UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Intergration der Auswertung statischer IF-Statements in die Datenabhängigkeitsanalyse im Rahmen des LooPo-Projektes

Programmierpraktikum

Verfasser: Harald Keimer

Aufgabensteller:
Prof. Christian Lengauer Ph. D.
Lehrstuhl für Informatik
Universität Passau

Passau, den 16. Juli 1997

Inhaltsverzeichnis

1 Einführung

Im Rahmen des LooPo-Projektes [EFG$^+$96] sind zwei Methoden zur Datenabhängigkeitsanaylse implementiert worden. Hierbei handelt es sich um den verallgemeinerten Ansatz von Banerjee [Ban93, Ban96, Kei97] sowie das Verfahren von Feautrier [Fea91, Kub97].

Da ein hohes Maß an Parallelität nur durch möglichst wenige Abhängigkeiten erreicht werden kann, kommt der Genauigkeit der Analyse eine große Bedeutung zu. In diesem Zusammenhang ist die Erweiterung der bestehenden Implementation um die Auswertung der Bedingungen von IF-statements erstrebenswert, weil dadurch eine deutliche Präzisierung der Abhängigkeiten erreicht werden kann.

Die Implementation dieser Erweiterung ist Gegenstand dieser Arbeit.

2 Bezeichnungen und Notation

In dieser Arbeit wird die Kenntnis der in der Datenabhängigkeitsanalyse üblichen Definitionen und Bezeichnungen vorausgesetzt. Die Notation richtet sich maßgeblich nach [Ban93, Fea91, Kei97].

Die Namen von Arrays erscheinen in Serifen, die von Funktionen werden kursiv wiedergegeben.

3 Darstellung des Problems

Einen der schwerwiegensten Einschnitte stellt der Verlust der Konvexität dar, der bei Betrachtung der durch IF-statements beschriebenen Iterationsräume auftritt (vgl. [Kei97], Abschnitt 5.2.3).

Ein einfaches Beispiel gibt das Programm aus Abbildung 1. Offensichtlich ist der Iterationsraum von $S2$ nicht konvex. Jedoch setzt die gesamte Theorie des Polyedermodells genau in diesem Punkt Konvexität voraus. Folglich müssen alle auftretenden Räume nötigenfalls in konvexe Teilräume zerlegt werden. Der Iterationsraum eines jeden Statements setzt sich dann aus einer Menge konvexer Teilräume zusammen.

Der naheliegenste Lösungsansatz besteht folglich darin, eine Umstellung in der Betrachtungsweise solcher Iterationsräume in der Weise vorzunehmen,

```
DO i = 1, 6
    DO j = 1, 6
        IF (i ≥ 3) ∧ (j ≥ 3) THEN
            {S1}
        ELSE
            {S2}
        END IF
    END DO
END DO
```

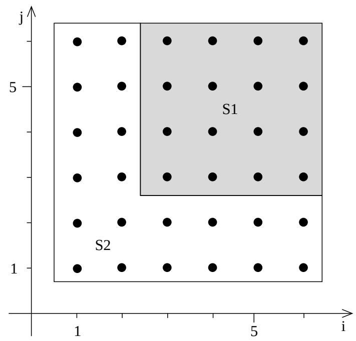


Abbildung 1: Beispiel für nicht-konvexe Iterationsräume

daß auf Mengen von Polyedern anstatt wie bisher auf einzelnen Polyedern gearbeitet wird. Allerdings würde sich eine solche Umstellung auf außerordentlich viele Module auswirken.

Betroffen wären:

- die Abhängigkeitsanalyse nach Banerjee (Modul <dependences>),
- die Abhängigkeitsanalyse nach Feautrier (Modul <feautrier>),
- die Singleassignmentkonversion (Module <singassgn>, [Kub97]),
- der Scheduler nach Feautrier (Module <scheduler>, [Wie95]),
- der Scheduler nach Darte-Vivien (Module <dvscheduler>, [Mei97]),
- der Allocator nach Dion-Robert (Module <drallocator>, [Mei97]),
- das Modul zum Filtern von Abhängigkeiten vor der grafischen Darstellung (Module <filter>, [Ell96, Wüs97]),
- die grafische Darstellung (Module <dispo2>,[Wüs95]) sowie
- die Zielcodeausgabe (Module <targout>, [Fab95]).

Wegen des sehr hohen zu erwartenden Aufwandes soll statt dessen eine Lösung gefunden werden, die Änderungen auf möglichst wenigen Modulen zur Folge hat.

Es wird sich zeigen, daß die alleinige Modifikation des <dependences>-Modules ausreichend ist.

Der Umfang der Eingabesprache des <dependences>-Modules ist bereits um IF-statements (jedoch ohne Auswertung der Bedingungen) erweitert worden. Hingegen verfügt das <feautrier>-Modul derzeit nicht über die Fähigkeit, IF-statements zu verarbeiten. Jedoch besteht für dieses Modul die Möglichkeit, IF-statements nach dem Einlesen des Eingabeprogrammes durch ein gesondertes Modul (<normloops>, [Gei97]) in der Weise zu normieren, daß sie anschließend durch das <feautrier>-Modul wie gewöhnliche Zuweisungen behandelt werden können. Aus den zuvor in der Einführung genannten Gründen ist eine Auswertung der Bedingungen von IF-statements auch für das <feautrier>-Modul wünschenswert.

Infolgedessen ist die Auswertung der Bedingungen insoweit von einer speziellen Methode der Datenabhängigkeitsanalyse zu trennen, daß sie als eigenständiger Programmteil von beiden implementierten Ansätzen genutzt werden kann.

