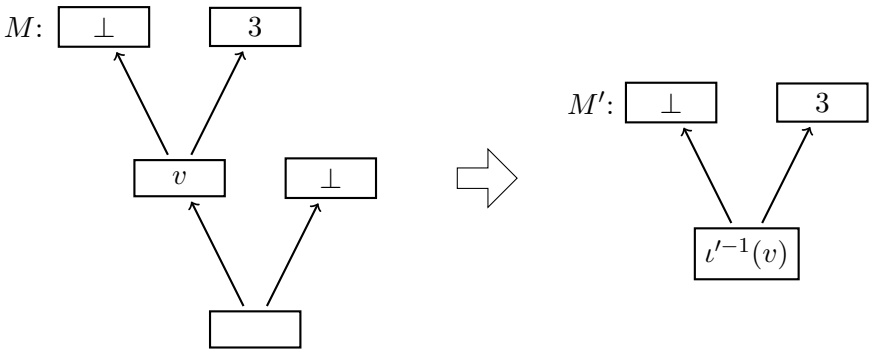


Abbildung 4.13: Illustration zu Gleichung 4.5.

und Gleichung 4.7 gefordert, dass beliebige Nachfolger des Wurzelknotens „abtrennbar“ sind, in dem Sinne, dass der entsprechende Untergraph durch einen $\perp$ Knoten ersetzt wird. In Abbildung 4.14 wird der Teilgraph, der von w aufgespannt wird, (der aus einem Knoten besteht) durch $\perp$ ersetzt. Für alle anderen Knoten soll das Muster M_w äquivalent sein, insbesondere ist der inverse Morphismus ι^{-1} für diese Knoten definiert.

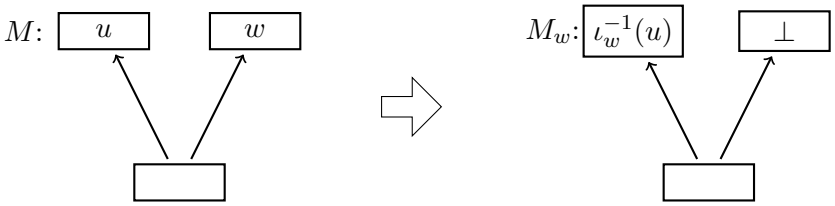


Abbildung 4.14: Illustration zu Gleichung 4.6 und Gleichung 4.7.

Lemma 7. *Sei $\mathcal{M}$ eine Mustermenge und G ein Programmgraph, für den eine Überdeckung mit Mustergraphen aus $\mathcal{M}$ existiert. Weiter sei (M, ι) ein Element der von einer RN-Reduktion ausgewählten Alternative $A_v \in \mathcal{A}_v$ der zugehörigen PB-Instanz $P = \pi_{\mathcal{M}}(G)$. Dann sind die Kosten zum Zeitpunkt der RN-Reduktion an jeder Alternative A mit $(M, \iota) \in A$ endlich.*

Beweis: Da G eine Überdeckung besitzt, existiert nach Satz 1 auch eine Lösung l von P, weshalb die Kosten der Alternative A_v endlich sind. Wir nehmen nun an, dass eine Alternative A mit $(M, \iota) \in A$ existiert, die unendliche Kosten besitzt. Da M als Mustergraph schwach zusammenhängend ist, existieren somit zwei Alternativen $A_u \ni (M, \iota)$ und $A_v \ni (M, \iota)$ adjazenter Knoten u und v, wobei A_u

unendliche und A_v endliche Kosten besitzt. Für jedes Element (M'_u, ι'_u) einer Alternative $A'_u \neq A_u$ von u gilt nach Lemma 5 $\iota'^{-1}_u(u) \nsim \iota^{-1}(u)$. Mit Gleichung 4.4 folgt, dass der Matrix-Eintrag zwischen A_v und A'_u im Falle $(v, u) \in E_P$ unendliche Kosten besitzt. Für den anderen Fall $(u, v) \in E_P$ gilt ebenfalls nach Lemma 5 $\iota'^{-1}_u(v) \nsim \iota^{-1}(v)$ und $\iota^{-1}(v) \neq \mathrm{rt}(M)$, da $u \in \mathrm{cod}(\iota)$, weshalb auch dieser Matrix-Eintrag nach Gleichung 4.4 unendliche Kosten besitzt. Insgesamt folgt, dass A_v mit keiner Alternative außer A_u kompatibel ist. Da A_u allerdings unendliche Kosten besitzt, folgt mit Lemma 6, dass auch A_v unendliche Kosten besitzt, was einen Widerspruch zur Voraussetzung darstellt. □

Im folgenden Lemma wollen wir zeigen, dass die Auswahl einer Wurzel eines komplexen Musters dazu führt, dass in ihrem aufgespannten Graph nur noch dieses Muster als Alternative zur Auswahl steht. Als Beispiel kann noch einmal Abbildung 4.9 betrachtet werden. An der Wurzel wurde LAC_1 ausgewählt und in den Knoten darüber, durch Anwendung von Relax, andere Alternativen als ungültig markiert.

Lemma 8. *Sei G ein Programmgraph, $\mathcal{M}$ eine Mustermenge und (M, ι) ein Element der von einer RN-Reduktion am Knoten u ausgewählten Alternative $A_u \in \mathcal{A}_u$ der zugehörigen PB-Instanz $P = \pi_{\mathcal{M}}(G)$. Weiter sei $v \in G(u)$ ein Knoten und $A_v \in \mathcal{A}_v$ eine Alternative mit $(M, \iota) \in A_v$. Dann sind die Kosten zum Zeitpunkt der nächsten RN-Reduktion an jeder Alternative $A'_v \neq A_v$ von v unendlich.*

Beweis: Da $v \in G(u)$ ist, existiert ein Pfad $(e_1, \ldots, e_n)$ von u nach v mit $\iota^{-1}(e_i) \in G(\iota^{-1}(u))$ für $i \in \{1, \ldots, n\}$. Da an u eine RN-Reduktion ausgeführt wurde, gilt die Behauptung für u. Wir zeigen die Behauptung mittels Induktion über den Pfad von u nach v. Sei also A_{v_i} eine Alternative von $v_i = \mathrm{src}(e_i)$ mit $(M, \iota) \in A_{v_i}$ und $A_{v_{i+1}}$ eine Alternative von $v_{i+1} = \mathrm{tgt}(e_i)$ mit $(M, \iota) \notin A_{v_{i+1}}$. Nach Voraussetzung gilt $\mathrm{typ}(\iota^{-1}(v_i)) \neq \bot$. Aus $(M, \iota) \notin A_{v_{i+1}}$ folgt mit Lemma 5, dass $\iota^{-1}(v_{i+1}) \nsim \iota'^{-1}(v_{i+1})$ für alle $(M', \iota') \in A_{v_{i+1}}$ gilt und der zugehörige Matrix-Eintrag nach Gleichung 4.4 unendliche Kosten besitzt. Da nach Induktionsvoraussetzung alle Alternativen $A'_{v_i} \neq A_{v_i}$ unendliche Kosten besitzen, gilt dies mit Lemma 6 auch für $A_{v_{i+1}}$. □

Als Nächsten zeigen wir, dass die Auswahl einer anderen Wurzelalternative keine Änderung für die Vorgänger darstellt. Eine solche Auswahl ist in Abbildung 4.15 verdeutlicht. Die Auswahl am Add-Knoten ändert sich von A zu AC, aber auf die Auswahl des Load-Knotens hat das keine Auswirkung.

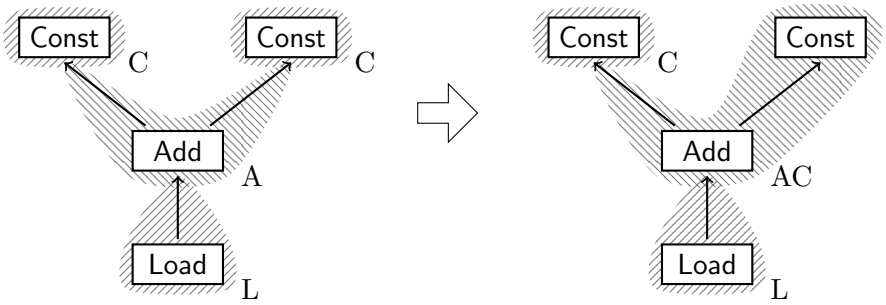


Abbildung 4.15: Reduktion eines Wurzelknotens.

Lemma 9. *Sei G ein Programmgraph und $e \in E_G$ eine Kante von $u = \mathrm{src}(e)$ nach $v = \mathrm{tgt}(e)$. Weiter sei $\mathcal{M}$ eine Menge von Mustergraphen, $U = U_G$ eine Überdeckung von G und A_u, A_v die durch die Lösung $l = \Psi(U)$ der PB-Instanz $\pi_{\mathcal{M}}(G)$ ausgewählten Alternativen von u bzw. v. Für $(M, \iota) \in A_v$ gelte $\iota(\mathrm{rt}(M)) = v$. Sei $A'_v \in \mathcal{A}_v$ eine weitere Alternative von v mit $\iota'(\mathrm{rt}(M')) = v$ für $(M', \iota') \in A'_v$. Dann gelten für $U' = (U \setminus \{(M, \iota)\}) \cup \{(M', \iota')\}$ am Knoten v alle Bedingungen einer Überdeckung von G.*

Beweis: Gleichung 4.1 wird direkt durch $(M', \iota') \in U'$ erfüllt. Da U eine Überdeckung ist und $\iota(\mathrm{rt}(M)) = v$ gilt, existiert kein weiteres Element $(M'', \iota'') \in U$ mit $\iota''(u) = v$ für ein $u \in V_{M''}$ mit $\mathrm{typ}(u) \neq \perp$. Daraus folgt, dass Gleichung 4.2 auch durch U' erfüllt ist. Für Gleichung 4.3 stellen wir fest, dass mit (M', ι') die von der Konklusion geforderte Überdeckung existiert und damit die Implikation zu wahr auswertet. □

Satz 3 (Existenz einer Lösung nach *RN*-Reduktionen)**.** *Sei G ein Programmgraph und $\mathcal{M}$ eine atomar-vollständige, kompositionale Menge von Mustergraphen. Wird beim Lösen der zugehörigen PB-Instanz $P = \pi_{\mathcal{M}}(G)$ eine RN-Reduktion angewendet, so existiert auch nach der Anwendung eine Lösung der PB-Instanz.*

Beweis: Wir zeigen durch vollständige Induktion über die Anzahl der *RN*-Reduktionen, dass nach jeder *RN*-Reduktion eine Überdeckung U von G existiert, deren Lösung $\Psi(U)$ mit den heuristischen Entscheidungen übereinstimmt. Da $\mathcal{M}$ atomar-vollständig ist, wissen wir aus dem Beweis zu Satz 2, dass eine Überdeckung U aus atomaren Mustergraphen für G existiert. Die grundlegende Idee des Beweises ist es, die heuristisch gewählten Alternativen nacheinander in die Überdeckung U zu integrieren.

Die Menge $V_I = \emptyset$ stellt ein Hilfskonstrukt für den Induktionsschritt des Beweises dar. Sie beinhaltet diejenigen Knoten, deren Auswahl durch heuristische Entscheidungen festgelegt wurde und besitzt die Invariante

$$\forall v \in V_G : v \notin V_I \Rightarrow \exists (M, \iota) \in U : \iota(rt(M)) = v. \tag{4.8}$$

Da die initiale Überdeckung U aus atomaren Mustergraphen besteht, gilt die Invariante für den Induktionsanfang.

Wir betrachten nun eine beliebige RN-Reduktion, insbesondere können bereits RN-Reduktionen angewendet worden sein. Dabei sei U die bisher konstruierte Überdeckung und $l = \Phi(U)$, die von U induzierte Lösung von P. Weiter sei v der Knoten, an dem die RN-Reduktion angewendet wurde und (M, ι) ein Element der ausgewählten Alternative. Die Menge

$$V' = \{u \in G(\iota^{-1}(v)) \mid \text{typ}(u) \neq \bot \wedge (M, \iota) \notin l(\iota(u))\}$$

beinhaltet alle von (M, ι) überdeckten Knoten, deren bisherige Lösung nicht der Wahl der RN-Reduktion entspricht. Alle anderen Knoten aus $G(\iota^{-1}(v))$ betrachten wir ab sofort als festgelegt:

$$V_I = V_I \cup (G(\iota^{-1}(v)) \setminus V').$$

Da wir die Überdeckung U nicht für alle Knoten aus V' gleichzeitig anpassen wollen, verwenden wir strukturelle Induktion über die Elemente von V'. Dazu wählen wir ein $v' \in \iota(V')$, so dass die Knotenmenge von $G(\iota^{-1}(v'))$ minimal bzgl. Teilmengenrelation ist.

Unser Ziel ist es, eine Überdeckung zu konstruieren, deren induzierte Lösung für v' eine Alternative $A \in \mathcal{A}_{v'}$ mit $(M, \iota) \in A$ auswählt. Dadurch erhalten wir inkrementell eine Überdeckung, die auch an v die heuristisch gewählte Alternative selektiert. Zusätzlich müssen wir sicherstellen, dass wir die Lösung der bereits festgelegten Knoten aus V_I nicht verändern.

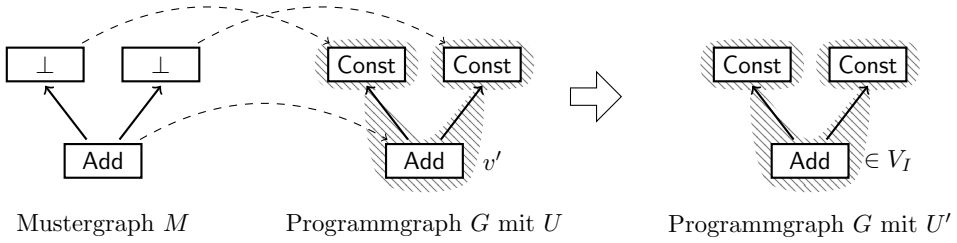


Abbildung 4.16: Austauschen äquivalenter Muster an v'.

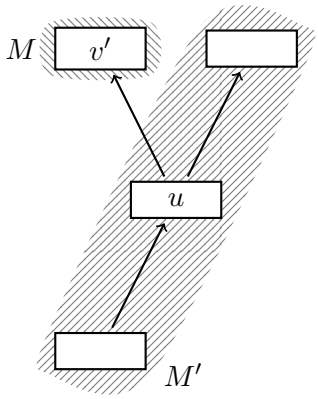
Für $(M', \iota') \in l(v')$ gilt $\iota'(rt(M')) = v'$, denn sonst ist nach Gleichung 4.8 $v' \in V_I$ und wir erhalten mit Lemma 8 und Lemma 7 einen Widerspruch zur Wahl von v'.

Wir setzen nun

$$U' = U \setminus \{(M', \iota')\} \cup \{(M, \iota)\}$$

und zeigen zunächst, dass U' die Bedingungen einer Überdeckung für alle Vorgänger von v' erfüllt. Sei dazu $u \in V_G$ ein Vorgänger von v', $A_u = l(u)$ die an u gewählte Alternative und $(M', \iota') \in A_u$.

1. **Fall:** $\iota'(\mathrm{rt}(M')) \neq u$**.** Aus Gleichung 4.8 folgt $u \in V_I$ und es genügt somit zu zeigen, dass $(M, \iota) \in A_u$ gilt.



Wir nehmen das Gegenteil an, sei also $(M, \iota) \notin A_u$ und somit nach Lemma 5 $\iota^{-1}(u) \nsim \iota'^{-1}(u)$. Da $u \in V_I$ folgt mit Lemma 8, dass A_u die einzige Alternative von u mit endlichen Kosten ist. Der Matrix-Eintrag zwischen den Alternativen A_u und $A_{v'} \ni (M, \iota)$ besitzt nach Gleichung 4.4, unabhängig von $\mathrm{typ}(\iota_v^{-1}(u))$, unendliche Kosten. Wegen Lemma 8 war A_u vor der RN-Reduktion die einzige Alternative mit endlichen Kosten, weshalb nach Gleichung 4.8 auch $A_{v'}$ unendliche Kosten besitzt. Mit Lemma 7 erhalten wir einen Widerspruch zur Auswahl der RN-Reduktion.

