UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Diplomarbeit

Kontrollfluß basierte Array-Abhängigkeitsanalyse in Schleifenprogrammen

Prüfer:

Prof. Christian Lengauer

Lehrstuhl für Programmierung

Betreuer:

Dr. Martin Griebl

Autorin:

Babette Eckart

19. September 2000

Danksagungen

Mein besonderer Dank gilt Martin Griebl, der mir diese Arbeit ermöglicht hat und durch geduldige Erläuterungen und sehr gute Ratschläge und Denkanstöße einen wesentlichen Teil zum Gelingen beigetragen hat. Herzlich bedanken möchte ich mich bei Herrn Prof. Christian Lengauer, der mich bei der Erstellung des Textes stark unterstützt hat. Auch dem Mitarbeiter des Lehrstuhls, Peter Faber, möchte ich danken, der durch verschiedene Eingabeprogrammbeispiele das globale Verständnis meinerseits erweitert hat. Vielen Dank an Axel Krauth, der ebenfalls an LooPo arbeitet und bei Bitten meinerseits immer prompt reagierte. Allen Beteiligten danke ich für das gute und freundschaftliche Klima. Es war immer ein großes Vergnügen mit ihnen zusammenzuarbeiten.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Bei der Entwicklung von Software ist Effizienz (nach der Korrektheit) eines der Hauptqualitätsmerkmale entstehender Programme. Es gibt verschiedene Möglichkeiten, wenig rechenzeitintensive Programme zu erstellen. Durch den Einsatz von Mehrprozessorrechnern ist es möglich geworden, Programmteile parallel abzuarbeiten. Dieser Parallelisierungsansatz wird auch im Rahmen von LooPo, einem Werkzeug, das am Lehrstuhl für Programmierung an der Universität Passau entwickelt wurde, verfolgt. Da sich das in dieser Arbeit erstellte Analyseprogramm in die Umgebung von LooPo eingliedert, verfolgt es ebenfalls diesen Ansatz.

Die meiste Rechenzeit spart man, indem man solche Programmteile parallel abarbeitet, die am häufigsten durchlaufen werden. Bei der genauen Betrachtung des Quelltextes von Programmen im Vergleich zu ihrem Laufzeitverhalten hat sich gezeigt [1], daß die meisten Programme neunzig Prozent ihrer Rechenzeit in zehn Prozent des Quelltextes, nämlich hauptsächlich in while- oder for-Schleifen verbringen. Deshalb ist es sinnvoll, gerade diesen Schleifen im Hinblick auf die parallele Abarbeitung besondere Aufmerksamkeit zu widmen. Dafür wurde auch LooPo entwickelt.

Parallele Programme können dadurch erstellt werden, daß parallelisierende Übersetzer geschaffen werden. Die Programmalgorithmen werden weiterhin sequentiell entwickelt, der Übersetzer übernimmt die Parallelisierung, indem er den Quelltext des Eingabeprogramms in ein Ausgabeprogramm, das parallele Befehle enthält, umwandelt. Vor der eigentlichen Parallelisierung eines Programms müssen dessen Kontroll- und Datenfluß genau untersucht werden und dabei Datenabhängigkeiten festgestellt werden. Nur voneinander unabhängige Teile können tatsächlich parallel verarbeitet werden [1]. Diese *Datenabhängigkeitsanalyse* ist das Thema dieser Arbeit.

Im Rahmen von LooPo wurden bereits Datenabhängigkeitsanalysen implementiert. Allerdings unterliegen deren bearbeitbare Eingabeprogramme einigen Einschränkungen. So kann zum Beispiel die Datenabhängigkeitsanalyse nach Feautrier [4] nur für Eingabeprogramme ohne bedingte Verzweigungen eingesetzt werden. Ein anderer Algorithmusansatz (ähnlich dem hier beschriebenen *Cffada*) bringt nur für perfekt verschachtelte Eingabeprogramme korrekte Ergebnisse.

Ziel dieser Arbeit ist es, einen Algorithmus zu entwickeln und zu implementieren, der eine optimierte Datenabhängigkeitsanalyse (im LooPo-Umfeld) für Eingabeprogramme mit bedingten Verzweigungen durchführen kann. Darüber hinaus sollen erlaubte Eingabeprogramme nicht notwendigerweise perfekt verschachtelt sein. Des weiteren werden Überlegungen angestellt, wie der Ansatz auf unstrukturierte Eingabeprogramme (siehe Abschnitt 3.3) erweitert werden kann.

Als Hilfswerkzeug für die Berechnungen im Rahmen der Algorithmen wird *Omega* verwendet (siehe Abschnitt 4.2), das auf Relationen operiert. Dieses Werkzeug birgt allerdings einige Probleme, die in dieser Arbeit diskutiert werden.

Zunächst werden im folgenden Kapitel die grundsätzlichen Definitionen und Begriffe der Abhängigkeitsanalyse eingeführt. Anschließend erfolgt in Kapitel 3 die Spezifikation des Eingabeprogramms. Kapitel 4 befaßt sich mit Lösungsansätzen und Entwurfsentscheidungen. Hier werden unter anderem die Probleme durch Omega erläutert. Das prinzipielle Vorgehen der Analyse wird mit den zugehörigen Algorithmen dargelegt, und einige Besonderheiten aufgezeigt. Für die Behandlung unstrukturierter Programme wird ein theoretischer Lösungsansatz vorgestellt.

Kapitel 5 beschäftigt sich mit den zusätzlichen Anforderungen, die durch die Einführung von Prädikaten entstehen. In Kapitel 6 wird die Bedienung des im Rahmen dieser Arbeit erstellten Analyseprogramms erklärt, wobei sämtliche Aufrufparameter mit ihren Auswirkungen kurz beschrieben werden. Die Datenstruktur dieses Analyseprogramms wird dargelegt. Um die Auswirkungen der eingeführten Algorithmen zu demonstrieren, wird eine vollständige Datenabhängigkeitsanalyse mit dem erstellten Analyseprogramm durchgeführt und die Ergebnisse nach den einzelnen Algorithmusschritten erläutert.

Kapitel 2

Allgemeine Definitionen zur Abhängigkeitsanalyse

In diesem Kapitel werden Begriffe und Definitionen, die für das Verständnis der Abhängigkeitsanalyse notwendig sind, vorgestellt und erklärt.

Weiterführende Informationen zu den Definitionen sind aus [6] zu entnehmen.

2.1 Räume

In der Datenabhängigkeitsanalyse, die in den weiteren Kapiteln beschrieben wird, sind Abbildungen von Räumen in Räume der zentrale Aspekt. Deshalb müssen hier einige Definitionen hinsichtlich benötigter Räume eingeführt werden.

2.1.1 Indexraum

In den weiteren Betrachtungen dieser Arbeit werden sich die Abbildungen (=Relationen) auf die *Indexräume* von Zugriffen beziehen. In der nun folgenden Definition wird der Begriff *Statement* zum ersten Mal verwendet, der eine Befehlszeile eines Programms repräsentiert.

Definition 1 (Indexraum) *Sei S ein Statement, welches von n Schleifen mit Indizes i_1,..., i_n umgeben wird. Für jeden einzelnen Index i_j gibt es bis zu $k_{(j)}$ untere und $k^{(j)}$ obere Grenzen (i.Z.: l_j bzw. u_j). Der* Indexraum D_S *des Statements S wird definiert durch:*

$$D_S = \left\{ (i_1, ..., i_n) \;\middle|\; \begin{array}{l} (i_1, ..., i_n) \in \mathbb{Z}^n \wedge \\ (\forall j : 1 \le j \le n : (\forall p : 1 \le p \le k_{(j)} \\ (\forall q : 1 \le q \le k^{(j)} : l_{j_p} \le i_j \le u_{j_q}))) \end{array} \right\}$$

Die *Dimensionalität* des Indexraumes ist die Anzahl n der umgebenden Schleifen. In Programm 2.1 hat also der Indexraum die Dimensionalität 2 und

$$D_S = \{(i,j)|1 \le i \le 5 \wedge 2 \le j \le 10\}.$$

Programm 2.1

```
DO i=1,5
  DO j=2,10
    A[i,j] = 2
  END DO
END DO
```

2.1.2 Datenraum

Der Begriff *Datenraum* bezieht sich auf den Zugriff auf ein bestimmtes Array.

Definition 2 (Datenraum) *Gegeben sei ein Array $A[k_1,...,k_n]$. Über die Grenzen der einzelnen Indexvariablen k_i wird eine Teilmenge eines n-dimensionalen Raumes beschrieben. Dieser Raum heißt* Datenraum *(i.Z. DR).*

Im obigen Beispielprogramm hat der Datenraum ebenfalls Dimensionalität 2, da der Zugriff $A[i,j]$ zwei Indizes enthält.

2.1.3 Operation

Da Zugriffe in *Operationen* stattfinden, muß auch der Begriff der Operation hier definiert werden. Jeder Zugriff hat einen *Indexvektor*, der die Indizes enthält, die durch den Zugriff referenziert werden können. Im obigen Beispielprogramm 2.1 ist der Indexvektor $\vec{i}$ für $A[i,j]$ gleich $\{i,j\}$.

Definition 3 (Operation) *Sei S ein Statement und $\vec{i}$ (der Indexvektor) $\in D_S$.*
Dann wird durch das Paar $< S, \vec{i} >$ eine Operation des Statements S mit Indexvektor $\vec{i}$ bezeichnet.

2.1.4 Indexfunktion

Die Verknüpfung zwischen Indexraum und Datenraum bildet die sogenannte *Indexfunktion*.

Definition 4 (Indexfunktion) *Sei S ein Statement mit Indexraum D_S, das von n Schleifen mit Indizes i_1, ..., i_n umgeben wird, und A ein in S verwendetes Array mit m-dimensionalem Datenraum, n, $m \in \mathbb{N}$, n, $m \geq 1$. Dann existiert für Programme, die der Eingabespezifikation (siehe Kapitel 3) genügen, stets eine affin lineare Funktion $g : D_S \rightarrow DR_S$, so daß gilt*

$$g(i_1, ..., i_n) \text{ stellt den Indexausdruck von } A \text{ dar.}$$

Diese Indexfunktion bildet jeden Punkt aus D_S auf denjenigen Punkt des Datenraumes ab, der während der Operation $< S, \vec{i} >$ durch A referenziert wird. Kommt A in S mehrfach vor, so existiert pro Vorkommen eine Indexfunktion.

Bei der Abhängigkeitsanalyse ist es erforderlich, diese Indexfunktion zu bestimmen. Dieser Vorgang wird als *Mapping* bezeichnet und in Abschnitt 4.1 näher beschrieben.

2.2 Ordnungen

Um Abhängigkeiten zwischen Zugriffen feststellen zu können, ist der Begriff der *Ordnung* essentiell wichtig. In den folgenden Abschnitten werden verschiedene Ordnungen eingeführt. Weitere Definitionen und nähere Erläuterungen dazu befinden sich in [2] und [4].

2.2.1 Lexikografische Ordnung

Die *lexikografische Ordnung* bezieht sich auf ein *später* oder *früher* im Programmablauf.

Definition 5 (Lexikografische Ordnung) *Seien* $\overrightarrow{i} = (i_1, ..., i_n)$ *und* $\overrightarrow{j} = (j_1, ..., j_n)$ *zwei Vektoren und sei* $k \in \mathbb{N}$, $1 \leq k \leq n$.
Der Vektor $\overrightarrow{i}$ *ist bezüglich der Komponente* k lexikografisch kleiner *als der Vektor* $\overrightarrow{j}$ *(auch* $\overrightarrow{i} \ll^k \overrightarrow{j}$*), genau dann, wenn*

$$(\forall m\colon 1 \leq m \leq k,\ m \in \mathbb{N}\colon i_m = j_m) \wedge (i_k < j_k).$$

Die lexikografische Ordnung *(in Zeichen* $\ll$*) ist also eine Disjunktion n linearer Prädikate:*

$$(\overrightarrow{i} \ll \overrightarrow{j}) \Longleftrightarrow (\exists k : 1 \leq k \leq n : \overrightarrow{i} \ll^k \overrightarrow{j})$$

2.2.2 Textuelle Ordnung

Im Unterschied zur lexikografischen Ordnung bezieht sich die *textuelle* Ordnung lediglich auf ein später oder früher im Text. Während bei der lexikografischen Ordnung ein Zugriff im Eingabeprogrammtext nach einem anderen stehen kann und sich trotzdem lexikografisch *vor* diesem befindet, ist das bei der textuellen Ordnung nicht möglich.

Definition 6 (Textuelle Ordnung) *Die* textuelle Ordnung *(auch* $<_{txt}$, $\leq_{txt}$*) wird ausschließlich durch die Reihenfolge in der Notation des Eingabeprogramms bestimmt.*
Seien S und T Statements.
Dann ist die textuelle Ordnung*:*

$$S <_{txt} T \Longleftrightarrow S \text{ steht im Programm vor } T$$

bzw.

$$S \leq_{txt} T \Longleftrightarrow (S \text{ steht im Programm vor } T) \vee (S = T)$$

2.2.3 Maximum einer Ordnung

Die Komposition der lexikografischen und der textuellen Ordnung liefert eine lineare Ordnung auf der Menge aller ausgeführten Operationen. Im folgenden wird für *in einer Ordnung vor* (also lexikografisch *oder* textuell) das Zeichen $\prec$ verwendet.

Abschnitt 4.3.4 wird sich mit dem Maximum dieser Ordnung beschäftigen, um für eine Folge von Operationen die *als letzte ausgeführte* herauszufinden.

2.3 Abhängigkeiten

Bevor auf die Berechnung von Datenabhängigkeiten ausführlich eingegangen wird, muß der Begriff der *Abhängigkeit* erklärt werden. Zudem werden im folgenden Abschnitt einige Begriffe definiert, die ebenfalls dazu dienen, die Abhängigkeitsberechnung zu ermöglichen.

2.3.1 Definition

Von *Abhängigkeiten* zwischen zwei Statements S und T spricht man dann, wenn beide Statements auf dasselbe Array A und in diesem auf dasselbe Element zugreifen.

Formell beschrieben heißt das:

Definition 7 (Abhängigkeit) *Seien S und T zwei nicht notwendigerweise verschiedene Statements, die beide auf das Array A zugreifen. Die zugehörigen Indexfunktionen seien g_S und g_T.*
Das Statement T ist abhängig vom Statement S (andere Schreibweise: $S\ \delta\ T$) genau dann, wenn es Operationen $< S, \overrightarrow{i} >$, $< T, \overrightarrow{j} >$ gibt, für die gilt:

$$\begin{gathered} \overrightarrow{i} \in D_S,\ \overrightarrow{j} \in D_T, \\ g_S(\overrightarrow{i}) = g_t(\overrightarrow{j}), \\ \mathit{und}\ < S,\ \overrightarrow{i} > \prec < T,\ \overrightarrow{j} > \end{gathered}$$

Durch diese Definition wird sichergestellt, daß

- beide Operationen tatsächlich ausgeführt werden
- die Zugriffe auf dasselbe Element erfolgen

- der Zugriff von Statement S vor dem Zugriff von Statement T erfolgt.

Der Hauptalgorithmus, der in dieser Arbeit beschrieben wird, zerlegt Statements in einzelne Operationen, bzw. einzelne Zugriffe, deren Abhängigkeiten zueinander untersucht werden können.

Diese Abhängigkeiten werden geschrieben als: $< S, \overrightarrow{i} > \delta < T, \overrightarrow{j} >$

Es gibt vier verschiedene Abhängigkeitsarten: Die Output-, Anti-, True- und Input-Abhängigkeit. Die Merkmale werden in Tabelle 2.1 aufgelistet.

Zugriff in S	Zugriff in T	Art
schreibend	schreibend	output
schreibend	lesend	true
lesend	schreibend	anti
lesend	lesend	input

Tabelle 2.1: Die verschiedenen Abhängigkeiten

Durch diese Unterscheidung ist es möglich, die genauen Abhängigkeiten zwischen Zugriffen eines Eingabeprogramms beschreiben zu können. Eine Hauptaufgabe erfüllt dabei die True-Abhängigkeit. Für jeden Lesezugriff A auf ein Element x wird hier beschrieben, durch welchen Schreibzugriff B der Wert für das Element x festgelegt wurde. Auch in der parallelen Ausführung des Eingabeprogramms muß dann B *vor* A stattfinden.

Durch die Anti-Abhängigkeit wird für einen schreibenden Zugriff A beschrieben, welche vorhergehenden Lesezugriffe er überschreibt. Für die parallele Ausführung bedeutet das, daß die Lesezugriffe auf jeden Fall sequentiell *vor* dem Schreibzugriff stattfinden müssen. Verschiedene Lesezugriffe auf dasselbe Array-Element dürfen dabei parallel zueinander stattfinden.

Bei der Output-Abhängigkeit gilt, daß Schreibzugriffe auf dasselbe Element nicht parallel ausgeführt werden dürfen. Die Input-Abhängigkeit dagegen hat für die parallele Ausführung von Programmen keine Bedeutung, da, wie gerade beschrieben, Lesezugriffe auf dasselbe Element auch parallel ausgeführt werden können, da sie das Element nicht verändern.

2.3.2 Abhängigkeitsgraph

Durch den *Abhängigkeitsgraph* werden die Abhängigkeiten zwischen verschiedenen Statements dargestellt und damit verdeutlicht.

Definition 8 (Abhängigkeitsgraph) *Ein (Statement-)Abhängigkeitsgraph ist ein Tupel $G = (V, E)$. Es gilt:*

- *V ist die Knotenmenge des Graphen. Die Knoten stellen Zuweisungen des Eingabeprogramms dar.*
- *E ist die Kantenmenge des Graphen. Die Kanten repräsentieren die Abhängigkeiten zwischen den Statements.*

Zu Programm 2.2 sieht der Abhängigkeitsgraph wie in Abbildung 2.1 aus. Die Pfeile stehen für True-Abhängigkeiten.

Programm 2.2

```
A[0] = 10                  {S1}
for (i=1; i<10; i++)
{
  A[i] = A[i-1]            {S2}
}
```

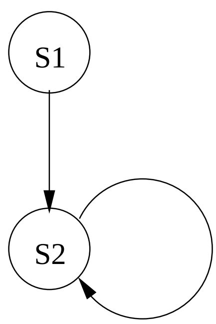

Abbildung 2.1: Beispiel für einen Abhängigkeitsgraphen

2.3.3 Quellraum und Zielraum

Eine Abhängigkeit führt von einem durch sie definierten *Quellraum* in den zugehörigen *Zielraum*.

