

LEHRSTUHL FÜR INFORMATIK II
RHEINISCH-WESTFÄLISCHE TECHNISCHE HOCHSCHULE AACHEN

ANALYSE UND OPTIMIERUNG LINEAREN CODES

Diplomarbeit

von

Stefan Rieger

(post@srieger.com)

Aachen,
3. August 2005

Ausgeführt unter der Leitung von

Prof. Dr. Klaus Indermark
Lehrstuhl für Informatik II
RWTH Aachen

Dr. Thomas Noll
Lehrstuhl für Informatik II
RWTH Aachen

Ich möchte allen herzlich danken, die zum Gelingen dieser Arbeit beigetragen haben.

Insbesondere danke ich Herrn Prof. Dr. Klaus Indermark für die Ermöglichung, Förderung und abschließende Begutachtung meiner Arbeit sowie Herrn Dr. Thomas Noll für seine sehr kompetente Betreuung.

Inhaltsverzeichnis

0	Einführung	3
0.1	Codeoptimierung in Compilern	3
0.2	Aufbau dieser Arbeit	4
1	Grundlagen	7
1.1	Aufbau von LC-Programmen	7
1.1.1	Syntax	7
1.1.2	Semantik	9
1.2	Eigenschaften von LC-Programmen	11
1.2.1	Termdarstellung von LC-Programmen	11
1.2.2	Äquivalenz	13
1.2.3	Kostenmaße	15
1.2.4	Optimalität	16
2	Optimierungsverfahren für LC-Programme	19
2.1	Klassische Optimierungsverfahren	19
2.1.1	Dead Code Elimination	20
2.1.2	Common Subexpression Elimination	23
2.1.3	Konstantenfaltung	25
2.1.4	Propagation von Kopien	28
2.2	DAG-Optimierung	31
2.2.1	DAG eines LC-Programms	32
2.2.2	Codegenerierung aus einem DAG	36
2.2.3	Korrektheit der DAG-Optimierung	40
2.2.4	Idempotenz der DAG-Optimierung	46
2.3	Sonstige Optimierungsverfahren	49
2.3.1	Speicherbedarfsreduzierung	50
2.3.2	Algebraische Transformationen	51
2.3.2.1	Operationsspezifische Gleichungen	51
2.3.2.2	Reduktion der Operationsstärke	52
3	Vergleich der Optimierungsverfahren	53
3.1	Optimalität der DAG-Optimierung	53
3.2	Beziehungen zwischen den klassischen Transformationen	55
3.2.1	Verbesserungsschritte für die Dead Code Elimination	56
3.2.2	Verbesserungsschritte für die Common Subexpression Elimination	58
3.2.3	Verbesserungsschritte für die Propagation von Kopien	59
3.2.4	Verbesserungsschritte für die Konstantenfaltung	60
3.2.5	Wahl der Reihenfolge für die Gesamttransformation	60
3.3	Simulation des DAG-Verfahrens durch Einzeltransformationen	64

3.3.1	Mängel der Gesamttransformation	65
3.3.1.1	Variablensubstitution als Programmtransformation	65
3.3.1.2	Nicht entfernbare Kopieranweisungen	66
3.3.1.3	Rückwärtige Kopierpropagation	67
3.3.1.4	Schwachstelle: Propagation von Kopien	71
3.3.2	Erweiterte Gesamttransformation	72
3.3.3	Äquivalenz von DAG-Verfahren und erweiterter Gesamttransformation	74
3.4	Fazit	79
4	Implementierung der Verfahren	81
4.1	Verarbeitung der Eingabe	82
4.1.1	Lexikalische Analyse mit FLEX	82
4.1.2	Syntaktische Analyse	83
4.1.3	Interne Darstellung	85
4.2	Implementierung der Optimierungsverfahren	87
4.2.1	Datenstrukturen für die einfachen Verfahren	88
4.2.2	Implementierung der einfachen Programmtransformationen	88
4.2.2.1	Umsetzung der Dead Code Elimination	88
4.2.2.2	Umsetzung der Common Subexpression Elimination	89
4.2.2.3	Umsetzung der Konstantenfaltung	89
4.2.2.4	Umsetzung der Propagation von Kopien	90
4.2.2.5	Umsetzung der rückwärtigen Kopierpropagation	90
4.2.2.6	Umsetzung der SSA-Transformation	91
4.2.3	Datenstrukturen für die DAG-Optimierung	92
4.2.4	Umsetzung der DAG-Optimierung	93
4.2.4.1	DAG-Konstruktion	94
4.2.4.2	DAG-Codegenerierung	95
	Zusammenfassung	97
	Literaturverzeichnis	99

0 Einführung

Heutzutage sind informationsverarbeitende Systeme allgegenwärtig. Die Effizienz und damit Wirtschaftlichkeit dieser Systeme wird zu einem großen Teil durch die auf ihnen laufende Software bestimmt. Trotz hoher Leistungssteigerungen der Hardware gibt es immer noch viele Anwendungen, für die eine (Laufzeit-)effiziente Implementierung von besonderer Wichtigkeit ist.

Das Ziel ist eine möglichst gute Ausnutzung der Hardware, die nicht durch eine schlecht optimierte Anwendung ausgebremst werden sollte. Um dies zu erreichen kommen mehrere Lösungen in Frage, die allerdings verschiedene Vor- und Nachteile haben.

Der effizienteste Ansatz ist sichererlich ein handoptimierter Code, der womöglich noch (teilweise) in Assemblersprache verfaßt ist. Dadurch hat der Programmierer die Möglichkeit, größtmöglichen Einfluß auf den Ressourceneinsatz zu nehmen. Jedoch kommt diese Methode heute eigentlich nur noch für Anwendungen kleineren Umfangs in Frage, für die die Effizienz von entscheidender Bedeutung ist oder die Ressourcen des eingesetzten Systems stark beschränkt sind (z.B. bei eingebetteten Systemen). Ansonsten wiegt die Laufzeiteinsparung die zusätzlich notwendige Arbeitszeit und die dadurch entstehenden Kosten nicht auf.

Für größere Projekte ist daher der Einsatz von höheren Programmiersprachen unabdingbar, und eine Optimierung „von Hand“ kommt in den meisten Fällen nicht mehr in Frage. Also ist es wichtig, daß der entsprechende Compiler eine automatische Optimierung durchführt, um nicht zuviel Leistung zu verschenken. Die Qualität einer automatischen Optimierung kann natürlich nicht an die einer manuellen Optimierung heranreichen, was an der Unentscheidbarkeit vieler programmspezifischer Probleme liegt.

Wenn wir von Optimierung von Programmen sprechen, ist daher oft nur eine Programm*verbesserung* gemeint, da eine wirkliche Optimierung, d.h. die Erzeugung eines kostenminimalen Programmes für ein bestimmtes Problem fast immer unentscheidbar und ansonsten¹ komplexitätsmäßig nicht durchführbar ist.

0.1 Codeoptimierung in Compilern

Es bietet sich an, einen Teil der Optimierung in einem Compiler zunächst auf einem zielsystemunabhängigen Zwischencode vor der eigentlichen Codegenerierung durchzuführen.

In Abbildung 0.1 ist die modulare Struktur eines Compilers grob dargestellt. Die hier näher betrachtete Optimierung des Zwischencodes ist dort durch Fettdruck hervorgehoben. Da der Zwischencode maschinenunabhängig ist, gehört dessen Generierung und Optimierung noch zum *Frontend* und muß daher nicht für jede Implementierungsplattform neu programmiert werden.

Nach der Maschinencode-Generierung ist u.U. noch eine auf die Zielmaschine zugeschnittene Optimierung durchzuführen, die z.B. die Registerzuordnung optimiert und evtl. vorhandene spezielle Funktionen oder Befehle ausnutzt, um die Laufzeit zusätzlich zu verbessern.

¹Entscheidbarkeit ist nur für sehr einfache Programmklassen gegeben; siehe Abschnitte 1.2.2 und 1.2.4.

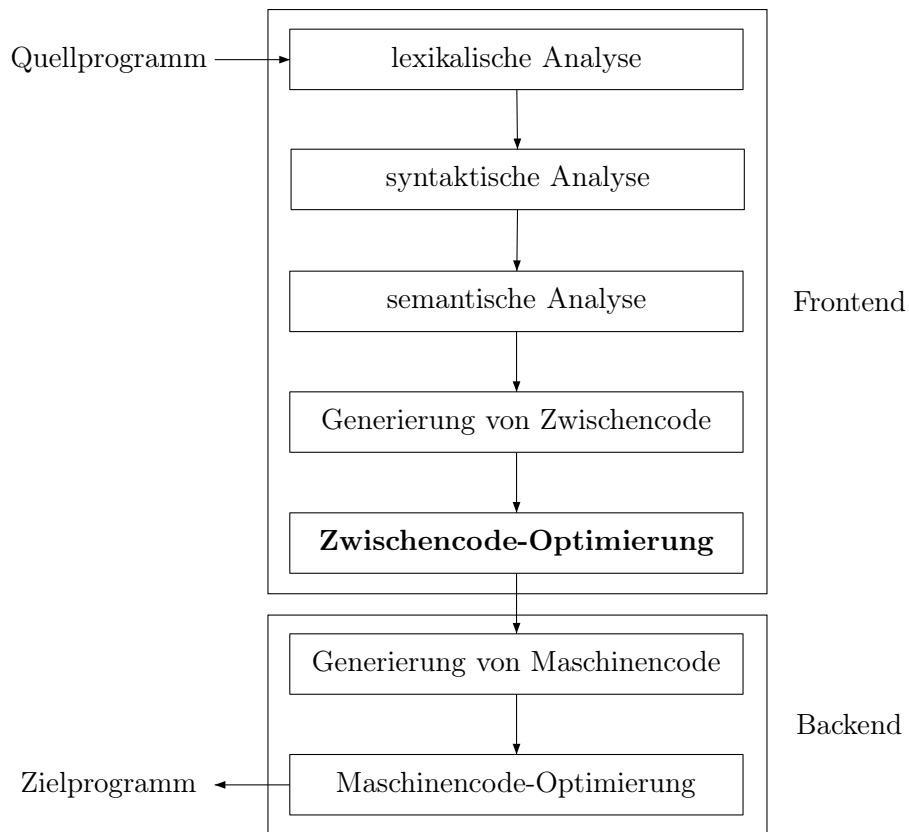


Abbildung 0.1: Aufbau eines Compilers (Quelle: [1])

In dieser Arbeit werden wir eine beschränkte Klasse von Zwischencode betrachten, den sogenannten *linearen Code* (LC). Dieser stellt eine Folge von Berechnungsbefehlen ohne Verzweigungen oder Sprünge dar, und findet sich in Programmen in *Grundblöcken*², also unter anderem in Schleifenrumpfen, die aufgrund der Iteration einen großen Einfluß auf die Gesamtlaufzeit haben. Es gilt laut [3] die Faustregel, daß die meisten Programme 90% ihrer Ausführungszeit in nur 10% ihres Codes verbringen. Ein aus linearem Code bestehendes Programm sei im folgenden als *LC-Programm* bezeichnet.

Da es sich um Zwischencode handelt, der noch in Code für die Zielmaschine zu übersetzen ist, wird von einer unbegrenzten Verfügbarkeit von Registern ausgegangen. Dies macht die Darstellung deutlich intuitiver, vereinfacht aber auch einige sonst schwere Probleme (siehe Abschnitt 2.2.2).

0.2 Aufbau dieser Arbeit

Nach dieser Einführung werden in Kapitel 1 zunächst einige Grundlagen vermittelt. Dabei geht es um die (formale) Syntax und Semantik von LC-Programmen (Abschnitt 1.1) und später in Abschnitt 1.2 um deren Eigenschaften. Darunter fallen etwa Kostenmaße, Äquivalenz und Optimalität, die es genauer zu untersuchen gilt.

In Kapitel 2 werden dann die einzelnen Optimierungsverfahren vorgestellt, wobei im wesentlichen

²Codesequenz ohne Verzweigung oder Sprünge, siehe auch [2].

zwischen den klassischen Einzeltransformationen (Abschnitt 2.1) und der graphbasierten DAG-Optimierung (Abschnitt 2.2) zu unterscheiden ist. Die jeweiligen Eigenschaften der Verfahren (wie z.B die Korrektheit) werden im Detail beleuchtet. Abschließend wird in Abschnitt 2.3 noch kurz auf weitere Optimierungsmethoden für LC-Programme eingegangen.

Kapitel 3 dient der Untersuchung der Beziehungen der in Kapitel 2 vorgestellten Verfahren untereinander. Die zentrale Frage dabei ist, ob die graphbasierte DAG-Optimierung durch eine Kombination der anderen klassischen Methoden substituieren läßt, d.h. ob der Ergebniscode gleich effizient ist. Die Untersuchung dieses Problems ist das Schwerpunktthema dieser Arbeit.

Im letzten Kapitel wird dann die im Rahmen dieser Diplomarbeit erstellte Implementierung der Optimierungsverfahren erläutert. Dabei wird sowohl auf die Umsetzung der einzelnen Verfahren (Abschnitt 4.2) als auch kurz auf notwendige Compilerbau-technische Grundlagen eingegangen (Abschnitt 4.1).

1 Grundlagen

Nachdem in der Einführung schon textuell erläutert wurde, wobei es sich bei LC-Programmen handelt, wollen wir in diesem Kapitel insbesondere auf die formalen Gesichtspunkte eingehen.

1.1 Aufbau von LC-Programmen

Ein LC-Programm besteht aus einem Vektor von Eingabevariablen, der vor der Ausführung des ersten Befehls mit den Eingabewerten initialisiert wird, einem Vektor von Ausgabevariablen, dessen Einträge allein als „Lösung“ von Interesse sind und einer Folge von Anweisungen in der Form von Wertzuweisungen. Wie bereits erwähnt, sind Sprünge oder Verzweigungen nicht erlaubt.

1.1.1 Syntax

Zur Beschreibung der Syntax von LC-Programmen benötigen wir zunächst einige formale Definitionen.

Definition 1.1.1 (Signatur): Eine Signatur $\Sigma = (F, C)$ ist ein Paar, bestehend aus

1. einer endlichen Menge von *Funktionssymbolen* (oder: *Operationssymbole*) $F := \bigcup_{i=1}^{\infty} F^{(i)}$, wobei $F^{(i)}$ die i -stelligen Funktionssymbole sind und
2. einer (nicht notwendigerweise endlichen) Menge von *Konstantensymbolen* C^1 .

Außerdem bezeichne im folgenden $V := \{x, y, z, \dots\}$ die (unendliche) Menge aller Variablen.

Beispiel 1.1.2: Ein Beispiel für eine Signatur ist $\Sigma_{\text{arithm}} = (\{+, *, -\}, \mathbb{Z})$ mit $+, * \in F^{(2)}$ und $- \in F^{(1)}$. Eine vereinfachte Notation dafür ist $\Sigma_{\text{arithm}} = (\{+^{(2)}, *^{(2)}, -^{(1)}\}, \mathbb{Z})$, bei der für jeden Operator die Stelligkeit direkt angegeben wird. Es ist zu beachten, daß \mathbb{Z} in diesem Fall nur *Konstantensymbole* bereitstellt, deren Bedeutung von der gewohnten abweichen kann (siehe dazu auch Abschnitt 1.1.2).

Definition 1.1.3 (LC-Programm): Ein LC-Programm ist ein Quadrupel $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit

- der Signatur $\Sigma = (F, C)$,
- dem Vektor paarweise verschiedener *Eingabevariablen* $\vec{v}_{in} = (x_1, \dots, x_s)$, $x_i \in V$ und
- dem Vektor paarweise verschiedener *Ausgabevariablen* $\vec{v}_{out} = (y_1, \dots, y_t)$, $y_i \in V$.
- $\beta = \alpha_1; \alpha_2; \dots; \alpha_n$ heißt *Block* und die α_i sind *Anweisungen* der Form $x \leftarrow e$ mit $x \in V$ und $e \in V \cup C \cup \{f(u_1, \dots, u_r) \mid f \in F^{(r)} \text{ und } \forall j \in \{1, \dots, r\} : u_j \in V \cup C\}$.

Es muß ferner gelten: Keine Variable darf direkt oder auf Umwegen über andere Variablen (mittels Anweisungen der Form $x \leftarrow y$, $x, y \in V$) sich selbst zugewiesen werden.

- Zu \vec{v}_{in} bzw. \vec{v}_{out} bezeichne $V_{in} := \{x_1, \dots, x_s\}$ bzw. $V_{out} := \{y_1, \dots, y_t\}$ die *Mengen* der Ein-/Ausgabevariablen.

¹Im Programm können dann natürlich nur endlich viele auftreten.

$$\begin{aligned}
\vec{v}_{in} &: (x, y) \\
\beta &: u \leftarrow 3; \\
&v \leftarrow x - y; \\
&w \leftarrow u + 1; \\
&x \leftarrow x - y; \\
&u \leftarrow x - y; \\
&z \leftarrow u * w; \\
&u \leftarrow 2 * u; \\
&v \leftarrow w - 1; \\
\vec{v}_{out} &: (u, v)
\end{aligned}$$

Abbildung 1.1: LC-Programm $\pi = (\Sigma_{arithm'}, \vec{v}_{in}, \vec{v}_{out}, \beta)$ (Quelle: Üb. zu [2])

- Es muß $V_{in} \cap V_{out} = \emptyset$ gelten² (keine Variable ist sowohl Eingabe- als auch Ausgabevariable).

Zusätzlich werden zu π die folgenden Bezeichnungen eingeführt:

- C_π sei die Menge der *tatsächlich* in π vorkommenden Konstantensymbole,
- V_π die Menge der im Programm π vorkommenden Variablen,
- V_α die Menge der in der Anweisung α vorkommenden Variablen,
- V_e die Menge der im Rechenausdruck e vorkommenden Variablen

Außerdem bezeichne \mathcal{LC} die Menge *aller* LC-Programme.

Anweisungen sind also einfache Wertzuweisungen an Variablen, deren rechte Seiten entweder nur aus Variablen/Konstanten oder aus der Applikation eines Operationssymbols auf Variablen/Konstanten bestehen. Es sind keine verschachtelten Operationsanwendungen (beliebigen Terme) erlaubt.

Beispiel 1.1.4: In Abbildung 1.1 ist ein einfaches LC-Programm dargestellt. Für dieses Programm gilt $\Sigma_{arithm'} := (\{+(^{(2)}, *(^{(2)}, -(^{(2)}), \mathbb{Z})$. Hierbei wurde im Gegensatz zur formalen Definition die übliche Infixnotation verwendet. Diese wird auch oft in Beispielen Verwendung finden, um die Übersichtlichkeit bei „klassischer Arithmetik“ zu erhöhen.

Zusätzlich zur obigen Definition eines LC-Programms setzen wir im folgenden immer die *Vollständigkeit* voraus:

Definition 1.1.5: Ein LC-Programm π heißt genau dann *vollständig*, wenn jede Variable vor ihrer Benutzung³ definiert wird. Formal: Seien die bis Anweisung α_i in $\beta = \alpha_1; \dots; \alpha_n$ definierten Variablen mit $DV_i \subseteq V_\pi$ bezeichnet. Die DV_i sind wie folgt definiert:

$$\begin{aligned}
DV_0 &:= V_{in} \\
DV_i &:= DV_{i-1} \cup \{x \in V \mid \alpha_i = x \leftarrow e\} \text{ für } i \in \{1, \dots, n\}
\end{aligned}$$

Es gilt nun: π ist genau dann vollständig, wenn

²Diese Forderung ist u.a. für die Korrektheit des DAG-Optimierungsverfahrens (Abschnitt 2.2) notwendig.

³Das Vorkommen einer Variable in \vec{v}_{out} kommt einer Benutzung gleich.

- $\alpha_i = x \leftarrow e$ mit $V_e \subseteq DV_{i-1}$ für alle $i \in \{1, \dots, n\}$ und
- $V_{out} \subseteq DV_n$.

Im folgenden gelte die Vollständigkeit für LC-Programme als generelle Voraussetzung, so daß auch in \mathcal{LC} nur vollständige Programme enthalten sind.

Die Definition läßt sich in ein einfaches iteratives Verfahren zum Test der Vollständigkeit umwandeln. Nur wenn π vollständig ist, kann garantiert werden, daß die Ausgabe allein von der Eingabe abhängig ist; sonst könnten undefinierte Variablen das Ergebnis beeinflussen. Es ist leicht ersichtlich, daß das Programm aus Abbildung 1.1 vollständig ist.

1.1.2 Semantik

Bisher haben wir uns nur mit der Struktur und nicht mit der Bedeutung von LC-Programmen auseinandergesetzt. Die Semantik eines LC-Programmes ist sowohl vom zugrundeliegenden Wertebereich der Variablen als auch der Interpretation der Operationssymbole und Konstanten abhängig.

Definition 1.1.6 (Interpretation): Für eine gegebene Signatur $\Sigma = (F, C)$ ist die zugehörige Interpretation eine Σ -Algebra $\mathfrak{A} := (A, \varphi)$ mit Wertebereich (Universum) A und Interpretationsfunktion

$$\begin{aligned} \varphi : F \cup C \cup A &\rightarrow \bigcup_{i=0}^{\infty} \{\delta \mid \delta : A^i \rightarrow A\} \\ \varphi(f) : A^r &\rightarrow A \text{ für } f \in F^{(r)}. \\ \varphi(c) &\in A \text{ für } c \in C \\ \varphi(a) &= a \text{ für alle } a \in A \end{aligned}$$

φ ordnet also einem in der Signatur enthaltenen Operationssymbol f eine Funktion $\varphi(f)$ über A , deren Stelligkeit der Stelligkeit von f entspricht und einer Konstanten einen Wert aus A zu. Außerdem werden alle Werte aus A auf sich selbst abgebildet. Dadurch ergibt sich später eine einfachere Notation (z.B. bei der Konstantenfaltung in Abschnitt 2.1.3).

Beispiel 1.1.7: Bei Σ_{arithm} könnte beispielsweise $\mathfrak{A} = (\mathbb{Z}, \varphi)$ sein und φ den Operationssymbolen ihre „übliche“ Bedeutung auf \mathbb{Z} und jedem Konstantensymbol aus \mathbb{Z} sich selbst als Konstante zuzuordnen. Es wären aber auch grundsätzlich andere Interpretationen denkbar, wie z.B. die „Boolsche Arithmetik“:

$$\begin{aligned} \mathfrak{A} = (\{0, 1\}, \varphi) \text{ mit } \varphi(+)(a, b) &= a \vee b \\ \varphi(*) &(a, b) = a \wedge b \\ \varphi(-)(a) &= 1 - a \\ \varphi(z) &= \begin{cases} 0 & \text{falls } z = 0 \\ 1 & \text{sonst} \end{cases} \text{ für } z \in \mathbb{Z} \end{aligned}$$

Intuitiv kann man sich ein Programm als eine Funktion vorstellen, die einen Vektor, der die Werte der Eingabevariablen repräsentiert (den *Eingabevektor*) auf einen Vektor, der die Werte der Ausgabevariablen darstellt (den *Ergebnisvektor*) abbildet.

Der aktuelle Zustand an einer beliebigen Position im Programm kann als eine Bindung von Variablen an entsprechende Werte an dieser Stelle angesehen werden. Ein Zustand läßt sich daher als Funktion $\sigma : V_\pi \rightarrow A$ darstellen.

Definition 1.1.8: Der Zustandsraum eines LC-Programms $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\Sigma := (F, C)$ und Interpretation $\mathfrak{A} := (A, \varphi)$ für Σ ist durch

$$Z := \{\sigma \mid \sigma : V_\pi \rightarrow A\}$$

festgelegt.

Aus semantischer Sicht beschreibt jede Anweisung α eine Transformation $t_\alpha : Z \rightarrow Z$ eines Zustands in einen anderen. Die Programmsemantik baut induktiv darauf auf.

Definition 1.1.9 (Semantik eines LC-Programms): Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ vollständig mit o.B.d.A. $\vec{v}_{in} := (x_1, \dots, x_s)$, $\vec{v}_{out} := (y_1, \dots, y_t)$, $V_\pi := \{x_1, \dots, x_s, y_1, \dots, y_t, z_1, \dots, z_l\}$. Außerdem sei $\mathfrak{A} := (A, \varphi)$ eine Interpretation von Σ .

Der Eingabevektor $\vec{in} := (in_1, \dots, in_s) \in A^s$ beschreibt die Zuordnung der Eingabewerte zu Eingabevariablen und bestimmt damit die anfängliche Variablenbelegung

$$\sigma_0 = \{x_1 \mapsto in_1, \dots, x_s \mapsto in_s, z_1 \mapsto a, \dots, z_l \mapsto a\}$$

mit beliebigem $a \in A$ (wegen der Vollständigkeit von π können die restlichen Programmvariablen beliebig initialisiert werden).

Die Semantik $\mathfrak{A}[\pi]$ von π bezüglich \mathfrak{A} kann induktiv wie folgt definiert werden:

- Die Semantik $\mathfrak{A}[\alpha] : Z \rightarrow Z$ einer Anweisung α ist $\mathfrak{A}[\alpha]\sigma := \sigma[x/\mathfrak{A}[e]\sigma]$, wobei

$$f[x/a](y) := \begin{cases} a & \text{falls } y = x \\ f(y) & \text{sonst} \end{cases}$$

die Änderung einer Funktion an einer Stelle ist. Für $\mathfrak{A}[e]\sigma$ gilt folgende Fallunterscheidung:

1. $e = y \in V_\pi \Rightarrow \mathfrak{A}[e]\sigma := \sigma(y)$
 2. $e = c \in C_\pi \Rightarrow \mathfrak{A}[e]\sigma := \varphi(c)$
 3. $e = f(u_1, \dots, u_r) \Rightarrow \mathfrak{A}[e]\sigma := \varphi(f)(\mathfrak{A}[u_1]\sigma, \dots, \mathfrak{A}[u_r]\sigma)$
- Für einen Block $\beta = \alpha_1; \dots; \alpha_n$ ist $\mathfrak{A}[\beta] = \mathfrak{A}[\alpha_n] \circ \dots \circ \mathfrak{A}[\alpha_1]$ ⁴.
 - Die Semantik $\mathfrak{A}[\pi] : A^s \rightarrow A^t$ von π ergibt sich dann zu

$$\mathfrak{A}[\pi](\vec{in}) = ((\mathfrak{A}[\beta]\sigma_0)(y_1), \dots, \underbrace{(\mathfrak{A}[\beta]\sigma_0)(y_j)}_{\text{Wert der Ausgabevariablen } y_j}, \dots, (\mathfrak{A}[\beta]\sigma_0)(y_t))$$

Die Semantik ist also unabhängig von der Benennung der Variablen definiert. Es kommt nur auf deren Reihenfolge im Eingabe- und Ausgabevektor an. Es handelt sich um eine *funktionale Semantik*.

Beispiel 1.1.10: Betrachten wir die Semantik einmal anhand eines (aus Übersichtlichkeitsgründen) sehr kurzen Beispielprogramms

$$\begin{aligned} \pi_{\text{kurz}} := (\Sigma_{\text{arithm}}, (x), (y, z), \beta) \text{ mit } \beta := & y \leftarrow -x; \\ & z \leftarrow 2 * y; \end{aligned}$$

⁴o ist die Funktionskomposition $(f \circ g)(x) = f(g(x))$.

Als Interpretation legen wir die übliche Bedeutung von $*$ und $-$ auf \mathbb{Z} zugrunde („Arithmetik auf \mathbb{Z} “). Die Programmsemantik für $in = (4)$ (d.h. $\sigma_0 = \{x \mapsto 4, y \mapsto 0, z \mapsto 0\}$) berechnet sich wie folgt:

$$\begin{aligned}
\mathfrak{A}[\beta]\sigma_0 &= \mathfrak{A}[z \leftarrow 2 * y] \circ \mathfrak{A}[y \leftarrow -x]\sigma_0 \\
&= \mathfrak{A}[z \leftarrow 2 * y]\sigma_0[y/\mathfrak{A}[-x]\sigma_0] \\
&= \mathfrak{A}[z \leftarrow 2 * y]\sigma_0[y/\varphi(-)(\mathfrak{A}[x]\sigma_0)] \\
&= \mathfrak{A}[z \leftarrow 2 * y]\sigma_0[y/\varphi(-)(\sigma_0(x))] \\
&= \mathfrak{A}[z \leftarrow 2 * y]\sigma_0[y/\varphi(-)(4)] \\
&= \mathfrak{A}[z \leftarrow 2 * y]\underbrace{\sigma_0[y/ - 4]}_{=: \sigma_1} \\
&= \sigma_1[z/\mathfrak{A}[2 * y]\sigma_1] \\
&= \sigma_1[z/\varphi(*) (\mathfrak{A}[2]\sigma_1, \mathfrak{A}[y]\sigma_1)] \\
&= \sigma_1[z/\varphi(*) (\varphi(2), \sigma_1(y))] \\
&= \sigma_1[z/\varphi(*) (2, -4)] \\
&= \sigma_1[z/ - 8] \\
&= \{x \mapsto 4, y \mapsto -4, z \mapsto -8\} \\
&= \{y \mapsto -4, z \mapsto -8\}
\end{aligned}$$

Man erhält somit:

$$\mathfrak{A}[\pi_{\text{kurz}}](4) = ((\mathfrak{A}[\beta]\sigma_0)(y), (\mathfrak{A}[\beta]\sigma_0)(z))(4) = (-4, -8)$$

Man sieht, daß die Anweisungen der Reihe nach abgearbeitet werden; die Variablen der linken Seite werden jeweils mit dem Wert (eigentlich der Semantik) des Ausdrucks der rechten Seite belegt. Beim Treffen auf ein Funktionssymbol f wird dessen Interpretation $\varphi(f)$ auf die Interpretationen der Argumente angewendet. Diese ergeben sich bei Variablen x aus der bisherigen Variablenbelegung $\sigma(x)$.

Konstanten werden ebenfalls durch ihre Interpretationen ersetzt. Die zuletzt erhaltene Belegung der Ausgabevariablen wird schließlich in den Ausgabevektor übertragen.

1.2 Eigenschaften von LC-Programmen

Da jetzt die Semantik, also die Bedeutung eines LC-Programms definiert ist, können weitere Eigenschaften von LC-Programmen untersucht werden. Einige davon beruhen auf der Termdarstellung von LC-Programmen.

1.2.1 Termdarstellung von LC-Programmen

Man kann ein LC-Programm bezüglich einer Ausgabevariablen als Term darstellen. Dies ist u.a. für die Definition der Semantik und für die Untersuchung der Äquivalenz von LC-Programmen hilfreich. Dazu benötigen wir zunächst den Begriff des Terms:

Definition 1.2.1 (Terme): Die Menge der Terme $T_\Sigma(X)$ bezüglich einer Signatur $\Sigma = (F, C)$ und einer Variablenmenge $X \subseteq V$ sei induktiv wie folgt definiert:

1. $C \subseteq T_\Sigma(X)$,
2. $X \subseteq T_\Sigma(X)$ und
3. $f(t_1, \dots, t_r) \in T_\Sigma(X)$ wenn $t_1, \dots, t_r \in T_\Sigma(X)$ und $f \in F^{(r)}$.

Ein variablenfreier Term $t \in T_\Sigma(\emptyset) =: T_\Sigma$ heißt *Grundterm*.

Auch die rechten Seiten von Zuweisungen in LC-Programmen sind Terme, für die allerdings nur eine Schachtelungsebene erlaubt ist. Zur Bestimmung einer Termdarstellung eines LC-Programms liegt es daher nahe, die rechten Seiten der Anweisungen mit den Ausgabevariablen beginnend rückwärtig „ineinanderzuschieben“.

Definition 1.2.2: Die *Termdarstellung* $t_\pi(y) \in T_\Sigma(V_{in})$ eines LC-Programms $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta := \alpha_1; \dots; \alpha_n$ bezüglich einer Ausgabevariablen $y \in V_{out}$ berechnet sich wie folgt:

$$\begin{aligned} t_\pi^{(n)}(y) &:= y \\ t_\pi^{(i-1)}(y) &:= t_\pi^{(i)}(y)[x/e] \text{ für } \alpha_i = x \leftarrow e \text{ und } i \in \{1, \dots, n\} \\ t_\pi(y) &:= t_\pi^{(0)}(y) \end{aligned}$$

Dabei sei $[x/t] : T_\Sigma(X) \rightarrow T_\Sigma(X)$ die Substitution aller Vorkommen der Variablen $x \in X$ durch den Term $t \in T_\Sigma(X)$.

Es werden also sukzessive die im jeweils aktuellen Term vorhandenen Variablen ersetzt, so daß am Ende nur noch die Eingabevariablen übrigbleiben. Substituiert man diese durch die Eingabewerte erhält man einen Grundterm.

Beispiel 1.2.3: Zur Verdeutlichung hier die Berechnung der Termdarstellung $t_\pi(u)$ des Programms aus Abbildung 1.1 bezüglich der Ausgabevariablen u :

i	α_i	$t_\pi^{(i)}$
0		$t_\pi^{(1)}[u/3] = *(2, -(-(x, y), y))$
1	$u \leftarrow 3;$	$t_\pi^{(2)}[v/-(x, y)] = *(2, -(-(x, y), y))$
2	$v \leftarrow x - y;$	$t_\pi^{(3)}[w/+(u, 1)] = *(2, -(-(x, y), y))$
3	$w \leftarrow u + 1;$	$t_\pi^{(4)}[x/-(x, y)] = *(2, -(-(x, y), y))$
4	$x \leftarrow x - y;$	$t_\pi^{(5)}[u/-(x, y)] = *(2, -(x, y))$
5	$u \leftarrow x - y;$	$t_\pi^{(6)}[z/*(u, w)] = *(2, u)$
6	$z \leftarrow u * w;$	$t_\pi^{(7)}[u/*(2, u)] = *(2, u)$
7	$u \leftarrow 2 * u;$	$t_\pi^{(8)}[v/-(w, 1)] = u$
8	$v \leftarrow w - 1;$	u

Es ergibt sich also $t_\pi(u) = *(2, -(-(x, y), y))$; man sieht, daß viele Anweisungen keinen Einfluß auf die Termdarstellung haben, da deren linke Seiten nicht im vorher berechneten Term vorkommen.

Wenn man die Termdarstellungen bezüglich aller Ausgabevariablen eines LC-Programms π ermittelt, erhält man eine andere Darstellungsform für π , die aber äquivalent zu π ist, denn die Semantik von π läßt sich mithilfe der Termdarstellung definieren.

Lemma 1.2.4: Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\Sigma := (F, C)$, $\vec{v}_{in} = (x_1, \dots, x_s)$ und $\vec{v}_{out} = (y_1, \dots, y_t)$. Dann gilt:

$$\mathfrak{A}[\pi](\vec{in}) = (\mathfrak{A}[t_\pi(y_1)[x_1/in_1, \dots, x_s/in_s]], \dots, \mathfrak{A}[t_\pi(y_t)[x_1/in_1, \dots, x_s/in_s]])$$

wobei die (Grund-)Termsemantik $\mathfrak{A}[\cdot]$ mit Grundterm $t \in T_{\Sigma'}$ und $\Sigma' = (F, C \cup A)$ wie folgt definiert ist:

1. $\mathfrak{A}[a] := a$ für $a \in A$
2. $\mathfrak{A}[c] := \varphi(c)$ für $c \in C$
3. $\mathfrak{A}[f(t_1, \dots, t_r)] := \varphi(f)(\mathfrak{A}[t_1], \dots, \mathfrak{A}[t_r])$ für $t_i \in T_{\Sigma'}$

Man setzt also einfach die Semantiken der Termdarstellungen von π bezüglich der Ausgabevariablen in den Ergebnisvektor ein, wobei die Eingabevariablen in den Termen durch die Eingabewerte ersetzt werden (dies ergibt einen Term über der neuen Signatur Σ').

Die Termsemantik ist die sukzessive Anwendung der Interpretationen der Funktionssymbole auf die jeweiligen Teilterme, wobei Konstanten durch sich selbst und Konstantensymbole durch φ interpretiert werden. Die Korrektheit dieser Vorgehensweise ist offensichtlich.

1.2.2 Äquivalenz

Für die Programmoptimierung ist der Begriff der *Äquivalenz* von zwei Programmen von entscheidender Bedeutung. Programme müssen so transformiert werden, daß das Ergebnisprogramm äquivalent zum Quellprogramm ist, sonst wäre das Optimierungsverfahren nicht korrekt.

Definition 1.2.5 (Äquivalenz): Zwei LC-Programme π_1 und π_2 über einer Signatur Σ heißen genau dann *äquivalent unter einer Interpretation* \mathfrak{A} ($\pi_1 \sim_{\mathfrak{A}} \pi_2$), wenn $\mathfrak{A}[\pi_1] = \mathfrak{A}[\pi_2]$. Gilt $\pi_1 \sim_{\mathfrak{A}} \pi_2$ für alle Interpretationen \mathfrak{A} , dann heißen sie *stark äquivalent* ($\pi_1 \sim \pi_2$).

Zwei äquivalente LC-Programme berechnen also die gleiche „Funktion“. Dabei ist klar, daß die starke Äquivalenz eine hinreichende Bedingung für die schwache Äquivalenz darstellt. Man mag sich fragen, inwieweit die (starke) Äquivalenz zweier LC-Programme, die ja eine stark beschränkte Teilklasse normaler Programme darstellen, entscheidbar ist.

Mithilfe der Termdarstellung erhält man eine in Linearzeit berechenbare Bedingung für die starke Äquivalenz:

Satz 1.2.6: Für die Termdarstellungen zweier LC-Programme $\pi_i := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_i)$, $i \in \{1, 2\}$ mit o.B.d.A. $\vec{v}_{in} = (x_1, \dots, x_s)$ und $\vec{v}_{out} = (y_1, \dots, y_t)$ ⁵ gilt $\pi \sim \pi'$, wenn $t_{\pi_1}(y_j) = t_{\pi_2}(y_j)$ für alle $j \in \{1, \dots, t\}$.

Beweis: Nach Lemma 1.2.4 gilt für die Semantik:

$$\mathfrak{A}[\pi_i](\vec{in}) = (\mathfrak{A}[t_{\pi_i}(y_1)[x_1/in_1, \dots, x_s/in_s]], \dots, \mathfrak{A}[t_{\pi_i}(y_t)[x_1/in_1, \dots, x_s/in_s]]), \quad i \in \{1, 2\}.$$

Gilt nun für alle $j \in \{1, \dots, s\}$ $t_{\pi_1}(y_j) = t_{\pi_2}(y_j)$, so gilt aufgrund derselben Eingabewerte für beide Programme ebenfalls $t_{\pi_1}(y_j)[x_1/in_1, \dots, x_s/in_s] = t_{\pi_2}(y_j)[x_1/in_1, \dots, x_s/in_s]$ für alle $j \in \{1, \dots, s\}$.

Die Anwendung der Semantikoperation $\mathfrak{A}[\cdot]$ auf zwei gleiche Terme ergibt natürlich auch das gleiche Ergebnis, da die Semantik deterministisch ist. \square

Daß der obige Satz nur für die starke Äquivalenz gilt, kann man sich leicht an zwei einfachen LC-Programmen klarmachen:

⁵Die Vektoren der Ein- und Ausgabevariablen müssen gemäß der Semantikdefinition die gleiche Länge haben. Daher kann man durch Umbenennung der Variablen eines Programms die gleiche Benennung von Ein- und Ausgabevariablen erreichen.

Beispiel 1.2.7: Sei $\pi_1 = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_1)$ und $\pi_2 = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_2)$ mit $\vec{v}_{in} = (x, y)$, $\vec{v}_{out} = (z)$, $\beta_1 = z \leftarrow x + y$ und $\beta_2 = z \leftarrow y + x$.

Für die beiden Termdarstellungen gilt $t_{\pi_1}(z) = +(x, y) \neq +(y, x) = t_{\pi_2}(z)$ und daher nicht $\pi_1 \sim \pi_2$. Wählen wir nun die Interpretation $\mathfrak{A} := (\mathbb{Z}, \varphi)$, wobei $\varphi(+)$ die übliche Addition auf den ganzen Zahlen ist. Es ist leicht ersichtlich, daß aufgrund des Kommutativgesetzes die Ergebnisse der beiden Berechnungen für alle Eingabewerte identisch sind und daher $\mathfrak{A}[\pi_1] = \mathfrak{A}[\pi_2]$, d.h. $\pi_1 \sim_{\mathfrak{A}} \pi_2$ gilt.

Man kann also nicht schließen, daß bei Ungleichheit der Termdarstellungen die Programme unter keinen Umständen äquivalent sind, da dies von der jeweiligen Interpretation abhängig ist.

Satz 1.2.8 (Unentscheidbarkeit der Äquivalenz): Gegeben seien zwei LC-Programme π_1 und π_2 über Σ und eine Interpretation \mathfrak{A} von Σ . Dann ist im allgemeinen nicht entscheidbar, ob $\pi_1 \sim_{\mathfrak{A}} \pi_2$ gilt.

Beweisidee: Sei $\pi = (\Sigma, \varepsilon, (y), \beta)$ ein LC-Programm ohne Eingabevariablen und mit genau einer Ausgabevariable y . Wähle $\Sigma := (\{+, /, \{1\}\})$ und definiere $\mathfrak{A} := (\varphi, \mathbb{Z})$ für $a, b \in \mathbb{Z}$ wie folgt:

$$\begin{aligned} \varphi(+)(a, b) &:= a +_{\mathbb{Z}} b \\ \varphi(/)(a, b) &:= \begin{cases} \lfloor \frac{a}{b} \rfloor & \text{falls } \frac{a}{b} > 0 \\ 0 & \text{falls } b = 0 \\ \lceil \frac{a}{b} \rceil & \text{sonst} \end{cases} \quad (\text{Erweiterte ganzzahlige Division}) \\ \varphi(1) &:= 1 \end{aligned}$$

Es reicht nun zu zeigen, daß das *Nulläquivalenzproblem* für π , d.h. ob $\mathfrak{A}[\pi] = (0)$ gilt, unentscheidbar ist. Die Korrektheit dieser Vereinfachung ist offensichtlich, denn es läßt sich leicht ein nulläquivalentes Programm $\pi' := (\Sigma, \varepsilon, (y), y \leftarrow 1/2)$ angeben.

Um die Unentscheidbarkeit zu beweisen bietet sich die Reduktion des Problems auf ein bekanntes unentscheidbares Problem an. Hierzu eignet sich das zehnte Hilbertsche Problem:

Entscheide, ob ein diophantisches Polynom⁶ $F(x_1, \dots, x_r)$ eine Lösung besitzt, d.h. ob $a_1, \dots, a_r \in \mathbb{N}$ existieren, so daß $F(a_1, \dots, a_r) = 0$ gilt.

Es ist bekannt, daß das zehnte Hilbertsche Problem unentscheidbar ist [5]. Man konstruiert nun ein Programm θ , das genau dann Null ausgibt, wenn das diophantische Polynom

$$F(x_1, \dots, x_r) := \sum_{j=1}^m c_j x_1^{j_1} \dots x_r^{j_r} - \sum_{j=m+1}^n c_j x_1^{j_1} \dots x_r^{j_r} \quad (c_j > 0, j_i > 0)$$

keine Lösung besitzt.

Gemäß dem Reduktionsprinzip, würde damit aus der Entscheidbarkeit des Nulläquivalenzproblems von θ die Entscheidbarkeit des zehnten Hilbertschen Problems folgen. Da dieses aber unentscheidbar ist, ist auch das Nulläquivalenzproblem von θ unentscheidbar.

Die Konstruktion von $\theta \in \mathcal{LC}$ ist recht aufwendig und kann in [4] nachgelesen werden.

Wie man sieht, ist schon bei wenigen, relativ einfachen Operationen die Entscheidbarkeit der Äquivalenz von LC-Programmen nicht mehr gewährleistet. Man kann jedoch auch Interpretationen angeben, bei denen man ein positives Entscheidbarkeitsresultat erhält. Eine solche ist die „klassische“ Arithmetik auf \mathbb{Z} mit $+, -, *$.

