Analyse und Transformation kontrollflußparalleler Programme

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik

der Universität Fridericiana zu Karlsruhe

genehmigte

Dissertation

von

Jürgen Vollmer

aus Achern/Baden

Tag der mündlichen Prüfung:	7. Mai 1996
Erster Gutachter:	Prof. Dr. Gerhard Goos
Zweiter Gutachter:	Prof. Dr. Bernhard Steffen

Inhaltsverzeichnis

Inhaltsverzeichnis i

Tabelle der Algorithmen v

1 Einleitung 1
- 1.1 Ziele, Motivation, Lösungsansatz, Resultate dieser Arbeit 1
- 1.2 Gliederung der Arbeit 4
- 1.3 Danksagungen 4

2 Ein einführendes Beispiel 5
- 2.1 Das Beispiel 5
- 2.2 Probleme eines einfaches Optimierers mit parallelen Programmen 6
- 2.3 Probleme mit der Befehlsumordnung in parallelen Programmen 6
- 2.4 Probleme mit dem Vertauschen von Speicherzugriffen bei parallelen Programmen 7
- 2.5 Eine mögliche Lösung dieser Probleme 8
- 2.6 Schlußfolgerungen 8

3 Grundlagen, Problemstellung und Literatur 9
- 3.1 Die Umgebung: das Rechnermodell und die parallele Programmiersprache *pPL* 9
  - 3.1.1 Das Rechnermodell 9
  - 3.1.2 Parallele Programmiersprachkonzepte 10
  - 3.1.3 Syntax und Semantik von der Beispielsprache *pPL* 11
  - 3.1.4 Die Spracherweiterungen pPL_{sync} und pPL_{lock} 13
  - 3.1.5 Semantikerhaltende Transformationen auf *pPL* 14
  - 3.1.6 Schlußfolgerungen 15
- 3.2 Optimierung paralleler Programme 16
  - 3.2.1 Grenzen und Möglichkeiten der Optimierung 16
  - 3.2.2 Korrektheit der Optimierung 16
  - 3.2.3 Abstrakte Interpretation 17
  - 3.2.4 Interleavingsemantik 18
  - 3.2.5 Beobachtbare Zustände 21
  - 3.2.6 Beispiele für Transformationen 22
  - 3.2.7 Schlußfolgerung 22
- 3.3 Das Problem 22
- 3.4 Grundlagen der Analyse sequentieller Programme 24
  - 3.4.1 Ziele der Analyse 24
  - 3.4.2 Kontrollflußanalyse 25
  - 3.4.3 Datenflußanalyse (DFA) 27

3.4.4 DFA-Spezifikation der Beispiele 36
3.4.5 Static Single Assignment Form 37
3.4.6 Schlußfolgerungen 39
3.5 Literaturüberblick über die Analyse und Darstellung paralleler Programme . 40
3.5.1 Verifikationsmethoden 40
3.5.2 Erkennung von Datenzugriffskonflikten 40
3.5.3 Kontrollflußanalyse 40
3.5.4 Datenflußanalyse 40
3.5.5 Static Single Assignment Form 41
3.5.6 Eigene Arbeiten 42
3.5.7 Schlußfolgerungen 42

4 Theorie der Datenflußanalyse paralleler Programme (pDFA) 43
4.1 Die Aufgabe 43
4.2 Der Lösungsansatz 44
4.3 Eigenschaften des $\mathcal{F}^{\mathcal{C}}$ und $\mathcal{F}^{\mathcal{B}}$ DFA-Rahmens 45
4.3.1 Definition und Eigenschaften des $\sqcap$-Operators und des $\boxast$-Operators . . 45
4.3.2 Zusammenhang zwischen $\sqcap$ und $\boxast$ 47
4.3.3 Eigenschaften von Kompositionsketten und Präfixen 49
4.4 DFA für pPL_0 Programme 50
4.4.1 Welche Information ist am Ende einer PAR-Anweisung gültig 50
4.4.2 Welche DFA-Information erreicht eine Anweisung innerhalb eines Prozeßrumpfes 53
4.5 DFA für pPL Programme 57
4.5.1 Welche Information ist am Ende einer PAR-Anweisung gültig 57
4.5.2 Welche DFA-Information erreicht eine Anweisung innerhalb eines Prozeßrumpfes 59
4.6 DFA für pPL_{sync} und pPL_{lock} Programme 60
4.7 Schlußfolgerungen 61

5 Datenstrukturen und Algorithmen 63
5.1 Notationen 63
5.2 Der parallele Kontrollflußgraph (pCFG) 63
5.2.1 Alternative Darstellungen 66
5.3 Die Dominanzrelation und Dominanzgrenze im pCFG 67
5.4 Berechnen der Menge sibl 68
5.5 Der Strukturansatz für parallele Programme 69
5.6 Der iterativ-rekursive pDFA-Algorithmus 70
5.6.1 Behandlung von Prozeduraufrufen 70
5.6.2 $\mathcal{D}^{\mathcal{C}}$-Probleme 70
5.6.3 $\mathcal{D}^{\mathcal{B}}$-Probleme 72
5.7 Parallel Static Single Assignment Form 75
5.7.1 Die ϕ-Funktion 75
5.7.2 Die ψ-Funktion 76
5.7.3 Die χ-Funktion 77
5.7.4 Algorithmen zur Bestimmung der pSSA-Nummern und pSSA-Funktionen 78
5.8 Schlußfolgerungen 83

6 Von der Theorie zur Praxis **85**
6.1 Optimierung durch Codeverschiebung – Einleitung 85
6.2 Codeverschiebung und die PAR-Anweisung 86
6.3 Elimination gemeinsamer Teilausdrücke 87
6.4 Lazy Code Motion 88
6.4.1 Beschreibung des Verfahrens 88
6.4.2 Implementierung – Wie werden die Grundprädikate bestimmt? 91
6.4.3 Übertragung auf parallele Programme 92
6.5 Implementierung 93
6.6 Messungen 93
6.7 Schlußfolgerungen 95

7 Zusammenfassung und Ausblick **97**
7.1 Das Problem 97
7.2 Beitrag dieser Arbeit 97
7.3 Der Lösungsweg 98
7.4 Ausblick 99

A Beweise **101**
Lemma 3.23 101
Korollar 4.4 101
Satz 4.9 101
Satz 4.16 102
Korollar 4.24 102
Korollar 4.27 103
Satz 4.29 104
Lemma 4.36 104
Korollar 4.37 105
Satz 4.39 105

Literatur **107**

Stichwortverzeichnis **115**

Tabellarischer Lebenslauf **121**

Eigene Veröffentlichungen **123**

Tabelle der Algorithmen

3.56 Sequentielle DFA 35
3.59 Transformation in SSA-Form 39
4.40 dfa_par_stmt$^{\mathcal{C}}$ 58
5.5 p_Dom 67
5.6 p_DF 68
5.8 p_DFA$^{\mathcal{C}}$ 70
5.9 p_DFA$^{\mathcal{C}}$_cfg 71
5.10 p_DFA$^{\mathcal{B}}$ 72
5.11 p_DFA$^{\mathcal{B}}$_GenKill 72
5.12 p_DFA$^{\mathcal{B}}$_GenKill_cfg 73
5.14 p_DFA$^{\mathcal{B}}$_InOut 74
5.15 p_DFA$^{\mathcal{B}}$_InOut_cfg 74
5.16 p_SSA 78
5.17 p_Insert_ϕ 79
5.18 p_Insert_χ 80
5.19 p_Rename_Basic_Block 80
5.20 p_Rename_Expr 82
5.21 p_Rename_Par_Stmt 82

1 Einleitung

Ziel dieser Dissertation ist es, Methoden zur Verbesserung der Rechenzeit paralleler Programme mittels Optimierung durch den Übersetzer zu entwickeln. Dazu wird erstens die theoretische Grundlage der Datenflußanalyse paralleler Programme hergeleitet, zweitens aus dieser praktische und effiziente Verfahren zur Analyse paralleler Programme abgeleitet und drittens aufgezeigt, daß sich, basierend auf diesen Analysemethoden, bekannte Codetransformationsverfahren einfach in einen Übersetzer für parallele Programmiersprachen integrieren lassen.

1.1 Ziele, Motivation, Lösungsansatz, Resultate dieser Arbeit

Die beiden folgenden Bemerkungen sind inzwischen unter Softwareentwicklern anerkannt:

1. Höhere Programmiersprachen und deren Übersetzer bilden die Implementierungsgrundlage moderner Software.
2. Parallelität und Nebenläufigkeit der Algorithmen und Rechnerhardware stellen sich immer mehr als eine der wichtigsten Methoden, sowohl zur Lösung sehr rechenintensiver Probleme, als auch der Implementierung inhärent paralleler Gegebenheiten heraus.

Durch die größere Abstraktion von der Zielmaschine, auf der ein Programm ablaufen soll, entstehen, im Vergleich zur (sehr aufwendigen) maschinennahen Programmierung, nicht zu vernachlässigende Effizienznachteile. Auf der anderen Seite unterliegen parallele Rechnerarchitekturen einem so schnellen Wandel, daß man bei der Erstellung paralleler Programme weitgehend von der konkreten parallelen Hardware abstrahieren muß, wenn sich Softwareinvestitionen langfristig auszahlen sollen.

Für den Übersetzer einer Programmiersprache ergibt sich somit neben der Aufgabe, aus dem Quellprogramm ein ablauffähiges Maschinenprogramm zu erzeugen, die Aufgabe, den Code so zu erzeugen, daß er an die Qualität direkter maschinennaher Programmierung heranreicht. Der Teil des Übersetzers, der diese Aufgabe erfüllt, wird *Optimierer* genannt. Ziel der Optimierung ist es, die Laufzeit (und früher wichtiger als heute, den Speicherbedarf) eines Programms zu verringern. Dies kann durch

- Reduktion der Rechenzeit (durch Ausführen von weniger oder „billigeren“ Maschinenbefehlen),
- Reduktion der Kommunikationszeiten (Prozessor zum Speicher, Prozessor zu anderen Prozessoren) oder
- Erhöhung der Nebenläufigkeit der Programmausführung

geschehen. Für sequentielle Rechnersysteme, sequentielle Programmiersprachen und deren Übersetzer sind Methoden zur Optimierung ein gut untersuchtes und verstandenes Gebiet der Informatik.

In die Laufzeit eines parallelen Programms gehen, weitaus stärker als für sequentielle Programme, die Kommunikationskosten ein, insbesondere der Zugriff auf den gemeinsamen oder verteilten Speicher. Wie wichtig dabei die Optimierung der Kommunikation in parallelen Programmen ist, belegt die Tatsache, daß sie i.d.R. um Größenordnungen langsamer als z.B. eine Gleitkommamultiplikation ist. Methoden zur Verbesserung (z.B. Reduktion der Anzahl der Kommunikationen, Überlappung von Kommunikation mit Berechnungen, Bündelung mehrerer Nachrichten zu einer, Vergrößerung der Nachrichtenlänge, etc.) sind z.B. in [PHILIPPSEN, 1993; ZIMMERMANN und LÖWE, 1994; LOEWE, 1996] zu finden. In dem Maße, wie diese Methoden die Laufzeiten verbessern, fällt die Rechenzeit der Prozesse wieder mehr ins Gewicht. Allerdings wird auf rechenzeitverbessernde Optimierungen in Übersetzern meist verzichtet, da die zugrundeliegenden Analysetechniken nur für sequentielle Programme zulässig sind.

In dieser Arbeit sollen die Fragen erörtert werden, die sich für die Analyse und (rechenzeitverbessernde) Transformation paralleler Programme ergeben.

Damit eine Optimierung, d.h. Programmtransformation, korrekt ist, muß sie für *alle möglichen* Programmausführungen korrekt sein. Die *Datenflußanalyse* (DFA) berechnet dazu aus dem Quellprogramm die Bedingungen, unter denen eine Transformation durchgeführt werden darf. KILDALL hat in seiner richtungweisenden Arbeit 1973 das Problem der u.U. exponentiell vielen möglichen Programmausführungen sequentieller Programme gelöst, indem er die DFA mittels Funktionen über beschränkten Halbverbänden modelliert hat. Man erhält dadurch einen einfachen Fixpunktalgorithmus, der für „reale“ Programme nur wenige Iterationen bis zur Konvergenz benötigt. Leider ist dieses Datenflußanalyseverfahren nur für sequentielle Programme geeignet.

Nehmen wir als Quellsprache eine parallele Programmiersprache an, die einen gemeinsamen Speicher für alle nebenläufig ausgeführten Prozesse vorsieht, so müssen für die DFA neben der nicht vorhersagbaren Sprungstruktur des Programms auch noch alle Interleavings der nebenläufig ausgeführten Anweisungen[1] betrachtet werden. Etwas formaler: sind P_1 und P_2 (aus vielen Anweisungen s bestehende) Prozeßrümpfe, die nebenläufig ausgeführt werden (`PAR` P_1 `|` P_2 `END`), dann bestimmt sich die DFA-Information, die nach der `PAR`-Anweisung gültig ist, durch[2]: $\bigsqcap_{\langle s_1;\ldots;s_n\rangle \in \text{Interleavings}(P_1,P_2)} f_{\langle s_1;\ldots;s_n\rangle}(x)$. Die Herausforderung einer Erschließung der DFA für parallele Programme ist es, die Zustandsexplosion in den Griff zu bekommen, welche durch die in der Anzahl der nebenläufig ausgeführten Anweisungen exponentiell vielen Interleavings entsteht.

Die Lösung basiert auf der „Entdeckung“ und Annahme zweier Eigenschaften von unären Funktionen über dem Halbverband $(\mathcal{L}, \sqsubseteq, \sqcap)$, der einer Datenflußanalyse zugrunde liegt:

1. Als semantische Funktionen, welche die Bedeutung einer Anweisung für die Datenflußanalyse beschreiben, kommen nur die Identitäts- und Konstantenfunktionen in Frage: $\mathcal{F} = \{\mathsf{id}\} \cup \{\mathsf{const}_c \mid c \in \mathcal{L}\}$ mit $\forall x \in \mathcal{L} : \mathsf{id}(x) \stackrel{\text{def}}{=} x$ bzw. $\mathsf{const}_c(x) \stackrel{\text{def}}{=} c$

2. Diese Funktionenräume können $\sqcap$-abgeschlossen sein: $\forall f, g \in \mathcal{F} : f \sqcap g \in \mathcal{F}$. (Für allgemeine DFA-Rahmen wird nur die Kompositionsabgeschlossenheit gefordert: $\forall f, g \in \mathcal{F} : f \circ g \in \mathcal{F}$.)

[1] Werden z.B. die Zuweisungen $s_1; s_2$ und s_3 nebenläufig ausgeführt (`PAR` $s_1; s_2$ `|` s_3 `END`), entspricht dies in der Interleavingsemantik dem sequentiellen Programm `CASE RANDOM() MOD 3 OF 0:` $s_1; s_2; s_3$ `| 1:` $s_1; s_3; s_2$ `| 2:` $s_3; s_1; s_2$ `END`.

[2] f_s (bzw. $f_{\langle s_1;\ldots;s_n\rangle}$) ist die semantische Funktion, die beschreibt, wie die Ausführung der Anweisung s (bzw. Anweisungsfolge $\langle s_1; \ldots; s_n\rangle$) die DFA-Information verändert. $\sqcap$ ist der „Vereinigungsoperator“, der die DFA-Informationen aus verschiedenen Pfaden bzw. Interleavings zusammenfaßt. In Kapitel 3 werden die Begriffe formal definiert.

Obwohl diese Bedingungen sehr restriktiv erscheinen, umfassen sie die große Menge der auf dem booleschen Halbverband basierenden Bitvektorprobleme. Diese bilden die Grundlage der wichtigsten (Rechenzeit verbessernden) Optimierungsmethoden, wie Elimination gemeinsamer Teilausdrücke, Elimination toten Codes oder Elimination partieller Redundanzen.

Basierend auf diesen Annahmen können wir zeigen, daß für die korrekte Datenflußanalyse von Bitvektorproblemen *keine* Interleavings untersucht werden müssen! Etwas formaler: Seien $P_1, P_2, \ldots, P_n$ (aus vielen Anweisungen s bestehende) Prozeßrümpfe, die nebenläufig durch die `PAR` P_1 `| ... |` P_n `END` Anweisung ausgeführt werden, dann kann, ohne alle Interleavings $\langle s_1; \ldots; s_m \rangle$ bestimmen zu müssen, $\bigsqcap_{\langle s_1;\ldots;s_m\rangle \in \text{Interleavings}(P_1,\ldots,P_n)} f_{\langle s_1;\ldots;s_m\rangle}(x)$ berechnet werden. Es reicht aus, die in Prozessen nebenläufig ausgeführten Anweisungen separat zu betrachten und diese DFA-Ergebnisse geeignet (mit Kosten $O(n)$, wobei n die Anzahl der nebenläufigen Prozeßrümpfe P_i ist) zusammenzufassen. Dies wird bewerkstelligt, indem aus den Annahmen Bitvektorgeleichungen abgeleitet werden, die die DFA-Information einer `PAR`-Anweisung aus denen der Prozeßrümpfe bestimmen.

Lassen wir die zweite Annahme ($\sqcap$-Abgeschlossenheit) fallen, kann der Halbverband größer sein, d.h. mehr als nur zwei Elemente haben. (Wir können zeigen, daß der boolesche DFA-Rahmen bis auf Isomorphie der einzige DFA-Rahmen ist, der beide Bedingungen erfüllt.) Der Preis, der für diese umfangreichere Wertemenge bezahlt werden muß, ist mäßig.

Damit haben wir das Problem der Zustandsexplosion für die wichtigste Klasse von DFA-Problemen (der Bitvektorprobleme) und deren Verallgemeinerung gelöst. Die Durchführung höchst wirksamer (nicht nur Rechenzeit vermindernder) Optimierungen ist nun möglich.

Die meisten der bisher bekannten Ansätze zur Analyse paralleler Programme schränken die zu untersuchende Programmiersprache stark ein, indem sie keinen gemeinsamen Speicher für nebenläufig ausgeführte Prozesse vorsehen. Analyseansätze anderer Autoren sind aufgrund ihrer außerordentlich großen Zeit- und Speicherkomplexität in der Praxis nicht einsetzbar. Die Ergebnisse unserer Arbeit ermöglichen es, parallele Programme (mit gemeinsamem Speicher) mit dem gleichen Aufwand zu analysieren, wie „vergleichbare" sequentielle.

Aus unseren Sätzen ergeben sich Hinweise, wie ein Kontrollflußgraph aufgebaut sein muß, der ein paralleles Programm darstellen soll. Mit einer modifizierten Version des Algorithmus von Kildall kann der Fixpunkt von Datenflußgleichungen über diesen parallelen Kontrollflußgraphen berechnet werden. Eine andere Anwendung ist die Erweiterung der *Static Single Assignment Form* [Cytron *et al.*, 1989] für parallele Programme.

Mit der „richtigen" DFA-Information ausgestattet, können nun bekannte Transformationen auch auf parallele Programme angewendet werden. Beispielhaft wurde im Rahmen des *ESPRIT*-Projektes Compare für die Quellsprache *Modula-P* [Vollmer, 1989] die Elimination gemeinsamer Teilausdrücke und Elimination partieller Redundanzen im Stil der *Lazy Code Motion* [Knoop *et al.*, 1994] implementiert. Messungen belegen, daß die Optimierung paralleler Programme effektiv und effizient ist.
Die drei Hauptergebnisse dieser Arbeit sind somit:

1. Die Theorie der Datenflußanalyse paralleler Programme (pDFA),
2. Datenstrukturen (pCFG, pSSA) und Algorithmen zur Analyse paralleler Programme, und
3. Implementierung von zwei höchst wirksamen Optimierungen für parallele Programme: die Elimination gemeinsamer Teilausdrücke sowie die *Lazy Code Motion*.

Mit dieser Arbeit stellen wir folgende These auf:

Die Optimierung paralleler Programme ist effektiv und effizient, einfach zu implementieren und hat keine Nachteile für die Optimierung sequentieller Programme, sowie sequentieller Programmteile innerhalb eines parallelen Programms.

1.2 Gliederung der Arbeit

Die Arbeit gliedert sich in drei Hauptteile:

1. Entwicklung der Theorie der Datenflußanalyse paralleler Programme (pDFA),
2. Entwurf der Datenstrukturen und Algorithmen zur Analyse paralleler Programme und
3. dem Schritt „von der Theorie zur Praxis“: Anwendung der Ergebnisse auf konkrete Optimierungsaufgaben: der Elimination gemeinsamer Teilausdrücke und der *Lazy Code Motion*.

Anhand eines einfachen Beispiels soll in Kapitel 2 aufgezeigt werden, was alles bei einer einfachen Übertragung sequentieller Optimierungstechniken auf parallele Programme schiefgehen kann, bzw. was eigentlich erwünscht ist. Die Grundlagen der Optimierung sequentieller Programme, paralleler Rechner und paralleler Programmiersprachen sowie einen Überblick über die aktuelle Literatur zur Analyse paralleler Programme sind in Kapitel 3 gegeben. Der Hauptbeitrag dieser Arbeit wird in den nächsten drei Kapiteln beschrieben: Kapitel 4 stellt die Theorie der Datenflußanalyse paralleler Programme (pDFA) vor und beweist die grundlegenden Sätze. Diese werden in Kapitel 5 dazu benutzt, Datenstrukturen und Algorithmen zu entwerfen bzw. die in Kapitel 3 eingeführten sequentiellen Methoden auf den parallelen Fall zu übertragen. Im folgenden Kapitel 6 wird mit ihnen ein konkretes Optimierungsverfahren (Elimination gemeinsamer Ausdrücke und partieller Redundanzen mittels der *Lazy Code Motion*) implementiert und vermessen. Kapitel 7 bildet den Abschluß und faßt die Resultate der Arbeit zusammen. Einfache Beweise, die nicht im Hauptteil vorkommen, sind in Anhang A angegeben.

1.3 Danksagungen

Die vorliegende Dissertation entstand während meiner Arbeit an dem *ESPRIT*-Projekt Compare bei der ehemaligen GMD-Forschungsstelle in Karlsruhe und später am Lehrstuhl von Professor Goos, Universität Karlsruhe. Ich möchte all denen danken, die zu dieser Arbeit beigetragen haben. Mein Dank gilt den Kollegen der Forschungsstelle und des Lehrstuhls sowie den Mitarbeitern (aus vier europäischen Ländern) des Compare-Projektes. Namentlich möchte ich vor allem die Karlsruher Kollegen Uwe Aßmann, Helmut Emmelmann, Thomas Müller und Jens Knoop von der Universität Passau erwähnen. Mein Dank gilt auch den Studenten Markus Armbruster, Holger Hopp, Christian von Roques, die bei der Implementierung des Modula-P-Übersetzers im Rahmen des Compare-Compilers mitgearbeitet haben. Andreas Winter implementierte als Diplomarbeit Teile des Optimierers [Winter, 1995]. Bedanken möchte ich mich auch bei Cornelia, meiner Frau, für unermüdliches Korrekturlesen. Professor Jähnichen (GMD Berlin) danke ich für seine Unterstützung in den Jahren an der GMD-Forschungsstelle Karlsruhe, die ein ausgezeichnetes Umfeld für Forschungsarbeiten bot. Professor Goos und Professor Steffen (Universität Passau), der das Zweitgutachten übernahm, danke ich für ihre Betreuung und anregenden Diskussionen während der Entstehung dieser Arbeit.

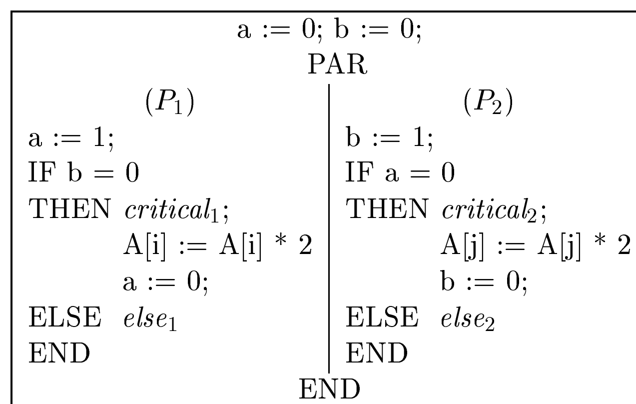
Jürgen Vollmer, im Dezember 1997

2 Ein einführendes Beispiel

Anhand eines einfachen Beispiels soll in diesem Kapitel aufgezeigt werden, was alles bei einer naiven Übertragung sequentieller Optimierungstechniken auf parallele Programme schiefgehen kann, bzw. was eigentlich erwünscht ist. Es zeigt ebenfalls, daß selbst ohne Optimierung auf der Zwischensprachebene die Umordnung von Maschinenbefehlen (durch z.B. den Assemblierer) zu falschem Maschinencode führen kann, wenn nicht Rücksicht auf die parallele Natur des Quellprogramms genommen wird. Und noch schlimmer: Da manche Prozessoren, wie der *DEC Alpha*, in gewissem Umfang Speicherzugriffe automatisch umordnen, kann es zur Laufzeit des parallelen Programms zu falschen Ergebnissen kommen.

2.1 Das Beispiel

Das Beispiel in Abbildung 2.1 zeigt einen Ausschnitt eines Synchronisierungsprogramms. Es basiert auf [LAMPORT, 1979] und wurde im Zusammenhang mit dem Design von Multiprozessoren geschildert. Obwohl kurz, ist es typisch, da es gewollten Indeterminismus ausdrückt. Viele „höhere" Programmiersprachenkonstrukte, wie Semaphore und Monitore, können darauf abgebildet werden.



Zuerst wird die Initialisierung von `a` und `b` ausgeführt, danach die `PAR` Anweisung, d.h.: die Programmanweisungen rechts und links des senkrechten Striches werden parallel und unabhängig voneinander ausgeführt. Jeder Prozeß kann lesend und schreibend auf die gemeinsamen Variablen zugreifen.

Abbildung 2.1: Teil eines Synchronisierungscodes.

Intuitiv ist klar, daß niemals beide Anweisungen *critical*$_1$ und *critical*$_2$ gleichzeitig ausgeführt werden können, da immer nur ein Prozessor eine Speicherzelle (Variable) schreiben kann. Allerdings können beide *else*-Teile ausgeführt werden. Auf den ersten Blick mag es erscheinen, daß dieses Programm einen Speicherzugriffskonflikt (engl: *race condition*) [NETZER und MILLER, 1992] hat, da beide Prozesse gemeinsame Variablen lesen und schreiben, ohne daß diese

Zugriffe synchronisiert werden. Aber dieses Verhalten ist gerade beabsichtigt. Es soll ein Synchronisationsmechanismus implementiert werden: entweder wird $critical_1$ oder $critical_2$ oder keines von beiden ausgeführt.

Eine offensichtlich erwünschte Optimierung ist, die Adresse des Feldelementes $A[i]$ (bzw. $A[j]$) als gemeinsamen Teilausdruck zu erkennen und zu eliminieren, also: Die Anweisung $A[i] := A[i] * 2$ soll durch $t := adr(A[i]); t := t \uparrow *2$ mit der Hilfsvariablen (Register) t ersetzt werden (analog im Prozeßrumpf P_2).

2.2 Probleme eines einfaches Optimierers mit parallelen Programmen

Die naive Übertragung sequentieller Optimierungstechniken könnte wie folgt lauten: Da beide Prozeßrümpfe immer ausgeführt werden, betrachten wir sie einfach getrennt, ohne Berücksichtigung der Zugriffe auf gemeinsame Variable. Das würde dann erlauben, die Werte der Variablen `a` und `b` nach ihrer Initialisierung an die Ausdrücke `a = 0` bzw. `b = 0` der `IF`-Anweisungen fortzupflanzen. Da sich diese Bedingungen dann immer zu TRUE auswerten, kann man auf die Auswertung dieser Ausdrücke zur Laufzeit verzichten, es muß auch kein Code für die beiden *else*-Teile erzeugt werden. Als Folge davon ergibt sich, daß beide *critical*-Teile ausgeführt werden – was natürlich der Absicht des Quellprogramms widerspricht und somit eine falsche Optimierung darstellt.

Obwohl für dieses Beispiel ein gemeinsamer Speicher vorausgesetzt wird, treten die gleichen Probleme bei einem System mit verteiltem Speicher auf, wenn die Zugriffe auf die gemeinsamen Variablen durch Aufrufe an das Betriebsystem ersetzt werden, welche die Werte vom entfernten Speicher holen.

2.3 Probleme mit der Befehlsumordnung in parallelen Programmen

Bei modernen RISC[1] Prozessoren erfolgt die interne Befehlsabarbeitung parallel durch sogenannte Fließbandverarbeitung (*pipelining*). Vereinfacht kann das wie folgt beschrieben werden: Der Prozessor besteht aus mehreren Funktionseinheiten, wie Befehlsdekodierer, Adreßberechnungseinheit, Operandenleseeinheiten und Operandenschreibeinheiten, arithmetisch-logische Recheneinheit. Diese Einheiten sind so hintereinander angeordnet, daß jede Einheit zu einer gegebenen Zeit einen Teil einer Maschineninstruktion bearbeitet und den Befehl mit dem nächsten „Takt" an die folgende Einheit weitergibt. Diese verzahnte Ausführung erlaubt einen größeren Befehlsdurchsatz. Eine genauere Beschreibung findet man z.B. in [Ungerer, 1995].

Tabelle 2.1 zeigt dem Prozeßrumpf P_1 entsprechenden Maschinencode, wie er für einen typischen RISC Prozessor erzeugt werden würde.

ldc	1, r0	Lade die Konstante `1` in Register `r0`.
st	r0, a	Speichere den Inhalt des Registers `r0` unter der Adresse von `a` ab.
ld	b, r1	Lade den Inhalt vom Speicher mit Adresse `b` in Register `r1`.
cmp	r1, 0	Vergleiche Inhalt eines Registers mit einer Konstante; setze Bedingungscode.
jeq	$then_1$	Bedingter Sprung zu $\texttt{then}_1$, wenn Bedingungscode *gleich* gesetzt ist.
	...	*Programm Code für den* $else_1$ *Teil*

Tabelle 2.1: Code für Prozeßrumpf P_1.

[1] **RISC**: *Reduced Instruction Set Computer*

Gegenwärtig sind die meisten Prozessoren deutlich schneller als ein Hauptspeicherzugriff. Deshalb kann es passieren, daß ein Prozessor „warten" muß, bis das Ergebnis einer Speicherleseoperation in einem Register präsent ist. Aber eigentlich müßte der Prozessor erst warten, wenn eine andere Operation diesen Wert verwenden will. Im allgemeinen kann es also günstiger sein, daß auf das Schreiben eines Registers nicht unmittelbar ein lesender Zugriff erfolgt. Ein Befehlsumordner (*instruction scheduler*) würde deshalb den Code aus Abbildung 2.1 durch den folgenden Code ersetzen:

```
ldc   1, r0
ld    b, r1
st    r0, a
cmp   r1, 0
jeq   then1
...
```

Tabelle 2.2: Umgeordneter Code von P_1.

Wird der Code von Prozeßrumpf P_2 entsprechend umgeordnet, so kann es passieren – wenn beide Prozessoren quasi synchron, aber doch unabhängig ihre Befehle abarbeiten – daß beide kritischen Programmteile $critical_i$ gleichzeitig ausgeführt werden:

Takt	t_1	t_2	t_3	t_4	...	
Prozessor$_1$:	ldc 1, r0;	ld b, r1; *(r1 = 0)*	st r0, a;	cmp r1, 0;	...	$critical_1$
Prozessor$_2$:	ldc 1, r0;	ld a, r1; *(r1 = 0)*	st r0, b;	cmp r1, 0;	...	$critical_2$

Tabelle 2.3: Eine mögliche Ausführungsfolge des Beispielprogramms aus Tabelle 2.2.

Damit Verfahren zur Befehlsumordnung (z.B. [MÜLLER, 1995]) auch auf parallele Programme angewendet werden können, müssen deren Programmanalysen für parallele Programme erweitert werden. Dazu können die in unserer Arbeit vorgestellten Datenflußanalysemethoden benutzt werden.

2.4 Probleme mit dem Vertauschen von Speicherzugriffen bei parallelen Programmen

Moderne Rechnersysteme haben mehrere Speicherbänke, auf die gleichzeitig zugegriffen werden kann. Sind die Adressen verschieden, kann es sinnvoll sein, daß die Speichereinheit Zugriffe in einer anderen Reihenfolge ausführt, als dies der Prozessor vorgibt. Bei sequentiellen Programmen macht dies keinen Unterschied[2].

Bei parallelen Programmen kann selbst ohne Befehlsumordnung diese automatische Speicherzugriffsumordnung zum gleichen aber falschen Ergebnis führen. Um diesen Effekt zu vermeiden, bieten diese Prozessoren eine sogenannte *memory barrier* Instruktion an, die erzwingt, daß alle Speicherzugriffe, die vor ihr abgesetzt wurden, abgeschlossen werden, bevor die nächste Instruktion ausgeführt werden kann. Würde diese Instruktion allerdings nach jedem Speicherzugriff ausgeführt, wäre eine deutliche Verlangsamung der Programmausführung die Folge.

In einem Übersetzer kann der Optimierer versuchen, möglichst wenige dieser Instruktionen zu erzeugen.

[2]Eine Diskussion verschiedener Speichermodelle und ihrer Auswirkungen kann z.B. in [GHARACHORLOO *et al.*, 1990; SPARC, 1992] gefunden werden.

2.5 Eine mögliche Lösung dieser Probleme

LAMPORT [1979] bietet eine Lösung an, die von AFEK *et al.* [1993] noch weiter formalisiert wird: Die Speicherzugriffe müssen folgende Bedingungen erfüllen:

1. Jeder Prozessor führt Speicherzugriffe in der Reihenfolge aus, in der sie im Programm erscheinen.
2. Alle Zugriffe auf eine einzelne Speicherzelle werden in der Reihenfolge ihres Auftretens abgearbeitet.

Es ist klar, daß diese Bedingungen viel zu restriktiv sind, da sie viele Optimierungen verbietet.

2.6 Schlußfolgerungen

MIDKIFF und PADUA [1990] geben noch weitere Beispiele, „… *demonstrating how the standard techniques fail when applied to determinate and non-determinate parallel programs.*"

Der Versuch, das Beispielprogramm zu optimieren, scheitert daran, daß die zugrundeliegende Information (der Wert der Variablen `a` bzw. `b` erreicht die `IF` Anweisungen auf jeden Fall) falsch ist. Die Umordnung der Instruktionen muß fehlschlagen, da die Abhängigkeit, die zwischen den Lade- und Speicherbefehlen der beiden Prozesse besteht, nicht berücksichtigt wird. Die Hardware, welche die Speicherzugriffe automatisch umordnet, kann solche Abhängigkeiten ebenfalls nicht erkennen.

In allen drei Fällen zeigt sich also: falsche oder fehlende Datenflußinformation kann zu Problemen bei der Ausführung paralleler Programme führen.

Deshalb kann der folgende Ratschlag häufig in Compiler-Handbüchern gefunden werden [PARSYTEC, 1994 *Programmer's Guide*, S. 18]:

> ***2.9 Multithreaded Programming and Compilers***
>
> *In a multithreaded environment special attention has to be payed to the use of variables which are shared by multiple threads. Using an optimization level greater than 0 during compilation allows the compiler to optimize code, which may result in an unintended behavior of the parallel program.*

Der Rest dieser Arbeit wird zeigen, wie man trotzdem korrekte und effektive Optimierungen eines parallelen Programms effizient durchführen kann.

3 Grundlagen, Problemstellung und Literatur

In diesem Kapitel werden die Grundlagen der Optimierung sequentieller und kontrollflußparalleler Programme vorgestellt, sowie ein Überblick über die aktuelle Literatur gegeben.

Zuerst definieren wir den Rahmen unserer Untersuchung: Dazu geben wir ein Maschinenmodell und die parallele Quellsprache (*pPL*) an, die optimiert werden soll (Abschnitt 3.1). Die *Interleavingsemantik* beschreibt die Bedeutung der Ausführung eines parallelen Programms auf diesem Maschinenmodell.

Aufgrund der Betrachtung des Begriffs *Korrektheit eines Optimierers* stellen wir fest, daß die Programmanalyse, die einer optimierenden Transformation vorausgeht, wie im sequentiellen Fall alle möglichen Programmausführungen berücksichtigen muß. Für parallele Programme heißt dies: alle Interleavings.

Daraus ergibt sich die Problemstellung unserer Arbeit: Da es exponentiell viele Interleavings gibt, ist explizite Untersuchung aller Interleavings praktisch nicht möglich. Ein erster Literaturüberblick zeigt, daß es bisher keine praktikablen Ansätze zur Lösung dieses Problems gibt (Abschnitt 3.3).

Die grundlegenden Definitionen, Sätze und Methoden der Programmanalyse (für sequentielle Programme) werden in Abschnitt 3.4 gegeben. Für den theoretischen Teil dieser Arbeit (DFA für parallele Programme) sind die Abschnitte 3.4.3.4 und 3.4.3.5 besonders wichtig[1]. Der praktische Teil der Arbeit (Datenstrukturen und Algorithmen) basiert auf den in Abschnitt 3.4.3.6 und 3.4.5 dargestellten Methoden. Beispiele für typische DFA-Probleme sind in Abbildung 3.3 und Abschnitt 3.4.4 zu finden.

Den Abschluß dieses Kapitels bildet ein Literaturüberblick über das Gebiet der *Analyse kontrollflußparalleler Programme*, soweit diese Arbeiten für unsere relevant sind (Abschnitt 3.5).

3.1 Die Umgebung: das Rechnermodell und die parallele Programmiersprache *pPL*

In diesem Abschnitt werden das dieser Arbeit zugrundeliegende parallele Rechnermodell und die parallele Programmiersprache *pPL* vorgestellt.

3.1.1 Das Rechnermodell

Zur Charakterisierung verschiedener Rechnerarchitekturen führt man sogenannte *Architekturtaxonomien* ein. Am bekanntesten ist die Taxonomie von Flynn [1966]. Sie geht von zwei

[1] Der Kenner der DFA-Terminologie kann die anderen Teile des Abschnittes 3.4 überschlagen.

Fragen aus: Bearbeitet die Maschine zu einem Zeitpunkt einen (*single*, S) oder mehrere (*multiple*, M) Befehle (*instruction*, I), und arbeitet sie mit einem oder mehreren Datenelementen (*data*, D) gleichzeitig? Die drei bekanntesten Kombinationen sind: *SISD*, der sequentielle von-Neumann-Rechner; *SIMD*, z.B. Vektorrechner; und ***MIMD***, der Rechnertyp, der unserer Arbeit zugrunde liegt. Die Optimierung von Programmen für SIMD-Rechner ist nicht Teil unserer Arbeit. Einen Überblick der dazu nötigen Methoden geben z.B. [Zima und Chapman, 1990; Wolfe, 1989, 1996].

Prozessoren eines MIMD-Systems lösen eine Aufgabe i.d.R. gemeinsam, sie müssen daher miteinander kommunizieren. Dies kann prinzipiell auf zwei Arten geschehen:

1. Jeder Prozessor hat ausschließlich lokalen Speicher, der nur ihm zugänglich ist. Alle Prozessoren sind über eine Anzahl von Kanälen (in)direkt miteinander verbunden. Über diese Kanäle können sie Nachrichten austauschen. Diese Systeme werden Systeme mit *verteiltem Speicher* genannt.
2. Die Prozessoren sind über einen *gemeinsamen Speicher* gekoppelt. Zusätzlich kann jeder Prozessor auch lokalen Speicher haben. Rechner mit *virtuellem gemeinsamen Speicher* sind Systeme, deren Hardware oder Betriebssystem jedem Prozessor die Existenz eines gemeinsamen Speichers vortäuscht. Dazu werden beim Zugriff auf eine virtuelle Speicherzelle Nachrichten zwischen dem anfordernden Prozessor und dem Prozessor, auf dem die (der virtuellen Speicherzelle entsprechende physikalische) Speicherzelle residiert ausgetauscht.

Ein weiteres Merkmal paralleler Rechner ist, wie der gleichzeitige Zugriff mehrerer Prozessoren auf *eine* Speicherzelle aufgelöst wird. Es ist möglich, daß entweder höchstens ein Prozessor zu einem Zeitpunkt zugreifen darf (*exclusive*, E), oder mehreren ist dies gleichzeitig erlaubt (*concurrent*, C). Unterscheidet man dann noch zwischen Lese- (*read*, R) und Schreibzugriffen (*write*, W) erhält man die, aus der Parallel Random Access Machine (PRAM) „Welt" bekannte Klassifikation (s. z.B. [Karp und Ranachandran, 1990]). Reale MIMD-Rechnersysteme implementieren entweder die **EREW** oder die **CREW-Speicherkonfliktauflösungsstrategie**. Abbildung 3.1 zeigt einen MIMD-Rechner schematisch.

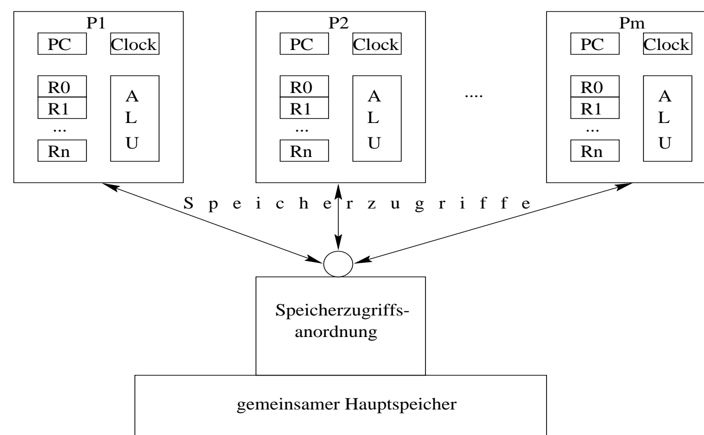
3.1 Definition (Angenommenes Rechnermodell)
Dieser Arbeit liegt ein MIMD-System mit gemeinsamen (virtuellem) Speicher zugrunde. Als Speicherkonfliktauflösungsstrategie wird CREW angenommen. □

3.1.2 Parallele Programmiersprachkonzepte

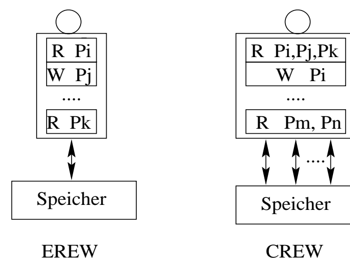
3.2 Definition (Prozess [Andrews und Schneider, 1988])
Ein **sequentielles Programm** spezifiziert die sequentielle Ausführung, **Prozeß** genannt, einer geordneten Folge von Anweisungen. Ein nebenläufiges (paralleles) Programm spezifiziert zwei oder mehr sequentielle Programme, die nebenläufig als **parallele Prozesse** ausgeführt werden. □

Wir erweitern diese Definition so, daß auch „geschachtelte" Parallelität erlaubt ist:

3.3 Definition (Prozess)
Ein **sequentielles Programm** spezifiziert die sequentielle Ausführung, **Prozeß** genannt, einer geordneten Folge von Anweisungen. Ein nebenläufiges (paralleles) Programm spezifiziert zwei oder mehr sequentielle oder parallele Programme, die nebenläufig als **parallele Prozesse** ausgeführt werden. □



Jeder Prozessor P_i hat einen eigenen Befehlszähler (PC), einen eigenen Satz von Registern ($R_1, \ldots R_n$), eine eigene Arithmetisch-Logische Einheit (ALU) und eine eigene Uhr ($Clock$). Jeder Prozessor kann über die Speicherzugriffsanordnungseinheit auf den gemeinsamen Speicher zugreifen. Jeder Rechner führt seine Anweisungen asynchron, d.h. unabhängig von allen anderen Prozessoren, aus.



Die Lese- (R) und Schreibzugriffe (W) werden so angeordnet, daß nur genau ein Prozeß zugreifen kann ($EREW$) oder mehrere Prozesse können gleichzeitig lesen, aber nur genau einer darf schreiben ($CREW$).

Abbildung 3.1: Modell eines MIMD-Rechnersystems

Parallele Sprachen sind solche, die es erlauben, die nebenläufige Ausführung von Anweisungen explizit, d.h. mittels spezieller Anweisungen, zu spezifizieren.

Da nebenläufige Prozesse i.d.R. eine Aufgabe gemeinsam lösen, müssen sie miteinander interagieren können. **Kommunikation** ermöglicht einem Prozeß, die Ausführung eines anderen Prozesses zu beeinflussen [Andrews und Schneider, 1988]. Die Kommunikation erfolgt konzeptionell durch die Benutzung einer gemeinsamen Variablen.

Häufig besteht die Notwendigkeit, daß ein Prozeß p eine Aktion a ausführt, bevor ein anderer Prozeß p' eine Aktion a' ausführt. Da aber im MIMD-Modell nichts über die relativen Ausführungsgeschwindigkeiten ausgesagt werden kann, müssen sich die beiden Prozesse synchronisieren. **Synchronisation** kann also als eine Menge von Bedingungen an die Reihenfolge des Auftretens von Ereignissen aufgefaßt werden [Andrews und Schneider, 1988].

Somit benötigen wir als parallele Programmiersprachenkonstrukte zum einen solche, die Prozesse deklarieren und nebenläufig ausführen, zum anderen Konstrukte, die es den Prozessen ermöglichen, Daten auszutauschen und sich zu synchronisieren.

Einen guten Überblick über parallele Sprachen und Konzepte geben z.B. [Andrews und Schneider, 1988; Gehani und McGettrick, 1988; Bal *et al.*, 1989; Shapiro, 1989].

3.1.3 Syntax und Semantik von der Beispielsprache *pPL*

Im folgenden definieren wir unsere kontrollflußparallele, imperative Beispielsprache *pPL* (*Parallel Programming Language*), an Hand derer unsere Untersuchungen durchgeführt werden.

Variablen müssen wie in anderen imperativen Programmiersprachen deklariert werden.

Als Typen seien die skalaren Grundtypen, Felder, Zeiger und Verbunde zugelassen. *pPL* enthält **einfache Anweisungen**, wie die Zuweisungen, Prozeduraufrufe, und **zusammengesetzte Anweisungen**, wie Schleifen oder bedingte Ausführung. Der parallelen Ausführung von Anweisungen unterliegt das MIMD-Rechnermodell mit der CREW-Speicherkonfliktauflösungsstrategie. Die Sprache ist an *Modula-P* [Vollmer, 1989] angelehnt.

Wir führen folgende Notation ein: s soll immer eine *einfache* Anweisung und S eine *zusammengesetzte* bezeichnen. Die Notation $s \in S$ soll heißen, daß s eine einfache Anweisung ist, die in S vorkommt.

3.4 Definition (Prozeduren)
Prozeduren sind vergleichbar zu *Modula-2* vereinbart.

```
Proc          ::= PROCEDURE Identifier Params ";" Decls ProcedureBody ";" .
ProcedureBody ::= BEGIN Stmts END .
```

Die Prozedurparameter `Params` und lokale Deklarationen `Decls` sind „wie üblich" definiert[2].

□

3.5 Definition (*pPL* Anweisungen)
Die parallele Anweisung, die zu den sequentiellen hinzukommt, ist die `PAR`-Anweisung.

```
Stmts ::= ([Label ":"] Stmt) //";" .
Stmt  ::= AssignStmt | ProcCallStmt | IfStmt | WhileStmt | GotoStmt | ... |
          ParStmt .
```

□

Bedingt durch die parallele Ausführung von Zuweisungen, können zwei Bedeutungen der Berechnung eines Ausdrucks definiert werden: Kommt eine Variable mehrfach in einem Ausdruck vor, hat diese Variable bei der Berechnung des Ausdruckswertes zur Laufzeit

- immer den gleichen Wert, oder
- es können sich verschiedene Werte ergeben[3].

Die Eindeutigkeit der Variablenwerte eines Ausdrucks läßt sich durch den Übersetzer ohne expliziten Synchronisationscode sicherstellen, indem zuerst die Werte der benutzten Variablen in Register geladen werden, und zur Ausdrucksberechnung nur auf diese Werte zugegriffen wird.

Eine weitere Anforderung könnte sein, daß die ganze Berechnung atomar ist, d.h. zuerst werden alle benutzten Variablen für andere Prozesse „gesperrt", erst dann kann genau ein Prozeß auf sie zugreifen. Da die Probleme mit dieser Semantik (z.B. Verklemmung von Prozessen, die auf eine Variable zugreifen möchten) weit in den Bereich der Betriebssysteme hineinreichen, erscheint es wenig sinnvoll, die dort bekannten Lösungen in eine Programmiersprache bzw. deren Übersetzer einzubetten. Statt dessen sollten Hilfsmittel zur Synchronisation in der Sprache, bzw. einer Bibliothek vorgesehen werden. Für die Zuweisung kann ähnlich argumentiert werden. Wir definieren deshalb:

[2] In der hier benutzen EBNF Syntaxnotation steht `X//Y` für eine Liste von `X` getrennt durch `Y` und `[X]` beschreibt einen optionalen `X` Teil. `"X"` steht für das terminale Sonderzeichen `X`.

[3] In sequentiellen Sprachen hat man ein vergleichbares Problem, wenn innerhalb eines Ausdrucks Funktionen aufgerufen werden können, die mittels Seiteneffekten Variablen ändern oder wenn die Ausdrücke selbst Zuweisungen enthalten dürfen (wie z.B. in der Programmiersprache *C*).

3.6 Definition (Ausdrücke)
Die Ausdruckssyntax entspricht der von „üblichen“ imperativen Programmiersprachen, z.B. *Modula-2*.
Die **Ausdrucksauswertung** ist weder atomar, noch müssen mehrere Zugriffe auf die gleiche Variable innerhalb des gleichen **Ausdrucks** den gleichen Wert ergeben. □

3.7 Definition (Zuweisung)

```
AssignStmt ::= Variable ":=" Expression.
```

Zuerst wird der Wert des Ausdrucks `Expression` berechnet. Danach wird er der Variablen `Variable` zugewiesen. □

3.8 Definition (PAR Anweisung)
Die `PAR`-**Anweisung** dient dazu, Prozesse zu deklarieren und zu starten.

```
ParStmt     ::= PAR ProcessBody //"|" END .
ProcessBody ::= [Replicator] Stmts .
Replicator  ::= "[" ReplicatorVariable ":" LowerBound TO UpperBound "]" .
LowerBound  ::= Expression .
UpperBound  ::= Expression .
```

Die Prozeßdeklaration (**Prozeßrumpf**) `ProcessBody` ist eine Liste von Anweisungen, denen ein **Replikator** `Replicator` vorausgehen kann. Hat `ProcessBody` keinen Replikator, wird genau ein Prozeß, der die Anweisungsfolge `Stmts` ausführt, erzeugt. Die Ausdrücke `UpperBound` und `LowerBound` sowie die Variable `ReplicatorVariable` müssen vom Typ `INTEGER` sein. Der Wert von `UpperBound` und `LowerBound` muß zur Übersetzungzseit nicht bekannt sein. Ein Replikator erzeugt `UpperBound-LowerBound+1` Prozesse, die alle die gleichen Anweisungen `Stmts` ausführen. Ist `UpperBound-LowerBound+1` ≤ 0, werden hierfür keine Prozesse erzeugt. Die durch `ProcessBody` erzeugten Prozesse werden nebenläufig asynchron ausgeführt. Jeder **replizierte Prozeß** kann auf die **Replikatorvariable** `ReplicatorVariable` nur lesend zugreifen. In jedem Prozeß hat `ReplicatorVariable` einen eindeutigen Wert aus dem Intervall `[LowerBound .. UpperBound]`, d.h. `ReplicatorVariable` ist innerhalb eines Prozeßes eine Konstante, die zur Laufzeit für *diesen* Prozeß bestimmt wird. Der Vaterprozeß, d.h. der Prozeß, der die `PAR`-Anweisung ausführt, wird solange suspendiert, bis alle Kindprozesse terminiert haben. Es darf, mittels einer `GOTO` Anweisung, weder in einen Prozeßrumpf hineingesprungen werden, noch darf aus einem Prozeßrumpf herausgesprungen werden[4]. □

3.1.4 Die Spracherweiterungen pPL_{sync} und pPL_{lock}

Die beiden folgenden Definitionen beschreiben Spracherweiterungen von *pPL*, mit denen explizite Synchronisation dargestellt werden können.

3.9 Definition (Typen)
Es gibt zwei weitere skalare Typen: `SEMA` und `REGION`. Auf den Werten ist als Vergleichsoperation nur die Gleichheit bzw. Ungleichheit definiert. □

3.10 Definition (pPL_{sync}: Die `SIGNAL`/ `WAIT` Anweisung)
Mit der **WAIT** und **SIGNAL** Anweisung können sich zwei Prozesse synchronisieren.

```
InitSigStmt ::= INIT "(" SemaphoreVariable ")" .
SignalStmt  ::= SIGNAL "(" SemaphoreVariable ")" .
WaitStmt    ::= WAIT "(" SemaphoreVariable ")" .
```

[4]Dies ist sicherlich keine große Einschränkung, kann doch solchen Sprüngen nur schwerlich eine Bedeutung zugeordnet werden.

`INIT(SemaphoreVariable)` initialisiert die Variable `SemaphoreVariable`, die vom Typ `SEMA` sein muß, indem sie ihr eine, in diesem Programmlauf eindeutige, Identifikationsnummer *sema_id* zuweist. Ein Prozeß, der die `WAIT` Anweisung ausführt, wird suspendiert, bis ein anderer Prozeß eine `SIGNAL` Anweisung ausführt, deren `SemaphoreVariable` den gleichen Wert hat, und umgekehrt. □

3.11 Definition (pPL_{lock}: Die `LOCK` Anweisung)
Mit der **LOCK** Anweisung können bedingte **kritische Regionen** dargestellt werden.

```
InitLockStmt ::= INIT "(" RegionVariable ")" .
LockStmt     ::= LOCK RegionVariable DO LockedStmts END .
LockedStmts  ::= Stmts .
```

`INIT(RegionVariable)` initialisiert die Variable **RegionVariable**, die vom Typ `REGION` sein muß, indem sie ihr eine, in diesem Programmlauf eindeutige, **Identifikationsnummer** *lock_id* zuweist. Alle **LockedStmts** Anweisungen eines Programms, für die die `RegionVariable` den gleichen Wert hat, formen eine **kritische Region**. Die Anweisungen einer kritischen Region können zu einem Zeitpunkt immer nur von einem Prozeß ausgeführt werden.
Will ein Prozeß p eine `LOCK` Anweisung ausführen und ein anderer Prozeß p' befindet sich in der kritischen Region, wird p suspendiert, bis p' die kritische Region verlassen hat.
Es darf, mittels einer `GOTO` Anweisung, weder in eine kritische Region hineingesprungen werden, noch darf aus ihr herausgesprungen werden. □

3.1.5 Semantikerhaltende Transformationen auf *pPL*

Innerhalb eines konkreten Übersetzters wird die konkrete Syntax eines Quellprogramms häufig vereinfacht dargestellt. Diese Transformationen müssen natürlich semantikerhaltend sein.

