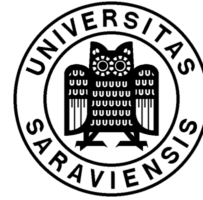
Universität des Saarlandes
Fachbereich 6.2 Informatik
Lehrstuhl für Programmiersprachen und Übersetzerbau




Diplomarbeit zur Erlangung des akademischen Grades
Diplom-Informatiker

# Statische Berechnung sicherer Schleifengrenzen auf Maschinencode

**Christoph Cullmann**

cullmann@babylon2k.de

27. März 2006

| | | |
|---|---|---|
| **Erstprüfer** | : | Prof. Dr. Reinhard Wilhelm |
| **Zweitprüfer** | : | Prof. Dr. Bernd Finkbeiner |
| **Betreuer** | : | Dr. Florian Martin |

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Saarbrücken, den 27. März 2006

# Danksagung

Ich bedanke mich bei Prof. Dr. Reinhard Wilhelm und Dr. Florian Martin für das interessante Thema und die eingehende Betreuung während der Durchführung meiner Arbeit. Prof. Dr. Bernd Finkbeiner gilt mein Dank für seine Bereitschaft, die Aufgabe des Zweitprüfers zu übernehmen.

Desweiteren bedanke ich mich bei allen Mitarbeitern der AbsInt Angewandte Informatik GmbH, besonders Michael Schmidt und Henrik Theiling, die mir hilfreich bei der Einarbeitung in das Framework von AbsInt zur Seite standen und mir über zahlreiche Anfangsschwierigkeiten hinweghalfen. Steffen Wiegratz danke ich für die Bereitstellung zahlreicher Testbeispiele für die Schleifenanalyse, welche bei der Evaluierung meiner Arbeit sehr hilfreich waren. Weitere Beispiele wurden unter anderem auch von Christian Hümbert beigesteuert. Dr. Reinhold Heckmann danke ich für das sehr hilfreiche Korrekturlesen.

Meinen Eltern und meiner Freundin Vanessa danke ich für die Unterstützung während meiner Studienzeit. Außerdem gilt mein Dank meinen beiden Mitstudenten Gernot Gebhard und Ingmar Stein für die gute Zusammenarbeit im Studium und die vielen hilfreichen Diskussionen im Verlauf dieser Arbeit.

# Inhaltsverzeichnis

**1 Einleitung** **1**

**2 Abstrakte Interpretation** **3**
2.1 Motivation . . . 3
2.2 Mathematische Grundlagen . . . 4
2.2.1 Verbandstheorie . . . 4
2.2.2 Galoistheorie . . . 7
2.2.3 Fixpunktiteration . . . 7
2.3 Abstrakte Interpretation . . . 10
2.4 Datenflussanalyse . . . 11
2.4.1 Einführung . . . 11
2.4.2 Formale Grundlagen . . . 14
2.4.3 Interprozedurale Analyse . . . 17

**3 PAG - Program Analyzer Generator** **19**
3.1 Einführung . . . 19
3.2 PAG Spezifika . . . 19
3.2.1 PAG Supergraph . . . 20
3.2.2 Schleifentransformation . . . 22
3.2.3 VIVU (Virtual Inlining & Virtual Unrolling) . . . 23
3.3 Generierung von Analysatoren . . . 23
3.3.1 Überblick . . . 24
3.3.2 DATLA . . . 24
3.3.3 FULA . . . 24

**4 Design der Schleifenanalyse** **25**
4.1 Ziele der Analyse . . . 25
4.1.1 Konkrete Zielsetzung . . . 25
4.1.2 Wichtigste Begrifflichkeiten . . . 27
4.2 Ausgangssituation . . . 28
4.2.1 Architektur von aiT . . . 28
4.2.2 Eingabeformat (CRL) . . . 30
4.2.3 Werteanalysator (daan) . . . 31

4.2.4 Bisherige Schleifenanalyse . . . 33
4.3 Getroffene Designentscheidungen . . . 33
4.3.1 Konzentration auf Zählschleifen . . . 33
4.3.2 Kooperation mit Werteanalyse . . . 34
4.3.3 Analyse geschachtelter Schleifen . . . 34
4.3.4 Parallele Analyse von mehreren Schleifen . . . 35

**5 Phasen der Analyse 37**
5.1 Überblick . . . 37
5.2 Klassifizierung der Schleifen . . . 38
5.2.1 Information aus der Werteanalyse . . . 38
5.2.2 Unerreichbare Schleifen . . . 38
5.2.3 Endlosschleifen . . . 39
5.2.4 Schleifen mit oberer Schranke . . . 40
5.2.5 Sonstige Schleifen . . . 40
5.3 Erkennung der Schleifenzähler . . . 41
5.3.1 Grundlegende Definitionen . . . 41
5.3.2 Information aus der Werteanalyse/CRL . . . 42
5.3.3 Erkennungsverfahren . . . 42
5.4 Berechnung der Schleifeninvarianten . . . 44
5.4.1 Ziel der Invariantenanalyse . . . 44
5.4.2 Verwendete Gleichungssysteme . . . 45
5.5 Auswertung der Schleifentests . . . 47
5.5.1 Extraktion der Gleichungssysteme . . . 47
5.5.2 Erstellung einer geschlossenen (Un-)Gleichung . . . 51
5.5.3 Lösen der (Un-)Gleichung . . . 53
5.5.4 Kombination der Ergebnisse . . . 57

**6 Invariantenanalyse 59**
6.1 Einführung & Ziele . . . 59
6.2 Abstrakter Verband . . . 60
6.3 Transferfunktion . . . 60
6.3.1 Call-Kanten . . . 60
6.3.2 Return-Kanten . . . 61
6.3.3 Normale Kanten . . . 62
6.4 Kombinationsfunktion . . . 64
6.5 Widening . . . 66
6.6 Spezifikation für PAG . . . 68

**7 Implementierung 69**
7.1 Überblick . . . 69
7.2 Schnittstellen & Interaktion . . . 70
7.2.1 daan Schnittstelle . . . 70
7.2.2 libloopy Schnittstelle . . . 71

7.2.3 daan & libloopy Interaktion . . . . . 72
7.3 libloopy Implementierung . . . . . 73
7.3.1 Grundgerüst . . . . . 73
7.3.2 Architekturabhängige Komponenten . . . . . 74
7.4 PowerPC Architektur . . . . . 74
7.4.1 LoopyArch . . . . . 76
7.4.2 Invariantenanalyse . . . . . 77

**8 Theoretische Diskussion** **81**
8.1 Fähigkeiten und Beschränkungen . . . . . 81
8.2 Verwandte Arbeiten . . . . . 85
8.2.1 AbsInt aiT . . . . . 85
8.2.2 Tidorum Bound-T . . . . . 86
8.2.3 Sjödin, Healy und Whalley . . . . . 86
8.2.4 Sonstige Arbeiten . . . . . 86

**9 Praktische Evaluation** **89**
9.1 Evaluationsverfahren . . . . . 89
9.2 Gewählte Testbeispiele . . . . . 90
9.3 Messergebnisse . . . . . 90
9.3.1 Interne Testprogramme . . . . . 91
9.3.2 Reale Industrie-Programme . . . . . 94
9.4 Auswertung . . . . . 95

**10 Zusammenfassung & Ausblick** **97**

**A Spezifikation der Flussanalyse** **99**
A.1 DATLA . . . . . 99
A.2 FULA . . . . . 101

**Literaturverzeichnis** **117**

Kapitel

# 1 Einleitung

In unserer heutigen Zeit werden immer mehr zeit- und sicherheitskritische Aufgaben von eingebetteten Systemen übernommen. Dies beginnt bei der Steuerung in einem Airbag-Controller des eigenen Automobils bis hin zur Flugkontrolle von modernen Passagierflugzeugen. Daher wird es zunehmend bedeutend, sicher zu stellen, dass die zugrunde liegende Hardware und die jeweiligen Anwendungen gewisse Zeitschranken einhalten. So werden konservative, jedoch auch möglichst exakte obere Schranken für die Worst Case Execution Time (*WCET*) der verantwortlichen Programmroutinen benötigt.

Um diese Aufgabe zu bewältigen, gibt es bereits Anwendungen, z.B. *aiT* von der AbsInt Angewandte Informatik GmbH. Diese erlauben es, unter Verwendung des Maschinencodes eines zu untersuchenden Programms, eine sichere obere Schranke für seine WCET zu berechnen. Um dies zu ermöglichen, müssen zahlreiche Analysen miteinander kombiniert werden, beginnend mit einer Rekonstruktion des Kontrollflusses über Schleifen-/Werteanalyse bis hin zu einer Cache-/Pipelineanalyse und der Ermittlung des Ausführungspfads mit der größten Ausführungszeit.

Die Schleifenanalyse ist hierbei ein integraler Bestandteil. Sie bestimmt sichere Schranken für die Anzahl der Iterationen der in dem zu analysierenden Programm enthaltenen Schleifen. Nur mit möglichst exakten Schleifengrenzen ist eine weitere Berechnung der WCET überhaupt erst möglich. Diese Arbeit stellt nun eine neue, datenflussbasierte Analysemethode zur Bestimmung von sicheren Schleifengrenzen vor. Der Hauptaugenmerk liegt auf der Erkennung der Grenzen von additiven Zählschleifen, da diese den größten Anteil der Schleifen stellen, die für eingebettete Anwendungen von Relevanz sind. Es ist meines Wissens die erste Publikation eines praktisch anwendbaren Verfahrens zur Bestimmung von Schleifengrenzen auf Maschinencode-Ebene.

Über diese Anwendung zur Berechnung der WCET hinaus, kann die vorgestellte Analyse auch ein hilfreiches Werkzeug in anderen Bereichen darstellen. Zum einen kann sie als Fehlerüberprüfung in der Softwareentwicklung eingesetzt werden. Es können Programmierfehler entdeckt werden, falls die erkannten Iterationsgrenzen von der erwünschten Spezifikation abweichen. Desweiteren können die Analyseergebnisse

zur post-pass Optimierung eingesetzt werden. So kann man sie zum Beispiel für das Abrollen von Schleifen verwenden bzw. zu deren Parallelisierung.

## Gliederung der Arbeit

Zunächst werden die Grundlagen der entwickelten Schleifenanalyse vorgestellt. Kapitel 2 liefert das theoretische Rüstzeug für diese Arbeit. Es führt in die abstrakte Interpretation und ihre Anwendung in der Programmanalyse ein. Zusätzlich liefert es Basisinformationen über mathematischen Konzepte, die für das Verständnis der Diplomarbeit notwendig sind. In Kapitel 3 wird dann der Program Analyzer Generator (kurz *PAG*) vorgestellt, welcher zur Generierung der in der Arbeit verwendeten Datenflussanalysatoren benutzt wird. Dabei wird sowohl auf PAG-spezifische Eigenschaften der Analyse eingegangen als auch kurz beschrieben, aus welchen Eingaben PAG einen Analysator erzeugt.

Die folgenden Kapitel beschäftigen sich dann mit der vorzustellenden Schleifenanalyse. Zuerst wird in Kapitel 4 die Zielsetzung der entwickelten Analyse sowie deren grundlegender Entwurf beschrieben. Desweiteren wird auf das Zusammenspiel mit dem Framework, in das sie eingebettet ist, eingegangen. Danach erfolgt in Kapitel 5 eine detaillierte Erklärung des Ablaufes einer Iteration der Analyse, aufgeteilt in ihre einzelnen Phasen. In Kapitel 6 wird dann ausführlich die verwendete Datenflussanalyse beschrieben. Kapitel 7 schließt die Beschreibung des Analyseverfahrens mit einem Einblick in seine konkrete Implementierung ab. Als erste Architektur wird hierzu die PowerPC Prozessorarchitektur gewählt, die in der heutigen Zeit in zahlreichen eingebetteten Systemen in der Fahrzeug- und Flugzeugbranche eingesetzt wird.

Nach der Vorstellung der neuen Schleifenanalyse enthält Kapitel 8 eine theoretische Diskussion ihrer Fähigkeiten. Desweiteren erfolgt eine theoretische Gegenüberstellung mit der schon existierenden Schleifenanalyse von aiT und anderen publizierten Ansätzen zur Berechnung von Schleifengrenzen. Kapitel 9 liefert dann als Evaluation einen realen Vergleich der Analyseergebnisse der neu vorgestellten Analyse mit der schon im Einsatz befindlichen Analyse von aiT. Hierbei werden auch reale Programme aus der Industrie als Testbeispiele verwendet.

Kapitel 10 beschließt die Arbeit mit einer Zusammenfassung des Erreichten sowie einem kurzen Ausblick auf mögliche Verbesserungen und Erweiterungen.

Kapitel
# 2 Abstrakte Interpretation

Dieses Kapitel führt das Prinzip der *Abstrakten Interpretation* ein, das die Grundlage der in dieser Arbeit verwendeten Technologie der Datenflussanalyse mittels PAG ist. Im Rahmen dieser Einführung werden auch die mathematischen Konzepte vorgestellt, die innerhalb der *Abstrakten Interpretation* verwendet werden: die Grundprinzipien der Verbandstheorie, die Galoistheorie und die Fixpunktiteration.

Es wird kein Überblick über das gesamte Feld der *Abstrakten Interpretation* geliefert, sondern es werden nur die für diese Arbeit relevanten Aspekte angesprochen. Für ausführlichere Informationen zur *Abstrakten Interpretation* sind sowohl [NNH99] als auch [WM97] zu empfehlen.

## 2.1 Motivation

In der Praxis will man oft Aussagen über das Verhalten von Programmen treffen. Dies reicht von einer relativ einfachen Vorzeichenanalyse der vorkommenden Variablen bis zu der Bestimmung der WCET. Viele dieser Aussagen kann man auf der konkreten Semantik des Programms direkt nicht berechnen, da diese die praktisch berechenbare Komplexität übersteigen.

Hier kann man nun das Konzept der *Abstrakten Interpretation* anwenden. Anstatt direkt über der konkreten und meist komplexe Semantik des zu untersuchenden Programmes zu rechnen, überführt man diese in eine weniger komplexe abstrakte Semantik. Hierbei wählt man diese so, dass nur die für die Aussage relevanten Teile der konkreten Sematik erhalten bleiben. Dadurch verringert man die Komplexität und macht die Gültigkeit der Aussage praktisch berechenbar.

**Beispiel 2.1.1 (Bestimmung von Vorzeichen)**
Ein in der Praxis relevantes Problem ist die Bestimmung des Vorzeichens von mathematischen Ausdrücken über Ganzzahlen. Als Operationen sind Addition (+) und

Multiplikation ($*$) erlaubt. Ganzzahlen sind beliebig groß, es erfolgt daher keine Betrachtung von Überlaufen. Die Signum-Funktion $\sigma$ ist gegeben durch:

$$\sigma(x) = \begin{cases} pos & x > 0 \\ zero & x = 0 \\ neg & x < 0 \end{cases}$$

Um das Vorzeichen eines Ausdruckes zu bestimmen, reicht es auf der Menge der Vorzeichen zu rechnen, dabei kann einem Ausdruck auch ein unbekanntes Vorzeichen zugeordnet werden. Dieses ist mit ? bezeichnet. Die abstrakten Varianten der Addition ($\tilde{+}$) und Multiplikation ($\tilde{*}$) sind in der Tabelle 2.1 definiert.

| $\tilde{+}$ | *pos* | *zero* | *neg* | ? |
|---|---|---|---|---|
| *pos* | *pos* | *pos* | ? | ? |
| *zero* | *pos* | *zero* | *neg* | ? |
| *neg* | ? | *neg* | *neg* | ? |
| ? | ? | ? | ? | ? |

| $\tilde{*}$ | *pos* | *zero* | *neg* | ? |
|---|---|---|---|---|
| *pos* | *pos* | *zero* | *neg* | ? |
| *zero* | *zero* | *zero* | *zero* | *zero* |
| *neg* | *neg* | *zero* | *pos* | ? |
| ? | ? | *zero* | ? | ? |

Tabelle 2.1: Definition von $\tilde{+}$ und $\tilde{*}$

Will man jetzt z.B. allgemein das Vorzeichen des Ausdrucks $z * z$ bestimmen, reicht es dessen Abstraktion zu betrachten, die nach Definition der abstrakten Multiplikation liefert, dass der Ausdruck entweder ein positives Vorzeichen hat oder Null ist.

Statt auf der konkreten Wertemenge der ganzen Zahlen zu arbeiten operiert man über der Menge der abstrakten Zustände $\{neg, zero, pos, ?\}$.

## 2.2 Mathematische Grundlagen

Um ein Verständnis der *Abstrakten Interpretation* zu erreichen, müssen deren mathematische Grundlagen bekannt sein. Daher werden nun die mathematischen Konzepte der Verbandstheorie, Galoistheorie und Fixpunktiteration eingeführt.

### 2.2.1 Verbandstheorie

Die Verbandstheorie stellt eine essentielle Grundlage für die *Abstrakte Interpretation* dar. Dieser Abschnitt liefert eine kurze Einführung in die wichtigsten für das weitere Verständnis nötigen Konzepte. Beispiele und Beweise zu den hier aufgeführten Definitionen und Sätzen sind in Kapitel 3 aus [Mar95] zu finden.

**Definition 2.2.1 (Partielle Ordnung)**
Sei $M$ eine Menge. Eine binäre Relation $\sqsubseteq \, \subseteq M \times M$ heißt *partielle Ordnung* auf $M$, wenn gilt:

1. Reflexivität:

   $$\forall a \in M : a \sqsubseteq a$$

2. Transitivität:

   $$\forall a, b, c \in M : a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$$

3. Anti-Symmetrie:

   $$\forall a, b \in M : a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$$

Eine Menge $M$ zusammen mit einer partiellen Ordnung $\sqsubseteq$ bezeichnet man als eine *partiell geordnete Menge* $(M, \sqsubseteq)$. □

**Definition 2.2.2 (Obere/Untere Schranke)**
Sei $(M, \sqsubseteq)$ eine partiell geordnete Menge und $A \subseteq M$. Ein $a \in M$ heißt eine *obere Schranke* von $A$, wenn gilt:

$$\forall b \in A : b \sqsubseteq a$$

$a$ heißt *kleinste obere Schranke* von $A$ ($\bigsqcup A$), falls:

1. $a$ obere Schranke von $A$ ist, also gilt: $\forall b \in A : b \sqsubseteq a$
2. für alle oberen Schranken $c$ von $A$ gilt: $a \sqsubseteq c$

$\bigsqcup$ bezeichnet man als *Vereinigung*, die Vereinigung von zwei Elementen $a$ und $b$ schreibt man als $a \sqcup b$.

Die *untere Schranke* bzw. *größte untere Schranke* von $A$ ($\bigsqcap A$) wird analog definiert. □

**Definition 2.2.3 ($\omega$-Kette)**
Sei $(M, \sqsubseteq)$ eine partiell geordnete Menge. Eine *$\omega$-Kette*[1] einer partiellen Ordnung ist eine aufsteigende Kette aus Elementen $a_0, a_1, a_2, \ldots$ von $M$ für die gilt:

$$a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots \sqsubseteq a_i \sqsubseteq \cdots$$

Gilt für eine $\omega$-Kette zusätzlich:

$$a_0 \sqsubset a_1 \sqsubset a_2 \sqsubset \cdots \sqsubset a_i \sqsubset \cdots$$

so wird diese auch als *streng aufsteigende $\omega$-Kette* bezeichnet. □

**Definition 2.2.4 (Vollständige partielle Ordnung)**
Sei $(M, \sqsubseteq)$ eine partiell geordnete Menge. Liegt für jede $\omega$-Kette $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots \sqsubseteq a_i \sqsubseteq \cdots$ die kleinste obere Schranke der Menge $\{a_i \mid i \in \omega\}$ in $M$, so heißt $(M, \sqsubseteq)$ *vollständige partielle Ordnung* (VPO). □

[1] Die geordnete Menge $(\mathbb{N}, \leq)$ bezeichnet man mit $\omega$

**Definition 2.2.5 (Aufsteigende Kettenbedingung)**
Eine partiell geordnete Menge $(M, \sqsubseteq)$ erfüllt die *aufsteigende Kettenbedingung*, wenn jede aufsteigende $\omega$-Kette endlich ist. □

**Definition 2.2.6 (Vollständiger Verband)**
Eine partiell geordnete Menge $(M, \sqsubseteq)$ heißt *vollständiger Verband*, wenn jede Teilmenge von $M$ eine kleinste obere und größte untere Schranke besitzt. Einen Verband schreibt man als Sechstupel $(M, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$, wobei $\bot = \bigsqcap M$ das kleinste und $\top = \bigsqcup M$ das größte Element ist. □

**Beispiel 2.2.1 (Vollständiger Verband)**
Betrachtet man die Menge der abstrakten Werte aus Beispiel 2.1.1 auf Seite 3 erweitert um ein kleinstes Element $\bot$, so stellt diese einen vollständigen Verband dar. Dieser ist mit seiner Ordnung in Abbildung 2.1 dargestellt.

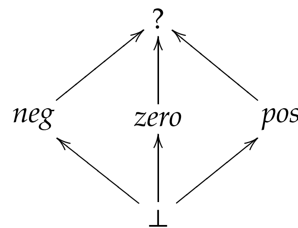



Abbildung 2.1: Vollständiger Verband der abstrakten Werte

**Definition 2.2.7 (Dualer Verband)**
Sei $V = (M, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ ein vollständiger Verband. $V_d = (M, \top, \bot, \sqsupseteq, \sqcap, \sqcup)$ ist der zu $V$ *duale Verband*. □

**Definition 2.2.8 (Monotone Funktion)**
Eine Funktion $f : A \to B$ zwischen zwei partiell geordneten Mengen $(A, \sqsubseteq)$ und $(B, \sqsubseteq)$ heißt *monoton*, wenn

$$\forall a, b \in A : a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$$

erfüllt ist. □

**Definition 2.2.9 (Stetige Funktion)**
Eine Funktion $f : A \to B$ zwischen zwei vollständig partiell geordneten Mengen $(A, \sqsubseteq)$ und $(B, \sqsubseteq)$ heißt *stetig* genau dann, wenn für alle $\omega$-Ketten $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots \sqsubseteq a_i \sqsubseteq \cdots$ in $A$

$$\bigsqcup_{i \in \omega} f(a_i) = f(\bigsqcup_{i \in \omega} a_i)$$

gilt. □

Stetige Funktionen sind immer monoton.

### 2.2.2 Galoistheorie

Die Abstraktion vom Konkreten zum Abstrakten, die der *Abstrakten Interpretation* mit ihren Namen gibt, erfolgt mit Hilfe von Konzepten aus der Galoistheorie. Diese werden nun im Folgenden eingeführt, weiterführende Informationen und viele Beispiele liefert [NNH99].

**Definition 2.2.10 (Galois Verbindung)**
Seien die vollständigen Verbände $(L, \sqsubseteq)$ und $(M, \sqsubseteq)$ sowie die monotonen Funktionen $\alpha : L \to M$ und $\gamma : M \to L$ gegeben. Eine *Galois Verbindung* zwischen diesen beiden Verbänden ist ein Viertupel $(L, \alpha, \gamma, M)$ genau dann, wenn gilt:

- $\gamma \circ \alpha \sqsupseteq id_L$
- $\alpha \circ \gamma \sqsubseteq id_M$

Die Funktion $\alpha$ wird als *Abstraktionsfunktion*, die Funktion $\gamma$ als *Konkretisierungsfunktion* bezeichnet. □

In einer Galois Verbindung $(L, \alpha, \gamma, M)$ können mehrere Elemente in $M$ existieren, die eine Abstraktion des selben Elements aus $L$ darstellen, da die Abstraktionsfunktion $\alpha$ nicht injektiv sein muss. Somit erhält der Verband $M$ eventuell mehr Elemente als für die Abstraktion von $L$ nötig. Um dies zu vermeiden, wird folgende Variante der Galois Verbindung in der *Abstrakten Interpretation* verwendet.

**Definition 2.2.11 (Galois Einbettung)**
Sei eine Galois Verbindung $(L, \alpha, \gamma, M)$ gegeben. Diese bezeichnet man als *Galois Einbettung* genau dann, wenn gilt:

$$\alpha \circ \gamma = id_M$$ □

Somit verliert man innerhalb einer Galois Einbettung keine Präzision, wenn man zunächst konkretisiert und dann wieder abstrahiert.

**Beispiel 2.2.2 (Abstraktionsfunktion)**
In dem Beispiel 2.1.1 auf Seite 3 erfolgt eine Abstraktion von ganzen Zahlen zu der Menge der Vorzeichen. Somit kann man die Vorzeichenfunktion $\sigma$ als Abstraktionsfunktion verwenden.

### 2.2.3 Fixpunktiteration

In der *Abstrakten Interpretation* müssen oft rekursive Gleichungssysteme gelöst werden, deren rechte Seite den zu definierenden Wert enthält. Dies ist zum Beispiel nötig, um die *Abstrakte Interpretation* zu verwenden und damit das Verhalten von Programmen zu analysieren, die Schleifen oder rekursive Prozeduren enthalten.

Einfache Beispiele für solche Gleichungssysteme sind die Fakultätsfunktion oder die Funktion zur Berechnung der Fibonacci-Zahlen:

$$fac(i) = \begin{cases} 1 & i = 0 \\ i * fac(i-1) & sonst \end{cases}$$

Jede Lösung muss diese rekursive Gleichung erfüllen. Lässt man als Lösung auch partielle Funktionen zu, erfüllt z.B. auch

$$f(i) = \begin{cases} \bot & i < 0 \\ i! & i \geq 0 \end{cases}$$

die Gleichung der Fakultätsfunktion.

Mittels eines einfachen iterativen Verfahrens kann man nun eine Funktion bestimmen, die auf einem endlichen Intervall $[0, \ldots, n]$ definiert ist. Hierzu beginnt man mit dem kleinsten Element des Lösungsraums $\bot$, der überall undefinierten Funktion und setzt diese auf der rechten Seite in die Definitionsgleichung ein. Dadurch erhält man eine Funktion, die für das triviale Intervall $[0, 0]$ definiert ist. Durch wiederholtes Einsetzen der jeweils erhaltenen Funktion kommt man so zu der gewünschten Funktion über $[0, \ldots, n]$. Um die gesuchte Funktion auf den natürlichen Zahlen zu finden, muss man den Grenzwert für $n \to \infty$ bilden.

**Definition 2.2.12 (Fixpunkt)**
Sei $f : M \to M$ eine Funktion. Ein Element $a \in M$ heißt *Fixpunkt* von $f$, wenn

$$f(a) = a$$

gilt. □

**Definition 2.2.13 (Präfixpunkt)**
Sei $f : M \to M$ eine Funktion. Ein Element $a \in M$ heißt *Präfixpunkt* von $f$, wenn

$$f(a) \sqsubseteq a$$

gilt. □

Folgender Satz formuliert mathematisch das intuitive obige Verfahren und garantiert eine Lösung.

**Satz 2.2.1 (Fixpunktiteration)**
Sei $(M, \sqsubseteq)$ eine vollständig partiell geordnete Menge mit dem kleinsten Element $\bot$ und $f : M \to M$ eine stetige Funktion. Man definiere nun $fix : (M \to M) \to M$ als:

$$fix(f) = \bigsqcup_{i \in \omega} f^i(\bot)$$

Dann ist $fix(f)$ ein Fixpunkt von $f$ und der kleinste Präfixpunkt von $f$, somit gilt:

1. $f(fix(f)) = fix(f)$
2. $\forall a \in M : f(a) \sqsubseteq a \Rightarrow fix(f) \sqsubseteq a$

**Satz 2.2.2 (Kleinster und größter Fixpunkt)**
Der kleinste Fixpunkt einer Funktion $f$ wird auch als $lfp(f)$ bezeichnet, der größte Fixpunkt als $gfp(f)$.

### Widening

Die Fixpunktiteration terminiert nur, wenn der Verband auf dem man operiert der aufsteigenden Kettenbedingung genügt. Oft will man allerdings auch eine Terminierung erreichen, falls die Kettenbedingung verletzt wird oder man will die Terminierung auf Kosten der Genauigkeit beschleunigen. Hierzu kann man einen Widening-Operator anwenden, wie in [CC77] und [CC92] beschrieben.

Das *Widening* erreicht die Terminierung, indem es die Ergebnisse der Iteration durch größere Verbandselemente sicher nach oben abschätzt. Die Kette der Abschätzungen muss so gewählt werden, dass sie endlich ist und die Iteration somit terminiert.

**Definition 2.2.14 (Widening)**
Sei $V$ der innerhalb einer Fixpunktiteration verwendete Verband. Ein *Widening-Operator* $\nabla \in V \times V \rightarrow V$ ist eine Abbildung, für die gilt:

$$\forall x, y \in V : x \sqsubseteq x \nabla y \wedge y \sqsubseteq x \nabla y$$

Für alle aufsteigenden Ketten $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n$ soll zusätzlich die Kette $y_0, y_1, \ldots y_n$ definiert durch

$$\begin{aligned} y_0 &= x_0 \\ &\vdots \\ y_{i+1} &= y_i \nabla x_i \end{aligned}$$

nicht streng aufsteigend sein. Die aufsteigende Iterationsfolge mit *Widening* definiert man als:

$$\begin{aligned} X^0 &= \bot \\ X^{i+1} &= \begin{cases} X^i & \text{falls } F(X^i) \sqsubseteq X^i \\ X^i \nabla F(X^i) & \text{sonst} \end{cases} \end{aligned}$$

Für vollständige, die aufsteigende Kettenbedingung erfüllende Verbände, ist die Operation zur Bildung der kleinsten oberen Schranke ein Widening. □

## 2.3 Abstrakte Interpretation

Nachdem nun die mathematischen Verfahren eingeführt sind, welche die Grundlage der *Abstrakten Interpretation* darstellen, wird nun das eigentliche Konzept der *Abstrakten Interpretation* definiert.

Hierzu benötigt man zunächst das Prinzip der *lokalen Konsistenz* von Funktionen.

**Definition 2.3.1 (Lokale Konsistenz)**
Sei eine Galois Einbettung $(L, \alpha, \gamma, M)$ gegeben. Eine konkrete Funktion $f : L \to L$ und eine abstrakte Funktion $f^\circ : M \to M$ heißen *lokal konsistent*, wenn gilt:

$$\forall x \in L : f(x) \sqsubseteq \gamma(f^\circ(\alpha(x)))$$ □

Abbildung 2.2 veranschaulicht diese Beziehung.

$$\begin{array}{ccc} \alpha(x) & \xrightarrow{f^\circ} & f^\circ(\alpha(x)) \\ \uparrow \alpha & & \sqsubseteq \downarrow \gamma \\ x & \xrightarrow{f} & f(x) \end{array}$$

Abbildung 2.2: Lokale Konsistenz

Damit ergibt sich für eine *Abstrakte Interpretation* folgende Definition.

**Definition 2.3.2 (Abstrakte Interpretation)**
Eine *Abstrakte Interpretation* besteht aus folgenden zwei Komponenten:

- einer Galois Einbettung $(L, \alpha, \gamma, M)$
- einem Paar von lokal konsistenten Funktionen $f : L \to L$ und $f^\circ : M \to M$

Will man nun Aussagen über die Funktion $f$ treffen, kann man hierzu auch ihre Abstraktion $f^\circ$ verwenden. Die Korrektheit wird durch die obigen beiden Bedingungen gewährleistet. □

Enthält die zu betrachtende Funktion $f$ bzw. $f^\circ$ eine Rekursion, so berechnet man mittels Fixpunktiteration ein Ergebnis. Hierbei ist folgende Beziehung nützlich.

**Satz 2.3.1 (Beziehungen der Fixpunkte)**
Sei eine *Abstrakte Interpretation* mittels der zugehörigen Galois Einbettung $(L, \alpha, \gamma, M)$ und den lokal konsistenten Funktionen $f$ und $f^\circ$ gegeben. Es ergeben sich folgende Beziehungen zwischen den Fixpunkten der beiden Funktionen:

- $lfp(f) \sqsubseteq \gamma(lfp(f^\circ))$
- $gfp(f) \sqsubseteq \gamma(gfp(f^\circ))$

## 2.4 Datenflussanalyse

Die *Datenflussanalyse* ist eine verbreitete Technik, um statische Laufzeitinformationen eines Programms zu berechnen. Sie wird in dieser Arbeit als wichtiges Hilfsmittel zur Berechnung von Schleifengrenzen eingesetzt. Sie kann als eine Anwendung der *Abstrakten Interpretation* auf die Programmanalyse aufgefasst werden. Dieser Abschnitt liefert eine kurze Beschreibung der für diese Arbeit relevanten Aspekte dieses Themengebiets. Für weitere Informationen und Beispiele empfehlen sich [NNH99], [Mar95] und [Sch05].

### 2.4.1 Einführung

Bei der Datenflussanalyse berechnet man zunächst den Kontrollflussgraph für das zu analysierende Programm. Dieser Graph ist eine Approximation aller zur Laufzeit möglichen Pfade des Programms. Bei der Berechnung des Graphen können Pfade eingeführt werden, die in der Realität nicht existieren. Da jedoch auf jeden Fall alle realen Ausführungspfade enthalten sind, ist diese Annäherunge konservativ und somit sicher.

Die eigentliche Analyse findet auf Basis dieses Graphen statt. Zunächst erhält jeder Knoten des Kontrollflussgraphen eine Funktion, die so genannte Transferfunktion des jeweiligen Knotens, die für den eingehenden abstrakten Datenflusswert einen ausgehenden Wert liefert. Kommen an einem Knoten mehrere Kanten an, werden deren Datenflusswerte mit einer Kombinationsfunktion zusammengefasst, dies kann je nach Problem z.B. die Vereinigung oder der Schnitt sein. Der so erhaltene Graph wird als Datenflussgraph bezeichnet. Er kann als ein rekursives Gleichungssystem aufgefasst werden, das die für die Analyse interessanten Teile der konkreten Semantik des Programms abstrahiert.

Für dieses System von Gleichungen kann mittels einer Fixpunktiteration eine Lösung bestimmt werden. Man erhält als Ergebnis abstrakte Datenflusswerte für die einzelnen Kanten.

**Beispiel 2.4.1 (Verfügbare Ausdrücke)**
Für ein Programm soll die Menge seiner verfügbaren arithmetischen Ausdrücke bestimmt werden. Das bedeutet, man ermittelt für jeden Programmpunkt eine Menge von arithmetischen Ausdrücken, die bereits berechnet wurden und deren enthaltene Variablen seit der Berechnung bis zu diesem Punkt nicht verändert wurden. Hierbei beachtet man nur nicht-triviale Ausdrücke.

Durch dieses Wissen kann man das jeweilige Programm optimieren, indem man die Werte der Ausdrücke in temporären Variablen zwischenspeichert und an den ermittelten Stellen wiederverwendet, was wiederholte Berechnungen erspart.

Für weitere Informationen ist Kapitel 2 aus [NNH99] zu empfehlen. Dort sind auch noch einige andere Beispiele für häufig genutzte Datenflussanalysen zu finden, z.B. zur Bestimmung der 'verfügbaren Definitionen' oder 'lebendigen Variablen'.