⁶Polynom F mit ganzzahligen Koeffizienten, wobei man sich nur für die ganzzahligen Lösungen ($F = 0$) interessiert.

Satz 1.2.9: Sei $\Sigma := (F, \mathbb{Z})$ mit $F := \{+(^{(2)}, -(^{(2)}, *(^{(2)})\}$, $\mathfrak{A} := (\varphi, \mathbb{Z})$, $\varphi(\circ) := \circ_{\mathbb{Z}}$ für $\circ \in F$ und $\varphi(z) = z$ für alle $z \in \mathbb{Z}$. Ferner seien $\pi_i := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_i) \in \mathcal{LC}$ für $i \in \{1, 2\}$ mit $\vec{v}_{in} = (x_1, \dots, x_s)$, $\vec{v}_{out} = (y_1, \dots, y_t)$ und $V_{\pi_1} = V_{\pi_2}$. Dann ist entscheidbar, ob $\pi_1 \sim_{\mathfrak{A}} \pi_2$ gilt.

Beweis: Beide Programme werden in eine Folge von Polynomen $P_j^{(i)} := P_j^{(i)}(x_1, \dots, x_s)$ für $j \in \{1, \dots, t\}$ umgewandelt. Hierbei ist $P_j^{(i)}$ die *Polynomdarstellung* von π_i bezüglich der Ausgabevariablen y_j . Sei n die maximale Anzahl der $*$ -Operatoren in beiden Programmen. Das Verfahren läuft in folgenden Schritten ab:

1. Berechne für $i \in \{1, 2\}$ und $j \in \{1, \dots, t\}$ die Termdarstellung $t_{\pi_i}(y_j)$ von π_i bezüglich y_j .
2. Wandle $t_{\pi_i}(y_j)$ für $i \in \{1, 2\}$, $j \in \{1, \dots, t\}$ in Polynome um. Für einen Term $t \in T_{\Sigma}(X)$ bestimmt man das zugehörige Polynom $P[t]$ induktiv wie folgt:
 - Falls $t = z \in \mathbb{Z}$, setze $P[t] := z$.
 - Falls $t = x \in X$, setze $P[t] := x$.
 - Falls $t = +(t_1, t_2)$ mit $t_1, t_2 \in T_{\Sigma}(X)$. Dann ist $P[t] := P[t_1] + P[t_2]$.
 - Falls $t = -(t_1, t_2)$ mit $t_1, t_2 \in T_{\Sigma}(X)$. Dann ist $P[t] := P[t_1] - (P[t_2])$.
 - Falls $t = *(t_1, t_2)$ mit $t_1, t_2 \in T_{\Sigma}(X)$. Dann ist $P[t] := (P[t_1]) \cdot (P[t_2])$.
3. Transformiere $P[t_{\pi_i}(y_j)]$ für alle $i \in \{1, 2\}$, $j \in \{1, \dots, t\}$ in ein Polynom der Form

$$P_j^{(i)}(x_1, \dots, x_s) := \sum_{k_1=0}^n \sum_{k_2=0}^n \dots \sum_{k_s=0}^n c_{k_1 k_2 \dots k_s}^{(i,j)} \cdot x_1^{k_1} \cdot x_2^{k_2} \cdot \dots \cdot x_s^{k_s}$$

durch sukzessives Ausmultiplizieren. Die Länge⁷ dieser Darstellung ist $O(n^s)$, d.h. sie ist exponentiell in der Anzahl der Eingabevariablen. Daher hat auch die Transformation mindestens diese Komplexität.

4. Prüfe, ob $c_{k_1 \dots k_s}^{(1,j)} = c_{k_1 \dots k_s}^{(2,j)}$ für alle $j \in \{1, \dots, t\}$, $k_l \in \{1, \dots, n\}$, $l \in \{1, \dots, s\}$. Falls obiges gilt, dann sind die beiden Programme äquivalent, sonst sind sie nicht äquivalent. \square

Dieses Verfahren ist praktisch aufgrund des immensen Aufwands normalerweise nicht einsetzbar. Laut [12] ist das Äquivalenzproblem für die klassische Arithmetik auf \mathbb{Z} (ohne Division) aber probabilistisch in Polynomzeit lösbar (dies gilt auch für die Arithmetik auf \mathbb{R} (mit Division)). Für die Optimierung⁸ von Programmen stellt die Äquivalenzprüfung jedoch nur ein Teilproblem dar (siehe Abschnitt 1.2.4).

1.2.3 Kostenmaße

Für die Optimierung von Programmen ist es notwendig, Programmen Kostenwerte zuzuordnen. Nur so können zwei Programme miteinander verglichen werden. Bei LC-Programmen ist die Kostenberechnung relativ einfach, da keine Schleifen vorhanden sind, bei denen zur Compilezeit nicht entscheidbar ist, wie oft sie durchlaufen werden.

Definition 1.2.10 (Kostenfunktion): Eine Funktion $c : \mathcal{LC} \rightarrow \mathbb{R}_0^+$ heißt Kostenfunktion, wenn für zwei LC-Programme $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; \dots; \alpha_n)$ und $\pi' := (\Sigma, \vec{v}'_{in}, \vec{v}'_{out}, \alpha_j; \dots; \alpha_k)$ mit $1 \leq$

⁷Anzahl der Summanden

⁸Diesmal ist die „wirkliche“ Optimierung gemeint: Die Erzeugung eines optimalen Programms bezüglich eines Kostenmaßes.

$j, k \leq n$ gilt:

$$c(\pi') \leq c(\pi) \quad (\text{Monotonie})$$

Je kleiner der Kostenwert $c(\pi)$ für ein Programm π ist, desto „besser“ ist π bezüglich c . Die Monotonie erzwingt, daß Teilprogrammen geringere Kosten zugeordnet werden, was unsinnigen Konstruktionen von Kostenfunktionen vorbeugt. Als Kostenmaße für $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$ und $\alpha_i := x_i \leftarrow e_i$ kommen u.a. in Frage:

1. Zahl der Anweisungen: $c_l(\pi) := n$
2. Anzahl der Operationen:

$$c_{op}(\pi) := |\{i \in \{1, \dots, n\} \mid e_i = f(u_1, \dots, u_r) \text{ mit } F^{(r)}\}|$$

3. Summe der Operationsgewichte: $c_{wop}(\pi) := \sum_{i=1}^n c_{wop}(\alpha_i)$ mit

$$c_{wop}(\alpha_i) := \begin{cases} w_j & \text{falls } \alpha_i = x \leftarrow f_j(u_1, \dots, u_r) \text{ mit } f_j \in F^{(r)} \\ a & \text{falls } \alpha_i = x \leftarrow c \text{ mit } c \in C \\ b & \text{falls } \alpha_i = x \leftarrow y \text{ mit } y \in V \end{cases}$$

wobei o.B.d.A. $F = \{f_1, \dots, f_m\}$ die Operationssymbole sind, w_j das Gewicht von Operation f_j ist, a die Kosten einer Konstantenzuweisung und b die Kosten einer Kopieranweisung sind.

4. Speicherbedarf: $c_{mem}(\pi) := |V_\pi|$

Für Aufzählungsverfahren von LC-Programmen gemäß ihrer Kosten (wie in Abschnitt 1.2.4) benötigt man noch eine zusätzliche Einschränkung von Kostenfunktionen:

Definition 1.2.11: Eine Kostenfunktion c heißt *endlich*, falls für alle $r \in \mathbb{R}_0^+$ gilt: $|c^{-1}(r)| < \infty$.

Es ist leicht ersichtlich, daß von den oben dargestellten Funktionen keine in jedem Fall endlich ist. c_l ist nur endlich, wenn die Menge der Konstantensymbole C endlich ist. c_{wop} ist endlich, falls $w_j > 0$ für alle j und $a, b > 0$ sowie ebenfalls $|C| < \infty$ gilt, denn dann führt jede Verlängerung eines Programms zu einer Kostenerhöhung und für einen Kostenwert existieren nur endlich viele Programme.

c_{op} ist hingegen nie endlich, da Kopieranweisungen und Zuweisungen von Konstanten die Kosten nicht erhöhen. Gleiches gilt für c_{mem} , hier können beliebig Anweisungen unter Verwendung schon definierter Variablen hinzugefügt werden.

1.2.4 Optimalität

Bei der Optimierung von LC-Programmen versucht man das Ausgangsprogramm so zu transformieren, daß man ein äquivalentes Programm erhält, dessen Kosten bezüglich einer gegebenen Kostenfunktion günstiger sind. Wie wir noch sehen werden, kann ein Kostenminimum meist nicht erreicht werden.

Definition 1.2.12: Sei c eine Kostenfunktion, $\pi, \pi_1, \pi_2 \in \mathcal{LC}$ mit $\pi_1 \sim_{\mathfrak{A}} \pi_2$ für eine Interpretation \mathfrak{A} und $opt : \mathcal{LC} \rightarrow \mathcal{LC}$ eine programmverbessernde Transformation. Dann gilt:

- π_1 ist c -besser als π_2 $:\Leftrightarrow \pi_1 \leq_c \pi_2$ $:\Leftrightarrow c(\pi_1) \leq c(\pi_2)$
- π ist c -optimal $:\Leftrightarrow \forall \pi' \in \mathcal{LC} \text{ mit } \pi' \sim_{\mathfrak{A}} \pi : \pi \leq_c \pi'$.
- π ist opt -optimal $:\Leftrightarrow opt(\pi) = \pi$.

Um die Optimalität eines LC-Programms zu entscheiden, ist es folglich nötig, die Äquivalenz von LC-Programmen zu entscheiden, welche aber nach Satz 1.2.8 im allgemeinen unentscheidbar ist. Dieses negative Resultat bedeutet, daß es in der Praxis für hinreichend komplexe Interpretationen unmöglich ist, ein äquivalentes optimales LC-Programm zu einem gegebenen Programm zu bestimmen. Daher können nur Approximationen berechnet werden bzw. programmverbessernde Transformationen auf das Ausgangsprogramm angewendet werden.

Wenn man die Arithmetik auf \mathbb{Z} (wie in Satz 1.2.9) zugrundelegt, ist die Bestimmung eines c -optimalen Programmes u.U. entscheidbar.

Algorithmus 1.2.13: Sei c eine endliche Kostenfunktion⁹. Bestimmung eines c -optimalen Programmes zu π mit Σ und \mathfrak{A} wie in Satz 1.2.9 (Arithmetik auf \mathbb{Z}):

1. Zähle alle LC-Programme ohne Konstanten gemäß aufsteigenden c -Kosten auf.
2. Prüfe für jedes Programm π' der Aufzählung, ob $\pi \sim_{\mathfrak{A}} \pi'$ gilt.
3. Gilt $\pi' \leq_c \pi$ für ein $\pi' \sim_{\mathfrak{A}} \pi$, breche ab und gebe aus: „ π' ist c -optimal“.
4. Gilt für ein π' der Aufzählung $\pi' =_c \pi$, breche das Verfahren ab und gebe aus: „ π ist c -optimal“.

Wenn man keine Endlichkeit der Kostenfunktion fordert, ist die Aufzählung (Schritt 1) nicht möglich und daher keine Entscheidbarkeit gegeben. Praktisch ist der Algorithmus dennoch nicht einsetzbar (selbst bei Verwendung des polynomiellen probabilistischen Ansatzes nach [12]), da zusätzlich zum Prüfen der Äquivalenz für jedes Programm auch noch eine Aufzählung der Programme durchgeführt werden muß.

Der Algorithmus 1.2.13 läßt sich übrigens in leicht modifizierter Form auch zum Finden eines stark äquivalenten, bzgl. c optimalen Programmes nutzen. Dabei kann auch bei Verwendung von c_l eine unendliche Menge von Konstantensymbolen verwendet werden, da die Menge der im Programm tatsächlich vorkommenden Konstanten endlich ist und keine Konstantenfaltung durchgeführt wird.

Praktisch ist dieser Ansatz aufgrund der sehr hohen Aufzählkomplexität dennoch nicht sinnvoll einsetzbar.

⁹Dies erlaubt die Benutzung von Konstanten nur dann, wenn nur endlich viele Konstantensymbole zugelassen sind.

2 Optimierungsverfahren für LC-Programme

Während in Kapitel 1 die grundlegenden Definitionen und Zusammenhänge dargestellt wurden, wenden wir uns in diesem Kapitel nun den Optimierungsverfahren für LC-Programme zu. Optimierungen laufen dabei normalerweise in zwei Phasen ab:

1. *Analysephase*: Sammlung der für die Transformation des Programms notwendigen *Analyseinformationen*. Das Programm bleibt zunächst unverändert.
2. *Transformation*: Durchführung der eigentlichen Optimierung; das Programm wird unter Benutzung der Analyseinformationen äquivalenzerhaltend modifiziert.

In Abschnitt 2.1 werden wir uns zunächst mit den „klassischen“ Optimierungsverfahren wie z.B. der Dead Code Elimination, bei der „nutzlose“, d.h. nicht ausgaberelevante Anweisungen entfernt werden oder der Common Subexpression Elimination, die Ersetzung gemeinsamer Teilausdrücke, beschäftigen. In Abschnitt 2.2 wird dann eine graphbasierte Optimierungsmethode für LC-Programmen vorgestellt, die DAG-Optimierung. Zuletzt werden in Abschnitt 2.3 noch einige weitere interessante Ansätze dargelegt.

Bevor wir die einzelnen Verfahren vorstellen, wollen wir zunächst einige grundlegende notwendige Eigenschaften einer Optimierungsfunktion definieren.

Definition 2.0.14 (Idempotenz): Eine Abbildung $T : \mathcal{LC} \rightarrow \mathcal{LC}$ heißt idempotent, falls $T(T(\pi)) = T(\pi)$ für alle $\pi \in \mathcal{LC}$ gilt.

Die Idempotenz besagt also, daß nach einer Anwendung einer Transformation T eine weitere Applikation von T keine Änderungen mehr hervorruft.

Definition 2.0.15 (Programmtransformation): Eine Abbildung $T : \mathcal{LC} \rightarrow \mathcal{LC}$ heißt Programmtransformation, wenn gilt:

1. $T(\pi) \sim_{\mathfrak{A}} \pi$ für eine Interpretation \mathfrak{A} (Korrektheit),
2. $T(T(\pi)) = T(\pi)$ (Idempotenz) und
3. für alle $\pi \in \mathcal{LC}$ ist $T(\pi)$ allein von π abhängig (Determinismus).

Man mag die Forderung der Idempotenz als nicht unbedingt notwendig ansehen, aber da die von uns genauer betrachteten Verfahren ohnehin idempotent sind, treten hierdurch keine Probleme auf. Der Determinismus sorgt dafür, daß man bei gleicher Eingabe immer das gleiche Ergebnis erhält. Dies ist insbesondere beweistechnisch von großer Bedeutung.

2.1 Klassische Optimierungsverfahren

Die klassischen Optimierungsverfahren für LC-Programme sind von einfacher Struktur und berücksichtigen jeweils einen zu optimierenden Teilaspekt. Im allgemeinen ist daher eine Verkettung der einzelnen Verfahren erforderlich, um eine gute Optimierung zu erreichen.

Die Analysephase aller klassischen Verfahren basiert auf dem gleichen Prinzip: Es wird eine Startinformation festgelegt und diese dann für jede Anweisung modifiziert, so daß man für jede Anweisung eine Analyseinformation erhält.

2.1.1 Dead Code Elimination

Die Dead Code Elimination dient der Entfernung von Anweisungen, die überflüssig sind, weil sie keinen Einfluß auf die Programmsemantik haben. Eine Anweisung $x \leftarrow e$ stellt *Dead Code* dar, falls x bis zu seiner Neudefinition nicht mehr (auf der rechten Seite einer Anweisung oder als Ausgabevariable) benutzt wird.

Als Analyse kommt die sogenannte *Live Variable Analysis* (oder auch *Needed Variable Analysis*) zum Einsatz, die für jede Anweisung eine Menge von „lebendigen“ Variablen bestimmt. Die Live Variable Analysis stellt eine *Rückwärtsanalyse* dar. Das bedeutet, daß ausgehend von den Ausgabevariablen die Mengen der notwendigen Variablen rückwärtig berechnet werden. Die Ausgabevariablen sind dabei zunächst die einzigen lebendigen Variablen (Startinformation).

Definition 2.1.1 (Live Variable Analysis): Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ ein LC-Programm mit $\beta = \alpha_1; \dots; \alpha_n$. Definiere für eine Anweisung $\alpha = x \leftarrow e$ die Transferfunktion $t_\alpha : \mathfrak{P}(V_\pi) \rightarrow \mathfrak{P}(V_\pi)$ wie folgt:

$$t_\alpha(M) := \begin{cases} (M \setminus \{x\}) \cup V_e & \text{falls } x \in M \\ M & \text{sonst} \end{cases}$$

Die Transferfunktionen $t_{\alpha_n}, \dots, t_{\alpha_1}$ bestimmen nun ausgehend von V_{out} die Mengen der notwendigen Variablen („live variables“):

$$\begin{aligned} LV_n &:= V_{out} \\ LV_{i-1} &:= t_{\alpha_i}(LV_i) \text{ für } i \in \{0, \dots, n-1\} \end{aligned}$$

Für eine Anweisung $x \leftarrow e$ wird also genau dann x aus der Analysemenge herausgenommen und die Variablen V_e der rechten Seite eingefügt, wenn x in der Analysemenge enthalten ist. Ist x in der Menge enthalten ist es eine lebendige Variable, zu deren Berechnung die Variablen der rechten Seite benötigt werden. Diese sind folglich dann im nächsten Schritt lebendig. Ist x nicht in der Analysemenge enthalten, handelt es sich bei der Anweisung um Dead Code, d.h. sie ist für die Programmsemantik irrelevant und kann weggelassen werden.

Definition 2.1.2 (Dead Code Elimination): Definiere für ein Programm $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta = \alpha_1; \dots; \alpha_n$ die Programmtransformation $T_{DC} : \mathcal{LC} \rightarrow \mathcal{LC}$ zur Beseitigung von Dead Code wie folgt:

$$\begin{aligned} T_{DC}(\pi) &:= (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta') \text{ mit } \beta' := t_{DC}(\alpha_1); \dots; t_{DC}(\alpha_n) \\ t_{DC}(\alpha_i = x \leftarrow e) &:= \begin{cases} \alpha_i & \text{falls } x \in LV_i \\ \varepsilon & \text{sonst} \end{cases} \end{aligned}$$

Die Berechnung von LV_0 könnte man zur Bestimmung der notwendigen Eingabevariablen benutzen, im folgenden wollen wir von der Entfernung überflüssiger Eingabevariablen aber absehen, da dies die Semantik, wie sie in Abschnitt 1.1.2 festgelegt wurde, verändern würde.

Beispiel 2.1.3: Zur Demonstration hier die Berechnungsschritte für das Beispielprogramm aus Abbildung 1.1:

i	α_i	LV_i	Dead Code?
1	$u \leftarrow 3;$	$t_{\alpha_2}(LV_2) = LV_2 = \{u, x, y\}$	Nein
2	$v \leftarrow x - y;$	$t_{\alpha_3}(LV_3) = LV_3 \setminus \{w\} \cup \{u\} = \{u, x, y\}$	Ja
3	$w \leftarrow u + 1;$	$t_{\alpha_4}(LV_4) = LV_4 \setminus \{x\} \cup \{x, y\} = \{w, x, y\}$	Nein
4	$x \leftarrow x - y;$	$t_{\alpha_5}(LV_5) = LV_5 \setminus \{u\} \cup \{x, y\} = \{w, x, y\}$	Nein
5	$u \leftarrow x - y;$	$t_{\alpha_6}(LV_6) = LV_6 = \{u, w\}$	Nein
6	$z \leftarrow u * w;$	$t_{\alpha_7}(LV_7) = LV_7 \setminus \{u\} \cup \{u\} = \{u, w\}$	Ja
7	$u \leftarrow 2 * u;$	$t_{\alpha_8}(LV_8) = LV_8 \setminus \{v\} \cup \{w\} = \{u, w\}$	Nein
8	$v \leftarrow w - 1;$	$V_{out} = \{u, v\}$	Nein

Es werden also die Anweisungen α_2 und α_6 von T_{DC} entfernt.

Die Dead Code Elimination ist unabhängig von der Programmsemantik, d.h. die Interpretation der Operationssymbole ist beliebig und verändert das Ergebnis des Verfahrens nicht. T_{DC} erzeugt also stark äquivalente Programme.

Im folgenden werden wir untersuchen, ob die Dead Code Elimination die Forderungen von Definition 2.0.15 erfüllt und damit eine Programmtransformation ist. Die wichtigste Eigenschaft ist dabei die Korrektheit, denn der Einsatz eines nicht korrekten Verfahrens wäre zwecklos.

Satz 2.1.4 (Korrektheit): Für alle $\pi \in \mathcal{LC}$ gilt $\pi \sim T_{DC}(\pi)$.

Beweis: Es ist zu zeigen, daß $\mathfrak{A}[\llbracket T_{DC}(\pi) \rrbracket] = \mathfrak{A}[\llbracket \pi \rrbracket]$ für jede Interpretation \mathfrak{A} gilt (alternativ wäre nach Satz 1.2.6 ein Beweis über die Termdarstellung möglich). Induktion über die Anzahl der Anweisungen $n \in \mathbb{N}$:

- $n = 0$:
 $\beta = \varepsilon$, also gilt trivialerweise $\mathfrak{A}[\llbracket T_{DC}(\pi) \rrbracket] = \mathfrak{A}[\llbracket \pi \rrbracket]$, da somit auch $\vec{v}_{out} = \varepsilon$ (sonst wäre die Vollständigkeit nicht gewährleistet).

- $n \rightarrow n + 1$:

Induktionsannahme: $\mathfrak{A}[\llbracket T_{DC}(\pi) \rrbracket] = \mathfrak{A}[\llbracket \pi \rrbracket]$ für alle $\pi \in \mathcal{LC}$ mit $\beta = \alpha_1; \dots; \alpha_n$.

Definiere $\pi_n := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_n)$ mit $\beta_n := \alpha_1; \dots; \alpha_n$. Außerdem sei $\pi_{n+1} := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta_{n+1}) \in \mathcal{LC}$ mit $\beta_{n+1} = \alpha_1; \dots; \alpha_n; \alpha_{n+1}$ und $\alpha_{n+1} = x \leftarrow e$.

Unterscheidung von zwei Fällen:

1. $x \notin V_{out}$:

α_{n+1} Dead Code und beeinflusst die Semantik von π_{n+1} nicht. Somit folgt für π_n :

$$\mathfrak{A}[\llbracket \pi_{n+1} \rrbracket] = \mathfrak{A}[\llbracket \pi_n \rrbracket] \underbrace{=}_{\text{I.A.}} \mathfrak{A}[\llbracket T_{DC}(\pi_n) \rrbracket] \underbrace{=}_{\text{gemäß Def.}} \mathfrak{A}[\llbracket T_{DC}(\pi_{n+1}) \rrbracket]$$

2. $x \in V_{out}$:

Für π_n gilt nach Induktionsannahme: $\mathfrak{A}[\llbracket T_{DC}(\pi_n) \rrbracket] = \mathfrak{A}[\llbracket \pi_n \rrbracket]$

Es gilt daher

$$\begin{aligned} (\mathfrak{A}[\llbracket \beta_{n+1} \rrbracket \sigma_0](y) &= (\mathfrak{A}[\llbracket \beta_n \rrbracket \sigma_0](y) \text{ für } y \in V_{out} \setminus \{x\} \\ (\mathfrak{A}[\llbracket \beta_{n+1} \rrbracket \sigma_0](x) &= (\mathfrak{A}[e](\mathfrak{A}[\llbracket \beta_n \rrbracket \sigma_0]))(x) \end{aligned}$$

Die Rückführung von $\mathfrak{A}[\llbracket T_{DC}(\pi_{n+1}) \rrbracket]$ auf $\mathfrak{A}[\llbracket T_{DC}(\pi_n) \rrbracket]$ funktioniert ähnlich:

Es gilt: $V_{out} = LV_{n+1} \Rightarrow LV_n = V_{out} \setminus \{x\} \cup V_e$.

In π_{n+1} und π_n werden daher die gleichen Anweisungen entfernt. Es folgt

$$\begin{aligned} \mathfrak{A}[\llbracket T_{DC}(\beta_{n+1}) \rrbracket \sigma_0](y) &= \mathfrak{A}[\llbracket T_{DC}(\beta_n) \rrbracket \sigma_0](y) \text{ für } y \in V_{out} \setminus \{x\} \\ \mathfrak{A}[\llbracket T_{DC}(\beta_{n+1}) \rrbracket \sigma_0](x) &= \mathfrak{A}[e](\mathfrak{A}[\llbracket T_{DC}(\beta_n) \rrbracket \sigma_0](x)) \end{aligned}$$

Insgesamt gilt also $\mathfrak{A}[\llbracket \pi_{n+1} \rrbracket] = \mathfrak{A}[\llbracket T_{DC}(\pi_{n+1}) \rrbracket]$. □

Die zweite eine Programmtransformation definierende Eigenschaft ist die Idempotenz.

Satz 2.1.5 (Idempotenz): *Die Dead Code Elimination ist idempotent, es gilt also $T_{DC}(T_{DC}(\pi)) = T_{DC}(\pi)$ für alle $\pi \in \mathcal{LC}$.*

Beweis: Es genügt zu zeigen, daß T_{DC} die LV -Informationen der nach der ersten T_{DC} -Anwendung noch vorhandenen Anweisungen nicht verändert. Wir betrachten hier zur Vereinfachung nur die Elimination einer einzelnen Anweisung. Eine induktive Erweiterung auf beliebig viele ist ohne weiteres möglich.

Es sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; \dots; \alpha_n)$ und $\pi' := T_{DC}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; \dots; \alpha_{i-1}; \alpha_{i+1}; \dots; \alpha_n)$, d.h. $\alpha_i = x \leftarrow e$ mit $x \notin LV_i$. Da die Live Variable Analysis eine Rückwärtsanalyse ist, bleiben LV_{i+1}, \dots, LV_n unverändert. Für den Fall $i = 1$ ist daher nichts mehr zu zeigen.

Für $i > 1$ gilt im Ausgangsprogramm π :

$$\begin{aligned} LV_{i-1} &= t_{\alpha_i}(LV_i) \\ &= \begin{cases} LV_i \setminus \{x\} \cup V_e & \text{falls } x \in LV_i \\ LV_i & \text{sonst} \end{cases} \\ &= LV_i \quad (\text{da } x \notin LV_i) \\ &= \begin{cases} V_{out} & \text{falls } i = n \\ t_{\alpha_{i+1}}(LV_{i+1}) & \text{sonst} \end{cases} \end{aligned}$$

In π' erhalten wir das gleiche Ergebnis:

$$LV_{i-1} = \begin{cases} V_{out} & \text{falls } i - 1 = n - 1 \quad (\text{d.h. } i = n) \\ t_{\alpha_{i+1}}(LV_{i+1}) & \text{sonst} \end{cases}$$

Annahme: T_{DC} entfernt zusätzlich $\alpha_j = x' \leftarrow e'$, $i \neq j$ aus π' .

Dann folgt $x' \notin LV_j$ in π' und $x' \notin LV_j$ in π . Dann wäre α_j aber schon nach der ersten Anwendung von T_{DC} aus π entfernt worden. □

Aus der Idempotenz folgt, daß man nach nur einer Anwendung der Dead Code Elimination schon ein bzgl. Dead Code Elimination optimales Programm erhält. Wir werden aber später sehen, daß ein T_{DC} -optimales Programm nach Zwischenschaltung einer anderen Optimierung u.U. nicht mehr T_{DC} -optimal ist, z.B. „erzeugt“ die Konstantenfaltung (siehe Abschnitt 2.1.3) Dead Code.

Die Dead Code Elimination arbeitet offensichtlich deterministisch, denn es gibt keine Wahlmöglichkeiten bei einem der Schritte. Das Resultat ist folglich eindeutig und es handelt sich bei der Dead Code Elimination um eine Programmtransformation nach Definition 2.0.15.

Nun wünscht man sich natürlich nicht nur, daß einige formale Eigenschaften erfüllt werden, sondern auch, daß man eine meßbare Verbesserung durch die Anwendung eines Verfahrens erreichen kann.

Korollar 2.1.6: *Für die Dead Code-optimierte Version $T_{DC}(\pi)$ von $\pi \in \mathcal{LC}$ gilt: $T_{DC}(\pi) \leq_l \pi$ und $T_{DC}(\pi) \leq_{op} \pi$.*

Die Korrektheit des Korollars ist sofort ersichtlich, da bei der Dead Code Elimination nur Anweisungen entfernt werden können, aber keine hinzugefügt werden. Daher kann die Zahl der Operationen und die Programmlänge nicht ansteigen.

2.1.2 Common Subexpression Elimination

Anders als die Dead Code Elimination kommt bei der Common Subexpression Elimination eine *Vorwärtsanalyse*, die *Available Expression Analysis*, zum Einsatz. Die induktive Berechnung beginnt folglich bei der ersten Anweisung. Während der Analysephase bestimmt man für jede Anweisung die Menge der dort *verfügbaren* Ausdrücke. Diese werden jeweils durch die Indizes der Anweisungen ihres ersten Auftretens repräsentiert.

Definition 2.1.7: Ein Ausdruck e ist an einer Position i *verfügbar*, falls e in einer Anweisung α_j mit $j < i$ berechnet und danach die in ihm auftretenden Variablen bis zur i -ten Anweisung nicht mehr verändert wurden.

Wenn nun die i -te Anweisung ebenfalls den Ausdruck e berechnet, kann man durch die Zwischenspeicherung des Berechnungsergebnisses in einer temporären Variablen eine Berechnung einsparen. Dabei wird aber die Programmlänge erhöht.

Definition 2.1.8 (Available Expression Analysis): Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta = \alpha_1; \dots; \alpha_n$. Definiere für $\alpha_i := x_i \leftarrow e_i$, $i \in \{1, \dots, n\}$ die Analysefunktion $t_{\alpha_i} : \mathfrak{P}(\{1, \dots, n\}) \rightarrow \mathfrak{P}(\{1, \dots, n\})$ durch:

$$t_{\alpha_i}(M) := \text{kill}_{\alpha_i} \circ \text{gen}_{\alpha_i}$$

wobei für die Hilfsfunktionen $\text{gen}_{\alpha_i}, \text{kill}_{\alpha_i} : \mathfrak{P}(\{1, \dots, n\}) \rightarrow \mathfrak{P}(\{1, \dots, n\})$ gilt:

$$\text{gen}_{\alpha_i}(M) := \begin{cases} M \cup \{i\} & \text{falls } e_i = f(u_1, \dots, u_r) \text{ und } \forall j \in M : e_j \neq e_i \\ M & \text{sonst} \end{cases}$$

$$\text{kill}_{\alpha_i}(M) := M \setminus \{j \in M \mid x_i \in V_{e_j}\}$$

Dies ergibt die Mengen der bei $i \in \{1, \dots, n\}$ verfügbaren Ausdrücke $AE_i \subseteq \{1, \dots, n\}$:

$$AE_1 := \emptyset$$

$$AE_{i+1} := t_{\alpha_i}(AE_i) \text{ für } i \in \{1, \dots, n\}$$

Durch die Berechnung der *AE*-Informationen können nun für jede Anweisung die verfügbaren Ausdrücke ermittelt werden. Für die anschließende Programmtransformation benötigt man aber die „rückwärtige“ Sicht: Es muß für den Operationsausdruck einer Anweisung bestimmt werden, an welchen Stellen, an denen er verfügbar ist, er sich wiederholt. Es soll zudem nur eine temporäre Variable erzeugt werden, falls dies auch nötig ist, also die Anzahl der Wiederholungen mindestens eins ist.

Definition 2.1.9 (Gültige Wiederholungen): Die Funktion $wh : \{1, \dots, n\} \rightarrow \mathfrak{P}(\{1, \dots, n\})$ gibt für jeden Ausdruck seine gültigen Wiederholungen an:

$$wh(i) := \{j \in \{i+1, \dots, n\} \mid i \in AE_j, e_i = e_j\}$$

Jetzt kann die Programmtransformation¹ definiert werden:

¹Daß es sich wirklich um eine Programmtransformation im formalen Sinne handelt, wird später gezeigt.

Definition 2.1.10 (Common Subexpression Elimination): Die Programmtransformation $T_{CS} : \mathcal{LC} \rightarrow \mathcal{LC}$ arbeitet wie folgt: Für $i \in \{1, \dots, n\}$ mit $wh(i) \neq \emptyset$ wähle $t_i \in V \setminus V_\pi$ und

1. ersetze $\alpha_i = x \leftarrow e$ durch $t_i \leftarrow e$; $x \leftarrow t_i$, sowie
2. ersetze für alle $j \in wh(i)$ die Anweisungen $\alpha_j = y \leftarrow e$ durch $y \leftarrow t_i$.

Beispiel 2.1.11: Auch für die Common Subexpression Elimination wollen wir wieder eine Beispielrechnung für das LC-Programm aus Abbildung 1.1 angeben:

i	α_i	AE_i	$wh(i)$	Neue Anweisung(en)
1	$u \leftarrow 3$;	\emptyset	\emptyset	
2	$v \leftarrow x - y$;	$t_{\alpha_1}(AE_1) = \emptyset$	$\{4\}$	$t_2 \leftarrow x - y$; $v \leftarrow t_2$
3	$w \leftarrow u + 1$;	$t_{\alpha_2}(AE_2) = (AE_2 \cup \{2\}) \setminus \emptyset = \{2\}$	\emptyset	
4	$x \leftarrow x - y$;	$t_{\alpha_3}(AE_3) = (AE_3 \cup \{3\}) \setminus \emptyset = \{2, 3\}$	\emptyset	$x \leftarrow t_2$
5	$u \leftarrow x - y$;	$t_{\alpha_4}(AE_4) = (AE_4 \cup \{4\}) \setminus \{2, 4\} = \{3\}$	\emptyset	
6	$z \leftarrow u * w$;	$t_{\alpha_5}(AE_5) = (AE_5 \cup \{5\}) \setminus \{3\} = \{5\}$	\emptyset	
7	$u \leftarrow 2 * u$;	$t_{\alpha_6}(AE_6) = (AE_6 \cup \{6\}) \setminus \emptyset = \{5, 6\}$	\emptyset	
8	$v \leftarrow w - 1$;	$t_{\alpha_7}(AE_7) = (AE_7 \cup \{7\}) \setminus \{6, 7\} = \{5\}$	\emptyset	

Wie man sieht, ist die Programmlänge um eins größer die des Ausgangsprogramms, es wurde aber eine Subtraktion eingespart.

Nun ist noch zu zeigen, daß die Common Subexpression Elimination eine Programmtransformation darstellt. Der Determinismus ist offenbar gegeben.

Satz 2.1.12 (Korrektheit): Für alle $\pi \in \mathcal{LC}$ gilt $\pi \sim T_{CS}(\pi)$, d.h. die Common Subexpression Elimination erhält die starke Äquivalenz.

Beweis: Nach Satz 1.2.6 reicht es für die starke Äquivalenz aus, zu zeigen, daß sich die Termdarstellung von π für die Ausgabevariablen nicht ändert, d.h. $t_\pi(v) = t_{T_{CS}(\pi)}(v)$ für alle $v \in V_{out}$.

Ähnlich wie im Beweis von Satz 2.1.5 betrachten wir dabei den vereinfachten Fall, daß genau ein Berechnungsausdruck mit genau einer gültigen Wiederholung von T_{CS} optimiert wird.

Es sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; \dots; \alpha_n)$ und $\pi' := T_{CS}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$ mit

$$\beta' := \alpha'_1; \dots; \alpha'_{n+1} := \alpha_1; \dots; \alpha_{i-1}; \alpha'_i; \alpha''_i; \alpha_{i+1}; \dots; \alpha'_j; \dots; \alpha_n.$$

D.h. $\alpha_i = x \leftarrow e$ und $\alpha_j = y \leftarrow e$ mit $i \in AE_j$. Dann ist $\alpha'_i = t_i \leftarrow e$, $\alpha''_i = x \leftarrow t_i$ und $\alpha'_j = y \leftarrow t_i$.

Wir betrachten jetzt den Substitutionsterm $s \in T_\Sigma(V_\pi)$ für y an der Stelle i in beiden Programmen. In π ist $s = e$, da sich e aufgrund $i \in AE_j$ nicht mehr verändert. In π' gilt hingegen $s = t_i[t_i/e] = e$, da t_i sonst außer an Stelle i sonst nirgends verändert wird.

Falls nun y in einem Term bei der Berechnung der Termdarstellung von π für $v \in V_{out}$ auftaucht, ergeben sich bei beiden Programmen die gleichen Substitutionen, und daher ist $t_\pi(v) = t_{T_{CS}(\pi)}(v)$ für alle $v \in V_{out}$. \square

Die Common Subexpression Elimination arbeitet wie die Dead Code Elimination rein syntaktisch; die Interpretation der Operations- und Konstantensymbole hat keinen Einfluß auf das Ergebnis. Zur Verbesserung des Optimierungseffekts wäre allerdings eine Erweiterung des Verfahrens unter Ausnutzung semantischer Informationen denkbar, man könnte etwa die Kommutativität von $+$ berücksichtigen, so daß $x + y$ und $y + x$ als gleich angesehen werden. Mithilfe von vorgeschalteten

algebraischen Transformationen (grober Überblick in Abschnitt 2.3.2) ließe sich dies ohne Modifikation von T_{CS} realisieren.

Wie bei der Dead Code Elimination reicht auch bei der Common Subexpression Elimination eine Anwendung aus, um ein T_{CS} -optimales Programm zu erhalten.

Korollar 2.1.13 (Idempotenz): *Es gilt $T_{CS}(T_{CS}(\pi)) = T_{CS}(\pi)$ für alle $\pi \in \mathcal{LC}$.*

Beweisidee: Da bereits bei der ersten Analyse alle verfügbaren Ausdrücke gefunden, ihre gültigen Wiederholungen entsprechend ersetzt, jedoch keine zusätzlichen Operationsausdrücke hinzugefügt werden, kann bei einer zweiten Applikation von T_{CS} nichts mehr optimiert werden.

Damit ist auch die Common Subexpression Elimination eine Programmtransformation.

Korollar 2.1.14: *Für $\pi \in \mathcal{LC}$ gilt $T_{CS}(\pi) <_{op} \pi$ aber $T_{CS}(\pi) >_l \pi$, falls $T_{CS}(\pi) \neq \pi$.*

Es ist klar, daß die Operationskosten durch die Common Subexpression Elimination reduziert werden, da Berechnungsausdrücke durch Kopieranweisungen ersetzt werden. Die Erhöhung der Programmlänge hängt mit dem Einfügen von zusätzlichen Zuweisungen an temporäre Variablen zusammen.

2.1.3 Konstantenfaltung

Die Konstantenfaltung führt eine partielle Auswertung des Eingabeprogramms durch Propagation der Programmkonstanten durch. Dadurch kann man u.U. die Berechnung von konstanten Ausdrücken zur Laufzeit einsparen. Die Semantik fließt hier in die Optimierung mit ein, da zur Auswertung konstanter Ausdrücke die Interpretationen der Konstantensymbole und der entsprechenden Operationssymbole von Bedeutung sind.

Aus formaler Sicht wäre auch eine Konstantenfaltung auf der Basis von Grundtermen denkbar. Diese müßten dann jedoch zur Erzeugung des Ergebnisprogramms sowieso ausgewertet werden (wenn man sie nicht zur Laufzeit auswerten will, wodurch keine Rechenzeit einspart werden kann). Zudem wäre eine umfangreiche Modifikation des Wertebereichs notwendig, d.h. es müßte die Menge der Grundterme T_Σ als Universum der Interpretation zugrundegelegt werden. Eine Unabhängigkeit von Interpretationen ist also auch dann nicht gegeben. Aus diesen Gründen wollen wir diesen Ansatz hier nicht weiter verfolgen.

Das nun vorgestellte Verfahren weicht in der formalen Darstellung etwas von dem aus [2] bekannten ab, da dort nicht zwischen Konstantensymbolen und den eigentlichen Konstanten unterschieden wird, und die Interpretation auf die Behandlung von Operationssymbolen beschränkt ist. Das Grundprinzip ist jedoch identisch.

In der Analysephase, der *Reaching Definition Analysis*, bestimmt man für jede Anweisung die dort gültigen (eingabeunabhängigen) Variablenwerte, dazu wird die aus Definition 1.1.9 bekannte Semantik von Anweisungen so erweitert, daß auch Ausdrücke mit teilweise unbekanntem Variablenwerten „ausgewertet“ werden können.

Definition 2.1.15 (Erweiterte Anweisungssemantik): Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit Signatur $\Sigma = (F, C)$ und $\beta = \alpha_1; \dots; \alpha_n$. Ferner sei $\mathfrak{A} = (A, \varphi)$ eine Interpretation von Σ . Der Zustandsraum sei nun $\bar{Z} := \{\sigma \mid \sigma : V_\pi \rightarrow A \cup \{\perp\}\}$, wobei \perp einen unbekanntem Variablenwert angibt. Die erweiterte

Semantik $\bar{\mathfrak{A}}[\alpha] : \bar{Z} \rightarrow \bar{Z}$ einer Anweisung $\alpha = x \leftarrow e$ für $\sigma \in \bar{Z}$ ist dann:

$$\bar{\mathfrak{A}}[\alpha]\sigma := \begin{cases} \mathfrak{A}[\alpha]\sigma & \text{falls } \forall v \in V_e : \sigma(v) \neq \perp \\ \sigma[x/\perp] & \text{sonst} \end{cases}$$

Mit anderen Worten: Wird auf eine Variable mit unbekanntem Wert eine Funktion angewendet, pflanzt sich der unbekannte Wert fort. Auf spezielle Eigenschaften der Operationen, wie z.B. $\forall x \in \mathbb{R} : 0 \cdot x = 0$ wird dabei nicht geachtet (*strikte* Behandlung).

Definition 2.1.16 (Reaching Definition Analysis): Seien $\pi, \mathfrak{A}, \bar{Z}$ wie in Definition 2.1.15 gegeben. Für eine Anweisung α ist die Transferfunktion $t : \bar{Z} \rightarrow \bar{Z}$ durch

$$t_\alpha(\sigma) := \bar{\mathfrak{A}}[\alpha]\sigma.$$

bestimmt. Nun erhalten wir die Variablenbelegungen $RD_i \in \bar{Z}$ für α_i durch sukzessive Transformation:

$$\begin{aligned} RD_1 &:= \sigma_\perp \text{ mit } \forall v \in V_\pi : \sigma_\perp(v) := \perp \\ RD_{i+1} &:= t_{\alpha_i}(RD_i) \text{ für } i \in \{1, \dots, n\} \end{aligned}$$

Aufgrund der Ersetzung der Konstantensymbole aus C durch Elemente aus A und der Auswertung von konstanten Berechnungsausdrücken (und damit dem Einfügen neuer Konstanten) in der Programmtransformation, besitzt das Ergebnisprogramm eine andere Signatur als das Ausgangsprogramm; sie unterscheidet sich genau in der Menge der Konstantensymbole.