Die wichtigste ist die sogenannte Umwandlung von Ausdrücken in die sogenannte *Drei-Addreß-Form*. Wenn im folgenden von der Programmiersprache *pPL* bzw. einem Programm in dieser Sprache die Rede ist, dann wird davon ausgegangen, daß Ausdrücke in der Drei-Adreß-From vorliegen[5].

3.12 Transformation (Drei-Adress-Form)
Zuweisungen und Ausdrücke werden in die **Drei-Adreß-Form** überführt (siehe z.B. [Waite und Goos, 1984; Aho *et al.*, 1986]). Der Übersetzer muß dazu **Hilfsvariablen** einführen. Diese Hilfsvariablen sind prozeßlokal, d.h. andere Prozesse können nicht auf sie zugreifen (vergleichbar mit den Registern eines Prozessors[6]).

Eine **Drei-Adreßanweisung** hat i.A. die Form `var := op` oder `var := op`$_1$ $\otimes$ `op`$_2$. Dabei steht:

`var` für eine im Quellprogramm deklarierte Variable (kurz: **Programmvariable** oder Hilfsvariable,

`op` für eine Konstante, eine Programmvariable oder eine Hilfsvariable.

`op`$_1$,`op`$_2$, nur für Konstanten oder Hilfsvariablen.

Bevor ein Wert einer Programmvariable benutzt werden kann, muß er an eine Hilfsvariable zugewiesen werden: „die Variable muß in ein Register geladen werden". Dazu wird jedesmal eine

[5]Soll bereits das Quellprogramm in Drei-Adreß-From vorliegen müßte man eine spezielle Variablenart „Hilfsvariable", mit den unten angegeben Eigenschaften in die Sprache einführen.

[6]Die Zuordnung dieser Hilfsvariablen auf reale Prozessorregister geschieht im Codegenerator bzw. Registerallokator des Compilers.

neue Hilfsvariable eingeführt. Da Hilfsvariablen nicht im von allen Prozessen gemeinsam benutzten Speicher liegen, werden als Lese- und Speicherzugriffe – im Sinn von Definition 3.19 – nur die Zugriffe auf Programmvariablen verstanden. Somit enthält jede Drei-Adreßanweisung höchstens einen Zugriff auf den gemeinsamen Speicher.

Die auf diese Weise möglicherweise entstehenden Inkonsistenzen zwischen Hauptspeicher und Prozessorregistern wären nur unter hohen Leistungseinbußen vermeidlich (s. Ausdrücke). Diese Inkonsistenzen stehen aber nicht im Widerspruch zur Sprachdefinition. □

3.13 Bemerkung
Da bei der Ausführung *einer* Anweisung dieser Dreiadreßform höchstens ein Zugriff auf den gemeinsamen Speicher erfolgt (die Hilfsvariablen sind prozeßlokal), ist die Zuweisung `var := op` bzw. `var :=` $op_1 \otimes op_2$ per Definition atomar. □

Zur Erleichterung der Darstellung kann es hilfreich sein, wenn ein Quellprogramm in einer normierten Form vorliegt. Wir benutzen dazu die folgenden Quellsprachtransformationen.

3.14 Transformation (`PAR`-Anweisungen mit mehr als zwei Rümpfen)
Eine `PAR` S_1 `| ... |` S_n `END` Anweisung ist zu einer geschachtelten `PAR`-Anweisung mit nur zwei „äußeren" Prozeßrümpfen äquivalent: `PAR` S_1 `| PAR` S_2 `| ... |` S_n `END END`. Diese Transformation wird nur für Beweise benutzt, wenn diese mittels Induktion über die Anzahl der Prozeßrümpfe geführt werden. □

3.15 Transformation (Replizierte Prozesse)
Die Herleitung unserer Ergebnisse wird durch die Annahme vereinfacht, daß für jeden Prozeßrumpf *immer* mindestens ein Prozeß erzeugt wird. Für replizierte Prozeßrümpfe gilt dies nicht unbedingt. Durch die folgende Quelltexttransformation können wir die Forderung erfüllen:

```
[var : LowerBound TO UpperBound] Stmts
```

wird durch

```
[var : LowerBound TO MAX(LowerBound,UpperBound)]
          IF LowerBound <= UpperBound THEN Stmts END
```

ersetzt. Der Prozeßrumpf als Ganzes wird somit mindestens einmal ausgeführt, die Anweisungen `Stmts` hingegen nur in Abhängigkeit vom Ergebnis des Tests. Im folgenden nehmen wir also o.B.d.A. `LowerBound` ≤ `UpperBound` an. □

3.16 Transformation (Prozedurrumpf als `PAR`-Anweisung)
Zur Darstellung der Analyse- und Transformationsalgorithmen ist es oft einfacher, den Prozedurrumpf (`ProcedureBody`)

```
BEGIN Stmts END
```

als eine `PAR`-Anweisung, bestehend aus nur einem Prozeßrumpf

```
BEGIN PAR Stmts END END
```

zu betrachten. □

3.1.6 Schlußfolgerungen

Die vorgestellte Beispielsprache *pPL* deckt die zwei der drei wesentlichen Konzepte des kontrollflußparallelen Programmierens ab: Spezifikation der Nebenläufigkeit und Kommunikation. Die Spracherweiterungen pPL_{sync} und pPL_{lock} decken das dritte wichtige Konzept, die explizite Synchronisation ab. Der Zugriff auf den gemeinsamen Speicher unterliegt keiner Einschränkung. Das ausgewählte Maschinenmodell (MIMD mit CREW-Speicherzugriff) entspricht gegenwärtig existierender Hardware.

Für den Rest dieser Arbeit steht der Begriff „paralleles Programm" immer für ein *pPL* Programm, ohne daß dabei die Übertragbarkeit unserer Resultate auf andere parallele Programmiersprachen prinzipiell eingeschränkt wird.

3.2 Optimierung paralleler Programme

In diesem Abschnitt wollen wir die Möglichkeiten und Grenzen der Optimierung durch den Übersetzer informell aufzeigen. Neben der Wirksamkeit der Optimierungen ist natürlich die Korrektheit des Optimierers von großem Interesse. Dazu erläutern wir, was wir unter Korrektheit verstehen wollen. Da Optimierungsprobleme i.d.R. NP-hart sind, müssen wir uns mit „Verbesserungen“, d.h. mit Näherungslösungen zufrieden geben. Dazu wird nicht die (über die Interleavings definierte) konkrete Bedeutung eines Programms betrachtet, sondern Abstraktionen von ihr, die auf dem Begriff der „beobachtbaren Zustände“ basieren. Einige typische Optimierungs- und Analyseprobleme werden vorgestellt.

3.2.1 Grenzen und Möglichkeiten der Optimierung

Für ein Problem gibt es immer mehrere Programme, die eine Lösung berechnen. Dabei ist wichtig, daß die Programme das gleiche Ein-/Ausgabe- und Terminierungsverhalten haben. Der Benutzer eines Programms ist natürlich daran interessiert, das „beste“ Programm zu benutzen. Kriterien können dabei die Laufzeit oder der Speicherverbrauch des Programms sein. Daneben gibt es natürlich noch Kriterien wie Wartbarkeit, Portabilität, etc. des Programms, diese sind aber für unsere Aufgabe nicht relevant. Leider ist das Problem, die Äquivalenz von Programmen festzustellen, nicht entscheidbar.

Ein optimierender Übersetzer beschreitet einen konstruktiven Weg: Er versucht ausgehend von einem bestimmten Programm, ein äquivalentes, aber verbessertes zu erzeugen. Das Problem, zu einem gegebenen Programm das *optimale* zu finden, ist leider wieder NP-vollständig bzw. NP-hart[7], folgerichtig sollte nur die Rede von einem *verbessernden* Übersetzer sein.

Trotz dieser negativen Aussagen ist es einem Übersetzer möglich, korrekte Verbesserungen an einem Programm durchzuführen. Die Verbesserungen basieren häufig darauf, Redundanzen – z.B. Berechnungen, die den gleichen Wert ergeben – eines Programms zu entfernen, Berechnungen zur Übersetzungszeit, anstatt zur Laufzeit auszuführen, komplizierte arithmetische Operationen durch einfachere zu ersetzen oder nie benutzte Programmanweisungen zu entfernen, etc. Es wird dabei nicht versucht, *alle* Möglichkeiten der Optimierung zu finden, sondern nur eine Teilmenge davon: solche, die sich mit vertretbarem Aufwand finden lassen.

3.2.2 Korrektheit der Optimierung

Ein durch Compileroptimierung transformiertes Programm soll das Gleiche berechnen, wie das Ausgangsprogramm. Daher muß eine Optimierung **konservativ** sein, d.h. für *alle* möglichen Programmabläufe soll das transformierte Programm bezüglich des Ausgangsprogramms, bei gleicher Eingabe die gleiche Ausgabe liefern.

Bei sequentiellen Programmen legt eine konkrete Eingabe eines Programms den folgenden Programmablauf, d.h. die Abfolge der ausgeführten Anweisungen, insbesondere die Abfolge der Lese- und Speicherbefehle und damit auch die Ausgabe fest. Daher ist einem sequentiellen Programm S eine Funktion $f_S : IN \rightarrow OUT$ zugeordnet. Jede Ausführung des Programms wird für die Eingabe $x \in IN$ immer den gleichen Wert $f_S(x) \in OUT$ berechnen.

Bei parallelen Programmen ist dies nicht so! Da in einem MIMD-Rechnersystem die Prozessoren parallel arbeiten, gibt es nicht *den* nächsten ausgeführten Befehl, es sind derer viele (pro Prozessor einer). Bezüglich einer globalen externen Uhr kann diese Menge nicht eindeutig bestimmt werden, da jeder Prozessor die Abarbeitung eines Befehls zu einem anderen

[7]Z.B.: Das Problem der optimalen Registerzuteilung kann auf das Graphfärbungsproblem abgebildet werden, welches NP-hart ist. Ob die k Register eines Prozessors ausreichen, die Register optimal zu belegen, ist NP-vollständig (vgl. das k-Färbungsproblem und Bestimmung der chromatischen Zahl).

Zeitpunkt (bezüglich der globalen Uhr) starten kann. Es bleibt gemäß unserem MIMD-Modell als beobachtbare Basis nur die Abfolge der Lese- und Schreibbefehle am gemeinsamen Speicher (s. Abbildung 3.1). Wird diese Abfolge betrachtet, so ist offensichtlich, daß bedingt durch Serialisierung der Speicherzugriffe (s. Abbildung 3.1) und Unabhängigkeit der Prozessoren die Ausführungsreihenfolge der Lese- und Speicherbefehle am Speicher nicht nur von der Eingabe, sondern auch vom „Zufall“[8] bestimmt wird. Daraus können bei gleicher Eingabe (erwünschterweise) verschiedene Ausgaben resultieren. Mit anderen Worten: Ein paralleles Programm P ist eine Funktion f_P, die eine Eingabe auf ein Menge von Ausgabewerten abbildet: $f_P : IN \to \{o_1, o_2, \dots\}$ mit $o_i \in OUT$. Zur Laufzeit l des Programms wird für die Eingabe x aus der Menge $f_P(x)$ ein Element ausgewählt und als Ergebnis dieses Programmlaufes betrachtet: $F_P : IN \times TRACE \to OUT$, wobei $TRACE$ die Reihenfolge der zur Laufzeit ausgeführten Speicherzugriffe kennzeichnet.

Im Gegensatz zu sequentiellen Programmen gibt es bei parallelen Programmen somit eine Unterscheidung zwischen dem, was ein Programm (bei gleicher Eingabe) zur Laufzeit berechnen *kann*, und dem, was in einem konkreten Programmlauf berechnet *wird*. Da diese Mehrdeutigkeit erwünscht sein kann, fordern wir von einem korrekten Optimierer:

3.17 FORDERUNG (KORREKTER OPTIMIERER)
Ein optimiertes paralleles Programm muß

- die gleichen Eingabewerte wie das Ausgangsprogramm akzeptieren und
- für jede Eingabe die gleiche Menge der Ausgabewerte berechnen *können*, wie das Ausgangsprogramm. □

Das folgende Beispiel möge dies verdeutlichen:

3.18 BEISPIEL
Das folgende sequentielle Programmfragment

```
b1 := a + 1 + 2 * c ; b2 := a + 1 + 2 * c ;
```

kann ohne weiteres zu

```
b1 := a + 1 + 2 * c ; b2 := b1 ;
```

verkürzt (optimiert) werden. (Dies läßt sich leicht mit gängigen Beweismethoden zeigen). Würden wir die gleiche Transformation im folgenden Programmfragment

```
a  := 1;
PAR
   b1  := a  + 1 + 2 * c ; b2  := a  + 1 + 2 * c ;
 |
   a  := 10;
END
```

durchführen, wären die berechneten Werte für `b1` und `b2` zwar nicht falsch, aber die Transformation würde verschiedene Ausgaben (nämlich diejenigen, bei denen sich die Werte von `b1` und `b2` unterscheiden) nicht zulassen. □

3.2.3 Abstrakte Interpretation

Es ist offensichtlich, daß wir zur Übersetzungszeit nicht alle möglichen Programmausführungen konkret, d.h. mit allen Details der Programmausführung untersuchen können. Ein erster Schritt wird sein, von der konkreten Semantik des Programms zu abstrahieren, ein zweiter

[8] Aus der makroskopischen Sicht des Programms, nicht aus der mikroskopischen der Hardware.

Schritt erlaubt es dann, die Betrachtung aller Programmausführungen durch das Lösen eines Gleichungsystems zu ersetzen (Abschnitt 3.4)[9].

Abstrakte Interpretation [Cousot und Cousot, 1977; Cousot, 1981] ist ein grundlegender Ansatz der Programmanalyse, welche den Zusammenhang zwischen konkreter Programmsemantik (**Standardsemantik**) einerseits und einer Abstraktion hiervon (**Nonstandardsemantik**) andererseits beleuchtet. Die Standardsemantik der Programmiersprache kann dabei operational oder denotational [Stoy, 1977] angegeben werden. Je mehr die Nonstandardsemantik von der Standardsemantik abstrahiert, desto unschärfer werden die abgeleiteten Aussagen, aber desto eher können diese Aussagen zur Übersetzungszeit effizient berechnet werden. In [Cousot und Cousot, 1979] werden dazu Methoden für den systematischen und, bezüglich der formalen Semantik korrekten Entwurf von Programmanalyserahmen bereitgestellt.

Einen Hinweis auf die Art der benötigen Nonstandardsemantik liefert uns die folgende Überlegung: Für die Optimierung eines Programms ist es oft wichtig zu wissen, welchen Wert eine Variable oder ein Ausdruck an einer Stelle im Programm hat. Da dieser i.allg. zur Übersetzungszeit nicht bestimmbar ist, begnügt man sich mit weniger starken Aussagen, wie z.B.: Welche Definition (Zuweisung an eine Variable) erreicht diesen Programmpunkt. Auf diesen „schwächeren“ Informationsgrundmengen wird eine Nonstandardsemantik der Programmiersprache definiert. Dennoch erlaubt sie wirkungsvolle Optimierungen. **Datenflußanalyse** ist ein anderer, gebräuchlicherer Name für diese Methode.

Als nächstes definieren wir mit der *Interleavingsemantik* ein Ausführungsmodell für die konkrete Semantik als auch für Nonstandardsemantiken.

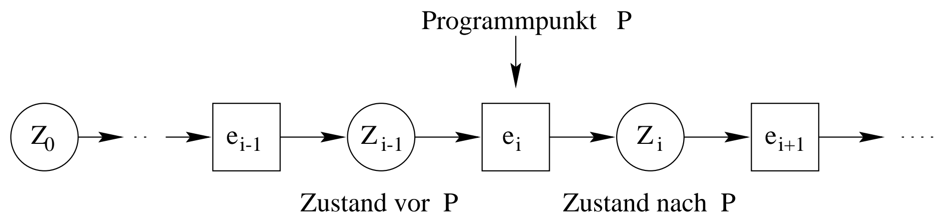
3.2.4 Interleavingsemantik

Ein durch Compileroptimierung transformiertes Programm soll das gleiche berechnen, wie das Ausgangsprogramm. Dazu müssen wir klären, was dabei der Begriff *Gleichheit* bedeutet und somit eine Semantik für die Ausführung paralleler Programme definieren. Die folgende Darstellung richtet sich auf unser Ziel: Analyse und Transformation kontrollflußparalleler Programme. Ein ausführliche Behandlung des Gebietes Semantik paralleler Programmierung findet sich z.B. in [Hoare, 1985; Best, 1995].

Ein (sequentielles) imperatives Programm wird wie folgt abgearbeitet: Ausgehend von einem Startzustand werden Zustände durch das Auftreten von Ereignissen solange ineinander überführt, bis das Programm eventuell terminiert, d.h. ein Endzustand erreicht ist (siehe Abbildung 3.2).

[9] Einen anderen, nicht so einfach auf parallele Programme übertragbaren Ansatz verfolgten [Wulf *et al.*, 1975]. Er basiert auf der Idee, daß es eine essentielle Reihenfolge (*essential ordering*) der Ausführung von Anweisungen innerhalb eines Programms gibt: Zwei Anweisungen s_1, s_2 sind essentiell angeordnet, wenn die Ausführung von $s_1; s_2$ und $s_2; s_1$ zu verschiedenen beobachtbaren Resultaten führt. Durch den Programmtext selbst ist allerdings eine viel strengere Ordnung – die initiale Ordnung (*initial order*) – vorgegeben, als für die eigentliche Berechnung nötig ist. Wulf *et al.* [1975] geben einen Algorithmus an, mit dem die essentielle Ordnungsrelation aus der initialen bestimmt werden kann. Damit definieren sie den Begriff des gemeinsamen Teilausdrucks und auf ihm den Begriff der Codeverschiebung formal nur über Aussagen dieser Ordnungsbeziehungen.

Eine ihrer zentralen Annahmen dabei ist, daß zwei textuell aufeinanderfolgende, lesende Speicherzugriffe auf die gleiche Variable immer den gleichen Wert liefern. Mit anderen Worten: aus der initialen Ordnung von Speicherzugriffen kann auf deren essentielle Nichtangeordnetheit geschlossen werden. In parallelen Programmen kann jedoch aus der initialen Ordnung dies *nicht* gefolgert werden: Die Variable kann in einem anderen Prozeß verändert werden. Zur Bestimmung, ob zwei Anweisungen essentiell geordnet sind, müssen also die Interleavings der Prozesse betrachtet werden. Somit kann die Methode von [Wulf *et al.*, 1975] leider nicht auf parallele Programme übertragen werden. Übertragbar ist jedoch die Bemerkung, daß die essentielle Ordnung – werden die Interleavings betrachtet – auch in einem optimierten parallelen Programm erhalten bleiben muß.



Z_0 ist der Startzustand, die e_j sind Ereignisse, (z.B. Speicherzugriffe durch den Prozessor) wie sie im Laufe des Programms auftreten. e_i und e_j können dabei von derselben Anweisung s im Programmtext ausgelöst werden, z.B. wenn s in einer Schleife ausgeführt wird.

Abbildung 3.2: Ablauf eines Programms als Zustandsüberführung

3.19 Definition (Ereignis, Zustand)
Ein **Ereignis** ist die Ausführung einer atomaren Aktion.
Als **Zustand** wird der gesamte Inhalt des Computerspeichers zu einem gegebenen Zeitpunkt (nach Abschluß der Ausführung eines Lese-/Speicherbefehls) verstanden. Anstatt des gesamten untypisierten Speicherinhaltes werden i.d.R. die typisierten Werte der Programmvariablen betrachtet, die das Programm bearbeitet. □

3.20 Bemerkung
Haben die Prozessoren je einen lokalen Zwischenspeicher (*cache*), so wird der Zeitpunkt „nach dem Abschluß des Lese- bzw. Schreibvorgangs" auf den gemeinsamen Speicher bezogen. Somit ist ein abgeschlossener Schreibvorgang einer, der von allen Prozessoren wahrgenommen werden kann. □

In einem sequentiellen Rechner ist der Folgezustand eines Zustandes zur Laufzeit eindeutig festgelegt, da die nächste auszuführende Anweisung eindeutig bestimmt ist. In einem MIMD-System ist der Folgezustand, bedingt durch die Serialisierung der Speicherzugriffe entsprechend der CREW/EREW-Regel, nicht immer eindeutig bestimmt. *Interleaving of threads*[10] ist eine einfache Möglichkeit, die Nebenläufigkeit von Prozessen zu studieren. Interleaving beruht auf der Annahme, daß parallele Programme durch die relative Ordnung von Ereignissen beschrieben werden können. Sind a und b Ereignisse, dann reicht es aus, die Ereignissequenzen „a vor b" und „b vor a" zu betrachten. Der Fall, daß a und b gleichzeitig ausgeführt werden, kann ignoriert werden.

3.21 Definition (Ereignisfolge)
Seien $e_1, e_2, \ldots, e_n$ Ereignisse, die während eines Programmlaufes in dieser Reihenfolge auftreten. Eine **Ereignisfolge** (engl.: *trace*) $p = \langle e_1; e_2; \ldots; e_n\rangle$ ist die geordnete Folge dieser Ereignisse. Die Anzahl $|p|$ der Elemente von p heißt **Länge** von p. $e \in p$ bedeutet, daß p das Ereignis e enthält. Seien $e', e'' \in p$, dann bedeutet $e' \ll_p e''$, daß e' vor e'' in p vorkommt. Ist p aus dem Kontext ersichtlich, wird dies zu $e' \ll e''$ verkürzt. Die **leere Ereignisfolge** wird durch $\langle\rangle$ dargestellt. Für eine nicht leere Ereignisfolge p definieren wir $\mathsf{head}(p) \stackrel{\text{def}}{=} e_1$ und $\mathsf{tail}(p) \stackrel{\text{def}}{=} \langle e_2; e_3; \ldots; e_n\rangle$. $p'; p'' \stackrel{\text{def}}{=} \langle e'_1; \ldots; e'_{n'}; e''_1; \ldots; e''_{n''}\rangle$ ist die **Konkatenation** der Ereignisfolgen $p' = \langle e'_1; \ldots; e'_{n'}\rangle$ und $p'' = \langle e''_1; \ldots; e''_{n''}\rangle$. Für Mengen P' und P'' von Ereignisfolgen definieren wir: $P'; P'' \stackrel{\text{def}}{=} \{p'; p'' | p' \in P' \text{ und } p'' \in P''\}$. Für $e \in p$ ist $p \downarrow e$ definiert durch $\{\langle e_1; e_2; \ldots; e_{i-1}\rangle | e_i = e\}$, anschaulich: die Menge der Ereignisfolgen p, bei denen alle Ereignisse aus p vor e auftreten. $p \downarrow e$ wird der **e-Anfang der Ereignisfolge** genannt. Ist P eine Menge von Ereignisfolgen, dann ist $P \downarrow e \stackrel{\text{def}}{=} \bigcup_{p \in P} p \downarrow e$. Seien p und q Ereignisfolgen, dann besteht die auf Anweisungen aus q **eingeschränkte Ereignisfolge** $p|_q$ nur aus Ereignissen

[10] Wörtlich übersetzt entspricht dies einer Metapher aus der Weberei: das Durchschießen von Fäden; *Interleaving* wird meist mit *Verzahnung* übersetzt, z.B. verzahnte Ausführung.

aus p, die auch in q vorkommen. Diese sind in der gleichen Reihenfolge wie in p angeordnet. $p|_{q',q''} \stackrel{\text{def}}{=} (p|_{q'})|_{q''}$. □

3.22 Definition (Interleaving)
Die erlaubte Aufeinanderfolge von Ereignissen heißt **Interleaving** oder sequentielle Ausführung eines Programms [Best, 1995]. Eine Ereignisfolge p ist ein Interleaving zweier Ereignissequenzen q' und q'', wenn die folgenden Bedingungen der Relation $p \bowtie (q', q'')$ erfüllt sind [Hoare, 1978]:

1. $\langle\rangle \bowtie (q', q'')$ genau dann, wenn $q' = \langle\rangle$ und $q'' = \langle\rangle$,
2. $p \bowtie (q', q'')$ genau dann, wenn $p \bowtie (q'', q')$.
3. $\langle e\rangle; p \bowtie (q', q'')$ genau dann, wenn ($q' \neq \langle\rangle$ und $\mathsf{head}(q') = e$ und $p \bowtie (\mathsf{tail}(q'), q'')$) oder ($q'' \neq \langle\rangle$ und $\mathsf{head}(q'') = e$ und $p \bowtie (q', \mathsf{tail}(q''))$). □

3.23 Lemma (†Anzahl der Interleavings[11])
Es gibt mindestens $2^{\min(|q'|,|q''|)}$ viele verschiedene Interleavings von q' und q''. □

Gilt $p \bowtie (q', q'')$, so haben die Ereignisse aus q' in p die gleiche Reihenfolge wie in q', ebenso die Ereignisse aus q''. Es ist also keine Ordnungsbeziehung in p für Ereignisse $e' \in q'$ und $e'' \in q''$ gegeben. Somit kann p auch als eine topologische Sortierung der Folgen q' und q'' betrachtet werden. Wir definieren:

3.24 Definition (TopSorts)
Seien $p' = \langle e'_1; e'_2; \ldots; e'_{n'}\rangle$ und $p'' = \langle e''_1; e''_2; \ldots; e''_{n''}\rangle$ Ereignisfolgen. $\mathsf{TopSorts}(p', p'')$ ist die Menge aller Ereignisfolgen, die durch **topologische Sortierung** von p' und p'' entstehen. Für zwei Ereignisse $e' \in p', e'' \in p''$ ist dabei keine Ordnungsbeziehung definiert, während für $e'_i, e'_j \in p'$ aus $i < j$ folgt $e'_i \ll_{p'} e'_j$ (analog für die Ereignisse aus p''). □

Offensichtlich gilt:

3.25 Lemma (Eigenschaften von TopSorts)
TopSorts *ist kommutativ und assoziativ.* □

3.26 Definition (TopSorts, Teil 2)
Seien p_i Ereignisfolgen, P, Q, P_i Mengen von Ereignisfolgen, dann läßt sich TopSorts erweitern zu:
$\mathsf{TopSorts}(P, Q) \stackrel{\text{def}}{=} \bigcup_{p \in P, q \in Q} \mathsf{TopSorts}(p, q)$.
$\mathsf{TopSorts}(P_1, \ldots, P_n) \stackrel{\text{def}}{=} \mathsf{TopSorts}(P_1, \mathsf{TopSorts}(P_2, \ldots, P_n))$.
Sei $k_i \geq 0$ eine natürliche Zahl. Als Argument von TopSorts bedeutet $k_i : p_i$, daß p_i als Argument k_i-mal auftritt. Ist $k_i = 0$, tritt p_i als Argument nicht in Erscheinung:

$\mathsf{TopSorts}(p_1, \ldots, k_i : p_i, \ldots, p_n) \stackrel{\text{def}}{=} \mathsf{TopSorts}(p_1, \ldots, \overbrace{p_i^1, \ldots, p_i^{k_i}}^{k_i\text{-mal}}, \ldots, p_n)$. Dabei haben die Ereignisse aus p_i eine Markierung l, so daß für $1 \leq l', l'' \leq k_i$ mit $l' \neq l''$ zwischen den Ereignissen $e_{j'}^{l'} \in p_i^{l'}, e_{j''}^{l''} \in p_i^{l''}$ keine Ordnungsbeziehung besteht.
Auf Mengen von Ereignisfolgen angewendet, ist $k_i : p_i$ analog definiert:

$\mathsf{TopSorts}(P_1, \ldots, k_i : P_i, \ldots, P_n) \stackrel{\text{def}}{=} \mathsf{TopSorts}(P_1, \ldots, \overbrace{P_i^1, \ldots, P_i^{k_i}}^{k_i\text{-mal}}, \ldots, P_n)$. □

3.27 Bemerkung
Wir werden „Ereignis“ und „Anweisung“ (genauer Ausführung einer Anweisung) synonym verwenden. Manche Ereignisse (Befehle) verändern dabei den Zustand (Speicher) nicht. □

[11] Sätze, Lemmata und Korollare, die mit † gekennzeichnet sind, werden in Anhang A bewiesen.

3.2.5 Beobachtbare Zustände

Die *beobachtbaren* Zustände spielen für die Definition des Gleichheitsbegriffes eines Optimierers eine zentrale Rolle. „Beobachtbar“ ist ein relativer Begriff: er ist abhängig von der Art der in Betracht stehenden Transformation, als auch von der Art der Programmbeobachtung durch den Programmbenutzer. Für die Korrektheit muß gezeigt werden, daß die Transformation eine Bedingung über die beobachtbaren Zustände invariant läßt. Zwei Beispiele mögen dies verdeutlichen:

3.28 Beispiel (Elimination gemeinsamer Teilausdrücke)
Ist von zwei Programmausdrücken festgestellt worden, daß sie *immer* den gleichen Wert berechnen, ist die zweite Berechnung redundant und kann eingespart werden. Ein Unterschied im Ein-/Ausgabeverhalten ist somit nicht festzustellen, wenn der Wert des Ausdrucks in einer (vom Übersetzer eingeführten) Hilfsvariablen gespeichert wird und später der Wert dieser Hilfsvariablen benutzt wird, statt den Ausdruck erneut auszuwerten. Der Programmierer kennt diese Hilfsvariable im Programm nicht und kann sie somit auch nicht beobachten, z.B. im Programm ausgeben. Wird das laufende Programm jedoch mit einem Überwachungswerkzeug (*Debugger*) untersucht, können sehr wohl Unterschiede festgestellt werden: Es existiert kein Maschinencode, der den zweiten Ausdruck berechnet, sondern nur solcher, der den Wert liefert, d.h. aus der Hilfsvariablen ausliest.

Es stellt sich nun die Frage, unter welchen Bedingungen zwei Ausdrücke den gleichen Wert berechnen. Sie tun es unter anderem, wenn:

- beide Ausdrücke syntaktisch gleich sind und
- alle in ihnen vorkommenden Variablen den gleichen Wert haben.

Die erste Bedingung ist leicht festzustellen. Die zweite Bedingung bedarf einigen Analyseaufwand. Zwei Benutzungen einer Variablen x liefern den gleichen Wert, wenn sie im Programmtext von nur einer Definition (d.h. Zuweisung an x) „erreicht werden“, und diese in beiden Fällen die gleiche ist.

Damit reicht es aus, die Menge der Definitionen einer Variablen als beobachtbaren Zustand für diese Transformation zu betrachten. Die konkreten Werte der Variablen interessieren hierfür nicht.

Zur Feststellung, welche Definitionen einer Variablen x, ein Auftreten von x erreichen, müssen wir, wie oben bereits geschildert, *alle* möglichen Programmausführungen betrachteten.

Damit ist die Korrektheit dieser Transformation nicht davon abhängig, ob das Programm ein sequentielles oder paralleles ist! □

3.29 Beispiel (Elimination toten Codes)
Kann gezeigt werden, daß bestimmte Anweisungen nie ausgeführt werden, können diese aus dem Programm entfernt werden. Der beobachtbare Zustand ist hier „eine Anweisung wird ausgeführt“ bzw. „eine Anweisung wird nicht ausgeführt“. Wieder müssen alle möglichen Programmabläufe betrachtet werden, wieder ist die Transformation unabhängig von der Art des Programms. □

Verallgemeinernd können wir somit festgestellen:

1. Jede Transformation hat ihren eigenen Begriff der Beobachtbarkeit, mit anderen Worten: arbeitet auf der ihr eigenen Sicht der Bedeutung des Programms. Basierend auf dieser Sicht wird die Korrektheit der Transformation definiert. Die Transformation ist korrekt, wenn die ihr eigene Invariante (über die beobachtbaren Zustände) bewiesen

werden kann. Kann für die Sicht gezeigt werden, daß sie mit der konkreten Bedeutung des Programms „übereinstimmt“, ist die Transformation auch bezüglich der konkreten Semantik des Programms korrekt.

2. Unabhängig von der Art des Programms, sequentiell oder parallel, müssen immer alle möglichen Programmabläufe betrachtet werden. Diese können durch die Eingabe determiniert sein (sequentielles Programm) oder die Menge aller Interleavings muß zusätzlich untersucht werden (paralleles Programm).
3. Da eine korrekte Transformation ein gegebenes Programm in ein zum Ausgangsprogramm äquivalentes überführt, kann die Korrektheit der ganzen Optimierungsphase auf die Betrachtung der Korrektheit einzelner Transformationen reduziert werden. Daß sich die lokal verbessernden Effekte einzelner Transformationen dabei global verstärken, als auch (leider) vermindern können, belegt die Praxis (und war, wie üblich bei NP-Problemen, zu erwarten).

3.2.6 Beispiele für Transformationen

Beispiele bekannter Analyseprobleme und Transformationen sind **Available-Expressions** (**AE**), **Busy-Expressions** (**BE**), **Reaching-Definitions** (**RD**), **Live-Variables** (**LV**) und **Constant-Computation** (**CC**). In Abbildung 3.3 werden die Problemstellungen und die intendierten Transformationen kurz skizziert. Wir benötigen dazu folgende Definition:

3.30 Definition (Menge der Teilausdrücke)
Die Menge der **Teilausdrücke** $\mathcal{TA}[e]$ eines Ausdrucks e ist definiert durch

$$\mathcal{TA}[e] \stackrel{\text{def}}{=} \begin{cases} \{x\} & \text{falls } e \equiv \boldsymbol{x} \text{ eine Programmvariable ist} \\ \{c\} & \text{falls } e \equiv \boldsymbol{c} \text{ eine Programmkonstante ist} \\ \{e\} \cup \mathcal{TA}[e_1] & \text{falls } e \equiv \otimes e_1 \\ \{e\} \cup \mathcal{TA}[e_1] \cup \mathcal{TA}[e_2] & \text{falls } e \equiv e_1 \otimes e_2 \end{cases}$$

Wobei $\otimes$ ein (unärer bzw. binärer) Quellsprachoperator ist. □

3.2.7 Schlußfolgerung

Ausgehend von der Untersuchung des Begriffes „korrekter Optimierer“ haben wir festgestellt, daß wir alle möglichen Programmabläufe bei der Programmanalyse betrachten müssen. Als Basis für parallele Programme bietet sich dafür die Interleavingsemantik an. Eine bedeutende Vereinfachung der Aufgabe besteht darin, sich für die intendierte Transformation eine spezielle Sicht der beobachtbaren Zustände zu definieren. Damit erreichen wir, daß die Korrektheit einer Transformation unabhängig von der Art des Programms (sequentiell oder parallel) ist.

3.3 Das Problem

Wie wir gesehen haben, müssen alle Programmabläufe untersucht werden, bevor eine optimierende Transformation durchgeführt werden darf. Dies bedeutet, alle Interleavings der Speicherzugriffsoperationen zu betrachten. Nehmen wir an, daß das Programm keine Schleifen enthält, sondern nur zwei Prozeßrümpfe. Der eine bestehe aus n' Anweisungen, die zu n'' (o.B.d.A. $n' \leq n''$) anderen Zuweisungen nebenläufig ausgeführt werden, so gibt es mindestens $O(2^{n'})$ Interleavings dieser Anweisungen! Damit wird jeder Versuch scheitern, alle Interleavings während der Programmanalyse zu betrachten. Die Herausforderung besteht somit darin, die Anzahl der zu untersuchenden Interleavings drastisch zu reduzieren.

	Beschreibung	Grundmenge	mögliche Optimierung
AE	*Available-Expressions*: Welche Ausdrücke erreichen einen Programmpunkt P?	$\mathcal{L}^{\text{exprs}}$: Menge aller Ausdrücke und deren Teilausdrücke, die im Programm vorkommen. Kommt im Quellprogramm z.B. der Ausdruck $e \equiv a + (b * c)$ vor, so zählen zu $\mathcal{L}^{\text{exprs}}$ der Ausdruck e und all seine Teilausdrücke: a , b , $b * c$ und $a + (b * c)$	Wird ein Ausdruck e in P und auf allen Pfaden, die P erreichen, berechnet, muß e in P nicht erneut berechnet werden; statt dessen kann der bereits berechnete Wert benutzt werden.
BE	*Busy-Expressions*: Welche Ausdrücke werden in Anweisungen, die P folgen, nochmals berechnet?		Weiß man, daß ein Ausdruck später nochmals berechnet wird, lohnt es sich, ihn in einer Hilfsvariable (Register) abzuspeichern, um diese dann zu benutzen.
RD	*Reaching-Definitions*: Welche Definitionen einer Variablen erreichen P?	$\mathcal{L}_{x}^{\text{defs}}$: Menge der Definitionen d_x : $x := expr$ an die Variable x .	Erreicht nur genau eine Definition d_x einer Variablen x den Programmpunkt P, kann man in P anstatt auf x zuzugreifen, den Wert $expr$ von x benutzen.
LV	*Live-Variables*: Wird der Wert einer Variablen in Anweisungen, die P folgen, benutzt, d.h. wird auf sie nochmals zugegriffen?	$\mathcal{L}^{\text{vars}}$: Menge der Variablen eines Programms.	Wird der Wert einer Variablen in Anweisungen, die P folgen, nicht benutzt, kann man sich die Zuweisung sparen.
CC	*Constant-Computation*: Hat eine Variable bzw. ein Ausdruck an P einen konstanten Wert, wenn ja, welchen?	$\mathcal{L}^{\text{cc}} \stackrel{\text{def}}{=} \{\bot, \top, c_1, c_2, \dots\}$; $\bot$: Variable hat keinen konstanten Wert; $\top$: es ist noch nichts bekannt; c_i Konstanten.	Hat eine Variable bzw. Ausdruck an P einen konstanten Wert, kann dieser benutzt werden.

Abbildung 3.3: Einige typische Analyse- und Transformationsprobleme

Als Lösung schlagen CHOW und HARRISON [1994] vor, die Programmiersprache einzuschränken: „*We also notice that the task of compiler analysis can be much simplified for programming languages that restrict the access of shared variables among concurrent threads – for example, read-only or one-writer-multiple-readers – because the number of non-redundant interleavings can be reduced significantly.*“ MIDKIFF und PADUA [1990] lehnen dies aber entschieden ab: „*Designers of parallel languages have either ignored the problem of optimizing programs written in those languages or have overcome the problem by prohibiting the sharing of data among threads of the program executing in parallel. ... As the examples of this paper demonstrate, the problem*[12] *can only be ignored if parallel programs are not optimized – a course of action that may incur a large performance penalty. Attempts to legislate the problem away by placing restrictions on data sharing lead to what we feel is an excessively restrictive programming model.*“

CHOW und HARRISON [1992a] benutzen Abstrakte Interpretation zur Analyse paralleler Programme mit gemeinsamem Speicher mit dem Ziel, Optimierungen zu ermöglichen. In [CHOW und HARRISON, 1992b, 1994] versuchen sie, das Problem der durch die Betrachtung aller Interleavings entstehenden Zustandsexplosion (*state explosion*) mittels der *stubborn set theory* [VALMARI, 1989, 1990] zu lösen. Obwohl die abstrakte Interpretation einen systematischen Ansatz zur Reduktion des Zustandsraumes durch das Zusammenfassen „miteinander verbundener“ Zustände bietet, sind sie nicht in der Lage, die Zahl signifikant für *jedes* Programm zu reduzieren. In [CHOW und HARRISON, 1994] stellen sie fest, daß „*However, higher degree of abstraction is often needed to obtain a reasonable state space, which results in lower analysis precision*“.

Basierend auf den folgenden Definitionen und Sätzen für die sequentielle Programmanalyse werden wir in Kapitel 4 die Lösung des Probleme der Zustandsexplosion vorstellen.

3.4 Grundlagen der Analyse sequentieller Programme

Dieser Abschnitt legt die Grundlagen der Programmanalyse sequentieller Programme, soweit sie für unsere Arbeit benötigt werden. Die in Abbildung 3.3 beschriebenen Transformationen werden vervollständigt.

Obwohl nirgendwo in der Literatur explizit vermerkt, sind die nachfolgend definierten Begriffe, Sätze und Algorithmen nur im Kontext sequentieller Programme definiert. Die Definitionen der Kontrollflußanalyse, Halbverbände und Datenflußanalyse folgen im wesentlichen [HECHT, 1977; KAM und ULLMAN, 1977; AHO *et al.*, 1986].

3.4.1 Ziele der Analyse

MUCHNICK und JONES [1981] definieren die Aufgaben und Ziele der Programmanalyse wie folgt:

> *Flow analysis is a tool for discovering properties of the run-time behavior of a program without actually running it. The properties discovered usually apply to all possible sequences of control and data flow, and so give global information impossible to obtain by individual runs or by inspection of only a part of the program. Frequently flow analysis can be viewed as executing the program in parallel over a symbolic, much simplyfied version of its real data domain and hence is known alternately by the names* symbolic execution *or* abstract interpretation. *The information obtained from flow analysis can be used to drive the optimization phase*

[12] Die in dem Papier geschilderten Fehler.

> *of a compiler, to generate invariant assertions of a program for verification, to improve software reliability by aiding the generation of program documentation and automation of a large part of the debugging process, and to direct the actions of a content- and goal-oriented development system.*

Grundlegend sind die Kontrollflußanalyse[13] und die Datenflußanalyse. Erstere dient dazu, Aussagen darüber zu treffen, in welcher Reihenfolge Anweisungen ausgeführt werden. Letztere erlaubt es, Aussagen über die Abhängigkeiten verschiedener Berechnungen zu finden. In einem optimierenden Übersetzer ist die richtige Wahl der Datenstrukturen eminent wichtig, da sie das Laufzeitverhalten von Analysen und Transformationen entscheidend beeinflussen. Statt den Kontroll- und den Datenfluß getrennt zu betrachten, bietet es sich an, diese Informationen in einer einzigen Datenstruktur zusammenzufassen. Eine mögliche Darstellung ist die *Static Single Assignment Form* [Alpern *et al.*, 1987; Rosen *et al.*, 1987; Cytron *et al.*, 1989].

3.4.2 Kontrollflußanalyse

In einem Übersetzer ist ein Quellprogramm meist als Kontrollflußgraph dargestellt. Anweisungen, die *immer* direkt hintereinander ausgeführt werden, faßt man zu Basisblöcken (Grundblöcken) zusammen, die die Knoten des Graphen bilden. Kanten entsprechen den Programmsprüngen, wie sie durch Bedingungen, Schleifen, etc. entstehen. Verfahren, wie der Kontrollflußgraph aufgebaut wird, finden sich z.B. in [Waite und Goos, 1984].

3.31 Definition (Basisblock)
Ein **Basisblock** ist eine nicht leere, geordnete Folge von Anweisungen mit der folgenden Eigenschaft: Entweder werden alle Anweisungen eines Basisblocks (beginnend mit der ersten, in der gegebenen Reihenfolge) ausgeführt, oder keine Anweisung des Basisblocks wird ausgeführt.
Die letzte Anweisung eines Basisblocks muß eine Sprunganweisung auf einen oder mehrere Basisblöcke sein. Eine Ausnahme bildet der Basisblock, der die `EndProcedure` Zwischensprachanweisung, die das textuelle Ende einer Prozedur kennzeichnet, enthält. Dieser Basisblock enthält nur die `EndProcedure` Anweisung und hat keine Nachfolger. Die Quellsprachanweisung `RETURN` wird als Sprung auf den `EndProcedure`-Basisblock betrachtet. □

3.32 Definition (Kontrollflussgraph)
Ein **Kontrollflußgraph** (**CFG**) ist ein Tripel $G \stackrel{\text{def}}{=} (N, E, n_0)$, wobei N die endliche Menge der **Knoten** (Basisblöcke) und $E \subseteq N \times N$ eine Menge gerichteter **Kanten** zwischen den Knoten ist. n_0 ist der eindeutige **Startknoten**. Für einen Knoten n ist die Menge der **Vorgänger** ($\mathsf{pred}(n)$) bzw. **Nachfolger** ($\mathsf{succ}(n)$) wie folgt definiert: $\mathsf{pred}(n) \stackrel{\text{def}}{=} \{n' \mid (n', n) \in E\}$, bzw. $\mathsf{succ}(n) \stackrel{\text{def}}{=} \{n' \mid (n, n') \in E\}$.
Ein **Pfad** $p \stackrel{\text{def}}{=} \langle n_1, \ldots, n_k \rangle$ von n_1 bis n_k ist eine Sequenz von Knoten $n_1, n_2, \ldots, n_k$, so daß für alle $1 \leq i < k$ gilt: $(n_i, n_{i+1}) \in E$. Dieser Pfad hat die **Länge** k. Der **leere Pfad** wird mit $\langle\rangle$. bezeichnet. Alle Knoten n eines Kontrollflußgraphen müssen vom Startknoten n_0 **erreichbar** sein, d.h. es gibt einen Pfad von n_0 nach n. $\mathsf{path}[n]$ ist die Menge aller Pfade von n_0 nach n inklusive n; $\mathsf{path}[n)$ ist die Menge aller Pfade von n_0 zu Vorgängern von n.
Zwei nicht leere Pfade $p' = \langle n'_1, \ldots n'_I \rangle$ und $p'' = \langle n''_1, \ldots n''_J \rangle$ **konvergieren** im Knoten n, wenn

$$n'_1 \neq n''_1$$
$$n'_I = n = n''_J$$

[13] „Kontrollfluß“ ist die – leider nicht ganz zutreffende, aber gebräuchliche – Übersetzung des englischen *control flow*, richtiger wäre „Ablauffluß“ oder „Steuerungsfluß“.

$$(n'_i = n''_j) \implies (i = I \text{ oder } j = J)$$

Anschaulich starten beide Pfade in verschiedenen Knoten und kommen am Ende zusammen. Ein Pfad kann dabei durchaus einen Zyklus haben, der in n beginnt und endet. Der Knoten n heißt **Konvergenzpunkt** der beiden Pfade.
Ein **single exit Kontrollflußgraph** ist ein Quadrupel $G \stackrel{\text{def}}{=} (N, E, n_0, n_{\text{exit}})$ mit dem Ausgangsknoten $n_{\text{exit}} \in N$. Von jedem Knoten aus N gibt es einen Pfad zu n_{exit}. Im folgenden gehen wir immer von einem *single exit* Kontrollflußgraph aus.
Ist G ein *single exit* Kontrollflußgraph, dann ist $G^{-1} \stackrel{\text{def}}{=} (N, \{(n', n'')|(n'', n') \in E\}, n_{\text{exit}}, n_0)$ der **umgekehrte Kontrollflußgraph** zu G. G^{-1} ist ebenfalls ein *single exit* Kontrollflußgraph. □

Dominatoren [Tarjan, 1974] sind Basisblöcke mit speziellen Kontrollflußeigenschaften, die dazu benutzt werden können, Schleifen zu erkennen, das Berechnen der DFA-Information zu beschleunigen oder die *Static Single Assignment Form* eines Programms zu berechnen (siehe Abschnitt 5.7).

3.33 Definition (Dominator)
Ein Knoten n eines CFG's **dominiert** einen Knoten m ($n \,\mathsf{dom}\, m$), wenn jeder Pfad vom Eingangsknoten n_0 zu m durch n geht. Die **Dominanzrelation** dom ist reflexiv und transitiv. Wenn n den Knoten m dominiert und $n \neq m$, wird m strikt von n dominiert (n sdom m).
Ein Knoten m **dominiert** einen Knoten n **direkt** ($m = \mathsf{idom}(n)$), wenn gilt:

$$m \ \mathsf{sdom}\ n \text{ und } \forall m' : m' \ \mathsf{sdom}\ n \text{ mit } m' \neq m \implies m' \ \mathsf{sdom}\ m.$$

Da aus n sdom m und m sdom folgt, daß $n = m$, kann die **Dominanzrelation** dom als **Dominatorbaum** dargestellt werden. Die Wurzel dieses Baumes ist der Eingangsknoten n_0.
Eine Kante $q \to z$ eines Kontrollflußgraphen ist eine **Rückwärtskante**, wenn der Zielknoten z den Quellknoten q dominiert ($z \,\mathsf{dom}\, q$). Alle anderen Kanten sind **Vorwärtskanten**.
Die **Dominanzgrenze** [Cytron *et al.*, 1991] (*dominance frontier*, $\mathsf{DF}(n)$) eines Knotens n ist die Menge aller CFG Knoten m, so daß n zwar einen Vorgänger von m dominiert, jedoch m nicht strikt von n dominiert wird:

$$\mathsf{DF}(n) \stackrel{\text{def}}{=} \{m | \exists m' \in \mathsf{pred}(m) : n \,\mathsf{dom}\, m' \text{ und } n \ \not{\mathsf{sdom}}\ m\}$$ □

Ein einfacher, die Dominanzrelation berechnender Algorithmus kann z.B. in [Aho *et al.*, 1986] gefunden werden[14]. Cytron *et al.* [1991] geben einen Algorithmus an, der die Dominanzgrenze mittels zweier einfacherer Mengen bestimmt.

Zur Berechnung der Datenflußinformation wird manchmal die **Intervallanalyse** [Cocke, 1970] benutzt. Intervalle sind Teilgraphen eines CFG's, die die Eigenschaft haben, daß alle Knoten des Intervalls nur über Pfade, die durch einen Intervallkopf gehen, von Knoten außerhalb des Intervalls erreichbar sind. Intervalle sind also mit Schleifen vergleichbar. Die folgenden Definitionen stammen aus [Ullman, 1975].

3.34 Definition (Intervall, Reduzibilität)
Sei G ein Kontrollflußgraph und h ein Knoten aus G. Das **Intervall** $I(h)$ mit dem Kopf h ist die eindeutig bestimmte Teilmenge der Knoten von G mit folgenden Eigenschaften:

1. $h \in I(h)$

[14] Er benötigt $O(|N|)$ Durchläufe und hat den Aufwand $O(|E||N|)$. Für typische Programme mit reduziblen Kontrollflußgraphen werden aber nur zwei Durchläufe benötigt (zitiert nach [Lengauer und Tarjan, 1979]). Lengauer und Tarjan geben einen Algorithmus an, der die Dominanzrelation mit Aufwand $O(|E| \log |N|)$ berechnet. Benutzen sie eine etwas komplexere Implementierung, sinkt der Aufwand auf $O(|E|\alpha(|E|, |N|))$ (α ist das funktionale Inverse der Ackermannfunktion). Harel [1985] beschreibt einen etwas komplexeren Algorithmus, der nur den Aufwand $O(|E|)$ hat.

2. Ist n ein Knoten ($n \neq n_0$) und sind alle Vorgänger von n in $I(h)$ enthalten, dann ist auch n in $I(h)$ enthalten.

Die Reihenfolge, in der ein die Intervalle berechnender Algorithmus die Knoten in ein Intervall einfügt, wird **Intervallordnung** genannt.
Ein Kontrollflußgraph kann durch folgende Konstruktion eindeutig in Intervalle zerlegt werden:

1. Bilde $I(n_0)$
2. Ist ein Knoten n noch nicht in einem Intervall enthalten, aber einer oder mehrere Vorgänger von n sind bereits in Intervallen enthalten, dann berechne $I(n)$.

Der **abgeleitete Graph** $I(G)$ ist definiert durch:

1. Die Knoten von $I(G)$ sind die Intervalle von G.
2. $I(G)$ enthält eine Kante (J, K), wenn es in G eine Kante von einem Knoten aus dem J entsprechenden Intervall zum Kopf des K entsprechenden Intervalls gibt.
3. Der initiale Knoten von $I(G)$ ist $I(n_0)$.

Die Folge von verschiedenen **Intervallgraphen**: $G, I(G), I(I(G)), \ldots$ wird die **abgeleitete Intervallfolge** genannt. Sie ist endlich und es gibt einen Graphen G^*, den **Grenzkontrollflußgraph** mit $G^* = I(G^*)$.
Ein Kontrollflußgraph G heißt **reduzibel**, wenn der Grenzkontrollflußgraph von G aus nur genau einem Knoten besteht und keine Kanten hat. Ansonsten heißt G **irreduzibel**.
Die **Tiefe** d eines reduziblen CFG's G ist die minimale Anzahl der nötigen Intervallbildungen bis G^* erreicht ist. □

3.35 Bemerkung
Programme, die keine explizite Sprunganweisung (`GOTO`) enthalten, bzw. nur „strukturierte" Sprünge (`RETURN` oder `EXIT`), sind reduzibel. Die Untersuchungen von Knuth [1971] belegen, daß die meisten in der Praxis vorkommenden *Fortran* Programme (die explizite Sprünge enthalten können) reduzibel sind. Die durchschnittliche Tiefe reduzibler Kontrollflußgraphen liegt bei 2.75 [Knuth, 1971].

Nichtreduzible Kontrollflußgraphen können durch eine *node splitting* genannte Technik in reduzible überführt werden, s. z.B. [Hecht, 1977]. □

3.4.3 Datenflußanalyse (DFA)

3.4.3.1 Einleitung

Die **Datenflußanalyse** (**DFA**), als eine Form der Nonstandardsemantik eines Programms, liefert offensichtlich im Vergleich zur konkreten Programmsemantik nur Abschätzungen über ein Programm, die aber für die Optimierung eines Programms ausreichend sind.

Die Datenflußanalyse eines Programms basiert auf drei Grundlagen: erstens einer Abstraktion der Programmzustände, **Datenflußinformation** genannt; zweitens der Festlegung, wie diese Informationen durch die Programmanweisungen bestimmt und verändert werden; und drittens, wie sie durch Folgen von Anweisungen verändert werden. Typische Abstraktionen der Zustände sind in Abbildung 3.3 auf Seite 23 angegeben.

Erste DFA-Ansätze sind bereits 1961 in einem FORTRAN Compiler zu finden [Hecht, 1977, S. 26]:

In 1961 Vyssotsky ... implemented a compile-time diagnosis facility using (intraprocedural) control flow analysis and data flow analysis as a postprocessor of a Bell Laboratories 7090 FORTRAN II *compiler. After constructing a control flow graph (for each procedure) of the source program, Vyssotsky (evidently) solved the „reaching definitions“ problem, using bit vectors to represent sets. By combining this global (intraprocedural) information about definitions reaching each block with the local (intrablock) exposed uses, potential errors were infered, thus generating appropriate messages. ... „Worst case“ assumptions were made for subprogram invocations.*

Cocke [1970] hat das Problem der Elimination gemeinsamer Teilausdrücke auf ein lineares Gleichungssystem mit booleschen Unbekannten abgebildet und ein Verfahren (die Intervallanalyse) angegeben, wie dieses effizient gelöst werden kann. Da es keine eindeutige Lösung gibt, wird der maximale (oder minimale) Fixpunkt bestimmt.