4 Ziel

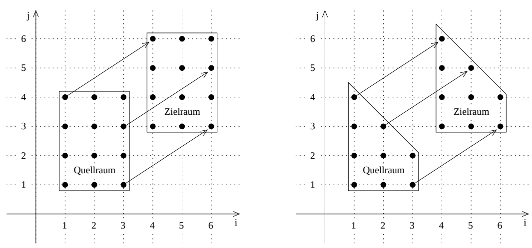
Das Ziel dieser Arbeit ist die Implementation der Auswertung von IF-Statements unter der Vorgabe, daß nur eine möglichst geringe Zahl von Modulen durch die vorzunehmenden Änderungen betroffen ist. Es sollen die vorhandenen Fähigkeiten des <dependences>-Moduls zur Verarbeitung von Kontrollflußstrukturen ausgebaut und um die Behandlung normalisierter IF-statements ergänzt werden. Die Auswertung ist so zu kapseln, daß eine Nachbearbeitung der Analyseergebnisse des <feautrier>-Modules (für normalisierte IF-statements) möglich ist.

5 Problemlösung

Die gesamte Implementation wird im <dependences>-Modul zu liegen kommen. Ein wesentlicher Grund für diese Wahl besteht in der Tatsache, daß IF-statements bei diesem Modul bereits zur Eingabesprache gehören und somit schon Fähigkeiten zum Erkennen und Verarbeiten von Kontrollflußstrukturen vorhanden sind.

```
DO i = 1, 6
     DO j = 1, 6
          IF (5 − j ≥ i) THEN
{S1}           X[i, j] = X[i–3, j–2]
          END IF
     END DO
END DO
```

(a) Eingabeprogramm



(b) ohne IF-Auswertung

(c) mit IF-Auswertung

Abbildung 2: Restriktion von Quell- und Zielraum

5.1 Idee

Die Umsetzung der Problemlösung erfolgt durch Ergänzung der Abhängigkeiten um die aus der Auswertung der Bedingungen gewonnene Information. Hierbei fließen die Bedingungen eines IF-Statements als zusätzliche Beschränkung in die Beschreibung des Quell- bzw. Zielraumes einer Abhängigkeit ein. Abbildung 2 gibt hierfür ein Beispiel. Zu dem Eingabeprogramm (a) liefert die Datenabhängigkeitsanalyse in der bisherigen Form die schleifenbasierte Flußabhängigkeit $S1\delta(3,2)S1$, deren Quell und Zielraum in (b) dargestellt werden. Durch Hinzufügen der Bedingung $5 - j \geq i$ des IF-Statements erhält man Beschränkungen dieser Räume, wie es (c) zeigt.

5.2 Strukturen, Funktionen, Arrays

5.2.1 Die Klasse PolyList

Zu Beginn erfolgt die Einführung einer Struktur zur Darstellung nicht-konvexer Iterationsräume. Wie zuvor in Abschnitt 3 ausgeführt, entspricht die Zerlegung solcher Räume in konvexe Teilräume einer Umstellung der bisherigen Betrachtungsweise von Iterationsräumen auf Mengen von Polyedern.

Für die Mengenimplementation bietet sich eine Listenstruktur an. Somit beschreiben Objekte der Klasse PolyList Elemente einer einfach azyklisch verketteten Liste von Polyedern.

Beispiel: Der durch das Programm in Abbildung 3 beschriebene Iterationsraum P des Statements $S2$, dargestellt in Abbildung 4(a), ist offensicht-

```
DO i = 1, 6
    DO j = 1, 6
        IF ((i ≤ 5 ∨ j ≥ 2) AND              {S1}
            (i ≤ 2 ∨ j ≤ 5)) THEN
            skip                              {S2}
        END IF
    END DO
END DO
```

Abbildung 3: Beispieleingabe zu Abschnitt 5.2.1

lich nicht konvex. Er kann jedoch durch die Menge $\{P1, P2, P3\}$ konvexer Teilräume ausgedrückt werden (vgl. Abbildung 4(b)).

Für einen Punkt x gilt also:

$$x \in P \quad \Longleftrightarrow \quad (x \in P1) \vee (x \in P2) \vee (x \in P3)$$

Eine Liste von Objekten der Klasse PolyList repräsentiert also einen Term in disjunktiver Normalform (DNF).

Es bietet sich daher an, als logische Basis die Funktionen AND und NOT zu implementieren.

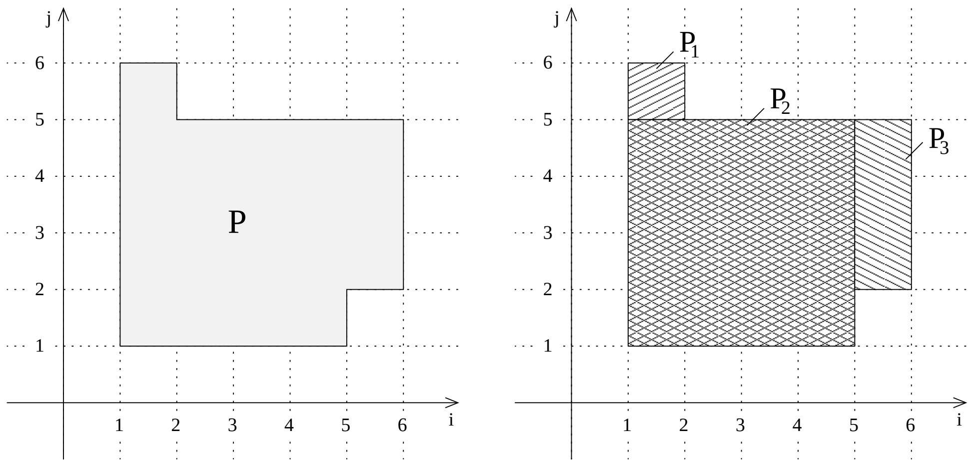
Die Klassendefinition sowie die Deklaration der wichtigsten Funktionen findet sich in Anhang A.

5.2.2 Bemerkung

Man beachte, daß die Zerlegung in Teilräume im allgemeinen nicht disjunkt ist.