Definition 2.1.17 (Konstantenfaltung): Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$, $\Sigma = (F, C)$ eine Signatur, $\beta = \alpha_1; \dots; \alpha_n$, $\alpha_i = x_i \leftarrow e_i$ und $\mathfrak{A} = (A, \varphi)$ eine Interpretation für Σ . Dann ist die Transformation $T_{CF} : \mathcal{LC} \rightarrow \mathcal{LC}$ zur Konstantenfaltung definiert durch:

$$T_{CF}(\pi) := ((F, A), \vec{v}_{in}, \vec{v}_{out}, \beta') \text{ mit } \beta' := x_1 \leftarrow \overline{RD}_1(e_1); \dots; x_n \leftarrow \overline{RD}_n(e_n)$$

wobei für $\sigma \in \bar{Z}$ (und $c \in C, y \in V, u_i \in V \cup C, f \in F^{(r)}$) gilt:

$$\begin{aligned} \bar{\sigma}(c) &:= \varphi(c) \\ \bar{\sigma}(y) &:= \begin{cases} \sigma(y) & \text{falls } \sigma(y) \neq \perp \\ y & \text{sonst} \end{cases} \\ \bar{\sigma}(f(u_1, \dots, u_r)) &:= \begin{cases} f(\bar{\sigma}(u_1), \dots, \bar{\sigma}(u_r)) & \text{falls } \exists i \in \{1, \dots, r\}, \text{ so daß } \sigma(u_i) = \perp \\ \varphi(f)(\bar{\sigma}(u_1), \dots, \bar{\sigma}(u_r)) & \text{sonst} \end{cases} \end{aligned}$$

Operationsausdrücke werden also komplett zu einer Konstanten ausgewertet, wenn die Belegungen aller Variablenargumente bekannt sind, ansonsten werden alle Variablen, deren Belegungen bekannt sind, durch die entsprechenden Konstantenwerte ersetzt. Konstantensymbole werden aufgrund der notwendigen Signaturänderung durch ihre Interpretationen substituiert.

Beispiel 2.1.18: Wenn man T_{CF} auf das Programm aus Abbildung 1.1 anwendet, ergibt sich folgende Rechnung:

i	α_i	RD_i	Neue Anweisung α'_i
1	$u \leftarrow 3;$	σ_{\perp}	$u \leftarrow \varphi(3); = \mathbf{u} \leftarrow \mathbf{3};$
2	$v \leftarrow x - y;$	$\bar{\mathfrak{A}}[\alpha_1] \sigma_{\perp} = \sigma_{\perp}[u/3] =: \sigma_1$	$v \leftarrow \bar{\sigma}_1(x) - \bar{\sigma}_1(y); = \mathbf{v} \leftarrow \mathbf{x} - \mathbf{y};$
3	$w \leftarrow u + 1;$	$\bar{\mathfrak{A}}[\alpha_2] \sigma_1 = \sigma_1[v/\perp] = \sigma_1$	$w \leftarrow \varphi(+)(\bar{\sigma}_1(u), \varphi(1)); = \mathbf{w} \leftarrow \mathbf{4};$
4	$x \leftarrow x - y;$	$\bar{\mathfrak{A}}[\alpha_3] \sigma_1 = \sigma_1[w/4] =: \sigma_2$	$x \leftarrow \bar{\sigma}_2(x) - \bar{\sigma}_2(y); = \mathbf{x} \leftarrow \mathbf{x} - \mathbf{y};$
5	$u \leftarrow x - y;$	$\bar{\mathfrak{A}}[\alpha_4] \sigma_2 = \sigma_2[x/\perp] = \sigma_2$	$u \leftarrow \bar{\sigma}_2(x) - \bar{\sigma}_2(y); = \mathbf{u} \leftarrow \mathbf{x} - \mathbf{y};$
6	$z \leftarrow u * w;$	$\bar{\mathfrak{A}}[\alpha_5] \sigma_2 = \sigma_2[u/\perp] =: \sigma_3$	$z \leftarrow \bar{\sigma}_3(u) * \bar{\sigma}_3(w); = \mathbf{x} \leftarrow \mathbf{u} * \mathbf{4};$
7	$u \leftarrow 2 * u;$	$\bar{\mathfrak{A}}[\alpha_6] \sigma_3 = \sigma_3[z/\perp] = \sigma_3$	$u \leftarrow \varphi(2) * \bar{\sigma}_3(u); = \mathbf{u} \leftarrow \mathbf{2} * \mathbf{u};$
8	$v \leftarrow w - 1;$	$\bar{\mathfrak{A}}[\alpha_7] \sigma_3 = \sigma_3[u/\perp] = \sigma_3$	$v \leftarrow \varphi(-)(\bar{\sigma}_3(w), \varphi(1)); = \mathbf{v} \leftarrow \mathbf{3};$

Wie bereits erwähnt, produziert die Konstantenfaltung zusätzlichen Dead Code, was man sich leicht am Beispiel veranschaulichen läßt: Die erste Anweisung ist z.B. überflüssig, da u bis zu seiner nächsten Zuweisung in Anweisung 5 auf keiner rechten Seite auftritt. Dies liegt daran, daß in Anweisung 3 eine Optimierung stattgefunden hat, und dort u nicht mehr benötigt wird.

Bei der Korrektheit der Konstantenfaltung muß, im Gegensatz zu den bisherigen Verfahren, nun die jeweilige Interpretation berücksichtigt werden. Die starke Äquivalenz gilt also diesmal nicht.

Satz 2.1.19 (Korrektheit): Für $\pi \in \mathcal{LC}$ und eine Interpretation \mathfrak{A} gilt $\pi \sim_{\mathfrak{A}} T_{CF}(\pi)$.

Beweis: Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\Sigma := (F, C)$, $\beta := \alpha_1; \dots; \alpha_n$ und $\alpha_i := x_i \leftarrow e_i$ sowie $\mathfrak{A} := (A, \varphi)$ eine Interpretation von Σ . Es reicht zu zeigen, daß $\mathfrak{A}[[e_i]] = \mathfrak{A}[[e'_i]]$ für alle $i \in \{1, \dots, n\}$, wobei $e'_i = \overline{RD}_i(e)$.

Fallunterscheidung gemäß Typ des Ausdrucks e_i :

1. $e_i = c \in C$

Dann ist $e'_i := \varphi(c)$ und es gilt dann für eine Variablenbelegung $\sigma \in Z$:

$$\mathfrak{A}[[e_i]]\sigma = \varphi(c) \stackrel{\text{Def. von } \varphi}{=} \varphi(\varphi(c)) = \mathfrak{A}[[e'_i]]\sigma$$

2. $e_i = y \in V$, zwei Teilfälle:

a) $RD_i(y) = \perp \Rightarrow e'_i = e_i$, d.h. die Semantiken sind trivialerweise gleich.

b) $RD_i(y) = a \in A \Rightarrow e'_i = a$. In diesem Fall gilt für $\sigma \in Z$:

$$\mathfrak{A}[[e_i]]\sigma = \underbrace{\sigma(y)}_{\in A} \stackrel{\text{Def. von } \varphi}{=} \varphi(\sigma(y)) = \mathfrak{A}[[e'_i]]\sigma$$

3. $e_i = f(u_1, \dots, u_r)$, wiederum zwei Fälle:

a) $\exists j \in \{1, \dots, r\}$, so daß $RD_i(u_j) = \perp$:

$e'_i = f(\overline{RD}_i(u_1), \dots, \overline{RD}_i(u_n))$ und es ist zu zeigen:

$$\begin{aligned} \mathfrak{A}[[e_i]]\sigma &= \varphi(f)(\mathfrak{A}[[u_1]]\sigma, \dots, \mathfrak{A}[[u_r]]\sigma) \\ &= \varphi(f)(\mathfrak{A}[[\overline{RD}_i(u_1)]]\sigma, \dots, \mathfrak{A}[[\overline{RD}_i(u_r)]]\sigma) = \mathfrak{A}[[e'_i]]\sigma \end{aligned}$$

Dazu muß bewiesen werden, daß $\mathfrak{A}[[u_j]]\sigma = \mathfrak{A}[[\overline{RD}_i(u_j)]]\sigma$ für $j \in \{1, \dots, r\}$. Da die u_j aber entweder Variablen oder Konstanten sind, folgt die Aussage bereits aus Fall 1 oder 2.

b) $\forall j \in \{1, \dots, r\}$ gilt entweder $RD_i(u_j) = a \in A$ oder $u_j \in C$.

Es ist $e'_i = \varphi(f)(\overline{RD}_i(u_1), \dots, \overline{RD}_i(u_n))$. Der einzige Unterschied zu Teilfall (a) ist die

zusätzliche Anwendung der Interpretation von f . Da diese zur Berechnung der Semantik aber sowieso angewendet werden muß, ergibt sich das gleiche Problem wie unter (a).

Die obigen Folgerungen lassen sich für alle e_i , $i \in \{1, \dots, n\}$ anwenden und daher ist der Satz bewiesen. \square

Korollar 2.1.20 (Idempotenz): *Es gilt $T_{CF}(T_{CF}(\pi)) = T_{CF}(\pi)$ für $\pi \in \mathcal{LC}$.*

Die Idempotenz der Konstantenfaltung ist intuitiv ebenfalls klar, da schon bei der ersten Anwendung alle Variablen, die konstante Werte besitzen, bzw. Operationen mit konstanten Operanden durch Konstanten ersetzt werden. Es bleiben also nur „unbekannte“ Variablen und Funktionssymbole, die mindestens einen unbekanntem Operanden besitzen, bestehen. Eine weitere Applikation von T_{CF} würde also keine Verbesserung bringen.

Da die Konstantenfaltung deterministisch arbeitet, besitzt sie somit die Eigenschaften einer Programmtransformation. Eine Betrachtung von T_{CF} unter dem Kostengesichtspunkt führt zu folgender Aussage:

Korollar 2.1.21: *Für $\pi \in \mathcal{LC}$ gilt $T_{CF}(\pi) \leq_{op} \pi$ und $T_{CF}(\pi) =_l \pi$.*

Die Richtigkeit des Korollars ist offensichtlich, da keine Operationen hinzugefügt, sondern höchstens ausgewertet und durch Konstanten ersetzt werden. Die Programmlänge wird nicht verändert, da keine Anweisungen entfernt werden. Eine Verbesserung bzgl. Codelänge ist zu erreichen, indem man eine Dead Code Elimination „nachschaltet“.

2.1.4 Propagation von Kopien

Kommen wir nun zur letzten der klassischen Optimierungen, der Propagation von Kopien, im folgenden abkürzend auch *Kopierpropagation* genannt. Die Kopierpropagation substituiert Variablen, die nach Kopieranweisungen benutzt werden, sofern möglich durch die Ursprungsvariable. Dabei werden auch transitive Beziehungen, sogenannte *Kopierketten*, berücksichtigt.

Dem Analyseschritt liegt die Idee zugrunde, für eine Programmstelle i die Menge der an i gültigen Kopien zu bestimmen, d.h. Zuweisungen $x \leftarrow y$ bei denen weder x noch y bis zu Position i neudefiniert wurde. Zusätzlich ist nach jeder Veränderung die *transitive Hülle* der entsprechenden Analysemenge zu berechnen.

Eine gültige Kopie wird durch die Tupeldarstellung (x, y, d) repräsentiert. Sie bedeutet, daß an der entsprechenden Programmposition x den Wert von y besitzt, wobei d die *transitive Tiefe* ist, d.h. es gibt d Kopieranweisungen, die eine Kette „von y nach x bilden“. Die transitive Tiefe muß zusätzlich erfaßt werden, um einen deterministischen Algorithmus zu erhalten.

Definition 2.1.22 (Bestimmung der gültigen Kopien): Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta = \alpha_1; \dots; \alpha_n$. Definiere für eine Anweisung $\alpha = x \leftarrow e$ die Transferfunktion

$$t_\alpha : V_\pi \times V_\pi \times \{1, \dots, n\} \rightarrow V_\pi \times V_\pi \times \{1, \dots, n\} \text{ mit}$$

$$t_{x \leftarrow e}(M) := trans(M \setminus \{(y, z, d) \in M \mid d \in \{1, \dots, n\}, x \in \{y, z\}\} \cup \{(x, e, 1) \mid e \in V_\pi \setminus \{x\}\})$$

Hierbei berechnet $trans : V \times V \times \mathbb{N} \rightarrow V \times V \times \mathbb{N}$ die *transitive Hülle* der ersten beiden Argumente

für ein $M \subseteq V \times V \times \mathbb{N}$:

$$\begin{aligned} \text{trans}(M) &:= \{(x, y, d) \mid \exists k \in \mathbb{N} : \exists z_1, \dots, z_k \in V, \exists d_0, \dots, d_k \in \mathbb{N}, \text{ so daß} \\ &\quad \{(x, z_1, d_0), (z_1, z_2, d_1), \dots, (z_{k-1}, z_k, d_{k-1}), (z_k, y, d_k)\} \subseteq M\} \\ &\text{wobei } d := \sum_{i=0}^k d_i \end{aligned}$$

Insbesondere sind für $k = 0$ auch alle $(x, y, d) \in M$ in $\text{trans}(M)$ enthalten. Nun können induktiv die Analyseinformationen $CP_i \subseteq V_\pi \times V_\pi \times \{1, \dots, n\}$ berechnet werden:

$$\begin{aligned} CP_1 &:= \emptyset \\ CP_{i+1} &:= t_{\alpha_i}(CP_i) \text{ für } i \in \{1, \dots, n-1\} \end{aligned}$$

Die CP -Informationen müssen transitiv abgeschlossen sein, um eine korrekte Rückverfolgung von Ketten von Kopieranweisungen zu ermöglichen. Die transitive Hülle muß dabei schon während der Berechnung der CP -Informationen bestimmt werden, denn sonst könnten transitive Beziehungen durch Elimination eines „Kettengliedes“ zerstört werden, obwohl die Beziehung weiterhin gilt (siehe dazu auch Beispiel 2.1.24). Dies wurde in dem im Rahmen von [2] vorgestellten Verfahren nicht berücksichtigt. Daher wäre dann unter Umständen eine mehrfache Anwendung des Verfahrens nötig, d.h. die Idempotenz der Kopierpropagation könnte nicht gewährleistet werden.

Die Programmtransformation nutzt die CP -Informationen, um Variablen in Anweisungen durch ihre gleichwertigen Vorgänger zu ersetzen (*Rücksstitution*).

Definition 2.1.23 (Propagation von Kopien): Die Programmtransformation $T_{CP} : \mathcal{LC} \rightarrow \mathcal{LC}$ für $\pi = (\Sigma, V_{in}, V_{out}, \beta) \in \mathcal{LC}$ mit $\beta = x_1 \leftarrow e_1; \dots; x_n \leftarrow e_n$ und $\Sigma = (F, C)$ ist gegeben durch

$$T_{CP}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta') \text{ mit } \beta' := x_1 \leftarrow \overline{CP}_1(e_1); \dots; x_n \leftarrow \overline{CP}_n(e_n)$$

wobei für $\overline{CP}(e)$ mit $CP \subseteq V_\pi \times V_\pi \times \{1, \dots, n\}$ sowie $c \in C$, $x \in V_\pi$, $f \in F^{(r)}$ folgende Definition gilt:

$$\begin{aligned} \overline{CP}(c) &:= c \\ \overline{CP}(x) &:= \begin{cases} y & \text{falls } \exists y \in V, \exists d \in \{1, \dots, n\} \text{ mit } (x, y, d) \in CP \\ & \text{und } \forall (x, y', d') \in CP : d' \leq d \\ x & \text{sonst} \end{cases} \\ \overline{CP}(f(u_1, \dots, u_r)) &:= f(\overline{CP}(u_1), \dots, \overline{CP}(u_r)) \end{aligned}$$

Durch die Wahl des Tupels mit der höchsten transitiven Tiefe für die Substitution, wird sichergestellt, daß Kopierketten immer komplett zurückverfolgt werden und das Verfahren deterministisch arbeitet, da aufgrund der Berechnung der transitiven Hülle für eine Variable x mehrere Tupel der Form (x, y, d) mit unterschiedlichen y und d existieren könnten.

Aufgrund des Verbots von (indirekten) „Selbstzuweisungen“ in Definition 1.1.3 können keine Einträge der Form (x, x, d) auftreten und die Terminierung ist garantiert. Die Programmtransformation verkürzt den Code vorerst nicht; eine Verbesserung ist erst durch eine darauffolgende Dead Code Elimination erzielbar, welche dann u.U. überflüssige Kopieranweisungen entfernt.

Beispiel 2.1.24: Da das bisher zur Demonstration der Optimierungsverfahren verwendete Beispielprogramm (aus Abb. 1.1) wegen fehlender Kopieranweisungen schlecht zur Demonstration von T_{CP} geeignet ist, wenden wir T_{CP} auf ein neues Beispielprogramm mit zwei Kopieranweisungen, welches in Abbildung 2.1 dargestellt ist, an:

$$\begin{aligned}
V_{in} &: x, y \\
\beta &: u \leftarrow x; \\
& y \leftarrow u + y; \\
& v \leftarrow u; \\
& u \leftarrow v * y; \\
& v \leftarrow u + v; \\
V_{out} &: u, v
\end{aligned}$$

Abbildung 2.1: $\pi = (\Sigma_{\text{arithm}}, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit Kopieranweisungen

i	α_i	CP_i	Neue Anweisung α'_i
1	$u \leftarrow x;$	\emptyset	$u \leftarrow \overline{CP}_1(x); = \mathbf{u} \leftarrow \mathbf{x};$
2	$y \leftarrow u + y;$	$\{(u, x, 1)\}$	$y \leftarrow \overline{CP}_2(u) + \overline{CP}_2(y); = \mathbf{y} \leftarrow \mathbf{x} + \mathbf{y};$
3	$v \leftarrow u;$	$\{(u, x, 1)\}$	$v \leftarrow \overline{CP}_3(u); = \mathbf{v} \leftarrow \mathbf{x};$
4	$u \leftarrow v * y;$	$\{(u, x, 1), (v, u, 1), (\mathbf{v}, \mathbf{x}, \mathbf{2})\}$	$u \leftarrow \overline{CP}_4(v) * \overline{CP}_4(y); = \mathbf{u} \leftarrow \mathbf{x} * \mathbf{y};$
5	$v \leftarrow u + v$	$\{(\mathbf{v}, \mathbf{x}, \mathbf{2})\}$	$v \leftarrow \overline{CP}_5(u) * \overline{CP}_5(v); = \mathbf{v} \leftarrow \mathbf{u} + \mathbf{x};$

In Anweisung 4 ergibt sich das durch Fettdruck hervorgehobene Tupel aus der Berechnung der transitiven Hülle. Würde man dieses nicht schon bei der Berechnung der CP -Informationen einfügen, so würde sich $CP_5 = \emptyset$ ergeben, obwohl v nach wie vor den Wert von x besitzt. Daher könnte man selbst bei Berücksichtigung von transitiven Ketten bei der Programmtransformation v in Anweisung 5 zunächst nicht durch x ersetzen. Deswegen wäre eine wiederholte Anwendung von T_{CP} nötig.

Man sieht leicht, daß die Anwendung einer Dead Code Elimination auf das obige Resultat die Anweisungen 1 und 3 eliminieren würde.

Korollar 2.1.25: Für $\pi \in \mathcal{LC}$ gilt $\pi =_l T_{CP}(\pi)$ und $\pi =_{op} T_{CP}(\pi)$.

Da nur Variablen durch andere Variablen ausgetauscht werden, jedoch keine Anweisungen entfernt oder Rechenausdrücke in anderer Weise modifiziert werden, ist die Aussage offensichtlich.

Wie die Dead Code Elimination und die Common Subexpression Elimination ist auch die Kopierpropagation unabhängig von der Interpretation, sie erhält also die starke Äquivalenz.

Satz 2.1.26 (Korrektheit): Für $\pi \in \mathcal{LC}$ gilt $\pi \sim T_{CP}(\pi)$.

Beweis: Nach Satz 1.2.6 reicht es zu zeigen, daß $t_\pi = t_{T_{CP}(\pi)}$ gilt. Dazu betrachten wir den vereinfachten Fall der Ersetzung einer Variablen in einer Anweisung.

Es sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \alpha_1; \dots; \alpha_n)$ und $\pi' := T_{CP}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$ mit

$$\beta' := \alpha'_1; \dots; \alpha'_n := \alpha_1; \dots; \alpha_{j-1}; \alpha'_j; \alpha_{j+1}; \dots; \alpha_n.$$

Es ist also $\alpha_j = z \leftarrow e$ mit $x \in V_e^2$ und $(x, y, d) \in CP_j$ und für alle $(x, y', d') \in CP_j$ gilt $d' \leq d$. Daher existieren $i_1, \dots, i_d \in \{1, \dots, n\}$ mit $i_k < i_l$ für $k < l$ und $i_d < j$ sowie paarweise verschiedene³

² x ist die zu ersetzende Variable.

³Gemäß Definition 1.1.3 kann keine Variable (über Umwegen) sich selbst zugewiesen werden.

$z_1, \dots, z_{d-1} \in V_\pi$, so daß

$$\begin{aligned} \alpha_{i_1} &= z_{d-1} \leftarrow y \\ &\vdots \\ \alpha_{i_k} &= z_{d-k} \leftarrow z_{d-k+1} \\ &\vdots \\ \alpha_{i_d} &= x \leftarrow z_1 \end{aligned}$$

und $(z_{d-k+1}, y, k-1) \in CP_{i_k}$ für $k \in \{2, \dots, d\}$ (da d maximal ist, ist dies für x die Kopierkette mit der größtmöglichen Länge). Dann ist $\alpha'_j = z \leftarrow e[x/y]$, d.h. die Variable x wird in e durch y ersetzt. Betrachten wir nun die Substitution für x in π :

In α_{i_d} wird x durch x_1 ersetzt, dann x_1 durch x_2 in $x_{i_{d-1}}$ und in α_{i_1} schließlich x_{d-1} durch y . Die Substitution ist also die gleiche.

Daraus folgt, daß falls in einem Term $t_\pi^{(j-1)}(v)$ bezüglich $v \in V_{out}$ die Variable x vorkommt (dies ist der Fall, falls in $t_\pi^{(j)}(v)$ z vorkam), x in π und in π' letztlich durch y ersetzt wird und damit die Termdarstellungen $t_\pi(v)$ und $t_{\pi'}(v)$ gleich sind.

Auch für die Kopierpropagation ist die Idempotenz gegeben, wodurch mit dem Determinismus die Bedingungen aus Definition 2.0.15 erfüllt sind:

Korollar 2.1.27: *Es gilt $T_{CP}(T_{CP}(\pi)) = T_{CP}(\pi)$ für $\pi \in \mathcal{LC}$.*

Durch die Berechnung der transitiven Hülle und die Wahl des Tupels mit der maximalen transitiven Tiefe wird die mehrfache Anwendung von T_{CP} vermieden. Doch kann z.B. nach einer Common Subexpression Elimination weiteres Kopier-Optimierungspotential entstehen.

Damit wäre dieser Abschnitt zu den klassischen Optimierungsverfahren abgeschlossen. Die Verfahren lassen sich so erweitern, daß sie sich für die Optimierung iterativer Programme eignen. In [3, 15] und [17] wird hauptsächlich auf diese Varianten eingegangen; eine Untersuchung formaler Eigenschaften des lokalen Falls lokalen Analyse innerhalb eines Basisblocks (entspricht LC-Programm) findet jedoch nicht statt.

Auch in aktuellen Compilern sind die klassischen Transformationen in erweiterter Form vertreten. So verwendet etwa der Open-Source C-Compiler GCC der „Free Software Foundation“ neben unzähligen anderen Verfahren auch die in diesem Abschnitt vorgestellten in einer lokalen und interprozeduralen Version [11].

Wir wollen uns nun der DAG-Optimierung zuwenden, die auf einem gänzlich anderen Ansatz als die klassischen Verfahren basiert. In Kapitel 3 werden wir dann detailliert auf die Beziehungen zwischen den klassischen Verfahren sowie deren Eigenschaften im Vergleich mit dem DAG-Verfahren eingehen.

2.2 DAG-Optimierung

Die DAG-Optimierung ist ein auf der Konstruktion eines gerichteten azyklischen Graphen basierendes Optimierungsverfahren für LC-Programme. Zunächst einmal die Definition eines DAG (*Directed Acyclic Graph*):

Definition 2.2.1 (DAG): Ein Graph $G = (K, L, lab, suc)$ heißt DAG, wenn

- K eine Menge von Knoten,
- L eine Menge von Markierungen,
- $lab : K \rightarrow L$ eine Markierungsfunktion,
- $suc : \subseteq K \times \mathbb{N} \rightarrow K$ eine partiell definierte⁴ Nachfolgerfunktion ist, und
- $\nexists k \in K$, so daß $\exists k_1, k_2, \dots, k_n \in K$ mit $k_1 = k$, $k_n = k$ und $suc(k_j, i_j) = k_{j+1}$ für $i_j \in \mathbb{N}$, $j \in \{1, \dots, n-1\}$ (G ist *azyklisch*).

Die Idee ist nun, das Ergebnis eines Funktionsausdrucks und seine Argumente durch (verschiedene) Knoten darzustellen und diese dann mithilfe der Nachfolgerfunktion zu verknüpfen.

2.2.1 DAG eines LC-Programms

Der DAG eines LC-Programms ist sozusagen eine graphische Darstellung der Termdarstellung von LC-Programmen mit dem Unterschied, daß gleiche Teilterme nur einmal repräsentiert werden. Ferner wird eine partielle Auswertung von Rechenausdrücken unter Ausnutzung von Konstanteninformationen durchgeführt. Deswegen wird in der Signatur des Zielprogramms die Menge der Konstantensymbole wie bei der Konstantenfaltung modifiziert werden.

Zusätzlich zum DAG werden eine Variablenbelegungsfunktion

$$val : \subseteq V_\pi \times \mathbb{N} \rightarrow K$$

mit $val(x, i) = k$, falls der mit k beginnende Teilgraph den Wert der Variable x nach i Rechenschritten darstellt, und eine Funktion

$$last : V_{out} \rightarrow \mathbb{N},$$

die einer Ausgabevariablen den Index ihrer letzten (finalen) Wertzuweisung zuordnet, benötigt.

Algorithmus 2.2.2 (DAG-Konstruktion): Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\Sigma := (F, C)$, $\beta := \alpha_1; \dots; \alpha_n$ und $\mathfrak{A} := (A, \varphi)$ eine passende Interpretation für Σ . Der DAG D_π von π besteht aus einem DAG $G = (K, L, lab, suc)$, einer Variablenbelegungsfunktion val und einer $last$ -Funktion⁵ (wie oben), d.h. $D_\pi := (G, val, last)$. Es wird $L := F \cup V_{in} \cup A$ gesetzt, um Knoten, die Rechenausdrücke repräsentieren, mit den entsprechenden Funktionssymbolen (und Variablen-/Konstantenknoten mit sich selbst) zu beschriften.

Der Graph G und die Funktionen val und $last$ werden induktiv wie folgt konstruiert:

1. Wähle $K := V_{in} \cup \varphi(C_\pi)$ mit $lab(k) = k \ \forall k \in K$ als Anfangsknoten⁶ und setze $val(x, 0) := x$ für alle $x \in V_{in}$, wobei

$$\varphi(C_\pi) := \{\varphi(c) \mid c \in C_\pi\}$$

2. *Induktionsvoraussetzung:* G , val und $last$ seien bereits bezüglich $\alpha_1; \dots; \alpha_i$ konstruiert, insbesondere gelte $val(x, i) \in K$ für $x \in DV_i$ ⁷.

⁴ $f : \subseteq A \rightarrow B$ bezeichne eine partielle Funktion f von A nach B .

⁵Die Initialisierung von $last$ ist irrelevant, da die Funktion aufgrund der Vollständigkeit von LC-Programmen sowieso für alle Ausgabevariablen geändert wird.

⁶Man könnte die Konstanten auch später „bei Bedarf“ einfügen, wie z.B. in der Implementierung der DAG-Optimierung (siehe Abschnitt 4.2.4).

⁷Die DV_i -Mengen wurden in Definition 1.1.5 vorgestellt.

Es sei $\alpha_{i+1} = x \leftarrow e$. Falls $x \in V_{out}$, wird $last$ wie folgt aktualisiert:

$$last := last[x/i + 1]$$

Ansonsten sind je nach der Struktur des Ausdrucks e unterschiedliche Fälle zu betrachten:

(i) $e = y \in V$

Nach Induktionsvoraussetzung repräsentiert $val(y, i) \in K$ schon den aktuellen Wert von y . Also muß G nicht erweitert werden; setze

$$\begin{aligned} val(x, i + 1) &:= val(y, i) \\ val(x', i + 1) &:= val(x', i) \text{ für } x' \neq x \end{aligned}$$

(ii) $e = c \in C$

Es existiert bereits $a \in K$. Daher findet keine Erweiterung von G statt, also

$$\begin{aligned} val(x, i + 1) &:= \varphi(c) \\ val(x', i + 1) &:= val(x', i) \text{ für } x' \neq x \end{aligned}$$

(iii) $e = f(u_1, \dots, u_r)$, $u_j \in V \cup C$, $f \in F^{(r)}$. Unterscheidung von weiteren Teilfällen:

(a) Für alle $j \in \{1, \dots, r\}$ gilt $u_j \in C$ oder $u_j \in V$ und $val(u_j, i) \in A$.

Es gilt dann $\varphi(f)(u'_1, \dots, u'_r) =: a \in A$ mit

$$u'_j := \begin{cases} \varphi(u_j) & \text{falls } u_j \in C \\ val(u_j, i) & \text{falls } u_j \in V \text{ und } val(u_j, i) \in A \end{cases}$$

(a1) $a \in \varphi(C_\pi)$:⁸ Keine Erweiterung von G .

$$\begin{aligned} val(x, i + 1) &:= a \\ val(x', i + 1) &:= val(x', i) \text{ für } x' \neq x \end{aligned}$$

(a2) $a \notin \varphi(C_\pi)$: Erweiterung von D_π um einen neuen Knoten a .

$$\begin{aligned} K &:= K \cup \{a\} \\ val(x, i + 1) &:= a \\ val(x', i + 1) &:= val(x', i) \text{ für } x' \neq x \end{aligned}$$

(b) Es gibt $j \in \{1, \dots, r\}$ mit $u_j \in V$ und $val(u_j, i) \notin A$.

(b1) $\exists k \in K$ mit $lab(k) = f$ und

$$suc(k, j) = \begin{cases} \varphi(u_j) & \text{falls } u_j \in C \\ val(u_j, i) & \text{falls } u_j \in V \end{cases}$$

Keine Erweiterung von G , da der Wert von e nach i Schritten schon durch k repräsentiert wird, setze also

$$\begin{aligned} val(x, i + 1) &:= k \\ val(x', i + 1) &:= val(x', i) \text{ für } x' \neq x \end{aligned}$$

⁸Es kann sein, daß in der Praxis nicht feststellbar ist, ob die entsprechende Konstante schon vorhanden ist (z.B. bei Fließkommazahlen schwierig). Dann nur Alternative (a2).

(b2) Sonst ($\nexists k \in K$ wie unter (b1)): Füge einen neuen Knoten k_{i+1} ein:

$$\begin{aligned} K &:= K \cup \{k_{i+1}\} \\ \text{lab}(k_{i+1}) &:= f \\ \text{suc}(k_{i+1}, j) &:= \begin{cases} \varphi(u_j) & \text{falls } u_j \in C \\ \text{val}(u_j, i) & \text{falls } u_j \in V \end{cases} \\ \text{val}(x, i+1) &:= k_{i+1} \\ \text{val}(x', i+1) &:= \text{val}(x', i) \text{ für } x' \neq x \end{aligned}$$

Im folgenden bezeichnen wir einen Knoten k als *Funktions-* oder *Operationsknoten*, falls er eine Beschriftung $\text{lab}(k) \in F$ besitzt, also einen Rechenausdruck repräsentiert. Knoten, die für Variablen bzw. Konstanten stehen, nennen wir *Variablen-* bzw. *Konstantenknoten* und Knoten, die keine Nachfolger bzgl. der *suc*-Funktion besitzen, heißen *Blätter*.

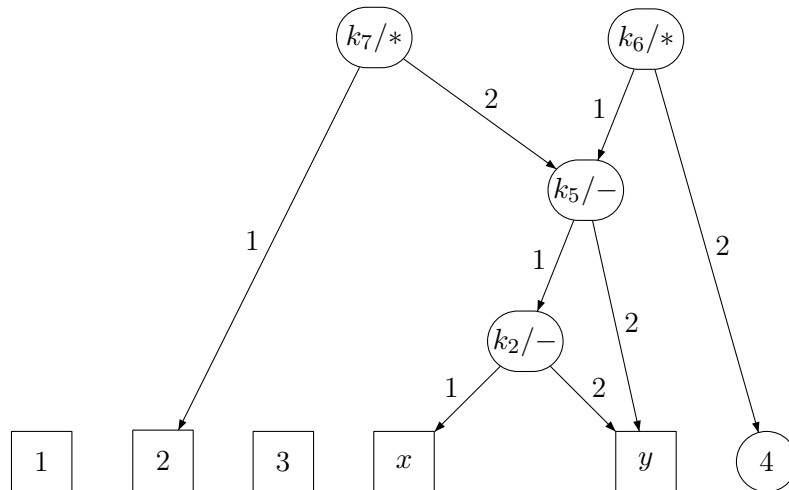
Nun wollen wir uns zur Veranschaulichung der DAG-Konstruktion ein Beispiel ansehen:

Beispiel 2.2.3: In Abbildung 2.2 ist der DAG zum bekannten Programm aus Abbildung 1.1 grafisch dargestellt. Die eckigen Knoten stellen die schon zu Beginn (vor Abarbeitung der Anweisungen) vorhandenen Knoten dar. Die restlichen wurden anhand der Anweisungen gemäß obiger Definition erzeugt.

Die Kanten repräsentieren die *suc*-Funktion, die Kantenummerierung ist für nicht-kommutative Operationen (wie z.B. $-$) zwingend notwendig. In die Tabelle der *val*-Funktion wurden nur Eintragungen vorgenommen, falls sich ein Wert verändert hat.

Betrachten wir einmal den Ablauf der Konstruktion des Graphen. Zunächst werden $V_{in} \cup \varphi(C_\pi)$ als Anfangsknoten in K eingefügt, das sind im Beispiel die Knoten $\{1, 2, 3, x, y\}$. Ferner gilt $\text{val}(x, 0) = x$ und $\text{val}(y, 0) = y$, da $\{x, y\} = V_{in}$. Der DAG D_π wird jetzt wie folgt konstruiert:

1. $\alpha_1 = u \leftarrow 3$: Es ist keine Erweiterung des Graphen erforderlich, da $\varphi(3) = 3 \in K$. Die *val*-Funktion muß um $\text{val}(u, 1) := 3$ ergänzt werden (alle anderen Werte werden von Schritt 0 übertragen) und es wird $\text{last}(u) := 1$ gesetzt.
2. $\alpha_2 = v \leftarrow x - y$: Da $\{\text{val}(x, 1), \text{val}(y, 1)\} \cap A = \emptyset$ und noch gar kein Funktionsknoten vorhanden ist, wird k_2 eingefügt. Er bekommt die Beschriftung $\text{lab}(k_2) := -$. Setze $\text{suc}(k_2, 1) := \text{val}(x, 1) = x$, $\text{suc}(k_2, 2) := \text{val}(y, 1) = y$ und $\text{val}(v, 2) := k_2$ (restliche *val*-Werte werden übertragen), sowie $\text{last}(v) := 2$.
3. $\alpha_3 = w \leftarrow u + 1$: Hier wird eine Funktion auf konstante Argumente angewendet. Es gilt $3 + \varphi(1) = 3 + 1 = 4$. Die Konstante 4 ist jedoch noch nicht im Graphen vorhanden. Daher wird ein neuer Knoten 4 eingefügt und $\text{val}(w, 3) := 4$ gesetzt.
4. $\alpha_4 = x \leftarrow x - y$: Für $x - y$ ist bereits der Knoten k_2 vorhanden, der die Bedingungen in Fall (iii)(b1) des Algorithmus erfüllt. Es wird also kein neuer Knoten erzeugt, sondern $\text{val}(x, 4) := k_2$ gesetzt.
5. $\alpha_5 = u \leftarrow x - y$: Aufgrund der Änderung von x in Anweisung 4 muß ein neuer Knoten k_5 mit $\text{lab}(k_5) = +$, $\text{suc}(k_5, 1) = k_2$ und $\text{suc}(k_5, 2) = y$ angelegt werden (die Argumente von $+$ sind offensichtlich nicht konstant). Ferner wird *val* durch $\text{val}(u, 5) := k_5$ aktualisiert und $\text{last}(u) := 5$ gesetzt.
6. $\alpha_6 = z \leftarrow u * w$: Da u ein nicht konstantes Argument von $*$ ist ($\text{val}(u, 5) = k_5 \notin A$) und bisher noch kein Knoten mit Nachfolgern k_5 und 4 sowie Beschriftung $*$ existiert, wird der Knoten k_6 mit $\text{lab}(k_6) = *$, $\text{suc}(k_6, 1) = k_2$ und $\text{suc}(k_6, 2) = 4$ erzeugt. Die *val*-Funktion wird durch $\text{val}(z, 6) := k_6$ modifiziert.



val-Funktion:

<i>i</i>	<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>
0				<i>x</i>	<i>y</i>	
1	3					
2		k_2				
3			4			
4				k_5		
5	k_5					
6						k_6
7	k_7					
8		3				

$$last = \{u \mapsto 7, v \mapsto 8\}$$

Abbildung 2.2: DAG D_π zum Programm π aus Abbildung 1.1

7. $\alpha_7 = u \leftarrow 2 * u$: Weil $val(u, 6) = k_5 \notin A$ und noch kein passender Knoten mit den entsprechenden Nachfolgern vorhanden ist, wird k_7 mit $lab(k_7) = *$, $suc(k_7, 1) = 2$ und $suc(k_7, 2) = k_5$ eingefügt. Außerdem: $val(u, 7) := k_7$ und $last(u) := 7$;
8. $\alpha_8 = v \leftarrow w - 1$: Hier gilt $val(w, 7) = 4 \in A$ und $\varphi(1) = 1 \in A$. Daher wird $\varphi(-)(4, 1) = 3$ ausgewertet und $val(v, 8) := 3$ gesetzt, da 3 bereits ein Knoten in K ist. Abschließend muß noch $last(v) := 8$ gesetzt werden.

Die DAG-Konstruktion beinhaltet bereits Teile der klassischen Transformationen Common Subexpression Elimination, Konstantenfaltung und Kopierpropagation:

- Es wird kein Operationsknoten für einen Rechenausdruck angelegt, für den ein Knoten gleicher Form bereits vorhanden ist, sondern mithilfe der val -Funktion nur ein „Zeiger“ auf den äquivalenten Knoten gesetzt.
- Konstante Ausdrücke werden soweit wie möglich ausgewertet. Entweder ergibt sich für den gesamten Ausdruck ein konstanter Wert (falls alle Argumente entweder Konstanten oder mit konstanten Werten belegte Variablen sind), oder aber es werden die im Ausdruck vorhandenen Variablen soweit wie möglich durch Konstanten ersetzt. Daher repräsentiert kein Knoten einen konstanten Operationsterm.
- Durch die val -Funktion werden auch „Kopierketten“ aufgelöst, denn bei einer Kopieranweisung wird der val -Eintrag der Variable der linken Seite auf denjenigen der Variable der rechten Seite gesetzt. Dadurch wird die Indirektion aufgelöst.

Die Entfernung von Dead Code wird hingegen erst bei der Programmtransformation berücksichtigt. In Abschnitt 3.1 werden wir genauer untersuchen, ob die im DAG-Verfahren enthaltenen Teile der klassischen Verfahren gleich mächtig wie diese sind, d.h. ob ein DAG-optimiertes Programm optimal bzgl. den klassischen Transformationen ist.

2.2.2 Codegenerierung aus einem DAG

Dieser Abschnitt behandelt die optimierende Programmtransformation. Eigentlich handelt es sich dabei nicht wirklich um eine Transformation, sondern um eine Codegenerierung aus dem DAG, denn das Ausgangsprogramm π wird nicht mehr benötigt; sein DAG D_π allein ist ausreichend.

Die den Ausgabevariablen zugeordneten Knoten induzieren den für die DAG-Codegenerierung benötigten Teilgraphen, der nur aus *ausgaberelevanten* Knoten besteht.

Definition 2.2.4 (Ausgaberelevante Knoten): Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta = \alpha_1; \dots; \alpha_n$ ein LC-Programm und $D_\pi = (G, val, last)$ sein DAG mit $G = (K, L, lab, suc)$. Ein Knoten $k \in K$ heißt ausgaberelevant, falls

1. ein $y \in V_{out}$ existiert, so daß $val(y, n) = k$, oder
2. ein ausgaberelevanter Knoten $k' \in K$ und ein $i \in \mathbb{N}$ existiert, so daß $suc(k', i) = k$.

Ausgaberelevant sind also diejenigen Knoten, die von einem Knoten, dem eine Ausgabevariablen zugeordnet ist, aus erreichbar sind. Die restlichen Knoten sind überflüssig und werden bei der Codegenerierung nicht berücksichtigt. Hierdurch wird die Elimination von Dead Code realisiert.

Für die Erzeugung des Ergebnisprogramms muß eine Bearbeitungsreihenfolge der Knoten des DAG festgelegt werden. Um für einen Operationsknoten Code generieren zu können, müssen vorher zuerst alle seine Nachfolger abgearbeitet werden, da sie Eingabewerte für die Berechnung darstellen. Es liegt daher nahe sich im Graphen „von unten nach oben hochzuarbeiten“. Für die Blätter sind dabei keine

zusatlichen Instruktionen notwendig, sie sind sofort verfugbar, da sie nur aus Eingabevariablen und Konstanten bestehen, die direkt in Rechenausdrucken verwendet werden konnen.

Wir wollen die Knotenreihenfolge im folgenden fest vorgeben. Dies dient zum einen zur Beseitigung von Nichtdeterminismus, als auch dazu, den Code in der Reihenfolge des Eingabeprogramms zu generieren. D.h. wenn im Eingabeprogramm eine Anweisung $x \leftarrow y+z$ vor einer Anweisung $v \leftarrow x*x$ steht, dann soll auch zuerst Code fur den Ausdruck $y+z$, der im DAG graphisch reprasentiert ist, generiert werden.

Diese Reihenfolge ist aufgrund der Vollstandigkeit der LC-Programme (Def. 1.1.5) immer gultig, d.h. man wird automatisch immer alle Sohnknoten abgearbeitet haben, bevor man Code fur den Vaterknoten generiert.

Algorithmus 2.2.5 ($T_{DAG} : \mathcal{LC} \rightarrow \mathcal{LC}$): Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\Sigma := (F, C)$ und $\beta := \alpha_1; \dots; \alpha_n$. Sei $\mathfrak{A} := (A, \varphi)$ eine Interpretation fur Σ und $D_\pi = (G, val, last)$ der DAG von π mit $G := (K, L, lab, suc)$, wobei K o.B.d.A. nur ausgaberelevante Knoten enthalte. Auerdem wird eine Substitutionsfunktion $\delta : K \rightarrow V_{out} \cup K$ und eine Zahlvariable $i \in \mathbb{N}$ mit Initialisierung $i := 0$ benotigt. Erzeuge das LC-Programm $T_{DAG}(\pi) := ((F, A), \vec{v}_{in}, \vec{v}_{out}, \beta')$ wie folgt (in etwas kleinerer Schrift sind Erlauterungen zu den einzelnen Schritten angegeben):

1. Setze $\beta' := \varepsilon$, $\delta := id$, $K_f := \{k \in K \mid k \text{ ist ein Blatt}\}$ und $V_{out}^f := \emptyset$.

Beginn mit der leeren Anweisungsfolge und der identischen Substitution. K_f gibt die bereits abgearbeiteten Knoten an. Zu Beginn sind das die Blatter, da diese sofort verfugbar sind, ohne da explizit Code fur sie generiert wird. V_{out}^f ist die Menge der Ausgabevariablen, denen bereits ein finaler Wert zugewiesen wurde. Diese werden nicht mehr fur neue Zuweisungen benutzt.

2. Sei $K' := \{k \in K \setminus K_f \mid \exists i \in \mathbb{N} \text{ mit } suc(k, i) \in K \setminus K_f\}$. Fallunterscheidung:

- a) Falls $K' \neq \emptyset$: Wahle einen Knoten $k_l \in K'$ mit $\forall k_j \in K' : l \leq j$ und inkrementiere $i := i + 1$.

- b) Sonst: Weiter bei Schritt 8.