Kildall hat in seiner Dissertation aus dem Jahre 1972 die Grundlagen für die auf der Theorie der Halbverbände basierende Datenflußanalyse gelegt, indem er bereits existierende Ansätze wissenschaftlich aufarbeitete und verallgemeinerte; bekannter ist der Artikel [Kildall, 1973]. Dabei werden die DFA-Informationen durch Werte eines Halbverbandes modelliert. Z.B. repräsentiert der boolesche Halbverband die Information *Eigenschaft* ***X*** *ist an diesem Programmpunkt P gültig, bzw. ungültig.* Die durch die Ausführung einer Zuweisung s bedingte Zustandsüberführung wird als eine Funktion f_s über diesen Halbverbandswerten angegeben. Werden zwei Anweisungen s_1 und s_2 unmittelbar hintereinander ausgeführt $(s_1; s_2)$, ist die DFA-Information durch die Funktionskomposition der entsprechenden Funktionen gegeben: $f_{s_2} \circ f_{s_1}$.

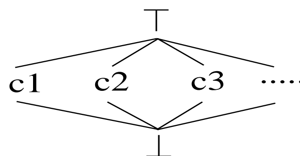
Eine Ausführung des Programms bis zu diesem Punkt P bildet einen **Ausführungspfad**, bei dem die Anweisungen sequentiell hintereinander ausgeführt werden. Damit die DFA konservativ ist, müssen alle möglichen Programmpfade vom Programmstart bis zu P betrachtet werden. Leider kann es exponentiell (in der Anzahl der Verzweigungen) oder sogar unendlich viele solcher Pfade geben, so daß es auf den ersten Blick unmöglich erscheint, sie alle zu betrachten. Kildall hat aber in seiner Arbeit dieser Hydra ihre Köpfe geraubt und einen effizienten iterativen DFA-Algorithmus entwickelt.

3.4.3.2 Halbverbände

3.36 Definition (Halbordnung)

Eine **partielle Ordnung** (auch **Halbordnung**) auf einer Menge $\mathcal{L}$ ist eine reflexive, antisymmetrische und transitive Relation $\sqsubseteq$. Das Paar $(\mathcal{L}, \sqsubseteq)$ wird **halb geordnete Menge** genannt. Für $x \sqsubseteq y$ und $x \neq y$ schreibt man auch $x \sqsubset y$. Statt $x \sqsubseteq y$ kann man auch $y \sqsupseteq x$ schreiben, analog statt $x \sqsubset y$ auch $y \sqsupset x$.

Sei $\mathcal{L}^{\preceq} = \{\top, \bot, c_1, c_2, \dots\}$. Die im folgenden graphisch dargestellte Halbordnung $\preceq$ heißt **flache Halbordnung**. Das größte Element ist $\top$ (**top**), das kleinste $\bot$ (**bottom**). Zwischen den anderen Elementen $c_1, c_2, \dots$ gibt es keine Ordnungsbeziehung:



Eine Menge von Elementen $x_1, x_2, \dots, x_n$ aus $\mathcal{L}$ wird **absteigende Kette** (**lineare Ordnung**) genannt, wenn $\forall 1 \leq i < n : x_i \sqsupset x_{i+1}$. Die **Länge** dieser Kette ist n. Eine halbgeord-

nete Menge heißt **beschränkt**[15], wenn es für alle $x \in \mathcal{L}$ eine Konstante b_x gibt, so daß jede absteigende Kette, die mit x beginnt, höchstens die Länge b_x hat.
Wir setzen im folgenden voraus, daß die Menge $\mathcal{L}$ mindestens zwei Elemente hat. □

3.37 Definition (Halbverband)
Ein **Halbverband** $(\mathcal{L}, \sqsubseteq, \sqcap)$ ist eine halb geordnete Menge $(\mathcal{L}, \sqsubseteq)$ mit einer binären Operation $\sqcap$ (**meet**[16]), so daß für $a, b, c \in \mathcal{L}$ folgendes gilt:

$$\begin{array}{rcll} a \sqcap a & = & a & \text{Idempotenz} \\ a \sqcap b & = & b \sqcap a & \text{Kommutativität} \\ a \sqcap (b \sqcap c) & = & (a \sqcap b) \sqcap c & \text{Assoziativität.} \end{array}$$

Für zwei Elemente $a, b \in \mathcal{L}$ gilt:

$$\begin{array}{rcl} a \sqsubseteq b & \Longleftrightarrow & a \sqcap b = a \\ a \sqsubset b & \Longleftrightarrow & a \sqcap b = a \text{ und } a \neq b. \end{array}$$

$(\mathcal{L}, \sqsubseteq, \sqcap)$ hat ein **Nullelement** $\bot$ (**bottom**), wenn $\forall x \in \mathcal{L} : x \sqcap \bot = \bot$, und ein **Einselement** $\top$ (**top**), wenn $\forall x \in \mathcal{L} : x \sqcap \top = x$. Von nun an nehmen wir an, daß $(\mathcal{L}, \sqsubseteq, \sqcap)$ ein Nullelement und ein Einselement hat.
Die $\sqcap$ Operation kann auf mehrere Elemente erweitert werden:

$$\bigsqcap_{i=1}^{n} x_i = x_1 \sqcap x_2 \sqcap \ldots \sqcap x_n \text{ mit } \bigsqcap_{x \in \emptyset} = \top$$

Ist $(\mathcal{L}, \sqsubseteq)$ beschränkt, kann für jede abzählbar unendliche Menge $S \subseteq \mathcal{L}$ der *meet* über alle Elemente $\bigsqcap_{x \in S} x$ definiert werden als $\bigsqcap_{x \in S} x = \lim_{n \to \infty} \bigsqcap_{i=1}^{n} x_i$. Da $(\mathcal{L}, \sqsubseteq)$ beschränkt ist, gibt es eine natürliche Zahl m mit: $\bigsqcap_{x \in S} x = \bigsqcap_{i=1}^{m} x_i$.
Basiert die *meet*-Operation auf der flachen Halbordnung, so notieren wir dies mit $\sqcap^{\preceq}$ □

3.38 Satz (Dualitätsprinzip)
Ist $\sqsubseteq$ eine Halbordnung, dann ist auch die inverse Relation $\sqsubseteq^{-1}$ eine Halbordnung. $\sqsubseteq^{-1}$ heißt die zu $\sqsubseteq$ duale Halbordnung. Ersetzt man alle Begriffe in einer wahren Aussage über alle Halbordnungen durch ihre dualen Entsprechungen, erhält man eine wahre Aussage über alle dazu dualen Halbordnungen[17].

Die dualen Entsprechungen sind:

Begriff	$\sqsubseteq$	$\sqsubset$	$\top$	$\bot$	$\sqcap$	*größter Fixpunkt*
dualer Begriff	$\sqsupseteq$	$\sqsupset$	$\bot$	$\top$	$\sqcup$	*kleinster Fixpunkt*

□

Als nächstes definieren wir die Transferfunktion, welche beschreibt, wie die Ausführung einer einzelnen Anweisung s die Datenflußinformation verändert, die vor ihrer Ausführung gültig war.

3.39 Definition (Funktionenraum)
Sei $(\mathcal{L}, \sqsubseteq, \sqcap)$ ein beschränkter Halbverband. Eine Menge $\mathcal{F}$ von einstelligen Funktionen (**Transferfunktionen**) über $\mathcal{L}$ heißt mit $\mathcal{L}$ assoziierter **monotoner Funktionenraum**, wenn die folgenden Bedingungen [M1] – [M4] erfüllt sind. Gilt zusätzlich noch [M5], heißt $\mathcal{F}$ mit $\mathcal{L}$ assoziierter **distributiver Funktionenraum**.

[15] Im englischen *of finite length* oder *bounded* oder *well-founded*, siehe [Hecht, 1977].
[16] Im Deutschen ist statt des englischen *meet* der Begriff *Minimumsoperator* üblich. Da wir $\sqcap$ sowohl für das Minimum als auch für das Maximum benutzen wollen, ist das englische *meet* neutraler.
[17] $\sqcup$ wird der **join** Operator genannt.

[M1] Alle Funktionen $f \in \mathcal{F}$ sind **monoton**: $\forall x, y \in \mathcal{L} : f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$. Dies ist äquivalent zu: $\forall x, y \in \mathcal{L} : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

[M2] Es gibt eine **Identitätsfunktion** $\mathsf{id} \in \mathcal{F}$ mit: $\forall x \in \mathcal{L} : \mathsf{id}(x) = x$.

[M3] $\mathcal{F}$ ist abgeschlossen bzgl. der Funktionskomposition: $\forall f, g \in \mathcal{F} : f \circ g \in \mathcal{F}$.

[M4] $\mathcal{L}$ ist der $\{\bot\}$-Abschluß bzgl. der $\sqcap$-Operation und Anwendung von Funktionen aus $\mathcal{F}$. (In [Kam und Ullman, 1976]: Für alle $x \in \mathcal{L}$ gibt es eine endliche Teilmenge $H \subseteq \mathcal{F}$ mit $x = \prod_{f \in H} f(\bot)$.)

[M5] Alle Funktionen $f \in \mathcal{F}$ sind **distributiv**: $\forall x, y \in \mathcal{L} : f(x \sqcap y) = f(x) \sqcap f(y)$. □

3.40 Definition (Fixpunkt)
Ein **Fixpunkt** einer Funktion $f \in \mathcal{F}$ ist ein $x \in \mathcal{L}$ mit $f(x) = x$. □

3.41 Definition
Sei[18] $f \in \mathcal{F}$ für $i \in \mathbb{N}_0$ ist $f^i(x)$ wie folgt definiert: $f^{i+1}(x) \stackrel{\text{def}}{=} f(f^i(x))$ und $f^0(x) \stackrel{\text{def}}{=} x$. □

Auf den Fixpunktsatz von Knaster-Tarski geht folgender, für die DFA grundlegende Satz zurück (zitiert nach [Hecht, 1977]):

3.42 Satz (Fixpunktsatz für Halbverbände)
Sei $f : \mathcal{L} \to \mathcal{L}$ eine monotone Funktion über einem beschränkten Halbverband $(\mathcal{L}, \sqsubseteq, \sqcap)$ mit Nullelement $\bot$. Der ***kleinste Fixpunkt*** *von f ist $f^k(\bot)$, wobei $k \in \mathbb{N}$ die kleinste Zahl ist, mit $f^k(\bot) = f(f^k(\bot))$* □

3.4.3.3 DFA-Rahmen

3.43 Definition (Datenflussanalyserahmen)
Ein **monotoner Datenflußanalyserahmen** ist ein Quadrupel $\mathcal{D} \stackrel{\text{def}}{=} (\mathcal{L}, \sqsubseteq, \sqcap, \mathcal{F})$ wobei $(\mathcal{L}, \sqsubseteq, \sqcap)$ ein beschränkter Halbverband und $\mathcal{F}$ ein mit $\mathcal{L}$ assoziierter monotoner Funktionenraum ist. Eine **Instanz** eines monotonen DFA-Rahmens ist ein Paar $\mathcal{I} \stackrel{\text{def}}{=} (G, M)$, wobei G ein Kontrollflußgraph, und $M : N \to \mathcal{F}$ eine Abbildung der Knotenmenge N auf Funktionen aus $\mathcal{F}$ ist. Ist $\mathcal{F}$ distributiv, wird $\mathcal{D}$ **distributiver Datenflußanalyserahmen** genannt. Ist $p = \langle n_1, \ldots, n_k \rangle$ ein Pfad, wird f_p definiert als $f_p \stackrel{\text{def}}{=} f_{n_k} \circ f_{n_{k-1}} \circ \ldots \circ f_{n_1}$. Für den leeren Pfad gilt: $f_{\langle\rangle} = \mathsf{id}$. □

Die DFA-Information kann entlang oder entgegen den Kanten des Kontrollflußgraphen ausgebreitet werden. Erstere DFA-Probleme werden **Vorwärts-Probleme**, letztere **Rückwärts-Probleme** genannt. Neben diesen **unidirektionalen Problemen** gibt es noch die **bidirektionalen Probleme**, deren Information sowohl von den Vorgängern als auch von den Nachfolgern abhängt. Dhamdhere und Khedker [1993] geben eine Bedingung an, unter welchen ein bidirektionales Problem in ein unidirektionales überführt werden kann. Für das bekannteste bidirektionale Problem, die **Elimination partieller Redundanzen** [Morel und Renvoise, 1979], geben Knoop *et al.* [1992] ein System mehrerer unidirektionaler Gleichungen an (s. Kapitel 6).

Im folgenden werden wir nur noch Vorwärtsprobleme betrachten, da mit dem (nach [Hecht, 1977, S.134] zitierten) Satz, Analoges auch für Rückwärtsprobleme gilt:

3.44 Satz (Vorwärts/Rückwärts-Konversion)
Ein Vorwärts- (Rückwärts-) Problem, das auf einem single exit *Kontrollflußgraphen definiert ist, kann in ein entsprechendes Rückwärts- (Vorwärts-) Problem über G^{-1} konvertiert werden, so daß das modifizierte Problem, die gleiche Lösung angibt wie das originale Problem.* □

[18] $\mathbb{N} \stackrel{\text{def}}{=} \{1, 2, \ldots\}$, Menge der positiven natürlichen Zahlen und $\mathbb{N}_0 \stackrel{\text{def}}{=} \{0, 1, 2, \ldots\}$.

3.45 Definition (Meet Over all Paths)
Die DFA-Information, die den Knoten n **erreicht** (oder vor n **gültig** ist), wird **Meet Over all Paths** (**MOP**) genannt und ist durch

$$\prod_{p \in \mathsf{path}[n]} f_p(c_0)$$

bestimmt. Der initiale Wert $c_0 \in \mathcal{L}$ hängt vom zu lösenden DFA-Problem ab. □

Leider ist der MOP in dieser Form nicht effektiv und effizient zu berechnen, da es exponentiell oder sogar unendlich viele Programmpfade geben kann.

Der nächste Abschnitte führt spezielle DFA-Rahmen und die Bitvektor-Darstellung eines DFA-Problems ein. Danach werden Methoden vorgestellt, mittels derer die DFA-Information effizient berechnet werden können.

3.4.3.4 Spezielle DFA-Rahmen

Der konstante DFA-Rahmen $\mathcal{D}^{\mathcal{C}}$ Als nächstes definieren wir einen speziellen Funktionenraum. Auf ihm basieren die theoretischen Resultate dieser Arbeit.

3.46 Definition (Konstanten Funktionenraum)
Sei $(\mathcal{C}, \sqsubseteq, \sqcap)$ ein beschränkter Halbverband und $\mathcal{F}^{\mathcal{C}}$ eine Menge von Funktionen $\mathcal{C} \to \mathcal{C}$, die nur die Identitätsfunktion id und für jedes Element c aus $\mathcal{C}$ die Konstantenfunktion const_c mit $\forall x \in \mathcal{C} : \mathsf{const}_c(x) = c$ enthält. $\mathcal{F}^{\mathcal{C}}$ heißt **konstanter Funktionenraum**. Die abkürzende Schreibweise $f = \mathsf{const}$ bedeutet, daß f eine Konstantenfunktion ist. □

Durch einfaches Nachrechnen ergibt sich folgendes Lemma:

3.47 Lemma
*$\mathcal{F}^{\mathcal{C}}$ ist ein monotoner und distributiver Funktionenraum. Der sich ergebende **konstante DFA-Rahmen** ($\mathcal{D}^{\mathcal{C}}$) ist monoton und distributiv.* □

Der boolesche DFA-Rahmen $\mathcal{D}^{\mathcal{B}}$ Viele DFA-Probleme können auf den booleschen Halbverband abgebildet werden, d.h. die Information ist an einem Programmpunkt gültig oder nicht.

3.48 Definition (Boolescher Halbverband)
Sei $\mathcal{B} \stackrel{\text{def}}{=} \{\bot, \top\}$. Die (Halb)ordnung der Elemente sei: $\bot \sqsubset \top$. $(\mathcal{B}, \sqsubseteq, \sqcap)$ wird der **boolesche Halbverband** genannt. Nullelement ist $\bot$, Einselement ist $\top$. □

3.49 Bemerkung
Interpretiert man die Werte von $\mathcal{B}$ als Wahrheitswerte TRUE und FALSE, dann zeigt einfaches Einsetzen der Definitionen, daß es nur zwei verschiedene binäre Operationen über $\mathcal{B}$ gibt, die eine *meet* Operation eines Halbverbandes sein können: $\wedge$ (*boolesche Konjunktion*) und $\vee$ (*boolesche Disjunktion*). Die anderen 14 binären Verknüpfungen über $\mathcal{B}$ haben nicht die in Definition 3.37 geforderten Eigenschaften, obwohl einige eine interessante Interpretation haben wie z.B. *xor*: Die DFA-Information ist genau dann vor einem Programmpunkt P gültig, wenn sie auf genau einem Pfad P erreicht.

Die Interpretation der beiden Operationen über den üblichen Werten TRUE und FALSE sind:

			$\sqcap = \wedge$, $\top =$ TRUE			$\sqcap = \vee$, $\top =$ FALSE		
a	b	$a \sqcap b$	a	b	$a \wedge b$	a	b	$a \vee b$
$\top$	$\top$	$\top$	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
$\top$	$\bot$	$\bot$	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
$\bot$	$\top$	$\bot$	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
$\bot$	$\bot$	$\bot$	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE

Ist nicht festgelegt, ob die Disjunktion oder Konjunktion als boolesche *meet*-Operation betrachtet werden soll, wird dies durch $\sqcap^{\cup/\cap}$ notiert. □

3.50 Korollar
Es gibt nur vier Funktionen $\mathcal{B} \to \mathcal{B}$: die beiden Konstantenfunktionen, die Identität und die Negation:

$$\begin{aligned} \mathsf{const}_\top(x) &\stackrel{\text{def}}{=} \top \text{ für alle } x \quad \text{(use)} \\ \mathsf{const}_\bot(x) &\stackrel{\text{def}}{=} \bot \text{ für alle } x \quad \text{(modify)} \\ \mathsf{id}(x) &\stackrel{\text{def}}{=} x \text{ für alle } x \quad \text{(identity)} \\ \overline{x} &\stackrel{\text{def}}{=} \top \text{ für } x = \bot \quad \text{(negation)} \\ \overline{x} &\stackrel{\text{def}}{=} \bot \text{ für } x = \top \end{aligned}$$

Bis auf die Negation sind alle Funktionen distributiv. Die Negation ist nicht monoton. Die Konstantenfunktionen werden häufig wie folgt interpretiert: use *definiert oder generiert eine Information;* modify *modifiziert oder invalidiert sie.* □

3.51 Lemma
Es gibt genau zwei Interpretationen der meet *Operation über $(\mathcal{B}, \sqsubseteq, \sqcap)$: boolesche Konjunktion und Disjunktion. Dabei entspricht die Konjunktion (Disjunktion) der Vorstellung, daß die Information, die vor einem Programmpunkt P gültig ist, auf* allen *Pfaden (auf* mindestens einem *Pfad), die P erreichen, gültig ist.*

Für jeden zwei-elementigen Halbverband $(\mathcal{B}, \sqsubseteq^{\cup/\cap}, \sqcap^{\cup/\cap})$ gibt es (bis auf Umbenennung der Elemente) genau einen mit ihm assoziierten monotonen ***booleschen Funktionenraum****: $\mathcal{F}^\mathcal{B} \stackrel{\text{def}}{=} \{\mathsf{const}_\top, \mathsf{const}_\bot, \mathsf{id}\}$. $\mathcal{F}^\mathcal{B}$ ist ein konstanter Funktionenraum.*
$\mathcal{D}^\mathcal{B} \stackrel{\text{def}}{=} (\mathcal{B}, \sqsubseteq^{\cup/\cap}, \sqcap^{\cup/\cap}, \mathcal{F}^\mathcal{B})$ wird der ***boolescher DFA-Rahmen*** *genannt.* □

Boolesche DFA-Probleme werden entsprechend dem *meet*-Operator eingeteilt in **Muß-Probleme** ($\sqcap = \cap$) und **Kann-Probleme** ($\sqcap = \cup$). Es gilt der folgende Satz, der nach [Hecht, 1977, S.134] zitiert wird:

3.52 Satz (Kann/Muss-Konversion)
Ein Kann- (Muß-) Problem kann in ein Muß- (Kann-) Problem transformiert werden, so daß die Lösung des modifizierten Problems eine Lösung des originalen Problems ergibt. □

3.4.3.5 Bitvektor-Darstellung des booleschen DFA-Rahmens

Endliche Mengen von „Programmobjekten"(z.B. Menge aller Ausdrücke oder Menge aller Variablen, die einen Programmpunkt erreichen) werden in der Implementierung meist sehr effizient als Bitvektoren dargestellt. Jedes solche Objekt wird durch eine Zahl aus dem Intervall $[1 \ldots |\text{Objekte}|]$ repräsentiert. Für die Klasse der $\mathcal{D}^\mathcal{B}$ DFA-Rahmen wird die Information eines Objektes an einem Programmpunkt durch die Werte TRUE und FALSE dargestellt: die Information ist entweder an dieser Stelle gültig oder nicht.

3.53 Definition
Ein **Bitvektor** ist die charakteristische Funktion bitvec einer endlichen Menge von Objektnummern $1 \ldots n = |\text{Objekte}|$. Also: $\mathsf{bitvec} : \{1 \ldots n\} \to \{\mathsf{TRUE}, \mathsf{FALSE}\}$. Da Bitvektoren endlichen Tabellen entsprechen, wird üblicherweise $\mathsf{bitvec}[i]$ statt des funktionalen $\mathsf{bitvec}(i)$ geschrieben.
Die Mengenoperationen $\cup, \cap, \overline{}$ werden für Bitvektoren durch die elementweise Anwendungen der booleschen Operationen $\vee, \wedge, \overline{}$ definiert. Die Mengendifferenz $a - b$ ist definiert durch

$a - b \stackrel{\text{def}}{=} a \cap \overline{b}$. Die leere Menge $\emptyset$ wird durch den Bitvektor repräsentiert, bei dem alle Elemente den Wert FALSE haben[19]. □

Für boolesche DFA-Probleme wird die Transferfunktion häufig als Mengengleichung in der folgenden Form angegeben:

$$f(x) = \mathsf{gen} \cup (x - \mathsf{kill}) \text{ oder } f(x) = \mathsf{gen} \cup x \cap \mathsf{transp}$$

gen (bzw. kill) stellt die DFA-Informationen dar, die von einer Anweisung bzw. einem Basisblock generiert (bzw. invalidiert) werden. transp stellt die Menge der DFA-Informationen dar, die nicht verändert werden. in bzw. out ist die Menge der Informationen, die vor bzw. nach einer Anweisung (oder Basisblock) gültig sind. Wie in Abschnitt 3.4.3.4 gezeigt, sind als $\sqcap$-Operation nur die boolesche Konjunktion und Disjunktion, also der Mengendurchschnitt und die Mengenvereinigung zulässig.

Wir definieren diese Mengen mit Hilfe der Transferfunktionen[20]. Für jede Anweisung s oder Basisblock b (oder wenn es egal ist, einfach σ) gibt es für jedes Programmobjekt o eine Transferfunktion f_σ^o.

3.54 Definition

Gegeben ein $\mathcal{D}^{\mathcal{B}}$-DFA-Problem. Sei σ entweder eine Anweisung oder ein Basisblock, c_0 die DFA-Information, die am Programmanfang σ_0 gilt und o eine Objektnummer.

$$\begin{aligned}
\mathsf{gen}_\sigma[o] = \mathsf{TRUE} &\text{ gdw } && f_\sigma^o = \mathsf{const}_\top \\
\mathsf{kill}_\sigma[o] = \mathsf{TRUE} &\text{ gdw } && f_\sigma^o = \mathsf{const}_\bot \\
\mathsf{transp}_\sigma[o] = \mathsf{TRUE} &\text{ gdw } && f_\sigma^o = \mathsf{id} \\
\mathsf{in}_\sigma[o] = \mathsf{TRUE} &\text{ gdw } && \prod_{p \in \mathsf{path}[\sigma)}^{\cup/\cap} f_p^o(c_0) = \top \\
\mathsf{out}_\sigma[o] = \mathsf{TRUE} &\text{ gdw } && \prod_{p \in \mathsf{path}[\sigma]}^{\cup/\cap} f_p^o(c_0) \stackrel{\dagger}{=} f_\sigma^o\Big(\prod_{p \in \mathsf{path}[\sigma)}^{\cup/\cap} f_p^o(c_0)\Big) = \top
\end{aligned}$$

Bemerkung: Die mit † markierte Gleichheit gilt, da $\mathcal{F}^{\mathcal{B}}$ distributiv ist. □

Aus diesen Definitionen folgen sofort (mittels Fallunterscheidung über die f_σ^o) die bekannten Gleichungen:

$$\begin{aligned}
\mathsf{out}_\sigma &= \mathsf{gen}_\sigma \cup \mathsf{in}_\sigma - \mathsf{kill}_\sigma \\
&= \mathsf{gen}_\sigma \cup \mathsf{in}_\sigma \cap \mathsf{transp}_\sigma
\end{aligned} \tag{3.1}$$

Sowie

$$\mathsf{in}_\sigma = \begin{cases} c_0 & \text{wenn } \sigma = \sigma_0 \\ \prod^{\cup/\cap}_{\sigma' \in \mathsf{pred}(\sigma)} \mathsf{out}_{\sigma'} & \text{sonst} \end{cases} \tag{3.2}$$

Und

$$\begin{aligned}
\mathsf{gen}_\sigma - \mathsf{kill}_\sigma &= \mathsf{gen}_\sigma \\
\mathsf{kill}_\sigma - \mathsf{gen}_\sigma &= \mathsf{kill}_\sigma
\end{aligned} \tag{3.3}$$

Es ist zu beachten, daß ein Objekt o in σ in nur genau einer der Mengen gen_σ, kill_σ oder transp_σ enthalten ist.

---

[19] Vorrangregeln für die Operatoren $\oplus$ und $-$ sind: $a \oplus b - c = a \oplus (b - c)$ und $a \oplus \bigoplus_{i \in I} b_i = a \oplus (\bigoplus_{i \in I} b_i)$

[20] Obwohl offensichtlich, ist diese Definition in der Literatur nicht zu finden.

3.4.3.6 Berechnen der DFA-Information

Der erste Ansatz, der versuchte, die DFA allgemein zu untersuchen, stammt von COCKE [1970]. Er stellt die DFA-Information als Bitvektoren dar und formuliert ein Gleichungsystem, das den Gleichungen 3.1 und 3.2 entspricht. Ein solches Gleichungsystem kann mit einem an die GAUSS'sche Eliminationsmethode zur Lösung linearer Gleichungen erinnernde Methode gelöst werden. Deshalb wird dieser DFA-Ansatz auch **Eliminationsansatz** genannt.

KILDALL [1972, 1973] basierte die DFA auf Halbverbänden und ermöglicht so, DFA-Probleme zu betrachten, die mit der Bitvektor-Darstellung nicht beschrieben werden können. Das Problem wird wieder mittels eines Gleichungssystems beschrieben, welches angibt, wie die DFA-Information für jeden Basisblock n in Abhängigkeit der DFA-Information der Vorgänger von n bestimmt wird:

$$\begin{aligned} \mathsf{in}_n &= \begin{cases} c_0 & \text{wenn } n = n_0 \\ \prod_{m \in \mathsf{pred}(n)} \mathsf{out}_m & \text{sonst} \end{cases} \\ \mathsf{out}_n &= f_n(\mathsf{in}_n) \end{aligned} \tag{3.4}$$

in_n ist die DFA-Information, die einen Basisblock n erreicht. out_n ist die DFA-Information, die am Ende von n (d.h. nach der letzten Anweisung von n) gültig ist. Für Rückwärtsprobleme muß pred durch succ sowie n_0 mit n_{exit} in den Gleichungen 3.4 ersetzt werden. Basierend auf dem Fixpunktsatz über Halbverbände kann ein iterativer Algorithmus zur Lösung angegeben werden. Daher heißt dieser Ansatz **Iterationsansatz**.

Ein dritter Ansatz, dem Eliminationsansatz ähnlich, ist der **Strukturansatz**, der die DFA auf dem Strukturbaum des Programms durchführt. Unsere Arbeit wird den strukturellen Ansatz für die PAR-Anweisung mit dem iterativen Ansatz für das übrige Programm verbinden.

Zur Bestimmung des Aufwands solcher Algorithmen, wenn sie mit Bitvektoren arbeiten, wird die Anzahl Durchschnitts-, Vereinigungs- und Negationsoperationen – der **Bitvektorschritte** – benutzt. Sei n die Anzahl der Knoten im CFG und e die Anzahl der Kanten, dann kann in der Praxis $e < 2n$ angenommen werden. d sei die Tiefe eines reduziblen CFG's.
Im folgenden gehen wir näher auf die ersten drei Ansätze ein. Weitere Methoden (z.B. T1-T2 Analyse, *Nodelisting*) sind in [HECHT, 1977] präsentiert.

Eliminationsansatz Eine einfache GAUSS'sche Elimination benötigt $O(ne) = O(n^2)$ Bitvektorschritte. Ist der CFG jedoch reduzibel, kann, basierend auf der abgeleiteten Intervallfolge, die Lösung schneller erfolgen. Dazu wird out_h für den Kopf h der Intervalle der Graphen $G_1, G2, \ldots, G^*$ in dieser Reihenfolge berechnet, dabei ist $\mathsf{in}_h = \emptyset$. Mit anderen Worten: out wird nur in Abhängigkeit von Knoten innerhalb des Intervalls berechnet. In einer zweiten Phase werden die Graphen in der Reihenfolge $G^*, \ldots, G_2, G_1$ besucht. Nun kann in_h in Abhängigkeit von Knoten innerhalb und außerhalb des Intervalls bestimmt werden; somit in für alle Knoten des Intervalls. Die Knoten eines Intervalls werden in Intervallordnung besucht. Der Aufwand beträgt $O(nd)$ Bitvektorschritte.

Iterationsansatz KILDALL bewies folgenden grundlegenden Satz:
3.55 SATZ (SEQUENTIELLE KOINZIDENZ)
Für distributive DFA-Probleme ist die größte Lösung des Gleichungssystems 3.4 mit dem MOP koinzident. □

Der iterative Algorithmus 3.56 berechnet die größte Lösung (manchmal auch **maximalen Fixpunkt**, MFP genannt) des Gleichungssystems 3.4. Er ist hier in der Form von [AHO *et al.*, 1986] wiedergegeben. Algorithmus 3.56 legt keine Reihenfolge fest, in der die Basisblöcke in der REPEAT Schleife besucht werden. Die Besuchsreihenfolge hat jedoch einen entscheidenden Einfluß auf die Konvergenzgeschwindigkeit. Üblicherweise werden die Basisblöcke in der

rPOSTORDER [HECHT, 1977, S.107] (auch *reverse depth-first order* genannt, [AHO *et al.*, 1986]) Reihenfolge besucht. Diese Besuchsreihenfolge entspricht einem umgekehrten Postfixdurchlauf des Tiefensuchbaums, der den Graphen aufspannt. Mit dieser Besuchsreihenfolge werden in einem reduziblen CFG nur $d+2$ Iterationen benötigt, so daß der Gesamtaufwand $O(nd)$ Bitvektorschritte beträgt. Genauere Angaben über die Komplexität des Algorithmus 3.56 finden sich z.B. in [HECHT und ULLMAN, 1973, 1975; KAM und ULLMAN, 1976; MARLOWE und RYDER, 1990].

3.56 ALGORITHMUS SEQUENTIELLE DFA (cfg : `Kontrollflußgraph`)
Berechne die größte Lösung des Gleichungssystems 3.4.

---

forall Basisblöcke n **do** /* *Initialisierung* */
  $n.\mathsf{in} \leftarrow \top$; $n.\mathsf{out} \leftarrow f_n(\top)$;
end
$n_0.\mathsf{in} \leftarrow c_0 \in L$; $n_0.\mathsf{out} \leftarrow f_{n_0}(c_0)$; /* *Initialwert c_0 hängt vom DFA-Problem ab.* */
repeat
  changed ← **false**;
  forall Basisblöcke n **do**
    old ← $n.\mathsf{out}$;
    $n.\mathsf{in} \leftarrow \prod_{n' \in \mathsf{pred}(n)} n'.\mathsf{out}$; $n.\mathsf{out} \leftarrow f_n(n.\mathsf{in})$;
    if *old* $\neq n.\mathsf{out}$ **then**
      changed ← **true**
    end
  end
until *changed* = **false** □

3.57 SATZ (KAM und ULLMAN [1977])
Für monotone und nicht-distributive DFA-Rahmen berechnet Algorithmus 3.56 den maximalen Fixpunkt, für diesen gilt MFP $\sqsubseteq$ MOP. Es gibt keinen Algorithmus, der für beliebige monotone und nicht-distributive DFA-Rahmen die MOP-Lösung berechnet. □

Strukturansatz Für strukturierte Programme (ohne `GOTO`'s, etc.) kann die DFA-Information effizienter berechnet werden, als dies für solche mit `GOTO`'s möglich ist. Statt den Eliminationsansatz oder den Fixpunktalgorithmus über dem Kontrollflußgraphen zu benutzen, reicht eine Bottum-Up / Top-Down Traversierung des Strukturbaumes des Programms aus. Bottum-Up werden die gen und kill Mengen eines jeden Strukturbaumknotens bestimmt, Top-Down die in und out Mengen.

Wir definieren als nächstes die Mengengleichungen über die Anweisungen im Strukturbaum des Programms, siehe [AHO *et al.*, 1986, S.611 ff], [BABICH und JAZAYERI, 1978]. Für alle Anweisungen gilt Gleichung (3.1). Ausdrücke, die nicht in Zuweisungen vorkommen, werden so behandelt, als ob sie zuvor an neue Hilfsvariablen zugewiesen werden. Für die Hintereinanderausführung $S_1; S_2$ gilt[21]:

$$\mathsf{in}_1 = \mathsf{in}_{1;2}$$
$$\mathsf{in}_2 = \mathsf{out}_1$$
$$\mathsf{out}_{1;2} = \mathsf{out}_2 = \mathsf{gen}_2 \cup \mathsf{gen}_1 - \mathsf{kill}_2 \cup \mathsf{in}_{1;2} - (\mathsf{kill}_1 \cup \mathsf{kill}_2) \quad (3.5a)$$
$$\mathsf{gen}_{1;2} = \mathsf{gen}_2 \cup \mathsf{gen}_1 - \mathsf{kill}_2 \quad (3.5b)$$
$$\mathsf{kill}_{1;2} = \mathsf{kill}_2 \cup \mathsf{kill}_1 - \mathsf{gen}_2 \quad (3.5c)$$
$$\mathsf{transp}_{1;2} = \mathsf{transp}_1 \cap \mathsf{transp}_2 \quad (3.5d)$$

[21] Das Symbol X_{σ_i} wird zu X_i abgekürzt.

Für Verzweigungen `IF` $\textit{expr}$ `THEN` S_1 `ELSE` S_2 `END` gilt[22]:

$$\begin{aligned}
\mathsf{in}_1 &= \mathsf{in}_2 = \mathsf{in}_{\mathtt{IF}} \\
\mathsf{out}_{\mathtt{IF}} &= \mathsf{out}_1 \sqcap^{\cup/\cap} \mathsf{out}_2 \\
\mathsf{gen}_{\mathtt{IF}} &= \mathsf{gen}_1 \sqcap^{\cup/\cap} \mathsf{gen}_2 \\
\mathsf{kill}_{\mathtt{IF}} &= \mathsf{kill}_1 \sqcup^{\cup/\cap} \mathsf{kill}_2 \\
\mathsf{transp}_{\mathtt{IF}} &= \mathsf{transp}_1 \cap \mathsf{transp}_2
\end{aligned}$$

Für die Schleife `REPEAT` S `UNTIL` $\textit{expr}$ gilt:

$$\begin{aligned}
\mathsf{in}_S &= \mathsf{in}_{\mathtt{REPEAT}} \sqcap^{\cup/\cap} \mathsf{out}_{\mathtt{REPEAT}} = \mathsf{in}_{\mathtt{REPEAT}} \sqcap^{\cup/\cap} (\mathsf{gen}_{\mathtt{REPEAT}} \cup \mathsf{in}_{\mathtt{REPEAT}} - \mathsf{kill}_{\mathtt{REPEAT}}) \\
\mathsf{out}_{\mathtt{REPEAT}} &= \mathsf{out}_S \\
\mathsf{gen}_{\mathtt{REPEAT}} &= \mathsf{gen}_S \\
\mathsf{kill}_{\mathtt{REPEAT}} &= \mathsf{kill}_S \\
\mathsf{transp}_{\mathtt{REPEAT}} &= \mathsf{transp}_S
\end{aligned}$$

3.4.4 DFA-Spezifikation der Beispiele

Wir sind nun in der Lage, die Beispiele aus Abbildung 3.3 auf Seite 23 vollständig zu spezifizieren. Bei den ersten vier Problemen handelt sich um die Frage: „Ist eine Information (Definition/Ausdruck erreicht Programmpunkt P) gültig oder nicht?“, also um boolesche DFA-Probleme. Die Mengen gen und kill werden für jede Anweisung bestimmt, s. Abb. 3.4.

RD	AE
$\mathsf{gen}_s = \{d_x\}$	$\mathsf{gen}_s = \{e \in \mathcal{TA}[\textit{expr}] \mid x \notin \mathcal{TA}[e]\}$
$\mathsf{kill}_s = \mathcal{L}_x^{\text{defs}} - \{d_x\}$	$\mathsf{kill}_s = \{e \in \mathcal{L}_x^{\text{exprs}} \mid x \in \mathcal{TA}[e]\}$

$x \in \mathcal{TA}[e]$ bedeutet, daß die Variable x im Ausdruck e vorkommt.

Abbildung 3.4: gen und kill der Transferfunktion der Anweisung x `:=` $\textit{expr}$ und die *Available-Expressions* und *Reaching-Definitions* DFA-Probleme

Beim CC-Problem kommt etwas Neues hinzu: um die Werte von Variablen zu bestimmen, müssen die Ausdrücke des Quellprogramms symbolisch ausgewertet werden. Symbolisch deshalb, weil wir die genauen Werte der beteiligten Variablen i.d.R. nicht kennen. Der symbolische Wert $I[x]$ der Variablen x ist ein Element aus der Menge „weiß noch nichts“, „ist definitiv keine Konstante“[23] und „hat definitiv den konstanten Wert c_i“. Die Transferfunktion des CC-Problems bestimmt für jede Variable und jede Anweisung den symbolischen Wert dieser Variablen. Die Regeln dazu sind in Abbildung 3.5 auf der nächsten Seite angegeben; entsprechend wird auch die *meet*-Operation für das CC-Problem ($\sqcap^{cc}$) definiert, s. Abbildung 3.6 auf der nächsten Seite.

Zum Schluß ist in Abbildung 3.7 die Klassifikation der genannten fünf Probleme angegeben.

[22]Unabhängig von der Problemklasse muß für transp offensichtlich der Mengendurchschnitt benutzt werden.
[23]Im Sinne von: x kann durchaus einen konstanten Wert haben, diese Analyse kann dies aber nicht feststellen.

Ausdruck e	Information $I[e]$	
$\boldsymbol{x}$	$\top$,	falls es noch keine Definition für $\boldsymbol{x}$ gibt
	$I[\boldsymbol{expr}]$	falls $\boldsymbol{expr}$ der Ausdruck ist, der $\boldsymbol{x}$ zugewiesen wurde.
$\otimes e_1$	$I[e_1]$,	wenn $I[e_1] \in \{\top, \bot\}$
	$\otimes' c_{e_1}$,	wenn $I[e_1] = c_{e_1}$
$e_1 \otimes e_1$	$\top$,	wenn $\forall i \in \{1, 2\} : I[e_i] \in \{\top, c_{e_i}\}$
	$\bot$,	wenn $\exists i \in \{1, 2\} : I[e_i] = \bot$
	$c_{e_1} \otimes' c_{e_2}$,	wenn $\forall i \in \{1, 2\} : I[e_i] = c_{e_i}$

$\boldsymbol{x}$ ist eine Quellprogrammvariable, $\otimes$ ist ein Operator der Quellsprache, $\otimes'$ der entsprechende Operator, mit dem der Übersetzer Werte von Konstantenausdrücken berechnen kann.

Abbildung 3.5: Symbolische Auswertung von Ausdrücken für das CC-Problem.

$I[\boldsymbol{x}] \stackrel{\text{def}}{=}$

$I'[\boldsymbol{x}]$	$I''[\boldsymbol{x}]$			
	$\bot$	c_i	$c_j, (j \neq i)$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
c_i	$\bot$	c_i	$\bot$	c_i
$\top$	$\bot$	c_i	c_j	$\top$

Sei $I'[\boldsymbol{x}]$ die DFA-Information für die Variable $\boldsymbol{x}$, die den Programmpunkt P über den Pfad p' erreicht. Entsprechend erreicht $I''[\boldsymbol{x}]$ den Punkt P über den Pfad p''. $I[\boldsymbol{x}]$ ist die DFA-Information, die an P für Variable $\boldsymbol{x}$ gilt. Die $\sqcap^{cc}$ basiert somit auf einer flachen Halbordnungbeziehung. Für mehr als zwei Pfade wird die Tabelle entsprechend definiert.

Abbildung 3.6: Die $\sqcap^{cc}$ Operation

3.4.5 Static Single Assignment Form

Als Grundlage vieler Optimierungen ist das Wissen, ob zwei Ausdrücke den gleichen Wert berechnen, von großer Bedeutung. Andererseits ist die Äquivalenz von Programmen und somit auch die Äquivalenz von Ausdrücken im allgemeinen nicht entscheidbar. Doch wie beim Optimieren von Programmen üblich, ist es ausreichend, wenn für eine große Klasse von Ausdrücken ihre Äquivalenz festgestellt werden kann.

Das erste Verfahren, Wertnumerierung, stammt von Cocke und Schwartz [1970]. Es teilt die Ausdrücke innerhalb eines Basisblocks in Äquivalenzklassen ein, die durch eine Nummer, ihre *Wertnummer*, gekennzeichnet sind. Alle Ausdrücke einer Äquivalenzklasse berechnen zur Laufzeit den gleichen Wert.

Da das Ziel unserer Darstellung die *Static Single Assignment Form* ist, benutzen wir eine etwas einfachere Version der Wertnumerierung. Ihre wesentliche Einschränkung liegt darin, nur für *syntaktische gleiche* Ausdrücke zu untersuchen, ob diese den gleichen Wert berechnen und somit innerhalb einer Klasse liegen. Abbildung 3.8 zeigt ein Beispiel für die vereinfachte Wertnumerierung.

Problem	Richtung	Art	$\sqcap$ Operator	Eigenschaft
AE	vorwärts	muß	Mengen-$\cap$	distributiv
BE	rückwärts	muß	Mengen-$\cap$	distributiv
RD	vorwärts	kann	Mengen-$\cup$	distributiv
LV	rückwärts	kann	Mengen-$\cup$	distributiv
CC	vorwärts	muß	$\sqcap^{cc}$	monoton

Abbildung 3.7: Klassifikation einiger DFA-Probleme.

(1)	e	$:=$	$a;$
(2)	f	$:=$	$b;$
(3)	x	$:=$	$(a+b)*(c+d);$
(4)	a	$:=$	$(e+f);$
(5)	y	$:=$	$(a+b)*(c+d);$

Durch „Ansehen" des Programmfragments stellt man fest, daß die Ausdrücke $(c+d)$ in allen Zeilen den gleichen Wert haben. Die Ausdrücke $(a+b)$ aus Zeile (3) und $(e+f)$ aus Zeile (4) berechnen ebenfalls den gleichen Wert. In diesem Programmfragment sind jedoch nur die Ausdrücke aus den Zeilen (3) und (5) syntaktisch gleich. Durch weitere Analyse stellt sich heraus, daß $(a+b)$ in Zeile (3) und Zeile (5) verschieden sind, da dazwischen der Operand a modifiziert wurde. Wertnumerierung stellt somit fest, daß nur die Ausdrücke $(c+d)$ in den Zeilen (3) und (4) garantiert den gleichen Wert haben.
Die vollständige Wertnumerierung gemäß [Cocke und Schwartz, 1970] würde feststellen, daß $e+f$ und $a+b$ aus Zeile (4) bzw. Zeile (3) den gleichen Wert berechnen (die Wertnummer von a ist gleich der von e bzw. die von e und b sind gleich (Zeilen (1) und (2)), damit berechnen $e+f$ und $a+b$ aus Zeile (4) bzw. Zeile (3) den gleichen Wert).

Abbildung 3.8: Ein Beispiel für Wertnumerierung.

Das Hauptproblem der vereinfachten Wertnumerierung ist es festzustellen, ob einer der Operanden eines Ausdrucks sich seit der letzten Berechnung des Ausdrucks geändert hat. Von Nachteil ist auch, daß die (vereinfachte und vollständige) Wertnumerierung nur innerhalb eines Basisblocks arbeitet und nicht über dessen Grenzen hinaus.

Wird jede Variable, die in zwei syntaktisch gleichen Ausdrücken enthalten ist, von genau einer Definition erreicht und stimmt diese Definition in beiden Ausdrücken überein, so berechnen die beiden Ausdrücke offensichtlich den gleichen Wert. Hat man also die *Reaching-Definitions* DFA-Information für ein Programm berechnet, läßt sich die Wertegleichheit von syntaktisch gleichen Ausdrücken leicht feststellen.

Ein anderer Lösungsweg für das erste Problem besteht in der Forderung, daß jeder Variablen nur ein einziges Mal ein Wert zugewiesen wird. Gilt dies, so folgt aus der syntaktischen Äquivalenz auch die Wertegleichheit der Ausdrücke (sofern alle Variablen vor ihrer Benutzung auch definiert werden müssen). Für imperative Programme ist dies natürlich viel zu restriktiv, weshalb nur gefordert wird, daß es pro Variable nur eine Zuweisung im Programmtext geben darf (*Static Single Assignment*, **SSA**). Bei jeder Zuweisung an eine Variable im originalen Programm wird ihr eine eindeutige SSA-Nummer zugeordnet. Als Variable eines Programms in SSA-Form wird das Paar *(Variablenname, Nummer)* aufgefaßt. Damit gibt es für jede Benutzung einer SSA-Variablen genau eine Definition im Programmtext. Syntaktische Gleichheit von Ausdrücken wird nun so definiert, daß diese Nummer einbezogen wird. Abbildung 3.9 zeigt ein Beispiel für die SSA-Form eines Programms.

(1)	e_1	$:=$	$a_1;$
(2)	f_1	$:=$	$b_1;$
(3)	x_1	$:=$	$(a_1+b_1)*(c_1+d_1);$
(4)	a_2	$:=$	$(e_1+f_1);$
(5)	y_1	$:=$	$(a_2+b_1)*(c_1+d_1);$

Die Zuweisung in Zeile (4) inkrementiert die Variablennummer von a. Die Ausdrücke (c_1+d_1) aus Zeile (3) und (5) berechnen den gleichen Wert. Da sich die Variablennummern in den Ausdrücken (a_1+b_1) aus Zeile (3) und (a_2+b_1) aus Zeile (5) unterscheiden, sind ihre Werte möglicherweise verschieden.

Abbildung 3.9: Programmfragment ohne Sprünge in SSA-Form.

Bleibt noch die Lösung des zweiten Problems: wie können diese Variablennummern über

Basisblockgrenzen hinweg benutzt werden? Betrachten wir das Programmfragment in Abbildung 3.10, so stellt sich die Frage: welche Zuweisung an Variable x erreicht die Benutzung von x in Zeile (6): entweder die aus Zeile (3) oder die aus Zeile (4). Damit hätten wir aber die *single assignment* Bedingung verletzt. Die Lösung des Problems liegt in der Einführung „künstlicher“ Zuweisungen $x_j := \phi(x_{i_1}, \ldots, x_{i_k})$ (**ϕ-Zuweisungen**). Die ϕ Funktion führt die verschiedenen Definitionen $(x_{i_1}, \ldots, x_{i_k})$ zusammen, ihre Bedeutung ist: „wird zur Laufzeit der i_j-te Vorgänger des Basisblocks ausgeführt, nimm den Wert von x_{i_j}.“.

(1) $x_1 := \ldots;\ y_1 := \ldots;$
(2) IF $x_1 > 2 * a_1 + 2 * y_1$
(3) THEN $x_2 := 2 * a_1 + b_1 * x_1;\ y_2 := \ldots;$
(4) ELSE $x_3 := 2 * a_1 + c_1 * x_1;$
(5) END;
(5’) $x_4 := \phi(x_2, x_3);$
(5”) $y_3 := \phi(y_1, y_2);$
(6) $y_1 := 2 * x_4 + 2 * a_1 + 2 * y_3;$

Einfügen einer neuen Zuweisung in Zeilen (5’,5”) mit einer ϕ-Funktion auf der rechten Seite. Dies erlaubt, die Numerierung über Basisblockgrenzen hinweg zu benutzen. Jeden Zweig der IF Anweisung erreichen die gleichen SSA-Variablen, z.B. x_1 in Zeile (3) und (4). In Zeilen (2), (3), (4) und (6) berechnet $2*a$ den gleichen Wert, während $2*x$ in den Zeilen (2) und (6) verschieden sind.

Abbildung 3.10: Programmfragment in SSA-Form.

Die ϕ-Zuweisungen werden am Anfang der Basisblöcke, die sie benötigen, eingefügt. Ein Basisblock n benötigt eine ϕ-Funktion für Variable v, wenn n der Konvergenzpunkt von zwei Pfaden mit verschiedenen Anfängen ist und diese Anfänge Zuweisungen an v enthalten (entweder „normale“ oder solche, deren rechte Seite selbst eine ϕ-Funktion ist). Diese Konvergenzpunkte werden mittels der Dominanzgrenzen von Basisblöcken bestimmt.

3.58 DEFINITION (STATIC SINGLE ASSIGNMENT FORM)
Ein Programm ist in ***Static Single Assignment Form***, wenn jede Variable Ziel genau einer Zuweisung im Programmtext ist. [CYTRON *et al.*, 1991] □

Die SSA-Form ist somit also nichts anderes als eine andere Darstellung der *Reaching-Definitions* DFA-Information. Der Algorithmus, der ein gegebenes Programm in ein entsprechendes in SSA-Form transformiert, arbeitet im Prinzip wie folgt:

3.59 ALGORITHMUS TRANSFORMATION IN SSA-FORM (*cfg* : Kontrollflußgraph)

1. Berechne die Dominanzgrenzen des Kontrollflußgraphen.
2. Benutze diese, um die Basisblöcke zu finden, in denen ϕ-Zuweisungen eingefügt werden müssen.
3. Die Variablen des ursprünglichen Programms werden numeriert, wobei jedes Auftreten der Variablen V durch ein entsprechendes Paar (V, i) ersetzt wird.

Der Algorithmus ist ausführlich in [CYTRON *et al.*, 1991] sowie im Abschnitt 5.7.4 (zusammen mit der Erweiterung für parallele Programme) beschrieben. □

3.4.6 Schlußfolgerungen

In diesem Abschnitt konnten nur die für unsere Arbeit benötigten, grundlegenden Verfahren der Programmanalyse vorgestellt werden. In folgenden Büchern werden weitere Methoden vorgestellt: [HECHT, 1977; MUCHNICK und JONES, 1981; WOLFE, 1989, 1996].

3.5 Literaturüberblick über die Analyse und Darstellung paralleler Programme

3.5.1 Verifikationsmethoden

Mit Beweiskalkülen wird versucht, ein sequentielles oder paralleles Programm bezüglich seiner Spezifikation als korrekt zu beweisen. In [Cousot, 1981] wird ein Programm als *diskretes dynamisches System* [Keller, 1976; Pnueli, 1977], d.h. als eine Übergangsrelation auf Zuständen aufgefaßt. Bereits in dieser allgemeinen Umgebung können für die DFA wichtige Resultate gewonnen werden. Basierend auf diesem Ausführungsmodell werden in [Cousot, 1984] Methoden zum Beweis von Invarianten wie z.B. partielle Korrektheit, Verklemmungsfreiheit etc. von parallelen Programmen mit gemeinsamem Speicher vorgestellt. Ebenfalls mit Beweismethoden für (datenunabhängige) parallele Programme beschäftigen sich Owicki und Gries [1976]. Cousot und Cousot [1980] untersuchen die Semantik von kommunizierenden sequentiellen Prozessen, wobei [Hoare, 1978, 1985] u.a. wieder das Ziel hat, Beweismethoden für diese zu finden.

3.5.2 Erkennung von Datenzugriffskonflikten

Häufig werden Konflikte beim Zugriff auf den gemeinsamen Speicher (*race conditions*) als Problem angesehen. Einige Arbeiten beschäftigen sich mit ihrer Erkennung zur Übersetzungszeit, z.B. [Bristow *et al.*, 1988; Balasundaram und Kennedy, 1989; Callahan und Subhlok, 1988; McDowell und Helmbold, 1989; Netzer und Miller, 1992]. Diese Arbeiten lassen sich jedoch nicht auf unsere Aufgabe übertragen:

> *However, there is a fundamental difference between static analysis for the purpose of debugging and program optimization. In the former, only well-synchronized programs (e.g. synchronized accesses to shared variables) are considered correct; unordered accesses to a shared variable are bugs to be detected. Thus the compiler can focus on analyzing the synchronization patterns. However, when we apply a program optimization, the first assumption is that input programs are correct and thus any optimization should result in consistency with the programs. (In fact, there are chaotic or intended nondeterministic algorithms where accesses to the same shared variable need not be synchronized.)* [Chow und Harrison, 1992b]

3.5.3 Kontrollflußanalyse

Im Zusammenhang mit der *Static Single Assignment Form* definieren Wolfe und Srinivasan [1991] einen hierarchischen parallelen Kontrollflußgraphen. In ihm gibt es zwei Sorten von Knoten: solche, die „normale" Basisblöcke darstellen und *Superknoten*, die einen ganzen Kontrollflußgraphen darstellen (s. Abschnitt 5.2.1) . Daß es allerdings noch keinen „Standardkontrollflußgraphen" für parallele Programme gibt, zeigt die Bemerkung „*It is unclear whether one can design a flow graph for general analyses of parallel programs.*" [Chow und Harrison, 1994].

3.5.4 Datenflußanalyse

Die Arbeiten von Reif [1979, 1984]; Reif und Smolka [1990] beschäftigen sich mit dem speziellen Problem der *Erreichbarkeit*, d.h. mit der Frage, ob eine bestimmte Anweisung in einem parallelen Programm ausgeführt werden kann. Als Programmiermodell wird angenommen, daß die Prozesse nur über Nachrichtenaustausch miteinander kommunizieren. Die Lösung des

Erreichbarkeitsproblems erlaubt Aussagen über die Verklemmungsfreiheit des Programms. Es wird gezeigt, daß das allgemeine[24] Erreichbarkeitsproblem unentscheidbar ist. Liegt eine statische Kommunikationsstruktur[25] vor, ist das Erreichbarkeitsproblem entscheidbar, benötigt aber unendlich oft exponentiell viel Platz („... *testing reachability is decidable but requires exponential space, infinitely often*") [Reif, 1984]. Deshalb wird versucht, das Problem mittels DFA näherungsweise zu lösen. Die DFA arbeitet über dem *event spanning graph*, der die Kontrollflußgraphen der Prozesse mit den durch die Kommunikation entstehenden Ereigniskanten anreichert.