2. **Fall:** $\iota'(\mathrm{rt}(M')) = u$**.** Diesen Fall behandelt Lemma 9.

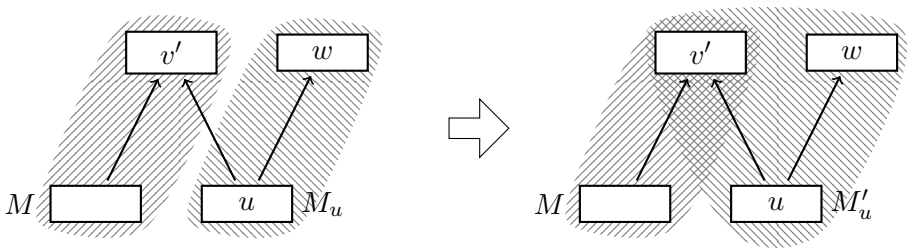
Alle Vorgänger erfüllen also die Bedingungen einer Überdeckung. Wenden wir uns nun dem Knoten v' selbst zu. Für den in Abbildung 4.16 gezeigten Fall $\iota(\mathrm{rt}(M)) = v'$ gilt wegen Lemma 9, dass v' ebenfalls die Bedingungen einer Überdeckung erfüllt. Weiter gilt $M \cong M'$ und $\iota(\mathrm{rt}(M)) = \iota'(\mathrm{rt}(M'))$ und somit ist U' eine Überdeckung von G. Aufgrund der Isomorphie $M \cong M'$ gilt Gleichung 4.8 auch für $V_I \cup \{v'\}$.

Es ist also noch der Fall $\iota(\mathrm{rt}(M)) \neq v'$ offen. In Abbildung 4.17 ist skizziert, wie die Überdeckung der Vorgänger von v' angepasst werden muss. Sei nun $u \in V_G$ ein Vorgänger von v' und $(M_u, \iota_u) \in l(u)$ ein Element der an u gewählten Alternative. Wir wollen die Existenz eines Elements (M'_u, ι'_u) einer Alternative $A'_u \in \mathcal{A}_u$ mit

$$G(\iota_u^{-1}(w)) \cong G(\iota_u'^{-1}(w)) \tag{4.9}$$

für alle $w \in succ(u) \cap V_I$ und

$$G(\iota^{-1}(v')) \cong G(\iota_u'^{-1}(v')). \tag{4.10}$$



zeigen. Dazu nehmen wir an, dass kein solches Element existiert, d.h. es ist Gleichung 4.9 oder Gleichung 4.10 verletzt. Wir betrachten zunächst den Fall, dass Gleichung 4.9 für einen Knoten $w \in V_I \cap succ(u)$ verletzt ist. Der bereits festgelegte Knoten w besitzt nach Lemma 8 nur eine Alternative $A_w = l(w)$ mit endlichen Kosten. Da l eine Lösung ist, sind die Kosten des Matrix-Eintrags zwischen A_u und A_w endlich. Wir betrachten nun den Matrix-Eintrag zwischen A'_u und $A_w \ni (M_w, \iota_w)$. Falls $\mathrm{typ}(\iota_u^{-1}(w)) = \perp$, so gilt nach Gleichung 4.4 $\iota_w^{-1} = \mathrm{rt}(M_w)$. Daraus folgt mit $G(\iota_u^{-1}(w)) \cong G(\iota_u'^{-1}(w))$ und Definition 15, dass $\mathrm{typ}(\iota_u'^{-1}(w)) \neq \perp$ und $\iota_u'^{-1}(w) \nsim \iota_w^{-1}(w)$ gelten, weshalb nach Gleichung 4.4 der zugehörige Matrix-Eintrag unendliche Kosten besitzt. Falls $\mathrm{typ}(\iota_u^{-1} \neq \perp$, so gilt nach Gleichung 4.4 $\iota_u^{-1}(w) \sim \iota_w^{-1}(w)$. Weiter folgt mit $G(\iota_u^{-1}(w)) \cong G(\iota_u'^{-1}(w))$ und Definition 15, dass $\iota_u^{-1}(w) \nsim \iota_u'^{-1}(w)$ gilt und somit der Matrix-Eintrag zwischen A'_u und A_w, im Falle von $\mathrm{typ}(\iota_u'^{-1}) \neq \perp$, unendliche Kosten besitzt. Für $\mathrm{typ}(\iota_u'^{-1}) = \perp$ folgt aus $\iota_u^{-1}(w) \sim \iota_w^{-1}(w)$ und $u \in \mathrm{cod}(M)_u$, dass $\iota_w^{-1} \neq rt(M_w)$ gilt und somit besitzt der Matrix-Eintrag zwischen A'_u und A_w auch in diesem Fall unendliche Kosten. Mit Lemma 6 folgt, dass auch A'_u unendliche Kosten besitzt.

Wenden wir uns nun dem Fall zu, dass Gleichung 4.10 verletzt ist, d.h. $\iota^{-1}(v') \nsim \iota_u'^{-1}(v')$ wegen $v' \in V'$. Weiter gilt $\iota(\mathrm{rt}(M)) \neq v'$ und somit besitzt der Matrix-Eintrag zwischen A'_u und $A_{v'} \ni (M, \iota)$ nach Gleichung 4.4 unendliche Kosten.

Insgesamt wir erhalten mit Lemma 6 und Lemma 7 ein Widerspruch zur Wahl der RN-Reduktion. Somit existiert eine Alternative $A'_u \in \mathcal{A}_u$, dessen Element (M'_u, ι'_u) Gleichung 4.9 und Gleichung 4.10 erfüllt. Diese Argumentation umfasst den Fall $u \in V_I$, da (M_u, ι_u) Gleichung 4.10 verletzt. Mit Gleichung 4.8 folgt also $\iota_u(\mathrm{rt}(M_u)) = u$.

Falls $\iota'_u(\mathrm{rt}(M'_u)) \neq u$, so existiert wegen der Kompositionalität von $\mathcal{M}$ nach Gleichung 4.5 ein Element $(M''_u, \iota''_u) \in A''_u$ einer Alternative $A''_u \in \mathcal{A}_u$ mit $\iota''_u(\mathrm{rt}(M''_u)) = u$, das ebenfalls Gleichung 4.9 und Gleichung 4.10 erfüllt. Wir können also ohne Einschränkung $\iota'_u(\mathrm{rt}(M'_u)) = u$ annehmen.

Sei nun $w \in \mathit{succ}(u) \setminus (V_I \cup \{v'\})$ ein bisher nicht betrachteter Nachfolger von u. Für $(M_w, \iota_w) \in l(w)$ gilt $\iota_w(\mathrm{rt}(M_w)) = w$. Es ist allerdings möglich, dass eine Alternative $A_w \in \mathcal{A}_w$ mit $(M'_u, \iota'_u) \in A_w$ existiert. Da $\mathcal{M}$ kompositional ist, folgt aus Gleichung 4.6 und Gleichung 4.7, dass ein Element $(M''_u, \iota''_u) \in A''_u$ einer Alternative $A''_u \in \mathcal{A}_u$ mit $\mathrm{typ}(\iota''^{-1}_u(w)) = \perp$ existiert. Dabei bleiben Gleichung 4.9 und Gleichung 4.10 erfüllt. Wir können demnach ohne Einschränkung $\mathrm{typ}(\iota'^{-1}_u(w)) = \perp$ annehmen.

Wir konstruieren eine Überdeckung U'' durch

$$U'' = U \setminus \left(\{(M', \iota')\} \cup \bigcup_{u \in \mathit{pred}(v')} \{(M_u, \iota_u)\} \right) \cup \left(\bigcup_{u \in \mathit{pred}(v')} \{(M'_u, \iota'_u)\} \right).$$

Da für alle $(M_{v'}, \iota_{v'}) \in \Psi(U')(v')$, $(M'_{v'}, \iota'_{v'}) \in \Psi(U'')(v')$ und alle Nachfolger $u \in \mathit{succ}(v')$ gilt $G(\iota^{-1}_{v'}(u)) \cong G(\iota'^{-1}_{v'}(u))$ sind die Bedingungen einer Überdeckung für die Nachfolger von v', wie bereits gezeigt, erfüllt. Allerdings bleibt noch zu zeigen, dass die Bedingungen auch für v', dessen Vorgänger und deren Nachfolger gelten.

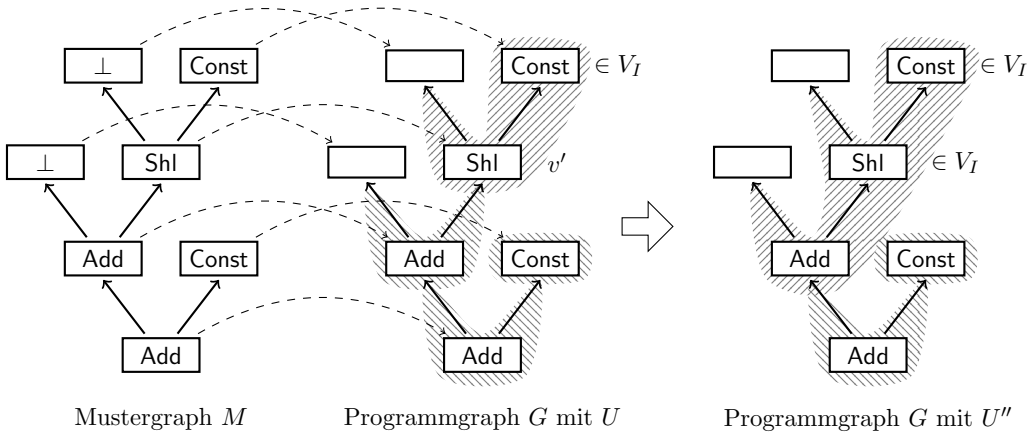


Abbildung 4.17: Inkrementelles Anpassen der Überdeckung an v'.

Da in Gleichung 4.10 alle konstruierten Mustergraphen (M'_u, ι'_u) bzgl. v' äquivalent sind, folgt Gleichung 4.2. Weiter gilt $\mathrm{typ}(\iota'^{-1}_u(v)) \neq \perp$ für alle (M'_u, ι'_u), weshalb auch Gleichung 4.3 erfüllt ist.

Die Überdeckungseigenschaft für die Vorgänger u von v' folgt aus Lemma 9. Für die Nachfolger $w \in \mathit{succ}(u) \setminus \{v'\}$ von u folgt aus der Wahl von (M'_u, ι'_u), dass

$G(\iota_u'^{-1}(w)) \cong G(\iota_u^{-1}(w))$ gilt und somit die Überdeckungseigenschaft direkt von U auf U'' übertragen wird. Weiter wurde an allen Vorgänger u von v' eine Alternative $A_u \ni (M_u', \iota_u')$ mit $\iota_u'(\mathrm{rt}(M_u')) = u$ ausgewählt und somit gilt Gleichung 4.8 für $V_I \cup \{v'\}$.

Wir haben somit eine Überdeckung konstruiert, deren Lösung l am Knoten v' die von der RN-Reduktion vorgegebene Alternative auswählt. Induktiv erhalten wir somit eine Überdeckung, die auch die am Knoten v gewählte Alternative enthält. Da diese Überdeckung bereits festgelegte Alternativen respektiert, erhalten wir induktiv eine Überdeckung, die alle heuristisch gewählten Alternativen beinhaltet. □

Zusammenfassend lässt sich also sagen, dass für eine Mustermenge, die atomar-vollständig und kompositional ist, die Bedingungen gegeben sind, dass eine Lösung garantiert werden kann.

4.6 Korrektheit der Spezifikation

Ein Vorteil der Generierung einer PBQP-basierten Befehlsauswahl im Gegensatz zu manueller Programmierung ist die Möglichkeit von zusätzlichen Tests. Eine Spezifikation in der beschriebenen Sprache kann auf Korrektheit überprüft werden. Unter Korrektheit der Spezifikation verstehen wir, dass die daraus generierte Befehlsauswahl Operationen der Zwischensprache in einem Programmgraphen vollständig transformieren kann, insbesondere muss für alle Programmgraphen eine Überdeckung gefunden werden.

Um eine Lösung des PBQP garantieren zu können, muss die Menge der Ersetzungsregeln zwei Bedingungen erfüllen, die statisch überprüft werden können.

atomar-vollständig (siehe auch Definition 18) Dazu muss getestet werden, ob für jedes atomare Muster eine Ersetzungsregel existiert. In der Phase der Befehlsauswahl können einige Knotentypen ausgeschlossen werden. Übrig bleiben Jmp, Cond, Const, SymConst, Load, Store, Conv, Add, Sub, Mul, Quot, Div, Psi, Mod, Abs, And, Or, Eor, Cmp, Shl und Shr. Zu jedem Knotentyp müssen auch alle möglichen Modi in T_{EAG} vorhanden sein.

Zu jedem Knotentyp ist die Anzahl der Nachfolger bekannt. Man kann also die generierte Regelmenge nach diesen atomaren Mustern durchsuchen. Sollten Muster fehlen, kann eine genaue Fehlermeldung ausgegeben werden, welche Muster noch zu spezifizieren sind.

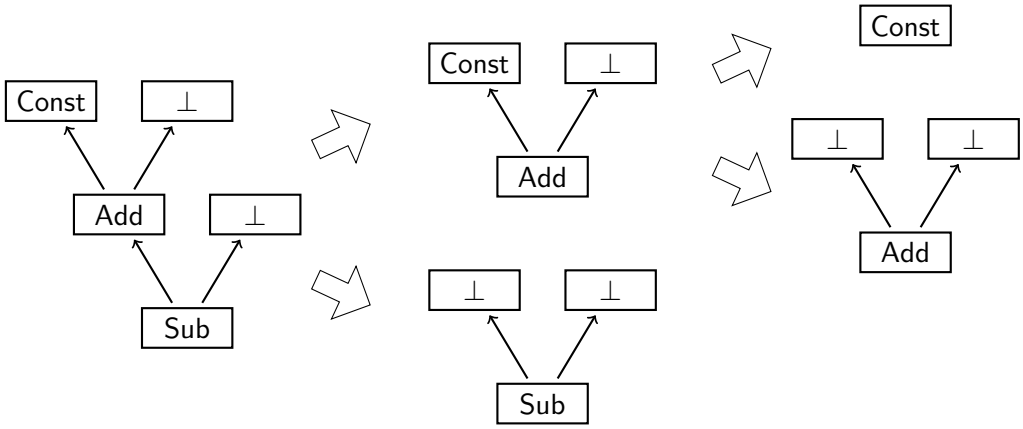


Abbildung 4.18: Implizierte Muster für eine kompositionale Mustermenge.

kompositional (siehe auch Definition 19) Nun müssen die komplexeren Muster daraufhin überprüft werden, dass auch jeweils die Untermuster ersetzt werden können.

Wenn man für ein gewurzeltes Muster einen beliebigen, direkten Nachfolger des Wurzelknotens abtrennt und durch einen $\perp$-Knoten ersetzt, muss das so entstandene Muster ebenfalls in der Mustermenge enthalten sein. Außerdem muss jedes Muster enthalten sein, das durch den aufgespannten Graphen eines Nachfolgers impliziert wird.

In Abbildung 4.18 ist ein Mustergraph mit seinen von der Kompositionalität geforderten Mustergraphen dargestellt. Die Kompositionalität fordert sowohl die Exisitenz des atomare Sub-Musters, als auch des Add-Musters mit einer Konstanten. Wegen letzterem Muster müssen zusätzlich noch die atomaren Add- bzw. Const-Muster vorhanden sein. Untypisierte Knoten sind als Spezialfall der Forderung, dass ein zugehöriges atomares Muster existieren muss, ausgenommen.

