Karlsruher Institut für Technologie

Institut für Programmstrukturen und Datenorganisation (IPD)

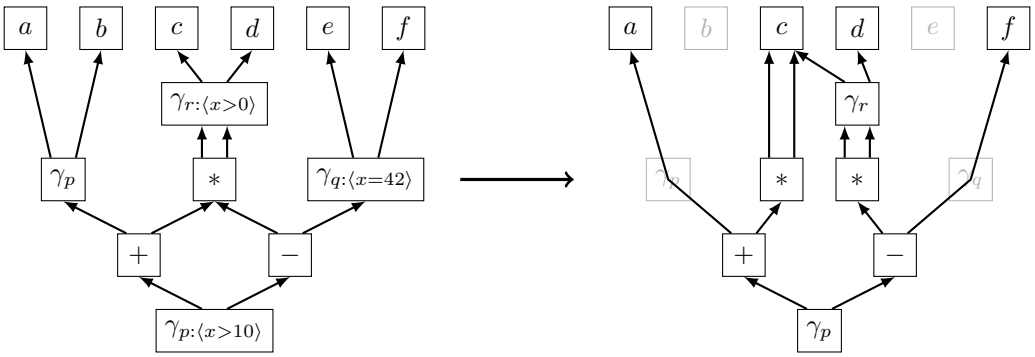
Lehrstuhl Prof. Dr.-Ing. Snelting

Optimierung einer funktionalen und referentiell transparenten Zwischendarstellung

Diplomarbeit von

Julian Oppermann

an der Fakultät für Informatik



Gutachter: Prof. Dr.-Ing. Gregor Snelting

Betreuender Mitarbeiter: Dipl.-Inform. Matthias Braun

Bearbeitungszeit: 1. Februar 2012 – 31. Juli 2012

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

Ort, Datum Unterschrift

Kurzfassung

Die funktionale und referentiell transparente Zwischendarstellung vFirm ist eine vielversprechende Umgebung für die Durchführung von architekturunabhängigen Optimierungen, da sich die Programmrepräsentation von vielen der im Eingabeprogramm gegebenen Abhängigkeiten löst und eine abstrakte Sicht auf die eigentliche Berechnung gibt.
In dieser Arbeit wird eine Optimierungsphase entworfen, um die Formulierung von tradionellen Optimierungen auf dieser Zwischendarstellung zu evaluieren. Zur Bewertung der Profitabilität der einzelnen Transformationen wird ein Kostenmodell vorgestellt, dass die Argumentation über die Anzahl der Berechnungen und die Verzweigungs- und Schleifenstruktur erlaubt.
Die Auswertung zeigt, dass das Kostenmaß sinnvoll ist und die formulierten Transformationen in einer Vielzahl von Funktionen Anwendung finden und dort die Kosten verbessern.

Inhaltsverzeichnis

1. Einleitung

Moderne Compiler sind immer auch optimierende Compiler. Das bedeutet, dass sie neben der reinen Übersetzung des Eingabeprogramms auch versuchen, dessen Laufzeit, den Speicherbedarf und den Energieverbrauch zu reduzieren.

Ein wichtiger Teil dieser Optimierung findet üblicherweise auf einer Zwischendarstellung statt, die nach Abschluss der Analysephase und vor Beginn der Codegenerierung aufgebaut wird. Auf der Zwischendarstellung werden architekturunabhängige Optimierungen durchgeführt. Es handelt sich um semantikerhaltende Transformationen der Darstellung, die die Berechnungen im Programm vereinfachen, reduzieren oder umstrukturieren. Diese Optimierungen profitieren von einer möglichst abstrakten Betrachtungsweise der Berechnungen.

Die vFirm-Darstellung gehört zur Klasse der funktionalen und referentiell transparenten Zwischendarstellungen. Sie zeichnet sich dadurch aus, dass die Ordnung der Operationen nur durch Datenabhängigkeiten festgelegt wird. Nicht-essentielle Kontrollabhängigkeiten sind nicht kodiert.

Der Vorteil dieser Darstellungsform ist, dass man eine sehr klare Sicht auf die zugrunde liegende Funktion einer Berechnung erhält. In dieser Arbeit wird anhand der folgenden Fragestellungen überprüft, wie gut die Darstellung für die Formulierung von Optimierungen zur Verringerung der Laufzeit geeignet ist.

Wie kann man die Laufzeit abschätzen? Das hohe Abstraktionsniveau der Darstellung bedeutet, dass die Laufzeit des fertig übersetzten Programms von vielen Faktoren abhängig ist, die zum Zeitpunkt der Optimierung unbekannt sind. Trotzdem benötigt man Kriterien, nach denen die Darstellung transformiert werden soll, um eine Verbesserung der Laufzeit zu erreichen.
Dazu wird ein einfaches Kostenmaß entwickelt, das neben der Anzahl der Berechnungen auch die Verzweigungs- und Schleifenstruktur des Programms berücksichtigt. Es wird überprüft, ob sich eine Verbesserung des Kostenmaßes auch in einer messbaren Verkürzung der Laufzeit von Testprogrammen widerspiegelt.

Wie kann man das Programm schneller machen? In dieser Arbeit wird eine Optimierungsphase entworfen, die die Effekte vieler traditioneller Optimierungen abdeckt. Dafür werden Transformationen gesucht, die die Kosten des Programms verringern.
Die Grundlage bilden auf verwandten Darstellungen beschriebene Verfahren. Sie werden angepasst und gemäß des Kostenmodells bewertet.
Dann wird eine auf Datenflussanalyse basierende Steuerflussoptimierung, ein Verfahren zur Auswertung von Schleifen und eine Transformation zur Umstrukturierung von Summen vorgestellt.
Die entwickelte Optimierungsphase ist eine geeignete Grundlage, um die Anwendbarkeit und die positiven Effekte der Transformationen auf Programmcode aus der Praxis zu zeigen. Der Umfang dieser Untersuchung geht über die bislang in der Literatur dokumentierten Ansätze hinaus.

2. Grundlagen

2.1. Notationen

2.1.1. Graphen

In dieser Arbeit werden Programme als gerichtete Graphen $G = (V, E)$ mit einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$ dargestellt.

Die Knoten $v \in V$ des Graphen sind attributiert. Mit $v.\mathsf{abc}$ bezeichnen wir ein Attribut abc. Jedem Knoten ist eine Operation $v.\mathsf{op}$ zugeordnet. Wir bezeichnen arithmetisch-logische Operationen mit ihren üblichen mathematischen Symbolen, beispielsweise schreiben wir für eine Addition $v.\mathsf{op} = +$. Alle weiteren Attribute werden im Folgenden bei Bedarf eingeführt.

Ein Knoten w ist ein Vorgänger von v, wenn eine Kante $v \to w \in E$ existiert. Je nach Kontext verwenden wir auch $a, b, \ldots$, um die Vorgänger zu bezeichnen. Die Vorgänger sind mit geordneten und benannten Eingängen verbunden und werden als Argumente für die Operation des Knotens verstanden. Wir notieren $v.\mathtt{xyz}$ für den Vorgänger am $\mathtt{xyz}$-Eingang. Textuell stellen wir Knoten auch als $v = v.\mathsf{op}(w_1, w_2)$ oder als $v = w_1\ v.\mathsf{op}\ w_2$ dar, wenn für die Operation die Infixschreibweise üblich ist.

Ein Knoten u ist ein Verwender von v, wenn eine Kante $u \to v \in E$ existiert. Die Menge der Verwender wird von v ausgehend nicht weiter unterschieden; uns interessiert aber die Anzahl der Verwender, die wir mit $\alpha(v)$ bezeichnen.

Rechts des Knotens ist ein Bezeichner angegeben, unter dem wir den Knoten im Text referenzieren. Fehlt er und ist die Operation im dargestellten Teilgraph eindeutig, identifizieren wir den Knoten mit seiner Operation.

Abbildung 2.1 fasst alle Notationen zusammen. Der Knoten v ist eine Addition mit den Argumenten a und b, wir notieren also $v.\mathsf{op} = +$, $v.\mathtt{left} = a$, $v.\mathtt{right} = b$ sowie $\alpha(v) = 1$, weil der Knoten $*$ der einzige Verwender ist.

Ein Pfad von v_1 nach v_n ist eine Kantenfolge $v_1 \to v_2, v_2 \to v_3, \ldots, v_{n-1} \to v_n$ mit $v_i \in V$ und $v_i \to v_{i+1} \in E$. Wir notieren kürzer $v_1 \to^* v_n$.

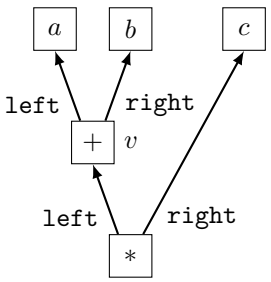


Abbildung 2.1.: Graphdarstellung

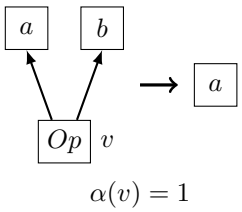


Abbildung 2.2.: Musterersetzung

2.1.2. Musterersetzung

Wir stellen Musterersetzungsregeln wie in Abbildung 2.2 dar. Beide Seiten haben einen eindeutigen Wurzelknoten. Unter dem Muster ist optional eine Bedingung angegeben. Passt die linke Seite des Musters auf einen Teilgraph, und ist die Bedingung erfüllt, wird der Wurzelknoten der linken Seite durch den Wurzelknoten der rechten Seite ersetzt. Knoten, die sich auf beiden Seiten befinden, bleiben unverändert. Knoten, die ausschließlich auf der linken Seite vorkommen, werden entfernt, wohingegen Knoten, die ausschließlich auf der rechten Seite auftreten, neu erstellt werden.

2.1.3. Konstanten

Konstanten setzen wir als **k**. Dabei unterscheiden wir nicht zwischen mathematischen Konstanten und Knoten, die eine konstante Operation modellieren.

Die booleschen Wahrheitswerte “wahr” und “falsch” stellen wir ebenfalls als Konstanten dar; wir schreiben **t** beziehungsweise **f**.

2.2. Datenflussanalyse in monotonen Rahmenwerken

Datenflussanalyse ist eine Form der Programmanalyse, die dazu dient, eine konservative Approximation der Menge von Werten zu berechnen, die an einer bestimmten Programmstelle zur Laufzeit auftreten können.

Es handelt sich um eine Abschätzung, weil eine Programmanalyse unter anderem aufgrund des Unwissens über Eingaben und Argumente zur Laufzeit und des Terminationsverhaltens nicht präzise sein kann. Diese Abschätzung ist konservativ, wenn das Analyseergebnis garantiert alle Werte enthält, die tatsächlich auftreten. Daneben kann es aber auch Werte enthalten, die zur Laufzeit nicht möglich sind.

In dieser Arbeit werden Datenflussanalysen in monotonen Rahmenwerken nach dem Schema von Nielson und Nielson [NNH05] konstruiert.

Die grundlegende Idee der Datenflussanalyse ist, dass jeder Anweisung der Zwischendarstellung ein eingehendes und ein ausgehendes Datenflussattribut zugeordnet wird. Die Anweisungen stehen untereinander durch einen Fluss in Verbindung, der die Abhängigkeiten zwischen verschiedenen Anweisungen spezifiziert. Das eingehende Attribut berechnet sich durch die Kombination der Attribute der Flussvorgänger, das ausgehende Attribut durch die Anwendung einer monotonen Transferfunktion auf das eingehende Attribut.

Es entsteht ein Gleichungssystem für die Datenflussattribute, dass iterativ gelöst werden kann und dessen Fixpunkt das Analyseergebnis darstellt.

Die Datenflussattribute sind Elemente eines vollständigen Verbands. Dann lässt sich der Knaster-Tarski-Fixpunktsatz [Dav08] anwenden; zusammen mit den monotonen Transferfunktionen garantiert dieser die Existenz des Fixpunkts.

2.2.1. Verbände

Die Definitionen und Sätze in diesem Unterabschnitt basieren auf [Dav08].

Definition 1. Sei P eine Menge, $S \subseteq P$ eine Teilmenge von P und $\sqsubseteq$ eine Halbordnung auf P, das heißt eine reflexive, antisymmetrische und transitive Relation.

Ein Element $x \in P$ ist eine *obere Schranke* von S, wenn $\forall s \in S : s \sqsubseteq x$ gilt. Analog ist $x \in P$ eine *untere Schranke* von S, wenn $\forall s \in S : s \sqsupseteq x$ gilt.
Ein $x \in P$ mit den Eigenschaften

1. x ist eine obere Schranke von S und 2. für alle oberen Schranken y von S gilt $x \sqsubseteq y$	1. x ist eine untere Schranke von S und 2. für alle unteren Schranken y von S gilt $x \sqsupseteq y$
heißt *kleinste obere Schranke* oder *Supremum* von S, bezeichnet durch die Notation $\bigsqcup S$.	heißt *größte untere Schranke* oder *Infimum* von S, bezeichnet durch die Notation $\bigsqcap S$.

Für $a, b \in P$ und falls Supremum beziehungsweise Infimum existieren, schreibt man kürzer $a \sqcup b$ für $\bigsqcup\{a, b\}$ und $a \sqcap b$ für $\bigsqcap\{a, b\}$.

Definition 2 (Verband)**.** Sei P eine nichtleere Menge und $\sqsubseteq$ eine Halbordnung auf P.

1. Wenn $x \sqcup y$ und $x \sqcap y$ für alle $x, y \in P$ existieren, ist $(P, \sqsubseteq)$ ein *Verband*.
2. Wenn $\bigsqcup S$ und $\bigsqcap S$ für alle $S \subseteq P$ existieren, ist $(P, \sqsubseteq)$ ein *vollständiger Verband*.

Lemma 1 (Produktverband)**.** Seien $(L, \sqsubseteq_L)$ und $(K, \sqsubseteq_K)$ Verbände.
Dann ist $((L, \sqsubseteq_L) \times (K, \sqsubseteq_K), \sqsubseteq)$ mit der komponentenweise Definition von Supremum und Infimum

$$(l_1, k_1) \sqcup (l_2, l_2) = (l_1 \sqcup l_2, k_1 \sqcup k_2)$$
$$(l_1, k_1) \sqcap (l_2, l_2) = (l_1 \sqcap l_2, k_1 \sqcap k_2)$$

ein Verband, dessen Halbordnung $\sqsubseteq$ sich komponentenweise aus $\sqsubseteq_L$ und $\sqsubseteq_K$ definiert.

Wir notieren $l \times k$ für ein Element des Produktverbands.

Lemma 2. Jeder endliche Verband ist vollständig.

Lemma 3 ("Connecting Lemma")**.** Sei $(P, \sqsubseteq)$ ein Verband und $a, b \in P$. Dann gilt

$$a \sqsubseteq b \quad \Leftrightarrow \quad a \sqcup b = b \quad \Leftrightarrow \quad a \sqcap b = a$$

Definition 3 ("Ascending Chain Condition")**.** Sei P eine Menge und $\sqsubseteq$ eine Halbordnung auf P. P erfüllt die Ascending Chain Condition, wenn es für jede Folge von Elementen $x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \ldots$ mit $x_i \in P$ ein $k \in N$ gibt, so dass $x_k = x_{k+1} = \ldots$ ist.

Lemma 4. Endliche Verbände erfüllen die Ascending Chain Condition.

Die Beweise dieser Lemmata findet man in [Dav08].

Lemma 5. Sei $(P, \sqsubseteq)$ ein Verband. Dann gilt für alle $a, b, c, d \in P$:

$$a \sqsubseteq b \quad \Longrightarrow \quad a \sqcup c \sqsubseteq b \sqcup c \quad \text{und} \quad a \sqcap c \sqsubseteq b \sqcap c$$

Beweis. Mithilfe des Connecting Lemmas und der Assoziativität, Kommutativität und Reflexivität der Supremumsoperation gilt

$$\begin{aligned}
& b \sqcup c = b \sqcup c \\
\Rightarrow \quad & (a \sqcup b) \sqcup c = b \sqcup c \\
\Rightarrow \quad & a \sqcup b \sqcup c \sqcup c = b \sqcup c \sqcup c \\
\Rightarrow \quad & (a \sqcup c) \sqcup (b \sqcup c) = b \sqcup c \\
\Rightarrow \quad & a \sqcup c \sqsubseteq b \sqcup c
\end{aligned}$$

Der Beweis für die Infimumsoperation verläuft analog. □

2.2.2. Monotone Rahmenwerke

Ein *monotones Rahmenwerk* [NNH05] besteht aus

- einem vollständigen Verband $(L, \sqsubseteq)$, der die Ascending Chain Condition erfüllt, mit der Kombinationsoperation[1] $\bigsqcup$, und
- einer Menge $\mathcal{F}$ von monotonen Funktionen $f : L \to L$, die die Identitätsfunktion enthält und unter der Funktionskomposition abgeschlossen ist.

Das Rahmenwerk modelliert also den Typ und die Berechnungsvorschrift der Datenflussattribute.

Zu einer *Instanz* eines solchen Rahmenwerks, das heißt der konkreten Anwendung der Analyse auf ein Programm, gehört neben dem Verband und der Menge der Funktionen zusätzlich

- eine Beschreibung des Flusses,
- eine Kennzeichnung der Randanweisungen durch Zuweisung eines initialen Werts ι, sowie
- eine Zuordnung der Transferfunktionen zu den Anweisungen.

Im Kontext der vFirm-Zwischendarstellung werden Anweisungen mit Knoten $v \in V$ und die Beschreibung des Flusses mit den Kanten $v \to w \in E$ der Zwischendarstellung identifiziert. Die Zuordnung der Transferfunktionen ist pro Knoten beziehungsweise pro Knotentyp gegeben.

[1] Bei der Kombinationsoperation handelt es sich nicht zwingend um die Supremumsoperation

3. Die vFirm-Darstellung

Die vFirm-Darstellung [Lie11] bildet die Grundlage unserer Optimierungsphase. Wir werden sie zunächst anhand einfacher Beispiele vorstellen und anschließend diese Intuition formalisieren.

3.1. vFirm am Beispiel

3.1.1. Ausdrücke

Beginnen wir mit der Funktion in Abbildung 3.1.

Die Bitoperationen sind durch Knoten repräsentiert. Sie berechnen einen Wert, indem sie ihre Operation auf die Werte ihrer Vorgängerknoten anwenden. Die Funktionsparameter a, b sind ebenfalls Knoten, die die beim Aufruf der Funktion übergebenen Werte repräsentieren. Die lokalen Variablen der Funktion tauchen in der Darstellung nicht mehr auf.

In der Graphdarstellung ist außer den Abhängigkeitskanten keine Reihenfolge kodiert. Um den Wert des `Return`-Knotens zu berechnen, können wir beispielsweise zuerst die Negationen, dann die Und-Verknüpfungen und zuletzt die Oder-Verknüpfung berechnen. Ebenso erlaubt ist, zuerst den rechten und anschließend den linken Teilgraph der Oder-Verknüpfung zu berechnen.

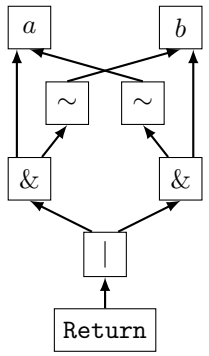
3.1.2. Zustand

Die Funktion in Abbildung 3.2 enthält zwei Funktionsaufrufe, deren Reihenfolge man nicht vertauschen darf, weil dies die beobachtbare Semantik des Programms verändern würde. Zur Modellierung dient ein spezieller Speicherwert *mem*. Jede zustandsbehaftete Operation konsumiert einen Speicherwert und produziert einen neuen. Zu Beginn der Funktion existiert ein initialer Speicherwert. Anschließend darf zu jedem Zeitpunkt stets nur ein Speicherwert lebendig sein. Bevor die Funktion verlassen wird, konsumiert der `Return`-Knoten den Speicherwert. Die Abhängigkeitskanten, die die Verwendung eines Speicherwerts darstellen, sind gestrichelt gezeichnet.

Das Beispiel zeigt ebenfalls die Verwendung von Projektionen (`Proj`-Knoten): Die Funktionsaufrufe produzieren zusätzlich zum Rückgabewert den beschriebenen, neuen Speicherwert. Aus diesem Tupel von Werten extrahiert die Projektion den durch ihren Index gekennzeichneten Wert.

```
1 int xor(int a, int b) {
2   int and1 =  a & ~b;
3   int and2 = ~a &  b;
4   return and1 | and2;
5 }
```

(a) Quelltext

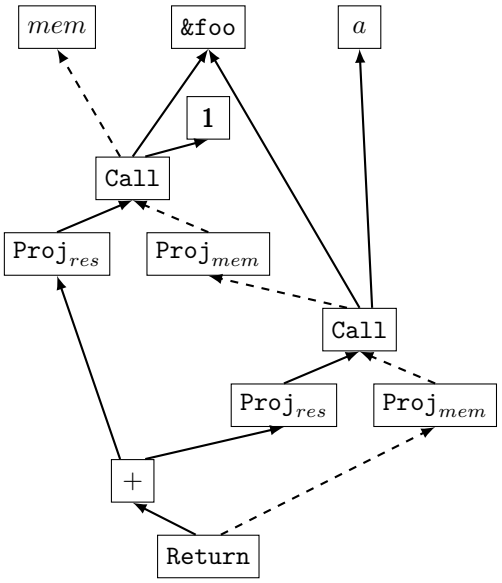


(b) Graph

Abbildung 3.1.: Ausdrücke in vFirm-Darstellung

```
1 int foo(int z);
2
3 int bar(int a) {
4   int x  = foo(1);
5       x += foo(a);
6   return x;
7 }
```

(a) Quelltext

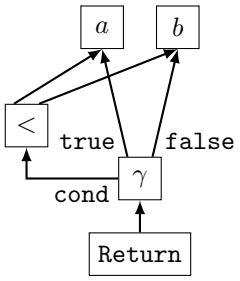


(b) Graph

Abbildung 3.2.: Zustandsbehaftete Operationen

```
1 int min(int a, int b) {
2   int m;
3   if (a < b)
4     m = a;
5   else
6     m = b;
7   return m;
8 }
```

(a) Quelltext

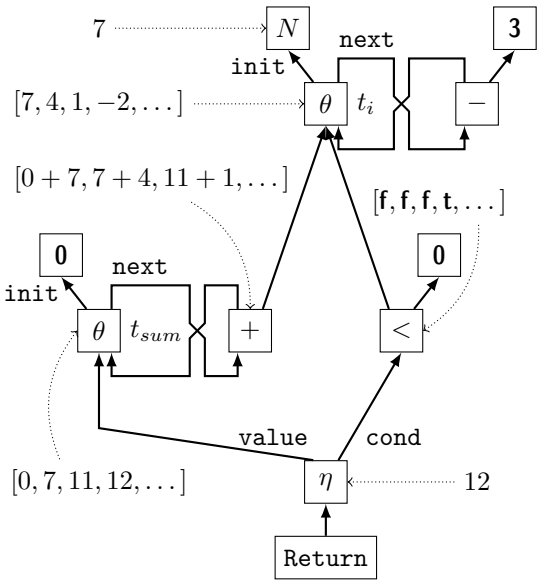


(b) Graph

Abbildung 3.3.: Verzweigungen

```
1 int sum(int N) {
2   int sum = 0, i;
3   for (i = N; i >= 0; i -= 3)
4     sum += i;
5   return sum;
6 }
```

(a) Quelltext



(b) Graph

Abbildung 3.4.: Schleifen. Eingezeichnet sind die produzierten Werte für $N = 7$.

3.1.3. Verzweigungen

Die Funktion in Abbildung 3.3 enthält eine Verzweigung. Im zugehörigen Graph ist sie durch einen γ-Knoten modelliert. Er funktioniert wie ein Umschalter mit den Eingängen `cond`, `true` und `false`. Zuerst wird der Knoten am `cond`-Eingang ausgewertet. Ist das Ergebnis **t**, wird der Wert des Knotens am `true`-Eingang weitergeleitet, andernfalls der Wert des Knotens am `false`-Eingang.

3.1.4. Schleifen

Abschließend betrachten wir die Darstellung von Schleifen anhand des Beispiels in Abbildung 3.4.

In unserem Beispiel ändern sich die Werte der lokalen Variable `sum` und der Iterationsvariable `i` in jeder Iteration der `for`-Schleife. In die Berechnung geht ihr Wert in der vorhergegangenen Iteration ein. Wir stellen diese Werte durch die θ-Knoten t_i und t_{sum} dar. Ein solcher θ-Knoten produziert eine (theoretisch) unendliche Liste von Werten, wobei das erste Listenelement mit dem Wert des Knotens am `init`-Eingang initialisiert wird, und alle weiteren Elemente durch die Auswertung des Knotens am `next`-Eingang bestimmt werden. Der Zugriff auf den Wert aus der vorhergegangenen Iteration wird durch die zyklische Verwendung des θ-Knotens modelliert.

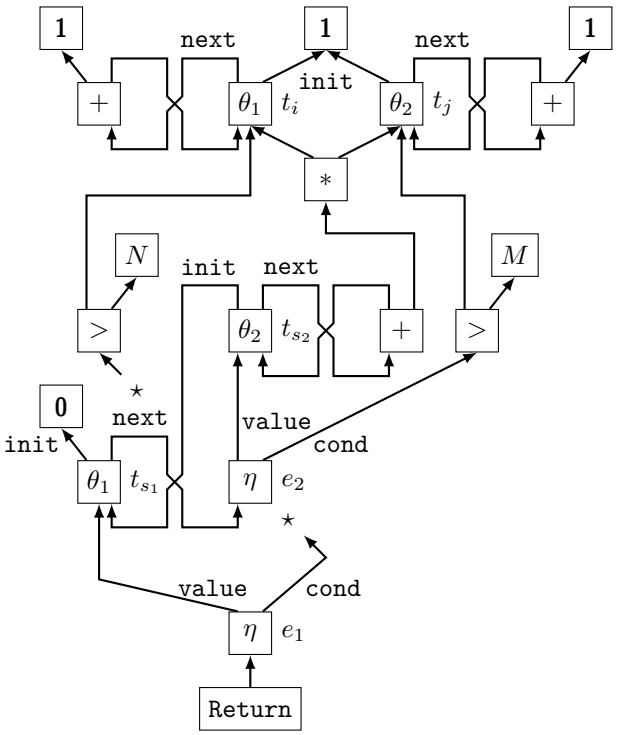
Im Beispiel sind die Wertelisten der Knoten für den Aufruf der Funktion mit $N = 7$ eingezeichnet. Knoten, die einen θ-Knoten verwenden, werden zu Listenoperationen gehoben; sie werden elementweise auf die Liste angewendet. Der Vergleichsknoten produziert also eine Liste von booleschen Werten.

Betrachten wir die Addition. Sie wird ebenfalls zu einer Listenoperation gehoben, im Unterschied zur Vergleichsoperation sind aber beide Argumente Listen. Die Addition verhält sich wie eine Vektoraddition, die Listen von t_i und t_{sum} werden elementweise addiert.

Der Wert von `sum` wird nach Verlassen der Schleife noch verwendet; zur Bestimmung des dann gültigen Werts dient der η-Knoten. Seine Argumente sind eine Liste von Werten am `value`-Eingang, sowie eine Liste von Bedingungen am `cond`-Eingang. Der η-Knoten bestimmt den Index des ersten Elements der `cond`-Liste mit dem Wert **t** und liefert dann das Element mit diesem Index aus der `value`-Liste zurück. Wir können uns die Bedingung als Abbruchbedingung vorstellen, deswegen ist sie die Negation der Bedingung im Kopf der `for`-Schleife.

```
1 int sumprod(int N, int M) {
2   int sum = 0, i, j;
3   for (i = 1; i <= N; i++)
4    for (j = 1; j <= M; j++)
5       sum += i*j;
6   return sum;
7 }
```

(a) Quelltext



(b) Graph

Abbildung 3.5.: Geschachtelte Schleifen.

Geschachtelte Schleifen Die Beispielfunktion in Abbildung 3.5 enthält eine geschachtelte Schleife. Der Wert von `i` ändert sich in jeder Iteration der äußeren Schleife, insgesamt also N-mal. Für eine bestimmte Iteration der äußeren Schleife ändert sich der Wert von `j` in jeder Iteration der inneren Schleife, nämlich M mal. Insgesamt ändert sich `j` $N \cdot M$-mal. Um diese unterschiedlichen Häufigkeiten der Neuberechnung darzustellen, tragen alle θ-Knoten t ein Attribut $t.\mathsf{depth}$, genannt *Schleifentiefe*, das der Anzahl der Verschachtelungsebenen entspricht. In unserem Beispiel wird `i` vom Knoten t_i mit der Schleifentiefe 1 und `j` vom Knoten t_j mit der Schleifentiefe 2 repräsentiert.

Der Wert von `sum` ändert sich ebenfalls mit jeder Iteration der inneren Schleife. Im Gegensatz zu `j` dürfen wir aber den Wert am Ende der inneren Schleife nicht verwerfen, sondern benötigen ihn als initialen Wert für die nächste Iteration (und damit die nächste Instanz der inneren Schleife). Deshalb wird `sum` durch die zwei θ-Knoten t_{s_1}, t_{s_2} dargestellt: t_{s_1} extrahiert mittels e_2 den Wert der Summe nach Termination der inneren Schleife; diesen Wert verwendet die nächste Instanz von t_{s_2} zur Initialisierung.

Die θ-Knoten der Schleifentiefe 2 produzieren Listen von Listen. Die Multiplikation berechnet beispielsweise:

$$[\underbrace{[1,2,3,\dots]}_{i=1},\underbrace{[2,4,6,\dots]}_{i=2},\underbrace{[3,6,9,\dots]}_{i=3},\dots]$$

Der η-Knoten e_2 erzeugt nun eine Liste von Werten, indem er aus jeder der geschachtelten Listen einen Wert selektiert. In unserem Beispiel gilt für $M = 2$:

$$t_{s_2} = [[1,3,\mathbf{6},\dots],[6,8,\mathbf{12},\dots],[12,15,\mathbf{21},\dots],\dots]$$

und e_2 selektiert die fett gedruckten Werte:

$$e_2 = [6,12,21,\dots]$$

Typ	Beschreibung
$\mathbb{I}$	Ganzzahlige Werte im Bereich $[\mathbb{I}_{\min}, \mathbb{I}_{\max}]$
$\mathbb{B}$	Boolesche Werte
$\mathbb{P}$	Adressen
$\mathbb{F}$	Fließkommawerte
$\mathbb{M}$	Speicherwert
$\mathbb{T}$	Tupelwerte
$\mathbb{Q}$	Kurzform für $\mathbb{I} \cup \mathbb{F} \cup \mathbb{P}$
$\mathbb{Q}^+$	Kurzform für $\mathbb{I} \cup \mathbb{F} \cup \mathbb{P} \cup \mathbb{M}$

Tabelle 3.1.: Typen der vFirm-Darstellung

Gruppe	Operation	Signatur	Bemerkung / Darstellung als Knoten
Arithmetik	`Add`	$\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$	Als Knoten: $+$
		$\mathbb{P} \times \mathbb{I} \to \mathbb{P}$	Adressrechnungen
		$\mathbb{I} \times \mathbb{P} \to \mathbb{P}$	
	`Sub`	$\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$	Als Knoten: $-$
		$\mathbb{P} \times \mathbb{I} \to \mathbb{P}$	Adressrechnungen
		$\mathbb{P} \times \mathbb{P} \to \mathbb{I}$	
	`Mul`	$\mathbb{I} \times \mathbb{I} \to \mathbb{I}$	Als Knoten: $*$
		$\mathbb{F} \times \mathbb{F} \to \mathbb{F}$	
Bitoperationen	`And`, `Or`	$\mathbb{I} \times \mathbb{I} \to \mathbb{I}$	Als Knoten: &, |
	`Not`	$\mathbb{I} \to \mathbb{I}$	Als Knoten: $\sim$
Vergleiche	`Cmp`	$\mathbb{Q} \times \mathbb{Q} \to \mathbb{B}$	Relation als v.rel modelliert
			Als Knoten: $<, \leq, =, \neq, \geq, >$
Zustandsbehaftete Operationen	`Load`	$\mathbb{M} \times \mathbb{P} \to \mathbb{T}$	Ergebnis: neuer Zustand und Wert
	`Store`	$\mathbb{M} \times \mathbb{P} \times \mathbb{Q} \to \mathbb{T}$	Ergebnis: neuer Zustand
	`Call`	$\mathbb{M} \times \mathbb{P} \times \mathbb{Q}^n \to \mathbb{T}$	Ergebnis: neuer Zustand und Rückgabewert
Konstanten	`Const`	$\mathbb{Q} \cup \mathbb{B}$	
	`SymConst`	$\mathbb{P}$	Symbolische Konstante
Projektion	`Proj`	$\mathbb{T} \to \mathbb{Q}^+$	Extrahiert Wert aus Tupel
Gatingfunktionen	γ	$\mathbb{B} \times \mathbb{Q}^+ \times \mathbb{Q}^+ \to \mathbb{Q}^+$	Verzweigung
	θ	$\mathbb{Q}^+ \times \mathbb{Q}^+ \to \mathbb{Q}^+$	Schleifenkopf
	η	$\mathbb{Q}^+ \times \mathbb{B} \to \mathbb{Q}^+$	Abbruchbedingung

Tabelle 3.2.: Ausgewählte Operationen

3.2. vFirm formal

In Liebe [Lie11] findet sich folgende Definition der vFirm-Darstellung:

Definition 4 (vFirm)**.** Ein vFirm-Programm ist ein attributierter und gerichteter Graph $G = (V, E, v_s, v_r)$ mit Knotenmenge V, Kantenmenge $E \subseteq V \times V$ und den ausgezeichneten `Start`-Knoten v_s und `Return`-Knoten v_r.

3.2.1. Operationen der Zwischendarstellung

Jedem Knoten v ist eine Operation v.op und ein Typ v.type zugeordnet. Die möglichen Typen sind in Tabelle 3.1 aufgelistet. Tabelle 3.2 zeigt eine Auswahl der wichtigsten Operationen in vFirm. Eine vollständige Beschreibung der Operationen und Typen gibt die Dokumentation der Zwischendarstellung Firm [TLB99], auf der vFirm basiert.

3.2.2. Auswertung

Führen wir zunächst die Wertfunktion für vFirm-Knoten ein.

Definition 5 (value-Funktion (nach [Lie11]))**.** Für einen Knoten v ist $\text{value}_L(v) \in v.\mathsf{type}$ der durch den Knoten berechnete Wert. Da bei Schleifen je nach Kontext verschiedene Werte produziert werden, gibt L die Indizes der involvierten Schleifen an.

Normale Knoten

Definition 6 (Wert von normalen Knoten)**.** Für einen Knoten v mit den Argumenten w_1 bis w_a und $v.\mathsf{op} \notin \{\gamma, \theta, \eta\}$ ist:

$$\text{value}_L(v) = v.\mathsf{op}(\text{value}_L(w_1), ..., \text{value}_L(w_a))$$

Der Wert eines Knotens kann also erst berechnet werden, wenn die Werte seiner Vorgänger ausgewertet wurden. Die Auswertung ist strikt in allen Argumenten. Darüber hinaus spezifiziert die Darstellung keine weitere Reihenfolge der Auswertung. Man kann auch sagen, dass die Auswertung eines Knotens die Auswertung seiner Vorgänger auslöst.

Ein vFirm-Graph wird ausgehend von v_r ausgewertet.

Die Semantik der Darstellung erlaubt es uns prinzipiell, Knoten zu beliebigen Zeitpunkten (wenn ihre Argumente bekannt sind) oder beliebig häufig auszuwerten. Im Falle von zustandsverändernden Operationen würde das aber die beobachtbare Programmsemantik verändern. Um die Korrektheit der Übersetzung zu gewährleisten, muss man eine Auswertungsstrategie verwenden, die dem aus der funktionalen Programmierung bekannten "call-by-need" entspricht. Die Strategie macht aus, dass der Wert eines Knotens genau einmal bei der ersten Auswertung berechnet wird. Anschließend wird der Wert gespeichert und bei allen folgenden Auswertungen des Knotens verwendet [Law07].

Bedingte Auswertung

Definition 7 (γ-Knoten)**.** Für einen Knoten $g = \gamma(\texttt{cond}, \texttt{true}, \texttt{false})$ ist:

$$\text{value}_L(g) = \begin{cases} \text{value}_L(\texttt{true}) & \text{wenn } \text{value}_L(\texttt{cond}) = \mathsf{t} \\ \text{value}_L(\texttt{false}) & \text{sonst} \end{cases}$$

Die Auswertung von γ-Knoten ist nur strikt im ersten Argument `cond`. In Abhängigkeit von dessen Wert wird eines der beiden anderen Argumente ausgewertet; es handelt sich also um eine bedingte Auswertung. Um die Semantik des Programms nicht zu verändern, darf beim Vorhandensein von zustandsbehafteter Operation ausschließlich der ausgewählte Vorgänger ausgewertet werden [Law07].

3.2.3. Schleifen

Schleifentiefe Wir haben im vorherigen Abschnitt gesehen, wie in Schleifen berechnete Werte durch θ-Knoten modelliert werden und beschrieben, dass diese Knoten Listen von Werten, Listen von Listen usw. in Abhängigkeit von der Schleifentiefe produzieren. θ-Knoten haben eine definierte Schleifentiefe, die sich aus dem Verschachtelungsgrad im Programm ergibt. Wir haben aber auch gesehen, dass Operationen, die θ-Knoten verwenden, in der gleichen Häufigkeit neue Werte beziehungsweise Listenelemente produzieren. Allgemein kann man sagen, dass das Argument, das sich am häufigsten ändert, die Frequenz der Neuberechnung eines Knotens bestimmt. η-Knoten wiederum extrahieren einen einzelnen Wert aus einer Liste. Um dies zu fassen, definieren wir die Schleifentiefe für alle Knotentypen.

Definition 8 (Schleifentiefe [Lie11])**.** Für einen Knoten v mit $v.\mathsf{op} \neq \theta$ ist:

$$d = \max_{w : v \to w \in E} \{w.\mathsf{depth}\} \cup \{0\}$$

$$v.\mathsf{depth} = \begin{cases} d - 1 & \text{wenn } v.\mathsf{op} = \eta \\ d & \text{sonst} \end{cases}$$

Wir nutzen nun den Index der Wertfunktion, um die Werte eines Knotens in verschiedenen Iterationen zu unterscheiden. Es handelt sich um einen Vektor $[i_1, i_2, \ldots, i_d]$, bestehend aus einem Iterationszähler für jede Schleifentiefe.

