Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

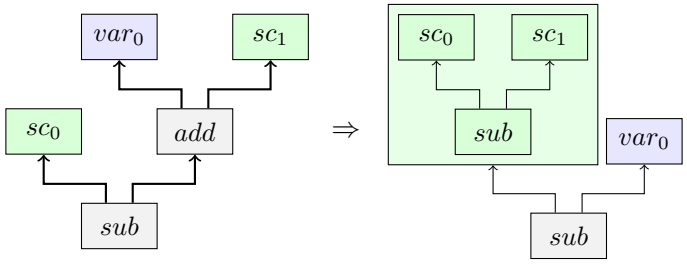
Lehrstuhl Prof. Dr.-Ing. Snelting

Generierung lokaler Optimierungen

Diplomarbeit von

Thomas Bersch

an der Fakultät für Informatik



$$
\begin{aligned}
0-(x+0) &\rightarrow 0-x \\
0-(x+1) &\rightarrow -1-x \\
0-(x+2) &\rightarrow -2-x \\
0-(x+3) &\rightarrow -3-x
\end{aligned}
$$

Gutachter: Prof. Dr.-Ing. Gregor Snelting

Betreuender Mitarbeiter: Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 24. Februar 2012 – 14. August 2012

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 14. August 2012

Kurzfassung

Lokale Optimierungen bilden einen wichtigen Teil, der von einem Übersetzer durchgeführten Optimierungen. Dabei handelt es sich zumeist um handgeschriebene Ersetzungsregeln, deren Umfang von den Kenntnissen der Entwickler des Übersetzers abhängig ist. Vor allem Konstanten werden bei der Betrachtung von Optimierungsregeln häufig außer Acht gelassen.

In dieser Arbeit wird ein neues Verfahren mit Namen Optgen vorgestellt, welches in der Lage ist lokale Optimierungsregeln automatisch zu generieren und dabei sämtliche Konstanten einer gegebenen Bitbreite zu berücksichtigen. Die erzeugten Optimierungsregeln werden zudem auf ihre Korrektheit hin bewiesen und in einer Regelmenge zusammengefasst, welche die Transformation einer Codesequenz in eine, bezogen auf ein Kostenmodell, optimale Codesequenz ermöglicht. Um die Menge der Konstanten berücksichtigen zu können, wird ein neues Konzept eingeführt, das es ermöglicht von Optimierungsregeln mit konkreten Konstanten zu abstrahieren und so eine Aggregation der zu berücksichtigenden Regeln zu erreichen.

Inhaltsverzeichnis

1 Einführung

Die Arbeit eines Übersetzers unterteilt sich in die Analyse des zu übersetzenden Quellprogramms und die Synthese in das gewünschte Zielprogramm. Der Übergang zwischen den beiden Phasen erfolgt in der Regel über eine Zwischendarstellung des Programms in einer maschinenunabhängigen Form, woraus im Anschluss das maschinenabhängige Zielprogramm erstellt wird. Wesentlich für die Effizienz des erzeugten Zielprogramms sind Optimierungen, also Transformationen der Zwischen- oder Zieldarstellung die während der Optimierungsphase vom Übersetzer vorgenommen werden.

Grundlegend lassen sich globale und lokale Optimierungen unterscheiden. Während globale Optimierungen eine Vielzahl von Informationen berücksichtigen und sich in der Regel über mehrere Grundblöcke erstrecken, berücksichtigen lokale Optimierungen nur wenige Anweisungen. In dieser Arbeit werden ausschließlich lokale Optimierungen betrachtet.

Lokale Optimierungen gibt es sowohl für die Zwischendarstellung als auch für die Zieldarstellung. Letztere ermöglicht oft zusätzliche maschinenabhängige Optimierungen, wie das Ausnutzen von Flags (Überlaufbit, Übertragsbit, etc.), die vom Statusregister eines Prozessors bereit gestellt werden, oder die Verwendung von Spezialbefehlen. Die erzielbare Verbesserung durch lokale Optimierung ist abhängig davon, wie viele lokale Optimierungsmöglichkeiten im Code vorhanden sind, wie viele davon durch den Übersetzer gefunden werden und ob die verwendeten Ersetzungsregeln die optimale Zieldarstellung erzeugen oder lediglich eine Verbesserung erreichen. Da es sich bei lokalen Optimierungen in heutigen Übersetzern meist um handgeschriebene Ersetzungsregeln handelt, ist die Anzahl der auffindbaren Optimierungen von den Kenntnissen des jeweiligen Entwicklers abhängig. Eine Optimierung kann nur gefunden und genutzt werden, wenn sie zum einen bekannt und zum anderen berücksichtigt wird.

1.1 Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung eines Verfahrens, welches in der Lage ist, mittels automatischer Mechanismen, systematisch Ersetzungsregeln zur lokalen Optimierung zu generieren. Hiermit soll es möglich sein, Entwicklern eine Auflistung an die Hand zu geben, in der alle lokalen Optimierungen bis zu einer festen Größe enthalten sind. Insbesondere sollen auch Konstanten berücksichtigt werden. Die Korrektheit der

so gewonnenen Optimierungsregeln soll unter Zuhilfenahme eines geeigneten Verfahrens (z. B. SAT-Solver oder SMT-Solver) bewiesen werden. Da zu erwarten ist, dass vor allem durch Konstanten eine Vielzahl strukturell ähnlicher Regeln generiert werden, sollen diese in möglichst allgemeinen Regeln zusammengefasst werden. Hierzu soll untersucht werden, wie eine solche Aggregation zu erreichen ist.

1.2 Aufbau der Arbeit

In Kapitel 2 werden zunächst die notwendigen Grundlagen behandelt. Es wird genauer auf das Thema lokale Optimierungen eingegangen. Des weiteren werden notwendige Formalisierungen eingeführt und das Konzept der Superoptimierung beschrieben. Zuletzt wird das SAT-Problem und dessen Erweiterung Satisfiability Modulo Theories, kurz SMT sowie die Zwischensprache Firm vorgestellt. In Kapitel 3 werden einige verwandte Ansätze vorgestellt und sowohl untereinander als auch mit dem Ansatz dieser Arbeit verglichen. Darauf aufbauend wird in Kapitel 4 die konkrete Umsetzung dieser Arbeit beschrieben. Der besondere Schwerpunkt liegt dabei auf dem Umgang mit Konstanten bei der Generierung von lokalen Optimierungsregeln. Kapitel 5 belegt anhand von Messergebnissen die Durchführbarkeit dieses Ansatzes, sowie den Nutzen, aber auch dessen Grenzen. Den Abschluss bildet Kapitel 6, in welchem eine Zusammenfassung der Arbeit sowie ein Ausblick auf zukünftige Erweiterungen gegeben wird.

2 Grundlagen

2.1 Lokale Optimierungen

Der Begriff *lokale Optimierungen* [2] fasst eine Vielzahl unterschiedlicher Optimierungstechniken zusammen. Das Ziel solcher Optimierungen ist es, die Berechnung von Ausdrücken hinsichtlich eines Kostenmodells[1] zu vereinfachen. Allen lokalen Optimierungen ist gemein, dass sie nur wenige Instruktionen berücksichtigen und keine aufwendigen Analyse-Techniken wie beispielsweise eine Datenfluss-Analyse erfordern. Eingesetzt werden solche Optimierungen sowohl zur Verbesserung der von einem Übersetzer erzeugten Zwischendarstellung als auch der Zieldarstellung. Nachfolgend werden einige wichtige lokale Optimierungen vorgestellt (siehe auch [15]).

Konstantenfaltung Bei der Konstantenfaltung werden Ausdrücke, die lediglich konstante Operanden besitzen und somit nicht von den Eingabedaten des Programms abhängig sind, bereits während der Übersetzungszeit ausgerechnet und durch ihr Ergebnis ersetzt. Die entsprechenden Operationen können so eingespart werden. Beispielsweise kann der Ausdruck 3+5 direkt durch die Konstante 8 ersetzt werden. Die Addition wird nicht länger benötigt.

Algebraische Vereinfachungen Oft lassen sich Ausdrücke durch das Ausnutzen algebraischer Umformungen in eine einfachere bzw. kostengünstigere Form bringen, d. h. es werden weniger oder günstigere Operationen verwendet. Ein häufig gezeigtes Beispiel ist der Ausdruck $x + 0$, der zu x vereinfacht werden kann. Hier wird ausgenutzt, dass es sich bei 0 um das neutrale Element der Addition handelt. Ein weiteres Beispiel ist der Ausdruck $x \wedge 0$. Ist ein Operand der $\wedge$-Verknüpfung 0, so ist das Ergebnis unabhängig vom zweiten Operanden ebenfalls 0.

Operatorvereinfachung Einige Operationen lassen sich durch günstigere, semantisch äquivalente Operationen ersetzen. Beispielsweise lässt sich eine Multiplikation mit 2 durch eine Addition mit sich selbst ersetzen. Es gilt: $x * 2 = x + x$. Etwas allgemeiner gilt auch: Eine Multiplikation mit einer 2-er Potenz ist äquivalent zu einem Links-Shift um den Exponenten. Da sowohl die Addition als auch die Shift-Operation auf heutigen Architekturen deutlich schneller ausgeführt werden kann als eine Multiplikation, kann ein Austausch von Vorteil sein.

[1] z. B. Anzahl der Instruktionen, Summe der notwendigen Taktzyklen, etc.

Reassoziierung Die Reassoziierung nutzt die algebraischen Eigenschaften der involvierten Operationen wie Assoziativität, Kommutativität und Distributivität aus um Teilausdrücke neu anzuordnen. Dadurch kann erreicht werden, dass zwei Konstanten aus unterschiedlichen Teilausdrücken in einem Teilausdruck zusammengefasst werden und somit eine Konstantenfaltung möglich wird. Die folgende Gleichung zeigt hierzu ein Beispiel: $(x + 1) + (y + 1) = (x + y) + (1 + 1) = (x + y) + 2$.

2.2 Graphen, Muster und Regeln

Das in Kapitel 4 vorgestellte Verfahren orientiert sich an maschinenunabhängigen Zwischensprachen und betrachtet Ausdrücke, wie sie im vorherigen Abschnitt vorkommen, als Muster in Graphform. An dieser Stelle werden die dazu notwendigen theoretischen Ansätze eingeführt.

Definition 1 (Gerichteter Graph)**.** *Ein gerichteter Graph G entspricht einem 4-Tupel $G = (V, E, src, dst)$ mit der Knotenmenge V, einer Kantenmenge E, sowie zwei Abbildungen $src \to V$ und $dst : E \to V$. $src(e)$ ordnet jeder Kante e einen Quellknoten und $dst(e)$ jeder Kante e einen Zielknoten zu.*

Definition 2 (Pfad)**.** *Sei G ein gerichteter Graph, E die Menge der Kanten von G, sowie $P = (e_0, \ldots, e_n), n \in \mathbb{N}_+$ eine Folge von Kanten aus E und i aus $\{0, \ldots, n-1\}$. P heißt Pfad, wenn für jede Kante $e_i \in E$ der Folge P gilt:*

$$dst(e_i) = src(e_{i+1})$$

Definition 3 (Zyklus)**.** *Sei G ein gerichteter Graph. Ein Zyklus C in G entspricht einem Pfad $P = (e_0, \ldots, e_n)$ für dessen Startknoten $src(e_0)$ sowie dessen Zielknoten $dst(e_n)$ gilt:*

$$src(e_0) = dst(e_n)$$

Definition 4 (Gerichteter azyklischer Graph)**.** *Ein gerichteter Graph G heißt azyklisch, wenn kein Zyklus C in G existiert.*

Definition 5 (Gewurzelter Graph)**.** *Ein gerichteter azyklischer Graph G heißt gewurzelt, wenn genau ein Knoten $v_w \in V$ existiert, so dass für alle anderen $v \in V$ ein Pfad $P = (e_0, \ldots, e_n)$ existiert, für den gilt: $src(e_0) = v_w \wedge dst(e_n) = v$. Der Knoten v_w wird als Wurzel von G bezeichnet.*

Mit den bisherigen Definitionen kann nun ein Ausdruck als graphbasiertes Muster wie folgt formal eingeführt werden:

Definition 6 (Muster)**.** *Ein Muster m entspricht einem 4-Tupel $m = (G, type, pos, idx)$ mit einem gewurzelten Graphen $G = (V, E, src, dst)$, einer Abbildung $type : V \to T$, einer Abbildung $pos : E \to \mathbb{N}_0$ und einer Abbildung $idx : V \to \mathbb{N}_0$.*

Die Abbildung type ordnet jedem Knoten $v \in V$ einen Typ $t \in T$ zu. T repräsentiert die Menge der zugelassenen Typen. Jede Operation, jede Konstante und die Variable bilden einen eigenen Typ t. Jedem dieser Typen wird durch die Abbildung $arity : T \to \mathbb{N}_0$ sein Ausgangsgrad zugeordnet. Jedem Typ t wird zudem durch die Abbildung $eval : T \times \mathbb{Z}^{arity(t)} \to \mathbb{Z}$ ein Ergebnis bezüglich seiner Semantik und einem Eingabevektor $tv \in \mathbb{Z}^{arity(t)}$ zugeordnet.

Die Abbildung pos ordnet jeder Kante $e \in E$ ihre Position bezüglich des Ausgangsgrades $arity(type(src(e)))$ des Quellknoten der Kante e zu. Für ein Muster mit Knotenmenge V und Kantenmenge E muss zudem gelten:

$$\forall e \in E : pos(e) < arity(type(src(e)))$$

$$\forall e, e' \in E : src(e) = src(e') \wedge pos(e) = pos(e') \Rightarrow e = e'$$

$$\forall v \in V : |\{e \in E_G : src(e) = v\}| = arity(type(v))$$

Die Abbildung idx ordnet jedem Knoten v einen Index zu, so dass gilt:

$$idx(v) < |\{v' \in V : type(v') = type(v)\}|$$

$$\forall v' \in V : idx(v') = idx(v) \wedge type(v') = type(v) \Rightarrow v = v'$$

Die Indizes aus der Abbildung *idx* werden immer pro Typ vergeben, d. h. eine Addition kann beispielsweise denselben Index wie eine Variable erhalten. Über den Index wird eine Ordnung der Knoten eines Typs innerhalb eines Muster erreicht. Vor allem für die Zuordnung von konkreten Zahlenwerten zu Variablen wird dies benötigt (vgl. Definition 9).

Um zwei Muster hinsichtlich eines Kostenmodells vergleichen zu können, wird eine Kostenfunktion benötigt, die jedem Muster einen Kostenwert entsprechend des gewählten Kostenmodells zuordnet. Dazu werden die beiden folgenden Definitionen eingeführt:

Definition 7 (Kosten eines Typs)**.** *Sei T die Menge der zugelassenen Typen. Die Abbildung $cost_T : T \to \mathbb{N}_0$ ordnet jedem Typ $t \in T$ seinen Kostenwert zu.*

Definition 8 (Kosten eines Musters)**.** *Sei m ein Muster, sowie $cost_M : M \to \mathbb{N}_0$ eine Abbildung, die jedem Muster einen Kostenwert zuordnet. Für die Kosten $cost_M(m)$ eines Musters m gilt:*

$$cost_M(m) = \sum_{v \in V} cost_T(type(v))$$

Anders ausgedrückt, die Kosten eines Muster m ergeben sich aus der Summe der Kosten jedes einzelnen Knotens v des Musters.

Definition 9 (Ergebnis eines Musters)**.** *Sei m ein Muster, G der zu m gehörende Graph, V die Menge der Knoten von G, sowie $n_{var} = |\{v \in V : type(v) = var\}|$ die Anzahl der Knoten des Musters mit dem Typ einer Variablen ($type(v) = var$). Sei weiter $tv \in \mathbb{Z}^{n_{var}}$ ein Eingabevektor, der jedem Knoten $v \in V$, mit $type(v) = var$ einen Wert (Konstante) $val = tv[idx(v)]$ zuordnet. val bildet selbst wieder eine Vektor $(val)^1$, so dass $eval(type(v), (val)^1)$ den Wert des Variablen-Knoten berechnet. Das Ergebnis eines Musters $m(tv)$ ergibt sich durch eine Postorder-Traversierung[1] von G, bei der für jeden Knoten v' die Abbildung $eval$ bezüglich der Ergebnisse seiner Vorgänger-Knoten ausgewertet wird.*

Abbildung 2.1 zeigt zu Definition 9 ein Beispiel. Der Eingabevektor $tv = (1, 2)$ wird auf das in der Abbildung gezeigte Muster mit der Semantik $(var_0 + var_1) + 2$ angewendet. Wie in der Definition beschrieben, wird in Postorder-Reihenfolge jeder Knoten durch die *eval* Abbildung ausgewertet. Es ergibt sich das Ergebnis $m(tv) = 5$.

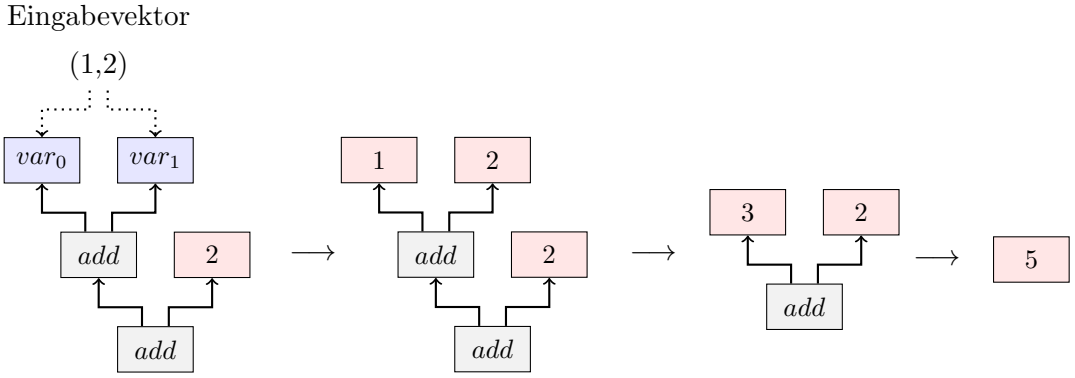


Abbildung 2.1: Beispiel zur Berechnung des Ergebnisses eines Musters bezüglich eines Eingabevektors.

Für die Aufstellung von Ersetzungsregeln ist es notwendig zu wissen, ob zwei Muster m und m' für alle Eingabevektoren jeweils dasselbe Ergebnis berechnen. Man spricht von semantischer Äquivalenz. Formal gilt:

Definition 10 (Semantische Äquivalenz)**.** *Seien m und m' zwei Muster, sowie V_m die Menge der Knoten von m und $V_{m'}$ die Menge der Knoten von m'. m und m' sind genau dann semantisch äquivalent, wenn für alle Eingabevektoren $tv \in \mathbb{Z}^n$, mit $n = max(|\{v \in V_m : type(v) = var\}|, |\{v \in V_{m'} : type(v) = var\}|)$ gilt:*

$$m(tv) = m'(tv)$$

Für zwei semantisch äquivalente Muster m und m' schreibt man: $m \equiv m'$.

[1] zuerst wird der linke Teilbaum, dann der rechte Teilbaum und zum Schluss die Wurzel behandelt

Definition 11 (Optimale Muster)**.** *Sei M eine Menge von Mustern. Ein Muster $m \in M$ ist genau dann optimal, wenn kein semantisch äquivalentes Muster $m' \in M$ existiert für das gilt:*

$$cost_M(m') < cost_M(m)$$

Mit den bisherigen Definitionen kann nun eine lokale Optimierung in Form einer Ersetzungsregel, kurz Regel, wie folgt formal eingeführt werden:

Definition 12 (Regel)**.** *Eine Regel $r = (m_l, m_r)$ besteht aus zwei Mustern m_l und m_r, für die gilt: $m_l \equiv m_r$. m_l wird linke Seite und m_r rechte Seite der Regel genannt. Ferner muss für die Kosten der beiden Muster gelten: $cost_M(m_l) \geq cost_M(m_r)$. Gilt $cost_M(m_l) > cost_M(m_r)$, so stellt r eine Optimierungsregel dar, gilt hingegen $cost_M(m_l) = cost_M(m_r)$ so wird r als Transformationsregel bezeichnet. Alternativ kann man für $r = (m_l, m_r)$ auch $m_l \to m_r$ schreiben.*

In Definition 12 wird der Fall eines teureren Musters auf der rechten Seite einer Regel explizit ausgeschlossen.

2.3 Superoptimierung

Im Kontext des Übersetzerbaus wird unter *Optimierung* die Transformation eines Ausdrucks, Musters oder einer Codesequenz verstanden, mit dem Ziel, diesen hinsichtlich eines Kostenmodells wie z. B. die erforderliche Rechenzeit zu verbessern. Dass das Ergebnis optimal ist oder das überhaupt eine Verbesserung erreicht wird, ist nicht garantiert. Ein Beispiel zeigt der folgende Ausdruck: $x - (x + 0)$. Eine anwendbare lokale Optimierung ist die Ersetzung des Teilausdrucks $x + 0$ durch x. Damit wird der gesamte Ausdruck zwar optimiert, indem eine Addition eingespart wird, aber noch nicht das optimale Ergebnis 0 erreicht.

An dieser Stelle setzt das Prinzip der *Superoptimierung* an. Der Begriff Superoptimierung umschreibt einen Vorgang, der darauf ausgerichtet ist, zu einem gegebenen Ausdruck bzw. Muster den optimalen Ausdruck mit gleicher Semantik zu ermitteln. Anders ausgedrückt bedeutet dies, dass zu einer gegebenen Codesequenz die einen Ausdruck repräsentiert, eine semantisch äquivalente Codesequenz gesucht wird, die bezüglich des gewählten Kostenmodells, wie die Summe der Instruktionen oder die Anzahl der notwendigen Taktzyklen, optimal ist.

Eingeführt wurde der Begriff der Superoptimierung erstmals von Massalin [14] worauf in Kapitel 3 näher eingegangen wird. Anwendung findet dieses Prinzip heute in verschiedenen Arbeiten wie [3] und [11].

2.4 Das Erfüllbarkeitsproblem der Aussagenlogik

Bei dem Erfüllbarkeitsproblem der Aussagenlogik [7], auch SAT genannt[1], handelt es sich um die Fragestellung, ob zu einer aussagenlogischen Formel F, bestehend aus booleschen Variablen $B = \{b_0, \ldots, b_n\}$ und logischen Operationen *und* ($\wedge$), *oder* ($\vee$) und *nicht* ($\neg$), eine Belegung $\beta : B \to \{wahr, falsch\}$ existiert, so dass F unter β zu *wahr* ausgewertet wird.

Beispiel 1 (Erfüllbares SAT Problem)**.** *Sei F die nachfolgend beschriebene aussagenlogische Formel mit den booleschen Variablen b_0, b_1:*

$$F = b_0 \wedge (b_0 \vee \neg b_1)$$

Mit $\beta : b_0 \mapsto wahr, b_1 \mapsto wahr$ existiert eine Belegung, so dass F zu wahr ausgewertet werden kann. F ist somit erfüllbar.

Beispiel 2 (Nicht-erfüllbares SAT Problem)**.** *Sei G die nachfolgend beschriebene aussagenlogische Formel mit den booleschen Variablen b_0, b_1:*

$$G = b_0 \wedge (\neg b_0 \vee b_1) \wedge (\neg b_0 \vee \neg b_1)$$

Für G existiert keine gültige Belegung β und somit ist G auch nicht erfüllbar.

SAT gehört zur Klasse der NP-vollständigen Probleme [7, 12]. Fortschritte in der Algorithmentechnik, der Einsatz von Heuristiken, effiziente Nutzung moderner Microprozessoren etc., haben in den letzten Jahren dazu beigetragen, dass heute Verfahren existieren mit denen das SAT-Problem trotz NP-Vollständigkeit gut handhabbar geworden sind. Moderne SAT-Solver sind in der Lage auch komplexere Instanzen eines SAT-Problems mit Hunderttausenden von Variablen zu lösen [9].

2.5 Satisfiability Modulo Theories

Bei dem Satisfiability Modulo Theories (SMT) Problem [5, 9] handelt es sich, wie bei SAT auch um ein Erfüllbarkeitsproblem für Formeln der Aussagenlogik (vgl. Abschnitt 2.4). Während bei SAT nur boolesche Variablen als Prädikate innerhalb einer Formel zulässig sind, können bei SMT auch Formeln mit komplexeren Prädikaten aus den unterschiedlichsten Theorien verwendet werden. Beispiele für diverse Theorien sind die Integer-Arithmetik, die Bitvektor-Arithmetik, Array-Theorie und die Theorie über Datenstrukturen.

Ein Beispiel einer solchen aussagenlogischen Formel mit zusätzlichen Theorien zeigt die nachfolgende Formel F:

$$F = (x + 5) > 6 \wedge (x + 5) < 9 \wedge \neg(x = 2), x \in \mathbb{N}_0 \tag{2.1}$$

[1] SAT, vom englischen satisfiability

Die Formel setzt sich zusammen aus der folgenden Instanz G des SAT-Problems mit den Prädikaten p_0, p_1, p_2 aus der Theorie der Integer-Arithmetik:

$$G = p_0 \wedge p_1 \wedge \neg p_2 \tag{2.2}$$

$$\begin{aligned} p_0 &= (x + 5) > 6 \\ p_1 &= (x + 5) < 9 \\ p_2 &= \neg(x = 2) \end{aligned}$$

Auch hier ist wieder eine Belegung β gesucht, so dass G unter β zu *wahr* ausgewertet werden kann. Da alle drei Prädikate durch eine $\wedge$-Verknüpfung miteinander verbunden sind, muss x so gewählt werden, dass sowohl p_0 als auch p_1 sowie p_2 *wahr* ergeben. Für $\beta : x \leftarrow 3$ ist dies der Fall. Für G existiert also eine gültige Belegung β und somit ist G auch erfüllbar.