Definition 9 (Quellraum) *Seien S und T Statements und bezeichne d die Abhängigkeit $S\ \delta\ T$*
Der Quellraum der Abhängigkeit d (auch QR(d)) wird folgendermaßen definiert:

$$QR(d) = \{ \overrightarrow{i} / \overrightarrow{i} \in D_S \wedge (\exists \overrightarrow{j} : \overrightarrow{j} \in D_T :< S, \overrightarrow{i} > \delta_d < T \overrightarrow{j} >)\}$$

Definition 10 (Zielraum) *Der Zielraum von d (auch ZR(d)) wird umgekehrt definiert durch:*

$$ZR(d) = \{ \overrightarrow{j} / \overrightarrow{j} \in D_T \wedge (\exists \overrightarrow{i} : \overrightarrow{i} \in D_S : < S, \overrightarrow{i} > \delta_d < T, \overrightarrow{j} >)$$

2.3.4 Direkte Abhängigkeit

Wie bereits in Abschnitt 2.3.1 eingeführt, werden zum Beispiel durch True-Abhängigkeiten sämtliche Schreibzugriffe zu einem Lesezugriff beschrieben, die auf dasselbe Element zugreifen und vorher stattfanden. Für die parallele Ausführung ist es aber nur interessant, den letzten dieser Schreibzugriffe zu identifizieren, da alle anderen durch diesen bereits wieder überschrieben wurden. Dieser *letzte* Zugriff steht in *direkter Abhängigkeit* zu dem Lesezugriff.

Definition 11 (direkte Abhängigkeit) *Es gelte $S\ \delta\ T$ wie oben eingeführt. Sei U ein beliebiges Statement, in dem A auftritt, g_U die zugehörige Indexfunktion, $\overrightarrow{k} \in D_U$ und $< U, \overrightarrow{k} >$ eine von $< S, \overrightarrow{i} >$ und $< T, \overrightarrow{j} >$ verschiedene Operation.*
Die Abhängigkeit $S\ \delta\ T$ ist genau dann direkt, wenn

$$g_U(\overrightarrow{k}) = g_T(\overrightarrow{j}), < U, \overrightarrow{k} > \prec < T, \overrightarrow{j} > \Rightarrow < U, \overrightarrow{k} > \prec < S, \overrightarrow{i} >.$$

Durch diese formelle Beschreibung wird festgelegt, daß die Speicherzelle $A[g_S(\overrightarrow{i})]$ als letztes durch die Operation $< S, \overrightarrow{i} >$ referenziert wurde, bevor der Zugriff in $< T, \overrightarrow{j} >$ erfolgt.

2.3.5 Kontrollflußgraph

Zur Beschreibung der Algorithmen in dieser Arbeit ist der Kontrollflußgraph unerläßlich, obwohl er nicht in direktem Zusammenhang mit einer theoretischen Abhandlung der Abhängigkeitsanalyse steht.

Ein solcher Graph beschreibt den Ablauf eines Programms. In der vorliegenden Arbeit erfolgt die Darstellung derart, daß die Knoten des Graphen Programmzustände darstellen und die Kanten für Statements stehen. Die Kanten sind gerichtet. Sie führen aus einem ursprünglichen Programmzustandsknoten heraus und münden in den Zustandsknoten, der aus der Ausführung der zugeordneten Operationen resultiert [9]. Die Knoten der Kontrollflußgraphen in dieser Arbeit dürfen maximal zwei Eingangskanten haben.

Für das Programm 2.3, das im folgenden als ein Leitbeispiel dienen wird, ist der Kontrollflußgraph in Abbildung 2.2 abgebildet. *ML* steht hier für einen *Mergeloop*-Knoten, das heißt den Eingangsknoten einer Schleife. Die Knotenbezeichnung *U* beschreibt einen nicht näher spezifizierten Knoten.

Programm 2.3

```
DO i=0,20
  DO j=0,20
    A[i+j+1] = 1
    if (P)
    {
      A[i+j] = 2
    }
    y = A[i+j]
  END DO
END DO
```

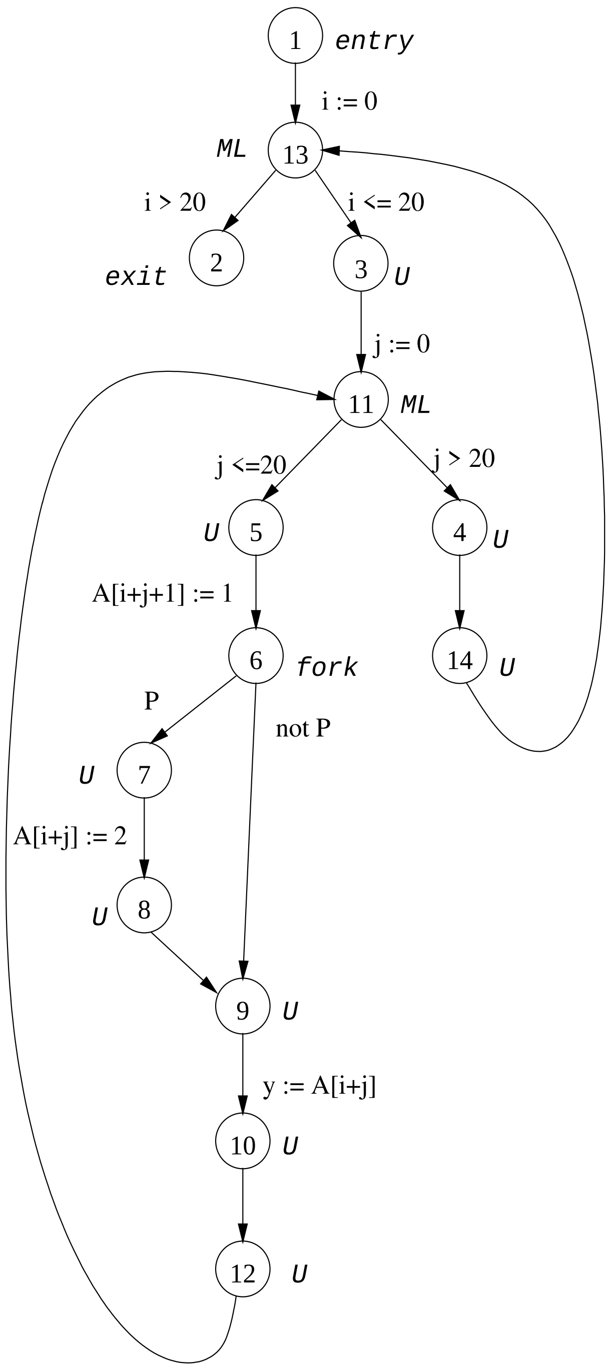

Abbildung 2.2: Kontrollflußgraph zu Beispielprogramm 2.3

Kapitel 3

Spezifikation des Eingabeprogramms

Die im Rahmen von LooPo entwickelten Datenabhängigkeitsanalysen sind nicht mächtig genug, sämtliche Eingabeprogramme korrekt zu analysieren. Deshalb müssen einige Einschränkungen für die möglichen Eingabeprogramme spezifiziert werden. Dieses Kapitel beschreibt die Bedingungen, denen ein Eingabeprogramm genügen muß, um von den später in dieser Arbeit vorgestellten Algorithmen behandelt werden zu können.

Zunächst soll im folgenden Abschnitt erläutert werden, welche Anforderungen ein Eingabeprogramm für die bisherigen Abhängigkeitsanalyse-Ansätze von LooPo erfüllen mußte. In den anschließenden Abschnitten werden dann einige der Restriktionen wieder aufgehoben, da es das Hauptziel dieser Arbeit ist, eine möglichst mächtige Analyse zu entwickeln.

3.1 Ursprüngliche Spezifikationen

In den bisherigen Abhängigkeits-Analyseansätzen mußten die Eingabeprogramme folgenden Bedingungen genügen:

- Sie bestehen ausschließlich aus for-Schleifen, Zuweisungen und skip-Statements (= leere Statements).
- Sie unterliegen einem statischen Kontrollfluß. Schleifengrenzen dürfen nur aus affin linearen Funktionen von umgebenden Schleifenindizes und Strukturparametern (symbolischen Konstanten) bestehen.

- Ihre Array-Indizes bestehen wie die Schleifengrenzen aus affin linearen Funktionen.
- Die Schrittweite der Schleifen beträgt $+1$.

3.2 Erweiterung bisheriger Ansätze

Da es für die oben genannten Bedingungen bereits implementierte Abhängigkeits-Analysen gibt, sollen für die in dieser Arbeit entwickelte Analyse einige der Einschränkungen nach Möglichkeit aufgehoben werden.

3.2.1 Array-Elemente als Schleifengrenze zulässig

Die erste Erweiterung, die hier vorgestellt wird, ist die Ausdehnung der erlaubten Schleifengrenzen auf Array-Elemente.

Eine solche Grenze wird durch einen Pseudoparameter ersetzt, mit dem dann die Berechnung normal durchführbar ist.

Für eine Schleife, in der die Obergrenze für einen Schleifenindex i $A[i]$ ist, würde $A[i]$ zum Beispiel durch

$$\text{param1} := \text{A[i]}$$

ersetzt werden. Beim Eintritt in die Schleife muß dann gelten:

$$\text{i} \leq \text{param1}$$

Damit geht die Abhängigkeit der oberen Grenze von i zwar für die Berechnung verloren, sie kann aber bei der Auswertung des Endergebnisses wieder einfließen, indem der Pseudoparameter wieder durch den ursprünglichen Wert ersetzt wird.

3.2.2 Bedingte Verzweigung

Schon in dem eingeführten Beispielprogramm 2.3 ist eine bedingte Verzweigung enthalten. Solche Verzweigungen sind für den im Rahmen dieser Arbeit implementierten Ansatz zugelassen. Es dürfen auch if-else-Konstrukte verwendet werden.

So ist zum Beispiel das Programm 3.1 zulässig.

Programm 3.1

```
DO i=1,10
  DO j=1,5
    if (i<j)
      A[i+1] = 2
    else
      A[i+j] = A[j-1]
  END DO
END DO
```

3.2.3 0-dimensionale Zugriffe

Es sind Zugriffe erlaubt und bearbeitbar, die von keiner Schleife umgeben werden.

Ein Beispiel wäre Programm 3.2.

Programm 3.2

```
DO i=1,10
  A[i] = i
END DO
B = A[2]
```

3.2.4 Nicht-perfekte Verschachtelung

Perfekt verschachtelte Programme sind solche, deren innere Schleifen von allen äußeren umgeben sind. Das bedeutet auch, daß es nur *eine* äußere Hauptschleife gibt, die sämtliche inneren enthält.

Das Programm 3.3 gehört zu den perfekt verschachtelten Programmen.

Programm 3.3

```
DO i=1,10
  DO j = 1, 5
    DO k = 0,3
      A[k] = A[k-1]
    END DO
  END DO
END DO
```

Die Aufhebung dieser Bedingung bedeutet, daß Schleifen beliebig in andere Schleifen geschachtelt sein können. Es darf in solchen Programmen auch mehrere äußere Schleifen geben. Man beachte das Beispielprogramm 3.4.

Programm 3.4

```
DO i=1,10
  DO j=1,5
    A[j] = 4
  END DO
  DO k=0,2
    A[k] = A[k-1]
  END DO
END DO
DO l=0,5
  A[l] = 0
END DO
```

3.3 Unzulässige Eingabeprogramme

Für die im Rahmen dieser Arbeit implementierte Abhängigkeitsanalyse sind unstrukturierte Eingabeprogramme und Eingabeprogramme, die *Gotos* enthalten, nicht zulässig.

Durch Verwendung von Goto in Programmen können Sprünge erzeugt werden, die eine ungewöhnliche Form des Kontrollflußgraphen bewirken und die Analyse deutlich erschweren. Wenn Sprünge zwischen Schleifen erlaubt sind, wird das Programm *unstrukturiert*. Das Programm 3.5 zeigt ein Goto-Programm und Abbildung 3.1 zeigt den zugehörigen Kontrollflußgraphen.

Um den Unterschied zu dem Kontrollflußgraphen aus Abbildung 2.2 deutlicher zu erkennen, ist eine allgemeinere Darstellung der Programmabläufe wie in den Abbildungen 3.2 und 3.3 hilfreich. Es ist leicht zu erkennen, daß der Graph für ein Programm *ohne* Gotos eine strukturierte Form hat, da Schleifen entweder komplett in anderen Schleifen enthalten sind, oder aber eine *äußere* Schleife bilden. Gotos dagegen erlauben den in Abbildung 3.2 dargestellten Ablauf, in dem *Schleifen*, also hier solche Programmteile, die durch eine Rückwärtskante wiederholt durchlaufen werden, beliebig verschachtelt sind.

Abbildung 3.4 zeigt ein Beispiel, in dem undeterministisch aus einer Schleife in die andere gesprungen werden kann, so daß auch dieses Beispiel zu den unstrukturierten Programmabläufen zählt. Diese Unstrukturiertheit von Programmen kann nicht nur durch Gotos verursacht werden, sondern auch durch andere Anweisungen, wie zum Beispiel *break*.

Die Bearbeitung solcher Eingabeprogramme ist zum momentanen Zeitpunkt durch die implementierten Algorithmen nicht möglich, da eine genauere Bestimmung und Bearbeitbarkeit von Indexvariablen und anderen Variablen notwendig wäre, als das im Moment von LooPo angeboten wird.

Bei der Weiterentwicklung der Datenabhängigkeitsanalyse sollten diese Eingabeprogramme weiterhin berücksichtigt werden. Deshalb folgt in Abschnitt 4.4 ein theoretischer Lösungsansatz, wie für sie eine Abhängigkeitsanalyse durchgeführt werden kann.

Programm 3.5

```
CONSTANT n;

i=0
j=1

lab1:
i= i+2              S1
A[i] = A[i-3]       S2
lab2:
y = A[2*i]          S3
j = j+1             S4
if (i<n)
{
  A[i+3] = 5        S5
  goto lab1
}
B[i] = A[i]         S6
if (j<=i)
{
  A[2*j] = A[j+2]   S7
  goto lab2
}
```

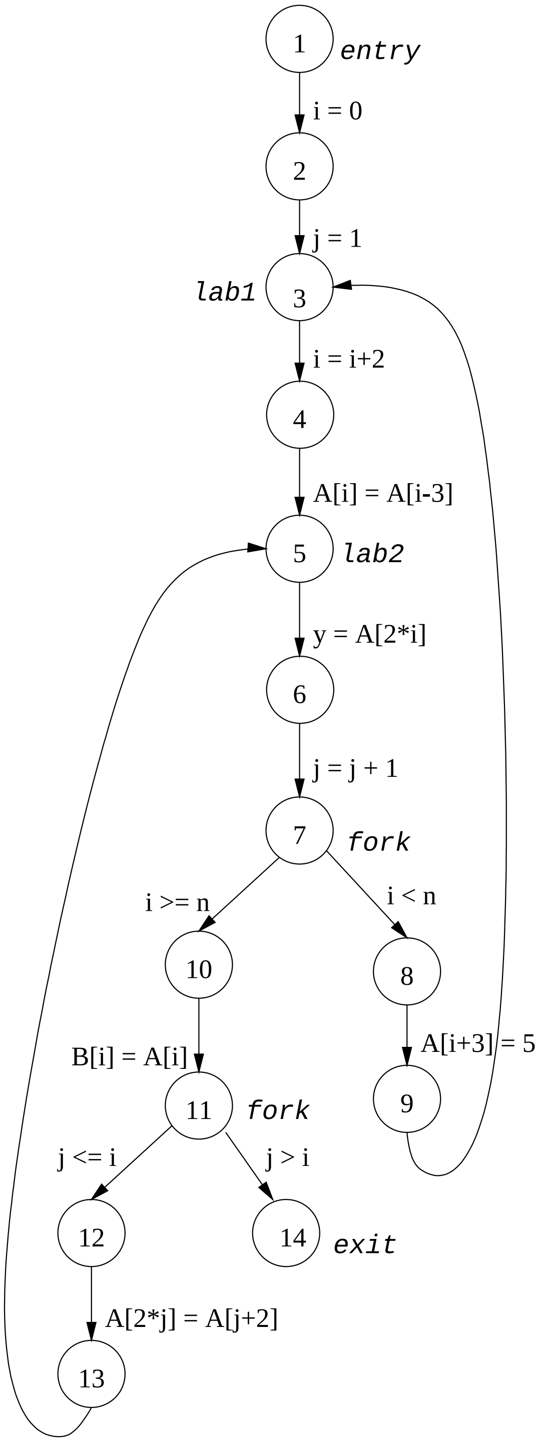

Abbildung 3.1: Kontrollflußgraph zu Beispielprogramm 3.5

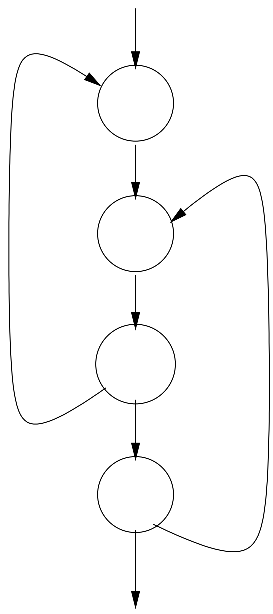

Abbildung 3.2: Kontrollflußgraph zu allgemeinen Goto-Programmen

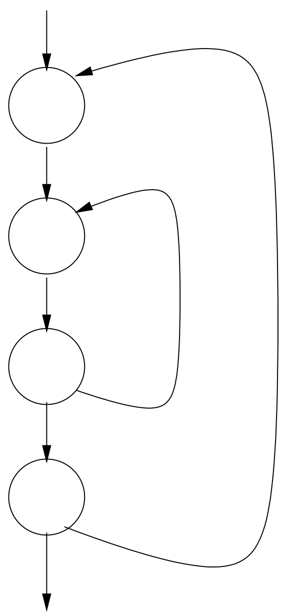

Abbildung 3.3: Kontrollflußgraph zu allgemeinen Schleifenprogrammen ohne Goto

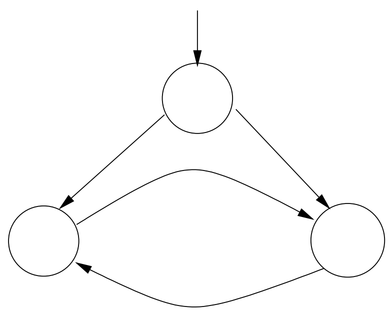

Abbildung 3.4: Kontrollflußgraph zu einem unstrukturierten Programm

Kapitel 4

Lösungsansätze und Entwurfsentscheidungen

Dieses Kapitel beschreibt die Lösungsansätze zur Datenabhängigkeitsanalyse der in Kapitel 3 spezifizierten Eingabeprogramme.

Bei der Entwicklung des in dieser Arbeit erstellten Analyseprogramms wurden zum Teil Vorgehensweisen untersucht, die aus verschiedenen Gründen nicht in die endgültige Implementierung übernommen wurden. Der Vollständigkeit halber werden auch diese hier beschrieben. Außerdem werden weitere Faktoren diskutiert, die den Entwurf des Analyseprogramms beeinflussen.

4.1 Relationen in der Abhängigkeitsanalyse

Für die Spezifikation von Abhängigkeiten sind Relationen gut geeignet, da sich durch sie Abbildungen zwischen Räumen gut darstellen lassen.