Beispielsweise ergibt sich für das Eingabeprogramm des vorherigen Abschnittes (Abbildung 3):

$$\begin{array}{ll}
 & (i \leq 5 \vee j \geq 2) \wedge (i \leq 2 \vee j \leq 5) \\
\Leftrightarrow & \{ \text{ Distributivität } \} \\
 & ((i \leq 5) \wedge (i \leq 2 \vee j \leq 5)) \vee ((j \geq 2) \wedge (i \leq 2 \vee j \leq 5)) \\
\Leftrightarrow & \{ \text{ Distributivität } \} \\
 & (i \leq 5 \wedge i \leq 2) \vee (i \leq 5 \wedge j \leq 5) \vee (j \geq 2 \wedge i \leq 2) \vee (j \geq 2 \wedge j \leq 5) \\
\Leftrightarrow & \{ \text{ Absorbtion, Redundanz } \} \\
 & (i \leq 2) \vee (i \leq 5 \wedge j \leq 5) \vee (j \geq 2 \wedge j \leq 5)
\end{array}$$



(a) nicht-konvexer Iterationsraum

(b) Zerlegung in konvexe Teilräume

Abbildung 4: Iterationsraum zur Eingabe in Abb. 3

Offensichtlich sind diese Polytope nicht disjunkt.

Desweiteren kann eine Menge paarweise nicht disjunkter Räume auch durch Negieren einer Konjunktion entstehen. Dieses wurde in [Kei97], Abschnitt 5.2.3, ausführlich dargelegt.

Besteht beispielsweise die Ausführungsbedingung eines IF-Statements aus einer Konjunktion ($\wedge\ i\ :\ i\ :\ cond_i$), so wie es Abbildung 1 zeigt, so ist für den ELSE-Zweig ein im allgemeinen nicht konvexer Ausführungsraum zu erwarten.

Prinzipiell läßt sich ein solcher Raum zum Beispiel mit der *Polyhedral Library* ([Wil93]) in paarweise disjunkte Räume zerlegen. Jedoch haben praktische Anwendungen Instabilitäten dieser Bibliothek für den Fall gezeigt, daß der zu zerlegende Raum in einer oder mehreren Dimensionen kollabiert. Anschaulich bedeutet das, daß der Raum in der jeweiligen Dimension keine Ausdehnung besitzt und stellt einen Entartungsfall dar, bei dem zum Beispiel ein Dreieck der Höhe 0 zu einer Strecke wird. Geometrisch handelt es sich bei diesem Dreick also um ein eindimensionales Objekt, jedoch besitzt es rechnerintern eine Beschreibung als als zweidimensionales Gebilde.
Genau dieser Unterscheidung liegt der Bedeutungswandel der Defintion „Dimension eines Raumes“ zugrunde, wie er in [Kei97], Kapitel 3, beschrieben wird.
Auftreten können Räume mit in diesem Sinne kollabierender Dimension ins-

besondere bei der erweiterterten Datenabängigkeitsanalyse nach Banerjee. Hier kommt es im Zuge der Dimensionsangleichung (vgl. Programmtransformationen [Kei97], Abschnitte 6.2.1.2 und 6.2.1.4) zu einer Einbettung niederdimensionaler Indexräume in höherdimensionale, durch welche die sich ergebenden Räume per Konstruktion in mindestens einer Dimension kollabieren.

Aufgrund dieser Instabilität wurde bei der Berechnung der Negation einer Konjunktion auf die Verwendung der o.g. Bibliothek verzichtet. Statt dessen führt die Implementation über die Disjunktion $(\vee\ i\ :\ i\ :\ \neg cond_i)$. So ergibt sich aus dem Programm in Abbildung 1 für das Statement $S2$ als Ausführungsbedingung $(i < 3) \vee (j < 3)$ und damit eine offensichtlich nicht disjunkte Zelegung.

5.2.3 Konversionsfunktionen

Die Implementation des Polyedermodells im Rahmen des LooPo-Projektes basiert auf abgeschlossenen Halbräumen, dargestellt durch Ungleichungen der Form $\vec{a}x + \vec{b} \geq 0$, für einen Variablenvektor x.

Weil Bedingungen von IF-Statements eine Vielzahl anderer Vergleichoperatoren enthalten können ($i < 3 \wedge (j \neq 1 \vee n = 0)$) sind eine Reihe von Funktionen erforderlich, die (Un-)Gleichungen der Form

$$\vec{a}'x + \vec{b}'\ op\ 0, \qquad op \in \{<, \leq, >, =, \neq\}$$

in die oben angeführte Form überführen.

Die Deklaration der wichtigsten Funktionen findet sich in Anhang B.

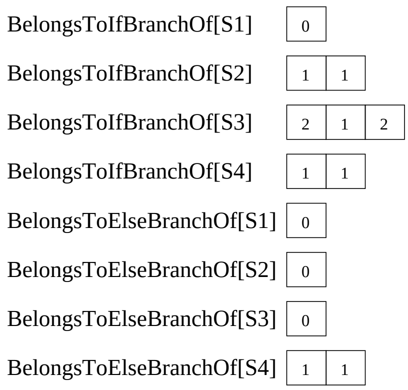
5.2.4 Arrays BelongsToIfBranchOf, BelongsToElseBranchOf

Die Auswertung der Bedingungen sollte möglichst unabhängig von der Frage erfolgen, ob das <normloops>-Modul zuvor gestartet wurde. Um ein einheitliches Vorgehen zu ermöglichen, muß folglich von dem konkreten Erscheinungbild eines solchen Statements, welches möglicherweise zu einer Zuweisung transformiert wurde (vgl. Abschnitt 3), abstrahiert werden.