Nach der obigen Darstellung des SMT-Problems wird deutlich, dass SMT im Grunde auch als Erweiterung von SAT um zusätzliche Theorien aufgefasst werden kann. In der Tat beruhen viele SMT-Solver auf effizienten SAT-Solvern bzw. auf effizienten Verfahren zur Lösung des SAT-Problems. In [9] wird die Kombination von SAT-Solvern und Theorie-Solvern genauer erläutert.

2.6 Firm und libFirm

Firm ist eine graphbasierte Zwischensprache, die Programme in die SSA-Form[1] darstellt. Entwickelt wurde Firm an der Universität Karlsruhe (TH) [16]. Firm stellt ein Programm als gerichteten Graphen dar, in dem Datenabhängigkeiten und Steuerfluss kombiniert sind. Die Knoten des Graphen repräsentieren den aus der jeweiligen Operation resultierenden Wert. Die Datenabhängigkeitskanten zeigen auf die zur Berechnung des Ergebnisses benötigten Argumente. Die Reihenfolge der Befehle innerhalb eines Grundblocks ist nicht explizit vorgegeben, sondern ergibt sich aus den Datenabhängigkeiten. So kann beispielsweise eine Multiplikation erst durchgeführt werden, wenn alle benötigten Argumente zur Verfügung stehen. Für Grundblöcke hingegen wird eine Reihenfolge teilweise durch den Steuerfluss festgelegt.

libFirm [13] ist die Implementierung von Firm in Form einer C-Bibliothek.

[1] Static Single Assignment Form

3 Verwandte Arbeiten

Ein wichtiger Teilaspekt der Arbeit eines heutigen Übersetzers bildet die Optimierung des von ihm generierten Codes, wie bereits in Kapitel 1 angeführt. Dies gilt sowohl für die Zwischen- als auch für die Zieldarstellung. Optimierungen transformieren die Darstellung eines Programms unter Beachtung dessen Semantik mit dem Ziel, eine Verbesserung dieser Darstellung bezüglich eines gewählten Kostenmodells zu erreichen. Es ist nicht sichergestellt, dass eine angewandte Optimierung bzw. die Menge der angewandten Optimierungen auch zu einem optimalen Ergebnis führen.

Massalin beschäftigt sich in [14] als einer der Ersten mit diesem Problem. Er setzt sich in dieser Arbeit mit der Frage auseinander, wie die optimale Codesequenz zu einer gegebenen Funktion aussieht und wie diese ermittelt werden kann. Als Kostenmaß legt Massalin die Anzahl der involvierten Instruktionen einer Codesequenz zugrunde. Die Idee seines Ansatzes basiert auf der Aufzählung aller Codesequenzen aufsteigender Länge und dem Vergleich jeder dieser Codesequenzen mit der ursprünglichen Codesequenz, hinsichtlich ihrer Semantik. Wird eine semantisch äquivalente Sequenz gefunden, folgt aus der Aufzählungsreihenfolge, dass es sich hierbei um eine kürzeste und somit optimale Sequenz, bezogen auf sein Kostenmodell handelt. Eine besondere Herausforderung stellt die Überprüfung der Äquivalenz zweier Codesequenzen dar. Massalin drückt hierzu eine Codesequenz als Menge von Termen aus, die aus booleschen Operationen und den Argumenten und Konstanten der Funktion bestehen und vergleicht diese anschließend mit den ebenso erzeugten Termen der ursprünglichen Codesequenz. Dieser Vergleich entspricht einem SAT-Problem. Da es sich bei diesem Vergleich um eine sehr Laufzeit-intensive Aufgabe handelt, verwendet Massalin zuvor einen weiteren Test. Dieser besteht darin, die beiden zu vergleichenden Funktionen bzw. Codesequenzen in ein ausführbares Programm zu übersetzen und mit einer Menge von Testdaten in Form von Eingabevektoren auszuführen. Anschließend werden die Ergebnisse verglichen. Unterscheiden sie sich, folgt daraus, dass auch die Semantik der Funktionen unterschiedlich sein muss. Eine weitere Reduktion der zu vergleichenden Codesequenzen erreicht Massalin durch den Ausschluss von Sequenzen, welche bereits als nicht-optimal bekannt sind. Dies ist möglich, da jede Codesequenz, die eine nicht-optimale Sequenz beinhaltet, durch eine kürzere Sequenz repräsentiert werden kann. Massalin gibt in seiner Arbeit einige interessante Beispiele von superoptimierten Programmen bzw. Codesequenzen an, denen ihre eigentliche Semantik nicht direkt anzusehen ist. Abbildung 3.1 zeigt als Beispiel die Signum-Funktion, einmal als C-Programm und einmal in ihrer Darstellung als superoptimiertes 8068-Assembler Programm mit lediglich 3 Instruktionen.

```
signum(x)
{
  if(x > 0)       return  1;    cwd         ;x in ax
  else if(x < 0)  return -1;    neg ax
  else            return  0;    adc dx,dx   ;signum(x) in dx
}
```

(a) C-Programm (b) 8086-Assembler Programm

Abbildung 3.1: Signum-Funktion – Als C-Programm und als superoptimiertes 8086-Assembler-Programm [14]

Die Idee von Massalin, wurde 2002 von Joshi, Nelson und Randall aufgenommen und floss in den von ihnen entwickelten Superoptimierer *Denali* [11] ein. Der Unterschied zwischen Denali und dem von Massalin verfolgten Ansatz besteht vorwiegend in der Art und Weise wie eine optimale Codesequenz zu einer gegebenen Funktion bestimmt werden kann. Massalins Ansatz zählt hierzu nacheinander Codesequenzen unabhängig von ihrer Semantik auf und bestimmt direkt im Anschluss, ob es sich um eine semantisch äquivalente Sequenz handelt. Der Ansatz bei Denali basiert hingegen darauf, zunächst mittels zuvor definierter Ersetzungsregeln, sog. Axiomen alle semantisch äquivalenten Berechnungsmöglichkeiten für jeden Ausdruck bzw. Teilausdruck einer Funktion zu bestimmen. Im Anschluss daran wird aus den diversen Möglichkeiten die optimale Sequenz ermittelt. Als Kostenmaß wird bei Denali nicht die Anzahl der Instruktionen zugrunde gelegt, sondern die tatsächlich benötigten Taktzyklen der involvierten Operationen. Da es bei Denali per Design nur semantisch äquivalente Transformationen der ursprünglichen Codesequenz gibt, ist die Überprüfung dieser Äquivalenz nicht notwendig. Dem entgegen ist hier das Problem der unterschiedlichen Kombinationsmöglichkeiten der einzelnen Berechnungsvarianten zu lösen. Dies geschieht schrittweise und basiert auf dem Beweis, dass kein Programm, bezogen auf die gewünschte Zielarchitektur, existiert welches die vorgegebene Funktion innerhalb von k Taktzyklen berechnet. Schlägt der Beweis fehl, existiert ein solches Programm, woraufhin alle Programme mit weniger als k Zyklen extrahiert werden und der Beweis für $k-1$ Zyklen von neuem beginnt. Dieser Vorgang wird solange fortgeführt, bis ein entsprechendes minimales k und damit ein optimales Programm bzw. eine optimale Codesequenz gefunden wird. Auch bei Denali kommt SAT als Beweisverfahren zum Einsatz.

Der Ansatz von Joshi, Nelson und Randall [11] besitzt bezüglich der Optimalität der ermittelten Programme diverse Einschränkungen, worauf auch in [11] hingewiesen wird. Zum einen ist es von der Menge der definierten Axiome abhängig, ob die optimale Darstellung gefunden wird. Nur wenn diese vollständig sind, ist gewährleistet, dass alle Berechnungsmöglichkeiten berücksichtigt werden. Bei dem Ansatz von Massalin gibt es eine solche Einschränkung nicht, da dieser per Design alle Berechnungsmöglichkeiten berücksichtigt. Zum anderen berücksichtigt Denali die Anzahl der für einen Befehl notwendigen

Taktzyklen. Diese müssen jedoch bekannt sein. Für Operationen ohne Speicherzugriff ist dies unproblematisch. Für Operationen mit Speicherzugriff ist es nahezu unmöglich diese genau vorherzusagen.

In [14] und [11] liegt der Schwerpunkt auf der Suche nach der optimalen Coderepräsentation zu einer gegebenen Funktion oder einem Programmteil. Bansal und Aiken führen diese Idee in [3] fort und nutzen das von Massalin eingeführte Prinzip der Superoptimierung zur automatischen Generierung von Ersetzungsregeln, die sie zur Peephole-Optimierung[1] verwenden. Diese so erzeugten Optimierungsregeln lassen sich anschließend speichern und für jeden neuen Übersetzungsvorgang wieder verwenden. Bansal und Aiken erreichen so eine Modularisierung, in der die teure Superoptimierungsphase ausgelagert werden kann. Der Übersetzer kann bei jedem Durchlauf auf die gefundenen Optimierungsregeln zugreifen und diese wie handgeschriebene Optimierungsregeln nutzen. Um die eigentlichen Optimierungsregeln zu generieren, gehen Bansal und Aiken folgendermaßen vor: Im ersten Schritt werden aus einer Menge ausgewählter Programme Codesequenzen extrahiert, für die eine optimale Ersetzung und damit eine Optimierung gefunden werden soll. Die Idee hierbei ist, nur solche Sequenzen zu berücksichtigen, die üblicherweise von Übersetzern erzeugt werden und solche zu vernachlässigen, die gar nicht oder nur sehr selten in Programmen vorkommen. Da es sich allerdings häufig um dieselben Codesequenzen mit lediglich unterschiedlicher Registerbelegung handelt, werden die ermittelten Codesequenzen in einem weiteren Schritt einer Normalisierung, von Bansal und Aiken *Canonicalization* genannt, unterzogen. Diese führt eine Umbenennung der Register durch. Jedes neu genutzte Register erhält dabei einen Namen in aufsteigender Reihenfolge, beginnend mit $r_0, r_1, \ldots, r_n$. Bansal und Aiken zeigen, dass dieses Verfahren die Anzahl der zu untersuchenden Codesequenzen deutlich reduziert. In einem zweiten Schritt werden alle möglichen Codesequenzen bis zu einer festen Größe aufgezählt. Diese werden anschließend mit den zuvor extrahierten Codesequenzen hinsichtlich ihrer Semantik verglichen und bei Übereinstimmung eine neue Optimierungsregel angelegt. Auch hier sorgt die Reihenfolge der Aufzählung dafür, dass immer die günstigste Sequenz einer Äquivalenzklasse[2] zuerst berücksichtigt wird. Der Vergleich erfolgt analog zu dem Vorgehen von Massalin. Zuerst werden die zu vergleichenden Codesequenzen übersetzt, ausgeführt und anschließend werden die Ergebnisse verglichen. Tritt keine Differenz der Ergebnisse auf, wird aus beiden Codesequenzen eine Äquivalenzrelation gebildet und diese in Form eines SAT-Problems auf ihre Gültigkeit hin geprüft.

Allen drei vorgestellten Arbeiten gemein ist, dass sie auf Assembler-Ebene arbeiten. Die Arbeit von Massalin sowie Denali zielen darauf ab, eine optimale Codesequenz zu einer gegebenen Funktion während des Übersetzungsvorgangs zu ermitteln, Bansal und Aiken hingegen erzeugen, unabhängig vom eigentlichen Übersetzungsvorgang, Regeln zur Peephole-Optimierung. Der Ansatz dieser Diplomarbeit orientiert sich an dem Verfahren

[1] Guckloch-Optimierung: Ein verschiebbares Fenster mit wenigen Befehlen wird untersucht und wenn möglich optimiert [1]

[2] Codesequenzen mit gleicher Semantik bilden eine Äquivalenzklasse

von Bansal und Aiken. In einem separaten Prozess werden Optimierungsregeln generiert, um sie für die Verwendung in einem Übersetzer nutzbar zu machen. Im Unterschied zu Bansal und Aiken setzt das hier vorgestellte Verfahren allerdings nicht auf Assembler-Ebene an, sondern verfolgt das Ziel, möglichst allgemeingültige Optimierungsregeln zu generieren, die unabhängig von der Zielarchitektur sind und bereits zur Optimierung der Zwischendarstellung, im Besonderen Firm, genutzt werden können.

Ein weitgehend unbeachteter Aspekt im Zusammenhang mit Optimierungsregeln ist der Umgang mit Konstanten. Bansal und Aiken berücksichtigen diese zwar, beschränken sich allerdings auf die beiden Konstanten 0 und 1 sowie zwei symbolische Konstanten c_0 und c_1, mit denen eingeschränkt von konkreten Konstanten abstrahiert werden kann. Dennoch sind Konstanten bei der Betrachtung bzw. Suche von Optimierungsregeln interessant. Einfache Regeln, wie sie beispielsweise in 3.1 und 3.2 dargestellt sind, werden zwar meist berücksichtigt, komplexere Regeln wie in 3.3 und 3.4 gezeigt, werden jedoch i. d. R. ignoriert.

$$(x * 2) \quad \rightarrow \quad x + x \tag{3.1}$$

$$x \;\&\; 0 \quad \rightarrow \quad 0 \tag{3.2}$$

$$(x + x) \mid \text{-2} \quad \rightarrow \quad \text{-2} \tag{3.3}$$

$$(x \mid 8) + \text{-8} \quad \rightarrow \quad x \;\&\; \text{-9} \tag{3.4}$$

Ein Ziel dieser Arbeit ist es, alle Konstanten mit in den Generierungsvorgang einzubeziehen. Dies stellt sicher, dass alle möglichen Muster berücksichtigt werden und so alle Optimierungsregeln gefunden werden können. Aufgrund ihrer Vielzahl und den sich hieraus ergebenen Kombinationsmöglichkeiten im Bezug auf die Menge der möglichen Muster, stellen Konstanten eine besondere Herausforderung an den Generierungsvorgang dar. In dieser Arbeit wird gezeigt, dass sich ein vollständiger Generierungsvorgang dennoch durchführen lässt.

4 Implementierung

Die in Kapitel 3 erwähnten Arbeiten betrachten alle Ausdrücke in Form von Codesequenzen auf Assembler-Ebene. Dem entgegen ist das in dieser Arbeit vorgestellte Verfahren, im Folgenden als OPTGEN bezeichnet, auf eine maschinenunabhängige Zwischendarstellung ausgerichtet, weswegen in Anlehnung an FIRM, Ausdrücke als Muster in Graphform betrachtet werden.

Dieses Kapitel teilt sich in zwei Themenbereiche. Der erste Teil befasst sich mit dem allgemeinen Aufbau und Ablauf des Verfahrens, während sich der zweite Teil speziell mit der Thematik von Konstanten befasst.

Zunächst noch einige Bemerkungen, die sich auf die in diesem Kapitel angeführten Beispiele beziehen:

Bemerkung 1 (Reihenfolge der Operanden einer Operation)**.** *Bei den in den Abbildungen dargestellten Mustern wird auf eine explizite Auszeichnung des Kanten-Index (pos-Abbildung), also der Angabe, ob es sich um den ersten oder zweiten Operanden handelt, der Übersicht wegen verzichtet. Es wird angenommen, dass die linke Kante dem linken bzw. ersten Operanden einer Operation zugeordnet ist und entsprechend die rechte Kante dem rechten bzw. zweiten Operanden.*

Bemerkung 2 (Datentypbreite)**.** *Für alle Beispiele in diesem, sowie in Kapitel 5 gilt, sofern nichts anderes angegeben ist, dass die Datentypbreite auf 8 Bit festgelegt ist. Der Wertebereich von* $-128, \ldots, 127$ *bzw.* $0, \ldots 255$ *wird als* $\mathbb{Z}_{8Bit}$ *angegeben.*

Bemerkung 3 (Semantik von Operationen)**.** *Die Semantik aller in den Beispielen verwendeter Operationen entspricht der Semantik der Operationen in lib*FIRM*. Es handelt sich dabei um eine Modulo-Semantik, ähnlich zu den entsprechenden Hardware-Befehlen auf Assembler-Ebene. Dies ist vor allem im Zusammenhang mit Überläufen zu beachten.*

4.1 Ansatz

Zunächst soll ein grober Überblick über die Arbeitsweise des hier vorgestellten Verfahrens OPTGEN zur Generierung lokaler Optimierungsregeln gegeben und die einzelnen Komponenten kurz erläutert werden.

Das Grundprinzip dieses Ansatzes orientiert sich, wie bei Bansal und Aiken [3], an der Idee von Massalin [14]. Es besteht darin, durch Aufzählung aller Muster und dem Vergleich ihrer Semantik, jeweils ein günstigstes und damit optimales Muster aus der Menge der semantisch äquivalenten Muster zu ermitteln und aus den Äquivalenzbeziehungen zwischen den Mustern einer solchen Menge, Ersetzungsregeln abzuleiten.

Die Idee ist nun, die einzelnen Muster nacheinander aufzuzählen und dabei deren Kosten, stufenweise, bis zu einer definierten Kostenobergrenze zu steigern. Hierdurch ergibt sich eine Sortierung der Muster nach ihrem Kostenwert $cost_M$ und es wird gewährleistet, dass immer das günstigste Muster zuerst behandelt wird. Nach der Erzeugung eines neuen Musters wird dieses mit den bereits erzeugten, optimalen Mustern hinsichtlich der Semantik verglichen. Besteht eine Äquivalenz, kann daraus eine Optimierungsregel oder eine Transformationsregel abgeleitet werden.

Aus dieser Grundidee ergibt sich eine logische Strukturierung in die Hauptkomponenten:

Mustererzeugung Erzeugt schrittweise, aufbauend auf bereits vorhandenen Mustern neue Muster. Die Kosten der erzeugten Muster nehmen dabei stufenweise, bis zu einer definierten Kostenobergrenze zu. Jedes Muster wird zur Generierung neuer Muster mit höheren Kosten der Menge der bereits erzeugten Muster hinzugefügt.

Musteranalyse Prüft jedes neu erzeugte Muster zunächst dahingehend, ob eine bereits erzeugte Regel anwendbar ist. Ist dies der Fall, wird das Muster von der Musteranalyse nicht weiter berücksichtigt. Andernfalls wird geprüft, ob bereits ein anderes, semantisch äquivalentes, optimales Muster vorhanden ist. Dies ist Aufgabe des Vorabtests und des Erfüllbarkeitstests. Existiert ein solches Muster, wird eine neue Regel erzeugt. Wird kein semantisch äquivalentes, optimales Muster gefunden, so wird das Muster zur Menge der optimalen Muster M_{Opt} hinzugefügt[1].

In Abbildung 4.1 ist der Ablauf in einem Übersichtsdiagramm dargestellt. Eine ausführliche Beschreibung der einzelnen Komponenten, sowie diverser Erweiterungen und Einschränkungen wird in den folgenden Abschnitten gegeben.

4.2 Mustererzeugung

4.2.1 Kostenmodell und Mustergröße

Ein Kostenmodell wird benötigt, um die einzelnen Muster untereinander vergleichen und jeweils ein optimales Muster auswählen zu können. Massalin [14] verwendet hierzu die Anzahl der Instruktionen, Denali [11] vergleicht die zu erwartenden Taktzyklen einer

[1] Aufgrund der Aufzählungsreihenfolge ist sichergestellt, dass kein günstigeres Muster mit gleicher Semantik existiert

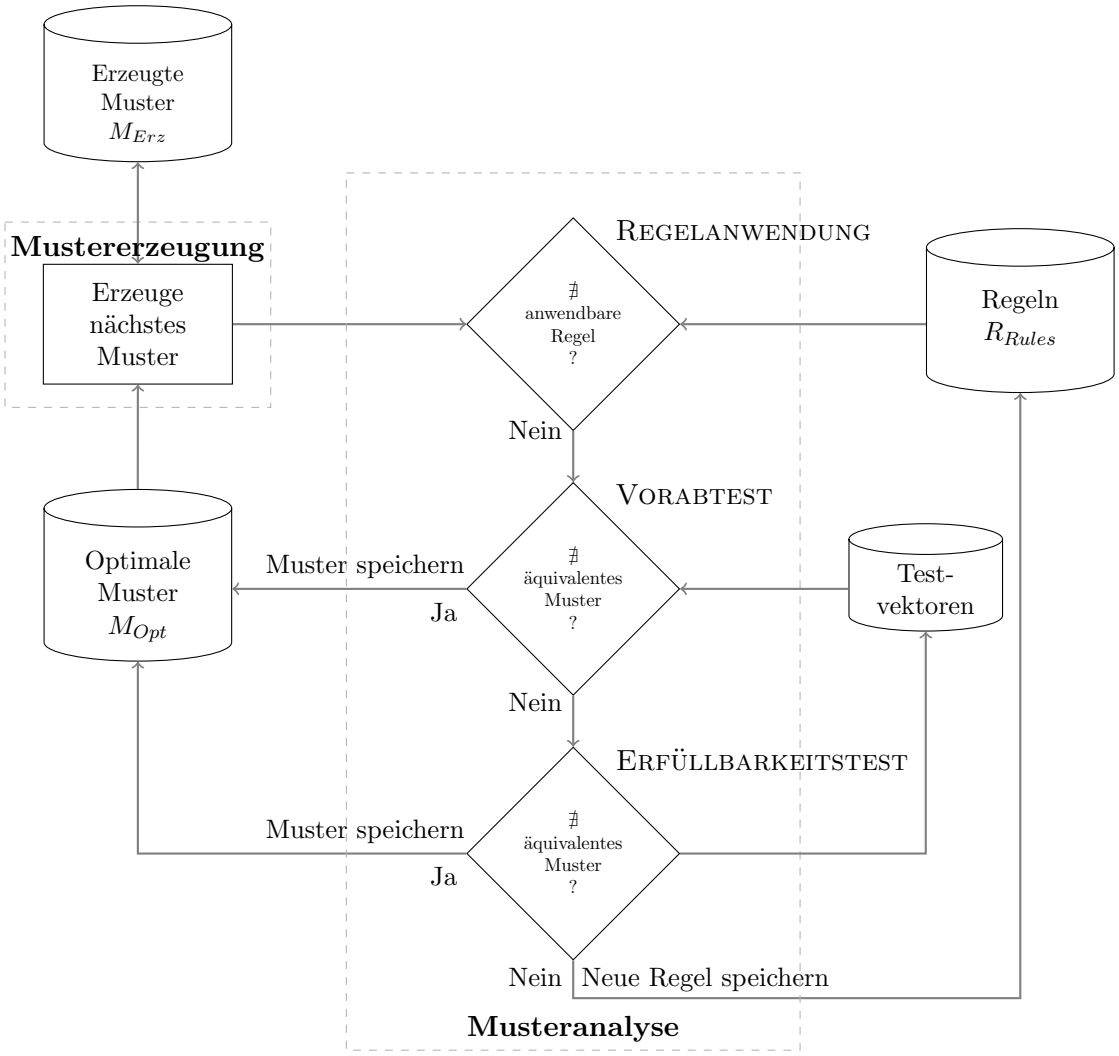


Abbildung 4.1: Übersichtsdiagramm – Grundlegender Ablauf des vorgestellten Verfahrens Optgen zur Erzeugung lokaler Optimierungsregeln.

Codesequenz, ebenso wie Bansal und Aiken [3]. Im Unterschied dazu, ist das hier vorgestellte Verfahren auf eine maschinenunabhängige Zwischensprache ausgerichtet, weswegen für eine Operation keine explizite Anzahl der notwendigen Taktzyklen angeben werden kann. Einfach den Ansatz von Massalin zu übernehmen und die Anzahl der Operationen als Kostenmaß zu verwenden erscheint allerdings auch etwas restriktiv, da gewisse Beziehungen bezüglich der benötigten Taktzyklen eines Befehls eigentlich für allen Architekturen bestehen (z. B. zwischen Addition und Multiplikation). Aus diesem Grund wird, als Kompromiss zwischen beiden Kostenmodellen, jeder Operation ein fester Kostenwert zugeordnet, so dass die Verhältnismäßigkeit im Sinne von *teurer als* oder *günstiger als*, wie sie beispielsweise bei einer IA32-Architektur besteht, gewahrt bleibt. Die Addition erhält deshalb einen niedrigeren Wert als die Multiplikation. Eine Aufstellung der entsprechenden Kosten findet sich in Tabelle 4.1, die auch in Kapitel 5 als Kostenmodell dient. Eine andere Kostenzuteilung ist allerdings genauso möglich.

Knotentyp:	neg	add	sub	mul	not	or	and	xor
Kosten:	1	1	1	2	1	1	1	1

Tabelle 4.1: Übersicht über Operationen und deren Kosten. Bei den Operationen not, or, and, xor handelt es sich nicht um logische Operationen, sondern um Bit-Operationen.