Definition 9 (erweiterte value-Funktion [Lie11])**.** Für einen Knoten v mit der Schleifentiefe $v.\mathsf{depth}$ ist:

$$\text{value}_{[i_1,\ldots,i_d]}(v) = \begin{cases} [\text{value}_{[i_1,\ldots,i_d,j]}(v), j = 0,1,2,\ldots] & \text{wenn } d < v.\mathsf{depth} \\ \text{value}_{[i_1,\ldots,i_{v.\mathsf{depth}}]}(v) & \text{wenn } d > v.\mathsf{depth} \end{cases}$$

Ist die Schleifentiefe $d < v.\mathsf{depth} + 1$, wird die Definition mehrmals angewendet und wir erhalten eine Liste von Listen [Lie11].

Hier sehen wir den Zusammenhang mit der Vorstellung der Wertelisten für einen Knoten v: Diese Betrachtungsweise gilt ausgehenden vom azyklischen Teil des Graphen, also bei Auswertung mit dem leeren Indexvektor $\text{value}_{[]}(v)$.

Im Beispiel in Abbildung 3.5 gilt für die Multiplikation

$$\text{value}_{[]}(*) = [\ \underbrace{[1,2,3,\ldots]}_{i_1=0},\ \underbrace{[2,4,6,\ldots]}_{i_1=1},\ \underbrace{[3,6,9,\ldots]}_{i_1=2},\quad \ldots]$$

($i_2 = 0$, $i_2 = 1$, $i_2 = 2$ bezeichnen die Elemente der inneren Listen.)

$$\begin{aligned} \text{value}_{[0]}(*) &= [1,2,3,\ldots] & \text{value}_{[1]}(*) &= [2,4,6,\ldots] \\ \text{value}_{[0,0]}(*) &= 1 & \text{value}_{[2,1]}(*) &= 6 \\ \text{value}_{[0,1]}(*) &= 2 & \text{value}_{[2,2]}(*) &= 9 \end{aligned}$$

Nun können wir die Wertfunktionen für θ- und η-Knoten definieren.

Definition 10 (θ-Knoten)**.** Für einen Knoten $t = \theta(\texttt{init}, \texttt{next})$ mit der Schleifentiefe $t.\mathsf{depth} = d$ ist:

$$\text{value}_{[i_0,\ldots,i_{d-1},i_d]}(t) = \begin{cases} \text{value}_{[i_0,\ldots,i_{d-1}]}(\texttt{init}) & \text{wenn } i_d = 0 \\ \text{value}_{[i_0,\ldots,i_{d-1},i_d-1]}(\texttt{next}) & \text{sonst} \end{cases}$$

In Abbildung 3.4 ist zum Beispiel:

$$\begin{aligned} \text{value}_{[1]}(t_i) &= \text{value}_{[0]}(t_i\,\texttt{.next}) \\ &= \text{value}_{[0]}(-) \\ &= \text{value}_{[0]}(t_i) - 3 \\ &= \text{value}_{[]}(t_i\,\texttt{.init}) - 3 \\ &= N - 3 \end{aligned}$$

Definition 11 (η-Knoten)**.** Für einen Knoten $e = \eta(\texttt{cond}, \texttt{value})$ gilt:

$$j = \min\{i : \text{value}_{[i_0,\ldots,i_d,i]}(\texttt{cond}) = \textbf{t}\} \cup \{\infty\}$$

$$\text{value}_{[i_0,\ldots,i_d]}(e) = \begin{cases} \text{value}_{[i_0,\ldots,i_d,j]}(\texttt{value}) & \text{wenn } j < \infty \\ \bot & \text{sonst} \end{cases}$$

Kann die Auswertung eines η-Knotens nicht abgeschlossen werden, das heißt wird der Wert am `cond`-Eingang nie **t**, handelt es sich um eine Endlosschleife, und der Wert des η-Knotens ist undefiniert ($\bot$).

3.2.4. Korrektheit

Zunächst gilt folgende allgemeine Forderung für graphbasierte Zwischendarstellungen: Jeder Knoten muss genau so viele Vorgänger haben, wie die Signatur seiner Operation vorschreibt, und die Typen der Vorgänger müssen den Typen in der Signatur entsprechen.

Darüber hinaus lässt die Darstellung von Schleifen Raum für semantisch falsche Graphen. Damit ein vFirm-Graph wohlgeformt ist, müssen folgenden Bedingungen gelten (siehe auch [Lie11]):

- Jeder Zyklus im Graphen muss über einen θ-Knoten t laufen und die Kante $t \to t.\texttt{next}$ enthalten.
- Für jeden θ-Knoten t muss gelten $t.\texttt{init}.\mathsf{depth} < t.\mathsf{depth}$ und $t.\texttt{next}.\mathsf{depth} = t.\mathsf{depth}$.
- Für jeden η-Knoten e muss gelten $e.\texttt{value}.\mathsf{depth} = e.\texttt{cond}.\mathsf{depth} = e.\mathsf{depth} + 1$.
- Beim `Return`-Knoten v_r muss $v_r.\mathsf{depth} = 0$ gelten.

Der letzte Punkt stellt sicher, dass der Graph für jeden Schleifenzugriff ausreichend viele η-Knoten enthält. Eine wichtige Eigenschaft ist, dass keine Kante $u \to v \in E$ mit $u.\mathsf{depth} < v.\mathsf{depth}$ und $u.\mathsf{op} \neq \eta$ existiert. Dies folgt aber aus der Definition der Schleifentiefe und lässt sich nicht zur Verifikation verwenden.

3.3. Grapheigenschaften

In diesem Abschnitt stellen wir einige wichtige Eigenschaften der vFirm-Graphen vor.

3.3.1. Vollständigkeit, referentielle Transparenz und Normalisierung

Die Graphen der vFirm-Darstellung sind vollständig, in dem Sinne, dass sie alle Informationen enthalten, die für eine spätere Codeerzeugung notwendig sind [TSTL09].

Darüber hinaus ist die Darstellung *referentiell transparent*. Dies bedeutet, dass man jeden Knoten auswerten und durch den berechneten Wert ersetzen darf, ohne das Ergebnis des Programms zu verändern [Law07].

Zusammen erlauben diese beide Eigenschaften die uneingeschränkte Anwendung von semantikerhaltenden Musterersetzungsregeln auf den Graphen.

Das hohe Abstraktionsniveau der vFirm-Darstellung wirkt normalisierend: Verschiedene Varianten eines Programms mit der gleichen Semantik werden auf einen eindeutigen vFirm-Graph abgebildet, wenn sie sich nur in Eigenschaften unterscheiden, die in der vFirm-Darstellung nicht spezifiziert sind [Sta11, Lie11]. Insbesondere werden Berechnungen normalisiert, die sich nur in der Anordnung der Operationen unterscheiden.

3.3.2. Toter Code

Definition 12. Wir nennen einen Knoten $v \in V$ *tot*, wenn $\alpha(v) = 0$ ist, der Knoten also keine Verwender hat.

Die Elimination von totem Code ist eine traditionelle Optimierung, die Berechnungen entfernt, die nichts zum Ergebnis des Programms beitragen.

In vFirm-Graphen und den verwandten Darstellungen passiert diese Transformation inhärent: Knoten, deren Werte nicht für die Auswertung des `Return`-Knotens benötigt werden, sind auch nicht von dort aus über Abhängigkeitskanten erreichbar [BBZ11].

Ein vFirm-Graph kann also toten Code prinzipbedingt nicht darstellen.

3.3.3. Bedingungsknoten

Folgende Definition erleichtert uns die Argumentation über eine bestimmte Art von Knoten:

Definition 13 (Bedingungsknoten [Lie11])**.**

$$\mathit{Conds} = \{p \in V \mid \exists g = \gamma(p, \texttt{true}, \texttt{false}) \in V\}$$

definiert die Menge der *Bedingungsknoten*.

Ein Bedingungsknoten ist ein Knoten, der sich am `cond`-Eingang eines γ-Knotens befindet. Wir nennen p auch das *Prädikat* von g.

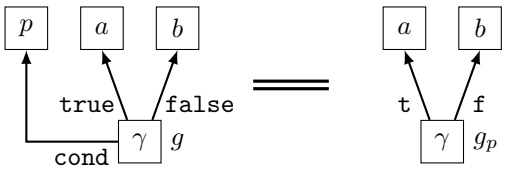


Abbildung 3.6.: Kompakte Notation für γ-Knoten

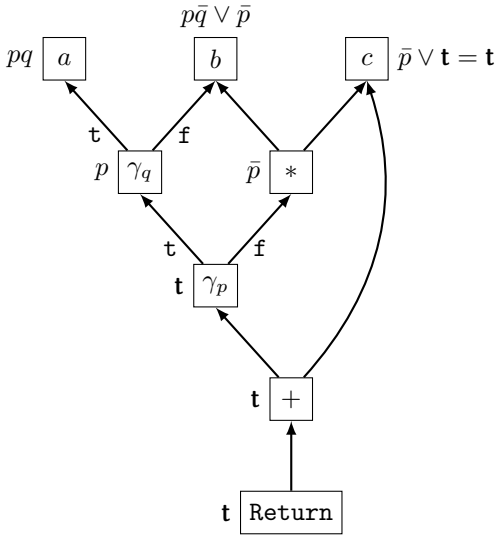


Abbildung 3.7.: Graph mit eingezeichneten Gatingbedingungen

Wir werden häufig die kompakte Darstellung des Prädikats als Index des zugehörigen γ-Knotens wie in Abbildung 3.6 verwenden.

3.3.4. Dominatoren

Die ursprünglich für Flussgraphen formulierte Dominanzrelation [Aho08] lässt auch auf vFirm-Graphen in Bezug zum `Return`-Knoten v_r definieren.

Definition 14 (Dominator)**.** Ein Knoten u dominiert einen Knoten v (u *dom* v), wenn alle Pfade $v_r \rightarrow^* v$ von v_r nach v über u laufen.

Definition 15 (direkter Dominator)**.** Ein Knoten u ist direkter Dominator eines Knotens v ($u = idom(v)$), wenn $\forall u' : u' \neq v \wedge u'$ *dom* v gilt: u' *dom* u.

Der direkte Dominator ist also der Knoten, der von allen Dominatoren von v (außer v selbst) dominiert wird.

Da jeder Knoten mit Ausnahme des Wurzelknotens einen eindeutigen direkten Dominator besitzt, kann man daraus eine Baumstruktur mit der Wurzel v_r und Kanten $(u, v) \Leftrightarrow u = idom(v)$ definieren, die *Dominatorbaum* genannt wird.

3.3.5. Gatingbedingungen

Die Gatingbedingung $\text{gc}(v)$ eines Knotens v gibt an, unter welchen Bedingungen v ausgewertet wird [Lie11, Sta11, Law07]. Es handelt sich um aussagenlogische Formeln, deren Elementaraussagen der Form $\text{value}_L(p) = \mathbf{t}$ und $\text{value}_L(p) = \mathbf{f}$ für einen nicht-konstanten Bedingungsknoten p sind. Wir notieren kürzer p für die erste Aussage und $\bar{p}$ und für die zweite Aussage.

Ein Knoten v mit einem einzigen Verwender u erbt dessen Gatingbedingung, es gilt also $\mathrm{gc}(v) = \mathrm{gc}(u)$. Hat v mehrere Verwender $u_1, \ldots, u_{\alpha(v)}$, ist seine Gatingbedingung die Disjunktion der Gatingbedingungen der Verwender $\mathrm{gc}(v) = \bigvee_{u_i} \mathrm{gc}(u_i)$.

Ist einer der Verwender ein γ-Knoten g mit $g.\texttt{cond} = p, p.\texttt{op} \neq \texttt{Const}$ und $g.\texttt{true} = v$, geht der Term $\mathrm{gc}(g) \wedge p$ in die Gatingbedingung von v ein. Analog haben wir $\mathrm{gc}(g) \wedge \bar{p}$, wenn $g.\texttt{false} = v$ ist.

Eine Sonderbehandlung betrifft Verwender u, die vom betrachteten Knoten v dominiert werden. In diesem Fall ignorieren wir die Gatingbedingung von u. Wir können uns die Situation als Rückwärtskante $u \to v$ eines zyklischen Teilgraphen vorstellen.

Die Gatingbedingung eines Knotens ist entweder $\mathbf{t}$, $\mathbf{f}$ oder sie besteht aus einer Disjunktion von Konjunktionen von Elementaraussagen $Q_1 \vee \cdots \vee Q_m$ mit $Q_i = x_{1,i} \wedge \cdots \wedge x_{n,i}$ und $x_{j,i} = q$ oder $\bar{q}$ für einen nicht-konstanten Bedingungsknoten q.

Die Konjunktionen von Elementaraussagen Q_i seien so vereinfacht, dass jede Elementaraussage nur einmal vorkommt und sie keine widersprüchlichen Aussagen enthalten[1]. Die einzelnen Q_i seien paarweise verschieden.

Die Disjunktionen werden mit den Regeln

$$\begin{aligned} \mathbf{t} \vee Q &= \mathbf{t} \\ \mathbf{f} \vee Q &= Q \\ r \vee (r \wedge Q) &= r \\ (r \wedge Q) \vee (\bar{r} \wedge Q) &= Q \end{aligned}$$

mit Elementaraussagen $r, \bar{r}$ und einer Konjunktion von Elementaraussagen Q vereinfacht [Lie11].

Abbildung 3.7 zeigt einen Beispielgraph, dessen Knoten mit den Gatingbedingungen gekennzeichnet sind. Knoten a wird nur ausgewertet, wenn p und q gelten. Knoten b ist über unterschiedliche Pfade erreichbar und hat daher eine disjunktive Gatingbedingung. Knoten c wiederum ist direkt vom `Return`-Knoten aus erreichbar und wird daher immer ausgewertet.

3.3.6. Schleifentiefe und Schleifeninvarianz

Von zentraler Bedeutung für die Transformation der Darstellung ist die Schleifentiefe, die wir bereits eingeführt haben.

Berechnung der Schleifentiefe Die Schleifentiefe kann gemäß Definition 8 in einer Post-order-Graphtraversierung berechnet werden. Eine Fixpunktiteration ist nicht notwendig, da Zyklen im Graphen immer einen θ-Knoten enthalten, dessen Schleifentiefe a priori bekannt ist. Bei der Traversierung ignoriert man die `next`-Eingänge der θ-Knoten vorerst und trägt sie in eine Warteschlange ein, die nach Abschluss der Traversierung abgearbeitet wird [Lie11].

Mit dem folgenden Begriff erfassen wir Knoten mit einer geringeren Schleifentiefe als ein aktuell betrachteter Knoten.

Definition 16 (schleifeninvariante Knoten)**.** Ausgehend von einem Knoten $v \in V$ sind alle Knoten $w \in V$ mit $w.\mathsf{depth} < v.\mathsf{depth}$ *schleifeninvariant.*

3.3.7. Schleifengruppen

In der vFirm-Darstellung gibt es keine eindeutige Zuordnung von θ- und η-Knoten zu den Schleifen im ursprünglichen Programm. Jeder in der Schleife veränderte Wert wird durch seinen eigenen θ-Knoten dargestellt; die Schleife zerfällt in unserer Darstellung in einen oder mehrere scheinbar unabhängige θ-Knoten. Aus der Definition der Wertfunktion kann man ableiten, dass benachbarte Knoten mit der gleichen Schleifentiefe immer gemeinsam pro Iteration ausgewertet werden.

Das motiviert die folgende Definition, die den verlorengegangen Zusammenhang der in einer Schleife berechneten Werte wiederherstellt.

[1] Solche Konjunktionen werden zu $\mathbf{f}$ vereinfacht.

Definition 17 (Schleifengruppe)**.** Die Partitionierung der Knotenmenge V in *Schleifengruppen* erfolgt nach folgender Relation für Knoten $a, b \in V$ mit $a, b.\mathsf{op} \neq \eta$:

$$\begin{aligned} a \sim b \quad &\Leftrightarrow \quad a \in S \wedge b \in S \\ &\Leftrightarrow \quad a.\mathsf{depth} = b.\mathsf{depth} \wedge (a \rightarrow^* b \vee b \rightarrow^* a) \end{aligned}$$

Zwei Knoten befinden sich in derselben Schleifengruppe genau dann, wenn ihre Schleifentiefe gleich ist und ein Pfad von a nach b oder umgekehrt existiert.

Die η-Knoten zählen wir nicht zu einer Schleifengruppe. Stattdessen assoziieren wir die Menge der η-Knoten, die Werte aus einer Schleifengruppe S extrahieren, mit dem Attribut $S.\mathsf{etas} = \{e : \exists e \rightarrow s \in E \text{ mit } s \in S \text{ und } e.\mathsf{op} = \eta\}$.

Schleifengruppen der Tiefe d sind durch η-Knoten mit Gruppen der Tiefe $d + 1$ verbunden. Daher ergibt sich eine Baumstruktur der Gruppen; man kann auch sagen, dass eine Gruppe der Tiefe $d + 1$ von einer eindeutigen Elterngruppe der Tiefe d umgeben wird.

Ein Verfahren zur Berechnung der Schleifengruppen wird im Abschnitt zu Schleifenoptimierungen vorgestellt. In Abbildung 5.18 auf Seite 62 sind die Schleifengruppen eines Beispielgraphen eingezeichnet.

4. Stand der Technik

Zur Klasse der funktionalen und referentiell transparenten Zwischendarstellungen gehören neben der vFirm-Darstellung noch der *Value State Dependence Graph* (VSDG) [Sta11, Law07, Joh04] beziehungsweise dessen Vorläufer *Value Dependence Graph* (VDG) [WCES94], die *Gated Data Dependence Graphs* [Upt06], sowie der *Program Expression Graph* (PEG) [TSTL09].

4.1. Vergleich der Darstellungen

Bei allen Zwischendarstellungen handelt es sich um Graphen mit der Eigenschaft, dass Knoten bei Bedarf ausgewertet werden und eine ternäre Gatingfunktion γ (bzw. ϕ im PEG) zur Darstellung einer bedingten Auswertung verwendet wird.

Bei der Darstellung von Operationen in Schleifen gibt es dagegen Unterschiede.

4.1.1. Unterschiede in der Schleifendarstellung

Betrachten wir zunächst die grundlegenden Konzepte.

Rekursive Teilgraphen Weise et al. [WCES94] und Lawrence [Law07] verwenden Rekursion, um Schleifen darzustellen. Die Zwischendarstellung ist ein hierarchischer Graph, in dem bestimmte Knoten einen Teilgraphen kapseln. Die Auswertung eines solchen Knotens löst dann die Auswertung des zugehörigen Teilgraphen aus. Eine Schleife wird durch rekursive Teilgraphen und die wiederholte Auswertung des zum Teilgraphen gehörenden Knotens modelliert.

Zyklischen Graphen Die vFirm-Darstellung nutzt, wie wir bereits gesehen haben, eingeschränkt zyklische Graphen mit den Gatingfunktionen θ und η.

Upton [Upt06] verwendet Gatingfunktionen μ und η, wobei die μ-Funktion äquivalent der θ-Funktion ist.

Tate et al. [TSTL09] nutzen eine θ-Funktion, trennen aber die η-Funktion in separate Funktionen `pass` und `eval` auf.

Azyklische Graphen Azyklische Graphen werden in den Arbeiten von Johnson [Joh04] und Stanier [Sta11] verwendet, die Schleifen durch die paarweise auftretenden Knotentypen θ_{head} und θ_{tail} repräsentieren. Zwischen den beiden Knoten steht ein azyklischer Teilgraph, der dem Schleifenkörper entspricht. Die Schleife wird durch einen impliziten Wertefluss von Fuß- zum Kopf-Knoten geschlossen. Die Schleifenabbruchbedingung ist eine Bedingung, die vom θ_{tail}-Knoten ausgewertet wird.

4.1.2. Darstellbarkeit von Nichttermination

Innerhalb des Value Dependence Graphs, des Programm Expression Graphs und der vFirm-Darstellung ist die Schleifendarstellung nicht zustandsbehaftet, außer es handelt sich explizit um eine Schleife des

```
1 int nt() {
2   int i = 0;
3   for (;; i++);
4   return 0;
5 }
```

Listing 4.1: Problematische Funktion für zustandslose Schleifendarstellung

Speicherwerts. Dadurch lässt sich die Nichttermination in einer Funktion wie in Listing 4.1 nicht ausdrücken, weil es nach der Schleife keinen Verwender des Werts von `i` gibt. In der Folge terminiert die Funktion nach dem Übersetzen.

In den Varianten des Value State State Dependence Graph sind die Schleifenkonstrukte hingegen grundsätzlich zustandsbehaftet. Diese Darstellungen sind daher nicht von der Terminationsproblematik betroffen.

4.1.3. Wertigkeit der Schleifenkonstrukte

Ein weiteres Unterscheidungsmerkmal ist die Darstellung von Werten, die aus derselben Schleife im Quelltext stammen.

In der vFirm- und der PEG-Darstellung werden solche Werte durch unabhängige θ-Knoten repräsentiert.

Die anderen Darstellungen gruppieren Werte pro Iteration in Tupel; im Falle der Darstellungen, die Gatingfunktionen zur Schleifendarstellung nutzen, entsteht dadurch eine eindeutige Zuordnung von θ- beziehungsweise μ-Knoten zu den η- oder θ_{tail}-Knoten. Die Darstellungen mit zustandsbehafteten Schleifen führen den Speicherwert dann als Element dieser Tupel mit.

4.2. Optimierung

In diesem Abschnitt wird vorgestellt, welche Ansätze zur Optimierung im Umfeld der funktionalen und referentiell transparenten Zwischendarstellungen bereits verfolgt wurden.

4.2.1. Value (State) Dependence Graph und Gated Data Dependence Graph

Traditionelle Optimierungen wie Konstantenfaltung, Elimination gemeinsamer Teilausdrücke, Umordnung von assoziativen und kommutativen Operationen, Kopienpropagation und die Elimination von totem und unerreichbarem Code lassen sich auf der VSDG-Darstellung einfach durchführen oder passieren implizit; lokale Transformationen von γ-Knoten vereinfachen den Steuerfluss des Programms [WCES94, Joh04, Upt06].

Darüber hinaus lag der Fokus der Autoren auf folgenden Aspekten:

Weise et al. [WCES94] nutzen die Darstellung, um Codeverschiebung, insbesondere die Elimination von partiellen Redundanzen durchzuführen. Die Transformation findet allerdings nicht direkt auf dem VDG statt, sondern erst nachdem der sogenannte “demand-based program dependence graph” im Zuge der Codegenerierung basierend auf dem VDG erstellt wurde.

Johnson [Joh04] nutzt die VSDG-Darstellung, um möglichst kompakten Code für den Einsatz auf eingebetteten Systemen zu erzeugen. Dazu sucht er in der Darstellung Teilgraphen, die an anderen Stellen wiederverwendet werden können, gruppiert Speicherzugriffe und geht das Phasenordnungproblem zwischen Codeverschiebung und Registerallokation an, indem er beide Aspekte gemeinsam durchführt.

Upton [Upt06] definiert eine auf abstrakter Interpretation basierende statische Analyse zur Beschränkung der möglichen Wertebereiche der Knoten im Graph. Er beschreibt das Finden von Induktionsvariablen und einige anschließend mögliche Schleifenoptimierungen, sowie eine Datenflussanalyse, die ermittelt, welche Bits eines Knotenwerts lebendig sind.

Lawrence [Law07] und Stanier [Sta11] untersuchen die Möglichkeiten zur Steuerflussoptimierung, die die Rekonstruktion des Steuerflusses bietet. Dazu nutzen sie die Freiheiten, die die Darstellung bei der Umordnung von Operationen (insbesondere auch der Verzweigungen entsprechenden γ-Knoten) bietet.

In dieser Arbeit werden Optimierungen untersucht, die ausschließlich auf der Zwischendarstellung, und nicht während des Abbaus stattfinden. Es werden ebenfalls traditionelle Optimierungen implementiert, allerdings ist das Ziel eine Verbesserung der Ausführungsgeschwindigkeit. Der Vorteil der hier vorgestellten Steuerflussoptimierung ist, dass ihr Effekt Ausgangspunkt für weitere Transformationen sein kann. Eine Evaluation der Anwendungsmöglichkeiten der vorgeschlagenen Optimierungen an Code aus praxisnahen Benchmarks liefert keine der genannten Arbeiten.

4.2.2. Program Expression Graph

In [TSTL09] stellen Tate et al. ihre Optimierungsmethode der *Equality Saturation* auf Program Expression Graphs vor. Die Idee des Verfahrens ist, bei Anwendbarkeit einer Optimierung deren Transformation nicht durchzuführen, sondern lediglich die Äquivalenz zwischen ursprünglicher und transformierter Variante zu vermerken und sie somit gleichzeitig in der Zwischendarstellung zu repräsentieren. Können auf diese Art keine weiteren Äquivalenzen hinzufügt werden, wählt eine globale Heuristik die gemäß eines statischen Kostenmodells beste Repräsentation aus.

Die Autoren berichten, dass eine Vielzahl von traditionellen und unerwarteten Optimierungen durch Anwendung einer Menge von Äquivalenzaxiomen durchgeführt wurden, sowie dass das Verfahren den Compilerentwickler von Phasenordnungsproblemen befreit. Eine Besonderheit ist, dass auch Optimierungen durchgeführt werden, die erst durch vorhergehende, verschlechternde Transformationen ermöglicht wurden.

Im Kontext der Äquivalenzklassen scheint die Durchführung von globalen Analysen problematisch zu sein, wie sie etwa für die später vorgestellte Elimination von redundanten γ-Knoten einsetzt werden, da die Repräsentanten einer Äquivalenzklasse völlig unterschiedliche Strukturen haben können. Entscheidet man sich, alle globalen Analysen auf der ursprünglichen Variante des Programms durchzuführen, kann man nicht von den Effekten anderen Transformationen profitieren. Phasenordnungsprobleme bei der Anwendung von Graphersetzungsmustern lassen sich durch das Einhalten von Normalisierungskonventionen mit eventueller Rückersetzung im Backend [Tra01] lindern. Die Modellierung der Auswahl der besten Variante als pseudo-boolesches Problem bringt wiederum eine gewisse zusätzliche Komplexität für die Compilerentwicklung mit sich.

Die in dieser Arbeit vorgestellten Optimierungen verfolgen daher den klassischen Ansatz, die Transformation direkt auf die Zwischendarstellung anzuwenden.

5. Optimierung der vFirm-Darstellung

Nachdem wir die vFirm-Darstellung kennengelernt haben, definieren wir nun das Ziel unserer Optimierung und stellen lokale Graphersetzungsmuster, Steuerfluss- und Schleifentransformationen sowie ein Umordnungsverfahren für Summen vor.

5.1. Ziel der Optimierung

In diesem Abschnitt behandeln wir die Frage, wie wir das zu übersetzende Programm transformieren müssen, um seine Ausführungsgeschwindigkeit zu verbessern, und entwickeln darauf aufbauend eine heuristische Kostenfunktion. Unser Ziel ist dann, die Kosten des Programms durch die Anwendung der später vorgestellten Optimierungen zu minimieren.

5.1.1. Allgemeine Überlegungen zur Ausführungsgeschwindigkeit

Zunächst müssen wir definieren, was es bedeutet, dass eine Version eines Programms schneller ist als eine andere.

Die Ausführungsgeschwindigkeit sei die Anzahl der Taktzyklen, die für die Berechnung eines gegebenen Problems benötigt wird. Eine Version `A` ist schneller als eine andere Version `B`, wenn `A` für die Berechnung des Problems weniger Taktzyklen benötigt als `B`.

Wie viele Taktzyklen ein Programm in der Realität tatsächlich braucht, ist nicht berechenbar, weil dies Kenntnis über die Eingaben und das dynamische Verhalten des Programms erfordern würde. Jede im fertig übersetzten Programm kodierte Instruktion kann beliebig viele oder auch keine Taktzyklen verursachen. Unterstützt der Prozessor Parallelität auf Befehlsebene[1], können auch mehr als eine Instruktion pro Taktzyklus bearbeitet werden.

Trotzdem lässt sich basierend auf der Anzahl der Instruktionen mit den folgenden Überlegungen eine vernünftige Heuristik für die Bestimmung der Anzahl der benötigten Taktzyklen aufbauen.

Unterteilen wir zunächst die Operationen in vier Klassen:

- Arithmetisch-logische Operationen und Vergleiche werden "lokal" im Rechenwerk des Prozessors mit Werten aus Registern berechnet und lassen sich größtenteils in einem Taktzyklus durchführen. Ausnahmen können je nach Prozessorarchitektur die Multiplikationen und Divisionen und allgemein die Fließkommaarithmetik darstellen.
- Die Dauer von Speicherzugriffen ist abhängig davon, wie weit entfernt das Ziel in der Speicherhierarchie liegt. In jedem Fall benötigen sie mehr Taktzyklen als arithmetisch-logische Operationen.
- Funktionsaufrufe können beliebig lange dauern, abhängig vom Verhalten der aufgerufenen Funktion. Allgemein müssen wir annehmen, dass sie deutlich mehr als einen Taktzyklus zur Laufzeit des Programms beitragen.
- Bedingte Sprünge interferieren mit der Pipelineverarbeitung, die alle modernen Prozessoren nutzen. Das Problem ist, dass das Sprungziel der Instruktion erst bekannt ist, wenn schon einige der nachfolgenden Instruktionen in die Pipeline geladen wurden. Im Falle eines Sprungs, den die Sprungvorhersage des Prozessors falsch vorhergesagt hat, muss die Pipeline geleert werden und es kommt zu Wartezyklen.

Nun werden nicht alle Instruktionen gleich häufig ausgeführt. Befindet sich eine Instruktion in einer Schleife, kann sie mehrmals ausgeführt werden und dadurch ein Vielfaches an Taktzyklen einer Instruktion außerhalb der Schleife zur Laufzeit des Programms beitragen. Dies gilt umso stärker, je tiefer verschachtelt die betreffende Schleife ist.

[1] engl. *instruction level parallelism*

Im Gegensatz dazu werden Instruktionen, die kontrollabhängig von einer Verzweigung sind, statistisch gesehen weniger häufig ausgeführt und verbrauchen dementsprechend weniger Taktzyklen, als solche, die unbedingt zur Ausführung kommen. Wir nehmen an, dass bedingte Sprünge, die nicht Teil eines Schleifenkopfs sind, beide Ziele gleichverteilt anspringen.

Die Codegröße des übersetzten Programms hat indirekt auch Einfluss auf die Ausführungsgeschwindigkeit. Wird der Programmcode so groß, dass er nicht mehr vollständig in den Instruktionscache des Prozessors passt, kann es zu langsamen Speicherzugriffen vor dem Laden einer neuen Instruktion kommen. Dies wirkt sich negativ auf die Geschwindigkeit aus.

5.1.2. Adaption auf die vFirm-Darstellung

Zwischen den Überlegungen des vorangegangenen Unterabschnitts und der vFirm-Darstellung liegt eine große Abstraktionslücke, die wir nun ausfüllen wollen.

Wir betrachten nun statt der Ausführung von Instruktionen die Auswertung von Knoten. Ein Knoten der vFirm-Darstellung wird ausgewertet, wenn *mindestens* einer seiner Verwender seinen Wert anfordert. Die Auswertung beginnt dabei beim `Return`-Knoten.

Wir treffen die Annahme, dass die Auswertung jedes Knotens durch eine Instruktion im übersetzten Programm implementiert wird. Davon ausgenommen sind die Projektionen, die nur für die Zwischendarstellung von Bedeutung sind, die echten und symbolischen Konstanten sowie spezielle Typkonversionen.

Für arithmetische Operationen, Speicherzugriffe und Funktionsaufrufe existiert eine eindeutige Zuordnung von Instruktionen zu Knotentypen. Hingegen gibt es keine Sprungbefehle in der vFirm-Darstellung. Die γ-Knoten repräsentieren eine bedingte Auswertung und kommen daher bedingten Sprüngen konzeptionell am nächsten. Wir nehmen also an, dass die Auswertung eines γ-Knotens durch einen bedingten Sprung implementiert wird.

Mehrfache Auswertung durch Schleifen Ein Knoten v mit einer Schleifentiefe $d > 0$ wird einmal pro Iteration jeder Verschachtelungstiefe ausgewertet. v hat die Wertfunktion $\text{value}_{[i_1,\ldots,i_d]}(v)$, die für alle möglichen Belegungen der Iterationszähler $i_1, \ldots, i_d$ einen neuen Wert berechnet. Wir nehmen an, dass jede Schleife betreten wird und $\mathbb{L}$ Iterationen durchläuft. Dann wird v insgesamt $\mathbb{L}^d$-mal ausgewertet.

Bedingte Auswertung Knoten, deren Auswertung ausschließlich unter der Bedingung eines γ-Knotens stattfindet, werden statistisch gesehen seltener ausgewertet. Dazu führen wir eine Auswertungswahrscheinlichkeit ein:

Definition 18 (exakte Auswertungswahrscheinlichkeit)**.** Für einen Knoten v definiert

$$P'_{\mathsf{eval}}(v) = P(\text{gc}(v) = \mathbf{t})$$

die *exakte Auswertungswahrscheinlichkeit* als die Wahrscheinlichkeit, dass die Gatingbedingung von v erfüllt wird. Dies entspricht der Wahrscheinlichkeit, dass ausgehend vom `Return`-Knoten über mindestens einen Pfad die Auswertung von v angefordert wird.

Zunächst ist für Knoten, die unbedingt ausgewertet werden $P'_{\mathsf{eval}}(v) = P(\mathbf{t} = \mathbf{t}) = 1$, und für Knoten die nie ausgewertet werden $P'_{\mathsf{eval}}(v) = P(\mathbf{f} = \mathbf{t}) = 0$.

Wir nehmen an, dass der Wert jedes nicht-konstanten Bedingungsknotens *gleichverteilt* $\mathbf{t}$ oder $\mathbf{f}$ ist und folglich $P(p = \mathbf{t}) = P(\bar{p} = \mathbf{t}) = \frac{1}{2}$ ist.

Besteht die Gatingbedingung ausschließlich aus einer Konjunktion von Elementaraussagen, multiplizieren sich die einzelnen Wahrscheinlichkeiten für die Elementaraussagen, zum Beispiel ist für $\text{gc}(v) = p \wedge \bar{q} \wedge r$ die Auswertungswahrscheinlichkeit $P'_{\mathsf{eval}}(v) = P(p \wedge \bar{q} \wedge r = \mathbf{t}) = P(p = \mathbf{t}) \cdot P(\bar{q} = \mathbf{t}) \cdot P(r = \mathbf{t}) = \frac{1}{8}$.

Enthält die Gatingbedingung Disjunktionen, berechnet sich die exakte Auswertungswahrscheinlichkeit als Quotient $\frac{\text{Anzahl erfüllender Belegungen der Bedingungsknoten}}{\text{Anzahl aller Belegungen der Bedingungsknoten}}$.

Wir müssen eine Abschätzung treffen, da die exakte Berechnung zu aufwändig ist.

Definition 19 (Auswertungswahrscheinlichkeit)**.** Sei $\mathrm{gc}(v) = Q_1 \vee \cdots \vee Q_m$ und seien die Q_i entweder $\mathbf{t}$, $\mathbf{f}$ oder Konjunktionen von $|Q_i|$ Elementaraussagen. Dann ist:

$$P_{\mathsf{eval}}(v) = \min\left\{1, \sum_{Q_i} P(Q_i = \mathbf{t})\right\} \quad \geq \quad P'_{\mathsf{eval}}(v)$$

mit

$$P(Q_i = \mathbf{t}) = \begin{cases} 1 & \text{wenn } Q_i = \mathbf{t} \\ 0 & \text{wenn } Q_i = \mathbf{f} \\ \left(\frac{1}{2}\right)^{|Q_i|} & \text{sonst} \end{cases}$$

Zur Berechnung von P_{eval} nehmen wir also die Unabhängigkeit der einzelnen Q_i an und beschränken die Wahrscheinlichkeit auf maximal 1. Dadurch erhalten wir eine Überabschätzung der Auswertungswahrscheinlichkeit. Im Folgenden beziehen sich alle Wahrscheinlichkeiten auf die Erfüllung der Formel, das heißt wir schreiben anstatt $P(X = \mathbf{t})$ kürzer $P(X)$.

Beispiel Die Knoten in Abbildung 3.7 auf Seite 23 haben folgende Auswertungswahrscheinlichkeiten:

- $P_{\mathsf{eval}}(\texttt{Return}) = P_{\mathsf{eval}}(+) = P_{\mathsf{eval}}(\gamma_p) = P_{\mathsf{eval}}(c) = 1$.
- $P_{\mathsf{eval}}(\gamma_q) = P_{\mathsf{eval}}(*) = \frac{1}{2}$.
- $P_{\mathsf{eval}}(a) = \frac{1}{4}$.
- $P_{\mathsf{eval}}(b) = \frac{1}{4} + \frac{1}{2} = \frac{3}{4}$.

Wir werden später bei der Betrachtung der Profitabilität der Transformationen folgende Überlegungen brauchen.

Lemma 6. In unserem Kostenmodell gilt:

$$P_{\mathsf{eval}}(v) \leq \min\left\{1, \sum_{u \to v \in E} P_{\mathsf{eval}}(u)\right\}$$

Das bedeutet, dass die Auswertungswahrscheinlichkeit eines Knotens v kleiner oder gleich der Summe der Auswertungswahrscheinlichkeiten seiner Verwender ist.

Beweis. Die Gatingbedingung von v ist die Disjunktion der Gatingbedingungen der Verwender von v. Ohne Vereinfachungen der disjunktiv verknüpften Terme[2] ist:

$$\begin{aligned} \mathrm{gc}(v) &= \bigvee_{u \to v \in E} \mathrm{gc}(u) \\ P_{\mathsf{eval}}(v) &= \min\left\{1, \sum_{Q \in \mathrm{gc}(v)} P(Q)\right\} \\ &= \min\left\{1, \sum_{u \to v \in E} \sum_{R \in \mathrm{gc}(u)} P(R)\right\} \\ &= \min\left\{1, \sum_{u \to v \in E} \min\left\{1, \sum_{R \in \mathrm{gc}(u)} P(R)\right\}\right\} \\ &= \min\left\{1, \sum_{u \to v \in E} P_{\mathsf{eval}}(u)\right\} \end{aligned}$$

[2]Stellen wir uns die Disjunktion in der Art einer Konkatenation vor.