Hierzu werden die beiden zweidimensionalen Arrays BelongsToIfBranchOf und BelongsToElseBranchOf eingeführt. Die Implementation erfolgt dynamisch als

IF ($cond_1$) **THEN** $\{S1\}$
    IF ($cond_2$)**THEN** $\{S2\}$
        skip $\{S3\}$
    ELSE
        skip $\{S4\}$
    END IF
END IF

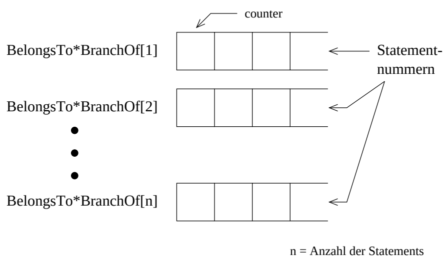
(a) Eingabeprogramm



(b) Arraybelegung

Abbildung 5: Arrays BelongsToIfBranchOf, BelongsToElseBranchOf

Vektor von Integervektoren. Dabei erhält jedes Statement des Eingabeprogrammes einen Vektor in jedem Array. Die Einträge bestehen aus den Nummern der dieses Statement umgebenden IF-Statements, wobei der erste Wert eines jeden Vektors dessen Länge angibt. Abbildung 5 gibt hierzu ein Beispiel. Für das Statement $S3$ des Eingabeprogrammes in (a) ergibt sich der Vektor BelongsToIfBranchOf[$S3$] $= [2, 1, 2]$. Er besagt, daß $S3$ zum IF-Zweig zweier Statements, nämlich $S1$ und $S2$, gehört.



5.2.5 Array IfCondition

Mit Hilfe der zuvor eingeführten Strukturen, Funktionen und Arrays lassen sich jetzt die Bedingungen eines IF-Statements ausdrücken. Dabei wird für jedes dieser Statements eine Liste aus Objekten der Klasse Polylist erzeugt, die in der oben ausgeführten Weise die Bedingung in DNF enthält.

Anschließend wird diese Liste in das Array IfCondition (dynamisch als Vektor von Listen implentiert) eingetragen.

Folglich repräsentiert das Array IfCondition zum Ende der Auswertung die gesamte aus den Bedingungen gewonnene Information.

5.2.6 Array ExecutionSpace

Im folgenden Schritt kann nun der im allgemeinen nicht-konvexe Iterationsraum eines Statements bestimmt werden. Er ergibt sich durch logische Verknüpfung der zuvor bestimmten Bedingungen bzw. deren Negation.

Beispielsweise erhält man den Ausführungsraum des Statements $S4$ des Programms in Abbildung 5(a) aus der Konjunktion von $cond_1$ und $\mathrm{NOT}(Cond_2)$.

Als Ergebnis liefert die Verknüpfung wiederum einen Term in DNF. Dieser wird in das Array ExecutionSpace (ebenso wie das Array IfConditions dynamisch implementiert) eingetragen.

Damit steht nun die erforderliche Information zur Verfügung, um die Quell- und Zielräume der Abhängigkeiten in der zuvor beschriebenen Weise (vgl. Abschnitt 5.1) zu beschränken.

5.3 Ablauf

Die Analyse der IF-Statements beginnt mit einer Vorverarbeitung, in deren Verlauf die Vernestung dieser Statements innerhalb des Eingabeprogrammes in die Arrays BelongsToIfBranchOf und BelongsToElseBranchOf übertragen wird (vgl. 5.2.4). Dies geschieht in der Funktion *surrounding_if*, die noch vor der Abhängigkeitsanalyse aufgerufen wird.

Die eigentliche Auswertung der Bedingungen erfolgt unmittelbar nach der Berechnung der Abhängigkeiten. Hierdurch bietet sich der Vorteil, nicht zwischen benutzerdefinierten ([Gün96]) und berechneten Abhängigkeiten unterscheiden zu müssen, da bei der durchzuführenden Auswertung der Bedingungen alle Abhängigkeiten dieselbe Behandlung erfahren.

Abbildung 9 gibt eine Übersicht über den gesamten Ablauf, auch im Zusammenspiel mit dem <feautrier>-Modul. Vorläufig jedoch beschränkt sich diese Beschreibung auf den Fall, daß das <dependences>-Modul (Banerjees Ansatz) im common-Mode aufgerufen wird. Auf die übrigen Modi wird in Abschnitt 8 eingegangen.

Prinzipiell läßt sich die Auswertung in drei Schritte unterteilen (siehe Abbildungen 6 und 7). Im ersten Schritt erfolgt die Belegung des Arrays IfConditions mit den Bedingungen der IF-Statements in DNF (vgl. Abschnitt 5.2.5). Die hierbei verwendete Funktion *extract_conditions* ist schematisch in Abbildung 8 dargestellt. Im darauffolgenden Schritt werden der Ausführungsraum jedes Statements im Eingabeprogramm bestimmt und in das Array ExecutionSpace (vgl. Abschnitt 5.2.6) eingetragen. Im letzten Schritt werden die Quell- und Zielräume jeder Abhängigkeit auf die zuvor in Abschnitt 3 beschriebene Weise beschränkt. Hierbei wird für eine Abhängigkeit d zwischen dem Quellstatement ss und dem Zielstatement ds die Menge aller Kombinationen von Ausführungsteilräumen gebildet. Stellt etwa die Menge $\{p_1, p_2, p_3\}$ bzw. $\{q_1, q_2\}$ von Polyedern den Ausführungsraum des Quell- bzw. Zielstatements dar, so wird zunächst der Quellraum durch p_1 und der Zielraum durch q_1 beschränkt. Für die übrigen fünf Kombinationen wird dann je eine Kopie des (originalen!) d erzeugt und um die jeweilige Kombination von Bedingungen ergänzt.

6 Beispiel

In diesem Abschnitt wird die gesamte Auswertung für das Eingabeprogramm in Abbildung 10(a) beispielhaft durchgeführt.

Zu Beginn erfolgt durch die Funktion *surrounding_if* die Vorverarbeitung der Eingabe. Mit den hierbei gewonnenen Informationen über die Vernestung der IF-Statements werden die beiden Integer-Arrays BolongsToIfBranchOf und BelongsToElseBranchOf belegt. Abbildung 11 veraunschaulicht die Instanziierung dieser Arrays für das gegebene Beispiel.

