Kombinierte statische Ermittlung von Zeigerzielen, Kontroll- und Datenfluss

Von der Fakultät
Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Stefan Staiger-Stöhr

aus Stuttgart in Baden-Württemberg

Hauptberichter:	Prof. Dr. E. Plödereder
Mitberichter:	Prof. Dr. R. Koschke
Tag der mündlichen Prüfung:	9. Dezember 2009

Institut für Softwaretechnologie der Universität Stuttgart

2009

Für Christina,
meinen Stern auf Erden

Copyright 2009 Stefan Staiger-Stöhr

Danksagungen

Die Erstellung einer Dissertation verlangt viel Aufwand und Arbeit. Das führt dazu, dass man als Autor zum Einen weniger Zeit als sonst für andere hat, und zum Anderen auch auf Hilfe hier und da angewiesen ist. Daher möchte ich mich an dieser Stelle bei den Menschen bedanken, die zur erfolgreichen Vollendung dieser Arbeit in der einen oder anderen Weise beigetragen haben.

Diese Arbeit entstand während meiner Tätigkeit am Lehrstuhl für Programmiersprachen und Übersetzerbau der Universität Stuttgart. Dem Lehrstuhlinhaber Herrn Prof. Dr. Plödereder danke ich ganz besonders für die Betreuung und Unterstützung dieser Arbeit sowie für seine Anregungen und die Diskussionen zum Thema. Herrn Prof. Dr. Koschke gilt mein Dank für die Tätigkeit als Mitberichter sowie für die Diskussionen im Rahmen der Bauhaus-Meetings.

Den Mitarbeitern des Lehrstuhls gilt mein Dank für die jahrelange gute Zusammenarbeit sowie für unsere zahlreichen Fachgespräche zu den alltäglichen und wissenschaftlichen Herausforderungen im Bauhaus-Projekt. Gleiches gilt für die ehemaligen Mitarbeiter, die nun mit der Firma Axivion ebenfalls viele Beiträge dazu leisten, dass Bauhaus läuft und die Infrastruktur für weitergehende Projekte wie die hier vorgestellte Analyse nutzbar ist.

Mein besonderer Dank gilt meiner Frau Christina, die mich immer unterstützt und geduldig die Arbeitszeiten eines Doktoranden ertragen hat. Eine Leben ohne sie kann ich mir nicht mehr vorstellen. Schließlich möchte ich auch meinen Eltern danken, die mich stets unterstützt und das Studium ermöglicht haben.

Inhaltsverzeichnis

Danksagungen **5**

Abkürzungsverzeichnis **15**

Zusammenfassung **17**

Abstract **19**

Tabellenverzeichnis **22**

Abbildungsverzeichnis **29**

1 Einleitung **31**
1.1 Motivation der Analyse 32
1.2 Thesen und Beiträge der Dissertation 34
1.3 Aufbau der Dissertation 39

2 Grundlagen und verwandte Arbeiten **43**
2.1 Kontrollfluss 44
2.2 Datenfluss 47
2.2.1 Spezifikation von Datenfluss-Problemen 48
2.2.2 Frameworks zur Lösung 51
2.2.3 Datenfluss-Darstellungen 53
2.3 Zeigeranalysen 55
2.3.1 Allgemeines 55
2.3.2 Fluss- und kontext-insensitive Analysen 57

2.3.3 Fluss-Sensitivität 60
2.3.4 Kontext-Sensitivität 67
2.3.5 BDD-basierte Zeigeranalysen 70

I Die Kernanalyse 73

3 Der Rahmen für die Analyse 75
3.1 Objekte und ihre Repräsentation 76
3.1.1 Abbildung auf Analyse-Objekte 77
3.1.2 Schwache Aktualisierungen 82
3.1.3 Zeigerziele 83
3.1.4 Datenstrukturen 85
3.2 Repräsentation eines Programmes 89
3.2.1 Lokaler Kontrollfluss und Instruktionen 89
3.2.2 Nicht initialisierte Objekte 94
3.3 Interprozedurale SSA-Form 95
3.3.1 Erweiterung der klassischen SSA-Form 95
3.3.2 Zwingende und potentielle Seiteneffekte 102
3.3.3 Dominanz und ϕ-Knoten 104
3.3.4 Objektgraphen 105

4 Kombinierte Zeiger-, Kontroll- und Datenfluss-Analyse 107
4.1 Abstrakte Formulierung der Kernanalyse 107
4.2 Beispiel 111
4.3 Grobes algorithmisches Vorgehen 114
4.3.1 Graphaufbau 115
4.3.2 Propagierung 116
4.3.3 Finalisierung 116
4.4 Vergleich zu Andersens Analyse 117
4.5 Vergleich mit anderen Zeigeranalysen 118
4.5.1 Fluss-sensitive Zeigeranalysen 119
4.5.2 Vergleich mit weiteren Zeigeranalysen 120
4.5.3 Fluss-Sensitivität versus -Insensitivität 123

5 Umsetzung des Graphaufbau-Schrittes 125

5.1 Genereller fluss-sensitiver Graphaufbau 126
5.2 Induzierte ISSA-Knoten ergänzen 129
5.2.1 Induzierte ϕ-Knoten 130
5.2.2 Induzierte Ausgänge 130
5.3 Manipulation von Objektgraphen 131
5.3.1 Einfügen eines Knotens 131
5.3.2 Erweiterung und Korrektur eines Objektgraphen 134
5.4 Verbindungen zwischen Objektgraphen 136
5.5 Suche nach der gültigen Definition 139
5.5.1 Der CFG-Dominanzbaum als Basis 139
5.5.2 Suche im lokalen Objektgraphen 140
5.6 Rückwärts orientierte ISSA-Kanten 142

6 Umsetzung des Propagierungs-Schrittes 145

6.1 Variationsmöglichkeiten für die Propagierung von Zeigerzielen 145
6.1.1 Propagierungsgraph 147
6.1.2 Prinzipieller Algorithmus 149
6.1.3 Propagierungsrichtung 150
6.1.4 Inkrementell oder erschöpfend 151
6.2 Algorithmus für die Zyklenkontraktion 153
6.3 Framework für die Propagierung 156
6.3.1 Erschöpfendes Verfahren 156
6.3.2 Inkrementelles Verfahren 157
6.3.3 Aktionen für neu erkannte Verbindungen 162

7 Algorithmen für weitere Aufgaben 167

7.1 Initialisierung und Finalisierung 167
7.1.1 Vorbereitungen für nachfolgende Analysen 168
7.1.2 Pruning 169
7.1.3 Toten Code entfernen 172
7.2 Rekursive Unterprogramme 173

7.2.1 Verwendung eines einzigen Objekts 174
7.2.2 Verwendung verschiedener Objekte 177
7.3 Fluss-insensitive Variante 178
7.3.1 Veränderte Darstellung 178
7.3.2 Änderungen am Verfahren 180
7.3.3 Alternatives Vorgehen 182

8 Erkennung starker Aktualisierungen 183
8.1 Problematik 183
8.1.1 Ursachen für schwache Aktualisierungen 184
8.1.2 Arten von starken Aktualisierungen 186
8.1.3 Wirkungsbereich einer starken Aktualisierung 188
8.2 Lösungsansatz 189
8.2.1 Potentielle einfache starke Aktualisierungen 190
8.2.2 Potentiell betroffene Datenfluss-Kanten 190
8.2.3 Zusammengesetzte starke Aktualisierungen 193
8.3 Der Blockade-Graph 194
8.3.1 Interprozedurale Dominanz für 1-Definitionen 195
8.3.2 Zusammensetzung des Blockade-Graphen 195
8.3.3 Algorithmen zur Berechnung und Anwendung des Blockade-Graphen 198
8.3.4 Indirekte Aufrufe 208
8.4 Beispiel 208
8.5 Umgang mit Fehlern 211
8.5.1 Eingrenzung der fehlerhaften Dereferenzierungen 212
8.5.2 Fortsetzung der Analyse 213

9 Kontext-abhängige Propagierung 217
9.1 Problematik 218
9.1.1 Ungenauigkeiten der Kernanalyse 218
9.1.2 Beispiel 220
9.2 Das IFDS-Framework 221
9.2.1 Der erweiterte ICFG 223

9.2.2 Das Tabulationsverfahren . . . 225
9.3 IFDS auf dem ISSA-Graphen . . . 229
9.3.1 Vergleich zu Toks Arbeit . . . 230
9.3.2 Der erweiterte ISSA-Graph . . . 231
9.3.3 Modifikationen am Tabulationsverfahren . . . 232
9.4 IFDS für den Propagierungs-Schritt . . . 235
9.5 Ein alternativer Drei-Stufen-Ansatz . . . 241
9.5.1 Erste Stufe: Same-Level-Pfade . . . 243
9.5.2 Zweite Stufe: Kontext-unabhängige Propagierung . . . 246
9.5.3 Dritte Stufe: Kontext-abhängige Propagierung . . . 247
9.5.4 Anwendung auf das Beispiel . . . 247
9.5.5 Inkrementelle Propagierung . . . 249
9.5.6 Datenstrukturen . . . 253
9.6 Kontext-(un)abhängige Zeigerziele . . . 253
9.6.1 Heapmodellierung . . . 254
9.6.2 Kontext-abhängiger Datenfluss . . . 255

II Evaluation 259

10 Theoretische Evaluation 261
10.1 Terminierung und Korrektheit . . . 261
10.1.1 Terminierung . . . 262
10.1.2 Korrektheit . . . 262
10.2 Größe der Resultate . . . 271
10.3 Laufzeit . . . 275
10.4 Kosten mit IFDS für die Propagierung . . . 279
10.4.1 Größe der Resultate . . . 279
10.4.2 Laufzeit . . . 280
10.4.3 Vergleich zum Framework-Ansatz . . . 284

11 Empirische Evaluation 287
11.1 Ziele der Evaluation . . . 287
11.2 Evaluationsszenario . . . 289

11.2.1 Integration in Bauhaus . . . 289
11.2.2 Ausprägungen der Analyse . . . 291
11.2.3 Hardware-Umgebung . . . 292
11.3 Methodik . . . 292
11.4 Analysierte Programme . . . 295

12 Evaluation der Resultate 305
12.1 Zeigeranalyse . . . 305
12.2 Kontrollfluss-Analyse . . . 321
12.3 Datenfluss-Analyse . . . 329
12.3.1 Starke und schwache Aktualisierungen . . . 337
12.3.2 Größenreduktionen . . . 345

13 Evaluation der Skalierbarkeit 353
13.1 Laufzeit . . . 353
13.2 Speicherbedarf . . . 366
13.3 Blockade-Graph und Relaxation . . . 370
13.4 Vergleich der Implementierungsvarianten . . . 376
13.4.1 Vorwärts-Propagierung . . . 377
13.4.2 Erschöpfende Zyklenkontraktion . . . 377
13.4.3 Inkrementell ohne Zyklenkontraktion . . . 381
13.4.4 Erschöpfende versus inkrementelle Propagierung . . . 386
13.5 Zusammenfassung der Evaluation . . . 389

III Schluss 395

14 Ansätze für weitere Arbeiten 397
14.1 Präzisionssteigerungen . . . 397
14.1.1 Kontext-abhängiger Datenfluss . . . 398
14.1.2 Relationale Probleme . . . 399
14.2 Maßnahmen für die Skalierbarkeit . . . 400
14.2.1 BDDs . . . 401
14.2.2 Einbeziehung der Zielanalyse . . . 401

14.2.3 Parallelisierung 401
14.2.4 Andere Beschleunigungen 402
14.2.5 Speicherreduktion für große Programme 403
14.3 Integration von Analysen und Transformationen 404
14.4 Adaptive Analyse 405
14.5 Unterstützung weiterer Sprachkonstrukte 405
14.5.1 Objektorientierung 406
14.5.2 Exceptions 406
14.5.3 Parallelität 407

15 Zusammenfassung und Abgrenzung **409**
15.1 Zusammenfassung und Fazit 409
15.2 Die neue Analyse im Vergleich 411
15.2.1 Vergleich zu fluss-insensitiven Verfahren 411
15.2.2 Nutzung des IDFG anstelle des ICFG 413
15.2.3 Nutzung der ISSA-Darstellung 415
15.2.4 Behandlung von starken Aktualisierungen 416
15.2.5 Realisierung der Kontext-Beachtung 416
15.2.6 Skalierbarkeit 417

Literatur **419**

Stichwortverzeichnis **430**

Abkürzungsverzeichnis

AST	Abstrakter Syntaxbaum
BDD	Binäres Entscheidungsdiagramm
CFG / ICFG	Intra-/interprozeduraler Kontrollflussgraph
CS / CI	Kontext-sensitiv/-insensitiv
DAG	Gerichteter azyklischer Graph
DFG / IDFG	Intra-/interprozeduraler Datenflussgraph
FS / FI	Fluss-sensitiv/-insensitiv
IR / IML	Zwischendarstellung; IML ist die IR von Bauhaus
LHS / RHS	Linke/rechte Seite einer Zuweisung
LoC	Anzahl Codezeilen
SCC	Starke Zusammenhangskomponente (Zyklus)
SSU / CSU	Einfache / zusammengesetzte starke Aktualisierung
VDG	Value Dependence Graph nach Ruf

Zusammenfassung

Datenfluss-basierte statische Programmanalysen benötigen Informationen zu den an einer Programmstelle eintreffenden gültigen Definitionen. Dieses Wissen wiederum basiert auf dem Kontrollfluss. Zeiger-indirekte Operationen verschleiern jedoch beides, Kontroll- und Datenfluss: An solchen Operationen benötigt die Analyse Wissen über die potentiellen Zeigerziele, was aber seinerseits ein Datenfluss-Problem ist.

Die vorliegende Dissertation bespricht ein neues Verfahren, um diese drei zentralen Probleme in Kombination zu lösen. Der wesentliche Unterschied zu bekannten Zeigeranalysen ist dabei, die Probleme zusammen zu betrachten und zugleich zu lösen. Das erlaubt es uns, die Aufgabe nicht als Datenfluss-Problem auf der Kontrollfluss-Darstellung zu betrachten, sondern stattdessen als Grapherreichbarkeits-Problem auf der Datenfluss-Darstellung. Der Ansatz hierzu ist sehr einfach: Die wechselseitige Bestimmung von Datenfluss-Beziehungen und das Propagieren von Zeigerzielen löst iterativ das Problem. Die Analyse ist dabei fluss-sensitiv und beherrscht starke Aktualisierungen; eine fluss-insensitive Betrachtung führt auf die bekannte Zeigeranalyse von Andersen. Wie diese Arbeit zeigt, erreichen wir die höhere Genauigkeit zu den gleichen asymptotischen Kosten. Die Dissertation beweist die Korrektheit des neuen Verfahrens und präsentiert eine ausführliche Evaluation, in der es sich dank Zyklenkontraktion als moderat quadratisch herausstellt.

Schließlich präsentiert die Dissertation mit einer Erweiterung der neuen Analyse die erste fluss-sensitive Zeigeranalyse, welche starke Aktualisierungen unterstützt, die sogenannte *meet-over-all-valid-paths*-Lösung berechnet und dabei nur die Komplexität $\mathcal{O}(n^4)$ besitzt. Dies sind weitere klare Fortschritte zum aktuellen Stand der Forschung.

Abstract

Dataflow-based static program analyses need to know the reaching definitions arriving at the program's statements. This information in turn is based on control-flow knowledge. However, pointer-indirect operations obfuscate both control- and data-flow: for these operations, the analysis has to know the potential pointer targets – but this in itself is a data-flow problem.

This dissertation describes a new analysis to solve these three central problems in combination. In contrast to existing pointer analyses, we view the problems in combination and solve them together. This allows us to solve the task not as a data-flow problem on some control-flow representation, but instead as a graph-reachability problem on a data-flow representation. Our approach for this is very simple: iteratively determining data-flow relations and propagating pointer targets solves the problem. The new analysis is flow-sensitive and supports strong updates; a flow-insensitive realization yields the famous pointer analysis first described by Andersen. This work shows that we achieve the higher precision with the same asymptotic complexity. The dissertation also proves the new analysis' correctness and presents a thorough evaluation in which the analysis scales quadratically.

Finally, the dissertation describes an extension to the new analysis which yields the first flow-sensitive pointer analysis that supports strong updates and achieves the so-called *meet-over-all-valid-paths* precision within the time complexity of $\mathcal{O}(n^4)$. This is another considerable improvement over the state of the art.

Tabellenverzeichnis

3.1 Arten von Analyse-Objekten 81
3.2 Attribute pro Objekt 86
3.3 Verwendung des Attributs *Referenz* je nach Objektart 86
3.4 Analyse-Objekte für das strukturierte Objekt 87
3.5 Analyse-Objekte für *object* bei genesteten Strukturen 88
3.6 Objekte für das strukturierte Array 89
3.7 Attribute pro Instruktion 90
3.8 Arten von Instruktionen 91

11.1 Die analysierten Programme 296
11.2 Größe der analysierten Programme 297
11.3 Objekte 300
11.4 Unmittelbar Zeiger-relevante Instruktionen 304

12.1 Ziele an indirekten Definitionen und Verwendungen 306
12.2 Indirekte Definitionen und Verwendungen mit unbekannten Zielen 310
12.3 Indirekte Definitionen und Verwendungen mit Null als Ziel . . 311
12.4 Durchschnittliche Zahl an Zeigerzielen an indirekten Definitionen 313
12.5 Zeigerziele an indirekten Aufrufen 322
12.6 Indirekte Aufrufe mit unbekanntem Ziel 323
12.7 Durchschnittliche Zahl an Zielen an indirekten Aufrufen 325
12.8 Anteil der indirekten Aufrufe mit genau einem Ziel (%) 326
12.9 Knoten im finalen ISSA-Graphen 330
12.10 Kanten im finalen ISSA-Graphen 331

12.11 Indirekte Definitionen mit genau einem Ziel, welches starke Aktualisierungen erlaubt . . . 339
12.12 Indirekte Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt . . . 340
12.13 Anzahl allgemeiner Definitionen im ISSA-Graphen vor dem Pruning . . . 347

13.1 Schritte der Propagierung . . . 356
13.2 Laufzeit-Übersicht (Sekunden) . . . 358
13.3 Maximaler Platzbedarf (MB) . . . 367
13.4 Laufzeitgewinne als Faktoren (feld-insensitiv) . . . 391
13.5 Laufzeitgewinne als Faktoren (feld-sensitiv) . . . 392
13.6 Gesamtfaktoren der Laufzeitgewinne . . . 394

Abbildungsverzeichnis

3.1 ISSA-Graph zum Codebeispiel 97
3.2 ISSA-Graph zum Codebeispiel 100
3.3 Objektgraph von s im Beispiel 106

4.1 Allgemeiner Ablauf der Analyse 108
4.2 Beispielprogramm und initiale Knoten 112
4.3 Zwischenergebnisse der 1. Iteration 112
4.4 Zustand nach dem Graphaufbau der 2. Iteration. Zugleich auch Resultat vor dem Pruning (nach der 2. Iteration) 113
4.5 Resultat nach dem Pruning 114
4.6 Schematischer Vergleich des Ansatzes zu anderen Vorgehen . . 121
4.7 Schematischer Vergleich zum Ansatz von Hasti und Horwitz . 122

5.1 Laufendes Beispiel zur Illustration der Algorithmen 126
5.2 Fluss-sensitiver Graphaufbau 127
5.3 Induzierte ISSA-Knoten ergänzen 130
5.4 Neuen Knoten in einen Objektgraphen einfügen 132
5.5 Neuen ϕ-Knoten in einen Objektgraphen einfügen 133
5.6 Objektgraph erweitern und alte Kanten anpassen 135
5.7 Kante zwischen Objektgraphen anlegen 138
5.8 Suche nach der gültigen Definition (via CFG-Dominanzbaum) 140
5.9 Suche nach der gültigen Definition (via Objektgraph) 141
5.10 Verändertes Anlegen induzierter ϕ-Knoten bei rückwärts orientierten Kanten 144

6.1 Aktualisierung des SCC-DAG 154

6.2 Aktionen an einer alten SCC bzw. einem ISSA-Knoten 155
6.3 Grober Ablauf der erschöpfenden Propagierung 156
6.4 Grober Ablauf der inkrementellen Propagierung 157
6.5 Laufendes Beispiel zur Illustration der Algorithmen 158
6.6 Nutzung des Arrays `Targets` 159
6.7 Verwendung eines Arrays von Propagierungselementen 160
6.8 Aktionen bei einer neu erkannten Verbindung 163
6.9 Erweiterung des ICFG und des Aufrufgraphen 164

7.1 In Tarjans SCC-Erkennung integriertes Prunen 171
7.2 Beispiel zu lokalen Variablen in rekursiven Funktionen 173
7.3 Objektgraph von x, wenn nur ein Objekt benutzt wird 175
7.4 Objektgraph von x (nicht-lokal) und x' (lokal), wenn zwei Objekte benutzt werden 177
7.5 Fluss-insensitive Aktionen bei einer neuen Verwendung 180
7.6 Fluss-insensitive Aktionen bei einer neuen Definition 181

8.1 Blockade-Graph im Beispiel 198
8.2 Konstruktion des initialen Blockade-Graphen 201
8.3 Respektierung des Blockade-Graphen bei der Suche nach einer gültigen Definition über den CFG-Dominanzbaum 203
8.4 Separater Test auf eine Blockade bei der Suche nach der gültigen Definition über den Objektgraphen 205
8.5 Aktualisierung des Blockade-Graphen bei Erkennung des ersten Zeigerziels einer indirekten Definition 207
8.6 Beispielprogramm und initiale Knoten 209
8.7 Zwischenergebnisse der 1. Iteration 209
8.8 Zwischenergebnisse der 2. Iteration 210
8.9 Ergebnis vor und nach dem Pruning 211

9.1 Beispielprogramm zur Kontext-Abhängigkeit 221
9.2 Identität als Graph dargestellt 223
9.3 Gen-/Kill als Graph dargestellt 224
9.4 Konkatenation der Transferfunktions-Graphen 224

9.5 Das Original-Tabulationsverfahren 227
9.6 Behandlung interprozeduraler Kanten im Original-Tabulationsverfahren 228
9.7 Erweiterter ISSA-Graph (Ausschnitt zum Beispiel) 231
9.8 Behandlung kontext-(un)abhängiger Pfade 234
9.9 Das modifizierte Tabulationsverfahren 236
9.10 Behandlung interprozeduraler Kanten im modifizierten Tabulationsverfahren 237
9.11 Transferfunktions-Graph an der Zeigerziel-Quelle 239
9.12 Erweiterter ISSA-Graph (Ausschnitt zum Beispiel) 240
9.13 Berechnung von Same-Level-Pfaden und Summary-Kanten für den ISSA-Graphen mittels der IFDS-Ideen 244
9.14 Behandlung interprozeduraler Kanten bei der Berechnung von Same-Level-Pfaden und Summary-Kanten für den ISSA-Graphen 245
9.15 ISSA-Graph für p und q zum Beispiel. Summary-Kanten sind gestrichelt eingezeichnet. 248
9.16 CS-Graph für p und q zum Beispiel 250
9.17 Inkrementelle Initialisierung der Berechnung von Same-Level-Pfaden und Summary-Kanten für den ISSA-Graphen 252

11.1 Hilfsobjekte 301
11.2 Initiale und potentielle Zeigerziele 301

12.1 Ziele an indirekten Adressnahmen 308
12.2 Anteil der indirekten Definitionen mit genau einem Ziel (feld-insensitiv) 314
12.3 Anteil der indirekten Definitionen mit genau einem Ziel (feld-sensitiv) 314
12.4 Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel nano (feld-insensitiv) 316
12.5 Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel nano (feld-sensitiv) 316

12.6 Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel bison (feld-insensitiv) 318
12.7 Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel bison (feld-sensitiv) 318
12.8 Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel fvwm (feld-insensitiv) 320
12.9 Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel fvwm (feld-sensitiv) 320
12.10 Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel bison (feld-insensitiv) 327
12.11 Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel bison (feld-sensitiv) 327
12.12 Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel fvwm (feld-insensitiv) 328
12.13 Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel fvwm (feld-sensitiv) 328
12.14 Anteil der Definitionen am ISSA-Graphen (feld-insensitiv) . . 333
12.15 Anteil der Definitionen am ISSA-Graphen (feld-sensitiv) . . . 333
12.16 Anteil der Verwendungen am ISSA-Graphen (feld-insensitiv) . 334
12.17 Anteil der Verwendungen am ISSA-Graphen (feld-sensitiv) . . 334
12.18 Anteil der Phi-Knoten am ISSA-Graphen (feld-insensitiv) . . . 336
12.19 Anteil der Phi-Knoten am ISSA-Graphen (feld-sensitiv) 336
12.20 Indirekte Definitionen mit mehr als einem Ziel (feld-insensitiv) 338
12.21 Indirekte Definitionen mit mehr als einem Ziel (feld-sensitiv) . 338
12.22 Anteil indirekter Arraymodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-insensitiv) 342
12.23 Anteil indirekter Arraymodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-sensitiv) 342
12.24 Anteil der Heapmodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-insensitiv) 343

12.25 Anteil der Heapmodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-sensitiv) . . . 343
12.26 Direkte Array-Modifikationen . . . 344
12.27 Sonstige schwache direkte Aktualisierungen . . . 344
12.28 Durch Pruning entfernte allgemeine Definitionen (feld-insensitiv) 348
12.29 Durch Pruning entfernte allgemeine Definitionen (feld-sensitiv) 348
12.30 Anzahl Knoten im SCC-DAG (feld-insensitiv) . . . 350
12.31 Anzahl Knoten im SCC-DAG (feld-sensitiv) . . . 350
12.32 Größe der maximalen SCC (feld-insensitiv) . . . 351
12.33 Größe der maximalen SCC (feld-sensitiv) . . . 351

13.1 Anzahl Iterationen (feld-insensitiv) . . . 354
13.2 Anzahl Iterationen (feld-sensitiv) . . . 354
13.3 Laufzeit-Trend (fluss-sensitiv, feld-insensitiv) . . . 360
13.4 Laufzeit-Trend (fluss- und feld-sensitiv) . . . 360
13.5 Laufzeit-Trend (fluss-sensitiv, feld-insensitiv mit starken Akt.) 361
13.6 Laufzeit-Trend (fluss- und feld-sensitiv mit starken Akt.) . . . 361
13.7 Laufzeit der einzelnen Analysen-Teile (fluss-sensitiv, feld-insensitiv) 363
13.8 Laufzeit der einzelnen Analysen-Teile (fluss- und feld-sensitiv) 363
13.9 Laufzeit der einzelnen Analysen-Teile (fluss- und feld-insensitiv) 365
13.10 Laufzeit der einzelnen Analysen-Teile (fluss-insensitiv, feld-sensitiv) . . . 365
13.11 Trend beim Speicherbedarf (fluss-sensitiv, feld-insensitiv) . . . 368
13.12 Trend beim Speicherbedarf (fluss- und feld-sensitiv) . . . 368
13.13 Trend beim Speicherbedarf (fluss-sensitiv, feld-insensitiv mit starken Akt.) . . . 369
13.14 Trend beim Speicherbedarf (fluss- und feld-sensitiv mit starken Akt.) . . . 369
13.15 Knoten im initialen Blockade-Graphen . . . 371
13.16 Kanten im initialen Blockade-Graphen . . . 371
13.17 Maximal blockierte Kanten . . . 372
13.18 Blockierende indirekte Definitionen im Fixpunkt . . . 373

13.19 Final als Fehler eingestufte indirekte Definitionen 373
13.20 Effekt der Relaxation: Zusätzliche allgemeine Definitionen nach dem (ersten) Fixpunkt 375
13.21 Effekt der Relaxation: Zusätzlich benötigte Iterationen 375
13.22 Laufzeit-Vergleich zur Propagierungsrichtung (feld-insensitiv) . 378
13.23 Laufzeit-Vergleich zur Propagierungsrichtung (feld-sensitiv) . . 378
13.24 Laufzeit-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-insensitiv) 379
13.25 Laufzeit-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-sensitiv) 379
13.26 Speicherbedarf-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-insensitiv) 380
13.27 Speicherbedarf-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-sensitiv) 380
13.28 Laufzeit-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-insensitiv) 382
13.29 Laufzeit-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-sensitiv) 382
13.30 Laufzeit-Trend (feld-insensitiv, inkrementell ohne Zyklen) . . . 383
13.31 Laufzeit-Trend (feld-sensitiv, inkrementell ohne Zyklen) 383
13.32 Laufzeit-Trend (feld-insensitiv mit starken Akt., inkrementell ohne Zyklen) 384
13.33 Laufzeit-Trend (feld-sensitiv mit starken Akt., inkrementell ohne Zyklen) 384
13.34 Speicherbedarf-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-insensitiv) 385
13.35 Speicherbedarf-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-sensitiv) 385
13.36 Laufzeit-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-insensitiv) 387
13.37 Laufzeit-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-sensitiv) 387

13.38 Speicherbedarf-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-insensitiv) 388
13.39 Speicherbedarf-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-sensitiv) 388

Kapitel 1

Einleitung

Software ist heutzutage in immens viele Anwendungsbereiche vorgedrungen und erfüllt immer komplexere Aufgaben. Dabei wird auch die Software selbst immer umfangreicher und komplexer. Dies bringt Fehler mit sich und erschwert es dem Menschen und Programmen, die Software zu verstehen. Für den Menschen ist dieses Programmverständnis in der Wartung wichtig, für einen Übersetzer dagegen, um mögliche und erlaubte Optimierungen zu erkennen und korrekt durchzuführen. Als Hilfsmittel für beide, Mensch und Übersetzer, haben sich *statische Analysen* profiliert. Statisch sind sie im Unterschied zu dynamischen Analysen deshalb, weil sie das Programm nicht ausführen, sondern lediglich seinen Quellcode (in geeigneter Form) betrachten und damit Aussagen über *alle* möglichen Ausführungen treffen können. Diese Analysen sammeln dabei Informationen über das betrachtete Programm, welche für die Beantwortung von Fragen des Wartungsingenieurs oder eben als Rahmen für Optimierungen zum Einsatz kommen.

Diese Dissertation behandelt eine solche statische Analyse zur Beantwortung zentraler Fragen zu Programmen. Dabei geht es um Aussagen über das *Verhalten* der Programme; eine Betrachtung der Programmstruktur, d.h. eine Architektur-Analyse, interessiert uns in dieser Arbeit nicht.

1.1 Motivation der Analyse

Das Programmverhalten, also eine Ausführung des Programmes, ist mit der Sichtweise einer operationalen Semantik eine Folge von im Quellcode festgelegten Aktionen. Diese Aktionen betreffen zumeist irgendwelche *Objekte*, indem sie auf diese zugreifen oder sie modifizieren. Objekte sind dabei auf unterster Ebene betrachtet Intervalle im (virtuellen) Speicher. Programme in einer höheren Sprache haben jedoch eine abstraktere Sicht über diverse Sprachkonzepte wie Variablen, deren Einsatz wir später noch näher beleuchten. Wenn wir also das Programmverhalten untersuchen wollen, so sind wir an den möglichen Abfolgen von Aktionen interessiert, an den von den Aktionen betroffenen Objekten sowie an den Wirkungen der einzelnen Aktionen. Für das Ziel des Programmverstehens müssen statische Analysen daher Informationen zu diesen Punkten liefern.

Die damit skizzierten Aufgaben sind im Allgemeinen jedoch unentscheidbar. Neben den der Analyse unbekannten Eingaben für die Programmausführung sind dabei hauptsächlich die wertebezogenen Fragen nach Verzweigungen sowie nach den Werten der Objekte als Operanden von Aktionen gemäß dem Satz von Rice unentscheidbar [Rice 1953]. Die Analyse kann schon allein deswegen keine perfekten Antworten liefern; stattdessen müssen wir uns mit möglichst präzisen Näherungen begnügen. Dies kann z.B. eine Aufzählung der möglichen Werte sein, die Beschreibung von Eigenschaften der möglichen Werte oder auch eine Angabe zur Herkunft der Werte sowie den problematischen Programmstellen, die eine präzise Angabe verhindern.

Diese letzte Idee, den Weg eines Objekts durch das Programm mit entsprechenden Überschätzungen oder Vereinfachungen zu betrachten, wird eine statische Analyse im Kern wohl verfolgen müssen, um zu einer Eingrenzung der möglichen Werte zu gelangen. Für eine möglichst genaue Bestimmung der Wege ist dabei ein Verständnis der möglichen Ausführungsreihenfolgen der Quellcode-Anweisungen wichtig. Diese Aufgabe ist unter dem Begriff der *Kontrollfluss-Analyse* bekannt und eines der Ziele unserer kombinierten Analyse. Wirklich relevant sind dabei nur Konstrukte, bei denen die Reihenfolge nicht direkt zu erkennen ist, wie z.B. indirekte Aufrufe oder Ausnahmen,

denn die übrigen Fälle sind mit längst bekannten Techniken sozusagen optimal gelöst.

Haben wir eine Vorstellung vom Kontrollfluss, so sind wir darauf aufbauend interessiert, die möglichen Reihenfolgen auf einzelne Objekte zu übertragen: Wo wird dem Objekt ein Wert zugewiesen, und wo wird genau dieser Wert des Objekts benutzt? Analog zum Kontrollfluss spricht man hierbei vom *Datenfluss*, der zweiten Aufgabe unserer kombinierten Analyse. Dieser Datenfluss ist jedoch leider nicht allein mit der Kenntnis des Kontrollflusses zu beantworten. Eine weitere Schwierigkeit liegt nämlich darin verborgen, die in einer Anweisung betroffenen Objekte zu bestimmen. Diese können vor allem über Zeiger lediglich indirekt beschrieben sein, so dass wir erst eine Analyse zur Bestimmung der betroffenen Objekte benötigen. Im Falle von Zeigern spricht man von einer *Zeigeranalyse*, und dies ist schließlich die dritte Aufgabe der kombinierten Analyse.

Die kombinierte Analyse behandelt damit wichtige Probleme, denen sich statische Analysen zwingend ausgesetzt sehen, wenn sie Informationen zu den eingangs geschilderten Fragen liefern sollen: Kontrollfluss, Datenfluss, und Zeigerziele. Wir bezeichnen die Analyse aufgrund der Beschäftigung mit diesen Kernthemen daher als *Kernanalyse*. Sie setzt lediglich eine mit gängigen Techniken des Übersetzerbaus bestimmbare Zwischendarstellung des Programmes voraus.

Mit unserer neuen Analyse verfolgen wir dabei den Ansatz, diese Probleme nicht getrennt zu betrachten. Der Grund dafür ist, dass es wechselseitige Beziehungen zwischen den Problemen gibt. So ist beispielsweise für die Kontrollfluss-Analyse die Bestimmung der Ziele von Funktionszeigern nötig, so dass wir eigentlich für die Kontrollfluss-Analyse bereits die Resultate der Zeigeranalyse benötigen. Umgekehrt möchte die Zeigeranalyse die Ziele (also die Werte) von Zeigern abschätzen, was ein Datenfluss-Problem darstellt. Dies wiederum setzt für entsprechende Präzision den Kontrollfluss voraus, womit sich der Kreis schließt. Diesen zirkulären Abhängigkeiten begegnen wir nun mit einer ganzheitlichen Lösung, und unsere erfreulichen Resultate bestätigen uns in diesem Vorgehen.

1.2 Thesen und Beiträge der Dissertation

Der vorige Abschnitt hat die Wichtigkeit der Thematik illustriert und unser Ziel verdeutlicht, eine statische Analyse zur Ermittlung von Zeigerzielen, Kontroll- und Datenfluss zu erstellen. Natürlich existieren bereits Analysen für die angeführten Probleme, im Bereich der Zeigeranalysen sogar kaum überschaubar viele. Im Unterschied zu vielen dieser Publikationen betrachten wir das Problem der Zeiger jedoch nicht isoliert, sondern im Kontext der anderen beiden Aufgaben.

Dabei zeigt sich eine Verwandtschaft unserer neuen Analyse mit einem sehr bekannten Vertreter der Zeigeranalysen. Die Zeigeranalyse von *Andersen* [Andersen 1994] ist seit ihrer Publikation vielfach aufgegriffen und weiterentwickelt worden. Wir sehen sie aufgrund der zahlreichen nachfolgenden Publikationen, welche auf diesem Verfahren aufbauen, als die einflussreichste Zeigeranalyse überhaupt. Sie besticht durch einen einfachen Ansatz bei akzeptabler Skalierbarkeit.

Die Genauigkeit dieser Analyse leidet jedoch darunter, dass der Kontrollfluss für die Abschätzung der Zeigerziele nicht beachtet wird; mit dem Fachvokabular gesprochen ist die Analyse *fluss-insensitiv* (nähere Ausführungen hierzu und zu vielen anderen Grundlagen bietet Kapitel 2). Dies ist nicht bloß ein Kompromiss aus Skalierbarkeit und Präzision: Weder Andersen selbst noch andere Forscher seitdem haben eine Möglichkeit gesehen, diese Analyse mit ähnlicher Skalierbarkeit und ähnlichem Ansatz *fluss-sensitiv* umzusetzen, d.h. unter Beachtung des Kontrollflusses. Andersen hat hierzu lediglich Probleme aufgezählt, die ihm für eine fluss-sensitive Ausprägung im Wege standen [Andersen 1994, S. 148]. Natürlich existieren fluss-sensitive Zeigeranalysen (z.B. von Wilson [1997]) sowie verwandte Verfahren wie Escape-Analysen (z.B. von Whaley und Rinard [1999]). Diese verfolgen jedoch einen anderen Ansatz (z.B. mit Datenfluss-Gleichungen) und sind vielfach leider teuer.

Andersens fluss-insensitives Verfahren ist in der Praxis relevant, so dass man gleiches für eine fluss-sensitive Ausprägung annehmen kann, wenn diese ähnlich skaliert wie die fluss-insensitive Originalversion. Die vorliegende Dis-

sertation liefert hierzu einen Beitrag, indem sie eine fluss-sensitive Ausprägung mit der asymptotischen Komplexität des fluss-insensitiven Verfahrens beschreibt und empirisch evaluiert. Die Verwandtschaft zu Andersens Verfahren erlaubt es uns dabei auch, die Zyklenkontraktion als bekannte Optimierung des fluss-insensitiven Verfahrens für eine deutlich bessere Skalierbarkeit zu übertragen.

Die erste These dieser Dissertation fasst einen theoretischen Aspekt dieses Beitrags zusammen:

These 1 *Es existiert eine fluss-sensitive Ausprägung von Andersens Zeigeranalyse mit ebenfalls kubischer asymptotischer Komplexität.*

Dieser Beitrag besteht somit aus zwei Komponenten:

1. Das bislang ungelöste Problem, die wohl einflussreichste Zeigeranalyse bei ähnlicher Skalierbarkeit und Einfachheit fluss-sensitiv auszuprägen, wird gelöst. Damit eröffnet sich die Möglichkeit, die Vorteile von Andersens Analyse mit der höheren Genauigkeit zu kombinieren. Bekannte Ideen zur Optimierung des Originals können auf die neue Analyse übertragen werden, um so die Skalierbarkeit zu verbessern.

2. Die asymptotische Komplexität und damit die theoretische Skalierbarkeit bleibt beim Übergang auf den fluss-sensitiven Fall gleich. In beiden Fällen bestimmt dabei das Problem der dynamischen transitiven Hülle des Graphen, über den die Zeigerziele propagieren, die Komplexität. Für fluss-sensitive Zeigeranalysen ist bislang kein Verfahren mit besserer Komplexität und lediglich eines [Ruf 1995] mit behaupteter gleicher Komplexität bekannt.

Die Fluss-Sensitivität ist eine Möglichkeit, die Genauigkeit der ursprünglichen Analyse zu erhöhen. Eine andere Möglichkeit besteht darin, Zeigerziele pro Variable und Funktion zu bestimmen und sodann die Funktionen im jeweiligen Aufrufkontext zu unterscheiden (vgl. Literaturdiskussion in Kapitel 2). Forscher haben bereits verschiedene Varianten dieser im Fachjargon *Kontext-Sensitivität* genannten Idee für Andersens Verfahren untersucht.

Diese Varianten unterscheiden sich im jeweils gewählten Kompromiss aus Genauigkeit und Geschwindigkeit. Eine solche Variante ist hierbei die sogenannte *meet over all valid paths*-Lösung (MOVP) [Nielson u. a. 1999], womit die korrekte Beachtung der stapelartigen Aufrufsemantik gemeint ist (vgl. Kapitel 2). Reps [1998] hat diese für Andersens Analyse skizziert.

Der zweite Beitrag der Dissertation befasst sich nun damit, diese MOVP-Lösung auch für unsere fluss-sensitive Fassung der Analyse anzubieten und damit erneut eine Parallele zu den Arbeiten rund um Andersens Analyse zu schaffen. Dies führt uns auf die zweite These der Dissertation:

These 2 *Es gibt eine fluss-sensitive Zeigeranalyse, die sowohl starke Aktualisierungen beherrscht als auch die MOVP-Präzision (in der Propagierung) erreicht und die Komplexität $\mathcal{O}(n^4)$ besitzt.*

Dem Autor ist keine andere Zeigeranalyse bekannt, welche zugleich die genannte Präzision erreicht und diese – im Vergleich zu anderen derartigen Verfahren geringe – Komplexität aufweist.

Die aus der Literatur bekannten fluss-sensitiven Zeigeranalysen unterscheiden sich nicht nur im Hinblick auf die Dimension der Kontext-Sensitivität in ihrer Genauigkeit. Ein weiterer Unterschied besteht darin, wie gut die Analyse sogenannte *starke Aktualisierungen* beherrscht. Damit ist die Frage gemeint, wann die Analyse an einer direkten oder Zeiger-indirekten Zuweisung an ein Objekt x verhindern kann, dass vorausgehende Zeigerziele von x über die Zuweisung hinaus propagieren. Im Datenfluss-Jargon gesprochen fragen wir also nach den *Kill*-Mengen der Zeigeranalyse. Während fluss-insensitive Verfahren keine Reduktion der Zeigerziele mittels starker Aktualisierungen erreichen können, eröffnet sich für fluss-sensitive Zeigeranalysen hier die Möglichkeit, die Genauigkeit noch einmal zu erhöhen.

Die Dissertation beschreibt nun eine Technik, mit der die fluss-sensitive Ausprägung von Andersens Analyse optional starke Aktualisierungen beherrscht. Diese Technik lässt sich auch auf die Variante mit MOVP-Präzision anwenden. In beiden Fällen erhöht sich die asymptotische Komplexität dabei nicht, was wir in der dritten These festhalten:

These 3 *Sowohl die fluss-sensitive Andersen-Analyse aus These 1 als auch die Variante mit MOVP-Präzision aus These 2 können zusätzlich indirekte starke Aktualisierungen unterstützen, ohne dass sich die Komplexität der Verfahren erhöht.*

Die bislang formulierten Beiträge beschäftigen sich mit theoretischen Ergebnissen zur Problematik der Zeigeranalysen. Die Literatur berichtet jedoch vielfach, dass das asymptotische Laufzeitverhalten dieser Analysen oft nicht mit den Beobachtungen an realen Beispielen korreliert. So sind selbst theoretisch exponentielle Verfahren auf manche Programme mit akzeptablem Zeitbedarf anwendbar. Es hat sich daher eingebürgert, jede neu vorgeschlagene Zeigeranalyse auch empirisch zu evaluieren. Eine solche empirische Auswertung soll dabei helfen, den realen Zeitbedarf sowie die tatsächlich erreichbare Präzision besser zu verstehen. Sie kann darüber hinaus Hinweise liefern, welche Ursachen für einen Genauigkeits- oder Geschwindigkeits-Verlust verantwortlich sind.

Diesem etablierten Vorgehen schließt sich die Dissertation an. Sie beschreibt unsere Implementierung der vorgeschlagenen Analysen sowie die Experimente, mit denen wir diese evaluiert haben. So kann die Arbeit Messwerte zur Skalierbarkeit und Präzision liefern. Die Evaluation vergleicht diese auch mit den entsprechenden Werten für die ursprüngliche, fluss-insensitive Andersen-Analyse. Die folgende Aussage fasst einige der Resultate zusammen:

In der Praxis verhält sich die kontext-insensitive Analyse (mit ein paar Ausreißern in beide Richtungen) unter Verwendung einer inkrementellen Zyklenkontraktion moderat quadratisch. Das ergeben Messungen über einige Programme im Bereich von 2 KSLoC - 210 KSLoC.

Diese Ergebnisse wurden ohne Verwendung der derzeit beliebten, aber in der asymptotischen Komplexität nicht gut beherrschbaren BDDs erzielt. Damit erreicht die neue Analyse zugleich eine akzeptable asymptotische Komplexität und praktische Skalierbarkeit. Der Einsatz und Nutzen einer Zyklenkontraktion weist wiederum Parallelen zu den Erfolgen bei Andersens Analyse auf.

Der wesentliche Unterschied unserer neuen Analyse im Vergleich zu bekannten fluss-sensitiven Zeigeranalysen ist die Integration der Datenfluss-Analyse. Während viele Verfahren die Zeigeranalyse als Problem auf der Kontrollfluss-Darstellung betrachten, haben wir eine Möglichkeit gefunden, diese als Problem auf der Datenfluss-Darstellung zu lösen – ohne dass jene bereits vorab durch konservative Überschätzungen errichtet werden muss. Dies formulieren wir als weitere These:

These 4 *Zeigeranalyse und Datenfluss-Analyse können statt nacheinander auch in Kombination erfolgen.*

Die Kombination kommt dabei ohne eine Vorab-Überschätzung aus, wie sie bisherige Analysen benötigten, die eine Abkehr von der Kontrollfluss-Darstellung als Basis benutzen (vgl. Kapitel 2). Eine Folgerung daraus soll aufgrund des zentralen Unterschieds zu den bekannten Verfahren explizit als letzte These angeführt sein:

These 5 *Eine fluss-sensitive Zeigeranalyse kann anstatt als Datenfluss-Problem auf der Kontrollfluss-Darstellung auch effizient als Grapherreichbarkeits-Problem auf der tatsächlichen, nicht vorab überschätzten Datenfluss-Darstellung realisiert werden.*

Während viele Publikationen auf dem Gebiet der Zeigeranalysen *Optimierungen* von bekannten Ansätzen oder deren Übertragung auf andere Sprachen beschreiben, liefert die Dissertation einen generell neuen *Ansatz* zur Lösung des Problems.

Grundsätzlich sehen wir dabei gute Gründe, sich für eine Zeigeranalyse mit der von uns erreichten Präzision und Skalierbarkeit zu interessieren. In der Literatur zeigt sich beispielsweise der Trend, vermehrt praktikable Verfahren mit höherer Präzision zu betrachten. Während Ende des 20. Jahrhunderts die fluss-insensitiven Verfahren im Vordergrund standen, drängen inzwischen die fluss-sensitiven Verfahren vor. Ein Grund dafür ist sicherlich die heute größere Rechenleistung, aufgrund derer man sich auch mit langsameren Verfahren in der Praxis beschäftigen kann. Ein anderer Grund ist das verstärkte Aufkommen von Programmanalyse-Werkzeugen, die anders

als ein Compiler eine hohe Genauigkeit für vernünftige Resultate benötigen. Eine zwar schnelle, aber ungenaue Analyse ist dafür weitgehend wertlos. Zudem besteht die Chance, dass sich eine höhere Genauigkeit für die Präzision und Skalierbarkeit nachfolgender Analysen auszahlt, so dass sich die Kosten insgesamt nivellieren.

Der Trend zum Verlangen nach höherer Genauigkeit bei gleichzeitig stärkerer verfügbarer Rechenleistung setzt sich aller Wahrscheinlichkeit nach fort. Daher betrachten wir nicht nur die Fluss-Sensitivität als Maßnahme für höhere Präzision, sondern studieren mit den starken Aktualisierungen und der MOVP-Präzision noch präzisere Varianten.

1.3 Aufbau der Dissertation

Im nächsten Kapitel besprechen wir als Basis dieser Arbeit einige Grundlagen und betrachten verwandte Arbeiten. Auch hier bereits genutzte Fachtermini werden darin erläutert. Die weitere Dissertation gliedert sich dann in drei Teile. Die Kapitel dieser Teile tragen dabei zum Nachweis der in den Thesen formulierten Behauptungen bei oder runden das Werk ab.

Teil I demonstriert, wie die Betrachtung des Zeigeranalyse-Problems auf eine kombinierte Lösung mit den genannten positiven Eigenschaften führt. Dieser Teil beschreibt die neue Analyse sowie die Erweiterungen für indirekte starke Aktualisierungen und zur Beachtung der Aufrufsemantik.

Im Einzelnen schildert dabei Kapitel 3 zunächst den Rahmen der Analyse, d.h. die Eingabe, auf der sie arbeitet, sowie die gewünschte Ausgabe. Das Kapitel schildert auch, wie die Analyse Objekte modelliert. Danach präsentiert Kapitel 4 unseren allgemeinen Ansatz zur Betrachtung der Zeigeranalyse als Datenfluss-Problem. Darin zeigen wir, dass eine mögliche Ausprägung des Ansatzes die bekannte Analyse von Andersen ist. Das Thema der Dissertation, nämlich eine fluss-sensitive Ausprägung desselben Ansatzes, wird dazu im Vergleich gesehen. Anschließend zeigen die Kapitel 5 bis 7, wie diese Analyse algorithmisch umgesetzt werden kann. Hierzu besprechen die Kapitel Pseudo-Code für die Algorithmen und gehen dabei auf verschiedene Umsetzungsmöglichkeiten ein.

Kapitel 8 behandelt die Problematik der indirekten starken Aktualisierungen und präsentiert unseren Ansatz zur Lösung. Neben dem Konzept bietet das Kapitel auch Pseudo-Code für die Umsetzung dieser Lösung als Erweiterung der Kernanalyse.

Kapitel 9 analysiert die Ungenauigkeiten der Kernanalyse, welche durch ihre Kontext-Insensitivität entstehen. Als Standardlösung für eine dieser Ungenauigkeiten kennt die Wissenschaft das IFDS-Framework, welches wir in diesem Kapitel vorstellen und auf unsere Problematik anwenden. Damit kann die Analyse pro Iteration Datenfluss-Probleme unter Beachtung der Aufrufsemantik lösen, und dies setzen wir speziell für den Zeigeranalysen-Teil ein. Im Unterschied zur klassischen Kontrollfluss-basierten Formulierung nutzen wir nun aber das Framework auf der Datenfluss-Darstellung, was eine einfachere Spezifikation der Datenfluss-Probleme erlaubt und damit auch ein Problem umgeht, welches bislang die Anwendung von IFDS bei fluss-sensitiven Zeigeranalysen verhindert hat. Die Situation legt dann jedoch auch die zusätzliche Betrachtung einer alternativen Idee für den Zeigeranalysen-Teil nahe, welche sich als asymptotisch effizienter herausstellt.

Teil II analysiert das vorgeschlagene Verfahren und seine Erweiterungen. Die Dissertation bietet dabei sowohl eine theoretische Untersuchung (Kapitel 10) als auch eine empirische Untersuchung (Kapitel 11 bis 13).

Im theoretischen Teil beweisen wir die Korrektheit, Terminierung, Laufzeit und Ergebnisgröße des Verfahrens sowie seiner Erweiterungen. Das Kapitel 10 schließt damit den Nachweis der vorhin genannten Thesen dieser Arbeit ab.

Der empirische Teil untersucht die Messwerte zur Skalierbarkeit und den Resultaten der Analyse auf einem breiten Spektrum an realen Programmen. Zunächst klärt Kapitel 11 unser Evaluationsszenario und stellt die analysierten Programme vor. Anschließend untersucht Kapitel 12 die Resultate der Analyse auf diesen Programmen. Das Kapitel 13 betrachtet danach die Effizienz verschiedener Implementierungsvarianten der Analyse. Zusammen bieten diese Untersuchungen Einblicke ins Verhalten und die Anwendbarkeit der neuen Analyse. Außerdem zeigt ein Vergleich mit Andersens Verfahren die Kosten und den Gewinn der Übertragung auf ein fluss-sensitives Ver-

fahren. Am Ende des Kapitels schließen wir den Teil ab mit einer kurzen Zusammenfassung der empirischen Auswertung.

Teil III schließlich rundet die Behandlung der Themen ab, indem Ideen zu weiteren Arbeiten im Kontext der hier diskutierten Analysen angeführt werden (Kapitel 14). Außerdem rekapituliert Kapitel 15 die Beiträge der Dissertation im Kontext existierender und möglicher zukünftiger Analysen und Anwendungen. Damit kann der Leser den Beitrag der Dissertation besser einordnen.

Ein ausführliches Literaturverzeichnis am Ende der Arbeit bietet einen Einstieg für weiterführende Studien und hilft, die richtigen Publikationen zu den diversen Bezügen im Text zu finden. Die Tabellen zu den konkreten Messwerten der empirischen Evaluation finden sich aus Platzgründen nicht in dieser Arbeit selbst, sondern können von der Webseite [Staiger-Stöhr 2009] bezogen werden.

Kapitel 2

Grundlagen und verwandte Arbeiten

Bevor wir die neuen Ideen und Resultate dieser Arbeit diskutieren, streifen wir in diesem Kapitel zunächst durch die relevanten Publikationen, deren Thematik unsere Aufgaben betrifft. Unsere Arbeit steht auf vielen dieser Schultern, indem sie publizierte Ideen aufgreift und entweder direkt nutzt, weiterentwickelt oder fundierte Gegenvorschläge unterbreitet. Den Streifzug durch die Literatur nutzen wir dabei zugleich, um grundlegende Begriffe und Erkenntnisse einzuführen, die im Weiteren eine Rolle spielen.

Diese Kapitel soll damit das Fundament zum Verständnis der Dissertation liefern. Eine Abgrenzung zu den hier genannten Werken erfolgt erst am Schluss der Arbeit in Kapitel 15.2, wenn wir dann beide Welten – die aus der Literatur bekannte und die in der Dissertation geschaffene – kennen. Außerdem verweist der laufende Text jeweils auf Quellen für Ideen und benennt Unterschiede. In diesem Kapitel erwähnen wir nur an einigen Stellen als Vorgriff auf die Hauptteile, welche Punkte für unsere Analyse relevant sind und wo neue Beiträge erbracht werden.

Da wir uns mit einer zentralen Thematik beschäftigen, ist die Literaturmenge kaum überschaubar. Zwangsweise können wir hier also nur eine Auswahl besprechen. Da die Zeigeranalyse dabei am wichtigsten ist, stellt sie den größten Teil des Kapitels dar.

2.1 Kontrollfluss

Der intraprozedurale Kontrollfluss lässt sich bequem als Graph darstellen. Dieser Kontrollfluss-Graph (engl. *control-flow graph*, CFG; Allen [1970]; Hecht [1977]) existiert in verschiedenen Varianten, die sich in der Granularität der Knoten unterscheiden. Die feinste Granularität hat ein CFG, der jeden Teilausdruck als einen Knoten auffasst; dies trifft man bei hohen, syntaxbasierten Darstellungen wie IML [Raza u. a. 2006] an. Etwas gröber ist die Nutzung einzelner Instruktionen als Knoten, was bei niederen Darstellungen wie dem 3-Address-Code [Aho u. a. 1986] gegenüber der feinsten Stufe zu bevorzugen ist, da die Instruktionen hinreichend einfach sind und somit die Reihenfolge der Operanden klar ist. Am gröbsten ist die Variante, welche Grundblöcke als Knoten benutzt. Diese scheint das Standardmodell zu sein, weil sie die Größe des CFG reduziert. Knoop u. a. [1998] identifizieren dies jedoch heutzutage eher als Nachteil gegenüber einer Darstellung auf Anweisungsebene.

Üblicherweise setzt man voraus, dass ein CFG genau einen Eingang und genau einen Ausgang besitzt, um klar definierte Einstiegs- und Rücksprungs-Punkte zu haben. Wir behalten dieses Modell bei.

Der interprozedurale Kontrollfluss-Graph (engl. *interprocedural control-flow graph*, ICFG; Sharir und Pnueli [1981], [Landi 1992, S. 7]) ist eine Möglichkeit, interprozeduralen Kontrollfluss darzustellen. Dieser Graph verbindet die CFGs der einzelnen Unterprogramme, indem jede Aufrufstelle durch zwei Knoten ersetzt wird: einen unmittelbar vor dem Aufruf mit Kante zum Aufrufziel, und einen unmittelbar nach dem Aufruf, zu dem vom Rücksprung aus eine Kante führt. Indirekte Aufrufe mit mehreren Zielen ergeben pro Aufrufziel je ein solches Kantenpaar (bei weiterhin genau einem Knoten vor und nach dem Aufruf).

Eine weitere Datenstruktur für den interprozeduralen Kontrollfluss ist der Aufrufgraph (engl. *call graph*, CG). Dieser zeigt nur noch die Aufrufbeziehungen zwischen den einzelnen Unterprogrammen, nicht jedoch die Details der Unterprogramme selbst. Die Knoten des Aufrufgraphen sind darum Unterprogramme, die Kanten stellen Aufrufe dar. Dabei kann man entweder die einzelnen Aufrufstellen eines Unterprogrammes differenzieren oder vereinen.

Eine Schwierigkeit für den Kontrollfluss sind Ausnahmen (engl. *exceptions*) und der von ihnen verursachte zusätzliche Kontrollfluss. Hier ist neben der Analyse zur Ermittlung des Flusses auch die gewünschte Darstellung eher unklar. Diese Problematik betrachten wir in der vorliegenden Arbeit nur im Ausblick in Abschnitt 14.5. Ebenso gehen wir nur dort auf parallele Programme ein, da dies ein umfangreiches eigenes Thema darstellt.

Weitere kleine Schwierigkeiten für den Kontrollfluss sind Anweisungen oder Funktionen, welche das Programm sofort beenden (z.B. `exit()` in C). Wenn wir sie ignorieren, überschätzen wir den tatsächlichen Kontrollfluss (und damit auch den Datenfluss). Dies ist also konservativ und aufgrund des eher seltenen Auftretens auch nur wenig ungenau. Eine explizite Beachtung erfordert Überlegungen zur gewünschten Darstellung; die naheliegende Idee, eine Kante zum Knoten für das Programmende zu erstellen, erzeugt jedenfalls zusätzliche Ausgänge für einen CFG und sorgt somit für Sonderfälle, die in den Analysen beachtet werden müssen.

Ein Resultat der Kontrollfluss-Analyse ist auch, Unterprogramme und Teile von Unterprogrammen zu identifizieren, die definitiv nicht ausgeführt werden. Dieser sogenannte *tote Code* tritt häufig im Zusammenhang mit Bibliotheken auf, von denen eine Anwendung nur Teile benutzt. Auf Kontrollfluss aufbauende Analysen können toten Code ignorieren; ohne Kontrollfluss-Aussagen jedoch muss auch toter Code mitanalysiert werden und kann durchaus die Ergebnisse für lebendigen Code beeinträchtigen (z.B. bei einer flussinsensitiven Zeigeranalyse).

Auflösung indirekter Aufrufe

Das meistdiskutierte Problem im Zusammenhang mit der Bestimmung des interprozeduralen Kontrollflusses ist die Ermittlung von Zielen für indirekte Aufrufe. Damit sind sowohl Aufrufe über Funktionszeiger (inklusive Prozedurparametern) als auch virtuelle Aufrufe in objektorientierten Sprachen gemeint. Generell kann man hierzu eine allgemeine Zeigeranalyse einsetzen (vgl. Abschnitt 2.3), wenn man mit der eigentlich entstehenden zirkulären Abhängigkeit der Analysen zurecht kommt.

Publikationen hierzu berichten, dass geringe Präzision für C-Programme bereits für gute Aufrufgraphen ausreicht [Milanova u. a. 2004]; für Java wird jedoch ein wenig Kontextbeachtung empfohlen. Die Studie von Milanova hat jedoch nur acht C-Programme bis zu einer Größe von 26 KLoC betrachtet, so dass unklar bleibt, ob sich die Aussage auf große Programme und andere Sprachen übertragen lässt. Die Studie hat aber keine Zahlen präsentiert, so dass neue Resultate schwer damit vergleichbar sind.

Die einfachsten Ansätze überschätzen die Ziele grob. Für Funktionszeiger ist hier vor allem der Ansatz bekannt und beliebt, die Ziele dadurch zu begrenzen, dass nur Funktionen als Ziel auftreten können, von denen irgendwo die Adresse genommen wird. Damit erhält jeder Funktionszeiger-Aufruf die gleiche Zielmenge. In typsicheren Sprachen kann man diese noch anhand der Signatur des Funktionszeigers begrenzen.

Für virtuelle Aufrufe haben sich daneben einige spezialisierte Verfahren herauskristallisiert, welche die Klassenhierarchie und die überhaupt erzeugten Objekte in die Betrachtung einbeziehen. Die Klassenhierarchie-Analyse (engl. *class hierarchy analysis*, CHA) von Dean u. a. [1995] betrachtet neben der Signatur die Menge der Redefinitionen einer Methode; anhand des statischen Typs des Objektes, auf dem die Methode aufgerufen wird, grenzt dieses Verfahren die potentiellen Ziele ein. Präziser ist das RTA-Verfahren von Bacon und Sweeney [1996], das zusätzlich beachtet, von welchen Klassen überhaupt Instanzen erzeugt werden. Shivers [1991] hat (mit Bezug zu funktionalen Sprachen) den allgemeinen, präziseren Ansatz namens k-CFA (von engl. *control-flow analysis*) eingeführt, wobei k die Zahl der zur Kontextunterscheidung betrachteten Aufrufer angibt. Das Verfahren von Tip und Palsberg [2000] liegt in seiner Genauigkeit zwischen RTA und 0-CFA. Grove und Chambers [2001] präsentieren ein Framework für verschiedene Verfahren sowie einen empirischen Vergleich. Darin kommen sie zu dem Schluss, dass ein fast linearer Algorithmus bereits einen Großteil der Optimierungen erlaubt, welche ihre präzisesten Verfahren ermöglichen.

In dieser Dissertation verwenden wir keine separate Analyse für indirekte Aufrufe, sondern nutzen eine allgemeine Zeigeranalyse.

2.2 Datenfluss

Auf dem Kontrollfluss aufbauend interessiert der Datenfluss zwischen den Variablen sowie zwischen den verschiedenen Vorkommnissen der gleichen Variable. Für die Betrachtung führen wir zunächst einige grundlegende Begriffe ein. Ziel einer Datenfluss-Analyse ist dabei die Konstruktion eines *Datenfluss-Graphen* (DFG) bzw. eines *interprozeduralen Datenfluss-Graphen* (IDFG) analog zum Kontrollfluss.

Eine Zuweisung lässt den Wert der rechten Seite (*RHS* nach engl. *right-hand side*) auf die linke Seite (*LHS* nach engl. *left-hand side*) fließen. Wie im Compilerbau üblich, bezeichnen wir eine Zuweisung an ein Objekt dabei als *Definition* dieses Objekts. Das Auslesen des Wertes eines Objekts bezeichnen wir als *Verwendung* (engl. *use*). Eine Definition oder Verwendung ist dabei *direkt*, falls das betroffene Objekt direkt syntaktisch ablesbar ist; andernfalls, wenn das Objekt erst als Ergebnis einer Operation vorliegt (z.B. Zeiger-Dereferenzierung), sprechen wir von einer *indirekten* Definition oder Verwendung. Bei einer indirekten Definition kann eine Unsicherheit über das modifizierte Objekt vorliegen, falls z.B. die Zeigeranalyse mehrere potentielle Ziele geliefert hat. Eine Definition, die genau ein Objekt betrifft, bezeichnen wir als *zwingende* Definition (engl. *must-def*), während alle übrigen Definitionen mit Unsicherheit behaftet sind und daher nur als *mögliche* Definition (engl. *may-def*) gelten.

Eine Definition kann darüber hinaus ein Objekt vollständig oder teilweise verändern. Fassen wir beispielsweise zur Verkleinerung des Mengengerüsts alle Felder eines strukturierten Objekts zu einem einzigen Analyse-Objekt zusammen, so modifiziert eine Zuweisung an eines der Felder dieses Analyse-Objekt nur teilweise. Für die Genauigkeit interessant sind vor allem Definitionen, welche zugleich zwingend sind und das Objekt vollständig betreffen. Diese Definitionen nennen wir *starke Aktualisierungen* (engl. *strong updates*), während die übrigen Definitionen *schwache Aktualisierungen* darstellen (engl. *weak updates*). Diese Begriffe spielen in der Dissertation eine wichtige Rolle, weil starke Aktualisierungen einer der Vorteile sind, die fluss-sensitive Zeigeranalysen gegenüber den fluss-insensitiven Analysen ausnutzen können.

Wir betrachten den einfachen Datenfluss, der schlicht einen Wert weiterreicht; Operationen auf Werten interpretieren wir nicht, lediglich die Dereferenzierung findet Beachtung. Typumwandlungen können wir ignorieren oder in typsicheren Sprachen zur Ausfilterung falscher Resultate nutzen. Neben dem Fluss von einer Definition zu ihren Verwendungen interessieren uns daher nur noch *Kopierzuweisungen*: Dies sind Zuweisungen, bei denen die rechte Seite das Resultat einer von uns unterstützten Operation ist:

- Die direkte Verwendung eines Objekts
- Die Zeiger-indirekte Verwendung eines oder mehrerer Objekte
- Ein Funktionsaufruf
- Die Typumwandlung des Resultats einer der unterstützten Operationen

Sehr ähnlich und von uns daher ebenfalls als Kopierzuweisung betrachtet sind Parameterübergabe und Funktionsrückgabe: Bei der Parameterübergabe bildet der Parameter die LHS und das Argument die RHS. Bei der Funktionsrückgabe ist die Zuweisung des Resultats am Aufruf die LHS, während die Rückgabe-Ausdrücke jeweils eine RHS bilden.

Innerhalb eines Unterprogrammes findet Datenfluss – abgesehen von Zyklen – nur in Vorwärtsrichtung des CFG statt. Interprozedural, auf dem Aufrufgraphen, ist die Situation jedoch komplizierter: Hier fließen Informationen sowohl auf- als auch abwärts. Der interprozedurale Datenfluss findet dabei außer über Parameter und Rückgabe auch über Seiteneffekte statt. Als *schreibenden Seiteneffekt* eines Unterprogrammes f bezeichnen wir dazu eine Definition eines nicht-lokalen Objekts, die auch am CFG-Ausgang von f noch gültig ist. Umgekehrt bildet eine am CFG-Eingang von f aktive Verwendung eines nicht-lokalen Objekts einen *lesenden Seiteneffekt*.

2.2.1 Spezifikation von Datenfluss-Problemen

Die Spezifikation eines Datenfluss-Problems (vgl. z.B. Marlowe und Ryder [1990]) umfasst eine Beschreibung der Fakten und der auf diese anzuwenden-

den Operationen. Die Fakten sind dabei in einem endlichen (Halb-)Verband (engl. *lattice*) $\mathcal{L}$ organisiert.

Die Veränderung von Fakten durch Anweisungen im Programm modellieren wir über *Transferfunktionen*. Die Menge aller Transferfunktionen bildet hierbei den *Funktionenraum* zum Verband.

Kanten des ICFG erhalten zur Spezifikation eines Problems als Annotation Transferfunktionen aus dem Funktionenraum. Für einen Pfad ergibt sich daraus mittels Anwendung des Kompositionsoperators implizit eine abgeleitete Transferfunktion. Als Sonderfall haben wir für den leeren Pfad die Identität als Transferfunktion. Für überlappende Pfade können wir den ebenfalls zur Problemspezifikation gehörenden *Konfluenzoperator* einsetzen, um eine einzelne resultierende Transferfunktion für den Teilgraphen zu erhalten. Damit haben wir Bestandteile der Spezifikation eines ICFG-basierten Datenfluss-Problems zusammengetragen:

Definition 2.2.1 Ein Datenfluss-Problem ist gegeben durch

- einen Verband $\mathcal{L}$ zur Modellierung der Fakten
- einen zu $\mathcal{L}$ passenden Funktionenraum $\mathcal{F}$ und Konfluenzoperator $\sqcap$
- eine Annotation der ICFG-Kanten mit Transferfunktionen $f \in \mathcal{F}$

Für die (effiziente) Berechenbarkeit muss man in der Regel weitere Anforderungen an die Transferfunktionen stellen.

Definition 2.2.2 Ein monotones Datenfluss-Problem ist ein Datenfluss-Problem, bei dem alle Transferfunktionen monoton sind, d.h. $\forall x, y \in \mathcal{L} : \forall f \in \mathcal{F} : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$ (wobei $\sqsubseteq$ die Ordnung aus dem Verband ist).

Diese Monotonität erlaubt zusammen mit der Endlichkeit des Verbandes einen gewöhnlichen Fixpunkt-Ansatz als Approximation der optimalen Lösung. Damit der so erhaltene Fixpunkt tatsächlich der optimalen Lösung entspricht, benötigen wir noch zusätzlich die Distributivität:

Definition 2.2.3 Ein distributives Datenfluss-Problem ist ein Datenfluss-Problem, bei dem alle Transferfunktionen distributiv sind, d.h. $\forall x, y \in \mathcal{L} : \forall f \in \mathcal{F} : f(x \sqcap y) = f(x) \sqcap f(y)$

Mit dieser Eigenschaft können wir den Konfluenzoperator vor oder nach der Anwendung einer Transferfunktion einsetzen. Distributivität erzwingt Monotonität, daher ist jedes distributive Problem auch monoton.

Für ein Datenfluss-Problem wie eben spezifiziert gilt es, die Lösung an jedem ICFG-Knoten zu berechnen. Die gesuchte Lösung an einem Knoten v ist dabei die Anwendung des Konfluenzoperators auf das Resultat der Transferfunktionen aller Pfade vom Programmstart zu v. Diese Lösung bezeichnet man als die *meet over all paths*-Lösung (MOP). Sie ist jedoch im Allgemeinen nicht berechenbar; man begnügt sich daher mit der *maximalen Fixpunkt-Lösung* (engl. *maximum fixpoint*, MFP), die sich durch einfache Iteration über die Datenfluss-Gleichungen bis zur Stabilisierung ergibt. Kildall [1973] schlug dieses Verfahren vor und zeigte, dass für Probleme, deren Transferfunktionen sowohl monoton als auch distributiv sind, die gesuchte MOP-Lösung mit der effizient berechenbaren MFP-Lösung übereinstimmt.

Im interprozeduralen Fall bemüht man sich, *unrealisierbare Pfade* auszuschließen. Darunter versteht man Pfade, bei denen Aufrufe und Rücksprünge nicht korrekt gepaart sind und damit der stapelartigen Aufrufsemantik widersprechen. Hier sucht man daher nicht nach der MOP-Lösung, sondern nach der präziseren MOVP-Lösung (engl. *meet over all valid paths*).

Wir formulieren in dieser Dissertation die Zeigeranalysen als Datenfluss-Problem. Wir formulieren sie jedoch nicht, wie allgemein üblich, auf der Kontrollfluss-Darstellung (ICFG), sondern auf der Datenfluss-Darstellung (IDFG). Die Spezifikation des Problems ist dabei nahezu trivial:

- Als Fakten wählen wir den Verband der Mengen über die potentiellen Zeigerziele, mit der vollständigen Menge als Top-Element und der leeren Menge als Bottom-Element.

- Der Konfluenzoperator ist die Mengen-Vereinigung.

- Als Transferfunktion an den IDFG-Kanten wählen wir die Identität; lediglich ausgehende Kanten von IDFG-Knoten an Zeigerziel-Quellen (vgl. Abschnitt 3.1.3) erhalten davon abweichend eine Funktion, welche konstant das jeweilige Zeigerziel liefert.

Der Verband ist endlich, die Transferfunktionen erfüllen alle typischerweise gewünschten Eigenschaften (Monotonität, Distributivität), so dass sich eine Vielzahl von Frameworks anwenden lässt.

2.2.2 Frameworks zur Lösung

Schon früh war bekannt, dass es große Ähnlichkeiten zwischen vielen Datenfluss-Problemen gibt. Daher begannen die Wissenschaftler auch früh, entsprechend allgemeine Frameworks aufzustellen, die vom Anwender nur noch die Spezifikation des Datenfluss-Problems erwarten und damit in der Lage sind, die Lösung zu berechnen. Solche Frameworks unterscheiden sich in der Mächtigkeit (welche Probleme können gelöst werden), in der Präzision, der Laufzeit, und der Frage, was als Spezifikation angegeben werden muss. Typischerweise ist man in der Lage, Terminierung, Korrektheit und Laufzeit der Framework-basierten Lösung für ein Datenfluss-Problem anzugeben, weil entsprechend allgemeine Resultate für die Frameworks bekannt sind.

Die ersten Frameworks beschränkten sich noch auf die Lösung intraprozeduraler Probleme, während heutzutage die interprozeduralen Probleme gefragt sind. Historisch begann die Zeit der intraprozeduralen Frameworks mit Kildalls Arbeit [Kildall 1973] zu monotonen, distributiven Problemen. Er beschrieb darin die auch heute noch grundlegende Lösungstechnik über eine Fixpunkt-Iteration. Alternativ dazu kann eine Arbeitsliste als Steuerung auftreten, aus der bis zur vollständigen Leerung jeweils ein Element entnommen und bearbeitet wird (was möglicherweise die Arbeitsliste wieder erweitert).

Diese beiden alternativen Modelle werden effizient wie folgt genutzt:

1. Eine Fixpunkt-Iteration traversiert in jeder Iteration den CFG und propagiert die Fakten. Für die beste Performanz erfolgt die Traversierung dabei in topologischer Ordnung (im azyklischen Fall) bzw. in umgekehrter Postfix-Reihenfolge (im zyklischen Fall).

2. Eine Arbeitsliste enthält die noch zu betrachtenden CFG-Knoten. Das Verfahren arbeitet iterativ deren Einträge ab. Zur Beschleunigung kann eine Warteschlange mit Prioritäten gemäß topologischer Ordnung benutzt werden (anstelle einer normalen Liste).

Im monotonen, distributiven Fall berechnet der Fixpunkt-Ansatz genau die gewünschte MOP-Lösung. Kam und Ullman [1977] zeigten, dass die Fixpunkt-Lösung (MFP) auch dann noch existiert und mit Kildalls Verfahren berechenbar ist, wenn man auf die Distributivität verzichtet; es gibt dann jedoch Probleme, für die diese Lösung schlechter als die eigentlich gesuchte MOP-Lösung ist. Kam und Ullman bewiesen jedoch auch, dass es keinen Algorithmus gibt, der für *jedes* monotone DFA-Problem die MOP-Lösung berechnet, so dass man sich mit der MFP-Lösung zufrieden geben muss.

Ein Beispiel für die frühe Betrachtung interprozeduraler Datenfluss-Analysen ist Rosens Werk [Rosen 1979]. Er beschreibt einen Fixpunkt-Ansatz für Bitvektor-Probleme. Ein Meilenstein für die Betrachtung interprozeduraler Probleme war die Arbeit von Sharir und Pnueli [Sharir und Pnueli 1981]. Darin präsentierten sie zwei Ansätze: den *funktionalen Ansatz* und den Ansatz mit *Aufruf-Strings*. Der funktionale Ansatz setzt die Idee fort, Datenfluss-Probleme via Transfer-Funktionen auf dem CFG zu beschreiben: Mit diesem Ansatz werden die Effekte eines Unterprogramms in einer Summary-Funktion beschrieben, die im Kontext als Transfer-Funktion für entsprechende Aufrufe eingesetzt wird.

Der Ansatz der Aufruf-Strings dagegen unterscheidet die Effekte einer Funktion je nach Aufrufkontext, wobei dieser Kontext über die Aufruf-Strings beschrieben ist. Ein solcher String benennt die letzten k Aufrufer, die somit unterschieden werden können. Dieses Vorgehen ist später allgemeiner (auch für rekursive Datenstrukturen) als *k-Limiting* bezeichnet worden. Sharir und Pnueli annotieren dabei die Datenfluss-Fakten an jedem Knoten mit einem solchen String. Beide Ansätze setzen Kildalls Framework interprozedural fort, d.h. sie gehen von einem endlichen Faktenverband und distributiven Transferfunktionen aus.

Die Koinzidenz von MFP und MOP hat Kildall für den intraprozeduralen Fall gezeigt. Knoop und Steffen [1992, Theorem 5.3] haben die gleiche Beziehung für interprozedurale Analysen nachgewiesen, wobei ihr Nachweis im Unterschied zu z.B. Sharir und Pnueli auch korrekt mit lokalen Variablen rekursiver Programme umgehen kann.

Von Reps und Kollegen stammen die beiden Frameworks IFDS und IDE,

wobei letzteres mächtiger ist. IFDS ist bekannt geworden als die erste Polynomialzeit-Lösung zur Berechnung der MOVP-Lösung für Datenfluss-Probleme [Reps 1998; Reps u. a. 1995, 1994]. IDE ging einen Schritt weiter und hob die Beschränkung der Endlichkeit der darin genutzen Faktenbasis auf [Sagiv u. a. 1996].

Marlowe und Ryder [1990] geben einen Überblick über verschiedene Datenfluss-Frameworks und deren Eigenschaften.

Neben Datenfluss-Analysen existieren auch Verfahren auf Basis der abstrakten Interpretation [Cousot und Cousot 1977] zur Durchführung (formaler) statischer Analysen. Dieser Ansatz liefert ein theoretisches Fundament für statische Analysen auf Basis einer abstrahierten Programmsemantik. Das Buch von Nielson u. a. [1999] gibt einen Überblick zu Programmanalysen mit starkem Fokus auf die abstrakte Interpretation.

2.2.3 Datenfluss-Darstellungen

Die Literatur kennt verschiedene Modelle, um Datenfluss darzustellen. Die naive Idee, alle Verbindungen zwischen Definitionen und Verwendungen explizit darzustellen (wie sie sich mittels klassischer gültiger Definitionen ergibt), kommt aufgrund der hohen Kantenzahl dabei fast nirgends zum Einsatz.

Am populärsten für die Darstellung intraprozeduralen Datenflusses ist die SSA-Form (engl. *static single assignment*) von Cytron u. a. [1991]. Ihre markanteste Eigenschaft besteht darin, dass sich jede Verwendung auf genau eine Definition bezieht. Um dies zu erreichen, müssen an Konfluenzpunkten des CFG, an denen verschiedene Definitionen des gleichen Objekts ankommen, künstliche Definitionen eingefügt werden. Diese sogenannten *ϕ-Knoten* sind die zentralen Bausteine der SSA-Form. Es ergibt sich dabei, dass dann jede Definition all ihre Verwendungen dominiert, und dass die ϕ-Knoten genau in den iterierten Dominanzgrenzen der Definitionen platziert werden müssen. Dabei dominiert ein Knoten u einen Knoten v, geschrieben als u dom v, wenn alle Pfade vom CFG-Startknoten zu v zwingend durch u führen. Die Dominanzbeziehung lässt sich für alle Knoten kompakt als Baum darstellen.

Es existieren verschiedene, sehr effiziente Verfahren zur Berechnung dieses *Domimanzbaumes*; vgl. Georgiadis [2005].

Um ein Programm in die SSA-Form zu transformieren, war bislang die Kenntnis aller Zeigerziele an indirekten Definitionen und Verwendungen notwendig [Hasti und Horwitz 1998, Abschnitt 1.3]; [Cytron und Gershbein 1993]. Unsere Arbeit liefert jedoch einen neuen Ansatz zur Konstruktion der SSA-Form, bei dem *während* der Konstruktion die Zeigerziele bestimmt werden.

Die interprozedurale Erweiterung der SSA-Form ist als ISSA-Form bekannt (engl. *interprocedural static single assignment*). Für den IDFG haben wir in dieser Arbeit die ISSA-Form als Darstellung gewählt. Die ISSA-Form wurde bereits in einem früheren Artikel beschrieben [Staiger u. a. 2007]. Darin wurde auch beschrieben, wie verschieden genaue (fluss-insensitive) Zeigeranalysen als Vorarbeit zum Aufbau der ISSA-Form eingesetzt werden können. In dieser Dissertation verwenden wir für die interprozeduralen Knoten eingängigere Namen als die technischen Bezeichnungen des Artikels. Kapitel 3 wird die ISSA-Darstellung ausführlicher beschreiben.

Die ISSA-Darstellung taucht auch in anderen Publikationen auf, dort jedoch nur als Mittel zum Zweck, nicht im Blickpunkt der Beschreibung. So liefert Liao [Liao 2000] eine sehr knappe Formulierung seiner Darstellung, die im Prinzip ISSA ist.

Sowohl SSA- als auch ISSA-Graphen enthalten in der Regel überflüssige Definitionen und ϕ-Knoten. Dabei ist ein Knoten überflüssig, wenn es keinen Pfad von ihm zu einer Verwendung gibt. Solche Knoten können ohne Verlust aus dem Graphen entfernt werden. Dieser Vorgang ist bekannt als *pruning* und findet meist erst nach der Konstruktion des (I)SSA-Graphen statt.

Alternative Datenfluss-Darstellungen existieren, sollen hier aber nur kurz erwähnt werden. Singer [2005] hat als symmetrische Alternative zu SSA die SSI-Darstellung erfunden (engl. *static single information form*). Neben den ϕ-Knoten an Konfluenzpunkten enthält diese weitere induzierte Knoten an Verzweigungen, zur Vereinigung der Verwendungen auf den verschiedenen Nachfolgerpfaden. Muchnick [1997] beschreibt seine WEB-Darstellung, bei der Datenfluss-Teilgraphen mit gemeinsamer Verwendung vereint werden.

2.3 Zeigeranalysen

2.3.1 Allgemeines

Eine *Zeigeranalyse* schätzt die Objekte ab, auf die ein Zeiger an einer bestimmten Stelle zeigen kann. Dagegen ermittelt eine *Alias-Analyse* eine Menge von Paaren von Objekten, die Aliase zueinander sind.

Das Problem, die Ziele von Zeigern mit einem vernünftigen Kompromiss aus Geschwindigkeit und Genauigkeit abzuschätzen, hat in den letzten 20 Jahren viel Aufmerksamkeit erhalten. Entsprechend groß ist die Zahl an einschlägigen Publikationen. Hind [2001] bietet einen bekannten und guten Überblick, ist allerdings inzwischen auch schon wieder veraltet. Erhalten geblieben ist jedoch die Klassifikation der Zeigeranalysen in verschiedenen Dimensionen. Diese Einteilung hat später auch Ryder [2003] ausgeführt. Wir geben hier kurz die wichtigsten für uns relevanten Dimensionen wieder:

Fluss-Sensitivität: Eine Analyse ist fluss-sensitiv (FS), wenn sie den Kontrollfluss beachtet, andernfalls fluss-insensitiv (FI). Eine fluss-insensitive Analyse betrachtet nur die Objekte und deren Beziehungen, nicht aber die feinere Unterscheidung nach Definitionen und Verwendungen. Alternativ kann man Fluss-Insensitivität auch als grobe Kontrollfluss-Überschätzung beschreiben, wobei der (I)CFG als vollständiger Graph angenommen wird.

Datenfluss-Sensitivität (Richtung): Eine Analyse ist gerichtet oder datenfluss-sensitiv, wenn sie bei einer Kopierzuweisung die Datenfluss-Richtung beachtet. Eine ungerichtete Analyse nimmt zugleich auch den umgekehrten Datenfluss von LHS zu RHS an, was im fluss-insensitiven Fall eine Unifikation der Objekte auf beiden Seiten bedeutet.

Kontext-Sensitivität: Eine Analyse ist kontext-sensitiv (CS), wenn sie die Resultate für ein Unterprogramm in verschiedenen Kontexten unterscheiden kann. Wenn sie nur ein zusammenfassendes Resultat für alle Kontexte liefert, ist die Analyse kontext-insensitiv (CI).

Heapmodellierung: Eine Analyse kann im Allgemeinen nicht entscheiden, wie oft eine dynamische Allokation durchlaufen wird. Die Zahl der Heapobjekte ist daher unbekannt und wird angenähert. Die einfachste Näherung betrachtet den Heap als ein einziges Objekt. Feiner ist die Unterscheidung nach Allokationsstelle: Bei diesem Standardmodell wird für jede Allokationsstelle im Programm ein Heapobjekt angenommen. Präziser ist natürlich die Unterscheidung dieser Allokationen im Kontext.

Feld-Sensitivität: Unterscheidet eine Analyse die Felder von Strukturen als eigene (Unter-)Objekte, so ist sie feld-sensitiv. Zur Reduktion des Mengengerüsts sind viele Analysen insbesondere für die Sprache C jedoch feld-insensitiv, d.h. sie unterscheiden die Felder nicht und kennen nur die Struktur als Ganzes.

Die genannten Abkürzungen für die Dimensionen lassen sich dabei kombinieren, so dass z.B. FSCI für die Klasse der fluss-sensitiven, aber kontext-insensitiven Verfahren steht.

Generell hat sich unter allen publizierten Zeigeranalysen kein klar bestes Verfahren herausgestellt. Stattdessen folgen einige Publikationen nun der Idee, eine auf das jeweilige zu lösende tatsächliche Problem zugeschnittene Zeigeranalyse zu konstruieren. Hierbei benennt die Analyse für das eigentliche Problem die relevanten Zeiger (und ggf. die gewünschte Genauigkeit), und nur für diese berechnet eine Zeigeranalyse die Ziele. Dieses Vorgehen ist bekannt als *bedarfsgetriebenes* (engl. *demand-driven*) oder *klientengetriebenes* (engl. *client-driven*) Vorgehen.

Aus theoretischer Sicht ist die Aufgabe natürlich unentscheidbar. Dies gilt selbst unter der üblichen Annahme in statischen Analysen, dass alle Kontrollfluss-Pfade auch tatsächlich ausführbar sind und dass das vollständige Programm vorliegt [Ramalingam 1994]. Beschränkt man die Programme

weiterhin darauf, dass alle Variablen skalar sind und das Typsystem eingehalten wird, so bleibt auch dann noch die Unentscheidbarkeit sogar der intraprozeduralen Zeigeranalyse bestehen [Chakaravarthy 2003, Theorem 2, S. 116]. Erst beim Verzicht auf Heapallokationen ist die Zeigeranalyse nach Chakaravarthy [2003] entscheidbar, wobei sich dann die Fluss-(In)Sensitivität verschieden auswirkt: Die intraprozedurale fluss-insensitive Zeigeranalyse ist dann in $\mathcal{P}$, während die fluss-sensitive Analyse noch immer *PSPACE*-hart ist. Die Lockerung der Voraussetzung wohldefinierter Typen macht dabei das FI-Problem $\mathcal{NP}$-hart [Horwitz 1997, S. 4] . Umgekehrt liegt das FS-Problem in $\mathcal{P}$, wenn man sich auf einstufige Zeiger beschränkt [Landi 1992, Theorem 4.2.1, S. 30].

Wir sehen also, dass es sich um ein schweres Problem handelt, so dass praktikable Zeigeranalysen einige Einschränkungen hinnehmen müssen. Die schnellsten Verfahren sind fast linear, dafür fluss-insensitiv und ungerichtet [Steensgaard 1996]. Für fluss-sensitive Verfahren ist die beste bekannte Laufzeit kubisch [Ruf 1995]. Die gleiche Abschätzung gilt für gerichtete fluss-insensitive Verfahren [Andersen 1994], jedoch mit deutlich besseren Konstanten. Grund für die kubische Zeit ist dabei in beiden Fällen das Problem der dynamischen transitiven Hülle. Beschränkt man sich auf Aliase durch Referenzparameter, so steht mit dem Verfahren von Cooper und Kennedy eine effiziente Lösung zur Verfügung [Cooper und Kennedy 1989].

Die Literatur zu Zeiger- und Alias-Analysen bietet in den letzten Jahren kaum neue Ansätze zur Lösung des Problems. Vielmehr beschäftigen sich die Arbeiten ausgiebig damit, bestehende Ansätze zu beschleunigen, z.B. durch Zyklenkontraktion oder Einsatz von BDDs als Datenstrukturen. Die Dissertation bietet beides: Einen neuen Ansatz und dessen Beschleunigung. Dabei beschreiben wir die Analyse als Zeigeranalyse, nicht als Alias-Analyse.

2.3.2 Fluss- und kontext-insensitive Analysen

Wir beginnen mit der Betrachtung der unpräzisesten Vertreter, also den FI-CI-Zeigeranalysen (kurz für: fluss- und kontext-insensitiv). Da unsere neue Analyse als fluss-sensitive Ausprägung von Andersens Analyse verstanden

werden kann, betrachten wir dabei diese zuerst und etwas ausführlicher. Zudem gilt diese Analyse als die wohl einflussreichste Zeigeranalyse bislang.

2.3.2.1 Andersens Analyse und Weiterentwicklungen

Andersen [1994] zerlegte die Zeigeranalyse in zwei Teile:

1. Problemspezifikation: Teilmengenbeziehungen (engl. *constraints*) zwischen Zeigerziel-Mengen stellen Bedingungen an die Lösung auf.

2. Lösung: Eine Fixpunkt-Iteration sucht nach der minimalen Lösung, indem sie in jeder Iteration die Zeigerziel-Mengen unter Beachtung der Bedingungen vergrößert.

Als FICI-Analyse nimmt das Verfahren eine Zeigerziel-Menge $Targets(p)$ pro (Zeiger-)Variable oder Heapobjekt p an. Der erste Schritt wandelt nun die Anweisungen des Programmes in Teilmengenbeziehungen um. So erzeugt das Verfahren z.B. für die Zuweisung $p = q$ die Inklusionsbeziehung $Targets(q) \subseteq Targets(p)$. Die Lösungsphase betrachtet diese Bedingungen als Regeln, um die Zeigerziel-Mengen zu vergrößern. Sie stoppt, wenn alle Beziehungen keine weiteren Veränderungen mit sich bringen und damit erfüllt sind. Dabei muss man beachten, dass im Zuge der Lösung auch neue Teilmengenbeziehungen erkannt werden: Existiert z.B. neben der oben angeführten Bedingung zusätzlich die Forderung $Targets(p) \subseteq Targets(r)$, so fordert der transitive Schluss auch die Relation $Targets(q) \subseteq Targets(r)$. Neben dem transitiven Schluss sorgen auch neu entdeckte Ziele für indirekte Operationen für neue Bedingungen.

Die Bedingungen lassen sich in Form des sogenannten *Constraint-Graphen* darstellen. Der Graph enthält dabei als Knoten die (Zeiger-)Objekte als Repräsentanten für ihre Zeigerzielmengen, und die Kanten drücken die Beziehungen aus. Für $Targets(q) \subseteq Targets(p)$ entsteht somit der Teilgraph $q \rightarrow p$. Die Aufgabe ist nun, die dynamische transitive Hülle des Graphen zu berechnen, d.h. die transitive Hülle unter der Erschwernis der ständig neu hinzukommenden Kanten.

Aufgrund seiner kubischen Laufzeit galt Andersens Verfahren zunächst als nicht skalierbar. Viele Publikationen haben sich daher mit Ideen zur Beschleunigung auseinander gesetzt. Die beliebteste und wohl sehr effektive Beschleunigung ist eine effiziente Zyklenerkennung während der Lösungsphase. Fähndrich u. a. [1998] erkannten Zyklen nach dem transitiven Schluss, während Heintze und Tardieu [2001] den Graphen dafür nicht erst transitiv schließen. Eine schnellere Zyklenerkennung ist auch der Beitrag von Hardekopf und Lin [2007]. Dieses Verfahren erkennt Zyklen auf dem initialen Graphen und erstellt mittels Platzhaltern für unbekannte Zeigerziele auch Schablonen für später auftauchende Zyklen. Der Artikel berichtet, dass damit ein 1,3 MLoC-Programm in knapp 500 Sekunden analysiert werden konnte. Auch Heintze und Tardieu [2001] berichten von einer Skalierbarkeit bis zu 1 MLoC.

Rountev u. a. [2001] haben Andersens Analyse auf objektorientierte Sprachen übertragen. Shapiro und Horwitz [1997] beschreiben ein parametrisiertes Verfahren, um den Raum zwischen Andersen und weniger präzisen Analysen zu erkunden. Bei geeigneter Wahl des Parameters entspricht ihr Verfahren dem von Andersen, bei anderer Wahl unpräziseren Algorithmen.

Pearce entwickelte eine effiziente feld-sensitive Variante von Andersens Analyse [Pearce 2005; Pearce u. a. 2007]. Er ordnete die Objekte dazu in einem Array an, wobei die Felder einer Struktur sukzessive Einträge belegen. Bei Varianten teilen sich die jeweils n-ten Felder das gleiche Objekt, d.h. den gleichen Eintrag im Objekte-Array. Auf diese Weise kann Pearce mit dem Array-Index ein Objekt bezeichnen. Für Heapobjekte, die in C bei der Allokation untypisiert sind, muss er jedoch als Typ den schlechtesten Fall, d.h. eine Variante über alle vorkommenden Typen, annehmen.

2.3.2.2 Andere fluss-insensitive Verfahren

Andere fluss-insensitive Verfahren versuchten, durch Verzicht auf Präzision skalierbarer zu sein. Wegen ihrer geringen Bedeutung für die vorliegende Dissertation streifen wir nur kurz die wichtigsten Vertreter. Das bekannteste Beispiel stammt von Steensgaard [1996]: Indem hier nicht nur der Kontrollfluss, sondern auch noch die Datenfluss-Richtung ignoriert wurde, gelang mittels

Union-Find ein fast linearer Algorithmus. Die Analyse fasst dabei mehrere Zeiger zu Äquivalenzklassen zusammen und bestimmt nur noch für diese Klassen – nicht mehr für einzelne Zeiger – die möglichen Ziele. Im Vergleich von Shapiro und Horwitz [1997] zeigte sich Andersens Analyse klar präziser als Steensgaards Verfahren, jedoch auch entsprechend langsamer.

Einen Mittelweg erkannte Das [2000], indem er die Datenfluss-Richtung nur über eine Stufe, nicht aber transitiv beachtete. Damit gelang ihm ein quadratisches Verfahren, dessen Präzision deutlich besser als die von Steensgaard wurde, bei immer noch akzeptabler Geschwindigkeit.

Kahlon [2008] nutzt verschieden genaue Analysen wie folgt: Zunächst bestimmt die schnellste und ungenaueste Analyse (Steensgaard) erste Gruppen von Objekten, wobei in jeder Gruppe die Objekte untereinander Aliase sind, zwischen den Gruppen jedoch keine solche Beziehung herrscht. Anschließend kann die nächstgenauere Analyse (Andersen) auf relevante Gruppen angewendet werden, muss also nicht das gesamte Programm betrachten, sondern jeweils nur die Anweisungen zu den ausgewählten Objekten. Jede Stufe dient dazu, das Programm in disjunkte Objektgruppen zu zerlegen und damit das Mengengerüst für die nächste – präzisere – Stufe zu verkleinern, indem dort nur noch die Programmteile betrachtet werden, welche die aktuelle Objektgruppe betreffen. Kahlons abschließende Objektgruppen waren in seinen Tests klein genug, um am Ende eine kontext-sensitive Analyse zu erlauben.

2.3.3 Fluss-Sensitivität

2.3.3.1 Von fluss-insensitiv zu fluss-sensitiv

Hasti und Horwitz [1998] erhöhen die Präzision einer fluss-insensitiven Analyse, indem sie wiederholt mit den aktuellen Zeigerzielen eine SSA-Form aufbauen und mit deren Hilfe dann bessere Zeigerziele berechnen. Sie äußern die Vermutung, im Fixpunkt fluss-sensitive Präzision zu erreichen, was jedoch nie bewiesen wurde. [Hardekopf und Lin 2009b, Abschnitt 3] widersprechen dieser Vermutung neuerdings sogar, allerdings ebenso ohne handfeste Belege. Dennoch hat sich die Vermutung festgesetzt in Form der Gleichung

„fluss-insensitive Zeigeranalyse + SSA = fluss-sensitive Zeigeranalyse“.

Ein anderer Übergang hin zu mehr Fluss-Sensitivität besteht darin, nur manche Objekte fluss-sensitiv zu behandeln. So analysieren Whaley und Lam [2002] Java-Programme und behandeln dabei lokale Variablen fluss-sensitiv, alle übrigen jedoch insensitiv. Für Programme in typsicheren Sprachen besteht weiterhin die Option, sich von vornherein auf die Zeiger-relevanten Operationen zu beschränken; in C ist dies kaum möglich.

Die Artikel von Hind und Pioli haben in der Forschergemeinde den Eindruck hinterlassen, dass Fluss-Sensitivität keinen für den zu zahlenden Preis nennenswerten Präzisionsgewinn liefert [Hind 2001; Hind und Pioli 1998, 2000]. Ursache waren Studien zum Vergleich der Präzision und der Kosten. In dieser Dissertation führen wir einen ähnlichen Vergleich durch, weil die Ergebnisse der ursprünglichen Studie möglicherweise durch die Auswahl der Testprogramme und der konkreten Analysen nur bedingt gelten.

Fluss-Sensitivität kann unter anderem durch starke Aktualisierungen gegenüber der Fluss-Insensitivität an Präzision gewinnen. Daher ist es ratsam, eine FS-Analyse zugleich auch feld-sensitiv zu realisieren, um mehr starke Aktualisierungen zu erhalten. Eine FI-Analyse dagegen kann diesen Vorteil nicht nutzen, und tatsächlich sind die meisten publizierten FI-Analysen für C-Programme auch feld-insensitiv.

2.3.3.2 Zeigeranalyse als Datenfluss-Problem

Will man vollständige Fluss-Sensitivität, so liegt es nahe, das Zeigeranalysenproblem als gewöhnliches Datenfluss-Problem zu studieren. Als Datenfluss-Problem über dem ICFG betrachtet ist eine fluss-sensitive Zeigeranalyse mit starken Aktualisierungen jedoch leider nicht distributiv. Das folgende Beispiel illustriert dies.

Beispiel 2.3.1 Die Fakten beschreiben zu jedem Objekt an einem ICFG-Knoten die dort bekannten Zeigerziele. Betrachten wir die indirekte Definition `*p = q` und die Variablen x, y, p, q. Die Transferfunktion f zur indirekten Definition hat folgende Gestalt für ein z aus dem Zustand in unmittelbar vor

der Anweisung:

$$f(z) = \begin{cases} \emptyset & in(p) = \emptyset \\ in(q) & in(p) = \{z\} \\ in(z) \cup in(q) & |in(p)| > 1 \wedge z \in in(p) \\ in(z) & sonst \end{cases}$$

Solange noch kein Zeigerziel bekannt ist, blockiert f die Propagierung aller Zeigerzielmengen. Bei genau einem Ziel werden dessen Zeigerziele ausgetauscht gegen diejenigen der rechten Seite, und bei mehr als einem Ziel liegt eine schwache Aktualisierung vor.

Diese Transferfunktion ist jedoch nicht distributiv. Dazu betrachten wir die beiden möglichen Eingangszustände

$$in_1 = (p \mapsto \{x\}, q \mapsto Q_1, x \mapsto \{y\}, y \mapsto \emptyset)$$

und

$$in_2 = (p \mapsto \{x, y\}, q \mapsto Q_2, x \mapsto \emptyset, y \mapsto \emptyset).$$

Werden diese zuerst zu einem Zustand vereint, bevor f auf das Resultat wirkt, so erhalten wir

$$f(in_1 \cup in_2) = (p \mapsto \{x, y\}, q \mapsto Q_1 \cup Q_2, x \mapsto \{y\} \cup Q_1 \cup Q_2, y \mapsto Q_1 \cup Q_2).$$

Die umgedrehte Reihenfolge (erst f, dann Vereinigung) liefert jedoch

$$f(in_1) \cup f(in_2) = (p \mapsto \{x, y\}, q \mapsto Q_1 \cup Q_2, x \mapsto Q_1 \cup Q_2, y \mapsto Q_2).$$

□

Im Unterschied zu klassischen Datenfluss-Problemen wie den gültigen Definitionen (bei denen ja ebenfalls starke Aktualisierungen beachtet werden), muss die Zeigeranalyse die Ziele der rechten Seite weiterreichen. Diese Transitivität dürfte es wohl sein, die im Unterschied zu den klassischen Problemen die Distributivität zerstört.

Diese fehlende Eigenschaft sorgt dafür, dass eine fluss-sensitive Zeigeranalyse nicht in die typischen Datenfluss-Frameworks passt. Algorithmus, Korrektheit, Terminierung und Laufzeit müssen daher jeweils einzeln betrachtet werden.