Das zu analysierende Programm wird in Listing 2.1 gezeigt. Die Syntax und Semantik der Befehle entsprechen der Programmiersprache *C*.

```
x = a * b;
y = a * x;
while ( x + y > a * b )
{
  a = a - 2;
  x = y + 4;
}
x = x * y;
```

Listing 2.1: Eingabeprogramm für Analyse

Um die Analyse durchzuführen, wird zunächst obiges Programm in seinen Kontrollflussgraph überführt. Dieser ist in Abbildung 2.3 dargestellt.

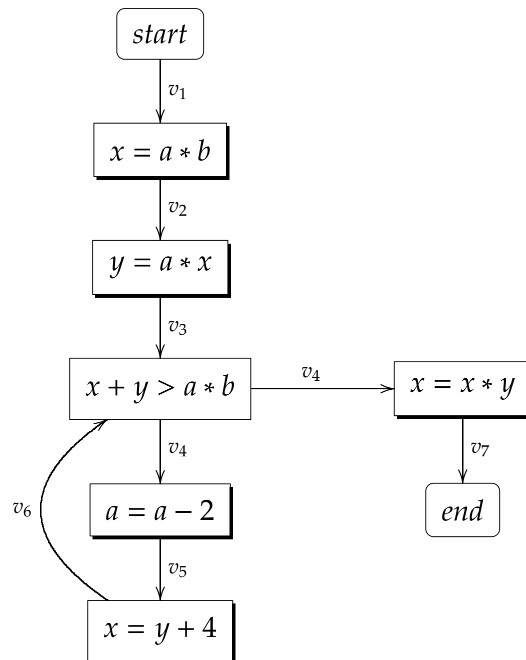



Abbildung 2.3: Kontrollflussgraph für das gewählte Beispiel

Nun wird dieser Graph zum Datenflussgraph erweitert. Die Knoten werden mit Funktionen annotiert, die auf dem Datenflusswert, einer Teilmenge der insgesamt

verfügbaren Teilausdrücke, operieren. Die Menge aller möglichen Teilausdrücke ist für unser Beispiel:

$$\{a * b, a * x, x + y, a - 2, y + 4, x * y\}$$

Diese Funktionen sind Abstraktionen der für diese Analyse wichtigen Aspekte der konkreten Semantik der dem jeweiligen Knoten zugeordneten Instruktion. Der dadurch erhaltene Graph ist in Abbildung 2.4 gezeigt. Die Syntaxfragmente in den Knoten wurden durch die erhaltenen Transferfunktionen ersetzt. Man besitzt nun eine Abstraktion des gesamten Programms in Form eines rekursiven Gleichungssystems, das nur noch auf der abstrakten Menge der möglichen Teilausdrücke operiert.

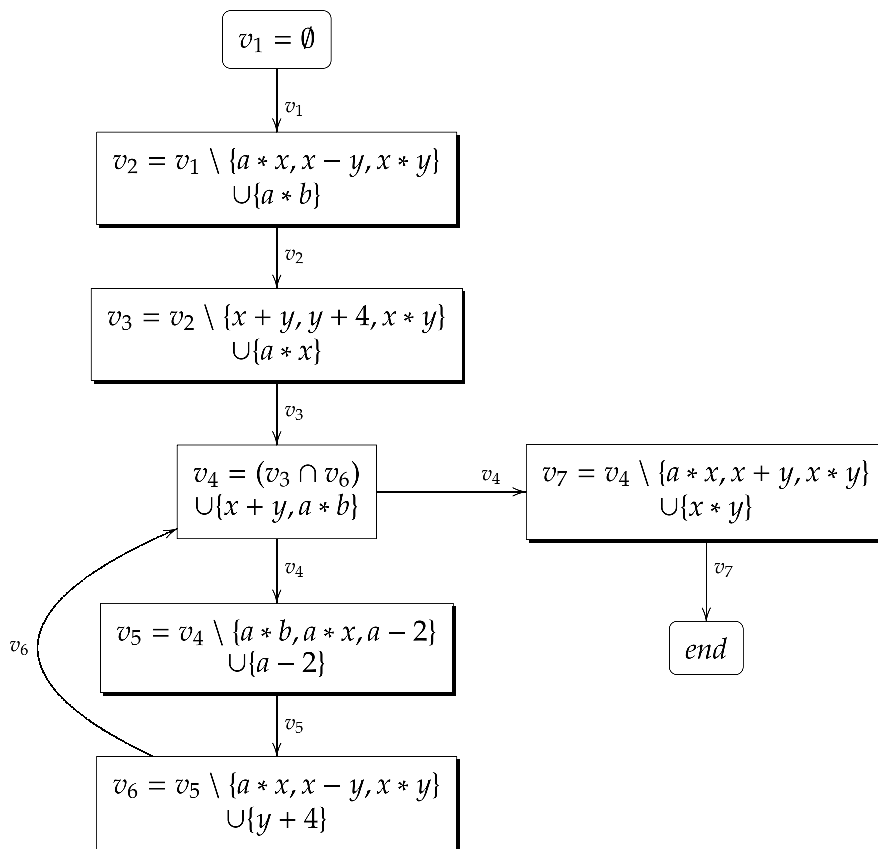



Abbildung 2.4: Datenflussgraph für das gewählte Beispiel

Dieses rekursive Gleichungssystem wird durch eine Fixpunktiteration gelöst. Als Ergebnis erhält man abstrakte Datenflusswerte für die einzelnen Kanten. In Abbildung 2.5 auf der nächsten Seite sind diese Werte an den Kontrollflussgraph annotiert.

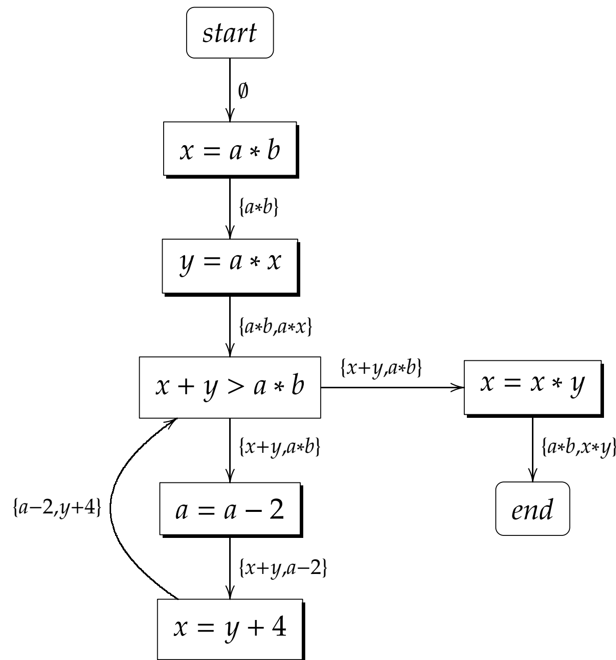



Abbildung 2.5: Kontrollflussgraph mit konkreten Datenflusswerten

### 2.4.2 Formale Grundlagen

Nach dieser Einführung in die Konzepte der Datenflussanalyse auf Basis der *Abstrakten Interpretation*, werden einige der grundlegenden Begriffe dieses Themengebietes definiert, die in dieser Arbeit ihre Anwendung finden.

**Definition 2.4.1 (Kontrollflussgraph)**
Ein *Kontrollflussgraph* (KFG) ist ein Viertupel $K = (N, E, s, e)$ mit einer Menge $N$ von Knoten, einer Menge $E \subseteq N \times N$ von gerichteten Kanten und jeweils einem eindeutigen Start- und Endknoten ($s$ bzw. $e$). Für den Startknoten gilt, dass er keine eingehenden Kanten besitzt, der Endknoten hingegen besitzt keine ausgehenden Kanten. Desweiteren muss eine Beschriftungsfunktion $F : N \to S$ existieren, die den Knoten Programmfragmente zuordnet. $S$ bezeichnet hierbei die Menge der Programmfragmente. Diese erhält man z.B. aus der Syntaxbaumdarstellung. □

Die Konstruktion von solchen Kontrollflussgraphen wird unter anderem in [All70] beschrieben. Die oben genannte Forderung nach eindeutigen Start- und Endknoten stellt keine Einschränkung der Allgemeinheit dar, denn jeder Graph lässt sich durch hinzufügen von maximal zwei zusätzlichen Knoten in diese Gestalt überführen.

**Definition 2.4.2 (Pfad)**
Sei $K = (N, E, s, e)$ ein Kontrollflussgraph. Eine Folge $\pi = n_1, n_2, \ldots, n_k$ von Knoten heißt ein *Pfad* von $n_1$ nach $n_k$ durch $K$, falls gilt: $\forall i \in [1, \ldots, k-1] : (n_i, n_{i+1}) \in E$. Mit $\epsilon$

wird der leere Pfad bezeichnet. $P[n_1, n_k]$ ist die Menge aller Pfade von $n_1$ nach $n_k$. Die Konkatenation zweier Pfade wird mit ';' bezeichnet. □

**Definition 2.4.3 (Datenflussproblem)**
Sei $V = (M, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ ein vollständiger Verband. $D = (K, V, \langle\rangle)$ heißt *Datenflussproblem* (DFP), wenn $K = (N, E, s, e)$ ein Kontrollflussgraph und $\langle\rangle : N \to (V \to V)$ eine Funktion ist, die den Knoten des Graphen Elemente aus $V \to V$ zuordnet. Diese werden als *Transferfunktionen* oder *Übergangsfunktionen* bezeichnet. □

**Definition 2.4.4 (Monotones/Distributives Datenflussproblem)**
Ein Datenflussproblem $D = (K, V, \langle\rangle)$ heißt *monoton/distributiv*, falls für alle Knoten $n$ von $K$ die Transferfunktion $\langle n \rangle$ monoton/distributiv ist. □

**Definition 2.4.5 (Pfadsemantik)**
Sei ein Datenflussproblem $D = (K, V, \langle\rangle)$ gegeben. Die Transferfunktion $\langle\rangle$ für Knoten $n$ des Kontrollflussgraphen läßt sich auf einen Pfad $\pi = n_1, \ldots, n_k \in P[n_1, n_k]$ wie folgt erweitern:

$$\langle \pi \rangle = \begin{cases} id & \text{falls } \pi = \epsilon \\ \langle (n_2, \ldots, n_k) \rangle \circ \langle n_1 \rangle & \text{sonst} \end{cases}$$

Monotonie und Distributivität gehen vom Datenflussproblem auf $\langle \pi \rangle$ über. □

Die gesuchte Lösung eines Datenflussproblems $D$ für einen Knoten $n$ des Kontrollflussgraphen ist die Vereinigung der Pfadsemantiken aller Pfade vom eindeutigen Startknoten $s$ nach $n$, angewandt auf den Startwert $\bot$.

**Definition 2.4.6 (MOP-Lösung)**
Sei ein Kontrollflussgraph $K = (N, E, s, e)$ und ein Datenflussproblem $D = (K, V, \langle\rangle)$ gegeben. Die *meet-over-all-paths Lösung* ist für alle $n \in N$ als

$$MOP(n) = \bigsqcup \{\langle \pi \rangle \mid \pi \in P[s, n]\}$$

definiert. □

Diese Definition kann meist nicht zur Berechnung einer Lösung verwendet werden, da die Menge $P[s, n]$ im Allgemeinen nicht endlich ist. Um eine Lösung zu bestimmen, muss eine algorithmische Lösungsstrategie gesucht werden, welche die Lösung approximiert. Eine solche Approximation heißt *korrekt*, falls sie größer oder gleich der gesuchten Lösung bezüglich der Ordnung $\sqsubseteq$ des jeweiligen Verbandes ist. Die MOP-Lösung ist die genauste aller Approximationen, daher wird sie auch *präzise* genannt. Eine solche algorithmische Lösungsstrategie ist die MFP-Lösung. Sie ist die Lösung, die man durch eine Anwendung der Fixpunktiteration erhält, wie sie bei der *Abstrakten Interpretation* beschrieben ist.

**Definition 2.4.7 (MFP-Lösung)**
Sei $K = (N, E, s, e)$ ein Kontrollflussgraph und $D = (K, V, \langle\rangle)$ ein Datenflussproblem. Als $pre(n)$ bezeichnet man die Menge der Informationen, die am Eingang eines Knoten $n \in N$ verfügbar sind, als $post(n)$ die Menge der Informationen, die am Ausgang eines Knoten gültig sind. Dies ist auch die gesuchte *minimal-fixpoint Lösung*. Es gelten hierbei folgende Berechnungsregeln:

$$pre(n) = \begin{cases} \bot & \text{falls } n = s \\ \bigsqcup\{post(m) \mid (m, n) \in E\} & \text{sonst} \end{cases}$$

$$post(n) = \langle n \rangle (pre(n))$$

Dies kann man umformulieren zu:

$$MFP(n) = \begin{cases} \langle s \rangle (\bot) & \text{falls } n = s \\ \langle n \rangle (\bigsqcup\{MFP(m) \mid (m, n) \in E\}) & \text{sonst} \end{cases}$$

Somit erhält man eine Berechnungsregel für die MFP-Lösung. □

In [KU77] beweist Kam, das die MFP-Lösung korrekt ist und für distributive Datenflussprobleme sogar der MOP-Lösung entspricht.

**Satz 2.4.1 (Korrektheit)**
Sei $K = (N, E, s, e)$ ein Kontrollflussgraph und $D = (K, V, \langle\rangle)$ ein monotones Datenflussproblem. Dann ist die MFP-Lösung eine korrekte Approximation der MOP-Lösung. Für alle $n \in N$ ist somit

$$MOP(n) \sqsubseteq MFP(n)$$

erfüllt.

**Satz 2.4.2 (Koinzidenz)**
Sei $K = (N, E, s, e)$ ein Kontrollflussgraph und $D = (K, V, \langle\rangle)$ ein distributives Datenflussproblem. Für alle $n \in N$ ist dann

$$MOP(n) = MFP(n)$$

erfüllt.

Die hier vorgestellten Definitionen und Sätze sind implizit von *Vorwärtsproblemen* ausgegangen, d.h. die Information an einem Knoten wird aus den Informationen seiner Vorgänger im Kontrollflussgraphen berechnet. Viele Analysen lassen sich jedoch nur als *Rückwärtsproblem* betrachten, so z.B. das Problem der 'lebendigen Variablen' (siehe [NNH99]). Diese lassen sich allerdings auf Vorwärtsprobleme über den invertierten Kontrollflussgraphen zurückführen, womit die hier angeführten Definitionen und Sätze wieder greifen. Der invertierte Graph $K^{-1}$ zu $K = (N, E, s, e)$ ist hierbei definiert als $K^{-1} = (N, E^{-1}, e, s)$.

Ebenfalls wurde implizit angenommen, dass die jeweiligen Probleme *Vereinigungsprobleme* seien, man also immer die Vereinigung der Informationen über alle Pfade berechnet. Jedoch existieren auch *Schnittprobleme*, die den Schnitt aller Informationen über alle Pfade berechnen. Schnittprobleme sind allerdings äquivalent zu Vereinigungsproblemen auf dem dualen Verband.

### 2.4.3 Interprozedurale Analyse

Bisher wurden nur *intraprozedurale* Datenflussanalysen behandelt. Es ist jedoch bei heutigen Programmiersprachen, seien es imperative wie C oder logische Sprachen wie Prolog, wichtig, Programme in ihrer Gesamtheit und somit über Prozedurgrenzen hinaus (*interprozedural*) zu analysieren. Problematische Punkte bei dieser Analyse sind rekursive Prozeduren sowie der modulare Aufbau moderner Programme.

Einige Methoden zur Bewältigung dieser Aufgabe werden in [Mar95] vorgestellt. Im Folgenden wird nur auf den *Call-String-Ansatz* eingegangen

#### Call-String-Ansatz

Der *Call-String-Ansatz* wurde von Sharir und Pnueli in [SP81] vorgestellt. Zunächst berechnet man hierzu einen einzigen zusammenhängenden Kontrollflussgraph, in dem Prozeduraufrufe und deren Rückkehr als einfache Übergänge betrachtet werden. Dabei entstehen unter Umständen Pfade, die in der eigentlichen Ausführung des Programms nicht vorkommen können, weil Aufrufstellen vermischt werden. Um die Datenflussinformationen nur entlang Pfaden zu propagieren, die Ausführungspfade darstellen, versieht man die Datenflussinformation mit einer kodierten Form ihrer Aufrufgeschichte. Diese Zusatzinformation bezeichnet man als *Call Strings*.

Solange keine rekursiven Prozeduren vorhanden sind, kann man die Vermischung von verschiedenen Aufrufstellen mit einem Call-String beschränkter Länge verhindern. Existieren jedoch rekursive Prozeduren, können die Call-Strings beliebig lang werden. Daher ist es nötig ihre maximale Länge zu beschränken, was zu einem Verlust an Information führen kann.

**Definition 2.4.8 (Supergraph)**
Sei $P$ ein Programm, das aus den Prozeduren $P_0, P_1, \ldots, P_n$ besteht. $P_0$ sei die Hauptprozedur des Programms. Zur Repräsentation dieses Programms dient ein gerichteter Graph $G^* = (N^*, E^*, s^*, e^*)$, der so genannte *Supergraph*. $G^*$ besteht hierbei aus einer Menge von intraprozeduralen Flussgraphen $G_0, G_1, \ldots, G_n$ der jeweiligen Prozeduren. Ein Prozeduraufruf wird hierbei durch einen *Call-*, einen *Local-* und einen *Return-Knoten* ersetzt.

Die einzelnen Flussgraphen sind durch Kanten an den eingefügten Knoten für die Aufrufe verbunden. Als neue Kanten kommen für einen Call-Knoten $c$, einen Local-Knoten $l$ und einen Return-Knoten $r$ eines Aufrufs einer Prozedur $P$ jeweils die folgenden hinzu:

- eine Aufrufkante von $c$ zum Startknoten von $P$
- eine Rückkehrkante vom Endknoten von $P$ zu $r$
- eine lokale Kante von $c$ nach $l$ und von $l$ nach $r$ □

Der lokale Knoten und die lokale Kante können verwendet werden, um Informationen zu propagieren, die nicht von dem Prozeduraufruf verändert werden. So kann man z.B. die Calling-Konvention einer Architektur abbilden, die sicherstellt, dass gewisse Register von der aufgerufenen Prozedur nicht verändert werden.

Die Abbildung 2.6 verdeutlicht die Transformation der Aufrufe anhand eines Beispiels.

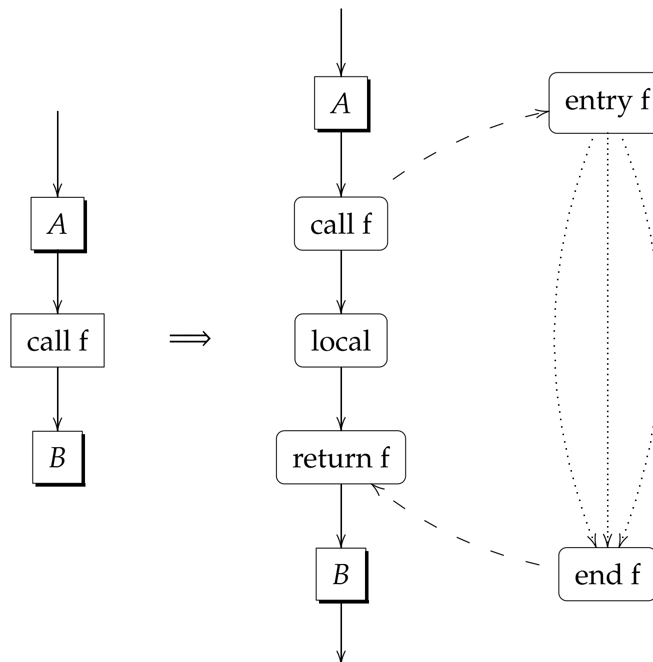



Abbildung 2.6: Transformation von Aufrufen in Supergraph

Der Supergraph kann nun wiederum als intraprozeduraler Kontrollflussgraph betrachtet werden und ermöglicht so die Anwendung der in diesem Abschnitt beschriebenen Datenflussanalysetechniken.

# Kapitel 3 PAG - Program Analyzer Generator

In diesem Kapitel wird auf den Program Analyzer Generator, kurz *PAG*, der AbsInt GmbH eingegangen. Er wird für die Erzeugung der in dieser Arbeit verwendeten Analysatoren benutzt. Daher ist zum Verständnis derselben Grundwissen über PAG und dessen Eigenschaften nötig.

## 3.1 Einführung

PAG entstand im Rahmen der Diplomarbeit von Florian Martin an der Universität des Saarlandes, siehe [Mar95]. Seit der ersten Implementierung wird PAG kontinuierlich weiterentwickelt. Einige Details hierzu kann man in [Mar99] finden. PAG wird in der AbsInt GmbH zur Erzeugung aller benötigten Datenflussanalysatoren verwendet und ist auf einer Vielzahl von Plattformen lauffähig, unter anderen auch Linux & Windows. Ähnlich wie bei dem Parsergenerator *yacc* wird aus einer Spezifikation eine C-Bibliothek erstellt, die eine Schnittstelle zum generierten Analysator liefert. Dieser Schnittstelle kann man dann einen Kontrollflussgraphen übergeben und die Analyse starten. Die Resultate sind nach Abschluss der Analyse ebenfalls über C-Funktionen zugänglich. Zur Spezifikation dienen zwei eigens entwickelte Sprachen, einmal DATLA, zur Beschreibung des zu verwendeten Verbands, und FULA, zur Spezifikation der Transferfunktion und anderer Aspekte des Datenflussproblems. Mehr Informationen zu Aufbau und Verwendung von PAG siehe [Abs05c]. Eine kurze Einführung liefert unter anderem [Mar98].

## 3.2 PAG Spezifika

In diesem Abschnitt werden nun einige Eigenschaften von PAG beschrieben, welche für die vorgestellte Schleifenanalyse von besonderer Bedeutung sind. Dies reicht von generellen Eigenschaften des Supergraphen in PAG bis zu speziellen Erweiterungen zur Analyse von geschachtelten Schleifen.

### 3.2.1 PAG Supergraph

PAG implementiert den in Abschnitt 2.4.3 auf Seite 17 vorgestellten *Call-String-Ansatz*. Hierzu wird der dort eingeführte *Supergraph* mittels einer *Grapherweiterung* zu einem *erweiterten Supergraph* ausgebaut.

**Definition 3.2.1 (Grapherweiterung)**
Sei $G^* = (N^*, E^*, s^*, e^*)$ ein Supergraph für ein Programm $P$ mit den Prozeduren $P_0, P_1, \ldots, P_n$. $arity : \{P_i \mid 0 \leq i \leq n\} \rightarrow [1..]$ sei eine Funktion, die für jede Prozedur die Stelligkeit der zugehörigen Datenflussinformation festlegt. Eine *Grapherweiterung* ist ein Tripel $(G^*, arity, map_c)$, wobei $map_c$ eine Familie von Funktionen ist, die für jede Aufrufstelle in $P$ die Tupelpositionen der aufrufenden Funktion denen der aufgerufenen Funktion zuordnen. Für einen Aufruf $c = \text{call } P_2$ in der Prozedur $P_1$ gilt also: $map_c : [1..arity(P_1)] \rightarrow [1..arity(P_2)]$. □

**Definition 3.2.2 (Erweiterter Supergraph)**
Sei $W = (G^*, arity, map_c)$ mit $G^* = (N^*, E^*, s^*, e^*)$ eine Grapherweiterung. Dann ist $G_E{}^* = (N_E{}^*, E_E{}^*, s_E{}^*, e_E{}^*)$ der zu $W$ gehörige *erweiterte Supergraph*, wenn gilt:

- $N_E{}^* = \{(n, i) \mid n \in N^* \wedge i \in [1..arity(n)]\}$
- $s_E{}^* = (s^*, 1)$
- $e_E{}^* = (e^*, 1)$
- $((n_1, i_1), (n_2, i_2)) \in E_E{}^*$ genau dann wenn $(n_1, n_2) \in E^*$ und eine der folgenden Bedingungen gilt:
  - $n_2$ ist ein Entry-Knoten und $i_2 = map_{n_1}(i_1)$
  - $n_1$ ist ein End-Knoten und $i_1 = map_c(i_2)$, wobei $c$ der zu $n_2$ gehörige Call-Knoten ist
  - $i_1 = i_2$ sonst □

Zur Veranschaulichung der beschriebenen Erweiterung liefert Abbildung 3.1 auf der nächsten Seite ein Beispiel für einen erweiterten Supergraph. Das zugrunde liegende Programm ist in Listing 3.1 auf der nächsten Seite angegeben. Es enthält eine Startprozedur *main*, die eine selbstrekursive Funktion $f$ aufruft. Der gezeigte Supergraph entsteht bei einer Call-String-Länge von 2.

Wird im Rahmen dieser Arbeit von einem *Supergraph* gesprochen, so ist der oben definierte *erweiterte Supergraph* gemeint.

**Definition 3.2.3 (Kontext)**
Sei $G_E{}^* = (N_E{}^*, E_E{}^*, s_E{}^*, e_E{}^*)$ ein Supergraph. Jeder Knoten von $G_E{}^*$ erhält durch die gewählte *Grapherweiterung* eine Menge von Datenflussinformationen. Jedes Element dieser Menge entspricht einem *Kontext* und ist einem eigenen *Call-String* zugehörig.□

```
int main (int argc, char *argv[])
{
  return f();
}

int f ()
{
  return f();
}
```

Listing 3.1: Code für Beispiel eines erweiterten Supergraphen

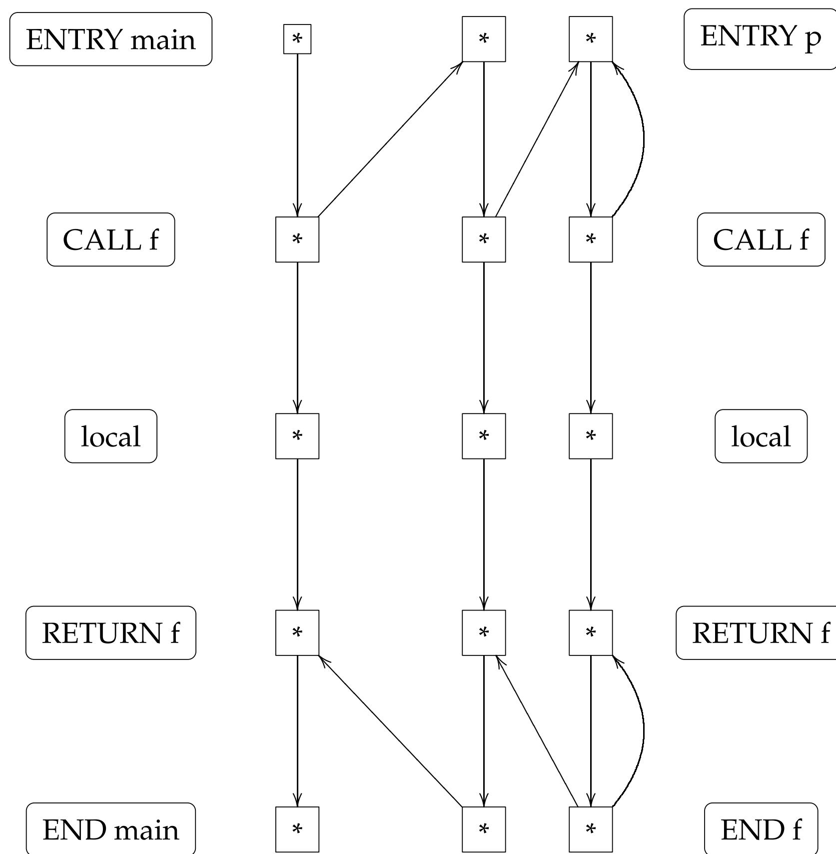



Abbildung 3.1: Erweiterter Supergraph

### 3.2.2 Schleifentransformation

Eine weitere Besonderheit von PAG ist sein Umgang mit Schleifen. Diese werden bei der Berechnung des Kontrollflussgraphen in rekursive Prozeduren umgewandelt. Durch diese so genannte *Schleifentransformation* kann man Techniken der interprozeduralen Datenflussanalyse auch auf Schleifen anwenden. Die in dieser Arbeit vorgestellte Schleifenanalyse arbeitet auf den so erhaltenen Schleifenprozeduren. Dies soll nun an Hand eines kleinen Beispiels verdeutlicht werden. Hierzu wird die genannte Transformation auf die Schleife aus Abbildung 3.2 angewendet. Das Ergebnis zeigt Abbildung 3.3. Dieses Verfahren wird unter anderem in [MAWF98] noch detaillierter erklärt.

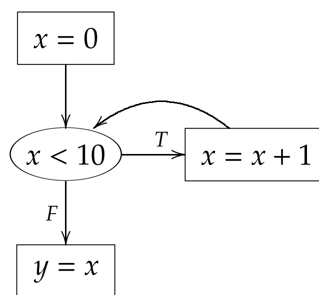



Abbildung 3.2: Einfache Schleife

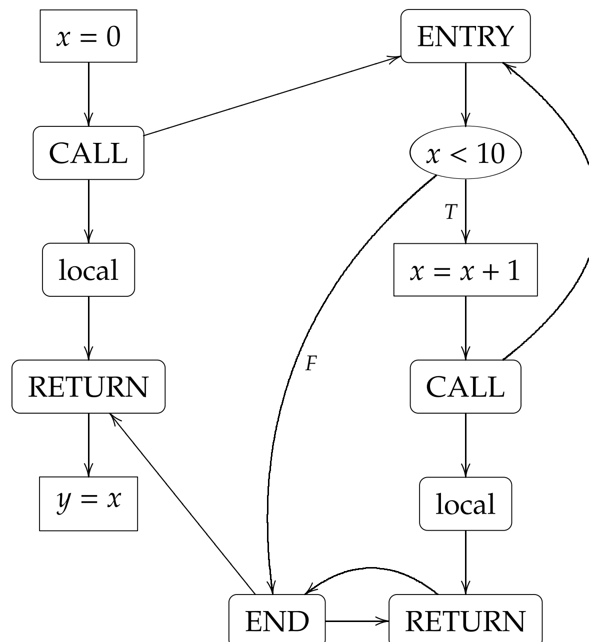



Abbildung 3.3: Einfache Schleife nach Transformation

### 3.2.3 VIVU (Virtual Inlining & Virtual Unrolling)

Da sich der *Call-String-Ansatz* in der Praxis als nicht ausreichend für eine möglichst exakte Analyse von geschachtelten Schleifen erwiesen hat, erweitert PAG diesen. Dabei kommt für solche Fälle eine spezielle Grapherweiterung bei der Erzeugung des erweiterten Supergraphen zum Einsatz, die so genannte *VIVU-Grapherweiterung*.

In der Praxis ist es hilfreich für die Präzision der WCET-Bestimmung, wenn man den ersten Schleifendurchlauf von den späteren getrennt betrachtet. Diesen Erfahrungen trägt die neue Methode Rechnung und ermöglicht eine Analyse nach diesen Gesichtspunkten.

Als Beispiel betrachte man nun ein Programm, das zwei geschachtelte Schleifen $l1$ und $l2$ enthält. Der erste Aufruf von $l1$ sei als $l1_f$ bezeichnet, jeder weitere als $l1_o$, die Aufrufe von $l2$ seien analog benannt. Würde das Programm mittels der Call-String-Methode betrachtet (Call-String-Länge sei 2), würden folgende Kontexte für die innere Schleife entstehen:

- $l1_f, l2_f$: erste Iteration der äußeren und inneren Schleife
- $l1_o, l2_f$: erste Iteration der inneren Schleife, $\geq$ 2te Iteration der Äußeren
- $l2_f, l2_o$: zweite Iteration der inneren Schleife
- $l2_o, l2_o$: $\geq$ 3te Iteration der inneren Schleife

Verwendet man hingegen das VIVU-Grapherweiterung (ebenfalls mit Länge 2), so erhält man folgende Kontexte:

- $l1_f, l2_f$: erste Iteration der äußeren und inneren Schleife
- $l1_f, l2_o$: erste Iteration der äußeren Schleife, $\geq$ 2te Iteration der Inneren
- $l1_o, l2_f$: $\geq$ 2te Iteration der äußeren Schleife, erste Iteration der Inneren
- $l1_o, l2_o$: $\geq$ 2te Iteration der äußeren Schleife, $\geq$ 2te Iteration der Inneren

Praktische Versuche haben die Verbesserung der Analyseergebnisse in dem für diese Arbeit interessanten Gebiet der WCET-Analyse bestätigt, wie z.B. [MAWF98] belegt. Deshalb findet diese Art der Grapherweiterung auch Anwendung in der Schleifenanalyse dieser Arbeit. Genauere Details über das VIVU-Grapherweiterung kann man in [Mar99] finden.

## 3.3 Generierung von Analysatoren

Nachdem nun die wichtigsten Besonderheiten von PAG in Bezug auf die Analyse erläutert wurden, wird nun ein kurzer Einblick in die Art und Weise, in der man Datenflussprobleme für PAG spezifiziert, gegeben.

### 3.3.1 Überblick

PAG stellt analog zu *yacc* ein Kommandozeilenprogramm bereit, das bei Angabe einer gültigen Spezifikation des Datenflussproblems einen Analysator für dasselbe generiert. Als Spezifikation benötigt PAG hierbei folgende drei Teile:

- Eine Spezifikation des zugrunde liegenden Verbandes in DATLA;
- Eine Spezifikation der Transferfunktion, Kombinationsfunktion und des zu verwendenden Widenings in FULA;
- Eine Auswahl der zu verwendenden Lösungsmethode.

Während PAG für die beiden ersten Teile Dateien in seinen eigenen Spezifikationssprachen erwartet, lässt sich der zu verwendende Algorithmus per Kommandozeilenschalter aus den Verfügbaren auswählen. Für eine genauere Beschreibung der möglichen Algorithmen sei auf [Abs05c] verwiesen. Im Folgenden sollen nur die für diese Arbeit relevanten beiden Spezifikationssprachen etwas näher betrachtet werden.

### 3.3.2 DATLA

*DATLA* (Datatype Definition Language) stellt die Mittel bereit, aus den gegebenen Grundtypen höhere Typen und Verbände zu konstruieren. Als Grundtypen stehen unter anderen vorzeichenbehaftete Ganzzahlen (*snum*) oder Zeichenketten (*str*) bereit. Man kann beliebig Tupel, Listen, Mengen und ähnliches konstruieren, sowie Aufzählungen erstellen. Neue Verbände können z.B. durch das Lifting schon existierender Verbände erzeugt werden. Detaillierte Angaben zu Syntax und Semantik von DATLA findet man in [Abs05c].

### 3.3.3 FULA

Die funktionale Sprache *FULA* dient der Spezifikation der Transferfunktion des Datenflussproblems. Zu den komfortablen Möglichkeiten der Sprache gehört zum Beispiel Patternmatching für Funktionsargumente und die einfache Einbindung von extern implementierten C-Funktionen. Auch hier ist für eine ausführlichere Beschreibung auf [Abs05c] zu verweisen.