Die Datenabhängigkeitsanalyse liefert offensichtlich eine direkte, schleifengetragene Output-Abhängigkeit $d_1 = S2\,\delta^o(2,1)\,S4$. Sie wird zusammen mit Quell- und Zielraum in Abbildung 10(b) dargestellt.

Im ersten Schritt der sich anschließenden Auswertung erfolgt die Belegung des Arrays IfCondition mit den Bedingungen der beiden IF-Statements:

IfCondition[S1]: $\boxed{i + j - 7 \geqq 0} \longrightarrow \boxed{-i - j + 5 \geqq 0}$

IfCondition[S3]: $\boxed{i - 3 \geqq 0} \longrightarrow \boxed{j - 6 \geqq 0}$

Danach wird die Verknüpfung dieser Bedingungen entsprechend der Vernestung der IF-Statements durchgeführt. Hierdurch finden sich die Ausfüh-

Eingabe: Eingabeprogrammm (vorverarbeitet durch *surrounding_if*)
Liste aller Abhängigkeiten

Ausgabe: Liste mit Abhängigkeiten, deren Quell- und Zielräume entsprechend den Ausführungsbedingungen der IF-Statements beschränkt wurden

```
PolyList condition, exec_space;
Boolean replace_done;

/*                                                              */
/* Schritt 1: Extraktion der Bedingungen                        */
/*                                                              */
for each IF-Statement <is> do
        condition = extract_conditions(<is>);
        IfCondition[<is>] = condition;
od

/*                                                              */
/* Schritt 2: Generieren der Ausführungsräume                   */
/*                                                              */
for each Statement <s> do
        exec_space = index_space(<s>);
        for each IF-Statement <is> do
                if (<s> belongs to IF-branch of <is>) then
                   exec_space = AND(exec_space, IfCondition[<is>]);
                fi
                if (<s> belongs to ELSE-branch of <is>) then
                   exec_space = AND(exec_space, NOT(IfCondition[<is>]));
                fi
        od
        ExecutionSpace[<s>] = exec_space;
od
```

Abbildung 6: Ablauf der Funktion *analyse_static_if*, Schritt 1 und 2

```
Polytope p1, p2,
Dependence d_orig, d_new;

/*                                                                        */
/* Schritt 3: Restriktion der Abhängigkeiten auf Ausführungsräume        */
/*                                                                        */
for each dependence <d> do
        ss = source_statement(<d>);
        ds = destination_statement(<d>);
        replace_done = FALSE;
        d_orig = copy(<d>);
        for each polytope <p1> in ExecutionSpace[ss] do
                for each polytope <p2> in ExecutionSpace[ds] do
                        if (replace_done = FALSE) then
                          source_space(<d>) =
                                intersect(source_space(<d>), <p1>);
                          destination_space(<d>) =
                                intersect(destination_space(<d>), <p2>);
                          replace_done = TRUE;
                        else
                          d_new = copy(d_orig);
                          source_space(d_new) =
                                intersect(source_space(d_new), <p1>);
                          destination_space(d_new) =
                                intersect(destination_space(d_new), <p2>);
                          append(d_new, DependenceList);
                        fi
                od
        od
od
```

Abbildung 7: Ablauf der Funktion *analyse_static_if*, Schritt 3

Eingabe: Knoten n des Teilbaumes, welcher die auszuwertende Bedingung repräsentiert

Ausgabe: Liste von Polyedern, deren Vereinigung den durch die Bedingung beschriebenen Raum dargestellt

```
PolyList left, right, result;
int op = n.get_operand();

/*                                                                        */
/* Fall 1: logische Funktion                                              */
/*                                                                        */
IF  ( op ∈ {AND, OR, NOT}) THEN
     left = extract_conditions (n.get_left_operand);
     right = extract_conditions (n.get_right_operand);
     result = apply_operator(op,left,right);
END IF

/*                                                                        */
/* Fall 2: Vergleichsoperator                                             */
/*                                                                        */
IF  ( op ∈ {≤, <, >, =, ≠}) THEN
     left = eval_left_operand(n);
     right = eval_right_operand(n);
     result = <op>2Geq(op,left,right);   /* <op>2Geq: geeignete Konversions- */
                                         /*           funktion (vgl. 5.2.3)   */
END IF

/*                                                                        */
/* Fall 3: Annahme einer impliziten Ungleichung                           */
/*        z.B.:  IF ( i − j ) THEN ...    ⟹    IF ( i − j ≠ 0 ) THEN ...  */
/*                                                                        */
IF  ( op ∉ {AND, OR, NOT, ≤, <, >, =, ≠}) THEN
     left = eval_left_operand(n);
     right = 0;
     result = Neq2Geq(op,left,right);
END IF

try_evaluation(result);                  /* zusätzliche Tests; siehe Abschnitt 7 */

return result;
```