Die fluss-sensitiven Zeigeranalysen bestehen wie die klassischen Datenfluss-Verfahren im Wesentlichen aus einer Fixpunkt-Iteration unter Anwendung spezifischer Transferfunktionen auf die einzelnen ICFG-Knoten. Ein Beispiel hierfür ist das Verfahren von Hind u. a. [1999]. Neben dem angeführten Problem der Nicht-Distributivität zeigt sich dabei aber das Problem, dass die Analyse die Fakten z.B. pro Grundblock speichern muss. Ein solcher Fakt (den wir auch als Zustand bezeichnen können) umfasst sämtliche Points-To-Relationen am jeweiligen Programmpunkt. Ein Grundblock verändert diesen Zustand jedoch oft nur marginal, pro Anweisung darin ist oft nur ein Objekt betroffen, bei indirekten Operationen ein paar Objekte. Die Mitführung der vollständigen Menge an jedem Grundblock erscheint daher wie unnötiger Ballast. Aus diesem Grund gibt es verschiedene Ansätze, alternative Darstellungen zum ICFG als Basis zu nutzen. Diese Alternativen orientieren sich mehr am Datenfluss.

2.3.3.3 Abkehr vom ICFG als Grundlage

Will man eine Datenfluss-orientierte Darstellung als Basis benutzen, so steht man vor dem Problem, dass zu deren präzisen Konstruktion die Zeigerziele bereits bekannt sein müssten – die Katze beißt sich in den Schwanz. Die publizierten Ansätze zur Verwendung einer ICFG-Alternative umgehen dies mit äußerst konservativen Überschätzungen der möglichen Datenflüsse: Zeigerziele werden über die Betrachtung der Adressnahme-Information überschätzt, und Seiteneffekte an Aufrufen müssen ebenfalls pessimistisch angenommen werden. Dies beeinflusst gegenüber einen hypothetischen optimalen Darstellung zumindest das Mengengerüst (unnötige Kanten und damit verbundene Arbeit), je nach Verfahren aber vielleicht auch die Präzision. Für typsichere Sprachen wie Java besteht die Möglichkeit, nur die Variablen zu betrachten, die von einem Referenztyp sind. In C reicht dies jedoch nicht aus.

Beispiele für solche ICFG-Alternativen sind:

- Der *sparse evaluation graph* (SEG) bei Choi u. a. [1993].
- Der *value dependence graph* (VDG) bei Ruf [1995].
- Zusammenfassungs-Graphen bei Whaley und Lam [2002].
- Die Points-To SSA-Form bei Gutzmann u. a. [2007] und Lundberg und Löwe [2007].

Die vorliegende Dissertation beschreitet hier einen neuen Weg und benutzt einen nicht vorab überschätzten, sondern im Wechsel mit der Zeigeranalyse berechneten Datenfluss-Graphen.

2.3.3.4 Interprozedurale Analyse

Die interprozedurale, fluss-sensitive Analyse steht vor dem Problem, dass ein bidirektionaler Informationsfluss stattfindet: Neue Zeigerziele in einer Funktion f müssen ggf. sowohl zu Aufrufern von f als auch zu in f gerufenen Funktionen fließen. Hier gibt es wiederum verschiedene Lösungsstrategien:

- Neben der lokalen Fixpunkt-Iteration in jeder Funktion kann eine globale Fixpunkt-Iteration über dem Aufrufgraphen zum Einsatz kommen [Hind u. a. 1999].
- Bei geeigneten Zusammenfassungen für Funktionen genügt ein erster Postorder-Durchlauf über den Aufrufgraphen zur Konstruktion der Zusammenfassungen, gefolgt von einem Preorder-Durchlauf zur Wertepropagierung [Chatterjee u. a. 1999].
- Eine Simulation der Ausführung unterbricht die Behandlung der aktuellen Funktion jeweils an Aufrufen und setzt im Aufrufziel fort. Geeignete Zusammenfassungen sind nötig, um zu viele Reanalysen zu vermeiden [Gutzmann u. a. 2007; Lundberg und Löwe 2007; Wilson 1997].

2.3.3.5 Grad der Fluss-Sensitivität

Bei näherer Betrachtung der verschiedenen Zeigeranalysen fällt auf, dass die Dimension der Fluss-Sensitivität tatsächlich mehr Werte als nur (in)sensitiv umfasst. Wir geben daher eine feinere Abstufung an, von ungenauer zu genauer:

FI vereinend Verfahren dieser Präzision sind fluss-insensitiv und berechnen nur ein globales Resultat pro Objektgruppe, wobei sie die Datenfluss-Richtung an allen Arten von Zuweisungen nicht präzise modellieren und so Objekte zu Gruppen vereinen.

Beispiele: Das [2000]; Steensgaard [1996].

FI global Verfahren dieser Präzision sind fluss-insensitiv und berechnen nur ein globales Resultat pro Objekt, beachten jedoch die Datenfluss-Richtung an Zuweisungen und vermeiden die entsprechenden Präzisionsverluste der vorigen Gruppe.

Beispiele: Andersen [1994]; Berndl u. a. [2003].

FI lokal Verfahren dieser Präzision sind fluss-insensitiv und berechnen ein Resultat pro Objekt pro Funktion.

Beispiele: Hind und Pioli [1998], Burke u. a. [1994].

FI/FS-Mix Verfahren dieser Präzision sind für lokale Variablen fluss-sensitiv, aber global fluss-insensitiv.

Beispiel: Whaley und Lam [2002].

FS Verfahren dieser Präzision sind fluss-sensitiv, unterstützen aber keine starken Aktualisierungen.

Beispiele: Gutzmann u. a. [2007]; Lundberg und Löwe [2007].

FS stark Verfahren dieser Präzision sind fluss-sensitiv und unterstützen direkte und indirekte starke Aktualisierungen.

Beispiele: Hardekopf und Lin [2009b]; Ruf [1995].

Das neue Verfahren aus dieser Dissertation erreicht dabei die höchste Präzisionsstufe in dieser Dimension. Noch höher anzusiedeln sind pfad-sensitive Verfahren.

2.3.3.6 Einige wichtige Verfahren

Eines der historisch ersten wichtigen Verfahren in der Klasse der fluss-sensitiven, kontext-insensitiven Zeigeranalysen stammt von Chase u. a. [1990]. Der Artikel führte die Thematik starker Aktualisierungen ein und besprach ein Verfahren, welches diese erkennen kann. Die Zeigeranalyse ist dabei ein typisches Datenfluss-Problem auf dem ICFG.

Ruf [1995] beschreibt eine Analyse, die anstelle des ICFG den VDG als Basis einsetzt. Wie bereits erwähnt, müssen dazu vorab konservative Abschätzungen der Zeigerziele erfolgen, um den VDG zu konstruieren. Rufs Analyse hat Ähnlichkeiten zu unserer neuen Analyse: Er arbeitet nicht auf dem ICFG als Basis, er kann durch verzögerte Aktionen mit starken Aktualisierungen umgehen, und er nennt (ohne Beweis) eine kubische Komplexität für sein Verfahren.

Als Stand der Dinge auf dem Gebiet der FSCI-Zeigeranalysen (kurz für fluss-sensitiv, kontext-insensitiv) sehen Hardekopf und Lin [2009b] das Verfahren von Hind und Pioli [1998]. Dieses Standardverfahren besteht aus einer globalen Fixpunkt-Iteration über dem Aufrufgraphen, innerhalb derer pro Funktion eine lokale Fixpunkt-Iteration stattfindet und die Zeigeranalyse als Datenfluss-Problem auf dem CFG behandelt.

Unsere Analyse beginnt mit der Herleitung erster Datenfluss-Beziehungen und berechnet danach erste Zeigerziele. Die Literatur kennt auch das umgekehrte Vorgehen: Lindenmaier u. a. [2005] starten damit, den Speicher durch ein einziges Objekt zu repräsentieren, und konstruieren damit eine Datenfluss-Darstellung. Anschließend nutzen sie deren Fluss-Sensitivität zur Reduktion der Zeigerziele.

Gutzmann u. a. [2007] sowie Lundberg und Löwe [2007] simulieren die Ausführung des Programmes, d.h. an einer Aufrufstelle unterbrechen sie die Analyse der aktuellen Funktion und analysieren die aufgerufene Funktion.

Um jedoch zu viele Reanalysen von Unterprogrammen zu vermeiden, müssen sie eine Zusammenfassung pro Unterprogramm in Kauf nehmen, die mit jeder Reanalyse des Unterprogrammes ungenauer wird. Zudem beherrscht das Verfahren keinerlei starke Aktualisierungen, womit ein Präzisionspotential fluss-sensitiver Analysen nicht ausgeschöpft wird.

Aktuelle FSCI-Zeigeranalysen stammen von Tok [2007] und Hardekopf und Lin [2009b], wobei wir letzteres im Abschnitt 2.3.5 zu BDD-basierten Verfahren besprechen werden. Tok hat in seiner Arbeit [Tok 2007; Tok u. a. 2006] eine gegebene, ICFG-basierte FSCI-Zeigeranalyse dadurch beschleunigt, dass die Arbeitsliste der noch zu betrachtenden ICFG-Knoten nicht mehr anhand der Kontrollfluss-Nachfolgerrelation gefüllt wird, sondern auf Basis von Datenabhängigkeiten. Dazu errichtet er inkrementell eine SSA-Form zur Ermittlung der Datenabhängigkeiten. Wir werden seine Ideen zum inkrementellen SSA-Aufbau auch für unser Verfahren einsetzen. Wir beschreiben dieses Vorgehen ausführlich in Kapitel 5.

2.3.4 Kontext-Sensitivität

Kontext-Sensitivität wurde sowohl für fluss-sensitive als auch für fluss-insensitive Analysen untersucht. [Ryder 2003, Kapitel 3] fasst für den FI-Fall zusammen, dass der Nutzen für C-Programme vom Programmierstil abhängt. Für objektorientierte Programme dagegen ist Kontext-Sensitivität nach dieser Quelle deutlich besser bis notwendig für brauchbar präzise Resultate. Ruf [1995] berichtet für C, dass eine kontext-sensitive Fassung seiner fluss-sensitiven Analyse keine nennenswert erhöhte Präzision brachte.

Die bereits angesprochenen beiden Ansätze (funktional und mit Aufruf-Strings) aus Sharir und Pnueli [1981] sind auch im Bereich der Zeigeranalysen zentral für die Realisierung der Kontext-Sensitivität.

2.3.4.1 Unterscheidung nach Aufrufpfaden

Der Ansatz von Sharir und Pnueli [1981], Aufruf-Strings zu verwenden, führt zu Zeigeranalysen, die verschiedene Kontexte eines Unterprogrammes anhand von Aufrufpfaden unterscheiden.

Bekannt in dieser Kategorie ist die Kontrollfluss-Analyse mit Beachtung der k letzten Aufrufer nach Shivers [1991]. Diese firmiert unter dem Namen k-CFA von engl. *control-flow analysis*. Häufig ist dabei k sehr klein (1 oder 2), um noch akzeptable Kosten zu erhalten. Am anderen Ende dieses Spektrums liegt die exponentielle fluss-sensitive Zeigeranalyse von Emami u. a. [1994]: Dieses Verfahren betrachtet jeden azyklischen Aufrufpfad von der Wurzel zu einer Funktion als verschiedenen Kontext.

Sridharan und Bodík [2006] beschreiben eine bedarfsgetriebene fluss-insensitive Analyse für Java-Programme, die nach Aufrufpfaden unterscheidet und dabei auch Heapobjekte kontext-sensitiv differenziert. Durch den bedarfsgetriebenen Ansatz erhoffen sich die Autoren dabei, dass die exponentielle Laufzeit in der Praxis nicht auftritt.

Lattner u. a. [2007] unterscheiden Heapobjekte nach allen azyklischen Aufrufpfaden. Für Effizienz müssen sie jedoch auf die ungenaueste Stufe der Fluss-Sensitivität zurückweichen, nämlich unifizierend fluss-insensitiv arbeiten.

2.3.4.2 Zusammenfassungen pro Unterprogramm

Der funktionale Ansatz von Sharir und Pnueli [1981] erwartet für jedes Unterprogramm ein oder mehrere Zusammenfassungen (engl. *summaries*), die als Transferfunktionen an den Aufrufstellen angewendet werden. Wilson hat hierbei die Idee partieller Transferfunktionen (PTF) eingeführt, die nur für bestimmte Kontexte anwendbar sind [Wilson und Lam 1995; Wilson 1997].

Während die Unterscheidung nach Aufrufpfaden wiederum im Kontrollfluss verhaftet ist, unterscheiden Verfahren mit Zusammenfassungen mehr anhand der Betrachtung von Daten (vor allem Zeigerziele). Wir besprechen hier kurz die zwei wichtigsten Vertreter, nämlich objektsensitive Analysen und Kontextunterscheidungen anhand von Aliasen.

Objekt-Sensitivität wurde im Artikel von Milanova u. a. [2005] eingeführt. Darin beschreiben die Autoren eine fluss-insensitive Zeigeranalyse für Java und berichten sehr gute Resultate. Die Studie von Lhotak und Hendren [2006] bestätigte, dass Objekt-Sensitivität einen deutlich besseren Effekt (in Bezug

auf Kosten/Nutzen) hat als die Unterscheidung nach Aufrufpfaden. Ob dies nur für Java gilt, ist jedoch offen.

Eine Kontext-Unterscheidung nach Aliasen finden wir bei Landi, Wilson und Chatterjee. Landis fluss-sensitive Alias-Analyse [Landi und Ryder 1992; Landi 1992] nutzt die Aliase des Aufrufers, um unrealisierbare Pfade auszuschließen. Der Algorithmus hat jedoch eine hohe polynomielle Komplexität, um dies zu erreichen. Stocks u. a. [1998] berichten später, dass Landis Verfahren bei Programmen mit ca. 20-25 KLoC bereits strauchelt.

Wilsons Analyse [Wilson und Lam 1995; Wilson 1997] ist eine sehr präzise Zeigeranalyse für C-Programme, die als eine der wenigen versucht, auch Zeigerarithmetik zu beachten und Elemente in Arrays zu unterscheiden. Pro Funktion berechnet die Analyse für jedes am Eingang anliegende relevante Aliasing-Muster unter den Parametern eine eigene PTF. Auch diese Analyse ist jedoch nicht skalierbar.

Chatterjee [Chatterjee 2000; Chatterjee u. a. 1999] entwickelte später einen anderen fluss- und kontext-sensitiven Ansatz. Dieser geht von einem vorberechneten Aufrufgraphen aus und arbeitet auf dessen SCC-DAG. Auf diesem führt er zunächst eine Postfix-Traversierung durch, welche die relevanten Aliaskonstellationen inferiert. Anschließend folgt ein Durchlauf in topologischer Ordnung, bei dem als Datenfluss-Problem die Zeigerziele unter Beachtung der verschiedenen tatsächlichen Aliase bestimmt werden. Während Chatterjee u. a. [1999] sehr gute Präzision demonstrierten, räumte Ryder [2003, Kapitel 3.1] später Skalierbarkeitsprobleme mit diesem Ansatz ein.

2.3.4.3 Die MOVP-Lösung des Zeigerproblems

Abschließend betrachten wir speziell die Aufgabe, die MOVP-Lösung für das Zeigerproblem zu bestimmen. Hier geht es also nur um eine eingeschränkte Kontext-Sensitivität insofern, dass unrealisierbare Pfade ausgeschlossen werden sollen. Dieses Problem ist für die Dissertation zur Untersuchung der Hypothesen relevant.

Das Problem der unrealisierbaren Pfade wurde schon früh in den Zeigeranalysen aufgegriffen. So präsentieren Landi und Ryder [1992] ein fluss-

sensitives Verfahren, das mit Hilfe von Aliasen erkennt, zu welchen Aufrufern zurückpropagiert werden darf. Später publizierten Reps u. a. [1995] das allgemeine IFDS-Framework für endliche, distributive Datenfluss-Probleme. Dieses lässt sich auf die von Shapiro und Horwitz [1997] präsentierte Formulierung der Analyse von Andersen anwenden, wie [Reps 1998, Kapitel 4.4] nach Melski skizziert.

Nicht ganz die MOVP-Präzision erreicht das Verfahren von Lundberg und Löwe [2007]. Die darin präsentierte simulierte Ausführung startet zwar mit MOVP-Präzision, verliert aber bei Reanalysen von Unterprogrammen immer mehr an Genauigkeit in Richtung MOP.

Choi u. a. [1993] beschreiben eine Alias-Analyse, die mit Seiteneffekten umgehen kann. Die intraprozedurale Analyse ist dabei als Datenfluss-Problem auf dem SEG spezifiziert. Interprozedural führt dieses Verfahren eine Fixpunkt-Iteration über dem (vorberechneten) Aufrufgraphen durch, wobei jede Iteration den Aufrufgraphen in topologischer Ordnung traversiert und dabei pro Unterprogramm die intraprozedurale Analyse erneut anstößt. Heapobjekte werden dabei präziser als nur nach Allokationsstelle unterschieden, und der Artikel skizziert eine Variation zur Berechnung der MOVP-Lösung.

Die vorliegende Dissertation erweitert dieses Wissen, indem sie eine FS-Zeigeranalyse mit MOVP-Präzision und der Komplexität $\mathcal{O}(n^4)$ vorstellt.

2.3.5 BDD-basierte Zeigeranalysen

Binary decision diagrams (BDDs) sind als Datenstruktur zur Repräsentation von Zeigerziel-Mengen beliebt. Sie versprechen, mit großen, ähnlichen Mengen gut zurecht zu kommen. Wir betrachten BDD-basierte Verfahren der Vollständigkeit halber, um den Stand der Forschung widerzuspiegeln.

Allgemein scheinen BDDs eine gute Skalierbarkeit zu zeigen, wenn es *viele* Zeigerziel-Mengen darzustellen gilt, jedoch nicht in einem Bereich nur „mittelvieler“ Mengen. So fällt das bei FICI-Analysen anfallende Mengengerüst wohl noch in den Bereich, in dem BDDs nicht optimal sind. Bei FICS und FSCI dagegen beginnen BDDs ihre Stärke auszuspielen, und für FSCS ist ein deutlicher Vorteil zu erwarten.

Berndl u. a. [2003] beschrieben als eine der ersten Gruppen den Einsatz von BDDs für Zeigeranalysen. Sie arbeiteten dabei mit einer Variante von Andersens Analyse für Java-Programme. Für diese FICI-Analyse war die BDD-Implementierung langsamer als bestehende Löser, jedoch vergleichbar. Für C-Programme lieferten Hardekopf und Lin [2007] ähnliche Resultate: In ihrer Implementierung zeigten sich die BDDs doppelt so langsam wie eine Realisierung mit dünn besetzten Bitmaps, dafür deutlich platzsparender.

Für Java-Programme zeigte sich Whaleys BDD-basierte, kontext-sensitive Variante von Andersens Analyse deutlich skalierbarer als konventionelle Ansätze [Whaley 2007; Whaley und Lam 2004]. Für C-Programme konnte dies jedoch nicht bestätigt werden: Hardekopf und Lin [2007] zitieren einen Artikel, der bei einer Übertragung des Verfahrens auf C-Programme eine deutlich schlechtere Laufzeit feststellen musste. Auch Zhus BDD-Einsatz in einer FICS-Analyse für C-Programme wurde nur für Programme bis zu 28 KLoC empirisch ausgewertet [Zhu 2002; Zhu und Calman 2004].

Zhu und Hardekopf haben BDDs für fluss-sensitive Zeigeranalysen eingesetzt. Zhu [2005] gelingt damit die FSCS-Analyse von Programmen bis zu 220 KLoC. Ein Nachteil dieser Analyse ist jedoch, dass sie keine starken Aktualisierungen unterstützt und damit das eigentliche Potential der Fluss-Sensitivität gegenüber der -Insensitivität vernachlässigt. Hardekopf und Lin [2009b] dagegen beherrschen starke Aktualisierungen, bieten aber nur ein FSCI-Verfahren. Mit BDDs ist diese Analyse in der Lage, Programme mit 300 KLoC zu untersuchen.

Problematisch an BDDs ist, dass sie im schlechtesten Fall exponentielle Kosten verursachen können. Ihr Verhalten ist kaum vorhersehbar und hängt von der Reihenfolge ab, in der die Variablen darin angeordnet werden. So berichtet Berndl u. a. [2003] von den dramatischen Auswirkungen verschiedener Variablenordnungen. Auch Whaley beschreibt in seiner Dissertation, dass es monatelangen, langsamen Fortschritt benötigte, um den richtigen Einsatz der BDDs zu erreichen [Whaley 2007, S. vii].

Die vorliegende Dissertation bleibt daher konventionelleren Datenstrukturen treu, die eine kubische Abschätzung des schlechtesten Falles erlauben.

Teil I

Die Kernanalyse

Kapitel 3

Der Rahmen für die Analyse

Bevor wir die Analyse an sich beschreiben, klärt dieses Kapitel zum besseren Verständnis den Kontext, in dem unsere Analyse abläuft. Dazu gehört eine kurze Festlegung der Zwischendarstellung als Repräsentation des zu analysierenden Programmes, eine Beschreibung der zu bestimmenden ISSA-Darstellung sowie die Festlegung einiger Begriffe. Das Kapitel schildert auch unseren Ansatz zur Modellierung von Objekten und Zeigerzielen. Wir unterstützen dabei sowohl ein feld-sensitives als auch ein feld-insensitives Modell. Die Unterscheidung diesbezüglich erfolgt hier konzeptionell bei der Erstellung der Programmrepräsentation, so dass die Analyse später keine weitere Unterscheidung treffen muss.

Wir beschreiben hier eine speziell auf die Zwecke der Analyse zugeschnittene Zwischendarstellung, um eine möglichst einfache Beschreibung zu erhalten. Im Rahmen eines Werkzeugs, das nicht nur die Kernanalyse anbietet, wird in aller Regel eine ausführlichere Darstellung benutzt werden. Es sollte jedoch keine Schwierigkeiten bereiten, mit den hier bei der Konzentration auf die Kernanalyse vorgenommenen Vereinfachungen als Abstraktionen im Kontext einer anderen Darstellung zurecht zu kommen.

Grundlage für die Analyse ist eine Zwischendarstellung (engl. *intermediate representation* oder auch *internal representation*, IR) für das zu analysierende Programm. Ein Frontend, wie es aus dem Übersetzerbau bekannt ist, erzeugt diese IR für jede Quellcode-Datei. Anschließend verbindet ein

Linker diese einzelnen Zwischendarstellungen zu einer einzigen IR für das gesamte Programm. Diese Technologie setzen wir voraus und besprechen sie hier nicht weiter. Wir setzen dabei keine bestimmte Programmiersprache für die zu analysierenden Programme voraus; jedoch werden sich die Beispiele und teilweise auch die Beschreibung in dieser Dissertation auf die Sprache C konzentrieren.

3.1 Objekte und ihre Repräsentation

Zunächst wollen wir klären, wie das Objektmodell der Analyse aussieht. Formal müssen wir hierzu die tatsächlichen Objekte der Programmausführungen abbilden auf eine endliche Objektmenge in der Analyse. Diese Abbildung weist daher verschiedenen realen Objekten den gleichen Repräsentanten zu. Zur besseren Unterscheidung sprechen wir in diesem Abschnitt von *tatsächlichen Objekten* und *Analyse-Objekten*. Später verwenden wir dann nur den einfachen Begriff des Objekts, wenn aus dem Kontext der richtige Bezug klar sein sollte. Ein tatsächliches Objekt ist dabei für unsere Zwecke ein zusammenhängender Abschnitt im Speicher, der für die Programmausführung eine logische (Daten-)Einheit bildet. Der Programmcode bezieht sich auf diese Objekte über programmiersprachliche Mittel wie Variablen und Zeigerindirektion.

Passend zum Fokus auf C-Programme im Rest der Arbeit beziehen wir uns hier auch auf die Objekte, welche für C-Programme behandelt werden müssen. Diese tatsächlichen Objekte wollen wir im Folgenden kurz auflisten. Wir konzentrieren uns dabei auf solche tatsächlichen Objekte, die im Programm selbst verankert sind oder von diesem angelegt werden; die Betrachtung eines beliebigen Adressbereichs als Objekt ist ebenso wie der Umgang mit extern verankerten Objekten (z.B. unbekannte Zeigerziele der Rückgabe von Bibliotheksfunktionen) ein Problem für sich und wird hier nur pauschal über Platzhalter für das Unbekannte behandelt werden.

Die tatsächlichen Objekte lassen sich nun danach klassifizieren, ob sie auf dem Stack oder auf dem Heap liegen. Ein Heapobjekt entsteht dynamisch durch eine Allokation (im Falle von C also durch einen Aufruf bekannter

Systemfunktionen wie `malloc`). Stackobjekte dagegen sind im Programm verankert. Während Heapobjekte anonym sind und nur durch Zeiger-indirekte Zugriffe (oder explizite Nutzung von Adressen) erreicht werden, sind die meisten Stackobjekte benannt und direkt über ihren Namen erreichbar. Das gilt für lokale und globale Variablen ebenso wie für Parameter. Auch Unterprogramme können wir als benannte Stackobjekte betrachten, wobei vor allem diejenigen interessieren, deren Adresse genommen wird. Die Funktionsrückgabe betrachten wir als anonymes tatsächliches Objekt, welches über `return`-Anweisungen modifiziert und bei einem Funktionsaufruf gelesen wird. Stringliterale sind ebenfalls anonyme Stackobjekte. Gleiches gilt für Aggregate, die in C zur Initialisierung von Arrays und Strukturen auftreten können.

Die tatsächlichen Objekte lassen sich auch danach einteilen, ob sie atomar oder strukturiert sind. Strukturierte Objekte sind dabei entweder Strukturen oder Varianten (`union`) und können auch genestet auftreten. Weiterhin sind Arrays eine Form der strukturierten Objekte. Die Literatur zu Zeiger- und Datenfluss-Analysen kennt vor allem für diese strukturierten tatsächlichen Objekte grobe Vereinfachungen bei der Modellierung als Analyse-Objekte.

3.1.1 Abbildung auf Analyse-Objekte

Wir bilden die genannten tatsächlichen Objekte wie folgt auf Analyse-Objekte ab:

- *Atomare Variablen und Parameter* werden über jeweils ein Stackobjekt modelliert.
- Die *atomare Funktionsrückgabe* wird jeweils durch ein Rückgabeobjekt modelliert.
- *Unterprogramme* werden auf jeweils ein Funktionsobjekt abgebildet.
- *Strukturen* werden im feld-insensitiven Modell auf ein einzelnes Analyse-Objekt pro Struktur abgebildet. Im feld-sensitiven Modell dagegen bilden wir jede Komponente der Struktur rekursiv auf ein oder mehrere Analyse-Objekte ab (abhängig von der Art der Komponente).

- *Varianten* werden im feld-insensitiven Modell auf ein einzelnes Analyse-Objekt pro Variante abgebildet. Im feld-sensitiven Modell dagegen bilden wir als Zwischenschritt jede Komponente der Variante rekursiv auf ein oder mehrere Analyse-Objekte ab. Abschließend benutzen wir dann von der Komponente, welche die meisten Analyse-Objekte erzeugt hat, die zugehörigen Analyse-Objekte als Modellierung für die gesamte Variante. Definiert das Programm also z.B.

```
typedef struct {int a; int b;} s;
typedef struct {int c; float d;} t;
typedef union {s x; t y;} myunion;
```

  so legt das feld-sensitive Modell für jede Instanz von *myunion* genau zwei Objekte an: Eines, das a und c vereint, sowie eines, das b und d vereint.

- *Arrays* betrachten wir wie ein einzelnes Objekt des Elementtyps. Bilden wir also Objekte des Elementtyps auf nur ein Analyse-Objekt ab, so modellieren wir das Array als ein einzelnes Arrayobjekt, und analog bei mehreren Objekten für den Elementtyp. Dies gilt auch in Varianten; wäre also im obigem Beispiel c ein Array von *int*s, würden immer noch die Aussagen von oben zur Anzahl der Objekte gelten.

- *Heapobjekte* betrachten wir wie eine Variante über alle Datentypen des Programmes, d.h. wir nehmen als Typ den größten Typ im Programm an. Feld-insensitiv ergibt dies lediglich ein Analyse-Objekt pro Heapobjekt, feld-sensitiv jedoch ein Analyse-Objekt pro „ausgeflachtem" Feld der (nach Zahl der zugeordneten Analyse-Objekte) größten Struktur. Wir benutzen dabei das Modell, pro Allokationsstelle nur ein Objekt anzunehmen. (Dies passt zu unserem kontext-insensitiven Algorithmus; eine kontext-sensitive Analyse kann hier eine höhere Präzision anstreben, was jedoch einige Veränderungen an den Algorithmen benötigt.)

- *Stringliterale* bilden wir zusammen auf ein einziges Stringobjekt ab.

- Die Adresse 0, die in C für den *Nullzeiger* benutzt wird, bilden wir auf ein einziges Nullobjekt ab.

- Andere tatsächliche Objekte (z.B. unbekannte Zeigerziele der Rückgabe von Bibliotheksfunktionen) bilden wir auf ein einziges Objekt mit Namen *Unknown_ Objects* ab.

Die Modellierung orientiert sich dabei am Modell von Pearce u. a. [2007], welches als das derzeit effizienteste für C gilt (für Feld-Sensitivität). Die feld-sensitive Ausprägung entspricht dabei einer Ausflachung der Strukturen, so dass kein Repräsentant für die Struktur oder für genestete Strukturen als Ganzes existiert. Die entstehenden Analyse-Ojekte für die ausgeflachten Komponenten werden dabei numeriert, so dass wir im Weiteren von Feld n einer Struktur reden können. Diese Modellierung beherrscht die C-Semantik, dass ein Zeiger auf eine Struktur auch als Zeiger auf das erste Feld betrachtet werden kann, sowie die Forderung, dass identische Anfangsstücke verschiedener Komponenten einer Variante als gemeinsamer Teil dieser Komponenten auftreten (im Sinne einer Zuweisung über eine Komponente, gefolgt von einer Verwendung über eine andere Komponente).

In Sprachen wie Java, bei denen Heapallokationen typisiert sind, lässt sich natürlich für jede Allokationsstelle ein Heapobjekt des richtigen Typs anlegen. In C dagegen kann eine Allokationsstelle Objekte unterschiedlichen Typs hervorbringen, so dass konservativ die Annahme eines beliebigen Typs des Programmes getroffen wurde. Die Zusammenlegung der Strings zu einem Objekt erfolgte aus Effizienzgründen (vgl. [Pearce u. a. 2007, S. 21]).

Wird eine niedere Zwischendarstellung als Eingabe für die Analyse genutzt (vgl. Abschnitt 3.2), so kommen Hilfsvariablen hinzu, um Zwischenergebnisse aufzunehmen. Prinzipiell können diese wie andere Variablen betrachtet werden. Zum größten Teil sind sie jedoch so einfach, dass sie eine fluss-insensitive Behandlung selbst bei fluss-sensitiver Genauigkeit erlauben. Dies ist für eine Hilfsvariable h der Fall, wenn

1. es nur eine einzige direkte Definition für h gibt, und

2. weder von h noch (im feld-sensitiven Modell) von einer anderen Hilfs-

variable zur Modellierung der Felder der gleichen Struktur die Adresse genommen wird.

Dann gibt es genau eine direkte und keine indirekte Definition für h, weswegen die fluss-insensitive Behandlung die gleiche Präzision erreicht wie eine fluss-sensitive Behandlung. Andere Hilfsvariablen, welche die Bedingungen nicht erfüllen, modellieren wir als normale Stackobjekte.

Zusätzlich zu den bisher genannten Objektarten unterstützt die Beschreibung der Analyse in dieser Arbeit auch *Objektgruppen*. Eine Objektgruppe ist dabei schlicht die Zusammenfassung einer Menge von Objekten, wobei wir mindestens zwei Elemente in der Menge annehmen (andernfalls entsteht kein Vorteil). Wir benutzen sie, um bei der Propagierung von Zeigerzielen nur einmal die Gruppe zu propagieren, anstelle der Propagierung aller einzelnen Objekte. Dies ist vor allem für Programme nützlich, in denen Strukturen oder Arrays über Aggregate initialisiert werden, wobei diese Aggregate die Adresse verschiedener Objekte nehmen. In solchen Fällen lassen sich oft die Objekte zusammen betrachten, deren Adressen genommen werden.

Beispiel 3.1.1 Betrachten wir die Initialisierung eines Arrays von Funktionszeigern:

```
symtab_func_ptr symtab_functions[]=
{&f1,&f2,...,&fn};
```

Da wir das Array nur über ein Analyse-Objekt modellieren, erzeugt dies die Instruktionen

```
symtab_functions = &f1;
symtab_functions = &f2; //partial update
...
symtab_functions = &fn; //partial update
```

Ab der zweiten Zuweisung müssen wir hier schwache Aktualisierungen benutzen (vgl. Abschnitt 3.1.2 unten). Wir können jedoch sowohl die Instruktionenzahl reduzieren als auch die Propagierung der Zeigerziele später beschleunigen, wenn wir diese Initialisierung über eine Objektgruppe zusammenfassen:

```
symtab_functions = &group1
```

Wobei hier $f1, f2, \ldots, fn$ als $group1$ zusammengefasst wurden. (Die Objekte sind jedoch für andere Situationen noch normal, d.h. ohne Zusammenfassung, verfügbar. Objektgruppen sind nur zusätzliche Objekte für Situationen wie die gezeigte, die durchaus häufig anzutreffen sind.) □

Tabelle 3.1 stellt zusammenfassend die verschiedenen Objekt*arten* zusammen, welche die Analyse damit unterscheidet. Einige dieser Objektarten behandelt die Analyse so, dass sie zwar als Zeigerziele und als Operanden der Adressnahme auftauchen, aber nicht als Objekte an Definitionen und Verwendungen (es gibt keine Knoten für Literale). Dies zeigt die letzte Spalte der Tabelle an.

Objektart	Objekte repräsentieren...	Datenfluss?
Funktion	ein Unterprogramm	nein
Rückgabe	Rückgabe eines Unterprogrammes	ja
Heap	alle Objekte einer Allokationsstelle	ja
Hilfsvariable	einfache Hilfsvariable	ja (FI)
Stack	Variable aus keiner der übrigen Kategorien	ja
Array	alle Elemente des Arrays	ja
String	alle Stringliterale des Programmes	nein
Null	Ziel eines Nullzeigers	nein
Unbekannt	nicht näher bekannte Objekte	nein
Gruppe	mind. 2 Objekte	(je Objekt)

Tabelle 3.1: Arten von Analyse-Objekten

Für die vorliegende Dissertation haben wir uns entschieden, nur sehr magere Typinformationen zu benutzen, da C für eine konservative Analyse keine großen Zusicherungen auf Typbasis bietet. Insbesondere vermerken wir daher keinen Typ zu den einzelnen Objekten, abgesehen von der in der Objektart codierten Information. In typsicheren Sprachen wie Java sollten jedoch die Typinformationen genutzt werden: Sie erlauben eine Ausfilterung einiger durch Überschätzung fälschlich angenommener Zeigerziele.

3.1.2 Schwache Aktualisierungen

Die Verwendung eines einzelnen Analyse-Objekts zur Repräsentation mehrerer tatsächlicher Objekte muss an einigen Stellen für eine korrekte Analyse bedacht werden. Immer dann, wenn ein Analyse-Objekt mehrere tatsächliche Objekte darstellt, die in einer Aufführung *zugleich* aktiv sein können (und im Programm zugreifbar sind), ist Vorsicht geboten. Das folgende Beispiel soll dies verdeutlichen:

```
s.a = 1;
s.b = 2;
printf (s.a);
```

Hier benutzen wir feld-insensitiv ein Analyse-Objekt zur Modellierung der beiden Felder `s.a` und `s.b`. Die Ausgabe am Schluss muss sich jedoch auf die erste Definition beziehen, auch wenn die zweite Definition aus Sicht der Analyse das gleiche Objekt modifiziert. Wir lösen dies über schwache Aktualisierungen: Betrachten wir die zweite Definition als schwach, so erreichen beide Definitionen die abschließende Verwendung. Die Korrektheit wird dabei mit einer größeren Ungenauigkeit erkauft. Für die folgenden Objekte benutzen wir auf diese Weise schwache Aktualisierungen:

- Arrayobjekte erzwingen schwache Aktualisierungen.
- Heapobjekte erzwingen schwache Aktualisierungen.
- Stack- und Hilfsobjekte zur Darstellung von Strukturen und Varianten erzwingen feld-insensitiv schwache Aktualisierungen.

In manchen Fällen ist es möglich, anhand der jeweiligen Instruktion dennoch für die Array- und strukturbezogenen Objekte starke Aktualisierungen zu benutzen. Diese Fälle sind (indirekt nur bei genau einem Ziel):

1. Direkte oder indirekte Strukturkopien: Diese Instruktionen modifizieren immer die gesamte Struktur, so dass (feld-insensitiv) das einzelne Objekt zur Darstellung der Struktur vollständig überschrieben wird.

2. Initialisierung mit Aggregaten: Die Initialisierung eines Arrays oder einer Struktur über ein Aggregat definiert dieses ebenfalls vollständig, so dass eine starke Aktualisierung vorliegt. Für Arrays realisieren wir dies so, dass die Zuweisungen im Rahmen des ersten Array-Elements starke Aktualisierungen sind. Die nachfolgenden Zuweisungen müssen jedoch schwach bleiben (vgl. Beispiel 3.1.1). Für Strukturinitialisierungen im feld-insensitiven Modell können wir analog die Belegung des ersten Feldes über eine starke Aktualisierung modellieren.

Um diese Situationen ausnutzen zu können, nehmen wir entsprechende Informationen in der Zwischendarstellung an. Außerdem sei auf Kapitel 8 verwiesen, das sich ausführlich mit der Thematik schwacher / starker Aktualisierungen auseinandersetzt.

3.1.3 Zeigerziele

Zur Klarheit wollen wir hier ein wenig Terminologie im Zusammenhang mit Zeigerzielen einführen:

Definition 3.1.2 Ein *Zeigerliteral* ist die Adresse eines Analyse-Objekts. Umgekehrt ist ein Analyse-Objekt ein *aktuelles Zeigerziel*, wenn es bereits ein Zeigerliteral für seine Adresse gibt.

Zeigerliterale entstehen dabei vor allem durch die explizite Adressnahme. Diese kann direkt (wie in `x = &y`) oder indirekt (wie in `x = &(p->f)`) erfolgen. Insgesamt erhalten wir die folgenden Möglichkeiten, an denen Zeigerliterale entstehen:

Definition 3.1.3 Eine *Zeigerziel-Quelle* ist eine dieser Instruktionen:

- Die Zuweisung der Adresse eines Objekts oder (indirekt) eines Feldes.
- Die Zuweisung des Nullzeiger-Literals.
- Die Allokation eines Heapobjekts.

- Der Aufruf einer externen Funktion mit Rückgabe, wenn für diese keine Effekte bekannt sind und für die Rückgabe pessimistisch abgeschätzt werden sollen.

- Die Zuweisung von Unknown_Objects über eine indirekte Verwendung (wie in `x = *y`).

Die letzten beiden Arten von Quellen erzeugen dabei stets ein Zeigerliteral für Unknown_Objects.

Die Menge der Zeigerziel-Quellen und darüber auch die Menge der Zeigerliterale bzw. der aktuellen Zeigerziele kann im Laufe der Analyse zunehmen. So könnte z.B. ein indirekter Aufruf als Ziel eine externe Funktion erhalten, und feld-sensitiv wächst die Menge der Zeigerliterale an einer Anweisung wie `x = &(p->f)` mit jedem für p erkannten Ziel. An einigen Stellen werden wir uns dafür interessieren, wie groß die Menge der aktuellen Zeigerziele werden kann. Diese obere Schranke wollen wir als die Menge der *potentiellen* Zeigerziele bezeichnen und im Folgenden einschränken.

Nur ein Teil der Analyse-Objekte kommt prinzipiell als Zeigerziel in Frage. Heap- und Arrayobjekte, Objektgruppen sowie die besonderen Objekte Unknown_Objects, String und Null sind stets potentielle Zeigerziele. Bei Arrays entspricht die Zuweisung der Arrayvariable an einen Zeiger der Adressnahme des Arrays; ebenso ist der indizierte Zugriff `a[i]` zunächst die Adressnahme des Arrayobjekts a, wobei anschließend auf diese Adresse ein Offset gemäß dem Index addiert wird, bevor die Dereferenzierung stattfindet.

Funktions- und Stackobjekte sind nur dann potentielle Zeigerziele, wenn ihre Adresse irgendwo genommen wird. Im feld-sensitiven Modell kann ein Objekt o zur Modellierung des Feldes i einer Struktur s darüber hinaus auch dann zum Zeigerziel werden, wenn dies bereits für ein Objekt o' zu einem anderen Feld der gleichen Struktur s gilt (z.B. über Anweisungen der Form `x = &(p->f)`). Dies lässt sich noch bei Strukturen (nicht aber bei Varianten) über den Feldnamen in derartigen Anweisungen einschränken sowie darüber, dass das in o' repräsentierte Feld eine kleinere Nummer als i haben muss.

Hilfsobjekte und Rückgabeobjekte dagegen treten nie als Zeigerziele auf.

3.1.4 Datenstrukturen

Wir modellieren die Analyse-Objekte in einem Array *Objects*, welches pro Analyse-Objekt einen Eintrag enthält. Die Instruktionen der Zwischendarstellung sowie die weiteren Datenstrukturen der Analyse können sich damit über den entsprechenden Array-Index auf ein Objekt beziehen. Den besonderen Index 0 verwenden wir für ein nicht näher beziffertes, irrelevantes Objekt (zur Markierung von nicht initialisierten Objekten; vgl. Abschnitt 3.2.2). Die Objekte zur Repräsentation einer Struktur belegen benachbarte Einträge im Array, so dass ein Index-Intervall die gesamte Struktur (und nur diese, inklusive genesteter Strukturen) abdeckt (vgl. Beispiel 3.1.4). Weiterhin sollen auch die Analyse-Objekte zur Repräsentation der Parameter eines Unterprogrammes gemäß ihrer Reihenfolge benachbarte Einträge belegen.

Zusätzlich benutzen wir ein zweites Array *Pointer_ Targets* zur Darstellung der Zeigerziele. Dieses zweite Array enthält dabei pro aktuellem Zeigerziel einen Eintrag, welcher lediglich den zugehörigen Index des betroffenen Objekts in *Objects* angibt. Die Reihenfolge der Einträge in diesem Array ist unerheblich. Zeigerziele können damit in der Analyse über den Index in dieses Array benannt werden. Die Verwendung dieses – deutlich kleineren – zusätzlichen Arrays erlaubt dabei einen kompakteren Indexbereich und bedeutet so z.B. bei Verwendung von Bitvektoren zur Darstellung einer Menge von Zeigerzielen eine Platzersparnis.

Beide Arrays sind potentiell dynamisch: Bei feiner Heapmodellierung (vgl. Kapitel 9) oder einer präziseren Modellierung im Zusammenhang mit rekursiven Funktionen (vgl. Kapitel 7) steigt die Zahl der Objekte im Laufe der Analyse, und bei feld-sensitiver Analyse steigt auch ohne dies die Zahl der aktuellen Zeigerziele. Die in dieser Arbeit besprochene Umsetzung beherrscht letzteres und kommt mit einer festen Menge an Objekten aus.

Pro Objekt nehmen wir die in Tabelle 3.2 genannten Attribute an. Das Referenz-Attribut nutzen wir je nach Objektart unterschiedlich. Tabelle 3.3 listet dies näher auf. Die hier erwähnten weiteren Arrays *Routines*, *Instructions* und *Object_ Groups* sind dabei Datenstrukturen zur Ablage von Informationen über alle Unterprogramme bzw. Instruktionen bzw. Objektgruppen.

Attribut	Kurzbeschreibung
Typ	Die Art des Objekts; vgl. Tabelle 3.1.
Feld	Pro Struktur werden feld-sensitiv Objekte für alle Felder angelegt. Diese erhalten dann eine fortlaufende Nummer, beginnend mit 1 für jedes strukturierte Objekt. In allen übrigen Fällen ist dieses Attribut 0.
Zielindex	Index im Array *Pointer_ Targets* (0, falls nicht zutreffend).
Referenz	Dieser Zahlenwert ist ein Verweis auf weitergehende Informationen. Je nach Objektart hat er unterschiedliche Bedeutung; vgl. Tabelle 3.3.

Tabelle 3.2: Attribute pro Objekt

Objektart	Bedeutung
Funktion	Index des zugehörigen Eintrags im Array *Routines*
Rückgabe	Index der zugehörigen Funktion in *Routines*
Heap	Index der Allokationsstelle in *Instructions*
Hilfsvariable	Index der einzigen Definition in *Instructions*
Stack	Index der zugehörigen Funktion in *Routines*
Array	Index der zugehörigen Funktion in *Routines*
String	ungenutzt
Null	ungenutzt
Unbekannt	ungenutzt
Gruppe	Index der Gruppe in *Object_ Groups*

Tabelle 3.3: Verwendung des Attributs *Referenz* je nach Objektart

Beispiel 3.1.4 Nehmen wir folgende Strukturen an:

```
struct s1 {int c; int d;};
struct s2 {int e; char f;};
struct s
{
    int a;
    int b[7];
    union u
    {
        struct s1 x;
        struct s2 y;
        int z[100];
    }
}
```

Nehmen wir weiterhin eine Deklaration `struct s object;` an und gehen davon aus, dass die Objekt-Indices 1 bis 99 bereits vergeben sind. Dann werden feld-sensitiv die folgenden Objekte an den jeweiligen Indices angelegt:

Index	Objektart	Objektbeschreibung
100	Stack	Feld $object.a$
101	Array	Feld $object.b$ (gesamtes Array)
102	Array	Felder $object.u.x.c, object.u.y.e, object.u.z$
103	Stack	Felder $object.u.x.d, object.u.y.f$

Tabelle 3.4: Analyse-Objekte für das strukturierte Objekt

Die ersten Felder der verschiedenen Komponenten der Variante wurden zusammen in ein Objekt gelegt, wobei als pessimistischste Objektart das Array benutzt wird (pessimistisch in dem Sinne, dass schwache Aktualisierungen nötig werden). Die Zusammenlegung sorgt auch für die richtige Erkennung des vom C-Standard abgesegneten Datenflusses in den Anweisungen:

```
object.u.x.c = 5;
printf (object.u.y.e);
```

Der C-Standard gibt jedoch keine ähnliche Garantie im Zusammenhang mit dem Array z. Auf den Standard berufen sich auch Pearce u. a. [2007], von denen wir das Objektmodell weitgehend übernommen haben. Das Modell kann das in der Praxis wohl auftretende Aliasing zwischen den Array-Elementen und den Feldern der Strukturen nicht vollständig abbilden; stattdessen modelliert es ein Aliasing zwischen allen Array-Elementen und c bzw. e. Da uns kein effizientes Modell mit besserer Präzision in diesem Punkt bekannt ist, sind wir bei diesem Pearceschen Ansatz geblieben.

Weiterhin zeigt das Beispiel, dass feld-sensitiv kein Objekt für eine Struktur oder Variante als Ganzes angelegt wird. Stattdessen wird die Struktur auf ihre Komponenten ausgeflacht. Feld-insensitiv dagegen würde im Beispiel nur ein Objekt für *object* erzeugt, wobei wiederum als pessimistischste Objektart das Array benutzt wird.

Nehmen wir abschließend noch an, die Struktur s hätte statt wie gezeigt den folgenden, einfacheren Aufbau:

```
struct s
{
    int z[100];
    struct s1 x;
    struct s2 y;
}
```

Index	Objektart	Objektbeschreibung
100	Array	Feld $object.z$ (gesamtes Array)
101	Stack	Feld $object.x.c$
102	Stack	Feld $object.x.d$
103	Stack	Feld $object.y.e$
104	Stack	Feld $object.y.f$

Tabelle 3.5: Analyse-Objekte für *object* bei genesteten Strukturen

Tabelle 3.5 illustriert dazu die dann vorhandenen Analyse-Objekte. Wir erkennen darin, wie genestete Strukturen „ausgeflacht“ werden. □

Beispiel 3.1.5 Als weiteres Beispiel betrachten wir ein strukturiertes Array:

```
struct t {char c; int f[17];};
struct s {int x; double y; struct t inner;};
struct s myarray[42];
```

Feld-insensitiv entsteht wiederum genau ein Objekt mit der Objektart Array. Feld-sensitiv entsteht für jedes Feld ein Objekt, und als Teil eines Arrays werden diese wieder von der Objektart Array angelegt: □

Index	Objektart	Objektbeschreibung
104	Array	Feld *myarray.x*
105	Array	Feld *myarray.y*
106	Array	Feld *myarray.inner.c*
107	Array	Feld *myarray.inner.f*

Tabelle 3.6: Objekte für das strukturierte Array

3.2 Repräsentation eines Programmes

3.2.1 Lokaler Kontrollfluss und Instruktionen

Wir haben keine spezielle Anforderung an die Höhe der Zwischendarstellung: Hohe und niedere Zwischendarstellungen können gleichermaßen als Grundlage dienen. Die Beschreibung in dieser Arbeit orientiert sich jedoch an einer niederen Darstellung auf Instruktionsebene, ähnlich dem Drei-Adress-Code [Aho u. a. 1986, S. 572ff]. Diese Instruktionen nutzen wir auch als Knoten in den intraprozeduralen Kontrollfluss-Graphen (engl. *control-flow graph*, CFG; Allen [1970]; Hecht [1977]), wobei sie zur Optimierung zu Grundblöcken gruppiert werden.

Wir gehen für unsere Zwischendarstellung davon aus, dass bereits eine lokale Kontrollfluss-Analyse ausgeführt wurde. Demzufolge ist pro Unterprogramm bereits ein CFG bekannt. Das erlaubt uns, im Folgenden die Kontrollfluss-verursachenden Instruktionen (wie Schleifen, Verzweigungen und Sprünge) zu ignorieren: Die darin verwendeten Ausdrücke (z.B. für Bedingungen)

modellieren wir noch. Die eigentliche Kontrollfluss-Aktion jedoch modellieren wir nicht mehr, ihr Effekt ist in der Struktur des CFG berücksichtigt worden. Wir nehmen ferner an, dass diese Kontrollfluss-Graphen jeweils mit der bekannten „Single-Entry, Single-Exit"-Eigenschaft vorliegen, damit wir einen klar definierten Ein- und Ausgang haben.[1]

Wir werden auch voraussetzen, dass zu jedem CFG bereits der Dominanzbaum bekannt ist, sowie zu jedem Grundblock darin die Dominanzgrenze. Diese Dominanzinformationen werden wir im Zusammenhang mit der SSA-Form ausnutzen.

Attribut	Beschreibung
Typ	Art der Instruktion; vgl. Tabelle 3.8
Operanden	Array von Operanden
Stark	Markiert die Fälle aus 3.1.2, in denen eine starke Aktualisierung angenommen werden kann

Tabelle 3.7: Attribute pro Instruktion

Instruktionen. Bezüglich der Form der Instruktionen nehmen wir an, dass jede Instruktion die Attribute aus Tabelle 3.7 besitzt. Ein Operand ist dabei ein direkter Objektzugriff (`x`), ein indirekter Objektzugriff (`*x`) oder ein indirekter Feldzugriff (`(*x)+d` mit $d > 0$). Letzteres taucht nur im feld-sensitiven Modell auf und ist zu verstehen als Zugriff auf die Objekte mit Index $t + d$ für alle Objekte t, welche als Ziel von x erkannt werden. Der direkte Feldzugriff verschwindet, weil der Operand unmittelbar das zum selektierten Feld gehörende Objekt benennen kann: Haben wir im Code z.B. den Ausdruck `object.a` zum Beispiel 3.1.4, so hat unsere feld-sensitive Modellierung für dieses Feld im Beispiel ein Analyse-Objekt mit dem Index 100 angelegt. Dieses kann der Operand nun direkt benennen, der ursprüngliche Feldzugriff wird zum direkten Objektzugriff. Ein Operand besitzt damit jeweils die folgenden Attribute:

[1] Insbesondere, wenn die Analyse zusätzlich Exceptions behandeln soll, müssen diese Annahmen ggf. entsprechend angepasst werden.

- Objekt: Index ins Objects-Array
- Dereferenzierung ja/nein
- Feldoffset d für `(*x)+d`

Je nach Art der Instruktion benötigen wir unterschiedlich viele solcher Operanden. Die verschiedenen Instruktionsarten und ihre Operanden zeigt uns Tabelle 3.8. Die hier getroffenen Annahmen und Vereinfachungen erlauben es uns dabei, mit nur wenigen verschiedenen Arten von Instruktionen auszukommen.

Art	Beispiel	Operanden
Kopie	`x = y`	LHS, RHS
Adressnahme	`x = &y`	LHS, RHS
Allokation	`x = malloc(y)`	LHS, RHS
Unärer Operator	`x = op y`	LHS, RHS
Binärer Operator	`x = y op z`	LHS, RHS 1 und RHS 2
Aufruf	`x = call f (p1,...,pn)`	LHS (optional), Funktion, Argumente
Aufruf mit Struktur-Rückgabe	`x1,...,xm = call f (p1,...,pn)`	Funktion, LHS 1 bis LHS m, Argumente

Tabelle 3.8: Arten von Instruktionen

Passend zu unserem Objektmodell nehmen wir hierbei an, dass Zeigerarithmetik auf die Identität abgebildet wurde: $p \pm i$ und die Präfix-/Postfix-Versionen von $++p$ bzw. $--p$ werden in den Instruktionen wie p behandelt. Dies ist für die in C abgesicherten Verwendungen sinnvoll, nämlich beim Traversieren eines Arrays oder Strings, was in unserem Objektmodell nicht zu einem anderen Objekt wechselt. Andere unäre und binäre Operatoren interessieren uns für die Zwecke der Analyse nicht näher, so dass wir sie zu allgemeinen Operatoren abstrahiert haben.

Wir schränken ferner ein, dass lediglich bei Kopien die LHS indirekt sein darf, und dann auch nur bei direkter RHS. In allen übrigen Instruktionen soll die LHS stets eine direkte Definition sein, gegenfalls durch Einführung einer Hilfsvariablen. Außerdem seien die Argumente in Aufrufen stets ohne

Dereferenzierung, und die LHS von Aufrufen (auch mit Struktur-Rückgabe) bestehe immer aus Hilfsvariablen. Man beachte, dass Aufrufe indirekt sein können (wenn der Funktionsoperand eine Dereferenzierung aufweist), weswegen das Rückgabe-Objekt nicht in jedem Fall direkt bekannt ist. Daher können wir nicht einfach einen Aufruf trennen in den Aufruf gefolgt von Instruktionen zur Übernahme des Resultats.

Feld-Sensitivität wird in den meisten Fällen schlicht durch entsprechendes Duplizieren der Instruktion (mit Anpassung der Operanden aufs nächste Feld) erreicht. Modellieren wir ein Argument über mehrere Analyse-Objekte (z.B. eine Struktur), so besitzt die Funktion in der IR entsprechend viele formale Parameter und die Argumentliste übergibt die Objekte für alle Felder einzeln. Bei Aufrufen mussten wir jedoch eine besondere Form annehmen, wenn die Rückgabe eine Struktur ist und die Analyse feld-sensitiv abläuft, da wir nicht den Aufruf mehrfach sehen wollen.

Die Heapallokation haben wir analog zur Adressnahme modelliert. Der Operand der RHS ist nie eine Dereferenzierung und benennt das Heapobjekt, welches an dieser Stelle allokiert wird.

Will man Zeigerziele in der Analyse vorwärts propagieren (vgl. Kapitel 5) und dabei die unbekannten Ziele der Rückgabe externer Funktionen erfassen, so sollte die IR hierzu nach dem Aufruf einer externen Funktion die Zuweisung der Adresse von *Unknown_ Objects* an die Hilfsvariable zur Aufnahme der Rückgabe enthalten.

Beispiel 3.2.1 Seien wieder die Typen aus Beispiel 3.1.4 gegeben sowie das zugehörige Objekt *object*. Betrachten wir in diesem Kontext den Zugriff auf ein Array-Feld. Beispielweise wird aus dem Ausdruck `object.b[2]`:

```
t1 = &101   // decay-to-pointer
t2 = t1     // Zeigerarithmetik als Identitaet modelliert
t3 = *t2    // Elementzugriff
```

Dies entspricht der C-Semantik, dass ein Array bei einem solchen Zugriff zunächst als Zeiger auf das erste Element betrachtet wird. Diesen sogenannten *decay-to-pointer* sehen wir in der ersten Zeile als Adressnahme des Objekts 101, das im Beispiel das Feld *object.b* repräsentiert. Auf diesen wird

als Offset der Array-Index (multipliziert mit der Elementtyp-Größe) addiert. Unsere Modellierung der Zeigerarithmetik bildet diese Addition jedoch auf die Identität ab. Abschließend erfolgt der Elementzugriff als Dereferenzierung des berechneten Zeigers. Eine Optimierung vorab kann natürlich feststellen, dass in diesem Fall direkt das Objekt 101 verwendet wird, und den Code entsprechend vereinfachen. Unsere Implementierung verwendet eine solche Optimierung.

Analog erfolgt die Modellierung mit variablem Index. So entsteht für `object.b[i]` die gleiche Sequenz, wobei zusätzlich die Verwendung von i explizit modelliert werden kann, damit die Analyse erkennt, dass dafür Datenfluss bestimmt werden sollte.

Für die Situation aus Beispiel 3.1.5 können wir den Zugriff auf ein strukturiertes Array betrachten: Der Zugriff `myarray[7].inner.f[1]` wird zu (104 war der Objektindex fürs erste Feld):

```
t1 = &104        // decay-to-pointer (myarray)
t2 = t1          // Zeigerarithmetik als Identitaet modelliert
t3 = (*t2)+3     // Elementzugriff mit Feldzugriff
t4 = &t3         // decay-to-pointer (f)
t5 = t4          // Zeigerarithmetik
t6 = *t5         // Elementzugriff
```

Hier ist der Elementzugriff `myarray[7]` mit einer Indexverschiebung gekoppelt, die den Zugriff auf das Feld `inner.f` darstellt. Der indirekte Feldzugriff ist für die Analyse damit lediglich eine Addition des richtigen (statisch bekannten) Offsets auf den Index des Objekts, welches als Ziel der Dereferenzierung erkannt wird. Anschließend sehen wir eine zweite Array-Zugriffssequenz, um auch den Teil `f[7]` richtig abzubilden. Wiederum kann eine Optimierung diesen Code vereinfachen, und zwar zum direkten Zugriff auf Objekt 107. □

3.2.2 Nicht initialisierte Objekte

Zur Vereinfachung nehmen wir an, dass die IR besondere Instruktionen enthält, um den Beginn des Lebensbereiches von Variablen zu markieren. Diese Instruktionen sind jeweils eine starke Aktualisierung der Form `x = 0`, d.h. die Zuweisung eines nicht näher bezifferten Objekts. Sie werden uns als Pseudo-Definitionen für nicht initialisierte Variablen dienen. Ihre Existenz vereinfacht die Beschreibung insofern, dass nun stets lokal eine Definition gefunden wird, wenn es sich um eine lokale Variable handelt (bzw. in der Programmwurzel für globale Objekte). Finden wir keine lokale Definition, so können wir umgekehrt auf ein nicht-lokales Objekt schließen.

Für Heapobjekte ist es naheliegend, analog an der Allokationsstelle eine künstliche Definition anzunehmen. Aufgrund der üblichen Analyse-Ungenauigkeit, welche pro Allokationsstelle ein Objekt annimmt, müsste diese jedoch eine schwache Aktualisierung sein:

```
x = (int)malloc_wrapper();
*x = 5;
y = (int)malloc_wrapper();
z = *y;
printf (*x);
```

In diesem Beispiel sei in `malloc_wrapper` die Allokation ohne explizite Initialisierung untergebracht. Die Analyse wird nun für `*x` und `*y` das gleiche Analyse-Heapobjekt benutzen. Daher darf keine starke Aktualisierung für `*y` als künstliche Definition stattfinden, denn sonst bezieht sich die nachfolgende Verwendung von `*x` fälschlicherweise darauf. Mit einer schwachen Aktualisierung jedoch bezieht sich die Verwendung von `*y` sowohl auf die künstliche Definition als auch auf die Zuweisung an `*x`. Da gleiches für die Verwendung von `*x` gilt, gibt es auf diese Weise keine präzise Erkennungsmöglichkeit, ob die Verwendung potentiell auf ein nicht initialisiertes Objekt zugreift.

Daher nehmen wir für Heapobjekte keine künstliche Definition an der Allokation an. Stattdessen setzen wir diese am Programmbeginn, d.h. in der Wurzel des Aufrufgraphen (als starke Aktualisierung) voraus, wie für globale Variablen.

3.3 Interprozedurale SSA-Form

Nach der gewünschten Eingabe für die Analyse beschreiben wir nun noch etwas detaillierter die berechnete Ausgabe. Prinzipiell glauben wir, dass die Analyse mit verschiedenen Datenfluss-Darstellungen arbeiten kann. Aufgrund einiger schöner Eigenschaften beschreiben wir sie jedoch spezifisch für die *interprozedurale SSA-Darstellung*, kurz ISSA [Staiger u. a. 2007]. Diese Darstellung ist kompakt (engl. *sparse*), was sowohl den Platzbedarf reduziert als auch die Laufzeit positiv beeinflusst. Außerdem erlaubt sie effiziente Datenstrukturen und elegante Algorithmen, wie wir später sehen werden. Gegenüber der klassischen (intraprozeduralen) SSA-Form kommen vor allem Seiteneffekte hinzu.

3.3.1 Erweiterung der klassischen SSA-Form

Grundlage unserer ISSA-Darstellung ist die bekannte intraprozedurale Datenfluss-Darstellung mit statischen Einmalzuweisungen (engl. *static single assignment*, SSA) [Cytron u. a. 1991]. Die SSA- und die ISSA-Form sind gerichtete Graphen, wobei Kanten einen potentiellen Datenfluss darstellen. Die Knoten symbolisieren wichtige Aktionen für den Datenfluss, vor allem Definitionen und Verwendungen.

Die SSA-Form enthält neben Definitionen und Verwendungen auch sogenannte ϕ-Knoten, um die Eigenschaft zu erreichen, dass sich eine Verwendung stets statisch auf genau eine Definition bezieht. Ein ϕ-Knoten wird dazu an Konfluenzpunkten des CFG eingesetzt, wenn dort über die zusammenkommenden Pfade verschiedene Definitionen eines Objekts vorliegen würden. Ein solcher Knoten ist dann eine künstliche neue Definition des Objekts. Konzeptionell können wir ihn als zusammengesetzten Knoten beschreiben, der für jeden Vorgänger des Konfluenzpunktes eine künstliche Verwendung als Eingang (*Argument*) besitzt. Der ϕ-Knoten selbst sitzt am Anfang des Konfluenzpunktes und stellt dort als Ausgang eine Definition bereit.

SSA wird oft so beschrieben, dass abschließend über eine Numerierung die Variablen umbenannt werden, so dass jede Variable genau eine Definition besitzt. Wir ersparen uns dies und benutzen statt Variablen Definitionen.

Ein ISSA-Knoten benennt stets das betroffene Objekt sowie den Operanden der Instruktion, so dass dies einheitlich möglich ist. So erkennen wir auch den Zugriff auf eine globale Variable in verschiedenen Unterprogrammen als Bezug auf die gleiche Variable, während die Variablen-Umbenennung hier Schwierigkeiten bereitet (ϕ-Knoten und die in Kürze erläuterten weiteren induzierten Knoten führen dort nämlich neue Variablen ein).

Für ein Unterprogramm in SSA-Form gelten ein paar nützliche Eigenschaften, die wir hier kurz zusammenfassen wollen. Für Beweise kann der Leser in vielen Werken nachschlagen, z.B. im Klassiker von Cytron u. a. [1991].

- Jede Verwendung bezieht sich statisch auf genau eine Definition. Graphentheoretisch gesprochen hat ein Knoten für eine Verwendung also stets Eingangsgrad 1.
- Eine Definition dominiert (im CFG) alle mit ihr direkt verbundenen Verwendungen.
- ϕ-Knoten befinden sich stets in der iterierten Dominanzgrenze (zum Grundblock) einer Definition.

Von diesen Eigenschaften werden wir regen Gebrauch machen, sowohl in der algorithmischen Umsetzung als auch später bei der Komplexitätsanalyse.

Typischerweise beschreiben Publikationen die SSA-Darstellung so, dass Kanten lediglich von Definitionen (oder ϕ-Knoten) zu Verwendungen (oder ϕ-Knoten) führen bzw. in umgekehrter Richtung angelegt sind. Wir ergänzen hier (zumindest konzeptionell) die *Zuweisungsbrücken*, welche den Datenfluss von der rechten zur linken Seite einer Kopierinstruktion erfassen. Außerdem erweitern wir die klassische SSA-Form, um schwache Aktualisierungen zu repräsentieren: Ist eine Definition d schwach, so betrachten wir sie konzeptionell als zusammengesetzt aus einer Verwendung als Eingang und der eigentlichen Definition als Ausgang. Für den Eingang gibt es dann wie für eine normale Verwendung eine Definition p, und wir verbinden diese mit der schwachen Aktualisierung über eine Kante $p \to d$. Damit stellen wir auch sicher, dass Zeigerziele vorausgehender Definitionen (wie p) zu allen Verwendungen von d fließen.

Beispiel 3.3.1 Betrachten wir folgenden Code:

```
s.f = 1;
if (...)
{
    s.g = value;
}
printf (s.f);
```

Eine feld-insensitive Modellierung benutzt hier ein Analyse-Objekt s für alle Felder der Struktur. Entsprechend sind die Zuweisungen an diese Felder als schwache Aktualisierungen von s zu modellieren. Damit ergibt sich folgender ISSA-Ausschnitt:

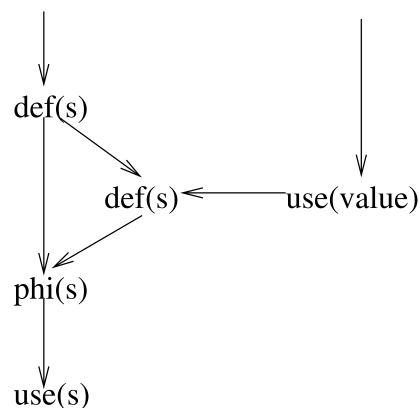


Abbildung 3.1: ISSA-Graph zum Codebeispiel

Wir haben hierbei die Kanten in Datenfluss-Richtung gezeichnet. Die beiden Definitionen sind schwach, was an der jeweils eingehenden Kante zu erkennen ist. Das Beispiel illustriert außerdem, dass wir keine Numerierung der Variablen zur Unterscheidung der einzelnen Definitionen benutzen. Stattdessen behandeln wir die Definitionen (und alle übrigen ISSA-Knoten) als Knoten in einem Graphen. Das Beispiel zeigt zudem, dass neben den Kanten zwischen Definitionen und Verwendungen auch Zuweisungsbrücken im ISSA-Graphen als Kanten auftauchen. □

ISSA erweitert nun diese klassische SSA-Form um Elemente zur Erfassung von Datenfluss über Unterprogramm-Grenzen hinweg. Dies umfasst ein- und ausgehende Parameter (inklusive der Funktionsrückgabe) und ein- und ausgehende Seiteneffekte.

Eingehende Parameter und Seiteneffekte. Für eingehenden Datenfluss von einer Aufrufstelle c zu einem Unterprogramm f existieren Knoten sowohl unmittelbar vor dem Aufruf als auch im Startknoten des CFG von f. Im CFG-Startknoten stellen sie eine künstliche Definition für den Parameter oder das Seiteneffekt-Objekt dar. Diese künstlichen Startdefinitionen nennen wir die *Eingänge* von f (engl. *entry*). Komplementär dazu besitzt die ISSA-Form an c Verwendungen für Argumente und aktuelle eingehende Seiteneffekte. Diese bezeichnen wir als *Unterausgänge* an c (engl. *subexit*). Von einem Unterausgang führt pro Aufrufziel eine Kante zum korrespondierenden Eingang des Aufrufziels. Bei mehr als einer Aufrufstelle für f haben die Eingänge damit ähnlich zu einem ϕ-Knoten mehrere Vorgänger. Tatsächlich bilden Unterausgänge und Eingänge zusammen die konzeptionelle Zerlegung eines ϕ-Knotens in Verwendungen für jeden Vorgänger und eine ausgehende Definition.

Ausgehende Parameter und Seiteneffekte. Für ausgehenden Datenfluss von einem Unterprogramm f zurück zu einer Aufrufstelle c existieren Knoten sowohl unmittelbar nach dem Aufruf als auch im Endknoten des CFG von f. Im CFG-Endknoten stellen sie eine künstliche Verwendung für den Parameter oder das Seiteneffekt-Objekt dar. Diese künstlichen Endverwendungen nennen wir die *Ausgänge* von f (engl. *exit*). Komplementär dazu besitzt die ISSA-Form an c Definitionen für die aufnehmenden Argumente und aktuelle ausgehende Seiteneffekte. Diese bezeichnen wir als *Untereingänge* an c (engl. *subentry*). Im lokalen Kontrollfluss des Aufrufers kommen sie stets nach allen Unterausgängen (und dem eigentlichen Aufruf) an c. Von einem Ausgang führt pro Aufrufstelle eine Kante zum korrespondierenden Untereingang der Aufrufstelle. Bei mehr als einem Aufrufziel für c (indirekter Aufruf) haben die Untereingänge damit ähnlich zu einem ϕ-Knoten

mehrere Vorgänger. Tatsächlich bilden Ausgänge und Untereingänge zusammen die konzeptionelle Zerlegung eines ϕ-Knotens in Verwendungen für jeden Vorgänger und eine ausgehende Definition.

Beispiel 3.3.2 Betrachten wir einen erweiterten Ausschnitt zum Beispiel 3.3.1:

```
int func (int value)
{
    s.f = 1;
    if (...)
    {
        s.g = value;
    }
    printf (s.f);
    return s.g;
}

void g ()
{
    int x = func (y);
    printf (s.f);
    ...
}
```

Damit wollen wir speziell die interprozeduralen Anteile der ISSA-Form näher beleuchten. Abbildung 3.2 zeigt den zugehörigen ISSA-Ausschnitt.

Auf der linken Seite ist in der Abbildung der Teilgraph zu $func$ dargestellt, auf der rechten Seite der Ausschnitt zu g. Intraprozedurale Kanten sind wie bisher durchgezogen, während interprozedurale Kanten gestrichelt gezeigt werden.

In $func$ erkennen wir im Vergleich zu Beispiel 3.3.1 die zusätzlich vorhandenen Eingänge für s und $value$ sowie die Knoten zur Darstellung der return-Anweisung. Außerdem liegt auch ein May-Def-Seiteneffekt für s vor, weswegen ein Ausgang für s hinzugekommen ist. Wir sehen, dass sich Eingänge wie Definitionen verhalten, Ausgänge dagegen wie Verwendungen. Der

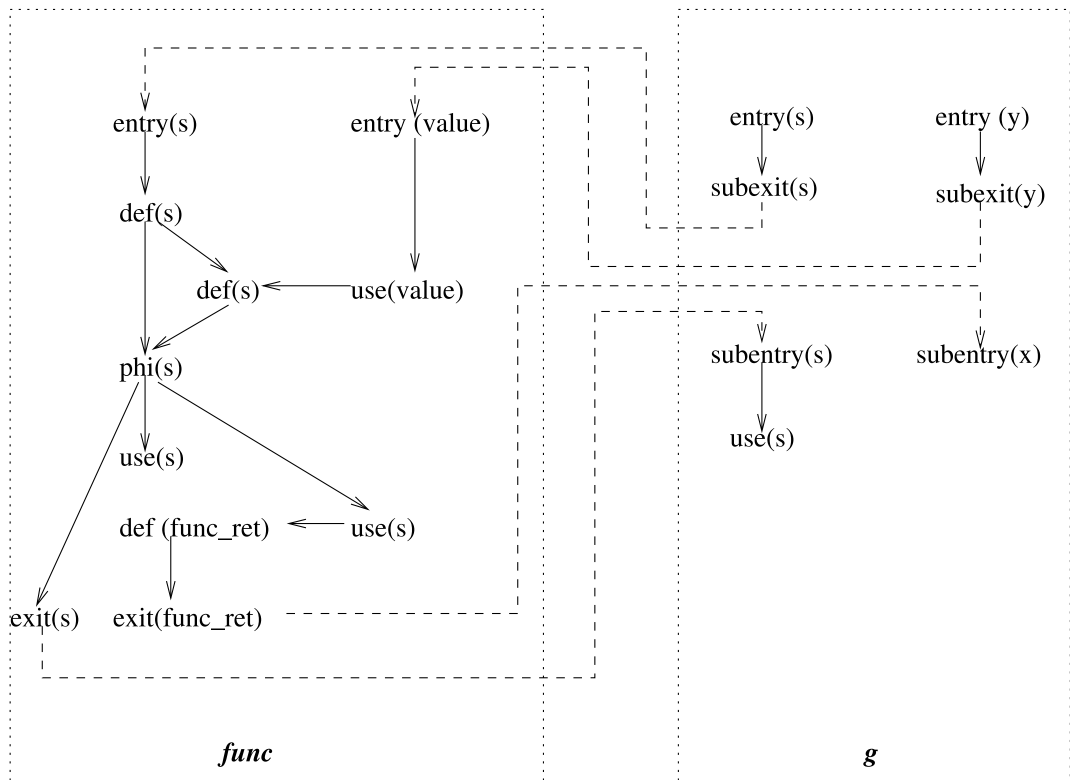


Abbildung 3.2: ISSA-Graph zum Codebeispiel

Eingang für *value* erhält für den in *g* gezeigten Aufruf seinen Wert vom dort befindlichen Unterausgang zum Argument *y*.

Für *s* symbolisiert der Eingang einen Zwischenschritt auf der interprozeduralen Suche nach der gültigen Definition, die hier aufgrund einer schwachen Aktualisierung nötig ist, um den Datenfluss konservativ zu modellieren. Komplementär dazu existiert am Aufruf ein Unterausgang zu *s*, der dort sozusagen die Suche (entgegen der Kontrollflussrichtung) fortsetzt und mangels lokaler Definition auch in *g* einen Eingang hervorruft.

Umgekehrt gibt es zum May-Def-Seiteneffekt auf *s* in *func*, d.h. zum zugehörigen Ausgang, einen Untereingang in *g*. Dieser fungiert dort als lokale Definition der Verwendung. Sehr ähnlich dazu findet auch der Ausgang für die Funktionsrückgabe (hier durch das Objekt *func_ret* symbolisiert) einen Untereingang als Partner. Da wir bei der IR angenommen haben, dass die

Rückgabe erst an eine neue Hilfsvariable zugewiesen wird (hier nicht gezeigt), gibt es zu jedem Objekt pro Aufrufstelle maximal einen Untereingang. Analog verhält es sich mit den Unterausgängen, wenn als Argumente neu eingeführte Hilfsvariablen zum Einsatz kommen. □

Ein Untereingang kann wie eine Definition eine starke oder schwache Aktualisierung sein, je nachdem, ob es sich um einen zwingenden oder nur einen möglichen Seiteneffekt handelt. Als schwache Aktualisierung besitzt er ebenfalls eine eingehende Kante von der lokal vorausgehenden Definition. Wir werden in Abschnitt 3.3.2 näher darauf eingehen, wie wir mit Untereingängen konkret verfahren.

Besitzt eine Funktion mehrere Ausgänge im Quellcode (explizite *return*-Anweisung), so bilden wir diese als Übergänge in den einzigen CFG-Endknoten ab. Gibt die Funktion einen Wert zurück, so bilden die Definitionen der Rückgabe (als spezielles Objekt, wie in Pascal) die letzten Elemente der Grundblöcke, die zum CFG-Endknoten führen. Bei mehreren Vorgängern des Endknotens entsteht für die SSA-Form dabei automatisch ein ϕ-Knoten als abschließende Definition der Rückgabe. So bezieht sich auch der Ausgang für die Rückgabe stets auf genau eine Definition. Analog kann man allgemeine ausgehende Parameter behandeln.

Technisch ist es nicht notwendig, tatsächlich Kanten zwischen Ausgängen und Untereingängen bzw. Unterausgängen und Eingängen anzulegen, wenn der Aufrufgraph und eine Abbildung von Instruktion und Objekt auf ISSA-Knoten mitgeführt wird. Ebenso haben die Knoten für Parameter und Argumente bereits Entsprechungen in der IR, so dass lediglich für Seiteneffekte etwas ergänzt werden muss.

Mit Hilfe der durch ISSA hinzugefügten Knoten können wir uns für die meisten Fälle wieder auf intraprozeduralen Datenfluss konzentrieren, da die Knoten interprozedurale Effekte (Definitionen und Verwendungen) lokal modellieren.

Oft kommt es uns nicht darauf an, welche Art von Knoten sich wie eine Definition verhält. Zur Vereinfachung fassen wir daher zusammen:

Definition 3.3.3 Als *allgemeine Definitionen* bezeichnen wir Definitionen, ϕ-Knoten, Eingänge und Untereingänge.

Als *allgemeine Verwendungen* bezeichnen wir Verwendungen, ϕ-Knoten, Ausgänge und Unterausgänge.

3.3.2 Zwingende und potentielle Seiteneffekte

Nicht nur für Definitionen stellt sich die Frage, ob sie stark oder schwach sind. Auch für induzierte Seiteneffekte müssen wir diese Frage beantworten. Ein schwacher Seiteneffekt ist dabei besser bekannt als potentieller Seiteneffekt (may-def) und jeder zugehörige Untereingang sollte nach dem bislang beschriebenen Modell eine eingehende Datenfluss-Kante erhalten. Mit einem Trick können wir dies jedoch vermeiden.

Untereingänge an direkten Aufrufen behandeln wir stets als starke Aktualisierungen: Es gibt keine Kante von einer vorausgehenden Definition zum Untereingang. Damit erscheint jeder schreibende Seiteneffekt auf den ersten Blick als zwingender Seiteneffekt (*must-def*). Handelt es sich jedoch lediglich um einen potentiellen Seiteneffekt (*may-def*), so hat dies zur Folge, dass ein Datenflusspfad quer durch das Unterprogramm existiert. Dafür sorgt unsere Modellierung von schwachen Aktualisierungen (falls eine solche der Auslöser für die Unsicherheit ist) sowie die Existenz von ϕ-Knoten (falls eine Kontrollfluss-Unsicherheit der Auslöser ist). Der so vorhandene Pfad quer durch das Unterprogramm ist dann von der lokal dem Untereingang vorausgehenden Definition erreichbar, verbindet diese also indirekt mit dem Untereingang. Das genügt zur korrekten Propagierung von Zeigerzielen. In Beispiel 3.3.2 gibt es zum potentiellen Seiteneffekt auf s einen Pfad vom $entry(s)$ zum $exit(s)$ in $func$. An der Aufrufstelle ist der Untereingang zu s ohne eingehende Kante, aber über den Pfad erreichen wir dennoch eine Verbindung von der lokal vorausgehenden Definition dorthin.

An indirekten Aufrufen ist jedoch Vorsicht geboten. Die Kontrollfluss-Unsicherheit über das tatsächliche Aufrufziel erlaubt einen zwingenden Seiteneffekt nur dann, wenn es in jedem möglichen Aufrufziel einen solchen (für das gleiche Objekt) gibt. Wir sehen die folgenden Möglichkeiten:

1. Generell an indirekten Aufrufen alle Seiteneffekte als schwache Aktualisierungen einstufen.

2. Bei nur einem Aufrufziel wie für einen direkten Aufruf verfahren, bei mehr als einem Ziel jedoch generell von nur potentiellen Seiteneffekten ausgehen.

3. Explizit pro Objekt überprüfen, ob jedes Aufrufziel zu einem zwingenden Seiteneffekt führt, und abhängig davon eine starke oder schwache Aktualisierung anlegen.

Diese Optionen sind nach Genauigkeit aufsteigend sortiert. Die zweite Option lässt sich praktisch mit den Kosten der ersten Option realisieren, da nur die Frage beantwortet werden muss, ob mehr als ein Aufrufziel vorliegt. Jedoch ist Vorsicht geboten, da die Analyse im Laufe der Iterationen weitere Zeigerziele erkennt und der Status eines Seiteneffekts damit möglicherweise von zwingend auf potentiell herabgestuft werden muss. Außerdem haben wir damit noch keine Strategie, den zwingenden Seiteneffekt vollständig zu beherzigen: Noch bevor ein Aufrufziel bekannt wird, sucht die Analyse vielleicht bereits aufgrund einer folgenden Verwendung an der Aufrufstelle nach einer gültigen Definition. Lassen wir sie weitersuchen und damit ggf. eine lokale Definition finden, so propagieren über diese Datenfluss-Beziehung eventuell bereits Zeigerziele, bevor wir überhaupt den starken Seiteneffekt erkennen. Eine Lösung hierfür, die auch mit der nachträglichen Herabstufung von zwingend auf potentiell umgehen kann, präsentiert später das Kapitel 8.

Die dritte Option schließlich erfordert neben der für Option 2 diskutierten Problematik weiteren Aufwand, um den Seiteneffekt einstufen zu können. Die Kandidaten lassen sich hierbei eingrenzen auf solche, zu denen überhaupt jedes Aufrufziel einen Ausgang besitzt. Für diese Kandidaten müssten wir nun aber pro Aufrufziel beurteilen, ob der Ausgang allein auf starke Aktualisierungen zurückgeht oder ob es einen Pfad vom Eingang zum Ausgang gibt, der den Seiteneffekt als nur potentiell klassifiziert. Diesen Aufwand ersparen wir uns und verbleiben stattdessen bei der ersten Option.

3.3.3 Dominanz und ϕ-Knoten

Zunächst ist bekannt, dass in der SSA-Form eine allgemeine Definition ihre Verwendungen dominiert. Nun betrachten wir jedes Argument eines ϕ-Knotens als Verwendung am Ende des zugehörigen, vorausgehenden Grundblockes. Umgekehrt besitzt jede allgemeine Verwendung genau eine Definition. Für einen ϕ-Knoten hat jedes Argument eine Definition. Bezogen auf den ϕ-Knoten selbst gilt die Dominanz nicht mehr unbedingt, denn der ϕ-Knoten wurde ja gerade in der Dominanzgrenze zumindest einiger dieser Definitionen angelegt. Es kann, muss aber nicht, der Fall eintreten, dass die Definition eines Arguments den ϕ-Knoten dominiert.

Es gilt jedoch, dass von den Definitionen p_i der Argumente eines ϕ-Knotens v maximal eine den Knoten v dominiert, und falls sie existiert, so dominiert sie die übrigen Definitionen p_i. Wir wollen die dominierende Definition hier mit d bezeichnen und mit einem kleinen Widerspruchsbeweis diese Aussage untermauern. Dazu betrachten wir die Suche nach der gültigen Definition für ein Argument von v: Diese Suche klettert den Dominanzbaum nach oben und stoppt bei der ersten passenden Definition. Sollte diese auch v dominieren, so benutzen wir sie als d. Das gleiche Vorgehen benutzen wir anschließend für die übrigen Argumente. Erhalten wir dort ein zweites $d' \neq d$ mit d' dom v, so gilt entweder d dom d' oder d' dom d. Wegen Symmetrie können wir ohne Einschränkung d dom d' annehmen. Daraus folgt, dass d' kein Dominator des ersten betrachteten Arguments ist, sonst hätte dessen Suche nach einer gültigen Definition bereits dort gestoppt. Jedoch ist jeder Dominator von v – also auch d' – ein Dominator aller direkten Vorgänger von v, sonst würde er nicht v dominieren. Das führt zum Widerspruch, so dass es maximal eine solche Definition d gibt, die v dominiert. Da die Suche aller Argumente nach einer gültigen Definition spätestens an d endet und bis dahin nur von d dominierte Knoten betrachtet werden, dominiert d die Definitionen zu allen Argumenten.

Definitionen der Argumente, die den ϕ-Knoten nicht dominieren, können immer noch umgekehrt vom ϕ-Knoten dominiert werden. Bei der einzigen dominierenden Definition liegt der umgekehrte Fall vor, da dann der ϕ-Knoten

kein Dominator sein kann.

Diese Überlegungen sind für die Algorithmen nützlich, die wir später für den Aufbau des ISSA-Graphen besprechen werden.

3.3.4 Objektgraphen

Für die Beschreibung der Analyse (und ebenso als Implementierungsidee) ist es nützlich, den ISSA-Graphen ein wenig zu strukturieren. Wir nutzen dabei nur eine einfache Partitionierung des Graphen in Partitionen, die jeweils die Knoten eines einzelnen Objekts umfassen, sowie die zwischen diesen Knoten verlaufenden Kanten. Einen solchen Teilgraphen nennen wir den *Objektgraphen* zum jeweiligen Objekt. Umgekehrt können wir den ISSA-Graphen dann als Zusammensetzung von Objektgraphen plus Kanten zwischen den Objektgraphen betrachten.

Weiterhin können wir den Objektgraphen eines Objekts noch unterteilen in die jeweiligen *lokalen* Objektgraphen. Ein lokaler Objektgraph (zum Objekt o und dem Unterprogramm f) besteht dabei aus allen Knoten des Objektgraphen von o, die im Unterprogramm f lokalisiert sind, sowie den (intraprozeduralen) Kanten zwischen diesen Knoten. Der Objektgraph zu o besteht damit aus den lokalen Objektgraphen von o zu allen Unterprogrammen sowie den interprozeduralen Kanten zwischen diesen.

Mit dieser Sichtweise können wir die Kanten eines ISSA-Graphen wie folgt klassifizieren:

- Intraprozedurale Intra-Objekt-Kanten: Dies sind die Kanten innerhalb eines lokalen Objektgraphen.
- Intraprozedurale Inter-Objekt-Kanten: Dies sind die übrigen intraprozeduralen Kanten. Sie entstehen durch Zuweisungsbrücken.
- Interprozedurale Intra-Objekt-Kanten: Dies sind die Kanten der Objektgraphen, welche die jeweiligen lokalen Objektgraphen verbinden.
- Interprozedurale Inter-Objekt-Kanten: Dies sind die übrigen interprozeduralen Kanten. Sie entstehen durch Parameterübergabe und Funktionsrückgabe.

Beispiel 3.3.4 Im Beispiel 3.3.2 besteht der dort gezeigte ISSA-Graphausschnitt aus den Objektgraphen zu s, $value$, x, y und $func_ret$ sowie den Kanten zwischen diesen Objektgraphen. Dies sind dabei die Zuweisungbrücke im *if* und für die Rückgabe als intraprozedurale Inter-Objekt-Kanten, sowie die Argumentübergabe und Resultatsübernahme beim Aufruf als interprozedurale Inter-Objekt-Kanten. Abbildung 3.3 zeigt speziell den Objektgraphen für s. Dieser besteht aus den zwei lokalen Objektgraphen für $func$ und g zu diesem Objekt. Die durchgezogenen Kanten sind dabei genau die intraprozeduralen Intra-Objekt-Kanten, während die gestrichelten Kanten genau die interprozeduralen Intra-Objekt-Kanten darstellen. □

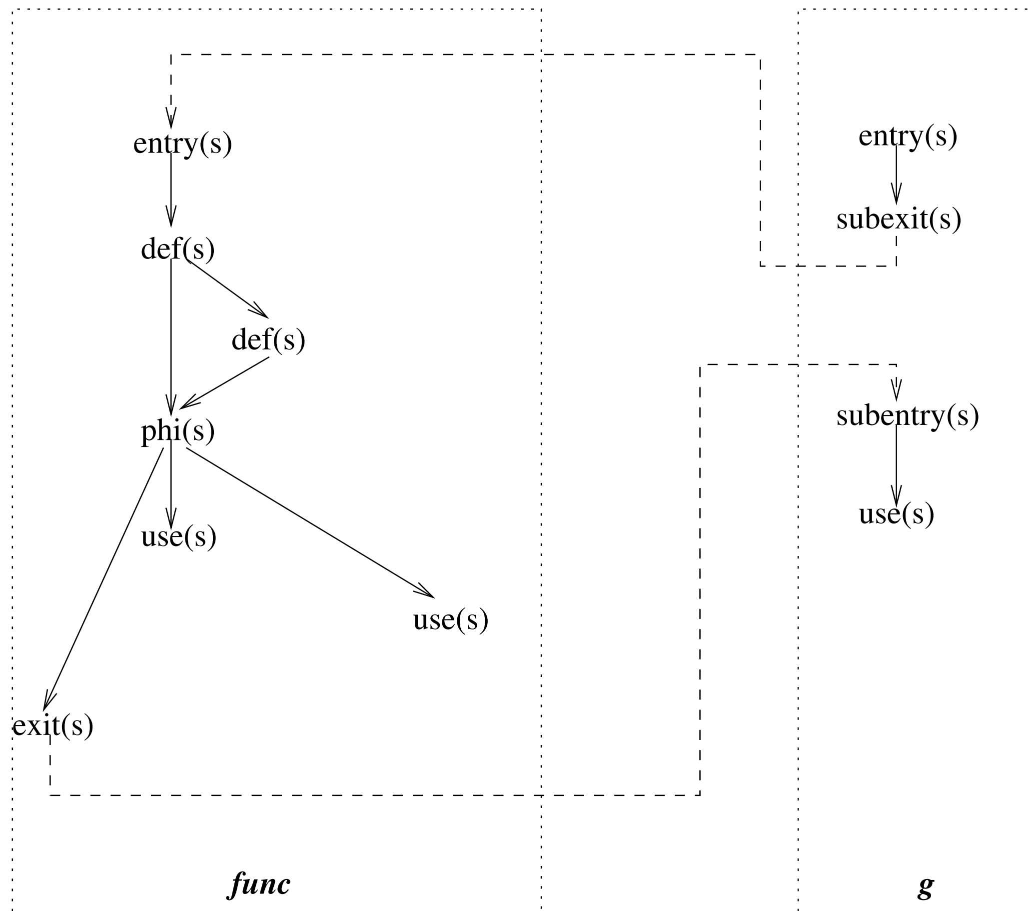


Abbildung 3.3: Objektgraph von s im Beispiel

Kapitel 4

Kombinierte Zeiger-, Kontroll- und Datenfluss-Analyse

Dieses Kapitel beschreibt die wesentliche Idee unserer kombinierten Analyse. Die Beschreibung ist absichtlich abstrakt gehalten: Der Fokus des Kapitels liegt auf der *Idee* der Kombination im Unterschied zu den bisher bekannten Verfahren. Die darauf folgenden Kapitel werden anschließend über verschiedene mögliche Konkretisierungen dieser Idee sprechen. Das Kapitel betrachtet neben dem Unterschied zu anderen Verfahren auch die Verwandtschaft zu Andersens Zeigeranalyse.

4.1 Abstrakte Formulierung der Kernanalyse

Zur Lösung der drei zentralen Aufgaben Kontrollfluss, Datenfluss und Zeigerziele gehen viele Publikationen den Weg, zunächst mit einer Zeigeranalyse eine konservative Überschätzung der Zeigerziele an Dereferenzierungen zu ermitteln. Anschließend konstruieren sie dann eine Datenfluss-Darstellung. Von diesem sequentiellen Ablauf weicht die kombinierte Analyse ab. Ihre Grundidee ist es, die beiden Aufgaben der Zeigeranalyse und Datenfluss-Ermittlung miteinander zu kombinieren. Dabei gelingt dann auch eine einfache Integration der Kontrollfluss-Analyse.

Die Kombination betrachtet die Zeigeranalyse selbst als eine weiterführende Datenfluss-Analyse. Tatsächlich können wir die Zeigerziele an Dereferenzierungen leicht bestimmen, wenn uns bereits eine konservative Datenfluss-Darstellung zur Verfügung steht. Nun drehen wir aber nicht einfach die Reihenfolge der klassischen Sequenz um; denn dann stünden wir vor dem Problem, wie die konservative Datenfluss-Darstellung bestimmt werden soll. Stattdessen besteht die wichtige Beobachtung hinter der Kombinationsidee darin, dass wir die Zeigeranalyse auch bereits auf einer *noch nicht konservativen* Datenfluss-Darstellung ausführen können. Dann haben wir zwar noch keine finale Lösung für die Aufgaben, sind dieser Lösung jedoch einen Schritt näher gekommen. Wie wir im Abschnitt 10.1 noch zeigen werden, genügt nun eine normale Fixpunkt-Iteration über diese simple Strategie, um schließlich konservative Resultate zu erhalten.

```
Compute direct ICFG and IDFG;
while not fixpoint reached loop
   Determine data-flow relations; -- step 1
   Propagate pointer targets;     -- step 2
end loop;
Finalize;
```

Abbildung 4.1: Allgemeiner Ablauf der Analyse

Abbildung 4.1 zeigt eine allgemeine Darstellung dieser neuen Analyse[1]. Grundlage für die Analyse ist dabei eine Zwischendarstellung (engl. *intermediate representation*, IR) für das zu analysierende Programm (vgl. Kapitel 3).

Die Analyse beginnt nun damit, intraprozedurale Kontrollfluss-Graphen (engl. *control-flow graph*, CFG; Allen [1970]; Hecht [1977]) mit aus dem Übersetzerbau bekannten Mitteln zu konstruieren. Verbindet man dabei für die direkten Aufrufstellen jeweils den Aufruf mit dem Eingangsknoten des CFG aus der Zielfunktion sowie den Ausgang des CFG mit dem Rücksprungspunkt

[1] Wir verwenden durchgehend englischsprachigen Pseudo-Code in einer Ada-ähnlichen Syntax

an der Aufrufstelle, so erhält man den interprozeduralen Kontrollfluss-Graphen (engl. *interprocedural control-flow graph*, ICFG; [Landi 1992, S. 7]). Diese Verfahren sind hinreichend bekannt und sollen hier nicht näher erläutert werden. Der (direkt ermittelbare Anteil des) ICFG ist bereits ein großer Teil der gewünschten Kontrollfluss-Darstellung. Zur Vervollständigung fehlen die interprozeduralen Verbindungen an indirekten Aufrufen sowie z.B. die Pfade, die sich aufgrund von Ausnahmen ergeben. Der Aufrufgraph als eigene Datenstruktur lässt sich dann bei Bedarf aus dem ICFG ablesen.

Da wir die Dominanz-bezogenen Eigenschaften der (I)SSA-Form nutzen wollen, berechnen wir zu jedem CFG auch den Dominanzbaum und die Dominanzgrenzen. Unsere Erfahrungen zeigen, dass alle diese Berechnungen selbst für große Programme sehr schnell ablaufen, und die Ergebnisse benötigen typischerweise nur wenig Speicherplatz.

Bemerkung 4.1.1 Tatsächlich benötigt die Analyse anschließend nur noch die Dominanzinformationen (sowie Aufrufbeziehungen), nicht den CFG selbst. Lässt sich die Dominanzinformation z.B. bei strukturiertem Code auch direkter aus dem AST bestimmen, so taucht der CFG in unserer fluss-sensitiven Analyse sogar überhaupt nicht auf.

Es folgt die analoge Bestimmung des interprozeduralen Datenfluss-Graphen (engl. *interprocedural data-flow graph*, IDFG). Der direkt aus der IR ermittelbare Anteil dieses Graphen umfasst Knoten für Definitionen und Verwendungen sowie Zuweisungsbrücken als Kanten[2]. Wir legen dabei auch künstliche Definitionen am Eingangsknoten einer Funktion für ihre Parameter an und modellieren die Parameterübergabe an direkten Aufrufen mit Zuweisungsbrücken zu diesen Definitionen. Entsprechend erfolgt die Modellierung einer Funktionsrückgabe (vgl. Beschreibung der IR und des ISSA-Graphen in Kapitel 3).

Das Herz der Analyse ist die Fixpunkt-Iteration. Diese startet mit dem soeben bestimmten initialen ICFG und IDFG und ergänzt darin die fehlenden

[2] Wir beschreiben hier zunächst eine konzeptionelle Sicht. Welche Knoten und Kanten eine konkrete Umsetzung tatsächlich explizit darstellt und welche implizit in der IR verankert bleiben, ist ein anderes Thema.

Knoten und Kanten. Jede einzelne Iteration umfasst die folgenden beiden Schritte:

1. Bestimme den Datenfluss gemäß der aktuell bekannten Zeigerziele.

2. Propagiere Zeigerziele auf dem aktuell bekannten Datenfluss-Graphen.

Dabei arbeiten beide Schritte jeweils auf den bis dahin bekannten Resultaten und ignorieren die Tatsache, dass weitere Resultate hinzukommen können und die alten Resultate teilweise revidieren. So kümmert sich Schritt 1 (der *Graphaufbau-Schritt*) nur um schon bekannte Datenfluss-Elemente und ermittelt die Beziehungen zwischen diesen. Dabei wird der CFG genutzt, um den Datenfluss von Definitionen zu Verwendungen zu erkennen. Im Falle des in dieser Arbeit genutzten ISSA-Graphen als IDFG ist hierzu im Wesentlichen der Dominanzbaum des CFGs notwendig, wie wir bei den Algorithmen noch sehen werden. Die für die ISSA-Form notwendigen induzierten Knoten (ϕ-Knoten und Seiteneffekt-Knoten) legt der Graphaufbau-Schritt ebenfalls an. Dass noch unbekannte Zeigerziele weitere Datenfluss-Elemente aufdecken werden, beachtet der Schritt jedoch nicht.

Anschließend benutzt Schritt 2 (der *Propagierungs-Schritt*) ausschließlich die soeben bestimmten Datenfluss-Beziehungen und nur bereits erkennbare Zeigerliterale und Dereferenzierungen. Gemäß den Datenfluss-Beziehungen propagiert dieser Schritt Zeigerziele von deren Quellen zu den Dereferenzierungen und indirekten Aufrufen. Dabei entstehen neue Knoten für indirekte Definitionen und Verwendungen sowie neue Aufrufziele. Feld-sensitiv entstehen an Operationen wie `&(p->f)` auch neue Zeigerliterale und Quellen für die Propagierung. Diese neuen Elemente werden jeweils zwischengespeichert und dann in der nächsten Iteration im Rahmen des folgenden Graphaufbau-Schrittes integriert. Auf diese Weise ändert sich der Graph, über den propagiert wird, nicht während der Propagierung, was technisch vorteilhaft ist.

Beide Schritte erweitern die alten Zwischenresultate lediglich, es wird nichts wieder gelöscht. Insbesondere behandelt die hier zunächst besprochene Variante der Analyse alle indirekten Definitionen als schwache Aktualisierungen. Im Falle des ISSA-Graphen werden so aus bisherigen Kanten Pfade,

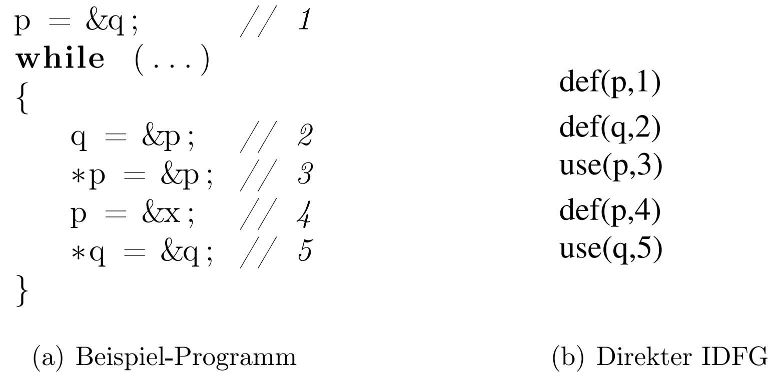
wenn sich eine indirekte Definition zwischen die Enden einer Kante schiebt. ICFG und IDFG wachsen damit monoton bis zum Fixpunkt.

Wir werden auch eine Erweiterung der Analyse besprechen, welche indirekte starke Aktualisierungen unterstützt und damit die Präzision erhöht. Die wesentliche Idee hierzu besteht darin, das Anlegen von Datenfluss-Kanten (und damit auch deren Nutzung für die Propagierung von Zeigerzielen) zu verzögern. Damit wird es möglich, den Kill-Effekt einer erst später erkannten indirekten starken Aktualisierung umzusetzen: Bis zum Bekanntwerden der indirekten starken Aktualisierung sind dann noch keine Zeigerziele einer ihr vorausgehenden anderen allgemeinen Definition an ihr vorbei propagiert worden. Diese Strategie zur Beherrschung der indirekten starken Aktualisierungen erfordert innerhalb der Fixpunkt-Iteration nur im Graphaufbau Veränderungen, um das Verzögern der Kanten-Erstellung zu realisieren. Wir werden dies in einem separaten Kapitel im Detail besprechen; bis dahin steht die Variante ohne diese Unterstützung im Vordergrund.