Prinzipiell können Muster beliebig groß werden wodurch sich beliebig viele Kombinationsmöglichkeiten ergeben. Daher ist eine Beschränkung der Mustergröße notwendig. Die einfachste Möglichkeit wäre, die Anzahl der Operationen oder die Anzahl der Knoten als Maß heranzuziehen. Allerdings wird dann die Menge der auffindbaren Optimierungsregeln eingeschränkt, indem Muster nicht mehr berücksichtigt werden, die zwar mehr Operationen oder Knoten, aber einen niedrigeren Kostenwert besitzen. Aus diesem Grund wird die Mustergröße anhand des Kostenmaßes beschränkt. Dies stellt sicher, dass alle Muster bis zu einer definierten Kostenobergrenze auch erzeugt werden.

4.2.2 Erzeugen neuer Muster

Der erste Schritt zur Generierung einer Regel besteht in der Erzeugung eines neuen Musters. Hierzu wird eine Operation mit einem, bzw. zwei bereits bekannten Mustern aus der Menge M_{Erz} zu einem neuen Muster kombiniert, das im Anschluss von der Musteranalyse untersucht wird.

Da zu Beginn des Generierungsvorgang noch keine Muster bekannt sind, müssen zunächst die Basismuster erzeugt werden. Ein Basismuster besteht aus einem einzelnen Knoten und repräsentiert einen Operanden, genauer eine Variable oder eine Konstante. Da Variablen und Konstanten einen Wert repräsentieren, aber keine Veränderung dieses Wertes darstellen und somit auch keine Rechenzeit erfordern, werden ihnen die Kosten 0 zugeordnet. Auf Assembler-Ebene entspricht diese Annahme dem Fall, dass sich die er-

forderlichen Werte bereits in den dafür vorgesehenen Registern befinden und somit kein separater Ladebefehl oder ähnliches erforderlich ist. Alternativ wäre es auch denkbar, den Ladebefehl mit dem der Wert einer Konstanten in ein Register geladen werden muss zu berücksichtigen. In diesem Fall ist auch für eine Konstante Rechenzeit erforderlich, weswegen ihr auch ein höherer Kostenwert als 0 zugeordnet werden könnte. Da ein solcher Ladebefehl allerdings nicht bei jeder Operation, die eine Konstante verwendet erforderlich ist, sondern das Laden einer Konstante auch implizit bei der Ausführung einer Operation erfolgen kann, wird diese Möglichkeit nicht weiter berücksichtigt. Einige Beispiele solcher Basismuster zeigt Abbildung 4.2. Entsprechend der Definition der Kostenfunktion $cost_M$ für Muster ergibt sich für die Basismuster ein Kostenwert von 0.

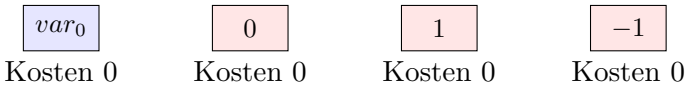


Abbildung 4.2: Muster ohne Operationen mit Kosten 0 (Basismuster).

Im nächsten Iterationsschritt werden Muster m mit einem Kostenwert $cost_M(m) = 1$ erzeugt. Hierzu wird eine Operation mit einem oder zwei Basismustern[1], je nachdem ob es sich um eine unäre oder binäre Operation handelt, kombiniert und so ein neues Muster erzeugt. Die Kosten des neuen Musters ergeben sich aus den Kosten der verwendeten Operation und den Kosten der involvierten Basismuster. In diesem Iterationsschritt können daher nur Operationen mit einem Kostenwert von 1 verwendet werden. Der beschriebene Vorgang wird fortgeführt, bis jede mögliche Kombination aus Operationen und Basismustern behandelt wurde, die für die aktuelle Kostenstufe zulässig ist. Der Fall, dass dasselbe Basismuster beide Operanden einer binären Operation bildet wird ebenfalls erlaubt. Dies ist für Ausdrücke wie beispielsweise $x + x$, also eine Addition einer Variablen x mit sich selbst nötig. Entsprechende Beispiele zu Mustern mit Kosten 1 werden in Abbildung 4.3 gezeigt.

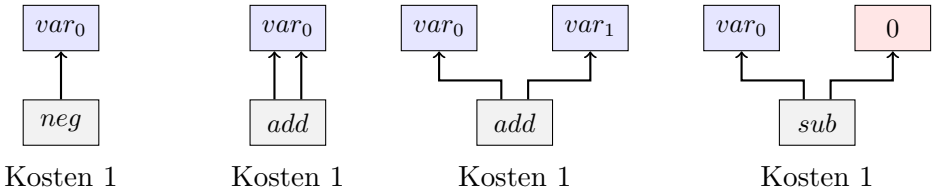


Abbildung 4.3: Abbildung einiger Muster der Kostenstufe 1.

Analog wird dieser Vorgang wiederholt und die Kostenstufe bei jeder Iteration um eins erhöht. So werden sukzessive alle möglichen Muster bis zu einer definierten Kostenobergrenze[2] gebildet. Zur Veranschaulichung zeigt Abbildung 4.4 zwei Beispiele komplexerer Muster mit höherem Kostenwert.

[1] sofern diese von der Musteranalyse als optimal erkannt wurden
[2] die Kostenobergrenze ist als Parameter aufzufassen und kann für jeden Generierungsvorgang variieren

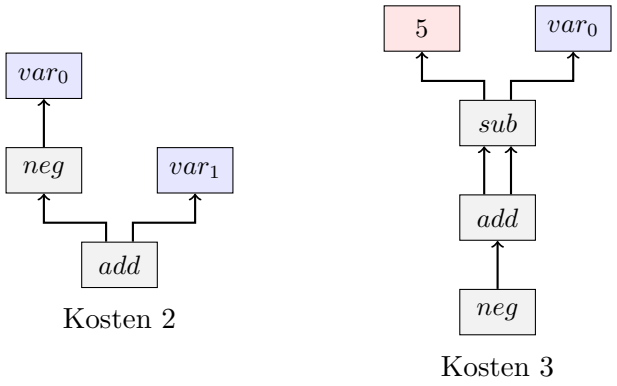


Abbildung 4.4: Beispiel zu komplexeren Mustern höherer Kostenstufen.

Reihenfolge der erzeugten Muster

Für die Geschwindigkeit des Generierungsvorgangs ist es von Vorteil, wenn allgemeinere Muster vor spezielleren Mustern erzeugt werden. Ein Muster m ist dann allgemeiner als ein Muster m', wenn sich m und m' nur dadurch unterscheiden, dass eine oder mehrere Konstanten in m' durch eine Variable in m ersetzt sind, wie dies beispielsweise in Abbildung 4.5 der Fall ist. Dadurch wird erreicht, dass auch allgemeinere Regeln zuerst erzeugt werden und so viele speziellere Muster gar nicht erst betrachtet werden müssen. Aus diesem Grund werden zunächst Variablen als Operanden verwendet, bevor im Anschluss Konstanten die Operanden bilden (vgl. Abbildung 4.5). Existiert bereits eine Regel für ein Muster mit Variablen, müssen alle spezielleren Muster nicht mehr von der Musteranalyse berücksichtigt werden. Hierauf wird in Abschnitt 4.3.1 noch einmal eingegangen.

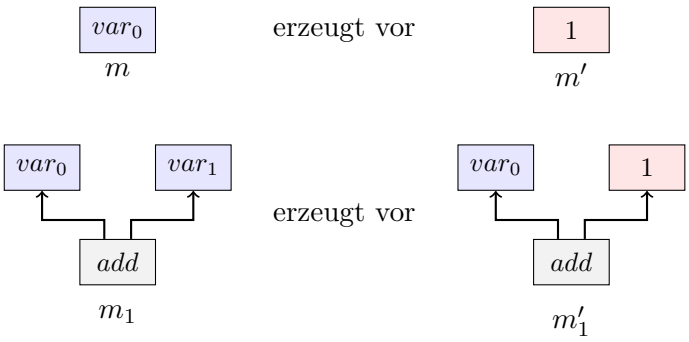


Abbildung 4.5: Allgemeinere Muster (links) werden vor spezielleren Mustern (rechts) erzeugt.

4.2.3 Normalisierung

Die Menge der möglichen Muster nimmt aufgrund der vielen Kombinationsmöglichkeiten von Operationen und Operanden schnell zu. Es sind jedoch nicht alle Muster für den Generierungsvorgang nötig. Durch Vereinheitlichung bzw. Normalisierung können bereits bei der Mustererzeugung überflüssige Muster ausgeschlossen werden.

Normalisierung bezüglich mehrerer Variablen

Viele Muster unterscheiden sich lediglich in der Anordnung ihrer Variablen, sind aber ansonsten bezüglich ihres Aufbaus identisch. Für n Variablen lassen sich $n!$ fast identische Muster erzeugen, die sich nur durch eine andere Reihenfolge ihrer Variablen bei einer Postorder-Traversierung des Mustergraphen unterscheiden. Für eine Regel ist die konkrete Benennung der Variablen jedoch unerheblich. Entscheidend ist nur, dass die relative Position jeder einzelnen Variablen erhalten bleibt. Dies soll durch die beiden in Abbildung 4.6 dargestellten Regeln verdeutlicht werden. Obwohl sich die Benennung der Variablen (Knoten 1 und Knoten 2) in den Regeln r und r' unterscheidet, drücken beide Regeln dieselbe Transformationsvorschrift aus.

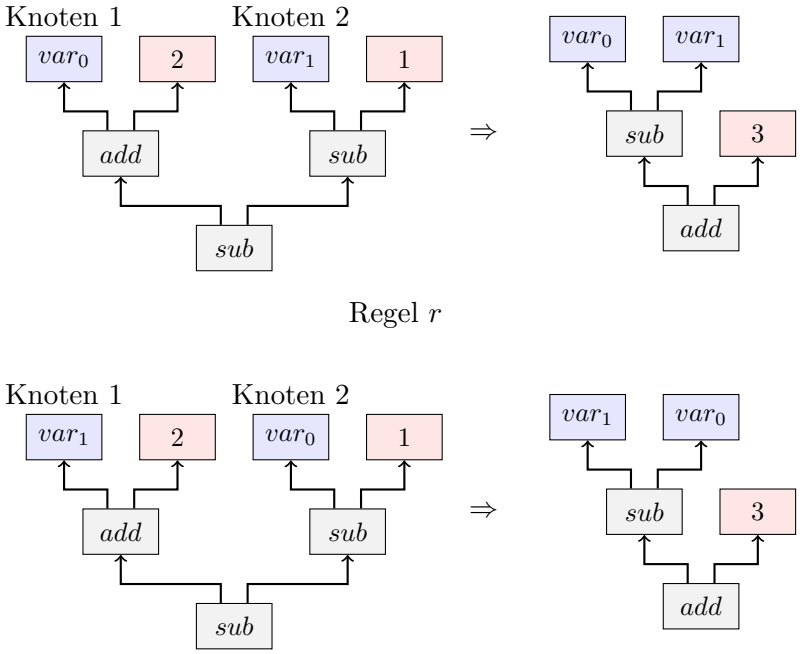


Abbildung 4.6: Viele Regeln unterscheiden sich lediglich in der Benennung ihrer Variablen, beschreiben aber dieselbe Transformationsvorschrift.

Um die Mustererzeugung zu entlasten, werden analog zu dem von Bansal und Aiken in [3] vorgeschlagenen Ansatz[1] nur Muster aufgezählt, deren Variablenanordnung, genauer die Nummerierung durch die idx-Abbildung, mit der Reihenfolge bei einer Postorder-Traversierung des Mustergraphen übereinstimmt. In der oben genannten Abbildung entspricht dies den beiden Mustern der Regel r.

Auch für die im späteren Abschnitt 4.3.1 vorgestellte Erkennung von nicht-optimalen Mustern durch das Ausnutzen bereits bekannter Regeln bietet dieses Vorgehen Vorteile. Existiert zu einem Muster m ein semantisch äquivalentes Muster und damit eine Ersetzungsregel, so ist diese Regel zwangsläufig auf alle Muster anwendbar, die sich lediglich in der Anordnung der Variablen von m unterscheiden. Werden solche Muster bereits bei der Aufzählung vermieden, kann die notwendige Rechenzeit für die Überprüfung, ob eine Regel anwendbar ist, eingespart werden.

Für die Suche nach einem semantisch äquivalenten Muster ist die Reihenfolge der Variablen jedoch nicht immer unerheblich, wie das Beispiel aus Abbildung 4.7 zeigt. Bei m und l handelt es sich um zwei Muster in der oben genannten Normalform, die in dieser Form nicht semantisch äquivalent sind. Durch eine Umbenennung bzw. Permutation der

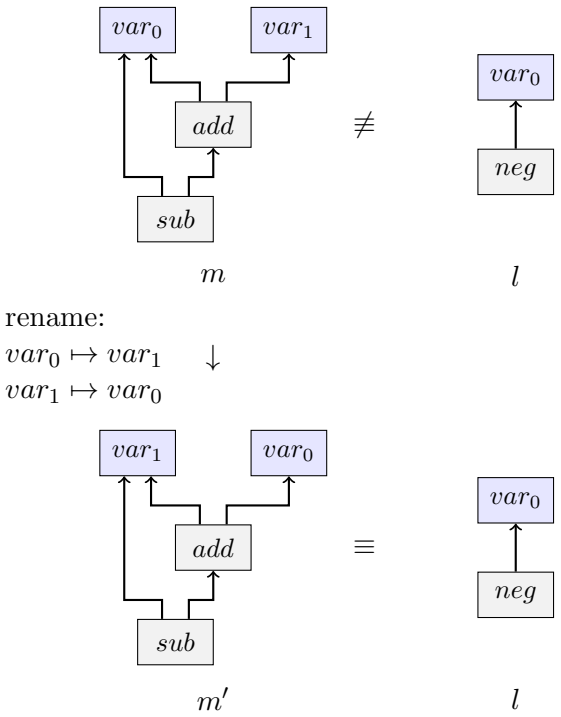


Abbildung 4.7: Manche Muster sind erst nach einer Umbenennung bzw. Permutation ihrer Variablen semantisch äquivalent.

[1] als Canonicalization bezeichnet

Variablen von m, wie in der Abbildung gezeigt, kann ein semantisch äquivalentes Muster zu l in Form von m' gefunden und damit eine neue Regel erzeugt werden.

Um zu garantieren, dass alle Ersetzungsregeln gefunden werden können, müssen neben dem eigentlichen Muster m auch alle Varianten von m berücksichtigt werden, die eine Permutation der Variablen von m darstellen. Zu diesem Zweck wird zunächst m durch den Vorabtest (Abschnitt 4.3.2) und den Erfüllbarkeitstest (Abschnitt 4.3.3) untersucht und anschließend nacheinander jede weitere Variante m', bis ein äquivalentes Muster gefunden oder alle Varianten untersucht wurden. In Listing 4.1 wird dieser Vorgang als Codeausschnitt dargestellt. Die vollständige Prozedur zeigt Listing 4.7 in Abschnitt 4.4.

Listing 4.1 Suche nach einem semantisch äquivalenten Muster

```
1: ...
2: p ← m                                          ▷ Beginne Suche mit m
3: m' ←⊥
4: while p ≠⊥ ∧ m' =⊥ do ▷ Solange eine Permutation existiert und kein semantisch
   äquivalentes Muster gefunden wurde
5:     M_Cand ← VORABTEST(m, M_Opt)                ▷ Kandidaten ermitteln
6:     m' ← ERFÜLLBARKEITSTEST(m, M_Cand)          ▷ äquivalentes Muster suchen
7:     if m' =⊥ then
8:         p ← GETNEXTPERMUTATION(p)               ▷ nächste Permutation erzeugen
9:     end if
10: end while
11: ...
```

Normalisierung bezüglich unterschiedlicher Operandentypen

Viele Muster sind aufgrund algebraischer Eigenschaften wie der Kommutativität semantisch äquivalent, unterscheiden sich aber in ihrer syntaktischen Struktur wie Abbildung 4.8 zeigt.

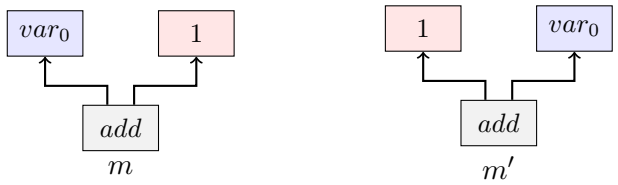


Abbildung 4.8: Abbildung zweier semantisch äquivalenter aber syntaktisch verschiedener Muster.

Die Anzahl solcher Muster kann reduziert werden. Hierzu wird eine Normalform eingeführt, die eine Anordnung der Operanden für kommutative Operationen festlegt. Es werden nur noch solche Muster betrachtet, die dieser Normalform genügen. Das Tauschen von Operanden bei kommutativen Operationen hat keinen Einfluss, sowohl auf die Semantik der einzelnen Operationen als auch auf die Semantik des gesamten Musters. Durch die Normalform bzw. die Normalisierung werden also keine semantisch unterschiedlichen Muster ausgeschlossen.

Praktisch wird mit dieser Normalisierung vor allem erreicht, dass sich Konstanten bei kommutativen Operationen immer auf der rechten Seite befinden, oder, falls es sich bei beiden Operanden um eine Konstante handelt, die Konstanten mit dem größeren Wert immer rechts steht. Variablen stehen, sofern es sich bei dem zweiten Operanden nicht um eine Konstante handelt ebenfalls immer rechts.

Die Definition der Normalform erfordert zunächst die Festlegung einer Totalordnung der in Definition 6 beschriebenen Menge T der verfügbaren Typen. Zu diesem Zweck sei eine Abbildung $id : T \to \mathbb{N}_0$ gegeben, die jedem Typ $t \in T$ eine eindeutige Identifikationsnummer (ID) zuordnet. Für den Typ der Variablen soll gelten, dass dessen Identifikationsnummer größer als die größte Identifikationsnummer aller Operationen, aber kleiner als die kleinste Identifikationsnummer aller Konstanten ist. Für Konstanten soll ferner gelten, dass deren Identifikationsnummer größer als die Identifikationsnummer von Variablen ist und dass die Identifikationsnummer mit abnehmendem Konstantenwert zunimmt. Ein Beispiel einer Ordnung auf der Menge der Typen, die den genannten Anforderungen genügt, zeigt Abbildung 4.9.

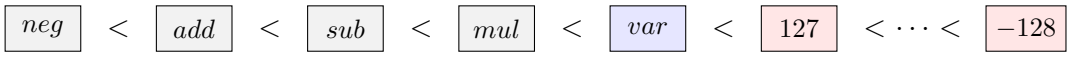


Abbildung 4.9: Abbildung einer gewählten Ordnung von Knotentypen.

Jetzt kann die Normalform wie folgt definiert werden:

Definition 13 (Normalform)**.** *Sei $id : T \to \mathbb{N}_0$ die oben genannte Abbildung. Ein Muster m mit der Knotenmenge V befindet sich dann in Normalform, wenn für jede kommutative Operation $o \in V$ und deren Operanden $o_{left}, o_{right} \in V$ gilt:*

$$id(type(o_{left})) < id(type(o_{right}))$$

Hier zeigt sich, warum die Reihenfolge von Variablen und Konstanten bezüglich der definierten Totalordnung wichtig ist. Da Konstanten immer die größere Identifikationsnummer gegenüber allen anderen Typen besitzen, befindet sich ein Muster nur dann in Normalform, wenn die Konstante dem rechten Operanden entspricht. Analog gilt dies auch für Variablen.

4.2.4 Reduktion durch Common Subexpression Elimination

Die *Common Subexpression Elimination* (CSE) ist eine klassische Optimierung, mit der sich mehrfache Berechnungen desselben Wertes vermeiden lassen. Hieraus ergibt sich eine weitere Möglichkeit, um die Anzahl der Muster zu reduzieren.

Generell existiert für jedes Muster, auf das die CSE anwendbar ist, eine Optimierungsregel, bestehend aus dem Muster vor der Anwendung der CSE und dem Muster nach dem Anwenden der CSE (siehe Abbildung 4.10).

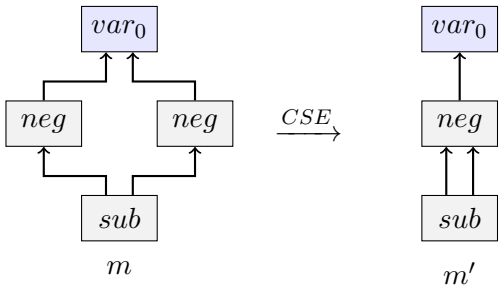


Abbildung 4.10: Muster vor und nach Anwendung der CSE.

Da aufgrund des Designs dieses Verfahrens jede Optimierungsregel mittels Erfüllbarkeitstest auf ihre Korrektheit hin bewiesen wird, dies aber sehr teuer[1] ist, werden Muster für die die CSE anwendbar ist gleich bei der Mustererzeugung verworfen. Die entsprechenden Muster, die sich durch Anwendung der CSE ergeben würden, existieren zu diesem Zeitpunkt aufgrund ihrer geringeren Kosten bereits. Auf die entsprechenden Optimierungsregeln wird durch das Ignorieren der zuvor genannten Muster verzichtet. Dies beschleunigt die Überprüfung von Mustern auf die Anwendbarkeit einer Regel, da die Menge der verfügbaren Regeln ebenfalls kleiner bleibt. Prinzipiell ist der hier vorgestellte Ansatz aber in der Lage solche Optimierungsregeln zu finden und zu berücksichtigen.

4.3 Musteranalyse

Anhand der Musteranalyse wird geprüft, ob es sich bei einem neu erzeugten Muster um ein optimales Muster handelt, oder ob bereits ein semantisch äquivalentes, optimales Muster existiert. Handelt es sich um ein optimales Muster, wird dieses der Menge der optimalen Muster hinzugefügt und steht für den Äquivalenztest und damit als mögliche rechte Seite einer Regel zur Verfügung. Existiert ein äquivalentes Muster, wird eine neue Optimierungs- oder Transformationsregel angelegt. Die Überprüfung erfolgt in drei Phasen, die im Folgenden erläutert werden.

[1] bezüglich der erforderlichen Rechenzeit

4.3.1 Erkennen von nicht-optimalen Mustern durch anwendbare Ersetzungsregeln

Die benötigte Zeit eines Generierungsvorgangs nimmt sowohl mit jeder Kostenstufe, als auch mit jeder neuen Operation zu. Der Grund dafür liegt in den vielen Kombinationsmöglichkeiten der Operationen und Operanden und der daraus resultierenden Zunahme möglicher Muster, die zu untersuchen sind. Um Muster, die nicht optimal sind, möglichst frühzeitig ausschließen zu können, werden diese zunächst dahingehend untersucht, ob eine bereits bekannte Ersetzungsregel anwendbar ist. Ist dies der Fall, handelt es sich bei dem untersuchten Muster nicht um ein optimales Muster[1] und braucht von der Musteranalyse nicht weiter berücksichtigt zu werden.

Anwendbarkeit von Regeln

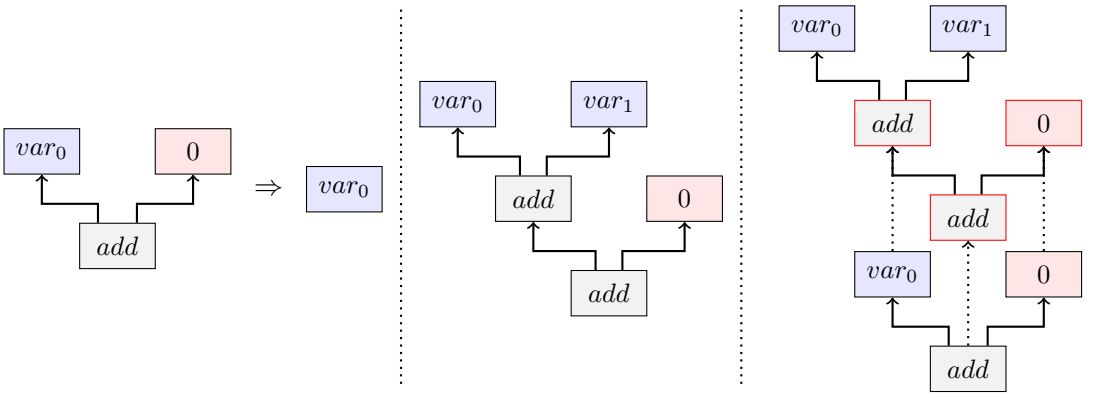
Um nicht-optimale Muster durch Regeln ausschließen zu können, ist zunächst zu klären, wann eine Regel angewendet werden kann. Intuitiv ist dies dann der Fall, wenn in einem Muster ein Teilmuster existiert, welches mit dem Muster der linken Seite einer Regel übereinstimmt. Formal entspricht dies dem Problem der Teilgraphisomorphie, worauf hier allerdings nicht näher eingegangen wird. Nur auf eine Besonderheit im Zusammenhang mit dieser Arbeit soll hier hingewiesen werden. Sei hierzu zunächst die Regel r und das Muster m, wie in Abbildung 4.11 (a), (b) dargestellt, gegeben. Intuitiv ist klar, dass sich m vereinfachen lässt indem r angewendet wird. Allerdings stimmt zunächst kein Teilgraph von m mit der linken Seite von r überein. Um dennoch die Anwendung von r auf m zu zu ermöglichen, muss die Bedingung, dass ein Teilmuster mit der linken Seite einer Regel übereinstimmen muss, entsprechend angepasst werden. Die Regel in der Abbildung wird dann anwendbar, wenn für den Knoten var_0 aus der Regel die Äquivalenz zu einem Knoten aus dem Muster m nicht gefordert wird. Dies wird in Abbildung 4.11 (c) dargestellt.