Abbildung 8: Ablauf der Funktion *extract_conditions*

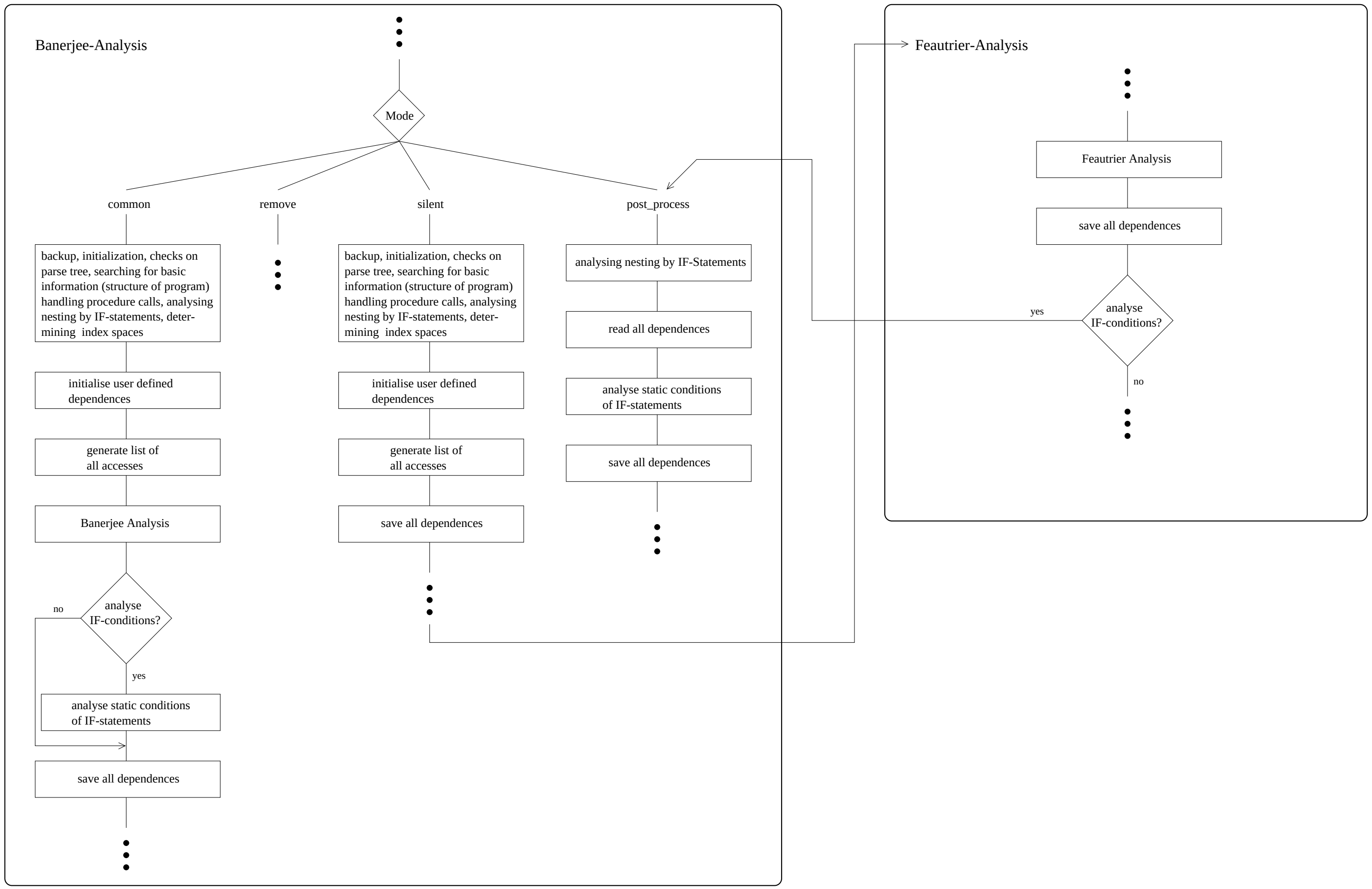
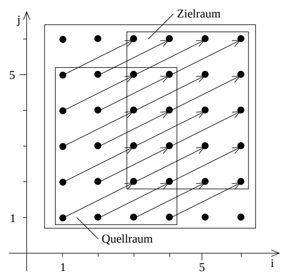


Abbildung 9: Ablaufschema der erweiterten Datenabhängigkeitsanalyse

```
DO i = 1, 6
    DO j = 1, 6
        IF (i + j ≠ 6) THEN                {S1}
            Y[i, j] =...                   {S2}
        ELSE
            IF (i ≥ 3 ∧ j > 5) THEN        {S3}
                Y[i − 2, j − 1] = ...      {S4}
            END IF
        END IF
    END DO
END DO
```

(a) Eingabeprogramm



(b) Outputabhängigkeit

Abbildung 10: Beispieleingabe

BelongsToIfBranchOf[S1]	0	BelongsToElseBranchOf[S1]	0
BelongsToIfBranchOf[S2]	1 1	BelongsToElseBranchOf[S2]	0
BelongsToIfBranchOf[S3]	0	BelongsToElseBranchOf[S3]	1 1
BelongsToIfBranchOf[S4]	1 3	BelongsToElseBranchOf[S4]	1 1

Abbildung 11: Belegung von BelongsToIfBranchOf, BelongsToElseBranchOf

rungsräume jedes Statements im Eingabeprogramm, welche dann in das Array ExecutionSpace eingetragen werden:

ExecutionSpace[S1]: $\dashv$

ExecutionSpace[S2]: $\boxed{i + j - 7 \geqq 0} \rightarrow \boxed{-i - j + 5 \geqq 0}$

ExecutionSpace[S3]: $\boxed{\begin{array}{r} i + j - 6 \geqq 0 \\ -i - j + 6 \geqq 0 \end{array}}$

ExecutionSpace[S4]: $\boxed{\begin{array}{r} i + j - 6 \geqq 0 \\ -i - j + 6 \geqq 0 \\ i - 3 \geqq 0 \end{array}} \rightarrow \boxed{\begin{array}{r} i + j - 6 \geqq 0 \\ -i - j + 6 \geqq 0 \\ j - 6 \geqq 0 \end{array}}$

