DISCLAIMER

Die zur Vorlesung erscheinenden Unterlagen sind nicht als vollständiges Skript gedacht. Sie alleine reichen auch nicht für eine Prüfungsvorbereitung. Gründe:

- Sie sind absolut informell und daher mißverständlich.
- Sie sind unvollständig.
- Sie enthalten keinerlei Beispiele.
- Sie sind ohne Vorlesung vermutlich unverständlich.

Was sollen diese Unterlagen dann überhaupt?

Sie sollen dazu dienen, den Inhalt zu strukturieren und das Aufschreiben von oft nur gesprochenen Erklärungen zum Teil unnötig werden zu lassen, damit man während der Vorlesung leichter mitdenken kann. Die starke, numerierte Gliederung der Unterlagen ermöglicht (hoffentlich) die einfache Ergänzung der Unterlagen durch eine eigene Mitschrift, die dann etwa die Formalisierungen und die Beispiele von der Tafel beinhalten könnte.

Für die Prüfungsvorbereitung kann sie als Leitfaden dienen, der zwar **voraussichtlich** alle wesentlichen Themengebiete der Vorlesung beinhaltet, sie jedoch meist nicht in der nötigen Tiefe behandelt.

Wenn die Vorbereitungszeit zu knapp wird, dann behalte ich mir natürlich vor, die Unterlagen für einzelne Vorlesungen nicht auszuarbeiten (wobei ich hoffe, daß das nicht der Fall sein wird).

Literatur

Signaturen: 80/ST 265 [B215 | W855 | Z71] und 80/SS 4800-715.

1 Konventionelle Abhängigkeitsanalyse

1.0.1 Prinzipielles Vorgehen am Beispiel

- synthetisierte Atribute für Zuweisungen im Parsebaum
  - gen[S]: die Definitionen eines Statements S
  - kill[S]: solche Definitionen, die von Statement S überschrieben werden
- Verallgemeinerung auf zusammengesetzte Anweisungen und auf Mengen von Anweisungen, insbesondere auf „basic blocks“
- Berechnung entlang des Syntaxbaums (bottom-up) durch *Datenflussgleichungen* – oder allgemein durch Fixpunktiteration.

zusammengesetzte Anweisungen Verallgemeinerung von gen[S] und kill[S] auf zusammengesetzte Anwesungen wie Sequenz, Konditional und Schleife.
Daher ist eine Abschätzung notwendig:

- gen legt fest: es könnte generiert worden sein.
- kill bedeutet, es ist garantiert gelöscht worden.

Die Berechnung erfolgt entlang des Parsebaums (bootom-up) durch *Datenflussgleichungen.*

1.0.2 Datenflussgleichungen

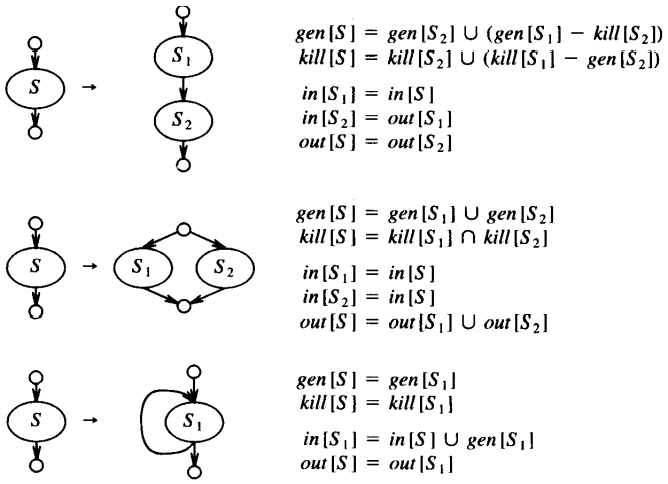


Abbildung 1: Datenflussgleichungen für zusammengesetzte Anweisungen

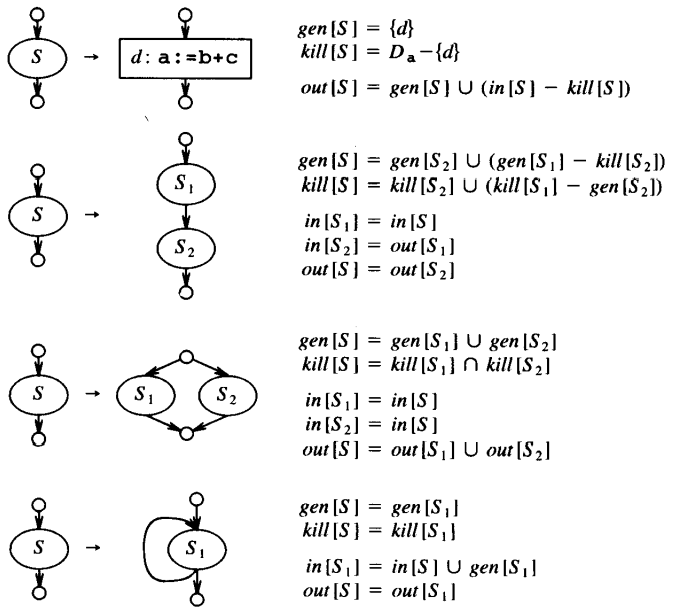


Abbildung 2: Datenflussgleichungen für „erreichende Definitionen“

1.0.3 Beispielprogramm

```
/* d1 */    i := m-1;
/* d2 */    j := n;
/* d3 */    a := u1;
            do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
                if e1 then
/* d6 */            a := u2
                else
/* d7 */            i := u3
            while e2
```

Abbildung 3: Beispielprogramm

1.0.4 Resultat

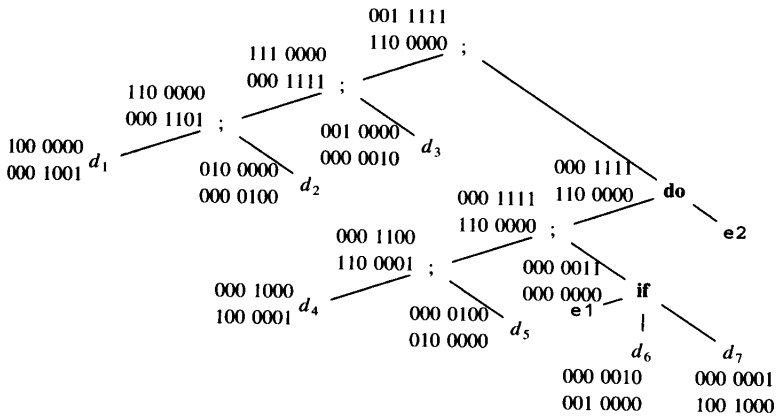


Abbildung 4: Resultat

1.0.5 Nachteile dieses Verfahrens

- Bitvektor-Darstellung ungeeignet für Arrays
- Konflikte, z. B. für kill, nur ungenau:
  - auf Variablennamen basiert (für Array ungeeignet)
  - keine Berücksichtigung von eventuell bekannten Schleifengrenzen
  - keine Berücksichtigung der sequentiellen Ausführungsordnung
- für jedes Analyseziel ein neues Datenflussgleichungssystem von Grund auf neu berechnen
- im allgemeinen Fall Fixpunktiteration nötig

2 Programm-Modell

1. Datentypen:
   (a) Als Datentypen erlauben wir nur (mehrdimensionale) Arrays mit beliebigen Einträgen. Beachte: Skalare sind 0-dimensionale Arrays.
   (b) Lediglich Indizes von Schleifen werden nicht als 0-dimensionale Arrays, sondern wirklich als Integers betrachtet.
   (c) Darüberhinaus gibt es symbolische Konstanten vom Typ Integer, die oft auch als Strukturparameter bezeichnet werden.
2. Kontrollstrukturen:
   (a) Das Basiselement ist die Zuweisung an ein Arrayelement.
   (b) Die einzige Kontrollstruktur ist die Schleife (beliebig verschachtelte Anwendung willkommen).
   (c) Als Schleifenunter- bzw. -obergrenzen sind im originalen Programm-Modell nur Maxima bzw. Minima affiner Ausdrücke in umgebenden Schleifenindizes und in Strukturparametern erlaubt; das wird auch der Schwerpunkt der Vorlesung sein, wenngleich diese Restriktionen mittlerweile theoretisch aufgelöst wurden.
3. Iterationen:
   (a) Durch die Schleifen entstehen verschiedene Instanzen von Zuweisungen (Operations).
   (b) Identifiziert werden Operations durch Angabe des Statements und des Iterationsvektors (Vektor, der sich aus den Werten der Indizes der umgebenden Schleifenindizes zusammensetzt).
   (c) Der Raum aller Operations heißt Index- oder Iterationsraum aller Statements. (Manchmal werden wir die Indexräume von verschiedenen Statements auch übereinanderlegen und somit nur einen Indexraum für mehrere Statements erhalten, falls die Statements dieselben umgebenden Schleifen besitzen.)
4. Ordnungen:
   (a) (partielle) Ordnung: reflexiv, transitiv, antisymmetrisch
   (b) minimale, maximale Elemente: fehlende Existenz von kleineren, größeren Elementen
   (c) größtes, kleinstes Element: alle anderen sind kleiner, größer
   (d) totale Ordnung: beliebige zwei Elemente vergleichbar
   (e) Kette, Antikette: Menge von nur (un-)vergleichbaren Elementen, genauer: A Antikette $\Leftrightarrow (\forall x, y : x, y \in A \wedge x \neq y : \neg((x \leq y) \vee (y \leq x)))$
   (f) maximale (Anti-) Kette: es gibt keine echte Obermenge, die eine (Anti-) Kette ist.
   (g) Theorem 1.1 in [**?**].
   (h) lexikographische, komponentenweise Ordnung: sollten bekannt sein.
5. Ausführungsreihenfolge:

(a) Für sequentielle Schleifen gilt: In einem perfekt verschachtelten Schleifensatz (die Zuweisungen stehen alle in demselben Rumpf der innersten Schleife) werden verschiedene Instanzen des Schleifenrumpfes gemäß der lexikographischen Ordnung der Iterationsvektoren abgearbeitet.

(b) Operation a ist gemäß der sequentiellen Ausführungsordnung kleiner als (also vor) Operation b, i.Z. $a \prec b$ gdw. ihre Iterationsvektoren, eingeschränkt auf gemeinsam umgebende Schleifen, lexikographisch kleiner sind, oder diese Vektoren gleich sind und a im Programmtext vor b steht.

6. Abhängigkeiten:

   (a) geben an, für welche Operations die Ausführungsreihenfolge zwingend vorgeschrieben ist. Formale Definition: später.