Allgemein wird daher erlaubt, dass für eine Variable aus einer Regel die Äquivalenz zu einem entsprechenden Knoten aus dem zu überprüfenden Muster hinsichtlich dessen Typs nicht bestehen muss. Ein Variable kann also alle anderen Knotentypen überdecken.

Einschränkung bei der Anwendbarkeit von Regeln

Bei der Anwendung von Regeln besteht eine Einschränkung, auf die im Folgenden anhand von Abbildung 4.12 eingegangen wird. Angenommen es existieren zwei Typen von Operationen op_1 und op_2 mit den Kosten $cost_T(op_1) = 1$ und $cost_T(op_2) = 2$. Weiter wird angenommen, es existiert eine Regel r, wie in der Abbildung dargestellt, die drei

[1] Bei Transformationsregeln kann das Muster zwar optimal sein, es handelt sich allerdings nicht um das erste Muster aus der Menge der Muster mit äquivalenter Semantik



(a) Regel r (b) Muster m (c) Anwendung von r auf m

Abbildung 4.11: Anwendung einer Regel auf ein Muster. Variablen einer Regel können Knoten mit anderem Typ überdecken.

hintereinander ausgeführte Operationen op_1 auf eine Operation op_2 abbildet. Die Gesamtkosten reduzieren sich hierdurch um 1. Wird r auf das in der Abbildung gezeigte Muster m angewendet, so reduzieren sich die Kosten nicht, obwohl es sich bei r um eine Optimierungsregel handelt. Vielmehr steigen die Kosten von vormals 4 auf 5, wie das Ergebnis m' zeigt. Der Grund besteht darin, dass eine Kante existiert, die ein echtes Teilmuster der Regel zum Ziel hat. In der Abbildung entspricht dies der Kante e_1 im Muster m. Aufgrund dieser Kante müssen die zwei folgenden op_1-Knoten erhalten bleiben. Es wird nur einer der drei op_1 Knoten entfernt, wodurch sich die Kosten nur um 1 verringern, es kommt aber ein neuer Knoten mit Kosten 2 hinzu. Somit ist das Endergebnis sogar teurer.

Um dieses Problem zu vermeiden, muss die Anwendbarkeit von Regeln eingeschränkt werden. Die Voraussetzung, dass ein Muster durch Anwendung einer Ersetzungsregel teurer werden kann ist, dass es zusätzliche Kanten (Datenabhängigkeiten, Referenzen) zu Knoten gibt, die von der Regel ersetzt werden. Die Referenz verhindert, dass diese Knoten aus dem Muster entfernt werden, da von ihnen noch andere Knoten bzw. Operationen abhängig sind (Mehrfachnutzung durch Knoten, die von der Anwendung der Regel nicht betroffen sind). Eine Ausnahme bildet der Knoten, der dem Wurzel-Knoten des linken Musters der Regel entspricht und Knoten, die keine Kosten verursachen, z. B. Variablen oder Konstanten. Entsprechend darf eine Regel nur angewendet werden, wenn die Anzahl der Eingangs-Kanten eines Muster-Knotens mit der Anzahl der Eingangs-Kanten des Regel-Knotens übereinstimmt.

Die beschriebene Einschränkung betrachtet nur den Fall, dass lediglich eine Regel angewendet wird. Berücksichtigt man die Anwendung mehrerer Regeln, kann ein Muster, welches zunächst durch die Anwendung einer Regel teurer wird, zum Schluss nach An-

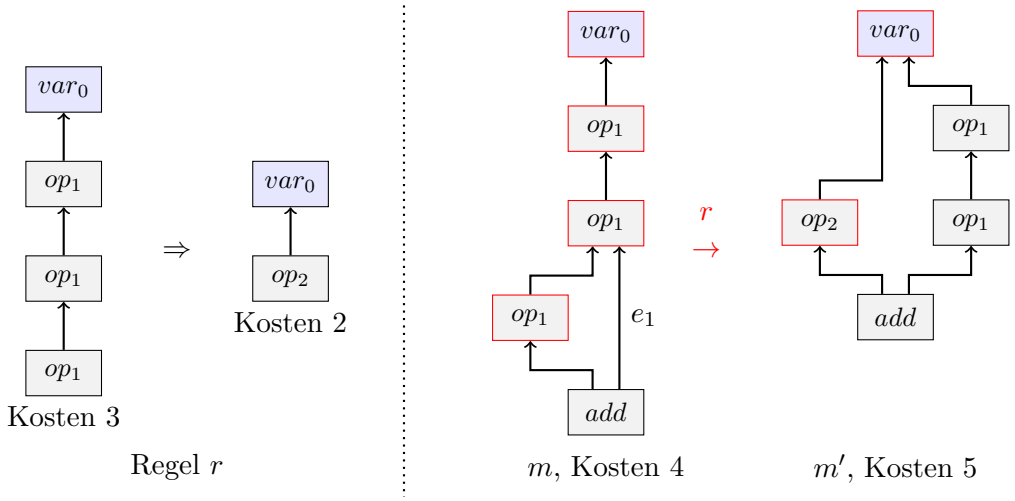


Abbildung 4.12: Nicht immer ist die Anwendung eine Optimierungsregel sinnvoll. Die Kosten von m nehmen durch Anwendung von r zu.

wendung mehrerer Regeln dennoch günstiger sein. Sollen solche Fälle berücksichtigt werden, wäre es für jedes Muster nötig, jede mögliche Kombination von nacheinander anwendbaren Regeln dahingehend zu untersuchen, ob eine Kombination existiert, deren resultierende Muster ebenso teuer, oder günstiger als das ursprüngliche Muster wären. Aus Komplexitätsgründen, die eine Überprüfung jeder Kombination mit sich bringt, wird dieser Ansatz im Rahmen dieser Arbeit nicht weiter verfolgt.

Verringerung des Suchraums durch Ausnutzen von Regeln

Wie zuvor beschrieben, werden alle Muster dahingehend untersucht, ob eine bereits bekannte Regel anwendbar ist, um so möglichst frühzeitig nicht-optimale Muster zu erkennen. Wenn man bedenkt, dass neue Muster auf der Grundlage bereits bekannter Muster erzeugt werden, stellt sich die Frage, ob es möglich ist, Muster auf die eine Regel anwendbar ist, bereits bei der Mustererzeugung zu vermeiden, indem nicht-optimale Muster nicht länger berücksichtigt werden. Würden nur Muster betrachtet, deren Graph einem gerichteten Baum entspricht, für die also zu jedem Knoten genau ein Pfad von der Wurzel aus existiert, träfe diese Annahme zu. Da in diesem Fall ein neues Muster je nach Stelligkeit der verwendeten Operation keinen, einen oder zwei Teilbäume, also Muster mit geringeren Kosten besitzt und diese nicht untereinander verbunden sind, kann jeder Teilbaum durch seine günstigste Repräsentation dargestellt werden. Alle anderen Fälle, also alle nicht-optimalen Teilbäume oder Muster können ignoriert werden, da auf sie ohnehin eine Ersetzungsregel anwendbar ist.

Für die in diesem Verfahren betrachteten Graphen bzw. Muster trifft dies allerdings nicht zu. Die Ursache hierfür ist die mögliche Mehrfachnutzung von Knoten durch unterschiedliche Teilmuster. Im Folgenden wird dies anhand von Abbildung 4.13 und Abbildung 4.14 erläutert. Angenommen es existieren die beiden Regel r und r' wie in der Abbildung gezeigt, sowie die Operationen op_1, op_2, op_3, op_4 mit den jeweiligen Kosten $cost_T(op_1) = cost_T(op_3) = 1$ und $cost_T(op_2) = cost_T(op_4) = 3$. Die Kosten der jeweiligen Muster sind in den Abbildungen angegeben.

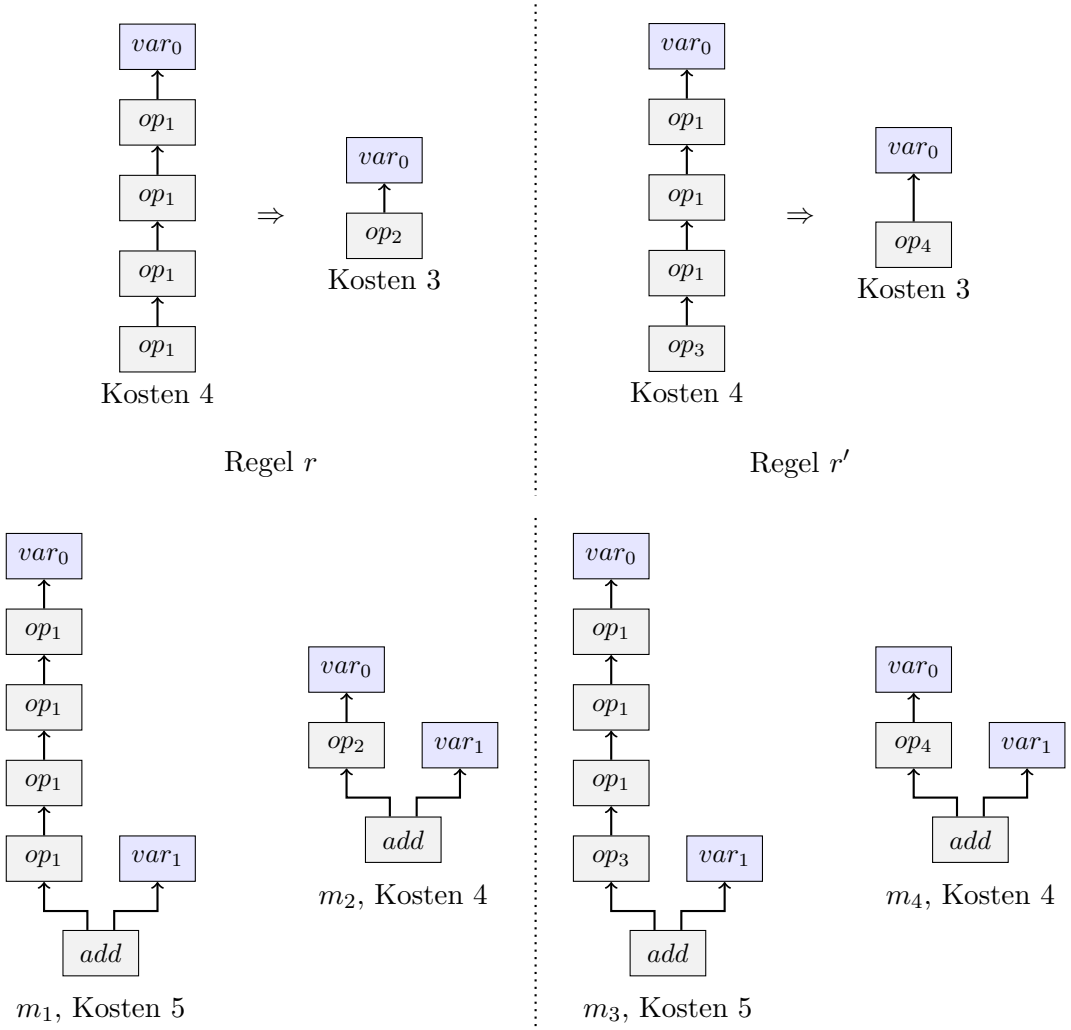
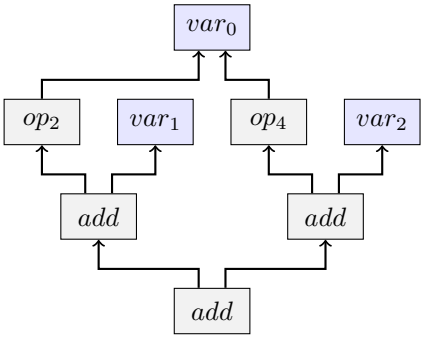
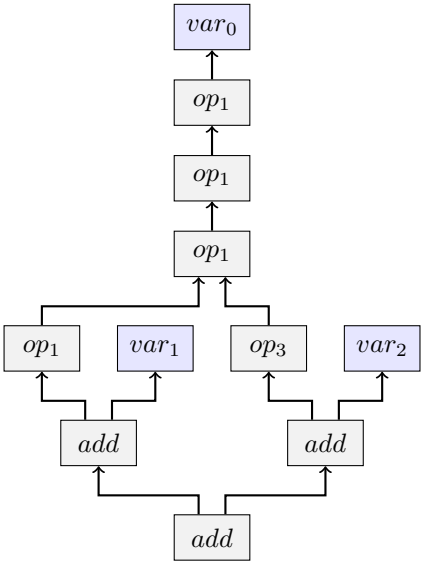


Abbildung 4.13: Abbildung zweier Regeln r und r' sowie jeweils ein optimales (m_2 und m_4) und ein nicht-optimales (m_1 und m_3) Muster, das aus der Kombination des rechten bzw. linken Musters der jeweiligen Regel mit einer Addition und einer Variablen erzeugt wurde.

Regel r ist auf das Muster m_1 anwendbar. Bei m_1 handelt es sich also nicht um das optimale Muster. Es sei angenommen, dass es sich bei m_2 um das zu m_1 gehörende optimale Muster handelt, welches auch durch Anwendung von r aus m_1 erzeugt werden kann. Analog verhält es sich mit dem Muster m_3, dessen zugehöriges optimales Muster m_4 ist und durch Anwendung der Regel r' aus m_3 erzeugt werden kann. In Abbildung 4.14 sind zwei weitere Muster m_5 und m_6 dargestellt. m_5 wurde durch Kombination der beiden optimalen Mustern m_2 und m_4 mit einer Addition erzeugt. m_6 hingegen besteht aus den beiden nicht-optimalen Mustern m_1 und m_3 sowie einer zusätzlichen Addition. Obwohl m_5 im Gegensatz zu m_6 aus optimalen Mustern besteht, ist m_6 günstiger. Der Grund hierfür besteht in der Mehrfachnutzung der drei op_1 Knoten nach dem var_0 Knoten in Muster m_6. Die Kosteneinsparung durch die Mehrfachnutzung der drei Knoten ist höher, als die Einsparung, die durch Anwendung der beiden Optimierungsregeln r und r' erreicht werden kann.



m_5, Kosten 9



m_6, Kosten 8

Abbildung 4.14: Abbildung zweier semantisch äquivalenter Muster m_5 und m_6. m_5 wurde aus den optimalen Mustern m_2 und m_4 und einer Addition erzeugt. m_6 wurde aus den beiden nicht-optimalen Mustern m_1 und m_3 sowie einer Addition erzeugt. Obwohl m_5 im Gegensatz zu m_6 aus optimalen Mustern erzeugt wurde, ist m_6 günstiger (vgl. Abbildung 4.13).

Da die Mustererzeugung auf der Erweiterung von Mustern mit geringerem Kostenwert beruht, genügt es nicht, ausschließlich optimale Muster zu berücksichtigen. Das eben gezeigt Beispiel belegt, dass für eine vollständige Suche nach dem optimalen Muster für eine Menge von semantisch äquivalenten Mustern auch nicht-optimale Muster zur Erzeugung neuer Muster berücksichtigt werden müssen. In dem eben vorgestellten Beispiel

würde das Muster m_6 nicht von der Mustererzeugung generiert werden, da auf die beiden zugrunde liegenden Muster m_1 und m_3 bereits eine Regel anwendbar ist und diese somit nicht optimal sind.

Eine Ausnahme bilden Muster, für die eine Regel existiert, deren rechte Seite ein Teilmuster der linken Seite ist, oder die lediglich den Wurzel-Knoten durch einen günstigeren Knoten ersetzen. Anders ausgedrückt, darf ein Muster nur dann von der weiteren Mustererzeugung ausgeschlossen werden, wenn eine Regel existieren, so dass durch deren Anwendung kein teureres Muster aufgrund von Mehrfachverwendern entstehen kann. Ein Beispiel zeigt Abbildung 4.15. Das Muster der rechten Seite der Regel r entspricht einem Teilmuster der linken Seite der Regel. Die Addition und die Konstante 0 werden entfernt. Wird r auf das ebenfalls in der Abbildung gezeigte Muster m angewendet, bleibt, trotz der Transformationsvorschrift von r, die Konstante 0 aufgrund einer Mehrfachverwendung bestehen. Da diese keine Kosten verursacht und auch durch r keine neuen Teilausdrücke dem Muster m' hinzugefügt werden, können sich die Kosten trotz Mehrfachverwendern nur verringern.

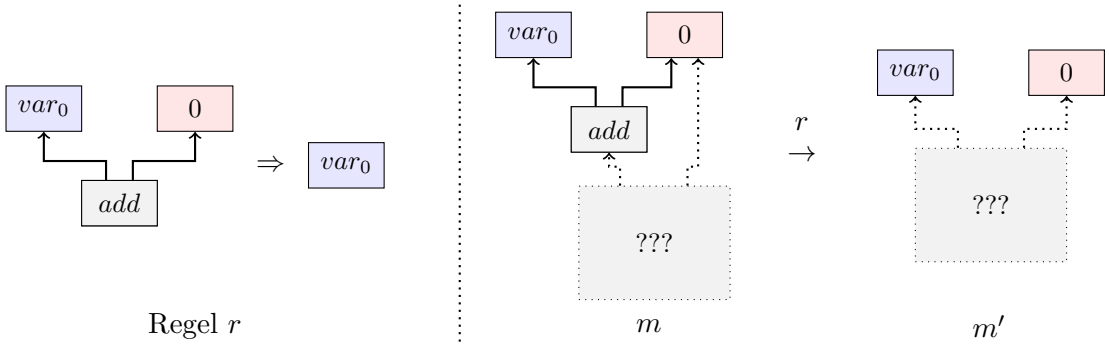


Abbildung 4.15: Die Konstante 0 bleibt auch nach Anwendung von r auf m aufgrund der Mehrfachverwendung bestehen. Da Konstanten den Kostenwert 0 besitzen werden allerdings keine neuen Kosten hinzugefügt.

4.3.2 Vorabtest

Der Vorabtest trifft für jedes neue Muster m eine Auswahl aus der Menge der bereits bekannten, optimalen Muster M_{Opt}. Die Idee dieses Tests besteht darin, die Ergebnisse eines neuen Musters bezüglich einer kleinen Menge von Testvektoren (Eingabevektoren) mit den Ergebnissen bereits vorhandener Muster zu vergleichen. Wird hierbei bereits eine Differenz festgestellt, folgt direkt, dass das entsprechende Musterpaar nicht äquivalent sein kann. Eine weitere Untersuchung durch den aufwendigeren Erfüllbarkeitstest ist nicht mehr notwendig. Das Ziel dieses Vorabtest ist es also, so viele Muster wie möglich auszuschließen, um den Erfüllbarkeitstest zu entlasten.

Für die Durchführung bestehen unterschiedliche Möglichkeiten, die sich im Zeitpunkt der Ausführung, sowie in der Zusammensetzung der Testvektoren unterscheiden.

Variante 1

Für jeden Vergleich zwischen einem neu erzeugten Muster m und einem bestehenden optimalen Muster m' werden die Ergebnisse neu berechnet und verglichen (vgl. Listing 4.2). Dies hat den Vorteil, dass keine Ergebnisse zwischengespeichert werden müssen, allerdings auch den Nachteil, dass die Berechnung der Ergebnisse für bereits bekannte Muster ständig wiederholt wird. Da die Ergebnisse für jeden Vergleich neu bestimmt werden, besteht bei dieser Variante die Möglichkeit, die Testvektoren bei jedem Vergleich zu variieren.

Listing 4.2 Vorabtest – Variante 1

```
 1: function VORABTEST1(m, M_Opt)
 2:     M_Cand ← CREATEEMPTYLIST()          ▷ Liste für Erfüllbarkeitstest Kandidaten
 3:     TV ← GETTESTVECTORS()                              ▷ Liste der Testvektoren
 4:
 5:     for all m' ∈ M_Opt do
 6:         equal ← true
 7:         for all t ∈ TV do
 8:             if RESULT(m, t) ≠ RESULT(m', t) then     ▷ Ergebnisse unterschiedlich?
 9:                 equal ← false                          ▷ ⇒ m' kein Kandidat
10:                 break
11:             end if
12:         end for
13:         if equal then
14:             STOREPATTERNINTO(M_Cand, m')                    ▷ m' ist Kandidat
15:         end if
16:     end for
17:
18:     return M_Cand
19: end function
```

Variante 2

Eine weitere Möglichkeit zeigt Listing 4.3. Die Idee dieser Variante besteht darin, die Ergebnisse für jedes neu erzeugte Muster m nur einmal zu berechnen und anschließend zu speichern (Zeile 6–11), so dass sie für spätere Vergleiche immer wieder verwendet werden können (Zeile 17). Dies spart Rechenzeit, benötigt jedoch Speicher. Zudem müssen die Testvektoren zu Beginn des Generierungsvorgangs festgelegt sein und können dann

nicht mehr verändert werden, da andernfalls die zu vergleichenden Ergebnisse auf unterschiedlichen Testvektoren beruhen würden und somit nicht mehr vergleichbar sind.

Listing 4.3 Vorabtest – Variante 2

```
1:  function VORABTEST2(m, M_Opt)
2:      M_Cand ← CREATEEMPTYLIST()        ▷ Liste für Erfüllbarkeitstest Kandidaten
3:      TV ← GETTESTVECTORS()                          ▷ Liste aller Testvektoren
4:      n ← COUNT(TV)                                  ▷ Anzahl der Testvektoren
5:
6:      i ← 0
7:      for all t ∈ TV do
8:          res ← RESULT(m, t)          ▷ wird für jedes Muster nur einmal bestimmt
9:          m.results[i] ← res
10:         i ← i + 1
11:     end for
12:
13:     for all m' ∈ M_Opt do
14:         i ← 0
15:         equal ← true
16:         while i < n do
17:             if m.results[i] ≠ m'.results[i] then     ▷ Ergebnisse unterschiedlich?
18:                 equal ← false                          ▷ ⇒ m' kein Kandidat
19:                 break
20:             end if
21:             i ← i + 1
22:         end while
23:         if equal then
24:             STOREPATTERNINTO(M_Cand, m')                ▷ m' ist Kandidat
25:         end if
26:     end for
27:
28:     return M_Cand
29: end function
```

Verwendete Variante

Im Rahmen dieser Arbeit wird eine Kombination aus beiden Varianten verwendet, die zusätzlich durch den Einsatz einer Hash-Funktion ergänzt wird. Listing 4.4 stellt den konkreten Ablauf dar. Um das Problem des höheren Speicherverbrauchs von Variante 2 gegenüber Variante 1 zu vermeiden, werden nicht die Ergebnisse gespeichert, sondern es wird aus ihnen ein Hashwert gebildet. Die hierzu verwendete Hashfunktion h besteht aus einer einfachen Gewichtung der Ergebnisse, die in der Reihenfolge in der sie berech-

net werden zunimmt und einer anschließenden Aufsummierung (Zeile 8). Mittels dieses Hashwertes kann das Muster m direkt einer Hash-Klasse $M_{h(m)}$ von Mustern zugeordnet werden. Die Hash-Klasse $M_{h(m)}$ kann direkt als Kandidatenmenge M_{Cand} verwendet werden. Ein expliziter Vergleich der Ergebnisse mit allen optimalen Mustern ist nicht zwingend notwendig.

Da die Hashwerte, die durch die beschriebene Hashfunktion berechnet werden nicht zwangsläufig eindeutig sind[1], können durch einen zusätzlichen Vergleich von Ergebnissen bezüglich derselben oder anderer Testvektoren weitere Muster ausgeschlossen werden. Hierzu wird der Vorabtest, analog zu Variante 1, ergänzt (Listing 4.4 Zeile 13–29), so dass die Muster mit demselben Hashwert noch einmal für eine Menge zusätzlicher Testvektoren TV_{ADD} explizit überprüft werden. Die Testvektoren werden zu Beginn des Generierungsvorgangs zufällig bestimmt.

Verbesserung durch systematisch erzeugte Testvektoren

Die hier beschriebene Vorgehensweise lässt sich weiter verbessern. Hierzu werden systematisch zusätzliche Testvektoren ermittelt, die zu einer weiteren Filterung der, durch den Hashwert des zu vergleichenden Musters $h(m)$, ausgewählten Hash-Klasse von Mustern $M_{h(m)}$ beitragen.

Zwei Muster m und m' sind dann semantisch äquivalent, wenn kein Testvektor bzw. Eingabevektor $tv \in \mathbb{Z}^n, n \in \mathbb{N}$ existiert, für den die Ergebnisse $m(tv)$ und $m'(tv)$ unterschiedlich sind. Umgekehrt gilt, dass zwei Muster genau dann nicht semantisch äquivalent sind, wenn ein solcher Eingabevektor tv existiert. Der im folgenden Abschnitt 4.3.3 beschriebene Erfüllbarkeitstest prüft genau diese Eigenschaft und erzeugt, falls $m \not\equiv m'$ gilt, einen Eingabevektor tv der dies beweist, für den also $m(tv) \neq m'(tv)$ gilt (siehe auch Listing 4.6 Zeile 9). Formal gilt für tv die folgende Definition 14.