Die hier verwendeten Relationen arbeiten mit Tupeln. Es gibt ein Eingangs- und ein Ausgangs-Tupel und zugehörige Übergangsbedingungen für jede Relation. Dadurch ist es möglich, zum Beispiel den Zusammenhang zwischen einem Zugriff in einem Statement und dem referenzierten Array herzustellen (=*Mapping*). Dabei werden alle in dem Zugriff referenzierten Indizes in das Eingangs-Tupel aufgenommen, sowie die Nummer des Statements im Eingabeprogramm. Unter der *Nummer* eines Statements oder Arrays versteht man hier eine Identifikationsnummer, die während der Bearbeitung vergeben wird und für Felder und Statements eindeutig ist. Das Ausgangs-Tupel enthält die tatsächlich von dem Zugriff betroffenen Werte der Indizes sowie die

Nummer des betroffenen Arrays.

Viel wichtiger für die in dieser Arbeit verwendeten Algorithmen sind die Abhängigkeiten zwischen verschiedenen Arrayzugriffen, da diese die Datenabhängigkeiten repräsentieren, die für die Parallelisierung benötigt werden. Für diese Datenabhängigkeit wird eine Relation erstellt, die Zugriffe miteinander in Verbindung setzt. Diese Relation repräsentiert die Gleichsetzung der beiden Indexfunktionen, die in Abschnitt 2.1.4 eingeführt wurde. Das Eingangs-Tupel enthält dann alle Indizes, die in dem Zugriff referenziert werden, für den gerade die Abhängigkeitsanalyse durchgeführt wird (im Folgenden wird dieser Zugriff als *Zielzugriff* bezeichnet). Die Nummer des Statements, in dem der Zielzugriff stattfindet, ist ebenfalls im Eingangs-Tupel enthalten. Im Ausgangs-Tupel sind die Indizes des Zugriffs enthalten, von dem der Zielzugriff abhängig ist (im Folgenden erhält dieser Zugriff den Namen *Quellzugriff*), und wieder die Nummer des übergeordneten Statements.

4.2 Verwendung von Omega

Da in der Analyse mit Relationen gearbeitet werden soll, ist der Einsatz eines Hilfswerkzeuges, das Relationen-Arithmetik anbietet, erstrebenswert. Ein solches Werkzeug ist *Omega*, dessen Eigenschaften und Anwendung in den folgenden Abschnitten diskutiert werden. Nähere Informationen dazu finden sich in [8] und [7].

4.2.1 Was ist Omega?

Omega ist ein Hilfswerkzeug, das Operationen auf Relationen anbietet. Neben einfachen Methoden zur *Quellraumberechnung* und *Zielraumberechnung* (Definition siehe Abschnitt 2.3.3) sind auch Operationen zur Verknüpfung von Relationen möglich. So ist z.B. die Bildung einer Schnittmenge oder Vereinigung und die Komposition von Relationen durchführbar.

Omega kann für Berechnungen auf Relationen als eigenständiges Programm aufgerufen oder als Bibliothek in C-Programme eingebunden werden.

4.2.2 Vorteile

Die Vorteile der Verwendung von Omega sind:

- Quell- und Zielraumberechnung
  Omega berechnet die Quell- und Zielräume zu einer gegebenen Relation automatisch. Eine eigene Implementierung der zugehörigen Algorithmen ist also nicht notwendig.

- Prädikate
  Ausdrücke, die bei if-Abfragen ausgewertet werden, werden durch *Prädikate* abgebildet. Je nach Belegung eines solchen Prädikats wird ein bestimmter Zweig des Kontrollflußgraphen beschritten. Daß Omega die Verwendung von Prädikaten unterstützt, ist wichtig, da die Analyse für Eingabeprogramme durchgeführt werden soll, die solche bedingten Verzweigungen enthalten. Es ist in Omega auch möglich, Prädikaten Parameter mitzugeben, so daß die Abhängigkeit einer Bedingung, zum Beispiel von umgebenden Schleifenindizes, dargestellt werden kann.
  Omega repräsentiert ein Prädikat, das zum Beispiel zwei Parameter (i, j) enthält und zum booleschen Wert true evaluiert, durch:

  $$\text{pred(i, j)} \geq 0.$$

- Verknüpfungen von Relationen
  Die Abhängigkeitsanalyse arbeitet mit der Verknüpfung von Relationen (z.B. Vereinigung, Schnittmenge, Komposition). Diese Operationen müßten ohne dieses Werkzeug mit hohem Aufwand implementiert werden, da sie sehr komplex werden können. Omega bietet diese Funktionalitäten durch Methodenaufrufe an und erspart somit viel Entwicklungszeit beim Erstellen des Analyseprogramms.

4.2.3 Grenzen

- Komposition unterschiedlich großer Relations-Tupel
  Der größte Nachteil von Omega ist, daß es keine Möglichkeit gibt, Relationen mit unterschiedlicher Eingangs- und Ausgangs-Tupel-Größe zu verknüpfen, außer bei der Komposition von Relationen.

  Für das Beispielprogramm 4.1 ist die Quelle der Abhängigkeit für den Lesezugriff auf das Array A in Statement $S3$ entweder das eindimensionale Statement $S1$ oder das zwei-dimensionale Statement $S2$.

Programm 4.1

```
DO m = 1,15
  A[1,m] = y          {S1}
END DO
DO i=1,10
  DO j=1,20
    A[i+1, j+2] = 0   {S2}
    X[i,j] = A[i,j]   {S3}
  END DO
END DO
```

Für die Darstellung der Abhängigkeit des Zielzugriffs von den Quellzugriffen müßten also ein- und zwei-dimensionale Tupel vereinigt werden.

Es wäre wünschenswert, daß folgendes Ergebnis für die True-Abhängigkeiten herauskäme:

$$\{[\ 1,\ j,\ S3] \rightarrow [\ j(=m),\ S1]:\ 1 \leq j \leq 15\} \cup$$
$$\{[\ i,\ j,\ S3] \rightarrow [\ i\text{-}1,\ j\text{-}2,\ S2]:\ 2 \leq i \leq 10 \wedge 3 \leq j \leq 20\}$$

Genau dieses Ergebnis läßt sich mit Omega nicht erzeugen.

Wie man dieses Problem im implementierten Analyseprogramm lösen, bzw. umgehen kann, wird in Abschnitt 5.2 im Rahmen der Prädikatsproblematik vorgestellt.

- Parameter in Prädikaten
  Als ein weiterer Nachteil ist zu sehen, daß Prädikate in Omega lediglich mit einer Anzahl von Parametern näher spezifiziert werden können. Es ist nicht möglich, genau anzugeben, von welchen Parametern das jeweilige Prädikat abhängt. Zu welchen Problemen diese Einschränkung führt, wird in Kapitel 5 verdeutlicht.

- Berechnung der Vereinigung
  Während der Berechnung von Abhängigkeiten werden häufig Vereinigungen von Relationen vorgenommen. Dabei tritt das Problem auf, daß bei der Vereinigung komplizierterer Relationen (z.B. solche mit Prädikaten) die Schnittmenge zweier Relationen doppelt auftaucht. Das ist zwar theoretisch korrekt aber praktisch ineffizient und kann wegen der teilweise sehr großen Relationen sogar zu Speicherüberläufen führen.

4.3 Prinzipielles Vorgehen

In Kapitel 2 wurde auf die Grundlagen der Datenabhängigkeitsanalyse eingegangen. Wie diese in Algorithmen umgesetzt werden können und was das Umfeld des entwickelten Analyseprogramms ist, wird in diesem Abschnitt dargelegt.

4.3.1 Umgebung des Analyseprogramms

Das im Rahmen dieser Arbeit entstandene Analyseprogramm wird im folgenden *Cffada* genannt("**C**ontrol **f**low based **f**uzzy **a**rray **d**ata flow **a**nalysis").

Es wurde im Rahmen des LooPo-Projekts am Lehrstuhl für Programmierung an der Universität Passau entwickelt. Das Werkzeug LooPo enthält mehrere Teilprogramme, die durch Hintereinanderausführung eine Umwandlung eines sequentiellen Eingabeprogramms in ein paralleles Ausgabeprogramm durchführen, soweit Datenabhängigkeiten dies zulassen. Hauptaufgabe von LooPo ist es somit, Abhängigkeiten zwischen Daten zu lokalisieren und anschließend Operationen zu finden, die parallel ausführbar sind, ohne Speicherkonflikte zu erzeugen. LooPo besitzt eine grafische Benutzeroberfläche zur Steuerung des Programmablaufs. Im folgenden werden die Schritte jeweils mit einer kurzen Beschreibung aufgelistet. Genauere Informationen zu LooPo finden sich in [10] und [5].

- Scanner und Parser
  Die Aufgabe dieser beiden Komponenten ist es, das Eingabeprogramm einzulesen und einen Parse-Baum sowie die zugehörige Symboltabelle zu erstellen. Zusätzlich werden Abhängigkeiten zwischen Zugriffen allgemein in einer Datei abgelegt, falls dies vom Benutzer gewünscht wird. Durch einen Schalter in der Oberfläche des Werkzeugs kann die Normalisierung des Eingabeprogramms erzwungen werden. Scanner und Parser werden automatisch aufgerufen, wenn in LooPo das Laden eines Eingabeprogramms angestoßen wird.

- Abhängigkeitsanalyse
  Hier kann zwischen verschiedenen Algorithmen gewählt werden. Das im Rahmen dieser Arbeit erstellte Analyseprogramm Cffada ist ebenfalls in dieser Auswahl enthalten. Sämtliche zur Datenabhängigkeitsanalyse notwendigen Algorithmen werden ausgeführt und die ebenfalls durch Auswahl festgelegten Abhängigkeiten werden in einer Datei abgelegt.

Außer Cffada gibt es noch die Analyse-Durchführung nach Feautrier [4] oder nach Banerjee [2].

- Scheduler und Allocator
  Durch diese Komponenten werden Operationen und Daten auf einem Netz von Prozessoren plaziert. Dabei wird versucht, eine möglichst geringe Distanz zwischen der Operation und den zugehörigen Daten zu erhalten.

- Target Code Generator
  Hier wird der Zielcode generiert, der in einer parallelen Programmiersprache dargestellt wird.

4.3.2 Eingabedaten für die Analyse

Im folgenden werden die von LooPo generierten Eingabedaten für die Datenabhängigkeitsanalyse beschrieben.

4.3.2.1 Kontrollflußgraph

Cffada kann bereits auf einem fertigen Kontrollflußgraphen des Eingabeprogramms aufsetzen. Der Kontrollflußgraph zu Beispielprogramm 4.2 ist in Abbildung 4.1 zu sehen.

Programm 4.2

```
DO i=0,20
  DO j=0,30
    DO k=0,25
      A[(3*i)+j]=5
    END DO
  END DO
  DO l=0,22
    y = A[i-l]
  END DO
END DO
```

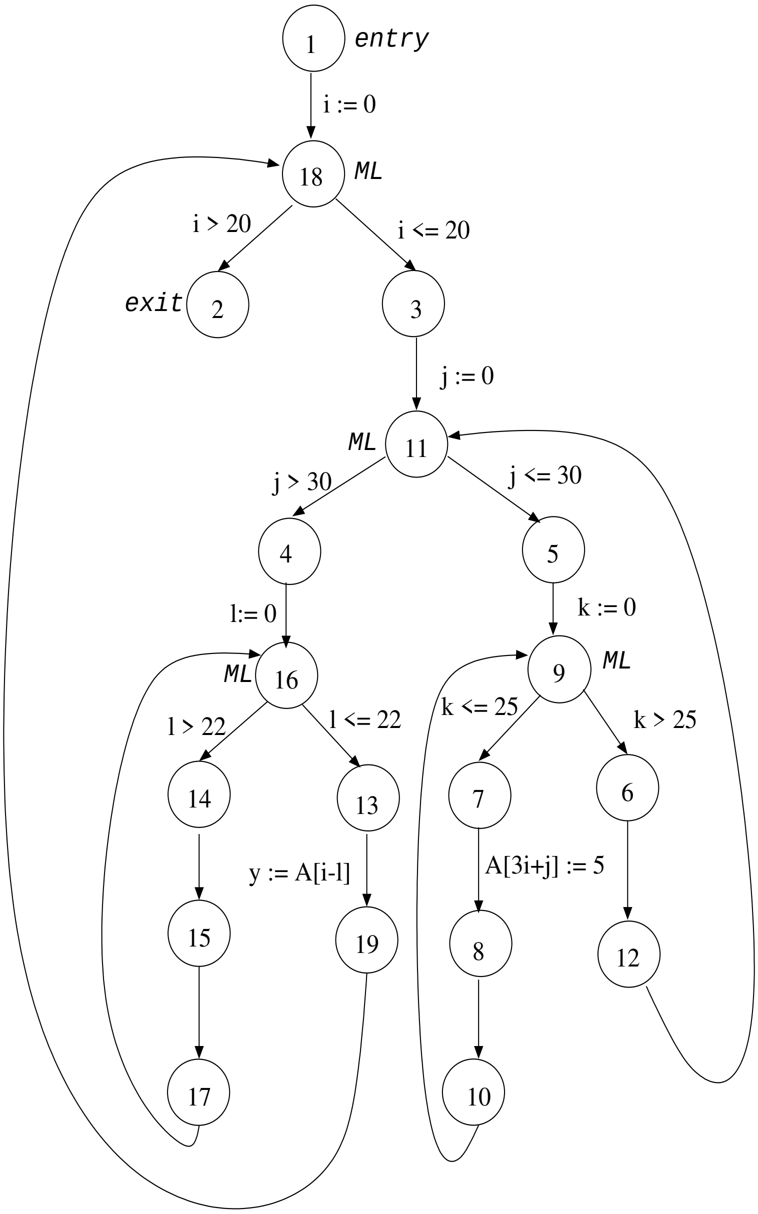

Abbildung 4.1: Kontrollflußgraph zu Beispielprogramm 4.2

4.3.2.2 Zugriffe

Cffada kann auf eine Eingabedatei zugreifen, die das Eingabeprogramm nicht nur in einzelne Statements unterteilt, sondern jeden Zugriff auf bestimmte Feldelemente dokumentiert.

Damit wäre zum Beispiel das Statement

A[i+1] = A[(5*i-2)] + B[j]

unterteilt in

- einen Schreibzugriff auf Array A und zwar auf das Element mit dem Index $i + 1$,
- einen Lesezugriff auf Array A, Element $5 * i - 2$ und
- einen Lesezugriff auf Array B, Element j.

Jeder Kante des Kontrollflußgraphen in Abbildung 4.1 kann eine Operation zugeordnet sein. Manche Kanten enthalten keine Operationen, sie dienen lediglich einer übersichtlicheren Ablaufverfolgung (z.B. die Kanten zwischen den Knoten 6 und 12 in Abbildung 4.1). Zusätzlich können die Kanten auch Verweise auf die beschriebenen einzelnen Zugriffe enthalten.

Je nach Art der Abhängigkeit (True-, Output- oder Anti-Abhängigkeit), für die die Analyse durchgeführt wird, werden die korrespondierenden Array-Element-Zugriffe auf bestehende Konflikte untersucht. So wird zum Beispiel bei der Analyse von True-Abhängigkeiten, also Lese-Schreib-Abhängigkeiten, untersucht, ob für Lesezugriffe Schreibzugriffe existieren, die *vor* dem Lesezugriff stattfinden und dasselbe Array-Element betreffen.

4.3.2.3 Prädikate

Ausdrücke, die einer bedingten Verzweigung in einem Eingabeprogramm als Bedingung zugeordnet sind und nicht zur Übersetzungszeit ausgewertet werden können, weil sie zum Beispiel dynamische Bedingungen enthalten, werden durch sogenannte Pseudo-Prädikate ersetzt. Solch ein Ausdruck wird bei der Analyse durch einen Prädikatsnamen (zum Beispiel *pred1*) ersetzt und an den Verzweigungskanten mit der jeweils verlangten Belegung abgespeichert. Omega stellt diese Belegung dar, wie in Abschnitt 4.2.2 beschrieben. Auch diese Information erhält Cffada über den Kontrollflußgraphen.

4.3.3 Algorithmen der Datenabhängigkeitsanalyse

In diesem Kapitel werden die zur betrachteten Datenabhängigkeitsanalyse benötigten Algorithmen erklärt. Zunächst wird der allgemeine Ablauf kurz skizziert.

Die Aufgabe der Analyse besteht darin, Quellzugriffe zu einem Zielzugriff zu finden, die *vor* dem Zielzugriff stattfinden. Daher ist es sinnvoll, ausgehend von dem Zielzugriff das betrachtete Eingabeprogramm bzw. dessen Kontrollflußgraphen *rückwärts* zu durchlaufen. Potentielle Quellzugriffe müssen bei diesem Durchlauf identifiziert und Konflikte bestimmt werden. Tatsächlich existierende Konflikte werden jeweils dem Ergebnis der Analyse hinzugefügt.

Dieses Vorgehen reicht nicht aus. Da auch Rückwärtskanten, also solche, die von einer Iteration einer Schleife in die vorhergehende führen, durchlaufen werden müssen, muß dort eine *Ordnungsbestimmung* (lexikografisch) hinsichtlich der betroffenen Indexvariablen durchgeführt werden können. Damit können auch hier *vorher* erfolgte Konfliktzugriffe bestimmt werden. Die Bestimmung einer solchen Ordnung wird in diesem Kapitel noch näher erläutert werden. Die sich ergebende Ordnung an einer konkreten Stelle des Graphen wird dann *aktuelle Ordnung* genannt. Die aufgelisteten allgemeinen Schritte werden in den nun folgenden Abschnitten näher erläutert.

4.3.3.1 Bestimmung der Konfliktrelationen

Bei der Graphabarbeitung werden potentielle Quellzugriffe dadurch charakterisiert, daß sie eine zugehörige Konfliktrelation haben, die erfüllbar ist. Das heißt, die korrespondierenden Zugriffe stehen tatsächlich in Konflikt.

Eine Konfliktsituation, also eine eventuelle Abhängigkeit, besteht dann, wenn das Array-Element der beiden Zugriffe gleich ist. Eine Grundvoraussetzung dafür ist, daß die Zugriffe auf Array-Elemente derselben Dimensionalität erfolgen. Zudem muß es mindestens eine Belegung des Quellindexvektors (der Vektor enthält alle Indexvariablen) und des Zielindexvektors geben, bei der der Index des Array-Elements für beide Zugriffe gleich ist. Im folgenden wird zur Vereinfachung der Erklärung davon ausgegangen, daß eine Abhängigkeitsanalyse für True-Abhängigkeiten durchgeführt werden soll.

Durch Inversion der Schreibrelation erhält man eine Relation, die referenzierte Array-Zellen auf den Indexvektor des schreibenden Statements abbildet. Bildet man nun eine Komposition der Leserelation mit dieser invertierten Relation, so erhält man eine Relation, die den Indexraum des Quellzugriffs auf den des Zielzugriffs abbildet. Dabei gilt die gerade beschriebene Bedingung, daß die Zugriffe auf dasselbe Array-Element erfolgen, da durch die Komposition die referenzierten Array-Indizes gleichgesetzt werden. Ist die sich ergebende Relation leer, also nicht erfüllbar, so gibt es keine Belegung der Vektoren, die zum Konflikt führen könnte. Die Zugriffe sind somit unabhängig voneinander.