K' bezeichnet die Menge der noch nicht bearbeiteten Knoten aus K , deren Nachfolger aber bereits abgearbeitet wurden. Diese mussen Operationsknoten sein, da nur die Blatter keine Operationsknoten sind und diese bereits zu Anfang in K_f eingefugt wurden. Der Knoten k_l wird so gewahlt, da er den kleinsten Index aller Knoten in K' hat. Dies dient der Einhaltung der Berechnungsreihenfolge des Eingabeprogramms (denn nach Algorithmus 2.2.2 ist l der Index der Anweisung, die zur Generierung von k_l fuhrte). Zudem wird dadurch der Determinismus gewahrleistet, denn sonst gabe es u.U. mehrere Wahlmoglichkeiten, die unterschiedliche Ergebnisse implizieren wurden.

3. Sei $V_{out}^{<l} := \{y \in V_{out} \setminus V_{out}^f \mid last(y) < l\} =: \{y_{i_1}, \dots, y_{i_q}\}$ mit $last(y_{i_j}) < last(y_{i_k})$ fur $j < k$. Setze $V_{out}^f := V_{out}^f \cup V_{out}^{<l}$ sowie $i := i + q$ und fuge fur $j = 1, \dots, q$ jeweils eine neue Anweisung ein:

$$\beta' := \beta' \cdot y_{i_j} \leftarrow \delta(val(y_{i_j}, n))$$

$V_{out}^{<l}$ ist die Menge an Ausgabevariablen, denen noch kein finaler Wert zugewiesen wurde, deren letzte Zuweisung im Ausgangsprogramm aber unterhalb der Nummer des aktuellen Knotens liegt. Diese mussen nun *vor* der Codegenerierung fur k_l mit Werten belegt werden, um die Zuweisungsreihenfolge des Ausgangsprogramms einzuhalten.

Die Reihenfolge der Zuweisungen erfolgt analog zum Ausgangsprogramm, denn die y_{i_j} sind nach *last*-Wert sortiert. Der Menge der mit finalen Werten belegten Ausgabevariablen wird auerdem die Menge $V_{out}^{<l}$ hinzugefugt, da auch diese jetzt definitionsgema ihren finalen Wert besitzen.

Die Zahlvariable wird um q erhohet, weil sie bei der Umbenennung in Schritt 5 genau die Nummer der

einzufügenden Anweisung repräsentieren soll und q Anweisungen „dazwischen geschoben“ werden.

4. Sei $V_{out}^{k_l} := \{y \in V_{out} \mid val(y, n) = k_l\}$. Falls $V_{out}^{k_l} \neq \emptyset$ wähle $y \in V_{out}^{k_l}$, so daß $\forall y' \in V_{out}^{k_l} : last(y) \leq last(y')$ und setze:

$$\begin{aligned}\delta &:= \delta[k_l/y] \\ V_{out}^f &:= V_{out}^f \cup \{y\}\end{aligned}$$

$V_{out}^{k_l}$ ist die Menge der dem Knoten k_l zugeordneten Ausgabevariablen. Daraus wird eine ausgewählt; falls es mehrere gibt, wird diejenige mit dem niedrigsten $last$ -Wert selektiert (Determinismus, Zuweisungsreihenfolge des Ausgangsprogramms). Dann wird die Substitution δ so aktualisiert, daß der aktuelle Knoten k_l durch δ in die neu gewählte Ausgabevariable y umbenannt wird. Zudem wird y in die Menge der „benutzten“ Ausgabevariablen V_{out}^f eingefügt.

5. Falls $\delta(k_l) \notin V_{out}$ setze $\delta := \delta[k_l/v_i]$.

Falls k_l keine Ausgabevariable zugeordnet ist, wird der Knoten in eine Variable v_i umbenannt. Man könnte auch den Knotennamen selbst als Variable verwenden, was aber dazu führen würde, daß das Verfahren wegen der Variablenbenennung nicht idempotent ist (im Ausgangsprogramm ist evtl. noch Dead Code vorhanden und daher liegen die Knotennummern weiter auseinander).

6. Es sei $lab(k_l) = f \in F^{(r)}$. Erweitere β' :

$$\beta' := \beta' \cdot \delta(k_l \leftarrow f(\delta(suc(k_l, 1)), \delta(suc(k_l, 2)), \dots, \delta(suc(k_l, r))));$$

Einfügen einer neuen Operationsanweisung für den gewählten Knoten k_l . Dabei werden k_l selbst, sowie alle Nachfolgerknoten, die als Argumente von f fungieren, gemäß δ umbenannt.

7. Setze $K_f := K_f \cup \{k_l\}$. Weiter mit Schritt 2.

Der Knoten k_l ist abgearbeitet und wird dementsprechend der Menge K_f hinzugefügt.

8. Sei $V_{out}^{<\infty} := V_{out} \setminus V_{out}^f =: \{y_{i_1}, \dots, y_{i_q}\}$ mit $last(y_{i_j}) < last(y_{i_k})$ für $j < k$. Für $j = 1, \dots, q$ setze:

$$\beta' := \beta' \cdot y_{i_j} \leftarrow \delta(val(y_{i_j}, n));$$

Nun müssen noch die bisher nicht berücksichtigten Ausgabevariablen (genauer: diejenigen, die noch keinen finalen Wert besitzen) mit ihren Werten gemäß val -Funktion belegt werden.

Der vom Algorithmus erzeugte Code liegt in der sogenannten *Static Single Assignment-Form* vor:

Definition 2.2.6 (Static Single Assignment-Form): Ein Programm $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$ und $\alpha_i := x_i \leftarrow e_i$ liegt in Static Single Assignment-Form (SSA-Form) vor, wenn für alle $i \in \{1, \dots, n\}$ gilt:

$$x_i \notin V_i \cup V_{in} \cup \bigcup_{k=1}^{i-1} V_{\alpha_k} \cup \bigcup_{k=i+1}^n \{x_k\}$$

Das bedeutet also, daß für jede Anweisung α eine neue, bisher unbenutzte/undefinierte Zuweisungsvariable verwendet wird, die auch nicht auf der rechten Seite von α selbst auftritt.

Dies ist natürlich bezüglich Speichereffizienz nicht unbedingt wünschenswert, allerdings nur schwer anders realisierbar. So würde z.B. eine Substitution von k_l durch x_l anstatt v_i in Schritt 5 des Algorithmus zwangsläufig zu Problemen führen, da u.U. auf vorherige Werte von Variablen zugegriffen werden müßte, die jedoch schon überschrieben wurden.

$$\begin{aligned}
\vec{v}_{in} &: (x, y) \\
\beta' &: v_1 \leftarrow x - y; \\
&v_2 \leftarrow v_1 - y; \\
&u \leftarrow 2 * v_2; \\
&v \leftarrow 3; \\
\vec{v}_{out} &: (u, v)
\end{aligned}$$

Abbildung 2.3: DAG-optimierte Version $T_{DAG}(\pi) =: \pi'$ des LC-Programms π aus Abbildung 1.1

Beispiel 2.2.7: Wenden wir den Algorithmus 2.2.5 einmal auf den DAG aus Abbildung 2.2 an. Dafür müssen zunächst die ausgaberelevanten Knoten bestimmt werden. Man sieht leicht, daß dies die Knoten $\{x, y, 2, 3, k_2, k_5, k_7\} =: K$ sind (da $val(u, 8) = k_7$) und $val(v, 8) = 3$). Nun kann der Algorithmus eingesetzt werden (links die Nummer des jeweiligen Schrittes):

1. $\beta' := \varepsilon, \delta := id, K_f := \{x, y, 2, 3\}, V_{out}^f := \emptyset, i := 0$.
2. $K' = \{k_2\} \Rightarrow l = 2$ und setze $i := i + 1 = 1$.
3. $V_{out}^{<2} = \emptyset \Rightarrow$ nichts weiter zu tun.
4. Da $V_{out}^{k_2} = \emptyset$, wird δ nicht verändert.
5. Da $\delta(k_2) = k_2 \notin V_{out}$, setze $\delta := \delta[k_2/v_1]$.
6. $\beta' := \beta' \cdot \delta(k_2) \leftarrow \delta(x) - \delta(y); = v_1 \leftarrow x - y$;
7. $K_f := K_f \cup \{k_2\} = \{x, y, 2, 3, k_2\}$.
2. $K' = \{k_5\} \Rightarrow l = 5, i := i + 1 = 2$.
3. $V_{out}^{<5} = \emptyset \Rightarrow$ nichts weiter zu tun.
4. Da $V_{out}^{k_5} = \emptyset$, bleibt δ unverändert.
5. $\delta(k_5) = k_5 \notin V_{out} \Rightarrow \delta := \delta[k_5/v_2]$.
6. $\beta' := \beta' \cdot \delta(k_5) \leftarrow \delta(k_2) - \delta(y); = \beta' \cdot v_2 \leftarrow v_1 - y$;
7. $K_f := K_f \cup \{k_5\} = \{x, y, 2, 3, k_2, k_5\}$
2. $K' = \{k_7\} \Rightarrow l = 7, i := i + 1 = 3$.
3. $V_{out}^{<7} = \emptyset \Rightarrow$ nichts weiter zu tun.
4. $V_{out}^{k_7} = \{u\}$, setze daher $\delta := \delta[k_7/u]$ (kein Vergleich der *last*-Einträge nötig, da nur ein Element).
5. Weil $\delta(k_7) = u \in V_{out}$, bleibt δ unverändert.
6. $\beta' := \beta' \cdot \delta(k_7) \leftarrow \delta(2) * \delta(k_5); = \beta' \cdot u \leftarrow 2 * v_2$;
7. $K_f := K_f \cup \{k_7\} = K$.
2. Sprung zu Schritt 8, da $K \setminus K^f = \emptyset$.
8. $V_{out}^{<\infty} = \{v\}$, also: $\beta' := \beta' \cdot v \leftarrow \delta(val(v, 8)); = \beta' \cdot v \leftarrow 3$;

Insgesamt ergibt sich also das stark verkürzte Programm aus Abbildung 2.3.

Da T_{DAG} wie die Konstantenfaltung eine partielle Auswertung von Rechenausdrücken durchführt, erhält man keine stark äquivalenten Programme. Zudem muß berücksichtigt werden, daß im DAG (und daher im Zielprogramm) alle Konstantensymbole bereits durch Konstanten aus der Wertemenge der entsprechenden Interpretation ersetzt worden sind.

Bei einer begrenzten Verfügbarkeit von Registern, kann je nach Knotenauswahlstrategie die Effizienz des Ergebnisprogramms variieren. Die optimale (kostenminimale) Codegenerierung aus einem ist dann DAG NP-vollständig [7]. Die Codegenerierung aus Bäumen, die Ausdrücke darstellen, ist jedoch auch in diesem Fall effizient möglich [8].

2.2.3 Korrektheit der DAG-Optimierung

Wie für die klassischen Optimierungen wollen wir auch die Korrektheit des DAG-Verfahrens beweisen. Dazu ist allerdings aufgrund der relativ komplexen Struktur ein höherer Aufwand vonnöten und der Beweis wird schrittweise erfolgen und größtenteils auf Termen arbeiten. Dafür müssen aus dem DAG eines LC-Programms Terme, die den DAG charakterisieren, generiert werden.

Definition 2.2.8 (Termdarstellung eines DAG): Sei $D_\pi = ((K, L, lab, suc), val, last)$ der DAG eines LC-Programms $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\Sigma := (F, C)$, $\beta := \alpha_1; \dots; \alpha_n$ und $\mathfrak{A} := (A, \varphi)$ eine Interpretation von Σ . Dann ist die Termdarstellung $t_{D_\pi}(y) \in T_\Sigma(V_{in})$ von D_π bezüglich der Ausgabevariablen $y \in V_{out}$ definiert als:

$$t_{D_\pi}(y) := t_{val(y,n)}$$

$$\text{und für } k \in K : t_k := \begin{cases} k & \text{falls } k \in A \cup V_{in} \\ f(t_{suc(k,1)}, \dots, t_{suc(k,r)}) & \text{falls } lab(k) = f \in F^{(r)} \end{cases}$$

Die Definition entspricht der intuitiven Vorstellung, d.h. alle Nachfolger eines Knotens stellen Teilterme des entsprechenden Knotenterms dar. Die durch das Zusammenfassen von Knoten, die die gleichen Ausdrücke berechnen, erhaltene Optimierung geht hier natürlich wieder verloren, denn wenn man den erhaltenen Term grafisch darstellt, erhält man einen Baum. Für die Korrektheit ist dies aber nicht von Bedeutung.

Beispiel 2.2.9: Am Beispiel des DAG aus Abbildung 2.2 und der Ausgabevariablen u ergibt sich:

$$\begin{aligned} t_{D_\pi}(u) &= t_{val(u,8)} \\ &= t_{k_7} \\ &= *(t_{suc(k_7,1)}, t_{suc(k_7,2)}) \\ &= *(t_2, t_{k_5}) \\ &= *(2, -(t_{suc(k_5,1)}, t_{suc(k_5,2)})) \\ &= *(2, -(t_{k_2}, t_y)) \\ &= *(2, -(-(t_{suc(k_2,1)}, t_{suc(k_2,2)}), y)) \\ &= *(2, -(-(t_x, t_y), y)) \\ &= *(2, -(-(x, y), y)) \end{aligned}$$

In diesem Beispiel ist die Termdarstellung des DAG identisch zur Termdarstellung des LC-Programms bezüglich u . Für v wäre dies allerdings nicht der Fall, da bei der DAG-Konstruktion eine Faltung von Konstanten vorgenommen wird. Es ergibt sich $t_{D_\pi}(v) = 3 \neq -(+(3, 1), 1) = t_\pi(v)$.

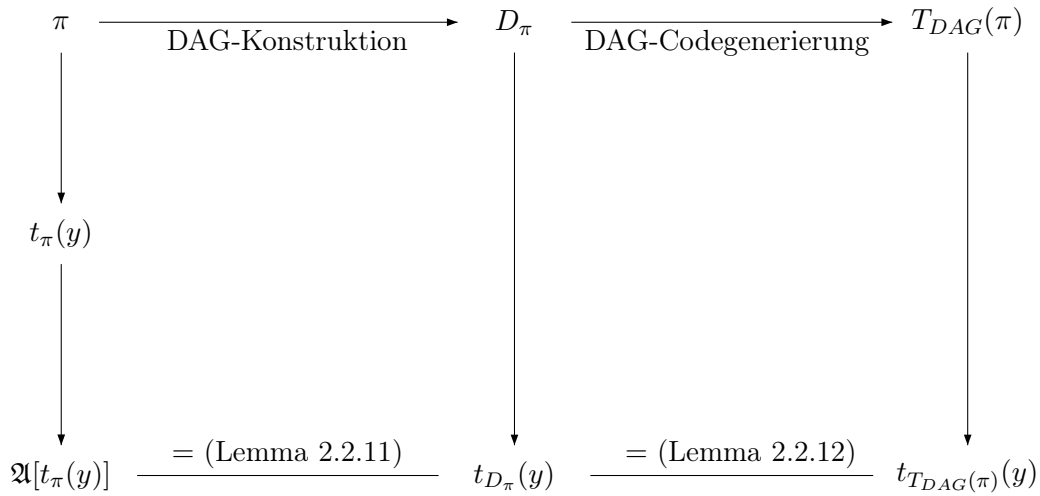


Abbildung 2.4: Struktur des Korrektheitsbeweises der DAG-Optimierung

Eine Angleichung der Terme aus einem LC-Programm und seinem DAG (jeweils bzgl. der gleichen Ausgabevariable) ist durch die Auswertung von konstanten Teiltermen (d.h. Grundtermen) zu erreichen.

Definition 2.2.10 (Partielle Auswertung von Termen): Sei $t \in T_\Sigma(X)$ mit $\Sigma := (F, C)$ und $\mathfrak{A} := (A, \varphi)$ eine Interpretation für Σ . Dann bezeichnet $\mathfrak{A}[t] \in T_{(F,A)}(X)$ den durch *partielle Auswertung* von t entstehenden Term. Dabei gilt für $c \in C$, $x \in X$, $f \in F^{(r)}$ und $t_1, \dots, t_n \in T_{(F,A)}(X)$:

$$\begin{aligned}
\mathfrak{A}[c] &:= \varphi(c) \\
\mathfrak{A}[x] &:= x \\
\mathfrak{A}[f(t_1, \dots, t_r)] &:= \begin{cases} \varphi(f)(\mathfrak{A}[t_1], \dots, \mathfrak{A}[t_r]) & \text{falls } \forall i \in \{1, \dots, r\} : \mathfrak{A}[t_i] \in A \\ f(\mathfrak{A}[t_1], \dots, \mathfrak{A}[t_r]) & \text{sonst} \end{cases}
\end{aligned}$$

Der Beweis zur Korrektheit der DAG-Optimierung läuft in zwei Hauptschritten ab: Zunächst zeigen wir die Korrektheit der DAG-Konstruktion, um dann, die Korrektheit der DAG-Konstruktion voraussetzend, zu zeigen, daß die Codegenerierung aus dem DAG korrekt ist. Anstatt aber einen direkten Beweis zu verwenden, werden wir mit den Termdarstellungen arbeiten, wie in Abbildung 2.4 grafisch dargestellt ist.

Danach ist es lediglich noch nötig, zu zeigen, daß aus der Termgleichheit auch die Gleichheit der Semantiken folgt.

Lemma 2.2.11 (Korrektheit der DAG-Konstruktion): Für $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$, seinen DAG D_π und eine Interpretation \mathfrak{A} für Σ gilt

$$\mathfrak{A}[t_\pi(y)] = t_{D_\pi}(y) \text{ für alle } y \in V_{out}.$$

Beweis: Sei $D_\pi := (G, val, last)$ mit $G := (K, L, lab, suc)$ und $\beta := \alpha_1; \dots; \alpha_n$ mit $\alpha_i := x_i \leftarrow e_i$. Außerdem gelte $\Sigma := (F, C)$ und $\mathfrak{A} := (A, \varphi)$. Induktion über den Termaufbau (für $y \in V_{out}$):

1. $\mathfrak{A}[t_\pi(y)] = a \in A$. Induktive Fallunterscheidung:

a) In π existiert eine Anweisung $\alpha_k = y \leftarrow c$ mit $c \in C$ und $y \notin \bigcup_{i=k+1}^n \{x_i\}$. Dann gilt

$$\text{val}(y, n) = \text{val}(y, k) = \varphi(c)$$

b) Es gibt eine Anweisung $\alpha_l = y \leftarrow z$ in π mit $y \notin \bigcup_{i=l+1}^n \{x_i\}$ und für z gilt wieder einer der Fälle (a), (b) oder (c) (Abbruch bei Fall (a) oder (c)(i)) mit $n = l - 1$ und $y = z$. Es ergibt sich:

$$\text{val}(y, n) = \text{val}(y, l) = \text{val}(z, l - 1)$$

c) Eine Anweisung $\alpha_m = y \leftarrow f(u_1, \dots, u_r)$ mit $f \in F^{(r)}$ existiert in π mit $y \notin \bigcup_{i=l+1}^n \{x_i\}$. Weitere Fallunterscheidung:

(i) $\forall j \in \{1, \dots, r\}$ gilt $u_j \in C$. Dann erhalten wir:

$$\text{val}(y, n) = \text{val}(y, m) = \varphi(f)(\varphi(u_1), \dots, \varphi(u_r))$$

(ii) $\exists j \in \{1, \dots, r\}$ gilt $u_j \in V$ und für u_j gilt entweder (a), (b) oder (c) mit $n = m - 1$ und $y = u_j$. Resultat:

$$\text{val}(y, n) = \text{val}(y, m) = \varphi(f)(u'_1, \dots, u'_r) \text{ mit } u'_j := \begin{cases} \text{val}(u_j, m - 1) & \text{falls } u_j \in V \\ \varphi(u_j) & \text{falls } u_j \in C \end{cases}$$

Dabei muß aufgrund des Induktionsprinzips $\text{val}(u_j, m - 1) \in A$ für $u_j \in V$ gelten.

Man erhält $\text{val}(y, n) = b \in A$ und daher $t_{\text{val}(y, n)} = b$ (da $n < \infty$ und ein Abbruch nur in den Teilfällen (a) und (c)(i) möglich ist). Es muß $b = a$ gelten, da in obiger Fallunterscheidung die gleichen Teilterme ausgewertet wurden wie in $\mathfrak{A}[t_\pi(y)]$, mit dem einzigen Unterschied, daß Auswertungen sofort erfolgt sind und nicht erst nach Konstruktion eines vollständigen Terms. Es gilt also insgesamt:

$$\mathfrak{A}[t_\pi(y)] = a = t_{D_\pi}(y)$$

In Abbildung 2.5 ist zur Veranschaulichung eine „Instanziierung“ dieses ersten Falls anhand eines Beispielprogrammausschnitts dargestellt.

2. $\mathfrak{A}[t_\pi(y)] = x \in V_{in}$. Induktive Unterscheidung der Möglichkeiten ähnlich wie im ersten Fall:

a) Es gibt eine Anweisung $\alpha_k = y \leftarrow z$ in π mit $y \notin \bigcup_{i=k+1}^n \{x_i\}$, $z \neq x$ und für z gilt entweder wieder (a) (Induktion: Kopierkette) oder aber (b) (Abbruch) mit jeweils $n = k - 1$ und $y = z$. Wir erhalten:

$$\text{val}(y, n) = \text{val}(y, k) = \text{val}(z, k - 1)$$

b) In π gilt $y = x$ und $y \notin \bigcup_{i=1}^n \{x_i\}$. Dieser Fall kann wegen $V_{in} \cap V_{out} = \emptyset$ nur eintreten, nachdem *mindestens einmal* Fall (a) eingetreten ist. Es gilt:

$$\text{val}(y, n) = \text{val}(y, 0) = \text{val}(x, 0)$$

Insgesamt ist dann $\text{val}(y, n) = x$ und es gilt $t_{\text{val}(y, n)} = x$. Deshalb folgt:

$$\mathfrak{A}[t_\pi(y)] = x = t_{D_\pi}(y)$$

3. $\mathfrak{A}[t_\pi(y)] = f(t_1, \dots, t_r)$ mit $t_j \in T_{(F,A)}(V_{in})$ und $f \in F^{(r)}$.

Induktionsannahme: Für alle $j \in \{1, \dots, r\}$ gilt bereits

$$t_j = \begin{cases} t_{val(u_j, l-1)} & \text{für } u_j \in V \\ t_{\varphi(u_j)} & \text{für } u_j \in C \end{cases}$$

In π muß eine Anweisung $\alpha_l = y \leftarrow f(u_1, \dots, u_r)$ existieren, wobei $y \notin \bigcup_{i=l+1}^n \{x_i\}$ (y nach α_l unverändert).

Fallunterscheidung:

- a) $\forall j \in \{1, \dots, r\} : t_j \in T_{(F,A)}$
 Dieser Fall kann nicht eintreten, da sonst $\mathfrak{A}[f(t_1, \dots, t_r)]$ eine Konstante wäre.
- b) $\exists j \in \{1, \dots, r\} : t_j \notin T_{(F,A)}$
 Es gibt einen Knoten $k \in K$ mit

$$\begin{aligned} lab(k) &= f \\ \forall j \in \{1, \dots, r\} : suc(k, j) &= \begin{cases} val(u_j, l-1) & \text{falls } u_j \in V \\ \varphi(u_j) & \text{falls } u_j \in C \end{cases} \\ val(y, n) &= k \end{aligned}$$

Dann folgt:

$$\begin{aligned} t_{D_\pi}(y) &= t_{val(y, n)} \\ &= t_k \\ &\stackrel{\text{Def.}}{=} f(t_{suc(k, 1)}, \dots, t_{suc(k, r)}) \\ &\stackrel{\text{IA}}{=} f(t_1, \dots, t_r) \\ &= \mathfrak{A}[t_\pi(y)] \end{aligned}$$

Damit gilt für alle $y \in V_{out}$ die Behauptung. □

Nun ist die Korrektheit von Algorithmus 2.2.5 zu zeigen. Auch hier basiert der Beweis auf den jeweiligen Termdarstellungen.

Lemma 2.2.12 (Korrektheit der DAG-Codegenerierung): Für $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ und seinen DAG D_π gilt

$$t_{D_\pi}(y) = t_{T_{DAG}(\pi)}(y) \text{ für alle } y \in V_{out}.$$

Beweis: π, D_π, Σ und \mathfrak{A} seien wie im Beweis von Lemma 2.2.11 definiert. Außerdem gelte $T_{DAG}(\pi) = ((F, A), \vec{v}_{in}, \vec{v}_{out}, \beta')$ mit $\beta' := \alpha'_1; \dots; \alpha'_m$ und $\alpha'_i := x'_i \leftarrow e'_i$. Induktion über den Termaufbau ($y \in V_{out}$):

1. $t_{D_\pi}(y) = a \in A$:

Es gilt also $val(y, n) = a$ in D_π . Dann wird gemäß Schritt 3 oder 8 von Algorithmus 2.2.5 eine Zuweisung $\alpha'_l = y \leftarrow a$ erzeugt und danach wird y in $T_{DAG}(\pi)$ nicht mehr verändert, d.h. $y \notin \bigcup_{i=l+1}^m \{x'_i\}$ (dies wird durch die Menge V_{out}^f garantiert, in die y nach der Erzeugung von

Ausschnitt aus π :

$$\begin{array}{llll}
\vdots & & & \\
\alpha_k : & x \leftarrow a & \text{Teilfall (a)} & x \notin \bigcup_{i=k+1}^{m_2-1} \{x_i\} \\
\vdots & & & \\
\alpha_{m_1} : & y \leftarrow f(b) & \text{Teilfall (c)(i)} & y \notin \bigcup_{i=m_1+1}^{m_2-1} \{x_i\} \\
\vdots & & & \\
\alpha_{m_2} : & z \leftarrow g(a, x, y) & \text{Teilfall (c)(ii)} & z \notin \bigcup_{i=m_2+1}^{l-1} \{x_i\} \\
\vdots & & & \\
\alpha_l : & y \leftarrow z & \text{Teilfall (b)} & y \notin \bigcup_{i=l+1}^n \{x_i\} \\
\vdots & & &
\end{array}$$

Berechnung von $val(y, n)$:

$$\begin{aligned}
val(x, m_2 - 1) &= val(x, k) \\
&= \varphi(a) \\
val(y, m_2 - 1) &= val(y, m_1) \\
&= \varphi(f)(\varphi(b)) \\
val(z, l - 1) &= val(z, m_2) \\
&= \varphi(g)(\varphi(a), val(x, m_2 - 1), val(y, m_2 - 1)) \\
&= \varphi(g)(\varphi(a), \varphi(a), \varphi(f)(\varphi(b))) \\
val(y, n) &= val(y, l) \\
&= val(z, l - 1) \\
&= \varphi(g)(\varphi(a), \varphi(a), \varphi(f)(\varphi(b)))
\end{aligned}$$

Abbildung 2.5: „Beispielinstanziierung“ des ersten Falls im Beweis von Lemma 2.2.11

α'_i eingefügt wird). Es ergibt somit für die Termdarstellung von $T_{DAG}(\pi)$:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t_{T_{DAG}(\pi)}^{(0)}(y) \\
&= y[x'_m/e'_m] \dots [x'_i/e'_i][x'_{i-1}/e'_{i-1}] \dots [x'_1/e'_1] \\
&= y[x'_m/e'_m] \dots [y/a][x'_{i-1}/e'_{i-1}] \dots [x'_1/e'_1] \\
&= y[y/a][x'_{i-1}/e'_{i-1}] \dots [x'_1/e'_1] && (\text{da } x'_i \neq y \text{ für alle } i > l) \\
&= a[x'_{i-1}/e'_{i-1}] \dots [x'_1/e'_1] \\
&= a && (\text{da } a \text{ keine Variablen enthält})
\end{aligned}$$

2. $t_{D_\pi}(y) = x \in V_{in}$:

Dann gilt $val(y, n) = x$ in D_π . Ähnlich wie im ersten Fall wird nach Schritt 3 oder 8 des Algorithmus eine Zuweisung $\alpha'_i = y \leftarrow x$ erzeugt und $y \notin \bigcup_{i=l+1}^m \{x'_i\}$, sowie $x \notin \bigcup_{i=1}^m \{x'_i\}$, da

$\bigcup_{i=1}^m \{x'_i\} \subseteq V_{out} \cup \{v_j | j \in \mathbb{N}\}$. Man erhält somit:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t_{T_{DAG}(\pi)}^{(0)}(y) \\
&= y[x'_m/e'_m] \dots [x'_l/e'_l][x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= y[x'_m/e'_m] \dots [y/x][x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= y[y/x][x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] && \text{(da } x'_i \neq y \text{ für alle } i > l) \\
&= x[x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= x && \text{(da } x'_i \neq x \text{ für alle } i < l)
\end{aligned}$$

Hinweis: Wenn nach Definition 1.1.3 nicht $V_{in} \cap V_{out} = \emptyset$ gefordert wäre, könnte nicht sichergestellt werden, daß x nicht doch oberhalb von α_l ein Wert zugewiesen wird. Ein Gegenbeispiel zur Korrektheit des DAG-Verfahrens wäre dann das folgende Programm:

$$\begin{aligned}
\pi := ((\emptyset, \{0\}), (x), (x, y), \beta) \notin \mathcal{LC} \text{ mit } \beta := & t \leftarrow x; \\
& x \leftarrow 0; \\
& y \leftarrow z;
\end{aligned}$$

Der Leser möge durch Anwendung des DAG-Verfahrens auf π selbst herausfinden, warum dem so ist.

3. $t_{D_\pi}(y) = f(t_1, \dots, t_r)$ mit $t_i \in T_{(F,A)}(V_{in})$ und $f \in F^{(r)}$:
Dann existiert ein $i \in \{1, \dots, r\}$, so daß $t_i \notin T_{(F,A)}$ (denn sonst wäre der Term durch eine Konstante ersetzt worden). Außerdem gibt es in D_π einen Knoten $k \in K$ mit $lab(k) = f$ und $val(y, n) = k$. Wegen Schritt 6 des Algorithmus 2.2.5 gibt es in β' eine Anweisung

$$\alpha'_i = \delta(k) \leftarrow f(\delta(suc(k, 1)), \delta(suc(k, 2)), \dots, \delta(suc(k, r))),$$

wobei $\delta(k) \in V_{out} \cup \{v_j | j \in \mathbb{N}\}$. Sei π_k das Programm, das man erhält, wenn man den Algorithmus auf den DAG anwendet, der nur alle von k aus erreichbaren Knoten enthält, d.h. die durch k eingeschränkte Knotenmenge K_k sei definiert durch

$$\begin{aligned}
k &\in K_k \\
k' &\in K_k, \text{ falls } \exists i \in \mathbb{N}, \exists k'' \in K_k, \text{ so daß } suc(k'', i) = k'
\end{aligned}$$

Induktionsannahme: Für alle $j \in \{1, \dots, r\}$ gilt bereits $t_j = t_{\pi_{suc(k, j)}}(\delta(suc(k, j)))$.

Es muß $y \in V_{out}^k$, d.h. $val(y, n) = k$ gelten. Fallunterscheidung:

- a) Für alle $y' \in V_{out}^k$: $last(y) \leq last(y')$. Dann gilt gemäß des Algorithmus (Schritt 4), daß $\delta(k) = y$ und $y \notin \bigcup_{i=l+1}^m \{x'_i\}$. Es folgt:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t_{T_{DAG}(\pi)}^{(0)}(y) \\
&= y[x'_m/e'_m] \dots [x'_{l+1}/e'_{l+1}][x'_l/e'_l][x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= y[x'_l/e'_l][x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= y[y/f(\delta(suc(k, 1)), \dots, \delta(suc(k, r)))] [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= f(\delta(suc(k, 1)), \dots, \delta(suc(k, r))) [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= f(\underbrace{\delta(suc(k, 1)) [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1]}_{=t_{\pi_{suc(k, 1)}}(\delta(suc(k, 1)))}, \dots, \underbrace{\delta(suc(k, r)) [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1]}_{=t_{\pi_{suc(k, r)}}(\delta(suc(k, r)))})
\end{aligned}$$

Aufgrund der Induktionsannahme folgt die Behauptung für diesen Teilfall.

- b) $\exists z \in V_{out}^k$ mit $z \neq y$, so daß $\forall z' \in V_{out}^k : last(z) \leq last(z')$. Es gilt $\delta(k) = z$ und es gibt eine Anweisung $\alpha'_p = y \leftarrow z$ in β mit $p > l$ (Schritt 3 oder 8). Dann ist $z \notin \bigcup_{i=l+1}^m \{x'_i\}$, sowie $y \notin \bigcup_{i=p+1}^m \{x'_i\}$ und es ergibt sich:

$$\begin{aligned}
t_{T_{DAG}(\pi)}(y) &= t_{T_{DAG}(\pi)}^{(0)}(y) \\
&= y[x'_m/e'_m] \dots [x'_{p+1}/e'_{p+1}] [y/z] [x'_{p-1}/e'_{p-1}] \dots [x'_l/e'_l] \dots [x'_1/e'_1] \\
&= y[y/z] [x'_{p-1}/e'_{p-1}] \dots [x'_{l+1}/e'_{l+1}] [x'_l/e'_l] [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= y[y/z] [x'_l/e'_l] [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= z[x'_l/e'_l] [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= z[z/f(\delta(suc(k, 1)), \dots, \delta(suc(k, r)))] [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= f(\delta(suc(k, 1)), \dots, \delta(suc(k, r))) [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1] \\
&= f(\delta(suc(k, 1)) [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1], \dots, \delta(suc(k, r)) [x'_{l-1}/e'_{l-1}] \dots [x'_1/e'_1])
\end{aligned}$$

Das Ergebnis ist identisch zu Fall (a).

Insgesamt gilt gemäß des Induktionsprinzips die Behauptung. \square

Mit den beiden obigen Lemmata ist die Korrektheit des gesamten DAG-Verfahrens beinahe offensichtlich.

Satz 2.2.13 (Korrektheit der DAG-Optimierung): Für $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ und eine Interpretation \mathfrak{A} für Σ gilt $\pi \sim_{\mathfrak{A}} T_{DAG}(\pi)$.

Beweis: Es reicht zu zeigen, daß aus $\mathfrak{A}[t_{\pi}(y)] = t_{T_{DAG}(\pi)}(y)$ für alle $y \in V_{out}$ folgt, daß $\mathfrak{A}[\pi] = \mathfrak{A}[T_{DAG}(\pi)]$. Dann gilt aufgrund der Lemmata 2.2.11 und 2.2.12 die Äquivalenz $\pi \sim_{\mathfrak{A}} T_{DAG}(\pi)$.

Es ist offensichtlich, daß $\mathfrak{A}[t] = \mathfrak{A}[\mathfrak{A}[t]]$ für $t \in T_{\Sigma}$ gilt. Daher folgt nach Lemma 1.2.4:

$$\begin{aligned}
\mathfrak{A}[\pi](\vec{in}) &= (\mathfrak{A}[t_{\pi}(y_1)[x_1/in_1, \dots, x_s/in_s]], \dots, \mathfrak{A}[t_{\pi}(y_t)[x_1/in_1, \dots, x_s/in_s]]) \\
&= (\mathfrak{A}[\mathfrak{A}[t_{\pi}(y_1)[x_1/in_1, \dots, x_s/in_s]], \dots, \mathfrak{A}[\mathfrak{A}[t_{\pi}(y_t)[x_1/in_1, \dots, x_s/in_s]]) \\
&= (\mathfrak{A}[t_{T_{DAG}(\pi)}(y_1)[x_1/in_1, \dots, x_s/in_s]], \dots, \mathfrak{A}[t_{T_{DAG}(\pi)}(y_t)[x_1/in_1, \dots, x_s/in_s]]) \\
&= \mathfrak{A}[T_{DAG}(\pi)](\vec{in})
\end{aligned}$$

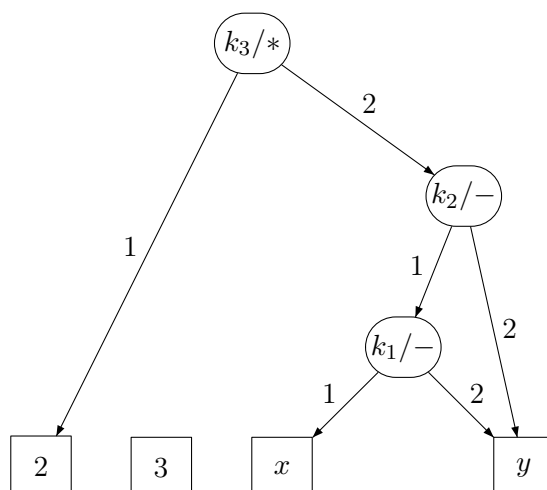
\square

2.2.4 Idempotenz der DAG-Optimierung

Wie für die klassischen Optimierungsverfahren ist auch für T_{DAG} die Idempotenz nachzuweisen, denn sonst erfüllt die DAG-Optimierung nicht die Anforderungen einer Programmtransformation.

Beispiel 2.2.14: Zunächst wollen wir uns das Verhalten der doppelten Anwendung von T_{DAG} am Beispiel des Programms π' aus Abbildung 2.3 anschauen. Durch die DAG-Konstruktion ergibt sich der DAG aus Abbildung 2.6. Man sieht, daß es sich um einen Teilgraphen des DAG aus Abbildung 2.2 handelt. Dabei sind die Knotennamen unterschiedlich, der generierte Code wäre aber identisch, was dadurch erreicht wird, daß in Schritt 5 von Algorithmus 2.2.5 Operationsknoten, denen keine Ausgabevariablen zugeordnet sind, in sukzessiv durchnummerierte Variablen umbenannt werden. Bei direkter Verwendung der Knotennamen gälte die Idempotenz auf keinen Fall.

Im neuen DAG sind nur jene Knoten vorhanden, die im Graphen aus Abbildung 2.2 ausgaberelevant waren; die anderen sind nämlich gar nicht in den Code von π' eingeflossen.



val-Funktion:

<i>i</i>	<i>u</i>	<i>v</i>	<i>x</i>	<i>y</i>	<i>v</i> ₁	<i>v</i> ₂
0			<i>x</i>	<i>y</i>		
1					<i>k</i> ₁	
2						<i>k</i> ₂
3	<i>k</i> ₃					
4		3				

$$last = \{u \mapsto 3, v \mapsto 4\}$$

Abbildung 2.6: $D_{\pi'}$ zum Programm π' aus Abbildung 2.2

Nun wollen wir formal überprüfen, ob unsere Beobachtung anhand eines Beispielprogramms allgemeingültig ist.

Satz 2.2.15 (Idempotenz): *Die DAG-Optimierung ist idempotent, d.h. für alle $\pi \in \mathcal{LC}$ gilt:*

$$T_{DAG}(T_{DAG})(\pi) = T_{DAG}(\pi)$$

Beweis: Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\Sigma = (F, C)$, $\beta := \alpha_1; \dots; \alpha_n$, Interpretation $\mathfrak{A} := (A, \varphi)$ für Σ und DAG $D_\pi := (G, val, last)$, wobei $G := (K, L, lab, suc)$, gegeben. Außerdem bezeichne $D_{\pi'} := (G', val', last')$ mit $G' := (K', L, lab', suc')$ den DAG von $\pi' := T_{DAG}(\pi) = ((F, A), \vec{v}_{in}, \vec{v}_{out}, \beta')$ mit $\beta' := \alpha'_1; \dots; \alpha'_m$. K und K' seien o.B.d.A. auf ausgaberelevante Knoten beschränkt.

Es ist zu zeigen:

1. Die Mengen der ausgaberelevanten Knoten sind jeweils gleich mächtig: $|K| = |K'|$.
2. Die Graphen G und G' sind (auf ausgaberelevante Knoten beschränkt) *isomorph*, d.h. es

existiert eine bijektive Abbildung $\iota : K \rightarrow K'$, so daß für alle $k \in K$ und $i \in \mathbb{N}$ gilt:

$$\begin{aligned} \text{suc}(k, i) = k' &\Rightarrow \text{suc}'(\iota(k), i) = \iota(k') \\ \text{lab}(k) &= \text{lab}'(\iota(k)) \end{aligned}$$

3. Für die *val*-Funktion gilt:

$$\forall y \in V_{out} : \iota(\text{val}(y, n)) = \text{val}'(y, m)$$

4. Definiere die Mengen der Operationsknoten aus K bzw. K' wie folgt:

$$\begin{aligned} K_{op} &:= \{k \in K \mid \exists r \in \mathbb{N} \exists f \in F^{(r)} : \text{lab}(k) = f\} =: \{k_{i_1}, \dots, k_{i_p}\} \\ K'_{op} &:= \{k \in K' \mid \exists r \in \mathbb{N} \exists f \in F^{(r)} : \text{lab}'(k) = f\} =: \{k_{j_1}, \dots, k_{j_p}\} \\ \text{mit } \forall l \in \{1, \dots, p-1\} &: k_{i_l} < k_{i_{l+1}} \text{ und} \\ \forall l \in \{1, \dots, p-1\} &: k_{j_l} < k_{j_{l+1}} \end{aligned}$$

Dann muß gelten:

$$\forall l \in \{1, \dots, p\} : \iota(k_{i_l}) = k_{j_l},$$

d.h. deren Reihenfolge wird eingehalten (sonst würde $T_{DAG}(\pi')$ u.U. eine andere Berechnungsreihenfolge aufweisen).

5. Die Relation der *last*-Einträge zu den Knotennummierungen und der *last*-Einträge untereinander bleibt erhalten:

$$\begin{aligned} \forall l \in \{1, \dots, p\} \forall y \in V_{out} &: i_l < \text{last}(y) \Rightarrow j_l < \text{last}'(y) \text{ bzw. } i_l > \text{last}(y) \Rightarrow j_l > \text{last}'(y) \\ \forall x, y \in V_{out} \text{ mit } x \neq y &: \text{last}(x) < \text{last}(y) \Rightarrow \text{last}'(x) < \text{last}'(y) \end{aligned}$$

Wenn obige Forderungen erfüllt sind, gilt die Idempotenz, da die DAG-Codegenerierung gemäß Algorithmus 2.2.5 deterministisch arbeitet und die Zuweisungsvariablen bis auf die Ausgabevariablen sukzessive durchnummeriert werden.

Die Gültigkeit der ersten Bedingung ist offensichtlich, denn genau die ausgaberelevanten Knoten K fließen in den Code von π' ein. Ein Knoten $k \in K$ muß in ähnlicher Form auch in K' vorhanden sein (bei Operationsknoten ist ein anderer Knotenname erlaubt), denn sonst wäre er bereits D_π nicht ausgaberelevant gewesen und der entsprechende Code nicht in π' eingeflossen. Umgekehrt kann es keinen Knoten $k' \in K'$ geben, der kein „Gegenstück“ in K hat, denn k' wurde aufgrund einer entsprechenden Operationsanweisung oder Konstantenzuweisung in π' erzeugt, die aus Knoten aus K generiert wurde.

Definiere ι durch

$$\begin{aligned} \forall k \in V \cup A &: \iota(k) := k \\ \forall l \in \{1, \dots, p\} &: \iota(k_{i_l}) := k_{j_l} \end{aligned}$$

Forderung 4 ist damit erfüllt; ι ist zusammen mit 1. bijektiv.

Für Beispiel 2.2.14 würde diese Definition bedeuten, daß $\iota(k_2) := k_1$, $\iota(k_5) := k_2$ und $\iota(k_7) := k_3$ gesetzt wird (die restlichen Knoten werden auf sich selbst abgebildet). Es werden dadurch im Beispiel keine Konflikte verursacht.

Zu 2: Operationsknoten aus K werden nach Schritt 2(a) des Algorithmus 2.2.5 in der durch ihre Nummerierung vorgegebenen Reihenfolge abgearbeitet. Daraus folgt, daß auch die Operationsanweisungen in π' in der Reihenfolge der Knotennummerierung erfolgen. Wenn daraus nun wieder neue Knoten generiert werden, besitzen diese die gleiche Ordnung (nicht Nummerierung!) wie die Knoten in K .

Daher gilt wegen der Definition von ι die Gleichheit der Beschriftungen $lab(k) = lab'(\iota(k))$ und die Isomorphie der Nachfolgerfunktionen $suc(k, i) = k' \Rightarrow suc'(\iota(k), i) = \iota(k')$ für $k, k' \in K$ und $i \in \mathbb{N}$ (da für diese, falls sie Operationsknoten sind, induktiv das gleiche gilt).

