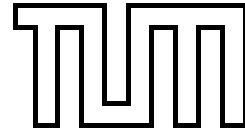


TECHNISCHE UNIVERSITÄT MÜNCHEN  
FAKULTÄT FÜR INFORMATIK



Diplomarbeit

# **Interprozedurale Analyse linearer Ungleichungen**

Andrea Flexeder

27.Juli 2005

Aufgabensteller: Prof. Dr. Helmut Seidl

Betreuer: Dipl.-Inf. Michael Petter

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst, und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Garching, den \_\_\_\_\_ Unterschrift: \_\_\_\_\_

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. Helmut Seidl für diese interessante Themenstellung und bei Michael Petter für die gute Betreuung bei der Bearbeitung meiner Diplomarbeit und die vielen wertvollen Hilfestellungen bedanken.  
Zudem gilt mein Dank meinen Eltern und meinem Freund Michael, welche mich während meines ganzen Studiums unterstützt haben.

# Inhaltsverzeichnis

<b>1</b>	<b>Abstrakt</b>	<b>V</b>
<b>2</b>	<b>Einleitung</b>	<b>1</b>
2.1	Gliederung . . . . .	1
2.2	Verwandte Ansätze . . . . .	2
<b>3</b>	<b>Zu analysierende Programmklasse</b>	<b>3</b>
3.1	Kontrollflussgraph . . . . .	3
3.2	Semantik des Kontrollflussgraphen . . . . .	7
3.3	Repräsentation des Kontrollflussgraphen . . . . .	9
3.4	Repräsentation eines Programmzustandes . . . . .	10
3.4.1	Konvexe Mengen . . . . .	10
3.4.2	Polyeder . . . . .	11
<b>4</b>	<b>Intraprozedurale Analyse mittels Polyedertheorie</b>	<b>15</b>
4.1	Bestimmung linearer Beziehungen zwischen Programmvariablen . . . . .	15
4.2	Repräsentation des Kontrollflussgraphen . . . . .	15
4.3	Abstrakte Interpretation . . . . .	16
4.4	Naiver Fixpunktalgorithmus . . . . .	18
4.4.1	Widening . . . . .	19
4.5	Implementation . . . . .	20
<b>5</b>	<b>Interprozedurale Analyse unter Verwendung von Linearer Algebra</b>	<b>23</b>
5.1	Abstrakte Semantik für die interprozedurale Analyse . . . . .	23
5.1.1	Abstraktion konkreter Programmzustände . . . . .	23
5.1.2	Berechnung der Effektmatrizen . . . . .	25
5.1.3	Berechnung der Zustandsmengen . . . . .	29
5.2	Inkrementeller Fixpunktalgorithmus . . . . .	32
5.2.1	Widening . . . . .	34
<b>6</b>	<b>Realisierung</b>	<b>37</b>
6.1	Implementation . . . . .	37
6.2	<i>convexHull()</i> . . . . .	40
6.3	<i>cut()</i> . . . . .	42
6.4	<i>append()</i> . . . . .	44

6.5	LP Problemstellungen . . . . .	47
6.5.1	Darstellbarkeit der Generatoren . . . . .	47
6.5.2	Lineare Abhängigkeit zweier Strahlen bei der Analyse von Bedingungen	48
6.5.3	Lineare Abhängigkeit bei Strahlen . . . . .	49
6.5.4	Lineare Abhängigkeit bei Punkten . . . . .	49
6.6	Leistungsanalyse . . . . .	50
6.6.1	Testfall: Schleife . . . . .	50
6.6.2	Testfall: Methodenaufruf . . . . .	51
6.6.3	Testfall: Kombination verschiedener Programmkonstrukte . . . . .	52
6.6.4	Anwendungsbeispiel . . . . .	55
6.6.5	Fazit . . . . .	56
6.7	Verbesserungsmöglichkeit . . . . .	57
6.8	Zusammenfassung . . . . .	58
<b>7</b>	<b>Hilfsmittel</b>	<b>59</b>
7.1	Parma Polyhedra Library . . . . .	59
7.2	Linear Programming . . . . .	59
7.2.1	Mathematische Definition . . . . .	59
7.2.2	Geometrische Interpretation . . . . .	61
7.2.3	GNU Linear Programming Kit . . . . .	62
7.3	polyinvar . . . . .	63
7.3.1	Architektonischer Aufbau . . . . .	63
7.3.2	Kontrollflussgraph . . . . .	64
<b>8</b>	<b>Zusammenfassung</b>	<b>65</b>
	<b>Abbildungsverzeichnis</b>	<b>67</b>
	<b>Tabellenverzeichnis</b>	<b>69</b>
	<b>Literaturverzeichnis</b>	<b>71</b>

# 1 Abstrakt

Diese Arbeit beschreibt eine intra- und auch interprozedurale Datenflussanalyse, welche an jedem Programmpunkt statisch die Beziehungen, die zwischen den Programmvariablen gelten, bestimmen können. Die intraprozeduralen Analyse, beruhend auf einem Modell von Cousot [CH78] interpretiert lineare Zuweisungen und Bedingungen und betrachtet die nicht linearen Konstrukte mit Hilfe von nicht linearen Zuweisungen. Mit dieser Abstraktion versucht man lineare Gleichheits- und Ungleichheitsbeziehungen zwischen den Programmvariablen in Form von Polyedern rauszufinden. Da man nicht nur eine Funktion, sondern ganze Programme als Zusammenspiel mehrerer Funktionen, analysieren möchte, ist eine interprozedurale Analyse nötig[MOS04]. Diese soll mit den Mitteln der linearen Algebra die affinen Beziehungen, welche zwischen den Programmvariablen an einem bestimmten Programmpunkt gelten, erkennen. Die Behandlung von Prozeduraufrufen steht dabei im Vordergrund.



## 2 Einleitung

Viele Bereiche im Compilerbau stützen sich auf die interprozeduralen Analysemethoden, um Informationen über die Werte der Variablen an den jeweiligen Programmpunkten zu erhalten. Die klassischen Datenflussprobleme können als Probleme über affinen Relationen gesehen werden [MOS04]. Das Ziel der vorliegenden interprozeduralen Analyse 5 ist es für jeden Programmpunkt alle gültigen affinen Beziehungen, die zwischen den Programmvariablen gelten, angeben zu können. Um Variablen oder Ausdrücke, die einen konstanten Wert haben, darzustellen, ist die affine Relation  $x = c$  zu formulieren, wohingegen sich ein symbolischer Wert zu  $x = x_1 + 7$  abbilden lässt.

Somit können auch affine Relationen angegeben werden, um Schleifeninduktionsvariablen oder äquivalente Ausdrücke, die syntaktisch verschieden sind, zu erkennen.

Affine Relationen sind auch im Gebiet der Programmverifikation sehr nützlich, da sie gültige Invarianten liefern, die nicht sofort aus dem Programm erkennbar sind. Ein Beispiel stellt die Erkennung von Schleifeninvarianten dar. Der Beweis der Korrektheit eines Programms kann so durch die gewonnenen affinen Relationen erheblich vereinfacht werden.

### 2.1 Gliederung

Nun ein kurzer Überblick über den Inhalt dieser Arbeit.

In Kapitel 3 soll die zu analysierende Programmklasse beschrieben werden, d.h. die theoretische Grundlage, um solche affinen Relationen anhand der Datenstruktur des Kontrollflussgraphen aufstellen und auch verifizieren zu können. Zudem werden in diesem Kapitel die konvexen Mengen und Polyeder beschrieben, mit Hilfe derer man die abstrahierte Darstellung eines Programmzustandes erhält.

Das Kapitel 4 widmet sich der intraprozeduralen Analyse, bei der die Abhängigkeiten zwischen den Programmvariablen mit Polyedern repräsentiert werden. Es stellt sich heraus, dass das beschriebene Verfahren für die intraprozedurale Analyse wenig effizient ist. Eine ständige Umwandlung der beiden Darstellungsarten eines Polyeders, was eine Durchführung der Analyse sehr teuer macht, und die nicht Beachtung von Prozeduraufrufen führt zu dem in Kapitel 5 eingeführten Analyseverfahren der interprozeduralen Analyse mittels affiner Relationen.

Eine experimentelle Implementation dieser theoretisch konzipierten interprozeduralen Analyse wird im darauffolgenden Kapitel 6 vorgenommen, wobei auf eine genaue Betrachtungsweise der essentiellsten Funktionen Wert gelegt wurde. Es werden ein paar durchgeführte Testfälle angeführt, um eine Einschätzung der Leistungsfähigkeit der Analyse zu erhalten. Anschliessend wird in 7 ein kurzer Überblick über die verwendeten Hilfsmittel gegeben, wobei die Bibliothek *PPL* zur Darstellung von Polyedern und das Programmanalysewerkzeug *poly-invar* zu erwähnen sind.

### 2.2 Verwandte Ansätze

Auf dem Gebiet der Programmanalyse gibt es viele theoretische Ansätze von denen die wenigsten praktisch implementiert sind. Da mir keine entsprechende Implementation eines Prototypen für die interprozedurale Analyse bekannt ist, konnten im Rahmen dieser Arbeit keine vergleichenden Bewertungen zu anderen Analysatoren aufgeführt werden.

Ein ähnlicher Ansatz wird in dem Analysewerkzeug **BRAIN** (Backward Reachability Analysis with INtegers) von Rybina und Voronkov [RV02], verfolgt, wobei hier jeder Programmzustand mit Hilfe eines Vektors über Integer-Werten beschrieben wird. **BRAIN** ist ein Model Checker für die Verifizierung von Erreichbarkeitseigenschaften. In der derzeitigen Fassung dieses Analysewerkzeugs ist die Technik des Widenings und der transitiven Hülle noch nicht integriert, so dass die Konvergenz dieser Analyse nicht immer gewährleistet werden kann, wie in [RV03] beschrieben.

Gulwani und Nacula beschreiben in [GN03] einen randomisierten Algorithmus für die intraprozedurale Analyse. Damit können zwar affine Relationen, die zwischen den Programmvariablen gelten, herausgefunden werden. Diese liegen jedoch nur in Form von linearen Gleichheitsbeziehungen vor, was für eine Erreichung eines präzisen Analyseergebnisses nicht förderlich ist.

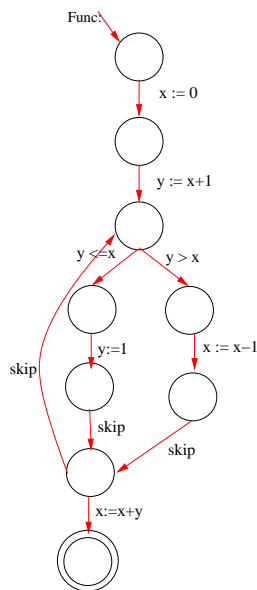
## 3 Zu analysierende Programmklasse

Anfänglich wollen wir die Programmklasse beschreiben, über welche unsere Analysen Aussagen über die zwischen den Programmvariablen gültigen Relationen liefern sollen. Die Programme, an denen die intra- und interprozedurale Analyse getestet wurde, sind in der Programmiersprache *Java* verfasst. Da Java einen grossen Sprachumfang aufweist, soll innerhalb der durchgeführten Analysen nur ein kleiner Teilbereich des gesamten Sprachbereichs abgedeckt werden. Für all die Konstrukte, die von unseren Analysen nicht behandelt werden können, wie zum Beispiel Felder oder Objekte, wird das Modell der nichtdeterministischen Zuweisungen eingesetzt. Wie aus den dargelegten Testfällen 6.6 ersichtlich ist, wurden vornehmlich die Zuweisungen von arithmetischen Ausdrücken an eine einzelne Variable betrachtet. Auch Ungleichheitsbedingungen können analysiert werden. Nur im Falle der interprozeduralen Analyse ist eine Untersuchung von Prozeduraufrufen möglich.

Zumal die Ausführungswege durch ein Programm mit abstrakten Syntaxbäumen nicht ausreichend modelliert werden, bedienen wir uns der Datenstruktur eines Kontrollflussgraphen, der als Grundlage unserer Analyse dient. In Abschnitt 3.1 wird das Prinzip eines Kontrollflussgraphen beschrieben, mit dessen Verwendung die Untersuchung der Relationen zwischen den Programmvariablen für die intraprozedurale und interprozedurale Analyse vorgenommen wird.

### 3.1 Kontrollflussgraph

Bei der Analyse von Programmen stützen wir uns auf Kontrollflussgraphen, deren Aufgabe darin besteht die Struktur des zu analysierenden Quellprogramms vereinfacht darzustellen. Daraus ergibt sich eine abstrakte Darstellung des vielschichtigen Programmcodes, wobei die Aspekte, die für unsere Analyse nicht informativ sind, weggelassen werden. Damit werden Programme, die oft relativ komplexe Ablaufstrukturen aufweisen, übersichtlich dargestellt und deren Semantik wird klar dargelegt. Diese abstrakte Datenstruktur ist ein gerichteter, teilweise zyklischer Graph. Jeder Knoten des Kontrollflussgraphen repräsentiert einen Zustand innerhalb des Programms und jede Kante, ein Kontrollfluss zwischen zwei Zuständen, stellt eine Programmanweisung dar. In unserer Programmanalyse versuchen wir Gleichungen aufzustellen, die an einem Zustand gelten und diesen Programmpunkt somit genau charakterisieren. Die Kanten werden mit unterschiedlichen Attributen versehen, je nach Abstraktion des zu Grunde liegenden Programmes.



Für die intraprozedurale Analyse besteht die Abstraktion in der Betrachtung deterministischer und nicht deterministischer Zuweisungen an eine einzelne Variable, wie auch bestimmte Bedingungen. Die in unserem zu analysierenden Programm vorkommenden Variablen, werden durch den Vektor  $\mathbf{x} = (x_1, \dots, x_k)$  definiert.

Der Wert dieses Vektors  $x = (x_1, \dots, x_k)$  charakterisiert den **Zustand** des Programms, an dem der Wert der Variablen verändert wird. Übergänge zwischen den betrachteten Zuständen bilden die gerichteten Kanten, die einen Zustand  $s$  in den Nachfolgezustand  $t$  überführen und mit der von der Abbildungsfunktion spezifizierten Beschriftung versehen sind.

Ein Kontrollflussgraph (CFG) wird durch ein 5-Tupel  $CFG = (N, E, A, s, r)$ , repräsentiert.

$N$ : Menge der Programmpunkte

$E \subseteq N \times N$ : Menge der gerichteten Kanten

$A : E \rightarrow Label$ : Abbildung, die jeder Kante eine Programmanweisung zuordnet

$s \in N$ : ein ausgezeichnete Startknoten

$r \in N$ : der Endknoten der Prozedur

Es wird genau eine Prozedur des Programms durch die Datenstruktur des Kontrollflussgraphen abstrahiert. Bei der intraprozedurale Analyse beschränkt man sich ausschliesslich auf die Untersuchung einer einzelnen Funktion. Das *Label*, mit der jede Kante annotiert ist, beschreibt die Menge der Syntaxbaumelemente der zu Grunde liegenden Sprache und setzt sich hierbei nur aus den Elementen von *Base* zusammen:

- deterministischen Zuweisungen,
- nicht deterministischen Zuweisungen,
- Bedingungen und

- skip - Anweisungen.

Die Funktionsaufrufe werden in dieser abstrahierten Darstellung der Programmstruktur ausgeblendet. Somit ergeben sich für die Abbildungsfunktion der intraprozeduralen Analyse eben erwähnte mögliche Beschriftungen:

$$\text{Label} \supseteq \text{Base} \mid \text{Base} := \{x_j := t; x_j := ?; \text{skip}; t \diamond 0; \mid \diamond \in \{\geq, \leq\}\} \subseteq A(e)$$

*Base* bezeichnet somit die Menge der Basiskanten, die Zuweisungen an eine einzelne Variable, skip-Anweisungen und nicht strikte Ungleichheitsbedingungen.

Seiteneffekte können nur durch eine Zuweisung entstehen, denn die Auswertung einer Bedingung zieht keine Veränderung der Werte der Programmvariablen nach sich.

- **Zuweisungen:**

diese werden in der Analyse nur dann betrachtet, wenn sie aus einem linearen Ausdruck bestehen, d.h. keine Methodenaufrufe oder Division bzw. Potenzierung einer Programmvariablen beinhalten. Sonst wird der Wert dieser Zuweisung auf unbekannt gesetzt.

**affine Zuweisungen :**

$$\mathbf{x}_j := t_0 + \sum_{i=1}^k t_i \mathbf{x}_i \quad t \in \mathbb{R}, \mathbf{x}_j \in \mathbf{x}$$

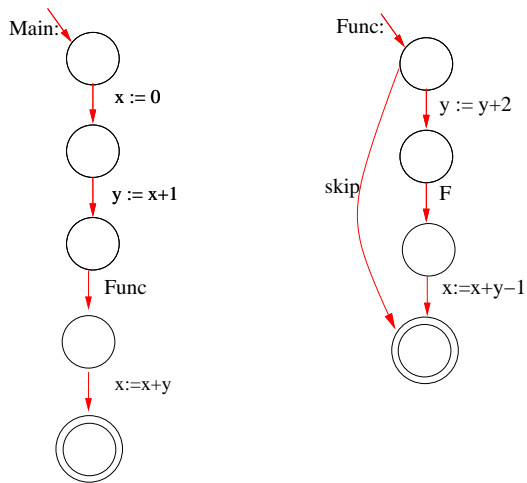
Die Zuweisungen  $\mathbf{x}_j := t$  werden in der Analyse nur dann betrachtet, wenn der zuzuweisende Term  $t$  linear ist, ansonsten wird der Wert der Zuweisung unbekannt d.h.  $\mathbf{x}_j := ?$ .

- **Wächter:**

können der Einfachheit halber nur in Form von nicht strikten Ungleichungen betrachtet werden. Strikte Ungleichungen kann man jedoch in äquivalente nicht strikte Ungleichungen transformieren und Gleichungen lassen sich durch zwei entgegengesetzte Ungleichheitsbeziehungen simulieren. Nur die negativen Gleichungen können nicht erfasst werden.

- **skip-Anweisungen:**

*skip*; findet dort Anwendung, wo Anweisungen innerhalb des zu analysierenden Programms, für die Analysezwecke nicht betrachtet werden sollen oder für die Repräsentierung von nicht linearen Bedingungen und Zuweisungen. Skip-Anweisungen wurden eingeführt, um die Programmanweisungen abstrakt darzustellen, die für unsere Analysezwecke uninteressant sind oder nicht von der Analyse fehlerfrei gehandhabt werden können.



Im Gegensatz dazu gibt es die interprozeduralen Analysen, die Prozeduraufufe innerhalb unseres abstrakten Modells berücksichtigen, um so ein ganzes Programm auszuwerten. Um die Effekte von Prozeduraufrufen auszuwerten, muss eine genauere Betrachtungsweise einer Prozedur erfolgen. Denn Prozeduren arbeiten sowohl auf **globalen**, wie auch **lokalen Variablen**. Es wird davon ausgegangen, dass alle Prozeduren die selbe Anzahl  $k$  an globalen Variablen, aber jede Prozedur ihre eigene Menge von lokalen Variablen führt.

Wie bereits in 3.1 deklariert, haben wir einen Variablenvektor  $\mathbf{x}$  für eine Prozedur, wobei wir die-

sen für die interprozedurale Analyse präzisieren müssen, in Anbetracht der Tatsache, dass wir zwischen globalen und lokalen Variablen unterscheiden. Der differenzierende Vektor

$$\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_{n_p})^T$$

bezeichnet die in einer Methode  $p$  vorkommenden Variablen, wobei die ersten  $k$  global und die restlichen  $n_p - k$  lokaler Natur sind.  $n_p$  soll andeuten, dass jede Prozedur  $p \in Proc$  eine andere Anzahl lokaler Variablen besitzt.

Es wird bei der interprozeduralen Analyse ein weiteres Sprachkonstrukt unterstützt, nämlich das der Prozeduraufufe.

- **Call:**

$$Call_p = \{e | A(e) \equiv p() \in Proc\}$$

bezeichnet die Kantenart, die einen Methodenaufuf verkörpert.

$Proc$  bezeichnet eine endliche Menge von Methodennamen  $p_i$ , die innerhalb des zu analysierenden Programms vorkommen können.

$$Proc = \bigcup p_i$$

Wie in der formalen Definition des Kontrollflussgraphen 3.1 beschrieben, wird eine Prozedur  $p \in Proc$  genau durch einen einzelnen Kontrollflussgraphen modelliert.