Eine optionale Nachbearbeitung schneidet unnütze Pfade des Datenfluss-Graphen ab (engl. *pruning*), d.h. sie löscht solche Knoten, deren Wert nirgendwo verwendet wird. Sie kann auch Vorbereitungen treffen für nachfolgend ausgeführte Zielanalysen, beispielsweise die Berechnung von starken Zusammenhangskomponenten und einer topologischen Ordnung für die ermittelten Graphen.

4.2 Beispiel

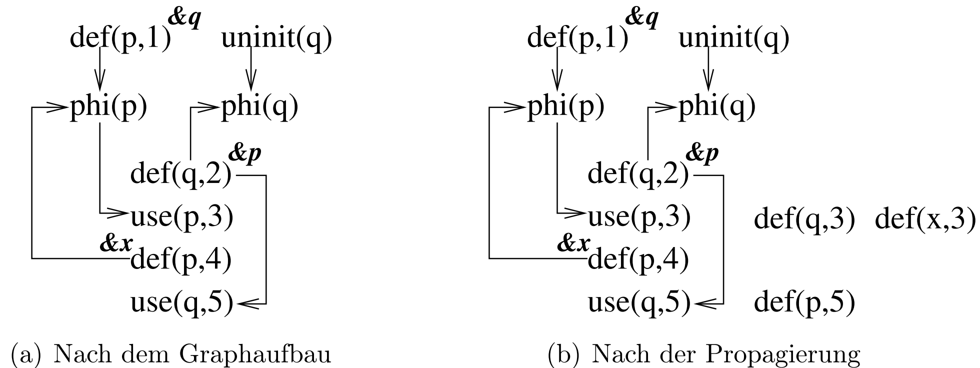
In diesem Abschnitt wollen wir das Vorgehen der Analyse an einem kleinen Beispiel Schritt für Schritt illustrieren. Wir benutzen dazu den in Abbildung 4.2 gezeigten Programmausschnitt. Zunächst wird hierzu die IR mitsamt dem CFG (bzw. dessen Dominanzbaum) bestimmt, was hier nicht weiter schwer ist. Der erste Graphaufbau-Schritt operiert auf dem direkt ermittelbaren IDFG, wobei wir wie beschrieben nur dessen Knoten vor der Iteration bestimmen. Dieser direkte IDFG ist ebenfalls in Abbildung 4.2 gezeigt. Hierbei kürzen wir Definitionen mit *def* und Verwendungen mit *use* ab und geben in Klammern das jeweils betroffene Objekt sowie die Instruktion an. Da das



(a) Beispiel-Programm (b) Direkter IDFG

Abbildung 4.2: Beispielprogramm und initiale Knoten

Auftreten eines Objekts als Operand der Adressnahme keine Verwendung darstellt (der Wert wird ja nicht gelesen), ist diese Angabe auch für die Instruktionen 3 und 5 eindeutig und bezieht sich dort jeweils auf die LHS.



(a) Nach dem Graphaufbau (b) Nach der Propagierung

Abbildung 4.3: Zwischenergebnisse der 1. Iteration

Abbildung 4.3 präsentiert nun die Zwischenergebnisse jeweils nach dem ersten Graphaufbau und nach der ersten Propagierung. Die Kanten geben die SSA-Kanten wieder, wobei wir sie hier in Datenfluss-Richtung orientiert haben. Wir erkennen auch die beiden ϕ-Knoten, welche der Graphaufbau für die Definitionen innerhalb der Schleife ergänzt hat. Die ϕ-Knoten verlangen hierbei auch nach einer Definition vor der Schleife, weswegen für q die künstliche Definition am Anfang des Gültigkeitsbereichs (als Signal für

nicht initialisiert) auftritt (die irrelevanten Knoten dieser Art haben wir der Übersichtlichkeit halber nicht aufgeführt, auch wenn die IR sie von Anfang an bereit hält). Zum besseren Verständnis der Propagierung haben wir die Zeigerziel-Quellen nach dem Graphaufbau-Schritt mit den dort jeweils relevanten Zeigerliteralen annotiert. Diese werden dann entlang der von den gezeigten Kanten gebildeten Pfade propagiert, so dass die Analyse am $use(p,3)$ die beiden Ziele q und x erkennt, sowie am $use(q,5)$ das Ziel p. Entsprechend führt dies zu drei neuen (indirekten) Definitionen, mit denen die folgende Iteration startet. Diese sind in der Abbildung rechts zu sehen.

Abbildung 4.4 illustriert den Verlauf dieser zweiten Iteration. Der Graphaufbau legt den ϕ-Knoten zu $def(x,3)$ sowie die zugehörigen Kanten an, wiederum mit Bezug auf die künstliche Definition am Beginn. Die neuen indirekten Definitionen betrachten wir wie beschrieben alle als schwache Aktualisierungen, was über die eingehenden Kanten an diesen Knoten dargestellt ist. Für p und q wurden die bisherigen Kanten innerhalb der Schleife durch Pfade über die neuen schwachen Aktualisierungen ersetzt, die sich zwischen die bisherigen Enden einer Kante geschoben haben. Nach dem Graphaufbau folgt wiederum die Propagierung. Verglichen mit der Propagierung der ersten Iteration hat sich die Zahl der Zeigerziel-Quellen erhöht, aber an den Verwendungen in Dereferenzierungen kommen keine neuen Ziele an. Die Fixpunkt-Iteration endet damit, und der berechnete ISSA-Graph sieht nach der zweiten Iteration aus wie in Abbildung 4.4 gezeigt.

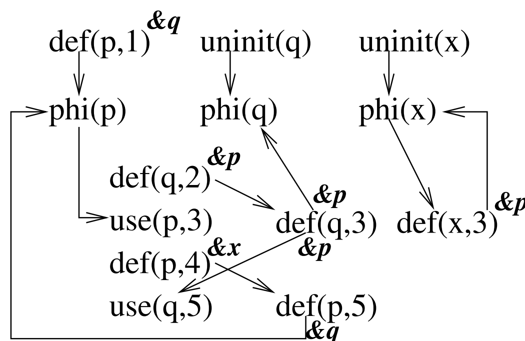


Abbildung 4.4: Zustand nach dem Graphaufbau der 2. Iteration. Zugleich auch Resultat vor dem Pruning (nach der 2. Iteration)

Abschließend kann nun ein Pruning den berechneten ISSA-Graphen bereinigen. So kann der ϕ-Knoten zu q (und dann auch die künstliche Definition am Beginn) offensichtlich entfallen, da er keine ausgehende Kante besitzt. Der ϕ-Knoten zu x verfügt zwar über eine ausgehende Kante, diese führt jedoch nicht zu einem Teilgraphen mit einer Verwendung. Gleiches gilt für die indirekte Definition auf x. Somit können sogar alle Knoten zu x gelöscht werden. Das Resultat dieser Bereinigung zeigt uns schließlich Abbildung 4.5. Das Pruning betrifft jedoch nur die Knoten und Kanten der Datenfluss-Darstellung; die Zeigerziel-Mengen als weitere Resultate an den einzelnen Dereferenzierungen bleiben unverändert (in der Abbildung nicht gezeigt), so dass z.B. x weiterhin als Zeigerziel in Zeile 3 erkennbar ist.

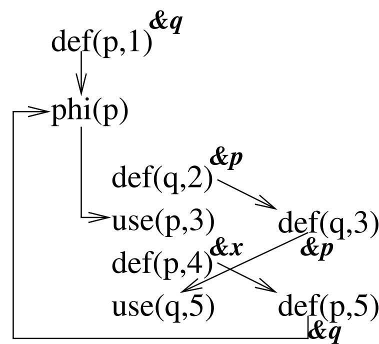


Abbildung 4.5: Resultat nach dem Pruning

4.3 Grobes algorithmisches Vorgehen

Nach der abstrakten Formulierung wollen wir nun kurz einen ersten Eindruck vermitteln, wie die Analyse algorithmisch umgesetzt werden kann. Der hier gegebene grobe Überblick dient zum besseren Verständnis der Details, die in den folgenden Kapiteln präsentiert werden. Außerdem können wir dann bereits in diesem Kapitel die fluss-sensitive Ausprägung mit Andersens fluss-insensitiver Analyse vergleichen.

4.3.1 Graphaufbau

Für eine kompakte Beschreibung der Aktionen im Graphaufbau-Schritt unterteilen wir den ISSA-Graphen in lokale *Objektgraphen* (vgl. Kapitel 3.3.4). Ein lokaler Objektgraph soll dabei stets (nur) alle Knoten zu einem bestimmten Objekt in einem bestimmten Unterprogramm enthalten, sowie die Kanten zwischen diesen Knoten. Der ISSA-Graph enthält dann neben den diversen lokalen Objektgraphen die intraprozeduralen Kanten zwischen Objektgraphen verschiedener Objekte im gleichen Unterprogramm, sowie die interprozeduralen Kanten zwischen lokalen Objektgraphen aus sich aufrufenden Unterprogrammen. In unserem Beispiel aus Abschnitt 4.2 enthält der ISSA-Graph vor dem Pruning demnach die Objektgraphen zu p, q und x, sowie einige „Interobjektgraphen-Kanten“.

Mit dieser Sichtweise können wir den Graphaufbau grob so beschreiben, dass er pro neu hinzugekommenem ISSA-Knoten n die folgenden Aktionen ausführen muss:

1. Erweitere den Objektgraphen zu n sowie ggf. die Objektgraphen in Aufrufern um induzierte Knoten (ϕ-Knoten und Seiteneffekt-Knoten).

2. Aktualisiere die Kanten des Objektgraphen zu n.

3. Aktualisiere die Kanten zwischen dem Objektgraphen von n und angrenzenden Objektgraphen.

Die im ersten Schritt hinzugefügten Knoten werden dann noch im gleichen Graphaufbau-Schritt ebenfalls wie neue Knoten behandelt, durchlaufen also ihrerseits ebenso die genannten Aktionen.

Zentrale Funktionen zur Umsetzung dieser Schritte werden dabei die Aufgaben erfüllen, zu einem Objekt an einer Instruktion *die* gültige (dominierende) Definition zu finden, sowie die Kanten anzupassen, zwischen deren Enden sich nun neue Knoten eingeschoben haben.

4.3.2 Propagierung

Der Propagierungs-Schritt löst ein Grapherreichbarkeits-Problem auf dem ISSA-Graphen: Er muss ermitteln, zwischen welchen Zeigerziel-Quellen und Dereferenzierungen es Pfade in diesem Graphen gibt. Das propagiert die jeweiligen Zeigerziele, um an Dereferenzierungen neue Definitionen, Verwendungen, Zeigerliterale oder Aufrufziele zu entdecken. Die Ermittlung der Verbindungen kann algorithmisch sowohl vorwärts (propagiere Quellen zu Dereferenzierungen) als auch rückwärts (propagiere Dereferenzierungen) erfolgen. Der Kernalgorithmus ist eine simple Tiefensuche. Da diese Aufgabe in jeder Iteration erneut anfällt, untersuchen wir zur Beschleunigung auch eine inkrementelle Variante, welche nur nach neuen Verbindungen im Vergleich zur vorigen Iteration sucht.

Der ISSA-Graph ist zumindest die konzeptionelle Grundlage. Tatsächlich können wir jedoch über diesem auch eine Hierarchie von Graphen durch Anwendung diverser Graphvereinfachungen konstruieren, um so die Eingabegröße für das Erreichbarkeitsproblem zu reduzieren. Die wohl bekannteste Vereinfachung ist die Kontraktion von Zyklen, so dass die Propagierung auf dem SCC-DAG des ISSA-Graphen stattfindet.

4.3.3 Finalisierung

Zum Abschluss bereinigt unsere kombinierte Analyse die Resultate und berechnet nützliche Zusammenfassungen für nachfolgende Analysen. Im Einzelnen erledigt die Finalisierung folgende Schritte:

1. Toten Code entfernen
2. Prunen des IDFG
3. Konstruktion des SCC-DAG zum ICFG, IDFG und Aufrufgraphen
4. Berechnung der topologischen Ordnung und der Postorder-Reihenfolge

Diese Aufgaben können unabhängig davon durchgeführt werden, wie der IDFG und die Ziele für indirekte Aufrufe berechnet wurden.

4.4 Vergleich zu Andersens Analyse

Wie selbstverständlich haben wir den abstrakt beschriebenen Ansatz bislang *fluss-sensitiv* ausgeprägt. Tatsächlich lässt er sich jedoch auch fluss-insensitiv ausprägen. In diesem Fall verwenden wir als IDFG nicht den ISSA-Graphen, sondern den bei Andersen so genannten *Constraint-Graphen*. Die Analyse-Objekte bilden die Knoten dieses Graphen. Die Kanten, welche mit Andersens Sichtweise Teilmengen-Beziehungen repräsentieren, können wir auch als grobe Überschätzung der Datenfluss-Beziehungen auffassen. Dafür genügt für deren Erkennung eine äußerst lokale Betrachtung: Die Beziehungen ergeben sich allein aus den Kopierzuweisungen inklusive der Parameterübergabe und Rückgabe. Im Unterschied zur fluss-sensitiven Variante ist kein CFG oder dessen Dominanzinformation als Basis notwendig.

Da auf diese Weise eine Überschätzung des IDFG im Vergleich zur fluss-sensitiven Realisierung stattfindet, können wir die Propagierung entlang der jeweiligen IDFG-Kanten als zweiten Schritt pro Iteration beibehalten. Wenn wir eine hinreichend abstrakte Bezeichnung für Knoten des IDFG annehmen (z.B. eine Nummer als Index in ein Array), so müssen wir lediglich den Graph-aufbau-Schritt austauschen. Dann erhalten wir eine alternative Formulierung für die bekannte Zeigeranalyse von Andersen [1994]. Abschnitt 7.3 wird hierzu die Algorithmen präsentieren.

Bemerkung 4.4.1 Vor dem Hintergrund dieser Betrachtung können wir den Beitrag der vorliegenden Dissertation auch griffig als *fluss-sensitive Andersen-Analyse* umschreiben. Unsere Ausprägung bringt jedoch mit sich, dass nicht nur Zeigerziele bestimmt werden, sondern dass deren Bestimmung inhärent mit dem Aufbau einer gewohnten Datenfluss-Darstellung verbunden ist (während dies in der fluss-insensitiven Ausprägung, falls gewünscht, erst nach Beendigung der Analyse erfolgt). Insofern greift die Titulierung als fluss-sensitive Andersen-Analyse mit ihrem Fokus auf Zeigerziele etwas zu kurz. Wir verwenden daher lieber die Bezeichnung als kombinierte Analyse.

Bei Andersens Analyse ist es interessant zu sehen, dass zunächst mit dem grob überschätzten Datenfluss gearbeitet wird; am Ende wird diese grobe

Überschätzung jedoch weggeworfen und – so man neben den Zeigerzielen eine Datenfluss-Darstellung mit Definitionen und Verwendungen wünscht – allein mit den berechneten Zeigerzielen eine präzisere Datenfluss-Darstellung hergeleitet [Staiger u. a. 2007]. Unsere kombinierte fluss-sensitive Analyse dagegen muss nichts berechnen, was hinterher wieder weggeworfen wird[3], und wir erhalten die höhere Präzision.

Die Verwandtschaft unserer Kernanalyse mit der wohl einflussreichsten Zeigeranalyse ist bemerkenswert. So hat Andersen selbst in seiner Dissertation bereits kurz betrachtet, ob sich sein Ansatz auch fluss-sensitiv realisieren lässt. Es ist ihm jedoch nicht gelungen, und er schloss diesen Abschnitt mit den Worten [Andersen 1994, S. 148]

> „Currently, we have no good solution to this problem.“

Die vorliegende Dissertation dagegen bietet eine fluss-sensitive Ausprägung. Damit erschließt sie die höhere Präzision mit dem gleichen einfachen Ansatz und der gleichen asymptotischen Komplexität (vgl. Kapitel 10). Wir hoffen zudem, dass sich umgekehrt auch die für Andersen bekannten Beschleunigungen auf unsere fluss-sensitive Realisierung übertragen lassen (für die Zyklenkontraktion demonstrieren wir dies bereits in der vorliegenden Arbeit). Diese betreffen fast ausschließlich den Propagierungs-Schritt, der für beide Verfahren das gleiche Problem löst.

4.5 Vergleich mit anderen Zeigeranalysen

Nach dem Vergleich mit Andersens Verfahren wollen wir in diesem Abschnitt unseren Ansatz mit weiteren bislang bekannten Verfahren vergleichen, und zwar mit:

- anderen fluss-sensitiven Zeigeranalysen;
- sowie anderen fluss-insensitiven Zeigeranalysen in Kombination mit einer Datenfluss-Berechnung.

[3] Pruning ist auch bei anderweitiger SSA-Konstruktion üblich

Mit diesem Vergleich zeigen wir, dass es sich um einen neuen Ansatz für fluss-sensitive Zeigeranalysen handelt. Abschließend diskutieren wir noch, warum Fluss-Sensitivität gegenüber Fluss-Insensitivität durchaus betrachtet werden sollte, auch wenn Publikationen ein negatives Licht darauf geworfen haben.

4.5.1 Fluss-sensitive Zeigeranalysen

Fluss-sensitive Zeigeranalysen betrachten die Aufgabe als klassisches Datenfluss-Problem. Insbesondere lösen sie die Aufgabe auf dem ICFG oder einer etwas komprimierteren Darstellung. Das führt zu mindestens drei größeren Problemen, die Hardekopf und Lin [2009b] als Herausforderungen aufzählen:

1. Als Datenfluss-Problem auf dem ICFG werden Zeigerziel-Mengen zu allen Kontrollfluss-Nachfolgern eines Knotens propagiert. Jedoch erhalten damit die CFG-Knoten viele unnötige Informationen. (Hardekopf und Lin betrachten nur eine Vorwärts-Propagierung.)

2. Die Transferfunktionen fordern teure Mengenoperationen. Außerdem müssen diese Funktionen häufig angewendet werden.

3. Das Datenfluss-Problem verlangt pro Grundblock die Speicherung der Fakten, z.B. als Out-Menge. Dadurch können die Speicheranforderungen einer FS-Analyse hoch sein.

Diese drei Probleme attackieren wir durch die Nutzung des ISSA-Graphen anstelle des ICFG als Grundlage. Dadurch

1. erhalten nur noch relevante (ISSA-)Knoten die jeweiligen Zeigerziele

2. können wir Zeigerziele punktweise anstatt als Menge propagieren

3. ist nur noch eine Menge der dafür relevanten Zeigerziele pro ISSA-Knoten nötig, die noch dazu nicht den Zeiger benennen muss, da dieser im Knoten gegeben ist.

Hardekopf selbst berichtet zwar ebenfalls von der Einsicht, dass die SSA-Form diese Vorteile besonders hervorhebt, kann diese jedoch nur für Variablen

einsetzen, welche nicht selbst Ziel eines Zeigers sind. Für seine neueste Arbeit [Hardekopf und Lin 2009a] hat er vorab eine FI-Zeigeranalyse eingesetzt, um die SSA-Form für alle Variablen zu ermöglichen.

Im Gegensatz dazu benötigen wir keine Vorabkonstruktion des IDFG oder eine vorab ausgeführte Zeigeranalyse. Stattdessen operieren wir auf dem tatsächlichen IDFG, der im Wechsel mit der Bestimmung der Zeigerziele errichtet wird. Der Korrektheitsbeweis in Abschnitt 10.1 ist die Grundlage für dieses Vorgehen: Er stellt sicher, dass trotz der Verwendung eines noch nicht konservativen IDFG die Resultate am Ende tatsächlich konservativ sind. Die Verwendung des IDFG vermeidet die unnötige Propagierung der ICFG-basierten Verfahren. Außerdem landen wir beim bekannten Problem der dynamischen transitiven Hülle für diesen Graphen und benötigen keine besonderen Transferfunktionen. Trotz seiner Einfachheit ist unser Ansatz damit eine zuvor noch nicht beschriebene Vorgehensweise für fluss-sensitive Zeigeranalysen mit Vorteilen gegenüber dem Stand der Dinge.

4.5.2 Vergleich mit weiteren Zeigeranalysen

Im Vergleich zu anderen Ansätzen beginnen wir mit der Datenfluss-Analyse mit optimistischen Zeigerzielen (nämlich initial gar keinen) und verfolgen sodann einen iterativen Wechsel mit der Zeigeranalyse, die ebenfalls nur schon bekannte Datenfluss-Beziehungen ausnutzt. Dies können wir in Diagrammen wie in Abbildung 4.6 mit anderen Ansätzen vergleichen. In dieser Abbildung sehen wir einen Vergleich der Kernanalyse mit dem typischen Vorgehen im Zusammenspiel mit einer FI-Zeigeranalyse, mit dem von Hasti und Horwitz skizzierten Vorgehen [Hasti und Horwitz 1998], sowie mit dem Karlsruher Ansatz, der den Speicher zuerst durch ein einziges Objekt abstrahiert [Lindenmaier u. a. 2005].

Im Vergleich zum klassischen Vorgehen mit fluss-insensitiven Zeigeranalysen sehen wir folgende Unterschiede:

1. Das klassische Vorgehen startet mit der Zeigeranalyse, nicht mit der Datenfluss-Analyse.

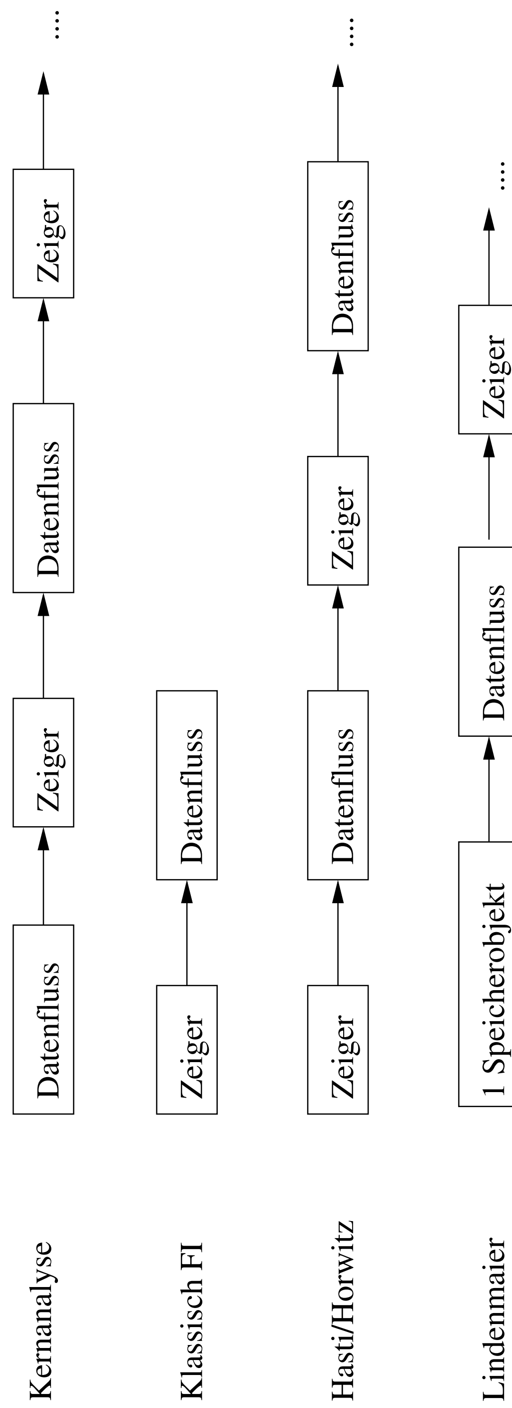


Abbildung 4.6: Schematischer Vergleich des Ansatzes zu anderen Vorgehen

2. Das klassische Vorgehen beendet die Zeigeranalyse mit konservativen Resultaten, bevor die Datenfluss-Analyse startet. Die Kernanalyse dagegen wechselt zwischen den Analysen mit jeweils noch nicht konservativen Resultaten.

3. Das klassische Vorgehen benötigt keine Fixpunkt-Iteration, sondern ist mit einmaliger Ausführung der beiden Analysen fertig.

Die Idee von Hasti und Horwitz, dieses klassische Vorgehen zu einer Fixpunkt-Iteration zu erweitern und dabei in jedem Schritt die Präzision zu erhöhen, unterscheidet sich von unserem Ansatz durch die Richtung, aus der das Resultat angenähert wird. Abbildung 4.7 illustriert dies: Sie zeigt schematisch die Größe der Zeigerzielmengen im Verlauf der Analyse. Die Kernanalyse beginnt mit leeren Mengen und vergrößert diese pro Iteration, bis am Ende konservative Resultate entstanden sind. Hasti und Horwitz dagegen starten mit einer fluss-insensitiven Überschätzung und versuchen, diese pro Iteration zu verbessern.

Nach eigener Auskunft arbeitet Prof. Seidl ähnlich wie wir vom Optimistischen her, jedoch im Rahmen der abstrakten Interpretation.

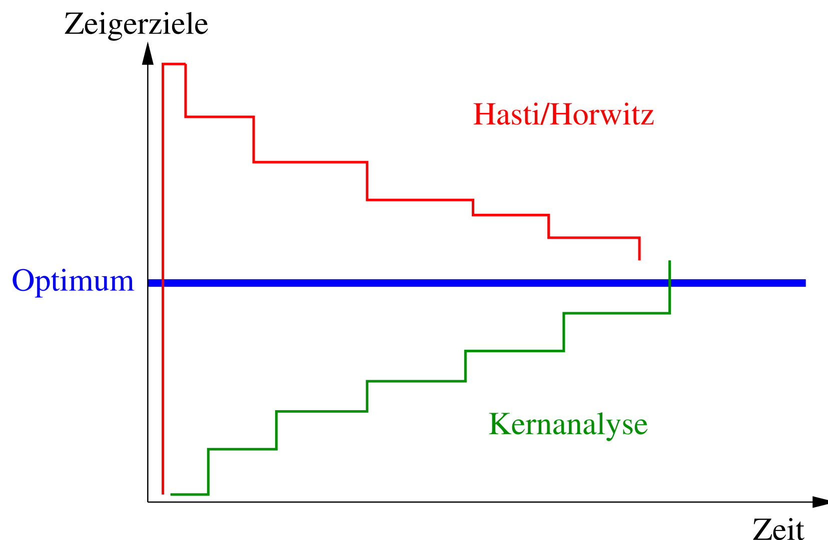


Abbildung 4.7: Schematischer Vergleich zum Ansatz von Hasti und Horwitz

4.5.3 Fluss-Sensitivität versus -Insensitivität

In der Literatur vorherrschend sind fluss-insensitive Verfahren, da Fluss-Sensitivität als nicht skalierbar für große Programme angesehen wird [Ryder 2003, Kapitel 3]. Eine Ursache für dieses Bild waren die Artikel von Hind und Pioli [Hind 2001; Hind und Pioli 1998, 2000]. Zunächst hatten diese Autoren verschiedene Verfahren miteinander verglichen; durch den sehr populären Übersichtsartikel [Hind 2001] wurde das darin ermittelte Ergebnis leider weit verbreitet und in der Folge als gegeben angesehen, dass Fluss-Sensitivität keinen nennenswerten Präzisionsgewinn für den zu zahlenden Preis liefert.

Es gibt jedoch gute Gründe, warum man aus dem Artikel von Hind und Pioli [1998] nicht voreilig diesen Schluss ziehen sollte:

- Der Artikel hat nur sehr kleine Programme analysiert. Bei diesen pflanzen sich Ungenauigkeiten der FI-Variante noch nicht so weit fort wie es bei großen Programmen möglich ist (z.B. globale Variablen mit vielen Beziehungen). Für große Programme ist daher ein deutlicherer Präzisionsgewinn durch Fluss-Sensitivität zu erwarten.

- Das als Vertreter der fluss-sensitiven Analysen betrachtete Verfahren beherrschte keine indirekten starken Aktualisierungen und nutzte somit nicht das volle FS-Potential. Umgekehrt errechnete das als Vertreter der fluss-insensitiven Analysen betrachtete Verfahren ein Alias-Set pro Objekt *und Funktion*, nicht wie bei Andersen global. Die Genauigkeit ist damit höher als bei gewöhnlichen fluss-insensitiven Analysen.

- Der Artikel hat nur kontext-insensitive Verfahren verglichen. Eine erhöhte Genauigkeit durch Kontext-Sensitivität mag den Präzisionsgewinn der Fluss-Sensitivität noch verstärken, z.B. durch weitere starke Aktualisierungen. Dies dürfte insbesondere beim häufigen Konstrukt eines Referenzparameters der Fall sein. So berichten Stocks u. a. [1998] von akzeptablen Kosten einer FSCS-Analyse gegenüber einer FICI-Analyse angesichts des Nutzens (in diesem Fall für die Bestimmung von Seiteneffekten).

- Die im Artikel betrachteten Verfahren waren feld-insensitiv, so dass auch die fluss-sensitive Analyse sehr viele schwache Aktualisierungen annehmen muss. Feld-Sensitivität würde deutlich mehr starke Aktualisierungen erlauben, so dass wiederum ein größerer Vorteil für die fluss-sensitive Variante entsteht.
- Die Kosten der Fluss-Sensitivität können sich in einer Analysen-Kette nivellieren, da nachfolgende Analysen von der höheren Präzision und dem damit verbundenen geringeren Mengengerüst profitieren können.

Die vorliegende Dissertation versucht, diese Kritikpunkte mit einem neuen Vergleich auszuräumen (vgl. Teil II). Unsere Evaluation betrachtet dazu Programme mit bis zu mehr als 200 KLoC und verwendet als fluss-sensitive Analyse die hier neu eingeführte Kernanalyse. Diese haben wir auch feld-sensitiv und mit der Unterstützung für indirekte starke Aktualisierungen vorliegen (vgl. Kapitel 8), so dass dieses Potential genutzt werden kann. Zum Vergleich werden wir Andersens Analyse einsetzen, um nachfolgend ebenfalls die ISSA-Form zu errichten.

Kapitel 5

Umsetzung des Graphaufbau-Schrittes

Bislang haben wir die kombinierte Analyse auf einer allgemein gehaltenen, konzeptionellen Ebene beschrieben. Eine konkrete Realisierung kann dieses Konzept auf verschiedene Weise umsetzen. Die jeweils gewählte Variante beeinflusst natürlich den Zeit- und Platzbedarf ebenso wie die Gestalt der Resultate.

In diesem Kapitel sowie in den folgenden beiden Kapiteln bieten wir konkrete Algorithmen an für eine Umsetzung der kombinierten Analyse. Im Fokus steht hierbei die fluss-sensitive, kontext-insensitive Ausprägung. Dieses Kapitel beginnt mit der Beschreibung für den Graphaufbau-Schritt. Die folgenden beiden Kapitel ergänzen dann die Algorithmen für die Propagierung sowie für einige weitere Aufgaben der Analyse. Insbesondere ergänzen wir in Abschnitt 7.2 die Behandlung einiger Sonderfälle im Zusammenhang mit Zyklen des Aufrufgraphen.

Zur Illustration einiger Aspekte der Algorithmen verwenden wir in diesem Kapitel das in Abbildung 5.1 gezeigte Beispiel eines C-Programmes. Dieses Programm hat keinen tieferen Sinn; es bietet jedoch einige Konstrukte, die sich für die Erklärungen eignen. Wir werden davon zu den jeweiligen Algorithmen passende Teilaspekte betrachten; eine vollständige Simulation des Verfahrens am Beispiel wäre zu langwierig.

```
int x;

int f (int* p) {
    if (p != 0) {
        x = *p;
        return (*p) + 1;
    } else
        return x;
}

int main () {
    int y = 1;
    int* q = &y;
    x = 2;
    return f (q);
}
```

Abbildung 5.1: Laufendes Beispiel zur Illustration der Algorithmen

5.1 Genereller fluss-sensitiver Graphaufbau

Unsere Implementierung des Graphaufbau-Schrittes ist vollständig inkrementell, d.h. sie betrachtet lediglich die im Vergleich zur letzten Iteration neu erkannten Knoten. In der ersten Iteration sind diese Knoten implizit gegeben als die direkt ersichtlichen Definitionen und Verwendungen. In allen weiteren Iterationen legt jeweils die Propagierung aus Iteration i eine Menge *New_ Nodes* von neuen Knoten an, um dem Graphaufbau der Iteration $i+1$ die notwendigen Daten zu liefern. Im Laufe des Graphaufbaus erkennt der Algorithmus neue, induzierte Knoten. Diese behandelt er noch in der gleichen Iteration analog zu anderen neuen Knoten. Am Ende des Graphaufbau-Schrittes löschen wir das dann veraltete *New_ Nodes* jeweils.

Wie üblich im Zusammenhang mit SSA-Algorithmen benutzt auch unsere Implementierung Dominanz-Informationen zur schnellen Verknüpfung von Definitionen und Verwendungen. Das Verfahren bestimmt neue Datenfluss-Kanten sowie von den neuen Knoten induzierte Knoten (Seiteneffekte und ϕ-Knoten) und aktualisiert damit den aktuellen ISSA-Graphen. Es propagiert zudem neue Seiteneffekte zu Aufrufern.

Unsere inkrementelle ISSA-Konstruktion basiert auf Ideen, die von T.B. Tok beschrieben wurden [Tok 2007; Tok u. a. 2006]. Seine Ausführungen dazu sind jedoch sehr knapp und teils unvollständig. Außerdem weichen wir an ein paar Stellen von seinen Vorschlägen ab. So nutzen wir z.B. für temporär nicht initialisiert erscheinende Objekte die Sonderdefinition am Beginn des jeweiligen Lebensbereiches, anstatt die zugehörigen allgemeinen Verwendungen in *New_Nodes* zu behalten.

Im Folgenden beschreiben wir zunächst die Hauptfunktion zum Graphaufbau und erklären anschließend die Hilfsfunktionen, die darin zum Einsatz kommen. Dabei nutzen wir die Sichtweise mit (lokalen) Objektgraphen und deren inkrementeller Konstruktion. Abschließend vollenden wir den Weg zu den Details und gehen auf Datenstrukturen für eine effiziente Umsetzung ein.

```
procedure FS_Graph_Construction is
begin
   for all N∈New_Nodes loop
      -- New_Nodes can grow, use as work list
      case N is
      when Definition_Node | Entry_Node | Subentry_Node =>
         Add_Induced_Nodes (N);
         Insert_Into_Object_Graph
           (N, Create_Edge => Is_Weak_Update (N));
         Connect_Object_Graphs (N);
      when Use_Node | Exit_Node | Subexit_Node =>
         Insert_Into_Object_Graph (N, Create_Edge => True);
         Connect_Object_Graphs (N);
      when Phi_Node =>
         Add_Induced_Nodes (N);
         Insert_Phi_Into_Object_Graph (N);
      end case;
   end loop;
end FS_Graph_Construction;
```

Abbildung 5.2: Fluss-sensitiver Graphaufbau

Abbildung 5.2 zeigt die Hauptfunktion zur Realisierung der fluss-sensitiven Ausprägung des Graphaufbau-Schrittes. Sie behandelt schlicht alle neuen Knoten in irgendeiner Reihenfolge. Die Schleife über New_Nodes muss je-

doch damit umgehen können, dass die Behandlung eines Knotens auch wieder neue Knoten erzeugt und somit New_Nodes erweitert.

Neue Definitionen und (Unter-)Eingänge Allgemeine Definitionen erzwingen möglicherweise neue ISSA-Knoten, weswegen wir das weiter unten beschriebene Add_Induced_Nodes aufrufen. Anschließend wollen wir den neuen Knoten in den jeweiligen Objektgraphen einfügen. Bei einer starken Aktualisierung benötigen wir dabei nicht unbedingt eine vorausgehende Definition, so dass der Create_Edge-Parameter für Insert_Into_Object_Graph über die Frage bestimmt werden muss, ob eine starke Aktualisierung vorliegt. Abschließend verbleibt noch die Aufgabe, verschiedene Objektgraphen miteinander zu verbinden. Die zugehörige Funktion Connect_Object_Graphs besprechen wir in Kürze.

In unserem Beispiel aus Abbildung 5.1 haben wir z.B. eine direkte Definition von x im Then-Zweig in f. Diese muss einen ϕ-Knoten nach dem `if` induzieren (d.h. am Anfang des Exit-Grundblocks im CFG), und da es sich um eine nicht-lokale Variable handelt, muss auch ein Ausgang entstehen. Dies übernimmt dann Add_Induced_Nodes. In diesem Fall handelt es sich bei der direkten Definition um eine starke Aktualisierung, so dass der Create_Edge-Parameter keinen Bedarf für eine Kante anzeigt. Connect_Object_Graphs versucht hier, den lokalen Objektgraphen von x in f mit dem Objektgraphen der RHS zu verbinden; dort ist jedoch zu diesem Zeitpunkt in der ersten Iteration noch kein Ziel bekannt, so dass nichts geschieht.

Neue Verwendungen und (Unter-)Ausgänge. Für allgemeine Verwendungen entfällt der Aufruf an Add_Induced_Nodes. Außerdem ist stets klar, dass wir eine vorausgehende Definition erwarten. Das erklärt den übersichtlichen Code zum Umgang mit einer neuen allgemeinen Verwendung. Abschließend verbinden wir auch hier verschiedene Objektgraphen miteinander.

Im Beispiel können wir nun die rechte Seite der eben betrachteten direkten Zuweisung an x inspizieren. Hier haben wir zunächst in der ersten Iteration eine Verwendung von p für die Dereferenzierung, und anschließend in der zweiten Iteration eine (indirekte) Verwendung des Zeigerziels y (vgl.

Beispiel 6.3.1). In beiden Fällen ermitteln wir die vorausgehende Definition. Für den Parameter p ist dies der Eingang am Start von f, und für die nichtlokale Variable y finden wir lokal keine, so dass ein Eingang angelegt werden muss. Für y befasst sich der Aufruf an Connect_Object_Graphs wieder mit der Verbindung der Objektgraphen, und diesmal kann eine solche zwischen den Graphen für x und y angelegt werden.

Neue ϕ-Knoten. Neue ϕ-Knoten wiederum induzieren ihrerseits möglicherweise andere ϕ-Knoten. Dafür ist es hier jedoch nicht nötig, verschiedene Objektgraphen miteinander zu verbinden. Die hier aufgerufene Funktion Insert_Phi_Into_Object_Graph unterscheidet sich vom bereits erwähnten Insert_Into_Object_Graph insofern, dass sie pro Argument des ϕ-Knotens nach einer Definition fahndet, und dass dies wie bei einer allgemeinen Verwendung immer geschieht.

Im Beispiel haben wir nur den ϕ-Knoten für x in f. Da das dort beendete `if` nicht weiter in äußere Schleifen oder `ifs` eingeschachtelt ist, muss Add_Induced_Nodes nichts tun (auch der Ausgang für x wurde schon von der direkten Definition hervorgerufen). Für das Argument zum Then-Zweig finden wir hier die direkte Definition von x. Für das andere Argument findet sich lokal jedoch keine vorausgehende Definition, so dass in f ein Eingang für x angelegt werden muss (wenn dies nicht bereits durch die Verwendung von x im Else-Zweig zuvor erledigt wurde).

5.2 Induzierte ISSA-Knoten ergänzen

Als Hilfsfunktion ist in Abbildung 5.3 Add_Induced_Nodes zu sehen. Sie wird für eine neue allgemeine Definition N aufgerufen, um benötigte ISSA-Artefakte zu ergänzen. N ist dabei noch nicht im Objektgraphen (das erledigt erst später die Insert-Funktion). Zu den benötigten Artefakten zählen ϕ-Knoten und Ausgänge für May-Def-Seiteneffekte (andere Seiteneffekt-Knoten werden wir später an anderer Stelle ergänzen). Das Prädikat zur Beantwortung der Frage, ob ein Ausgang angelegt werden muss, haben wir der Übersichtlichkeit halber in eine separate Funktion ausgelagert.

```
procedure Add_Induced_Nodes (N : in Node) is
begin
   -- add phi nodes, if not there already
   for all BB ∈ Dom_Frontier (N.Instruction.BB) loop
      Add_Phi_Node (BB, N.Object);
   end loop;
   -- add may/must-def side-effect, if necessary
   if Creates_Exit (N) then
      Add_Exit (Get_Routine (N.Instruction), N.Object);
   end if;
end Add_Induced_Nodes;

function Creates_Exit (N : in Node) return Boolean is
   F : constant Routine := Get_Routine (N.Instruction);
begin
   return Objectgraph (F, N.Object) has no defs and then
          Is_Nonlocal (F, N.Object);
end Creates_Exit;
```

Abbildung 5.3: Induzierte ISSA-Knoten ergänzen

5.2.1 Induzierte ϕ-Knoten

Für eine neue allgemeine Definition müssen wir gemäß SSA-Regeln ϕ-Knoten in der iterierten Dominanzgrenze anlegen. Wir erledigen dies schrittweise: Für eine neue Definition legen wir nur in der unmittelbaren Dominanzgrenze ϕ-Knoten an (wenn nicht bereits geschehen) und fügen sie der Menge neuer Knoten hinzu. Dadurch behandelt sie der Algorithmus später selbst und ergänzt sodann die ϕ-Knoten in deren Dominanzgrenze, etc. Alternativ kann man mit dem Gedanken spielen, sofort in Add_Induced_Nodes die Kante zwischen der allgemeinen Definition N und den davon induzierten ϕ-Knoten einzuziehen; darauf gehen wir in Abschnitt 5.6 ein.

5.2.2 Induzierte Ausgänge

Eine Definition für ein nicht-lokales Objekt verursacht direkt oder indirekt einen ausgehenden Seiteneffekt, sowohl fluss-insensitiv als auch fluss-sensitiv. Dies können wir auch bei einer nachfolgenden starken Aktualisierung

so betrachten: Es gibt in jedem Fall einen Seiteneffekt auf das nicht-lokale Objekt, ob nun direkt wegen der neuen Definition oder einer anderen. Die genaue Ermittlung der auslösenden Definition ist lediglich für die einzuziehende Datenfluss-Kante relevant. Dies übernimmt initial der Ausgang als Verwendung, bei später eingeschobenen Definitionen die Inspektion der Verwendungen der vorausgehenden Definition. Daher legen wir hier bei einer neuen allgemeinen Definition für ein nicht-lokales Objekt stets einen ausgehenden Seiteneffekt an, wenn er nicht schon existiert. Diese Bedingung ist einfach zu prüfen: Wenn bereits eine allgemeine Definition im zugehörigen lokalen Objektgraphen existiert, so hat diese den Ausgang schon angelegt (oder es ist keiner nötig, d.h. ein lokales Objekt). Dieses Prädikat realisiert Creates_Exit. Dazu muss man wissen, dass die induzierten Knoten zunächst nur der Menge New_Nodes hinzugefügt und erst später selbst in die Objektgraphen aufgenommen werden. Eine Implementierung kann sich jedoch auch dafür entscheiden, dies sofort zu tun, wenn sie Creates_Exit anders realisiert.

Für unser laufendes Beispiel hatten wir bereits erwähnt, dass die direkte Definition von x in f einen Ausgang und einen ϕ-Knoten induziert. Die direkte Definition von x im Hauptprogramm erzeugt ebenfalls einen Ausgang, denn über *main* steht in der Analyse eine künstliche Funktion als Wurzel des Aufrufgraphen. In dieser Funktion werden zuerst die globalen Variablen initialisiert, dann *main* gerufen, und am Ende eventuelle Finalisierungen modelliert. Für y entsteht jedoch in *main* trotz der dort vorhandenen Definition kein Ausgang, da es sich um eine lokale Variable handelt.

5.3 Manipulation von Objektgraphen

5.3.1 Einfügen eines Knotens

Abbildung 5.4 zeigt die Funktion Insert_Into_Object_Graph. Diese ermittelt zum neuen ISSA-Knoten N die (derzeit) letzte vorausgehende gültige Definition, um eine Datenfluss-Kante anlegen zu können. Falls keine gefunden wird, so muss es sich wegen unserer Definitionsknoten für nicht initialisierte Objekte am Anfang von deren Lebensbereich um ein nicht-lokales

```
procedure Insert_Into_Object_Graph
   (N : in Node;
    Create_Edge : in Boolean)
is
begin
    Reaching_Def := Find_Reaching_Def (N);
    if Create_Edge then
        if Reaching_Def = none then
            Reaching_Def := Add_Entry (N);
        end if;
        Add_Edge (Reaching_Def, N);
    end if;
    Adjust_Object_Graph (Reaching_Def, N);
end Insert_Into_Object_Graph;
```

Abbildung 5.4: Neuen Knoten in einen Objektgraphen einfügen

Objekt handeln. In diesem Fall müssen wir davon ausgehen, dass die benötigte Definition über einen Aufrufer ins aktuelle Unterprogramm gelangt. Entsprechend legen wir dann einen Eingang für das Objekt an und fügen ihn der Menge New_Nodes hinzu. Später, wenn er behandelt wird, werden wir die dazugehörigen Unterausgänge an den jeweiligen Aufrufstellen anlegen, um dort nach der Definition zu fahnden.

Abschließend rufen wir die im nächsten Abschnitt besprochene Funktion Adjust_Object_Graph auf, um N in die Datenstruktur des Objektgraphen einzutragen und notwendige Korrekturen an den Kanten auszuführen. Da hierfür die dominierende Definition (sofern existent) als Parameter verlangt wird, suchen wir diese selbst für eine starke Aktualisierung N. In dem Falle jedoch können wir uns das Einziehen der Kante sowie das eventuelle Anlegen eines Eingangs sparen, was über den zweiten Parameter Create_Edge unterschieden wird.

Die Funktion Find_Reaching_Def übernimmt die Suche nach der dominierenden gültigen Definition. Wir werden in Abschnitt 5.5 Implementierungsvarianten für diese zentrale Aufgabe diskutieren.

ϕ-Knoten. Ist N ein neuer ϕ-Knoten, so sehen die Aktionen beim Einfügen geringfügig anders aus. Dann verwenden wir stattdessen die in Abbildung 5.5 gezeigte Funktion Insert_Phi_Into_Object_Graph.

```
procedure Insert_Phi_Into_Object_Graph (N : in Node) is
begin
   -- for the phi's arguments seen as use nodes
   for V in N.Arguments loop
      Insert_Into_Object_Graph (V, Create_Edge => True);
   end loop;
   -- for the phi node seen as definition
   Reaching_Def := Find_Reaching_Def (N);
   Adjust_Object_Graph (Reaching_Def, N);
end Insert_Phi_Into_Object_Graph;
```

Abbildung 5.5: Neuen ϕ-Knoten in einen Objektgraphen einfügen

Bei einem ϕ-Knoten führen wir die obigen Schritte pro Argument des Knotens aus, wobei wir zwingend eine gültige Definition pro Argument erwarten. Hier betrachten wir also den ϕ-Knoten zumindest konzeptionell als Verwendung in jedem vorausgehenden Grundblock (was wir als Argument bezeichnen) sowie als Definition. Ob für die Argumente tatsächlich Knoten als Datenstrukturen angelegt werden, bleibt der Implementierung überlassen.

Abschließend wollen wir den ϕ-Knoten auch als Definition betrachtet in den Objektgraphen einfügen. Dabei können wiederum Kantenkorrekturen notwendig sein. Daher enthält der Algorithmus die abschließende Suche nach dessen dominierender Definition (die nicht zwingend existieren muss), um die Korrektur durchführen zu können. Die Suche kann man sich dabei ersparen, wenn zufällig eine der Definitionen für die Argumente nicht nur das jeweilige Argument, sondern auch den ϕ-Knoten dominiert; dann hat man da bereits gefunden, wonach in der Abbildung abschließend separat gesucht wird. In unserem laufenden Beispiel ist dies der Fall, da für das zum Else-Zweig gehörende Argument des ϕ-Knotens von x als Definition der dafür anzulegende Eingang am Start von f ermittelt wird; dieser dominiert dann den ϕ-Knoten selbst. Mit den Überlegungen aus Abschnitt 3.3.3 wissen wir auch, dass es maximal eine solche dominierende Definition gibt.

5.3.2 Erweiterung und Korrektur eines Objektgraphen

Da die Zeigerziele erst im Laufe der Analyse bekannt werden, müssen wir auch die einzelnen Objektgraphen dynamisch errichten. Außer dem Hinzufügen eines neu erkannten ISSA-Knotens n müssen wir dabei auch die bisherigen Kanten *innerhalb* eines Objektgraphens anpassen. Das ist dann der Fall, wenn n zwischen den Enden einer bisherigen Kante auftaucht. Dies kann sowohl innerhalb einer Iteration geschehen (s. Beispiel) als auch aufgrund dessen, dass manche Knoten des Objektgraphen in einer späteren Iteration als andere Knoten bekannt werden. Betrachten wir das folgende Beispiel:

```
S1: X = 0;
    if (...)
S2:     X = 1;
S3: Y = X;
```

Wenn wir von einer Implementierung ausgehen, welche die Reihenfolge der Bearbeitung dieser Anweisungen nicht festlegt, so könnten wir Anweisung S3 vor S2 bearbeiten. Entsprechend ist dann noch kein ϕ-Knoten zu S2 bekannt, so dass die Verwendung von X in S3 die Definition in S1 als die gültige Definition ansieht. Die so entstehende Kante zwischen Definition und Verwendung von X müssen wir dann später korrigieren, wenn der ϕ-Knoten erscheint. Analog verhält es sich, wenn S2 eine indirekte Definition mit Ziel X wäre oder ein Aufruf mit einem Seiteneffekt auf X.

Die Funktion Adjust_Object_Graph aus Abbildung 5.6 übernimmt diese Anpassungen des Objektgraphen. Sie fügt zunächst den neuen Knoten N in seinen Objektgraphen ein, bevor anschließend die bestehenden Kanten korrigiert werden. Wir nutzen dabei die Dominanzeigenschaften der SSA-Form aus: Eine zu korrigierende Kante muss von der N dominierenden allgemeinen Definition *Dominator* zu einem Knoten V verlaufen, der von N dominiert wird, es sei denn, V ist ein ϕ-Knoten. Dann gilt stattdessen die Dominanzbedingung zur Verankerung der Argumente am Ende der vorausgehenden Grundblöcke, wie es die Abbildung 5.6 zeigt. Zur Durchführung der Operation bei ϕ-Knoten nehmen wir hier an, dass pro Argument A die derzeit gültige Definition in $Def(A)$ vermerkt ist.

```
procedure Adjust_Object_Graph
  (Dominator : in Node;
   N : in Node)
is
begin
   Add_Node (Dominator, N);
   return if N is not a general definition or Dominator = none;
   for all Dominator → V ∈Local_Edges loop
      if Is_Phi (V) then
         Delete := True;
         for all A ∈ Arguments (V) loop
            if Def (A) = Dominator then
               if N dom A.Basic_Block.Last_Instruction then
                  Def (A) := N; Add N → V if not done yet;
               else   Delete := False;
               end if;
            end if;
         end loop;
         remove Dominator → V  if Delete;
      elsif N dom V then
         change Dominator → V  to N → V
      end if;
   end loop;
end Adjust_Object_Graph;
```

Abbildung 5.6: Objektgraph erweitern und alte Kanten anpassen

Betrachten wir die Veränderung des Objektgraphen von X im Beispiel von eben. Nehmen wir an, nach S1 wird S3 zuerst bearbeitet und findet für X die Definition in S1 (legt also eine entsprechende Kante $S1 \rightarrow S3$ an). Anschließend folgt die Bearbeitung von S2. Dazu gehört der Aufruf von Add_Induced_Nodes, welcher den ϕ-Knoten einfügt. Adjust_Object_Graph für die Definition S2 betrachtet dann die von S1 ausgehenden Kanten, also diejenige zu S3. Weil S2 jedoch nicht deren Endknoten dominiert, muss keine Korrektur erfolgen. Anders dagegen die abschließende Behandlung des ϕ-Knotens: Diese findet als den ϕ-Knoten dominierende Anweisung ebenfalls S1, so dass wieder in Adjust_Object_Graph die Kante $S1 \rightarrow S3$ betrachtet wird. Der ϕ-Knoten dominiert den Endknoten dieser Kante. Daher verschieben wir diese, um den gewünschten Objektgraphen zu erreichen.

5.4 Verbindungen zwischen Objektgraphen

Es verbleibt noch der Umgang mit neuen Kanten *zwischen* verschiedenen Objektgraphen. Diese kommen an verschiedenen Stellen zustande:

- *Lokale* Kanten zwischen Objektgraphen entstehen durch Kopierzuweisungen. Im Einzelnen ergeben sich die folgenden Kanten:
  - Für `x = y` entsteht die Kante von der Verwendung von y zur Definition von x.
  - Für `x = (*y)+d` entsteht für jedes Objekt t, das als Ziel von y ermittelt wurde, eine Kante von der Verwendung des Objekts mit Index $t + d$ zur Definition von x.
  - Für `(*x)+d = y` entsteht für jedes Objekt t, das als Ziel von x ermittelt wurde, eine Kante von der Verwendung von y zur Definition des Objekts mit Index $t + d$.

  Die letzten beiden Arten von Kanten können dabei nicht in der ersten Iteration auftreten, da dann noch keine Zeigerziele bekannt sind.

- *Interprozedurale* Kanten entstehen durch Seiteneffekte sowie für die Parameterübergabe und Funktionsrückgabe. Im Einzelnen ergeben sich die folgenden Kanten für eine Aufrufkante von der Aufrufstelle c zum Unterprogramm f:
  - Für einen May-Use-Seiteneffekt von f auf x: Der Effekt ist über einen Eingang e in f modelliert, zu dem an c ein Unterausgang u vorhanden sein muss. Dann wird die Kante $u \to e$ angelegt.
  - Für einen May-Def-Seiteneffekt von f auf x: Der Effekt ist über einen Ausgang a in f modelliert, zu dem an c ein Untereingang u vorhanden sein muss. Dann wird die Kante $a \to u$ angelegt.
  - Für Parameter p von f: Ist p eine Ellipse, so gibt es pro Argument an c an einer Position größer als der des letzten formalen Parameters von f eine Kante vom Unterausgang des Arguments zum Eingang für p. Andernfalls, wenn p der n-te Parameter ist,

verbindet eine Kante das n-te Argument (dessen Unterausgang) an c mit dem Eingang für p.

– Für die Rückgabe von f: Falls c die Rückgabe an eine (Hilfs)variable h zuweist, so existiert die Kante vom Ausgang für das Rückgabe-Objekt zu dieser Definition von h an c, welche formal als Untereingang fungiert. Im Falle einer strukturierten Rückgabe gibt es für jedes Rückgabe-Objekt eine entsprechende Kante zur jeweiligen Hilfsvariable h_i an c.

Abbildung 5.7 zeigt nun Connect_Object_Graphs. Mit dieser Funktion ermitteln wir, ob zu einem neuen Knoten N ein Partnerknoten $Partner$ existiert, so dass eine Kante zwischen N und $Partner$ (in irgendeiner Richtung) notwendig wird.

Der Partner für die Verbindung ist dabei noch nicht zwingend als neuer Knoten behandelt worden. Im Falle eines Ausgangs ist er oft (nämlich mindestens an allen direkten Aufrufstellen zum aktuellen Unterprogramm) noch nicht einmal bekannt als neuer Knoten. Falls der Knoten noch nicht bekannt ist, liefert Add_Subentry *none* und übernimmt die Meldung des neuen Knotens; andernfalls liefert es den bereits bestehenden Untereingang. Analog verhält sich Add_Subexit. Get_Def_Node, Get_Use_Node, sowie Get_Entry und Get_Exit liefern ebenfalls *none*, wenn der jeweilige Knoten noch nicht existieren sollte, müssen diesen dann aber nicht als neu vermelden.

Es gibt verschiedene Wege, um mit dem Problem umzugehen, dass der Partner für den Connect-Aufruf noch nicht existiert. Die Abbildung zeigt die Strategie, dass beide Partner Connect aufrufen, wobei nur derjenige Erfolg haben wird, der als letzter von beiden erzeugt wird. Dann ist der andere schon vorhanden, und früher kann die Verbindungskante nicht eingezogen werden.

Für die interprozeduralen Kanten ist die Kenntnis des Aufrufgraphen wichtig. Dieser kann jedoch im Laufe der Analyse anwachsen, wenn neue Ziele für indirekte Aufrufe bekannt werden. In einem solchen Fall werden die hier gezeigten Aktionen für die jeweils neu erkannte Aufrufkante nachgeholt. Abschnitt 6.3.3 wird mehr dazu beschreiben.

```
procedure Connect_Object_Graphs (N : in Node) is
begin
   case N is
   when Entry =>
      for all c∈Call_Sites calling Get_Routine (N.Instruction) loop
         Partner := Add_Subexit (c, N.Object);
         Add_Interprocedural_Edge (Partner, N);
      end loop;
   when Exit =>
      for all c∈Call_Sites calling Get_Routine (N.Instruction) loop
         Partner := Add_Subentry (c, N.Object);
         Add_Interprocedural_Edge (N, Partner);
      end loop;
   when Subentry =>
      for all f∈Callees (N.Instruction) loop
         Partner := Get_Exit (f, N.Object);
         Add_Interprocedural_Edge (Partner, N);
      end loop;
   when Subexit =>
      for all f∈Callees (N.Instruction) loop
         Partner := Get_Entry (f, N.Object);
         Add_Interprocedural_Edge (N, Partner);
      end loop;
   when Definition =>
      if N.Instruction∈ Copy_Assignments and then
         RHS (N.Instruction) is not a dereference
      then
         Partner := Get_Use_Node (RHS (N.Instruction));
         Add_Local_Intergraph_Edge (Partner, N);
      end if;
   when Use =>
      if N.Instruction∈ Copy_Assignments and then
         LHS (N.Instruction) is not a dereference
      then
         Partner := Get_Def_Node (LHS (N.Instruction));
         Add_Local_Intergraph_Edge (N, Partner);
      end if;
   end case;
end Connect_Object_Graphs;
```

Abbildung 5.7: Kante zwischen Objektgraphen anlegen

5.5 Suche nach der gültigen Definition

Die Algorithmen haben zur Suche nach der gültigen Definition die bislang noch nicht besprochene Funktion Find_Reaching_Def benutzt. In diesem Abschnitt besprechen wir nun, wie man diese Funktion umsetzen kann, und welche Datenstrukturen dazu passen.

Formal betrachtet bildet Find_Reaching_Def einen Knoten N bzw. das zugehörige Tupel $(Instruktion, Objekt)$ ab auf den ISSA-Knoten der derzeit gültigen Definition. Die gesuchte Definition D dominiert dabei N und wird von keiner anderen Definition mit der gleichen Eigenschaft unterdrückt. D ist damit die erste Definition zu $N.Object$, die wir im Dominanzbaum antreffen, wenn wir diesen ab $N.Instruction$ aufwärts traversieren.

5.5.1 Der CFG-Dominanzbaum als Basis

Das führt direkt zu einem Implementierungsansatz, wie er in Abbildung 5.8 gezeigt ist. Der Einfachheit halber ist der Pseudocode hier ohne eine Gruppierung von Anweisungen zu Grundblöcken abgefasst. Für eine Definition prüft dabei Get_Def, ob diese das gleiche Objekt wie N betrifft, so dass eine gültige Definition vorliegt. In diesem Falle realisiert diese Funktion auch gleich die Abbildung auf den zugehörigen Definitions-Knoten. Falls keine Definition des Objekts vorliegt, so liefert die Funktion dagegen *none*.

Damit haben wir aber erst einen Teil der Aufgabe erfüllt, nämlich die Suche nach der dominierenden Instruktion mit einer geeigneten Definition. Es fehlt noch die Abbildung von dieser Instruktion auf den ISSA-Knoten zur Definition, d.h. die Realisierung von Get_Def.

Für Instruktionen, die genau ein Objekt definieren, ist dies einfach durch ein Array realisierbar, welches mit der Instruktionsnummer indiziert wird. Es verbleiben damit die Instruktionen, die mehrere Objekte definieren könnten, also indirekte Definitionen, ϕ-Knoten- und (Unter-)Eingangsinstruktionen. Hierfür haben wir in unserer Implementierung einige Ideen ausprobiert; leider zeigte sich jedoch keine davon zugleich effizient und einigermaßen moderat im Platzbedarf. So ist beispielsweise der Aufbau einer einfach verketteten Liste von Knoten an jeder Instruktion kompakt, verlangt aber eine lineare

```
function Find_Reaching_Def (N : Node) return Node
is
   IDom : Node := N.Dominator;
   Def : Node;
begin
   while IDom ≠ none loop
      Def := Get_Def (IDom, N.Object);
      if Def ≠ none then
         return Def;
      end if;
      IDom := IDom.Dominator;
   end loop;
   return none;
end Find_Reaching_Def;
```

Abbildung 5.8: Suche nach der gültigen Definition (via CFG-Dominanzbaum)

Suche danach, ob einer dieser Knoten das gerade relevante Objekt betrifft. Umgekehrt ist ein mit der Objektnummer indiziertes Array pro Instruktion deutlich zu platzhungrig. Auch die Idee, im jeweiligen lokalen Objektgraphen eine Suchstruktur für schon bekannte Instruktionen aufzubauen, ist wegen der Suche *pro Instruktion* nicht effizient – diese Idee führt uns jedoch zu einem alternativen Implementierungsansatz für Find_Reaching_Def, welcher nicht den CFG-Dominanzbaum traversiert, sondern eine *einmalige* Suche im lokalen Objektgraphen verwendet.

5.5.2 Suche im lokalen Objektgraphen

Betrachten wir die Alternative, Objektgraphen als Basis für die Suche nach der gültigen Definition zu benutzen. Zugrunde liegt der Gedanke, dass die Suche nach der dominierenden Definition nur bereits bekannte Definitionen *des Objekts* betrachten muss, nicht alle lokalen Instruktionen. Also protokollieren wir alle allgemeinen Definitionen in einer geeigneten, dynamisch veränderlichen Suchstruktur. Find_Reaching_Def für einen Knoten N muss dann darin den direkten Dominator für N finden.

Für die konkrete Datenstruktur legt dies zunächst die Speicherung der allgemeinen Definitionen der Objektgraphen als Bäume gemäß der Domi-

nanzrelation nahe. Jedoch liefert dies noch immer keine direkte Abbildung von *N.Instruction* auf den gesuchten Knoten im Objektgraphen; es muss noch eine Suche im Baum stattfinden. Tatsächlich haben wir diese Variante verglichen mit der Alternative, den Objektgraphen als einfache Liste (statt als Baum) zu speichern. Da sich kein nennenswerter Geschwindigkeitsunterschied zeigte, der Baum aber pro Knoten einen Verweis auf einen anderen Knoten mehr benötigt, bevorzugen wir die Listen-Implementierung.

Der Hintergedanke bei der Listenimplementierung ist der, dass pro lokalem Objektgraphen (d.h. pro Objekt und Unterprogramm) in der Regel nicht allzuviele allgemeine Definitions-Knoten auftauchen (sagen wir ca. 20). Das rechtfertigt es, die lineare Suche in der Liste gegenüber der Baumsuche zu akzeptieren. Abbildung 5.9 zeigt diese Realisierung von Find_Reaching_Def. Als Vorteil gewinnt man den reduzierten Platzbedarf, denn für die Realisierung der Liste genügt pro Knoten ein Verweis auf den nächsten Knoten der Liste, sowie pro lokalem Objektgraphen die Benennung des Listenkopfes. Da die Reihenfolge innerhalb der Liste für die Korrektheit irrelevant ist, fügen wir eine neue Definition immer am Beginn der Liste mit schnellem konstantem Aufwand ein.

```
function Find_Reaching_Def (N : in Node) return Node
is
   F : constant Routine := Get_Routine (N.Instruction);
   Def : Node := Objectgraph (F, N.Object).First;
   IDom : Node := none;
begin
   while Def ≠ none loop
      if Def.Instruction dom N.Instruction and then
         (IDom = none or else IDom.Instruction dom Def.Instruction)
      then
         IDom := Def;
      end if;
      Def := Def.Next;
   end loop;
   return IDom;
end Find_Reaching_Def;
```

Abbildung 5.9: Suche nach der gültigen Definition (via Objektgraph)

5.6 Rückwärts orientierte ISSA-Kanten

Bislang haben wir die Algorithmen für eine Implementierung beschrieben, welche in ihren Datenstrukturen die ISSA-Kanten in Datenfluss-Richtung orientiert. Dies ist in Adjust_Object_Graph nötig gewesen, um die von *Dominator* ausgehenden Kanten zu kennen (vgl. Abbildung 5.6). In diesem Abschnitt diskutieren wir kurz die Alternative, die Kanten in den Datenstrukturen entgegen der Flussrichtung anzulegen. Die Propagierung (vgl. nächstes Kapitel) kann mit beidem umgehen, daher müssen wir nur klären, wie der Graphaufbau angepasst werden muss.

Das Verlockende an der Rückwärtsorientierung der Kanten liegt darin, dass die Datenstrukturen für die Kanten einfacher werden: Jede allgemeine Verwendung hat dann genau eine ausgehende Kante, ebenso jede schwache Aktualisierung. Bei der Orientierung in Flussrichtung dagegen haben wir es stets mit einer Menge von Kanten pro Knoten zu tun. Wir müssen jedoch hierzu gleich betonen, dass dieser Vorteil der Rückwärtsorientierung nur in Bezug auf die ISSA-Kanten innerhalb der Objektgraphen gilt. Für die interprozeduralen Übergänge sowie die lokalen Verbindungen zwischen Objektgraphen bringt uns die SSA-Form hierfür keinen Vorteil.

Die einzig wirklich notwendige Änderung am Graphaufbau für rückwärts orientierte Kanten betrifft Adjust_Object_Graph. Dieses soll alle vom Dominator ausgehenden Kanten überprüfen, doch die Identifikation dieser Kantenmenge gestaltet sich bei rückwärts orientierten Kanten aufwändiger. Unsere gegenwärtige Strategie hierfür inspiziert alle lokalen Knoten des passenden Objekts und prüft jeweils die Dominanz. Dies lässt sich mit der im vorigen Abschnitt beschriebenen Suche nach der gültigen Definition vereinen, bei der ebenfalls die Liste der Knoten aus dem lokalen Objektgraphen durchlaufen wird. Einen wichtigen Unterschied gibt es jedoch: Während die Suche nach der gültigen Definition ausschließlich allgemeine Definitionen betrachtet, müssen wir für Adjust_Object_Graph auch die allgemeinen Verwendungen durchlaufen. Das beeinflusst das Mengengerüst. Aufgrund dieses teureren Vorgehens nutzt unsere Implementierung die Orientierung in Datenfluss-Richtung.

Es soll jedoch nicht verschwiegen werden, dass die Rückwärtsorientierung im Zusammenhang mit ϕ-Knoten einen Vorteil bietet. Dieser betrifft die Situation in Add_Induced_Nodes (vgl. Abbildung 5.3), bei der die ϕ-Knoten zu einer neuen allgemeinen Definition angelegt werden. An dieser Stelle kennen wir die allgemeine Definition N sowie die zugehörigen ϕ-Knoten. Die zuvor präsentieren Algorithmen haben dieses Wissen nicht ausgenutzt, sondern später pauschal für alle Argumente der ϕ-Knoten nach deren Definition gesucht. Stattdessen könnte man auch direkt N mit den (jeweils von N dominierten Argumenten der) ϕ-Knoten verbinden und sich auf die Korrektur des Objektgraphen verlassen, wenn dazwischen eine andere Definition residiert. Genauer: Man prüft zu jedem ϕ-Knoten P zu N, welche Argumente von N dominiert werden. Gibt es noch keine Definition zu einem solchen Argument, so erzeugen wir die Kante von N dorthin. Andernfalls kann immer noch N zwischen der alten Definition des Arguments und dem Argument sitzen. Dieser Fall wird dann jedoch dadurch behandelt, dass für N der Objektgraph korrigiert wird. Dies behandelt ebenso die Situation, dass die im ersten Fall eingezogene Kante von N zum Argument wegen einer anderen Definition dazwischen nicht stimmt.

Für dieses Vorgehen müssen wir jedoch wissen, welches die gegenwärtige Definition eines Arguments von P ist. Bei der Rückwärtsorientierung ist dies nun gegeben. Die Graphkorrektur für die Argumente können wir dabei sofort durchführen und die Betrachtung von ϕ-Knoten als Kanten-Endpunkte in Adjust_Object_Graph (vgl. oben) entsprechend einsparen: Ist zu einem Argument von P bereits die Definition D eingetragen, wir finden nun aber D dom N und N dominiert das Argument, so können wir gleich die alte Kante ersetzen (N ist die neue Definition zum Argument). Die so veränderte Passage von Add_Induced_Nodes zeigt uns Abbildung 5.10.

Man beachte, dass wir später dennoch bei der Behandlung des ϕ-Knotens all dessen Argumente inspizieren müssen: Solche, die dann noch keine Definition erhalten haben, beziehen sich derzeit bei einem lokalen Objekt auf die künstliche Definition an dessen Lebensbeginn bzw. bei einem nicht-lokalen Objekt auf einen anzulegenden Eingang.

```
for all BB ∈ Dom_Frontier (N.Instruction.BB) loop
   Phi := Add_Phi_Node (BB, N.Object);
   for all A ∈ Phi.Arguments loop
      if N dom A then
         D := Def (A);
         if D = none then
            Add_Edge (N, A);
         elsif D dom N then
            change D→A to N→A;
         end if;
      end if;
   end loop;
end loop;
```

Abbildung 5.10: Verändertes Anlegen induzierter ϕ-Knoten bei rückwärts orientierten Kanten

Kapitel 6

Umsetzung des Propagierungs-Schrittes

In diesem Kapitel setzen wir die Schilderung der algorithmischen Umsetzung unserer Analyse fort. Nach den Algorithmen für den Graphaufbau im vorigen Kapitel folgen nun die Algorithmen zur Propagierung von Zeigerzielen. Zeigerarithmetik unterstützen wir dabei nur wie in Kapitel 3 beschrieben als Identität auf den betroffenen Objekten, so dass wir pro Iteration das Grapherreichbarkeits-Problem zwischen den Zeigerziel-Quellen und den Dereferenzierungen zu lösen haben.

6.1 Variationsmöglichkeiten für die Propagierung von Zeigerzielen

Zur Lösung dieses (auch für Andersens Analyse anfallenden) Problems gibt es viele Variationsmöglichkeiten. Die Literatur kennt im Wesentlichen den Ansatz, die (dynamische) transitive Hülle des Graphens zu berechnen, über den die Zeigerziele propagiert werden. In unserem Falle ist dies der ISSA-Graph, bei Andersen der Constraint-Graph. Um die Aufgabe zu beschleunigen, können dabei diverse Graphvereinfachungen wie eine Zyklenkontraktion zum Einsatz kommen. (Unsere Evaluation in Kapitel 13 zeigt, dass dies tatsächlich die Effizienz spürbar verbessert.)

Im Folgenden betrachten wir einige Variationsmöglichkeiten:

- *Propagierungsrichtung:* Wir können vorwärts propagieren, d.h. Zeigerziel-Quellen in Datenfluss-Richtung zu den Dereferenzierungen, oder umgekehrt.

- *Propagierungsgraph:* Wir können direkt auf dem ISSA-Graphen propagieren oder auf einer reduzierten Fassung, die im Wesentlichen nur noch die allgemeinen Definitionen enthält. Außerdem können wir den Graphen durch eine Zyklenkontraktion weiter verkleinern.

- *Inkrementell oder erschöpfend:* Wir können in jeder Iteration erneut das Problem vollständig lösen oder nur die Veränderungen im Graphen zugrunde legen.

- *Knoten oder Ziele:* Wir können das Problem wie für einen allgemeinen Graphen lösen, d.h. transitive Kanten zwischen Knoten ermitteln, oder stattdessen pro Knoten die *Zeigerziele* vermerken, die dorthin propagiert wurden.

- *Reihenfolge:* Wir können eine gewöhnliche Tiefensuche verwenden oder in topologischer Ordnung vorgehen (besonders im Falle des SCC-DAG).

Diese Variationsmöglichkeiten haben uns dazu veranlasst, die folgenden Abstraktionen im Algorithmus zu verwenden:

- *Quellen* und *Senken*, abhängig von der Richtung.

- *Propagierungselemente*, abhängig von der Richtung und der Frage, ob Knoten oder Ziele propagiert werden. Dies ist eine Abbildung von Quellen auf Elemente, welche rückwärts stets die Identität ist, und vorwärts stets die Zeigerziel-Quelle auf das Zeigerziel abbildet.

- *Propagierungsgraph* als Abstraktion über dem ISSA-Graphen, abhängig von den gewählten Vereinfachungen bzw. der errichteten Hierarchie von Graphen.

Daneben gibt es für den prinzipiellen Algorithmus noch die Frage, ob er eine Tiefensuche oder eine Traversierung in topologischer Ordnung ist, und ob er in jeder Iteration alles oder nur die Unterschiede propagiert.

Wir diskutieren zunächst in den folgenden Abschnitten Vor- und Nachteile der verschiedenen Varianten, bevor wir dann im Anschluss ein konkretes Framework zur Unterstützung einiger der Varianten beschreiben.

6.1.1 Propagierungsgraph

Zunächst stellt sich die Frage, auf welchem Graphen wir die Propagierung durchführen, um Quellen und Senken miteinander zu verbinden. Prinzipiell muss dies auf dem ISSA-Graphen erfolgen, jedoch lässt sich dieser für die Propagierungszwecke auch vereinfachen. Wir betrachten hier zwei solche Vereinfachungen, die das Mengengerüst deutlich reduzieren und damit die Propagierung beschleunigen können. Unsere Implementierung kann beide benutzen, und wie die Evaluation in Teil II noch zeigen wird, gelingt ihr damit eine deutliche Laufzeit-Verbesserung.

6.1.1.1 Konzentration auf allgemeine Definitionen

Ein Weg zur Reduktion der Graphgröße besteht darin, den ISSA-Graphen auf allgemeine Definitionen zu reduzieren und die Kanten entsprechend anzupassen. Das bedeutet, dass wir während der gesamten Fixpunkt-Iteration im Graphaufbau eine reduzierte ISSA-Form erstellen, die nur aus allgemeinen Definitionen besteht, und erst nach Erreichen des Fixpunktes in einem Durchgang die fehlenden Knoten ergänzen. Die übrigen (allgemeinen Verwendungs-)Knoten fallen somit innerhalb der Fixpunkt-Iteration weg – bis auf die Verwendungen von Zeigern in Dereferenzierungen. Diese annotieren wir an ihre jeweilige Definition, um neu erkannte Zeigerziele der Definition an den richtigen Stellen zu behandeln.

6.1.1.2 Zyklenkontraktion

Wie im fluss-insensitiven Fall können wir auch Zyklen kontrahieren und die Propagierung auf dem SCC-DAG durchführen. Das reduziert die Eingabe-

größe für die (teure) Propagierung, verursacht dafür aber die Kosten der Zyklenkontraktion. Neben dem Zeitbedarf (v.a. fürs „Liften“ der Kanten vom ISSA-Graphen auf den SCC-DAG) ist hier auch der Platzbedarf bei expliziter Speicherung des SCC-DAG relevant. Tatsächlich ist Andersens Verfahren in den letzten Jahren vor allem über clevere Methoden zur Zyklenerkennung beschleunigt worden, und unsere Messungen legen nahe, dass ein ähnlicher Effekt auch für die hier vorgestellte fluss-sensitive Analyse existiert.

Für den Ansatz der Zyklenkontraktion können wir zwei orthogonale Dimensionen mit jeweils zwei verschiedenen Möglichkeiten herausarbeiten. Die erste Dimension benennt den zugrunde liegenden Graphen, von dem Zyklen gesucht und kontrahiert werden. Die beiden Möglichkeiten hierfür sind:

1. Der vollständige ISSA-Graph.

2. Der auf allgemeine Definitionen reduzierte ISSA-Graph.

Die Verkleinerung des Graphen aus dem vorigen Abschnitt lässt sich also mit der Zyklenkontraktion kombinieren.

Die zweite Dimension unterscheidet danach, ob der SCC-DAG der Iteration i zur Beschleunigung der nächsten Zyklenkontraktion in Iteration $i+1$ genutzt wird oder nicht. Die beiden Möglichkeiten hierfür sind:

1. Eine erschöpfende Zyklenkontraktion, bei der die Zyklenkontraktion in Iteration $i+1$ ausschließlich den zugrunde liegenden Graphen betrachtet, nicht aber den alten SCC-DAG.

2. Eine inkrementelle Zyklenkontraktion, welche den alten SCC-DAG als kompakte Zusammenfassung der darin abgedeckten Knoten und Kanten ausnutzt.

Zur Zyklenkontraktion sei noch angemerkt, dass wir uns wieder entscheiden können, in welche Richtung wir in den Datenstrukturen des SCC-DAG dessen Kanten orientieren. Diese Richtung kann der des zugrunde liegenden Graphen folgen oder umgekehrt sein. Unsere Implementierung unterstützt beides.

6.1.1.3 Weitere Ansätze

Da die Kanten des ISSA-Graphen im Unterschied zum Constraint-Graphen jeweils klar zu Programmabschnitten zugeordnet werden können, ist sogar eine ganze Hierarchie von Graphabstraktionen auf Basis der Programmstruktur denkbar. So können wir beispielsweise die Propagierung unterteilen in eine lokale Propagierung ab den Eingängen der Funktionen, sowie eine globale Propagierung auf dem Graph aus Quellen und Eingängen.

Ein alternativer Ansatz zur transitiven Hülle besteht in der Verwendung der Reduktion auf das von Melski und Reps [2000] untersuchte sogenannte CFL-Erreichbarkeitsproblem. Für dieses Problem ist ein leicht subkubischer Algorithmus bekannt. Wir betrachten dieses Modell in der vorliegenden Arbeit jedoch nicht weiter.

6.1.2 Prinzipieller Algorithmus

Zur Berechnung der transitiven Hülle (für die relevanten Quellen oder Senken) können wir verschiedene Verfahren einsetzen. So können wir z.B. an jeder Quelle eine Tiefensuche starten, um den von dieser Quelle aus erreichbaren Teil zu ermitteln. Im Unterschied zu einer gewöhnlichen Hüllenberechnung haben wir bei der Vorwärts-Propagierung jedoch verschiedene äquivalente Quellen, nämlich alle Zeigerziel-Quellen, welche das gleiche Zeigerziel auf die Reise schicken. Dieses Sparpotential nutzen wir so, dass wir in den Tiefensuchen zusammen jeden Knoten nur einmal *pro Zeigerziel* (nicht pro Quelle) besuchen. Alternativ kann die Tiefensuche auch die erreichten Knoten als Menge zurückpropagieren.

Neben der Tiefensuche besteht noch die Option, Quellen und Senken über eine Graphtraversierung in topologischer Ordnung miteinander zu verbinden. Im azyklischen Fall (d.h. vor allem nach Zyklenkontraktion) genügt dann eine einzelne Traversierung über den gesamten Graphen. Da im Zyklus stets die gleiche Menge an Propagierungselementen an alle Knoten gelangt, ist innerhalb einer SCC keine Iteration notwendig, wenn wir alle Quellen in der SCC vorab kennen (ansonsten ein zusätzliche Durchgang über die SCC).

6.1.3 Propagierungsrichtung

Allgemein versuchen wir die Frage zu beantworten, zwischen welchen Quellen und Senken es eine Verbindung im Propagierungsgraphen gibt. Für die Vorwärts-Propagierung in Datenfluss-Richtung sind die Zeigerziel-Quellen die Quellen und die Dereferenzierungen die Senken; bei der Rückwärts-Propagierung verhält es sich umgekehrt. Ohne Zyklenkontraktion mag die Richtung, in der die Datenstrukturen intern die ISSA-Kanten speichern, den Ausschlag für die eine oder andere Propagierungsrichtung geben. Mit Zyklenkontraktion kann man umgekehrt die Richtung der SCC-DAG-Kanten nach der gewünschten Propagierungsrichtung festlegen. Ansonsten sehen wir die folgenden Vorteile für die Rückwärts-Propagierung:

- Wir müssen nicht pauschal jedes Vorkommen der Zahl 0 in einem C-Programm als das Null-Literal annehmen. Ebenso müssen wir nicht pauschal für jeden Aufruf an eine externe Funktion annehmen, dass das Resultat auf Unknown_Objects zeigt. Stattdessen genügt es, die tatsächlich zu Dereferenzierungen gelangenden Vertreter dieser Sorten zu behandeln, was die Zahl der Quellen und den zu traversierenden Graphanteil reduziert. Im Falle der externen Funktionen können wir dann auch ohne großen Mehraufwand die verschiedenen Effekte unterschiedlicher Aufrufe an externe Funktionen unterscheiden, z.B. wenn gewisse Abschätzungen zu einigen bekannt sind. Die Vorwärts-Propagierung muss hierfür verschiedene Propagierungselemente auf den Weg bringen.

- Mit Fokus auf die Anwendung der Analyse im Kontext von darauf aufbauenden Klienten-Analysen (engl. *client analyses*) ergibt sich die Option, nur diejenigen Dereferenzierungen als Quellen zu betrachten, welche den Klienten besonders interessieren (oder in der Folge als wichtig erachtet werden). Es ist sogar denkbar, pro Dereferenzierung eine unterschiedlich genaue Propagierung (z.B. fluss-sensitiv oder -insensitiv) anzuwenden, um so den Aufwand auf die relevanten Stellen zu konzentrieren. Somit erlaubt diese Richtung ein bedarfs- oder klienten-gesteuertes Vorgehen (engl. *demand driven* oder *client driven*). (Umgekehrt

mag es natürlich auch Szenarien geben, welche sich für die Verbreitung einer Zeigerziel-Quelle oder eines Zeigerzieles interessieren und damit die Vorwärts-Richtung präferieren. Es ist daher lohnenswert, in einem Werkzeug beide Richtungen zu unterstützen, ähnlich wie beim Slicing.)

- Der zu besuchende Graphanteil wird auch dadurch reduziert, dass wir davon ausgehen können, dass wir nur relevante Teilgraphen traversieren, die wirklich zu einer Quelle (oder den besonderen Definitionen für nicht initialisierte Zeiger) führen. Umgekehrt besucht dagegen die Vorwärts-Traversierung durchaus Teilgraphen, die nicht zu einer Dereferenzierung führen, da sie nur aus induzierten ISSA-Knoten bestehen (und somit am Ende im Pruning eliminiert werden).

- Die Zahl der Quellen bleibt konstant, während dies bei Vorwärts-Propagierung z.B. im feld-sensitiven Szenario nicht der Fall ist.

Auf den ersten Blick taucht jedoch als Nachteil auf, dass wir die Quellen auch als Propagierungselemente nutzen müssen und nicht wie bei der Vorwärts-Propagierung eine kompaktere Menge dafür haben. Man kann aber die jeweilige Definition zur Dereferenzierung als Propagierungselement benutzen und auf diese Weise für mehrere Quellen das gleiche Propagierungselement erreichen. Dann muss man jedoch damit umgehen können, dass sich diese Definition pro Dereferenzierung im Laufe der Analyse mehrfach ändern kann. Dieses Modell (auf dem SCC-DAG) scheint nach unseren Messungen das effizienteste zu sein.

6.1.4 Inkrementell oder erschöpfend

Die erschöpfende Propagierung führt in jeder Iteration erneut die Tiefensuche oder Traversierung in topologischer Ordnung aus, um die dann gültigen Beziehungen zwischen Quellen und Senken zu ermitteln. Die erschöpfende Propagierung ist damit relativ einfach zu implementieren und hat den zusätzlichen Vorteil, dass kein Speicher benötigt wird, um Zeigerziele zu vermerken (außer an Dereferenzierung natürlich).

Da eine fluss-sensitive Zeigeranalyse wie unsere einen präziseren und damit kleineren Datenfluss-Graphen erzeugt als eine fluss-insensitive Analyse, ist es tatsächlich möglich, dass die kombinierte Analyse mit weniger Speicher auskommt als eine Sequenz aus fluss-insensitiver Zeigeranalyse und ISSA-Analyse. Dies ist eine wichtige Beobachtung, da fluss-sensitive Zeigeranalysen oftmals am Speicherplatzproblem scheitern und dieses Problem bereits der Fluss-Sensitivität allgemein nachgesagt wird. Unsere Vergleichsmessungen in Kapitel 13 zeigen, dass der Platzbedarf fluss-sensitiv vergleichbar ist zum Bedarf beim ISSA-Aufbau mit fluss-insensitiver Zeigeranalyse; mit indirekten starken Aktualisierungen ist er sogar geringer.

Das inkrementelle Verfahren dagegen nimmt einen höheren Speicherverbrauch zu Gunsten einer besseren Laufzeit in Kauf. Wir beobachten, dass unsere Analyse niemals Knoten oder Kanten löscht, sondern lediglich hinzufügt oder verfeinert: Innerhalb des Graphaufbau-Schrittes der ersten Iteration ist die Reihenfolge der Operationen insofern egal, dass noch keine Propagierung stattgefunden hat. Ab der zweiten Iteration erscheinen dann nur noch Knoten und Kanten aufgrund von Zeigerzielen oder indirekten Aufrufen. Die so ergänzten Definitionen sind jedoch (in der hier besprochenen ersten Analyseversion) stets schwache Aktualisierungen (im Falle von Untereingängen ggf. über Pfade statt Kanten realisiert). Das bedeutet, dass ein einmal zu einem Knoten propagiertes Zeigerziel niemals wieder davon weggenommen werden muss.

Das berechtigt die Idee, Zeigerziele dort zu speichern, wohin sie schon propagiert wurden. In einer inkrementellen Implementierung meldet der Graphaufbau-Schritt also die Menge *neuer* Datenfluss-Kanten, auf welcher der Propagierungs-Schritt dann arbeitet. Für jede neue Kante starten wir dabei eine Propagierung für alle Propagierungselemente, die am Startknoten der Kante vermerkt sind. Es genügt daher, wenn der Graphaufbau all jene Kanten als neu meldet, welche mindestens ein Propagierungselement an ihrem Startknoten tragen.

Das Abspeichern von Propagierungselementen an wirklich allen Knoten benötigt natürlich viel Speicherplatz. Das macht die Alternative attraktiv, die Ziele nur an ausgewählten Knoten zu speichern. Die ISSA-Darstellung

erlaubt uns hierbei eine einfache Auswahl: Wenn wir die Zeigerziele (nur) an allen allgemeinen Definitionen speichern, so haben wir die Garantie, dass jeder zweite Knoten auf einem Pfad über gespeicherte Zeigerziele verfügt (da es keine direkte Kante zwischen zwei Verwendungen gibt). Dies ist wichtig für Komplexitätsbetrachtungen, da wir mittels der gespeicherten Ziele überprüfen, ob eine Propagierung abgebrochen werden kann. Allgemein können wir uns hier auf die Graphvereinfachungen stützen, die wir oben zur Verkleinerung des Propagierungsgraphen betrachtet haben.

6.2 Algorithmus für die Zyklenkontraktion

Für die Erkennung von Zyklen und darauf aufbauend die Konstruktion des SCC-DAG nutzen wir im Kern Tarjans Algorithmus. Diesen starten wir für jede jeweils noch nicht betrachtete Zeigerziel-Quelle bzw. Dereferenzierung(sdefinition) als Wurzel. Die erschöpfende Zyklenkontraktion ist damit aus der Literatur klar. Wir erwähnen hierzu nur, dass die Effizienz unserer Implementierung stark davon abhing, wie viele Allokationen zum Aufbau des SCC-DAG nötig waren. Beispielsweise erzielten wir gute Beschleunigungen dadurch, erst zu zählen, wie viele Kanten der SCC-DAG aufweist, dann einmalig ein Array in dieser Größe zu allokieren und sukzessive zu füllen, anstatt gleich zu füllen und notwendigerweise immer wieder neuen oder zusätzlichen Platz zu allokieren.

Es verbleibt somit die Beschreibung der inkrementellen Zyklenkontraktion, d.h. der Ausnutzung eines alten SCC-DAG. Hierfür gehen wir davon aus, dass der Graphaufbau-Schritt jeweils die Menge der neu erzeugten Kanten im zugrunde liegenden (ggf. reduzierten) ISSA-Graphen benennt (wobei wir nur solche speziell benötigen, deren Startknoten von einer alten SCC überdeckt ist). Diese Kanten verlaufen zunächst zwischen ISSA-Knoten. Wir liften sie vor der Zyklenkontraktion auf den alten SCC-DAG, indem der Startknoten der Kante durch die ihn überdeckende SCC ersetzt wird. Sollte es eine solche SCC auch für den Endknoten der Kante geben, so liften wir auch diesen. Verläuft die geliftete Kante dann vollständig innerhalb einer alten SCC, so können wir sie ganz ignorieren.

```
procedure Update_SCC_DAG is
begin
  -- initialization
  DFS_Number := 0;
  Stack := empty;
  for all V∈Nodes loop
    DFS (V) := 0;
    In_Component (V) := False;
    Head (V) := ∞;
    Visited (V) := False;
  end loop;
  -- main loop
  for all S∈Sources loop
    SCC := Get_Old_SCC (S);
    if SCC = none then
      if not Visited (S) then
        Visit (S);
      end if;
    elsif not Visited (SCC) then
      Visit (SCC);
    end if;
  end loop;
  -- finalization: destroy old SCC-DAG, use new one
  ...
end Update_SCC_DAG;
```

Abbildung 6.1: Aktualisierung des SCC-DAG

Abbildung 6.1 illustriert nun die Hauptfunktion für die inkrementelle Zyklenkontraktion. Wie im gewöhnlichen Tarjan-Algorithmus besteht sie im Wesentlichen aus einer Schleife über alle noch nicht behandelten Quellen. Die Besonderheit liegt nun darin, dass ein Knoten entweder eine alte SCC ist oder ein Knoten des zugrunde liegenden (ggf. reduzierten) ISSA-Graphen. Die Abbildungen behandeln diese so ähnlich wie möglich; unsere Implementierung nutzt z.B. positive Array-Indices für Knoten und negative Indices für SCCs. Initialisierung und Finalisierung regeln Auf- und Abbau der Tarjan-Datenstrukturen, und die Finalisierung ersetzt den alten SCC-DAG schließlich mit den dann neu berechneten Informationen.

Abbildung 6.2 zeigt das Unterprogramm zum Besuchen eines einzelnen

```
procedure Visit (N : in Node_Or_SCC) is
begin
   Visited (N) := True;
   Push N on Stack;
   DFS_Number := DFS_Number + 1;
   DFS (N) := DFS_Number;
   Head (N) := DFS_Number;
   for all S ∈ Succ (N) loop
      if not Visited (S) then
         Visit (S);
      end if;
      if In_Component (S) = 0 and then
         Head (S) < Head (N)
      then
         Head (N) := Head (S);
      end if;
   end loop;
   if DFS (N) = Head (N) then
      Process_Cycle (N);
   end if;
end Visit;
```

Abbildung 6.2: Aktionen an einer alten SCC bzw. einem ISSA-Knoten

Knotens bzw. einer alten SCC. Es unterscheidet sich kaum vom Analogon im normalen Tarjan-Verfahren. Lediglich die Schleife über die Nachfolger des Knotens muss die beiden Knotenarten unterscheiden. Für eine alte SCC besucht sie die alten SCC-DAG-Kanten sowie geliftete neue Kanten; für einen ISSA-Knoten nutzt sie die normale Kantendarstellung.

Das nicht gezeigte `Process_Cycle` schließlich nimmt die Knoten bzw. alten SCCs der neuen SCC vom Stack und merkt sich die Zusammensetzung der neuen SCC. Außerdem wird die von Tarjan benötigte Information `In_Component` für diese Knoten gesetzt.

Die Unterschiede unserer inkrementellen Fassung zum gewöhnlichen Tarjan-Algorithmus sind damit gering. Wir haben jedoch in unseren Messungen festgestellt, dass gerade die Einsparung der ISSA-Kanten, die bereits mit dem alten SCC-DAG abgedeckt sind, einen spürbaren Geschwindigkeitsvorteil bedeutet.

6.3 Framework für die Propagierung

Wir beschränken uns nun auf ein Vorgehen mittels Tiefensuche. Wir nehmen einen Propagierungsgraphen an, d.h. eventuelle Graphvereinfachungen gegenüber dem vollständigen ISSA-Graphen sollen bereits erfolgt sein. Dann unterscheidet sich die Propagierung vor allem danach, ob sie erschöpfend oder inkrementell erfolgt.

6.3.1 Erschöpfendes Verfahren

Für jede Quelle starten wir eine rekursive Tiefensuche auf dem von dort erreichbaren Teilgraphen. Dieser Ablauf ist in Abbildung 6.3 gezeigt.

```
procedure Exhaustive_Propagation is
begin
   for all S ∈ Sources loop
      if not visited (S) then
         Exhaustive_DFS (S, S.Prop_Element);
      end if;
      Mark all nodes as not visited;
   end loop;
end Exhaustive_Propagation;

procedure Exhaustive_DFS (V : Prop_Node; T : Prop_Object)
is
begin
   Mark V as visited;
   if V ∈ Sinks and then not PointsTo (V, T) then
      PointsTo (V, T) := True;
      Add_Target (V, T);
   end if;
   for all S ∈ Succ(V) loop
      if not visited (S) then
         Exhaustive_DFS (S, T);
      end if;
   end loop;
end Exhaustive_DFS;
```

Abbildung 6.3: Grober Ablauf der erschöpfenden Propagierung

Der Typ Prop_Node repräsentiert nun einen Knoten des Propagierungsgraphen und Prop_Object stellt ein Propagierungselement dar. Zu einer Quelle S nehmen wir an, dass *S.Prop_Element* das zugehörige Propagierungs-Element liefert. Pro Senke speichern wir uns in der zugehörigen Menge *PointsTo* ab, welche Elemente bereits eingetroffen sind. Mit der weiter unten beschriebenen Prozedur Add_Target führen wir die Aktionen aus, welche beim Eintreffen eines neuen Elements nötig sind.

```
procedure Incremental_Propagation is
begin
   for all e ∈ New_Edges loop
      for all t ∈ e.From.Prop_Elements loop
         if not PointsTo (e.To, t) then
            Incremental_DFS (e.To, t);
         end if;
      end loop;
   end loop;
end Incremental_Propagation;

procedure Incremental_DFS (V : Prop_Node; T : Prop_Object)
is
begin
   PointsTo (V, T) := True;
   if V ∈ Sinks then
      Add_Target (V, T);
   end if;
   for all s ∈ Succ(V) loop
      if not PointsTo (s, T) then
         Incremental_DFS (s, T);
      end if;
   end loop;
end Incremental_DFS;
```

Abbildung 6.4: Grober Ablauf der inkrementellen Propagierung

6.3.2 Inkrementelles Verfahren

Inkrementell starten wir für jede neue Kante eine Propagierung für alle Zeigerziele, die am Startknoten der Kante vermerkt sind. (Falls die Zyklenkon-

traktion genutzt wird, sind die neuen Kanten im SCC-DAG gemeint; andernfalls die neuen Kanten des (ggf. reduzierten) ISSA-Graphen.) Die *PointsTo*-Mengen existieren nun für alle Knoten, nicht mehr nur für Senken. Als Datenstruktur hierfür kann man z.B. Bitvektoren einsetzen. Dieser Ablauf ist in Abbildung 6.4 gezeigt.

Beispiel 6.3.1 Betrachten wir erneut das Beispiel aus dem vorigen Kapitel:

```
int x;

int f (int* p) {
      if (p != 0) {
            x = *p;
            return (*p) + 1;
      } else
            return x;
}

int main () {
      int y = 1;
      int* q = &y;
      x = 2;
      return f (q);
}
```

Abbildung 6.5: Laufendes Beispiel zur Illustration der Algorithmen

Nehmen wir der Einfachheit halber eine Vorwärts-Propagierung auf dem vollen ISSA-Graphen an. In der ersten Iteration liefert der Graphaufbau die darin neu erzeugten ISSA-Kanten. Darunter befindet sich auch die Kante von der Initialisierung des Zeigers q in *main* zum Unterausgang des Zeigers am Aufruf von f. Diese Kante ist deswegen als Start relevant, weil ihr Startpunkt bereits ein Propagierungselement besitzt (nämlich das Ziel y). Kanten, bei denen der Startknoten kein Propagierungselement besitzt, können wir ignorieren (der Graphaufbau muss sie also gar nicht erst separat melden). Von dieser Kante startet nun die Tiefensuche Incremental_DFS wie in der Abbildung gezeigt. Sie legt das aktuelle Propagierungselement T (hier: das Ziel

y) an jedem besuchten Knoten ab. In unserem einfachen Beispiel erreicht die Tiefensuche nach dem Unterausgang für q den Eingang für den Parameter p und von diesem aus die Verwendungen von p in f. Darunter befinden sich auch die Dereferenzierungen als Senken, für welche entsprechend Add_Target gerufen wird.

In der zweiten Iteration erzeugt der Graphaufbau wieder neue Kanten, nämlich für die indirekten Verwendungen von y in f und die daraus resultierenden Knoten, die eine Verbindung zur Initialisierung in *main* herstellen. Unter diesen neuen Kanten besitzt jedoch keine einen Startknoten, der ein Propagierungselement trägt. Daher erfolgt in der zweiten Iteration keine weitere Tiefensuche, und der Fixpunkt ist erreicht. □

Die Abbildung 6.4 zeigt das Verfahren als Tiefensuche pro Propagierungselement des Startknotens. Tatsächlich kann man all diese Elemente des Startknotens auch en bloc in einem Array mit einer einzigen Tiefensuche propagieren, falls die Kosten eines einzelnen Kantenbesuchs vergleichsweise hoch sind.

Dieses Array betrachten wir dabei als in zwei Bereiche unterteilt: Der erste Bereich enthält die Propagierungselemente, welche für den gerade besuchten Knoten (initial: Startknoten der Kante) neu sind. Der zweite Bereich dahinter enthält solche Elemente, die zwar am Start neu waren, auf dem gegenwärtigen Pfad aber schon bekannt sind und daher auf diesem nicht weiter beachtet werden. Sie könnten jedoch wieder relevant werden, wenn die Tiefensuche aus einem Seitenast der Rekursion zurückkehrt und dann mit einem anderen Pfad fortsetzt. Abbildung 6.6 illustriert die Zweiteilung des Arrays.

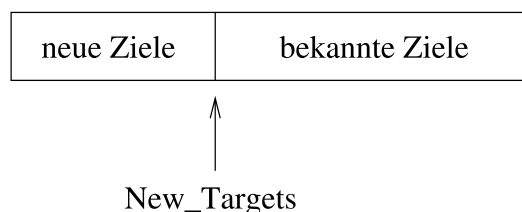


Abbildung 6.6: Nutzung des Arrays `Targets`

```
procedure Incremental_Propagation is
begin
   for all e ∈ New_Edges loop
      Targets := e.From.Prop_Elements;
      New_Targets := |Targets|;
      Incremental_DFS (e.To);
   end loop;
end Incremental_Propagation;

procedure Incremental_DFS (V : Prop_Node)
is
   Old_New_Targets : constant Natural := New_Targets;
begin
   Index := 1;
   for I in 1 .. New_Targets loop
      T := Targets (Index);
      if PointsTo (V, T) then
         Tmp := Targets (New_Targets);
         Targets (New_Targets) := T;
         Targets (Index) := Tmp;
         New_Targets := New_Targets − 1;
      else
         Index := Index + 1;
         PointsTo (V, T) := True;
         if V ∈ Sinks then
            Add_Target (V, T);
         end if;
      end if;
   end loop;
   if New_Targets > 0 then
      for all s ∈ Succ(V) loop
         Incremental_DFS (s);
      end loop;
   end if;
   New_Targets := Old_New_Targets;
end Incremental_DFS;
```

Abbildung 6.7: Verwendung eines Arrays von Propagierungselementen

Abbildung 6.7 zeigt, wie die Tiefensuche mit diesem Array abläuft. Sie nutzt dabei aus, dass die Reihenfolge der Elemente innerhalb der beiden Bereiche jeweils egal ist. Ein schon bekanntes Element T wird mit dem Element vertauscht, das an der Grenze zum Bereich der schon bekannten Elemente liegt. Anschließend wird diese Grenze so nach links verschoben, dass das Element als bekannt zählt. Nach der Rückkehr aus der Rekursion genügt dabei die Wiederherstellung der alten Grenze, um den vorigen Zustand (im Sinne einer Menge) zu erhalten.