Definition 14 (Zeuge)**.** *Seien m und m' zwei Muster. Sei weiter $tv \in \mathbb{Z}^n, n \in \mathbb{N}$ ein Eingabevektor für den gilt: $m(tv) \neq m'(tv)$. Daraus folgt direkt: $m \not\equiv m'$. v stellt einen Beweis hierfür dar. Man nennt tv einen Zeugen für $m \not\equiv m'$.*

Ein Muster m bildet zusammen mit allen anderen zu ihm semantisch äquivalenten Mustern eine Äquivalenzklasse $[m]_{\equiv}$. Aus der Definition der semantischen Äquivalenz lässt sich direkt folgern, dass, falls für zwei Muster m und m' sowie einen Eingabevektor tv gilt: $m_0(tv) \neq m'_0(tv)$, auch für alle anderen $m'' \in [m]_{\equiv}$ gilt: $m''(tv) \neq m'(tv)$. Ein Zeuge tv ist demnach auch Zeuge für ganze Äquivalenzklassen $[m]_{\equiv} \neq [m']_{\equiv}$.

Diese Eigenschaft eines Zeugen tv lässt sich für den Vorabtest nutzen. Hierzu werden alle Zeugen, also alle Eingabevektoren, die vom Erfüllbarkeitstest (vgl. Abschnitt 4.3.3) bei einem fehlgeschlagenen Äquivalenzbeweis erzeugt werden in einer Liste gespeichert,

[1] Aus unterschiedlichen Ergebnissen lassen sich unter Umständen dieselben Hashwerte berechnen

Listing 4.4 Vorabtest – Verwendete Variante

```
1: function VORABTEST(m, M_Opt)
2:     TV ← GETTESTVECTORS()                          ▷ Liste der Testvektoren
3:     m_cand ←⊥
4:
5:     i ← 1
6:     for all t ∈ TV do
7:         res ← RESULT(m, t)
8:         m.hash ← m.hash + (res ∗ i)          ▷ Berechnung des Hashwertes h(m)
9:         i ← i + 1
10:    end for
11:
12:    M_h(m) ← GETHASHEQUALPATTERN(m.hash, M_Opt)
13:    M_Cand ← CREATEEMPTYLIST()                  ▷ Kandidatenliste erzeugen
14:    TV_h(m) ← GETTESTVECTORSBYHASH(m.hash)          ▷ weitere Testvektoren
15:    TV_ADD ← GETADDTESTVECTORS()          ▷ Liste zusätzlicher Testvektoren
16:
17:    for all m′ ∈ M_h(m) do
18:        equal ← true
19:        for all t ∈ TV_h(m) ∪ TV_ADD do
20:            if RESULT(m, t) ≠ RESULT(m′, t) then      ▷ Ergebnisse unterschiedlich?
21:                equal ← false                              ▷ ⇒ m′ kein Kandidat
22:                break
23:            end if
24:        end for
25:        if equal then
26:            m_cand ← m′                                ▷ m′ ist einziger Kandidat
27:            break
28:        end if
29:    end for
30:
31:    return m_Cand
32: end function
```

sofern das zu untersuchende Muster optimal ist, wenn also kein semantisch äquivalentes Muster existiert. Die Eingabevektoren für nicht-optimale Muster, also für solche für die bereits ein äquivalentes Muster existiert, brauchen nicht gespeichert zu werden, da diese Muster für keinen Äquivalenz-Vergleich mehr herangezogen werden. Stattdessen wird immer das bereits vorhandene Muster verwendet. Zu jeder Hash-Klasse bzw. jedem Hashwert $h(m)$ existiert eine eigene Liste $TV_{h(m)}$ zur Speicherung der so ermittelten Eingabevektoren (vgl. Listing 4.6 Zeile 16 in Abschnitt 4.3.3). Mit diesen Eingabevektoren wird die Menge der Muster einer Hash-Klasse $M_{h(m)}$, die als Kandidaten für den Erfüllbarkeitstest in Frage kommen, weiter gefiltert.

Für die Eingabevektoren und Muster derselben Hash-Klasse besteht folgender Zusammenhang: Angenommen, es wird ein neues Muster m mit dem Hashwert $h(m)$ erzeugt und auf dieses Muster ist keine bisher ermittelte Regel anwendbar. Das Muster wir also durch den Vorabtest untersucht. Es lassen sich drei Fälle unterscheiden:

- $M_{h(m)} = \emptyset$:
  Es gilt, dass die Menge der Muster $M_{h(m)}$ einer Hash-Klasse noch leer ist. Demnach ist auch die Menge der Eingabevektoren $TV_{h(m)}$ bezüglich einer Hash-Klasse noch leer. Das neue Muster m kann direkt zur Menge der optimalen Muster M_{opt}, sowie zu dieser Hash-Klasse hinzugefügt werden, da noch kein anderer Kandidat für eine mögliche Äquivalenz existiert.

- $|M_{h(m)}| = 1$:
  Es gilt, dass bisher nur ein Muster m' der Hash-Klasse $M_{h(m)}$ zugeordnet ist. Demzufolge ist die Liste der Eingabevektoren $TV_{h(m)}$ noch leer. Für das neue Muster m besteht entweder die Möglichkeit, dass es zu dem bereits vorhandenen Muster m' äquivalent ist, oder nicht. Falls $m \equiv m'$ gilt, ist das Muster nicht optimal und es wird eine neue Regel erzeugt. Falls jedoch $m \not\equiv m'$ gilt, ist m' eine Kandidat und es wird durch den Erfüllbarkeitstest ein Zeuge tv ermittelt, welcher beweist, dass m und m' nicht semantisch äquivalent sind. m ist in diesem Fall ein optimales Muster und wird der Menge M_{opt}, sowie der Hash-Klasse $M_{h(m)}$ hinzugefügt, sowie tv der Menge $TV_{h(m)}$.

- $|M_{|h(m)|} > 1$:
  Es gilt, dass sich bereits mehrere optimale, d. h. semantisch nicht äquivalente Muster in der Hash-Klasse $M_{h(m)}$ befinden. Für das neue Muster m besteht, wie im vorherigen Fall die Möglichkeit, dass ein Muster $m' \in M_{h(m)}$ existiert, zu dem m äquivalent ist, oder dass es noch kein solches Muster gibt. Ist m zu einem Muster $m' \in M_{h(m)}$ äquivalent, dann gilt für jeden Eingabevektor $tv \in TV_{h(m)}$: $m(tv) = m'(tv)$. Da jedoch für jedes andere Muster $m'' \in M_{h(m)}$, mit $m' \not\equiv m''$ mindestens ein $tv' \in TV_{h(m)}$ existiert, so dass $m'(tv') \neq m''(tv')$ gilt, folgt, dass es kein anderes Muster außer $m' \in M_{h(m)}$ geben kann für das alle Eingabevektoren $tv \in TV_{h(m)}$ ein Ergebnis liefern, welches mit dem von m für dieselben Eingabevektoren übereinstimmt. Es gibt demnach nur genau ein Muster, für das alle bisher

ermittelten Eingabevektoren dasselbe Ergebnis wie m liefern. Demzufolge ist auch höchstens ein Kandidat durch den Erfüllbarkeitstest zu untersuchen. Falls m zu keinem $m' \in M_{h(m)}$ semantisch äquivalent ist, dann existiert ebenfalls höchstens ein m', so dass für jeden Eingabevektor $tv \in TV_{h(m)}$ $m(tv) \neq m'(tv)$ gilt, da für jedes Paar (m', m'') von Mustern aus $M_{h(m)}$ mindestens ein Zeuge $tv \in TV_{h(m)}$ existiert, der beweist, dass $m' \not\equiv m''$ gilt.

In der vorherigen Betrachtung wurde der Fall ausgelassen, das es zwar für ein neues Muster m eine Menge von semantisch äquivalenten Kandidaten geben kann, diese jedoch alle durch die im vorherigen Abschnitt beschriebene Menge von zusätzlichen Testvektoren TV_{ADD} ausgeschlossen wurden. In diesem Fall ist kein Erfüllbarkeitstest erforderlich und es wird auch kein Zeuge aufgrund eines Äquivalenzbeweises von m zu einem anderen Muster generiert. Dies ändert jedoch die Situation nicht, da sich zwar kein diesbezüglicher Zeuge bzw. Eingabevektor in der Hash-Klasse $TV_{h(m)}$ befindet, jedoch in der Menge TV_{ADD}.

4.3.3 Erfüllbarkeitstest

Der im vorherigen Abschnitt 4.3.2 beschriebene Vorabtest liefert für ein neues Muster m eine Vorauswahl M_{cand} von Mustern die möglicherweise semantisch äquivalent zu m sind. Durch die Verwendung von Zeugen wird erreicht, dass sich höchstens ein Muster m_{cand} in der Menge M_{cand} befinden kann. Für dieses Paar ist noch zu beweisen oder zu widerlegen, dass $m \equiv m_{cand}$ gilt. Dies ist die Aufgabe des hier beschriebenen Erfüllbarkeitstests.

Ein einfacher Ansatz um die semantische Äquivalenz zwischen zwei Mustern zu beweisen wäre es, entsprechend der Definition, die Ergebnisse zweier Muster für jeden möglichen Eingabevektor zu vergleichen. Es ist offensichtlich, dass dies aufgrund der Komplexität der Eingabemenge ein unpraktikabler Ansatz ist und daher ein effizienteres Beweisverfahren vorteilhaft wäre.

Für das hier vorgestellte Verfahren ebenso geeignet ist die Frage, ob zwei Muster m und m' nicht semantisch äquivalent sind, ob also ein Eingabevektor $tv \in \mathbb{Z}^n$ existiert, für den $m(tv) \neq m'(tv)$ gilt. Dieses Problem lässt sich wie folgt als Erfüllbarkeitsrelation der Aussagenlogik formulieren und mittels entsprechender Programme wie SAT-Solvern oder SMT-Solvern lösen:

$$m \not\equiv m' \Leftrightarrow \exists tv \in \mathbb{Z}^n : m(tv) \neq m'(tv)$$

Zur Veranschaulichung folgen zwei Beispiele.

Beispiel 3 (Semantische Äquivalenz)**.** *Gegeben seien die beiden Muster $m(x, y) = x + y$ und $m'(x, y) = x * y$. Die Frage ist nun, ob es zwei Werte x und y gibt, so dass die Ergebnisse von m und m' verschieden sind. Als Erfüllbarkeitsrelation formuliert:*

$$\exists x \in \mathbb{N}_0, y \in \mathbb{N}_0 : (x + y) \neq x * y$$

Wie leicht zu erkennen ist, kann dies durch $x = 2, b = y$ erfüllt werden, womit gezeigt ist, dass m und m' nicht semantisch äquivalent sind.

Beispiel 4 (Semantische Äquivalenz)**.** *Betrachten wir nun die beiden Muster $m = x + 0$ und $m' = x$. Dieses mal existiert kein x, so dass die Aussage:*

$$(x + 0) \neq x$$

wahr wird. Die beiden Muster m und m' sind semantisch äquivalent.

Für die Verwendung eines SAT-Solvers, wie dies bei Bansal und Aiken [3] der Fall ist, ist es notwendig, die beschriebenen Erfüllbarkeitsrelationen als boolesche Formeln darzustellen. Insbesondere muss dazu die Semantik der involvierten Muster in eine Formel aus booleschen Variablen und Operationen transformiert werden. Dies wird hier vermieden, indem anstelle eines SAT-Solvers ein SMT-Solver zur Lösung der Äquivalenzbeweise verwendet wird. Der Vorteil besteht darin, dass neben booleschen Formeln zusätzliche Theorien und die darin enthaltenen Operationen etc. genutzt werden können. Von besonderem Interesse ist die Bitvektor-Theorie, da sie einen Datentyp mit endlichem Wertebereich und entsprechende Operationen mit Modulo-Semantik (vgl. Abschnitt 2.2) zur Verfügung stellt. Für die Formulierung von SMT-Problemen existiert der Standard SMT-Lib v2 [4]. Hierdurch wird ein einfacher Austausch der SMT-Solver ermöglicht. Ein Beispiel in SMT-LIB v2 Notation zeigt das nachfolgende Listing 4.5.

Listing 4.5 SMT-Lib v2 Beispiel

```
1: (declare-fun x () (_ BitVec 8))
2: (assert (not (= (bvadd x (_ bv0 8)) x)))        ▷ x + 0 ≠ x
3: (check-sat)
```

Das Beispiel prüft die semantische Äquivalenz der beiden Muster $x + 0$ und x. In Zeile 1 wird zunächst eine neue Variable x als Bitvektor der Breite 8 Bit definiert. Zeile 2 beschreibt das Problem und Zeile 3 startet den Lösungsvorgang.

Das im folgenden angegebene Listing 4.6 beschreibt den Ablauf des angesprochenen Erfüllbarkeitstest in algorithmischer Form. In Zeile 4 wird das konkrete Problem für $m \equiv m'$ durch die CreateProof-Funktion erzeugt, welches anschließend in Zeile 5 durch die Solve-Funktion gelöst wird. Ist das Ergebnis dieser Funktion *wahr*, die Bedingung also

nicht erfüllbar[1], sind beide Muster m und m' äquivalent. Ist das Ergebnis jedoch für jedes $m' \in M_{Cand}$ *erfüllbar*[2], so ist m ein optimales Muster. Wie in Abschnitt 4.3.2 angesprochen, werden die Zeugen, also Eingabevektoren tv, die beweisen, dass $m \not\equiv m'$ gilt, extrahiert und falls m optimal ist, in der Liste $TV_{h(m)}$ gespeichert (Zeile 9, 10, 16).

Listing 4.6 Erfüllbarkeitstest

```
1:  function ErfüllbarkeitstestTest(m, m_Cand)
2:      TV_Tmp ← CreateEmptyList                    ▷ Erzeuge List für gefundene Zeugen
3:      if m_cand ≠⊥ then
4:          proof ← CreateProof(m, m_cand)          ▷ Erzeuge Beweis-Instanz
5:          unsat ← Solve(proof)                    ▷ Beweis lösen
6:          if unsat then
7:              return m_cand                       ▷ m_cand ist semantisch äquivalentes Muster
8:          else
9:              v ← ExtractProofVector(proof)       ▷ Zeuge extrahieren
10:             InsertInto(TV_Tmp, v)               ▷ Zeuge zwischenspeichern
11:         end if
12:     end if
13:
14:     TV_h(m) ← GetTestvectorsByHash(h(m))
15:
16:     Append(TV_h(m), TV_Tmp)                     ▷ Neue Zeugen zur Liste hinzufügen
17:     return ⊥                                    ▷ Kein äquivalentes Muster ⇒ m optimal
18: end function
```

4.4 Zusammenwirken der Komponenten

In diesem Abschnitt soll der vollständige Ablauf eines Generierungsvorgangs noch einmal zusammengefasst werden. Anschließend folgt ein ausführliches Beispiel.

4.4.1 Ablauf

Listing 4.7 stellt den Ablauf eines Generierungsvorgangs in algorithmischer Form dar. Die Prozedur Generate wird mit einem Wert *limit* aufgerufen, der die gewünschte Kostenobergrenze für den durchzuführenden Generierungsvorgang festlegt. Zunächst werden drei Listen R_{Rules}, M_{Opt} und M_{Erz} erzeugt. R_{Rules} dient zur Speicherung der generierten Regeln, M_{Opt} beinhaltet die optimalen Muster und M_{Erz} speichert alle erzeugten

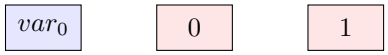
[1] engl. unsatisfiable
[2] engl. satisfiable

Muster (auch optimale) (Zeile 2–4). Die Funktion CreateNextPattern (Zeile 6 und 29) erzeugt, wie in Abschnitt 4.2.2 beschrieben, das jeweilige nächste Muster, basierend auf der Menge der bisher erzeugten Muster M_{Erz}. Dies wird solange fortgeführt bis die in *limit* festgelegte Kostenobergrenze überschritten wird (Zeile 7). In Zeile 8–29 erfolgt die Musteranalyse. Zunächst wird durch die Funktion RuleApplicableTest geprüft, ob eine der bisher ermittelten Regeln auf das neu erzeugte Muster m anwendbar ist (siehe Abschnitt 4.3.1). Als nächstes folgt die Suche nach einem semantisch äquivalenten Muster. Hierbei werden alle Muster p berücksichtigt, die eine Permutation der Variablen von m darstellen. Die Funktion CreateNextPermutation erzeugt dazu jeweils die nächste Permutation von m. Für jedes dieser Muster p folgt zunächst die Analyse durch den Vorabtest, der aus der Menge der optimalen Muster M_{Opt} eine Kandidatenmenge M_{Cand} bestimmt, wie in Abschnitt 4.3.2 erläutert. Diese besteht aufgrund der Verwendung von Zeugen aus nur einem Muster m_{cand}. Anschließend wird dieses Muster durch den Erfüllbarkeitstest auf seine semantische Äquivalenz zu m untersucht. Ist m_{cand} semantisch äquivalent zu m, so wird eine neue Regel erzeugt (Zeile 16–17) und mit dem nächsten Muster m fortgefahren. Existiert kein semantisch äquivalentes Muster, handelt es sich bei m um ein optimales Muster und m wird zur Liste M_{Opt} hinzugefügt (Zeile 24).

4.4.2 Beispiel

Für das Beispiel wird eine stark vereinfachte Zwischensprache betrachtet, die lediglich eine Addition als Operation kennt, Variablen mit einer Datentypbreite von einem Bit und die beiden Konstanten 0 und 1.

Zunächst werden nacheinander die in Abbildung 4.16 gezeigten Basismuster mit Kosten 0 für eine Variable var_0, die Konstante 0 und die Konstante 1 erzeugt. Bisher existieren noch keine Regeln, also wird auch kein Muster verworfen. Wie man leicht erkennen kann, ist zudem keines der Muster semantisch äquivalent zu einem anderen, weswegen es auch keine neue Regel gibt.



(a) (b) (c)

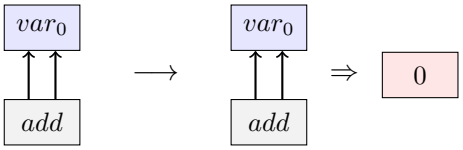
Abbildung 4.16: Abbildung aller Basismuster (Operationen: *add*, Datentypbreite: 1 Bit).

Im nächsten Iterationsschritt werden die Muster mit Kosten 1 aus den Basismustern und einer Operation, der Addition gebildet. Abbildung 4.17 zeigt diese Muster, sowie die daraus resultierenden Aktionen.

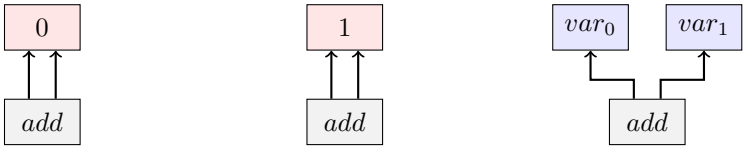
Für das erste Muster der Kostenstufe 1, das Muster (d), welches aus der Addition und dem Basismuster (a) gebildet wurde, existiert mit dem Muster (b) aus Abbildung 4.16 ein

Listing 4.7 Ablauf des Generierungsvorgangs in algorithmischer Form

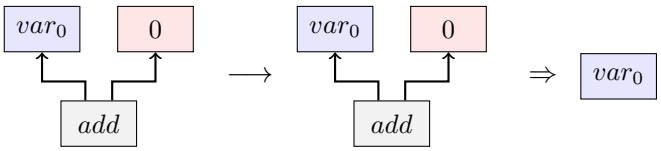
```
1:  procedure GENERATE(limit)
2:      R_Rules ← CREATEEMPTYLIST()                          ▷ Liste speichert Regeln
3:      M_Opt ← CREATEEMPTYLIST()                  ▷ Liste speichert optimale Muster
4:      M_Erz ← CREATEEMPTYLIST()                  ▷ Liste speichert alle erzeugten Muster
5:
6:      m ← CREATENEXTPATTERN(M_Erz)                          ▷ erstes Muster erzeugen
7:      while m.costs ≤ limit do
8:          result ← RULEAPPLICABLETEST(R_Rules, m)
9:          if result ≠ true then
10:             p ← m, m' ←⊥
11:             while p ≠⊥ ∧ m' =⊥ do
12:                 m_Cand ← VORABTEST(p, M_Opt)                  ▷ Kandidat ermitteln
13:
14:                 m' ← ERFÜLLBARKEITSTEST(p, m_Cand)
15:                 if m' ≠⊥ then                    ▷ äquivalentes Muster gefunden?
16:                     r ← CREATENEWRULE(p, m')
17:                     STORERULEINTO(R_Rules, r)
18:                 else                ▷ weitersuchen für strukturell identische Muster
19:                     p ← CREATENEXTPERMUTATION(p)          ▷ nächste Permutation
20:                 end if
21:             end while
22:
23:             if m' =⊥ then
24:                 STOREPATTERNINTO(M_Opt, m)                ▷ m ist optimales Muster
25:             end if
26:         end if
27:
28:         STOREPATTERNINTO(M_Erz, m)
29:         m ← CREATENEXTPATTERN(M_Erz)
30:     end while
31: end procedure
```



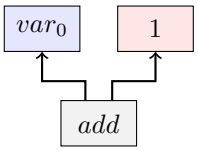
(d) äquiv. zu (b) neue Regel r_0



(e) entf. durch r_0 (f) entf. durch r_0 (g) optimal



(h) äquiv. zu (a) neue Regel r_1



(i) optimal

Abbildung 4.17: Abbildung aller Muster und Regeln der Kostenstufe 1 (Operationen: *add*, Datentypbreite: 1 Bit).

semantisch äquivalentes, günstigeres Muster. Daher wird eine neue Regel $r_0 : (d) \rightarrow (b)$ erzeugt und der Menge der gefundenen Regeln hinzugefügt. Für die beiden nächsten Muster (e) und (f) kann diese Regel direkt angewendet werden. (e) und (f) werden verworfen. Auf das Muster (g) kann weder eine Regel angewendet werden, noch existiert ein semantisch äquivalentes Muster. (g) ist daher optimal und wird der Menge der optimalen Muster hinzugefügt. Für das folgende Muster (h) existiert wieder ein semantisch äquivalentes Muster: (a). Hieraus ergibt sich eine weitere Regel $r_1 : (h) \rightarrow (a)$. Das letzten Muster (i) dieser Kostenstufe ist wieder ein optimales Muster.

4.5 Erweiterung im Zusammenhang mit Konstanten

Konstanten stellen aufgrund ihrer Vielzahl und der damit verbundenen starken Zunahme erzeugbarer Muster eine besondere Herausforderung an den Generierungsprozess hinsichtlich der benötigten Rechenzeit dar. Der nachfolgende Abschnitt befasst sich mit dieser Thematik und zeigt einen Lösungsansatz auf, der den im vorherigen Teil beschriebenen Generierungsvorgang so erweitert, dass dieser in der Lage ist sämtliche Konstanten bei der Mustererzeugung und damit bei der Suche nach Ersetzungsregeln zu berücksichtigen.

4.5.1 Symbolische Konstanten und komplexe symbolische Konstanten

Ein erster Schritt um viele Konstanten auf einmal handhaben zu können, besteht in der Abstraktion von Konstanten, durch symbolische Konstanten. Hierzu wird ein neuer Operanden-Typ *sc* der Menge der zugelassenen Typen hinzugefügt (vgl. Definition 6). Die Kosten $cost_T(sc)$ dieses neuen Typs entsprechen den Kosten von Konstanten, also dem Wert 0. Für den Ausgangsgrad gilt wie bei Konstanten: $arity(sc) = 0$. Eine symbolische Konstante entspricht formal der folgenden Definition 15.

Definition 15 (Symbolische Konstante)**.** *Ein Knoten $v \in V_G$ eines Musters m mit dem zugehörigen Graphen G heißt symbolische Konstante, wenn gilt: $type(v) = sc$. Der Typ sc besitzt den Kostenwert* 0*, entsprechend betragen die Kosten einer symbolischen Konstante ebenfalls* 0*. Eine symbolische Konstante entspricht einem Stellvertreter für jede mögliche Konstante. Bei der Anwendung einer Regel überdeckt daher eine symbolische Konstante eine herkömmliche Konstante.*

Um symbolische Konstanten in den Generierungsvorgang zu integrieren, wird, genau wie für Konstanten, zu Beginn der Mustererzeugung ein Basismuster mit einem Knoten vom Typ einer symbolischen Konstante generiert.

Symbolische Konstanten sind allgemeiner als Konstanten und verhalten sich ähnlich zu Variablen. Für die Anwendung von Regeln gilt, dass eine Regel auch dann anwendbar

ist, wenn anstelle einer Konstante mit demselben Wert, im linken Muster der Regel eine symbolische Konstante steht. Im Gegensatz zu Variablen gilt bei symbolischen Konstanten diese Eigenschaft nur für konkrete Konstanten, nicht aber für andere Operationen.