Falls solche geforderten Mustergraphen nicht gefunden werden, ist es möglich eine genaue Fehlermeldung auszugeben, welche Muster noch zu spezifizieren sind. Dies ist aussagekräftiger als die Meldungen bisheriger Codegenerator-Generatoren, wie beispielsweise BEG[ESL89].

Für atomar-vollständige Mustermengen lassen sich fehlende Ersetzungsregeln sogar automatisch erzeugen, indem für den entsprechenden Mustergraph eine Überdeckung gefunden wird und die verwendeten Regeln kombiniert werden. Das folgende Lemma zeigt, dass nur endlich viele zusätzliche Mustergraphen notwendig sind, um eine Mustermenge zu einer kompositionalen Mustermenge zu erweitern.

Lemma 10 (Kompositionale Hülle)**.** *Für jede endliche atomar-vollständige Mustermenge $\mathcal{M}$ existiert eine endliche kompositionale Mustermenge $\mathcal{M}'$ mit $\mathcal{M} \subseteq \mathcal{M}'$.*

Beweis: Da die Vereinigung zweier kompositionaler Mustermenge wieder kompositional ist, genügt es die Aussage für einelementige Mustermenge $\mathcal{M} = \{M\}$ zu zeigen. Dazu verwenden wir vollständige Induktion über die Anzahl der typisierten Knoten von M. Besitzt M nur einen typisierten Knoten, so ist M atomar und $\mathcal{M}$ ist bereits kompositional. Wir betrachten nun den allgemeinen Fall, dass M mehr als einen typisierten Knoten besitzt. Sei dazu $d = \deg(\mathrm{rt}(M))$ der Grad des Wurzelknotens. Um Gleichung 4.5 für einen Nachfolger zu erfüllen, benötigen wir *einen* weiteren Mustergraphen M', der zusammen mit den atomaren Mustergraphen eine Überdeckung von M bildet. Ebenso benötigen wir für jeden Nachfolger *einen* Mustergraphen um Gleichung 4.6 und Gleichung 4.7 zu erfüllen. All diese zusätzlichen Mustergraphen besitzen, aufgrund der fehlenden Wurzel bzw. des untypisierten Nachfolgers, weniger typisierte Knoten als M und sind nach Induktionsvoraussetzung in endlichen kompositionalen Mustermenge enthalten. Die Vereinigung von $\{M\}$ mit diesen $2 \cdot d$ Mustermengen ist somit kompositional und endlich. □

Die Auswahl der besten Überdeckung des Mustergraphen kann in gewohnter Weise durch Formulierung einer PBQP-Instanz durchgeführt werden. In diesem Fall kann sogar ein optimaler PBQP-Lösungsalgorithmus benutzt werden, da die Mustergraphen im Normalfall nur wenige Knoten besitzen.

Zusammenfassend genügt es also eine atomar-vollständige Mustermenge zu *spezifizieren*, da somit automatisch dafür gesorgt werden kann, dass die *generierte* Mustermenge kompositional ist. Dies garantiert uns wiederum, dass die PBQP-Befehlsauswahl in jedem Fall eine gültige Lösung findet.

4.7 Komplexität der PBQP-Transformation

Die Komplexität des PBQP-Lösungsalgorithmus ist $O(nm^3)$[EBS+08], wobei n die Knotenzahl im Graphen und m die maximale Größe der Kostenvektoren darstellt. Für PB-Instanzen entspricht m der maximalen Anzahl von Äquivalenzklassen $s = \max_{v \in V} |\mathcal{A}_v|$ eines Knotens. Die Schranke gilt auch für unsere Erweiterung um ∞-Propagierung, die aufgrund der beschränkten Gesamtanzahl von Vektor-Einträgen ebenfalls in $O(nm^3)$ liegt. Das kritische Element für die Laufzeit ist also die Anzahl der Alternativen.

Der bisher beschriebene PBQP-Löser wird an jedem Knoten eine passende Alternative auswählen. Um das Prinzip der Rematerialisierung zu unterstützen, muss ein Knoten allerdings die Möglichkeit haben, sowohl ersetzt, als auch konsumiert zu werden. Eine Alternative ist dann also ein Paar aus zwei Alternativen im herkömmlichen Sinne. Paare aus zwei verschiedenen Alternativen, in denen der Knoten jeweils Wurzel eines Musters ist, können ignoriert werden. Die Vektorgröße m kann nun

allerdings quadratisch in s werden. In der Praxis konnten wir Vektorgrößen bis 64 beobachten.

Statt Paaren könnten auch Tripel oder noch größere Tupel verwendet werden, um noch komplexere Auswahlmöglichkeiten zu geben. Damit wäre die Vektorgröße $m \leq s^x$, im schlimmsten Fall für jeden Verwender ein eigenes Muster, also ist x kleiner als die maximale Anzahl der Verwender eines Knotens. Eine weitere Schranke für die Anzahl solcher Kombinationen ist durch 2^s gegeben.

5 Implementierung

Dieses Kapitel beschreibt unsere Implementierung einer Befehlsauswahl. Diese Phase der Codegenerierung folgt auf die Abbildung des ABI, in der bereits Epilog und Prolog von Funktionen, als auch die Funktionsaufrufe abgearbeitet wurden. Die Knoten der Firm-Zwischensprache sollen in Knoten der Zielarchitektur überführt werden.

5.1 Gesamtarchitektur

Die Befehlsauswahlphase erhält einen Firm-Graphen und gibt einen modifizierten Firm-Graphen aus, der nur Operationen der Zielsprache enthält. Unsere Implementierung gliedert die Graphmodifikation in drei Schritte auf, die im unteren Teil von Abbildung 5.1 dargestellt sind:

1. Mustersuche im Programmgraphen.
2. Filtern der Muster durch Abbildung auf PBQP.
3. Ersetzung durch Knoten der Zielarchitektur.

Aus der Spezifikation einer Abbildung der Firm-Zwischensprache auf eine Zielarchitektur generieren wir eine Menge an Ersetzungsregeln. Zu jeder Regel gehört ein Muster, dessen sämtliche Vorkommen im ersten Schritt im Programmgraphen gesucht werden. Außerdem ist zu jeder Regel eine Ersetzung gegeben, die das Muster so transformiert, dass Operationen der Zielarchitektur verwendet werden. Der Zwischenschritt des Filterns stellt sicher, dass keine Konflikte zwischen den Ersetzungen bestehen und alle Knoten ersetzt werden. Dazu wird eine PBQP-Instanz aufgebaut, deren Lösung eine entsprechende Teilmenge der gefundenen Muster ausgewählt.

5.2 Mustersuche

Um die Muster der Ersetzungsregeln im Programmgraphen zu suchen, verwenden wir das Graphersetzungssystem GrGen, welches nicht nur DAGs unterstützt, sondern allgemeine Graphen, und sich deswegen als flexibles Werkzeug für eine explo-

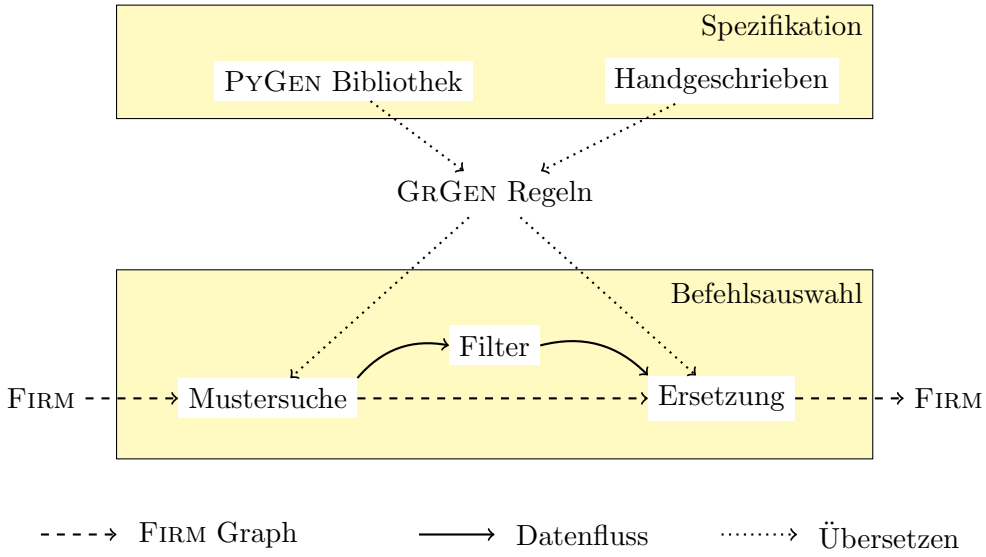


Abbildung 5.1: Architektur der Befehlsauswahlphase.

rative Implementierung eines Prototyps anbietet. Diese Mächtigkeit erzwingt aber eine unnötige syntaktische Komplexität bei der Spezifikation der Ersetzungsregeln, da GrGen keine kompakte Schreibweise mit Klammerausdrücken anbietet. Deswegen haben wir als zusätzliche Ebene einen Prototyp einer Spezifikationssprache PyGen entwickelt, der es erlaubt, gewurzelte DAG-Muster durch einfache Baumstrukturen anzugeben. Aus PyGen-Regeln werden in einem Zwischenschritt GrGen-Regeln erzeugt und das GrGen-Werkzeug generiert daraus eine Repräsentation in C-Quelltext, die in libFirm eingebunden wird.

5.2.1 PyGen-Regeln

Fragment 5.1 zeigt eine einfache PyGen-Regel zur Spezifikation einer Ersetzung eines Sub-Knotens durch einen ia32_Sub-Knoten. Der Wurzelknoten Sub hat, entsprechend seines Typs, zwei Nachfolger. Ein IR_node entspricht dem undefinierten Typ $\perp$ unserer Formalisierung. Erzeugt wird daraus, implizit durch Transformation der Wurzel des Musters, ein ia32.Sub-Knoten.

Fragment 5.1: Eine einfache PyGen-Regel zur Ersetzung eines Sub-Knotens.

```
print Rule(pattern=Sub(IR_node(), IR_node()),
                       modify= ia32.Sub())
```

5.2.2 GrGen-Regeln

In Fragment 5.2 ist die aus Fragment 5.1 generierte[1] GrGen-Regel zu sehen. Das zu suchende Muster ist nun nicht mehr als Klammerausdruck dargestellt, sondern Kanten werden explizit durch Pfeile `-->` angegeben. Die Knoten haben Namen erhalten, wie man am besten an den beiden IR_node-Knoten `ln` und `rn` sieht. Die erstellten Eingangskanten `pos0le` und `pos1re` sind in PyGen-Darstellung implizit, müssen in GrGen aber angegeben werden. Eine Kante kann keine Attribute besitzen, weswegen die Position in den Namen kodiert ist. Die Kante `pos1re` ist also an Position 1 ihres Quellknotens. Der `Mode` entspricht dem Typ $t \in T_{EAG}$ eines expliziten Abhängigkeitsgraphen und muss explizit als Knoten im Musterteil auftauchen, damit er mit dem Namen `m_iu` im Ersetzungsteil dem transformierten Knoten zugewiesen werden kann. Diese Modellierung von Typen als Knoten ist unhandlich, erlaubt aber auf Attribute dieses Knotens zuzugreifen.

Fragment 5.2: Eine einfache GrGen-Regel um einen Sub-Knoten zu ersetzen.

```
rule sub_iu {
    sub:Sub;
    sub -pos0le:df-> ln:IR_node;
    sub -pos1re:df-> rn:IR_node;
    // sub is not commutative, so position is important

    m_iu:Mode_Iu;

    modify {
      t_sub:ia32_Sub<sub>; // actual replacement
      t_sub -:has_mode-> m_iu;

      // creating dependency edges
      t_sub -pos4_nle:parameter->ln;
      t_sub -pos5_nre:parameter->rn;
    }
}
```

Die Repräsentation als GrGen-Regel hat einige Nachteile, welche die Regel aufblähen. Kanten, insbesondere ihre Position, können nicht modifiziert werden. So werden stattdessen die ausgehenden Kanten `pos0le` und `pos1re` nach GrGen-Semantik gelöscht und neue Kanten `pos4_nle` und `pos5_nre` müssen erstellt werden. Diese Gründe sprechen für die Verwendung des PyGen-Generators, es ist aber ebenso möglich, GrGen-Regeln von Hand zu schreiben.

[1] Der Lesbarkeit wegen wurde die Regel leicht geändert.

5.2.3 Mustervalidierung

GrGen unterstützt in der von uns verwendeten Variante nicht das Suchen von Pfaden unbestimmter Länge, so dass die Mustersuche als Zwischenlösung auf unsere Bedürfnisse angepasst wurde. In Abbildung 3.3 wurde das Problem der Zyklenbildung angesprochen. Das Problem tritt auf, wenn ein Pfad von einem Muster über andere Knoten in das ursprüngliche Muster zurück existiert, wie in Abbildung 5.2 dargestellt. Das Muster `Add(Load(), IRNode())` wird zwar gefunden, ist aber ungültig, da noch ein zusätzlicher Pfad von `Add` zu `Load` besteht. Sollten beide Knoten zu einer Instruktion transformiert werden, würde der zusätzliche Pfad zu einem Zyklus werden, der es unmöglich macht ein gültiges Programm auszugeben.

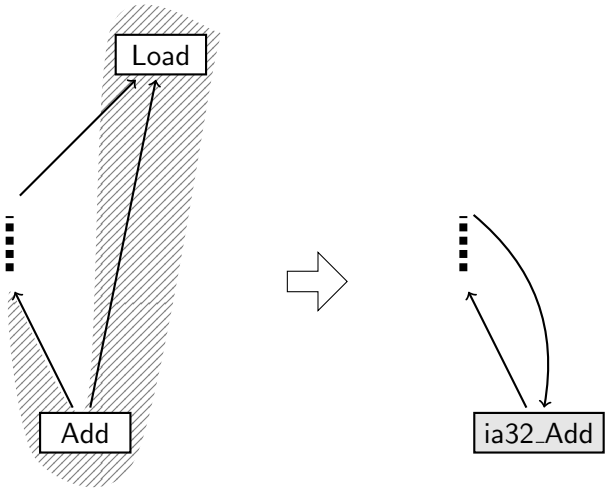


Abbildung 5.2: Gefahr der Zyklenbildung.

Ein Pfad unbestimmter Länge ließe sich unter Verwendung von Sterngraphgrammatiken[Jak08] als GrGen-Regel ausdrücken, allerdings wird diese Funktionalität nur von GrGen.NET angeboten, aber nicht in der von uns verwendeten GrGen-Version. Wir haben deshalb unsere Mustersuche erweitert, so dass alle gefundenen Muster darauf geprüft werden, dass keine derartigen Pfade existieren.

5.3 Filter

Da gefundene Muster sich gegenseitig überdecken können, sorgt ein Filter mit Hilfe einer Abbildung auf PBQP dafür, dass eine Auswahl der Ersetzungen gefunden wird, die konfliktfrei ist und alle Knoten transformiert. Dabei unterstützt unsere Imple-

mentierung, im Gegensatz zu dem in Kapitel 4 formalisierten PBQP-Aufbau, auch Rematerialisierung, auf deren Vor- und Nachteile wir im Laufe dieses Abschnitts eingehen wollen. Zusätzlich stellen wir unser Kostenmodell vor.