Kapitel
# 4 Design der Schleifenanalyse

In diesem Kapitel wird nun das eigentlich Thema dieser Arbeit vorgestellt: die neue datenflussbasierte Analysemethode zur Bestimmung von Schleifengrenzen.

Zunächst werden die Ziele der Analyse angesprochen und diese dann zusammen mit einigen der wichtigsten verwendeten Begriffen explizit definiert.

Danach wird auf die Umgebung eingegangen, in welche die Analyse eingebettet wird, um die Verwendung ihrer Ergebnisse für die Berechnung der WCET von Maschinencode-Programmen nutzbar zu machen.

Zuletzt wird die grundsätzliche gewählte Vorgehensweise für die entwickelte Analyse vorgestellt. Die eigentliche Beschreibung des Ablaufs der einzelnen Phasen der Analyse erfolgt in Kapitel 5. Auf die zum Einsatz kommende Datenflussanalyse wird Kapitel 6 eingehen.

## 4.1 Ziele der Analyse

Nun werden detailliert die Ziele der angestrebten Analyse beschrieben, da diese eine erhebliche Auswirkung auf die Details der Wahl des Analyseverfahrens besitzen. Zunächst wird die Motivation zur Berechnung von Schleifengrenzen dargelegt. Danach werden einige formale Definitionen für schon länger informell verwendete Begriffe geliefert: was versteht man in dem Kontext dieser Analyse z.B. unter einer Schleifeniteration? Zuletzt werden einige der konkreten Ziele angesprochen, die mit der neuen Analyse erreicht werden sollen.

### 4.1.1 Konkrete Zielsetzung

Das Ziel der neuen Schleifenanalyse ist die Berechnung von sicheren Schranken für Zählschleifen. Die Schleifenanalyse arbeitet wie der Rest der *aiT-Architektur* direkt auf Maschinencode.

Da man an einer möglichst exakten WCET-Schranke interessiert ist, benötigt man nicht nur konservative, sondern auch möglichst genaue Schranken. Desweiteren will man zusätzlich zu den oberen Schranken auch untere Schranken bestimmen. Kann man die Durchlaufzahl auf ein exaktes Intervall einschränken, ermöglicht dies weitere Verbesserungen der Werteanalyse. So kann man z.B. die Adressen von Speicherzugriffen innerhalb einer Schleife genauer eingrenzen.

**Beispiel 4.1.1 (Mehr Präzision durch bekannte untere Schranke)**
Berechnet die Schleifeanalyse für das Programm aus Listing 4.1 neben der oberen auch die untere Schranke für die Iterationszahl der enthaltenen Schleife, so kennt man exakt die Adresse des darauf folgenden Speicherzugriffs, da der Wert von i bekannt ist. Dies ermöglicht eine exaktere Bestimmung der WCET.

```
int main (int argc, char *argv[])
{
  int buffer[1000];
  int i = 0;

  while (i < 16)
    ++i;

  return buffer[i];
}
```

Listing 4.1: Wichtige untere Schranke

Besonders die Interaktion mit dem Speicher hat eine wichtige Bedeutung für die WCET. Daher können die genaueren Adressen der Zugriffe zu erheblichen Verbesserungen in den Analyseergebnissen führen. Auch kann dies eine Analyse überhaupt erst ermöglichen, da inexakte Speicherzugriffe zu einem exponentiellen Anstieg des Speicherverbrauchs und der Analysedauer der Cache- und Pipelineanalyse führen können.

Neben diesem generellen Ziel verfolgt die neue Schleifenanalyse noch folgende zentrale Aspekte:

- *Portierbarkeit*
  Die zu erstellende Analyse soll möglichst einfach auf eine Vielzahl von Architekturen anwendbar und portierbar sein. Diesem Konzept wird vor allem in der Implementierung Sorge getragen, welche die architekturspezifischen Teile sorgfältig von den allgemeinen Bestandteilen trennt und Schnittstellen anbietet, diese zu kapseln. Mehr dazu in Kapitel 7.
- *Compiler-Unabhängigkeit*
  Die momentane Schleifenanalyse von *aiT* muss speziell auf jeden zu unterstützenden Compiler und eventuell sogar auf die verschiedenen Compilerversionen

angepasst werden. Ein Ziel der neuen Analyse ist es, diesen Anpassungsaufwand zu vermeiden. Die praktische Evaluation in Kapitel 9 gibt Aufschluss, wie gut dieses Ziel erreicht wurde.

### 4.1.2 Wichtigste Begrifflichkeiten

Hier werden nun einige Begriffe definiert, die für die Analyse von Bedeutung sind. Dadurch kann das informell schon gegebene Ziel der Ermittlung von Schleifengrenzen formal präzisiert werden.

**Definition 4.1.1 (Schleife)**
Eine *Schleife* entspricht ihrer transformierten Form, wie sie in Kapitel 3 vorgestellt wurde. □

Eine Schleife ist somit eine endrekursive Prozedur, was eine Anwendung der verwendeten interprozeduralen Analysetechnik erst erlaubt.

**Definition 4.1.2 (Schleifeniteration)**
Sei eine Schleife $l$ gegeben. Eine *Schleifeniteration* dieser Schleife $l$ entspricht einem Aufruf der zugehörigen Schleifenroutine. Die erste Iteration erfolgt durch den Aufruf des äußeren Schleifenaufrufs, alle weiteren durch den des inneren rekursiven Aufrufs. □

**Beispiel 4.1.2 (Vergleich zu Schleifen in C)**
Die oben definierten Schleifeniterationen entsprechen nur bei *do-while*-Schleifen der Zahl der Ausführungen des Schleifenrumpfs. Bei *while-* und *for*-Schleifen ist die Iterationszahl um eins größer als die Ausführungszahl des Rumpfs.

**Definition 4.1.3 (Äußerer Schleifenkontext)**
Den Kontext, in dem der äußere Aufruf der Schleifenroutine stattfindet, bezeichnet man als *äußeren Schleifenkontext*.

Mehrere solcher äußeren Kontexte können entstehen, falls die Prozedur, die den äußeren Schleifenaufruf enthält, von mehreren unterscheidbaren Aufrufstellen ausgeführt wird. □

**Definition 4.1.4 (Innere Schleifenkontexte)**
Alle Kontexte, die durch den Aufruf der Schleifenroutine von außen oder durch den rekursiven Aufruf erzeugt werden, bezeichnet mal als *innere Schleifenkontexte*.

Ist das Abrollen der Schleifen durch die verwendete Grapherweiterung verhindert, kann nur ein innerer Kontext pro äußeren Kontext entstehen, nämlich durch den äußeren Schleifenaufruf. Erlaubt die Grapherweiterung ein $n$-faches Abrollen, können noch zusätzlich $n$ innere Kontexte pro äußeren Kontext entstehen. Diese repräsentieren die ersten $n$ Iterationen der jeweiligen Schleife. □

**Definition 4.1.5 (Schleifengrenze)**
Eine *obere* bzw. *untere Schleifengrenze* für eine Schleife $l$ ist eine obere bzw. untere Schranke für die Anzahl ihrer möglichen Schleifeniterationen. Man spricht von einer *sicheren Schleifengrenze*, falls diese eine konservative Schleifengrenze für den zu betrachtenden äußeren Schleifenkontext darstellt. □

Aufbauend auf diesen Definitionen gelangt man zu folgender formalen Zusammenfassung des gewünschten Analyseergebnisses:

**Ergebnis der Schleifenanalyse**

Sei ein Eingabeprogramm $P$ mit den Schleifen $l_1, \ldots, l_n$ gegeben. Aufgabe der hier vorgestellten Analyse wird es sein, möglichst exakte sichere untere und obere Schleifengrenzen für die vorhandenen Schleifen zu bestimmen. Jeder äußere Schleifenkontext von $l_1$ bis $l_n$ wird getrennt betrachtet.

## 4.2 Ausgangssituation

Die Ergebnisse der vorgestellten Analyse werden in die weiteren Berechnungen des existierenden WCET-Analysators *aiT* der AbsInt GmbH einfließen. Das erlaubt eine praktische Evaluierung der Nützlichkeit des neu entworfenen Verfahrens.

Um dies zu erreichen, muss die Analyse so konstruiert werden, dass sie auch mit den anderen Teilen der *aiT-Architektur* zusammenspielt und somit die momentan in *aiT* enthaltene Schleifenanalyse ersetzen kann. Dies ermöglicht eine gute Gegenüberstellung der beiden Analysemethoden in der in Kapitel 9 folgenden Evaluation.

Damit die mit dieser Vorgehensweise verbundenen Designentscheidungen verständlich werden, wird nun im Folgenden die allgemeine Architektur von *aiT* dargestellt. Detailliert wird auf das für die Schleifenanalyse relevante Eingabeformat der Analysekomponenten und die verwendete Werteanalyse eingegangen. Auch wird kurz die bisher in *aiT* verwendete Schleifenanalyse dargestellt,

### 4.2.1 Architektur von aiT

Der *WCET-Analysator aiT* basiert auf der stufenweisen Kooperation einer Vielzahl von Komponenten. Einen Überblick über den Aufbau einer WCET-Analyse in *aiT* liefert Abbildung 4.1 auf der nächsten Seite.

Als Eingabe erhält *aiT* ein Programm in Form des Maschinencodes der jeweiligen Zielplattform, also z.B. als ausführbare Datei für die *PowerPC-Architektur*. Aus diesem

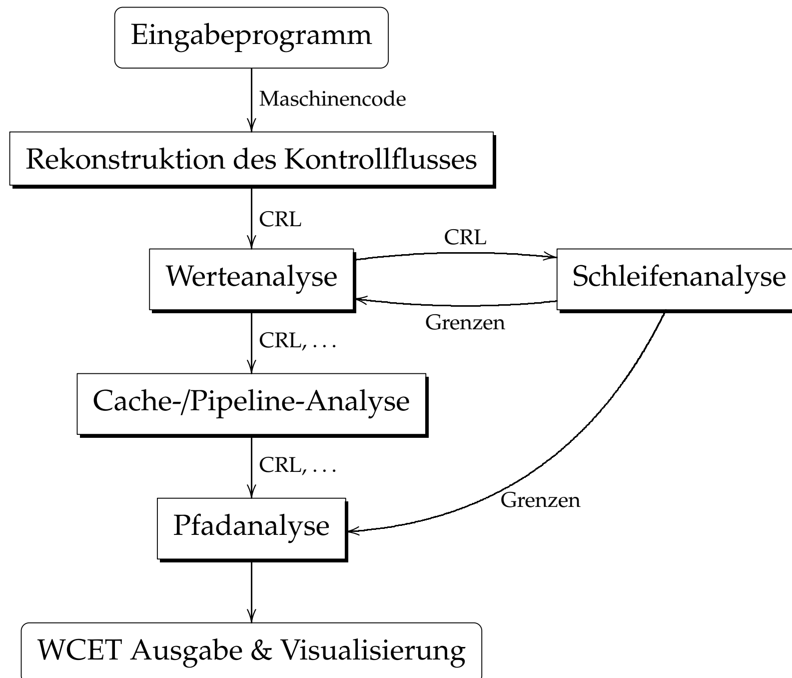



Abbildung 4.1: Ablauf einer Analyse von aiT

wird dann von einer *aiT-Komponente* der Kontrollflussgraph berechnet. Als Austauschformat für den Kontrollflussgraph wird das *CRL-Format* verwendet. Dieses wird später noch beschrieben, da es auch die Eingabe der Schleifenanalyse ist.

Danach folgt der für die Schleifenanalyse interessante Teil. Auf Basis des errechneten Kontrollflussgraphen wird eine Werteanalyse vorgenommen. Hierbei kooperiert die Werteanalyse mit der Schleifenanalyse. Zunächst wird eine Werteanalyse vorgenommen unter der Annahme alle Schleifen seien unbeschränkt, und danach wird eine Schleifenanalyse angestoßen, um Schleifengrenzen auf Basis der erhaltenen Daten der Werteanalyse zu berechnen. Dieses Verfahren wird iterativ wiederholt, diese Iteration wird als *Feedback-Iteration* bezeichnet. Hierbei wird mittels der Ausgabe der vorhergehenden Iteration jeweils die Werteanalyse verfeinert, um in der nächsten Iteration eventuell bessere Grenzen zu finden. Die Ausgabe des letzten Durchlaufs der Feedback-Iteration wird später der Pfadanalyse übergeben, um die Ausführungszahl der Basisblöcke der Schleifen zu ermitteln. An dieser Stelle wird auch die neue Schleifenanalyse integriert. Daher wird das *CRL-Format* als Eingabe verwendet und ebenfalls eine Kooperation mit dem Werteanalysator (*daan*) angestrebt.

Sind Schleifen- und Werteanalyse beendet, beginnt die Cache- und Pipeline-Analyse, welche sowohl das Timingverhalten der Prozessorpipeline als auch der angebundenen Speicher und Caches miteinbezieht. Als Ergebnisse erhält man Ausführungszeiten für die einzelnen Basisblöcke des Programms. Nun wird in einer Pfadanalyse der längste Pfad des zu analysierenden Programms bestimmt, wobei auch die berechne-

ten Schleifengrenzen mit einfließen. Für weitere Informationen zur Pfadanalyse sei [The98] empfohlen. Als Ergebnis erhält man die konkrete WCET, falls möglich, und es wird eine Visualisierung erstellt.

Detailiertere Informationen über den Aufbau und die Funktionsweise von *aiT* erhält man z.B. in [Abs05b]. Die Visualisierung der Kontrollflussgraphen, die in dieser Arbeit verwendet wird, erfolgt mittels des Graphviewer *aiSee* (für nähere Informationen siehe [Abs05a]).

### 4.2.2 Eingabeformat (CRL)

Als Eingabe für die Schleifenanalyse dient unter anderem der Kontrollflussgraph in seiner *CRL-Repräsentation* (Control Flow Representation Language). Dieser Kontrollflussgraph wird aus dem Maschinencode rekonstruiert.

Die CRL-Beschreibung enthält das Programm untergliedert in Prozeduren, Basisblöcke und Instruktionen. Durch Attribute kann man auf die Instruktionsnamen der einzelnen Instruktionen zugreifen bzw. auf deren Argumente. Die Argumente sind schon in Ziel- und Quellparameter unterteilt. Somit kann man unter Verwendung der jeweiligen Architekturbeschreibung die für die Analyse benötigte Semantik der einzelnen Instruktionen erfassen.

**Beispiel 4.2.1 (CRL-Datei)**
Als Grundlage für das Beispiel dient das C-Programm aus Listing 4.2 auf der nächsten Seite. Dieses enthält eine einfache Funktion *main* mit einer einzelnen Schleife.

Die Kontrollflussrekonstruktion erhält als Eingabe das Kompilat dieses Programms für die *PowerPC-Architektur*. Dieses wurde mit einem in der Industrie eingesetzen Compiler erzeugt. Der rekonstruierte Kontrollfluss wird in Abbildung 4.2 auf Seite 36 illustriert.

Listing 4.3 auf Seite 32 zeigt die vereinfachte CRL-Beschreibung des Beispielprogramms. Hierbei sind alle für die Schleifenanalyse nicht notwendigen Attribute entfernt. Aus Gründen der Übersichtlichkeit werden nur die für den Aufruf der Schleifenroutine nötigen Teile und die Schleifenroutine dargestellt.

Die CRL-Beschreibung enthält sehr viele Informationen. Die Schleifenanalyse benötigt hiervon nur einen Bruchteil. Hier die für die Erkennung der Schleifengrenzen interessanten Teile:

**Definition 4.2.1 (Instruktionskennung)**
Jede Instruktion in der CRL Beschreibung hat ein *genname* Attribut, das den eindeutigen Namen der jeweiligen Instruktion enthält. Mit Hilfe dieses Namens lässt sich die Semantik der Instruktion bestimmen. □

```c
int main (int argc, char *argv[])
{
  int i = 0;

  while (i < 16)
    ++i;

  return i;
}
```

Listing 4.2: Beispielprogramm in C

**Definition 4.2.2 (Ziele einer Instruktion)**
Für jede Instruktion gibt es die Attribute $dst_1, \ldots, dst_n$. Diese enthalten die Ziele der jeweiligen Instruktion. Zum einen können dies konkret bekannte Ziele wie Register sein, zum anderen erst noch zu bestimmende abstrakte Ziele wie der Speicher. □

**Definition 4.2.3 (Quellen einer Instruktion)**
Analog wie für die Ziele enthält jede Instruktion die Attribute $src_1, \ldots, src_n$. Diese geben an, welche Ressourcen von der Instruktion gelesen werden. □

Wie sich im Verlauf der Beschreibung der Analyse zeigen wird, genügen diese hier vorgestellten drei unterschiedlichen Attributgruppen, um die vorzustellende Schleifenanalyse zu ermöglichen.

### 4.2.3 Werteanalysator (daan)

Der in *aiT* verwendete *Werteanalysator daan* berechnet statisch die möglichen Inhalte von Registern bzw. Speicherzellen für die einzelnen Basisblöcke/Instruktionen für jeden Kontext. Als Wertebereich verwendet er die Intervalldarstellung. Das heißt, er gibt Auskunft in welchem Intervall sich der Wert z.B. eines Registers an der jeweiligen Stelle im Supergraph bewegt.

Der Analysator geht in der ersten Iteration von unbeschränkten Schleifen bei der Analyse aus. Die Schleifenanalyse versorgt ihn dann mit den erkannten Schleifengrenzen, was die berechneten Ergebnisse verbessert. So kann man exaktere Intervalle bestimmen.

Zusätzlich zu dieser Bestimmung von Speicher- und Registerinhalten erkennt der Werteanalysator auch, ob Kanten niemals durchlaufen werden können. Ist dies der Fall, werden diese als *infeasible* markiert. Dieser Ausschluss von unmöglichen Pfaden im Kontrollflussgraph trägt auch maßgeblich zu der Genauigkeit der erreichten Ergebnisse in der gesamten WCET-Analyse bei.

```
start with f1;

routine main_L1: name="main.L1", loop="1" {
  entry b5: {
    edges to b7/f, main_L1_exit/t;
    contains {
      0x00100084:4 "cmpi cr0, 0, r31, +16": genname="cmpi"
        , dst1="cr0", src3="r31", src4="+16";

      0x00100088:4 "bc 4, cr0.lt, 0x100094.f <0x100094>": genname="bc_Cond"
        , src1="4", src2="cr0", src3="lt", src4="0x100094", src5="f";
    }
  }

  block b7: backedges="b5"
  {
    edges to main_L1_rec/t;
    contains {
      0x0010008c:4 "addi r31, r31, +1": genname="addi", dst1="r31", src2="r31", src3="+1";

      0x00100090:4 "b 0x100084 <0x100084>": genname="b";
    }
  }

  block main_L1_rec: loopcallrec="1"
  {
    calls main_L1;
    edges to main_L1_exit;
  }

  exit main_L1_exit;
}

routine f1: name="main" {
  entry b0: {
    edges to main_L1_first;
    contains {  ............

      0x00100080:4 "addi r31, zero, +0": genname="addi", dst1="r31", src2="zero", src3="+0";
    }
  }

  block main_L1_first: loopcall="1"
  {
    calls main_L1;
    edges to b9;
  }

  block b9:
  {
    edges to x;
    contains {  ............
    }
  }

  exit x;
}
```

Listing 4.3: Teil der CRL Ausgabe für das Beispiel

Grundlage des Werteanalysators daan, wie er heute in *aiT* verwendet wird, ist die Diplomarbeit von Martin Sicks, die sich der Bestimmung von Speicherzugriffsadressen widmet, siehe hierzu [Sic97].

### 4.2.4 Bisherige Schleifenanalyse

Die momentan in *aiT* verwendete Schleifenanalyse ist in den Werteanalysator daan integriert. Sie besteht aus einer Menge von Pattern für bekannte Schleifenkonstrukte für von *aiT* unterstützte Industriecompiler.

Es erfolgt eine Kooperation der Schleifenanalyse mit der Werteanalyse. In mehreren Runden wird zunächst die Werteanalyse durchgeführt und danach mittels Patternmatching Schleifengrenzen ermittelt. Die erhaltenen Grenzen werden in der nächsten Runde von der Werteanalyse zur Verbesserung ihrer Ergebnisse verwendet.

## 4.3 Getroffene Designentscheidungen

Nachdem nun die Ziele der Analyse klar definiert sind und die Ausgangssituation bekannt ist, werden die grundsätzlichen Designentscheidungen erklärt. Hierbei handelt es sich um die wichtigsten Aspekte: die Beschränkung auf Zählschleifen, die Kooperation mit der schon existierenden Werteanalyse, die Art der Behandelung von geschachtelten Schleifen und die Notwendigkeit der parallelen Analyse mehrerer Schleifen.

### 4.3.1 Konzentration auf Zählschleifen

Aus der Erfahrung mit eingebetteten Systemen hat sich gezeigt, dass der vorherrschende Schleifentyp die klassische *Zählschleife* ist. Nur selten kommen Schleifenkonstrukte vor, in denen die Iterationszahl nicht nur von einem lediglich additiv modifizierten Zähler bestimmt wird. Die in *aiT* schon vorhandene Schleifenanalyse kann auch fast nur solche Zählschleifen erkennen und liefert trotzdem meist für die WCET-Bestimmung ausreichende Ergebnisse.

Somit stellt es sich als sinnvoll dar, sich auf diesen Schleifentypus zu konzentrieren. Daher ist die hier vorzustellende Analyse mit dem Ziel entwickelt, Grenzen von Zählschleifen zu bestimmen. Hierbei wird Augenmerk darauf gelegt, dass nach Möglichkeit alle Arten von Zählschleifen erkannt werden, also auch solche Schleifen, in denen ein ungenaues Intervall als Inkrement/Dekrement pro Iteration dient und welche mehrere Schleifentests besitzen. Diese Berücksichtigung von Schleifen mit ungenauem Inkrement/Dekrement kann die erreichten Ergebnisse gegenüber der aktuellen Analyse verbessern, die nur exakte Veränderungen des Schleifenzählers berücksichtigt.

Da Zählschleifen in der Praxis fast ausschließlich auf ganzzahligen Zählvariablen beruhen, ist auch diese Analyse auf Schleifen mit Integervariablen als Zähler beschränkt; Fließkommaarithmetik wird nicht in die Analyse miteinbezogen.

Das abschließende Kapitel 10 wird einen Ausblick auf mögliche Erweiterungen dieses Verfahrens auf z.B. Schleifen, deren Zähler multiplikativ verändert wird, liefern.

### 4.3.2 Kooperation mit Werteanalyse

Die bisherige Schleifenanalyse von *aiT* arbeitet direkt mit der Werteanalyse zusammen. Dies hat sich als fruchtbare Kooperation herausgestellt, da die Werteanalyse im Besitz vieler für die Schleifenanalyse relevanter Ergebnisse ist.

Zum einen klassifiziert die Werteanalyse Kanten wie schon erwähnt wenn möglich als infeasible. So kann man Schleifen von der Analyse aussparen, die nie im Programmverlauf erreicht werden. Desweiteren kann man dadurch auch Endlosschleifen erkennen, wie im nächsten Kapitel beschrieben wird.

Zusätzlich liefert die Werteanalyse die Adressen der Speicherzugriffe der einzelnen Instruktionen für alle Kontexte, sowie die Belegung der einzelnen Speicherzellen und Register.

Daher wird diese Kooperation weiterhin beibehalten. Hierzu wird die Schleifenanalyse als Erweiterung des Werteanalysators daan entworfen. Die Werteanalyse wird mehrmals ausgeführt. Nach jeder Ausführung erfolgt eine Anwendung der Schleifenanalyse und die Werteanalyse erhält als Feedback für den nächsten Lauf die bisher errechneten Schleifengrenzen. Dies wird bis zu einem Fixpunkt wiederholt, an dem keine neuen Schleifengrenzen gefunden werden bzw. sich die gefundenen nicht weiter ändern. Dieser Zyklus kann jederzeit abgebrochen werden, da die gefundenen Grenzen stets konservativ sind.

### 4.3.3 Analyse geschachtelter Schleifen

Viele zu analysierende Programme enthalten *geschachtelte Schleifen*. Zum einen können diese direkt geschaltet sein, z.B. wie in Listing 4.4 auf der nächsten Seite. Eine andere Möglichkeit ist, das eine Prozedur eine Schleife enthält und diese Prozedur innerhalb einer äußeren Schleife aufgerufen wird, wie in Listing 4.5 gezeigt ist.

Hierbei hängt oft, wie in den beiden gegebenen Beispielen, die Anzahl der Iterationen der inneren Schleife von den Grenzen der äußeren Schleife ab. Daher sollten die Schleifen ihrer Schachtelungstiefe nach getrennt analysiert werden, da die Grenzen der äußeren Schleife bekannt sein müssen, um die der inneren zu bestimmen. Dazu wird zunächst die äußerste Schicht untersucht und sich dann nach Innen vorgearbeitet.

Diese Designentscheidung bedeutet, dass zusätzlich zu der iterativen Kooperation mit der Werteanalyse auch noch innerhalb eines Durchlaufs der *Feedback-Iteration* eine innere Schleife über die Schachtelungstiefe der Schleifen stattfindet. Diese innere Iteration wird im Folgenden als *Schachtelungstiefe-Iteration* bezeichnet.

```
void main ()
{
  for (int i=0; i < 100; ++i)
    for (int j=0; j < i; ++j)
      something();
}
```

Listing 4.4: Direkt geschachtelte Schleifen

```
void inner (int max)
{
  for (int i=0; i < max; ++i)
    something();
}

void main ()
{
  for (int i=0; i < 100; ++i)
    inner(i);
}
```

Listing 4.5: Indirekt geschachtelte Schleifen

### 4.3.4 Parallele Analyse von mehreren Schleifen

Von PAG generierte Analysatoren arbeiten weitaus effektiver auf großen Kontrollflussgraphen als auf mehreren kleineren Teilen des Graph getrennt.

Daher ist es sinnvoller, möglichst viele Schleifen in einem Lauf der verwendeten Datenflussanalyse zu betrachten. Da man die einzelnen Schachtelungstiefen getrennt untersuchen will, werden alle Schleifen derselben Tiefe gleichzeitig analysiert. Dieses Vorgehen erfordert einige Vorkehrungen in der Gestaltung des Datenflussproblemes, die im Kapitel 6 zur Sprache kommen.

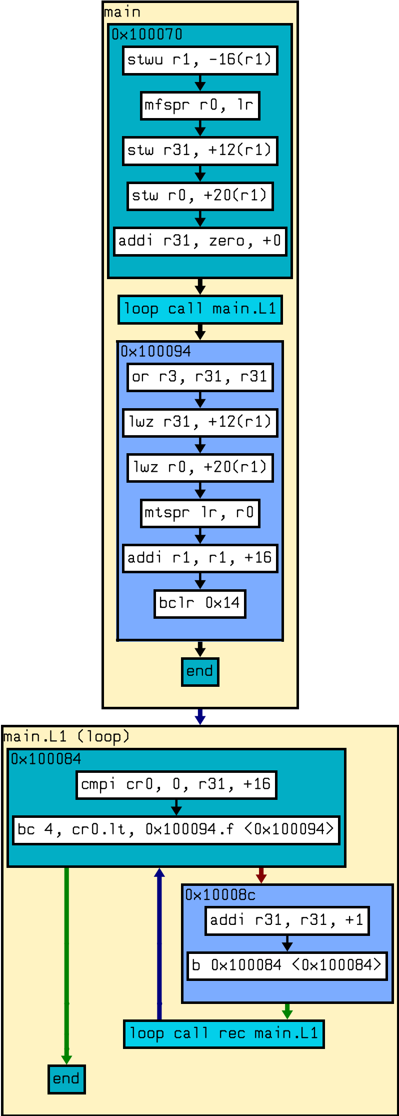



Abbildung 4.2: Kontrollfluss des Beispielprogramms

# Kapitel 5 Phasen der Analyse

Nachdem das zurückliegende Kapitel die grundlegenden Designentscheidungen dargestellt hat, wird nun der Ablauf einer Runde der *Schachtelungstiefe-Iteration* der Schleifenanalyse beschrieben. Es wird detailliert auf die einzelnen Phasen der Analyse eingegangen und die verwendeten Methoden werden erklärt.

Im Laufe der Beschreibung der einzelnen Stufen werden auch die verwendeten Begriffe definiert. Zusätzlich wird nach und nach eingeführt, welche Informationen die Werteanalyse zur Verfügung stellen muss und welche aus der CRL-Beschreibung extrahiert werden müssen.

Da das verwendete Datenflussproblem von besonderem Interesse ist, wird es in Kapitel 6 getrennt ausführlich behandelt.

## 5.1 Überblick

Aufgabe eines Durchlaufs der *Schachtelungstiefe-Iteration* ist es, für eine gegebene Menge von Schleifen gleicher Schachtelungstiefe eine Menge von sicheren Schleifengrenzen zu berechnen. Dabei ist man, wie in Kapitel 4 erläutert, sowohl an oberen als auch an unteren Schranken interessiert. Jeder äußere Schleifenkontext wird getrennt untersucht.

Dazu wird ein mehrstufiger Prozess verwendet. Es werden folgende Analysephasen unterschieden:

1. Klassifizierung der Schleifen
2. Erkennung potentieller Schleifenzähler
3. Invariantenanalyse
4. Erkennen & Lösen der Abbruchbedingungen

Diese Phasen werden in Reihenfolge ihres Auftretens im Analyseprozess hier eingeführt und erklärt.

## 5.2 Klassifizierung der Schleifen

Der zuerst einsetzende Prozess im Ablauf einer Analyserunde ist die Klassifizierung der einzelnen Schleifen der zu analysierenden Schleifenmenge. Zunächst wird die dafür verwendete Ausgabe der Werteanalyse eingeführt. Im Folgenden sind die einzelnen zu unterscheidenden Klassen von Schleifen definiert und die jeweils verwendete Methode zur Erkennung auf Basis der Information der Werteanalyse wird erläutert.

### 5.2.1 Information aus der Werteanalyse

Die Werteanalyse basiert auf einer Datenflussanalyse. Sie kann befragt werden, wie der eingehende Datenflusswert für eine Instruktion in einem Basisblock des Kontrollflussgraphen ist, sowie welcher Datenflusswert über eine ausgehende Kante hinter einer Instruktion weitergereicht wird.

Dies lässt sich nutzen, um zu bestimmen, ob die Ausführung eine Instruktion nie erreichen kann, bzw. über welche Kanten sie nach der Instruktion niemals weitergeführt wird. Hierfür führen wir folgende beiden Begriffe ein:

**Definition 5.2.1 (Unerreichbarer Basisblock)**
Ein Basisblock $b$ in einem Analysekontext $c$ des Kontrollflussgraphen eines Programms ist *unerreichbar*, wenn die Werteanalyse ihn als unerreichbar klassifiziert. □

**Definition 5.2.2 (Unausführbare Kante)**
Eine ausgehende Kante $k$ eines Basisblocks $b$ in einem Analysekontext $c$ des Kontrollflussgraphen Programms ist *unausführbar*, wenn die Werteanalyse sie als unausführbar klassifiziert. □

### 5.2.2 Unerreichbare Schleifen

**Definition 5.2.3 (Unerreichbare Schleifen)**
Sei eine Schleife $l$ gegeben, zusammen mit dem äußeren Call-Knoten $cn$ der Schleifenroutine. Ist der Call-Knoten $cn$ in einem äußeren Schleifenkontext unerreichbar, so ist die Schleife $l$ in diesem Kontext *unerreichbar*. □

Für alle äußeren Schleifenkontexte, in denen die jeweilige Schleife als unerreichbar klassifiziert wurde, wird sie mit den Iterationsgrenzen $[0, 0]$ annotiert. Diese äußeren Kontexte werden in den späteren Analysephasen nicht mehr genauer betrachtet, da die optimalen Grenzen schon ermittelt sind. Dieses Vorgehen kann eine erhebliche Einsparung an Speicherbedarf und Zeit bei großen Programmen mit sich bringen.

**Beispiel 5.2.1 (Unerreichbare Schleife)**
Das in Listing 5.1 dargestellte C-Programm enthält eine Schleife, bei der die Werteanalyse feststellen kann, dass sie nicht erreichbar ist. In keiner Ausführung der *main*-Funktion kann je die eingebettete Schleife ausgeführt werden.

```
int main (int argc, char *argv[])
{
  int i = 0;

  if (i > 0)
  {
    for (int j=0; j < 10; ++j)
      ++i;
  }

  return i;
}
```

Listing 5.1: Beispiel einer unerreichbaren Schleife

### 5.2.3 Endlosschleifen

**Definition 5.2.4 (Endlosschleifen)**
Sei eine Schleife $l$ gegeben, zusammen mit dem äußeren Call-Knoten $cn$ der Schleifenroutine und dem zugehörigem Return-Knoten $r$. Ist die lokale Kante die einzige ausführbare eingehende Kante des Return-Knotens $r$ für einen äußeren Schleifenkontext $c$, so ist diese Schleife in diesem Kontext eine *Endlosschleife*. □

In allen Kontexten, in denen eine Schleife als endlos erkannt wurde, wird diese mit den Grenzen $[1, \infty]$ annotiert. Auch diese Schleifen werden von der späteren Analyse ausgespart. Die obere Schranke kann nicht weiter verbessert werden, da die Werteanalyse definitiv festgestellt hat, dass der Return-Knoten der Schleife niemals aus der Schleifenroutine heraus wieder erreicht werden kann.

**Beispiel 5.2.2 (Unerwünschte Endlosschleife)**
Listing 5.2 auf der nächsten Seite zeigt ein Programm mit einer Schleife, die als endlos klassifiziert wird. Sie enthält einen typischen Programmierfehler, eine Vertauschung von $i$ und $j$ beim Inkrementieren. Die vorgeschaltete Klassifizierung erspart die aufwändige weitere Analyse solcher Schleifen.

**Beispiel 5.2.3 (Erwünschte Endlosschleife)**
Gewollte Endlosschleifen gibt es oft am Programmende, z.B. in der *exit*-Routine.

```
int main (int argc, char *argv[])
{
  int i = 0;

  for (int j=0; j < 10; ++i)
      ++i;

  return i;
}
```

Listing 5.2: Beispiel einer Endlosschleife

### 5.2.4 Schleifen mit oberer Schranke

**Definition 5.2.5 (Schleifen mit oberer Schranke)**
Sei eine Schleife $l$ gegeben, zusammen mit den Call-Knoten $cn$ des rekursiven Aufrufs und den inneren Schleifenkontexten $c_1, \ldots, c_n$ für den äußeren Schleifenkontext $c$. Die inneren Schleifenkontexte seien so geordnet, dass $c_j$ für die j-te Iteration steht. Ist ab einem inneren Kontext $c_i$ der Call-Knoten $cn$ in allen Kontexten $c_i, \ldots, c_n$ unerreichbar, so hat diese Schleife eine *obere Schranke* für diesen äußeren Kontext $c$. Diese obere Schranke ist $i$. □

Diese Schleifen erhalten die sichere obere Grenze $i$. Sie werden von der Analyse nicht ausgespart, da man noch die untere Schranke bestimmen muss. Selbst wenn die später folgenden Stufen der Schleifenanalyse keine weiteren Informationen beitragen, kann die obere Schranke immer noch verwendet werden. Dies kann z.B. für den Fall nützlich sein, dass die Werteanalyse durch das eingestellte Abrollen Grenzen für eine multiplikative Schleife findet.