Grunwald und Srinivasan [1992a,b, 1993] entwickeln Datenflußgleichungen für das *Reaching-Definitions* (RD) DFA-Problem zur Analyse explizit paralleler Programme. Sie betrachten eine parallele Programmiersprache, in welcher die Nebenläufigkeit durch eine Variante der `cobegin/coend` Anweisung spezifiziert wird (*parallel sections*). Ein Prozeß kann auf die Terminierung eines anderen mittels der `wait` Anweisung warten. Die Autoren basieren ihre Analysen auf der Tatsache, daß sie im Übersetzer für die Prozeßrümpfe eine *Copy In / Copy Out* Semantik[26] für die gemeinsamen Variablen annehmen. Damit können die parallelen Sektionen unabhängig voneinander optimiert werden (vgl. das Beispiel in Kapitel 2). Die Programmiersprache ihrer Wahl, PCF Fortran [PCF, 1991], läßt diese Interpretation zu. *PCF Fortran* fordert, daß parallele Sektionen, mit Ausnahme an den Synchronisationspunkten, datenunabhängig sind. Existieren dennoch Abhängigkeiten, werden diese als Anomalien begriffen. Replikatoren bzw. parallele `forall` Schleifen sind nicht in der Sprache enthalten.

Sie betrachten Programme mit und ohne Synchronisierung durch eine `wait` Anweisung. Zur Modellierung des Datenflusses für die parallelen Sektionen benötigen sie zwei weitere Gleichungssysteme: *MustKillIn/Out* bestimmt die durch die parallele Sektion invalidierte RD-Information und *LocalIn/Out* die in der `cobegin/coend` Anweisung gültige RD-Information. Zur Bestimmung der Lösung wird ein iterativer Algorithmus benutzt. Diesen Gleichungssystemen liegt keine formale Herleitung zugrunde, sondern sie werden nur informell über die syntaktische Struktur motiviert. Replizierte Prozesse können mit ihrer Argumentation nicht behandelt werden.

3.5.5 Static Single Assignment Form

Alpern *et al.* [1988]; Rosen *et al.* [1988]; Cytron *et al.* [1989, 1991] definieren das Konzept der SSA-Form und geben Algorithmen an, mit denen die (minimale) SSA Form eines Programms bestimmt werden kann. Cytron und Ferrante [1995]; Sreedhar und Gao [1995] geben lineare Algorithmen zur Berechnung der Einfügestellen der ϕ-Funktionen an. Brandis und Mössenböck [1994] haben für strukturierte Programme (ohne `GOTO`'s, etc.) ein effizientes Ein-Paß-Verfahren zur Erzeugung der SSA-Form entwickelt.

Srinivasan und Wolfe [1991]; Srinivasan und Grunwald [1991]; Srinivasan *et al.* [1993] definieren eine Erweiterung der SSA-Form für explizit parallele Programme. Ihre Untersuchungen basieren auf *PCF Fortran*. Die Abhängigkeiten, die durch die *PCF Fortran* `wait` Anweisung entstehen, werden durch den *Parallel Precedence Graph* abgebildet. Zusammen mit den Kontrollflußabhängigkeiten des *Parallel Control Flow Graph* definieren sie einen *Extended Flow Graph*. Analog zum sequentiellen Fall wird über diesem die *Parallel*

[24] Die Kommunikationsstruktur ist nicht statisch festgelegt, d.h. die Kanäle in den *send* und *receive* Anweisungen dürfen variabel sein.

[25] Die Kanäle sind Konstanten.

[26] ***Copy In /Copy Out Semantik***: Jeder Prozeßrumpf hat eine eigene lokale Kopie der Variablen. Initial haben diese den Wert, den die Variablen vor der Ausführung der parallelen Sektionen haben. Nach der Terminierung werden die Werte zusammengeführt. Dies ist vergleichbar dem gleichnamigen Parameterübergabemechanismus.

Dominance Frontier eingeführt. Die ϕ-Zuweisungen werden dann entsprechend dieser eingefügt. Der *Parallel Precedence Graph* wird zur Aufdeckung von Anomalien in parallelen Sektionen benutzt. Die Einfügung von ψ-Zuweisungen, die diese – ähnlich der ϕ-Zuweisung, zusammenfassen, bauen auf dem *Parallel Precedence Graph* auf.

3.5.6 Eigene Arbeiten

Die Grundlagen dieser Arbeit (Betrachtung aller Interleavings; Ausnützen der Eigenschaften der Bitvektor-DFA; die Gleichungen aus Satz 4.39) sind in [Vollmer, 1994] erstmalig beschrieben. Da die Sätze allerdings „direkt" für die Bitvektor-DFA formuliert sind, sind die Beweise, obwohl ihnen die gleiche Idee zugrunde liegt, sehr umfangreich. [Vollmer, 1995] ist der Vorgänger in Form und Inhalt des Theoriekapitels 4.

Neben der Betrachtung aller Interleavings gibt es jedoch noch eine andere Möglichkeit, die Pfade zu beschreiben, die ein paralleles Programm kennzeichnen. Dazu wird in [Knoop, Steffen und Vollmer, 1996] ein Ansatz benutzt, der dem *functional approach* zur interprozeduralen Analyse sequentieller Programme von Sharir und Pnueli [1981] folgt. Prozeduren werden als *collections of structured blocks* verstanden, für die es gilt, eine Eingabe/Ausgabe-Funktion zu berechnen. Danach werden Prozeduraufrufe als *super operations* betrachtet, die durch ihre Eingabe/Ausgabe-Funktion charakterisiert werden. Im Kontrollflußgraph wird für jeden Prozeduraufruf eine Kante zum Prozeduranfang und für die Rückkehr aus der Prozedur eine Kante von der Prozedur zum Aufrufer eingefügt. Damit erhält man eine Obermenge der interprozedural möglichen Pfade, die durch eine Konsistenzbedingung definiert sind („die Rückkehrkante muß der Aufrufkante entsprechen"). Im parallelen Fall werden die Prozeßrümpfe einer `PAR` Anweisung und die `PAR`-Anweisung selbst entsprechend behandelt. Man kann so für einen Basisblock die Menge der *Interleaving-Vorgängerbasisblöcke* definieren. Die Eigenschaft, die dieses Vorgehen erlaubt, ist die $\sqcap$-Abgeschlossenheit (s. Satz 4.2) des booleschen DFA-Rahmens. Diese Vorgehensweise liegt mehr in der „Tradition" der Arbeiten von Knoop und Steffen, siehe z.B. [Knoop und Steffen, 1992].

In [Knoop, Steffen und Vollmer, 1995] wird die *Busy Code Motion* (s. Abschnitt 6.4) basierend auf obigem Ansatz auf parallele Programme übertragen. Dazu wird für einen Basisblock b die Transferfunktion $f_b \in \{\mathsf{const}_\top, \mathsf{const}_\bot, \mathsf{id}\}$ in Termen der Grundprädikate TRANSP und COMP bestimmt.

3.5.7 Schlußfolgerungen

Die Arbeiten über die abstrakte Interpretation deuten an, wie wichtig es ist, der Analyse eine fundierte semantische Grundlage zu geben. Andererseits weisen sie auf die Notwendigkeit hin, eine Basis zu wählen, die einfach genug ist, um eine effiziente Analyse zu ermöglichen. Damit sind bereits zwei Kriterien genannt, die diese Arbeit erfüllen soll. Ein weiteres Merkmal ist die Auswahl der Programmiersprache, auf der die Analysen und Optimierungen arbeiten. Es wird sich zeigen, daß es nicht nötig ist für die Analyse die *Copy In / Copy Out* Semantik zu fordern. Da sie so auf keinem existierenden Rechnersystem mit gemeinsamem Speicher implementiert bzw. effizient implementierbar ist, muß ein realistischeres Maschinenmodell angenommen werden. Diese Forderungen sind, wie in diesem Kapitel beschrieben, in unserer Arbeit erfüllt.

Das größte Problem der Analyse paralleler Programme ist offensichtlich die Reduktion der Analysezustände, welche durch den unbeschränkten, nebenläufigen Zugriff auf den gemeinsamen Speicher entstehen. Es wird in Kapitel 4 gelöst.

Da es noch keine einheitliche Begriffswelt für die Programmanalyse paralleler Programme gibt, muß sie in dieser Arbeit definiert werden.

4 Theorie der Datenflußanalyse paralleler Programme (pDFA)

Dieses Kapitel präsentiert unsere *Theorie der Datenflußanalyse paralleler Programme* (**pDFA**).

Das Hauptproblem der DFA paralleler Programme besteht in der Reduktion der Anzahl der zu untersuchenden Interleavings. Basiert das Analyseproblem auf dem Konstanten-DFA-Rahmen ($\mathcal{D}^{\mathcal{C}}$), können wir zeigen: Es müssen keine Interleavings betrachtet werden! Es reicht für die Analyse aus, die Prozeßrümpfe separat zu betrachten und die DFA-Informationen der einzelnen Prozeßrümpfe geeignet zusammenzufassen. Dazu ist nur ein in der Anzahl der Prozeßrümpfe linearer Aufwand nötig. Für den Booleschen-DFA-Rahmen ($\mathcal{D}^{\mathcal{B}}$) können wir aus den Annahmen Bitvektorgeleichungen abgeleiten, die die DFA-Information einer `PAR`-Anweisung aus denen der Prozeßrümpfe bestimmen.

Die folgenden beiden Fragen leiten dabei die Untersuchungen:

1. welche Information ist nach einer `PAR`-Anweisung gültig und
2. welche Information erreicht eine Anweisung innerhalb eines Prozeßrumpfes.

Diese Aufteilung ermöglicht eine einfache Formulierung der Theorie und der sich daraus ergebenden Algorithmen.

4.1 Die Aufgabe

Wie wir in Abschnitt 3.2 gesehen haben, müssen für die Datenflußanalyse immer alle Programmabläufe untersucht werden. Für die `PAR`-Anweisung bedeutet dies:

1. Die nach einer `PAR`-Anweisung gültige DFA-Information ist durch den *Meet* über alle, dieser `PAR`-Anweisung entsprechenden, Interleavings (*Meet over all Interleavings, MOI*) gegeben:
$$\mathrm{MOI}_{\mathtt{PAR}}(x) = \prod_{p \in \text{Interleavings } (\mathtt{PAR})} f_p(x)$$
2. Die vor einer Anweisung s eines Prozeßrumpfes gültige DFA-Information ist ebenfalls durch alle Interleavings bestimmt. Betrachtet man ein Interleaving p mit $s \in p$, dann ist $f_{p\downarrow s}(x)$ die Information, die vor s in diesem Interleaving gültig ist. Insgesamt also ist die Information, die s erreicht:
$$\mathrm{MOI}_{s \in \mathtt{PAR}}(x) = \prod_{p \in \text{Interleavings } (\mathtt{PAR})} f_{p\downarrow s}(x)$$

Das Problem der Datenflußanalyse für parallele Programme besteht nun darin,
$$\prod_{p \in \text{Interleavings } (\mathtt{PAR})} f_p(x) \text{ bzw. } \prod_{p \in \text{Interleavings } (\mathtt{PAR})} f_{p\downarrow s}(x)$$
zu bestimmen. Da es exponentiell viele Interleavings gibt, kann der Weg nicht darin liegen, alle Interleavings und den Wert der entsprechenden Transferfunktionen explizit zu bestimmen.

4.2 Der Lösungsansatz

Im folgenden skizzieren wir die Lösung dieses Problems, die dann im Rest dieses Kapitels vollständig hergeleitet wird.

Die Hintereinanderausführung von Anweisungen $s_1; \ldots ; s_n$ wird in der DFA durch Funktionskomposition $f_n \circ f_{n-1} \circ \cdots \circ f_1$ der Transferfunktionen f_i der Anweisungen s_i modelliert. $f_n \circ f_{n-1} \circ \cdots \circ f_1$ wird **Kompositionskette** genannt. Die grundlegende Eigenschaft der Funktionen aus $\mathcal{F}^{\mathcal{C}}$ (und damit auch der aus $\mathcal{F}^{\mathcal{B}}$) ist, daß der Wert einer Kompositionskette „fast unabhängig von der Vorgeschichte der Anweisungsfolge“ ist. Mit anderen Worten: nicht alle Anweisungen s_i werden zur Berechnung des Wertes der Kompositionskette $f_n \circ f_{n-1} \circ \cdots \circ f_1(x)$ benötigt. Genauer: entweder sind alle $f_i = \mathsf{id}$, dann gilt: $f_n \circ f_{n-1} \circ \cdots \circ f_1(x) = x$, oder es gibt ein größtes $j \in \{1, \ldots , n\}$ mit $f_j \neq \mathsf{id}$, d.h. $f_j = \mathsf{const}_c$ ist eine Konstantenfunktion (für $k > j$ gilt damit $f_k = \mathsf{id}$). Damit ist aber der Wert $f_j(x) = c$ und unabhängig von x. Dies aber bedeutet, daß alle Anweisungen, die vor s_j stehen, nicht zum Wert der Kompositionskette beitragen! Diese Beobachtung bildet die Grundlage der theoretischen Ergebnisse dieser Arbeit und löst das Hauptproblem: Reduktion der Interleavings.

Nehmen wir an, daß nur zwei Anweisungen s_1 und s_2 parallel ausgeführt werden. Die zugehörigen Transferfunktionen seien f_{s_1} bzw. f_{s_2}. Dann ist:

$$\begin{aligned}\bigsqcap_{p \in \text{Interleavings}(\texttt{PAR } s_1 | s_2 \texttt{ END})} f_p(x) &= f_{s_1;s_2}(x) \sqcap f_{s_2;s_1}(x) \\ &= f_{s_2} \circ f_{s_1}(x) \sqcap f_{s_1} \circ f_{s_2}(x)\end{aligned}$$

Mit obiger Beobachtung können wir dies bestimmen zu:

$$= \begin{cases} \mathsf{id}(x) & \text{wenn } f_{s_1} = f_{s_2} = \mathsf{id} \\ f_{s_1}(x) & \text{wenn } f_{s_1} \neq \mathsf{id}, f_{s_2} = \mathsf{id} \\ f_{s_2}(x) & \text{wenn } f_{s_1} = \mathsf{id}, f_{s_2} \neq \mathsf{id} \\ f_{s_1}(x) \sqcap f_{s_2}(x) & \text{wenn } f_{s_1} \neq \mathsf{id}, f_{s_2} \neq \mathsf{id} \end{cases}$$

Zur einfacheren Notation führen wir neben dem elementweisen *Meet*-Operator ($\sqcap$), für Funktionen einen weiteren ($\boxast$) ein, der diese Fallunterscheidung widerspiegelt. Somit erhalten wir:

$$\bigsqcap_{p \in \text{Interleavings}(\texttt{PAR } s_1 | s_2 \texttt{ END})} f_p(x) = f_{s_1}(x) \boxast f_{s_2}(x) \tag{4.1}$$

Damit haben wir die DFA-Information berechnet, die am Ende der `PAR`-Anweisung gültig ist, ohne die Interleavings betrachten zu müssen!

Für die Berechnung von $\bigsqcap_{p \in \text{Interleavings } (\texttt{PAR})} f_{p \downarrow s}(x)$ können wir, basierend auf diesen Überlegungen, ein ähnlich überraschendes Resultat zeigen: Statt $f_{p \downarrow s}$ für alle Interleavings p zu untersuchen, genügt es, die Anweisungen der parallel ausgeführten Prozeßrümpfe separat zu betrachten und deren Transferfunktionen geeignet zu verknüpfen.

Zur einfacheren Herleitung dieser Resultate nehmen wir zuerst eine sehr eingeschränkte Teilsprache, genannt pPL_0 an (s. Definition 4.20). Sie besteht nur aus Zuweisungen und nicht-geschachtelten `PAR` Anweisungen. Wir zeigen dann, daß Gleichung (4.1) auch für Anweisungsfolgen S_1, S_2, die aus mehr als zwei Anweisungen bestehen gilt (mit entsprechender Definition von $\boxast$, s. Definition 4.5 und Satz 4.23 sowie Korrolar 4.25). Sie gilt auch noch, wenn zwei Mengen $M1 = \{S_1^1, S_1^2, ...\}, M2 = \{S_2^1, S_2^2, ...\}$ von Anweisungensfolgen parallel ausgeführt werden (s. Satz 4.26b). Ausgehend von dieser Verallgemeinerung kann die Einschränkung auf pPL_0 aufgegeben werden, denn für die Datenflußanalyse kann

- eine `IF` ... `THEN S`$_1$ `ELSE S`$_2$ `END` Anweisung aufgefasst werden als Menge von zwei Aunweisungsfolgen $\{S_1, S_2\}$,

- eine `WHILE ... DO S END` Anweisung als Menge von Anweisungesfolgen $\{\epsilon, \quad S, \quad S;S, S;S;S,...\}$ verstanden werden, wobei ϵ der leeren Anweisungsfolge und $S;\cdots;S$ der mehrfachen Hintereinanderausführung des Rumpfes S entspricht, und

- eine, nun geschachtelte, `PAR` Anweisung durch die Menge ihrer Interleavings gekennzeichnet werden.

Satz 4.39 präsentiert die Bitvektor DFA-Gleichungen für die `PAR`-Anweisung, im Stile der Gleichungen für die `IF`- oder `REPEAT`-Anweisung.

Explizite Synchronisation von Prozessen kann als Reihenfolgebedingung an die Anweisungen verschiedener Prozeßrümpfe aufgefaßt werden. Dadurch reduziert sich die Zahl der möglichen Interleavings einer `PAR`-Anweisung. Ignoriert man die Synchronisation werden somit zu viele Interleavings betrachtet. Für die Datenflußanalyse bedeutet dies, daß die Information unschärfer ist. Als Folge davon werden Optimierungen verboten, die eigentlich erlaubt wären (wenn nur die möglichen Interleavings betrachtet würden). In Abschnitt 4.6 werden wir die Resultate auf Programme mit expliziter Synchronisation übertragen.

Der Rest dieses Kapitels wird der formalen Herleitung dieser Idee und den Schlüssen gewidmet, die sich daraus für die DFA paralleler Programme ergeben.

4.3 Eigenschaften des $\mathcal{F}^{\mathcal{C}}$ und $\mathcal{F}^{\mathcal{B}}$ DFA-Rahmens

In diesem Abschnitt werden die Eigenschaften der Funktionen aus $\mathcal{F}^{\mathcal{C}}$ und $\mathcal{F}^{\mathcal{B}}$ vorgestellt, die die Basis für die effiziente und korrekte DFA paralleler Programme sind.

4.3.1 Definition und Eigenschaften des $\sqcap$-Operators und des $\circledast$-Operators

4.1 Definition ($f \sqcap g$)
Seien f und g Funktionen aus $\mathcal{F}$. Die Funktion $h \stackrel{\text{def}}{=} f \sqcap g$ ist gegeben durch:

$$\forall x \in \mathcal{L} : h(x) \stackrel{\text{def}}{=} f(x) \sqcap g(x).$$

Für eine Menge von Funktionen $F = \{f_1, f_2, \dots\}$ mit $f_i \in \mathcal{F}$ wird $\bigsqcap_{f \in F} f$ entsprechend definiert. Dabei soll $\bigsqcap_{f \in \emptyset} f = \mathsf{id}$ gelten. □

4.2 Satz (Abgeschlossenheit von $\mathcal{F}^{\mathcal{B}}$ bezüglich $\sqcap$)
*$\mathcal{F}^{\mathcal{B}}$ ist bezüglich des $\sqcap$-Operators **abgeschlossen**:*

$$\forall f, g \in \mathcal{F}^{\mathcal{B}} : f \sqcap g \in \mathcal{F}^{\mathcal{B}}$$

□

Beweis (Satz 4.2)
Die Behauptung des Satzes folgt offensichtlich aus der Gültigkeit der folgenden Gleichung, die im weiteren gezeigt wird:
Sei F eine Menge von Funktionen aus $\mathcal{F}^{\mathcal{B}}$, dann gilt:

$$\bigsqcap_{f \in F} f = \begin{cases} \mathsf{const}_\perp & \text{wenn } F \neq \emptyset \text{ und } \exists f \in F : f = \mathsf{const}_\perp \\ \mathsf{const}_\top & \text{wenn } F \neq \emptyset \text{ und } \forall f \in F : f = \mathsf{const}_\top \\ \mathsf{id} & \text{sonst} \end{cases} \tag{4.2}$$

Wir müssen die Behauptung elementweise für alle Werte $x \in \mathcal{B}$ zeigen, dazu betrachen wir die einzelnen Fälle der Fallunterscheidung:

Gilt $\exists f \in F : f = \mathsf{const}_\perp$ oder $\forall f \in F : f = \mathsf{const}_\top$ ist Gleichung (4.2) offensichtlich erfüllt (dies entspricht den ersten beiden Zeilen der Fallunterscheidung).

Die dritte Zeile gilt für $F = \emptyset$ per Definition.

Sei nun $F \neq \emptyset$, $\nexists f \in F : f = \mathsf{const}_\bot$ und $\not\forall f \in F : f = \mathsf{const}_\top$, dann muß F mindestens id enthalten und kann noch zusätzlich $\mathsf{const}_\top$ enthalten. Wenn F nur diese beiden Funktion enthält, dann gilt für

$x = \bot \implies \bigsqcap_{f \in F} f(x) = \bot$, da $\mathsf{id} \in F$
$x = \top \implies \bigsqcap_{f \in F} f(x) = \top$, da $f \in \{\mathsf{id}, \mathsf{const}_\top\}$

Und somit $\bigsqcap_{f \in F} f = \mathsf{id}$.

□

4.3 SATZ (NICHTABGESCHLOSSENHEIT VON $\mathcal{F}^{\mathcal{C}}$ BEZÜGLICH $\sqcap$)
$\mathcal{F}^{\mathcal{C}}$ ist bezüglich des Meet-*Operators nicht abgeschlossen genau dann, wenn*

- $(\mathcal{C}, \sqsubseteq, \sqcap)$ *kein Einselement oder*
- $\mathcal{C}$ *mehr als zwei Elemente hat.*

□

BEWEIS (SATZ 4.3)

Wir zeigen, daß die Funktion $h_c \stackrel{\text{def}}{=} \mathsf{const}_c \sqcap \mathsf{id}$ für geeignete $c \in \mathcal{C}$ nicht in $\mathcal{F}^{\mathcal{C}}$ enthalten ist.

- Den Fall eines ein-elementigen Halbverbandes haben wir per Definition ausgeschlossen. Sei $\mathcal{C} = \{\bot, c_1, \dots\}$ und $(\mathcal{C}, \sqsubseteq, \sqcap)$ habe kein Einselement. Sei $c \in \mathcal{C}, c \neq \bot$.

  Da $(\mathcal{C}, \sqsubseteq, \sqcap)$ kein Einselement hat, kann nicht $\forall x \in \mathcal{C} : x \sqcap c = x$ gelten (sonst wäre c ein Einselement). Damit kann h_c nicht die Identitätsfunktion sein. h_c ist auch nicht eine Konstantenfunktion, da $h_c(c) = c$ und $h_c(\bot) = \bot$.

- Da $\mathcal{F}^{\mathcal{C}}$ nicht $\sqcap$-abgeschlossen ist, wenn $(\mathcal{C}, \sqsubseteq, \sqcap)$ kein Einselement hat, nehmen wir an, daß $(\mathcal{C}, \sqsubseteq, \sqcap)$ ein Einselement $(\top)$ hat. Sei $\mathcal{C} = \{\bot, \top, c_1, \dots\}$. Sei $c \in \mathcal{C}, c \notin \{\bot, \top\}$.

  Da $h_c(\bot) = \bot$, $h_c(\top) = c$ und $h_c(c) = c$, kann h_c weder die Identitäts- noch die Konstantenfunktion sein.

Somit liegt h_c in beiden Fällen nicht in $\mathcal{F}^{\mathcal{C}}$. Damit ist $\mathcal{F}^{\mathcal{C}}$ unter beiden Bedingungen nicht $\sqcap$-abgeschlossen.

Für die umgekehrte Beweisrichtung sei $\mathcal{F}^{\mathcal{C}}$ nicht $\sqcap$-abgeschlossen. Wir nehmen an, daß die Behauptung – $(\mathcal{C}, \sqsubseteq, \sqcap)$ hat kein Einselement, oder $\mathcal{C}$ hat mehr als zwei Elemente – nicht gelte. Damit wäre $\mathcal{C} = \mathcal{B}$. Somit führen diese Annahmen mit Satz 4.2 zu einem Widerspruch.

□

Zusammenfassend ergibt sich also:

4.4 KOROLLAR ($^{\dagger 1}$)
Es gibt in $\mathcal{F}^{\mathcal{C}}$ (bis auf Umbenennung) nur genau einen $\sqcap$-abgeschlossenen Funktionenraum: $\mathcal{F}^{\mathcal{B}}$. □

Neben dem elementweisen $\sqcap$-Operator benötigen wir noch einen weiteren *Meet*-Operator ($\boxtimes$) über den Funktionen aus $\mathcal{F}^{\mathcal{C}}$, der nicht elementweise sondern wie folgt definiert ist:

[1] Sätze, Lemmata und Korollare, die mit † gekennzeichnet sind, werden in Anhang A bewiesen.

4.5 Definition ($f \circledast g$)
Seien f und g Funktionen aus $\mathcal{F}^{\mathcal{C}}$. Die Funktion $h \stackrel{\text{def}}{=} f \circledast g$ ist gegeben durch:

$$h \stackrel{\text{def}}{=} \begin{cases} \mathsf{id} & \text{wenn } g = \mathsf{id}, f = \mathsf{id} \\ f & \text{wenn } g = \mathsf{id}, f \neq \mathsf{id} \\ g & \text{wenn } g \neq \mathsf{id}, f = \mathsf{id} \\ \mathsf{const}_{c_f \sqcap c_g} & \text{wenn } g = \mathsf{const}_{c_g}, f = \mathsf{const}_{c_f} \end{cases}$$

Für eine Menge von Funktionen $F = \{f_1, f_2, \dots\}$ mit $f_i \in \mathcal{F}^{\mathcal{C}}$ wird $\boxed{*}_{f \in F} f$ entsprechend definiert:

$$\boxed{*}_{f \in F} f \stackrel{\text{def}}{=} \begin{cases} \mathsf{id} & \text{wenn } \forall f \in F : f = \mathsf{id} \\ \mathsf{const}_c & \text{sonst, wobei } c = \bigsqcap_{f \in F, f = \mathsf{const}_{c_f}} c_f \end{cases}$$

Dabei soll $\boxed{*}_{f \in \emptyset} f = \mathsf{id}$ gelten. □

Aus dieser Definition folgt offensichtlich:

4.6 Satz (Abgeschlossenheit von $\mathcal{F}^{\mathcal{C}}$ bezüglich $\circledast$)
$\mathcal{F}^{\mathcal{C}}$ ist bezüglich des $\circledast$-Operators ***abgeschlossen****:* $\forall f, g \in \mathcal{F}^{\mathcal{C}} : f \circledast g \in \mathcal{F}^{\mathcal{C}}$ □

Für Funktionen aus $\mathcal{F}^{\mathcal{B}}$ ergibt sich folgende einfache Regel zur Berechnung von $\circledast$:

4.7 Lemma
Seien $F = \{f_1, f_2, \dots\}$ mit $f_i \in \mathcal{F}^{\mathcal{B}}$, dann gilt:

$$\boxed{*}_{f \in F} f = \begin{cases} \mathsf{id} & \text{wenn } \forall f \in F : f = \mathsf{id} \\ \mathsf{const}_{\bot} & \text{wenn } \exists f \in F : f = \mathsf{const}_{\bot} \\ \mathsf{const}_{\top} & \text{sonst} \end{cases}$$

□

Anwenden der Definition 3.37 bestätigt:

4.8 Korollar
Aus der Definition von $\circledast$ ergibt sich als Halbordnung $\sqsubseteq^{*}$ der Funktionen aus $\mathcal{F}^{\mathcal{C}}$:

$$\forall f \in \mathcal{F}^{\mathcal{C}} : \mathsf{const}_{\bot} \sqsubseteq^{*} f \sqsubseteq^{*} \mathsf{id}$$

und für $f = \mathsf{const}_{c_f}$ und $g = \mathsf{const}_{c_g}$ gilt

$$f \sqsubseteq^{*} g \iff c_f \sqsubseteq c_g$$

Damit ist $(\mathcal{F}^{\mathcal{C}}, \sqsubseteq^{*}, \circledast)$ ein Halbverband über den Funktionen aus $\mathcal{F}^{\mathcal{C}}$. $\mathsf{const}_{\bot}$ ist das $\bot$-Element und id das $\top$-Element des Halbverbandes. □

4.9 Satz (†Monotonie und Distributivität)
Seien f und g Funktionen aus $\mathcal{F}$. Sind f und g monoton (distributiv), dann ist $h = f \sqcap g$ monoton (distributiv).
Seien f und g Funktionen aus $\mathcal{F}^{\mathcal{C}}$, dann ist $h = f \circledast g$ distributiv und somit auch monoton. □

4.3.2 Zusammenhang zwischen $\sqcap$ und $\circledast$

4.10 Satz (Zusammenhang $\sqcap$ und $\circledast$)
Seien $f, g \in \mathcal{F}^{\mathcal{C}}$, dann gilt: $f \circledast g = f \circ g \sqcap g \circ f$. □

Beweis (Satz 4.10)

Eine Fallunterscheidung ergibt:

$f = g = \mathsf{id}$	offensichtlich.
$f = \mathsf{id}, g = \mathsf{const}_g$	$\mathsf{id} \boxast g = g$ und $g \circ \mathsf{id} = \mathsf{id} \circ g = g$.
$f = \mathsf{const}_f, g = \mathsf{id}$	analog.
$f = \mathsf{const}_f, g = \mathsf{const}_g$	$f \boxast g = f \sqcap g$ und $f \circ g \sqcap g \circ f = f \sqcap g$.

□

Dieser Satz läßt sich mittels folgender Definition auf mehr als zwei Funktionen erweitern.

4.11 Definition (Einfache Permutation)

Die Menge der **einfachen Permutationen** $\mathsf{s_perm}(1, n)$ der Zahlen $1, \ldots, n$ ist gegeben durch:

$$\begin{aligned}\mathsf{s_perm}(1, n) \stackrel{\text{def}}{=} \{&\langle 1, 2, 3, \ldots, n-1, n\rangle,\\ &\langle n, 2, 3, \ldots, n-1, 1\rangle,\\ &\langle 1, n, 3, \ldots, n-1, 2\rangle,\\ &\ldots,\\ &\langle 1, 2, 3, \ldots, n, n-1\rangle\}\end{aligned}$$

Es wird also die i-te Zahl mit der letzten Zahl in der Folge $\langle 1, 2, \ldots, n\rangle$ vertauscht.
$\vec{\imath} \stackrel{\text{def}}{=} \langle i_1, i_2, \ldots, i_n\rangle$ ist ein Element dieser Menge. □

4.12 Bemerkung

Die Menge $\mathsf{s_perm}(1, n)$ enthält nur n Elemente. □

4.13 Korollar

Seien $f_1, \ldots, f_n \in \mathcal{F}^{\mathcal{C}}$, dann gilt:

$$\mathop{\boxast}_{i=1,\ldots,n} f_i = \bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{i_n} \circ \cdots \circ f_{i_1})$$

□

Beweis (Korollar 4.13)

Sind alle $f_i = \mathsf{id}$ gilt die Behauptung offensichtlich.

Es gebe nun also mindestens ein $f_i \neq \mathsf{id}$, dann können für jedes $\vec{\imath} \in \mathsf{s_perm}(1, n)$ in der Funktionskomposition $f_{i_n} \circ \cdots \circ f_{i_1}$ alle f_{i_j} entfernt werden, für die $f_{i_j} = \mathsf{id}$ gilt, ohne den Wert dieser Funktionskomposition zu ändern und jede Funktionskomposition besteht noch aus mindestens einer Funktion. Da nun diese „bereinigten“ Funktionskompositionen nur aus Konstantenfunktionen bestehen, ergibt sich der $\sqcap$ über diese Funktionskompositionen zu $\bigsqcap_{k \in \{1,\ldots,n\}, f_k \neq \mathsf{id}} f_k$ (da jede dieser Funktionen f_k einmal als „erste“ Funktion $(f_k \circ \ldots)$ in einer der „bereinigten“ Funktionskomposition vorkommt). Mit der Definition von $\boxast$ gilt die Behauptung auch in diesem Fall.

□

4.14 Bemerkung (Bedeutung des $\boxast$-Operators)

Informell gesprochen drückt Satz 4.10 aus, daß $\boxast$ der *Meet*-Operator für die `PAR`-Anweisung ist[2]: Sind f und g die Transferfunktionen zweier Anweisungen, die parallel ausgeführt werden sollen, dann ist $f \circ g$ die Transferfunktion des einen Interleavings und $g \circ f$ die des anderen. □

[2] So wie $\sqcap$ der *Meet*-Operator für sequentielle Zusammenlauf ist.

4.15 Bemerkung
Obwohl nicht für alle Funktionen $f, g \in \mathcal{F}^{\mathcal{C}}$ gilt: $f \sqcap g \in \mathcal{F}^{\mathcal{C}}$, gilt dies für $\boxast$: $\forall f, g \in \mathcal{F}^{\mathcal{C}}$: $f \boxast g = f \circ g \sqcap g \circ f \in \mathcal{F}^{\mathcal{C}}$ (Satz 4.6). □

Der nächste Satz und sein Korollar zeigen weitere Zusammenhänge zwischen $\sqcap$ und $\boxast$ auf.

4.16 Satz (†Distributivität zwischen $\sqcap$ und $\boxast$)
Seien $f, g, h \in \mathcal{F}^{\mathcal{B}}$, dann gilt:

$$f \boxast (g \sqcap h) = (f \boxast g) \sqcap (f \boxast h) \qquad \textit{und} \qquad f \sqcap (g \boxast h) = (f \sqcap g) \boxast (f \sqcap h)$$

□

4.17 Korollar
Satz 4.16 gilt i.A. nicht für die Funktionen aus $\mathcal{F}^{\mathcal{C}}$. □

4.3.3 Eigenschaften von Kompositionsketten und Präfixen

Anschaulich besagt der nächste Satz, daß der Wert einer Kompositionskette nur durch die „letzte" Funktion bestimmt ist, die nicht die Identität ist, oder aber alle Funktionen sind die Identität. Diese Eigenschaft der Funktionen aus $\mathcal{F}^{\mathcal{C}}$ ermöglicht die effiziente Datenflußanalyse paralleler Programme!

4.18 Satz (Wert einer Kompositionskette)
Sei $f_1, \cdots, f_n \in \mathcal{F}^{\mathcal{C}}$. Dann gilt:

$$\exists i \in \{1, \ldots, n\} : f_n \circ \cdots \circ f_1 = f_i \textit{ und } \forall j \in \{i+1, \ldots, n\} : f_j = \mathsf{id}$$

□

Beweis (Satz 4.18)
Mittels Induktion über die Länge der Kompositionskette.

$n = 1$ Offensichtlich.

$n \to n+1$ Mit der Induktionsvoraussetzung gilt $f_n \circ \cdots \circ f_1(x) = f_i(x), i \leq n$ und $\forall i < j \leq n : f_j = \mathsf{id}$.
Sei $f_{n+1} = \mathsf{const}$ eine Konstantenfunktion, dann gilt offensichtlich $f_{n+1} \circ \cdots \circ f_1(x) = \mathsf{const} = f_{n+1}$.
Sei nun $f_{n+1} = \mathsf{id}$ die Identitätsfunktion, dann ist $f_{n+1} \circ \cdots \circ f_1(x) = f_i(x)$ und $\forall i < j \leq n = 1 : f_j = \mathsf{id}$.

□

Der nächste Satz sagt etwas über den Wert, der mit dem $\sqcap$-Operator verbundenen Kompositionsketten aus. Die betrachteten Kompositionsketten haben dabei eine ganz bestimmte Form: Die Werte aller „Präfixe" einer Kompositionskette sollen mit $\sqcap$ verknüpft werden. Dieser Satz bildet die Basis zur Beantwortung der Frage, welche DFA-Information innerhalb einer `PAR`-Anweisung vor einer Anweisung s gültig ist.

4.19 Satz (Meet von Präfixen)
Seien $f_1, \ldots, f_n \in \mathcal{F}^{\mathcal{C}}$. Dann gilt für beliebige $x \in \mathcal{C}$:

$$x \;\sqcap\; f_1(x) \;\sqcap\; f_2 \circ f_1(x) \;\sqcap\; \ldots \;\sqcap\; f_n \circ f_{n-1} \cdots \circ f_1(x) = x \sqcap \bigsqcap_{i \in \{1, \ldots, n\}} f_i(x)$$

□

Beweis (Satz 4.19)

Wir zeigen für $n = 2$: $x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) \sqcap g(x)$.
Mit einer Fallunterscheidung nach der Art der Funktion g:
Wenn $g = \mathsf{const}$ dann: $x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) \sqcap g(x)$.
Wenn $g = \mathsf{id}$ dann: $x \sqcap f(x) \sqcap g(f(x)) = x \sqcap f(x) = x \sqcap f(x) \sqcap g(x)$.

Mittels Induktion über die Länge der Kompositionskette erhalten wir die Behauptung des Satzes.

□

4.4 DFA für $\boldsymbol{pPL}_0$ Programme

4.20 Definition (pPL_0)
Ein $\boldsymbol{pPL}_0$ Programm besteht nur aus Zuweisungen und `PAR` Anweisungen. Die Prozeßrümpfe dürfen dabei repliziert sein, aber sie können nur Zuweisungen enthalten.

```
Stmts       ::= (ParStmt | AssignStmt)//";" .
ParStmt     ::= PAR ProcessBody //"|" END .
ProcessBody ::= [Replicator] (AssignStmt //";") .
```

Da es uns nicht auf die konkrete Syntax ankommt, kürzen wir replizierte Prozeßrümpfe `[`*`var`* `:`*`LowerBound`* `TO` *`UpperBound`* `] S` mit k `:S` ab, wobei $k = \mathit{UpperBound} - \mathit{LowerBound} + 1$. k muß keine Konstante zur Übersetzungszeit sein. Wir nehmen im folgenden mit Transformation 3.15 immer $k \geq 1$ an. Ist ein Prozeßrumpf nicht repliziert, entspricht dies `1:S`. Wir können somit o.B.d.A. immer replizierte Prozeßrümpfe annehmen. □

Wir betrachten im folgenden nur pPL_0 Programme, von denen wir o.B.d.A. annehmen, daß die Zuweisungen in Drei-Adreß-Form übersetzt sind. Zunächst bestimmen wir die Information, die am Ende einer `PAR`-Anweisung gültig ist. Im folgenden Abschnitt wird dann für eine Anweisung innerhalb eines Prozeßrumpfes hergeleitet, von welcher DFA-Information sie erreicht wird.

4.4.1 Welche Information ist am Ende einer PAR-Anweisung gültig

Wir berechnen die Transferfunktion, die der Menge aller Interleavings von zwei parallel ausgeführten Anweisungsfolgen p_1 und p_2 entspricht. Diese Menge ist durch $\mathsf{TopSorts}(p_1, p_2)$ gegeben (s. Abschnitt 3.2.4). Mit der folgenden Definition gilt deshalb der erste Satz 4.22.

4.21 Definition ($f_p, f_{p;q}, f_P, f_{P;Q}$)
Sei $p = \langle s_1; \ldots; s_n\rangle$ eine Folge von Anweisungen und $f_{s_i} \in \mathcal{F}^{\mathcal{C}}$ die Transferfunktion, die der Anweisung s_i zugeordnet ist. $f_p \stackrel{\text{def}}{=} f_{s_n} \circ f_{s_{n-1}} \circ \cdots \circ f_{s_1}$ ist die Transferfunktion, die der Anweisungsfolge p zugeordnet ist. $f_{\langle\rangle} \stackrel{\text{def}}{=} \mathsf{id}$, entsprechend für die leere Anweisungsfolge $\langle\rangle$. Sind $p_1, \ldots p_n$ Anweisungsfolgen, dann ist die der Hintereinanderausführung $p_1; \ldots; p_n$ der p_i entsprechende Transferfunktion definiert als: $f_{p_1;\ldots;p_n} \stackrel{\text{def}}{=} f_{p_n} \circ \cdots \circ f_{p_1}$. Sind $P, P_1, \ldots, P_n$ Mengen von Anweisungsfolgen, dann ist $f_P(x) \stackrel{\text{def}}{=} \bigsqcap_{p \in P} f_p(x)$ und entsprechend $f_{P_1;\ldots;P_n}(x) \stackrel{\text{def}}{=} \bigsqcap_{p_1 \in P_1, \ldots, p_n \in P_n} f_{p_1;\ldots;p_n}(x)$. □

4.22 Satz ($\mathrm{MOI}_{\mathtt{PAR}} = \mathsf{TopSorts}(p_1, \ldots, p_n)$)
Gegeben sei ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen. Seien $p_1, \ldots, p_n$ Anweisungsfolgen und $k_1, \ldots, k_n \in \mathbb{N}$. Für die pPL_0 Anweisung `PAR` $k_1 : p_1$ `|...|` $k_n : p_n$ `END` und alle $x \in \mathcal{C}$ gilt:

$$f_{\mathtt{PAR}\ k_1:p_1\ |\ldots|\ k_n:p_n\ \mathtt{END}}(x) = \bigsqcap_{p \in \mathsf{TopSorts}(k_1:p_1, \ldots, k_n:p_n)} f_p(x)$$

$f_{\texttt{PAR}\ k_1:p_1\ |\ldots|\ k_n:p_n\ \texttt{END}}(x)$ ist die DFA-Information, die am Ende der `PAR`*-Anweisung gültig ist, wenn vor ihr x gilt.* □

Die nächsten Sätze und Korollare zeigen, daß und wie die Transferfunktion bzw. der Wert der Transferfunktion einer (replizierten) `PAR`-Anweisung bestimmt werden kann, *ohne Interleavings zu betrachten*!

4.23 Satz (MOI$_{\texttt{PAR}}$)
Gegeben ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen. Seien $p_1 = \langle s'_1; \ldots; s'_{n'}\rangle$ und $p_2 = \langle s''_1; \ldots; s''_{n''}\rangle$ zwei Folgen von Drei-Adreßanweisungen. Seien $k_1, k_2 \in \mathbb{N}$. Dann gilt:

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:p_1, k_2:p_2)} f_p = \begin{cases} \mathsf{id} & \text{wenn } f_{p_1} = f_{p_2} = \mathsf{id} \\ f_{p_1} \sqcap f_{p_2} & \text{wenn } f_{p_1} = \mathsf{const}_{c_1} \text{ und } f_{p_1} = \mathsf{const}_{c_2} \\ f_{p_1} & \text{wenn } f_{p_1} = \mathsf{const}_{c_1} \text{ und } f_{p_2} = \mathsf{id} \\ f_{p_2} & \text{wenn } f_{p_2} = \mathsf{const}_{c_2} \text{ und } f_{p_1} = \mathsf{id} \end{cases}$$

□

Beweis (Satz 4.23)
Zuerst nehmen wir $k_1 = k_2 = 1$ an. Gemäß der Fallunterscheidung zeigen wir:

1. Wenn $f_{p_1} = f_{p_2} = \mathsf{id}$, dann muß für alle Anweisungen $s \in p_1$ bzw. $s \in p_2$ gelten, daß $f_s = \mathsf{id}$. Damit gilt $f_p = \mathsf{id}$ für alle $p \in \mathsf{TopSorts}(p_1, p_2)$.
2. Sind sowohl f_{p_1}, als auch f_{p_2} eine Konstantenfunktion, dann gibt es mit Satz 4.18 Anweisungen $s' \in p_1$ und $s'' \in p_2$, so daß $f_{p_1} = f_{s'}$ bzw. $f_{p_2} = f_{s''}$. Weiterhin gilt für alle Anweisungen s, die in p_1 auf s' folgen ($s' \ll_{p_1} s$), daß $f_s = \mathsf{id}$ (ebenso für s'' und p_2). Jedes $p \in \mathsf{TopSorts}(p_1, p_2)$ enthält alle Anweisungen aus p_1 und p_2. Damit kann nun $s' \ll_p s''$ oder $s'' \ll_p s'$ gelten. Nehmen wir $s' \ll_p s''$ an, dann gilt für alle Anweisungen $s \in p$ mit $s'' \ll_p s$, daß $f_s = \mathsf{id}$. Damit: $f_p = f_{s''}$. Im anderen Fall ($s'' \ll_p s'$) gilt $f_p = f_{s'}$. Da $\mathsf{TopSorts}(p_1, p_2)$ Anweisungsfolgen mit beiden Anordnungsmöglichkeiten von s' und s'' enthält, folgt die Behauptung.
3. Sei o.B.d.A $f_{p_1} = \mathsf{id}$ und $f_{p_2} \neq \mathsf{id}$. Wieder muß für alle $s \in p_1$ gelten: $f_s = \mathsf{id}$. Für alle $p \in \mathsf{TopSorts}(p_1, p_2)$ gilt somit $f_p = f_{p_2}$.

Offensichtlich ist es für den Beweis ohne Belang, wie oft die Anweisungen aus p_i in einem Interleaving p vorkommen. Damit gilt die Behauptung auch für $k_i \geq 1$.

□

Als Erweiterung auf n nebenläufig ausgeführte Anweisungsfolgen ergibt sich:

4.24 Korollar (†)
Gegeben sei ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen. Seien $p_1, \ldots, p_n$ Anweisungsfolgen und $k_1, \ldots, k_n \in \mathbb{N}$. Jede Anweisungsfolge p_i ist Element einer der beiden Mengen $\mathcal{CONST}$ und $\mathcal{ID}$, die wie folgt definiert sind:

$$\mathcal{CONST} \stackrel{\text{def}}{=} \{p_i \mid f_{p_i} = \mathsf{const}\}$$

$$\mathcal{ID} \stackrel{\text{def}}{=} \{p_i \mid f_{p_i} = \mathsf{id}\}$$

Es gilt:

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:p_1, \ldots, k_n:p_n)} f_p = \bigsqcap_{p \in \mathcal{CONST}} f_p$$

□

Mit der Definition von ⊠ (Def 4.5) können wir somit Bemerkung 4.14 für Anweisungsfolgen bestätigen:

4.25 Korollar

Gegeben sei ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen. Seien $p_1, \ldots, p_n$ Anweisungsfolgen und $k_1, \ldots, k_n \in \mathbb{N}$. Es gilt:

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)} f_p = f_{p_1} \boxast f_{p_2} \boxast \ldots \boxast f_{p_n}$$

□

Sowohl für eine Verallgemeinerung der DFA auf beliebige *pPL*-Programme, als auch zur Bestimmung der DFA-Information, die eine Anweisung innerhalb eines Prozeßrumpfes erreicht, müssen wir Mengen von Anweisungsfolgen betrachten, die nebenläufig ausgeführt werden. Wir benötigen somit den folgenden Satz:

4.26 Satz (MOI$_{\mathtt{PAR}}$ für Mengen von Anweisungsfolgen)

Seien $P_1 = \{p_1^1, p_1^2, \ldots\}, P_2 = \{p_2^1, p_2^2, \ldots\}$ Mengen von Anweisungsfolgen, $k_1, k_2 \in \mathbb{N}$.

a) Gegeben ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen, *dann gilt:*

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:P_1,k_2:P_2)} f_p = f_{P_1} \circ f_{P_2} \sqcap f_{P_2} \circ f_{P_1}$$

b) Gegeben ein $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen, *dann gilt:*

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:P_1,k_2:P_2)} f_p = f_{P_1} \boxast f_{P_2}$$

□

Beweis (Satz 4.26)

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:P_1,k_2:P_2)} f_p = \bigsqcap_{p_1 \in P_1, p_s \in P_2} \Big(\bigsqcap_{p \in \mathsf{TopSorts}(k_1:p_1,k_2:p_2)} f_p \Big)$$

Mit Korollar 4.25 und Satz 4.10

$$= \bigsqcap_{p_1 \in P_1, p_s \in P_2} (f_{p_1} \circ f_{p_2} \sqcap f_{p_2} \circ f_{p_1})$$

$$= \bigsqcap_{p_1 \in P_1, p_s \in P_2} (f_{p_1} \circ f_{p_2}) \quad \sqcap \quad \bigsqcap_{p_1 \in P_1, p_s \in P_2} (f_{p_2} \circ f_{p_1})$$

Da die Funktionen aus $\mathcal{F}^{\mathcal{C}}$ distributiv sind (Lemma 3.47)

$$= \bigsqcap_{p_1 \in P_1} (f_{p_1}(\bigsqcap_{p_2 \in P_2} f_{p_2})) \quad \sqcap \quad \bigsqcap_{p_2 \in P_2} (f_{p_2}(\bigsqcap_{p_1 \in P_1} f_{p_1}))$$

Mit Definition 4.21

$$= f_{P_1} \circ f_{P_2} \sqcap f_{P_2} \circ f_{P_1}$$

Sind die Funktionen $f \in \mathcal{F}^{\mathcal{B}}$, dann sind f_{P_1} und f_{P_2} ebenfalls aus $\mathcal{F}^{\mathcal{B}}$ und somit auch aus $\mathcal{F}^{\mathcal{C}}$ (da die Funktionen aus $\mathcal{F}^{\mathcal{B}}$ bezüglich $\sqcap$ abgeschlossen sind, Satz 4.2). Damit darf Satz 4.10 angewendet werden:

$$= f_{P_1} \boxast f_{P_2}$$

Der letzte Schritt gilt nur, wenn ein $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen zugrunde liegt. Denn: liegt ein allgemeiner $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen zugrunde, gilt $f_{P_1} \in \mathcal{F}^{\mathcal{C}}$ im allgemeinen nicht mehr (s. Satz 4.3 und Korollar 4.4.

□

Dieser Satz läßt sich leicht auf mehr als zwei Sequenzen erweitern.

4.27 Korollar (†)

Seien $P_1 = \{p_1^1, p_1^2, \ldots\}, \ldots, P_n = \{p_n^1, p_n^2, \ldots\}$ Mengen von Anweisungsfolgen, $k_1, \ldots, k_n \in \mathbb{N}$.

a) Gegeben ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen, *dann gilt:*

$$\prod_{p \in \mathsf{TopSorts}(k_1:P_1,\ldots,k_n:P_n)} f_p = \prod_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{P_{i_n}} \circ \cdots \circ f_{P_{i_1}})$$

b) Gegeben ein $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen, *dann gilt:*

$$\prod_{p \in \mathsf{TopSorts}(k_1:P_1,\ldots,k_n:P_n)} f_p = f_{P_1} \boxtimes \ldots \boxtimes f_{P_n}$$

□

4.28 Bemerkung
Jede andere Teilmenge der Menge aller Permutationen der Zahlen $1, \ldots, n$ würde den gleichen Zweck erfüllen, solange jede Ausführungssequenz mindestens einmal als letztes Element einer Permutation vorkommt.

□

4.4.2 Welche DFA-Information erreicht eine Anweisung innerhalb eines Prozeßrumpfes

Dieser Abschnitt beantwortet die Frage, welche DFA-Information vor einer Anweisung s gültig ist. Im sequentiellen Fall ist die Antwort offensichtlich: in^i_s – die Information welche vor s gültig ist – ist diejenige, die am Ende der unmittelbar (textuell) vorausgehenden Anweisung gültig war. Gibt es mehrere Vorgänger, von denen zur Übersetzungszeit nicht entschieden werden kann, welche zur Laufzeit ausgeführt wird, wird der *Meet* über die „Ausgangsinformation" aller direkten Vorgänger (im Kontrollflußgraphen) berechnet und benutzt.

In parallelen Programmen kann so aus der textuellen Reihenfolge nicht mehr auf die Ausführungsreihenfolge zur Laufzeit geschlossen werden. Deshalb betrachten wir wieder die Menge der Interleavings. Die Information, welche die Anweisung s innerhalb eines bestimmten Interleavings p erreicht, ist durch die in p der Anweisung s vorausgehenden Anweisungen $p \downarrow s$ gegeben: es muß also $\prod_{p \in \text{Interleavings}} f_{p\downarrow s}(x)$ berechnet werden:

4.29 Satz ($\mathrm{MOI}_{s\in\mathtt{PAR}}$)
Gegeben sei ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen. Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung mit $s \in p \in P$ und $k_1, \ldots, k_n \in \mathbb{N}$. Für die pPL_0 Anweisung ***PAR*** $\boldsymbol{k}_1$: $\boldsymbol{p}_1$ | ... | $\boldsymbol{k}_n$: $\boldsymbol{p}_n$ ***END*** *ist*

$$MOI_{s\in\mathtt{PAR}} = f_{\{p\downarrow s \;|\; p \in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)\}}(x)$$

die DFA-Information, die vor s gültig ist. □

Zur Berechnung der DFA-Information, die s erreicht, ist allerdings eine andere Darstellung von der Menge der s-Anfänge von Vorteil. Wir definieren dazu:

4.30 Definition (prefixes)
Für eine gegebene Anweisungsfolge p ist $\mathsf{prefixes}(p)$ als die Menge aller **Präfixe** von p definiert. Sie enthält sowohl die leere Sequenz, als auch ganz p.
Formal: $\mathsf{prefixes}(p) \stackrel{\text{def}}{=} \{p\} \cup \bigcup_{s \in p} p \downarrow s$. Für eine Menge von Pfaden P wird $\mathsf{prefixes}(P)$ definiert durch: $\mathsf{prefixes}(P) \stackrel{\text{def}}{=} \bigcup_{p \in P} \mathsf{prefixes}(p)$. □

Anschaulich bedeutet das nächste Lemma, daß in einem Interleaving $p \in \mathsf{TopSorts}(p_1, \ldots, p_n)$ vor der Anweisung $s \in p_i$ die Anweisungen aus den „teilweise ausgeführten" Prozeßrümpfen vorkommen, die parallel zu s ausgeführt werden.

4.31 LEMMA (†)
Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung, die in einem p_i vorkommt und $k_1, \ldots, k_n \in \mathbb{N}$. Für die pPL_0 Anweisung **PAR** $k_1 : p_1$ | ... | $k_n : p_n$ **END** *gilt:*

$$\{p \downarrow s \mid p \in \mathsf{TopSorts}(k_1 : p_1, \ldots, k_n : p_n)\} = \mathsf{TopSorts}(k_1 : \mathsf{prefixes}(p_1), \ldots, k_{i-1} : \mathsf{prefixes}(p_{i-1}), (k_i - 1) : \mathsf{prefixes}(p_i),\ p_i \downarrow s, k_{i+1} : \mathsf{prefixes}(p_{i+1}), \ldots k_n : \mathsf{prefixes}(p_n))$$

□

Folgendes Beispiel soll dies verdeutlichen:

4.32 BEISPIEL
Gegeben sei die **PAR**$s_1; s_2; \ldots; s_n | s$ **END** Anweisung. Entweder wird s als erste Anweisung: $s; s_1; s_2; \ldots; s_n$, als zweite Anweisung: $s_1; s; s_2; \ldots; s_n$, als dritte: $s_1; s_2; s; s_3 \ldots; s_n$, etc. oder als letzte Anweisung: $s_1; s_2; \ldots; s_n; s$ ausgeführt. Damit „erreichen“ also die Sequenzen $\{\langle\rangle, \langle s_1\rangle, \langle s_1; s_2\rangle, \ldots, \langle s_1; s_2; \ldots; s_n\rangle\}$ die Anweisung s. □

Führen wir die Menge der zu einer Anweisung s parallel ausgeführten Anweisungen $\mathsf{sibl}[s]$ ein, können die folgenden Resultate einfacher formuliert werden[3].