Mit den Vereinfachungen aus Abschnitt 3.3.5 wird nun die Anzahl der Terme in der Disjunktion reduziert oder bleibt gleich. Ebenso enthält $\mathrm{gc}(v)$ mehrfach vorkommende Terme nur einmal. Da keine neuen oder veränderten Terme hinzukommen, ist die Auswertungswahrscheinlichkeit dann kleiner oder gleich der Summe der Auswertungswahrscheinlichkeiten der Verwender. □

Lemma 7. Für Knoten v', v'' und $g = \gamma(p, v', v'')$ mit $\alpha(v') = \alpha(v'') = 1$ gilt:

$$P_{\mathsf{eval}}(v') + P_{\mathsf{eval}}(v'') \geq P_{\mathsf{eval}}(g)$$

Die Summe der Auswertungswahrscheinlichkeiten der Knoten, die ausschließlich vom `true`- und `false`-Eingang eines γ-Knotens g verwendet werden, ist gleich oder größer der Auswertungswahrscheinlichkeit von g.

Beweis. Sei $\mathrm{gc}(g) = Q_1 \vee \cdots \vee Q_m$, dann ist $\mathrm{gc}(v') = (Q_1 \wedge p) \vee \cdots \vee (Q_m \wedge p)$ und $\mathrm{gc}(v'') = (Q_1 \wedge \bar{p}) \vee \cdots \vee (Q_m \wedge \bar{p})$.

Es gilt:

$$\sum_{Q \in \mathrm{gc}(g)} P(Q \wedge p) + \sum_{Q \in \mathrm{gc}(g)} P(Q \wedge \bar{p}) = \sum_{Q \in \mathrm{gc}(g)} P(Q)$$

weil für die einzelnen Summanden

$$P(Q_i \wedge x) = \begin{cases} \frac{1}{2} \cdot P(Q_i) & \text{wenn } x, \bar{x} \notin Q_i \quad (\text{weil } P(x) = \frac{1}{2}) \\ P(Q_i) & \text{wenn } x \in Q_i \\ 0 & \text{wenn } \bar{x} \in Q_i \end{cases}$$

und dadurch für alle Paare gilt: $P(Q_i \wedge p) + P(Q_i \wedge \bar{p}) = P(Q_i)$. Wir bilden auf beiden Seiten das Minimum mit 1

$$\min\left\{1, \sum_{Q \in \mathrm{gc}(g)} P(Q \wedge p) + \sum_{Q \in \mathrm{gc}(g)} P(Q \wedge \bar{p})\right\} = \min\left\{1, \sum_{Q \in \mathrm{gc}(g)} P(Q)\right\}$$

und erhalten:

$$\min\left\{1, \sum_{Q \in \mathrm{gc}(g)} P(Q \wedge p) + \sum_{Q \in \mathrm{gc}(g)} P(Q \wedge \bar{p})\right\} = P_{\mathsf{eval}}(g)$$

$$\min\left\{1, \sum_{Q \in \mathrm{gc}(g)} P(Q \wedge p)\right\} + \min\left\{1, \sum_{Q \in \mathrm{gc}(g)} P(Q \wedge \bar{p})\right\} \geq P_{\mathsf{eval}}(g)$$

$$P_{\mathsf{eval}}(v') + P_{\mathsf{eval}}(v'') \geq P_{\mathsf{eval}}(g)$$

□

Lemma 8. Für einen Knoten v, dessen einziger Verwender der `true`- oder `false`-Eingang eines γ-Knotens g mit Bedingungsknoten p ist, gilt:

$$P_{\mathsf{eval}}(v) \leq P_{\mathsf{eval}}(g)$$

Beweis. Sei $\mathrm{gc}(g) = Q_1 \vee \cdots \vee Q_m$. Dann ist $\mathrm{gc}(v) = (Q_1 \wedge p) \vee \cdots \vee (Q_m \wedge p)$ mit:

$$P(Q_i \wedge p) = \begin{cases} P(Q_i) \cdot P(p) & \text{wenn } \{p, \bar{p}\} \notin Q_i \\ P(Q_i) & \text{wenn } p \in Q_i \\ 0 & \text{wenn } \bar{p} \in Q_i \end{cases}$$

Für alle Q_i gilt also $P(Q_i \wedge p) \leq P(Q_i)$, und die Aussage folgt. □

Codegröße Zur Abschätzung der resultierenden Codegröße nehmen wir an, dass jeder Knoten, der eine auszuführende Operation darstellt, durch eine Instruktion implementiert wird.

5.1.3. Kostenfunktion

Zusammengefasst haben wir in unserem Modell also drei Möglichkeiten, um die Knoten eines vFirm-Graphen bezüglich ihrer erwarteten Laufzeit zu verbessern:

1. Elimination
2. Verringerung der Auswertungswahrscheinlichkeit
3. Verringerung der Schleifentiefe

Daraus lässt sich folgende Heuristik für die Bestimmung der Kosten des Knotens v ableiten:

$$c(v) = P_{\mathsf{eval}}(v) \cdot \mathbb{L}^{v.\mathsf{depth}} \cdot \omega(v)$$

$\mathbb{L}$ ist der Schätzwert für die Anzahl an Iterationen, die Schleifen im Durchschnitt durchlaufen. Wir wählen $\mathbb{L} = 10$.

$\omega(v)$ definiert ein von der Operation des Knotens abhängiges, statisches Kostenmaß. Wir setzen $\omega(v) = 0$ für alle Knotentypen, die zur Laufzeit nicht ausgeführt werden müssen. Dazu gehören Projektionen, Konstanten und Konversionen von einem Integertyp in einen Typ kleinerer oder gleicher Bitbreite. Für γ-Knoten sei $\omega(v) = 3$.

Unsere Optimierungen zielen nicht speziell auf die Optimierung von Speicherzugriffen und sind intraprozedural, deswegen unterscheiden wir die Kosten von arithmetisch-logischen Operationen und Speicherzugriffen oder Funktionsaufrufen nicht weiter, und setzen für sie alle $\omega(v) = 1$.

Die zu minimierenden Kosten für den Graphen betragen dann:

$$C(G = (V, E)) = \sum_{v \in V} c(v)$$

Tate et al. [TSTL09] verwendet für die PEG-Darstellung ebenfalls eine Kostenfunktion, bei der die Schleifentiefe exponentiell eingeht. Sie enthält aber keinen zur Auswertungswahrscheinlichkeit äquivalenten Faktor.

Haben mehrere Graphen die gleichen Kosten, bevorzugen wir den mit der geringsten Codegröße, abgeschätzt durch

$$C_s(G) = |\{v \in V : \omega(v) > 0\}|$$

5.2. Lokale Optimierungen

Die in der Firm-Darstellung verfügbaren lokalen Optimierungen [Tra01, Lin02, BBZ11] stehen aufgrund der Verwandtschaft der Darstellungen auch für vFirm-Graphen zur Verfügung.

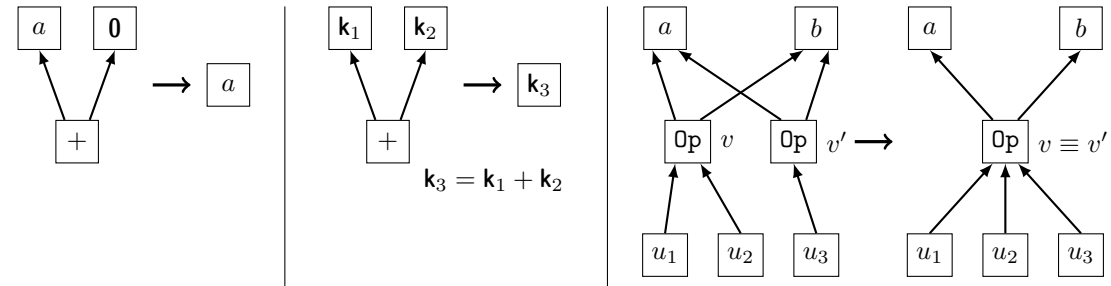
Die lokalen Optimierungen werden als Musterersetzungen implementiert. Man unterscheidet verbessernde und normalisierende Regeln. Bei verbessernden Regeln hat die rechte Seite der Regel weniger Knoten[3] als die linke Seite. Normalisierende Regeln bringen den Graphen in eine bestimmte Form, um die Anzahl der Varianten in den Vorbedingungen anderer Regeln zu reduzieren. So wird beispielsweise bei kommutativen Operationen ein konstantes Argument immer zum rechten Argument gemacht.

Der Graph wird ausgehend vom `Return`-Knoten traversiert, wobei die Vorgänger eines Knotens immer vor dem Knoten selbst besucht werden. Für den aktuellen Knoten wird dann versucht, eine verbessernde Regel, und wenn das nicht möglich ist, eine normalisierende Regel anzuwenden.

Die Regeln müssen dabei so spezifiziert sein, dass eine anwendbare Regel eindeutig ist, und dass die normalisierenden Regeln keine Ersetzungszyklen entstehen lassen [Tra01].

Wir stellen im Folgenden die wichtigsten Regeln kurz vor.

[3] In [Tra01] haben die Operationen unterschiedliche Kosten, dementsprechend gibt es dort Regeln, die teure Operationen durch billigere ersetzen. Bezogen auf unser Kostenmodell wäre eine solche Regel kostenneutral.



(a) Algebraische Identität (b) Konstantenfaltung (c) Knotenverschmelzung

Abbildung 5.1.: Lokale Optimierungen

5.2.1. Algebraische Vereinfachungen

Das Ausnutzen von algebraischen Identitäten ist eine wichtige Vereinfachung des Programms. Beispiele für Identitäten sind $a + 0 = a$, $a * 0 = 0$ oder $a - a = 0$. Eine umfangreichere Übersicht gibt [Muc97]. Abbildung 5.1a zeigt die erste Regel als Graphtransformation. Alle Transformationen dieser Kategorie eliminieren Knoten und verringern damit die Kostenfunktion des Programms. Im gezeigten Beispiel reduzieren sie sich um die Kosten der Addition $c(+)$.

Zudem können durch die Transformation weitere Knoten unerreichbar werden, die dann wiederum in einem späteren Schritt eliminiert werden können.

5.2.2. Konstantenfaltung

Bei der Konstantenfaltung ersetzen wir Operationen mit konstanten Argumenten durch eine neue Konstante, deren Wert sich durch die Anwendung der Operation auf ihre Argumente berechnet. Abbildung 5.1b zeigt dies am Beispiel einer Addition. Die Konstantenfaltung verringert die Kosten um $c(Op)$, da die Operation durch eine Konstante ersetzt wird.

Die neu erstellte Konstante kann weitere Faltungen bei ihren Verwendern ermöglichen.

5.2.3. Knotenverschmelzung

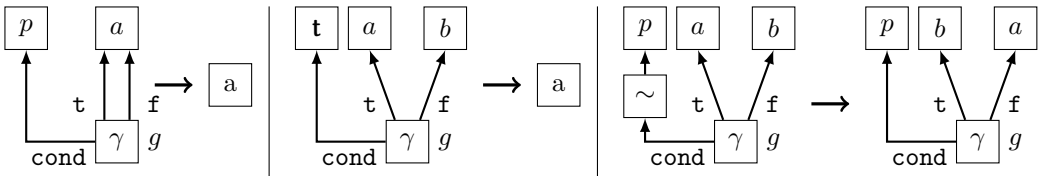
Zwei Knoten der gleichen Operation, die dieselben Vorgänger und identische Attribute haben, berechnen in vFirm denselben Wert und können deshalb verschmolzen werden. [Law07, Tra01]. Abbildung 5.1c zeigt die Transformation. Solche Knoten können durch den Einsatz einer Hashtabelle effizient gefunden werden. Die Transformation eliminiert einen Knoten und leitet seine Verwender auf den verbleibenden Knoten um.

Betrachten wir die Kosten des Programms bei Verschmelzung von zwei Knoten v und v', wobei $v \equiv v'$ den verschmolzenen beziehungsweise den nach der Transformation übrigbleibenden Knoten bezeichnet.

Die Schleifentiefe (aufgrund identischer Vorgänger) und die statischen Kosten ω für die Operation (gemäß der Vorbedingungen) sind für v, v' und $v \equiv v'$ gleich.

Wir können uns den Knoten $v \equiv v'$ als Knoten mit den Verwendern v und v' vorstellen. Nach Lemma 6 gilt dann $P_{\mathsf{eval}}(v \equiv v') \leq \min\{1, P_{\mathsf{eval}}(v) + P_{\mathsf{eval}}(v')\}$.

Die Auswertungswahrscheinlichkeit ist also kleiner oder gleich der Summe der Auswertungswahrscheinlichkeiten der ursprünglichen Knoten. Insgesamt verringert die Knotenverschmelzung die Kosten oder lässt sie gleich. Nach der Verschmelzung enthält der Graph jedoch einen Knoten weniger, so dass die Transformation aufgrund der Verbesserung von C_s durchgeführt wird. Darüber hinaus hat sie eine normalisierende Wirkung.



(a) Identische Vorgänger (b) Konstantes Prädikat (c) Negiertes Prädikat

Abbildung 5.2.: Vereinfachungen des γ-Knotens

5.3. Lokale Optimierungen mit γ-Knoten

Ein herausragendes Merkmal der vFirm-Darstellung ist, was Lawrence als "uniformity of expressions" [Law07] bezeichnet. Alle Knotentypen der Darstellung haben eine Semantik, die sich zum Zeitpunkt der Optimierung auswerten lässt. Insbesondere gilt das auch für den γ-Knoten. Wir können also durch Transformationen des vFirm-Graphen den impliziten Steuerfluss des Programms verändern [WCES94].

5.3.1. Vereinfachung der γ-Knoten

Identische Argumente SameArg

Abbildung 5.2a zeigt einen γ-Knoten $g = \gamma(p, a, a)$, dessen Argumente am `true`- und `false`-Eingang identisch sind. g hat keine Funktion, da er unabhängig vom Wert seines Prädikats den gleichen Vorgänger anfordert. g wird eliminiert und seine Nutzer werden mit a verbunden, dadurch verbessern sich die Kosten des Graphs um $c(g)$. Die Auswertungswahrscheinlichkeit von a ändert sich nicht. Falls p keine weiteren Verwender hat, wird der Knoten unerreichbar und kann ebenfalls mit positiven Auswirkungen auf die Kosten eliminiert werden. Die Transformation wird auch in [Upt06] beschrieben.

Konstantes Prädikat ConstCond

Sei nun das Prädikat des γ-Knotens $g = \gamma(\mathbf{t}, a, b)$ in Abbildung 5.2b konstant. Die Nutzer von g werden mit dessen `true`-Eingang verbunden und g wird eliminiert [WCES94, Joh04, Upt06]. Die Kosten verbessern sich um $c(g)$. Der Vorgänger am `false`-Eingang kann durch die Transformation unerreichbar werden.

In unserem Kostenmodell hat ein γ-Knoten mit konstantem Prädikat keinen Einfluss auf die Auswertungswahrscheinlichkeit von a und b, weil diese Situation schon bei der Berechnung der Kosten berücksichtigt wurde.

Die Transformation für den Fall, dass das Prädikat den Wert $\mathbf{f}$ hat, erfolgt analog.

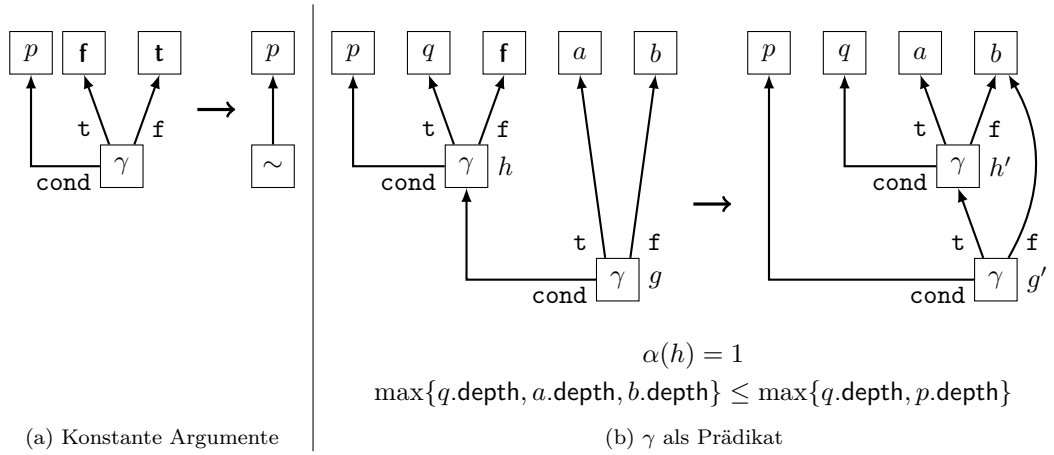
Vergleichen wir die Kosten eines Graphen vor und nach der Ersetzung des Bedingungsknotens eines γ-Knotens g durch eine Konstante. Es vorkommen, dass die Kosten sich aufgrund der erhöhten Auswertungswahrscheinlichkeit der Knoten auf der einen Seite von g verschlechtern. Diese Verschlechterung resultiert jedoch aus der falschen Annahme der Gleichverteilung der Werte des ersetzten Bedingungsknotens; das zu optimierende Programm profitiert selbstverständlich von einer solchen Konstantenfaltung, die auch weitere Optimierungsmöglichkeiten schaffen kann.

Negiertes Prädikat NegCond

Ist der Knoten am Bedingungseingang eines γ-Knotens $g = \gamma(\sim p, a, b)$ eine Negation, vertauschen wir die `true`- und `false`-Vorgänger von g und verwenden das Argument der Negation als neuen Bedingungsknoten. Hatte die Negation keine weiteren Verwender, ist der Knoten anschließend tot und die Kosten reduzieren sich um $c(\sim)$. Ansonsten wirkt die Transformation normalisierend.

Boolesche γ-Knoten ModeBGamma

γ-Knoten g, die boolesche Werte produzieren ($g.\mathsf{type} = \mathbb{B}$) und deren Argumente beide Konstanten sind, erfahren wie in Abbildung 5.3a dargestellt eine Sonderbehandlung. Die Konstanten müssen verschieden sein, denn andernfalls wäre der Knoten durch die oben beschriebenen Vereinfachungen schon



(a) Konstante Argumente

(b) γ als Prädikat

Abbildung 5.3.: Transformation boolescher γ-Knoten

eliminiert worden. Ist $g = \gamma(p, \mathbf{t}, \mathbf{f})$, wird g durch p ersetzt; ist $g = \gamma(p, \mathbf{f}, \mathbf{t})$, ersetzen wir g durch die Negation von p. Die Elimination von g verbessert im ersten Fall die Kosten des Programms um $c(g)$, im zweiten Fall immerhin um $c(g) - c(\sim)$.

γ-Knoten in Prädikaten GammaOnCond

Betrachten wir Abbildung 5.3b. Sei $g = \gamma(h, a, b)$ und $h = \gamma(p, q, \mathbf{f})$, wobei a, b beliebige Knoten und q ein variabler boolescher Wert ist. Es handelt sich also um einen γ-Knoten, dessen Prädikat ein weiterer γ-Knoten mit genau einer Konstante[4] ist.

Wir sehen, dass im Fall $p = \mathbf{f}$ immer $\mathbf{f}$ als Prädikat für g verwendet wird. Das nutzen wir aus, in dem wir einen neuen Knoten $g' = \gamma(p, h' = \gamma(q, a, b), b)$ erzeugen, wenn h keine weiteren Verwender hat. Dieser Teilgraph besteht zwar immer noch aus zwei γ-Knoten, es gilt aber nach Lemma 8 $P_{\mathsf{eval}}(h') \leq P_{\mathsf{eval}}(h) = P_{\mathsf{eval}}(g')$ durch das Verschieben der Auswertung von h' unter die Bedingung p. Die Auswertungswahrscheinlichkeiten von p und q verändern sich nicht.

Wie beim Ersetzen des Prädikats eines γ-Knotens durch eine Konstante stoßen wir auch hier auf eine falsche Annahme der Gleichverteilung des Werts von $g.\texttt{cond} = h$. In dieser Konstellation ist eine präzisere Annahme $P(h = \mathbf{t}) = \frac{1}{4}$ und $P(h = \mathbf{f}) = \frac{3}{4}$. Durch die Transformation machen wir diese Verteilung explizit erkennbar für unsere Kostenfunktion, und $P_{\mathsf{eval}}(a)$ nimmt ab, aber $P_{\mathsf{eval}}(b)$ vergrößert sich. Die Transformation ist gewinnbringend, weil die Auswertung von h' wie erwähnt bedingt statt unbedingt ist.

Der neue Knoten h' darf allerdings keine höhere Schleifentiefe aufweisen als h, weil sonst die Kosten des Programms steigen. Daher gilt als weitere Vorbedingung für die Transformation, dass $h'.\mathsf{depth} = \max\{q.\mathsf{depth}, a.\mathsf{depth}, b.\mathsf{depth}\} \leq \max\{q.\mathsf{depth}, p.\mathsf{depth}\} = h.\mathsf{depth}$ ist.

Von diesem Muster gibt es insgesamt vier Varianten, abhängig von der Position und dem Wert der booleschen Konstante. Ist q der `false`-Vorgänger von h, befindet sich h' entsprechend vor dem `false`-Eingang von g'. Der andere Eingang von g' wird mit a verbunden, wenn die Konstante den Wert $\mathbf{t}$ hat. In allgemeinerer Form wird diese Regel auch in [Law07] beschrieben.

[4] Hätte h zwei konstante Vorgänger, würde der Knoten durch obige Regeln eliminiert werden. Mit zwei variablen Vorgänger ist die Transformation nicht gewinnbringend.

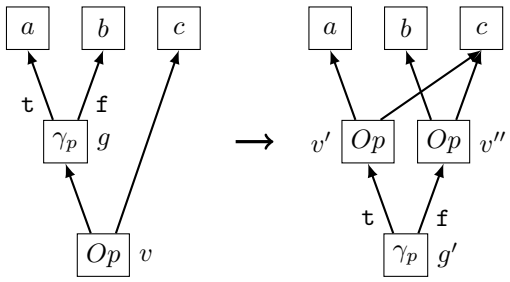


Abbildung 5.4.: Allgemeine γ-Distribution

5.3.2. Die γ-Distribution

Die γ-Operation ist distributiv [WCES94, TSTL09], das heißt es gilt:

$$Op(\gamma(p, a, b), c) = \gamma(p, Op(a, c), Op(b, c))$$

Diese Eigenschaft kann man ausnutzen, um eine Operation v "durch" einen γ-Knoten g zu propagieren. Zunächst duplizieren wir g zu g'. Dann wird v für jeden der beiden Zweige von g' einmal dupliziert. Dabei wird der Eingang, der ursprünglich mit g verbunden war, in den Kopien mit g's `true`- beziehungsweise `false`-Vorgängern verbunden. Wir werden diese Kopien v', v'' im Folgenden auch Spezialisierungen nennen. Die Transformation ist in Abbildung 5.4 für eine binäre Operationen gezeigt. Sie ist aber auch für Operationen anderer Stelligkeiten möglich.

Durch die Duplikationen wächst nur die Anzahl der Knoten. Die Menge der Operationen, die auf jedem Pfad ausgewertet werden müssen, ändert sich nicht.

Mehrere γ-Vorgänger Hat eine Operation v mehrere γ-Knoten $g_i(p, a_i, b_i)$ als Argumente, und haben die g_i das gleiche Prädikat p, dann ist die Distribution parallel über alle g_i möglich. Weiterhin werden nur zwei Spezialisierungen benötigt, die anstelle von g_i mit a_i oder b_i verbunden werden.

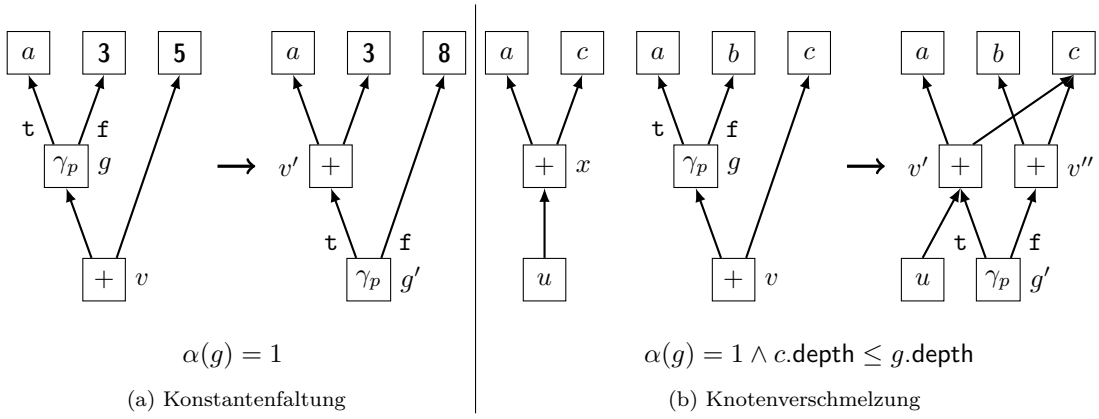
Tupelwerte Die beschriebene γ-Distribution ist zunächst nur für skalare Werte definiert. Soll eine Operation propagiert werden, die Tupelwerte erzeugt, benötigen wir für jedes Element des Tupels einen eigenen neuen γ-Knoten, dessen `true`- und `false`-Eingänge mit einer entsprechenden Projektion aus dem Wert einer Spezialisierung verbunden werden.

Die γ-Distribution kann die Kosten des Programms verbessern, verschlechtern oder unverändert lassen. Im Allgemeinen ist sie daher keine Optimierung. Wir identifizieren nun die verschiedenen Situationen.

Zunächst können wir alle γ-Knoten g mit mehr als einem Verwender, also $\alpha(g) > 1$, ausschließen. Nach der Transformation verbleibt der ursprüngliche Knoten im Graph, und die Kosten verschlechtern sich durch die duplizierten Knoten auf jeden Fall. Im Folgenden gelte also $\alpha(g) = 1$.

Betrachten wir nun die Schleifentiefen der beteiligten Knoten. Die Anwendung der Transformation ist immer kostenverschlechternd, wenn die Schleifentiefe von c größer als die Schleifentiefe von g ist.

Nach der Distribution erhöht sich $g'.\mathsf{depth}$ auf $c.\mathsf{depth} > g.\mathsf{depth}$. Bezogen auf das Programm verschieben wir g' also in eine Schleife. Dies spiegelt sich auch in der Kostenfunktion wieder: Die Auswertungswahrscheinlichkeit von g und g' sind gleich, die Schleifentiefe und damit auch die Kosten von g' sind aber echt größer als die von g. Die Spezialisierungen von v haben nach der Transformation die gleiche Schleifentiefe wie v selbst, die Summe ihrer Auswertungswahrscheinlichkeiten und dementsprechend auch ihrer Kosten ist aber nach Lemma 7 mindestens gleich groß.



$\alpha(g) = 1$

(a) Konstantenfaltung

$\alpha(g) = 1 \wedge c.\mathsf{depth} \leq g.\mathsf{depth}$

(b) Knotenverschmelzung

Abbildung 5.5.: γ-Distribution mit ...

Untersuchen wir die Situation $c.\mathsf{depth} \leq g.\mathsf{depth}$. Wir können die Schleifentiefe von v auch ausdrücken als $v.\mathsf{depth} = \max\{a.\mathsf{depth}, b.\mathsf{depth}, c.\mathsf{depth}, p.\mathsf{depth}\}$. Für die Schleifentiefen der Spezialisierungen gilt $v'.\mathsf{depth} = \max\{a.\mathsf{depth}, c.\mathsf{depth}\}$ und $v''.\mathsf{depth} = \max\{b.\mathsf{depth}, c.\mathsf{depth}\}$. Wir sehen, dass $v'.\mathsf{depth} \leq v.\mathsf{depth}$ und $v''.\mathsf{depth} \leq v.\mathsf{depth}$ ist. Nach Lemma 7 kann aber die Summe der Auswertungswahrscheinlichkeiten der Spezialisierungen größer als die Auswertungswahrscheinlichkeit von v sein, so dass wir keine verbessernde Transformation garantieren können. Ausgehend von dieser Sitation können wir aber die γ-Distribution mit nachfolgenden Optimierungen kombinieren. Darauf gehen wir im nächsten Unterabschnitt ein.

γ-Distribution zur Reduktion der Schleifentiefe GammaLoop

Zunächst betrachten wir aber den Fall, dass eine der Spezialisierungen eine geringere Schleifentiefe erhält, also o.b.d.A $v'.\mathsf{depth} < v''.\mathsf{depth} \leq v.\mathsf{depth}$ ist. Die Kosten von v' sind mindestens um den Faktor $\frac{1}{\mathbb{L}}$ kleiner als die Kosten von v, weil auch die Auswertungswahrscheinlichkeit von v' nach Lemma 8 kleiner oder gleich $P_{\mathsf{eval}}(v)$ ist. Leider ist es für $P_{\mathsf{eval}}(v'') > 1 - \frac{1}{\mathbb{L}}$ möglich, dass sich die Kosten um maximal $\frac{1}{\mathbb{L}}$ verschlechtern, weil dann die Summe der Kosten der Spezialisierungen die Kosten von v übersteigen. Wir führen die Transformation trotzdem immer durch, weil die Verschlechterung aus der Überabschätzung von P_{eval} resultiert[5] und nur in Grenzfällen eintritt.

Lawrence [Law07] beschreibt die "splitting"-Transformation als allgemeinere Form der γ-Distribution. Dabei werden auch Knoten, die den γ-Knoten nicht verwenden, als zusätzlichen Tupelwert unter seine bedingte Auswertung gestellt. Dies ist in der vFirm-Darstellung nicht anwendbar, weil jeder γ-Knoten nur zwischen skalaren Werten auswählt. Knotenverschmelzung und die Eliminationsregel für γ-Knoten mit identischen Vorgängern würden diese Transformation wieder rückgängig machen.

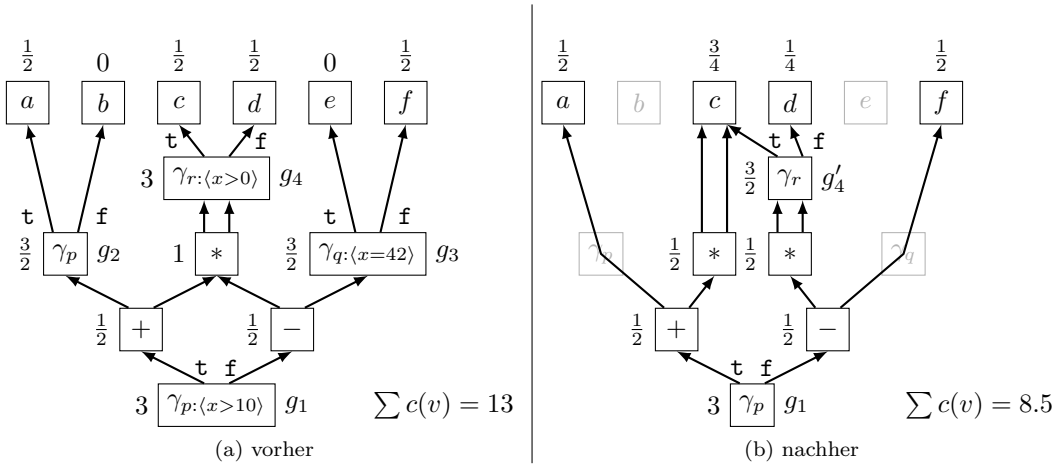
5.3.3. Anwendungen der γ-Distribution

Wir führen im Rahmen dieser Arbeit die Kombinationen der γ-Distribution mit der Konstantenfaltung und Knotenverschmelzung für arithmetisch-logische Operationen durch, und benötigen daher die Distribution von Tupelwerten nicht. Beide Kombinationen werden auch in [Law07] erwähnt.

Die folgenden Transformationen und Kostenrechnungen sind für Knoten mit genau einem γ-Vorgänger beschrieben. Hat ein Knoten mehrere γ-Vorgänger mit demselben Bedingungsknoten, passiert die Distribution durch alle parallel, was sich zusätzlich positiv auf die Kosten des Programms auswirkt.

Konstantenfaltung Die Vorbedingungen für die Transformation lauten: Eine binäre, arithmetisch-logische Operation v hat einen γ-Knoten g sowie eine Konstante als Argumente. Wenn g mindestens einen konstanten Vorgänger hat und selbst keine weiteren Verwender außer v, ist die Transformation

[5] Mit der exakten Auswertungswahrscheinlichkeit würde Lemma 7 statt der $\geq$-Beziehung die Gleichheit $P'_{\mathsf{eval}}(v') + P_{\mathsf{eval}}(v'') = P'_{\mathsf{eval}}(v)$ liefern.



(a) vorher (b) nachher

Abbildung 5.6.: Beispielgraph für die Elimination von redundanten γ-Knoten. Links / oberhalb der Knoten sind ihre Kosten angegeben.

anwendbar. Da die Schleifentiefe von Konstanten 0 ist, gilt automatisch unsere Forderung $\mathsf{k}.\mathsf{depth} \leq g.\mathsf{depth}$.

Diese Situation ist in Abbildung 5.5a beispielhaft für eine Addition gezeigt. Für unäre Operationen reicht es, wenn der Operand ein entsprechender γ-Knoten ist.

Die Kosten für die nicht-konstanten Argumente von g (im Beispiel der Knoten a) und die Anzahl und Schleifentiefe der Knoten im Graph bleiben gleich. Allerdings gilt $c(v') \leq c(v)$ nach Lemma 8, weil die Auswertung von v' nun von g' abhängig ist. Insgesamt verbessert die Transformation also die Kosten des Programms.

Knotenverschmelzung Für die Anwendung der Knotenverschmelzung erzeugen wir die beiden Spezialisierungen temporär und überprüfen, ob mindestens eine von ihnen schon im Graph vorhanden ist. Um eine Verbesserung der Kosten zu erhalten, müssen wir prüfen, dass der γ-Knoten keine weiteren Verwender hat und dass $c.\mathsf{depth} \leq g.\mathsf{depth}$ gilt. Ist das der Fall, führen wir die γ-Distribution durch. Abbildung 5.5b zeigt die Transformation. Aus Gründen der Übersichtlichkeit sind die Knoten a und c zweimal dargestellt.

Die Auswertungswahrscheinlichkeiten der Knoten a, b und c und damit ihre Kosten ändern sich nicht. Für v' gelten die gleichen Überlegungen wie bei Durchführung einer normalen Knotenverschmelzung von x und v', sie ist kostenneutral oder -verbessernd. Im Vergleich zum eliminierten Knoten v findet die Auswertung von v' sowie v'' in Abhängigkeit von p statt, es gilt also $P_{\mathsf{eval}}(v'') \leq P_{\mathsf{eval}}(v)$ nach Lemma 8 und somit $c(v'') \leq c(v)$.

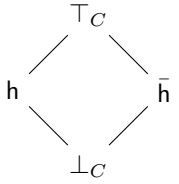
Die Spezialisierungen von v verursachen zusammen Kosten, die kleiner oder gleich den Kosten von v sind. Aufgrund der Vorbedingungen kann g eliminiert werden, und g', v' und v'' haben keine größeren Schleifentiefen als die ursprünglichen Knoten. Die Anwendung dieser Transformation verbessert die Kosten in unserem Modell oder lässt sie zumindest unverändert.

5.4. Elimination von redundanten γ-Knoten

Mit den lokalen Optimierungen des vorigen Abschnitts konnten wir in bestimmten Fällen γ-Knoten auswerten und eliminieren und dadurch den impliziten Steuerfluss unseres Programms vereinfachen. γ-Knoten mit konstantem Prädikat sind jedoch selten.

Wir wissen, dass die `true`- und `false`-Vorgänger eines γ-Knotens mit Prädikat p unter der Bedingung "p gilt" beziehungsweise "p gilt nicht" ausgewertet werden.

Betrachten wir den Beispielgraph in Abbildung 5.6a. Die Auswertung von g_2 wird nur über Knoten $+$ angefordert, wenn bei g_1 die Bedingung p zu t ausgewertet wurde. Damit kennen wir p's Wert während



(a) Hasse-Diagramm

$$\mathsf{h} \sqcup \bar{\mathsf{h}} = \top_C \qquad \mathsf{h} \sqcap \bar{\mathsf{h}} = \bot_C$$
$$x \sqcup \bot_C = x \qquad x \sqcap \bot_C = \bot_C$$
$$x \sqcup \top_C = \top_C \qquad x \sqcap \top_C = x$$
$$x \sqcup x = x \qquad x \sqcap x = x$$

(b) Suprema und Infima

Abbildung 5.7.: Verband $L_C = (B, \sqsubseteq)$

der Auswertung von g_2 und können + stattdessen direkt mit a verbinden. Der Knoten g_3 verwendet zwar einen anderen Bedingungsknoten als g_1, dessen (negierte) Bedingung impliziert aber, dass die Bedingung q nie wahr sein kann. Der Knoten f kann also direkt mit der Substraktion verbunden werden. Die Bedingung r von g_4 könnte von p impliziert werden, diese Information ist jedoch nicht bei g_4 verfügbar. Dupliziert man die Multiplikation und g_4, so lässt sich eine Kopie des γ-Knotens eliminieren. Wir erhalten den Graph in Abbildung 5.6b.

In diesem Abschnitt stellen wir ein Verfahren zur Elimination redundanter γ-Knoten mittels der drei hier exemplarisch gezeigten Transformationen vor.

Dazu bestimmen wir für jeden Knoten, welche Bedingungen bei jeder Auswertung gelten, und bezeichnen sie als *Pfadinformationen*.

Enthalten die Pfadinformationen eines γ-Knotens g eine Aussage über sein Prädikat p, so können wir den Bedingungseingang von g statt mit p mit einer Konstante verbinden.