Beispiel 6.3.2 Nehmen wir an, vor dem Besuchen des Knotens V hätte das Array `Targets` folgenden Inhalt:

$$4, 7, 1, 2 \mid 6, 99, 17, 42$$

Der Strich symbolisiert hierbei die Trennung zwischen (im Aufrufer) neuen und dort schon bekannten Zielen, d.h. $New_Targets = 4$. Nehmen wir weiterhin an, dass an V bereits die Elemente 7 und 2 bekannt (d.h. in der $PointsTo$-Menge) sind. Dann beginnt die Verarbeitung mit $Index = 1$:

$$\mathbf{4}, 7, 1, 2 \mid 6, 99, 17, 42$$

Zur Verdeutlichung haben wir das Element am aktuellen $Index$ jeweils fett gesetzt. Da das Element 4 an V neu ist, wird der Index um eins erhöht und die 4 in die $PointsTo$-Menge aufgenommen. Sollte V eine Senke sein, so würden wir an dieser Stelle Add_Target aufrufen.

$$4, \mathbf{7}, 1, 2 \mid 6, 99, 17, 42$$

Im nächsten Schritt treffen wir auf das bereits bekannte Element 7. Es wird daher mit dem Element an der Position New_Targets vertauscht, und New_Targets wird nach links verschoben. Damit ist die 7 in den Bereich der bekannten Elemente gewandert.

$$4, \mathbf{2}, 1 \mid 7, 6, 99, 17, 42$$

Der Index hat sich nicht erhöht, so dass als nächstes die gerade an diese Position getauschte 2 behandelt wird. Auch diese ist bereits bekannt und wird entsprechend umgesetzt.

$$4, \mathbf{1} \mid 2, 7, 6, 99, 17, 42$$

Abschließend betrachten wir die 1 und erkennen sie als neues Ziel an V.

Da wir noch neue Ziele haben, erfolgt nun die Rekursion zu den Nachfolgern von V. Kehren wir von dort zurück, setzen wir die Grenze New_Targets zurück auf den Wert, den sie vor der oben besprochenen Verarbeitung für V hatte. Der Zustand bei Rückkehr zum Aufrufer ist damit

$$4, 1, 2, 7 \mid 6, 99, 17, 42$$

Wir sehen, dass dies als Menge betrachtet tatsächlich den Anfangszustand wiederhergestellt hat. □

6.3.3 Aktionen für neu erkannte Verbindungen

Abbildung 6.8 zeigt die Hilfsfunktion *Add_ Target*, welche immer dann aufgerufen wird, wenn ein neues Propagierungselement an eine Senke gelangt. Bei Vorwärts-Propagierung ist dann ein Zeigerziel an eine Dereferenzierung gekommen, bei Rückwärts-Propagierung eine Dereferenzierung (oder deren Definition) an eine Zeigerziel-Quelle. Sollte der Propagierungsgraph eine Vereinfachung derart besitzen, dass ein Knoten mehrere ISSA-Senken zusammenfasst (z.B. nach Zyklenkontraktion), so muss Add_Target eine Schleife über all diese Senken ausführen. Dies gilt ebenso für die Propagierungselemente, welche ihrerseits mehrere Zeigerziele (vorwärts) oder Dereferenzierungen (rückwärts) zusammenfassen können. Add_Target entpackt somit die Pakete, die wir für bessere Effizienz geschnürt haben. Die eigentlichen Aktionen werden dann von Add_Data_Target bzw. Add_Call_Target ausgeführt, je nachdem, ob es sich bei einer Dereferenzierung um eine Daten-Dereferenzierung oder einen indirekten Aufruf handelt. Sollte die Dereferenzierung einen Feldoffset zum Zeigerziel addieren, so führen wir diese Addition aus, um zum

```
procedure Add_Target (V : in Prop_Node; O : in Prop_Object) is
begin
   if Forward_Propagation then
      for all T ∈ Targets (O) loop
         for all D ∈ Data_Derefs (V) loop
            Add_Data_Target (D, T);
         end loop;
         for all D ∈ Call_Derefs (V) loop
            Add_Call_Target (D, T);
         end loop;
      end loop;
   else
      for all T ∈ Targets (V) loop
         for all D ∈ Data_Derefs (O) loop
            Add_Data_Target (D, T);
         end loop;
         for all D ∈ Call_Derefs (O) loop
            Add_Call_Target (D, T);
         end loop;
      end loop;
   end if;
end Add_Target;
```

Abbildung 6.8: Aktionen bei einer neu erkannten Verbindung

richtigen Objekt zu gelangen (nicht gezeigt). Allerdings filtern wir dabei auch einige irrtümliche Ziele aus, wenn nämlich durch die Addition die Objekte zur Repräsentation der Felder eines strukturierten Objekts verlassen werden.

Neues Ziel an einer Daten-Dereferenzierung. Wenn ein neues Zeigerziel an einer Daten-Dereferenzierung ankommt, so führt dies zur Erzeugung eines Datenfluss-Knotens. Steht die Dereferenzierung auf der LHS einer Instruktion, so ergibt sich eine indirekte Definition, andernfalls eine indirekte Verwendung. Die nicht gezeigt Hilfsfunktion Add_Data_Target übernimmt diese Aufgabe und legt den neuen Knoten in die Menge New_Nodes für den nächsten Graphaufbau-Schritt.

Neues Ziel an einem indirekten Aufruf. Wenn das Zeigerziel an einem indirekten Aufruf ankommt, so führt dies zur Erweiterung des ICFG und des Aufrufgraphen. Bei einem Funktionszeiger ist das Zeigerziel unmittelbar ein weiteres Ziel des Aufrufes. Soll die Analyse auch objektorientierte Programme unterstützen, so muss das Verfahren hier für einen virtuellen Methodenaufruf anhand des dynamischen Typs des Zeigerziels die aufzurufende(n) Implementierung(en) der angegebenen Methode bestimmen (nicht gezeigt wegen Fokus auf C). In allen Fällen müssen wir formale und aktuelle Parameter verbinden sowie bereits erkannte formale Seiteneffekte des Aufrufzieles an die Aufrufstelle propagieren. Diese Aufgaben erfüllt die Funktion *Add_ Call_ Target* aus Abbildung 6.9.

```
procedure Add_Call_Target (C : Callsite; T : Object)
is
   Target_Func : Routine := T;
begin
   add Target_Func to C.Known_Targets;
   Connect arguments (C) and parameters (Target_Func);
   for all e ∈ Target_Func.Formal_Side_Effects loop
      add corresponding actual side-effect to New_Nodes;
   end loop;
end Add_Call_Target;
```

Abbildung 6.9: Erweiterung des ICFG und des Aufrufgraphen

Filterung der Ziele. Bislang haben wir angenommen, dass jedes ankommende Ziel an einer Dereferenzierung Sinn ergibt und entsprechende Aktionen ausgeführt. Durch die diversen Ungenauigkeiten der Analyse ist dies jedoch nicht immer der Fall. Daher filtern wir in Wahrheit die ankommenden Zeigerziele und behandeln nur diejenigen, welche alle Filter einer Dereferenzierung passieren. Wir unterscheiden daher die Menge *PointsTo* an jedem Knoten des Propagierungs-Graphen (welche ungefiltert ist) von der Menge der gefilterten Ziele der Dereferenzierungen. Die Filter sind dabei im Einzelnen:

1. Die besonderen Objekte für Null, Unknown_Objects und String erzeugen keinen Datenfluss und stellen auch kein Aufrufziel dar. Sie werden daher ausgefiltert. Für eine konservativere Abschätzung im Zusammenhang mit Unknown_Objects kann dieses jedoch in die Menge der potentiellen Zeigerziele umgewandelt werden.

2. Funktionen als Ziele werden an Daten-Dereferenzierung ausgefiltert. Umgekehrt werden an indirekten Aufrufen über Funktionszeiger alle Ziele ausgefiltert, die keine Funktion darstellen. An virtuellen Aufrufen werden alle Ziele ausgefiltert, die kein Objekt vom erwarteten statischen Typ (oder einer abgeleiteten Klasse) sind.

3. In typsicheren Sprachen kann der erwartete Typ an der Dereferenzierung mit dem Typ des Zeigerziels abgeglichen werden. (Dies kann natürlich auch bereits zwischendurch an Typkonversionen usw. geschehen.) Dieser Filter empfiehlt sich insbesondere bei feld-insensitiver Propagierung zur Eliminierung vieler irrtümlicher Ziele. Auch für indirekte Aufrufe ist dies möglich, und hier sogar zu einem gewissen Grad in C.

Kapitel 7

Algorithmen für weitere Aufgaben

Die vorigen beiden Kapitel haben die Algorithmen für die wesentlichen Bestandteile der Kernanalyse (Graphaufbau und Propagierung) beschrieben. In diesem Kapitel vervollständigen wir nun das Bild für eine Umsetzung, indem wir Algorithmen für weitere Teilaufgaben der Analyse vorstellen. Dazu zählen Initialisierung und Finalisierung sowie die Behandlung rekursiver Unterprogramme. Abschließend demonstriert dieses Kapitel auch, wie wir die Analyse so abändern können, dass sie (nur noch) fluss-insensitive Präzision besitzt. Dies beschreibt unsere für den Vergleich im Rahmen der empirischen Evaluation genutzte Umsetzung einer Andersen-Analyse auf der gleichen Codebasis.

7.1 Initialisierung und Finalisierung

Die Initialisierung vor der Fixpunkt-Iteration bereitet die Datenstrukturen vor. So erkennt sie beispielsweise den direkt erkennbaren Teil des Aufrufgraphen und die Zyklen darin. Falls es noch nicht vorab in einer Analyse geschehen ist, sollte hier auch die Bestimmung der Kontrollfluss-Graphen mitsamt den zugehörigen Dominanzinformationen stattfinden.

Zum Abschluss bereinigt unsere kombinierte Analyse in der Finalisierung die Resultate und berechnet nützliche Zusammenfassungen für nachfolgende Analysen. Im Einzelnen erledigt die Finalisierung folgende Schritte:

1. Toten Code entfernen
2. Prunen des IDFG
3. Konstruktion des SCC-DAG zum ICFG, IDFG und Aufrufgraphen
4. Berechnung der topologischen Ordnung und der Postorder-Reihenfolge

Die nächsten Abschnitte erläutern diese Schritte. Sie sind unabhängig von den Algorithmen zur Konstruktion des ISSA-Graphen anwendbar.

7.1.1 Vorbereitungen für nachfolgende Analysen

Nachfolgende Analysen, die den ICFG, den ISSA-Graphen oder den Aufrufgraphen benutzen, können von einer Reihe einfacher Maßnahmen nach Erreichen des Fixpunktes profitieren. Dazu zählen wir:

- Die Konstruktion des SCC-DAG zum jeweiligen Graphen.
- Die Berechnung der topologischen Ordnung des jeweiligen SCC-DAG.
- Die Berechnung von Pre- und Postorder-Nummern für jeden Graphen und den zugehörigen SCC-DAG.

Diese Aufgaben benötigen nur einfache Linearzeit und lassen sich in ein einziges Verfahren kombinieren. Nachfolgende Analysen können dann beispielsweise Vorwärtsprobleme auf dem SCC-DAG des ISSA-Graphen in topologischer Ordnung für bestmögliche Performanz lösen. Für Rückwärtsprobleme steht die Postorder-Reihenfolge zur Verfügung.

Sollen auch intraprozedurale Analysen stattfinden, die das Wissen um schwache Seiteneffekte benötigen, so können wir abschließend auch diese Information mittels Postorder-Traversierung des Aufrufgraphen ermitteln und so lokale Vorgänger für Untereingänge bestimmen. Im Falle der kontext-sensitiven Analyse (vgl. Kapitel 9) kann diese Aussage auch an die Summary-Kanten annotiert werden.

7.1.2 Pruning

Der ISSA-Graph kann Teilgraphen enthalten, die nie zu einer (interessanten) Verwendung führen. Die in solchen Teilgraphen enthaltenen Zuweisungen und von ihnen induzierte Knoten sind damit unnötig. Wir wollen diese Teilgraphen daher identifizieren, dem Anwender melden, und aus dem ISSA-Graphen entfernen. Dieses Entfernen ist für SSA-Formen als *Pruning* bekannt.

Unsere Analyse erzeugt derartige Teilgraphen auf verschiedenen Wegen, wie wir aus den vorausgegangenen Kapiteln erfassen können:

- Im Programm vorhandene, aber nie verwendete Zuweisungen gelangen in den ISSA-Graphen und induzieren weitere Knoten. Auch kann es sein, dass die Zuweisung noch verwendet wird, ein induzierter Knoten jedoch nicht mehr. Eine Quelle nicht verwendeter Zuweisungen stellen z.B. Parameter dar, die nicht benutzt werden.

- Die inkrementelle ISSA-Konstruktion kann vorausgehende Definitionen für eingehende Seiteneffekte und potentiell nicht initialisierte Objekte erzeugen, welche am Ende unnötig sind, weil eine lokale Definition auftauchte.

- Starke Aktualisierungen können dafür sorgen, dass eine vorausgehende Definition keine Verwendungen mehr besitzt. Ursache dafür ist der Vorgang, die Kanten eines Objektgraphen anzupassen, nachdem sich ein neuer Knoten eingeschoben hat.

- Tatsächlich kann eine Definition ohne Verwendung immer noch Nachfolger besitzen (nämlich SSA-Artefakte wie ϕ-Knoten und Seiteneffekt-Knoten), so dass sich auch Zyklen solcher Definitionen bilden können. Im Beispiel aus Abschnitt 4.2 haben wir einen solchen Fall für x gesehen.

Mit rückwärts orientierten Kanten starten wir pro Verwendung, die erhalten bleiben soll, eine Tiefensuche zur Markierung der Knoten, die transitiv zu einer solchen Stelle führen. Anschließend können die unmarkierten Knoten

entfernt werden. Die Verwendungen, die erhalten bleiben sollen, hängen dabei z.B. davon ab, was für anschließende Analysen relevant ist. Beispielsweise nutzen wir folgende Heuristik, die schon einiges abschneidet, aber nicht ganz so viel wie z.B. die klassische Orientierung rein an extern sichtbaren Effekten (und außerdem allein über ISSA-Kanten durchgeführt werden kann):

- Verwendungen von Zeigern in Dereferenzierungen inkl. indirekten Aufrufen (siehe unten für eine Verzögerung diesbezüglich),
- Unterausgänge an Aufrufstellen, die eine externe Funktion aufrufen,
- Verwendungen bei unären oder binären Operatoren.

Verwendungen in Kopierzuweisungen (d.h. die RHS in Anweisungen wie z.B. `x = y` und alle indirekten Verwendungen auf der RHS in Anweisungen wie `x = *p`; ebenso auf Wunsch auch bei unären / binären Operatoren) werden nicht als Startpunkte markiert: Sie überleben nur dann, wenn die LHS überlebt, so dass sich ein transitives Pruning ergibt. Dies kann man auch für Dereferenzierungen der RHS so gestalten, d.h. im Beispiel `x = *p` markiert man die Verwendung von p auch erst dann, wenn die LHS markiert wird. Bei einer indirekten Definition wie `*p = y` kann man ähnlich die Markierung der Verwendung von p so lange hinauszögern, bis eine der indirekten Definitionen an dieser Stelle markiert wird.

Bei vorwärts orientierten Kanten basiert die Erkennung der Teilgraphen, die zu keiner solchen Verwendung führen, auf der Erkennung von allgemeinen Definitionen (und Zyklen aus diesen) ohne Nachfolger. Unser Pruning betrachtet daher den SCC-DAG des berechneten ISSA-Graphen. Wenn wir diesen z.B. für die Zyklenkontraktion in den Propagierungen sowieso explizit konstruiert haben, so können wir dabei die Kanten des SCC-DAG in Rückwärtsorientierung anlegen und damit wie oben vorgehen.

Es bleibt somit nur noch die Frage nach einem Pruning-Verfahren bei Vorwärts-Kanten ohne explizite SCC-DAG-Konstruktion. Dazu integrieren wir das Pruning in Tarjans SCC-Erkennung, welche ihrerseits auf einem Postorder-Durchlauf des IDFG basiert. Dieses Verfahren zeigt Abbildung 7.1.

```
procedure Visit (v) is
begin
   pre[v] := NextPre; NextPre := NextPre + 1;
   head[v] := pre[v];
   InComponent[v] := False;
   Has_Succ_Outside[v] := False;
   push (stack, v);
   for all w ∈ succ(v) loop
      if w unvisited then
         Visit (w);
      end if;
      if w no longer exists then
         continue; -- can happen because of pruning
      end if;
      if InComponent[w] then
         Has_Succ_Outside[v] := True;
      else
         head[v] := min (head[v], head[w]);
      end if;
   end loop;
   if head[v] = pre[v] then -- head of a new SCC
      prunable := True;
      scc_to_prune := ∅;
      loop
         w := pop (stack);
         InComponent[w] := True;
         if Is_Relevant_Use (w) or else Has_Succ_Outside[w] then
            prunable := False;
         elsif prunable then
            add w to scc_to_prune;
         end if;
      until w = v;
      if prunable then -- remove the SCC and incoming edges
         remove all elements of scc_to_prune;
      end if;
   end if;
end Visit;

procedure Tarjan_With_Pruning is
begin
   NextPre := 1;
   for all v ∈ ISSA − Graph loop
      if v unvisited then Visit (v);
      end if;
   end loop;
end;
```

Abbildung 7.1: In Tarjans SCC-Erkennung integriertes Prunen

Erkennt die Traversierung eine Kante zu einem Nachfolger außerhalb der aktuellen SCC, so merken wir uns dies am Knoten. Wenn der Tarjan-Algorithmus den Kopf einer SCC entdeckt, durchläuft er alle Mitglieder der SCC noch einmal. Dabei können wir nun feststellen, ob eines dieser Mitglieder eine im oben diskutierten Sinne interessante Verwendung ist oder die Markierung für eine Kante zu einem außerhalb der SCC befindlichen Nachfolger trägt. Gibt es in der SCC *kein* solches Mitglied, so löschen wir die SCC aus dem Graphen. Darauf muss die Rekursion vorbereitet sein; daher der Test, ob ein Nachfolger nach der Rückkehr aus einer Rekursion überhaupt noch existiert. Unsere tatsächliche Implementierung löscht die zu prunenden Knoten nicht sofort. Stattdessen werden sie nur markiert und beim finalen Speichern des Resultats übergangen. Dies war technisch vorteilhaft, da ansonsten die Menge der Nachfolger verändert wird, während ein Iterator darüber traversiert.

7.1.3 Toten Code entfernen

Den IDFG haben wir mittels Pruning aufgeräumt. Ähnlich können wir aus den Kontrollfluss-Abstraktionen ICFG und Aufrufgraph toten Code entfernen. Dazu ermitteln wir die Grapherreichbarkeit vom Start des Programmes aus. Nicht erreichbare Programmteile sind tot und können verschwinden. Da eine Funktion auch dann tot ist, wenn sie zwar aufgerufen wird, diese Aufrufe aber nur aus totem Code stammen, benutzen wir als Basis den ICFG. Die auf diese Weise als tot erkannten Funktionen entfernen wir anschließend auch aus dem Aufrufgraphen, sollte dieser explizit und separat vorhanden sein.

Die Teilgraphen des IDFG, die für toten Code erzeugt wurden, können als Konsequenz ebenso gelöscht werden. Daher führen wir das Prunen auch erst nach der Eliminierung von totem Code durch. Alternativ kann man die kombinierte Analyse auch nur auf schon definitiv als lebendig eingestuftem Code ausführen, um von vornherein toten Code nicht zu beachten.

Das Entfernen der toten Elemente reduziert das Mengengerüst für nachfolgende Analysen. Außerdem kann die Meldung der toten Elemente an den Benutzer eine nützliche Information für diesen sein.

7.2 Rekursive Unterprogramme

Werden lokale Variablen als Zeigerziele in anderen Unterprogrammen benutzt, so müssen wir sie dort als nicht-lokale Objekte erkennen und entsprechend Seiteneffekte erzeugen. Die bislang beschriebene Implementierung beherrscht dies für azyklische Aufrufgraphen ohne Probleme. Innerhalb von Zyklen ist jedoch Vorsicht geboten.

```
void f (int i, int* p)
{
   int x = 1;
   int* p_value = p:
   if (i = 1) p_value = &x;
   if (i < 3) f (i + 1, p_value);
   if (i = 3) *p = 42;
   printf (x);
   x = 2;
   printf (*p);
}

f (1, &y);
```

Abbildung 7.2: Beispiel zu lokalen Variablen in rekursiven Funktionen

Beispiel 7.2.1 Wir nutzen das Beispiel aus Abbildung 7.2 zur Illustration. Dieses zeigt eine direkt rekursive Funktion f, welche potentiell die lokale Variable x als Zeigerziel an andere Instanzen von sich weitergibt. Betrachten wir speziell den gezeigten Aufruf $f(1, \&y)$. Für die Beschreibung wollen wir dabei die Instanzen von f mit dem jeweiligen Wert von i als Subskript (an f und x) unterscheiden, so dass der Aufruf also die Instanz f_1 hervorruft. Diese ruft ihrerseits f_2 auf und übergibt dabei die Adresse von x_1 als zweites Argument. f_2 wiederum ruft f_3 und reicht hierbei die Adresse von x_1 durch. f_3 schließlich setzt keinen weiteren rekursiven Aufruf ab und setzt $x_1 = 42$. Anschließend wird x_3 (d.h. 1) ausgegeben. Die folgende Zuweisung an x_3 ist ohne Verwendung und die abschließende Ausgabe in f_3 gibt den Wert von x_1 (d.h. 42) aus.

Nun setzen sich die Anweisungen in f_2 nach dem rekursiven Aufruf fort. Es folgt daher die Ausgabe von x_2 (d.h. 1) und abschließend die Ausgabe von x_1 (d.h. 42). Schließlich kehrt auch der rekursive Aufruf in f_1 zurück. Die Ausgabe des lokalen x gibt hier x_1 (also 42) aus, bevor die anschließende Zuweisung dieses überschreibt. Die finale Ausgabe in f_1 schließlich gibt den (hier unbekannten) Wert von y aus.

Die Herausforderung ist nun, eine akzeptable konservative ISSA-Form hierzu zu konstruieren. Das betrifft die Frage, wie viele Analyse-Objekte für x angelegt werden, und wie die Objektgraphen zu diesen Objekten aussehen. □

Wir sehen zwei prinzipielle Ansätze zum Umgang mit einer solchen Situation:

1. Die Verwendung verschiedener Objekte für den lokalen und den nicht-lokalen Fall. Dies stellt auf den ersten Blick eine elegante Modellierung dar, muss aber zusätzliche Objekte einführen und ein paar Besonderheiten beachten.

2. Die Verwendung eines einzelnen Objekts, das sowohl den lokalen als auch den nicht-lokalen Fall repräsentiert. Während dieses Modell die Probleme des ersten Modells vermeidet, müssen wir nun jedoch im Zyklus alle Definitionen des Objekts als schwach ansehen und Seiteneffekte erzeugen.

Auch [Ruf 1995, S. 2, Fußnote 4] kennt die beiden Modelle und berichtet, dass der Ansatz mit zwei Objekten schon auf Cooper zurückgeht. Unsere Implementierung nutzt dabei das Modell, mit nur einem Objekt auszukommen. Dieses Vorgehen beschreiben wir im Folgenden und gehen anschließend kurz auf die Alternative ein.

7.2.1 Verwendung eines einzigen Objekts

Unsere Implementierung würde für das Beispiel nur ein Analyse-Objekt zu x anlegen. Dieses muss nun alle x_i in sich vereinen, insbesondere also jeweils den lokalen und den nicht-lokalen Fall abdecken. Die Datenflüsse der beiden

Fälle verschmelzen somit. Abbildung 7.3 zeigt, wie in dieser Variante der Objektgraph von x zum Beispiel aussieht. Hierbei haben wir interprozedurale Kanten wieder gestrichelt gezeichnet.

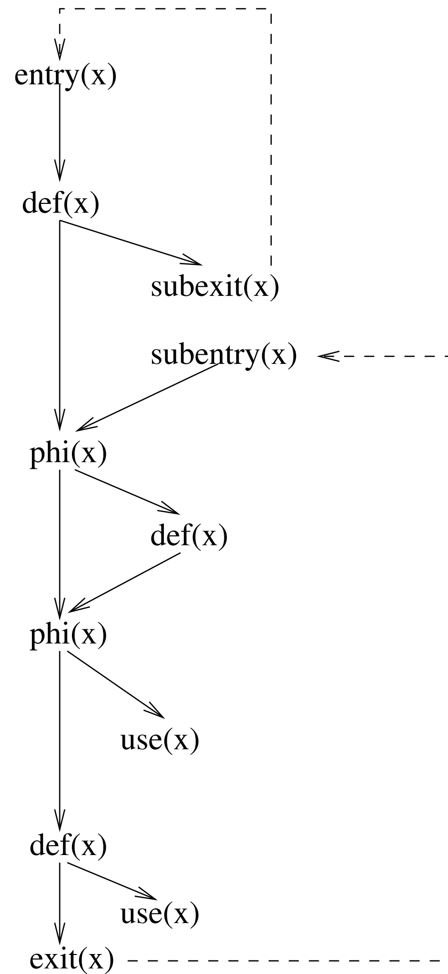


Abbildung 7.3: Objektgraph von x, wenn nur ein Objekt benutzt wird

Für eine konservative Abdeckung müssen wir nun selbst direkte Definitionen auf das lokale Objekt als schwache Aktualisierung behandeln. Die Abbildung zeigt dies, indem der Objektgraph von x eine eingehende Kante für die Definition zu $x = 2$ aufweist. Damit gelingt es, dass (unter anderem) der richtige Datenfluss von der indirekten Definition (aus f_3) zum abschließenden `printf (*p)` in f_2 gelangt.

Diese Abschwächung der direkten Definitionen können wir natürlich sofort bei der Erkennung von Aufrufgraph-Zyklen für alle darin befindlichen lokalen Variablen durchführen, die ein potentielles Zeigerziel sind. Wir können dies aber auch noch hinauszögern, wobei wir mehrere Optionen sehen:

1. Sobald das Objekt sein Unterprogramm zu einem Eingang verlässt.

2. Sobald das Objekt über einen Eingang in sein Unterprogramm gelangt oder ein Untereingang dafür entsteht.

3. Erst bei der Dereferenzierung.

Die ersten beiden Punkte sind leicht in der Propagierung zu erkennen. Beim letzten Punkt stellt sich die Frage, wie an der Dereferenzierung entschieden werden kann, ob das lokale oder das nicht-lokale Objekt als Ziel vorliegt. Sicherlich kann man auch ganz auf diese Entscheidung verzichten und dabei in Kauf nehmen, dass bei ausschließlich lokalen Dereferenzierungen unnötigerweise abgeschwächt wird. Die Erweiterung zur Erkennung kontextabhängiger Zeigerziele liefert uns später die Infrastruktur gratis, um bei der Dereferenzierung den nicht-lokalen Fall zu erkennen. Ohne diese Erweiterung kann z.B. die Propagierung nach wie vor Fall (2) oben erkennen (Objekt gelangt von außen ins Unterprogramm), dann aber nur das Objekt markieren. Die Dereferenzierung eines markierten Objekts zeigt dann den Zeitpunkt für die Abschwächung an. Dieses Modell verwendet unsere Implementierung.

Die Abschwächung selbst betrachtet schlicht die direkten Definitionen im Unterprogramm, zu dem die lokale Variable gehört. Zu jeder dieser Definitionen sucht sie dann die gültige Definition, wie wir es für den Graphaufbau allgemein besprochen haben, und erstellt damit die Kante.

Die Erkennung von Zyklen im Aufrufgraphen starten wir am Ende der Propagierung, wenn eine neue Aufrufkante zwischen verschiedenen SCCs gefunden wurde (und zwingend in Iteration 1). Als Algorithmus setzen wir dabei Tarjans SCC-Erkennung ein.

7.2.2 Verwendung verschiedener Objekte

Alternativ zum beschriebenen Vorgehen kann man auch je ein Objekt für den lokalen und den nicht-lokalen Fall verwenden. (Mehrere Objekte für die nicht-lokalen Fälle erscheint einerseits schwer zu realisieren, andererseits kaum eine Verbesserung zu versprechen.)

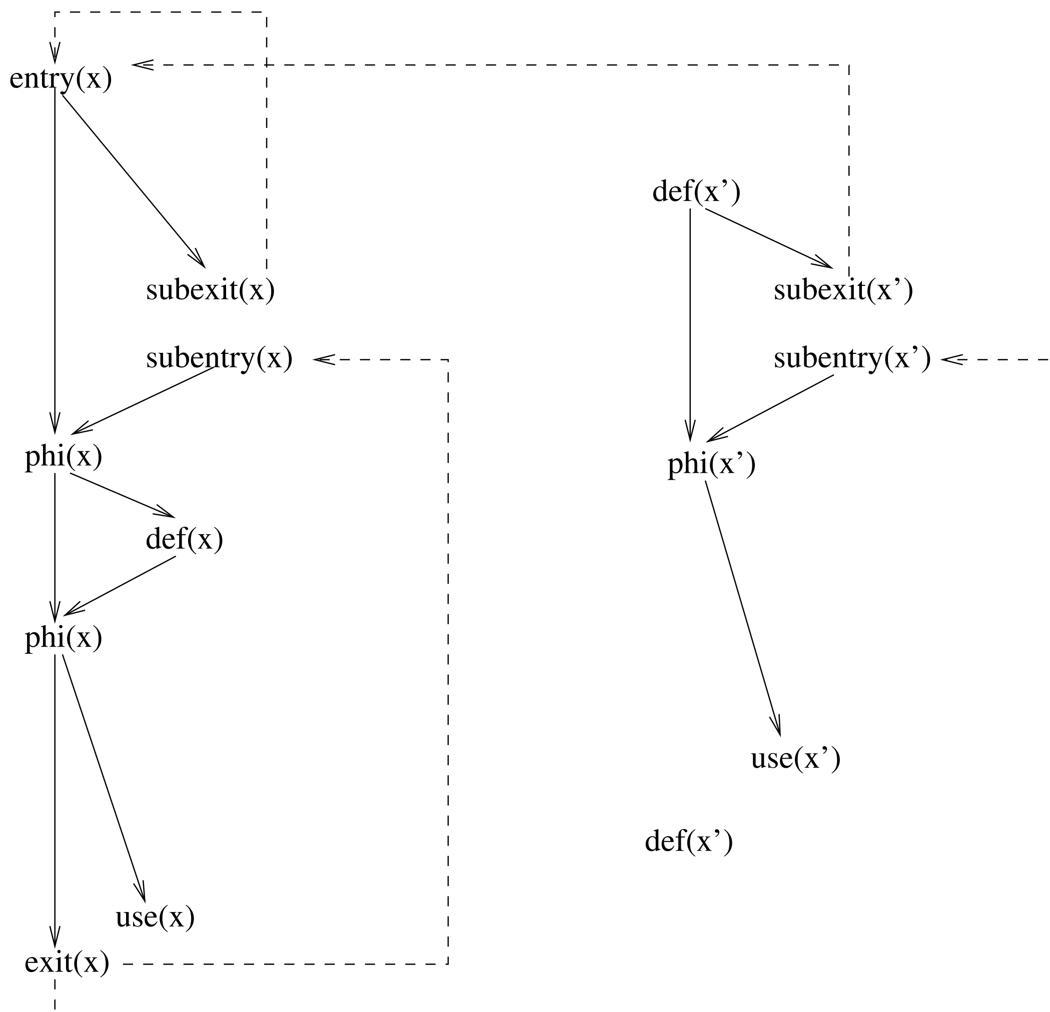


Abbildung 7.4: Objektgraph von x (nicht-lokal) und x' (lokal), wenn zwei Objekte benutzt werden

Abbildung 7.4 zeigt die beiden zugehörigen Objektgraphen. Hier haben wir das Analyse-Objekt fürs lokale x mit x' bezeichnet und x für das nicht-

lokale Analyse-Objekt verwendet. Wir erkennen, dass diese Variante eine höhere Präzision bietet. Das Beispiel verdeutlicht aber auch ein Problem dieser Variante: Nun gibt es an einer einzigen Aufrufstelle mehr als einen Unterausgang zum Eingang von x. Gleiches gilt für die Untereingänge.

Während diese Variante also lokal den Datenfluss des lokalen und des nicht-lokalen Falles trennen kann, ist die Besonderheit zu beachten, dass es keine injektive Beziehung zwischen formalen und aktuellen Seiteneffekten pro Objekt an einer Aufrufstelle gibt. Dies muss im Algorithmus (und ggf. in den Datenstrukturen) bedacht werden. Darüber hinaus muss die Propagierung ebenfalls den Wechsel zwischen den Objekten realisieren: Verlässt das lokale Objekt x' sein Unterprogramm als aktuelles Ziel (hier von p_value) entlang einer Datenfluss-Kante zu einem Eingang, so muss es ab dort als das nicht-lokale Objekt propagiert werden. Der umgekehrte Weg muss dabei wieder in beide Objekte übersetzen. Und schließlich muss eine Implementierung dieser Variante auch beachten, dass dadurch die Zahl der Objekte dynamisch ansteigt. Wegen dieser Besonderheiten haben wir es vorgezogen, die Variante mit nur einem Objekt zu implementieren.

7.3 Fluss-insensitive Variante

Dieser Abschnitt schildert die Umsetzung der fluss-insensitiven Variante, d.h. die Reduktion auf Andersens Verfahren. Dabei können wir lokale und globale Fluss-Insensitivität unterscheiden: Assoziieren wir mit einem Objekt global genau ein Datenfluss-Element, so haben wir *globale Fluss-Insensitivität.* Assoziieren wir dagegen pro Unterprogramm genau ein Element mit einem Objekt, so haben wir die präzisere *lokale Fluss-Insensitivität* (vgl. Burke u. a. [1994]; Hind und Pioli [1998]). Zum Vergleich mit Andersens Originalformulierung haben wir die gröbere global fluss-insensitive Variante umgesetzt.

7.3.1 Veränderte Darstellung

Zur global fluss-insensitiven Propagierung von Zeigerzielen benötigt man anstelle des ISSA-Graphen den Constraint-Graphen. Wie wir in Abschnitt 4.4

beschrieben haben, können wir den Constraint-Graphen dabei auch als grob überschätzten IDFG auffassen: Die Analyse-Objekte bilden die Knoten dieses IDFG, und die Kanten (Constraints) sind im Grunde Überschätzungen der Datenfluss-Beziehungen zwischen diesen.

Diese Überlegungen skizzieren die Datenstrukturen des fluss-insensitiven IDFG: Ein Array, indiziert mit der Nummer eines Objekts, liefert uns pro Objekt die ausgehenden Kanten des IDFG sowie die Programmstellen, an denen Zeigerziele des Objekts eingesetzt werden müssen. Beide Mengen wachsen im Laufe der Analyse monoton.

Bei den Dereferenzierungen benötigen wir keinen Test, ob eine solche Programmstelle bereits einem Objekt zugeordnet wurde: Dies müssen wir sowieso über alle Objekte hinweg an einer Stelle abprüfen, da auch ein anderes Objekt zum gleichen Ziel an gleicher Stelle führen kann. Ein indirekter Knoten listet daher immer alle Objekte auf, die er bereits repräsentiert. Es empfiehlt sich daher ein dynamisch wachsendes Array oder eine einfach verkettete Liste als kompakte und effiziente Datenstruktur für die Menge der Dereferenzierungen eines Objekts.

Die ausgehenden Datenfluss-Kanten jedoch müssen wir auf Enthaltensein überprüfen, damit sie nicht doppelt eingefügt werden. Zugleich kann diese Menge stark anwachsen, insbesondere für Heapobjekte und globale Variablen. Eine mögliche Datenstruktur dafür ist ein Bitvektor, der für jedes Objekt das zur Objektnummer gehörende Bit bei direkter Erreichbarkeit gesetzt hat. Diese Struktur ist sehr effizient und erlaubt zusätzlich, ggf. auch die transitive Erreichbarkeit aufzunehmen.

Wir können erneut eine erschöpfende und eine inkrementelle Propagierung realisieren, vorwärts oder rückwärts. Für die inkrementelle Variante müssen wir an jedem Objekt noch die Zeigerziele auflisten, so dass hier z.B. ein weiterer Bitvektor pro Objekt benötigt wird (diesmal nur in der Größe, die alle potentiellen Zeigerziele erfasst).

Da wir den Kontrollfluss unbeachtet lassen, benötigen wir keinen ICFG. Der Aufrufgraph wird jedoch mitgeführt, um Zyklen darin zu erkennen und die Ziele indirekter Aufrufe zu vermerken.

7.3.2 Änderungen am Verfahren

Initialisierung. Der initiale IDFG besteht aus allen Objekten, welche im Programm vorhanden sind. Bei kontext-insensitiver Heapmodellierung und Feld-Insensitivität ändert sich die Zahl der Objekte auch nicht mehr, so dass wir bereits alle Knoten des IDFG nach der Initialisierung kennen. Konzeptionell ergänzt die Initialisierung wie im fluss-sensitiven Fall die Zuweisungsbrücken, auch für Parameterübergaben, als Kanten des IDFG.

Veränderter Graphaufbau. Die Initialisierung und Propagierung melden dem Graphaufbau weiterhin die neu erkannten Datenfluss-Elemente als Definitionen und Verwendungen. Für eine Verwendung prüft der Graphaufbau, ob eine neue Zuweisungsbrücke vorliegt. Falls ja, so wird diese dem IDFG für alle Objekte der linken Seite konzeptionell hinzugefügt und der Propagierung als neue Kante gemeldet (zwischen den beteiligten Objekten, wobei Schleifen vermieden werden). Handelt es sich bei der Verwendung um eine Dereferenzierung, so merken wir uns diese am Objekt und betrachten diese Verbindung als neue Kante. Abbildung 7.5 illustriert das Vorgehen für eine neue Verwendung.

```
procedure New_FI_Use (N : Node) is
begin
    if Is_RHS_Of_Copy (N) then
        LHS := Get_LHS_Of_Copy (N);
        Add_Edge (N.Object, LHS.Object);
    elsif Is_Dereference (N) then
        Add N to Derefs (N.Object);
        Add_Edge (N.Object, N);
    end if;
end New_FI_Use;
```

Abbildung 7.5: Fluss-insensitive Aktionen bei einer neuen Verwendung

Die Aktionen für eine neue Definition sind ähnlich. So wird ebenfalls auf neue Zuweisungsbrücken geprüft. Dafür entfällt der Test auf Kanten zu Dereferenzierungen, diese wurden bereits bei der Erkennung derselben ergänzt. Wie im fluss-sensitiven Fall beachten wir wieder die Möglichkeit, dass neue

initiale Zeigerziele entstehen können. Neben eventuellen Zuweisungsbrücken melden wir dies der Propagierung. Abbildung 7.6 zeigt die Aktionen für eine neue Definition.

```
procedure New_FI_Def (N : Node) is
begin
   if Is_LHS_Of_Copy (N) then
      RHS := Get_RHS_Of_Copy (N);
      Add_Edge (RHS.Object, N.Object);
   end if;
   if Has_Pointer_Literals (N) then
      Add_Pointer_Literals (N);
   end if;
end New_FI_Def;
```

Abbildung 7.6: Fluss-insensitive Aktionen bei einer neuen Definition

Propagierung. Die Propagierung kann erfreulicherweise ohne Veränderung übernommen werden: Es gilt weiterhin, das Grapherreichbarkeits-Problem auf dem IDFG zu lösen. Gelangt ein Zeigerziel auf diese Weise zu einer Dereferenzierung, so wird dem Graphaufbau eine neue Definition oder Verwendung gemeldet bzw. im Falle eines indirekten Aufrufs der Aufrufgraph erweitert. Dabei werden auch die Zuweisungsbrücken für Parameterübergabe angelegt.

Finalisierung. Damit wir die Resultate vergleichen können, rufen wir nach Erreichen des Fixpunktes einmalig den Graphaufbau der fluss-sensitiven Variante mit allen erkannten Definitionen und Verwendungen auf. Das propagiert die Seiteneffekte und erstellt die ISSA-Form, welche abschließend gepruned werden kann. Als Konsistenztest können wir sogar die vollständige fluss-sensitive Variante mit den fluss-insensitiven Resultaten als initiale IDFG-Knoten ausführen: Nach einer Iteration muss da bereits Schluss sein, die Propagierung kann keine neuen Ziele erkennen.

7.3.3 Alternatives Vorgehen

Die bisher beschriebene Umsetzung entspricht dem klassischen Vorgehen: Erst schließen wir die Zeigeranalyse mit konservativen Resultaten ab, bevor wir mit der Konstruktion des ISSA-Graphen beginnen. Diese Version dient daher für den Vergleich zum klassischen Vorgehen.

Alternativ dazu können wir jedoch die fluss-sensitive Implementierung auch nur minimal abwandeln: Der Graphaufbau bestimmt wie im fluss-sensitiven Fall den ISSA-Graphen. Zusätzlich bestimmt er die Beziehungen auf der Ebene der Objekte als Abstraktion davon. Die Propagierung erfolgt nun auf dieser Abstraktion.

Die Resultate der beiden Analysen unterscheiden sich nicht. Die Laufzeit dürfte in der klassischen Formulierung etwas besser sein, da dort keine Korrekturen am ISSA-Graphen notwendig werden. Auch der Speicherbedarf ist dort geringer, weil die Beziehungen zwischen den Objekten gelöscht werden können, sobald damit die möglichen Dereferenzierungsziele bestimmt sind. Daher präferieren wir die klassische Formulierung, auch wenn die Alternative näher an der fluss-sensitiven Version liegt.

Kapitel 8

Erkennung starker Aktualisierungen

In diesem Kapitel stellen wir eine Technik vor, mit der die Kernanalyse indirekte starke Aktualisierungen beherrscht. Damit kann die Analyse dann eine zusätzliche Quelle für erhöhte Präzision gegenüber fluss-insensitiven Zeigeranalysen ausschöpfen.

Das Kapitel stellt zuerst die Problematik sowie unsere Idee zur Lösung dieser Problematik vor, bevor es auf die Modifikation der algorithmischen Umsetzung eingeht.

8.1 Problematik

Die Kernanalyse ging bislang bei allen indirekten Definitionen konservativ von einer schwachen Aktualisierung aus. Dies ist eine pessimistische Überschätzung, falls der dereferenzierte Zeiger auf genau ein Ziel verweist, welches zudem noch mit der Definition vollständig modifiziert wird. Die Schwierigkeit für unsere Analyse bei der Erkennung solcher Situationen besteht darin, dass sie nicht weiß, ob im weiteren Verlauf der Analyse noch weitere Ziele für den dereferenzierten Zeiger ankommen. Hierzu müsste bekannt sein, welche Zeiger bereits konservativ überschätzt vorliegen. Dieses Wissen ist in der Kernanalyse aber erst mit Erreichen des Fixpunktes vorhanden.

Beispielsweise geht die Kernanalyse in folgendem Code zunächst davon aus, dass die zweite Anweisung für x irrelevant ist – es sei denn, wir haben bereits definitiv $x \in Targets(p)$ ermittelt:

```
x = 1;
*p = 2;
... = x;
```

Falls wir später doch noch $x \in Targets(p)$ ermitteln, so behandelt die Kernanalyse ab dann die zweite Anweisung als *zusätzliche* Definition für x. Damit handelt es sich bei der zweiten Anweisung um eine schwache Aktualisierungen (engl. *weak update*). Dieses Resultat ist nur dann zu pessimistisch, wenn p auf x *und nur* auf x zeigt, wobei es sich zusätzlich noch um eine vollständige Aktualisierung handeln muss. Dann wäre die Anweisung eine starke Aktualisierung (engl. *strong update*).

8.1.1 Ursachen für schwache Aktualisierungen

Zunächst wollen wir die Ursachen für schwache Aktualisierungen besser verstehen, um daraufhin gezielt nach Lösungen suchen zu können. Wie bereits in Abschnitt 3.1.2 angerissen, verwendet die Kernanalyse in folgenden Situationen schwache Aktualisierungen:

Indirekte Definitionen Die Kernanalyse behandelt indirekte Definitionen generell als schwache Aktualisierungen.

Modifikation von Feldern Die Feld-insensitive Analyse behandelt Definitionen von Feldern strukturierter Objekte als schwache Aktualisierungen.

Modifikation von Arrays Ein Array modelliert die Analyse wie ein einzelnes Element. Die fehlende Unterscheidung der Elemente führt dazu, dass stets eine schwache Aktualisierung benutzt werden muss.

Rekursive Funktionen Im rekursiven Fall kann das Analyse-Objekt für eine lokale Variable sowohl die lokale Instanz der Variable bezeichnen als auch nicht-lokale Instanzen, die als Zeigerziel zugreifbar sind.

Wir sehen, dass schwache Aktualisierungen einerseits eine Eigenschaft bestimmter Objekte sind (Arrays, strukturierte Objekte, lokale Variablen rekursiver Unterprogramme), andererseits aber auch von der konkreten Definition abhängen.

Für Feldzugriffe besteht die Lösung darin, die feld-sensitive Ausprägung der Analyse zu benutzen. Für rekursive Funktionen könnte man die präzisere Modellierung einsetzen. Damit bleibt als Hauptthema die Frage, wie indirekte starke Aktualisierungen erkannt und berücksichtigt werden können. Dies führt uns auf eine weitere Differenzierung der schwachen Aktualisierungen für indirekten Definitionen:

Mehrere Ziele Erkennen wir mehr als ein Zeigerziel, so haben wir für jedes davon nur eine potentielle Definition. Damit können wir keine starke Aktualisierung annehmen.

C-Strings, Felder und Arrays Ist das einzige Ziel der indirekten Definition ein C-String, so handelt es sich faktisch um ein Array. Entsprechend können wir für C-Strings wie auch für Arrays keine starken Aktualisierungen annehmen. Für Feld-Modifikationen ist die feld-insensitive Ausprägung wie im Falle direkter Definitionen betroffen.

Rekursive Funktionen Hier gelten die Argumente wie im Falle direkter Definitionen.

Heapobjekte Heapobjekte stellen vor allem in C ein großes Problem dar: Sie sind bei der Allokation untypisiert, ein Analyse-Heapobjekt kann für viele Heapobjekte (auch unterschiedlichen Typs) stehen, und in der Regel sind sie strukturierte Objekte. Insbesondere da das gleiche Analyse-Objekt verschiedenen parallel existierenden realen Objekten entsprechen kann, müssen wir konservativ schwache Aktualisierungen annehmen.

Andere indirekte Definitionen mit einem Ziel Die verbleibenden indirekten Definitionen mit genau einem Ziel könnten wir prinzipiell als starke Aktualisierungen unterstützen. Der konzeptionelle Ansatz der Analyse aus Abschnitt 4 muss dafür jedoch angepasst werden.

Neben den direkten und indirekten Definitionen stellt sich die Frage nach einer starken oder schwachen Aktualisierung auch für Seiteneffekte als induzierte Knoten. Außerdem kann ein indirekter Aufruf einen Einfluss haben, da die dort möglicherweise vorhandene Unsicherheit über das Aufrufziel jeglichen starken Seiteneffekt abschwächt.

8.1.2 Arten von starken Aktualisierungen

Die Betrachtung einer einzelnen indirekten Definition wie eben kann *einfache* starke oder schwache Aktualisierungen ergeben. Darüber hinaus existieren jedoch auch *zusammengesetzte* (komplexe) starke oder schwache Aktualisierungen. Dies ergeben sich aus der Betrachtung von Teilgraphen des IDFG. Ein Beispiel:

```
x = 1;
if (...) *p = 2;
else *p = 3;
... = x;
```

Jede indirekte Definition in den beiden Zweigen des *if*s kann für sich (einfach) stark oder schwach sein. Aufgrund der Kontrollfluss-Überschätzung müssen wir jedoch einfache starke oder schwache Aktualisierungen an Konfluenzpunkten zusammenführen und konservativ die Ausführung des Zweiges mit der schwächsten Aktualisierung annehmen. Ist also eine der beiden indirekten Definitionen schwach für x oder gar keine Definition für x, so erreicht die direkte Definition die Verwendung nach dem *if*.

Nur in dem Falle, dass auf allen Zweigen eine (direkte oder indirekte) starke Aktualisierung vorliegt, handelt es sich um eine zusammengesetzte starke Aktualisierung für den Teilgraphen.

Wir fassen unsere Betrachtung in den folgenden Definitionen zusammen:

Definition 8.1.1 Eine Definition d ist eine *einfache starke Aktualisierung* (engl. *simple strong update*, SSU), falls d genau ein reales Objekt modifiziert und das zugehörige Analyse-Objekt dabei vollständig modifiziert. Wir sprechen dann von einer starken Aktualisierung für das modifizierte Objekt.

Andernfalls handelt es sich um eine einfache schwache Aktualisierung (für alle modifizierten Objekte).

Die Verallgemeinerung auf Teilgraphen statt einzelner Definitionen:

Definition 8.1.2 Ein zusammenhängender Teilgraph G des ICFG ist eine *zusammengesetzte starke Aktualisierung* (engl. *composite strong update*, CSU) für ein Objekt x, falls die folgenden Bedingungen erfüllt sind:

1. G besitzt genau einen Eingang s und genau einen Ausgang t
2. Kein Knoten $v \in G$ außer $v = s$ dominiert t strikt
3. s hat mehrere Nachfolger
4. t hat mehrere Vorgänger
5. auf allen Pfaden von s nach t findet eine SSU für x statt in einem von s und t verschiedenen Knoten

Dieses Konzept entspricht dem *cover-value* aus [Plödereder 1980, S. 8-33].

Hier und im Folgenden gehen wir dabei von einem ICFG auf der Ebene einzelner Anweisungen oder Ausdrücke aus, nicht von einer Grundblock-basierten Darstellung. Da G nur den einen Eingang s besitzt, gilt für eine CSU $G(s,t)$ stets s dom t. Die Definition beschränkt dabei die Teilgraphen auf sinnvolle Weise: Die Beschränkung auf genau einen Ein- und Ausgang stellt sicher, dass wir abgeschlossene Einheiten betrachten. Gäbe es in G einen weiteren Knoten v außer s mit v dom t, so könnten wir $G(s,t)$ zerlegen in die beiden kleineren Graphen $G_1(s,v)$ und $G_2(v,t)$ und diese jeweils unabhängig voneinander betrachten. Da Pfade sehr einfach über eine darin befindliche SSU abgedeckt werden können, fordern wir vom Ein- und Ausgang jeweils einen nichttrivialen Ausgangs- bzw. Eingangsgrad. G besitzt damit mehrere Pfade von s nach t. Verschiedene Pfade mögen dabei weitere gemeinsame Knoten haben, aber außer s und t liegt kein Knoten auf allen Pfaden. Damit muss es auch insgesamt mehr als eine SSU in G geben. Der Ausgang t liegt dabei in der iterierten Dominanzgrenze von mindestens einer dieser SSU; dies werden wir später ausnutzen.

Die Definitionen einer einfachen starken Aktualisierung spricht bewusst von genau einem *realen* Objekt im Unterschied zu einem *Analyse*-Objekt. Deutlich wird dieser Unterschied vor allem bei Heapobjekten und deren typischen Modellierung in der Analyse: Da werden mehrere an einer Allokationsstelle allokierten Objekte zu einem Analyse-Objekt zusammengefasst. Für solche Objektgruppen können wir nicht ohne Weiteres eine starke Aktualisierung annehmen.

Es ist klar, dass indirekte Definitionen mit bereits mehr als einem erkannten Zeigerziel keine starken Aktualisierungen sein können, da die Menge an Zeigerzielen höchstens wächst. Wir müssen daher nur indirekte Definitionen mit bislang einem oder keinem Zeigerziel als potentielle starke Aktualisierungen betrachten. Für die weitere Betrachtung benennen wir diese beiden Fälle, weil sie durchaus verschiedene Techniken erfordern:

Definition 8.1.3 Eine indirekte Definition ohne derzeit bekannte Zeigerziele nennen wir eine *0-Definition*. Eine indirekte Definition mit genau einem derzeit bekannten Zeigerziel nennen wir eine *1-Definition*.

Der Begriff einer starken Aktualisierung (*strong update*) stammt wohl zuerst aus Chase u. a. [1990]. Sie präsentieren in ihrer Arbeit auch einige Kriterien zur Erkennung derselben.

8.1.3 Wirkungsbereich einer starken Aktualisierung

Wir wollen nun noch klären, welchen Einfluss eine starke Aktualisierung hat. Ihre unmittelbare Wirkung soll sein, dass keine vorausgehende Definition des modifizierten Objekts an nachfolgende Verwendungen gelangt. Hier müssen wir die Begriffe *vorausgehend* und *nachfolgend* präzisieren, da die zeitlichen Abläufe aller möglichen Ausführungen in der statischen Analyse verschmolzen und nicht mehr einfach linear sind.

Im Falle der vorausgehenden Definitionen ist dies simpel, da eine starke Aktualisierung schlicht alle ankommenden Definitionen löscht. Bei den nachfolgenden Verwendungen müssen wir den Wirkungsbereich einer starken Aktualisierung auf diejenigen Verwendungen begrenzen, die sicherlich in

allen Ausführungen später ausgeführt werden. Technisch führt uns das auf die *Dominanz* als Hilfsmittel. Ist s eine starke Aktualisierung, d eine gültige Definition vor s (ausgedrückt durch $reaches(d, s)$) und u eine fragliche nachfolgende Verwendung, so muss für eine lokale Wirkung von s gelten: s dom u. Dann wurde u sicher in allen Ausführungen nach s ausgeführt. Um derartige Reihenfolgen auch global betrachten zu können, benötigen wir statt der typischen Dominanz die *interprozedurale Dominanz*.

Definition 8.1.4 Eine einfache starke Aktualisierung s verhindert den Datenfluss von einer Definition d zu einer Verwendung u, falls das folgende Prädikat erfüllt ist:

$$\begin{aligned} blocks(s, d \rightarrow u) \quad ::= \quad & object(s) = object(d) = object(u) \\ & \wedge\, reaches(d, s) \wedge s \text{ dom } u \end{aligned}$$

Eine zusammengesetzte starke Aktualisierung G mit Eingang s und Ausgang t verhindert den Datenfluss von einer Definition d zu einer Verwendung u, falls das folgende Prädikat erfüllt ist:

$$\begin{aligned} blocks(G(s,t), d \rightarrow u) \quad ::= \quad & object(G(s,t)) = object(d) = object(u) \\ & \wedge\, reaches(d, s) \wedge t \text{ dom } u \end{aligned}$$

Innerhalb einer CSU können wir die Dominanz jedoch nicht anwenden, um die CSU-Bedingung selbst zu überprüfen. Hierfür werden wir in Kürze einen anderen Ansatz aufzeigen.

8.2 Lösungsansatz

In diesem Abschnitt beschreiben wir die Überlegungen zu unserer Erweiterung der Kernanalyse, die indirekte starke Aktualisierungen erkennt und respektiert. Diese Erweiterung erhöht damit die Präzision. Sie unterbindet die Propagierung von Zeigerzielen von d nach u, falls es eine einfache starke Aktualisierung s gibt mit $blocks(s, d \rightarrow u)$ oder eine zusammengesetzte starke Aktualisierung $G(s, t)$ mit $blocks(G(s, t), d \rightarrow u)$.

Für einfache direkte starke Aktualisierungen beherrscht bereits die Kernanalyse diese Semantik. Für die Erweiterung muss die Analyse jedoch die potentiellen indirekten starken Aktualisierungen ausfindig machen und respektieren. Wir beschreiben nun zunächst das Vorgehen für einfache starke Aktualisierungen; Abschnitt 8.2.3 ergänzt abschließend die zusammengesetzten Fälle.

8.2.1 Potentielle einfache starke Aktualisierungen

Die Ermittlung der potentiellen, einfachen, indirekten starken Aktualisierungen ist einfach anhand der Zahl bisher bekannter Zeigerziele. Damit können wir die indirekten Definitionen als 0-Definition, 1-Definition oder uninteressant für diese Erweiterung einstufen. Eine weitere Eingrenzung ist möglich, falls die indirekte Definition prinzipiell keine starken Aktualisierungen zulässt. Z.B. bei einer feld-insensitiven Analyse ist dies bei Feldzugriffen der Fall.

Die Grundidee unseres Umgangs mit den so eingegrenzten *potentiellen* starken Aktualisierungen besteht darin, sie wie *tatsächliche* starke Aktualisierungen zu behandeln, bis das Gegenteil bewiesen ist. Die potentielle starke Aktualisierung blockiert damit betroffene Datenfluss-Kanten erst einmal, verhindert also eine Propagierung von Zeigerzielen entlang dieser Verbindungen.

Dieses Vorgehen ist in dem Sinne optimistisch, dass wir den Fall mit höherer Präzision annehmen und die Menge der zu propagierenden Fakten erst einmal unterschätzen. Wir vergrößern damit die Unterschätzung noch, die in der Kernanalyse aufgrund noch unbekannter Datenfluss-Beziehungen oder Zeigerziele vorhanden ist. Wie wir zeigen werden, führt auch diese größere Unterschätzung am Ende zu konservativen Resultaten.

8.2.2 Potentiell betroffene Datenfluss-Kanten

In Abschnitt 8.1.3 haben wir mit den *blocks*-Prädikaten formalisiert, welche Datenfluss-Kanten von einer tatsächlichen starken Aktualisierung betroffen sind. Dies übertragen wir nun auf potentielle starke Aktualisierungen, d.h. wir beschreiben die potentiell betroffenen Datenfluss-Kanten.

Für eine 1-Definition ist bereits ein Objekt bekannt; damit lässt sich Definition 8.1.4 direkt anwenden. Für eine 0-Definition jedoch ist dies nicht der Fall. Hier müssen wir daher zunächst klären, welche Objekte potentiell betroffen sein können, um darauf aufbauend dann die potentiell betroffenen Kanten zu bestimmen.

8.2.2.1 Potentiell betroffene Objekte

Wie uns der Korrektheitsbeweis später noch darlegen wird, könnten wir prinzipiell alle Objekte als potentiell betroffen einstufen. Dies erhöht jedoch unnötig das Mengengerüst im Verfahren, das wir zum Umgang mit 0-Definitionen noch beschreiben werden. Eine Beschränkung der potentiell betroffenen Objekte ist relativ leicht und deutlich möglich über die folgenden Bedingungen:

1. Das Objekt muss ein potentielles Zeigerziel sein, denn ansonsten wird s sicher keine Definition dafür hervorbringen.

2. Das Objekt muss prinzipiell starke Aktualisierungen erlauben, denn ansonsten ist die Blockade unnötig.

In typsicheren Sprachen kann zudem der statisch bekannte Typ der Zeigerziele an s genutzt werden, um Objekte mit unpassendem Typ von vornherein als nicht von s betroffen einzustufen.

Die potentiellen Zeigerziele hat bereits die Kernanalyse ermittelt, um z.B. die Maximalgröße von Zeigerziel-Mengen zu kennen. Der Test, ob ein Objekt Bedingung (1) erfüllt, ist damit ohne Mehrkosten möglich; ebenso das Iterieren über alle potentiellen Zeigerziele.

Die notwendigen Informationen für Bedingung (2) kennen wir ebenfalls bereits weitgehend, da die Kernanalyse für direkte Definitionen auch schon die Frage entscheiden musste, ob eine schwache oder starke Aktualisierung vorliegt. So erlauben z.B. lokale Variablen rekursiver Programme in unserer Umsetzung keine starke Aktualisierung, falls ihre Adresse genommen wird (vgl. Abschnitt 7.2). Ergänzen müssen wir hier nur die Betrachtung von Objekten, die ausschließlich in indirekten (nicht in direkten) Definitionen auftauchen. Das sind Heapobjekte und anonyme Stackobjekte (z.B. C-Stringliterale und Arrays).

Ein C-String entspricht einem Array, so dass wir seine Elemente wie bei allen Arrays nicht unterscheiden. Die Analyse darf daher generell keine starken Aktualisierungen für diese Objektsorte annehmen.

Für Heapobjekte haben wir ein ähnliches Problem wie für lokale Variablen rekursiver Unterprogramme: Das gleiche Analyse-Objekt kann mehreren zugleich lebendigen realen Heapobjekten entsprechen. Diese Problematik wurde z.B. auch schon von Chatterjee [2000] erwähnt. Das folgende Beispiel soll dies verdeutlichen:

```
p = malloc_wrapper (...);
q = malloc_wrapper (...);
*q = 1;
*p = 2;
print (*q);
```

Hier zeigen (bei der Modellierung wie in Kapitel 3 besprochen) sowohl p als auch q auf das gleiche Analyse-Objekt o; wir haben damit zwei sukzessive indirekte Definitionen für o. Wir müssen aber sicherstellen, dass die erste dieser Definitionen zur abschließenden Ausgabe gelangt. Ohne weitere Informationen (die z.B. eine Same-Value-Eigenschaft etablieren könnten) und bei der angesprochenen Modellierung mit nur einem Objekt pro Allokationsstelle nutzen wir die Strategie, generell nur schwache Aktualisierungen für Heapobjekte zuzulassen. Künftige Arbeiten können hier eine höhere Präzision anstreben, um mehr Möglichkeiten für indirekte starke Aktualisierungen zu schaffen.

8.2.2.2 Wirkungsbereich von 0-Definitionen

Für 0-Definitionen können wir mit diesen Überlegungen den Wirkungsbereich nun wie folgt begrenzen:

Definition 8.2.1 Eine 0-Definition s, welche eine potentielle starke Aktualisierung darstellt, verhindert den Datenfluss von einer Definition d zu

einer Verwendung u, falls das folgende Prädikat erfüllt ist:

$$\begin{aligned}\overline{blocks}(s, d \rightarrow u) \quad ::= \quad & object(d) = object(u) \in Candidates \\ & \wedge\ reaches(d, s) \wedge s \text{ dom } u\end{aligned}$$

Hierbei enthält *Candidates* alle potentiell betroffenen Objekte gemäß den Ausführungen des vorigen Abschnitts. Gegenüber Definition 8.1.4 ist diese Einschränkung hinzugekommen; dafür können wir das hier unbekannte Objekt der noch zu bestimmenden Konkretisierung für s ignorieren.

8.2.3 Zusammengesetzte starke Aktualisierungen

Für zusammengesetzte starke Aktualisierungen (CSU) müssen wir neben dem indirekten Fall auch die Situationen mit direkten Definitionen betrachten. Neben der Bestimmung der (potentiell) betroffenen Kanten ist dabei auch die eigentliche Erkennung der starken Aktualisierung zu besprechen. Für die nachfolgenden Betrachtungen nutzen wir die ISSA-Form aus, die uns mit den ϕ-Knoten die Unterstützung der CSU erleichtert.

Ist $G(s, t)$ eine tatsächliche CSU, so existiert auf jedem ICFG-Pfad von s nach t eine einfache starke Aktualisierung. Dabei gibt es auch welche, für die t in der iterierten Dominanzgrenze liegt. Daher enthält der ISSA-Graph in t einen ϕ-Knoten, auf den sich alle nachfolgende Verwendungen beziehen. Die CSU beherrschen wir somit bereits implizit dadurch, dass jede einzelne einfache starke Aktualisierung darin den Datenfluss von vorausgehenden Definitionen unterbindet. Damit gibt es auf keinem ICFG-Pfad einen IDFG-Pfad von vorausgehenden Definitionen über s und t zu nachfolgenden Verwendungen.

Betrachten wir zur Illustration das Beispiel, das wir weiter oben schon angeführt hatten:

```
x = 1;
if (...) *p = 2;
else *p = 3;
... = x;
```

Wenn hier eine tatsächliche CSU für x vorliegt, d.h. wenn p (nur) auf x zeigt, so haben wir in beiden Zweigen des `ifs` jeweils eine indirekte starke Aktualisierung für x, also eine SSU. Diese beiden Definitionen bilden die Definitionen des ϕ-Knotens für x, der dann nach dem `if` angelegt wird. Die nachfolgende Verwendung von x bezieht sich am Ende auf diesen ϕ-Knoten. Wir müssen daher nur erreichen, dass sich die Verwendung zuvor nicht auf eine weiter vorne befindliche Definition bezieht, und dass die indirekten Definitionen für x tatsächlich als solche angelegt werden, d.h. ohne eingehende Kante.

Kehren wir zum allgemeinen Fall zurück. Unsere Überlegungen bedeuten, dass mit den direkten SSUs auch die direkten CSUs behandelt sind. Das liegt daran, dass bereits die Kernanalyse ohne besondere Unterstützung für CSUs und indirekte starke Aktualisierungen die direkten SSUs beherrscht, die wiederum (wie nun diskutiert) für die direkten CSUs ausreichen.

Für indirekte starke Aktualisierungen wie im Beispiel ist dies jedoch nicht der Fall: Solange die in $G(s,t)$ enthaltenen SSUs noch allesamt 0-Definitionen sind, fehlt der ϕ-Knoten in t. Der Wirkungsbereich der 0-Definitionen (d.h. im Beispiel der indirekten Definitionen in den Zweigen des `ifs`) reicht dabei wegen fehlender Dominanz nicht zu t oder darüber hinaus, so dass ohne eine spezielle Unterstützung die auf t folgenden Verwendungen sich noch auf eine Definition vor s beziehen. Für die Erkennung einer potentiellen CSU müssen wir daher Teilgraphen mit den Bedingungen aus Definition 8.1.2 identifizieren, wobei wir „tatsächliche SSU“ durch „potentielle SSU“ ersetzen. Dann können wir denselben Lösungsansatz wie für einfache starke Aktualisierungen anwenden, d.h. eine potentielle CSU wie eine reale CSU behandeln, bis das Gegenteil bewiesen ist.

8.3 Der Blockade-Graph

Wir wissen nun, welche Objekte und Datenfluss-Kanten potentiell von einer starken Aktualisierung betroffen sind. Während die Bestimmung der Objekte dabei klar ist, fordern die *blocks*-Prädikate für die Kanten die Überprüfung der interprozeduralen Dominanz. In diesem Abschnitt wollen wir näher darauf eingehen, wie wir diese Aufgabe effizient erfüllen können. Außerdem

zeigen wir, wie zusammengesetzte starke Aktualisierungen erkannt werden. Konkrete Algorithmen dafür präsentieren wir in Abschnitt 8.3.3.

8.3.1 Interprozedurale Dominanz für 1-Definitionen

Ist das bislang einzige bekannte Objekt an einer 1-Definition lokal, so kann diese Definition keine interprozeduralen Auswirkungen haben. Um das Prädikat $blocks(s, d \rightarrow u)$ auszuwerten, können wir uns daher in diesem Fall auf die lokale Dominanz zurückziehen. Für den Fall eines nicht-lokalen Objekts beobachten wir folgendes: Eine 1-Definition ist bereits ein vollwertiges Datenfluss-Element, d.h. ein Knoten im IDFG. Entsprechend hat der Graphaufbau dafür auch bereits Seiteneffekte propagiert. Hinzu kommt, dass wir Untereingänge an direkten Aufrufen stets als starke Aktualisierungen angelegt haben. Beides zusammen sorgt dafür, dass die interprozeduralen Effekte der potentiellen starken Aktualisierung bereits beachtet werden. Daher können wir uns auch in diesem Fall auf die lokale Dominanz zurückziehen. Algorithmisch ist die Überprüfung der lokalen Dominanzbedingung im Rahmen unserer ISSA-Implementierung einfach und ohne nennenswerte Mehrkosten möglich (vgl. Abschnitt 8.3.3). 1-Definitionen als potentielle starke Aktualisierungen sind damit mit der kombinierten Analyse leicht zu behandeln.

8.3.2 Zusammensetzung des Blockade-Graphen

0-Definitionen sind noch keine Datenfluss-Elemente. Daher existieren auch noch keine Seiteneffekte, welche uns die interprozeduralen Effekte auf lokale Dominanz zurückführen lassen. Ebenso fehlen im IDFG noch die ϕ-Knoten, welche eine einfache Beherrschung der CSU ermöglichen.

Unsere Lösung für dieses Problem ist so einfach wie effektiv: Wir bestimmen in einem separaten Graphen, welche indirekten Definitionen und welche von diesen induzierte Knoten dem IDFG noch hinzugefügt werden könnten. Dieser separate Graph ist der Blockade-Graph.

Der Blockade-Graph nimmt damit vorweg, welche Knoten mit dem ersten Zeigerziel an einer indirekten Definition dem IDFG hinzugefügt würden. Hier nutzen wir aus, dass eine Definition auf ein nicht-lokales Objekt immer einen

Seiteneffekt-Knoten induziert – daher ist klar, dass die Propagierung nicht vorzeitig stoppen muss. Wir überschätzen mit dem Blockade-Graphen somit nur im Falle eines lokalen Objekts als Zeigerziel. Der Korrektheitsbeweis wird uns zeigen, dass dies keine Probleme bereitet.

Definition 8.3.1 Der *Blockade-Graph* B zu einem Programm besitzt als Knoten eine Teilmenge der Instruktionen. Diese Teilmenge setzt sich zusammen aus den indirekten Definitionen ohne bislang bekanntes Zeigerziel sowie aus von diesen transitiv induzierten Knoten gemäß den Regeln

1. Für $v \in B$ alle Knoten $w \in DF^{+}(v)$, womit die iterierte Dominanzgrenze gemeint ist. w wird jedoch nur dann aufgenommen, wenn es zu jedem CFG-Vorgänger mindestens einen Knoten in B gibt, der w induziert.

2. Für $v \in B$ alle Knoten für Aufrufstellen zur v enthaltenden Funktion f, falls v dom $CFG_Exit(f)$.

Kanten existieren von einem Knoten $v \in B$ zu allen von ihm induzierten Knoten.

Die Regel 1 für induzierte Knoten schließt eigentlich induzierte ϕ-Knoten aus, wenn zu einem CFG-Vorgänger kein Element aus B existiert. Denn in diesem Fall ist für jede Definition aus der Richtung dieses Vorgängers bereits ein Ziel bekannt und damit ein realer ϕ-Knoten im IDFG angelegt. Ein Teilgraph, der in w endet, kann somit nur noch eine CSU für bereits in diesem ϕ-Knoten bekannte Objekte sein. Der existierende ϕ-Knoten jedoch reicht aus, um die Wirkung der CSU umzusetzen: Seine Eingänge werden pro Pfad in der potentiellen CSU blockiert (durch 0- oder 1-Definitionen), wenn dort noch eine SSU möglich ist. Nachfolgende Verwendungen beziehen sich auf den ϕ-Knoten und erhalten damit Zeigerziele von vorausgehenden Definitionen nur, wenn sie einen IDFG-Pfad zu diesem ϕ-Knoten etabliert haben. Ein solcher Pfad existiert aber nur, wenn ein Teilgraph keine CSU mehr sein kann.

Beispiel 8.3.2 Betrachten wir zur Verdeutlichung eine erweitere Fassung unseres Beispiels:

```
void f (int* p)
{
    x = 1;
    while (...)
    {
        if (...) *p = 2;
        else *p = 3;
        ... = x;
    }
    *p = 5;
}

void main ()
{
    x = 2;
    f (&x);
    printf (x);
}
```

Hier haben wir drei indirekte Definitionen (jeweils über p), die initial in B aufgenommen werden. Die indirekte Definition `*p = 5` dominiert dabei den Ausgang des CFGs von f, so dass die Aufrufstelle in *main* gemäß Regel (2) ebenfalls in B aufgenommen wird. In diesem Beispiel ist das auch nötig, damit die Ausgabe in *main* nicht als Definition die Zuweisung in *main* benutzt, wie es ohne besondere Unterstützung der Fall wäre.

Die beiden anderen indirekten Definitionen haben beide in ihrer Dominanzgrenze die Instruktion unmittelbar hinter dem `if`, wo auch ein ϕ-Knoten entstehen würde. (Unsere IR hält hierfür spezielle Instruktionen ohne Inhalt bereit, so dass im Beispiel noch eine solche Dummy-Instruktion vor der Verwendung von x existiert.) Gemäß Regel (1) wird diese Dummy-Instruktion nun ebenfalls zu B hinzugefügt.

Diese Instruktion enthält ihrerseits die Instruktion für den ϕ-Knoten im Schleifenkopf in ihrer Dominanzgrenze. Da es jedoch in B keinen Knoten gibt, der diese Instruktion auch über den zweiten CFG-Vorgänger (also über den Code vor der Schleife) induziert, greift die besondere Formulierung in Regel (1) und nimmt den Schleifenkopf nicht auf. Der initiale Blockade-Graph für das Beispiel sieht damit wie in Abbildung 8.1 aus. □

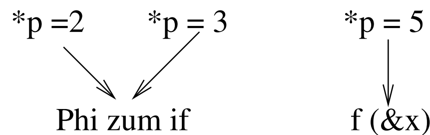


Abbildung 8.1: Blockade-Graph im Beispiel

Unsere Analyse kann nun 0-Definitionen wie 1-Definitionen behandeln, mit dem Unterschied, dass dazu der Blockade-Graph berücksichtigt werden muss anstelle des IDFG.

8.3.3 Algorithmen zur Berechnung und Anwendung des Blockade-Graphen

Wenden wir uns nun dem konkreten Vorgehen zur Berechnung und Anwendung des Blockade-Graphen zu. Zunächst skizziert dieser Abschnitt dazu eine grobe Vorstellung von der Integration in die bisherige Kernanalyse, bevor anschließend die Algorithmen präsentiert werden.

Den initialen Blockade-Graphen berechnen wir in der Initialisierung, ausgehend von allen indirekten Definitionen (die zu diesem Zeitpunkt allesamt 0-Definitionen sind), welche starke Aktualisierungen ergeben könnten. Mittels lokaler Dominanz und Dominanzgrenzen sowie dem direkten Aufrufgraphen gelingt hier in Linearzeit die Konstruktion von B.

Angewendet wird der Blockade-Graph bei der Suche nach einer gültigen Definition, die wir zur Ermittlung der ISSA-Kanten zwischen allgemeinen Definitionen und Verwendungen benutzen. Diese Suche prüft nun, ob auf dem Weg zwischen Dominator und aktuellem Knoten ein Knoten $v \in B$

liegt und damit die Kante blockiert. Falls dem so ist, wird die Kante nicht angelegt, sondern sozusagen zurückgestellt.

Erkennt die Analyse später ein Zeigerziel für eine der in B enthaltenen 0-Definitionen, so wandert diese von B in den IDFG. Insbesondere müssen wir sie also mit allen ausgehenden Kanten aus B löschen. Ferner passen wir B dadurch an, dass wir (auch transitiv) induzierte Knoten über die wiederholte Anwendung folgender Löschregeln entfernen, bis sich keine weitere Veränderung mehr ergibt:

1. Erniedrigt sich durch das Löschen der Eingangsgrad einer in B induzierten ϕ-Knoten-Instruktion, so löschen wir auch diese mitsamt allen eingehenden Kanten aus B: Sie gehört nicht mehr gemäß Regel (1) aus dem vorigen Abschnitt zu B.

2. Außerdem wollen wir alle ausschließlich von nun gelöschten Knoten induzierten Aufrufstellen aus B entfernen. Hierzu betrachten wir den SCC-DAG von B und löschen darin alle Knoten, bei denen der Eingangsgrad auf Null sinkt. (Aufrufstellen können mehrere eingehende Kanten aus der gleichen Funktion haben. Gäbe es z.B. in unserem Beispiel eine zweite indirekte Definition in f, welche den Ausgang des CFGs dominiert, so hätte diese auch eine Kante zur Aufrufstelle in $main$. Daher müssen wir warten, bis der Eingangsgrad auf Null sinkt.)

In den folgenden Unterabschnitten zeigen wir, wie man den Ansatz zur Erkennung starker Aktualisierungen in die Algorithmen der bisher beschriebenen Kernanalyse integrieren kann. Dazu müssen wir die nun eingeführten Blockaden erkennen und respektieren.

Im Vergleich zur bisherigen Kernanalyse führen die Blockaden weiteren Aufwand ein; auf der anderen Seite aber verhindern sie die Propagierung einiger Zeigerziele und reduzieren damit die Graphgröße. Die Auswirkungen dieser Erweiterung auf den Zeit- und Platzbedarf sind daher zweischneidig, die Resultate werden jedoch präziser.

Die hier vorgestellte Implementierung auf ISSA-Basis benötigt keine Veränderungen im Propagierungs-Schritt. Auch der Graphaufbau-Schritt ändert sich nur wenig. Der Hauptaufwand besteht in der Verwaltung und Beachtung des Blockade-Graphen.

8.3.3.1 Initialer Blockade-Graph

Die Initialisierung der kombinierten Analyse konstruiert nun den initialen Blockade-Graphen vor der Fixpunkt-Iteration. Dazu trägt sie alle indirekten Definitionen als Knoten ein, denn diese haben zu Beginn alle noch kein Zeigerziel. Die Bestimmung der restlichen (von diesen induzierten) Knoten ist eine Propagierung dieser initialen Knoten aufwärts im Aufrufgraphen sowie eine Traversierung der Dominanzgrenzen aller Knoten. Da wir über indirekte Aufrufe keine starken Aktualisierungen propagieren, genügt uns hier der schon in der Initialisierung bekannte Anteil des Aufrufgraphen.

Abbildung 8.2 zeigt das Verfahren zur Konstruktion des initialen Blockade-Graphen. Die wachsende Menge *Nodes* der Knoten von B wird als Worklist benutzt, jeder darin enthaltene Knoten wird einmal betrachtet. Zum jeweils aktuellen Knoten ermittelt das Verfahren die ausgehenden Kanten zur Aufnahme in die Menge *Edges* aller Kanten von B, sowie die induzierten Knoten. Aufrufstellen c zum Unterprogramm f werden aufgenommen oder mit einer neuen Kante verbunden, falls der aktuelle Knoten den Ausgang von f dominiert. Für potentielle ϕ-Knoten ist etwas mehr Aufwand nötig, da sie nur aufgenommen werden dürfen, wenn zu jedem CFG-Vorgänger mindestens eine Kante existiert. Daher merken wir uns die Kandidaten zunächst in einer separaten Menge *Phis* und die bislang gefundenen eingehenden Kanten in *Phi_Pred*, gruppiert nach CFG-Vorgängern. Sobald genügend viele eingehende Kanten gefunden sind, wandert der Kandidat und mit ihm die eingehenden Kanten von *Phis* bzw. *Phi_Pred* nach B.

Knoten in B, die sich dominieren, müssen jeweils eine Kante zu den gemeinsam induzierten Knoten haben. Das Verfahren stellt dies sicher. Der Algorithmus endet damit, den SCC-DAG von B zu berechnen, damit ein späteres Löschen von Knoten keine Zyklen ohne Eingänge übrig lässt.

```
procedure Create_Blocking_Graph
is
begin
   Nodes := Edges := Phis := Phi_Pred (·) := ∅;
   for all v ∈ Indirect_Definitions loop
      add v to Nodes;
   end loop;
   -- add nodes induced by new nodes
   for all new v ∈ Nodes loop
      for all w ∈ DF(v) loop
         if w ∈ Nodes then
            for all p ∈ pred(w), v dom p loop
               add v → w[p] to Edges;
            end loop;
         else
            if w ∈ Phis then
               for all p ∈ pred(w), v dom p loop
                  add v to Phi_Pred (w,p);
               end loop;
               if ∀p ∈ pred (w) Phi_Pred (w,p) ≠ ∅ then
                  move w to Nodes and Phi_Pred (w,·) to Edges;
               end if;
            else
               add w to Phis;
               for all p ∈ pred(w), v dom p loop
                  Phi_Pred (w,p) := {v};
               end loop;
            end if;
         end if;
      end loop;
      if v dom CFG_Exit (func(v)) then
         for all c ∈ calls to func(v) loop
            if c ∉ Nodes then
               add c to Nodes;
            end if;
            add v → c to Edges;
         end if;
      end if;
   end loop;
   use Tarjan's algorithm to compute SCC-DAG;
end Create_Blocking_Graph;
```

Abbildung 8.2: Konstruktion des initialen Blockade-Graphen

Beispiel 8.3.3 Wenden wir das Verfahren auf Beispiel 8.3.2 an. Nachdem die drei indirekten Definitionen in $Nodes$ aufgenommen sind, beginnt die Hauptschleife. Nehmen wir an, sie betrachtet zuerst die Instruktion `*p = 2` als Knoten v. Dann wird sie als Knoten $w \in DF(v)$ die Instruktion unmittelbar nach dem `if` in die Menge $Phis$ aufnehmen und v als Phi_Pred des Then-Zweigs eintragen. Als nächstes sei v die Instruktion `*p = 3`. Hier ist nun das gleiche $w \in DF(v)$ bereits in $Phis$. Zuerst wird daher v als Phi_Pred für den Else-Zweig eingetragen. Anschließend stellt der Algorithmus fest, dass w in den Graphen aufgenommen werden muss, und verschiebt den Knoten entsprechend von $Phis$ nach $Nodes$.

Anschließend sei v die Zuweisung `*p = 5`. Hier stellt das Verfahren die Dominanz zum Ausgang von f fest, weswegen die Aufrufstelle in $main$ in B aufgenommen wird. Nehmen wir an, diese würde gleich darauf als neues v betrachtet. Auch sie dominiert den Ausgang, diesmal von $main$. Gibt es eine Wurzel im Aufrufgraphen über $main$, so würde nun darin der Aufruf an $main$ als Knoten in B aufgenommen (das Beispiel hat dies der Einfachheit halber nicht ausgeführt, so dass wir auch nicht weiter darauf eingehen.)

Es bleibt noch die ϕ-Knoten-Instruktion nach dem `if` als Knoten v für die Hauptschleife. Diese betrachtet die ϕ-Knoten-Instruktion im Schleifenkopf als $w \in DF(v)$ und nimmt diese in $Phis$ auf. Dort jedoch verbleibt sie bis zum Ende des Verfahrens, wird also nicht in B aufgenommen. □

Verwendet man die Suche nach der gültigen Definition über den lokalen Objektgraphen, so konstruiert die Initialisierung zusätzlich noch die lokalen Dominanzbäume zu B. Deren Verwendung bespricht der nächste Abschnitt.

8.3.3.2 Erkennung blockierter Kanten

Mit Hilfe des Blockade-Graphen müssen wir nun erkennen, welche Datenfluss-Kanten blockiert sind. Diese blockierten Kanten wollen wir dann gar nicht erst anlegen. Dies unterbindet die Propagierung von Zeigerzielen, d.h. eine voreilige Nutzung der Kanten, ohne weiteres Zutun. Sollte sich eine Blockade später als falsch herausstellen, so holen wir dann das Einziehen der Kante einfach nach.

Die Frage, ob eine Kante blockiert ist oder nicht, wollen wir daher direkt bei der Erzeugung von Datenfluss-Kanten beantworten. Wir haben die Datenfluss-Kanten dabei über die Suche nach der gültigen bzw. vorausgehenden Definition erzeugt (vgl. z.B. Abbildung 5.8). Daher integrieren wir die Beachtung des Blockade-Graphen in diese Suche. Die sich ergebende neue Suche zeigt Abbildung 8.3. Wir stellen fest, dass die Bedingungen für (potentielle) starke Aktualisierungen dabei hervorragend zur ISSA-Darstellung passen, die bereits auf Dominanz-Informationen zurückgreift.

```
function Find_Reaching_Def (N : Node) return (Node, Boolean)
is
   IDom : Node := N.Dominator;
   Blocked : Boolean := False;
begin
   while IDom ≠ none loop
      if Is_Reaching_Def (IDom, N) then
         return (IDom, Blocked);
      end if;
      if not Blocked and IDom ∈ B then
         Add N to IDom.Blocked_Nodes;
         Blocked := True;
         if not Is_General_Def (N) then
            return (IDom, Blocked);
         end if;
      end if;
      IDom := IDom.Dominator;
   end loop;
   return (none, Blocked);
end Find_Reaching_Def;
```

Abbildung 8.3: Respektierung des Blockade-Graphen bei der Suche nach einer gültigen Definition über den CFG-Dominanzbaum

Während der Suche prüfen wir dabei nun für jede angetroffene (dominierende) Anweisung, ob sie in B enthalten ist oder nicht. Falls ein solcher Knoten aus B angetroffen wird, so liegt eine potentielle starke Aktualisierung zwischen dem Knoten N und seiner gültigen Definition. Würde die Kernanalyse eine Kante zwischen der Definition und N einziehen, so müssen wir dies nun unterbinden, weil die Kante blockiert ist. Das signalisiert

Find_Reaching_Def über den nun zusätzlich zurückgegebenen booleschen Wert, den der Aufrufer entsprechend beachten muss.

Um später die blockierte Kante nachholen zu können, falls sich die Blockade als falsch herausstellt, vermerken wir dabei noch den Knoten N an demjenigen Knoten, der bei der Suche als erstes als ein Knoten aus B angetroffen wurde. Ist N ein ϕ-Knoten, so merken wir uns dabei sowohl den Knoten als auch den Eingang, für den wir nach der Definition gesucht hatten. Im nächsten Abschnitt werden wir zeigen, wie wir damit die Kante konkret nachholen. Falls noch weitere Knoten aus B bei der Suche angetroffen werden, können wir sie hier ignorieren, da wir beim Nachholen der Kante explizit noch einmal die gleichen Überprüfungen durchführen.

Daher können wir die Suche in vielen Fällen auch bereits beim Antreffen des ersten Knotens aus B abbrechen. Lediglich bei einer allgemeinen Definition setzen wir sie bis zum Ende fort, denn dort besteht neben dem eventuellen Einziehen einer Datenfluss-Kante noch die Aufgabe, dominierte Verwendungen der gültigen Definition zu angeln. Dies geschieht auch bei blockierter Kante. Ist N eine potentielle oder tatsächliche starke Aktualisierung, so suchen wir zwar nach der vorausgehenden Definition, jedoch wollen wir keine Kante einziehen, sondern nur dominierte Verwendungen angeln. In diesem Fall können wir daher den Blockade-Graphen ignorieren und die ursprüngliche Suche verwenden.

In unserem Beispiel wird auf diese Weise z.B. die Verwendung von x im `printf`-Aufruf in *main* bei der Suche nach ihrer Definition am Aufruf `f(&x)` blockiert. Ähnlich gelangt die Suche für die Verwendung von x innerhalb der Schleife (nach dem `if`) nur bis zur ϕ-Knoten-Instruktion.

Verwendet man die Suche nach der gültigen Definition über eine Darstellung des lokalen Objektgraphens, so ist eine andere Strategie nötig. Dann nämlich betrachtet die Suche nicht jede Anweisung zwischen N und dessen (zu findenden) Dominator. Daher kann der Test, ob dazwischen ein Element aus B liegt, nicht wie oben direkt in die Suche integriert werden. Wir verwenden stattdessen einen separaten Test im Anschluss an die (unveränderte) Suche nach der gültigen Definition. Um diesen effizient realisieren zu können, organisieren wir den Blockade-Graphen lokal jeweils ebenfalls mit Hilfe der

Dominanzinformation, d.h. wir errichten den Dominanzbaum zu B: Zu jedem Knoten in B bestimmen wir den direkten Dominator in B, und zu jeder Instruktion bestimmen wir ebenfalls den direkten Dominator in B. Der Test, ob nun ein blockierender Knoten aus B zwischen N und dessen Dominator D liegt, nutzt diesen Dominanzbaum.

```
function Is_Blocked (N : Node; D : Node) return Boolean
is
   IDom : B_Node := B_Dominator (N.Instruction);
begin
   while IDom ≠ none and then not IDom dom D loop
      if Is_Still_In_B (IDom) then
         Add N to IDom.Blocked_Nodes;
         return True;
      end if;
      IDom := IDom.Dominator;
   end loop;
   return False;
end Is_Blocked
```

Abbildung 8.4: Separater Test auf eine Blockade bei der Suche nach der gültigen Definition über den Objektgraphen

Abbildung 8.4 zeigt den separaten Test, ob eine Blockade zwischen einem Knoten N und dessen Dominator D liegt. Diesmal durchläuft $IDom$ den Dominanzbaum zu B, und zwar bis entweder ein Knoten gefunden wird, der noch in B ist und damit blockiert, oder bis ein Knoten des Dominanzbaumes angetroffen wird, der D dominiert, oder bis die Wurzel des lokalen Dominanzbaumes erreicht ist. Während wir zwar im Laufe der Analyse Knoten aus B löschen (vgl. nächsten Abschnitt), bleibt der Dominanzbaum zu B für diese Tests bestehen: Ein Knoten im Dominanzbaum muss also erst auf Existenz in B überprüft werden, z.B. mittels einer Markierung.

8.3.3.3 Aktionen bei einer neuen Definition

Im Laufe der Analyse erkennen wir Zeigerziele für indirekte Definitionen. Die Strategie beim ersten Ziel besteht darin, es zunächst wie eine tatsächliche starke Aktualisierung zu behandeln (außer, das Objekt lässt prinzipiell keine

starken Aktualisierungen zu). Das bedeutet, dass für das erste Ziel keine Kante von der vorausgehenden Definition eingezogen wird. Wie im vorigen Abschnitt beschrieben, suchen wir diese vorausgehende Definition jedoch, um ggf. einige Verwendungen zu angeln.

Erhält die indirekte Definition später ein zweites Ziel, so haben wir damit für das erste Ziel nachträglich erkannt, dass es sich um eine schwache Aktualisierung handelt. Daher holen wir dann das Einziehen der Kante von der (dann) vorausgehenden Definition nach, indem wir die Suche aus dem vorigen Abschnitt verwenden. Die Definition für das zweite und alle weiteren Ziele ist von Beginn an als schwache Aktualisierung bekannt und führt die Suche sofort durch.

Beim ersten Ziel für eine indirekte Definition wird aus der bisherigen 0-Definition eine 1-Definition. Entsprechend müssen wir den bisher in B vorhandenen Knoten für die 0-Definition löschen. Dazu gehört auch, die von diesem Knoten induzierten Knoten im Blockade-Graphen anzupassen: Aufrufstellen, die nun Eingangsgrad Null haben (bzw. Zyklen mit diesem Grad) werden gelöscht, und Kandidaten für ϕ-Knoten, bei denen nun nicht mehr jeder Eingang eine eingehende Kante in B besitzt, verschwinden ebenso.

Sobald wir die erste Konkretisierung für eine indirekte Definition behandeln, rufen wir für den zugehörigen Knoten $V \in B$ die in Abbildung 8.5 gezeigte Prozedur *Update_Blocking_Graph* auf. Diese aktualisiert den Blockade-Graphen. Der Einfachheit halber beschreibt die Abbildung das Vorgehen auf B. Das tatsächliche Verfahren arbeitet jedoch auf dem SCC-DAG zu B, um Zyklen korrekt behandeln zu können.

Auf einem azyklischen Graphen ist die Bestimmung des betroffenen Teilgraphen nicht weiter schwer, wenn jeder Knoten über die Zahl der eingehenden Kanten verfügt. Eine Tiefensuche ab der auslösenden indirekten Definition, welche an allen Knoten mit noch weiteren eingehenden Kanten stoppt, erfüllt die Aufgabe, Knoten mit dem zukünftigen Eingangsgrad Null zu identifizieren. Der Test für ϕ-Knoten kann während dieser Tiefensuche wie im Pseudo-Code gezeigt erfolgen.

Die nun aus dem Blockade-Graphen entfernten Knoten haben zuvor in aller Regel einige Kanten blockiert. Neben der Aktualisierung des Blockade-

```
procedure Update_Blocking_Graph (V : Node)
is
begin
   for all (N,E) ∈ V.Blocked_Nodes loop
      -- restart search for reaching definition
      if Is_Phi_Node (N) then
         Postponed_New_Phi (N, E);
      elsif Is_Definition_Or_Subentry (N) then
         Postponed_New_Def (N);
      else
         Postponed_New_Use (N);
      end if;
   end loop;
   remove V from Nodes (B);
   for all S ∈ succ(V) loop
      S.Incoming_Edges := S.Incoming_Edges - 1;
      if S.Incoming_Edges = 0 then
         Update_Blocking_Graph (S);
      elsif Is_Phi_Candidate (S) and then
         not ∀p ∈ pred(S) ∃pred (S,p)∈ B
      then
         Update_Blocking_Graph (S);
      end if;
   end loop;
end Update_Blocking_Graph;
```

Abbildung 8.5: Aktualisierung des Blockade-Graphen bei Erkennung des ersten Zeigerziels einer indirekten Definition

Graphen sorgt die Funktion daher auch dafür, dass diese Kanten nun nicht mehr blockiert sind. Dazu haben wir uns an jedem Knoten in B diejenigen Knoten gemerkt, für welche wir nach der gültigen Definition gesucht haben, aber auf eine blockierte Kante getroffen sind. Da die Blockade nun zumindest wegen des gerade aus B entfernten Knoten nicht mehr gilt, behandeln wir die vermerkten Knoten fast so, als wären sie erst jetzt neu in den IDFG aufgenommen worden. Insbesondere starten wir die Suche nach der vorausgehenden Definition neu. Diese Strategie ist simpel und behandelt auch den Fall, dass die Kante möglicherweise noch durch andere Knoten aus B blockiert ist. Die Prozeduren *Postponed_New_...* erfüllen dies.

8.3.4 Indirekte Aufrufe

Bislang haben wir nur direkte Aufrufe beachtet. Interprozedurale Effekte potentieller starker Aktualisierungen über indirekte Aufrufe hinweg haben wir ignoriert. In vielen Fällen ist dies sogar in dem Sinne korrekt, dass wir nicht ohne Weiteres besser sein können: Die Kontrollfluss-Unsicherheit bei mehr als einem Aufrufziel verbietet, dass der Aufruf lokal vorbeiziehende Datenfluss-Kanten blockiert, wenn eine solche Blockade nicht in allen Aufrufzielen vorliegt.

Passend dazu, dass wir echte Seiteneffekte an indirekten Aufrufen ebenfalls aufgrund der Kontrollfluss-Unsicherheit immer nur als schwach annehmen, verzichten wir auch in puncto indirekte starke Aktualisierungen auf eine Propagierung über indirekte Aufrufe hinweg.

Wollte man indirekte Aufrufe dennoch unterstützen, so ist auch dies möglich. Dazu müssten wir einen indirekten Aufruf ohne bekannte Ziele zunächst wie eine 0-Definition behandeln und entsprechend in den Blockade-Graphen aufnehmen. Beim ersten erkannten Aufrufziel überprüfen wir dann, ob dieses noch in B enthalten ist. Wenn nicht, muss der indirekte Aufruf und ausschließlich davon induzierte Knoten aus B entfernt werden. Erkennen wir später weitere Aufrufziele für einen indirekten Aufruf, der noch in B verankert ist, so müssen wir den Aufruf entfernen, falls ein Aufrufziel nicht mehr in B ist. (Unsere Implementierung verzichtet jedoch auf diese Behandlung von indirekten Aufrufen.)

8.4 Beispiel

Betrachten wir erneut das Beispiel aus Abschnitt 4.2. Dieses Programmfragment haben wir zusammen mit dem initialen IDFG in Abbildung 8.6 noch einmal dargestellt. Der initiale Blockade-Graph hierzu ist einfach: Er besteht aus den beiden indirekten Definitionen. Die davon induzierten (vorweggenommenen) ϕ-Knoten sind nicht in B, da sie jeweils keinen blockierenden Vorgänger auf dem Pfad von vor der Schleife besitzen.

Für dieses Beispiel wird der Graphaufbau nun im Unterschied zu Ab-

```
p = &q;          // 1
while (...)
{
    q = &p;      // 2
    *p = &p;     // 3
    p = &x;      // 4
    *q = &q;     // 5
}
```

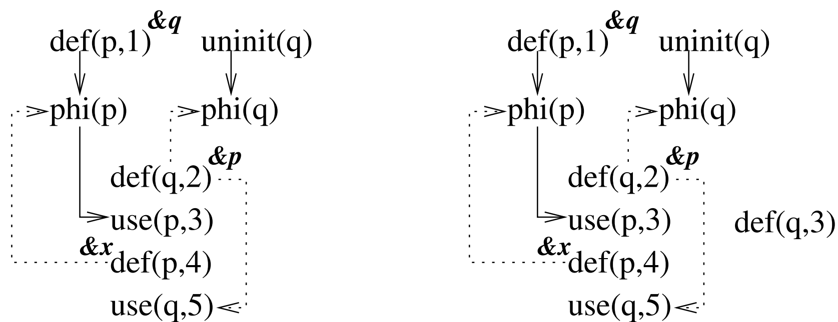
(a) Beispiel-Programm

def(p,1)
def(q,2)
use(p,3)
def(p,4)
use(q,5)

(b) Direkter IDFG

Abbildung 8.6: Beispielprogramm und initiale Knoten

schnitt 4.2 bei der Suche nach der gültigen Definition jeweils den Blockade-Graphen inspizieren. Sucht er beispielsweise für den ϕ-Knoten zu p nach dessen gültiger Definition innerhalb der Schleife, so trifft er auf die indirekte Definition in Zeile 5. Diese ist in B und blockiert somit die Kante von der Definition in Zeile 4. Ähnlich verhält es sich mit der Kante für q, welche die indirekte Definition in Zeile 3 überqueren müsste. Abbildung 8.7 zeigt links das somit entstehende Zwischenresultat nach dem Graphaufbau-Schritt der ersten Iteration. Dabei haben wir blockierte Kanten gepunktet gezeichnet. Diese Kanten dienen nur zur Veranschaulichung, sie sind nicht im derzeitigen ISSA-Graphen enthalten.



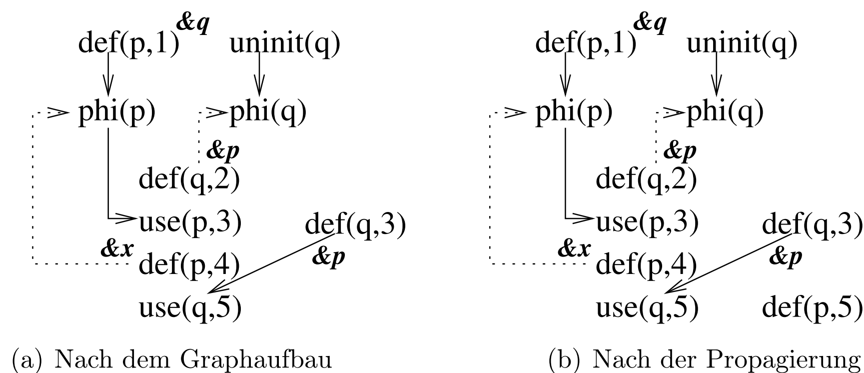
(a) Nach dem Graphaufbau

(b) Nach der Propagierung

Abbildung 8.7: Zwischenergebnisse der 1. Iteration

Die rechte Seite der Abbildung verdeutlicht die Auswirkungen auf den ersten Propagierungs-Schritt. Dieser findet nun nur noch für das Ziel q einen Weg zur Dereferenzierung in Zeile 3. Das in der Kernanalyse ursprünglich propagierte Ziel x wird ebenso wie das Ziel p nicht mehr propagiert, da die Kanten hierfür fehlen. Somit erzeugt die erste Iteration nur eine indirekte Definition.

Der folgende Graphaufbau-Schritt in Iteration 2 baut diese indirekte Definition in den ISSA-Graphen ein. Die Definition selbst wird dabei zunächst als starke Aktualisierung angesehen, benötigt also keine eingehende Kante. Die indirekte Definition wird aus B entfernt, weil wir nun bereits ein Ziel kennen. Das führt dazu, dass die zuvor zurückgestellte Kante für q (von Zeile 2 zu Zeile 5) nun verworfen und die Suche nach der gültigen Definition für $use(q, 5)$ erneut durchgeführt wird. Diese findet die neue indirekte Definition, womit eine Kante von dieser ausgehend angelegt wird. Das Resultat sehen wir in der linken Hälfte der Abbildung 8.8.



(a) Nach dem Graphaufbau

(b) Nach der Propagierung

Abbildung 8.8: Zwischenergebnisse der 2. Iteration

In der Abbildung sehen wir rechts das Ergebnis des sich anschließenden Propagierens. Diesmal können wir nur das Ziel p zur Dereferenzierung in Zeile 5 propagieren und dafür eine indirekte Definition erkennen.

Der Graphaufbau der dritten Iteration integriert diese in den ISSA-Graphen, ebenfalls als starke Aktualisierung. Die Blockaden durch die indirekte Definition werden aufgehoben und die zugehörigen Kanten neu ermittelt. Das

ergibt den neuen ISSA-Graphen, wie in Abbildung 8.9 links zeigt. Es ist zugleich der finale Graph vor dem Pruning, weil die Propagierung keine neuen Ziele finden kann. Ein abschließendes Pruning kann nun die Definitionen ohne ausgehende Kanten löschen. Das trifft hier auch den ϕ-Knoten zu q und in der Folge den Knoten für ein potentiell nicht initialisiertes q. Das Ergebnis sehen wir in Abbildung 8.9(b).