Eine mögliche Vorgehensweise bei interprozeduralen Analysen ist es die Kontrollflussgraphen, die ja eine abstrakte Darstellung einer einzelnen Prozedur bedeuten, miteinander zu verbinden. Dabei wird jeder Funktion ein übergeordneter Eintrittsknoten zugeordnet und die Parameter einer Methode werden durch neue Knoten dargestellt. Findet ein Methodenaufuf statt, so wird jedem aktuellen Parameter des Aufrufs der entsprechende formale Parameter,

mit welchem die Funktion letztendlich rechnet, durch Einführung von neuen Knoten und Kanten zugewiesen. Auch die entgegengesetzte Richtung, wo jeder formale Parameter wieder auf den aktuellen Parameter übertragen werden muss, wird mit Hilfe von weiteren Knoten und Kanten modelliert. Die Bedeutung erklärt sich darin, dass für jeden Funktionsaufruf der richtige Kontext bewahrt werden muss. Jede aufgerufenen Funktion existiert genau ein Mal im interprozeduralen Kontrollflussgraph. Insgesamt erhält man einen grossen zusammenhängenden Kontrollflussgraph, bei dem die einzelnen Kontrollflussgraphen für jede Funktion direkt miteinander verbunden werden.

Bei der zweiten Möglichkeit wird die Analyse auf einem System von nicht zusammenhängenden Kontrollflussgraphen, durchgeführt. Hierbei werden die Kontrollflussgraphen nicht miteinander verbunden, sondern getrennt betrachtet. In der in dieser Arbeit erstellten interprozeduralen Analyse 5 wird genau dieses Modell - eine Menge nicht miteinander verbundener Kontrollflussgraphen - verwendet. Die Gesamtheit der Graphen repräsentiert den Datenfluss des zu analysierenden Programms, wobei jeder dieser Graphen genau eine Prozedur modelliert. Für diesen Zweck soll das allgemeine Modell des Kontrollflussgraphen entsprechend erweitert werden.

Wie gerade beschrieben, soll die interprozedurale Analyse die semantischen Eigenschaften eines ganzen Programms berechnen und ist somit viel komplexer und weniger abstrakt als eine intraprozedurale Analyse. Deswegen muss die Abbildungsfunktion für den Kontrollflussgraphen  $CFG_p = (N, E, A, s_p, r_p)$  für die interprozedurale Analyse um Prozeduraufrufe erweitert werden.

$$A : E \rightarrow Label' \mid Label' = Label \cup Procs$$

Mit  $s_p \in N$  wird der Eintrittspunkt in die Prozedur und mit  $r_p \in N$  der Rücksprungknoten aus der Prozedur  $p$  bezeichnet.

Nun können zusätzlich zu den für die intraprozedurale Analyse beschriebenen möglichen Annotationen auch noch Zuweisungen in Form von Prozeduraufrufen behandelt werden.

- **Prozeduraufrufe:**

$$x_j := p();$$

Bei einem Prozeduraufruf wird der Variablen  $x_j$  der Rückgabewert des Aufrufs der Prozedur  $p$  zugeordnet.

## 3.2 Semantik des Kontrollflussgraphen

Um Aussagen über die zwischen den Programmvariablen geltenden Relationen treffen zu können, erfolgt eine abstrahierte Betrachtungsweise aller möglichen Programmabläufe bis zu einem bestimmten Programmpunkt. Ein Programmablauf stellt sich als Folge von Anweisungen dar:

$$s \equiv s_1; \dots; s_m$$

Somit wird der Programmablauf  $s$  aus den Annotationen der einzelnen Kanten auf dem Pfad von einem Startknoten  $u$  zu dem betrachteten Knoten  $v$  gebildet. Da ein Programmablauf  $s$  aus den Kantenannotationen des Pfades von einem Startknoten bis zu dem spezifizierten Zielknoten besteht, spielt der Effekt  $\mathbf{S}$  einer Kante eine wesentliche Rolle in der Erstellung des Systems von Untermengenrelationen.

$$\begin{aligned} [\text{S1}] \quad \mathbf{S}(v_{start_p}) &\supseteq \{\varepsilon\} \\ [\text{S2}] \quad \mathbf{S}(v) &\supseteq \mathbf{S}(u); \mathbf{S}(e) \text{ wenn } e = (u, v) \in \mathbf{Base} \end{aligned}$$

[S1] beschreibt, dass am Eintrittspunkt in eine Prozedur  $p$  ein leerer Programmablauf vorliegt. Mit der Untermengenrelation [S2] wird angegeben, wie bei Erreichen der Kante  $e = (u, v)$  der bis Programmpunkt  $u$  gültige Programmpfad um die Kante von  $u$  nach  $v$  vergrößert wird.

Die intraprozedurale Analyse linearer Ungleichungen stellt eine abstrakte Interpretation eines für das zu analysierende Programm spezifischen Relationensystems dar. Dieses System von Relationen wird durch alle möglichen Programmabläufe gekennzeichnet. Eine endliche Folge affiner Anweisungen verkörpert somit einen einzelnen Programmablauf  $s$ :

$$s \equiv s_1; \dots; s_m$$

Die Menge der Programmabläufe bis zu einem bestimmten Programmpunkt beschreibt die kleinste Lösung eines Systems von Untermengenrelationen auf Programmablaufmengen. Ausgangsbasis für eine Abbildung dieses Relationensystems ist die isolierte Betrachtung des Effektes einer einzelnen Grundkante  $e$  auf die Programmablaufmenge. Welche Auswirkung  $\mathbf{S}$  eine individuell betrachtete Kante  $e \in \mathbf{Base}$  auf den Programmzustand hat, ist abhängig von der Annotation  $A(e)$  dieser Grundkante.

#### **affine Zuweisung :**

Eine affine Zuweisung  $A(e) \equiv x_j := t$  führt eine einzelne Anweisung aus.

$$\mathbf{S}(e) = \{x_j := t\}$$

#### **nicht deterministische Zuweisung :**

Der Effekt einer nicht deterministischen Anweisung  $x_j := ?$  ist, dass der Variablen  $x_j$  irgendeine unbekannte Konstante zugeordnet werden kann.

$$\mathbf{S}(e) = \{x_j := c \mid c \in \mathbb{F}\}.$$

#### **skip Anweisungen :**

Eine skip-Anweisung *skip* hat keinerlei Auswirkung auf den Programmzustand.

$$S(e) = \{\}$$

### 3.3 Repräsentation des Kontrollflussgraphen

Die Programmabläufe können als Aneinanderreihung von Effekten

$$s \equiv s_1; \dots; s_m;$$

interpretiert werden. Diese Reihe von Effekten kann nun wieder durch eine Transformation des Zustandsvektors  $\mathbf{x}$  beschrieben werden, welcher die Zustandsänderung des Variablenvektors  $x$  für einen Zustand  $x \in \mathbb{F}^n$  festhält. Die Effekte  $S(e)$ , welche aus einem Programmablauf resultieren, führen zu einer Transformation des zugrundeliegenden Programmzustandes  $x$ , was bedeutet, dass der Variablenvektor  $\mathbf{x}$  verändert werden muss.

$$[[s]]x = [[s_1; \dots; s_m]]x$$

Um den Effekt eines ganzen Programmablaufs zu erhalten, muss diese Aneinanderreihung der einzelnen Effekte schrittweise aufgelöst werden.

$$[[s_1; \dots; s_{m-1}]] \circ (S(e)x)$$

Bei diesem Abbau der Effektkette werden die einzelnen Effekte miteinander kombiniert, so dass letztendlich ein kompletter Effekt des Programmablaufs angegeben werden kann. Diese Effekte können zusammengefasst werden, so dass die folgende Assoziativität gilt:

$$S_{m-1} \circ (S_m x) = (S_{m-1} \circ S_m) x$$

Diese assoziative Beziehung zwischen den Effekten ist bei der folgenden Interpretation der Effekte gegeben und darf nicht als grundsätzlich gültig angenommen werden. Abhängig von der Art der betrachteten Kante, ergeben sich unterschiedliche Transformationen der Zustandsmengen.

Jede Zuweisung veranlasst dabei eine Transformation des dazugehörigen Programmzustandes.

- **Transformation für eine affine Zuweisung  $\mathbf{x}_j := t$**

$$[[\mathbf{x}_j := t]]x = x[\mathbf{x}_j \mapsto t(x)]$$

- **Transformation für eine nicht affine Zuweisung  $\mathbf{x}_j := ?$**

$$\forall c \in \mathbb{F} : [[\mathbf{x}_j := ?]]x = x[\mathbf{x}_j \mapsto c]$$

- **Transformation für eine skip-Anweisung** *skip*;

$$\llbracket skip \rrbracket x = x$$

Die Transformation einer skip-Kante ist die Identitätsfunktion, da diese Anweisung keine Veränderung des Zustandsvektors nach sich zieht.

- **Transformation für eine lineare Bedingung**  $c \diamond 0 \mid \diamond \in \{\geq, \leq\}$

$$\llbracket (c \diamond 0) \rrbracket x = x \mid (c \times x) \diamond 0$$

Mit  $(c \times x)$  muss die Summe über die Produkte der einzelnen Komponenten gebildet werden. Das Ergebnis dieser Aufaddition ist dann gemäss des Ungleichheitsoperators der Bedingung auf 0 abzuprüfen, so dass anhand dieses Vergleichs festgestellt werden kann, ob die Bedingung erfüllt wird.

- **Transformation für einen Funktionsaufruf**  $x_j := f()$

Bei Auftreten eines Funktionsaufrufes  $x_j := f()$  ist die folgende Transformation auf dem Variablenvektor  $x$  durchzuführen:

$$\llbracket (x_j := f()) \rrbracket x = (x \oplus (x_j \mapsto (\llbracket r_f \rrbracket (\mathbf{0} \oplus a_{f_i} \mapsto x_i))_{return}))$$

Diese Transformationsvorschrift gibt an, dass der Rückgabewert *return* der Funktion an die Variable  $x_j$  zuzuweisen ist. Ausserdem sind die Werte der globalen Variablen  $x_i$  entsprechend zu setzen, wenn diese durch den Funktionsaufruf verändert wurden. So kann der Effekt eines Funktionsaufrufs  $f()$  beschrieben werden.

## 3.4 Repräsentation eines Programmzustandes

Mit Einführung der Kontrollflussgraphen haben wir ein Mittel gefunden haben, um die Programmstruktur abstrakt zu beschreiben. Nun brauchen wir nur noch eine Möglichkeit die Abhängigkeiten, die zwischen den Programmvariablen gelten an jedem Programmzustand mit Hilfe einer endlichen Darstellung zu charakterisieren. Dabei wollen wir die abstrahierte Abbildung mit den Formalismen der konvexen Mengen und der Polyeder kennzeichnen.

### 3.4.1 Konvexe Mengen

In diesem Abschnitt sollen die **konvexen Mengen** definiert werden, da diese eine endliche Darstellungsform der Zustände liefern. Mit deren Hilfe können affine Relationen zwischen den einzelnen Programmpunkten angegeben werden.

Eine Menge  $C \subseteq \mathbb{F}^n$  heißt **konvex**, wenn  $\forall x, y \in C$  auch die Verbindungsstrecke

$$[x, y] := \{(1 - \lambda)x + \lambda y \mid \lambda \in [0, 1]\}$$

zu  $C$  gehört.

Ein Teilraum  $C \subseteq \mathbb{F}^n$  ist genau dann konvex, wenn für alle Elemente  $g_i \in C$  gilt:

$$\sum_i \lambda_i g_i \in C \mid \lambda_i \geq 0; \mid \sum_i \lambda_i = 1$$

Mit  $g_i$  werden die Generatoren, bestehend aus Punkten, Strahlen und Linien, bezeichnet.

Die Schnittmenge konvexer Mengen ist wiederum konvex, wohingegen die Vereinigungsmenge konvexer Mengen nicht zwingendermaßen konvex sein muss. Man könnte sagen, dass die konvexen Teilmengen eines Vektorraums ein Hüllensystem formen.

Die **konvexe Hülle**  $\text{conv}(C)$  einer Menge  $C \subseteq \mathbb{F}^n$  ist die Menge aller Elemente  $g_i$ , die sich als konvexe Kombination von Elementen aus  $C$  schreiben lassen

$$\text{conv}(C) = \{ \sum_i \lambda_i g_i \mid g_i \in C, \lambda_i \geq 0, \sum_i \lambda_i = 1. \}$$

oder als Durchschnitt aller konvexen Mengen, die  $C$  enthalten. Die konvexe Hülle der Menge  $C$  ist die kleinste konvexe Menge, die  $C$  enthält.

Es ist kurz anzumerken, dass eine Hyperebene auch konvex ist, ebenso wie die durch die Hyperebene erzeugten Halbräume.

#### *Einschub: Linearkombination*

Aus der linearen Algebra ist bekannt, dass eine Linearkombination  $x$ , bestehend aus endlich vielen Elementen  $x_1, \dots, x_n$  einer Menge  $C$  die Summe von beliebigen Vielfachen dieser Elemente darstellt. Um die Vielfachen berechnen zu können, sind Koeffizienten  $\lambda_i$  zu wählen, mit denen die Elemente multipliziert werden.

$$x = \sum_{i=1}^n \lambda_i x_i$$

In einem Vektorraum ist die Linearkombination von Vektoren mit Koeffizienten aus dem Körper des Vektorraums wieder ein Element des Vektorraums. Lassen sich alle Elemente des Vektorraums als Linearkombination aus einer Menge  $C$  darstellen, so bildet  $C$  eine Basis des Vektorraums.

### 3.4.2 Polyeder

#### Formale Repräsentation

Eine weitere mögliche Darstellungsform eines Programmzustandes bilden die *Polyeder*, die in diesem Abschnitt mathematisch dargestellt werden. Die Klasse der Polyeder soll in der intra-prozeduralen Analyse 4 die Menge der Variablenbelegungen so gut als möglich approximieren.

Für die Analyse wird ein Vektor  $\mathbf{x}$  über die in der Prozedur vorkommenden Variablen definiert  $\mathbf{x} = (x_1, \dots, x_n)$ . Ausgangspunkt für die formale Repräsentation der linearen Beziehungen der Programmvariablen ist der Raum  $\mathbb{F}$ , in dem sich die Werte der Programmvariablen befinden können. Für ein Polyeder  $Poly$  gibt es zwei verschiedene Darstellungsformen,

- das **Relationensystem** (RS):  
das Tupel  $RS(Poly) := (A, \mathbf{b})$
- und das **Generatorensystem** (GS):  
das Tupel  $GS(Poly) := (P, R, L)$ .

Auf diese beiden möglichen mathematischen Repräsentationen eines Polyeders wird nun näher eingegangen.

Algebraisch betrachtet kann ein Polyeder als Lösungsmenge eines linearen Systems von Ungleichungen

Relationensystem (RS):

$$\{\mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}\} = \{\mathbf{x} \mid a_i^T \mathbf{x} \leq b_i, i = 1, \dots, m\}, \text{ mit}$$

$$A = [a_1, \dots, a_m]^T \in \mathbb{F}^{m \times n},$$

$$\mathbf{b} \in \mathbb{F}^m$$

$$\mathbf{x} \in \mathbb{F}^n$$

veranschaulicht werden. Dabei gestaltet sich eine **Ungleichung** als  $\{\sum_{i=1}^n (a_i x_i) \leq b_i\}$ .

Dies entspricht genau dem Durchschnitt von endlich vielen abgeschlossenen Halbräumen. Somit sind auch die polyhedralen Mengen konvex.

Im folgenden wird davon ausgegangen, dass sich dieses System in Minimalform befindet, so dass keine redundanten Nebenbedingungen enthalten sind und die Anzahl an Gleichungen maximal gehalten wird.

Die zweite Darstellungsform eines Polyeders wird geometrisch durch ein Tupel bestimmt, welches aus einer endlichen Menge von Punkten  $P$ , Strahlen  $R$  und Linien  $L$  zusammengesetzt ist.

Generatorensystem (GS):  $(P, R, L)$

Ein aus Punkten bestehender Teilbereich des zu betrachtenden Raumes kann mit Hilfe von Punkten folgendermassen aufgespannt werden:

Teilbereich, der von Punkten aufgespannt wird:

$$p = \sum_{i=1}^n (\lambda_i p_i);$$

$$0 \leq \lambda_i \leq 1; \quad \sum_{i=1}^n \lambda_i = 1; \quad i = 1, \dots, n$$

Nun werden die Teilräume beschrieben, die mit Hilfe von Strahlen und Linien, welche auf die Punkte des gerade dargestellten Bereichs aufsetzen, aufgespannt werden.

Teilbereich, der von Strahlen aufgespannt wird:

$$r = \sum_{j=1}^n (\mu_j r_j);$$

$$0 \leq \mu_j \leq +\infty; \quad j = 1, \dots, n$$

Auch die Linien, die weder in der positiven noch in der negativen Richtung in ihrer Unendlichkeit eingeschränkt werden, können nur in Kombination mit einem Aufpunkt existieren.

Teilbereich, der von Linien aufgespannt wird:

$$l = \sum_{k=1}^n (\eta_k l_k)$$

$$-\infty \leq \eta_k \leq +\infty; \quad k = 1, \dots, n$$

Polyeder verkörpern somit genau die abgeschlossenen konvexen Mengen.

### Umwandlung der Polyederdarstellungen

Die Notwendigkeit der Umwandlung eines geschlossenen konvexen Polyeders, gegeben als Relationensystem, in das entsprechende Generatorsystem ergibt sich daraus, dass es einfacher ist bestimmte Operationen auf geometrischer Ebene durchzuführen.

Um zum Beispiel eine Inklusionsbeziehung zwischen zwei Polyedern  $Poly_1 \subseteq Poly_2$  herzustellen, ist es notwendig beide Darstellungen zur Hand zu haben. Denn dabei müssen alle geometrischen Elemente von  $Poly_1$  das Relationensystem von  $Poly_2$  erfüllen.

Ausgehend von der geometrischen Repräsentation eines Polyeders, lässt sich ein Relationensystem dadurch gewinnen, dass man ein Gleichungssystem auf der konvexe Hülle der Punkte, Strahlen und Linien aufstellt, siehe[CH78].

Um umgekehrt vom Relationensystem in ein Generatorsystem überzugehen, sind alle Eckpunkte des Polyeders zu bestimmen, was mit Hilfe des Simplexverfahrens von Linear Programming (LP) 7.2 erreicht wird [CH78]. Das Finden der geometrischen Darstellung für ein Polyeder kann trotz der Optimierungsmöglichkeiten, die lineares Programmieren 7.2 bietet, sehr kostspielig werden. Diese teure Umwandlung der einen Darstellungsform des Polyeders in die andere sollte weitestgehend vermieden werden.



# 4 Intraprozedurale Analyse mittels Polyedertheorie

## 4.1 Bestimmung linearer Beziehungen zwischen Programmvariablen

Bei der hier betrachteten Programmanalyse von Cousot [CH78] stützt man sich auf eine statische Ermittlung eines Systems von linearen Gleichungen und Ungleichungen. Ziel der Cousot'schen Methode ist es, Relationen zwischen Programmvariablen angeben zu können. Wie in 3.1 erwähnt, werden auch hier an jeder Kante des Kontrollflussgraphen die Programmanweisungen durch die entsprechenden Transformationen modelliert. Dieses Vorgehen führt zu Abhängigkeiten der Ungleichungssysteme für jeden Programmpunkt, welche sich mit Hilfe eines iterativen Fixpunktalgorithmus lösen lassen.

## 4.2 Repräsentation des Kontrollflussgraphen

Ausgehend von dem in 3.1 definierten endlichen zusammenhängenden Kontrollflussgraphen für eine intraprozedurale Analyse soll hier der Programmzustand konkret mit Hilfe von Polyedern aus 3.4.2 dargestellt werden.

Zugrunde liegt das in Kapitel 3.1 beschriebene Modell eines Kontrollflussgraphen, wobei nur Basiskanten *Base* betrachtet werden.

$$Label \supseteq Base$$

An jedem Programmpunkt sind beide Darstellungsformen eines Polyeders *Poly*, wie in 3.4.2 beschrieben, verfügbar. Das ist sowohl

- das Relationensystem  $RS(Poly) = (A, \mathbf{b})$ ,
- als auch das Generatorsystem  $GS(Poly) = (P, R, L)$ .

Die Kanteneffekte bestehen in einer Änderung des am Eingangsknoten *s* vorliegenden Polyeders  $Poly_s$  in ein transformiertes Polyeder  $Poly_t$ , das am Zielknoten *t* gilt.