Zu 3: Unterscheidung nach dem Typ des von $val(y, n)$ referenzierten Knotens:

- $val(y, n) = a \in A$: Dann wird in π' eine Zuweisung $\alpha_l = y \leftarrow a$ erzeugt (wegen Schritt 3 oder 8), welche die letzte an y ist (da π' gemäß der DAG-Codegenerierung in SSA-Form vorliegt). Die Position von α_l hängt von $last(y)$ ab. Dann gilt auch $val'(y, m) = a$ und wegen $\iota(a) = a$ für alle $a \in A$ folgt die Behauptung.
- $val(y, n) = x \in V_{in}$: Da $V_{in} \cap V_{out} = \emptyset$ wird eine Anweisung $\alpha_l = y \leftarrow x$ in π' generiert (Schritt 3/8). Ähnlich wie oben folgt damit $val'(y, m) = x$ und mit $\iota(x) = x$ für alle $x \in V_{in}$ gilt die Behauptung.
- $val(y, n) = k_{i_q}$ mit $lab(k_{i_q}) = f \in F^{(r)}$: Es wird eine entsprechende Operationsanweisung

$$\alpha_l = \delta(k_{i_q}) \leftarrow f(\delta(suc(k_{i_q}, 1)), \dots, \delta(suc(k_{i_q}, r)))$$

in π' eingefügt. Es gilt entweder $\delta(k_{i_q}) = y$ (falls $y \in V_{out}^{k_{i_q}}$ und $\nexists y' \in V_{out}^{k_{i_q}}, y' \neq y$ mit $last(y') < last(y)$), oder aber $\delta(k_{i_q}) = y' \in V_{out}$ und für ein $l' > l$ gilt $\alpha_{l'} = y \leftarrow y'$ (Schritt 3/8).

Bei α_l handelt es sich nach Schritt 2 genau um die q -te Operationsanweisung (dazwischen können noch Konstanten- und Variablenzuweisungen an Ausgabevariablen auftreten). Im DAG $D_{\pi'}$ werden daher zunächst $q - 1$ Operationsknoten $k_{j_1}, \dots, k_{j_{q-1}}$ für die vorangehenden Operationsanweisungen angelegt und schließlich k_{j_q} . Es gilt dann natürlich $val(y, m) = k_{j_q}$.

Wegen $\iota(k_{i_l}) = k_{j_l}, \forall l \in \{1, \dots, p\}$ ist die Behauptung auch für diesen letzten Fall korrekt.

Abschließend ist noch die fünfte Bedingung zu zeigen:

Es gelte $i_l < last(y)$ für ein $l \in \{1, \dots, p\}$ und ein $y \in V_{out}$. Dann wird in π' zuerst Code für k_{i_l} generiert, bevor y einen Wert erhält (in Schritt 3 werden nur für $last$ -Einträge, die kleiner als die aktuelle Knotennummer sind, Anweisungen eingefügt). Folglich gilt dann auch $j_l < last'(y)$. Für die restlichen Fälle lassen sich analoge Folgerungen anstellen, denn die $last$ -Werte und die Knotennummern bestimmen die Anweisungsreihenfolge in π' , welche sich wiederum auf die Knotennummerierung und die $last$ -Funktion in $D_{\pi'}$ überträgt. \square

Da das DAG-Verfahren deterministisch arbeitet, ist T_{DAG} somit eine Programmtransformation.

2.3 Sonstige Optimierungsverfahren

Dieser Abschnitt soll einen groben Überblick über weitere Optimierungsverfahren für LC-Programme geben. Es findet jedoch keine detaillierte Untersuchung der einzelnen Eigenschaften dieser Verfahren statt, da diese für die weiteren Betrachtungen in Kapitel 3 nicht relevant sind.

2.3.1 Speicherbedarfsreduzierung

Bei der Speicherbedarfsreduzierung geht es darum, die Anzahl der im Programm verwendeten Variablen zu verringern. Dabei wird für eine Zuweisung, falls möglich, eine alte Variable benutzt, anstatt eine neue zu verwenden. Das hier vorgestellte Verfahren wurde im Rahmen von [2] entwickelt.

In der Analysephase werden alle Variablen, die vor bzw. nach der zu untersuchenden Anweisung auftreten, „aufgesammelt“. Dafür ist eine *kombinierte Analyse* erforderlich, die sowohl eine Vorwärts- als auch eine Rückwärtsanalyse umfaßt.

Definition 2.3.1: Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$. Definiere die Transferfunktion $t_\alpha : \mathfrak{P}(V_\pi) \rightarrow \mathfrak{P}(V_\pi)$ für eine Anweisung $\alpha := x \leftarrow e$ und $M \subseteq V_\pi$ durch:

$$t_\alpha(M) := M \cup \{x\} \cup V_e$$

Die Mengen der *vor* und *nach* der i -ten Anweisung auftretenden Variablen $V_i^<$ und $V_i^>$ ergeben sich dann wie folgt:

$$\begin{aligned} V_1^< &= V_{in} \\ V_i^< &= t(V_{i-1}^<) \text{ für } i \in \{2, \dots, n\} \\ V_n^> &= V_{out} \\ V_i^> &= t(V_{i+1}^>) \text{ für } i \in \{1, \dots, n-1\} \end{aligned}$$

Man versucht nun, eine bisher unbenutzte Variable der linken Seite nach Möglichkeit durch eine vorher definierte Variable zu ersetzen.

Definition 2.3.2 (Speicherbedarfsreduzierung): Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\Sigma = (F, C)$, $\beta = \alpha_1; \dots; \alpha_n$ und $\alpha_i = x_i \leftarrow e_i$. Ordne jeder Anweisung α_i eine Substitution $\delta_i : V_\pi \rightarrow V_\pi$ zu:

$$\begin{aligned} \delta_0 &:= id \\ \delta_i &:= \begin{cases} \delta_{i-1}[x_i/y] & \text{falls } x_i \notin \delta_{i-1}(V_i^<) \cup V_{out} \text{ und } \exists y \in \delta_{i-1}(V_i^< \setminus V_i^>) \\ \delta_{i-1} & \text{sonst} \end{cases} \end{aligned}$$

Dann berechnet sich die Optimierungsfunktion $T_M : \mathcal{LC} \rightarrow \mathcal{LC}$ wie folgt:

$$T_M(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta') \text{ mit } \beta' := \delta_1(x_1) \leftarrow \bar{\delta}_0(e_1); \delta_2(x_2) \leftarrow \bar{\delta}_1(e_2); \dots; \delta_n(x_n) \leftarrow \bar{\delta}_{n-1}(e_n)$$

Dabei ist $\bar{\delta}$ die *monotone Fortsetzung* von δ und gegeben durch

$$\begin{aligned} \bar{\delta}(c) &:= c \text{ für } c \in C \\ \bar{\delta}(x) &:= \delta(x) \text{ für } x \in V \\ \bar{\delta}(f(u_1, \dots, u_r)) &:= f(\bar{\delta}(u_1), \dots, \bar{\delta}(u_r)) \text{ für } u_i \in V \cup C, f \in F^{(r)} \end{aligned}$$

In obiger Version ist das Verfahren nichtdeterministisch, da es bei der Wahl von y mehrere Möglichkeiten geben kann. Man könnte die freie Wahl von y durch ein deterministisches Auswahlverfahren einschränken.

In einer Zuweisung $x \leftarrow e$ wird die Variable x durch eine vorher definierte Variable y ersetzt, wenn

- x eine neue, bisher undefinierte Variable ist (sonst keine Einsparung),
- x keine Ausgabevariable ist (sonst ließe sich x sowieso nicht aus dem Programm entfernen),

- y schon vorher auftrat (da ansonsten keine Verbesserung) und
- y später nicht mehr benutzt wird.

Dabei sind bei den Überprüfungen der Mengenmitgliedschaft jeweils die Substitutionen der Variablen zu berücksichtigen, d.h. die Variablen der entsprechenden Mengen gemäß der bisherigen Ersetzungen umzubenennen.

Beispiel 2.3.3: Nun zur Illustration eine Beispielrechnung anhand eines kurzen Beispielprogramms

$$\begin{aligned} \pi_{\text{memdemo}} := (\Sigma_{\text{arithm}}, (x, y), (z), \beta) \text{ mit } \beta := & v \leftarrow 2 * x; \\ & w \leftarrow v * v; \\ & z \leftarrow v + w; \end{aligned}$$

wobei für die Substitutionen nur diejenigen Argumente angegeben werden, die nicht auf sich selbst abgebildet werden:

i	α_i	$V_i^<$	$V_i^>$	δ_i	Neue Anweisung α'_i
1	$v \leftarrow 2 * x;$	$\{x, y\}$	$\{v, w, z\}$	$\{v \mapsto x\}$	$x \leftarrow 2 * x;$
2	$w \leftarrow v * v;$	$\{v, x, y\}$	$\{v, w, z\}$	$\{v \mapsto x, w \mapsto y\}$	$y \leftarrow x * x;$
3	$z \leftarrow v + w;$	$\{v, w, x, y\}$	$\{z\}$	$\{v \mapsto x, w \mapsto y\}$	$z \leftarrow x + y;$

Man sieht, daß in der optimierten Fassung nur noch 3 anstatt 5 Variablen verwendet werden. Aufgrund des Nichtdeterminismus in T_M , wäre auch alternativ eine Ersetzung von v durch y und w durch x möglich gewesen.

T_M ergibt stark äquivalente Programme, da ja, ähnlich wie bei der Kopierpropagation, rein syntaktisch vorgegangen wird und nur Variablen durch andere ersetzt werden.

Die Speicherbedarfsreduzierung ließe sich eventuell sinnvoll nach einer DAG-Optimierung anwenden, da diese ja Code in SSA-Form erzeugt.

2.3.2 Algebraische Transformationen

Bei den meisten bisher betrachteten Verfahren wurden vorhandene spezielle algebraische Eigenschaften der Operationen in der betrachteten Interpretation nicht berücksichtigt. Algebraische Transformationen werden (wie die klassischen Verfahren) in aktuellen Compilern, beispielsweise im GCC-Compiler, eingesetzt [11].

2.3.2.1 Operationsspezifische Gleichungen

Für bestimmte Operationen und Wertebereiche gelten spezielle Gleichungen, wie z.B. für Kommutativität, neutrale Elemente. Assoziativität oder Distributivität können nur ausgenutzt werden, wenn man den aus mehreren Anweisungen entstehenden Operationsterm bei der Optimierung betrachtet oder gleich mit der Termdarstellung des LC-Programms arbeitet. Dies liegt an der Beschränkung der Verschachtelungstiefe von Ausdrücken auf 1, d.h. es kann nur einen Operator (Funktionssymbol) auf der rechten Seite einer Anweisung geben.

Gültige Gleichungen könnte man in einer Art Gleichungssystem abspeichern, wobei Ausdrücke, die die linke Seite einer Gleichung „matchen“ entsprechend durch die rechte ersetzt werden (\rightarrow *Termersetzung* [6]). Dies würde natürlich nur dann Sinn machen, wenn dadurch auch eine Verbesserung im Ergebnis erreicht würde.

Beispiel 2.3.4: Eine Zuweisung $x \leftarrow y * 0$ ($\varphi(*) = *_{\mathbb{Z}}$) würde in der Konstantenfaltung einen unbekanntes Wert für x bedeuten, sofern y undefiniert ist. Aber unter Berücksichtigung der für $*_{\mathbb{Z}}$ geltenden Gleichung $\forall z \in \mathbb{Z} : z * 0 = 0$ ist die Variable x nach Ausführung der Anweisung definitiv mit 0 belegt. Man könnte also $x \leftarrow y * 0$ durch $x \leftarrow 0$ ersetzen.

Die Ersetzungen sollten dabei *vor* der eigentlichen Optimierung bei vorgegebener Interpretation durchgeführt werden. Dadurch ist eine Abänderung der bisher vorgestellten Verfahren unnötig.

2.3.2.2 Reduktion der Operationsstärke

Ein anderer Ansatz ist die Ersetzung von Operationen, wobei eine (in Bezug auf Rechenzeit) teurere Operation durch eine günstigere ersetzt wird. In Frage kämen z.B. die Ersetzung von Exponentiation durch Multiplikation, also z.B. Ersetzung von $x \leftarrow y^2$ durch $x \leftarrow y * y$ oder die Ersetzung von Multiplikation durch Addition (Anweisungen wie $x \leftarrow 2 * y$ durch $x \leftarrow y + y$ substituieren).

Ob dies letztendlich günstiger ist, hängt natürlich auch von der Zielhardware ab. Es ist allerdings fraglich, ob solche Regeln hinreichend oft anwendbar sind, um den Implementierungsaufwand zu rechtfertigen.

3 Vergleich der Optimierungsverfahren

Nach den formalen Grundlagen und der Vorstellung der einzelnen Optimierungsverfahren, wenden wir uns nun dem Hauptteil dieser Arbeit zu: dem Vergleich der DAG-Optimierung mit den klassischen Verfahren.

Zunächst wird die Optimalität der DAG-Optimierung bezüglich der klassischen Verfahren in Abschnitt 3.1 untersucht, um dann später im Abschnitt 3.3 zu überprüfen, ob sich das DAG-Verfahren durch eine Kombination bzw. wiederholte Anwendung der klassischen Verfahren „simulieren“ läßt, so daß die „Optimierungsstärke“ vergleichbar ist. Dazu müssen vorher die zwischen den klassischen Transformationen geltenden Beziehungen analysiert werden, d.h. inwieweit sich die Transformationen gegenseitig beeinflussen (Abschnitt 3.2).

3.1 Optimalität der DAG-Optimierung

Die im letzten Kapitel vorgestellte DAG-Optimierung scheint sehr effektiv zu arbeiten, ergab sich doch in einem Beispiel (Abb. 2.3) ein im Vergleich zu den klassischen Transformationen sehr kurzes Ergebnisprogramm. Wir wollen nun klären, ob das DAG-Verfahren im allgemeinen Fall tatsächlich optimal bezüglich aller klassischen Verfahren ist. Zunächst untersuchen wir dazu die Dead Code Elimination.

Satz 3.1.1 (Optimalität bzgl. T_{DC}): Für alle $\pi \in \mathcal{LC}$ ist $T_{DAG}(\pi)$ optimal bzgl. T_{DC} , d.h. es gilt $T_{DC}(T_{DAG}(\pi)) = T_{DAG}(\pi)$.

Beweis: Sei π ein LC-Programm mit n Anweisungen und $\pi' := T_{DAG}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta = \alpha_1; \dots; \alpha_m$ und $\alpha_i := x_i \leftarrow e_i$. Außerdem sei $D_\pi = ((K, L, lab, suc), val, last)$ der DAG von π . Man nehme nun an, daß ein $j \in \{1, \dots, m\}$ existiert, so daß $x_j \notin LV_j$ (in π').

Es sind zwei Fälle zu unterscheiden:

1. x_j wird nach α_j nie mehr benutzt ($x_j \notin V_{e_i}, i > j$).

Dann existiert in D_π ein Knoten $\delta^{-1}(x_j)$ ¹, auf den keine weiteren Kanten zeigen, d.h.

$$\forall i \in \mathbb{N} \text{ und } \forall k \in K : suc(k, i) \neq \delta^{-1}(x_j)$$

Außerdem ist x_j keine Ausgabevariable, denn sonst wäre α'_j kein Dead Code. Es existiert also kein $y \in V_{out}$, so daß $val(y, n) = \delta^{-1}(x_j)$. Dann ist $\delta^{-1}(x_j)$ nach Definition 2.2.4 nicht ausgaberelevant und die Anweisung α_j wäre niemals erzeugt worden.

2. x_j wird in einer Anweisung α_l mit $l > j$ neudefiniert ($x_l = x_j$), aber $x_j \notin V_{e_i}, k < i \leq l$.

Algorithmus 2.2.5 erzeugt gemäß seiner Definition Ausgabeprogramme, in denen keine Variable mehrfach auf der linken Seite einer Zuweisung auftritt (SSA-Form). Daher ergibt sich ein Widerspruch.

Da in beiden Fällen ein Widerspruch auftritt kann es keine Anweisungen in $T_{DAG}(\pi)$ geben, die Dead Code darstellen. \square

¹ δ sei die Substitution, die man nach Abarbeitung des kompletten Algorithmus 2.2.5 erhält.

Wie bereits aufgrund der Elimination von nicht ausgaberelevanten Knoten zu vermuten war, erzeugt die DAG-Optimierung also tatsächlich Ausgabeprogramme, die sich nicht weiter mit der Dead Code Elimination optimieren lassen. Ähnlich sieht es auch für die Common Subexpression Elimination aus.

Satz 3.1.2 (Optimalität bzgl. T_{CS}): *Es gilt $T_{CS}(T_{DAG}(\pi)) = T_{DAG}(\pi)$ für alle $\pi \in \mathcal{LC}$.*

Beweis: Seien π , π' und D_π wie im Beweis von Satz 3.1.1 gegeben. Ähnlich wie dort führen wir einen Beweis per Widerspruch:

Annahme: Es existieren $i, j \in \{1, \dots, m\}$ mit $e_i = e_j = f(u_1, \dots, u_r)$, so daß $j \in wh(i)$, d.h. e_j ist eine gültige Wiederholung von e_i . T_{CS} würde diese mithilfe einer temporären Variable eliminieren.

Da Algorithmus 2.2.5 für jeden Operationsknoten im DAG genau eine Anweisung erzeugt, müßten folglich zwei Knoten $k, k' \in K$ existieren, wobei $suc(k, j) = suc(k', j)$ für alle $j \in \{1, \dots, r\}$ und $lab(k) = lab(k') = f \in F^{(r)}$.

Dies ist aber gemäß der DAG-Konstruktion in Algorithmus 2.2.2, Teilabschnitt 3(b)(b1) nicht möglich, da nachdem k im DAG erzeugt wurde, kein k' mit obigen Eigenschaften angelegt, sondern nur die *val*-Funktion aktualisiert worden wäre. Aus dem Widerspruch folgt die Behauptung. \square

Die Optimalität der DAG-Optimierung bezüglich der Konstantenfaltung zu zeigen, ist etwas aufwendiger:

Satz 3.1.3 (Optimalität bzgl. T_{CF}): *Es gilt $T_{CF}(T_{DAG}(\pi)) = T_{DAG}(\pi)$ für ein $\pi \in \mathcal{LC}$.*

Beweis: Es reicht, zu zeigen, daß für alle $i \in \{1, \dots, n\}$ und alle $y \in V_\pi$ gilt:

$$RD_i(y) \in A \Rightarrow val(y, i-1) \in A$$

Dann kann die Konstantenfaltung bei Anwendung auf π' keine zusätzliche Änderung mehr bewirken (da T_{DAG} bereits das gesamte Ersetzungspotential ausgenutzt hat). Vollständige Induktion über $i \in \{1, \dots, n\}$:

- $i = 1$: $RD_1(y) = \sigma_\perp(y) = \perp$
- $i \rightarrow i + 1$: Fallunterscheidung nach der Variable y :
 1. Falls $y = x_i$ gilt:

$$\begin{aligned} RD_{i+1}(y) &= RD_i[x_i/\bar{\mathfrak{A}}[e_i]RD_i](y) \\ &= \bar{\mathfrak{A}}[e_i]RD_i \\ &\in A \end{aligned}$$

Und es folgt:

$$\begin{aligned} &\forall z \in V_{e_i} : RD_i(z) \in A \\ &\stackrel{IA}{\Rightarrow} \forall z \in V_{e_i} : val(z, i-1) \in A \\ &\Rightarrow val(y, i) \in A \text{ (gemäß DAG-Konstruktion)} \end{aligned}$$

2. Sonst ($y \neq x_i$) ergibt sich:

$$\begin{aligned} RD_{i+1}(y) &= RD_i[x_i/\bar{\mathfrak{A}}[e_i]]RD_i(y) \\ &= RD_i(y) \\ &\in A \\ &\stackrel{IA}{\Rightarrow} val(y, i-1) \in A \\ &\Rightarrow val(y, i) \in A \text{ (weil } y \neq x_i) \end{aligned}$$

Aufgrund des Induktionsprinzips folgt die Gültigkeit der Behauptung. \square

Jetzt ist nur noch die Optimalität des DAG-Verfahrens im Hinblick auf die Kopierpropagation zu untersuchen.

Satz 3.1.4 (Optimalität bzgl. T_{CP}): *Es gilt $T_{CP}(T_{DAG}(\pi)) = T_{DAG}(\pi)$ für ein $\pi \in \mathcal{LC}$.*

Beweis: Der Algorithmus 2.2.5 zur Codeerzeugung aus einem DAG erzeugt nur Kopieranweisungen als finale Zuweisungen an Ausgabevariablen der Form $y \leftarrow z$ mit $y \in V_{out}$ (Schritte 3 und 8). Die Ausgabevariable y wird jedoch danach im vom den Algorithmus erzeugten Programm nicht mehr benutzt, da bei Zugriffen auf den entsprechenden Wert immer direkt auf die dem Operationsknoten zugeordnete Variable (oder Eingabevariable) z zugegriffen wird.

Es kann zwar $z \in V_{out}$ gelten, dann ist z aber einem Operationsknoten zugeordnet, d.h. $\exists k \in K : \delta(k) = z$ mit $lab(k) = f \in F$. Somit sind zwar die CP -Informationen nicht alle leer, eine Variablensubstitution ist jedoch nicht möglich. \square

Nun, da wir erkannt haben, daß die DAG-Optimierung bezüglich allen vier klassischen Optimierungsverfahren optimal ist, wäre die Gültigkeit der Rückrichtung interessant. Anhand der Beispiele in Kapitel 2 haben wir gesehen, daß keine einzelne der klassischen Optimierungen ein DAG-optimales Programm liefert. Daher kommen nur Kompositionen der Einzeltransformationen in Frage.

3.2 Beziehungen zwischen den klassischen Transformationen

Um eine günstige Abfolge der klassischen Verfahren zu finden, ist es sinnvoll, zu untersuchen, welche Beziehungen zwischen den klassischen Optimierungsverfahren gelten. In Abschnitt 2.1.3 hatten wir schon gesehen, daß etwa nach der Konstantenfaltung eine Dead Code Elimination sinnvoll ist, denn ansonsten wird die Anzahl der Anweisungen nicht reduziert. Ähnlich verhielt es sich mit der Kopierpropagation.

Diese Beobachtungen wollen wir nun etwas formalisieren, um dann am Ende dieses Abschnitts eine sinnvolle Abfolge der Einzeltransformationen angeben zu können. Es gilt dabei zu prüfen, ob ein bezüglich eines Verfahrens A optimales LC-Programm nach Anwendung eines Verfahren B mit A weiter optimiert werden kann.

Definition 3.2.1: Seien $T_i : \mathcal{LC} \rightarrow \mathcal{LC}$, $i \in \{1, 2\}$ zwei Programmtransformationen und π ein LC-Programm. Außerdem gelte $T_1(\pi) = \pi$, d.h. π ist T_1 -optimal. Gilt

$$T_2(\pi) \neq \pi \Rightarrow T_1(T_2(\pi)) \neq T_2(\pi)$$

- für alle T_1 -optimalen $\pi \in \mathcal{LC}$, dann heißt T_2 stark T_1 -verbessernd ($T_2 \rightrightarrows T_1$)

- für (mindestens) ein T_1 -optimales $\pi \in \mathcal{LC}$, so heißt T_2 (schwach) T_1 -verbessernd ($T_2 \rightarrow T_1$).

Im folgenden betrachten wir die klassischen Optimierungsverfahren einzeln und bestimmen für jedes, welche Optimierungen sich zur Verbesserung/Zwischenschaltung eignen.

3.2.1 Verbesserungsschritte für die Dead Code Elimination

In Abschnitt 2.1.4 hatten wir anhand eines Beispiels schon erkannt, daß die Kopierpropagation u.U. Dead Code erzeugt. Nun wollen wir dieses Verhalten etwas genauer untersuchen.

Lemma 3.2.2: T_{CP} ist schwach T_{DC} -verbessernd.

Beweis: Es reicht zu zeigen,

1. daß ein T_{DC} -optimales LC-Programm π existiert, welches nicht T_{CP} -optimal ist, für das $T_{DC}(T_{CP}(\pi)) \neq T_{CP}(\pi)$ gilt ($T_{CP} \rightarrow T_{DC}$) und
2. daß ein T_{DC} -optimales LC-Programm π existiert, welches nicht T_{CP} -optimal ist, für das $T_{DC}(T_{CP}(\pi)) = T_{CP}(\pi)$ gilt ($T_{CP} \not\rightarrow T_{DC}$).

Zu 1: Das Programm aus Abbildung 2.1 ist offensichtlich T_{DC} -optimal. Wir wenden nun T_{DC} auf $\pi' := T_{CP}(\pi)$ (siehe Beispiel 2.1.24) an:

i	α'_i	LV'_i	Dead Code?
1	$u \leftarrow x;$	$\{x, y\}$	Ja
2	$y \leftarrow x + y;$	$\{x, y\}$	Nein
3	$v \leftarrow x;$	$\{x, y\}$	Ja
4	$u \leftarrow x * y;$	$\{u, x\}$	Nein
5	$v \leftarrow u + x$	$\{u, v\}$	Nein

Es können jetzt aufgrund der Zwischenschaltung der Kopierpropagation zwei Anweisungen entfernt werden. Also gilt mindestens $T_{CP} \rightarrow T_{DC}$.

Zu 2: Eine etwas modifizierte Fassung von π ist π'' :

$$\pi'' := (\Sigma_{\text{arithm}}, (x, y), (u, v), \beta) \text{ mit } \beta : \begin{aligned} &u \leftarrow x; \\ &x \leftarrow u + y; \\ &v \leftarrow u; \\ &u \leftarrow v * x; \end{aligned}$$

Man sieht leicht, daß das Programm keinen Dead Code enthält. Die Anwendung von $T_{DC} \circ T_{CP}$ auf π'' ergibt:

i	α'_i	CP_i	$T_{CP}(\pi'')$	LV_i	Dead Code?
1	$u \leftarrow x;$	\emptyset	$u \leftarrow x;$	$\{u, x, y\}$	Nein
2	$x \leftarrow u + y;$	$\{(u, x, 1)\}$	$x \leftarrow x + y;$	$\{u, x\}$	Nein
3	$v \leftarrow u;$	\emptyset	$v \leftarrow u;$	$\{u, v, x\}$	Nein
4	$u \leftarrow v * x;$	$\{(v, u, 1)\}$	$u \leftarrow u * x;$	$\{u, v\}$	Nein

Da nach der Kopierpropagation die Dead Code Elimination keine Änderung bewirkt, folgt, daß $T_{CP} \Rightarrow T_{DC}$ nicht gilt. \square

Auch nach einer Konstantenfaltung ist eine Dead Code Elimination sinnvoll.

Lemma 3.2.3: T_{CF} ist schwach T_{DC} -verbessernd.

Beweis: Beweis durch die Angabe von zwei Beispielprogrammen, analog zum Beweis von Lemma 3.2.2:

$$\pi := ((\{f\}, \{a\}), \varepsilon, (y), \beta) \text{ mit } \beta : x \leftarrow a;$$

$$y \leftarrow f(x);$$

Außerdem sei $\mathfrak{A} := (\varphi, A)$. π ist T_{DC} -optimal und ist eine Konstantenfaltung möglich: Die zweite Anweisung wird durch $y \leftarrow \varphi(f)(\varphi(a))$ ersetzt. Dann eliminiert die Dead Code Elimination die erste Anweisung, da x nirgendwo mehr benötigt wird.

Man könnte vermuten, daß dies in ähnlicher Form immer möglich ist, da es ja immer mindestens eine Konstantenzuweisung der Form $x \leftarrow a$ oder eine Anweisung mit einer Berechnung auf konstanten Argumenten ($x \leftarrow f(a_1, \dots, a_r)$) geben muß, um Ersetzungen vornehmen zu können. Die Konstantenfaltung ersetzt dann alle Vorkommen von x , bis x einen neuen Wert erhält. Dies würde bewirken, daß die Variable x nicht in der entsprechenden LV -Menge enthalten ist und damit die Konstantenzuweisung bzw. die Operationsanweisung mit konstanten Argumenten Dead Code werden.

Es gibt aber eine Ausnahme: x ist eine Ausgabevariable. Dann ist keine Entfernung möglich, da der Wert von x noch benötigt wird. Also bleibt für das Programm $\pi' := ((\{f\}, \{a\}), \varepsilon, (x, y), \beta)$, in dem x zusätzlich zu y eine Ausgabevariable ist, eine Dead Code Elimination auch nach einer Konstantenfaltung wirkungslos. \square

Im Gegensatz zu den anderen beiden Verfahren ist nach einer Common Subexpression Elimination nicht mit zusätzlichem Dead Code zu rechnen.

Lemma 3.2.4: T_{CS} ist nicht T_{DC} -verbessernd.

Beweis: Wir gehen o.B.d.A. von der vereinfachten Situation aus, daß nur ein Ausdruck gültige Wiederholungen besitzt. Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta = \alpha_1; \dots; \alpha_n$ gegeben und sei α_k eine Anweisung mit $wh(k) = \{j_1, \dots, j_m\} \neq \emptyset$ mit $j_l < j_{l+1}$, $l \in \{1, \dots, m-1\}$. Dann hat der geänderte Teil von $\pi' := T_{CS} := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$ mit $\beta' := \alpha'_1; \dots; \alpha'_q$ folgende Struktur:

$$\begin{array}{l} \vdots \\ t_k \leftarrow e_k \\ x_k \leftarrow t_k \\ \vdots \\ x_{j_1} \leftarrow t_k \\ \vdots \\ \vdots \\ x_{j_m} \leftarrow t_k \\ \vdots \end{array} \left. \begin{array}{l} \vphantom{\vdots} \\ \vphantom{x_k} \\ \vphantom{x_{j_1}} \\ \vphantom{x_{j_m}} \end{array} \right\} \begin{array}{l} \text{Sektion A} \\ \text{Sektion B} \\ \text{Sektion C} \end{array}$$

Annahme: Es existiert ein $l \in \{1, \dots, q\}$, so daß α'_l Dead Code ist. Fallunterscheidung:

1. α'_i liegt in Programmsektion C:

Da in Sektion C keine Änderungen von T_{CS} vorgenommen worden sind, und die Dead Code Elimination eine Rückwärtsanalyse ist, sind die LV -Analyseinformationen identisch zu den LV' -Informationen², d.h. α'_i kann kein Dead Code sein.

2. α'_i liegt in Programmsektion B:

Feststellung ()*: Es kann hier keine Zuweisung der Form $x \leftarrow e$ mit $x \in V_{e_k}$ vorkommen, da die Ersetzungen sonst so nicht durchgeführt worden wären (k wäre ab dieser Anweisung nicht mehr in den AE -Analysemengen enthalten).

Deshalb werden nur Variablen aus den LV' -Mengen entfernt, die auch aus den entsprechenden LV -Mengen entfernt wurden. Außerdem befindet sich statt den Variablen aus V_{e_k} t_k in den LV' -Mengen. Dies kann aber wegen (*) nicht zu einer Bildung von Dead Code führen.

Also ist α'_i kein Dead Code.

3. α'_i liegt in Programmsektion A:

Aufgrund der Anweisung $t_k \leftarrow e_k$ ist $V_{e_k} \subseteq LV'_{k-1}$ und $t_k \notin LV'_{k-1}$. Daher ergibt sich keine Änderung zu den vorherigen Analysemengen. Daraus folgt, daß α'_i kein Dead Code sein kann.

Die Annahme wurde damit widerlegt und die Behauptung des Lemmas gilt. \square

3.2.2 Verbesserungsschritte für die Common Subexpression Elimination

Bei der Common Subexpression Elimination kann die vorherige Anwendung der Kopierpropagation u.U. die Ersetzung zusätzlicher Ausdrücke ermöglichen. Dies ist intuitiv dadurch erklärbar, daß zunächst ungleiche Ausdrücke durch die Substitution von Variablen angeglichen werden können.

Lemma 3.2.5: *Die Propagation von Kopien T_{CP} ist schwach T_{CS} -verbessernd.*

Beweis: Wie im Beweis von Lemma 3.2.2 sind zwei Beweisschritte notwendig.

1. Man betrachte das einfache Programm

$$\begin{aligned} \pi := (\Sigma_{\text{arithm}}, (x, y), (u, v), \beta) \text{ mit } \beta : v \leftarrow x; \\ u \leftarrow x + y; \\ v \leftarrow v + y; \end{aligned}$$

Es ist leicht ersichtlich, daß in π keine gemeinsamen Teilausdrücke enthalten sind, die durch T_{CS} eliminiert werden könnten. Nach der Anwendung von T_{CP} erhält man jedoch:

$$\begin{aligned} \pi' := T_{CP}(\pi) := (\Sigma_{\text{arithm}}, (x, y), (u, v), \beta') \text{ mit } \beta' : v \leftarrow x; \\ u \leftarrow x + y; \\ v \leftarrow x + y; \end{aligned}$$

Jetzt führt eine Common Subexpression Elimination zu einer Veränderung und es ergibt sich:

$$\begin{aligned} \pi'' := T_{CS}(\pi') := (\Sigma_{\text{arithm}}, (x, y), (u, v), \beta'') \text{ mit } \beta'' : v \leftarrow x; \\ t_2 \leftarrow x + y; \\ u \leftarrow t_2; \\ v \leftarrow t_2; \end{aligned}$$

Folglich ist T_{CP} schwach T_{CS} -verbessernd.

² LV_i seien die Analyseinformationen der Live Variable Analysis für π und LV'_i entsprechend für π' .

2. Daß $T_{CP} \Rightarrow T_{CS}$ nicht gilt, ist trivial, denn T_{CP} führt auch Ersetzungen in solchen Programmen durch, die gar keine Operationsausdrücke enthalten, und somit führt T_{CS} in diesem Fall zu keiner Veränderung. \square

Die Konstantenfaltung kann in Spezialfällen die Ersetzung von Ausdrücken mit konstanten Operanden durch T_{CS} ermöglichen, während die Dead Code Elimination offensichtlich keinerlei Verbesserung bringt, da sie nur Anweisungen entfernt.

Korollar 3.2.6: *Es gilt:*

1. T_{CF} ist schwach T_{CS} -verbessernd und
2. T_{DC} ist nicht T_{CS} -verbessernd.

Beweisskizze:

1. Die Aussage folgt mit dem Beispielprogramm

$$\begin{aligned} \pi := (\Sigma_{\text{arithm}}, (x), (u, v), \beta') \text{ mit } \beta : & y \leftarrow 4; \\ & u \leftarrow x + 4; \\ & v \leftarrow x + y; \end{aligned}$$

unter Verwendung der auf \mathbb{Z} üblichen Interpretation. Man kann recht einfach Programme angeben, für die dies nicht gilt ($T_{CF} \not\Rightarrow T_{CS}$).

2. Klar, da die Dead Code Elimination durch die Entfernung von Anweisungen die Verfügbarkeit von Ausdrücken nicht erhöhen kann.

3.2.3 Verbesserungsschritte für die Propagation von Kopien

In Abschnitt 3.2.2 wurde gezeigt, daß die Kopierpropagation die Common Subexpression Elimination begünstigt. Diese Aussage gilt ebenfalls in umgekehrter Richtung.

Lemma 3.2.7: *T_{CS} ist schwach T_{CP} -verbessernd.*

Beweis: Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ ein T_{CP} -optimales Programm mit $\beta := \alpha_1; \dots; \alpha_n$, welches nicht T_{CS} -optimal ist. Dann existieren $k, i_j \in \{1, \dots, n\}$, $j \in \{1, \dots, m\}$, $k < i_j < i_{j+1}$ für $j \in \{1, \dots, m-1\}$, so daß $wh(k) = \{i_1, \dots, i_m\}$.

$\alpha_k = x \leftarrow e$ wird nun durch $t_k \leftarrow e$; $x \leftarrow t_k$ ersetzt und $\alpha_{i_j} = x_{i_j} \leftarrow e$ durch $x_{i_j} \leftarrow t_k$. Werden x oder die x_{i_j} nun später im Programm vor ihrer Neudefinition benutzt (Auftreten auf rechter Seite), dann wird diese Benutzung gemäß T_{CP} durch t_k ersetzt.

In dem Fall, daß keine Benutzung der Variablen x oder x_{i_j} vor ihrer Neudefinition auftritt (es handelt sich bei α_k bzw. α_{i_j} um Dead Code), verändert T_{CP} das Programm nicht (sofern keine weiteren k mit $wh(k) \neq \emptyset$ existieren). \square

Wie wir später sehen werden, ist es bei manchen Programmen möglich, wiederholt die Common Subexpression Elimination und die Kopierpropagation im Wechsel anzuwenden und bei jeder Anwendung eine Verbesserung zu erreichen.

Korollar 3.2.8: *Weder T_{CF} noch T_{DC} sind T_{CP} -verbessernd.*

Die Gültigkeit des Korollars ist klar. Denn bei T_{CF} werden Variablen bzw. komplette Operationsausdrücke durch Konstanten ersetzt. Dies reduziert das Kopier-Optimierungspotential. Gleiches gilt

für die Entfernung von Anweisungen bei der Dead Code Elimination.

3.2.4 Verbesserungsschritte für die Konstantenfaltung

Als einzige der klassischen Optimierungsverfahren profitiert die Konstantenfaltung nicht von der vorherigen Durchführung eines der anderen Verfahren. Dies liegt u.a. daran, daß die Konstantenfaltung die erweiterte Anweisungsemantik nutzt, um Anweisungen soweit wie möglich auszuwerten, während die anderen Verfahren „nur“ syntaxbasiert arbeiten.

Lemma 3.2.9: *Es gilt:*

1. $T_{DC} \not\rightarrow T_{CF}$,
2. $T_{CS} \not\rightarrow T_{CF}$ und
3. $T_{CP} \not\rightarrow T_{CF}$.

Beweis: Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta := \alpha_1; \dots; \alpha_n$ ein T_{CF} -optimales Programm.

1. Wenn in π aufgrund der Dead Code Elimination Anweisungen entfernt werden, erhöht sich dadurch offensichtlich nicht die Zahl der als konstant bekannten Variablen, im Gegenteil, sie verringert sich. Daher ist $T_{CF}(T_{DC}(\pi)) = T_{DC}(\pi)$.
2. Sei $\alpha_k = x \leftarrow e$ eine Anweisung in π mit $wh(i) = \{i_1, \dots, i_m\} \neq \emptyset$. e muß von der Form $e = f(u_1, \dots, u_r)$ sein, wobei für mindestens ein $j \in \{1, \dots, r\}$ gelten muß, daß $u_j \in V$ und $RD_k(u_j) = \perp$ (denn sonst wäre π nicht T_{CF} -optimal). Dann gilt auch $RD_{k+1}(x) = \perp$.

α_k wird durch $t_k \leftarrow e$; $x \leftarrow t_k$; ersetzt. Es gilt dann $RD_{k+1}(t_k) = \perp$ und folglich auch $RD_{k+1}(x) = \perp$. Die Ersetzungen der Anweisungen $\alpha_{i_j} = x_{i_j} \leftarrow e$ durch $x_{i_j} \leftarrow t_k$ führen daher ebenfalls zu undefinierten x_{i_j} .

Es ergeben sich folglich keine zusätzlichen Optimierungsmöglichkeiten bei darauffolgender Anwendung der Konstantenfaltung.

3. In π gelte für eine Anweisung $\alpha_k = x \leftarrow e$:

$$\exists y \in V_e, \exists z \in V_\pi \text{ s.d. } (y, z, d) \in CP_k \text{ mit maximalem } d$$

d.h. $T_{CP}(\pi) \neq \pi$. Dann muß gelten:

- $RD_k(y) = \perp$, da sonst π nicht T_{CF} -optimal wäre.
- $RD_k(z) = \perp$, sonst wäre $(y, z, d) \notin CP_k$, da in der Kopieranweisung $\alpha_j = z' \leftarrow z$ (Anfang der zu y führenden Kopierkette) die Variable z durch eine Konstante ersetzt worden wäre.

Daraus folgt allerdings, daß nach der Ersetzung von y durch z keine weitere Verbesserung mit T_{CF} zu erreichen ist. \square

3.2.5 Wahl der Reihenfolge für die Gesamttransformation

In den vorangegangenen Abschnitten wurden die zwischen den klassischen Programmtransformationen gültigen Verbesserungsrelationen analysiert und deren Gültigkeit bewiesen. In Abbildung 3.1 sind sie nochmals zusammenfassend dargestellt. Es fällt auf, daß nie die starke Verbesserungsrelation gilt, was dazu führt, daß bei manchen Programmen gewisse Optimierungsschritte in einer Kette von Transformationen wirkungslos bleiben werden.

So erkennt man zum Beispiel, daß die Konstantenfaltung von keiner anderen Optimierung beeinflusst wird, d.h. sie eignet sich sehr gut als erster Optimierungsschritt. Die Dead Code Elimination

	T_{DC}	T_{CS}	T_{CF}	T_{CP}
T_{DC}	-	-	-	-
T_{CS}	-	-	-	\rightarrow
T_{CF}	\rightarrow	\rightarrow	-	-
T_{CP}	\rightarrow	\rightarrow	-	-

Abbildung 3.1: Verbesserungsrelationen zwischen den klassischen Verfahren

wiederum führt zu keiner Verbesserung der Optimierungsqualität der anderen Verfahren, man sollte sie also als letztes anwenden.

Für die Common-Subexpression Elimination und die Kopierpropagation ist die Situation etwas komplizierter, da sie sich wechselseitig beeinflussen. Eine wiederholte Anwendung beider Verfahren ist deshalb unumgänglich. Wir werden nun der Frage nachgehen, wieviele Iterationsschritte vonnöten sind.

Satz 3.2.10: *Es existiert eine Folge von LC-Programmen $(\pi_n)_{n \in \mathbb{N} \setminus \{0\}}$, so daß erst nach n -maliger Anwendung von $T := T_{CP} \circ T_{CS}$ keine Änderung mehr erfolgt.*

Beweis: Wähle $\pi_n := (\Sigma, (x), (y, z), \beta_n)$ mit $\Sigma = (F, \emptyset)$ und $F = \{f^{(2)}\}$ wobei β_n induktiv gegeben sei:

$$\begin{aligned}\beta_1 &:= y \leftarrow f(x, x); \\ & \quad z \leftarrow f(x, x);\end{aligned}$$

$$\begin{aligned}\beta_{n+1} &:= \beta_n; \\ & \quad y \leftarrow f(y, x); \\ & \quad z \leftarrow f(z, x);\end{aligned}$$

Zu zeigen:

$$T^{n+1}(\pi_n) = T^n(\pi_n), \text{ aber } \nexists i < n : T^{i+1}(\pi_n) = T^i(\pi_n)$$

Es gilt $\beta_n = \alpha_1; \dots; \alpha_{2n+2}$. Die erste Anwendung von T_{CS} auf π_n ergibt:

$$\begin{aligned}t_1 &\leftarrow f(x, x); \\ y &\leftarrow t_1; \\ z &\leftarrow t_1; \\ y &\leftarrow f(y, x); \\ z &\leftarrow f(z, x); \\ &\vdots\end{aligned}$$

Es sind keine weiteren Ersetzungen möglich, da die Ausdrücke ab Anweisung 4 nie verfügbar werden (die *kill*-Funktion entfernt den entsprechenden Index sofort nach dem Einfügen durch *gen* wieder).

Nun substituiert die Kopierpropagation jeweils das erste f -Argument in den Anweisungen 4 und 5:

$$\begin{aligned}
 t_1 &\leftarrow f(x, x); \\
 y &\leftarrow t_1; \\
 z &\leftarrow t_1; \\
 y &\leftarrow f(t_1, x); \\
 z &\leftarrow f(t_1, x); \\
 y &\leftarrow f(y, x); \\
 z &\leftarrow f(z, x); \\
 &\vdots
 \end{aligned}$$

Auch hier ist keine weitere Optimierung möglich, da die Tupel $(y, x, 1)$ und $(z, x, 1)$ aus den CP -Informationen wegen der Neudefinition von y und z ab Anweisung 6 ungültig sind.