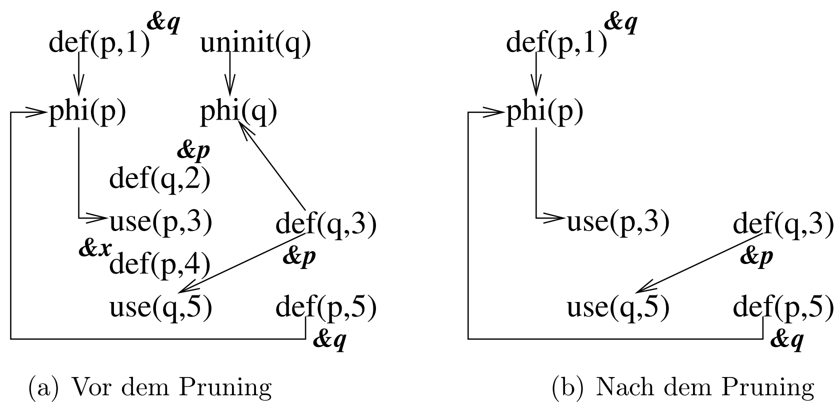


Abbildung 8.9: Ergebnis vor und nach dem Pruning

Das zeigt, dass unsere Technik für dieses Beispiel in der Lage ist, die optimale Lösung herzuleiten. Es zeigt aber auch, dass durch den Einsatz von Blockaden die Zahl der Iterationen steigen kann.

8.5 Umgang mit Fehlern

Eine indirekte Definition, welche durch einen Programmierfehler nie ein Ziel erhält, wird in der bislang beschriebenen Analyse bis zum Ende blockieren. Dies wiederum kann als Folge nach sich ziehen, dass für andere Dereferenzierungen eine Unterschätzung der Zeigerziele erfolgt, weil die indirekte Definition dafür notwendige Datenfluss-Kanten blockiert. Wünschenswert wäre es, diese Folgen eines Fehlers zu reduzieren. Dieser Abschnitt stellt hierzu eine Strategie vor.

Grob gesprochen besteht die Strategie aus zwei Teilen:

1. Eingrenzung der indirekten Definitionen, die als Fehler vermutet werden.
2. Freigabe der durch die vermuteten Fehler verursachten Blockaden und Fortsetzung der Analyse (*Relaxation*).

Die folgenden Abschnitte schildern unsere Umsetzung dieser beiden Aufgaben.

8.5.1 Eingrenzung der fehlerhaften Dereferenzierungen

Ein erster Gedanke auf dem Weg zur Eingrenzung der Fehler ist die Überlegung, einfach alle indirekten Definitionen als Fehler einzustufen, die bei Erreichen des Fixpunktes noch blockieren. Da die Folgen eines Fehlers aber auch dazu führen können, dass weitere indirekte Definitionen kein Zeigerziel erhalten, obwohl für diese eigentlich kein Programmierfehler vorliegt, verdächtigen wir damit zu viele Stellen.

Wir wollen daher genauer hinsehen, um falsche und echte Fehler zu unterscheiden. Unsere Strategie ist nun wie folgt: *Eine indirekte Definition ist dann als Fehler einzustufen, wenn sie kein Ziel erhalten hat und es einen ISSA-Pfad zu ihr gibt, der an einem Definitionsknoten für ein nicht initialisiertes Objekt startet.* Bei einer Rückwärts-Propagierung lässt sich dies leicht dadurch erkennen, dass die Dereferenzierung an einem entsprechenden Knoten eintrifft. Eine Vorwärts-Propagierung kann entweder ein gesondertes Zeigerziel (mit spezieller Semantik) verwenden oder im Fixpunkt über eine Rückwärts-Tiefensuche direkt nach einer entsprechenden Verbindung suchen.

Indirekte Definitionen, welche keinen Programmierfehler darstellen, im Fixpunkt aber dennoch kein Ziel erhalten haben, weisen bis auf durch Überschätzungen entstehende falsche Pfade keine derartige Verbindung auf. Stattdessen endet eine Rückwärts-Tiefensuche ab der Dereferenzierung an blockierten Kanten (bzw. die Dereferenzierung selbst wurde dorthin propagiert). Damit lässt sich also mit dem Kriterium die Menge der wahrscheinlichen Programmierfehler eingrenzen. Wir können weiterhin die Menge der potentiell fehlerhaften indirekten Definitionen auf solche eingrenzen, die nicht in totem Code liegen.

Bemerkung 8.5.1 Die Verschärfung des Kriteriums auf „*alle* Pfade zur Dereferenzierungen sollen an einem Knoten für ein nicht initialisiertes Objekt beginnen" erscheint uns etwas weniger geeignet: Die Menge der Fehler wird hierbei weiter unterschätzt, da durch Kontrollfluss-Überschätzung auch für echte Fehler ein blockierter Pfad vorliegen kann. (Eine Unterschätzung derart, dass aus gleichem Grunde auch ein Ziel vorliegen kann, findet bereits statt.) Daher bevorzugen wir das Existenz-Kriterium.

8.5.2 Fortsetzung der Analyse

Das Wissen um die vermuteten Fehlerstellen ist zum Einen ein nützliches Resultat für den Anwender oder für eine Analyse, die diese Stellen etwas genauer daraufhin durchleuchtet, ob wirklich ein Fehler vorliegt. Wir wollen es aber zum Anderen auch dazu nutzen, die Folgen der Fehler für die Analyse zu reduzieren. Unser Ansatz hierzu betrachtet schlicht alle als Fehler vermuteten indirekten Definitionen im Fixpunkt so, als hätten sie ein Ziel erhalten, und setzt die Analyse fort. Der grobe Ablauf sieht damit wie folgt aus:

1. Initialisierung mit Konstruktion des Blockade-Graphen
2. Fixpunkt-Iteration unter Beachtung der Blockaden
3. Eingrenzung der fehlerhaften indirekten Definitionen
4. Erneuter Graphaufbau-Schritt, wobei die vermuteten falschen indirekten Definitionen so behandelt werden, als hätten sie ein Ziel
5. Fortsetzung der Fixpunkt-Iteration (beginnend mit der Propagierung)
6. Ggf. Wiederholung der Schritte 3ff.
7. Finalisierung

Die ursprüngliche Fixpunkt-Iteration wird damit mit einer weiteren Fixpunkt-Iteration umschlossen, in welcher jeweils die Fehler-Relaxationsstrategie zum Einsatz kommt. So erhalten wir eine Folge von Fixpunkten mit

monoton wachsenden Zeigerziel-Mengen. Gleichzeitig schrumpft in der Regel die Liste der vermuteten Fehler pro Iteration. Ändert sich daran nichts mehr, so haben wir einen finalen Fixpunkt erreicht. Dann können wir die finale Fehlerliste als vermutete Fehler ausgeben. Die Relaxation kann jedoch auch dazu führen, dass echte Fehler aus dem Blickfeld der Analyse verschwinden, wenn durch die kurzfristige Freigabe der Blockaden eine falsche Verbindung aufgebaut wird. Daher können wir auch zusätzlich die früheren Fehlerlisten angeben bzw. zu einer Gewichtung nutzen, welche indirekte Definitionen wir wie sehr als Fehler einstufen: Solche, die bis zur finalen Fehlerliste übrig bleiben, vermuten wir sehr stark als Fehler, während solche, die nach der ersten Relaxation aus der Liste verschwinden, ein geringeres Gewicht erhalten.

Beispiel 8.5.2 Eine indirekte Definition, die im ersten Fixpunkt als Fehler eingestuft wurde, danach jedoch nicht mehr, kann dennoch ein echter Fehler sein. Dazu betrachten wir folgenden Code:

```
void f ()
{
    if (B) *p = ...;
}

void g ()
{
    B = true;
    f();
}

void h ()
{
    B = false;
    p = &x;
    *q = &y;
    f();
}
```

Uns interessiert dabei die indirekte Definition in f. Die Aufrufstelle in g liefert keine Definition für p, so dass auf diesem Wege p an der Dereferenzierung nicht initialisiert erscheint. Die andere Aufrufstelle in h liefert bis zum ersten Fixpunkt ebenfalls keine Definition, da die dortige indirekte Definition über q blockiert. Unsere Strategie erkennt beide indirekten Definitionen im ersten Fixpunkt als Fehler und setzt daher fort, als hätten beide ein Ziel. Das führt dazu, dass die Dereferenzierung von p transitiv eine Definition erhält, nämlich diejenige aus h, und damit auch ein Zeigerziel. Sie wird daher im zweiten Fixpunkt nicht mehr als Fehler eingestuft, obwohl der Aufruf aus g immer noch einen Fehler verursachen würde. □

Kapitel 9

Kontext-abhängige Propagierung

In diesem Kapitel schildern wir, wie das bekannte IFDS-Framework zur Lösung von Datenfluss-Problemen so auf unsere Analyse übertragen werden kann, dass die Datenfluss-Probleme nun auf dem ISSA-Graphen anstelle des ICFG spezifiziert und gelöst werden. Dabei kann sowohl der finale ISSA-Graph als auch der Zustand in einer Iteration der Analyse als Basis dienen. Letzteres nutzen wir aus, um den Propagierungs-Schritt mit Hilfe des Frameworks zu realisieren.

Der Vorteil dieses Vorgehens liegt darin, dass das Framework eine höhere Präzision erreicht als die bisherige kontext-insensitive Propagierung. Es sorgt dafür, dass Zeigerziele nur zu denjenigen Aufrufern zurückfließen, die sie auch verursacht haben, anstatt sie immer zu allen Aufrufern zu propagieren. Damit erreichen wir für den Zeigeranalysen-Teil die MOVP-Präzision (meet-over-all-valid-paths).

Ein weiterer Vorteil besteht natürlich darin, dass weit mehr Datenfluss-Probleme als nur die Propagierung der Zeigerziele mit dem Framework gelöst werden können. Gegenüber der ICFG-basierten Framework-Variante sehen wir hier den Vorteil, dass der ISSA-Graph als Grundlage die Transferfunktionen vereinfacht und den Aufwand auf die Datenfluss-relevanten Stellen konzentriert.

9.1 Problematik

9.1.1 Ungenauigkeiten der Kernanalyse

Die kontext-insensitive Natur der Kernanalyse bringt zumindest die folgenden drei Ungenauigkeiten mit sich:

1. Wir vereinen innerhalb eines Unterprogrammes f die Zeigerziele an einem IDFG-Knoten, auch wenn sie aus verschiedenen Kontexten stammen. In der Folge vereinen wir auch die IDFG-Teile, die durch Dereferenzierung entstehen.

2. Wir beachten auch bei der umgekehrten Flussrichtung von f zurück in seine Aufrufer nicht die verschiedenen Kontexte: Die Kernanalyse propagiert in dieser Richtung ein Zeigerziel zu jedem Aufrufer, auch wenn es ursprünglich weder aus diesem noch aus f selbst kam (sondern aus anderen Aufrufern).

3. Die kontext-insensitive Analyse modelliert den Heap sehr grob, da pro Allokationsstelle nur ein Heapobjekt angenommen wird.

Nach Tok [2007] bezeichnen wir mit Bezug auf die Flussrichtung der Ziele im Aufrufgraphen die erste Ungenauigkeit als *Abwärts-Kontext-Insensitivität*, die zweite analog als *Aufwärts-Kontext-Insensitivität*. Die dritte Ungenauigkeit bezeichnen wir als *kontext-insensitives Heapmodell*.

In diesem Kapitel wollen wir primär darauf hinarbeiten, die Aufwärts-Kontext-Insensitivität des Zeigeranalysen-Teils als Ungenauigkeit zu vermeiden. Sekundär ergeben sich dabei jedoch auch Ideen, die beiden anderen Ungenauigkeiten zu attackieren.

Betrachten wir die Aufwärts-Kontext-Insensitivität näher. Das zugrunde liegende Problem bei dieser Ungenauigkeit ist, dass es im ICFG – und in der Folge im IDFG – Pfade gibt, die definitiv keinem jemals stattfindenden Ausführungspfad entsprechen: beispielsweise von Aufrufer 1 nach f und von dort zurück nach Aufrufer 2. Daher charakterisiert man zunächst die Pfade:

Definition 9.1.1 Sei G ein ICFG, wobei wir der Einfachheit halber pro Aufrufstelle genau ein Ziel annehmen (d.h. Verzweigungen mit `if` für indirekte Aufrufe). Sei $e = u \rightarrow v$ eine Kante aus G, dann ist $annotation(e)$ gegeben durch:

- $(_u$, falls u eine Aufrufstelle ist
- $)_{pred(v)}$, falls v der Rückkehr-Knoten unmittelbar nach einer Aufrufstelle ist
- ε sonst

Für einen Pfad $p = e_1 \cdots e_l$ ist dann $annotation(p)$ gegeben durch die Konkatenation der Annotationen: $annotation(p) = annotation(e_1) \cdots annotation(e_l)$.

Mit dieser Annotation können wir die Pfade klassifizieren:

Definition 9.1.2 Ein Pfad p heißt *balanciert*, falls $annotation(p) \in L(G_B)$ für die Grammatik G_B mit Startsymbol B und den Produktionen

$$B \rightarrow \varepsilon \mid BB \mid (_c B)_c \qquad \forall c \in Calls$$

Ein Pfad p heißt *kontext-unabhängig*, falls $annotation(p) \in L(G_I)$ für die Grammatik G_I mit Startsymbol I und den Produktionen (B wie oben)

$$I \rightarrow B \mid B)_c I \qquad \forall c \in Calls$$

Ein Pfad p heißt *kontext-abhängig*, falls $annotation(p) \in L(G_S)$ für die Grammatik G_S mit Startsymbol S und den Produktionen

$$S \rightarrow B(_c S \mid B(_c \mid B(_c B \qquad \forall c \in Calls$$

Ein Pfad p heißt *gültig*, falls $annotation(p) \in L(G_I) \cup L(G_S)$.

Damit können wir die oben genannte Aufwärts-Kontext-Insensitivität auch als Problem der ungültigen Pfade auffassen. Entsprechend spricht die

Literatur dann auch im Unterschied zur aus dem intraprozeduralen Szenario gewohnten *meet-over-all-path*(MOP)-Lösung von der *meet-over-all-valid-path*(MOVP)-Lösung eines Problems, wenn die Aufrufsemantik beachtet wird.

Wie bereits in Kapitel 2.3.4.3 angesprochen, existiert zur Berechnung der MOVP-Lösung von Datenfluss-Problemen das IFDS-Framework von Reps u. a. [1995]. IFDS ist jedoch nur für distributive Probleme anwendbar. Wie wir mit Beispiel 2.3.1 gezeigt haben, sorgen direkte und indirekte starke Aktualisierungen dafür, dass die Transferfunktionen einer ICFG-basierten Spezifikation einer fluss-sensitiven Zeigeranalyse nicht distributiv sind. Daher beschreibt der Artikel [Guyer und Lin 2005, S. 4] auch den Stand der Dinge in Bezug auf IFDS für fluss-sensitive Zeigeranalysen wie folgt:

> this class does not include pointer analysis, particularly when it supports strong updates, which removes the distributive property.

Diesem Problem treten wir nun entgegen, indem wir – passend zu unserer kombinierten Analyse – diese Thematik vom ICFG auf den IDFG übertragen. Wir betrachten also Datenfluss-Pfade und annotieren diese analog zu den Kontrollfluss-Pfaden. Eine Datenfluss-Kante von einem Ausgang zu einem Untereingang an der Aufrufstelle c erhält somit die Annotation $)_c$, während eine Kante von einem Unterausgang an c zu einem Eingang die Annotation $(_c$ erhält. So können wir analog von gültigen und kontext-(un)abhängigen Datenfluss-Pfaden sprechen.

Im vorliegenden Kapitel beschreiben wir nun eine Erweiterung der Kernanalyse, welche das Problem der ungültigen Datenfluss-Pfade für die Propagierung vollständig löst. Die kombinierte Analyse wird damit also aufwärts-kontext-sensitiv: sie berechnet die MOVP-Lösung für das Problem der fluss-sensitiven Zeigeranalyse.

9.1.2 Beispiel

Wir verwenden das C-Programm aus Abbildung 9.1 zur Illustration der Problematik der Aufwärts-Kontext-Insensitivität. Die Funktion g besitzt hier zwei Aufrufer (h und k) und ruft ihrerseits f auf. Interessant ist vor allem

```
int **p, **q;
int *x,*y;
int a,b;
void f() {**p = 1; q = p;}
void g() {x = &a; f();}
void h() {a = 0; p = &x; g(); x = *q;}
void k() {b = 2; p = &y; y = &b; g(); y = *q;}
```

Abbildung 9.1: Beispielprogramm zur Kontext-Abhängigkeit

die zweifache Dereferenzierung in f mit ihren Auswirkungen. Im Kontext $h \rightarrow g \rightarrow f$ zeigt dort p auf x, welches seinerseits auf a zeigt. Im Kontext $k \rightarrow g \rightarrow f$ jedoch zeigt p auf y, welches seinerseits auf b zeigt.

Die kontext-insensitive Kernanalyse ermittelt hier für p und q die Zeigerziel-Menge $\{x, y\}$ und für die dann noch einmal dereferenzierten Knoten für diese Variablen die einelementigen Mengen $\{a\}$ und $\{b\}$. Die Zeigerziele und Seiteneffekte werden an alle Aufrufer zurückpropagiert. Dies führt in h und k zu Ungenauigkeiten, wo nun jeweils ein Seiteneffekt auf a und b vorliegt und wo nun die abschließende Dereferenzierung von q zwei Ziele liefert.

Als Ursache können wir mit unserer Terminologie feststellen, dass die Propagierung ungültige ICFG- und IDFG-Pfade beschreitet: Die ICFG-Pfade $h \rightarrow g \rightarrow k$ sowie $k \rightarrow g \rightarrow h$ sind ungültig, und in der Folge die IDFG-Pfade entlang dieser.

9.2 Das IFDS-Framework

Von T. Reps und anderen existiert seit den 1990er Jahren das sogenannte *IFDS*-Framework zur Formalisierung und Lösung von Datenfluss-Problemen mit MOVP-Präzision [Reps 1998; Reps u. a. 1995, 1994]. Die Abkürzung IFDS steht dabei für *interprocedural finite distributive subset* und benennt damit die Probleme, für welche das Framework geeignet ist. Diese müssen über eine endliche Faktenmenge mit distributiven Transferfunktionen beschrieben sein und ein Teilmengen-(Vereinigungs-)Problem darstellen. Das

Framework ist inzwischen der Standard zur Lösung solcher Probleme mit MOVP-Genauigkeit. Es wurde auch bereits auf eine Variante von Andersens fluss-insensitiver Zeigeranalyse angewendet, um deren Genauigkeit zu erhöhen [Reps 1998]. Wir wollen daher näher betrachten, wie sich IFDS analog für unsere fluss-sensitive Analyse nutzen lässt.

Wir haben bereits geschildert, dass eine ICFG-basierte Spezifikation einer fluss-sensitiven Zeigeranalyse wegen starker Aktualisierungen nicht für IFDS zugänglich ist. Die kombinierte Analyse dagegen operiert auf dem IDFG, und hier stellen wir fest, dass wir pro Iteration tatsächlich ein distributives Problem lösen wollen. Daran ändert auch die Unterstützung für indirekte starke Aktualisierungen nichts, die wir im Kapitel 8 besprochen haben, weil sie nur Änderungen im Graphaufbau-Schritt verlangt. Der Propagierungs-Schritt löst auch dann noch das Grapherreichbarkeitsproblem zwischen Zeigerziel-Quellen und Dereferenzierungen ohne Besonderheiten an bestimmten Knoten oder Kanten. Wir können daher als Grundlage für diesen Abschnitt das zu lösende Problem wie folgt beschreiben: *Gegeben der aktuelle ISSA-Graph, berechne darauf die MOVP-Lösung für das Grapherreichbarkeitsproblem zwischen Zeigerziel-Quellen und Dereferenzierungen.*

Im Folgenden beschreiben wir daher zunächst IFDS, bevor wir eine allgemeine Übertragung des Frameworks auf den ISSA-Graphen vorstellen. Anschließend schildern wir, wie das so übertragene Framework speziell für den Zweck der Propagierung von Zeigerzielen genutzt werden kann.

Grob betrachtet besteht die von Reps vorgeschlagene Lösung aus zwei Schritten:

1. Zuerst werden die Transferfunktionen als gerichtete, azyklische Graphen dargestellt. Diese Graphen werden zusammengesetzt und bilden den sogenannten *erweiterten ICFG* (engl. *exploded ICFG*, auch als *exploded supergraph* bezeichnet).

2. Im zweiten Schritt wird das zu lösende Problem als besonderes Grapherreichbarkeits-Problem auf dem erweiterten ICFG betrachtet und mit dem sogenannten *Tabulationsverfahren* gelöst.

Die folgenden Abschnitte gehen näher auf diese Schritte ein.

9.2.1 Der erweiterte ICFG

Interprozedurale Datenflussprobleme werden gewöhnlich auf dem ICFG spezifiziert. Dazu gehört ein Verband an Fakten, um die es geht, sowie pro ICFG-Kante eine Transferfunktion zur Beschreibung der Effekte dieser Kante (bzw. des Startknotens) auf eventuell vorliegende Fakten am Startknoten. Zusammen mit initialen Fakten und einem Konfluenzoperator zur Anwendung an ICFG-Konfluenzpunkten spezifizieren diese Angaben das Problem (vgl. Kapitel 2.2.1).

IFDS geht ebenfalls von einer solchen Spezifikation aus, mit den bereits erwähnten Zusatzforderungen an die Transferfunktionen und den Faktenverband. Das Framework verlangt dabei die Spezifikation der Transferfunktionen als bipartiten Graph. Dieser enthält jeden Fakt einmal als Eingangsknoten und einmal als Ausgangsknoten. Kanten verlaufen ausschließlich von Eingangsknoten zu Ausgangsknoten und repräsentieren den Effekt der ICFG-Kante auf den jeweiligen Eingabefakt. Zusätzlich existiert der besondere Fakt λ als Ein- und Ausgabeknoten, um geschickt alle *Gen*-Aktionen darstellen zu können.

Beispiel 9.2.1 Betrachten wir die Transferfunktion $f(x) = x$, d.h. die Identität. Gibt es nun die Fakten $d_1, \ldots, d_n$, so können wir diese Funktion darstellen als Graph wie in der folgenden Abbildung 9.2:

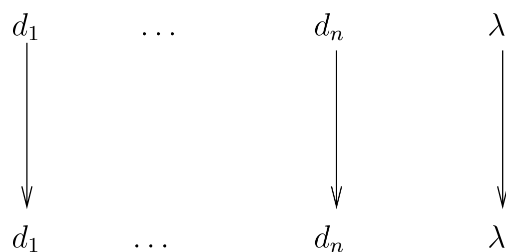


Abbildung 9.2: Identität als Graph dargestellt

Ein wenig komplizierter ist beispielsweise der Graph zur Transferfunktion $f : S \to (S \setminus \{d_1\}) \cup \{d_n\}$. Diese Funktion realisiert exemplarisch die typische Gen- und Kill-Aktion. Abbildung 9.3 zeigt den dazugehörigen Graphen. IFDS

spezifiziert die Transferfunktions-Graphen so, dass es keine Kante $x \rightarrow d_n$ gibt, wenn eine Kante $\lambda \rightarrow d_n$ existiert. Daher gibt es keine ausgehende Kante zum oberen d_n. Bis auf d_1 sieht der Graph für die übrigen Fakten aus wie bei der Identität. d_1 dagegen wird gelöscht, so dass auch für diesen Eingabefakt keine ausgehende Kante existiert. □

Abbildung 9.3: Gen-/Kill als Graph dargestellt

Die Kante $\lambda \rightarrow \lambda$ ist in jeder Transferfunktion vorhanden. Der Aufbau dieser Graphen erlaubt es, die Transferfunktionen sehr einfach miteinander zu verbinden: Die Ausgabefakten zu einer ICFG-Kante $u \rightarrow v$ verschmelzen mit den Eingabefakten aller von v ausgehenden ICFG-Kanten. Abbildung 9.4 zeigt dies für die beiden Beispielfunktionen, wenn diese aufeinander folgen.

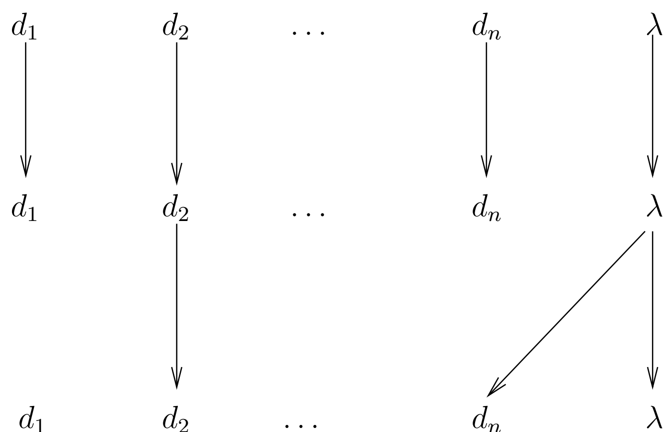


Abbildung 9.4: Konkatenation der Transferfunktions-Graphen

Wäre hier der Knoten zur zweiten Transferfunktion ein Konfluenzpunkt (d.h. hätte er mehrere Vorgänger), so würde die untere Reihe der Fakten alle Ausgabefakten der einzelnen Graphen für die eingehenden Kanten vereinen. Damit können wir die Transferfunktions-Graphen zu allen ICFG-Kanten miteinander verbinden. Der so definierte Graph ist der erweiterte ICFG. Pro ICFG-Knoten enthält er alle Fakten (darum die Forderung nach nur endlich vielen Fakten), plus den speziellen Fakt λ. Pro ICFG-Kante kann er bis zu $(d+1)^2$ Kanten aufweisen, wenn es d Fakten gibt.

Das Datenflussproblem lässt sich nun auf dem erweiterten ICFG als spezielles Grapherreichbarkeits-Problem betrachten: Berechne alle gültigen Pfade vom Knoten des erweiterten ICFG aus, der dem speziellen Fakt λ am Startknoten des ICFG entspricht. Markiert man dabei die Knoten, die auf diesen gültigen Pfaden liegen, so ergibt sich die MOVP-Lösung am Ende, indem man die an einem ICFG-Knoten markierten Faktenknoten aufsammelt.

In unserem Beispiel aus nur zwei Transferfunktionen würden wir also am obersten λ-Knoten als Repräsentant für „am Anfang liegt die leere Faktenmenge vor“ starten. Von diesem erreichbar ist lediglich der Fakt d_n am Knoten, welcher der unteren Reihe entspricht. Diese einelementige Menge am besagten Knoten wäre demzufolge die Lösung für das Beispiel.

9.2.2 Das Tabulationsverfahren

Das Tabulationsverfahren ist die mit IFDS vorgeschlagene Lösung des speziellen Grapherreichbarkeits-Problems [Reps u. a. 1995, S. 7]. Es verwaltet in einer Arbeitsliste gültige Pfade von Eingangsfakten der Unterprogramm-Startknoten des ICFG zu Fakten anderer ICFG-Knoten des jeweils gleichen Unterprogrammes (sogenannte *Same-Level-Pfade*). Das Verfahren arbeitet iterativ diese Liste ab, wobei die Behandlung eines Pfades daraus jeweils auch neue Pfade hinzufügen kann. Zu Beginn sind die ausgehenden Kanten des designierten Startknotens in der Liste enthalten.

Die Aktionen des Verfahrens unterscheiden sich sodann je nach Art des Zielknotens des gerade aus der Arbeitsliste entfernten und betrachteten Pfades. Im Folgenden gehen wir näher auf die einzelnen Fälle ein.

Intraprozeduraler Zielknoten. Intraprozedurale Knoten sind der einfachste Fall, sie verlangen keine besonderen Aktionen. Für den Pfad $u \xrightarrow{*} v$ nimmt der Algorithmus schlicht für jede Kante $v \to w$ den Pfad $u \xrightarrow{*} w$ in die Arbeitsliste sowie in die zusätzlich verwaltete, monoton wachsende, finale Liste von gültigen Pfaden auf, falls nicht schon früher geschehen.

Zielknoten am CFG-Ausgang. Endet der betrachtete Pfad mit einem Knoten am Ende eines Unterprogrammes, so wurde ein gültiger Pfad durch das gesamte Unterprogramm erkannt. Dies speichert das Verfahren als Zusammenfassung des Unterprogrammes ab. Diese Zusammenfassungen werden genutzt, um in Aufrufern gültige Pfade über den Aufruf hinweg zu konstruieren. Dazu betrachtet das Tabulationsverfahren alle Aufrufstellen, die zum Unterprogramm führen. Ist der aktuelle Pfad $e \xrightarrow{*} x$, so ermittelt es dort alle Vorgänger von e im erweiterten ICFG, sowie alle Nachfolger von x an gleicher Aufrufstelle. Damit lassen sich an der Aufrufstelle Abkürzungen von den ermittelten Unterausgängen zu den Untereingängen bilden. Diese speichert das Verfahren als *Summary-Kanten* ab. Gibt es zusätzlich bereits einen gültigen Pfad von einem Knoten am CFG-Eingang des Aufrufers zu einer solchen Aufrufstelle, so kann dieser zu den nun erreichbaren Rückkehrpunkten des Aufrufs verlängert werden. Diese neuen Pfade werden dann ebenfalls in die Arbeitsliste und die finale Liste gelegt.

Zielknoten an einer Aufrufstelle. Endet der betrachtete Pfad mit einem Knoten an einem Aufruf, so legt das Verfahren Pfade in die Arbeitsliste und die finale Liste für jeden vom Zielknoten aus erreichbaren Nachfolger. Diese Nachfolger sind Knoten am CFG-Eingangsknoten des Aufrufzieles. Die neuen Pfade sind jeweils trivial, sie gehen vom jeweiligen Eingang aus und enden auch an diesem. Für indirekte Aufrufe müssen hier alle Aufrufziele betrachtet werden (die Beschreibung bei Reps u. a. [1995] verzichtet darauf).

Außerdem kann es bereits *Summary-Kanten* für den Aufruf geben. Falls dem so ist, werden sie genutzt, um den aktuellen Pfad über die Aufrufstelle hinaus zu verlängern. Diese neuen Pfade werden dann ebenfalls in die Arbeitsliste und die finale Liste gelegt.

```
procedure Tabulation
is
begin
    Same_Level_Paths := {Entry(main), λ → Entry(main), λ};
    Work_List := Same_Level_Paths;
    Summary_Edges := ∅;
    while Work_List ≠ ∅ loop
        Take any path E = U,d → V,d' from Work_List
        if Is_Call (V) then
            Visit_Call (E);
        elsif Is_Exit (V) then
            Visit_Exit (E);
        else -- intraprocedural edge
            for all V,d' → S,d'' ∈ Exploded_Edges loop
                Propagate (U,d → S,d'');
            end loop;
        end if;
    end loop;
    for all V ∈ Nodes loop
        Facts (V) := {d | ∃d' : U,d' → V,d ∈ Same_Level_Paths};
    end loop;
end Tabulation;

procedure Propagate (E : Exploded_Path)
is
begin
    if E ∉ Same_Level_Paths then
        Insert E into Same_Level_Paths;
        Insert E into Work_List;
    end if;
end Propagate;
```

Abbildung 9.5: Das Original-Tabulationsverfahren

```
procedure Visit_Call (U,d → V,d': Exploded_Path)
is
begin
   -- assumption: one callee (direct call); process down edges
   for all V,d' → Entry(callee),d'' ∈ Exploded_Edges loop
      Propagate (Entry(callee),d'' → Entry(callee),d'');
   end loop;
   -- inspect summary edges
   for all V,d' → Subentry(V),d'' ∈ Exploded_Edges ∪ Summary_Edges loop
      Propagate (U,d → Subentry(V),d'');
   end loop;
end Visit_Call;

procedure Visit_Exit (U,u → V,v: Exploded_Path)
is
begin
   for all C ∈ Call_Sites_To (Get_Routine (V)) loop
      -- concatenate down - same level - up to summary edge
      for all C,c → U,u ∈ Exploded_Edges loop -- down
         for all V,v → Returnsite(C),r ∈ Exploded_Edges loop --up
            if C,c → Returnsite(C),r ∉ Summary_Edges then
               Insert C,c → Returnsite(C),r into Summary_Edges;
               for all U',e → C,c ∈ Same_Level_Paths loop
                  Propagate (U',e → Returnsite(C),r);
               end loop;
            end if;
         end loop;
      end loop;
   end loop;
end Visit_Exit;
```

Abbildung 9.6: Behandlung interprozeduraler Kanten im Original-Tabulationsverfahren

Abbildung 9.5 zeigt den Pseudo-Code zum Tabulationsverfahren. Wir haben hierzu die Formulierung aus Reps u. a. [1995] durch eine Darstellung mit Ada-ähnlicher Syntax ersetzt. Die Knoten des erweiterten ICFG bestehen jeweils aus einem ICFG-Knoten (großes Symbol, z.B. V) und einem Fakt (kleines Symbol, z.B. d), so dass eine Kante bzw. ein Pfad in diesem Graphen im Pseudo-Code die Gestalt $V, d \rightarrow W, d'$ hat. Die Menge *Exploded_ Edges* bezeichnet die Kanten des erweiterten ICFG.

Im Original wird anstelle des Symbols U (für den Startknoten eines Pfades) stets betont, dass es sich hierbei um den CFG-Eingang des jeweiligen Unterprogrammes handelt. Dies gilt natürlich in der hier gezeigten Formulierung ebenso. Da wir im Rahmen unserer Übertragung auf den ISSA-Graphen an dieser Stelle weitere Möglichkeiten schaffen werden, haben wir uns entschlossen, gleich das abstrakte Symbol zu verwenden.

Abbildung 9.6 zeigt die zum Verfahren gehörenden Hilfsfunktionen für interprozedurale Übergänge. Lediglich die Aktionen am CFG-Ausgang sind etwas aufwändiger. Zum besseren Verständnis nutzen wir dort den gleichen Buchstaben für einen ICFG-Knoten und für den an diesem betrachteten Fakt.

Dieser Algorithmus vermeidet unnötige Arbeit, indem er lokale, gültige Pfade nur von solchen Eingängen aus betrachtet, welche global bereits als erreichbar erkannt wurden.

9.3 IFDS auf dem ISSA-Graphen

Betrachten wir nun die Aufgabe, die Ideen aus dem IFDS-Framework für unsere Analyse nutzbar zu machen, speziell mit dem Hintergedanken, dieses für das Problem der Zeigerziel-Propagierung anzuwenden. Wir bemerken dabei die Problematik, dass IFDS auf dem ICFG aufbaut, während wir den ISSA-Graphen als Grundlage einsetzen. Tauschen wir diese Basis aus, so müssen wir entsprechend den *erweiterten ISSA-Graphen* bilden. In der Folge müssen wir das Tabulationsverfahren ein wenig anpassen, um mit der veränderten Grundlage zurecht zu kommen. Diese beiden Schritte besprechen wir im Folgenden, nachdem wir unseren Ansatz mit demjenigen von T. Tok verglichen haben.

9.3.1 Vergleich zu Toks Arbeit

In einer verwandten Arbeit hat Tok [2007] bereits ähnliche Gedanken verfolgt, weswegen wir hier kurz die Unterschiede klarstellen. Sein Ziel ist es, Zeit- und Platzbedarf von IFDS zu optimieren, indem nicht der vollständige erweiterte ICFG gebildet wird. Stattdessen erreicht Tok, dass im erweiterten ICFG nur die Fakten an einem ICFG-Knoten v zur Erweiterung benutzt werden, die an v definiert oder verwendet werden. Andere Fakten resultieren bei der Konstruktion des vollständigen erweiterten ICFG an v in Knoten mit genau einer eingehenden und genau einer ausgehenden Kante (vgl. die Transferfunktion in Abbildung 9.4). Die sich daraus ergebenden trivialen Pfade im vollständigen erweiterten ICFG verkürzt Tok also zu Kanten.

Um dies zu erreichen, geht Tok von einer zuvor ausgeführten fluss-sensitiven Zeigeranalyse aus, die neben den Zeigerzielen auch Def-Use-Ketten bestimmt haben muss. Im Unterschied zu ihm verwenden wir stattdessen die ISSA-Darstellung für den Datenfluss und haben auch die Anwendung im Sinne, die Zeigeranalyse selbst mit dem Framework zu realisieren. Tok bleibt außerdem weiterhin dabei, die Datenfluss-Probleme auf dem ICFG zu spezifizieren. Er beschreibt lediglich eine Optimierung für die Umsetzung der Transferfunktionen in den erweiterten ICFG. Für die dem Original-Framework zugänglichen Probleme berichtet Tok dabei, dass seine Optimierung Laufzeit und Platzbedarf spürbar reduziert (Faktor 2.6). Unsere Idee ist es hingegen, die Transferfunktionen auf dem ISSA-Graphen zu definieren, und so zum erweiterten ISSA-Graphen zu gelangen. Das erlaubt es uns, auch die Zeigeranalyse mit starken Aktualisierungen über distributive Transferfunktionen anzugeben, während sich mit Toks Arbeit noch nichts am Problem der Distributivität für die ICFG-basierte Spezifikation geändert hat.

Die Vereinfachung der Transferfunktionen und damit die Anwendbarkeit von IFDS auf weitere Probleme sehen wir als einen Vorteil unseres Vorgehens im Vergleich zu Tok. Nachteilig dagegen wirkt sich das größere Mengengerüst (erweiterter ISSA-Graph versus erweiterter ICFG) aus. Außerdem benötigen wir kleinere Anpassungen am Tabulationsverfahren, um mit der veränderten Grundlage zurecht zu kommen.

9.3.2 Der erweiterte ISSA-Graph

Um den erweiterten ISSA-Graphen zu konstruieren, legen wir analog zur Konstruktion des erweiterten ICFG pro ISSA-Knoten und Fakt des auf dem ISSA-Graphen zu lösenden Datenfluss-Problems einen Faktenknoten an. Die Verbindungen dieser Knoten stammen aus den (auf dem ISSA-Graphen spezifizierten) Transferfunktionen des Datenfluss-Problems. Wir gehen davon aus, dass diese Transferfunktionen in der Regel einfacher sind als die entsprechenden Transferfunktionen, wenn der ICFG als Basis zur Lösung des gleichen Problems verwendet wird. Ein Beispiel hierfür sind die Transferfunktionen für die Propagierung, wie wir sie in Abschnitt 9.4 besprechen werden.

Im (erweiterten) ISSA-Graphen treten nun die Eingänge und Ausgänge anstelle der CFG-Eingänge und CFG-Ausgänge des (erweiterten) ICFG. Untereingänge ersetzen die Rückkehrpunkte nach einem Aufruf, und Unterausgänge ersetzen die Aufrufstellen. Entsprechend verlaufen nun z.B. Summary-Kanten von Fakten an Unterausgängen zu Fakten an Untereingängen.

Beispiel 9.3.1 Zur Illustration greifen wir das Beispiel aus Abbildung 9.1 auf. Abbildung 9.7 zeigt uns dazu einen Ausschnitt des erweiterten ISSA-Graphen. Gezeigt ist der Teilgraph für p und q in der Funktion h unter Verwendung einfacher Transferfunktionen, wie wir sie in Kürze für die Propagierung von Zeigerzielen besprechen werden.

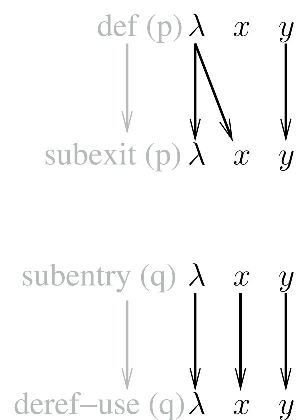


Abbildung 9.7: Erweiterter ISSA-Graph (Ausschnitt zum Beispiel)

Die ausgegrauten Knoten und Kanten sind diejenigen des ISSA-Graphen und nur als Orientierung angegeben. Zum erweiterten ISSA-Graphen gehören allein die Knoten und Kanten, welche normal stark dargestellt sind.

Nach der Konstruktion des erweiterten ISSA-Graphen können wir nun analog zum Original die Grapherreichbarkeit unter Beachtung der Aufrufsemantik bestimmen. In diesem Beispiel wäre ein Pfad von λ an der Definition von p zu x an der Dereferenzierung zu ermitteln, um den Datenfluss des Zeigerziels x zur Dereferenzierung zu beachten. Dazu müssen wir mit einem modifizierten Tabulationsverfahren die Summary-Kanten an der Aufrufstelle anlegen. □

9.3.3 Modifikationen am Tabulationsverfahren

Mit der Konstruktion des erweiterten ISSA-Graphen haben wir die Grundlage für die Übertragung des Frameworks geschaffen. Da der ISSA-Graph andere Eigenschaften als der ICFG aufweist, sind überdies Anpassungen am Tabulationsverfahren nötig. Dabei geht es um die Unterstützung für mehrere Startpunkte (und damit die Unterstützung für kontext-unabhängige Pfade).

Wir stellen fest, dass das Originalverfahren lediglich den Programmstart als initialen Same-Level-Pfad benutzt hat. Die Erzeugung von Fakten zwischendurch erfolgte über die besondere Modellierung der Transferfunktionen, bei denen dann eine von λ ausgehende Kante die Faktenerzeugung symbolisierte (vgl. Abbildung 9.3). Der ISSA-Graph ist im Unterschied zum ICFG[1] jedoch nicht zusammenhängend: Es gibt Knoten ohne eingehende Kanten, z.B. starke Aktualisierungen wie in `p = &x`. Das verhindert, dass wir mit nur einem Startpunkt auskommen, weil es keinen Pfad von einem ISSA-Knoten am Programmstart zu diesen Knoten gibt. Solch ein Pfad wäre aber notwendig, damit im erweiterten Graphen ein Pfad von λ am Start zum erzeugten Fakt führen kann.

Unsere Lösung besteht darin, mehrere initiale Same-Level-Pfade zu erlauben. Dies wird uns dann für die Zeigeranalyse auch die Unterstützung

[1] Reps u.a. betrachten keine indirekten Aufrufe und dergleichen, so dass stets der volle ICFG erreichbar ist

kontext-unabhängiger Pfade bzw. Ziele erlauben.

Zunächst bedeutet dies eine veränderte Initialisierung des Verfahrens. Die Menge Same_Level_Paths wird nun mit der folgenden Menge an Kanten gefüllt:

$$\{V, \lambda \rightarrow V, \lambda \mid indegree(V) = 0\}$$

(Eine Anwendung des Frameworks kann diese Menge natürlich weiter beschränken, wenn die jeweilige Problemstellung dies zulässt.) Man beachte hierzu, dass Transferfunktionen an die Kanten des ISSA-Graphen annotiert werden. Werden also an einem solchen Knoten V Fakten erzeugt, so sorgt dies im erweiterten ISSA-Graphen für Kanten, die von den zu V gehörenden Knoten – darunter auch demjenigen zu λ – ausgehen. Dementsprechend betrachtet der Algorithmus diese als erste Verlängerungen der initialen Pfade. Sollte daher eine dieser Transferfunktionen direkt einen Fakt generieren (z.B. den Fakt `&x` im Rahmen der Propagierung für die Definition V zur Anweisung `p = &x`), so wird dies nicht übersehen.

Die veränderte Initialisierung führt nun aber auch dazu, dass das Symbol U für den Startknoten eines Same-Level-Pfades nicht mehr unbedingt für einen Eingang im jeweiligen Unterprogramm steht. Denn neben den Eingängen treten nun weitere Startknoten für solche Pfade auf, die jeweils einen kontext-unabhängigen Pfad markieren:

- Alle Knoten V aus der obigen Menge für die Initialisierung.
- Untereingänge als Resultat davon, einen kontext-unabhängigen Pfad zu einem Aufrufer zu propagieren.

Damit haben wir auch bereits eine weitere Stelle angedeutet, an der Modifikationen notwendig sind: Wenn Visit_Exit aufgerufen wird, kann dies nun für einen kontext-abhängigen oder -unabhängigen Pfad geschehen. Das Unterscheidungsmerkmal ist dabei, ob der Startknoten des ankommenden Pfades ein Eingang ist oder nicht. Handelt es sich um einen Eingang und damit um einen kontext-abhängigen Pfad, erfolgt weiterhin wie im Original das Anlegen der Summary-Kanten in den Aufrufern sowie deren Nutzung, um dort bis zum Unterausgang reichende Same-Level-Pfade zu verlängern.

Bei einem kontext-unabhängigen Pfad jedoch geschieht dies nicht. Stattdessen erfolgt die Propagierung zu den Aufrufern, indem dort jeweils ein neuer Same-Level-Pfad angelegt wird. Sei V, v der Endknoten des aktuellen Same-Level-Pfades an einem Ausgang V. Im ISSA-Graphen gibt es dann für jede Aufrufstelle einen interprozeduralen Übergang $V \rightarrow S$ zu einem Untereingang S. Im erweiterten ISSA-Graphen gehören Kanten aus der Transferfunktion zu diesem Rücksprung. Sei $V, v \rightarrow S, s$ eine solche Kante. Dann wollen wir modellieren, dass an S nun der Fakt s auf einem gültigen (kontext-unabhängigen) Pfad vorliegt. Daher erzeugen wir den Same-Level-Pfad $S, \lambda \rightarrow S, s$. Außerdem müssen wir jedoch auch darauf folgende Gen-Aktionen unterstützen, d.h. wir müssen auch modellieren, dass λ an S vorliegt. Da jedoch in jeder Transferfunktion, also auch der für den Rücksprung, die Kante $\lambda \rightarrow \lambda$ vorliegt, ist hierfür keine zusätzliche Aktion notwendig: Dieser Fall wird behandelt, wenn das Verfahren V mit $v = \lambda$ betrachtet (und das geschieht, wenn mindestens ein v an V betrachtet wird), weil wir dann die Kante $V, \lambda \rightarrow S, \lambda$ finden.

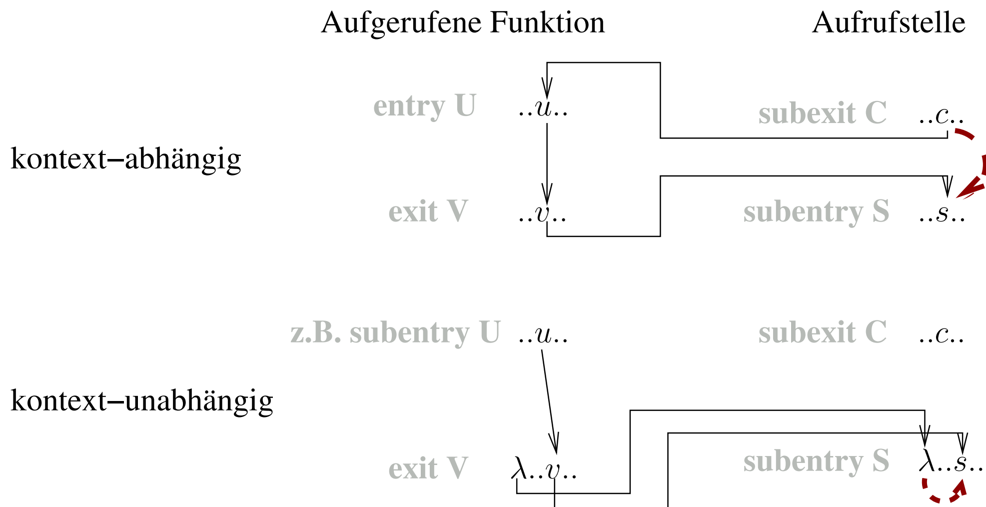


Abbildung 9.8: Behandlung kontext-(un)abhängiger Pfade

Abbildung 9.8 illustriert die Behandlung der kontext-unabhängigen Pfade im Unterschied zu den kontext-abhängigen Pfaden. Wir sehen darin für beide Fälle links eine schematische Darstellung der Situation in der aufge-

rufenen Funktion, in der ein Pfad zum Ausgang V mit dem Fakt v vorliegt. Ferner sehen wir auf der rechten Seite die Auswirkungen auf eine Aufrufstelle an diese Funktion. Im oben gezeigten kontext-abhängigen Fall entsteht eine gestrichelt und als Bogen gezeichnete Summary-Kante zwischen dem Fakt c am Unterausgang C und dem Untereingang S mit Fakt s. Sie repräsentiert den Pfad des erweiterten ISSA-Graphen, der durch die aufgerufene Funktion führt. Im kontext-unabhängigen Fall dagegen entsteht der ebenfalls gestrichelt und als Bogen gezeichnete Pfad $S, \lambda \rightarrow S, s$.

Abbildung 9.9 zeigt die Hauptfunktion des so modifizierten Tabulationsverfahrens, und Abbildung 9.10 ergänzt die beiden Hilfsfunktionen für interprozedurale Übergänge. Die beschriebene Modifikation ist gut zu erkennen, wobei die größten Veränderungen in Visit_Exit nötig sind. Wir haben in dieser Formulierung auch die Unterstützung für indirekte Aufrufe, d.h. für mehrere Ziele an einer Aufrufstelle, in Visit_Call explizit gemacht.

Damit steht nun ein allgemeines Framework zur Verfügung, mit dem man Datenfluss-Probleme auf dem ISSA-Graphen mit MOVP-Präzision lösen kann. Für uns interessant ist die Anwendung auf den Zwischenresultaten, die ebenfalls bereits ISSA-Graphen darstellen und damit dem Framework zugänglich sind. Auf diese Weise können wir pro Iteration Datenfluss-Probleme mit MOVP-Präzision lösen und dies in die Analyse integrieren.

9.4 IFDS für den Propagierungs-Schritt

Unser Fokus liegt nun auf der Anwendung des übertragenen Frameworks für eine präzisere Propagierung der Zeigerziele. Dazu spezifizieren wir die Propagierung als Datenfluss-Problem auf dem ISSA-Graphen. Als Fakten benutzen wir hierzu in einer Iteration die bis dahin bekannten Zeigerliterale (diese Menge kann im Laufe der Iterationen anwachsen).

Für die Transferfunktionen müssen wir bedenken, dass wir nun auf dem ISSA-Graphen operieren, nicht auf dem ICFG. So entspricht z.B. die Zuweisung `p = q` im ISSA-Graphen zwei mit einer Kante verbundenen Knoten. Da hier schlicht die Ziele der RHS an die LHS weitergegeben werden, haben wir als Transferfunktion die Identität. Diese Einfachheit steht in deutlichem

```
procedure ISSA_Tabulation
is
begin
   for all V ∈ ISSA–Nodes, indegree(V) = 0 loop
      Insert V,λ → V,λ into Same_Level_Paths;
   end loop;
   Work_List := Same_Level_Paths;
   Summary_Edges := ∅;
   while Work_List ≠ ∅ loop
      Take any edge E = U,d → V,d' from Work_List
      if Is_Subexit (V) then
         ISSA_Visit_Call (E);
      elsif Is_Exit (V) then
         ISSA_Visit_Exit (E);
      else
         for all V,d' → S,d'' ∈ Exploded_Edges loop
            Propagate (U,d → S,d'');
         end loop;
      end if;
   end loop;
   for all V ∈ Nodes loop
      Facts (V) := {d | ∃d' : U,d' → V,d ∈ Same_Level_Paths};
   end loop;
end ISSA_Tabulation;

procedure Propagate (E : Exploded_Path)
is
begin
   if E ∉ Same_Level_Paths then
      Insert E into Same_Level_Paths;
      Insert E into Work_List;
   end if;
end Propagate;
```

Abbildung 9.9: Das modifizierte Tabulationsverfahren

```
procedure ISSA_Visit_Call (U,d → V,d': Exploded_Path)
is
begin
   -- process down edges
   for all C ∈ Callees (V.Instruction) loop
      for all V,d' → E,d'' ∈ Exploded_Edges, E ∈ Entries(C) loop
         Propagate (E,d'' → E,d'');
      end loop;
   end loop;
   -- inspect summary edges
   for all V,d' → S,d'' ∈ Summary_Edges, S.Instruction = V.Instruction loop
      Propagate (U,d → S,d'');
   end loop;
end ISSA_Visit_Call;

procedure ISSA_Visit_Exit (U,u → V,v: Exploded_Path)
is -- distinguish CI and CS propagation
begin
   if Is_Entry (U) then
      -- concatenate down-same level-up to summary edge
      for all C,c → U,u ∈ Exploded_Edges loop -- down
         for all V,v → S,s ∈ Exploded_Edges,
                  S.Instruction = C.Instruction
         loop --up-edge
            if C,c → S,s ∉ Summary_Edges then
               Insert C,c → S,s into Summary_Edges;
               for all U',e → C,c ∈ Same_Level_Paths loop
                  Propagate (U',e → S,s);
               end loop;
            end if;
         end loop;
      end loop;
   else -- then it is a CI case
      for all V,v → S,s ∈ Exploded_Edges loop
         Propagate (S,λ → S,s); -- to have it CI in caller, too
      end loop;
   end if;
end ISSA_Visit_Exit;
```

Abbildung 9.10: Behandlung interprozeduraler Kanten im modifizierten Tabulationsverfahren

Kontrast zur ICFG-basierten Welt, die an dieser Stelle gerade bei starken Aktualisierungen ihre Distributivität und damit die Anwendbarkeit von IFDS verliert (vgl. Beispiel 2.3.1). In unserer Analyse hat die Umsetzung der Problematik (in)direkter starker Aktualisierungen bereits im Graphaufbau stattgefunden, so dass sie vom Propagierungs-Schritt entkoppelt ist. Insbesondere erlaubt uns dies auch die Nutzung von IFDS für die Propagierung zusammen mit der Unterstützung indirekter starker Aktualisierungen.

Wäre die Zuweisung eine schwache Aktualisierung, so dass neben den Zielen der RHS auch die Ziele weitergereicht werden müssen, die zuvor für p gültig waren, so hätten wir im ISSA-Graphen mit dem Definitionsknoten einen Konfluenzpunkt mit zwei Vorgängern. Das bedeutet, dass hier der Konfluenzoperator angewendet wird, und das ist für unser Problem die Vereinigung. Damit erfordert auch eine schwache Aktualisierung keine andere Transferfunktion. Das Datenfluss-Problem auf dem ISSA-Graphen ist, wie in Kapitel 6 beschrieben, nunmal nichts anderes als ein einfaches Grapherreichbarkeitsproblem.

Lediglich an Stellen, die ein neues Zeigerziel generieren, müssen wir andere Transferfunktionen einsetzen. Diese Stellen sind die Zeigerziel-Quellen, also im Wesentlichen die Adressnahme und Heapallokationen. Diese Stellen entsprechen auch den Knoten, die wir als initiale Same-Level-Pfade betrachten müssen (für eine Vorwärts-Propagierung).

Betrachten wir als Beispiel die Instruktion `x = &y`. Der ISSA-Graph besitzt hier einen Definitionsknoten v für die LHS. Die Transferfunktion muss die Generierung des Fakts `&y` als Minigraph modellieren. Im Beispiel `x = &y` müssten damit die von v ausgehenden ISSA-Kanten annotiert werden. Den zur Transferfunktion gehörigen Graphen zeigt uns Abbildung 9.11. Darin entsteht aus λ der Fakt y, während für die übrigen Ziele $(t_1, \ldots)$ keine Kante benötigt wird. Offensichtlich ist auch diese Art von Transferfunktionen distributiv, wie von IFDS gefordert.

Damit haben wir alles zusammengetragen, um das auf den ISSA-Graphen übertragene Framework für die Propagierung anzuwenden. Das so modifizierte Verfahren löst das beschriebene Datenflussproblem mit MOVP-Präzision. Am Ende können wir in der finalen Liste lokaler gültiger Pfade (bzw. einer

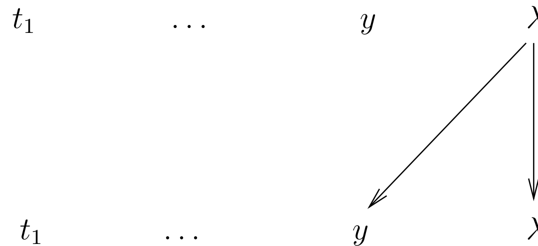


Abbildung 9.11: Transferfunktions-Graph an der Zeigerziel-Quelle

Markierung der Faktenknoten) ablesen, welche Zeigerziele an welchem ISSA-Knoten ankommen, wobei uns natürlich besonders die Dereferenzierungen interessieren. Hier können wir auch unterscheiden, ob ein Fakt auf einem kontext-unabhängigen oder -abhängigen Pfad angekommen ist; entsprechend handelt es sich jeweils um ein kontext-(un)abhängiges Ziel.

Beispiel 9.4.1 Zur Illustration greifen wir erneut das Beispiel aus Abbildung 9.1 auf. Abbildung 9.12 zeigt uns dazu einen im Vergleich zu Abbildung 9.7 größeren Ausschnitt des erweiterten ISSA-Graphen für die Propagierung in der ersten Iteration. Gezeigt ist der Teilgraph für p und q in den Funktionen f, g und h. Der Teilgraph in k ähnelt dem für h, außer dass dort das Zeigerziel y anstelle von x vorkommt; der Übersichtlichkeit halber haben wir ihn deswegen weggelassen.

Die ausgegrauten Knoten und Kanten sind wiederum diejenigen des ISSA-Graphen und nur als Orientierung angegeben. Interprozedurale ISSA-Kanten haben wir der Übersichtlichkeit halber für diesen Graphen nicht gezeigt. Das Verfahren interessiert sich allein für die Knoten und Kanten des erweiterten ISSA-Graphen, welche normal dargestellt sind.

Für diesen Ausschnitt legt die Initialisierung die Kante $def_h(p), \lambda \to def_h(p), \lambda$ als (kontext-unabhängigen) Same-Level-Pfad an. (Zur besseren Unterscheidung indizieren wir im Text die ISSA-Knoten mit der jeweils zugehörigen Funktion.) Die Abarbeitung der Arbeitsliste verlängert diesen zunächst mit den beiden von diesem Knoten ausgehenden Kanten. Deren Endpunkte befinden sich an einem Unterausgang, so dass wir entsprechend den interprozeduralen Übergang realisieren: Die Pfade $entry_g(p), \lambda \to entry_g(p), \lambda$

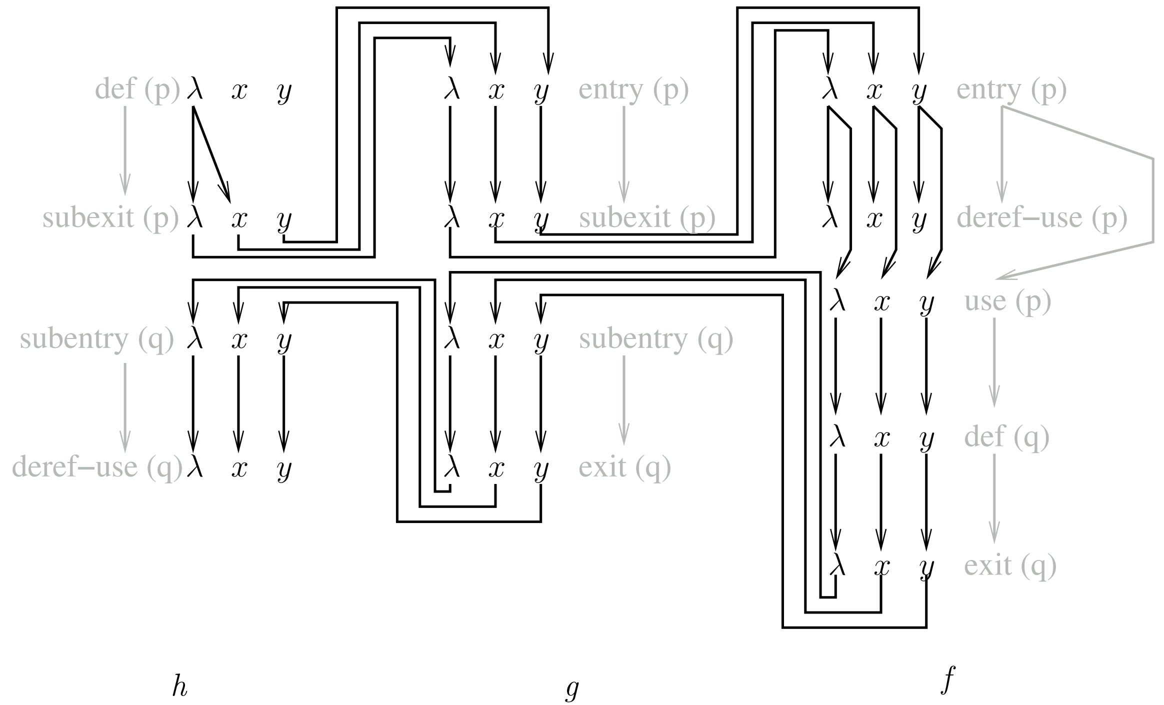


Abbildung 9.12: Erweiterter ISSA-Graph (Ausschnitt zum Beispiel)

sowie $entry_g(p), x \rightarrow entry_g(p), x$ werden angelegt. In der Folge betrachtet der Algorithmus diese Pfade in g und verlängert sie dort. In g erfolgt dann erneut ein Aufruf mit analogem Verhalten wie zuvor. Innerhalb von f geschieht im Anschluss wenig Spannendes, die Transferfunktionen sind nur die Identität und einen Aufruf gibt es nicht. Schließlich ist z.B. der Same-Level-Pfad $entry_f(p), x \rightarrow exit_f(q), x$ an der Reihe. Sein Endknoten ist an einem Ausgang, und der Pfad ist kontext-abhängig. Daher werden an allen Aufrufstellen Summary-Kanten erzeugt (vgl. Abbildung 9.8).

Das Beispiel hat dabei nur eine solche Aufrufstelle (in g) und nur eine Kante, die von einem Knoten am dortigen Unterausgang zum Startpunkt des Same-Level-Pfades führt, d.h. zu $entry_f(p), x$. Entsprechend wird nur die Summary-Kante $subexit_g(p), x \rightarrow subentry_g(q), x$ angelegt. Diese verlängert dabei auch den Pfad $entry_g(p), x \rightarrow subexit_g(p), x$ zu $entry_g(p), x \rightarrow subentry_g(q), x$. Daraus wird später der Pfad $entry_g(p), x \rightarrow exit_g(q), x$. Dieser wiederum erzeugt an den Aufrufstellen Summary-Kanten: Sowohl in h

als auch in k wird eine solche zwischen dem Unterausgang für p und dem Untereingang für q angelegt (jeweils für x als Fakt) – aber eben keine Verbindung zwischen beiden Funktionen. Schließlich gelangen wir damit zum Pfad $def_h(p), x \rightarrow deref\text{-}use_h(q)$, so dass an der Dereferenzierung in h das Ziel x kontext-unabhängig auftaucht (analog y in k). Die höhere Präzision zeigt sich darin, dass jede dieser Dereferenzierungen im Unterschied zur kontext-insensitiven Propagierung jeweils nur ein Ziel erhält. □

9.5 Ein alternativer Drei-Stufen-Ansatz

Wir haben nun gesehen, wie man das allgemeine Framework für eine präzisere Propagierung instantiieren kann. Nun wollen wir als Alternative dazu ein weiteres Verfahren vorstellen, welches die Einfachheit des Datenflussproblems ausnutzt, um asymptotisch effizienter als die Instantiierung zu werden (vgl. Kapitel 10 für die Laufzeitabschätzungen). Dazu schildern wir in diesem Abschnitt unsere Idee und geben den resultierenden Pseudo-Code an. Abschließend werden wir auch betrachten, mit welchen Datenstrukturen sich dieses alternative Verfahren umsetzen lässt.

Der alternative Ansatz zerlegt das Problem in drei Teile: Die Bestimmung der Same-Level-Pfade sowie die Propagierung entlang kontext-unabhängiger bzw. kontext-abhängiger Pfade. Nur der erste Teil ist mit der MOVP-Problematik befasst, die anderen beiden Teile sind wieder normale Grapherreichbarkeits-Probleme. Damit arbeiten wir den Teil heraus, der sich mit der MOVP-Thematik auseinandersetzen muss. Da die anderen Teile wieder Beschleunigungsideen wie einer Zyklenkontraktion zugänglich sind, dürfte dieser Teil zugleich auch der Flaschenhals in der Praxis sein. Der alternative Ansatz liefert uns neben dieser Herausarbeitung auch den Vorteil, dass eine inkrementelle Version über die Iterationen der Analyse hinweg (und damit wenigstens eine asymptotische Beschleunigung) vergleichsweise einfach zu realisieren ist.

Wie wir im Beispiel gesehen haben, taucht bei der Propagierung lediglich die Identität als Transferfunktion auf. Die davon abweichenden Transferfunktionen an Zeigerziel-Quellen können wir in der Initialisierung beachten. Das erlaubt uns folgendes:

- Wir müssen die Transferfunktionen nicht explizit als Minigraphen darstellen; die Kanten des erweiterten ISSA-Graphen sind damit nur konzeptionell, nicht aber in den Datenstrukturen vorhanden.

- Wir benötigen den Sonderfakt λ nicht, da wie gesagt nur an den Zeigerziel-Quellen die Generierung von Fakten abgebildet werden muss.

Wir wollen noch einen Schritt weitergehen und die Same-Level-Pfade sowie die Summary-Kanten zwischen ISSA-Knoten anstatt zwischen den Knoten des erweiterten ISSA-Graphen anlegen. Grundlage hierfür ist die Beobachtung, dass die Same-Level-Pfade und Summary-Kanten aufgrund der Identität als Transferfunktion für alle Fakten an einem ISSA-Knoten den gleichen Aufbau haben (siehe z.B. Abbildung 9.12). Zur Angabe eines solchen Pfades genügt dem Verfahren dann die Angabe der beiden beteiligten ISSA-Knoten.

In unserem Beispiel würde dies bedeuten, dass wir z.B. bei der Betrachtung der Zeigerziel-Quelle `p=&x` in h den Pfad $def_h(p) \rightarrow def_h(p)$ anlegen. Dieser wird dann bis zum Unterausgang verlängert, so dass die Bestimmung der Same-Level-Pfade in der aufgerufenen Funktion g erfolgt etc. Damit erreichen wir letztlich auch die Dereferenzierungen in f und h.

Das Beispiel kann uns jedoch auch ein Problem mit dem Ansatz illustrieren: Betrachtet der Algorithmus später die Zeigerziel-Quelle `p=&y` in k, so wird irgendwann der Pfad $def_k(p) \rightarrow subexit_k(p)$ abgearbeitet. In der Folge betrachten wir wieder den darüber erreichbaren Eingang von p in der aufgerufenen Funktion g. Doch da die Differenzierung der ISSA-Knoten nach Zeigerzielen weggefallen ist, gilt dieser bereits als besucht – es wird kein neuer Same-Level-Pfad in Propagate angelegt. Das hat zur Konsequenz, dass wir für das Zeigerziel y nicht zu den schon für x besuchten Dereferenzierungen gelangen! Dieses Problem müssen wir nun noch beheben. (Bei den Summary-Kanten dagegen gibt es kein analoges Problem.)

Unsere Lösung hierfür ist ein dreistufiger Ansatz: Die erste Stufe bestimmt die Same-Level-Pfade und Summary-Kanten wie beschrieben zwischen ISSA-Knoten. Insbesondere dank der Summary-Kanten können wir dann im Anschluss die eigentliche Propagierung in zwei weitere Stufen auf-

trennen: Stufe zwei propagiert ausschließlich entlang kontext-unabhängiger Pfade („strikt nach oben" im Aufrufgraphen), während Stufe drei ausschließlich entlang kontext-abhängiger Pfade vorgeht („strikt nach unten" im Aufrufgraphen). Durch die Beschränkung auf jeweils eine interprozedurale Richtung sind diesen beiden Propagierungsstufen damit von der MOVP-Problematik entkoppelt und können mit den schon bekannten Möglichkeiten für die Propagierung aus Kapitel 6 implementiert werden.

Die folgenden Abschnitte schildern die drei Stufen dieses Ansatzes, dessen Grundideen einige Parallelen zum interprozeduralen Slicing von Horwitz u. a. [1988] aufweisen. Wir beschreiben das Vorgehen für eine Vorwärtspropagierung in der zweiten Stufe (also von Zeigerziel-Quellen ausgehend) und eine Rückwärtspropagierung (von Dereferenzierungen ausgehend) in der dritten Stufe. Andere Kombinationen sind ebenfalls denkbar. In Abschnitt 9.5.4 werden wir die hier beschriebenen Ideen anhand unseres Beispiels illustrieren.

9.5.1 Erste Stufe: Same-Level-Pfade

Besprechen wir zunächst die erste Stufe, d.h. die Berechnung der Same-Level-Pfade und Summary-Kanten für den ISSA-Graphen. Abbildung 9.13 zeigt dazu den Pseudo-Code, und Abbildung 9.14 ergänzt die Hilfsfunktionen zur Behandlung interprozeduraler Übergänge. Der Ursprung im IFDS-Framework ist klar zu erkennen. Der Unterschied besteht wie erwähnt in der Vereinfachung der Kanten und Pfade zu solchen zwischen (nicht erweiterten) ISSA-Knoten. Ansonsten ist der Code direkt demjenigen des auf ISSA übertragenen Frameworks entnommen (vgl. Abbildungen 9.9 und 9.10).

Die Abbildungen zeigen die Berechnung aller Summary-Kanten und Same-Level-Pfade. Für die folgenden beiden Propagierungsstufen benötigen wir sodann jeweils nur einen Teil der berechneten Same-Level-Pfade zusammen mit interprozeduralen ISSA-Kanten. Wir nutzen hier also die Same-Level-Pfade als Resultat der ersten Stufe und ergänzen dazu ein paar ISSA-Kanten; alternativ könnte man auch von einem Teil des ISSA-Graphen ausgehen und in diesem die Summary-Kanten ergänzen.

```
procedure ISSA_Summaries is
begin
   for all D ∈ Sources loop
      -- D is a general definition
      Insert D → D into Same_Level_Paths;
   end loop;
   Work_List := Same_Level_Paths;
   Summary_Edges := ∅;
   while Work_List ≠ ∅ loop
      Take any path E = U → V from Work_List;
      if Is_Subexit (V) then
         ISSA_Summaries_Visit_Call (E);
      elsif Is_Exit (V) then
         ISSA_Summaries_Visit_Exit (E);
      else -- same-level edge
         for all V → S ∈ Edges loop
            Propagate (U → S);
         end loop;
      end if;
   end loop;
end ISSA_Summaries;

procedure Propagate (E = U → V : Path) is
begin
   if E ∉ Same_Level_Paths then
      Insert E into Same_Level_Paths;
      Insert E into Work_List;
   end if;
end Propagate;
```

Abbildung 9.13: Berechnung von Same-Level-Pfaden und Summary-Kanten für den ISSA-Graphen mittels der IFDS-Ideen

```
procedure ISSA_Summaries_Visit_Call (U → V: Path)
is
begin
   -- process down edges
   for all V → E ∈ Edges loop -- E is entry
      Propagate (E → E);
   end loop;
   -- inspect summary edges
   for all V → S ∈ Summary_Edges loop
      Propagate (U → S);
   end loop;
end ISSA_Summaries_Visit_Call;

procedure ISSA_Summaries_Visit_Exit (U → V: Path)
is -- distinguish CI and CS propagation
begin
   if Is_Entry (U) then
      -- concatenate down-same_level-up to summary edge
      for all C ∈ Subexits_To (U) loop --down
         Take S from V → S ∈ Edges, S.Instruction = C.Instruction; --up
         if C → S ∉ Summary_Edges then
            Insert C → S into Summary_Edges;
            for all U' → C ∈ Same_Level_Paths loop
               Propagate (U' → S);
            end loop;
         end if;
      end loop;
   else -- then it is a CI case
      for all V → S ∈ Edges loop
         Propagate (S → S); -- to have it CI in caller, too
      end loop;
   end if;
end ISSA_Summaries_Visit_Exit;
```

Abbildung 9.14: Behandlung interprozeduraler Kanten bei der Berechnung von Same-Level-Pfaden und Summary-Kanten für den ISSA-Graphen

9.5.2 Zweite Stufe: Kontext-unabhängige Propagierung

Die Propagierung entlang kontext-unabhängiger Pfade operiert auf einem Graphen (dem *CI-Graphen*), den wir wie folgt definieren können: Der CI-Graph besitzt als Knoten eine Teilmenge der ISSA-Knoten, und zwar jene, die als Knoten an den Kanten des CI-Graphen vorkommen. Diese Kanten sind dabei einige der interprozeduralen ISSA-Kanten sowie eine Teilmenge der Same-Level-Pfade, die mit dem Algorithmus aus den Abbildungen 9.13 und 9.14 berechnet wurden. Einen Same-Level-Pfad $U \rightarrow V$ nehmen wir als Kante auf, wenn U eine Quelle oder ein Untereingang ist und V eine Senke oder ein Ausgang. Eine interprozedurale ISSA-Kante $U \rightarrow V$ nehmen wir auf, wenn U ein Ausgang und V ein Untereingang ist (das lässt sich natürlich beschränken auf solche Kanten, bei denen mindestens einer dieser Knoten durch einen Same-Level-Pfad oder diese Regel selbst bereits im CI-Graphen enthalten ist).

Die erste Stufe hat bei der Bestimmung der Same-Level-Pfade bereits die Summary-Kanten beherzigt. Diese wiederum bilden eine Zusammenfassung in Situationen, in denen der Datenfluss eigentlich zwischendurch im Aufrufgraphen absteigt (zu einer gerufenen Funktion) und erst später wieder zur Aufrufstelle zurückkehrt. Dadurch, dass solch ein Intermezzo also bereits bei der Konstruktion der Same-Level-Pfade bedacht wurde, können wir uns nun rein auf die Flussrichtung von einer Funktion zu ihren Aufrufern (sowie innerhalb einer Funktion) konzentrieren.

Die Propagierung kann nun wie bereits erwähnt mit den schon bekannten Möglichkeiten aus Kapitel 6 implementiert werden, diesmal auf dem CI-Graphen. Auch eine Zyklenkontraktion kann hier zur Beschleunigung zum Einsatz kommen, da wieder ein normales Grapherreichbarkeitsproblem vorliegt. Dabei wollen wir die Zeigerziele nun nicht nur zu den Dereferenzierungen propagieren, sondern auch zu den Untereingängen, denn von diesen können möglicherweise Verbindungen zu kontext-abhängigen Pfaden ausgehen. Das wird in der dritten Stufe respektiert.

9.5.3 Dritte Stufe: Kontext-abhängige Propagierung

Die Propagierung entlang kontext-abhängiger Pfade operiert passend dazu auf dem *CS-Graphen*, der über die folgende Kantenmenge (und die darüber erfassten Knoten) definiert ist:

- Einen Same-Level-Pfad $U \to V$ nehmen wir als Kante auf, wenn U eine Quelle, ein Eingang oder ein Untereingang ist und V eine Senke oder ein Unterausgang.

- Eine interprozedurale ISSA-Kante $U \to V$ nehmen wir auf, wenn U ein Unterausgang und V ein Eingang ist.

Hier beachten wir auch Verbindungen zwischen Untereingängen und Senken bzw. Unterausgängen. Damit können wir an den Untereingängen die in der zweiten Stufe dorthin propagierten kontext-unabhängigen Ziele abgreifen.

Die Propagierung kann nun ebenfalls mit den schon bekannten Möglichkeiten aus Kapitel 6 implementiert werden, diesmal auf dem CS-Graphen. Als Rückwärtspropagierung werden hier beispielweise die Dereferenzierungen auf den umgedrehten Kanten des CS-Graphen propagiert. Erreichen sie eine Quelle oder einen Untereingang mit einem Ziel aus der zweiten Stufe, so haben wir ein neues Ziel für die Dereferenzierungen gefunden.

9.5.4 Anwendung auf das Beispiel

Zur Illustration der Ideen wenden wir das so beschriebene alternative Verfahren auf das Beispiel aus Abbildung 9.1 an. In Abbildung 9.12 hatten wir dazu den erweiterten ISSA-Graphen gesehen. Dieser ist nun nicht mehr notwendig, da in der ersten Stufe des alternativen Verfahrens Same-Level-Pfade und Summary-Kanten zwischen ISSA-Knoten ermittelt werden. Abbildung 9.15 zeigt uns wieder einen Ausschnitt des zum Beispiel gehörenden ISSA-Graphen nach dem Graphaufbau in der ersten Iteration. Diesmal haben wir auch den Teilgraphen in der Funktion k aufgenommen und die interprozeduralen ISSA-Kanten eingezeichnet. Die gestrichelten Kanten stellen die in der ersten Stufe berechneten Summary-Kanten dar.

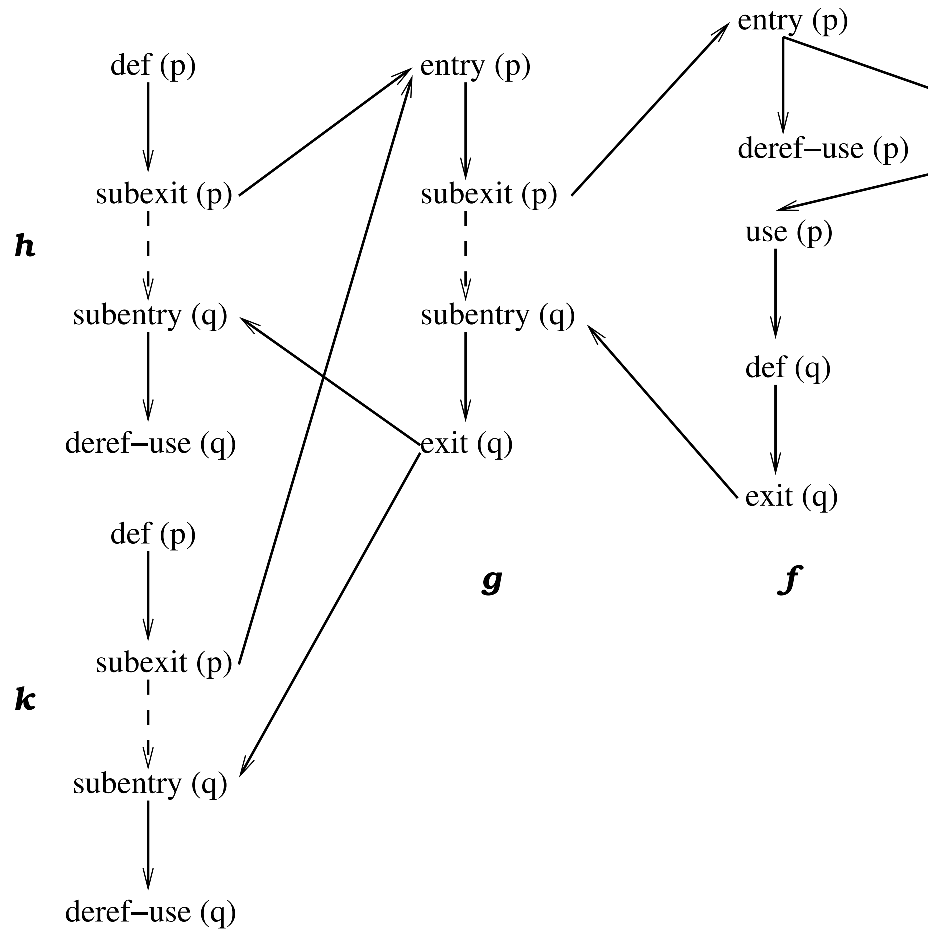


Abbildung 9.15: ISSA-Graph für p und q zum Beispiel. Summary-Kanten sind gestrichelt eingezeichnet.

An dieser Abbildung können wir auch noch einmal das Problem verdeutlichen, warum nicht gleich in der ersten Stufe die Zeigerziele propagiert werden: Nehmen wir wieder an, die Quelle in h wird zuerst beachtet. Sie erzeugt dann alle Same-Level-Pfade und Summary-Kanten in g und f. Wenn anschließend die Quelle in k zu Same-Level-Pfaden führt, so ist $entry_g(p)$ schon behandelt, so dass dort keine Neuberechnung von Pfaden erfolgt. Für die Summary-Kanten ist dies unproblematisch: Diese wurden auch bereits für die Aufrufstelle in k angelegt, weil die Aktionen an einem Ausgang alle Aufrufer betrachten. Dadurch, dass jedoch keine Same-Level-Pfade in g (und dann in f) neu betrachtet werden, gibt es hier auch noch kein Besuch an der Dereferenzierung in f von der Quelle in k ausgehend.

Der CI-Graph für die zweite Stufe in diesem Beispiel enthält die beiden Same-Level-Pfade $def(p) \rightarrow deref\text{-}use(q)$ in den Funktionen h und k. Zusätzlich werden noch in diesen beiden Funktionen die Pfade $subentry(q) \rightarrow deref\text{-}use(q)$ aufgenommen sowie einige weitere Pfade; diese sind jedoch alle nicht von den Quellen aus erreichbar und darum uninteressant. Mit dem CI-Graphen wird das Zeigerziel x an die Dereferenzierung in h und das Zeigerziel y an die Dereferenzierung in k propagiert.

Einen größeren erreichbaren Teil dagegen besitzt der CS-Graph, den wir in Abbildung 9.16 sehen. Wir können hierbei klar die Fokussierung auf die Datenfluss-Richtung vom Aufrufer zum Aufgerufenen erkennen. Auch die Quellen sind in diesem Fall wieder im Graphen enthalten, da sie eine Verbindung zu einem Unterausgang aufweisen. Die Propagierung entlang dieses Graphen wird nun feststellen, dass es einen Weg von beiden Quellen zur Dereferenzierung in f gibt, und entsprechend dort die Ziele eintragen.

9.5.5 Inkrementelle Propagierung

Bislang haben wir ein erschöpfendes Verfahren besprochen, welches in jeder Iteration der Analyse erneut die drei Stufen vollständig ausführt. Nun wollen wir analog zum kontext-insensitiven Fall eine dazugehörige inkrementelle Variante vorstellen. Diese soll die Gesamtlaufzeit senken, indem sie die einmal berechneten Informationen später wiederverwendet.

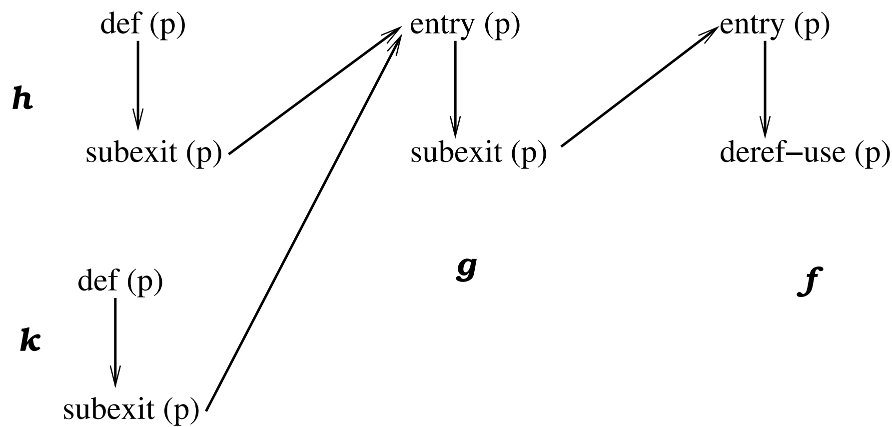


Abbildung 9.16: CS-Graph für p und q zum Beispiel

Die wesentliche Aufgabe hierfür ist eine inkrementelle Fassung der ersten Stufe, da wir bereits aus Kapitel 6 wissen, wie wir die anschließenden Propagierungsstufen inkrementell gestalten können.

Als Eingabe erhält die erste Stufe im inkrementellen Szenario die Menge der in der jeweiligen Iteration neu erkannten ISSA-Kanten. Zusätzlich nehmen wir an, dass von der vorherigen Iteration die Same-Level-Pfade und Summary-Kanten aufbewahrt wurden. Nun sollen die zusätzlich vorhandenen Same-Level-Pfade und Summary-Kanten bestimmt werden.

Betrachten wir dazu eine neue ISSA-Kante $A \to B$, wobei A entweder eine Zeigerziel-Quelle oder bereits in der letzten Iteration in der Bestimmung der Same-Level-Pfade besucht worden sein soll. Wir wollen zu dieser Kante nun ermitten, welche initialen Same-Level-Pfade wir in die Arbeitsliste legen müssen, damit das schon bekannte Verfahren nach dieser veränderten Initialisierung wie gewohnt ablaufen kann.

Handelt es sich bei A um eine Zeigerziel-Quelle, so können wir schlicht $A \to B$ in die initiale Arbeitsliste sowie die Menge der Same-Level-Pfade legen. Andernfalls hat A, da es in der vorigen Iteration besucht wurde, eine eingehende ISSA-Kante. Dies kann auch dann der Fall sein, wenn A eine Zeigerziel-Quelle ist. Wir unterscheiden lokale und interprozedurale Fälle:

Neue intraprozedurale Kante. Im Falle einer intraprozeduralen Kante nutzen wir die bislang bekannten Same-Level-Pfade der vorigen Iteration, die in A enden. Für jeden solchen Pfad $U \to A$ legen wir nun $U \to B$ in die initiale Arbeitsliste sowie die Menge der Same-Level-Pfade.

Neue interprozedurale Kante: A ist ein Ausgang. Wenn A ein Ausgang und entsprechend B ein Untereingang ist, müssen wir die Aktionen für einen Ausgang für diesen Untereingang nachholen. Wieder betrachten wir dazu die Same-Level-Pfade der vorigen Iteration, die in A enden. Für jeden solchen Pfad $U \to A$ führen wir dann die Aktionen aus der Funktion ISSA_Summaries_Visit_Exit aus, allerdings nur für die Aufrufstelle von B. Das legt potentiell neue Summary-Kanten an dieser Aufrufstelle an und kann im Aufrufer neue Same-Level-Pfade hervorrufen.

Neue interprozedurale Kante: A ist ein Unterausgang. Wenn A ein Unterausgang und entsprechend B ein Eingang ist, müssen wir die Aktionen für einen Aufruf mit diesem Untereingang nachholen. Wie in der Funktion ISSA_Summaries_Visit_Call rufen wir daher $Propagate(B \to B)$ auf. Für die Beachtung schon existierender Summary-Kanten verlängern wir dann die Same-Level-Pfade der vorigen Iteration, die in A enden. Jeden solchen Pfad $U \to A$ verlängern wir mit den Summary-Kanten $A \to S$ und rufen entsprechend $Propagate(U \to S)$ auf.

Die so beschriebenen Initialisierungsaktionen füllen auf Basis der neu aufgebauten ISSA-Kanten die initiale Arbeitsliste und erstellen bereits neue Same-Level-Pfade und Summary-Kanten. Daran anschließend lassen wir nun das bekannte Verfahren aus den Abbildungen 9.13 und 9.14 ablaufen (d.h. ohne die dortige Initialisierung). Es wird die Arbeitsliste abarbeiten und dabei ggf. weitere Same-Level-Pfade und Summary-Kanten erstellen. Die aus der vorigen Iteration bekannten Pfade und Summary-Kanten werden dabei beachtet, so dass bereits bekannte Informationen nicht noch einmal berechnet werden. Abbildung 9.17 zeigt uns abschließend den Pseudo-Code für die inkrementelle Initialisierung.

```
procedure Incremental_ISSA_Summaries is
begin
   for all A → B ∈ New_Edges loop
      if A ∈ Sources then
         Propagate (A → B);
      end if;
      if Is_Subexit (A) then -- new down-edge
         Propagate (B → B);
         for all U → A ∈ Same_Level_Paths loop
            for all A → S ∈ Summary_Edges loop
               Propagate (U → S);
            end loop;
         end loop;
      elsif Is_Exit (A) then -- new up-edge
         for all U → A ∈ Same_Level_Paths loop
            if Is_Entry (U) then
               Take C from C ∈ Subexits_To (U),
                  C.Instruction = B.Instruction;
               if C → B ∉ Summary_Edges then
                  Insert C → B into Summary_Edges;
                  for all U' → C ∈ Same_Level_Paths loop
                     Propagate (U' → B);
                  end loop;
               end if;
            else -- then it is a CI case
               Propagate (B → B);
            end if;
         end loop;
      else
         for all U → A ∈ Same_Level_Paths loop
            Propagate (U → B);
         end loop;
      end if;
   end loop;
   while Work_List ≠ ∅ loop
      ...
   end loop;
end Incremental_ISSA_Summaries;
```

Abbildung 9.17: Inkrementelle Initialisierung der Berechnung von Same-Level-Pfaden und Summary-Kanten für den ISSA-Graphen

9.5.6 Datenstrukturen

Die Datenstrukturen für den Algorithmus der ersten Stufe für die Arbeitsliste, die Same-Level-Pfade und die Summary-Kanten sollten ein schnelles Einfügen erlauben. Abgesehen von der Arbeitsliste benötigen wir außerdem einen schnellen Enthaltenseins-Test.

Die Arbeitsliste kann eine einfach verkettete Liste sein, ggf. als wachsendes Array realisiert. Man kann sie auch gänzlich ersetzen durch eine Umstellung des Verfahrens auf eine rekursive Tiefensuche anstelle der Schleife.

Für die Same-Level-Pfade schlagen wir vor, einen Pfad $U \to V$ dadurch zu speichern, dass wir an V den Knoten des eingehenden Pfades vermerken. Als zu vermerkende Knoten U kommen dabei die lokal vorhandenen Eingänge sowie die lokalen Startpunkte kontext-unabhängiger Pfade in Frage. Die lokalen Eingänge können wir numerieren und die Nummer als einen Array-Index verwenden. Damit können wir an jedem V ein boolesches Array anlegen, in dem ein gesetzter Eintrag an Index i bedeutet, dass es einen Same-Level-Pfad vom Eingang mit der Nummer i zu V gibt. Analog kann man die Startpunkte der kontext-unabhängigen Pfade numerieren (also lokale Quellen und Untereingänge). Die Verwendung des Arrays erlaubt dabei schnelle Operationen in konstanter Zeit.

Für die Summary-Kanten können wir analog an jeder Aufrufstelle die Untereingänge numerieren und damit an den Unterausgängen jeweils ein Array mitführen. Hier speichern wir die Summary-Kante $V \to S$ also am Knoten V, weil der Algorithmus zur Behandlung eines Ausgangs nur diesen kennt und die davon ausgehenden Summary-Kanten erfahren muss.

9.6 Kontext-(un)abhängige Zeigerziele

Die Unterscheidung in kontext-(un)abhängige Ziele haben wir bislang nicht ausgenutzt. Dies wäre jedoch zumindest auf zweierlei Arten möglich:

1. Kontext-unabhängige Ziele können wir für eine feinere Modellierung des Heaps benutzen.

2. Kontext-abhängige Ziele können wir für eine kontext-abhängige Datenfluss-Modellierung nutzen.

Beides soll hier nur kurz skizziert werden, um interessante Möglichkeiten für weitergehende Arbeiten anzudeuten. Nachdem wir mit dem Einsatz des übertragenen IFDS-Frameworks bereits die eingangs erwähnte Problematik der Aufwärts-Kontext-(In)Sensitivität adressiert haben, erlauben diese beiden Punkte nun Verbesserungen bezüglich der anderen Ungenauigkeiten.

9.6.1 Heapmodellierung

Für die Heapmodellierung stellen wir fest, dass wir nun Situationen erkennen, in denen das bislang für eine Allokationsstelle benutzte Analyse-Objekt in mehrere Objekte verfeinert werden kann: Immer dann, wenn an einem Ausgang als Fakt ein Heapobjekt erkannt wird und dabei ein kontext-unabhängiges Ziel darstellt, haben wir das Ende einer Funktion erreicht, die aus Sicht des Aufrufers das Heapobjekt allokiert (und direkt oder über nicht-lokale Variablen etc. zurückgibt). Betrachten wir dazu ein Beispiel aus Kapitel 3 erneut:

```
x = (int)malloc_wrapper();
*x = 5;
y = (int)malloc_wrapper();
z = *y;
printf (*x);
```

Nehmen wir hierzu der Einfachheit halber folgenden Code für `malloc_wrapper` an:

```
void* malloc_wrapper
{
    return malloc (...);
}
```

Die bisherige Analyse erzeugt hier lediglich ein Analyse-Objekt. Nun können wir jedoch erkennen, dass dieses kontext-unabhängig am ISSA-Ausgangsknoten für die Rückgabe von `malloc_wrapper` ankommt. Entsprechend können

wir die Propagierung an den Aufrufstellen mit jeweils unterschiedlichen Analyse-Objekten fortsetzen, die eine Verfeinerung des ursprünglichen Objekts darstellen. Auf diese Weise zeigen x und y auf zwei verschiedene Objekte anstatt auf ein gemeinsames Objekt.

Für diese Verfeinerung ist es nötig, kontext-unabhängige Ziele zu erkennen, da die Verfeinerung nicht für ein kontext-abhängiges Ziel stattfinden darf: Dieses wird an der Aufrufstelle z.B. als Parameter übergeben, in der gerufenen Funktion schwach aktualisiert, und dann wieder an den Aufrufer zurückgegeben. Hier handelt es sich noch um das gleiche Objekt, während wir im obigen Beispiel sicher sagen konnten, dass die beiden Aufrufe in einer Ausführung verschiedene Objekte zurückgeben.

Die Verfeinerung an den Aufrufstellen lässt sich nun mit der Erkennung kontext-unabhängiger Ziele anpacken. Die Idee lässt sich dabei auch transitiv fortsetzen: Oben haben wir für eine Verfeinerung nur Aufrufe an die Funktion betrachtet, welche ihrerseits die Allokation enthält. Das Schema ist jedoch auch bei transitiven Aufrufern anwendbar, d.h. wenn der oben gezeigte Code z.B. x zurückgibt, dann können wir das Heapobjekt, auf welches x zeigt, an allen Aufrufstellen dazu weiter verfeinern.

9.6.2 Kontext-abhängiger Datenfluss

Die Analyse behandelt bislang an einer Dereferenzierung alle dort ankommenden Zeigerziele auf die gleiche Weise. Nun können wir jedoch für Zeigerziele, die aus einem Aufrufer stammen, statt des normalen Datenfluss-Knotens eine kontext-sensitive Modellierung anstreben. Die Idee hierfür besteht darin, bei einem kontext-abhängigen Ziel, welches über den Same-Level-Pfad $U, d \rightarrow V, d$ zur Dereferenzierung V gelangte, den Eingang U im Datenfluss-Knoten zu beachten, der hier normalerweise für d angelegt würde. (Der Einfachheit halber nutzen wir hier wieder die Beschreibung mit Same-Level-Pfaden im erweiterten ISSA-Graphen.) Dadurch, dass wir die Herkunft auf diese Weise mitführen, können wir die Kontext-Abhängigkeit im Datenfluss ausdrücken.

Greifen wir dazu das Beispiel aus Abbildung 9.1 wieder auf. Darin gab es in der Funktion f die Mehrfachdereferenzierung `**p = 1`, wobei p eine globale Variable ist. Deren Zeigerziel stammt aus dem Aufrufer, so dass in f im ISSA-Graphen ein Eingang U für p existiert. Bezeichnen wir die Verwendung von p in obiger Anweisung mit V, so berechnet das Verfahren den Same-Level-Pfad $U, x \to V, x$ für die Quelle aus h, während für die Quelle aus k der Pfad $U, y \to V, y$ erscheint. Bislang wurde nun für beide Ziele jeweils ein Datenfluss-Knoten angelegt und die Kontext-Unterscheidung damit verwischt.

Die Idee für eine kontext-sensitive Modellierung in diesem Beispiel besteht nun darin, den Eingang U selbst als (kontext-abhängiges, abstraktes) Objekt zu verwenden, und für dieses Objekt einen Datenfluss-Knoten anzulegen (anstelle der beiden Knoten für x und y). Auch dafür können wir dann den Graphaufbau lokal betreiben, d.h. ϕ-Knoten und dergleichen für das abstrakte Objekt anlegen. An den interprozeduralen Übergängen (d.h. bei der Propagierung von Seiteneffekten) muss eine Übersetzung stattfinden, so dass im Aufrufer jeweils wieder x bzw. y statt U benutzt wird.

Dieser einfache Grundgedanke muss jedoch für eine korrekte Analyse noch ausgebaut werden:

- Zum Einen müssen wir Aliasing beachten. Gäbe es z.B. in f auch noch direkte Definitionen oder Verwendungen von x, so müssten diese im Graphaufbau mit den Knoten für U (welches ja auch x bezeichnen kann) verbunden werden, damit kein potentieller Datenfluss übersehen wird. Ein Ansatz hierzu ist es, Ziele von U, die lokal unter Aliasing leiden, mit zusätzlichen Knoten (neben denen für U) zu behandeln. Zur Reduktion des Analyse-Aufwands kann auch die Zahl der noch wirklich genau behandelten verschiedenen Aliasing-Konstellationen begrenzt werden, so dass ab z.B. 5 verschiedenen Konstellationen für alle weiteren Fälle auf ein Worst-Case-Szenario zurückgegriffen wird.

- Zum Anderen kann auch im Aufrufer als korrektes Pendant für U wieder ein abstraktes Objekt (anstatt x bzw. y) vorkommen. In unserem Beispiel ist dies der Fall, weil das Zeigerziel für p nicht aus den direkten

Aufrufern von f stammt, sondern von deren Aufrufern. Ein Seiteneffekt auf U in f muss daher in g als Seiteneffekt auf den zugehörigen Eingang U_g abgebildet werden.

Eine Erweiterung der Analyse, welche in diese Richtung vorstößt, erhält damit neben der präziseren Propagierung auch eine präzisere Datenfluss-Modellierung. Die Darstellung mit kontext-abhängigen, abstrakten Objekten kann hierbei sogar das Mengengerüst verkleinern, weil alle Ziele, die über U nach f gelangen, im Alias-freien Fall mit einem gemeinsamen Knoten auskommen.

Teil II

Evaluation

Kapitel 10

Theoretische Evaluation

In diesem Kapitel analysieren wir das vorgeschlagene Verfahren. Wir beweisen die Korrektheit des Vorgehens und bestimmen asymptotische Schranken für Laufzeit und Speicherbedarf, sowohl für die Kernanalyse ohne als auch mit der speziellen Unterstützung für indirekte und zusammengesetzte starke Aktualisierungen. Auch die im vorigen Kapitel angerissene Möglichkeit, IFDS anzuwenden, werden wir am Ende kurz untersuchen. Damit beweist dieses Kapitel die Thesen der Dissertation. Die nachfolgenden Kapitel dieses Teiles vervollständigen das Bild mit einer empirischen Evaluation.

10.1 Terminierung und Korrektheit

Die im Wechsel ausgeführten Schritte der Zeiger- und der Datenfluss-Analyse greifen auf noch nicht konservative Zwischenresultate der jeweils anderen Analyse zurück. Daher müssen wir uns davon überzeugen, dass im Fixpunkt schließlich tatsächlich korrekte – d.h. konservativ überschätzte – Resultate vorliegen. Außerdem sollten wir sicherstellen, dass die Analyse den Fixpunkt auch erreicht. Darum zeigen wir in diesem Abschnitt nun die Terminierung und Korrektheit der kombinierten Analyse. Da beim Umgang mit noch nicht konservativen Resultaten stets Vorsicht geboten ist, stellt der Korrektheitsbeweis einen zentralen Teil der Dissertation dar. Wir geben ihn daher in einiger Detailliertheit und formal an.

10.1.1 Terminierung

Der von der Analyse iterativ berechnete IDFG beginnt mit dem (endlich großen) direkt ermittelbaren IDFG. Anschließend wächst die Zahl der Knoten und Kanten darin von Iteration zu Iteration monoton: Knoten werden nur ergänzt, und Kanten entweder ergänzt oder in dem Sinne verfeinert, dass eine bestehende Kante einen anderen Startknoten erhält. Dies gilt sowohl in der Ausprägung der Analyse, die indirekte Definitionen stets als schwache Aktualisierungen betrachtet, als auch in der Ausprägung mit der Unterstützung für indirekte starke Aktualisierungen. Der vollständige Graph, der jedes Objekt als potentielles Zeigerziel ansieht (und entsprechende Knoten an Dereferenzierungen anlegt) und alle möglichen Kanten enthält, ist sicherlich eine endliche obere Schranke. Gleiches gilt für den ICFG. Spätestens an dieser oberen Schranke kommt die Analyse zu einem Fixpunkt, womit die Terminierung gesichert ist.

10.1.2 Korrektheit

Sei P das zu analysierende Programm und $\mathcal{E}$ eine beliebige Ausführung von P. Wir wollen zeigen, dass die Analyse für $\mathcal{E}$ eine korrekte Lösung bestimmt. Damit ergibt sich insgesamt auch die Aussage, dass die Analyse korrekte Resultate für alle Ausführungen von P und damit für P selbst liefert.