### 4.3 Abstrakte Interpretation

Bei der Definition der abstrakten Interpretation soll die konkrete Semantik aus 3.2 durch die abstrakte Semantik ersetzt werden. Hierbei muss für jeden konkreten Wert eine Abstraktion gefunden werden, die diesen konkreten Wert unter Zuhilfenahme einer Abstraktionsfunktion beschreibt. Die abstrakte Semantik muss so gewählt werden, dass die Berechnung immer terminiert und die Abstraktion gute Rückschlüsse auf das Programmverhalten ermöglicht. Das Polyeder  $Poly$  stellt eine Abstraktion  $\mathbf{S}^\#(s)$  eines Programmdurchlaufs  $\mathbf{S}(s)$ , ausgehend von einem Startpunkt  $s$  bis zu einem Zielpunkt  $t$ , dar. Diese abstrakte Interpretation  $\mathbf{S}^\#$  des Relationensystems  $\mathbf{S}$  über Programmlaufmengen liefert gerade die Berechnung der Polyeder, eine Abstraktion der konkreten Programmzustände. Das abstrahierte Teilmengenrelationssystem  $\mathbf{S}^\#$  kann so veranschaulicht werden:

$$\begin{aligned} [\mathbf{S1}]^\# \quad \mathbf{S}^\#(s_{start_f}) &\supseteq \{\varepsilon\} \\ [\mathbf{S2}]^\# \quad \mathbf{S}^\#(v) &\supseteq \mathbf{S}^\#(u); \mathbf{S}^\#(e) \quad \text{wenn } e = (u, v) \in \mathbf{Base} \end{aligned}$$

Die konkrete Semantik aus 3.2 lässt sich unter Verwendung der Abstraktionsfunktion beschreiben, welche entsprechend des Labels der Kante die jeweilige Transformation durchführt. Dabei verdeutlicht der Effekt einer Kante wie das Polyeder am Zielprogrammmpunkt  $t$  durch die Transformationsvorschrift aus dem Polyeder des Quellknotens  $s$  beeinflusst wird.

Im folgenden wird eine Formulierung für den Effekt der Anwendung dieser abstrakten Funktion entsprechend der Art der besuchten Kante für den jeweiligen Programmknoten gegeben.

Der Kanteneffekt einer Zuweisung  $A(e) \equiv x_i := E(\mathbf{x})$  auf einem Polyeder  $Poly$  ist abhängig von der Art der Zuweisung.

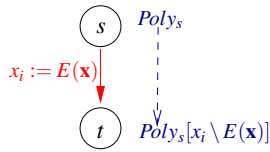
- **Affine Zuweisung:**

Zuweisung linearer Ausdrücke  $E(\mathbf{x})$ :

$$A(e) \equiv x_i := E(\mathbf{x}) \mid E(\mathbf{x}) := \sum_k (a_k x_k) + b \mid \mathbf{a}, b \in \mathbb{F}$$

**Effekt einer affinen Zuweisung:**

$$\llbracket x_i := E(\mathbf{x}) \rrbracket^\# GS(Poly) = \{g[x_i/E(\mathbf{x})] \mid g \in GS(Poly)\}$$



Der Effekt einer Zuweisung eines linearen Ausdrucks an eine Programmvariable  $x_i$  gestaltet sich in einer Veränderung der Basis des  $\mathbb{F}^n$ , indem die Werte in den Generatoren des Polyeders  $Poly_s$  des Quellknotens  $s$  durch den linearen Ausdrucks  $E(\mathbf{x})$  zu substituieren sind. Das Polyeder  $Poly_t$  des Zielknotens berechnet sich somit dadurch, dass jedes Vorkommen der Programmvariable  $x_i$  im Polyeder  $Poly_s$  durch  $E(\mathbf{x})$  ersetzt wird.

• **Nicht affine Zuweisungen:**

Zuweisung nicht linearer Ausdrücke  $E(x)$ :

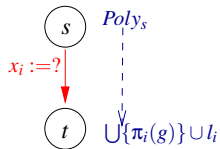
$$A(e) \equiv x_i := ?$$

**Effekt einer nicht affinen Zuweisung:**

$$\llbracket x_i := ? \rrbracket^\# GS(Poly) = \bigcup \{ \pi_i(g) \} \cup l_i \mid g \in GS(Poly)$$

$$\pi_i(g) = \{ g = g_0 + g_1 x_{i_1} + \dots + g_{i-1} x_{i_{i-1}} + g_{i+1} x_{i_{i+1}} + \dots + g_n x_n \mid g \in GS(Poly) \}$$

$$l : l(x_i) = 1 \mid \forall j = 1, \dots, n \mid j \neq i : l(x_j) = 0$$



Erfolgt die Zuweisung eines nicht linearen Ausdrucks  $E(\mathbf{x})$  (was z.B. bei einem Methodenaufruf der Fall ist) an die Programmvariable  $x_i$ , so führt dies zu einem durch diese Transformation beeinflussten Polyeder  $Poly_t$  des Zielknotens  $t$ .

$x_i$  muss dabei aus allen Generatoren des Polyeders  $Poly_s$  des Zielknotens  $t$  beseitigt werden. Es ist somit keine Information über den Wert der Variablen  $x_i$  an Programmpunkt  $t$  bekannt, da jeder Wert aus  $\mathbb{F}$  zuzuordnen wäre. Diese Aussage lässt sich durch Hinzufügen der Linie  $l_i$  zum Polyeder  $Poly_t$  geometrisch ausdrücken.

• **Bedingungen:**

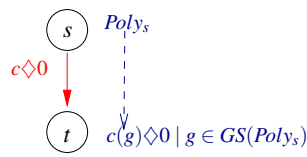
lineare Ungleichheitsbedingung:  $A(e) \equiv c \diamond 0 \mid \diamond \in \{ \geq, \leq \}$

**Effekt einer linearen Ungleichheitsbedingung:**

Eine Bedingung  $c$  ist als Hyperebene zu interpretieren, welche das vorliegende Polyeder  $Poly_s$  schneidet.

$$c : \mathbf{x} \rightarrow \mathbb{N}$$

$$\llbracket c \diamond 0 \rrbracket^\# GS(Poly) = \{ c(g) \diamond 0 \mid g \in GS(Poly) \}$$



Polyeder  $Poly_t$  hinzuzufügen.

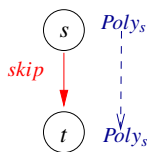
Ist die Kante mit einer linearen Ungleichheitsbedingung beschriftet, so bewirkt diese, dass das Polyeder  $Poly_t$  des Zielprogrammunktes  $t$  nur die Generatoren enthält, welche die Bedingung erfüllen. Zusätzlich sind die Generatoren, die einen Schnittpunkt mit der Gleichheitsbedingung ergeben, zu dem

- **skip-Anweisungen:**

skip-Anweisung:  $A(e) \equiv skip$

**Effekt einer skip-Anweisung:**

$$[[skip]]^\# GS(Poly) = GS(Poly)$$



Eine *skip*-Anweisung bewirkt keine Transformation des Zielknotens  $t$ . Diese wird so umgesetzt, dass das an Programmpunkt  $s$  gültige Polyeder  $Poly_s$  einfach an den Programmpunkt  $t$  übernommen wird.

## 4.4 Naiver Fixpunktalgorithmus

Nun versucht man nach Festlegung der theoretischen Mittel das betrachtete Datenflussproblem mit Hilfe eines iterativen Fixpunktalgorithmus zu lösen. Dabei wird der Kontrollflussgraph unter Verwendung des Besuchermusters vollständig und effizient in einer Vorwärtsanalyse traversiert. Bei der Weitergabe von Werten unter Verwendung des Fixpunktalgorithmus berechnet sich das gültige Polyeder für einen Knoten aus den Informationen des Vorgängerknotens im Kontrollflussgraphen. Die Analyse arbeitet so lange, bis sich ihre Ergebnisse stabilisieren. Zur Durchführung unserer Datenflussanalyse verwendet man einen *naiven iterativen Worklistalgorithmus*.

Bei diesem Verfahren werden nicht in jeder Iteration die Werte an allen Knoten neu berechnet, sondern nur für diejenigen an denen sich die Eingangsinformation geändert hat. Hierfür verwendet man eine Worklist, um sich diese Knoten zu merken. Wie der Algorithmus zur naiven Fixpunktiteration 4.1 verdeutlicht, wird in jedem Schritt ein Polyeder über den Kontrollflussgraphen weitergereicht. Gelangt man an einen Knoten im Kontrollflussgraph, so muss zuerst überprüft werden, ob das zu propagierende Polyeder schon durch das an diesem Programmpunkt vorhandene Polyeder ausgedrückt wird. Ist dies der Fall, so wird das Polyeder nicht mehr weitergegeben, da es keine zusätzliche Information für die Analyse liefert. Andernfalls wird dieser Programmzustand mit Hilfe des zu propagierenden Polyeders beschrieben. Das neue Polyeder wird entsprechend der Transformationen über alle ausgehenden Kanten weitergereicht.

Nun muss der in 4.1 vorgestellte naive Fixpunktalgorithmus noch dahingehend untersucht werden, ob er terminiert. Das bedeutet, dass im Falle einer Terminierung des Algorithmus keine unendlichen Schleifen vorkommen dürfen. Die einzige Möglichkeit einer Nichtterminierung wäre durch die *while*-Schleife aus Zeile 14 gegeben.

```
while workset  $\neq \emptyset$  do
```

Erst wenn das *workset* leer ist, ist das Abbruchkriterium für diese Schleife gegeben. Somit muss  $Poly[v]$  das Polyeder enthalten, das alle an diesem Programmpunkt  $v$  gültigen Polyeder bereits ausdrückt. Allein die Erfüllung dieser Bedingung verhindert ein weiteres Füllen des *worksets* mit durch den Kontrollflussgraph zu propagierenden Polyedern. Jedes Polyeder  $p_s$ , für das wir überprüfen müssen, ob es in dem bereits zugrundeliegende Polyeder  $Poly[v]$  enthalten ist, liegt in dem maximalen Polyeder, das den ganzen betrachteten Raum  $\mathbb{F}$  umfasst.

Um nach einer endlichen Anzahl von Schritten ein maximales Polyeder zu erhalten, wird die im nächsten Abschnitt 4.4.1 beschriebene Technik des *Widenings* verwendet. An jedem Programmpunkt wollen wir ein maximales Polyeder führen, dessen Generatorensystem aus einer minimalen Anzahl an Generatoren und dessen Relationensystem aus einer minimalen Anzahl an Relationen besteht.

Die Erweiterung des für einen bestimmten Programmzustand geltenden Polyeders um das neue Polyeder erfolgt unter Bildung der konvexen Hülle, wie in 3.4.1 definiert.

$$Poly[v] := Poly[v] \sqcup \{p_s\};$$

#### 4.4.1 Widening

Der Definitionsbereich der konvexen Polyeder kann zu unendlichen aufsteigenden Ketten führen. Somit ist es nötig, um die Konvergenz der Fixpunktberechnung zu erzwingen, den Mechanismus des **Widenings** einzusetzen. Widening kommt z.B. bei der abstrakten Auswertung von Schleifen zu tragen, da man hierbei eine Obergrenze einer strikt ansteigenden unendlichen Kette in einer endlichen Anzahl an Berechnungsschritten erhalten möchte. Die Wirkungsweise des Wideningoperators wird nun näher erläutert.

##### Definition

Der *Wideningoperator*  $\nabla$  ist eine Funktion auf einem Verband  $V$ , die folgendermassen definiert wird:

$\nabla \in V \times V \rightarrow V \mid \forall x, y \in V$  gilt:

$$x \sqsubseteq x \nabla y, y \sqsubseteq x \nabla y$$

Für alle monoton aufsteigenden Ketten  $x_0 \sqsubseteq x_1 \sqsubseteq \dots \sqsubseteq x_n$  gilt:

die Kette  $y_0, y_1, \dots, y_n$  mit  $y_0 = x_0, \dots, y^{i+1} = y^i \nabla x^i$  ist nicht streng monoton ansteigend.

Die aufwärts gerichtete Iterationssequenz mit Hilfe des Wideningoperators gestaltet sich in

$$X^0 = \perp$$

$$X^{i+1} = X^i, \text{ wenn } f(X^i) \sqsubseteq X^i$$

$$X^{i+1} = X^i \nabla f(X^i), \text{ sonst}$$

### Wideningoperator in Bezug auf Polyeder

Wendet man den Wideningoperator auf die beiden Polyeder  $Poly_1 = \{A_1, \mathbf{b}_1\}$ ,  $Poly_2 = \{P_2, R_2, L_2\}$  an, so ist das Resultat das Polyeder, welches alle Relationen von  $Poly_1$  enthält, die auch von allen Generatoren aus  $Poly_2$  erfüllt werden. Es werden die Relationen aus dem Beschreibungstupel für das Polyeder entfernt, die nicht schnell stabilisieren, wobei möglichst viel Information über das an dem Programmpunkt gültige Relationensystem beibehalten werden soll. Geometrisch gesehen führt Widening zu einer Ersetzung von Punkten durch Strahlen und Linien [CH78]. Wie die Definition des Widenings 4.4.1 zeigt, ist es notwendig beide Darstellungsformen eines Polyeders zur Hand zu haben.

## 4.5 Implementation

Um eine Implementation der theoretisch formulierten Analyse durchzuführen, wurde die Bibliothek *PPL* 7.1 verwendet. Diese bietet eine API, um Operationen auf Polyedern anzuwenden. Die Polyeder repräsentieren in der intraprozeduralen Analyse *polyhedralAnalysis* die Zustände. Die Darstellung eines Polyeders als Polyederobjekt, bestehend aus dem Generatoren- und dem Relationensystem, ist nicht sinnvoll, da sämtliche Transformationen nur auf einer Darstellung berechnet werden.

In unserer Analyse verwalten wir, abweichend von der Forderung von Cousot parallel immer beide Repräsentationen eines Polyeders an jedem Programmpunkt zu führen, nur die geometrische Darstellung eines Polyeders, das Generatorensystem. Hiermit muss nicht an jedem Programmpunkt das Relationensystem zu dem entsprechend der Transformationsvorschrift beeinflussten Generatorensystem neu berechnet werden, was die Effizienz steigert.

Auf die duale Darstellung eines Polyeders kann trotzdem nicht verzichtet werden, da einige Operationen in *PPL* sowohl das Generatorensystem wie auch das Relationensystem zu einem Polyeder benötigen. Diese kostenintensive Konvertierung in das Relationensystem, welche man eigentlich vermeiden wollte, ist bei Widening und dem Bilden der konvexen Hülle von Polyedern erforderlich.

Wie in 4.4 dargestellt, wird nur ein naiver Fixpunktalgorithmus verwendet, so dass in jedem

Schritt ein ganzes Generatorensystem propagiert werden muss. Effizienter wäre hingegen der Einsatz einer inkrementellen Fixpunktiteration, bei der nur einzelne Generatoren weitergegeben und auf Redundanz geprüft werden müssen. Dieser Ansatz kann mit Hilfe von *PPL* nicht verfolgt werden, da die neu berechneten Generatoren, die programmiertechnisch in einem Generatorensystemobjekt verwaltet werden, nicht erkennbar sind. Das Generatorensystem muss immer in minimaler Form sein, so dass kein Generator durch eine Linearkombination der anderen Generatoren ausgedrückt werden kann. Dazu steht unter *PPL* die Funktion *minimize* zur Verfügung, welche jedoch wieder nur auf ein Polyederobjekt, also der dualen Darstellung eines Polyeders, angewendet werden kann.

Für die durchgeführte Analyse zeigen die Operationen zu wenig Transparenz, um einen Gewinn an Effizienz zu erzielen. Auch in Bezug auf die Funktionalität des Widenings ist zu wenig Transparenz gegeben. In *PPL* wird ein auf Heuristiken beruhendes Verfahren des Widenings, das sogenannte *BHRZ03 Widening* verwendet, was in [RBZ03] ausführlicher erläutert wird. Hierbei hat der Entwickler keine Möglichkeit darauf Einfluss zunehmen, welche Heuristik beim Einsatz dieser Wideningfunktion Anwendung findet.

Um eine interprozedurale Analyse durchzuführen, müssten die Kontrollflussgraphen miteinander verbunden werden. Dies bedeutet einen grossen Informationsverlust, da am Ende einer Prozedur nichtdeterministische Sprünge an alle Aufrufstellen dieser Prozedur erfolgen müssten. Mit dieser naiven Analyse müsste ein sehr aufwendiges Umfängen betrieben werden, welches noch dazu wenig präzise Aussagen liefert. Damit wir trotzdem den Effekt eines ganzes Programms beschreiben können, wird in dem nächsten Kapitel 5 eine interprozedurale Analyse beschrieben, die nur auf einem Generatorensystem arbeitet und somit eine Konvertierung in das Relationensystem nutzlos macht.

```

1 // fixpointiteration with given Node v, given Polyhedron p and
// predefined Set of Edges E and Nodes N
Set fixpointiteration (Node v, Polyhedron p, Set E, Set N) {
//Initialisation
Set [] Poly = new Set (|N|);
6 forall n ∈ N do
Poly[n] := ∅
od

//create a new workset with visited node and calculated GeneratorSystem
11 Set workset := {(v, p)};

//naive fixpointiteration
while workset ≠ ∅ do
let u_s ∈ workset
16 begin
(u_s, p_s) := extract(workset);
if (p_s ∉ ∪ Poly[v]) {
//compute convex hull of p_s and prior polyhedron
Poly[v] := Poly[v] ∪ {p_s};
21 //skip-Edge
forall (u, v) ∈ E | A((u, v)) ≡ skip do
workset := workset ∪ {p_s, v};
od
//linear Assignment-Edge
26 forall (u, v) ∈ E | A((u, v)) ≡ x_i := t do
workset := workset ∪ {(p_s)[x_i t], v};
od
//non linear Assignment-Edge
forall (u, v) ∈ E | A((u, v)) ≡ x_i :=? do
31 workset := workset ∪ {(π_i(p_s) ∪ l(x_i)), v};
od
//linear Assertion-Edge
forall (u, v) ∈ E | A((u, v)) ≡ c△0 do
36 workset := workset ∪ {(c(p_s)△0), v};
od
fi
end
od
41 }

```

Abbildung 4.1: Naiver Fixpunktalgorithmus

# 5 Interprozedurale Analyse unter Verwendung von Linearer Algebra

## 5.1 Abstrakte Semantik für die interprozedurale Analyse

Für die Untersuchung affiner Beziehungen zwischen den Programmvariablen dient als Ausgangspunkt eine abstrakte Interpretation aller möglichen Programmabläufe bis zu einem bestimmten Zustand. Wie erwähnt stellt sich ein Programmablauf als eine Folge von Anweisungen dar. Abhängig von den Annotationen der Kanten, kann für jedes Programm ein spezifisches System von Untermengenrelationen aufgestellt werden, dessen Lösung den Effekt eines Programms beschreibt. Zur Berechnung dieses Effektes werden zwei verschiedene Analysen definiert, wobei bei der ersten Analyse die Programmablaufmengen **E** mit Hilfe von **Matrizen** und bei der darauf aufbauenden Analyse diese Programmablaufmengen **K** unter Verwendung von **Vektoren** abstrahiert werden. Die beiden Bezeichnungen **E** und **K** sollen die unterschiedliche Darstellungsweise der konvexen Mengen verdeutlichen, die die Programmzustände charakterisieren. Nur ein Zusammenspiel der beiden Analysen gestattet Aussagen innerhalb der interprozeduralen Analyse.

### 5.1.1 Abstraktion konkreter Programmzustände

Nun verwenden wir die konvexen Mengen aus 3.4.1, um eine Abstraktion der Menge der Programmläufe von einem Startpunkt  $s$  zu einem Zielpunkt  $t$  vorzunehmen. Die Berechnung der konvexen Mengen erfolgt durch eine Fixpunktiteration über die in 3.2 aufgestellten Teilmengenrelationensysteme. Da die konvexen Mengen einen konkreten Programmzustand abstrahiert darstellen, wird ein einzelner Programmpunkt als ein Unterraum des  $\mathbb{F}^{(n+1) \times (n+1)}$  interpretiert. Es wird somit ein Unterraum eines  $(n+1) \times (n+1)$ -dimensionalen Vektorraumes an einem bestimmten Programmpunkt aufgespannt, wenn in der Analyse  $n$  Programmvariablen vorkommen. Zur Spezifikation dieser Interpretation eines Programmzustandes wird eine Abstraktionsfunktion  $\alpha$  definiert, welche die Abstraktion des Teilmengenrelationensystems über alle Programmablaufmengen auf Teilmengenrelationensysteme über die konvexen Mengen symbolisiert.

$$\alpha : 2^{\text{Runs}} \rightarrow \mathbf{Sub}(\mathbb{F}^{(n+1) \times (n+1)})$$

Mit  $\mathbf{Sub}(\mathbb{F}^{(n+1) \times (n+1)})$  wird ein aufspannender Unterraum eines  $\mathbb{F}^{(n+1) \times (n+1)}$ -dimensionalen Vektorraumes bezeichnet. Für die Interpretation der Programmzustände als konvexe Mengen soll ein realisierbarer Algorithmus angegeben werden. Damit dies möglich ist, muss zuerst

eine endliche Darstellung der konvexen Mengen gefunden werden. Die Generatoren einer konvexen Menge spannen unter Bildung der konvexen Hülle  $\text{conv}$  über diese Generatoren einen Unterraum auf.