Die Anweisungen 4 bis $2n + 2$ besitzen nun (bis auf Variablenumbenennungen) wieder die gleiche Struktur wie zu Anfang. Es werden also nach jeder Anwendung der Transformation T genau zwei aufeinander folgende Ausdrücke ersetzt. Da das Programm $2n$ Operationsausdrücke enthält, sind insgesamt genau n Anwendungen nötig bis keine Änderung mehr erfolgt. \square

Erwähnenswert ist, daß es auf die Reihenfolge von T_{CS} und T_{CP} nicht ankommt. Im obigen Beweis wurde zuerst T_{CS} angewendet, da im Programm zuerst eine Kopierpropagation möglich war. Man kann aber genausogut eine Programmfolge angeben, bei dem eine Common Subexpression Elimination als erster Schritt notwendig ist.

Beispiel 3.2.11: Wähle dazu statt β_1 im obigen Programm

$$\begin{aligned}
 \beta'_1 &:= y \leftarrow x; \\
 &z \leftarrow x;
 \end{aligned}$$

Man sieht sofort, daß man hier mit T_{CP} beginnen muß.

Daher ist ggf. eine zusätzliche Anwendung der Komposition der beiden Transformationen einzuplanen, denn die erste Transformation könnte wirkungslos bleiben. Es wäre interessant zu wissen, ob es eine obere Schranke für die Anzahl der Iterationen von $T_{CS} \circ T_{CP}$ gibt.

Satz 3.2.12 (Obere Schranke für Anwendungszahl von $T_{CS} \circ T_{CP}$): Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$ und $\alpha_i := x_i \leftarrow e_i$ gegeben. Setze

$$m := \left| \{i \in \{1, \dots, n\} \mid e_i = f(u_1, \dots, u_r) \text{ für ein } f \in F^{(r)} \text{ und } u_j \in V \cup C\} \right|.$$

Dann gilt für $T := T_{CS} \circ T_{CP}$:

$$T^{m+1}(\pi) = T(T^{m+1}(\pi))$$

Beweis: m repräsentiert die Anzahl der Operationsausdrücke im Programm π . Nach jeder Anwendung der Common Subexpression Elimination, bei der nicht $T_{CS}(\pi) = \pi$ gilt, d.h. noch Veränderungen stattfinden, enthält das Programm mindestens einen komplexen Ausdruck weniger.

Da T_{CP} idempotent ist, folgt, daß in dem Fall, daß T_{CS} keine Änderung mehr bewirkt, auch zwei aufeinanderfolgende Anwendungen von T_{CP} kein zusätzliches „Optimierpotential“ bezüglich T_{CS} erzeugen können (dies gilt natürlich analog für den symmetrischen Fall, daß sich zuerst ein T_{CP} -optimales Programm ergibt).

$T_{CF}(\pi)$	$T_{CS}(T_{CF}(\pi))$	$T_{CP}(T_{CS}(T_{CF}(\pi)))$	$T_{DC}(T_{CP}(T_{CS}(T_{CF}(\pi))))$
$u \leftarrow 3;$	$u \leftarrow 3;$	$u \leftarrow 3;$	
$v \leftarrow x - y;$	$\mathbf{t_2} \leftarrow \mathbf{x} - \mathbf{y};$	$t_2 \leftarrow x - y;$	$t_2 \leftarrow x - y;$
	$v \leftarrow \mathbf{t_2};$	$v \leftarrow t_2;$	
$w \leftarrow 4;$	$w \leftarrow 4;$	$w \leftarrow 4;$	
$x \leftarrow x - y;$	$x \leftarrow \mathbf{t_2};$	$x \leftarrow t_2;$	
$u \leftarrow x - y;$	$u \leftarrow x - y;$	$u \leftarrow \mathbf{t_2} - y;$	$u \leftarrow t_2 - y;$
$x \leftarrow u * 4;$	$x \leftarrow u * 4;$	$x \leftarrow u * 4;$	
$u \leftarrow 2 * u;$	$u \leftarrow 2 * u;$	$u \leftarrow 2 * u;$	$u \leftarrow 2 * u;$
$v \leftarrow 3;$	$v \leftarrow 3;$	$v \leftarrow 3;$	$v \leftarrow 3;$

Abbildung 3.2: Anwendung von T_{COPT} auf π aus Abbildung 1.1

Es wird also nach maximal $m + 1$ Iterationen ein Fixpunkt von T erreicht, weil die erste Anwendung von T_{CS} u.U. wirkungslos bleibt. \square

Die oben angegebene untere Schranke läßt sich u.U. noch weiter verfeinern. Da T_{CS} bei jeder „erfolgreichen“ Anwendung mindestens zwei komplexe Ausdrücke optimiert, und die den temporären Variablen zugewiesenen Ausdrücke keine weiteren gültigen Wiederholungen haben dürften, ist zu vermuten, daß $\lfloor \frac{m}{2} \rfloor + 1$ Applikationen von T ausreichen.

In der Praxis ist es sicherlich sinnvoll, nicht „blind“ die im schlechtesten Fall nötige Anzahl an Applikationen zu verwenden, sondern ein bedarfsgesteuertes Verfahren einzusetzen.

Definition 3.2.13 (T_{CPCS} als Fixpunktiteration): Für $\pi \in \mathcal{LC}$ und $T := T_{CP} \circ T_{CS}$ sei die Transformation $T_{CPCS} : \mathcal{LC} \rightarrow \mathcal{LC}$ definiert durch:

$$T_{CPCS}(\pi) := \begin{cases} \pi & \text{falls } T(\pi) = \pi \\ T_{CPCS}(T(\pi)) & \text{sonst} \end{cases}$$

T_{CS} und T_{CP} werden also solange im Wechsel angewendet, bis ein Fixpunkt erreicht ist. Dieser existiert gemäß Satz 3.2.12.

Aus den bisherigen Beobachtungen und der Tabelle in Abbildung 3.1 können wir nun eine Reihenfolge für die Teiltransformationen ableiten, um einen möglichst guten Optimierungseffekt zu erzielen:

1. Konstantenfaltung (nicht von anderen Transformationen verbesserbar)
2. Wiederholte Anwendung von Common Subexpression Elimination und Kopierpropagation im Wechsel
3. Dead Code Elimination (wird durch T_{CP} und T_{CF} begünstigt)

Eine andere Reihenfolge wäre ungünstig, z.B. würde ein Vorziehen der Dead Code-Elimination an den Anfang dazu führen, daß durch die anderen Transformationen entstehender Dead Code nicht entfernt werden kann. Es wäre aber denkbar, die Dead Code Elimination zusätzlich direkt nach der Konstantenfaltung (also vor T_{CPCS}) anzuwenden, da durch die Entfernung von Anweisungen in manchen Fällen der Fixpunkt von T_{CPCS} etwas schneller erreicht wird. Dies würde allerdings auch die Komplexität (vor allem beweistechnisch) erhöhen, daher soll davon abgesehen werden.

Definition 3.2.14 (Gesamttransformation): $T_{COPT} := T_{DC} \circ T_{CPCS} \circ T_{CF}$ sei die aus den klassischen Verfahren zusammengesetzte Gesamttransformation.

Beispiel 3.2.15: In Abbildung 3.2 ist ein Rechenbeispiel für die T_{COPT} -Transformation für unser Beispielprogramm aus Abbildung 1.1 dargestellt, wobei als Ausgangspunkt das bereits T_{CF} -optimierte Programm aus Beispiel 2.1.18 dient. Aus Übersichtlichkeitsgründen wurde hierbei auf eine detaillierte Auflistung der einzelnen Analyseinformationen verzichtet.

Bereits nach einer Anwendung von $T_{CP} \circ T_{CS}$ wird hier der Fixpunkt erreicht (erkennbar daran, daß in $T_{CP}(T_{CS}(T_{CF}(\pi)))$ keine zwei gleichen Operationsausdrücke mehr existieren).

Es ist nun noch zu prüfen, ob T_{COPT} die Anforderungen von Definition 2.0.15 erfüllt.

Satz 3.2.16: T_{COPT} ist eine Programmtransformation.

Beweis: Es sind drei Eigenschaften zu zeigen, die aber alle aus bereits bewiesenen Aussagen folgen:

1. T_{COPT} ist korrekt.

Da T_{COPT} aus korrekten Verfahren aufgebaut wurde (die Korrektheit der klassischen Verfahren wurde in Abschnitt 2.1 gezeigt), ist die Gesamttransformation ebenfalls korrekt.

2. T_{COPT} ist idempotent.

T_{CF} ist nach Lemma 3.2.9 von keinem der klassischen Verfahren verbesserbar. Daher ist

$$T_{CF}(T_{COPT}(\pi)) = T_{COPT}(\pi) \quad (*)$$

Die nun folgende Anwendung von T_{CP} kann wegen (*), der Idempotenz von T_{CP} (nach Definition) und aufgrund der beiden Korollare 3.2.6 und 3.2.8, nach denen weder $T_{DC} \rightarrow T_{CS}$ noch $T_{DC} \rightarrow T_{CP}$ gilt, ebenfalls nichts mehr ändern (wenn (*) nicht gelten würde, wäre nach Korollar 3.2.6 eine Veränderung möglich gewesen, da $T_{CF} \rightarrow T_{CS}$). Wir erhalten also:

$$T_{CP}(T_{CF}(T_{COPT}(\pi))) = T_{COPT}(\pi) \quad (**)$$

Die nachfolgende Dead Code Elimination führt wegen (**) und der Idempotenz von T_{DC} ebenfalls zu keiner Änderung. Also ist T_{COPT} idempotent.

3. T_{COPT} arbeitet deterministisch.

Die Aussage folgt aus dem Determinismus der Teilverfahren und daraus, daß nach Satz 3.2.12 eine obere Schranke für die Anzahl der Anwendungen von $T_{CS} \circ T_{CP}$ existiert (sonst wäre die Terminierung gefährdet). \square

3.3 Simulation des DAG-Verfahrens durch Einzeltransformationen

In diesem Abschnitt werden wir versuchen, aus Einzeltransformationen ein Verfahren zu konstruieren, das der DAG-Optimierung bezüglich Codeeffizienz gleichkommt. Dies ist insbesondere auch deswegen interessant, da sich die klassischen Transformationen relativ einfach für iterative Programme erweitern lassen [2], während die DAG-Optimierung hier konstruktionsbedingt passen muß. Die Gesamttransformation T_{COPT} käme als ein Kandidat für eine solche zusammengesetzte Transformation in Frage.

Die direkte Gleichheit $T_{COPT} = T_{DAG}$ kann offensichtlich nicht gelten, da sich schon aufgrund von Variablenbenennungen Unterschiede ergeben (man möge dazu etwa das T_{COPT} -optimierte Programm aus Abbildung 3.2 mit dem DAG-optimierten Programm aus Abbildung 2.3 vergleichen).

Algorithmus 2.2.5 benennt die Variablen (bis auf die Ein-/Ausgabevariablen) sukzessive um, während T_{COPT} die im Programm vorhandenen Variablen weiterverwendet. Dies im DAG-Verfahren zu ändern wäre, falls überhaupt möglich, mit erheblichem Aufwand verbunden. Daher muß ein anderer Ansatz gewählt werden.

Definition 3.3.1 (Äquivalenz von Programmtransformationen): Seien $T_1, T_2 : \mathcal{LC} \rightarrow \mathcal{LC}$ zwei Programmtransformationen. T_1 heißt genau dann äquivalent zu T_2 ($T_1 \sim T_2$), wenn für alle $\pi \in \mathcal{LC}$ gilt:

$$T_2(T_1(\pi)) = T_1(\pi) \wedge T_1(T_2(\pi)) = T_2(\pi)$$

Zwei Programmtransformationen sind folglich äquivalent, wenn sie zueinander optimal sind, also die Anwendung von T_2 auf ein T_1 -optimales Programm keine Änderungen mehr bewirkt und umgekehrt.

3.3.1 Mängel der Gesamttransformation

Man erkennt, daß auch diese Eigenschaft für unsere Kandidaten T_{DAG} und T_{COPT} nicht gelten kann, da bei einer „Verschaltung“ der Form $T_{DAG}(T_{COPT}(\pi))$ die DAG-Optimierung immer noch Veränderungen an $T_{COPT}(\pi)$ vornehmen wird; das Variablenbenennungsproblem besteht nämlich nach wie vor.

Andererseits lieferte die zusammengesetzte Gesamttransformation in Beispiel 3.2.15 bis auf die Variablenbenennung das gleiche Resultat wie das DAG-Verfahren. Es liegt also die Vermutung nahe, daß durch eine Variableneubenennung nach der Anwendung von T_{COPT} ein zu T_{DAG} äquivalentes Verfahren konstruiert werden kann.

3.3.1.1 Variablensubstitution als Programmtransformation

Wir wollen also zur Beseitigung des oben beschriebenen Problems eine Programmtransformation angeben, die ein Programm so transformiert, daß sich eine zum DAG-Verfahren passende Variablenbenennung und damit Code in SSA-Form ergibt.

Definition 3.3.2 (SSA-Transformation): Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$ und $\alpha_i := x_i \leftarrow e_i$. Definiere für $i \in \{0, \dots, n\}$ Substitutionsfunktionen $\rho_i : V \rightarrow V$:

$$\rho_0 = id$$

$$\text{und für } i \in \{1, \dots, n\} : \rho_i = \begin{cases} \rho_{i-1}[x_i/x_i] & \text{falls } x_i \in V_{out} \text{ und } x_i \neq x_j \text{ für } j > i \\ \rho_{i-1}[x_i/v_i] & \text{sonst} \end{cases}$$

Hierbei seien die v_i paarweise verschiedene Variablen, die nicht in π auftreten.

Die Transformation $T_{SSA} : \mathcal{LC} \rightarrow \mathcal{LC}$ berechnet sich wie folgt:

$$T_{SSA}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$$

$$\text{mit } \beta' := \rho_1(x_1) \leftarrow \bar{\rho}_0(e_1);$$

$$\vdots$$

$$\rho_k(x_k) \leftarrow \bar{\rho}_{k-1}(e_k);$$

$$\vdots$$

$$\rho_n(x_n) \leftarrow \bar{\rho}_{n-1}(e_n);$$

hierbei ist $\bar{\rho}_i$ die monotone Fortsetzung von ρ_i (wie in Definition 2.3.2 für δ definiert).

Von T_{SSA} werden alle linksseitigen Variablen, außer Ausgabevariablen in ihrer letzten Zuweisung, in neue Variablen umbenannt. Dadurch erhält man offensichtlich ein Programm in SSA-Form. Für die v_i gibt i immer den Index der Anweisung ihres ersten Auftretens an.

Beispiel 3.3.3: Wir wollen die Definition anhand eines Beispiels demonstrieren, dazu verwenden wir $T_{COPT}(\pi)$ aus Beispiel 3.2. Für die ρ_i werden nur diejenigen Argumente angegeben, die nicht auf sich selbst abgebildet werden. Es ergibt sich folgende Berechnung (zur Erinnerung: $\vec{v}_{out} = (u, v)$):

$T_{COPT}(\pi)$	ρ_i	$T_{SSA}(T_{COPT}(\pi))$
$t_2 \leftarrow x - y;$	$\{t_2 \mapsto v_1\}$	$\rho_1(t_2) \leftarrow \bar{\rho}_0(x - y); = v_1 \leftarrow x - y;$
$u \leftarrow t_2 - y;$	$\{t_2 \mapsto v_1, u \mapsto v_2\}$	$\rho_2(u) \leftarrow \bar{\rho}_1(t_2 - y); = v_2 \leftarrow v_1 - y;$
$u \leftarrow 2 * u;$	$\{t_2 \mapsto v_1\}$	$\rho_3(u) \leftarrow \bar{\rho}_2(2 * u); = u \leftarrow 2 * v_2;$
$v \leftarrow 3;$	$\{t_2 \mapsto v_1\}$	$\rho_4(v) \leftarrow \bar{\rho}_3(3); = v \leftarrow 3;$

Das Resultat ist identisch zu $T_{DAG}(\pi)$ (Abbildung 2.3). Mit der Idempotenz der DAG-Optimierung (Satz 2.2.15) folgt daher $T_{DAG}(T_{SSA}(T_{COPT}(\pi))) = T_{SSA}(T_{COPT}(\pi))$ für dieses Beispielprogramm.

Für die SSA-Transformation werden wir ausnahmsweise einmal nicht alle Eigenschaften von Programmtransformationen einzeln überprüfen, denn Korrektheit, Idempotenz und Determinismus sind offensichtlich. Da das Verfahren entwickelt wurde, um „DAG-kompatible“ Variablennamen zu erhalten, ist T_{DAG} optimal bezüglich T_{SSA} .

Auch für iterative Programme mit Verzweigungen und Sprüngen läßt sich eine SSA-Form berechnen [13, 14], welche für viele globale Optimierungen in iterativen Programmen hilfreich ist.

3.3.1.2 Nicht entfernbare Kopieranweisungen

Wir könnten nun versuchen, die Äquivalenz von $T_{SSA} \circ T_{COPT}$ und T_{DAG} zu zeigen. Dieser Beweisversuch wäre jedoch zum Scheitern verurteilt.

Satz 3.3.4: *Es gilt $T_{SSA} \circ T_{COPT} \not\sim T_{DAG}$, d.h. es gibt ein $\pi \in \mathcal{LC}$, so daß*

$$T_{DAG}(T_{SSA}(T_{COPT}(\pi))) \neq T_{SSA}(T_{COPT}(\pi)).$$

Beweis: Betrachten wir das Programm π_1 (mit den Ausgabevariablen y und z) aus dem Beweis von Satz 3.2.10 und wenden $T_{SSA} \circ T_{COPT}$ auf π_1 und schließlich T_{DAG} auf das erhaltene Ergebnis an. Es ergeben sich folgende Resultate:

π_1	$T_{SSA}(T_{COPT}(\pi_1))$	$T_{DAG}(T_{SSA}(T_{COPT}(\pi_1)))$
	$v_1 \leftarrow f(x, x);$	
$y \leftarrow f(x, x);$	$y \leftarrow v_1;$	$y \leftarrow f(x, x);$
$z \leftarrow f(x, x);$	$z \leftarrow v_1;$	$z \leftarrow y;$

Damit ist die Behauptung bewiesen. □

Die Common Subexpression Elimination sorgt für die Einführung einer neuen temporären Variable, die dann in Kopieranweisungen den Ausgabevariablen zugewiesen wird. Die Kopierpropagation kann diese Kopieranweisungen nicht rückgängig machen, da sozusagen rückwärtig substituiert werden müßte.

Dieser Fall kann übrigens nur auftreten, wenn die Kopierzuweisungen an Ausgabevariablen erfolgen, die später keinen neuen Wert erhalten. Denn sonst ist eine Propagation möglich und die entsprechenden Kopieranweisungen werden Dead Code.

Bei der Anwendung des DAG-Verfahrens auf $T_{SSA}(T_{COPT}(\pi_1))$ erhält man hingegen nur einen Operationsknoten k_1 , auf den $val(y, 3)$ und $val(z, 3)$ verweisen. Bei der Codegenerierung mittels Algorithmus 2.2.5 wird folglich in Schritt $\delta(k_1) := y$ gesetzt, da $last(y) = 2 < last(z) = 3$. Dadurch wird eine Kopieranweisung bei der Codegenerierung eingespart.

3.3.1.3 Rückwärtige Kopierpropagation

Um trotz der im vorangegangenen Abschnitt aufgetauchten Probleme ein T_{DAG} -äquivalentes Verfahren zu erhalten, bieten sich folgende Lösungsansätze an:

1. Änderung der DAG-Codegenerierung (Algorithmus 2.2.5), so daß keine Umbenennungen von Knotennamen in Ausgabevariablen mehr stattfinden und für jede Ausgabevariable eine Kopieranweisung eingefügt wird.
2. Modifikation der Common Subexpression Elimination zur Verhinderung der Einfügung von unnötigen Kopieranweisungen.
3. Entwurf eines neuen Verfahrens, das in Kombination mit den bisherigen eine T_{DAG} -äquivalente Transformation ermöglicht. Es sollte die im Beweis von Satz 3.3.4 auftretenden „Anomalien“ eliminieren.

Der erste Ansatz ist sicherlich der am wenigsten sinnvolle, denn ein Optimierungsverfahren abzuschwächen und dadurch ein schlechteres Resultat als das mögliche in Kauf zu nehmen, ist wenig wünschenswert. Die Modifikation der Common Subexpression Elimination würde sich recht kompliziert gestalten, da in manchen Fällen das Einfügen von zwei Kopieranweisungen unvermeidbar ist, um die Korrektheit zu gewährleisten. Es wäre dann u.U. ein zusätzlicher Analyseschritt nötig und die bisherigen Untersuchungen zur Common Subexpression Elimination (Korrektheit, Idempotenz, Verhalten im Hinblick auf die anderen Verfahren) müßten erneut durchgeführt werden.

Im folgenden wollen wir also den dritten Ansatz verfolgen und eine Programmtransformation entwerfen, die die zusätzlichen Kopierzusweisungen an Ausgabevariablen nachträglich entfernt. Das Verfahren arbeitet mit einer Rückwärtsanalyse.

Definition 3.3.5: Sei $\pi = (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ ein LC-Programm mit $\Sigma = (F, C)$, $\beta = \alpha_1; \dots; \alpha_n$ und $\alpha_i = x_i \leftarrow e_i$. Definiere für eine Anweisung α_i und $I := \mathfrak{P}((V_\pi \times V_\pi \times \{1, \dots, n\}) \cup V_\pi)$ die Transferfunktion $t_{\alpha_i} : I \rightarrow I$ durch:

$$t_{\alpha_i} := gen_{\alpha_i} \circ kill_{\alpha_i} \text{ mit } kill, gen : I \rightarrow I$$

$$kill_{\alpha_i}(M) := M \setminus \{(y, z, j) \in M \mid x_i = y \text{ oder } z \in V_{\alpha_i}, j \in \{1, \dots, n\}\}$$

$$gen_{\alpha_i}(M) := M \cup \{(y, x_i, i) \mid e_i = y \in V \setminus (M \cup V_{out}) \text{ und } x_i \in V_{out} \setminus M\} \cup \{x_i\}$$

Die Transferfunktionen $t_{\alpha_n}, \dots, t_{\alpha_1}$ bestimmen nun ausgehend von der Startinformation \emptyset die Mengen der Analyseinformationen:

$$RC_n := \emptyset$$

$$RC_i := t_{\alpha_{i+1}}(RC_{i+1}) \text{ für } i \in \{n-1, \dots, 1\}$$

In der Analysemenge RC_i einer Anweisung α_i ist also genau dann das Tupel (y, x, j) enthalten, falls in einer Anweisung α_j , $j > i$ eine Zuweisung der Form $x \leftarrow y$ mit $x \in V_{out}$ und $y \notin V_{out}$ erfolgt ist, y nach α_i auf keiner linken Seite auftaucht, und x ebenfalls außer in α_j nach α_i nicht neudefiniert wird sowie zwischen α_i und α_j nicht verwendet wird. Der dritte Eintrag j wird im folgenden Algorithmus zur Garantie des Determinismus benötigt.

Das zusätzliche „Aufsammeln“ der linksseitigen Variablen dient zur einfachen Überprüfung, ob eine Variable später (nach α_j) neudefiniert wird.

Algorithmus 3.3.6 (Rückwärtige Kopierpropagation): Sei π wie in Definition 3.3.5 und eine Variablensubstitution $\delta : V_\pi \rightarrow V_\pi$ gegeben. Definiere $T_{RC} : \mathcal{LC} \rightarrow \mathcal{LC}$ durch $T_{RC}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$, wobei β' wie folgt berechnet wird:

1. Setze $\delta := id$ und $\beta' := \varepsilon$.
2. Für $i = 1, \dots, n$ (Fallunterscheidung nach dem Typ von e_i):
 - a) $e_i = f(u_1, \dots, u_r)$ für ein $f \in F^{(r)}$:
 Falls ein $y \in V_{out}$ existiert, so daß $(x_i, y, j) \in RC_i$ (wenn mehrere solcher y existieren, wähle das Tupel mit dem kleinsten j)³ und setze

$$\begin{aligned}\beta' &:= \beta' \cdot y \leftarrow \bar{\delta}(e_i) \\ \delta &:= \delta[x_i/y]\end{aligned}$$

b) Sonst: Weitere Fallunterscheidung

- (i) $e_i = y \in V$ und $\delta(y) = \delta(x_i)$: Keine Änderung von β' .
- (ii) Sonst: Setze $\beta' := \beta' \cdot \bar{\delta}(\alpha_i)$.

Man erkennt, daß keine transitiven Beziehungen berücksichtigt werden. Das ist zur Beseitigung der in Abschnitt 3.3.1.2 festgestellten Anomalie aber auch gar nicht notwendig. Zur Verdeutlichung wollen wir den Algorithmus anhand eines Beispiels demonstrieren:

Beispiel 3.3.7: Als Ausgangsprogramm verwenden wir die T_{COPT} -optimierte Fassung des Programms π_2 aus dem Beweis des Satzes 3.2.10, d.h. $\pi := T_{COPT}(\pi_2) := ((\{f^{(2)}\}, \emptyset), (x), (y, z), \beta)$ mit

$$\begin{aligned}\beta &:= t_1 \leftarrow f(x, x); \\ & t_4 \leftarrow f(t_1, x); \\ & y \leftarrow t_4; \\ & z \leftarrow t_4;\end{aligned}$$

Es ergibt sich dann folgende Rechnung (sei $\pi' := T_{RC}(\pi) := ((\{f^{(2)}\}, \emptyset), (x), (y, z), \beta')$):

i	β	RC_i	δ	β'
1	$t_1 \leftarrow f(x, x);$	$\{t_4, y, z\}$	id	$t_1 \leftarrow f(x, x);$
2	$t_4 \leftarrow f(t_1, x);$	$\{(t_4, y, 3), (t_4, z, 4), y, z\}$	$id[t_4/y]$	$y \leftarrow f(t_1, x);$
3	$y \leftarrow t_4;$	$\{(t_4, z, 4), z\}$	$id[t_4/y]$	
4	$z \leftarrow t_4;$	\emptyset	$id[t_4/y]$	$z \leftarrow y;$

Wenn man nun auf π' die Transformation T_{SSA} anwendet und danach das DAG-Verfahren auf das Resultat, erhält man wieder π'^4 . Für dieses Beispiel gilt wurde also der erwünschte Effekt durch die Einführung der rückwärtigen Kopierpropagation erzielt.

Die rückwärtige Kopierpropagation führt nur Substitutionen für Variablen durch, die keine Ausgabevariablen sind. Dies ist für die Vollständigkeit des Programms (Wohldefiniertheit der Variablen) unabdingbar, denn sonst könnte durch den Wegfall einer Zuweisung an eine Ausgabevariable diese undefiniert sein.

Das Verfahren reicht jedoch aus, um das oben beschriebene Phänomen der zusätzlichen Kopierzweisungen an Ausgabevariablen zu beseitigen, da die Common Subexpression Elimination neue

³Zur Einhaltung des Determinismus notwendig.

⁴Auf eine Demonstration wird hier verzichtet. Der Leser möge dies selbst nachprüfen.

Variablen einführt, die nur einmal einen Wert erhalten und selbstverständlich keine Ausgabevariablen sein können.

Wie bei allen klassischen Verfahren außer der Konstantenfaltung gilt auch für T_{RC} die starke Äquivalenz von Quell- und Zielprogramm.

Satz 3.3.8 (Korrektheit): Für alle $\pi \in \mathcal{LC}$ gilt $\pi \sim T_{RC}(\pi)$.

Beweis: Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\Sigma := (F, C)$, $\beta := \alpha_1; \dots; \alpha_n$ sowie $\alpha_i = x_i \leftarrow e_i$ und $T_{RC}(\pi) := \pi' := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$. Wir zeigen für alle $y \in V_{out}$, daß $t_\pi(y) = t_{\pi'}(y)$ gilt. Nach Satz 1.2.6 folgt dann die Behauptung.

Wir betrachten o.B.d.A. den vereinfachten Fall, daß T_{RC} nur eine Variable substituiert. Dies reicht aus, da die Ersetzungen unabhängig voneinander sind, d.h. man könnte den Algorithmus so modifizieren, daß er bei jeder Anwendung nur eine Substitution vornimmt (damit wäre natürlich keine Idempotenz gegeben).

Es liegt also folgende Situation vor:

1. Es existiert ein $i \in \{1, \dots, n\}$, so daß $e_i = f(u_1, \dots, u_r)$ für ein $f \in F^{(r)}$.
2. $(x_i, x_j, j) \in RC_i$ für ein $j > i$ und $\nexists (x_i, v, k) \in RC_i$ mit $k < j$.
3. $x_i \notin V_{out}$, $x_j \in V_{out}$.
4. $\alpha_j = x_j \leftarrow x_i$.
5. $x_j \notin \bigcup_{k=j+1}^n \{x_k\} \cup \bigcup_{k=i+1}^{j-1} \{x_k\} \cup \bigcup_{k=i+1}^j \{V_{e_k}\}$, da sonst 2. aufgrund von Definition 3.3.5 nicht gelten würde.
6. Aus 2. folgt ebenfalls: $x_i \notin \bigcup_{k=i+1}^n \{x_k\}$

Für β' ergibt sich dann gemäß Algorithmus 3.3.6:

$$\begin{aligned}
 x_1 &\leftarrow e_1; \\
 x_2 &\leftarrow e_2; \\
 &\vdots \\
 x_{i-1} &\leftarrow e_{i-1}; \\
 x_j &\leftarrow e_i; \\
 x_{i+1} &\leftarrow \bar{\delta}(e_{i+1}); \quad (\text{mit } \delta = id[x_i/x_j]) \\
 &\vdots \\
 x_{j-1} &\leftarrow \bar{\delta}(e_{j-1}); \\
 x_{j+1} &\leftarrow \bar{\delta}(e_{j+1}); \\
 &\vdots \\
 x_n &\leftarrow \bar{\delta}(e_n);
 \end{aligned}$$

Wegen 6. reicht es für $k > i$, $k \neq j$, $\bar{\delta}$ nur auf e_k und nicht die gesamte Anweisung α_k anzuwenden.

Nach Definition 1.2.2 gilt damit für die Termdarstellung von π bezüglich $y \in V_{out}$:

$$\begin{aligned}
t_\pi(y) &= y[x_n/e_n] \dots [x_{j+1}/e_{j+1}][x_j/e_j][x_{j-1}/e_{j-1}] \dots [x_{i+1}/e_{i+1}][x_i/e_i][x_{i-1}/e_{i-1}] \dots [x_1/e_1] \\
&\stackrel{4}{=} y[x_n/e_n] \dots [x_{j+1}/e_{j+1}][x_j/\mathbf{x}_i][x_{j-1}/e_{j-1}] \dots [x_{i+1}/e_{i+1}][\mathbf{x}_i/\mathbf{x}_j][\mathbf{x}_j/\mathbf{e}_i][x_{i-1}/e_{i-1}] \dots [x_1/e_1] \\
&\stackrel{5,6}{=} y[x_n/e_n][\mathbf{x}_i/\mathbf{x}_j] \dots [x_{j+1}/e_{j+1}][\mathbf{x}_i/\mathbf{x}_j][x_j/x_i][\mathbf{x}_i/\mathbf{x}_j][x_{j-1}/e_{j-1}][\mathbf{x}_i/\mathbf{x}_j] \dots \\
&\quad [x_{i+1}/e_{i+1}][x_i/x_j][x_j/e_i][x_{i-1}/e_{i-1}] \dots [x_1/e_1] \\
&= y[x_n/e_n][x_i/x_j] \dots [x_{j+1}/e_{j+1}][\mathbf{x}_i/\mathbf{x}_j][x_{j-1}/e_{j-1}][x_i/x_j] \dots \\
&\quad [x_{i+1}/e_{i+1}][x_i/x_j][x_j/e_i][x_{i-1}/e_{i-1}] \dots [x_1/e_1] \\
&= y[x_n/\bar{\delta}(e_n)] \dots [x_{j+1}/\bar{\delta}(e_{j+1})][x_{j-1}/\bar{\delta}(e_{j-1})] \dots [x_{i+1}/\bar{\delta}(e_{i+1})][x_j/e_i][x_{i-1}/e_{i-1}] \dots [x_1/e_1] \\
&= t'_\pi(y)
\end{aligned}$$

T_{RC} arbeitet also korrekt. □

Damit die rückwärtige Kopierpropagation die Eigenschaften einer Programmtransformation gemäß Definition 2.0.15 erfüllt, muß zusätzlich die Idempotenz gelten. Der Determinismus der Transformation ist hingegen offensichtlich.

Korollar 3.3.9: Für alle $\pi \in \mathcal{LC}$ gilt $T_{RC}(T_{RC}(\pi)) = T_{RC}(\pi)$, d.h. T_{RC} ist idempotent.

Beweisskizze: Die Aussage ist klar, denn bei der ersten Anwendung von T_{RC} werden, überall, wo es gemäß der Definition von T_{RC} möglich ist, Ausdrücke der Form $t \leftarrow f(u_1, \dots, u_r)$ mit $t \notin V_{out}$ (*) durch $y \leftarrow f(u_1, \dots, u_r)$ ersetzt mit $y \in V_{out}$. Bei einer zweiten Applikation von T_{RC} sind dann aufgrund der Beschränkung (*) keine weiteren Ersetzungen möglich. □

Wir wollen uns abschließend der Frage widmen, ob die DAG-Optimierung T_{RC} -optimal ist. Diese Eigenschaft hatten wir für alle klassischen Transformationen in Abschnitt 3.1 gezeigt.

Satz 3.3.10: Es gilt $T_{RC}(T_{DAG}(\pi)) = T_{DAG}(\pi)$ für alle $\pi \in \mathcal{LC}$.

Beweis: Sei $T_{DAG}(\pi) := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\Sigma := (F, C)$ und $\beta := \alpha_1; \dots; \alpha_n$. Man nehme an, es gelte $T_{RC}(T_{DAG}(\pi)) \neq T_{DAG}(\pi)$. Dann gibt es eine Anweisung $\alpha_j = y \leftarrow x$ in $T_{DAG}(\pi)$, so daß $y \in V_{out}$ sowie eine Anweisung $\alpha_i = x \leftarrow f(u_1, \dots, u_r)$ für ein $f \in F^{(r)}$ und $i < j$. Außerdem muß $(x, y, j) \in RC_i$ gelten. Eine wichtige Voraussetzung für $(x, y, j) \in RC_i$ ist, daß $x \notin V_{out}$ gelten muß.

Bei der Codegenerierung für einen DAG (Algorithmus 2.2.5) werden Kopieranweisungen für Ausgabevariablen eingefügt, die noch keinen Wert haben (Schritte 3 oder 8). Sonst treten keinerlei Kopieranweisungen auf. Die Kopieranweisung α_j kann nur dann eingefügt werden, wenn bereits eine andere Ausgabevariable als linke Seite in α_i bei der Codegenerierung für den entsprechenden Operationsknoten eingefügt wurde (Modifikation der Substitutionsfunktion δ) oder gar keine Operationsknoten existieren (dann existiert auch α_i nicht in obiger Form). Für α_i gilt demnach $x \in V_{out}$ und es ergibt sich ein Widerspruch. □

Das neue Verfahren wurde speziell entworfen, um die Probleme von T_{COPT} zu beseitigen, also zusätzliche Kopierzweisungen an Ausgabevariablen zu entfernen. Daher sollte es genau nach T_{COPT} angewendet werden, denn es erzeugt keinen Dead Code. Die neue Reihenfolge wäre ist also jetzt:

$$T_{SSA} \circ T_{RC} \circ T_{COPT} =: T_{COPT}'$$

Die reine Umbenennungstransformation T_{SSA} wird zuletzt ausgeführt, denn sie soll die „Variablenkompatibilität“ zu T_{DAG} gewährleisten.

3.3.1.4 Schwachstelle: Propagation von Kopien

Man mag annehmen, daß mit der modifizierten Transformation $T_{COPT'}$ endlich die Äquivalenz zu T_{DAG} gilt. Dies ist jedoch immer noch nicht der Fall:

Satz 3.3.11: *Es existiert ein $\pi \in \mathcal{LC}$, so daß $T_{DAG}(T_{COPT'}(\pi)) \neq T_{COPT'}(\pi)$, also gilt nicht $T_{COPT'} \sim T_{DAG}$.*

Beweis: Konstruktion eines LC-Programms π als Gegenbeispiel:

$$\begin{aligned} \pi := ((\{f, g\}, \emptyset), (x), (y, z), \beta) \text{ mit } \beta := & t \leftarrow x; \\ & x \leftarrow f(x); \\ & y \leftarrow t; \\ & z \leftarrow g(x); \end{aligned}$$

Die Anwendung von $T_{COPT'}$ auf π und die nachfolgende Applikation von T_{DAG} ergeben:

$T_{COPT'}(\pi)$	$T_{DAG}(T_{COPT'}(\pi))$
$v_1 \leftarrow x;$	
$v_2 \leftarrow f(x);$	$v_1 \leftarrow f(x);$
$y \leftarrow v_1;$	$y \leftarrow x;$
$z \leftarrow g(v_2);$	$z \leftarrow g(v_1);$

Damit gilt die Äquivalenz nicht. □

Wollen wir den Ablauf der Berechnung von $T_{COPT'}$ auf π einmal etwas genauer betrachten, um die Schwachstelle zu identifizieren:

1. Es sind keine Konstanten im Programm vorhanden, also kann T_{CF} nichts verändern.
2. Es gibt keine Ausdrücke mit gleichem Operationssymbol, also bleibt auch T_{CS} wirkungslos.
3. T_{CP} kann keine Optimierungen durchführen, da das zunächst generierte Tupel $(t, x, 1)$ wegen der Anweisung $x \leftarrow f(x)$ wieder aus CP_2 gelöscht wird, also in CP_3 nicht mehr vorhanden ist. Es ist jedoch offensichtlich, daß y eine Kopie des „ersten“ x zugewiesen bekommt. Nach einer Umbenennung des linksseitigen x in der zweiten Anweisung (entsprechende Änderungen der Folgeanweisungen vorausgesetzt) könnte eine Optimierung durchgeführt werden.

Da T_{CPCS} keine Änderung bewirkt, bricht die Iteration (nach einem weiteren Schritt) ab.

4. Da kein Dead Code in π vorkommt, beläßt auch T_{DC} π unverändert.
5. T_{RC} bleibt wirkungslos, da die erste Anweisung keinen Operationsausdruck enthält.
6. T_{SSA} nimmt schließlich eine Umbenennung vor.

Die Kopierpropagation ist also nicht „stark“ genug. Interessant ist außerdem, daß eine zweite Anwendung von $T_{COPT'}$ auf $T_{COPT'}(\pi)$ eine Optimierung bewirken würde. Daraus folgt natürlich auch, daß $T_{COPT'}$ nicht idempotent ist und $T_{SSA} \rightarrow T_{CP}$ gilt.

Dieses Problem tritt offenbar nicht auf, wenn das Eingabeprogramm in SSA-Form vorliegt, d.h. für jede Zuweisung eine neue, bisher unbenutzte, Variable verwendet wird, denn dann können die Kopierketten nicht durch Überschreiben eines Wertes unterbrochen werden. Um diese Eigenschaft herzustellen reicht die Anwendung von T_{SSA} .

3.3.2 Erweiterte Gesamttransformation

Die Transformation $T_{COPT'}$ ist also so zu modifizieren, daß bei jeder Anwendung der Kopierpropagation ein Programm vorliegt, das die Single-Assignment-Eigenschaft erfüllt. Man könnte vor jede Anwendung der Kopierpropagation einen T_{SSA} -Schritt schalten, denn dann ist dies auf jeden Fall sichergestellt.

Es reicht jedoch aus, ganz am Anfang ein einziges Mal die SSA-Transformation auf das Eingabeprogramm anzuwenden. Die Gründe hierfür sind:

- Da T_{CF} keine neuen Anweisungen einfügt und nur Ersetzungen von rechtsseitigen Variablen und Ausdrücken durch Konstanten vornimmt, bleibt die SSA-Eigenschaft auch nach der Konstantenfaltung erhalten.
- Die Common Subexpression Elimination fügt zwar neue Anweisungen ein, diese verwenden jedoch neue, noch nicht im Programm vorhandene Zuweisungsvariablen. Daher erhält auch T_{CS} die SSA-Eigenschaft.
- T_{CP} nimmt wie die Konstantenfaltung nur Ersetzungen auf rechten Seiten von Zuweisungen vor. D.h. auch nach der Anwendung von T_{CP} liegt nach wie vor Code in SSA-Form vor (dies ist wichtig, da T_{CS} und T_{CP} ja mehrmals im Wechsel ausgeführt werden können und bei jeder Anwendung von T_{CP} die Eigenschaft noch erfüllt sein muß).

Damit sind nun die Voraussetzungen geschaffen, um eine neue Transformation zu definieren, die hoffentlich äquivalent zum DAG-Verfahren ist.

Definition 3.3.12 (Erweiterte Gesamttransformation): Für $\pi \in \mathcal{LC}$ sei die erweiterte Gesamttransformation $T_{XOPT} : \mathcal{LC} \rightarrow \mathcal{LC}$ definiert durch:

$$T_{XOPT} := T_{COPT'} \circ T_{SSA} = T_{SSA} \circ T_{RC} \circ T_{DC} \circ \underbrace{(T_{CP} \circ T_{CS})^*}_{T_{CPCS}} \circ T_{CF} \circ T_{SSA}$$

Hierbei soll die \star -Notation andeuten, daß solange iteriert wird, bis ein Fixpunkt (nach endlich vielen Schritten, siehe Satz 3.2.12) erreicht wird.

Bevor wir prüfen, ob T_{XOPT} eine Programmtransformation nach Definition 2.0.15 ist, muß noch die Gültigkeit einiger Verbesserungsrelationen, an denen die rückwärtige Kopierpropagation beteiligt sein könnte, geklärt werden:

Lemma 3.3.13: *Es gelten die folgenden Beziehungen:*

1. $T_{RC} \not\rightarrow T_{DC}$
2. $T_{RC} \rightarrow T_{CS}$
3. $T_{RC} \rightarrow T_{CP}$
4. $T_{RC}(T_{CP}(\pi)) = T_{CP}(\pi)$ für alle $\pi \in \mathcal{LC}$, die in SSA-Form vorliegen.
5. $T_{RC}(T_{CPCS}(\pi)) = T_{CPCS}(\pi)$ für alle $\pi \in \mathcal{LC}$ in SSA-Form.
6. $T_{RC} \not\rightarrow T_{CF}$

Beweis:

1. Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$ ein T_{DC} -optimales Programm und in π existiere eine Operationsanweisung $\alpha_k = x \leftarrow f(u_1, \dots, u_r)$ sowie eine Kopieranweisung $\alpha_l = y \leftarrow x$ mit $l > k$ und $y \in V_{out}$, $x \notin V_{out}$ und es gelte $(x, y, l) \in RC_k$ ($\nexists (x, v, m) \in RC_k$

mit $m < l$). Dann wird α_k durch $\alpha'_k := y \leftarrow f(u_1, \dots, u_r)$ ersetzt und α_l aus dem Programm entfernt.

Annahme: Durch die Entfernung von α_l entsteht Dead Code.

Es gibt also eine Anweisung $\alpha_i = z \leftarrow e$ mit $z \notin LV'_i$, wobei LV'_i die Live Variable-Analysemengen in $\pi' := T_{RC}(\pi)$ und LV_i jene in π seien. Für $k \leq i < l$ gilt nun $x \notin LV'_i$ und $y \in LV'_i$, während in π $t \in LV_i$ und $y \notin LV_i$ galt. Für $i < k$ bzw. $i > l$ ergibt sich aufgrund obiger Modifikation keine Änderung.

Da aber nach Definition 3.3.5 bei $(x, y, l) \in RC_k$ gilt, daß x in α_i , $i > k$ nicht als Zuweisungsvariable benutzt wird, kann durch die Änderung kein zusätzlicher Dead Code auftreten.

