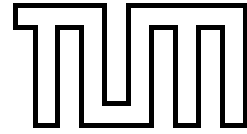


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK



Diplomarbeit

Berechnung von polynomiellen Invarianten

Michael Petter

11. Oktober 2004

Aufgabensteller: Prof. Dr. Helmut Seidl

Betreuer: Prof. Dr. Helmut Seidl

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

Garching, den _____

Unterschrift: _____

Inhaltsverzeichnis

1	Einleitung	1
1.1	Motivation	1
1.2	Verwandte Ansätze	1
2	Theoretische Hintergründe	3
2.1	Operationelle Semantik	3
2.1.1	Unterstützte Sprachelemente	3
2.1.2	Repräsentation als Kontrollflußgraph	4
2.1.3	Semantik des Kontrollflußgraphen	5
2.1.4	Zusammenfassung	7
2.2	Verifikation von polynomiellen Invarianten	7
2.2.1	Formale Problemspezifikation	8
2.2.2	Abstrakte Interpretation der operationellen Semantik	9
2.2.3	Der Fixpunktalgorithmus	11
2.2.4	Zusammenfassung	13
2.3	Herleitung von polynomiellen Invarianten festen Grades	14
2.3.1	Erzeugung eines generischen Polynoms	14
2.3.2	Rekonstruktion der Invarianten	15
2.3.3	Anpassung des Algorithmus	15
2.3.4	Vektoren als Alternative	16
2.3.5	Rekonstruktion der Invarianten	18
3	Experimentelle Implementation	21
3.1	Architektur	21
3.1.1	Aufgabenverteilung	21
3.1.2	Schnittstellen	22
3.2	Testreihe	25
3.2.1	Realisierung mit Modulen statt Idealen	26
3.2.2	Die Geometrische Reihe	26
3.2.3	Die Potenzsumme	27
3.2.4	Der Einfluß der Variablenanzahl	28
3.2.5	Optimierte Generatorauswahl	31
3.3	Vergleich	32
3.3.1	lcm	32

3.3.2	prod	33
3.3.3	cubes	33
3.3.4	factor	34
3.3.5	Zusammenfassung	35
4	Dokumentation	37
4.1	Anwenderdokumentation	37
4.1.1	Voraussetzungen	37
4.1.2	Installation	37
4.1.3	Benutzung	38
4.2	Entwicklung	40
4.2.1	Eingabemodul	40
4.2.2	Rechenengine	40
5	Ergebnisse	43
5.1	Fazit	43
5.2	Erweiterungsmöglichkeiten	43
5.2.1	Sprachumfang	43
5.2.2	Zusicherungen und Abfragen	47
5.2.3	Zeitliche Begrenzung und Widening	48
	Abbildungsverzeichnis	51
	Tabellenverzeichnis	53

1 Einleitung

1.1 Motivation

Die Antwort auf die Frage, ob eine polynomielle Invariante an einem Programmpunkt gültig ist, ist vielseitig verwendbar. Viele Probleme aus der Programmanalyse lassen sich als polynomielle Invarianten formulieren [?]. So läßt sich zum Beispiel die Analyse auf gleiche Variablen ausdrücken als Invariante $x = y$. Die Frage der Konstantenpropagation läßt sich auch sehr einfach als Polynom ausdrücken $x = c$. Auch weniger triviale Analysen wie die Entdeckung von symbolischen Konstanten $x = 7z^2 + 5$ sind sehr einfach als Polynomielle Invarianten formulierbar. Sogar die Entdeckung von syntaktisch verschiedenen äquivalenten Ausdrücken, oder von Schleifen-Induktionsvariablen ist damit gegeben.

Auch für den Bereich der Programmverifikation ist eine Ableitung von Invarianten sehr interessant, so können damit zum Beispiel Schleifeninvarianten entdeckt werden. Die entdeckten Relationen können einen Korrektheitsbeweis wesentlich unterstützen und vereinfachen.

In dieser Arbeit soll ein Verfahren untersucht werden, um eben solche polynomiellen Invarianten an Kontrollflußgraphen verifizieren und sogar herleiten zu können. Im ersten Kapitel werden die theoretischen Hintergründe zusammengefasst dargestellt, die in [?] zu diesem Thema präsentiert werden. Da sich dieses Verfahren jedoch auf Ideale von Polynomen, Gröbnerbasen und Reduktionen abstützt, was potentiell sehr zeitaufwendige Operationen nach sich zieht, ist nicht klar, wie performant eine Realisierung dieses Verfahrens ist. Im Zuge des zweiten Kapitels wird nun beschrieben welche Erfahrungen mit einer eigenen Implementierung des vorher beschriebenen Algorithmus gemacht werden konnten. Im dritten Kapitel wird kurz auf die Installation und Benutzung dieses Werkzeugs eingegangen. Schliesslich präsentiert das letzte Kapitel Möglichkeiten zum weiteren Ausbau des Prototypen und gibt einen Ausblick auf weiterführende Konzepte.

1.2 Verwandte Ansätze

Bisher ist mir nur eine weitere Implementation eines Programmes zur Überprüfung von polynomiellen Invarianten bekannt, die auf die Arbeit von Enric Rodríguez Carbonell [?] zurück geht. Carbonells Ansatz basiert auch auf der Verwendung von Idealen, jedoch schränken die Vorbedingungen seiner Analyse die Verwendbarkeit seines Algorithmus stark ein: Scheinbar sind nur einfache Loop-Programme auf Schleifeninvarianten analysierbar, es bestehen also nur sehr beschränkte Einsatzmöglichkeiten. Auch scheint seine Analyse nur einfache Linearkombinationen von Variablen bei Zuweisungen zu beachten, nicht wie in diesem Ansatz beliebige

1 Einleitung

Polynome. Behandlungen von Verzweigungen und deren Wächter scheinen nicht explizit vorzukommen, nur implizit durch die Annahme einer globalen nichtdeterministischen Mehrfachverzweigung. Ein Vergleich der Leistung der Implementationen der beiden Verfahren findet sich in Abschnitt 3.3.

2 Theoretische Hintergründe

Um zu Erläutern, auf welche Art und Weise *Polyinvar* arbeitet, zeigt der erste Teil dieses Abschnitts unter Abstützung auf [?], auf welcher mathematischen Basis die Verifikation beziehungsweise Herleitung von Polynomiellen Invarianten ablaufen kann. Abschnitt 2.2 beschreibt, wie eine Verifikation von Invarianten in diesem Umfeld stattfinden kann. In Abschnitt 2.3 schliesslich wird erläutert, wie das Verfahren zur Verifikation erweitert werden kann, um die Herleitung von polynomiellen Invarianten zumindest bis zu einem festen Grad zu bewerkstelligen.

2.1 Operationelle Semantik

Der erste Schritt bei der Analyse beliebiger Programme in Bezug auf polynomielle Invarianten oder auch anderer Eigenschaften ist immer die Suche nach einer geeigneten Datenstruktur, die alle relevanten Schritte des zu analysierenden Programmes abbildet und uninteressante Teile ausblendet. Wie so oft in der Programmanalyse greift die Untersuchung der Invarianten auf das Prinzip der Kontrollflußgraphen zurück. Im Zuge dieses Abschnitts wird eine geeignete operationelle Semantik für diesen Graphen entworfen.

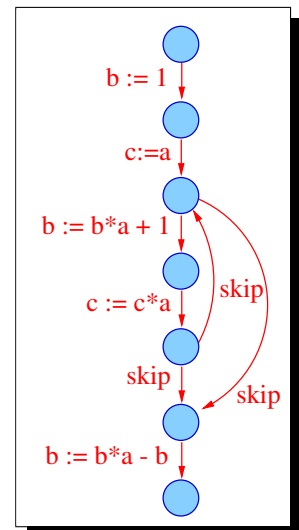
2.1.1 Unterstützte Sprachelemente

Für die Formulierung von Programmen, an denen die Analyse auf polynomielle Invarianten getestet werden kann, dient für den ersten Prototypen *Java*. Das Problem bei der Verwendung von *Java* zu solchen Zwecken ist der damit einhergehende enorme Sprachumfang, der sich einer Analyse entgegenstellt. Für den im Rahmen dieser Arbeit implementierten Prototypen ist der Sprachumfang viel zu groß, wesentlich für die hier betrachtete Analyse sind nur bestimmte Teile der gesamten Sprache.

Notwendig sind arithmetische Ausdrücke mit und Zuweisungen an ganzzahlige Variablen, sowie die Existenz von Gleichheits- und Ungleichheitsausdrücken. Merkmale wie Methodenaufrufe, Arrays, Objekteigenschaften und Nebenläufigkeit werden nicht von dieser Analyse betrachtet. Um selbst bei solchen Konstrukten nicht auf eine Analyse verzichten zu müssen bedient man sich der Modelle der nichtdeterministischen Verzweigung und Zuweisungen. Auch können arithmetische Ausdrücke nicht komplett behandelt werden, da die Division Aussagen über polynomielle Invarianten verhindert [?].

2.1.2 Repräsentation als Kontrollflußgraph

Die Analyse eines Programmes in Bezug auf polynomielle Invarianten basiert auf Kontrollflußgraphen. Kontrollflußgraphen stellen eine Abstraktion von beliebigen Programmen und Algorithmen dar. Man geht dabei von gerichteten Graphen aus, deren Knoten Zustände des Programmes zur Laufzeit darstellen, und deren Kanten Anweisungen des Programmes symbolisieren. Da nicht für jede Programmanalyse auch jede Anweisung eine relevante Zustandsänderung ergibt, bietet sich die Möglichkeit, die uninteressanten Zuweisungen auszublenden, indem man sogenannte *skip*; Anweisungen an deren Stelle einbaut. Auch Kontrollstrukturen wie Schleifen werden mit Hilfe von *skip*; Anweisungen im Kontrollflußgraphen realisiert.



Bedingungen, so wie sie in Schleifenköpfen und Verzweigungen vorkommen, können von Kontrollflußgraphen, sofern keine besondere Behandlung von Wahrheitswerten vorgenommen werden muss, durch nichtdeterministische Verzweigungen gelöst werden.

Bei den hier betrachteten Kontrollflußgraphen interessieren uns nur wenige verschiedene Kantenarten, die für die folgende Analyse eine Rolle spielen. Dabei handelt es sich um Zuweisungen an einzelne Variablen, sowie um bestimmte Wächter.

Für unsere Analysen definieren wir den Vektor der in einer Methode vorkommenden Variablen als $\mathbf{x} = (x_1, \dots, x_k)$. Der **Zustand** eines Programmes ist gegeben durch den Wert $x = (x_1, \dots, x_k)$, den diese Variablen an einem Programmpunkt beinhalten.

Die Kanten unseres Graphen stellen die Übergänge zwischen den betrachteten Zuständen dar. Daher sind die Kanten $e = (u, s, v)$ mit den Anweisungen s beschriftet, die einen Zustand u in den Nachfolgezustand v überführen. Da wir uns nur für Wächter und Zuweisungen interessieren, gibt es nur wenige Arten von Beschriftungen s :

Zuweisungen werden von unserer Analyse nur dann vollständig erfasst, wenn nur polynomielle Terme p zugewiesen werden. Sobald ein unbekannter Subterm wie zum Beispiel eine Division auftritt, wird der Wert der Zuweisung unbekannt, und in unserem Fall mit ? symbolisiert

Wächter können von unserer Analyse nur dann erfasst werden, wenn es sich um negative polynomielle Wächter ($p \neq 0$) handelt. Kombinierte Wahrheitsausdrücke können allerdings durch geeignete Strukturbildung teilweise simuliert werden.

Skipanweisungen *skip*; entstehen durch sonstige uninteressante Anweisungen im Originalprogramm, oder durch die Repräsentation von Strukturen wie Schleifen und Verzweigungen mit Bedingungen, die nicht durch negative polynomielle Wächter ausgedrückt werden können.

Eine ganze Methode kann daher formal dargestellt werden durch einen Kontrollflußgraphen $G = (N, E, A, s)$. Dabei handelt es sich um

\mathbf{N} , die Menge der Programmpunkte

$\mathbf{E} \subseteq N \times N$, die Menge der Kanten

$\mathbf{A} : E \rightarrow \text{Lab}$, eine Abbildung von Kanten auf zugehörige Beschriftungen, also Zuweisungen, *skip*;-Anweisungen und negierte polynomielle Wächter

\mathbf{s} , der Eintrittspunkt in die Methode

2.1.3 Semantik des Kontrollflußgraphen

Um polynomielle Invarianten zu untersuchen, geht diese Arbeit von der abstrakten Interpretation aller möglichen Programmabläufe bis zu einem bestimmten Zustand aus. Ein einzelner Programmablauf wird dabei dargestellt als Folge von Anweisungen:

$$r \equiv r_1; \dots; r_m$$

Der Programmablauf r kann also zusammengesetzt werden, aus den Beschriftungen der Kanten des Pfades vom Startknoten bis zum betrachteten Zielknoten. Die Menge der Programmabläufe, die den betrachteten Programmpunkt erreichen, kann verstanden werden als die kleinste Lösung für ein für dieses Programm spezifisches System von Untermengenrelationen auf Programmablaufmengen. Bei der Darstellung dieses Systems betrachtet man zuerst die Auswirkungen der Hinzunahme einzelner Kanten e auf die Programmablaufmenge. Der Effekt \mathbf{R} einer einzelnen Kante auf den Programmzustand hängt davon ab, welche Beschriftung $A(e)$ sie trägt:

- Ein Wächter $e = (u, p \neq 0, v)$ wird direkt als Effekt umgesetzt:

$$\mathbf{R}(e) = \{p \neq 0\}$$

- Eine Polynomzuweisung $e = (u, x_j := p, v)$ wird ebenfalls direkt umgesetzt:

$$\mathbf{R}(e) = \{x_j := p\}$$

- Unbekannte Zuweisungen $e = (u, x_j := ?, v)$ werden aufgefangen, indem man an dieser Stelle einfach davon ausgeht, dass an diese Variable eine unbekannte neue Konstante zugewiesen wird:

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

- Skip-Anweisungen $e = (u, \text{skip}, v)$ haben gar keinen Effekt auf den Programmzustand:

$$\mathbf{R}(e) = \{\}$$

Insgesamt lassen sich also die Pfade im Kontrollflußgraph, die gültigen Programmabläufen bis zum Zielknoten t entsprechen, durch die kleinste Lösung folgenden Systems von Untermengenrelationen \mathbf{R} angeben:

$$\begin{aligned} \text{[R1]} \quad \mathbf{R}(t) &\supseteq \{\epsilon\} \\ \text{[R2]} \quad \mathbf{R}(u) &\supseteq f_e(\mathbf{R}(v)), \text{ wenn } e = (u, s, v) \in E \end{aligned}$$

wobei

$$f_e(\mathbf{R}) = \{r; t \mid r \in \mathbf{R}(e) \wedge t \in \mathbf{R}\}$$

Untermengenrelation [R1] setzt fest, daß der Zielknoten immer an der Spitze jedes gültigen Programmpfades stehen muss. Untermengenrelation [R2] beschreibt, wie der bestehende Programmpfad von v um das Stück von u nach v erweitert werden muss, wenn die Kante $e = (u, s, v)$ als nächste betrachtet wird. Der gesamte Programmpfad wird somit also sozusagen rückwärts aufgebaut.