7. Graphen

   (a) (un)gerichtete Graphen: Knoten, Bögen/Kanten (mit/ohne Richtung) (Anm: wir verwenden "Kanten" auch im gerichteten)

   (b) Wege, Pfade: klar; Pfad beachtet die Richtung

   (c) Zyklen, Schleifen: Weg, Pfad mit Anfang = Ende

   (d) schwach zusammenhängend: ungeachtet der Richtung

   (e) stark zusammenhängend: mit Beachtung der Richtung von jedem zu jedem

   (f) n-fach zusammenhängend: nach dem Löschen von n–1 beliebigen Kanten immer noch zusammenhängend

   (g) schwache, starke Zusammenhangskomponenten: Teilgraphen, die schwach, stark zusammenhängend sind

   (h) azyklische Kondensation: pro starke Zusammenhangskomponente ein Knoten; die "schwachen Kanten" übernehmen

8. Abhängigkeitsgraphen:

   (a) Iterationsabhängigkeitsgraph: je Operation (oder für jeweils übereinandergelegte Operationen) ein Knoten mit Abhängigkeiten als Kanten; sehr genau, aber u.U. unbeschränkt.

   (b) Statementabhängigkeitsgraph: nur je Statement ein Knoten; kleiner, aber ungenauer.

9. Parallelisierung

   (a) wirklich notwendige Ordnung von den Abhängigkeiten vorgeschrieben

   (b) paralleles Programm (Herleitung später)

2 Polyedermodell

2.1 Das ursprüngliche Modell

1. Restriktionen:
   (a) perfekt verschachtelt
   (b) Schleifengrenzen: affine Ausdrücke in Strukturparametern und umgebenden Indizes und Maxima bzw. Minima davon (für Unter- bzw. Obergrenzen)
   (c) Arrayindizes affin in den Schleifenindizes und Strukturparametern
   (d) uniforme Abhängigkeiten
   (e) ein Schedule und eine Allokation für den kompletten Rumpf; Schedule zunächst eindimensional
2. Modell:
   (a) n-dimensionaler Schleifensatz im n-dimensionalen Raum (je Schleife eine Dimension)
   (b) jeder affine Ausdruck in den Schleifengrenzen definiert einen Halbraum
   (c) Polyeder: der Durchschnitt endlich vieler Halbräume
   (d) Polytop: beschränktes Polyeder
   (e) (Quell-)Indexraum ist ein Polytop
   (f) Abhängigkeiten durch Pfeile im Indexraum repräsentiert
   (g) Quellpolytop beinhaltet alle nötigen Informationen
3. Raum-Zeit-Abbildung:
   (a) Schedule: Funktion, die jeder Operation einen (logischen) Ausführungszeitpunkt zuordnet, und dabei die durch die Abhängigkeiten vorgegebenen Bedingungen berücksichtigt
   (b) für das Modell: affine Funktion
   (c) graphische Bestimmung s. Beispiel der Vorlesung
   (d) mathematische Bestimmung: lineare Programmierung. Aufgabe: minimiere die affine Funktion unter der Nebenbedingung, daß ihr Wert für einen abhängigen Punkt größer ist als ihr Wert an dem Punkt, von dem er abhängt.
   (e) Allokation: Funktion, die jeder Operation einen (virtuellen) Prozessor zur Ausführung zuordnet
   (f) für das Modell: affine Funktion, die zum Schedule linear unabhängig ist (wegen der realen Maschinen und wegen des Modells nötig!)
   (g) Raum-Zeit-Abbildung: die (mehrdimensionale) affine Abbildung, repräsentiert durch die Transformationsmatrix, die sich aus Schedule und Allokation zusammensetzt, und so jeder Operation einen Raum-Zeit-Punkt der Ausführung zuweist. Zusätzliche Restriktion an die Allokation: die Raum-Zeit-Matrix muß unimodular sein (Determinante betragsmäßig 1)
4. Transformation des Modells:
   (a) Problem: jede einzelne Dimension kann teilweise in Raum und teilweise in der Zeit sein

   (b) Ziel: Separation – jede Dimension entweder im Raum oder in der Zeit
   (c) Weg: Basiswechsel = Anwendung der Raum-Zeit-Abbildung auf den Quell-Indexraum
   (d) Resultat: "verzerrtes" Polytop: Zielpolytop

5. Zurück zum Programm:
   (a) Aufgabe: "scanning" der Punkte des Zielpolytops
   (b) parallele Schleifen für die Raumdimensionen und sequentielle Schleifen für die Zeitdimensionen
   (c) Weg: im Quell-Ungleichungssystem die Indizes durch "$T^{-1}*$Zielindizes" ersetzen und auflösen
   (d) abhängig von der gewünschten Reihenfolge der Zieldimensionen: (a)synchrones Programm
   (e) Quellindizes durch "$T^{-1}*$Zielindizes" ersetzen

2.2 Das erweiterte Modell

Erweiterungen:

1. affine Abhängigkeiten
2. stückweise affine Schedules
3. per-Statement-Schedule
4. nicht-perfekte Verschachtelung
5. nicht-unimodulare Raum-Zeit-Abbildungen
6. beliebige (z.B. nicht-invertierbare) Raum-Zeit-Abbildungen
7. beliebige Schleifentypen
8. allgemeine Array-Indizes

3 Mathematische Grundlagen

1. unimodulare Abbildungen
   (a) ganzzahlig; Determinante=±1; Inverses ganzzahlig
   (b) Bilder von Polytopen unter unimodularen Abbildungen
   (c) elementare Zeilentransformationen sind unimodulare Abbildungen:
      i. Reversal: Multiplikation einer Zeile mit -1 (!)
      ii. Interchange: Vertauschen zweier Zeilen
      iii. Skewing: ein Vielfaches einer Zeile zu einer anderen addieren
   (d) Unimodularität abgeschlossen unter Multiplikation
   (e) Wesentliches Verfahren: Zeilen-Stufen-Reduktion (Echelon Reduktion) einer Matrix A durch eine unimodulare Transformation U: $U * A = S$
   (f) Möglichkeit: Diagonalisierung durch anschließende elementare Spaltenoperationen V: $S * V = D$
2. ganzzahlig-lineare Gleichungssysteme
   (a) "normale" Gauß-Elimination liefert alle rationalen Lösungen
   (b) Einsetzen der frei wählbaren Variablen führt i.A. zu Brüchen in den nicht frei wählbaren Variablen
   (c) Nachbearbeitung der freien Variablen ist möglich (um ganzzahlige Lösungen zu erhalten). Frage: gibt es eine direktere Berechnung?
   (d) Bekannt: eine ganzzahlig-lineare Gleichung ist lösbar gdw. der ggT g der Koeffizienten a die rechte Seite c teilt
   (e) ggT(a) steht in der Zeilen-Stufen-Form des Spaltenvektors (a) oben; erste Zeile von U liefert die Multiplikatoren für a
   (f) Idee: ggT-Restriktion als erstes einsetzen um die Brüche zu vermeiden, die bei der Rückwärtssubstitution à la Gauß bei der letzten Substitution entstehen. Löungsmöglichkeiten: Spalten-Stufen-Form (ungewohnt) oder "transponierte Darstellung". Daher ab sofort:
   (g) Variablenvektor x von links multiplizieren (damit Matrizen transponiert) und Zeilen-Stufen-Transformation. Also: $x*A = c$. Dann gilt für eine einzelne Gleichung $x * a = c$ nach unimodularer Zeilen-Stufen-Transformation des Spaltenvektors a:
   (h) Menge aller Lösungen: $(x_1, \cdots, x_m) = (c/g, t_2, \cdots, t_m) * U$
   (i) Verallgemeinerung: $x * A = c$ lösbar gdw. $\exists t : t * S = c$. Menge aller Lösungen: $x = t * U$.

Beispiel
Ziel: Eine ganzzahlige Lösung ist gewünscht
Erinnerung: $U * A = S$ und $x = t * U$

Gleichungssystem:

$$\begin{aligned} 4x_1 + 6x_2 &= 8 \\ 15x_1 + 21x_2 + 6x_3 &= 9 \end{aligned}$$

Lösung nach Algorithmus:

$$\left(\begin{array}{cc|ccc} 4 & 15 & 1 & 0 & 0 \\ 6 & 21 & 0 & 1 & 0 \\ 0 & 6 & 0 & 0 & 1 \end{array}\right) \rightsquigarrow \left(\begin{array}{cc|ccc} 2 & 6 & -1 & 1 & 0 \\ 0 & 3 & 3 & -2 & 0 \\ 0 & 0 & -6 & 4 & 1 \end{array}\right)$$

Somit:

$$(t_1, t_2, t_3) * \begin{pmatrix} 2 & 6 \\ 0 & 3 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 8 & 9 \end{pmatrix}$$

$\Rightarrow t_1 = 4$, $t_2 = \frac{(9-6*4)}{3} = -5$, $t_3 \in \mathbb{Z}$

$$\left(\begin{array}{c|cc} 4 & 1 & 0 \\ 6 & 0 & 1 \end{array}\right) \rightsquigarrow \left(\begin{array}{c|cc} 2 & -1 & 1 \\ 0 & 3 & -2 \end{array}\right)$$

Damit ergibt sich als ggT die 2 und als Lösung des Ursprungssystems:

$$x = (x_1, x_2, x_3) = t * U =$$
$$(4, -5, t_3) * \begin{pmatrix} -1 & 1 & 0 \\ 3 & -2 & 0 \\ -6 & 4 & 1 \end{pmatrix} = (-19 - 6t_3, 14 + 4t_3, t_3)$$

3. lineare Ungleichungssysteme und Fourier-Elimination sukzessive Elimination der einzelnen Variablen: sei x_j die zu eliminierende Variable
   (a) sortiere die Ungleichungen in untere Schranken für x_j, obere Schranken für x_j und Ungleichungen, die x_j nicht beinhalten
   (b) normiere die beschränkenden Ungleichungen (Koeffizienten der zu eliminierenden Variablen alle gleich eins)
   (c) lösche die x_j beschränkenden Ungleichungen und füge stattdessen die Paare aller möglichen Ungleichungskombinationen ein, die sich ergibt, wenn man alle Unterschranken von x_j allen Oberschranken von x_j gegenüberstellt
   (d) das resultierende System hat u.U. wesentlich mehr Ungleichungen, aber eine Variable weniger; also: eliminiere nächste Variable
   (e) erfolgreiche Termination: keine Ungleichung oder keine Variable geblieben (keine Ungleichung = unbeschränkte Variable)
   (f) erfolglose Termination: die letzte Ungleichung (ohne Variable!!) ist nicht erfüllt

Beispiel
Lösung mit Fourier-Motskin-Algorithmus:

$$\begin{aligned} 0 &\leq t - p \\ t - p &\leq n \\ p &\geq 0 \\ p &\leq t - p + 2 \end{aligned}$$

Somit:

(a) innerste Schleife

$$\left.\begin{array}{rll} p & \leq t & \leq n+p \\ 2p-2 & \leq t & \leq n+p \end{array}\right\} \text{ t unabhängig; Schleifenrumpf:}$$

for $(t = \lceil max(p, 2p-2)\rceil; t \leq \lfloor min(n+p, n+p)\rfloor; t++)$ **do**
end

(b) t eliminiert
$0 \leq p \leq n+2\}$ erweiterter Schleifenrumpf:

for $(p = 0; p \leq n+2; p++)$ **do**
  for $(t = \lceil max(p, 2p-2)\rceil; t \leq \lfloor min(n+p, n+p)\rfloor; t++)$ **do**
  end
end

(c) p eliminiert
$0 \leq n+2\}$ lösbar, aber nur im Rationalen.