Jeder Programmlauf soll durch die Abstraktionsfunktion  $\alpha$  auf eine konvexe Menge von Generatoren  $G$ , abgebildet werden:

$$\alpha(R) = \mathbf{conv}(\{G_r \mid r \in R\})$$

Somit erhalten wir für jeden einzelnen Programmlauf eine Basis für das Generatorensystem.

$$\alpha(\{r\}) = \mathbf{conv}(\{G_r\})$$

Da wir jeden Programmzustand mit Hilfe einer minimalen Basis von Generatoren charakterisieren wollen, muss an jedem Programmpunkt ein minimales Generatorensystem abgespeichert werden, was durch die konvexe Hülle über dieses Generatorensystem  $GS$  bewirkt wird.

$$\langle \forall g \in GS \rangle_{\text{conv}}$$

Die Anwendung dieses Operators der konvexen Hülle liefert gerade eine minimale Basis, welche einen Unterraum des  $\mathbb{F}^{(n+1) \times (n+1)}$  aufspannt und einen Programmzustand beschreibt. Im folgenden kann mit den Basisgeneratoren  $\langle G \rangle_{\text{conv}}$  an Stelle einer Generatorenmenge gerechnet werden ohne dass ein Präzisionsverlust zu verzeichnen wäre. Zudem ist es nötig für einen einzelnen Generator  $g$  überprüfen zu können, ob er bereits in dem vorliegenden Generatorensystem  $GS$  enthalten ist, oder durch seine Hinzunahme ein neuer Unterraum aufgespannt wird.

$$g \in \langle GS \rangle_{\text{conv}}$$

Bei diesem Test muss darauf geachtet werden, ob sich  $g$  als Linearkombination der in  $GS$  enthaltenen Generatoren darstellen lässt und im Falle einer linearen Abhängigkeit durch geeignete Kombination der Generatoren aus  $GS$  dargestellt werden kann. Das Vorgehen bei Durchführung dieser Abprüfung wird in 7.2 angegeben. Somit haben wir mit der Abbildung der konvexen Mengen eine endliche Darstellungsform für einen Programmpunkt gefunden. Nach jeder Durchführung einer Transformation ist die konvexe Hülle des entstandenen Generatorensystems  $\langle \rangle_{\text{conv}}$  zu bilden. Somit wird am Zielprogrammpunkt  $t$  stets eine minimale Basis der Generatoren abgelegt. Die Spezifikation der konvexen Hülle  $\text{convexHull}()$  wird in 6.2 genau erläutert.

### Analyse zur Beschreibung der Kanteneffekte

Für jeden Programmpunkt führen wir Generatoren für die konvexen Mengen. Dabei sind zwei verschiedene Darstellungen der Generatoren zu differenzieren. Bei der Durchführung der ersten Analyse gestaltet sich ein Generator in Form einer Matrix. In dieser Analyse werden **Effektmatrizen** zur Beschreibung der vorzunehmenden Transformationen der entsprechenden

Kantenarten als Hilfsmittel für die zweite Analyse erzeugt, in der die Zustände mit Hilfe dieser Matrizen transformiert werden können. Da wir in unserer Analyse von  $n$  Programmvariablen ausgehen, wird ein  $(n + 1) \times (n + 1)$ -dimensionaler Vektorraum mit Hilfe von  $(n + 1) \times (n + 1)$  Matrizen aufgespannt. Die  $n + 1$ -te Zeile in der Matrix charakterisiert die Art der entsprechenden Effektmatrix. Ist der konstante Wert in dieser Zeile eine 1, so handelt es sich um eine Punktmatrix, im Falle einer 0 um eine Strahl- oder Linienmatrix. Die  $n + 1$ -te Spalte stellt den konstanten Term  $t_0$  dar. Hier folgt ein kurzer Überblick über die 3 verschiedenen Matrizenarten.

$$E_{point} = \left( \begin{array}{c|c} I_n & \mathbf{0} \\ \hline 0 & 1 \end{array} \right)$$

$$E_{ray} \left( \begin{array}{c|c} I_n & \mathbf{0} \\ \hline \mathbf{0} & 0 \end{array} \right) \quad E_{line} = \left( \begin{array}{c|c} I_n & \mathbf{0} \\ \hline \mathbf{0} & 0 \end{array} \right)$$

Unter Verwendung dieser Matrizen lassen sich die Generatoren in der ersten durchzuführenden Analyse realisieren.

### Analyse zur Beschreibung der Zustandsmengen

Aufbauend auf die Analyse zur Berechnung der Effektmatrizen, gestaltet sich die Darstellung der Generatoren innerhalb der zweiten Analyse in Form von Vektoren für Punkte, Strahlen und Linien.

$$K_{point} = (x_1, \dots, x_n, 1);$$

$$K_{ray} = (x_1, \dots, x_n, 0)$$

$$K_{line} = (x_1, \dots, x_n, 0)$$

Auch hierbei dient die Position  $n + 1$  zur Bezeichnung der Art des Generators, wobei die 1 einen Punkt und die 0 sowohl den unidirektionalen als auch den bidirektionalen Strahl charakterisiert. Die Effektmatrizen dienen nur zur Transformation der Programmezustände.

### 5.1.2 Berechnung der Effektmatrizen

In der ersten durchgeführten Analyse werden mit Hilfe der aufzustellenden Untermengenrelationensysteme **Effektmatrizen** erstellt, die abhängig von den Annotationen der Kanten auf abstrakter Ebene eine Überführung eines Programmzustandes in den nachfolgenden Programmzustand beschreiben. Mit den Effektmatrizen  $E$  sollen die bis dahin vorgenommenen Effekte festgehalten werden. Durch eine Multiplikation der Einzeleffektmatrizen erhalten wir einen ganzen Sammeffekt, so dass letztendlich der Effekt einer Methode mit Hilfe einer solchen Sammeffektmatrix dargestellt werden kann.

Wie in 3.2 beschrieben, wird ein Programmablauf als eine Aneinanderreihung von Effekten

dargestellt. Diese Effektkette muss rekursiv aufgebrochen werden, so dass sich eine Transformation auf dem Programmablauf ergibt. Bei der zu betrachtenden Analyse wird diese Transformation durch eine Effektmatrix  $E$  beschrieben.

$$[[s]]x = [[s_1; \dots; s_n]]x = (E_1 \cdot \dots \cdot E_m)x \mid x \in \mathbb{F}^m$$

Aus einem kompletten Programmablauf erhalten wir durch Verknüpfung der einzelnen Effektmatrizen eine Matrix, die den Effekt einer Methode beschreibt.

Die Berechnung der Effektmatrizen als Hilfsmittel zur Erhaltung der konvexen Mengen, was in der zweiten Analyse 5.1.3 beschrieben wird, erfolgt durch die Analyse der folgenden aufgestellten Programmabläufe. Hierbei wird eine Matrix  $E$  als Abstraktion  $\mathbf{E}$  der Menge der Programmläufe  $\mathbf{E}(s)$  von einem Programmpunkt  $s$  bis zu einem Zielknoten  $t$  bezeichnet.

- Untermengenrelation [E0] für den Anfangsknoten einer Methode:

$$[\text{E0}] \quad \mathbf{E}(s_{start_f}) \supseteq I$$

Diese Untermengenrelation besagt, dass die Effektmatrix, die den Anfangspunkt einer Methode  $f$  darstellt, die symmetrische  $n \times n$  Einheitsmatrix  $I$  bezeichnet.

- Untermengenrelation [E1] für eine *skip*;-Anweisung:

$$[\text{E1}] \quad \mathbf{E}(v) \supseteq \langle \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} \quad \text{wenn } e = (u, v) \mid e \equiv \text{skip};$$

Die Untermengenrelation [E1] soll ausdrücken, dass eine *skip*;-Anweisung weder die Domäne noch die Effektmatrix beeinflusst. Somit werden alle am Programmpunkt  $u$  gültigen Matrizen unverändert an den Programmpunkt  $v$  übernommen, ohne dass am Knoten  $u$  Transformationen an den Generatoren vorgenommen werden müssen.

- Untermengenrelation [E2] für eine affine Zuweisung:

$$[\text{E2}] \quad \mathbf{E}(v) \supseteq \langle [[x_i := t]]^\# \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} \quad \text{wenn } e = (u, v) \mid e \equiv x_i := t;$$

Mit [E2] schildern wir die Untermengenrelation, die durch die Kante  $(u, v)$  das Generatorensystem des Zielknotens  $v$  mit den Generatoren aus  $u$  in Beziehung bringt. Im Falle der affinen Zuweisung resultiert dies in einer Multiplikation der Matrizen am Programmpunkt  $u$  mit der Effektmatrix der affinen Zuweisung, so dass das Produkt dieser Matrizenmultiplikation den Programmpunkt  $v$  charakterisiert. Der Effekt von  $[[x_i := t]]^\#$  ist eine Punktmatrix  $E_{affin}$  von folgender Gestalt:

$$E_{affin} = \left( \begin{array}{c|c} I_{n-i} & \mathbf{0} \\ \hline t_1 \dots t_n & t_0 \\ \hline \mathbf{0} & I_{n-i-1} \end{array} \right)$$

- Untermengenrelation [E3] für eine nicht affine Zuweisung:

$$[E3] \quad \mathbf{E}(v) \supseteq \langle \{L_i\} \cup \llbracket x_i := ? \rrbracket^\# \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} \quad \text{wenn } e = (u, v) \mid e \equiv x_i := ?; \mid L_i(i, i) = 1;$$

Bei der Relation [E3], die die Programmaufmenge für eine nichtdeterministische Zuweisung an einer Kante beschreibt, ist zusätzlich zur Multiplikation mit der Effektmatrix  $E_{\overline{affin}}$ , welche die Beseitigung aller Werte der Programmvariablen  $x_i$  zur Folge hat, noch eine Erweiterung mit einer Linienmatrix  $L_i$  durchzuführen, die an der Stelle  $(i, i)$  eine 1 hat. Diese Linie besagt, dass keinerlei Information über die Variable  $x_i$  nach Passieren der mit  $x_i := ?$  annotierten Kante besteht.

$$E_{\overline{affin}} = \left( \begin{array}{c|c} I_{n-i} & \mathbf{0} \\ \hline \mathbf{0} \dots \mathbf{0} & I_{n-i-1} \end{array} \right) \quad L_i = i \begin{pmatrix} & & & i \\ \mathbf{0} & \dots & \mathbf{0} & \\ \mathbf{0} & \dots & 1 & \dots \mathbf{0} \\ \mathbf{0} & \dots & \mathbf{0} & \end{pmatrix}$$

- Untermengenrelation [E4] für einen Prozeduraufruf:

$$[E4] \quad \mathbf{E}(v) \supseteq \langle \llbracket x_i := return \rrbracket^\# embed(\mathbf{E}(s_{end_f}, \mathbf{E}(u)), \mathbf{E}(v)) \rangle_{conv} \quad \text{wenn } e = (u, v) \mid e \equiv x_i := f();$$

Genauer zu spezifizieren ist bei einem Methodenaufruf ein gültiges Relationensystem. Dabei ist der Effekt der Funktion  $embed(\mathbf{E}(s_{end_f}), \mathbf{E}(u))$  auf den Effekt der affinen Zuweisung  $\llbracket x_i := return \rrbracket^\#$  anzuwenden. Die Funktion  $embed$  berechnet sozusagen den Effekt eines Prozeduraufrufs.

$$embed(X, Y) = T' \cdot (X \cdot T) \cdot Y + Y \cdot T''$$

mit der  $n \times n$  Matrix

$$T'' = \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & I_{n-k} \end{pmatrix},$$

der  $n \times m$  Matrix

$$T' = \begin{pmatrix} I_k & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}$$

und der  $m \times n$  Matrix  $T$ , deren Aufbau in der nachfolgenden Beschreibung dargelegt wird. Mit  $X$  wird die  $m \times m$  Effektmatrix der aufgerufenen Prozedur bezeichnet.  $Y$  bezeichnet eine  $n \times n$  Matrix des Callers, die an dem Programmpunkt unmittelbar vor dem Prozeduraufruf gültig ist.

### 1. $\mathbf{X} \cdot \mathbf{T}$ :

Bei einem Eintritt in die aufgerufene Funktion  $f$ , von nun an mit *Callee* bezeichnet, müssen die Werte der globalen Variablen, die allen Prozeduren unserer betrachteten Programmklasse gemeinsam sind, in den aufgerufenen Kontext übernommen werden. Die aktuellen Parameter des Aufrufers, *Caller* genannt, sind den formalen Parametern der Methode zuzuordnen.

Dieses Mapping der aktuellen auf die formalen Parameter und die Erhaltung der Werte der globalen Variablen wird durch die Multiplikation der Matrix  $T$  von rechts an die Effektmatrix des Endknotens der Prozedur  $T$  durchgeführt.

### 2. **Matrix $T$** :

Wenn wir davon ausgehen, dass die Effektmatrix  $X$  des Callers eine symmetrische  $m \times m$  Matrix und die am Programmpunkt  $u$  gültige Matrix  $Y$  eine symmetrische  $n \times n$  Matrix ist, so muss die zu erstellende Matrix  $T$   $m$  Zeilen und  $n$  Spalten besitzen. Die ersten  $k$  Zeilen von  $T$  bestehen aus der Einheitsmatrix  $I_k$  und an der Stelle  $(i, j)$  ist nur dann eine 1 zu setzen, wenn der formale Parameter  $i$  durch den aktuellen Parameter  $j$  auszutauschen ist. Ansonsten enthält diese Matrix nur Nulleinträge.

### 3. $(\mathbf{X} \cdot \mathbf{T}) \cdot \mathbf{Y}$ :

Nun müssen noch die formalen Parameter des Callees entsprechend der Werte der aufrufenden Variablen des Callers initialisiert werden, was mit einer Multiplikation mit  $Y$  erreicht wird.

### 4. $\mathbf{T}' \cdot ((\mathbf{X} \cdot \mathbf{T}) \cdot \mathbf{Y})$ :

Um die lokalen Variablen des Callees auszublenden, findet eine Multiplikation von links mit der Matrix  $T'$  statt, welche zur Erhaltung der Werte der globalen Variablen dient, wenn wir wieder in den aufrufenden Kontext zurückkehren.

Nach dieser Effektberechnung des Aufrufs der Prozedur  $f$  ist eine  $n \times n$  Matrix entstanden, nachdem die Änderungen an den Werten der globalen Variablen übernommen wurden.

### 5. $\mathbf{T}' \cdot ((\mathbf{X} \cdot \mathbf{T}) \cdot \mathbf{Y}) + \mathbf{Y} \cdot \mathbf{T}''$ :

Durch das Verlassen des Callees wurde der global definierten Variable *return* das Ergebnis des Methodenaufrufs zugeordnet. Abschliessend müssen nur noch die Werte der lokalen Variablen des Callers  $Y$  wiederhergestellt werden, was die Addition des Produktes  $Y \cdot T''$  erfüllt.

Letztendlich muss der Wert der Variablen *return*, der den Effekt des Methodenaufrufs beschreibt, noch an  $x_i$  zugewiesen werden. Dies kann entsprechend des in [E2] definierten Relationenschemas für affine Zuweisungen durchgeführt werden. Diese Transformationsvorschrift für Methodenaufrufe resultiert wieder in einer affinen Transformation.

- Untermengenrelation [E5] für Wächter:

$$[E5] \quad \mathbf{E}(v) \supseteq \langle \llbracket x_{(u,v)} := t \rrbracket^{\#} \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} \quad \text{wenn } e = (u, v) \mid e \equiv t \diamond 0;$$

Mit der Untermengenrelation [E5] soll die Effektmatrix für das Vorkommen einer Bedingung beschrieben werden. Auch hier werden nur nicht strikte Ungleichheitsbedingungen betrachtet. Es wird für jede Bedingung eine neue Variable  $x_{(u,v)}$  eingeführt, welcher der Wert der Bedingung  $t$  zugeordnet wird. Die Auswertung der Bedingung  $t$ , bzw. Auflösung der neu eingeführten Variable, erfolgt erst im nachhinein beim Durchlauf des Kontrollflussgraphen in der zweiten Analyse.

Mit diesem aufgestellten Relationensystem können die Effektmengen jedes Programmpunktes iterativ berechnet werden. In einer Fixpunktiteration werden diese Mengen so lange aktualisiert, bis sich ein stabiler Zustand eingestellt hat.

$$\begin{array}{ll}
 [E0] \quad \mathbf{E}(s_{start_f}) \supseteq I & \\
 [E1] \quad \mathbf{E}(v) \supseteq \langle \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv skip; \\
 [E2] \quad \mathbf{E}(v) \supseteq \langle \llbracket x_i := t \rrbracket^{\#} \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv x_i := t; \\
 [E3] \quad \mathbf{E}(v) \supseteq \langle \{L_i\} \cup \llbracket x_i := ? \rrbracket^{\#} \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv x_i := ?; \\
 [E4] \quad \mathbf{E}(v) \supseteq \langle \llbracket x_i := return \rrbracket^{\#} embed(\mathbf{E}(s_{end_f}, \mathbf{E}(u)), \mathbf{E}(v)) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv x_i := f(); \\
 [E5] \quad \mathbf{E}(v) \supseteq \langle \llbracket x_{(u,v)} := t \rrbracket^{\#} \mathbf{E}(u), \mathbf{E}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv t \diamond 0;
 \end{array}$$

Anhand dieses Relationensystems kann man leicht erkennen, dass der Effekt eines Programmpfades immer auf den Effekten des Vorgängerknotens aufbaut.

Die nächste Analyse 5.1.3 beschreibt die Effekte der einzelnen Kanten unter Zuhilfenahme des Abstraktionsmittels der konvexen Mengen, aufbauend auf den in diesem Abschnitt definierten Effektmatrizen.

### 5.1.3 Berechnung der Zustandsmengen

Die zweite Analyse dient zur Berechnung der konvexen Mengen der Generatoren in Form von **Vektoren**. Es werden mit den konvexen Menge alle gültigen affinen Relationen, die zwischen den einzelnen Programmvariablen gelten, bestimmt. Dabei greifen wir auf die in der vorhergehenden Analyse, in der alle Kontrollflussgraphen der zu analysierenden Klasse mit Effektmatrizen beschrieben wurden, berechneten Effektmatrizen der einzelnen Kanten zurück. Nun wollen wir die Programmablaufmengen für alle Programmabläufe bis zu einem bestimmten Programmpunkt innerhalb der zu analysierenden Prozedur als die kleinste Lösung des folgenden Hilfssystems von Untermengenrelationen **K** veranschaulichen.

- Untermengenrelation [K0] für den Startpunkt einer Methode:

$$[K0] \quad \mathbf{K}(s_{start_f}) \supseteq p \cup \{l_i \mid l_i = (0, \dots, 1, \dots, 0) \in lines \mid \forall i\}$$

Am Startknoten der zu analysierenden Prozedur  $f$  gehen wir von der konvexen Menge bestehend aus dem Einheitspunkt

$$p = (1, \dots, 1)$$

und den Linien  $l_i$  für jede einzelne Variable, aus.

$$l_i = (0, \dots, 1, \dots, 0) \mid \forall i = 1, \dots, n$$

- Untermengenrelation [K1] für eine affine Zuweisung:

$$[K1] \quad \mathbf{K}(v) \supseteq \langle \llbracket x_i := t \rrbracket^{\#} \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} \text{ wenn } e = (u, v) \mid e \equiv x_i := t;$$

Mit der Untermengenrelation [K1] kann die an dem Programmpunkt  $v$  gültige konvexe Menge errechnet werden, wenn die Kante  $e = (u, v) \in Base$  eine affine Zuweisung bezeichnend analysiert wird. Dabei ist dieselbe Abstraktionsfunktion anzuwenden, wie bei [E1] mit dem Unterschied, dass diese nicht auf Matrizen, sondern Vektoren anzuwenden ist. Somit zeichnet sich der einzige Unterschied in der Verknüpfung des Effektes einer affinen Zuweisung mit der am Programmpunkt  $u$  gültigen Menge, die im Falle von  $\mathbf{E}$  aus Effektmatrizen und bei  $\mathbf{K}$  aus Vektoren besteht. Es ist auf der ersten Abstraktionsebene eine Matrizenmultiplikation und auf der zweiten Abstraktionsebene eine Matrix-Vektor-Multiplikation durchzuführen.

- Untermengenrelation [K2] für eine nicht affine Zuweisung:

$$[K2] \quad \mathbf{K}(v) \supseteq \langle \{l_i\} \cup \llbracket x_i := ? \rrbracket^{\#} \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} \text{ wenn } e = (u, v) \mid e \equiv x_i := ?;$$

Diese Untermengenrelation [K2] ist ganz analog zur Untermengenrelation [E3] definiert, nur dass hier unsere Generatoren als Vektoren dargestellt werden.

- Untermengenrelation [K3] für eine *skip*-Anweisung:

$$[K3] \quad \mathbf{K}(v) \supseteq \langle \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} \text{ wenn } e = (u, v) \mid e \equiv skip;$$

Diese ist ganz analog zu [E1] zu spezifizieren, so dass eine *skip*-Anweisung auch auf der nächsten Abstraktionsebene keine Transformation des Generators mit sich bringt.