Programmabläufe haben wir bis jetzt als Kette von Effekten

$$r \equiv r_1; \dots; r_m$$

kennengelernt. Diese Effekte resultieren nun in einer Zustandsänderung des Variablenvektors x , der hinter jedem Programmzustand liegt. Diese Zustandsänderung lässt sich mit einer partiellen Transformation dieses Vektors ausdrücken: Effekte $\mathbf{R}(e)$ wirken sich daher auf die Menge der Zustände aus, indem Sie diese entweder einschränken, oder verändern. Die konkreten Effekte hängen wieder einmal von der Art der betrachteten Kante ab:

- Polynomielle Zuweisungen bewirken folgende Transformation:

$$\begin{aligned} \llbracket x_j := p \rrbracket x &= (x_1, \dots, x_{j-1}, p(x), x_{j+1}, \dots, x_k) \\ \text{dom}(\llbracket x_j := p \rrbracket) &= \{x \in \mathbb{F}^k\} \end{aligned}$$

- nichtdeterministische Zuweisungen funktionieren analog dazu:

$$\begin{aligned} \forall_{c \in \mathbb{F}} \llbracket x_j := ? \rrbracket x &= (x_1, \dots, x_{j-1}, c, x_{j+1}, \dots, x_k) \\ \text{dom}(\llbracket x_j := ? \rrbracket) &= \{x \in \mathbb{F}^k\} \end{aligned}$$

- Wächter haben folgende Auswirkungen

$$\begin{aligned} \llbracket p \neq 0 \rrbracket x &= (x_1, \dots, x_k) \\ \text{dom}(\llbracket p \neq 0 \rrbracket) &= \{x \in \mathbb{F}^k \mid p(x) \neq 0\} \end{aligned}$$

- *skip*-Anweisungen beeinflussen weder die Domäne, noch den Zustandsvektor, die bewirkte Transformation ist die Identitätsfunktion:

$$\llbracket skip \rrbracket x = (x_1, \dots, x_k)$$

$$\text{dom}(\llbracket skip \rrbracket) = \{x \in \mathbb{F}^k\}$$

Wenn man die einzelnen Effekte zum Effekt des kompletten Programmablaufs kombiniert, erhält man eine partielle Transformation, die auf Programmläufe definiert ist. Diese partielle Transformation kann man durch mehrere Polynome $q_0, \dots, q_k \in \mathbb{F}[\mathbf{X}]$ darstellen, so dass für alle $f \equiv \llbracket r \rrbracket$ gilt:

$$f(x) = (q_1(x), \dots, q_k(x)) \quad \forall x \in \text{dom}(f)$$

$$\text{dom}(f) = \{x \in \mathbb{F}^k \mid q_0(x) \neq 0\}$$

Konkret setzen sich diese Polynome durch rekursive Anwendung der Einzelfunktionen zusammen:

$$\llbracket r \rrbracket x = \llbracket r_1; r_2; \dots; r_m \rrbracket x = \llbracket r_2; \dots; r_m \rrbracket \circ (\llbracket r_1 \rrbracket x)$$

Der Einfluß auf die Domäne wird auch rekursiv berechnet:

$$x \in \text{dom}(\llbracket r; r' \rrbracket) \quad \text{gdw}$$

$$x \in \text{dom}(\llbracket r' \rrbracket \circ \llbracket r \rrbracket) \quad \text{gdw}$$

$$q_0(x) \neq 0 \wedge q'_0(q_1(x), \dots, q_k(x)) \neq 0 \quad \text{gdw}$$

$$(q_0 \cdot q'_0(q_1/x_1, \dots, q_k/x_k))(x) \neq 0$$

2.1.4 Zusammenfassung

Eine analytische Betrachtung von Algorithmen macht eine abstrahierte Darstellung der relevanten Methoden notwendig. Für die Analyse von Programmcode auf polynomielle Invarianten eignet sich die Verwendung von Kontrollflußgraphen. Mit der operationellen Semantik kann man alle nicht relevanten Teile des Programmcodes ausblenden und sich auf die Teile des Programmes konzentrieren, die den für Invarianten relevanten Zustand eines Programmes ändern, und die nicht zu komplex für diese Analyse sind. Die formale Spezifikation dieser Semantik ist das Ergebnis dieser Bemühungen.

2.2 Verifikation von polynomiellen Invarianten

Im nächsten Schritt soll nun erläutert werden, wie in [?] vorgegangen wird, um eine beliebige polynomielle Gleichung auf ihre Gültigkeit entlang einer Menge von Programmläufen hin zu untersuchen.

2.2.1 Formale Problemspezifikation

Dazu muss zuerst ein Weg gefunden werden um den Einfluß des zu analysierenden Programmcodes auf die betrachtete Relation auszudrücken. Man wählt sich also eine polynomielle Relation $p = 0$ mit einem beliebigen $p \in \mathbb{F}[\mathbf{X}]$, die am frei wählbaren Zielprogrammumpunkt t bezogen auf dessen Zustand x_t gültig sein soll. Dieser Zustand x_t erfüllt die Relation p , wenn gilt: $p(x_t) = 0$. Diese Eigenschaft drücken wir durch $x_t \models p$ aus.

Die polynomielle Relation p ist für alle die Zustände x gültig, die in der Domäne des Programmlaufs r enthalten sind und die nach Anwendung des Programmlaufs die Relation erfüllen. Die allgemeinste notwendige Vorbedingung für die Gültigkeit von $p(x_t) = 0$ ist daher

$$x \notin \text{dom}(\llbracket r \rrbracket) \vee \llbracket r \rrbracket \models p$$

Diese Bedingung lässt sich unter der Berücksichtigung, dass jeder Programmlauf r durch spezifische Polynome q_0, \dots, q_k dargestellt werden kann, so umformulieren:

$$\begin{aligned} \forall x \quad & x \notin \text{dom}(\llbracket r \rrbracket) \vee \llbracket r \rrbracket x \models p \\ \text{gdw} \quad & q_0(x) = 0 \vee p(q_1(x), \dots, q_k(x)) = 0 \\ \text{gdw} \quad & q_0(x) = 0 \vee p[q_1/x_1, \dots, q_k/x_k](x) = 0 \\ \text{gdw} \quad & (q_0 \cdot p[q_1/x_1, \dots, q_k/x_k])(x) = 0 \end{aligned}$$

Die allgemeinste Vorbedingung kann also wieder als polynomielle Relation ausgedrückt werden. Wenn man diese Relation als Funktion definiert, die jeder polynomiellen Relation die zugehörige allgemeinste Vorbedingung zuordnet, so bekommt man die totale Funktion:

$$\llbracket r \rrbracket^\top p_t = q_0 \cdot p[q_1/x_1, \dots, q_k/x_k]$$

Die einzige Vorbedingung, die zum Programmstart x_s gültig sein kann, ist, daß keine Bedingung gilt, außer $0 = 0$. Daher lässt sich folgendes Lemma aufstellen:

$$x \notin \text{dom}(\llbracket r \rrbracket) \vee \llbracket r \rrbracket \models p_t \quad \Leftrightarrow \quad \forall_{r \in \mathbf{R}(s)} \llbracket r \rrbracket^\top p_t = 0$$

Um herauszufinden, ob eine polynomielle Relation p_t an einem Zielknoten gültig ist, sind wir daran interessiert, ob gilt:

$$\{\llbracket r \rrbracket^\top p_t \mid r \in \mathbf{R}(s)\} \subseteq \{0\}$$

Für diese Menge muß eine endliche Darstellung gefunden werden, die effizient berechnet werden kann. Diese Menge S der allgemeinsten Vorbedingung ist immer eine Menge von Polynomen, also immer eine Teilmenge des Polynomfeldes $S \subseteq \mathbb{F}[\mathbf{X}]$. Für Teilmengen F von Ringen wie dem Polynomfeld $\mathbb{F}[\mathbf{X}]$ gilt, dass immer ein kleinstes Ideal $\langle F \rangle$ existiert, dass F enthält:

$$\langle F \rangle = \{r_1 g_1 + \dots + r_n g_n \mid n \geq 0; r_i \in R; g_i \in G\}$$

F wird in diesem Fall auch Generatormenge von $\langle F \rangle$ genannt. Um zu überprüfen, ob $F \subseteq \{0\}$ ist, eignet sich folgende Eigenschaft von Idealen:

$$\forall F \subseteq R \quad \langle F \rangle = \{0\} \Leftrightarrow F \subseteq \{0\}$$

Folglich kann man in unserem Fall, ohne Genauigkeit zu verlieren, mit Idealen statt mit Mengen rechnen. Laut [?] wird jedes Ideal I , über einem kommutativen Polynomring über endlich viele Variablen endlich erzeugt. Das heißt:

$$\exists G \subseteq \mathbb{F}[\mathbf{x}] \quad |G| \neq \infty \wedge I = \langle G \rangle$$

Um also die Gültigkeit einer polynomiellen Relation an einem Zielknoten zu überprüfen, bleibt noch, das Ideal I zu berechnen:

$$I = \langle \{ \llbracket r \rrbracket^\top p_t \mid r \in \mathbf{R}(s) \} \rangle$$

Diese Berechnung kann durch Analyse der Programmablaufmengen erfolgen, wie im folgenden Abschnitt beschrieben wird.

2.2.2 Abstrakte Interpretation der operationellen Semantik

Das Ideal I aus dem letzten Abschnitt kann als Abstraktion $\mathbf{R}^\#(s)$ der Menge der Programmläufe $\mathbf{R}(s)$ vom Startpunkt s bis zum Zielpunkt t gesehen werden. Die Berechnung dieses Ideals wird durch eine abstrakte Interpretation $\mathbf{R}^\#_{p_t}$ des Teilmengenrelationensystems \mathbf{R} aus der operationellen Semantik ermöglicht. Es soll also das Teilmengenrelationensystem über Programmlaufmengen

$$[\mathbf{R1}] \quad \mathbf{R}(t) \supseteq \{\varepsilon\}$$

$$[\mathbf{R2}] \quad \mathbf{R}(u) \supseteq f_e(\mathbf{R}(v)), \text{ wenn } e = (u, s, v) \in E$$

abstrahiert werden auf ein Teilmengenrelationensystem über Ideale durch die Anwendung einer Abstraktionsfunktion α :

$$\alpha : 2^{\text{Runs}} \rightarrow I_X$$

α soll jeden Programmlauf auf ein Ideal, das seine allgemeinste Vorbedingung enthält, abbilden:

$$\alpha(R) = \langle \{ \llbracket r \rrbracket^\top p_t \mid r \in R \} \rangle$$

Durch die Anwendung dieser Abstraktion werden aus Programmläufen Ideale:

$$\forall u \in N \quad \mathbf{R}^\#(u) = \alpha(\mathbf{R}(u))$$

So läßt sich das Teilmengenrelationssystem $\mathbf{R}^\#_{p_t}$ formulieren:

$$[\mathbf{R1}]^\# \quad \mathbf{R}^\#(t) \supseteq \langle \{ p_t \} \rangle$$

$$[\mathbf{R2}]^\# \quad \mathbf{R}^\#(u) \supseteq f_e^\#(\mathbf{R}^\#(v)), \text{ if } e = (u, s, v) \in E$$

2 Theoretische Hintergründe

Dabei ist die Abstrakte Funktion $f_e^\#$ passend zu f_e so definiert, dass gilt:

$$f_e^\# : I_X \rightarrow I_X$$

$$f_e^\#(\langle G \rangle) = \langle \{ \llbracket r \rrbracket^\top p_t \mid r \in \mathbf{R}(e), p \in G \} \rangle$$

also kommutiert $f_e^\#$ mit f_e unter der Abbildung α :

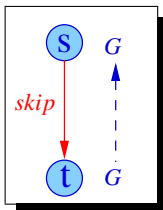
$$f_e^\#(\alpha(R)) = \alpha(f_e(R))$$

Wie aus [?] ersichtlich, läßt sich $f_e^\#$ also auf folgende Berechnungsschemata zurückführen:

$$f_e^\# \langle G \rangle = \begin{cases} \langle G \rangle, & \text{wenn } A(e) \equiv \text{skip} \\ \langle \{p \cdot q \mid q \in G\} \rangle, & \text{wenn } A(e) \equiv p \neq 0 \\ \langle \{q[p/x_j] \mid q \in G\} \rangle, & \text{wenn } A(e) \equiv x_j := p \\ \langle \bigcup \{ \pi_j(q) \mid q \in G \} \rangle, & \text{wenn } A(e) \equiv x_j := ? \end{cases}$$

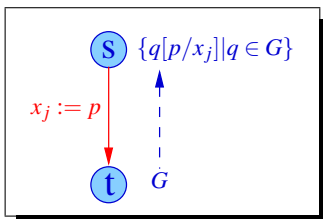
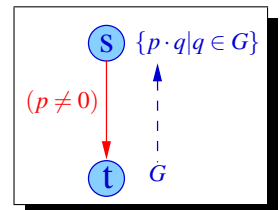
$$\text{mit } \pi_j : \mathbb{F}[\mathbf{X}] \rightarrow 2^{\mathbb{F}[\mathbf{X}]}$$

$$\pi_j(p) = \{p_i \mid p = p_0 + p_1 \cdot x_j + \dots + p_b \cdot x_j^b \wedge i \in [0, \dots, b]\}$$



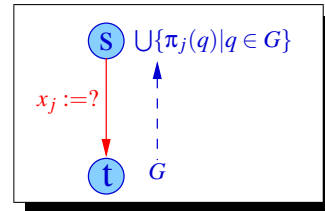
Die *skip* Anweisung wird dabei also so behandelt, daß alle an Programmpunkt t gültigen Invarianten einfach an den Programmpunkt übernommen werden. Das führt dazu, dass an eine solche Kante keine Transformation am Programmpunkt s auslöst.

Polynomielle Wächter wirken sich dagegen sehr wohl auf die Quellprogrammpunkte aus. Ein polynomieller Wächter führt dazu, daß alle Generatorpolynome q des Zielprogrammpunktes t mit dem Polynom p multipliziert werden müssen, um zu den Generatorpolynomen des Quellprogrammpunktes s zu werden.



Polynomielle Zuweisungen an Programmvariablen x_j führen zu Substitutionen in den Generatorpolynomen q des Zielpunktes t . Die Generatorpolynome des Programmpunktes s werden berechnet, durch die Ersetzung jedes Vorkommens der Programmvariable x_j durch das Polynom p .

Nichtdeterministische Zuweisungen an Programmvariablen x_j führen zu einer Beseitigung dieser Programmvariablen aus den Generatorpolynomen. Die Generatorpolynome des Programmpunkts s werden daher durch die Koeffizientenpolynome der verschiedenen Potenzen der Variablen x_j in den Generatorpolynomen q des Programmpunkts t bestimmt.



Dies sind alle Mittel, die für die Analyse von Programmen in Bezug auf polynomielle Invarianten benötigt werden. Als nächstes ist die Realisierung eines Fixpunktalgorithmus nötig, der das Teilmengenrelationensystem $\mathbf{R}_{\rho_t}^\#$ löst.

2.2.3 Der Fixpunktalgorithmus

Vom theoretischen Standpunkt her lässt sich also ein sehr klarer Algorithmus zur Überprüfung von polynomiellen Invarianten formulieren. Eine direkte Implementierung eines Fixpunktalgorithmus zu den in Abschnitt 2.2.2 eingeführten Teilmengenrelationensystemen würde sehr viele Neuberechnungen der zugrundeliegenden Ideale oder Module nach sich ziehen. Da die zugehörige Rechenoperation doch sehr teuer ist, ist es naheliegend, nach Wegen zu suchen, diese Berechnungen zu umgehen.