Als Beispiel dient wieder das Beispielprogramm 4.2, und betrachtet wird der (einzige) Lesezugriff $A[i - l]$ an der Kante zwischen Knoten 14 und 15. Der Schreibzugriff auf das Array A erfolgt mit $A[3i + j]$ auf der Kante zwischen Knoten 7 und 8. Bei Betrachtung der Indexfunktionen der einzelnen Zugriffe fällt auf, daß jeweils sämtliche Indizes im Eingangs-Tupel der Relation erscheinen. Die Begründung dafür liegt darin, daß das im Rahmen dieser Arbeit implementierte Analyseprogramm das in Abschnitt 4.2.3 verdeutlichte Problem dadurch löst, daß der Indexraum für jeden Zugriff *alle* im Eingabeprogramm enthaltenen Indexvariablen enthält, auch wenn diese für den Zugriff möglicherweise nicht umgebend sind. Dadurch haben im weiteren Programmablauf die zu verknüpfenden Relationen immer dieselbe Kardinalität und Operationen wie Schnittmenge oder Vereinigung können darauf durch Omega ausgeführt werden.

Im Beispielprogramm 4.2 sehen die jeweiligen Relationen folgendermaßen aus (x bzw. y ist der Index des Zugriffs):

- Die Leserelation:

  $$[\text{ i, j, k, l, 12}] \rightarrow [\text{ x, 1}]\text{: l} + \text{x} = \text{i} \wedge 0 \leq \text{i} \leq 20 \wedge 0 \leq \text{l} \leq 22$$

- Die Schreibrelation:

  $$[\text{ i, j, k, l, 1}] \rightarrow [\text{ y, 1}]\text{: y} = 3\text{i} + \text{j} \wedge 0 \leq \text{i} \leq 20 \wedge 0 \leq \text{j} \leq 30 \wedge 0 \leq \text{k} \leq 25$$

- Das Inverse der Schreibrelation:
  Wie oben beschrieben, erhält man hier eine Abbildung der referenzierten Array-Zellen auf den Indexvektor des Quellzugriffs.

  $$[\text{ y', 1}] \rightarrow [\text{ i', j', k', l', 1}]\text{: y'} = 3\text{i'} + \text{j'} \wedge 0 \leq \text{i'} \leq 20 \wedge 0 \leq \text{j'} \leq 30 \wedge 0 \leq \text{k'} \leq 25$$

- Die Komposition der Inversion mit der Leserelation:
  Man erhält die Relation, die (falls erfüllbar) die möglichen Belegungen der Quell- und Zielzugriffsvektoren zeigt, die den Zugriff auf das identische Array-Element bewirken.

$$[\,i, j, k, l, 12] \rightarrow [\,i', i - l - 3i', k', l', 1]:\ 0 \leq k' \leq 25 \wedge i \leq 20 \wedge l + 3i' \leq i \wedge 0 \leq l \wedge 0 \leq i'$$

Die sich ergebende Konfliktrelation ist erfüllbar, die Zugriffe stehen also in Konflikt zueinander.

Diese Relationen müssen vor der Graphabarbeitung in einem eigenen Durchlauf berechnet und bei den Zugriffen mit abgelegt werden. Der Algorithmus für diesen Durchlauf ist Algorithmus 4.1. Für die Darstellung des Algorithmus zur Belegung der Kanten wird wiederum davon ausgegangen, daß eine Abhängigkeitsanalyse für True-Abhängigkeiten durchgeführt werden soll.

Algorithmus 4.1

```
für (alle Kanten des Graphen)
{
  für (alle Lesezugriffe an der Kante)
  {
    für (alle Kanten des Graphen)
    {
      für (alle Schreibzugriffe an der Kante)
      {
        wenn (Zugriff erfolgt auf gleiches Array)
        {
          bestimme die Konfliktrelation;
          wenn (Konfliktrelation erfüllbar)
          {
            schreibe die Konfliktrelation an die Kante;
            markiere die Kante als Konfliktkante;
          }
        }
      }
    }
    veranlasse die Graphabarbeitung für diesen
    Lesezugriff
  }
}
```

4.3.3.2 Aktuelle Ordnung

Ein kurzer Einblick in die Bedeutung der *aktuellen Ordnung* wurde schon zu Beginn des aktuellen Kapitels gegeben. Es wird eine lexikografische Ordnung benötigt, die an jeder Stelle des Graphen berechnet werden kann und eine Relation zwischen verschiedenen Iterationen umgebender Schleifen bietet.

Idee

Während der gesamten Abhängigkeitsanalyse für einen bestimmten Zugriff sind nur solche Indexvariablen für die Ordnung relevant, die den Zielzugriff umgeben. Die Graphabarbeitung erfolgt entgegengesetzt zu der Abarbeitung des Eingabeprogramms und damit *von innen nach außen* im Kontrollflußgraphen. Das heißt, innere Schleifen werden zuerst rückwärts durchlaufen, bevor äußere rückwärts durchlaufen werden. An einer Rückwärtskante findet ein Schleifeniterationswechsel statt, was sich in einer Änderung der aktuellen Ordnung auswirkt. Die referenzierte Indexvariable ist aktuell kleiner als vor dem Durchlaufen der Rückwärtskante. Genau diese *ist-kleiner*-Beziehung muß in der Relation ersichtlich sein. Sobald eine äußere Schleife rückwärts durchlaufen wird, ist die Ordnung auf inneren Indizes nicht mehr relevant, da sich Quell- und Zielzugriff in verschiedenen Iterationen des äußeren Index befinden und somit bereits die *lexikografisch-kleiner*-Ordnung erfüllt ist.

Diese Idee wird nun in abstrakter Form kurz an Beispielprogramm 4.3 illustriert:

Programm 4.3

```
DO i=... (1)
  DO j=... (2)
    DO k=... (3)
      ....
    END DO
  END DO
END DO
```

Programm 4.3 ist perfekt verschachtelt und enthält die Schleifen 1 bis 3 (siehe Kontrollflußgraph in Abbildung 4.2) mit den dazugehörenden Indexvariablen i,j,k. Es werden nun die Indexvariablen an den Rückwärtskanten zum aktuellen Berechnungszeitpunkt ($<$Variablenname$>$') mit denen eines in der innersten Schleife angenommenen Zielzugriffs (<Variablenname>) in Relation gesetzt. Am Zielzugriff wird zunächst folgende Ordnung bestimmt:

$$i' = i, j' = j, k' = k$$

Die Belegung der Variablen zum aktuellen Berechnungszeitpunkt ist dieselbe wie die des Zugriffs, da noch keine Rückwärtskante durchlaufen wurde.

Beim Durchlauf der Rückwärtskante zu Schleife 3 ergibt sich:

$$i' = i, j' = j, k' < k$$

Da die k-Rückwärtskante nun durchlaufen wurde, ist der Wert von k zum aktuellen Zeitpunkt (k') kleiner als der Zugriffswert von k. Die beiden anderen Schleifen wurden (noch) nicht durchlaufen, deshalb sind deren Indexvariablen gleich.

An der Rückwärtskante zu Schleife 2 wird die Ordnung der innersten Indexvariablen irrelevant und die aktuelle Ordnung lautet demzufolge:

$$i' = i, j' < j, \text{k und k' in beliebiger Konstellation (wird weggelassen)}$$

Hier ist tatsächlich nur wichtig, daß die Rückwärtskante der zweiten Schleife durchlaufen wurde und somit j' kleiner ist als j. Die weiteren Berechnungen erfolgen analog.

Für diese Berechnungen müssen an jeder Kante des Kontrollflußgraphen die umgebenden Schleifen der Kante bekannt sein.

Bestimmung der umgebenden Schleifen für jede Kante

Die umgebenden Schleifen für sämtliche Kanten werden in einem Durchlauf über den Kontrollflußgraphen bestimmt. Durch den Algorithmus 4.2 wird der Graph in Tiefensuche durchlaufen, und an jeder Kante die umgebenden Schleifen in einem Vektor abgelegt. Für die Korrektheit des Algorithmus bekommen die Knoten außerdem einen *besucht*-Indikator.

Im bereits aufgezeichneten Kontrollflußgraphen führt das zu der in Abbildung 4.3 gezeigten Kantenbelegung (die Numerierung der Kanten - in fettgedruckten Zahlen - wird hier eingeführt, damit diese referenziert werden können).

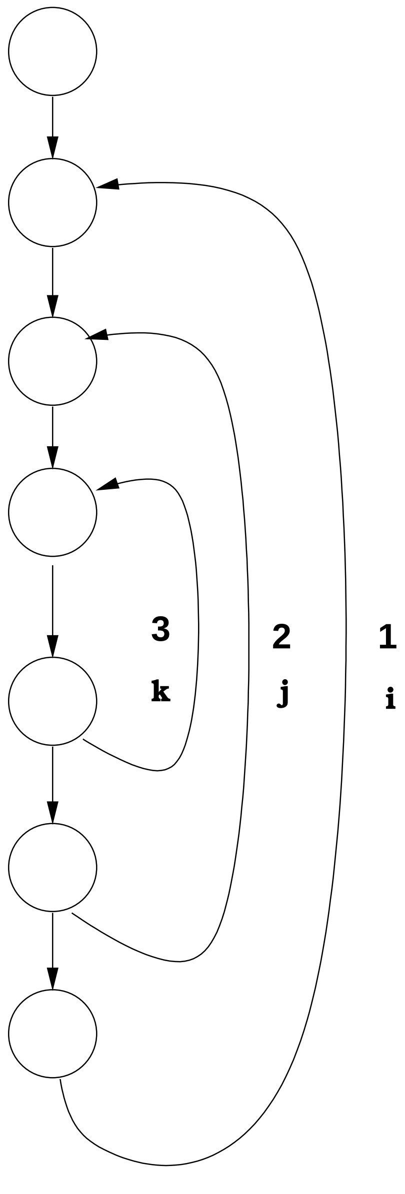

Abbildung 4.2: Kontrollflußgraph zu Programm 4.3

Algorithmus 4.2

```
lege die Ausgangskante des Eingangs-Knotens auf den
Kanten-Stack
solange (Kanten-Stack nicht leer)
{
  nimm die oberste Kante vom Stack
  nimm den Knoten an der Spitze der Kante
  wenn (der aktuelle Knoten wurde noch nicht besucht
        und ist nicht vom Typ ‘‘exit’’)
  {
    setze das besucht-Flag am Knoten
    für (alle Ausgangskanten des Knoten)
    {
      kopiere den Index-Vektor der aktuellen Kante an
       die Ausgangskante
      wenn (die Kante führt in eine neue Schleife)
      {
        füge in den Kantenvektor dieser Kante zusätzlich
        den Index an der Kante ein
      }
      lege die Kante auf den Kanten-Stack
    }
  }
}
```

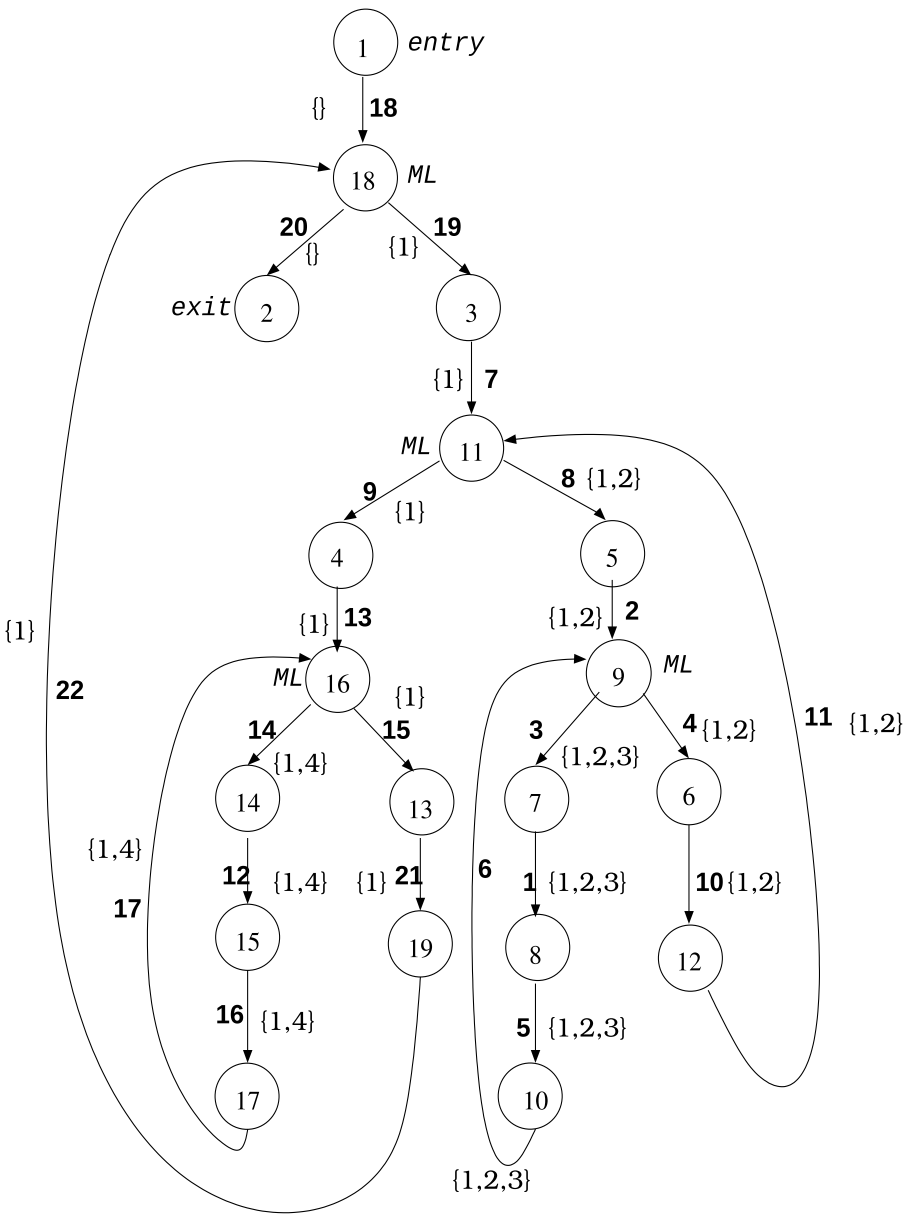

Abbildung 4.3: Kontrollflußgraph zu Programm 4.2 mit Kantenvektorbelegung

Bestimmung der aktuellen Ebene

Bevor beschrieben wird, wie die aktuelle Ordnung an jeder Kante des Graphen bestimmt werden kann, wird der Begriff der *aktuellen Ebene* eingeführt. Diese Ebene beschreibt den Schleifenindex, bezogen auf die umgebenden Schleifen des Zielzugriffs, der als letzter zurückgezählt wurde. Sie hält folglich den Index der Schleife, deren Zugriffe auf Abhängigkeiten untersucht werden, falls noch keine der sie umgebenden Schleifen durch eine Rückwärtskante betreten wurde.

Wenn eine Rückwärtskante durchlaufen wird, so wird die aktuelle Ebene dann und nur dann auf ihren Index gesetzt, wenn dieser Index

- auch ein umgebender Schleifenindex des betrachteten Zielzugriffs ist und
- die aktuelle Ebene nicht bereits einen die Rückwärtskante umgebenden Schleifenindex repräsentiert.

Durch dieses Vorgehen wird sichergestellt, daß an jedem Punkt des Graphen der aktuell zurückgezählte und relevante Schleifenindex bekannt ist.

Berechnung der aktuellen Ordnung

Hier soll die Frage geklärt werden, wie zu einem Berechnungszeitpunkt während der Graphabarbeitung die *aktuelle Ordnung* bestimmt werden kann. Die Definition von Ordnungen zwischen verschiedenen Zugriffen wurde in Abschnitt 2.2 eingeführt. Ein Zugriff hat dann und nur dann *vor* dem aktuell betrachteten Zielzugriff stattgefunden, wenn er sich entweder in einer früheren Schleifeniteration einer gemeinsam umgebenden Schleife (*lexikografisch vor*), oder *textuell vor* dem Zielzugriff befindet.

Im Beispielprogramm 4.2 erfolgt der Schreibzugriff textuell vor dem Lesezugriff. Die einzige gemeinsam umgebende Schleife ist die mit Index i. Daraus folgt, daß die *aktuelle Ordnung* des Eingabeprogramms lediglich in Bezug auf i interessant ist. Für einen Konflikt relevante Zugriffe können sich also nur im selben i-Durchlauf befinden (textuell früher), oder in einer früheren i-Iteration (lexikografisch früher).

Die aktuelle Ordnung beschreibt, welche Bedingungen in Bezug auf Schleifenindizes an einer bestimmten Kante des Graphen gelten. Diese Bedingung wird dargestellt als Beziehung zwischen dem Iterationsvektor zum Zeitpunkt des Zielzugriffs (Vektor aus *In_ <Index>*) und dem zum Zeitpunkt des Quellzugriffs (Vektor aus *Out_ <Index>*).

Wird die Rückwärtskante einer Schleife durchlaufen, so ist ein eventuell zu bearbeitender Quellzugriff, der bei der weiteren Bearbeitung berücksichtigt wird, früher einzuordnen als der aktuell betrachtete Zielzugriff. In der in dieser Arbeit verwendeten Notation wird das ausgedrückt als:

$$\mathrm{Out\_x} < \mathrm{In\_x}$$

wobei x der Schleifenindex der Rückwärtskante ist.

An der Rückwärtskante einer Schleife wird die aktuelle Ordnung mit Hilfe von Algorithmus 4.3 gefunden. Nach der Ausführung des Algorithmus besteht die aktuelle Ordnung aus einer Relation, die für alle noch nicht behandelten umgebenden Schleifenindizes des Zielzugriffs, die auch im Kantenvektor der aktuell betrachteten Kante enthalten sind, Gleichheitsrelationen enthält und für die aktuelle Ebene eine $<$-Relation.

Der Algorithmus wird nur ausgeführt, wenn der Index an der Rückwärtskante im Schleifenvektor des Zielzugriffs ist. Ist diese Bedingung erfüllt und die aktuelle Ebene ist nicht Element des aktuellen Schleifenvektors, dann wurde diese Kante noch nicht zurückgezählt. Der aktuelle Schleifenindex wird in diesem Fall zurückgezählt und die aktuelle Ebene mit diesem Index belegt.

Durch dieses Vorgehen wird sichergestellt, daß eine Schleife, die den Zielzugriff nicht umgibt, die aktuelle Ordnung nicht beeinflußt. Da eine solche Schleife zu einem früheren Zeitpunkt ausgeführt wird als der Zielzugriff und keine Überschneidung der Indizes vorliegt, muß keine lexikografische Ordnung zwischen Iterationen dieser Schleife berücksichtigt werden.