5.3.1 Erweiterung des PBQP-Aufbaus

Aus Gründen der Codequalität haben wir den Aufbau des PBQP erweitert. Betrachten wir dazu als Beispiel den Programmgraphen in Abbildung 5.3. Am Const-Knoten

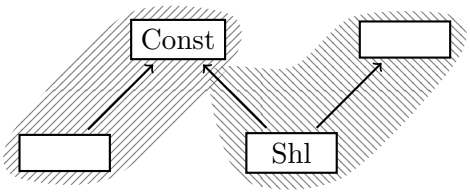


Abbildung 5.3: Eine Rematerialisierung des Const-Knotens ermöglicht die Auswahl der dargestellten Mustergraphen.

besteht ein Konflikt, denn das linke Muster möchte den Knoten konsumieren. Das rechte Muster dagegen verlangt einen materialisierten Wert. Da auch Ersetzungen existieren, die den Knoten atomar ersetzen, ist es möglich eine bessere Befehlsauswahl zu treffen, die den Const-Knoten sowohl konsumiert, als auch materialisiert. Beispielsweise könnten die beiden Verwender in verschiedenen Grundblöcken zu finden sein, wobei in einem Fall der Registerdruck Priorität hat und im anderen Fall die Befehlslänge.

Wegen Beispielen dieser Art empfinden wir Rematerialisierung als notwendig für eine gute Befehlsauswahl. Unsere Implementierung erweitert den Aufbau des PBQP, so dass jede Alternative, die keine Wurzel ist, mit einer beliebigen Wurzelalternative kombiniert werden kann. Die Anzahl der Alternativen eines Knoten wird dadurch zwar quadratisch zur Anzahl der Äquivalenzklassen, allerdings ist diese Anzahl klein, so dass wir keine spürbaren Auswirkungen auf die Laufzeit feststellen konnten.

5.3.2 Problematik der Erweiterung

Es gibt Muster, die Knoten überdecken aber nicht modifizieren, d.h. sie nicht retypisieren und keine ein- oder ausgehenden Kanten löschen oder anlegen. Wenn man Rematerialisierung zulässt, kann das zu Problemen führen, wie in Abbildung 5.4 dargestellt. Im ersten Schritt werden zwei Muster gefunden, die ersetzt werden können, wobei wir das Muster am Load_1-Knoten zuerst behandeln. Es spricht zunächst nichts dagegen, die Ersetzungsregel anzuwenden. Der Add-Knoten würde zwar keine Ver-

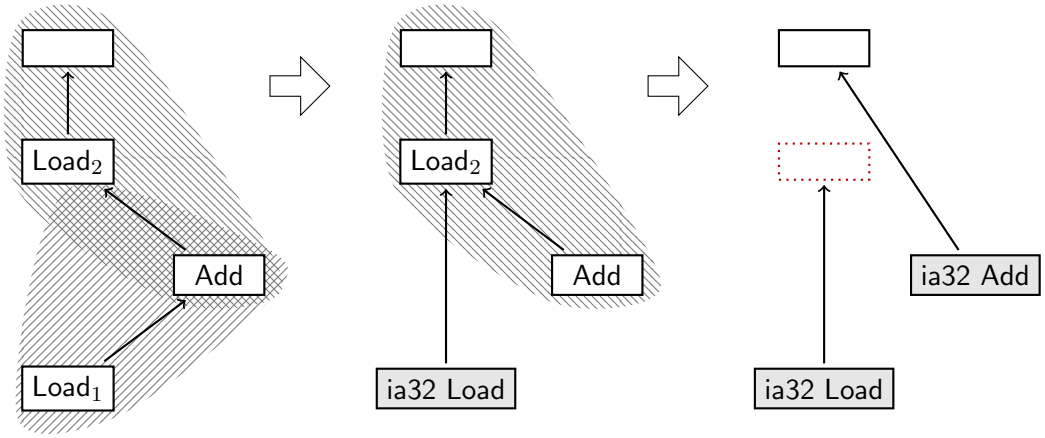


Abbildung 5.4: Das Load-Add-Load-Problem.

wender mehr besitzen, der Programmgraph ist aber korrekt. Das Problem ist, dass im zweiten Schritt eine Vorbedingung des anderen Musters verletzt wird. Der überdeckte Load_2-Knoten darf keinen zweiten Verwender besitzen, da ein volatiler Load-Knoten immer genau einmal vom Prozessor ausgeführt werden muss. Als Zusicherung[2] löscht die Add-Ersetzungsregel den Load_2-Knoten. Im dritten Schritt hat der ia32_Load-Knoten der ersten Ersetzung einen gelöschten Knoten als Vorgänger. Der Programmgraph ist jetzt ungültig, da notwendige Vorbedingungen für spätere Phasen verletzt sind und es nicht mehr möglich ist ein gültiges Programm auszugeben.

Um dieses Problem zu lösen, muss beim Aufbau des PBQP entschieden werden, dass die Add-Regel, die den Load_2-Knoten überdeckt, keine gültige Alternative darstellt, falls der Add-Knoten selbst vom Load_1-Muster überdeckt wird. Folgende Regel garantiert, dass keine zusätzlichen Verwender von Ladeoperationen erzeugt werden: Ein volatiler Knoten darf nicht gleichzeitig konsumiert und durch Lade- bzw. Speicheroperation ersetzt werden. Entsprechend dieser Regel vermerkt unsere Implementierung beim Aufbau des PBQP in der Kostenmatrix einen ∞-Eintrag für die Kombination der beiden Muster zwischen Load_1- und Add-Knoten.

Leider kamen wir im Rahmen dieser Arbeit nicht dazu, den Beweis der Lösungsgarantie auf Rematerialisierung zu erweitern. Da wir trotz intensiver Bemühungen kein Programm konstruieren konnten, für das keine PBQP-Lösung gefunden wird, liegt die Vermutung nahe, dass eine solche Erweiterung des Beweises möglich ist.

[2] engl.: assertion

5.3.3 Kostenmodell

Wie bereits in Abschnitt 4.3 beschrieben, kann die Auswahl der Ersetzungsregeln durch ein Kostenmodell gesteuert werden. Die Kosten in unserer Implementierung für eine Auswahl an einem Knoten berechnet sich grob wie folgt:

1. Je größer das Gesamtmuster, das zur Auswahl gehört, desto teurer die Auswahl. Es werden trotzdem meist große Muster ausgewählt, da mehrere kleine Muster zusammen mehr kosten als ein großes Muster, allerdings werden keine unnötig großen Muster ausgewählt. Rematerialisierung wird nur selten genutzt, weil meist ein kleineres Muster mit geringeren Kosten zur Verfügung steht, das den materialisierten Wert nutzt, statt den Knoten zu konsumieren.
2. Für Immediate- und Lea-Muster gibt es eine geringe Vergünstigung. So wird ein Lea- gegenüber einem Add-Muster bevorzugt, was sich positiv auf die Möglichkeiten der Registerzuteilung auswirkt, da die Registerbeschränkungen des Add-Befehls restriktiver sind. Dadurch können weniger Auslagerungen notwendig sein, was wiederum weniger Speicherzugriffe und damit eine bessere Codequalität bedeutet. Ein Immediate-Knoten bedeutet oft einen geringeren Registerdruck und somit ebenfalls weniger Auslagerungen.
3. Die Kosten werden multipliziert mit der Ausführungshäufigkeit, die von einem früheren Programmlauf stammt oder geschätzt sein kann. Dies hat den Effekt, dass häufig ausgeführte Befehle eine höhere Priorität besitzen. Somit werden beispielsweise innere Schleifen bei der Befehlsauswahl bevorzugt behandelt.

Die Kantenmatrizen bleiben von unserem Kostenmodell unberührt, da sich all unsere Ideen stets durch Änderungen der Musterkosten realisieren ließen. Allerdings können die Kostenmatrizen bei der in Abschnitt 8.2.3 angesprochenen Erweiterung eine wichtige Rolle einnehmen. In Abschnitt 7.1 bewerten wir unser Kostenmodell in Hinblick auf die erreichte Codequalität.

5.4 Ersetzung

Um die Ersetzung vorzunehmen, benutzen wir den Graphersetzungsmechanismus von GrGen. Jedes ausgewählte Muster gehört zu einer Ersetzungsregel, die angibt, wie dieser Teil des Programmgraphen transformiert wird. Die Auswahl, die durch die Filterphase getroffen wurde, stellt sicher, dass keine Konflikte zwischen den Ersetzungsregeln auftreten, so können Ersetzungen unabhängig voneinander vorgenommen werden.

6 Sprachentwurf

Es ist wünschenswert, die Befehlsauswahl auf einer möglichst hohen Ebene mit einer mächtigen Beschreibungssprache zu spezifizieren. So können Fehler besser erkannt und behoben werden, da die Darstellung knapper und weniger redundant ist. Bei der Implementierung unserer Befehlsauswahl haben wir ein Programm entworfen, dass aus einer knappen Darstellung von 37 Ersetzungsregeln über 7000 Ersetzungsmuster generiert.

In diesem Kapitel wollen wir zuerst eine Anforderungsanalyse an eine Spezifikationssprache erstellen, die Mechanismen des Prototyps PyGen dokumentieren und anschließend seine Grenzen aufzeigen. Um die Semantik der PyGen-Sprache zu erläutern, werden einige Implementierungsdetails angeführt. Das Ziel ist es mit diesem Wissen eine Sprache zu entwickeln, um schneller und zuverlässiger eine Befehlsauswahl zu erzeugen, als es durch manuelle Programmierung möglich ist.

6.1 Anforderungsanalyse

Die Spezifikation einer Befehlsauswahl lässt sich in verschiedene Module zerlegen. So gilt es zunächst die Knoten der Zwischensprache bzw. der Zielarchitektur zu modellieren. Diese können anschließend zu (Muster-)Graphen zusammengefügt werden. Zuletzt werden Ersetzungsregeln spezifiziert, die einen Mustergraphen in einen Zielarchitektur-spezifischen Graphen überführen.

Da sich die Knoten der Zwischensprache kaum von Zielarchitektur-spezifischen Knoten unterscheiden, können diese durch eine gemeinsame Spezifikationssprache modelliert werden. Die Knotenbeschreibung muss neben dem Knotennamen auch Attribute umfassen, die Eigenschaften des Knotens, wie zum Beispiel Kommutativität, beschreiben.

Mit den spezifizierten Knoten können (Muster-)Graphen formuliert werden, die im Rahmen einer Befehlsauswahl verwendet werden können. Die gewurzelten DAGs aus Kapitel 4 können durch geklammerte Ausdrücke beschrieben werden. So beschreibt Add(Const(),Const()) eine baumartiges Muster für die Addition von zwei Konstanten. Um die angesprochenen DAGs spezifizieren zu können, werden Knotennamen

vergeben, die anschließend wiederverwendet werden. Sollen im obigen Beispiel beide Const-Knoten identisch sein, so würde der Graph durch Add(x:Const(),x) spezifiziert werden.

Bei Mustergraphen ist es zusätzlich notwendig Bedingungen zu spezifizieren, die für einen Fund des Mustergraphen erfüllt sein müssen. Diese können sich auf Knoten-Attribute beziehen oder auch durch zusätzliche Graphen ausgedrückt werden. Ein Beispiel für letztere Bedingungen stellen die in Abschnitt 3.5 angesprochenen volatilen Lade-Operationen dar.

Um zwei Graphen zu einer Ersetzungsregel zu kombinieren, müssen die Operanden des neuen Befehls mit Knoten des Mustergraphen identifiziert werden. Dies lässt sich über gleiche Knotennamen bewerkstelligen, die dafür sorgen, dass der Knoten auch nach der Ersetzung erhalten bleibt. Eine weitere Notwendigkeit von Ersetzungsregeln stellen Attributzuweisungen dar, durch die der erzeugte Knoten Attributwerte der Musterknoten übernehmen kann.

6.2 Der Prototyp PyGen

Um die im vorherigen Abschnitt beschriebenen Anforderungen zu präzisieren, haben wir den Prototyp PyGen entwickelt, der Ersetzungsregeln für die IA-32-Architektur generiert. Neben den bereits angesprochenen Anforderungen verfolgt PyGen das Ziel, eine kompakte Darstellung der Regeln zu ermöglichen. Die dazu eingesetzten Techniken werden in Abschnitt 6.2.2 vorgestellt. Zuvor wollen wir jedoch eine allgemeine Einführung in PyGen geben.

6.2.1 Einführung

PyGen umfasst eine Beschreibungssprache, die in Python[Pyt] eingebettet wurde[1] und deshalb Konstrukte dieser Sprache wiederverwendet. Aus den spezifizierten Ersetzungsregeln generiert PyGen eine GrGen-Regeldatei. Diese wird anschließend mit GrGen in C-Anweisungen übersetzt, die in libFirm eingebunden werden.

Knoten

Die Einbettung der Beschreibungssprache in Python zeigt sich beispielsweise dadurch, dass Knoten als Klassen und Knotenattribute als Klassenattribute dargestellt werden. So zeigt das folgende Beispiel die Spezifikation eines Add-Knotens:

[1] engl.: embedded domain specific language (EDSL)

```
class Add:
  commutative = True
```

Da alle Add-Knoten kommutativ sind, wird das commutative-Attribut mit True initialisiert.

Graphen

Der Idee einen Graph als Term von Knotenobjekten zu formulieren, konnte in PyGen direkt umgesetzt werden. Das folgende Beispiel zeigt einen Mustergraphen des `ia32_Lea`-Befehls:

```
Add(Shl(sc=ScaleConst(), index=IR_node()), base=IR_node())
```

Die `IR_node`-Knoten entsprechen dabei den untypisierten Knoten unserer Formalisierung aus Abschnitt 4.1. Diese Knoten besitzen die Namen `base` und `index`, die vom Ersetzungsgraphen verwendet werden können. In PyGen ist es immer erlaubt, dass zwei Knoten eines Musters auf denselben Knoten des Programmgraphen abgebildet werden. Dies ist natürlich nur möglich, wenn die Knoten den gleichen Typ besitzen. Im obigen Beispiel können `base` und `index` also auf denselben Knoten des Programmgraphen abgebildet werden, während keiner der beiden Knoten mit dem `ScaleConst`-Knoten zusammenfallen kann. Müssen hingegen zwei Knoten identisch sein, so werden bereits vergebene Knotennamen wiederverwendet:

```
Add(Shl(sc=ScaleConst(), index=IR_node()), index)
```

Für Mustergraphen ist es ebenfalls notwendig, Bedingungen für die Suche anzugeben. Diese werden in den von PyGen spezifizierten Knoten verankert. So prüft der obige `ScaleConst`-Knoten, ob sein Wert 1, 2 oder 3 ist, da nur 2 Bit des `ia32_Lea`-Befehls für die Skalierung des `index` zur Verfügung stehen.

Ersetzungsregeln

Um den ersten beiden obigen Mustergraphen zu einer Ersetzungsregel auszubauen, benötigen wir einen zugehörigen Ersetzungsgraphen. Dieser besitzt die Form

```
ia32_Lea(base, index)
```

und verwendet die base- und index-Knoten des Mustergraphen wieder. Dadurch bleiben diese Knoten bei der Ersetzung erhalten.

Wählt man feste Namen für diese wiederverwendeten Knoten, so brauchen diese nicht bei jeder Regeln explizit angegeben werden. In PyGen bekommt das obige ia32_Lea deshalb keine Knoten übergeben, sondern geht davon aus, dass im Muster zwei Knoten mit dem Namen `base` und `index` vorkommen. Wir haben allerdings gesehen, dass für den ia32_Lea-Knoten auch Mustergraphen existieren, die keinen base-Knoten besitzen. Dies stellt kein Problem dar, da solche optionalen Knoten mit Standard-Werten vorbelegt sind, die dafür sorgen, dass eine konsistente Ersetzungsregel generiert wird. Im folgenden ist eine PyGen-Regel für den ia32_Lea-Befehl gezeigt:

```
Rule(
pattern =
 Add(Shl(sc=ScaleConst(), index=IR_node()), base=IR_node()),
modify = ia32_Lea()
)
```

Wie man sieht, verwendet die Regel-Spezifikation ebenfalls Klammerausdrücke, deren Elemente der Musterteil (`pattern`) und die Ersetzung (`modify`) sind.