**Beispiel 5.2.4**
Eine multiplikative Schleife ist in Listing 5.3 auf der nächsten Seite gezeigt. Die neue Schleifenanalyse erkennt jedoch nur additive Schleifen. Somit wären keine Grenzen bestimmbar. Allerdings kann die Werteanalyse, wenn die gewählte VIVU-Grapherweiterung eine genügend große Anzahl virtueller Abrollungen erlaubt, direkt berechnen, wann die Schleife abbricht. Ist z.B. maximal zehnfaches Abrollen erlaubt, wird die obere Grenze der Schleife erkannt werden.

### 5.2.5 Sonstige Schleifen

Alle Schleifen, die nicht in die schon vorgestellten drei Klassen fallen, werden an die folgenden Stufen weitergereicht. Über diese Schleifen kann alleine mit den Informationen über Erreichbarkeit und Ausführbarkeit nichts ausgesagt werden.

```
int main (int argc, char *argv[])
{
  int i = 0;

  for (int j=2; j < 8; j=j*j)
      ++i;

  return i;
}
```

Listing 5.3: Multiplikative Schleife

## 5.3 Erkennung der Schleifenzähler

Die Schleifenanalyse dieser Arbeit ist auf Zählschleifen fokussiert. Um die Grenzen von Zählschleifen zu berechnen ist es nötig, die Veränderung des Zählers der Schleife während einer Iteration zu erfassen. Dies ist Aufgabe einer der folgenden Analysestufen. Dazu müssen zunächst alle potentielle Zähler ermittelt werden.

### 5.3.1 Grundlegende Definitionen

Zunächst werden die Begrifflichkeiten definiert, die zum Verständnis dieser Phase relevant sind: Was sind Variablen für diese Analyse und was sind Schleifenzähler?

**Definition 5.3.1 (Variable)**
Eine *Variable* für die Schleifenanalyse ist entweder eine *Speicherzelle*, ein *Prozessorregister* oder die spezielle Null-Variable.

Eine Speicherzelle ist hierbei beschrieben durch die Adresse im Speicher zusammen mit der Zugriffsbreite, ein Prozessorregister durch seine Registernummer und die reale Registerbreite.

Eine Variable lässt sich somit als ein Tripel $V = (T, A, W)$ schreiben, wobei $T$ der Variablentyp, (Register, Speicher oder Null), $A$ die Adresse oder Registernummer und $W$ die Zugriffs- bzw. Registerbreite als natürliche Zahl, gemessen in Bytes, ist. □

Obige Definition erlaubt, dass sich Variablen überlappen. So können zwei unterschiedliche Variablen z.B. sich überlappende Speicherzellen oder Teilregister beschreiben. Während der Analyse werden solche gegenseitigen Beeinflussungen miteinbezogen, um konservativ zu bleiben. Dafür sorgt die Funktionsweise der Transferfunktion des verwendeten Datenflussproblems, wie sie in Kapitel 6 beschrieben wird.

**Definition 5.3.2 (Schleifenzähler)**
Ein *Schleifenzähler* ist eine Variable, die im Laufe einer Iteration modifiziert wird und die Einfluss auf eine Abbruchbedingung der Schleife besitzt. □

In der weiteren Analyse werden nur Scheifenzähler betrachtet, deren Startintervall zu Beginn der ersten Iteration bekannt ist. Nur mit diesen kann sinnvoll weitergerechnet werden.

### 5.3.2 Information aus der Werteanalyse/CRL

Diese Phase benötigt mehr als die schon eingeführten Daten zur Erreichbarkeit und Ausführbarkeit. Die Werteanalyse muss Informationen über die Speicher- und Registerinhalte liefern sowie die von einzelnen Instruktionen verwendeten Adressen zum Lesen und Beschreiben des Speichers. Weiterhin werden die von einer Instruktion verwendeten Register benötigt. Da die Schleifenanalyse auf den vorher eingeführten Variablen arbeitet, wird dieser Sachverhalt nun wie folgt definiert:

**Definition 5.3.3 (Variablenwert)**
Sei eine Variable $v$ gegeben. Für eine Instruktion $i$ in einem Basisblock $b$ des Kontrollflussgraphen eines zu analysierenden Programms ist der *Variablenwert* von $v$ in einem Kontext $c$ gegeben durch das zugehörige Intervall, das die Werteanalyse als Inhalt der Speicherzelle bzw. des Prozessorregisters, auf die $v$ verweist, ermittelt hat. Dieser Wert ist gültig vor Ausführung der Instruktion $i$. □

**Definition 5.3.4 (Benutzte Variablen)**
Sei eine Instruktion $i$ in einem Basisblock $b$ des Kontrollflussgraphen eines zu analysierenden Programms gegeben. In einem Kontext *c benutzt* diese Instruktion folgende Variablen:

- Variablen für die Registernummern, die in der CRL-Beschreibung für die Instruktion $i$ annotiert sind, werden je nach dem ob sie als Quelle oder Ziel definiert sind, gelesen oder modifiziert
- Variablen für die Speicherzellen, die nach Informationen der Werteanalyse benutzt werden, werden je nach dem ob sie als Quelle oder Ziel von der Werteanalyse klassifiziert sind, gelesen oder modifiziert □

### 5.3.3 Erkennungsverfahren

Diese Analysephase hat das Ziel, alle für die folgenden Datenflussanalyse relevanten Variablen zu erfassen, die den zu analysierenden Schleifen als Schleifenzähler dienen könnten.

Jeder äußere Schleifenkontext jeder zu analysierenden Schleife wird in dieser Stufe einzeln betrachtet. Die Ermittlung der potentiellen Zähler erfolgt dabei in drei Schritten:

#### Aufsammeln aller in der Schleife verwendeten Variablen

Bei diesem Schritt wird zunächst die CRL-Beschreibung bzw. die Ausgabe der Werteanalyse verwendet, um zu erfahren, welche Variablen von Instruktionen innerhalb der Schleifenroutine benutzt werden. Hierbei werden alle inneren Schleifenkontexte betrachtet, die für den gegebenen äußeren Kontext existieren. Alle erhaltenen Variablen werden zunächst für die weitere Verarbeitung gesammelt.

#### Ermittlung der Startintervalle

Für diese verwendeten Variablen werden nun alle Variablenwerte zu Beginn der ersten Iteration ermittelt. Dabei wird die Ausgabe der Werteanalyse verwendet, um den Variablenwert jeder einzelnen gesammelten Variablen für die Call-Instruktion im äußeren Call-Knoten der Schleifenroutine zu ermitteln. Als Kontext dient der äußere Schleifenkontext, der betrachtet wird. Dadurch erhält man das Startintervall der jeweiligen Variable zu Beginn der ersten Iteration.

#### Aussortieren der unbrauchbaren Variablen

Alle Variablen deren Startintervalle unbeschränkt sind, werden nicht weiter betrachtet. Sie bieten keine Möglichkeit zur weiteren Analyse, da man einen Schleifenzähler, dessen Startwert unbekannt ist, nicht für weitere Berechnungen verwenden kann. Zurück bleibt eine Menge von Variablen mit bekannten beschränkten Startintervallen, dieses sind potentielle Schleifenzähler.

**Beispiel 5.3.1**
Der in Abbildung 5.1 auf der nächsten Seite dargestellte Kontrollflussgraph gehört zu einer Schleifenroutine eines PowerPC-Programms.

Der erste Schritt würde für diese Schleife folgende Variablen ermitteln:

$$(Register, 26, 4), (Register, 27, 4), (Register, 29, 4), (Register, 30, 4)$$

Diese werden von den *addi*- und *cmp*- Instruktionen gelesen bzw. modifiziert, was man den jeweiligen *src* und *dst* Attributen der CRL-Beschreibung entnehmen kann. Die Werteanalyse zeigt, dass keine Speicherzugriffe erfolgen und somit keine Variablen für Speicherzellen liefern.

Angenommen, dass die Werteanalyse für den zu betrachtenden äußeren Kontext nur für die Variable $(Register, 30, 4)$ ein beschränktes Intervall liefert, nämlich $[1, 10]$.

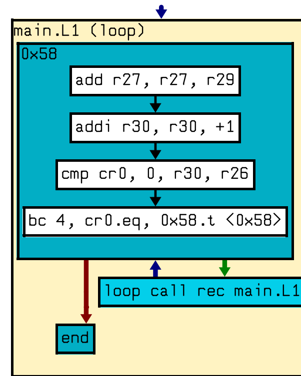



Abbildung 5.1: Kontrollfluss der Schleifenroutine

Dann werden alle Variablen bis auf $(Register, 30, 4)$ entfernt. Somit erhält man für diese Schleife als potentiellen Schleifenzähler nur die Variable $(Register, 30, 4)$.

## 5.4 Berechnung der Schleifeninvarianten

Diese Phase ist das eigentliche Herzstück der neuen Analyse. Nachdem man durch die vorhergehende Phase eine Menge potentieller Schleifenzähler erhalten hat, will man nun deren Veränderung während einer Iteration berechnen. Dies geschieht symbolisch auf einer eigens dafür eingeführten Notation unter Verwendung einer Datenflussanalyse. Diese Instanz der Datenflussanalyse wird im Folgenden als *Invariantenanalyse* bezeichnet.

Im Folgenden sind nun zunächst die Ziele sowie die formalen Grundlagen dieser Stufe aufgezeigt. Dann wird kurz die Invariantenanalyse eingeführt. Eine genauere Beschreibung dieser Analyse erfolgt in Kapitel 6.

### 5.4.1 Ziel der Invariantenanalyse

Die Invariantenanalyse hat das Ziel, an jede Kante des Supergraphen innerhalb der betrachteten Schleifenroutinen eine Menge von Schleifeninvarianten zu annotieren. Diese werden in einer späteren Stufe dazu genutzt, um die Abbruchbedingungen der Schleife auszuwerten und mögliche Schranken zu berechnen.

Die Invarianten werden als Gleichungen dargestellt. Die Sprache, in der diese Gleichungen geschrieben sind, ist speziell für diese Analyse entworfen, im Folgenden soll diese als *IVALA* bezeichnet werden. Die Gleichungen beschreiben, wie sich Variablen vom Beginn der Schleifeniteration bis zu dem jeweiligen Punkt im Kontrollflussgraph in dem entsprechenden Kontext verändern. Da die Analyse auf Zählschleifen abzielt,

wurde die Menge der darstellbaren Formeln so beschränkt, dass nur lineare Veränderungen durch Addition von Intervallen ausdrückbar sind. Die Analyse arbeitet mit Integer-Arithmetik, d.h. es werden also nur Ganzzahlen betrachtet, Fließkommazahlen werden nicht berücksichtigt. Dies ist in der Praxis nur eine kleine Einschränkung, weil Zählschleifen normalerweise nur über ganzzahligen Schleifenzählern operieren.

Die Invariantenanalyse ist für die Konstruktion dieser Gleichungen zuständig, sie trifft keine Aussagen über die Iterationsdauer oder wertet Schleifentests aus. Dies ist den späteren Phasen der Analyse überlassen. Trotzdem ist dies das Kernstück des Verfahrens, da hier die Semantik des Maschinencodes in eine später auswertbare Form abstrahiert wird. Aus den Maschineninstruktionen werden die Veränderungen der potentiellen Schleifenzähler extrahiert und in Gleichungen umgeformt.

### 5.4.2 Verwendete Gleichungssysteme

Um die Invariantenanalyse beschreiben zu können, muss zunächst die Sprache *IVALA* definiert werden, in der die von der Analyse verwendeten Gleichungen notiert sind. Dazu wird die Syntax dieser Gleichungen eingeführt und danach deren Semantik definiert.

#### Syntax der Gleichungen

**Definition 5.4.1 (Konstante)**
Eine *Konstante* ist in *IVALA* ein nichtleeres beschränktes Intervall über den ganzen Zahlen. □

Das heißt Konstanten sind zum einen exakte Intervalle wie $[1, 1]$ aber auch ungenaue Intervalle wie $[1, 10]$. Nicht als Konstanten gültig sind hingegen Intervalle der Form $[1, \infty[$.

**Definition 5.4.2 (Ausdruck)**
In *IVALA* ist ein *Ausdruck* ein Quadrupel $W = (V, K, Z, S)$. Hierbei ist $V$ eine Variable, $K$ eine Konstante, $Z$ eine natürliche Zahl, welche die Zugriffsbreite in Byte darstellt, und $S$ die Kennung für die Vorzeichenerweiterung: nicht erweitert (*notext*), erweitert mit eins (*ext*) oder unbekannt (*unknown*). □

**Definition 5.4.3 (Gleichung)**
Eine *Gleichung* in *IVALA* ist ein Tupel $G = (V, M)$, wobei $V$ eine Variable und $M$ eine endliche nicht leere Menge von Ausdrücken ist. □

Eine gültige Gleichung in *IVALA* ist z.B.:

$$((Register, 30, 4), \{((Register, 20, 4), [1, 10], 4, ext), ((Speicher, \mathrm{0xffff}, 4), [1, 1], 2, notext)\})$$

### Semantik der Gleichungen

**Definition 5.4.4 (Semantik einer Gleichung)**
Sei eine Gleichung $G = (V_0, M)$ gegeben. Die Menge $M$ der Ausdrücke sei gegeben als:

$$M = \{(V_1, K_1, Z_1, S_1), \ldots, (V_n, K_n, Z_n, S_n)\}$$

Sei diese Gleichung nun Teil einer an eine Kante des Supergraphen innerhalb einer Schleifenroutine in einem Kontext $c$ annotierten Gleichungsmenge. Die Ausdrucksmenge $M$ beschreibt alle möglichen Inhalte der Variable $V_0$ an dieser Stelle.

Jeder Ausdruck steht hierbei für eine Formel $V_k + K_k$, der als Wert das Ergebnis der Addition der Konstante $K_k$ zu dem Inhalt der Variable $V_k$ hat. Im Fall, dass die Variable den Typ *Null* besitzt, hat diese den Wert 0. Ansonsten ist der Wert der Variable $V_k$ der Wert beim Beginn der jeweiligen Schleifeniteration, also an der eingehenden Kante des Entry-Knotens der Schleifenroutine im aktuellen Kontext $c$.

Die zusätzlichen Komponenten $Z_k$ und $S_k$ der jeweiligen Ausdrücke geben an, welche Zugriffsbreite bei der Verwendung des Ausdrucks zu beachten ist und ob und wie der Inhalt vorzeichenerweitert wird. Das liefert die notwendigen Informationen, um Überläufe zu erkennen und festzustellen, ob der Inhalt einer Variablen eine vorzeichenlose oder vorzeichenhafte Ganzzahl ist. □

**Beispiel 5.4.1 (Anwendungsbeispiel)**
Sei der Kontrollfluss einer Schleifenroutine gegeben wie in Abbildung 5.1 auf Seite 44 dargestellt.

Register 30 wird nur von einer Instruktion der Schleife verändert, von *addi r30, r30, +1*. Diese Instruktion ist eine Addition der Konstanten 1 auf den Wert des Registers 30. Somit wird Register 30 in jeder Iteration um eins erhöht.

Die Menge von Gleichungen an der ausgehenden Kante der obigen Instruktion enthält nach der Invariantenanalyse folgende Gleichung:

$$((\mathit{Register}, 30, 4), \{((\mathit{Register}, 30, 4), [1, 1], 4, \mathit{notext})\})$$

Das bedeutet, dass Register 30 in jeder Schleifeniteration an dieser Stelle den Wert, den es zu Beginn der jeweiligen Iteration besaß addiert mit dem Intervall $[1, 1]$, hat:

$$R30 = R30 + [1, 1]$$

Zusätzlich muss man die Breite von 4 Byte und die nicht vorhandene Vorzeichenerweiterung beachten.

## 5.5 Auswertung der Schleifentests

In der letzten Phase der Schleifenanalyse wird das Ergebnis der Invariantenanalyse genutzt, um alle Schleifentests einer Schleife zu untersuchen. Es wird versucht, die Abbruchbedingung zu rekonstruieren und mit Hilfe der berechneten Schleifeninvarianten zu lösen. Diesen Vorgang kann man in mehrere Teile untergliedern: Zunächst die Extraktion der Ungleichung für die Abbruchbedingung (maschinenabhängig), dann das Lösen der erhaltenen Ungleichung: Dies führt ein dafür entwickelter Solver maschinenunabhängig durch. Zuletzt werden alle Ergebnisse, die man für die Schleifentests einer Schleife erhalten hat kombiniert.

### 5.5.1 Extraktion der Gleichungssysteme

Jeder Schleifentest einer Schleife wird nach seiner Abbruchbedingung untersucht, um diese in ein System von später zu lösenden Gleichungen zu transformieren. Hierzu muss zunächst definiert werden, was ein Schleifentest einer Schleife ist.

**Definition 5.5.1 (Schleifentest)**
Sei $l$ die Schleifenroutine einer Schleife im Kontrollflussgraphen. Der End-Knoten der Routine hat zumindest eine eingehende Kante. Eine davon entsteht durch die Schleifentransformation und kommt von dem Return-Knoten des inneren rekursiven Aufrufs von $l$. Jede weitere eingehende Kante kommt von einem Basisblock, der *Schleifentest* genannt wird. □

Im Folgenden wird die bei der Schleifentransformation erzeugte Kante zum End-Knoten der Schleife ignoriert, es werden nur die sonstigen eingehenden Kanten des End-Knotens dargestellt.

**Beispiel 5.5.1 (Schleifentest)**
Abbildung 5.2 auf der nächsten Seite zeigt eine Schleifenroutine. Ihr einziger Schleifentest ist der Block $0x43c$.

Die Schleifentests werden jetzt semantisch untersucht. Es kommt auf die jeweilige Implementierung für die Architektur an, wie die zugrundeliegende Abbruchbedingung im Detail erkannt wird. Das ist neben der Definition des Datenflussproblems der zweite der beiden architekturabhängigen Komponenten dieser Analyse.

Allgemein gilt, dass die letzte Instruktion des Schleifentests ein bedingter Sprung ist. Wäre es kein bedingter Sprung, würde dieser Basisblock nicht zu der Schleifenroutine gehören, sondern würde von der Schleifentransformation nicht mit aufgenommen. Durch den bedingten Sprung existieren zwei ausgehenden Kanten für den Schleifentest: eine um in der Schleiferoutine zu bleiben, eine um diese zu verlassen.

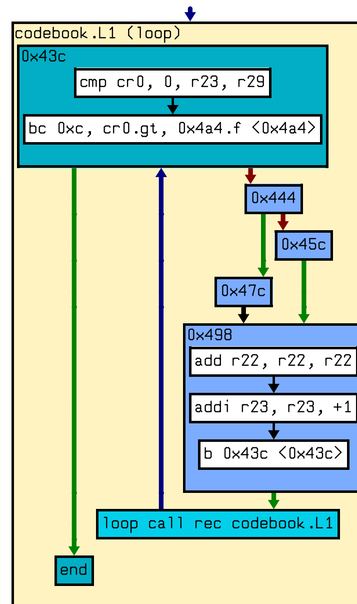



Abbildung 5.2: Schleifenroutine mit einem Schleifentest

Bedingte Sprünge sind typischerweise an den Wert eines Condition-Registers gekoppelt. Es erfolgt nun auf allen Architekturen ein Rückwärts-Slicen, um die letzte Instruktion im Schleifentest zu finden, die diese Bedingung verändert. Kann keine solche Instruktion identifiziert werden oder ist die gefundene Instruktion nicht von der jeweiligen Implementierung auswertbar, wird dieser Schleifentest für die weitere Analyse verworfen und erhält die sichere Schleifengrenze $[1, \infty]$.

**Beispiel 5.5.2 (Auswerten eines Schleifentests)**
Im Schleifentest *0x43c* der Schleife aus Abbildung 5.2 hängt der bedingte Sprung am Ende des Tests nur von dem Condition-Register *cr0* ab. Dieses wird von der *cmp*-Instruktion direkt davor gesetzt.

Wurde die Instruktion gefunden, die das Condition-Register setzt, so wird versucht, diese als Gleichung darzustellen, deren einzige Variable auf eine Schleifeninvariante zurückführbar ist. Zunächst wird untersucht ob alle Operanden der Instruktion bis auf eine konstant sind. Die Informationen dazu liefert die Werteanalyse. Falls diese Bedingung nicht erfüllt ist, erhält dieser Schleifentest die sichere Grenze $[1, \infty]$. Ansonsten bleibt nur eine freie Variable, die weiter analysiert werden muss.

**Beispiel 5.5.3 (Abbruchbedingung)**
Im oben betrachteten Schleifentest *0x43c* operiert die *cmp*-Instruktion auf zwei Registern, $R23$ und $R29$. Die Werteanalyse zeigt, das $R29$ konstant $[16, 16]$ ist. Daher bleibt nur $R23$ als freie Variable übrig, die weiter untersucht werden muss. Als Abbruchbedingung erhält man $R23 > [16, 16]$.

Dann wird überprüft, ob für die freie Variable, die in der Gleichung für die Abbruchbedingung verwendet wird, eine Gleichung im eingehenden Datenflusswert der Instruktion existiert, die das Condition-Register setzt. Ist dies nicht der Fall, erhält der Test als Schranke $[1, \infty]$.

Bei der weiteren Berechnung werden nur Gleichungen betrachtet, die der gesuchten Variable maximal zwei Ausdrücke zuordnen. Zwei Ausdrücke sind nur dann zulässig, wenn einer von ihnen konstant ist, also eine Variable vom Typ *Null* enthält. Dies stellt keine große Einschränkung in der Analyse von Schleifen dar, mehr dazu in der theoretischen Diskussion in Kapitel 8.

Für die Variable des nicht konstanten Ausdrucks wird ebenfalls eine Gleichung im Datenflusswert gesucht, allerdings in dem eingehenden Datenflusswert des rekursiven Aufrufs der Schleife. Diese Gleichung beschreibt die Veränderung dieser Variablen innerhalb einer kompletten Schleifeniteration. Auch hier werden nur Gleichungen mit maximal zwei Ausdrücken beachtet, von denen höchstens einer nicht konstant sein darf.

Sind alle diese Schritte erfolgreich, so erhält man ein Gleichungssystem für den jeweiligen Schleifentest. Andernfalls wird dieser Schleifentest nicht weiter betrachtet und erhält als Schranke $[1, \infty]$.

**Satz 5.5.1 (Erhaltenes Gleichungssystem)**
Das erhaltene Gleichungssystem besteht aus einer (Un-)Gleichung, welche die Abbruchbedingung mathematisch darstellt, und zwei Gleichungen, welche die Veränderung der darin vorkommenden Variable während einer Iteration beschreiben.

- Die Abbruchbedingung: eine (Un-)Gleichung mit einer Variablen $v$ und einer Konstanten $c$
- Die Gleichung für diese Variable $v$: $v = (i + c_1)$ oder $v = k_1$
- Die Gleichung für die Variable $i$ auf der $v$ beruht: $i = (i + c_2)$ oder $i = k_2$

Die erste (Un-)Gleichung des Systems gibt die Abbruchbedingung an. Ist diese erfüllt, erfolgt der Abbruch der Schleife. Dies hängt nur von dem Wert von $v$ ab. Die Veränderung von $v$ vom Beginn einer Schleifeniteration bis zu der Überprüfung der Bedingung wird durch die zweite Gleichung angegeben. Die Veränderung des $v$ zugrundeliegenden Schleifenzählers $i$ während einer Schleifeniteration über allen möglichen Pfaden ist in der dritten Gleichung beschrieben.

Die möglichen (Un-)Gleichungen, die von der Analyse als Abbruchbedingung ge-

handhabt werden können, sind auf folgende Varianten eingegrenzt:

$$v = c$$
$$v \neq c$$
$$v < c$$
$$v > c$$
$$v \leq c$$
$$v \geq c$$

Die $v = k_1$ und $i = k_2$ Teile der beiden letzten Gleichungen sind optional, sie tauchen nur auf, wenn der jeweiligen Variablen innerhalb der Schleife eventuell eine Konstante zugewiesen wird. Dies wird z.B. verwendet, wenn aus einer inneren Schleife die äußere Schleife abgebrochen werden soll.

Die in der Implementierung verwendeten Gleichungen enthalten zusätzlich noch andere Informationen, die in den Gleichungen der Datenflusswerte eingebettet sind: die Breite der Variable in Bytes und deren Vorzeichenerweiterung. Diese Informationen werden von dem Solver verwendet, um Überläufe festzustellen.

**Beispiel 5.5.4 (Erstellung des Gleichungssystems)**
Abbildung 5.3 zeigt den Kontrollfluss einer Schleifenroutine.

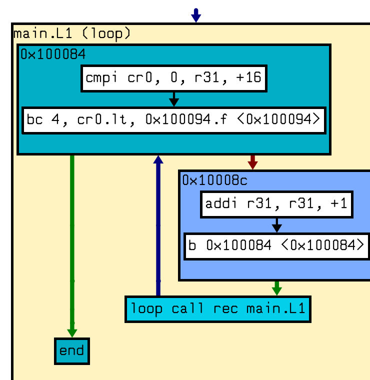



Abbildung 5.3: Beispiel für Gleichungskonstruktion

Der erste Basisblock der Schleife ist ihr einziger Schleifentest. Der bedingte Sprung an seinem Ende hängt von dem Condition-Register ab, das die Compare-Instruktion direkt davor setzt. Diese Compare-Instruktion vergleicht Register 30 mit der Konstanten 16. Somit erhält man folgende Ungleichung für den Abbruch der Schleife:

$$R30 \geq [16, 16]$$

Der eingehende Datenflusswert der Compare-Instruktion enthält für $R30$ folgende Gleichung:

$$R30 = R30 + [0, 0]$$

$R30$ wurde also seit Beginn der Schleifeniteration nicht verändert. Die Gleichung für $R30$ vor dem rekursiven Aufruf ist:

$$R30 = R30 + [1,1]$$

$R30$ wird also in einer kompletten Schleifeniteration immer um genau eins erhöht.

Damit hat man folgende drei Gleichungen erhalten:

$$R30 \geq [16,16]$$
$$R30 = R30 + [0,0]$$
$$R30 = R30 + [1,1]$$

**Beispiel 5.5.5 (Anwendung auf eine weitere Schleife)**
Für die Schleifenroutine aus Abbildung 5.2 auf Seite 48 wurde für den einzigen vorhandenen Schleifentest als Abbruchbedingung $R23 > [16,16]$ bestimmt. Innerhalb der Schleifeniteration wird $R23$ nicht verändert bevor dieser Test erfolgt. Jedoch wird $R23$ in jeder Schleifeniteration nach dem Test durch die *addi*-Instruktion um eins erhöht. Daher ergibt sich für den Schleifentest folgendes Gleichungssystem:

$$R23 > [16,16]$$
$$R23 = R23 + [0,0]$$
$$R23 = R23 + [1,1]$$

**Beispiel 5.5.6 (Sinnvolle Zuweisung einer Konstant)**
Listing 5.4 auf der nächsten Seite zeigt eine äußere Schleife, deren Schleifenzähler entweder jeweils um eins erhöht wird oder durch die innere Schleife auf einen Wert gesetzt wird, der den Abbruch der äußeren Schleife verursacht.

Als Gleichungssystem für den einzigen Schleifentest der äußeren Schleife erhält man:

$$i \geq [10,10]$$
$$i = i + [0,0]$$
$$i = i + [1,1] \text{ oder } i = [10,10]$$

### 5.5.2 Erstellung einer geschlossenen (Un-)Gleichung

Alle Schleifentests, die noch weiter betrachtet werden müssen, sind nun mit Gleichungssystemen annotiert. Diese müssen nun in Beziehung mit der Iterationszahl gebracht werden und alle anderen freien Variablen eliminiert werden. Ziel ist es, eine einzelne geschlossene (Un-)Gleichung zu erhalten, die nur noch von der Iterationszahl abhängt.

Ohne Beschränkung der Allgemeinheit sei die erste Gleichung des Systems:

$$v < c$$

```
int main (int argc, char *argv[])
{
  int i=0;

  while (i < 10)
  {
    ++i;

    for (int j=0; j < 10; ++j)
    {
      if (brichBeideSchleifenAb())
      {
        i = 10;
        break;
      }
    }
  }

  return i;
}
```

Listing 5.4: Zuweisung einer Konstante an einen Schleifenzähler

Der Schleifentest prüft also, ob die Konstante unterschritten wird. Alle anderen möglichen Varianten der Abbruchbedingungen werden analog behandelt. Die restlichen Gleichungen des Systems seien:

$$v = (i + c_1) \text{ oder } v = k_1$$
$$i = (i + c_2) \text{ oder } i = k_2$$

Durch Einsetzen der Gleichungen ineinander erhält man eine geschlossene Gleichung, welche die Anzahl der rekursiven Aufrufe *reccalls* sowie den Startwert $c_0$ des Schleifenzählers $i$ als einzige freien Variablen besitzt. Da man an konservativen Ergebnisse erzielen will, muss man eine Konjunktion der vorher disjunktiv verbundenen Teile erstellen, da alle Teilgleichungen gelten müssen damit der Abbruch wirklich erfolgt.

$$(c_0 + (\mathit{reccalls} * c_2) + c_1) < c \text{ und } k_1 < c \text{ und } (k_2 + c_1) < c$$

Da der Startwert des Schleifenzählers durch die Werteanalyse bekannt ist, kann man unter Verwendung dieser Formel die minimal und maximal möglichen rekursiven Aufrufe der Schleife bestimmen. Die Anzahl der Schleifeniterationen wurde als Anzahl der rekursiven Aufrufe der Schleifenroutine plus dem äußeren Aufruf definiert. Daher kann man aus der Anzahl der rekursiven Aufrufe *reccalls* die Anzahl der Schleifeniterationen berechnen, indem man eins zu *reccalls* hinzuzählt.

**Beispiel 5.5.7 (Anwendung auf vorhergehendes Beispiel)**
In Beispiel 5.5.4 auf Seite 50 wurde schon ein Gleichungssystem für einen Schleifentest eines Programms erstellt, nämlich:

$$\begin{aligned} v &\geq [16, 16] \\ v &= i + [0, 0] \\ i &= i + [1, 1] \end{aligned}$$

Als Startwert für $i$ ermittelt die Werteanalyse $[0, 0]$. Es ergibt sich daher die geschlossene Formel:

$$[0, 0] + (\text{reccalls} * [1, 1]) + [0, 0] \geq [16, 16]$$

### 5.5.3 Lösen der (Un-)Gleichung

Allen Schleifentests, die nicht mit den Schranken $[1, \infty]$ aussortiert wurden, wurde jeweils eine Gleichung zugeordnet. Die jeweilige Gleichung muss gelöst werden, um ein sicheres Paar aus unterer und oberer Grenze zu bestimmen. Diese Aufgabe übernimmt ein Solver, der als Eingabe die Gleichung erwartet und die gesuchten Grenzen zurückgibt.

Die möglichen Eingabegleichungen sind:

$$\begin{aligned} &(c_0 + (\text{reccalls} * c_2) + c_1) = c \text{ und } k_1 = c \text{ und } (k_2 + c_1) = c \\ &(c_0 + (\text{reccalls} * c_2) + c_1) \neq c \text{ und } k_1 \neq c \text{ und } (k_2 + c_1) \neq c \\ &(c_0 + (\text{reccalls} * c_2) + c_1) < c \text{ und } k_1 < c \text{ und } (k_2 + c_1) < c \\ &(c_0 + (\text{reccalls} * c_2) + c_1) > c \text{ und } k_1 > c \text{ und } (k_2 + c_1) > c \\ &(c_0 + (\text{reccalls} * c_2) + c_1) \leq c \text{ und } k_1 \leq c \text{ und } (k_2 + c_1) \leq c \\ &(c_0 + (\text{reccalls} * c_2) + c_1) \geq c \text{ und } k_1 \geq c \text{ und } (k_2 + c_1) \geq c \end{aligned}$$

Die beiden rechten Teile der Konjunktion erhält man, wenn in der Schleife dem Schleifenzähler eine Konstante zugewiesen wird. Ob sie erfüllt sind, lässt sich mittels Intervallarithmetik ermitteln. Sind die beiden konstanten Teile erfüllt, wird von einer unteren Grenze für die Schleifeniterationen von 1 ausgegangen, da der Abbruch potentiell in jeder Iteration auftreten kann. Andernfalls wird die Schleife vielleicht nie verlassen, daher wird als sichere obere Grenze $\infty$ verwendet.

Bei der Behandlung des linken Teils unterscheidet der Solver die folgenden drei Gruppen von Gleichungen:

1. $c_0 + (\text{reccalls} * c_2) + c_1 = c$
2. $c_0 + (\text{reccalls} * c_2) + c_1 \neq c$

3. $c_0 + (\text{reccalls} * c_2) + c_1 < c$
   $c_0 + (\text{reccalls} * c_2) + c_1 > c$
   $c_0 + (\text{reccalls} * c_2) + c_1 \leq c$
   $c_0 + (\text{reccalls} * c_2) + c_1 \geq c$

Als zusätzliche Informationen erhält man die Breite in Byte und die Vorzeichenerweiterung, die zu beachten sind. Diese werden benötigt, damit die Implementierung des Solvers Überläufe erkennen kann. Intern rechnet der Solver mit einer größeren Bitbreite als von der zu untersuchenden Architektur verwendet wird, also so z.B. mit 64 Bit bei der PPC-Architektur, die selbst nur 32 Bit breite Register besitzt. Soll eine Architektur analysiert werden, die 64 Bit breite Register unterstützt, so wird der Solver selbst mit 128 Bit breiten Zahlen rechnen. Somit kann der Solver durch eine Überprüfung, ob die berechneten Werte der Schleifenzähler innerhalb der geforderten Bytebreite verbleiben, Überläufe erkennen. Stellt der Solver einen Überlauf fest, liefert er als Ergebnis die sichere Schranke $[1, \infty]$.

**Lösen der Gleichungen mit =**

Ist eines der in der Gleichung vorkommenden konstanten Intervalle ungenau, kann keine Aussage getroffen werden ob der Zielwert je exakt erreicht wird. Für diesen Fall werden die sicheren Grenzen $[1, \infty]$ gewählt. Ist das Intervall $c_2$ $[0, 0]$, kann durch Intervallarithmetik der Wert der linken Seite bestimmt werden und geprüft werden ob die Gleichung erfüllt ist. Falls nicht, ist ebenfalls vom Intervall $[1, \infty]$ auszugehen. Ist sie erfüllt, bricht die Schleife garantiert in der ersten Iteration ab, daher werden als Grenzen $[1, 1]$ erhalten.