Algorithmus 4.3

```
solange ( Elemente im Kantenvektor vorhanden und aktuelles
          Element x ist nicht gleich der aktuellen Ebene )
{
  wenn (es gibt noch keine Relation)
    bilde eine neue Relation
  verknüpfe durch UND Out_x = In_x
}
// für die aktuelle Ebene
füge der Relation die Ungleichung
Out_aktuelleEbene < In_aktuelleEbene hinzu.
```

Zusätzlich bewirkt dieses Vorgehen, daß auch Schleifen, die innerhalb der aktuell zurückgezählten, also die aktuelle Ordnung bestimmenden Schleife liegen, keine Rolle mehr für die aktuelle Ordnung spielen (vgl. Abschnitt 4.3.3.2).

Im Beispielprogramm 4.2 gelten die Belegungen der aktuellen Ebene und der aktuellen Ordnung beim Durchlauf wie in Tabelle 4.1 dargestellt.

Kante	aktuelle Ebene	aktuelle Ordnung
12	0	$Out\_1 = In\_1 \wedge Out\_4 = In\_4$
17	4	$Out\_1 = In\_1 \wedge Out\_4 < In\_4$
22	1	$Out\_1 < In\_1$

Tabelle 4.1: Aktuelle Ordnung und aktuelle Ebene an Kanten

Damit die Bestimmung der aktuellen Ordnung an sämtlichen Kanten des Graphen garantiert werden kann, bedarf es der Beachtung eines Sonderfalls. Verringert sich die Anzahl der Werte im Index-Vektor beim Übergang von einer Kante zur nächsten, so ist die aktuelle Ordnung neu zu bestimmen. Befindet sich die aktuelle Ebene nämlich nicht mehr im Vektor der neuen Kante, so ist die Bedingung $Out\_aktuelleEbene < In\_aktuelleEbene$ nicht mehr relevant und muß aus der aktuellen Ordnung entfernt werden. Algorithmus 4.4 beschreibt dieses Vorgehen.

Algorithmus 4.4

```
für (alle Elemente x im aktuellen Kantenvektor)
{
  wenn (x ist im aktuellen Ebenenvektor)
  {
    füge der Relation die Gleichung Out_x = In_x hinzu
  }
}
```

Tabelle 4.2 zeigt exemplarisch am Beispielprogramm 4.2, wie sich die aktuelle Ordnung beim Übergang von der Kante 14 auf die Kante 13 verändert.

Kante	aktuelle Ebene	aktuelle Ordnung
14	4	$Out\_1 = In\_1 \wedge Out\_4 < In\_4$
13	4	$Out\_1 = In\_1$

Tabelle 4.2: Aktuelle Ordnung und aktuelle Ebene an Kanten - Sonderfall

4.3.3.3 Abhängigkeitsanalyse im Graphen

Dieser Abschnitt erläutert den Graphdurchlauf während der Abhängigkeitsanalyse. Um Aussagen über die Korrektheit des Graphabarbeitungsalgorithmus treffen zu können, wird zunächst kurz beschrieben, wie die Abhängigkeiten beim Durchlauf in das Ergebnis der Analyse gelangen. Eine detaillierte Beschreibung der Bestimmung der *Abhängigkeitsrelation* wird später bei der Beschreibung der Kantenabarbeitung geliefert.

Hinzunehmen von Abhängigkeiten

Beim Durchlauf des Graphen werden immer dann, wenn eine Konflikt-Kante bearbeitet wird, die abhängigen Definitionen desselben Array-Elements zum bisherigen Ergebnis hinzugenommen, wenn die Konfliktrelation geschnitten mit der aktuellen Ordnung (siehe Abschnitt 4.3.3.2) eine erfüllbare Belegung besitzt. Durch das Rückwärtsabarbeiten wird sichergestellt, daß man zuerst den Zugriff, der dem betrachteten am nächsten liegt, findet. Das ist der letzte Zugriff, der vor dem betrachteten Zugriff stattfindet. Es liegt hier eine direkte Abhängigkeit (siehe Abschnitt 2.3.4) vor, die in die Menge der abhängigen Zugriffe übernommen wird. Wie sichergestellt werden kann, daß immer nur solche direkten, im folgenden auch oft *optimal* genannten, Abhängigkeiten betrachtet werden, wird in Abschnitt 4.3.4 ausführlich betrachtet.

Nun soll gezeigt werden, wie der Algorithmus der Graphabarbeitung funktioniert.

Konkrete Graphabarbeitung

Da der Kontrollflußgraph gerichtet ist, haben die Kanten ein Ende und eine Spitze. Zur Vereinfachung wird im folgenden von dem *Quellknoten* gesprochen, wenn er sich am Ende der Kante befindet und vom *Zielknoten*, wenn er sich an der Spitze der Kante befindet.

Durch die bisher vorgestellen Algorithmen werden die Konfliktrelationen zu dem aktuell zu betrachtenden Zielzugriff an den zugehörigen Kanten gespeichert. Die Abhängigkeitsanalyse für diesen Zugriff erfolgt nun durch *Rückwärtsdurchlaufen* des Graphen (siehe Abschnitt 4.3.3). Ausgehend von der Kante, an der der Zielzugriff erfolgt, wird der Graph nach Algorithmus 4.5 durchlaufen. Die folgenden Abschnitte verdeutlichen den Ablauf und die Effizienz des Algorithmus. Zunächst wird die Vorgehensweise an den Knoten und an den Kanten erklärt.

Knotenabarbeitung

Die Knoten des Kontrollflußgraphen stellen Zustände während der Programmausführung dar. Sie vereinigen die Relationen der in sie mündenden Kanten.

Kantenabarbeitung

Die Abarbeitung der Kanten ist der zentrale Punkt der Abhängigkeitsanalyse. Es gibt in den hier betrachteten Kontrollflußgraphen drei Arten von zu behandelnden Kanten:

- ***Empty-Kanten:***
  Das sind Kanten, an denen keine Operationen durchgeführt werden. Darunter werden auch if-Abfrage-Kanten gezählt. An diesen Kanten wird bei ihrer Abarbeitung lediglich die Relation des Zielknotens übernommen.

- ***Sloop-Kanten:***
  Diese Kanten stellen die Rückwärtskanten des Graphen dar, also solche, die von einer Iteration einer Schleife in die vorhergehende führen. An einer solchen Kante wird die aktuelle Ebene neu belegt, falls die betroffene Schleife auch den Zielzugriff umgibt und noch keine die Sloop-Kante umgebende Schleife rückwärts durchlaufen wurde (siehe Abschnitt 4.3.3.2).

- ***Konflikt-Kanten:***
  Diese Kanten enthalten Operationen, die Zugriffskonflikte verursachen. An ihnen wird die Konflikt-Relation abgespeichert.

  Bei der Abarbeitung wird die Schnittmenge der Konfliktrelation mit der aktuellen Ordnung gebildet, so daß eine Relation entsteht, die die tatsächliche Abhängigkeit des Zielzugriffs von der momentan betrachteten Quelle enthält (Abhängigkeitsrelation).

  Für das Beispielprogramm 2.2 hätte die Zuweisungskante $S2$ der Schleife folgende Konfliktrelation (wiederum werden True-Abhängigkeiten betrachtet):

  $$[i,2] \rightarrow [i',2]\ i = 1+i' \wedge 1 \leq i' \leq 9$$

  Zum Zeitpunkt nach dem Rückwärtsdurchlaufen der Schleifenkante, wäre die aktuelle Ordnungsrelation:

  $$[i,2] \rightarrow [i',2]\ i' < i$$

Das heißt, der Schreibzugriff in $S2$ gehört zu einer früheren Schleifeniteration als der Lesezugriff in $S2$.
Nach dem Bilden der Schnittmenge resultiert:

$$[i,2] \rightarrow [i',2] \; i = 1+i' \land 1 \leq i' \leq 9$$

Das Ergebnis ist genau das intuitiv erwartete: Die Zugriffe sind voneinander abhängig genau dann, wenn der Index i des Lesezugriffs um eins größer ist als der Index i' des Schreibzugriffs.

Die resultierende Relation muß zu der am Zielknoten der Kante hinzugenommen werden, sofern sie nicht schon in dieser enthalten ist. Dabei wird die in Abschnitt 4.3.4 eingeführte *Optimierung* vorgenommen.

Algorithmus

In Algorithmus 4.5 wird ein zu bearbeitender Knoten-Stack verwendet, auf den zur Initialisierung außer bei der Berechnung von Anti-Abhängigkeiten (man beachte Abschnitt 4.5) der Quellknoten der Zugriffskante gelegt wird. Für jeden Knoten auf diesem Stack werden die Eingangskanten betrachtet und die Kantenberechnungen werden durchgeführt. Dabei werden immer zuerst die Rückwärtskanten von Schleifen durchlaufen bevor die Schleife verlassen wird, um eine äußere Schleife abzuarbeiten. Nach der Bearbeitung einer Kante wird ihr Quellknoten auf den Stack gelegt, so daß die Berechnungen entlang den Pfaden im Kontrollflußgraphen stattfinden und damit der Ablauf des Eingabeprogramms in entgegengesetzter Richtung nachgestellt wird.

Komplexität

Algorithmus 4.5 hat eine Komplexität von $O(n^2)$, wobei n die Anzahl perfekt verschachtelter Schleifen ist.

Begründung:
Ohne Schleifen würde jede Kante des Graphen genau einmal durchlaufen werden. Wenn der Algorithmus zum ersten Mal auf eine Rückwärtskante, also eine Schleife stößt, durchläuft er diese. Daß er tatsächlich zum ersten Mal an diese Rückwärtskante kommt wird durch die Bedingung, daß der Knoten-Stack leer sein muß, gesichert. Nachdem die Schleife abgearbeitet, also komplett durchlaufen wurde, wird die Rückwärtskante wieder als “nicht bearbeitet” markiert, damit sie nochmal durchlaufen werden kann, falls sie von einer weiteren Schleife umgeben wird. Jede Kante wird also m mal durchlaufen, wobei m die Anzahl der sie umgebenden Schleifen ist.

Im Beispielprogramm 4.2 würden die Kanten 1 und 5 beispielsweise dreimal durchlaufen werden, die Kanten 2 und 13 aber nur zweimal.

Algorithmus 4.5

```
nimm den obersten Knoten vom Stack
wenn (Knoten hat zwei Eingangskanten)
{
  wenn (Knoten ist Merge-Knoten)
  {
    wenn (Knoten-Stack ist leer)
    {
      wenn (Rückkante noch nicht behandelt)
      {
        markiere die Kante als bearbeitet
        führe Kantenberechnung aus
        führe Knotenberechnung des Quellknotens der Kante aus
        lege Quellknoten auf den Bearbeitungsstack
      }
      sonst
      {
        setze die behandelt-Markierung zurück
        führe Kantenberechnung an der anderen Kante aus
        führe Knotenberechnung des Quellknotens der Kante aus
        lege Quellknoten auf den Bearbeitungsstack
      }
    } // Knoten-Stack leer
  } // Knoten ist Merge-Knoten
  sonst
  {
    für (beide Eingangskanten)
    {
      markiere die Kante als bearbeitet
      führe Kantenberechnung an der Kante aus
      führe Knotenberechnung am Quellknoten der Kante aus
      lege Quellknoten auf den Bearbeitungsstack
    }
  }
} //Knoten hat zwei Eingangskanten
sonst
{
  führe für die Eingangskante die Kantenberechnung durch
  führe die Knotenberechnung für den Quellknoten der Kante aus
  lege Quellknoten auf den Bearbeitungsstack
}
```

Terminierung

Der Algorithmus terminiert am Eingangsknoten des Graphen, da dieser keine Eingangskanten besitzt, der Stack der zu bearbeitenden Knoten also leer wird.

Korrektheit

Der Algorithmus liefert korrekte Ergebnisse, da jede Kante so oft durchlaufen wird, wie sie umgebende Schleifen hat, und dadurch jeder an den Kanten enthaltene Zugriff unter jeder möglichen aktuellen Ordnung betrachtet wird. Für das Ziel der Datenabhängigkeitsanalyse bedeutet das:
Jeder Quellzugriff, der *vor* dem Zielzugriff stattfindet, wird auf Konflikte untersucht; erfüllbare Konfliktrelationen werden aufgesammelt, so daß sämtliche Quellzugriffe im Endergebnis enthalten sind.

4.3.3.4 Lösungsansatz ohne Betrachtung der aktuellen Ordnung

Wie bereits bei der Graphabarbeitung in Abschnitt 4.3.3.3 erläutert, sind für die Analyse von Abhängigkeiten nur solche Zugriffe interessant, die *vor* dem aktuell betrachteten Zielzugriff erfolgt sind. Es liegt nahe zu vermuten, daß zur Bestimmung solcher Zugriffskonflikte die aktuelle Ordnung nicht unbedingt notwendig wäre, sondern lediglich durch die Art der Abarbeitung, nämlich durch einen Rückwärtsdurchlauf durch den Graphen, diese Bedingung gesichert wäre.

Programm 4.4

```
DO i=1,10
  A[i] = 5
  B = A[i] + 1
END DO
```

Für die True-Abhängigkeiten in Beispielprogramm 4.4 erhält man ohne Betrachtung der aktuellen Ordnung das Ergebnis:

$$[\ i,\ 2] \rightarrow [\ i,\ 1]:\ 1 \leq i \leq 10$$

Dieses Ergebnis ist korrekt.

Bereits bei Programm 4.5 aber resultiert ein nicht korrektes Ergebnis.

Programm 4.5

```
DO i=1,10
  A[i] = A[i] + 1
END DO
```

Ergebnis:

$$[\text{ i}, 1] \rightarrow [\text{ i}, 1]\text{: } 1 \leq \text{i} \leq 10$$

Korrekt wäre in diesem Fall, daß es keine Abhängigkeiten gibt, da der Schreibzugriff immer erst nach dem Lesezugriff erfolgt.

Durch das Weglassen der aktuellen Ordnung wird hier vernachlässigt, daß der Index i nach dem Rückwärtsdurchlaufen der Schleifenkante nicht mehr derselbe ist, sondern bei der neuen Betrachtung des Zugriffs kleiner sein muß.

Dieses Gegenbeispiel zeigt, daß es nicht möglich ist, korrekte Abhängigkeitsanalysen ohne Betrachtung der aktuellen Ordnung durchzuführen.

4.3.4 Optimierung

Unter *Optimierung* versteht man in diesem Zusammenhang, daß im Ergebnis einer Abhängigkeitsanalyse nur solche Zugriffe auftauchen, die nicht *vor* dem Zielzugriff nochmal überschrieben werden. Um dieses Ziel zu erreichen, sind im Wesentlichen zwei Schritte notwendig:

- Für eine aktuell betrachtete Operation, die mehrere Zugriffe als potentielle Quellen enthält, wird nur der Zugriff zur Abhängigkeitsrelation hinzugenommen, der tatsächlich dem Zielzugriff am nächsten ist, also nicht mehr überschrieben werden kann.
- Es werden nur Zugriffe auf Elemente hinzugenommen, für die es noch keine *Definition* gibt. Mit Definition ist gemeint, daß es für einen Zielzugriff auf ein Element bereits einen zugehörigen Quellzugriff gibt.

Diese beiden Punkte werden im Folgenden getrennt betrachtet.

4.3.4.1 Finden des lokalen Maximums

Für Zugriffe innerhalb von Schleifen wurde bislang nur danach unterschieden, ob sie textuell vor einem anderen Zugriff passieren, bzw. lexikografisch vorher. Um zu bestimmen, welcher Zugriff tatsächlich *als letzter* vor dem Zielzugriff passiert, muß das *Maximum* der Zugriffe in den Schleifeniterationen gefunden werden.

In Relationen ausgedrückt heißt das, daß es keinen Zugriff mit einer Relation geben darf, der dieselben Bedingungen enthält wie das vermeintliche Maximum und in einer Indexdimension einen höheren Wert enthält.

Dieses Vorgehen wird nun an einem Beispiel verdeutlicht.

Für Programm 4.6 sollen die True-Abhängigkeiten festgestellt werden.

Programm 4.6

```
CONSTANT n;

DO i= 0, n-1
  A[0] = i      S1
END DO
S = A[0]        S2
```

Beim Durchrechnen des Graphen ergibt sich an der Schreibzugriffskante $S1$ die Abhängigkeitsrelation (Konfliktrelation des Zugriffs geschnitten mit der aktuellen Ordnung):

$$[\mathrm{S2}] \rightarrow [\mathrm{i}, \mathrm{S1}]\colon\ 0 \leq \mathrm{i} < \mathrm{n}$$

Das heißt, für den Lesezugriff in $S2$ sind sämtliche Schreibzugriffe in $S1$ mögliche Quellen.
Diese Relation soll optimiert werden, und somit der letzte Zugriff bestimmt werden. Es muß gelten:

$$[\mathrm{S2}] \rightarrow [\mathrm{i}, \mathrm{S1}]\colon\ (0 \leq \mathrm{i} < \mathrm{n}) \wedge \neg\exists\ (\ \alpha\colon\ (\ \alpha \neq i \wedge 0 \leq \alpha < n))$$

Die Auswertung dieser Relation resultiert in:

$$[\mathrm{S2}] \rightarrow [\mathrm{i, S1}]\text{: n} = 1\text{+i'} \wedge 0 \leq \text{i'}$$

Das entspricht dem erwarteten Ergebnis: Der Lesezugriff erfolgt auf das Element, das in der letzten Schleifeniteration geschrieben wurde, nämlich bei $i' = n - 1$.

4.3.4.2 Hinzunahme noch nicht definierter Zugriffe

Nachdem das Maximum der Zugriffsrelation gefunden ist, werden nur solche Zugriffe zur endgültigen Ergebnisrelation einer Kante hinzugenommen, die nicht bereits in dieser enthalten sind. Dafür wird der Definitionsbereich der gerade berechneten Relation auf den Bereich beschränkt, der sich durch Wegnehmen des Definitionsbereiches der bereits bekannten Relation am Knoten ergibt. Die Begründung hierfür ist wiederum das Rückwärtsdurchlaufen des Graphen. Alle bereits berechneten Quellzugriffe haben *nach* dem aktuell betrachteten stattgefunden. Folglich sind nur solche zusätzlich noch interessant, die nicht im Definitionsbereich der bisherigen Ergebnisrelation enthalten sind.

Auch dieser Mechanismus soll durch ein Beispiel verdeutlicht werden. Betrachtet wird Programm 4.7. In diesem Programm erhält man nach der Abarbeitung der zweiten Schleife abhängige Zugriffe für $i = 1$ bis 10. Nun muß für den Zugriff $i = 0$ der Schreibzugriff in der j-Schleife durch das oben beschriebene Verfahren bestimmt werden.