5.4.1. Analyse der Pfadinformationen

Ein lebendiger Knoten v ist über mindestens einen Pfad mit dem `Return`-Knoten verbunden.

Enthält einer dieser Pfade W einen γ-Knoten $v_g = \gamma(p, v_{g+1}, b)$, gilt die Bedingung p bei der Auswertung der nachfolgenden $v_i, i > g$ auf diesem Pfad. Analog gilt die Negation von p, falls $v_g = \gamma(p, a, v_{g+1})$ ist.

Uns interessiert nun, welche Bedingungen auf *allen* Pfaden gelten, die v erreichen, weil dann statisch bekannt ist, dass der Knoten immer unter diesen Bedingungen ausgewertet wird.

Wir wollen die Bedingungen, die auf allen Pfaden gelten, mittels einer Datenflussanalyse in einem monotonen Rahmenwerk nach dem Schema von Nielson und Nielson (siehe 2.2) ermitteln[6].

Ein solches monotones Rahmenwerk besteht aus einem vollständigen Verband und einer Menge von monotonen Transferfunktionen. Beginnen wir mit der Definition des Verbands.

Verband Wir definieren zunächst die Menge $B = \{\bot_C, \bar{\mathsf{h}}, \mathsf{h}, \top_C\}$ und darauf die in Abbildung 5.7a im Hasse-Diagramm eingezeichnete Halbordnung $\sqsubseteq$.

Lemma 9. $L_C = (B, \sqsubseteq)$ ist ein vollständiger Verband.

Beweis. Wir sehen leicht, dass für alle Elemente von B das Supremum und das Infimum existieren, da $h \sqcup \bar{h} = \top_C$, $h \sqcap \bar{h} = \bot_C$ und alle anderen Elemente direkt oder transitiv in Relation stehen. In Abbildung 5.7b sind alle Suprema und Infima aufgelistet. L_C erfüllt somit die Verbandseigenschaft; da B endlich ist, ist L_C gleichzeitig ein vollständiger Verband. □

Definition 20 (Einfacher Analyseverband)**.**

$$L_C^{\Pi} = \prod_{p \in Conds} L_C$$

[6] Alternativ könnte man die Bedingungen aus den Gatingbedingungen ablesen, die Datenflussanalyse hat jedoch den Vorteil, dass wir eine korrekte Analyse durch Konstruktion erhalten. Außerdem ist die Analyse der implizierten Bedingungen leichter zu formulieren.

definiert den Produktverband, in dem der Wert jedes Bedingungsknotens durch ein Element von L_C repräsentiert wird.

Für ein $l \in L_C^\Pi$ bezeichnet $l[p]$ die Komponente eines Bedingungsknotens $p \in \mathit{Conds}$.

L_C^Π ist ein Verband, weil das Produkt von Verbänden wiederum ein Verband ist, dessen Relation und Operationen komponentenweise definiert sind (siehe 2.2.1).

Dieses Konstrukt ist folgendermaßen zu interpretieren: Jedem Bedingungsknoten ist ein Verbandselement zugeordnet, das angibt, ob die Bedingung gilt (h), nicht gilt ($\bar{\mathsf{h}}$), ob ihr Wert unbekannt ($\top_C$) oder widersprüchlich ist ($\bot_C$).

Der Produktverband hat ein größtes Element $\top_C^\Pi = \top_C \times \cdots \times \top_C$ und ein kleinstes Element $\bot_C^\Pi = \bot_C \times \cdots \times \bot_C$. Die Notation

$$\top_C^\Pi[p \mapsto l] = \top_C \times \cdots \times \underbrace{l}_{p} \times \cdots \times \top_C$$

mit $p \in \mathit{Conds}, l \in L_C$ beschreibt ein Element des Produktverbands, in dem die p zugehörige Komponente den Wert l hat, und alle anderen Komponenten $\top_C$ sind.

Für die Analyse wird jedem Knoten v ein Element des Verbands L_C^Π als zusätzliches Attribut $v.\mathsf{pi}$ zugeordnet, das die Pfadinformationen repräsentiert.

Gleichungen und Transferfunktionen Da in der vFirm-Darstellung das Konzept der Grundblöcke nicht anwendbar ist, operiert unsere Analyse stattdessen auf einzelnen Knoten.

Um die Formulierung zu erleichtern, fügen wir vor den `true`- und `false`-Eingängen aller γ-Knoten jeweils einen Knoten vom neuen Knotentyp `Path` ein. Ein solcher `Path`-Knoten y hat selbst keine Funktion, wird aber mit einem Attribut $y.\mathsf{branch}$ annotiert. Dabei handelt es sich um ein Tupel (p, l_C) aus einem Bedingungsknoten $p \in \mathit{Conds}$ und einem Element des booleschen Verbands $l_C \in \{\mathsf{h}, \bar{\mathsf{h}}\} \subseteq L_C$, welches angibt, ob sich der `Path`-Knoten vor dem `true`- beziehungsweise `false`-Eingang des γ-Knotens befindet.

Der Analyse liegt ein Fluss zugrunde, der den Datenabhängigkeitskanten entspricht. Konkret bedeutet das für einen Knoten v, dass wir die Informationen seiner Verwender zusammenführen müssen, dann eine Transferfunktion auf die Vereinigung anwenden und deren Ergebnis als neuer Wert für das Attribut $v.\mathsf{pi}$ speichern.

Die Gleichungen der Analyse für einen Knoten v lauten:

$$\begin{aligned} PI_\circ(v) &= \bigsqcup \{PI_\bullet(u) \mid u \to v \in E\} \sqcup \iota_v \\ \iota_v &= \begin{cases} \top_C^\Pi & \text{wenn } v.\mathtt{op} = \mathtt{Return} \\ \bot_C^\Pi & \text{sonst} \end{cases} \\ PI_\bullet(v) &= f_v^{PI}(PI_\circ(v)) \end{aligned}$$

Für einen Knoten v ist die Transferfunktion gegeben durch[7]:

$$f_v^{PI}(l) = \begin{cases} l \sqcap \top_C^\Pi[q \mapsto m] & \text{wenn } v.\mathsf{op} = \mathtt{Path} \wedge v.\mathsf{branch} = (q, m) \\ l & \text{sonst} \end{cases}$$

Für alle $v \in V, v.\mathtt{op} \neq \mathtt{Path}$ ist f_v^{PI} trivialerweise monoton. Sei v also ein `Path`-Knoten und $(q, m) = v.\mathsf{branch}$. Es ist zu zeigen, dass $\forall a, b \in L_C^\Pi : a \sqsubseteq b \implies f_v^{PI}(a) \sqsubseteq f_v^{PI}(b)$ ist. Wir bilden das komponentenweise Infimum mit $\top_C^\Pi[q \mapsto m]$. Bis auf die Komponente, die q repräsentiert, werden die Infima mit $\top_C$ gebildet und die entsprechenden Komponenten daher nicht verändert. Aus der Annahme folgt, dass $a[q] \sqsubseteq b[q]$ ist. Dann ist mit Lemma 5 auch $a[q] \sqcap m \sqsubseteq b[q] \sqcap m$ und die Monotonie der Transferfunktion ist gezeigt.

[7] Die Definition gilt zunächst für azyklische Graphen. Für zyklischen Graphen erfolgt später noch eine Erweiterung.

Rahmenwerk Wir haben ein monotones Rahmenwerk definiert. L_C^Π ist vollständig und endlich erfüllt somit die Ascending Chain Condition. Die komponentenweise Supremumsoperation wird benutzt, um Informationen von mehreren eingehenden Pfaden zusammenzuführen.

Die Menge der Funktionen $f_v, v \in V$ besteht aus monotonen Funktionen für jeden Knoten, enthält die Identitätsfunktion und ist unter der Komposition abgeschlossen.

Instanz Für einen konkreten vFirm-Graphen besteht eine Instanz unserer Analyse aus dem Verband und den Transferfunktionen des Rahmenwerks. Die Zuordnung der Transferfunktionen zu Knoten wurde bereits bei der Definition erledigt. Einen Sonderfall stellt nur der `Return`-Knoten dar; er wird mit dem Verbandselement $\top_C^\Pi$ initialisiert.

Durchführung Die Analyse wird nun, unterstützt durch eine Arbeitsliste, bis zum Fixpunkt iteriert, dessen Existenz durch das Rahmenwerk gewährleistet ist.

Beispiel In Abbildung 5.8 ist unser um die `Path`-Knoten erweiterter Beispielgraph zusammen mit dem resultierenden Gleichungssystem dargestellt. Da der Graph azyklisch ist, können wir die Lösung des Gleichungssystems direkt angeben. Für einen Knoten v gilt $v.\mathsf{pi} = PI_\bullet(v)$. Da sich $PI_\circ$ und $PI_\bullet$ nur für `Path`-Knoten unterscheiden, ist bei allen anderen Knotentypen nur eine gemeinsame Gleichung angegeben. Beim Knoten $*$ fließen unterschiedliche Informationen über den Bedingungsknoten p zusammen. Das Ergebnis der Zusammenführung ist daher $\top_C$ ("Unwissen") für die zu p gehörige Komponente.

Analyse in zyklischen Graphen Für die Analyse in zyklischen Teilgraphen gilt es, zwei Besonderheiten zu beachten.

Wir müssen bei der Analyse eine Unterapproximation des Ergebnisses zulassen. Das bedeutet, dass Knoten, die im Rahmen der Analyse noch nicht besucht wurden, bei der Kombination der Datenflussinformationen der Verwender ausgespart werden. Die konservative Alternative wäre, dort keine Information, also das Verbandselement $\top_C^\Pi$ anzunehmen. So würden allerdings keine Pfadinformationen in zyklische Teilgraphen hinein propagiert werden.

Dies würde beispielsweise bei einem θ-Knoten t passieren, der vom `Return`-Knoten aus ausschließlich über den `true`- oder `false`-Eingang eines γ-Knotens erreichbar ist. Dann wird der gesamte zyklische Teilgraph nur bedingt ausgewertet. Diese Pfadinformation geht aber bei t verloren, weil dessen andere Verwender (noch) keine Informationen liefern.

Für die ignorierten Knoten wird sichergestellt, dass sie sich auf der Arbeitsliste befinden und später besucht werden. Anschließend werden auch die Knoten, deren Datenflussinformation unter der Auslassung einiger Verwender berechnet wurden, aktualisiert. So ist garantiert, dass die Analyse nicht vor dem Erreichen eines konservativen Ergebnisses endet.

Betrachten wir nun die Situation in Abbildung 5.9. Das Prädikat p der beiden γ-Knoten habe die gleiche Schleifentiefe wie der θ-Knoten. Auf den ersten Blick sieht es so aus, als ob wir bei g_2 die Bedingung durch eine Konstante ersetzen könnten.

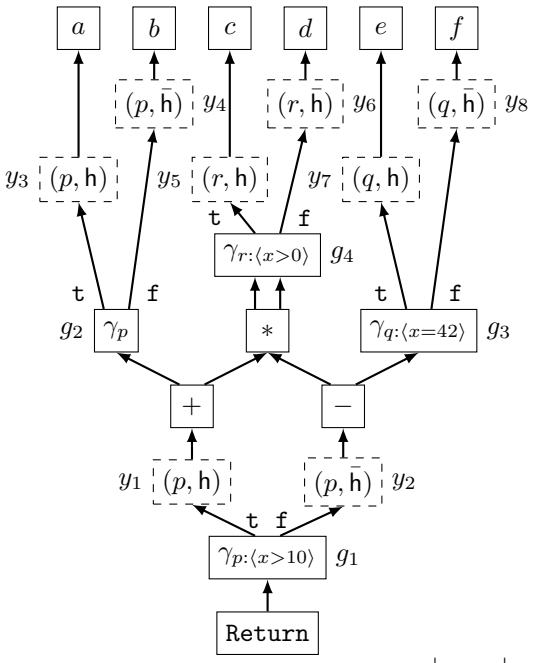
Das Problem besteht hier darin, dass g_1 und g_2 unterschiedliche Listenelemente der von p produzierten Werte verwenden. Wenn g_1 den Knoten t in der Iteration i auswertet, fordert dieser die Auswertung von g_2 in der Iteration $i-1$ an (vgl. Definition 10). Wir dürfen bei g_2 also nicht die Kenntnis des Werts von p annehmen.

Die Transferfunktion für θ-Knoten t wird daher erweitert:

$$f_t^{PI}(l) = \mathrm{filter}_C(l) \quad \text{mit}$$

$$\mathrm{filter}_C(l) = \prod_{p \in Conds} \begin{cases} l[p] & \text{wenn } p.\mathsf{depth} < t.\mathsf{depth} \\ \top_C & \text{sonst} \end{cases}$$

Diese Funktion ist weiterhin monoton. Seien $a, b \in L_C^\Pi$ mit $a \sqsubseteq b$. In den Verbandselementen werden Komponenten für Bedingungsknoten mit zu großer Schleifentiefe gleichermaßen auf $\top_C$ gesetzt. Alle



	$\downarrow p$	$\downarrow q$	$\downarrow r$
$PI_\bullet(\texttt{Return}) = \top_C^\Pi$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_{\circ/\bullet}(g_1) = PI_\bullet(\texttt{Return})$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_\circ(y_1) = PI_\bullet(g_1)$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_1) = PI_\circ(y_1) \sqcap \top_C^\Pi[p \mapsto \mathsf{h}]$	$= \mathsf{h}$	$\times \top_C$	$\times \top_C$
$PI_\circ(y_2) = PI_\bullet(g_1)$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_2) = PI_\circ(y_1) \sqcap \top_C^\Pi[p \mapsto \bar{\mathsf{h}}]$	$= \bar{\mathsf{h}}$	$\times \top_C$	$\times \top_C$
$PI_{\circ/\bullet}(+) = PI_\bullet(y_1)$	$= \mathsf{h}$	$\times \top_C$	$\times \top_C$
$PI_{\circ/\bullet}(-) = PI_\bullet(y_2)$	$= \bar{\mathsf{h}}$	$\times \top_C$	$\times \top_C$
$PI_{\circ/\bullet}(g_2) = PI_\bullet(+)$	$= \mathsf{h}$	$\times \top_C$	$\times \top_C$
$PI_{\circ/\bullet}(g_3) = PI_\bullet(-)$	$= \bar{\mathsf{h}}$	$\times \top_C$	$\times \top_C$
$PI_{\circ/\bullet}(*) = PI_\bullet(+) \sqcup PI_\bullet(-)$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_{\circ/\bullet}(g_4) = PI_\bullet(*) \sqcup PI_\bullet(*)$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_\circ(y_3) = PI_\bullet(g_2)$	$= \mathsf{h}$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_3) = PI_\circ(y_3) \sqcap \top_C^\Pi[p \mapsto \mathsf{h}]$	$= \mathsf{h}$	$\times \top_C$	$\times \top_C$
$PI_\circ(y_4) = PI_\bullet(g_2)$	$= \mathsf{h}$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_4) = PI_\circ(y_4) \sqcap \top_C^\Pi[p \mapsto \bar{\mathsf{h}}]$	$= \bot_C$	$\times \top_C$	$\times \top_C$
$PI_\circ(y_5) = PI_\bullet(g_4)$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_5) = PI_\circ(y_5) \sqcap \top_C^\Pi[r \mapsto \mathsf{h}]$	$= \top_C$	$\times \top_C$	$\times \mathsf{h}$
$PI_\circ(y_6) = PI_\bullet(g_4)$	$= \top_C$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_6) = PI_\circ(y_6) \sqcap \top_C^\Pi[r \mapsto \bar{\mathsf{h}}]$	$= \top_C$	$\times \top_C$	$\times \bar{\mathsf{h}}$
$PI_\circ(y_7) = PI_\bullet(g_3)$	$= \bar{\mathsf{h}}$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_7) = PI_\circ(y_7) \sqcap \top_C^\Pi[q \mapsto \mathsf{h}]$	$= \bar{\mathsf{h}}$	$\times \mathsf{h}$	$\times \top_C$
$PI_\circ(y_8) = PI_\bullet(g_3)$	$= \bar{\mathsf{h}}$	$\times \top_C$	$\times \top_C$
$PI_\bullet(y_8) = PI_\circ(y_8) \sqcap \top_C^\Pi[q \mapsto \bar{\mathsf{h}}]$	$= \bar{\mathsf{h}}$	$\times \bar{\mathsf{h}}$	$\times \top_C$

Abbildung 5.8.: Datenflussgleichungen der Pfadinformationsanalyse für den Beispielgraph. Die `Path`-Knoten sind gestrichelt dargestellt.

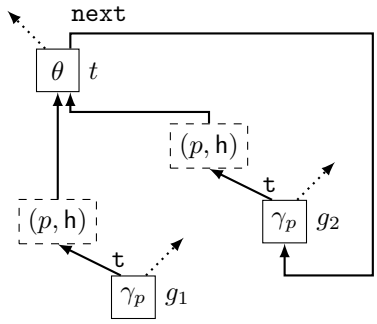
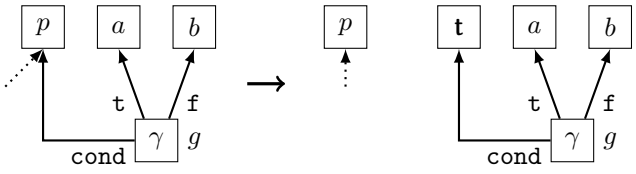


Abbildung 5.9.: Problematische Situation in einem zyklischen Teilgraph: Der Bedingungsknoten p habe die gleiche Schleifentiefe wie t. g_2's Prädikat darf nicht ersetzt werden.



$$g.\mathsf{pi}[p] = \mathsf{h}$$

Abbildung 5.10.: Transformation auf Grundlage der Pfadinformationen

anderen Komponenten bleiben unverändert. Aus der Annahme folgt dann, dass $\text{filter}_C(a) \sqsubseteq \text{filter}_C(b)$ ist.

Transformation: Auswertung von Bedingungsknoten ERG-Eval

Wir definieren eine Funktion $\text{eval}_C(p, l) : \mathit{Conds} \times L_C^{\Pi} \to L_C$, die einen Bedingungsknoten p im Kontext der Pfadinformation l auswertet:

$$\text{eval}_C(p, l) = l[p]$$

Sei $g = \gamma(p, a, b)$. Wenn $\text{eval}_C(p, g.\mathsf{pi}) = \mathsf{h}$ oder $\text{eval}_C(p, g.\mathsf{pi}) = \bar{\mathsf{h}}$ ist, können wir g's `cond`-Eingang anstelle von p mit dem konstanten Knoten **t** beziehungsweise **f** verbinden. Abbildung 5.10 zeigt diese Transformation. Der Bedingungsknoten p bleibt im Graph bestehen, weil er zumindest von dem γ-Knoten verwendet wird, der die Pfadinformation produziert hat. Die Auswertungswahrscheinlichkeiten im Graph ändern sich aufgrund ihrer Definition nicht. g wird dann durch eine spätere Anwendung der lokalen Optimierung (Abb. 5.2b) eliminiert. Deren Kostenbilanz gilt demnach auch für diese Transformation.

Ein Knoten v mit $\exists q \in \mathit{Conds} : v.\mathsf{pi}[q] = \bot_C$ wird nie ausgewertet, weil die Pfadinformationen einen Widerspruch enthalten. Es handelt sich also um unerreichbaren Code. Nach der Elimination aller γ-Knoten mit konstantem Prädikat haben solche Knoten keine Verwender mehr. Die Knoten sind tot und werden aus der Darstellung entfernt.

In unserem Beispielgraph aus Abbildung 5.6a gilt für die Auswertung von p in g_2's Pfadinformation $\text{eval}_C(p, g_2.\mathsf{pi}) = g_2.\mathsf{pi}[p] = \mathsf{h}$.

Korrektheit Nach der Transformation bleiben die berechneten Pfadinformationen eine konservative Approximation. Konnte beispielsweise p zu **t** ausgewertet werden, ist weiterhin $a.\mathsf{pi}[p] = \mathsf{h}$. Für den anderen Zweig gilt $a.\mathsf{pi}[p] = \bot_C$, b ist also unerreichbar. Beim Zusammenführen der Pfadinformationen werden unerreichbare Verwender automatisch ignoriert, da $\bot_C$ das neutrale Element der Supremumsoperation ist.

Der Fall, dass p durch **f** ersetzt wurde, verhält sich analog. Alle anderen Komponenten $a.\mathsf{pi}[q]$ beziehungsweise $b.\mathsf{pi}[q], p \neq q \in \mathit{Conds}$ sind von der Transformation nicht betroffen.

5.4.2. Analyse von Implikationen

Um noch mehr γ-Knoten auswerten zu können, wollen wir nun die Analyse verallgemeinern, indem wir auch Bedingungen ersetzen, die von einer anderen Bedingung impliziert werden. Wissen wir beispielsweise, dass x ein Integerknoten ist und eine Bedingung $x \leq 17$ gilt, können wir Bedingungen wie $x \leq 21$ oder $x = 33$ zu t beziehungsweise f auswerten. In unserem Beispielgraph aus Abbildung 5.6a gilt etwa $\bar{p} \Rightarrow \bar{q}$ sowie $p \Rightarrow r$.

Ähnliche Verfahren zur Ausnutzung von Implikationen zwischen Vergleichen wurden auf anderen Zwischendarstellungen unter anderem von [KWB+05] und [MW95] beschrieben. In [Tra01] und [BE96] werden ebenfalls Intervalle zur Beschreibung der Wertebereiche von skalaren Operationen verwendet. [Har77] leitet Wertebereiche von Vergleichsoperationen ab.

Definition 21 (Beschränkungsknoten)**.**

$$\mathit{Constraints} = \{p \in \mathit{Conds} \mid \exists x \in V \ : \ p = \mathtt{Cmp}(x, \mathsf{k}) \wedge x.\mathsf{mode} = \mathbb{I} \wedge p.\mathsf{rel} \in \{<, \leq, =, \neq, \geq, >\}\}$$

definiert die Menge der *Beschränkungsknoten* als Teilmenge der Bedingungsknoten. Ein Beschränkungsknoten ist ein Vergleich eines Integerknotens x mit einer Konstante k gemäß der Relation $\prec$. Für ein $p \in \mathit{Constraints}$ notieren wir kürzer $p = \langle x \prec \mathsf{k} \rangle$.

$$\mathit{CValues} = \{x \mid \exists \ \langle x \prec \mathsf{k} \rangle \in \mathit{Constraints}\}$$

definiert die Menge der *beschränkten Integerknoten*.

Verband Anstatt die Implikationen zwischen den Beschränkungsknoten direkt zu bestimmen, nutzen wir folgenden Zusammenhang aus [Dav08]: Sei x ein Integerknoten und $p = \langle x \prec \mathsf{k} \rangle, q = \langle x \prec' \mathsf{l} \rangle \in \mathit{Constraints}$.

$$p \Rightarrow q \quad \Leftrightarrow \quad \{y \in \mathbb{I} \mid y \prec \mathsf{k}\} \quad \subseteq \quad \{z \in \mathbb{I} \mid z \prec' \mathsf{l}\}$$

Wir betrachten also die Mengen von Integerzahlen, für die die von den Beschränkungsknoten repräsentierten Vergleiche wahr sind. Die Potenzmenge der Integerzahlen ist mit ihren $2^{|\mathbb{I}|}$ Elementen zusammen mit der Mengeninklusion ein riesiger, aber dennoch endlicher Verband und damit ein Kandidat für unsere Analyse. Aufgrund des hohen Aufwands, beliebige Teilmengen von Integerzahlen zu verwalten, betrachten wir stattdessen das Intervall, in dem alle möglichen Werte eines Knotens liegen. Dies ist eine konservative Approximation.

Um die Aussage "p impliziert q" zu erhalten fordern wir, dass alle Werte, die p erfüllen, auch q erfüllen. Die Überabschätzung macht aus, dass das p repräsentierende Intervall Werte enthalten kann, für die p nicht wahr ist, von denen wir aber fordern, dass sie q erfüllen, um die Implikationsaussage zu treffen.

Definition 22 (Menge der Intervalle)**.**

$$I = \{[x_l, x_u] \mid x_l, x_u \in \mathbb{I}, \ x_l \leq x_u\} \cup \{\emptyset\}$$

ist die Menge der Intervalle von Integerzahlen mit der unteren Schranke x_l und der oberen Schranke x_u sowie dem speziellen leeren Intervall.

Lemma 10. $L_X = (I, \subseteq)$ ist ein vollständiger Verband.

Beweis. Die Mengeninklusion $\subseteq$ ist unabhängig vom zugrunde liegenden Mengensystem reflexiv, antisymmetrisch und transitiv, folglich also eine Halbordnung [Dei10].

Seien $r, s \in I$ mit $r = \emptyset \vee r = [r_l, r_u]$ und $s = \emptyset \vee s = [s_l, s_u]$ und o.B.d.A $r_l \leq s_l$ wenn $r, s \neq \emptyset$.[8]

[8] Bildlich wäre r also das "linke" Intervall.

Wir definieren:

$$r \sqcup s = \begin{cases} r & \text{wenn } s = \emptyset \\ s & \text{wenn } r = \emptyset \\ [\min(r_l, s_l), \max(r_u, s_u)] & \text{sonst} \end{cases} \quad \text{(Supremum)}$$

$$r \sqcap s = \begin{cases} \emptyset & \text{wenn } r = \emptyset \vee s = \emptyset \\ \emptyset & \text{wenn } r_u < s_l \\ [\max(r_l, s_l), \min(r_u, s_u)] & \text{sonst} \end{cases} \quad \text{(Infimum)}$$

Es ist zu zeigen, dass $r \sqcup s$ die kleinste obere Schranke und $r \sqcap s$ die größte untere Schranke von r, s ist.

Man sieht leicht, dass für $r, s \neq \emptyset$ gilt:

$$r \subseteq s \quad \Leftrightarrow \quad \forall x \in r : x \in s \quad \Leftrightarrow \quad r_l \geq s_l \wedge r_u \leq s_u$$

Zeigen wir zunächst, dass $r \sqcup s$ eine obere Schranke für r, s ist.

$$\begin{array}{lllll} \text{Fall 1:} & r \neq \emptyset, s = \emptyset & \rightsquigarrow & r \sqcup s = r \Rightarrow r \subseteq r, \emptyset \subseteq r & \checkmark \\ \text{Fall 2:} & r = \emptyset, s \neq \emptyset & \rightsquigarrow & r \sqcup s = s \Rightarrow \emptyset \subseteq s, s \subseteq s & \checkmark \\ \text{Fall 3:} & r, s \neq \emptyset & \rightsquigarrow & r \sqcup s = [\min(r_l, s_l), \max(r_u, s_u)] & \Rightarrow \\ & r \subseteq r \sqcup s & \Leftrightarrow & r_l \geq \min(r_l, s_l) \wedge r_u \leq \max(r_u, s_u) & \checkmark \\ & s \subseteq r \sqcup s & \Leftrightarrow & s_l \geq \min(r_l, s_l) \wedge s_u \leq \max(r_u, s_u) & \checkmark \end{array}$$

Um zu zeigen, dass $r \sqcup s$ die kleinste obere Schranke ist, nehmen wir an, es existiert ein $t \in I$ mit $r, s \subseteq t \subseteq r \sqcup s$ und $t \neq r \sqcup s$.

$$\begin{array}{lllll} \text{Fall 1:} & r \neq \emptyset, s = \emptyset & \rightsquigarrow & r \subseteq t \subseteq r \sqcup s = r & \Rightarrow \quad t = r \sqcup s \quad \lightning \\ \text{Fall 2:} & r = \emptyset, s \neq \emptyset & \rightsquigarrow & s \subseteq t \subseteq r \sqcup s = s & \Rightarrow \quad t = r \sqcup s \quad \lightning \\ \text{Fall 3:} & r, s \neq \emptyset & \rightsquigarrow & r \sqcup s = [\min(r_l, s_l), \max(r_u, s_u)], \quad t = [t_l, t_u] & \\ & & \Rightarrow & (t_l \leq r_l \wedge t_u \geq r_u) \wedge (t_l \leq s_l \wedge t_u \geq s_u) \wedge & \\ & & & (t_l \geq \min(r_l, s_l) \wedge t_u \leq \max(r_u, s_u)) & \\ & & \Leftrightarrow & (\min(r_l, s_l) \leq t_l \leq r_l, s_l) \wedge (r_u, s_u \leq t_u \leq \max(r_u, s_u)) & \\ & & \Leftrightarrow & t_l = \min(r_l, s_l) \wedge t_u = \max(r_l, s_l) & \\ & & \Leftrightarrow & t = r \sqcup s \quad \lightning & \end{array}$$

In allen drei Fällen stoßen wir auf einen Widerspruch zur Annahme. Damit ist gezeigt, dass $\sqcup$ die Supremumsoperation definiert.

Weisen wir nun nach, dass $r \sqcap s$ die größte untere Schranke von r, s definiert. $r \sqcap s$ ist eine untere Schranke für r, s, da:

$$\begin{array}{lllll} \text{Fall 1:} & r = \emptyset \vee s = \emptyset & \rightsquigarrow & r \sqcap s = \emptyset \Rightarrow \emptyset \subseteq r, s & \checkmark \\ \text{Fall 2:} & r, s \neq \emptyset, r_u < s_l & \rightsquigarrow & r \sqcap s = \emptyset \Rightarrow \emptyset \subseteq r, s & \checkmark \\ \text{Fall 3:} & r, s \neq \emptyset, r_u \geq s_l & \rightsquigarrow & r \sqcap s = [\max(r_l, s_l), \min(r_u, s_u)] & \Rightarrow \\ & r \sqcap s \subseteq r & \Leftrightarrow & \max(r_l, s_l) \geq r_l \wedge \min(r_u, s_u) \leq r_u & \checkmark \\ & r \sqcap s \subseteq s & \Leftrightarrow & \max(r_l, s_l) \geq s_l \wedge \min(r_u, s_u) \leq s_u & \checkmark \end{array}$$

Um zu zeigen, dass keine größere untere Schranke als $r \sqcap s$ existiert, nehmen wir das Gegenteil an, also $\exists t \in I$ mit $r \sqcap s \subseteq t \subseteq r, s$ und $t \neq r \sqcap s$.

$$
\begin{array}{lllll}
\text{Fall 1:} & r = \emptyset \vee s = \emptyset & \rightsquigarrow & r \sqcap s = \emptyset \subseteq t \subseteq \emptyset \quad \Rightarrow \quad t = r \sqcap s \quad \lightning \\
\text{Fall 2:} & r, s \neq \emptyset, r_u < s_l & \rightsquigarrow & r \sqcap s = \emptyset \subseteq t \subseteq r, s \quad \Rightarrow \\
& \text{Fall 2a:} & & t = \emptyset \quad \Rightarrow \quad t = r \sqcap s \quad \lightning \\
& \text{Fall 2b:} & & t = [t_l, t_u] \\
& & \Rightarrow & (t_l \geq r_l \wedge t_u \leq r_u) \wedge (t_l \geq s_l \wedge t_u \leq s_u) \\
& & \Rightarrow & t_u \leq r_u < s_l \leq t_l \\
& & \Rightarrow & t_u < t_l \quad \Rightarrow \quad t \notin I \quad \lightning \\
\text{Fall 3:} & r, s \neq \emptyset, r_u \geq s_l & \rightsquigarrow & r \sqcap s = [\max(r_l, s_l), \min(r_u, s_u)] \subseteq t \subseteq r, s \quad t = [t_l, t_u] \\
& & \Rightarrow & (t_l \geq r_l \wedge t_u \leq r_u) \wedge (t_l \geq s_l \wedge t_l \leq s_u) \wedge \\
& & & (\max(r_l, s_l) \geq t_l \wedge \min(r_u, s_u) \leq t_u) \\
& & \Leftrightarrow & t_l = \max(r_l, s_l) \wedge t_u = \min(r_u, s_u) \quad \Rightarrow \quad t = r \sqcap s \quad \lightning
\end{array}
$$

Auch hier führen alle Fälle zum Widerspruch zur Annahme. Damit ist gezeigt, dass $\sqcap$ die Infimumsoperation definiert.

Da für alle $r, s \in I$ ein Supremum und ein Infimum existieren, ist die Verbandseigenschaft für L_X gezeigt. Die Grundmenge I ist endlich, da die Menge der Integerzahlen endlich ist. L_X ist dadurch automatisch vollständig. □

In L_X ist das größte Element $\top_X = [\mathbb{I}_{\min}, \mathbb{I}_{\max}]$ und das kleinste Element $\bot_X = \emptyset$. Wie beim booleschen Verband steht auch hier $\top_X$ für Unwissen (der repräsentierte Knoten kann alle Werte annehmen) und $\bot_X$ für Widerspruch (der Knoten kann keinen Wert annehmen).

Definition 23 (Erweiterte Analyseverband)**.**

$$L_X^{\Pi} = \prod_{p \in Conds} L_C \times \prod_{y \in CValues} L_X$$

definiert den Verband, in dem

- der Wert jedes Bedingungsknotens durch ein Element von L_C, und
- der Wert jedes beschränkten Integerknotens durch ein Element von L_X

repräsentiert wird. Für ein $l \in L_X^{\Pi}$ bezeichnet $l[p]$ die Komponente eines Bedingungsknotens $p \in Conds$ und $l[x]$ die Komponente eines beschränkten Integerknotens $x \in CValues$.

Das größte Element von L_X^{Π} ist $\top_X^{\Pi} = \underbrace{\top_C \times \cdots \times \top_C}_{|Conds|} \times \underbrace{\top_X \times \cdots \times \top_X}_{|CValues|}$, und analog ist das kleinste Element $\bot_X^{\Pi} = \bot_C \times \cdots \times \bot_C \times \bot_X \times \cdots \times \bot_X$.

Die Notation

$$\top_X^{\Pi}[p \mapsto l][x \mapsto m] = \top_C \times \cdots \times \underbrace{l}_{p} \times \cdots \times \top_C \times \top_X \times \cdots \times \underbrace{m}_{x} \times \cdots \times \top_X$$

beschreibt das größte Element des Produktverbands, in dem die zu $p \in Conds$ gehörende Komponente auf l und die $x \in CValues$ gehörende Komponente auf den Wert m gesetzt wird. Die zweite Ersetzung sei optional; $\top_X^{\Pi}[p \mapsto l]$ entspricht dann $\top_C^{\Pi}[p \mapsto l]$.

Für einen Knoten v sei nun das Attribut $v.\mathsf{pi}$ im Folgenden ein Element von L_X^{Π}.

$\prec$ \ l	h	$\bar{h}$
$<$	$[\mathbb{I}_{\min}, k-1]$	$[k, \mathbb{I}_{\max}]$
$\leq$	$[\mathbb{I}_{\min}, k]$	$[k+1, \mathbb{I}_{\max}]$
$=$	$[k, k]$	$[\mathbb{I}_{\min}, \mathbb{I}_{\max}]$
$\neq$	$[\mathbb{I}_{\min}, \mathbb{I}_{\max}]$	$[k, k]$
$\geq$	$[k, \mathbb{I}_{\max}]$	$[\mathbb{I}_{\min}, k-1]$
$>$	$[k+1, \mathbb{I}_{\max}]$	$[\mathbb{I}_{\min}, k]$

Tabelle 5.1.: Funktion $CI(\langle x \prec \mathsf{k} \rangle, l)$ zur Modellierung eines Beschränkungsknotens als Intervall

Gleichungen und Transferfunktionen Die Änderung am Verband erfordern eine Anpassung der Gleichungen:

$$
\begin{aligned}
PI_\circ(v) &= \bigsqcup \{PI_\bullet(u) \mid u \to v \in E\} \sqcup \iota_v \\
\iota_v &= \begin{cases} \top_X^\Pi & \text{wenn } v\texttt{.op} = \texttt{Return} \\ \bot_X^\Pi & \text{sonst} \end{cases} \\
PI_\bullet(v) &= f_v^{PI}(PI_\circ(v))
\end{aligned}
$$

Die Transferfunktionen werden erweitert zu:

$$
f_v^{PI}(l) = \begin{cases} l \sqcap \top_X^\Pi[p \mapsto m][x \mapsto CI(\langle x \prec \mathsf{k} \rangle, m)] & \text{wenn } v.\mathsf{branch} = (p = \langle x \prec \mathsf{k} \rangle, m) \\ l \sqcap \top_X^\Pi[p \mapsto m] & \text{wenn } v.\mathsf{branch} = (p, m) \wedge p \notin \mathit{Constraints} \\ \mathrm{filter}_X(l) & \text{wenn } v.\mathsf{op} = \theta \\ l & \text{sonst} \end{cases}
$$

mit

$$
\mathrm{filter}_X(l) = \prod_{p \in Conds} \begin{cases} l[p] & \text{wenn } p.\mathsf{depth} < v.\mathsf{depth} \\ \top_C & \text{sonst} \end{cases} \times \prod_{x \in CValues} \begin{cases} l[x] & \text{wenn } x.\mathsf{depth} < v.\mathsf{depth} \\ \top_X & \text{sonst} \end{cases}
$$

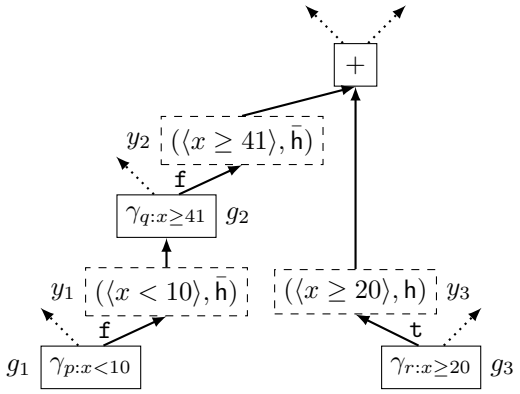
Tabelle 5.1 definiert die Funktion $CI : \mathit{Constraints} \times B \to I$, die das passende Intervall zu dem als erstes Argument gegebenen Beschränkungsknoten liefert. Das zweite Argument modelliert, dass die Beschränkung gilt (h) oder nicht gilt ($\bar{h}$).

Die $\neq$-Relation lässt sich nicht präzise durch ein Intervall darstellen. Das Intervall, das alle möglichen Werte der Relation überspannt, ist $\top_X$.

Die Monotonie von f_v^{PI} ist wie bei der einfachen Analyse gewährleistet, weil wir maximal zwei Komponenten durch Infimumsbildung verändern und alle anderen Komponenten unberührt lassen.