4. affine anstatt linearer Transformationen:

(a) Strukturparameter werden wie zusätzliche Variablen behandelt, die "zufällig" nur einen einzigen Wert zur Laufzeit annehmen.
Um sicherzustellen, daß sich ihr Wert nicht ändert, werden in den Transformationsmatrizen Zeilen eingefügt, die jeden Strukturparameter auf sich selbst abbilden.

(b) Eine additive Konstante wird wie ein zusätzlicher Strukturparameter behandelt, der zufällig nur den Wert 1 annimmt.

(c) Mathematischer Hintergrund: homogene Koordinaten.
Der Vektor $(x_1, \cdots, x_n, \lambda)$ in homogenen Koordinaten entspricht für $\lambda \neq 0$ dem Vektor $(\frac{x_1}{\lambda}, \cdots, \frac{x_n}{\lambda})$ in den üblichen kartesischen Koordinaten. Für $\lambda = 0$ entspricht der Vektor $(x_1, \cdots, x_n, \lambda)$ einem Punkt im Unendlichen in Richtung $(x_1, \cdots, x_n)$.

4 Abhängigkeitsanalyse im Überblick

4.1 Abhängigkeiten im allgemeinen

Abhängigkeiten in einem Programm spiegeln die durch das Programm zwingend vorgeschriebene Ausführungsreihenfolge von *Instanzen von Anweisungen* wider. In imperativen Programmen erfolgt ihre Berechnung üblicherweise durch die Ermittlung von Speicherstellen, auf die mehrfach zugegriffen wird.

Bedingungen Eine Operation o_t ist abhängig von einer Operation o_s, gdw. folgende Bedingungen erfüllt sind:

1. o_s und o_t müssen auf dieselbe Speicherzelle zugreifen (*Speicherkonflikt*).
2. o_s und o_t müssen beide durch das Programm wirklich generiert werden (*Existenz*).
3. o_s wird gem. der sequentiellen Ausführungsreihenfolge vor o_t ausgeführt—es kann nur eine später ausgeführte Instanz von einer früher ausgeführten Instanz abhängig sein (*Ordnung*).
4. Zwischen den zwei Instanzen wird der Wert der Speicherzelle nicht mehr modifiziert, d.h., es besteht eine "direkte Abhängigkeit" (*Optimierung*).
5. Mindestens einer der in Konflikt stehenden Zugriffe muß schreibend sein (*Bernsteinzusatz*).

Bemerkungen: Die Optimierung wird bei der praktischen Berechnung oft vernachlässigt.
Die Bernstein-Bedingungen werden i.a. explizit angegeben (3 Fälle) und beinhalten Ordnung, Existenz und Speicherkonflikt implizit. Je nach Analyse-Methode wird die Zusatzbedingung von Bernstein implizit bei der Auswahl der zu betrachtenden Zugriffspaare berücksichtigt.

4.2 Abhängigkeiten in Schleifenprogrammen

Die Abhängigkeits-Analysetechniken sind für verschachtelte Schleifenprogramme mit Arrays als einziger Datenstruktur relativ gut entwickelt (Skalar-Variable können dabei als null-dimensionale Arrays interpretiert werden).
Damit ergibt sich:

1. Instanzen von Anweisungen sind durch die Anweisung selbst und den zugehörigen Iterationsvektor (Vektor der Werte aller umgebenden Schleifenindizes) eindeutig charakterisiert.
2. Der Indexraum einer Anweisung ist die Menge aller durch das Schleifenprogramm aufgezählten Indexvektoren für diese Anweisung.
3. Die sequentielle Ausführungsreihenfolge von zwei Instanzen ist gegeben durch die lexikographische Ordnung auf dem gemeinsamen Präfix ihrer Iterationsvektoren und—bei deren Gleichheit—durch die textuelle Reihenfolge der Anweisungen im Programmtext.
4. Speicherkonflikt liegt genau dann vor, wenn zwei Instanzen auf dasselbe Array zugreifen und für jede Array-Dimension auch denselben Index liefern. ACHTUNG: Nicht die Schleifen-Indizes, sondern die Array-Indizes nach dem Einsetzen der Schleifen-Indizes (=Iterationsvektoren) müssen übereinstimmen.

Beispiel: 3 Schleifen, 2-dim. Arrays.

Begriffe:

1. Die Abhängigkeit geht von der *Quelle* zum *Ziel*.
2. Im Fall von perfekt verschachtelten Schleifen ist der *Abhängigkeitsvektor* (oder *Distanzvektor*) ist die Differenz: Ziel-Iterationsvektor minus Quell-Iterationsvektor. Er ist also immer lexikographisch nicht-negativ (also hinsichtlich des zeitlichen Ablaufs).
3. Eine Abhängigkeit heißt *uniform*, wenn der Abhängigkeitsvektor an jedem Punkt des Indexraumes identisch ist.
4. Durch *Richtungsvektoren*, in denen die Werte der Abhängigkeitsvektoren durch ihr Signum ersetzt werden, kann man selbst nicht-uniforme Abhängigkeiten (zu Lasten der Genauigkeit) effizient darstellen.
5. Eine Abhängigkeit wird von derjenigen Verschachtelungstiefe *getragen*, deren erster Eintrag im Richtungsvektor größer als null ist. Ist der Richtungsvektor gleich dem Nullvektor, so heißt die Abhängigket *schleifenunabhängig*.

Je nachdem, ob das Ziel oder die Quelle ein lesender oder ein schreibender Zugriff ist, werden verschiedene *Typen* von Abhängigkeiten unterschieden:

Ziel	Quelle	
	liest	schreibt
liest	input	true
schreibt	anti	output

Bemerkungen:

1. I.A. werden input dependences wegen der Bernstein-Zusatzbedinung vernachlässigt.
2. True dependences, die das Optimierungskriterium erfüllen, heißen auch flow dependences. (ACHTUNG: Diese Begriffe werden in der Literatur uneinheitlich verwendet.)

Graphische Darstellung: Die graphische Darstellung erfolgt entweder durch den *Iterations-Abhängigkeitsgraphen* oder den *Statement-Abhängigkeitsgraphen*.

5 Abhängigkeitsanalyse nach Banerjee

5.1 Normalisierung des Schleifenkopfes

Die Behandlung von Schrittweiten ungleich von eins, insbesondere von negativen Schrittweiten, macht die Abhängigkeitsanalyse aufwendig. Daher normalisiert man Schleifen oft wie folgt:

for $i := x$ **to** y **step** s **do**		**for** $i' := 0$ **to** $\frac{y-x}{s}$ **step** *1* **do**
$\mid \quad R(i)$	$\rightarrow$	$\mid \quad R(i' \cdot s + x)$
end		**end**

(vgl. [**?**], S. 175)

5.2 Konflikt-Gleichungssystem

Zwei Operations o_s und o_t müssen auf dieselbe Speicherzelle zugreifen, d.h., die Variablennamen und die Array-Indizes, die o_s und o_t ansprechen, müssen in allen Komponenten übereinstimmen. Für die entsprechenden Iterationsvektoren i_s und i_t muß also in Matrixnotation gelten:

$$i_s * A + a_0 = i_t * B + b_0,$$

wobei jede Spalte von A (B) die Koeffizienten eines Array-indexes beinhaltet und a_0 (b_0) den konstanten Anteil bechreibt [**?**].
Anders notiert:

$$(i_s, i_t) * \begin{pmatrix} A \\ -B \end{pmatrix} = b_0 - a_0$$

Lösung wie im Abschnitt für mathematische Grundlagen beschrieben.

5.3 Existenz-Ungleichungssystem

Die zwei Operations o_s und o_t müssen beide durch das Programm wirklich generiert werden, d.h., die Iterationsvektoren i_s und i_t müssen im jeweiligen Indexraum der entsprechenden Statements sein. Matrixnotation:

$$p_0 \leq i * P \qquad \text{bzw.} \qquad i * Q \leq q_0$$

für die unteren bzw. oberen Grenzen der Indexräume. Diese Ungleichungen müssen natürlich sowohl für i_s als auch für i_t aufgestellt werden.
Die – ggfs. parametrisierten – Lösungen des Konflikt-Gleichungssystems kann man in das Ungleichungssystem einsetzen und so die Lösbarkeit innerhalb des Indexraums überprüfen, bzw. ggfs. die Wahl der Parameter einschränken.

5.4 Ordnungsrelation und Abhängigkeitstypen

Bis hierher haben wir zwei Operations o_1 und o_2, die eine Abhängigkeit verursachen. Frage: Wer ist Quelle, wer Ziel? Wir wissen: Operation o_s muß gem. der sequentiellen Ausführungsreihenfolge vor o_t ausgeführt werden. Lösung: teile die (ggfs. parametrisierten) Paare von Iterationsvektoren so in zwei Mengen ein, daß in einer stets gilt $o_1 \prec o_2$ und in der anderen $o_2 \prec o_1$ (ggfs. unter Aufteilung der Räume für die Wahl der Parameter). Dadurch sind jeweils Quelle und Ziel identifiziert, was damit auch die Typen der Abhängigkeiten lt. Tabelle im vorigen Abschnitt festlegt. (Die Ordnung ist bei dem Verfahren nach Banerjee nicht a priori bekannt.)

5.5 Optimierung

Die Optimierung wird bei Banerjee komplett vernachlässigt.

5.6 Randbemerkung: Sonderfälle

Banerjee stellt Sonderfälle vor, in denen sich die Analyse etwas vereinfacht.

1. "regulärer" Schleifensatz: perfekt, $P = Q$ und $A = B$, dann für $d = i_s - i_t$. Dadurch

$$d * A = a_0 - b_0 \qquad \text{und} \qquad p_0 - q_0 \leq d * P \leq q_0 - p_0$$

   . Die Halbierung der Variablenzahl macht das Gleichungssystem einfacher und die Ordnung handlicher.