Der in Abbildung 2.1 realisierte Ansatz dazu ist, statt einer naiven Fixpunktiteration eine inkrementelle Fixpunktiteration zu starten, also nicht über ganzen Idealen oder Modulen zu iterieren, sondern jeweils nur einzelne Generatorkandidaten über den Kontrollflußgraphen zu propagandieren. Diese Generatorkandidaten werden bei Erreichen eines Knotens dahingehend überprüft, ob sie bereits von der bislang existierenden Basis abgedeckt werden. Falls sie nicht schon dargestellt werden können, bedeuten sie für die Iteration einen Informationsgewinn an dieser Stelle, und werden zur Basis hinzugenommen. Des weiteren werden neue Generatoren auch über alle ausgehenden Kanten unter den jeweiligen Transformationen weiterpropagandiert. Sobald ein Generatorkandidat bereits von der entsprechenden Basis abgedeckt werden kann, wird er nicht mehr weitergegeben. Nachdem alle Module und Ideale endlich erzeugt werden können, terminiert der Algorithmus, sobald das System vollständig ist.

Die Terminierung des Algorithmus ist folgendermaßen gegeben: Nicht terminieren könnte der Algorithmus nur an einer Stelle. Dabei handelt es sich um die `while`-Schleife aus Zeile 8. Die Abbruchbedingung dieser Schleife wird erst dann erfüllt, wenn W irgendeinmal leer wird. Die einzigen Stellen, an denen allerdings W überhaupt aufgefüllt wird, sind alle durch die `if`-Verzweigung in Zeile 12 abgeschirmt. Dieser Bereich wird also nicht mehr durchlaufen, wenn $G[v]$ ein Ideal erzeugt, das alle an diesem Programmpunkt v gültigen Polynome erzeugt. Laut Hilberts Theorem, wie in [?] geschildert, kann jedes Ideal, auf das wir im Zuge dieser Analyse stoßen, mit einer endlichen Generatormenge erzeugt werden. Es wird also nach einer endlichen Anzahl von Schritten eine Menge von Polynomen erzeugt worden sein, die alle Polynome generiert, die an diesem Programmpunkt gelten sollen. Folglich werden keine neuen Aufträge in die Worklist W eingetragen, und die `while`-Schleife terminiert.

Eine Möglichkeit zur Reduktion der Komplexität dieses Algorithmus findet sich bei genau-

```

1 // Fixpoint algorithm with target node and target polynom given
  Set fixpointiteration (Node  $u_t$ , Polynom  $p_t$ , Set Edges, Set Nodes) {
  // For each node create an empty set of generator polynoms
  Set []  $G \leftarrow$  new Set[Nodes];
  forall ( $u \in$  Nodes)  $G[u] \leftarrow \emptyset$ ;
6 // Create a new worklist with the target tuple
  Set  $W \leftarrow \{(p_t, u_t)\}$ ;
  while ( $W \neq \emptyset$ ) {
  // in each iteration pop an arbitrary tuple from the worklist
  ( $p, v$ )  $\leftarrow$  extract( $W$ );
11 // test, whether the propagated invariant is a new one
  if ( $p \notin \langle G[v] \rangle$ ) {
  // add the newly found invariant to this set of generators
   $G[v] \leftarrow G[v] \cup \{p\}$ ;
  // propagate a new invariant, depending on it's kind
16 forall ( $(u', skip', v) \in$  Edges) // skip
     $W \leftarrow W \cup \{(p, u)\}$ ;
    forall ( $(u', x_j := t', v) \in$  Edges) // assign
       $W \leftarrow W \cup \{(p[t/x_j], u)\}$ ;
    forall ( $(u', (q \neq 0)', v) \in$  Edges) // assert
21  $W \leftarrow W \cup \{(p \cdot q, u)\}$ ;
    forall ( $(u', x_j := ?', v) \in$  Edges) // invalidate
      let  $p \Rightarrow p_0 + p_1 x_j^1 + \dots + p_k x_j^k$ 
        in  $W \leftarrow W \cup \{(p_0, u), \dots, (p_k, u)\}$ ;
  }
26 }
  // returns the set of generator polynoms for the start node
  return  $\langle G[u_{start}] \rangle$ ;
}

```

Abbildung 2.1: Fixpunktalgorithmus

rer Betrachtung der Einzeloperationen aus 2.1. In Zeile 14 erfolgt zum Beispiel die Erweiterung des Ideals um einen neuen Generator über die Anweisung:

```

14  $G[v] \leftarrow G[v] \cup \{p\}$ ;

```

Unmittelbar davor in Zeile 12 geht jedoch die Überprüfung, ob das Polynom p bereits in diesem Ideal $G[v]$ enthalten ist, voran:

```

12 if ( $p \notin \langle G[v] \rangle$ )

```

Um diese Inklusion zu überprüfen, wird intern überprüft, ob p durch die Generatorenmenge $G[v]$ dargestellt werden kann. Diese Überprüfung erfolgt durch die Reduktion des Kandidaten durch die Generatoren auf den Restteil, der nicht durch die Generatoren dargestellt werden kann. Ist dieser Reduktionsrest nun Null, so konnte der Kandidat erfolgreich durch die anderen Generatoren dargestellt werden, und ist somit im Ideal enthalten. Bleibt jedoch ein unreduzierbarer Restteil bestehen, so konnte der Kandidat nicht durch vorhandene Generatoren

dargestellt werden.

Normalerweise würde man nun wie im inkrementellen Fixpunktalgorithmus aus 2.1 den so gefundenen neuen Generator einfach hinzufügen. Eine theoretisch gesehen bessere Möglichkeit wäre nun, anstatt des langen ursprünglichen Kandidaten, den Reduktionsrest als neuen Generator mit in das Ideal aufzunehmen, da dieser ja bereits auf seinen wesentlichen Neubeitrag zum Ideal reduziert wurde. Dies erscheint besonders im Hinblick darauf geschickt, dass durch die Substitutionen, die bei dieser Fixpunktiteration anfallen, die propagandierten Generatorkandidaten enorm anwachsen. Eine Verwendung der Reduktionsreste als neue Generatoren würde potentiell der anwachsenden Generatorlänge, und damit dem Bremsen der Verarbeitungsgeschwindigkeit, entgegenwirken.

Diese Optimierung ist nicht besonders schwer in den bereits vorhandenen Fixpunktalgorithmus aus 2.1 einzubetten, Zeile 12 bis 14 müssten nur ersetzt werden mit den Anweisungen aus Abbildung 2.2. Zur Sicherheit sind in diesem Quellcodeteil auch noch Prüfungen eingebettet, ob nun der ursprüngliche Generatorkandidat oder der Reduktionsrest kleiner sind. Wie genau die Semantik dieses Tests ist, wird im Abschnitt 3.2.5 erläutert und anhand von Benchmarks bewertet.

```

12      // obtain reduced polynomial
      Polynomial r = reduce(p, G[v]);
14      if (r ≠ 0) {
15          // test which polynom to add to generator list
          // with < as an arbitrary relation
17          if (r < p) G[v] ← G[v] ∪ {r};
18          else G[v] ← G[v] ∪ {p};

```

Abbildung 2.2: Optimierte Generatorentwicklung

Die Laufzeit dieses inkrementellen Fixpunktalgorithmus ist insgesamt abhängig von mehreren Einflüssen: Einerseits ist nicht offensichtlich, wie oft und wie lange Polynome erzeugt werden, die noch nicht in den Idealen der jeweiligen Programmpunkte enthalten sind. Andererseits ist die Laufzeit ausserdem davon abhängig, wie schnell berechnet werden kann, ob ein Polynom bereits in dem zur jeweiligen Generatormenge zugehörigen Ideal enthalten ist. Zudem trägt auch die Propagation der durch Substitution schnell wachsenden Polynome erheblich zur Komplexitätssteigerung bei.

Genauere Antworten zur Verwendbarkeit dieses Algorithmus in der Praxis soll durch die anschliessende experimentelle Implementation geklärt werden.

2.2.4 Zusammenfassung

Die formale Spezifikation der Gültigkeitsbedingung von polynomiellen Invarianten ermöglicht die Umformung der operationellen Semantik der Kontrollflußgraphen aus dem ersten Teil in ein abstraktes problembezogenes Teilmengenrelationensystem.

Dieses Teilmengenrelationensystem ist dank der Abstützung auf Ideale und ihre Generatormengen durch einen Fixpunktalgorithmus lösbar. Statt einer naiven Fixpunktiteration wählt man in diesem Fall eine inkrementelle Iteration. Die Laufzeit dieses Algorithmus ist nicht direkt abzuschätzen, potentiell könnten sehr lange Laufzeiten im ungünstigsten Fall auftreten. Genauere Ergebnisse soll die Auswertung des Prototypen im nächsten Kapitel erbringen.

2.3 Herleitung von polynomiellen Invarianten festen Grades

Der Algorithmus aus dem vorigen Abschnitt setzt voraus, dass bereits eine Vermutung besteht, welche Polynomgleichung eine Invariante für das betrachtete Programm darstellt. Diese Invariante kann dann verifiziert werden. Für die praktische Verwendung ist es jedoch wünschenswert, gar keine Überlegungen darüber anstellen zu müssen, welche Polynome dafür in Frage kommen, sondern sich diese automatisch berechnen zu lassen. Auch für dieses Problem bietet [?] eine entsprechende Vorgehensweise, die sehr stark auf dem bereits im letzten Abschnitt geschilderten Algorithmus aufbaut.

Nötig dafür ist eine erweiterte Untersuchung der allgemeinsten Vorbedingung. Wenn nämlich ein generisches Polynom p_d beliebigen Grades d an die Funktion $\llbracket r \rrbracket^\top$ übergeben wird, so enthält das resultierende Ideal alle Bedingungen, die mindestens notwendig sind, damit das generische Polynom erfüllt wird; also ein Gleichungssystem über die generischen Koeffizienten von p_d . Wenn also dieses Gleichungssystem aufgelöst wird, kann die eigentlich geltende Invariante rekonstruiert werden.

2.3.1 Erzeugung eines generischen Polynoms

In einem generischen Polynom mit Grad d befinden sich beliebige Monome $b \cdot x_1^{i_1} \cdot \dots \cdot x_k^{i_k}$, für die gilt:

$$\sum_{m=1}^k i_m \leq d$$

$$D_d = \{(i_1, \dots, i_k) \mid \sum_{m=1}^k i_m \leq d\}$$

Ein generisches Polynom p_d mit Grad d sieht so aus:

$$p_d = \sum_{\sigma=(i_1, \dots, i_k) \in D_d} \mathbf{a}_\sigma \cdot x_1^{i_1} \cdot \dots \cdot x_k^{i_k}$$

Bei der Definition dieses Polynoms fallen also zusätzliche Variablen \mathbf{a}_σ an, die alle in der Menge \mathbf{A}_d enthalten sind:

$$A_d = \{\mathbf{a}_\sigma \mid \sigma \in D_d\}$$

$$|A_d| = \binom{|x| + d}{d}$$

Die Monome m müssen auch algorithmisch konstruiert werden, was in Abbildung 2.3 gezeigt wird. Mit Hilfe dieses Algorithmus kann also die Monome für ein generisches Polynom er-

```

// d: desired total maximal degree
// v: maximal number of variables per monomial
Set partition(int d, int v) {
    return partition(v-1, d, new int[v]);
5 }
// r: remaining degree points
// s: start of cursor
// e: array of exponents
Set partition(int s, int r, int[] e) {
10 if (r=0) // we have already reached maximum degree
    return {e};
    else { // still some degree points left
        Set P=∅; // collects all valid exponent distributions
        forall (i≤s) { // advance cursor sequentially
15     ei = ei + 1; // increment cursor element
        P = P ∪ partition(i, r-1, e); // continue with next degree
        ei = ei - 1; // restore cursor element
        }
        return P; // return all distributions so far
20 }
    }
}
```

Abbildung 2.3: Exponenten für Monome erzeugen

zeugen, auf das man dann den Algorithmus zur Bestimmung der allgemeinsten Vorbedingung anwendet.

2.3.2 Rekonstruktion der Invarianten

Als Resultat der Berechnung der allgemeinsten Vorbedingung für ein generisches Polynom erhält man nun ein System von Gleichungen über den zusätzlich eingeführten Variablen \mathbf{a}_σ . Dieses Gleichungssystem ist ein lineares Gleichungssystem, da durch keine einzige der auf die Polynome angewendeten Funktionen der Grad der generischen Koeffizienten erhöht werden kann [?]. Lineare Gleichungssysteme können einfach durch Umwandlung in die Zeilenstufen-Form aufgelöst werden. Da die Verwendung von Idealen und Polynomen jedoch nicht besonders geeignet ist, die automatische Ableitung von Invarianten vorzunehmen verweise ich hier auf Abschnitt 2.3.5, in welchem das Verfahren für Module und Vektoren beschrieben ist.

2.3.3 Anpassung des Algorithmus

Um also polynomielle Invarianten herzuleiten sind nur kleine Vor- und Nachverarbeitungen um den bereits existierenden Algorithmus zur Verifikation von polynomiellen Invarianten not-

wendig. Diese Veränderung werden in Abbildung 2.4 verdeutlicht.

```

// infers invariants up to degree d
Polynom infer(int d, State u_t, Set Vars, Set Edges, Set Nodes) {
    // create generic polynomial
    Polynom p_d = 0;
5   forall(σ ∈ partition(d, |Vars|))
        p_d = p_d + a_σ.merge(Vars, σ);
    // get weakest precondition for generic polynomial
    Ideal I = fixpointiteration(u_t, p_d, Edges, Nodes);
    // calculate values for generic coefficients
10  Vector v = resolveLES(I);
    // substitute coefficients against values
    forall(a_σ ∈ A_d)
        p_d = p_d[v_a_σ/a_σ];
    return p_d;
15 }

```

Abbildung 2.4: Polynomielle Invarianten herleiten

2.3.4 Vektoren als Alternative

Alternativ zur Errechnung der Vorbedingung für ein generisches Polynom ist die Umformulierung des Problems als Berechnung einer Vorbedingung für die Gültigkeit eines generischen Vektors. Die einzelnen Komponenten der Vektoren sind dabei zu Anfangs die einzelnen Monome, wie sie auch im generischen Polynom vorkommen. Die Multiplikation mit einer eigenen Unbekannten spart man sich in diesem Fall für jedes Monom, da die Komponentenstruktur von Vektoren in diesem Fall die gewünschte Kennzeichnung der Monome übernehmen.

Eine Basis aus mehreren Vektoren wird dabei zusammengefasst als Modul, für das sich analog zu den Idealen auch Gröbnerbasen bestimmen lassen, wie in [?] geschildert wird. Die eigentliche Berechnung dieser Basen verspricht etwas günstiger abzulaufen, da die zusätzlichen generischen Variablen \mathbf{a}_σ , die im Falle einer Formulierung mit Polynomen notwendig wären wegfallen. Zwar werden sie im Laufe der Basisbestimmung dann wieder intern erzeugt, jedoch speziell gekennzeichnet, und auf eine Rechnung reduziert, die auf sie verzichtet. Daher kann die Basisbestimmung mit weniger Variablen durchgeführt werden, und läuft komplexitätstechnisch günstiger ab.

Um also beliebige Relationen bis zu einem gewissen Grad n analysieren zu können, generiert man daher mit Hilfe des Algorithmus aus Abbildung 2.3 den n -vollständigen Vektor v_t . Diesen propagandiert man dann an Stelle des generischen Polynoms p_t am Zielknoten x_t als gewünschte Zielbedingung, und rechnet die dazu nötige Vorbedingung aus:

$$[[r]]^\top v_t = q_0 \cdot v[q_1/x_1, \dots, q_k/x_k]$$

Um herauszufinden, welche Relationen nun am Zielknoten gelten, ist es nötig, folgende Relationenmenge auszuwerten:

$$\{[[r]]^\top v_t \mid r \in \mathbf{R}(s)\}$$

Um diese Auswertung vornehmen zu können, benötigen wir analog zur Vorgehensweise beim Nachweis einer einzelnen Relation eine endliche Darstellung der Menge aller gültigen Relationen. Die Mengen der gültigen Relationen werden in diesem Fall dargestellt durch Module, die auch wiederum über Generatoren repräsentiert werden können.