Mit symbolischen Konstanten lässt sich also ein Muster so verallgemeinern, dass es speziellere Muster, mit konkreten anstelle von symbolischen Konstanten, überdecken kann. Für die Aufstellung von allgemeineren Regeln im Zusammenhang mit Konstanten ist dies jedoch noch nicht ausreichend. Insbesondere für die Konstantenfaltung müssen nach wie vor alle Regeln mit expliziten Konstanten aufgezählt werden:

$$\begin{aligned} 0 + 0 &\rightarrow 0 \\ 1 + 0 &\rightarrow 1 \\ 1 + 1 &\rightarrow 2 \\ &\vdots \end{aligned}$$

Alleine für die Addition in $\mathbb{Z}_{8Bit}$ sind schon $256^2 = 65536$ Regeln möglich. Vorteilhaft wäre es jedoch, wenn die Konstantenfaltung für eine Operation in einer einzigen Regel zusammengefasst werden könnte. Es stellt sich also die Frage, wie eine solche Regel aussehen kann:

$$sc_0 + sc_1 \rightarrow ???$$

Die linke Seite der Regel gestaltet sich einfach. Es wird eine Operation und je nach Stelligkeit dieser Operation ein oder zwei Operanden in Form von symbolische Konstanten benötigt. Damit lässt sich das entsprechende Muster für die linke Seite der Regel aufstellen, z. B. $sc_0 + sc_1$ (siehe Abbildung 4.18).

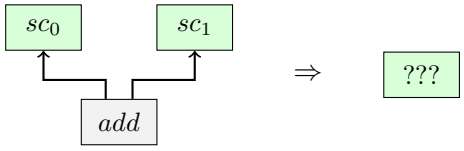


Abbildung 4.18: Regel zu Konstantenfaltung – Rechte Seite noch unklar.

Zu klären bleibt, wie die rechte Seite der Regel aussehen muss. Eine einfache symbolische Konstante, z. B. sc_2 die jede beliebige Konstante repräsentieren kann ist hierfür zu allgemein. Bei Anwendung der Regel, durch beispielsweise einen Übersetzer, wäre nicht definiert wie der Wert von sc_2 konkret aussehen muss. Es wird demnach eine symbolische Konstante benötigt, deren Wert in Abhängigkeit von anderen symbolischen Konstanten angegeben werden kann. Hierzu wird das Konzept der symbolischen Konstanten um *komplexe symbolische Konstanten* erweitert, die nicht nur einen Wert repräsentieren können, sondern denen auch eine Semantik in Form einer Berechnungsfunktion zugeordnet werden kann. Etwas formaler:

Definition 16 (Komplexe symbolische Konstante)**.** *Als komplexe symbolische Konstante sc_{clx} werden alle symbolischen Konstanten bezeichnet, denen eine Berechnungsfunktion $f : SC^n \rightarrow SC$ mit mindestens einer Operation, sowie Operanden $SC^n_{in} = \{sc^0_{in}, \ldots, sc^n_{in}\}$ in Form von symbolischen Konstanten zugeordnet sind. Die Kosten einer komplexen Konstante betragen, analog zu symbolischen Konstanten* 0*.*

Die Forderung nach einem Kostenwert von 0 wie bei Konstanten und symbolischen Konstanten ist trotz der zugehörigen Operationen sinnvoll, da sich aus einer komplexen Konstante bei Anwendung der Regel beispielsweise in einem Übersetzer immer ein konkreter Wert in Form einer Konstante ergibt. Die Operationen, die zur komplexen Konstante gehören müssen zur Laufzeit eines Programms nicht mehr ausgeführt werden.

Die Semantik einer komplexen symbolischen Konstante lässt sich in Form eines Musters f darstellen, welches der entsprechenden Berechnungsfunktion entspricht. Der Wert einer komplexen symbolischen Konstante kann dann als Ersetzungsregel aufgefasst werden, die den vom Muster f abhängigen Wert auf den Wert sc_{clx} der Konstante abbildet. Ein Beispiel einer solchen Ersetzungsregel zeigt die folgende Formel:

$$f : sc_0 + sc_1 \mapsto sc_{clx_0}$$

Für den Wert gilt also:

$$sc_{clx_0} = sc_0 + sc_1$$

Mit Hilfe solcher komplexer symbolischer Konstanten lässt sich nun die Semantik der Konstantenfaltung für eine Operation als Ersetzungsregel in einer allgemeinen Form darstellen. Abbildung 4.19 zeigt eine solche allgemeine Regel zur Konstantenfaltung für die Addition. Die Addition der beiden symbolischen Konstanten s_0 und s_1 wird auf die neue symbolische Konstante sc_{clx_0} abgebildet, deren abstrakter Wert sich aus $s_0 + s_1$ ergibt.

Abbildung 4.19: Durch eine komplexe Konstante kann die rechte Seite der Regel aus Abbildung 4.18 angegeben werden.

In Abbildung 4.20 wird ein Muster gezeigt (linke Seite), für das mehrere Optimierungen zusammenspielen müssen, um zu einem optimalen Muster zu gelangen. Dazu sind zunächst durch eine Reassoziierung die Konstanten zusammenzufassen und anschließend

eine Konstantenfaltung durchzuführen. Mit komplexen symbolischen Konstanten kann eine allgemeine Regel angegeben werden (die in der Abbildung gezeigt wird), welche die Optimierung für alle möglichen Konstanten-Kombinationen in einer Regel zusammenfasst.

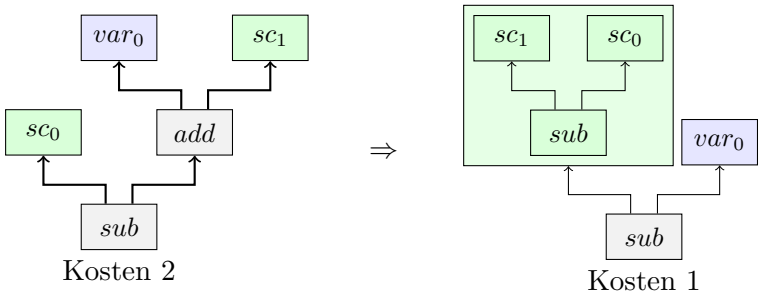


Abbildung 4.20: Abbildung einer Regel, die Reassoziierung und Konstantenfaltung kombiniert. Durch symbolische und komplexe symbolische Konstanten ist die Regel für alle Konstanten gültig.

Einschränkung bei der Verwendung von komplexen symbolischen Konstanten

Der Einfachheit wegen werden komplexe symbolische Konstanten nur auf der rechten Seite einer Regel erlaubt, da hier bei Anwendung der Regel die Operanden für die Berechnungsfunktion bereits durch die linke Seite festgelegt und somit bekannt sind. Werden komplexe symbolische Konstanten auf der linken Seite einer Regel zugelassen, müssen die Operanden gegebenenfalls aus dem Wert einer Konstante rekonstruiert werden.

Zur Veranschaulichung betrachten wir die beiden in Abbildung 4.21 dargestellten semantisch äquivalenten Muster m und m'. Es sind zwei Regeln denkbar: $r : m \to m'$ und $r' : m' \to m$. Da komplexe Konstanten auf der linken Seite einer Regel ausgeschlossen sind, wird die Regel r' nicht erzeugt werden, für dieses Beispiel soll sie dennoch betrachtet werden. Bei Anwendung der Regel r ist die Berechnung der komplexen symbolischen Konstante einfach. Der Wert ergibt sich aus $sc_0 + sc_0$. Wird beispielsweise r auf ein Muster angewendet, so dass sc_0 der Konstante 4 entspricht, ergibt sich die komplexe Konstante aus $sc_0 + sc_0$ bzw. $4 + 4 = 8$. Der umgekehrte Fall ist komplizierter. Wird r' auf ein Muster angewendet, so dass die komplexe Konstante einer 8 entspricht, muss die Konstante erst halbiert werden, um auf sc_0 bzw. 4 zu kommen. Auf ein Muster mit einer ungeraden Konstante ist r' zudem gar nicht anwendbar. Auch dies müsste extra überprüft werden.

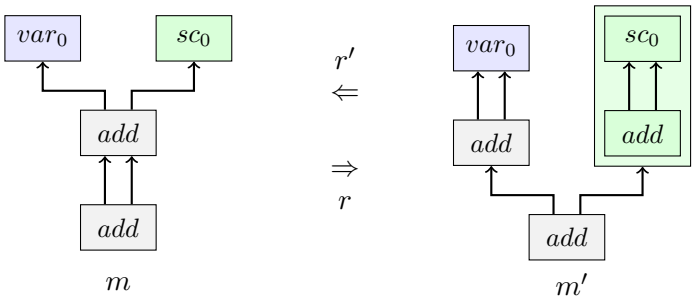


Abbildung 4.21: Eine Regel zwischen m und m' kann in beide Richtungen definiert werden – Zwei Regeln sind möglich.

4.5.2 Eingeschränkte symbolische Konstanten

Ein weiterer Schritt, um von Regeln mit konkreten Konstanten zu abstrahieren, ist die Einführung symbolischer Konstanten mit eingeschränktem, aber festem oder von anderen symbolischen Konstanten abhängigem Wertebereich. Dazu wird nun erlaubt, dass symbolischen Konstanten eine Bedingung zugeordnet werden kann, die den möglichen Wertebereich den die symbolische Konstante repräsentiert einschränkt.

Definition 17 (Eingeschränkte symbolische Konstante)**.** *Als eingeschränkte symbolische Konstante sc_{cnd} werden alle symbolischen Konstanten bezeichnet, denen eine Bedingung B zugeordnet ist, die den Wertebereich der symbolischen Konstante einschränkt.*

Es werden die zwei nachfolgend beschriebenen Arten von Einschränkungen unterschieden:

Einschränkungen ohne Argument

Hierunter wird eine symbolische Konstanten sc verstanden, deren Wertebereich unabhängig von anderen symbolischen Konstanten eingeschränkt ist. Einige mögliche Einschränkungen sind im Folgenden aufgeführt:

- alle Konstanten $c \in \mathbb{Z}_{8Bit} : c < 0$
- alle Konstanten $c \in \mathbb{Z}_{8Bit} : c \geq 0$
- alle Konstanten $c \in \mathbb{Z}_{8Bit} : c > 0$
- alle Konstanten $c \in \mathbb{Z}_{8Bit} : c > 8$

Abbildung 4.22 zeigt eine Regel, die auf der Einschränkung der symbolischen Konstante sc_0 auf Werte $sc_0 < 0$ beruht. Für Werte $sc_0 > 0$ gilt diese Regel hingegen nicht.

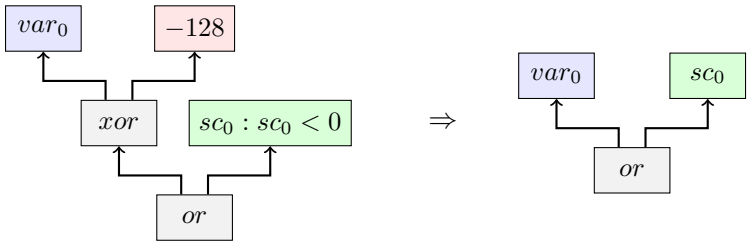


Abbildung 4.22: Abbildung einer Regel mit eingeschränkter symbolischer Konstante, Einschränkung ohne Argument (z. B. $sc_0 \mapsto -2$).

Einschränkungen mit einem Argument

Oft sind Regeln nur für bestimmte Kombinationen von Konstanten gültig, wie es bei der in Abbildung 4.23 dargestellten Regel der Fall ist. Diese ist genau für solche Muster anwendbar, deren Konstante an der Position von sc_1 dem invertierten Bitmuster der Konstante an der Position von sc_0 entspricht. Konkrete Beispiele hierfür wären u. a. $sc_0 \mapsto 2, sc_1 \mapsto -3$ oder $sc_0 \mapsto 5, sc_1 \mapsto -6$. Um solche Abhängigkeiten darstellen zu können, werden daher komplexere Einschränkungsmöglichkeiten benötigt, die den Wertebereich abhängig von einer anderen symbolischen Konstante einschränken. Einige Einschränkungen sind nachfolgend aufgeführt:

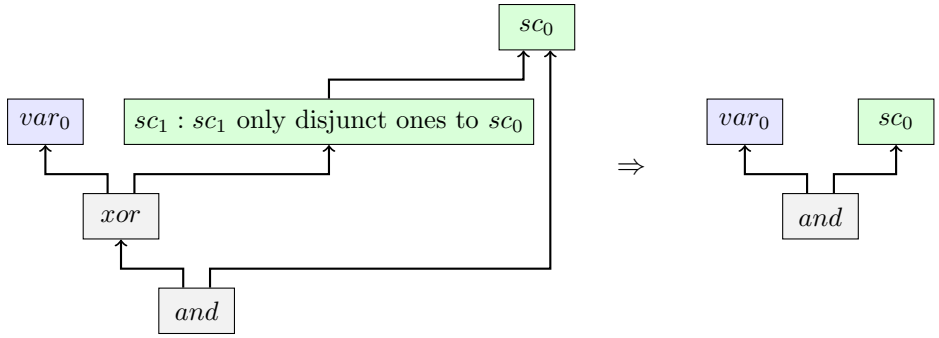
- sc_1 entspricht dem invertierten Bitmuster von sc_0 (inverted)
- sc_1 enthält nur gesetzte Bits, die auch in sc_0 gesetzt sind (common ones)
- sc_1 besitzt nur gesetzte Bits, die in sc_0 nicht gesetzt sind (disjunct ones)

4.5.3 Geschwindigkeit vs. Vollständigkeit bei der Regel-Aggregation

Durch die Verwendung symbolischer Konstanten wird eine Aggregation der Regeln erreicht. Es stellt sich nun die Frage, zu welchem Zeitpunkt diese Aggregation stattfinden soll. Prinzipiell bestehen hierzu zwei mögliche Varianten mit jeweils Vor- und Nachteilen, auf die im Folgenden eingegangen wird.

Frühe Aggregation

Bei der frühen Aggregation wird versucht, möglichst früh allgemeine Regeln (durch symbolische Konstanten) zu finden, so dass speziellere Muster aufgrund der bereits vorhandenen allgemeineren Regel nicht mehr betrachtet werden müssen. Dies beschleunigt den Generierungsvorgang, da vor allem weniger Muster durch den langsamen Erfüllbarkeits-



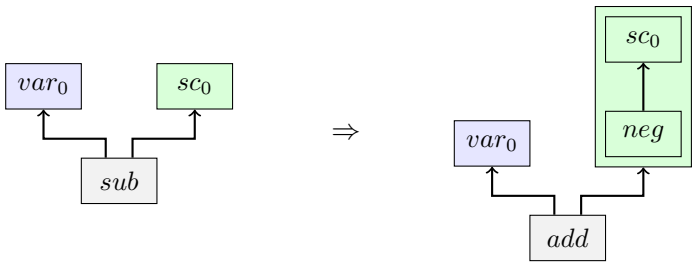
Regel r

Abbildung 4.23: Abbildung einer Regel mit eingeschränkter symbolischer Konstante, Einschränkung mit einem Argument. r kann nur auf Muster angewendet werden, deren Konstante an der Stelle von sc_1 höchstens gesetzte Bits besitzt, die in der Konstante an der Stelle von sc_0 nicht gesetzt sind (z. B. $sc_0 \mapsto 2 \Rightarrow sc_1 \mapsto -3$).

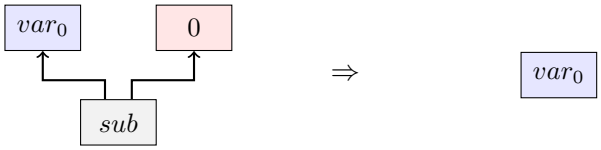
test zu prüfen sind. Andererseits bringt diese frühe Aggregation auch eine Einschränkung mit sich, da nicht mehr jedes Muster explizit untersucht wird. Eventuelle Spezialfälle für bestimmte Konstanten werden so möglicherweise nicht erkannt. Dies verdeutlicht auch die Abbildung 4.24.

Laut der Regel r können alle Muster der Form $var_0 - sc_0$ wie z. B. $x - 5$ auf das entsprechende Muster $var_0 + (-sc_0)$, also z. B. $x + (-5)$ abgebildet werden. Die Regel ist auf alle Muster dieser Form, bei denen sc_0 durch eine konkrete Konstante ersetzt wurde, anwendbar, weswegen die entsprechenden Muster verworfen werden. Insbesondere gilt dies für das Muster $var_0 - 0$. Tatsächlich lässt sich $var_0 - 0$ noch günstiger als var_0 schreiben. Diese Regel wird aufgrund der zuvor beschriebenen allgemeineren Regel r nicht gefunden. Dies ist ein Beispiel dafür, wie, bedingt durch Aggregation, nicht alle spezielleren Optimierungsregeln gefunden werden.

Dieses Problem stellt nur auf den ersten Blick eine Einschränkung dar. Angenommen es existiert eine Transformationsregel $r : m \to m'$, die eine Optimierungsregel $r' : p \to p'$ verdeckt, weil r auf p anwendbar ist und somit für p kein semantisch äquivalentes Muster gesucht wird. Zu p muss allerdings ein semantisch äquivalentes Muster q existieren, welches dem Muster entspricht, dass sich durch Anwendung von r auf p ergibt. Für q gilt, da es sich bei r um eine Transformationsregel handelt: $cost_M(p) = cost_M(q)$. Weiter muss ein semantisch äquivalentes, günstigeres Muster p' zu p existieren, da sonst eine Optimierungsregel r' überhaupt nicht möglich wäre. Aus $p \equiv q$ und $p \equiv p'$ folgt: $q \equiv p'$. Es ist also möglich eine Regel $r'' : q \to p'$ zu erzeugen. Da weiter gilt, dass p' günstiger ist als p, und p ebenso teuer wie q ist, folgt zwangsläufig das p' vor q erzeugt wird und



Regel r



Regel r'

Abbildung 4.24: Durch Aggregation kann das Auffinden speziellerer Regeln verhindert werden. Regel r ist auf das linke Muster aus Regel r' anwendbar, weswegen r' zwar möglich ist, aber nicht erzeugt wird.

r'' in Form einer Optimierungsregel erzeugt wird. Das Muster p wird daher zwar nicht direkt auf das Muster p' abgebildet, aber es existieren zwei Regeln r und r'', so dass p über das Muster q in zwei Schritten auf das günstigere Muster p' abgebildet werden kann.

Nachträgliche Aggregation

Entgegen der zuvor beschriebenen frühen Aggregation, kann eine Aggregation auch erst im Nachhinein durchgeführt werden. Dies kann entweder ganz zum Schluss oder am Ende jeder Kostenstufe geschehen. Der Nachteil besteht darin, dass jedes Muster explizit behandelt, also generiert und überprüft werden muss. Es ergibt sich allerdings der Vorteil, dass so auch alle möglichen Regeln gefunden werden können.

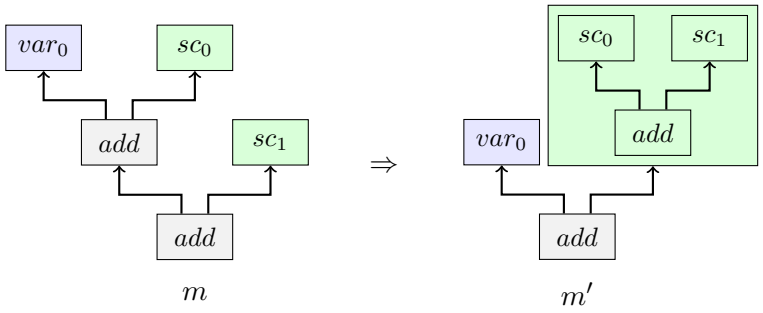
Aufgrund der hohen Komplexität in Bezug auf die Rechenzeit erscheint die früher Aggregation geeigneter. Aus diesem Grund wird auf die nachträgliche Aggregation im Rahmen dieser Arbeit verzichtet.

4.5.4 Zyklisch anwendbare Regeln

Auf ein spezielles Problem in Zusammenhang mit der Anwendbarkeit von Regeln soll hier im Besonderen eingegangen werden. Hierzu zunächst zwei einführende Beispiele:

Beispiel 1: Verhindern von Regeln

Abbildung 4.25 zeigt eine Optimierungsregel r, die zwei beliebige Konstanten reassoziiert und anschließend zu einer Konstante faltet. Es wird eine Addition eingespart.



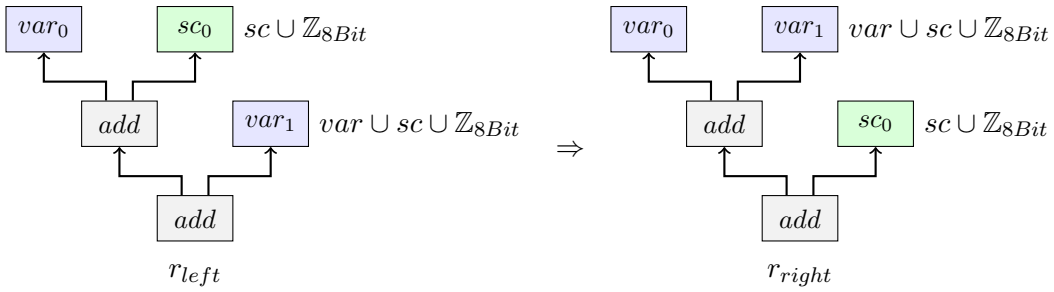
Regel r

Abbildung 4.25: Optimierungsregel – Es werden zwei Optimierungsschritte in einer Regel zusammengefasst. sc_0 und sc_1 werden reassoziiert und anschließend zu einer Konstante gefaltet. Eine Addition wird eingespart.

In Abbildung 4.26 wird eine weitere Regel r' gezeigt, die eine Normalisierung durchführt. Durch Anwendung von r' wird eine Konstante oder symbolische Konstante mit var_1 vertauscht. Für den weiteren Generierungsvorgang bewirkt diese Regel, dass keine weiteren Muster betrachtet werden, die eine Konstante oder symbolische Konstante an Stelle von sc_0 in Muster r_{left} besitzen. Stattdessen werden nur noch die semantisch äquivalenten Muster, die dem Muster r_{right} oder einem spezielleren Muster entsprechen, berücksichtigt. Ein zu r_{right} spezielleres Muster ist beispielsweise gegeben, wenn sc_0 in Muster r_{right} durch eine Konstante ersetzt wird. Abbildung 4.26 zeigt zudem für die beiden Knoten var_0 und sc_0 die Mengen der Typen, von denen Knoten mit diesem Typ bei Anwendung der Regel überdeckt werden können[1].

Die Regel r' bringt allerdings ein Problem mit sich. Angenommen r' sowie m' aus Abbildung 4.25 existieren bereits und m aus Abbildung 4.25 ist gerade durch die Mustererzeugung generiert worden und soll nun durch die Musteranalyse untersucht werden. m

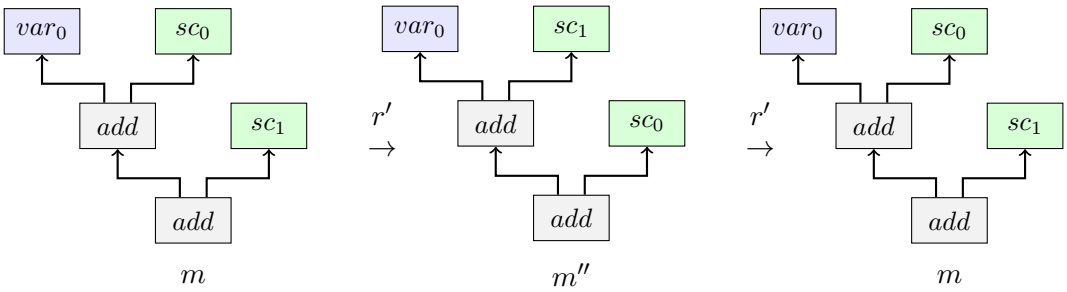
[1] Für Variablen müssen eigentlich auch alle anderen Knotentypen mit in die Menge der zulässigen Typen aufgenommen werden. Für das Beipsiel ist dies jedoch unerheblich.



Regel r'

Abbildung 4.26: Zyklisch anwendbare Regel – r_{left} und r_{right} besitzen eine gemeinsame Teilmenge von Mustern, die sie überdecken, z. B. Muster m aus Abbildung 4.25. r' ist sowohl auf m, als auch auf das resultierende Muster anwendbar.

wird jedoch von der Musteranalyse direkt verworfen, da r' auf m anwendbar ist. Nach der Transformationsregel r' ist die normalisierte Variante von m ein Muster m'' (vgl. Abbildung 4.27), das dieselbe Struktur, also denselben Aufbau wie m besitzt und sich lediglich durch die vertauschten symbolischen Konstanten sc_0 und sc_1 unterscheidet. Hier beginnt das Problem. r' ist nicht nur auf m, sondern auch auf dessen normalisierte Variante m'' anwendbar, womit die normalisierte Variante von m'' wieder m ist. Es entsteht ein Zyklus, indem m durch zweifaches anwenden von r' immer wieder auf sich selbst abgebildet wird. Für die Musteranalyse würde dies bedeuten, dass sowohl m als auch die normalisierte Variante m'' verworfen werden. Demnach wird nie ein Muster mit dem Aufbau von m durch den Vorabtest oder den Erfüllbarkeitstest behandelt und die in in Abbildung 4.25 gezeigte Regel r wird nie gefunden.



Regel r'

Abbildung 4.27: r' ist sowohl auf m als auch auf das resultierende Muster m'' anwendbar. Das ursprüngliche Muster m wird wieder hergestellt.

Beispiel 2: Entfernen optimaler Muster

Für dieses Beispiel wird aus Darstellungsgründen angenommen, dass die Datentypbreite von Variablen auf 3 Bit reduziert und der Wertebereich von Konstanten auf {-4,...,3} beschränkt ist. Ausgangspunkt dieses Beispiels ist die Abbildung 4.28. Wie sich zeigt, existiert für jedes Muster aus der Abbildung mit negativer Konstante ein semantisch äquivalentes Muster mit positiver Konstante. Bei näherer Betrachtung erkennt man, das für die Konstanten ein Zusammenhang durch die Relation $sc0_{new} = sc0_{old} + (-4)$ gegeben ist. Mittels komplexer symbolischer Konstanten lässt sich eine verallgemeinerte Regel angeben, die in Abbildung 4.29 gezeigt wird.