Beispiel Die prinzipielle Vorgehensweise bei der Analyse bleibt unverändert. Abbildung 5.11 zeigt daher die Besonderheiten der erweiterten Analyse an einem kleineren Teilgraph.

Die vierte Komponente des Verbandselements repräsentiert den Wertebereich des Integerknotens x. Wir sehen an den `Path`-Knoten, dass nun zwei Komponenten verändert werden, falls es sich bei dem Bedingungsknoten um einen Beschränkungsknoten handelt. Bei y_2 erhalten wir durch Anwendung der definierten Infimumsoperationen einen relativ präzisen Wertebereich für x. Die Anwendung der Supremumsoperation beim Zusammenführen der eingehenden Informationen bei + invalidiert unser Wissen über die Werte von p, q und r, erhält aber eine Einschränkung für den Wertebereich von x.



$$
\begin{array}{rll}
 & \downarrow p \quad \downarrow q \quad \downarrow r \quad \downarrow x & \\
PI_\bullet(g_1) = \top_X^\Pi & = \top_C \times \top_C \times \top_C \times \top_X & \\
PI_\bullet(g_3) = \top_X^\Pi & = \top_C \times \top_C \times \top_C \times \top_X & \\
PI_\circ(y_1) = PI_\bullet(g_1) & = \top_C \times \top_C \times \top_C \times \top_X & \\
PI_\bullet(y_1) = PI_\circ(y_1) \sqcap \top_X^\Pi[p \mapsto \bar{\mathsf{h}}][x \mapsto [10, \mathbb{I}_{\max}]] & = \bar{\mathsf{h}} \quad \times \top_C \times \top_C \times [10, \mathbb{I}_{\max}] & \\
PI_{\circ/\bullet}(g_2) = PI_\bullet(y_1) & = \bar{\mathsf{h}} \quad \times \top_C \times \top_C \times [10, \mathbb{I}_{\max}] & \\
PI_\circ(y_2) = PI_\bullet(g_2) & = \bar{\mathsf{h}} \quad \times \top_C \times \top_C \times [10, \mathbb{I}_{\max}] & \\
PI_\bullet(y_2) = PI_\circ(y_2) \sqcap \top_X^\Pi[q \mapsto \bar{\mathsf{h}}][x \mapsto [\mathbb{I}_{\min}, 40]] & = \bar{\mathsf{h}} \quad \times \bar{\mathsf{h}} \quad \times \top_C \times [10, 40] & \\
PI_\circ(y_3) = PI_\bullet(g_3) & = \top_C \times \top_C \times \top_C \times \top_X & \\
PI_\bullet(y_3) = PI_\circ(y_3) \sqcap \top_X^\Pi[r \mapsto \mathsf{h}][x \mapsto [20, \mathbb{I}_{\max}]] & = \top_C \times \top_C \times \mathsf{h} \quad \times [20, \mathbb{I}_{\max}] & \\
PI_{\circ/\bullet}(+) = PI_\bullet(y_2) \sqcup PI_\bullet(y_3) & = \top_C \times \top_C \times \top_C \times [10, \mathbb{I}_{\max}] &
\end{array}
$$

Abbildung 5.11.: Beispielgraph und Datenflussgleichungen für die Analyse von implizierten Bedingungen. Es gilt die Annahme, dass bei g_1 und g_2 noch keine Pfadinformation verfügbar ist.

Transformation: Auswertung von implizierten Bedingungsknoten ERG-Implied

Wir definieren eine weitere Funktion $\text{eval}_X(p, l) : \mathit{Conds} \times L_X^\Pi \to L_C$, die einen Bedingungsknoten p im Kontext der Pfadinformation l auswertet. Dabei wird versucht, p zuerst als einfachen Bedingungsknoten auszuwerten. Schlägt das fehl, versuchen wir, ihn als Beschränkungsknoten auszuwerten. Es ist:

$$\text{eval}_X(p, l) = l[p] \sqcap \begin{cases} \text{eval}'_X(p, l) & \text{wenn } p \in \mathit{Constraints} \\ \top_C & \text{sonst} \end{cases}$$

Die Auswertung eines Beschränkungsknotens durch ein Intervall modelliert die Funktion $\text{eval}'_X : \mathit{Constraints} \times L_X^\Pi \to L_C$, definiert als:

$$\text{eval}'_X(q, m) = \begin{cases} h & \text{wenn } q = \langle x \neq \mathsf{k} \rangle \wedge (l[x] \subseteq CI(\langle x < \mathsf{k} \rangle, h) \vee l[x] \subseteq CI(\langle x > \mathsf{k} \rangle, h)) \\ \bar{h} & \text{wenn } q = \langle x = \mathsf{k} \rangle \wedge (l[x] \subseteq CI(\langle x < \mathsf{k} \rangle, h) \vee l[x] \subseteq CI(\langle x > \mathsf{k} \rangle, h)) \\ n & \text{wenn } l[x] \subseteq CI(q, n) \quad (n \in \{h, \bar{h}\}) \\ \top_C & \text{sonst} \end{cases}$$

Die Ungleich-Relation ist durch das größte Verbandselement $\top_X$ kodiert. Die Teilmengenrelation in $l \subseteq \top$ ist daher immer erfüllt, was natürlich falsche Ergebnisse liefert. Wir führen eine Sonderbehandlung für diese Fälle ein, und testen stattdessen auf "kleiner oder größer".

Wie schon bei der einfachen Analyse ersetzen wir bei einem γ-Knoten $g = \gamma(p, a, b)$ das Prädikat durch die Konstante t, wenn $\text{eval}_X(p, g.\mathsf{pi}) = \mathsf{h}$ ist, beziehungsweise f, wenn $\text{eval}_X(p, g.\mathsf{pi}) = \bar{\mathsf{h}}$ ergibt. In allen anderen Fällen ist keine Aussage über den Wert von p möglich.

Durch die Transformation ändern sich die Auswertungswahrscheinlichkeiten der Vorgänger des γ-Knotens, weil bei deren Berechnung die hier gefundene Implikation nicht beachtet wird. Insgesamt können sich die Kosten aufgrund der erhöhten Auswertungswahrscheinlichkeiten der nach der Konstantenfaltung des γ-Knotens übrigbleibenden Vorgänger verschlechtern. Wie auch bei der entsprechenden lokalen Transformation gilt auch hier, dass diese Verschlechterung aus der falschen Annahme der Gleichverteilung der Werte des ersetzten Bedingungsknotens resultiert und der Graph von einer solchen Konstantenfaltung profitiert.

In unserem Beispielgraph aus Abbildung 5.6a gilt für die Auswertung von $q = \langle x = 42 \rangle$ in g_3's Pfadinformation (es wird der zweite Fall der Definition von eval'_X verwendet):

$$\begin{aligned} \text{eval}_X(q, g_3.\mathsf{pi}) &= g_3.\mathsf{pi}[q] \sqcap \text{eval}'_X(q, g_3.\mathsf{pi}) \\ &= \top_C \sqcap \bar{\mathsf{h}} \\ \text{weil } g_3.\mathsf{pi}[x] &= [\mathbb{I}_{\min}, 10] \subseteq [\mathbb{I}_{\min}, 41] = CI(\langle x < 42 \rangle, \mathsf{h}) \end{aligned}$$

Transformation: Pfadspezifische Konstanten ERG-Constified

In bestimmten Fällen können wir die Argumente eines Knotens aufgrund seiner Pfadinformation durch eine Konstante ersetzen. Wenn $v \to x \in E, x \in \mathit{CValues}$ und $v.\mathsf{pi}[x] = [\mathsf{k}, \mathsf{k}]$, ersetzen wir die Kante durch $v \to \mathsf{k}$.

Durch die Transformation kann x nie unerreichbar werden, weil der Knoten ein Argument der Vergleichsoperation ist, auf der die hier verwendete Pfadinformation basiert. Durch das Wegfallen der Kante $v \to x$ werden $P_{\mathsf{eval}}(x)$ und damit $c(x)$ kleiner oder bleibt unverändert. Die Transformation ermöglicht eventuell eine nachfolgende Konstantenfaltung von v.

5.4.3. Knoten-Duplikation

Beim Knoten $*$ im Beispielgraph aus Abbildung 5.6a gehen die Informationen über den Bedingungsknoten p bei der Kombination der Datenflussinformation der Verwender verloren, obwohl diese Information in der Folge bei g_4 genutzt werden könnte.

In diesem Fall ist es sinnvoll, $*$ und g_4 zu duplizieren, um die Pfadinformationen zu erhalten. Um zu verhindern, dass Knoten dupliziert werden, die keine weitere Optimierung ermöglichen, berechnen wir in entgegengesetzter Richtung zur Analyse der Pfadinformationen für jeden Knoten v die Menge der

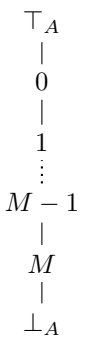


Abbildung 5.12.: Der Verband $L_A = (Q, \geq)$.

erwarteten Bedingungsknoten, die Prädikat und Entfernung der nächsten γ-Knoten angeben. Hier werden die Datenflussinformationen der Vorgänger kombiniert und nach Anwendung der Transferfunktion an die Verwender weitergegeben.

In [TG08] wird mit sogenannten "useful dataflow facts" die Duplikation von Grundblöcken gesteuert, um destruktives Zusammenführen von Datenflussinformationen zu verhindern.

Verband

Lemma 11.

$$L_A = (Q, \geq) \quad \text{mit } Q = \underbrace{[0, M]}_{\subset \mathbb{Z}} \cup \{\top_A, \bot_A\} \text{ und } 0 \geq \top_A \text{ und } \bot_A \geq M$$

ist ein vollständiger Verband.

Beweis. Alle Elemente von Q stehen direkt oder transitiv in Relation, deswegen ist für alle $r, s \in Q, r \geq s$ das Supremum definiert durch $r \sqcup s = s$ und das Infimum durch $r \sqcap s = r$. L_D erfüllt die Verbandseigenschaft, und da die Grundmenge Q endlich ist, handelt es sich um einen vollständigen Verband. □

Abbildung 5.12 zeigt das Konstrukt. Man beachte, dass die Relation "größer gleich" lautet - daher ist das größte Verbandselement $\top_A$ bezüglich der natürlichen Ordnung der Zahlen kleiner als alle anderen Elemente, und umgekehrt ist das kleinste Verbandselement $\bot_A$ größer als alle anderen Elemente.

Die Konstante M gibt an, über welche Anzahl von Knoten wir die Information über die Benutzung eines Prädikats propagieren wollen; sie hat maximal den Wert $|V|$.

Ein Verbandselement von L_A ist als Distanz zum nächsten γ-Knoten mit einem bestimmten Bedingungsknoten zu interpretieren. $\top_A$ bedeutet, dass der repräsentierte Bedingungsknoten auf keinem Pfad zum `Start`-Knoten noch einmal von einem γ-Knoten verwendet wird. Das Element $\bot_A$ wiederum sagt aus, dass der repräsentierte Knoten zwar verwendet wird, die Distanz aber größer als der maximale Abstand ist, bei dem wir noch duplizieren wollen.

Definition 24 (Analyseverband für erwartete Bedingungsknoten)**.**

$$L_A^{\Pi} = \prod_{p \in Conds} L_A$$

definiert den Produktverband, in dem für jeden Bedingungsknoten $p \in Conds$ der kürzeste Abstand zum nächsten γ-Knoten, dessen Prädikat p ist, repräsentiert.

Für ein $l \in L_A^{\Pi}$ bezeichnet die Notation $l[p]$ die zu einem Bedingungsknoten p gehörige Komponente des Produktverbandelements.

Das größte Element von L_A ist analog zu den anderen Analyseverbänden $\top_A^\Pi = \top_A \times \cdots \times \cdots \times \top_A$, und das kleinste Element ist $\bot_A^\Pi = \bot_A \times \cdots \times \ldots \bot_A$. Ebenso übernehmen wir die gewohnte Schreibweise

$$\top_A^\Pi[p \mapsto l] = \top_A \times \cdots \times \underbrace{l}_{p} \times \cdots \times \top_A$$

um das Verbandselement zu konstruieren, das in der p zugehörigen Komponente den Wert l hat und bei dem in allen anderen Komponenten das neutrale Element bezüglich der Infimumsoperation gesetzt ist.

Jeder Knoten v erhält ein Attribut $v.\mathsf{ac}$, dessen Typ L_A^Π ist.[9]

Gleichungen und Transferfunktionen Die Gleichungen der Analyse für einen Knoten v lauten:

$$\begin{aligned} AC_\circ(v) &= \bigsqcup \{AC_\bullet(w) \mid v \to w \in E\} \sqcup \iota_v \\ \iota_v &= \begin{cases} \top_A^\Pi & \text{wenn } \nexists v \to w \in E \\ \bot_A^\Pi & \text{sonst} \end{cases} \\ AC_\bullet(v) &= f_v^{AC}(AC_\circ(v)) \end{aligned}$$

Für einen Knoten v ist die Transferfunktion gegeben durch:

$$\mathrm{inc}(l) = l'_1 \times \cdots \times l'_n \quad \text{mit } l'_i = \begin{cases} \top_A & \text{wenn } l_i = \top_A \\ \bot_A & \text{wenn } l_i \geq M \\ l_i + 1 & \text{sonst} \end{cases}$$

$$\mathit{MultiValNodes} = \{v \in V \mid v.\mathsf{op} = \texttt{Load} \vee (v.\mathsf{op} = \texttt{Call} \wedge v.\mathsf{returnType} \neq \texttt{void})\}$$

$$f_v^{AC}(l) = \begin{cases} \mathrm{inc}(l) \sqcap \top_A^\Pi[v.\mathsf{cond} \mapsto 0] & \text{wenn } v.\mathsf{op} = \gamma \\ \top_A^\Pi & \text{wenn } v.\mathsf{op} = \theta \vee v \in \mathit{MultiValNodes} \\ l & \text{wenn } v.\mathsf{op} = \texttt{Proj} \\ \mathrm{inc}(l) & \text{sonst} \end{cases}$$

Lemma 12. Für alle $v \in V$ ist f_v^{AC} monoton.

Beweis. Betrachten wir die Funktion inc und zwei Elemente $r, s \in L_A^\Pi$ mit $r \sqsubseteq s$. Bei Anwendung von $\mathrm{inc}(r)$ bleibt eine Komponente gleich, falls sie den Wert $\top_A$ hat. Da $\top_A$ das größte Element ist, ändert sich die Relation in dieser Komponente nicht, unabhängig davon, wie $\mathrm{inc}(s)$ sie verändert. Analog bleibt die Relation $\mathrm{inc}(r) \sqsubseteq \mathrm{inc}(s)$ bestehen, wenn bei der Anwendung von $\mathrm{inc}(s)$ eine Komponente unverändert $\bot_A$ bleibt. Für alle anderen Fälle ist die Inkrementierung einer Komponente um eins monoton. inc ist also monoton.

Im Falle von γ-Knoten wird nach der (monotonen) Inkrementierung des Verbandselement l nur die Komponente verändert, die dem zugehörigen Bedingungsknoten entspricht. Für diese Komponente wird die Distanz fest auf 0 gesetzt, so dass $r, s \in L_A, r \sqsubseteq s$ nach Anwendung der Transferfunktion in eben dieser Komponente gleich sind, und alle anderen Komponenten entsprechend der Annahme noch in Relation stehen.

Für θ-Knoten und Tupelknoten mit mehr als einer Projektion werden alle erwarteten Bedingungsknoten gelöscht, weil wir diese Knotentypen nicht lokal duplizieren können. Für diese Knoten ist die Transferfunktion trivialerweise monoton.

Projektionen werden bei der Berechnung der Distanz ignoriert; ihre Transferfunktion ist die Identitätsfunktion, die ebenfalls monoton ist.

Für alle anderen Knotentypen wird die inc-Funktion auf das eingehende Verbandselement angewendet; deren Monotonie zeigten wir bereits. □

[9] ac ist die Abkürzung des englischen Begriffs “anticipated conditions”.

$$
\begin{array}{rll}
 & \downarrow p & \downarrow q \quad \downarrow r \\
AC_\bullet(a,b,c,d,e,f) = \top_A^\Pi & = \top_A \times \top_A \times \top_A \\
AC_\circ(g_2) = AC_\bullet(a) \sqcup AC_\bullet(b) & = \top_A \times \top_A \times \top_A \\
AC_\bullet(g_2) = \mathrm{inc}(AC_\circ(g_2)) \sqcap \top_A^\Pi[p \mapsto 0] & = 0 \quad \times \top_A \times \top_A \\
AC_\circ(g_4) = AC_\bullet(c) \sqcup AC_\bullet(d) & = \top_A \times \top_A \times \top_A \\
AC_\bullet(g_4) = \mathrm{inc}(AC_\circ(g_4)) \sqcap \top_A^\Pi[r \mapsto 0] & = \top_A \times \top_A \times 0 \\
AC_\circ(g_3) = AC_\bullet(e) \sqcup AC_\bullet(f) & = \top_A \times \top_A \times \top_A \\
AC_\bullet(g_3) = \mathrm{inc}(AC_\circ(g_3)) \sqcap \top_A^\Pi[r \mapsto 0] & = \top_A \times 0 \quad \times \top_A \\
AC_\circ(*) = AC_\bullet(g_4) \sqcup AC_\bullet(g_4) & = \top_A \times \top_A \times 0 \\
AC_\bullet(*) = \mathrm{inc}(AC_\circ(*)) & = \top_A \times \top_A \times 1 \\
AC_\circ(+) = AC_\bullet(g_2) \sqcup AC_\bullet(*) & = 0 \quad \times \top_A \times 1 \\
AC_\bullet(+) = \mathrm{inc}(AC_\circ(+)) & = 1 \quad \times \top_A \times 2 \\
AC_\circ(-) = AC_\bullet(*) \sqcup AC_\bullet(g_3) & = \top_A \times 0 \quad \times 1 \\
AC_\bullet(-) = \mathrm{inc}(AC_\circ(-)) & = \top_A \times 1 \quad \times 2 \\
AC_\circ(g_1) = AC_\bullet(+) \sqcup AC_\bullet(-) & = 1 \quad \times 1 \quad \times 2 \\
AC_\bullet(g_1) = \mathrm{inc}(AC_\circ(g_1) \sqcap \top_A^\Pi[p \mapsto 0]) & = 0 \quad \times 2 \quad \times 3
\end{array}
$$

Abbildung 5.13.: Datenflussgleichungen zur Analyse der erwarteten Bedingungen für den Graphen aus Abbildung 5.6a. Wir nehmen an, dass a-f keine erwarteten Bedingungen produzieren.

Beispiel Anhand unseres Beispielgraphen aus Abbildung 5.6a stellen wir in Abbildung 5.13 das Gleichungssystem der Analyse auf und können wie gewohnt aufgrund des azyklischen Graphen gleich die Lösung angeben. Der berechnete Attributwert $g_1.\mathsf{ac} = 0 \times 2 \times 3$ ist folgendermaßen zu interpretieren: Die nächste Verwendung des Bedingungsknotens p als Prädikat eines γ-Knotens erfolgt mit der Distanz 0, das heißt, bei g_1. q wird mit der Distanz 2 bei g_3, der über den Knoten $-$ erreichbar ist, verwendet. Die nächste Verwendung von r ist g_4 mit Distanz 3, erreichbar über $+ \to *$ beziehungsweise $- \to *$. Im Beispiel sind beide Möglichkeiten, g_4 zu erreichen, gleich weit entfernt. Allgemein würde die Analyse die kürzeste Distanz ermitteln.

Transformation: Duplikation ERG-Dup

Die grundlegende Idee für die Duplikation eines Knotens v ist: Lässt sich ein Bedingungsknoten aus der Menge der erwarteten Bedingungen unter den Pfadinformationen von v nicht auswerten, in den Pfadinformationen eines Verwenders u von v jedoch schon, so dupliziere v zu v' und verbinde u mit v'. Da u dann der einzige Verwender von v' ist, übernimmt v' die Pfadinformationen von u.

Für arithmetisch-logische Knoten, die nur einen Wert produzieren, können wir die Transformation entsprechend durchführen. Formalisiert duplizieren wir also einen Knoten v, wenn $\exists p \in \mathit{Conds}$ mit

$$v.\mathsf{ac}[p] \in [0, M] \quad \wedge \quad \mathrm{eval}_X(p, v.\mathsf{pi}) = \top_C \quad \wedge \quad \exists u \to v \in E : \mathrm{eval}_X(p, u.\mathsf{pi}) \neq \top_C$$

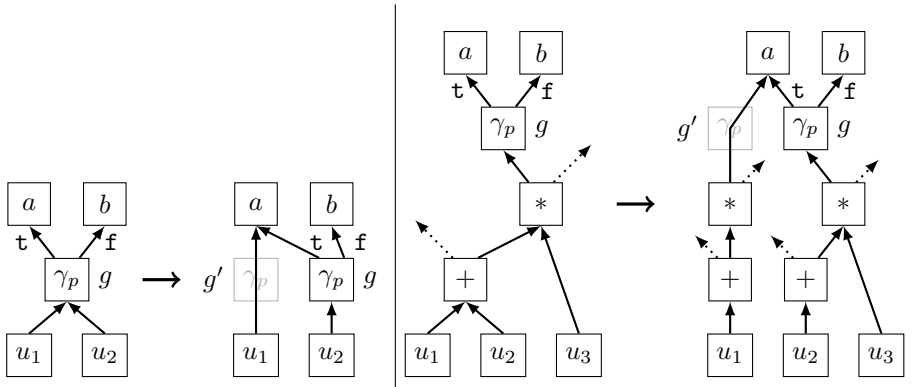
gilt.

Knoten, die einelementige Tupel produzieren (`Store` und `Call` bei Funktionen ohne Rückgabewert) lassen sich analog behandeln. Wir duplizieren v, wenn $\exists p \in \mathit{Conds}$ mit

$$v.\mathsf{ac}[p] \in [0, M] \quad \wedge \quad \mathrm{eval}_X(p, v.\mathsf{pi}) = \top_C \quad \wedge \quad \exists u \to j \to v \in E : \mathrm{eval}_X(p, u.\mathsf{pi}) \neq \top_C$$

gilt. Die Projektion j muss entsprechend mitkopiert werden.

In Abbildung 5.14a sehen wir, wie der Verwender u_1 direkt mit dem entsprechenden Vorgänger von g verbunden wird, weil seine Pfadinformation eine Aussage über p enthält.



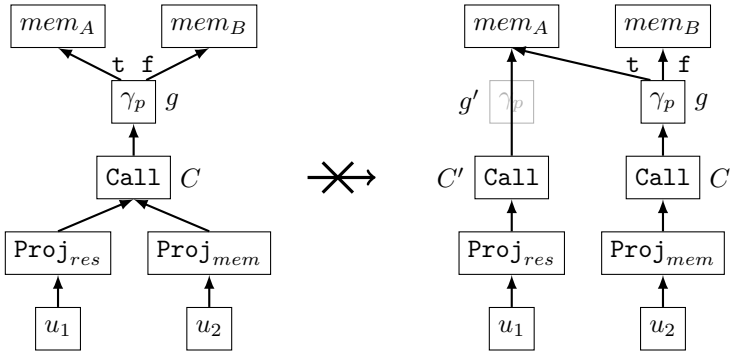
$\text{eval}_X(p, u_1.\mathsf{pi}) = \mathsf{h}$
$\text{eval}_X(p, u_2.\mathsf{pi}) = \top_C$
$g.\mathsf{ac}[p] = 0$

(a) Distanz = 0

$\text{eval}_X(p, u_1.\mathsf{pi}) = \mathsf{h}$
$\text{eval}_X(p, u_2.\mathsf{pi}) = \text{eval}_X(p, u_3.\mathsf{pi}) = \top_C$
$g.\mathsf{ac}[p] = 0 \quad *.\mathsf{ac}[p] = 1 \quad +.\mathsf{ac}[p] = 2$

(b) Distanz > 0

Abbildung 5.14.: Knotenduplikation aufgrund erwarteter Bedingungsknoten mit ...



$\text{eval}_X(p, \texttt{Proj}_{res}.\mathsf{pi}) = \mathsf{h}$
$\text{eval}_X(p, \texttt{Proj}_{mem}.\mathsf{pi}) = \top_C$

Abbildung 5.15.: Inkorrekte Duplikation von tupelwertigen Knoten.

Abbildung 5.14b zeigt an einem Beispiel, wie eine verwertbare Pfadinformation durch sukzessive Duplikation der Addition, der Multiplikation und zuletzt des γ-Knotens bis dorthin verfügbar gemacht wird.

Hat ein Knoten mehr als eine Projektion, ist die Situation komplizierter. Eine später verwertbare Pfadinformation, die nicht über alle Projektionen verfügbar ist, würde die Duplikation des zugehörigen Knotens auslösen. Dabei ginge die Zuordnung der Projektionen verloren. Betrachten wir einen Funktionsaufruf wie C in Abbildung 5.15, über dessen Ergebnisprojektion $\texttt{Proj}_{res}$ eine verwertbare Pfadinformation hereinkommt, über die Speicherprojektion $\texttt{Proj}_{mem}$ jedoch nicht.

Würden wir naiv duplizieren, erzeugen wir eine Kopie C' des Funktionsaufrufs, von der wir ausschließlich den Rückgabewert verwenden. C hat anschließend nur eine Projektion des Speicherwerts. Da wir die Duplikation g' von g optimieren werden, können C und C' nicht mehr verschmolzen werden. Die naive Transformation erzeugt einen Graphen mit zwei Funktionsaufrufen, für den wir keinen Code erzeugen können.

Aufgrund dessen lässt sich die Duplikation bei Knoten mit mehr als einer Projektion nicht lokal durchführen.

Eine Knotenduplikation verschlechtert prinzipiell das Programm in unserem Kostenmodell. Dies ist der Fall, weil sie die Umkehroperation der Knotenverschmelzung ist, und wir gesehen haben, dass die Verschmelzung kostenneutral oder -verbessernd ist. In Situationen, in denen die Duplikation kostenneutral ist, erhöht sie dennoch die statische Codegröße. Eine Ausnahme bildet die Duplikation bei Distanz 0 wie in Abbildung 5.14a, wenn man die anschließende sichere Elimination von g' mit einbezieht.

Die Duplikation und anschließende γ-Elimination bewirkt eine Änderung der Auswertungswahrscheinlichkeiten und kann zur Entstehung weiterer Optimierungsmöglichkeiten beitragen. Die Auswertungswahrscheinlichkeiten der Vorgänger der betroffenen Knoten können sich positiv sowie negativ verändern. Die ursprünglichen Knoten haben nach der Duplikation weniger Verwender und möglicherweise gelten präzisere Pfadinformationen.

Trotzdem lassen sich diese Auswirkungen zum Zeitpunkt der Duplikation nicht vorhersagen. Wir überprüfen im Rahmen der Evaluation, für welche Distanzen die Duplikation gewinnbringend für den gesamten Graph ist.

Im Beispiel am Anfang dieses Abschnitts (Abbildung 5.6b) betragen die Kosten nach der Elimination redundanter γ-Knoten, aber *ohne* Anwendung der Knotenduplikation, 10 statt 8.5.

5.4.4. Vollständiger Ablauf

```
1:  function EliminateRedundantGammas(graph)
2:      mergeNodes ← t
3:      repeat
4:          evaluated ← f, duplicated ← f
5:          PerformLocalOpts
6:          AnalyzePathInfo
7:          evaluated ← EvaluateConds
8:
9:          PerformLocalOpts
10:         AnalyzeAnticipatedConds
11:         duplicated ← DuplicateNodes
12:         mergeNodes ← ¬ duplicated
13:     until ¬ evaluated ∧ ¬ duplicated
14: end function
```

Algorithmus 1: Elimination von redundantem γ-Knoten

Die Analysen und Transformationen kombinieren wir zu Algorithmus 1. Wir iterieren so lange, bis in einer Iteration kein Bedingungsknoten mehr ausgewertet und auch keine Knoten mehr dupliziert werden.

Die Funktionen AnalyzePathInfo und AnalyzeAnticipatedConds berechnen den Fixpunkt der Analyse der Pfadinformationen beziehungsweise der erwarteten Bedingungsknoten. EvaluateConds führt die beiden beschriebenen, auf den Pfadinformationen basierenden Transformationen aus. DuplicatedNodes dupliziert Knoten, wenn es auf Grundlage der erwarteten Bedingungsknoten sinnvoll erscheint. PerformLocalOpts entspricht der Durchführung der im vorherigen Abschnitt beschrieben lokalen Optimierungen.

Ob die Knotenverschmelzung durchgeführt wird, hängt vom Wert der Variable “mergeNodes” ab. Die lokalen Optimierungen sind in Zeile 4 insbesondere wichtig, um möglichst viele Bedingungsknoten zu verschmelzen. Fand in der vorhergehenden Iteration eine Duplikation statt, muss die Knotenverschmelzung allerdings abgeschaltet sein, um den Effekt nicht gleich wieder rückgängig zu machen. Nach der Analyse und Auswertung der Bedingungsknoten ist es erforderlich, die γ-Knoten, die ein konstantes Prädikat erhalten haben, durch die erneute Anwendung der lokalen Optimierungen zu eliminieren.

Beispiel Die Anwendung des Verfahrens auf den Beispielgraphen aus Abbildung 5.6a erfolgt in folgenden Schritten:

1. Falls nicht bereits vorher geschehen, werden die Bedingungsknoten von g_1 und g_2 durch PerformLocalOpts verschmolzen.
2. AnalyzePathInfo berechnet die Pfadinformationen wie in Abbildung 5.8 gezeigt. Zusätzlich werden dabei die implizierten Bedingungen berechnet.
3. EvaluateConds wertet die Bedingungsknoten von g_2 und g_3 aus und verbindet deren `cond`-Eingang mit der entsprechenden Konstante.
4. PerformLocalOpts ersetzt g_2 und g_3 durch Anwendung der lokalen Ersetzungsregel.
5. AnalyzeAnticipatedConds berechnet die Mengen der erwarteten Bedingungsknoten (Abbildung 5.13).
6. DuplicateNodes dupliziert aufgrund dieser Information die Knoten $*$ und g_4.
7. PerformLocalOpts läuft nun mit deaktivierter Knotenverschmelzung.
8. AnalyzePathInfos berechnet die neuen Pfadinformation. Dieses Mal gibt es keinen Informationsverlust bei der Multiplikation.
9. EvaluateConds kann die Bedingung der Kopie von g_4 auswerten.
10. PerformLocalOpts eliminiert die Kopie von g_4.
11. Es ergeben sich keine Möglichkeiten zur Duplikation.
12. In der nächsten Iteration des Algorithmus finden wir keine verwertbaren Pfadinformationen mehr und auch keine Duplikationskandidaten. Der Algorithmus terminiert, der Graph entspricht nun Abbildung 5.6b.

5.5. Schleifenoptimierungen

Nachdem sich die vorherigen Abschnitte vornehmlich damit befasst haben, Knoten entweder zu eliminieren oder die Fälle zu reduzieren, in denen sie ausgewertet werden, wollen wir jetzt die Häufigkeit ihrer Auswertung reduzieren. In unserem Kostenmodell entspricht dies einer Reduktion der Schleifentiefe.

5.5.1. Optimierung von additiven θ-Knoten

Definition 25 (Additive θ-Knoten)**.**

$$\begin{aligned} \mathit{AdditiveThetas} = \{t \in V \mid t = \theta(w_0, a = +(t, w_+))\} \\ \text{mit } w_0.\mathsf{depth}, w_+.\mathsf{depth} < t.\mathsf{depth} \\ \text{und } t.\mathsf{mode} = \mathbb{I} \text{ und } \alpha(a) = 1 \end{aligned}$$

definiert die Menge der *additiven* θ-Knoten.

Bei diesen Knoten handelt es sich um Integer-θ-Knoten t mit einem schleifeninvarianten *Initialwert* w_0 und einer Addition a am `next`-Eingang, deren Argumente t sowie ein schleifeninvarianter *Inkrement* w_+ sind und a keine Verwender außer t hat. Wir notieren kürzer $t = \theta_{w_0}^{w_+}$.

Abbildung 5.16 zeigt dieses Muster. Ein additiver θ-Knoten $t = \theta_{w_0}^{w_+}$ produziert Werte t_i, deren Abstand konstant ist. Der Wert des Knotens in der i-ten Iteration ist $t_i = w_0 + i \cdot w_+$.

Additive θ-Knoten eignen sich für zwei spezielle Transformationen. Additionen und Multiplikationen, deren Argumente ein solcher θ-Knoten und ein anderer, schleifeninvarianter Wert sind, können wir durch den θ-Knoten propagieren.

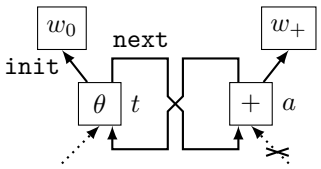


Abbildung 5.16.: Muster eines additiven θ-Knotens. Es gilt $w_0.\mathsf{depth}, w_+.\mathsf{depth} < t.\mathsf{depth}, a.\mathsf{depth}$ sowie $\alpha(a) = 1$.

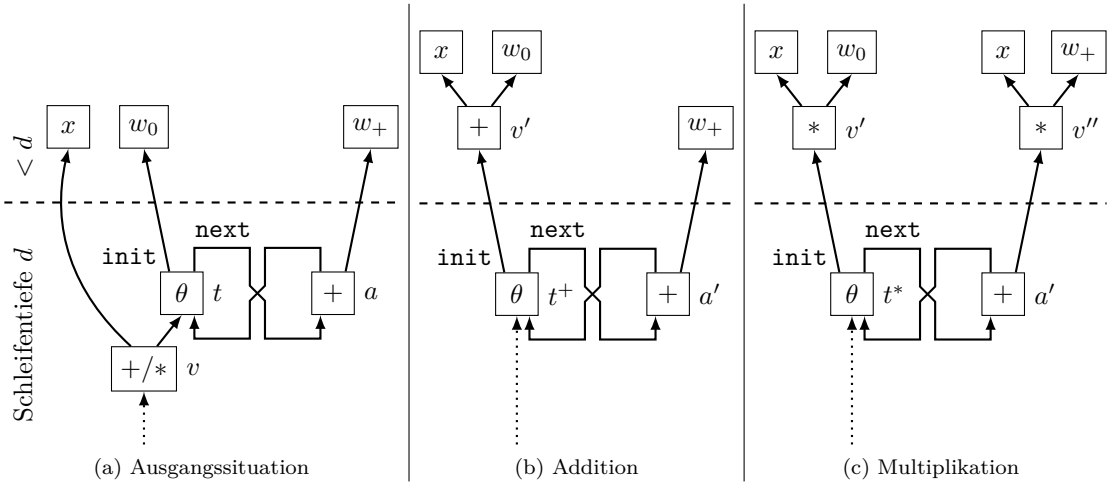


Abbildung 5.17.: Transformation additiver θ-Knoten

Die Optimierung additiver θ-Knoten entspricht dem Effekt der *(Loop) Strength Reduction* [Muc97, Mor98, CSV01]. Ähnliche Transformationen werden auch in [Upt06] und [TSTL09] beschrieben.

Ein Beispiel für eine solche Situation sind die Adressrechnungen bei Arrayzugriffen in Schleifen. Betrachten wir den Zugriff $A[i]$ für eine Indexvariable i. In der Zwischendarstellung wird die zugehörige Adressrechnung als $adr = base(A) + i \cdot s$ dargestellt, wobei $base(A)$ die Startadresse und s die Größe des Elementtyps von A angibt. Man kann die Multiplikation in jeder Iteration sparen, indem man stattdessen $adr = base(A)$ initialisiert und anschließend in jeder Iteration $adr = adr + s$ addiert.

Die Beschränkung auf additive θ-Knoten ist dabei aus Gründen der Einfachheit gewählt. Die vorgestellten Transformationen sind auch anwendbar, wenn sich auf dem Zyklus eines θ-Knotens mehrere Additionen befinden. Die im nächsten Abschnitt beschriebene Umordnungstransformation normalisiert den Graph so, dass sich möglichst wenige Additionen auf dem Zyklus befinden, so dass das Muster des additiven θ-Knotens häufig passt.

Das Muster in Abbildung 5.17a zeigt eine Operation v mit den Argumenten t und x, wobei $t = \theta_{w_0}^{w_+}$ und x schleifeninvariant bezüglich t sei. v ist entweder eine Integeraddition oder -multiplikation.

Additionen $v = t + x$ ist durch die Abhängigkeit zu t schleifenabhängig und produziert Werte der Form $v_i = (w_0 + i \cdot w_+) + x$.

Wir erzeugen einen neuen additiven θ-Knoten $t^+ = \theta_{w_0+x}^{w_+}$ durch Ausnutzung der Assoziativität und Kommutativität der Integeraddition; Abbildung 5.17b zeigt das Resultat.

t^+ produziert Werte der Form $t_i^+ = (w_0 + x) + i \cdot w_+ = v_i + x$. Da sowohl w_0 und x schleifeninvariant zu t sind, ist auch die neu entstandene Addition $v' = w_0 + x$ im Gegensatz zu v schleifeninvariant zu t. Eine Intuition der Transformation ist, dass wir die gesamte Wertefolge um den Wert von x *verschieben*.

Multiplikationen $v = t \cdot x$ produziert Werte der Form $v_i = (w_0 + i \cdot w_+) \cdot x$. Wir wenden das Distributivgesetz an, um einen neuen additiven θ-Knoten $t^* = \theta^{w_+ \cdot x}_{w_0 \cdot x}$ wie in Abbildung 5.17c zu erzeugen; t^* produziert Werte der Form $t^*_i = (w_0 \cdot x) + i \cdot (w_+ \cdot x) = v_i \cdot x$. Wie im Falle der Addition sind die neuen Multiplikationen $v' = w_0 \cdot x$ und $v'' = w_+ \cdot x$ schleifeninvariant zu t. Durch diese Transformation *skalieren* wir die Wertefolge mit dem Faktor x.

Neuverbinden der Verwender StrengthRed

Die Verwender von v verbinden wir anschließend mit dem entsprechenden neu erstellten Knoten t^+ beziehungsweise t^*. Wenn t keine weiteren Verwender außer a hat, ist der Knoten tot und kann eliminiert werden. a selbst hat keinen Einfluss auf die Lebendigkeit von t.