2. "rechteckiger" Schleifensatz: $P = Q = I_m$ (Identität) und A und B in Diagonalform (aber nicht unbeding identisch). Dadurch unabhängige m Gleichungen mit je 2 Variablen, die unabhängig voneinander aufgelöst werden können. Die Zeilenstufenreduktion wird daher etwas übersichtlicher, aber im Endeffekt keine spürbare Erleichterung.

5.7 Beispiel zur Abhängigkeitsanalyse nach Banerjee

Programmcode:

for $i = 10$ **to** 200 **do**
  for $j = 7$ **to** 167 **do**
    S: $X[2i + 3, 5j - 1, j]$ **:=...**
    T: ... := ... $X[i - 1, 2i - 6, 3j + 2]$
  end
end

Konfliktgleichungssystem: (gestrichene Var. f. T)

$$\begin{aligned} 2i + 3 &= i' - 1 \\ 5j - 1 &= 2i' - 6 \\ j &= 3j' + 2 \end{aligned}$$

nach Banerjee:

$A = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 5 & 1 \end{pmatrix}, a_0 = (3, -1, 0)$
$B = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}, b_0 = (-1, -6, 2)$
$(i, j) * A + a_0 = (i', j') * B + b_0$
$\Leftrightarrow (i, j, i', j') * \begin{pmatrix} A \\ -B \end{pmatrix} = b_0 - a_0$
Eingesetzt:
$$(i, j, i', j') * \begin{pmatrix} 2 & 0 & 0 \\ 0 & 5 & 1 \\ -1 & -2 & 0 \\ 0 & 0 & -3 \end{pmatrix} = (-4, -5, 2)$$
Lösen

$$\left(\begin{array}{ccc|cccc} 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 5 & 1 & 0 & 1 & 0 & 0 \\ -1 & -2 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & -3 & 0 & 0 & 0 & 1 \end{array}\right) \rightsquigarrow \left(\begin{array}{ccc|cccc} -1 & -2 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 2 & 0 \\ 0 & 0 & 1 & 5 & 4 & 10 & 1 \\ 0 & 0 & 0 & 15 & 12 & 30 & 4 \end{array}\right)$$

$(t_1, t_2, t_3, t_4) * S = (-4, -5, 2)$
$\Rightarrow t_1 = 4; t_2 = 3; t_3 = -1; t_4 \in \mathbb{Z}$
$\Rightarrow (i, j, i', j') = (t_1, t_2, t_3, t_4) * U = (-2 + 15t_4; -1 + 12t_4; 30t_4; -1 + 4t_4)$ somit kann man folgende Abhängigkeiten folgern:
$\langle(-2 + 15t_4; -1 + 12t_4), S\rangle \overset{?}{\leftrightarrow} \langle(30t_4; -1 + 4t_4), T\rangle$ für $t_4 \in \mathbb{Z}$
Existenzungleichungssystem:

$$P = Q = \left(\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array}\right), p_0 = (10, 7), q_0 = (200, 167)$$

Gleichungen:

$$\underbrace{\begin{array}{ccc} 10 & \leq -2 + 15t_4 & \leq 200 \\ 7 & \leq -1 + 12t_4 & \leq 167 \end{array}}_{\text{Existenz von}(i,j)\text{bei S}} \quad \underbrace{\begin{array}{ccc} 10 & \leq 30t_4 & \leq 200 \\ 7 & \leq -1 + 4t_4 & \leq 167 \end{array}}_{\text{Existenz von}(i,j)\text{in T}}$$

$\overset{FM}{\rightsquigarrow} 2 \leq t_4 \leq 6$
Ordnung: Richtung der Abhängigkeit $\leftarrow$ oder $\rightarrow$?

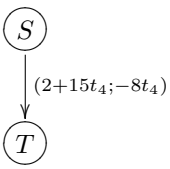
Lexikographische Ordnung: $i < i'$ oder $i = i' \wedge j < j'$

$$\begin{aligned} i < i' &\Leftrightarrow -2 + 15t_4 < 30t_4 \\ &\Leftrightarrow 15t_4 > -2 \\ &\Leftrightarrow t_4 > -\frac{2}{15} \end{aligned} \tag{1}$$

da $t_4 \geq 2 \Leftrightarrow true$
Lösung $\underbrace{\langle(-2 + 15t_4; -1 + 12t_4), S\rangle}_{\text{schreibt}\Rightarrow\text{true Dependence}} \rightarrow \underbrace{\langle(30t_4; -1 + 4t_4), T\rangle}_{\text{liest}}$ für $2 \leq t_4 \leq 6$

Distanzvektoren: „Ziel-Quelle“
$(30t_4; -1 + 4t_4) - (-2 + 15t_4; -1 + 12t_4) =$
$(2 + 15t_4; -8t_4)$ für $2 \leq t_4 \leq 6$).
Richtungsvektor: $(+, -)$
Level: 1

Statement-Abhängigkeits-Graphen:



Optimierung: nicht bei Banerjee.

Somit führt Banerjee zum Abhängigkeitsvektor und gibt alle Abhängigkeiten inklusive einiger eigentlich nutzloser aber für den Algorithmus notwendiger „Initialisierungsabhängigkeiten“ zurück (Initialisierungsschrott).

6 Datenflußanalyse und Single-Assignment-Konversion

Die Datenflußanalyse nach Feautrier [**?**] ist aufwendiger als die von Banerjee, berücksichtigt aber auch die Optimierung.

6.1 Ansatz

Im Gegensatz zu Banerjee werden nicht alle Abhängigkeiten gesucht, sondern nur echte Datenflußabhängigkeiten. Daher werden nicht alle Zugriffspaare untersucht, sondern zu jedem Lesezugriff werden alle Schreibzugriffe als mögliche Quellen betrachtet. Der Lesezugriff ist damit die Senke; der Typ der Abhängigkeiten ist *flow*.
Durch diesen Ansatz wird die Existenz des Lesezugriffs zu einer immer erfüllten Voraussetzung, die für das Konfliktgleichungssystem und die Existenz der Quelle oder auch die Optimierung zusätzliche Informationen bedeutet.
Das System aus Konfliktgleichungen, Existenzungleichungen und Ordnungsungleichungen wird einem Tool zur parametrisierten, ganzzahlig linearen Optimierung (z.B. PIP) in einem Aufwasch präsentiert.
Die (lokal bereits optimierten) Lösungsergebnisse werden in einem zweiten Arbeitsgang "gemischt".
Resultat: für Polyeder: eindeutige Quellen je Lesezugriff.

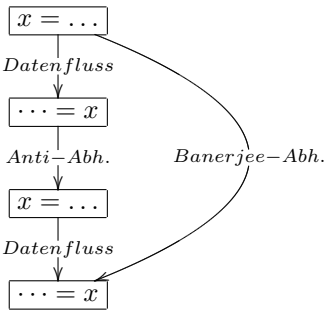
6.2 Single-Assignment-Konversion

1. einmalige Zuweisung durch "Umleiten" der Schreibzugriffe in neue Arrays, die voll indiziert sind.

2. Wiederherstellung der Semantik durch Einsetzen der Quelle in den Lesezugriffen.

6.3 Verfahren von Feautrier:

Bei Feautrier ist die Ordnung das *Input Kriterium*. Damit sind alle Gleichungen in ein System pro Dimension aufstellbar. (Für die Antiabhängigkeiten (wg. der Inputvernachlässigung) gibt es keine derartige Optimierung, da man nicht überlesen kann.)



Konfliktgleichungssystem: gleich Banerjee

Existenz-Ungleichungen: Welche „Writes "schreiben den an der aktuellen Stelle gelesenen Wert? Welches „Write "ist das letzte? ← Optimierung

Read-Existenz: Kontext für die Analyse, Feautrier liefert nur optimierten True-Deps., also die Flow-Dependences.

Folge: die Ordnung wird Input des Verfahrens:
als Ungleichungssysteme (je Fall für die lexikographiesche Ordnung eine)

6.3.1 Beispiel

Programmcode:

for $i = 0$ **to** n **do**
  R: $A[0,0] := ...$
  S: $X[i] := A[i,0]$
  for $j = 1$ **to** n **do**
    for $k = 1$ **to** n **do**
      T: $A[i+k-1, j-1] := ...$
    end
  end
end

Frage: Wer hat den Wert geschrieben, der in $\langle (i), S \rangle$ gelesen wird?

S-T-Konflikt

1. *Instanz:* $\langle (i), S \rangle$, $\langle (i', j', k'), T) \rangle$
2. *Konfliktgleichungssystem:*
$$\left.\begin{array}{l} i{=}i' + k' - 1 \\ 0{=}j' - 1 \end{array}\right\} \Leftrightarrow \begin{array}{l} k'{=}i - i' + 1 \\ j'{=}1 \end{array}$$
3. *Existenzungleichungen:* nur für i', j', k'

$$\begin{array}{l} 0 \leq i' \leq n \\ 1 \leq j' \leq n \\ 1 \leq k' \leq n \end{array}$$

   Kontext: $0 \leq i \leq n$
   (Da hier kein Interesse für Anti-Abhängigkeiten besteht, erfolgt alles im Kontext von i. „Ich bin Statement S, welche Iteration hat geschrieben, was ich gerade lese?“)
4. *Ordnung:* soll $\langle (i', j', k'), T \rangle \rightarrow \langle (i), S \rangle$

$$(i' < i) \vee \underbrace{(i' = i \wedge \text{T textuell vor S})}_{false}$$

$$\begin{array}{cccccc} i' & j' & k' & i & n & 1 \\ -1 & 0 & -1 & 1 & 0 & 1 \\ 1 & 0 & 1 & -1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 \\ -1 & 0 & 0 & 1 & 0 & -1 \end{array} \geq 0$$

(Die ersten vier Zeilen sind das Konfliktgleichungssystem, dann folgen 6 Zeilen lang das Existenzungleichungssystem, dem sich dann eine Kontextzeile und eine Ordnung ($i' < i$) anschließt.) Lösung des Gleichungssystems mit FM.