Die Verwendung von Modulen und Vektoren erfordert nur sehr geringe Modifikationen am ursprünglichen Fixpunktalgorithmus, wie in Abbildung 2.5 zu sehen ist.

```

// Fixpoint algorithm with target node and degree d given
Set fixpointiteration (Node u_t, int d, Set Vars, Set Edges, Set Nodes) {
  // For each node create an empty set of generator polynoms
  Vector v_t = (σ | σ ∈ map(partition(d, |Vars|), Vars));
5  Set [] G ← new Set[|Nodes|];
  forall (u ∈ Nodes) G[u] ← ∅;
  // Create a new worklist with the target tuple
  Set W ← {(v_t, u_t)};
  while (W ≠ ∅) {
10  // in each iteration pop an arbitrary tuple from the worklist
    (v, t) ← extract(W);
    // test, whether the propagated invariant is a new one
    if (v ∉ ⟨G[t]⟩) {
      // add the newly found invariant to this set of generators
15  G[t] ← G[t] ∪ {v};
      // propagate a new invariant, depending on it's kind
      forall ((s, skip, t) ∈ Edges) // skip
        W ← W ∪ {(v, s)};
      forall ((s, x_j := p', t) ∈ Edges) // assign
20  W ← W ∪ {(v[p/x_j], s)};
      forall ((s, (p ≠ 0)', t) ∈ Edges) // assert
        W ← W ∪ {(p · v, s)};
      forall ((s, x_j := ?, t) ∈ Edges) // invalidate
        let i ∈ [0...k]
25  in let v_i ⇒ (p_0 x_j^i, p_1 x_j^i, ..., p_k x_j^i)
          in W ← W ∪ {(v_0, u), ..., (v_k, u)};
    }
  }
  // returns the set of generator polynoms for the start node
30 return ⟨G[u_start]⟩;
}

```

Abbildung 2.5: Fixpunktalgorithmus mit Modulen und Vektoren

Im Großen und Ganzen gibt G nun nicht mehr eine Menge von Generatoren für ein Ideal, sondern für ein Modul an. Ansonsten verläuft der Algorithmus analog zur Version mit Polynomen und Idealen.

2.3.5 Rekonstruktion der Invarianten

Als Resultat der Fixpunktiteration aus dem vorigen Abschnitt erhalten wir eine Generatorenmenge, die als Basis für das Polynommodul am Startpunkt dient. Aus dieser Menge von Vektoren muß nun noch die Menge von gültigen Invarianten rekonstruiert werden.

Dazu betrachtet man die einzelnen Generatorvektoren als System von Relationen zwischen Koeffizienten der Monome des Generischen Polynoms. Nachdem jeder Vektor gleich null sein muß, trägt jeder Vektor zu einer Koeffizientenrelation der folgenden Form bei:

$$v_1 \Rightarrow \mathbf{a}_1 = p_1$$

⋮

$$v_k \Rightarrow \mathbf{a}_k = p_k$$

In diesem System versucht man nun, alle Koeffizienten mit den wenigen freien Variablen, die im System existieren, darzustellen, indem man die Gleichungen zuerst ineinander und dann in das generische Polynom einsetzt:

$$p_d = \sum_{\sigma=(i_1, \dots, i_k) \in D_d} \mathbf{a}_\sigma \cdot x_1^{i_1} \dots x_k^{i_k} [p_\sigma / \mathbf{a}_\sigma]$$

Das daraus resultierende Polynom muß dann nur noch nach Koeffizienten faktorisiert werden, um einzelne Teilpolynome q_σ zu erhalten, die genau den gesuchten Invarianten entsprechen:

$$p_d = \sum_{\sigma=(i_1, \dots, i_k) \in D_d} \mathbf{a}_\sigma \cdot q_\sigma$$

Den Algorithmus zu dieser Vorgehensweise findet man in Abbildung 2.6

```

// derives all invariants with a degree less or equal than d
Set derive (Node ut, int d, Set Vars, Set Edges, Set Nodes) {
  // calculate how much monoms are in the generic polynomial
  int #monoms =  $\binom{|Vars|+d}{d}$ ;
5 // calculate the module representing the weakest precondition
  Set M = fixpointiteration(ut, d, Vars, Edges, Nodes);
  // Generate generic polynomial
  Polynomial pd = 0;
  forall (σ ∈ partition(d, |Vars|)) {
10   pd = pd + aσ · merge(Vars, σ);
  }
  // initialize all coefficient factors with 0
  ∀ipi = 0;
  forall vk ∈ M {
15   // every vector in the module contributes to the coefficient factors
   let vk = (q1, ..., qk, 0, ..., 0);
   in pi = pi + qi;
  }
  // replace all generic coefficients by their polynoms
20 forall 0 < i ≤ #monoms {
   pd = pd[pi/ai];
  }
  // sort polynomials by remaining generic coefficients
  let pd =  $\sum_{i=1}^{\#monoms} \mathbf{a}_i \cdot r_i$ ;
25   in Set I = {ri | ri ≠ 0};
  // return valid invariants
  return I;
}

```

Abbildung 2.6: Algorithmus zur Herleitung von Invarianten

3 Experimentelle Implementation

Um die Erkenntnisse aus der theoretischen Abhandlung zu verifizieren und auf ihre Praxistauglichkeit hin zu überprüfen, ist die Implementation eines Prototypen unabdingbar. Im Rahmen dieser Diplomarbeit entstand zu diesem Zweck das Programmanalysewerkzeug mit dem Namen *Polyinvar*.

Im folgenden Teil sollen der Aufbau von *Polyinvar*, die Testumgebung, der Ablauf der Versuchsreihe und die aus den Tests gewonnenen Resultate erläutert werden.

3.1 Architektur

Die Umsetzung des Fixpunktalgorithmus aus dem letzten Kapitel in einen Prototypen erfolgte in modularer Bauweise, so daß nachträglich leicht einzelne Bausteine gegen Alternativen ausgetauscht werden können. Als Implementationssprache wurde *Java* gewählt, um sich nicht auf eine spezielle Hardwareplattform festzulegen.

Da jedoch die mathematischen Bibliotheken von *Java* nicht sonderlich gut für die Behandlung von Polynomen, Idealen und Standardbasen geeignet sind, stützt sich *Polyinvar* auf ein Spezialprogramm für diesen Bereich mit dem Namen *Singular* ab.

Für die grafische Veranschaulichung der Kontrollflußgraphen ist die Berechnung einer sinnvollen planaren Anordnung der Knoten und Kanten notwendig, was jedoch auch nicht Schwerpunkt dieser Arbeit sein sollte. Daher stützt sich *Polyinvar* auch in diesem Bereich auf Spezialprogramme ab, zum Beispiel auf *Graphviz* oder *aiSee*.

Auch die Generierung eines Kontrollflußgraphen aus Quellcode ist eher eine Routinesache als besonders relevant für diese Arbeit. Daher stellt *Polyinvar* ein Plugin für *Eclipse* bereit, das aus *Java*-Klassen mit Hilfe der *Eclipse* API *JDT* entsprechende Kontrollflußgraphen erstellt.

3.1.1 Aufgabenverteilung

Die Arbeitsweise von *Polyinvar* ist in Abbildung 3.1 dargestellt. Im ersten Schritt wird durch das mitgelieferte *Eclipse*-Plugin ein Kontrollflußgraph zur Verfügung gestellt. Die dafür notwendigen Klassen sind im Paket *cfggenerator.jar* enthalten, während die Deklaration des Datenmodells des Kontrollflußgraphen in die Datei *cfgstructure.jar* ausgelagert ist.

Der fertige Kontrollflußgraph wird nun direkt vom Fixpunktalgorithmus in *polyengine.jar* als Datenstruktur verwendet. Bei der Iteration werden die auftretenden Programmeigenschaften auf mathematische Operationen zurückgeführt, die *Polyinvar* mit Unterstützung von *Singu-*

lar auswertet. Die Ankoppelung an *Singular* ist über ein Adapterpaket `singular.jar` gelöst. Diese Schnittstelle ist gleichzeitig so flexibel gestaltet, daß ein beliebiges Mathematikpaket, welches die geforderten Operationen unterstützt mit Hilfe einer geeigneten Adapterklasse angedockt werden kann.

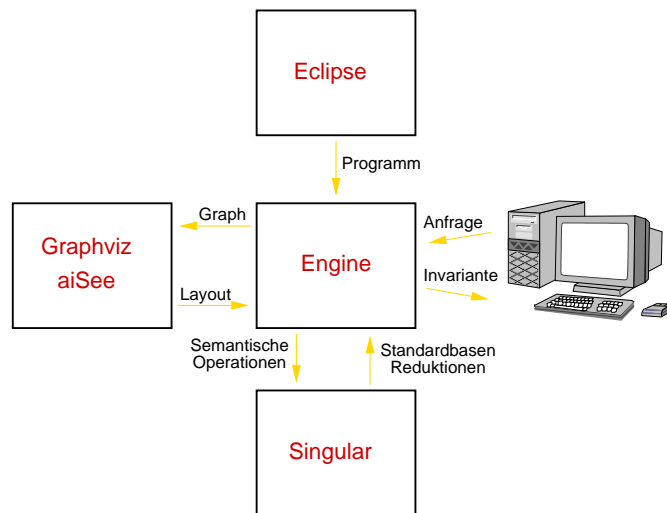


Abbildung 3.1: Prinzipielle Funktionsweise

Für die Steuerung des Programmes sind mehrere Schnittstellen vorgesehen. Für die Steuerung über die Kommandozeile wird ein Konsolenprogramm bereitgestellt, was durch die Verwendung von *CLI* sehr flexibel ist. Eine Integration in eigene Programme ist möglich, indem die bereitgestellte Nachrichtenschnittstelle aus `animator.jar` verwendet wird. Diese Schnittstelle wird auch für die grafische Animation des Algorithmus in `gui.jar` verwendet. Dabei verwendet die GUI die Pakete *JGraph* und *JGraphT* für die grafische Darstellung des Kontrollflußgraphen. Das Layout dieses Graphen wird erzeugt durch eine Anbindung an *aiSee* oder *Graphviz*.

3.1.2 Schnittstellen

In Abbildung 3.2 kann man die Beziehungen zwischen den einzelnen Programmkomponenten ablesen.

Parserschnittstelle

Eclipse fungiert für diese Programmanalyse als Frontend. Es bietet mit *JDT* eine API, die den Eclipse-eigenen Parser zugänglich macht. Es wäre auch gut möglich ein anderes Frontend für *Polyinvar* zu schreiben, das den Kontrollflußgraphen aus einer anderen Quellsprache erzeugt.

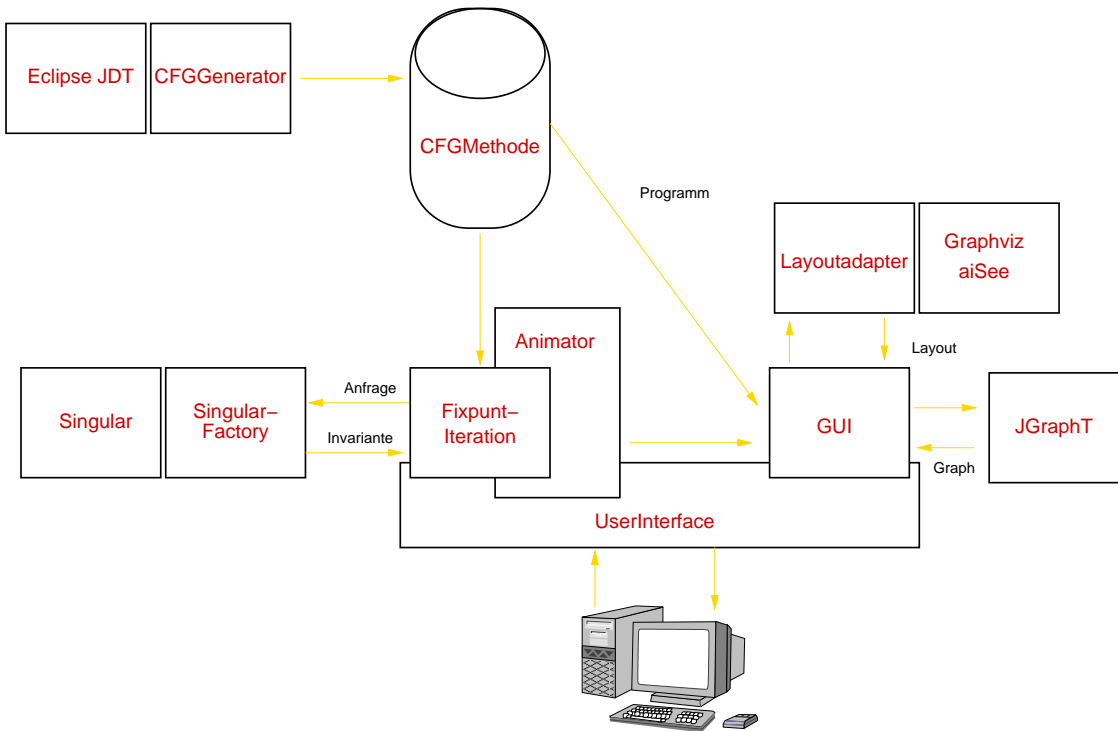


Abbildung 3.2: Detaillierte Funktionsweise

Allein die Ausgabe muss aus einer Hashtable von CFGMethoden, wie im nächsten Abschnitt beschrieben, bestehen. Diese kann dann als Objektstrom serialisiert werden, und von *Polyinvar* als Eingabedaten verarbeitet werden.

Um ein Plugin an *Eclipse* anzukoppeln muss eine entsprechende XML-Datei `plugin.xml` verfasst werden, in der die einzubindende Klasse spezifiziert werden muss. Der Kern des Plugins besteht in einer Sammlung von Visitoren, die den von *Eclipse* über *JDT* bereitgestellten Syntaxbaum durchlaufen, und einen für *Polyinvar* tauglichen Kontrollflußgraphen generieren.

Bei der Generierung des Kontrollflußgraphen wird allerdings nur eine Untermenge von *Java* berücksichtigt. An Stellen, wo Konstrukte auftauchen, die nicht unterstützt werden, aber nicht ignoriert werden können, wird daher mit nichtdeterministischen Zuweisungen gearbeitet. Nicht unterstützt werden Arrayzugriffe, Zugriffe auf Objekte und nichtlokale Variablen, sowie Methodenaufrufe. Strategien für die Unterstützung Teile dieser Merkmale werden später aufgelistet.

Kontrollflußgraph

Kontrollflußgraphen sind als CFGMethode Klassen in `cfgstructure.jar` deklariert. Sie bestehen aus einer Klasse `CFGState` für die Knoten und einer Klasse `CFGEde` für die Kanten

des Kontrollflußgraphen. Die einzelnen Kantenausprägungen werden durch Unterklassen realisiert. Dabei gilt die Zuordnung aus Tabelle 3.1

Kante	Klasse
nicht deterministische Zuweisung	CFGUnknownAssignmentEdge
deterministische Zuweisung	CFGAssignmentEdge
Wächter	CFGAssertionEdge
Skipkanten	CFGNullEdge

Tabelle 3.1: Zuordnung Kanten - Klassen

Eine Methode in `CFG`Methode wird durch Ihren Anfangs- und Ihren Endknoten repräsentiert, so dass eine komplette Klasse einfach als Sammlung ihrer Methoden dargestellt werden kann.