10.1.2.1 Annahmen

Um zunächst den Korrektheitsbeweis zu vereinfachen, gehen wir von einigen Voraussetzungen bezüglich der Ausführung und der verwendeten Programmkonstrukte aus. Später diskutieren wir, wie einige dieser Einschränkungen aufgehoben werden können.

Anforderungen zu den im Programm verwendeten Konstrukten. Im Wesentlichen nehmen wir eine IR wie in Kapitel 3 beschrieben an. Dies betrifft insbesondere auch die Modellierung der Zeigerarithmetik als Identität, d.h. für die Korrektheit in diesem Punkt gehen wir davon aus, dass Zeigerarithmetik nur für die Traversierung eines Strings oder allgemeinen

Arrays innerhalb der Grenzen eingesetzt wird. Gegenüber der Sprache C und den daraus potentiell resultierenden IR-Konstrukten schränken wir weiterhin ein:

- Es wird keine Parallelität benutzt.
- Es werden keine Ausnahmen benutzt (in C via setjmp/longjmp).
- Es gibt keine Variablen, die durch nicht aus dem Programm ersichtliche Effekte verändert werden (d.h. Verbot des C-Attributs *volatile*).

Anforderungen an die Ausführung. Einige Aspekte sind nicht in der Programmiersprache verwurzelt, sondern von der Ausführung abhängig. Hier setzen wir voraus:

- Die Ausführung ist endlich.
- Die Analyse arbeitet auf der vollständigen IR (kein unbekannter Code durch Bibliotheken oder fehlende Einheiten). Außerdem entspricht die IR dem ausgeführten Code, d.h. es wird kein Code zur Laufzeit generiert oder modifiziert, und wir können Compiler-Optimierungen bei der Code-Generierung ignorieren.
- Es gibt keine Fehler durch Zeiger, z.B. Dereferenzierung ohne gültiges Ziel und doppelte Deallokation.

10.1.2.2 Repräsentation der Ausführung

Die Idee für den Korrektheitsbeweis besteht nun darin, die Ausführung des Programmes in dem von der Analyse konstruierten IDFG nachzuvollziehen. Der Vorteil dabei ist die Linearität des Ausführungsablaufes in der Zeit, was schließlich einen einfachen Induktionsbeweis erlaubt. Zum Vergleich der Ausführung mit dem IDFG der Analyse müssen wir hierzu zunächst den Speicher auf Objekte und die Einzelschritte der Ausführung auf Definitionen und Verwendungen abbilden.

Sei $\mathcal{M}$ der lineare Speicher, auf dem die Ausführung arbeitet, und $\mathcal{X}$ die Menge der Speicherzugriffe, welche die Ausführung lesend oder schreibend durchführt. Außerdem sei $\mathcal{I}$ die Menge der Ausdrücke in der Zwischendarstellung für das Programm, wie sie auch zur Codegenerierung für P verwendet wurden.

Definition 10.1.1 Zur Beschreibung von Speicherzugriffen existieren die folgenden Funktionen:

$pos : \mathcal{X} \longrightarrow \mathcal{I}$	Position des Zugriffs in der IR
$mem : \mathcal{X} \longrightarrow \mathcal{P}(\mathcal{M})$	Betroffener Speicherbereich
$kind : \mathcal{X} \longrightarrow \{read, write\}$	Art des Zugriffs
$val : \mathcal{X} \longrightarrow \mathcal{M}$	Ziel eines Zeigers, sonst undefiniert

Außerdem benennt für einen Zugriff die Funktion
$defs : \mathcal{X} \longrightarrow \mathcal{P}(\mathcal{X}) : x \mapsto \{y \mid kind(y) = write \wedge mem(x) \cap mem(y) \neq \emptyset\}$
die schreibenden Zugriffe auf zumindest Teile des gleichen Speicherbereichs.

Die Speicherzugriffe von $\mathcal{E}$ erfolgen sequentiell. Die sich daraus ergebende lineare Folge von Zugriffen sei $v_1, \ldots, v_n$. Für einen Lesezugriff $v_i \in \mathcal{X}$ sei dann

$$\begin{aligned}\mathcal{D}_i \quad = \quad & \{v_j \in defs(v_i) \mid j < i \\ & \wedge \nexists k : j < k < i \wedge v_k \in defs(v_i) \wedge mem(v_j) \subseteq mem(v_k)\}.\end{aligned}$$

Diese Menge $\mathcal{D}_i$ enthält die für v_i relevanten schreibenden Speicherzugriffe. Sie definieren zusammen den Wert, den v_i liest.

Definition 10.1.2 Der *Ausführungsgraph* $G_{exe}(\mathcal{E})$ besteht aus den Speicherzugriffen $\mathcal{X}$ als Knoten und der Kantenmenge

$$E = \{v_j \rightarrow v_i \mid v_i, v_j \in \mathcal{X} \wedge kind(v_i) = read \wedge v_j \in D_i\}.$$

Dieser Graph repräsentiert den tatsächlichen Datenfluss der Ausführung. Um das Soll-Resultat für $\mathcal{E}$ zu erhalten, müssen wir noch den tatsächlichen Speicher auf die in der Analyse benutzten Objekte abbilden. Sei dazu $\mathcal{O}$ die Menge der Stack- und Heapobjekte, welche die Analyse für das Programm

unterscheidet. Diese hängt von der konkreten Implementierung der Analyse ab, z.B. davon, ob die Analyse feld-sensitiv ist oder nicht. Dazu gehört die Funktion $Object : \mathcal{M} \longrightarrow \mathcal{O}$, welche jedem in $\mathcal{E}$ referenzierten Speicherplatz ein Analyse-Objekt zuordnet. Sie bildet in der Regel verschiedene Speicherplätze auf das gleiche Objekt ab. Daraus und unter Beachtung der kontext-insensitiven Zusammenlegung bei gleicher IR-Position ergibt sich auch ein Zusammenziehen des Ausführungsgraphen:

Definition 10.1.3 Zwei Knoten $v_i, v_j \in \mathcal{X}$ sind äquivalent (geschrieben $v_i \sim v_j$), falls gilt: $Object(v_i) = Object(v_j) \wedge pos(v_i) = pos(v_j)$. Die Äquivalenzklasse für v_i bezeichnen wir mit $[v_i]$.

Die gleiche IR-Position impliziert dabei die gleiche Art von Zugriff. Im Falle Zeiger-indirekter Operationen impliziert sie aber noch nicht das gleiche Objekt, so dass diese Bedingung separat hinzugefügt wurde.

Definition 10.1.4 Der *Objekt-Ausführungsgraph* G'_{exe} besteht aus den Knoten $\{[v_i] \mid v_i \in \mathcal{X}\}$ und der Kantenmenge

$$\{[v_j] \to [v_i] \mid v_j \to v_i \in G_{exe} \wedge [v_i] \neq [v_j]\}$$

Der Objekt-Ausführungsgraph ist das Soll-Resultat einer kontext-insensitiven Zeiger- und Datenfluss-Analyse. Das von unserer Analyse berechnete Ist-Resultat liegt zunächst in ISSA-Form vor. Es gibt jedoch eine bijektive Abbildung zwischen dieser Darstellung und der eher naiven Darstellung, in der nur noch Definitionen und Verwendungen als Knoten vorhanden und mit Kanten verbunden sind. Schwache Aktualisierungen sollen hierbei auch nicht mehr über eine Kante der vorausgehenden Definitionen zur schwachen Aktualisierung dargestellt werden; stattdessen sollen die vorausgehenden Definitionen direkt eine Kante zu den folgenden Verwendungen erhalten. Unter Verwendung dieser Abbildung können wir daher für die Korrektheit auch von dieser anderen, mit G'_{exe} vergleichbaren Darstellung ausgehen. Analog können wir den Graphaufbau statt als ISSA-Konstruktion auch als Bestimmung der gültigen Definitionen auffassen.

10.1.2.3 Induktiver Korrektheitsbeweis

Bezeichnen wir den von unserer Analyse berechneten IDFG in dieser vergleichbaren Darstellung mit G_{ana}, so wollen wir für die Korrektheit zeigen:

$$G'_{exe} \subseteq G_{ana}.$$

Wir beweisen dazu induktiv über die v_i die folgenden beiden Eigenschaften für alle $v_i \in \mathcal{X}$:

$$[v_i] \in G_{ana} \tag{10.1}$$

$$val(v_i) \in PointsTo([v_i]) \text{ , falls } v_i \text{ ein Zeigerzugriff ist} \tag{10.2}$$

Unter Annahme der Korrektheit der Bestimmung gültiger Definitionen folgt daraus nämlich, dass G_{ana} die Kanten von G'_{exe} enthält:

Lemma 10.1.5 *Gelten (10.1) und (10.2) für alle $v_i \in \mathcal{X}$, so enthält G_{ana} bei korrekter Bestimmung gültiger Definitionen auch alle Kanten aus G'_{exe}.*

Beweis: Dank (10.1) enthält G_{ana} alle Knoten von G'_{exe}. Wegen (10.2) sind die berechneten Zeigerziel-Mengen höchstens eine Überschätzung der realen Mengen. Liegt tatsächlich eine Überschätzung vor, so erzeugt die Analyse für indirekte Zugriffe mehr Knoten als in G'_{exe} vorhanden. Eine andere Quelle für zusätzliche Knoten gibt es im Verfahren nicht. Diese zusätzlichen Knoten fügen im Vergleich zu einem Szenario ohne zusätzliche Knoten lediglich Kanten hinzu, es geht keine Kante verloren. Denn mit unseren Annahmen hat in G'_{exe} jede indirekte Definition mindestens ein Ziel, und somit bei Überschätzung in G_{ana} höchstens mehr Ziele. In diesem Falle legen wir jedoch für alle Ziele der indirekten Definition eine schwache Aktualisierung an, so dass keine Kante verloren geht. Für die Betrachtung der Kanten können wir daher annehmen, dass G_{ana} und G'_{exe} exakt die gleichen Knoten besitzen.

Es bleibt zu zeigen, dass G_{ana} alle Kanten von G'_{exe} enthält. Aufgrund der identischen Knotenmenge und der Korrektheit der Bestimmung gültiger Definitionen (bei indirekten starken Aktualisierungen wieder mit unserer Annahme, dass keine Dereferenzierungen ohne Ziel sind und somit keine Blockaden übrig bleiben) ist dies der Fall, wenn noch einmal der Graphaufbau-

Schritt auf der finalen Knotenmenge ausgeführt wird. Die Fixpunkt-Iteration gewährleistet diese Ausführung jedoch. □

Insbesondere für die Variante mit Erkennung indirekter starker Aktualisierungen haben wir dieses Lemma hier explizit formuliert. Wir sehen darin, dass die Annahme einfließt, dass keine indirekte Definition im Fixpunkt blockiert, wenn gemäß (10.2) eine Überschätzung der Zeigerziele erreicht wurde. Enthält P tatsächlich einen solchen Programmierfehler für eine indirekte Definition, so können zwei Aspekte zu einem nicht konservativen Verhalten führen:

1. Für die fehlerhafte Stelle wird möglicherweise genau ein Ziel erkannt und daher eine indirekte starke Aktualisierung angelegt. Nehmen wir folgendes Beispiel:

   ```
   x = 1;
   *p = 5;
   y = x;
   ```

   Nehmen wir an, die Ausführung läuft hier in eine Dereferenzierung ohne Ziel, aber die Analyse hat durch Überschätzung x als (einziges) Ziel für p erkannt. Dann wird die Analyse als Definition für die Verwendung von x eben jene indirekte Definition (als starke Aktualisierung) bestimmen. Für die Ausführung dagegen hängt das Verhalten ab der fehlerhaften Dereferenzierung – und damit auch die Definition für die Verwendung von x – von hier nicht bekannten Umständen ab (z.B., ob die Sprache eine Ausnahme erzeugt, das Betriebssystem das Programm abbricht etc.). In diesem Sinne können wir keine Aussage zu G'_{exe} treffen, und damit auch nicht zu der Frage, ob G_{ana} eine konservative Abschätzung dazu ist.

2. Die fehlerhafte Stelle könnte auch in der Analyse ohne Ziel bleiben und damit weiterhin blockieren. Wie in Abschnitt 8.5 besprochen, kann dies als Folge die Zeigerziele an anderer Stelle beeinträchtigen. Dort haben wir jedoch auch unsere Relaxations-Strategie beschrieben, die damit umgehen kann.

Der Induktionsbeweis läuft nun über die v_i, d.h. über die Knoten des Ausführungsgraphen G_{exe}, in der Reihenfolge der Ausführung. Der Rückgriff auf die Ausführung liefert uns dabei eine klare lineare Abfolge als Basis für den Beweis. Wir besprechen zunächst den Beweis für die Variante ohne indirekte starken Aktualisierungen und im Anschluss die notwendigen Anpassungen für diese Erweiterung.

Induktionsanfang: Wir beginnen diesen Beweis der beiden Eigenschaften mit v_1. Da dies die (zeitlich) erste Aktion der Ausführung ist, handelt es sich um eine direkte Definition oder Verwendung. Derartige direkte Knoten sind bereits im initialen IDFG enthalten, so dass der Induktionsanfang für (10.1) gezeigt ist. Handelt es sich um einen Schreibzugriff auf einen Zeiger, so muss dieser ein Zeigerliteral zuweisen, da keine Leseoperation voranging. Zeigerliterale sind uns jedoch direkt aus der IR bekannt. Handelt es sich um einen Lesezugriff auf einen Zeiger, so hat dieser Zeiger kein Ziel, weswegen wir auch den Induktionsanfang für (10.2) als gesichert ansehen können.

Induktionsschritt: Für den Induktionsschritt betrachten wir ein $v_i \in \mathcal{X}$ und können davon ausgehen, dass (10.1) und (10.2) für alle $v_j, j < i$ bereits gilt. Für $[v_i] \in G_{ana}$ müssen wir nur Zugriffe v_i betrachten, die als Resultat einer Zeigeroperation zustandekommen (denn die übrigen sind bereits im initialen IDFG enthalten). Sei v_j dazu die Verwendung des Zeigers in der Dereferenzierung, für die wir gerade v_i als Ziel betrachten. Dann wurde v_j (als lesender Zugriff) vor v_i ausgeführt, also $j < i$. Demnach gelten die beiden Eigenschaften (10.1) und (10.2) für v_j. Damit aber ist ausgesagt, dass die Analyse das an v_j benutzte Zeigerziel kennt. Folglich legt sie einen Knoten für das Zeigerziel der Dereferenzierung an, also $[v_i] \in G_{ana}$.

Für den Beweis der zweiten Eigenschaft unterscheiden wir danach, ob v_i ein Lese- oder Schreibzugriff auf einen Zeiger ist. Ist es ein Schreibzugriff, so ist das Zeigerziel entweder ein direkt zugewiesenes Zeigerliteral (bekannt aus der IR) oder der Wert einer vorausgegangenen Zeiger-Verwendung v_j. Dann aber ist $j < i$ und damit das Ziel von v_j bekannt. Falls schließlich v_i eine Zeiger-Verwendung ist, so haben wir induktiv alle $v_j \in \mathcal{D}_i$ bereits mit korrektem Zeigerziel in G_{ana} und über Datenfluss-Kanten mit v_i verbunden. Auch in diesem Fall propagiert die Analyse daher das richtige Ziel zu v_i.

Dieser Induktionsbeweis sichert uns die beiden Eigenschaften (10.1) und (10.2) zu. Zusammen mit dem Lemma 10.1.5 haben wir damit gezeigt:

Satz 10.1.6 *Die kombinierte Analyse terminiert mit korrekten Resultaten.*

Damit haben wir die These 4 aus der Einleitung nachgewiesen:

Zeigeranalyse und Datenfluss-Analyse können statt nacheinander auch in Kombination erfolgen.

Für die erweiterte Analyse, welche mit Hilfe des Blockade-Graphen auch indirekte starke Aktualisierungen beherrscht, gilt der Beweis mit leichten Veränderungen:

Der Induktionsanfang bleibt unverändert. Für den Induktionsschritt beobachten wir, dass die geforderte Dominanzkonstellation eine zeitliche Reihenfolge auferlegt: Gilt $blocks(s, d \to u)$ oder $\overline{blocks}(s, d \to u)$, so wird d sicher vor s ausgeführt und s wiederum sicher vor u. Die Induktion betrachtet die Knoten daher ebenfalls in dieser Reihenfolge: d, s, u.

Das bedeutet insbesondere, dass vor der Betrachtung von u die potentielle starke Aktualisierung s betrachtet wird. Mit der Induktionsannahme kommt das in der Ausführung benutzte Zeigerziel korrekt an s an. Dann ist s entweder eine 1-Definition oder hat bereits mehr als ein Ziel. Betrachten müssen wir nur den Fall, dass s genau ein Zeigerziel hat. Ist dieses Ziel das Objekt von u, so bleibt die Blockade von Zeigerzielen entlang der Kante $d \to u$ bestehen. In allen anderen Fällen heben wir die Blockade auf, womit der obige Beweis der Kernanalyse greift.

Eine bleibende Blockade ist jedoch gerechtfertigt: Dann war s schließlich eine Definition des Objekts von u, womit dort nicht der Wert der vorausgegangenen Definition d verwendet wird. An u kommen somit die korrekten Zeigerziele an, was für den Induktionsschritt ausreicht.

Damit haben wir auch gezeigt:

Satz 10.1.7 *Die erweiterte Analyse terminiert mit korrekten Resultaten.*

10.1.2.4 Lockerung der Voraussetzungen

Die für den Korrektheitsbeweis getroffenen Voraussetzungen lassen sich lockern. So kann man die Anforderung fallen lassen, dass die Ausführung endlich sein muss. Dann betrachtet man die Knoten v_i bis zu einem Zeitpunkt t. Der Beweis zeigt, dass die Ausführung bis zu diesem Zeitpunkt korrekt modelliert ist. Dies gilt jedoch für jeden beliebigen endlichen Zeitpunkt t, und mehr interessiert nicht.

Die Dereferenzierung eines Zeigers ohne gültiges Ziel führt zu einem undefinierten Verhalten der Ausführung. Dieses Verhalten hängt von vielen Faktoren ab, nicht zuletzt auch vom Betriebssystem. Es ist nicht klar, was *Korrektheit* der Analyse in solchen Fällen bedeuten soll. Wir erlauben der Analyse, solche Fälle zu ignorieren. Es ist dabei sicherlich wünschenswert, einen Hinweis auf den Programmierfehler zu generieren. Aufgrund der Zusammenfassung aller möglichen Ausführungen in einem IDFG kann die Analyse jedoch an einem solchen Punkt nicht immer direkt erkennen, dass ein Zeigerziel fehlt oder ungültig ist: Es könnte ein gültiges Ziel auf einem anderen Pfad ankommen.

Eine simple Diagnose derartiger Fehler über die Ausgabe der Dereferenzierungen ohne Ziel nach Erreichen des Fixpunktes übersieht daher unter Umständen ein paar reale Fehler. Für eine konservative Diagnose wäre konzeptionell eine Überprüfung aller Pfade zu jeder Dereferenzierung nötig. Effizient kann man dies generell über eine Rückwärts-Tiefensuche ab jeder Dereferenzierung erreichen: Endet eine solche Suche an mindestens einem Knoten, der weder einen Vorgänger besitzt noch ein Zeigerliteral trägt, so wurde ein Pfad ohne Ziel gefunden. Verwendet die Implementierung für die Propagierung bereits das Vorgehen, Dereferenzierungen zurück zu propagieren, so gelingt ihr diese Diagnose bereits. Abschnitt 8.5 hat bereits eine ähnliche Strategie vorgestellt, um die Folgen einer Blockade im Fixpunkt zu reduzieren.

Die Anforderung, den vollständigen Code des zu analysierenden Programmes samt darin verwendeter Bibliotheken zu kennen, ist typisch für interprozedurale Analysen, jedoch in der Praxis nicht immer erfüllbar. Ein Ausweg besteht darin, die wichtigen Effekte einer Bibliotheksfunktion als Zusammen-

fassung der Analyse bekannt zu machen. Die Zusammenfassung kann manuell oder durch eine Voranalyse der Bibliothek geschehen. Generell ist die Bibliotheksproblematik ein schwieriges Thema für sich und soll darum nicht näher ausgeführt werden. Unsere Analysenbeschreibung hat hier nur das Objekt Unknown_Objects eingesetzt.

10.2 Größe der Resultate

Zur weiteren Analyse des Verfahrens bestimmen wir nun Schranken für die Größe der Resultate. Es ist also unser Ziel in diesem Abschnitt, folgende Größen zu begrenzen:

- ISSA-Graphgröße: Anzahl der Knoten und Kanten
- Aufrufgraph: Anzahl der Knoten und Kanten im Aufrufgraphen
- Seiteneffekte: Anzahl der Seiteneffekt-Knoten und -Kanten
- Zeigermengen: Größe der Zeigerzielmengen

Wann immer möglich, sollte die Abschätzung anhand der verschiedenen Eingabe-Größen erfolgen. Hierzu zählen insbesondere:

- n_{ir} = Anzahl Knoten in der Zwischendarstellung des zu analysierenden Programmes P.
- f = Anzahl Unterprogramme in P.
- c = Anzahl der Aufrufstellen in P.
- g = Anzahl der nicht-lokalen Objekte in P.

Annahmen. Wir gehen von folgenden Annahmen aus, die sich auch in unserer Evaluation alle als zutreffend erwiesen haben und außerdem so auch in anderen Publikationen auftauchen: Es ist vernünftig anzunehmen, dass mit wachsender Programmgröße die Zahl der Unterprogramme steigt, nicht

jedoch die Größe der einzelnen Unterprogramme oder die Anzahl der Parameter pro Unterprogramm. Diese Größen können wir durch eine Konstante abschätzen. Als Konsequenz daraus ist auch die Zahl an Grundblöcken pro Unterprogramm (bzw. die Größe eines CFG) sowie die Höhe der zugehörigen Dominanzbäume durch eine Konstante beschränkt. Ebenso können wir nun die Größe einer (iterierten) Dominanzgrenze als konstant annehmen, was wiederum die Zahl der ϕ-Knoten pro Objekt und Unterprogramm konstant begrenzt.

Seiteneffekte. Die Anzahl an Seiteneffekt-Knoten lässt sich relativ leicht abschätzen: Pro nicht-lokalem Objekt erstellen wir höchstens zwei formale Seiteneffekt-Knoten pro Unterprogramm. Analog existieren am Schluss höchstens zwei aktuelle Seiteneffekt-Knoten pro nicht-lokalem Objekt und Aufrufstelle. Das ergibt eine obere Schranke von $2g(f + c)$ für die Anzahl an Seiteneffekt-Knoten. (Dieses Resultat gilt auch für andere Verfahren zur Berechnung der Seiteneffekte.)

Aufrufgraph. Der Aufrufgraph hat (ohne Entfernung toten Codes) exakt f Knoten, pro Unterprogramm einen. Jeder direkte Aufruf ergibt genau eine Kante. Für indirekte Aufrufe hängt die Zahl der Kanten von der Zeigeranalyse ab; wir bezeichnen die Summe aller Aufrufziele für indirekte Aufrufe hier mit t_{calls}. Dann hat der Aufrufgraph $c + t_{calls}$ Kanten. In der Praxis ist t_{calls} in den meisten Fällen sehr klein; der schlechteste Fall, dass jeder indirekte Aufruf an alle Unterprogramme gehen könnte, tritt nicht auf. Eine bessere Schranke ergibt sich schon durch die Anzahl f_{at} der Unterprogramme, deren Adresse genommen wird, als Zielmenge für jeden solchen Aufruf. Das liefert uns die Schranke $t_{calls} \leq c_{indirect} * f_{at}$. Dies ist jedoch immer noch schlimmstenfalls quadratisch in der Programmgröße, da beide Faktoren (in der Regel sogar deutlich) kleiner als n_{ir} sind. Wir bevorzugen daher die Abschätzung $t_{calls} \ll n_{ir}$, welche wir leider nicht theoretisch nachweisen können, die aber in ausnahmslos allen untersuchten Beispielen klar gilt. Bezeichnen wir allgemein die Größe des Aufrufgraphen mit $|CG| = f + c + t_{calls}$, so können wir diese unter Ausnutzung der Abschätzung begrenzen durch $|CG| \in \mathcal{O}(n_{ir})$,

also linear in der Programmgröße. (Ohne die Präzision der Zeigeranalyse zu betrachten, also wie hier erfolgt, gilt diese Abschätzung für jedes Verfahren zur Berechnung des Aufrufgraphen.)

ISSA-Graph. Mit den vorgenannten Resultaten können wir nun die Größe des Datenfluss-Graphen abschätzen:

Lemma 10.2.1 *Der ISSA-Graph enthält höchstens $n \in \mathcal{O}(g * f)$ Knoten.*

Beweis: Wir bestimmen die Zahl der Knoten pro Unterprogramm: An einer Aufrufstelle existieren maximal zwei Knoten pro Objekt, an den übrigen Ausdrücken höchstens ein Knoten pro Objekt. Für jeden Grundblock existiert höchstens ein ϕ-Knoten pro Objekt. Nun haben wir jedoch nur konstant viele Grundblöcke, Aufrufstellen und Ausdrücke pro Unterprogramm, so dass wir pro Objekt und Unterprogramm nur eine konstante Zahl an Knoten erhalten. In einem Unterprogramm sind dabei nur $g + const$ viele Objekte im Einsatz, womit sich $n \in \mathcal{O}(g * f)$ ergibt. □

Lemma 10.2.2 *Der ISSA-Graph hat höchstens $m \in \mathcal{O}(g * |CG|)$ Kanten.*

Beweis: Pro Kante im Aufrufgraphen existieren maximal zwei Datenfluss-Kanten für jedes nicht-lokale Objekt, sowie weitere, konstant viele Datenfluss-Kanten für die Parameter. Die Zahl der interprozeduralen Datenfluss-Kanten können wir daher mit $\mathcal{O}(g * (c + t_{calls}))$ abschätzen.

Intraprozedural zählen wir die eingehenden Kanten der Knoten. Dabei haben alle Knoten außer den ϕ-Knoten höchstens eine eingehende Kante gemäß der *single-assignment*-Eigenschaft. ϕ-Knoten haben eine eingehende Kante pro vorausgehendem Grundblock, was jedoch eine Konstante ist. Daher ist die Zahl der intraprozeduralen Kanten linear in der Zahl der Knoten, also mit dem vorigen Lemma in $\mathcal{O}(g * f)$. An dieser Abschätzung ändern auch die Zuweisungsbrücken nichts: Jeder ISSA-Knoten hat entweder nur maximal eine eingehende oder nur maximal eine ausgehende solche Kante. Ihre Zahl ist daher ebenso auf $\mathcal{O}(g * f)$ begrenzt. □

Die Größe des Graphen ist damit schlimmstenfalls quadratisch in der Programmgröße.

Größe der Zeigerziel-Mengen. Die erschöpfende Propagierung benötigt keinen weiteren Platz dauerhaft (es fällt lediglich der Platzbedarf für eine Tiefensuche auf dem Stack an). Das inkrementelle Vorgehen speichert jedoch Zeigerziele zwischendurch. Bezeichnen wir die Zahl allgemeiner Definitionen mit d und die Zahl potentieller Zeigerziele mit t_{max}, so benötigt die inkrementelle Variante hierfür $\mathcal{O}(d * t_{max})$ Speicher.

Größe mit indirekten starken Aktualisierungen

Durch die höhere Präzision schrumpft der IDFG und die Zeigerziel-Mengen. Dafür benötigen wir jedoch den Platz für den Blockade-Graphen und die an diesem annotierten blockierten Kanten.

Der Blockade-Graph erreicht gleich zu Beginn seine maximale Größe, da die Analyse lediglich Teilgraphen daraus löscht. Somit können wir die Zahl der Knoten in B direkt abschätzen: Maximal jede indirekte Definition, jede direkte Aufrufstelle und jeder CFG-Vereinigungspunkt treten als Knoten auf; insgesamt also höchstens $\mathcal{O}(n_{ir})$ Knoten. Als Kanten haben wir Kanten von lokalen Knoten zu Aufrufstellen der jeweiligen Funktion sowie lokale Kanten zu den Kandidaten für ϕ-Knoten. Mit unserer Annahme einer konstanten Größe der einzelnen CFG gibt es pro Funktion nur konstant viele Knoten, die eine Kante zu Aufrufstellen besitzen; wir zählen also $\mathcal{O}(c)$ solcher Kanten. Ebenso kann jeder Kandidat für einen ϕ-Knoten nur konstant viele eingehende Kanten erhalten, da es lokal nicht mehr vorausgehende Knoten gibt. Die Zahl der Kanten ist damit ebenfalls linear in der Eingabegröße, und somit auch die Größe von B insgesamt.

Zur Abschätzung des Speicherbedarfs für die blockierten Knoten beobachten wir: Jeder Knoten des finalen ISSA-Graphen ist zu jedem Zeitpunkt an höchstens einem Knoten aus B als blockiert annotiert. Das begrenzt den dafür nötigen Speicherbedarf auf $\mathcal{O}(n)$.

10.3 Laufzeit

Wir nutzen die Resultate über die Größe der berechneten Informationen zur Abschätzung der Laufzeit unserer kombinierten Analyse. Neben den schon eingeführten Größen nutzen wir dazu noch i für die Anzahl der Iterationen (welche sich nur schwer eingrenzen lässt, in der Praxis aber niedrig ausfällt).

Wir gehen davon aus, dass das Einfügen eines Zeigerziels in konstanter Zeit erfolgt; ebenso der Test, ob ein Ziel bereits in der Menge vorhanden ist. Bitvektoren als Datenstruktur für die Zeigerziele eines Knotens sind eine Möglichkeit, diese Anforderung zu erfüllen. Wiederum zeigen wir das Resultat zunächst für die Variante ohne indirekte starke Aktualisierungen, und nennen im Anschluss die notwendigen Anpassungen für diese Erweiterung.

Satz 10.3.1 *Die asymptotische Laufzeit der kombinierten Analyse liegt in* $\mathcal{O}((t_{max} * g + t_{calls}) * |CG|)$

Beweis: Anstatt die Zahl der Iterationen zu zählen und eine Laufzeitabschätzung pro Iteration durchzuführen, berechnen wir die Summe der Laufzeiten aller Graphaufbau-Schritte (σ_1) und aller Propagierungs-Schritte (σ_2). Die Initialisierung ist dabei linear in der Programmgröße und damit vernachlässigbar; die Gesamtlaufzeit ergibt sich aus $\sigma_1 + \sigma_2$.

Graphaufbau. Der Graphaufbau-Schritt betrachtet jeden Knoten des finalen ISSA-Graphen genau einmal als neuen Knoten. Die Suche nach der gültigen Definition, wie sie in leichten Variationen für fast alle Knoten zum Einsatz kommt (vgl. Abbildung 5.8), hat konstante Kosten aufgrund der konstanten Höhe der Dominanzbäume. Pro Knoten finden nur konstant viele dieser Suchen statt, so dass sie zusammen die Kosten $\mathcal{O}(n)$ verursachen.

Die Erzeugung neuer Unterein- und -ausgänge mitsamt ihren Verbindungen zu Aus- bzw. Eingängen kostet uns $\mathcal{O}(g * |CG|)$. Die Erzeugung neuer ϕ-Knoten besucht pro neuer allgemeiner Definition jeden Eintrag der Dominanzgrenze; hiervon gibt es jedoch nur konstant viele, so dass sich die Kosten auf $\mathcal{O}(n)$ belaufen.

Jede allgemeine Definition kann nur mit konstant vielen allgemeinen Verwendungen des gleichen Objekts verbunden sein, da es pro Unterprogramm

und Objekt nur konstant viele Knoten gibt. Da wir einen Dominanztest in konstanter Zeit ausführen können, verursacht das Angeln von dominierten Verwendungen über alle Knoten hinweg maximal die Kosten $\mathcal{O}(n)$.

Mit Lemma 10.2.1 für die Anzahl an Knoten ergeben sich daraus für den Graphaufbau die Gesamtkosten

$$\sigma_1 \in \mathcal{O}(g * |CG|).$$

Propagierung. Betrachten wir die Kosten für zwei der verschiedenen angesprochenen Implementierungen:

- Für die erschöpfende Propagierung: Pro Knoten und Kante werden pro Iteration maximal t_{max} Zeigerziele propagiert bzw. vereint. Das ergibt die Gesamtkosten $\sigma_2 = \mathcal{O}(i * t_{max} * g * |CG|)$.

- Für die inkrementelle Variante: Jede Kante kann höchstens einmal als neu und damit als Start einer Propagierung für ein Objekt betrachtet werden. Außerdem wird jede Kante pro Objekt auch höchstens einmal besucht, da wir die Zeigerziele genügend oft speichern. Dies ergibt die Gesamtkosten $\sigma_2 = \mathcal{O}(t_{max} * g * |CG|)$.

Prinzipiell sorgt das Angeln von Verwendungen dafür, dass eine solche Verwendung möglicherweise mehrfach besucht wird, nämlich pro Definition (im Laufe der Iterationen) einmal. Da dies aber nur konstant oft pro Verwendung eintreten kann, erhöht sich die Komplexität nicht gegenüber einem Modell, das jeden Knoten nur einmal besucht. Somit erhalten wir für den inkrementellen Ansatz

$$\sigma_2 = \mathcal{O}(t_{max} * g * |CG|).$$

Rekursive Funktionen. Maximal pro indirektem Aufrufziel einmal müssen wir die Erkennung von Zyklen im Aufrufgraphen anwenden. Jede Anwendung läuft in Linearzeit über den Aufrufgraphen, daher die Kosten $\mathcal{O}(t_{calls} * |CG|)$. Dabei werden insgesamt maximal alle ISSA-Knoten noch einmal betrachtet.

Finalisierung. Pruning sowie die Berechnung von starken Zusammenhangskomponenten und der topologischen Ordnung laufen in einer Zeit proportional zur Größe des ISSA-Graphen, also in $\mathcal{O}(g * |CG|)$. □

Auf die Zyklenkontraktion sind wir hier nicht eingegangen: Sie hat zwar einen erheblichen Nutzen in der Praxis, aber uns ist nicht bekannt, wie wir damit auch eine konservative Verbesserung der asymptotischen Schranke erreichen könnten.

KOROLLAR 10.3.2 *Mit $t_{calls} \ll n_{ir}$ liegt die Laufzeit der Analyse in $\mathcal{O}(n_{ir}^3)$.*

Beweis: Wir haben $t_{max}, g, f, c, t_{calls} \ll n_{ir}$ und damit insgesamt die Abschätzung $\sigma_1 + \sigma_2 \ll {n_{ir}}^3$. □

Dies beweist die These 1 dieser Dissertation:

Es existiert eine fluss-sensitive Ausprägung von Andersens Zeigeranalyse mit ebenfalls kubischer asymptotischer Komplexität.

Verglichen mit anderen fluss-sensitiven Verfahren ist diese Komplexität gering. Der Vergleich zu Andersen ist natürlich mit groben Abschätzungen verbunden, so dass die Laufzeit für Andersens Verfahren in der Praxis aufgrund besserer Konstanten (vor allem: deutlich kleinerer Propagierungsgraph) immer noch um Faktoren schneller sein dürfte.

Zusammen mit dem Korrektheitsbeweis untermauert das Resultat auch die These 5:

Eine fluss-sensitive Zeigeranalyse kann anstatt als Datenfluss-Problem auf der Kontrollfluss-Darstellung auch effizient als Grapherreichbarkeits-Problem auf der tatsächlichen, nicht vorab überschätzten Datenfluss-Darstellung realisiert werden.

Man beachte, dass uns der Einsatz der ISSA-Darstellung dabei geholfen hat, mit einer relativ kleinen Anzahl an Kanten auszukommen. Das hat einen positiven Einfluss auf die Größe der Resultate und das im Propagierungs-Schritt zu lösende Grapherreichbarkeits-Problem. Die Propagierung dominiert dabei die Laufzeitkosten, die Kosten des Graphaufbaus sind lediglich linear in der Ergebnisgröße.

Laufzeit mit indirekten starken Aktualisierungen

Die Konstruktion des initialen Blockade-Graphen betrachtet jeden Knoten des zu konstruierenden Blockade-Graphen genau einmal als neuen Knoten. Die Betrachtung der Dominanzgrenze für einen neuen Knoten traversiert nur konstant viele Knoten. Für jeden dieser Knoten betrachten wir alle CFG-Vorgänger – wiederum eine konstante Zahl – und führen Dominanztests durch, was ebenfalls konstante Kosten verursacht. Die Betrachtung der durch einen neuen Knoten induzierten (Kandidaten für) ϕ-Knoten läuft damit in konstanter Zeit ab. Die Betrachtung der Aufrufstellen zur aktuellen Funktion fügt jedesmal zumindest eine entsprechende Kante ein. Die Zeit hierfür ist damit insgesamt durch die im vorigen Abschnitt ermittelte Zahl c dieser Kanten begrenzt. Der Aufbau des Blockade-Graphen hat damit die Kosten $\mathcal{O}(n_{ir})$. Die Kosten der Initialisierung steigen damit asymptotisch nicht im Vergleich zum Verfahren ohne diese Erweiterung.

Ein Test, ob ein Dominator bei der modifizierten Suche nach der gültigen Definition in B liegt oder nicht, sowie die Abspeicherung des Knotens an einem solchen Dominator, erfolgen jeweils in konstanter Zeit. Sie erhöhen damit die Kosten einer einzelnen Suche nicht. Die Suche kann nun aber mehrfach für jeden Knoten angestoßen werden – jedoch höchstens einmal pro Dominator. Da es nur konstant viele Dominatoren pro Knoten gibt, haben wir also auch weiterhin nur konstant viele Suchen nach der gültigen Definition pro Knoten. Auch hier haben wir daher keine erhöhten asymptotischen Kosten im Vergleich zur Kernanalyse ohne starke Aktualisierungen.

Das Zurückhalten und eventuelle Nachholen der Kante von der vorausgehenden Definition zur ersten Konkretisierung einer indirekten Definition erhöht die Kosten mit dem gleichen Argument ebenfalls nicht.

Es bleiben nur noch die Kosten zur inkrementellen Verwaltung des Blockade-Graphen. Insgesamt entfernen wir maximal wieder alle Knoten und Kanten aus B, was wiederum Linearzeit benötigt. Beim Entfernen eines Knotens behandeln wir alle daran vermerkten blockierten Knoten, aber die Kosten hierfür haben wir bereits gezählt.

Summa summarum zeigt uns diese Überlegung, dass der Einsatz der Blo-

ckaden die Komplexität der Analyse nicht erhöht. Dies gilt auch für den Fall der kontext-abhängigen Propagierung, die orthogonal zu den indirekten starken Aktualisierungen ist. Das beweist die These 3:

Sowohl die fluss-sensitive Andersen-Analyse aus These 1 als auch die Variante mit MOVP-Präzision aus These 2 können zusätzlich indirekte starke Aktualisierungen unterstützen, ohne dass sich die Komplexität der Verfahren erhöht.

10.4 Kosten mit IFDS für die Propagierung

10.4.1 Größe der Resultate

Für die Analyse verwenden wir die Symbole aus den vorigen Abschnitten erneut. Dort hat g die Zahl der nicht-lokalen Objekte angegeben. Da die Zahl der Parameter pro Unterprogramm begrenzt ist, können wir nun die Zahl der Eingänge pro Unterprogramm mit $\mathcal{O}(g)$ abschätzen. Die gleiche Abschätzung hatten wir auch bereits für die Zahl der Unterein- bzw. -ausgänge pro Aufrufstelle.

Betrachten wir die präzisere Propagierung ohne die in Kapitel 9 am Ende erwähnten Erweiterungen (feinere Heapmodellierung, kontext-abhängiger Datenfluss). Außerdem konzentrieren wir uns zunächst auf den alternativen Drei-Stufen-Ansatz. Am Ende werden wir auch den Framework-Ansatz für die Propagierung untersuchen (und feststellen, dass der Drei-Stufen-Ansatz asymptotisch effizienter ist). Die Größe des IDFG ist dann dank der höheren Präzision höchstens kleiner geworden. Für die Propagierung benötigen wir nun jedoch zusätzlich die Same-Level-Pfade sowie die Summary-Kanten.

Im schlechtesten Fall haben wir an jeder Aufrufstelle eine Summary-Kante zwischen jedem Unterein- und -ausgang. Bei c Aufrufstellen führt uns dies auf den Platzbedarf von $\mathcal{O}(c * g^2)$.

Die Same-Level-Pfade haben pro Knoten ein Array verlangt, in dem die eingehenden Pfade markiert wurden. Dabei kann von jedem lokalen Eingang ein kontext-abhängiger Pfad zum Knoten V führen, d.h. wir haben bis zu

$\mathcal{O}(g)$ kontext-abhängige Pfade pro Knoten. Die kontext-unabhängigen Pfade starten an lokalen Quellen und Untereingängen. Höchstens jede lokale Instruktion kann eine Quelle sein, und mit unserer Annahme von konstant vielen Instruktionen pro Unterprogramm ergibt dies nur konstant viele Same-Level-Pfade von Quellen zu V. Aus dem gleichen Grund gibt es lokal nur konstant viele Aufrufstellen, wobei jede $\mathcal{O}(g)$ Untereingänge haben kann. Zusammen gibt es daher auch nur $\mathcal{O}(g)$ kontext-unabhängige Same-Level-Pfade pro Knoten. Dann haben wir pro Knoten den Platzbedarf $\mathcal{O}(g)$ für alle Same-Level-Pfade.

Betrachten wir noch die Größe des CI-Graphen und des CS-Graphen für die zweite und dritte Stufe der Propagierung. Der CI-Graph enthält lokal jeweils die Same-Level-Pfade zwischen Quellen bzw. Untereingängen und Ausgängen bzw. Senken. Mit den obigen Argumenten haben wir jeweils $\mathcal{O}(g)$ mögliche Start- und Endpunkte pro Funktion, insgesamt also höchstens $\mathcal{O}(f * g^2)$ Kanten im CI-Graphen.

Für den CS-Graphen gilt die gleiche Abschätzung: Hier hatten wir die Same-Level-Pfade zwischen einer Quelle, einem Eingang oder einem Untereingang und einer Senke oder einem Unterausgang betrachtet. Wieder gibt es somit jeweils $\mathcal{O}(g)$ mögliche Start- und Endpunkte pro Funktion.

10.4.2 Laufzeit

Mit der in Abschnitt 9.5 beschriebenen Umsetzung der präziseren Propagierung haben wir die Kosten $\mathcal{O}(1)$ für das Hinzufügen einer Summary-Kante sowie für den Test auf Enthaltensein. Gleiches gilt für die Same-Level-Pfade und bezüglich des Einfügens auch für die Arbeitsliste.

Bestimmen wir zuerst die Laufzeit der ersten Stufe, d.h. der Berechnung der Same-Level-Pfade und der Summary-Kanten.

Lemma 10.4.1 *Eine Ausführung der ersten Stufe des alternativen Verfahrens hat die Kosten $\mathcal{O}(g^3 * |CG|)$.*

Beweis: Die Größe des IDFG ist höchstens kleiner geworden, so dass sich am Graphaufbau nichts ändert. Wir analysieren daher nur die Kosten der Pro-

pagierung mit den gleichen Abschätzungen für m und n (nach Lemma 10.2.1 und 10.2.2: $n + m \in \mathcal{O}(g * |CG|)$.)

Begrenzen wir zunächst die Kosten der Hilfsfunktionen in den Abbildungen 9.13 und 9.14.

- Die Funktion Propagate hat konstante Kosten pro Aufruf.

- Die Funktion ISSA_Summaries_Visit_Call betrachtet pro Aufruf die ausgehenden Kanten eines Unterausgangs V. Pro Aufrufziel gibt es dabei höchstens einen Nachfolger, so dass die Zahl der Kanten durch die Zahl der Aufrufziele beschränkt ist. Weiterhin betrachtet diese Hilfsfunktion die vom Unterausgang ausgehenden Summary-Kanten. Hier gibt es maximal pro Untereingang der Aufrufstelle eine Kante, weswegen wir also die Kosten $\mathcal{O}(g)$ für die Summary-Kanten veranschlagen.

- Die Funktion ISSA_Summaries_Visit_Exit hat im Falle eines kontext-unabhängigen Pfades konstante Kosten pro Aufrufer, da es zu jedem Aufrufer höchstens eine Kante gibt. Für einen kontext-abhängigen Pfad betrachtet die Funktion pro Aufrufer jeweils eine Kante zum Eingang U sowie eine Kante vom Ausgang V zurück zum Aufrufer. Für jeden Aufrufer kann dabei eine Summary-Kante entstehen und die am Unterausgang endenden Same-Level-Pfade verlängern. Da es maximal $\mathcal{O}(g)$ viele solcher Pfade gibt, belaufen sich auch die Laufzeit-Kosten pro Aufrufer auf $\mathcal{O}(g)$.

Durch den Schutz in Propagate wird jeder Same-Level-Pfad höchstens einmal in die Arbeitsliste und die Menge aller Same-Level-Pfade eingetragen. Über alle Knoten hinweg haben wir dabei maximal $\mathcal{O}(n * g)$ Same-Level-Pfade, und damit maximal so viele Durchläufe der Schleife über die Arbeitsliste in der Hauptfunktion der Propagierung.

Wiederum aus dem Grund, dass es nur $\mathcal{O}(g)$ viele Same-Level-Pfade gibt, die an einem Knoten V enden, wird ein bestimmter Endknoten V des in der Schleife gewählten Pfades maximal $\mathcal{O}(g)$ Mal besucht. Das bestimmt die Zahl der Aufrufe an die Hilfsfunktionen: ISSA_Summaries_Visit_Call und

ISSA_Summaries_Visit_Exit werden damit höchstens $\mathcal{O}(g)$ Mal pro Unterausgang bzw. Ausgang aufgerufen, also $\mathcal{O}(g^2 * c)$ bzw. $\mathcal{O}(g^2 * f)$ Mal. Dadurch verursacht ISSA_Summaries_Visit_Call insgesamt die Kosten $\mathcal{O}(g^3 * c)$ für Summary-Kanten, sowie $\mathcal{O}(g^2 * |CG|)$ weitere Aufrufe an die Funktion Propagate. (Hier konnten wir die Kosten pro Aufrufziel über mehrere Knoten hinweg aufsummieren zur Größe des Aufrufgraphen.) Für ISSA_Summaries_Visit_Exit erhalten wir analog (und wieder mit der Summation über mehrere Knoten für die Behandlung der Kosten pro Aufrufer) die Kosten $\mathcal{O}(g^2 * |CG|)$ für alle kontext-unabhängigen Fälle zusammen, sowie die Kosten $\mathcal{O}(g^3 * |CG|)$ für die kontext-abhängigen Fälle.

Damit haben wir für die Hilfsfunktionen insgesamt die Kosten

$$\mathcal{O}(g^3 * c) + \mathcal{O}(g^2 * |CG|) + \mathcal{O}(g^2 * |CG|) + \mathcal{O}(g^3 * |CG|) = \mathcal{O}(g^3 * |CG|).$$

Hinzu kommen die Kosten der Hauptfunktion für $\mathcal{O}(n * g)$ Schleifendurchläufe, in denen als weitere Kosten lediglich die intraprozeduralen Kantenbesuche anfallen. Da jeder Knoten V höchstens $\mathcal{O}(g)$ Mal betrachtet wird, belaufen sich diese Kosten bei m Kanten zusammen auf $\mathcal{O}((n + m) * g)$. Wegen $n + m \in \mathcal{O}(g * |CG|)$ haben wir damit die Kosten der ersten Stufe bestimmt als $\mathcal{O}(g^3 * |CG|)$. □

Damit kennen wir die Kosten über alle Iterationen hinweg für die erschöpfende Variante durch eine Multiplikation mit dem Faktor i für die Zahl der Iterationen. Anders dagegen sieht es bei der inkrementellen Umsetzung der ersten Stufe aus:

Lemma 10.4.2 *Die inkrementelle Umsetzung der ersten Stufe des alternativen Verfahrens hat über alle Iterationen hinweg die Kosten* $\mathcal{O}(g^3 * |CG|)$.

Beweis: Die Kosten $\mathcal{O}(g^3 * |CG|)$ haben wir oben für eine einmalige Ausführung ermittelt, wobei die Abschätzung auch für eine Ausführung auf dem finalen Graphen gültig ist. Jetzt müssen wir uns nur vergewissern, dass das inkrementelle Vorgehen im Vergleich zu einer einmaligen Ausführung auf dem finalen Graphen keine asymptotischen Mehrkosten verursacht.

Die Aufbewahrung der Same-Level-Pfade und Summary-Kanten über die Iterationen hinweg sorgt zusammen mit den Tests, ob ein Pfad bzw. eine

Kante schon darin enthalten ist, dafür, dass jeder solche Pfad bzw. jede solche Kante über alle Iterationen hinweg nur einmal in der jeweiligen Menge und damit auch in der Arbeitsliste auftaucht. Damit gelten die Abschätzungen für die Kosten der Hilfsfunktionen und der Hauptschleife aus dem vorigen Lemma nun über alle Iterationen hinweg betrachtet.

Hinzu kommen jedoch die Initialisierungskosten für die in Abbildung 9.17 gezeigten Aktionen. Erneut mit der Abschätzung von höchstens $\mathcal{O}(g)$ eingehenden Same-Level-Pfaden pro Knoten sowie dem Wissen, dass jede Kante insgesamt nur einmal als neu gemeldet wird, können wir diese Kosten erfassen:

- Für eine intraprozedurale neue Kante haben wir die Kosten $\mathcal{O}(g)$, über alle Iterationen hinweg also die Kosten $\mathcal{O}(m * g)$

- Für eine Kante von einem Unterausgang zu einem Eingang haben wir die Kosten $\mathcal{O}(g^2)$, insgesamt also die Kosten $\mathcal{O}((c + t_{calls}) * g^3)$.

- Für eine Kante von einem Ausgang zu einem Untereingang entstehen ebenfalls maximal die Kosten $\mathcal{O}(g^2)$ und damit für all diese Kanten zusammen die Kosten $\mathcal{O}((c + t_{calls}) * g^3)$.

Zusammen kostet uns die Initialisierung daher maximal $\mathcal{O}((c + t_{calls}) * g^3)$.

□

Nachdem wir damit die Kosten der ersten Stufe kennen, bleiben uns noch die Kosten der weiteren beiden Stufen. Dazu greifen wir auf die Resultate für die Kosten der kontext-insensitiven Propagierung zurück und übertragen diese auf die veränderte Graph-Grundlage. So hat, wie in Satz 10.3.1 gezeigt, die erschöpfende Propagierung die Kosten $\mathcal{O}(t_{max})$ pro Iteration und Knoten bzw. Kante, während die inkrementelle Propagierung diese Kosten pro Knoten bzw. Kante, aber über alle Iterationen hinweg, besitzt. Bei der Berechnung der Größe haben wir für den CI-Graphen und den CS-Graphen jeweils n Knoten und $\mathcal{O}(m * g)$ Kanten angegeben. Damit erhalten wir die Kosten $\mathcal{O}(t_{max} * (n + m * g))$ für die beiden abschließenden Propagierungsstufen. Mit Lemma 10.2.1 und Lemma 10.2.2 ergibt sich das folgende Resultat:

Satz 10.4.3 *Die Laufzeit der kombinierten Analyse mit MOVP-Präzision in der Propagierung von Zeigerzielen liegt in $\mathcal{O}((t_{max} + g) * g^2 * |CG|)$ mit inkrementeller Propagierung und in $\mathcal{O}(i * (t_{max} + g) * g^2 * |CG|)$ mit erschöpfender Propagierung.*

Korollar 10.4.4 *Mit $t_{calls} \ll n_{ir}$ liegt die Laufzeit der so erweiterten (inkrementellen) Analyse in $\mathcal{O}(n_{ir}^4)$.*

Beweis: Wir haben $t_{max}, g, f, c, t_{calls} \ll n_{ir}$. □

Die Korrektheit und Präzision der präziseren Propagierung ergibt sich aus den für IFDS bekannten Resultaten diesbezüglich. Zusammen mit dem grundsätzlichen Korrektheitsbeweis für unsere Analyse haben wir also auch die These 2 gezeigt:

> *Es gibt eine fluss-sensitive Zeigeranalyse, die sowohl starke Aktualisierungen beherrscht als auch die MOVP-Präzision (in der Propagierung) erreicht und die Komplexität $\mathcal{O}(n^4)$ besitzt.*

10.4.3 Vergleich zum Framework-Ansatz

Zum Abschluss wollen wir die Resultate mit denjenigen vergleichen, die man für die Anwendung des auf ISSA übertragenen IFDS-Frameworks erhält. Vermutlich lässt sich auch dafür eine inkrementelle Umsetzung erreichen; da der Framework-Ansatz jedoch bereits im Vergleich der erschöpfenden Varianten schlechter abschneidet (und wir keine Option sehen, die Ursache dafür durch den Übergang auf eine inkrementelle Fassung zu beseitigen), haben wir keine Bestrebungen in diese Richtung unternommen. Entsprechend vergleichen wir hier auch nur die erschöpfenden Varianten der beiden Alternativen.

Zunächst stellen wir fest, dass die Same-Level-Pfade und Summary-Kanten im Framework-Fall zwischen erweiterten ISSA-Knoten verlaufen. Wir multiplizieren daher die für den Drei-Stufen-Ansatz bekannten Zahlen jeweils mit der Anzahl an Fakten $\mathcal{O}(t_{max})$. Der erweiterte ISSA-Graph hat somit $\mathcal{O}(n * t_{max})$ Knoten, und dank der einfachen Transferfunktionen auch $\mathcal{O}(m * t_{max})$ Kanten.

Eine Summary-Kante kann zwar zwischen beliebigen Unteraus- und -eingängen einer Aufrufstelle verlaufen, jedoch stets mit dem gleichen Fakt am Unteraus- und -eingang. Pro Aufrufstelle haben wir damit maximal $\mathcal{O}(g^2 * t_{max})$ Summary-Kanten.

Auch für die Zahl der Same-Level-Pfade nutzen wir aus, dass der Fakt der gleiche bleibt (abgesehen von den Sonderfällen an Zeigerziel-Quellen). Damit haben wir pro erweitertem ISSA-Knoten maximal $\mathcal{O}(g)$ eingehende Same-Level-Pfade. Dies wiederum begrenzt die Zahl der Schleifendurchläufe zur Abarbeitung der Arbeitsliste auf $\mathcal{O}(g * n * t_{max})$ Durchläufe. Die Kosten pro Unterausgang sind gleich zum Drei-Stufen-Ansatz, jedoch haben wir diesmal $\mathcal{O}(g * c)$ Unterausgänge mit jeweils t_{max} Fakten, für die für bis zu $\mathcal{O}(g)$ eingehende Same-Level-Pfade ein Aufruf erfolgt. Das ergibt $\mathcal{O}(g^2 * c * t_{max})$ Aufrufe und entsprechend zusammen die Kosten $\mathcal{O}(g^2 * |CG| * t_{max} + g^3 * c * t_{max})$.

Analog werden nun die Aktionen für einen Ausgang $\mathcal{O}(g^2 * f * t_{max})$ Mal angestoßen. Damit verursachen sie zusammen die Kosten $\mathcal{O}(g^2 * |CG| * t_{max} + g^3 * |CG| * t_{max})$. Insgesamt erhalten wir als Resultat:

Satz 10.4.5 *Eine Ausführung des auf ISSA übertragenen IFDS-Frameworks mit den Transferfunktionen der Zeigerziel-Propagierung hat die Komplexität* $\mathcal{O}(g^3 * |CG| * t_{max})$.

Im Vergleich zum Drei-Stufen-Ansatz haben wir hier einen Faktor mehr, d.h. insgesamt eine Dimension mehr. Der Drei-Stufen-Ansatz ist damit bereits in der erschöpfenden Variante asymptotisch effizienter. Mathematisch gesehen liegt der Grund darin, dass die Größe t_{max} hier multiplikativ, im Drei-Stufen-Ansatz jedoch additiv auftaucht.

Kapitel 11

Empirische Evaluation

Dieses Kapitel schildert den Kontext unserer experimentellen Evaluation der kombinierten Analyse. Zunächst diskutieren wir, welche Aspekte sinnvollerweise evaluiert werden sollten. Anschließend überlegen wir eine geeignete Methodik, um diese Ziele zu erreichen, und beschreiben die analysierten Programme sowie das Testsystem. Die nachfolgenden Kapitel besprechen dann die Resultate und die Effizienz der Analyse gemäß unseren Versuchen.

11.1 Ziele der Evaluation

Es soll eben nicht (nur) gemessen werden, was einfach zu messen ist. Stattdessen ist es notwendig, vorab zu überlegen, welche Aspekte evaluiert werden sollten. Dies sind für uns:

- Korrektheit der Implementierung: Ist die Implementierung inkorrekt, so sind alle Zahlen wertlos (auf Grund der Unstetigkeit, also den durchaus großen Auswirkungen kleiner Fehler, sind die Resultate bei Fehlern vermutlich auch nicht annähernd korrekt). Gleichzeitig überprüfen wir hier auch experimentell die theoretische Korrektheit der Analyse.

- Skalierbarkeit: Für die Frage, wie gut die Analyse in der Praxis tatsächlich anwendbar ist, ist neben der theoretischen Aussage zur Skalierbarkeit auch eine Evaluation notwendig, da die Theorie nur asymptotische

Abschätzungen und keine konstanten Faktoren liefert. Hierunter zählen neben Laufzeit und Speicherbedarf – die sehr von der Implementierung und der Hardware abhängen – auch im Ansatz verankerte Eigenschaften wie die Anzahl an Iterationen, Knoten und Kanten.

- Differentielle Skalierbarkeitsanalyse: Welche Stellen oder Aspekte sind für die Skalierbarkeit kritisch? Wo wird die meiste Zeit, der meiste Platz verbraucht?

- Präzision der Resultate: Wie gut sind die fluss-sensitiven Resultate gegenüber den fluss-insensitiven? Welche Auswirkungen hat die Feld-Sensitivität?

- Differentielle Präzisionsanalyse: Welche Arten von Ungenauigkeit gibt es, und wie stark treten sie jeweils auf? Welche Auswirkungen haben sie jeweils? Wie vergleichen sich die verschiedenen Strategien zur Lösung von Teilproblemen, z.B. Feld/Fluss-(In)Sensitivität, und warum?

- Überprüfung und Präzisierung der theoretischen Resultate: Auch bei einem Papier-Beweis können sich Fehler einschleichen; es ist daher ratsam, die berechneten Resultate empirisch zu validieren. Gleichzeitig kann man damit untersuchen, wie gut die theoretische Seite die praktischen Fälle tatsächlich modelliert. Da viele theoretischen Aussagen asymptotisch sind und von Konstanten abstrahieren, sollte ihre Aussagekraft gut überprüft werden, speziell für nicht ganz so große Programme.

- Auswirkung der Varianten: Die Evaluation soll untersuchen, welche Auswirkungen die beschriebenen verschiedenen Implementierungsvarianten auf die vorgenannten Aspekte haben. Beispielsweise wollen wir untersuchen, ob die inkrementelle Propagierung auch in der Praxis schneller ist als die erschöpfende Propagierung.

11.2 Evaluationsszenario

11.2.1 Integration in Bauhaus

Wir haben die Kernanalyse im Rahmen des Bauhaus-Projekts [Koschke 2008; Raza u. a. 2006] implementiert und evaluiert. Das Projekt Bauhaus ist eine Kooperation der Abteilung Programmiersprachen des Instituts für Softwaretechnik der Universität Stuttgart und der Universität Bremen. Die Firma Axivion verkauft Teile davon an Kunden, über die entsprechende Rückmeldungen aus der Praxis einfließen. Ziel des Projekts ist die Entwicklung von Methoden und Werkzeugen zur Unterstützung und Erleichterung des Programmverstehens bei der Software-Wartung. Hierzu wurde eine umfangreiche Infrastruktur geschaffen, die es erlaubt, Programmanalysen auf hohen und niedrigen Abstraktionsebenen durchzuführen.

Bauhaus verfügt über ein Frontend auf Basis des bekannten EDG-Frontends, um Programme in vielen Dialekten der Sprachen C und C++ in die Bauhaus-eigene IR namens IML (*I*nter*M*ediate *L*anguage) zu übersetzen. Ein vom Autor entwickelter effizienter Linker verbindet anschließend die einzelnen IML-Dateien zu einer einzigen IML für das Gesamtprogramm. IML vereint dabei einen abstrakten Syntaxbaum (AST) mit anderen Zwischendarstellungen (z.B. ICFG) und Analyse-Resultaten, die als Attribute an die Knoten des Syntaxbaumes annotiert werden.

Vor der Anwendung unserer Kernanalyse haben wir für jedes Testprogramm die IML erstellt und die in Bauhaus vorhandene intraprozedurale Kontrollfluss-Analyse zum Aufbau der lokalen CFG samt ihrer Dominanzbäume angewendet. Anschließend haben wir daraus eine niedere IR im Stile der in Kapitel 3 beschriebenen Zwischendarstellung generiert. Hierbei können wir sowohl eine feld-sensitive als auch eine feld-insensitive Variante generieren, und zwar derart, dass die nachfolgende Analyse keine Unterscheidung diesbezüglich treffen muss. Auf dieser IR arbeitet die Kernanalyse und ergänzt die Informationen zu Datenfluss, Zeigerzielen und indirekten Aufrufen. Die Resultate können anschließend wieder in die IML übernommen werden oder als Basis weiterer Analysen auf der niederen IR dienen.

Die Kernanalyse selbst sammelt statistische Daten nur zu Größen, die nach der Ausführung nicht mehr messbar sind. Dies sind insbesondere die Metriken zu Laufzeit und Platzbedarf. Für die Ermittlung aller übrigen Messwerte haben wir ein weiteres Werkzeug implementiert, welches sowohl die Resultatsdateien der Analyse als auch Eingabedateien (zur Ermittlung der Größe der Eingabe) vermisst. Schließlich wurde noch ein Werkzeug entwickelt, welches alle Messwerte unabhängig von der erhebenden Quelle zusammen betrachtet, abgeleitete Größen aus diesen errechnet (wie z.B. den prozentualen Vergleich eines Messwertes für zwei verschiedene Varianten der Analyse) und daraus schließlich die Tabellen und Diagramme für die Evaluation generiert.

Die Implementierung der Kernanalyse und der Hilfswerkzeuge erfolgte wie für die meisten Bauhaus-Werkzeuge in der Sprache Ada95 und umfasst etwas mehr als 14,5 KSLoC. Für den Umgang mit IML wurde die in Bauhaus realisierte IML-Klassenbibliothek mit knapp 300 Klassen und mehr als 400 KSLoC genutzt. Die Bibliothek für die niedere IR kommt dagegen mit weniger als 3 KSLoC aus. Weiterhin nutzt die Implementierung einige generische Datentypen aus einer in Bauhaus vorhandenen Bibliothek, welche viele Elemente einer vor der Einführung von Ada2005 fehlenden Standardbibliothek enthält und noch einmal mehr als 35 KLoC Code beisteuert. Die Werkzeuge zur Auswertung und Aufbereitung der Resultate umfassen etwas mehr als 5 KSLoC, und das Werkzeug zur Ermittlung der niederen IR bringt ebenfalls noch einmal 5,2 KSLoC auf die Waage.

Zur Durchführung unserer Messungen haben wir Bauhaus und damit auch die Kernanalyse mit dem GNAT-Pro-Compiler von Adacore übersetzt. Assertions waren dabei aktiviert, und für Optimierungen wurde die Stufe O2 gewählt.

Bauhaus verändert sich ständig, so dass eine genaue Angabe der benutzten Version für die Wiederholbarkeit der Experimente wichtig ist. Zudem können so Änderungen identifiziert werden, die einen Einfluss auf die Resultate der Analyse haben. Der Bauhaus-Quellcode wird derzeit in einem SVN-Repository verwaltet (Versionsverwaltung). Unseren Experimenten lag Bauhaus in der SVN-Versionsnummer 27820 zu Grunde.

11.2.2 Ausprägungen der Analyse

Unsere Implementierung beherrscht die folgenden Ausprägungen der Kernanalyse:

- Feld-Sensitivität ja/nein (über entsprechende IR-Generierung)
- Fluss-Sensitivität ja/nein
- Bei Fluss-Sensitivität wahlweise Unterstützung indirekter starker Aktualisierungen
- Propagierung inkrementell oder erschöpfend
- Bei erschöpfender Propagierung wahlweise mit vorausgehender erschöpfender oder inkrementeller Zyklenkontraktion
- Mit Zyklenkontraktion kann die Propagierung wahlweise vorwärts oder rückwärts erfolgen, ansonsten nur vorwärts

Wir konstruieren dabei in den Iterationen den reduzierten ISSA-Graphen, der nur die allgemeinen Definitionen enthält. Am Ende wird daraus der vollständige ISSA-Graph in einem Durchgang ermittelt. Diese Ergänzung der allgemeinen Verwendungen haben wir mit dem Pruning kombiniert, damit nicht erst Knoten ergänzt werden, die später dann wieder dem Pruning zum Opfer fallen. Die ISSA-Kanten sind stets vorwärts orientiert und die Suche nach der gültigen Definition nutzt pro lokalem Objektgraphen eine verkettete Liste von Knoten.

Wir hatten auch eine Variante implementiert, welche eine inkrementelle kontext-abhängige Propagierung ähnlich der Ideen aus Kapitel 9 realisiert. Diese zeigte sich jedoch selbst für einige kleine Programme bereits ineffizient, so dass wir sie nicht mit in die Evaluation aufgenommen haben (wir hätten aufgrund der langen Laufzeiten keine Zahlen für die größeren Testprogramme präsentieren können.) Als Grund vermuten wir unter anderem das Fehlen einer geeigneten Zyklenkontraktion sowie eben die höhere prinzipielle Komplexität. Wir gehen davon aus, dass eine Umsetzung auch die Ideen zur kontext-abhängigen Datenfluss-Darstellung realisieren sollte, um

den Graphen kleiner (und präziser) zu machen. Dies wäre ein interessantes Thema für eine weiterführende Arbeit.

Bei der Auswertung der Ergebnisse werden wir uns daher auf die sechs verschiedenen Präzisionsstufen konzentrieren, die sich als Kombination aus den ersten drei der oben angeführten Dimensionen ergeben. Bei der Betrachtung der Effizienz vergleichen wir zusätzlich die effizienteste Realisierung (mit inkrementeller Zyklenkontraktion) mit den anderen Implementierungsvarianten, die sich nur auf Laufzeit und Platzbedarf auswirken.

11.2.3 Hardware-Umgebung

Die Experimente fanden auf dem Abteilungsrechner pslx4 statt. Dieser ist ein 64-Bit-System und verfügt über 8 AMD Opteron-Prozessoren, die jeweils mit 3 GHz getaktet sind. Da keine Parallelisierung vorgenommen wurde, läuft die Analyse jeweils nur auf einem einzelnen Prozessor. Insgesamt stehen auf diesem System 64 GB Arbeitsspeicher zur Verfügung. Das Betriebssystem des Rechners ist Debian Linux.

11.3 Methodik

Um die Ziele der Evaluation zu erreichen und dabei störende, verschleiernde Effekte möglichst auszuschalten und gleichzeitig die Wiederholbarkeit und Nachvollziehbarkeit zu gewährleisten, sind wir wie folgt vorgegangen. Pro Testprogramm führen wir diese Schritte durch:

1. IML-Generierung aus dem Quellcode mittels Frontend und Linker
2. Lokale Kontrollflussanalyse: CFG und Dominanzbäume
3. Generierung der niederen IR
4. Vermessung der Eingabe
5. Ausführung der Kernanalyse
6. Vermessung des Resultats

Die letzten beiden Schritte werden dabei für alle betrachteten Analyse-Konfigurationen durchgeführt. Abschließend erfolgt dann der Vergleich der Resultate aus diesen verschiedenen Konfigurationen.

Wie bereits beschrieben, finden die meisten Messungen hierbei in zwei separaten Werkzeugen auf der als Resultat generierten Datei statt. Damit reduzieren wir Einflüsse der Messungen auf die Analyse und können diese Messungen unabhängig von der Ausführung der Analyse anwenden, also z.B. auch zur Kontrolle und zum späteren Nachvollziehen ohne erneutes Analysieren. Die automatische Steuerung des gesamten Vorgangs von der IML-Erzeugung bis zur Generierung der Tabellen und Diagramme dieser Dissertation über Shell-Skripte und Makefiles helfen dabei, Inkonsistenzen zu vermeiden. Gleichzeitig sind die Resultate und Einzelschritte mit minimalem manuellen Aufwand wiederholbar. Die Messungen erfolgen bis auf die abschließenden Vergleiche jeweils unabhängig voneinander und von früheren Messungen derselben Konfiguration auf dem gleichen Programm.

Die Messung der Laufzeit unterliegt stets Schwankungen aufgrund der unterschiedlichen Belastung des Rechners durch verschiedene Prozesse. Unser Testrechner wurde während der Messungen fast ausschließlich für die Durchführung der Analyse verwendet, so dass wir auf mehrfache Messungen verzichtet haben. Die Messung selbst erfolgte dabei programmseitig über die vom Linux-Betriebssystem zur Verfügung gestellte „Uhr" in Jiffies und wurde in Millisekunden umgerechnet. Laufzeiten aus Kernel-Teilen sind dabei weitgehend vom System herausgerechnet. Mit diesen Maßnahmen erreichen wir, dass nur die Zeit der Analyse gemessen wird, weitgehend ohne störende Einflüsse.

Der Speicherbedarf wird ebenfalls mit Hilfe der Prozess-Informationen des Betriebssystems ermittelt. Das System kann jedoch nicht von sich aus das Maximum liefern. Stattdessen müssen wir an allen relevanten Stellen den aktuellen Verbrauch abfragen. Dadurch kann es auch geschehen, dass wir einen temporär höheren Speicherbedarf nicht erfassen. Eine stichprobenhafte gleichzeitige Beobachtung des mit dem Programm *top* laufend gemessenen Verbrauchs ergab jedoch keine Auffälligkeiten.

Für eine abstraktere Messung der Laufzeit haben wir gezählt, welche Schritte wie oft ausgeführt werden. Neben der Zahl der Iterationen fand hier eine Erhebung der Zahl der besuchten Kanten im Rahmen der Propagierung statt. Diese Größen haben den Vorteil, dass sie im Algorithmus verankert sind und nicht den Schwankungen des Systems unterliegen. Neben der Übertragbarkeit auf andere Systeme sehen wir in den abstrakteren Messgrößen dabei auch den Vorteil, Informationen über das Mengengerüst und damit potentiell geeignete Datenstrukturen und Teilverfahren zu erhalten.

Für die Evaluation haben wir zwei Sorten von Testprogrammen eingesetzt:

1. Reale Programme für eine möglichst realistische Bewertung der Analyse in der Praxis
2. Handgeschriebene, kleine Testprogramme zum vollständigen Test einzelner Konstrukte

Das finale Analyseverhalten und die Resultate hängen dabei stets von einer Vielzahl an Faktoren ab, die sich kaum einzeln untersuchen lassen. Die Resultate sind daher immer ein Mix aus verschiedenen Einflussfaktoren. Gleichzeitig sind Algorithmen wie die Kernanalyse notorisch unstetig: eine kleine Änderung am zu analysierenden Programm oder am Algorithmus kann große Auswirkungen auf das Resultat haben. Die Übertragung der Resultate von den hier untersuchten Programmen auf andere Programme ist daher riskant; wir können nur exemplarische Resultate präsentieren. Jedoch komplementiert unsere theoretische Evaluation diese Situation, indem sie Resultate hergeleitet hat, die für alle Programme gelten. So können wir z.B. guten Gewissens wegen der kubischen Komplexität von einer gewissen generellen Skalierbarkeit ausgehen, wohingegen BDD-basierte Verfahren diese Zusicherung nicht geben können. Die theoretische Untersuchung sowie die Auswahl realer Programme helfen außerdem dabei, die Gefahr einer künstlichen, nicht übertragbaren Laborsituation bestmöglich auszuräumen. Die Auswahl der realen Testprogramme versucht dabei, ein möglichst breites Spektrum an verschiedenen Eingabesituationen für die Analyse abzudecken. Damit soll die Gefahr umgangen werden, durch die besondere Auswahl gewisse Einflussfaktoren zu

begünstigen oder zu unterdrücken. Der Abschnitt 11.4 liefert hierzu nähere Informationen.

Die Korrektheit der Implementierung als Grundvoraussetzung für sinnvolle Resultate haben wir dabei über mehrere Maßnahmen untersucht:

- Code-Inspektion/Review
- Assertions im Code
- Vergleich von Soll und Ist für die handgeschriebenen Testfälle
- Vergleich der Resultate verschiedener Konfigurationen, die eine Inklusionsbeziehung einhalten müssen: So sollte z.B. der IDFG der FI-Zeigeranalyse eine Obermenge des FS-IDFG sein
- Inspektion der Resultate für viele Teile aller eingesetzten realen Programme

Es ist klar, dass dies jeweils nur Fehler aufdecken kann und keine absolute Sicherheit gibt. Ein vollständiger Korrektheitsbeweis für die Implementierung liegt jedoch jenseits des Machbaren, und das hier gewählte Vorgehen entspricht bekannten, anerkannten Strategien.

11.4 Analysierte Programme

Auf den folgenden Seiten findet sich eine kurze Beschreibung der analysierten Programme. Zu jedem Programm versuchen wir, relevante Charakteristiken zu erfassen.

Zunächst listet Tabelle 11.1 die verwendeten Programme auf. Die Liste der Programme ist hierbei alphabetisch sortiert. Neben dem Programmnamen geben wir auch die Versionsnummer an. Die Kurzbeschreibungen zeigen, dass wir uns nicht auf Programme eines bestimmten Anwendungsgebietes beschränkt haben. Alle Programme sind Open-Source-Programme in der Sprache C.

Die weiteren Tabellen und Diagramme in diesem Abschnitt beschreiben die Größe dieser analysierten Programme mit verschiedenen Metriken. Die

Programm	Version	Beschreibung
bash	3.1	Shell
bc	1.06	Kommandozeilen-Rechner
bison	2.3	Parsergenerator
clara	2003-12-14	OCR-Programm
cook	2.26	Kompilationswerkzeug
flex	2.5.35	Scannergenerator
fvwm	2.5.26	Fenster-Manager
grep	2.5.1a	Kommandozeilen-Programm für Suche mit regulären Ausdrücken
less	418	Textdateibetrachter in der Konsole
mc	4.5.55	Midnight-Commander für Verzeichnis- und Dateioperationen
mutt	1.5.18	E-Mail-Programm (Konsole)
nano	1.2.3	Editor
nethack	3.4.3	Adventure-Spiel
sed	4.1	Stream-Editor
tcc	0.9.23	C-Compiler
tcsh	6.15.00	Shell
time	1.7	Werkzeug zur Messung der Laufzeit
units	1.86	Umrechnung von Einheiten
wget	1.10.1	Lädt Webseiten und Daten herunter
zsh	4.3.6	Shell

Tabelle 11.1: Die analysierten Programme

Programm	LoC	SloC	Instr. (insens.)	Instr. (sens.)	Funktionen	Direkte Aufrufe	Indirekte Aufrufe
time	2.322	1.627	1.000	1.002	25	138	2
units	5.761	3.442	6.046	5.970	82	615	3
nano	13.750	13.322	16.916	17.296	212	2.241	2
bc	14.371	10.139	10.955	10.955	149	1.024	21
flex	26.207	22.563	26.364	26.063	304	2.461	1
less	26.343	17.834	24.308	23.043	473	2.053	5
sed	30.920	28.293	26.106	26.264	270	1.741	1
grep	31.502	24.818	16.843	16.868	178	1.031	11
wget	39.365	29.626	46.969	46.805	539	3.661	23
bison	42.856	77.596	43.433	44.431	1.015	3.304	209
tcc	51.598	43.496	30.343	30.140	330	2.496	3
clara	56.472	30.367	76.623	74.995	468	4.251	12
tcsh	71.086	54.927	73.595	73.971	1.049	7.438	27
cook	108.363	68.488	52.854	53.650	1.564	6.263	75
mutt	108.425	83.572	137.229	136.812	1.441	11.651	657
zsh	121.570	95.673	119.369	119.400	1.056	8.218	436
bash	133.578	108.687	154.646	151.781	2.397	11.846	98
mc	178.336	126.943	115.932	116.559	2.180	11.598	200
fvwm	211.794	181.415	172.186	173.399	1.883	10.029	93
nethack	261.671	214.127	450.392	446.288	2.610	27.107	849

Tabelle 11.2: Größe der analysierten Programme

Tabelle 11.2 ist dabei nach der wohl bekanntesten Metrik *Lines of Code* sortiert. Diese Metrik haben wir mit dem bekannten Werkzeug *wc* bestimmt (angewendet auf alle Dateien mit den Endungen c und h). Zusätzlich haben wir mit dem Werkzeug `sloccount` von David Wheeler die Metrik *Source Lines of Code (SLoC)* erhoben, bei der Leerzeilen nicht gezählt werden. Doch auch diese Metrik kann kritisiert werden: Sie zählt die Größe aller Dateien in einem Verzeichnis, unabhängig von der Programmiersprache (welchen Einfluss haben Shell-Skripte im Vergleich zu C-Quelldateien auf die Größe?) und unabhängig davon, wie die Dateien verwendet werden (manche fließen nicht ins fertige Programm ein, andere – wie Headerdateien – mehrfach). Quellcode, den das Programm aus anderen Verzeichnissen hinzuzieht (v.a. Include-Dateien von Bibliotheken) fließt gar nicht ein, obwohl er einen erheblichen Anteil ausmachen kann. Innerhalb einer Datei werden Leerzeilen nicht gezählt, aber z.B. Makros, die ja typischerweise an vielen Stellen expandiert werden, zählen nur einfach. Dies ist allgemeiner bekannt als ein Problem mit dem Präprozessor: zählt man vor oder nach Ausführung des Präprozessors? Unsere Metrik wurde vor dessen Einsatz erhoben, da ansonsten Headerdateien zigfach zählen.

Aufgrund dieser Kritiken haben wir zum Vergleich noch eine weitere bekannte Metrik aufgeführt: Die Zahl der Instruktionen in der IR. Diese haben wir nach Feld-Sensitivität bzw. -Insensitivität getrennt angegeben, da sie nach der IR-Generierung erhoben wird. Diese Metrik vermisst dann zwar die tatsächliche Eingabegröße für die Analyse, ist jedoch leider nicht zwischen verschiedenen Analyse-Werkzeugen vergleichbar. Somit ergänzen sich (S)LoC und diese Metrik. Die Betrachtung der Tabelle zeigt uns, dass die feld-sensitive IR durchaus kleiner als ihr insensitives Gegenstück werden kann, wenn die feinere Modellierung entsprechende Optimierungen erlaubt. Außerdem sehen wir, dass die Größe nach LoC sowie die Größe nach Instruktionen differieren, sowohl in der sich ergebenden Sortierung als auch in der Größenordnung. So vertauscht sich z.B. die Platzierung von bc und nano bei einer Sortierung nach Instruktionen anstelle der benutzten Sortierung. Außerdem beträgt die Differenz zwischen den beiden größten Programmen nur etwa 50 KLoC, aber mehr als 250.000 Instruktionen.

Alle weiteren Tabellen, Diagramme und Schaubilder sind nach dieser Metrik (Zahl der Instruktionen) sortiert, weil sie die tatsächliche Eingabegröße besser wiedergibt.

Die Tabelle beschreibt weiterhin die initiale Größe des Aufrufgraphen, indem sie die Anzahl f an Unterprogrammen angibt, für welche die Zwischendarstellung einen Rumpf enthält. Außerdem zählt sie die Zahl der Aufrufstellen c getrennt nach direkten und indirekten Aufrufen.

Die Tabelle 11.3 betrachtet die Objekte der Testprogramme, sowohl feld-insensitiv als auch feld-sensitiv. Dazu differenzieren wir nach verschiedenen Objektarten. Angegeben ist, wie viele Objekte jeweils Stack-, Array-, Heap- oder Rückgabeobjekte sind. Außerdem sehen wir die Zahl der Objektgruppen (vgl. Kapitel 3 für die verschiedenen Objektarten). Für die Gesamtzahl an Objekten fehlen dann jeweils noch die Unterprogramme (deren Zahl aus der vorigen Tabelle ersichtlich ist), die besonderen Objekte für Null, Unknown_Objects und Strings (sofern benötigt), sowie die Hilfsobjekte, die stets fluss-insensitiv behandelt werden können. Deren Zahl erkennen wir im Diagramm 11.1.

Bei der Betrachtung der Objekte erkennen wir insbesondere (und wenig überraschend), dass die einfachen Hilfsobjekte die Mehrheit der Objekte ausmachen. Eine gesonderte fluss-insensitive Behandlung dieser Objekte in der fluss-sensitiven Ausprägung lohnt sich daher. Die Verbindung der Verwendungen solcher Hilfsobjekte zur jeweiligen Definition geschieht dabei schon in der Initialisierung, da Hilfsobjekte per Definition kein (auch nur potentielles) Zeigerziel sind und nur genau eine Definition besitzen. Die Kosten für diese Objekte sind damit gering. Die Zahl der Heapobjekte (d.h. der Allokationsstellen) variiert zwischen den einzelnen Programmen, der Anteil an allen Objekten ist jedoch gering.

Erfreulicherweise ist der Zuwachs bei der Zahl der feld-sensitiv angelegten Objekte überschaubar, wobei die Modellierung vor allem den Anteil der Heapobjekte nach oben treibt (als Faktor tritt dabei jeweils die Zahl der ausgeflachten Felder der größten Struktur auf). Dagegen sinkt die Zahl der Objektgruppen, weil nun nicht mehr die statische Initialisierung einer Struktur zusammengefasst werden kann.

	Feld-insensitiv					Feld-sensitiv				
Programm	Stack	Array	Heap	Rückgabe	Objektgruppen	Stack	Array	Heap	Rückgabe	Objektgruppen
time	94	2	1	4	8	110	5	29	4	2
units	430	27	2	50	27	465	34	58	50	5
bc	730	30	11	51	9	735	33	319	51	3
grep	1.263	55	20	85	66	1.400	62	580	85	7
nano	1.204	12	1	108	25	1.515	18	32	125	1
less	1.874	64	10	222	138	2.067	86	290	222	36
sed	2.167	41	44	177	16	2.459	59	1.716	177	4
flex	1.539	127	37	129	114	1.600	132	1.073	129	3
tcc	2.121	61	1	117	23	2.414	91	51	117	1
bison	4.488	150	9	484	32	5.401	174	954	490	5
wget	3.277	127	3	311	206	3.739	161	294	311	12
cook	4.949	136	9	590	158	5.938	172	261	590	5
tcsh	5.738	386	3	495	134	6.706	416	93	495	18
clara	5.703	190	9	183	95	6.038	300	396	183	15
mc	10.198	415	49	1.023	604	12.592	752	4.508	1.031	84
zsh	9.087	178	5	624	600	10.178	215	210	632	30
mutt	10.143	621	12	866	412	12.136	689	528	867	61
bash	13.617	249	24	1.472	1.243	14.960	318	696	1.472	121
fvwm	15.057	286	12	694	403	29.217	391	8.208	694	97
nethack	33.273	1.004	139	1.216	754	35.266	1.405	19.043	1.216	103

Tabelle 11.3: Objekte

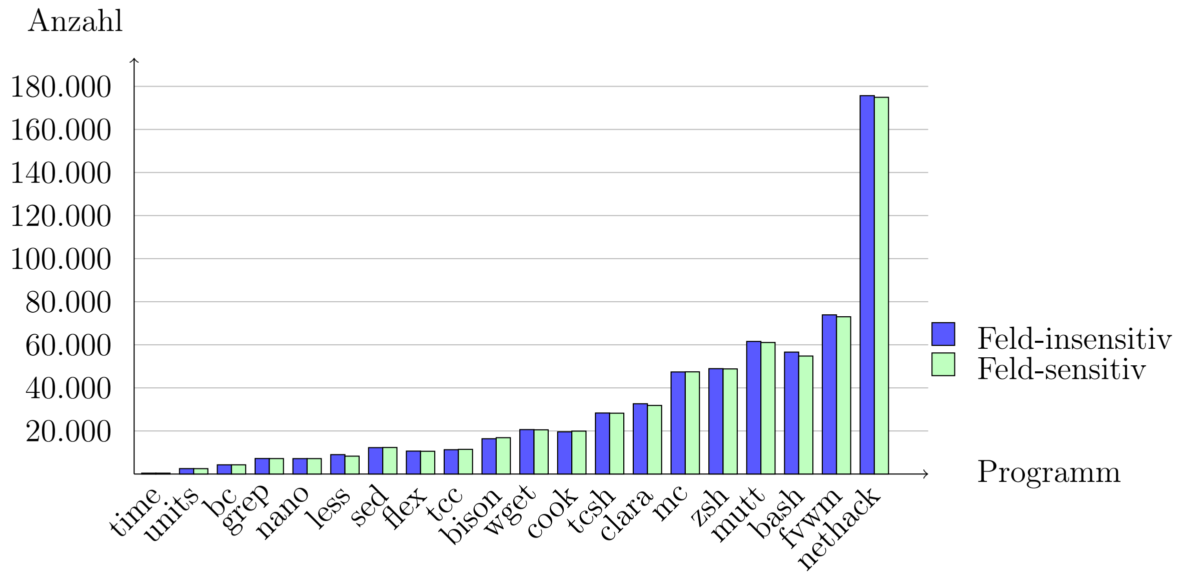


Abbildung 11.1: Hilfsobjekte

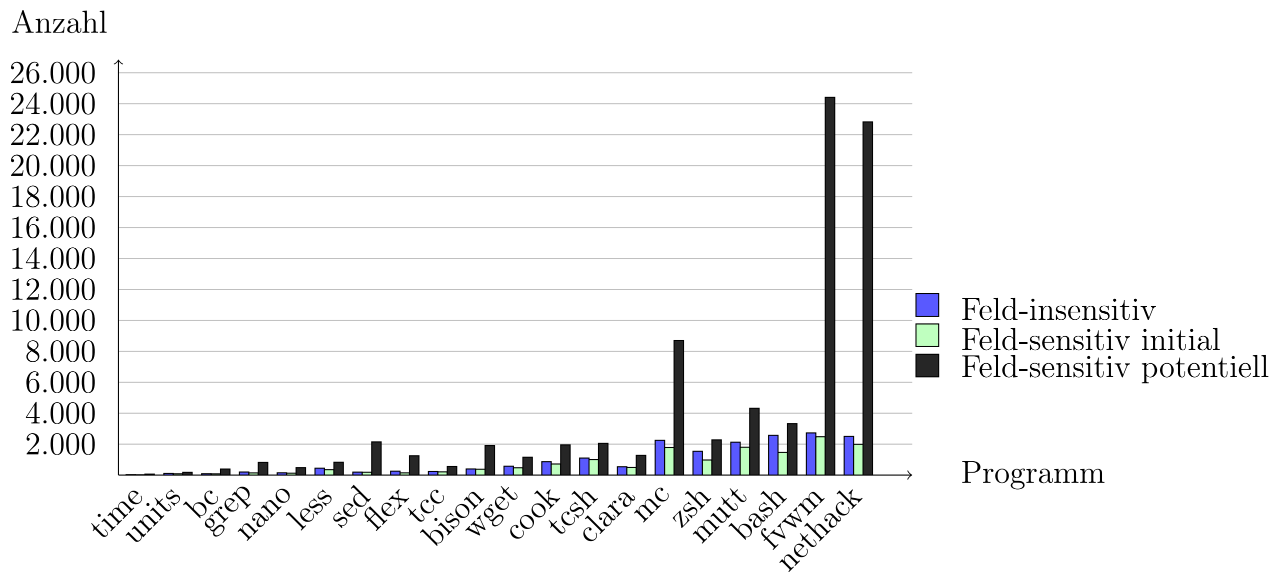


Abbildung 11.2: Initiale und potentielle Zeigerziele

Für die Zeigeranalyse relevant ist die Zahl der potentiellen Ziele. Diagramm 11.2 bietet uns diese Größe, wobei es feld-sensitiv nach potentiellen und bereits am Anfang aktuellen Zeigerzielen unterscheidet. Feld-insensitiv wächst die Menge der aktuellen Zeigerziele nicht, so dass beide Zahlen identisch wären. Feld-sensitiv dagegen sehen wir große Unterschiede in den beiden Zahlen, d.h. die Menge der Ziele kann hier noch spürbar anwachsen im Verlauf der Analyse.

Schließlich betrachten wir mit der Tabelle 11.4 weitere relevante Eingabegrößen für den Zeigeranalyse-Teil. Gezeigt ist die Zahl der Instruktionen, die markante Stellen für die Zeigeranalyse sind: Dereferenzierungen (aufgetrennt nach indirekten Definitionen und Verwendungen) und Adressnahmen (feld-sensitiv auch indirekt möglich wie in `&(p->x)`). Die Zahl der indirekten Adressnahmen ist in allen Programmen relativ gering. Wie man sieht, ist die Zahl der Adressnahmen in vielen Fällen höher als die Zahl der Zeigerziele, was unser Vorgehen in der Propagierung bestärkt, Zeigerziel-Quellen und Propagierungselemente zu unterscheiden.

Die verschieden genaue Modellierung in der IR wirkt sich auf die Anwendbarkeit (und Effektivität) von Optimierungen der IR aus. Beispielsweise taucht feld-insensitiv durch die ungenauere Modellierung öfter als feld-sensitiv das gleiche Objekt in benachbarten Instruktionen auf, was dann die Streichung doppelter Anweisungen oder anderweitige Zusammenfassungen einiger Instruktionen (z.B. im Zusammenhang mit Objektgruppen, ähnlich zu Beispiel 3.1.1) erlaubt. Tatsächlich sehen wir in der Tabelle, dass die Zahl der indirekten Definitionen bzw. Verwendungen feld-sensitiv größer ist.

Bei den direkten Adressnahmen dagegen beobachten wir auch Effekte, welche feld-sensitiv die Zahl der Adressnahmen reduzieren. So prüfen wir beispielsweise für jedes Objekt global, ob es potentiell irgendwo verwendet wird; falls nicht, können Definitionen des Objekts gestrichen werden. Die feinere Modellierung kann erreichen, dass dies öfter der Fall ist. Das trifft insbesondere gerne Felder von statisch initialisierten Strukturen, denen eine Adressnahme zugewiesen wird. Die Streichung der unnötigen Definition reduziert somit die Zahl der Adressnahmen, während feld-insensitiv die Verwendung eines anderen Feldes der Struktur dieselbe Optimierung verhindert.

So erklären sich die Differenzen, die wir in der Tabelle sehen können.

Diese verschiedenen Metriken zur Eingabegröße werden uns helfen, die Messwerte zu den Resultaten besser zu verstehen. Wie man erkennt, deckt unsere Auswahl an zu analysierenden Programmen ein breites Spektrum unterschiedlicher Werte in den Metriken ab. Damit können wir das Verhalten der Analyse für verschiedenste Konstellationen untersuchen und laufen nicht Gefahr, durch ein begrenztes Eingabespektrum Aussagen herzuleiten, die nur für wenige Programme gelten. Wir sehen auch, dass die ausgewählten Programme durchaus groß sind, also eine Herausforderung an die Effizienz darstellen und zur realistischen Überprüfung der Skalierbarkeit dienen können.

	Feld-insensitiv			Feld-sensitiv			
Programm	Ind. Definitionen	Ind. Verwendungen	Direkte Adressnahmen	Ind. Definitionen	Ind. Verwendungen	Ind. Adressnahmen	Direkte Adressnahmen
time	6	82	123	28	82	3	122
units	173	437	970	173	437	21	946
bc	323	944	1.288	329	951	2	1.283
grep	458	1.344	1.746	553	1.362	18	1.744
nano	236	888	2.608	236	888	10	2.987
less	377	834	3.213	380	837	11	3.263
sed	876	2.658	2.948	912	2.708	60	2.983
flex	787	1.610	3.803	794	1.610	4	3.722
tcc	799	1.968	2.924	908	2.112	122	2.887
bison	1.804	5.032	4.498	1.952	5.549	101	4.779
wget	1.049	2.276	5.768	1.058	2.284	23	5.819
cook	1.176	4.161	6.165	1.409	4.290	627	6.523
tcsh	1.268	4.340	11.306	1.306	4.354	99	11.778
clara	1.666	4.199	10.800	1.666	4.200	214	10.566
mc	3.166	8.968	16.143	3.685	9.301	235	16.471
zsh	2.497	7.758	15.485	2.542	7.786	38	15.788
mutt	3.227	11.930	21.97	3.259	11.943	394	22.694
bash	3.076	8.834	22.081	3.115	8.856	25	21.070
fvwm	6.561	16.510	25.799	7.149	17.127	604	28.285
nethack	4.383	23.833	63.486	5.024	24.499	415	60.462

Tabelle 11.4: Unmittelbar Zeiger-relevante Instruktionen

Kapitel 12

Evaluation der Resultate

In diesem Kapitel präsentieren wir die Resultate der Analyse, wie wir sie im Rahmen des im vorigen Kapitel vorgestellten Szenarios erhalten haben. Das Kapitel vergleicht hierzu mit Diagrammen und Tabellen die unterschiedlichen Resultate für sechs verschiedene Präzisionsstufen der kombinierten Analyse: fluss-insensitiv, fluss-sensitiv und fluss-sensitiv mit indirekten starken Aktualisierungen, wobei diese drei Ausprägungen jeweils sowohl feld-insensitiv als auch feld-sensitiv ausgewertet wurden. Die exakten Messwerte zu den Diagrammen finden sich in einem separaten Dokument, das man von der Webseite [Staiger-Stöhr 2009] beziehen kann.

12.1 Zeigeranalyse

Für die Betrachtung der kombinierten Analyse als Zeigeranalyse untersuchen wir sowohl das Mengengerüst als auch typische Metriken für die Präzision (Skalierbarkeit evaluieren wir gemeinsam im nächsten Kapitel). Wir konzentrieren uns auf Datendereferenzierungen, d.h. indirekte Definitionen und Verwendungen (feld-sensitiv auch indirekte Adressnahmen). Die indirekten Aufrufe betrachten wir als Teil der Kontrollfluss-Analyse weiter unten.

Die Tabelle 12.1 zeigt nun zunächst das Mengengerüst, d.h. die Gesamtzahl an erkannten Zielen. Damit man die sechs verschiedenen Präzisionsstufen gut vergleichen kann, haben wir das jeweilige Resultat für alle zusammen

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	81	0	0	103	0	0
units	1.834	-72	0	1.810	-201	-32
bc	5.619	-140	0	4.098	-53	-5
grep	8.043	-3.896	-4	2.057	-166	-8
nano	3.351	-23	0	4.413	-33	0
less	61.674	-13.870	-443	30.340	-4.025	-308
sed	79.633	-58	-220	19.782	-31	-60
flex	5.205	-10	0	5.028	-10	0
tcc	15.369	-895	-722	21.927	-1.227	-688
bison	47.773	-2.279	0	1.126.605	-5.164	0
wget	16.445	-2.070	0	13.086	-3.720	-18
cook	583.861	-36.358	-656	824.597	-74.252	-14.358
tcsh	471.097	-54.008	-5.860	426.259	-50.280	-12.099
clara	12.724	-701	0	9.983	-699	-7
mc	1.578.396	-577.452	-173.213	23.919	-7.878	-152
zsh	845.496	-197.714	-84.578	1.736.631	-992.869	-110.269
mutt	1.498.740	-39.456	-39.657	518.231	-14.544	-29.635
bash	1.943.480	-21.280	-12.706	1.543.880	-18.816	-10.102
fvwm	1.606.582	-36.592	-7.501	1.000.328	-26.313	-25.127
nethack	6.875.127	-245.751	-6.730	1.863.076	-81.695	-290

Tabelle 12.1: Ziele an indirekten Definitionen und Verwendungen

jeweils auf einer Seite aufgeführt. Da die Zahlen um mehrere Größenordnungen differieren, nutzen wir eine Tabelle als Darstellungsform. Diese ist nach der Größe der Programme (gemessen als Zahl der Instruktionen) sortiert und enthält links die feld-insensitiven, rechts die feld-sensitiven Werte. Die FI-Spalten geben den absoluten fluss-insensitiven Wert wieder. Für einen besseren Vergleich der Auswirkungen der Fluss-Sensitivität sind die beiden weiteren Spalten in jedem Tabellenteil als Differenzen zum jeweils links davon befindlichen Wert angegeben: Der fluss-sensitive Wert (FS) als Differenz FS - FI und der Wert mit zusätzlichen indirekten starken Aktualisierungen als Differenz zu FS. Bei Messwerten wie hier bei den Zeigerzielen erkennen wir damit klar die zunehmende Präzision.

Für die Tabellen und Diagramme zur Zahl der Zeigerziele wurden alle Ziele gezählt, die einen Datenfluss-Knoten als Resultat auslösen (vgl. Tabelle 3.1) oder sich auf ein String-Objekt beziehen. Funktionen als offensichtliche Falschpositive sind beispielsweise nicht eingeflossen, und Sonderziele wie Null und Unknown_Objects reichen wir in anderen Tabellen weiter unten nach. Objektgruppen wurden vor der Zählung expandiert, so dass sie in der Betrachtung keine Rolle spielen. Das in den Zahlen erkenntliche Mengengerüst wirkt sich also direkt auf die Größe der Datenfluss-Darstellung aus.

Feld-insensitiv erkennen wir eine deutliche Zunahme der Gesamtzahl an Zeigerzielen bei den größeren Programmen. Das Maximum erreicht nethack mit über sechs Millionen Zielen. Das Mengengerüst für die Analyse ist somit bis clara weitgehend harmlos, darüber aber ist die absolute Größe deutlich spürbar. Auffallend ist auch die deutliche Zunahme vom zweitgrößten zum größten Programm, sowie das kleine Mengengerüst der Programme bis wget. Bezüglich der verschiedenen Präzisionsstufen sehen wir nur wenige Unterschiede. Bei mc, zsh und nethack sind die Verbesserungen gut zu sehen, die durch die Fluss-Sensitivität im Vergleich zur Fluss-Insensitivität erreicht wurden. Bei tcsh, mutt und fvwm fallen sie geringer aus. Die zusätzliche Unterstützung indirekter starker Aktualisierungen ergibt prozentual oft nur kleine Unterschiede. Die großen Programme profitieren aber immer noch sichtbar von der höheren Präzision, mit nur geringen Verbesserungen bei den drei größten Programmen.

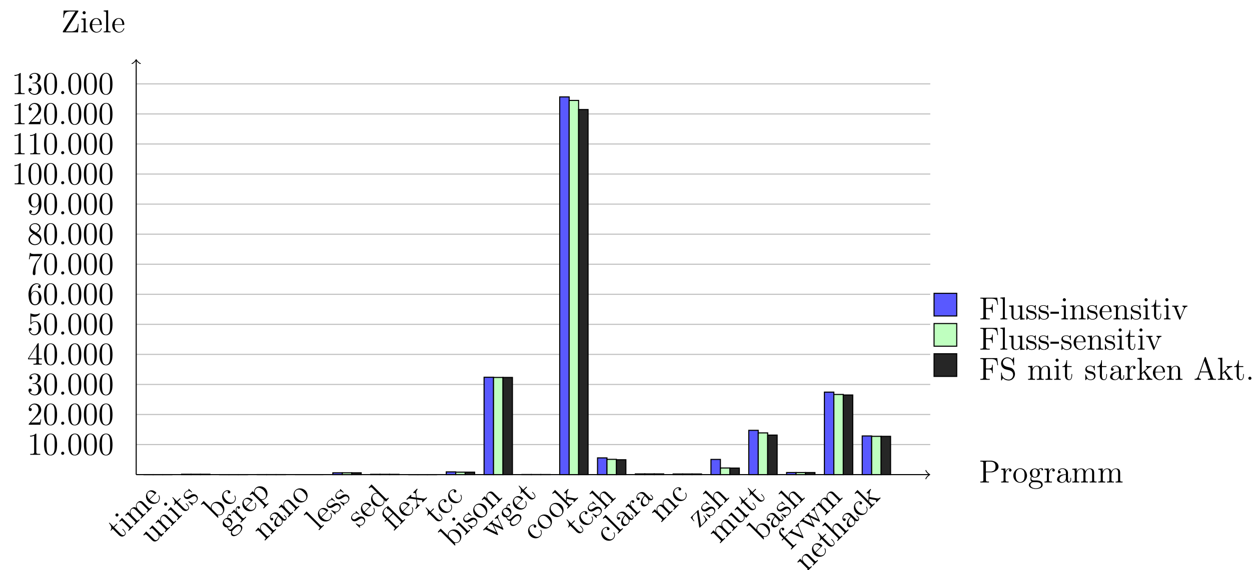


Abbildung 12.1: Ziele an indirekten Adressnahmen

Im Vergleich dazu erkennen wir feld-sensitiv ein deutlich kleineres Mengengerüst bei vielen Programmen. Diese Art der Präzisionssteigerung scheint also einen deutlicheren Effekt auf die Resultate der Zeigeranalyse zu haben. So schrumpft das Mengengerüst z.B. fluss-sensitiv bei mc auf nur noch 1.6% der feld-insensitiven Größe. Hier führt die höhere Präzision unter anderem zur Erkennung von ca. 7% mehr totem Code (die Zahlen zum toten Code präsentieren wir aus Platzgründen nicht vollständig). Neben der allgemeinen Reduktion der Zahl der Ziele verschwinden damit auch alle Ziele, die feld-insensitiv in diesem Bereich gemeldet wurden. Jedoch finden wir auch die umgekehrten Fälle: So ist das fluss-sensitive Mengengerüst für bison über 24 Mal so groß wie feld-insensitiv. Zwei Gründe verursachen dieses Phänomen:

1. Zusätzliche Zeigerziele: Indirekte Adressnahme wie `&(p->x)` erzeugt feld-sensitiv Zeigerziele, die es insensitiv nicht gab. Abbildung 12.1 illustriert die Zahl dieser zusätzlichen Zeigerziele. Darin erkennen wir für bison mehr als 30.000 zusätzliche Zeigerziel-Quellen, die durch solche Instruktionen entstehen. Bei bison übersetzen sie sich zudem in viele neue Ziele (nicht nur neue Quellen für bereits anderswo bekannte Ziele).