- Untermengenrelation [K4] für einen Prozeduraufruf:

$$[K4] \quad \mathbf{K}(v) \supseteq \langle (\mathbf{E}(e) \cdot \mathbf{K}(u)), \mathbf{K}(v) \rangle_{conv} \text{ wenn } e = (u, v) \mid e \equiv x_i := f();$$

Bei der Untermengenrelation [K4] ist der Effekt des Funktionsaufrufs bereits in der vorangegangenen Analyse beschrieben worden, so dass an dieser Stelle lediglich die Effektmatrizen  $\mathbf{E}(e)$  zur Beschreibung des Effektes der Funktion auf die am Startknoten vorliegenden Generatoren  $\mathbf{K}(u)$  anzuwenden ist, um die am Zielknoten geltende konvexe Menge von Generatoren zu erhalten. Somit ist nur eine einfache Matrix- Vektor-Multiplikation mit den in der ersten Analyse errechneten Effektmatrizen vorzunehmen.

- Untermengenrelation [K5] für Wächter:

$$[K5] \quad \mathbf{K}(v) \supseteq \langle \llbracket t \diamond 0 \rrbracket^{\#} \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} \text{ wenn } e = (u, v) \mid e \equiv t \diamond 0;$$

Treffen wir in unserer Analyse auf eine Bedingung so muss diese ausgewertet werden. Der Effekt eines Wächters wird so spezifiziert:

$$\llbracket t \diamond 0 \rrbracket^{\#} X = x \mid x \in X \mid \{ \langle t \times x \diamond 0 \rangle \cup cut(X, t, \diamond) \}_{conv}$$

Die auf das Generatorensystem  $X$  vorzunehmende Transformation resultiert in einem Generatorensystem, bestehend aus allen Generatoren  $x \in X$ , welche die Bedingung erfüllen. Ausserdem ist noch die Schnittfläche zwischen der als Hyperebene zu interpretierenden Bedingung mit den Generatoren der konvexen Menge zu der Generatorenmenge der  $x \in X$  hinzuzufügen. Diese Schnittflächenberechnung wird von der Funktion  $cut$  durchgeführt, welche in 6.3 definiert wird.

- Untermengenrelation [A0] zur Auflösung der eigens eingeführten Bedingungsvariablen:

$$[A0] \quad \mathbf{A}(s) \supseteq \forall x_{u,v} \mid \llbracket x_{(u,v)} \diamond_{(u,v)} 0 \rrbracket^{\#} \mathbf{K}(s)$$

Diese Untermengenrelation erklärt die Auflösung der Bedingungsvariablen, welche in der ersten Analyse bei Vorkommen einer Bedingung speziell eingeführt wurden. Sobald die konvexe Menge des Zielpunktes angegeben werden soll, werden all diese Variablen aufgelöst und in das zugrundeliegende Generatorensystem geeignet eingebaut.

Nun wurde separat für jeden Typ Kante eine Untermengenrelation aufgestellt, um die Generatoren des Zielpunktes  $t$  ausgehend von einem Startpunkt  $s$  unter Durchführung der entsprechenden Transformation zu erhalten. Zusammenfassend erhalten wir folgendes Relationensystem, welches aufgelöst werden muss, um die an einem bestimmten Programmpunkt gültigen

Generatoren zu erhalten.

$$\begin{array}{ll}
 [\text{K0}] & \mathbf{K}(s_{start_f}) \supseteq p \cup \{l_i \mid l_i = (0, \dots, 1, \dots, 0) \in lines \mid \forall i\} \\
 [\text{K1}] & \mathbf{K}(v) \supseteq \langle [[x_i := t]]^{\#} \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv x_i := t; \\
 [\text{K2}] & \mathbf{K}(v) \supseteq \langle \{l_i\} \cup [[x_i := ?]]^{\#} \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv x_i := ?; \\
 [\text{K3}] & \mathbf{K}(v) \supseteq \langle \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv skip; \\
 [\text{K4}] & \mathbf{K}(v) \supseteq \langle (\mathbf{E}(e) \cdot \mathbf{K}(u)), \mathbf{K}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv x_i := f(); \\
 [\text{K5}] & \mathbf{K}(v) \supseteq \langle [[t \diamond 0]]^{\#} \mathbf{K}(u), \mathbf{K}(v) \rangle_{conv} & \text{wenn } e = (u, v) \mid e \equiv t \diamond 0; \\
 [\text{A0}] & \mathbf{A}(s) \supseteq \forall x_{u,v} \mid [[x_{(u,v)} \diamond_{(u,v)} 0]]^{\#} \mathbf{K}(s)
 \end{array}$$

Durch die Aufstellung dieses Relationensystems, welches die Anweisungssequenzen beschreibt, die zu einem bestimmten Programmpunkt führen, haben wir unsere operationale Semantik für den Kontrollflussgraphen spezifiziert.

## 5.2 Inkrementeller Fixpunktalgorithmus

Um Zeit und Kosten einer ständigen Neuberechnung der affinen Relationen zu sparen wurde hier im Gegensatz zu 4.4 zur Berechnung der in 5.1.2 und in 5.1.3 aufgestellten Teilmengenrelationensysteme ein **inkrementeller Fixpunktalgorithmus** verwendet. Die Effizienz liegt darin, nicht eine ganze konvexe Menge durch den Kontrollflussgraphen zu reichen, sondern immer nur über einzelnen Generatoren die Iteration durchzuführen. Somit ist an jedem einzelnen Knoten der zu propagierende Generator daraufhin zu überprüfen, ob der durch eine lineare Kombination der bereits existierenden Generatoren dargestellt werden kann. Kann dieser Generator durch die zugrundeliegende konvexe Menge ausgedrückt werden, wird er nicht mehr weiter durch den Kontrollflussgraphen gereicht.

Andernfalls besagt die Tatsache, dass der Generator nicht schon darstellbar ist, dass wir eine neue Information über die Wertebereiche der Programmvariablen gefunden haben. Deshalb muss der Generator als Erweiterung in die zugrundeliegende konvexe Menge aufgenommen werden. Weiterhin werden neue Generatoren über alle ausgehenden Kanten des derzeitigen Knotens unter Anwendung der entsprechenden Transformationsvorschrift über den Kontrollflussgraphen weitergegeben. Nun ist der in 5.1 vorgestellte inkrementelle Algorithmus noch dahingehend zu überprüfen, dass er nach endlich vielen Schritten terminiert, wie in 4.4. Das einzige Vorkommen einer Schleife stellt die *while*-Schleife in Zeile 14 dar.

```
while worklist  $\neq \emptyset$  do
```

Diese wird nur dann nicht mehr durchlaufen, wenn die mit Hilfe der Bedingung  $g_s \notin \sqcup GS[v]$  gefüllte *worklist* irgendwann mal leer ist. Ein Füllen der *worklist* wird nur dann vorgenommen, wenn das Generatorensystem  $GS[v]$  den zu propagierenden Generator nicht als lineare Kombination ihrer Generatoren darstellen kann. Sonst wird der Generator nicht weiterpropagiert.

```

// fixpointiteration with given Node v, given Generator g and
// predefined Set of Edges E and Nodes N
Set fixpointiteration (Node v, Generator g, Set E, Set N) {
4 //Initialisation
  Set[] GS = new Set(|N|);
  forall n ∈ N do
    GS[n] := ∅
  od
9
  //create a new worklist with visited node and calculated GeneratorSystem
  Set worklist := {(v,g)};

  //naive fixpointiteration
14 while worklist ≠ ∅ do
  let u_s ∈ worklist
  begin
    (u_s, g_s) := extract(worklist);
    Set toPropagate := ∪ GS[v].append(g_s, threshold)
19
    if (toPropagate ≠ ∅) {
      //append new generator g_s to prior GeneratorSystem
      GS[v] := append(GS[v], {g_s}, threshold);
      //skip-Edge
24 forall (u,v) ∈ E | A((u,v)) ≡ skip do
        worklist := worklist ∪ {g_s, v};
      od
      //linear Assignment-Edge
      forall (u,v) ∈ E | A((u,v)) ≡ x_i := t do
29        worklist := worklist ∪ {(g_s)[x_i t], v};
      od
      //non linear Assignment-Edge
      forall (u,v) ∈ E | A((u,v)) ≡ x_i :=? do
34        worklist := worklist ∪ {(π_i(g_s) ∪ l(x_i)), v};
      od
      //linear Assertion-Edge
      forall (u,v) ∈ E | A((u,v)) ≡ c△0 do
        worklist := worklist ∪ {(c(g_s)△0), v};
      od
39 fi
  end
  od
}

```

Abbildung 5.1: inkrementeller Fixpunktalgorithmus

Es muss also nach einer endlichen Anzahl von Durchläufen durch den Kontrollflussgraph ein Generatorensystem erzeugt worden sein, das alle Generatoren ausdrücken kann, die an diesem

Programmzustand gelten sollen. Diese Terminierung der *while*-Schleife, so dass keine neuen Aufträge mehr zur *worklist* hinzugenommen werden, kann mit *Widening* 5.2.1 erreicht werden.

### 5.2.1 Widening

Wie in 4.4.1 angedeutet, ist die Technik des Widenings bei der Fixpunktiteration anzuwenden, damit der Algorithmus terminiert, was jedoch Präzisionsverlust bedeutet. In der Implementation des inkrementellen Fixpunktalgorithmus, wurde eine einfache Version des Widenings eingebaut. Diese Technik kommt an einem *Vereinigungsknoten* zur Anwendung, wo mehrere Kanten zusammenfließen, gesetzt dem Fall, dass die Punktmenge des an diesem Knoten zugrunde liegenden Generatorensystems einen bestimmten **Schwellwert** *threshold* überschreitet. Als Grenzwert wurde in der Implementation der vorliegenden Analyse die Anzahl der Variablen der jeweiligen zu analysierenden Prozedur bemessen.

Sobald das an einem Programmpunkt vorliegende Generatorensystem zu viele Punkte fasst, wird mit Hilfe von Widening ein Strahl  $r_w$  erzeugt, was zu einer Vergrößerung des konvexen Bereichs führt.

Es ist offensichtlich, dass sich die Punktmenge des Generatorensystems nur vergrößern kann, wenn ein weiterer Punkt  $p$  in das Generatorensystem aufgenommen wird, der durch seine Hinzunahme zu einem neuen konvexen Objekt führt. Damit man nicht möglicherweise unbegrenzt viele Durchläufe des Fixpunktalgorithmus mit Punkten, die sich nur in einigen Komponenten unterscheiden, erzielt, soll ein Strahl, der diese Abhängigkeit der Punkte darstellt, weiterpropagiert werden. Da der Punkt  $p$  neue Information bedeutet und als Produkt eines Schleifendurchlaufs als lineare Kombination seiner *Vorgängerknoten* aus vorherigen Durchläufen  $p = \sum_{i=1}^n \lambda_i p_i$  darstellbar ist, wird daraus ein Strahl gebildet, dessen Entstehung in der Funktion *evolveRay* beschrieben wird.

```
//evolve ray of linear dependent points
Ray evolveRay (Set gs, Point p) {
    return newRay(p - ( $\exists p_{max} \in p_1, \dots, p_n \in points \mid p = \sum_i \lambda_i p_i \mid \lambda_{max} maximal$ ));
}
```

Abbildung 5.2: Generierung eines Strahls aus linear abhängigen Punkten

Zudem muss Widening auch dann eingesetzt werden, wenn die Hinzunahme eines Punktes in die konvexe Menge genau einen alten Punkt ersetzt. Diese Tatsache wird in der Algorithmusbeschreibung der Funktion *append()* in 6.9 dargelegt. Der Bereich den die Generatoren der konvexen Menge aufspannen wird durch die vorgenommene Punktersetzung grösser, aber die Generatorenmenge wird nicht stärker. Denn wie in 6.3 dargestellt, sind die Linien die do-

```
Ray widen(Set gs, Point p) {  
    return evolveRay(gs, p);  
}
```

Abbildung 5.3: Prozedur *widen()* zur Durchführung des Widenings

minantesten Generatoren, da sie den grössten Bereich aufspannen, darunter die Strahlen und auf unterster Dominanzebene befinden sich die Punkte. Somit muss an dieser Stelle Widening angewendet werden, so dass auch hier ein Strahl entsteht. Die eigentliche Funktionalität des Widenings wird von der austauschbaren Funktion *evolveRay()* durchgenommen, welche aus linear abhängigen Punkten einen Strahl erzeugt.

Um herauszufinden welche Punkte des Generatorsystems linear abhängig sind, kann ein LP Problem 7.2 formuliert werden. Als Antwort auf das in 6.5.4 definierte Optimierungsproblem erhalten wir für jeden Punkt  $p_i$  ein  $\lambda_i$ . Es wird der Punkt  $p_{max}$  mit maximalem  $\lambda_{max}$ -Wert gewählt und durch Subtraktion von  $p$  der Strahl  $r_w$  gebildet, welcher nun genau die Abhängigkeit der Punkte darstellt.



## 6 Realisierung

Im Rahmen dieser Diplomarbeit entstand ein erster Prototyp für die in 5 beschriebene Konzeption der interprozeduralen Programmanalyse. In diesem Kapitel soll eine kurze Beschreibung der Implementation der theoretischen Konzepte gegeben werden, welche anschliessend an Hand einer algorithmischen Darstellungsweise verdeutlicht werden. Bis auf die Implementation der beiden Untermengenrelationen  $A[0]$  und  $E[5]$ , welche sich auf die Behandlung von Bedingungen in der ersten Analyse bei der Darstellung von Effektmatrizen beziehen, wurde die theoretische Beschreibung der interprozeduralen Analyse in Java umgesetzt. In einzelnen Abschnitten wird dabei auf die Funktionen näher eingegangen:

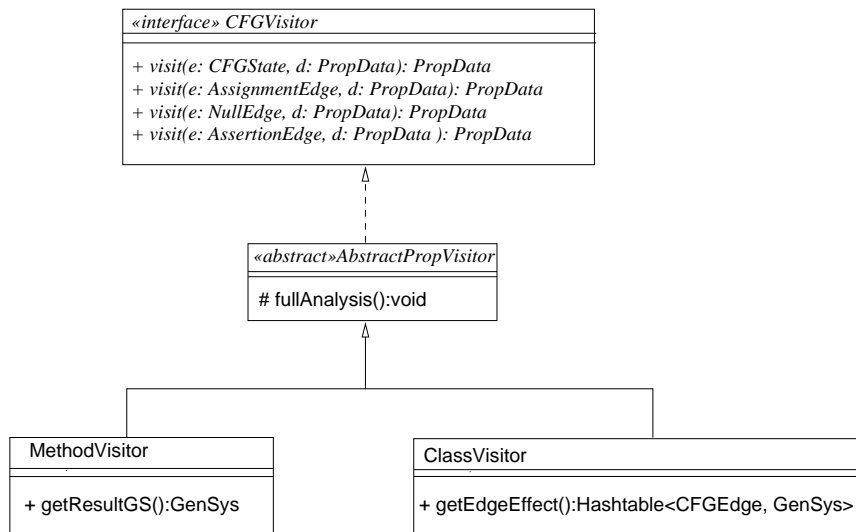
- *convexHull* zur Bildung der konvexen Hülle,
- *cut* was die Schnittflächenberechnung bei Bedingungskanten durchführt
- *append* eine Methode, welche die Anwendung von *widen* und *convexHull* kapselt

Da sich bei der mathematischen Definition dieser Funktionen einige Problemstellungen ergaben, die sich auf ein LP Problem zurückführen lassen, wird in 6.5 deren Formulierung erläutert.

Abschliessend erfolgt noch in 6.6 eine Leistungsanalyse des Prototypen an gewählten Beispielen und sich daraus ergebenden Verbesserungsmöglichkeiten.

### 6.1 Implementation

In der Verifikation der theoretisch beschriebenen interprozeduralen Analyse mittels linearer Ungleichungen wurde diese Programmanalyse, deren Arbeitsweise und Besonderheiten an dieser Stelle kurz erläutert werden, in der Programmiersprache Java implementiert.



Dabei wurden aufbauend auf die in 5 spezifizierte Architektur zwei Fixpunktiterationen entsprechend des Ablaufs aus 5.2 ausimplementiert. Diese beiden Visitoren sollen die Kanteneffekte mit Hilfe der Datenstruktur Generatorensystem darstellen. Die Notwendigkeit zweier verschiedener Fixpunktiterationen liegt in der differenzierten Betrachtungsweise der Generatoren.

Im ersten Teil der Analyse, realisiert durch *ClassVisitor* werden mit den Generatoren die Transformationsmatrizen für den jeweiligen Programmzustand beschrieben, wohingegen bei der zweiten Fixpunktiteration *MethodVisitor* die Relationen zwischen den Programmvariablen durch die abstrahierte Darstellungsweise von Punkten, Strahlen und Linien als Vektoren erfolgt. Dadurch kann an jedem beliebigen Programmzustand der Wertebereich für die einzelnen Programmvariablen angegeben werden.

Die Aufgabe des *ClassVisitors* ist die Berechnung von Effektmatrizen für die jeweiligen Kanten, so dass ganz leicht für jeden eingehenden Generator durch Anwendung dieser Transformationsmatrizen die am Ausgang geltende Relation beschrieben werden kann. Dabei wird von jeder Methode der zu analysierenden Klasse eine Abstraktion derart kalkuliert, dass die Abhängigkeit der Information am Methodenende mit der zu Beginn der Methode geltenden Information dargestellt wird. Das in Abbildung 6.1 präsentierte UML-Diagramm soll einen kleinen Einblick in die Implementierung der Matrizenobjekte geben.

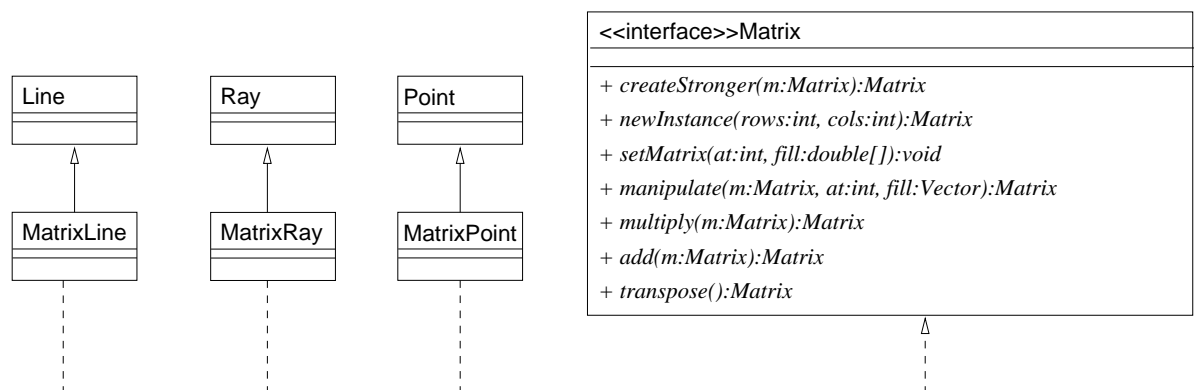


Abbildung 6.1: Darstellung der Matrizenobjekte

Auf diese Effektberechnung kann bei der Fixpunktiteration innerhalb des *MethodVisitors* an den Aufrufstellen der entsprechenden Prozedur zurückgegriffen werden, so dass der *MethodVisitor* eine ganz einfache intraprozedurale Analyse auf dem Kontrollflussgraphen der zu untersuchenden Funktion beschreibt. Abbildung 6.2 soll die Vektorenobjekte, die innerhalb des *MethodVisitors* an den Programmpunkten abgespeichert werden, nochmals betonen.

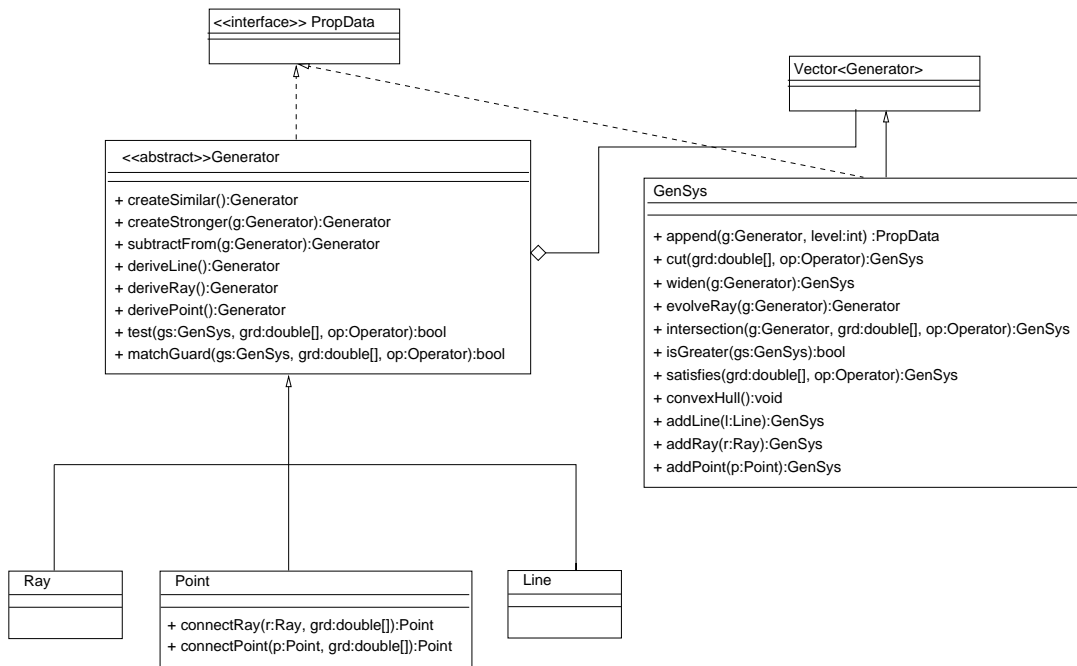


Abbildung 6.2: Darstellung der konvexen Mengen als Vektoren

Um die Grundoperationen auf den konvexen Mengen ausführen zu können, wurden die mathematischen Objekte der Generatoren auf Java-Seite geeignet repräsentiert. Man verwendet die beiden Datenobjekte eines Generatorsystems in Form eines Vektors über den Generatoren und in Form von Effektmatrixen.