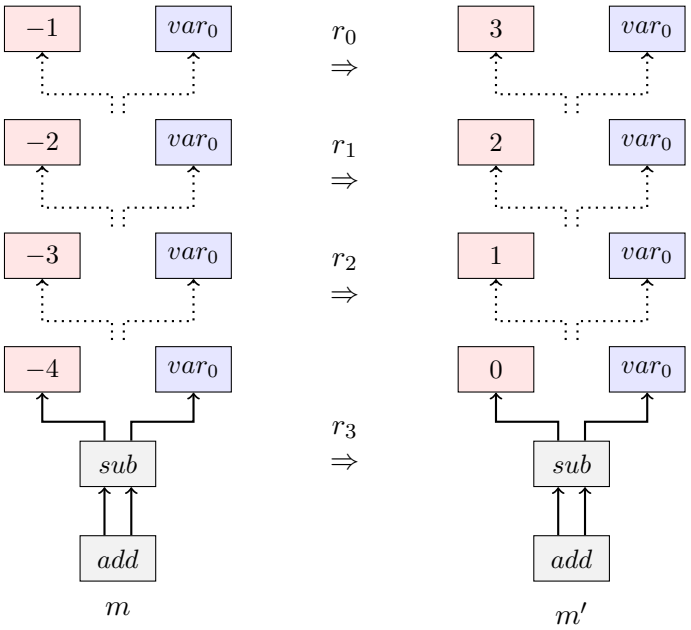
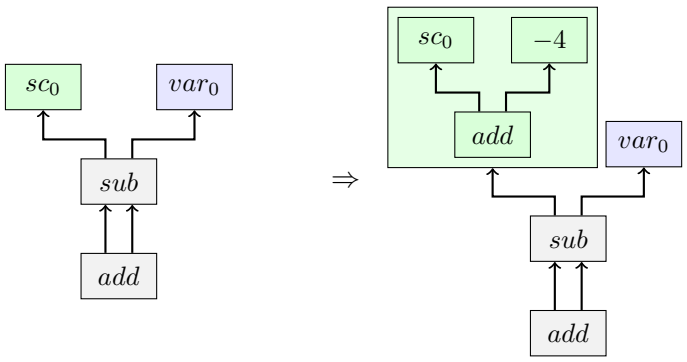


Abbildung 4.28: Für jedes Muster mit negativer Konstante (linkes) existiert ein semantisch äquivalentes Muster mit positiver Konstante (rechts). Pro Paar ist eine Regel möglich.

Hier besteht, ähnlich zu dem vorherigen Beispiel, das Problem, dass die in Abbildung 4.29 dargestellte Regel r nach ihrer Anwendung auf ein Muster m auch auf das resultierende Muster m' angewendet werden kann. Beispielsweise wird die Konstante -4 durch r auf die Konstante 0 abgebildet. Da der Knoten sc_0 im linken Muster der Regel aber alle Konstanten überdecken kann, ist die Regel auch auf das Muster m' mit der Konstante 0 anwendbar. 0 wird hierdurch wieder auf -4 abgebildet. Auch hier ergibt sich ein Zyklus von anwendbaren Regeln, genauer von r, durch den nicht nur die Muster aus Abbildung 4.28 mit einer negativen Konstante, sondern auch die, zu diesen Mustern semantisch äquivalenten, optimalen Muster mit positiver Konstante entfernt werden.



Regel r

Abbildung 4.29: Abbildung einer verallgemeinerten Regel – r_0, r_1, r_2, r_3 aus Abbildung 4.28 lassen sich durch symbolische Konstanten in einer Regel r zusammenfassen.

Die optimalen Muster bilden jedoch die Grundlage für neue Muster und werden als mögliche rechte Seite von Regeln benötigt. Sie dürfen nicht entfernt werden.

Verallgemeinerung

Wie in den beiden vorherigen Beispielen gezeigt wird, sind Regeln, oder allgemeiner eine Folge von Regeln $R = (r_0, \ldots, r_n), n \in \mathbb{N}_+$ problematisch, wenn es möglich ist, aus einem Muster m durch Anwendung einer solchen Folge R von Regeln wieder das Ursprungsmuster m zu erzeugen.

Hierzu zunächst eine Definition:

Definition 18 (Regelzyklus)**.** *Ein Regelzyklus innerhalb einer Regelmenge RM, besteht dann, wenn ein Muster m, sowie eine Folge von Regeln $R = (r_0, \ldots, r_n)$, mit $r_0, \ldots, r_n \in RM$ existiert, so dass gilt:*

$$m \xrightarrow{r_0} \ldots \xrightarrow{r_n} m$$

Regelzyklen in Regelmengen stellen bei der Verwendung der Regelmenge, beispielsweise in einem Übersetzer ein Problem dar. Bei jeder Verwendung einer Regel aus dieser Menge ist zu prüfen, ob ein Zyklus besteht, da ansonsten eine Terminierung bei der Anwendung der Regeln nicht garantiert ist. Schon aus diesem Grund ist es sinnvoll, Regelzyklen bereits beim Erzeugen einer Regelmenge zu verhindern. Darüber hinaus stören Regelzyklen auch den eigentlichen Generierungsvorgang wie Beispiel 1 und 2 zeigen. Das Entfernen optimaler Muster aufgrund eines Regelzyklus ist besonders kritisch. In diesem Fall ist nicht mehr garantiert, dass es sich bei einem Muster m auf der rechten Seite einer

Regel wirklich um das optimale Muster aus der Menge der zu m semantisch äquivalenten Muster handelt.

Sollen Regelzyklen in einer Menge von Regeln ausgeschlossen werden, ist für jede neue Regel, die dieser Menge hinzugefügt werden soll, zu prüfen, ob anschließend ein Zyklus besteht. Falls ja, darf die Regel nicht der Menge hinzugefügt werden. Da dieses Vorgehen zusätzlichen Aufwand für den Generierungsprozess bedeutet, wird im Rahmen dieser Arbeit eine solche Überprüfung nicht im vollen Umfang durchgeführt. Stattdessen wird ein alternatives Vorgehen gewählt, von dem zwar nicht erwiesen ist, dass es tatsächlich alle Zyklen verhindert. Allerdings konnte während der Implementierung und Evaluierung kein Fall beobachtet werden, der zu diesbezüglichen Problemen geführt hat.

Lösungsansatz

Bevor auf den eigentlich Vorgang zur Lösung, oder genauer zur Begrenzung des Problems eingegangen wird, folgt noch eine Vorüberlegung.

Eine Folge von Regeln mit mindestens einer Optimierungsregel kann keinen Regelzyklus bilden, da für eine Optimierungsregel $r : r_l \to r_r$ mit den beiden Mustern r_l und r_r, sowie deren Kosten $cost_M(r_l)$ und $cost_M(r_r)$ gilt: $cost_M(r_l) > cost_M(r_r)$. Die Kosten nehmen für das resultierende Muster ab. Eine umgekehrte Regel $r' : r'_l \to r'_r$ mit $cost_M(r'_l) < cost_M(r'_r)$ wird nach Definition 12 ausgeschlossen, weswegen es nicht möglich ist ein teureres Muster zu erzeugen. Eine Rekonstruktion des ursprünglichen Musters m ist nach Anwendung einer Optimierungsregel mit den zugelassenen Regeln nicht möglich.

Für Folgen, die ausschließlich aus Transformationsregeln bestehen, sind zwei Fälle zu unterscheiden. Folgen mit mindestens einer Transformationsregel, die ein Muster auf ein strukturell unterschiedliches Muster abbildet und Folgen, bei denen alle Transformationsregeln nur aus strukturell identischen Mustern aufgebaut sind.

Definition 19 (strukturell identische Muster)**.** *Zwei Muster m und m' sind strukturell identisch, wenn sich die Muster nur im Typ der Knoten v mit Ausgangsgrad $arity(type(v)) = 0$ unterscheiden.*

Anders formuliert sind zwei Muster strukturell identisch, wenn sich die Muster lediglich dadurch unterscheiden, dass in einem der Muster anstelle einer Variablen ein Konstante oder eine symbolische Konstante, oder anstelle einer symbolischen Konstante eine Konstante befindet.

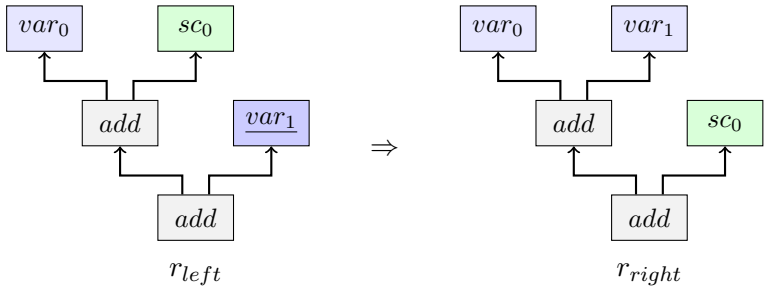
Für Transformationsregeln mit zwei strukturell unterschiedlichen Mustern besteht der folgende Zusammenhang. Angenommen es existieren zwei strukturell unterschiedliche Muster m und m' sowie eine Regel $r : m \to m'$. Es gilt aufgrund der Aufzählungsreihenfolge das m' vor m erzeugt wurde. Daraus folgt, dass es keine Regel $r' : m' \to m$ geben

kann. Existiert eine weiteres Muster m'' sowie eine weitere Regel $m' \rightarrow m''$, dann gilt analog, dass m'' vor m' erzeugt wurde woraus direkt folgt, dass m'' auch vor m erzeugt wurde. Ein Rekonstruktion von m durch eine zweite Regel ist also auch nicht möglich. Diese Überlegung lässt sich analog für drei oder mehr Regeln fortführen. Es ist jedoch nicht ausgeschlossen, dass eine Transformationsregel existiert, die auf ein Teilmuster von m' oder eines der folgenden Muster anwendbar ist und dadurch wieder das ursprüngliche Muster m erzeugt werden kann.

Bei Transformationsregeln mit zwei strukturell identischen Mustern können definitiv Regelzyklen entstehen. Dies zeigen die Beispiele 1 und 2 zu Beginn dieses Abschnittes. Der Grund besteht darin, dass es für zwei strukturell identische Muster möglich ist, dass beide eine gemeinsame Teilmenge von spezielleren Mustern überdecken. Ein Muster m' gilt gegenüber einem Muster m als spezieller, wenn m und m' strukturell identisch sind und m' von m überdeckt werden kann, nicht aber umgekehrt.

Die Idee um dieses Problem zu lösen besteht darin, das Muster der linken Seite einer Regel $r : r_{left} \rightarrow r_{right}$ mit zwei strukturell identischen Mustern r_{left} und r_{rigth} derart einzuschränken, dass dieses zum einen nicht auf das Muster r_{right} und zum anderen auch auf kein spezielleres Muster als r_{right} angewendet werden kann. Variablen, Konstanten oder symbolische Konstanten, die nicht an einer Änderung durch die Regel beteiligt sind, stellen hierbei kein Problem dar. Sie tragen nicht dazu bei, dass durch die Anwendung der Regel eine andere Regel anwendbar wird, die nicht bereits zuvor anwendbar gewesen wäre. Ein Beispiel für eine solche Variable ist var_0 aus Abbildung 4.30. Problematisch sind solche Knoten, die durch die Regel verändert werden, wie beispielsweise die symbolische Konstante sc_0 aus Abbildung 4.30, die als Operand einer anderen Operation zugeordnet wird. Konstanten müssen bei der Prüfung auf die Anwendbarkeit einer Regel immer genau übereinstimmen, also auch denselben Wert besitzen. Stimmen Konstanten in der Regeldefinition hinsichtlich ihrer Position und ihres Wertes nicht überein, ist die entsprechende Regel auch nicht auf das Muster anwendbar, welches sich aus der Anwendung der Regel ergibt. Bleiben noch Variablen und symbolische Konstanten, die an einer Änderung durch die Regel beteiligt sind. Diese können auch Knoten mit anderen Typen überdecken. Um zu verhindern, dass zwei Muster eine gemeinsame Teilmenge von Mustern überdecken, werden nacheinander zunächst die Variablen und anschließend symbolische Konstanten, in der Reihenfolge ihrer Indizes auf ihren eigentlichen Typ beschränkt. D. h. eine Variable darf dann nur noch Variablen und symbolische Konstanten nur noch symbolische Konstanten bei der Anwendung einer Regel überdecken. Die Einschränkung wird sukzessive fortgeführt bis für ein Knotenpaar (v, v') gilt, dass v und v' keine gemeinsame Teilmenge von möglichen Knotentypen überdecken können, wobei v aus einem Muster m stammt und v' dem Knoten entspricht, der sich in einem zu m strukturell identischen Muster m' an derselben Position wie v befindet. In Abbildung 4.30 wird dies bereits durch die Einschränkung von var_1 erreicht. var_1 aus Muster r_{left} kann nur noch Variablen überdecken, während sc_0 nur Konstanten und symbolische Konstanten überdecken kann.

Komplexe sowie eingeschränkte symbolische Konstanten schränken, wie bereits angesprochen den Wertebereich der Konstanten, die sie überdecken können ein. Dies ist bei der hier beschriebenen Einschränkung von Mustern zu berücksichtigen. Ein Beispiel zeigt Abbildung 4.31. sc_0 im linken Muster kann nur negative Konstanten überdecken, während für die komplexe symbolische Konstante im rechten Muster, bedingt durch die Einschränkung von sc_0, nur positive Konstanten als Ergebnis möglich sind. Daher ist keine weitere Einschränkung der symbolischen Konstante sc_0 nötig.



Regel r

Abbildung 4.30: Abbildung einer Regel mit eingeschränkter Variable var_1. var_1 darf bei der Anwendung der Regel nur noch anderen Variablen überdecken.

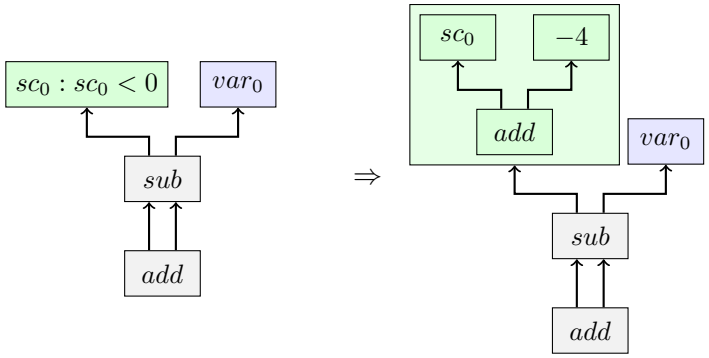


Abbildung 4.31: Aufgrund der Einschränkung von sc_0 im linken Muster auf negative Konstanten, kann die komplexe Konstante im rechten Muster nur positive Werte annehmen. -4 entspricht der Konstante mit dem geringsten Wert bei einer Bitbreite von 3 Bit.

5 Evaluation

Aufgrund der vielen Kombinationsmöglichkeiten von Operationen, Variablen und Konstanten ist zu erwarten, dass die Menge der erzeugbaren Muster schon bei kleinen Kostenstufen sehr groß ist. Neben der Anzahl der erzeugten Optimierungs- und Transformationsregeln wird in diesem Abschnitt vor allem evaluiert, wo die Grenzen der Durchführbarkeit des in dieser Arbeit vorgestellten Verfahren OPTGEN liegen. Ein weiterer betrachteter Aspekt ist die Frage, ob durch dieses Verfahren neue Optimierungsregeln gefunden werden und ob dieses Regeln in der Praxis Anwendung finden.

5.1 Testumgebung

Generierungsvorgang Die Testumgebung für den Generierungsvorgang bildet ein System aus Intel Core i5-3450 mit einer Taktfrequenz von 3100 MHz und 16 384 MB Arbeitsspeicher. Als Softwareplattform wird ein Ubuntu Linux in der Version 12.04 64-Bit eingesetzt. Zur Durchführung der Äquivalenzbeweise, wie in Abschnitt 4.3.3 beschrieben, wird der von Microsoft Research entwickelte SMT-Solver Z3 [8] in der Version 4.0 verwendet.

Evaluierung der erzeugten Optimierungsregeln Als Testumgebung für die Evaluierung der erzeugten Optimierungsregeln wird ein System aus einem Intel Core i3-550 Prozessors mit einer Taktfrequenz von 3200 MHz sowie 1878 MB Arbeitsspeicher eingesetzt. Als Betriebssystem dient ein Ubuntu Linux in der Version 12.04 in der 32-Bit Variante. Testgrundlage bildet die SPEC CINT2000 und CINT2006 Benchmark-Suite. Für die Messungen wird die Taktfrequenz der CPU auf 1200 MHz beschränkt.

5.2 Messungen

5.2.1 Generierungsvorgang

Die untersuchte Implementierung von OPTGEN berücksichtigt die folgenden Operationen: $not, neg, or, and, xor, add, sub, mul$. Die Datentypbreite und damit auch der Wertebereich von Konstanten beträgt 8 Bit, also 256 Konstanten.

Tabelle 5.1 zeigt die ermittelten Ausführungszeiten der beschriebenen Implementierung von Optgen für die einzelnen Kostenstufen sowie die Anzahl der erzeugten Muster und der generierten Regeln. Für die Kostenstufe 3 war keine Ausführung mit einer vollständigen Menge aller Operationen in annehmbarer Zeit möglich. Aus diesem Grund wurde eine partielle Generierung, einmal für die arithmetischen Operationen neg, add, sub, mul und einmal für die Bit-Operationen not, or, and, xor für die Kostenstufe 3 durchgeführt. Die Ergebnisse sind in Tabelle 5.1 aufgeführt. Die Menge der erzeugten Muster nimmt durch die Erhöhung der Kostenstufe von eins auf zwei deutlich zu. In noch stärkerem Maße steigt die benötigte Zeit, von 31 s auf knapp 1,25 h (4 499 s). Da Optgen nicht bei jedem Übersetzungsvorgang ausgeführt werden muss, sondern nur einmal bei der Entwicklung des Übersetzers, erscheint dieser Wert in Anbetracht dessen, dass alle Konstanten berücksichtigt wurden als akzeptabel.

Kostenstufe	1	2	3 (Arithm.)	3 (Bit)
Erzeugte Muster	363 734	4 189 969	8 173 051	85 052 711
Optimale Muster	1 738	364 164	74 792	2 011 961
Optimierungsregeln	17	99	90	162 640
Transformationsregeln	6	264 892	346	412 539
Laufzeit	31 s	4 499 s	374 s	621 548 s

Tabelle 5.1: Übersicht über die Ausführungszeiten, die erzeugten Muster und Regeln, aufgeschlüsselt nach Kostenstufen.

Die benötigte Laufzeit eines Generierungsvorgangs hängt stark von den involvierten Operationen ab. Dies ist an den großen Unterschieden zwischen den Laufzeiten der Kostenstufe 3 für arithmetische Operationen und der Kostenstufe 3 für Bit Operationen zu erkennen. Eine Erklärung ist, dass sich Regeln mit ausschließlich arithmetischen Operation besser aggregieren lassen, wofür auch die niedrigere Anzahl der ermittelten Optimierungs- und Transformationsregeln spricht.

Dieser Eindruck bestätigte sich bereits während der Implementierung von Optgen. Es zeigt sich, dass sich vor allem für Muster mit Bit-Operationen und Konstanten viele Regeln ergeben, die auf Abhängigkeiten zwischen den Konstanten eines Musters basieren. Durch das schrittweise hinzufügen eingeschränkter symbolischer Konstanten mit neuen Bedingungen, war jedes mal eine Laufzeitverbesserung und eine Verringerung der erzeugten Regel zu beobachten. Evtl. lässt sich das Konzept der eingeschränkten symbolischen Konstanten weiter ausbauen, so dass eine weitere Beschleunigung des Generierungsvorgangs erreicht wird.

Aufgrund der langen Laufzeiten stellt sich die Frage, welche Phase des Verfahrens den größten Anteil der Laufzeit beansprucht. Wie zu erwarten ist und wie auch den Laufzeiten aus Tabelle 5.2 entnommen werden kann, ist dies sowohl der Vorabtest als auch der Erfüllbarkeitstest.

	zusätzliche Testvektoren			Zeugen	kombiniert	
	1000/2000	1000/1000	1000/100	1000/0	1000/1000	2000/1000
Vorabtests	1 177 108	1 177 264	1 180 883	1 180 883	1 177 264	1 067 103
Erfüllbarkeitstests	971 915	945 897	1 086 231	413 454	378 485	379 988
Vorabtest Fehlentsch.	66,26%	69,97%	65,35%	22,31%	14,07%	15,02%
Mustererzeugung	61 s	63 s	59 s	64 s	61 s	62 s
Regel anwendbar	531 s	534 s	526 s	552 s	543 s	539 s
Vorabtest	3 031 s	2 874 s	2 882 s	3 602 s	3 252 s	2 370 s
Erfüllbarkeitstest	2 845 s	2 790 s	2 882 s	1 598 s	1 530 s	1 528 s
Gesamtlaufzeit	6 468 s	6 263 s	6 349 s	5 815 s	5 386 s	4 499 s

Tabelle 5.2: OPTGEN Ausführungsstatistik für unterschiedliche Konfigurationen des Vorabtests, Kostenstufe: 2, Operationen: $not, neg, or, and, xor, add, sub, mul$.

Der Erfüllbarkeitstest lässt sich nicht durch Parameter beeinflussen und ist auf die Qualität des Vorabtests angewiesen, um weniger Laufzeit zu benötigen, also weniger Muster behandeln zu müssen. Daher wird im Folgenden der Einfluss des Vorabtests bei unterschiedlicher Parametrisierung untersucht. Die Ergebnisse sind in Tabelle 5.2 zusammengefasst.

Evaluiert wurden unterschiedliche Konfigurationen des Vorabtests. Die ersten drei Messungen betrachten eine feste Anzahl von Testvektoren (1000 Stück) für die Ermittlung der Hash-Klasse $M_{h(m)}$, sowie je 2000, 1000 und 100 zusätzliche Testvektoren $tv \in TV_{ADD}$, mit denen die Muster einer Hash-Klasse gefiltert werden. Die Verwendung von Testvektoren (Zeugen), die sich aus dem Erfüllbarkeitstest ergeben, wurde hierzu deaktiviert. Die vierte Messung untersucht, wie sich der Vorabtest gegenüber den vorherigen drei Testfällen verhält, wenn die Muster einer Hash-Klasse nur mit Zeugen aus dem Erfüllbarkeitstest gefiltert werden. Die fünfte und sechste Messung betrachtet eine Kombination beider Ansätze. Hierbei wurde die Anzahl der Testvektoren, die zur Berechnung des Hashwerts verwendet werden variiert (1000 und 2000 Testvektoren). Tabelle 5.2 zeigt neben den Laufzeiten der einzelnen Phasen von OPTGEN auch die Anzahl der durchgeführten Vorabtests, die Anzahl der durchgeführten Erfüllbarkeitstest und den prozentualen Anteil an Fehlentscheidungen, die der Vorabtest trifft. Eine Fehlentscheidung des Vorabtests liegt dann vor, wenn zwei Muster vom Vorabtest als äquivalent angesehen werden, der Erfüllbarkeitstest dies aber widerlegt.

Die Anzahl der Fehlentscheidungen ohne die Verwendung von Zeugen erscheint mit mehr als 65% hoch. Wie Tabelle 5.2 zeigt, bringt der Mehraufwand durch eine größere Anzahl zusätzlicher Testvektoren nicht zwangsläufig auch eine geringere Fehlentscheidungsrate oder Gesamtlaufzeit. Das Verhältnis aus Kosten und Nutzen ist für die getestete Konfiguration mit 1000 zusätzlichen Testvektoren am besten, obwohl die Fehlentscheidungsrate mit 69,97% am höchsten ist. Die Verwendung von Zeugen bringt eine deutliche Verbesserung, wie sich an der niedrigeren Fehlentscheidungsrate von 22,31% und an der

kürzeren Gesamtlaufzeit von 5386 s zeigt. Dass der Aufwand durch die Verwendung von zusätzlichen Testvektoren $tv \in TV_{ADD}$ dennoch gerechtfertigt ist, belegt die nächste Messung. Hier wurden sowohl Zeugen als auch zusätzliche Testvektoren (hier wurde die beste Konfiguration aus den ersten drei Messungen verwendet) genutzt. Die Fehlentscheidungsrate konnte um weitere 8% auf 14% verringert werden. Die sechste Messung zeigt, dass ein weitere Verbesserung möglich ist, wenn die Anzahl der Testvektoren, die für die Hashwert-Berechnung verwendet werden, erhöht wird. Zwar wird die Qualität des Vorabtests nicht verbessert, aber die Gesamtlaufzeit deutlich verringert. Eine Begründung hierfür ist eine größere Streuung der Hashwerte und dadurch eine Verringerung der Anzahl der Muster pro Hash-Klasse, so dass der Vorabtest weniger Kandidaten pro neuem Muster untersuchen muss. Hieraus resultiert eine Laufzeitabnahme des Vorabtests um 882 s auf 2 370 s gegenüber dem vorherigen fünften Testfall.