2. Wie Tabelle 11.4 und die zugehörige Diskussion gezeigt haben, gibt es in der feld-sensitiven IR mehr Dereferenzierungen als in der insensitiven IR. So erhöht sich diese Zahl z.B. für bison um 11,2% (766 zusätzliche Dereferenzierungen), für cook um 18,5% und für fvwm um 7,8%.

Die unterschiedlichen Präzisionsstufen hinterlassen auch feld-sensitiv eher wenig Spuren. Eine Ausnahme ist der starke Präzisionsgewinn durch Fluss-Sensitivität bei zsh. Dieser kommt unter anderem dadurch zustande, dass fluss-sensitiv zwei indirekte Aufrufziele weniger erkannt werden, was seinerseits zu einer größeren Menge an totem Code führt. Außerdem ist bei zsh der Durchschnitt der Zeigerziele fluss-insensitiv hoch (vgl. unten), d.h. es liegt fluss-insensitiv eine generelle Ungenauigkeit vor. Hier gelang es der Fluss-Sensitivität, für viele Dereferenzierungen die Zielmenge von 267 Zielen auf 132 Ziele quasi zu halbieren, was den größten Beitrag für die im Diagramm sichtbare Diskrepanz zur Fluss-Insensitivität ausmacht.

Auch bei anderen Programmen gewinnen wir gegenüber einer Analyse im Andersen-Stil, jedoch nicht ganz so stark. Die indirekten starken Aktualisierungen verbessern die Resultate dagegen weniger deutlich.

Die Tabelle 12.2 ergänzt die Betrachtung der Zeigerziele, indem sie die Zahl an indirekten Definitionen und Verwendungen angibt, die jeweils das Sonderziel Unknown_Objects erhalten haben. Die Tabelle zeigt uns einen klaren Trend mit zunehmender Programmgröße, wobei es kaum Unterschiede in den verschiedenen Präzisionsstufen gibt. Die klare Zunahme bei zunehmender Größe spricht dafür, dass man sich für eine präzise Analyse großer Programme Gedanken über den Umgang mit Aufrufen an externe Funktionen und deren Effekte machen sollte.

Ein sehr ähnliches Bild erhalten wir für Null als besonderes Zeigerziel. Die Tabelle 12.3 präsentiert hierfür die Zahl der indirekten Definitionen und Verwendungen, die potentiell auf Null zeigen können. Wieder werden die Resultate feld-sensitiv etwas besser, und wieder gelingt der Fluss-Sensitivität nur ein geringer Gewinn gegenüber der Fluss-Insensitivität.

Als weitere typische Metrik zur Präzision betrachtet Tabelle 12.4 die durchschnittliche Zahl an Zeigerzielen pro Dereferenzierung. Da uns für starke Aktualisierungen speziell die indirekten Definitionen interessieren, haben

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	21	0	0	21	0	0
units	426	-17	0	320	-8	-8
bc	964	-30	0	964	-30	-1
grep	813	-16	-2	599	-16	-2
nano	906	-26	0	832	-26	0
less	704	-74	-6	630	-76	-6
sed	1.821	-3	0	1.871	-1	0
flex	1.624	-8	0	1.577	-8	0
tcc	1.320	-25	-70	1.312	-27	-36
bison	2.839	-71	0	3.018	-136	0
wget	1.764	-40	0	1.557	-30	0
cook	3.781	-66	-4	3.926	-60	-67
tcsh	4.028	-118	-30	3.959	-110	-30
clara	4.589	-18	0	4.540	-15	0
mc	8.989	-126	-6	8.294	-63	-5
zsh	7.609	-225	-371	7.599	-223	-375
mutt	8.886	-212	-177	8.008	-250	-73
bash	10.116	-44	-62	10.092	-44	-62
fvwm	13.321	-253	-58	13.260	-297	-292
nethack	22.343	-447	-22	21.195	-477	-3

Tabelle 12.2: Indirekte Definitionen und Verwendungen mit unbekannten Zielen

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	8	0	0	8	0	0
units	346	-20	0	313	-26	-8
bc	854	-28	0	825	-28	0
grep	837	-24	0	463	-87	0
nano	891	-43	0	891	-43	0
less	920	-30	-5	909	-27	-5
sed	1.954	-27	0	1.710	-54	-23
flex	1.277	-133	0	1.221	-134	0
tcc	1.605	-86	-20	1.497	-84	-33
bison	3.272	-130	0	3.339	-140	0
wget	1.621	-143	0	1.555	-148	0
cook	3.717	-220	-5	3.591	-313	-60
tcsh	4.002	-169	-23	3.901	-172	-29
clara	4.224	-160	0	4.182	-159	-19
mc	7.258	-281	-3	6.106	-604	-48
zsh	7.697	-174	-381	7.647	-172	-438
mutt	12.572	-99	-37	11.900	-93	-37
bash	9.922	-107	-61	9.891	-107	-61
fvwm	14.474	-424	-41	13.591	-364	-235
nethack	24.295	-340	-12	23.942	-583	-12

Tabelle 12.3: Indirekte Definitionen und Verwendungen mit Null als Ziel

wir diese Statistik getrennt für indirekte Definitionen und Verwendungen erhoben.

Für die indirekten Definitionen zeigen sich auch bereits bei kleineren Programmen zweistellige Werte und insgesamt recht hohe Absolutwerte für den Durchschnitt. Feld-insensitiv ist erneut eine deutliche Anhebung des Durchschnitts bei größeren Programmen zu sehen, allerdings mit Ausnahmen. Diesmal erkennen wir auch größere Unterschiede zwischen der fluss-insensitiven und fluss-sensitiven Analyse bei einigen Programmen. Im Falle von mc, zsh und mutt ist darüber hinaus auch ein Gewinn durch die indirekten starken Aktualisierungen zu beobachten.

Der feld-sensitive Vergleich zeigt uns passend zum Mengengerüst eine Verbesserung des Durchschnitts in vielen Fällen. Insbesondere unser größtes Programm profitiert davon. Andere Programme wie bison erfahren dagegen eine Verschlechterung, ebenfalls analog zum Mengengerüst. Insgesamt ist hier kein klarer Anstieg mehr mit zunehmender Programmgröße zu verzeichnen. Die Präzision der Analyse ist also bezüglich dieser Metrik über die hier betrachteten Größenunterschiede hinweg nicht korreliert.

Das Bild bei den indirekten Verwendungen ist ähnlich, weswegen wir die Zahlen dazu nicht explizit aufführen.

Für die Präzision und im Hinblick auf die Unterstützung indirekter starker Aktualisierungen interessieren uns besonders die indirekten Definitionen, für die genau ein Ziel erkannt wurde. Dazu zeigen die Abbildungen 12.2 und 12.3 den prozentualen Anteil dieser indirekten Definitionen (ohne diejenigen, die nur auf Unknown_Objects zeigen) an allen indirekten Definitionen. Dieser Anteil ist bei vielen Programmen erfreulich hoch. Die Diagramme legen aber auch nahe, dass der Anteil der auf diese Weise präzise aufgelösten Dereferenzierungen bei den großen Programmen kleiner ausfällt: Dort, wo das Mengengerüst schon umfangreich war, ist die Präzision in diesem Punkt auch nicht so gut wie bei den übrigen Programmen.

Tendenziell schafft wieder die Fluss-Sensitivität eine gewisse Verbesserung gegenüber Andersens Analyse, und die indirekten starken Aktualisierungen erreichen nur kleine weitergehende Präzisionssteigerungen. Bei tcsh und fvwm erleben wir die Überraschung, dass trotz der höheren Präzision

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	1	0	0	1	0	0
units	3,24	-0,34	0	3,10	-0,20	0
bc	5,74	-0,35	0	4,02	-0,21	0
grep	5,88	-2,47	0	1,75	-0,12	-0,01
nano	3,73	-0,29	0	4,33	-0,32	0
less	39,31	-12,31	-0,35	25,22	-7,96	-0,25
sed	17,16	-0,08	0	5,24	-0,04	-0,02
flex	3,81	-0,13	0	3,71	-0,13	0
tcc	5,71	-0,27	-0,07	6,75	-0,35	-0,12
bison	6,35	-0,30	0	113,71	-0,81	0
wget	3,69	-0,18	0	3,92	-0,27	0
cook	81,04	-14,01	-0,09	111,56	-22,81	-0,28
tcsh	54,60	-9,40	-0,25	49,05	-8,70	-0,92
clara	3,25	-0,23	0	2,93	-0,22	-0,01
mc	116,67	-42,04	-12,50	3,55	-0,36	-0,01
zsh	55,60	-19,30	-3,65	118,40	-78,21	-5
mutt	77,91	-5,36	-2,53	27,68	-2,57	-1,35
bash	153,77	-2,49	-0,21	120,22	-2,10	-0,17
fvwm	45,96	-1,33	-0,02	25,80	-0,94	-0,07
nethack	203,96	-17,45	-0,24	51,26	-9,25	0,01

Tabelle 12.4: Durchschnittliche Zahl an Zeigerzielen an indirekten Definitionen

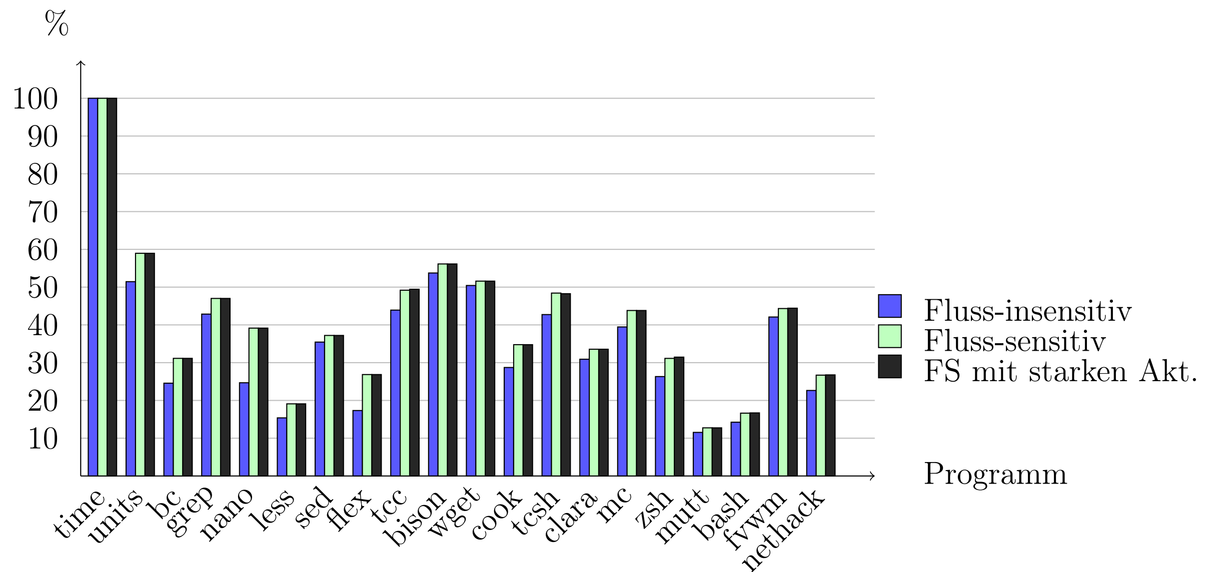


Abbildung 12.2: Anteil der indirekten Definitionen mit genau einem Ziel (feld-insensitiv)

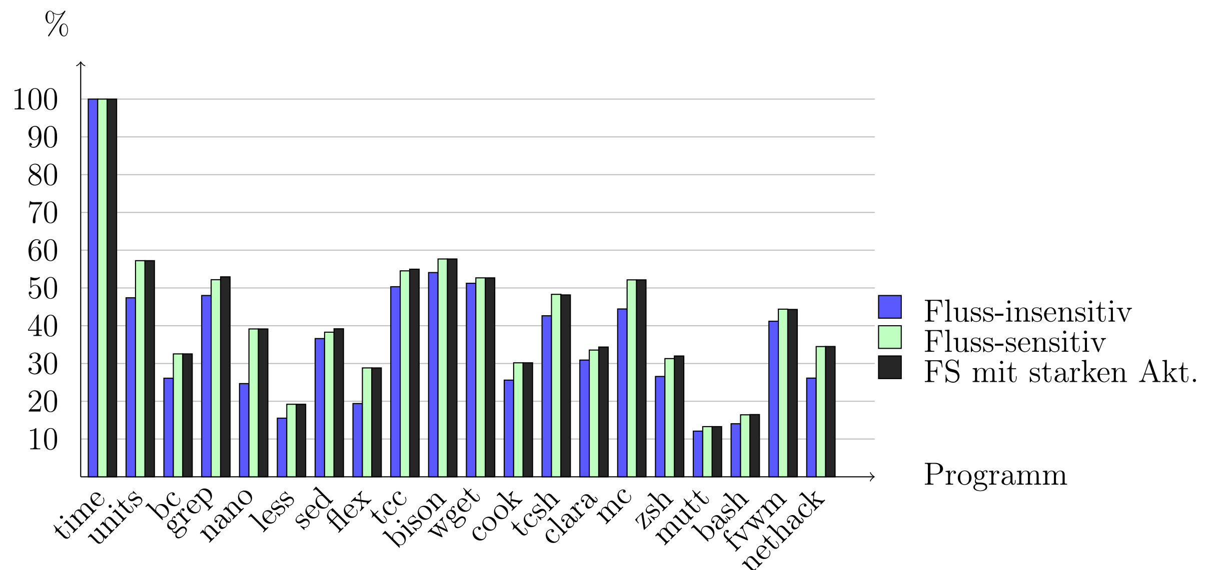


Abbildung 12.3: Anteil der indirekten Definitionen mit genau einem Ziel (feld-sensitiv)

bei indirekten starken Aktualisierungen der Anteil geringer ausfällt als ohne diese Erweiterung. Eine nähere Betrachtung dieser Fälle hat ergeben, dass es sich um indirekte Definitionen handelt, die mit der Erweiterung gar kein Ziel erhalten – die Präzision ist in diesem Sinne also tatsächlich gestiegen, auch wenn das Diagramm zunächst anderes nahelegt. Neben (eher seltenen) echten Programmierfehlern handelt es sich bei solchen Dereferenzierungen ohne Ziel um Folgen der Überschätzung an anderer Stelle: Wird für einen indirekten Aufruf ein eigentlich nicht zutreffendes Ziel erkannt, so kann dies dazu führen, dass der Zielfunktion unpassende Parameter übergeben werden (z.B. Zahlen, wo Zeiger erwartet werden). In der Folge propagiert dann möglicherweise auch kein Ziel an Dereferenzierungen dieser Parameter. In typsicheren Sprachen sollte die Beachtung der Typen in der Analyse derartige Probleme deutlich mildern. Weiterhin verlassen sich einige Programme auf eine implizite Initialisierung von Zeigern mit dem Nullzeiger. Da wir in solchen Fällen statt des Nullzeiger-Literals einen nicht initialisierten Zeiger in der Analyse sehen, erscheinen eigentlich tote Zweige wie

```
-- p nicht initialisiert
if (p != NULL) x = *p;
```

in der Analyse ebenfalls als Dereferenzierungen ohne Ziel.

Feld-sensitiv steigt der Anteil der indirekten Definitionen mit genau einem Ziel erfreulicherweise auch bei den großen Programmen. Hier sehen wir auch öfter als zuvor einen Gewinn durch die Unterstützung indirekter starker Aktualisierungen. Es scheint, als würde diese Erweiterung erst mit grundsätzlich höherer Präzision der Analyse greifen, da ansonsten zwar die Zahl der Zeigerziele gesenkt wird, aber eben noch nicht auf eine einelementige Menge an vielen indirekten Definitionen, die noch dazu eine starke Aktualisierung erlauben.

Um die Situation näher zu beleuchten, haben wir zusätzlich die vollständige Verteilung der Zeigerziele in den Histogrammen 12.4 bis 12.9 exemplarisch für drei der Programme angegeben. Nano als Vertreter der kleinen Programme zeigt uns schon, dass an vielen Dereferenzierungen die Menge der potentiellen Ziele klar begrenzt werden konnte. Neben den gut aufgelösten

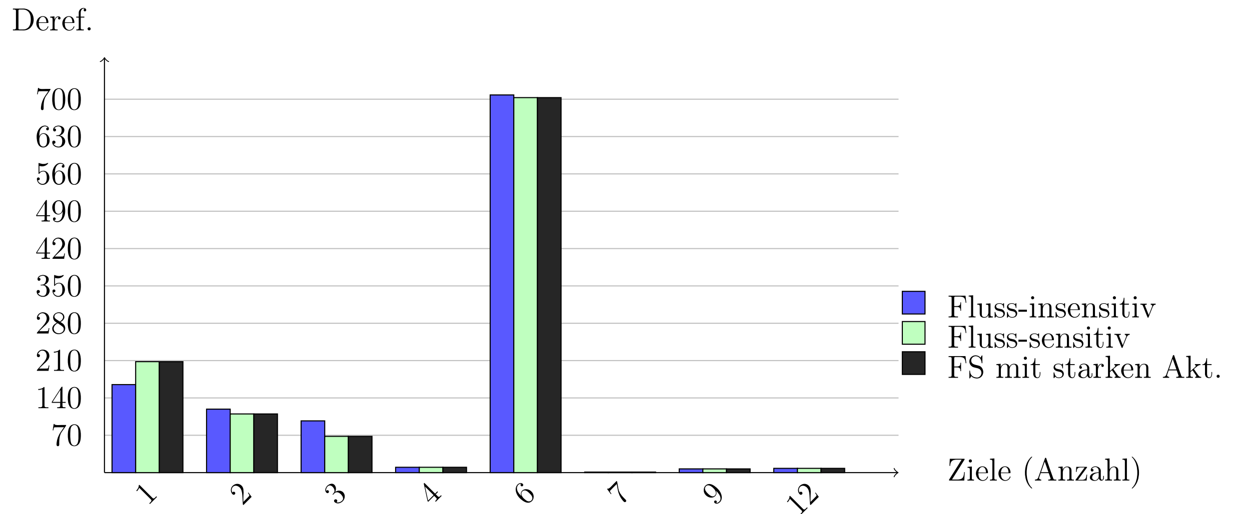


Abbildung 12.4: Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel nano (feld-insensitiv)

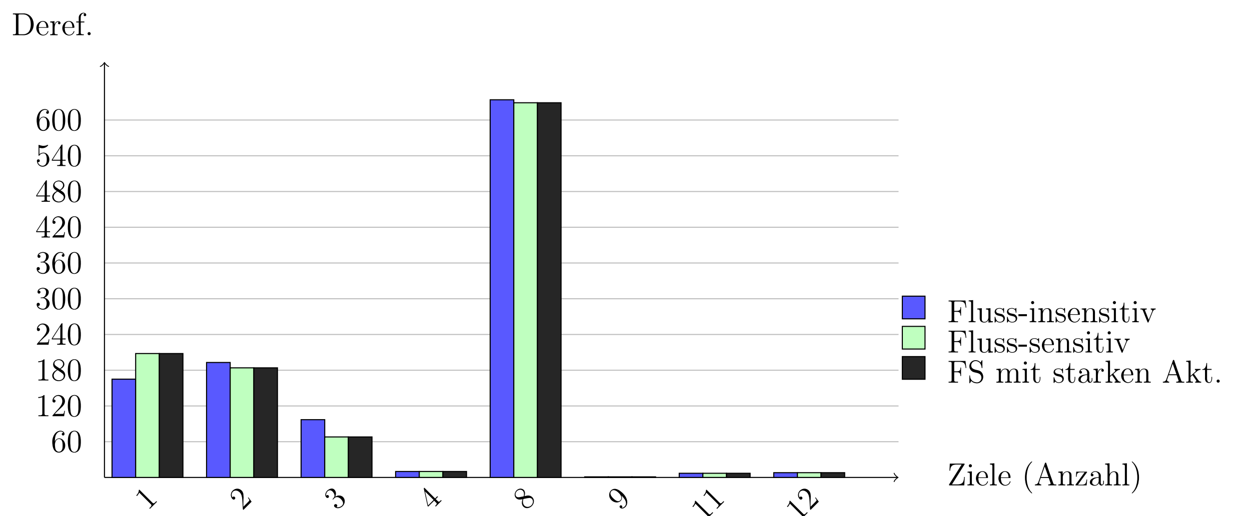


Abbildung 12.5: Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel nano (feld-sensitiv)

Dereferenzierungen sehen wir jedoch auch eine auffällig hohe Ansammlung von Dereferenzierungen mit genau sechs Zielen.

Dieses Phänomen, dass neben einer deutlichen Zahl von gut aufgelösten Dereferenzierungen auch eine zweite Gruppe von Dereferenzierungen mit einer bestimmten höheren Zahl an Zielen auftaucht, konnten wir bei fast allen Programmen beobachten. Die zweite Gruppe wird in der Regel durch die Zusammenlegung verschiedener tatsächlicher Objekte zu einem Analyse-Objekt ausgelöst: So besitzen viele Programme statisch initialisierte Strukturen und Arrays, deren Komponenten jeweils die Adresse verschiedener Objekte zugewiesen wird. Feld-insensitiv existiert jedoch nur ein Objekt für die gesamte Struktur (und für Arrays analog), so dass dieses eine Objekt nun auf alle Ziele zeigt. Dereferenzierungen solcher Objekte führen dann zu einer hohen Zahl an Zielen. Gleichzeitig sind es diese Fälle, in denen unsere Verwendung von Objektgruppen (für die bei der Initialisierung zugewiesenen Ziele) erfolgreich Zeit und Platz bei der Propagierung einsparen kann. Zum Phänomen tragen aber auch Heapobjekte bei, für die neben den genannten Ungenauigkeiten noch das Problem hinzukommt, dass alle Heapobjekte einer Allokationsstelle vereint werden. Damit werden auch deren Zeigerziele vereint.

Im Falle von nano sehen wir, dass die Feld-Sensitivität diese zweite Gruppe von Dereferenzierungen nicht auflösen konnte. Während die Zahl der Dereferenzierungen in dieser Gruppe etwas kleiner geworden ist, hat sich die Zahl der Zeigerziele an jeder dieser Dereferenzierungen von sechs auf acht erhöht. In diesem Fall liegt es daran, dass die feld-sensitive IR für nano mehr direkte Adressnahmen und in der Folge auch mehr aktuelle Zeigerziele aufweist als ihr (in diesem Punkt stärker optimiertes) feld-insensitives Gegenstück. Zahlenmäßig zugenommen hat dagegen die Gruppe mit genau zwei Zielen.

Positiv fällt auf, dass die Fluss-Sensitivität sowohl mit als auch ohne eine Unterscheidung der Felder die Zahl der Dereferenzierungen erhöhen konnte, welche genau ein Ziel haben. Dabei handelt es sich genau um 43 Dereferenzierungen, welche fluss-insensitiv zwei, drei oder gar sechs bzw. acht Ziele erhalten haben.

Als zweites Beispielprogramm haben wir bison herausgepickt: Es liegt bezüglich der Größe ungefähr in der Mitte aller betrachteten Programme und

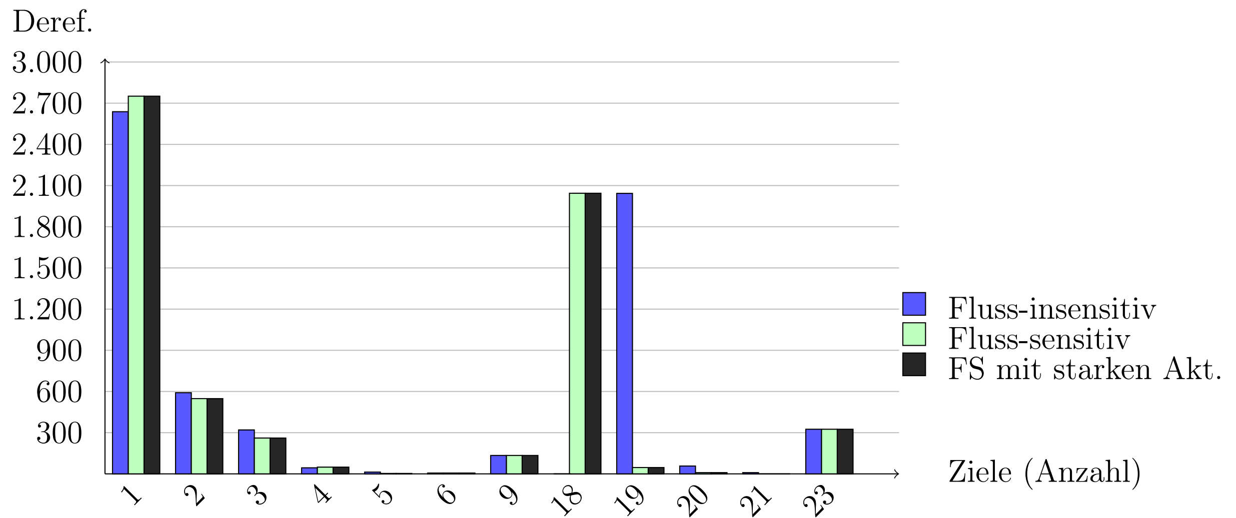


Abbildung 12.6: Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel bison (feld-insensitiv)

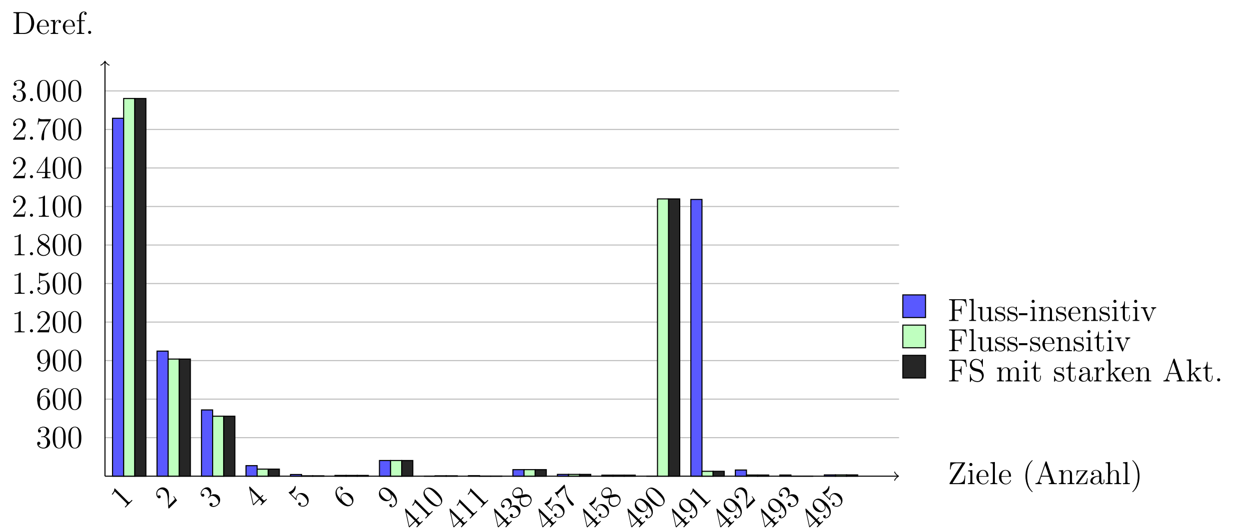


Abbildung 12.7: Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel bison (feld-sensitiv)

zeigte sowohl bei der durchschnittlichen als auch bei der Gesamtzahl der Ziele feld-sensitiv einen hohen Wert im Vergleich zu ähnlich großen Programmen. Wir erkennen in den Histogrammen 12.6 und 12.7 erneut das für nano beschriebene Phänomen, diesmal jedoch mit einer höheren Zahl an Zielen für die zweite Gruppe. Feld-sensitiv steigt diese Zahl noch einmal, was auch den schon besprochenen Zuwachs im Mengengerüst erklärt. Hier schlägt ebenfalls die oben schon erwähnte Problematik zu, dass die indirekte Adressnahme neue Zeigerziele generiert. In bison finden wir 101 solcher Anweisungen der Form `&(p->x)`, und jede davon kann – abhängig von der Zahl der für p erkannten Ziele – mehrere Ziele hervorbringen. Positiv ist aber festzustellen, dass die Zahl der Dereferenzierungen mit genau einem Ziel feld-sensitiv ebenfalls deutlich zugenommen hat.

Die Fluss-Sensitivität kann die Zahl der Ziele bei der zweiten Gruppe um eines erniedrigen und die Zahl der Dereferenzierungen mit genau einem Ziel erhöhen. Die zweite Gruppe spaltet sich also nach den Präzisionsstufen auf. Die indirekten starken Aktualisierungen haben dagegen bei diesem Programm keine Auswirkungen.

Schließlich haben wir auch die Verteilung für eines unser größten Testprogramme, nämlich fvwm, in den Abbildungen 12.8 und 12.9 dargestellt. Die Verteilung geht hier deutlich mehr in die Breite als bei den kleineren Beispielen, zeigt aber auch wieder das schon diskutierte Phänomen. Zur besseren Übersicht sind nicht alle X-Werte eingezeichnet.

Sowohl feld-insensitiv als auch feld-sensitiv sehen wir diesmal ähnlich zu bison einen klaren Gewinn durch die Fluss-Sensitivität. Zum Einen konnte damit für die zweite Gruppe von Dereferenzierungen mit einer hohen Zahl an Zielen diese Zahl an Zielen reduziert werden. Zum Anderen gibt es zugleich auch mehr Dereferenzierungen mit genau einem Ziel. Feld-sensitiv ist die Streuung insgesamt breiter. Dort erleben wir bei der Gruppe mit genau einem Ziel wieder das Phänomen, dass mit der Unterstützung indirekter starker Aktualisierungen einige Dereferenzierungen kein Ziel mehr erhalten und die Zahl bei genau einem Ziel daher sinkt. Feld-insensitiv dagegen beobachten wir einen leichten Anstieg in der Gruppe mit genau einem Ziel, wenn wir indirekte starke Aktualisierungen unterstützen.

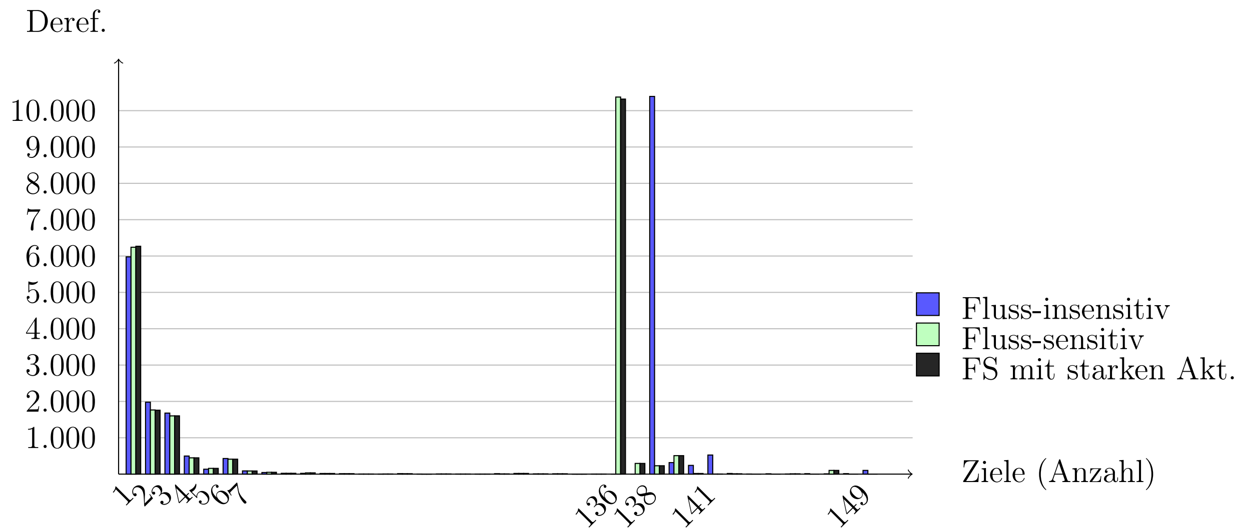


Abbildung 12.8: Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel fvwm (feld-insensitiv)

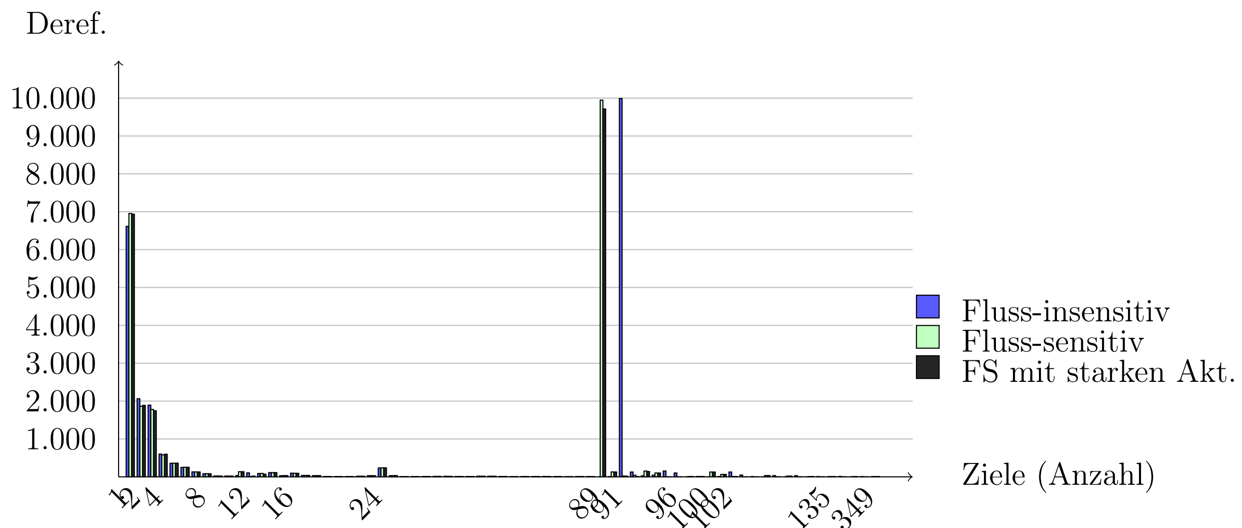


Abbildung 12.9: Histogramm zu den Zeigerzielen an indirekten Definitionen und Verwendungen am Beispiel fvwm (feld-sensitiv)

12.2 Kontrollfluss-Analyse

Im Rahmen der Kontrollfluss-Analyse betrachten wir die Auflösung indirekter Aufrufe. Hierfür präsentieren wir die gleichen Metriken wie für die Datendereferenzierungen. Die folgenden Tabellen berücksichtigen immer nur die Ziele, die tatsächlich Funktionen sind. Die Tabelle 12.5 beginnt die Betrachtungen mit dem Mengengerüst, d.h. der Gesamtzahl an Zielen für indirekte Aufrufe. Die beiden Programme time und tcc besitzen keinen lebendigen indirekten Aufruf mit einer Funktion als Ziel (es gibt jedoch welche mit dem Sonderziel Unknown_Objects, vgl. unten).

Wieder zeigt sich ein deutlicher Unterschied zwischen der Hälfte der kleineren und der größeren Programme. Diesmal taucht jedoch bison als deutlicher Ausreißer bei den kleineren Programmen auf: Mit fast fünftausend Zielen tanzt dieses Programm im Vergleich aus der Reihe. Eine nähere Betrachtung des Codes zeigt, dass bison allein für die Darstellung von Mengen von Bits vier verschiedene Implementierungen besitzt. Zu jeder Implementierung gehören einige Funktionen, und diese werden allesamt indirekt genutzt. Bei den größeren Programmen haben wir ähnlich das Programm zsh als einen Ausreißer mit sehr vielen Zielen für indirekte Aufrufe.

Die Auswirkungen der Fluss-Sensitivität sind bei den indirekten Aufrufen deutlich geringer als bei den Datendereferenzierungen. Lediglich zsh und mutt profitieren (feld-insensitiv auch mc), wobei zsh feld-sensitiv noch einmal durch indirekte starke Aktualisierungen gewinnt. Die Feld-Sensitivität kann die Zahl der Aufrufziele dagegen zum Teil deutlich reduzieren. Besonders stark gewinnt hier mc, aber auch z.B. zsh, fvwm und nethack erhalten spürbar weniger Aufrufziele. Es scheint, als würde die Feld-Sensitivität insbesondere bei größeren Programmen einen lohnenswerten Präzisionsgewinn in Bezug auf die indirekten Aufrufe darstellen, während der Unterschied bei kleineren Programmen nicht so deutlich ist.

Die Tabelle 12.6 präsentiert zusätzlich die Zahl der indirekten Aufrufe, für die noch weitere, unbekannte Ziele vorliegen könnten (d.h. das Ziel Unknown_Objects). Auch hier weist zsh wieder die größte Zahl auf, und andere Programme mit einer hohen Zahl an ermittelten Aufrufzielen haben ebenfalls

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	0	0	0	0	0	0
units	16	0	0	16	0	0
bc	42	0	0	42	0	0
grep	32	0	0	32	0	0
nano	62	0	0	62	0	0
less	62	0	0	62	0	0
sed	2	0	0	2	0	0
flex	2	0	0	2	0	0
tcc	0	0	0	0	0	0
bison	4.861	0	0	4.128	0	0
wget	75	0	0	68	0	0
cook	3.961	0	0	3.723	0	0
tcsh	316	0	0	316	0	0
clara	51	0	0	51	0	0
mc	5.873	-56	0	326	0	0
zsh	19.534	-6	0	15.571	-2	-52
mutt	2.010	-83	0	1.793	-83	0
bash	1.328	0	0	971	0	0
fvwm	4.436	0	0	747	0	0
nethack	7.072	0	0	1.273	0	0

Tabelle 12.5: Zeigerziele an indirekten Aufrufen

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	0	0	0	0	0	0
units	1	0	0	1	0	0
bc	0	0	0	0	0	0
grep	0	0	0	0	0	0
nano	2	0	0	0	0	0
less	0	0	0	0	0	0
sed	1	0	0	0	0	0
flex	1	0	0	1	0	0
tcc	3	0	0	3	0	0
bison	146	0	0	134	-7	0
wget	10	0	0	10	0	0
cook	60	0	0	60	0	0
tcsh	4	0	0	2	0	0
clara	1	0	0	1	0	0
mc	128	-1	0	54	-1	0
zsh	379	-1	0	266	-1	0
mutt	56	0	0	56	0	0
bash	17	0	0	15	0	0
fvwm	71	-1	0	56	-1	0
nethack	0	0	0	0	0	0

Tabelle 12.6: Indirekte Aufrufe mit unbekanntem Ziel

eine nicht zu vernachlässigende Zahl an Aufrufen mit potentiell unbekannten Zielen. Das größte Programm nethack dagegen konnte ohne eine solche Unsicherheit aufgrund von Aufrufen an externe Funktionen analysiert werden.

In der Tabelle 12.7 sehen wir die durchschnittliche Zahl an erkannten Funktionen für indirekte Aufrufe. Dieser Durchschnitt fällt für die Programme sehr unterschiedlich aus. Die großen Programme neigen wieder zu einem hohen Durchschnitt, jedoch ist die Präzision für mutt und nethack überraschend gut. Auf der anderen Seite zeigt nano als kleines Programm einen schlechten Durchschnitt. Hier hilft wiederum die genauere Betrachtung des Histogrammes. Für nano ist dieses besonders einfach, denn für beide indirekten Aufrufe des Programmes werden jeweils 31 Ziele erkannt. Da fluss-sensitiv Aufrufe ohne Ziel bleiben (und nicht gezählt wurden), steigt der Durchschnitt im Falle von mc und zsh. Dagegen können unsere beiden größten Testprogramme sehr gut von der Feld-Sensitivität profitieren. Das bestärkt die Aussage von vorhin, dass diese Art der Präzisionssteigerung offensichtlich insbesondere bei großen Programmen zum Tragen kommt.

Für die Präzision zeigen wir in der Tabelle 12.8 außerdem den prozentualen Anteil der indirekten Aufrufe, welche wir mit den Analyse-Resultaten durch genau ein Ziel ersetzen können. Bei den größeren Programmen sehen wir einen höheren Anteil. Z.B. bei bash können feld-insensitiv erfreulicherweise mehr als 30% aufgelöst werden. Auffällig viele kleinere Programme verwenden jedoch ausschließlich indirekte Aufrufe, für die mehr als ein Ziel festgestellt wird. Die Feld-Sensitivität erreicht auch in dieser Statistik einen großen Sprung für nethack, welches damit bei über 90% der indirekten Aufrufe genau ein Ziel erhält.

In den Histogrammen 12.10 und 12.11 für bison erkennen wir deutlich die Spuren des schon skizzierten Codeaufbaus mit den indirekten Aufrufen an die verschiedenen Implementierungen von Bitmengen. Es zeigen sich hier mehrere Peaks bei vergleichsweise hohen Zahlen an Zielen. Die Feld-Sensitivität erzeugt eine Gruppe von Aufrufen mit der niedrigen Zahl von vier Zielen. Dafür ist die Gruppe mit den meisten Zielen in der Anzahl der darin subsumierten indirekten Aufrufe und in der Zahl der Aufrufziele geschrumpft, so dass sich eine klare Präzisionsverbesserung ergibt.

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	0	0	0	0	0	0
units	5,33	0	0	5,33	0	0
bc	2	0	0	2	0	0
grep	4	0	0	4	0	0
nano	31	0	0	31	0	0
less	12,40	0	0	12,40	0	0
sed	2	0	0	2	0	0
flex	2	0	0	2	0	0
tcc	0	0	0	0	0	0
bison	35,48	0	0	30,13	0	0
wget	3,95	0	0	3,78	0	0
cook	55,01	0	0	51,71	0	0
tcsh	12,15	0	0	12,15	0	0
clara	4,64	0	0	4,64	0	0
mc	43,50	0,23	0	3,51	0	0
zsh	46,62	0,10	0	37,25	0,08	-0,04
mutt	3,07	-0,09	0	2,74	-0,09	0
bash	17,25	0	0	12,61	0	0
fvwm	63,37	0	0	21,34	0	0
nethack	8,35	0	0	1,50	0	0

Tabelle 12.7: Durchschnittliche Zahl an Zielen an indirekten Aufrufen

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	0	0	0	0	0	0
units	0	0	0	0	0	0
bc	0	0	0	0	0	0
grep	10	0	0	10	0	0
nano	0	0	0	0	0	0
less	0	0	0	0	0	0
sed	0	0	0	0	0	0
flex	0	0	0	0	0	0
tcc	0	0	0	0	0	0
bison	0,67	0	0	1,34	0	0
wget	8,70	0	0	8,70	0	0
cook	5,56	0	0	5,56	0	0
tcsh	3,70	0	0	3,70	0	0
clara	0	0	0	0	0	0
mc	7,47	0	0	9,80	0	0
zsh	2,79	0	0	2,79	0	0
mutt	0,46	0,31	0	0,46	0,31	0
bash	40	0	0	41,18	0	0
fvwm	5,81	0	0	8,14	0	0
nethack	11,92	0	0	93,39	0	0

Tabelle 12.8: Anteil der indirekten Aufrufe mit genau einem Ziel (%)

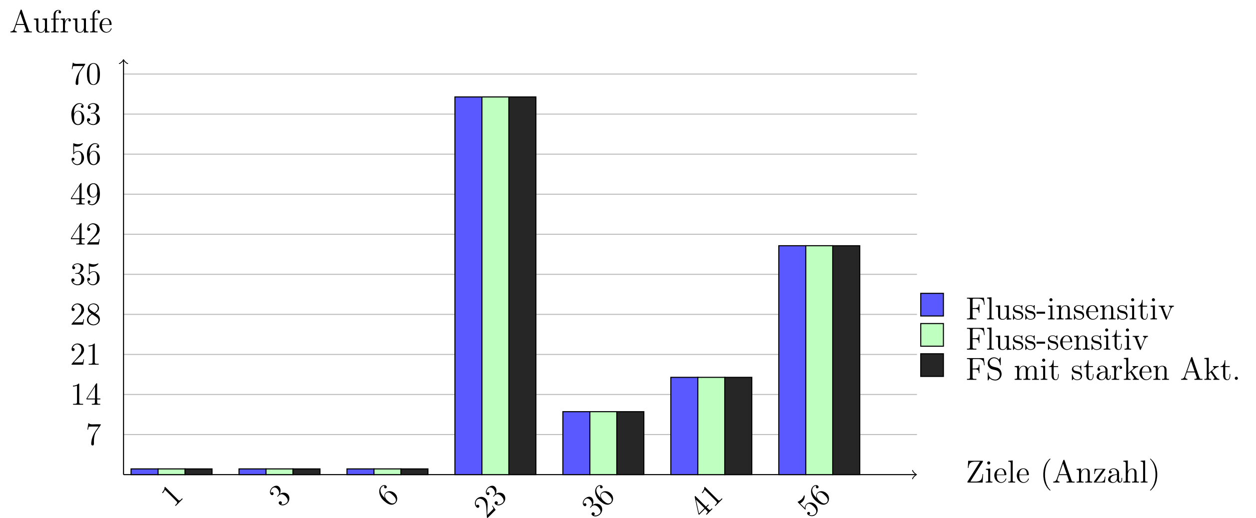


Abbildung 12.10: Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel bison (feld-insensitiv)

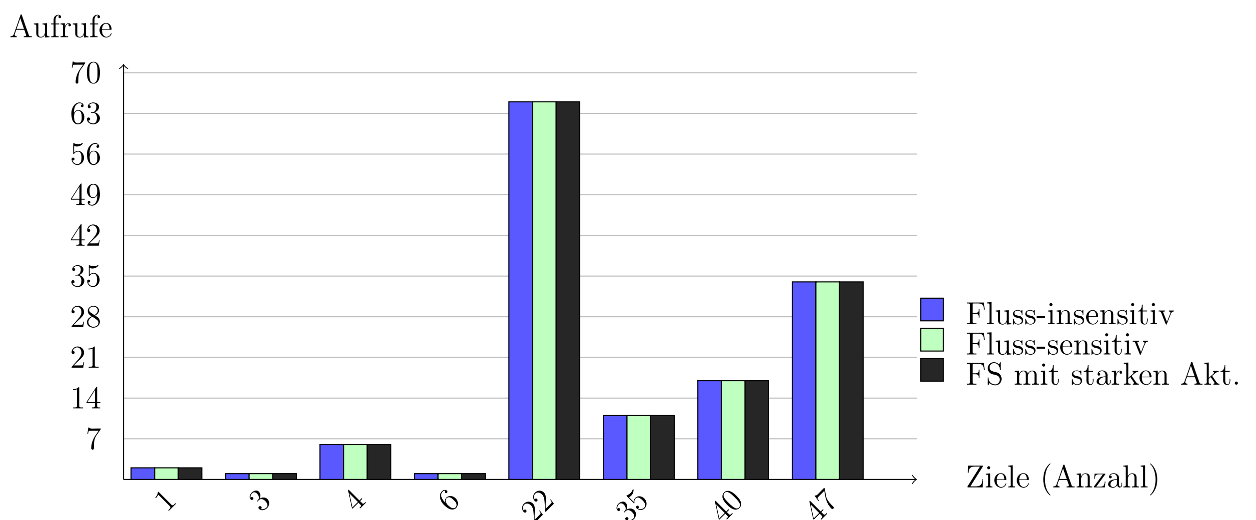


Abbildung 12.11: Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel bison (feld-sensitiv)

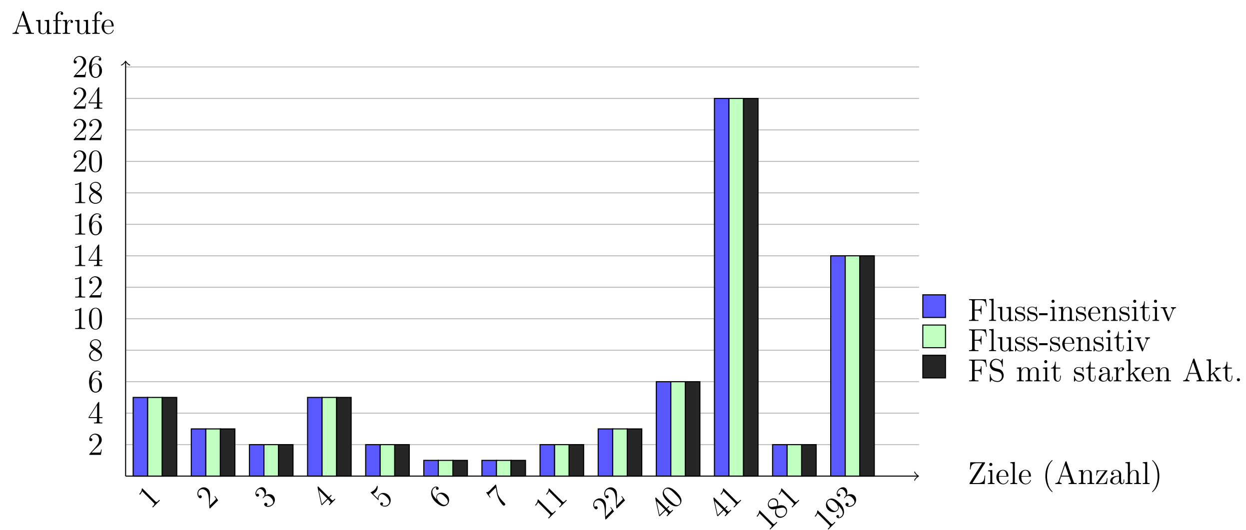


Abbildung 12.12: Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel fvwm (feld-insensitiv)

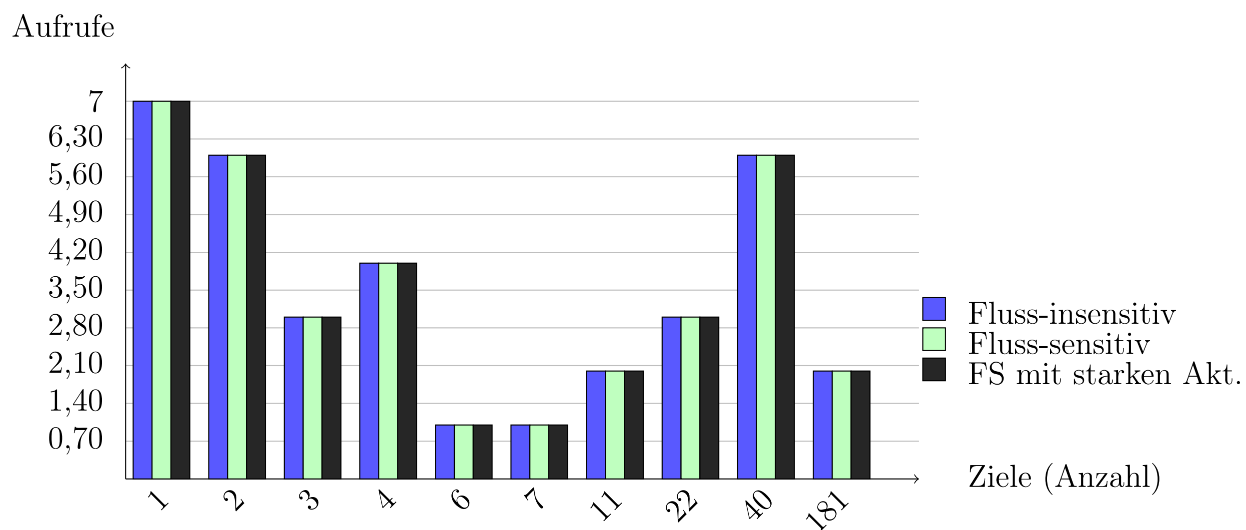


Abbildung 12.13: Histogramm zu den Zeigerzielen an indirekten Aufrufen am Beispiel fvwm (feld-sensitiv)

Für fvwm zeigen uns die Abbildungen 12.12 und 12.13 eine breitere Verteilung mit einer deutlich höheren Zahl an Aufrufen in der letzten Gruppe. Feld-sensitiv verschwinden einige Ziele, insbesondere auch die Gruppe mit 193 Zielen. Tatsächlich gibt es dafür einige Aufrufe, die nur noch das Ziel Unknown_Objects und/oder den Nullzeiger erhalten. Diese Sonderziele wurden für die Histogramme nicht mitgezählt.

12.3 Datenfluss-Analyse

Wenden wir uns nun den Resultaten zu, die wir dem Datenfluss-Part der kombinierten Analyse zurechnen können. Hier interessiert uns Größe und Gestalt des berechneten ISSA-Graphen. Die folgenden Diagramme besprechen zunächst den finalen ISSA-Graphen nach dem Pruning (d.h. das Endresultat im Datenfluss-Bereich), bevor sich ein Unterabschnitt der Frage zuwendet, welchen Effekt das Pruning und die Verwendung des lediglich aus allgemeinen Definitionen bestehenden reduzierten ISSA-Graphen haben, wie ihn unsere Analyse während der Fixpunkt-Iteration bestimmt.

Die Tabelle 12.9 illustriert zunächst den Trend für die Zahl der Knoten im finalen ISSA-Graphen. Der Verlauf der Werte korreliert zumindest fluss-insensitiv mit dem für die Zahl an Zeigerzielen, die für das jeweilige Programm erkannt wurden (vgl. Tabelle 12.1). Wir können daher davon ausgehen, dass die Zahl der Zeigerziele und somit die Präzision des Zeigeranalysen-Parts einen direkten Einfluss auf die Größe des ISSA-Graphen hat. In absoluten Zahlen gesprochen ist das Mengengerüst für die Analyse teilweise enorm, bei den größten Programmen müssen wir feld-insensitiv mit mehr als zehn Millionen Knoten zurecht kommen, feld-sensitiv beim größten Programm sogar mit mehr als zweihundert Millionen Knoten. Das beeinflusst die Skalierbarkeit (vgl. nächstes Kapitel).

Bei der Betrachtung der Tabelle stellen wir jedoch einen wesentlichen Unterschied zu derjenigen für die Zeigerziele fest, was den Effekt der Unterstützung indirekter starker Aktualisierungen angeht. Die Zahl der Knoten kann durch diese Erweiterung zum Teil deutlich gesenkt werden, während wir bei den Zeigerzielen kaum eine Auswirkung gesehen hatten.

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	1.302	0	0	1.580	0	0
units	20.960	-67	-1.208	32.068	-176	-2.588
bc	57.922	-213	-77	73.618	-56	-100
grep	44.283	-10.567	-26	37.487	-437	-334
nano	106.067	-38	-10.613	144.909	-53	-13.533
less	678.579	-24.092	-16.905	737.426	-13.299	-19.920
sed	221.484	-154	-443	222.753	-268	-1.138
flex	64.587	-116	-38	78.307	-149	-59
tcc	385.150	-1.373	-130.971	841.908	-3.999	-268.792
bison	214.625	-2.904	-627	3.098.213	20.161	-4.025
wget	183.481	-2.131	-316	206.220	-10.955	-523
cook	6.281.101	-123.440	-394.841	14.869.193	-41.149	-1.110.458
tcsh	6.875.640	-546.077	-1.352.926	11.505.884	-1.056.127	-3.025.548
clara	329.204	-532	-33	393.366	-479	-274
mc	9.827.751	-1.834.983	-1.444.910	9.624.798	-820.365	-1.111.032
zsh	12.564.629	-1.206.245	-3.586.048	20.755.220	-2.761.692	-5.517.479
mutt	9.178.389	-330.821	-549.547	13.250.539	-415.803	-1.094.019
bash	21.490.200	-347.340	-3.175.486	27.070.828	-498.288	-4.248.340
fvwm	11.915.453	-528.308	-3.610.220	47.012.840	-2.957.304	-19.537.996
nethack	62.526.736	-2.126.944	-15.639.784	209.499.664	-3.886.944	-25.719.392

Tabelle 12.9: Knoten im finalen ISSA-Graphen

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	1.352	-68	0	1.472	-29	-5
units	25.804	-1.379	-1.791	40.778	-2.305	-3.952
bc	71.925	-4.079	-2.575	89.756	-3.344	-2.598
grep	58.703	-16.342	-37	49.605	-1.757	-507
nano	107.794	-5.987	-14.980	205.883	-6.947	-66.768
less	701.715	-49.833	-27.351	798.885	-34.310	-62.048
sed	260.563	-14.666	-1.074	278.213	-5.376	-3.110
flex	110.174	-2.027	-37	132.629	-1.748	-58
tcc	483.849	-77.896	-162.018	1.011.469	-140.879	-348.647
bison	233.240	-19.017	-1.139	3.291.872	-248.616	-4.294
wget	196.784	-5.939	-410	231.020	-14.498	-1.200
cook	6.006.302	-940.338	-481.163	14.658.039	-2.414.271	-1.499.586
tcsh	7.951.563	-1.665.805	-1.699.591	15.301.799	-3.855.519	-4.667.627
clara	406.959	-5.574	-80	473.824	-5.004	-289
mc	11.192.172	-3.064.039	-1.639.573	13.351.668	-1.944.183	-2.086.699
zsh	17.191.866	-4.049.549	-4.806.786	31.735.020	-7.647.088	-10.664.076
mutt	11.722.344	-1.765.439	-941.737	19.015.864	-3.227.656	-2.827.847
bash	23.717.788	-2.482.322	-4.318.274	34.151.840	-3.165.758	-8.821.998
fvwm	16.860.648	-3.529.545	-4.950.534	89.935.760	-15.857.464	-45.984.208
nethack	73.415.584	-10.261.968	-19.077.032	232.145.648	-20.338.384	-40.036.128

Tabelle 12.10: Kanten im finalen ISSA-Graphen

Wir sehen dafür die folgenden Ursachen:

- Indirekte starke Aktualisierungen werden wohl oft für Zeigerziele erkannt, die ihrerseits nicht selbst wieder Zeiger sind. Dadurch zeigt sich der Effekt nicht in der Zahl der Zeigerziele, wohl aber in der Zahl der Knoten und Kanten.

- Indirekte starke Aktualisierungen reduzieren zunächst einmal die Zahl der Kanten, da keine Verbindung von einer vorausgehenden Definition eingezogen wird. In der Folge werden aber auch z.B. weniger Eingänge angelegt, wie es ansonsten bei der Suche nach der vorausgehenden (aber nun dank einer starken Aktualisierung nicht benötigten) Definition geschieht.

- Die Reduktion der Kantenzahl wirkt sich durch das Pruning noch einmal auf die Knotenzahl aus, da das Pruning nun mehr Knoten entfernen kann.

Zusätzlich zu den Knoten betrachten wir mit der Tabelle 12.10 noch die Zahl der *intra*prozeduralen Kanten. Die interprozeduralen Kanten speichert unsere Implementierung nur implizit, sie sind daher für den Platzbedarf irrelevant (nicht jedoch für den Zeitbedarf der Propagierung). Die Verläufe zu Knoten und Kanten ähneln sich sehr deutlich. Das bedeutet, dass der Quotient m/n aus Kanten und Knoten nahezu konstant ist über die hier betrachteten Größenordnungen von Programmen. Damit haben wir auch bei unserer interprozeduralen Analyse lokal jeweils diese gute SSA-Eigenschaft, welche wir auch in der theoretischen Abschätzung genutzt haben (als $|Kanten| = |Knoten| + |\phi| * const$). Außerdem bestätigt dies auch unsere oben angeführte Begründung für den deutlichen Effekt der indirekten starken Aktualisierungen: Die Zahl der Kanten sinkt, und daraufhin auch die Zahl der Knoten.

Als nächstes wollen wir die Zusammensetzung des ISSA-Graphen näher beleuchten. Dafür präsentieren die Abbildungen 12.14 bis 12.17 den Anteil der Definitionen und Verwendungen an den Knoten. Dieser Anteil nimmt mit größer werdenden Programmen klar ab, d.h. je größer ein Programm,

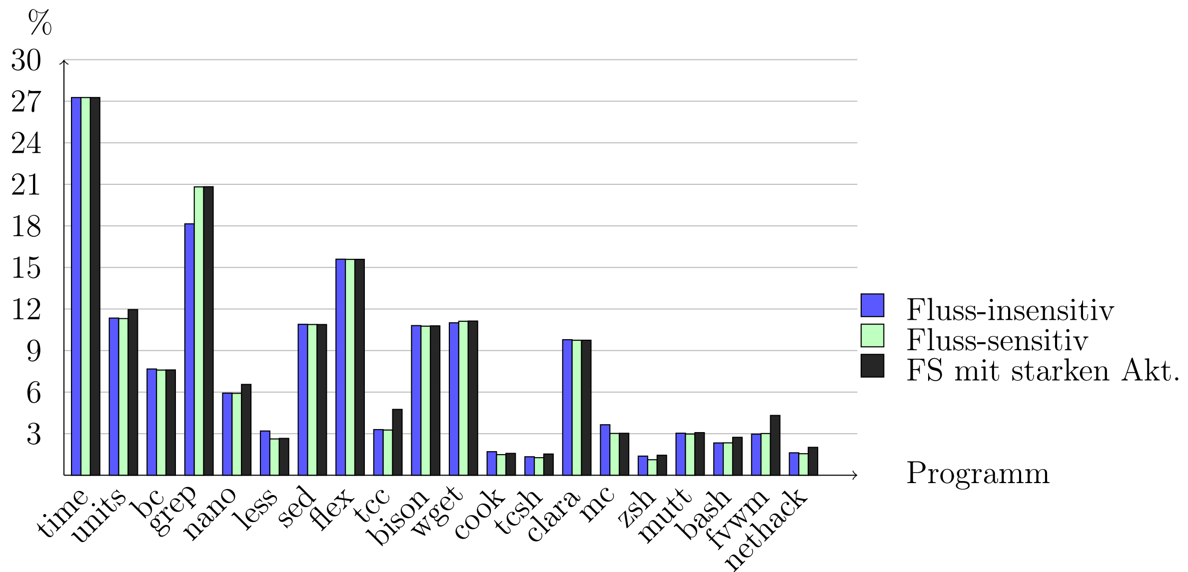


Abbildung 12.14: Anteil der Definitionen am ISSA-Graphen (feld-insensitiv)

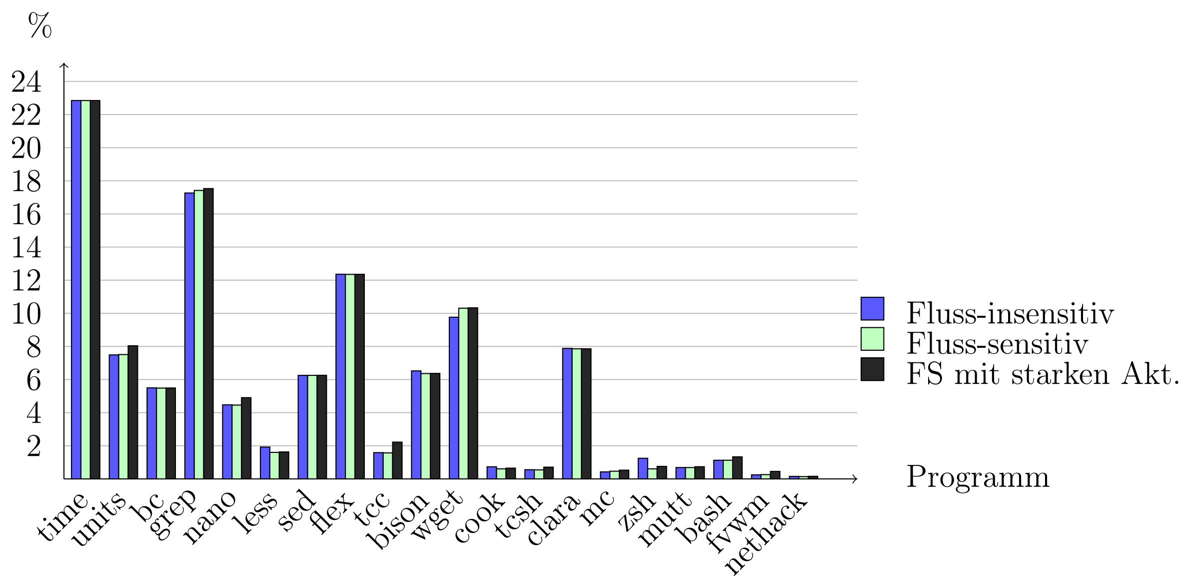


Abbildung 12.15: Anteil der Definitionen am ISSA-Graphen (feld-sensitiv)

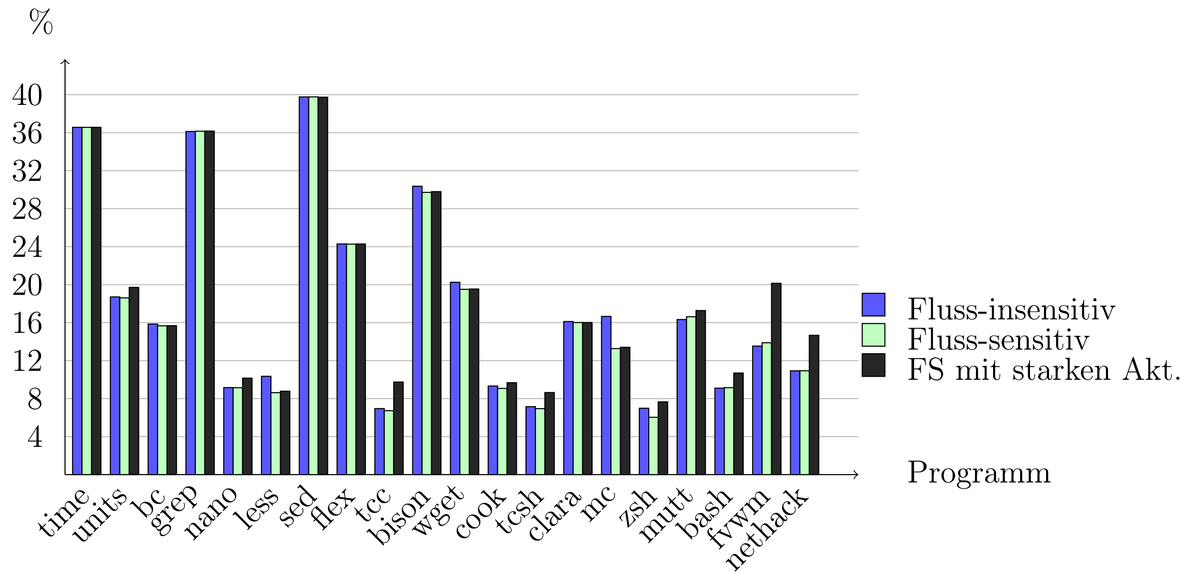


Abbildung 12.16: Anteil der Verwendungen am ISSA-Graphen (feld-insensitiv)

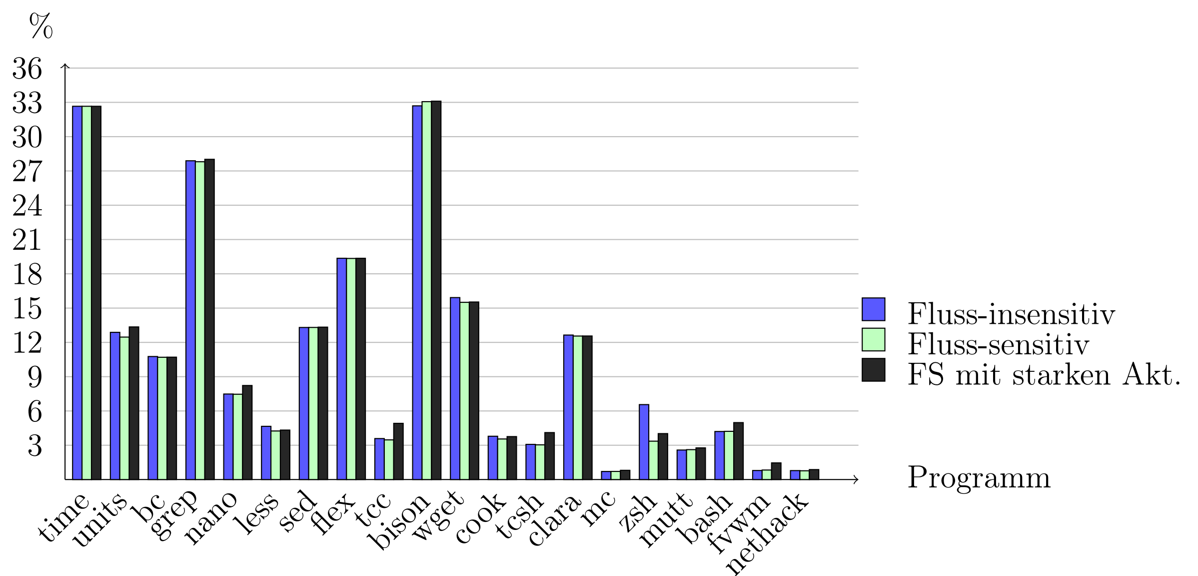


Abbildung 12.17: Anteil der Verwendungen am ISSA-Graphen (feld-sensitiv)

desto stärker bestimmen die ISSA-Artefakte (ϕ-Knoten und Knoten für interprozeduralen Datenfluss) die Gestalt des Graphen. Die Abnahme ist bei den Definitionen deutlicher als bei den Verwendungen. Der Anteil der Definitionen wird fast schon verschwindend gering, so dass man für weitere Verbesserungen der Analyse darüber nachdenken sollte, andere Darstellungen als ISSA zu verwenden, oder speziell für ISSA-Artefakte Optimierungen zu suchen. Feld-sensitiv sind die Anteile nochmals geringer als feld-insensitiv, da nun für die Felder einzeln die überproportional vielen ISSA-Artefakte angelegt werden. Dadurch, dass die Gesamtzahl an Knoten im ISSA-Graphen variiert, bei schlechterer Präzision zunächst mehr Definitionen und Verwendungen auftauchen, und das Pruning damit auch keinen klaren Trend erzeugt, ergibt sich bei den Anteilen kein einheitlicher Trend für die drei Abstufungen hinsichtlich der Fluss-Sensitivität. Tendenziell, aber nicht immer, ist der Anteil der Definitionen und Verwendungen bei höherer Präzision ebenfalls höher. Die Verwendungen stellen einen größeren Anteil als die Definitionen, was wir auch als einen Teil der Rechtfertigung der Idee sehen, während der Fixpunkt-Iteration ohne diese Knoten auszukommen.

Um den Anteil der interprozeduralen Knoten und damit der Ergänzung von ISSA im Vergleich zu SSA besser zu erfassen, zeigen uns die Abbildungen 12.18 und 12.19 den Anteil der ϕ-Knoten an allen Knoten. Der nun noch fehlende Anteil geht damit auf das Konto der interprozeduralen Knoten. Wir sehen im Diagramm, dass die ϕ-Knoten einen stattlichen Anteil ausmachen, mit keinem klaren Trend bezüglich der Programmgröße. Lediglich ein leichter Anstieg mit zunehmender Programmgröße lässt sich feststellen. Feld-sensitiv ist die Situation sehr ähnlich, sowohl bezüglich des Verlaufs als auch der Höhe der Anteile.

Relativ wenig ϕ-Knoten tauchen z.B. im ISSA-Graphen für cook auf. Tatsächlich stellen Definitionen, Verwendungen und ϕ-Knoten in diesem Programm zusammen nur etwas weniger als 25% aller Knoten. Der große Rest von drei Viertel aller Knoten dient zur Repräsentation des interprozeduralen Datenflusses. Dies lässt darauf schließen, dass in cook die Funktionen vielfach über globale Variablen oder Heapobjekte miteinander kommunizieren. Tatsächlich verwendet das Programm eine Symboltabelle, über die indirekte

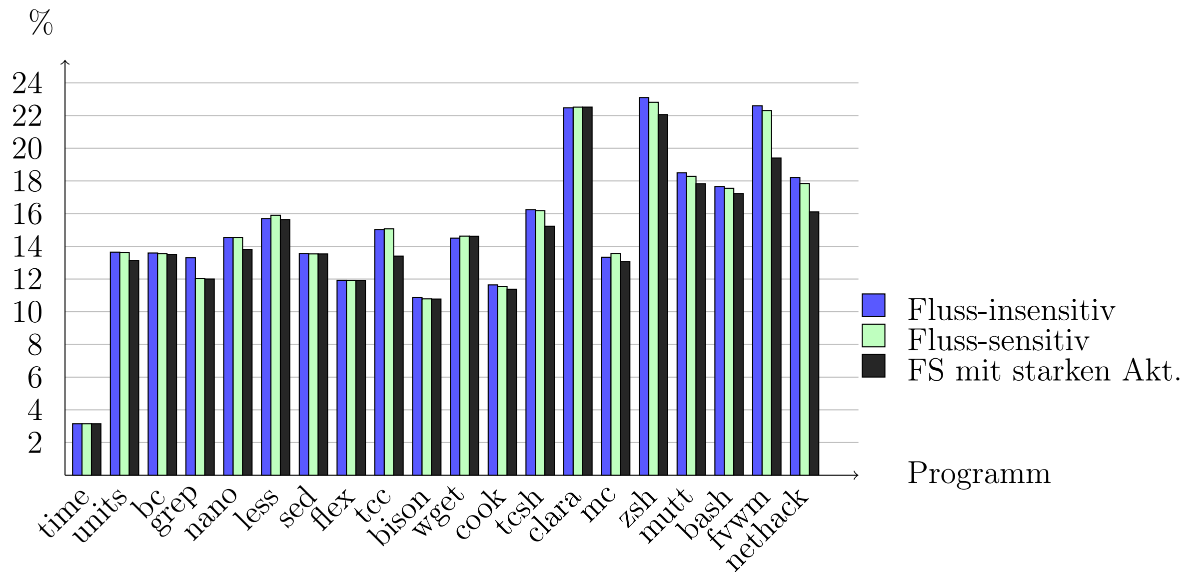


Abbildung 12.18: Anteil der Phi-Knoten am ISSA-Graphen (feld-insensitiv)

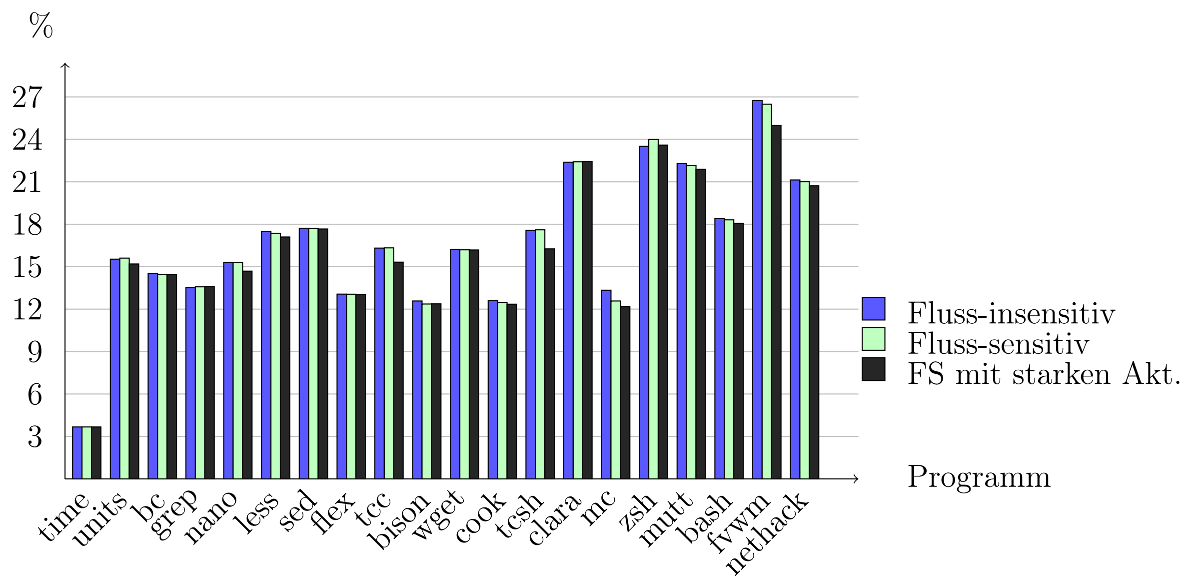


Abbildung 12.19: Anteil der Phi-Knoten am ISSA-Graphen (feld-sensitiv)

Aufrufe erfolgen. Insgesamt wurden knapp 4000 Ziele für indirekte Aufrufe erkannt, wobei die Symboltabelle als Array in der Analyse die Ziele für die verschiedenen Funktionen vereint. Dadurch entsteht interprozeduraler Datenfluss, der in Wahrheit nicht stattfindet.

12.3.1 Starke und schwache Aktualisierungen

Im Hinblick auf den Nutzen eines präziseren Objektmodells für Arrays und Strukturen sowie für die Erweiterung zur Unterstützung indirekter starker Aktualisierungen wollen wir auch quantifizieren, wodurch schwache Aktualisierungen entstehen. Diesem Zweck dienen die folgenden Untersuchungen.

Die Abbildungen 12.20 und 12.21 zeigen zunächst, wie viele indirekte Definitionen mehr als ein Zeigerziel erhalten haben und daher schwache Aktualisierungen sind. Wir erkennen einen deutlichen Anstieg mit zunehmender Programmgröße. Die Fluss-Sensitivität kann diese Zahl in einigen Fällen reduzieren. Im Vergleich dazu sehen wir in den Tabellen 12.11 und 12.12 die Zahl der indirekten Definitionen mit genau einem Ziel. Sie stellen die unmittelbaren Kandidaten für indirekte starke Aktualisierungen dar. Gemäß den Ausführungen in Kapitel 3 erlauben einige Ziele jedoch keine starken Aktualisierungen. Darum haben wir die Darstellung getrennt in solche Fälle, die eine starke Aktualisierung erlauben, und die übrigen. Ihre Zahl ist zusammen beachtlich, was unsere Beschäftigung mit den indirekten starken Aktualisierung rechtfertigt. Bei bison übersteigt ihre Zahl sogar deutlich die Zahl derjenigen mit mehr als einem Ziel. Durch Definitionen ohne Ziel sinkt wiederum in manchen Fällen mit höherer Präzision die gezeigte Anzahl.

Bei der Betrachtung der Fälle, die starke Aktualisierungen erlauben, muss man bedenken, dass dies sowohl mit als auch ohne Unterstützung für indirekte starke Aktualisierungen gilt. Jedoch kann die Analyse nur mit der Unterstützung dieses Potential ausnutzen, ohne sie wird trotz allem eine schwache Aktualisierung angelegt. Die Zahl dieser Fälle nimmt leicht zu, wobei wir fv-wm als positiven Ausreißer erkennen (feld-sensitiv auch bison). Feld-sensitiv liegt die Zahl im Allgemeinen höher, so dass wir hier einen positiven Effekt dieser Präzisionssteigerung sehen.

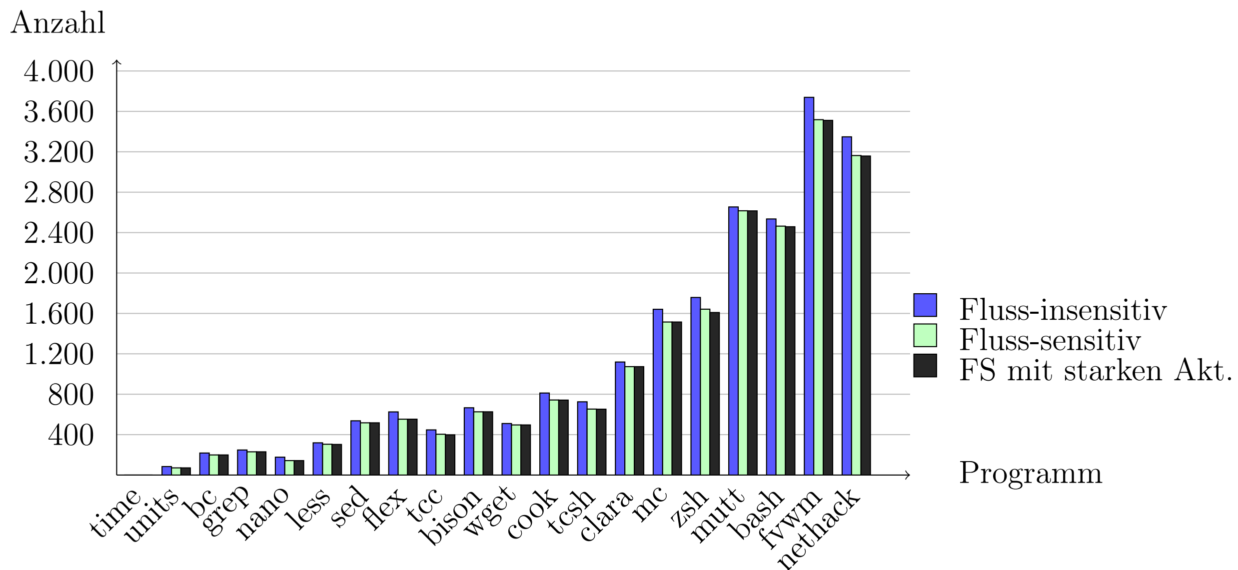


Abbildung 12.20: Indirekte Definitionen mit mehr als einem Ziel (feld-insensitiv)

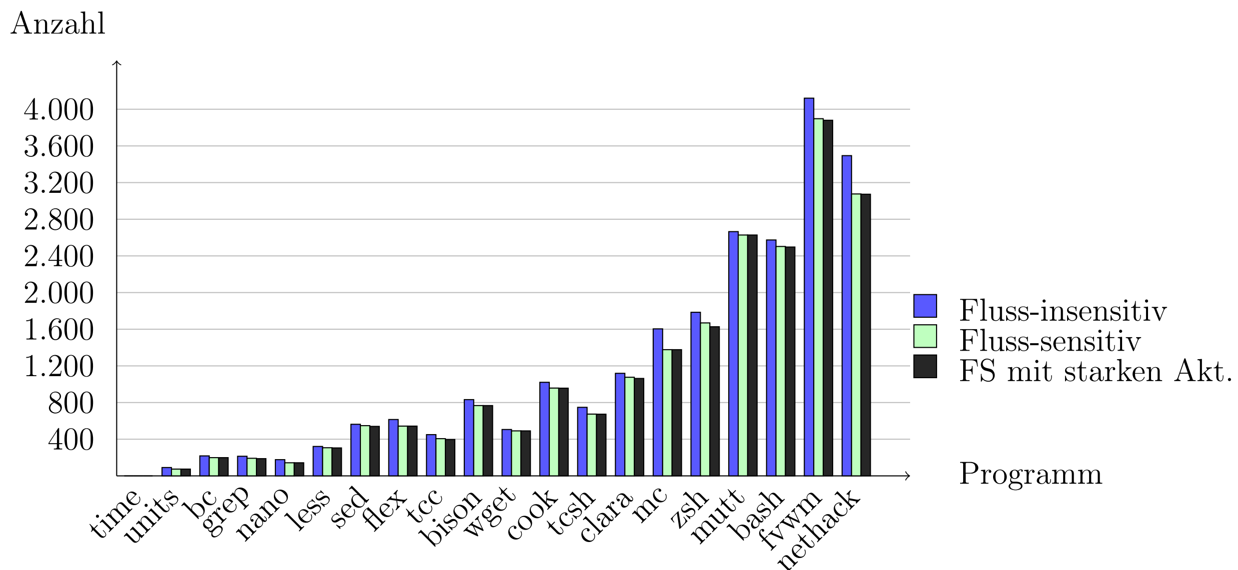


Abbildung 12.21: Indirekte Definitionen mit mehr als einem Ziel (feld-sensitiv)

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	0	0	0	5	0	0
units	20	0	0	46	4	0
bc	2	0	0	2	0	0
grep	6	0	0	76	0	0
nano	18	0	0	18	0	0
less	26	0	0	27	0	0
sed	56	0	0	215	0	0
flex	26	0	0	41	0	0
tcc	14	0	0	37	1	0
bison	9	0	0	759	0	0
wget	67	2	0	208	0	0
cook	28	0	0	38	0	0
tcsh	54	0	-2	128	0	-2
clara	57	0	0	116	32	0
mc	56	0	0	105	-2	0
zsh	61	0	0	138	0	0
mutt	115	1	0	194	1	0
bash	71	0	0	84	0	0
fvwm	210	23	0	984	0	-3
nethack	84	1	0	243	4	0

Tabelle 12.11: Indirekte Definitionen mit genau einem Ziel, welches starke Aktualisierungen erlaubt

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	6	0	0	23	0	0
units	69	13	0	36	13	0
bc	69	19	0	75	19	0
grep	180	18	0	216	0	4
nano	40	34	0	40	34	0
less	32	14	0	32	14	0
sed	259	20	0	110	15	8
flex	105	72	0	107	72	0
tcc	336	42	4	419	37	4
bison	996	36	0	221	65	0
wget	462	10	0	334	15	0
cook	300	69	0	314	63	0
tcsh	487	72	0	428	74	0
clara	448	45	0	389	11	13
mc	1.020	116	0	1.185	227	0
zsh	588	116	10	529	116	20
mutt	241	36	0	183	36	0
bash	358	71	2	345	71	2
fvwm	2.515	122	6	1.910	224	2
nethack	906	181	4	1.077	414	2

Tabelle 12.12: Indirekte Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt

Für ein besseres Verständnis der Ursachen für indirekte Definitionen, die zwar genau ein Ziel haben, aber dennoch nicht in starke Aktualisierungen übersetzt werden können, weil es das Ziel nicht erlaubt, betrachten wir nun die häufigsten Ursachen. Dazu geben die Abbildungen 12.22 und 12.23 den Anteil dieser indirekten Definitionen an, welche sich auf Arrays beziehen und aus diesem Grund nur schwache Aktualisierungen sein können. Der Trend scheint hierbei eine leichte Zunahme des Anteils mit größer werdenden Programmen zu sein, wobei es viele Ausnahmen gibt. In manchen Programmen (z.B. bei less, tcc und mutt) stellen die Arraymodifikationen mehr als die Hälfte dieser schwachen Aktualisierungen. Auf der anderen Seite gibt es aber auch Beispiele wie bison, cook und fvwm, bei denen der Anteil sehr gering ist. Die Feld-Sensitivität ändert an diesen Ergebnissen nicht viel: Z.B. sed, tcc und mutt haben dann einen höheren Array-Anteil, während er bei nethack sinkt. Die Fluss-Sensitivität reduziert die Bedeutung der Arrays in dieser Betrachtung einige Male.

Als zweite Ursache betrachten wir mit den Abbildungen 12.24 und 12.25 die Heapobjekte. Hier sehen wir eine generell höhere Bedeutung. Feld-sensitiv nimmt der Anteil der Heapobjekte als Verursacher sogar noch zu, da hier die feld-insensitive Zusammenlegung als weiterer Verursacher wegfällt. Die Diagramme sprechen eine klare Sprache: Will man mehr indirekte starke Aktualisierungen unterstützen, so sollte man sich Gedanken zum Umgang mit Heapobjekten machen.

Die weiteren Verursacher für indirekte Definitionen mit einem Ziel, welche aber nur schwache Aktualisierungen sind, betrachten wir hier nicht näher. Dazu zählen feld-insensitiv die Zusammenlegung der Felder zu Strukturobjekten sowie z.B. Situationen, in denen die Adresse eines Objekts als Parameter an eine externe Funktion übergeben wird, wofür unsere Implementierung den Aufruf ebenfalls vorsichtshalber als schwache Aktualisierung des Objekts betrachtet (als Zuweisung von Unknown_Objects). Ist das Objekt in diesem Fall nur indirekt gegeben, so ergibt dies eine indirekte Definition und damit auch feld-sensitiv weitere schwache Aktualisierungen.

Schließlich gibt es noch direkte Definitionen, die schwache Aktualisierungen darstellen. Zu dieser Gruppe zeigen die Diagramme 12.26 und 12.27,

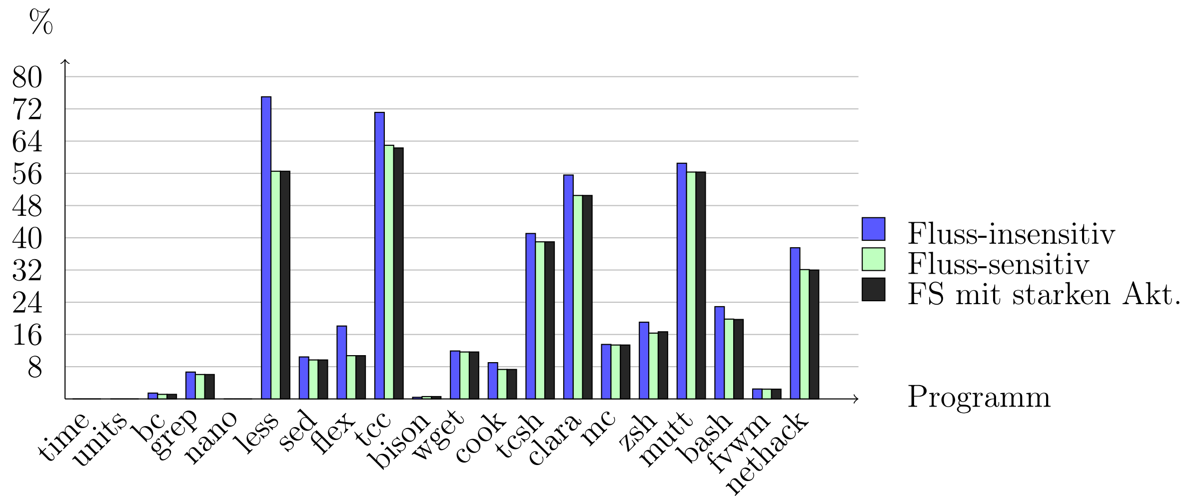


Abbildung 12.22: Anteil indirekter Arraymodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-insensitiv)

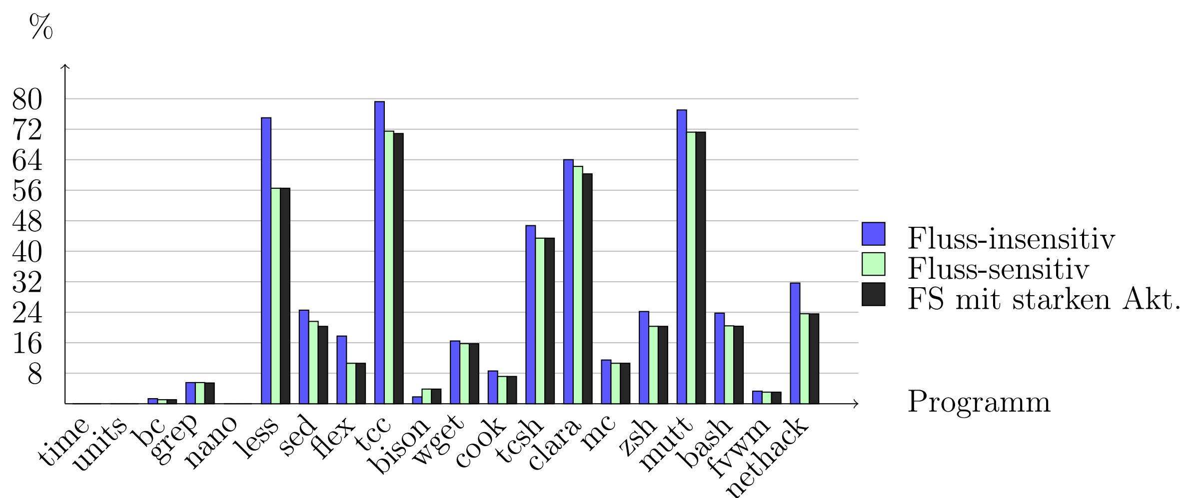


Abbildung 12.23: Anteil indirekter Arraymodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-sensitiv)

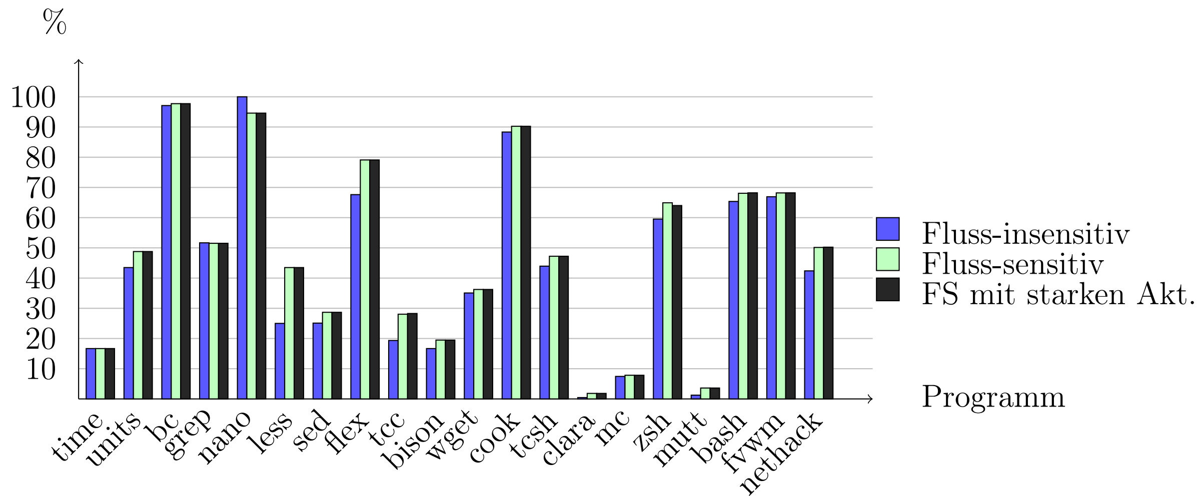


Abbildung 12.24: Anteil der Heapmodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-insensitiv)

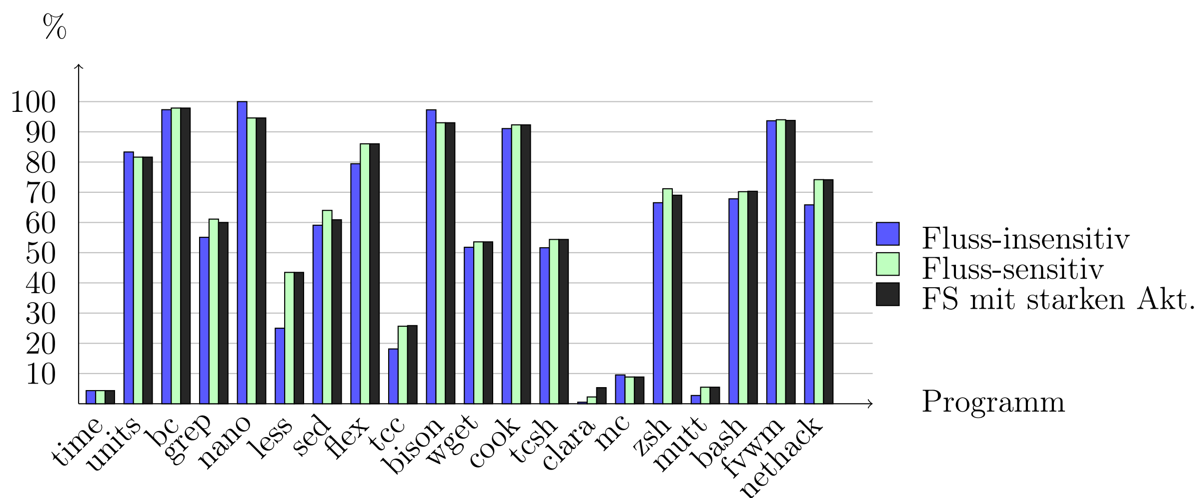


Abbildung 12.25: Anteil der Heapmodifikationen an den indirekten Definitionen mit genau einem Ziel, welches keine starken Aktualisierungen erlaubt (feld-sensitiv)

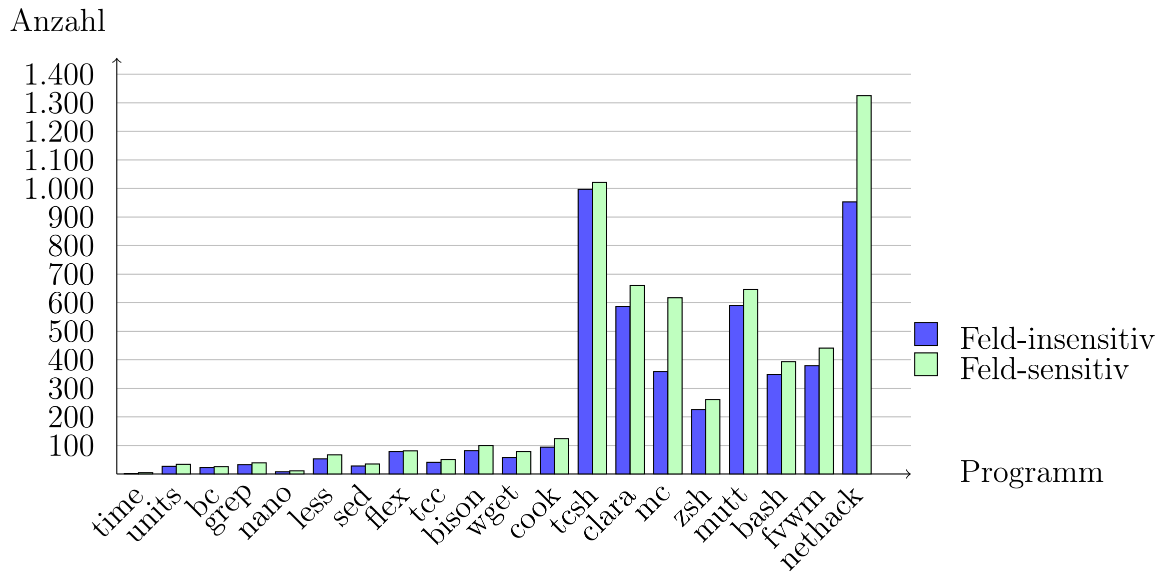


Abbildung 12.26: Direkte Array-Modifikationen

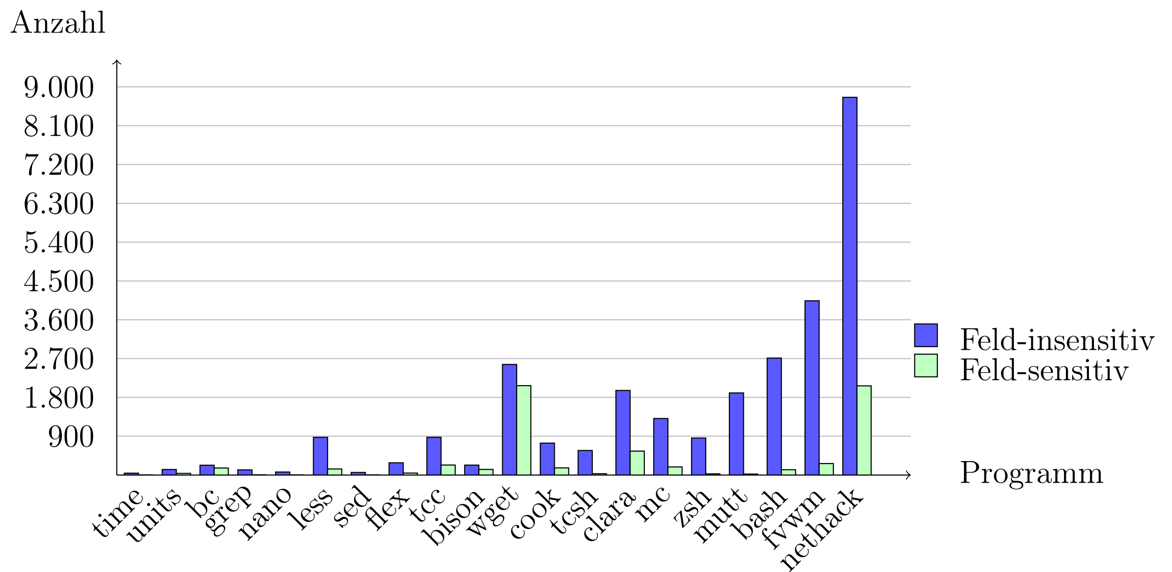


Abbildung 12.27: Sonstige schwache direkte Aktualisierungen

wie viele jeweils aufgrund eines Arrays bzw. anderer Gründe zustande kamen (vor allem wegen der feld-insensitiven Zusammenlegung der Felder von strukturierten Objekten und der eben erwähnten Behandlung von Objekten, deren Adresse an externe Funktionen übergeben wird). Diese Größen hängen nur von der IR (nicht aber von der Fluss-Sensitivität) ab, weswegen die Diagramme jeweils nur zwei Werte pro Programm zeigen.