Für die übrigen Fälle formt man die Eingangsgleichung um in:

$$\text{reccalls} = \frac{(c - c_1 - c_0)}{c_2}$$

Da alle Intervalle exakt sind, können diese als Ganzzahlen aufgefasst und die mögliche Anzahl an Iterationen direkt berechnet werden. Ist das Ergebnis keine Ganzzahl, wird als Grenze wieder $[1, \infty]$ ausgegeben, da dann der Endwert nicht exakt erreicht wird. Ansonsten erhält man das exakte Intervall $[\text{reccalls} + 1, \text{reccalls} + 1]$.

**Lösen der Ungleichungen mit ≠**

Sei die Eingabegleichung:

$$c_0 + (\text{reccalls} * c_2) + c_1 \neq c$$

Dieses Problem kann man zurückführen auf die Lösung von

$$c_0 + (\text{reccalls} * c_2) + c_1 < c \text{ oder } c_0 + (\text{reccalls} * c_2) + c_1 > c$$

Man löst diese beiden Ungleichungen mit der im nächsten Punkt vorgestellten Methode und erhält so je ein Intervall für jeden der Fälle. Diese Intervalle werden zu einer gemeinsamen Grenze vereinigt.

### Lösen der sonstigen Ungleichungen

Sei als Eingabe die folgende Gleichung gegeben:

$$c_0 + (\text{reccalls} * c_2) + c_1 > c$$

Auch hier ist entscheidend, welche Grenzen das Intervall $c_2$ besitzt. Es existieren folgende vier Fälle:

1. Ist $c_2$ das triviale Intervall $[0, 0]$, kann der konstante Wert der linken Seite ausgerechnet und die Bedingung überprüfen werden. So ergibt sich entweder $[1, 1]$ oder $[1, \infty]$ als Grenze.

2. Ist die obere Grenze des Intervalls $c_2$ eine negative Zahl, so muss überprüft werden, ob die Bedingung schon vor dem ersten rekursiven Aufruf erfüllt ist. Ist sie es nicht, handelt es sich um eine Endlosschleife.

3. Haben die Intervallgrenzen von $c_2$ unterschiedliche Vorzeichen oder ist eine der Grenzen 0, kann nicht mit Sicherheit bestimmt werden, ob die Bedingung jemals erfüllt wird, da nicht feststellbar ist, ob der Schleifenzähler streng monoton wächst. In diesem Fall ist die Grenze $[1, \infty]$.

4. Der interessante Fall ist der, dass die untere Grenze von $c_2$ positiv ist. Die in der Ungleichung vorkommenden Intervalle $c, c_0, c_1, c_2$ seien definiert als $[l, u], [l_0, u_0], [l_1, u_1], [l_2, u_2]$.

   Die untere Schranke lässt sich durch folgende Gleichung bestimmen:

   $$\text{recalls}_{\text{lower}} = \frac{(l - u_1 - u_0)}{u_2}$$

   Hierbei wird im Falle, dass $\text{recalls}_{\text{lower}}$ keine ganze Zahl ist, aufgerundet, ansonsten wird eins hinzuaddiert.

   Analog wird die obere Schranke berechnet durch:

   $$\text{recalls}_{\text{upper}} = \frac{(u - l_1 - l_0)}{l_2}$$

   Auch hier erfolgt wieder Aufrundung bzw. Aufaddition von eins.

| Gleichungsart | $\text{recalls}_{\text{lower}}$ | $\text{recalls}_{\text{upper}}$ | Aufrunden | Eins addieren |
|---|---|---|---|---|
| $>$ | $\frac{(l-u_1-u_0)}{u_2}$ | $\frac{(u-l_1-l_0)}{l_2}$ | Ja | Ja |
| $<$ | $\frac{(u-l_1-l_0)}{l_2}$ | $\frac{(l-u_1-u_0)}{u_2}$ | Ja | Ja |
| $\geq$ | $\frac{(l-u_1-u_0)}{u_2}$ | $\frac{(u-l_1-l_0)}{l_2}$ | Ja | Nein |
| $\leq$ | $\frac{(u-l_1-l_0)}{l_2}$ | $\frac{(l-u_1-u_0)}{u_2}$ | Ja | Nein |

Tabelle 5.1: Formeln zur Lösung der Gleichungen

Die sonstigen Gleichungsarten $<$, $\geq$ und $\leq$ werden analog behandelt. Bei $\geq$ entfällt jedoch das Aufaddieren von eins und bei den $<$ und $\leq$ Varianten handelt es sich um die dualen Formulierungen von $>$ und $\geq$. Tabelle 5.1 zeigt alle Formeln zur Lösung der verschiedenen Gleichungsarten.

**Beispiel 5.5.8 (Lösen des Beispiels)**
Für Beispiel 5.5.7 auf Seite 53 ist folgende geschlossene Gleichung für die Anzahl der rekursiven Schleifenaufrufe bekannt:

$$[0,0] + (\text{reccalls} * [1,1]) + [0,0] \geq [16,16]$$

Durch Anwendung des Lösungsverfahren für $\geq$-Gleichungen ergeben sich folgende Werte für die Anzahl der rekursiven Aufrufe:

$$\text{recalls}_{\text{lower}} = \frac{16}{1} = 16$$
$$\text{recalls}_{\text{upper}} = \frac{16}{1} = 16$$

Daher erhält man folgende Grenzen für diese Schleife: $[17,17]$.

**Beispiel 5.5.9 (Komplexere Intervalle)**
Gegeben sei die Ungleichung:

$$[123,167] + (\text{reccalls} * [-4,-1]) + [2,4] \leq [32,48]$$

Durch Anwendung des Lösungsverfahren für $<$-Gleichungen findet man folgende Werte:

$$\text{recalls}_{\text{lower}} = \frac{48-123-2}{-4} = 19,25$$
$$\text{recalls}_{\text{upper}} = \frac{32-167-4}{-1} = 139$$

Als Grenzen ergibt dies $[20,139]$.

### Komplexität

Die hier vorgestellte Lösungmethode für die Gleichungsarten verlangt nur einfache Additions- und Divisions-Operationen sowie einige Auswertungen von Vergleichen. Da die Formelgröße statisch beschränkt ist auf die maximal drei Elemente der linken Seite und die rechte Konstante, kann das Lösen dieser Formeln in konstanter Zeit durchgeführt werden.

Da die Formeln, wenn auch um endliche Intervalle erweitert, eine Untermenge der *Presburger-Arithmetik* darstellen, könnte man auch einen Solver für diese verwenden, wie z.B. die Omega-Bibliothek. Dies würde jedoch eine Laufzeit von schlimmstenfalls

$$2^{2^{2^{\text{Größe der Formel}}}}$$

verursachen und man müsste die Formeln vorher so umformen, dass keine Intervalle mehr enthalten sind und dass die Überläufe betrachtet werden, was eine Vergrößerung derselben zur Folge hätte. Für mehr Informationen zur Verwendung der Presburger-Arithmetik ist [Pug94] zu empfehlen.

Auf Grund der wesentlich höheren Komplexität des Lösungsvorgangs bei Verwendung der *Presburger-Arithmetik* wurde davon Abstand genommen.

## 5.5.4 Kombination der Ergebnisse

Nach dem Durchlauf des Solvers besitzen nun alle Schleifentests sichere Schranken. Im Folgenden seien die Schleifentests einer Schleife als $e_0, \ldots, e_n$ bezeichnet. Die ihnen als Grenzen zugeordneten Intervalle seien $i_0, \ldots, i_n$.

Nun wird eine intraprozedurale Dominator-Analyse ausgeführt, um festzustellen, ob ein Schleifentest der Schleifenroutine auf jedem Pfad vom Eingang der Routine bis zu ihrem rekursiven Aufruf liegt, d.h. ob er diesen also dominiert.

**Definition 5.5.2 (Intraprozedurale Dominanz)**
Sei der Kontrollflussgraph einer Routine $r$ mit Entry-Knoten $e$ gegeben. Ein Knoten $k$ dieses Graphen *dominiert* einen Knoten $l$, wenn gilt:

$$\forall \pi = P[e, l] : k \in \pi \qquad \square$$

Nur Schleifentests, die den rekursiven Aufruf dominieren, können zu der Festlegung der oberen Grenze beitragen. Alle anderen können nur die untere Grenze vermindern. Die Grenzen für die gesamte Schleife berechnen sich dann wie folgt:

**Satz 5.5.2 (Untere Schleifengrenze)**
Die untere Schleifengrenze für eine Schleife ist die untere Schranke aller ermittelten Grenzen $i_0, \ldots, i_n$ für die jeweiligen Schleifentests. Es spielt keine Rolle, ob der jeweilige Schleifentest den rekursiven Aufruf dominiert oder nicht.

**Satz 5.5.3 (Obere Schleifengrenze)**
Die obere Schleifengrenze ist die untere Schranke aller Grenzen $i_k$ der Schleifentests $e_k$, die den rekursiven Aufruf der Schleife dominieren.

Mittels Kombination der Intervalle für alle Schleifentests ergibt sich eine sichere untere und obere Schranke für die Iterationszahl der jeweiligen Schleife. Da jeder äußere Schleifenkontext getrennt betrachtet wird, erhält man eigenständige Schranken für jeden dieser Kontexte.

Kapitel

# 6 Invariantenanalyse

Die vorangegangenen beiden Kapitel haben die Funktionsweise der Schleifenanalyse und den Ablauf einer ihrer Iterationen beschrieben. Dabei wurde gezeigt, wie man unter Verwendung von Schleifeninvarianten, die durch eine Datenflussanalyse, der Invariantenanalyse, berechnet werden, sichere Schleifengrenzen bestimmen kann. Bisher ist die Invariantenanalyse als *Blackbox* behandelt worden, nur die Syntax und Semantik der darin verwendeten Formeln, die man dann aus den Datenflusswerten erhält, wurden erläutert.

In diesem Kapitel wird die Invariantenanalyse selbst beschrieben. Zunächst wird eine Einführung in die Zielsetzung des Datenflussproblems erfolgen. Danach wird der zugrundeliegende Verband eingeführt und auf die Transfer- und Kombinationsfunktion bzw. das verwendete Widening eingegangen. In dem abschließenden Abschnitt wird dann die Spezifikation des Datenflussproblems für *PAG* angesprochen.

## 6.1 Einführung & Ziele

Das Ziel der Invariantenanalyse ist es, eine Menge von Gleichungen für Schleifeninvarianten an jede Kante zu annotieren. Diese beschreiben die Modifikation von Schleifenzählern innerhalb einer Schleifeniteration vom Eintritt in die Schleifenprozedur bis zum Eintritt in den aktuellen Knoten.

Da alle diese Invarianten eine Veränderung einer Variable vom Beginn der Schleifenprozedur bis zu ihrem rekursiven Aufruf beschreiben, ist das Datenflussproblem vorwärts gerichtet, es handelt sich also um ein *Vorwärtsproblem*.

Zu Beginn der Schleifenprozedur beginnt die Invariantenanalyse mit einer Startmenge von Gleichungen, die alle zu betrachtenden potentiellen Schleifenzähler beschreiben. Aufgabe der Transferfunktion ist es, diese Gleichungsmenge unter Berücksichtigung der Semantik der Instruktionen anzupassen, um die Veränderungen der jeweiligen Variablen zu beobachten.

Da innerhalb einer Schleife häufig weitere Schleifen und Prozeduraufrufe vorkommen, muss die Analyse diese ebenfalls berücksichtigen können, um möglichst exakte Ergebnisse zu ermitteln. Sie ist daher eine *interprozedurale* Analyse.

## 6.2 Abstrakter Verband

In dem vorhergehenden Kapitel wurden in Abschnitt 5.4.2 auf Seite 45 die Gleichungen vorgestellt, auf denen die Invariantenanalyse arbeitet. Der für die Invariantenanalyse verwendete Verband dient zur Formalisierung dieser Gleichungen aus *IVALA*.

Die schon für diese Formeln in *IVALA* eingeführten Variablen und Ausdrücke sind Tupel von Ganzzahlen und Aufzählungselementen. Die Konstanten aus *IVALA* werden als Tupel von Ganzzahlen dargestellt. Der Datenflusswert ist eine Funktion, die Variablen eine Menge von Ausdrücken zuordnet. Eine solche Zuordnung entspricht gerade einer Gleichung in *IVALA*, die Funktion kann somit also ein Gleichungsmenge interpretiert werden.

## 6.3 Transferfunktion

Die Aufgabe der Transferfunktion ist es, die für die Schleifenanalyse relevante Semantik des Programms in die Funktion einzubringen, die den Variablen die Ausdrücke zuordnet. Diese Funktion wird je nach Semantik der Instruktion modifiziert.

Es werden die folgenden drei verschiedenen Arten von Kanten in ihrer Behandlung unterschieden:

### 6.3.1 Call-Kanten

Bei den Call-Kanten werden zwei Varianten unterschieden. Je nachdem, ob es sich um den Aufruf einer zu analysierenden Schleife oder um einen sonstigen Aufruf handelt, ändert sich die Funktionsweise der Transferfunktion.

#### Äußerer und rekursiver Aufruf einer zu analysierenden Schleife

Die Transferfunktion an dieser Kante ergibt konstant eine Anfangsmenge von Gleichungen für den zugehörigen äußeren Kontext der Schleife. Diese Menge besteht aus je einer Gleichung für jeden potentiellen Schleifenzähler, den die vorhergehende Phase der Analyse gefunden hat. Jede Gleichung ist dabei eine Selbstzuweisung der Variable.

Dass dies nicht nur bei dem erstmaligen äußeren Aufruf geschieht, sondern auch für den rekursiven Aufruf, liegt daran, dass man Invarianten über das Verhalten dieser Variablen innerhalb einer Iteration berechnen will. Somit muss man also am Anfang jeder Iteration wieder mit diesen abstrakten Startwerten beginnen.

**Beispiel 6.3.1 (Anfangsgleichungsmenge)**
Angenommen die vorhergehende Phase der Analyse findet folgende potentiellen Schleifenzähler:

- Register 5 und 30
- Speicherzellen $\langle 0\text{xff00}, 4\rangle$ und $\langle 0\text{xffff}, 4\rangle$

Dann erhält man folgende Startmenge von Gleichungen:

$$\begin{aligned}
(\mathit{Register}, 5, 4) &\rightarrow \{((\mathit{Register}, 5, 4), [0,0], 4, \mathit{notext})\} \\
(\mathit{Register}, 30, 4) &\rightarrow \{((\mathit{Register}, 30, 4), [0,0], 4, \mathit{notext})\} \\
(\mathit{Speicher}, 0\text{xff00}, 4) &\rightarrow \{((\mathit{Speicher}, 0\text{xff00}, 4), [0,0], 4, \mathit{notext})\} \\
(\mathit{Speicher}, 0\text{xffff}, 4) &\rightarrow \{((\mathit{Speicher}, 0\text{xffff}, 4), [0,0], 4, \mathit{notext})\}
\end{aligned}$$

#### Sonstige Aufrufe

Bei allen anderen Call-Kanten ist die Transferfunktion die Identität.

### 6.3.2 Return-Kanten

Auch bei den Return-Kanten werden zwei Klassen unterschieden. Zum einen die Return-Kante aus einer zu analysierenden Schleife, die zu dem äußeren Aufruf gehört, zum anderen alle sonstigen Return-Kanten.

#### Verlassen der Schleife nach Außen

Die Transferfunktion an der Return-Kante, die zu dem äußeren Aufruf der Schleife gehört, ergibt konstant eine leere Formelmenge. Die Invariantenanalyse bestimmt nur Invarianten für die inneren Kanten der betrachteten Schleifenprozeduren und der von dort aufgerufenen Prozeduren, daher wird außerhalb dieser keine Information benötigt.

#### Sonstige Return-Kanten

An allen sonstigen Return-Kanten werden wie bei den sonstigen Call-Kanten die Datenflusswerte unverändert weiterpropagiert.

### 6.3.3 Normale Kanten

Sei eine Instruktion $i$ mit einer ausgehenden Kante $e$ gegeben. Der eingehende Datenflusswert für $i$ enthalte die Gleichungsmenge $eq$. Der ausgehende Datenflusswert berechnet sich nun in zwei Schritten. Zuerst werden alle Gleichungen aus $eq$ entfernt, deren Zielvariable von der aktuellen Instruktion $i$ verändert wird. Dadurch erhält man eine Menge $eq_{clean}$. Zu dieser Menge $eq_{clean}$ werden in einem zweiten Schritt dann neue Gleichungen hinzugefügt, die man auf Basis der eingehenden Menge $eq$, der Semantik der Instruktion $i$ und der Informationen aus der Werteanalyse berechnen kann. Als Ausgabe erhält man dann eine dritte Menge von Gleichungen $eq_{result}$.

Diese beiden Schritte werden als *Gleichungselimination* und *Gleichungskreation* bezeichnet.

**Beispiel 6.3.2 (Verwendete Beispiele für PowerPC)**
Für die Beispiele in den folgenden Beschreibungen werden diese Daten verwendet:

- Eingehende Formelmenge $eq$:

$$\begin{aligned}
(Register, 30, 4) \rightarrow &\{((Speicher, \text{0xffff}, 4), [1, 1], 4, notext),\\
&((Register, 23, 4), [0, 0], 4, notext)\}\\
(Register, 10, 4) \rightarrow &\{((Register, 10, 4), [0, 0], 4, notext)\}\\
(Speicher, \text{0xffff}, 4) \rightarrow &\{((Speicher, \text{0xffff}, 4), [0, 0], 4, notext)\}
\end{aligned}$$

- Untersuchte Instruktionen $i_1$ und $i_2$:
  - $i_1$: addi R30, R30, 2
    Semantik: $R30$ wird um zwei erhöht
  - $i_2$: stw R30, 0xffff
    Semantik: Inhalt von $R30$ wird in Speicherzelle $\langle \text{0xffff}, 4\rangle$ geschrieben

#### Gleichungselimination

Die *Gleichungselimination* hilft mit, dass die Analyse *sichere* Invarianten liefert. Dieser Teil der Transferfunktion ist weitgehend architekturunabhängig implementierbar, da die Informationen, welche Register geschrieben werden, aus den CRL-Attributen der Instruktion und die geschriebenen Speicherbereiche aus der Werteanalyse stammen. Jedoch unterstützen einige Architekturen, darunter auch die *PowerPC* Architektur, *Load-Multiple-Operationen*, in denen mehr Register geschrieben werden als in den CRL-Attributen vermerkt ist. Für diesen Fall muss ein Teil der zugehörigen Transferfunktion pro Architektur definiert werden.

Würde man die CRL-Repräsentation mit einer erweiterten Semantik versehen, könnte man auch diese architekturabhängigen Teile entfallen lassen und die *Gleichungselimination* vollkommen architekturunabhängig gestalten.

**Beispiel 6.3.3 (Instruktion $i_1$)**
Die CRL-Attribute besagen, dass nur Register $R30$ geschrieben wird, und die Werteanalyse zeigt, dass keine Speicherbereiche geschrieben werden. Daher muss nur die Gleichung für $R30$ aus $eq$ entfernt werden. Dies liefert für das gegebene $eq$ folgende Menge $eq_{clean}$:

$$(Register, 10, 4) \rightarrow \{((Register, 10, 4), [0, 0], 4, notext)\}$$
$$(Speicher, 0\text{xffff}, 4) \rightarrow \{((Speicher, 0\text{xffff}, 4), [0, 0], 4, notext)\}$$

**Beispiel 6.3.4 (Instruktion $i_2$)**
In diesem Fall besagen die CRL-Attribute, dass keine Register geschrieben werden. Die Werteanalyse zeigt, dass der Speicherbereich $\langle 0\text{xffff}, 4\rangle$ geschrieben wird. Aus $eq$ müssen somit alle Gleichungen, die Speicherzellen aus diesem Bereich beschreiben, entfernt werden. Aus $eq$ erhält man dann folgende Menge $eq_{clean}$:

$$\begin{aligned}(Register, 30, 4) \rightarrow \{&((Speicher, 0\text{xffff}, 4), [1, 1], 4, notext),\\ &((Register, 23, 4), [0, 0], 4, notext)\}\\ (Register, 10, 4) \rightarrow \{&((Register, 10, 4), [0, 0], 4, notext)\}\end{aligned}$$

### Gleichungskreation

Da diese Schleifenanalyse auf Zählschleifen über ganzzahligen Zählern ausgerichtet ist, genügt es, diese Überführung der Semantik für Additions-/Subtraktions- sowie Zuweisungsoperationen zu implementieren. Hierbei können Zuweisungen auch das Laden und Speichern von Registern im Speicher beinhalten sowie die Vorzeichenerweiterung oder die Maskierung der oberen Bits, um den Wertebereich einzuengen.

Da die jeweilige Semantik der Instruktion von der betrachteten Architektur und der jeweiligen Instruktion abhängt, ist die Implementierung dieses Teiles der Transferfunktion vollständig architekturabhängig. Daher wird die Gleichungskreation nur anhand der beiden Beispieleingaben veranschaulicht.

**Beispiel 6.3.5 (Instruktion $i_1$)**
Die Semantik der Instruktion ist, dass $R30$ der Inhalt von $R30$ inkrementiert um 2 zugewiesen wird. Somit ist $R30$ gleichzeitig Quelle und Ziel der Operation. Zunächst wird gesucht, ob für die Quelle $R30$ eine beschreibende Gleichung in $eq$ enthalten ist. Es ist:

$$\begin{aligned}(Register, 30, 4) \rightarrow \{&((Speicher, 0\text{xffff}, 4), [1, 1], 4, notext),\\ &((Register, 23, 4), [0, 0], 4, notext)\}\end{aligned}$$

Nun wird zu jedem Ausdruck dieser Gleichung das Intervall $[2, 2]$ hinzuaddiert, wie es die Semantik von $i_1$ verlangt. Dadurch wird folgende neue Ausdrucksmenge berechnet:

$$\{((Speicher, 0\text{xffff}, 4), [3, 3], 4, notext), ((Register, 23, 4), [2, 2], 4, notext)\}$$

Also ist $eq_{result}$:

$$\begin{aligned}
(Register, 30, 4) &\rightarrow \{((Speicher, \text{0xffff}, 4), [3,3], 4, notext),\\
&\quad ((Register, 23, 4), [2,2], 4, notext)\}\\
(Register, 10, 4) &\rightarrow \{((Register, 10, 4), [0,0], 4, notext)\}\\
(Speicher, \text{0xffff}, 4) &\rightarrow \{((Speicher, \text{0xffff}, 4), [0,0], 4, notext)\}
\end{aligned}$$

**Beispiel 6.3.6 (Instruktion $i_2$)**
Durch die Instruktion $i_2$ wird der Inhalt des 4-byte breiten Registers $R30$ in die Speicherzelle $\langle \text{0xffff}, 4\rangle$ gespeichert. Quelle der Operation ist Register $R30$, Ziel die Speicherzelle $\langle \text{0xffff}, 4\rangle$. Die eingehende Ausdrucksmenge für $R30$ ist:

$$\{((Speicher, \text{0xffff}, 4), [1,1], 4, notext), ((Register, 23, 4), [0,0], 4, notext)\}$$

Die ist nun die neue rechte Seite für $(Speicher, \text{0xffff}, 4)$. Die resultierende Ausgabemenge $eq_{result}$ ist daher:

$$\begin{aligned}
(Register, 30, 4) &\rightarrow \{((Speicher, \text{0xffff}, 4), [1,1], 4, notext),\\
&\quad ((Register, 23, 4), [0,0], 4, notext)\}\\
(Register, 10, 4) &\rightarrow \{((Register, 10, 4), [0,0], 4, notext)\}\\
(Speicher, \text{0xffff}, 4) &\rightarrow \{((Speicher, \text{0xffff}, 4), [1,1], 4, notext),\\
&\quad ((Register, 23, 4), [0,0], 4, notext)\}
\end{aligned}$$

**Behandlung von Guards**

Manche Architekturen unterstützen prädikatierte Instruktionen, wie z.B. die *ARM-Architektur*. Diese werden nur ausgeführt, wenn ihr Prädikat (*Guard*) erfüllt ist. Es gibt drei Fälle, je nachdem, was über den Wahrheitswert des *Guards* aus den Informationen der Werteanalyse bekannt ist. Hierfür wird aus der Transferfunktion $f$ ein $f^\circ$ definiert:

$$f^\circ(x) = \begin{cases} x & \text{Prädikat nicht erfüllt} \\ f(x) & \text{Prädikat erfüllt} \\ x \sqcup f(x) & \text{Wahrheitswert nicht bekannt} \end{cases}$$

## 6.4 Kombinationsfunktion

An Stellen, an denen der Kontrollfluss von verschiedenen Pfaden wieder zusammenfließt, muss eine *Kombinationsfunktion* angewendet werden. Von dieser werden die Datenflusswerte der eingehenden Kanten des jeweiligen Knoten so kombiniert, dass

man eine obere Schranke der eingehenden Datenflusswerte erhält. Die Kombination erfolgt jeweils paarweise.

Bei der Invariantenanalyse ist der Datenflusswert eine Funktion, die Variablen eine Menge von Ausdrücken zuordnet. Somit muss die Kombinationsfunktion über dieser Funktion definiert werden. Dazu wird zunächst eine Kombinationsfunktion für Ausdrucksmengen definiert.

**Definition 6.4.1 (Kombination von Ausdrucksmengen)**
Seien zwei Ausdrucksmengen $a$ und $b$ gegeben. Um die Kombinationsmenge $c$ dieser beiden Mengen zu berechnen, vereinigt man zunächst beide zu einer Menge $t$. Innerhalb von $t$ fasst man nun alle Elemente zusammen, welche dieselbe Variable enthalten, wobei die jeweiligen Intervalle zu einem neuen Intervall vereinigt werden.

Haben zu vereinigende Ausdrücke unterschiedliche Byte-Breiten, wird die kleinste Byte-Breite für den entstehenden neuen Ausdruck verwendet. Sind die Vorzeichenerweiterungen unterschiedlich, wird die Vorzeichenerweiterung des neuen Ausdrucks auf *unknown* gesetzt.

Diese Kombination von $a$ und $b$ sei im Folgenden als $a \sqcup_{comb} b$ bezeichnet. □

**Beispiel 6.4.1 (Kombination von Ausdrucksmengen)**
Seien folgende beiden Eingabemengen $a$ und $b$ gegeben:

$$\begin{aligned} a =& \{((Speicher, \mathrm{0xffff}, 4), [1,1], 4, notext), ((Register, 10, 4), [0,0], 4, notext), \\ & ((None, 0, 0), [1,1], 4, notext)\} \\ b =& \{((Register, 5, 4), [4,4], 4, notext), ((Register, 10, 4), [1,1], 2, ext), \\ & ((None, 0, 0), [5,5], 4, unknown)\} \end{aligned}$$

$a \sqcup_{comb} b$ ist dann:

$$\begin{aligned} &\{((Speicher, \mathrm{0xffff}, 4), [1,1], 4, notext), ((Register, 5, 4), [4,4], 4, notext), \\ &((Register, 10, 4), [0,1], 2, unknown), ((None, 0, 0), [1,5], 4, unknown)\} \end{aligned}$$

**Definition 6.4.2 (Kombinationsfunktion)**
Seien zwei zu kombinierende Datenflusswerte $f_1$ und $f_2$ gegeben. Der kombinierte Datenflusswert $f$ ist definiert als:

$$f(x) = \begin{cases} f_1(x) \sqcup_{comb} f_2(x) & f_1(x) \neq \emptyset \wedge f_2(x) \neq \emptyset \\ \emptyset & \text{sonst} \end{cases}$$ □

Durch die Anwendung dieser Kombinationsfunktion auf zwei eingehende Funktionen erhält man eine neue Funktion, die jeder Variable alle die Ausdrücke zuordnet, welche in beiden Eingaben möglich waren. Zuordnungen für Variablen, die nur in einer der beiden eingehenden Funktionen auf eine nichtleere Ausdrucksmenge abgebildet wurden, entfallen.

**Beispiel 6.4.2 (Kombinationsfunktion)**
Seien folgende beiden eingehende Datenflusswerte $a$ und $b$ gegeben:

$$\begin{aligned}
&a: \\
(\mathit{Register}, 5, 4) &\rightarrow \{((\mathit{Register}, 6, 4), [3, 5], 4, \mathit{notext})\} \\
(\mathit{Register}, 10, 4) &\rightarrow \{((\mathit{Register}, 10, 4), [0, 0], 4, \mathit{notext})\} \\
(\mathit{Speicher}, 0\text{xffff}, 4) &\rightarrow \{((\mathit{Speicher}, 0\text{xffff}, 4), [1, 1], 4, \mathit{notext})\}
\end{aligned}$$

$$\begin{aligned}
&b: \\
(\mathit{Register}, 10, 4) &\rightarrow \{((\mathit{Register}, 10, 4), [1, 1], 2, \mathit{ext}), \\
&\quad ((\mathit{Register}, 20, 4), [0, 0], 4, \mathit{ext})\} \\
(\mathit{Speicher}, 0\text{xffff}, 4) &\rightarrow \{((\mathit{Speicher}, 0\text{xffff}, 4), [4, 4], 4, \mathit{notext})\}
\end{aligned}$$

Als Kombination ergibt sich:

$$\begin{aligned}
(\mathit{Register}, 10, 4) &\rightarrow \{((\mathit{Register}, 10, 4), [0, 1], 2, \mathit{unknown}), \\
&\quad ((\mathit{Register}, 20, 4), [0, 0], 4, \mathit{ext})\} \\
(\mathit{Speicher}, 0\text{xffff}, 4) &\rightarrow \{((\mathit{Speicher}, 0\text{xffff}, 4), [1, 4], 4, \mathit{notext})\}
\end{aligned}$$

## 6.5 Widening

Die Menge der Variablen in *IVALA* ist für ein konkretes Eingabeprogramm beschränkt. Die auftretenden Gleichungsmengen sind daher ebenfalls beschränkt. Auch die jeweiligen Ausdrucksmengen der Gleichungen sind beschränkt, da sie maximal ein Element für jede mögliche Variable enthalten.

Die additive Konstante in den Ausdrücken kann ein beliebiges Intervall darstellen. Auch wenn die Transferfunktion monoton ist und die Kombinationsfunktion immer eine obere Schranke berechnet, kann sich das Intervall bei jeder Iteration verändern.

**Beispiel 6.5.1 (Problem der additiven Konstante)**
In Listing 6.1 auf der nächsten Seite ist ein C-Programm gegeben, das zwei geschachtelte Schleifen beinhaltet. Die Variable $j$ wird in der inneren Schleife verändert, ist jedoch auch ein potentieller Schleifenzähler der äußeren und ist somit im Datenflusswert als Zielvariable einer Gleichung enthalten. Jedoch wird nun diese innerhalb einer inneren Schleife verändert. Die Fixpunktiteration zur Lösung des Datenflussproblems wird somit nicht terminieren, da in jeder Iteration die additive Konstante in der Gleichung für $j$ um eins größer wird.

```
int i = 0, j = 0;

while (i < 1000)
{
  for (int m=0; m < something(); ++m)
    ++j;

  if (j > 100)
    break;

  ++i;
}
```

Listing 6.1: Widening Beispielprogramm

**Definition 6.5.1 (Verwendetes Widening)**
Seien $f_1$ und $f_2$ die Datenflusswerte zweier aufeinanderfolgenden Iterationen der Fixpunkiteration. Der vom Widening berechnete neue Datenflusswert $f$ ist definiert als:

$$f(x) = \begin{cases} f_2(x) & f_1(x) = \emptyset \lor f_1(x) = f_2(x) \\ \emptyset & \text{sonst} \end{cases}$$ □

**Beispiel 6.5.2 (Anwendung des Widening)**
Eingabe des Widenings seien die folgenden Datenflusswerte $f_1$ und $f_2$:

$$\begin{aligned} & f_1 : \\ (Register, 10, 4) & \rightarrow \{((Register, 10, 4), [0, 0], 4, notext)\} \\ (Speicher, \text{0xffff}, 4) & \rightarrow \{((Speicher, \text{0xffff}, 4), [1, 1], 4, notext)\} \end{aligned}$$

$$\begin{aligned} & f_2 : \\ (Register, 5, 4) & \rightarrow \{((Register, 5, 4), [1, 4], 4, notext)\} \\ (Register, 10, 4) & \rightarrow \{((Register, 10, 4), [0, 1], 4, notext)\} \end{aligned}$$

Das Ergebnis des Widenings ist:

$$(Register, 5, 4) \rightarrow \{((Register, 5, 4), [1, 4], 4, notext)\}$$

Die Gleichung für $R10$ wurde entfernt, weil sich deren Ausdrucksmenge seit der letzten Iteration geändert hat.

## 6.6 Spezifikation für PAG

Die Invariantenanalyse ist für *PAG* in *DATLA* und *FULA* spezifiert. Die komplette Spezifikation inklusive der architekturabhängigen Teile für die *PowerPC* Implementierung sind in Anhang A zu finden.

Kapitel
# 7 Implementierung

Zunächst wird in diesem Kapitel ein Überblick über die Gesamtimplementierung der Schleifenanalyse geliefert. Anschließend werden die Schnittstellen, die der Werteanalysator und die Schleifenanalyse bieten, dargestellt und die Interaktion zwischen beiden erklärt. Danach wird auf die Implementierung der Schleifenanalyse eingegangen, wobei zunächst der architekturunabhängige Teil vorgestellt wird und dann die abhängigen Teile aufgezählt werden. Zuletzt werden dann die Implementierungsdetails für die *PowerPC*-Architektur beschrieben.

## 7.1 Überblick

Die vorgestellte Analyse ist in Form einer C-Bibliothek implementiert. Der Werteanalysator *daan* linkt gegen diese und die Iteration der Analyse findet innerhalb des Werteanalysatorprozesses statt. Abbildung 7.1 zeigt eine Übersicht über die Struktur der Implementierung und deren Kooperation mit dem Werteanalysator. Die Bibliothek wird im Folgenden als *libloopy* bezeichnet.

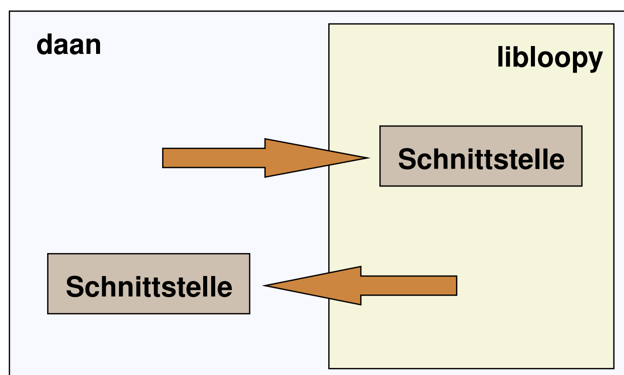



Abbildung 7.1: Struktur der Implementierung

## 7.2 Schnittstellen & Interaktion

Im Folgenden werden nun die beiden Schnittstellen beschrieben, jene des Werteanalysators und jene der Analysebibliothek. Dann wird auf die Integration in den *daan* und die eigentliche Implementierung der *libloopy* eingegangen.