Da die in 5.2 beschriebene Fixpunktiteration in jedem Schritt überprüft, ob ein zu propagierender Generator bereits durch das vorliegende Generatorsystem ausgedrückt werden kann, ist eine spezielle Operation nötig, die diese geforderte Funktionalität besitzt. Man kann dieses spezielle Problem auf eine allgemeinere Problemstellung zurückführen, was mit Hilfe von **Linear Programming** gelöst werden kann, wie in 7.2.3 beschrieben. Hierfür wurde eine Fassade für den LP Solver *glpk* konstruiert, so dass das Problem der konvexen Hülle sehr einfach mit dieser Wrapperfunktionalität, die *glpk* einnimmt, berechnet werden kann.

Die Implementierung dieses Prototypen umfasst viele Funktionen im Umgang mit den konve-

nen Mengen zur Repräsentation der Programzustände, von denen hier nur die Realisierung der essentiellen Gesichtspunkte vorgestellt wird.

## 6.2 *convexHull()*

Die in unserer Analyse berechneten konvexen Mengen stellen die Repräsentation eines Programzustandes dar. Es ist wünschenswert an jedem Programmpunkt eine minimale Generatorenmenge abzuspeichern, die jedoch den darzustellenden Vektorraum vollständig beschreibt. Da die Bildung der konvexen Hülle über unser Generatorsystem zu einer einfacheren und kleineren konvexen Menge führt, wird auch der Aufwand in der Anwendung der Transformationsfunktionen auf die einzelnen Generatoren reduziert.

Das Ziel der konvexen Hülle ist es ein Generatorsystem zu erhalten, welches nur die Basisgeneratoren für den aufzuspannenden Körper enthält. In diesem Abschnitt werden die realisierungstechnischen Aspekte und die Funktionsweise der Prozedur *convexHull()* aus unserem Prototypen kurz dargelegt.

Um die konvexe Hülle eines Generatorsystems bestehend aus Punkten, Strahlen und Linien

```
Set convexHull (Set gs) {  
2  Set result := ∅;  
  forall (l ∈ gs_lines)  
    result := result ∪ addLine(l, gs);  
  
  forall (r ∈ gs_rays)  
7   result := result ∪ addRay(r, gs);  
  
  forall (p ∈ gs_points)  
    result := result ∪ addPoint(p, gs);  
  
12 return result;  
}
```

Abbildung 6.3: Prozedur *convexHull()* zur Durchführung der konvexen Hülle

zu bilden, wird ein minimales Generatorsystem iterativ aufgebaut. Zu Beginn hat man ein leeres Generatorsystem, welches sukzessive mit den Generatoren des ursprünglichen Generatorsystems *gs* zu füllen ist. Es werden nacheinander die Generatoren von *gs* entnommen und in das Ergebnisgeneratorsystem *result* nur dann eingefügt, wenn sie nicht schon als Linearkombination bereits enthaltener Elemente darstellbar sind. Diese Überprüfung kann unter Formulierung eines LP Problems vorgenommen werden, was in 6.5.1 ausführlich dargestellt wird.

Hierbei muss die Iteration zuerst auf oberster Ebene die Menge der Linien *lines* durchlaufen

und nur die Linien  $l \in \text{lines}$  in das minimale System aufnehmen, welche linear unabhängig sind. Diese Überprüfung wird in der Prozedur `addLine()` vorgenommen und garantiert, dass nur linear unabhängige Linien in `result` abgelegt werden. Da die Linien am dominantesten sind, müssen diese zuerst untersucht und linear abhängige Linien entfernt werden. Die Linien können als bidirektionale Strahlen gesehen werden und beschreiben somit separat betrachtet den grössten Bereich.

Weiterhin wird die Menge der Strahlen `rays` in Augenschein genommen, welche in das minimale Generatorensystem aufgenommen werden dürfen. Dabei ist zu beachten, dass ein Strahl  $r$ , der eine Linearkombination anderer Strahlen darstellt, als Linie mit `createLine(r)` zu unserem System hinzugenommen wird, sofern diese Linie nicht durch die schon im Ergebnisgeneratorensystem `result` enthaltenen Linien darstellbar ist. Andernfalls ist eine Abprüfung auf linear abhängige Strahlen durchzuführen. Diese Funktionalität wird in der Methode `addRay()` nachfolgend dargelegt. In der Abbildung soll die doch etwas komplexere Funktion `addRay()` vorgestellt werden. Die Antwort auf die Frage nach der Existenz einer Linearkombination der

```

Set addRay(Ray r, Set gs) {
2  Set result := 0;
  if ( (|gs| < 1) || (|rays| < 1) ) {
    result := result ∪ {r};
    gs := gs ∪ {r};
  } else {
7   if (r ∉ {lines ∪ rays})
     if (∃ r1, ..., ri ∈ rays | r = ∑ λi · ri) {
       //there are linear dependent rays
       forall (ri | λi < 0, i = 1, ..., n)
         Linel := convertToLine(ri);
12        result := result ∪ {l};
         gs := gs ∪ {l} \ {ri};
       } else {
         result := result ∪ {r};
         gs := gs ∪ {r};
17        //repeated pass through gs to remove redundant generators
         forall (g ∈ gs)
           gs := gs \ {g};
           if ( (|gs| = 0) || (g ∉ gs) ) {
22             gs := gs ∪ {g};
           }
         }
       }
     }
   }
   return result;
27 }

```

Abbildung 6.4: Prozedur `addRay`

Strahlen und einer Angabe der Koeffizienten einer gültigen Linearkombination wird in 6.5.3 erläutert. Analog dazu ist der Aufbau der Methoden `addLine()` und `addPoint()`, wobei ein Unterschied in der Formulierung der Inklusion liegt und der Inklusionstest aus Zeile 8 bei Linien und Punkten nicht durchlaufen wird.

```
//in addLine(Line l, Set gs)
if (l ∉ lines)
```

Abbildung 6.5: Inklusionstest von `addLine()`

```
//in addPoint(Point p, Set gs)
if (p ∉ {lines ∪ rays ∪ points})
```

Abbildung 6.6: Inklusionstest von `addPoint()`

Als letztes werden die Punkte *points* betrachtet, wohingegen nur die linear unabhängigen zuzufügen sind, die also nicht als Linearkombination von Punkten, Strahlen und Linien darstellbar sind.

Für die eben geschilderten zu revidierenden Inklusionsüberprüfungen wurde eine Reihe von Tests mit *glpk* implementiert, die für jeden Generator durchzuführen sind. Unter Anwendung des **Erfüllbarkeitsproblems** von LP 7.2 kann für jeden Generator festgestellt werden, ob er sich bereits als Linearkombination anderer Generatoren ausdrücken lässt. Ist dies der Fall, so befindet sich der Generator schon innerhalb des konvexen Bereichs und darf nicht zu den Basis bildenden Generatoren hinzugenommen werden. In Abschnitt 6.5.1 wird diese Abprüfung auf das Vorhandensein einer Linearkombination als ein entsprechendes LP Problem formuliert.

### 6.3 *cut()*

Stösst man beim Traversieren des Kontrollflussgraphen auf eine Bedingung *guard*, so müssen die Schnittflächen mit der Hyperebenengleichung *guard*, welche die einzuhaltende Bedingung geometrisch repräsentiert, ermittelt werden. Dabei muss das bereits vorliegende Generatorsystem *gs* in die Generatoren die diese Bedingung erfüllen  $gs_{guard}$  und solche die die Bedingung nicht erfüllen  $gs_{\overline{guard}}$  aufgeteilt werden.

Bei der Betrachtung der Generatorenart der Linien sind zwei Fälle zu unterscheiden.

Zum einen kann eine Linie *l* parallel zur Hyperebene verlaufen und wird diese somit niemals schneiden. Diese Linie *l* spielt für die Berechnung von Schnittpunkten keine Rolle und kann ausser Betracht gelassen werden.

Andernfalls kann eine Linie  $l$  die Hyperebene  $guard$  schneiden und wird deshalb in zwei Strahlen mit  $createRay(l)$  und  $createRay(-l)$  aufgespalten, welche gemäss eben genannter Aufspaltung in die jeweiligen Generatorenmengen  $gs_{guard}$ ,  $gs_{\overline{guard}}$  einzufügen sind. Diese Aufspaltung der Linie in zwei Strahlen kann durchgeführt werden, da eine Linie einen bidirektionalen Strahl verkörpert. Damit ergeben sich für unsere Schnittpunktberechnung vier zu betrachtende Mengen. Diese sind jeweils miteinander zu kombinieren, um Schnittpunkte mit der Hyperebene zu erlangen.

	$Points_{\overline{guard}}$	$Rays_{\overline{guard}}$
$Points_{guard}$	×	×
$Ray_{guard}$	×	×

Tabelle 6.1: Kombination von  $gs_{guard}$  und  $gs_{\overline{guard}}$

Ein Generator  $g$  erfüllt die Bedingung, genau dann, wenn er sich in der Hyperebene befindet. Mathematisch betrachtet bedeutet dies, dass der Generator in die Hyperebenengleichung eingesetzt wird und gemäss des Operators der Bedingung eine Abprüfung auf 0 ausgeführt wird. Ist diese Prüfung erfolgreich, so befindet sich  $g$  in der Hyperebene.

Zur Beantwortung der Fragestellung, ob der Generator genau auf der Hyperebene liegt, muss bei der Abprüfung auf 0 der Bedingungsoperator durch den Gleichheitsoperator ersetzt werden. Nur ein in der Hyperebene oder ein sich auf dem erfüllenden Halbraum befindlicher Generator

```
bool matchesGuard(Guard guard, Operator op, Generator g) {
    return op.compare( $\sum_i guard_i \cdot g_i$ , 0)
}
```

Abbildung 6.7: Überprüfung ob Strahlen eine Bedingung erfüllen

darf im Kontrollflussgraphen weitergegeben werden.

Wie diese Schnittfläche errechnet wird, verdeutlicht der vorliegende Algorithmus 6.8. Ist der Generator ein Punkt  $p_{\overline{guard}}$  so muss der Punkt mit allen anderen Punkten  $p_i \in gs_{guard}$  verbunden werden. Die daraus resultierenden Schnittpunkte der Strecke mit der Hyperebene werden in dem Ergebnisgeneratorensystem  $result$  abgespeichert.

Als nächstes müssen alle Punkte  $p \in p_{\overline{guard}}$  als Aufpunkte aller Strahlen  $r_i \in gs_{guard}$  gesetzt werden und wiederum werden die Schnittpunkte mit der Hyperebene ermittelt, die dann  $result$  erweitern.

Dasselbe Verfahren ist auf die inversen Punkte und inversen Strahlen anzuwenden. Nun sind noch die beiden Strahlenmengen zu kombinieren, um einen Schnittpunkt in Form eines Strahls

```

Set cut (Set gs, Guard guard, Operator op) {
2  Set result := 0;
   //combination of all good points with all bad points
   //to obtain an intersection point with the hyperplane guard
   forall (pg ∈ Points_guard)
     forall (pb ∈ Points_guard)
7    λ := - $\frac{\sum_i \text{guard}_i \cdot p_{b_i}}{\sum_i \text{guard}_i \cdot p_{g_i}}$ ;
       result := result ∪ new Point (λ · (pb - pg) + pg);

   //combination of all good points with all bad rays
   //to obtain an intersection point with the hyperplane guard
12  forall (pg ∈ Points_guard)
     forall (rb ∈ Rays_guard)
       λ := - $\frac{\sum_i \text{guard}_i \cdot p_{g_i}}{\sum_i \text{guard}_i \cdot r_{b_i}}$ ;
       result := result ∪ new Point (λ · rb + pg);

17  //combination of all bad points with all good rays
   //to obtain an intersection point with the hyperplane guard
   forall (pb ∈ Points_guard)
     forall (rg ∈ Rays_guard)
       λ := - $\frac{\sum_i \text{guard}_i \cdot p_{b_i}}{\sum_i \text{guard}_i \cdot r_{g_i}}$ ;
22  result := result ∪ new Point (λ · rg + pb);

   //combination of all bad rays with all good rays
   //to obtain an intersection ray with the hyperplane guard
   forall (rb ∈ Rays_guard)
27  forall (rg ∈ Rays_guard)
       result := result ∪ new Ray (λ1 · rb + λ2 · rg)
         | λ1 ·  $\sum_i \text{guard}_i \cdot r_{b_i}$  + λ2 ·  $\sum_i \text{guard}_i \cdot r_{g_i}$  = 0 ∧ λi ≥ 1;
       convexHull(result);
   return result;
32 }

```

Abbildung 6.8: Berechnung der Schnittflächen mit der Hyperebene

zu erhalten. Die Linearkombination  $r$  zweier Strahlen  $\mu_1 \cdot r_{\text{guard}} + \mu_2 \cdot r_{\text{guard}}$  kann wieder unter zu Hilfenahme von LP erstellt werden. Dabei werden Werte für  $\mu_1, \mu_2$  berechnet. Diese Problemstellung wird in 6.5.2 formuliert.

## 6.4 append()

Um stets ein minimales Generatorsystem an jedem Programmpunkt zu führen und an den Vereinigungsknoten Widening anzuwenden, wurde die Methode *append()* implementiert, die diese Funktionalität übernimmt. Der Aufruf dieser Funktion erfolgt an jedem besuchten Kno-

ten des Kontrollflussgraphen.

Der Algorithmus von *append()* wird nachfolgend dargestellt. Innerhalb von *append()* wird

```

GeneratorSystem append(GeneratorSystem gs, Generator g, int threshold) {
2  Set result := ∅;
   if (g ∈ gs) {
       return result;
   } else {
       Set tmp := gs ∪ {g};
7      Set changes := convexHull(tmp);
       //new generator system is greater -> gain of information
       if (gs ≤ tmp) {
           //do widening if there are too many points
           if (|tmp.getPoints()| ≥ threshold) {
12              tmp := tmp \ {g}
                  changes := addRay(tmp ∇ g, tmp);
           }
       } else {
           //system has not become greater -> widening to stabilise
17          //fixpoint iteration
           tmp := tmp \ {g}
           changes := addRay(tmp ∇ g, tmp);
       }
   }
22  gs := tmp;
   return changes;
}

```

Abbildung 6.9: Prozedur *append()* mit Widening und Bilden der konvexen Hülle

überprüft, ob die Hinzunahme des Generators  $g$  in das an diesem Knoten vorliegende Generatorsystem  $gs$  zu einem Informationsgewinn führt und diese konvexe Menge vergrößert. In diesem Fall ist das Generatorsystem um  $g$  zu erweitern. Überschreitet durch die Hinzunahme von  $g$  die Punktmenge *points* einen bestimmten Schwellwert, so muss Widening angewendet werden, wie in 4.4.1 beschrieben. Als Schwellwert wird hier die Anzahl der im Programm vorkommenden Variablen verwendet.

Falls die Hinzunahme des Generators das bestehende System nicht vergrößert hat, was genau dann eintritt, wenn  $g$  einen schon enthaltenen Generator ersetzt, kommt ebenfalls die Technik des Widening zur Anwendung, um die Terminierung des Fixpunktalgorithmus 5.2 sicher zu stellen. Eine kurze Anmerkung zu der Bedingung aus Zeile 9 des Algorithmuses der Prozedur *append()*.

```

1  if (gs ≤ tmp) {

```

Dieser Vergleich der Grösse zweier Generatorensysteme darf nicht auf dem Vergleich der Anzahl der Elemente von  $|gs|$  mit der Anzahl der Generatoren in  $|tmp|$  beruhen. Denn dabei ist die Priorität, die die einzelnen Generatoren haben, mit zu berücksichtigen. Da die Linien am dominantesten sind, muss zuerst die Anzahl der Linien von  $gs$  mit der Linienmenge von  $tmp$  verglichen werden. Ist hier eine strikte Ungleichheitsbeziehung gegeben, so ist der Test beendet, andernfalls muss derselbe Vergleich bei den niederpriorien Generatoren umgesetzt werden.

## 6.5 LP Problemstellungen

Im Umgang mit den konvexen Mengen 3.4.1 ergaben sich einige mathematische Probleme wie z.B. "Lässt sich ein Punkt als Linearkombination anderer Generatoren ausdrücken?". Dabei kann auf die beiden Problemstellungen des **Optimierungsproblems** und des **Erfüllbarkeitsproblems** aus LP zurückgegriffen werden, um eine Lösung für unsere Fragestellungen zu erhalten.

Hier wird ein kleiner Überblick über die Formulierung der mathematischen Probleme in lineare Gleichungssysteme gegeben, welche dann unter Verwendung des LP Solvers *glpk* 7.2.3 die gewünschte Antwort liefern.

### 6.5.1 Darstellbarkeit der Generatoren

Da man an jedem Programmpunkt eine minimale konvexe Menge speichern möchte, ist ein Enthaltenseinstest erforderlich. Dieser soll für den zu prüfenden Generator angeben, ob er sich bereits als Linearkombination der Generatoren des vorliegenden Generatorsystems ausdrücken lässt. Diese Formulierung eines LP Problems findet Anwendung bei der Funktion der konvexen Hülle in 6.2, wobei für jeden einzelnen Generator ein Inklusionstest bewerkstelligt werden muss.

Für Punkte, Strahlen und Linien sind separate **Erfüllbarkeitsprobleme** anzugeben. Denn eine *ja - nein*-Antwort ist bei einer Verwendung dieses Tests vollkommen ausreichend.

- Punkt  $p$ :

Für einen Punkt  $p$  ist zu testen, ob er sich als Linearkombination der zugrunde liegenden Punkte, Strahlen und Linien ausdrücken lässt, wobei folgende Bedingungen einzuhalten sind: ( $p = \sum_{i=1}^m \lambda_i p_i + \sum_{i=m+1}^n \mu_i r_{i-m} + \sum_{i=n+1}^o \eta_i l_{i-n}$ )

$$\forall i \leq m \quad 0 \leq \lambda_i \leq 1,$$

$$\forall m < i \leq n \quad 0 \leq \mu_i \leq +\infty,$$

$$\forall n < i \leq o \quad -\infty \leq \eta_i \leq +\infty$$

$$\sum_{i=1}^m 1 \cdot \lambda_i + \sum_{i=m+1}^n 0 \cdot \mu_i + \sum_{i=n+1}^o 0 \cdot \eta_i = 1$$

Dieses Problem muss minimiert werden.

- Strahl  $r$ :

Bei einem Strahl  $r$  ist die Existenz einer Linearkombination der Strahlen und Linien zu validieren, was wie folgt zu spezifizieren ist: ( $r = \sum_{i=1}^m \mu_i r_i + \sum_{m+1}^n \eta_i l_{i-m}$ )

$$\forall 1 < i \leq m \quad 0 \leq \mu_i \leq +\infty,$$

$$\forall m < i \leq n \quad -\infty \leq \eta_i \leq +\infty$$

Auch dieses Problem ist zu minimieren.

- Linie  $l$ :

Eine Linie  $l$  ist dann linear unabhängig, wenn sie nicht als Linearkombination anderer Linien repräsentierbar ist.

$$(l = \sum_{i=1}^n \eta_i l_i)$$

$$\forall i \leq n \quad -\infty \leq \eta_i \leq +\infty$$

Auch hier ist das aufgestellte LP Problem zu minimieren, um geeignete Werte für die  $\lambda_i$  zu erhalten.

Findet der LP Solver adäquate  $\lambda_i, \mu_i, \eta_i$ , die sowohl innerhalb der vorgegebenen Grenzen liegen als auch das lineare Gleichungssystem erfüllen, so lässt sich der Generator als Linearkombination darstellen und ist linear abhängig.