Damit der erzeugte ia32_Lea-Befehl auch die richtige Skalierung verwendet, muss sein `scale`-Attribut auf den Wert des ScaleConst-Knotens sc gesetzt werden:

```
ia32_Lea.scale = sc.intval
```

Diese Zuweisung ist Teil der ia32_Lea-Spezifikation und wird nur ausgeführt, wenn der Mustergraph den Knoten sc enthält.

Bei den bisherigen Beispielen wurde nur *ein* IA-32-spezifischer Knoten erstellt. Allerdings exisitieren auch Regeln, die mehrere dieser Knoten erzeugen. Ein Beispiel stellt die in Abbildung 6.1 gezeigte Konvertierung eines `unsigned int`-Wertes in einen `float`-Wert dar, die bei präziser[2] Fließkomma-Arithmetik drei Speicher- und zwei Ladebefehle erfordert[3]. Die Konvertierung der Zweierkomplement- in die IEEE-754-Kodierung erfordert drei Speicherzugriffe, da kein Befehl zur Verfügung steht, um einen Wert direkt im Register umzuwandeln. Zunächst muss der 4 Byte `unsigned int`-Wert gespeichert und um 4 führende Nullbyte erweitert werden, da der anschließende Ladebefehl 8 Byte lädt. Nach diesem Ladebefehl befindet sich der Wert zwar richtig kodiert in einem Fließkommaregister, allerdings ist seine Präzision zu hoch für einen 32-Bit-Fließkommawert. Deshalb wird der Wert noch einmal in 32 Bit gespeichert, um ihn durch erneutes Laden auf die korrekte Genauigkeit zu runden.

[2] siehe auch /fp Visual C++ Compiler Options:
http://msdn.microsoft.com/en-us/library/e7s85ffb.aspx

[3] mit SSE2-Befehlssatz geht es allerdings auch durch genau einen Befehl: CVTDQ2PS

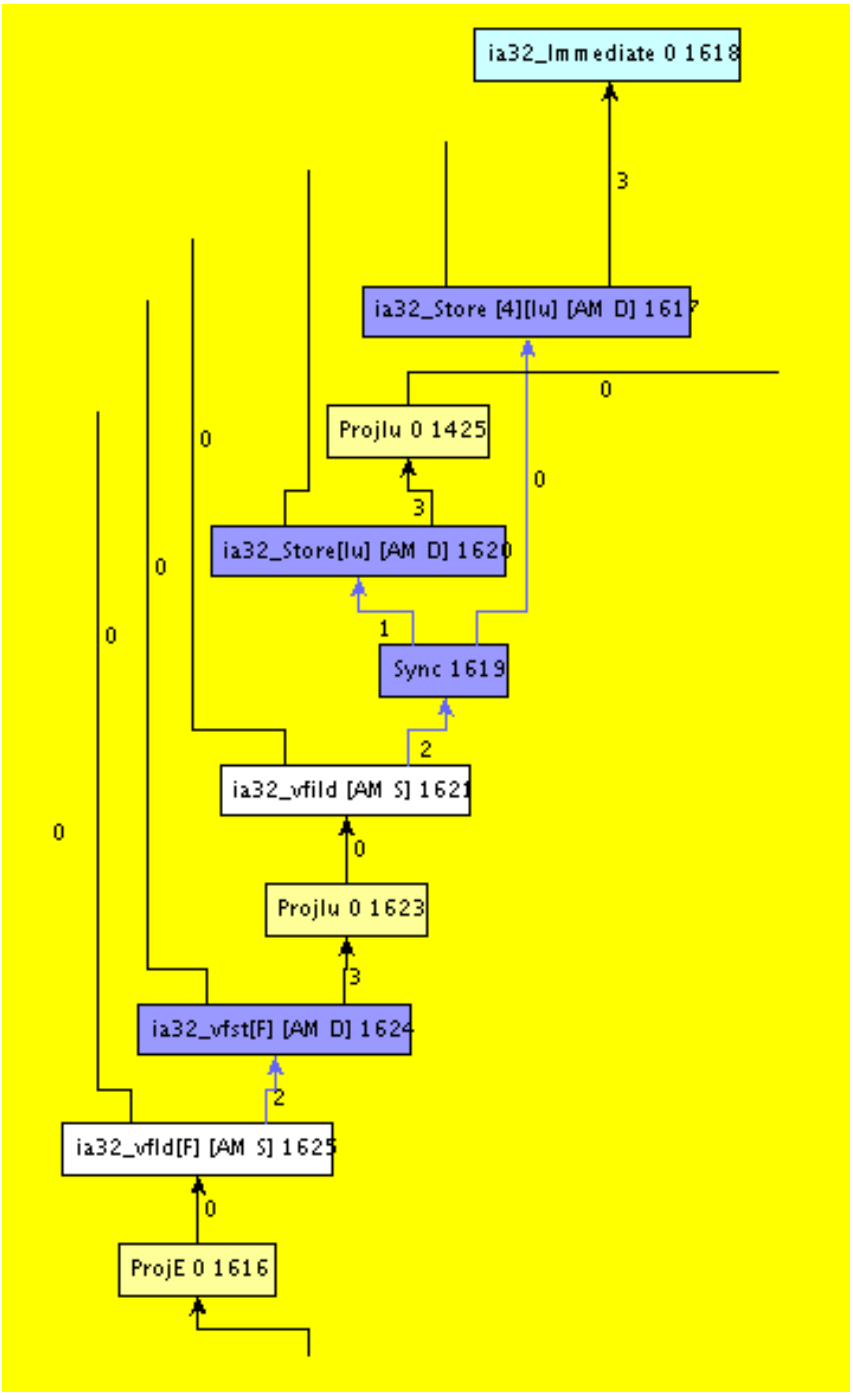


Abbildung 6.1: Darstellung einer Konversion eines vorzeichenlosen `int`-Wertes (der `ProjIu 0 1425` Knoten) in einen vorzeichenbehafteten Wert (der `ProjE 0 1616` Knoten).

Da sich Ersetzungsgraphen ebenfalls als Klammerausdrücke darstellen lassen, sind solche Graphen, wie im obigen Beispiel, nicht problematisch. Sie zeigen jedoch die Notwendigkeit einer kompakten Repräsentation, da viele, teilweise redundante Graphen dieser Größe existieren.

6.2.2 Techniken zur kompakten Darstellung

Soweit bisher vorgestellt, ist PyGen eine hinreichende Spezifikationssprache, welche die Anforderungen aus Abschnitt 6.1 erfüllt. Da jedoch eine kompakte Darstellung der Ersetzungsregeln wünschenswert ist, bietet PyGen weitere Techniken an.

Vererbung von Knoteneigenschaften

Bei der Deklaration der Knotentypen fällt auf, dass die Attribute vieler Knoten identisch sind. Wenn diese Attribute für jeden Knoten explizit annotiert werden müssen, geschehen leicht Fehler durch kleine Inkonsistenzen, da die Spezifikation durch die redundanten Anweisungen unübersichtlich wird. Wünschenswert wäre es, diese Knoten zu Gruppen zusammenzufassen und die Attribute vergeben zu können, ohne sich wiederholen zu müssen. Da Knoten in PyGen als Klassen repräsentiert werden, können durch gemeinsame Superklassen Knotengruppen entstehen und Attribute ohne Wiederholung deklariert werden.

```
class BaseNode:
  commutative = True
class And    (BaseNode): pass
class Or     (BaseNode): pass
class Not    (BaseNode): pass
class Shl    (BaseNode):
  commutative = False
```

Im Beispiel erben die Knotenklassen `And`, `Or` und `Not` alle jeweils von der Klasse `BaseNode`. Gleiches gilt auch für die Klasse `Shl`, allerdings setzt diese zusätzlich das Attribut `commutative` auf `False`. Die `pass` Anweisung kennzeichnet einen leeren Klassenrumpf.

Teilregeln und Schablonen

Manche Regelteile, wie die Adressberechnung, wiederholen sich, so dass es sich anbietet, diese Teilmuster in einen eigenen Knoten zu kapseln.

```
Store(AddressCalc(), mem=IR_node(), store_val=IR_node())
Load(AddressCalc(), mem=IR_node())
```

In den beiden Beispielen repräsentiert `AddressCalc` ein komplexes Teilmuster, das auf diese Weise nicht in beiden Regeln explizit formuliert werden muss.

Dual dazu existieren Regeln, die bis auf Teilmuster gleich sind. Für diesen Fall benutzen wir einen Mechanismus, der unterschiedliche Teilmuster in eine Regelschablone einsetzt.

```
pats = (
    Load(AddressCalc(), mem=IR_node()),
    Load(mem=IR_node(), symconst=SymConst()),
    Load(mem=IR_node(), base=IR_node()) )
for pat in pats:
  print Rule(pattern=pat, modify=ia32.Load())
```

In diesem Beispiel sind in `pats` drei Muster angegeben, mit denen jeweils durch Einsetzen als `pat` eine Regel spezifiziert wird, die das Muster durch ein `ia32_Load` ersetzt.

Ausfalten und Klonen von Regeln

Es gibt Befehle mit optionalen Fähigkeiten. Beispielsweise kann der `ia32_Store`-Befehl entweder ein Register speichern oder direkt einen festen Wert. Abgesehen von diesem kleinen Unterschied sehen die beiden Regeln gleich aus. PyGen erlaubt es, diese Regel *einmal* zu spezifizieren und anschließend in ihre Varianten „auszufalten“:

```
contexts.split('consume_immediate', True, False)
```

Varianten werden in sogenannten *Kontexten* verwaltet, die einer PyGen-Regel zugeordnet sind. Jede spezifizierte Regel erhält einen Kontext, in dem beliebige Werte unter Bezeichnungen gespeichert werden. Eine Regel kann auch mehr als einen Kontext enthalten, was dazu führt, dass GrGen-Regeln entsprechend des Kontexts in der jeweiligen Regelvariante erzeugt werden. Ein Kontext enthält beispielsweise Informationen, welcher Knoten einem bestimmten Namen zugeordnet ist. Das Erzeugen zusätzlicher Kontexte aus dem einen Ursprungskontext einer Regel bezeichnen wir als „Ausfalten“.

Die `split`-Anweisung im obigen Beispiel verdoppelt während der ersten PyGen-Phase die Kontextmenge und setzt das Attribut `consume_immediate` jeweils einmal auf `True` und einmal auf `False`. Die Kontextmenge der Regel ist nun doppelt so groß. Die Regel erzeugt bei der Ausgabe für jeden Kontext eine GrGen-Regel, so dass nun

für jede vorher existierende Variante jeweils eine Variante mit `consume_immediate` gleich `True` und gleich `False` existiert. Diesen Mechanismus bezeichnen wir als Ausfalten einer Regel.

Manchmal reicht das einfache Ausfalten alleine nicht aus, weil die Varianten auf unterschiedliche Art weiter aufgefalten werden müssen. Beim Ausfalten in die verschiedenen Varianten von Typkonversionen müssen manche Konversionen als „strikt“ markiert werden, d.h. der Kontextwert `strict_conv` muss auf `True` gesetzt werden. Eine strikte Konversion ist durch das Quellprogramm vorgegeben und nicht nur zur Typanpassung eingefügt. Andere Konversionen dagegen *können* strikt sein, d.h. die Regel muss in zwei Varianten ausgefaltet werden, da der Kontextwert `strict_conv` sowohl `True` als auch `False` sein kann. Dazu bietet unsere Implementierung Klon- und Verschmelzmethoden an.

```
mergelist = list()
for conv in conversions:
  c = contexts.clone()
  if maybe_strict(conv):
    c.split('strict_conv', True, False)
  if must_be_strict(conv):
    c.set('strict_conv', True)
  mergelist.append(c)
contexts.clear_and_merge(mergelist)
```

In diesem Beispiel werden Klone `c` der Kontextmenge erstellt und weiter verarbeitet. Es wird `strict_conv` entweder auf wahr gesetzt oder der Kontext weiter ausgefaltet und zwei Varianten für wahr und falsch erstellt. Anschließend wird die ursprüngliche Kontextmenge mit dem Aufruf `contexts.clear_and_merge(mergelist)` durch die Vereinigung aller Klone ersetzt.

Es ist auch möglich das `split`-Beispiel oben als Klonoperation darzustellen, was allerdings nicht als kompakt bezeichnet werden kann:

```
mergelist = list()
for x in (True, False):
  c = contexts.clone()
  c.set('consume_immediate', x)
  mergelist.append(c)
contexts.clear_and_merge(mergelist)
```

6.3 Grenzen des Prototyps

Durch die Entwicklung des Prototypen PyGen haben wir viele Erkenntnisse über Entwurf und Implementierung eines Befehlsauswahlgenerators gewonnen. Um während der Entwicklung des Prototyps flexibel zu bleiben, haben wir Sprache und Generator nicht scharf getrennt, was sich negativ auf längerfristige Wartbarkeit und Erweiterbarkeit auswirkt. Außerdem lässt unser Prototyp einige Sprachkonstrukte vermissen, die ihn von einem ausgereiften Befehlsauswahlgenerator unterscheiden:

Unterstützung von Mustergraphen mit mehreren Wurzeln. Einige Maschinenbefehle lassen sich nicht durch Mustergraphen mit nur einer Wurzel darstellen. In unserer theoretischen Betrachtung und Implementierung finden sich nur gewurzelte Muster und es ist möglich damit eine Befehlsauswahl zu erstellen. Bestimmte Operationen und damit Optimierungen müssen in diesem Fall allerdings ungenutzt bleiben. Ein Beispiel stellt ein Ladebefehl mit Postinkrement dar, der von einer Adresse lädt und diese dann inkrementiert, denn dessen Mustergraph besitzt zwei Wurzeln, die einmal den geladen Wert und einmal die modifizierte Adresse repräsentieren. Ein Befehlsauswahlgenerator sollte die Möglichkeit bieten, solche DAG-Muster zu spezifizieren und zu verarbeiten.

Einbinden beliebiger Funktionen. In einigen Fällen ist es vorteilhaft auf bereits vorhandene Funktionalität des Übersetzers zurückzugreifen oder komplexe Berechnungen in eigene Funktionen auszulagern, um Codeduplikation zu vermeiden und die spezifizierten Regeln übersichtlich zu halten. So können bei ARM-Architekturen Const-Knoten, die einen Wert der Form $(k \mod 2^8)$ `ror` $(2k' + 1)$ besitzen, direkt als Werte im Assemblercode erzeugt werden. Da der Übersetzer bereits eine Funktion zum Prüfen dieser Form bereitstellt, kann diese für die Mustersuche wiederverwendet werden.

Durch die Verwendung von GrGen entstehen weitere Einschränkungen, die sich auf die Möglichkeiten der Mustersuche auswirken:

Verwenden geeigneter Suchstrategie. GrGen erstellt für jeden Programmgraph dynamisch einen Suchplan, der eine effiziente Suche ermöglicht, aber auch eine hohe Initialisierungszeit der Mustersuche (ca. 1,2 Sekunden) mit sich zieht. Allerdings ist ein statischer Suchplan für uns Zwecke ausreichend und würde die besagte Initialisierungszeit drastisch reduzieren.

Die Integration von GrGen war ursprünglich für Optimierungen konzipiert und deshalb für die Suche nach Vorkommen *eines* Musters optimiert. Bei der Verwendung von GrGen für die Befehlsauswahl müssen allerdings alle Vorkommen einer Vielzahl von Mustergraphen gefunden werden, so dass es sich anbietet alle Mustergraphen auf einmal zu suchen, um gemeinsame Teilgraphen nicht wiederholt zu betrachten. Auf diese Problematik gehen wir in Abschnitt 8.2.1 ein.

Ignorieren von irrelevanten Konversionen. Die von uns zu transformierenden Graphen können Conv-Knoten beinhalten, die bei der Mustersuche ignoriert werden können. Die GrGen Integration von libFirm stellt allerdings keinen Mechanismus bereit, um diese Knoten zu übergehen, weshalb die entsprechenden Muster nicht gefunden werden.