### 7.2.1 daan Schnittstelle

Der Werteanalysator wird um eine Schnittstelle erweitert, die es erlaubt, die für die Schleifenanalyse notwendigen Daten zu erfragen. Hierbei werden die in Listing 7.1 deklarierten C-Funktionen im *daan* implementiert.

```
// Ist der Block/Instruktion nicht erreichbar?
extern int loopy_incoming_dfi_is_bottom (int block, int instruction, int context);

// Ist die Kante nicht ausführbar?
extern int loopy_outgoing_dfi_is_bottom (int block, int instruction, int context,
                                         int edge);

// Liefert Wert eines Registers
extern o_value loopy_value_register (int block, int instruction, int context, int reg);

// Liefert Wert einer Speicherzelle
extern o_value loopy_value_memory (int block, int instruction,
                                   int context, o_value cell, int width);

// Liefert Adressen der Speicherzugriffe
extern void loopy_access_memory (int block, int instruction, int context,
                                 o_value **raddr, int **rwidth,
                                 o_value **waddr, int **wwidth);
```

Listing 7.1: daan Schnittstelle

#### Semantik der Funktionen

- *loopy_incoming_dfi_is_bottom*
  Liefert für ein gegebenes Tripel aus Block, Instruktion und Kontext des Supergraphen zurück, ob der eingehende Datenflusswert $\perp$ ist.
- *loopy_outgoing_dfi_is_bottom*
  Liefert für ein gegebenes Quadrupel aus Block, Instruktion, Kontext und ausgehender Kante des Supergraphen zurück, ob der Datenflusswert der gegebenen Kante $\perp$ ist.
- *loopy_value_register*
  Wert des gegebenen Registers für das Tripel aus Block, Instruktion und Kontext des Supergraphen. Der Wert ist hierbei ein Intervall, das von der Werteanalyse im letzten Durchlauf der *Feedback-Iteration* berechnet wurde.

- *loopy_value_memory*
  Analog zu *loopy_value_register*, jedoch Wert der gegebenen Speicherzelle. Der Wert ist hierbei ein Intervall, das von der Werteanalyse im letzten Durchlauf der *Feedback-Iteration* berechnet wurde.
- *loopy_access_memory*
  Liefert für gegebenes Tripel aus Block, Instruktion und Kontext die lesend und schreibend zugegriffenen Speicherbereiche zurück, die vom *daan* bestimmt wurden.

Da diese Funktionen nur Schnittstellen zu schon in dem Werteanalysator *daan* vorhandenen Funktionen sind, wird nicht weiter auf die Art ihrer Implementierung eingegangen.

### 7.2.2 libloopy Schnittstelle

Analog zum *daan* stellt auch die *libloopy* einige Schnittstellenfunktionen bereit, die der *daan* im Rahmen der Analyse aufruft. Diese sind in Listing 7.2 dargestellt

```
// Initialisierung der Schleifenanalyse
int loopy_init (KFG cfg, int maxrounds);

// Startet einen Durchlauf der Analyse
int loopy_run ();

// Schreibt die berechneten Schleifengrenzen als Feedback für Werte-/Pfadanalyse in eine Datei
int loopy_write_pag (const char *filename);
```

Listing 7.2: libloopy Schnittstelle

#### Semantik der Funktionen

- *loopy_init*
  Diese Funktion initialisiert die *libloopy* Bibliothek. Sie muss einmal vor Analysebeginn aufgerufen werden. Der zu analysierende Kontrollflussgraph und die erlaubte Höchstzahl an Durchläufen für die *Feedback-Iteration* wird hierbei übergeben.
- *loopy_run*
  Führt eine Iteration der Schleifenanalyse aus. Gibt zurück, ob noch eine weitere Iteration nötig ist oder die erlaubte Höchstzahl von Iterationen schon erreicht ist.
- *loopy_write_pag*
  Schreibt die errechneten Schleifengrenzen der letzten Analyseiteration in eine

Datei. Diese wird als Feedback vom Werteanalysator vor der nächsten Analyse eingelesen.

Auf die Implementierung der Bibliothek, die hinter dieser Schnittstelle steht wird in Abschnitt 7.3 auf der nächsten Seite eingegangen.

### 7.2.3 daan & libloopy Interaktion

Die Schleifenanalyse ist in den *daan* integriert. Hierzu ist ein Auszug aus der Hauptfunktion des *daan* in Listing 7.3 gezeigt. Dieser Ausschnitt ist vereinfacht; es wurden Implementierungsdetails des Werteanalysators entfernt. Die Interaktion zwischen Werteanalyse und der Schleifenanalyse wird dabei herausgestellt.

```
int main (int argc, char *argv[])
{
  ...

  // Schleifenanalyse initialisieren
  loopy_init (cfg, maxrounds);

  ...

  // sind wir in der ersten Iteration?
  bool is_first_round = true;
  while (true)
  {
    // Schleifengrenzen einlesen, ab der zweiten Iteration
    if (!is_first_round)
      pag_init_vivu_4_loopmapping (out_filename);

    // Eine Iteration der Werteanalyse ausführen
    daan_value_analysis ();

    // Eine Iteration der Schleifenanalyse starten
    int res = loopy_run ();

    // Schleifengrenzen ausgeben
    loopy_write_pag (out_filename);

    // Müssen wir noch eine Runde machen?
    if (!res)
      break;

    is_first_round = false;
  }

  ...
}
```

Listing 7.3: Integration der Schleifenanalyse

Zu Beginn wird die Schleifenanalyse initialisiert. Danach wird iterativ immer zunächst eine Werteanalyse ausgeführt, welche die Schleifengrenzen der letzten Iteration als Eingabe erhalten hat. Dann erfolgt die Schleifenanalyse, gefolgt von der Ausgabe ihrer Ergebnisse. Abgebrochen wird diese Iteration erst, wenn entweder die

maximale Anzahl von Durchläufen erfolgt ist, die der Benutzer spezifiziert hat, oder die Schleifenanalyse für alle Schleifen und Kontexte die gleichen Ergebnisse wie in der letzten Iteration gefunden hat.

Die Interaktion von Seiten der Schleifenanalyse findet hauptsächlich während der Ausführung von *loopy_run* statt. Dabei beschränkt sich die Analyse auf die Verwendung der vorher vorgestellten Schnittstellenfunktionen des *daan*, die genügen, um alle Informationen zu erhalten, welche die Analyse benötigt.

## 7.3 libloopy Implementierung

Nach dem nun die Schnittstellen geklärt sind, wird auf die interne Implementierung der Analysebibliothek *libloopy* eingegangen. Zunächst ist der architekturunabhängige Teil von Interesse, der die Funktionsweise festlegt. Danach werden die architekturspezifischen Teile angesprochen. Deren Implementierung für die *PowerPC*-Architektur wird dann Thema des nächsten Abschnitts 7.4 auf der nächsten Seite sein.

### 7.3.1 Grundgerüst

Die Funktionsweise der Analyse wird durch die Implementierung der *loopy_run* Funktion vorgegeben. Diese ist in vereinfachter Form in Listing 7.4 dargestellt.

```
int loopy_run ()
{
  ...

  // Schleifen klassifizieren
  classifyLoops ();

  // Dominatoranalyse ausführen, für die Klassifizierung der Schleifentests
  loopy_dompdom_doit ();

  // Schleifen analysiern, Schichtenweise
  for (int nesting = 0; nesting <= maxNesting; ++nesting)
  {
    // potentielle Schleifenzähler einsammeln
    collectingVars ();

    // Datenflussanalyse ausführen
    loopy_dataflow_doit ();

    // Schleifengrenzen berechnen!
    calculateBounds ();
  }

  ...
}
```

Listing 7.4: loopy_run()

Die Implementierung geht nach den in Kapitel 5 beschriebenen Phasen vor, allerdings klassifiziert sie zunächst die Schleifen aller Schachtelungsebenen im voraus. Innerhalb der *Schachtelungstiefe-Iteration* erfolgt die Erkennung der Schleifenzähler, die Invariantenanalyse und das Bilden und Lösen der Gleichungen für die Schleifentests.

Auf die Implementierung der Datenflussanalyse wird hier nicht weiter eingegangen. Das zugrundeliegende Datenflussproblem wurde in Kapitel 6 vorgestellt. Die architekturspezifischen Teile werden für die *PowerPC*-Architektur in diesen Kapitel später beschrieben. Der eigentliche Analysator wird von PAG generiert und folgt den Charakteristiken, die in Kapitel 3 erläutert wurden.

### 7.3.2 Architekturabhängige Komponenten

Damit die Analyse mit geringem Aufwand auf andere Architekturen portiert werden kann, wurden die architekturspezifischen Teile in eine eigene Klasse ausgelagert, die man für die jeweilige Architektur implementieren muss. Diese Klasse zusammen mit Teilen der Transferfunktion sind die architekturabhängigen Teile.

Die Funktionen der *LoopyArch* Klasse abstrahieren die Bezeichnungen der Instruktionen der jeweiligen Architektur, die in der Invariantenanalyse und der Behandlung der Schleifentests verwendet werden. Die Definition dieser Klasse ist in Listing 7.5 auf der nächsten Seite gegeben.

Desweiteren wird die Registeranzahl festgelegt und ein Mapping zwischen Textbezeichnung von Registern und ihrer Nummer vorgenommen. Auch wird die Registergröße in Bytes angegeben, dies für jedes Register einzeln, um Architekturen mit verschieden großen Registern und Teilregistern zu erlauben. Die Funktion *registerKillsRegister* erlaubt zu überprüfen, ob das Schreiben in ein Register *a* Einfluss auf ein Register *b* hat, man also auch Informationen über dessen Inhalt vernichten muss Auch dies dient der Unterstützung von Teilregistern.

Die Funktion *instructionExecuted* dient zur Überprüfung, ob eine Instruktion im Supergraph immer, manchmal oder nie ausgeführt wird. Dies erfolgt durch einer Auswertung des jeweiligen *Guards* der Instruktion, falls die Architektur solche erlaubt.

Zuletzt gibt es die Funktion *calculateBound*, die für eine gegebene Schleife und ihren äußeren Aufrufkontext die Schleifengrenze berechnet.

## 7.4 PowerPC Architektur

Die *PowerPC*-Version der Schleifenanalyse wurde erstellt durch das Implementieren der *LoopyArch*-Klasse für diese Architektur und der architekturspezifischen Teile des

```
class LoopyArch
{
  public:
    // Anzahl der Instruktionsnamen
    static int gennames ();

    // Array mit den Instruktionsnamen
    static char const ** gennameData ();

    // Index des ungültigen Instruktionsnamen
    static int gennameIllegal ();

    // Anzahl der Register
    // Alle Register werden von 0 bis Anzahl - 1 durchnummeriert
    static int registers ();

    // Umwandlung von Registernamen zu Registernummer
    // Gibt -1 bei ungültigem Namen zurück
    static int registerNumber (const char *regname);

    // Größe des gegebenen Registers in Byte
    static int registerSize (int regNumber);

    // Wenn man den Inhalt von Register a verändert,
    // verändert sich auch der Inhalt von Register b?
    // Klar, wenn a == b, sonst nur wenn b Teilregister oder Elternregister von a
    static bool registerKillsRegister (int a, int b);

    // Wird die Instruktion im gegebenen Kontext ausgeführt?
    // 2 == immer, 1 == manchmal, 0 == nie
    static int instructionExecuted (KFG cfg, KFG_NODE block, int instruction, int context);

    // Berechnet die Schleifengrenzen für den gegebenen Aufruf und äußeren Kontext
    static void calculateBound (LoopyLoopContext *c, int ctx, LoopyLoopCall *call,
                                int &min, int &max, const LoopyVectorVar &vars,
                                const LoopyVectorInt &innerContexts);
};
```

Listing 7.5: LoopyArch

Datenflussproblems. Zunächst wird auszugsweise die *LoopyArch*-Klasse für *PowerPC* betrachtet, danach auf die Anpassungen im Datenflussproblem eingegangen.

### 7.4.1 LoopyArch

Die Implementierung der Klasse *LoopyArch* für *PowerPC* hat von der Anzahl der Quellcodezeilen ihren Schwerpunkt in der Auflistung aller Instruktionsnamen der Architektur und in der Berechnung der Schleifengrenzen. Die Registerbehandlung ist nicht komplex, da diese Architektur keine Teilregister unterstützt, ebenso wie die Behandlung von *Guards*, welche die *PowerPC* Architektur nicht implementiert.

Die Aufzählung der Instruktionsnamen für die spätere Verwendung in der Behandlung der Schleifentests und dem Datenflussproblem ist ein C-Array von C-Strings der Namen. Durch Zeichenkettenvergleich wird die jeweilige Instruktion aus der CRL-Beschreibung eindeutig einer Ganzzahl zugeordnet. Dies kann man sich wie in Listing 7.6 vorstellen.

```
static char const *genname_data[] =
  {
    "illegal",
    "twi",
    "mulli",
    "subfic",
    "cmpli",
    "cmpi",
    "addic",
    "addicD",
    "addi",
    "addis",
    ...
```

Listing 7.6: PowerPC Instruktionsnamen

Die für die untersuchten Integeroperationen verwendeten 32 Register sowie das Link- und Counter-Register sind alle 4 Bytes breit und disjunkt, es gibt keine Teilregister. Somit ist die Abbildung auf die Registernummern 0 bis 33 sehr einfach gehalten. Listing 7.7 auf der nächsten Seite zeigt den zugehörigen Auszug aus der Implementierung.

Die *calculateBound*-Funktion iteriert über alle Schleifentests der Schleife, mittels dem in Kapitel 5 beschriebenen Verfahren die Grenzen des jeweiligen Schleifentests berechnet und dann die Grenzen aller Tests kombiniert. Diese Funktion wird für jede Schleife für jeden äußeren Schleifenkontext getrennt aufgerufen, damit man die gewünschten Informationen auf Kontextbasis erhält. Die Implementierung folgt der Beschreibung im oben genannten Kapitel, passt diese jedoch den Gegebenheiten der Architektur an, also den jeweiligen Instruktionsnamen für die bedingten Sprünge, deren Argumenten, der Semantik der Instruktionen, welche die Condition-Register manipulieren und sonstigem.

```
int LoopyArch::registers ()
{
  return 34;
}

int LoopyArch::registerNumber (const char *regname)
{
  if (strlen(regname) < 1)
    return -1;

  if (strcmp (regname, "lr") == 0)
    return 32;

  if (strcmp (regname, "ctr") == 0)
    return 33;

  if (regname[0] != 'r')
    return -1;

  int reg =  atoi (regname+1);

  if (reg > 31)
    return -1;

  return reg;
}

int LoopyArch::registerSize (int)
{
  return 4;
}

bool LoopyArch::registerKillsRegister (int a, int b)
{
  return a == b;
}
```

Listing 7.7: PowerPC Registerbehandlung

### 7.4.2 Invariantenanalyse

Die Anpassungen im Invariantenanalyse geschehen an zwei Stellen: in der DATLA- und in der FULA-Beschreibung, die PAG erhält. Zusätzlich zu den architekturunabhängigen Teilen werden einzelne Elemente und Funktionen hier architekturspezifisch definiert. Diese Teile werden automatisch zusammengefügt und PAG übergeben, um den Analysator zu generieren. Die unabhängigen Teile folgen der Beschreibung in Kapitel 6.

#### DATLA Anpassungen

In der DATLA-Spezifikation wird eine Aufzählung für die jeweiligen Instruktionsnamen der Architektur angelegt, die zu der Liste in der jeweiligen *LoopyArch*-Klasse passt. Dies zeigt Listing 7.8 auf der nächsten Seite ausschnittsweise.

```
genname_enum = (
  ppc__illegal,
  ppc__twi,
  ppc__mulli,
  ppc__subfic,
  ppc__cmpli,
  ppc__cmpi,
  ppc__addic,
  ppc__addicD,
  ppc__addi,
  ppc__addis,
  ...
```

Listing 7.8: PowerPC Instruktions-Enumeration

### FULA Anpassungen

In FULA werden folgenden drei Funktionen definiert:

- *callee_saved_reg :: snum -> bool*
  Diese Funktion gibt an, ob eine gegebene Registernummer zu der Menge von Registern gehört, die nach der Aufrufkonvention der Architektur von der aufgerufenen Funktion gesichert und wieder hergestellt werden müssen.
- *destroy_on_reg_write :: equations,genname_enum, ... -> equations*
  Diese Funktion hat die Aufgabe, alle Gleichungen aus der gegebenen Gleichungsmenge zu entfernen, die Register beschreiben, deren Inhalt von der gegebenen Instruktion geschrieben wird. Dies erlaubt die Berücksichtigung von *Load-Multiple*-Instruktionen.
- *new_equations :: equations,genname_enum, ... -> equations*
  Diese Funktion ist das eigentliche Herz der Datenflussanalyse. Für eine gegebene Instruktion und eingehende Formelmenge errechnet sie, welche neuen Formeln man nach dieser Instruktion erhält.

In Listing 7.9 auf der nächsten Seite wird auszugsweise die Implementierung dieser Funktionen in FULA für die *PowerPC*-Architektur gezeigt. Die vollständige Spezifikation ist im Anhang A zu finden.

```
// PPC SPECIFIC PART OF THE CALLING CONVENTIONS
callee_saved_reg (reg) = (13 <= reg) && (reg <= 31);

// destroy vars for registers we write to
// load multiple
destroy_on_reg_write (x : eq, ppc__lmw, dsts, srcs)
    = if ((x!1!2)!1!3 = v_reg && ((name_to_var (dsts!1!8))!2!3 <= (x!1!2)!2!3)
           && ((x!1!2)!2!3 <= 31))
      then destroy_on_reg_write (eq, ppc__lmw, dsts, srcs)
      else x : destroy_on_reg_write (eq, ppc__lmw, dsts, srcs) endif;

// normal other stuff
destroy_on_reg_write (x : eq, gn, dsts, srcs)
    = if (  name_to_var (dsts!1!8) = x!1!2 || name_to_var (dsts!2!8) = x!1!2
         || name_to_var (dsts!3!8) = x!1!2 || name_to_var (dsts!4!8) = x!1!2
         || name_to_var (dsts!5!8) = x!1!2 || name_to_var (dsts!6!8) = x!1!2
         || name_to_var (dsts!7!8) = x!1!2 || name_to_var (dsts!8!8) = x!1!2
         )
      then destroy_on_reg_write (eq, gn, dsts, srcs)
      else x : destroy_on_reg_write (eq, gn, dsts, srcs) endif;

// here comes the interesting stuff ;)
// insert new equations for the given genname ;)
new_equations :: equations,genname_enum,str_8,str_8,imacc,snum,snum,snum -> equations;

// addi dst, src, const
// do overflow check
new_equations (eq, ppc__addi, dsts, srcs, mem, block, instruction, context)
 = addi_helper (eq, dsts, srcs, 0);

// ppc__bc_DCtr (decrement the ctr register!!)
// do overflow check
new_equations (eq, ppc__bc_DCtr, dsts, srcs, mem, block, instruction, context)
 = let dvar = name_to_var(dsts!4!8); // destination register
       incr = -1; // increment
       values = find_values (eq, name_to_var(srcs!4!8)); // values to increment
   in if (dvar!1!3 = v_invalid || values = [])
        then [] // nothing to do, we don't know the source stuff
        else if (add_to_values (values, incr) = []) then []
               else [(dvar, add_to_values (values, incr))] endif
      endif;

...
```

Listing 7.9: PowerPC FULA Anpassungen

Kapitel

# 8 Theoretische Diskussion

Nachdem die vorhergehenden Kapitel ausführlich das Design und die Funktionsweise der Schleifenanalyse dargelegt haben, diskutiert dieses Kapitel nun ihre theoretischen Möglichkeiten und liefert einen Vergleich mit anderen Analysen in diesem Gebiet.

## 8.1 Fähigkeiten und Beschränkungen

Ziel der vorgestellten Analyse ist die Berechnung sicherer Grenzen für Zählschleifen, die am weitesten verbreitete Art von Schleifenkonstrukten.

Die Invariantenanalyse berechnet Schleifeninvarianten und diese werden dann benutzt, um eine geschlossene Formel für die Anzahl der möglichen rekursiven Aufrufe der Schleifenprozedur zu erhalten, wie sie die einzelnen Abbruchbedingungen der Schleife zulassen. Es wurde weiter gezeigt, das für jeden Schleifentest eine Gleichung einer der folgenden Formen verwendet wird:

$$(c_0 + (\text{reccalls} * c_2) + c_1) = c \text{ und } k_1 = c \text{ und } (k_2 + c_1) = c$$
$$(c_0 + (\text{reccalls} * c_2) + c_1) \neq c \text{ und } k_1 \neq c \text{ und } (k_2 + c_1) \neq c$$
$$(c_0 + (\text{reccalls} * c_2) + c_1) < c \text{ und } k_1 < c \text{ und } (k_2 + c_1) < c$$
$$(c_0 + (\text{reccalls} * c_2) + c_1) > c \text{ und } k_1 > c \text{ und } (k_2 + c_1) > c$$
$$(c_0 + (\text{reccalls} * c_2) + c_1) \leq c \text{ und } k_1 \leq c \text{ und } (k_2 + c_1) \leq c$$
$$(c_0 + (\text{reccalls} * c_2) + c_1) \geq c \text{ und } k_1 \geq c \text{ und } (k_2 + c_1) \geq c$$

Nun stellt sich die Frage, welche Arten von Zählschleifen damit wirklich abgedeckt werden können und für welche Konstrukte dieses Verfahren bzw. diese Gleichungen nicht genügen.

Die Struktur der Gleichungen erlaubt die Erkennung von Grenzen für Zählschleifen, die folgenden Bedingungen genügen:

1. Das Startintervall $c_0$ muss durch die Werteanalyse bekannt sein.

2. Das Intervall $c$, das als Vergleichselement in der Abbruchbedingung dient, muss durch die Werteanalyse bekannt sein.

3. Der Schleifenzähler darf nur durch Addition von beschränkten Intervallen $c_1$ und $c_2$ in einer Iteration verändert werden. $c_2$ beschreibt hierbei die Veränderung über eine ganze Iteration hinweg, $c_1$ nur bis zur Prüfung der Abbruchbedingung. Zusätzlich ist noch die Zuweisung einer Konstanten zum Schleifenzähler erlaubt.

4. Die Abbruchbedingung muss ein einfacher Vergleich mit einem der folgenden Modi sein:

$$=, \neq, <, >, \leq, \geq$$

Jede dieser Bedingungen führt natürlich zu einer damit einhergehenden Einschränkung für die Erkennung von Grenzen bei speziellen Schleifenkonstrukten. Diese seien nun im Folgenden aufgeführt:

1. *Unbekannter Startwert*
   Kann die Werteanalyse kein beschränktes Startintervall bestimmen, kann keine weitere Berechnung erfolgen. Dies kann zum Beispiel passieren, wenn die Länge der *Call-Strings* zu klein gewählt wird, so dass die Aufrufe von Prozeduren mit Schleifen gemischt werden und die Startwerte der Schleifen von den Parametern abhängen.

2. *Unbekannter Endwert*
   Wie bei dem Startwert, führt auch ein unbekannter Endwert zu keinem Analyseergebnis. Da die Analyse von einem konstanten Endwert ausgeht, können somit nur Schleifen analysiert werden, deren Abbruchbedingung einen Vergleich mit einer Konstanten darstellt. Hierbei heißt konstant, dass die Werteanalyse ein beschränktes Intervall liefert, das für alle Iterationen gilt.

   Es ist keine Analyse von Schleifen möglich, deren Abbruch mit einem innerhalb der Schleife sich verändernden Endwert verknüpft ist. Listing 8.1 liefert hierfür ein einfaches Beispiel.

```
int i = 0, j = 10;

while (i < j)
{
  i += 2;
  j++;
}
```

Listing 8.1: Schleife mit nicht-konstantem Endwert

3. *Nicht unterstützte Veränderungen des Schleifenzählers*
   Wird der Schleifenzähler während einer Iteration nicht nur durch Addition eines konstanten Intervalls bzw. Zuweisung einer Konstanten verändert, so schlägt die Berechnung von verwendbaren Schleifeninvarianten fehl.

   Eine Zählschleife für die man durch diese Beschränkung keine Grenzen erkennen könnte ist in Listing 8.2 gegeben. Die Variable in der Abbruchbedingung hängt von 3 verschiedenen Schleifenzählern ab, solche Konstrukte kommen jedoch in der Praxis nur selten vor.

```
int a = 0, b = 0, c = 0, i = 0;

while (i < 10)
{
  a += 1;
  b += 2;
  c += 3;

  if (something())
    i = a;
  else if (something2())
    i = b;
  else
    i = c;
}
```

Listing 8.2: Nicht analysierbare Schleife

Die Erkennung von klassischen Zählschleifen wird durch diese Einschränkung nicht behindert, da diese den Schleifenzähler nur einmal pro Iteration um eine Konstante erhöhen. Auch sind Grenzen für Schleifen erkennbar, die zwar den Schleifenzähler an mehreren Stellen im Kontrollfluss während einer Iteration verändern, jedoch jedes mal nur um eine Konstante, womit man in der Datenflussanalyse wieder das Intervall dieser Veränderungen berechnen kann.

Durch die Möglichkeit, auch die Zuweisung von Konstanten zu behandeln, sind ebenfalls Schleifen behandelbar, die z.B. eine innere Schleifen enthalten, welche durch Zuweisung einer Konstante die äußere Schleife zum Abbruch zwingt, wie Listing 8.3 auf der nächsten Seite zeigt. Für diese Schleifen wird dann allerdings die sichere, aber eventuell ungenaue untere Schleifengrenze von eins berechnet. Auch wird dieser vorzeitige Abbruch bei der oberen Schleifengrenze nicht berücksichtigt, diese bleibt damit zwar konservativ, ist aber möglicherweise zu groß.

Nicht erkennbar sind Grenzen für Schleifen, die ihren Schleifenzähler durch

```
for (int i=0; i < 1000; ++i)
{
  for (int j=0; j < 100; ++j)
  {
    // brich beide Schleifen ab, falls Ergebnis korrekt
    if (richtigesErgebnis (i, j))
    {
      i = 1000;
      break;
    }
  }
}
```

Listing 8.3: Innere Schleife bricht äußere ab

Manipulationen jenseits von Addition und Subtraktion manipulieren, also z.B. durch Multiplikation. Auch muss das Intervall, das zu dem Schleifenzähler addiert wird, beschränkt sein. Daher sind z.B. die Grenzen der Schleifen aus Listing 8.4 nicht erkennbar.

```
// multiplikative Schleife
for (int i=1; i < 10; i=2*i)
  doSomething ();

// i wird in innerer Schleife modifiziert
for (int i=0; i < 1000; ++i)
{
  for (int j=0; j < something(); ++j)
    i++;
}
```

Listing 8.4: Nicht-erkennbare Schleifen

4. *Form der Abbruchbedingung*
   Die Gleichungen erlauben nur Abbruchbedingungen, die einen einfachen Vergleich mit einer Konstanten darstellen. Da die meisten Compiler jedoch auch für kompliziertere Abbruchbedingungen (z.B. $i > 10 \| j > 20$) mehrere Schleifentests generieren, die jeweils nur eine einzelne Bedingung erhalten (hier also jeweils $i > 10$ und $j > 20$), können auch diese erkannt werden. Dieses Vorgehen wird von dem Compiler durch die *Shortcut-Evaluation* verlangt, die viele Programmiersprachen vorschreiben. Dies beinhaltet auch die im Bereich der eingebetteten System so verbreiteten Sprachen *C* und *C++*.

Wie weit diese theoretischen Beschränkungen Auswirkungen auf die praktische An-

wendbarkeit der Analyse haben, wird die praktische Evaluation im folgenden Kapitel zeigen.

## 8.2 Verwandte Arbeiten

Die Berechnung von Schleifengrenzen ist kein neues Gebiet, jedoch gibt es kaum publizierte Verfahren, die auch in der Praxis anwendbar sind. Diese Arbeit stellt meines Wissens das erste publizierte Verfahren da, das auch der Praxis genügt und erfolgreich implementiert wurde.

Dennoch gibt es eine Anzahl von Verfahren, die in der Praxis erfolgreich angewendet werden, wenn auch ihre Interna nie wirklich veröffentlicht wurden. Hierzu zählt sowohl die alte Implementierung der Schleifenanalyse des WCET-Analysators *aiT* als auch jene, die in dem *aiT* ähnlichen *Bound-T* ihre Anwendung findet.

Eine weitere interessante und vor allem auch implementierte Arbeit stammt von Sjödin, Healy und Whalley [SHW98]. Auch ihre Analyse arbeitet auf Ebene des Maschinencodes.

### 8.2.1 AbsInt aiT

Der WCET-Analysator *aiT* besitzt schon seit längerem eine eingebaute Schleifenanalyse. Sie ist analog zu der neu vorgestellten Analyse in die WCET-Berechnung eingebunden, was in Kapitel 4 beschrieben wurde.

Diese Schleifenanalyse basiert auf *Pattern-Matching*. Für die am meisten verwendeten Schleifenkonstrukte, die von in der Industrie verwendeten Compilern generiert werden, ist jeweils ein *Pattern* eingebaut, das auf den erzeugten Kontrollfluss passt. Diese Vorgehensweise ist konstruktionsbedingt sehr abhängig von der verwendeten Compilerversion, mit der die zu analysierenden Programme erstellt wurden. Außerdem ist es aufwändig Pattern zu entwerfen, die auch für Schleifen mit mehreren Inkrements pro Schleifeniteration bzw. verschiedenen Inkrements auf unterschiedlichen Pfaden innerhalb einer Schleifeniteration zurechtkommen.

Die praktische Verwendbarkeit dieser Art von Analyse hat ihr erfolgreicher jahrelanger Einsatz in dem kommerziell verwendeten Analysator *aiT* unter Beweis gestellt. Jedoch ist der Aufwand, diese Analyse an neue Compiler anzupassen, verhältnismäßig hoch.

### 8.2.2 Tidorum Bound-T

*Bound-T* ist ebenfalls ein kommerzieller WCET-Analysator, der eine eingebaute Schleifenanalyse besitzt. Aus den Veröffentlichungen über den Analysator, wie z.B. [HLS00] oder [HS02], erhält man einige Informationen über die Funktionsweise der verwendeten Analyse.

Auch diese Schleifenanalyse beschränkt sich, wie die der vorliegenden Arbeit, auf Zählschleifen. Es werden die Schleifenzähler der jeweiligen Schleifen ermittelt und deren Veränderungen innerhalb der Schleifeniterationen durch Formeln abstrahiert. Auf Basis dieser Formeln werden dann die Abbruchbedingungen evaluiert und somit die Grenzen berechnet. Die Formeln sind hierbei in der Presburger-Arithmetik formuliert und zur Lösung wird das schon vorher erwähnte Omega-System verwendet. Zur Konstruktion dieser Formeln wird jedoch nur der Schleifenrumpf untersucht. Es erfolgt keine intraprozedurale Analyse der dort aufgerufenen Prozeduren. Weitere Details, wie z.B. die Behandlung von Überläufen oder die Performance sind nicht öffentlich bekannt.

### 8.2.3 Sjödin, Healy und Whalley

Diese Analyse wird in der Veröffentlichung [SHW98] dargestellt. Sie arbeitet ebenfalls auf Maschinencode, als Architektur wurde die *SPARC*-Architektur gewählt.

Das beschriebene Verfahren kann, wie die hier vorgestellte Analyse, mit Schleifen mit mehreren Ausgängen umgehen. Es werden ähnlich zu der Schleifenanalyse dieser Arbeit ebenfalls Formeln für die einzelnen Abbruchbedingungen aufgestellt und deren Lösungen dann zu einer gemeinsamen Grenze kombiniert. Allerdings wird verlangt, dass nur exakte Inkrements/Dekrements innerhalb einer Schleifeniteration erfolgen.

Es ist keine Werteanalyse integriert und es erfolgt keine interprozedurale Analyse. Werden Funktionen innerhalb der Schleife aufgerufen, kann die Analyse somit keine Aussagen mehr über die Veränderung des Schleifenzählers treffen, sieht man von durch die Calling-Konvention geschützten Registern ab. Ist die Abbruchbedingung nicht an eine Konstante im Maschinencode gebunden, so muss der Benutzer manuell die Start- und Endwerte eingeben.

### 8.2.4 Sonstige Arbeiten

Von oben erwähnten auch praktisch eingesetzten bzw. zumindest implementierten Verfahren abgesehen, sind die meisten anderen publizierten Arbeiten rein theoretischer Natur.

Die in diesen Arbeiten vorgestellten Analysen beziehen sich meist nicht auf die von dieser Arbeit untersuchte Maschinencode-Ebene. Die meisten untersuchen das Problem der Ermittelung von Schleifengrenzen auf Ebene des Quellcodes einer realen Programmiersprache bzw. auf Ebene einer speziell für dieses Problem entworfenen Spezifikationssprache. Somit arbeiten sie auf einem anderen Detailierungsgrad, was ihre Anwendung in der WCET-Bestimmung erschweren kann.

Eine Arbeit von Christer Sandberg beschäftigt sich mit einer syntaktischen Form der Analyse von Schleifen auf Ebene des C-Quellcodes von Programmen [San04].

In [Pug94] wird unter Verwendung der Presburger-Arithmetik die Zahl von Schleifendurchläufen für eine mathematische Darstellung von Schleifen berechnet.

# Kapitel 9 Praktische Evaluation

Nach der theoretischen Diskussion in Kapitel 8 werden nun die praktischen Fähigkeiten der neuen Schleifenanalyse anhand von Testbeispielen evaluiert. Es wird ein Vergleich mit der schon existierenden Schleifengrenzenberechnung von *aiT* erfolgen und die Anwendbarkeit auf reale Programme aus der Industrie gezeigt werden.

## 9.1 Evaluationsverfahren

Die Evaluation der entwickelten Analyse erfolgt unter Verwendung der Implementierung für den PowerPC. Diese ermöglicht die Untersuchung von Programmen für die Prozessortypen MPC555/565 und MPC755. Diese spielen in der heutigen Zeit sowohl in der Automobilindustrie als auch in der Luftfahrt eine wichtige Rolle. Der MPC755 ist z.B. in der Flugkontrolle des gerade in der Entwicklung befindlichen *A380* von *Airbus* im Einsatz.

Als Kontrahent im direkten Vergleich zu der neuen Analyse dient eine *aiT-Version* vom Januar 2006. Auch diese befindet sich in der Industrie erfolgreich in Verwendung. Diese *aiT-Version* wurde so angepasst, dass man zur Laufzeit zwischen beiden Analysen umschalten kann. Hierbei wird nur die Schleifenanalyse ersetzt, es erfolgen keine sonstigen Änderungen. Dadurch ist die direkte Vergleichbarkeit der Ergebnisse gewährleistet.