4.33 DEFINITION (sibl)
Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung mit $s \in p_i \in P$ und $k_1, \ldots, k_n \in \mathbb{N}$. $\mathsf{sibl}[s]$ ist die Menge aller Anweisungen, die in der pPL_0 Anweisung **PAR** $k_1 : p_1$ | ... | $k_n : p_n$ **END** parallel zu s ausgeführt werden können:

$$\mathsf{sibl}[s] \stackrel{\text{def}}{=} \begin{cases} \bigcup\limits_{\substack{t \in p_j \\ 1 \le j \ne i \le n}} t & \text{wenn } k_i = 1 \\ \bigcup\limits_{\substack{t \in p_j \\ 1 \le j \le n}} t & \text{wenn } k_i > 1 \end{cases}$$

□

4.34 BEMERKUNG
Die Eigenschaft $s' \in \mathsf{sibl}[s]$ von s' ist eine rein syntaktische, die leicht aus dem Quellprogramm extrahiert werden kann. Ist ein Prozeßrumpf repliziert, sind seine Anweisungen nur einmal in sibl enthalten. □

Mit der Charakterisierung der Menge der Anweisungssequenzen von Lemma 4.31, die s-Anfänge sind, kann die DFA-Information berechnet werden, die s erreicht.

4.35 SATZ ($\mathrm{MOI}_{s \in \mathtt{PAR}}$)
Gegeben sei ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen. Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung mit $s \in p_i \in P$ und $k_1, \ldots, k_n \in \mathbb{N}$. Für die pPL_0 Anweisung **PAR** $k_1 : p_1$ | ... | $k_n : p_n$ **END** *gilt:*

$$f_{\{p \downarrow s \mid p \in \mathsf{TopSorts}(k_1 : p_1, \ldots, k_n : p_n)\}} = f_{p_i \downarrow s} \sqcap \bigsqcap_{\substack{t \in \mathsf{sibl}[s] \\ f_t = \mathsf{const}}} f_t$$

□

Zum Beweis benötigen wir den auf Satz 4.19 basierenden Hilfssatz.

[3] sibl von engl.: *sibling*, Geschwister.

4.36 Lemma (†)
Unter den Voraussetzungen des Satzes 4.35 gilt:

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:\mathsf{prefixes}(p_1),\ldots,k_n:\mathsf{prefixes}(p_n))} f_p = \mathsf{id} \sqcap \bigsqcap_{\substack{t \in p \in \{p_1,\ldots,p_n\} \\ f_t = \mathsf{const}}} f_t$$

□

Beweis (Satz 4.35)

Sei $s \in p_i$. Wir nehmen zuerst $k_i = 1$ an. Sei

$$T \stackrel{\text{def}}{=} \mathsf{TopSorts}(k_1 : \mathsf{prefixes}(p_1), \ldots, k_{i-1} : \mathsf{prefixes}(p_{i-1}), k_{i+1} : \mathsf{prefixes}(p_{i+1}), \ldots, k_n : \mathsf{prefixes}(p_n)).$$

Mit Lemma 4.31 und der Assoziativität von TopSorts gilt:

$$\{p \downarrow s \mid p \in \mathsf{TopSorts}(k_1 : p_1, \ldots, k_n : p_n)\}$$
$$= \mathsf{TopSorts}(k_1{:}\mathsf{prefixes}(p_1),\ldots,k_{i-1}{:}\mathsf{prefixes}(p_{i-1}),p_i{\downarrow}s,k_{i+1}{:}\mathsf{prefixes}(p_{i+1}),\ldots,k_n{:}\mathsf{prefixes}(p_n))$$
$$= \mathsf{TopSorts}(T, p_i \downarrow s)$$

Mit Satz 4.26:

$$f_{\mathsf{TopSorts}(T,p_i\downarrow s)}(x) = f_{p_i\downarrow s} \circ f_T(x) \quad \sqcap \quad f_T \circ f_{p_i\downarrow s}$$

Mit Lemma 4.36:

$$= f_{p_i\downarrow s} \circ (\mathsf{id} \sqcap \bigsqcap_{\substack{t \in p \in T \\ f_t = \mathsf{const}}} f_t) \quad \sqcap \quad f_{p_i\downarrow s} \sqcap \bigsqcap_{\substack{t \in p \in T \\ f_t = \mathsf{const}}} (f_t \circ f_{p_i\downarrow s})$$

Da $p_i \downarrow s$ nur ein Element enthält und $\mathcal{F}^{\mathcal{C}}$ bezüglich der Funktionskomposition abgeschlossen ist, ist $f_{p_i\downarrow s}$ entweder die Identität, oder eine Konstantenfunktion. Deshalb:

$$= f_{p_i\downarrow s} \sqcap \bigsqcap_{\substack{t \in p_j, 1 \le j \ne i \le n \\ f_t = \mathsf{const}}} f_t$$

Ist p_i repliziert, d.h. $k_i > 1$, können $k_i - 1$ Prozesse parallel zu s ablaufen, die alle p_i ausführen. Somit muß obige Definition von T durch

$$T \stackrel{\text{def}}{=} \mathsf{TopSorts}(k_1 : \mathsf{prefixes}(p_1), \ldots, k_{i-1} : \mathsf{prefixes}(p_{i-1}), k_i - 1 : \mathsf{prefixes}(p_i), \ldots, k_{i+1} : \mathsf{prefixes}(p_{i+1}) \ldots, k_n : \mathsf{prefixes}(p_n))$$

ersetzt werden. Damit entfällt die Bedingung $j \neq i$, die durch die Annahme $k_i = 1$ eingeführt wurde (Beachte: $p \downarrow s$ hat immer noch nur ein Element). Damit gilt:

$$f_{\{p\downarrow s \mid p \in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)\}} = f_{p_i\downarrow s} \sqcap \begin{cases} \bigsqcap_{\substack{t \in p_j, 1 \le j \ne i \le n \\ f_t = \mathsf{const}}} f_t & \text{wenn } k_i = 1 \\ \bigsqcap_{\substack{t \in p_j, 1 \le j \le n \\ f_t = \mathsf{const}}} f_t & \text{wenn } k_i > 1 \end{cases}$$

Zusammen mit der Definition von sibl ergibt sich die Behauptung.

□

Da in Satz 4.35 im Term $\bigsqcap_{t \in \mathsf{sibl}[s], f_t = \mathsf{const}} f_t(x)$ die Funktionen f_t konstant sind, kann dieser Term für den $\sqcap^{\preceq}$ bzw. $\sqcap^{\cup/\cap}$-Operator durch eine einfache Fallunterscheidung berechnet werden. Wir geben diese nur für den booleschen DFA-Rahmen an:

4.37 Korollar (†)
Gegeben sei ein $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen. Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung mit $s \in p_i \in P$ und $k_1, \ldots, k_n \in \mathbb{N}$. Für die pPL_0 Anweisung **PAR** $\boldsymbol{k}_1$: $\boldsymbol{p}_1$ | ... | $\boldsymbol{k}_n$: $\boldsymbol{p}_n$ **END** *gilt:*

$$f_{\{p\downarrow s \mid p\in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)\}}(x) = \begin{cases} \mathsf{const}_\bot & \textit{wenn } \exists t \in \mathsf{sibl}[s] : f_t = \mathsf{const}_\bot \\ f_{p_i\downarrow s} & \textit{sonst} \end{cases}$$

□

Wir führen die Notationen $\mathsf{in}^;$ und $\mathsf{in}^{||}$ ein, anschaulich bedeuten sie: Für die Anweisung s repräsentiert $\mathsf{in}^;_s$ die DFA-Information, die s auf einem „sequentiellen“ Ausführungspfad erreicht, d.h. keine der zu s parallelen Anweisungen wird vor s ausgeführt. $\mathsf{in}^{||}_s$ ist die Information, die s erreicht, wenn alle möglichen Ausführungspfade betrachtet werden.

4.38 Definition ($\mathsf{in}^;, \mathsf{in}^{||}$)
Gegeben sei ein $\mathcal{D}^\mathcal{C}$ DFA-Rahmen. Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung mit $s \in p_i \in P$ und $k_1, \ldots, k_n \in \mathbb{N}$. Für die pPL_0 Anweisung `PAR` $\boldsymbol{k}_1$: $\boldsymbol{p}_1$ | ... | $\boldsymbol{k}_n$: $\boldsymbol{p}_n$ `END` und die Anweisung s definieren wir:

$$\mathsf{in}^;_s(x) \stackrel{\text{def}}{=} f_{p_i\downarrow s}(x)$$

$$\mathsf{in}^{||}_s(x) \stackrel{\text{def}}{=} f_{\{p\downarrow s \mid p\in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)\}}(x)$$

Gegeben sei ein $\mathcal{D}^\mathcal{B}$ DFA-Rahmen. Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung mit $s \in p_i \in P$ und $k_1, \ldots, k_n \in \mathbb{N}$. Für die pPL_0 Anweisung `PAR` $\boldsymbol{k}_1$: $\boldsymbol{p}_1$ | ... | $\boldsymbol{k}_n$: $\boldsymbol{p}_n$ `END` und die Anweisung s sind die Bitvektoren $\mathsf{in}^;_s$ und $\mathsf{in}^{||}_s$ für die Objektnummer o wie folgt definiert:

$$\mathsf{in}^;_s[o] \stackrel{\text{def}}{=} \mathsf{TRUE} \iff f^o_{p_i\downarrow s}(x) = \top$$

$$\mathsf{in}^{||}_s[o] \stackrel{\text{def}}{=} \mathsf{TRUE} \iff f^o_{\{p\downarrow s \mid p\in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)\}}(x) = \top$$

Dabei ist x die DFA-Information, die vor der `PAR`-Anweisung gültig ist. □

Zum Schluß fassen wir unsere bisherigen Ergebnisse zusammen und formulieren für den $\mathcal{D}^\mathcal{B}$ DFA-Rahmen die Gleichungssysteme, welche die Basis der Datenflußanalyse paralleler Programme bilden.

4.39 Satz (†Bitvektor-Gleichungen für pPL_0 Programme)
Gegeben sei ein $\mathcal{D}^\mathcal{B}$ DFA-Rahmen. Sei $P = \{p_1, \ldots, p_n\}$ eine Menge von Anweisungsfolgen, s eine Anweisung mit $s \in p \in P$ und $k_1, \ldots, k_n \in \mathbb{N}$. Seien $\mathsf{gen}_{p_i}, \mathsf{kill}_{p_i}$ und transp_{p_i}, die gemäß den Gleichungen (3.5) für die Hintereinanderausführung von Anweisungen bestimmten Mengen für p_i. gen_s und kill_s sind gemäß Definition 3.54 für die Anweisung s bestimmten Mengen. $\mathsf{in}^;_s$ kann für jede Anweisung s gemäß den Gleichungen (3.5) für die Hintereinanderausführung von Anweisungen berechnet werden. $\mathsf{in}^;_{\texttt{PAR}}$ ($\mathsf{in}^;_{p_i}$) repräsentiert die DFA-Information, die vor der `PAR` Anweisung (vor der ersten Anweisung des Prozeßrumpfes p_i) gültig ist[4]. Es gilt:
Für Muß-Vorwärts-DFA-Probleme:

$$\mathsf{in}^{||}_s = \mathsf{in}^;_s - \bigcup_{t\in \mathsf{sibl}[s]} \mathsf{kill}_t$$

$$\mathsf{gen}_{\texttt{PAR}\ k_1:p_1\ |\ldots|\ k_n:p_n\ \texttt{END}} = \bigcup_{i\in\{1,\ldots,n\}} \mathsf{gen}_{p_i} - \bigcup_{i\in\{1,\ldots,n\}} \mathsf{kill}_{p_i}$$

$$\mathsf{kill}_{\texttt{PAR}\ k_1:p_1\ |\ldots|\ k_n:p_n\ \texttt{END}} = \bigcup_{i\in\{1,\ldots,n\}} \mathsf{kill}_{p_i}$$

[4] $X_{\texttt{PAR}}$ ist die Abkürzung für $X_{\texttt{PAR}\ k_1:p_1\ |\ldots|\ k_n:p_n\ \texttt{END}}$.

Für Kann-Vorwärts-DFA-Probleme:

$$\mathsf{in}^{||}_s = \mathsf{in}^{;}_s \cup \bigcup_{t \in \mathsf{sibl}[s]} \mathsf{gen}_t$$

$$\mathsf{gen}_{PAR\ k_1:p_1\ |\ldots|\ k_n:p_n\ END} = \bigcup_{i \in \{1,\ldots,n\}} \mathsf{gen}_{p_i}$$

$$\mathsf{kill}_{PAR\ k_1:p_1\ |\ldots|\ k_n:p_n\ END} = \bigcup_{i \in \{1,\ldots,n\}} \mathsf{kill}_{p_i} - \bigcup_{i \in \{1,\ldots,n\}} \mathsf{gen}_{p_i}$$

Für beide Problemarten:

$$\mathsf{transp}_{PAR\ k_1:p_1\ |\ldots|\ k_n:p_n\ END} = \bigcap_{i \in \{1,\ldots,n\}} \mathsf{transp}_{p_i}$$

$$\mathsf{in}^{;}_{p_i} = \mathsf{in}^{;}_{PAR}$$

$$\mathsf{out}_{PAR\ k_1:p_1\ |\ldots|\ k_n:p_n\ END} = \mathsf{gen}_{PAR} \cup \mathsf{in}_{PAR} - \mathsf{kill}_{PAR} = \mathsf{gen}_{PAR} \cup (\mathsf{in}_{PAR} \cap \mathsf{transp}_{PAR})$$

□

4.5 DFA für *pPL* Programme

Bisher durften Prozeßrümpfe nur aus Zuweisungen bestehen. Diese Annahme wurde getroffen, um unsere Resultate leichter herleiten zu können. In diesem Abschnitt werden wir diese Beschränkung fallen lassen, und die Theorie der Datenflußanalyse paralleler Programme auf beliebige parallele Programme (genauer: beliebige *pPL*-Programme) erweitern. Grundlage dazu bilden, wie in Abschnitt 4.2 beschrieben, die Anweisungsmegen, die einer `IF`, `WHILE` oder `PAR` Anweisung entsprechen.

4.5.1 Welche Information ist am Ende einer `PAR`-Anweisung gültig

Die Grundlage zur Berechnung der DFA-Information, die am Ende einer `PAR`-Anweisung gültig ist, bildet der Satz 4.26, der beschreibt wie die Transferfunktion für `PAR`-Anweisungen berechnet werden, wenn deren Prozeßrümpfe aus Mengen von Anweisungsfolgen bestehen.

Die $\sqcap$-Abgeschlossenheit des $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmens ermöglicht einen effizienteren Algorithmus zur Berechnung der DFA-Information, so daß wir separate Algorithmen für $\mathcal{D}^{\mathcal{C}}$ und $\mathcal{D}^{\mathcal{B}}$ angeben. In Kapitel 5 werden wir die nötigen Datenstrukturen vorstellen und den Algorithmus detailliert beschreiben.

4.5.1.1 Iterativer Algorithmus für *pPL* Programme und $\mathcal{D}^{\mathcal{C}}$ DFA-Probleme

Wir entwerfen den DFA-Algorithmus für *pPL*-Programme in zwei Schritten:

1. Schritt Ein Prozeßrumpf R_i einer *pPL* Anweisung `PAR` $k_1 : R_1$ `| ... |` $k_n : R_n$ `END` ist durch die Menge $\mathcal{R}_i$ der Zuweisungsfolgen, die er potentiell ausführen kann, charakterisiert. Somit:

$$f_{\mathtt{PAR}\ k_1:R_1\ |\ldots|\ k_n:R_n\ \mathtt{END}}(x) = \bigsqcap_{p \in \mathsf{TopSorts}(k_1:\mathcal{R}_1,\ldots,k_n:\mathcal{R}_n)} f_p(x)$$

Mit Korollar 4.27:

$$= \bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n)} f_{\mathcal{R}_{i_n}}(f_{\mathcal{R}_{i_{n-1}}}(\ldots(f_{\mathcal{R}_{i_1}}(x))))$$

$\mathcal{R}_i$ und die darin enthaltenen Anweisungsfolgen können endlich oder unendlich sein. $\mathcal{R}_i$ und die Anweisungsfolgen, die $\mathcal{R}_i$ enthält, können zur Übersetzungszeit i.d.R. nicht bestimmt werden. Enthält der Prozeßrumpf R_i keine `PAR`-Anweisung, kann mit dem iterativen Algorithmus 3.56 (Berechnung des Fixpunktes eines DFA-Gleichungssystems) für R_i und ein gegebenes $x \in \mathcal{C}$ der Wert $f_{R_i}(x) = \bigsqcap_{p \in \mathcal{R}_i} f_p(x)$ berechnet werden.

Wenn kein R_i eine `PAR`-Anweisung enthält, kann jedem Prozeßrumpf ein Kontrollflußgraph (s. Definition 3.32) zugeordnet werden. Damit sind wir in der Lage, $f_{\texttt{PAR } k_1:p_1 \texttt{ |...| } k_n:p_n \texttt{ END}}(x)$ zu bestimmen:

4.40 ALGORITHMUS DFA_PAR_STMT$^{\mathcal{C}}$ (x :$\mathcal{C}$) : $\mathcal{C}$
Berechne $y = f_{\texttt{PAR } k_1:R_1 \texttt{ |...| } k_n:R_n \texttt{ END}}(x)$

$y \leftarrow \top$;
forall $\vec{\imath} \in \mathsf{s_perm}(1,n)$ **do**
  $x' \leftarrow x$;
  forall $k \in \{1,\ldots,n\}$ **do** /* *Berechne* $x' = f_{\mathcal{R}_{i_n}}(f_{\mathcal{R}_{i_{n-1}}}(\ldots(f_{\mathcal{R}_{i_1}}(x))))$ */
    $x' \leftarrow f_{\mathcal{R}_{i_k}}(x')$;
    /* $f_{\mathcal{R}_{i_k}}(x')$ *berechnet für den Startwert* x' *mittels des iterativen Algorithmus 3.56 die DFA-Information, die am Ende des Prozeßrumpfes gilt. Das Argument* x' *ist die DFA-Information, die am Eingangsknoten des Prozeßrumpfes* R_{i_k} *gültig ist.* */
  end
  $y \leftarrow y \sqcap x'$;
end
return y ; □

2. Schritt Die `PAR`-Anweisung ist natürlich Teil einer Anweisungsfolge im Quellprogramm für die wir die DFA-Information berechnen wollen. Der iterative Algorithmus 3.56 muß deshalb geändert werden:

- Enthält ein Basisblock b eine `PAR`-Anweisung, wird Algorithmus DFA_PAR_STMT$^{\mathcal{C}}$ zur Berechnung von $f_b(x)$ benutzt.
- Die zu den Prozeßrümpfen dieser `PAR`-Anweisung gehörenden Basisblöcke werden nicht besucht.

Da für $\mathcal{D}^{\mathcal{C}}$ DFA-Probleme diese Funktion nicht explizit bestimmt werden kann, sondern für ein gegebenes x nur ihren Wert, wird DFA_PAR_STMT$^{\mathcal{C}}$ jedesmal aufgerufen, wenn der iterative Algorithmus den Basisblock b besucht.

Damit der so veränderte Algorithmus terminiert, müssen die Transferfunktionen der Basisblöcke monoton sein. Da die Transferfunktionen der einfachen Anweisungen aus $\mathcal{F}^{\mathcal{C}}$ sind, ist diese Forderung erfüllt. Mit Satz 4.9 ist die Transferfunktion der `PAR`-Anweisung ebenfalls monoton, obwohl sie nicht in $\mathcal{F}^{\mathcal{C}}$ enthalten ist.

Enthält ein Prozeßrumpf weitere `PAR`-Anweisungen, erfolgt dieser Vorgang rekursiv[5].

Der Aufwand dieses Algorithmus ergibt sich wie folgt: Besteht der Prozeßrumpf R_i einer `PAR`-Anweisung mit n_p Rümpfen aus m_i Basisblöcken und ist der zugehörige CFG reduzibel mit Tiefe d_i, dann ist der Zeitaufwand, um einmal $x' = f_{\mathcal{R}_{i_{n_p}}}(f_{\mathcal{R}_{i_{n_p-1}}}(\ldots(f_{\mathcal{R}_{i_1}}(x))))$ zu berechnen: $O(\sum_{i=1}^{n_p} \text{Aufwand}(f_{\mathcal{R}_i})) = O(\sum_{i=1}^{n_p} m_i * d_i)$. Da es nur n_p verschiedene einfache Permutationen der Zahlen $1,\ldots,n_p$ gibt, ergibt sich der Gesamtaufwand des Alogrithmuses

[5] Diese kurze Skizze des Algorithmus läßt natürlich noch viele Fragen offen (z.B. wie genau wird ein paralleles Programm im Kontrollflußgraphen dargestellt), die aber in Kapitel 5 beantwortet werden.

also zu $O(n_p * \sum_{i=1}^{n_p} m_i * d_i)$. Ist $d_p = \max(d_i)$ die größte Tiefe und $m_p = \max(m_i)$ die größte Zahl der Basisblöcke aller Prozeßrümpfe dieser PAR-Anweisung, dann vereinfacht sich dies zu $O(n_p^2 * d_p * m_p)$.

Ist D die Tiefe und M die Anzahl der Basisblöcke des reduziblen Kontrollflußgraphen, der eine PAR-Anweisung enthält (ohne die Basisblöcke der Prozeßrümpfe), beträgt der Gesamtaufwand des modifizierten iterativen Algorithmus $O((M + \mathit{Aufwand}(\text{DFA_PAR_STMT}^{\mathcal{C}})) * D) = O((M + n_p^2 * d_p * m_p) * D)$.

Enthält eine Prozedur k PAR-Anweisungen (auch geschachtelte) und setzen wir $m = M + m_1 + \cdots + m_k$ (die Anzahl aller Basisblöcke dieser Prozedur), $n = \max(n_1, n_2, \ldots, n_k)$, und $d = \max(D, d_1, d_2, \ldots d_k)$, so beträgt der Aufwand des modifizierten iterativen Algorithmus für $\mathcal{D}^{\mathcal{C}}$ DFA-Probleme $O(n^2 * m * d)$ Bitvektorschritte.

Um wieviel größer ist dieser Aufwand im Vergleich zur DFA von sequentiellen Programmen? Wenn wir alle PAR-Anweisungen eines Programms durch die sequentielle Hintereinanderausführung ihrer Prozeßrümpfe ersetzen, erhalten wir ein (in Bezug auf die DFA) „vergleichbar komplexes“ sequentielles Programm. Da die, den Prozeßrümpfen entsprechenden CFG's die *Single-Entry/Single-Exit* Eigenschaft haben, ist die Tiefe dieses sequentiellen Programms gleich der maximalen Tiefe der Teil-CFG's. Der iterative Algorithmus 3.56 benötigt zur Analyse dieses sequentiellen Programms $O(m * d)$ Bitvektorschritte.

Der Aufwand des modifizierten Algorithmus zur Analyse eines parallelen Programms für ein $\mathcal{D}^{\mathcal{C}}$-Problem ist somit quadratisch (in der größten Anzahl von Prozeßrümpfen einer PAR-Anweisung) größer, als der für ein vergleichbares sequentielles Programm.

4.5.1.2 Iterativer Algorithmus für *pPL* Programme und $\mathcal{D}^{\mathcal{B}}$ DFA-Probleme

Für $\mathcal{D}^{\mathcal{B}}$ DFA-Probleme kann dieser Algorithmus verbessert werden, da die Transferfunktionen $\sqcap$-abgeschlossen sind. Statt für jeden Prozeßrumpf $\boldsymbol{R}_i$ den iterativen Algorithmus aufzurufen, können wir mit den Bitvektorgleichungen aus Satz 4.39 in einer „Vorberechnungsphase“ die Transferfunktion bestimmen, die der ganzen PAR-Anweisung entspricht. Da $\mathcal{D}^{\mathcal{B}}$ $\sqcap$-abgeschlossen ist, ist die Transferfunktion der PAR-Anweisung monoton und distributiv.

Diese Vorberechnungsphase berechnet mit $O(m)$ Bitvektorschritten die Mengen gen und kill für alle Basisblöcke (s. Algorithmus P_DFA$^{\mathcal{B}}$_GENKILL_CFG in Kapitel 5). Damit werden $O(m * d)$ Bitvektorschritte benötigt, um die DFA-Information für ein $\mathcal{D}^{\mathcal{B}}$-Problem von parallelen Programmen zu berechnen.

Eine andere Möglichkeit der Berechnung basiert auf Satz 4.26 und Lemma 4.7, mit dem die Transferfunktion einer PAR-Anweisung einfach bestimmt werden kann. Diese Vorgehensweise wurde in KNOOP, STEFFEN und VOLLMER [1996] gewählt.

4.5.2 Welche DFA-Information erreicht eine Anweisung innerhalb eines Prozeßrumpfes

Grundlage zur Berechnung der DFA-Information, die vor einer Anweisung s eines Prozeßrumpfes gültig ist, bilden Satz 4.35 und Korollar 4.37. Wie im vorigen Abschnitt begründet, kann jeder Prozeßrumpf $\boldsymbol{R}_i$ durch die Menge $\mathcal{R}_i$ der möglichen Anweisungsfolgen charakterisiert werden. Selbst wenn wir diese Mengen nicht explizit kennen, so können die Anweisungsfolgen nur aus Anweisungen bestehen, die in den Prozeßrümpfen vorkommen. Erweitern wir die Definition von sibl für Mengen von Anweisungsfolgen, ist Satz 4.35 auch für Mengen von Anweisungsfolgen gültig:

4.41 KOROLLAR ($\mathrm{MOI}_{s \in \mathtt{PAR}}(x)$)
Gegeben ein $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen. Seien $P_1, \ldots, P_n$ Mengen von Anweisungsfolgen, $k_1, \ldots, k_n \in$

$\mathbb{N}$. *Sei $s \in p \in P_i$.*

$$f_{\{p\downarrow s \mid p \in \mathsf{TopSorts}(k_1:P_1,\ldots,k_n:P_n)\}}(x) = f_{P_i\downarrow s}(x) \sqcap \prod_{\substack{t \in \mathsf{sibl}[s] \\ f_t = \mathsf{const}}} f_t(x)$$

□

Der Beweis folgt dem von Satz 4.35 und ist hier nicht angegeben. Es bleibt zu bemerken, daß die Mengen P_i unendlich viele Elemente haben dürfen. Da das Quellprogramm nur endlich lang ist, werden die Anweisungsfolgen immer nur aus endlich vielen verschiedenen Anweisungen bestehen, so daß die rechte Seite der Gleichung effektiv berechenbar ist. Für den $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen kann somit Korollar 4.37 auf Mengen von Anweisungsfolgen übertragen werden.

Zur Berechnung von $f_{\{p\downarrow s \mid p \in \mathsf{TopSorts}(k_1:P_1,\ldots,k_n:P_n)\}}(x)$ wird somit im wesentlichen ein Algorithmus benötigt, der $\mathsf{sibl}[s]$ bestimmt. Er ist in Kapitel 5 angegeben.

4.6 DFA für *pPL$_{sync}$* und *pPL$_{lock}$* Programme

Explizite Synchronisation von Prozessen kann als Reihenfolgebedingung an die Anweisungen verschiedener Prozeßrümpfe aufgefaßt werden [Andrews und Schneider, 1988]. Dadurch reduziert sich die Zahl der möglichen Interleavings einer `PAR`-Anweisung. Leider ist es i.d.R. nicht möglich die Menge der ausgeschlossenen Interleavings exakt zu bestimmen. Ignoriert man deshalb die Synchronisation, werden somit zu viele Interleavings betrachtet[6]. Der folgende Satz erlaubt es dennoch, die berechnete DFA-Information zu benutzen, da sie zwar unschärfer aber immer noch konservativ ist: Optimierungen werden nicht erlaubt, die bei exakter Kenntnis der möglichen Interleavings erlaubt wären[7].

4.42 Satz
Sei $(\mathcal{L}, \sqsubseteq, \sqcap)$ ein Halbverband und seien A, B Teilmengen von $\mathcal{L}$, mit $A \subseteq B$. Dann gilt:

$$\prod_{b \in B} b \sqsubseteq \prod_{a \in A} a$$

□

Definieren wir $f^*_{\mathtt{PAR}}(x)$ bzw. $\mathsf{in}^{\|}{}^*_s(x)$ als die Information, die nach der `PAR`-Anweisung bzw. die vor der Anweisung s in einer `PAR`-Anweisung gültig ist, wenn nur die erlaubten Interleavings betrachtet werden, dann können wir mit Satz 4.42 folgern:

4.43 Satz (DFA für *pPL$_{sync}$* und *pPL$_{lock}$*-Programme)
Für pPL-Programme, die `SIGNAL`- und `WAIT`-Synchronisationsanweisungen enthalten und $\mathcal{D}^{\mathcal{C}}$ $\sqcap$-DFA-Probleme gilt:

$$f_{\mathtt{PAR}}(x) \sqsubseteq f^*_{\mathtt{PAR}}(x)$$
$$\mathsf{in}^{\|}_s(x) \sqsubseteq \mathsf{in}^{\|}{}^*_s(x)$$

□

Da dieser Satz davon ausgeht, daß die Betrachtung zusätzlicher Interleavings immer noch ein korrektes Optimierungsergebnis liefern, gilt er nicht für $\sqcup$-Halbverbände.

[6] Dies ist vergleichbar zum `if .. then .. else .. end` bei dem auch beide Teile untersucht werden.

[7] Dies ist vergleichbar mit der Situation von nur monotonen DFA-Problemen für sequentielle Programme: Der iterative Algorithmus berechnet für sie nur den maximalen Fixpunkt (MFP), für den gilt: MFP $\sqsubseteq$ MOP.

4.7 Schlußfolgerungen

Dieses Kapitel legt die theoretische Grundlage der Datenflußanalyse paralleler Programme. Obwohl es u.U. exponentiell viele Interleavings gibt, zeigt es, daß DFA effizient möglich ist. Zwei Eigenschaften von Datenflußanalyseproblemen bilden die Grundlage:

- Die Transferfunktionen sind entweder die Identität oder konstant.
- Die Menge der Transferfunktionen sind $\sqcap$-abgeschlossen.

Die große und für die Optimierungen eines Übersetzers wichtigste Klasse der Bitvektor-DFA-Probleme erfüllt beide Anforderungen, so daß diese Eigenschaften keine Einschränkung für die Praxis bedeuten. Die resultierenden Bitvektor-DFA-Algorithmen sind genauso effizient, wie die für sequentielle Programme. Hat ein DFA-Problem nur die erste Eigenschaft, so ist eine effiziente Berechnung der DFA-Information immer noch möglich, wenn auch der Aufwand quadratisch größer (in der größten Anzahl von Prozeßrümpfen einer `PAR`-Anweisung) als für eine Bitvektor-DFA des gleichen Programms ist. Enthält ein paralleles Programm explizite Synchronisationsanweisungen, ist die DFA (für $\sqcap$-Probleme) immer noch möglich. Sie liefert dann jedoch nicht das genaue Resultat, sondern nur ein unschärferes, aber immer noch korrektes.

5 Datenstrukturen und Algorithmen

In diesem Kapitel werden einige Datenstrukturen und Algorithmen vorgestellt, mit denen parallele Programme analysiert werden können. Sie basieren auf Vorbildern, die für die Programmanalyse sequentieller Programme entworfen wurden. Diese werden für unsere Aufgabe angepaßt.

5.1 Notationen

Die Bezeichner der in diesem Kapitel eingeführten Datenstrukturen haben Namen, die an die Zwischensprache CCMIR-P[1] [LIBOUREL, VAN SOMEREN und VOLLMER, 1993] des COMPARE-Projektes erinnern[2]. In CCMIR ist ein Programm als Kontrollflußgraph dargestellt. Ausdrücke der Quellsprache werden in Ausdrücke in CCMIR abgebildet, wobei CCMIR-Ausdrücke seiteneffektfrei sein müssen. Enthält ein Quellsprachausdruck Seiteneffekte (z.B. Funktionsaufrufe oder Zuweisungen) müssen diese Quellsprachausdrücke in mehrere CCMIR-Anweisungen zerlegt werden[3]. Diese Eigenschaft setzen wir bei unseren Algorithmen voraus.

Zur Notation der Algorithmen benutzen wir eine imperative, an *Modula* erinnernde Syntax und Semantik. Es werden folgende Schriftarten benutzt: ***Variablen*** , `Benutzertypen`, PROZEDURAUFRUFE und **Schlüsselworte**. Der Zuweisungsoperator ist „$\leftarrow$". Alle Parameter eines Algorithmus werden mit der *Call-by-Value* Semantik übergeben, es sei denn, daß vor dem Parametername **var** steht, was *Call-by-Reference* bedeutet. Die **forall** Schleife läuft sequentiell über alle Elemente der angegeben Struktur.

5.2 Der parallele Kontrollflußgraph (pCFG)

Bevor wir einen konkreten Algorithmus zur Berechnung der DFA-Information angeben können, müssen wir die Datenstrukturen erklären, die uns geeignet erscheinen, ein paralleles Programm im Übersetzer darzustellen.

Die `PAR` S_1 `|` ... `|` S_n `END` Anweisung wird vergleichbar einem Prozeduraufruf übersetzt und ausgeführt:

---

[1] *Common* COMPARE *Medium Intermediate Representation, Parallel extension*

[2] Vergleicht man die folgenden Ausführungen mit Kapitel 3 oder AHO *et al.* [1986] stellt man jedoch fest, daß die hier vorgestellten Datenstrukturen und Algorithmen nicht spezifisch für CCMIR sind.

[3] Z.B. `a := f(b) * 2` wird zu `t := f(b); a := t * 2` mit einer Hilfsvariablen `t`. In vereinfachter CCMIR-Darstellung:

```
mirFuncCall(Func=mirObjectAddr(Obj=f),
            Args=LIST_mirExpr(mirContent(Addr=mirObjectAddr(Obj=b)),NIL),
            Result=mirObjectAddr(Obj=t));
mirAssign(Lhs=mirObjectAddr(Obj=a),
          Rhs=mirMult(Left=mirContent(Addr=mirObjectAddr(Obj=t)),
                      Right=mirConst(Val=2));
```

- Wie bei Prozeduren gibt es keine Sprünge in einen Prozeßrumpf hinein oder aus ihm heraus[4].
- Wie bei Prozedurrümpfen, wird jedem Prozeßrumpf ein separater Kontrollflußgraph zugeordnet. Er hat einen eindeutigen Eingangs- und Ausgangsknoten. Da Sprünge in einen Prozeßrumpf oder aus einem Prozeßrumpf verboten sind, sind die CFG's disjunkt.
- So wie ein Prozeduraufruf die Ausführung der (im Übersetzer durch den CFG der Prozedur gegebenen) Anweisungen der Prozedur veranlaßt, startet die (CCMIR) `mirParallel`-Anweisung Prozesse, die durch die CFG's der Prozeßrümpfe $S_1, \ldots, S_n$ spezifiziert sind. Die Ausführung des Prozesses, der eine `mirParallel`-Anweisung ausführt, wird suspendiert, bis alle Kindprozesse terminiert haben.

5.1 Definition (PAR-Zwischensprachanweisung)
Die `PAR`-Anweisung der Quellsprache wird durch die **mirParallel** Anweisung in der Zwischensprache dargestellt. Die Zwischensprachanweisung hat die gleiche Bedeutung wie die Quellsprachanweisung. Ein Prozeßrumpf wird dabei durch den **Prozeßrumpfdeskriptor** (**PROCESS_BODY**) beschrieben. Die Attribute sind:

PROCESS_BODY	
cfg : *CFG*	Der CFG, welcher einem Prozeßrumpf entspricht.
rep : *REPLICATOR*	Darstellung der Replikatorinformation (falls der Prozeßrumpf repliziert ist).
n : `INTEGER`	Anzahl der direkt in diesem Prozeßrumpf enthaltenen `mirParallel`-Anweisungen.
pars : `ARRAY OF mirParallel`	*pars*[*1* ... *n*] verweist auf die in diese Prozeßrumpf direkt enthaltenen `mirParallel`-Anweisungen.

`mirParallel`	
n : `INTEGER`	Anzahl der Prozeßrümpfe, $n \geq 0$.
bodies : `ARRAY OF` *PROCESS_BODY*	*bodies*[*1* ... *n*] enthält die Prozeßrumpfdeskriptoren der Prozeßrümpfe, d.h. repräsentiert die parallelen Kanten.

Ist $s_1; \ldots ; s_i; s_{i+1}; \ldots ; s_k$ die Folge der Anweisungen eines Basisblocks, und ist s_i eine `mirParallel`-Anweisung wird nach der Termination aller Kindprozesse, die Ausführung mit der Anweisung s_{i+1} fortgesetzt. □

5.2 Definition (Paralleler Kontrollflussgraph)
Ein **paralleler Kontrollflußgraph** (**pCFG**) einer Prozedur ist ein Quintupel

$$pG \stackrel{\text{def}}{=} (\mathcal{G}, p\mathcal{S}, \mathcal{N}, \mathcal{E}, pE, G_0)$$

wobei

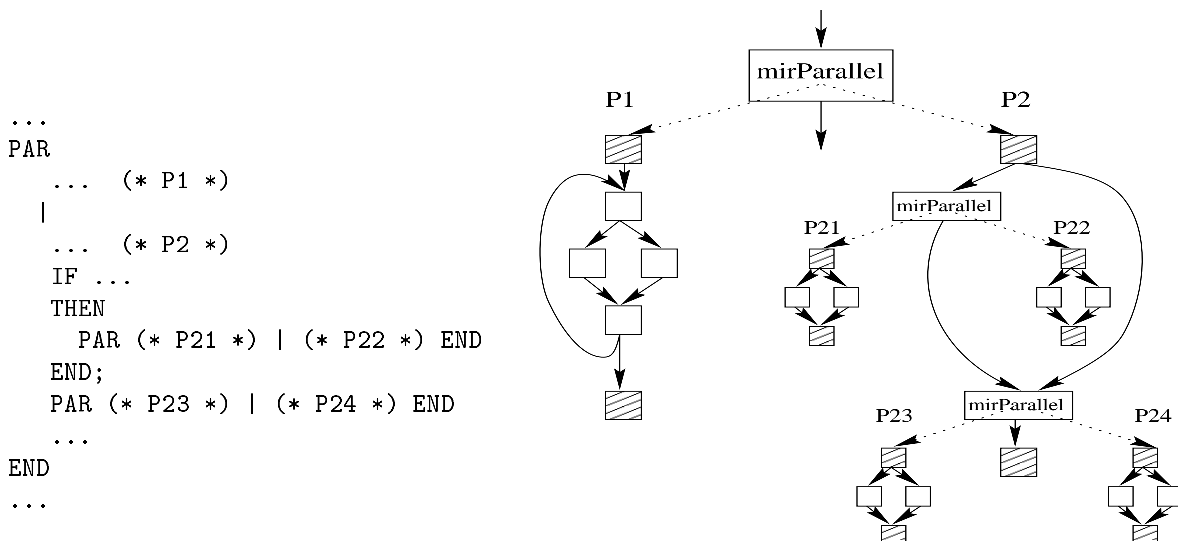
- $\mathcal{G} = \{G \mid G = (N_G, E_G, entry_G, exit_G)$ ist ein CFG$\}$. Die in $\mathcal{G}$ enthaltenen CFG's sind disjunkt: $\forall G', G'' \in \mathcal{G} : N_{G'} \cap N_{G''} = \emptyset$ und $E_{G'} \cap E_{G''} = \emptyset$. $G_0 \in \mathcal{G}$ ist der **Eingangskontrollflußgraph** der Prozedur.
- $p\mathcal{S}$ ist die Menge aller `mirParallel`-Zwischensprachanweisungen der Prozedur.
- $\mathcal{N} = \bigcup_{G \in \mathcal{G}} N_G$ die Menge aller Basisblöcke der Prozedur.
- $\mathcal{E} = \bigcup_{G \in \mathcal{G}} E_G$ die Menge aller **Sprungkanten** der Prozedur.

[4] Siehe Definition 3.8.

- $pE \subseteq p\mathcal{S} \times \mathcal{G}$ die Menge der **parallelen Kanten**, die eine `mirParallel`- (bzw. `PAR`-) Anweisung mit dem zugehörigen Prozeßrumpf „verbinden“.

Die Prozedur sei gemäß Transformation 3.16 auf Seite 15 normiert. G_0 ist der Prozeßrumpf, welcher dem Prozedurrumpf entspricht.
Die Vorgänger bzw. Nachfolger eines Basisblocks sind nur bezüglich der Sprungkanten definiert (vgl. Definition 3.32 auf Seite 25). Somit hat in G $entry_G$ keine Vorgänger und $exit_G$ keine Nachfolger.
Jedem Prozeßrumpf P einer Quellsprach `PAR`-Anweisung ist ein Kontrollflußgraph $CFG(P)$ zugeordnet. (Genauer: einem Prozeßrumpf ist ein **Prozeßrumpfdeskriptor** zugeordnet, der den CFG als Attribut enthält). Dieser CFG hat einen eindeutigen Eingangs- und einen eindeutigen Ausgangsknoten ($entry_{CFG(P)}$ bzw. $exit_{CFG(P)}$)[5].
Eine `mirParallel`-Anweisung ist **direkt** in einem Prozeßrumpf P **enthalten**, wenn sie als Anweisung in einem $CFG(P)$-Knoten vorkommt. □

Ein graphische Darstellung eines Prozedurfragmentes ist in folgender Abbildung gegeben.



Dem auf der linken Seite dargestellten Programmfragment entspricht der auf der rechten Seite gezeigte parallele Kontrollflußgraph. Parallele Kanten sind als punktierte, Sprungkanten als durchgezogene Pfeile gekennzeichnet. Die gestreiften Kästchen stellen Eingangs- bzw. Ausgangsbasisblöcke dar.

Abbildung 5.1: Darstellung von `PAR`-Anweisungen in einem parallelen Kontrollflußgraphen.

Gibt es geschachtelte `PAR`-Anweisungen, kann dem pCFG eine baumartige Struktur überlagert werden. Bei der Berechnung der DFA-Information im $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen wird sie benötigt.

5.3 Definition (Prozessrumpfbaum)
Ein **Prozeßrumpfbaum** einer Prozedur ist ein Tupel

$$pB \stackrel{\text{def}}{=} (D \cup p\mathcal{S}, K)$$

wobei

- D die Menge aller **Prozeßrumpfdeskriptoren** und $p\mathcal{S}$ die Menge aller `mirParallel`-Anweisungen der Prozedur ist.

[5]In CCMIR kennzeichnen die `mirBeginProcessBody` bzw. `mirEndProcessBody` Zwischensprachanweisungen (vgl. Definition 3.31) diese Basisblöcke.

- $K \subseteq (D \times p\mathcal{S}) \cup (p\mathcal{S} \times D)$ die Menge der gerichteten Kanten im Prozeßrumpfbaum ist.

Eine `mirParallel`-Anweisung hat $n \geq 0$ Prozeßrumpfdeskriptoren, die den Prozeßrümpfen entsprechen, als Kinder. Ein Prozeßrumpfdeskriptor hat $m \geq 0$ `mirParallel`-Anweisungen als Kinder: die direkt in diesem Prozeßrumpf enthaltenen `PAR`-Anweisungen. □

Für das Programmfragment aus Abbildung 5.1 ist der Prozeßrumpfbaum in der nächsten Abbildung dargestellt.

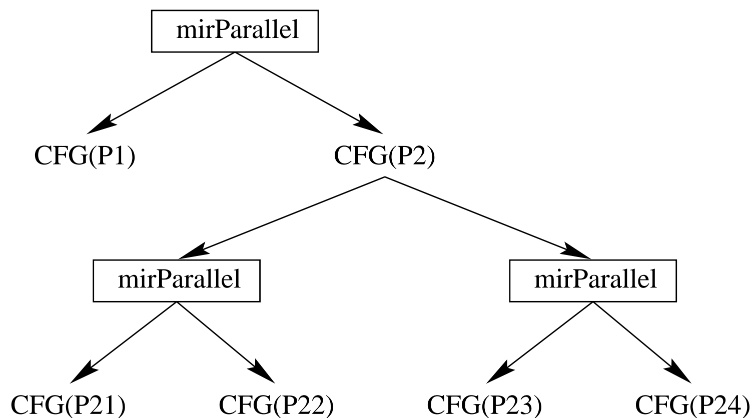


Abbildung 5.2: Darstellung des Prozeßrumpfbaumes für das Programm aus Abbildung 5.1

In der Beschreibung der Algorithmen benutzen wir folgende Bezeichnungen:

5.4 Definition
procedure.pcfg ist der pCFG der Prozedur *procedure* . *procedure.pcfg.entry* ist der entsprechende Eingangs-CFG (G_0). *procedure.ptree* ist die Wurzel des **Prozeßrumpfdeskriptorbaumes** und zeigt auf die äußerste `PAR`-Anweisung der normierten Prozedur (s. Transformation 3.16 auf Seite 15). Für einen Kontrollflußgraphen *cfg* ist *cfg.entry* (*cfg.exit*) der Eingangsknoten (Ausgangsknoten) von *cfg* . *b.pred* (*b.succ*) ist die Menge der Vorgänger (Nachfolger) des Basisblocks *b* . *b.stmts* ist die Folge der Anweisungen des Basisblocks *b* . □

5.2.1 Alternative Darstellungen

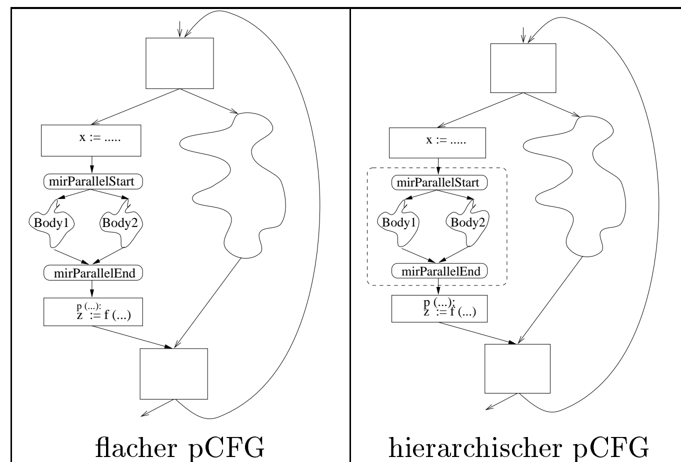
Andere Möglichkeiten der Darstellung eines parallelen Programms sind z.B.:

Flacher pCFG: Die `PAR`-Anweisung wird in mehrere Basisblöcke aufgebrochen. Es gibt spezielle Anfangs- und Endknoten (`mirParallelStart` und `mirParallelEnd`). Der Anfangsknoten `mirParallelStart` hat als Nachfolger die Eingangsknoten der entsprechenden Prozeßrumpfkontrollflußgraphen. Die Ausgangsknoten der Prozeßrumpf-CFG's haben als Nachfolger den `mirParallelEnd` Knoten. Diese Darstellung wurde für die Zwischensprache *MOBIL-P* [Vollmer, 1989; Schröer und Vollmer, 1992] gewählt, welche die Basis der ersten Implementierung von *Modula-P* bildete, siehe Abbildung 5.3(a).

Hierarchischer pCFG: Es gibt zwei Sorten von Knoten: solche, die „normale“ Basisblöcke darstellen, und *Superknoten*, die einen ganzen Kontrollflußgraphen darstellen. Man erhält somit einen hierarchischen Graphen, vgl. Wolfe und Srinivasan [1991] und Abbildung 5.3(b).

Der Nachteil des flachen pCFG's ist, daß die Teilgraphen, die einem Prozeßrumpf entsprechen nicht einfach zu finden sind. Beim hierarchischen pCFG müssen unterschiedliche Knotenarten in der Implementierung gehandhabt werden.

```
REPEAT ...
  IF ...
  THEN
     x := ...;
     PAR Body1 | Body2 END
     p (...); z := f (...);
  ELSE
  END; ...
UNTIL ...
```



Das gestrichelten Rechteck ist ein Superknoten, die durchgezogenen sind „normale" Basisblöcke.

Abbildung 5.3: Zwei andere mögliche pCFG's für das Programmfragment

5.3 Die Dominanzrelation und Dominanzgrenze im pCFG

Die parallele Dominanzrelation wird „lokal" nur für den CFG eines Prozeßrumpfes definiert. In Definition 3.33 wird dazu „Eingangsknoten" durch „Eingangsknoten des Prozeßrumpfes" ersetzt. In Algorithmus P_DOM muß daher nur die Initialisierung gegenüber dem Algorithmus für sequentielle Programme verändert werden.

Die parallele Dominanzgrenze wird ebenfalls nur lokal für den CFG eines Prozeßrumpfes definiert. Somit muß der Algorithmus, der die parallele Dominanzgrenze berechnet, für jeden Prozeßrumpf separat aufgerufen werden. Die Details sind im Algorithmus P_DF beschrieben.

5.5 ALGORITHMUS P_DOM (*procedure* : `mirProcedure`)
Berechne die parallele Dominanzrelation aller Basisblöcke einer Prozedur.

Die Dominatoren eines Basisblocks sind als Mengen *doms* : **set of** `mirBasicBlock` dargestellt. Wir folgen dem in [AHO *et al.*, 1986, S. 671] angegebenen Algorithmus. Im Unterschied zum sequentiellen Fall gibt es in einem pCFG mehr als nur einen Basisblock ohne Vorgänger: alle Eingangsblöcke von Prozeßrümpfen („Tote" Basisblöcke sind bereits aus dem pCFG entfernt). Jeder Eingangsknoten ist Wurzel eines Dominatorbaumes des CFG's eines Prozeßrumpfes. Dies macht sich allerdings nur in der Initialisierung bemerkbar. Da die CFG's der Prozeßrümpfe disjunkt sind, kann die Dominanzrelation für alle Basisblöcke des pCFG's gleichzeitig berechnet werden.
Alternativ könnte der Algorithmus aus [AHO *et al.*, 1986, S. 671] für jeden CFG des pCFG's separat aufgerufen werden.

/* *Initialisiere* */
forall Basisblöcke *b* **in** *procedure.pcfg* **do**
  if *b.pred* = **empty then** /* *b ist ein Prozeßrumpfeingangsbasisblock* */
    b.doms $\leftarrow \{b\}$;
  else
    b.doms $\leftarrow \{p | p \in$ *procedure.pcfg*$\}$; /* *alle Basisblöcke* */
  end
end

```
/* Iterative Berechnung der Dominatormengen */
repeat
  changed ← false;
  forall Basisblöcke b in procedure.pcfg do
    old ← b.doms ;
    b.doms ← {b} ∪ ⋂_{p∈b.pred} p.doms
    if old ≠ b.doms then
      changed ← true
    end
  end
until changed = false                                            □
```

5.6 ALGORITHMUS P_DF (*root* : `mirBasicBlock`)
Berechne die parallele Dominanzgrenze des Basisblocks *root* und all seiner Kinder im Dominatorbaum.

Die Dominanzgrenze eines Basisblocks ist als Menge *DF* : **set of** `mirBasicBlock` dargestellt. Wir folgen dem in [CYTRON *et al.*, 1991, S. 466] angegebenen Algorithmus. Da im pCFG alle Prozeßrumpfeingangsknoten die Wurzeln von disjunkten Dominatorbäumen sind, muß dieser Algorithmus für alle Eingangsknoten der CFG's eines pCFG's der zu analysierenden Quellprozedur aufgerufen werden. Ansonsten gibt es keinen Unterschied zum Algorithmus für sequentielle Programme. *b.idom* : `mirBasicBlock` ist der eindeutige direkte Dominator des Basisblocks *b* , er wurde zuvor aus der Dominanzrelation berechnet (s. [AHO *et al.*, 1986]).

```
/* Bottom-Up Traversierung des Dominatorbaumes, dessen Wurzel root ist. */
forall Basisblöcke child in Kinder des Dominatorbaumknotens root do
  P_DF (child );
end

root.DF ← ∅;
/* Berechne DF-local, siehe [CYTRON et al., 1991]. */
forall Basisblöcke succ in root.succ do
  if succ.idom ≠ root then
    root.DF ← root.DF ∪ {succ };
  end
end

/* Berechne DF-up, siehe [CYTRON et al., 1991]. */
forall Basisblöcke child in Kinder des Dominatorbaumknotens root do
  forall Basisblöcke df in child.DF do
    if df.idom ≠ root then
      root.DF ← root.DF ∪ {df };
    end
  end
end                                                              □
```

5.4 Berechnen der Menge sibl

Die Menge der Anweisungen sibl$[s]$, die zu einer gegeben Anweisung s in einem beliebigen *pPL*-Programm nebenläufig ausgeführt werden können, wird mittels eines „Hilfsattributes *all_stmts* berechnet. *mirParallel.all_stmts* ist die Menge der Dreiadreßanweisungen, die in den zugehörigen Prozeßrümpfen vorkommen. *PROCESS_BODY.all_stmts* ist die Menge

der Dreiadreßanweisungen die in den Basisblöcken des zugehörigen CFG's vorkommen und in allen Prozeßrümpfen von geschachtelten `PAR`-Anweisungen dieses Prozeßrumpfes. (Die folgende Formulierung kann als Basis einer Attributgrammatik betrachtet werden, für die ein Auswerter generiert wird.)

Die Menge der Anweisungen die ein Prozedurrumpf bzw. eine `mirParallel`-Anweisung enthält ist definiert durch:

$$\mathtt{mirParallel.all_stmts} \leftarrow \bigcup_{j \in \{1,\ldots,\mathtt{mirParallel.n}\}} \mathtt{mirParallel.bodies}[j]\mathtt{.all_stmts}$$

Neben den Anweisungen des CFG's eines Prozeßrumpfes P, gehören auch die Anweisungen von in ihm geschachtelte `PAR`-Anweisungen zu $P.\mathtt{all_stmts}$.

$$\mathtt{PROCESS_BODY.all_stmts} \leftarrow \mathtt{PROCESS_BODY.cfg.stmts} \cup \bigcup_{j \in \{1,\ldots,\mathtt{PROCESS_BODY.n}\}} \mathtt{PROCESS_BODY.pars}[j]\mathtt{.stmts}$$

Diese Mengen können in einem *bottom-up* Durchlauf durch den Prozeßrumpfbaum bestimmt werden.