2. Gegeben sei das folgendes T_{CS} -optimale Programm π :

$$\begin{aligned} \pi := ((\{f, g\}, \emptyset), (x), (y_1, y_2, y_3), \beta) \text{ mit } & \beta := x \leftarrow f(x); \\ & y_1 \leftarrow x; \\ & y_2 \leftarrow g(x); \\ & y_3 \leftarrow g(y_1); \end{aligned}$$

Nach der Anwendung von T_{RC} erhält man

$$\begin{aligned} \pi' := T_{RC}(\pi) = ((\{f, g\}, \emptyset), (x), (y_1, y_2, y_3), \beta') \text{ mit } & \beta' := y_1 \leftarrow f(x); \\ & y_2 \leftarrow g(y_1); \\ & y_3 \leftarrow g(y_1); \end{aligned}$$

Eine Anwendung von T_{CS} auf π' führt nun zu einer entsprechenden Optimierung.

3. Eine Verbesserung der Kopierpropagation ist bei dem untenstehenden T_{CP} -optimalen Programm π möglich:

$$\begin{aligned} \pi := ((\{f, g\}, \emptyset), (x), (y, z), \beta) \text{ mit } & \beta := t \leftarrow x; \\ & x \leftarrow f(x); \\ & y \leftarrow x; \\ & z \leftarrow g(t); \end{aligned}$$

Anweisung 2 sorgt durch Überschreiben von x dafür, daß das entsprechende Tupel $(t, x, 1)$ wieder gelöscht wird. Die Angabe der einzelnen Berechnungsschritte möchten wir uns hier sparen.

4. Ein Entfernen von Einträgen aus den CP -Informationen durch gezieltes Überschreiben von Variablen ist unmöglich, wenn π in SSA-Form ist. Dann sind die entsprechenden Tupel ab dem ersten Auftreten immer verfügbar und es können alle Vorkommen einer Kopiervariable auf der rechten Seite ersetzt werden. Dann kann sich aber durch T_{RC} kein neues Ersetzungspotential ergeben, da Kopieranweisungen entfernt anstatt hinzugefügt werden.
5. Aus 4. und der iterativen Anwendung von $T_{CP} \circ T_{CS}$ folgt, daß die Möglichkeit durch Kopieranweisungen die „wahren“ Argumente eines Funktionsausdrucks zu „verschleiern“, verhindert wird. Für das Programm aus Fall 2 würde beispielsweise schon die einmalige Anwendung von T_{CP} eine Optimierungsmöglichkeit für T_{CS} ergeben. Wenn nun π zusätzlich in SSA-Form vorliegt, können die entsprechenden T_{CP} -Substitutionen wie im Fall 3 nicht verhindert werden.
6. Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$ ein T_{CF} -optimales Programm und π' , α_k und α_l seien wie unter 1. gegeben, so daß $(x, y, l) \in RC_k$ und $\nexists (v, w, m) \in RC_k$ mit $m < l$.

Dann gilt $RD_k(x) = RD_l(y) = \perp$, denn sonst wäre $f(u_1, \dots, u_r)$ zu einer Konstante auswertbar gewesen. Da auch nach der Transformation in π' $RD'_k(y) = RD'_l(y) = \perp$ gilt, und alle Vorkommen von x durch y ersetzt werden, ändert sich nichts an der Konstanteninformation in den weiteren Anweisungen. Eine zusätzliche Anwendung von T_{CF} würde also nichts bewirken.

Damit wurden alle sechs Aussagen gezeigt. \square

Jetzt können wir überprüfen, ob T_{XOPT} die Programmtransformations-Eigenschaften erfüllt.

Satz 3.3.14: *Die erweiterte Gesamttransformation ist eine Programmtransformation, denn es gilt:*

1. T_{XOPT} ist korrekt; für eine Interpretation \mathfrak{A} gilt $T_{XOPT}(\pi) \sim_{\mathfrak{A}} \pi$ für alle $\pi \in \mathcal{LC}$ mit zu \mathfrak{A} passender Signatur.
2. T_{XOPT} ist idempotent, d.h. $T_{XOPT}(T_{XOPT}(\pi)) = T_{XOPT}(\pi)$ für alle $\pi \in \mathcal{LC}$.
3. T_{XOPT} arbeitet deterministisch; das Ergebnis ist allein durch die Eingabe bestimmt.

Beweis:

1. Die Aussage ist klar, denn alle an T_{XOPT} beteiligten Verfahren sind korrekt. Die entsprechenden Beweise finden sich für die klassischen Verfahren im Abschnitt 2.1 und für T_{RC} gilt die Korrektheit nach Satz 3.3.8; T_{SSA} wurde in Abschnitt 3.3.1.1 als „offensichtlich korrekt“ eingestuft. Wegen der Konstantenfaltung gilt nur die schwache Äquivalenz, obwohl alle anderen beteiligten Verfahren die starke Äquivalenz erhalten.
2. Sei $\pi \in \mathcal{LC}$ und $\pi' := T_{XOPT}(\pi)$. Da T_{SSA} offensichtlich idempotent ist, gilt $T_{SSA}(\pi') = \pi'$. Die Konstantenfaltung ist nach Lemma 3.2.9 von keinem der klassischen Verfahren verbesserbar und es gilt nach Lemma 3.3.13 genausowenig $T_{RC} \rightarrow T_{CF}$. Aufgrund der Idempotenz von T_{CF} folgt nun $T_{CF}(\pi') = \pi'$.

T_{CPCS} ist seiner Definition nach idempotent (Fixpunktiteration), aber es gilt $T_{SSA} \rightarrow T_{CPCS}$ wie wir anhand des Beispielprogramms aus dem Beweis von Satz 3.3.11 gesehen haben. Da jedoch T_{SSA} sowieso vorab ausgeführt wird, und die SSA-Eigenschaft der Eingabe dadurch garantiert ist (denn T_{CF} , T_{CS} und T_{CP} erhalten die SSA-Eigenschaft; siehe Abschnittsbeginn), ist keine weitere Änderung zu erwarten. Mit $T_{DC} \not\rightarrow T_{CP}$ und $T_{DC} \not\rightarrow T_{CS}$ sowie der Aussage 5 aus Lemma 3.3.13 folgt dann $T_{CPCS}(\pi') = \pi'$.

Da $T_{RC} \not\rightarrow T_{DC}$ gilt (Punkt 1 in Lemma 3.3.13), kann T_{DC} ebenfalls keine Änderungen mehr an π' vornehmen, es gilt also $T_{DC}(\pi') = \pi'$. Wegen der Idempotenz von T_{RC} (Korollar 3.3.9) erhalten wir $T_{RC}(\pi') = \pi'$. Für den abschließenden T_{SSA} -Schritt gilt ebenfalls $T_{SSA}(\pi') = \pi'$, da die anderen Verfahren keine Änderungen bewirkt haben, und die Variablenbenennung daher noch durch den letzten T_{SSA} -Schritt des ersten T_{XOPT} -„Durchlaufs“ vorgegeben ist.

3. Der Determinismus folgt aus dem Determinismus der Einzeltransformationen. \square

3.3.3 Äquivalenz von DAG-Verfahren und erweiterter Gesamttransformation

Nun ist zu prüfen, ob die Änderungen an T_{COPT} ausreichen, um die erweiterte Gesamttransformation T_{XOPT} äquivalent zum DAG-Verfahren zu machen. Vorab sind einige Hilfsaussagen zu zeigen.

Lemma 3.3.15: *Sei π ein Dead-Code-optimales LC-Programm und $D_\pi = (G, val, last)$ mit $G = (K, L, lab, suc)$ sein DAG. Dann sind alle $k \in K$ ausgaberelevant.*

Beweis: Es gelte $\pi =: (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta = \alpha_1; \dots; \alpha_n$ und $\alpha_i = x_i \leftarrow e_i$. Außerdem gilt $x_i \in LV_i$

wegen der Voraussetzung, daß π optimal bzgl. T_{DC} ist. Für $x \in V_\pi$ und $i \in \{1, \dots, n\}$ reicht es aus, zu zeigen:

$$x \in LV_i \Rightarrow \text{val}(x, i) \text{ ist ausgaberelevant}$$

Rückwärtige Induktion über $i \in \{1, \dots, n\}$:

- $i = n$:
 $x \in LV_n = V_{out} \Rightarrow \text{val}(x, n)$ ausgaberelevant gemäß Definition 2.2.4 (Fall 1).
- $i \rightarrow i - 1$:
 $x \in LV_{i-1} = LV_i \setminus \{x_i\} \cup V_{e_i}$ da $x_i \in LV_i$ gemäß Voraussetzung. Fallunterscheidung:
 1. $x \in V_{e_i}$ und $e_i = f(u_1, \dots, u_r)$. Dann existiert ein $j \in \{1, \dots, r\}$ und $\text{suc}(\text{val}(x_i, i), j) = \text{val}(x, i-1)$. Aufgrund der Induktionsannahme und Definition 2.2.4 (Fall 2) folgt: $\text{val}(x, i-1)$ ist ausgaberelevant.
 2. $x \in V_{e_i}$ und $e_i = x$. Dies ist nach Definition 1.1.3 für LC-Programme unzulässig.
 3. $x \in LV_i \setminus \{x_i\}$. Mit der Induktionsannahme folgt die Ausgaberelevanz von $\text{val}(x, i)$. Wegen $x_i \neq x$ ist dann gemäß DAG-Konstruktion (Alg. 2.2.2) auch $\text{val}(x, i-1)$ ausgaberelevant. \square

Als werden wir beweisen, daß die Konstanteninformationen der Konstantenfaltung mindestens so genau wie diejenigen von T_{DAG} sind. Dazu zeigen wir die Umkehrung der im Beweis von Satz 3.1.3 gezeigten Aussage.

Lemma 3.3.16: *Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ mit $\beta := \alpha_1; \dots; \alpha_n$. Dann gilt für alle $i \in \{1, \dots, n\}$ und alle $y \in V_\pi$:*

$$\text{val}(y, i-1) \in A \Rightarrow RD_i(y) \in A$$

Beweis: Induktion über $i \in \{1, \dots, n\}$:

- $i = 1$: $RD_1(y) = \sigma_\perp(y) = \perp$
- $i \rightarrow i + 1$: Fallunterscheidung nach dem Ausdruck e_i und der Variable y :
 1. Falls $e_i = f(u_1, \dots, u_r)$ und $y = x_i$ gilt:

$$\text{val}(y, i) = \varphi(f)(u'_1, \dots, u'_r) \in A$$

$$\text{wobei } u'_j := \begin{cases} \text{val}(u_j, i-1) & \text{falls } u_j \in V \\ u_j & \text{falls } u_j \in A \end{cases}$$

Dann folgt:

$$\begin{aligned} & \forall j \in \{1, \dots, r\} : u'_j \in A \\ & \stackrel{\text{IA}}{\Rightarrow} \forall j \in \{1, \dots, r\} : RD_i(u_j) \in A \text{ falls } u_j \in V, \text{ oder } u_j \in A \\ & \Rightarrow RD_{i+1}(y) = RD_i[x_i / \bar{\mathfrak{A}}[e_i] RD_i](y) = \bar{\mathfrak{A}}[e_i] RD_i \in A \end{aligned}$$

2. Falls $e_i = z \in V$ und $y = x_i$ gilt:

$$\begin{aligned} & \text{val}(y, i) = \text{val}(z, i-1) \in A \\ & \stackrel{\text{IA}}{\Rightarrow} RD_i(z) \in A \\ & \Rightarrow RD_{i+1}(y) = RD_i[x_i / \bar{\mathfrak{A}}[e_i] RD_i](y) = \bar{\mathfrak{A}}[e_i] RD_i = RD_i(z) \in A \end{aligned}$$

3. Falls $e_i = a \in A$ und $y = x_i$ gilt:

$$\begin{aligned} \text{val}(y, i) &= a \\ &= RD_i[x_i/\bar{\mathfrak{A}}[e_i]]RD_i(y) \\ &= RD_{i+1}(y) \end{aligned}$$

4. Sonst $y \neq x_i$:

$$\begin{aligned} \text{val}(y, i) &= \text{val}(y, i-1) \in A \\ &\stackrel{\text{IA}}{\Rightarrow} RD_i(y) \in A \\ &\Rightarrow RD_{i+1}(y) = RD_i[x_i/\bar{\mathfrak{A}}[e_i]]RD_i(y) = RD_i(y) \in A \end{aligned}$$

Also gilt die Behauptung. \square

Es gilt außerdem zu untersuchen, ob T_{CPCS} bei einem Eingabeprogramm in SSA-Form optimal in dem Sinne arbeitet, daß bei der DAG-Konstruktion eines $T_{CPCS} \circ T_{CF}$ -optimalen LC-Programms für jeden Operationsausdruck ein neuer Knoten anzulegen ist. Wäre dies nicht der Fall, dann würde $T_{XOPT} \sim T_{DAG}$ nicht gelten.

Die Konstantenfaltung muß dabei vorab ausgeführt werden, da auch bei der DAG-Konstruktion eine partielle Auswertung von Ausdrücken unter Kenntnis der mit Konstanten belegten Variablen durchgeführt wird (d.h. die Aussage wäre sonst nicht erfüllt).

Lemma 3.3.17: Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ ein LC-Programm in SSA-Form mit $\Sigma := (F, C)$, $\beta := \alpha_1; \dots; \alpha_n$, $\alpha_i := x_i \leftarrow e_i$, $T_{CPCS}(T_{CF}(\pi)) = \pi$ und sei $D_\pi = (G, \text{val}, \text{last})$ mit $G := (K, L, \text{lab}, \text{suc})$ der DAG von π sowie $\mathfrak{A} := (A, \varphi)$ eine Interpretation von Σ . Dann existiert für jedes $e_i \notin V \cup A$ ein $k_i \in K$.

Beweis: Man nehme an, für ein $l \in \{1, \dots, n\}$ mit $e_l = f(u_1, \dots, u_r)$, $f \in F^{(r)}$ existiere kein $k_l \in K$. Dann existiert gemäß der DAG-Konstruktion (Alg. 2.2.2) ein $l' < l$, so daß $k_{l'} \in K$ mit

$$\text{suc}(k_{l'}, j) = \begin{cases} \text{val}(u_j, l) & \text{falls } u_j \in V \\ u_j & \text{falls } u_j \in A \end{cases} \quad (*)$$

Nach Lemma 3.3.16 kann nicht $\text{val}(u_j, l) \in A$ für alle $j \in \{1, \dots, r\}$ mit $u_j \in V$ gelten, da π optimal bzgl. der Konstantenfaltung ist. Die Anweisung $\alpha_{l'}$ habe die Form $x_{l'} \leftarrow f(u'_1, \dots, u'_r)$.

Fallunterscheidung (induktiv):

1. Für alle $j \in \{1, \dots, r\}$ gilt $u'_j = u_j \in A \cup V$.
Dann folgt: $l' \in AE_l$ und $l \in wh(l')$. Damit würde aber nicht $T_{CS}(\pi) = \pi$ gelten und folglich auch nicht $T_{CPCS}(T_{CF}(\pi)) = \pi$. Widerspruch.
2. Es existiert ein $j \in \{1, \dots, r\}$, so daß $u_j \neq u'_j$, und $u_j, u'_j \in V$ wegen Lemma 3.3.16. Dann gibt es verschiedene Möglichkeiten:
 - a) Es existiert ein $k < l$ mit $\alpha_k = u_j \leftarrow z$, $z \in V$ und $\text{val}(z, k-1) = \text{val}(u_j, l)$ oder für ein $k' < l'$ mit $\alpha_{k'} = u'_j \leftarrow z'$ gilt $\text{val}(z', k'-1) = \text{val}(u'_j, l')$. Beides ist unmöglich, da dann die Entfernung von Kopien wegen der SSA-Form des Programms u_j bzw. u'_j jeweils durch z bzw. z' ersetzt hätte.
 - b) Es gibt ein $k < l$ und ein $k' < l'$, mit $\alpha_k = u_j \leftarrow g(s_1, \dots, s_p)$ sowie $\alpha_{k'} = u'_j \leftarrow g(s'_1, \dots, s'_p)$ und es gilt wieder (*) mit $u_j = s_j$, $u'_j = s'_j$, $f = g$, $l = k$, $l' = k'$ und $r = p$.

Da $n < \infty$ muß sich irgendwann eine Terminierung der Induktion ergeben, die aber nur in Fall 1 oder 2(a) eintreten kann. Folglich ergibt sich ein Widerspruch und die Behauptung gilt. \square

In einem T_{XOPT} -optimalen Programm können ferner keine beliebigen Typen von Kopieranweisungen auftreten, wie folgendes Lemma zeigt:

Lemma 3.3.18: *Sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) \in \mathcal{LC}$ ein T_{XOPT} -optimales Programm mit $\beta := \alpha_1; \dots; \alpha_n$ mit $\alpha_i := x_i \leftarrow e_i$. Dann gilt für alle $i \in \{1, \dots, n\}$:*

$$\alpha_i = x \leftarrow y \quad \Rightarrow \quad x \in V_{out} \wedge y \in V_{in} \cup V_{out}$$

Beweis:

1. x muß eine Ausgabevariable sein, da sonst aufgrund der SSA-Eigenschaft von π , die aufgrund des Einsatzes von T_{SSA} sichergestellt ist, sämtliche Vorkommen von x in α_j , $j > i$ durch y substituiert worden wären und α_i damit zu Dead Code würde. Widerspruch zur T_{XOPT} -Optimalität.
2. Sei $y \notin V_{in} \cup V_{out}$. Dann muß es eine Operationsanweisung $\alpha_{i'} = y \leftarrow f(u_1, \dots, u_r)$ für $i' < i$ geben. Eine Konstantenzuweisung zur Wertbelegung kommt wegen T_{CF} nicht in Frage und eine Kopieranweisung ist nach Fall 1 unmöglich.

Wegen der SSA-Eigenschaft von π wird weder x noch y außer in α_i und $\alpha_{i'}$ jemals ein Wert zugewiesen. Außerdem kommt y oberhalb von α_i nicht im Programm vor. Daher würde dann T_{RC} eine Ersetzung von y durch x vornehmen und die Anweisung α_i entfernen. Dies ist ein Widerspruch zur Optimalität bzgl. T_{XOPT} . \square

Nun sind alle Voraussetzungen erfüllt und wir können mit dem Beweis der eigentlich interessanten Aussage beginnen.

Satz 3.3.19 (Optimalität von T_{XOPT} bzgl. T_{DAG}): *Für alle $\pi \in \mathcal{LC}$ gilt:*

$$T_{XOPT}(\pi) = T_{DAG}(T_{XOPT}(\pi))$$

Beweis: Für $\pi_0 \in \mathcal{LC}$ und $\Sigma := (F, C)$ sei $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta) := T_{XOPT}(\pi_0)$ mit $\beta := \alpha_1; \dots; \alpha_n$. Mit $\mathfrak{A} := (A, \varphi)$ sei eine Interpretation von Σ gegeben. Außerdem sei $D_\pi := (G, val, last)$ mit $G := (K, L, lab, suc)$ der DAG von π und $T_{DAG}(\pi) =: \pi' =: (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta')$ mit $\beta' := \alpha'_1; \dots; \alpha'_n$.

π liegt wegen der abschließenden Anwendung von T_{SSA} in SSA-Form vor. Daher gilt für alle $i \in \{1, \dots, n\}$ und alle $v \in V_\pi$: Falls $val(v, i) \in K$ (Funktionswert ist definiert) ist $val(v, i) = val(v, n)$. Wegen Lemma 3.3.15 kann es außerdem in K nur ausgaberelevante Knoten geben, Algorithmus 2.2.5 ist also direkt anwendbar.

Wir zeigen, daß $\alpha_i = \alpha'_i$ für $i \in \{1, \dots, n\}$ gelten muß. Sei $\alpha_i = x \leftarrow e$, Fallunterscheidung:

1. $e = a \in A$: Dieser Fall ist nur für $x \in V_{out}$ möglich, da sonst alle weiteren Vorkommen⁵ von x durch a ersetzt worden wären, und α_i somit Dead Code wäre.

Nach der DAG-Konstruktion (Alg. 2.2.2) gilt dann $a \in K$, $val(x, i) = val(x, n) = a$ und $last(x) = i$, da später keine Neudefinition von x wegen der SSA-Eigenschaft möglich ist. Folglich wird gemäß Algorithmus 2.2.5 bei der Auswahl des nächsten Knotens k_l , $l > i$ vor der Codegenerierung für k_l zunächst gemäß Schritt 3 eine Zuweisung an y durchgeführt, d.h. $\alpha'_i = x \leftarrow a = \alpha_i$. Oder aber es existiert kein Operationsknoten wie oben, dann ergibt sich das gleiche Resultat mit Schritt 8.

⁵Wegen der SSA-Eigenschaft von π wird x nie neudefiniert.

2. $e = y \in V$: Nach Lemma 3.3.18 gilt $x \in V_{out}$ und es sind zwei Fälle zu unterscheiden:
- $y \in V_{in}$: Es gilt $val(x, i) = y$ und $last(x) = i$. Dann ist analog zum ersten Fall gemäß Schritt 3 oder 8 des Algorithmus zur DAG-Codegenerierung $\alpha'_i = x \leftarrow y$.
 - $y \in V_{out}$: Dann gibt es eine Anweisung $\alpha_l = y \leftarrow f(u_1, \dots, u_r)$ mit $l < i$ in π (da Konstantenzuweisungen an y und weitere Kopieranweisungen offensichtlich ausscheiden) und nach Lemma 3.3.17 wurde ein Knoten k_l , der α_l repräsentiert, angelegt. Folglich gilt $val(x, i) = val(y, i - 1) = k_l$, $last(y) = l$ und $last(x) = i$. In π' existiert also schon eine Operationsanweisung für k_l mit y als Zuweisungsvariable (diese ist α_l). Folglich wird gemäß Schritt 3 oder 8 analog zu Fall 1 $\alpha'_i := x \leftarrow y$ gesetzt.
3. $e = f(u_1, \dots, u_r)$ für ein $f \in F^{(r)}$: Nach Lemma 3.3.17 existiert dann ein Knoten $k_i \in K$ mit

$$\begin{aligned} lab(k_i) &= f \\ suc(k_i, j) &= u'_j \text{ mit } u'_j = \begin{cases} val(u_j, i - 1) & \text{falls } u_j \in V \\ u_j & \text{sonst} \end{cases} \\ val(x, i) &= k_i \end{aligned}$$

Weitere Fallunterscheidung nach dem Typ von y :

- (a1) $x \in V_{out}$: Es ist $last(x) = i$ wegen der SSA-Eigenschaft von π . Folglich ist x der Eintrag mit dem kleinsten $last$ -Wert aller $V_{out}^{k_i}$ in Schritt 4 (denn die restlichen $z \in V_{out}^{k_i}$ können ihren Wert nicht vor x erhalten). Also wird $\delta := \delta[k_i/x]$ gesetzt.

Der in Schritt 6 generierte Code sieht wie folgt aus:

$$\begin{aligned} \alpha_i &= \delta(k_i) \leftarrow f(\delta(suc(k_i, 1)), \dots, \delta(suc(k_i, r))) \\ &= x \leftarrow f(\delta(suc(k_i, 1)), \dots, \delta(suc(k_i, r))) \end{aligned}$$

- (b1) $x \notin V_{out}$: Also ist $x = v_i$ (wegen der abschließenden Anwendung von T_{SSA}). Es muß nun $V_{out}^{k_i} = \emptyset$ gelten, denn für eine Ausgabevariable $z \neq x$ mit $val(z, n) = k_i$ gilt:
- Eine Wertzuweisung an z mit einer Kopierkette ist nicht möglich, da wegen $v_i \notin V_{in} \cup V_{out}$ Lemma 3.3.18 verletzt wäre.
 - Eine Zuweisung eines zu α_i äquivalenten Operationsausdrucks an z ist nicht möglich, da π optimal bzgl. T_{CPCS} ist.

Wegen $\delta(k_i) \notin V_{out}$ wird dann in Schritt 5 $\delta := \delta[k_i/v_i]$ gesetzt. Also ergibt sich:

$$\begin{aligned} \alpha_i &= \delta(k_i) \leftarrow f(\delta(suc(k_i, 1)), \dots, \delta(suc(k_i, r))) \\ &= v_i \leftarrow f(\delta(suc(k_i, 1)), \dots, \delta(suc(k_i, r))) \end{aligned}$$

Nun ist noch die Gültigkeit von $\delta(suc(k_i, j)) = u_j$ für $j \in \{1, \dots, r\}$ zu zeigen. Fallunterscheidung:

- (a2) $u_j \in A$: Die Behauptung gilt sofort gemäß DAG-Konstruktion.
- (b2) $u_j \in V$: Es existiert ein $l < i$, so daß $\alpha_l = u_j \leftarrow e_l$ in π . Dabei kann, da π in SSA-Form ist, u_j nur in der Anweisung α_l einen Wert zugewiesen bekommen. Also gilt $val(u_j, i) = val(u_j, l)$.

Falls e_l ein Operationsausdruck ist und nach Lemma 3.3.17 ein Knoten $k_l \in K$. Unterscheidung nach dem Typ von u_j :

- (i) $u_j \notin V_{out}$: Es gilt

$$\delta(suc(k_i, j)) = \delta(val(u_j, i)) = \delta(val(u_j, l)) = \delta(k_l) = v_l = u_j,$$

da ähnlich wie im Fall 3(b1) $V_{out}^{k_l} = \emptyset$ sein muß.

- (ii) $u_j \in V_{out}$: $\delta(suc(k_i, j)) = \delta(k_l) = u_j$ (da der *last*-Eintrag von u_j automatisch der niedrigste aller Variablen aus $V_{out}^{k_l}$ ist).

Wenn $e_l \in V_{out}$ muß nach Lemma 3.3.18 $e_l \in V_{in} \cup V_{out}$ gelten. Erneute Fallunterscheidung:

- (i) $e_l \in V_{in}$: Dann gilt, da π optimal bzgl. T_{CP} ist, und vor Anwendung von T_{CP} die SSA-Eigenschaft galt, $u_j = x$. Da ebenfalls $val(u_j, i-1) = val(u_j, l) = x$, $suc(k_i, j) = val(u_j, i-1) = x$ und $\forall v \in V_{in} : \delta(v) = v$ ist $\delta(suc(k_i, j)) = u_j = x$.
- (ii) $e_l \in V_{out}$: Dann gibt es ein $l' < l$ mit $\alpha_{l'} = e_l \leftarrow g(s_1, \dots, s_p)$, da wegen T_{CP} eine weitere Kopieranweisung $e_l \leftarrow z$ mit $z \in V_{in} \cup V_{out}$ nicht in Frage kommt (denn dann wäre e_l durch z ersetzt worden).

Also gilt $val(u_j, i-1) = val(u_j, l) = val(e_l, l-1) = val(e_l, l')$ und $suc(k_i, j) = k_{l'}$. Dann ist $u_j = e_l$, $\delta(k_{l'}) = e_l$ (gemäß DAG-Codegenerierung, Schritt 4) und daher $\delta(suc(k_i, j)) = u_j = e_{l'}$.

Insgesamt ist T_{XOPT} also optimal bezüglich T_{DAG} . □

Für die Äquivalenz der erweiterten Gesamttransformation und der DAG-Optimierung muß noch die Rückrichtung betrachtet werden:

Hauptsatz 3.3.20 (Simulation des DAG-Verfahrens): *Die beiden Programmtransformationen T_{XOPT} und T_{DAG} sind äquivalent (es gilt $T_{XOPT} \sim T_{DAG}$).*

Beweis: Nach Definition 3.3.1 ist noch zu zeigen:

$$\forall \pi \in \mathcal{LC} : T_{XOPT}(T_{DAG}(\pi)) = T_{DAG}(\pi) \quad (*)$$

Die andere Richtung gilt nach Satz 3.3.19.

In Abschnitt 3.1 wurde gezeigt, daß T_{DAG} optimal bezüglich der klassischen Optimierungsverfahren ist. Die Optimalität bezüglich T_{RC} gilt gemäß Satz 3.3.10. T_{DAG} ist ebenfalls T_{SSA} -optimal, da die Variablenbenennung exakt der der DAG-Codegenerierung entspricht.

Die Anwendung eines klassischen Verfahrens, von T_{RC} oder T_{SSA} auf ein T_{DAG} -optimales Programm bewirkt folglich keine Änderungen. Eine Verkettung dieser Verfahren kann dann auch keine Änderung bewirken. Also gilt (*). □

3.4 Fazit

Mit der erweiterten Gesamttransformation haben wir ein aus Einzeltransformationen zusammengesetztes Verfahren entwickelt, dessen Optimierungsqualität derjenigen der DAG-Optimierung entspricht. Dabei ist der Implementierungsaufwand beim Einsatz aller Einzeltransformationen u.U. größer als der für die DAG-Optimierung, die sich jedoch nicht für den Zwischencode iterativer Programme eignet.

Da die klassischen Verfahren aber ohnehin in aktuellen Compilern eingesetzt werden [11], und auch iterative Versionen der SSA-Transformation existieren [13, 14], ist nur eine zusätzliche Implemen-

tierung der rückwärtigen Kopierpropagation notwendig (sofern die dadurch erreichten minimalen Änderungen den Aufwand rechtfertigen). So betrachtet, wäre der Einsatz von T_{XOPT} sinnvoller.

Durch die wiederholte Anwendung von T_{CPCS} müssen allerdings in ungünstigen Fällen (die aber praktisch kaum vorkommen dürften) evtl. Abstriche beim Laufzeitverhalten des Optimierungsalgorithmus gemacht werden.

4 Implementierung der Verfahren

Bisher haben wir uns nur mit den theoretischen Aspekten der vorgestellten Verfahren zur Optimierung von LC-Programmen beschäftigt. In diesem Kapitel wird eine experimentelle Implementierung in der Programmiersprache C vorgestellt, die alle an der erweiterten Gesamttransformation beteiligten Verfahren und die DAG-Optimierung realisiert. Als Signatur wird $\Sigma = (F, C)$ mit $F = \{+^{(2)}, -^{(2)}, *^{(2)}, /^{(2)}, -^{(1)}\}$ und $C = \mathbb{Z}$ oder $C = \mathbb{R}$ zugrundegelegt. Die Darstellung von Operationsausdrücken erfolgt in der Infixnotation.

Als Interpretation dient die übliche Bedeutung der Operationssymbole auf \mathbb{Z} bzw. auf \mathbb{R} . Die Wertebereiche werden im Rechner natürlich endlich durch die der Integer- bzw. Fließkommazahlen repräsentiert. Da die Implementierung hauptsächlich zu Testzwecken erstellt wurde, wurde die Effizienz etwas vernachlässigt.

Der Quellcode kann unter dem Namen `LC-Optimierer.tar.gpg` von `da.srieger.com` heruntergeladen werden. Die Datei liegt verschlüsselt vor, auf der referenzierten Seite befindet sich eine kurze Anleitung zur Entschlüsselung. Dazu benötigen sie ein Passwort:

4dv7hpxq

Durch das Überprüfen von einem der folgenden Hashwerte (der verschlüsselten Datei) stellen Sie sicher, daß die Datei seit der Abgabe dieser Arbeit nicht verändert wurde:

SHA1: 0C24 AC00 E90D A563 7F45 FA7B 3611 ACF3 1ECC 1A68
MD5: D5 87 34 91 73 B3 84 31 79 69 D2 92 9A A1 28 19

Im Archiv befindet sich zudem eine Datei mit dem Namen `Anleitung.txt`, die die Bedeutung der einzelnen Dateien beschreibt (u.a. Zuordnung der Optimierungsverfahren zu Dateien, Hinweise zum Kompilieren und zu Aufrufparametern).

```
vin: x y;  
vout: u v;  
u <- 3;  
v <- x - y;  
w <- u + 1;  
x <- x - y;  
u <- x - y;  
z <- u * w;  
u <- 2 * u;  
v <- w - 1;
```

Abbildung 4.1: Programm aus Abb. 1.1 als Eingabe für den Optimierer

```

:
SIGN [-+]
ALPHA [A-Za-z]
DIGIT [0-9]
:
{SIGN}?({DIGIT}+"."DIGIT*|DIGIT*"."{DIGIT}+)([eE]{SIGN}?{DIGIT}+)?
    { yyfval=atof(yytext); return T_FLOAT; }

{ALPHA}({ALPHA}|{DIGIT})*
    { return T_VAR; }
:

```

Abbildung 4.2: Reguläre Ausdrücke zur Erkennung von Fließkommazahlen und Variablenbezeichnungen

4.1 Verarbeitung der Eingabe

Bevor die eigentlichen Optimierungen angewendet werden können, muß das Eingabeprogramm (als Textdatei, siehe Beispiel in Abbildung 4.1) eingelesen werden, die dann lexikalisch und syntaktisch analysiert und in eine programminterne Darstellung umgewandelt werden. Dazu bedienen wir uns der bewährten Methoden aus dem Compilerbau [1].

Es ist eine lexikalische und eine syntaktische Analyse der Eingabe durchzuführen. Die semantische Analyse ist in unserem Fall mit der Syntexanalyse „verflochten“ und dient dem Aufbau einer programminternen Darstellung des Eingabeprogramms.

4.1.1 Lexikalische Analyse mit Flex

Die lexikalische Analyse wird von dem Unix-Programm FLEX („fast lexical analyser generator“) übernommen. FLEX erhält eine Spezifikationsdatei als Eingabe, in der reguläre Ausdrücke einzelne *Symbolklassen* definieren. Eine Symbolklasse repräsentiert eine Sprache über dem ASCII-Alphabet.

Aus der Spezifikation, die auch benutzerdefinierten C-Code enthalten kann, generiert FLEX einen *Scanner*¹ (oder auch *Lexer*) in Form einer C-Quellcodedatei, der das Eingabe-LC-Programm in eine Folge von Teilstrings, sogenannte *Lexeme* zerlegt, die den einzelnen Symbolklassen zugeordnet sind (also Elemente der entsprechenden Sprachen sind).

Lexeme in LC-Programmen sind z.B. der Zuweisungsoperator <-, die Zahl 13 oder die Variable x. Die entsprechenden Symbolklassen lauten in diesem Fall ASSIGN, INT oder VAR. Als Sprachen betrachtet, enthält ASSIGN nur ein Wort, nämlich den Zuweisungsoperator, während INT alle Integer-Zahlen und VAR alle Variablenamen enthält.

Beispielhaft wollen wir hier die regulären Ausdrücke für die Erkennung von Fließkommazahlen und Variablenbezeichnern in der FLEX-Spezifikation² näher betrachten. Der entsprechende Ausschnitt ist in Abbildung 4.2 dargestellt.

Zunächst werden die Ausdrücke SIGN für das Vorzeichen (+ oder -), ALPHA für einen Groß- oder Kleinbuchstaben und DIGIT für Ziffern definiert. Unter Verwendung dieser wird nun der Aufbau

¹Programm zur lexikalischen Analyse; führt „Pattern-Matching“ auf der Eingabe aus.

²Die gesamte Spezifikation findet sich in der Datei lc.1.

einer Fließkommazahl definiert:

- Optionales Vorzeichen³: $\{\text{SIGN}\}?$
- Zahlenfolge mit Dezimalpunkt, wobei entweder vor oder hinter dem Punkt Zahlen stehen müssen:
 $\{\text{DIGIT}\}^+ \cdot \{\text{DIGIT}\}^* | \{\text{DIGIT}\}^* \cdot \{\text{DIGIT}\}^+$
- Optionaler Exponent, der mit **e** oder **E** eingeleitet wird, ein optionales Vorzeichen enthalten kann und mit einer nichtleeren Zahlenfolge endet: $[\text{eE}]\{\text{SIGN}\}?\{\text{DIGIT}\}^+$

Der Punkt ist innerhalb der regulären Ausdrücke in Hochkommata zu setzen, da er sonst jedes Zeichen matcht (spezielle Bedeutung bei FLEX). Ein Variablenbezeichner beginnt mit einem Buchstaben; der Rest besteht aus Buchstaben oder Ziffern.

In den geschweiften Klammern sind C-Befehle angegeben, die ausgeführt werden, wenn der Ausdruck durch die Eingabe gematcht wird. Im Fall der Fließkommazahl wird der Variablen `yy1val` der Binärwert der Zahl zugewiesen und das *Token* `T_FLOAT` zurückgeliefert (der vorher gespeicherte Binärwert der Zahl heißt *Attribut* des Tokens). Ein Token ist der Identifikator der entsprechenden Symbolklasse.

Es gilt das *First-Longest-Match*-Prinzip: Wähle den am weitesten oben stehenden regulären Ausdruck, der die längste noch von einem Ausdruck erkannte Zeichenfolge matcht.

Ein Scanner kann mithilfe eines sogenannten *Backtrack*-Automaten konstruiert werden [1]. Dieser ist eine Erweiterung des Vereinigungsautomaten der regulären Ausdrücke. Eine detaillierte Beschreibung von FLEX findet man in seiner Dokumentation [9].

4.1.2 Syntaktische Analyse

Nach der lexikalischen Analyse muß nun die syntaktische Struktur des Eingabeprogramms erkannt werden, dies übernimmt der *Parser*. Eine kontextfreie Grammatik dient dabei der Spezifikation der syntaktischen Struktur. Das zugrundeliegende Terminalalphabet ist die Menge der Symbolklassen, deren Token der Scanner liefert.

Das Problem bei kontextfreien Grammatiken ist, daß die von deterministischen Kellerautomaten erkennbaren kontextfreien Sprachen eine *echte* Teilmenge der nichtdeterministisch erkennbaren kontextfreien Sprachen darstellen [18].

Bei der Syntaxanalyse gibt es im wesentlichen zwei unterschiedliche Ansätze, um dieses Problem zu umgehen:

- Beim *Top-Down*-Ansatz versucht man aus dem Startsymbol der Grammatik durch Regelanwendung das gewünschte Terminalwort abzuleiten. Zur deterministischen Entscheidung bei Regelalternativen wird ein *Lookahead* auf der Eingabe eingesetzt.

Grammatiken, die sich für diese Form der Analyse eignen, sind die sogenannten *LL(1)*-Grammatiken, eine beschränkte Teilklasse kontextfreier Grammatiken, bei denen ein *Lookahead* von einem Symbol zur deterministischen Analyse ausreicht. Dazu muß für jede Regel eine *Lookahead-Menge* berechnet werden [1, 3], die angibt, welche Zeichen bei Anwendung der Regel als nächstes auf der Eingabe stehen dürfen.

- Als Alternative bietet sich die *Bottom-Up*-Analyse an, bei der ausgehend vom zu analysierenden Terminalwort Regeln „rückwärtig“ angewendet werden, bis man das Startsymbol erhält.

³Vorzeichen könnte man auch als Operatoren erkennen, was aber bei der Grammatik für die Syntaxanalyse (Abschnitt 4.1.2) zu Schwierigkeiten führen würde (*LL(1)*-Eigenschaft).

Dieser Ansatz ist mit Lookahead mächtiger als die Analyse mittels LL(1)-Grammatiken und wird z.B. vom Unix-Parsergenerator BISON eingesetzt [10].

Da es sich in unserem Fall um relativ einfache syntaktische Strukturen handelt, kommen wir mit LL(1)-Grammatiken aus. Eine Grammatik ist genau dann eine LL(1)-Grammatik, wenn bei Regelalternativen der Schnitt der Lookahead-Mengen der einzelnen Regeln leer ist. Wie wir gleich sehen werden, erhält man oft nach der ersten „intuitiven“ Konstruktion noch keine LL(1)-Grammatik und muß Modifikationen vornehmen.

Erster Ansatz einer Grammatik G für LC-Programme (Terminalsymbole in Großbuchstaben):

$$\begin{array}{ll}
 \text{program} & \rightarrow \text{VIN } vlist \text{ SEM VOUT } vlist \text{ SEM } cmdlist & (1) \\
 vlist & \rightarrow \text{VAR } vlist \mid \varepsilon & (2, 3) \\
 cmdlist & \rightarrow cmd \text{ cmdlist } \mid \varepsilon & (4, 5) \\
 cmd & \rightarrow \text{VAR ASSIGN } exp \text{ SEM} & (6) \\
 exp & \rightarrow vc \text{ OP } vc \mid \text{OP VAR } \mid vc & (7, 8, 9) \\
 vc & \rightarrow \text{VAR} \mid \text{INT} \mid \text{FLOAT} & (10, 11, 12)
 \end{array}$$

Die Bezeichnungen der einzelnen Symbole sowie deren Verwendung dürften relativ leicht auf die jeweilige Bedeutung schließen lassen. Die Berechnung der Lookahead-Mengen für die Regeln mit Alternativen ergibt:

Regel-Nr.	Lookahead
2	{VAR}
3	{SEM}
4	{VAR}
5	{ ε }
7	{VAR,INT,FLOAT}
8	{OP}
9	{VAR,INT,FLOAT}
10	{VAR}
11	{INT}
12	{FLOAT}

Man erkennt sofort, daß die Regelalternativen 7 und 9 die gleiche Lookahead-Menge besitzen. Daher ist in diesem Fall keine deterministische Entscheidung möglich. Z.B. ist bei der Ableitung von exp und einem INT-Symbol auf der Eingabe nicht klar, ob exp nach $vc \text{ OP } vc$ oder nach vc abzuleiten ist. Man könnte diese Problematik durch Ausprobieren und Backtracking in den Griff bekommen, was aber nicht sehr effizient ist. Einen solchen Ansatz verfolgt z.B. PROLOG mit den sogenannten *Definite Clause Grammars* (mehr dazu in [16]).

G ist folglich keine LL(1)-Grammatik. Durch eine (äquivalenzerhaltende) Modifikation der Grammatik kann man jedoch versuchen, dieses Problem zu beseitigen. Durch Einführung eines neuen Nichtterminals $exp2$ kann in den Regeln 7 und 9 die Entscheidung zur Regelwahl „hinter“ vc verlagert werden. Dies ist in folgender Grammatik G' geschehen (Änderungen zu G sind durch Fettdruck hervorgehoben):

$$\begin{array}{ll}
 \text{program} & \rightarrow \text{VIN } vlist \text{ SEM VOUT } vlist \text{ SEM } cmdlist & (1) \\
 vlist & \rightarrow \text{VAR } vlist \mid \varepsilon & (2, 3) \\
 cmdlist & \rightarrow cmd \text{ cmdlist } \mid \varepsilon & (4, 5) \\
 cmd & \rightarrow \text{VAR ASSIGN } exp \text{ SEM} & (6) \\
 \textbf{exp} & \rightarrow \textbf{vc } \textbf{exp2} \mid \textbf{OP VAR} & \textbf{(7, 8)} \\
 \textbf{exp2} & \rightarrow \textbf{OP } \textbf{vc} \mid \varepsilon & \textbf{(9, 10)} \\
 vc & \rightarrow \text{VAR} \mid \text{INT} \mid \text{FLOAT} & (10, 11, 12)
 \end{array}$$

```

void _vlist (...) {
    switch(sym){
        case T_VAR:
            ...
            match(T_VAR);
            _vlist (...);
            break;
        case T_SEM:
            break;
        default:
            err_invalid_symbol("_vlist");
    }
}

```

Abbildung 4.3: C-Funktion für das Nichtterminalsymbol *vlist*

Obiges Vorgehen nennt man *Linksfaktorisierung* (Details dazu in [1, 3]). G' ist eine LL(1)-Grammatik, denn die Lookahead-Mengen der modifizierten Regeln sind konfliktfrei:

Regel-Nr.	Lookahead
7	{VAR,INT,FLOAT}
8	{OP}
9	{OP}
10	{SEM}

Der implementierte Parser arbeitet nach dem Prinzip des *rekursiven Abstiegs*: Für jedes Nichtterminal wird eine Funktion (bzw. Prozedur) angelegt, die dann beim Auftreten eines Nichtterminals auf einer rechten Regelseite die diesem zugeordnete Funktion aufruft. Für Terminalsymbole wird die `match`-Funktion aufgerufen, die prüft, ob das nächste Eingabesymbol mit dem erwarteten Terminalsymbol übereinstimmt. Falls ja, wird vom Scanner das nächste Symbol angefordert (ansonsten Ausgabe eines Fehlers). Für jedes Nichtterminalsymbol mit mehr als einer Regel entscheiden die Lookheadsymbole über die Auswahl der Alternative.