Zwischenergebnis:
$\langle (i', 1, i - i' + 1), T\rangle \rightarrow \langle (i), S\rangle$ *
für $0 \leq i' < i \leq n$, $1 \leq i - i' + 1 \leq n$
für Kontext $0 \leq i \leq n$

5. *Optimierung*
   gesucht: letzte (lex. größte) Instanz von T mit *, die vor $\langle (i), S\rangle$ ausgeführt wird.
   Lösung: $i' = i - 1$
   $\Rightarrow \langle (i-1, 1, 2), T\rangle \rightarrow \langle (i), S\rangle$ für $i \geq 1$

S-R-Konflikt

1. $\langle (i), S\rangle$,$\langle (i'), R\rangle$
2. $0 = i \wedge 0 = 0$
3. $0 \leq i' \leq n$ im Kontext $0 \leq i \leq n$
4. $i' < i \vee (i' = i \wedge$ R vor S$) \Leftrightarrow i' \leq i$
5. $i' = i$
   $\Rightarrow \langle (i), R\rangle \rightarrow \langle (i), S\rangle$ für $i = 0$
   $\Rightarrow \langle (0), R\rangle \rightarrow \langle (0), S\rangle$

Datenfluss:

src($\langle (i), S\rangle$) =
if $i = 0$ **then**
| $\langle (0), R\rangle$
else
| $\langle (i-1, 1, 2), T\rangle$
end

Vorsicht: Falls die Bedingungen der Schleifenindizes nicht disjunkt sind, geht die Abhängigkeit von Maximum der Quellen (bzgl. der Ausführungsordnung) aus.

6.4 Verfahren via CfFada:

CfFada heißt ausgeschrieben „Controllflow based Fuzzy array dataflow“.
Dieses weiteres Verfahren wird motiviert durch die Existenz von „if-Statements“ , „unstrukturierte Programme“ und die erhöhte „Präzision“.

Idee:

1. Zusammenführen von
   (a) traditionellen (universell einsetzbar, ungenau) und
   (b) einem Feautrier-ähnlichem Verfahren
2. (Effizienz)

Prinzipielles Vorgehen:

1. Konstruiere den Kontrollflußgraphen (CFG)
2. Für jeden Zugriff a:
   (a) Annotiere den CFG mit den für a relevanten Relationen - unter Berücksichtigung der umgebenden Prädikate
   (b) starte mit leerer Abhängigkeitsmenge (alles bottom) und Ordnungsrelation $(=, ..., =)$ bei a. $(i = i' = \ldots)$
   (c) durchlaufe den CFG rückwärts
   (d) bei Schleifen: durchlaufe die Rückwärtskanten von innen nach außen und passe die aktuell gültige Ordnungsrelation an
   (e) bei „conflicting accesses“: mische die neuen und alten Abhängigkeiten unter Berücksichtigung der Optimierung

Anmerkungen:

1. Implementierung mit Omega (Presburger Arithmetik) → Genauigkeit
2. unstrukturierte Programme → Abschätzung der Ordnungsrelation
3. vorzeitiges Analyse-Ende möglich, wenn alle möglichen letzten Quellen gefunden wurden.
4. Der Hauptunterschied zu Feautrier besteht in der Tatsache, dass CfFada mit if-Statemants umgehen kann.
5. textuell getragene Abhängigkeiten entspricht sprachlich den Schleifenunabhängigen Abhängigkeiten
6. Mischen: Schneiden und Optimierung
7. $\square :=$ Merge/Mischen
8. $W(6) :=$ Write-Set von 6

6.4.1 Beispiel zur Abhängigkeitsanalyse nach CfFada

Programmcode:

for $i = 0$ **to** N **do**
  for $j = 0$ **to** N **do**
    3: $A[i + j + 1] =$ **...**
    if P **then**
      6: $A[i + j] =$ **...**
    else
    end
    7: ... $= A[i + j]$
  end
end

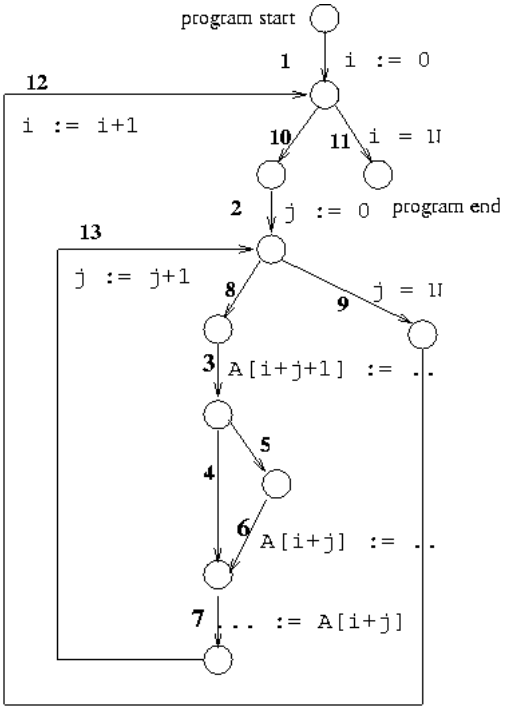


Abbildung 5: Ablaufgraph

Kontext
Read-Statement 7
Operationen: $\langle (i, j), 7 \rangle$ für $0 \leq i, j \leq N$

Annotierung
Statement 3
Operationen: $\langle (i', j'), 3 \rangle$ für $\underbrace{0 \leq i', j' \leq N}_{Existenz} \wedge \underbrace{i' + j' + 1 = i + j}_{Konflikt}$

Statement 6
Operationen: ...

Durchlauf

1. Ordnung: $i = i' \wedge j = j'$
   $src^{(0)} = \perp$

   (a) Start: 7

   (b) die 4 hoch: (Kante mit Label 4)
   Keine neuen Quellen
   $scr_4^{(1)} = \{\perp\}$

(c) die 6 hoch: (überprüfe Existenzun- und Konfliktgleichungen)
$src_6^{(1)} = src^{(0)} \square W(6) =$
$\begin{cases} \langle (i,j), 6 \rangle, & \text{für } P'(i,j) \\ \bot, & \text{sonst} \end{cases}$

(d) bei n1 Mischen:
$scr_{n1}^{(1)} = \begin{cases} \langle (i,j), 6 \rangle, & \text{für } p'(i,j) \\ \bot, & \text{sonst} \end{cases}$

(e) die 3 hoch:
Keine neuen Quellen, da die aktuelle Ordnung $\lightning$ Konfliktgleichung. (Wäre P immer true, könnte man an n1 abbrechen.)

(f) die 13 hoch $\Rightarrow$

2. Ordnung: $i = i' \wedge j' < j$

(a) die Kanten 7, 4 bringen nichts neues.

(b) die 6 hoch:
Keine neuen Quellen, da die aktuelle Ordnung $\lightning$ Konfliktgleichung.

(c) n1: nichts neues zu mischen $\Rightarrow$ alles bleibt

(d) die 3 hoch:
$src^{(2)} = src^{(1)} \square W(3) =$
$= \begin{cases} \langle (i',j'), 3) \rangle, & \text{für } i'+j'+1 = i+j \wedge \neg P'(i,j) \wedge 0 \leq i' \leq N \wedge 0 \leq j' \leq N \\ & \text{mit aktueller Ordnung} \\ \langle (i,j), 6 \rangle, & \text{für } P'(i,j) \\ \bot, & \text{sonst} \end{cases}$
$\overset{j'+1=j}{=} \begin{cases} \langle (i,j-1), 3 \rangle, & \text{für } \neg P'(i,j) \wedge j > 0 \wedge 0 \leq j-1 \leq N \\ \langle (i,j), 6 \rangle, & \text{für } P(i,j) \\ \bot, & \text{sonst, d. h. } j = 0 \wedge \neg P(i,0) \end{cases}$

(e) Kante 8: -

(f) Kante 2: -

(g) Kante 10: -

(h) Kante 12: -

3. Ordnung: $i' < i, j'$ beliebig

(a) Kante 9,13,7,4: -

(b) Kante 6:
wegen Optimierung: $j = 0 \wedge \neg P'(i,j)$
Ordnung: $i' < i$
Konflikt: $i' + j' = i + 0$
Existenz: $P'(i', j')$
neue Quellen: $\langle (i', i - i'), 6 \rangle$ für $j = 0, i' < i, P(i', i - i') \wedge \neg P(T, i -$

T) für $i' < T \leq i$

$$src_{n1}^{(3)} = \begin{cases} \langle (i,j), 6 \rangle, & \text{für } P(i,j) \\ \langle (i,j-1), 3 \rangle, & \text{für } j \geq 1 \wedge \neg P(i,j) \\ \langle (i',i-i'), 6 \rangle, & \text{für } j = 0 \wedge i' < i \wedge P(i',i-i') \wedge \neg P(T,i-T) \\ & \text{für } i' < T \leq i \wedge i' \geq 0 \wedge i \geq 1 \\ \bot, & \text{für } j = 0 \wedge \neg P(i',i-i') \text{ für alle } 0 \leq i' \leq i \end{cases}$$

(c) Kante 3:

wegen Optimierung: $i' = i-1, j = 0 \overset{Konfl.Gl.}{\Rightarrow} j' = 0$
neue Quelle: $\langle (i-1,0), 3 \rangle$für$j = 0 \wedge i \geq 1 \wedge \neg P(i,0) \wedge \neg P(i-1,1)$

$$\Rightarrow src = \begin{cases} \langle (i,j), 6 \rangle, & \text{für } P(i,j) \\ \langle (i,j-1), 3 \rangle, & \text{für } j \geq 1 \wedge \neg P(i,j) \\ \langle (i-1,1), 6 \rangle, & \text{für } j = 0 \wedge \neg P(i,j) \wedge P(i-1,1) \\ \langle (i-1,0), 3 \rangle, & \text{für } j = 0 \wedge \neg P(i,j) \wedge \neg P(i-1,1) \wedge i \geq 1 \\ \bot, & \text{für } i = j = 0 \wedge \neg P(0,0) \end{cases}$$

(d) Kanten 8, 2, 10, 1: -

Da die Quelle für $(0,0)$ nicht gefunden wird, kann nicht vorab abgebrochen werden und der Algorithmus terminiert erst mit dem vollständigen Durchlauf.

7 Vereinfachte Abhängigkeitsanalyse

Anstatt die existierenden Abhängigkeiten oder gar den Datenfluss genau zu bestimmen, kann man die Abhängigkeitsanalyse auch als Entscheidungsproblem betrachten: "Gibt es Instanzen zweier Statements, so dass diese Instanzen abhängig voneinander sind?"
Man ist in diesem Fall nicht an den Instanzen interessiert, sondern nur noch an der *Abhängigkeit zwischen Statements* mit einem vorgegebenen Richtungsvektor.
Grund: Die NP-harte ganzzahlig lineare Programmierung soll aus Rechenzeitgründen durch (semi-exakte) Heuristiken ersetzt werden. Idee: Algorithmus sagt "unabhängig", dann ist das garantiert, ansonsten gehen wir (pessimistischerweise) von einer Abhängigkeit aus.
Wir werden im folgenden eine Auswahl an solchen Heuristiken vorstellen.