Da die `PAR`-Anweisung, die dem normierten Prozedurrumpf entspricht, nicht in einer anderen `PAR`-Anweisung enthalten ist, gilt:

$$\mathtt{procedure.ptree.sibl} \leftarrow \emptyset$$

Für alle Prozeßrümpfe R_i einer `mirParallel`-Anweisung:

$$\mathtt{mirParallel.bodies}[i]\mathtt{.sibl} \leftarrow \mathtt{mirParallel.sibl} \cup \begin{cases} \bigcup_{j \in \{1,\ldots,\mathtt{mirParallel.n}\}} \mathtt{mirParallel.bodies}[j]\mathtt{.all_stmts} & \text{falls } (k_i > 1) \\ \bigcup_{j \in \{1,\ldots,\mathtt{mirParallel.n}, j \neq i\}} \mathtt{mirParallel.bodies}[j]\mathtt{.all_stmts} & \text{falls } (k_i = 1) \end{cases}$$

Für alle `mirParallel`-Anweisungen P_i eines Prozeßrumpfes:

$$\mathtt{PROCESS_BODY.pars}[i]\mathtt{.sibl} \leftarrow \mathtt{PROCESS_BODY.sibl}$$

Diese Mengen können in einem *top-down* Durchlauf durch den Prozeßrumpfbaum bestimmt werden. Der Aufwand ist somit linear in der Anzahl der Dreiadreßanweisungen.

5.5 Der Strukturansatz für parallele Programme

Für sequentielle strukturierte Programme kann die DFA-Information von $\mathcal{D}^{\mathcal{B}}$-Problemen wie in Abschnitt 3.4.3.6 beschrieben, berechnet werden. Wird die `PAR`-Quellsprachanweisung im Strukturbaum dargestellt (vergleichbar der `IF`-Anweisung), können die Überlegungen aus Abschnitt 4.5 (DFA für *pPL* Programme) auf den Strukturansatz übertragen werden. Dies bedeutet, daß die Bitvektorgleichungen aus Satz 4.39 auch auf beliebige *pPL* Programme angewendet werden dürfen. Die Analyse berechnet (wie im sequentiellen Fall auch) zuerst in einem *bottom-up* Durchlauf die gen- und kill-Mengen. Danach in einem *top-down* Durchlauf die $\mathsf{in}^{;}$- und out-Mengen. Zur Berechnung der Menge sibl (und somit $\mathsf{in}^{||}$) werden die im vorigen Abschnitt 5.4 definierten Mengen $\mathtt{all_stmts}$ und $\mathtt{sibl}$ benutzt. Statt die Berechnung im Prozeßrumpfbaum durchzuführen, kann sie natürlich auch im Strukturbaum ausgeführt werden. Ist n die Zahl der Knoten im Strukturbaum (= Anzahl der Anweisungen in Programm), werden somit $O(n)$ Bitvektorschritte zur Berechnung der sibl, gen, kill, $\mathsf{in}^{;}$, $\mathsf{in}^{||}$ und out-Mengen benötigt.

5.6 Der iterativ-rekursive pDFA-Algorithmus

Basierend auf den Überlegungen aus Abschnitt 4.5 können wir die konkreten Algorithmen zur Berechnung der DFA-Information angeben. In Abschnitt 4.5 wird der Aufwand der hier nur ausführlicher dargestellten Algorithmen begründet. Für den $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen können wir mit Algorithmus P_DFA$^{\mathcal{C}}$ die DFA-Information berechnen, die vor bzw. nach jedem Basisblock gültig ist. Für den $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen berechnet dies Algorithmus P_DFA$^{\mathcal{B}}$. O.B.d.A. nehmen wir für $\mathcal{D}^{\mathcal{C}}$ Vorwärtsprobleme mit beliebigem $\sqcap$-Operator und für $\mathcal{D}^{\mathcal{B}}$ Kann-Vorwärtsprobleme an. Zur präziseren Beschreibung der Algorithmen noch einige Definitionen:

5.7 Definition
Für $\mathcal{D}^{\mathcal{C}}$ DFA-Probleme hat jede Anweisung und jeder Basisblock des pCFG's zwei Attribute ***inS*** und ***out*** , welche Werte aus $\mathcal{C}$ annehmen können. ***inS*** repräsentieren die DFA-Information in$^{;}$ (s. Definition 4.38) und ***out*** entsprechend die Information, die nach der Anweisung bzw. Basisblock gültig ist. Die Transferfunktion ist durch das Attribut ***f*** einer Anweisung bzw. eines Basisblockes gegeben.
Für $\mathcal{D}^{\mathcal{B}}$ DFA-Probleme werden die DFA-Informationen mittels Bitvektoren (**bitset**) implementiert. Jeder Basisblock hat vier Bitvektorattribute ***inS*** , ***out*** , ***gen*** und ***kill*** , welche die Transferfunktionen und deren Werte darstellen. □

5.6.1 Behandlung von Prozeduraufrufen

Der Einfachheit halber werden Prozeduraufrufe „pessimistisch“ behandelt. Das bedeutet, ein Prozeduraufruf zerstört alle Informationen, die textuell vor dem Prozeduraufruf gültig sind: Ohne weitergehende Analyse der aufgerufen Prozedur muß man davon ausgehen, daß die gerufene Prozedur jede Variable verändern kann. Verfahren zur besseren Behandlung von Prozeduraufrufen, der sogenannten interprozeduralen DFA, sind z.B. in [Burke und Choi, 1992; Knoop *et al.*, 1992; Reps *et al.*, 1995] beschrieben.

5.6.2 $\mathcal{D}^{\mathcal{C}}$-Probleme

5.8 Algorithmus P_DFA$^{\mathcal{C}}$ (***procedure*** : mirProcedure, ***initial*** : $\mathcal{C}$)
Berechne DFA-Information, die vor bzw. nach jedem Basisblock der ganzen Prozedur ***procedure*** gültig ist. ***initial*** ist der Wert, den das ***inS*** -Attribut des Eingangsknoten des Eingangs-CFG's der Prozedur ***procedure*** haben soll.
Der Einfachheit halber nehmen wir o.B.d.A. an, daß eine `mirParallel`-Anweisung in einem Basisblock alleine steht, d.h. die einzige Nichtsprunganweisung des Basisblocks ist.

---

var *out* : $\mathcal{C}$;
forall Basisblöcke *b* **in** *procedure.pcfg* **do** /* *Initialisiere DFA-Attribute* */
  Berechne aus den Transferfunktionen *s.f* aller Anweisungen $s \in b$ ($s \neq$ `mirParallel`), die Transferfunktion *b.f* ;
  /* *Für Anweisungen* $\neq$ `mirParallel`*-ist dies möglich, da* $\forall f, g \in \mathcal{F}^{\mathcal{C}} : f \circ g \in \mathcal{F}^{\mathcal{C}}$. *Für die* `mirParallel` *Anweisung kann* f *nicht berechnet werden (da* $f \sqcap g \notin \mathcal{F}^{\mathcal{C}}$*), sondern nur* $f(x)$. *Deshalb ist es einfacher, wenn ein Basisblock, der eine* `mirParallel`*-Anweisung enthält, diese die einzige Anweisung in* *b* *ist.* */
  b.inS $\leftarrow \top$; *b.out* $\leftarrow \top$;
end
out $\leftarrow$ P_DFA$^{\mathcal{C}}$_CFG (*procedure.pcfg.entry* , *initial*); □

5.9 ALGORITHMUS P_DFA$^{\mathcal{C}}$_CFG (*cfg* : CFG, *in_cfg* : $\mathcal{C}$) : $\mathcal{C}$

Berechne für alle Basisblöcke, die im Kontrollflußgraphen *cfg* enthalten sind (also nur die über Sprungkanten erreichbaren Knoten), die Attribute *inS* und *out* . *in_cfg* ist die DFA-Information, die am Anfang des Eingangsknotens (*cfg.entry*) gültig ist. Die DFA-Information, die nach dem Ausgangsknoten (*cfg.exit*) gültig ist, wird als Resultat dieses Algorithmus an den Aufrufer dieser Prozedur zurückgegeben.
Dieser Algorithmus basiert auf dem Fixpunktalgorithmus 3.56 auf Seite 35.

```
repeat /* Fixpunktschleife */
  changed ← false;
  forall Basisblöcke b in cfg (in rPOSTORDER) do
    /* Die Basisblöcke können optional in rPOSTORDER Reihenfolge besucht werden. Die Fix-
       punktschleife konvergiert dann i.d.R. schneller. */
    old ← b.out ;
    b.inS ← ⊓_{p∈pred(b)} p.out;
    if b enthält eine mirParallel-Anweisung s then
      s.inS ← b.inS ; /* s ist die erste und einzigste Nichtsprunganweisung in b */
      s.out ← ⊤; /* Initialisiere für die folgende Schleife. */
      forall i⃗ ∈ s_perm(1, s.n) do /* Berechne f_PAR(x) mit Kor. 4.27(b). */
        var x : C ← s.inS ;
        /* Berechne den Wert eines Interleavings: */
        /* f_{p_{i_1};...;p_{i_k};...;p_{i_n}}(x) = f_{p_{i_n}}(...(f_{p_{i_k}}(...(f_{p_{i_1}}(x)...). */
        forall k ∈ i⃗ do
          x ← P_DFA^C_CFG (s.bodies[k] .cfg , x ); /* Berechne f_{p_{i_k}}(x) */
          /* Durch geeignete Datenstrukturen lassen sich bereits berechnete Paare (x_i, f_R(x_i))
             für jeden Prozeßrumpf R speichern. Damit können einige Aufrufe von P_DFA^C_CFG
             eingespart werden. */
        end
        s.out ← s.out ⊓ x ;
      end /* forall einfache Permutationen */
      b.out ← s.out
    else /* b enthält keine PAR-Anweisung. */
      b.out ← b.f (b.inS )
    end /* Enthält b eine PAR-Anweisung? */
    if old ≠ b.out then
      changed ← true
    end
  end /* forall Basisblöcke */
until changed = false

/* Berechne für alle Anweisung s dieses CFG's inS */
forall Basisblöcke b in cfg do
  var out : C ← b.inS ;
  forall Anweisungen s in b.stmts do
    s.inS ← out
    if s = mirParallel then
      out ← s.out ; /* Wurde bereits bestimmt. */
    else
      out ← s.f (s.inS );
    end
  end
end /* Berechne s.inS */
return cfg.exit.out ;
```

□

5.6.3 $\mathcal{D}^{\mathcal{B}}$-Probleme

Die folgenden Algorithmen sind o.B.d.A. für Kann-Vorwärts-DFA-Probleme formuliert.

5.10 ALGORITHMUS P_DFA$^{\mathcal{B}}$ (*procedure* : mirProcedure, *initial* : **bitset**)

Berechne die DFA-Information, die vor bzw. nach jedem Basisblock der ganzen Prozedur *procedure* gültig ist. *initial* ist der Wert, den das *inS* -Attribut des Eingangsknoten des Eingangs-CFG's der Prozedur *procedure* haben soll.
Ein Basisblock kann eine oder mehrere `mirParallel`-Anweisungen enthalten.

---

forall Basisblöcke *b* **in** *procedure.pcfg* **do** /* *Initialisiere DFA-Attribute* */
  b.inS ← ∅; *b.out* ← ∅; *b.gen* ← ∅; *b.kill* ← $\overline{\emptyset}$;
end
/* *Vorberechnung der gen und kill Mengen aller Anweisungen, Basisblöcke, Prozeßrümpfe und mirParallel-Anweisungen.* */
P_DFA$^{\mathcal{B}}$_GENKILL (*procedure.ptree*);

/* *Berechnung der Fixpunktlösung* */
P_DFA$^{\mathcal{B}}$_INOUT (*procedure.ptree* , *initial*); □

5.11 ALGORITHMUS P_DFA$^{\mathcal{B}}$_GENKILL (*ptree* : pTREE)

Berechne *gen* und *kill* für alle Anweisungen, Basisblöcke und Prozeßrümpfe, von Prozeßrümpfen und `mirParallel`-Anweisungen, die im Prozeßrumpfbaum mit Wurzel *ptree* enthalten sind.
Dazu wird der Prozeßrumpfbaum „von unten nach oben" traversiert. Die *gen* und *kill* Mengen der `mirParallel`-Anweisung werden gemäß Satz 4.39 berechnet. Anschaulich bedeutet „von unten nach oben", daß *gen* und *kill* für die „innersten" (tiefsten geschachtelten) `PAR`-Anweisungen zuerst berechnet werden.

---

if *ptree* = `mirParallel` **then**
  /* *Berechne gen und kill für die Prozeßrümpfe dieser mirParallel-Anweisung* */
  forall Prozeßrumpfdeskriptoren *p* **in** *ptree.bodies* **do**
    P_DFA$^{\mathcal{B}}$_GENKILL (*p*);
  end
  /* *Berechne gen und kill der mirParallel-Anweisung* */
  $ptree.gen \leftarrow \bigcup_{p \in ptree.bodies} p.cfg.exit.gen;$
  $ptree.kill \leftarrow (\bigcup_{p \in ptree.bodies} p.cfg.exit.kill) - ptree.gen;$
else /* *ptree ist ein Prozeßrumpfdeskriptor* */
  /* *Berechne gen und kill für alle mirParallel-Anweisungen, die in diesem Prozeßrumpf direkt enthalten sind.* */
  forall `mirParallel`-Anweisungen *s* **in** *ptree.pars* **do**
    P_DFA$^{\mathcal{B}}$_GENKILL (*s*);
  end
  /* *Jetzt sind gen und kill für alle in diesem Prozeßrumpf enthaltenen mirParallel-Anweisungen bestimmt. Berechne nun für alle anderen Anweisungen, jeden Basisblock und den ganzen Prozeßrumpf gen und kill* */
  P_DFA$^{\mathcal{B}}$_GENKILL_CFG (*cfg*);
end □

5.12 ALGORITHMUS P_DFA$^{\mathcal{B}}$_GENKILL_CFG (cfg : CFG)

Berechne für ein $\mathcal{D}^{\mathcal{B}}$ DFA-Problem die Transferfunktionen aller Anweisungen ($\neq$ `mirParallel`) und Basisblöcke, die in cfg enthalten sind. Berechne die Transferfunktion des ganzen Kontrollflußgraphen cfg .
Die *gen* und *kill* Mengen der in diesem CFG enthaltenen `mirParallel`-Mengen müssen vor Aufruf dieses Algorithmus bereits bestimmt sein.
Jeder Basisblock hat Bitvektorattribute *genOut* und *killOut* , die nur für diese Berechnung benötigt werden.
Als Ergebnis ist die Transferfunktion des gesamten CFG's durch *cfg.exit.gen* und *cfg.exit.kill* gegeben.

---

forall Basisblöcke *b* **in** *cfg* **do** /* *Initialisierung* */
  b.genOut $\leftarrow \emptyset$; *b.killOut* $\leftarrow \overline{\emptyset}$;
end
forall Basisblöcke *b* **in** *cfg* in rPOSTORDER Reihenfolge **do**
  Berechne *gen* und *kill* für jede Anweisung ($\neq$ `mirParallel`) aus *b* und für *b* selbst;
  /* *gen und kill einer Anweisung ($\neq$ `mirParallel`) wird gemäß dem zu lösenden DFA-Problem berechnet (siehe Abschnitt 3.4.4). Die gen und kill Mengen der in diesem CFG enthaltenen `mirParallel`-Mengen sind bereits bestimmt.*
  Die Gleichungen zur Bestimmung dieser Mengen für die Hintereinanderausführung von Anweisungen (siehe Gleichungen 3.5 auf Seite 35) bilden die Grundlage der Berechnung für einen Basisblock. */
  /* *Berechne gen und kill für den ganzen Prozeßrumpf.* */
  $b.genOut \leftarrow b.gen \cup (\bigcup_{p \in \mathrm{pred}(b)} p.genOut) - b.kill$;
  $b.killOut \leftarrow b.kill \cup (\bigcap_{p \in \mathrm{pred}(b)} p.killOut) - b.gen$;
end □

5.13 LEMMA

Gegeben ein $\mathcal{D}^{\mathcal{B}}$ Kann-Vorwärts DFA-Problem und ein Kontrollflußgraph cfg . Nach Ausführung des Algorithmus P_DFA$^{\mathcal{B}}$_GENKILL_CFG *gilt, daß cfg.exit.genOut zusammen mit cfg.exit.killOut die Transferfunktion bestimmen, die dem Kontrollflußgraphen cfg als ganzes entspricht.*

Der Aufwand zur Berechnung der Mengen ist linear in der Anzahl der Knoten, die in cfg enthalten ist. □

BEWEIS (LEMMA 5.13)

Ein die Mengen *genOut* bzw. *killOut* berechnender Fixpunktalgorithmus bestimmt diese Mengen für *alle* Knoten. Algorithmus P_DFA$^{\mathcal{B}}$_GENKILL_CFG hingegen berechnet diese Mengen nur für den Ausgangsknoten korrekt. Für alle anderen werden nur Teilmengen der korrekten Mengen bestimmt.

In der Berechnungsschleife werden die Basisblöcke in umgekehrter Postfixordnung besucht. Damit gilt für alle Voränger p eines Basisblocks b , daß wenn die Kante von $p \to b$ eine Vorwärtskante ist, die Mengen *gen, kill, genOut* und *killOut* von p bereits berechnet sind. Für Rückwärtskanten sind die Mengen noch nicht berechnet, sondern nur initialisiert. (Die Quellknoten von Rückwärtskanten werden zu einem späteren Zeitpunkt durch den Algorithmus besucht.) Bezeichnet $b.genOut^{+}$ den korrekten Wert, wenn für alle Vorgänger diese Mengen bereits berechnet wären, dann gilt somit $b.genOut \subseteq b.genOut^{+}$. (Da $\emptyset$ bzw. $\overline{\emptyset}$ die neutralen Element der Mengenvereinigung bzw. des -schnitts sind).

Sei p ein Vorgänger von b , so daß die Kante $p \to b$ eine Rückwärtskante ist. Die „fehlenden" Elemente der *b.genOut* Menge sind gerade diejenigen, die in $p.outGen = p.gen \cup p.genIn - p.kill$. enthalten sind. Da p im weiteren Verlauf des Algorithmus

noch besucht wird, werden die Elemente *p.gen* später „noch berücksichtigt". Die Elemente aus *p.genIn* stammen von bereits besuchten Knoten ab. Anloges gilt für *killOut* .

Da der Ausgangsknoten eines CFG's keine Nachfolger hat, kann er auch nicht Ziel einer Rückwärtskante sein. Damit sind die Mengen aller Voränger von *cfg.exit* berechnet, und somit bestimmen *cfg.exit.genOut* und *cfg.exit.killOut* die Transferfunktion, die dem Kontrollflußgraphen *cfg* entspricht.

Jeder Basisblock wird in beiden Schleifen nur einmal besucht, damit ist der Aufwand linear in der Anzahl der Basisblöcke des CFG's.

□

5.14 ALGORITHMUS P_DFA$^{\mathcal{B}}$_INOUT (*ptree* : pTREE, *initial* : **bitset**)

Berechne *inS* und *out* für alle Anweisungen, Basisblöcke und Prozeßrümpfe und `mirParallel`-Anweisungen, die im Prozeßrumpfbaum mit Wurzel *ptree* enthalten sind. *inital* ist der Wert, den die *inS* Menge des Eingangsknotens (falls *ptree* ein Prozeßrumpfdeskriptor ist) bzw. den die *inS* Menge der `mirParallel`-Anweisung haben soll (falls *ptree* eine `mirParallel`-Anweisung ist).

Dazu wird der Prozeßrumpfbaum „von oben nach unten" traversiert. Die *inS* und *out* Mengen der `mirParallel`-Anweisung werden gemäß Satz 4.39 berechnet.

---

if *ptree* = `mirParallel` **then**
  /* *Berechene* inS *und* out *für die Prozeßrümpfe dieser* `mirParallel`*-Anweisung.* */
  forall Prozeßrumpfdeskriptoren *p* **in** *ptree.bodies* **do**
    P_DFA$^{\mathcal{B}}$_INOUT (*p* , *initial*); /* *Am Anfang eines Prozeßrumpfes ist die DFA-Information gültig, die die* `mirParallel`*-Anweisung erreicht, vgl. Satz 4.39.* */
  end
else /* *ptree* *ist ein Prozeßrumpfdeskriptor* */
  P_DFA$^{\mathcal{B}}$_INOUT_CFG (*ptree.cfg* , *initial*); /* *Berechene mit dem iterativen Algorithmus die Fixpunktlösung.* */
  /* *Nun sind für alle in diesem CFG direkt enthaltenen* `mirParallel`*-Anweisungen* inS *und* out *berechnet. Diese Mengen müssen nun noch für die Prozeßrümpfe dieser* `mirParallel`*-Anweisungen bestimmt werden.*
  Die Information die vor einer `mirParallel`*-Anweisung gültig ist, ist auch am Anfang der Prozeßrümpfe dieser* `mirParallel`*-Anweisung gültig. (s. Satz 4.39).* */
  forall `mirParallel`-Anweisungen *s* **in** *ptree.pars* **do**
    P_DFA$^{\mathcal{B}}$_INOUT (*s* , *s.inS*);
  end
end □

5.15 ALGORITHMUS P_DFA$^{\mathcal{B}}$_INOUT_CFG (*cfg* : CFG, *in_cfg* : **bitset**)

Berechne die *inS* und *out* Mengen, von allen Basisblöcken aus *cfg* und allen Anweisungen dieser Basisblöcken. *in_cfg* ist die DFA-Information, die den Eingangsblock von *cfg* erreicht.

Die *gen* und *kill* Mengen aller Basisblöcke und aller Anweisungen müssen bereits berechnet sein. Die *out* Mengen müssen mit $\emptyset$ initialisiert sein.

Wir benutzen den iterativen Algorithmus (s. 3.56 auf Seite 35).

---

cfg.entry.inS ← *in_cfg* ;
repeat /* *Fixpunktschleife* */
  changed ← **false**;

```
    forall Basisblöcke b in cfg (in rPOSTORDER) do
      /* Die Basisblöcke können optional in rPOSTORDER besucht werden. Die Fixpunktschleife
         konvergiert dann i.d.R. schneller. */
      old ← b.out ;
      b.inS  ←  ⋃_{p∈pred(b)} p.out;
      b.out  ← b.gen ∪ b.inS − b.kill;
      if old ≠ b.out then
        changed ← true
      end
    end
  until changed = false

  /* Berechne für alle Anweisungen s dieses CFG's inS */
  forall Basisblöcke b in cfg do
    var out : bitset ← b.inS ;
    forall Anweisungen s in b do
      s.inS ← out ; s.out ← s.gen ∪ s.inS − s.kill; /* s. Gleichung 3.5 */
      out   ← s.out ;
    end
  end
```
□

5.7 Parallel Static Single Assignment Form

Die *Static Single Assignment Form* ist, wie in Abschnitt 3.4.5 festgestellt, nichts anderes als eine andere Darstellung der *Reaching-Definitions* Datenflußinformation. RD fällt in die Klasse der $\mathcal{D}^{\mathcal{B}}$ DFA-Probleme und ist ein Kann-Vorwärtsproblem. Wir können somit Satz 4.39 als Basis einer Erweiterung der SSA benutzen. Ziel ist eine der Definition 3.58 entsprechende Erweiterung der SSA-Form, der *Parallel Static Single Assignment Form* (pSSA), mit der auch parallele Programme dargestellt werden können.

Der Erweiterung liegt die folgende Idee zugrunde. Im sequentiellen Fall müssen ϕ-Zuweisungen an eine Variable x in einen Basisblock b immer dann eingefügt werden, wenn mehrere Definitionen von x diesen Basisblock b erreichen, d.h. mehrere Pfade vom Programmanfang bis zu diesem Basisblock enthalten Zuweisungen an x. Die ϕ-Funktion in der SSA-Darstellung eines Programms hat somit die dem DFA $\sqcap(\cup)$ Operator entsprechende Bedeutung.

Im parallelen Fall können wir wieder die beiden Fälle unterscheiden:

- welche DFA-Information ist nach einer PAR Anweisung gültig, und
- welche DFA-Information erreicht eine Anweisung.

Hierbei bedeutet „DFA-Information“ „Definitionen der Variablen x“. Die Sätze 4.39 und 4.35 und deren Folgerungen geben die Antwort auf diese Fragen. Wir müssen die, durch die parallele Ausführung entstehende DFA-Information, in der parallelen Erweiterung der SSA-Form darstellen. Dazu führen wir zwei, der obigen Unterscheidung entsprechende Funktionen ein, die ψ- und χ-Funktionen. Diese neuen Funktionen werden nur der Übersichtlichkeit wegen eingeführt, es könnte genauso alles mit einer einzigen Funktion, der ϕ-Funktion, erklärt werden.

5.7.1 Die ϕ-Funktion

Die Berechnung der Basisblöcke, die eine ϕ-Zuweisung für eine Variable v benötigen, geht von den Basisblöcken aus, in denen es „gewöhnliche“ Zuweisungen an v gibt (siehe [Cytron *et al.*,

1991], bzw. Algorithmus 5.19). Es stellt sich nun die Frage, welche Rolle eine `mirParallel`-Anweisung in einem Basisblock bezüglich der Zuweisungen an v spielt. Die Interleavingsemantik der `PAR`-Anweisung gestattet es, sie als eine Menge von Anweisungen aufzufassen, welche Werte an die in den Prozeßrümpfen vorkommenden Variablen zuweist. Hat also ein Prozeßrumpf eine Zuweisung an v, muß somit auch dem Basisblock b der die zugehörige `mirParallel`-Anweisung beinhaltet, eine Zuweisung an v zugerechnet werden. Diese Bemerkung wird in Algorithmus P_INSERT_ϕ durch die mit † markierten `REPEAT` Schleife implementiert.

5.7.2 Die ψ-Funktion

Die Übertragung der Gleichung

$$\prod_{p \in \mathsf{TopSorts}(k_1:P_1,\ldots,k_n:P_n)} f_p(x) = \prod_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{P_{i_n}} \circ \cdots \circ f_{P_{i_1}}) \overset{\text{mit Def 4.21}}{=} \prod_{\vec{\imath} \in \mathsf{s_perm}(1,n)} f_{P_{i_1};\ldots;P_{i_n}}(x)$$

aus Korollar 4.27(a) auf das RD-Problem bedeutet, daß nach der `PAR`-Anweisung all diejenigen Definitionen einer Variablen x gültig sind, die das Ende irgendeines Prozeßrumpfes dieser `PAR`-Anweisung erreichen. Damit jedes Benutzen einer Variablen x immer nur von einer Definition von x erreicht wird, müssen wir wieder, wie im sequentiellen Fall, eine neue Zuweisung $x_j := \psi(x_{i_1}, \ldots, x_{i_n})$ einfügen. Die Bedeutung der ψ-Funktion ist die gleiche wie die der ϕ-Funktion: sie führt die verschiedenen Definitionen $(x_{i_1}, \ldots, x_{i_n})$ zusammen. Da das Ende eines Prozeßrumpfes immer nur von genau einer SSA-Variablen x_{i_l} erreicht wird, hat die ψ-Funktion, die nach einer `PAR` Anweisung (mit n Prozeßrümpfen) für die Variable x eingefügt werden muß, nur n Argumente. Ihre Semantik ist: „wähle den Wert x_{i_l} der zur Laufzeit zuletzt ausgeführten Zuweisung an x aus“. Diese Zuweisung stammt aus dem Prozeßrumpf S_l. Folgen einer `PAR`-Anweisung mehrere ψ-Zuweisungen, muß immer der gleiche Prozeßrumpf ausgewählt werden. Es treten sonst Probleme auf, wie sie in Abbildung 5.6 für die χ-Funktion geschildert sind.

Eine Zuweisung $x_j := \psi(\ldots)$ wird unmittelbar nach der entsprechenden `PAR`-Anweisung eingefügt. Diese neue Zuweisung muß dann natürlich wie alle anderen Zuweisungen behandelt werden. Das Einfügen der ψ-Zuweisungen wird im gleichen Schritt wie die Numerierung der Variablen durchgeführt (s. Algorithmus P_RENAME_BASIC_BLOCK). Trifft man in einem Basisblock b auf eine `mirParallel`-Anweisung s, werden zuerst in allen Prozeßrümpfen von s die Variablen durch entsprechende SSA-Variablen ersetzt. Sind alle Prozeßrümpfe von s abgearbeitet, kennt man auch die ψ-Zuweisungen, die für s benötigt werden. Danach wird die Ersetzung in b mit der s folgenden Anweisung fortgesetzt. Wird die Numerierung in dieser Reihenfolge durchgeführt, sind einige ψ-Zuweisungen überflüssig. Es können folgende Fälle unterschieden werden (siehe folgende Abbildung): Eine ψ-Zuweisungen an die Variable x muß nicht eingefügt werden wenn:

- kein Prozeßrumpf eine Zuweisung an die Variable x enthält oder
- nur genau ein Prozeßrumpf Zuweisungen an x enthält.

$a_1 := 1;\ b_1 := 2;$

`PAR`

$b_2 := 3 * a_1;$	$c_2 := 4 * a_1;$
$c_1 := 5;$	

`END;`

$c_3 := \psi(c_1, c_2);$

$x_1 := a_1 + b_2 + c_3;$

Für die Variable a braucht keine ψ-Zuweisung eingefügt werden, da sie in keinem Prozeßrumpf verändert wird. Für b wird keine benötigt, da sie in nur genau einem Prozeßrumpf verändert wird.

Abbildung 5.4: Eine `PAR`-Anweisung mit eingefügten ψ-Funktionen.

Der vollständige Algorithmus besteht aus den beiden Routinen p_Rename_Basic_Block und p_Rename_Par_Stmt.

5.7.3 Die χ-Funktion

Die Übertragung der Gleichung

$$\mathsf{in}_s^{||} = \mathsf{in}_s^{;} \cup \bigcup_{t \in \mathsf{sibl}[s]} \mathsf{gen}_t$$

aus Satz 4.39 auf das RD-Problem bedeutet, daß die Anweisung s von Definitionen erreicht wird, die von parallel zu s oder immer vor s (d.h. sequentiell vor s) ausgeführten Anweisungen stammen. Damit jedes Benutzen einer Variablen x immer nur von einer Definition von x erreicht wird, müssen wir wieder, wie im sequentiellen Fall, eine neue Zuweisung $x_j := \chi(x_{i_1}, \ldots, x_{i_k})$ einfügen[6]. Die Bedeutung der χ-Funktion ist die gleiche, wie die der ψ-Funktion. Offensichtlich kann es sehr viele Argumente der χ-Funktion geben, da jede χ-Zuweisung in einem Prozeßrumpf als Argument in χ-Funktionen in dazu parallelen Prozeßrümpfen erscheint. Ein Beispiel ist in Abbildung 5.5 gegeben.

```
                    PAR
(1) a1 := 1;                | a2 := 2;
(2) b1 := 1;                | b2 := 2;
(3) a3 := χ(a1, a2, a5);    | a5 := χ(a1, a2, a3, a4);
(4) b3 := χ(b1, b2, b5);    | b5 := χ(b1, b2, b3, b4);
(5) x1 := a3 + b3;          | z1 := a5 − b5;
(6) a4 := χ(a3, a2, a5);    |
(7) b4 := χ(b3, b2, b5);    |
(8) y1 := a4 ∗ b4;          |
                    END
```

Jede χ-Zuweisung muß wie eine gewöhnliche Zuweisung behandelt werden, d.h. sie ist in $\bigcup_{t \in \mathsf{sibl}[s]} \mathsf{gen}_t$ enthalten. Z.B. kommt a_3 im Argument der χ-Zuweisung an a_5 vor und umgekehrt.

Abbildung 5.5: Eine PAR-Anweisung mit eingefügten χ-Funktionen.

Zur Vermeidung sehr langer Argumentlisten, könnte man nur „echte“ Definitionen, d.h. Zuweisungen, die im Quellprogramm stehen, als Argumente der χ-Funktion zulassen. Dann jedoch treten gravierende Fehler auf, wie sie in Abbildung 5.6 auf der nächsten Seite geschildert sind.

Eine andere Fehlerquelle durch inkonsistente Auswahl der Argumente der χ-Funktionen ist, daß eine Variable an sich selbst zugewiesen werden kann, und somit keinen definierten Wert hätte. In Abbildung 5.5 könnte a_3 via der Zuweisung an a_5 an sich selbst zugewiesen werden.

All diesen Schwierigkeiten entkommt man, indem auf die Argumente der ψ- und χ-Funktion verzichtet wird. Dies wird auch noch durch die Tatsache erleichtert, daß im Gegensatz zur ϕ-Funktion, Optimierungen aus der Kenntnis der Argumente der ψ- und χ-Funktion keinen Nutzen ziehen können. ψ- und χ-Funktionen sind somit nullstellige Funktionen, die für die Analyse eines Programms nur die Information zur Verfügung stellen, daß ein Programmpunkt durch die parallele Ausführung von Anweisungen von mehreren Definitionen erreicht wird. Eine Zuweisung $x_j := \chi()$ wird unmittelbar vor jeder Benutzung von x eingefügt. Die Details der Implementierung sind in Algorithmus p_Insert$_\chi$ beschrieben.

[6]Diese neue Zuweisung muß dann natürlich wie alle anderen Zuweisungen behandelt werden, s. Abbildung 5.5.

	PAR				PAR	
(1)	$a_1 := 1;$	$a_2 := 2;$		(1)	$a_1 := 1;$	$a_2 := 2;$
(2)	$b_1 := 1;$			(2)	$b_1 := 1;$	
(3)	$a_3 := \psi(a_1, a_2);$			(3)	$a_3 := a_2;$	
(4)	$x_1 := a_3 + b_1;$			(4)	$x_1 := a_3 + b_1;$	
(5)	$a_4 := \psi(a_1, a_2);$			(5)	$a_4 := a_1;$	
(6)	$y_1 := a_4;$			(6)	$y_1 := a_4;$	
	END				END	
	(a)				(b)	

Die Auswahlsemantik der χ-Funktionen ermöglicht, durch inkonsistente Auswahl, Wertekombinationen, die durch kein Interleaving erzeugt werden können. Tabelle (a) zeigt ein Programmfragment mit eingefügten χ-Funktionen. In Tabelle (b) hat die χ-Funktion bestimmte Definitionen ausgewählt. Die Interleavingsemantik der `PAR`-Anweisung ermöglicht folgende Ergebnisse: $(x = 2, y = a = 1), (x = 2, y = a = 2), (x = 3, y = a = 2)$. Kann nun jede χ-Funktion ihre Auswahl unabhängig von den anderen treffen, wäre folgendes möglich: in Zeile (3) wird die Definition a_2 und in Zeile (5) die Definition a_1 gewählt. Diese Auswahl liefert das, gemäß der Interleavingsemantik, unmögliche Resultat: $(x = 3, y = a = 1)$.

Abbildung 5.6: Inkonsistente Auswahl der Definitionen durch die χ-Funktion.

5.7.4 Algorithmen zur Bestimmung der pSSA-Nummern und pSSA-Funktionen

5.16 ALGORITHMUS P_SSA (*procedure* : `mirProcedure`)

Berechne die parallele Static Single Assignment Form für die Quellprozedur *procedure* . Für jeden Basisblock werden die SSA-Nummern aller Variablen vermerkt, die am Anfang (*ssa_nr_in*) und am Ende (*ssa_nr_out*) des Basisblockes gültig sind.

Eine Quellprogrammvariable v wird in der Zwischensprache CCMIR-P durch den Operator `mirObjectAddr(`$v, \ldots$`)` dargestellt. Der Wert von v ist durch den Zwischensprachausdruck `mirContent(mirObjectAddr(`$v, \ldots$`))` gegeben.
Die SSA Form wird durch das Attribut *ssa_nr:* **integer** des `mirObjectAddr` Operators dargestellt. Es repräsentiert die SSA-Nummer dieser Variablen.

var *CurVarNr* : **array** [`mirVariable`] **of integer** ← [0, ..., 0] ; /* *Initiale SSA-Nummern der Variablen.* */
var *LastVarNr* : **array** [`mirVariable`] **of integer** ← [-1, ..., -1] ; /* *Zuletzt vergebene SSA-Nummer einer Variablen, s. 5.19.* */

P_DOM (*procedure.pcfg*); /* *Berechne die parallele Dominanzrelation, Algorithmus 5.5.* */
forall Basisblöcke *root* **in** Prozeßeingangsblöcke in *procedure.pcfg* **do**
  P_DF (*root*); /* *Berechne die parallele Dominanzgrenzen, Algorithmus 5.6.* */
end
P_INSERT_ϕ (*procedure.pcfg*); /* *Füge ϕ-Zuweisungen ein, Algorithmus 5.17.* */
P_INSERT_χ (*procedure.pcfg*); /* *Füge χ-Zuweisungen ein, Algorithmus 5.18.* */
P_RENAME_BASIC_BLOCK (*procedure.pcfg.entry.entry* , *procedure* , *CurVarNr* , *LastVarNr*);
/* *Benenne die Programmvariablen in SSA Variablen um, d.h. berechne für jedes Auftreten einer Variable ihre SSA Nummer, Algorithmus 5.19.* */ □

5.17 ALGORITHMUS P_INSERT_ϕ (*procedure* : mirProcedure)
Füge ϕ-Zuweisungen in die Basisblöcke des pCFG's der Prozedur *procedure* ein.

Wir folgen dem Algorithmus aus [CYTRON *et al.*, 1991, Abb. 11]. Die äußerste Schleife geht über alle Variablen v, die in der zu analysierenden Prozedur *procedure* sichtbar sind. *IterCount* zählt in welcher Iteration man gerade ist. *WorkList* ist initial die Menge der Basisblöcke, die eine Zuweisung an v enthalten. Für jeden Basisblock b, der einmal in der *WorkList* enthalten ist, wird sichergestellt, daß alle Basisblöcke, die in seiner Dominanzgrenze liegen, eine ϕ-Zuweisung erhalten. Benötigt ein Basisblock b eine ϕ-Zuweisung, wird b in die *WorkList* aufgenommen (falls es nicht bereits schon mal darin enthalten war). Pro Iteration muß für jeden Basisblock vermerkt werden, ob er bereits bearbeitet wurde, und ob für die gerade betrachtete Programmvariable bereits eine ϕ-Zuweisung eingefügt wurde. Anstatt dazu boolesche Flags, die in jeder Iteration zurückgesetzt werden müssen, werden Integer-Array's (*HasAlready,Considered*) benutzt, die folgende Bedeutung haben: In Iteration *IterCount* , in der die Programmvariable v im Basisblock b betrachtet wird, gilt:

HasAlready $[b] <$ *IterCount*	In b ist noch keine ϕ-Zuweisung an v eingefügt worden.
HasAlready $[b] =$ *IterCount*	In b gibt es bereits eine ϕ-Zuweisung für v.
Considered $[b] <$ *IterCount*	b wurde in dieser Iteration noch nicht betrachtet.
Considered $[b] =$ *IterCount*	b wurde in dieser Iteration bereits betrachtet.

Der Unterschied zum sequentiellen Fall besteht darin, daß in die initiale *WorkList* neben den Basisblöcken, die eine Zuweisung an v im Quellprogramm enthalten, noch weitere Basisblöcke aufgenommen werden müssen: Für die Zwecke dieses Algorithmus, könnte die mirParallel-Anweisung aufgefaßt werden, als ob sie nur eine Folge der Zuweisungen ihrer Prozeßrümpfe wäre, die in nicht näher spezifizierter Weise ausgeführt werden. Daraus ergibt sich aber, daß wenn ein Basisblock v eine Zuweisung an die Variable v enthält, und b Teil eines Prozeßrumpfes einer mirParallel-Anweisung p ist, der Basisblock b_p der p enthält, ebenfalls eine Zuweisung an v enthält. Daher muß b_p ebenfalls in die initiale *WorkList* aufgenommen werden. Ist p Teil einer geschachtelten PAR-Anweisung, müssen natürlich alle umgebenden PAR-Anweisungen ebenso behandelt werden.

```
var IterCount : integer ← 0;
var HasAlready,Considered : array [mirBasicBlock] of integer ← [0, ... , 0] ;
var WorkList : list of mirBasicBlock ← ⟨⟩;

forall Variablen v in procedure do
  IterCount ← IterCount + 1;
  /* Initialisiere Arbeitsliste */
  forall Basisblöcke b in procedure.pcfg , b enthält eine Zuweisung an v do
    /* Ist b Teil eines Prozeßrumpfes, muß der Basisblock, der die zugehörige mirParallel-
       Anweisung enthält, ebenfalls in WorkList aufgenommen werden. */
    repeat /* † */
      if Considered[b] < IterCount then /* b ist noch nicht in der WorkList enthalten. */
        Considered[b] ← IterCount ;
        WorkList ← cons(WorkList , b );
      end
      b ← b.my_par_stmt_basic_block ; /* Ist b selbst Teil eines Prozeßrumpfes, liefere den
      Basisblock der die mirParallel Anweisung beinhaltet oder nil. */
    until b = nil /* d.h. b war nicht Teil eines Prozeßrumpfes */
  end
```

```
    while WorkList ≠ ⟨⟩ do /* Finden der Basisblöcke, die eine φ-Zuweisung für v benötigen. */
      var b : mirBasicBlock ← take(WorkList ); /* Nimm ein Element und entferne es. */
      forall Basisblock d in b.DF do
        if HasAlready[d] < IterCount then /* Es wurde noch keine φ-Zuweisung in b ein-
        gefügt */
          Füge „v : = φ(...);“ an den Anfang der Liste der Anweisungen d.stmts ein;
          HasAlready[d] ← IterCount ;
          if Considered[d] < IterCount then /* d in dieser Iteration noch nicht betrach-
          tet. */
            Considered[d] ← IterCount ;
            WorkList ← cons(WorkList , d );
          end
        end
      end
    end
  end
```
□

5.18 Algorithmus p_Insert_χ (*procedure* : mirProcedure)

Füge, falls nötig, vor dem Benutzen einer Variablen x eine χ-Zuweisung ein.

Als Vorberechnung, die den Zugriff auf die hier benötigten Daten beschleunigt, werden für jeden Basisblock die Menge der möglicherweise zu ihm parallel ausführbaren Basisblöcke und die Menge der Variablen, die in ihm verändert werden, bestimmt.

```
  forall Basisblöcke b in procedure.pcfg do
    if b.my_par_stmt ≠ nil then /* b ist Teil eines Prozeßrumpfes in einer PAR-Anweisung, und
    kann somit parallel zu anderen Basisblöcken ausgeführt werden. */
      forall Anweisungen stmt in b.stmts do
        forall Variablen var in Variablen, die von stmt benutzt werden do
          if var wird in einer zu stmt parallelen Anweisung verändert then
            Füge χ-Zuweisung „var : = χ();“ vor stmt in die Liste b.stmts ein.
          end
        end
      end
    end
  end
```
□

5.19 Algorithmus p_Rename_Basic_Block (*b* : mirBasicBlock, *procedure* : mirProcedure, *CurVarNr* : **array** [mirVariable] **of integer**, **var** *LastVarNr* : **array** [mirVariable] **of integer**)

Umbenenung der Variablen in SSA-Variablen, d.h. berechne für jedes Auftreten einer Variable v

- im Basisblock *b* ,
- in allen Kindern von *b* im Dominatorbaum mit Wurzel *b* und
- in allen Prozeßrümpfen, von mirParallel-Anweisungen, die in *b* vorkommen,

die SSA Nummer von v. Fügt nach einer mirParallel-Anweisung die nötigen ψ-Zuweisungen ein. Berechne *ssa_in_nr* , *ssa_out_nr* , die SSA-Nummern aller Variablen am Anfang bzw. am Ende der betrachteten Basisblöcke. Genauer: Gibt es in b eine ϕ-Zuweisung

an v, so ist `ssa_in_nr`[v] die SSA-Nummer, die v durch diese ϕ-Zuweisung erhält, ansonsten die SSA-Nummer, die v am Anfang von b hat. Eine Variable v wird im Basisblock `b` nicht verändert, wenn `b.ssa_in_nr`[v] = `b.ssa_out_nr`[v] . Diese Routine wird pro zu analysierender Prozedur mit dem Prozedureingangsknoten aufgerufen.

Wir folgen im Groben dem Algorithmus, der in [Cytron *et al.*, 1991, Abb. 12] für den sequentiellen Fall gegeben ist.
Die beiden Array's `CurVarNr`[v] und `LastVarNr`[v] speichern für die Variable v die SSA-Nummer, die sie an der „gegenwärtigen" Stelle im Basisblock hat, bzw. die SSA-Nummer, die zuletzt für v vergeben wurde. Die beiden Nummern müssen innerhalb eines Basisblocks nicht übereinstimmen. Der originale Algorithmus benutzt für jede Variable v einen Keller von SSA-Nummern statt der lokalen Array Variablen `CurVarNr` .
Wird an v ein neuer Wert zugewiesen, wird `LastVarNr`[v] inkrementiert und dieser Wert als neue SSA-Nummer für v benutzt. Betrachtet man den initialen Aufruf dieses Algorithmus, bedeutet die SSA-Nummer -1, daß v benutzt wird, bevor an v ein Wert zugewiesen wurde.
Eine Variable v hat in allen Kindern eines Basisblocks b im Dominatorbaum, am Basisblockanfang (noch vor den ϕ-Zuweisungen) die gleiche SSA-Nummer wie am Ende von b. Gibt es in CFG-Nachfolgern von b ϕ-Zuweisungen an v, hat das Argument, das b entspricht, ebenfalls die SSA-Nummer, die am Ende von b gilt.
Im parallelen Fall kann wie in p_Insert_ϕ argumentiert werden, daß die `mirParallel`-Anweisung wie eine Folge von Zuweisungen an Variablen betrachtet werden kann. Kommt also in `b` eine `mirParallel` Anweisung vor, müssen zuerst alle Prozeßrümpfe dieser Anweisung numeriert werden (mit dem Algorithmus p_Rename_Par_Stmt), bevor mit der nächsten Anweisung in `b` fortgefahren werden kann. p_Rename_Par_Stmt liefert als Ergebnis eine Liste der ψ-Zuweisungen, die für diese `mirParallel`-Anweisung nötig ist.

```
var sl : list of mirStmt ← ⟨⟩; /* Liste von ψ-Zuweisungen, die nach einer mirParallel-
Anweisung eingefügt werden muß. */
b.ssa_in_nr ← CurVarNr ;
/* Umbenennung der Variablen dieses Basisblockes */
forall Anweisungen stmt in Liste der Anweisungen von b.stmts do
  case stmt of
  • mirAssign:
    var v : mirVariable ← stmt.lhs.variable
    if stmt.rhs ∉ {φ, ψ, χ} then /* Umbenennung der rechten Seite gewöhnlicher Zuweis. */
      p_Rename_Expr (stmt.rhs , CurVarNr );
    end
    /* Erzeuge neue SSA-Variable, d.h. eine neue SSA-Nummer und benenne die linke Seite dieser
       Zuweisung um. */
    CurVarNr[v] , LastVarNr[v] ← LastVarNr[v] + 1;
    p_Rename_Expr (stmt.lhs , CurVarNr );
    if stmt.rhs = φ then /* b.ssa_in_nr[v] soll die SSA-Nummer sein, die durch diese
    Anweisung erzeugt wird. */
      b.ssa_in_nr[v] ← CurVarNr[v] ;
    end
```

- `mirParallel` :
  sl ← P_RENAME_PAR_STMT (*stmt* , *procedure* , *CurVarNr* , *LastVarNr*);
  /* *CurVarNr* wird hier modifiziert. Nur für Variablen, die in höchstens einem Prozeßrumpf dieser `mirParallel`-Anweisung verändert werden, hat *CurVarNr* den korrekten Wert. Alle anderen Variablen erhalten durch die folgenden ψ-Zuweisungen erneut korrekte SSA-Nummern. */
  Füge *sl* nach dieser Anweisung *stmt* in *b.stmts* ein.
- `mirIf`,`mirCase`:
  /* Umbenennen der Bedingungen der `mirIf` und `mirCase` Anweisungen. */
  P_RENAME_EXPR (*stmt.cond* , *CurVarNr*);

end
end

/* Umbenennung der Argumente der ϕ-Funktionen in allen Nachfolgern dieses Basisblocks. */
forall Basisblöcke *succ* **in** *b.succ* **do**
  forall ϕ-Zuweisungen $stmt_\phi$ **in** *succ* **do**
    var *j* : **integer** ← WhichPredecessor(*succ* ,*b*); /* *b* ist der *j* -te Vorgänger von *succ* . */
    $stmt_\phi$*.rhs.ops*[*j*] ← *CurVarNr*[v] ; /* Ersetze das *j* -te Argument dieser ϕ-Funktion durch die SSA-Variable V_i, wobei i die aktuelle Nummer der Variablen V ist. */
  end
end
b.ssa_out_nr ← *CurVarNr* ;

/* Besuche die Kinder im Dominatorbaum */
forall Basisblöcke *child* **in** Kinder des Dominatorbaumknotens *b* **do**
  P_RENAME_BASIC_BLOCK (*child* , *procedure* , *CurVarNr* , *LastVarNr*);
end □

5.20 ALGORITHMUS P_RENAME_EXPR (*expr* : `mirExpr`, *CurVarNr* : **array** [`mirVariable`] **of integer**)

Annotiere alle im Ausdruck *expr* vorkommenden Variablen v mit der SSA-Nummer, die durch *CurVarNr*[v] gegeben ist.

In der Zwischensprache CCMIR-P gibt es nur seiteneffektfreie Ausdrücke. Es gibt ein- und zweistellige arithmetische Operatoren und Konstanten.
Es gibt keine Unterschiede zwischen dem sequentiellen und parallelen Fall.

case *expr* **of**
- `mirObjectAddr`:
  expr.ssa_nr ← *CurVarNr*[*expr.variable*] ;
- `mirBinaryExpr`:
  P_RENAME_EXPR (*expr.left* ,*CurVarNr*); P_RENAMEEXPR (*expr.right* ,*CurVarNr*);
- `mirUnaryExpr`:
  P_RENAME_EXPR (*expr.value* , *CurVarNr*);
- `mirContent`:
  P_RENAME_EXPR (*expr.address* , *CurVarNr*);
- `mirConstExpr`:
  /* enthält keine Variablen */.

end □

5.21 ALGORITHMUS P_RENAME_PAR_STMT (*stmt* : mirStmt, *procedure* : mirProcedure, **var** *CurVarNr* : **array** [mirVariable] **of integer**, **var** *LastVarNr* : **array** [mirVariable] **of integer**) : **list of** mirStmt

Numeriere alle Variablen, die in den Prozeßrümpfen der mirParallel Anweisung *stmt* vorkommen. Liefere eine Liste von ψ-Zuweisungen, die für diese mirParallel-Anweisung benötigt werden. *CurVarNr* und *LastVarNr* haben die gleiche Bedeutung wie in Algroithmus P_RENAME-BASIC_BLOCK.

Wird v in keinem Prozeßrumpf verändert, ändern sich die Werte von *CurVarNr*[v] und *LastVarNr*[v] nicht; es wird keine ψ-Zuweisung erzeugt. Wird v nur in einem Prozeßrumpf modifiziert, wird keine ψ-Zuweisung benötigt; *CurVarNr*[v] und *LastVarNr*[v] ist dann die SSA-Nummer, die v am Ende dieses Prozeßrumpfes hat. Wird v in mehr als einem Prozeßrumpf zugewiesen, wird eine ψ-Zuweisung erzeugt, *CurVarNr*[v] und *LastVarNr*[v] haben die Werte, die v am Ende des zuletzt besuchten Prozeßrumpfes hat.

Alle Variablen in den Prozeßrümpfen haben initial die SSA-Nummern, die durch *CurVarNr* gegeben ist.

```
var ModificationCount : array [mirVariable] of integer ← [0,..., 0] ; /* Wieviele
Prozeßrümpfe von stmt verändern eine Variable v? */
var sl : list of mirStmt ← slist; /* Liste der ψ-Zuweisungen, die für stmt benötigt werden. */
var InitCurVarNr : array [mirVariable] of integer ← CurVarNr;

forall Prozeßrümpfe process in stmt.processes do
  var BeforeProcess_LastVarNr : array [mirVariable] of integer ← LastVarNr ;
        /* LastVarNr vor dem Besuch eines Prozeßrumpfes. */
  p_Rename_Basic_Block (process.EntryBasicBlock , InitCurVarNr , LastVarNr
  );

  /* Für welche Variablen braucht man ψ-Zuweisungen? */
  forall Variablen v in procedure do
    if LastVarNr[v] ≠ BeforeProcess_LastVarNr[v] then /* v in process verändert. */
      CurVarNr[v] ← LastVarNr[v] ;
      ModificationCount[v] ← ModificationCount[v] + 1;
      if ModificationCount[v] = 2 then /* v wurde in mindestens zwei Rümpfen modifi-
      ziert. */
        sl ← cons (sl , „v : = ψ();“); /* v braucht eine ψ-Zuweisung. */
      end
    end
  end
end
return sl ;
```
□

5.8 Schlußfolgerungen

Dieses Kapitel stellte die Datenstrukturen und Algorithmen der Programmanalyse für parallele Programme vor. Diese basieren auf den Methoden, die für sequentielle Programme benutzt werden. Die Korrektheit unserer Algorithmen ergibt sich aus der Theorie der Datenflußanalyse paralleler Programme und der Korrektheit der Analysealgorithmen für sequentielle Programme. Zusammen mit den Aufwandsabschätzungen aus Abschnitt 4.5 zeigt sich, daß die Analyse paralleler Programme sehr effizient durchgeführt werden kann.

Enthält ein paralleles Programm Synchronisationsanweisungen, muß an den hier vorgestellten Datenstrukturen und Analysealgorithmen nichts verändert werden. Aus Abschnitt 4.6 wissen wir, daß die Analyseresultate korrekt, aber u.U. zu pessimistisch sind.

6 Von der Theorie zur Praxis

Am Beispiel der Optimierung durch Codeverschiebung wird gezeigt, daß die „klassischen", auf der Bitvektor-DFA beruhenden Transformationen einfach auf parallele Programme übertragen werden können. Die Effektivität und Effizienz dieser Optimierung wird durch Messungen belegt.

6.1 Optimierung durch Codeverschiebung – Einleitung

Ziel der Optimierung durch **Codeverschiebung** (*Code Motion*) ist es,

1. die mehrfache Berechnung eines Ausdrucks innerhalb eines Basisblocks zu vermeiden (Elimination gemeinsamer Teilausdrücke[1]) [Cocke, 1970; Aho *et al.*, 1986] und
2. die Berechnung eines Ausdrucks von einer häufig ausgeführten Stelle (z.B. innerhalb einer Schleife) an eine weniger häufig ausgeführte (z.B. vor die Schleife) zu verschieben (Elimination partieller Redundanzen[2]) [Morel und Renvoise, 1979].