Als Ergebnisse werden jeweils tabellarisch die Anzahl bzw. Werte der erkannten Schleifengrenzen der beiden Analysen gegenübergestellt sowie die in den jeweiligen Läufen ermittelte WCET in Zyklen. Somit erhält man sowohl einen Vergleich der Schleifenanalysen an sich als auch deren Einfluss auf die Genauigkeit der ermittelten WCET. Zusätzlich erfolgt bei den größeren Testbeispielen ein Vergleich der Laufzeiten der beiden Schleifenanalyseverfahren.

## 9.2 Gewählte Testbeispiele

Zum Einsatz kommen zwei Arten von Testbeispielen. Auf der einen Seite werden Testprogramme betrachtet, die zur Qualitätskontrolle von *aiT* und dessen Schleifenanalyse im Entwicklungsprozess von *AbsInt* eingesetzt werden. Zum anderen werden Teile von industrieller Software untersucht, wie sie z.B. in der Automobil- oder Flugzeugbranche vorkommen.

Um die Abhängigkeit von einem bestimmten Compiler zu untersuchen, werden die Testprogramme in zwei Varianten vorkommen. Zum einen werden Kompilate eines in der Industrie eingesetzten *DiabData Compilers* analysiert, zum anderen Kompilate des freien *GNU C Compilers*.

Die Pattern der bisherigen Schleifenanalyse wurden auf den *DiabData Compiler* optimiert. Daher kann dieser Vergleich zeigen wie sehr (oder auch nicht) diese Pattern damit wirklich auf einen Compiler beschränkt sind. Auch wird dies Aufschluss geben, ob die neue Analysemethode tatsächlich über Compilergrenzen hinweg funktioniert.

## 9.3 Messergebnisse

Im Folgenden werden nun die Messergebnisse der untersuchten Testprogramme aufgelistet. Neben diesen reinen Rohdaten wird auch eine Wertung der Resultate erfolgen, sowohl über das Verhalten der neuen Analyse für sich als auch im Vergleich zu der schon existierenden Analyse.

Die Tabellen mit den Ergebnissen stellen jeweils der Anzahl der in dem Programm enthaltenen Schleifen die Anzahl der Schleifen entgegen, für die sichere Schranken erkannt wurden. Hierbei unterscheidet man zwischen den Resultaten der neuen Analyse (*loopy*) und der schon vorhandenen Analyse (*aiT*). Die WCET wird in Zyklen angegeben, die bei einer üblichen Konfiguration der untersuchten Architektur auftreten. Auch wird zwischen den Zeiten für die *loopy*- und *aiT*-Resultate der Schleifenanalyse unterschieden.

Bei der Untersuchung der Schleifentests werden statt der Anzahl die Sollschranken sowie die erkannten Schranken aufgelistet, da diese Programme je nur eine Schleife enthalten und man so besser die Qualität der Analyse beurteilen kann. Die WCET wird bei diesen Tests in der Tabelle ausgelassen, da sie nur von der einen gezeigten Schleifengrenze abhängt.

Für die Industrieprogramme werden in einer weiteren Tabelle die Laufzeiten der beiden Analyseverfahren gegenübergestellt. Als Rechner für diese Messungen diente ein 3.2 GHz Pentium 4 mit 2 GB RAM unter *Linux*.

### 9.3.1 Interne Testprogramme

An internen Testprogrammen werden Programme, die einer Compilertestsuite entstammen, und synthetische Schleifentests unterschieden. Beide werden in der Qualitätssicherung von *aiT* eingesetzt.

Bei den Compilertests handelt es sich um eine Mischung verschiedener Typen von Programmen, die viele mögliche C-Sprachkonstrukte abdecken, und so auch eine Vielzahl von verschiedenen Schleifenkonstrukten enthalten. Einige sind auch klassiche Compilerbenchmarks, wie etwa EDN oder Dhrystone. Hier werden bei den verschiedenen Compilern teils unterschiedliche Tests verwendet, da nicht alle von beiden Compilern übersetzt werden konnten.

Die Schleifentests hingegen sind generiert. Sie sind in die drei in C vorhandenen Schleifenkonstrukte *do*, *for* und *while* unterteilt und hierbei für die drei Datentypen *char*, *int* und *long* ausgelegt. Den jeweils verwendeten Schleifentyp und Datentyp erkennt man am Namen des jeweiligen Tests. Da *int* und *long* auf der verwendeten PowerPC-Architektur gleich sind (je 32 Bit breit), werden bei der Messung nur die Tests für *char* und *int* untersucht. Einen Schwerpunkt setzen die Schleifentests bei den sehr häufig verwendeten *for*-Schleifen. Diesen Schwerpunkt spiegelt auch die Auswahl der hier verwendeten Tests wieder.

Die Testprogramme werden in zwei Teilen betrachtet: zunächst die Versionen, die mit einem *DiabData Compiler* erstellt wurden, der auch in der Industrie zum Einsatz kommt, dann die Erzeugnisse des *GNU C Compilers* der Version 3.3. Als Zielplatform für diese Testprogramme wurde der *MPC755* gewählt, die Ergebnisse der Schleifenanalyse sind allerdings auch auf die anderen *PowerPC*-Derivate übertragbar.

#### DiabData Kompilate

In Tabelle 9.1 werden die Ergebnisse der Compilerbenchmarks aufgeführt, unter Angabe der Zahl der korrekt erkannten Schleifen. Tabelle 9.2 gibt Auskunft über die berechneten Schranken für die Schleifentests.

| Test | Schl. | Grenzen (aiT) | Grenzen (loopy) | WCET (aiT) | WCET (loopy) |
|---|---|---|---|---|---|
| coverc | 3 | 3 | 3 | 8312 | 8312 |
| edn | 14 | 14 | 14 | 272957 | 272957 |
| loop3 | 120 | 120 | 120 | 3784 | 3784 |

Tabelle 9.1: Compilerbenchmarks (DiabData)

| Test | Soll | Ist (aiT) | Ist (loopy) |
|---|---|---|---|
| do_char_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| do_char_008 | [16] | [16] | [16] |
| do_char_009 | [16] | [16] | $[1,\infty]$ |
| do_char_010 | [1,16] | [1,16] | $[1,\infty]$ |
| do_int_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| do_int_008 | [16] | [16] | [16] |
| do_int_009 | [16] | [16] | [16] |
| do_int_010 | [1,16] | [1,16] | [1,16] |
| for_char_001 | [17] | $[1,\infty]$ | [17] |
| for_char_017 | [17] | [17] | [17] |
| for_char_049 | [1] | [1,17] | [1,17] |
| for_char_058 | [17] | $[1,\infty]$ | [17] |
| for_char_061 | [9] | $[1,\infty]$ | [9] |
| for_char_062 | [17] | $[1,\infty]$ | [17] |
| for_int_001 | [17] | $[1,\infty]$ | [17] |
| for_int_017 | [17] | [17] | [17] |
| for_int_049 | [1] | [1,17] | [1,17] |
| for_int_058 | [17] | $[1,\infty]$ | [17] |
| for_int_061 | [9] | $[1,\infty]$ | [9] |
| for_int_062 | [17] | $[1,\infty]$ | [17] |
| while_char_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| while_char_008 | [17] | [17] | [17] |
| while_char_009 | [17] | $[1,\infty]$ | $[1,\infty]$ |
| while_char_010 | [1,17] | $[1,\infty]$ | $[1,\infty]$ |
| while_int_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| while_int_008 | [17] | [17] | [17] |
| while_int_009 | [17] | [17] | [17] |
| while_int_010 | [1,17] | [1,17] | [1,17] |

Tabelle 9.2: Synthetische Schleifentests (DiabData)

Wie die Zahlen erkennen lassen, liegt die neue Schleifenanalyse *loopy* bei den Compilerbenchmarks gleich auf mit der etablierten Analyse von *aiT*. Bei den Schleifentests zeigt die neue Analyse kleine Schwächen bei einigen *char*-Schleifen, die darauf beruhen, dass hier die Behandlung eines Spezialfalls in den Pattern für bessere Ergebnisse sorgt. In den nicht erkannten Beispielen addiert der Compiler in der Schleife 255 auf eine *char*-Variable, um sie um eins zu dekrementieren. Dies führt in der neuen Analyse zu der Annahme, es gäbe einen Überlauf, während die alte Analyse exakt diesen Fall getrennt behandelt. Bei der Behandlung der *for*-Schleifen hingegen zeigt sich die Flexibilität des neuen Verfahrens, da hier zahlreiche Beispiele nicht von den bisher implementierten Pattern abgedeckt werden.

### GNU Kompilate

Die Ergebnisse der Messungen für die Compilerbenchmarks sind in der Tabelle 9.3 vermerkt, die der Schleifentests in Tabelle 9.3.

| Test | Schl. | Grenzen (aiT) | Grenzen (loopy) | WCET (aiT) | WCET (loopy) |
|---|---|---|---|---|---|
| loop3 | 120 | 0 | 120 | - | 20516 |
| lisper | 1 | 0 | 1 | - | 698 |
| miniloop | 1 | 1 | 1 | 166 | 166 |
| simpleloop | 1 | 0 | 1 | - | 915 |

Tabelle 9.3: Compilerbenchmarks (GNU)

| Test | Soll | Ist (aiT) | Ist (loopy) |
|---|---|---|---|
| do_char_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| do_char_008 | [16] | $[1,\infty]$ | [16] |
| do_char_009 | [16] | $[1,\infty]$ | [16] |
| do_char_010 | [1, 16] | $[1,\infty]$ | [1, 16] |
| do_int_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| do_int_008 | [16] | $[1,\infty]$ | [16] |
| do_int_009 | [16] | $[1,\infty]$ | [16] |
| do_int_010 | [1, 16] | $[1,\infty]$ | [1, 16] |
| for_char_001 | [17] | $[1,\infty]$ | [17] |
| for_char_017 | [17] | $[1,\infty]$ | [17] |
| for_char_049 | [1] | $[1,\infty]$ | [1, 17] |
| for_char_058 | [17] | $[1,\infty]$ | [17] |
| for_char_061 | [9] | $[1,\infty]$ | [9] |
| for_char_062 | [17] | $[1,\infty]$ | [17] |
| for_int_001 | [17] | $[1,\infty]$ | [17] |
| for_int_017 | [17] | $[1,\infty]$ | [17] |
| for_int_049 | [1] | $[1,\infty]$ | [1, 17] |
| for_int_058 | [17] | $[1,\infty]$ | [17] |
| for_int_061 | [9] | $[1,\infty]$ | [9] |
| for_int_062 | [17] | $[1,\infty]$ | [17] |
| while_char_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| while_char_008 | [17] | $[1,\infty]$ | [17] |
| while_char_009 | [17] | $[1,\infty]$ | [17] |
| while_char_010 | [1, 17] | $[1,\infty]$ | [1, 17] |
| while_int_001 | $[1,\infty]$ | $[1,\infty]$ | $[1,\infty]$ |
| while_int_008 | [17] | $[1,\infty]$ | [17] |

| Test | Soll | Ist (aiT) | Ist (loopy) |
|---|---|---|---|
| while_int_009 | [17] | $[1, \infty]$ | [17] |
| while_int_010 | $[1, 17]$ | $[1, \infty]$ | $[1, 17]$ |

Tabelle 9.4: Synthetische Schleifentests (GNU)

Für diesen Compiler zeigen die Ergebnisse klar, dass die neue Analyse der alten überlegen ist. Es gibt kein Testbeispiel bei dem die alte Analyse bessere Ergebnisse als das neue Verfahren liefert. Meist liefert nur die neue Analyse überhaupt verwendbare Grenzen.

### 9.3.2 Reale Industrie-Programme

Das Ziel dieser Diplomarbeit ist es, eine auch real einsetzbare Schleifenanalyse zu erstellen. Somit genügt die schon erfolgte Anwendung der Analyse auf künstlich erzeugte Tests und nicht real verwendeten Beispielcode nicht. Die Analyse muss auch ihre Fähigkeiten in der Untersuchung von realen Programmen unter Beweis stellen. Hierzu werden nun einige Messergebnisse aufgelistet, die an realen Programmen aus der Flugzeug- und Fahrzeugindustrie erzielt wurden. Diese wurden mit dem *DiabData Compiler* übersetzt.

Bei den untersuchten Programmen handelt es sich um Teile von Fahrzeug- und Flugzeugsteuerungen, also um sicherheitskritische Software, die harten Echtzeitschranken unterliegt. Aus Datenschutzgründen können die Programme nicht konkret benannt werden. Einige der in ihnen enthaltenen Schleifen können von der Analyse nicht erkannt werden, diese sind vom Benutzer annotiert worden. Diese Annotationen sind jedoch für beide durchgeführten Analysen gleich, um die Vergleichbarkeit zu gewährleisten.

Tabelle 9.5 zeigt die Ergebnisse von Programmen für den *MPC755*, wie sie in der Flugzeugbranche zum Einsatz kommen, Tabelle 9.6 auf der nächsten Seite zeigt einige Ergebnisse von Programmen für den *MPC555*, wie sie in der Automobilindustrie verwendet werden.

Die neue Schleifenanalyse erkennt alle Grenzen, die auch die alte Analyse findet. Zusätzlich kann sie in den Tests *mpc755_4* und *mpc555_2* für jeweils eine weitere Schleife die Grenzen erkennen. Hier scheitert die alte Analyse an Schleifen, die mehrere Schleifentests besitzen und nicht von den enthaltenen Pattern erfasst werden.

Bei den Analyselaufzeiten für die Testbeispiele ist die neue Analyse maximal um den Faktor 2-3 langsamer. Die genauen Laufzeiten sind in den Tabellen 9.7 auf der nächsten Seite und 9.8 auf der nächsten Seite vermerkt.

| Test | Schl. | Grenzen (aiT) | Grenzen (loopy) | WCET (aiT) | WCET (loopy) |
|---|---|---|---|---|---|
| mpc755_1 | 42 | 42 | 42 | 76579 | 76579 |
| mpc755_2 | 26 | 24 | 24 | 16545 | 16545 |
| mpc755_3 | 2 | 2 | 2 | 39983 | 39983 |
| mpc755_4 | 1 | 0 | 1 | - | 1156 |

Tabelle 9.5: Industrie-Programme (MPC755) - Analyseergebnisse

| Test | Schl. | Grenzen (aiT) | Grenzen (loopy) | WCET (aiT) | WCET (loopy) |
|---|---|---|---|---|---|
| mpc555_1 | 12 | 6 | 6 | 139745 | 139745 |
| mpc555_2 | 3 | 2 | 3 | - | 2367 |

Tabelle 9.6: Industrie-Programme (MPC555) - Analyseergebnisse

| Test | Laufzeit/Sek. (aiT) | Laufzeit/Sek. (loopy) |
|---|---|---|
| mpc755_1 | 43,54 | 63,75 |
| mpc755_2 | 3,82 | 9,25 |
| mpc755_3 | 0,53 | 0,77 |
| mpc755_4 | 0,47 | 0,69 |

Tabelle 9.7: Industrie-Programme (MPC755) - Laufzeiten

| Test | Laufzeit/Sek. (aiT) | Laufzeit/Sek. (loopy) |
|---|---|---|
| mpc555_1 | 5,45 | 7,19 |
| mpc555_2 | 0,81 | 1,16 |

Tabelle 9.8: Industrie-Programme (MPC555) - Laufzeiten

## 9.4 Auswertung

Die Messergebnisse zeigen, dass die neue Schleifenanalyse ihre Aufgabe in fast allen untersuchten Beispielen mindestens so gut ausführt wie die alte Schleifenanalyse und diese in vielen übertrifft. Die Laufzeit bleibt hierbei maximal um den Faktor 2-3 hinter der alten Analyse zurück.

Nur einzelne Spezialfälle, die mit den aktuellen Pattern erfasst werden können, bereiten der neuen Analyse Schwierigkeiten. Diese Probleme könnte man durch eine Kombination der neuen Analyse mit den für diese Fälle verwendeten Pattern lösen.

Hingegen ist sie im Vergleich zu der alten Analyse praktisch unabhängig vom verwendeten Compiler. So konnten ohne Anpassungen sehr gute Ergebnisse für den *GNU C*

*Compiler* erreicht werden, wohingegen hierzu an der alten Analyse eine umfassende Anpassung der vorhandenen Pattern erforderlich wäre.

Somit kann man, soweit es die Menge der untersuchten Programme zulässt, folgern, dass die neue Analyse dem von ihr angestrebten Anspruch, auch real einsetzbar zu sein, genügt.

# Kapitel 10 Zusammenfassung & Ausblick

In dieser Arbeit wurde gezeigt, wie man mit Hilfe einer Datenflussanalyse ein Analyseverfahren erstellt, das sichere Schleifengrenzen für Zählschleifen berechnen kann. Hierbei wurde eine externe Werteanalyse miteinbezogen und die erhaltene Schleifenanalyse so konzipiert, dass sie in ein existierendes System zur Bestimmung der WCET integriert werden kann. Zielsetzung der Schleifenanalyse war neben der Berechnung der Schranken auch, dass die erhaltene Analyse, trotz ihrer Anwendung auf Maschinencode, leicht auf verschiedene Architekturen portierbar ist und unabhängig von den zur Erstellung der Programme verwendeten Compilern bleibt.

Dieses neue Analyseverfahren wurde exemplarisch für die *PowerPC*-Architektur implementiert, die in der heutigen Fahrzeug- und Flugzeugtechnik eine bedeutende Rolle spielt. Es wurde eine klare Trennung zwischen maschinenabhängigen und maschinenunabhängigen Teilen vorgenommen, um eine spätere Portierung auf andere Architekturen zu gewährleisten.

Die praktische Anwendbarkeit der Analyse und die Unabhängigkeit gegenüber verschiedenen Compilerarten wurde unter Verwendung von Industrie-Programmen und synthetischen Testbeispielen unter Beweis gestellt. Es zeigte sich, dass die neue Analyse mit unterschiedlichen Compilern funktioniert und die alte Schleifenanalyse von *aiT* hierbei bei weitem übertrifft. Trotz dieser größeren Allgemeinheit bleibt die Laufzeit der neuen Analyse nur um einen kleinen Faktor hinter der alten Analyse zurück.

Die für die Messungen verwendete *PowerPC*-Implementierung wurde im Rahmen der Diplomarbeit vollständig in den kommerziell verwendeten WCET-Analysator *aiT* integriert und steht als Alternative zu der schon existierenden Analyse bereit. Somit stellt das in dieser Arbeit vorgestellte Verfahren meines Wissens die erste auf Maschinencode-Programme anwendbare Analyse zur Bestimmung von Schleifengrenzen dar, die sowohl praktisch anwendbar und implementiert ist, als auch publiziert wurde.

## Ausblick

Die erstellte Analyse ist offen genug konzipiert, um noch genügend Raum für zukünftige Erweiterungen und Verbesserungen zu bieten. Folgende Aspekte können hierbei von großem Interesse sein:

- Portierung auf weitere Architekturen:
  In der heutigen Zeit kommen auch *ARM*-Prozessoren in zahlreichen eingebetteten Systemen zum Einsatz und z.B. der neue Infineon *TriCore*-Prozessor bekommt mehr und mehr Bedeutung. Es wäre interessant, die Schleifenanalyse auch für diese Architekturen zu implementieren.

- Kombination der alten und neuen Schleifenanalyse:
  Durch Integration der Pattern aus der alten Schleifenanalyse für Spezialfälle, welche die neue Analyse nicht abdeckt, kann das Analyseergebnis weiter verbessert werden.

- Erweiterung der Gleichungen um Multiplikation mit einer Konstanten:
  Das Ziel, Grenzen für Zählschleifen zu erkennen, wird im Moment erfolgreich bewältigt, ohne Multiplikation innerhalb der verwendeten Formeln zu erlauben. Um auch noch multiplikative Schleifen zu behandeln, könnte man jedoch diese dahingehend erweitern.

Anhang

# A Spezifikation der Flussanalyse

Im Folgenden sind die *DATLA*- und *FULA*-Spezifikationen des verwendeten Datenflussproblems vollständig aufgelistet. Hierbei handelt es sich um die Version, welche in der *PowerPC*-Implementierung zum Einsatz kommt.

## A.1 DATLA

```
SET

/**
 * PPC gennames
 */
genname_enum = (
    ppc__illegal, ppc__twi, ppc__mulli, ppc__subfic, ppc__cmpli, ppc__cmpi, ppc__addic,
    ppc__addicD, ppc__addi, ppc__addis, ppc__bc_DCtr_Cond, ppc__bcl_DCtr_Cond,
    ppc__bca_DCtr_Cond, ppc__bcla_DCtr_Cond, ppc__bc_Cond, ppc__bcl_Cond, ppc__bca_Cond,
    ppc__bcla_Cond, ppc__bc_DCtr, ppc__bcl_DCtr, ppc__bca_DCtr, ppc__bcla_DCtr, ppc__bc,
    ppc__bcl, ppc__bca, ppc__bcla, ppc__sc, ppc__b, ppc__bl, ppc__ba, ppc__bla, ppc__mcrf,
    ppc__bclr_DCtr_Cond, ppc__bclrl_DCtr_Cond, ppc__crnor, ppc__rfi, ppc__rfci, ppc__crandc,
    ppc__isync, ppc__crxor, ppc__crnand, ppc__crand, ppc__creqv, ppc__crorc, ppc__cror,
    ppc__bclr_Cond, ppc__bclrl_Cond, ppc__bcctr_Cond, ppc__bcctrl_Cond, ppc__bclr_DCtr,
    ppc__bclrl_DCtr, ppc__bclr, ppc__bclrl, ppc__bcctr, ppc__bcctrl, ppc__rlwimi, ppc__rlwimiD,
    ppc__rlwinm, ppc__rlwinmD, ppc__rlwnm, ppc__rlwnmD, ppc__ori, ppc__oris, ppc__xori,
    ppc__xoris, ppc__andiD, ppc__andisD, ppc__cmp, ppc__tw, ppc__subfc, ppc__subfcD, ppc__addc,
    ppc__addcD, ppc__mulhwu, ppc__mulhwuD, ppc__mfcr, ppc__lwarx, ppc__lwzx, ppc__slw,
    ppc__slwD, ppc__cntlzw, ppc__cntlzwD, ppc__and, ppc__andD, ppc__cmpl, ppc__subf,
    ppc__subfD, ppc__dcbst, ppc__lwzux, ppc__andc, ppc__andcD, ppc__mulhw, ppc__mulhwD,
    ppc__mfmsr, ppc__dcbf, ppc__lbzx, ppc__neg, ppc__negD, ppc__lbzux, ppc__nor, ppc__norD,
    ppc__wrtee, ppc__subfe, ppc__subfeD, ppc__adde, ppc__addeD, ppc__mtcrf, ppc__mtmsr,
    ppc__stwcxD, ppc__stwx, ppc__wrteei, ppc__stwux, ppc__subfze, ppc__subfzeD, ppc__addze,
    ppc__addzeD, ppc__mtsr, ppc__stbx, ppc__subfme, ppc__subfmeD, ppc__addme, ppc__addmeD,
    ppc__mullw, ppc__mullwD, ppc__mtsrin, ppc__dcbtst, ppc__stbux, ppc__icbt, ppc__add,
    ppc__addD, ppc__dcbt, ppc__lhzx, ppc__eqv, ppc__eqvD, ppc__tlbie, ppc__eciwx, ppc__lhzux,
    ppc__xor, ppc__xorD, ppc__mfdcr, ppc__mfspr, ppc__lhax, ppc__tlbia, ppc__mftb, ppc__lhaux,
    ppc__sthx, ppc__orc, ppc__orcD, ppc__ecowx, ppc__sthux, ppc__or, ppc__orD, ppc__mtdcr,
    ppc__dccci, ppc__divwu, ppc__divwuD, ppc__mtspr, ppc__dcbi, ppc__nand, ppc__nandD,
    ppc__dcread, ppc__divw, ppc__divwD, ppc__mcrxr, ppc__subfco, ppc__subfcoD, ppc__addco,
    ppc__addcoD, ppc__lswx, ppc__lwbrx, ppc__lfsx, ppc__srw, ppc__srwD, ppc__subfo, ppc__subfoD,
    ppc__tlbsync, ppc__lfsux, ppc__mfsr, ppc__lswi, ppc__sync, ppc__lfdx, ppc__nego, ppc__negoD,
    ppc__lfdux, ppc__subfeo, ppc__subfeoD, ppc__addeo, ppc__addeoD, ppc__mfsrin, ppc__stswx,
    ppc__stwbrx, ppc__stfsx, ppc__stfsux, ppc__subfzeo, ppc__subfzeoD, ppc__addzeo, ppc__addzeoD,
    ppc__stswi, ppc__stfdx, ppc__subfmeo, ppc__subfmeoD, ppc__addmeo, ppc__addmeoD, ppc__mullwo,
    ppc__mullwoD, ppc__dcba, ppc__stfdux, ppc__addo, ppc__addoD, ppc__lhbrx, ppc__sraw,
    ppc__srawD, ppc__srawi, ppc__srawiD, ppc__eieio, ppc__tlbsx, ppc__tlbsxD, ppc__sthbrx,
```

```
    ppc__extsh, ppc__extshD, ppc__tlbre, ppc__extsb, ppc__extsbD, ppc__iccci, ppc__divwuo,
    ppc__divwuoD, ppc__tlbwe, ppc__icbi, ppc__stfiwx, ppc__icread, ppc__divwo, ppc__divwoD,
    ppc__dcbz, ppc__lwz, ppc__lwzu, ppc__lbz, ppc__lbzu, ppc__stw, ppc__stwu, ppc__stb, ppc__stbu,
    ppc__lhz, ppc__lhzu, ppc__lha, ppc__lhau, ppc__sth, ppc__sthu, ppc__lmw, ppc__stmw, ppc__lfs,
    ppc__lfsu, ppc__lfd, ppc__lfdu, ppc__stfs, ppc__stfsu, ppc__stfd, ppc__stfdu, ppc__fdivs,
    ppc__fdivsD, ppc__fsubs, ppc__fsubsD, ppc__fadds, ppc__faddsD, ppc__fsqrts, ppc__fsqrtsD,
    ppc__fres, ppc__fresD, ppc__fmuls, ppc__fmulsD, ppc__fmsubs, ppc__fmsubsD, ppc__fmadds,
    ppc__fmaddsD, ppc__fnmsubs, ppc__fnmsubsD, ppc__fnmadds, ppc__fnmaddsD, ppc__fcmpu, ppc__frsp,
    ppc__frspD, ppc__fctiw, ppc__fctiwD, ppc__fctiwz, ppc__fctiwzD, ppc__fdiv, ppc__fdivD,
    ppc__fsub, ppc__fsubD, ppc__fadd, ppc__faddD, ppc__fsqrt, ppc__fsqrtD, ppc__fsel, ppc__fselD,
    ppc__fmul, ppc__fmulD, ppc__frsqrte, ppc__frsqrteD, ppc__fmsub, ppc__fmsubD, ppc__fmadd,
    ppc__fmaddD, ppc__fnmsub, ppc__fnmsubD, ppc__fnmadd, ppc__fnmaddD, ppc__fcmpo, ppc__mtfsb1,
    ppc__mtfsb1D, ppc__fneg, ppc__fnegD, ppc__mcrfs, ppc__mtfsb0, ppc__mtfsb0D, ppc__fmr,
    ppc__fmrD, ppc__mtfsfi, ppc__mtfsfiD, ppc__fnabs, ppc__fnabsD, ppc__fabs, ppc__fabsD,
    ppc__mffs, ppc__mffsD, ppc__mtfsf, ppc__mtfsfD
      || ppc__addi | ppc__illegal)

/**
 * The generic part of the loopy.set
 * above the PPC/ARM/... implementations will include their gennames and other specific stuff
 */

/* A value is an intervall where:
 *
 * - top/bot means invalid value (aka bot)
 * - bot or top as interval border does not means "+/-infinity", but
 *   +/- MAX_INT, so that all operations should check for overflows.
 */
ivalue       = ibound * ibound

/**
 * normal interval
 */
vvalue = snum * snum

/**
 * Type of the Variable
 */
var_type = (v_none,
            v_mem,
            v_reg,
            v_invalid
            || v_none | v_invalid)

/**
 * Variable
 * consists of: type * address/registernumber * width
 */
var = var_type * snum * snum

/**
 * Here we go to formulize the equations ;)
 */

/**
 * value for var, might be:
 * var + value, if var_type == memory || register
 * value, if var_type == none
 * third part is the width of the var in bytes
 * fourth part == signedness
 */
var_value = var * vvalue * snum * var_signed

/**
 * Are we sign extended or not?
```

```
 */
var_signed = (s_none, s_notext, s_ext, s_mixed || s_none | s_mixed)

/**
 * list of values
 */
var_values = list(var_value)

/**
 * a equation for given target var
 */
equation = var * var_values

/**
 * set of all equations known
 */
equations = list (equation)

/**
 * internal carrier
 * first call id * calling context id * equations set
 */
loopy_carrier_i = loopinfo * equations

/**
 * 8*str (for destinations + sources)
 */
str_8 = str * str * str * str * str * str * str * str

/**
 * memory access info ;)
 */
imacc = ivalue * snum * ivalue * snum

DOMAIN

ibound     = flat (snum)

loopinfo = flat (vvalue)

loopy_carrier = flat(loopy_carrier_i)

/* EOF */
```