Mathematikpaket

Zur Umsetzung der abstrakten Interpretation der operationellen Semantik aus Abschnitt 2.2.2 ist eine mathematische Bibliothek notwendig, die eine Implementation der Darstellung von Polynomen, Idealen, Vektoren und Modulen und ihrer Operationen aufweist. Die benötigte Funktionalität ist als Paket von Interfaces gegeben, sodaß durch Ausnutzung des Abstract Factory Patterns an dieser Stelle ein geeignetes Paket für die Anknüpfung an das bevorzugte Mathematikpaket eingebunden werden kann.

Für die experimentelle Implementierung von *Polyinvar* wurde daher im Paket `singular.jar` ein Adapter für die Software *Singular* eingebaut. Er besteht zum größten Teil aus einem Adapter, der alle Operationen als Befehle für *Singular* umformuliert, und per Ausgabestrom an den Dialogmodus von *Singular* weiterreicht. Die Javaklassen fungieren dabei nur als Stellvertreter der entsprechenden Datenfelder in *Singular*.

Fixpunktiterationsalgorithmus

Das Zentrum von *Polyinvar* ist jedoch der Fixpunktalgorithmus, der im Paket `polyinvar.jar` untergebracht ist. Eine Skizze dieses Algorithmus ist in Abbildung 2.1 zu sehen.

Als Datenstruktur fungiert dabei der Kontrollflußgraph aus Abschnitt 3.1.2. Mit Hilfe des Mathematikpakets wird dann die allgemeinste Vorbedingung ausgerechnet und ausgewertet. Bei der Berechnung generiert *Polyinvar* Statusmeldungen und schickt diese an interessierte Beobachter.

Animation

Die Animation der Fixpunktiteration erfolgt über mehrere Schnittstellen. Im ersten Schritt erfolgt der Import der Programmstruktur als Kontrollflußgraph. Dazu wird wieder das Datenmodell aus `cfgstructure.jar` verwendet. Um einen Layoutvorschlag für diesen Graphen

zu erhalten muß eine Anfrage an das entsprechende Layoutprogramm gestellt werden. Dazu muss der Kontrollflußgraph zuerst in das richtige Eingabeformat gebracht werden, und das Ergebnis hinterher wieder richtig eingelesen und zugeordnet werden. Für diese Aufgaben sind Kindklassen von `LayoutAdapter` verantwortlich. Derzeit existiert ein Layoutadapter für *Graphviz*.

Für die Ausgabe des Graphen auf dem Bildschirm muß der Kontrollflußgraph noch in das Datenmodell für *JGraphT* überführt werden. *JGraphT* wiederum kapselt das Paket *JGraph*, mit welchem dann letztendlich der Kontrollflußgraph auf dem Bildschirm angezeigt werden kann.

Die eigentliche Umsetzung der Animation erfolgt durch geschickte Ankopplung an *Polyinvar* über den bereitgestellten Nachrichtenmanager. Dieser benachrichtigt den beobachtenden Prozess bei jeder Statusänderung und gibt ihm Gelegenheit, den Ablauf des Algorithmus Schritt für Schritt durchzuführen.

3.2 Testreihe

Als Testumgebung für die folgenden Leistungsmessungen und Tuningmaßnahmen diente ein handelsüblicher PC, dessen Leistungsdaten in Tabelle 3.2 abzulesen sind.

Merkmal	Ausprägung
Architektur	Intel
Prozessor	AMD Athlon XP 3000+
RAM	1024 MB
OS	Linux 2.6.4
Java	1.4.2
Singular	2.0.5

Tabelle 3.2: Versuchsumgebung

Für diesen Versuchsaufbau lagen bereits alle betrachteten Methoden in Form von Kontrollflußgraphen vor, so wie sie vom Kontrollflußgenerator einzeln vorher erzeugt wurden. Die Zeit, die der Kontrollflußgenerator dazu benötigte, den Graphen zu erstellen ist daher nicht Bestandteil der Zeitwerte, die in den folgenden Kapiteln angegeben werden.

Im folgenden sollen die einzelnen Beispiele, an Hand derer *Polyinvar* getestet wurde, kurz vorgestellt werden. Mit Hilfe der geometrischen Reihen und der Potenzsummen soll überprüft werden, wie leistungsfähig der Inferenzalgorithmus bei steigendem Polynomgrad arbeitet. Die Versuchsreihe mit den Variablen testmethoden dagegen soll beleuchten, wie sich die Zahl der deklarierten Variablen auf die Bearbeitungsgeschwindigkeit des Herleitungsalgorithmus auswirkt, schliesslich ist die Komplexität der Berechnungen von Basen und Reduktionen abhängig von der Anzahl der Variablen im zugrundeliegenden Polynomring.

3.2.1 Realisierung mit Modulen statt Idealen

Eine Implementierung der iterativen Fixpunktiteration aus 2.1 auf Basis von Polynomen und Idealen ist im ersten Moment naheliegend. Jedoch stellt man bei einer solchen Implementierung dieses Algorithmus sehr schnell bei ersten Tests fest, dass die Komplexität der zugrundeliegenden Operationen die Verwendung des Algorithmus bereits ab der Herleitung von Invarianten von Grad 2 sehr träge abläuft, und die Ableitung höherer Invarianten komplett verhindert.

Ein naheliegender Verdacht für den Grund dieses schlechten Abschneidens ist, dass die Ausstattung des generischen Polynoms p_d mit $|A_d|$ weiteren Variablen die Berechnung der Standardbasen für Ideale sehr negativ beeinflusst - diese ist doppelt exponentiell abhängig von der Anzahl der in \mathbf{X} bekannten Variablen [?].

Eine Möglichkeit, die Einführung von weiteren Variablen zu verhindern besteht im Umstieg von Idealen über Polynomen auf Module über Vektoren, wie in Abschnitt 2.3.4 geschildert wurde. Module ermöglichen das Ausdrücken derselben Eigenschaften wie Ideale, nur dass dabei ausser den Ringvariablen keinerlei zusätzliche Variablen benötigt werden. In [?] steht beschrieben, wie die Bildung von Gröbnerbasen für Module zwar die eben vermiedenen zusätzlichen Variablen trotzdem einführt, dieses aber gesondert behandelt, so dass das ganze Problem der Findung einer Gröbnerbasis für Module auf ein kleineres Problem der Findung einer Gröbnerbasis für Ideale zurückführbar ist. Dieses kleinere Problem ist somit auch effizienter lösbar.

Der Umstieg von Idealen auf Module führt damit in der Praxis dazu, daß Beispiele wie `potSumm2` oder `geoReihe2` und schwieriger zu analysierende Methoden überhaupt erst in erträglichen Zeitrahmen auf generische Invarianten hin analysiert werden können.

3.2.2 Die Geometrische Reihe

Das erste Serie von Beispielprogrammen ist angelehnt an die Berechnung der allgemeinen geometrischen Reihe, wie in Tabelle 3.3 zu erkennen ist. Der numerische Anteil des Methodennamens deutet an, welchen Grad die jeweilige Invariante haben müßte.

Name	Berechnung	Invariante	Zeit	
geoReihe1	$x = (z-1) \cdot \sum_{k=0}^K z^k$	$y = z^K$	$x = y - 1$	< 1s
geoReihe2	$x = \sum_{k=0}^K z^k$	$y = z^{K-1}$	$x \cdot (z-1) = yz - 1$	1s
geoReihe3	$x = \sum_{k=0}^K a \cdot z^k$	$y = z^{K-1}$	$x \cdot (z-1) = azy - a$	1,5s

Tabelle 3.3: Geometrische Reihen

```
// yielding 1+x-y == 0
public static void geoReihe1(String[] args){
    int z = Integer.parseInt(args[0]);
    int x = 1;
```

```

5      int y = z;
      while (args!=null){
          x = x*z + 1;
          y = z*y;
      }
10     x = x*(z-1);
    }
    // yielding 1+xz-x-zy == 0
    public static void geoReihe2(String[] args){
        int z = Integer.parseInt(args[0]);
15     int x = 1;
        int y = 1;
        while (args!=null){
            x = x*z + 1;
            y = y*z;
20     }
    }
    // yielding zx-x+a-azy == 0
    public static void geoReihe3(String[] args){
        int z = Integer.parseInt(args[0]);
25     int a = Integer.parseInt(args[1]);
        int x = a;
        int y = 1;
        while (args!=null) {
            x = x*z + a;
30     y = y*z;
        }
    }
}

```

Die Herleitung aller dieser Invarianten gelingt, die entsprechende Invariante ist in allen Fällen schnell berechnet.

3.2.3 Die Potenzsumme

Die zweite Serie von Beispielprogrammen basiert auf der Berechnung von Potenzsummen, wie in Tabelle 3.4 zu erkennen ist. Der numerische Anteil des Methodennamens deutet an, welchen Grad die jeweilige Invariante haben müßte.

Name	Berechnung	Invariante	Zeit	
potSumm1	$x = \sum_{k=0}^K 1$	$y = \sum_{k=0}^K 1$	$x = y$	< 1s
potSumm2	$x = \sum_{k=0}^K k$	$y = \sum_{k=0}^K 1$	$2x = y^2 + y$	1s
potSumm3	$x = \sum_{k=0}^K k^2$	$y = \sum_{k=0}^K 1$	$6x = 2y^3 + 3y^2 + y$	1,2s
potSumm4	$x = \sum_{k=0}^K k^3$	$y = \sum_{k=0}^K 1$	$4x = y^4 + 2y^3 + y^2$	104,8s

Tabelle 3.4: Potenzsummen

```
// yielding x=y
public static void potSumm1 (String[] args){
    int y = 0;
    int x = 0;
5    while (args!=null){
        y=y+1;
        x=x+1;
    }
}

// yielding 2x-y2-y == 0
10 public static void potSumm2 (String[] args){
    int y = 0;
    int x = 0;
    while (args!=null){
15        y=y+1;
        x=x+y;
    }
}

// yielding 6x-2y3-3y2-y == 0
20 public static void potSumm3 (String[] args){
    int y = 0;
    int x = 0;
    while (args!=null){
25        y=y+1;
        x=y*y+x;
    }
}

//yielding 4x-y4-2y3-y2 == 0
30 public static void potSumm4 (String[] args){
    int y = 0;
    int x = 0;
    while (args!=null){
35        y=y+1;
        x=y*y*y+x;
    }
}
```

Auch in diesem Fall kommt der Algorithmus auf jede Anfrage hin zum richtigen Ergebnis, braucht jedoch für die Herleitung der Invariante vom Grad 4 schon wesentlich länger.

3.2.4 Der Einfluß der Variablenanzahl

Mit der dritten Serie von Beispielprogrammen wird versucht, der Einfluß der Anzahl der vorhandenen Variablen auf die Analyse zu untersuchen. Dazu dient das Beispiel der Potenzsummenmethode `potSumm3`, jeweils mit einer zunehmenden Zahl von lokalen Variablen ausgestattet. Die Berechnung, sowie die Invariante bleibt dabei wie in Tabelle 3.5 gleich. Dennoch werden durch die Vorbelegung der Variablen weitere zufällige Invarianten entstehen. Der Zif-

fernanteil des Methodennamens deutet diesmal an, wieviele Variablen im Methodenrumpf deklariert wurden.

Name	Berechnung	Invariante
vartestX	$x = \sum_{k=0}^K k^2$	$y = \sum_{k=0}^K 1$ $6x = 2y^3 + 3y^2 + y$

Tabelle 3.5: Einfluß der Variablenzahl

```

public static void vartest2(String[] args){
    int y = 0;
    int x = 0;
    while (args!=null){
5         y=y+1;
           x=y*y+x;
    }
}

public static void vartest4(String[] args){
10    int y = 0;
    int x = 0;
    int z = 0;
    int a = 1;
    while (args!=null){
15         y=y+1;
           x=y*y+x;
    }
}

public static void vartest8(String[] args){
20    int y = 0;
    int x = 0;
    int z = 0, a = 1, b = 2, c = 3, d = 4, e = 5;
    while (args!=null){
25         y=y+1;
           x=y*y+x;
    }
}

public static void vartest16(String[] args){
30    int y = 0;
    int x = 0;
    int z = 0, a = 1, b = 2, c = 3, d = 4, e = 5;
    int f = 6, g = 7, h = 8, i = 9, j = 10, k = 11;
    int l = 12, m = 13;
    while (args!=null){
35         y=y+1;
           x=y*y+x;
    }
}

public static void vartest32(String[] args){
40    int ya = 0;
    int xa = 0;

```

3 Experimentelle Implementation

```
45     int za = 0, aa = 1, ba = 2, ca = 3, da = 4, ea = 5;
        int fa = 6, ga = 7, ha = 8, ia = 9, ja = 10, ka = 11;
        int la = 12, ma = 13, na = 14, oa = 15, pa = 16, qa = 17;
        int ra = 18, sa = 19, ta = 20, ua = 21, va = 22, wa = 23;
        int xb = 24, yb = 25, zb = 26, ab = 27, bb = 28, cb = 29;
        while (args!=null){
            ya=ya+1;
            xa=ya*ya+xa;
50     }
}
```

Tabelle 3.6 verdeutlicht das Ergebnis der Versuchsreihe. Erfreulich ist, dass scheinbar selbst 32 verschiedene Variablen die Laufzeit des Inferenzalgorithmus nicht so stark verschlechtern, dass er gar nicht terminiert. Auch das gute Abschneiden der 8 vorhandenen Variablen selbst bis Grad 6 ist ein guter Erfolg.

# Variablen \ Grad	3	4	5	6
vartest2	1.130s	3.152s	55.357s	1240.404s
vartest4	1.340s	3.787s	49.317s	267.384s
vartest8	2.147s	18.765s	219.799s	2560.095s
vartest16	13.910s	154.356s	>3000.000s	>3000.000s
vartest32	179.562s	> 3000.000s	>3000.000s	>3000.000s

Tabelle 3.6: Laufzeit abhängig vom Grad der Analyse

In Abbildung 3.3 findet man die Messwerte grafisch aufbereitet.

Der Grund für die starke Beeinflussung der Algorithmenlaufzeit durch die Variablenanzahl liegt in der doppelt exponentiell von der Variablenzahl abhängigen Laufzeit begründet [?]. Trotzdem wird dieser Effekt überlagert vom Einfluß des maximalen Grads der Polynome für die Herleitung von Invarianten. Das liegt daran, dass der die Anzahl der Monome des generischen Polynoms zwar auch von der Anzahl der vorkommenden Variablen abhängt, noch stärker allerdings vom Grad selbiger.

Dies ist auch sehr einfach an der Abschätzung (mit Hilfe der Stirling-Formel) der Anzahl der Monome des generischen Polynoms abhängig von seinem Grad d und den zur Verfügung stehenden Variablen $|x|$ zu sehen:

$$|A_d| = \binom{|x|+d}{d} \sim \left(1 + \frac{|x|}{d}\right)^d \cdot \left(1 + \frac{d}{|x|}\right)^{|x|}$$

Diese Formel weist exponentielles Wachstum in Abhängigkeit vom Grad d der zu generieren Monome auf. Nebenbei ist ihr Wachstum auch exponentiell abhängig von der Anzahl der beteiligten Variablen.