Programm 4.7

```
DO j=0,5
  A[j] = 2      S1
END DO
DO i= 0, 10
  A[i+1] = 5    S2
  y = A[i]      S3
END DO
```

Die Abhängigkeitsrelation für den Zugriff $A[j]$ in $S1$ ist:

$$[\,j, i, S3] \rightarrow [\,j', i', S1]:\ i = j' \wedge 0 \leq j' \leq 5$$

Die bereits vor dieser Kante berechnete Ergebnisrelation, also die Vereinigung der bisher gefundenen Quellen, ist:

$$[\,j, i, S3] \rightarrow [\,j', i', S2]:\ i = 1{+}i' \wedge 0 \leq i' \leq 9$$

Definitionsbereich dieser Ergebnisrelation:

$$[\,j, i, S3]:\ 1 \leq i \leq 10$$

Der Definitionsbereich der neuen Relation wird wie beschrieben eingeschränkt. Es ergibt sich:

$$[\,j, i, S3] \rightarrow [\,j', i', S1]:\ j' = 0 \wedge i = 0$$

Als Endergebnis (*alte* Relation und der neue, bisher noch nicht definierte Zugriff) resultiert wie erwartet:

$$\begin{gathered}[\,j, i, S] \rightarrow [\,j', i', T]:\ j' = 0 \wedge i = 0 \wedge S = S3 \wedge T = S1 \vee \\ i = 1{+}i' \wedge S = S3 \wedge T = S2 \wedge 0 \leq i' \leq 9\end{gathered}$$

Durch diese Vorgehensweise ist sichergestellt, daß an jedem Punkt der Graphabarbeitung immer nur optimierte Ergebnisse vorliegen.

4.4 Behandlung unstrukturierter Eingabeprogramme

Dieses Kapitel widmet sich einer theoretisch möglichen Datenabhängigkeitsanalyse unstrukturierter Eingabeprogramme, z.B. solcher mit Gotos. Es exisitert noch keine implementierte Version dieses Lösungsansatzes in LooPo. Ein theoretischer Ansatz wird in [3] diskutiert. LooPo bietet aber die Voraussetzungen für eine Implementierung der Lösungsansätze noch nicht.

So wäre es zum Beispiel unerläßlich, Indexvariablen im Eingabeprogramm zu lokalisieren. Im Beispielprogramm 3.5 müßten die Inkrementierungen von

i und j als Zugriffe auf Indexvariablen identifiziert werden. Die Kanten, an denen diese Zugriffe stattfinden, müßten deshalb wie die Rückwärtskanten in for-Schleifen berechnet werden. Wenn man keine Einschränkungen an die Art der Indexvariablenzugriffe macht, dann muß die aktuelle Ordnung an diesen Kanten unter Umständen aufwendig untersucht werden. Im Beispielprogramm 3.5 ist bereits der einfache Zugriff $i = i + 2$ ein Sonderfall. Für alle Indexvariablen i gilt:

$$\mathrm{i} = \mathrm{i'} + 2^{*}\mathrm{k} \text{ mit } \mathrm{k} \geq 0.$$

Da die momentan im LooPo-Umfeld implementierten Algorithmen nicht vorsehen, daß ein Prädikat einer if-Abfrage während der Graphabarbeitung ausgewertet wird, ist es in Goto-Programmen nicht möglich an einer konkreten Stelle zu bestimmen, wie oft ein bestimmter Zweig durchlaufen wurde.

In dem Beispielprogramm 3.5 wird j in jedem der beiden möglichen Ablaufpfade inkrementiert. Wenn man in den Pfad mit der Bedingung $i \geq n$ verzweigt, wurde j bereits einige Male inkrementiert. Durch das Prädikat kann man bei der Analyse nicht bestimmen wie oft, und kennt daher keinen genauen Wert für j. In einem komplizierteren Beispiel könnte ein beliebiges Abwechseln zwischen Pfaden auftreten (wie im Programmgraph in Abbildung 3.4), so daß das Problem noch stärker zur Geltung kommt.

Auch in [3] wird nur von möglichen Ablaufsequenzen gesprochen. Es ist nicht immer möglich, den letzten Quellzugriff vor dem Zielzugriff zu bestimmen. Zudem kann kein maximaler Index als Abbruchbedingung für das Einschlagen von Goto-Pfaden angegeben werden. Die Begründung liegt wiederum in der Verwendung von Prädikaten als Endekriterium. Aus diesen Gründen ist eine Optimierung hinsichtlich des lokalen Maximums nicht möglich.

An dem besprochenen Beispiel soll nun für einen Lesezugriff gezeigt werden, wie das berechenbare Ergebnis aussehen könnte.

Betrachtet wird Lesezugriff $A[i]$ auf der Kante zwischen den Knoten 10 und 11. Die Pfade werden bezogen auf durchlaufene Knoten angegeben. Knoten 10 und 7 bringen keine Ergebnisse. An der Kante zwischen Knoten 6 und Knoten 7 wird die aktuelle Ordnung neu bestimmt und es ergibt sich:

$$\mathrm{j'} < \mathrm{j} \text{ (entspricht Out\_2} < \text{In\_2)}$$

Wie im bereits bekannten Algorithmus 4.5 wird jetzt die Rückwärtskante betrachtet(Knoten 5 nach Knoten 13). Hier wird im Unterschied zu den Rückwärtskanten der for-Schleifen keine besondere Berechnung durchgeführt. Der erste Quellzugriff, der zu bearbeiten ist, ist $A[2 * j]$. Für eine Abhängigkeit muß gelten:

$$\mathrm{i} = 2 * \mathrm{j'}$$

Der Pfad über Knoten 12, 11 und 10 bis Knoten 7 ergibt keine neuen Ergebnisse. An der Kante zu Knoten 6 muß keine neue Berechnung der aktuellen Ordnung durchgeführt werden, da die allgemeine Ordnung $j' < j$ weiterhin gültig ist. Der Schreibzugriff an der Kante zwischen Knoten 5 und 4 ist wieder eine potentielle Quelle. Es muß gelten:

$$\mathrm{i'} = \mathrm{i}$$

Da i bisher nicht referenziert wurde, gilt diese Bedingung an dieser Stelle. Alle Zugriffe, die noch nicht in dem bisherigen Ergebnis enthalten sind, werden zur Ergebnisrelation hinzugenommen.

An der nächsten Kante wird i referenziert. Hier muß wie oben beschrieben die aktuelle Ordnung genauer spezifiziert werden:

$$\mathrm{i} = \mathrm{i'} + 2*\mathrm{k}, \text{ mit } \mathrm{k} \geq 1$$

Die Rückwärtskante wird beschritten und man stößt auf den möglichen Quellzugriff $A[i + 3]$. Hier muß gelten:

$$\mathrm{i} = \mathrm{i'} + 3 \wedge \mathrm{i} = \mathrm{i'} + 2*\mathrm{k} \wedge \mathrm{k} \geq 1$$

Daraus ergibt sich, daß dieser Zugriff keine Abhängigkeiten zum Zielzugriff enthält, da sich die Bedingungen gegenseitig ausschließen. Die Ergebnisrelation ist folglich an dieser Stelle leer.

Da die Ordnung für den speziellen Kontrollflußgraphen des Beispielprogramms nicht mehr verändert wird und keine weiteren Quellzugriffe vorhanden sind, ist das Gesamtergebnis:

$$\{[\mathrm{\ i, j, S6}] \rightarrow [\mathrm{\ i', j', S7}]\text{: } \mathrm{i} = 2 * \mathrm{j'}\} \cup$$
$$\{[\mathrm{\ i, j, S6}] \rightarrow [\mathrm{\ i', j', S2}]\text{: } \mathrm{i'} = \mathrm{i}\}$$

Enthält ein Eingabeprogramm unstrukturierte Teile mit umgebenden Schleifen, so erfolgt die Berechnung der aktuellen Ordnung analog der Berechnung in strukturierten Eingabeprogrammen: Wenn eine äußere Schleife zurückgezählt wird, spielen Ordnungen innerhalb dieser Schleife keine Rolle mehr, da sie bereits berücksichtigt wurden. Solange Sprünge nur innerhalb von Schleifen vorkommen, ist dieses Prinzip für Goto-Programme korrekt. Gibt es Sprünge zwischen verschiedenen Schleifen, so muß ein Verfahren entwickelt werden, das Pfade als *für die momentane Schleifenordnung schon betrachtet* markiert, damit nicht endlos zwischen den Schleifen hin- und hergesprungen wird.

Es erscheint schwer, eine korrekte Analyse bei von der Abhängigkeitsanalyse nicht bestimmbarem Verhalten des Eingabeprogramms durch solche *Schleifensprüngen* durchzuführen. Es können nicht sämtliche Permutationen von Pfadkombinationen berücksichtigt werden, demnach können lediglich unvollständige oder nicht optimierte Ergebnisse erzielt werden.

4.5 Unterschiede zwischen Abhängigkeiten

Im Rahmen des LooPo-Projekts ist es nicht vorgesehen, Input-Abhängigkeiten zu berechnen, da konkurrierende Lesezugriffe gleichzeitig, also parallel stattfinden dürfen (vgl. Abschnitt 2.3.1). Sie bedeuten keine Veränderung von Speicherbelegungen und spielen somit für die Parallelisierung keine Rolle. Zwischen der Berechnung von Output- und True-Abhängigkeiten bestehen keine wesentlichen Unterschiede. Die Anti-Abhängigkeiten dagegen bedürfen einer gesonderten Betrachtung.

Die erste Besonderheit besteht darin, daß bereits der Lesezugriff im selben Statement wie der betrachtete Schreibzugriff als möglicher Kandidat für eine Abhängigkeit in Betracht gezogen werden muß.

Der wesentlich wichtigere Unterschied besteht aber darin, daß eine Optimierung, wie sie in Abschnitt 4.3.4 beschrieben wurde, für Anti-Abhängigkeiten nicht sinnvoll ist. Im Programm 4.8 überschreibt der eine Schreibzugriff auf $A[0]$ sämtliche Lesezugriffe, die zuvor auf dieses Element stattgefunden haben. Hier sollen demnach auch sämtliche Lesezugriffe im Ergebnis enthalten sein.

Wünschenswert wäre, daß im Ergebnis nur die Lesezugriffe erscheinen, die nicht bereits durch einen anderen Schreibzugriff überschrieben wurden. In Programm 4.9 würde ohne Optimierung auch der Zugriff aus $S1$ in das Ergebnis mitaufgenommen werden. Die aktuelle Version von Cffada macht das ohne Optimierung ebenfalls so. Tatsächlich wurde aber der Zugriff aus $S1$ in $S2$ wieder überschrieben und es besteht somit keine Abhängigkeit mehr zu

Programm 4.8

```
CONSTANT n;

DO i= 0, n-1
  B[i] = A[0]
END DO
A[0] = 10
```

dem Schreibzugriff in $S4$.

Programm 4.9

```
CONSTANT n;

S = A[0]         S1
DO j=0,10
  A[i] = 1       S2
END DO
DO i= 0, n-1
  B[i] = A[0]    S3
END DO
A[0] = 10        S4
```

Bei der optimierten Analyse von Anti-Abhängigkeiten müßten auch Schreibzugriffe mitberechnet werden. Dazu wären einige Programmerweiterungen an Cffada notwendig, die im Rahmen dieser Arbeit nicht vorgesehen sind.

Kapitel 5

Probleme durch Prädikate

Auf den Einsatz von Prädikaten unter Verwendung von Omega wurde bereits in Abschnitt 4.2.2 eingegangen. das verwendete Hilfswerkzeug Omega bietet die Möglichkeit, Prädikate mit Parametern zu erzeugen und dann zu verwenden.

5.1 Problemstellung

In Abschnitt 4.2.3 wurde der Nachteil des Einsatzes von Omega aufgezeigt: Man kann bei der Verwendung von Prädikaten nur die Anzahl der gewünschten Parameter angeben. Omega numeriert diese, so daß sie wie Indexvariablen aussehen (zum Beispiel *In1*). Da die Prädikate häufig von Indexvariablen abhängig sind, wären Indexvariablen als Parameter kein Problem, unter der Voraussetzung, daß man diejenigen als Paramter angeben könnte, von denen das Prädikat tatsächlich abhängt. Ein Beispiel für diese Problemstellung bietet Programm 5.1.

Sobald ein Index als Parameter in einem Prädikat erscheint, wird er in die Berechnungen miteinbezogen. Das führt zu einigen Fallunterscheidungen in den Konfliktrelationen und auch in den Abhängigkeitsberechnungen, die nicht nötig wären. Da man aber die Parameter der Prädikate nicht explizit angeben kann, muß immer die Gesamtanzahl der Indexvariablen als Parameteranzahl angegeben werden, um eine korrekte, wenn auch zu genaue, Berechnung zu gewährleisten.

Während der Berechnung der Abhängigkeiten führen diese Fallunterscheidungen zu einem stark erhöhten Rechenaufwand. Das Ergebnis hat einen

Programm 5.1

```
CONSTANT n;

DO i = 0,20
  if (P)             S1
    A[3*i] = A[i]    S2
END DO
DO j = 0,20
  if (L)             S3
    A[j] = A[j-2]    S4
END DO
```

proportional zur Anzahl der Fallunterscheidungen größeren Umfang, der vermeidbar wäre, würde das Werkzeug Omega eine genaue Spezifizierung der Parameter erlauben.

Bei der Konfliktbildung zwischen dem Zielzugriff $A[j-2]$ in $S4$ und dem Quellzugriff $A[j]$ ebenfalls in $S4$ im Beispielprogramm 5.1 ergibt sich als Ergebnis durch die Verwendung von Omega (die Prädikate werden nach dem Statement, in dem sie vorkommen, numeriert):

$$\begin{gathered}
[\text{ i, j, 4}] \rightarrow [\text{ i', j', 4}]: \\
\text{j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{i'} < \text{i} \wedge \text{pred3(i', j')} = \text{T} \wedge \text{pred3(i, j)} = \text{T} \\
\vee \\
\text{j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{i} < \text{i'} \wedge \text{pred3(i', j')} = \text{T} \wedge \text{pred3(i, j)} = \text{T} \\
\vee \\
\text{j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{i} = \text{i'} \wedge \text{pred3(i', j')} = \text{T} \wedge \text{pred3(i, j)} = \text{T}
\end{gathered}$$

Das erwartete Ergebnis wäre dagegen:

$$\begin{gathered}
[\text{ i, j, 4}] \rightarrow [\text{ i', j', 4}]: \\
\text{j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{pred3(i', j')} = \text{T} \wedge \text{pred3(i, j)} = \text{T}
\end{gathered}$$

Der Unterschied zwischen dem erwarteten und dem tatsächlichen Ergebnis liegt in den Fallunterscheidungen für i. Da weder der Quell- noch der Zielzugriff i beinhalten, bzw. die Schleife mit Index i beide Zugriffe nicht umgibt, sind die Fallunterscheidungen nicht notwendig. Sie werden von Omega eingefügt, da der Index als Parameter im Prädikat erscheint. Warum i hier überhaupt in der Relation erscheint, wird in Abschnitt 4.3.3.1 begründet.

Das gewünschte Ergebnis der Konfliktbildung wäre:

$$[\text{ i, j, 4}] \rightarrow [\text{ i', j', 4}]:$$
$$\text{j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{pred3(j')} = \text{T} \wedge \text{pred3(j)} = \text{T}$$

Durch die unnötigen Fallunterscheidungen ergibt sich bei der *normalen* Berechnung als Endergebnis:

(1) {[i, j, 4] → [i', j', 2]: j = 2+3i' ∧ i = i' ∧ 0 ≤ i' ≤ 6 ∧ **3+3i' ≤ j'** ∧ pred1(i',j') = T ∧ pred3(i,j) = T ∧ U} ∪

(2) {[i, j, 4] → [i', j', 2]: j = 2+3i' ∧ i = i' ∧ 0 ≤ i' ≤ 6 ∧ **j' ≤ 3i'+1** ∧ pred1(i',j') = T ∧ pred3(i,j) = T ∧ U} ∪

(3) {[i, j, 4] → [i', j', 2]: j = 2+3i' ∧ i = i' ∧ 0 ≤ i' ≤ 6 ∧ **j = 2+3i'** ∧ pred1(i',j') = T ∧ pred3(i,j) = T ∧ U} ∪

(4) {[i, j, 4] → [i', j', 2]: j = 2+3i' ∧ **0, i+1 ≤ i' ≤ 6** ∧ pred1(i',j') = T ∧ pred3(i,j) = T ∧ U} ∪

(5) {[i, j, 4] → [i', j', 2]: j = 2+3i' ∧ **0 ≤ i' ≤ 6, i-1** ∧ pred1(i',j') = T ∧ pred3(i,j) ∧ U} ∪

(6) {[i, j, 4] → [i', j', 4]: 2+j' = j ∧ 2 ≤ j ≤ 20 ∧ **i' < i** ∧ pred3(i,j) = T ∧ pred3(i',j')} ∪

(7) {[i, j, 4] → [i', j', 4]: 2+j' = j ∧ 2 ≤ j ≤ 20 ∧ **i < i'** ∧ pred3(i,j) = T ∧ pred3(i',j') = T} ∪

(8) {[i, j, 4] → [i', j', 4]: 2+j' = j ∧ 2 ≤ j ≤ 20 ∧ **i' = i** ∧ pred3(i,j) = T ∧ pred3(i',j') = T}

Fälle 1 bis 3 enthalten die unnötigen Fallunterscheidungen für die Abhängigkeit von Statement $S2$ und j, nämlich $j < j'$, $j > j'$ und $j = j'$. Der Index j wird immer durch $2 + 3i'$ ersetzt, wodurch die Fallunterscheidungen nicht sofort erkennbar sind. Sie sind in der Aufzählung **fett** dargestellt. Fälle 4 und 5 handeln die verschiedenen Fälle für i ab, immer noch für den Zugriff in Statement 2.

Fälle 6 bis 8 handeln Vergleiche zwischen i und i' ab, hier wird die Abhängigkeit von Statement $S4$ betrachtet. Auch diese drei Fälle wären nicht nötig, da i das Statement $S4$ nicht als Index umgibt.

In Fall 1 tritt bereits das erste Mal *UNKNOWN*, hier kurz U auf. Dieses Auftreten entsteht durch das Bilden der *Domain*(Definitionsbereich) von Relationen und bedeutet, daß die anderen Prädikatbelegungen nicht bekannt sind. In Fall 1 wird zum Beispiel über das Prädikat $pred1(i, j)$ für den Zielzugriff keine Aussage gemacht. Das ist auch nicht notwenig für die Analyse. Durch die Parameter in Verbindung mit der Domain enthalten die Ergebnisse aber bei der Verwendung von Omega diese *Unbekannt*-Aussage.

5.2 Lösungsansätze

Im Rahmen dieser Arbeit wurde ein Ansatz verfolgt, die Ergebnisrelationen trotz des oben beschriebenen Problems auf die tatsächlich erwünschten Ausgaben zu reduzieren. Dabei wurden drei Ansätze untersucht, die im folgenden beschrieben werden.

Bemerkung:
Das Auftreten von U (=unbekannt) entsteht bei der Verwendung von Omega, wenn durch die Bildung der Domain, also dem Definitionsbereich einer Relation, Prädikate nicht mehr erscheinen und somit durch *Unbekannt* ersetzt werden müssen.

Die Folge hiervon ist, daß in den Ergebnisrelationen zwar ein Prädikat für die Indizes des Zielzugriffs und eines für den Quellzugriff vorhanden ist, aber dennoch U auftaucht.