Dabei muß natürlich der gleiche Wert berechnet werden. Gleichzeitig darf die Berechnung nur auf Programmpfaden ausgeführt werden, auf denen sie im nicht optimierten Programm auch auftritt. Das Beispiel aus Abbildung 6.1 zeigt, daß es mehrere Einfügemöglichkeiten für verschiebbare Ausdrücke gibt:

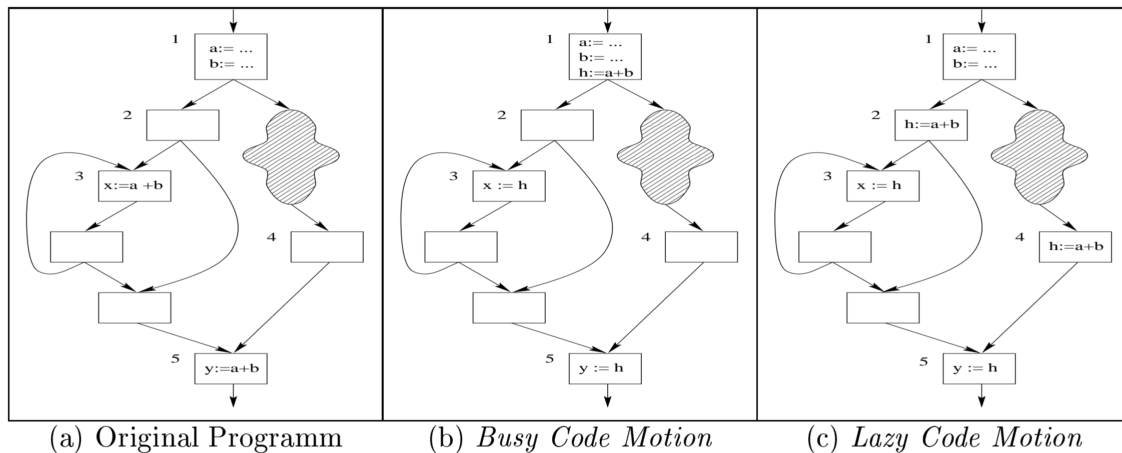
Die ***Busy Code Motion*** (**BCM**) plaziert den Ausdruck „so früh wie möglich", während die ***Lazy Code Motion*** (**LCM**) ihn „so spät wie möglich" einfügt. In beiden Fällen wird der Ausdruck $a + b$ zur Laufzeit gleich oft berechnet. Der Unterschied liegt in der „Lebensdauer" der Hilfsvariablen h. In der *Busy Code Motion* existiert sie innerhalb des schraffierten Teil-CFG's, ohne daß sie darin benutzt wird. Da diese Hilfsvariablen i.d.R. auf Maschinenregister abgebildet werden, erhöht sich der sogenannte *Registerdruck* und kann zu zusätzlichen Speicher- und Ladebefehlen führen.

Die Codeverschiebung wird durch zwei Ausdrucksmengen REPLACE und INSERT gesteuert. Die Berechnung eines Ausdrucks $\boldsymbol{expr}$ wird von den Basisblöcken, in denen $\boldsymbol{expr} \in$ REPLACE gilt, in die Basisblöcke verschoben, für die $\boldsymbol{expr} \in$ INSERT gilt. Verschieben bedeutet dabei, daß die Berechnung durch den Zugriff auf eine Hilfsvariable $h_{\boldsymbol{expr}}$ ersetzt wird, die in den Einfügebasisblöcken zugewiesen wird ($h_{\boldsymbol{expr}} := \boldsymbol{expr}$).

Morel und Renvoise [1979] formulieren die Aufgabe der Elimination partieller Redundanzen als ein bidirektionales Problem und geben einen Bitvektoralgorithmus an, welcher annähernd der *Busy Code Motion* entspricht. Mehrere Autoren beschäftigten sich mit den Nachteilen dieses Algorithmus, z.B. [Drechsler und Stadel, 1988; Dhamdhere, 1991; Dhamdhere und Harish, 1993].

[1] engl.: Common Subexpression Elimination, CSE

[2] Diese Redundanzen werden deshalb *partiell* genannt, da ein Ausdruck, obwohl er einen Programmpunkt nicht auf allen Pfaden erreicht, u.U. dennoch verschoben werden kann.



(a) Original Programm (b) *Busy Code Motion* (c) *Lazy Code Motion*

In diesem Beispiel wird nur der Ausdruck `a+b` betrachtet. Im Originalprogramm wird er nur in den Basisblöcken (3) und (5) berechnet. Die *Busy Code Motion* verschiebt die Berechnung in den Basisblock (1), die *Lazy Code Motion* in die Basisblöcke (2) und (4)

Abbildung 6.1: Lazy / Busy Code Motion

Eine grundlegende Verbesserung wird von Knoop *et al.* [1992, 1994] vorgestellt. Sie geben ein System von vier unidirektionalen DFA-Gleichungssystemen an, mit dem sowohl *Busy Code Motion* als auch *Lazy Code Motion* implementiert werden kann (s. Abbildung 6.2 und 6.3). Sie zeigen, daß ihr Verfahren der *Lazy Code Motion* berechnungsoptimal ist: Weder kann die Zahl der Berechnungen weiter reduziert, noch die Lebensdauer der benötigten Hilfsvariablen verringert werden.

Lazy Code Motion beschreibt nur die globalen Effekte der Codeverschiebung und nimmt an, daß alle lokalen Redundanzen bereits eliminiert wurden. Im nächsten Abschnitt wird die Implementierung der CSE für parallele Programme beschrieben, im darauf folgenden die der LCM.

6.2 Codeverschiebung und die PAR-Anweisung

Bei der Anwendung eines Verfahrens zur Codeverschiebung im Kontext paralleler Programme ergeben sich zwei neue Möglichkeiten der Codeverschiebung: Berechnungen können von der Umgebung der `PAR`-Anweisung in ihre Prozeßrümpfe hinein, oder Berechnungen können aus den Prozeßrümpfen heraus verschoben werden[3].

Wird eine Berechnung aus einem Prozeßrumpf heraus oder hinverschoben kann es zu den in [Knoop, Steffen und Vollmer, 1996] beschriebenen Annomalien führen. Diese Probleme können auf zwei Arten gelöst werden:

1. In [Knoop, Steffen und Vollmer, 1996] werden die DFA-Gleichungen für REPLACE und INSERT so modifiziert, daß diese Codeverschiebung nur noch unter bestimmten Bedingungen erlaubt sind.
2. Die Codeverschiebung aus einem Prozeßrumpf heraus oder in ihn hinein wird generell verboten. Dieses Verbot wird nicht in den Gleichungen ausgedrückt, sondern auf andere Art und Weise eingehalten. Damit können die Gleichungen unverändert (aus dem sequentiellen Kontext) übernommen werden.

[3]Analog können bei der interprozeduralen Codeverschiebung Berechnungen in eine aufgerufene Prozedur hinein oder aus ihr heraus gezogen werden.

Die zweite Lösung muß nicht unbedingt zu schlechtern Programmlaufzeiten führen, wenn man folgendes bedenkt: Damit eine Codeverschiebung möglichst wirksam ist, sollten die berechneten Werte in Prozessorregistern und nicht in Hilfsvariablen, die im Speicher allokiert sind, gehalten werden. Da aber die Register lokal zu einem Prozessor sind (d.h. andere Prozessoren können nicht auf sie zugreifen) können die Werte von Berechnungen, die über die Prozeßrumpfgrenzen hinaus verschoben werden sollen nicht in Registern gespeichert werden. Damit müssen sie in Hilfsvariablen gespeichert werden. Es entstehen somit zusätzliche Kosten für den Speicherzugriff auf den gemeinsamen Speicher.

Wir haben uns in unserer Implementierung (auch aus technischen Gründen, siehe Abschnitt 6.6) für den zweiten Ansatz entschieden. Die *Lazy Code Motion* konnte somit wie in [Knoop *et al.*, 1994] beschrieben benutzt werden kann.

6.3 Elimination gemeinsamer Teilausdrücke

Als Anwendung der pSSA-Form haben wir die vereinfachte Elimination gemeinsamer Teilausdrücke implementiert, die auf der syntaktischen Äquivalenz der Ausdrücke beruht. Der erste Schritt in der vereinfachten CSE ist es, die *syntaktisch* äquivalenten Ausdrücke im Programm zu finden (vgl. Abschnitt 3.4.5). Im zweiten Schritt wird festgestellt, welche Ausdrücke einer solchen Ausdrucksklasse den gleichen Wert berechnen[4]. Zum Schluß muß die erste Berechnung eines Ausdrucks einer neu eingeführten Hilfsvariablen h zugewiesen werden. Ebenso müssen alle Berechnungen der wertegleichen Ausdrücke durch einen Zugriff auf h ersetzt werden.

Wie in Abschnitt 3.4.5 bemerkt, gibt es (mindestens) zwei Möglichkeiten, die Wertegleichheit zu berechnen: einerseits mittels der *Reaching-Definitions*-Datenflußanalyse und andererseits durch Überführen des Programms in die *Static Single Assignment Form.* Wir haben uns für die SSA-basierte Methode entschieden (und implementiert), um die Anwendbarkeit mehrerer Methoden auf die Optimierung paralleler Programme zu zeigen[5].

6.1 Definition (Syntaktische Gleichheit und Wertgleichheit von Ausdrücken)
Zwei Ausdrücke e_1 und e_2 sind **syntaktisch gleich**, wenn

- die Operatoren der Ausdrücke (z.B. `mirPlus`[6] oder `mirCompare`) gleich sind und
- die Typen der Ausdrücke (z.B. `mirInteger`, `mirReal`) gleich sind, sowie
- die Operanden von e_1 syntaktisch gleich zu denen von e_2 sind und
- falls der Operator weitere Attribute hat, diese ebenfalls gleich sind (z.B. hat `mirCompare` ein Attribut `relation` und `mirIntConst` ein Attribut `value`). Die SSA-Nummer von Variablen wird bei der syntaktischen Gleichheit nicht berücksichtigt.

Zwei Ausdrücke e_1 und e_2 sind **wertgleich**, wenn

- e_1 und e_2 syntaktisch gleich sind und
- die Operanden von e_1 wertegleich zu denen von e_2 sind und
- gleiche Variablen die gleiche SSA Nummer haben. □

Mit diesen Gleichheitsbegriffen können Äquivalenzrelationen über Ausdrücken definiert werden. Da hier zwei Gleichheitsrelationen definiert sind, gibt es auch zwei Äquivalenzrelationen. Ein im Programm vorkommender Ausdruck wird **Berechnung** genannt und gehört genau einer syntaktischen und einer wertegleichen **Ausdrucksklasse** an. In der Implementierung spricht man von **Ausdruckstabellen**. Für die Bestimmung der syntaktischen

[4]In der Implementierung können diese beiden Schritte gleichzeitig ausgeführt werden.
[5]DFA-Gleichungen werden für die LCM benutzt.
[6]Wir benutzen hier wieder die CCMIR Notationen.

Äquivalenzklassen macht es offensichtlich keinen Unterschied, ob ein sequentielles oder paralleles Programm untersucht wird.

Da wir in der parallelen Form der SSA ψ- und χ-Zuweisungen in das Quellprogramm einfügen, welche die Effekte der parallelen Ausführung von Zuweisungen ausdrücken, kann diese Definition der Wertegleichheit direkt für parallele Programme übernommen werden. Damit unterscheidet sich ein Algorithmus zur Berechnung der Ausdruckstabellen für parallele Programme nicht von dem für sequentielle Programme!

Auch für die DFA-basierte Berechnung der Wertegleichheit unterscheidet sich der Algorithmus zur Erstellung der Ausdruckstabellen für sequentielle Programme nicht von dem für parallele Programme, da hier die $\mathsf{in}^{\|}$ Mengen die Rolle der SSA-Nummern übernehmen.

6.4 Lazy Code Motion

6.4.1 Beschreibung des Verfahrens

Die folgende Darstellung der *Lazy Code Motion* für sequentielle Programme richtet sich nach [Knoop *et al.*, 1994]. Eine Motivation dieses Verfahrens, der Beweis der Korrektheit und der Optimalität werden dort gegeben.

Von nun an wird angenommen, daß die CSE bereits durchgeführt wurde. Für die Darstellung wird nur eine einzige *syntaktische* Ausdrucksklasse betrachtet. Verschoben werden dann Berechnungen dieser Klasse. Die Gleichungssysteme der BCM (Abbildung 6.2) und LCM (Abbildung 6.3) basieren auf drei Grundprädikaten über Ausdrucksklassen (N-COMP, X-COMP und TRANSP), welche die in einem Basisblock vorkommenden Berechnungen charakterisieren. Diese werden für jeden Basisblock lokal, d.h. ohne Berücksichtigung der Vorgänger bzw. Nachfolger des Blockes, berechnet. Hat man für jeden Basisblock diese Prädikate berechnet, können die DFA-Gleichungssysteme für BCM und LCM gelöst werden. Kapitel 3 gibt dazu die nötigen Hilfsmittel für den sequentiellen Fall. Die folgenden Definitionen erläutern die dazu nötigen Begriffe.

6.2 Definition (Eingangs- Ausgangsberechnungen)
Eine **Modifikation** einer syntaktischen Ausdrucksklasse E ist eine Zuweisung an eine Variable v, die in E enthalten ist ($v \in \mathcal{TA}[E]$). Für E kann ein Basisblock b in zwei verschiedene Teile aufgespalten werden. In den

Eingangsteil, der alle Anweisungen vom Basisblockanfang bis zu und einschließlich der letzten Modifikation von E enthält und den

Ausgangsteil, der die restlichen Anweisungen enthält.

Bedingt durch die bereits durchgeführte CSE enthält jeder Eingangsteil höchstens eine Berechnung von E vor seiner ersten Modifikation, die **Eingangsberechnung**. Ein Ausgangsteil enthält ebenfalls höchstens eine Berechnung von E, die **Ausgangsberechnung**. Weitere Berechnungen von E sind von zwei Modifikationen umgeben, und sind somit irrelevant für die globale Codeverschiebung. Der **Einfügepunkt** im Eingangsteil ist unmittelbar vor einer Berechnung von E (falls es diese gibt), sonst unmittelbar vor der ersten Modifikation von E (falls es diese gibt) oder am Ende des Eingangsteils (falls es weder eine Eingangsberechnung noch eine Modifikation gibt). Der Einfügepunkt im Ausgangsteil ist unmittelbar vor der Ausgangsberechnung (falls es diese gibt) oder sonst am Ende des Ausgangsteils. □

6.3 Definition (Grundprädikate der BCM und LCM)
Für jeden Basisblock b und die syntaktische Ausdrucksklasse E werden die folgenden **Grundprädikate** definiert:

$\mathsf{TRANSP}_b[E]$ b enthält keine Modifikationen von E, d.h. b ist für E **transparent**.

1a) *Safety* Analyse: *Down-Safety*

$$\mathsf{N\text{-}D\text{-}SAFE}_b = \mathsf{N\text{-}COMP}_b + \mathsf{TRANSP}_b \cdot \mathsf{X\text{-}D\text{-}SAFE}_b$$

$$\mathsf{X\text{-}D\text{-}SAFE}_b = \mathsf{X\text{-}COMP} + \begin{cases} \mathsf{FALSE} & \text{wenn } n = exit \\ \prod\limits_{p \in \mathsf{succ}(b)} \mathsf{N\text{-}D\text{-}SAFE}_p & \text{sonst} \end{cases}$$

Berechne Fixpunkt: $\mathsf{N\text{-}D\text{-}SAFE}^*$ und $\mathsf{X\text{-}D\text{-}SAFE}^*$

1b) *Safety* Analyse: *Up-Safety*

$$\mathsf{N\text{-}U\text{-}SAFE}_b = \begin{cases} \mathsf{FALSE} & \text{wenn } n = n_0 \\ \prod\limits_{p \in \mathsf{pred}(b)} (\mathsf{X\text{-}COMP}_p + \mathsf{X\text{-}U\text{-}SAFE}_p) & \text{sonst} \end{cases}$$

$$\mathsf{X\text{-}U\text{-}SAFE}_b = \mathsf{TRANSP}_b \cdot (\mathsf{N\text{-}COMP}_b + \mathsf{N\text{-}U\text{-}SAFE}_b)$$

Berechne Fixpunkt: $\mathsf{N\text{-}U\text{-}SAFE}^*$ und $\mathsf{X\text{-}U\text{-}SAFE}^*$

2) *Earliestness* Analyse

$$\mathsf{N\text{-}EARLIEST}_b \stackrel{\text{def}}{=} \mathsf{N\text{-}D\text{-}SAFE}^*_b \cdot \prod_{p \in \mathsf{pred}(b)} \overline{(\mathsf{X\text{-}U\text{-}SAFE}^*_p + \mathsf{X\text{-}D\text{-}SAFE}^*_b)}$$

$$\mathsf{X\text{-}EARLIEST}_b \stackrel{\text{def}}{=} \mathsf{X\text{-}D\text{-}SAFE}^*_b \cdot \overline{\mathsf{TRANSP}_b}$$

Es muß kein Fixpunkt berechnet werden.

3) Einfüge- und Ersetzungspunkte für die *Busy Code Motion*

$$\mathsf{N\text{-}INSERT}^{BCM}_b \stackrel{\text{def}}{=} \mathsf{N\text{-}EARLIEST}_b$$
$$\mathsf{X\text{-}INSERT}^{BCM}_b \stackrel{\text{def}}{=} \mathsf{X\text{-}EARLIEST}_b$$

$$\mathsf{N\text{-}REPLACE}^{BCM}_b \stackrel{\text{def}}{=} \mathsf{N\text{-}COMP}_b$$
$$\mathsf{X\text{-}REPLACE}^{BCM}_b \stackrel{\text{def}}{=} \mathsf{X\text{-}COMP}_b$$

Es muß kein Fixpunkt berechnet werden.

$\overline{}$ und $\cdot$, $\prod$, bzw. $+$, $\sum$ stehen für die logische Negation, Konjunktion, bzw. Disjunktion. Der Präfix N- (X-) der Prädikatsnamen steht für *eNtry* (*eXit*), s. Definition 6.3. Die Fixpunktlösungen eines Gleichungsystems werden mit einem * an den Prädikatsnamen markiert.

Abbildung 6.2: Die *Busy Code Motion* DFA-Gleichungsysteme

1) Löse die *Busy Code Motion* Gleichungssysteme.
2) *Delayability* Analyse

$$\mathsf{N\text{-}DELAYED}_b = \mathsf{N\text{-}EARLIEST}_b + \begin{cases} \mathsf{FALSE} & \text{wenn } n = n_0 \\ \prod_{p \in \mathsf{pred}(b)} \overline{\mathsf{X\text{-}COMP}_p} \cdot \mathsf{X\text{-}DELAYED}_p & \text{sonst} \end{cases}$$

$$\mathsf{X\text{-}DELAYED}_b = \mathsf{X\text{-}EARLIEST}_b + \mathsf{N\text{-}DELAYED}_b \cdot \overline{\mathsf{N\text{-}COMP}_b}$$

Berechne Fixpunkt: $\mathsf{N\text{-}DELAYED}^*$ und $\mathsf{X\text{-}DELAYED}^*$

3) *Latestness* Analyse

$$\mathsf{N\text{-}LATEST}_b \stackrel{\text{def}}{=} \mathsf{N\text{-}DELAYED}_b \cdot \mathsf{N\text{-}COMP}_b$$

$$\mathsf{X\text{-}LATEST}_b \stackrel{\text{def}}{=} \mathsf{X\text{-}DELAYED}_b \cdot (\mathsf{X\text{-}COMP} + \sum_{p \in \mathsf{succ}(b)} \overline{\mathsf{N\text{-}DELAYED}_p^*})$$

Es muß kein Fixpunkt berechnet werden.

4) *Isolation* Analyse

$$\mathsf{N\text{-}ISOLATED}_b = \mathsf{X\text{-}EARLIEST}_b + \mathsf{X\text{-}ISOLATED}$$

$$\mathsf{X\text{-}ISOLATED}_b = \prod_{p \in \mathsf{succ}(b)} \mathsf{N\text{-}EARLIEST}_p + \overline{\mathsf{N\text{-}COMP}_p} \cdot \mathsf{N\text{-}ISOLATED}_p$$

Berechne Fixpunkt: $\mathsf{N\text{-}ISOLATED}^*$ und $\mathsf{X\text{-}ISOLATED}^*$

5) Einfüge- und Ersetzungspunkte für die *Lazy Code Motion*

$$\mathsf{N\text{-}INSERT}_b^{LCM} \stackrel{\text{def}}{=} \mathsf{N\text{-}LATEST}_b \cdot \overline{\mathsf{N\text{-}ISOLATED}_b^*}$$

$$\mathsf{X\text{-}INSERT}_b^{LCM} \stackrel{\text{def}}{=} \mathsf{X\text{-}LATEST}_b \cdot \overline{\mathsf{X\text{-}ISOLATED}_b^*}$$

$$\mathsf{N\text{-}REPLACE}_b^{LCM} \stackrel{\text{def}}{=} \mathsf{N\text{-}COMP}_b \cdot \overline{\mathsf{N\text{-}LATEST}_b \cdot \mathsf{N\text{-}ISOLATED}_b}$$

$$\mathsf{X\text{-}REPLACE}_b^{LCM} \stackrel{\text{def}}{=} \mathsf{X\text{-}COMP}_b \cdot \overline{\mathsf{X\text{-}LATEST}_b \cdot \mathsf{X\text{-}ISOLATED}_b}$$

Es muß kein Fixpunkt berechnet werden.

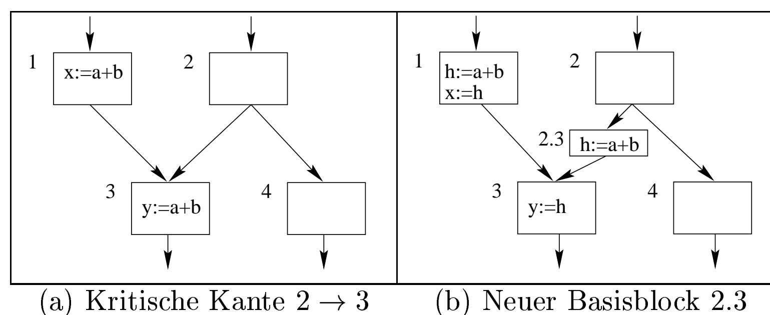
Abbildung 6.3: Die *Lazy Code Motion* DFA-Gleichungssysteme

N-COMP $_b[E]$ b enthält eine Eingangsberechnung e für E.
X-COMP $_b[E]$ b enthält eine Ausgangsberechnung e für E. □

Transformationen müssen sicher sein, d.h. Berechnungen dürfen nicht auf Pfaden eingeführt werden, auf denen sie vorher nicht existierten. Eine offensichtliche Quelle der Verhinderung der Transformation sind kritische Kanten:

6.4 Definition (Kritische Kanten)
Zwischen zwei Basisblöcken p und q existiert eine **kritische Kante** $p \to q$, wenn p direkter Vorgänger von q ist und p noch andere direkte Nachfolger und q weitere Vorgänger hat. □

Durch Einfügen neuer Basisblöcke können diese Kanten eliminiert werden. Der Algorithmus ist offensichtlich und in Abbildung 6.4 am Beispiel gezeigt. Bei der hier benutzten Definition eines parallelen Kontrollflußgraphen, ist klar, daß es für diesen Algorithmus keinen Unterschied zwischen sequentiellen und parallelen Programmen gibt.



(a) Kritische Kante $2 \to 3$ (b) Neuer Basisblock 2.3

Nur die Einfügung des neuen Basisblocks „$p.q$“ ermöglicht es, daß ein Ausdruck aus q „nach vorne“ verschoben werden kann. Würde im Beispiel der Ausdruck $a + b$ von Basisblock 3 nach 2 verschoben, würde $a + b$ im transformierten Programm auch auf Pfaden berechnet, auf denen er im Ausgangsprogramm nicht vorkommt (z.B. in Pfaden, die Basisblock 4 enthalten).

Abbildung 6.4: Kritische Kanten

6.4.2 Implementierung – Wie werden die Grundprädikate bestimmt?

In [Knoop *et al.*, 1994] werden keine Hinweise gegeben, wie die Grundprädikate X-COMP, N-COMP und TRANSP bestimmt werden können. Klar ist, daß eine explizite Aufspaltung der Basisblöcke in Eingangs- und Ausgangsteil für jede im Programm vorkommende syntaktische Ausdrucksklasse aus Effizienzgründen nicht vorgenommen werden kann.

Eine vereinfachte Form der RD- und LV-Analyse ist die Grundlage der Berechnung dieser Prädikate. Die Vereinfachung besteht darin, daß die RD/LV Informationen nur lokal für einen Basisblock b berechnet werden:

RD': Welche in b vorkommenden Definitionen von Variablen erreichen eine Berechnung e. Es wird dabei so getan, als ob jede Variable vor der ersten Programmanweisung in b initial definiert würde.

LV': Welche Definitionen von Variablen aus e erreichen das Ende von b.

Wird e unter dieser eingeschränkten Sicht nur von initialen Definitionen von Variablen $v \in \mathcal{TA}[e]$ erreicht, ist e eine Eingangsberechnung. Erreichen alle Definitionen von Variablen $v \in \mathcal{TA}[e]$, die e erreichen, das Ende von b, so ist e eine Ausgangsberechnung. Erreichen alle initialen Definitionen von Variablen einer syntaktischen Ausdrucksklasse E das Ende von b, ist der Basisblock für E transparent.

Statt nun eigens dafür eine DFA durchzuführen, können die bei der SSA-Transformation eingeführten SSA-Nummern benutzt werden: wir führen dazu für jeden Basisblock b zwei Tabellen $b.ssa_in_nr$ und $b.ssa_out_nr$ ein, die für jede Variable v die SSA-Nummer angibt, die sie am Anfang bzw. Ende des Basisblocks hat. „Am Anfang“ heißt dabei: Gibt es in b eine ϕ-Zuweisung an v, so ist $b.ssa_in_nr[v]$ die SSA-Nummer, die v durch diese ϕ-Zuweisung erhält, ansonsten die SSA-Nummer, die v bei der ersten Benutzung von v in b hat. Die Berechnung der SSA-Nummern durch Algorithmus 5.19 p_Rename_Basic_Block liefert diese Information ohne Mehraufwand. Offensichtlich gilt:

6.5 Satz (Berechnung der Grundprädikate)
Sei E eine syntaktische Ausdrucksklasse und b Basisblock, dann gilt:

- $\mathsf{TRANSP}_b[E] = \textbf{true} \iff \forall$ *Variablen* $v \in \mathcal{TA}[E] : b.ssa_in_nr[v] = b.ssa_out_nr[v]$. *Sind die Anfangs- und End-SSA-Nummern einer Variablen gleich, enthält b keine Zuweisung an sie.*
- $\mathsf{N\text{-}COMP}_b[E] = \textbf{true} \iff \exists e \in E$: *e ist eine Berechnung in b und* $\forall$ *Variablen* $v \in \mathcal{TA}[e] : v.ssa_nr = b.ssa_in_nr[v]$. *Sind diese SSA-Nummern am Anfang von b gleich wie in e, wurde an keine Variable v etwas zugewiesen. Analog dazu:*
- $\mathsf{X\text{-}COMP}_b[E] = \textbf{true} \iff \exists e \in E$: *e ist eine Berechnung in b und* $\forall$ *Variablen* $v \in \mathcal{TA}[e] : v.ssa_nr = b.ssa_out_nr[v]$. □

Dieser Satz ist die Basis für eine einfache Berechnung der Grundprädikate. In der Implementierung werden natürlich, mittels der Bitvektortechnik, die Menge aller Ausdrucksklassen betrachtet.

6.4.3 Übertragung auf parallele Programme

Zur Berechnung der Mengen TRANSP, N-COMP und X-COMP eines Basisblockes b müssen

1. alle Berechnungen des Basisblockes und
2. alle Modifikationen dieser Berechnungen bzw. aller syntaktischen Ausdrucksklassen

bekannt sein. Im parallelen Fall müssen wir neben den Anweisungen aus b auch noch die nebenläufig zu den Anweisungen aus b ausgeführten Zuweisungen betrachten, also Satz 4.35 auf das lokale *Reaching-Definitions*-Problem (RD') anwenden. Diese Information ist in unserer Implementierung durch die χ-Zuweisung bereits repräsentiert (s. Abschnitt 5.7.3).

Enthält ein Basisblock eine `mirParallel`-Anweisung, so berechnen die Prozeßrümpfe Ausdrücke und sind für Modifikationen von Ausdrücken „verantwortlich“. Zur einfacheren Bestimmung der TRANSP, N-COMP und X-COMP Mengen eines Basisblockes, der eine `mirParallel`-Anweisung enthält, fordern wir, daß diese Anweisung die einzige Nichtsprunganweisung dieses Basisblockes ist. Zur Berechnung der lokalen Mengen TRANSP, N-COMP und X-COMP eines eine `mirParallel`-Anweisung enthaltenden Basisblockes wird somit der Algorithmus 5.11, p_DFA$^{\mathcal{B}}$_GenKill, leicht modifiziert, angewendet (statt der lokalen Mengen gen und kill werden TRANSP, N-COMP und X-COMP berechnet). Zuweisungen der Prozeßrümpfe an Variablen werden durch ψ-Zuweisungen im Basisblock, der die `mirParallel`-Anweisung enthält, dargestellt (s. Abschnitt 5.7.1). Für alle anderen Basisblöcke kann Satz 6.5 zur Bestimmung der Mengen TRANSP, N-COMP und X-COMP benutzt werden.

Die Berechnung der Fixpunktlösungen der Prädikate (Mengen) durch den iterativ-rekursiven Algorithmus, muß wie in Abschnitt 4.5 beschrieben und vergleichbar zu Algorithmus 5.6.3 (p_DFA$^{\mathcal{B}}$) durchgeführt werden. Im Unterschied zu p_DFA$^{\mathcal{B}}$ ist die Vorberechnungsphase bei der *Delayability*- und *Isolation*-Analyse nicht notwendig, da die entsprechenden (für diese Analysen basisblocklokalen) Mengen bereits berechnet sind. Zur Bestimmung der Lösung der

Earliestness-, *Latestness*-Analysen und der Berechnung der Einfüge- und Ersetzungspunkte wird keine Fixpunktberechnung benötigt. Deshalb können die Basisblöcke des pCFG's in beliebiger Reihenfolge besucht werden.

6.5 Implementierung

Die Berechnung der „zusammenfassenden“ Gleichungen für EARLIEST, LATEST, INSERT und REPLACE sowie die Fixpunktberechnung wird im *Modula-P* Compiler mit einem Werkzeug namens *Optimix* [ASSMANN, 1995a,b] implementiert. *Optimix* erzeugt aus kurzen und prägnanten Spezifikationen der DFA-Gleichungen effiziente Algorithmen, s. Abbildung 6.5.

```
EARS ComputeDSAFE(all_exprs : EXPR_SET) /* N-D-SAFE und X-D-SAFE */
RANGE Block <= LIST(mirBasicBlock); /* Basisblöcke des CFG's */
BEGIN {*
  FORALL_BBS (ParamBlock, b) /* Erzeugen und initialisieren der Bitvektoren */
    mirBasicBlock_set_DSAFE_IN  (b, EXPR_SET_create(all_exprs, SETF_init_empty));
    mirBasicBlock_set_DSAFE_OUT (b, EXPR_SET_create(all_exprs, SETF_init_empty));
  END_FORALL_BBS
*}

RULES /* Gleichungssystem */
  DSAFE_IN (Block,Expr) :- COMP_IN(Block,Expr).
  DSAFE_IN (Block,Expr) :- TRANSP(Block,Expr), DSAFE_OUT(Block,Expr).
  DSAFE_OUT(Block,Expr) :- COMP_OUT(Block,Expr).
  DSAFE_OUT(Block,Expr) :- FORALL Succ: pcfg_succ(Block,Succ), DSAFE_IN(Succ,Expr).
```

Abbildung 6.5: Beispiel einer *Optimix* Regel.

Für jede syntaktische Ausdrucksklasse wird eine Hilfsvariable angelegt. Diese haben in der internen Darstellung eine Markierung, die sie als solche von anderen Programmvariablen unterscheidet. Überschneiden sich die Lebenszeiten dieser Hilfsvariablen nicht, ist der Registerallokator des mit dem Werkzeug *BEG* [EMMELMANN *et al.*, 1989; EMMELMANN, 1994] erzeugten Codegenerators in der Lage, mehrere Hilfsvariable auf ein Register abzubilden, ansonsten werden mehrere Register benutzt. Damit muß dieses Problem nicht bereits bei der Zuordnung Ausdrucksklasse / Hilfsvariable gelöst werden.

6.6 Messungen

Da kein Mehrprozessorsystem während der Arbeit zur Verfügung stand, wurde die nebenläufige Ausführung der Anweisungen mittels eines *POSIX Thread* Paketes[7] auf einem Ein-Prozessor SPARC Arbeitsplatzrechner unter SUNOS implementiert. Jeder Prozeßrumpf wurde dazu als separate Prozedur übersetzt, welche wiederum als nebeläufiger *Thread* gestartet werden konnte. Aus diesem technischen Grunde, musste eine Codeverschiebung in eine Prozeßrumpf hinein oder aus ihm heraus verboten werden.

Abbildung 6.6 beschreibt die vermessenen Programme kurz. Abbildung 6.7 zeigt die Laufzeitmessungen für einige parallele Programme. Sie belegen, daß die Optimierungen effektiv sind.

Von den parallelen Programmen wurden auch sequentielle Varianten erstellt, die das gleiche Problem mit einem ähnlichen Algorithmus lösen. Abbildung 6.8 zeigt die entsprechenden Laufzeitmessungen. Zum Vergleich wurden diese mit dem *Modula-2* Übersetzer *MOCKA* [SCHRÖER, 1988a] übersetzt. *MOCKA* implementiert ebenfalls CSE und die Elimination

[7]Entwickelt von Chris Provenzano, University of California, Berkeley.

partieller Redundanzen [Schröer, 1988b]. Diese vergleichende Messung zeigt, daß die Nebenläufigkeit einige Möglichkeiten der Optimierung verbietet. Sie belegt aber auch, daß die Möglichkeit, parallele Programme optimieren zu können, keine Nachteile für die Optimierung sequentieller Programme bzw. sequentieller Programmteile innerhalb von parallelen Programmen nach sich zieht.

Die Programme wurden einerseits so gewählt, daß sie sehr gut optimiert werden können aber andererseits auch die Grenzen aufzeigen, bzw. die optimierten Programme sogar langsamer werden als die nicht optimierten. Betrachtet man die Programme näher, bei denen die Optimierung sehr erfolgreich ist, so enthalten sie viele Adressberechnungen, die von Zugriffen auf ein- oder mehrdimensionale Felder stammen, die innerhalb geschachtelter Schleifen ausgeführt werden. Ein Verschieben einer solchen Berechnung (z.B. die Berechnung der Adresse von `c[i,j]` der Anweisung `c[i,j] := c[i,j] + a[i,k] * b[k,j]` einer Matrizenmultiplikation) aus einer Schleife heraus, wirkt sich somit besonders positiv aus. Bei den Gegenbeispielen liegt die Ursache darin, daß es sehrwohl gemeinsame Teilausdrücke gibt, die auch aus den Schleifen herausgezogen werden können, aber der zusätzliche Aufwand zu Zwischenspeicherung nicht amortisiert wird (z.B. weil die Schleifen nicht oft genug durchlaufen werden).

Problem	Beschreibung
welle	Elektromagnetische Wellenausbreitung in einem verlustbehafteten Dielektrikum über einem ein-dimensionalen Gitter.
pi	π wird durch Integration über dem Einheitskreis bestimmt. Die Integration erfolgt näherungsweise mit der Rechteckregel. Anhand dieses Beispiels werden in [Babb, 1988] verschiedene Programmiermethoden paralleler Rechnersysteme verglichen.
fft	Fast Fourier Transformation, siehe [Cormen *et al.*, 1990; JáJá, 1992].
sieb	Primzahlberechnung mit dem Sieb des Erathostenes. Als Grundlage der parallelen Version dient ein aus [Anderson, 88] übernommenes *Ada* Programm.
ProdCons	Das Produzenten-Konsumenten-Problem mit beschränktem Puffer.
BCD Zahlen	Addition zweier BCD codierter Ganzzahlen. Ein Prozeß addiert die Ziffern ohne Überträge, ein weiterer addiert die Überträge.
mm	Zwei Prozesse multiplizieren zwei Gleitkommamatrizen. Beide Prozesse lesen die gleichen Eingabematrizen und schreiben das Ergebnis in die gleiche Zielmatrix. Diese Aufgabe berechnet natürlich nichts Sinnvolles, ist vielmehr ein Modell von Berechnungen, bei der mehrere Prozesse in das selbe ARRAY schreiben und aus dem selben lesen.
Nur als sequentielle Version	
bench1	Ein Satz von Programmen (Permutationen berechnen, Türme von Hanoi, 8-Dame, Integer /Real Matrizenmultiplikation, QuickSort, BubbleSort, FFT), bekannt als der *Hennesy-Benchmark*.

Abbildung 6.6: Beschreibung der Beispiele

Parallele Programme			
Problem	Vorher (t_v)	Optimiert (LCM) (t_o)	Beschleunigung ($1 - t_o/t_v$)
welle	6.06	1.89	68%
pi	2.64	2.43	8%
fft	9.65	8.32	14%
sieb	2.50	2.38	5%
ProdCons	2.32	2.48	-7%
mm	10.06	4.34	57%
BCD Zahlen	3.20	2.80	13%

Abbildung 6.7: Messungen an parallelen Programmen (Zeiten in Sekunden)

Sequentielle Programme						
Problem	Vorher (t_v)		Optimiert (t_o)		Beschleunigung ($1 - t_o/t_v$)	
	cmc	mocka	cmc	mocka	cmc	mocka
welle	5.88	8.87	1.61	2.37	73%	73%
pi	2.81	3.09	2.42	2.64	14%	15%
fft	9.74	11.15	8.28	9.64	14%	14%
sieb	3.64	4.24	2.63	4.24	28%	0%
mm	8.86	16.55	4.17	9.03	53%	45%
BCD Zahlen	3.10	3.40	2.40	3.60	23%	-6%
bench1	1.83	1.69	1.61	1.48	12%	12%

cmc ist der in dieser Arbeit entwickelte *Modula-P* Compiler

Abbildung 6.8: Messungen an sequentiellen Programmen (Zeiten in Sekunden)

6.7 Schlußfolgerungen

Unsere Implementierung hat gezeigt, daß die Übertragung von Optimierungen, deren Analysen auf dem booleschen Halbverband basieren, auf parallele Programme mit wenigen, einfachen und systematischen Änderungen der Verfahren möglich ist. Zu beachten sind dabei die Effekte, die eine Codeverschiebung in einen Prozeßrumpf hinein oder aus ihm heraus haben können. Messungen belegen, daß die Optimierung paralleler Programme profitabel ist.

7 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, kontrollflußparallele Programme zu optimieren. Dazu wurden erstens die theoretischen Grundlagen hergeleitet, zweitens auf diesen basierend Datenstrukturen und Algorithmen entworfen bzw. erweitert und drittens, als Nachweis der Nützlichkeit, ein komplexes Optimierungsverfahren implementiert.

7.1 Das Problem

Diese Arbeit trägt zur Lösung der Probleme der *Analyse und Transformation kontrollflußparalleler Programme* bei. Kurz skizziert sind dies:

Die Standardtechniken können nicht direkt auf parallele Programme angewendet werden [Midkiff und Padua, 1990]. In der Praxis wird deshalb davon abgeraten, den Optimierer des Übersetzers zu benutzen [Parsytec, 1994]. Der Grund liegt in der Programmanalyse, die einer Optimierung vorangeht. Sie ist nur für sequentielle Programme definiert und liefert nur für diese korrekte Ergebnisse. Die Übertragung der Analysemethoden auf parallele Programme wird durch die, – durch Parallelität bedingte – Zustandsexplosion erschwert. Deren Ursache ist der Zugriff von mehreren parallel ablaufenden Prozesse auf den gemeinsamen Speicher. Damit die Datenflußanalyse (DFA) durch den Übersetzer überhaupt praktikabel wird, muß somit die Zahl der relevanten Zustände drastisch reduziert werden. Chow und Harrison [1994] stellen lapidar fest: „*The presence of concurrent constructs greatly complicate program analysis problems and challenge the current compiler technique*“.

7.2 Beitrag dieser Arbeit

Aufgrund einer Analyse des Begriffs der Korrektheit der Optimierung kontrollflußparalleler Programme wurde die Notwendigkeit sichtbar, bei der Programmanalyse alle (in der Anzahl der nebenläufig ausgeführten Anweisungen exponentiell vielen) Interleavings zu betrachten.

Für die Klasse der Bitvektor-Datenflußanalyseprobleme, welche die wichtigsten DFA-Probleme umfaßt, konnten wir zeigen, daß zur *Berechnung* der korrekten DFA-Information von parallelen Programmen die Interleavings *nicht* betrachtet werden müssen! Statt dessen genügt es, die Prozeßrümpfe separat zu analysieren und die Analyseresultate geeignet (mit Aufwand O(Anzahl der Prozeßrümpfe)) zusammenzufassen. Dieses Ergebnis beruht nicht auf speziellen Eigenschaften eines konkreten DFA-Problems, sondern nur auf Eigenschaften des booleschen Halbverbandes und unären, monotonen Funktionen über ihm.

Die sich aus dieser Theorie der Datenflußanalyse paralleler Programme (pDFA) ergebenden Sätze wurden dann auf die „Standardproblemformulierungen“ der DFA sequentieller Programme übertragen. Da die Begrifflichkeiten in diesem Bereich noch nicht feststehen, wurden „parallele Versionen“ einiger, im Bereich der Optimierung sequentieller Programme gängigen Begriffe und Methoden definiert, sowie die nötigen Algorithmen entworfen bzw. bereits existierende erweitert.

Mit diesen erweiterten Begriffen, Datenstrukturen und Methoden ausgestattet, wurde ein komplexes Optimierungsverfahren implementiert: die Elimination partieller Redundanzen (mittels *Lazy Code Motion*) und die dazu nötige Elimination gemeinsamer Teilausdrücke.

7.3 Der Lösungsweg

Unser Lösungsweg besteht im einzelnen aus den folgenden Schritten:

Auswahl des Maschinenmodells und der Programmiersprachkonzepte
Als Maschinenmodell wurde eine MIMD-Rechnerarchitektur mit gemeinsamem Speicher zugrunde gelegt. Die Programmiersprache, mit der ein solcher Rechner programmiert wird, basiert auf folgenden Konzepten: Spezifikation der Nebenläufigkeit im Stile der `cobegin/coend` und `forall do in parallel` Anweisungen; Prozesse können uneingeschränkt auf gemeinsamen Speicher zugreifen; es können dynamisch viele Prozesse zur Laufzeit erzeugt werden; diese können sich synchronisieren.

Zugrundelegen einer Semantik
Als Basis der konkreten Semantik und der für die Optimierung wichtigen abstrakten Semantik dienen die Interleavings, die die parallele Ausführung eines Programms kennzeichnen. Dies stellt für reale MIMD-Rechnersysteme, die üblicherweise ein CREW-Speicherverhalten aufweisen, keine Einschränkung dar.

Untersuchung des Begriffs der korrekten Optimierung paralleler Programme
Es wurde begründet, warum die Korrektheit einer Transformation unabhängig von der Art des Programms (sequentiell oder parallel) ist, solange bei der Programmanalyse nur alle Programmpfade betrachtet werden.

Untersuchung der Eigenschaften des booleschen Halbverbandes
Sie förderte die Eigenschaften eines Halbverbandes $(\mathcal{L}, \sqsubseteq, \sqcap)$ zutage, auf denen die Theorie der Datenflußanalyse paralleler Programme fußt:

1. Als semantische Funktionen, welche die Bedeutung einer Anweisung für die DFA beschreiben, betrachten wir nur die Identitäts- und Konstantenfunktionen.
2. Analog zu der Rolle, die der $\sqcap$-Operator für sequentielle Programme spielt, wird ein „neuer“ *Meet*-Operator ($\boxast$) über dem Funktionenraum hergeleitet und eingeführt, der die DFA- Bedeutung der `PAR`-Anweisung widerspiegelt.

Diese beiden Eigenschaften erlauben es, daß für die Berechnung der DFA-Information paralleler Programme keine Interleavings untersucht werden müssen.

Obwohl diese Bedingungen sehr restriktiv erscheinen, umfassen sie die große Menge der auf dem booleschen Halbverband basierenden Bitvektorprobleme. Diese bilden die Grundlage der wichtigsten (Rechenzeit verbessernden) Optimierungsmethoden wie Elimination gemeinsamer Teilausdrücke, Elimination toten Codes oder Elimination partieller Redundanzen.

Übertragung in die Praxis
Für den Optimierer wichtige Datenstrukturen und Algorithmen konnten nun theoretisch fundiert entwickelt werden: der parallele Kontrollflußgraph, die parallele *Static Single Assignment Form*, der iterativ-rekursive Algorithmus zur Berechnung des Fixpunktes der DFA-Gleichungsysteme.

Implementation
Mit diesen Methoden gewappnet, konnte die Implementierung zweier anspruchsvoller Optimierungen – Elimination gemeinsamer Teilausdrücke und Codeverschiebung partiell redundanter Ausdrücke – begonnen werden. Wie vermutet, unterscheidet sich die Implementierung für parallele Programme nur wenig von der für sequentielle. Der so

implementierte Übersetzer kann die sequentiellen Anteile eines parallelen Programms bzw. sequentielle Programme ohne Einbußen optimieren. Messungen belegen, daß es sich lohnt, parallele Programme zu optimieren.

Damit konnten wir unsere These belegen:

> Die Optimierung paralleler Programme ist effektiv und effizient, einfach zu implementieren und hat keine Nachteile für die Optimierung sequentieller Programme, sowie sequentieller Programmteile innerhalb eines parallelen Programms.

7.4 Ausblick

Mit der Lösung des Hauptproblems der DFA paralleler Programme, der Vermeidung der Zustandsexplosion, ergeben sich sofort neue Möglichkeiten außerhalb und innerhalb eines Übersetzers.

Die Idee, ein paralleles System durch die Interleavings der Ereignisse zu beschreiben, wird in vielen Gebieten innerhalb und außerhalb der Informatik angewendet. Kann von dem konkreten System ein Modell erstellt werden, das sich mit Funktionen aus $\mathcal{F}^{\mathcal{C}}$ oder $\mathcal{F}^{\mathcal{B}}$ beschreiben läßt, dann können unsere Ergebnisse verwendet werden, um Aussagen über diese Systeme zu treffen, ohne daß der vollständige Zustandsraum betrachtet werden muß.

Für die Optimierung eines Programms durch den Übersetzer sind weitere Verbesserungen möglich. Z.B. kann eine genauere Analyse der expliziten Synchronisation zur Verschärfung der DFA-Information benutzt werden. Diese ermöglicht dann die Anwendung weiterer Optimierungen. Verbesserungen sind auch im Bereich der Behandlung von Prozeduraufrufen möglich. Die Techniken der interprozeduralen Analyse (z.B. [Burke und Choi, 1992; Knoop *et al.*, 1992; Reps *et al.*, 1995]) sollten dazu auf parallele Programme übertragen werden.

Neben diesen Analyseaufgaben eröffnet die Parallelität der Hardware immer neue Möglichkeiten der Optimierungen: Moderne Prozessoren erlauben es, in die Steuerung des Speichersystems einzugreifen, und somit das von der Hardware realisierte Speicherkonsistenzmodell zu verändern [Gharachorloo *et al.*, 1990]. Unter dem Gesichtspunkt der einfachen Programmierbarkeit paralleler Rechner und der leichten Verständlichkeit der Programme wird ein *stark konsistentes Speichersystem* vorgezogen. Eine deutlich höhere Rechenleistung wird, allerdings zum Preis der schwierigeren Programmierbarkeit, mit einem *schwach konsistenten Speichersystem* erreicht. Für sequentielle Programme und Einprozessorsysteme unterscheiden sich beide Konzepte nicht; erst durch die Parallelität der Hard- und Software zeigen sich die Vorteile – und damit auch die Probleme.

Hier ist wieder der Übersetzerbau gefordert, die Lücke zwischen der sicheren und einfachen Programmierung einerseits, und der Ausnutzung aller Hardwaremerkmale durch geeignete Optimierungen andererseits, zu schließen.

ENDE

A Beweise

Dieser Anhang enthält die im Hauptteil ausgelassenen Beweise, die mit † markiert sind.

Beweis (Lemma 3.23 auf Seite 20)
Soll ein Interleaving aus den beiden Folgen erzeugt werden, muß entsprechend Punkt (3) der Definition 3.22 das erste Element aus einer der beiden Folgen entnommen werden und an das bestehende (initial leere) Interleaving angehängt werden. Im i-ten Schritt ($0 < i \leq \min(|q'| + |q''|)$) können somit 2^i viele verschiedene Anfänge eines Interleavings erzeugt werden. Sei o.B.d.A. $|q'| \leq |q''|$. Wählt man in den ersten $|q'|$ Schritten jedesmal das erste Element von q' aus, so gibt es für die Elemente aus q'' keine weiteren Wahlmöglichkeiten (die Elemente aus q' sind alle bereits „aufgebraucht"). Damit hat die Menge aller Interleavings zumindest $2^{\min(|q'|,|q''|)}$ Elemente.
□

Beweis (Korollar 4.4 auf Seite 46)
Halbverbände mit nur einem Element werden per Definition nicht betrachtet. Da $\mathcal{F}^{\mathcal{B}}$ (bis auf Umbenennung) der einzige mit $(\mathcal{B}, \sqcap)$ assoziierte monotone Funktionenraum ist (s. Definition 3.51), haben alle anderen Halbverbände, denen ein monotoner Funktionenraum assoziiert ist, mehr als zwei Elemente. Da der boolesche Halbverband ein Einselement hat, folgt die Behauptung mit den beiden Sätzen 4.2 und 4.3.
□

Beweis (Satz 4.9 auf Seite 47)
Für die erste Behauptung müssen wir zeigen:

Monotonie Seien f, g monoton. Mit Definition 3.39 ist zu zeigen: $\forall x, y \in \mathcal{L} : x \sqsubseteq y \implies h(x) \sqsubseteq h(y)$. Da $x \sqsubseteq y \iff x \sqcap y = x$ gilt, muß $x \sqsubseteq y \implies h(x) \sqcap h(y) = h(x)$ gezeigt werden.
Sei $x \sqsubseteq y$, dann gilt:

$$h(x) \sqcap h(y) = f(x) \sqcap g(x) \sqcap f(y) \sqcap g(y)$$

Da f, g monoton sind gilt: $f(x) \sqcap f(y) = f(x)$ und $g(x) \sqcap g(y) = g(x)$

$$= f(x) \sqcap g(x)$$
$$= h(x)$$

Distributivität Seien f, g distributiv. Mit Definition 3.39 ist zu zeigen: $\forall x, y \in \mathcal{L} : h(x \sqcap y) = h(x) \sqcap h(y)$.
$h(x \sqcap y) = f(x \sqcap y) \sqcap g(x \sqcap y) = f(x) \sqcap g(x) \sqcap f(y) \sqcap g(y) = h(x) \sqcap h(y)$

Die zweite Behauptung ergibt sich aus Satz 4.6 und Lemma 3.47.
□

Beweis (Satz 4.16 auf Seite 49)

Da für $\mathcal{F}^{\mathcal{B}}$ der $\sqcap$-Operator abgeschlossen ist, gilt $f \sqcap g \in \mathcal{F}^{\mathcal{B}}$. Mit Satz 4.10:

$$f \boxast (g \sqcap h) = f \circ (g \sqcap h) \sqcap (g \sqcap h) \circ f$$

Die Funktionen aus $\mathcal{F}^{\mathcal{C}}$ sind distributiv, und mit Def 4.1:

$$= f \circ g \sqcap f \circ h \sqcap g \circ f \sqcap h \circ f$$

Mit Satz 4.10:

$$= (f \boxast g) \sqcap (f \boxast h)$$

Die zweite Behauptung wird mittels Fallunterscheidung gezeigt

f	g	h	$f \sqcap (g \boxast h)$	$(f \sqcap g) \boxast (f \sqcap h)$
id	id	id	id	id
id	id	const_{c_h}	$\mathsf{id} \sqcap \mathsf{const}_{c_h}$	$\mathsf{id} \sqcap \mathsf{const}_{c_h}$
id	const_{c_g}	const_{c_h}	$\mathsf{id} \sqcap \mathsf{const}_{c_g} \sqcap \mathsf{const}_{c_h}$	$\mathsf{id} \sqcap \mathsf{const}_{c_g} \sqcap \mathsf{const}_{c_h}$
const_{c_f}	id	id	$\mathsf{const}_{c_f} \sqcap \mathsf{id}$	$\mathsf{const}_{c_f} \sqcap \mathsf{id}$
Für den folgenden Fall müssen die Funktionen separat untersucht werden[1]				
const_{c_f}	id	const_{c_h}	$\mathsf{const}_{c_f} \sqcap \mathsf{const}_{c_h}$	$(\mathsf{const}_{c_f} \sqcap \mathsf{id}) \boxast (\mathsf{const}_{c_f} \sqcap \mathsf{const}_{c_h})$
$\mathsf{const}_{\top}$	id	$\mathsf{const}_{\top}$	$\mathsf{const}_{\top}$	$\mathsf{const}_{\top}$
$\mathsf{const}_{\top}$	id	$\mathsf{const}_{\perp}$	$\mathsf{const}_{\perp}$	$\mathsf{const}_{\perp}$
$\mathsf{const}_{\perp}$	id	$\mathsf{const}_{\top}$	$\mathsf{const}_{\perp}$	$\mathsf{const}_{\perp}$
$\mathsf{const}_{\perp}$	id	$\mathsf{const}_{\perp}$	$\mathsf{const}_{\perp}$	$\mathsf{const}_{\perp}$
const_{c_f}	const_{c_g}	const_{c_h}	$\mathsf{const}_{c_f} \sqcap \mathsf{const}_{c_g} \sqcap \mathsf{const}_{c_h}$	$\mathsf{const}_{c_f} \sqcap \mathsf{const}_{c_g} \sqcap \mathsf{const}_{c_h}$

□

Beweis (Korollar 4.24 auf Seite 51)

Der Beweis folgt dem von Satz 4.23, weshalb wir die Beweisschritte nur kurz skizzieren:

- Sind alle $f_{p_i} = \mathsf{id}$, dann gilt $\prod_{p \in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)} f_p = \mathsf{id}$. Damit ist $\mathcal{CONST} = \emptyset$ Per Definition gilt: $\prod_{p \in \mathcal{CONST}} f_p = \mathsf{id}$, und somit die Behauptung.
- Andernfalls gibt es mindestens ein p_i mit $f_{p_i} \neq \mathsf{id}$, damit ist $\mathcal{CONST} \neq \emptyset$. Für alle p_j für die $f_{p_j} = \mathsf{id}$ gilt, muß $\forall s \in p_j : f_s = \mathsf{id}$ gelten. Somit ändern diese Anweisungen s den Wert eines Interleavings $p \in \mathsf{TopSorts}(k_1 : p_1, \ldots, k_n : p_n)$ nicht.