In Abbildung 6.2 besitzt der Add-Knoten zwei Verwender, wobei der Load-Knoten einen anderen Typen benötigt und deshalb ein Conv-Knoten zwischengeschaltet ist. Der Conv-Knoten lässt sich nicht entfernen, da dazu der Typ des Add-Knotens geändert werden müsste und dieser dann wieder inkompatibel zu seinem zweiten Verwender wäre. Eine Mustersuche, die irrelevante Konversionen ignoriert, könnte das LA-Muster aus Abbildung 4.5 hier finden.

6.4 Sprachskizze

Wir haben nun die Fähigkeiten und Beschränkungen von PyGen betrachtet und kennen die Anforderungen an eine kompakte, flexible und mächtige Spezifikationssprache für eine Befehlsauswahl. In diesem Abschnitt wird eine Sprache beschrieben, die alle gewünschten Anforderungen erfüllt. Der Schwerpunkt liegt dabei auf den Mechanismen und Fähigkeiten — nicht auf Details wie der Modularisierung oder der Syntax. Das Design einer Sprache wird oft unterschätzt und gerade im Bezug auf Eigenheiten von speziellen Zielarchitekturen ist es schwer, die Hindernisse bereits in der Planungsphase zu erkennen. Ziel ist es, unsere Erkenntnisse von der Implementierung des Prototyps für die IA-32-Architektur in eine Notation zu bringen, die PyGen an Effizienz und Mächtigkeit übertrifft. Diese domänenspezifische Sprache[4]

[4] engl.: domain specific language (DSL)

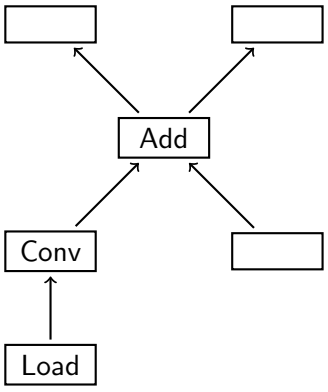


Abbildung 6.2: Ein ignorierbarer Conv-Knoten.

umfasst sowohl die Spezifikation von Knoten der Zwischensprache und der Zielarchitektur als auch die Beschreibung der Ersetzungsregeln zur Transformation.

6.4.1 Mechanismen

Wie auch bei PyGen beschreiben wir eine Reihe von Mechanismen. Da die Sprache zusätzliche Konstrukte aus Abschnitt 6.3 unterstützt, ist die Ausdrucksmächtigkeit größer. Die Anzahl der verwendeten Mechanismen ist jedoch geringer.

Knoten

Zur Spezifikation von Knotentypen müssen Bedingungen und Attributen angegeben werden. Dies ist für Knoten der Zwischen- wie auch der Zielsprache notwendig. Beispielhaft sei die Deklaration eines Add-Knotens gegeben.

$$\mathsf{Add}[2] \begin{cases} \leq \mathsf{Node} \\ \text{commutative:bool} \leftarrow \top \\ \text{mode:ir\_mode\_t} \\ \texttt{is\_valid(self)} \end{cases}$$

Ein Add-Knoten hat zwei Nachfolger, was durch die annotierte [2] deklariert ist. Der Generator kann so überprüfen, ob die Signatur zumindest der Anzahl der Argumente nach, wie in Firm definiert, eingehalten wird und den Entwickler auf Fehler in Mustern hinweisen. Im Rumpf der Deklaration sind die vier möglichen Arten von Angaben zu Knoten.

Vererbung Durch das $\leq$ Symbol wird eine Vererbung[5] gekennzeichnet. Semantisch ist diese Vererbung äquivalent dazu, den Rumpf der Node-Deklaration an die Stelle der Vererbung zu kopieren. Ausnahme sind statische Attribute - diese werden vom Kindknoten überschrieben.

Statische Attribute Attribute sind durch Namen und Typ deklariert und erhalten einen festen Wert zugewiesen. Wir gehen davon aus, dass das Rahmenwerk entsprechende Typen definiert und diese korrekt sind. Eine Überprüfung der Typen wird im Rahmen dieser Beschreibung nicht betrachtet. Durch den Punktoperator kann auf den Wert eines Attributs zugegriffen werden. Nach dem obigen Beispiel ist Add.commutative $= \top$.

Dynamische Attribute Der Wert eines solchen Attributes wird erst durch den bei der Mustersuche gepassten Knoten festgelegt.

[5] Diese Bedeutung des $\leq$ Symbols ist der Notation von Cardelli zur objektorientierten Typtheorie entnommen, wo es ebenfalls als Vererbung interpretiert wird.

Bedingung Ein Ausdruck in Syntax der zu generierenden Sprache (für unsere Beispiele C) der zu wahr oder falsch auswertet. Insbesondere können beliebige Funktionen aufgerufen werden, um vorhandene Funktionalität des Übersetzers zu nutzen. Der Ausdruck muss zu wahr auswerten, damit der Knoten gepasst werden kann. Die Zusatzbedingungen von PyGen können mit diesem Mechanismus realisiert werden. Als Variablen sind die Knotennamen und deren Attribute verfügbar. Außerdem muss der Knoten selbst durch eine Variable adressierbar sein, wie im Beispiel durch `self` dargestellt. Dieser Mechanismus kann beispielsweise dazu genutzt werden, um Konstanten, die nur bestimmte Werte annehmen können, zu spezifizieren.

Muster

Gewurzelte DAGs lassen sich als Klammerausdruck mit benannten Knoten ausdrücken. Ein Knoten wird einmal mit seinem Typen deklariert und kann im selben Ausdruck durch seinen Namen referenziert werden. Im folgenden Beispiel hat die Konstante x zwei verschiedene Add-Vorgänger, wie auch in Abbildung 6.3 zu sehen. Der Klammerausdruck ist dadurch kein Baum mehr, sondern ein DAG.

Add(x:Const, Add(x, Node))

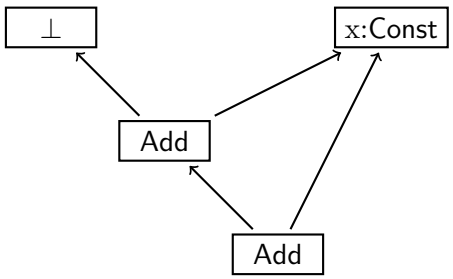


Abbildung 6.3: DAG-förmiges Add-Muster.

Mit einer Liste von solchen Ausdrücken lassen sich allgemeine DAGs beschreiben, also Muster mit mehreren Wurzeln. Um eine solche Liste zu kennzeichnen, benutzen wir einen speziellen Knoten τ. Zum Beispiel würden zwei Load-Knoten, die von derselben Adresse laden, wie in Abbildung 6.4 abgebildet, als Ausdruck so aussehen:

τ(Load(x:Node), Load(x))

Für Ausdrücke aus mehreren τ-Knoten ist durch zwei Ersetzungsregeln angegeben, wie diese zu einem τ reduziert werden können. Die Zahl und Reihenfolge der Teilausdrücke eines τ-Knotens ist dabei nicht relevant.

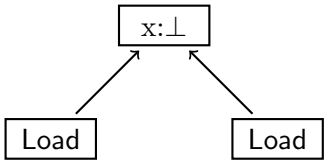


Abbildung 6.4: Zwei Ladeoperationen von derselben Adresse.

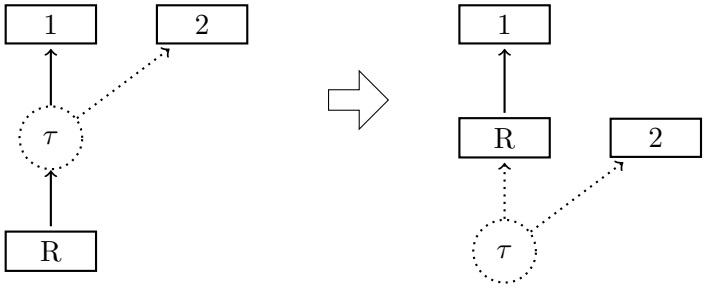


Abbildung 6.5: Ein τ-Knoten kann zur Wurzel gemacht werden, ohne den spezifizierten DAG zu verändern.

$$\begin{aligned} \mathrm{x}(\tau(\mathrm{a,b,}\ldots)\mathrm{,c}) &\Rightarrow \tau(\mathrm{x(a,c),b,}\ldots) \\ \tau(\tau(\mathrm{a,b,}\ldots)\mathrm{,c,}\ldots) &\Rightarrow \tau(\mathrm{a,b,c,}\ldots) \end{aligned}$$

Die erste Regel ermöglicht ein „Schieben“ der τ-Knoten zur Wurzel und die zweite Regel ein Zusammenfassen mehrerer τ-Knoten. Dass die zweite Regel die Bedeutung des Ausdrucks erhält, ist trivial. Es ist aber nicht intuitiv verständlich, wie eine Kante zu einem τ-Knoten im Mustergraphen zu deuten ist. Dieser Fall kann aber durch Verwendung von Hilfsknoten, wie wir sie weiter unten beschreiben, vorkommen. Am klarsten scheint das Verständnis, dass die Kante damit indirekt auf den ersten Nachfolger des τ-Knoten gerichtet ist. Nach dieser Auffassung erhält die erste Regel die Semantik des Ausdrucks. In Abbildung 6.5 ist die Regel visualisiert und es ist zu sehen, dass der Graph, wenn man die gepunkteten Teile des τ Knotens ignoriert, nach der Transformation der gleiche ist. Auf diese Art kann jeder Ausdruck zu einer Normalform reduziert werden, die höchstens einen τ-Wurzelknoten enthält.

Ersetzungsregeln

Nachdem nun Muster spezifiziert werden können, bleiben noch die Ersetzungsregeln. Eine Ersetzung besteht aus einem zu suchenden Muster M, einem Graph G, in den transformiert werden soll, und zusätzlichen Zuweisungen oder Bedingungen A_i. Die Transformation ist durch einen Pfeil dargestellt, also $M \Rightarrow G\{A_1, \ldots, A_n\}$. Die atomare Ersetzung einer Addition kann dargestellt werden als

x:Add(left:Node, right:Node) ⇒ x:ia32_Add(left, right)
{ x.mode ← `unify`(x.mode) }

Auf der rechten Seite der Ersetzung können, wie auch links, ganze Knotenmuster stehen. Im Beispiel ist links das atomare Add-Muster dargestellt mit den beiden Nachfolgern left und right. So ist es möglich mehrere Befehle auszugeben, wie es beispielsweise bei der Typkonversion notwendig sein kann. Die `unify`-Funktion, die in der Zuweisung benutzt wird, ist ein Aufruf einer Funktion des Übersetzers, in dessen Codegenerierungsphase diese Ersetzungsregel verwendet wird.

Die Bedingungen einer Ersetzung, insbesondere Bedingungen von Knoten auf der rechten Seite, müssen zum Zeitpunkt der Mustersuche geprüft werden. Wenn ein Muster gefunden wurde, muss die Ersetzung möglich sein, insbesondere muss jede Knotenbedingung der rechten Seite einer Ersetzung zu wahr auswerten. Die Beispielersetzung beinhaltet eine Anweisung: Die Zuweisung des `mode`-Attributes, welche dafür sorgt, dass der verschmolzene Knoten denselben Attributwert besitzt wie der x-Knoten der linken Seite. Es können weitere Anweisungen spezifiziert sein. Insbesondere können Ausdrücke in der Syntax der zu generierenden Sprache angegeben sein. Das erlaubt es, beliebige zusätzliche Funktionen des Übersetzers aufzurufen.

Durch den gleichen Wurzelnamen legt eine Regel fest, dass Vorgänger übernommen werden. Im Beispiel oben bedeutet dies, dass jeder Vorgänger des x:Add-Knotens nach der Transformation ein Vorgänger des x:ia32_Add-Knotens ist. Diese Angabe ist vor allem dann wichtig, wenn mehrere Wurzeln transformiert werden sollen. Im folgenden Fall ist ein Muster mit zwei Wurzeln gegeben, dessen Ersetzung darin besteht, die beiden Wurzeln zu verschmelzen. Der Wurzelknoten der rechten Seite bekommt dazu zwei Namen gleichzeitig zugewiesen.

τ(x:Load(a:Node), y:Load(a)) ⇒ x,y:Load(a) { }

Bedingungsmuster

Es gibt Fälle, bei denen die Anwendungsbedingung eines Musters ebenfalls als Muster formuliert werden kann. So kann beispielsweise ein spezieller, zusätzlicher Verwender explizit verlangt werden. Als Beispiel sei folgende Regel gegeben, die eine Not-Operation atomar ersetzt unter der Bedingung, dass ein Load-Nachfolger mit einem weiteren Not-Verwender existiert. Diese Möglichkeit ist mit dem Existenzquantor markiert.

τ(x:Not(y:Load), ∃Not(y)) ⇒ x:ia32_Neg(y) { }

Auf der anderen Seite kann eine bestimmte Mustererweiterung verboten werden, was mit dem negierten Existenzquantor dargestellt ist. Die folgende Regel ist nur anzuwenden, wenn es keine weiteren Not-Verwender der Ladeoperation gibt.

τ(x:Not(y:Load(a:Node)), ∄Not(y)) ⇒ x:ia32_NegL(a) { }

PyGen kann keine Muster mit mehreren Wurzeln spezifizieren. Da dieser Mechanismus allerdings für wenige ausgewählte Fälle notwendig ist, wurden als Zwischenlösung einige wenige Spezialknoten erstellt. Ein Beispiel ist die Konsumierung einer volatilen Ladeoperation von einer Rechenoperation. Eine solche Ersetzungsregel darf nur angewendet werden, wenn der volatile Ladebefehl keinen zweiten Verwender besitzt, da sonst die Ladeoperation mehrmals durchgeführt werden würde und sich dadurch die Programmsemantik ändert. Mit der beschriebenen Sprache lässt sich dieses Muster allerdings darstellen:

τ(Add(l:Load(Node)), ∄Node(l))

Hilfsknoten und Mustervarianten

Da eine Musterbeschreibung oft redundante Teilmuster enthält, führen wir zusätzliche Hilfsknoten ein. So lassen sich beispielsweise ganze Ausdrücke einem Hilfsknoten zuweisen und später als Platzhalter verwenden. Im folgenden Beispiel wird ein Hilfsknoten helper in einem Ausdruck verwendet. Der zweite Ausdruck ist gleichbedeutend und entspricht dem Einsetzen des helper-Musters.

helper = Add(x:Const, Add(x, Node))
Add(helper, y:Node) ≡ Add(Add(x:Const, Add(x, Node)), y:Node)

Dies realisiert den PyGen-Mechanismus zur Angabe von Teilregeln. Vor allem wird diese Technik nützlich in Verbindung mit Varianten, die durch ∨ gekennzeichnet werden. Sie entsprechen dem Ausfalten von Regeln. Im folgenden Beispiel werden dem helper-Knoten zwei verschiedene Varianten zugewiesen.

helper = Add(Node, Node) ∨ Add(const:Const, Node)

Diese Varianten sind innerhalb von Ersetzungsregeln von Bedeutung, da für jede Variante eine eigene Regel erzeugt werden muss. Der Schablonenmechanismus von PyGen lässt sich durch Hilfsknoten ausdrücken.

helper = op2:Node ∨ immediate:Const
a:Add(op1:Node, helper) ⇒ a:ia32_lea(op1, op2 ∨ immediate)

Wenn ein Muster an mehreren Stellen Varianten besitzt, wird das kartesische Produkt gebildet. Dieses Ausfalten von Varianten kann allerdings durch Bedingungen eingeschränkt sein. Eine Regel, die unerfüllbare Bedingungen enthält, wird ignoriert. Die implizite Bedingung ist, dass auf der rechten Seite einer Ersetzung undeklariert verwendete Namen auf der linken Seite deklariert sein müssen. Im obigen Beispiel führt das dazu, dass für die op2-Variante im helper-Knoten auch auf der rechten Seite der Ersetzung nur die op2-Variante gültig ist.