7.1 GCD-Test [?]

Eindimensionale Arrays: Seien $X(a_1 * i_1 + \cdots + a_m * i_m + a_0)$ und $X(b_1 * i_1 + \cdots + b_n * i_n + b_0)$ die zu untersuchenden Arrayzugriffe. Wenn $\gcd(a_1, \cdots, a_m, b_1, \cdots, b_n)$ die Differenz $(b_0 - a_0)$ nicht teilt, dann kann es die vermutete Abhängigkeit nicht geben (Kapitel 3, Punkt 2.(g)).

Mehrdimensionale Arrays: Führe den einfachen GCD-Test unabhängig für alle Gleichungen (=Arraydimensionen) durch. Unabhängig, sobald unabhängig als Resultat für irgendeine Gleichung.

7.1.1 Beispiele

Folgende Beispiele eins und zwei sind garantiert lösbar jedoch leider nur potenziell Abhängig, da die Existenz und Ordnung nicht berücksichtig werden.

1. $$4 \cdot x_1 + 6 \cdot x_2 = 8$$

   ganzzahlig Lösbar:

   $$\begin{aligned} ggT(4,6) &| 8 \\ 2 &| 8 \end{aligned}$$

2. $$x_1 + 6 \cdot x_2 = 9 \rightarrow \text{Lösbar, da } ggT = 1$$

3. $$\left.\begin{aligned} 4 \cdot x_1 + 6 \cdot x_2 = 8 \quad &\leftarrow \text{lösbar} \\ x_1 + 6 \cdot x_3 = 9 \quad &\leftarrow \text{lösbar} \end{aligned}\right\} \text{potentiell lösbar}$$

   Ein System von Gleichungen.

   Lösungsansatz:

   $$\begin{pmatrix} x_1, & x_2, & x_3 \end{pmatrix} \begin{pmatrix} 4 & 1 \\ 6 & 0 \\ 0 & 6 \end{pmatrix} = \begin{pmatrix} 8, & 9 \end{pmatrix}$$

   Wir hatten: Statt $x \cdot A = b$
   löse $t \cdot S = b$
   mit S ist Zeilenstufenform von A. Dann $x = t \cdot U \Rightarrow S = U \cdot A$

Verallgemeinerter gcd-Test: Berechne die in Kapitel 3, Punkt 2.(i) angegebene Existenz von t, stoppe aber dann mit der genauen Berechnung der Abhängigkeiten. (Damit werden auch die Räume nicht überprüft.) Dies garantiert die gleichzeitige Erfüllung aller GCD-Tests für sämtliche Arraydimensionen.

7.1.2 Beispiel

Kein Interesse an den Werten von x, somit Abbruch nach Berechnung von t. Das ist erlaubt, da U unimodular und damit „ganzzahligkeitserhaltend“. Im obigen Beispiel:

$$A \rightsquigarrow \begin{pmatrix} 2 & -1 \\ 0 & 3 \\ 0 & 0 \end{pmatrix}$$

$$\Rightarrow 2 \cdot t_1 = 8 \Rightarrow t_1 = 4$$

$$-t_1 + 3 \cdot t_2 = 9$$

$$\Leftrightarrow 3 \cdot t_2 = 13 \Rightarrow \text{nicht ganzzahlig lösbar}$$

mehrdimensionale GCD-Test ist „semi-korrekt“

1. einmal unlösbar ⇒ gesamtes System unlösbar
2. alle lösbar ⇒ vielleicht lösbar

erweiterter GCD-Test ist (voll) Korrekt (aber berücksichtigt nicht die Existenz und nicht die Ordnung).

7.2 Separability-Test ([?], S. 149 ff.)

Voraussetzung: je Gleichung kommt nur eine Schleifenvariable vor. Der Test ist exakt und stellt im wesentlichen eine optimierte, d.h., auf die Voraussetzung zugeschnittene Version der allgemeinen Lösung dar. $a \cdot i + b \cdot i' = ...$
Vorgehen:
Teste $a \stackrel{?}{=} 0 \stackrel{ja}{\rightarrow}$ Lösung durch Einsetzen in vorberechnete Formel
sonst teste $b \stackrel{?}{=} 0 \stackrel{ja}{\rightarrow}$ Lösung durch Einsetzen in vorberechnete Formel
sonst teste $a \stackrel{?}{=} b \stackrel{ja}{\rightarrow}$ Lösung durch Einsetzen in vorberechnete Formel
sonst alles durchrechnen.

7.3 Two-Variable-Exact-Test ([?], S. 238 f.)

Voraussetzung: jede Gleichung hat maximal zwei Variablen (z.B., eindimensionaler Indexraum). Der Test ist exakt und stellt “vorberechnete” Lösungsmuster für die gemäß Voraussetzung zugeschnittene Lösung dar. Vgl. auch [**?**], S. 66 ff.
Vorgehen:
1. Konfliktgleichung lösbar ? nein → fertig.
sonst 2. Existenzungleichung lösbar ? nein → fertig.
sonst 3. Ordnung lösbar ? nein → fertig.

7.4 Extreme-Value-Test ([?], S. 236 ff.)

Grobidee: Die Unter- bzw. Obergrenzen werden für die Schleifenindizes in den Arrayausdrücken substituiert. Wenn Überlagerung auftreten kann, dann wird von einer Abhängigkeit ausgegangen. Etwas genauer:
Eindimensionale Arrays, zunächst eine Schleife: Setzt in die Konflikt-Gleichung die Variablengrenzen für i_s und i_t so ein, daß der Wertebereich des Variablenteils der Konfliktgleichung abgeschätzt wird (primitive Maximierung/Minimierung), und testet dann, ob die Konstante im Wertebereich liegt.
Verallgemeinerung auf mehrere Schleifen: Wenn Test auf $(*, *, \cdots)$ nicht reicht, dann Aufteilen in $(<, *, \cdots)$, $(=, *, \cdots)$ und $(>, *, \cdots)$. Idee: $<$, $=$ und $>$ schränken die erlaubten Werte für die Indizes i_s und i_t in der aktuellen Schleifendimension ein (im Bsp. Dim. 1). Damit: für jeden der drei Fälle neu testen und ggfs. in die nächsten Dimensionen weiterverfeinern.

7.4.1 Beispiel

Konfliktlgiechung: $-M + i - i' = 0$
Grenzen:

$$\begin{aligned} 1 &\leq M \\ 0 \leq i &\leq 9 \\ 0 \leq i &\leq 9 \end{aligned}$$

Vorgehen: 1) Eliminiere in der Konfliktgleichung eine Variable (auflösen eine nach der anderen, indem die Grenzen maximiert bzw. minimiert wird.)
2) Teste, ob die rechte Seite von der Konfliktgleichung in den Grenzen liegt.

lower	upper	elim
$-M + i - i'$	$-M + i - i'$	i'
$-M + i - 9$	$-M + i - 0$	i
$-M - 9$	$-M + 9$	M
$-\infty$	8	

Weil $0 \in]-\infty, 8] \Rightarrow$ potenziell abhängig.

Beschränkungen:

1. maximal eine Untergrenze und eine Obergrenze je Variable
2. Sortierung der Variablen so, dass die Grenzen der einen Variable unabhängig von den anderen sind. (aus Aufwandsgründen ist hier die Verwendung von FM nicht zu empfehlen)

Erweiterung: Hinzunahme der Ordnung
wir nehmen

$$\begin{aligned} i &< i' \\ \text{oder } i &= i' \\ \text{oder } i &> i' \end{aligned}$$

zu den Grenzen hinzu.
In unserem Beispiel angewandt:

$$0 \leq i < i' \leq 9$$
$$0 \leq i$$

$$\left.\begin{array}{rl} i & < i' \\ i & \leq 9 \end{array}\right\} \lightning$$

$$\lightning \left\{\begin{array}{rl} 0 & \leq i' \\ i' & \leq 9 \\ i & < i' \end{array}\right.$$

Wegen (2) darf man nur bei einem Konfliktpaar $i < i'$ verwenden.

$$\begin{array}{rcccc} & & & & \textit{Alternative} \\ \Rightarrow & 0 & \leq i & \leq 9 & \left(\begin{array}{ccc} 0 & \leq i & < i' \\ 0 & \leq i' & \leq 9 \end{array}\right) \\ & i & < i' & \leq 9 & \\ \Leftrightarrow & i+1 & \leq i' & & \end{array}$$

lower	upper	elim
$-M+i-i'$	$-M+i-i'$	i'
$-M+i-9$	$-M+i-(i+1)$	i
$-M-9$	$-M+1$	M
$-\infty$	-2	

$\Rightarrow$ Keine Abhängigkeit da $0 \notin]-\infty, -2]$
Bei mehreren Arraydimensionen:

1. Test jeder Konfliktgleichung seperat
2. Keine Abhängigkeit wenn mindestens ein Test fehlschlägt

Beispiel
Bei mehreren Schleifendimensionen:
Im Prinzip: $\{<, =, >\}^{d \leftarrow \text{Anzahl Schleifen}}$ somit 3^d Fälle

Variante: Bounds-Test Führe die Abschätzung nicht auf der in Normalform gebrachten Konflikt-Gleichung durch, sondern mache die Abschätzung auf den beiden Array-Zugriffen separat und teste dann auf Überlappung. Vorteil: die Semantik ist klarer, was eine verbesserte von-Hand Analyser erlaubt (s. Übung).

7.5 Omega-Test ([?])

Der Omega-Test kombiniert ein eigenständiges Eliminationsverfahren für ganzzahlige Gleichungen mit einer Erweiterung des Fourier-Motzkin Verfahrens.

1. Wenn in einer Gleichung ein Koeffizient einer Variablen betragsmäßig gleich eins ist, dann löse nach dieser Variablen auf und setze sie in allen anderen Gleichungen ein. Dadurch reduziert sich die Zahl der Gleichungen um eins. Gehe zu 1.
2. Ansonsten:
   (a) Sei k der Index der Variablen mit dem betragsmäßig kleinsten Koeffizienten $|a_k|$ mit $a_k \neq 0$.
   (b) Setze $m := |a_k| + 1$.

(c) Mit der Definition

$$a \widehat{\mod} b = a - b * \left\lfloor \frac{a}{b} + \frac{1}{2} \right\rfloor$$

berechne die Substitution

$$x_k = -sign(a_k) * m * \sigma + \sum_{i \in V \setminus \{k\}} sign(a_k) * (a_i \widehat{\mod} m) * x_i$$

und die daraus resultierende Gleichung

$$-|a_k|\sigma + \sum_{i \in V \setminus \{k\}} \left(\left\lfloor \frac{a_i}{m} + \frac{1}{2} \right\rfloor + (a_i \widehat{\mod} m) \right) * x_i = 0,$$

wobei σ eine neue Variable ist, die x_k ersetzt.