  Dehalb müssen nur die p_i betrachtet werden, für die $p_i \in \mathcal{CONST}$ gilt.

  Wie im Beweis zu Satz 4.23 gibt es für jedes $p_i \in \mathcal{CONST}$ eine Anweisung s, so daß $f_{p_i} = f_s$. Da es in $\mathsf{TopSorts}$ ein Interleaving p gibt mit $f_p = f_s$ und $\forall s' \in p$ mit $s \ll_p s' : f_{s'} = \mathsf{id}$ (s. Punkt (2) von Beweis zu Satz 4.23), gilt:

$$\prod_{p \in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)} f_p = \prod_{p \in \mathcal{CONST}} f_p$$

□

Für den Beweis von Korollar 4.27 benötigen wir noch ein kleines Lemma:

A.1 Lemma

Seien $f_1, \ldots, f_{n+1} \in \mathcal{F}^{\mathcal{C}}$ dann gilt (mit Korollar 4.13):

$$f_1 \boxast \ldots \boxast f_{n+1} = f_{n+1} \boxast \Big(\prod_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{i_n} \circ \cdots \circ f_{i_1}) \Big)$$

Da $\prod_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{i_n} \circ \cdots \circ f_{i_1}) \in \mathcal{F}^{\mathcal{C}}$ darf Satz 4.10 angewendet werden:

$$= f_{n+1} \circ \Big(\prod_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{i_n} \circ \cdots \circ f_{i_1}) \Big) \sqcap \Big(\prod_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{i_n} \circ \cdots \circ f_{i_1} \circ f_{n+1}) \Big) \qquad (\dagger)$$

[1] Beachte: $\sqcap$ und $\boxast$ sind kommutativ und $\mathsf{id} \sqcap \mathsf{const}_{\top} = \mathsf{id}$ und $\mathsf{id} \sqcap \mathsf{const}_{\perp} = \mathsf{const}_{\perp}$

Andererseits gilt Korollar 4.13, und somit

$$= \bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n+1)} (f_{i_{n+1}} \circ \cdots \circ f_{i_1})$$

□

Beweis (Korollar 4.27 auf Seite 52)

Für die Bauptung a) ist

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:P_1,\ldots,k_n:P_n)} f_p = \bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{P_{i_n}} \circ \cdots \circ f_{P_{i_1}})$$

zu zeigen. Der Beweis wird mittels vollständiger Induktion über n geführt. Für $n = 2$ gilt die Behauptung mit Satz 4.26.

Die Behauptung b) folgt mit Korollar 4.13) aus a), wenn ein $\mathcal{D}^{\mathcal{B}}$ DFA-Rahmen zugrunde liegt.

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1:P_1,\ldots,k_{n+1}:P_{n+1})} f_p =$$

Mit Definition von TopSorts.

$$= \bigsqcap_{p' \in P_{n+1}} (\bigsqcap_{p'' \in \mathsf{TopSorts}(P_1,\ldots,P_n)} (\bigsqcap_{p \in \mathsf{TopSorts}(p',p'')} f_p))$$

Mit Satz 4.26.

$$= \bigsqcap_{p' \in P_{n+1}} (\bigsqcap_{p'' \in \mathsf{TopSorts}(P_1,\ldots,P_n)} (f_{p'} \circ f_{p''} \sqcap f_{p'} \circ f_{p''}))$$

Distributivität der Funktionen aus $\mathcal{F}^{\mathcal{C}}$.

$$= \bigsqcap_{p' \in P_{n+1}} (f_{p''} \circ (\bigsqcap_{p'' \in \mathsf{TopSorts}(P_1,\ldots,P_n)} f_{p''}) \sqcap (\bigsqcap_{p'' \in \mathsf{TopSorts}(P_1,\ldots,P_n)} (f_{p''} \circ f_{p'})))$$

Anwenden der Induktionsvoraussetzung.

$$= \bigsqcap_{p' \in P_{n+1}} (f_{p''} \circ (\bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{P_{i_n}} \circ \cdots \circ f_{P_{i_1}})) \sqcap \bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n)} (f_{P_{i_n}} \circ \cdots \circ f_{P_{i_1}} \circ f_{p''}))$$

Mit Gleichung † aus Lemma A.1.

wobei: $Q_{i_j} = P_{i_j}$, wenn $1 \leq i_j \leq n$ und $Q_{i_j} = p''$, wenn $i_j = n+1$.

$$= \bigsqcap_{p' \in P_{n+1}} (\bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n+1)} (f_{Q_{i_{n+1}}} \circ \cdots \circ f_{Q_{i_1}})$$

Distributivität der Funktionen aus $\mathcal{F}^{\mathcal{C}}$.

$$= \bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n+1)} (f_{P_{i_{n+1}}} \circ \cdots \circ f_{P_{i_1}})$$

In diesem letzten Schritt geht allerdings die Eigenschaft verloren, daß $\bigsqcap_{\vec{\imath} \in \mathsf{s_perm}(1,n+1)} (f_{P_{i_{n+1}}} \circ \cdots \circ f_{P_{i_1}}) \in \mathcal{F}^{\mathcal{C}}$ im allgemeinen gilt, da für Funktionen aus $\mathcal{F}^{\mathcal{C}}$ $\sqcap$ nicht abgeschlossen ist.

□

Beweis (Satz 4.29 auf Seite 53)

Gegeben ein $s \in p_i$, zu zeigen ist:

$$\{p \downarrow s \mid p \in \mathsf{TopSorts}(k_1 : p_1, \dots, k_n : p_n)\} = \qquad (*)$$
$$\mathsf{TopSorts}(k_1 : \mathsf{prefixes}(p_1), \dots, k_{i-1} : \mathsf{prefixes}(p_{i-1}), (k_i - 1) : \mathsf{prefixes}(p_i),\ \ p_i \downarrow s, k_{i+1} : \mathsf{prefixes}(p_{i+1}), \dots k_n : \mathsf{prefixes}(p_n))$$

Abkürzend definieren wir: $\mathcal{A}_s \stackrel{\text{def}}{=} \{p \downarrow s \mid p \in \mathsf{TopSorts}(k_1 : p_1, \dots, k_n : p_n)\}$. Wir beweisen durch Fallunterscheidung:

$\subseteq$ Sei ein $p \in \mathcal{A}_s$ gegeben. Wir nehmen $k_i = 1$ an. Sei q_j für alle $j \in \{1, \dots, n\}$ definiert als $q_j \stackrel{\text{def}}{=} p|_{p_j}$, die auf p_j eingeschränkte Anweisungsfolge p. Da TopSorts die relativen Ordnungen der Anweisungen der einzelnen p_j in p erhält, haben in q_j die Anweisungen die gleiche Reihenfolge wie in p_j. Damit ist $q_j \in \mathsf{prefixes}(p_j)$. Bevor s ausgeführt werden kann, müssen alle Anweisungen t mit $t \ll_{p_i} s$ ausgeführt werden. Damit enthält $q_i = p|_{p_i}$ offensichtlich alle Anweisungen aus $p_i \downarrow s$. Somit gilt $p \in \mathsf{TopSorts}(\mathsf{prefixes}(p_1), \dots, p_i \downarrow s, \dots, \mathsf{prefixes}(p_n))$.

Für $k_i > 1$ gilt die Behauptung ebenfalls. Man stellt sich vor, daß alle Anweisungen $t \in p_j$ mit Markierungen versehen sind. Diese sind so gewählt, daß festgestellt werden kann, an welcher Argumentposition von TopSorts die Anweisungsfolge p_j vorkam. (Ein repliziertes p_j ist mehrfaches Argument von TopSorts). Nun kann, wie im nicht replizierten Fall geschlossen werden, indem $q_j^l \stackrel{\text{def}}{=} p|_{p_j}^l$ definiert wird, wobei l die Argumentposition bezeichnet.

$\supseteq$ Zu jedem $1 \le j \le n$ wählen wir ein $q_j \in \mathsf{prefixes}(p_j)$ aus. Ist p_j, $j \ne i$ repliziert werden k_j Folgen q_j^l mit $1 \le l \le k_j$ ausgewählt. Offensichtlich gilt für alle $q \in \mathsf{TopSorts}(q_1, \dots, q_j^1, \dots, q_j^{k_j}, \dots, p_i \downarrow s, \dots, q_n)$, daß $q \in \mathcal{A}_s$ ist. Wenn p_i repliziert (also $k_i > 1$) ist, werden $k_i - 1$ Prefixe von p_i „hinzugenommen“: Für $q \in \mathsf{TopSorts}(q_1, \dots, q_j^1, \dots, q_j^{k_j}, \dots, q_i^1, \dots, q_i^{k_i - 1}, p_i \downarrow s, \dots, q_n)$ gilt immer noch: $q \in \mathcal{A}_s$.

□

Beweis (Lemma 4.36 auf Seite 55)

Wir zeigen für $n = 2$

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1 : \mathsf{prefixes}(p_1), k_2 : \mathsf{prefixes}(p_2))} f_p = \mathsf{id} \sqcap \bigsqcap_{\substack{t \in p \in \{p_1, p_2\} \\ f_t = \mathsf{const}}} f_t$$

Die Verallgemeinerung auf beliebiges n kann mit Induktion über n bewiesen werden.

$$\bigsqcap_{p \in \mathsf{TopSorts}(k_1 : \mathsf{prefixes}(p_1), k_2 : \mathsf{prefixes}(p_2))} f_p$$

$$= (\bigsqcap_{p \in \mathsf{prefixes}(p_1)} f_p) \circ (\bigsqcap_{p \in \mathsf{prefixes}(p_2)} f_p) \quad \sqcap \quad (\bigsqcap_{p \in \mathsf{prefixes}(p_1)} f_p) \circ (\bigsqcap_{p \in \mathsf{prefixes}(p_2)} f_p)$$

Mit Satz 4.19

$$= (\mathsf{id} \sqcap \bigsqcap_{\substack{t \in p_1 \\ f_t = \mathsf{const}}} f_t) \circ (\mathsf{id} \sqcap \bigsqcap_{\substack{t \in p_2 \\ f_t = \mathsf{const}}} f_t) \quad \sqcap \quad (\mathsf{id} \sqcap \bigsqcap_{\substack{t \in p_2 \\ f_t = \mathsf{const}}} f_t) \circ (\mathsf{id} \sqcap \bigsqcap_{\substack{t \in p_1 \\ f_t = \mathsf{const}}} f_t)$$

Da $\bigsqcap_{\substack{t \in p_i \\ f_t = \mathsf{const}}} f_t$ eine Konstantenfunktion ist:

$$= \mathsf{id} \quad \sqcap \quad \bigsqcap_{\substack{t \in p_1 \\ f_t = \mathsf{const}}} f_t \quad \sqcap \quad \bigsqcap_{\substack{t \in p_2 \\ f_t = \mathsf{const}}} f_t$$

□

Beweis (Korollar 4.37 auf Seite 55)

Mit Satz 4.35 gilt für den $\mathcal{D}^{\mathcal{C}}$ DFA-Rahmen, und somit auch für $\mathcal{D}^{\mathcal{B}}$:

$$f_{\{p\downarrow s \mid p \in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)\}}(x) = f_{p_i\downarrow s}(x) \sqcap^{\cup/\cap} \bigsqcap^{\cup/\cap}_{\substack{t\in\mathsf{sibl}[s]\\ f_t=\mathsf{const}}} f_t(x)$$

Wir müssen zeigen:

$$f_{\{p\downarrow s \mid p \in \mathsf{TopSorts}(k_1:p_1,\ldots,k_n:p_n)\}}(x) = \begin{cases} \mathsf{const}_\bot & \text{wenn } \exists t \in \mathsf{sibl}[s] : f_t = \mathsf{const}_\bot \\ f_{p_i\downarrow s} & \text{sonst} \end{cases}$$

Da $\top \sqcap \bot = \bot$ folgt aus Satz 4.35 die erste Zeile der Behauptung. Es gebe nun kein $t \in \mathsf{sibl}[s]$ mit $f_t = \mathsf{const}_\bot$. Damit gilt $\forall t \in \mathsf{sibl}[s]$ mit $f_t = \mathsf{const}$, daß $f_t = \mathsf{const}_\top$. Somit folgt aus Satz 4.35 die zweite Zeile der Behauptung.

□

Beweis (Satz 4.39 auf Seite 56)

Wir betrachten nur ein einziges Objekt, seine Objektnummer sei o und sei $\sqcap = \cap$:
Mit Satz 4.35:

$$\begin{aligned}
\mathsf{in}^{\|}_{\mathtt{PAR}}[o] = \mathsf{TRUE} &\iff f^o_{p_i\downarrow s}(x) = \top \text{ und } \bigsqcap_{\substack{t\in\mathsf{sibl}[s]\\ f_t=\mathsf{const}}} f_t(x) = \top \\
&\iff \mathsf{in}^{;}_{\mathtt{PAR}}[o] = \mathsf{TRUE} \text{ und } \forall t \in \mathsf{sibl}[s] : f^o_t(x) \neq \mathsf{const}_\bot \\
&\iff \mathsf{in}^{;}_{\mathtt{PAR}}[o] \wedge \bigwedge_{t\in\mathsf{sibl}[s]} (\mathsf{kill}_t[o] = \mathsf{FALSE}) \\
&\iff \mathsf{in}^{;}_{\mathtt{PAR}}[o] \wedge \overline{\bigvee_{t\in\mathsf{sibl}[s]} \mathsf{kill}_t[o]}
\end{aligned}$$

Mit Korollar 4.27 und Lemma 4.7:

$$\begin{aligned}
\mathsf{gen}_{\mathtt{PAR}}[o] = \mathsf{TRUE} &\iff f^o_{\mathtt{PAR}} = \top \\
&\iff \exists i \in \{1,\ldots,n\} : f_{p_i} \neq \mathsf{id} \text{ und } \nexists i \in \{1,\ldots,n\} : f_{p_i} = \bot \\
&\iff (\bigvee_{i\in\{1,\ldots,n\}} \mathsf{gen}_{p_i} \vee \bigvee_{i\in\{1,\ldots,n\}} \mathsf{kill}_{p_i}) = \mathsf{TRUE} \text{ und} \\
&\qquad (\bigvee_{i\in\{1,\ldots,n\}} \mathsf{kill}_{p_i}) = \mathsf{FALSE} \\
&\iff (\bigvee_{i\in\{1,\ldots,n\}} \mathsf{gen}_{p_i} \wedge \overline{\bigvee_{i\in\{1,\ldots,n\}} \mathsf{kill}_{p_i}}) = \mathsf{TRUE}
\end{aligned}$$

Mit Korollar 4.27 und Lemma 4.7:

$$\begin{aligned}
\mathsf{kill}_{\mathtt{PAR}}[o] = \mathsf{TRUE} &\iff f^o_{\mathtt{PAR}} = \bot \\
&\iff \exists i \in \{1,\ldots,n\} : f_{p_i} \neq \mathsf{id} \text{ und } \exists i \in \{1,\ldots,n\} : f_{p_i} = \bot \\
&\iff (\bigvee_{i\in\{1,\ldots,n\}} f^o_{p_i} = \bot) = \mathsf{TRUE} \\
&\iff (\bigvee_{i\in\{1,\ldots,n\}} \mathsf{kill}_{p_i}) = \mathsf{TRUE}
\end{aligned}$$

Mit Korollar 4.27 und Lemma 4.7:

$$\mathsf{transp}_{\mathtt{PAR}}[o] = \mathsf{TRUE} \iff \forall i \in \{1, \ldots, n\} : f_{p_i} = \mathsf{id}$$
$$\iff (\bigwedge_{i \in \{1,\ldots,n\}} \mathsf{transp}_{p_i}) = \mathsf{TRUE}$$

Die Aussage über $\mathsf{in}^{;}_{p_i}$ ergibt sich aus der Definition von $\mathsf{in}^{;}$. Die Aussagen über $\mathsf{out}_{\mathtt{PAR}}$ entsprechen denen aus Gleichung 3.1 auf Seite 33. Den Beweis für Kann als *Meet*-Operator erhält man durch Dualisierung (s. Satz 3.38 auf Seite 29).

□

Literatur

Afek Y., Brown G., Merritt M. *Lazy Caching. ACM Transactions on Programming Languages and Systems*, 15(1):182–205, Januar 1993.

Aho A.V., Sethi R., Ullman J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

Alpern B., Wegman M.N., Zadeck F.K. *Detecting Equalities of Variables in Programs.* Technischer Bericht CS-87-24, Brown University, Rhode Island, Oktober 1987.

Alpern B., Wegman M.N., Zadeck F.K. *Detecting Equalities of Variables in Programs.* In POPL [1988].

Anderson G. *An Ada Multitasking Solution for the Sieve of Eratosthenes. Ada letters*, 8(5):71ff, September 88.

Andrews G.R., Schneider F.B. *Concepts and Notations for Concurrent Programming.* In Gehani und McGettrick [1988], Kapitel 1, Seiten 3–69.

Assmann U. *Generierung von Programmoptimierungen mit Graphersetzungssystemen.* Doktorarbeit, Universität Karlsruhe, Fakultät für Informatik, Juni 1995a.

Assmann U. *OPTIMIX Language Report for OPTIMIX 7.0.* Interner Bericht 31/95, Universität Karlsruhe, Fakultät für Informatik, Juli 1995b.

Babb R.G., Herausgeber. *Programming Parallel Processors.* Addison-Wesley, 1988.

Babich W.A., Jazayeri M. *The Method of Attributes for Data Flow Analysis. Acta Informatica*, 10:345–272, 1978. Part I Exhaustive Analysis, Part II Demand Analysis.

Bal H.E., Steiner J.G., Tanenbaum A.S. *Programming Languages for Distributed Computing Systems. ACM Computing Surveys*, 21(3):261–322, September 1989.

Balasundaram V., Kennedy K. *Compile-Time Detection of Race Conditions in a Parallel Program.* In *Proceedings of Fourth International Conference on Supercomputing*, Seiten 175–185. 1989.

Best E. *Semantik – Theorie sequentieller und paralleler Programmierung.* Vieweg Verlag, Braunschweig, 1995.

Brandis M.M., Mössenböck H. *Single-Pass Generation of Static Single-Assignment Form for Structured Languages. ACM Transactions on Programming Languages and Systems*, 16(6):1684–1998, November 1994.

Bristow G., Drey C., Edwards B., Riddle W. *Anomaly Detection in Concurrent Programs.* In Gehani und McGettrick [1988], Kapitel 23, Seiten 567–585.

Burke M., Choi J.D. *Precice and Efficient Integration of Interprocedural Alias Information Into Data-Flow Analysis. ACM Letters on Programming Languages and Systems*, 1(1):14–21, 1992.

Callahan D., Subhlok J. *Static Analysis of Low-Level Synchronization.* In *Workshop on Parallel and Distributed Debuggung*, Seiten 100–111. ACM, Mai 1988. Sigplan Notices, Vol. 24, Nr. 1, Jan. 1989.

Chow J.H., Harrison W.L. *Compile-time Analysis of Parallel Programs that Share Memory.* In *19. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Seiten 130–141. ACM, Januar 1992a. Albuquerque, New Mexico.

Chow J.H., Harrison W.L. *A General Framework for Analyzing Shared-Memory Parallel Programs.* In *1992 International Conference on Parallel Processing (ICPP), Conference Proceedings*, Seiten II192–II199. August 1992b.

Chow J.H., Harrison W.L. *State Space Reduction in Abstract Interpretation of Parallel Programs.* In *Proceedings of the 1994 International Conference on Computer Languages ICCL'94, Toulouse, France*, Seiten 277–288. IEEE, IEEE Computer Society Press, Mai 1994.

Cocke J. *Global Common Subexpression Elimination. ACM SIGPLAN Notices*, 5(7):20–24, Juli 1970.

Cocke J., Schwartz J. *Programming Languages and their Compilers.* Technischer Bericht, Courant Institute of Mathematical Sciences, New York University, April 1970.

Cormen T.H., Leisersion C.E., Rivest R.L. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

Cousot P. *Semantic foundations of program analysis.* In Muchnick und Jones [1981], Seiten 303–342.

Cousot P. *Invariance Proof Methods And Analysis Techniques For Parallel Programs.* In Biermann A., Guiho G., Kodratoff Y., Herausgeber, *Automatic Program Construction techniques*, Kapitel 12, Seiten 243–271. MacMillan, Lonon, 1984.

Cousot P., Cousot R. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Constructing or Approximation of Fixpoints.* In *4. ACM Symposium on Principles of Programming Languages*, Seiten 238–252. ACM, Januar 1977. Los Angeles, California.

Cousot P., Cousot R. *Systematic Design of Program Analysis Frameworks.* In POPL [1979], Seiten 269–282. San Antonio, Texas.

Cousot P., Cousot R. *Semantic Analysis of Communicating Sequential Processes.* In Bakker J.d., Leeuwen J.v., Herausgeber, *Automata, Languages and Programming, 7. Colloquium*, Band 85 von *Lecture Notes in Computer Science*, Seiten 119–133. Springer Verlag, Heidelberg, New York, Juli 1980. Noordwijkerhout, NL.

Cytron R., Ferrante J. *Efficiently Computing ϕ Nodes On-The-Fly. ACM Transactions on Programming Languages and Systems*, 17(3):487–506, 1995.

Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K. *An Efficient Method of Computing Static Single Assignment Form.* In *16. ACM Symposium on Principles of Programming Languages*, Seiten 25–35. ACM, Januar 1989. Austin, Texas.

Cytron R., Ferrante J., Rosen B.K., Wegman M.N., Zadeck F.K. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

Dhamdhere D.M. *Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise. ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

Dhamdhere D.M., Harish P. *An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Replacement. ACM Transactions on Programming Languages and Systems*, 15(2):312–336, April 1993.

Dhamdhere D.M., Khedker U.P. *Complexity of Bidirectional Data Flow Analysis.* In POPL [1993], Seiten 397–408. Charleston, South Carolina.

Drechsler K.H., Stadel M. *A Solution to a Problem with Morel and Renvoise's* Global Optimization by Suppression of Partial Redundancies. *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, Oktober 1988.

Emmelmann H. *Codeselektion mit regulär gesteuerter Termersetzung.* Doktorarbeit, Universität Karlsruhe, Fakultät für Informatik, Mai 1994.

Emmelmann H., Schröer F.W., Landwehr R. *BEG – a Generator for Efficient Back Ends. ACM SIGPLAN Notices*, 24(7):227–327, Juli 1989.

Flynn M. *Very High Speed Computing Systems. Proceedings IEEE*, 1966.

Gehani N., McGettrick A.D., Herausgeber. *Concurrent Programming.* Addison-Wesley, 1988.

Gharachorloo K., Lenoski D., Laudon J., Gibbons P., Gupta A., Hennessy J. *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.* In *17. International Symposium on Computer Architecture, Conference Proceedings, ACM SIGARCH Vol 18, Nr. 2*, Seiten 15–26. Juni 1990.

Grunwald D., Srinivasan H. *Data Flow Equations of Explicitly Parallel Programs.* Technischer Bericht CU-CS-605-92, University of Colorado at Boulder, Department of Computer Science, Juli 1992a.

Grunwald D., Srinivasan H. *A Monotone Data Flow System for Analyzing Explicitly Parallel Programs.* Technischer Bericht CU-CS-614-92, University of Colorado at Boulder, Department of Computer Science, Oktober 1992b.

Grunwald D., Srinivasan H. *Data Flow Equations of Explicitly Parallel Programs.* In *PPoPP 93.* ACM SIGPLAN Notices, 1993.

Harel D. *A Linear Time Algorithm for Finding Dminators in Flow Graphs and Related problems.* In *Proceedings of the 17th ACM Symposium on Theory of Computing*, Seiten 185–194. ACM, Mai 1985.

Hecht M.S. *Flow analysis of computer programs.* North Holland, Amsterdam, NL, 1977.

Hecht M.S., Ullman J.D. *Analysis of a Simple Algorithm for Global Data Flow Problems.* In *ACM Symposium on Principles of Programming Languages*, Seiten 207–217. Oktober 1973.

Hecht M.S., Ullman J.D. *A Simple Algorithm for Global Data Flow Analysis Problems. SIAM*, 4(4):519–532, Dezember 1975.

Hoare C. *Communicating Sequential Processes. Communications of the ACM*, 21(8):666–677, August 1978.

Hoare C. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, Inc., 1985.

JáJá J. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

Kam J.B., Ullman J.D. *Global data flow analysis and iterative algorithms. Journal of the ACM*, 23:158–171, 1976.

Kam J.B., Ullman J.D. *Monotone Data Flow Analysis Frameworks. Acta Informatica*, 7:305–317, 1977.

Karp R.M., Ranachandran V. *Parallel Algorithms for Shared-Memory Machines.* In von Leeuwen J., Herausgeber, *Handbook of Theoretical Computer Science*, Kapitel 17, Seiten 869–941. Elsevier Science Publishers, Amsterdam, NL, 1990.

Keller R. *Formal Verification of Parallel Programs. Communications of the ACM*, 19(7):371–384, Juli 1976.

Kildall G.A. *Global Expression Optimization During Compilation.* Doktorarbeit, University of Washington, Seattle, Washington, Juni 1972.

Kildall G.A. *A Unified Approach to Global Program Optimization.* In *ACM Symposium on Principles of Programming Languages*, Seiten 194–206. Oktober 1973.

Knoop J., Rüthing O., Steffen B. *Lazy Code Motion.* In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, Band 27(7) von *ACM SIGPLAN Notices*, Seiten 224–234. Juni 1992.

Knoop J., Rüthing O., Steffen B. *Optimal Code Motion: Theory and Practice. ACM Transactions on Programming Languages and Systems*, 16(4):1177–1155, Juli 1994.

Knoop J., Steffen B. *The Interprocedural Coincidence Theorem.* In Kastens U., Pfahler P., Herausgeber, *Compiler Construction*, Band 641 von *Lecture Notes in Computer Science*, Seiten 125–140. Springer Verlag, Heidelberg, New York, Oktober 1992. slightly updated version.

Knoop J., Steffen B., Vollmer J. *Optimal Code Motion for Parallel Programs.* Technischer Bericht MIP-9511, Universität Passau, Fakultät für Mathematik und Informatik, September 1995.

Knoop J., Steffen B., Vollmer J. *Code motion for parallel programs.* Technischer Bericht LiTH-IDA-R-96-12, Addenda to the Proceedings of the 6th International Symposium on Compiler Construction *(CC'96) (Linköping, Sweden)*, Department of Computer and Information Sciences, Linköping University, 1996a.

KNOOP J., STEFFEN B., VOLLMER J. *Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. ACM Transactions on Programming Languages and Systems*, 18(3):268–299, Mai 1996b.

KNUTH D. *An Empirical Study of FORTRAN Programs. Software–Practice and Experience*, 1(2):105–134, April 1971.

LAMPORT L. *How to make a multiprocessor computer that correctly executes multi process programs. IEEE Transactions on Computers*, c-28(9):690–691, September 1979.

LENGAUER T., TARJAN R.E. *A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems*, 1:121–141, Juli 1979.

LIBOUREL M., VAN SOMEREN H., VOLLMER J. *CCMIR Definition – Specification in SDL, Rationale and Background.* The COMPARE Consortium, Februar 1993. Release: 6.1.

LOEWE W. *Optimierung paralleler Programme.* Doktorarbeit, Universität Karlsruhe, Fakultät für Informatik, Juni 1996. to be published.

MARLOWE T.J., RYDER B.G. *Properties of data flow frameworks – A unified model. Acta Informatica*, 28:121–163, Dezember 1990.

MCDOWELL C.E., HELMBOLD D.P. *Debugging Concurrent Programs. ACM Computing Surveys*, 21(4):593–622, Dezember 1989.

MIDKIFF S., PADUA D.A. *Issues in the Optimization of Parallel Programs.* In *Proceedings of the 1990* International Conference on Parallel Processing, Seiten 105–113, Volume II. August 1990.

MOREL E., RENVOISE C. *Global Optimization by Supression of Partial Redundancies. Communications of the ACM*, 22(2):96–103, Februar 1979.

MUCHNICK S.S., JONES N.D., Herausgeber. *Program Flow Analysis.* Prentice-Hall, Inc., 1981.

MÜLLER T. *Effiziente Verfahren zur Befehlsanordung.* Doktorarbeit, Universität Karlsruhe, Fakultät für Informatik, Juni 1995.

NETZER R.H., MILLER B.P. *What are Race Conditions? Some Issues and Formalizations. ACM Letters on Programming Languages and Systems*, 1(1):74–88, März 1992.

OWICKI S., GRIES D. *An Axiomatic Proof Technique for Parallel Programs I. Acta Informatica*, 6:319–340, 1976.

PARSYTEC. *Parix 1.2 for PowerPC.* Parsytec, Computer GMBH, Aachen, Germany, August 1994.

PCF. *Parallel Computing Forum: PCF Fortran. Fortran Forum*, 10(3), September 1991. special issue.

PHILIPPSEN M. *Optimierungstechniken zur Übersetzung paralleler Programmiersprachen.* Doktorarbeit, Universität Karlsruhe, Fakultät für Informatik, 1993.

PNUELI A. *The Temporal Logic of Programs.* In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science.* 1977.

POPL. *6. ACM Symposium on Principles of Programming Languages.* ACM, Januar 1979. San Antonio, Texas.

POPL. *15. ACM Symposium on Principles of Programming Languages.* ACM, Januar 1988.

POPL. *20. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, Januar 1993. Charleston, South Carolina.

POPL. *22. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, Januar 1995. San Francisco, California.

Reif J.H. *Data Flow Analysis of Distributed Communicating Processes.* In POPL [1979], Seiten 257–268. San Antonio, Texas.

Reif J.H. *Data Flow Analysis of Distributed Communicating Processes.* Technischer Bericht TR-12-83, Harvard University, Center for Research in Computing Technology, September 1984.

Reif J.H., Smolka A.A. *Data Flow Analysis of Distributed Communicating Processes. International Journal of Parallel Programming*, 19(1):1–30, Januar 1990.

Reps T., Sagiv M., Horwitz S. *Precise Interprocedural Dataflow Analysis via Graph Reachability.* In POPL [1995], Seiten 49–61. San Francisco, California.

Rosen B.K., Wegman M.N., Zadeck F.K. *Global Value Numbers and Redundant Computations.* Technischer Bericht CS-87-25, Brown University, Rhode Island, Oktober 1987.

Rosen B.K., Wegman M.N., Zadeck F.K. *Global Value Numbers and Redundant Computations.* In POPL [1988].

Schröer F. *Das GMD Modula-2 Entwicklungssystem. GMD-Spiegel*, 1988a.

Schröer F. *Districts: A Foundation for the Suppression of Partial Redundancies.* GMD – Bericht 304, GMD Forschungsstelle an der Universität Karlsruhe, August 1988b.

Schröer F., Vollmer J. *MOBIL(-P), Intermediate Compiler Languages for (Explicit Parallel) Imperative Languages.* Technischer Bericht, GMD Forschungsstelle an der Universität Karlsruhe, August 1992.

Shapiro E. *The Family of Concurrent Logic Programming Languages. ACM Computing Surveys*, 21(3):412–510, September 1989.

Sharir M., Pnueli A. *Two Approaches to Interprocedural Data Flow Analysis.* In Muchnick und Jones [1981], Seiten 189–234.

SPARC. *The SPARC Architecture Manual, Version 8.* SPARC International Inc, Menlo Park, California, USA, 1992.

Sreedhar V.C., Gao G.R. *A Linear Algorithm for Placing ϕ-Nodes.* In POPL [1995], Seiten 62–73. San Francisco, California.

Srinivasan H., Grunwald D. *An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs.* Technischer Bericht CU-CS-564-91, University of Colorado at Boulder, Department of Computer Science, Dezember 1991.

Srinivasan H., Hook J., Wolfe M. *Static Single Assignment for Explicitly Parallel Programs.* In POPL [1993], Seiten 260–272. Charleston, South Carolina.

SRINIVASAN H., WOLFE M. *Analysing Programs with Explicit Parallelism.* In Banerjee U., Gelernter D., Nicolau A., Padua D., Herausgeber, *Languages and Compilers for Parallel Computing*, Band 589 von *Lecture Notes in Computer Science.* Springer Verlag, Heidelberg, New York, August 1991. 4th International Workshop, Santa Clara, Proceedings.

STOY J.E. *Denotational Sematics: The Scott-Starchey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

TARJAN R.E. *Finding dominators in directed graphs. SIAM*, 3(1):355–365, 1974.

ULLMAN J.D. *Data Flow Analysis.* Technischer Bericht 179, Princeton University, März 1975. Contained in *Proceedings of the International Summer School on Program Analysis and Optimization, Israel Institute of Technology, August 25–29, 1975.*

UNGERER T. *Mikroprozessortechnik, Architektur und Funktionsweise superskalerer Mikroprozessoren.* International Thomson Publishing, Bonn, 1995.

VALMARI A. *Eliminating Redundant Interleavings During Concurrent Program Verification.* In Odijk E., Rem M., Syre J.C., Herausgeber, *PARLE'89, Parallel Architectures and Languages Europe, Volume II*, Band 366 von *Lecture Notes in Computer Science*, Seiten 89–103. Springer Verlag, Heidelberg, New York, 1989.

VALMARI A. *Stubborn Sets for Reduced State Space Generation.* In Rozenberg G., Herausgeber, *Advances in Petri Nets 1990*, Band 483 von *Lecture Notes in Computer Science*, Seiten 491–515. Springer Verlag, Heidelberg, New York, 1990.

VOLLMER J. *Kommunizierende sequentielle Prozesse in Modula-2; Entwurf und Implementierung eines Transputer – Entwicklungssystems.* Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, Mai 1989.

VOLLMER J. *Dataflow Equations for Parallel Programs that Share Memory.* Technischer Bericht GMD-1101-dfepp, Release 1.1, Universität Karlsruhe, Fakultät für Informatik, Januar 1994. Deliverable 2.11.1 of the ESPRIT Project COMPARE # 5933.

VOLLMER J. *Data Flow Analysis of Parallel Programs.* In *Proceedings of the IFIP WG 10.3 Working Conference on* Parallel Architectures and Compilation Techniques, *PACT'95, Limassol, Cyprus*, Seiten 168–177. Juni 1995.

WAITE W.M., GOOS G. *Compiler Construction.* Springer Verlag, Heidelberg, New York, 1984.

WINTER A. *Ein Optimierer für Modula-P – Entwurf und Implementierung.* Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, Oktober 1995.

WOLFE M. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

WOLFE M. *High Performance Compilers for Parallel Computing.* Addison-Wesley, 1996.

WOLFE M., SRINIVASAN H. *Data Structures for Optimizing Programs with Explicit Parallelism.* In Zima H., Herausgeber, *Parallel Computing, 1. Int. ACPC Conference Salzburg, Austria*, Band 591 von *Lecture Notes in Computer Science*, Seiten 139–156. Springer Verlag, Heidelberg, New York, September 1991.

WULF W., JOHNSSON R.K., WEINSTOCK C.B., HOBBS S.O., GESCHCKE C.M. *The Design of an Optimizing Compiler*, Band 2 von *Programming Languages Series.* Elsevier Science Publishers, Amsterdam, NL, 1975.

Zima H., Chapman B. *Super Compilers for Parallel and Vector Computers*. Addison-Wesley, 1990. ACM Press.

Zimmermann W., Löwe W. *An Approach to Machine-Independent Parallel Programming*. In *Parallel Programming, CONPAR 94–VAPP VI, Lecture Notes in Computer Science, 854*, Seiten 277–288. Springer Verlag, Heidelberg, New York, 1994.

Stichwortverzeichnis

Symbole
$\sqcup$, *siehe* Dualitätsprinzip
$\sqcap$, *siehe* Halbverband
$\sqcap^{\cup/\cap}$, *siehe* Halbverband, boolescher
$\sqcap^{\preceq}$, *siehe* Halbverband
$\sqcap_{f\in F} f$, 45
$\boxast_{f\in F} f$, 47
$\sqsubseteq, \sqsubset, \sqsupseteq, \sqsupset, \preceq$, *siehe* Halbordnung
$\preceq$, *siehe* Halbordnung,flache
$\lll$, *siehe* Ereignisfolge
χ, *siehe* Static Single Assignment Form, parallele
χ-Zuweisung, 80
ϕ, *siehe* Static Single Assignment Form
ϕ-Zuweisung, 39, 75, 79
ψ, *siehe* Static Single Assignment Form, parallele
ψ-Zuweisung, 76
$\mathcal{C}$, *siehe* Funktionenraum, konstanter
$\mathsf{const}_\top$,$\mathsf{const}_\bot$, *siehe* Funktionenraum, boolescher
const, *siehe* Funktionenraum, konstanter
d, *siehe* Tiefe eines Kontrollflußgraphen
$\mathcal{D}$, *siehe* DFA-Rahmen
$\mathcal{D}^{\mathcal{B}}$, *siehe* DFA-Rahmen, boolescher
$\mathcal{D}^{\mathcal{C}}$, *siehe* DFA-Rahmen, konstanter
DF, *siehe* Dominanzgrenze
dom, *siehe* Dominator
$\langle e_1; e_2; \ldots; e_n\rangle$, *siehe* Ereignisfolge
$\langle\rangle$, *siehe* Ereignisfolge
$e \in p$, *siehe* Ereignisfolge
f_p, 50
f_s, 50
$f \circ g$, *siehe* Funktionenraum
$\mathcal{F}$, *siehe* Funktionenraum
$\mathcal{F}^{\mathcal{B}}$, *siehe* Funktionenraum, boolescher
$\mathcal{F}^{\mathcal{C}}$, *siehe* Funktionenraum, konstanter
f^i, *siehe* Fixpunkt
head, *siehe* Ereignisfolge
$I(h)$, *siehe* Intervall
$\vec{\imath}$, *siehe* einfache Permutation
id, *siehe* Funktionenraum, boolescher
id, *siehe* Funktionenraum, konstanter
idom, *siehe* Dominator, direkter
$\mathsf{in}^{\|}$, 56
$\mathsf{in}^{;}$, 56
$k_i : p_i$, 20
$\mathcal{L}$, *siehe* Halbverband
$(\mathcal{L}, \sqsubseteq)$, *siehe* Halbordnung
$(\mathcal{L}, \sqsubseteq, \sqcap)$, *siehe* Halbverband
$(\mathcal{L}, \sqsubseteq, \sqcap, \mathcal{F})$, *siehe* DFA-Rahmen
n_0, *siehe* Startknoten
$\mathbb{N}$, 30
$\mathbb{N}_0$, 30
$\langle n_1, \ldots, n_k\rangle$, *siehe* Pfad
N-COMP, 91
$p'; p''$, *siehe* Konkatenation
$p \downarrow e, P \downarrow e$, *siehe* Ereignisfolge
$p \mid_q$, *siehe* Ereignisfolge,eingeschränkte
$p \bowtie (q', q'')$, *siehe* Interleaving
$\mid p \mid$, *siehe* Ereignisfolge
path, *siehe* Kontrollflußgraph
pred, *siehe* Vorgänger
prefixes, *siehe* Präfixe
$s \in S$, 12
sdom, *siehe* Dominator
sibl, 54, 59
s_perm, *siehe* einfache Permutation
ssa_in_nr, 92
ssa_out_nr, 92
succ, *siehe* Nachfolger
$\mathcal{TA}$, *siehe* Teilausdrücke
tail, *siehe* Ereignisfolge
TopSorts, *siehe* Sortierung, topologische
TRANSP, 88
X-COMP, 91

A
abgeleitete Intervallfolge, 34
abgeleiteter Graph, 27
Abschluß, 45, 47
absteigende Kette, 28
Abstrakte Interpretation, 18
AE, *siehe* Available-Expressions
Anweisung
  einfache, 12
  zusammengesetzte, 12
atomare Berechnung, 12
Ausdruck, 13
Ausdrucksauswertung, 13
Ausdrucksklasse, 87
Ausdruckstabellen, 87
Ausführungspfad, 28
Ausgangsberechnung, 88
Ausgangsteil, 88
Available-Expressions, 22

B
Basisblock, 25
BCM, *siehe* Busy Code Motion
BE, *siehe* Busy-Expressions
Befehlsumordner, 7
Berechnung, 87
beschränkt, 29
bidirektionales Problem, 30, 85
Bitvektor, 32
Bitvektor-Darstellung, 31
Bitvektorschritt, 34
Busy Code Motion, 85
Busy-Expressions, 22

C
CC, *siehe* Constant-Computation
CCMIR-P, 63, 78, 82
CFG, *siehe* Kontrollflußgraph, sequentieller, 64
Code Motion, 85
Codeverschiebung, 85
Common Subexpression Elimination, 85
Constant-Computation, 22
Copy In /Copy Out Semantik, 41
CREW, 10, 98
CSE, *siehe* Common Subexpression Elimination

D
Datenflußanalyse, 18, 25, 27
Datenflußanalyserahmen
  boolescher, 32, 43
  distributiv, 30
  konstanter, 31, 43
  monoton, 30
Datenflußinformation, 27
Definition einer Variablen, 23
DFA, *siehe* Datenflußanalyse
DFA_PAR_STMT$^{\mathcal{C}}$, 58
direkt enthalten, 65
Dominanzgrenze, 26
  parallele, 67, 68
Dominanzrelation, 26
  parallele, 67
Dominator, 26
  -baum, 26
  direkter, 26
dominiert, 26
Drei-Adreß-Form, 14
Drei-Adreßanweisung, 14
duale Halbordnung, 29
Dualitätsprinzip, 29

E
Einfügepunkt, 88
einfache Permutation, 48
Eingangsberechnung, 88
Eingangskontrollflußgraph, 64
Eingangsteil, 88
Einselement, 29
Elimination gemeinsamer Teilausdrücke, 85
Elimination partieller Redundanzen, 30, 85
Eliminationsansatz, 34
Ereignis, 19
Ereignisfolge, 19
  Anfang, 19
  eingeschränkte, 19
EREW, 10
erreichbar, 25
erreicht, 31

F
FALSE, 31
Fixpunkt, 30
Fließbandverarbeitung, 6
Funktion
  distributiv, 30
  monoton, 30
Funktionenraum
  boolescher, 32

distributiver, 29
konstanter, 31
monotoner, 29

G
gültig, 31
größter Fixpunkt, 29
Grenzkontrollflußgraph, 27
Grundblock, 25
Grundprädikate, 88

H
halb geordnete Menge, 28
beschränkte, 29
Halbordnung, 28
flache, 28
Halbverband, 29
boolescher, 31
Hilfsvariablen, 14

I
Identifikationsnummer, 14
Identitätsfunktion, 30
Instanz, 30
instruction scheduler, 7
Interleaving, 20, 43, 50, 53, 60
Intervall, 26
-analyse, 26, 28
-folge, 27
-graph, 27
-ordnung, 27, 34
irreduzibel, 27
Iterationsansatz, 34

J
join, 29

K
Kanal, 10
Kann-Problem, 32
Kanten, 25
kleinster Fixpunkt, 30
Knoten, 25
Kommunikation, 11
Kommunikationszeit, 1
Kompositionskette, 44
Konkatenation, 19
konservativ, 16, 60
Kontrollflußanalyse, 25
Kontrollflußgraph, 25
paralleler, 40, 64
single exit, 26
umgekehrter, 26
Konvergenzpunkt, 26
konvergieren, 25
kritische Kante, 91
kritische Region, 14

L
Länge, 19, 25, 28
Lazy Code Motion, 85
LCM, *siehe* Lazy Code Motion
leere Ereignisfolge, 19
leere Pfad, 25
lineare Ordnung, 28
Live-Variables, 22
LOCK, 14
LockedStmts, 14
LV, *siehe* Live-Variables

M
maximaler Fixpunkt, 34
meet, 29
Meet über alle Interleavings, 43
Meet Over all Paths, 31
memory barrier, 7
MFP, *siehe* maximaler Fixpunkt
MIMD, 10, 98
`mirObjectAddr`, 78
mirParallel, 64
Modifikation, 88
Modula-P, 12, 66
MOI, *siehe* Meet über alle Interleavings
MOP, *siehe* Meet Over All Paths
Muß-Problem, 32

N
Nachfolger, 25
Nonstandardsemantik, 18
Nullelement, 29

O
Optimierer, 1
Optimierung, 1

P
P_DF, 68
P_DFA$^{\mathcal{B}}$, 72
P_DFA$^{\mathcal{C}}$, 70
P_DFA$^{\mathcal{B}}$_GENKILL, 72
P_DFA$^{\mathcal{B}}$_GENKILL_CFG, 73
P_DFA$^{\mathcal{B}}$_INOUT, 74
P_DFA$^{\mathcal{B}}$_INOUT_CFG, 74

p_DFA^c_cfg, 71
p_Dom, 67
p_Insert_χ, 80
p_Insert_ϕ, 79
p_Rename_Basic_Block, 80
p_Rename_Expr, 82
p_Rename_Par_Stmt, 82
p_SSA, 78
PAR-Anweisung, 13
Parallel Random Access Machine, 10
parallele Prozesse, 10
parallele Sprache, 11
parallelen Kanten, 65
partielle Ordnung, 28
PCF Fortran, 41
pCFG, *siehe* Kontrollflußgraph, paralleler
pDFA, 43
Pfad, 25
pipelining, 6
pPL, 11
pPL_0, 50
pPL_{lock}, 14, 60
pPL_{sync}, 14, 60
Präfixe, 53
PRAM, *siehe* Parallel Random Access Machine
Problem
  bidirektionale, 30
  kann, 32
  muß, 32
  rückwärts, 30
  unidirektionale, 30
  vorwärts, 30
PROCESS_BODY, 64
Programmpunkt, 19
Programmvariable, 14
Prozeß, 10
  -rumpf, 13
  -rumpfbaum, 65
  replizierter, 13
prozeßlokal, 14
Prozeßrumpfdeskriptor, 64, 65
Prozeßrumpfdeskriptorbaum, 66
pSSA, *siehe* Static Single Assignment Form, parallele

R

Rückwärts-Problem, 30
Rückwärtskante, 26, 73
race condition, 5, 40
RD, *siehe* Reaching-Definitions
Reaching-Definitions, 22, 41, 75, 87, 92
Rechenzeit, 1
reduzibel, 27
RegionVariable, 14
Replikator, 13
  -variable, 13
reverse depth-first order, 35
RISC, 6
rPOSTORDER, 35, 71, 73, 75

S

Sequentielle DFA, 35
sequentielles Programm, 10
SIGNAL, 13
Sortierung
  topologische, 20
Speicher
  gemeinsamer, 10
  lokaler, 10
  verteilter, 10
  virtueller gemeinsamer, 10
Speicherkonfliktauflösungsstrategie, 10
Speicherbänke, 7
Sprungkanten, 64
SSA, *siehe* Static Single Assignment Form
Standardsemantik, 18
Startknoten, 25
Static Single Assignment
  Form, 75
    parallel, 75
Static Single Assignment Form, 25, 26, 37, 39, 40, 87
  parallele, 78
Strukturansatz, 34
Synchronisation, 11, 13
syntaktisch gleich, 87

T

Teilausdrücke, 22
Tiefe eines Kontrollflußgraphen, 27
Transferfunktion, 29, 33, 36
Transformation in SSA-Form, 39
transparent, 88
TRUE, 31

U

unidirektionales Problem, 30

V

Verzahnung, 19

Vorgänger, 25
Vorwärts-Problem, 30
Vorwärtskante, 26, 73

W
WAIT, 13
wertgleich, 87
Wertnumerierung, 37

Z
Zustand, 19
Zuweisung, 13

Tabellarischer Lebenslauf

Zur Person

Name:	Jürgen Vollmer
Geburtsdatum:	27.01.1961
Geburtsort:	Achern, Ortenaukreis

Schulbildung

67/68 – 70/71	Grundschule Achern
71/72 – 79/80	Gymnasium Heimschule Lender, Sasbach / Achern
26.06.80	Abschluß mit dem Abitur (Abschlußnote 1.9)
07/80 – 09/81	Grundwehrdienst

Hochschulstudium

09/81 – 05/89	Studium der Informatik an der Universität Karlsruhe
10.05.89	Studienabschluß als Diplom-Informatiker (Abschlußnote sehr gut)

Nebentätigkeiten während des Studiums

10/83 – 07/85	Informatik I, II Tutor an der Universität Karlsruhe
08/84 – 10/84	Werkstudent bei Siemens, Karlsruhe
02/85 – 04/85	Werkstudent bei Siemens, Karlsruhe
07/85 – 09/85	wissenschaftliche Hilfskraft bei SINTEF Trondheim, Norwegen
08/86 – 10/86	wissenschaftliche Hilfskraft an der Universität Trondheim, Norwegen
03/87 – 05/89	wissenschaftliche Hilfskraft an der GMD Forschungsstelle Karlsruhe

Berufstätigkeiten

09/89 – 12/93	wissenschaftlicher Ansgesteller an der GMD Forschungsstelle Karlsruhe im ESPRIT-Projekt COMPARE
07/92 – 12/93	technische Leitung des ESPRIT Projectes PREPARE innerhalb der GMD
01/94 – 05/96	wissenschaftlicher Ansgesteller an der Universität Karlsruhe, Lehrstuhl Prof. Goos, weiterhin im ESPRIT-Projekt COMPARE
	Während der Zeit bei der GMD und an der Universität: verantwortlich für die Wartung, Weiterentwicklung und den Vertrieb des Modula-2 Compilers *MOCKA*

Eigene Veröffentlichungen

Diplomarbeit

VOLLMER J. *Kommunizierende sequentielle Prozesse in Modula-2; Entwurf und Implementierung eines Transputer – Entwicklungssystems.* Diplomarbeit, Universität Karlsruhe, Fakultät für Informatik, Mai 1989.

Technische Berichte

VOLLMER J. *The Compiler Construction System GENTLE – Manual and Tutorial.* GMD – Bericht 508, GMD Forschungsstelle an der Universität Karlsruhe, Februar 1991. Note: GENTLE was defined by F.W. Schröer in: *Three Compiler Specifications*, GMD – Studien Nr. 166, 1989.

SCHRÖER F., VOLLMER J. *MOBIL(-P), Intermediate Compiler Languages for (Explicit Parallel) Imperative Languages.* Technischer Bericht, GMD Forschungsstelle an der Universität Karlsruhe, August 1992.

LIBOUREL M., VAN SOMEREN H., VOLLMER J. *CCMIR Definition – Specification in SDL, Rationale and Background.* The COMPARE Consortium, Februar 1993. Release: 6.1.

VOLLMER J. *Dataflow Equations for Parallel Programs that Share Memory.* Technischer Bericht GMD-1101-dfepp, Release 1.1, Universität Karlsruhe, Fakultät für Informatik, Januar 1994. Deliverable 2.11.1 of the ESPRIT Project COMPARE # 5933.

KNOOP J., STEFFEN B., VOLLMER J. *Parallelism for Free: Efficient and Optimal Bitvector Analysis for Parallel Programs.* Technischer Bericht MIP-9409, Universität Passau, Fakultät für Mathematik und Informatik, August 1994.

VOLLMER J. *Data Flow Analysis of Parallel Programs.* Interner Bericht 19/95, Universität Karlsruhe, Fakultät für Informatik, März 1995.

KNOOP J., STEFFEN B., VOLLMER J. *Optimal Code Motion for Parallel Programs.* Technischer Bericht MIP-9511, Universität Passau, Fakultät für Mathematik und Informatik, September 1995.

Konferenzbeiträge

VOLLMER J. *Modula-P, A Language for Parallel Programming. Proceedings of the First International Modula-2 Conference October 11-13, 1989, Bled, Yugoslavia*, Seiten 75–79, 1989.

VOLLMER J. *Modula-P, Language Reference Manual.* In Bender K., Herausgeber, *Tagungsband RISC 90*, Seiten 79–87. Forschungszentrum Informatik, Universität Karlsruhe, Dezember 1990.

VOLLMER J. *Experiences with Gentle: Efficient Compiler Construction Based on Logic Programming.* In Maluszynski J., Wirsing M., Herausgeber, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming – PLILP 1991*, Band 528 von *Lecture Notes in Computer Science*, Seiten 425–426. Springer Verlag, Heidelberg, New York, August 1991. Note: GENTLE was defined by F.W. Schröer in: *Three Compiler Specifications*, GMD – Studien Nr. 166, 1989.

VOLLMER J., HOFFART R. *Modula-P, A Language For Parallel Programming: Definition and Implementation on a Transputer Network.* In *Proceedings of the 1992 International Conference on Computer Languages ICCL'92, Oakland, California*, Seiten 54–64. IEEE, IEEE Computer Society Press, April 1992.

VOLLMER J. *Modula-P, A Language For Parallel Programming, and the Implementation of its Channel Communication.* In *PARS - Mitteilungen Nr. 10, GI Workshop Parallelrechner und Parallelsprachen, PARS 92, Schloß Dagstuhl, Germany 26–28. Feb. 1992*, Seiten 59–67. Juli 1992a. ISSN 0177-0454.

VOLLMER J. *Die Programmiersprache Modula-P.* In Grebe R., Baumann M., Herausgeber, *Abstraktband des 4. bundesweiten Transputer-Anwender-Treffens TAT'92, Klinikum der RWTH Aachen*, Seiten 114–115. September 1992b.

VOLLMER J. *Data Flow Analysis of Parallel Programs.* In *Proceedings of the IFIP WG 10.3 Working Conference on* Parallel Architectures and Compilation Techniques*, PACT'95, Limassol, Cyprus*, Seiten 168–177. Juni 1995.

KNOOP J., STEFFEN B., VOLLMER J. *Parallelism for free: Bitvector analyses → No state explosion!* In Tokoro M., Pareschi R., Herausgeber, *Proceedings of the International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'95, Aarhus, Denmark*, Band 1019 von *Lecture Notes in Computer Science*, Seiten 264 – 289. Springer Verlag, Heidelberg, New York, 1995a.

KNOOP J., STEFFEN B., VOLLMER J. *Optimal Code Motion for Parallel Programs – Extended Abstract.* In *Proceedings of the 12th GI Workshop on* Alternative Konzepte für Sprachen und Rechner*, Physikzentrum Bad Honnef, May 2–4 1995.* Technical Report University of Koblenz, Mai 1995b.

KNOOP J., STEFFEN B., VOLLMER J. *Code motion for parallel programs.* Technischer Bericht LiTH-IDA-R-96-12, Addenda to the Proceedings of the 6th International Symposium on Compiler Construction *(CC'96) (Linköping, Sweden)*, Department of Computer and Information Sciences, Linköping University, 1996.

Zeitschriften

KNOOP J., STEFFEN B., VOLLMER J. *Parallelism for Free: Efficient and Optimal Bitvector Analyses for Parallel Programs. ACM Transactions on Programming Languages and Systems*, 18(3):268–299, Mai 1996.

Sonstiges

DIETRICH R., VOLLMER J. *Modula-P: Transputer-Netzwerkprogrammierung im Griff.* In *Jahresbericht der GMD 1991/1992.* 1992.