Durch Varianten kann es vorkommen, dass Anweisungen einer Ersetzungsregel nicht ausgeführt werden können, da der entsprechende Knoten in der aktuellen Variante nicht existiert. In diesem Fall wird die Anweisung ignoriert. Für das obige Beispiel würde die Regelanweisung { a.imm ← immediate.value } nur für den Fall ausgeführt werden, dass helper als immediate:Const ersetzt wird. Für den Fall, dass helper = op2:Node ist, existiert kein Knoten mit Namen immediate und die Zuweisung wird ignoriert.

Ersetzungsvarianten

Unterschiedliche Ersetzungen in einer Regel können notwendig werden, wenn sich Varianten im Muster befinden. Im folgenden Beispiel wird, je nachdem, welchen Wert src.mode besitzt, ein Befehl zur Ganzzahl- oder Fließkommaaddition erzeugt.

a:Add(left:Node, right:Node) ⇒
    a:ia32_add(left, right) { src.mode is integer },
    a:ia32_fadd(left, right) { src.mode is float }

Die Angabe mehrerer Ersetzungsmöglichkeiten auf der rechten Seite führt dazu, dass das Muster links auch entsprechend mehrfach gefunden wird. Die Schreibweise ist äquivalent dazu mehrere Regeln mit identischen linken Seiten zu spezifizieren. Welche Regel letztendlich angewendet wird, hängt dann von Kostenmodell und PBQP-Löser ab. Falls allerdings Bedingungen im Ersetzungsrumpf stehen, müssen diese bei der Suche geprüft werden, was bedeutet, dass sich die Muster der linken Seite durchaus unterscheiden können.

6.4.2 Grenzen

Die beschriebene Sprache ist zur Spezifikation einer Befehlsauswahl. Sie besitzt allerdings bereits große Ähnlichkeit mit der Spezifikation einer ganzen Codegenerierung, wie sie beispielsweise BEG[ESL89] bietet. Die Sprache wäre vor allem um Registerbeschränkungen der einzelnen Befehle und Assemblercodierung zu erweitern, was sich leicht in den Knotenspezifikationen unterbringen ließe.

Zusätzlich müsste aber noch eine Maschinenbeschreibung entwickelt werden, die das Registerlayout beschreibt. So muss die Größe der einzelnen Register beschrieben werden und ob sie Teil eines anderen Registers sind. Zur Registerzuteilung muss es möglich sein, einzelne Register auszulagern, was im Detail von der Zielarchitektur abhängt.

7 Bewertung

Während wir uns im vorherigen Kapitel unter anderem mit den Sprachkonstrukten von PyGen befasst haben, betrachten wir nun die von PyGen generierte Befehlsauswahl. Als Referenz steht uns die handgeschriebene Befehlsauwahl der libFirm zur Verfügung, die auf einem Greedy-Algorithmus basiert. Da libFirm einen modularen Austausch der Befehlsauswahl-Verfahren erlaubt, kann die Befehlsauswahl unabhängig von den anderen Phasen des Codegenerators evaluiert werden.

7.1 Codequalität

Um Aussagen über die Qualität des erzeugten Codes machen zu können, verwenden wir Testprogramme aus der CINT2000-Sammlung der SPEC[1], die sich zum Vergleich verschiedener Übersetzer (und Prozessoren) etabliert hat.

In Tabelle 7.1 sind die gemessenen Laufzeiten der Programme der generierten Befehlsauswahl („PBQP“) und der handgeschriebenen Befehlsauswahl („libFirm“) zu sehen. Um zu zeigen, wie gut libFirm im Vergleich zu bekannten Übersetzern ist, wurde zusätzlich eine Messung mit GCC 4.2.1 vorgenommen. Die Testprogramme wurden mit den gleichen (Optimierungs-)Parametern übersetzt und auf einem Pentium 4 mit 2,4 GHz und 1 GB Arbeitsspeicher ausgeführt.

Die Messungen zeigen, dass unsere Befehlsauswahl im Durchschnitt geringfügig langsamere Programme erzeugt und somit keine Verbesserung gegenüber der handgeschriebenen Befehlsauswahl erzielen konnte. Mögliche Gründe für diese Laufzeitdifferenz sind:

1. Die Möglichkeiten der Zielarchitektur werden nicht voll ausgenutzt.
2. Das Kostenmodell spiegelt die realen Laufzeiten nur ungenügend wider.
3. Heuristische Entscheidungen führen zu PBQP-Lösungen, die stark von der optimalen Lösung abweichen.

Zur Untersuchung des dritten Grundes haben wir einen weiteren Lösungsalgorithmus implementiert, der mittels roher Gewalt die optimale Lösung einer PBQP-Instanz

[1] Standard Performance Evaluation Corporation

Testprogramm	GCC	libFirm	PBQP	$\frac{\text{libFirm}}{\text{PBQP}}$
164.gzip	187	182	186	97.7%
175.vpr	276	296	292	101.3%
176.gcc	123	130	135	96.5%
181.mcf	327	329	327	100.5%
186.crafty	135	124	134	92.5%
197.parser	257	261	264	98.8%
253.perlbmk	250	196	220	89.1%
254.gap	128	139	142	97.9%
255.vortex	200	206	214	96.3%
256.bzip2	236	242	244	99.2%
300.twolf	459	512	479	107.0%
			Durchschnitt	97.9%

Tabelle 7.1: SPEC Vergleich der Laufzeiten der generierten Programme zwischen PBQP- und handgeschriebener Befehlsauswahl.

ermittelt. Dabei greifen wir, analog zum heuristischen Lösungsalgorithmus, wenn möglich, auf informationserhaltende Reduktionen zurück. Trotz dieser Optimierung ist die Laufzeit im schlechtesten Fall exponentiell und skaliert somit nicht für große Programmgraphen.

Der in Tabelle 7.2 dargelegte Vergleich der beiden Algorithmen für die CINT2000 Programme zeigt, dass 99% der heuristischen Reduktionen die optimale Alternative ausgewählt haben. Weiter fällt auf, dass die Heuristik, gegenüber den informationserhaltenden $R1$- und $R2$-Reduktion, nur selten zur Anwendung kommt, weshalb insgesamt $99,9998\%$ aller Knoten optimal reduziert wurden. Der heuristische Lösungsalgorithmus des PBQP kann somit als Ursache für die angesprochene Laufzeitdifferenz ausgeschlossen werden.

Bei der Entwicklung von PyGen konnten wir beobachten, dass sich die Laufzeit der Programme durch zusätzliche Mustergraphen verbesserte. Wir vermuten deshalb, dass weitere Regeln eine noch bessere Laufzeit nach sich ziehen. Könnte man die in Abschnitt 6.3 beschriebenen Einschränkungen der Mustersuche aufheben, könnten weitere Regeln spezifiziert werden. Dies erscheint uns als der effektivste Weg zur weiteren Verbesserung der Laufzeiten.

Einen weiteren Ansatz zur Laufzeitverbesserung stellt das Kostenmodell dar. Unser Kostenmodell basiert auf statischen Musterkosten, die dynamisch mit der geschätzen Ausführungshäufigkeit des Befehls multipliziert werden. Dadurch wird die Auswahl an Knoten innerer Schleifen höher gewichtet als an Knoten, deren Befehle nur einmal ausgeführt werden. Allerdings erschwert dieses Vorgehen die Auswahl von Mustern, die sich über Schleifengrenzen hinweg erstrecken. Allgemein besteht die Frage, wann grundblockübergreifende Muster vorteilhaft sind. Falls sich für diese Frage gute Heuristiken finden lassen, ist ebenfalls eine Verbesserung der Laufzeiten zu erwarten.

Testprogramm	$R1$	$R2$	RN	sub	$\frac{\text{RN}}{\text{R1+R2+RN}}$	$\frac{\text{sub}}{\text{R1+R2+RN}}$
164.gzip	6.550	581	0	0	0%	0%
175.vpr	18.412	4.528	42	0	0,137%	0%
176.gcc	244.541	33.426	52	2	0,012%	0,00048%
181.mcf	1.782	233	1	0	0,033%	0%
186.crafty	24.263	6.819	12	0	0,031%	0%
197.parser	25.297	3.340	0	0	0%	0%
253.perlbmk	99.917	14.869	3	0	0,002%	0%
254.gap	80.558	19.402	47	0	0,034%	0%
255.vortex	112.548	17.750	9	0	0,005%	0%
256.bzip2	6.529	660	2	0	0,019%	0%
300.twolf	31.602	8.538	32	0	0,065%	0%
	651.999	110.146	200	2	0,018%	0,00018%

Tabelle 7.2: Trotz heuristischer RN-Reduktionen konnten fast alle Funktionen der SPEC-Programme optimal überdeckt werden.

7.2 Übersetzerlaufzeit

Die Leistungsfähigkeit eines Übersetzers wird zumeist an der Qualität des erzeugten Kompilats gemessen, jedoch gebührt der Laufzeit des Übersetzungsvorgangs ebensoviel Beachtung, da auch große Programme, wie z.B. Betriebssysteme, in akzeptabler Zeit übersetzt werden sollen.

Da unsere Befehlsauswahl sich klar in Phasen unterteilen lässt, wollen wir deren Laufzeit getrennt nach Suche, Auswahl und Ersetzung betrachten. Dazu haben wir die Laufzeiten für drei Programmgraphen verschiedener Größenordnungen in Tabelle 7.3 aufgetragen. Wie man erkennen kann, dominiert die Laufzeit der Mustersuche die Laufzeiten der PBQP-basierten Auswahl der Befehle und der anschließenden Ersetzung.

Knoten	Suche	Auswahl	Ersetzung	PBQP	libFirm	$\frac{\text{PBQP}}{\text{libFirm}}$
71	401 ms	1 ms	1 ms	925 ms	1 ms	925
1.449	420 ms	134 ms	9 ms	1.197 ms	10 ms	119
5.734	3.562 ms	34 ms	20 ms	4.225 ms	65 ms	65
54.948	7.781 ms	259 ms	146 ms	8.862 ms	588 ms	15
105.376	25.555 ms	1.005 ms	497 ms	29.265 ms	2.294 ms	13

Tabelle 7.3: Vergleich der Übersetzungslaufzeiten der generierten Befehlsauswahl (PBQP) mit der handgeschriebenen Befehlsauswahl (libFirm).

Weiter fällt auf, dass die generierte Befehlsauswahl (PBQP) selbst für kleine Programmgraphen bereits eine Laufzeit von einer Sekunde benötigt, was sich auf die

Initialisierung der Mustersuche zurückführen lässt. Die verwendete GrGen-Implementierung benötigt aufgrund der großen Menge an Regeln eine einmalige Initialisierungszeit von ca. 0,5 Sekunden. Diese Initialisierungszeit ist in der Suchphase nicht eingerechnet, die PBQP-Gesamtzeit ist also die Summe aus Initialisierung, Suche, Auswahl und Ersetzung. Zusätzlich fallen für jede Funktion ca. 0,4 Sekunden zur Anfertigung eines dynamischen Suchplans[Bat05] an. Bei großen Programmgraphen kommt den Initialisierungszeiten weniger Gewichtung zu und die Laufzeit der generierten Befehlsauswahl nähert sich der Laufzeit der handgeschriebenen Befehlsauswahl an. Das erreichte Verhältnis von 13 zeigt, dass die generierte Befehlsauswahl für große Funktionen skaliert.

Die hohen Initialisierungszeiten der Mustersuche führen zu einem schlechten Laufzeitverhalten bei Programmen mit vielen kleinen Funktionen. Abhilfe würde die Einführung eines statischen Suchplans schaffen, der die dynamischen Suchpläne ersetzt. Dadurch würde die Initialisierungszeit von 0,4 Sekunden für jede Funktion entfallen.

Auch die Arbeitsweise der Mustersuche kann besser an das Szenario der Befehlsauswahl angepasst werden. Zurzeit werden die Mustergraphen unabhängig voneinander gepasst, d.h. es werden Mustergraph für Mustergraph alle Vorkommen im Programmgraph gesucht. Allerdings haben viele Mustergraphen gemeinsame Teilgraphen, die bereits durch die Spezifikation identifiziert sind oder durch zusätzliche Analysen bestimmt werden können. Mit geeigneten Suchstrategien können diese gemeinsamen Teilgraphen wiederverwendet werden.

7.3 Spezifikationsumfang

Der größte Vorteil einer generierten Befehlsauswahl ist deren kompakte Spezifikation und die damit verbundene Erleichterung bei der Anbindung neuer Zielarchitekturen an die gegebene Zwischensprache. In diesem Abschnitt vergleichen wir den Umfang beider Befehlsauswahlvarianten in libFirm.

Unsere Spezifikation der Befehlsauswahl besteht im Kern aus dem Regelgenerator PyGen, der die über 7000 Ersetzungsregeln generiert. Der Rest der Implementierung, wie Mustersuche und Ersetzung, ist generisch und wird deswegen nicht zur Spezifikation gerechnet. In Tabelle 7.4 ist ein Überblick über die Länge der einzelnen Module von PyGen aufgeführt.

Datei	Zeilen	Funktion
am_test.py	143	Ersetzungsregeln
ext_ia32.py	75	Hilfsknoten
firm.py	577	Knoten der Zwischensprache
ia32.py	925	Knoten der Zielarchitektur
rules.py	74	Bibliothek
utils.py	83	Bibliothek
Insgesamt	1876	

Tabelle 7.4: Zeilenanzahl von PyGen gemessen mit `wc -l`.

Der IA-32-spezifische Teil von PyGen, bestehend aus den Knoten der Zielarchitektur, Hilfsknoten und den Ersetzungsregeln, umfasst weniger als 1200 Zeilen. Im Vergleich dazu besitzt die handgeschriebene Befehlsauswahl[2] fast 9000 Zeilen. Dieser Unterschied von fast einer Größenordnung erklärt sich durch die Verwendung einer domänenspezifischen Sprache und der daraus resultierenden kompakten Darstellung.

[2] Gerechnet wurden die Dateien `ia32_transform.c`, `ia32_spec.pl`, `ia32_common_transform.c` und `ia32_address_mode.pl` (Zeilen gezählt durch `wc -l`)

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Mit dieser Arbeit haben wir den Aufbau und die Lösung eines PBQP einer Befehlsauswahl formalisiert und gezeigt, unter welchen Voraussetzungen eine Lösung garantiert werden kann. Die beiden Kernbedingungen einer Mustermenge – „atomar-vollständig“ und „kompositional“ – sind einfach überprüfbar und hinreichend, um mit unserem Algorithmus für alle Firm-Programmgraphen eine gültige Befehlsauswahl zu treffen. Durch diese Verifikation kann sich der Entwickler auf die Zuverlässigkeit der verwendeten Algorithmen verlassen.

Anhand der Umsetzung einer praktisch verwendbaren Befehlsauswahl im libFirm-Rahmenwerk haben wir die Anforderungen an eine effektive Spezifikationssprache dokumentiert und anschließend eine Sprache entworfen, die mit einer übersichtlichen Anzahl an Mechanismen diese Anforderungen erfüllt. Eine Spezifikation in dieser Sprache bietet einen besseren Überblick als eine händische Implementierung. Voraussetzungen zur PBQP-Auswahl können statisch anhand der spezifizierten Regelmenge überprüft werden. Der Regelübersetzer kann den Entwickler über fehlende Muster informieren und fehlende Regeln aus anderen Regeln zusammensetzen. Unser entworfenes Werkzeug führt durch den Prozess der Spezifikation einer Befehlsauswahl zu einem robusten Produkt.

Wir haben eine Regelmenge für die Zielarchitektur IA-32 erstellt, um unsere generierte PBQP-Befehlsauswahl mit einer handgeschriebenen Befehlsauswahl vergleichen zu können. Die Geschwindigkeit der übersetzten Programme liegt, eine hinreichend umfangreiche Regelmenge vorausgesetzt, gleichauf mit anderen Übersetzern. Wir konnten in unserer Implementierung konkrete Verbesserungsmöglichkeiten identifizieren und einen effizienten Entwurf von Sprache und Implementierung angeben.