Folge: Die nicht-ersetzten Variablen haben echt kleinere Koeffizienten (max. 2/3 des ursprünglichen Wertes), wodurch nach einigen Iterationen Koeffizienten vom Betrag eins entstehen.

(d) Das neue System entsteht, indem man die Gleichung, in der a_k steht, durch die soeben berechnete Gleichung ersetzt und in allen anderen (Un-)Gleichungen x_k durch den soeben berechneten Ausdruck für die Substitution ersetzt (und dann vereinfacht).

(e) Falls noch Gleichungen vorhanden sind, gehe zu 1.
Ansonsten gibt es nur noch Ungleichungen!

3. Bei widersprüchlichen Ungleichungen: Unlösbar. Stop.

4. Bei gegengleichen Ungleichungen: behandle sie als Gleichung. (Aus Effizienzgründen beim Suchen nach gegengleichen Ungleichungen auch gleich: eliminiere trivial redundante Ungleichungen.)

5. Falls nun höchtens eine Variable: Ganzzahlig lösbar. Stop.

6. Sonst: Dimensionsreduktion durch Fourier-Motzkin. Achtung: ganzzahlige Variante:

   (a) Wähle eine geeignete Variable zur Elimination aus (berücksichtige "exakte Schatten", kombinatorische Explosionsvermeidung bei der Paar-Bildung, kleine Koeffizienten).

   (b) Berechne mit FM den "reellen Schatten": für jedes der Paare $a * x \leq \alpha$ und $b * x \geq \beta$ eliminiere x durch $b * \alpha - a * \beta \geq 0$.

   (c) Wenn keine ganzzahlige Lösung im reellen Schatten, dann ganzzahlig unlösbar. Stop.

   (d) Berechne den "dunklen Schatten" mit FM aus den Ungleichungen $b * \alpha - a * \beta \geq (a-1) * (b-1)$ für jedes der Paare $a * x \leq \alpha$ und $b * x \geq \beta$. [Anmerkung: $a * (x+1) \leq \alpha$ und $b * (x-1) \geq \beta$, also umgeformt $b * \alpha - a * \beta \geq 2 * a * b$, produzieren einen "noch dunkleren Schatten".]

   (e) Wenn ganzzahlige Lösungen im dunklen Schatten, dann ganzzahlig lösbar. Stop.

   (f) Wenn reeller und dunkler Schatten identisch (*exakte Projektion*), dann siehe Lösbarkeit am reellen Schatten – etwa für $a = 1$ oder $b = 1$.

(g) Ansonsten: Teste Lösbarkeit in der folgenden Serie (wegen variablem i und variabler Untergrenze) von Problemen: Füge zum Ursprungsproblem die Gleichung $b*x = \beta + i$ hinzu, wobei $\beta \leq b*x$ eine Untergrenze von x ist und, für den größten in einer oberen Grenze für x vorkommenden Koeffizienten a, i in folgendem Intervall liegt: $0 \leq i \leq \lfloor \frac{a*b-a-b}{a} \rfloor$. Lösbarkeit des Ursprungssystems gdw. mindestens ein Element der Serie lösbar.

Anmerkung: Neben der Entscheidungs-Variante gibt es auch eine Lösungsvariante, auch für parametrische Probleme.

7.5.1 Beispiel

$$7 \cdot x + 12 \cdot y \qquad +31 \cdot z - 17 = 0 \tag{2}$$
$$3 \cdot x + 5 \cdot y \qquad +14 \cdot z - 7 = 0 \tag{3}$$
$$1 \leq x \qquad \leq 40 \tag{4}$$

$\Rightarrow a_k = 3 \Rightarrow m = 4$

$$5 \widehat{\mod} 4 = 5 - 4 \cdot \underbrace{\lfloor \frac{5}{4} + \frac{1}{2} \rfloor}_{1} = 1$$
$$14 \widehat{\mod} 4 = 14 - 4 \cdot \underbrace{\lfloor \frac{14}{4} + \frac{1}{2} \rfloor}_{4} = -2$$
$$-7 \widehat{\mod} 4 = -7 - 4 \cdot \underbrace{\lfloor \frac{-7}{4} + \frac{1}{2} \rfloor}_{-2} = 1$$

alte Variable x, neue Variable σ

$$x = -4\sigma + y - 2 \cdot z + 1$$
$$7 \cdot x = -28\sigma + 7 \cdot y - 14 \cdot z + 7$$

$$-3\sigma + 2 \cdot y + 2 \cdot z - 1 = 0 \tag{5}$$
$$-28\sigma + 19 \cdot y + 17 \cdot z - 10 = 0 \tag{6}$$
$$1 \leq -4\sigma + y - 2 \cdot z - 1 \leq 40 \tag{7}$$

$a_k = 2, m = 3,$ alte Variable: $y,$ neue Variable: τ

$$-2 \widehat{\mod} 3 = 2 - 3 \cdot 1 = -1 - 1 \widehat{\mod} 3 = -1 - 3 \cdot 0 = -1 \Rightarrow y = -1 \cdot 3 \cdot \tau - z - 1$$

$$-2\tau + -1\sigma + 0z - 1 \tag{8}$$
$$-57\tau - 2z - 28\sigma - 29 = 0 \tag{9}$$
$$1 \leq -4\sigma - 3\tau - 3z \leq 40 \tag{10}$$

$$\sigma = -2\tau - 1 \tag{11}$$
$$-\tau - 2z - 1 = 0 \tag{12}$$
$$1 \leq 5\tau - 3z + 4 \leq 40 \tag{13}$$
$$\tag{14}$$

$$\tau = -2z - 1 \tag{15}$$
$$1 \leq -13z - 1 \leq 40 \quad \Leftrightarrow \tag{16}$$
$$2 \leq -13z \leq 41 \quad \Leftrightarrow \tag{17}$$
$$\lceil \frac{-41}{13} \rceil \leq z \leq \lfloor \frac{-2}{13} \rfloor \quad \Leftrightarrow \tag{18}$$
$$-3 \leq z \leq -1 \tag{19}$$

7.6 Kombinationen

Natürlich kann man beliebige Heuristiken miteinander kombinieren, um die Präzision zu erhöhen. Etwa: Falls Separability-Test anwendbar, dann berechne mit diesem ein exaktes Ergebnis; ansonsten gehen von Abhängigkeit aus, wenn weder der einfache GCD Test noch der Bounds-Test die Abhängigkeit ausschließen können.
Die Kombination von gcd-Test und Extreme-Value-Test, angewandt auf jede einzelne Array-Koordinate und auf die Linearisierung des Arrayzugriffs (etwa durch zeilenweise Speicherung des Arrays) wurde aufgrund der Einfachheit dieser Tests früher in Compilern häufig verwendet. Heutzutage verwendet man eher den verallgemeinerten gcd-Test und die Fourier-Motzkin-Projektion, sofern man nicht gleich die exakte Berechnung (ohne Optimierung) durchführt. (Vgl. [**?**], S. 249)

8 Normalisierung der Array-Indizes

Einfache Abhängigkeitsanalyse macht keine symbolische Auswertung, insbesondere setzt sie Skalarvariablen, die als Arrayindizes auftreten, nicht ein, obwohl man deren Wert manchmal bestimmen kann. Beispielsweise wäre im folgenden Programm jede Instanz der Array-Zuweisung von jeder anderen ausgabeabhängig, weil der Arrayindex x als unbekannt angenommen wird:

for $i := 1$ **to** n **do**
  $x := 2 \cdot i + 4$;
  $A[x] := ...$;
end

8.1 Skalar-Vorwärts-Ersetzung

Direktes Einsetzen der Definition des Skalars in den Arrayindex liefert das linke Programm. ACHTUNG: Gültigkeitsbereiche beachten! Im rechten Programm ist das Ersetzen des Arrayindexes j durch seine Definition $i + 1$ *verboten* (vgl. [**?**], S. 178 f.):

for $i := 1$ **to** n **do**
  $x := 2 \cdot i + 4$;
  $A[2 \cdot i + 4] := ...$;
end

$j := i + 1$;
$i := 0$;
$A[j] := ...$;

8.1.1 Beispiel

for $i = ..$ **to** n **do**
  $x = 2i + 4$
  $... = A[x]$
end

Abhängigkeit: $i \rightarrow i + 1$ schon wegen x

$\rightsquigarrow$

for $i = 1$ **to** 1000 **do**
  $... = A[2i + 4]$
end

$\Rightarrow$ keine Abhängigkeit

8.2 Wrap-Around-Variablen-Ersetzung

Ähnlich zu Fall 8.1; allerdings ist die Skalar-Definitioin textuell nach dem Arrayzugriff.
Das Problem ist einfach mit *loop rerolling* zu lösen, wenn die Initialisierung "zum Schleifenprogramm paßt"; im allgemeinen ist aber *loop peeling* nötig, um Unregelmäßigkeiten an den Schleifengrenzen zu beseitigen. Den Hauptanwendungsbereich für diese Transformation bilden zyklische Arrays (daher wrap-around; vgl. [**?**], S. 183).
Beispiel: Wenn $c = 2$ ist, dann ist das linke Programm äquivalent zu dem mittleren; ansonsten ist es nur äquivalent zu dem rechten Programm:

```
x := c;
for i := 0 to n do
|   A[x] := ...;
|   x := 2 · i + 4;
end
```

```
for i := 0 to n do
|   x := 2 · i + 4 − 2;
|   A[x] := ...;
end
x := 2 · n + 4;
```

```
A[c] := ...;
for i := 1 to n do
|   x := 2 · i + 4 − 2;
|   A[X] := ...;
end
x := 2 · n + 4;
```

8.3 Induktions-Variablen-Ersetzung

Einfache Induktionsvariablen sind Variablen, denen nur durch (ggfs. mehrere) Terme $v := v + c_0$ Werte zugewiesen werden; allgemeine Induktionsvariablen sind Variablen, denen einmalig mit einem Term der Form $v' := c_1 * v + c_2$ ein Wert zugewiesen wird, wobei v eine einfache Induktionsvariable sein muß. Sie können stets durch einen affinen Ausdruck in den Schleifenindizes ersetzt werden:

```
x := 2;
for i := 0 to n do
|   x := x + 2;
|   A[x] := ...;
end
```

ist äquivalent zu

```
for i := 0 to n do
|   x := 2 · i + 4;
|   A[x] := ...;
end
```

Anschließend an 8.2 und 8.3 kann man 8.1 anwenden, und schließlich kann man das entstandene Programm mit Hilfe von *Dead-Code-Eliminierung* noch vereinfachen.

9 Eliminieren von Abhängigkeiten

9.1 Scalar Renaming

Anti- und Output-Abhängigkeiten können u.U. durch einfaches Umbenennen von Variablen eliminiert werden. Zentrale Technik: Ermitteln der "erreichenden Definitionen".