5.2.1 Prädikate ohne Parameter

Der erste hier beschriebene Lösungsansatz besteht darin, Prädikate ganz ohne Parameter zu verwenden. In Programm 5.1 sieht dieselbe Konfliktsituation wie oben dann folgendermaßen aus:

$$[\text{ i, j, 4}] \rightarrow [\text{ i', j', 4}]\text{: j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{pred3} = \text{T}$$

Das Gesamtergebnis der Abhängigkeitsanalyse für diesen Zielzugriff lautet:

$$[\text{ i, j, 4}] \rightarrow [\text{ i', j-2, 4}]\text{: } 2 \leq \text{j} \leq 20 \wedge \text{pred3} = \text{T}$$

Dieses Resultat ist für das betrachtete Eingabeprogramm nur ein Teil des Gesamtergebnisses. Durch den Verzicht auf Parameter kann nicht ausgedrückt werden, daß dasselbe Prädikat zum Beispiel für den Zielzugriff TRUE und für den Quellzugriff FALSE liefert. Korrekt wäre, daß auch der erste Schreibzugriff eine potentielle Quelle für den Zugriff $A[j-2]$ darstellt, nämlich dann, wenn gilt:

- L ergibt TRUE für den Zielzugriff UND
- P ergibt TRUE für den Quellzugriff UND
- L ergibt FALSE für alle potentiellen Quellzugriffe

Damit ist dieser Lösungsansatz nicht immer anwendbar, da die erzeugten Ergebnisse unter bestimmten Voraussetzungen nicht korrekt sind.

Es würde helfen, das Zielprädikat ganz wegzulassen, da es im Grunde nur den Kontext darstellt und nicht Ergebnis der Analyse ist.

5.2.2 Gleichsetzen von Index-Variablen (1)

Dieser Lösungsansatz verringert die unnötigen Fallunterscheidungen dadurch, daß Indexvariablen, die beide Zugriffe *nicht* umgeben, gleichgesetzt werden. Da idealerweise diese Variablen nicht in den Übergangsbedingungen der Relationen erscheinen, wenn sie nicht in der Parameterliste des Prädikates stehen, bleibt das Ergebnis korrekt, es wird lediglich vereinfacht. Für die betrachteten Zugriffe im Beispiel 5.1 ist i nicht umgebend, also kann die in Abschnitt 5.1 dargestellte Fallunterscheidung nach $i' < / > / = i$ unterbleiben.

Algorithmus 5.1 zeigt, wie die nicht umgebenden Indexvariablen bestimmt und gleichgesetzt werden. Der Algorithmus setzt nach der Bildung der Konfliktrelation auf.

Algorithmus 5.1

```
erzeuge eine neue leere Relation mit denselben Indexvariablen
  wie in der Konfliktrelation
für (alle Indexvariablen in der Relation)
{
  wenn (die Variable ist in keinem der Index-Vektoren
        der Kanten der beiden Zugriffe)
  {
    füge der neuen Relation eine Gleichheitsrelation für die
      Variable durch UND-Verknüpfung hinzu
  }
}
schneide die ursprüngliche Konfliktrelation mit der hier
  berechneten und speichere das Ergebnis als neue
  Konfliktrelation
```

Im aktuellen Beispiel beginnt der Algorithmus mit der Konfliktrelation aus Abschnitt 5.1. Nach der Schleife über die Indexvariablen resultiert folgende Relation (das x steht für die Statementnummer, die in der ursprünglichen Relation als Variable enthalten ist, hier aber keine Rolle spielt und nicht belegt wird):

$$[\text{ i, j, x}] \rightarrow [\text{ i', j', x}]\text{: i} = \text{i'}$$

Durch das anschließende Bilden der Schnittmenge erhält man:

$$[\text{ i, j, 4}] \rightarrow [\text{ i', j', 4}]\text{:}$$
$$\text{i'} = \text{i} \wedge \text{j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{pred3(i', j')} = \text{T} \wedge \text{pred3(i, j)} = \text{T}$$

Im Endergebnis spart diese Methode so viele Fallunterscheidungen, wie sie auch schon in der Konfliktbildung eliminieren konnte. Im Beispiel also lediglich zwei. Es sind noch sechs Fälle im Resultat enthalten.

$$\text{(1) }\{[\text{ i, j, 4}] \rightarrow [\text{ i', j', 2}]\text{: j} = 2{+}3\text{i'} \wedge \text{i} = \text{i'} \wedge 0 \leq \text{i'} \leq 6 \wedge \mathbf{3{+}3i'} \leq \mathbf{j'} \wedge$$
$$\text{pred1(i',j')} = \text{T} \wedge \text{pred3(i,j)} = \text{T} \wedge \text{U}\} \cup$$
$$\text{(2) }\{[\text{ i, j, 4}] \rightarrow [\text{ i', j', 2}]\text{: j} = 2{+}3\text{i'} \wedge \text{i} = \text{i'} \wedge 0 \leq \text{i'} \leq 6 \wedge \mathbf{j'} \leq \mathbf{3i'{+}1} \wedge$$
$$\text{pred1(i',j')} = \text{T} \wedge \text{pred3(i,j)} = \text{T} \wedge \text{U}\} \cup$$
$$\text{(3) }\{[\text{ i, j, 4}] \rightarrow [\text{ i', j', 2}]\text{: j} = 2{+}3\text{i'} \wedge \text{i} = \text{i'} \wedge 0 \leq \text{i'} \leq 6 \wedge \mathbf{j'} = \mathbf{2{+}3i'} \wedge$$
$$\text{pred1(i',j')} = \text{T} \wedge \text{pred3(i,j)} = \text{T} \wedge \text{U}\} \cup$$
$$\text{(4) }\{[\text{ i, j, 4}] \rightarrow [\text{ i', j', 2}]\text{: j} = 2{+}3\text{i'} \wedge \mathbf{0,\ i{+}1} \leq \mathbf{i'} \leq \mathbf{6} \wedge \text{pred1(i',j')} = \text{T} \wedge$$
$$\text{pred3(i,j)} = \text{T} \wedge \text{U}\} \cup$$
$$\text{(5) }\{[\text{ i, j, 4}] \rightarrow [\text{ i', j', 2}]\text{: j} = 2{+}3\text{i'} \wedge \mathbf{0} \leq \mathbf{i'} \leq \mathbf{6,\ i\text{-}1} \wedge \text{pred1(i',j')} = \text{T} \wedge$$
$$\text{pred3(i,j)} = \text{T} \wedge \text{U}\} \cup$$
$$\text{(6) }\{[\text{ i, j, 4}] \rightarrow [\text{ i', j', 4}]\text{: }.\mathbf{i'} = \mathbf{i} \wedge \text{j} = 2{+}\text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{pred3(i',j')} =$$
$$\text{T} \wedge \text{pred3(i,j)} = \text{T}\}$$

Die Fälle 1 bis 5 sind im Vergleich zu dem ersten Ergebnis gleich geblieben. Lediglich die Unterscheidungen für j in den Fällen 6 bis 8 entfallen und damit bleibt der alte Fall 8 übrig.

5.2.3 Gleichsetzen von Index-Variablen (2)

Als Erweiterung des ersten Ansatzes zum Gleichsetzen von Indexvariablen ergibt sich die Möglichkeit, alle Indexvariablen, die *einen* der beiden Zugriffe *nicht* umgeben, gleichzusetzen, um unnötige Fallunterscheidungen zu vermeiden. Bei der Konfliktfindung bleiben die Ergebnisse damit korrekt, da ein bei einem Zugriff nicht-umgebender Index einen beliebigen Wert haben darf. Insbesondere kann er denselben Wert haben wie bei dem anderen Zugriff.

Algorithmus 5.2 zeigt das Vorgehen, das wiederum auf der Konfliktrelation aus Abschnitt 5.1 aufsetzt.

Algorithmus 5.2

```
erzeuge eine neue leere Relation mit denselben Indexvariablen
  wie in der Konfliktrelation
für (alle Indexvariablen in der Relation)
{
  wenn (die Variable ist in einem der Index-Vektoren
        der Kanten der beiden Zugriffe nicht
        enthalten)
  {
    füge der neuen Relation eine Gleichheitsrelation für die
    Variable durch UND-Verknüpfung hinzu
  }
}
schneide die ursprüngliche Konfliktrelation mit der hier
  berechneten und speichere das Ergebnis als neue
  Konfliktrelation
```

Im Beispielprogramm 5.1 können nun nicht mehr dieselben Zugriffe wie in den beiden vorhergehenden Abschnitten verwendet werden, da sich hier keine weitere Vereinfachung ergibt. Daher wird nun derselbe Zielzugriff $A[j-2]$ betrachtet, aber die andere Quelle, nämlich $A[3*i]$. Die ursprüngliche Konfliktrelation zu diesen Zugriffen ist sehr umfangreich:

$$
\begin{gathered}
[\text{ i, j, 4}] \rightarrow [\text{ i', j', 2}]\text{:} \\
(1)\ \text{j} = 2 + 3\text{i'} \wedge \mathbf{0 \leq i' \leq 6,\ i\text{-}1} \wedge \text{pred1(i',j')=T} \wedge \text{pred3(i, j)=T} \\
\vee \\
(2)\ \text{j} = 2 + 3\text{i'} \wedge \mathbf{0,\ i{+}1 \leq i' \leq 6} \wedge \text{pred1(i',j')=T} \wedge \text{pred3(i,j)=T} \\
\vee \\
(3)\ \text{j} = 2{+}3\text{i'} \wedge \text{i=i'} \wedge 0 \leq \text{i'} \leq 6 \wedge \mathbf{j' \leq 3i'{+}1} \wedge \text{pred1(i',j')=T} \wedge \\
\text{pred3(i,j)=T} \\
\vee \\
(4)\ \text{j} = 2{+}3\text{i'} \wedge \text{i=i'} \wedge 0 \leq \text{i'} \leq 6 \wedge \mathbf{3{+}3i' \leq j'} \wedge \text{pred1(i',j')=T} \wedge \\
\text{pred3(i,j)=T} \\
\vee \\
(5)\ \text{j'} = 2 + 3\text{i'} \wedge \mathbf{j = 2 + 3i'} \wedge \text{i} = \text{i'} \wedge 0 \leq \text{i'} \leq 6 \wedge \text{pred3(i,j)=T}
\end{gathered}
$$

Die ersten beiden Fälle bilden die Fallunterscheidungen für $i < i'$ und $i > i'$. Die restlichen setzen i und i' gleich und verwirklichen die Möglichkeiten für j, nämlich $j < j'$, $j > j'$ und $j = j'$. Es gilt wieder, daß für j der Ausdruck $2 + 3i'$ gesetzt wird.

Durch den ersten Vereinfachungsalgorithmus können keine Einsparungen erzielt werden, da es keine Variable gibt, die beide Statements *nicht* umgibt. Durch die Anwendung von Algorithmus 5.2 erhält man als Relation nach der Schleife über die Indexvariablen:

$$[\text{ i, j, x}] \rightarrow [\text{ i', j', x}]\text{: i'} = \text{i} \wedge \text{j'} = \text{j}$$

Nach dem Schneiden mit der Konfliktrelation ergibt sich:

$$
\begin{gathered}
[\text{ i, j, 4}] \rightarrow [\text{ i', j', 2}]\text{:} \\
\text{j'} = 2 + 3\text{i} \wedge \text{j} = 2 + 3\text{i} \wedge \text{i'} = \text{i} \wedge 0 \leq \text{i} \leq 6 \wedge \text{pred3(i, j)} = \text{T}
\end{gathered}
$$

Durch das Gleichsetzen wurden sämtliche Fälle eliminiert, bei denen nach größer oder kleiner unterschieden wurde. Somit bleibt nur Fall 5 übrig.

Das Endergebnis wird dadurch um vier Fallunterscheidungen im Vergleich zu der ersten Vereinfachung verringert und lautet nun:

$$
\begin{gathered}
\{[\text{ i, j, 4}] \rightarrow [\text{ i', 3i+2, 2}]\text{: j} = 2{+}3\text{i} \wedge \text{i'=i} \wedge 0 \leq \text{i} \leq 6 \wedge \text{pred1(i',j')} = \text{T} \wedge \\
\text{pred3(i, j)} = \text{T} \wedge \text{U}\} \\
\cup \\
\{[\text{ i, j, 4}] \rightarrow [\text{ i', j', 4}]\text{: i'} = \text{i} \wedge \text{j} = 2 + \text{j'} \wedge 0 \leq \text{j'} \leq 18 \wedge \text{pred3(i', j')} = \text{T} \wedge \\
\text{pred3(i, j)} = \text{T }\}
\end{gathered}
$$

Das entspricht dem erwarteten Ergebnis bis auf die Parameter der Prädikate.

Bereits für das einfache Beispielprogramm 4.2 ergeben sich bei der Anwendung des beschriebenen Algorithmus 5.2 von den optimierten Ergebnissen abweichende Relationen. Die Optimierung funktioniert durch das Gleichsetzen nicht mehr, da bei der Findung des Maximums von Quellzugriffen die Indexvariablen der Quellzugriffe optimal gesetzt werden sollen. Dies ist durch das Gleichsetzen von Indexvariablen des Quellzugriffs mit Indexvariablen des Zielzugriffs oft nicht mehr möglich. Des weiteren werden bei der Angabe der *noch nicht definierten Zugriffe* ebenfalls Fallunterscheidungen gemacht, die nicht erwünscht sind.

Das Ergebnis von Programm 4.2 sieht nach Anwendung von Algorithmus 5.2 folgendermaßen aus:

$$\begin{gathered}
\{[\text{ i, j, k, l, 12}] \rightarrow [\text{ i', j', k', i-j-3i', 1}]\text{:} \\
\text{j+l+3i'} = \text{i} \wedge 0 \leq \text{i'} < \text{i} \leq 20 \wedge 0 \leq \text{k} \leq 25 \wedge 0 \leq \text{j} \wedge \text{j+3i'} \leq \text{i}\} \cup \\
\{[\text{ 0, 0, k, 0, 12}] \rightarrow [\text{ 0, 0, k, 0, 1}]\text{: } 0 \leq \text{k} \leq 25\}
\end{gathered}$$

Ohne die Algorithmusanwendung und korrekt optimiert muß es heißen:

$$\begin{gathered}
\{[\text{ i, j, k, l, 12}] \rightarrow [\text{ i', i-l-3i', 25, l', 1}]\text{:} \\
\text{i'+2} \leq \text{i} \leq 20 \wedge \text{i} \leq \text{2+l+3i'} \wedge 0 \leq \text{i'} \wedge \text{l+3i'} \leq \text{i} \wedge 0 \leq \text{l}\} \cup \\
\{[\text{ 1, j, k, l, 12}] \rightarrow [\text{ 0, -l+1, 25, l', 1}]\text{: } 0 \leq \text{l} \leq 1\} \cup \\
\{[\text{ 0, j, k, 0, 12}] \rightarrow [\text{ 0, 0, 25, l', 1}]\}
\end{gathered}$$

Bis auf die Optimierung ist das Ergebnis korrekt. Die undefinierten Zielzugriffe dagegen enthalten zum Beispiel folgende Ausgaben:

$$\begin{gathered}
\{[\text{ i, j, k, l, 12}]\text{: } 0 \leq \text{i} \leq 20 \wedge 0 \leq \text{l} \leq 22 \wedge 26 \leq \text{k}\} \cup \\
\{[\text{ i, j, k, l, 12}]\text{: } 0 \leq \text{i} \leq 20 \wedge 0 \leq \text{l} \leq 22 \wedge \text{k} \leq \text{-1}\} \cup \ldots.
\end{gathered}$$

Diese Fallunterscheidungen für k sind nicht notwendig und als Ergebnis nicht erwünscht. Durch Weglassen der unnötigen Bedingungen für Indexvariablen werden die undefinierten Zielzugriffe wieder auf diejenigen reduziert, die nach einer korrekten Optimierung resultieren.

Somit ist dieser Ansatz nicht geeignet, optimierte Ergebnisse zu erzielen.

5.2.4 Fazit

Die Ausführungen haben gezeigt, daß jeder der Lösungsansätze für bestimmte Arten von Eingabeprogrammen keine korrekten Ergebnisse liefert. Eine Entscheidung für eine der Lösungen muß für jedes Eingabeprogramm getrennt getroffen werden.

Die einzig wirkliche Lösung des Problems besteht darin, die Prädikate mit ihren tatsächlichen Parametern zu versehen, wie es eingangs bereits beschrieben wurde.

Kapitel 6

Bedienung und Ablauf des Analyseprogramms Cffada

6.1 Aufrufparameter

Das Analyseprogramm Cffada kann mit verschiedenen Aufrufparametern gestartet werden. Informationen zu Art und Anzahl erhält der Anwender, wenn er die Option -help benutzt. Der Zweck der Parameter wird im folgenden näher erläutert.

optimize_anti_dependency: Durch Setzen dieses Parameters kann die Optimierung auch bei der Berechnung von Anti-Abhängigkeiten erzwungen werden (vgl. Abschnitt 4.5).

true, anti, output, all: Diese Parameter bestimmen die Art der zu berechnenden Abhängigkeiten. Als Default ist TRUE eingestellt, sobald aber einer der Parameter angegeben wird, gilt diese Einstellung nicht mehr. Es können auch beliebig viele dieser Parameter gleichzeitig angegeben werden.

pred_without_params: Dieser Schalter gibt an, daß die Prädikate ohne Parameter in die Berechnung eingehen. Damit können Programmläufe wie in Abschnitt 5.2.1 beschrieben erzielt werden.

simplification 1/2: Durch diesen Parameter wird einer der in Abschnitt 5.2.2 (bei Wert = 1) und 5.2.3 (Wert = 2) beschriebenen Algorithmen über den jeweils angegebenen Wert ausgewählt.

Des weiteren gibt es noch einige Parameter, die beim Debuggen oder Nachvollziehen von Programmabläufen helfen sollen. Im folgenden werden nur diejenigen erwähnt, die im Rahmen dieser Arbeit neu hinzugekommen sind.

FUNCOUT: Dieser Parameter ist ein Schalter, mit dem man aktivieren kann, daß bei jedem Ein- und Austritt aus Methoden von Cffada eine Ausgabe erfolgt.

simplify_debug: Mit diesem Parameter und einem Wert zwischen 1 und 9 dazu, kann man verschiedene Ausgaben der Vereinfachungsalgorithmen erzwingen, die auch für die Beispiele in dieser Arbeit als Grundlage dienten.

optimization_debug: Durch Setzen eines hohen Wertes hinter diesem Parameter werden Informationen während der Optimierung der Ergebnisse ausgegeben.

6.2 Datenstruktur

Im Wesentlichen ist die Datenstruktur des Analyseprogramms eine Erweiterung des eingelesenen Kontrollflußgraphen. Da an Knoten keine Operationen ausgeführt werden, werden hier lediglich Ergebnisse gespeichert. Zu diesem Zweck hat der Knoten ein Attribut, dem eine Relation zugewiesen werden kann.