Im letzten Schritt wird der Quell- und Zielraum der Abhängigkeit d_1 beschränkt. Da sowohl der Ausführungsraum von $S2$, als auch derjenige von $S4$ aus je zwei Teilräumen bestehen, entstehen die folgenden vier Abhängigkeiten:

d_1	
Quellraum:	$i - 1 \geq 0$
	$-i + 4 \geq 0$
	$j - 1 \geq 0$
	$-j + 5 \geq 0$
	$i + j - 7 \geq 0$
Zielraum:	$i - 3 \geq 0$
	$-i + 6 \geq 0$
	$j - 2 \geq 0$
	$-j + 6 \geq 0$
	$i + j - 6 \geq 0$
	$-i - j + 6 \geq 0$

d_2	
Quellraum:	$i - 1 \geq 0$
	$-i + 4 \geq 0$
	$j - 1 \geq 0$
	$-j + 5 \geq 0$
	$i + j - 7 \geq 0$
Zielraum:	$i - 3 \geq 0$
	$-i + 6 \geq 0$
	$j - 2 \geq 0$
	$-j + 6 \geq 0$
	$i + j - 6 \geq 0$
	$-i - j + 6 \geq 0$
	$j - 6 \geq 0$

d_3	
Quellraum:	$i - 1 \geq 0$
	$-i + 4 \geq 0$
	$j - 1 \geq 0$
	$-j + 5 \geq 0$
	$-i - j + 5 \geq 0$
Zielraum:	$i - 3 \geq 0$
	$-i + 6 \geq 0$
	$j - 2 \geq 0$
	$-j + 6 \geq 0$
	$i + j - 6 \geq 0$
	$-i - j + 6 \geq 0$

d_4	
Quellraum:	$i - 1 \geq 0$
	$-i + 4 \geq 0$
	$j - 1 \geq 0$
	$-j + 5 \geq 0$
	$-i - j + 5 \geq 0$
Zielraum:	$i - 3 \geq 0$
	$-i + 6 \geq 0$
	$j - 2 \geq 0$
	$-j + 6 \geq 0$
	$i + j - 6 \geq 0$
	$-i - j + 6 \geq 0$
	$j - 6 \geq 0$

Offensichtlich enthalten die Zielraumbeschreibungen der Abhängigkeiten d_2 und d_4 Widersprüche, so daß diese Abhängigkeiten eliminiert werden können. Ferner überlegt man sich leicht, daß d_1 ausschließlich aus lexikografisch

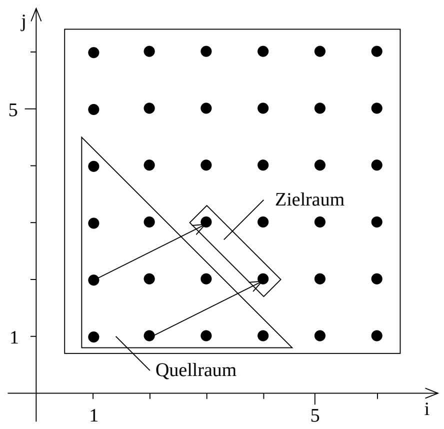


Abbildung 12: Einschränkung der Outputabhängigkeit

negativen Distanzvektoren besteht und damit keine zulässige Abhängigkeit beschreibt.

Folglich findet sich als Endergebnis die Abbhängigkeit d_3, die in Abbildung 12 dargestellt wird.

Man beachte, daß allein aus der durch diese Analyse gewonnenen Information eine weitere Einschränkung der Quellraumes nicht möglich ist. Diese erfolgt durch eine gesonderte Analyse zur Raumbegrenzung in Anlehnung an [Kei97], Abschnitt 6.2.6.

7 Zusätzliche Tests

Im Rahmen der Implementation ist die Auswertung der Bedingungen um einige Tests erweitert worden.

Ziel dieser Tests ist die Bestimmung der maximalen Terme innerhalb einer Bedingung, die konstant TRUE oder FALSE liefern. Die Ergebnisse sollen den Benutzer in der Weise unterstützen, als ihm Möglichkeiten zur Programmvereinfachung aufgezeigt werden. So können beispielsweise im Rahmen einer dead-code-Elimination solche Programmteile entfernt werden, deren Ausführungsbedingungen widersprüchlich sind. Darüberhinaus können Bedingungen, die zu TRUE oder FALSE äquivalente Terme enthalten, vereinfacht werden.

Basierend auf der Darstellung eines aus n Klauseln bestehenden Terms T

$$T := \ (\vee i : 1 \leq i \leq n : t_i)$$

in DNF gilt:

$$T \equiv \text{FALSE} \quad \Longleftrightarrow \quad (\forall\, i \,:\, 1 \leq i \leq n \,:\, t_i \equiv \text{FALSE}) \tag{7.0.1}$$

Ist T Teil einer Bedingung des Statements S, so bezeichne D_S die Menge von Bedingungen, welche den Indexraum von S beschreiben. Es gilt:

$$T \equiv \text{TRUE} \quad \Longleftrightarrow \quad (\exists\, i \,:\, 1 \leq i \leq n \,:\, (t_i \wedge D_S) \equiv D_S) \tag{7.0.2}$$

Die Äquivalenz (7.0.1) ist gleichbedeutend mit der Tatsache, daß der Ausführungsraum von S unter der Bedingung T leer ist. Hingegen sagt (7.0.2) aus, daß sich durch die Hinzunahme der Bedingung T keine Einschränkung für den Indexraum von S ergibt.

Enthalte ein Eingabeprogramm beispielsweise ein IF-Statement mit dem folgenden Term T als Bedingung

$$T := \ (A \vee B) \wedge C$$

wobei für A und C angenommen werde, daß sie äquivalent zu TRUE sind.

Die interne Repräsentation beschreibt T in DNF:

$$T := \ \underbrace{(A \wedge C)}_{=:\, t_1} \vee \underbrace{(B \wedge C)}_{=:\, t_2}$$

Gemäß der Annahme gilt:

$$t_1 \ \equiv \ \text{TRUE}$$

Durch Anwendung von (7.0.2) ergibt sich:

$$T \ \equiv \ \text{TRUE}$$

Folglich ist T selbst der maximale, zu TRUE äquivalente Term in der Bedingung des IF-Statements. Damit wird stets der IF-Zweig dieses Statements ausgeführt und es können sowohl der ELSE-Zweig als auch das IF-Statement selbst entfernt werden.

Die soeben beschriebene Auswertung wird durch die Funktion *try_evaluation* übernommen, welche von *extract_conditions* (vgl. Abbildung 8) aufgerufen wird.

8 Zusammenspiel mit anderen Modulen

Eine wesentliche Anforderung an diese Arbeit bestand in der Kapselung der Auswertung, sodaß diese von verschiedenen Modulen genutzt werden kann (vgl. Abschnitt 4).