Kostenbilanz Die Transformation verringert die Kosten des Graphs, wenn $\alpha(t) = 2$, der ursprüngliche additive θ-Knoten t also anschließend eliminiert werden kann.

Die Auswertungswahrscheinlichkeiten und statischen Kosten der Knoten im Teilgraph $\{t, a, v\}$ in Abbildung 5.17a sind gleich. Dies gilt auch für ihre Kopien t^+, t^*, a', v' und v''. Die Kosten für x, w_0 und w_+ ändern sich nicht.

Sei $d = t.\mathsf{depth} = a.\mathsf{depth} = v.\mathsf{depth}$, $d' = \max\{x.\mathsf{depth}, w_0.\mathsf{depth}\}$, $d'' = \max\{x.\mathsf{depth}, w_+.\mathsf{depth}\}$ und $K = P_{\mathsf{eval}}(v) \cdot \omega(v)$ der unveränderliche Anteil in den Kosten aller Knoten.

Es gilt $d', d'' < d$ aufgrund der gewählten Vorbedingungen für die Transformation.

Vor der Transformation betragen die Kosten:

$$c(t) + c(a) + c(v) = 3 \cdot K \cdot \mathbb{L}^d$$

Ist v eine Addition, ist die Kostendifferenz nach der Transformation wie in Abbildung 5.17b:

$$\begin{aligned} c(t^+) + c(a') + c(v') - c(t) - c(a) - c(v) &= 2 \cdot K \cdot \mathbb{L}^d + K \cdot \mathbb{L}^{d'} - 3 \cdot K \cdot \mathbb{L}^d \\ &= K \cdot (\mathbb{L}^{d'} - \mathbb{L}^d) \quad < 0 \end{aligned}$$

Ist v eine Multiplikation, ist die Kostendifferenz nach der Transformation wie in Abbildung 5.17c:

$$\begin{aligned} c(t^*) + c(a') + c(v') + c(v'') - c(t) - c(a) - c(v) &= 2 \cdot K \cdot \mathbb{L}^d + K \cdot \mathbb{L}^{d'} + K \cdot \mathbb{L}^{d''} - 3 \cdot K \cdot \mathbb{L}^d \\ &= K \cdot (\mathbb{L}^{d'} + \mathbb{L}^{d''} - \mathbb{L}^d) \\ &\leq K \cdot (2 \cdot \mathbb{L}^{d-1} - \mathbb{L}^d) \quad < 0 \end{aligned}$$

Beide Transformationen verringern also die Kosten im Graph[10].

Kann der ursprüngliche additive θ-Knoten nicht eliminiert werden, erhöhen sich Kosten des Programms um die Kosten für die neu erstellten Knoten. Dann führen wir die Transformation nicht durch.

Anpassen von Vergleichsoperationen LFTR

Hat t weitere Verwender neben v, ist eine häufig anzutreffende Situation, dass es sich dabei um Vergleichsoperationen handelt. Dann versuchen wir, den Vergleich so anzupassen, dass er auf den von t^+ beziehungsweise t^* produzierten Werten operiert, um t anschließend eliminieren zu können.

Die Anpassung der Vergleichsoperation ist in der Literatur als *Linear Function Test Replacement* [Muc97, Mor98, CSV01] bekannt und wird auch in [Upt06] auf eine vFirm-ähnliche Zwischendarstellung angewandt.

[10] Zur Erinnerung: $\mathbb{L} = 10$

Ist $c = \texttt{Cmp}(t, \prec, N)$ mit einem schleifeninvarianten Argument N, erzeugen wir eine weitere Kopie von v mit Argument N. Dabei müssen wir beachten, dass die Äquivalenz

$$\begin{aligned} t \prec N \quad &\Leftrightarrow \quad t + x \prec N + x \\ \text{bzw.} \quad &\Leftrightarrow \quad t \cdot x \prec N \cdot x \end{aligned}$$

nur gilt, wenn für alle möglichen Werte $Y = \{y \in \mathbb{I} : (\exists i : t_i = y) \vee N = y\}$, die t und N annehmen können, die Funktion $f : Y \to \mathbb{I}$ mit $f(y) = y + x$ bzw. $f(y) = y \cdot x$ streng monoton wachsend ist. Mathematisch ist dies für die Addition immer und für die Multiplikation unter der Bedingung erfüllt, dass x positiv ist. Die Multiplikation mit einem negativen x ist streng monoton fallend; hier gilt die Äquivalenz mit umgedrehter Relation $\succ$.

Im Rahmen der Integerarithmetik wird die Monotonie verletzt, wenn ein möglicher Wert $z \in Y$ existiert, so dass $z + x$ oder $z \cdot x$ einen Über- oder Unterlauf[11] verursachen.

Um garantieren zu können, dass keine Überläufe auftreten, benötigt man möglichst präzise Informationen über die Wertebereiche der Variablen. Upton [Upt06] beschreibt eine geeignete Analyse. Zudem muss man statisch eine obere Schranke für den Iterationszähler der Schleifengruppe ermitteln.

Für diese Arbeit treffen wir die Annahme, dass die in Schleifen berechneten Werte nicht überlaufen[12]. Dann müssen wir die Werte von v's Argument x und den Vergleichsoperanden N kennen. Für konstante Werte können wir leicht prüfen, ob wir $N + x$ beziehungsweise $N \cdot x$ ohne Überlauf berechnen können. Gelingt das, führen wir die Anpassung der Vergleichsoperation durch. Ist $x < 0$, müssen wir die Relation noch umdrehen. Da in dieser Situation die Kopie von v sofort durch die Konstantenfaltung eliminiert wird, verursacht die Anpassung keine neuen Kosten.

5.5.2. Berechnung der Schleifengruppen

Wir haben die Schleifengruppen in Definition 17 als Partitionierung der Knotenmenge eingeführt.

Nach dieser Definition befinden sich zwei Knoten a, b in der gleichen Schleifengruppe, wenn sie die gleiche Schleifentiefe besitzen und ein Pfad $a \to^* b$ oder umgekehrt existiert.

Betrachten wir zwei Knoten v und w mit $v.\mathsf{op}, w.\mathsf{op} \neq \eta$, die durch eine Kante $v \to w$ verbunden sind. Wir nennen ihre Schleifengruppen $L_v = v.\mathsf{loop}$ und $L_w = w.\mathsf{loop}$. Zur Verdeutlichung der im folgenden beschriebenen Situationen beziehen wir uns auf den Beispielgraph in Abbildung 5.18.

- Haben beide Knoten die gleiche Schleifentiefe $v.\mathsf{depth} = w.\mathsf{depth}$, gehören sie der gleichen Schleifengruppe an, es gilt also $L_v = L_w$.

  Beispiele hierfür sind die Kanten $t_1 \to a_1$ in L_2 und $t_7 \to *$ in L_3.

- Ist die Schleifentiefe von w kleiner als die von v, können die beiden Knoten nicht zur gleichen Gruppe gehören. Sei beispielsweise $v.\mathsf{depth} = w.\mathsf{depth} + 1$. L_v wird von seiner Elterngruppe $L_v.\mathsf{parent}$ umgeben. Die Knoten der Elterngruppe haben die gleiche Schleifentiefe wie w, und können w über die Knoten in L_v und schließlich über v erreichen. In diesem Fall gilt also $L_v.\mathsf{parent} = L_w$.

  Diese Situation tritt beispielsweise bei der Kante $t_5 \to -$ ein: Es existiert der Pfad $a_1 \to e_2 \to * \to t_5 \to -$, aufgrund dessen der Knoten $-$ in L_2 liegt.

- Ist die Differenz in der Schleifentiefe größer als eins, muss man entsprechend öfter die umgebende Schleifengruppe ermitteln.

  Die Kante $a_4 \to c$ liefert uns die Äquivalenz $L_2.\mathsf{parent}.\mathsf{parent} = L_0$.

- Ist v ein η-Knoten, zählen wir ihn zu der Schleifengruppe seiner Vorgänger, und müssen deshalb mit einer um eins inkrementierten Schleifentiefe rechnen.

[11] Im Folgenden unterscheiden wir o.b.d.A nicht weiter zwischen Über- und Unterlauf.

[12] Der Benutzer des Compilers sollte diese Annahme explizit durch das Setzen eines entsprechenden Parameters bestätigen.

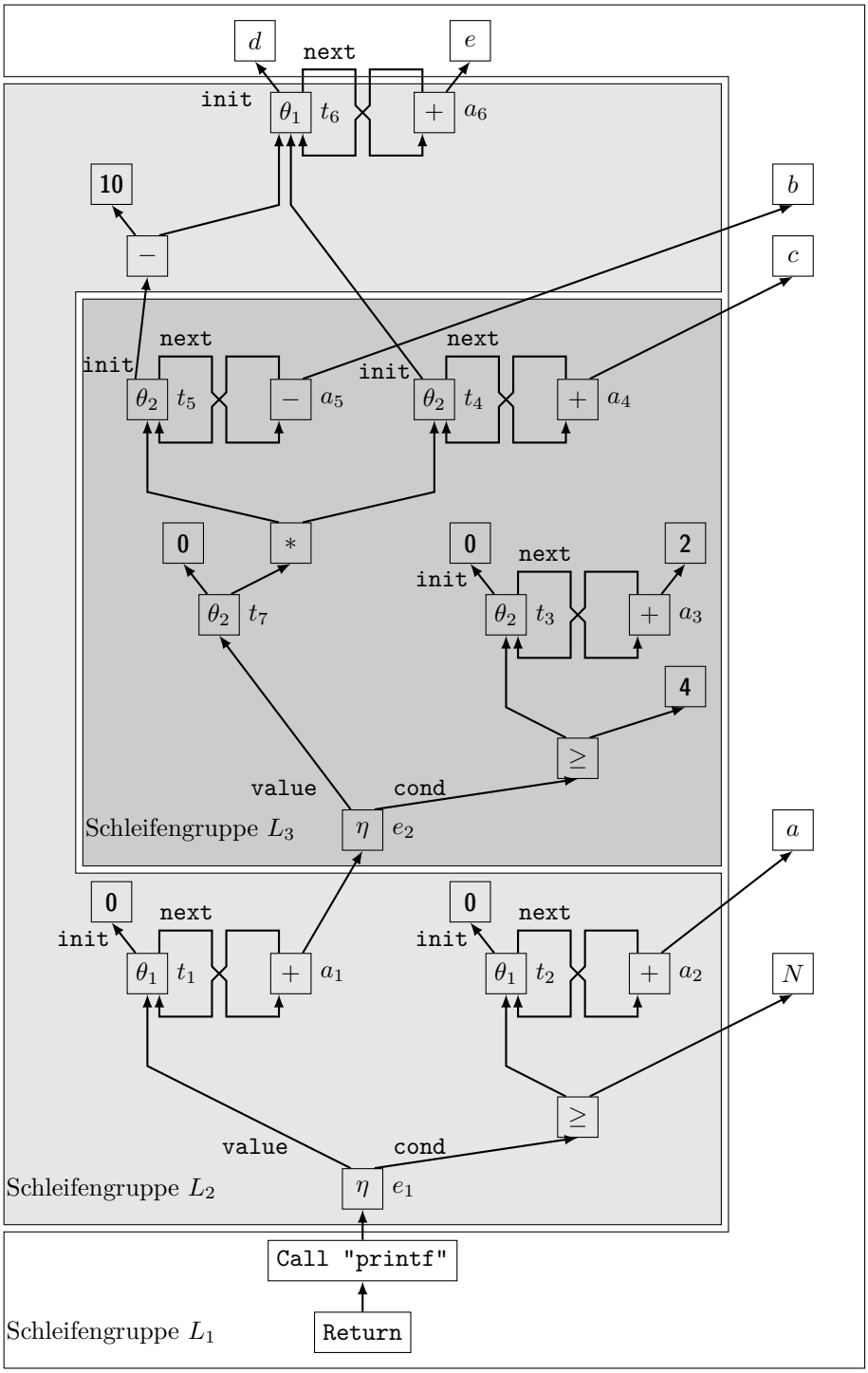


Abbildung 5.18.: Beispielgraph mit Einteilung in Schleifengruppen

```
1: function MakeLoopClusters
2:     for all v ∈ V do
3:         MakeLoop(v)
4:     end for
5:
6:     for all v ∈ V do
7:         for all v → w ∈ E do
8:             δ ← v.loop.depth − w.depth      // v.loop.depth = v.depth + 1 wenn v.op = η
9:             P ← FindAncestor(v.loop, δ)
10:
11:            if w.op ≠ η then
12:                UnionLoops(P, w.loop)
13:            else
14:                P.hasNested ← t
15:                if w.loop.parent ≠ nil then
16:                    UnionLoops(P, w.loop.parent)
17:                else
18:                    w.loop.parent ← S
19:                end if
20:            end if
21:        end for
22:    end for
23: end function
```

Algorithmus 2: Berechnung der Schleifengruppen

- Ist w ein η-Knoten, vereinigen wir nicht L_w mit L_v oder der ermittelten umgebenden Gruppe, sondern vermerken diese als Elterngruppe von L_w. Hatte L_w bereits eine Elterngruppe, wissen wir nun, dass die bestehende und die neu ermittelte Elterngruppe identisch sein müssen.

  Im Beispiel gibt es jeweils nur einen Kandidaten für die Elterngruppe, so sorgt etwa die Kante `Call` $\to e_1$ für $L_2.\mathsf{parent} = L_1$.

Die Differenz der Schleifentiefen kann in wohlgeformten vFirm-Graphen nie negativ werden.

Wir modellieren die Schleifengruppen als disjunkte Mengen einer Union-Find-Datenstruktur [Cor05] mit den Funktionen MakeLoop, FindLoop und UnionLoops in Algorithmus 3. Die UnionLoops-Prozedur sorgt neben der Vereinigung der Mengen auch dafür, dass die zusätzlichen Attribute, also die Elterngruppe, die Menge der zugehörigen η-Knoten, die Anzahl der Knoten in der Gruppe und die Information, ob die Gruppe geschachtelte Gruppen enthält, in den neuen Repräsentanten transferiert werden. Haben die zu vereinigenden Mengen unterschiedliche Elterngruppen, werden diese ebenfalls vereinigt.

FindAncestor folgt δ-mal der Referenz zur Elterngruppe. Für $\delta = 0$ gilt $\text{FindAncestor}(L, 0) = \text{FindLoop}(L)$.

Die Funktion MakeLoopCluster in Algorithmus 2 initialisiert jeden Knoten mit einer einelementigen Schleifengruppe.

Anschließend werden für jeden Knoten die oben beschriebenen Äquivalenzen zwischen seiner Schleifengruppe und der Gruppen seiner Vorgänger ermittelt und durch Vereinigung der Mengen in der Datenstruktur vermerkt.

```
24: function MakeLoop(v)
25:     L.link ← L
26:     L.depth ← v.depth + (v.op = η ? 1 : 0)
27:     L.parent ← nil
28:     L.etas ← (v.op = η ? {v} : ∅)
29:     L.nMembers ← 1
30:     L.hasNested ← f
31:     v.loop ← L
32: end function
33:
34: function FindLoop(L)
35:     while L.link ≠ L do
36:         L ← L.link
37:     end while
38:     return L
39: end function
40:
41: function FindAncestor(L, δ)
42:     L ← FindLoop(L)
43:     if δ = 0 then
44:         return L
45:     end if
46:     FindAncestor(L.parent, δ − 1)
47: end function
48:
49: function UnionLoops(L1, L2)
50:     L1 ← FindLoop(L1)
51:     L2 ← FindLoop(L2)
52:
53:     if L1 ≠ L2 then
54:         L2.link ← L1
55:
56:         if L1.parent ≠ nil ∧ L2.parent ≠ nil then
57:             UnionLoops(L1.parent, L2.parent)
58:         else if L2.parent ≠ nil then
59:             L1.link ← L2.parent
60:         end if
61:
62:         if L1.etas ≠ nil ∧ L2.etas ≠ nil then
63:             L1.etas ← L1.etas ∪ L2.etas
64:         else if L2.etas then
65:             L1.etas ← L2.etas
66:         end if
67:
68:         L1.nMembers ← L1.nMembers + L2.nMembers
69:         L1.hasNested ← L2.hasNested | L2.hasNested
70:     end if
71: end function
```

Algorithmus 3: Hilfsfunktionen für Berechnung der Schleifengruppen

5.5.3. Schleifenoptimierung durch Elimination von θ-Knoten?

In einem vFirm-Graphen können Teilgraphen mit θ-Knoten vorkommen, die sich auf den ersten Blick optimieren lassen. Das kanonische Beispiel für solch ein Muster ist ein θ-Knoten $t = \theta(w_0, t)$, der einen schleifeninvarianten Wert explizit zu einer Werteliste hebt. Da t offensichtlich nichts anderes tut, als die implizite Schleifenhebung durchzuführen, liegt seine Elimination nahe. Dadurch kann sich aber die Schleifentiefe seiner Verwender reduzieren - die Transformation hätte nicht-lokale Effekte. Es kann passieren, dass η-Knoten dadurch inkorrekt werden, weil sie Werte mit zu geringen Schleifentiefen an ihren Eingängen vorfinden.

Transformationen dieser Art müssen also von den η-Knoten ausgehen. Der folgende Unterabschnitt stellt ein entsprechendes Verfahren vor.

5.5.4. Ausrollen von Schleifengruppen

```
1:  function UnrollNode(v, i, d)
2:      if v.depth < d then
3:          return v
4:      end if
5:      if v.op = θ then
6:          return UnrollTheta(v, i)
7:      end if
8:
9:      v' ←copy(v)
10:     for all w : v' → w ∈ E do
11:         w ← UnrollNode(w, i)
12:     end for
13:     return v'
14: end function
15:
16: function UnrollTheta(t, i)
17:     j ← |t.iters|
18:     if j = 0 then
19:         t.iters[0] ← t.init
20:         j ← j + 1
21:     end if
22:
23:     while j ≤ i do
24:         t.iters[j] ← UnrollNode(t.next, j − 1)
25:         j ← j + 1
26:     end while
27:     return t.iters[i]
28: end function
```

Algorithmus 4: Ausrollen von Schleifengruppen

In vFirm produziert ein Knoten v mit einer Schleifentiefe $d > 0$ pro Belegung des Indexvektors $[i_1, \ldots, i_d]$ einen Wert, wird aber statisch nur durch einen Knoten dargestellt. Wir wollen nun den Wert des Knotens für einen festen Schleifenindex $i^\star$ der d-ten Schleifentiefe explizit darzustellen. Formal erzeugen wir einen Knoten v' der Schleifentiefe $d-1$, dessen Wert $\text{value}_{[i_0,\ldots,i_{d-1},i^\star]}(v)$ ist.

Den Wert des neuen Knotens können wir aus der Wertfunktion herleiten. Betrachten wir zum Beispiel den Knoten t_7 mit Schleifentiefe 2 in Abbildung 5.18, den wir in der zweiten Iteration auswerten wollen.

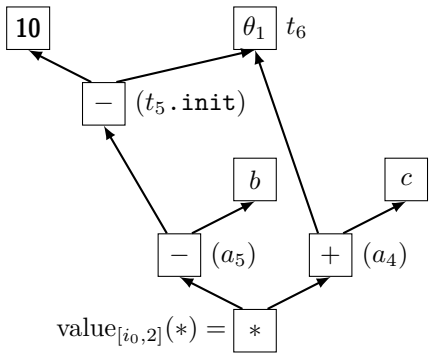


Abbildung 5.19.: Auswertung des Knotens $*$ aus Abbildung 5.18 in der zweiten Iteration. Neben den Knoten ist die ursprüngliche Knotenbezeichnung in Klammern angegeben.

$$\begin{aligned}
\text{value}_{[i_0,2]}(t_7) &= \text{value}_{[i_0,1]}(*) \\
&= \text{value}_{[i_0,1]}(t_5) \cdot \text{value}_{[i_0,1]}(t_4) \\
&= \text{value}_{[i_0,0]}(a_5) \cdot \text{value}_{[i_0,0]}(a_4) \\
&= [\text{value}_{[i_0,0]}(t_5) - \text{value}_{[i_0,0]}(c)] \cdot [\text{value}_{[i_0,0]}(t_4) + \text{value}_{[i_0,0]}(b)] \\
&= [\text{value}_{[i_0]}(-) - \text{value}_{[]}(c)] \cdot [\text{value}_{[i_0]}(t_6) + \text{value}_{[]}(b)]
\end{aligned}$$

Wir sehen, dass die letzte Zeile nur noch aus Werten der Schleifentiefen 0 und 1 besteht. Dies passiert, weil wir die Auswertung des Knotens schrittweise zu den initialen Werten der beteiligten θ-Knoten zurückführen. Die θ-Knoten selbst, die den Wertefluss zwischen den Iterationen herstellen, tauchen in der Berechnung anschließend nicht mehr auf. Wir nennen dieses Verfahren *Ausrollen* aufgrund der Ähnlichkeit zur gleichnamigen traditionellen Schleifenoptimierung [Mor98]. Abbildung 5.19 stellt das Ergebnis des Ausrollens als Teilgraph dar.

Wir wollen das Ausrollen nutzen, um η-Knoten zu eliminieren, indem wir die Terminationsiteration ermitteln und dann den Wert des Knotens am `value`-Eingang in eben dieser Iteration auswerten.

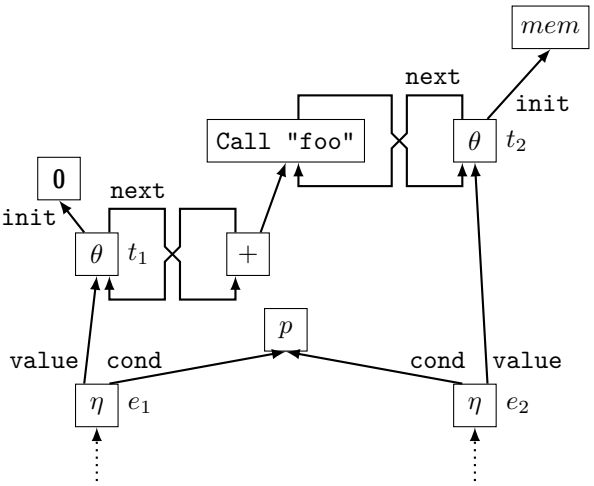
Dazu müssen wir die `value`-Vorgänger *aller* η-Knoten einer Schleifengruppe gemeinsam ausrollen. Ein teilweises Ausrollen ist nicht zulässig, weil das Ausrollen einer Duplikation entspricht. Werden Berechnungen mehrfach ausgeführt, ist dies ineffizient. Enthält die Schleifengruppe aber zustandsverändernde Operationen, existieren durch die Duplikation mehr als ein lebendiger Speicherwert, so dass wir aus dem Graphen keinen Code generieren können.

In Abbildung 5.20 wird nur der `value`-Vorgänger von e_2 in der dritten Iteration ausgerollt. Bei der Auswertung von e_1 werden aber die Rückgabewerte des Funktionsaufrufs aus dem Zyklus von t_2 benötigt. Nach der Transformation ist mehr als ein Speicherwert lebendig. Führt man das Programm dennoch aus, wird die Funktion “foo” sechsmal aufgerufen.

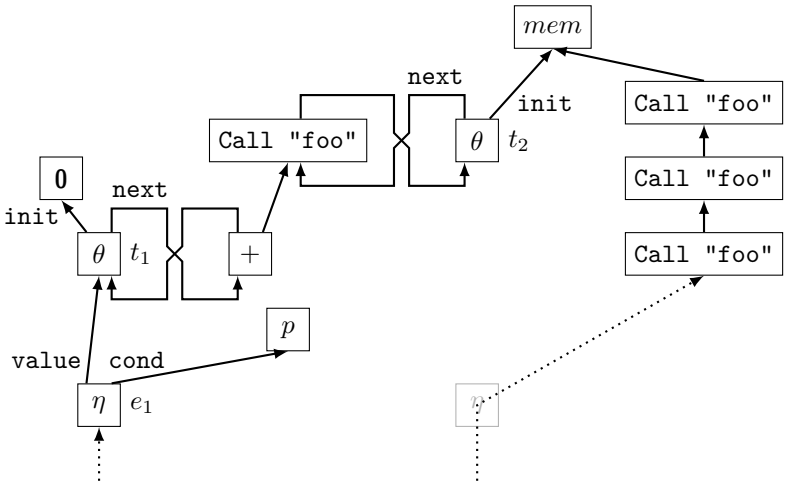
Algorithmus 4 zeigt unser Verfahren zum Ausrollen von Schleifengruppen. Wir fordern, dass die Schleifengruppe keine weiteren, tiefer geschachtelten Gruppen enthält, sowie dass alle η-Knoten denselben Knoten am Bedingungseingang haben. Dies ist der einfachste und am häufigsten auftretende Fall.

Die Prozedur UnrollNode rollt einen Knoten v bezüglich einer Schleifengruppe der Tiefe d in der i-ten Iteration aus. Ist $v.\mathsf{depth} < d$, ist der Knoten schleifeninvariant und kann direkt verwendet werden.

Handelt es sich um einen θ-Knoten, wird die Prozedur UnrollTheta mit dem θ-Knoten und der angeforderten Iteration aufgerufen. Alle θ-Knoten t erhalten ein Attribut $t.\mathsf{iters}$, in dem wir für jeden Iterationsindex den Knoten vermerken, der t's Wert in der entsprechenden Iteration repräsentiert. Für die nullte Iteration ist dies der Knoten am `init`-Eingang. Die j-te Iteration ermitteln wir, indem wir das Ausrollen des `next`-Vorgängers zur $j-1$-ten Iteration anfordern. Dies machen wir so lange, bis wir den Wert von t in der i-ten Iteration kennen.



(a) vorher



(b) teilweise ausgerollt

Abbildung 5.20.: Problematische Situation beim teilweisen Ausrollen Schleifengruppen. Hier gezeigt: Ausrollen von e_2.`value` in Iteration 3.

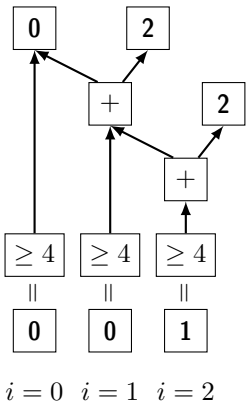


Abbildung 5.21.: Ausrollen des Bedingungsknotens der Schleifengruppe L_3 aus Abbildung 5.18.

Für alle anderen Knotentypen führen wir eine Duplikation von v für die angeforderte Iteration durch und verbinden die Kopie mit Kopien der Vorgänger von v.

η-Knoten können nicht vorkommen, weil wir nur Schleifengruppen behandeln, die keine verschachtelten Gruppen enthalten.

5.5.5. Ermittlung der Terminationsiteration

Damit wir eine Schleifengruppe ausrollen können, müssen wir statisch ermitteln, ob es eine Iteration $i^\star$ gibt, in der der Bedingungsknoten der zur Gruppe gehörenden η-Knoten zu **t** ausgewertet wird. Die η-Knoten fordern dann die Werte ihrer `value`-Vorgänger in Iteration $i^\star$ an.

Wir rollen Bedingungsknoten beginnend bei der nullten Iteration mit aufsteigender Iterationsnummer aus und führen eine Konstantenfaltung auf dem jeweiligen ausgerollten Knoten durch.

Ergibt die Faltung den Wert **t**, handelt es sich bei der aktuellen Iteration um $i^\star$.

Ermitteln wir dagegen den Wert **f**, gehen wir zur nächsten Iterationsnummer weiter[13].

Lässt sich der ausgerollte Bedingungsknoten nicht konstant auswerten, brechen wir ab, weil wir dann nicht garantieren können, dass es genau eine Terminationsiteration gibt.

In Abbildung 5.21 ist das Ausrollen des Bedingungsknotens der Schleifengruppe L_3 aus Abbildung 5.18 gezeigt. Dort ermitteln wir $i^\star = 2$.

5.5.6. Anwendungen des Ausrollens

Ein unbeschränktes Ausrollen von Schleifengruppen kann zu einem exzessiven Code-Wachstum führen. Wir wollen daher die Transformation nur durchführen, wenn entweder $i^\star$ klein ist, oder sich die abgerollten Iterationen durch nachfolgende Optimierungen zusammenfassen lassen.

Frühe Termination

Unroll

Betrachten wir für den ersten Fall einen Knoten v einer auszurollenden Schleifengruppe mit der Tiefe d. v wird maximal[14] einmal für jede Auswertung in der Schleifengruppe zu v_i dupliziert, insgesamt also $i^\star$ mal. Die Auswertungswahrscheinlichkeiten der Kopien vergrößern sich nicht, und die statischen

[13] Für eine praktische Implementierung muss man das Abrollen der Abbruchbedingung nach einer vernünftigen Anzahl von Versuchen aufgeben.

[14] Der Knoten wird nur einmal dupliziert, wenn er von keinem θ-Knoten der Schleifengruppe verwendet wird, sich also beispielsweise zwischen einem η-Knoten und einem θ-Knoten ($\eta \to v \to \theta \nrightarrow^* v$) befindet.

Kosten bleiben ebenfalls gleich, allerdings verringert sich die Schleifentiefe um eins. Für die Summe der Kosten aller Kopien v_j gilt dann:

$$\begin{aligned}\sum_{j=1}^{i^\star} P_{\mathsf{eval}}(v_j) \cdot \omega(v_j) \cdot \mathbb{L}^{d-1} &\leq \sum_{j=1}^{i^\star} P_{\mathsf{eval}}(v) \cdot \omega(v) \cdot \mathbb{L}^{d-1} \\ &= i^\star \cdot P_{\mathsf{eval}}(v) \cdot \omega(v) \cdot \mathbb{L}^{d-1}\end{aligned}$$

Dann schätzen wir ab:

$$\begin{aligned}i^\star \cdot P_{\mathsf{eval}}(v) \cdot \omega(v) \cdot \mathbb{L}^{d-1} &\leq c(v) \\ \Leftrightarrow i^\star &\leq \mathbb{L}\end{aligned}$$

Wird eine Schleifengruppe also seltener iteriert, als wir durchschnittlich annehmen, dürfen wir in unserem Kostenmodell ausrollen. Zusätzlich rollen wir nur Schleifengruppen bis zu einer bestimmten oberen Grenze aus, um die Aufblähung der Codegröße zu verhindern. Für diese Implementierung fordern wir, dass die Schleifengruppe weniger als 32 Knoten hat. Der Effekt des Ausrollens auf das fertig übersetzte Programm ist die Einsparung eines Schleifenkonstrukts und der dafür erforderlichen bedingten Steuerflussoperationen.

Beispiele für die Anwendung des zweiten Falls sind:

Konstantenfaltung UnrollConst

Die abgerollten Iterationen lassen sich mit den bekannten Mechanismen statisch auswerten. Dies ist möglich, wenn alle schleifeninvarianten Knoten, die zur Berechnung der `value`-Eingänge der η-Knoten beitragen, Konstanten sind[15].

Abbildung 5.22a zeigt eine solche Schleifengruppe. Durch die vollständige Konstantenfaltung (Abbildung 5.22b) werden die Kosten des Teilgraphen auf 0 reduziert.

Umordnung von Summen UnrollReassoc

Die abgerollten Iterationen bilden eine Summe (siehe Abschnitt 5.6), die sich umordnen und kompakter darstellen lässt, wie zum Beispiel in Abbildung 5.22c. Die Umordnung lässt sich anwenden, wenn die Schleifengruppe nur einen η-Knoten enthält und die an der Wertberechnung des η-Knotens beteiligten Knoten entweder Integeradditionen, -subtraktionen oder θ-Knoten vom gleichen Typ sind.

Die Knoten der umgebenden Schleifengruppe bilden die n Argumente der Summe. Dafür muss die auszurollende Schleifengruppe mindestens $n-1$ Additionen bzw. Subtraktionen enthalten.

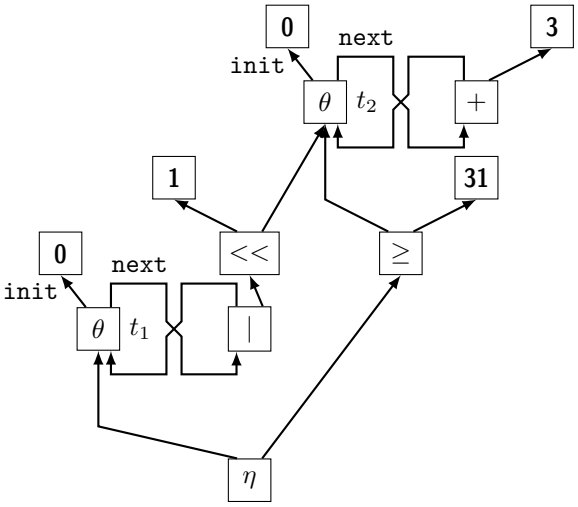
Die entstehende Summe hat maximal n Additionen ($n-1$ für die Argumente, 1 für den konstanten Anteil) und $n-1$ Multiplikationen, die Faktoren für die einzelnen Argumente darstellen, wie die transformierte Version des Beispiels in Abbildung 5.22d zeigt. Die Größe der Terminationsiteration $i^\star$ spielt für die Kosten keine Rolle, weil sie sich nur in den Faktoren der Argumente widerspiegelt. Würde im Beispiel der η-Knoten in Iteration 42 ausgewertet, würden sich die Faktoren von a und b zu 84 und -42 ändern, und der konstante Anteil zu 903 werden.

Wir nehmen wieder an, dass die Knoten in der Schleifengruppe dieselbe Auswertungswahrscheinlichkeit besitzen. Additionen und Subtraktionen haben überdies $\omega(+) = \omega(-) = 1$. Dann gilt für die Kosten der ausgerollten und umgeordneten Summe auf der linken Seite in Relation zu den Kosten der Additionen in der Schleifengruppe auf der rechten Seite der Ungleichung:

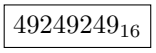
$$\begin{aligned}(2 \cdot n - 1) \cdot \mathbb{L}^{d-1} &\leq (n-1) \cdot \mathbb{L}^d \\ \Leftrightarrow \quad \underbrace{\frac{2 \cdot n - 1}{n-1}}_{\leq 3 \text{ für } n=1,2,\ldots} &\leq \mathbb{L}\end{aligned}$$

Mit unserer Annahme $\mathbb{L} = 10$ ist die Transformation immer kostenmindernd. Begrenzend ist nur der Speicherbedarf des Compilers zur temporären Darstellung der ausgerollten Schleifengruppe, für deren Größe die Implementierung eine sinnvolle obere Schranke besitzen sollte.

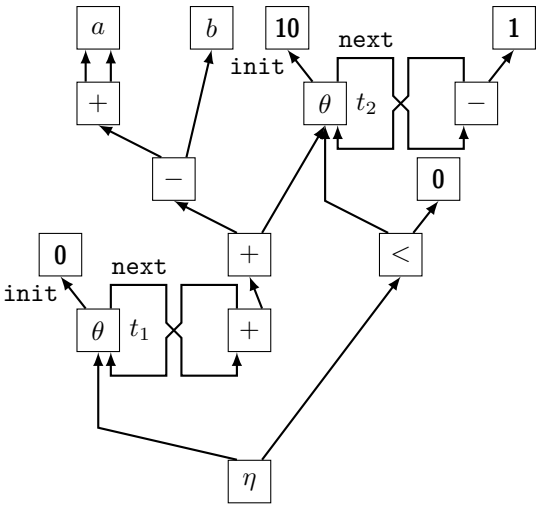
[15] Die Forderung schließt auch das Vorhandensein von zustandsbehafteten Operationen aus, weil ein initialer Zustand immer von einem nicht-konstanten, schleifeninvarianten Knoten kommen muss.



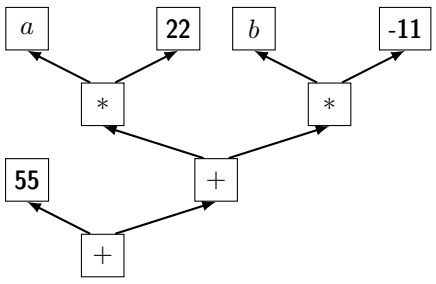
(a) Schleifengruppe mit konstanten Operationen



(b) nach Ausrollen und Konstantenfaltung

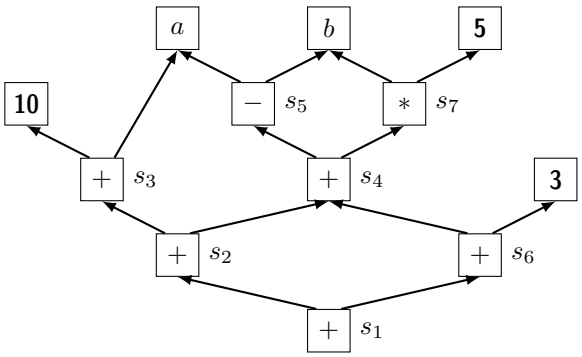


(c) Schleifengruppe bestehend aus Additionen und Subtraktionen

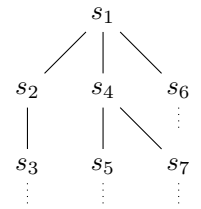


(d) nach Ausrollen und Umordnung

Abbildung 5.22.: Kombinationen der Ausrolltransformation



(a) Graph



(b) zugehöriger Dominatorbaum

$$Args_S = \{a, b\}$$

$$S = \left(s_1, \quad k_S + \sum_{x \in Args_S} n_{x,S} \cdot x\right)$$

$$= (s_1, \quad 13 + 3 \cdot a + 8 \cdot b)$$

(c) Zugehörige Summen-Datenstruktur S

Abbildung 5.23.: Beispiel von Knoten aus *SumNodes* mit einem eindeutigen Wurzelknoten

5.6. Umordnung von Summen

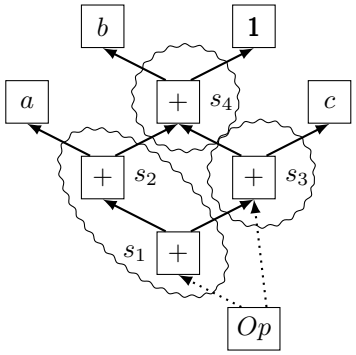
Wir wollen nun die Assoziativität und Kommutativität der Integeraddition ausnutzen, um Teilgraphen bestehend aus Additionen, Subtraktionen und Multiplikationen mit Konstanten zu vereinfachen und die Schleifentiefe der beteiligten Knoten zu verringern.