9.1.1 Beispiel

$$\begin{array}{rcl} A = ... & & A = ... \\ = A & & = A \\ = A & \rightsquigarrow & = A \\ A = ... & & A' = ... \\ = A & & = A' \end{array}$$

9.2 Scalar Expansion

Schleifengetragene Anti- und Output-Abhängigkeiten von nicht voll indizierten Variablen können durch volle Indizierung der Variablen eliminiert werden (vgl. Single-Assignment-Konversion).

9.2.1 Beispiel

for $i = 1$ **to** 1000 **do**
| A = ...
| = A
| = A
end

$\rightsquigarrow$

for $i = 1$ **to** 1000 **do**
| $A[i] = ...$
| $= A[i]$
| $= A[i]$
end

9.3 Variable Copying

Antiabhängigkeiten kann man u.U. durch eine vorab erstellte Hilfskopie des Arrays einfach eliminieren.

9.3.1 Beispiel

for $i = 0$ **to** n **do**
| $A[i] = ...$
| $= A[i] + A[i+1]$
end

$\rightsquigarrow$

for $i = 0$ **to** n **do**
| $A'[i] = A[i+1]$
end

for $i = 0$ **to** n **do**
| $A[i] = ...$
| $= A[i] + A'[i]$
end

9.4 Index Set Splitting

Wenn die Abhängigkeit in unterschiedlichen Teilen des Indexraumes verschieden ist, dann kann man u.U. die Schleifen so aufteilen, daß schleifengetragene Abhängigkeiten eliminiert werden.

9.5 Single-Assignment-Konversion

1. Feautriers Abhängigkeitanalyse laufen lassen $\rightsquigarrow$ „Datenfluss“
2. Jedes „write“ wird zu einem neuen „write“ in ein neues, voll indiziertes Array.
3. Jedes „read“ wird ersetzt durch sein ... gemäß Datenfluss

9.6 Weitere Möglichkeiten

Node splitting (komplexe Statements aufbrechen), loop peeling, unrolling, rerolling (Unregelmäßigkeiten in den Schleifen eliminieren und vorgegebene Statement-Reihenfolge ermöglichen) und idom recognition (Mustererkennung zum Einsatz von Reduktionen).

10 Ausgewählte Parallelisierungstechniken

10.1 Iteration Graph Partitioning

Idee: *Parallel kann ausgeführt werden, was nicht voneinander abhängig ist.*
Die schwachen Zusammenhangskomponenten des Iterations-Abhängigkeitsgraphen werden zueinander parallel und in sich sequentiell ausgeführt.
Bei verschachtelten Schleifen berechnet man dazu für jede Schleifendimension den größten gemeinsamen Teiler *gcd* der Abhängigkeitsvektoren. Grund: der Abhängigkeitsgraph zerfällt in gcd viele unabhängige Teile (die Iterationen i, die durch i modulo gcd in unterschiedliche Klassen fallen, können nie voneinander abhängig sein).
Anschließend kann man mechanisch jede Schleife durch ein Paar von Schleifen ersetzen: eine parallele Schleife zählt die möglichen Werte für "Index modulo gcd" auf und eine sequentielle die jeweils auftretenden Resultate der ganzzahligen Division "Index / gcd" ([**?**], 124ff).
for $i := x$ to y do $R(i)$ enddo wird, wenn der größte gemeinsame Teiler der Abhängigkeitsvektoren der Dimension i gleich g ist, zu

forall $m := 0$ **to** $g - 1$ **do**
  for $d := \lceil \frac{x-m}{g} \rceil$ **to** $\lfloor \frac{y-m}{g} \rfloor$ **do**
    $R(m + g \cdot d)$
  end
end

Im folgenden nehmen wir der Einfachheit halber an, daß der zu parallelisierende Schleifensatz perfekt verschachtelt ist.

10.2 Schleifenpermutation

Ideen: *Eine Schleife kann parallel ausgeführt werden, wenn sie keine Abhängigkeiten trägt*, und: *Abhängigkeitsvektoren dürfen nie negativ sein.*
Damit kann man zwei Schleifen genau dann vertauschen, wenn durch die entsprechende Permutation im Richtungsvektor kein lexikographisch negativer Richtungsvektor entsteht.
Wenn alle Abhängigkeiten (ggf. nach Permutation) durch eine Anzahl von äußeren Schleifen getragen wird, dann kann man die inneren alle parallel ausführen.

10.3 Unimodulare Transformationen

Idee: *Durch Mischen (Scheren, "skewing") von einer (inneren) Schleife in eine andere (äußere) kann man die tragende Schleife einer Abhängigkeit verändern.*
Im folgenden sei m die Dimensionalität des Schleifensatzes. Um konsistent mit dem Space-Time-Mapping-Ansatz zu sein (Kapitel 2), multiplizieren wir Vektoren nun wieder von rechts an die Transformationsmatrix (im Gegensatz zu [**?**, **?**] und dem Beschluß in Kapitel 3). Damit werden auch die Abhängigkeitsvektoren zu Spaltenvektoren (im Gegensatz zur vorangegangenen Abhängigkeits-Vorlesung). Die *Distanzmatrix* (=*Abhängigkeitsmatrix*) $\mathcal{D}$ ist dann eine $m \times r$-Matrix, die spaltenweise aus den Distanzvektoren (=Abhängigkeitsvektoren) $d_1, \cdots, d_r$ aufgebaut ist.

10.3.1 Parallelisierung innerer Schleifen

Im Fall von uniformen Abhängigkeiten kann man jedes Schleifenprogramm so transformieren, daß alle Schleifen außer der äußersten Schleife parallel sind. Die dazu

nötige Transformationsmatrix wird so konstruiert, daß alle Abhängigkeiten von der äußersten Schleife getragen werden.
Etwas präziser: es wird ein Vektor u (erste Zeile der Transformationsmatrix) gesucht, so daß

1. $gcd(u_1, \cdots, u_m) = 1$
2. $u_1 * \mathcal{D}_{1k} + \cdots + \mathcal{D}_{mk} * u_m \geq 1$ für alle Abhängigkeitsvektoren $1 \leq k \leq r$
3. $\max_{I \in \text{Indexraum}}(u * I) - \min_{I \in \text{Indexraum}}(u * I) + 1$ minimal

Eine greedy-Heuristik, die sog. Hyperebenenmethode von Lamport, berücksichtigt nur das erste und zweite Kriterium exakt und nähert das dritte an.

Die gesamte Matrix wird dann konstruiert, indem man sie unimodular ergänzt: trage den Vektor u als erste Zeile in eine $m \times m$-Matrix ein, streiche dann eine Spalte mit Eintrag eins in der ersten Zeile (existiert bei Lamport's Verfahren) und die erste Zeile selbst, und fülle den Rest mit der Einheitsmatrix der Dimension $m-1$ auf. Zusammen mit den gestrichenen Einträgen ergibt sich dann eine unimodulare Matrix (Determinante ist ± 1).

10.3.2 Parallelisierung äußerer Schleifen

Wenn ρ der Zeilenrang der Abhängigkeitsmatrix ist, dann kann man nur $m-\rho$ viele äußere Schleifen in parallele Schleifen umwandeln. Grund: da äußere Parallelschleifen keine Abhängigkeit tragen können, dürfen alle transformierten Abhängigkeitsvektoren in den ersten Zeilen nur Null-Einträge haben, d.h., $T * \mathcal{D}$ muß mit $m-\rho$ Nullzeilen beginnen. Da der Rang von $\mathcal{D}$ wegen der Unimodularität von T gleich dem Rang von $T * \mathcal{D}$ ist, muß $\mathcal{D}$ schon $m-\rho$ linear abhängige Zeilen besitzen.
Es ist dann allerdings möglich, alle Abhängigkeiten von der $m-\rho+1$-ten Schleife tragen zu lassen, so daß wieder nur eine einzige sequentielle Schleife im Zielprogramm nötig ist.

10.3.3 Asynchronität

Die soeben gemachte Feststellung scheint im Widerspruch zu der Aussage zu stehen, daß man durch Wahl der Zielschleifenreihenfolge im Space-Time-Mapping-Ansatz synchrone oder asynchrone (also jederzeit auch $m-1$ äußere Parallelschleifen!) erreichen kann. Auflösung: beim Space-Time-Mapping-Ansatz ist Kommunikation und/oder Synchronisation von asynchron parallel arbeitenden Prozessoren erlaubt ("DOACROSS"), bei allen anderen vorgestellten Methoden arbeiten parallele Prozessoren stets unabhängig voneinander ("DOALL").

10.4 Loop Distribution

Idee: *Man gibt bei Bedarf jedem Statement seine eigenen Schleifen.*
Loop Distribution arbeitet als einziges der hier vorgestellten Verfahren mit dem Statement-Abhängigkeitsgraphen. Im Gegensatz zu den unimodularen Transformationen (aber ähnlich dem Iteration Graph Partitioning) beinhaltet die Methode bereits die Code-Generierung. Diese Methode arbeitet sehr effizient und liefert gute Ergebnisse.
Vorgehen (rekursiv über die Schleifendimensionen):

1. Berechne den Statement-Abhängigkeitsgraphen.
2. Berechne die *azyklische Kondensation*, d.h. die starken Zusammenhangskomponenten und die darauf geltende Halbordnung.
3. Für jede starke Zusammenhangskomponente baue eine neue Schleife: sequentiell, falls eine Abhängigkeit existiert, parallel sonst.

4. Diese Schleifen werden gemäß der Halbordnung auf den Zusammenhangskomponenten sequentiell komponiert.
5. Entferne alle bereits berücksichtigten (von den soeben eingeführten sequentiellen Schleifen getragenen) Abhängigkeiten und fahre mit dem Restgraphen (und damit der nächst-inneren Dimension) bei 2. fort.

10.5 Polyedermodell

Das Polyedermodell stellt eine Vereinigung aller Verfahren innerhalb eines mathematischen Frameworks dar. Etwas genauer: das Polyedermodell ist eine Erweiterung des ursprünglichen Polytopmodells (des Space-Time-Mapping-Ansatzes), allerdings mit stark erweiterter Anwendbarkeit, z.B. nicht-perfekte Verschachtelung, separate Transformationen je Statement, die stückweise linear sein können, allgemeine for- und while-Schleifen, if-Anweisungen, affine Abhängigkeiten, Abschätzung der Abhängigkeiten bei nicht-affinen Array-Indizes (s. Kapitel 2). Der wesentliche Nachteil, den man sich durch diese Flexibilität erkauft, ist eine aufwendige Zielcodegenerierung (aufwendig sowohl zur Übersetzungs-, als auch zur Laufzeit).