In Abbildung 4.3 ist der Code der dem Nichtterminalsymbol *vlist* zugeordneten C-Funktion dargestellt⁴. Das `switch`-Konstrukt dient der Unterscheidung der möglichen Lookheadsymbole. Falls der Lookahead VAR ist (also `sym` den Wert `T_VAR` hat), wird die Regel zwei angewendet. Nach dem „Matchen“ von `T_VAR` erfolgt der rekursive Aufruf von `_vlist`, da *vlist* auch auf der rechten Regelseite auftritt.

Wenn der Lookahead das Semikolon ist, wird das Nichtterminalsymbol *vlist* stattdessen gemäß der dritten Regel zu ε abgeleitet (nichts weiter zu tun). Bei anderem Lookahead wird ein Fehler ausgegeben.

4.1.3 Interne Darstellung

Zusätzlich zur Erkennung der syntaktischen Struktur wird beim Parsen des Eingabeprogramms eine interne Darstellung erzeugt. Die dieser zugrundeliegende Datenstruktur ist in Abbildung 4.4 graphisch und in Abbildung 4.5 als C-Code dargestellt.

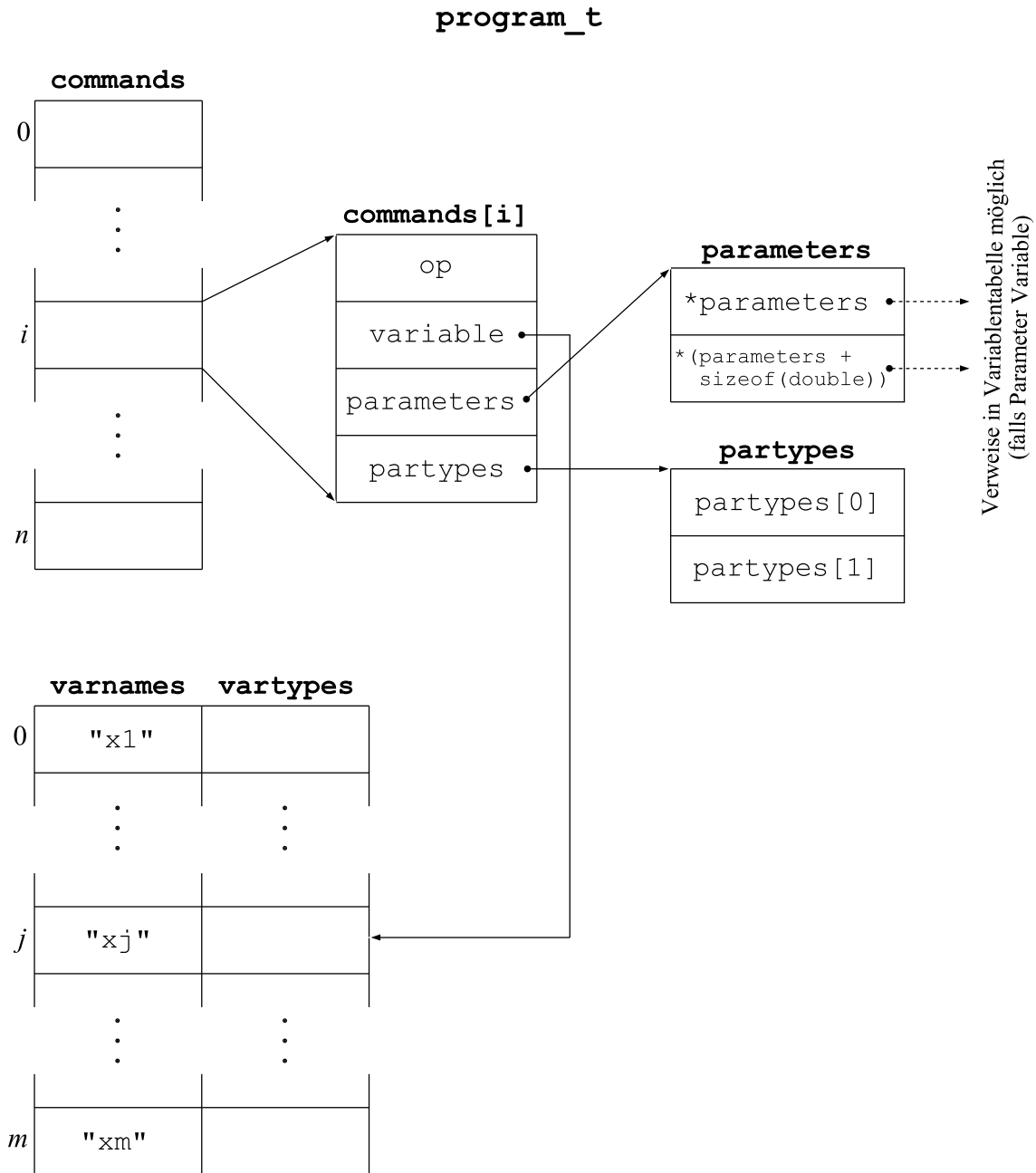


Abbildung 4.4: Interne Darstellung von LC-Programmen: Graphische Darstellung

```

typedef struct{
    op_t op;
    int variable;
    void* parameters;
    par_t* partypes;
} command_t;

typedef struct{
    char** varnames;
    var_t* vartypes;
    command_t* commands;
    int nvariables;
    int ncommands;
} program_t;

```

Abbildung 4.5: Interne Darstellung von LC-Programmen: C-Code

`program_t` gliedert sich in zwei Hauptkomponenten:

- Tabelle von Variablennamen und Typen (Arrays `varnames` und `vartypes`):
 - `varnames` enthält die Variablenbezeichner des Ausgangsprogramms als Zeichenfolgen.
 - `vartypes` gibt zu jedem Variablenbezeichner einen Typ an, also etwa ob es sich um eine Ein-/Ausgabe- oder Programmvariable handelt und ob die Variable vorher definiert wurde (Gewährleistung der Vollständigkeit des Programms).
- Dynamisches Feld `commands`, das die einzelnen Anweisungen enthält. Jede Anweisung (Typ `command_t`) enthält wiederum vier Einträge:
 - `op` gibt die zugrundeliegende Operation an, d.h. ob $+(2)$, $-(2)$, $*(2)$, $/(2)$, $-(1)$ verwendet wird, oder eine Kopieranweisung oder Konstantenzuweisung vorliegt.
 - `variable` gibt den Index der Variablen der linken Seite in der Variablentabelle an.
 - `parameters` ist ein `void`-Zeiger auf einen Speicherbereich der Größe `2*sizeof(double)`. Zusammen mit den Parametertypen in `partypes` können die Werte der Parameter ausgelesen werden. Es kann sich dabei um drei verschiedene Typen handeln:
 - * Variablenindex (`int`), falls Parameter eine Variable ist,
 - * Integerkonstante (`int`), falls Parameter ein Wert aus \mathbb{Z} ist oder
 - * Fließpunktkonstante (`double`), falls Parameter ein Wert aus \mathbb{R} ist.
 - Im zweielementigen Feld `partypes` werden die Typen der Parameter (wie oben angegeben) abgespeichert. Diese müssen vor einer Dekodierung von `parameters` ausgelesen werden, u.a. auch wegen des unterschiedlichen Speicherbedarfs für `int`- und `double`-Variablen.

Die gewählte Kodierung mag kompliziert erscheinen, ermöglicht jedoch eine weitgehende Typunabhängigkeit ohne für jeden Typ eigene Variablen bereitzustellen. Die Typen müßten dann ohnehin abgespeichert werden.

- Die Variablen `nvariables` und `ncommands` geben die Größe der jeweiligen Felder an und wurden in der Grafik nicht explizit dargestellt, entsprechen aber $m + 1$ und $n + 1$. Die Typen `op_t`, `var_t` und `par_t` sind einfache Aufzählungstypen (`enum`).

Die Datenstrukturen werden beim Parsen gefüllt, z.B. bietet es sich an, in der Funktion `_vlist` die Variablentabelle zu füllen. Die dazu notwendigen Befehle wurden in Abbildung 4.3 aus Übersichtlichkeitsgründen weggelassen.

Die den Nichtterminalen zugeordneten Funktionen besitzen zusätzlich Parameter, denn es müssen Werte im Ableitungsbaum transferiert werden. So muß etwa in `_vc` der Wert und Typ eines Parameters ermittelt werden, der dann weiter „nach oben“ gereicht wird, um in `_cmd` für die aktuelle Anweisung eingetragen zu werden. Dieser Prozeß entspricht in gewisser Weise der *semantischen Analyse*, die sich auch mit *Attributgrammatiken* (siehe [1, 3]) realisieren läßt.

4.2 Implementierung der Optimierungsverfahren

Die in den Kapiteln 2 und 3 vorgestellten Optimierungsverfahren wurden weitgehend entsprechend ihrer Definitionen umgesetzt. Vor der Beschreibung der jeweiligen Implementierung werden die verwendeten Datenstrukturen etwas erläutert.

⁴Der Code wurde auf das wesentliche gekürzt. Auslassungen sind durch „...“ angedeutet.

4.2.1 Datenstrukturen für die einfachen Verfahren

Bei den einfachen Transformationen stellen die Analyseinformationen Mengen dar. Zur Mengendarstellung werden *Bitvektoren* verwendet, die zur Darstellung einer beliebigen Teilmenge T einer Obermenge M verwendet werden können. Dazu muß allerdings eine (implizite) vollständige Ordnung der Elemente in M bestehen. In der Praxis ist diese jedoch eigentlich immer gegeben, weil die Elemente von M gewöhnlich in einem geordneten Array abgespeichert werden.

Beispiel 4.2.1: Sei $M := \{u, v, w, x, y, z\}$ eine Menge (von Variablen). Dann repräsentiert der Bitvektor $b := (0, 1, 0, 1, 1, 0) \in \mathbb{B}^6$ die Teilmenge $M_b := \{v, x, y\}$; es sind also genau diejenigen Elemente in M_b enthalten, deren Eintrag in b auf 1 gesetzt wurde (die implizit vorausgesetzte Ordnung ist die lexikographische).

In der vorliegenden Implementierung wurden einige Hilfsfunktionen zur Verarbeitung von Bitvektoren definiert, die u.a. die Speicherallokation, Änderung von einzelnen Bits, Abfrage von Werten von Bits und das Kopieren von Bitvektoren ermöglichen.

Die Wahl der Obermenge M erfolgt je nach Verfahren in Abhängigkeit von $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta := \alpha_1; \dots; \alpha_n$:

- Dead Code Elimination: $M := V_\pi$ (Ordnung gemäß Deklarationsreihenfolge)
- Common Subexpression Elimination: $M := \{1, \dots, n\}$ (Ordnung gemäß $<_{\mathbb{N}}$)
- Kopierpropagation: $M := V_\pi \times V_\pi$, wobei die zugehörige Ordnung $<$ wie folgt definiert ist: $(a, b) < (c, d)$ gilt genau dann, wenn entweder $a < c$ oder $a = c$ und $b < d$. Der dritte Eintrag der Tupel, die transitive Tiefe, wird aus Effizienzgründen in einem gesonderten Feld erfasst.
- Rückwärtige Kopierpropagation: wie Kopierpropagation, auch hier gesonderte Behandlung des Zeilenindex (3. Tupeleintrag).

Für die Konstantenfaltung wird eine weitere Datenstruktur verwendet, die eine Variablenbelegungsfunktion modelliert. Diese besteht aus einem Speicherbereich (auf den ein `void`-Zeiger zeigt), der die Werte der Variablen enthält, wobei für jede Variable Speicher der Größe `sizeof(double)` reserviert wird. Ähnlich wie schon bei den Parametertypen werden dann die Typen in einem gesonderten Feld festgehalten. Hier sind zusätzlich zu Integer- und Fließpunktzahlen als Werttypen auch undefinierte Variablen erlaubt.

4.2.2 Implementierung der einfachen Programmtransformationen

Mithilfe der im vorangegangenen und im Abschnitt 4.1.3 beschriebenen Datenstrukturen war eine an die jeweilige Definition angelehnte Implementierung bei Dead Code Elimination und Common Subexpression Elimination relativ einfach zu realisieren. Bei der Konstantenfaltung und der Kopierpropagation war hingegen schon etwas mehr „Handarbeit“ erforderlich. Im folgenden gelte $\pi := (\Sigma, \vec{v}_{in}, \vec{v}_{out}, \beta)$ mit $\beta := \alpha_1; \dots; \alpha_n$ und $\alpha_i := x_i \leftarrow e_i$. In Klammern werden jeweils die entsprechenden Zeilennummern in der zugehörigen Quellcodedatei angegeben.

4.2.2.1 Umsetzung der Dead Code Elimination

Für jede Anweisung α_i wird ein Bitvektor, der die LV_i -Menge repräsentiert, angelegt. Dann werden rückwärtig, beginnend mit dem Bitvektor, in dem nur die Einträge der Ausgabevariablen auf 1 gesetzt sind, die Analyseinformationen berechnet (21-29). Dabei werden die Werte des vorher

berechneten Bitvektors in den zu berechnenden übertragen (23) und dann die entsprechenden Veränderungen gemäß Definition 2.1.1 vorgenommen (24-28)

In das Ausgabeprogramm werden nur diejenigen Anweisungen eingefügt, für die im Bitvektor für die linksseitige Variable eine 1 enthalten ist (34-35).

4.2.2.2 Umsetzung der Common Subexpression Elimination

Verwendung von zwei Bitvektoren pro Anweisung α_i : $\mathbf{AE}[i]$ repräsentiert die Analyseinformation AE_i und $\mathbf{wh}[i]$ das Ergebnis von $wh(i)$. Die Berechnung der $\mathbf{AE}[i]$ erfolgt dreischrittig:

1. Kopieren des Werts von $\mathbf{AE}[i-1]$ in $\mathbf{AE}[i]$ (78).
2. Berechnen von $gen_{\alpha_{i-1}}(AE_{i-1})$ (81-89): Falls es sich bei e_{i-1} um einen Operationsausdruck handelt und ein Ausdruck $e_j = e_i$, $j \in \{1, \dots, i-2\}$ existiert, so daß das j -te Bit in $\mathbf{AE}[i-1]$ gesetzt ist, wird das $(i-1)$ -te Bit in $\mathbf{AE}[i]$ auf 1 gesetzt, ansonsten findet keine Änderung statt.
3. Anwendung von $kill_{\alpha_{i-1}}$ auf das durch Schritt 2 modifizierte $\mathbf{AE}[i]$ (92-97): Prüfe, ob für ein $j \in \{1, \dots, i-1\}$ mit $i \in AE_k$ der Ausdruck e_j die Variable x enthält (Auslesen der Parameter mit Typprüfung). Ist dies der Fall wird das j -te Bit in $\mathbf{AE}[i]$ auf 0 gesetzt.

Als nächstes wird für alle $i \in \{1, \dots, n\}$ $\mathbf{wh}[i]$ berechnet (100-103). Dazu wird für jede Anweisung α_i geprüft, ob ein $j \in \{i+1, \dots, n\}$ existiert, so daß das i -te Bit in $\mathbf{AE}[j]$ gesetzt ist und $e_i = e_j$ gilt (Prüfung der Gleichheit mittels Hilfsfunktion `expressions_equal`). Falls ja, wird in $\mathbf{wh}[i]$ das j -te Bit auf 1 gesetzt.

Für die Programmtransformation ist für jede Anweisung α_i zu unterscheiden (Test mittels Hilfsfunktion `bv_allzero`):

1. $wh(i) \neq \emptyset$: Erzeugung einer neuen Variable mit dem Namen `ti` und Anlegen der entsprechenden Befehle gemäß Definition 2.1.10 (109-113).
2. $wh(i) = \emptyset$: Prüfe, ob ein $j \in \{1, \dots, i-1\}$ existiert, so daß das i -te Bit in $\mathbf{wh}[j]$ gesetzt ist. Dann wird die entsprechende Kopieranweisung $x_i \leftarrow t_j$ eingefügt. Trifft dies nicht zu, wird α_i unverändert übernommen (116-124).

4.2.2.3 Umsetzung der Konstantenfaltung

Die Konstantenfaltung nutzt die Datenstruktur für Variablenbelegungen und verwendet die Hilfsfunktion `cmd_evaluate`, der als Argument eine Variablenbelegung übergeben wird. `cmd_evaluate` (139-219) führt eine partielle Auswertung einer Anweisung durch, wobei eine Unterscheidung nach Typen, Definiertheit und Operationen notwendig ist.

Für jede Anweisung α_i wird eine Variablenbelegung $\mathbf{RD}[i]$ (entspricht der RD_i -Information) mithilfe von `cmd_evaluate` berechnet (237-240). Dann werden bei der Programmtransformation folgende Fälle unterschieden:

1. Die Variablenbelegung $\mathbf{RD}[i+1]$ für α_{i+1} enthält an der Position von x_i einen definierten Wert a (e_i ließ sich zu a auswerten): In das Ergebnisprogramm wird eine Konstantenzuweisung $x_i \leftarrow a$ eingefügt (246-248).
2. Sonst: Überprüfung der Parameter auf ihre Typen; für Variablen $x \in V_{e_i}$ wird bei definiertem Eintrag a in der Variablenbelegung $\mathbf{RD}[i]$ dieser übernommen und x durch a ersetzt. Ansonsten bleiben die alten Parameterwerte und -typen erhalten (251-271).

4.2.2.4 Umsetzung der Propagation von Kopien

Wie bereits erwähnt werden die *CP*-Analysemengen durch Bitvektoren CP dargestellt, die Teilmengen von $M := V_\pi \times V_\pi$ modellieren und der dritte Eintrag wird mittels eines zusätzlichen Arrays D gleicher Länge dargestellt. Es gibt also für jedes Paar (x, y) , $x, y \in V_\pi$ einen Eintrag in den Bitvektoren, wobei reflexive Einträge (x, x) , $x \in V_\pi$ nie auf 1 gesetzt werden.

Benutzung von drei Hilfsfunktionen:

- **CP_reachable** (279-288) prüft, ob eine Variable y von einer Startvariablen x aus in einer gegebenen (nicht transitiv abgeschlossenen) *CP*-Menge⁵ transitiv erreichbar ist. Dazu wird zunächst ein Eintrag der Form (x, v, d) für ein $v \in V_\pi$ gesucht und dann rekursiv mit der Startvariablen v weitergesucht. Es handelt sich also gewissermaßen um eine Tiefensuche. Als Ergebnis wird die transitive Tiefe zurückgeliefert.

In Abbildung 4.6 ist dazu ein Beispiel für einen (fiktiven) Aufrufbaum angegeben, wobei in den Knoten jeweils Start- und Zielvariable vermerkt wurden. Die vorangehende Nummer gibt die Generierungsreihenfolge an. An denen mit BT markierten Knoten ist ein Backtracking erforderlich. Der dritte Eintrag in den Tupeln wurde aus Übersichtlichkeitsgründen weggelassen.

- **CP_trans** (292-305) berechnet für eine gegebenen *CP*-Information die transitive Hülle. Dazu wird für alle nichtgesetzten Bits, die ein Paar (x, y) repräsentieren, die Erreichbarkeit von y von x aus mithilfe von **CP_reachable** geprüft. Ist y von x erreichbar, wird das entsprechende Bit auf 1 gesetzt, die transitive Tiefe aktualisiert und wieder beim ersten Bit angefangen, da ja jetzt u.U. durch zusätzliche Einträge wiederum neue Variablen erreichbar sind. Es handelt sich also um eine Fixpunktberechnung.
- **CP_subst** (309-327) substituiert gemäß der übergebenen *CP*-Information die Variablen der rechten Seite eines Ausdrucks. Dabei wird gemäß Definition 2.1.23 für eine Variable jeweils der Eintrag mit der maximalen Tiefe gesucht, um die Idempotenz zu garantieren.

Bei der Berechnung der *CP*-Informationen wird ähnlich wie bei den anderen Verfahren vorgegangen. Für jede Anweisung α_i werden zunächst alle Bits in $\text{CP}[i+1]$ auf 0 gesetzt, die ein Tupel repräsentieren, das x_i enthält (357-358). Dann wird im Falle einer Kopieranweisung $x_i \leftarrow y$ das zu (x_i, y) das zugehörige Bit auf 1 gesetzt und die transitive Tiefe mit 1 initialisiert (362-364). Abschließend wird die transitive Hülle mithilfe von **CP_trans** berechnet (366).

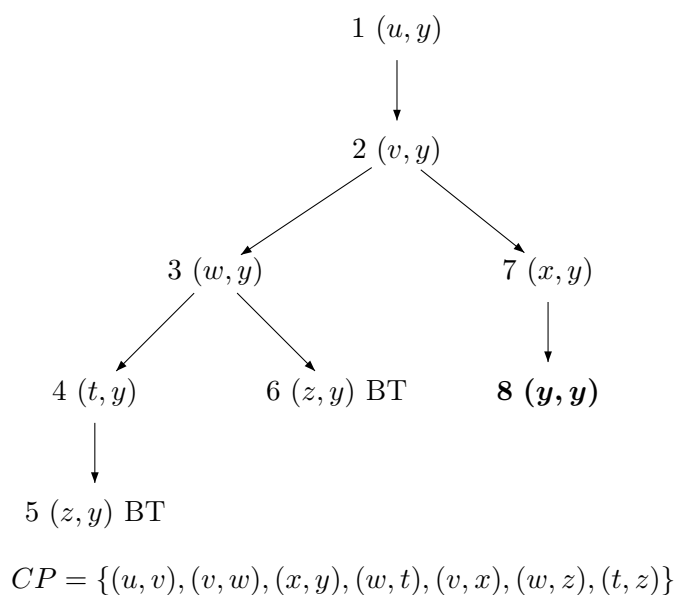
Die Programmtransformation (370-376) ist relativ einfachstrukturiert und baut größtenteils auf der Funktion **CP_subst** auf, die die notwendigen Substitutionen vornimmt.

4.2.2.5 Umsetzung der rückwärtigen Kopierpropagation

Zum Einsatz kommen auch hier Bitvektoren, wobei das Verfahren auf der gleichen Mengendarstellung wie die Kopierpropagation aufbaut. Da jedoch zusätzlich zu den Paaren von Variablen auch einzelne Variablen eingefügt werden, wird eine Bitvektor-Länge von $|V_\pi|^2 + |V_\pi|$ Bits verwendet. Die ersten $|V_\pi|^2$ Bits dienen zur Darstellung einer Menge von Paaren (wie bei T_{CP}), während die restlichen $|V_\pi|$ festlegen, ob eine bestimmte (einzelne) Variable in der Menge enthalten ist. Ein zusätzliches Feld J dient zur Verwaltung der Zeilenindizes (3. Eintrag in den Tupeln).

Nach der Speicherallozierung wird **delta**, welches die Substitution δ des Algorithmus 3.3.6 repräsentiert mit der Identität initialisiert (401). Für jede Anweisung $i = n - 1, \dots, 1$ (Rückwärtige Analyse)

⁵Die Implementierung arbeitet mit Bitvektor- anstatt Mengendarstellung. Hier wurde die Mengendarstellung wegen der höheren Anschaulichkeit gewählt.

Abbildung 4.6: Aufrufbaum von `CP_reachable(CP, u, y, ...)`

werden nun folgende Schritte durchgeführt:

1. Übertragen der Werte des Bitvektors `RC[i+1]` in `RC[i]` (für `J[i+1]` analog) (405-406).
2. Berechnen von $kill_{\alpha_{i+1}}(RC_{i+1})$: Setzen aller Bits in `RC[i]` auf Null, die Paare der Form (x_{i+1}, v) , $v \in V$ oder (v, x_{i+1}) , $v \in V$ repräsentieren (412-413). Außerdem werden für alle Variablen z der rechten Seite die Bits der Einträge (v, z) , $v \in V$ in `RC[i]` auf Null gesetzt (414-418).
3. Anwendung von $gen_{\alpha_{i+1}}$ auf das durch Schritt 2 veränderte `RC[i]`: Es werden die notwendigen Bedingungen zum Einfügen eines Tupels gemäß Definition 3.3.5 geprüft und bei positivem Ergebnis das Tupel $(e_{i+1}, x_{i+1}, i + 1)$ mit $e_{i+1} \in V$ in RC_i durch Setzen des entsprechenden Bits in `RC[i]` und Aktualisierung des entsprechenden `J[i]`-Eintrags eingefügt (421-429).
4. Das der Variable x_{i+1} zugeordnete Bit in `RC[i]` wird ebenfalls gesetzt (430).

Die Programmtransformation (434-461) läuft analog zum Algorithmus 3.3.6 ab. Beim Antreffen einer Operationsanweisung, wird das für die Variable der linken Seite passende Tupel mit minimalem Zeilenindex in der Analysemenge ermittelt (440-445) und dann für eine spätere Modifikation von `delta` der zu aktualisierende Index zwischengespeichert (446).

Schließlich wird die noch unveränderte Substitution auf die rechte Seite der aktuellen Anweisung angewandt, dann erfolgt bei Bedarf das Update von `delta` und die Anweisung (nach Anwendung des neuen `delta` auf die linke Seite) ins Ausgabeprogramm eingefügt (448-460).

4.2.2.6 Umsetzung der SSA-Transformation

T_{SSA} läßt sich relativ einfach implementieren. Anstatt wie in Definition 3.3.2 für jede Anweisung eine eigene Substitution zu verwenden, wird ein Feld `rho` verwendet, das mit jeder Iteration so aktualisiert wird, daß man eine Repräsentation der nächsthöheren ρ -Funktion erhält.

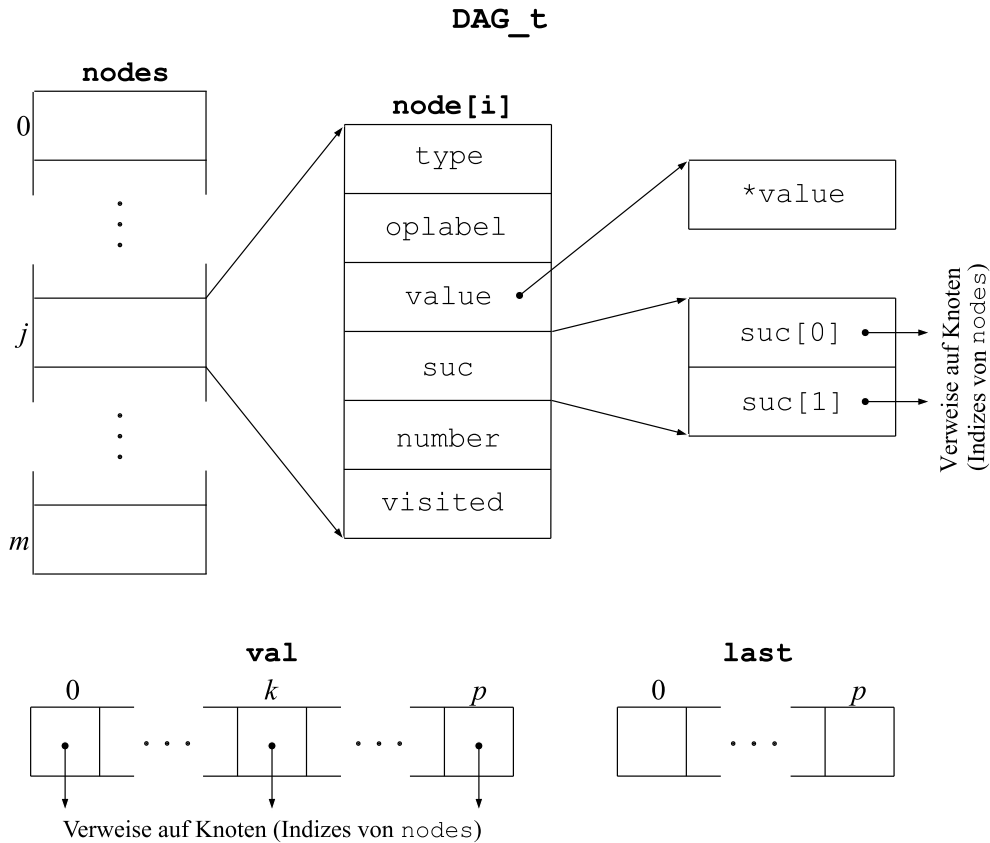


Abbildung 4.7: Datenstruktur zur Repräsentation eines DAG (Grafik)

ρ bildet die Variablenindizes des Ausgangsprogramms auf die Variablenindizes des Zielprogramms ab, so daß eine relativ einfache Anwendung möglich ist.

Nach dem Einfügen der Ein- und Ausgabevariablen in das Ergebnisprogramm (482-486) werden zunächst in einem **last**-Array die Indizes der letzten Zuweisungen an Ausgabevariablen festgehalten (487-488). Durch einen Vergleich des **last**-Eintrags einer Ausgabevariable, die in einer Anweisung eine Zuweisung erhält, mit dem Index der Anweisung kann schließlich entschieden werden, ob die Ausgabevariable erhalten bleibt (501) oder ob eine neue Variable anzulegen ist (504-505).

Die Substitution der Variablen auf der rechten Seite einer Anweisung (492-498) erfolgt dabei vor der Aktualisierung von ρ (507), so daß sichergestellt ist, daß die rechte Seite noch mit der alten Substitution „bearbeitet“ wird.

4.2.3 Datenstrukturen für die DAG-Optimierung

Für die Darstellung eines DAG sind etwas kompliziertere Datenstrukturen notwendig, als die in den vorangegangenen Abschnitten vorgestellten. Die Knoten müssen miteinander verkettet werden (*suc*-Funktion) und die Zuordnung von Variablen zu Knoten muß gespeichert werden (*val*-Funktion).

In den Abbildungen 4.7 und 4.8 ist die zur DAG-Darstellung verwendete Datenstruktur **DAG_t** als Grafik und als C-Code dargestellt. Hier nun eine kurze Beschreibung der einzelnen Komponenten:

- **nodes** ist ein (dynamisches) Feld, dessen Einträge die Knoten des DAG repräsentieren. Durch

```

typedef struct{
    DAGnodetype_t type;
    op_t oplabel;
    int variable;
    void* value;
    int suc [2];
    int number;
    int visited;
} DAGnode_t;

typedef struct{
    DAGnode_t* nodes;
    int nnodes;
    int* val;
    int* last;
} DAG_t;

```

Abbildung 4.8: Datenstruktur zur Repräsentation eines DAG (C-Code)

das Array ergibt sich ein Vorteil gegenüber einer Datenstruktur, die auf der Verzeigerung der einzelnen Knoten basiert:

Es ist ein einfacher Durchlauf durch alle Knoten des DAG möglich, um z.B. bestimmte Knoten zu finden (Auswahl des nächsten Knoten bei der Codegenerierung aus dem DAG).

Die einzelnen Komponenten eines Eintrags in `nodes` sind folgende:

- `type` gibt den Typ eines DAG-Knotens an; es werden Variablen-, Operations- und Konstantenknoten (Integer-/Fließpunktzahl) unterschieden.
- Durch `oplabel` wird die Operationsbeschriftung eines Knotens analog zur *lab*-Funktion festgelegt.
- `value` zeigt auf einen Speicherbereich der Größe `sizeof(double)`, der bei Konstanten- bzw. Variablenknoten den Konstantenwert bzw. einen Index der Variablen-tabelle enthält. Die Dekodierung ist nur zusammen mit `type` möglich (da `void`-Zeiger).
- `suc` enthält die Indizes der Nachfolger des aktuellen Knotens.
- `number` speichert die Knotennummer, die später bei der Codegenerierung gemäß Algorithmus 2.2.5 die Benennung von Nicht-Ausgabewerten bestimmt.
- Die Statusvariable `visited` dient bei der Codegenerierung aus dem DAG dazu, zu überprüfen, ob ein Knoten schon abgearbeitet wurde (Repräsentation von K_f).
- `val` repräsentiert die *val*-Funktion. Da es dabei immer nur auf die letzten, aktuellsten Werte ankommt, muß nicht für jede Anwendung ein Array bereitgestellt werden. Die Einträge in `val` sind Knotenindizes.
- `last` ist die Representation der *last*-Funktion, die Ausgabewerte die Anweisungsnummer ihrer letzten Zuweisung zuordnet⁶.
- `nnodes` dient nur zum Speichern der Länge des `nodes`-Feldes (Speicherverwaltung) und entspricht der Anzahl der Knoten im DAG.

4.2.4 Umsetzung der DAG-Optimierung

Bei der Implementierung der DAG-Optimierung finden zahlreiche Hilfsfunktionen Verwendung, u.a. sind dies:

⁶Aus Gründen der Einfachheit ist in `last` für jede Variable ein Eintrag enthalten (Entfallen der Indexumrechnung). Dies wäre natürlich effizienter lösbar.

- `DAG_insert_node` (23-39) fügt einen neuen Knoten in den DAG ein. Dabei wird Speicher für den neuen Knoten alloziert und die Parameter entsprechend gesetzt. Zurückgeliefert wird der Index des neuen Knotens in `nodes`.
- `DAG_find_or_insert_node` (43-56) prüft vor dem Einfügen eines Knotens unter Verwendung von `DAG_insert_node`, ob schon ein gleichartiger gemäß Algorithmus 2.2.2, Fall 3(b1) vorhanden ist (in diesem Fall wird nur der Index des gefundenen Knotens zurückgeliefert).
- `DAG_node_output_relevant` (60-68) überprüft die Ausgaberelevanz eines Knotens. Dazu werden in einer Schleife alle Knoten durchlaufen und für jeden Knoten k geprüft, ob eine Ausgabevariable mit Index i existiert, so daß `val[i]` auf k verweist. Ist dies nicht der Fall müssen, falls k Vorgänger besitzt, diese auf Ausgaberelevanz getestet werden (rekursiver Aufruf).
- `DAG_choose_node` (72-83) ist die Umsetzung von Schritt 2 von Algorithmus 2.2.5. Es wird der erste ausgaberelevante Knoten (in Erzeugungsreihenfolge) gewählt (und sein Index zurückgeliefert), dessen `visited`-Attribut nicht gesetzt ist, während die seiner Nachfolger gesetzt sind.
- `DAG_choose_output_var` (87-97) entspricht dem 4. Schritt von Algorithmus 2.2.5 und liefert gemäß `val`-Tabelle den Index einer einem Knoten zugeordneten Ausgabevariable mit dem kleinsten `last`-Wert zurück (sofern vorhanden).
- `DAG_insert_copy_instructions` (101-132) ist die Implementierung von Schritt 3 und 8. Beim Aufruf kann eine obere Schranke für den zu berücksichtigen `last`-Wert der Ausgabevariablen angegeben werden. Dann wird einer Schleife Code für die entsprechenden Ausgabevariablen (in der Reihenfolge der `last`-Werte) generiert und die Anzahl der eingefügten Anweisungen zurückgeliefert.

4.2.4.1 DAG-Konstruktion

Nach der Speicherallozierung und Initialisierung werden zuerst die Eingabevariablen als Knoten mittels `DAG_insert_node` eingefügt. Die Einträge in der `val`-Tabelle werden dementsprechend gesetzt (167-171).

Es werden nacheinander alle Anweisungen des Eingabeprogramms abgearbeitet. Für eine Anweisung $\alpha_i = x \leftarrow e$ wird der `last`-Eintrag von x auf i gesetzt, falls $i \in V_{out}$ (176). Dann wird zwischen folgenden Fällen unterschieden:

1. $e = y \in V$: Es wird nur die `val`-Tabelle aktualisiert (179).
2. $e = c \in C$: Ggf. Einfügen eines neuen Konstantenknotens mit `DAG_find_or_insert_node` und Aktualisierung der `val`-Tabelle (181-182). In Algorithmus 2.2.2 wurden die Konstanten schon zu Anfang in den DAG eingefügt, die hier leicht abgeänderte Behandlung gefährdet jedoch offensichtlich nicht die Korrektheit.
3. $e = f(u_1, \dots, u_r)$: Zuerst wird nach Eintragung der Parameter, die unter Benutzung der `val`-Tabelle zu ermitteln sind, in eine Variablenbelegungstabelle (185-192) mit `cmd_evaluate` eine partielle Auswertung von e durchgeführt (193). Es sind dann zwei weitere Fälle zu unterscheiden:
 - a) In der Ergebnisvariablenbelegung ist ein definierter Wert für x eingetragen. Dann wird bei Bedarf eine Konstante eingefügt und die `val`-Tabelle aktualisiert (195-196).
 - b) x ist in der Ergebnisvariablenbelegung undefiniert. Es wird, falls noch kein „äquivalenter“ Knoten vorhanden ist, ein neuer Knoten mit `DAG_find_or_insert_node` eingefügt,

dessen Nachfolger sich bei Variablen über die `val`-Tabelle und bei Konstanten direkt ergeben (201-205). Eine Konstante muß u.U. eingefügt werden, falls sie noch nicht im DAG ist (203-205).

4.2.4.2 DAG-Codegenerierung

Die Erzeugung des Ergebnisprogramms läuft weitgehend analog zu Algorithmus 2.2.5 in folgenden Schritten ab:

1. Einfügen der Ein- und Ausgabevariablen in das Ergebnisprogramm und Setzen der Variablen-substitution `delta` (entspricht dem δ im Algorithmus) auf die Identität (213-215).
2. Wählen des zu bearbeitenden Knotens `node` mit `DAG_choose_node` und inkrementieren des Anweisungszählers `instr_counter`, der dem i aus Algorithmus 2.2.5 entspricht (216-127).
3. Ggf. werden Zuweisungen an Ausgabevariablen mit `DAG_insert_copy_instructions` eingefügt, wobei die Knotennummer als obere Schranke für den `last`-Wert verwendet wird. `instr_counter` wird um die Zahl der eingefügten Anweisungen erhöht (218).
4. Prüfen mit `DAG_choose_output_var`, ob `node` eine Ausgabevariable zugeordnet ist (119). Bei mehreren wird diejenige mit dem niedrigsten `last`-Wert von der Funktion zurückgeliefert (219).
 - Falls ja, wird die Substitution `delta` aktualisiert und in einem Bitvektor die entsprechende Ausgabevariable als benutzt markiert (221-222).
 - Falls nein, wird eine neue Variable vi erzeugt und ins Ergebnisprogramm eingefügt, wobei i der Wert in `instr_counter` ist. Außerdem wird `delta` so geändert, daß für den Knotenindex von `node` die Nummer der neuen Variablen geliefert wird (225-226).
5. Generierung von Code für `node`: Die Parameter werden einzeln dekodiert, mit `delta` die in ihnen vorkommenden Variablen substituiert und die Parameter in temporäre Variablen kopiert (228-239). Dann wird eine neue Anweisung unter Anwendung von `delta` auf die Variable der linken Seite eingefügt (240).
6. Für alle noch nicht berücksichtigten Ausgabevariablen (Ermittlung mittels des oben erwähnten Bitvektors) werden Wertzuweisungen wie in Algorithmus 2.2.5, Schritt 8 nach `last`-Werten sortiert eingefügt (243). Der Aufruf von `DAG_insert_copy_instructions` erfolgt diesmal ohne Beschränkung.

Zusammenfassung

Diese Arbeit behandelt die automatische Analyse und nachfolgende Optimierung von linearem Code, einer auf eine lineare Abfolge von Wertzuweisungen beschränkte Klasse von Zwischencode in Compilern. Linearer Code findet sich in iterativen Programmen in Schleifenrumpfen, daher ist seine Effizienz besonders wichtig.

Bei den vorgestellten Optimierungsverfahren gibt es zwei unterschiedliche Ansätze:

Die *klassischen Einzeltransformationen* wie Dead Code Elimination, Common Subexpression Elimination, Konstantenfaltung und Propagation von Kopien behandeln jeweils einen zu optimierenden Teilaspekt, um das Programm schließlich dahingehend zu transformieren.

Das graphbasierte *DAG-Verfahren* erzeugt eine komplette Programmrepräsentation in Form eines Graphen und erzeugt dann aus diesem Code, ohne auf das Ausgangsprogramm zurückgreifen zu müssen. Das Resultat ist dabei, wie formal gezeigt wird, optimal im Hinblick auf die klassischen Transformationen.

Das zentrale Thema dieser Arbeit ist es, zu klären, inwieweit man durch Kombination der klassischen Verfahren eine Transformation konstruieren kann, die ähnlich guten Code wie das DAG-Verfahren liefert. Nach der Untersuchung der Abhängigkeiten der klassischen Verfahren untereinander, wird schließlich eine Gesamttransformation entworfen, die die Einzeltransformationen günstig kombiniert.

Es stellt sich jedoch später heraus, daß diese Gesamttransformation immer noch nicht so mächtig wie das DAG-Verfahren ist. Daher sind einige Verbesserungen in Form weiterer optimierender Transformationen notwendig, um schließlich ein aus zahlreichen Einzelschritten zusammengesetztes Optimierungsverfahren zu erhalten, das die gewünschte Eigenschaft besitzt: Es ist optimal gegenüber der DAG-Transformation, was durch einen formalen Beweis bestätigt wird.

Zusätzlich zu den theoretischen Überlegungen wurde ebenfalls eine experimentelle Implementierung der Optimierungsverfahren erstellt, mit der deren Funktionalität getestet werden kann. Abschließend wird genauer auf die dort eingesetzten Methoden, Strategien und Datenstrukturen eingegangen.

Literaturverzeichnis

- [1] K. Indermark: *Compilerbau*, Vorlesung im SS 2004 an der RWTH Aachen
- [2] K. Indermark: *Programmanalyse und Compileroptimierung*, Vorlesung im SS 2003 an der RWTH Aachen
- [3] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986
- [4] O. H. Ibarra, B. S. Leininger: *The Complexity of the Equivalence Problem for Straight-Line Programs*, 12th Annual ACM Symposium on Theory of Computing, Los Angeles, 1980
- [5] M. Davis, Y. Matijasevic, J. Robinson: *Hilbert's Tenth Problem. Diophantine equations: positive aspects of a negative solution*, Proceedings of Symposia in Pure Mathematics, Vol. 28, S. 323-378, American Mathematical Society, 1976
- [6] J. Giesl: *Termersetzungssysteme*, Vorlesung im SS 2004 an der RWTH Aachen
- [7] A. V. Aho, S. C. Johnson, J. D. Ullman: *Code Generation for Expressions with Common Subexpressions*, J. of the Association for Computing Machinery, Vol 24, No 1, S. 146-160, 1977
- [8] A. V. Aho, S. C. Johnson: *Optimal Code Generation for Expression Trees*, Journal of the Association for Computing Machinery, Vol. 23, No. 3, S. 488-501, 1976
- [9] Free Software Foundation, *Flex, a fast lexical analyser generator*, GNU Project, www.gnu.org/software/flex, 1998
- [10] Free Software Foundation, *Bison, The Yacc-compatible Parser Generator*, GNU Project, www.gnu.org/software/bison, 2004
- [11] Red Hat, Inc., *GCC Optimization* (Überblick über die im GCC-Compiler implementierten Optimierungsverfahren), www.redhat.com/software/gnupro/technical/gnupro_gcc.html
- [12] O. H. Ibarra, S. Moran: *Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs*, Journal of the Association for Computing Machinery, Vol 30, No 1, S. 217-228, 1983
- [13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems, Vol 13, No 3, S. 451-490, 1991
- [14] J. Aycock, R. N. Horspool: *Simple Generation of Static Single-Assignment Form*, Proceedings of the 9th International Conference on Compiler Construction, S. 110-124, 2000
- [15] Steven S. Muchnick: *Advanced Compiler Design and Implementation*, Morgan Kaufman, San Francisco, 1997
- [16] E. Shapiro, L. Sterling: *The Art of Prolog*, MIT Press, Cambridge, 2001
- [17] R. Morgan: *Building an Optimizing Compiler*, Digital Press, 1998
- [18] J. E. Hopcroft, R. Motwani, J. D. Ullman: *Introduction to automata theory, languages, and computation*, Addison-Wesley, S. 247-249, 2001