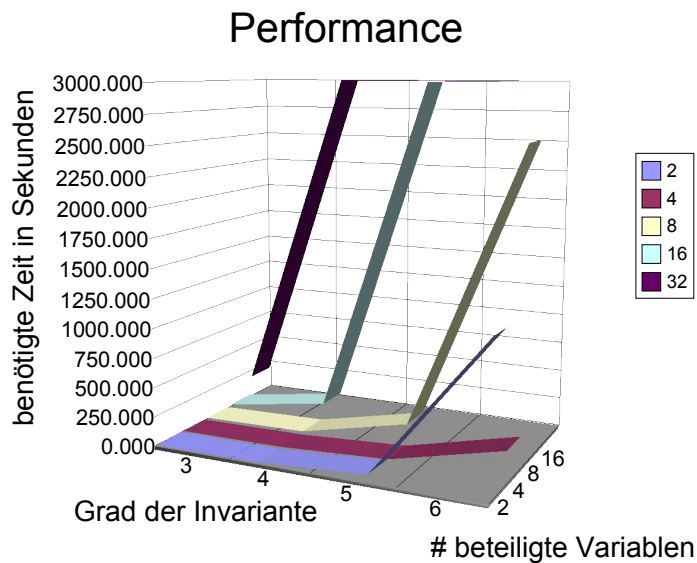


Abbildung 3.3: Einfluß von Grad und Anzahl der Variablen

Allein schon die große Anzahl der so erzeugten Monome sorgt also für ein überdimensional hohes Anwachsen der Polynomlänge, und verschlechtert somit die Laufzeit von Reduktionen und Basisbildungen enorm.

3.2.5 Optimierte Generatorauswahl

Ein weiterer Punkt, an dem noch Optimierung möglich ist, ist die Stelle im inkrementellen Fixpunktalgorithmus, an der die Generatormengen um einen neuen Generator angereichert werden. Wie schon in Abschnitt 2.2.3 angedeutet, ist nicht auf Anhieb klar, ob nun der Originalkandidat oder der Reduktionsrest besser für eine Generatorerweiterung geeignet sind. Verschiedene Strategien für die Auswahl des richtigen Kandidaten sind hier denkbar.

So wäre die herkömmliche Strategie, einfach den Originalkandidaten zu den Generatoren hinzuzufügen. Die gegenteilige Vorgehensweise, also die sture Hinzunahme des Reduktionsrests. Auch dynamische Verfahren, die sich nach der Anzahl der Monome der Kandidaten oder dem maximalen Grad der Monome richten sind denkbar. Eine Kombination von verschiedenen dynamischen Methoden wäre ausserdem naheliegend.

Das Verhalten der einzelnen Strategien ist in Tabelle 3.7 zu finden. In der Praxis ergeben sich also tatsächlich relevante Unterschiede zwischen den einzelnen Vorgehensweisen. Am besten schneidet die Methode ab, als primäres Auswahlkriterium für die Erweiterung den jeweils maximalen Grad der beiden Kandidaten zu nehmen, und dann bei Gleichheit die Anzahl der

Monome entscheiden zu lassen. Das überrascht dann doch insofern, als dass man denkt, die Reduktion des Originalkandidaten würde den Rest auf jeden Fall kürzer machen, und damit geeigneter als das Original.

Strategie	potSumm4 / -p 4	geoReihe3 / -p 4
Originalkandidat	126.38s	35.59s
Monomärmster Kandidat	114.17s	17.43s
Reduktionsrest	106.25s	17.55s
Kandidat mit kleinstem Grad	105.55s	20.45s
Kombinierte Kriterien	101.99s	17.98s

Tabelle 3.7: Effekt von verschiedenen Strategien zur Generatorenerweiterung

Bei günstigen Verhältnissen lassen sich also durch den Einsatz von geeigneten Strategien scheinbar wie in der Statistik zu sehen zwischen 20% und 50% Laufzeit einsparen.

3.3 Vergleich

Um die Mächtigkeit der Analyse, die hinter *Polyinvar* steht einordnen zu können, soll *Polyinvar* an dieser Stelle mit anderen Analysewerkzeugen für die Verifikation bzw. Inferenz von polynomiellen Invarianten verglichen werden. Für den Vergleich werden hier die Beispiele von der Homepage zum Analysewerkzeug von Rodríguez-Carbonell, welches in [?] erwähnt wird, herangezogen, obwohl die beiden Werkzeuge sich im Funktionsumfang nicht genau gleichen.

3.3.1 lcm

Als erstes Beispiel in diesem Zusammenhang soll hier das Programm `lcm`, betrachtet werden, dass nur leicht an die Quellsprache *Java* angepasst wurde.

```
// yields xu+yv-ab == 0
public static void lcm(String[] arg) {
    int x, y, u, v, a, b;
    a = Integer.parseInt(arg[0]);
5    b = Integer.parseInt(arg[1]);
    x = a;
    y = b;
    u = b;
    v = 0;
10    while (x != y) {
        while (x > y) {
            x = x - y;
            v = v + u;
        }
    }
}
```

```

15         while (x < y) {
                y = y - x;
                u = u + v;
            }
    }
20     //return u+v;
}

```

Bei diesem Beispiel konnte die Invariante $x \cdot u + y \cdot v = a \cdot b$ sowohl verifiziert als auch inferiert werden, bei einem Zeitaufwand, wie er in Tabelle 3.8 angegeben ist.

3.3.2 prod

Etwas anders gestaltet sich der Versuch mit Beispielprogramm `prod`, für das die Invariante $z + x \cdot y = a \cdot b$ gelten sollte. In diesem Fall schlägt schon die Verifikation der Invariante fehl, und die Inferenz ergibt zwar eine Menge von gefundenen Invarianten, die aber nicht die gesuchte Invariante enthalten.

```

public static void prod(String[] arg) {
    int x, y, z, a, b;
    a = Integer.parseInt(arg[0]);
    b = Integer.parseInt(arg[1]);
5   x = a;
    y = b;
    z = 0;
    while (y != 0) {
        if (y % 2 == 1) {
10          z = z + x;
            y = y - 1;
        }
        x = 2 * x;
        y = y / 2;
15    }
    //return z;
}

```

Wenn man sich nun `prod` im Detail ansieht, so kommt man zu dem Schluss, dass die beiden Divisionsoperatoren im Quelltext in Zeile 9 und 14 wohl doch zu viel Ungenauigkeiten in der Analyse produzieren, und so ein exaktes Ergebnis verhindern. Dennoch liegt auch hier sehr schnell ein Ergebnis vor.

3.3.3 cubes

Für das Beispielprogramm `cubes` wiederum sollen mehrere Invarianten überprüft, beziehungsweise gefunden werden. Dabei handelt es sich um die Invarianten $z = 6 \cdot n + 6$ sowie $x = n^3$ als auch $y = 3 \cdot n^2 + 3 \cdot n + 1$.

```
// yields a lot
public static void cubes(String[] args) {
    int N, n, x, y, z;
    N = Integer.parseInt(args[0]);
5    n = 0; x = 0; y = 1; z = 6;
    while (n <= N) {
        n = n + 1;
        x = x + y;
        y = y + z;
10        z = z + 6;
    }
}
```

Alle Invarianten konnten gezeigt werden, als auch, zusammen mit einer Menge weiterer Invarianten, hergeleitet werden. Dieses Beispiel ist bei nachträglicher Betrachtung auch besonders gut geeignet für eine Analyse mit *Polyinvar* - es treten keine Funktionsaufrufe auf, und jede Zuweisung ist divisionsfrei.

3.3.4 factor

Schliesslich wird noch für einen Vergleich die Methode `factor` angeführt, für die die Invariante $d^2 \cdot q - 4 \cdot r \cdot d + 4 \cdot rp \cdot d - 2 \cdot q \cdot d + 8 \cdot r = 8 \cdot n$ zutreffen müsste

```
public static void factor(String[] args) {
    int r, rp, q, d, s, t;

    int n = Integer.parseInt(args[0]);
5    int dd = Integer.parseInt(args[1]);

    d = dd;
    r = n % d;
    rp = n % (d - 2);
10    q = 4 * (n / (d - 2) - n / d);

    s = (int) Math.sqrt(n);
    while ((s >= d) && (r != 0)) {
        if (2*r-rp+q<0) {
15            t = r;
            r = 2*r-rp+q+d+2;
            rp = t;
            q = q+4;
            d = d+2;
20        } else if ((2*r-rp+q>=0) && (2*r-rp+q<d+2)) {
            t = r;
            r = 2*r-rp+q;
            rp = t;
            d = d+2;
25        } else if ((2*r-rp+q>=0) && (2*r-rp+q>=d+2)) {
            t = r;
            r = 2*r-rp+q+d+2;
            rp = t;
            q = q+4;
            d = d+2;
        }
    }
}
```

```

                                &&(2*r-rp+q<2*d+4)) {
    t = r;
    r = 2*r-rp+q-d-2;
    rp = t;
    q = q-4;
    d = d+2;
} else /* ((2*r-rp+q>=0)&&(2*r-rp+q>=2*d+4)) */
{
    t = r;
    r = 2*r-rp+q-2*d-4;
    rp = t;
    q = q-8;
    d = d+2;
}
}
//return d;
}

```

Auch in diesem Fall ist es mit *Polyinvar* weder möglich, die Invariante zu verifizieren, noch sie herzuleiten. Im Falle dieses Beispiels geriet *Polyinvar* zudem zu einem Zustand, in dem die Iteration abgebrochen wurde, weil eine Berechnung zu lange dauert. Vermutlich führen die große Zahl von beteiligten Variablen zusammen mit der hohen Zahl der nicht unterstützten Verzweigungswächter zu einem sehr stark aufgeächerten Graphen, der zu einer sehr komplexen Fixpunktiteration führt. Scheinbar führt diese zu einer Konfiguration der Module und Vektoren, die nicht sehr performant berechenbar ist. An dieser Stelle muss daher leider noch auf eine Inferenz von Invarianten verzichtet werden. Jedoch weist die mißlungene Verifikation darauf hin, dass auch die Inferenz nicht zur gewünschten Invariante geführt hätte.

3.3.5 Zusammenfassung

Die Ergebnisse mit denjenigen Testprogrammen, die von beiden Analysewerkzeugen verarbeitet werden können, werden in Tabelle 3.8 dargestellt. Generell läßt sich sagen, dass die behandelten Beispiele nicht das gesamte Leistungsspektrum von *Polyinvar* ausleuchten, da dieses auch mit Zuweisungen und Invarianten von wesentlich höherem Grad umgehen kann. Da jedoch keine öffentlich zugängliche Version der anderen Analysesoftware existiert, konnte im Rahmen dieser Arbeit kein ausgeprägter Vergleich durchgeführt werden. Leider geht daraus hervor, dass *Polyinvar* in der hier beschriebenen Version noch nicht leistungsfähig genug ist, um alle Fälle von polynomiellen Invarianten erkennen zu können. Allerdings sind noch einige Möglichkeiten für Verbesserungen in Abschnitt 5.2 offen, und für die Fälle, in denen *Polyinvar* akzeptable Ergebnisse liefert, tut es das in relativ kurzer Zeit.

Programm	Verifikation	Inferenz
lcm	1,45s	3,5s / Grad 2
prod	*1,35s	* 3,0s / Grad 3
cubes	1,3s	2,6s / Grad 3
factor	*1,5s	* ... / Grad 1

Mit * markierte Berechnungen führten nicht zum gewünschten Ergebnis

Tabelle 3.8: Vergleichswerte

4 Dokumentation

Die Dokumentation von *Polyinvar* soll einerseits für die Benutzung des Pakets als auch für den weiteren Ausbau der Software dienlich sein.

4.1 Anwenderdokumentation

Dieser Abschnitt ist für die Anleitung zur reinen Benutzung von *Polyinvar* gedacht, zum Beispiel zum Zwecke der Lehre oder der Forschung.

4.1.1 Voraussetzungen

Die Mindestvoraussetzung für die Benutzung von *Polyinvar* ist ein Computer, mit einem lauffähigen Betriebssystem, entwickelt wurde *Polyinvar* auf einem Linuxsystem. Des Weiteren wird von folgenden installierten Programmen ausgegangen:

- *Java* JRE Version 1.4.2
- *Singular* Version 2.0.5
- *Eclipse* Version 3.0
- *Graphviz* Version 0.96c (für die Nutzung der GUI)
- *Java* SDK Version 1.4.2 (für die Umwandlung)
- *Ant* Version 1.5.4 (für die Umwandlung)

Programme wie *Singular* oder *Graphviz* sollten dabei so installiert worden sein, dass sie als Befehl auf der Kommandozeile eingetippt werden können. Für manche Systeme wie *MS Windows* ist dafür noch eine weitere Einrichtung der Umgebungsvariablen, die den Ausführungspfad bestimmen notwendig.

4.1.2 Installation

Die Installation von *Polyinvar* wird in zwei Phasen durchgeführt. In der ersten Phase muss der Kontrollflußgenerator als Plugin in *Eclipse* eingebaut werden. Dazu muss der Komplette Inhalt des Verzeichnisses `plugins` in das Verzeichnis `plugins` des Stammeclipseverzeichnisses kopiert werden.

Die Installation des Analysewerkzeugs gestaltet sich sehr einfach; es ist keine weitere Installation notwendig, als die zur Ausführung benötigten `jar`-Dateien in ein beliebiges Verzeichnis zu kopieren. Die entsprechenden Programmkomponenten können direkt daraus gestartet werden.

4.1.3 Benutzung

Die Verwendung von *Polyinvar* ist durch zwei Arbeitsschritte gekennzeichnet. Im ersten Schritt muss *Eclipse* gestartet werden, für welches im Rahmen dieses Programmpakets ein Plugin geschrieben wurde. Dieses Plugin erscheint als Schaltfläche in der Schaltflächenleiste des Navigators von *Eclipse*, wie in Abbildung 4.1 zu sehen ist.

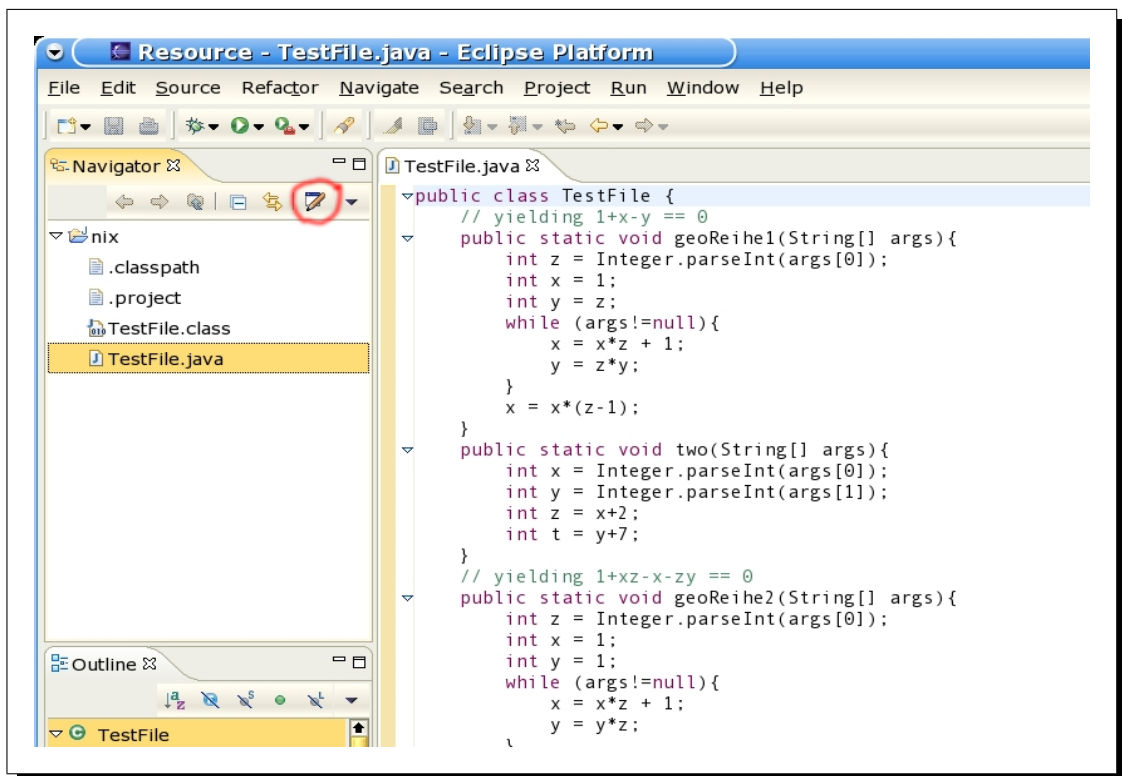


Abbildung 4.1: Eclipse mit CFGGenerator

Bei der Betätigung der Schaltfläche wird die markierte Klasse komplett in eine Menge von Kontrollflußgraphen umgewandelt. Im danach erscheinenden Dialog können diese Kontrollflußgraphen zur Weiterverarbeitung abgespeichert werden.