Unsere Arbeit liefert die Grundlage für ein Werkzeug zur Generierung einer Befehlsauswahl, die keine signifikanten Nachteile gegenüber einer händischen Programmierung zeigt. Solch ein Werkzeug kann zusätzlich eine Spezifikation auf Korrektheit überprüfen und dem Entwickler gezielte Hilfestellungen geben, um eine leistungsfähige Befehlsauswahl schnell und zuverlässig zu erstellen.

8.2 Ausblick

Im Rahmen dieser Diplomarbeit waren einige interessante Ideen aus Zeitgründen nicht weiter zu verfolgen.

8.2.1 Verbesserte Mustersuche

Wie schon in Abschnitt 6.3 und Abschnitt 7.2 angesprochen, ist die Mustersuche ein Problem in Hinblick auf die Laufzeit des Übersetzers. Verbessern würde sich diese Situation durch das Zusammenlegen der Suchpläne. Statt für jede Ersetzungsregel einzeln den Programmgraphen zu durchsuchen, könnte man einen gemeinsamen Suchplan erstellen und den Programmgraphen nur einmal absuchen. Dazu müsste man die Suchpläne normalisieren und statt mehrerer Suchpläne *einen* Automaten generieren. Dieser Automat müsste dann für jeden Knoten des Programmgraphen einmal die Mustermenge absuchen und eine Menge an gefunden Ersetzungsregeln angeben. Zusätzlich können bereits gefundene Teilmuster wiederverwendet werden. Dieses Zusammenlegen sollte den Speicherverbrauch und die Größe des Übersetzers stark reduzieren. Der Programmgraph muss nur einmal statt mehrere tausend Male traversiert werden, was die Laufzeit drücken sollte.

Die Implementierung könnte durch einen alternativen Codegenerator in GrGen realisiert werden, allerdings müsste dazu GrGen in die Lage versetzt werden gemeinsame Teilmuster zu identifizieren. Teilgraphisomorphie ist allerdings ein NP-vollständiges Problem. Einfacher ist es direkt aus der Spezifikationssprache ein Suchprogramm zu generieren. Man könnte dadurch das Ausfalten in tausende von Regeln vermeiden und direkt eine Mustersuche wie beispielsweise bei CGGG[Boe98] generieren ohne GrGen zu benutzen.

Ein zweites Problem der Mustersuche durch GrGen ist die Schwierigkeit, das Ignorieren von irrelevanten Konversionen zu integrieren. Diese Verbesserung sollte in der Mustersuche integriert sein, ohne dass Muster diese Varianten explizit angeben müssen. Durch Sterngraphgrammatiken ließe sich diese Verbesserung als Muster formulieren, allerdings ist diese Funktionalität in der von uns verwendeten GrGen-Version nicht enthalten.

8.2.2 Verbessertes Kostenmodell

Das von uns implementierte Kostenmodell ist sehr einfach. So wird Rematerialisierung zwar unterstützt, aber aufgrund des Kostenmodells selten eingesetzt. Unser Modell impliziert ein ähnliches Verhalten wie die bisherige Befehlsauswahl, nämlich

ein bevorzugtes Auswählen möglichst großer Muster und die Transformation in möglichst wenig Befehle.

Je nach Verwendungsgebiet werden unterschiedliche Anforderungen an den Übersetzer gestellt. In einem Fall mag die Größe des erzeugten Codes wichtiger sein und in einem anderen Fall die Ausführungsgeschwindigkeit. Übersetzer wie beispielsweise der `gcc` ermöglichen es durch Argumente wie `-Os` oder `-O3` die Übersetzung zu beeinflussen. Es ist denkbar, ein parametrierbares Kostenmodell zu entwerfen, das diese Argumente miteinbezieht.

8.2.3 Muster mit mehreren Wurzeln

Da unsere Implementierung vereinzelt[1] mehrfachgewurzelte Muster benutzt, hat sich gezeigt, dass die PBQP-Modellierung auch diesen Fall verarbeiten kann. Die von uns entworfene Sprache unterstützt die Spezifikation entsprechender Ersetzungsregeln. Wiedermann et al.[EBS+08] implementierte ein Verfahren, das eine Zyklenbildung, die durch Verwendung solcher Muster entstehen kann, verhindert. Diese Tatsachen deuten darauf hin, dass es möglich sein sollte, solche Muster zu verwenden. Inwiefern der Beweis der Garantie einer Lösung für diesen Fall erweitert werden kann, ist allerdings unklar.

Muster mit mehreren Wurzeln sind allerdings sehr nützlich, da sie die Verwendung reichhaltiger Befehle ermöglichen. Schösser[Sch07] entwickelte ein Verfahren, um derartige Befehle auf einfache Art zu spezifizieren und in einer Befehlsauswahl zu nutzen. Die Abbildung auf PBQP bietet genügen Flexibilität um besondere Eigenschaften von Befehlen in das Kostenmodell zu integrieren. Für SIMD-Befehle werden auf IA-32 spezielle Register benutzt, so dass zur Anwendung möglicherweise zusätzliche Transferkosten anfallen. Diese Eigenart ließe sich beispielsweise durch entsprechende Einträge in den Kostenmatrizen modellieren.

8.2.4 Verifikation von Erweiterungen des PBQP-Aufbaus

Der von uns verifizierte PBQP-Aufbau lässt sich um verschiedene Aspekte erweitern. Beispiele sind die im obigen Abschnitt angesprochenen Mustergraphen mit mehreren Wurzeln, oder auch die in Abschnitt 5.3.1 thematisierte Erweiterung um Rematerialisierung. Für diese Erweiterungen existiert keine Verifikation der Korrektheit, was eine große Hürde für den praktischen Einsatz darstellt. Während unser Formalisierungsansatz auch für diese Erweiterungen geeignet ist, muss der Beweis, angefangen

[1] Diese Mehrfachwurzeln treten in Verbindung mit Proj-Knoten auf, die in PyGen nur implizit spezifiziert werden.

beim Begriff der Überdeckung, angepasst werden. Dabei ist zu prüfen, ob zusätzliche Bedingungen notwendig sind, um eine korrekte Befehlauswahl zu garantieren.

8.2.5 Schwächere Bedingungen durch automatische Verifikation

Nach unsere Meinung lassen sich Gleichung 4.6 und Gleichung 4.7 der Kompositionalität abschwächen. Eine Verifikation von schwächeren Bedingungen ist allerdings aufwendig und fehleranfällig. Deshalb sollte eine Formalisierung des Problems in Prädikatenlogik erster Stufe vorgenommen werden, die den Einsatz von automatischen Beweisern zur Korrektheits-Garantie der PBQP-Befehlsauswahl ermöglicht.

Ein weiterer Schritt wäre die Verifikation bzgl. einer gegebenen Mustermenge, ohne dabei den „Umweg“ über gegebene Eigenschaften der Mustermenge zu beschreiten. Es muss sich zeigen, ob die Beweiser in der Lage sind, die strukturellen Eigenschaften einer Mustermenge automatisch zu erkennen. Ist dies der Fall, so kann die Anzahl der zusätzlich generierten Mustergraphen auf ein Minimum reduziert werden.

Literaturverzeichnis

[Bat05] Batz, Gernot V.: *Graphersetzung für eine Zwischendarstellung im Übersetzerbau*, Universität Karlsruhe, IPD Goos, Diplomarbeit, 2005. http://www.info.uni-karlsruhe.de/papers/da_batz.pdf

[BCHS98] Briggs, Preston ; Cooper, Keith D. ; Harvey, Timothy J. ; Simpson, Taylor L.: Practical improvements to the construction and destruction of static single assignment form. In: *Softw. Pract. Exper.* 28 (1998), July, Nr. 8, 859–881. http://portal.acm.org/citation.cfm?id=295551. – ISSN 0038–0644

[Boe98] Boesler, Boris: *Codeerzeugung aus Abhängigkeitsgraphen*, Universität Karlsruhe (TH), IPD, Diplomarbeit, Jun 1998. http://www.info.uni-karlsruhe.de/~boesler/thesis.ps.gz

[CQOB92] Cohen, Guy ; Quadrat, Jean-Pierre ; Olsder, Geert J. ; Baccelli, François: *Synchronization and Linearity, An Algebra for Discrete Event Systems*. 1992

[EBS+08] Ebner, Dietmar ; Brandner, Florian ; Scholz, Bernhard ; Krall, Andreas ; Wiedermann, Peter ; Kadlec, Albrecht: Generalized instruction selection using SSA-graphs. In: *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–104–0, S. 31–40

[EKS03] Eckstein, Erik ; König, Oliver ; Scholz, Bernhard: Code Instruction Selection Based on SSA-Graphs. In: *SCOPES*, 2003, S. 49–65

[Ert99] Ertl, M. A.: Optimal code selection in DAGs. In: *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1999. – ISBN 1–58113–095–3, S. 242–249

[ESL89] Emmelmann, H. ; Schröer, F.-W. ; Landwehr, Rudolf: BEG: a generator for efficient back ends. In: *SIGPLAN Not.* 24 (1989), Nr. 7, S. 227–237. http://dx.doi.org/10.1145/74818.74838. – DOI 10.1145/74818.74838. – ISSN 0362–1340

[FHP92] Fraser, Christopher W. ; Henry, Robert R. ; Proebsting, Todd A.: BURG: fast optimal instruction selection and tree parsing. In: *SIGPLAN Not.* 27 (1992), Nr. 4, S. 68–76. http://

dx.doi.org/http://doi.acm.org/10.1145/131080.131089. – DOI http://doi.acm.org/10.1145/131080.131089. – ISSN 0362–1340

[GBG+06] Geiss, Rubino ; Batz, Gernot V. ; Grund, Daniel ; Hack, Sebastian ; Szalkowski, Adam M.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, A. (Hrsg.) ; Ehrig, H. (Hrsg.) ; Montanari, U. (Hrsg.) ; Ribeiro, L. (Hrsg.) ; Rozenberg, G. (Hrsg.): *Graph Transformations - ICGT 2006*, Springer, 2006 (Lecture Notes in Computer Science), 383 – 397. – Natal, Brasil

[GFH82] Ganapathi, Mahadevan ; Fischer, Charles N. ; Hennessy, John L.: Retargetable Compiler Code Generation. In: *ACM Comput. Surv.* 14 (1982), Nr. 4, S. 573–592. http://dx.doi.org/10.1145/356893.356897. – DOI 10.1145/356893.356897. – ISSN 0360–0300

[GG78] Glanville, R. S. ; Graham, Susan L.: A new method for compiler code generation. In: *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA : ACM, 1978, S. 231–254

[GTB+08] Geiss, Rubino ; Taentzer, Gabriele ; Biermann, Enrico ; Bisztray, Dénes ; Bohnet, Bernd ; Boneva, Iovka ; Boronat, Artur ; Geiger, Leif ; Horvath Ákos ; Kniemeyer, Ole ; Mens, Tom ; al, Benjamin N.: Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In: Schürr, A. (Hrsg.) ; Nagl, M. (Hrsg.) ; Zündorf, A. (Hrsg.): *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)* Bd. NN, Springer, 2008, S. 514–539. – http://www.springerlink.com/content/105633/

[Hac07] Hack, Sebastian: *Register Allocation for Programs in SSA Form*, Universität Karlsruhe, Diss., October 2007. http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532

[HO82] Hoffmann, Christoph M. ; O'Donnell, Michael J.: Pattern Matching in Trees. In: *J. ACM* 29 (1982), Nr. 1, S. 68–95. http://dx.doi.org/10.1145/322290.322295. – DOI 10.1145/322290.322295. – ISSN 0004–5411

[Jak04] Jakschitsch, Hannes: *Befehlsauswahl auf SSA-Graphen*, IPD Goos, Diplomarbeit, November 2004. http://www.info.uni-karlsruhe.de/papers/da_jakschitsch.pdf

[Jak08] Jakumeit, Edgar: *Mit GrGen.NET zu den Sternen – Erweiterung der Regelsprache eines Graphersetzungswerkzeugs um rekursive Regeln mittels Sterngraphgrammatiken und Paargraphgrammatiken*, Universität Karlsruhe (TH), IPD, Diplomarbeit, jul 2008. http://www.info.uni-karlsruhe.de/papers/da_jakumeit.pdf

[NK97] Nymeyer, A. ; Katoen, J. p.: Code generation based on formal BURS theory and. In: *Acta Informatica* 34 (1997), Nr. 8, S. 597–635. http://

dx.doi.org/10.1007/s002360050099. – DOI 10.1007/s002360050099. – ISSN 0001–5903 (Print) 1432–0525 (Online)

[PLG88] Pelegrí-Llopart, E. ; Graham, S. L.: Optimal code generation for expression trees: an application BURS theory. In: *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1988. – ISBN 0–89791–252–7, S. 294–308

[Pyt] *Python Programming Language*. http://www.python.org

[Sch07] Schösser, Andreas: *Graphersetzungsregelgewinnung aus Hochsprachen und deren Anwendung*, Universität Karlsruhe (TH), IPD, Diplomarbeit, 9 2007. http://www.info.uni-karlsruhe.de/papers/da_schoesser.pdf

[SE02] Scholz, Bernhard ; Eckstein, Erik: Register allocation for irregular architectures. In: *LCTES-SCOPES*, ACM, 2002. – ISBN 1–58113–527–0, S. 139–148

[TLB99] Trapp, Martin ; Lindenmaier, Götz ; Boesler, Boris: Documentation of the Intermediate Representation FIRM / Universität Karlsruhe, Fakultät für Informatik. Universität Karlsruhe, Fakultät für Informatik, Dec 1999 (1999-14). – Forschungsbericht. – 0–40 S. – http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz

[Tra01] Trapp, Martin: *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.*, Universität Karlsruhe, Fakultät für Informatik, Diss., Oct. 2001. http://www.amazon.de/exec/obidos/ASIN/3540423524/qid=1039790257/sr=1-1/

[WGHB95] Wilson, T ; Grewal, G ; Henshall, S ; Banerji, D: An ILP-Based Approach to Code Generation. In: *Code Generation for Embedded Processors*, Kluwer, 1995, S. 103–118

[Wie08] Wiedermann, Peter: *A generalized PBQP instruction selector for the LLVM compiler framework*, Technische Universität Wien, Diplomarbeit, 2008

Abbildungsverzeichnis

Index

Danksagung

Wir danken Professor Goos, dass er diese Doppel-Diplomarbeit ermöglicht hat, und Rubino Geiß für seine Motivation und Unterstützung. Besonderer Dank geht an Andreas Lochbihler für die Korrektur des Korrektheitsbeweises und an Edgar Jakumeit für die ausführlichen Kommentare zu unserer Arbeit. Auch allen anderen Korrekturlesern herzlichen Dank für all die Tips und Verbesserungsvorschläge. Für die Hilfe bei der Implementierung und den Details von libFirm danken wir Matthias Braun und Michael Beck, sowie Christoph Mallon, der auch bis in die Nacht mit uns diskutierte. Ein weiteres Dankeschön geht an Katja Weisshaupt für die liebevolle Erledigung des Papierkriegs und Bernd Traub für Pflege und Aufzucht der kleinen und großen Rechenmonster.

Weiterer Dank geht an alle Teilnehmer des 17-Uhr-Entspannungsrituals, welches den Arbeitstag immer wieder aufgelockert hat. Ebenfalls danken wollen wir dem Ehepaar Kroll und allen anderen, die mit Kuchen, Keksen und anderen Köstlichkeiten die Küche zu einem Platz der Gemeinschaft und des Austauschs gemacht haben.

Andreas Zwinkau dankt außerdem seiner Freundin Anne-Kristin, die den Stress geduldig mitgetragen hat, und seinen Eltern für die treue Unterstützung. S.D.G.

Sebastian Buchwald dankt seiner Familie für die Liebe und Unterstützung während des Studiums.