## A.2 FULA

```
NODE

// is this node a loop call?
loop_call_type : snum#
loop_return_type : snum#

// destinations
dst1*     : str
dst2*     : str
dst3*     : str
dst4*     : str
dst5*     : str
dst6*     : str
dst7*     : str
dst8*     : str
```

```
// sources
src1*     : str
src2*     : str
src3*     : str
src4*     : str
src5*     : str
src6*     : str
src7*     : str
src8*     : str

loopy_genname            : genname_enum#

POSITION

loop_first_call : snum#
loop_context : snum#

loopy_block : snum#
loopy_instruction : snum#
loopy_context : snum#

loop_start_equations : equations#

memory_access :imacc#

instruction_executed :snum#

PROBLEM loopy

prefix     : loopy_
direction  : forward
carrier    : loopy_carrier
retfunc    : return_dfi
combine    : combine_carrier
widening   : widening_set
equal      : eq_set
init       : bot
init_start : lift((bot, []))


TRANSFER

// handle loop calls special
// for all loops in the current nesting depth, init with loop vars for loop start on first call
// recursive call should give bot as flow, as we only want to calc the equations for one round
// for all other loops and procedures, just pass flow around
_,crl_call:   case loop_call_type of
                1 => if @ = lift((bot, []))
                       then lift((lift((loop_first_call, loop_context)),
                                  combine_equations (equations_sort(loop_start_equations),
                                    equations_sort(loop_start_equations))))
                       else lift((top, []))
                     endif;
                2 => if (@ != bot && @ != top)
                       then if (drop (@)!1!2 != bot && drop (@)!1!2 != top)
                                   then lift ((drop (@)!1!2,
                         combine_equations (equations_sort(loop_rec_equations ((drop(drop (@)!1!2))!1!2,
                                                                               (drop(drop (@)!1!2))!2!2)),
                                         equations_sort(loop_rec_equations ((drop(drop (@)!1!2))!1!2,
                                                                               (drop(drop (@)!1!2))!2!2)))))
                                   else @
                                endif
                       else @
                     endif;
                x => @;
```

```
            endcase;

// propagate flow value from before call
_,crl_local: if loop_call_type = 0 then @ else bot endif;

// handle loop return special
return_node(),_ : case loop_return_type of
                1 => lift((bot, []));
                x => @;
            endcase;

// normal transition
normal_node(),_ : case @ of
                lift((bot,b)) => lift ((bot, b));
                lift((top,b)) => lift ((top, b));
                lift((a,b)) => case instruction_executed of
                                // instruction will not be executed
                                0 => @;

                                // instruction will sometimes be executed
                                1 => lift ((
                                         a,

                                         // in the sometimes case: combine the original equations with the
                                         // result of the update equations stuff
                                         equations_sort(combine_equations(b,
                                            update_equations (b,
                                                           loopy_genname,
                                                           (dst1,dst2,dst3,dst4,dst5,dst6,dst7,dst8),
                                                           (src1,src2,src3,src4,src5,src6,src7,src8),
                                                           memory_access, loopy_block,
                                                           loopy_instruction, loopy_context)))
                                    ));

                                // always executed
                                2 => lift ((
                                         a,

                                         update_equations (b,
                                                           loopy_genname,
                                                           (dst1,dst2,dst3,dst4,dst5,dst6,dst7,dst8),
                                                           (src1,src2,src3,src4,src5,src6,src7,src8),
                                                           memory_access, loopy_block,
                                                           loopy_instruction, loopy_context)
                                    ));

                               endcase;
                a => a;
            endcase;
_,_ : @;

SUPPORT

// equations for recursive call of loop in correct nesting depth
// first par: procnum, second: context
loop_rec_equations :: snum,snum -> equations;

// get ivalue of given register
// args: register,block,instruction,context
register_value :: snum,snum,snum,snum -> ivalue;

// negate a given ivalue
negate_value :: ivalue -> ivalue;

//
```

```
// equality
//

eq_set :: loopy_carrier,loopy_carrier -> bool;
eq_set (a, b) = a = b;

//
// interprocedual, use the stuff we get from the return
// two types: if dif_loc == bottom, we are a loop call return, else we are a real call
// in second case, apply calling convention
//
return_dfi :: loopy_carrier,loopy_carrier -> loopy_carrier;
return_dfi (bot, dfi_ret) = dfi_ret;
return_dfi (top, dfi_ret) = dfi_ret;
return_dfi (_, bot) = bot;
return_dfi (_, top) = top;
return_dfi (lift((ctx1, dfi_loc)), lift((ctx2, dfi_ret)))
 = lift ((ctx2, calling_conv (dfi_loc, dfi_ret)));

//
// widening
//
widening_set :: loopy_carrier,loopy_carrier -> loopy_carrier;
widening_set (_, top) = top;
widening_set (top, _) = top;
widening_set (a, bot) = a;
widening_set (bot, b) = b;
widening_set (lift((ctx1, a)), lift((ctx2, b)))
 = lift ((ctx2, wide_equations (a, b)));

// widening helper
wide_equations :: equations, equations -> equations;
wide_equations (a, []) = [];
wide_equations (a, (x, v) : eq)
 = if find_values (a, x) != [] && find_values (a, x) != v
   then wide_equations (a, eq) else (x, v) : wide_equations (a, eq) endif;

//
// carrier values combining, e.g. combining of the carrier and the equationlist inside
//

combine_carrier :: loopy_carrier,loopy_carrier -> loopy_carrier;
// plain bot & top
combine_carrier (_, top) = top;
combine_carrier (top, _) = top;
combine_carrier (a, bot) = a;
combine_carrier (bot, b) = b;
// other stuff, 2 equation lists lifted twice
combine_carrier (lift((ctx1, a)), lift((ctx2, b)))
  = lift((ctx1, equations_sort(combine_equations(a, b)))); // fix this later, lub on the ctx

// combine two equation lists
combine_equations :: equations,equations -> equations;
combine_equations ([], b) = [];
combine_equations (ahead : atail, b) = result_equation (ahead, b, []) ++ combine_equations (atail, b);

// get the [result equation], if there are equations which will kill it [] is returned
result_equation :: equation,equations,equations -> equations;
result_equation (a, [], res) = res;
result_equation (a, bhead : btail, res) = if equation_match (a, bhead)
   then result_equation (a, btail, combine_equation(a, bhead))
   else if equation_distinct (a, bhead) then result_equation (a, btail, res)
                                        else result_equation (a, btail, []) endif endif;

// test if two equations cover exactly the same destination
```

```
equation_match :: equation,equation -> bool;
equation_match (a, b) = ((a!1!2)!1!3 = (b!1!2)!1!3) // same types
                          && (
                             ( ((a!1!2)!1!3 = v_reg) && ((a!1!2)!2!3 = (b!1!2)!2!3) ) // same registers
                          || ( ((a!1!2)!1!3 = v_mem)
                               && ((a!1!2)!2!3 = (b!1!2)!2!3)
                               && ((a!1!2)!3!3 = (b!1!2)!3!3) ) // same mem cells
                             );

// test if equations are distinct
// this is true, if either the type of the destination var is different or
// for memory vars the addresses are distinct or for register vars the register number
// is different
equation_distinct :: equation,equation -> bool;
equation_distinct (a, b) = ((a!1!2)!1!3 != (b!1!2)!1!3) // different var types
                          || ( ((a!1!2)!1!3 = v_reg) && ((a!1!2)!2!3 != (b!1!2)!2!3)) // different registers
                          || ( ((a!1!2)!1!3 = v_mem)
                               && ( ( ((a!1!2)!2!3 + (a!1!2)!3!3) <= (b!1!2)!2!3 )
                               || ( (a!1!2)!2!3 >= ((b!1!2)!2!3 + (b!1!2)!3!3) ) )); // different mem cells


// only called for two equations for same destination variable
// will combine the values + signed info + width info
// if failure, will return empty list, else list with the one result equation
combine_equation :: equation,equation -> equations;
combine_equation ((v, va), (_, vb))
  = let resv = combine_values (var_values_sort (va ++ vb)); in if resv = [] then [] else [(v, resv)] endif;

// combine two valuelists of one var
combine_values :: var_values -> var_values;
combine_values ([]) = [];
combine_values (x : []) = [x];
combine_values (x : l) = comb_values_i (x, l);

// combine the values of var_values for the same var
comb_values_i :: var_value, var_values -> var_values;
comb_values_i (x, []) = [x];
comb_values_i ((v1, c1, w1, s1), (v2, c2, w2, s2) : l)
  = if (v1 != v2)
      then (v1, c1, w1, s1) : comb_values_i ((v2, c2, w2, s2), l)
      else comb_values_i ((v1, merge_vvalue (c1, c2),

                           // wide = min width
                           if w1 < w2 then w1 else w2 endif,

                           // signedness = lub!
                           if (w1 = w2 && (w1 = 4 || s1 = s2)) then s1
                           else s_mixed
                           endif)
                    , l)
    endif;

// merge intervals
merge_vvalue :: vvalue,vvalue -> vvalue;
merge_vvalue ((a, b), (x, y))
  = (if (a < x) then a else x endif, if (b > y) then b else y endif);

//
// use the power: use the calling convention
//

// apply calling convention
calling_conv :: equations,equations -> equations;
calling_conv (eq, eqafterreturn)
  = equations_sort(calling_conv_new_equations (eq) ++ calling_conv_destroy_equations (eqafterreturn));
```

```
// kill all registers which are saved if convention is valid
calling_conv_destroy_equations :: equations -> equations;
calling_conv_destroy_equations ([]) = [];
calling_conv_destroy_equations (x : l)
  = if ((x!1!2)!1!3 = v_reg && callee_saved_reg ((x!1!2)!2!3)) then calling_conv_destroy_equations (l)
                                else x : calling_conv_destroy_equations (l)
    endif;

// get the old valid equations!!
calling_conv_new_equations :: equations -> equations;
calling_conv_new_equations ([]) = [];
calling_conv_new_equations (x : l)
  = if ((x!1!2)!1!3 = v_reg && callee_saved_reg ((x!1!2)!2!3)) then x : calling_conv_new_equations (l)
                                else calling_conv_new_equations (l)
    endif;

//
// updating of the equations by instructions
// two pass stuff
// first: to be safe, destroy all info about vars we will write to by this instruction
// second: to get info, insert new equations for destination vars if possible
//

// update_equations just first destroys to be safe and than inserts
update_equations :: equations,genname_enum,str_8,str_8,imacc,snum,snum,snum -> equations;
update_equations (eq, gn, dsts, srcs, mem, block, instruction, context)
  = equations_sort(new_equations (eq, gn, dsts, srcs, mem, block, instruction, context)
                    ++ destroy_equations (eq, gn, dsts, srcs, mem));

// destroy the bad guys
destroy_equations :: equations,genname_enum,str_8,str_8,imacc -> equations;
destroy_equations (eq, gn, dsts, srcs, mem)
  = destroy_on_reg_write(destroy_on_mem_write (eq, mem), gn, dsts, srcs);

// destroy vars for memory cells we write to
destroy_on_mem_write :: equations,imacc -> equations;
destroy_on_mem_write ([], mem) = [];
destroy_on_mem_write (x : eq, mem)
  = if (((x!1!2)!1!3 = v_mem) && memcell_included ((x!1!2)!2!3, (x!1!2)!3!3, mem!3!4, mem!4!4))
    then destroy_on_mem_write (eq, mem) else x : destroy_on_mem_write (eq, mem) endif;

// is this memcell (address + width) included into the area given by the value + access width?
memcell_included :: snum,snum,ivalue,snum -> bool;
memcell_included (maddr, mwidth, arange, wwidth)
  = case arange of
          (bot, _) => (wwidth > 0);
          (top, _) => (wwidth > 0);
          (_, bot) => (wwidth > 0);
          (_, top) => (wwidth > 0);
          (lift(a), lift(b)) => (wwidth > 0) && (maddr+mwidth > a && maddr < b+wwidth);
    endcase;

// does on register kill another
register_kills_register :: snum,snum -> snum;

//
// dst/src to var (only for register stuff)
// yes, this is dumb atm, I will source it out to C
//
name_to_var :: str -> var;
name_to_var (x) = if str_to_register (x) != -1 then (v_reg, str_to_register (x), 4)
                                               else (v_invalid, 0, 4) endif;

// macc to var, if possible!!!, else return var with invalid flag
```

```
macc_to_var :: ivalue,snum -> var;
macc_to_var ((bot, _), _) = (v_invalid, 0, 4);
macc_to_var ((top, _), _) = (v_invalid, 0, 4);
macc_to_var ((_, bot), _) = (v_invalid, 0, 4);
macc_to_var ((_, top), _) = (v_invalid, 0, 4);
macc_to_var ((a, b), 1) = if drop(a) = drop(b) then (v_mem, drop(a), 1) else (v_invalid, 0, 4) endif;
macc_to_var ((a, b), 2) = if drop(a) = drop(b) then (v_mem, drop(a), 2) else (v_invalid, 0, 4) endif;
macc_to_var ((a, b), 4) = if drop(a) = drop(b) then (v_mem, drop(a), 4) else (v_invalid, 0, 4) endif;
macc_to_var (_, _) = (v_invalid, 0, 4);

// str to register number, -1 if invalid
str_to_register :: str -> snum;

// str to integer
str_to_snum :: str -> snum;

// merge sort for equations
equations_merge :: equations, equations -> equations;
equations_merge ([], b) = b;
equations_merge (a, []) = a;
equations_merge (a : al, b : bl)
  = if (var_less (a!1!2, b!1!2)) then a:equations_merge(al, b:bl) else b:equations_merge(a:al,bl) endif;

equations_sort :: equations -> equations;
equations_sort ([]) = [];
equations_sort (a : al)
  = if al = [] then a : al else equations_merge (equations_sort([a]),equations_sort(al)) endif;

// merge sort for var_values
var_values_merge :: var_values, var_values -> var_values;
var_values_merge ([], b) = b;
var_values_merge (a, []) = a;
var_values_merge (a : al, b : bl)
  = if (var_less (a!1!4, b!1!4)) then a:var_values_merge(al, b:bl) else b:var_values_merge(a:al,bl) endif;

var_values_sort :: var_values -> var_values;
var_values_sort ([]) = [];
var_values_sort (a : al)
  = if al = [] then a : al else var_values_merge (var_values_sort([a]),var_values_sort(al)) endif;

// var <
var_less :: var,var -> bool;
var_less ((v_none, a, _), (v_none, b, _)) = a < b;
var_less ((v_reg, a, _), (v_reg, b, _)) = a < b;
var_less ((v_mem, a, _), (v_mem, b, _)) = a < b;
var_less ((v_invalid, a, _), (v_invalid, b, _)) = a < b;
var_less ((v_none, _, _), _) = true;
var_less ((v_reg, _, _), (t, _, _)) = t != v_none;
var_less ((v_mem, _, _), (t, _, _)) = t != v_none && t != v_reg;
var_less ((v_invalid, _, _), _) = false;

//
// cut the higher bytes
//
cut_higher_bytes :: var_values,snum -> var_values;
cut_higher_bytes ([], _) = [];
cut_higher_bytes ((v, c, w, s) : vs, nw)
  = if (w < nw) then (v, c, nw, if (s = s_notext)
                               then s_notext else s_mixed endif) : cut_higher_bytes (vs, nw)
                else (v, c, nw, s_notext) : cut_higher_bytes (vs, nw) endif;

//
// extend sign
//
sign_extend :: var_values,snum,snum -> var_values;
```

```
sign_extend ([], _, _) = [];
sign_extend ((v, c, w, s) : vs, from, to)
  = if (w < from) then (v, c, from, if (s = s_ext)
                                   then s_ext else s_mixed endif) : sign_extend (vs, from, to)
                 else (v, c, from, s_ext) : sign_extend (vs, from, to) endif;

//
// give back values for given var in given carrier, if not found, return empty list
//
find_values_in_carrier :: loopy_carrier, var -> var_values;
find_values_in_carrier (bot, _) = [];
find_values_in_carrier (top, _) = [];
find_values_in_carrier (lift((bot, _)), _) = [];
find_values_in_carrier (lift((top, _)), _) = [];
find_values_in_carrier (lift((_, eq)), var) = find_values (eq, var);

//
// find equation for given var in equations list, if no found, return (var, [], )
//
find_values :: equations, var -> var_values;
find_values ([], var) = [];
find_values (x : eq, var) = if (var = x!1!2) then x!2!2 else find_values (eq, var) endif;

//
// add a constant to the given var_values
// includes overflow check!
//
add_to_values :: var_values,snum -> var_values;
add_to_values ([], _) = [];
add_to_values ((v, (c1, c2), w, s) : vs, inc)
  = if (overflow (c1, inc) || overflow (c2, inc))
      then []
      else (v, (c1+inc, c2+inc), w, s) : add_to_values (vs, inc)
    endif;

// check for overflow if we add b to a!
overflow :: snum, snum -> bool;
overflow (a, b) = if (b = 0)
                    then false
                    else if (b > 0)
                           then (a+b < a)
                           else (a+b > a)
                         endif
                  endif;

//
// DONE ;)
//
//
// PPC SPECIFIC PART OF THE CALLING CONVENTIONS
//
callee_saved_reg :: snum -> bool;
callee_saved_reg (reg) = (13 <= reg) && (reg <= 31);

//
// destroy vars for registers we write to
//
destroy_on_reg_write :: equations,genname_enum,str_8,str_8 -> equations;
destroy_on_reg_write ([], gn, dsts, srcs) = [];

// load multiple
destroy_on_reg_write (x : eq, ppc__lmw, dsts, srcs)
    = if ((x!1!2)!1!3 = v_reg && ((name_to_var (dsts!1!8))!2!3 <= (x!1!2)!2!3)
          && ((x!1!2)!2!3 <= 31))
      then destroy_on_reg_write (eq, ppc__lmw, dsts, srcs)
```

```
      else x : destroy_on_reg_write (eq, ppc__lmw, dsts, srcs) endif;

// normal other stuff
destroy_on_reg_write (x : eq, gn, dsts, srcs)
    = if (  name_to_var (dsts!1!8) = x!1!2 || name_to_var (dsts!2!8) = x!1!2
         || name_to_var (dsts!3!8) = x!1!2 || name_to_var (dsts!4!8) = x!1!2
         || name_to_var (dsts!5!8) = x!1!2 || name_to_var (dsts!6!8) = x!1!2
         || name_to_var (dsts!7!8) = x!1!2 || name_to_var (dsts!8!8) = x!1!2
         )
      then destroy_on_reg_write (eq, gn, dsts, srcs)
      else x : destroy_on_reg_write (eq, gn, dsts, srcs) endif;

//
// here comes the interesting stuff ;)
// insert new equations for the given genname ;)
//
new_equations :: equations,genname_enum,str_8,str_8,imacc,snum,snum,snum -> equations;

// addi dst, src, const
// do overflow check
new_equations (eq, ppc__addi, dsts, srcs, mem, block, instruction, context)
 = addi_helper (eq, dsts, srcs, 0);

// addic dst, src, const
// do overflow check
new_equations (eq, ppc__addic, dsts, srcs, mem, block, instruction, context)
 = addi_helper (eq, dsts, srcs, 100);

// addicD dst, src, const
// do overflow check
new_equations (eq, ppc__addicD, dsts, srcs, mem, block, instruction, context)
 = addi_helper (eq, dsts, srcs, 100);

// add dst, src, src2
// do overflow check
new_equations (eq, ppc__add, dsts, srcs, mem, block, instruction, context)
  = add_helper (eq, dsts, srcs, block, instruction, context);

// addD dst, src, src2
// do overflow check
new_equations (eq, ppc__addD, dsts, srcs, mem, block, instruction, context)
  = add_helper (eq, dsts, srcs, block, instruction, context);

// addo dst, src, src2
// do overflow check
new_equations (eq, ppc__addo, dsts, srcs, mem, block, instruction, context)
  = add_helper (eq, dsts, srcs, block, instruction, context);

// addoD dst, src, src2
// do overflow check
new_equations (eq, ppc__addoD, dsts, srcs, mem, block, instruction, context)
  = add_helper (eq, dsts, srcs, block, instruction, context);

// subf dst, src, src2
// do overflow check
new_equations (eq, ppc__subf, dsts, srcs, mem, block, instruction, context)
  = subf_helper (eq, dsts, srcs, block, instruction, context);

// subfD dst, src, src2
// do overflow check
new_equations (eq, ppc__subfD, dsts, srcs, mem, block, instruction, context)
  = subf_helper (eq, dsts, srcs, block, instruction, context);

// subfo dst, src, src2
// do overflow check
```

```
new_equations (eq, ppc__subfo, dsts, srcs, mem, block, instruction, context)
  = subf_helper (eq, dsts, srcs, block, instruction, context);

// subfoD dst, src, src2
// do overflow check
new_equations (eq, ppc__subfoD, dsts, srcs, mem, block, instruction, context)
  = subf_helper (eq, dsts, srcs, block, instruction, context);

// ppc__bc_DCtr (decrement the ctr register!!)
// do overflow check
new_equations (eq, ppc__bc_DCtr, dsts, srcs, mem, block, instruction, context)
 = let dvar = name_to_var(dsts!4!8); // destination register
       incr = -1; // increment
       values = find_values (eq, name_to_var(srcs!4!8)); // values to increment
   in if (dvar!1!3 = v_invalid || values = [])
        then [] // nothing to do, we don't know the source stuff
        else if (add_to_values (values, incr) = []) then []
               else [(dvar, add_to_values (values, incr))] endif // add information about destination var
      endif;

// cut higher 16/24 bits
new_equations (eq, ppc__rlwinm, dsts, srcs, mem, block, instruction, context)
 = let dvar = name_to_var(dsts!1!8); // destination register
       values = find_values (eq, name_to_var(srcs!2!8)); // values to increment
   in if (dvar!1!3 = v_invalid || values = [] || srcs!3!8 != "0" || srcs!5!8 != "0x1f")
        then [] // nothing to do, we don't know the source stuff
        else if (srcs!4!8 = "0x10") // cut higher 16 bit
               then [(dvar, cut_higher_bytes (values, 2))]
               else if (srcs!4!8 = "0x18") // cut higher 24 bit
                      then [(dvar, cut_higher_bytes (values, 1))]
                      else []
                    endif
               endif
      endif;

// move == or d, s, s
new_equations (eq, ppc__or, dsts, srcs, mem, block, instruction, context)
 = or_move_helper (eq, dsts, srcs, mem);

// move. == or d, s, s
new_equations (eq, ppc__orD, dsts, srcs, mem, block, instruction, context)
 = or_move_helper (eq, dsts, srcs, mem);

// move from special register
new_equations (eq, ppc__mfspr, dsts, srcs, mem, block, instruction, context)
 = reg_move_helper (eq, dsts!1!8, srcs!2!8);

// move to special register
new_equations (eq, ppc__mtspr, dsts, srcs, mem, block, instruction, context)
 = reg_move_helper (eq, dsts!1!8, srcs!2!8);

// sign extensions, extend half
new_equations (eq, ppc__extsh, dsts, srcs, mem, block, instruction, context)
 = do_sign_extend (eq, dsts!1!8, srcs!2!8, 2, 4);

// sign extensions, extend half + set condition
new_equations (eq, ppc__extshD, dsts, srcs, mem, block, instruction, context)
 = do_sign_extend (eq, dsts!1!8, srcs!2!8, 2, 4);

// sign extensions, extend byte
new_equations (eq, ppc__extsb, dsts, srcs, mem, block, instruction, context)
 = do_sign_extend (eq, dsts!1!8, srcs!2!8, 1, 4);

// sign extensions, extend byte + set condition
new_equations (eq, ppc__extsbD, dsts, srcs, mem, block, instruction, context)
```

```
 = do_sign_extend (eq, dsts!1!8, srcs!2!8, 1, 4);

//
// loads (lwz*)
//

// lwz dst, src[mem]
new_equations (eq, ppc__lwz, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, false);

// lwzu dst, src[mem]
new_equations (eq, ppc__lwzu, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, true);

// lwzx dst, src[mem]
new_equations (eq, ppc__lwzx, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, false);

// lwzux dst, src[mem]
new_equations (eq, ppc__lwzux, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, true);

// lhz dst, src[mem]
new_equations (eq, ppc__lhz, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, false);

// lhzu dst, src[mem]
new_equations (eq, ppc__lhzu, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, true);

// lhzx dst, src[mem]
new_equations (eq, ppc__lhzx, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, false);

// lhzux dst, src[mem]
new_equations (eq, ppc__lhzux, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, true);

// lha dst, src[mem]
new_equations (eq, ppc__lha, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, true, false);

// lhau dst, src[mem]
new_equations (eq, ppc__lhau, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, true, true);

// lhax dst, src[mem]
new_equations (eq, ppc__lhax, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, true, false);

// lhaux dst, src[mem]
new_equations (eq, ppc__lhaux, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, true, true);

// lbz dst, src[mem]
new_equations (eq, ppc__lbz, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, false);

// lbzu dst, src[mem]
new_equations (eq, ppc__lbzu, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, true);

// lbzx dst, src[mem]
new_equations (eq, ppc__lbzx, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, false);
```

```
// lbzux dst, src[mem]
new_equations (eq, ppc__lbzux, dsts, srcs, mem, block, instruction, context)
 = load_from_mem (eq, dsts!1!8, srcs!1!8, mem, false, true);

//
// stores (stw*, sth*, stb*)
//

// stw src, dst[mem]
new_equations (eq, ppc__stw, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, false);

// stwu src, dst[mem]
new_equations (eq, ppc__stwu, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, true);

// stwx src, dst[mem]
new_equations (eq, ppc__stwx, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, false);

// stwux src, dst[mem]
new_equations (eq, ppc__stwux, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, true);

// stb src, dst[mem]
new_equations (eq, ppc__stb, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, false);

// stbu src, dst[mem]
new_equations (eq, ppc__stbu, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, true);

// stbx src, dst[mem]
new_equations (eq, ppc__stbx, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, false);

// stbux src, dst[mem]
new_equations (eq, ppc__stbux, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, true);

// sth src, dst[mem]
new_equations (eq, ppc__sth, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, false);

// sthu src, dst[mem]
new_equations (eq, ppc__sthu, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, true);

// sthx src, dst[mem]
new_equations (eq, ppc__sthx, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, false);

// sthux src, dst[mem]
new_equations (eq, ppc__sthux, dsts, srcs, mem, block, instruction, context)
 = store_to_mem (eq, dsts!1!8, srcs!1!8, mem, true);

//
// load/store multiple
//

// lmw dst, src[mem]
new_equations (eq, ppc__lmw, dsts, srcs, mem, block, instruction, context)
 = if (invalid_mem_for_multiple ((name_to_var (dsts!1!8))!2!3, mem!1!4, mem!2!4))
     then []
```

```
     else load_multiple (eq, (name_to_var (dsts!1!8))!2!3, drop((mem!1!4)!1!2))
   endif;

// stmw dst, src[mem]
new_equations (eq, ppc__stmw, dsts, srcs, mem, block, instruction, context)
 = if (invalid_mem_for_multiple ((name_to_var (srcs!1!8))!2!3, mem!3!4, mem!4!4))
     then []
     else store_multiple (eq, (name_to_var (srcs!1!8))!2!3, drop((mem!3!4)!1!2))
   endif;

// dummy fallback, no new equations
new_equations (_, _, _, _, _, _, _, _) = [];

//
// helpers for store multiple word stuff
//

// is mem access valid for lmw/stmw? : startregister + mem access
invalid_mem_for_multiple :: snum, ivalue, snum -> bool;
invalid_mem_for_multiple (_, (top, _), _) = true;
invalid_mem_for_multiple (_, (bot, _), _) = true;
invalid_mem_for_multiple (_, (_, top), _) = true;
invalid_mem_for_multiple (_, (_, bot), _) = true;
invalid_mem_for_multiple (startreg, (a, b), 4) = (drop(b) - drop(a)) = ((32-startreg) * 4);
invalid_mem_for_multiple (_, _, _) = true;

// load the stuff: startregister, startmemaddress
load_multiple :: equations, snum, snum -> equations;
load_multiple (_, 32, _) = [];
load_multiple (eq, reg, addr) = let dvar = (v_reg, reg, 4); // destination register
      values = find_values (eq, (v_mem, addr, 4)); // content of memory cell to load
   in if (values = [])
        then load_multiple (eq, reg+1, addr+4)
        else (dvar, load_values (values, 4, false)) : load_multiple (eq, reg+1, addr+4)
      endif;

// store the stuff: startregister, startmemaddress
store_multiple :: equations, snum, snum -> equations;
store_multiple (_, 32, _) = [];
store_multiple (eq, reg, addr) = let dvar = (v_mem, addr, 4); // destination memory cell
       values = find_values (eq, (v_reg, reg, 4)); // content of register to store
   in if (values = [])
        then store_multiple (eq, reg+1, addr+4)
        else (dvar, store_values (values, 4)) : store_multiple (eq, reg+1, addr+4)
      endif;

//
// load value from memory, generic (perhaps with update)
//
load_from_mem :: equations,str,str,imacc,bool,bool -> equations;
load_from_mem (eq, dst, src, mem, signext, update)
 = let dvar = name_to_var(dst); // destination register
       values = find_values (eq, macc_to_var(mem!1!4, mem!2!4)); // content of memory cell to load
   in if (dvar!1!3 = v_invalid || values = [])
        then  [] // nothing to do, we don't know the source stuff
        else [(dvar, load_values (values, mem!2!4, signext))] // add information about destination var
      endif;

//
// store value in memory, generic (perhaps with update)
//
store_to_mem :: equations,str,str,imacc,bool -> equations;
store_to_mem (eq, dst, src, mem, update)
 = let dvar = macc_to_var(mem!3!4, mem!4!4); // destination memory cell
       values = find_values (eq, name_to_var(src)); // content of register to store
```

```
   in if (dvar!1!3 = v_invalid || values = [])
        then [] // nothing to do, we don't know the source stuff
        else [(dvar, store_values (values, dvar!3!3))] // add information about destination var
      endif;

//
// load value from memory, generic (perhaps with update)
//
do_sign_extend :: equations,str,str,snum,snum -> equations;
do_sign_extend (eq, dst, src, from, to)
 = let dvar = name_to_var(dst); // destination register
       values = sign_extend (find_values (eq, name_to_var(src)), from, to); // content of memory cell to load
   in if (dvar!1!3 = v_invalid || values = [])
        then  [] // nothing to do, we don't know the source stuff
        else [(dvar, values)] // add information about destination var
      endif;

//
// addi helper
//
addi_helper :: equations,str_8,str_8,snum -> equations;
addi_helper (eq, dsts, srcs, zeroreg)
 = let dvar = name_to_var(dsts!1!8); // destination register
       incr = str_to_snum (srcs!3!8); // increment
       values = find_values (eq, name_to_var(srcs!2!8)); // values to increment
   in if (dvar!1!3 = v_invalid)
        then [] // nothing to do, we don't know the source stuff
        else if ((name_to_var(srcs!2!8))!2!3 = zeroreg) // handle register 0 special if source
               then [(dvar, [((v_none, 0, 4), (incr, incr), 4, s_notext)])]
               else if (values = [] || add_to_values (values, incr) = []) then []
                       else [(dvar, add_to_values (values, incr))] endif
             endif
        endif;

//
// add helper
//
add_helper :: equations,str_8,str_8,snum,snum,snum -> equations;
add_helper (eq, dsts, srcs, block, instruction, context)
  = if (find_values(eq, name_to_var(srcs!2!8)) != []
        && is_restricted(register_value (str_to_register(srcs!3!8), block, instruction, context)))
      then add_right (name_to_var(dsts!1!8), find_values(eq, name_to_var(srcs!2!8)),
                      register_value (str_to_register(srcs!3!8), block, instruction, context))
      else if (dsts!1!8 = srcs!3!8)
           then add_right (name_to_var(dsts!1!8), find_values(eq, name_to_var(srcs!3!8)),
                           register_value (str_to_register(srcs!2!8), block, instruction, context))
           else []
      endif
    endif;

//
// subf helper
//
subf_helper :: equations,str_8,str_8,snum,snum,snum -> equations;
subf_helper (eq, dsts, srcs, block, instruction, context)
  = if (is_restricted(register_value (str_to_register(srcs!2!8), block, instruction, context)))
      then add_right (name_to_var(dsts!1!8), find_values(eq, name_to_var(srcs!3!8)),
                      negate_value (register_value (str_to_register(srcs!2!8), block, instruction, context)))
      else []
    endif;

//
// helpers for add/subf helper
//
is_restricted :: ivalue -> bool;
```

```
is_restricted ((bot, _)) = false;
is_restricted ((top, _)) = false;
is_restricted ((_, top)) = false;
is_restricted ((_, bot)) = false;
is_restricted (_) = true;

add_right :: var,var_values,ivalue -> equations;
add_right (dvar, values, addv)
  = if (dvar!1!3 = v_invalid || values = [] || !is_restricted(addv))
      then []
      else let nv = add_interval_to_values (values, drop(addv!1!2), drop(addv!2!2));
           in if (nv = []) then [] else [(dvar, nv)] endif
    endif;

add_interval_to_values :: var_values, snum, snum -> var_values;
add_interval_to_values ([], _, _) = [];
add_interval_to_values ((v, (c1, c2), w, s) : vs, a, b)
  = if (overflow (c1, a) || overflow (c2, b))
      then []
      else (v, (c1+a, c2+b), w, s) : add_interval_to_values (vs, a, b)
    endif;

//
// this var get's stored into cell with given width
// adjust all var_values
//
store_values :: var_values,snum -> var_values;
store_values ([], _) = [];
store_values ((v, c, w, s) : vs, nw)
  = if (nw > w) then (v, c, w, s) : store_values (vs, nw)
    else (v, c, nw, s_notext) : store_values (vs, nw) endif;

//
// this var get's loaded into register, perhaps even sign extended!!
// adjust all var_values
//
load_values :: var_values,snum,bool -> var_values;
load_values ([], _, _) = [];
load_values ((v, c, w, s) : vs, lw, false)
  = if (w < lw && s != s_notext) then (v, c, w, s_mixed) : load_values (vs, lw, false)
                                 else (v, c, w, s_notext) : load_values (vs, lw, false) endif;
load_values ((v, c, w, s) : vs, lw, true)
  = if (w < lw && s != s_ext) then (v, c, w, s_mixed) : load_values (vs, lw, true)
                              else (v, c, w, s_notext) : load_values (vs, lw, true) endif;

//
// move helper (for the special "or x a a" move)
//
or_move_helper :: equations,str_8,str_8,imacc -> equations;
or_move_helper (eq, dsts, srcs, mem)
 = let dvar = name_to_var(dsts!1!8); // destination register
       s1var = name_to_var(srcs!2!8);
       s2var = name_to_var(srcs!3!8);
       values = find_values (eq, name_to_var(srcs!2!8)); // values to move
   in if (dvar!1!3 = v_invalid || values = [] || s1var != s2var)
        then [] // nothing to do, we don't know the source stuff
        else [(dvar, values)] // add information about destination var
      endif;

//
// move helper (straight register move!)
//
reg_move_helper :: equations,str,str -> equations;
reg_move_helper (eq, dst, src)
 = let dvar = name_to_var(dst); // destination register
```

```
       values = find_values (eq, name_to_var(src)); // values to move
   in if (dvar!1!3 = v_invalid || values = [])
        then [] // nothing to do, we don't know the source stuff
        else [(dvar, values)] // add information about destination var
      endif;

//
// DONE ;)
//
```

# Literaturverzeichnis


[Abs05a] AbsInt Angewandte Informatik GmbH. *aiSee Manual*, 2005.

[Abs05b] AbsInt Angewandte Informatik GmbH. *aiT Manual*, 2005.

[Abs05c] AbsInt Angewandte Informatik GmbH. *PAG User's Manual*, 2005.

[All70] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[HLS00] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-case execution time analysis for digital signal processors. In *Proceedings of the European Signal Processing Conference 2000 (EUSIPCO 2000)*. Space Systems Finland Ltd, 2000.

[HS02] Niklas Holsti and Sami Saarinen. Status of the bound-T WCET tool. Space Systems Finland Ltd, 2002.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Mar95] Florian Martin. Entwurf und Implementierung eines Generators für Datenflußanalysatoren. Diplomarbeit, Universität des Saarlandes, Fachbereich 14, 1995.

[Mar98] Florian Martin. PAG – An Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[Mar99] Florian Martin. *Generation of Program Analyzers*. PhD thesis, Universität des Saarlandes, 1999.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*. Springer, March/April 1998.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Pug94] William Pugh. Counting solutions to Presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–134, 1994.

[San04] Christer Sandberg. Inspection of industrial code for syntactical loop analysis. In *WCET 2004 Workshop*, Catania, July 2004.

[Sch05] Marc Schlickling. Generisches Slicing auf Maschinencode. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes, Saarbrücken, 2005.

[SHW98] Mikael Sjödin, Christopher Healy, and David Whalley. Bounding loop iterations for timing analysis. In *RTAS '98: Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, Washington, DC, USA, 1998. IEEE Computer Society.

[Sic97] Martin Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diplomarbeit, Universität des Saarlandes, Fachbereich 14, 1997.

[SP81] Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[The98] Henrik Theiling. Über die Verwendung ganzzahliger linearer Programmierung zur Suche nach längsten Programmpfaden. Technical report, 1998.

[WM97] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung, 2. Auflage*. Springer, 1997.

# Stichwortverzeichnis

**A**

A380 .... 89
AbsInt .... 1
Abstrakte Interpretation .... 3, 10
Airbus .... 89
aiSee .... 30
aiT .... 1, 28
Ausdruck .... 45

**C**

Compiler
  DiabData .... 90
  GNU .... 90
CRL .... 29
CRL-Repräsentation .... 30

**D**

daan .... 29
Datenflussanalyse .... 11
  interprozedural .... 17
  intraprozedural .... 17
Datenflussproblem .... 15
  distributives .... 15
  Koinzidenz .... 16
  Korrektheit .... 16
  monotones .... 15
  Rückwärtsproblem .... 16
  Schnittproblem .... 17
  Vereinigungsproblem .... 17
  Vorwärtsproblem .... 16
DATLA .... 24
Dominanz .... 57

**E**

erweiterter Supergraph .... 20

**F**

Feedback-Iteration .... 29
Fixpunkt .... 8
Fixpunktiteration .... 7
FULA .... 24
Funktionen
  monotone .... 6
  stetige .... 6

**G**

Galois Einbettung .... 7
Galois Verbindung .... 7
Galoistheorie .... 7
geschachtelte Schleifen .... 34
Gleichung .... 45
Gleichungselimination .... 62
Grapherweiterung .... 20
Guard .... 74

**I**

Interprozedurale Analyse
  Call-String-Ansatz .... 17
  Supergraph .... 17
Invariantenanalyse .... 44
IVALA .... 44

**K**

Ketten
  $\omega$-Kette .... 5
  aufsteigende Kettenbedingung .... 6
  streng aufsteigende $\omega$-Kette .... 5
Konstante .... 45
Kontext .... 20
Kontrollflussgraph .... 14

## L


libloopy . . . . . 69
Linux . . . . . 90
Load-Multiple . . . . . 78
lokale Konsistenz . . . . . 10


## M


MFP-Lösung . . . . . 16
MOP-Lösung . . . . . 15


## O


Omega-Bibliothek . . . . . 57
Ordnung
  partielle . . . . . 4
  vollständige partielle . . . . . 5


## P


PAG . . . . . 19
Pattern-Matching . . . . . 85
Pfad . . . . . 14
Pfadsemantik . . . . . 15
PowerPC . . . . . 73
Präfixpunkt . . . . . 8
Presburger-Arithmetik . . . . . 57
Prozessorregister . . . . . 41


## S


Schachtelungstiefe-Iteration . . . . . 35
Schleife . . . . . 27
Schleifengrenze . . . . . 28
Schleifeniteration . . . . . 27
Schleifentest . . . . . 47
Schleifentransformation . . . . . 22
Schleifenzähler
  Definition . . . . . 42
Schranke
  größte untere . . . . . 5
  kleinste obere . . . . . 5
  obere . . . . . 5
  untere . . . . . 5
Shortcut-Evaluation . . . . . 84
SPARC . . . . . 86
Speicherzelle . . . . . 41


## V


Variable . . . . . 41
Variablenwert . . . . . 42
Verband
  dualer . . . . . 6
  vollständiger . . . . . 6
Verbandstheorie . . . . . 4
Vereinigung . . . . . 5
VIVU-Grapherweiterung . . . . . 23


## W


WCET . . . . . 1
WCET-Analysator . . . . . 28
Widening . . . . . 9
Widening-Operator . . . . . 9


## Z


Zählschleifen . . . . . 33