### 6.5.2 Lineare Abhängigkeit zweier Strahlen bei der Analyse von Bedingungen

Um in unserer interprozeduralen Analyse die Auswirkung einer Bedingungskante zu erhalten, sind u.a. die Strahlen aus  $r_{guard}$  und  $\overline{r_{guard}}$  miteinander zu kombinieren, so dass eine Projektion in Form eines Strahls angegeben werden kann. Die Kombination zweier Strahlen bedeutet eine Linearkombination derselben zu bilden, so dass diese, eingesetzt in die Hyperbenengleichung, eben diesen Projektionsstrahl liefern. Diese Problemstellung lässt sich als **Optimierungsproblem** ausdrücken, das als Ergebnis geeignete Werte für  $\mu_1, \mu_2$  ausgibt.

$$(\mu_1 \cdot r_1 + \mu_2 \cdot r_2 = 0)$$

$$\lambda_i \geq 1$$

Dieses Problem muss minimiert werden.

### 6.5.3 Lineare Abhängigkeit bei Strahlen

Wie bereits erwähnt soll das Generatorensystem, auf dem die Programmanalyse arbeitet, möglichst klein bleiben. Diese Problemstellung taucht bei Anwendung der Funktion der konvexen Hülle in 6.2 auf. Um nicht beliebig viele Strahlen in das Generatorensystem aufzunehmen, werden Strahlen in Linien umgewandelt. Dazu ist zu ermitteln, ob ein Strahl  $r$  durch eine bereits vorliegende Menge von Strahlen  $r_1, \dots, r_n$  als Linearkombination

$$r = \mu_1 r_1 + \dots + \mu_n r_n$$

dargestellt werden kann.

Ausgangspunkt für dieses LP Problem ist ein Strahl  $r$  und eine Menge von Strahlen  $r_i$ , wobei man wissen möchte ob  $r$  als Linearkombination von Strahlen  $r_j \mid j \leq i$  realisierbar ist.

Um eine Linearkombination der  $r_j$  angeben zu können, muss ein **Optimierungsproblem** formuliert werden, welches wiederum geeignete Werte für  $\mu_i$  liefert und wie folgt ausgedrückt werden kann:

$$(r = \sum_i \mu_i \cdot r_i)$$

$$-\infty \leq \mu_i \leq +\infty$$

Hier liegt eine Maximierung des aufgestellten Systems vor.

### 6.5.4 Lineare Abhängigkeit bei Punkten

Für die Funktion des Widenings aus 5.2.1, die aus zwei linear abhängigen Punkten einen Strahl bildet, ist es nötig die linear abhängige Punktemenge des Generatorensystems zu kennen. Ebenso wie in 6.5.3 muss für die Ermittlung einer Linearkombination von Punkten  $p_i$  ein **Optimierungsproblem** angegeben werden, welches maximiert werden muss. Dieses liefert passende Werte für die  $\lambda_i$ .

$$(p = \sum_i \lambda_i \cdot p_i)$$

$$-\infty \leq \lambda_i \leq +\infty$$

$$\sum_i \lambda_i = 1$$

Als Ergebnis erhalten wir für jeden Punkt  $p_i$  ein  $\lambda_i$ , so dass deren Kombination den Punkt  $p$  repräsentiert.

## 6.6 Leistungsanalyse

In diesem Abschnitt wird eine Auswertung von elementaren Beispielprogrammen vorgenommen, um die Aussagekraft und Leistungsfähigkeit der Realisierung der vorgestellten Konzeption zu bewerten. Da die vorgenommene Recherche keine geeigneten Testdateien als Eingabe für unsere Analyse oder gar Analyseergebnisse zum Vergleich ergab, wurde die Versuchsreihe an selbst geschriebenen und eigens gewählten Beispielprogrammen durchgeführt und konnte in ihrer Leistungsbewertung nicht mit anderen "Vergleichswerten" gleich gesetzt werden.

Ausgangspunkt für diese kleine Testreihe bilden die Kontrollflussgraphen, die bereits separat für alle betrachteten Methoden der zu analysierenden Programmklasse vom Kontrollflussgenerator generiert wurden. Nun werden einzelne Beispiele, mittels derer die Implementation der in 5 dargelegten Konzeption getestet wurde, und ihre Ergebnisse kurz beschrieben.

Zuerst werden die einzelnen Programmkonstrukte wie Schleifen und Methodenaufrufe separat betrachtet, um deren Aussagekraft und Auswirkung auf das Generatorensystem klar aufzuzeichnen. Anschliessend erfolgt erst eine Überprüfung der Leistungsstärke unserer Analyse anhand eines Programms, welches eine Kombination der einzelnen Konstrukte beinhaltet. Abschliessend wird noch anhand eines Anwendungsbeispiels ein möglicher Einsatzbereich dieses Prototypen aufgezeigt. Bei den nachfolgend beschriebenen Testfällen wird die konexe Menge, die am Endpunkt der zu analysierenden Methode gilt, als Ergebnis der Analyse betrachtet.

### 6.6.1 Testfall: Schleife

Dieses Beispielprogramm dient der spezialisierten Betrachtung von Schleifen, die ja in Programmen ein sehr häufig vorkommendes Sprachkonstrukt darstellen.

In diesem kleinen Testprogramm ohne interprozedurale Aktion soll die lineare Funktion *loop* analysiert werden. Dabei wird die Variable *x* in einer Schleife bei jedem Schleifendurchlauf um eins hochgezählt und sobald die Schleifenbedingung unerfüllt bleibt ihr Wert zurückgegeben. In der Tabelle 6.11 wird das Generatorensystem graphisch dargestellt, welches nach Terminierung der Fixpunktiteration des *MethodVisitor* auf dem Kontrollflussgraphen der Methode *loop* den Endzustand der Methode charakterisieren soll. Wie formal spezifiziert kann jeder Programmvariablen eine Komponente des Vektors zugeordnet werden, wobei die erste Komponente jeweils *return* bezeichnet und die letzte Komponente den konstanten Term angibt. In der vorliegenden Implementation der Analyse wurde *return* als eine globale Variable festgelegt, so dass die Programmanweisung *return x*; als eine Zuweisung *return = x*; des Wertes der Variablen *x* an *return* interpretiert wird.

Damit der Fixpunktalgorithmus des *MethodVisitors* terminiert, muss beim Auftreten von Schleifen (Zeile 7) Widening angewendet werden, wodurch die Linie und der Strahl des Ergebnissgeneratorensystems entstehen. Diese beiden Generatoren vergrössern den Ergebnisbereich drastisch, so dass schon bei diesem kleinen Testprogramm mit zwei Variablen ein unpräzises

```

public class LoopTest{
    int loop(){
4       int x, y;
        x=1;
        y=1;
        while ( y < 10){
            y=y+1;
9         x=x+y;
        }
        return x;
    }
}

```

Abbildung 6.10: LoopTest

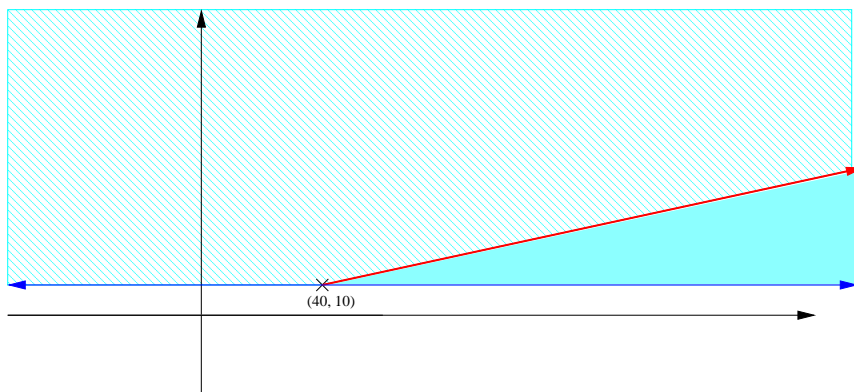


Abbildung 6.11: Graphische Darstellung des Ergebnisses der Analyse zu LoopTest

Ergebnis vorliegt.

### 6.6.2 Testfall: Methodenaufruf

Der nächste Test beinhaltet die selbe Funktionalität wie das erste Programm, wobei die Addition mit Hilfe eines Prozeduraufrufs verwirklicht werden soll und ein sequentieller Programmablauf vorliegt.

```

public class MethTest{
2  int meth(){
    int x, y;
    x=1;
    y=1;
    y=y+1;
7  x=add(x, y);
}

```

```
        return x;
    }

    int add(int i, int j){
12     int result;
        result= i+j;
        return result;
    }
}
```

Resultat	
Point	[3.0,3.0,2.0,1.0]

Tabelle 6.2: Ergebnispunkt

Als Ergebnis dieses Testprogramms erhalten wir am Methodenende von *meth* ein sehr präzises Ergebnis in Form eines Punktes. Dieses Resultat 6.2 kommt dadurch zustande, dass die Effektmenge der aufgerufenen Methode *add* nur aus einer Effektmatrix besteht, nämlich einer Punktmatrix.

### 6.6.3 Testfall: Kombination verschiedener Programmkonstrukte

In den nächsten beiden Testfällen sollen die eben getrennt betrachteten Konstrukte der Schleifen und Methodenaufrufe, deren Auswirkung auf das Ergebnis gerade beschrieben wurde, in einem Programmablauf miteinander verknüpft werden.

```
public class Test{
    int calc(){
        int i;
4        i=10;
        int b=0;
        int c=0;
        int x=1;
        while(i >0){
9            b=i;
            i =sub(i,x);
            int tmp = c;
            c =sum(b,i);
            c=sum(c,tmp);
14        }
        return c;
    }

    int sum(int c, int d){
19        return c+d;
    }
    int sub(int c, int d){
```

```
    return c-d;  
  }  
24 }  
}
```

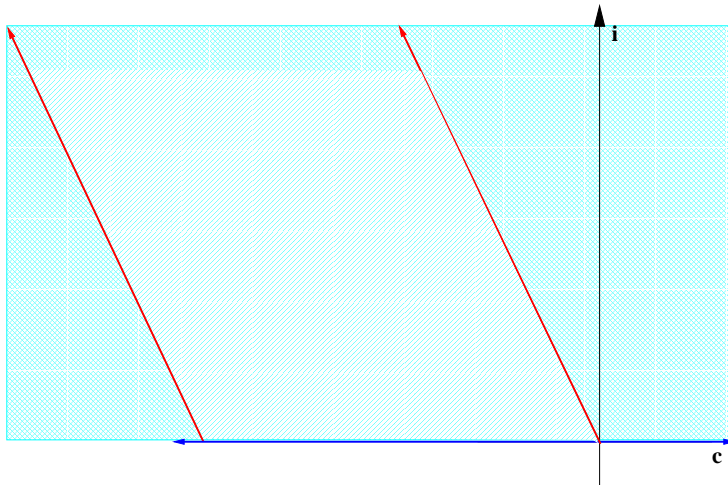


Abbildung 6.12: Graphische Darstellung des Ergebnisses der Analyse zu MethTest

Mit Hilfe dieser Graphik 6.6.3 soll ein Teilaspekt des Ergebnissgeneratorsystems dargestellt werden, indem der Zusammenhang zwischen den Programmvariablen  $i$  und  $c$  verdeutlicht werden soll.

```

1  public class Test{
    int calc(){
        int i;
        i=10;
        int b=0;
6   int c=0;
        int x=1;
        while(i >0){
            b=i;
            i =sub(i,x);
11         int tmp = c;
            c =sum(b,i);
            c=sum(c,tmp);
        }
        int y =0;
16        for(int z=0; z<3; z=z+1){
            y=y+1;
        }
        return c;
    }
21
    int sum(int c, int d){
        return c+d;
    }
    int sub(int c, int d){
26        return c-d;
    }
}

```

Resultat	
Line <sub>1</sub>	[172.0,0.0, 1.0, 172.0, 1.0, 171.0,0.0,0.0,0.0]
Line <sub>2</sub>	[1558.0,0.0, 10.0, 1558.0, 10.0, 1548.0,0.0,0.0,0.0]
Ray <sub>1</sub>	[-19.0, -9.0, -10.0, -19.0, -1.0,0.0,0.0,0.0,0.0]
Ray <sub>2</sub>	[0.0,0.0,0.0,0.0,0.0,0.0,1.0,1.0,0.0]
Point <sub>1</sub>	[-21.1̄,0.0, -11.1̄, -21.1̄, -0.1̄,0.0,3.0,3.0,1.0]
Point <sub>2</sub>	[0.0,0.0,0.0,0.0,0.0,0.0,3.0,3.0,1.0]

Tabelle 6.3: Ergebnisgeneratorensystem für Test2

Das zweite Testprogramm, welches eine weitere Schleife enthält, liefert das Ergebnisgeneratorensystem 6.3, das mit zwei Linien und zwei Strahlen einen sehr grossen Bereich aufspannt. Durch das Vorhandensein der Schleifen in unseren Testprogrammen müssen wir unter Anwendung des Widenings einen Präzisionsverlust verzeichnen, so dass unsere geometrische Interpretation des Methodenendes Linien enthält.

### 6.6.4 Anwendungsbeispiel

Ein mögliches Anwendungsgebiet für die in der vorliegenden Arbeit entstandene Implementierung einer interprozeduralen Programmanalyse wäre **Array Bound Checking**, wie das Beispiel 6.13 verdeutlichen soll, welches eine Matrix-Vektor-Multiplikation beschreibt. Dabei möchte man die gültigen Speicheradressen wissen, die bei der Durchführung der Multiplikation angesprochen werden. Mit dem nachfolgenden Beispielprogramm zur Berechnung der Multiplikation einer  $3 \times 3$ -Matrix mit einem Vektor der Länge 3 wird die Matrix in einem eindimensionalen Vektor der Länge 9 abgespeichert. Diese Transformation der zweidimensionalen Matrix in einen eindimensionalen Vektor wird in der Funktion *translate* modelliert.

```
1 public class ArrayBoundCheck{
    public static void main(){
        int i;
        int j;
        int offset1;
6       int offset2;
        int offset3;
        i=0;
        j=0;
        offset1=25;
11      offset2=34;
        offset3=37;
        while(i<3){
            while(j<3){
                int address1 = translate(i,j, offset1); //a[i,j]
16         int address2 = translate2(j,offset2); //b[j]
                int address3 = translate2(i,offset3); //c[i]
                // c[i]:= b[j] a[i,j]
                j=j+1;
                i=i+1;
21      }
        }
    }

    public static int translate(int x, int y, int z){
26     int result = x*3+y+z;
        return result;
    }

    public static int translate2(int a, int b){
31     int c = a+b;
        return c;
    }
}
```

Abbildung 6.13: ArrayBoundCheck

Als Ergebnis der Ausführung der Funktion *main* erhält man anhand des vorher bekannten Offsets für die entsprechenden Objekte die Adressen der Speicherzellen, auf die zugegriffen wird, was die Variablen *address1*, *address2* und *address3* beinhalten. Die konkreten Ergebnisse werden in der Tabelle 6.4 zusammengefasst.

Resultat	
Line	[37.0, 0.0, 0.0, 25.0, 34.0, 37.0, 25.0, 34.0, 37.0, 0.0]
Ray	[38.0, 1.0, 1.0, 25.0, 34.0, 37.0, 29.0, 35.0, 38.0, 0.0]
Point <sub>1</sub>	[113.0, 3.0, 3.0, 75.0, 102.0, 111.0, 83.0, 104.0, 113.0, 1.0]
Point <sub>2</sub>	[114.0, 3.0, 3.0, 100.0, 136.0, 148.0, 87.0, 105.0, 114.0, 1.0]

Tabelle 6.4: Ergebnissgeneratorsystem für ArrayBoundCheck

### 6.6.5 Fazit

Mit Hilfe der durchgeführten Testfälle konnte gezeigt werden, dass die Implementation der theoretischen Konzeption nach [MOS04] in der Lage ist ein Programm schnell zu analysieren. Die Ergebnisse sind doch noch etwas ungenau, da durch die Anwendung des Widening beim Durchlaufen einer Schleife ein Strahl gebildet wird. Da in den kleinen Testprogrammen wenige Variablen vorkommen und als Schwellwert zur Anwendung der Wideningfunktion diese Variablenanzahl gesetzt wird, bekommen wir sehr schnell viele Strahlen. Diese Strahlen sind linear abhängig und führen bei der Bildung der konvexen Hülle zu einer Erweiterung des Generatorsystems um eine Linie. Somit wird der konvexe Bereich ziemlich gross.

In der Tabelle 6.5 soll ein kurzer Eindruck vermittelt werden, wie viele Iterationen die beiden Analysen *ClassVisitor* und *MethodVisitor* für die jeweiligen Programme benötigen, bis sich ein Ergebnissgeneratorsystem eingependelt hat und die Analyse terminiert.

	ClassVisitor	MethodVisitor
LoopTest	76	83
MethTest	23	16
Test1	135	88
Test2	277	149
ArrayBoundCheck	112	165

Tabelle 6.5: Iterationsanzahl der einzelnen Testprogramme

Die Tabelle 6.6 zeigt die Grösse des zu analysierenden Kontrollflussgraphen bzw. der ganzen Menge von Kontrollflussgraphen für den *ClassVisitor*.

	CFGClass	CFGMethod
LoopTest	12	12
MethTest	14	9
Test1	22	16
Test2	29	23
ArrayBoundCheck	31	23

Tabelle 6.6: Anzahl der zu durchlaufenden Zustände innerhalb jedes Visitors

## 6.7 Verbesserungsmöglichkeit

Aus Mangel an Entwicklungszeit liessen sich einige Ideen, die die Leistungsfähigkeit der Analyse verbessern könnten, nicht mehr realisieren. Dabei wäre anzudenken, eine andere Form des Widening zu implementieren. In der momentanen Fassung der Analyse wird ein sehr einfaches Widening angewandt, welches aus zwei linear abhängigen Punkten einen Strahl bildet. Ein besserer Ansatz wäre es darauf zu achten, welche Richtung der zu bildende Strahl hat, so dass nicht sofort wieder im nächsten Schritt Widening anzuwenden ist, und man so viele Strahlen und somit eine wenig aussagefähige Analyse erhält.

Eine weitere Korrektur bestünde darin, da die Analyse auf *double*-Werten an Stelle von *integer*-Werten arbeitet, nur *integer*-Werte für die Beschreibung der Programzustände zu gestatten. Diese Ungenauigkeit der Wertebereiche liess sich dahingehend nicht vermeiden, dass Linear Programming, angewendet auf unsere Problemstellung, *double*-Werte zurückliefert. Im Zuge dieser Eingliederung von *glpk* wurde die Analyse so verändert, dass sie fortan auf *double*-Werten arbeitet.

Damit die derzeitige Version der implementierten Analyse adäquat in der Praxis eingesetzt werden kann, sind Erweiterungen vorzunehmen. Dabei ist eine Vergrösserung des zu analysierenden Sprachschatzes erforderlich. Um für unsere Analysezwecke ein ganzes Programm untersuchen zu können, musste der Sprachumfang von Java stark eingegrenzt werden. Feld- und auch Objektzugriffe wurden in die Analyse bisher noch nicht eingearbeitet, sondern nur als nichtdeterministische Zuweisungen modelliert. Die Betrachtung dieser Sprachkonstrukte würde die bestehende Analyse in ihren Aussagen präzisieren.

Trifft unsere Analyse auf arithmetische Ausdrücke, in denen Divisionen, Modulooperationen oder Potenzierungen einer Variablen vorkommen, so werden diese ebenfalls als nichtdeterministische Zuweisungen behandelt. Somit können weiterhin affine Relationen zwischen den Programmvariablen angegeben werden. Auch bei der Analyse von Bedingungen ist darauf zu achten, dass sich die Bedingung in linearer Form gestaltet. Zudem muss noch eine Möglichkeit gefunden werden, damit Negativbedingungen  $x \neq 0$  analysiert werden können.

Um die Aussagestärke unserer Analyse zu präzisieren, sollte ein *Narrowing* angewendet werden. Unter Einsatz dieses Operators kann die mittels *Widening* berechnete obere Annäherung an den Fixpunkt weiter verbessert werden. Dabei muss für den *Narrowing* Operator

$\Delta \in [V \times V \rightarrow V]$  folgendes erfüllt sein:

$$\forall x, y \in V : (y \sqsubseteq x) \Rightarrow (y \sqsubseteq (x\Delta) \sqsubseteq x)$$

Ein Ausbau der Analyse mit diesen Erweiterungen würde in der Praxis sicherlich Anwendung finden.

### 6.8 Zusammenfassung

In diesem Kapitel wurden die wesentlichen javatechnischen Realisierungsaspekte an Hand von Codefragmenten dargelegt und die Funktionsweise des Prototypen an Beispielprogrammen demonstriert. Dabei stellte sich heraus, dass vermutlich präzisere Ergebnisse erhalten werden können, wenn eine effektivere Wideningfunktion zum Einsatz kommt und nach Durchlauf der Analyse zur Verfeinerung des resultierenden Generatorensystems *Narrowing* statt findet, so dass die genauigkeitshindernden Linien und Strahlen teilweise entfernt oder sogar ganz weggelassen werden können.

## 7 Hilfsmittel

In diesem Kapitel sollen kurz die für die Implementierung verwendeten Hilfsmittel aufgeführt werden. Bei der intraprozeduralen Analyse wurden die Programmzustände mit Polyedern realisiert, wozu die Bibliothek *PPL* geeignete Objekte und Methoden bereit stellt. Implementierungstechnisch wurde für die Realisierung der interprozeduralen Analyse auf die Infrastruktur von *polyinvar* zurückgegriffen, ein Programmanalysewerkzeug zur Verifikation polynomieller Invarianten.