Die zentrale Klasse des Analyseprogramms heißt Cffada. Ihre Datenstruktur sieht folgendermaßen aus:

Datenstruktur von Cffada:

cfg: Hinter diesem Attribut verbirgt sich der Kontrollflußgraph zu dem betrachteten Eingabeprogramm. Seine Datenstruktur wird im Anschluß erklärt.

accessVector: In diesem Vektor werden die Schleifenindizes gehalten, die den untersuchten Zielzugriff umgeben.

actualLevelVector: Dieses Attribut enthält eine Kopie des Schleifenindex-Vektors von der Kante, an der die aktuelle Ebene belegt wurde.

lastEdgeVector: Dieses Attribut enthält für die Berechnung den Vektor der zuletzt berechneten Kante.

actorder: Dieses Attribut ist vom Typ Relation und enthält nach der Berechnung die aktuelle Ordnung.

toBeVisited: Dies ist eine Liste von Knoten, die noch bearbeitet werden müssen.

predicates: Ein Vektor, der die im Eingabeprogramm enthaltenen Prädikate speichert.

conflict, nodestate, edgestate: Diese Attribute stellen jeweils ein Array von Relationen dar.

maxdatadim, maxdim, actualLevel,dep_kind: Dies sind Attribute, die während der Berechnung ihrem Namen entsprechende Werte tragen.

Datenstruktur von Graph:

nodes: Vektor von Knoten

edges: Vektor von Kanten

Datenstruktur von Knoten:

edges: Vektor der ausgehenden Kanten

inEdges: Vektor der eingehenden Kanten

nodeName: Name des Knoten

number: Nummer des Knoten

predicate: Zeiger auf das eventuell vorhandene Prädikat

pred_type: Typ des Pfads des Prädikats (z.B. else, then)

Datenstruktur der Kanten:

name: Bezeichnung der Kante

number: Nummer der Kante

head: Knoten an der Spitze der Kante

tail: Knoten am Ende der Kante

relation: Die zur Kante gehörende Operation, bzw. eine Bedingung, ausgedrückt durch ein Prädikat.

reference: Zeiger auf eine *EdgeReference* (siehe unten)

traversed: Zähler, der angibt, wie oft die Kante während der Berechnung durchlaufen wurde.

loopDepth: Vektor, der die umgebenden Schleifenindizes beinhaltet.

Datenstruktur der Referenz:

writes: Vektor der Schreibzugriffe (Datentyp *Access*)
reads: Vektor der Lesezugriffe (Datentyp Access)

Datenstruktur der Zugriffe (Access):

relation: Die zum Zugriff gehörende Relation
data: wird nicht verwendet
name: Name des betroffenen Arrays
arrayid: Id des betroffenen Arrays

6.3 Beispielablauf des Cffada

Zum Abschluß dieser Arbeit soll an einem abrundenden Beispiel ein kompletter Ablauf des erstellten Analyseprogramms gezeigt werden. Die vorgestellten Algorithmen werden in der Anwendung gezeigt und dadurch zusammengefaßt. Damit dieser Abschnitt nicht zu umfangreich wird, werden nur die Schlüsselstellen der Abhängigkeitsanalyse aufgezeigt. Das Eingabeprogramm ist Programm 6.1.

Den für Cffada zu Grunde liegenden Kontrollflußgraphen zeigt Abbildung 6.1. In dieser Abbildung sind die Nummern der Kanten in **Fettdruck** gezeigt. Die umgebenden Schleifenindizes sind als Nummern in einem Vektor an der Kante zusammengefaßt und werden in geschweiften Klammern dargestellt.

Nun startet eine Abhängigkeitsanalyse für True-Abhängigkeiten. Zusätzlich wird der Parameter *pred_without_params* gesetzt, damit die Ergebnisse nicht durch unnötige Fallunterscheidungen aufgebläht werden. In diesem Beispiel muß nicht zwischen Ziel- und Quellzugriffs-Parametern der Prädikate unterschieden werden, da nur ein Schreibzugriff $A[k, k]$ von dem Prädikat P abhängig ist. Deshalb liefert der Lösungsansatz aus Abschnitt 5.2.1 korrekte Ergebnisse.

Der zu betrachtende Zielzugriff ist der Lesezugriff auf $A[k-1, k-1]$ an der Kante 11. Begonnen wird mit der Bestimmung der Konfliktrelationen.

Programm 6.1

```
DO i=0,10
  A[0,0] = i
END DO
DO j=1,20
  DO k=1,20
    if (P)
      A[k,k] = 5
    A[1,k] = A[k-1,k-1]
  END DO
END DO
```

Kante	Konfliktrelation
1	[i, j, 1, 11] → [i', j', k', 1]: $0 \leq$ i' $\leq 10 \wedge 1 \leq$ j ≤ 20
7	[i, j, k, 11] → [i', j', k-1, 7]: $1 \leq$ j' $\leq 20 \wedge 1 \leq$ k ≤ 19 $\wedge\ 1 \leq$ j $\leq 20 \wedge$ pred=T
11	[i, j, 2, 11] → [i', j', 1, 11]: $1 \leq$ j' $\leq 20 \wedge 1 \leq$ j ≤ 20

Tabelle 6.1: Konfliktrelationen an den Kanten

Potentielle Kandidaten für Quellzugriffe befinden sich an den Kanten 1, 7 und 11. Die Konfliktrelationen lauten wie in Tabelle 6.1.

Der Durchlauf des Graphen beginnt am Knoten 13. An der Kante 10 werden keine Zugriffe gefunden, auch an der Kante 9 nicht. Im weiteren werden nur noch solche Kanten erwähnt, an denen sich Änderungen der bisherigen Ergebnisrelation ergeben können. Die Kante 7 ist die erste Konfliktkante, die zu berechnen ist. Da die aktuelle Ordnung zu diesem Zeitpunkt $k = k'$ ist, ergibt sich eine leere Schnittmenge, das Ergebnis ändert sich nicht.

An der Kante 16 ergibt sich die aktuelle Ordnung:

$$[\ i, j, k] \rightarrow [\ i', j', k']:\ j' = j \wedge k' < k$$

An der Kante 11 erhält man durch Schneiden dieser Ordnung mit der Konfliktrelation der Kante:

$$[\ i, j, 2, 11] \rightarrow [\ i', j, 1, 11]:\ 1 \leq j \leq 20$$

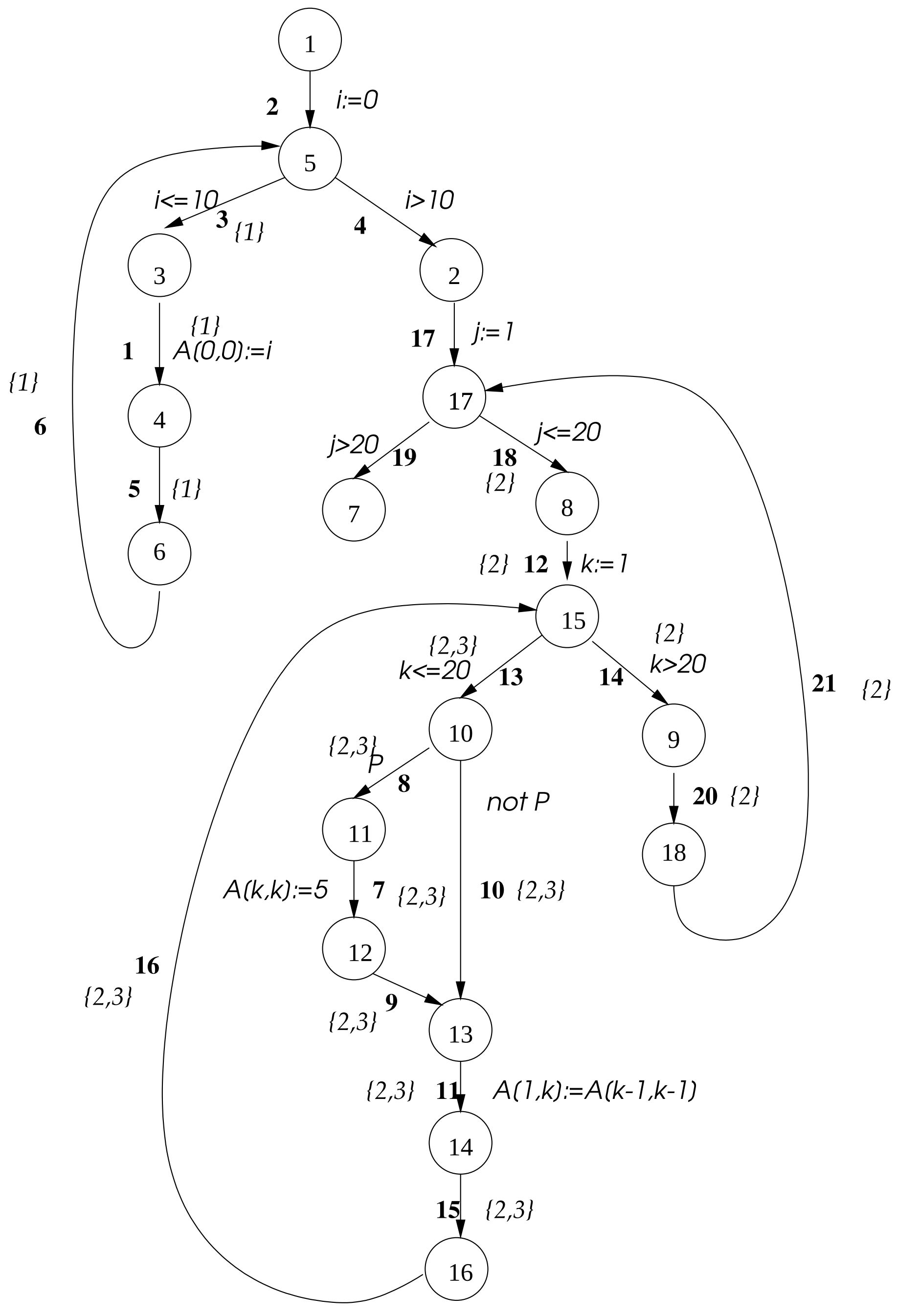

Abbildung 6.1: Kontrollflußgraph des Beispielprogramms

Auch an der Kante 7 wird die Ergebnisrelation um die Schnittmenge mit der aktuellen Ordnung erweitert:

$$[\text{ i, j, k, 11}] \rightarrow [\text{ i', j, k-1, 7}]\text{: } 1 \leq \text{j} \leq 20 \wedge 3 \leq \text{k} \leq 20 \wedge \text{pred=T}$$

Die weitere Graphabarbeitung führt über die Rückwärtskante der j-Schleife wieder in die k-Schleife. Die aktuelle Ordnung ist nun:

$$[\text{ i, j, k}] \rightarrow [\text{ i', j', k'}]\text{: j'} < \text{j}$$

Weder an der Kante 11 noch an der Kante 7 ergeben sich neue Quellzugriffe, da durch die Optimierung sichergestellt wird, daß nur solche Ergebnisse hinzugenommen werden, die noch nicht darin enthalten sind. In einer früheren Iteration der j-Schleife ist es aber nicht der Fall, daß Schreibzugriffe auf Elemente erfolgen, die *vor* dem Lesezugriff nicht mehr überschrieben werden. Die Ergebnisrelation bleibt also unverändert. Beim Verlassen der j-Schleife wird die aktuelle Ordnung auf *neutral* gesetzt, sie ist TRUE.

An der Kante 1 ist die Schnittmenge mit der aktuellen Ordnung nicht leer. Durch die Optimierung des lokalen Ergebnisses ergibt sich, daß lediglich der letzte Zugriff der Schleife relevant ist. Die optimierte Relation der Kante wird übernommen:

$$[\text{ i, j, 1, 11}] \rightarrow [\text{ 10, j', k', 1}]\text{: } 1 \leq \text{j} \leq 20$$

Damit ist die Berechnung dieses Beispiels beendet. Das Endresultat lautet:

$$\begin{gathered}\{[\text{ i, j, 1, 11}] \rightarrow [\text{ 10, j', k', 1}]\text{: } 1 \leq \text{j} \leq 20\} \cup \\ \{[\text{ i, j, k, 11}] \rightarrow [\text{ i', j, k-1, 7}]\text{: } 1 \leq \text{j} \leq 20 \wedge 3 \leq \text{k} \leq 20 \wedge \text{pred=T}\} \cup \\ \{[\text{ i, j, 2, 11}] \rightarrow [\text{ i', j, 1, 11}]\text{: } 1 \leq \text{j} \leq 20\}\end{gathered}$$

Dadurch ergibt sich, daß folgende noch undefinierten Lesezugriffe existieren:

$$[\text{ i, j, k, 11}]\text{: } 1 \leq \text{j} \leq 20 \wedge 3 \leq \text{k} \leq 20 \wedge \text{pred=F}$$

Der Abhängigkeitsgraph zu diesem Beispiel ist in Abbildung 6.2 aufgezeichnet.

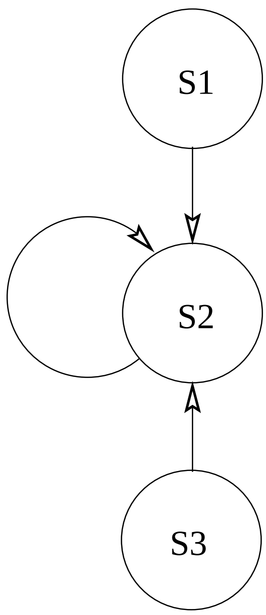

Abbildung 6.2: Abhängigkeitsgraph des Beispielprogramms

Kapitel 7

Zusammenfassung mit Ausblick

Zu Beginn dieser Arbeit waren Datenabhängigkeitsanalysen im LooPo-Umfeld entweder auf perfekt verschachtelten Eingabeprogrammen mit bedingten Verzweigungen möglich, oder auf solchen ohne bedingte Verzweigungen, die nicht perfekt verschachtelt sind. Für jede der beiden Eingabeprogrammausprägungen existiert ein eigener Datenabhängigkeitsanalyse-Algorithmus. Die im Rahmen dieser Arbeit entstandene Implementierung Cffada kann beide Programmarten behandeln und erweitert so das Potential von LooPo.

Bei der Analyse wird der Kontrollflußgraph eines Eingabeprogramms abgearbeitet. Den Kanten des Graphen sind die Operationen dieses Eingabeprogramms zugeordnet. Die Knoten repräsentieren Eingabeprogrammzustände. Die Analyse benutzt Relationen, um die verschiedenen Array-Zugriffe miteinander in Beziehung zu setzen. Die Bedingungen von bedingten Verzweigungen werden durch Prädikate ersetzt, die in die Relationen mit aufgenommen werden. Dadurch ist die Voraussetzung dafür geschaffen, Eingabeprogramme mit bedingten Verzweigungen zu analysieren. Die resultierenden Abhängigkeiten enthalten für die verschiedenen Zugriffe auch jeweils die notwendige Belegung der Bedingung (also TRUE oder FALSE).

Damit nicht perfekt verschachtelte Schleifen korrekt bearbeitet werden können, wurde ein Verfahren entwickelt, das die lexikografische Ordnung durch Identifizierung der umgebenden Schleifenindizes und Berücksichtigung der bereits durchlaufenen Schleifen berechnet. Diese lexikografische Ordnung ist zusammen mit der textuellen Ordnung auch bei der Optimierung der Ergebnisse sehr wichtig. Die Ergebnisse des erstellten Analyseprogramms Cffada sind für viele der erlaubten Eingabeprogramme (siehe Abschnitt 3) *optimiert*. Optimiert bedeutet hier, daß in den abhängigen Quellzugriffen im Endergebnis keine Zugriffe mehr enthalten sind, die vor dem betrachteten Zielzugriff nochmal überschrieben werden.

Cffada liefert jedoch nicht für alle erlaubten Eingabeprogramme optimierte Ergebnisse. Für Eingabeprogramme mit bedingten Verzweigungen müssen Einschränkungen gemacht werden, verursacht durch das verwendete Hilfswerkzeug Omega. Es gibt verschiedene Lösungsansätze für dieses Problem, die in Kapitel 5 aufgezeigt wurden. Verursacht wird das Problem durch die Prädikate, die bei der Verwendung von Omega mit Parametern versehen werden können. Da diese Parameter nicht konkret angegeben werden können, resultieren teils falsche, teils nur nicht optimierte Ergebnisse. Die beschriebenen Lösungsansätze sind das Weglassen von Parametern und das Gleichsetzen von nicht umgebenden Indexvariablen der verschiedenen Zugriffe. Alle Ansätze bergen auch Nachteile bzw. bringen nicht die gewünschten Ergebnisse für manche der möglichen Eingabeprogramme.

Die Behandlung unstrukturierter Eingabeprogramme ist durch das erstellte Analyseprogramm nicht möglich. Es fehlen noch einige Vorarbeiten (z.B. Identifizierung der inkrementierten Indexvariablen, siehe Abschnitt 3.3), die in LooPo ergänzt werden müßten. In Abschnitt 4.4 wurden die Probleme bei der Bearbeitung solcher Eingabeprogramme zusammen mit einem Lösungsvorschlag erläutert. Die Bestimmung einer aktuellen Ordnung ist in unstrukturierten Eingabeprogrammen wesentlich komplizierter als in strukturierten, da Rückwärtskanten *beliebig* durchlaufen werden können. Eine genaue Betrachtung möglicher Werte von Indexvariablen und Prädikatbelegungen an Verzweigungen ist hier vonnöten.

Der Analyse-Algorithmus sollte an dieser Stelle weiterentwickelt werden, um die parallele Abarbeitung auch für unstrukturierte Programme im LooPo-Umfeld zu ermöglichen. Auch die Einbindung von Eingabeprogrammen mit while-Schleifen kann vorgenommen werden. Allerdings muß die Verwendung von Omega überdacht werden, da die Probleme im Zusammenhang mit den Prädikaten immer wieder auftreten werden und die Aussagekraft der Ergebnisse erheblich schmälern. Durch zusätzliche Informationen über die Belegung der Prädikate, die während der ganzen Berechnung mit berücksichtigt werden, könnte man dieses Problem vermutlich lösen. Eine andere Möglichkeit ist, die Relationen-Mathematik selbst zu implementieren und dabei die Belegung der Prädikate anders als in Omega mit angebbaren Parametern zu versehen (siehe Kapitel 5).

Literaturverzeichnis

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Norwell, Mass.: Kluwer Academic Publishers, 1993.

[3] J.-F. Collard and M. Griebl. A precise fixpoint reaching definition analysis for arrays. In J. Ferrante, editor, *Languages and Compilers for Parallel Computing (LCPC'99)*, LNCS. Springer-Verlag, 1999. to appear.

[4] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb. 1991.

[5] M. Griebl and C. Lengauer. The loop parallelizer LooPo. In M. Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.

[6] H. Keimer. Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze. Diplomarbeit, 1997.

[7] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega calculator and library. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, 1996.

[8] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The omega library. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, 1996.

[9] J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? In K. Koskimies, editor, *Compiler Construction (CC'98)*, pages 63–79, Lisbon, 1998. Springer LNCS 1383.

[10] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

Eidesstattliche Erklärung

Hiermit erkläre ich, Babette Eckart, eidesstattlich, daß ich die Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 19. September 2000