Diese Optimierung heißt in der englischsprachigen Literatur *Reassociation* [Muc97]. In [Mor98] werden unter anderem Bäume von Additionen als Summe von Produkten dargestellt.

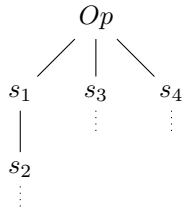
Definition 26.

$$SumNodes = \{v \in V : v.\texttt{op} \in \{\texttt{Add}, \texttt{Sub}\} \vee (v.\texttt{op} = \texttt{Mul} \wedge v.\texttt{right}.\texttt{op} = \texttt{Const}) \wedge v.\texttt{type} = \mathbb{I}\}$$

definiert die Menge der Integeradditionen, -subtraktionen und -multiplikationen mit einem konstanten Faktor.



(a) Graph



(b) zugehöriger Dominatorbaum

$$S_1 = (s_1, \quad 0 + 1 \cdot a + 1 \cdot s_3 + 1 \cdot s_4)$$
$$S_2 = (s_3, \quad 0 + 1 \cdot c + 1 \cdot s_4)$$
$$S_3 = (s_4, \quad 1 + 1 \cdot b)$$

(c) zugehörige Summen-Datenstrukturen

Abbildung 5.24.: Beispiel von Knoten aus *SumNodes* mit mehreren Verwendern

Definition 27 (Summe)**.** Im Kontext der Umordnungstransformation ist eine *Summe* ein maximal großer, zusammenhängender Teilgraph $S \subseteq \mathit{SumNodes}$ mit der Eigenschaft, dass ein ausgewiesener Wurzelknoten r_S alle Knoten $v_S \in S$ dominiert. Alle Knoten der Summe haben den gleichen Integertyp[16]. Andere Knoten, die von Mitgliedern einer Summe verwendet werden, heißen *Argumente* und sind Element der Menge Args_S.

Der von r_S berechnete Wert ist eindeutig bestimmt durch die verrechneten Werte aller konstanten Argumente, der Menge Args_S und einem Faktor $n_{a,S}$ für jedes Argument $a \in \mathit{Args}_S$.

Wir repräsentieren eine Summe daher nicht als Knotenmenge, sondern als Tupel der Form:

$$S = \left(\boldsymbol{r_S} \quad , \quad \boldsymbol{k_S} + \sum_{a \in \mathit{Args}_S} \boldsymbol{n_{a,S}} \cdot a \right)$$

Abbildung 5.23 zeigt eine Summe und ihre Repräsentation als Tupel von Wurzelknoten und Wert. Wir sehen, dass der Graph innerhalb der zur Summe gehörigen Knoten nicht baumartig sein muss; die mehrfache Verwendung von Teilergebnissen ist dort kein Problem. Das Argument a geht über s_3 und s_5 in die Summe ein. Da s_4 zweimal in der Summe verwendet wird, zählt auch s_5 doppelt, und es ergibt sich insgesamt ein Faktor 3 für a.

Ähnliches gilt für b: Als rechtes Argument von s_5 hat der Knoten zunächst den Faktor -2. Mit der doppelten Verwendung von s_7 ergibt sich schließlich der Faktor 8.

Die Forderung nach der Dominanz des Wurzelknotens über die übrigen Knoten schließt aus, dass es innerhalb der Summe Teilsummen gibt, die von Verwendern außerhalb der Summe genutzt werden. In Abbildung 5.24 ist ein Teilgraph mit Knoten aus *SumNodes* dargestellt, der keinen eindeutigen Wurzelknoten besitzt, weil s_1 und s_3 Verwender außerhalb der Summe besitzen. Wir erhalten drei Summen, wobei die Wurzelknoten s_3 und s_4 als Argumente der Summen S_1 beziehungsweise S_2 auftreten.

Interpretieren wir die Summen stattdessen überlappend, also

$$\begin{aligned} S'_1 &= (s_1, \quad 2 + 1 \cdot a + 1 \cdot b + 1 \cdot c) \\ S'_2 &= (s_3, \quad 1 + 1 \cdot b + 1 \cdot c) \\ S'_3 &= (s_4, \quad 1 + 1 \cdot b) \end{aligned}$$

enthalten sie implizit die Duplikation der gemeinsamen Berechnungen. Bei der Rekonstruktion kann nicht garantiert werden, dass sich wieder Teilsummen bilden, die verschmolzen werden können.

5.6.1. Bestimmung der Summen

Die Berechnung der Summe findet in zwei Stufen statt.

Zunächst ermitteln wir mit der Prozedur FindSums in Algorithmus 5 für jeden Knoten v, zu welcher Summe er gehört, und vermerken diese eindeutige Zuordnung im neuen Attribut $v.\mathsf{sum}$. Wir traversieren den Dominatorbaum. v gehört keiner Summe an, wenn $v \notin \mathit{SumNodes}$. Sei also $v \in \mathit{SumNodes}$. Wenn der Elternknoten Mitglied einer Summe ist, erbt v dessen Summe. Ansonsten erzeugen wir eine neue Summe mit v als Wurzelknoten.

Im Beispiel in Abbildung 5.24 ist $\mathit{Op} \notin \mathit{SumNodes}$ und gehört damit keiner Summe an. Die Kinder dieses Knotens im Dominatorbaum s_1, s_3 und s_4 sind die Wurzelknoten von S_1, S_2 und S_3. s_1 ist der Elternknoten von s_2 und $s_1, s_2 \in \mathit{SumNodes}$, daher gehört s_2 ebenfalls zu S_1.

Nachdem die im Graph vorhandenen Summen bekannt sind, müssen wir noch die Argumente erfassen. Jede Summe wird ausgehend vom Wurzelknoten entlang der Datenabhängigkeitskanten von der Prozedur CollectArgs aus Algorithmus 5 traversiert.

Dabei führen wir einen Faktor n als Parameter mit, der zu Beginn 1 ist. Ist der aktuelle Knoten eine Subtraktion oder Multiplikation, wird der Faktor vor dem rekursiven Aufruf angepasst, also negiert oder mit dem Faktor der Multiplikation multipliziert.

[16] In vFirm ist dies automatisch gewährleistet, weil der Wechsel des Integertyps das Vorhandensein eines `Conv`-Knotens erfordert.

```
1: function FindSums(v, Σ)
2:     v.sum ← Σ
3:     for all w ∈ V : idom(w) = v do
4:         σ ← nil
5:         if w ∈ SumNodes then
6:             if Σ = nil then
7:                 σ ← new Sum(r_σ ← w)
8:             else
9:                 σ ← Σ
10:            end if
11:        end if
12:        FindSums(w, σ)
13:    end for
14: end function
15:
16: function CollectArgs(v, n)
17:    if v.op ∈ {Add, Sub} then
18:        if v.left.sum = v.sum then
19:            CollectArgs(v.left, n)
20:        else
21:            InsertArg(v.sum, v.left, n)
22:        end if
23:
24:        n′ ← (v.op = Sub) ? (−n) : n
25:        if v.right.sum = v.sum then
26:            CollectArgs(v.right, n′)
27:        else
28:            InsertArgs(v.sum, v.right, n′)
29:        end if
30:    else  // v.op = Mul
31:        n′ = n · v.right
32:        CollectArgs(v.left, n′)
33:    end if
34: end function
35:
36: function InsertArg(∑, v, n)
37:    if v.op = Const then
38:        k_Σ ← k_Σ + v · n
39:    else
40:        Args_Σ ← Args_Σ ∪ {v}
41:        n_{v,Σ} = n_{v,Σ} + n
42:    end if
43: end function
```

Algorithmus 5: Bestimmung der Summen

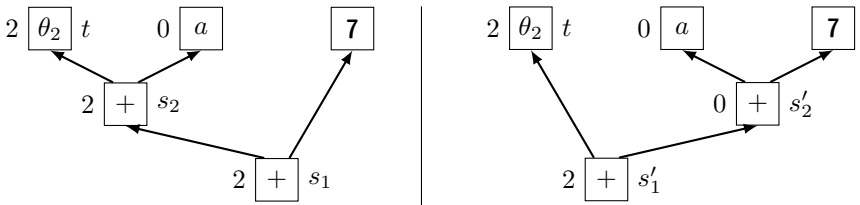
CollectArgs$(s_1, 1)$	$S{=}(s_1, 0)$
CollectArgs$(s_2, 1)$	$S{=}(s_1, 0)$
CollectArgs$(s_3, 1)$	$S{=}(s_1, 0)$
InsertArg$(\mathbf{10}, 1)$	$S{=}(s_1, \mathbf{10})$
InsertArg$(a, 1)$	$S{=}(s_1, 10 + \mathbf{1 \cdot a})$
CollectArgs$(s_4, 1)$	$S{=}(s_1, 10 + 1 \cdot a)$
CollectArgs$(s_5, 1)$	$S{=}(s_1, 10 + 1 \cdot a)$
InsertArg$(a, 1)$	$S{=}(s_1, 10 + \mathbf{2} \cdot a)$
InsertArg$(b, -1)$	$S{=}(s_1, 10 + 2 \cdot a + \mathbf{(-1) \cdot b})$
CollectArgs$(s_7, 1)$	$S{=}(s_1, 10 + 2 \cdot a + (-1) \cdot b)$
InsertArg$(b, 5)$	$S{=}(s_1, 10 + 2 \cdot a + \mathbf{4} \cdot b)$
CollectArgs$(s_6, 1)$	$S{=}(s_1, 10 + 2 \cdot a + 4 \cdot b)$
CollectArgs$(s_4, 1)$	$S{=}(s_1, 10 + 2 \cdot a + 4 \cdot b)$
CollectArgs$(s_5, 1)$	$S{=}(s_1, 10 + 2 \cdot a + 4 \cdot b)$
InsertArg$(a, 1)$	$S{=}(s_1, 10 + \mathbf{3} \cdot a + 4 \cdot b)$
InsertArg$(b, -1)$	$S{=}(s_1, 10 + 3 \cdot a + \mathbf{3} \cdot b)$
CollectArgs$(s_7, 1)$	$S{=}(s_1, 10 + 3 \cdot a + 3 \cdot b)$
InsertArg$(b, 5)$	$S{=}(s_1, 10 + 3 \cdot a + \mathbf{8} \cdot b)$
InsertArg$(\mathbf{3}, 1)$	$S{=}(s_1, \mathbf{13} + 3 \cdot a + 8 \cdot b)$

Abbildung 5.25.: Erfassung der Argumente für die S_1 aus Abbildung 5.23

Knoten, die mehrere Verwendungen innerhalb der Summe haben, werden entsprechend häufiger besucht. Variable Argumente werden von der Prozedur InsertArg gemäß des aktuellen Faktors in die Summendatenstruktur eingetragen, Konstanten werden mit dem Faktor multipliziert und zum konstanten Anteil addiert.

Für das Beispiel in Abbildung 5.23 ist der Ablauf der Erfassung der Argumente in Abbildung 5.25 gezeigt.

Wahrung der Überlaufeigenschaften Der aktuelle Faktor, die Faktoren $n_{a,S}$ in der Summendatenstruktur und der konstante Anteil k_S haben den gleichen Integertyp wie die Knoten der Summe. Dadurch ist sichergestellt, dass beim Verrechnen der Argumente dieselben Überlaufeigenschaften zugrunde liegen wie beim Ausführen der Operationen im Programm.



(a) ungünstige Form (b) günstige Form

Abbildung 5.26.: Verringerung der Schleifentiefe in einer Summe durch Neuordnung der Argumente. Die Schleifentiefe ist links der Knoten gekennzeichnet.

5.6.2. Rekonstruktion

Eine Summe mit N Summanden benötigt $N-1$ binäre Additionen.

Durch das Erfassen der Summen versuchen wir Anzahl der Summanden zu verringern, indem wir konstante Argumente falten und variable Argumente zusammenfassen. Wird dabei der Faktor eines Arguments 0, verkleinert dies die Anzahl der Summanden ebenfalls.

Desweiteren benötigt die "ausgeklammerte" Form des Distributivgesetzes weniger Operationen:

$$\underbrace{x \cdot a_1 + \cdots + x \cdot a_n}_{n \text{ Addition und } n \text{ Multiplikationen}} \quad = \quad \underbrace{x \cdot (a_1 + \cdots + a_n)}_{n \text{ Additionen und 1 Multiplikation}}$$

Diese Form bevorzugen wir bei der Rekonstruktion.

Als zweiter Ansatzpunkt für die Kostenreduktion von Summen in unserem Modell ist, durch geschickte Ordnung der Additionen die Schleifentiefe einiger Knoten zu senken. Daher sollen die Argumente mit der höchsten Schleifentiefe möglichst spät in die Auswertung der Summe einfließen, weil dann Additionen mit Argumenten geringerer Schleifentiefe nicht unnötig häufig stattfinden. In Abbildung 5.26a ist die Summe aus t, a und **7** ungünstig dargestellt. Mit jeder Neuauswertung von t müssen die Addition von a und der Konstante erneut berechnet werden. Beide Knoten s_1, s_2 der Summe haben die Schleifentiefe 2.

Abbildung 5.26b zeigt eine bessere Anordnung: die Addition von a und **7** ist schleifeninvariant und wird daher nur einmal berechnet, die Schleifentiefe von s'_2 ist dementsprechend 0.

Eine Anordnung der Ausdrücke nach der Schleifenzugehörigkeit erfolgt auch in [Mor98]; in [BC94] wird dazu ein Rank definiert, der der Schleifentiefe der vFirm-Darstellung ähnlich ist.

Basierend auf diesen Überlegungen stellen wir folgende Heuristik auf: Zuerst gruppieren wir die variablen Argumente nach ihrer Schleifentiefe. Innerhalb dieser Gruppen fassen wir wiederum die Argumente nach ihren Faktoren zusammen. Anschließend sortieren wir die Argumente noch nach ihrer internen Knotennummer[17].

Daraus konstruieren wird folgendermaßen einen Graphen: Als erstes werden die Argumente innerhalb der Faktor-Gruppen addiert, wir erhalten eine Teilsumme pro Schleifentiefe und Faktor. Jede dieser Teilsummen wird mit dem entsprechenden Faktor multipliziert, und die Multiplikationen werden wiederum zu einer Teilsumme pro Schleifentiefe addiert. Zu letzt addieren wir die Teilsummen pro Schleifentiefe aufsteigend sortiert nach der Schleifentiefe auf - der konstante Anteil bildet mit der Teilsumme der kleinsten auftretenden Schleifentiefe den Anfang.

Betrachten wir die Rekonstruktion am Beispiel der Summe links oben in Abbildung 5.27. Die Gruppierung nach Schleifentiefe ist gegenüber der Gruppierung nach Faktoren priorisiert; deswegen befindet sich a nicht in einer Gruppe mit c, obwohl beide mit dem Faktor 4 in die Summe eingehen. Abbildung 5.27a zeigt das Ergebnis der Gruppierung.

[17] Diese ist zwar willkürlich, stellt aber eine Form der Normalisierung dar.

$S = (r_S, \quad 11 + 4 \cdot a + 1 \cdot b + 4 \cdot c + 3 \cdot t_1 + 1 \cdot t_2)$
mit $t_2.\mathsf{depth} = 2, \quad a, t_1.\mathsf{depth} = 1, \quad b, c.\mathsf{depth} = 0$

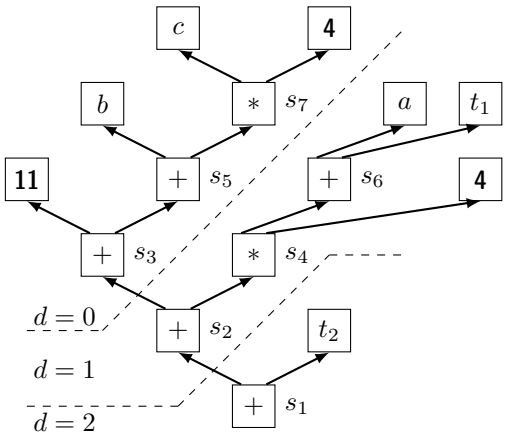
$\Rightarrow$ Konstanter Anteil: 11

$\Rightarrow$ Schleifentiefe $d = 0$
- Faktor 1: b
- Faktor 4: c

$\Rightarrow$ Schleifentiefe $d = 1$
- Faktor 4: a, t_1

$\Rightarrow$ Schleifentiefe $d = 2$
- Faktor 1: t_2

(a) Anordnung der Summe



(b) rekonstruierter Graph

Abbildung 5.27.: Rekonstruktion einer Summe

Nun können wir die Teilsummen konstruieren. s_5 ist die Teilsumme der Schleifentiefe 0 und addiert die Teilsummen der Faktoren 1 sowie 4, welche beide aber nur ein Argument enthalten. s_4 ist die Teilsumme der Schleifentiefe 1, die der Teilsumme des Faktors 4 entspricht, welche wiederum a und t_1 enthält. Für die Schleifentiefe 2 gibt es nur ein Argument, somit ist t_2 die Teilsumme der Tiefe 2. Schließlich addiert s_3 den konstanten Anteil und die Teilsumme der Schleifentiefe 0, s_2 addiert s_3 und die Teilsumme der Schleifentiefe 1 und s_1 addiert s_2 und die Teilsumme der Schleifentiefe 2.

Kosten

Betrachten wir eine Summe

$$S = \left(r_S, \quad k_S + \sum_{a \in Args_S} n_{a,S} \cdot a \right)$$

Die Menge

$$D = \{a.\mathsf{depth} \mid a \in Args_S\}$$

enthält alle auftretenden Schleifentiefen der Argumente von S. Für eine bestimmte Schleifentiefe d beschreibt

$$N_d = \{n_{a,S} \mid a \in Args_S \wedge a.\mathsf{depth} = d\}$$

die Menge der Faktoren der Argumente dieser Tiefe. Schließlich ist

$$V_{d,n} = \{a \mid a \in Args_S \wedge a.\mathsf{depth} = d \wedge n_{a,S} = n\}$$

die Menge der Argumente der Schleifentiefe d, die mit dem Faktor n auftreten. Rekonstruieren wir S gemäß der vorgestellten Heuristik, betragen die Kosten in unserem Kostenmodell

$$c(S) = \underbrace{(k_S \neq 0?) \cdot \mathbb{L}^{\min D}}_{(1)} + \underbrace{\sum_{d \in D} \mathbb{L}^d \cdot \left[\underbrace{(d \neq \min D?)}_{(3)} + \underbrace{|N_d| - 1}_{(4)} + \underbrace{\sum_{n \in N_d} \left(\underbrace{(n \neq 1?)}_{(6)} + \underbrace{|V_{d,n} - 1|}_{(7)} \right)}_{(5)} \right]}_{(2)}$$

Das Ergebnis des Fragezeichenoperators sei 1, falls die davorstehende Bedingung gilt, und sonst 0.

In der Formel steht (1) für die Addition des konstanten Anteils mit der Teilsumme der kleinsten Schleifentiefe, falls vorhanden. (2) beschreibt die Kosten der Teilsummen pro Schleifentiefe, die sich als Produkt von $\mathbb{L}^d$ und der Anzahl der Knoten der Tiefe berechnen. Zunächst muss eine Teilsumme pro Tiefe mit der Teilsumme der nächstkleineren Tiefe addiert werden (3). Anschließend werden die Teilsummen pro Faktor aufaddiert (4). Eine Teilsumme pro Faktor (5) besteht aus der Multiplikation, falls der Faktor nicht 1 ist (6), und der Addition der Argumente (7). Auswertungswahrscheinlichkeit und statische Kosten sind für alle erzeugten Additionen gleich, und aus Gründen der Übersichtlichkeit hier weggelassen.

Im Beispiel in Abbildung 5.27 ergibt die Kostenrechnung:

$$\begin{aligned}
D &= \{0, 1, 2\} \\
N_0 &= \{1, 4\} \quad N_1 = \{4\} \quad N_2 = \{1\} \\
V_{0,1} &= \{b\} \quad V_{0,4} = \{c\} \quad V_{1,4} = \{a, t_1\} \quad V_{2,1} = \{t_2\} \\
c(S) &= 1 \cdot \mathbb{L}^0 + \mathbb{L}^0 \cdot (|N_0| - 1 + |V_{0,1}| - 1 + 1 + |V_{0,4}| - 1) + \mathbb{L}^1 \cdot (1 + |N_1| - 1 + 1 + |V_{1,4}| - 1) \\
&\quad + \mathbb{L}^2 \cdot (1 + |N_2| - 1 + |V_{2,1}| - 1) \\
&= \mathbb{L}^2 + 3 \cdot \mathbb{L} + 3
\end{aligned}$$

Die Heuristik ist durch die Priorisierung auf die Reduktion der Schleifentiefe in vielen Fällen gewinnbringend für die Kosten des Programms; sie findet aber nicht in allen Fällen eine kostenminimale Struktur. Ein worst-case-Beispiel sind Summen, in denen alle Argumente den gleichen, von 1 verschiedenen Faktor haben, und alle Argumente paarweise verschiedene Schleifentiefen besitzen.

Um eine Verbesserung der Kosten zu gewährleisten, berechnen wir beim Ermitteln der Summe ihre Kosten und vergleichen diese vor der Rekonstruktion mit obiger Kostenrechnung.

Transformation Reassoc

Hat die rekonstruierte Summe geringere Kosten als die ursprüngliche Struktur, ersetzen wir den Wurzelknoten durch die letzte Teilsumme pro Schleifentiefe.

5.6.3. Adressrechnungen

Wir wollen das Verfahren nun noch für Adressrechnungen erweitern, die in der vFirm-Darstellung einen eigenen Typ haben, prinzipiell aber wie vorzeichenlose Ganzzahlarithmetik funktionieren. Die Erweiterung ist wichtig, um beispielsweise auch bei den bereits erwähnten Adressrechnungen für Arrayzugriffe schleifeninvariante Teilausdrücke zu erkennen.

Die Signaturen für die Addition und Subtraktion (siehe Tabelle 3.2 auf Seite 19) erlauben bei Adressoperationen, dass die Argumente einen anderen Typ haben als die Operation selbst. Im Gegensatz zu unterschiedlich großen Integertypen enthält die Zwischendarstellung hierbei keine expliziten `Conv`-Knoten. Außerdem existiert keine Multiplikation für Adressen.

Für Summen des Adresstyps treffen wir daher folgende Anpassungen:

- Die Faktoren der Argumente werden anstatt als Werte des Adresstyps als Integerzahlen der gleichen Bitbreite gespeichert und die erforderlichen Rechnungen bei der Erfassung der Summe werden gemäß den Regeln dieses Typs durchgeführt.
- Bei der Rekonstruktion fügen wir explizit `Conv`-Knoten ein, falls die Typen der Argumente nicht zum Typ des Wurzelknotens passen. Diese Konversionen haben in unserem Kostenmodell statische Kosten ω von 0, weil sie zwischen Ganzzahlen mit gleicher Bitbreite umwandeln und daher zur Laufzeit keinen Effekt haben. Die Kostenrechnung für die Transformation bleibt weiterhin gültig.

6. Auswertung

In diesem Kapitel überprüfen wir, dass mit einer Verbesserung der Kostenfunktion eine Verbesserung der Laufzeit von Testprogrammen einhergeht und dass die beschriebenen Optimierungen in Programmcode aus der Praxis Anwendung finden. Anschließend folgt eine Bewertung der Implementierung der Optimierungsphase.

6.1. Laufzeitmessung an Testprogrammen

Zunächst wollen wir zeigen, dass die Wahl der Kostenfunktion sinnvoll war, dass also eine Reduktion der Kosten auch in einer messbaren Verbesserung der Ausführungszeit des Programms resultiert.

6.1.1. Versuchsaufbau

Wir führen die Laufzeitmessungen an Testfunktionen `eval1` bis `eval4` durch, die die verschiedenen Ansätze zur Kostenreduktion zeigen sollen:

- `eval1` (Listing B.1): Elimination von Knoten durch die Umordnung der Summe und Ausnutzung von Konstantenfaltung und Distributivität.
- `eval2` (Listing B.2): Vereinfachung des impliziten Steuerflusses durch die Elimination von γ-Knoten und Verringerung der Auswertungswahrscheinlichkeit von Knoten durch die γ-Distribution.
- `eval3` (Listing B.3): Verringerung der Schleifentiefe von Knoten durch die Summenumordnung und die Optimierung eines additiven θ-Knotens.
- `eval4` (Listing B.4): Kombination der Verringerung der Auswertungswahrscheinlichkeit und Schleifentiefe einiger Knoten.

Dabei vergleichen wir die Laufzeit des optimierten Testprogramms mit der des unoptimierten Programms, das nur in die vFirm-Darstellung transformiert und anschließend wieder abgebaut wurde.

Wir betrachten die Laufzeit vieler Aufrufe der `eval`-Funktionen aus einer Schleife heraus[1]. Die verstrichene Zeit messen wir mit der `time`-Funktion der "Bourne Again Shell" (bash); wir verwenden dabei den Wert für die "user time", das heißt, der Laufzeit des Programms abzüglich der Zeit, die das Programm für Systemaufrufe verbraucht hat. Unsere Programme führen während der Messung mit Ausnahme der Ausgabe ganz am Ende der `main`-Funktion keine Systemaufrufe durch.

Die Messungen werden auf einem mit 3,2 GHz getakteten Core i3-Prozessor von Intel unter Ubuntu Linux 10.04 durchgeführt. Die Ergebnisse beziehen sich auf die kleinste Laufzeit aus 10 Programmläufen.

6.1.2. Messergebnisse

Die Ergebnisse der Laufzeitmessungen zeigt Tabelle 6.2. Die darin verwendeten Bezeichnungen für die Transformationen sind in Tabelle 6.1 aufgeführt.

Wir sehen, dass ein qualitativer Zusammenhang zwischen der Verbesserung der Kosten und der Verbesserung der Laufzeiten besteht.

Wir erwarten allerdings keine quantitative Abhängigkeit, da der Einfluss der Codegenerierung und der Eigenschaften der Prozessorarchitektur auf die reale Laufzeit eines Programms groß sind. Gerade diese Aspekte haben wir aber in unserem Kostenmodell abstrahiert.

[1] siehe `main`-Funktion in den Listings: Das erste Argument `N = argv[1]` des Aufrufs bestimmt die Anzahl der Iterationen, insgesamt werden N^4 Iterationen durchlaufen

Transformation	Beschreibung
ConstCond	Elimination von γ-Knoten mit konstantem Bedingungsknoten
SameArg	Elimination von γ-Knoten mit identischen `true`- und `false`-Argumenten
ModeBGamma	Elimination von booleschen γ-Knoten mit konstanten Argumenten
NegCond	Vertauschung von `true`- und `false`-Argumenten eines γ-Knotens, an dessen Bedingungseingang sich eine Negation befindet
GammaOnCond	Transformation eines γ-Knotens am Bedingungseingang eines weiteren γ-Knotens
GammaCF	γ-Distribution mit anschließender Konstantenfaltung
GammaCSE	γ-Distribution mit anschließender Knotenverschmelzung
GammaLoop	γ-Distribution, die einen schleifeninvarianten Ausdruck extrahiert
Reassoc	Kostenmindernde Umordnung einer Summe
ERG-Eval	Auswertung eines Bedingungsknotens durch die Pfadinformationsanalyse
ERG-Implied	Auswertung eines Bedingungsknotens durch eine Implikation (enthält nicht die Fälle, die sich direkt aufgrund des Prädikats auswerten ließen)
ERG-Dup	Duplikation, um Pfadinformation zu erhalten
ERG-Constified	Ersetzen eines Arguments durch eine Konstante aufgrund der Pfadinformationen
Unroll	Ausrollen einer hinreichend kleinen Schleifengruppe
UnrollConst	Ausrollen einer konstanten Schleifengruppe
UnrollReassoc	Ausrollen einer Schleifengruppe mit anschließender Summenumordnung
StrengthRed	Optimierung eines additiven θ-Knotens
LFTR	Anpassung eines Vergleichsoperation

Tabelle 6.1.: Bezeichnungen der Transformationen

Name	Iter.	Optimierungen	Kosten			Laufzeit [sec]		
			vorher	nachher	%	vorher	nachher	%
`eval1`	250^4	$1 \times$ Reassoc	11	6	-45 %	17,1	14,7	-14 %
`eval2`	250^4	$1 \times$ GammaCF $1 \times$ GammaCSE $3 \times$ ERG-Eval $13 \times$ ERG-Dup $12 \times$ ERG-Implied	20	13	-32 %	19,7	13,6	-30 %
`eval3`	150^4	$1 \times$ Reassoc $1 \times$ StrengthRed $1 \times$ LFTR	122	65	-47 %	12,7	9,8	-22 %
`eval4`	150^4	$1 \times$ GammaOnCond $2 \times$ GammaLoop	148	123	-17 %	19,0	16,0	-15 %

Tabelle 6.2.: Ergebnisse der Laufzeitmessungen

```
1  int loop_dup(int a, int b) {
2   int i, j = 0;
3   for (i = 0; i < a; i++) {
4    j = foo(i);
5    if (i == b) {
6     printf("%d\n", j);
7     break;
8    }
9   }
10  return 0;
11 }
```

Listing 6.1: Eine Funktion, die eine Duplikation der Schleife während des Abbaus der vFirm-Darstellung auslöst.

6.1.3. Probleme bei Laufzeitmessungen an realen Benchmarks

Die Durchführung von Laufzeitmessungen mittels etablierter Benchmarks, wie beispielsweise den Programmen aus SPEC CINT2000 [spe], ist mit der derzeitigen vFirm-Implementierung nicht möglich.

Liebe [Lie11] beschreibt, dass das verwendete Abbauverfahren Schleifen duplizieren muss, wenn sie unter verschiedenen Gatingbedingungen ausgewertet werden. Bei zustandsfreien Schleifen ist das ineffizient, weil Berechnungen mehrfach ausgeführt werden. Die Duplikation von zustandsbehaften Schleifen führt jedoch zu einem inkorrekten Programm, weil mehrere Zustandswerte gleichzeitig lebendig sind.

Eine Duplikation von Schleifen ist schon bei sehr einfachen Programm wie dem in Listing 6.1 erforderlich. Konstrukte dieser Art kommen sehr häufig in realem Programmcode vor, so dass diese Problematik die Übersetzbarkeit von Benchmarks stark einschränkt.

Während der Bearbeitungszeit sind mehrere Versuche unternommen worden, um den vFirm-Abbau entsprechend zu erweitern. Leider konnten nur kleinere Fehler korrigiert werden. Das Hauptproblem der Schleifenduplikation ist konzeptueller Natur und konnte in der verfügbaren Zeit nicht gelöst werden.

6.2. Effektivität der Optimierungen

Nun wollen wir zeigen, dass die Optimierungen auf realen Programmcode angewendet werden können und positive Auswirkungen auf die Kosten der transformierten Funktionen haben.

6.2.1. Testprogramme

Die Programme zur Bewertung der Effektivität der Optimierungsphase stammen aus der Testumgebung des LLVM-Compilers [llv]. Dort sind synthetische Benchmarks und Programmcode aus vielen verschiedenen Quellen zusammengetragen.

Einige Quelldateien lassen sich aufgrund von Syntaxfehlern nicht übersetzen. Hinzu kommen folgende Beschränkungen der vFirm-Implementierung, die ebenfalls das Überspringen der entsprechenden Funktionen auslösen.

Irreduzibler Steuerfluss Schleifen ohne eindeutigen Kopf, das heißt mit mehreren Eintrittsstellen, können derzeit nicht in die vFirm-Darstellung transformiert werden [Lie11].

Nichttermination Im Rahmen der Einführung der vFirm-Darstellung haben wir bereits erwähnt, dass die verwendeten Schleifenkonstrukte nicht explizit zustandsbehaftet sind. Dies hat zur Folge, dass eine Endlosschleife, deren Werte keine Verwender außerhalb der Schleife besitzen, als tot angesehen und entsprechend eliminiert wird. Es können bezüglich der Schleifentiefe inkorrekte Graphen entstehen, wenn benötigte η-Knoten fehlen und der `Return`-Knoten eine Schleifentiefe > 0 erhält.

Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
besser	794	299	181	133	112	96	92	92	77
unverändert	4803								
schlechter	19	3	1	0					

(a) Veränderung der Kostenfunktionen der Graphen

Transformation	Anwendungen in...			Transformation	Anwendungen in...		
	Graphen	proz.	absolut		Graphen	proz.	absolut
ConstCond	275	4,9 %	522	ERG-Eval	107	1,9 %	180
SameArg	5	0,1 %	7	ERG-Dup	0	0,0 %	0
ModeBGamma	106	1,9 %	119	ERG-Implied	88	1,6 %	215
NegCond	8	0,1 %	18	ERG-Constified	40	0,7 %	66
GammaOnCond	54	1,0 %	186	Unroll	131	2,3 %	236
GammaCF	203	3,6 %	622	UnrollConst	1	0,0 %	1
GammaCSE	76	1,4 %	160	UnrollReassoc	0	0,0 %	0
GammaLoop	66	1,2 %	197	StrengthRed	298	5,3 %	690
Reassoc	57	1,0 %	102	LFTR	253	4,5 %	569

(b) Anwendungen der Transformationen

Tabelle 6.3.: Ergbnisse für abgeschaltete Knotenduplikation.

Fehlende `Return`-Knoten Funktionen ohne `Return`-Knoten können wir nicht darstellen, weil der definierte Startpunkt für die Auswertung des Graphen fehlt. Ein Beispiel dafür sind Funktionen, die ausschließlich durch einen Aufruf der Systemfunktion `exit` verlassen werden.

Hoher Ressourcenverbrauch Bei sehr komplizierten Eingabeprogrammen benötigen die Aufbauphase der Darstellung sowie die Berechnung der Gatingbedingungen für die Kostenfunktion unverhältnismäßig viele Resourcen. Wir überspringen Funktionen, die nicht innerhalb von 60 Sekunden und mit 1 GiB Speicher zu übersetzen sind.

Nach Abzug der betroffenen Funktionen bleiben 5616 Graphen übrig. Diese Graphen werden auf Veränderungen der Kostenfunktion durch die Optimierungsphase untersucht. Aufgrund der Probleme im Abbau der vFirm-Darstellung brechen wir die Übersetzung nach der Optimierung ab.

6.2.2. Ergebnisse

Betrachten wir zunächst die Ergebnisse der Optimierungsphase mit abgeschalteter Knotenduplikation während der Elimination redundanter γ-Knoten in Tabelle 6.3.

Die Transformationen haben fast 800 Graphen bezüglich der Kostenfunktion verbessert, und weniger als 20 Graphen verschlechtert. Bei den betroffenen Graphen wurden die Transformationen GammaOnCond oder ERG-Implied durchgeführt, bei denen wir die Problematik bezüglich falscher Annahmen der Gleichverteilung der Werte von Bedingungsknoten in unserem Kostenmodell beschrieben haben.

Alle Transformationen mit Ausnahme der Kombination von Schleifenausrollen und Umordnungstransformation wurden angewandt. Es fällt auf, dass die Summe der Anwendungen von ERG-Eval und ERG-Implied größer ist als die absolute Anzahl der Anwendungen von ConstCond, welche nach Auswertung eines Bedingungsknotens erfolgen sollte. In manchen Fällen sind einige γ-Knoten nach

Dist.	Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
0	besser	1005	400	253	189	151	115	101	101	79
	unverändert	4588								
	schlechter	23	4	2	0					
1	besser	1029	419	256	189	151	115	101	101	79
	unverändert	4560								
	schlechter	27	3	2	0					
3	besser	1115	447	267	198	158	118	103	103	79
	unverändert	4458								
	schlechter	43	6	4	1					
5	besser	1124	451	274	201	158	118	103	103	79
	unverändert	4442								
	schlechter	50	8	7	1					

(a) Veränderung der Kostenfunktionen der Graphen

Dist.	Transf.	Anwendungen in...			Transformation	Anwendungen in...		
		Graphen	proz.	absolut		Graphen	proz.	absolut
0	ConstCond	555	9,9 %	5656	ERG-Eval	374	6,7 %	12421
	SameArg	211	3,8 %	5310	ERG-Dup	333	5,9 %	14857
					ERG-Implied	170	3,0 %	3470
					ERG-Constified	41	0,7 %	70
1	ConstCond	586	10,4 %	6608	ERG-Eval	410	7,3 %	13340
	SameArg	220	3,9 %	5358	ERG-Dup	373	6,6 %	16479
					ERG-Implied	177	3,2 %	3579
					ERG-Constified	43	0,8 %	72
3	ConstCond	707	12,6 %	13018	ERG-Eval	539	9,6 %	19566
	SameArg	244	4,3 %	6203	ERG-Dup	513	9,1 %	29031
					ERG-Implied	193	3,4 %	4752
					ERG-Constified	73	1,3 %	250
5	ConstCond	723	12,9 %	25004	ERG-Eval	565	10,1 %	38370
	SameArg	249	4,4 %	9577	ERG-Dup	534	9,5 %	65954
					ERG-Implied	200	3,6 %	7296
					ERG-Constified	73	1,3 %	1810

(b) Anwendungen der Transformationen

Tabelle 6.4.: Auszug aus den Ergbnissen für Knotenduplikation mit den Distanzen 0, 1, 3, und 5

Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
besser	748	231	76	36	19	4	4	4	2
unverändert	4847								
schlechter	21	5	3	1					

(a) Veränderung der Kostenfunktionen der Graphen

Transformation	Anwendungen in...			Transformation	Anwendungen in...		
	Graphen	proz.	absolut		Graphen	proz.	absolut
StrengthRed	355	6,3 %	821	LFTR	310	5,5 %	700

(b) Anwendungen der Transformationen

Tabelle 6.5.: Auszug aus dem Ergbniss mit deaktivierter Knotenduplikation und Ausroll-Transformation.

Veränderung ihres Prädikats äquivalent, so dass zuerst eine Knotenverschmelzung und dann erst die Ersetzung des γ-Knotens erfolgt.

Fazit Die in dieser Arbeit vorgestellte Optimierungsphase hat in vielen Fällen eine deutlich positive Wirkung auf die Darstellung der Programme, und dies, obwohl wir fast ausschließlich konservative Transformationen durchführen, in dem Sinne, dass wir zeigen konnten, dass die Transformationen die Kosten nie verschlechtern.