Geöffnet werden können diese Dateien dann im zweiten Schritt mit dem eigentlichen Analysewerkzeug. Angeboten werden zwei fertige Analysewerkzeuge, ein Kommandozeilenwerkzeug und ein grafisches Programm.

Kommandozeilenwerkzeug

Das Kommandozeilenwerkzeug dient primär zur Verwendung im Batchbetrieb, es ermöglicht eine Ansteuerung des Algorithmus über die Befehlszeile, so wie in Abbildung 4.2 dargestellt.

```
usage: java -jar polyengine.jar
  -d,--debug          debug [LEVEL], from FINEST to SEVERE
  -f,--file           Inputfile [FILE]
  -g,--gui            GUI start
  s -h,--help         this help screen
  -m,--method         analysed [METHOD]
  -p,--polydegree    detect polynomials up to degree [N]
  -v,--verifypoly    verify [POLYNOMIAL]
```

Abbildung 4.2: Kommandozeilenoptionen für Polyinvar

Mit Hilfe dieser Parameter kann man also sehr genau steuern, wie sich *Polyinvar* verhalten soll. Unter anderem ist es möglich, durch das Setzen von Debugleveln sehr genau mitzuverfolgen, welche Schritte der Algorithmus durchführt.

Eine Anfrage mit dem Parameter `-p` führt zu einer Inferenzanalyse, basierend auf Modulen und Vektoren, während `-v` eine Verifikationsanalyse mit Idealen und Polynomen auslöst.

Über die Kommandozeile können allerdings nur Analysen, die sich auf das gesamte Programm beziehen, durchgeführt werden. Eine Invariante bezieht sich damit also immer auf das gesamte Programm. Für detailliertere Interaktion mit dem Algorithmus ist man daher auf die grafische Schnittstelle angewiesen.

Grafische Animation

Zur Demonstration der Funktionsweise des Analysealgorithmus, oder für eine komplexere Interaktion mit dem Analysewerkzeug dient die grafische Animation, die in der ausführbaren jar-Datei `gui.jar` bereitgestellt wird. Es existieren hier Dialoge zum Öffnen und laden spezifischer Methoden aus der erzeugten Kontrollflußgraphensammlung. Nach dem Laden der entsprechenden Methoden und der Auswahl des Analysemodus, können nun einzelne Programmpunkte angewählt werden, um Invariantenkandidaten für die Analyse im Verifikationsmodus entgegenzunehmen. Diese führen zu einem Anwachsen der Abarbeitungsliste, die im linken oberen Fensterteil wie in 4.3 dargestellt wird. Diese Liste kann nun durch drücken auf die `next` Schaltfläche Schritt für Schritt abgebaut werden.

Wenn die Liste komplett abgebaut ist, kann am ersten Programmpunkt mit Hilfe des Steuerungsteils nachgeprüft werden, ob die zugehörige Basis leer ist, oder ob sie tatsächlich aus Generatoren besteht.

Für eine Analyse im Inferenzmodus kann auch über die Selektion einzelner Programmpunkte ein generisches Polynom erzeugt werden, welches dann über die Iteration Informationen an

alle anderen Programmpunkte verteilt. Sobald die Liste wieder leer ist, kann auch hier der erste Programmpunkt angewählt werden, damit eine Extraktion der gültigen Invarianten möglich ist.

4.2 Entwicklung

Als Entwickler bietet *Polyinvar* vielfältige Ansatzmöglichkeiten für Erweiterungen oder Adaptionen. Zum Beispiel wäre es möglich, einen Parser für Kontrollflußgraphen für eine andere Sprache zu schreiben, auch unabhängig von *Eclipse*. Eine andere Möglichkeit ist die Abstützung der Berechnungen auf eine andere Mathematikbibliothek, in der Hoffnung auf eine effizientere Implementierung von Idealen oder Modulen über Polynome.

Nicht zuletzt ist auch die Architektur des gesamten Werkzeugs interessant, vor dem Hintergrund der Adaption des vorhandenen Codes auf andere Fixpunktanalysen. Auch die grafische Animation des Kontrollflußgraphen oder die Generierungsfunktion von Kontrollflußgraphen aus *Eclipse* heraus könnten für andere Analysatoren nützlich sein.

4.2.1 Eingabemodul

In der vorliegenden Fassung existiert für *Polyinvar* nur eine einzige Möglichkeit, Kontrollflußgraphen aus fertigen Programmen zu generieren. In diesem Fall nimmt *Eclipse Polyinvar* einen Großteil der Arbeit ab, indem der in *Polyinvar* bereitgestellte `CFGGenerator` den über die *JDT* API bereitgestellten abstrakten Syntaxbaum direkt von *Eclipse* entgegen, und wandelt diesen in den gewünschten Kontrollflußgraphen um.

Nachdem dieser `CFGGenerator` vom eigentlichen Analysewerkzeug entkoppelt ist, kann seine Aufgabe sehr leicht von einem anderen Tool übernommen werden. Wichtig ist nur, dass dieses andere Werkzeug dann einen Kontrollflußgraphen unter Zuhilfenahme der selben Grundbausteinklassen konstruiert. Die erzeugten Kontrollflußgraphen müssen dann nur noch unter ihrem Namen in einer Hashtabelle abgelegt und als Objektstrom in eine Datei abgelegt werden. Die Definition der Graph-Grundbausteine ist im Paket `cfgstructure.jar` enthalten, und kann jederzeit von anderen Java-Programmen benutzt werden.

Denkbar wäre zum Beispiel ein eigenständiger Java-Parser, der auch ohne *Eclipse* lauffähig wäre. Auch die Einbettung von anderen Sprachen wäre möglich.

4.2.2 Rechenengine

Ein weiteres Modul, welches gegen ein eigenes ausgetauscht werden kann ist das Mathematikpaket. Die Anbindung von *Polyinvar* an seine Mathematikbibliothek geschieht auf eine generische Weise, indem der Fixpunktalgorithmus nur auf Interfaces arbeitet. Diese Interfaces werden über eine abstrakte Fabrikenklasse erzeugt und für die Iteration verwendet.

In der vorliegenden Fassung stützt sich *Polyinvar* auf *Singular* ab, für das eine solche Fabrikenklasse und entsprechende Stellvertreterklassen selbst geschrieben worden sind. Intern

geben diese Klassen über Ein-/Ausgabeströme die Aufgaben an die darunterliegende *Singular* Instanz weiter.

Solange die vorgegebenen Schnittstellen eingehalten werden, kann *Polyinvar* somit zur Zusammenarbeit mit einem beliebigen Mathematikpaket gebracht werden.

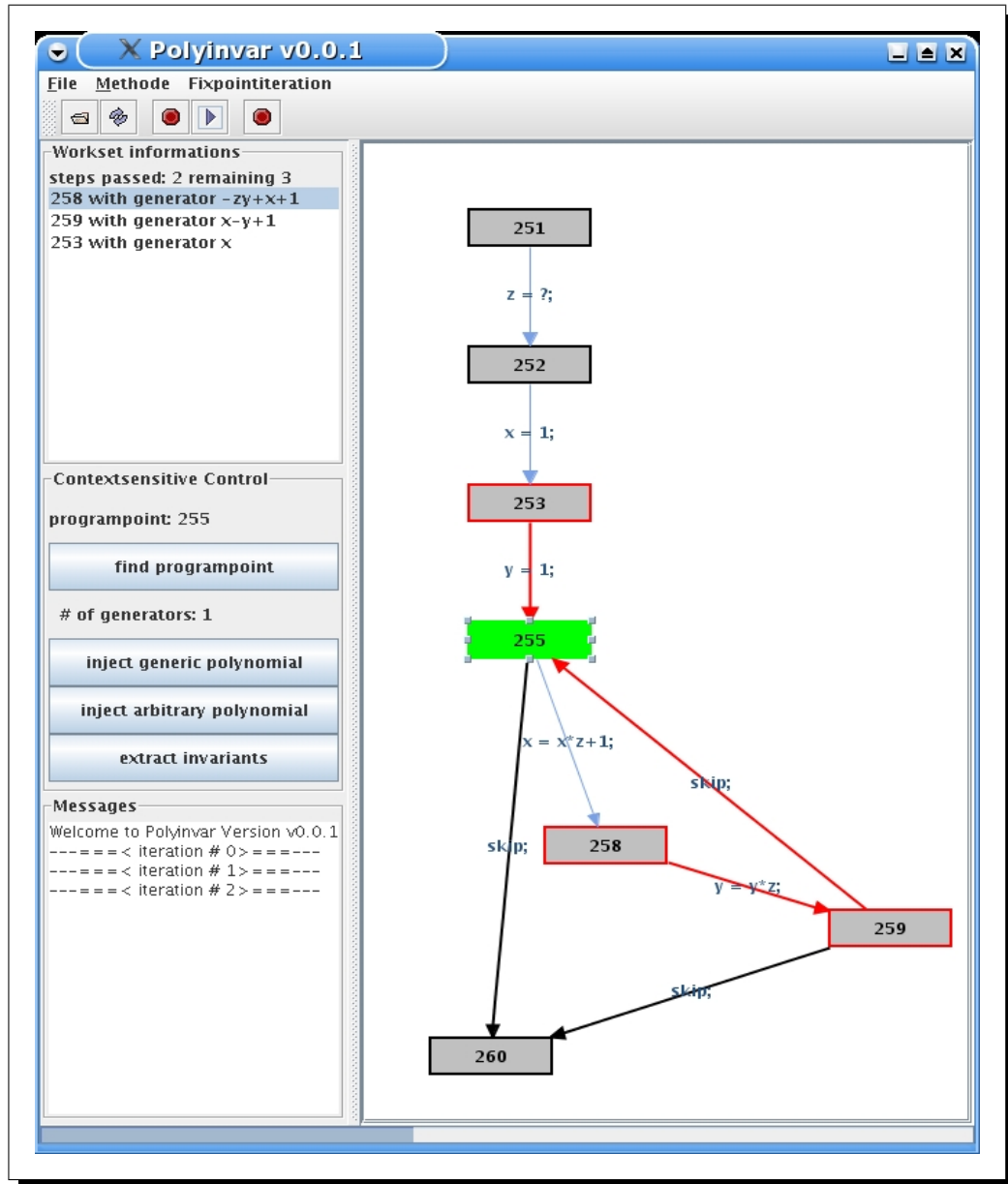


Abbildung 4.3: Polyinvar - GUI

5 Ergebnisse

5.1 Fazit

Mit *Polyinvar* liegt eine Werkzeugsammlung vor, mit deren Hilfe Methoden aus Java-Klassen in Kontrollflußgraphen umgewandelt werden können. Diese Kontrollflußgraphen können anschließend auf spezielle polynomielle Invarianten oder generische Polynome bis zu einem beschränkten Grad hin untersucht werden.

Die Laufzeit des Werkzeugs ist dabei doch zumindest bei den betrachteten Laborbeispielen so kurz, dass die Verifikation von Invarianten jedesmal funktionierte und die Herleitung von Invarianten in den meisten Fällen zu akzeptablen Ergebnissen führte. Leider sind nicht alle Beispiele, die mit Konkurrenzprodukten analysierbar zu sein scheinen auch vollständig mit *Polyinvar* benutzbar, was sich jedoch durch die anschliessend beschriebenen Erweiterungsmöglichkeiten ändern könnte.

5.2 Erweiterungsmöglichkeiten

Die momentan existierende Version von *Polyinvar* arbeitet für Laborbeispiele zwar schon ganz gut, jedoch ist ein weiterer Ausbau der Leistungsmerkmale für den praktischen Einsatz unabdingbar. Auch eine Verbesserung des Laufzeitverhaltens ist denkbar, aber nicht ohne weitere Modifikation des Werkzeugs erreichbar.

5.2.1 Sprachumfang

In der momentanen Fassung von *Polyinvar* ist die Unterstützung von grundlegenden Javaanweisungen implementiert, so daß vollständige Programme geschrieben und analysiert werden können. Jedoch wurden an manchen Stellen aus Mangel an Entwicklungszeit spezielle Sprachkonstrukte nicht unterstützt. In zukünftigen Versionen von *Polyinvar* könnten diese Mängel Schritt für Schritt ausgeglichen werden.

Vererbte Objekt- und Klassenvariablen sind momentan noch nicht mit in die Analyse eingebunden. Jede Zuweisung, die eine Objektvariable enthält, wird momentan als nichtdeterministische Zuweisung behandelt.

Für eine genauere Analyse mit Unterstützung von objektorientierten Eigenheiten wäre zuerst eine Aufrufgraphenanalyse notwendig, damit überhaupt ein korrekter Kontrollflußgraph generiert werden kann.

Arithmetische Ausdrücke

In der Standardversion von *Polyinvar* werden arithmetische Ausdrücke, in denen Divisionen und Modulooperationen vorkommen generell als nichtdeterministische Zuweisungen behandelt. Diese war notwendig, um die Entscheidbarkeit des Algorithmus abzusichern, wie in [?] beschrieben. Jedoch wäre es denkbar, zumindest für arithmetische Ausdrücke, die nur konstante Zahlen beinhalten, Divisionen und Modulooperationen zuzulassen. Für die allgemeinere Version, der Zulassung von reinkonstanten Divisoren wäre eine Überarbeitung der allgemeinsten Vorbedingung notwendig.

Numerische Identifikatoren

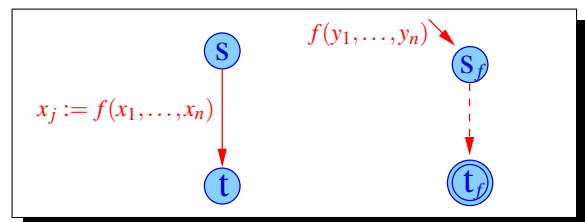
Momentan arbeitet *Polyinvar* so, dass Identifikatoren aus dem analysierten Programm direkt als Identifikatoren des darunterliegenden Mathematikpakets übernommen werden. Das reicht für die Laborfälle vollkommen aus, gestaltet die Bedienung über die Kommandozeile sogar sehr komfortabel aber schließt einen Einsatz in der Praxis aus, da nicht garantiert werden kann, dass Identifikatoren nicht in geschachtelten Blöcken wiederverwendet werden.

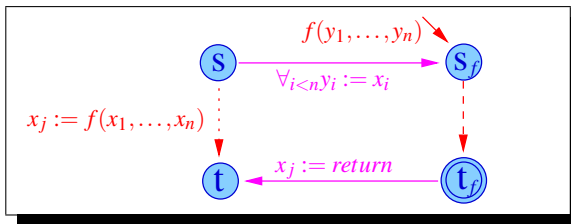
Als Lösung dafür bietet sich an, für das Mathematikpaket abstrakte numerische Identifikatoren einzubauen, die an der Benutzeroberfläche dann in sprechendere Namen zurückübersetzt werden müssen. Dies setzt eine weitere Entwicklung der grafischen Oberfläche und der Benutzerinteraktionsmöglichkeiten voraus.