5.2.2 Evaluation der erzeugten Optimierungsregeln

Heutige Übersetzer beherrschen ein Vielzahl lokaler Optimierungen. Durch Optgen sollen die letzten noch nicht berücksichtigten lokalen Optimierungsmöglichkeiten gefunden werden, so dass ein Übersetzer um diese erweitert werden kann. Ob solche Fälle existieren und ob diese bei der Übersetzung von Programmen in der Praxis vorkommen, wird in diesem Abschnitt evaluiert.

Die Grundlage dieser Evaluierung bildet libFirm [13], einmal in der Originalversion und einmal in einer, um ausgewählte Optimierungsregeln, die von Optgen erzeugt wurden, erweiterten Variante. Bei den ausgewählten Optgen-Regeln handelt es sich um 46 Regeln die aus den Kostenstufen 2, 3 (nur arithmetische Operationen) und 3 (nur Bit-Operationen), die von libFirm aktuell nicht unterstützt werden. Für beide Varianten wurde die Anzahl der durchgeführten lokalen Optimierungen bei der Übersetzung der in Tabelle 5.3 aufgelisteten Programme der SPEC CINT2000 und CINT2006 bestimmt. Die zweite Spalte der Tabelle gibt die Anzahl der von libFirm durchgeführten lokalen Optimierungen an. Die nächsten Spalten beziehen sich auf die erweiterte Variante libFirm + Optgen. Der Reihe nach aufgelistet sind, die Gesamtanzahl der durchgeführten lokalen Optimierungen, der Anteil der angewendeten Optgen-Regeln, die Differenz der angewandten Optimierungen zu libFirm, sowie eine Aufschlüsselung der verwendeten Optgen-Regeln und deren Häufigkeit.

Die während der Übersetzung der Testprogramme angewandten Optgen-Regeln sind in Tabelle 5.4 aufgeführt. Von den 46 implementierten Optgen-Regeln können 14 Regeln in der SPEC in der Summe 830 mal angewendet werden. Da nicht alle Regeln implementiert wurden, besteht die Möglichkeit, dass noch weitere von libFirm nicht betrachtete Regeln existieren, die in der SPEC anwendbar sind. Weiterhin zeigt die Tabelle, welche der neuen Regeln, die im Rahmen der Übersetzung durch libFirm + Optgen in der SPEC anwendbar sind, von den beiden Übersetzern GCC 4.6.3 [10] und Clang 3.0.6 [6] unterstützt (✓) oder nicht unterstützt (×) werden. Bei der in Regel Nr. 10 verwende-

Testprogramm	libFirm	libFirm + Optgen			
		gesamt	Optgen-Regeln	Δ libFirm	Regel-Nr. [Häufigkeit]
164.gzip	1 389	1 391	2	2	1[1], 2[1]
175.vpr	2 850	2 853	3	3	1[3]
176.gcc	60 317	60 332	19	15	1[10], 3[4], 4[2], 5[1], 9[1], 10[1]
181.mcf	368	368	0	0	
186.crafty	7 804	7 851	78	47	3[1], 4[73], 6[2], 7[2]
197.parser	2 507	2 511	6	4	1[3], 3[3]
253.perlbmk	18 684	18 691	7	7	1[3], 2[1], 3[1], 4[1], 8[1]
254.gap	25 081	25 136	130	55	1[28], 2[87], 3[2], 4[13]
255.vortex	9 987	9 988	1	1	1[1]
256.bzip2	1 058	1 060	2	2	1[2]
300.twolf	6 810	6 817	7	7	1[4], 5[3]
400.perlbench	45 186	45 202	16	16	1[4], 2[6], 3[1], 4[3], 8[1], 11[1]
401.bzip2	2 880	2 882	2	2	1[1], 10[1]
403.gcc	169 956	170 535	226	579	1[21], 2[2], 3[3], 4[8], 6[2], 13[177], 14[13]
429.mcf	457	459	2	2	5[2]
445.gobmk	26 113	26 099	15	-14	2[2], 3[2], 4[6], 11[5]
456.hmmer	9 009	9 008	2	-1	4[1], 12[1]
458.sjeng	5 099	5 113	8	14	1[1], 3[4], 4[3],
464.h264ref	22 725	22 976	304	251	1[259], 2[16], 4[15], 3[1], 10[13]
Summe:	418 280	419 272	830	992	

Tabelle 5.3: Anzahl angewendeter lokaler Optimierung durch libFirm und durch libFirm mit zusätzlichen Optgen Regeln (libFirm + Optgen) in der SPEC CINT2000 und CINT2006, Auflistung der angewandten Optgen-Regeln.

ten symbolischen Konstante $sc_0[and\_set : sc1]$ handelt es sich um eine eingeschränkte symbolische Konstante, die von sc_1 abhängig ist. sc_0 darf nur Werte annehmen, die nur an höherwertigen Bit-Stellen, als die höchste Position einer Eins in sc_1, Einsen besitzen (z. B. $sc_1 \mapsto 00001\_\_\_ \Rightarrow sc_0 \mapsto \_\_\_\_0000$).

Um herauszufinden, ob die angewandten Optgen-Regeln in ihrer Summe Auswirkungen auf die Laufzeiten der übersetzten Programme haben, wurden zunächst die Laufzeiten der durch libFirm erzeugten Programme, mit den Laufzeiten der durch libFirm + Optgen erzeugten Programme verglichen. Dies ergab jedoch keine verwertbaren Ergebnisse, da sich die Laufzeitdifferenzen innerhalb der Messschwankungen bewegen und somit keine definitive Aussage über eine Laufzeitänderung möglich ist. Um dennoch eine Aussage über die Auswirkungen der zusätzlichen Regeln geben zu können wurde statt der Laufzeit die Anzahl der ausgeführten Instruktionen pro Testprogramm ermittelt. Die Ergebnisse sind in Tabelle 5.5 dargestellt.

Nr.	Regel			CINT2000	CINT2006	GCC	Clang
1	$(x + x) + (y + y)$	$\rightarrow$	$(x + y) + (x + y)$	55	286	✓	✓
2	$(x + x) - (y + y)$	$\rightarrow$	$(x - y) + (x - y)$	89	26	✓	×
3	$(sc_0 - x) + sc_1$	$\rightarrow$	$(sc_0 + sc_1) - x$	11	11	✓	✓
4	$-(x + sc_0)$	$\rightarrow$	$-sc_0 - x$	89	36	✓	✓
5	$-((x - y) + z)$	$\rightarrow$	$y - (x + z)$	4	2	✓	×
6	$\sim ((\sim y) \mid x)$	$\rightarrow$	$(\sim x)$ & y	2	2	×	✓
7	$(x \mid y) \mid (x \hat{} z)$	$\rightarrow$	$(x \mid y) \mid z$	2	0	×	×
8	$(x - z) + (z - y)$	$\rightarrow$	$x - y$	1	1	✓	✓
9	$(x$ & $y) \mid y$	$\rightarrow$	y	1	0	✓	✓
10	$(x + sc_0[and\_set : sc_1])$ & sc_1	$\rightarrow$	x & $sc1$	1	14	✓	✓
11	$(x + z) + (y - z)$	$\rightarrow$	$x + y$	0	6	✓	✓
12	$x - ((x - y) + z)$	$\rightarrow$	$y - z$	0	1	✓	✓
13	$\sim (sc_0 + x)$	$\rightarrow$	$(\sim sc_0) - x$	0	177	×	✓
14	$\sim (sc_0 - x)$	$\rightarrow$	$x + (\sim sc_0)$	0	13	×	✓
Summe				255	575		

Tabelle 5.4: Übersicht über angewandte, von OPTGEN erzeugte, Optimierungsregeln in der SPEC CINT2000 und CINT2006 während der Übersetzung durch libFIRM + OPTGEN, sowie die Unterstützung (✓) der jeweiligen Regel durch GCC 4.6.3 und Clang 3.0.6.

Auffällig ist vor allem das Testprogramm 429.mcf aus der SPEC CINT2006. Obwohl nur zwei OPTGEN-Regeln zusätzlich angewandt wurde, erhöht sich die Anzahl der ausgeführten Instruktionen mit +7% deutlich. Bei den beiden angewendeten Optimierungsregeln handelt es sich um die Regel Nr. 5 (siehe Tabelle 5.4). Bei der Untersuchung des erzeugten Assembler-Codes konnte festgestellt werden, dass aufgrund von Mehrfachverwendern zwar die Negation, wie sie im linken Teil der Regel angegeben ist, eingespart werden konnte, dafür allerdings zusätzlich eine Addition $(x + z)$ und Subtraktion $(y - (x + z))$ eingefügt wurde. Die zuvor bestehende Subtraktion $(x - y)$ und Addition $(x - y) + z$ wurden aufgrund weiterer Verwender nicht, wie von der Regel vorgesehen, entfernt. Dies hat zur Folge, dass zwar eine Instruktion eingespart werden konnte, aber zwei weitere hinzugefügt wurden. In Summe wurde also eine Instruktion ergänzt. Die Vermutung besteht, dass sich solche Probleme mit Mehrfachverwendern durch eine entsprechende Überprüfung bei der Anwendung der Optimierungsregel umgehen lassen. Allerdings muss die Anwendung einer Regel trotz Mehrfachverwendern nicht zwangsläufig zum Nachteil sein. Es besteht ebenso die Möglichkeit, dass sich eben durch die Anwendung weitere Optimierungsmöglichkeiten ergeben, mit denen sich ein noch besseres Endergebnis erreichen lässt (vgl. auch Abschnitt 4.3.1).

Entgegen diesem negativen Fall, zeigt die Tabelle auch Testprogramme, in denen durch zusätzliche OPTGEN-Regeln eine Verringerung der ausgeführten Instruktionen erreicht werden konnte. Das deutlichste Beispiel hierfür ist das Testprogramm 197.parser aus der SPEC CINT2000 mit einer Reduktion der Instruktionsanzahl um 1,69%.

Testprogramm	libFirm	libFirm + Optgen	Δ (absolut)	Δ (relativ)
164.gzip	62 347 626 366	62 347 429 962	−196 404	-0,000315%
175.vpr	99 955 218 064	99 955 220 031	+1 967	+0,000001968%
176.gcc	29 795 244 112	29 791 431 886	−3 812 226	-0,012794747%
181.mcf	47 797 407 693	47 797 407 693	0	0%
186.crafty	185 831 710 074	185 912 361 663	+80 651 589	+0,043400337%
197.parser	308 017 902 184	302 821 746 944	−5 196 155 240	-1,686965336%
253.perlbmk	53 759 937 292	53 766 679 989	+6 742 697	+0,012542234%
254.gap	220 775 302 143	220 653 059 866	−122 242 277	-0,055369544%
255.vortex	107 653 832 582	107 653 832 578	−4	-0,000000004%
256.bzip2	92 157 935 485	91 951 785 028	−206 150 457	-0,223692573%
300.twolf	293 181 139 036	294 571 111 723	+1 389 972 687	+0,474100309%
400.perlbench	872 343 315 088	867 091 925 262	−5 251 389 826	-0,601986596%
401.bzip	402 262 917 260	402 339 572 836	+76 655 576	+0,019056088%
403.gcc	155 542 530 329	155 511 735 674	−30 794 655	-0,019798222%
429.mcf	325 872 347 604	348 602 311 397	+22 729 963 793	+6,975112789%
445.gobmk	349 210 157 205	349 388 912 294	+178 755 089	+0,051188399%
456.hmmer	1 507 704 655 829	1 507 704 574 772	−81 057	-0,000005376%
458.sjeng	2 325 192 637 818	2 325 934 531 367	+741 893 549	+0,031906756%
464.h264ref	1 470 209 673 282	1 461 291 882 409	−8 917 790 873	-0,606565923%
Durchschnitt			288 211 785	+0,23156924%

Tabelle 5.5: Vergleich der ausgeführten Instruktionen für jedes Testprogramm, sowie die absoluten und relativen Änderungen durch Erweiterung von libFirm um zusätzliche Optgen-Regeln.

Zusammengefasst belegen die in der Tabelle 5.5 aufgeführten Ergebnisse, dass mittels Optgen auch Regeln gefunden werden, die sich in der Praxis, wenn auch nicht immer, gewinnbringend einsetzen lassen.

5.2.3 Nicht-unterstützte Optimierungsregeln

Wie im vorherigen Abschnitt 5.2.2 bereits gezeigt wurde, existieren sowohl für libFirm, als auch GCC und Clang Optimierungsregeln, die aktuell nicht unterstützt werden. Durch eine stichprobenhafte Überprüfung weiterer, durch Optgen erzeugter, Optimierungsregeln konnten noch andere Regeln entdeckt werden, die von dem einen oder anderen Übersetzer nicht berücksichtigt werden. Abbildungen 5.1, und 5.2 zeigen zwei dieser Regeln. Bei der Überprüfung, ob eine Optimierungsregel von einem der Übersetzer beherrscht wird, fiel auf, dass vorwiegend Regeln die sich auf Konstanten beziehen, diejenigen sind, die für die Übersetzer noch unbekannt sind. Also genau die Fälle, auf denen ein Hauptaugenmerk dieser Diplomarbeit liegt. Mit Optgen existiert nun ein Verfahren, mit dem auch solche Fälle zukünftig aufgedeckt und somit berücksichtigt werden können.

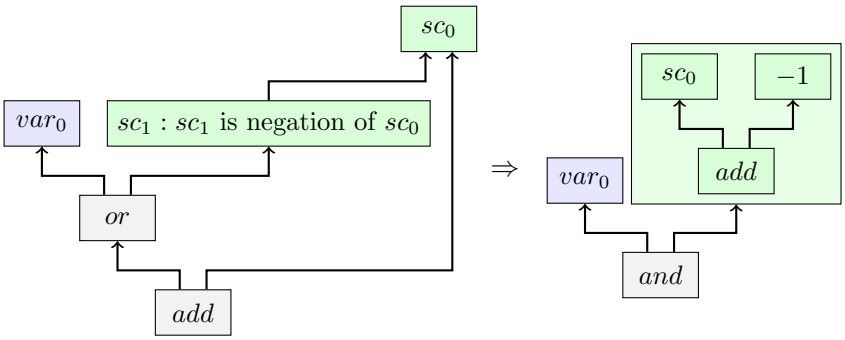


Abbildung 5.1: Optimierungsregel mit eingeschränkter symbolischer Konstante sc_1. sc_1 muss dem negierten Wert von sc_0 entsprechen.

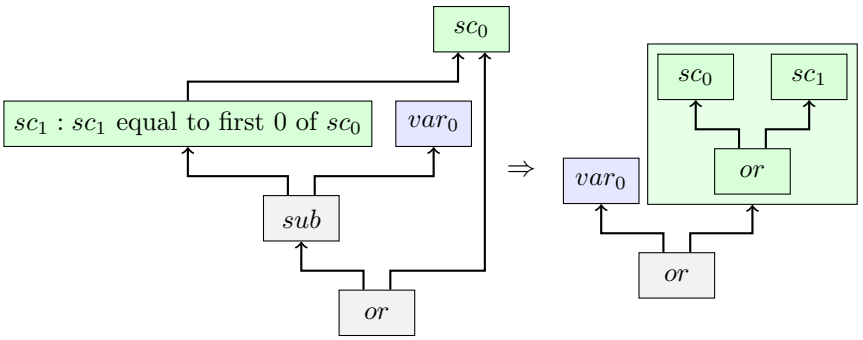


Abbildung 5.2: Optimierungsregel mit eingeschränkter symbolischer Konstante sc_1. sc_1 muss von der niederwertigsten Stelle bis zur ersten Stelle mit einer 0 in sc_0 mit sc_0 übereinstimmen (z. B. $sc_0 \mapsto 11110111 \Rightarrow sc_1 \mapsto \_\_\_\_0111$).

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Das in dieser Arbeit vorgestellte Verfahren OPTGEN ermöglicht die Generierung von Ersetzungsregeln zur lokalen Optimierung. Mittels eines Beweisverfahrens wird die Korrektheit dieser Regeln zudem verifiziert. Als Ergebnis liefert das Verfahren eine Regelmenge, die eine Transformation von Codesequenzen in eine, bezüglich des in dieser Arbeit gewählten Kostenmodells, optimale Darstellung ermöglicht.

Ein wichtiger Aspekt dieser Arbeit besteht in der Berücksichtigung aller Konstanten. Hierdurch wird erreicht, dass sämtliche möglichen Ausdrücke bzw. Muster bei Generierung der Optimierungsregeln berücksichtigt werden. Durch die Erweiterung des Verfahrens um symbolische, komplexe symbolische und eingeschränkte symbolische Konstanten wird eine Möglichkeit zur Verfügung gestellt, mit der die Vielzahl der sich aus den Konstanten ergebenden Muster beherrschbar wird.

Zur Beschleunigung des Verfahrens werden Ansätze vorgestellt, die eine weitere Reduktion der zu untersuchenden Ausdrücke bzw. Muster ermöglichen. Hierzu gehört die Anwendung bereits bekannter Regeln, Normalisierungsverfahren, sowie die zuvor genannte Aggregation von Regeln mit Konstanten zu allgemeineren Regeln. Weiterhin werden Probleme und Einschränkungen, die sich aus den genannten Ansätzen ergeben aufgezeigt.

Eine weitere Beschleunigung des Verfahren wird durch die Verwendung von Testvektoren (Zeugen) erreicht, die sich aus fehlgeschlagenen Äquivalenzbeweisen zwischen zwei Mustern extrahieren lassen. Diese werden genutzt um eine niedrigere Fehlentscheidungsrate bei der Vorauswahl von semantisch äquivalenten Kandidaten zu einem Muster zu erzielen.

Durch die Implementierung dieses Verfahrens und einer anschließenden Evaluierung wird die Durchführbarkeit, insbesondere im Bezug auf die Berücksichtigung aller Konstanten bewiesen. Zudem zeigt die Evaluierung, dass nicht nur in Optimierungen gängige Konstanten wie 0, 1 oder -1 von Interesse sein können, sondern dass auch Optimierungsregeln existieren, die sich auf spezielle Konstanten, Mengen von Konstanten mit gleichen Eigenschaften oder alle Konstanten beziehen. Hierzu sind vor allem Abhängigkeiten zwischen den Konstanten innerhalb eines Musters von Bedeutung.

6.2 Ausblick

6.2.1 Parallelisierung

Bisher arbeitet der Generierungsprozess sequentiell. Nacheinander werden die Arbeitsschritte Mustererzeugung, Anwendbarkeit einer Regel prüfen, Vorabtest und Erfüllbarkeitstest ausgeführt. In einer Parallelisierung steckt das Potential, die Gesamtlaufzeit zu verkürzen und so eine höhere durchführbare Kostenstufe als drei oder mehr zu ermöglichen. Für eine Parallelisierung sind zwei Ansatzpunkte denkbar. Einer dieser Ansatzpunkte stellt der Erfüllbarkeitstest dar, der aufgrund seines hohen Anteils an der Gesamtlaufzeit eines Generierungsvorgangs (vgl. Kapitel 5) ein hohes Sparpotenzial besitzt. Hier wäre entweder eine parallele Überprüfung mehrere Kandidaten denkbar oder die Nutzung eines parallel arbeitenden SMT-Solvers um so die Überprüfung eines einzelnen Kandidaten zu beschleunigen. Auch der Vorabtest lässt sich parallelisieren, indem entweder mehrere Muster gleichzeitig überprüft werden oder die Überprüfung eines einzelnen Musters parallelisiert wird. Der andere Ansatzpunkt betrifft die komplette sequentielle Abarbeitung der einzelnen Arbeitsschritte für jedes neue Muster. Dieses Konzept lässt sich zumindest partiell parallelisieren, so dass mehrere Muster gleichzeitig erzeugt und analysiert werden können. Eine parallele Mustererzeugung innerhalb einer Kostenstufe ist immer unproblematisch, da zur Erzeugung neuer Muster nur Muster mit geringeren Kosten, also aus einer vorherigen, abgeschlossenen Kostenstufe genutzt werden. Etwas problematischer gestaltet sich die Parallelisierung der Musteranalyse. Prinzipiell müssten bei der Suche nach einem semantisch äquivalenten Muster alle optimalen Muster berücksichtigt werden. Durch die, wie in Abschnitt 4.3.2 beschriebene Zuordnung eines Musters zu einer Äquivalenzklasse können Muster, die sich in unterschiedlichen Äquivalenzklassen befinden parallel behandelt werden.

6.2.2 Automatische Generierung von Bedingungen

Die Bedingungen für die in Abschnitt 4.5.2 vorgestellten eingeschränkten symbolischen Konstanten wurden bisher von Hand ermittelt und implementiert. Dieses Vorgehen ist sehr zeitaufwendig, da sowohl die Mustermenge, als auch die Menge der erzeugten Regeln durch Konstanten schnell unüberschaubar wird und sich Abhängigkeiten zwischen den Konstanten eines Muster nur schwer erkennen lassen. Um höhere Kostenstufen dennoch beherrschbar zu machen, sind weitere Mechanismen notwendig, um Bedingungen zwischen den Konstanten eines Musters automatisch zu ermitteln. Ein noch zu untersuchender Ansatzpunkt hierfür könnte sein, aus bereits erzeugten Regeln, durch Vergleich und Analyse der zugehörigen Konstanten, Abhängigkeiten zwischen diesen abzuleiten. Diese können wiederum genutzt werden um allgemeinere Regeln zu generieren, welche die konkreten Regeln ersetzen.

6.2.3 Anwendung von Optimierungsregeln

Mit Optgen existiert ein Verfahren, das in der Lage ist lokale Optimierungen automatisch zu generieren. Ein anderes Problem beseht in der Verwendung dieser Regeln. Die Zunahme der ausgeführten Instruktionen in einigen Testprogrammen (vgl. Kapitel 5), insbesondere bei 429.mcf aus der SPEC CINT2006 zeigen, dass es nicht immer von Vorteil ist jede mögliche Optimierungsregel anzuwenden. Die Frage, die sich stellt, ist wann dies der Fall ist. Bei 429.mcf liegt das Problem in der Mehrfachverwendung von Teilausdrücken. Eine mögliche Einschränkung für die Anwendbarkeit von Optimierungsregeln könnte sein, nur Regeln anzuwenden wenn keine Mehrfachverwendung eines von der Regel betrachten Teilausdrucks besteht. Umgekehrt kann aber auch der Fall eintreten, dass zwar im Moment durch die Anwendung einer Regel eine Verschlechterung vorliegt, diese aber weitere Optimierungen und in der Summe ein besseres Ergebnis ermöglicht.

Literaturverzeichnis

[1] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2. Prentice Hall, 2006. – ISBN 0321486811

[2] Bagwell, John T. Jr.: Local optimizations. In: *Proceedings of a symposium on Compiler optimization*. New York, NY, USA : ACM, 1970, 52–66

[3] Bansal, Sorav ; Aiken, Alex: Automatic generation of peephole superoptimizers. In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA : ACM, 2006 (ASPLOS-XII). – ISBN 1–59593–451–0, 394–403

[4] Barrett, Clark ; Stump, Aaron ; Tinelli, Cesare: The SMT-LIB standard: Version 2.0. In: *SMT Workshop*, 2010

[5] Biere, A. ; Biere, A. ; Heule, M. ; Maaren, H. van ; Walsh, T.: *Handbook of Satisfiability: Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands : IOS Press, 2009. – ISBN 1586039296, 978–1586039295

[6] Clang: *Clang: a C language family frontend for LLVM*. `http://clang.llvm.org`. Version: Juli 2012

[7] Cook, Stephen A.: The complexity of theorem-proving procedures. In: *Proceedings of the third annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1971 (STOC '71), 151–158

[8] De Moura, Leonardo ; Bjørner, Nikolaj: Z3: an efficient SMT solver. In: *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*. Berlin, Heidelberg : Springer-Verlag, 2008 (TACAS'08/ETAPS'08). – ISBN 3–540–78799–2, 978–3–540–78799–0, 337–340

[9] De Moura, Leonardo ; Bjørner, Nikolaj: Satisfiability modulo theories: introduction and applications. In: *Commun. ACM* 54 (2011), September, 69–77. `http://dx.doi.org/10.1145/1995376.1995394`. – DOI 10.1145/1995376.1995394. – ISSN 0001–0782

[10] Gough, Brian J. ; Stallman, Richard M.: *An Introduction to GCC*. Network Theory Ltd., 2004. – ISBN 0954161793

[11] Joshi, Rajeev ; Nelson, Greg ; Randall, Keith: Denali: a goal-directed superoptimizer. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. New York, NY, USA : ACM, 2002 (PLDI '02). – ISBN 1–58113–463–0, 304–314

[12] Levin, Leonid: Universal sorting problems. In: *Problems of Information Transmission* (1973), Nr. 9, S. 265–266

[13] Lindenmaier, Götz: libFIRM – a Library for Compiler Optimization Research Implementing FIRM. Version: Sep 2002. `http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps`. Universität Karlsruhe, Fakultät für Informatik, Sep 2002 (2002-5). – Forschungsbericht. – 75 S.

[14] Massalin, Henry: Superoptimizer: a look at the smallest program. In: *Proceedings of the second international conference on Architectual support for programming languages and operating systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1987 (ASPLOS-II). – ISBN 0–8186–0805–6, 122–126

[15] Muchnick, Steven S.: *Advanced compiler design and implementation*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. – ISBN 1–55860–320–4

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