Hierzu ist ein spezieller Modus (post_process) eingeführt worden (siehe Abbildung 9), der die Nachbearbeitung der Analyseergebnisse eines anderen Modules erlaubt.

Die Vorgehensweise bleibt nahezu unverändert: zu Beginn erfolgt, wie in Abschnitt 5.3 beschrieben, eine Vorverarbeitung des Eingabeprogrammes. Anstelle der nachfolgenden Abhängigkeitsanalyse (Banerjee), werden die Ergebnisse eines anderen Modules (z.B. Feautrier) eingelesen. Danach erfolgt in gewohnter Weise eine Beschränkung der Abhängigkeiten auf die Ausführungsräume.

Durch diesen Modus ist folglich eine Schnittstelle zur Nachbearbeitung der Ergebnisse eines beliebigen Verfahrens zur Datenabhängigkeitsanalyse geschaffen worden.

9 Zusammenfassung

Mit der vorliegenden Arbeit konnten alle an die Implementation gestellten Anforderungen erfüllt werden. Die vorzunehmenden Änderungen beschränken sich auf ein Modul, jedoch ist auch die Nachbearbeitung der Analyseergebnisse anderer Module durch die Schaffung eines ensprechenden Modus' ermöglicht worden. Die Entscheidung für die Beschränkung der Quell- und Zielräume sowie der gewählte Ablauf der Auswertung erlauben es, benutzerdefinierte Abhängigkeiten in der gleichen Weise zu behandeln, wie die durch die Datenabhängigkeitsanalyse berechneten. Die enstehenden Abhängigkeiten sind auf deutlich kleineren Räumen definiert, jedoch wirkt sich hier die stark steigende Zahl der Abhängigkeiten nachteilig auf die Laufzeit des Parallelisierers aus. Auch setzt die Beschränkung von Quell- und Zielraum anstelle des Indexraumes die Fähigkeit des Schedulers voraus, stückweise affine Schedules zu erzeugen (z.B. Feautrier). Aus diesem Grund ist bei der Verwendung des Lamport-Schedulers keine Verbesserung durch die Auswertung der IF-Statements zu erwarten.

Anhang A: PolyList

Die Klassendefinition findet sich in der Datei dependences/poly_list.h.

```
class PolyList
{
        protected:

          Ineq_system*     polyhedron;
          PolyList*        next;

        public:

          // constructors
          PolyList( Ineq_system* );
          PolyList( const PolyList* ,const BOOLEAN = SHALLOW );

          // destructors
          ~PolyList( void );

          // "get"-routines
          Ineq_system* get_polyhedron( void ) const;
          PolyList*    get_next( void ) const;

          // manipulating-routines
          PolyList* append( PolyList* );

          // "info"-routines
          void print( void ) const;
          int length( void ) const;
};

PolyList* and( PolyList*&, PolyList*& );
void      delete_poly_list( PolyList*& );
PolyList* negate( PolyList* );
```

Anhang B: mat_ineq

Die Deklarationen der wichtigsten Funktionen befinden sich in der Datei dependences/mat_ineq.h.

```
#ifndef DEPENDENCES_MAT_INEQ_H

  #define DEPENDENCES_MAT_INEQ_H

  // C/C++ header files
  #include <assert.h>

  // header files of other modules
  #include "lmaths.h"


  void   Ge2Geq( Matrix& );
  void   Le2Leq( Matrix& );
  void   Le2Geq( Matrix& );
  void   Leq2Geq( Matrix& );
  void   Eq2Geq( Matrix& );
  void   NotGeq( Matrix& );
  void   NotGe( Matrix& );
  void   NotLeq( Matrix& );
  void   NotLe( Matrix& );

#endif
```

Anhang C: Abbildungsverzeichnis

Literatur

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1993.

[Ban96] Utpal Banerjee. *Dependence Analysis.* Kluwer Academic Publishers, Boston/Dordrecht/London, 1996.

[EFG+96] Nils Ellmenreich, Peter Faber, Martin Griebl, Robert Günz, Harald Keimer, Wolfgang Meisl, Sabine Wetzel, Alexander Wüst, and Christian Wieninger. Loopo, Loop Parallelization in the Polytope Model. Technical report, Universität Passau, 1996.

[Ell96] Nils Ellmenreich. Das Frontend zu Loopo. Technical report, Universität Passau, 1996.

[Fab95] Peter Faber. Loopo Target Output. Technical report, Universität Passau, 1995.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20:23–53, 1991.

[Gei97] Max Geigl. Parallelization of Loop Nests with General Bounds in the Polyhedron Model. Master's thesis, Universität Passau, 1997.

[Gün96] Robert Günz. The new Loopo Scanner and Parser. Technical report, Universität Passau, 1996.

[Kei97] Harald Keimer. Datenabhängigkeitsanalyse zur Schleifenparallelisierung: Vergleich und Erweiterung zweier Ansätze. Master's thesis, Universität Passau, 1997.

[Kub97] Radko Kubias. Array-Datenflußanalyse & Single-Assignment-Konversion. Master's thesis, Universität Passau, 1997.

[Mei97] Wolfgang Meisl. Practical Methods for Scheduling and Allocation in the Polytope Model. Master's thesis, Universität Passau, 1997.

[Wie95] Christian Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Master's thesis, Universität Passau, 1995.

[Wil93] D. K. Wilde. *A library for doing polyhedral operations.* Publication interne 785, Institute de Recherche en Informatique et Systemès Aléatoires, Rennes, December 1993.

[Wüs95] Alexander Wüst. Dispo2, Grafische Darstellung der Indexräume vernesteter Schleifen. Technical report, Universität Passau, 1995.

[Wüs97] Alexander Wüst. Grafische Darstellung des Indexraumes von Anweisungen in Schleifenprogrammen. Master's thesis, Universität Passau, 1997.