Methodenaufrufe

In der derzeitigen Fassung von *Polyinvar* werden Methodenaufrufe gehandhabt, indem ihr Effekt als nichtdeterministische Zuweisung behandelt wird. Diese Vorgehensweise zieht potentiell einen Informationsverlust nach sich, der bei Kenntnis der aufgerufenen Methode nicht sein muss. Daher wäre eine genauere Analyse an dieser Stelle wünschenswert.

Für die Analyse von Methodenaufrufen bieten sich mehrere Verfahren an. Zum Einen wäre eine an den Call-String Ansatz aus [?] angelehnte Methode denkbar. Konkret müßten dabei an jeder Stelle, an der ein Funktionsaufruf $x_j = f(x_1, \dots, x_n)$ erfolgt, spezielle *enter* und *combine* Sequenzen eingefügt werden. In diesen Sequenzen müßten die Variablenspeicherungen zur Parameter- und Rückgabewertübergabe erfolgen.





Der Nachteil dieser Methode ist sicherlich, dass mit jedem Methodenaufruf auch mehr Verbindungskanten zum Methoden - Kontrollflußgraphen geschaffen werden. Das führt zu einem Verlust der Genauigkeit, bei einer steigender Zahl von Methodenaufrufen der selben

Methode, wobei aber trotzdem noch die Hoffnung besteht, mehr herauszufinden, als bei der Behandlung als nichtdeterministische Zuweisung.

Zum Anderen wäre es noch möglich eine Art von Heuristik für die Analyse von Funktionsaufrufen zu verwenden. Zum Beispiel könnte man annehmen, dass eine Methode effektiv nur maximal r verschiedene Rückgabewerte y_r liefert. Unter dieser Annahme formuliert man die Invariante p_r :

$$p_r = (ret - y_1) \cdot \dots \cdot (ret - y_r)$$

Nun rechnet man wie gewohnt die allgemeinste Vorbedingung für die aufgerufene Methode f aus. Als Resultat erhält man die Bedingungen t_i , die eintreffen müssen, damit einer der r Rückgabewerte berechnet wird.

$$\llbracket f \rrbracket p_r = \langle t_1, \dots, t_s \rangle$$

Für diese Polynome t_i berechnet man im zweiten Schritt, wann die Annahme der maximal r Rückgabewerte falsch war. Dazu stellt man zuerst ein Sammelpolynom q auf

$$q(t_1, \dots, t_s) = (t - t_1) \cdot \dots \cdot (t - t_s)$$

für welches man dann wieder die Vorbedingung unter der Methode f berechnet:

$$\llbracket f \rrbracket q = \langle q_1, \dots, q_l \rangle$$

Diese Polynome q_i geben nun an, welche Bedingungen gelten müssen, wenn das Ergebnis der Methode f nicht unter den r verschiedenen angenommen ist. In diesem Fall könnte man das Ergebnis nur noch durch eine nichtdeterministische Zuweisung symbolisieren.

Zusammenfassend gesagt bedeutet das, dass man, sobald man die entsprechenden Vorbedingungen t_i und q_i für ein festes r vorberechnet hat, jeden Methodenaufruf ersetzen kann durch eine Kombination einer nichtdeterministischen Zuweisung mit einer Menge an Wächtern und einer Menge an polynomiellen Zuweisungen, wie in Abbildung 5.1 dargestellt.

Fremde Objektvariablen

Für Zugriffe auf fremde Objektvariablen kann man sich darauf einschränken, nur Zugriffe über `set` und `get` Methoden zuzulassen, alle anderen Zugriffe werden als nichtdeterministische Zuweisungen interpretiert. Wenn nun solche `sets` oder `gets` ausser auf lokale oder formale Parameter auch auf Objektvariablen des jeweiligen Objekts zugreifen, hat man mehrere Möglichkeiten, dies zu behandeln.

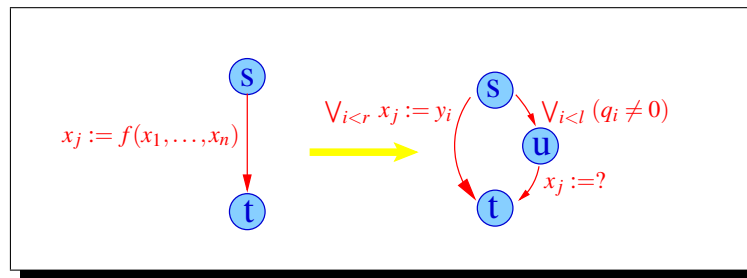


Abbildung 5.1: Behandlung eines Methodenaufrufs

Die allgemeinere Variante geht davon aus, dass man für alle Objekte einer Klasse dieselbe Variable zur Analyse verwendet. Dies führt also dazu, dass bei jedem Zugriff auf Objektmethoden einer Klasse für eine Klasse immer dieselbe Methode analysiert wird.

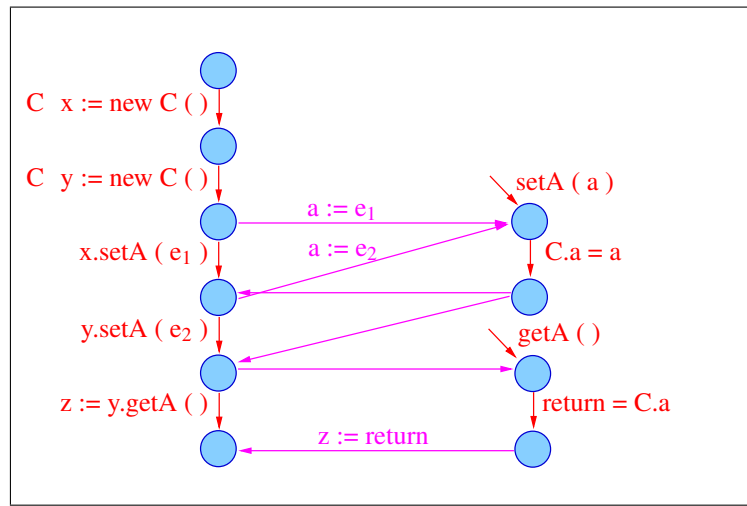


Abbildung 5.2: Behandlung eines Fremdoobjektzugriffs

Bei der Analyse von Methoden nach dem Call-String Ansatz wie in Abschnitt 5.2.1 beschrieben ergibt sich dadurch der Effekt, dass sobald Schreibzugriffe auf mehrere Objekte einer Klasse stattfinden, die Analyse unscharf wird. Die dabei entstehende Situation ist in Abbildung 5.2 illustriert. Dieser Schärfeverlust könnte zum Beispiel dadurch kompensiert werden, dass man nach der zweiten beschriebenen Variante der Methodenanalyse vorgeht. Durch den Parameter r kann man dabei steuern, wieviele verschiedene Werte get jeweils liefern kann.

Eine zweite Möglichkeit zur genaueren Analyse wäre zum Beispiel die Vereinbarung, nicht die Objekte aller Klassen gleich zu behandeln, sondern nur die Objekte, die an der selben Stelle im Kontrollflußgraphen, also durch das selbe `new` entstanden sein können, gemeinsam zu verwalten. Dadurch erhöht sich allerdings die Größe des Kontrollflußgraphen verglichen

mit der vorigen Lösung, da nun spezifisch für jede Objektinstanziierung eigene `get` und `set` Methoden kreiert werden müssen.

5.2.2 Zusicherungen und Abfragen

Als sinnvolle Erweiterung zur bisherigen Vorgehensweise bietet sich an, eine Möglichkeit zur Deklaration von Zusicherungen und Abfragepunkten in das Analysewerkzeug zu integrieren. Dabei gibt es zwei Realisierungsmöglichkeiten.

Zum Einen wäre der Einbau einer Interaktionsmöglichkeit in das Animationsprogramm denkbar, mit deren Hilfe der Benutzer Abfragepunkte markieren und Zusicherungen zuordnen könnte.

Die andere Version bestünde in der Erweiterung des unterstützten Sprachumfanges um die Möglichkeit, innerhalb von Kommentaren Abfragepunkte markieren zu können, oder Behauptungen postulieren zu können. Ein denkbare Beispiel könnte aussehen wie in Abbildung 5.3 Auch das Analysewerkzeug von Rodríguez-Carbonell [?] unterstützt eine ähnliche Vorgehens-

```

public class Test {
    public static int test(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
5       int z, t;
        // !assert! (t) AND (z)
        z=x+2;
        t=z+7;
        // !query! (x-z+2) AND (z-7-t)
10      }
    }

```

Abbildung 5.3: Beispiel für Eigenschaften in Kommentaren

weise. Im Unterschied zu *Polyinvar* jedoch läuft das Werkzeug auf einer eigenen Sprache, in der Behauptungen und Abfragen als Syntaxbestandteile enthalten sind.

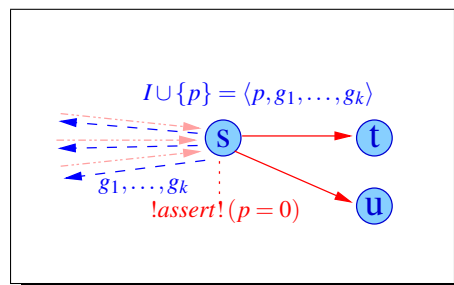


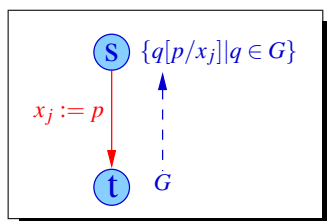
Abbildung 5.4: Behandlung einer Zusicherung

Eine mögliche Realisierung von Zusicherungen wäre, den Kontrollflußgraphen an der entsprechenden Stelle mit einer Vorinitialisierung zu versehen. So würde eine Zusicherung so funktionieren, dass das entsprechende Polynom zum Ideal oder Modul des Programmpunktes hinzugefügt wird. Diese Hinzufügung darf allerdings nicht zu einer weiteren Propagation des entsprechenden neuen Generators führen, wie in Abbildung 5.4 gezeigt wird. Daher ist diese Form der Zusicherungsbehandlung auch nur mit dem inkrementellen Fixpunktalgorithmus verwendbar.

5.2.3 Zeitliche Begrenzung und Widening

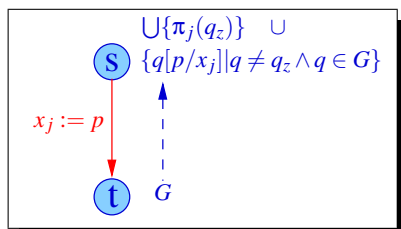
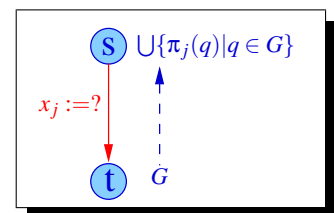
Ein weiterer Ansatz für die Laufzeitoptimierung stellt einen Kompromiß zwischen Genauigkeit und Berechenbarkeit dar. Nachdem die Berechnung der Reduktion durch Generatormengen von Idealen und Modulen sehr aufwendig sein, wie die obere Schranke aus [?] zeigt, und einen Großteil der Rechenzeit in Anspruch nehmen kann, sollten an dieser Stelle angesetzte Modifikationen auch unter ungünstigen Umständen für eine akzeptable Lösung in akzeptabler Zeit sorgen.

Zum Beispiel wäre denkbar, Reduktionen mit einem Zeitlimit zu versehen, so dass zu aufwendige Reduktionen nach dem Aufbrauchen des zugeteilten Zeitkontingents einfach abbrechen. Als Reaktion auf diesen Abbruch sollte die aufwendige Reduktion in eine oder auch mehrere weniger aufwendige Reduktionen umgewandelt werden.



Nehmen wir an, wir betrachten eine Zuweisungskante $x_j = p$. Es ist denkbar, dass ein Generatorpolynom q_z der Generatormenge G_t durch die vorgenommene Substitution $q'_z = q_z[p/x_j]$ sehr stark aufgebläht wird und nicht mehr in beschränkter Zeit festgestellt werden kann, ob q'_z durch die anderen Generatoren von G_s auf Null reduziert werden kann.

Eine abgeschwächte, aber unter Umständen immer noch ausreichend genaue Näherung an das eigentlich gewollte Ergebnis besteht darin, die entsprechende Kante so zu behandeln, als ob es sich um eine nichtdeterministische Zuweisung handeln würde. Der Nachteil an dieser Lösung ist, dass sowohl die Generatormenge G an Programmpunkt s wieder komplett neu berechnet werden muss, als auch alle Programmpunkte, die auf dem Weg vom Start bis s liegen.



Denkbar wäre außerdem eine Kombination beider Vorgehensweisen, bei denen lediglich das Generatorpolynom q_z , welches nicht in ansprechender Zeit reduziert werden kann, in seine x_j Vorfaktoren aufgeteilt wird. Das entspricht der Propagation einer allgemeineren Information als die der Propagation des gesamten Polynoms. Dadurch bleibt das Ergebnis korrekt, wird aber unter Umständen etwas ungenauer.

Diese Methode hätte den Vorteil, dass sich die Änderung für die Fixpunktiteration nur lokal

auf den Programmpunkt s auswirkt, und zudem nicht so viel Genauigkeit preisgibt wie die Behandlung der Kante als nichtdeterministische Zuweisung. Zudem ist diese Methode sehr gut kompatibel mit dem Prinzip der inkrementellen Fixpunktiteration.

Abbildungsverzeichnis

2.1	Fixpunktalgorithmus	12
2.2	Optimierte Generatorentwicklung	13
2.3	Exponenten für Monome erzeugen	15
2.4	Polynomielle Invarianten herleiten	16
2.5	Fixpunktalgorithmus mit Modulen und Vektoren	17
2.6	Algorithmus zur Herleitung von Invarianten	19
3.1	Prinzipielle Funktionsweise	22
3.2	Detaillierte Funktionsweise	23
3.3	Einfluß von Grad und Anzahl der Variablen	31
4.1	Eclipse mit CFGGenerator	38
4.2	Kommandozeilenoptionen für Polyinvar	39
4.3	Polyinvar - GUI	42
5.1	Behandlung eines Methodenaufrufs	46
5.2	Behandlung eines Fremdobjektzugriffs	46
5.3	Beispiel für Eigenschaften in Kommentaren	47
5.4	Behandlung einer Zusicherung	47

Tabellenverzeichnis

3.1	Zuordnung Kanten - Klassen	24
3.2	Versuchsumgebung	25
3.3	Geometrische Reihen	26
3.4	Potenzsummen	27
3.5	Einfluß der Variablenzahl	29
3.6	Laufzeit abhängig vom Grad der Analyse	30
3.7	Effekt von verschiedenen Strategien zur Generatorenerweiterung	32
3.8	Vergleichswerte	35

Literaturverzeichnis

- [BWa] Thomas Becker and Volker Weispfenning. In [?], Kapitel 10.4, pages 485–488.
- [BWb] Thomas Becker and Volker Weispfenning. In [?], Appendix, pages 511–514.
- [May97] Ernst W. Mayr. Some complexity results for polynomial ideals. *Journal of complexity*, 13(3):305–325, 1997.
- [MOS04] Markus Müller-Olm and Helmut Seidl. Computing Polynomial Program Invariants. *Information Processing Letters (IPL)*, 91(5):233–244, 2004.
- [RCK04] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. *International Symposium on Symbolic and Algebraic Computation 2004 (ISSAC'04)*, 2004.
- [SP] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In [?], Kapitel 7, pages 189–233.