Da wir bei der Realisierung der interprozeduralen Analyse mittels linearer Ungleichungen auf einige mathematische Probleme gestossen sind, für die nicht eigens ein spezieller Algorithmus entwickelt werden sollte, haben wir versucht diese allgemein als ein *LP* Problem zu formulieren, welches sich mit Hilfe eines *LP Solvers* lösen lässt. Für die Lösung dieser LP Fragestellungen wird *glpk* verwendet, dessen Funktionsweise in 7.2.3 beschrieben wird.

### 7.1 Parma Polyhedra Library

Die Parma Polyhedra Library (PPL) ist eine C++ Bibliothek, die eine Menge von Funktionen zur Verfügung stellt, die auf konvexen Polyedern arbeiten. Intern werden beide Darstellungsarten eines Polyeders verwaltet, das Relationensystem und auch das Generatorensystem. Bei PPL bleiben durch die Entkopplung der Benutzerschnittstellen die implementierungstechnischen Details dem Benutzer fast ganz verborgen. Zum Beispiel kann die interne Repräsentation des Relationen- und Generatorensystems vom Anwender nicht angesprochen werden, sondern es können nur die zur Verfügung gestellten Methoden verwendet werden. Zur Implementierung der von Cousot spezifizierten Analyse wurde PPL der Version 0.7 von der Homepage, die im Dezember 2004 herausgegeben wurde, verwendet.

### 7.2 Linear Programming

#### 7.2.1 Mathematische Definition

Viele mathematische Probleme lassen sich mit Hilfe eines allgemeinen Problems, des sogenannten LP Problems, ausdrücken. Das zu lösende Problem muss nur in ein lineares Gleichungssystem transformiert werden und kann anschliessend mit Hilfe eines LP Solvers gelöst werden. Es ist kein zusätzlicher Aufwand für die Entwicklung eines spezialisierten Algorithmus nötig, zumal eine ganze Reihe von LP Solvern zur Verfügung steht, mit deren Hilfe sehr

effizient eine Lösung für das Problem angegeben werden kann. Für die Implementierung der interprozeduralen Analyse wurde *glpk* verwendet.

Lineares Programmieren gliedert sich in das **Erfüllbarkeitsproblem** und das **Optimierungsproblem**, was je nach Problem zu verschiedenen Arten einer Antwort führt.

Beim Erfüllbarkeitsproblem soll die Frage nach einem Vektor  $\mathbf{x}$ , der das lineare Ungleichungssystem  $A\mathbf{x} \leq \mathbf{b}$  löst, beantwortet werden, während man beim Optimierungsproblem stets die beste Lösung für das angegebene Problem erhalten möchte.

Bei einem LP Problem muss eine lineare Zielfunktion über einer Menge von linearen Gleichungen und Ungleichungen optimiert werden. Dabei kann die Zielfunktion entweder maximiert oder minimiert werden. Das System von linearen Gleichungen und Ungleichungen ist meist unterbestimmt, so dass bei der Wahl des Lösungsvektors  $\mathbf{x}$ , um die Zielfunktion zu maximieren, grosse Freiheit herrscht.

**Zielfunktion:**  $(c_1x_1 + \dots + c_nx_n) = k$

$$\text{Nebenbedingungen: } \begin{pmatrix} a_{11}x_1 & \dots & a_{1n}x_n & \leq & b_1 \\ \vdots & \ddots & \vdots & & \vdots \\ a_{m1}x_1 & \dots & a_{mn}x_n & \leq & b_m \end{pmatrix}$$

Jeder Vektor  $\mathbf{x}$ , der die Nebenbedingungen erfüllt, heisst *zulässige Lösung* für dieses lineare Optimierungsproblem. Alle Komponenten  $A, \mathbf{x}, \mathbf{b}, \mathbf{c}$  des LP Problems müssen kompatible Dimensionen haben.

**Standardform:**  $\max\{\mathbf{c}^T\mathbf{x} = \sum_{j=1}^n c_jx_j \mid A\mathbf{x} \leq \mathbf{b}; x_1, \dots, x_n \geq 0\}$

wobei  $A$  eine reelle, meist nicht quadratische,  $m \times n$  Matrix,

$\mathbf{b} = (b_1, \dots, b_m)^T$ ,  $\mathbf{c} = (c_1, \dots, c_m)^T$  reelle Vektoren mit im voraus bekannten Koeffizienten sind.

Um ganz allgemein diverse mathematische Probleme, die sich als lineares Gleichungssystem modellieren lassen, behandeln zu können, kann die Standardform verändert werden. Die meisten LP Solver erlauben im Gegensatz zur Standardform, wo Variablen nur positive Werte annehmen dürfen, das Setzen von allgemeinen Grenzen  $l_i \leq x_i \leq u_i$ . Dabei können die Werte der Vektorkomponenten  $x_i$  dynamisch gesetzt werden. Es kann eine kleinste untere Schranke  $l_i$ , die auch  $-\infty$  sein darf, und auch eine grösste obere Schranke  $u_i$  mit maximalem Wert  $+\infty$  angegeben werden.

Somit ist die Möglichkeit gegeben, dass der Variablenvektor  $\mathbf{x}$  ohne explizite Grenzen vorkommen kann.

Eine weitere mögliche Variation der Standardform besteht in der Formulierung von Gleichheitsbedingungen innerhalb der Nebenbedingungen  $a_i \mathbf{x} = b_i$ . Wie bereits angedeutet, kann das LP Problem auch minimiert werden. Für die Lösung des mathematischen Problems, nachdem es auf ein LP Problem reduziert wurde, ist nun darauf zu achten, welche Form der Lösung man erhalten möchte. Reicht es die Existenz einer Lösung zu wissen oder ist man an den Komponenten des Ergebnisvektors interessiert. Je nach Problemfall muss die entsprechende Frage an den LP Solver gestellt werden.

### Optimale Lösung eines LP Problems

Um die optimale Lösung eines LP Problems zu beschreiben, gehen wir des weiteren von der Standardform aus. Bei dieser Problemstellung möchte man die beste Lösung für die angegebene Zielfunktion. Der zulässige Bereich eines LP Problems ist die Menge aller  $\mathbf{x}$ , die die Nebenbedingungen erfüllen.

$$F = \{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$$

Jeder Punkt von  $F$  stellt eine mögliche Lösung dar. Ein Punkt  $\mathbf{x}^* \in F$  ist eine optimale Lösung, wenn  $\mathbf{c}^T \mathbf{x}^*$  der maximale Wert von  $\mathbf{c}^T \mathbf{x}$  bezüglich der Zielfunktion ist.  $F$  ist eine geschlossene, konvexe Untermenge des  $\mathbb{R}^n$ , da ja der Durchschnitt konvexer Mengen wieder konvex und der Durchschnitt geschlossener Mengen ebenfalls geschlossen ist.

### Erfüllbarkeit eines LP Problems

Der Erfüllbarkeitstest liefert lediglich die Antwort, ob es überhaupt einen Ergebnisvektor für die spezifizierte Problemstellung gibt oder nicht. Der Vorteil dieser Überprüfung liegt darin, dass das Problem schnell zu lösen ist, da nicht explizit nach dem besten Ergebnis gesucht werden muss, sondern man sich mit einer *ja-nein* Antwort begnügt.

Ein lineares Programm muss nicht lösbar sein. Dies wäre der Fall, wenn der zulässige Bereich leer ist

$$\{\mathbf{x} : A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$$

oder aber die Zielfunktion  $\mathbf{x} \mapsto \mathbf{c}^T \mathbf{x}$  auf dem zulässigen Bereich nicht nach oben beschränkt ist.

## 7.2.2 Geometrische Interpretation

Jedes lineare Programm lässt sich geometrisch interpretieren. So beschreibt jede lineare Gleichung über den Variablen  $x_1, \dots, x_n$  eine Hyperebene im  $n$ -dimensionalen Raum. Die dazugehörige lineare Ungleichung klassifiziert alle Punkte, die auf der einen Seite der Hyperebene liegen inklusive der Ebene selbst. Wie man leicht erkennen kann, wird durch eine Ungleichung ein Halbraum charakterisiert, der alle gültigen Lösungen der Ungleichung beschreibt. Hat man nun aber ein endliches lineares Ungleichungssystem, so besteht deren Lösungsmenge aus den

Punkten, die alle Ungleichungen erfüllen. Dies ist geometrisch gesehen gerade der Schnitt der Halbräume, was zu einem konvexen Polyeder führt. Dieses  $n$ -dimensionale Polyeder muss jedoch nicht beschränkt sein. Auch die zu maximierende Zielfunktion selbst definiert eine Hyperebene, wobei der Abstand zum Ursprung (Zielwert) noch nicht bekannt ist. Verändert man die Steigung dieser Ebene durch den Ursprung bis alle Ungleichungen erfüllt sind, so hat man einen Zielwert gefunden.

### 7.2.3 GNU Linear Programming Kit

Um grosse skalierte LP Probleme lösen zu können, wurde das in ANSI C verfasste GNU Linear Programming Kit **glpk** von der Homepage (Version 4.8 von Januar 2005) verwendet, deren C-Routinen wie aus einer Bibliothek aufgerufen werden können. Bei **glpk** wird das LP Problem wie in [Mak05] folgendermassen definiert:

**Zielfunktion** :  $Z = c_1x_{m+1} + \dots + c_nx_{m+n} + c_0$

**Lineare Bedingungen** : 
$$\begin{pmatrix} a_{11}x_{m+1} & \cdots & a_{1n}x_{m+n} & \leq & x_1 \\ \vdots & \ddots & \vdots & & \vdots \\ a_{m1}x_{m+1} & \cdots & a_{mn}x_{m+n} & \leq & x_m \end{pmatrix}$$

**Grenzen der Variablen** : 
$$\begin{pmatrix} l_1 & \leq & x_1 & \leq & u_1 \\ \vdots & & \vdots & & \vdots \\ l_{m+n} & \leq & x_{m+n} & \leq & u_{m+n} \end{pmatrix}$$

- $x_1, \dots, x_m$  Hilfsvariablen, die den Zeilen in der Matrix für die Nebenbedingungen entsprechen,
- $x_{m+1}, \dots, x_{m+n}$  Strukturvariablen, welche die Spalten der Matrix darstellen,
- $l_1, \dots, l_{m+n}$  die untere Schranke,
- $u_1, \dots, u_{m+n}$  die obere Schranke der Variablen

Die Schranken für die Variablen können auf  $+\infty, -\infty$  und feste Werte gesetzt werden. Eine Lösung des LP Problems ist gegeben, wenn Werte für die Strukturvariablen und die Hilfsvariablen angegeben werden können, so dass alle linearen Bedingungen erfüllt werden, die Variablen in ihren vorgegebenen Schranken sind und eine maximale Lösung für die Zielfunktion (im Falle einer Maximierung des LP Problems) angegeben werden kann.

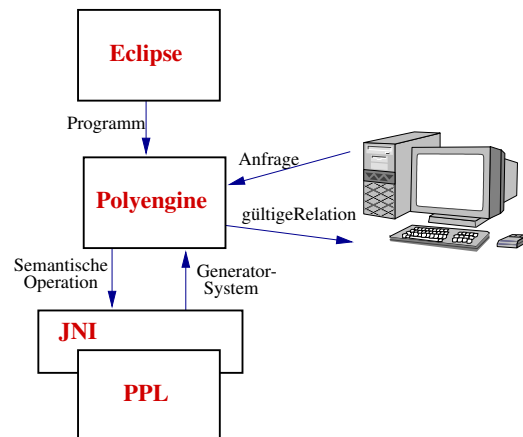


Abbildung 7.1: Aufbau der vorgegebenen Infrastruktur für die intraprozedurale Analyse

Bei *glpk* wird intern zur Lösung eines LP Problems die **Simplex Methode** gemäss [Mak05] verwendet. Diese numerische Vorgehensweise versucht in mehreren Iterationen, bei denen das ursprüngliche lineare System von Nebenbedingungen in eine sogenannte Simplextabelle transformiert wird, die optimale Lösung zu finden.

## 7.3 polyinvar

Zugrundeliegend für die Implementation der intraprozeduralen 4 und interprozeduralen 5 Analyse ist das Programmanalysewerkzeug *polyinvar* von [Pet05], dessen Aufbau kurz elaboriert werden soll.

### 7.3.1 Architektonischer Aufbau

*Polyinvar* ist ein in Java verfasstes Programmpaket, dessen Arbeitsweise in Abbildung 7.3.1 schematisch visualisiert wird.

Unter Zuarbeit der *Eclipse API JDT* werden zu der zu analysierenden Java-Klasse die jeweiligen Kontrollflussgraphen generiert. Das Programmanalysewerkzeug *polyinvar* beinhaltet das Paket *cfggenerator.jar*, welches den Kontrollflussgraphen unter zu Hilfenahme des *Eclipse-Plugins* aus dem zu analysierenden Programm erstellt. In *cfgstructure.jar* wird die Datenstruktur deklariert. Im Programmanalysator *polyhedralAnalysator.jar* kann die Fixpunktiteration für die intraprozedurale Analyse auf dem so generierten Kontrollflussgraphen direkt arbeiten. Da die in dieser Arbeit behandelten Analysen auf der mathematischen Darstellungsweise der Polyeder bzw. konvexen Mengen beruhen, wurde die Bibliothek *PPL* herangezogen. Die eigentlichen Berechnungen werden von dem Wrapper *polyhedralAnalysis* durchgeführt, der in der Programmiersprache C++ verfasst wurde und mit Hilfe von *PPL 7.1* diverse Manipulatio-

nen an den Polyedern vornimmt. *PolyhedralAnalysis* wird unter Benutzung des *Java Native Interfaces JNI* angesprochen.

### 7.3.2 Kontrollflussgraph

Bei der Generierung des Kontrollflussgraphen, auf dem die Programmanalyse aufbaut, wird der von *JDT* zur Verfügung gestellte abstrakte Syntaxbaum durchlaufen und nur bestimmte Konstrukte der Programmiersprache *Java* werden unterstützt. Programmiertechnische Gebilde, die wir in unserer Analyse nicht betrachten wollen, werden mit Hilfe von nicht deterministischen Anweisungen modelliert, wie zum Beispiel Objekt- oder Arrayzugriffe und im Falle der intraprozeduralen Analyse auch die Prozeduraufrufe. Ein Kontrollflussgraph stellt die abstrakte Interpretation einer einzelnen Prozedur dar und wird bei der intraprozeduralen Analyse über die *CFGMethod* und bei einer interprozeduralen Analyse über *CFGClass*, einer Sammlung von *CFGMethoden* deklariert. Dargestellt wird ein Kontrollflussgraph durch eine Klasse für die Knoten *CFGState* und einer für die Kanten *CFGEdge*. Hierbei sind je nach Programmanweisung drei Kantenarten zu unterscheiden:

- *CFGAssignmentEdge* → deterministische und nicht deterministische Zuweisungen
- *CFGAssertionEdge* → Bedingungen
- *CFGNullEdge* → skip-Anweisungen

## 8 Zusammenfassung

Zu Beginn dieser Arbeit wurde eine intraprozedurale Analyse beschrieben, die jeden Programmpunkt mit diesem Programmzustand charakterisierenden Polyedern beschreibt. Dabei wurde auf die theoretische Spezifikation der intraprozeduralen Programmanalyse nach Cousot mit Hilfe der zwei Darstellungsformen von Polyedern zurückgegriffen. Polyeder können als Relationensystem und als Generatorensystem dargestellt werden. Cousot fordert für seine Analyse eine ständige Verwaltung beider Darstellungsformen. Der Nachteil bei dieser Analyse ist die teure Operation der Umwandlung der einen Polyederdarstellung in die andere.

So wurde eine zweite Analyse angegangen, die nur auf dem Generatorensystem arbeitet und nicht nur intraprozedural, sondern interprozedural angewendet werden kann. Dabei ist nur am Ende der Analyse eine Umwandlung in ein Ungleichungssystem nötig, das trivialerweise schon dargestellt wird. Diese Analyse ist zweistufig, so dass in der ersten Analyse alle Kontrollflussgraphen des zu analysierenden Programmes durchlaufen werden. Hierbei werden Effektmatrizen erstellt, die den Übergang von einem Startpunkt  $s$  in einen Zielpunkt  $t$  charakterisieren. Darauf aufbauend wird die zweite Analyse durchgeführt, wobei nur ein Kontrollflussgraph, nämlich der für die zu analysierende Prozedur durchlaufen wird. Es wird an jedem Knoten eine konvexe Menge, bestehend aus Punkten, Strahlen und Linien, abgespeichert und mit Hilfe der Effektmatrizen aus der ersten Analyse kann entsprechend der Kantenart eine Überführung eines Programmzustandes in den nächsten vorgenommen werden.

Zur Verifizierung dieser theoretischen Erkenntnisse wurde eine Implementation versucht, deren Effizienz an kleinen Testprogrammen untersucht wurde. Hierbei zeigt sich, dass das Vorkommen von Schleifen das Analyseergebnis durch die Bildung von Strahlen und Linien sehr ungenau macht und den Wertebereich der Programmvariablen vergrößert. Diese ungenauen Ergebnisse können durch den Einsatz von Narrowing oder die Verwendung einer anderen Wideningmethodik wieder präzisiert werden. Die Implementation hat gezeigt, dass eine Analyse beruhend auf konvexen Mengen sehr schnell ein korrektes Ergebnis zur Beschreibung der affinen Relationen zwischen den Programmvariablen liefert. Ein möglicher Einsatzbereich wäre bei Compileroptimierungen denkbar, wie das Beispiel 6.6.4 andeuten soll.



# Abbildungsverzeichnis

4.1	Naiver Fixpunktalgorithmus . . . . .	22
5.1	inkrementeller Fixpunktalgorithmus . . . . .	33
5.2	Generierung eines Strahls aus linear abhängigen Punkten . . . . .	34
5.3	Prozedur <i>widen()</i> zur Durchführung des Widenings . . . . .	35
6.1	Darstellung der Matrizenobjekte . . . . .	38
6.2	Darstellung der konvexen Mengen als Vektoren . . . . .	39
6.3	Prozedur <i>convexHull()</i> zur Durchführung der konvexen Hülle . . . . .	40
6.4	Prozedur <i>addRay</i> . . . . .	41
6.5	Inklusionstest von <i>addLine()</i> . . . . .	42
6.6	Inklusionstest von <i>addPoint()</i> . . . . .	42
6.7	Überprüfung ob Strahlen eine Bedingung erfüllen . . . . .	43
6.8	Berechnung der Schnittflächen mit der Hyperebene . . . . .	44
6.9	Prozedur <i>append()</i> mit Widening und Bilden der konvexen Hülle . . . . .	45
6.10	LoopTest . . . . .	51
6.11	Graphische Darstellung des Ergebnisses der Analyse zu LoopTest . . . . .	51
6.12	Graphische Darstellung des Ergebnisses der Analyse zu MethTest . . . . .	53
6.13	ArrayBoundCheck . . . . .	55
7.1	Aufbau der vorgegebenen Infrastruktur für die intraprozedurale Analyse . . . . .	63



# Tabellenverzeichnis

6.1	Kombination von $gs_{guard}$ und $gs_{\overline{guard}}$ . . . . .	43
6.2	Ergebnispunkt . . . . .	52
6.3	Ergebnisgeneratorensystem für Test2 . . . . .	54
6.4	Ergebnisgeneratorensystem für ArrayBoundCheck . . . . .	56
6.5	Iterationsanzahl der einzelnen Testprogramme . . . . .	56
6.6	Anzahl der zu durchlaufenden Zustände innerhalb jedes Visitors . . . . .	57



# Literaturverzeichnis

- [CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program . *Conference Record of the 5th Annunal ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.
- [GN03] Sumit Gulwani and George C. Necula. Discovering Affine Equalities Using Random Interpretation. *POPL*, pages 74–84, 2003.
- [Mak05] Andrew Makhorni. GNU Linear Programming Kit - Reference Manual. pages 7–16, 2005.
- [MOS04] Markus Mueller-Olm and Helmut Seidl. Precise Interprocedural Analysis through Linear Algebra. *POPL*, 2004.
- [Pet05] Michael Petter. Berechnung von polynomiellen Invarianten. 2005.
- [RBZ03] Elia Ricci Roberto Bagnara, Patricia M. Hill and Enea Zaffanella. Precise Widening Operators for Convex Polyhedra. *10th International Symposium on Static Analysis*, 2003.
- [RV02] Tatiana Rybina and Andrei Voronkov. Using canonical representations of solutions to speed up infinite-state model checking. *14th International Conference on Computer Aided Verification*, 2002.
- [RV03] Tatiana Rybina and Andrei Voronkov. A logical reconstruction of reachability. *5th International Andrei Ershov Memorial Conference*, 2003.