6.2.3. Auswirkungen der Knotenduplikation

Wir konnten bei der Betrachtung der Kostenbilanz der Knotenduplikation in Abschnitt 5.4.3 aufgrund der nicht-lokalen Auswirkungen nicht die Profitablität der Transformation beweisen, haben aber gesehen, dass im dort beschriebenen Beispiel eine Verbesserung der Kosten eingetreten ist.

Nun betrachten wir die Auswirkungen der Knotenduplikation mit maximaler Distanz 0, 1, 3 und 5; Tabelle 6.4 enthält Auszüge[2] aus den Ergebnissen.

Zunächst fällt auf, dass bereits die Aktivierung der Duplikation mit maximaler Distanz 0 die Anzahl der verbesserten Graphen deutlich erhöht, und nur 4 Graphen zusätzlich verschlechtert.

Die sehr hohe absolute Anzahl der Anwendungen der Elimination redundanter γ-Knoten wird durch einen einzelnen Graphen verursacht, der alleine für über 12000 × ERG-Dup, 9000 × ERG-Eval und 2800 × ERG-Implied sorgt. Auf diesen Graphen gehen wir gleich noch im Rahmen einer Fallstudie ein.

Eine weitere Erhöhung der Distanz sorgt zwar für eine Erhöhung der Anzahl der verbesserten Graphen, gleichzeitig steigt aber auch die Anzahl der verschlechterten Graphen an. Die Verteilung der Graphen verändert sich allerdings positiv. Mit einer Distanz von 0 erhalten wir 400 Graphen, die sich um mindestens 5 % verbessert haben, wohingegen mit einer Distanz von 5 schon 451 Graphen diese Marke erreichen. 12 zusätzliche Graphen wurden um > 15% verbessert.

Fazit Mit allen getesteten Distanzen erzielt die Knotenduplikation eine Verbesserung der Kosten in mehr Graphen und geht mit einer moderaten Verschlechterung in wenigen Graphen einher. Der größte Effekt liegt allerdings schon in der Aktivierung der Duplikation mit Distanz 0. Diese Transformation können wir aber auch lokal durchführen, so dass die Analyse der erwarteten Bedingungsknoten entfallen kann.

[2] Die Anwendbarkeit der anderen Optimierungen verändert sich nicht wesentlich; die vollständigen Ergebnisse befinden sich um Anhang in den Tabellen A.1 bis A.4.

6.2.4. Auswirkungen des Schleifenausrollens

Es fällt auf, dass bei den Ergebnissen in Tabelle 6.3 77 Graphen eine Verbesserung der Kostenfunktion um mehr als 50 % erfahren haben. Ein Großteil dieser Verbesserungen geht auf die Transformation Unroll zurück, die die Schleifentiefe von tief verschachtelten Knoten senkt, wie wir in Tabelle 6.5 sehen. Dort ist ein Auszug der Ergebnisse ohne Ausrolltransformation und ohne Knotenduplikation gegeben.

Der Grund für die große Verbesserung in unserem Kostenmodell ist, dass die Schleifentiefe exponentiell in die Kosten eingeht. Somit bedeutet schon die Reduktion der Schleifentiefe um eins, dass die Kosten der Kopie eines ausgerollten Knotens nur noch ein Zehntel seiner ursprünglichen Kosten betragen. Zudem werden θ-Knoten eliminiert.

Wir sehen aber auch, dass das Schleifenausrollen so gut wie keine Kostenverschlechterungen verschleiert. Durch die Deaktivierung der Transformation treten nur zwei Graphen zusätzlich auf, die eine Verschlechterung der Kosten erfahren haben.

Fazit Die Kostenverbesserungen des Schleifenausrollens sind quantitativ im Vergleich zu den Effekten der anderen Optimierungen sicherlich zu groß, es handelt sich aber dennoch um eine gewinnbringende Transformation.

6.2.5. Fallstudien

Wir betrachten nun bemerkenswerte Anwendungen unserer Optimierungen an konkreten Funktionen.

Elimination redundanter γ-Knoten

Im Anhang in Listing C.1 findet sich ein Auszug der Funktion `MTDecodeP` aus `TimberWolfMC/mt.dc`. Sie ist für eine sehr große Anzahl der Anwendungen von ERG-Eval, ERG-Implied und ERG-Dup bei aktivierter Knotenduplikation verantwortlich. Wir wollen untersuchen, welchen Effekt die Elimination redundanter γ-Knoten für diese Funktion hat.

Interessant sind dabei die Verzweigungen: Sie enthalten Vergleiche der Variablen `a`, `b`, `c` und `d` mit Konstanten, die faul ausgewertet werden.

Die erste Verzweigung wird durch die vier γ-Knoten g_1 bis g_4 in Abbildung 6.1 dargestellt, die zweite Verzweigung entsprechend durch g_5 bis g_8. Die Rümpfe der Verzweigungen seien in dem Graphen als Knoten `/* A */` und `/* B */` vertreten.

Die Analyse ergibt zunächst keine Pfadinformation bei g_5, wir sehen aber, dass alle Verwender Informationen zur Bedingung $\langle a = 0 \rangle$ liefern. Nach Duplikation von g_5 und Elimination sämtlicher Kopien erhalten wir den Graphen rechts oben in der Abbildung.

Nun haben wir eine ähnliche Situation bei g_6. Die Verwender g_3 und g_4 liefern die Pfadinformation $b \in [1, 1]$, mit der wir den Vergleich $\langle b = -1 \rangle$ zu **f** auswerten können. Wir duplizieren g_6 und eliminieren zwei Kopien; mit der Pfadinformation von g_2 können wir g_6 nicht auswerten, der Knoten bleibt unverändert im Graph. g_3 und g_4 verwenden anschließend g_9.

Wir sehen, dass g_9 ebenfalls nur Verwender mit Pfadinformationen bezüglich `a` hat. Das Verfahren geht dort nun analog weiter.

Durch die Transformation verändern wir die Struktur der Vergleiche. Bezogen auf den Quelltext springen wir beispielsweise nach der Auswertung von `a == 0` zu **f** in Zeile 3 nicht zur nächsten Verzweigung in Zeile 9, sondern direkt zu Zeile 25. Ebenso springen wir für `c == 1` = **f** in Zeile 3 nicht zu Zeile 9, sondern zu Zeile 15, weil wir `b == 1` schon getestet haben.

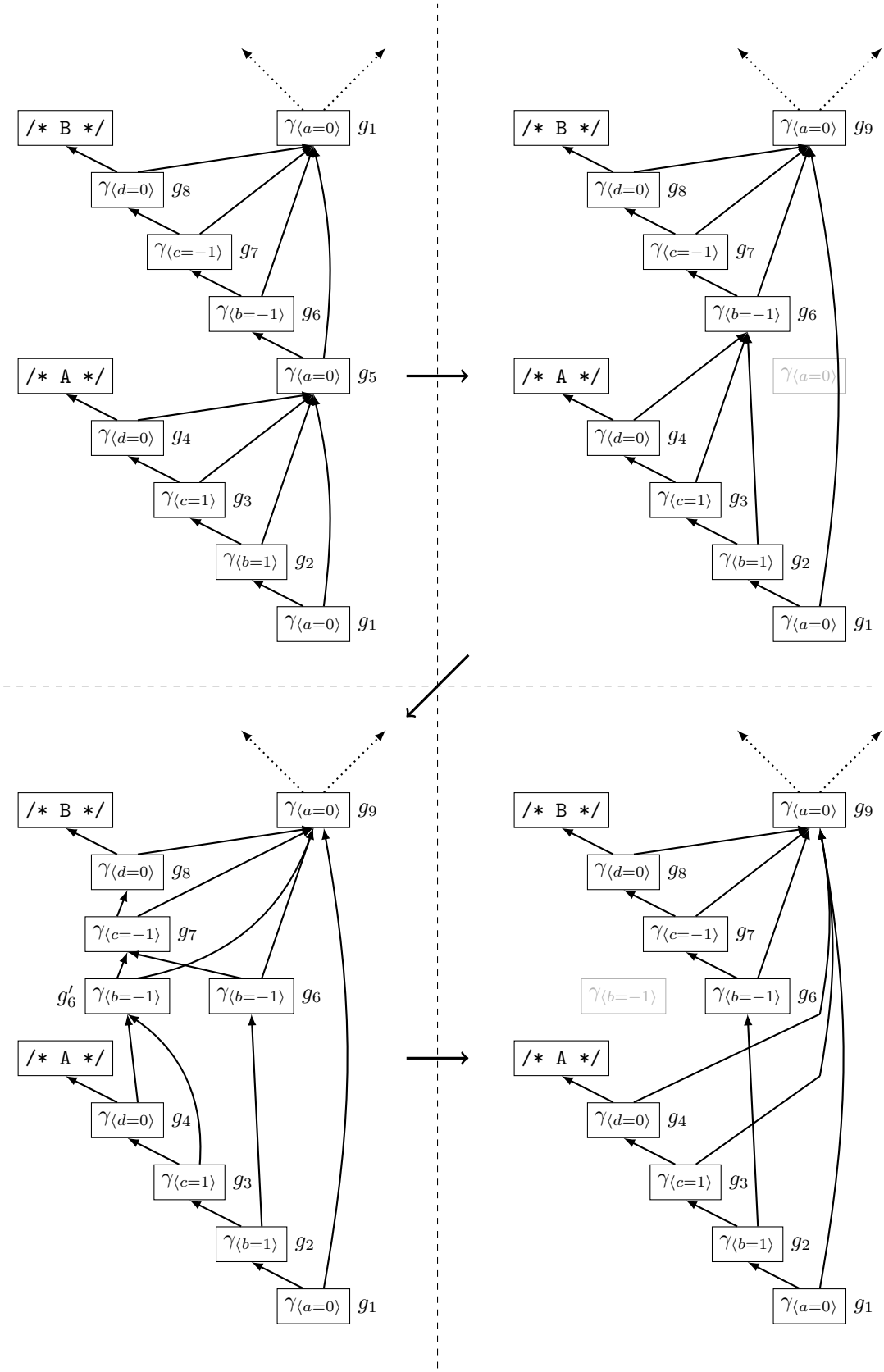


Abbildung 6.1.: Schematischer vFirm-Graph der Funktion `MTDecodeP` mit sukzessiver Anwendung der Elimination redundanter γ-Knoten. Der linke Vorgänger der g_i ist g_i`.true`.

Ausrollen von Schleifen

Die Transformation Unroll demonstrieren wir an der Funktion `Cos` aus `SingleSource/Benchmarks/Stanford/Oscar.c`, die sich im Anhang in Listing C.2 befindet.

Die Schleife terminiert immer nach 9 Iterationen und ist klein genug zum Ausrollen. In jeder der ausgerollten Iterationen sind die Werte von `i` und `factor` bekannt, was insbesondere die Auswertung der beiden γ-Knoten, die die geschachtelte Verzweigung modellieren, erlaubt.

Anschließend erhalten wir einen Graphen, der keine Steuerflussänderungen mehr enthält.

6.3. Bewertung der Implementierung

Zum Abschluss betrachten wir, an welchen Stellen uns die Eigenschaften der vFirm-Darstellung die Implementierung der Optimierungsphase erleichtert haben.

Auswertung bei Bedarf In der vFirm-Darstellung lassen sich prinzipiell keine Codeverschiebungstransformationen durchführen, weil neben der Auswertung bei Bedarf keine Ordnung der Operationen spezifiziert ist.

Wir führen jedoch Transformationen durch, die den von der Anordnung unabhängigen Teil solcher Optimierungen entsprechen. Die Knotenverschmelzung kennzeichnet redundante Berechnungen im gesamten Graph; in Kombination mit der γ-Distribution finden wir sogar Berechnungen, die nicht auf allen Programmpfaden redundant sind.

Analog geben wir durch das Senken der Schleifentiefe von Knoten der Abbauphase den Hinweis für eine Platzierung der Berechnung außerhalb von Schleifen.

Die Umordnungstransformation hat durch die fehlenden Ordnungsbeschränkungen mehr Freiheiten bei der Erfassung und Rekonstruktion einer Summe; sie kann Argumente aus dem ganzen Graph enthalten und hat dadurch einen größeren Geltungsbereich.

Diese Trennung der Aspekte ist sinnvoll: Die Transformationen auf der Darstellung sind einfach, und die Platzierung der Operationen wird einmal an zentraler Stelle während der Abbauphase entschieden.

Steuerflusstransformation Die Möglichkeit, γ-Knoten ebenfalls durch Musterersetzungsregeln zu transformieren, macht die Implementierung der Steuerflussoptimierungen sehr einfach. Gegenüber traditionellen Zwischendarstellungen haben die Transformationen ausschließlich lokale Auswirkungen.

Schleifen-sensitive Optimierungen Die Darstellung von Schleifen durch die Gatingfunktionen θ und η entspricht einer Vorberechnung der Schleifenanalyse während des Aufbaus der vFirm-Darstellung. Dort können wir dann die Schleifentiefe effizient berechnen, womit wir bei Bedarf im Analyseteil der Optimierungen die Schleifenstruktur des Programms berücksichtigen können.

7. Zusammenfassung und Ausblick

7.1. Zusammenfassung

In dieser Arbeit wurde eine Optimierungsphase für die funktionale und referentiell transparente Zwischendarstellung vFirm entworfen, implementiert und evaluiert.

Zunächst wurde ein Kostenmodell hergeleitet, das berücksichtigt, wie wahrscheinlich und wie oft eine Operation ausgewertet wird, und dadurch die Argumentation über die Optimierung der Verzweigungs- und Schleifenstruktur erlaubt.

Damit konnte für bekannte und neue Formulierungen von traditionellen Optierungen gezeigt werden, dass sie bis auf wenige Ausnahmen die Kosten des Programms verbessern.

Die Optimierungsphase umfasst Transformationen, deren Anwendung vergleichbare Effekte wie die Durchführung von Konstantenfaltung, Elimination gemeinsamer Teilausdrücke, Elimination von totem und unerreichbarem Code, Jump Threading, Strength Reduction, Ausrollen von Schleifen und Umordnung von assoziativen und kommutativen Operationen hat.

Die Implementierung der Optimierungen profitiert von den spezifischen Eigenschaften der Darstellung und ist daher auf den ersten Blick einfach. Dennoch gibt es in manchen Formulierungen kritische Stellen bezüglich der Korrektheit oder der Kostenbilanz der Optimierung; diese Probleme wurden herausgearbeitet und gelöst.

Die Betrachtung der Laufzeit von Beispielprogrammen hat ergeben, dass eine Verringerung der Kosten eine Verbesserung der Ausführungsgeschwindigkeit bewirkt.

Anschließend wurde durch die Optimierung von über 6500 Funktionen aus praxisnahem Programmcode überprüft, dass die vorgestellten Verfahren Anwendung finden. Mit aktivierter Knotenduplikation bei der Elimination redundanter γ-Knoten konnte in über 1000 Funktionen eine Verbesserung der Kosten erzielt werden.

7.2. Ausblick

Die Programmoptimierung mit der vFirm-Darstellung ist erfolgsversprechend. In diesem Abschnitt folgen nun Ideen, wie die Entwicklung der Optimierungsphase fortgeführt werden könnte.

Erweiterung der vFirm-Abbauphase Das zentrale Problem der Darstellung sind die Beschränkungen in der Abbauphase, die die Codeerzeugung für eine Vielzahl von Programmen verhindern. Hier sollte versucht werden, Ansätze aus der Literatur wie beispielsweise in [Sta11] auf die Darstellung anzupassen. Gelingt die Codeerzeung für alle Eingabeprogramme, sollte die Abbauphase auf Möglichkeiten zur Optimierung untersucht werden, da die Entscheidungen, die dort getroffen werden, einen großen Einfluss auf die resultierende Laufzeit des übersetzten Programms haben. Auch die Untersuchung des Zusammenspiels von Transformationen der Zwischendarstellung und des Abbauverfahrens ist interessant.

Kombination der Elimination redundanter γ-Knoten mit einer Wertebereichsanalyse Bei der Berechnung der Pfadinformationen für die Elimination redundanter γ-Knoten werden von den Vergleichsoperationen im Programm Wertebereiche für bestimmte Knoten abgeleitet. Hier wäre es nun interessant, diese Information als Grundlage für eine Wertebereichsanalyse von anderen Knoten zu verwenden.

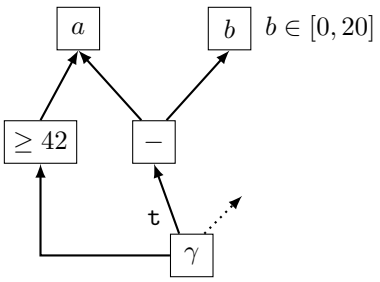


Abbildung 7.1.: Beispiel für die Kombination von Pfadinformations- und Wertebereichsanalyse.

Im Beispiel in Abbildung 7.1 könnte man dann unter Kenntnis von $b \in [0, 20]$ ableiten, dass $- \in [22, \mathbb{I}_{\max}]$ gilt, weil uns die Pfadinformation $a \in [42, \mathbb{I}_{\max}]$ liefert.

Von diesen pfadsensitiven Wertebereichen profitieren alle Analysen, die präzise Bereichsinformationen benötigen. Möglicherweise können auch weitere Vergleichsoperationen ausgewertet werden.

Erweiterungen der Umordnungstransformation Die Umordnungstransformation lässt sich leicht für Teilgraphen bestehend aus bitweisen Und- oder Oder-Operationen, jeweils mit Negationen, formulieren. Konstante Argumente werden wie bei Summen verrechnet; für variable Argumente muss man nur speichern, ob sie normal oder negiert vorkommen. Kommt ein Argument gleichzeitig normal und negiert vor, wird der konstante Anteil 0, falls es sich um Und-Operationen handelt; im Falle von Oder-Operation wird der konstante Anteil $-1 = 1 \ldots 1$.

Die Darstellung würde sich außerdem für eine Rekonstruktion eignen, die, wie in [CEK08] beschrieben, alle im Graph vorhandenen Summen betrachtet, um möglichst viele wiederverwendbare Teilsummen zu erhalten.

Neue Optimierungen Die Abstraktionen der vFirm-Darstellung machen traditionelle Optimierungen einfach und schaffen sozusagen neue Kapazitäten für die Durchführung von neuen, komplexeren Optimierungen.

In dieser Arbeit wurden Transformationen durch die Berücksichtigung der Schleifentiefe mächtiger. Es wäre nun interessant zu untersuchen, ob es weitere Programmeigenschaften gibt, mit deren Hilfe neue Kombinationen von Optimierungen entstehen, die zusammen einen größeren Effekt als die Einzeltransformationen haben.

Literaturverzeichnis

[Aho08] Aho, Alfred V. (Hrsg.): *Compiler : Prinzipien, Techniken und Werkzeuge*. 2., aktualisierte Aufl. München [u.a.] : Pearson Studium, 2008 (it Informatik). – ISBN 978–3–8273–7097–6

[BBZ11] Braun, Matthias ; Buchwald, Sebastian ; Zwinkau, Andreas: Firm—A Graph-Based Intermediate Representation / Karlsruhe Institute of Technology. Version: 2011. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000025470`. Karlsruhe, 2011 (35). – Forschungsbericht

[BC94] Briggs, Preston ; Cooper, Keith D.: Effective partial redundancy elimination. In: *SIGPLAN Not.* 29 (1994), June, 159–170. `http://dx.doi.org/http://doi.acm.org/10.1145/773473.178257`. – DOI http://doi.acm.org/10.1145/773473.178257. – ISSN 0362–1340

[BE96] Blume, William ; Eigenmann, Rudolf: Demand-driven, symbolic range propagation. Version: 1996. `http://dx.doi.org/10.1007/BFb0014197`. In: Huang, Chua-Huang (Hrsg.) ; Sadayappan, Ponnuswamy (Hrsg.) ; Banerjee, Utpal (Hrsg.) ; Gelernter, David (Hrsg.) ; Nicolau, Alex (Hrsg.) ; Padua, David (Hrsg.): *Languages and Compilers for Parallel Computing* Bd. 1033. Springer Berlin / Heidelberg, 1996. – ISBN 978–3–540–60765–6, 141-160. – 10.1007/BFb0014197

[CEK08] Cooper, Keith ; Eckhardt, Jason ; Kennedy, Ken: Redundancy elimination revisited. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA : ACM, 2008 (PACT '08). – ISBN 978–1–60558–282–5, 12–21

[Cor05] Cormen, Thomas H. (Hrsg.): *Introduction to algorithms*. 2nd.ed. Cambridge, Mass. [u.a.] : MIT Press [u.a.], 2005. – ISBN 0–262–03293–7

[CSV01] Cooper, Keith D. ; Simpson, L. T. ; Vick, Christopher A.: Operator strength reduction. In: *ACM Trans. Program. Lang. Syst.* 23 (2001), September, 603–625. `http://dx.doi.org/http://doi.acm.org/10.1145/504709.504710`. – DOI http://doi.acm.org/10.1145/504709.504710. – ISSN 0164–0925

[Dav08] Davey, Hilary A. Brian A. ; Priestley P. Brian A. ; Priestley: *Introduction to lattices and order*. 2. ed., 4. print. Cambridge [u.a.] : Cambridge Univ. Press, 2008. – ISBN 978–0–521–78451–1

[Dei10] Deiser, Oliver: *Einführung in die Mengenlehre*. Springer Berlin Heidelberg, 2010 (Springer-Lehrbuch). – ISBN 978–3–642–01445–1

[Har77] Harrison, W.H.: Compiler Analysis of the Value Ranges for Variables. In: *Software Engineering, IEEE Transactions on* SE-3 (1977), may, Nr. 3, S. 243 – 250. `http://dx.doi.org/10.1109/TSE.1977.231133`. – DOI 10.1109/TSE.1977.231133. – ISSN 0098–5589

[Joh04] Johnson, N.: *Code Size Optimization for Embedded Processors*, University of Cambridge, Diss., 2004

[KWB+05] Kreahling, William C. ; Whalley, David ; Bailey, Mark W. ; Yuan, Xin ; Uh, Gang-Ryung ; Engelen, Robert van: Branch elimination by condition merging. In: *Software: Practice and Experience* 35 (2005), Nr. 1, 51–74. `http://dx.doi.org/10.1002/spe.627`. – DOI 10.1002/spe.627. – ISSN 1097–024X

[Law07] Lawrence, Alan C.: Optimizing compilation with the Value State Dependence Graph / University of Cambridge, Computer Laboratory. 2007 (705). – Forschungsbericht

[Lie11] Liebe, Olaf: *Eine funktionale, vollständige und referentiell transparente Zwischendarstellung für Übersetzer*, Karlsruher Institut für Technlogie, Diplomarbeit, 2011

[Lin02] Lindenmaier, Götz: libFIRM – A Library for Compiler Optimization Research Implementing FIRM / Universität Karlsruhe, Fakultät für Informatik. Version: September 2002. `http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps`. Universität Karlsruhe, Fakultät für Informatik, September 2002 (2002-5). – Forschungsbericht. – 75 S.

[llv] *LLVM Testumgebung*. `http://llvm.org/docs/TestingGuide.html`

[Mor98] Morgan, Robert: *Building an optimizing compiler*. Boston [u.a.] : Butterworth-Heinemann, 1998. – ISBN 1–55558–179–X

[Muc97] Muchnick, Steven S.: *Advanced compiler design and implementation*. San Francisco, Calif. : Morgan Kaufmann, 1997. – ISBN 1–55860–320–4

[MW95] Mueller, Frank ; Whalley, David B.: Avoiding conditional branches by code replication. In: *SIGPLAN Not.* 30 (1995), June, 56–66. `http://dx.doi.org/http://doi.acm.org/10.1145/223428.207116`. – DOI http://doi.acm.org/10.1145/223428.207116. – ISSN 0362–1340

[NNH05] Nielson, Flemming ; Nielson, Hanne R. ; Hankin, Chris: *Principles of program analysis : with 51 tables*. Corr. 2. print. Berlin : Springer, 2005. – ISBN 3–540–65410–0

[spe] *SPEC CINT2000 Benchmarks*. `http://www.spec.org/cpu2000/CINT2000/`

[Sta11] Stanier, James: *Removing and Restoring Control Flow with the Value State Dependence Graph*, School of Informatics University of Sussex, Diss., 2011

[TG08] Thakur, Aditya ; Govindarajan, R.: Comprehensive path-sensitive data-flow analysis. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA : ACM, 2008 (CGO '08). – ISBN 978–1–59593–978–4, 55–63

[TLB99] Trapp, M. ; Lindenmaier, G. ; Boesler, B.: Documentation of the Intermediate Representation Firm / Fakultät für Informatik, Universität Karlsruhe. 1999 (1999-14). – Forschungsbericht

[Tra01] Trapp, Martin: *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.*, University of Karlsruhe, Faculty of Informatik, Diss., Oct. 2001

[TSTL09] Tate, Ross ; Stepp, Michael ; Tatlock, Zachary ; Lerner, Sorin: Equality saturation: a new approach to optimization. In: *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–379–2, S. 264–276

[Upt06] Upton, E.: *Compiling with Data Dependence Graphs*, University of Cambridge, Diss., 2006

[WCES94] Weise, Daniel ; Crew, Roger F. ; Ernst, Michael ; Steensgaard, Bjarne: Value dependence graphs: representation without taxation. In: *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1994. – ISBN 0–89791–636–0, S. 297–310

A. Weitere Auswertungsergebnisse

Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
besser	1005	400	253	189	151	115	101	101	79
unverändert	4588								
schlechter	23	4	2	0					

(a) Veränderung der Kostenfunktionen der Graphen

Transformation	Anwendungen in...			Transformation	Anwendungen in...		
	Graphen	proz.	absolut		Graphen	proz.	absolut
ConstCond	555	9,9 %	5656	ERG-Eval	374	6,7 %	12421
SameArg	211	3,8 %	5310	ERG-Dup	333	5,9 %	14857
ModeBGamma	108	1,9 %	122	ERG-Implied	170	3,0 %	3470
NegCond	8	0,1 %	18	ERG-Constified	41	0,7 %	70
GammaOnCond	54	1,0 %	186	Unroll	133	2,4 %	237
GammaCF	203	3,6 %	622	UnrollConst	1	0,0 %	1
GammaCSE	76	1,4 %	160	UnrollReassoc	0	0,0 %	0
GammaLoop	66	1,2 %	197	StrengthRed	298	5,3 %	694
Reassoc	57	1,0 %	102	LFTR	253	4,5 %	573

(b) Anwendungen der Transformationen

Tabelle A.1.: Ergbnisse für Knotenduplikation mit maximaler Distanz 0.

Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
besser	1029	419	256	189	151	115	101	101	79
unverändert	4560								
schlechter	27	3	2	0					

(a) Veränderung der Kostenfunktionen der Graphen

Transformation	Anwendungen in...			Transformation	Anwendungen in...		
	Graphen	proz.	absolut		Graphen	proz.	absolut
ConstCond	586	10,4 %	6608	ERG-Eval	410	7,3 %	13340
SameArg	220	3,9 %	5358	ERG-Dup	373	6,6 %	16479
ModeBGamma	108	1,9 %	122	ERG-Implied	177	3,2 %	3579
NegCond	8	0,1 %	18	ERG-Constified	43	0,8 %	72
GammaOnCond	56	1,0 %	190	Unroll	133	2,4 %	239
GammaCF	205	3,7 %	626	UnrollConst	1	0,0 %	1
GammaCSE	78	1,4 %	166	UnrollReassoc	0	0,0 %	0
GammaLoop	67	1,2 %	198	StrengthRed	298	5,3 %	692
Reassoc	57	1,0 %	102	LFTR	253	4,5 %	571

(b) Anwendungen der Transformationen

Tabelle A.2.: Ergbnisse für Knotenduplikation mit maximaler Distanz 1.

Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
besser	1115	447	267	198	158	118	103	103	79
unverändert	4458								
schlechter	43	6	4	1					

(a) Veränderung der Kostenfunktionen der Graphen

Transformation	Anwendungen in...			Transformation	Anwendungen in...		
	Graphen	proz.	absolut		Graphen	proz.	absolut
ConstCond	707	12,6 %	13018	ERG-Eval	539	9,6 %	19566
SameArg	244	4,3 %	6203	ERG-Dup	513	9,1 %	29031
ModeBGamma	110	2,0 %	132	ERG-Implied	193	3,4 %	4752
NegCond	8	0,1 %	18	ERG-Constified	73	1,3 %	250
GammaOnCond	62	1,1 %	365	Unroll	133	2,4 %	240
GammaCF	210	3,7 %	791	UnrollConst	1	0,0 %	1
GammaCSE	88	1,6 %	182	UnrollReassoc	0	0,0 %	0
GammaLoop	69	1,2 %	201	StrengthRed	298	5,3 %	692
Reassoc	57	1,0 %	102	LFTR	253	4,5 %	573

(b) Anwendungen der Transformationen

Tabelle A.3.: Ergbnisse für Knotenduplikation mit maximaler Distanz 3.

Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
besser	1124	451	274	201	158	118	103	103	79
unverändert	4442								
schlechter	50	8	7	1					

(a) Veränderung der Kostenfunktionen der Graphen

Transformation	Anwendungen in...			Transformation	Anwendungen in...		
	Graphen	proz.	absolut		Graphen	proz.	absolut
ConstCond	723	12,9 %	25004	ERG-Eval	565	10,1 %	38370
SameArg	249	4,4 %	9577	ERG-Dup	534	9,5 %	65954
ModeBGamma	110	2,0 %	155	ERG-Implied	200	3,6 %	7296
NegCond	8	0,1 %	18	ERG-Constified	73	1,3 %	1810
GammaOnCond	69	1,2 %	949	Unroll	133	2,4 %	240
GammaCF	215	3,8 %	1448	UnrollConst	1	0,0 %	1
GammaCSE	98	1,7 %	247	UnrollReassoc	0	0,0 %	0
GammaLoop	70	1,2 %	203	StrengthRed	298	5,3 %	692
Reassoc	57	1,0 %	102	LFTR	253	4,5 %	573

(b) Anwendungen der Transformationen

Tabelle A.4.: Ergbnisse für Knotenduplikation mit maximaler Distanz 5.

Δ	≥ 0%	> 5%	> 10%	> 15%	> 20%	> 25%	> 30%	> 40%	> 50%
besser	748	231	76	36	19	4	4	4	2
unverändert	4847								
schlechter	21	5	3	1					

(a) Veränderung der Kostenfunktionen der Graphen

Transformation	Anwendungen in...			Transformation	Anwendungen in...		
	Graphen	proz.	absolut		Graphen	proz.	absolut
ConstCond	172	3,1 %	378	ERG-Eval	107	1,9 %	180
SameArg	3	0,1 %	5	ERG-Dup	0	0,0 %	0
ModeBGamma	74	1,3 %	87	ERG-Implied	88	1,6 %	215
NegCond	8	0,1 %	18	ERG-Constified	40	0,7 %	66
GammaOnCond	54	1,0 %	186	Unroll	0	0,0 %	0
GammaCF	202	3,6 %	621	UnrollConst	0	0,0 %	0
GammaCSE	76	1,4 %	160	UnrollReassoc	0	0,0 %	0
GammaLoop	66	1,2 %	197	StrengthRed	355	6,3 %	821
Reassoc	57	1,0 %	102	LFTR	310	5,5 %	700

(b) Anwendungen der Transformationen

Tabelle A.5.: Ergbnisse mit deaktivierter Knotenduplikation und Ausroll-Transformation.

B. Quelltexte der Testprogramme

```
1 int eval1(int a, int b, int c, int d) {
2   return 10+a+b+c+d+20+d+c+b+a+30;
3 }
4
5 int main(int argc, char** argv) {
6   int N = atoi(argv[1]);
7   int a,b,c,d, X=0;
8   for (a = 0; a < N; a++)
9     for (b = 0; b < N; b++)
10      for (c = 0; c < N; c++)
11        for (d = 0; d < N; d++)
12          X += eval1(a,b,c,d);
13
14  printf("N␣=␣%d,␣X␣=␣%d\n", N, X);
15  return 0;
16 }
```

Listing B.1: Testprogramm zur Knotenreduktionen

```
1 int eval2(int a, int b, int c, int d) {
2   int x;
3   if (a > 0 && b > 0 && c == 1)
4     x = 10 + (d ? 5 : a+b);
5   else if (a > 0 && b <= 0 && c == 0)
6     x = a + (d ? 3 : b);
7   else if (a <= 0 && b == 0 && c < 0)
8     x = a+b;
9   else
10    x = 0;
11  return x;
12 }
13
14 int main(int argc, char** argv) {
15  int N = atoi(argv[1]);
16  int a,b,c,d, X=0;
17  for (a = 0; a < N; a++)
18    for (b = 0; b < N; b++)
19      for (c = 0; c < N; c++)
20        for (d = 0; d < N; d++)
21          X += eval2(a,b,c,d&1);
22
23  printf("N␣=␣%d,␣X␣=␣%d\n", N, X);
24  return 0;
25 }
```

Listing B.2: Testprogramm zur Steuerflussoptimierung

```
1 int eval3(int a, int b, int c, int d) {
2   int x = 0, i;
```

```
3   for (i = 0; i < 42; i+=3) {
4     x += a+i;
5     x += b+i;
6     x += c+i;
7     x += d+i;
8   }
9   return x;
10 }
11
12 int main(int argc, char** argv)
13 // wie eval_1
```

Listing B.3: Testprogramm zur Schleifenoptimierung

```
1 int eval4(int a, int b, int c, int d) {
2   int x = 0, i;
3   for (i = 0; i < 20; i++) {
4     x += b * ((i > 1 ? 1 : c) ? d*i : -a);
5   }
6   return x;
7 }
8
9 int main(int argc, char** argv)
10 // wie eval_1
```

Listing B.4: Testprogramm zur Steuerfluss- und Schleifenoptimierung

C. Quelltexte der Fallstudien

```
1  Bool MTDecodeP(MT *t,char **s) {
2    ...
3    if(a == 0 && b == 1 && c == 1 && d == 0) { /* A */
4      MTMX(t); MTRotate(t,0,-1); MTTranslate(t,tx,ty);
5      if(tx != 0 || ty != 0) sprintf(cif,"MX␣R␣0␣-1␣T␣%d␣%d",tx,ty);
6      else                   sprintf(cif,"MX␣R␣0␣-1");
7    } else if(a == 0 && b == -1 && c == -1 && d == 0) { /* B */
8      MTMX(t); MTRotate(t,0,1); MTTranslate(t,tx,ty);
9      if(tx != 0 || ty != 0) sprintf(cif,"MX␣R␣0␣1␣T␣%d␣%d",tx,ty);
10     else                   sprintf(cif,"MX␣R␣0␣1");
11   } else if(a == 0 && b == 1 && c == -1 && d == 0) { /* C */
12     MTRotate(t,0,-1); MTTranslate(t,tx,ty);
13     if(tx != 0 || ty != 0) sprintf(cif,"R␣0␣-1␣T␣%d␣%d",tx,ty);
14     else                   sprintf(cif,"R␣0␣-1");
15   } else if(a == 0 && b == -1 && c == 1 && d == 0) { /* D */
16     MTRotate(t,0,1); MTTranslate(t,tx,ty);
17     if(tx != 0 || ty != 0) sprintf(cif,"R␣0␣1␣T␣%d␣%d",tx,ty);
18     else                   sprintf(cif,"R␣0␣1");
19   } else if(a == 1 && b == 0 && c == 0 && d == 1) { /* E */
20     MTTranslate(t,tx,ty);
21     if(tx != 0 || ty != 0) sprintf(cif,"T␣%d␣%d",tx,ty);
22     else                   cif[0] = EOS;
23   } else if(a == -1 && b == 0 && c == 0 && d == -1) { /* F */
24     MTRotate(t,-1,0); MTTranslate(t,tx,ty);
25     if(tx != 0 || ty != 0) sprintf(cif,"R␣-1␣0␣T␣%d␣%d",tx,ty);
26     else                   sprintf(cif,"R␣-1␣0");
27   } else if(a == -1 && b == 0 && c == 0 && d == 1) { /* G */
28     MTMX(t); MTTranslate(t,tx,ty);
29     if(tx != 0 || ty != 0) sprintf(cif,"MX␣T␣%d␣%d",tx,ty);
30     else                   sprintf(cif,"MX"); }
31   else if(a == 1 && b == 0 && c == 0 && d == -1) { /* H */
32     MTMY(t); MTTranslate(t,tx,ty);
33     if(tx != 0 || ty != 0) sprintf(cif,"MY␣T␣%d␣%d",tx,ty);
34     else                   sprintf(cif,"MY");
35   }
36   ...
37 }
```

Listing C.1: Funktion aus `MultiSource/Benchmarks/Prolangs-C/TimberWolfMC/mt.c` der LLVM-Testsuite (umformatiert und gekürzt).

```
1  float Cos(float x) {
2    /* computes cos of x (x in radians) by an expansion */
3    int i, factor;
4    float result, power;
5
6    result = 1.0f;
7    factor = 1;
8    power = x;
9    for (i = 2; i <= 10; i++) {
10     factor = factor * i;
11     power = power * x;
12     if ((i & 1) == 0) {
13       if ((i & 3) == 0)
14         result = result + power / factor;
15       else
16         result = result - power / factor;
17     }
18   }
19   return (result);
20 }
```

Listing C.2: Funktion aus `SingleSource/Benchmarks/Stanford/Oscar.c` der LLVM-Testsuite (umformatiert).

Danksagung

Ich danke allen Mitarbeitern des Lehrstuhls Programmierparadigmen, die für eine herzliche und entspannte Atmosphäre gesorgt und die Erstellung der vorliegenden Arbeit mit kleinen Ablenkungen erleichtert haben. Insbesondere gilt mein Dank Matthias Braun für die Betreuung der Arbeit und Sebastian Buchwald für seine Hilfe bei der Reparatur der vFirm-Implementierung und für seine Bereitschaft, sich für Probleme jeglicher Art Zeit zur Diskussion zu nehmen.
Ich danke meinen Eltern, dass sie mir dieses Studium ermöglicht und mich bei allen Entscheidungen unterstützt haben, und Melanie, die die Höhen und Tiefen bei der Erstellung der Arbeit mit mir durchgestanden hat.