Die Zahl der direkten Array-Modifikationen nimmt mit der Programmgröße zu, wobei es wieder Ausreißer wie z.B. tcsh gibt. Feld-sensitiv sehen wir mehr Fälle, da strukturierte Arrays mit mehreren Objekten modelliert werden. In der Gruppe der sonstigen Verursacher fällt im Vergleich deutlich der Zugewinn durch die Feld-Sensitivität auf: Diese Gruppe fällt dann kleiner aus, weil die Verwendung der Strukturobjekte als Verursacher entfällt.

12.3.2 Größenreduktionen

Unsere Implementierung arbeitet nicht direkt auf dem bislang vermessenen ISSA-Graphen. Stattdessen erfolgen zwei spürbare Größenreduktionen:

1. Anstelle aller ISSA-Knoten verwenden wir einen reduzierten Graphen, der nur noch die allgemeinen Definitionen enthält.

2. Eine weitere Beschleunigung und Größenreduktion entsteht durch die Verwendung des SCC-DAG anstelle des (reduzierten) ISSA-Graphen, wie es in unserer effizientesten Implementierung der Fall ist.

Diese Größenreduktionen wollen wir hier näher beleuchten. Außerdem betrachten wir hier auch das Pruning, welches zwar nicht die Größe während der Fixpunkt-Iteration reduziert, wohl aber für nachfolgende Werkzeuge. (Der bislang vermessene finale ISSA-Graph ist der Graph, wie ihn die Analyse nach dem Pruning speichert.)

Zunächst zeigt die Tabelle 12.13 die Größe des reduzierten ISSA-Graphen vor dem Pruning. Auch die fluss-insensitive Variante haben wir so umgesetzt, dass sie nach ihrem Fixpunkt zunächst diese Form des ISSA-Graphen errichtet, bevor die Finalisierung mit dem Pruning und dem Ergänzen der allgemeinen Verwendungen erfolgt. Die Zahlen zeigen eine schnelle Zunahme

der Anzahl an allgemeinen Definitionen bei den größten Programmen. Sie ähneln im Verlauf und im Unterschied der Präzisionsstufen der Tabelle 12.9 für die Werte des finalen Graphen. Die Absolutwerte jedoch sind nur noch etwas mehr als halb so hoch wie die Zahl der Knoten im finalen ISSA-Graphen. Bedenkt man, dass der finale Graph durch das Pruning sogar noch spürbar verkleinert wurde, so ist dies ein deutliches Argument für unseren Ansatz, während der Fixpunkt-Iteration nur die allgemeinen Definitionen zu benutzen. Die fluss-sensitive Zunahme bei nano ist ein zufälliges, Reihenfolge-bedingtes Artefakt: Es entstehen hier Eingänge (etc.) bei der Suche der gültigen Definition, welche jedoch am Ende wieder gepruned werden können, da zwischenzeitlich eine solche Definition aufgetaucht ist.

Um den Effekt des Prunings zu messen, liefern die Abbildungen 12.28 und 12.29 den prozentualen Anteil der dadurch aus dem in der vorigen Tabelle vermessenen reduzierten Graphen entfernten allgemeinen Definitionen. (Da unsere Finalisierung die Ergänzung der allgemeinen Verwendungen und das Pruning zusammen durchführt, anstatt erst alle Verwendungen einzufügen und anschließend einen beachtlichen Teil davon wieder zu löschen, können wir keine Vergleichszahl für allgemeine Verwendungen bieten.)

Feld-insensitiv erkennen wir, dass mit zunehmender Programmgröße ein immer kleinerer Teil des Graphen durch das Pruning entfernt wird, wobei die Anteile zwischen den Programmen wieder schwanken. Feld-sensitiv dagegen ist kein Trend erkennbar. Hier können wir tendenziell einen größeren Anteil des Graphen prunen, wobei auch bei großen Programmen wie fvwm noch mehr als 40% entfernt werden. Die bei Feld-Sensitivität separaten Datenflussgraphen der einzelnen Felder scheinen also jeweils einige Knoten zu verursachen, die am Ende dem Pruning zum Opfer fallen, während dies bei der Zusammenlegung der Felder pro Struktur nur einmal auftritt und eine höhere Chance auf eine Verwendung besitzt, die das Pruning verhindert, da die Verwendung eines einzigen Feldes für die Erhaltung des Datenflusses zur gesamten Struktur genügt. Es ist also durchaus ein Zeichen der höheren Präzision, dass im Vergleich zur feld-insensitiven Variante ein größerer Graphteil entfernt werden kann. Die drei Fluss-Sensitivitätsstufen haben dennoch keine kontinuierlich gültige Relation zueinander.

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	804	0	0	960	0	0
units	12.366	-43	-462	19.098	-26	-1.201
bc	33.635	-192	-53	43.590	-119	-53
grep	28.949	-6.029	-9	27.558	-418	-219
nano	64.754	-109	-7.703	153.990	1.816	-62.306
less	363.240	-9.100	-11.529	460.602	-9.491	-41.901
sed	98.144	-101	-45	130.699	-217	-513
flex	40.539	-67	-8	48.796	-102	-16
tcc	222.944	-263	-75.969	499.197	-1.593	-180.194
bison	109.708	-1.703	-242	1.347.899	-12.820	-708
wget	105.022	-351	-157	131.404	-5.450	-615
cook	3.363.794	-104.551	-214.837	9.066.774	-310.071	-873.837
tcsh	3.770.611	-341.738	-846.542	7.919.277	-976.084	-2.915.817
clara	223.849	-272	-35	263.675	-253	-85
mc	5.533.336	-852.999	-726.836	9.972.414	-872.459	-1.554.701
zsh	7.871.669	-771.891	-2.399.724	16.717.713	-1.941.428	-6.815.158
mutt	4.887.759	-257.738	-291.048	9.381.651	-905.730	-1.254.672
bash	13.473.454	-277.421	-2.644.146	23.194.780	-438.442	-7.357.966
fvwm	7.246.012	-402.712	-2.497.471	55.607.904	-5.925.120	-31.711.060
nethack	34.181.444	-2.528.380	-9.864.124	133.936.936	-4.349.576	-24.121.600

Tabelle 12.13: Anzahl allgemeiner Definitionen im ISSA-Graphen vor dem Pruning

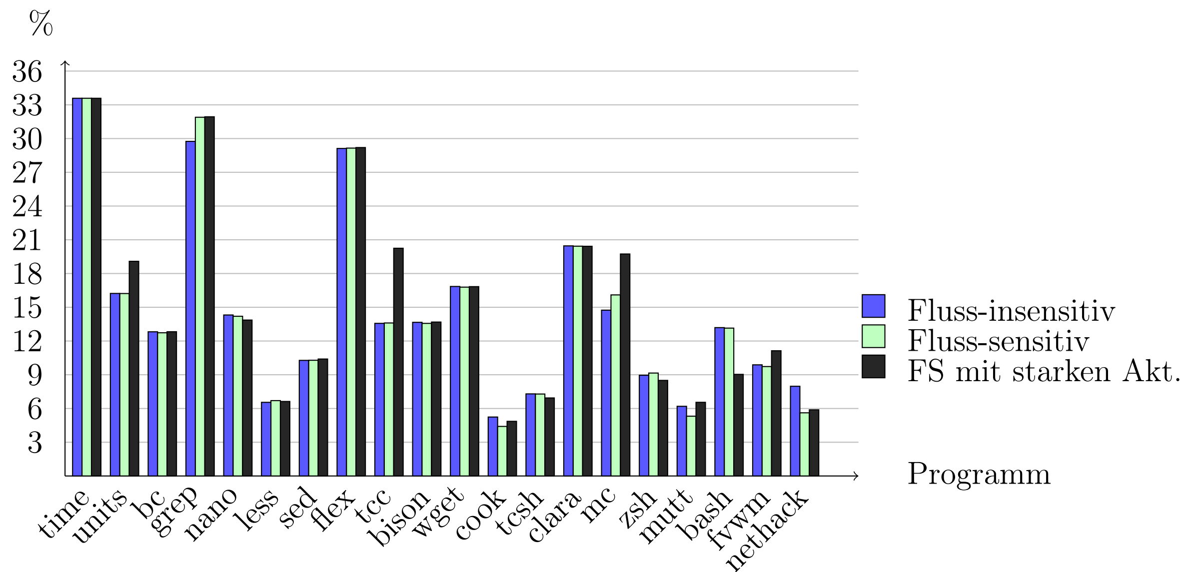


Abbildung 12.28: Durch Pruning entfernte allgemeine Definitionen (feld-insensitiv)

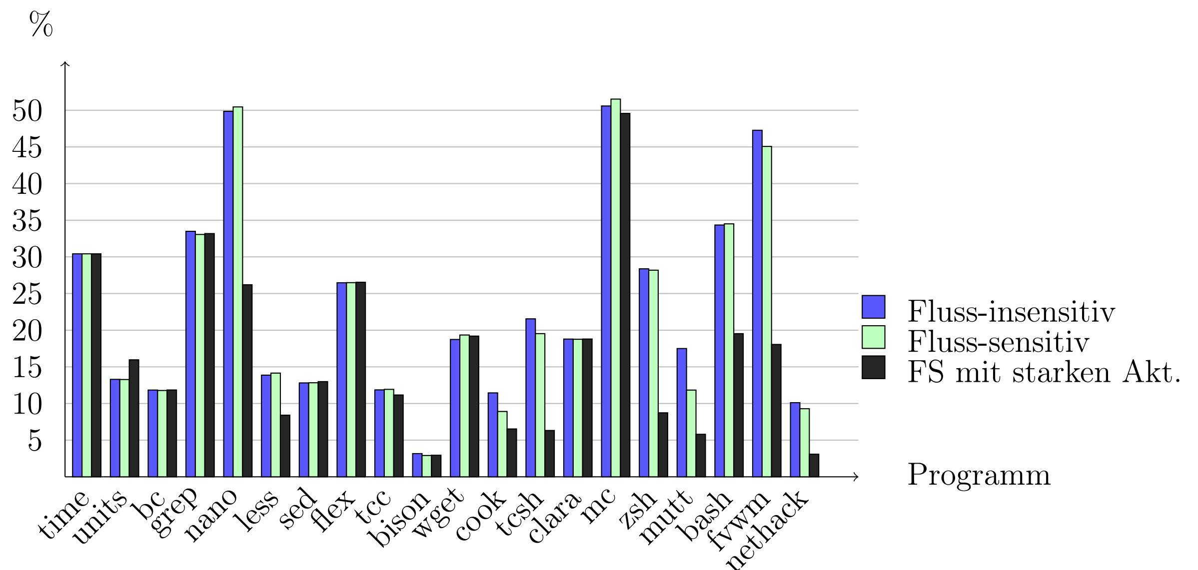


Abbildung 12.29: Durch Pruning entfernte allgemeine Definitionen (feld-sensitiv)

Schließlich haben wir zur weiteren Größenreduktion den SCC-DAG des reduzierten ISSA-Graphen für die Propagierung genutzt. Dessen Größe (in Zahl der Knoten) liefern uns die Abbildungen 12.30 und 12.31. In diesen Diagrammen haben wir nur die fluss-sensitiven Ausprägungen aufgeführt, da die fluss-insensitive Ausprägung den SCC-DAG des nicht vergleichbar kleinen Constraint-Graphen einsetzt.

Wir erkennen in den Diagrammen wieder ein Wachstum des SCC-DAG mit der Programmgröße, diesmal aber gemäßigter als ohne die Zyklenkontraktion. Neben dem sanfteren Wachstum fallen auch die deutlich kleineren Absolutzahlen auf. Hier haben wir es nun mit ein paar hunderttausend Knoten zu tun, während es ohne Zyklenkontraktion noch mehrere Millionen waren. Diese Effektivität der Zyklenkontraktion schlägt sich auch in einer deutlichen Laufzeitverbesserung nieder, wie wir im nächsten Kapitel sehen werden. Mit indirekten starken Aktualisierungen hat der SCC-DAG dem Diagramm zu Folge in der Regel mehr Knoten als ohne diese Erweiterung. Dies ist jedoch wieder ein Zeichen der höheren Präzision: Dadurch, dass Kanten fehlen, sinkt auch die Chance auf Zyklen. Bei grep und nano dagegen ist die Reduktion der Knotenzahl durch die höhere Präzision gewichtiger, so dass dort die Zahl der Knoten im SCC-DAG mit indirekten starken Aktualisierungen sinkt.

Abschließend betrachten wir mit den Abbildungen 12.32 und 12.33 die Größe der maximalen SCC im SCC-DAG. Das vermittelt einen Eindruck davon, wie groß das Sparpotential mit einer Zyklenkontraktion ist. Bei nethack finden sich z.B. grob die Hälfte aller Knoten vor dem Pruning in einer einzigen SCC wieder. Dass so viele Knoten in einer einzigen SCC landen, ist nicht ungewöhnlich: Es passt zu einem Resultat von Erdös, nach dem in einem zufälligen Graphen eine sehr große SCC zu finden ist, sobald die Zahl der Kanten ungefähr der Zahl der Knoten entspricht oder gar höher ist (vgl. z.B. [Erdös und Rényi 1961, S. 345f]). Zwar haben wir keine zufälligen Graphen, aber dennoch scheint es zuzutreffen, dass eine große SCC vorhanden ist.

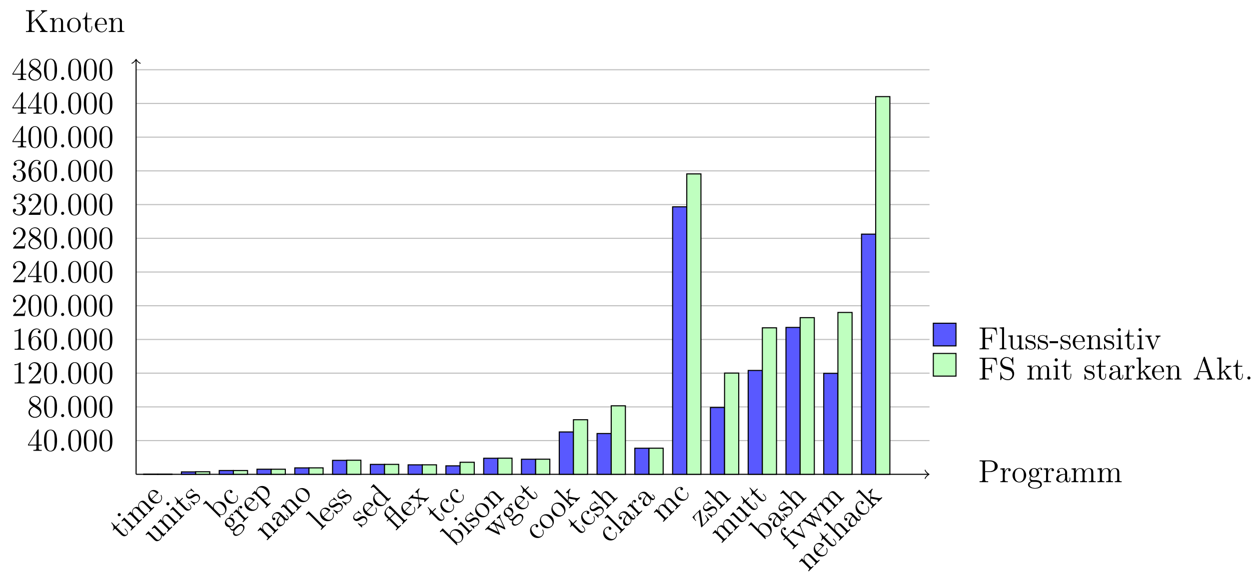


Abbildung 12.30: Anzahl Knoten im SCC-DAG (feld-insensitiv)

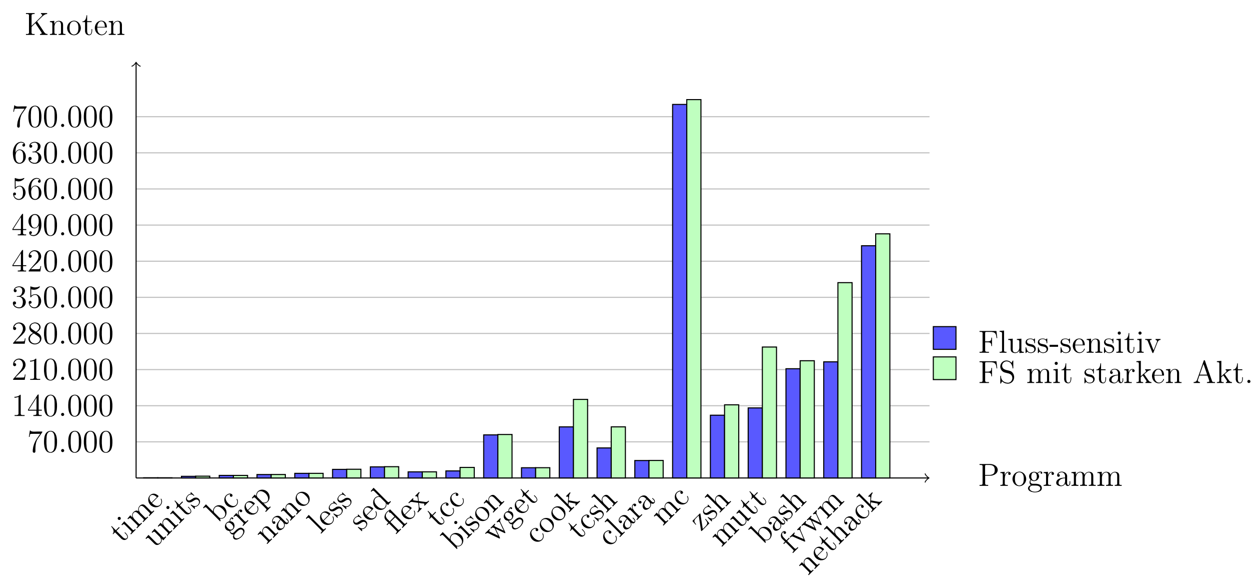


Abbildung 12.31: Anzahl Knoten im SCC-DAG (feld-sensitiv)

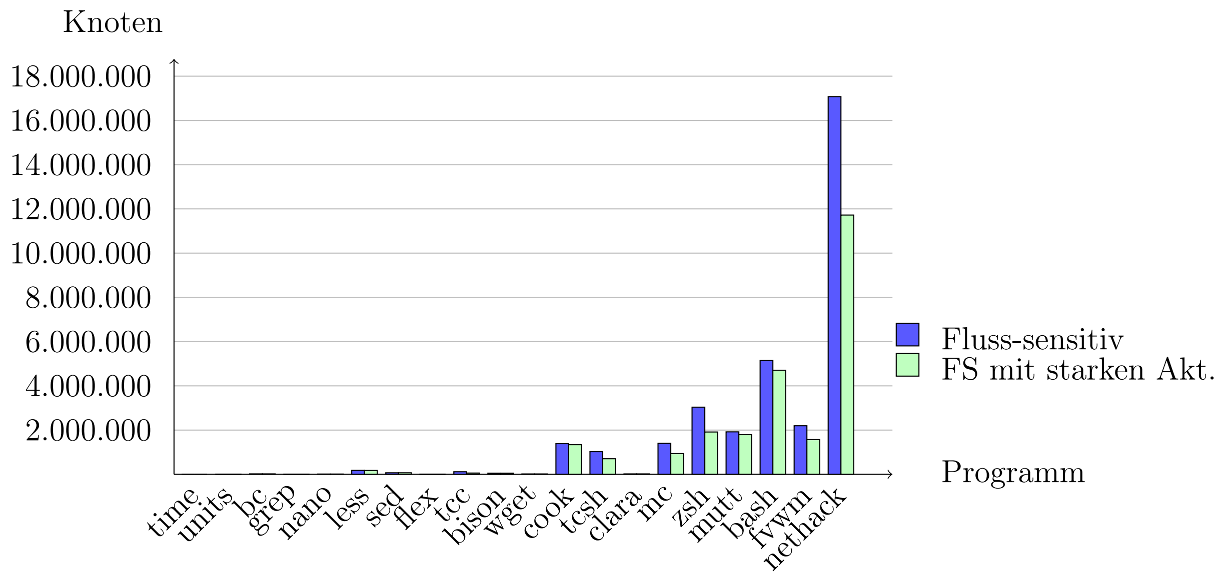


Abbildung 12.32: Größe der maximalen SCC (feld-insensitiv)

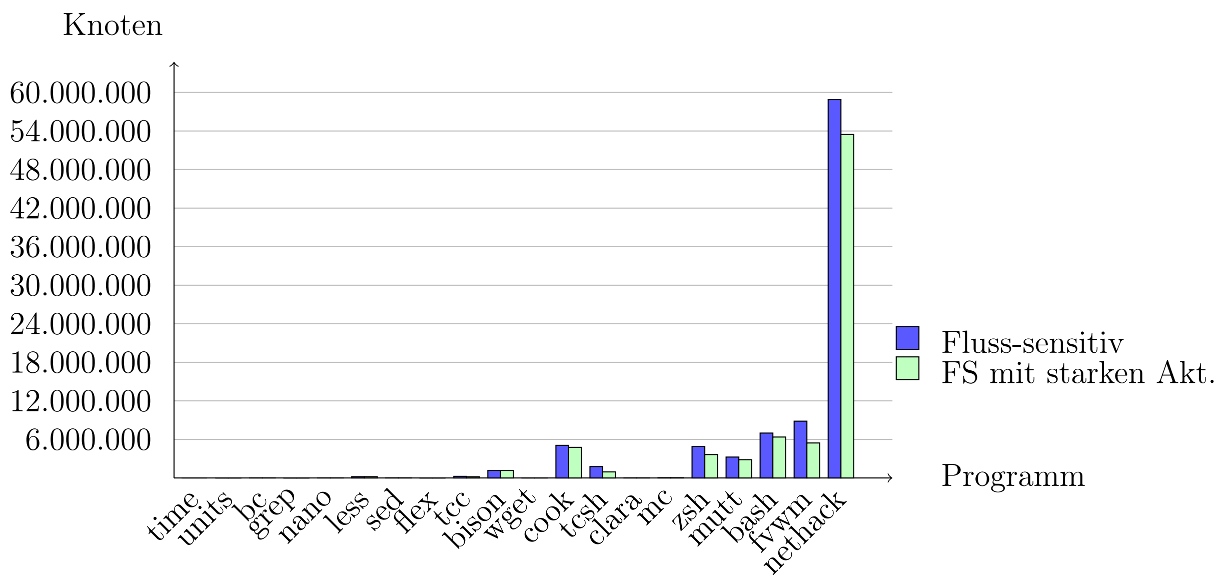


Abbildung 12.33: Größe der maximalen SCC (feld-sensitiv)

Kapitel 13

Evaluation der Skalierbarkeit

Mit der Betrachtung der Resultate haben wir einen wesentlichen Aspekt für die empirische Evaluation der Analyse im letzten Kapitel behandelt. In diesem Kapitel setzen wir die Evaluation fort, indem wir den Aspekt der Skalierbarkeit untersuchen. Neben den offensichtlichen Größen wie Laufzeit und Speicherbedarf betrachten wir auch einige spezielle Größen für die Erweiterung zur Unterstützung indirekter starker Aktualisierungen. Wir benutzen unsere effizienteste Implementierung (mit inkrementeller Zyklenkontraktion und erschöpfender Rückwärts-Propagierung) als Referenz und vergleichen deren Effizienz in Abschnitt 13.4 mit anderen Implementierungsvarianten. Abschließend enthält das Kapitel eine Zusammenfassung der empirischen Evaluation.

13.1 Laufzeit

Zunächst zeigen uns die beiden Abbildungen 13.1 und 13.2 die im Algorithmus verankerte, nicht von der Umgebung abhängige Zahl der Iterationen. Für die beiden Analysevarianten mit indirekten starken Aktualisierungen sehen wir wie gehabt die Zahlen mit Relaxation. Die klar zu erkennenden Ausschläge in den Diagrammen bei diesen Varianten gehen auch auf das Konto eben dieser Relaxation. Den Effekt der Relaxation werden wir im Detail in Abschnitt 13.3 besprechen.

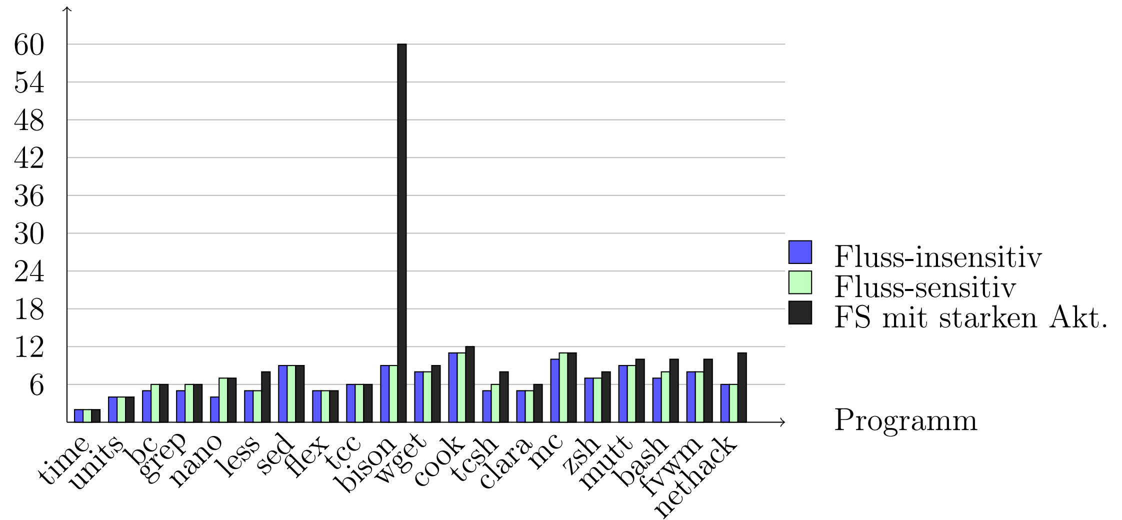


Abbildung 13.1: Anzahl Iterationen (feld-insensitiv)

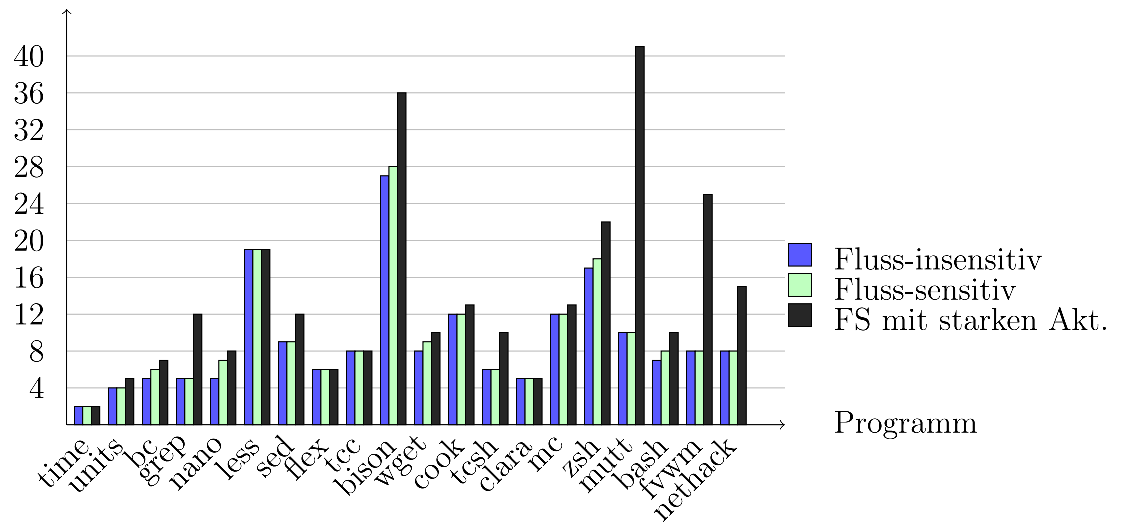


Abbildung 13.2: Anzahl Iterationen (feld-sensitiv)

Die Zahl der Iterationen bleibt ansonsten selbst für die großen Programme erfreulich gering. Das bedeutet eine generell gute Konvergenz der Analyse und macht die erschöpfende Propagierung für große Programme interessant, bei denen der Speicherbedarf der inkrementellen Variante möglicherweise die Anwendbarkeit der Analyse begrenzt.

Feld-sensitiv steigt die Zahl der Iterationen bei vielen Programmen an. Das Programm bison ist hiervon deutlich betroffen. Das passt zu den Beobachtungen aus dem vorigen Kapitel, in dem wir bereits begründet haben, warum dieses Programm feld-sensitiv auch spürbar mehr Zeigerziele und damit verbunden einen größeren ISSA-Graphen erhält. Less und zsh sind weitere Beispiele mit mehr Iterationen.

Außerdem erkennen wir mit den Betrachtungen in Abschnitt 13.3, dass die Unterstützung für indirekte starke Aktualisierungen auch ohne Relaxation die Zahl der Iterationen leicht erhöht. Dagegen macht es interessanterweise kaum einen Unterschied, ob die Analyse fluss-sensitiv läuft oder nicht. Auffällig ist, dass fluss-insensitiv nie mehr Iterationen als fluss-sensitiv benötigt werden, und nur in manchen Fällen weniger Iterationen. Die höhere Präzision benötigt also nicht zwingend mehr Iterationen, muss dafür aber wie im vorigen Kapitel gesehen mit einem größeren Graphen zurecht kommen.

Als weitere abstrakte Größe mit Bezug zur Laufzeit präsentieren wir in Tabelle 13.1 die Zahl der Schritte, die insgesamt in der Propagierung (als dem in der Regel teuersten Analysenteil) ausgeführt werden. Ein Schritt ist dabei das Traversieren einer Kante im Propagierungsgraphen. Die hier gezeigten Werte gelten für unsere effizienteste Implementierung, welche die inkrementelle Zyklenkontraktion nutzt und auf dem SCC-DAG rückwärts (erschöpfend) propagiert. Der Vergleich mit anderen Propagierungs-Strategien folgt später.

Die Zahl dieser Schritte erreicht astronomische Größenordnungen. Sie hängt unter anderem davon ab, wie groß der jeweilige SCC-DAG ist. So war dieser z.B. bei nethack relativ groß und für die Analyse mit indirekten starken Aktualisierungen noch einmal deutlich größer als ohne diese Erweiterung. Das erklärt teilweise die hier nun zu beobachtenden Unterschiede der verschiedenen Analyse-Ausprägungen.

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	93	40	0	99	40	0
units	97.470	32.448	-651	20.962	24.605	13.742
bc	312.672	281.276	-5.836	183.154	226.979	88.276
grep	294.095	107.335	-29.031	22.500	17.058	65.627
nano	637.935	554.038	-14.024	271.669	887.592	-48.158
less	2.099.339	3.546.813	5.197.797	2.292.075	12.524.366	-242.415
sed	18.516.080	1.821.504	-187.968	1.979.921	1.369.127	720.023
flex	515.319	224.089	1.917	456.831	234.055	1.964
tcc	1.457.200	2.229.812	-694.967	1.695.641	2.875.881	-1.087.033
bison	116.537.648	-91.166.000	238.679.568	998.590.208	39.720.384	498.845.376
wget	2.566.033	167.868	449.246	1.542.734	286.475	667.020
cook	895.767.488	-447.672.096	26.216.864	634.376.960	-127.639.200	31.152.992
tcsh	73.749.136	71.087.472	47.369.808	63.486.252	46.035.540	86.074.368
clara	1.810.260	1.927.306	1.023.064	808.268	776.610	1.382
mc	2.225.012.736	-1.436.297.856	-137.666.752	3.042.025	2.498.689	519.810
zsh	568.888.192	-269.734.272	-14.901.120	1.596.962.304	-853.621.376	-50.150.272
mutt	2.676.330.240	-566.878.336	280.777.344	480.166.464	26.235.520	1.985.056.512
bash	3.236.535.296	-1.242.680.704	659.620.736	2.090.980.736	-885.390.720	423.595.264
fvwm	3.844.414.208	-2.390.188.032	610.876.800	205.028.672	393.122.624	1.621.021.568
nethack	2.585.192.960	3.395.585.536	9.124.582.400	501.902.240	2.567.032.064	3.634.416.896

Tabelle 13.1: Schritte der Propagierung

Interessant finden wir die Beobachtung, dass die Zahl der Schritte der fluss-insensitiven Analyse trotz des eigentlich deutlich kleineren Constraint-Graphen durchaus in einigen Beispielen in der gleichen Größenordnung wie bei der fluss-sensitiven Analyse liegt. Das bedeutet, dass die Zyklenkontraktion und die höhere Präzision (hier mit dem Effekt der geringeren Kantenzahl pro Knoten) dafür sorgen, dass im fluss-sensitiven Propagierungs-Schritt nicht wesentlich mehr Arbeit getan werden muss als fluss-insensitiv. Tatsächlich gibt es auch einige Beispiele, bei denen durch die genannten Faktoren die Arbeit im Sinne der durchzuführenden Kantenbesuche fluss-sensitiv deutlich geringer ausfällt als fluss-insensitiv (z.B. mc und bash). Mit den starken Aktualisierungen dagegen erleben wir wie besprochen in einigen Fällen eine höhere Zahl an Iterationen, was sich bei der erschöpfenden Propagierung schnell in einer höheren Zahl an Schritten niederschlägt. Außerdem hatten wir im letzten Kapitel auch gesehen, dass durch die geringere Zahl an Kanten in dieser Analyse-Variante die Zyklenkontraktion nicht mehr ganz so effektiv ist. Die höhere Iterationen-Zahl – verbunden mit dem größeren Graphen – schlägt feld-sensitiv auch bei bison durch, während dort feld-insensitiv die hohe Zahl an Iterationen mit indirekten starken Aktualisierungen aufgrund eines eher kleinen Graphens nicht ganz so viel Einfluss zeigt.

Als nächstes betrachten wir die konkrete Gesamtlaufzeit (in Sekunden) der Analyse auf unserem Testsystem. Dazu zeigt uns die Tabelle 13.2 die gewohnte Übersicht zu den sechs verschiedenen Präzisionsstufen. Diese Darstellung erlaubt uns den Vergleich der verschiedenen Analyse-Ausprägungen sowie eine Beurteilung der Absolutwerte.

Wenn wir die Absolutwerte betrachten, so stellen wir eine erfreulich schnelle Analyse fest. Abgesehen von nethack gelingt die Analyse der Programme in jeweils weniger als zehn Minuten. Feld-insensitiv genügen der fluss-sensitiven Analyse sogar fünf Minuten (etwas mehr bei bash mit starken Aktualisierungen), feld-sensitiv immerhin noch ungefähr acht Minuten. Nethack scheint mit einer klar schlechteren Laufzeit aus dem Rahmen zu fallen. Feld-insensitiv benötigt die langsamste Analyse für dieses Programm etwas mehr als eine halbe Stunde, feld-sensitiv schon mehr als eine Stunde. Wir werden jedoch gleich anhand des Trends bezüglich der Zahl der Instruktionen sehen, dass

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	0,04	-0,01	0,01	0,06	0	0
units	0,18	-0,01	0,01	0,23	-0,01	0,01
bc	0,59	-0,09	0,02	0,74	-0,16	0,07
grep	0,42	-0,09	0,01	0,38	-0,06	0,18
nano	0,83	0	-0,01	1,67	-0,40	-0,27
less	3,74	-0,97	0,33	4,89	-0,60	0,06
sed	1,92	-0,26	0,05	1,34	-0,18	0,22
flex	1,16	-0,40	0,04	1,18	-0,42	0,05
tcc	2,62	-0,50	-0,49	4,95	-0,78	-1,04
bison	9,04	-5,49	23,41	68,97	-13,53	16,49
wget	1,80	-0,36	0,24	1,80	-0,36	0,25
cook	79,54	-31,07	9,01	124,13	-1,17	17,61
tcsh	51,96	-16,12	-0,15	102,06	-38,49	-10,83
clara	2,93	-0,72	0,37	3,38	-1,06	0,19
mc	204,16	-109,31	-2,48	87,73	14,90	7,74
zsh	250,12	-75,38	-26,98	440,28	-106,52	-81,03
mutt	204,62	-82,13	20,16	182,34	-36,97	138,34
bash	441,99	-150,57	45,06	441,79	-105,83	8,88
fvwm	378,43	-200,88	-1,26	548,96	-40,35	-122,81
nethack	1.870,74	-549,57	713	4.037,79	52,93	990,47

Tabelle 13.2: Laufzeit-Übersicht (Sekunden)

es sich auch dabei im Gegenteil um gute (nämlich in der Komplexität niedrige) Ergebnisse handelt. Zudem ist bei diesem Programm auch das fluss-insensitive Vorgehen in gleichem Maße langsam.

Im Vergleich der verschiedenen Präzisionsstufen stellen wir fest, dass die fluss-sensitive Analyse sogar schneller sein kann als ihr fluss-insensitives Pendant, wenn dieses als Vorarbeit zur Berechnung der ISSA-Form benutzt wird. Die Erweiterung mit indirekten starken Aktualisierungen scheint den Zeitbedarf dagegen zu erhöhen. Das passt zu den Beobachtungen bei den vorhin angeführten abstrakteren Messwerten: Beispielsweise bei nethack gingen die Zahl der Iterationen sowie die Zahl der Propagierungsschritte mit der Erweiterung deutlich nach oben. Es verwundert daher nicht, einen ebensolchen Anstieg bei der gemessenen Laufzeit zu beobachten.

Die Feld-Sensitivität erhöht die Laufzeit in allen Abstufungen der Fluss-Sensitivität, jedoch unterschiedlich stark. So kehrt sich beispielsweise bei nethack die Relation von fluss-sensitiv zu fluss-insensitiv um: feld-insensitiv konnte die genauere Analyse schneller sein, feld-sensitiv dagegen nicht mehr.

Neben diesem Vergleich der Analysen-Ausprägungen und der Beurteilung der Absolutwerte ist für die Skalierbarkeit auch der Trend wichtig, wie sich die Laufzeit mit der Programmgröße entwickelt. Dazu zeigen wir zunächst für die fluss-sensitive Analyse ohne indirekte starke Aktualisierungen in den Abbildungen 13.3 und 13.4, wie sich die gerade besprochenen Laufzeiten darstellen, wenn wir die Programme auf der X-Achse maßstabsgetreu zu ihrer Größe (in Instruktionen) auftragen. Unter anderem mit dem Ziel, diesen Trend besser abschätzen zu können, hatten wir nethack als ein Programm aufgenommen, welches deutlich größer als die übrigen ist.

Die Schaubilder zeigen uns nun einen sehr erfreulichen Trend: Gerade unter Einbeziehung von nethack kann man das Wachstum als moderat ansteigende Parabel über der Zahl der Instruktionen sehen (Exponent 2,01 für die Regressionskurve nach OpenOffice). Dies gilt auch feld-sensitiv. Wir sehen jedoch auch, dass eine klare Aussage ohne nethack noch nicht möglich ist: Die Programme mit einer Größe zwischen 40 KInst und 200 KInst streuen deutlich, wobei z.B. bash mit ungefähr 300 Sekunden deutlich länger braucht als das größere fvwm. Feld-sensitiv ist die Streuung geringer.

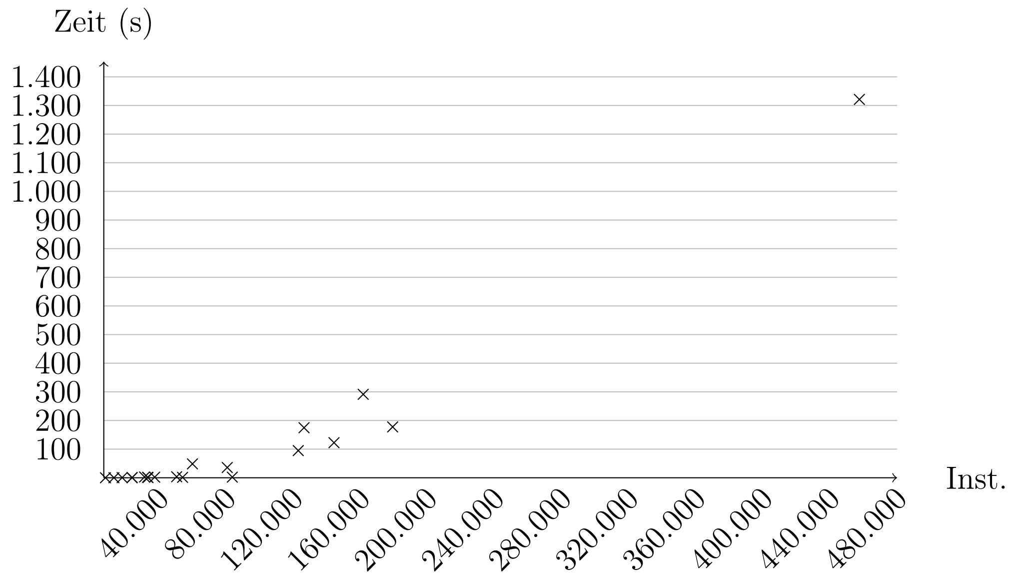


Abbildung 13.3: Laufzeit-Trend (fluss-sensitiv, feld-insensitiv)

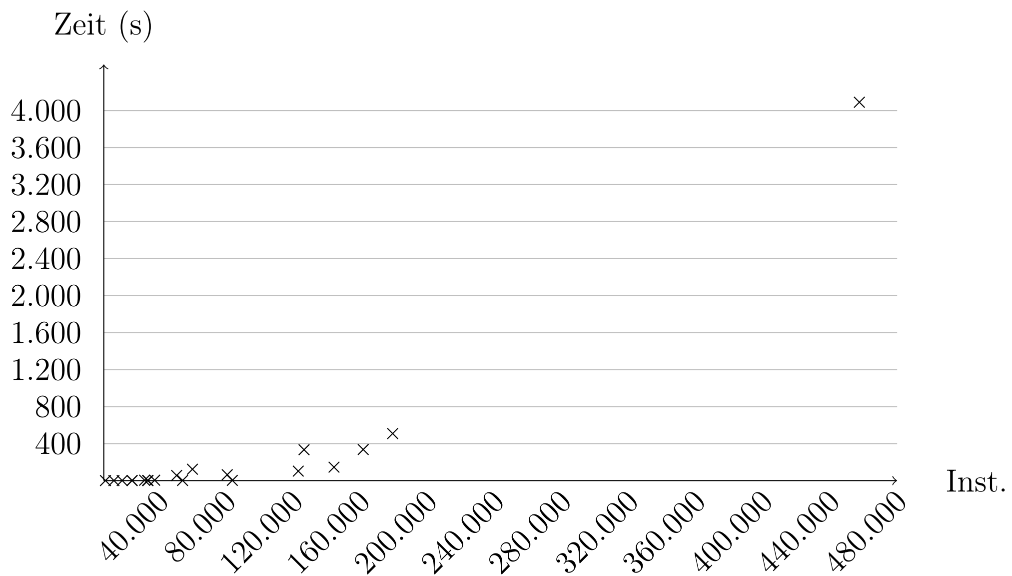


Abbildung 13.4: Laufzeit-Trend (fluss- und feld-sensitiv)

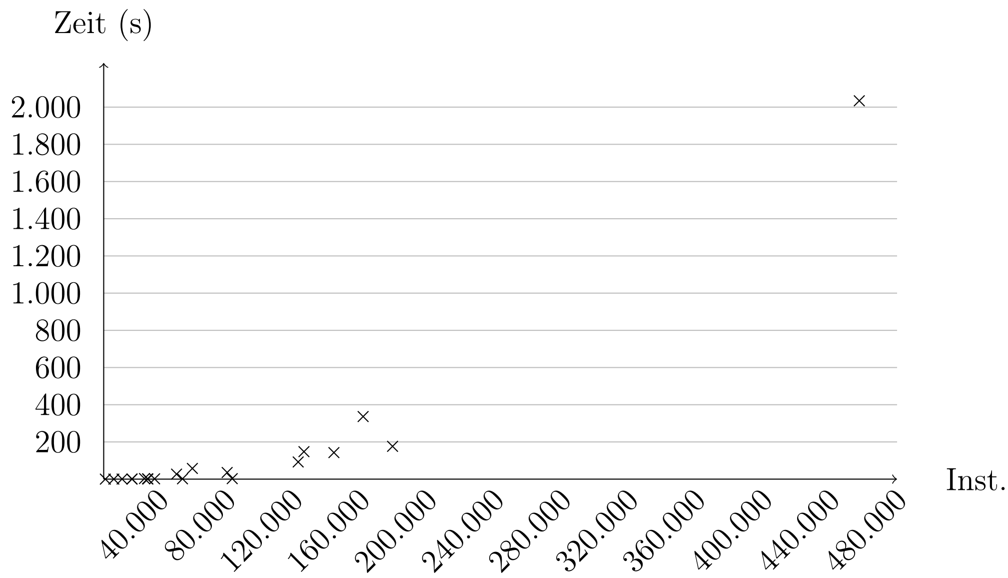


Abbildung 13.5: Laufzeit-Trend (fluss-sensitiv, feld-insensitiv mit starken Akt.)

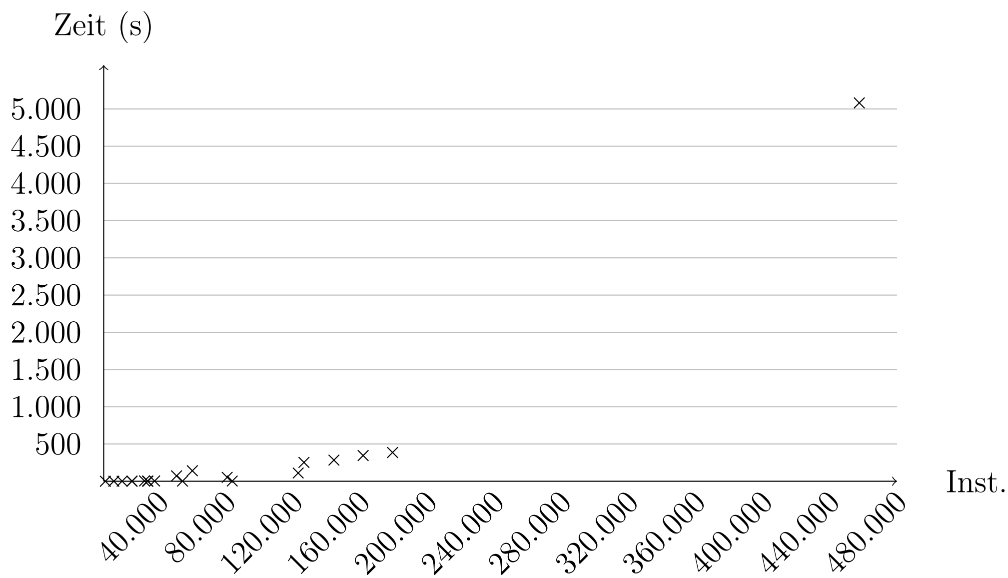


Abbildung 13.6: Laufzeit-Trend (fluss- und feld-sensitiv mit starken Akt.)

Die Abbildungen 13.5 und 13.6 zeigen analog den Trend für die fluss-sensitive Analyse mit der Unterstützung für indirekte starke Aktualisierungen. Auch hier bestätigt sich das quadratisch wirkende Verhalten der Analyse. Feld-sensitiv schlägt der Effekt der Relaxation bei nethack durch, der für mehr Iterationen und damit einhergehend mehr Propagierungsschritte sorgt. Eine inkrementelle Propagierung auf dem inkrementell konstruierten SCC-DAG oder eine andere Strategie anstelle der Relaxation könnte möglicherweise dafür sorgen, dass der hier erkennbare Trend flacher ansteigt, denn abgesehen von nethack erscheint das Wachstum moderat.

Nun wollen wir noch ein wenig aufschlüsseln, welche Teile der Analyse welchen Anteil an der Gesamtlaufzeit haben. Die Abbildungen 13.7 und 13.8 zeigen dazu exemplarisch für die fluss-sensitive Analyse ohne indirekte starke Aktualisierungen den Zeitbedarf einzeln für die drei Teile Graphaufbau, Propagierung und Finalisierung (für die Diskussion sind die in den Diagrammen nicht sichtbaren kleineren Programme nicht wichtig.) Die Initialisierung ist in allen Fällen vernachlässigbar schnell. Für Graphaufbau und Propagierung haben wir die Summe über alle Iterationen gebildet. Die Finalisierung enthält unser kombiniertes Pruning und Ergänzen der allgemeinen Verwendungen.

Insbesondere bei den größten Testprogrammen sticht in dieser Betrachtung feld-insensitiv die Propagierung als teuerster Analyse-Teil hervor. Trotz inkrementeller Zyklenkontraktion und Reduktion auf allgemeine Definitionen spürt man hier also immer noch die im Grapherreichbarkeitsproblem innewohnenden Kosten. Im Übrigen passt dies auch zu unseren theoretischen Resultaten, bei denen ebenfalls die Propagierung eine Dimension teurer war als der Graphaufbau. Interessant ist auch die Feststellung, dass die Finalisierung mit der Ergänzung der allgemeinen Verwendungen ungefähr so teuer ist wie der über die Iterationen verteilte Graphaufbau für die allgemeinen Definitionen.

Feld-sensitiv bemerken wir bei den größten Programmen eine deutliche Zunahme der Kosten für den Graphaufbau und die Finalisierung im Verhältnis zur Propagierung. Im vorigen Kapitel hatten wir bereits festgestellt, dass feld-sensitiv die Zahl der Knoten und Kanten im ISSA-Graphen deutlich ansteigt. Der Graphaufbau muss auf jeden Fall mit diesem größeren Mengen-

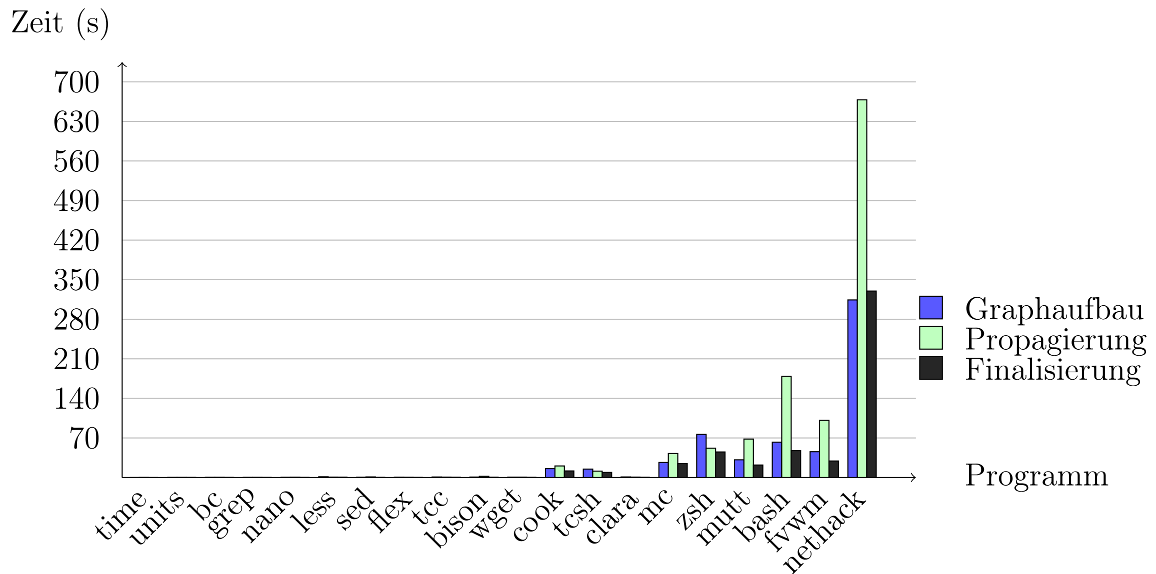


Abbildung 13.7: Laufzeit der einzelnen Analysen-Teile (fluss-sensitiv, feld-insensitiv)

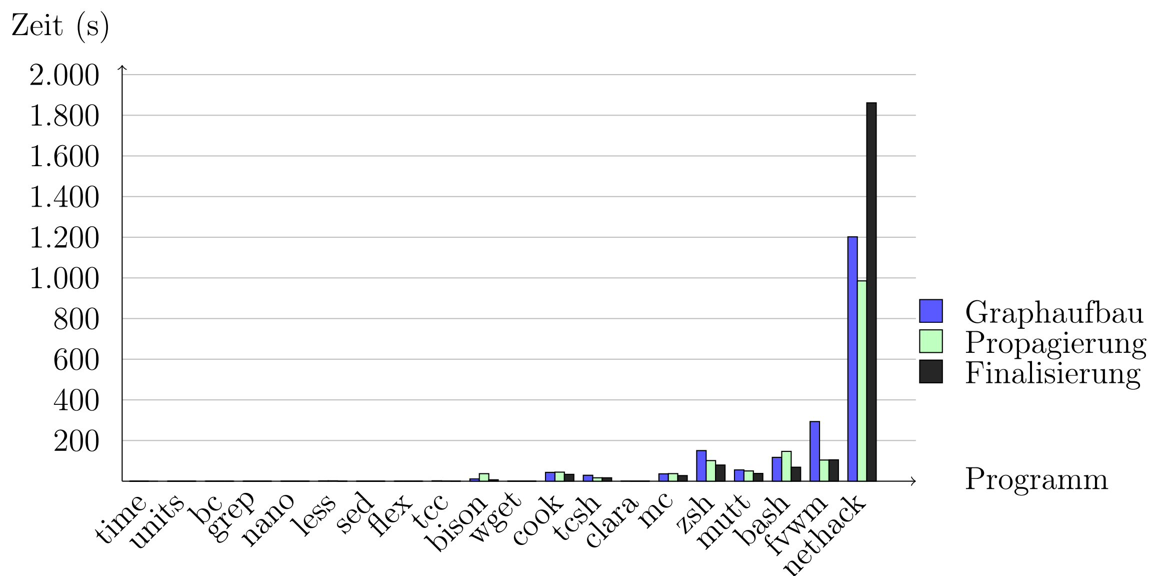


Abbildung 13.8: Laufzeit der einzelnen Analysen-Teile (fluss- und feld-sensitiv)

gerüst zurecht kommen, was die gestiegene Absolutzeit für diesen Teil erklärt. Die Propagierung auf der anderen Seite muss nur diejenigen Graphteile betrachten, die zu den Zeigern gehören. Wenn ein Programm in seinen strukturierten Objekten daher viele Felder hat, die keine Zeiger sind, so steigen die Kosten für deren Graphaufbau, während die Propagierung jedoch nicht so stark darunter leidet. Da die Finalisierung im Wesentlichen ebenfalls ein Graphaufbau ist, gilt für sie die gleiche Aussage wie für den Graphaufbau-Schritt während der Iterationen.

Für einen fairen Vergleich mit der fluss-insensitiven Analyse haben wir auch für diese Variante die Laufzeit-Komponenten aufgeschlüsselt. Die Abbildungen 13.9 und 13.10 präsentieren die dafür relevanten Zahlen. Diesmal beziffern die Zeiten für Graphaufbau und Propagierung also Aufbau und Nutzung des Constraint-Graphen. Zusätzlich sehen wir den Zeitbedarf für den Aufbau der zunächst auf allgemeine Definitionen reduzierten ISSA-Form sowie für die daran anschließende Finalisierung, welche auch hier das Pruning und die Ergänzung der allgemeinen Verwendungen umfasst.

Feld-insensitiv zeigen sich die verschiedenen Teile alle gleichermaßen als recht bedeutsam für die Gesamtlaufzeit. Die Gewichtung der Teile variiert dabei von Programm zu Programm. Wie wir schon bei der abstrakten Größe der Propagierungsschritte festgestellt hatten, gelingt fluss-insensitiv trotz kleinerem Graphen nicht automatisch eine deutlich schnellere Propagierung. Wir bemerken auch, dass die eigentliche Zeigeranalyse durchaus teurer sein kann als der abschließende Aufbau der Datenfluss-Darstellung. Nethack jedoch ist ein Beispiel für den umgekehrten Fall, so dass diesbezüglich keine klare Aussage getroffen werden kann.

Im feld-sensitiven Diagramm erkennen wir deutlich, dass die Zeigeranalyse im fluss-insensitiven Fall nur einen geringen Anteil an der Gesamtlaufzeit hat. Die Kosten dieser Variante entstehen danach, bei der Verarbeitung der Zeiger-Resultate für den Aufbau der ISSA-Form. Als Erklärung geben wir zu bedenken, dass der Constraint-Graph als zentrales Element während der Zeigeranalyse feld-sensitiv kaum größer wird. Man vergleiche hierzu die kaum gestiegene Zahl der Objekte, welche ja die Knoten dieses Graphen ausmachen. Daher bleiben die Kosten der Zeigeranalyse relativ niedrig, im Unterschied

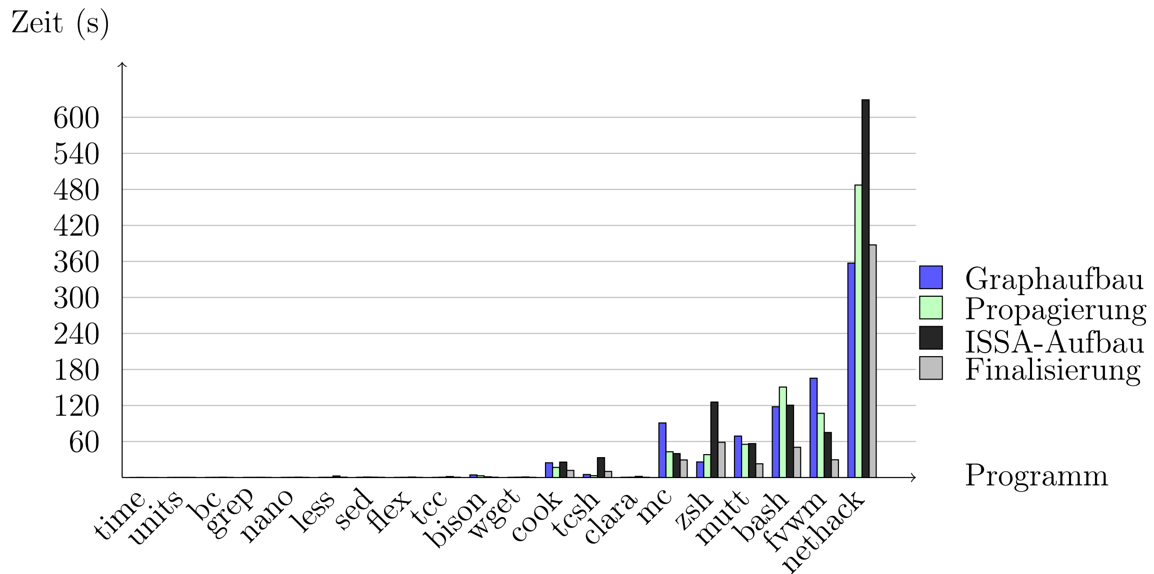


Abbildung 13.9: Laufzeit der einzelnen Analysen-Teile (fluss- und feld-insensitiv)

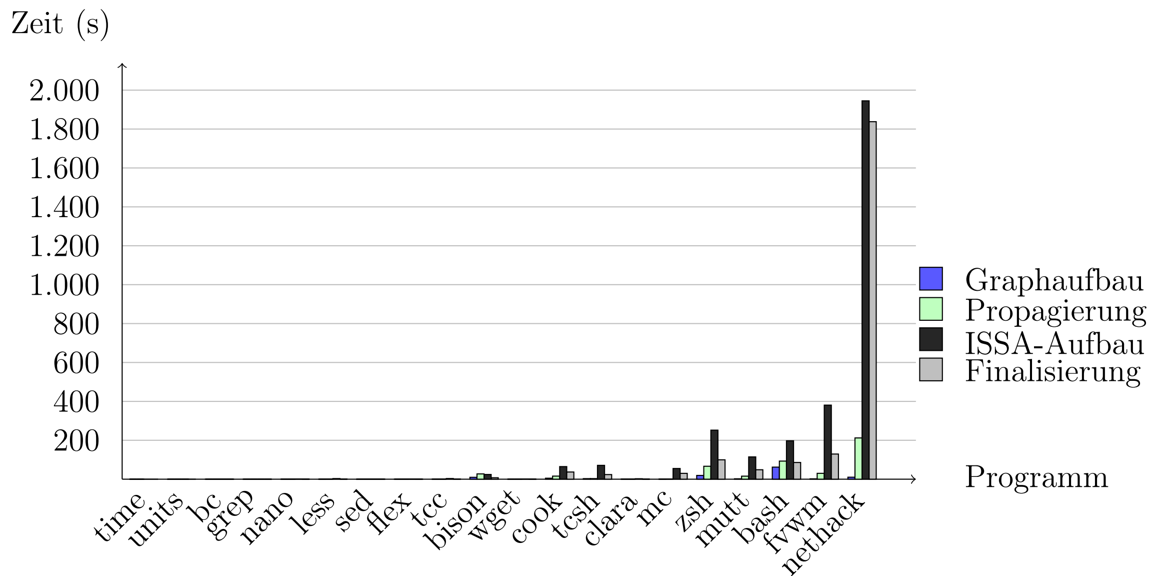


Abbildung 13.10: Laufzeit der einzelnen Analysen-Teile (fluss-insensitiv, feld-sensitiv)

zum fluss-sensitiven Szenario. Der ISSA-Aufbau am Ende jedoch spürt das oben schon erwähnte Phänomen des höheren Mengengerüsts für den fluss-sensitiven Graphaufbau.

13.2 Speicherbedarf

Für eine Bewertung der Skalierbarkeit betrachten wir in diesem Abschnitt noch den Platzbedarf der Analyse. Die Tabelle 13.3 bietet hierzu die Übersicht zu den sechs Präzisionsstufen. Wir sehen darin jeweils den maximalen gemessenen Platzbedarf.

Im Vergleich stellen wir mit der Betrachtung dieser Zahlen fest, dass der Platzbedarf fluss-sensitiv in der Regel leicht höher ist als fluss-insensitiv, wobei der fluss-insensitive Bedarf erst nach dem Fixpunkt entsteht. Zsh ist eine Ausnahme, bei der es sich umgekehrt verhält. Dieses Programm konnte von der höheren Präzision profitieren und war uns bereits aufgrund der dadurch reduzierten Zahl an Zeigerzielen aufgefallen. Mit indirekten starken Aktualisierungen jedoch gelingt zumindest bei den größten Programmen eine deutliche Platzersparnis. Das passt zu unserer Beobachtung des letzten Kapitels, dass dort der finale ISSA-Graph um einiges kleiner ist. Die höhere Präzision erreicht also eine Platzersparnis.

Anders dagegen die Feld-Sensitivität als Maßnahme zur Präzisionsverbesserung. Sie erhöht das Mengengerüst (Größe des ISSA-Graphen) deutlich, und damit nun auch den Platzbedarf. Auf einem typischen 32-Bit-System mit weniger als 3 GB Speicher pro Anwendung sind die größten beiden Programme fvwm und nethack damit nicht mehr analysierbar.

Auch beim Speicherbedarf interessieren wir uns wieder für den Trend bezogen auf die Größe der Programme. Dazu dienen die Diagramme 13.11 bis 13.14. Sie zeigen den Speicherbedarf für die fluss-sensitiven Analyse-Varianten. Deutlicher als bei der Laufzeit ist hier eine lineare Zunahme zu erkennen. Eine Ausnahme davon sehen wir auch diesmal in der feld-sensitiven Variante mit indirekten starken Aktualisierungen: Wie bei der Laufzeit ist hier der Trend etwas schlechter als in den übrigen Varianten. Der Trend ist damit erfreulich, die Absolutzahlen sind jedoch recht hoch.

	Feld-insensitiv			Feld-sensitiv		
Programm	FI	FS - FI	FS stark - FS	FI	FS - FI	FS stark - FS
time	46,11	1,98	0,07	52,26	1,98	-0,05
units	69,41	3,42	0,38	67,78	11,03	0,33
bc	89,81	4,22	0,56	90,52	10,14	0,71
grep	117,51	1,47	0,98	120,66	5,11	1,10
nano	114,98	5,71	1,27	119,63	5,23	-0,61
less	150,20	10,84	1,91	151,50	14,90	3,43
sed	158,11	6,14	1,06	164,85	8,24	1,44
flex	157,45	3,35	1,64	162,63	4,57	1,31
tcc	172,05	7,98	1,75	179,35	11,96	0,98
bison	261,93	-2,63	26,88	410,34	64,66	79,62
wget	254,52	-0,05	2,81	258,46	5,57	0,15
cook	392,60	59,72	12,02	760,52	66,94	26,07
tcsh	468,42	23,89	-4,01	692,67	24,25	-203,84
clara	378,88	5,52	3,41	379,66	6,71	3,82
mc	888,84	32,29	-112,69	999,41	172,64	-27,69
zsh	971,76	-152,19	18,70	1.538,46	-237,04	-194,19
mutt	904,57	27,66	-40,62	1.068,58	-136,60	194,44
bash	1.498,84	93,12	-269,12	1.908,02	-23,80	-287,17
fvwm	1.211,04	63,91	-147,34	3.659,02	35,64	-1.702,55
nethack	4.030,84	74,31	-901,37	9.402,29	184,01	-1.208,07

Tabelle 13.3: Maximaler Platzbedarf (MB)

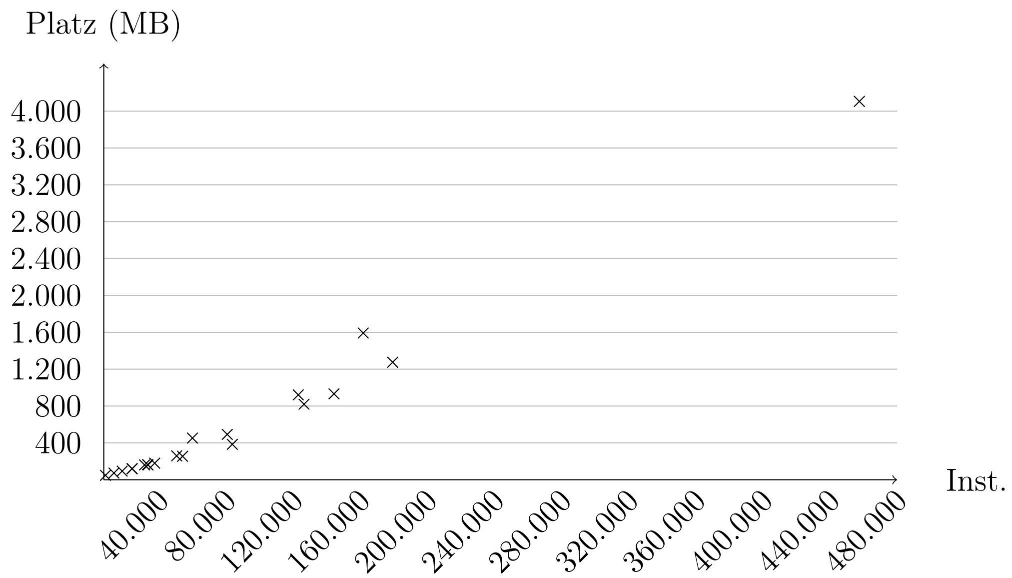


Abbildung 13.11: Trend beim Speicherbedarf (fluss-sensitiv, feld-insensitiv)

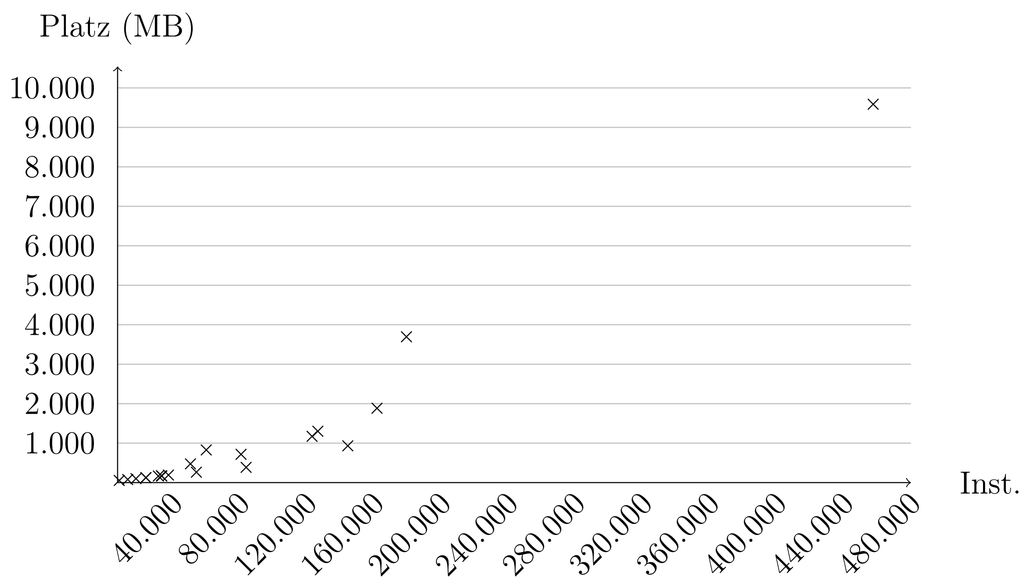


Abbildung 13.12: Trend beim Speicherbedarf (fluss- und feld-sensitiv)

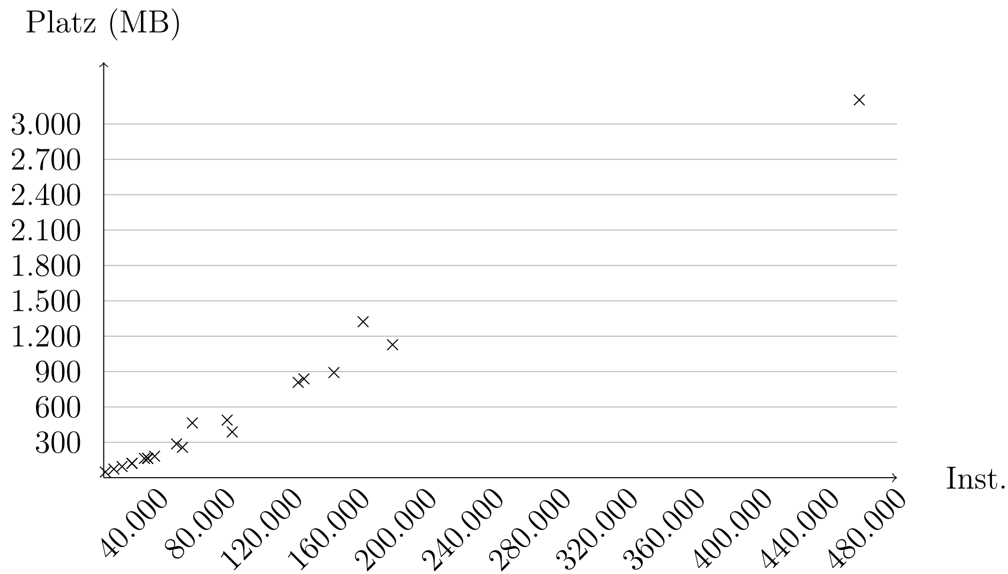


Abbildung 13.13: Trend beim Speicherbedarf (fluss-sensitiv, feld-insensitiv mit starken Akt.)

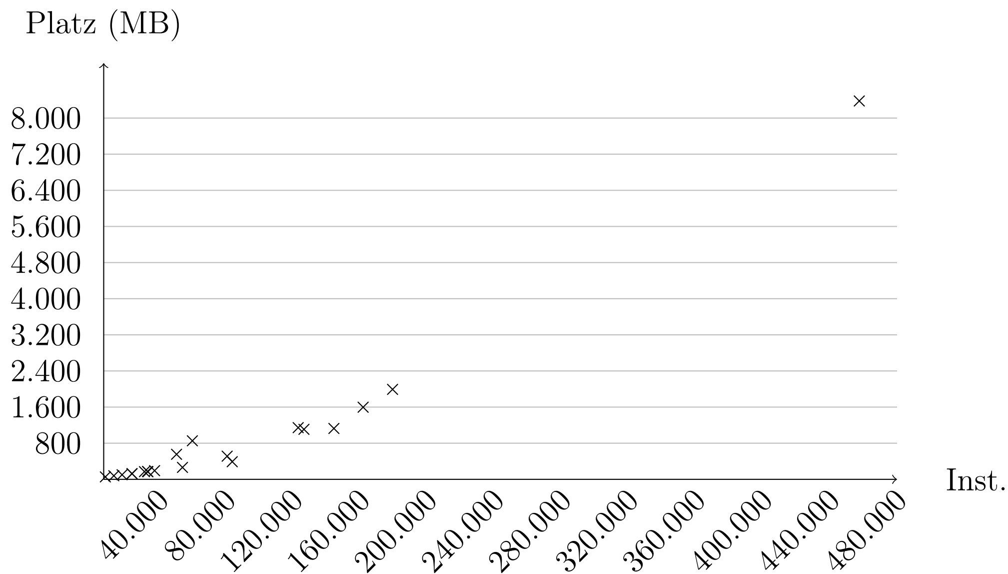


Abbildung 13.14: Trend beim Speicherbedarf (fluss- und feld-sensitiv mit starken Akt.)

13.3 Blockade-Graph und Relaxation

In diesem Abschnitt wollen wir die Analyse mit der Unterstützung für indirekte starke Aktualisierungen in ein paar Aspekten näher beleuchten. Wir interessieren uns dabei für die Kosten, die durch den Blockade-Graphen und die Relaxation im Fixpunkt entstehen, weswegen wir die Untersuchungen in diesem Kapitel angesiedelt haben.

Bezüglich der Größe des Blockade-Graphen zeigen uns die Diagramme 13.15 und 13.16 die Zahl der Knoten und Kanten des Graphen. Da wir nur noch eine der drei Fluss-Sensitivitäts-Stufen betrachten müssen, zeigt jedes Diagramm nun den feld-insensitiven und den feld-sensitiven Wert.

Wir erkennen eine moderate Zunahme der Knotenzahl mit zunehmender Programmgröße, wobei der Anstieg feld-sensitiv etwas steiler ausfällt. Die Absolutwerte sind aber vergleichsweise harmlos. Bei den Kanten ist die Zunahme ebenfalls gering, wobei nethack eine Ausnahme darstellt. Generell ist die Zahl der Kanten im Verhältnis zur Knotenzahl gering, oft nahe an eins oder gar darunter. Der Graph ist also sehr dünn und insgesamt nur moderat groß.

Um sowohl die Wirkung des Blockade-Graphen als auch den Platzbedarf für die blockierten Kanten einschätzen zu können, betrachten wir mit Abbildung 13.17 die Zahl der nach einem Graphaufbau-Schritt maximal blockierten Kanten (über den gesamten Blockade-Graphen aufsummiert). Hier zeigen sich feld-sensitiv hohe Spitzenwerte, doch das ist erklärbar: Die Feld-Sensitivität erlaubt mehr Blockaden, da feld-insensitiv die Strukturobjekte nie blockieren können. Außerdem ist die Zahl der indirekten Definitionen in einigen Fällen (z.B. bei nethack) höher als in der feld-insensitiven IR.

Mit höherer Präzision der Analyse steigt also der Effekt des Blockade-Graphen. Wie wir bei der Betrachtung des Speicherbedarfs jedoch gesehen haben, wirkt sich die hohe Zahl an maximal blockierten Knoten nicht spürbar auf den Platzbedarf der Analyse insgesamt aus.

Betrachten wir als nächstes, wie viele indirekte Definitionen im Fixpunkt noch kein Ziel erhalten haben und damit blockieren. Abbildung 13.18 zeigt uns diese Zahlen als Diagramm. Die größeren Programme scheinen auch zu

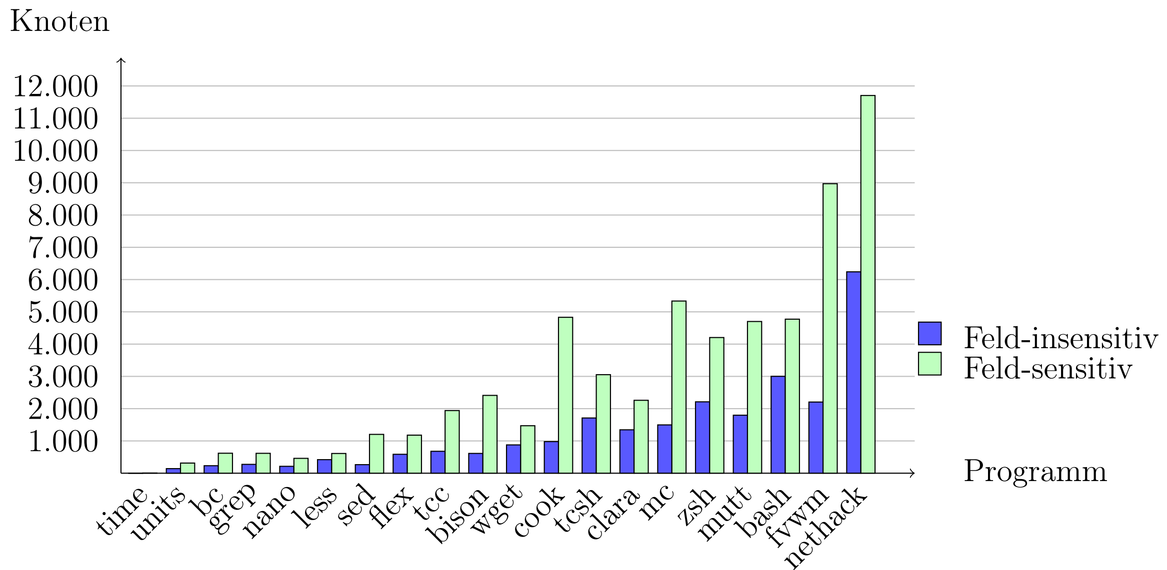


Abbildung 13.15: Knoten im initialen Blockade-Graphen

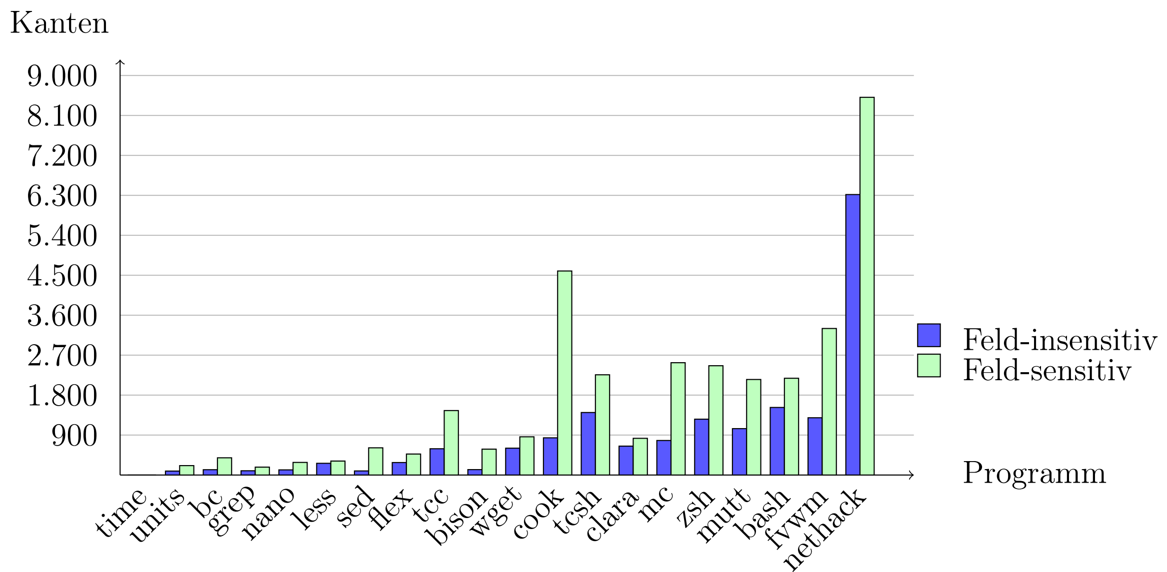


Abbildung 13.16: Kanten im initialen Blockade-Graphen

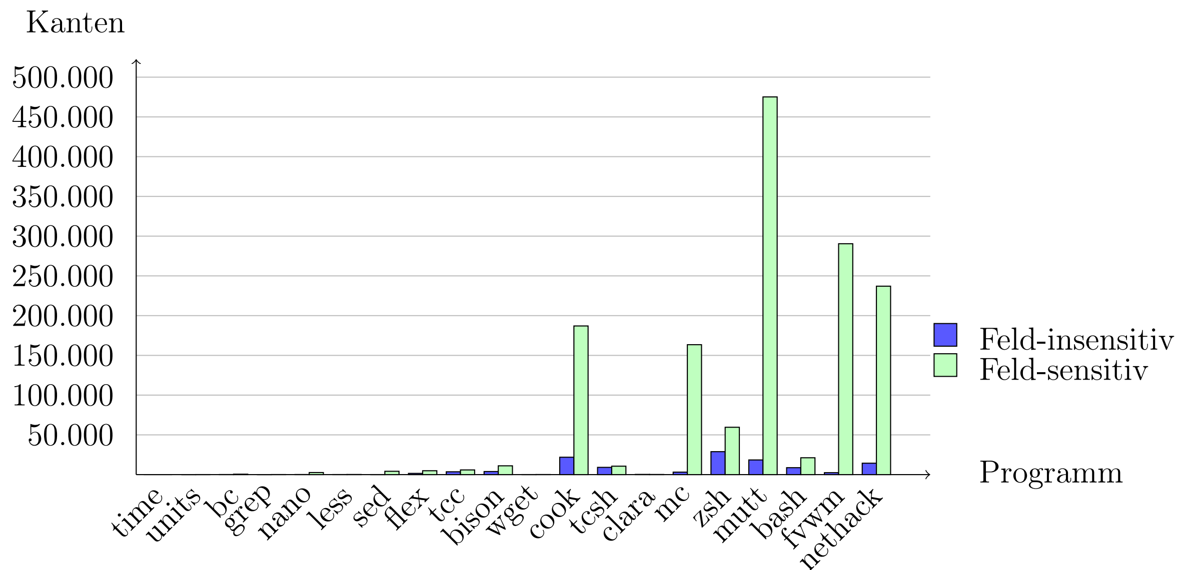


Abbildung 13.17: Maximal blockierte Kanten

mehr blockierenden Definitionen zu neigen, aber es zeichnet sich kein klarer Trend ab. Feld-sensitiv gibt es im Allgemeinen mehr Blockaden im Fixpunkt, aber mit bison erkennen wir auch eine deutliche Ausnahme. Weiterhin zeigt uns das Diagramm, dass auch kleinere Programme wie grep blockierende Definitionen im Fixpunkt aufweisen.

Diese Zahlen sprechen dafür, dass man sich Gedanken über den Umgang mit solchen Fällen machen sollte. Unsere Strategie hierzu bestand in der Relaxation und Fortsetzung der Analyse. Kommt die Analyse auch damit zum Fixpunkt, so stufen wir die verbleibenden blockierenden indirekten Definitionen als Fehler ein. Abbildung 13.19 zeigt uns, dass es davon noch einige gibt. Es stechen hier insbesondere die feld-insensitive Analyse von bison und die feld-sensitiven Analysen von mutt und nethack hervor. Verglichen mit dem vorigen Diagramm stellen wir aber auch fest, dass mit der Relaxation viele Blockaden verschwunden sind, z.B. bei fvwm. Die Relaxation als Strategie scheint damit durchaus wirkungsvoll. Weitere Arbeiten können nun näher betrachten, wie viele echte Programmierfehler damit erkannt oder verwischt wurden.

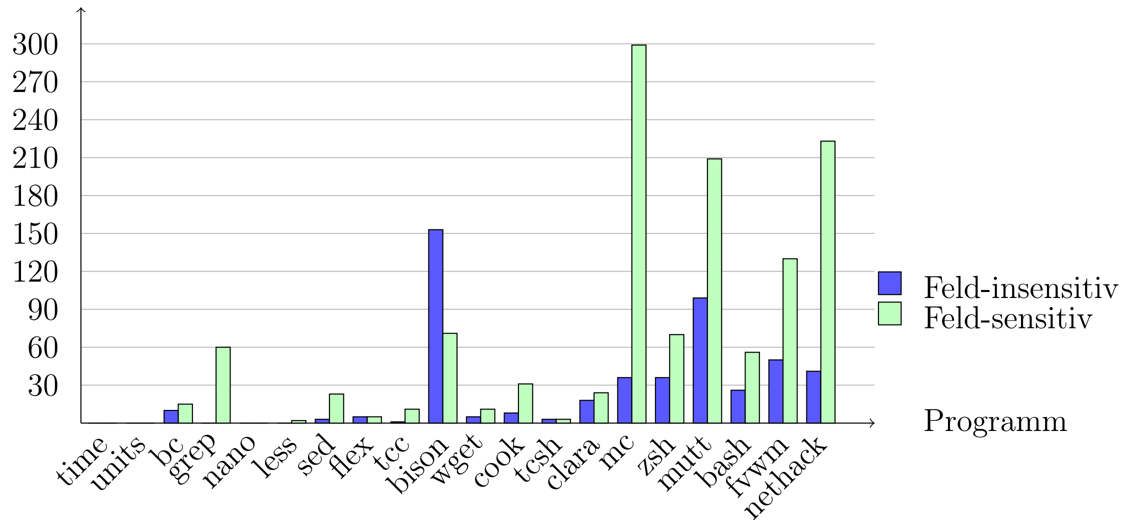


Abbildung 13.18: Blockierende indirekte Definitionen im Fixpunkt

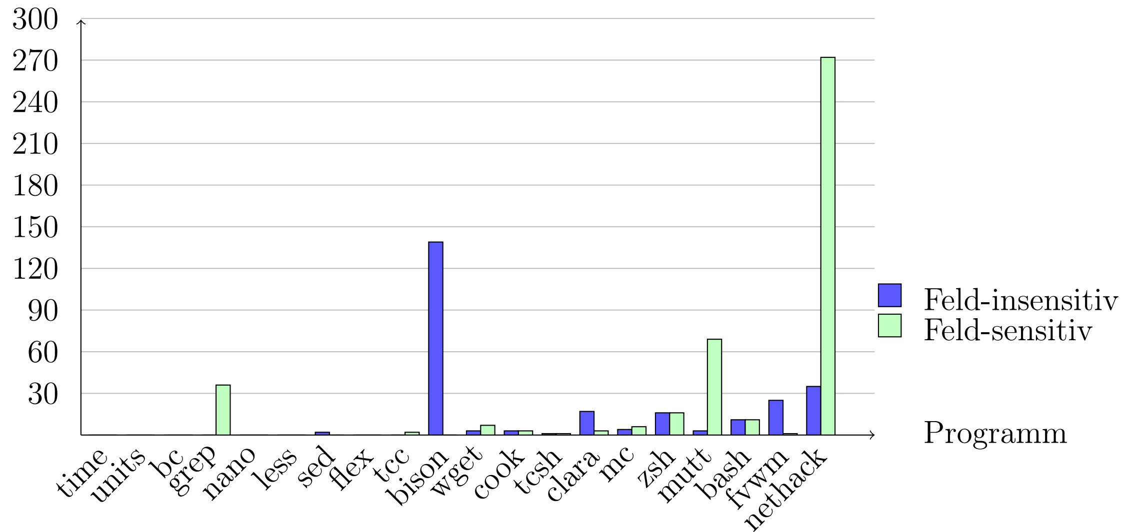


Abbildung 13.19: Final als Fehler eingestufte indirekte Definitionen

Schließlich beleuchten wir die Wirkung der Relaxation noch aus zwei anderen Aspekten. Diagramm 13.20 verdeutlicht, dass durch die Relaxation nicht nur die Zahl der blockierenden indirekten Definitionen sinkt, sondern umgekehrt in manchen Fällen auch noch einige Knoten für den reduzierten ISSA-Graphen ermittelt werden. Bei nethack scheint sich die Fortsetzung der Analyse in diesem Sinne zu lohnen, um die Folgen der blockierenden Definitionen zu mildern. Jedoch muss hier beachtet werden, dass die feld-sensitive Analyse dieses Programmes insgesamt rund 105 Millionen Knoten im reduzierten ISSA-Graphen berechnet. Im Vergleich dazu fällt der Effekt der Relaxation damit wieder gering aus. In anderen Fällen (z.B. bei bash) ergibt die Fortsetzung dagegen keine nennenswerten Veränderungen am Graphen.

Ein nachteiliger Effekt der Relaxation ist es, dass durch die Fortsetzung natürlich die Zahl der Iterationen ansteigt. Das Diagramm 13.21 verdeutlicht, dass die Zahl der Iterationen durchaus wie im Falle von bison sehr deutlich zunehmen kann. Abgesehen von diesem einem Extremfall ist der Effekt jedoch überschaubar. Zusammen mit der Feststellung, dass die Relaxation relativ wenig Knoten ergänzt, wirkt sie sich nur bei einer erschöpfenden Propagierung auf die Laufzeit aus.

Bei bison ist es so, dass im ersten Fixpunkt lediglich 23 blockierende indirekte Definitionen als Programmierfehler eingestuft werden. Die Fortsetzung der Analyse sorgt dann jedoch mit neuen indirekten Aufrufzielen dafür, dass weitere solche Definitionen hinzukommen. Dieses Spiel wiederholt sich mehrere Male, bis am Ende über hundert indirekte Definitionen ohne Zeigerziel vorliegen und als Fehler eingestuft werden. Bei den neuen Aufrufzielen sind jeweils einige fälschliche Ziele dabei, wodurch die Parameter keine Zeigerziele erhalten und daher zu solchen Definitionen führen. Die Relaxation bringt damit in diesem Sinne unglücklich überschätzte Zeigerziele zustande, die ihrerseits für eine weitere Verschlechterung sorgen. Hier sollte es helfen, die Strategie so zu ändern, dass indirekte Definitionen, die seit dem letzten Fixpunkt neu hinzukamen und ohne Zeigerziel vorliegen, nicht als Fehler im Sinne der Relaxation zählen. Weiterhin vermuten wir, dass das Problem in typsicheren Sprachen insofern verschwindet, dass die Beachtung der Typen falsche Aufrufziele mit ihren Folgen verhindert.

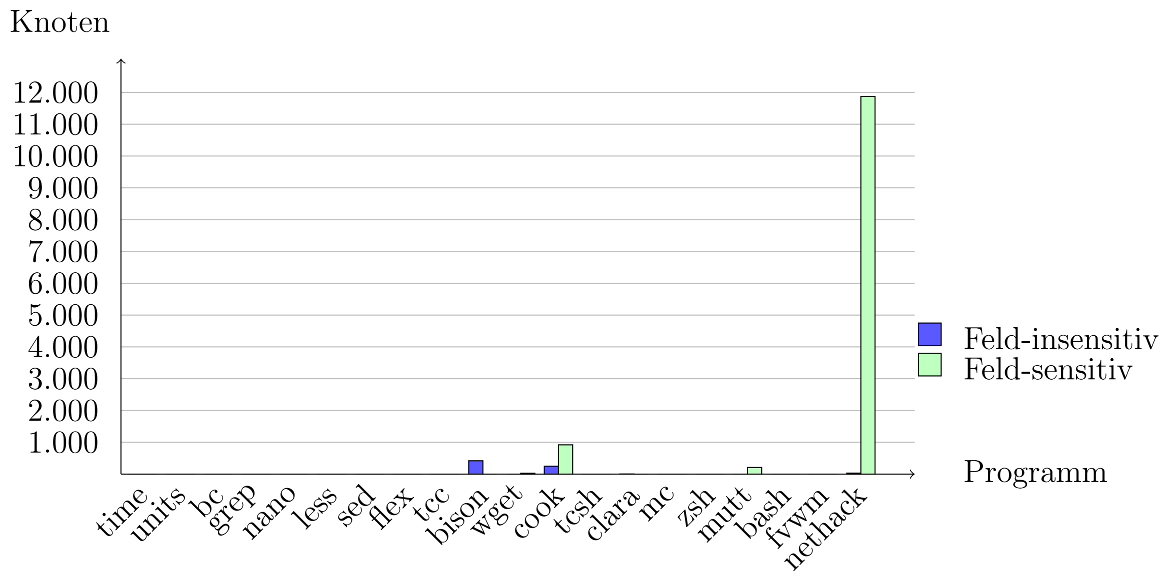


Abbildung 13.20: Effekt der Relaxation: Zusätzliche allgemeine Definitionen nach dem (ersten) Fixpunkt

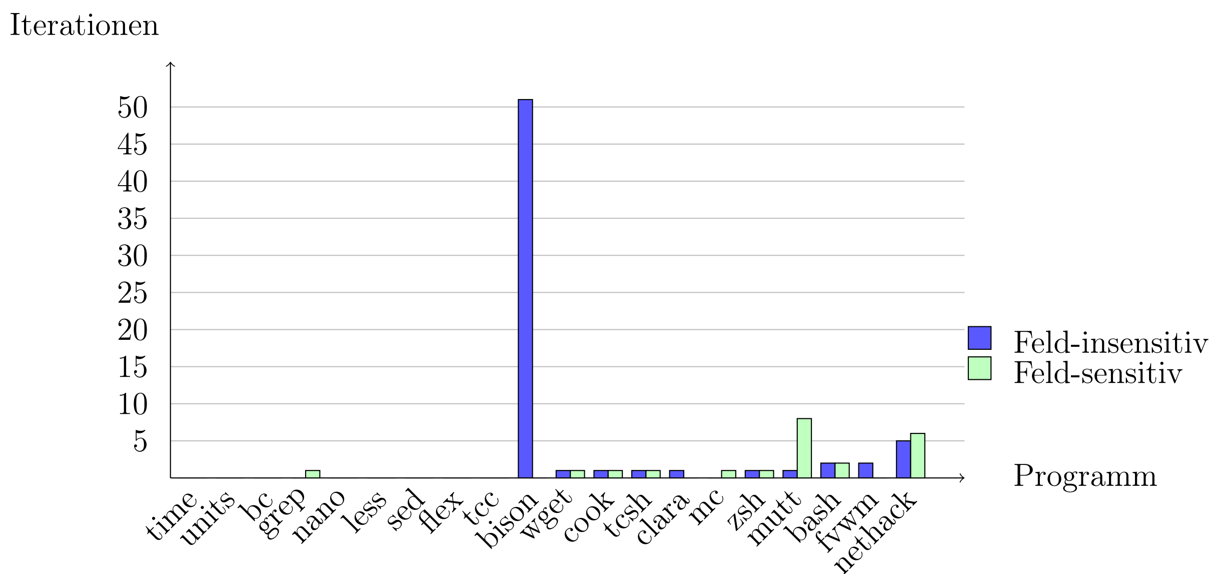


Abbildung 13.21: Effekt der Relaxation: Zusätzlich benötigte Iterationen

13.4 Vergleich der Implementierungsvarianten

Die Effizienz haben wir bislang für unsere schnellste Implementierung betrachtet: Mit inkrementeller Zyklenkontraktion und Rückwärtspropagierung auf dem SCC-DAG. Nun wollen wir noch erläutern, wie sich die anderen Varianten dazu vergleichen. Im Einzelnen vergleichen wir die Effizienz mit den folgenden weiteren Ausprägungen der Analyse:

- Mit inkrementeller Zyklenkontraktion, aber Vorwärts-Propagierung. Dieser Vergleich zeigt exemplarisch den Unterschied zwischen Vorwärts- und Rückwärts-Propagierung.

- Mit erschöpfender Zyklenkontraktion und Rückwärts-Propagierung. Dieser Vergleich zeigt den Nutzen unserer inkrementellen Zyklenkontration im Unterschied zu einer wiederholten vollständigen Zyklenkontraktion.

- Ohne Zyklenkontraktion, mit inkrementeller Vorwärts-Propagierung. Damit vermessen wir die Ausprägung, welche in der theoretischen Analyse die kubische Komplexität ergeben hat. Wir lernen daraus auch über den praktischen Nutzen der Zyklenkontraktion.

- Ohne Zyklenkontraktion, mit erschöpfender Vorwärts-Propagierung. Mit diesem Vergleich untersuchen wir den Nutzen der inkrementellen Propagierung im Unterschied zur erschöpfenden Realisierung.

Die Vergleiche betrachten die vier fluss-sensitiven Varianten der Analyse (mit oder ohne indirekte starke Aktualisierungen, jeweils feld-sensitiv oder nicht), da uns die Fluss-Insensitivität in dieser Arbeit nur für Vergleichszwecke interessiert. Für einen übersichtlichen Vergleich präsentieren wir wieder jeweils zwei Diagramme auf einer Seite, je eines für den feld-insensitiven bzw. feld-sensitiven Vergleich. Die dabei weniger gut erkennbaren kleinen Programme sind für den Vergleich nicht so relevant und können außerdem in der zusammenfassenden Tabelle in Abschnitt 13.5 betrachtet werden. Zusätzlich führen wir im inkrementellen Fall ohne Zyklenkontraktion den Trend bezüglich der Programmgröße separat dazu in einem Schaubild an.

13.4.1 Vorwärts-Propagierung

Die Abbildungen 13.22 und 13.23 vergleichen die Auswirkungen der Wahl der Propagierungsrichtung auf die Laufzeit. Beide Versionen nutzen den jeweils inkrementell berechneten SCC-DAG als Basis. Die Legendenbezeichnung „Referenz“ bezieht sich hier wie in den folgenden Abschnitten auf die Referenz-Variante mit Rückwärts-Propagierung. Die damit verglichene Variante benennt die Legende jeweils als „Andere Variante“.

Trotz der gleichen Basis zeigen sich bei den großen Programmen deutliche Unterschiede. Die Rückwärts-Propagierung stellt sich da als klar schneller heraus, was unsere Überlegungen aus Kapitel 6 stützt. Dort hatten wir schon vermutet, dass die weniger intuitive Vorgehensweise der Propagierung von Dereferenzierungen zu Zeigerziel-Quellen Vorteile hat. Dies sehen wir nun auch empirisch bestätigt. Da auch die abstraktere Größe der Zahl der Propagierungs-Schritte wächst (nicht gezeigt), liegen die Laufzeit-Unterschiede tatsächlich im Algorithmus und nicht an Schwankungen in der Testumgebung. Feld-insensitiv ist der Vorteil der Rückwärts-Propagierung tendenziell größer, jedoch nicht immer (bison ist wieder eine Ausnahme).

Im Unterschied zu den deutlichen Laufzeit-Differenzen ist der Platzbedarf der beiden Varianten fast identisch. Wir verzichten daher auf die Angabe der Diagramme. Zusammenfassend ergibt der Vergleich, dass die Rückwärts-Propagierung zu bevorzugen ist.

13.4.2 Erschöpfende Zyklenkontraktion

Als nächstes ermitteln wir den Nutzen der inkrementellen Zyklenkontraktion gegenüber der erschöpfenden Zyklenkontraktion. Für die Messungen haben wir jeweils die Rückwärts-Propagierung auf dem so errichteten SCC-DAG genutzt. Die Diagramme 13.24 und 13.25 zeigen den Laufzeit-Vergleich.

Auch in diesem Vergleich zeigt sich eine deutliche Differenz. Die Wiederverwendung des SCC-DAG aus der vorigen Iteration bringt mit steigender Programmgröße zunehmend deutlichere Geschwindigkeitsvorteile. Die Aufbewahrung des alten SCC-DAG resultiert in einem geringfügig erhöhten Speicherbedarf, wie uns die Abbildungen 13.26 und 13.27 nahelegen.

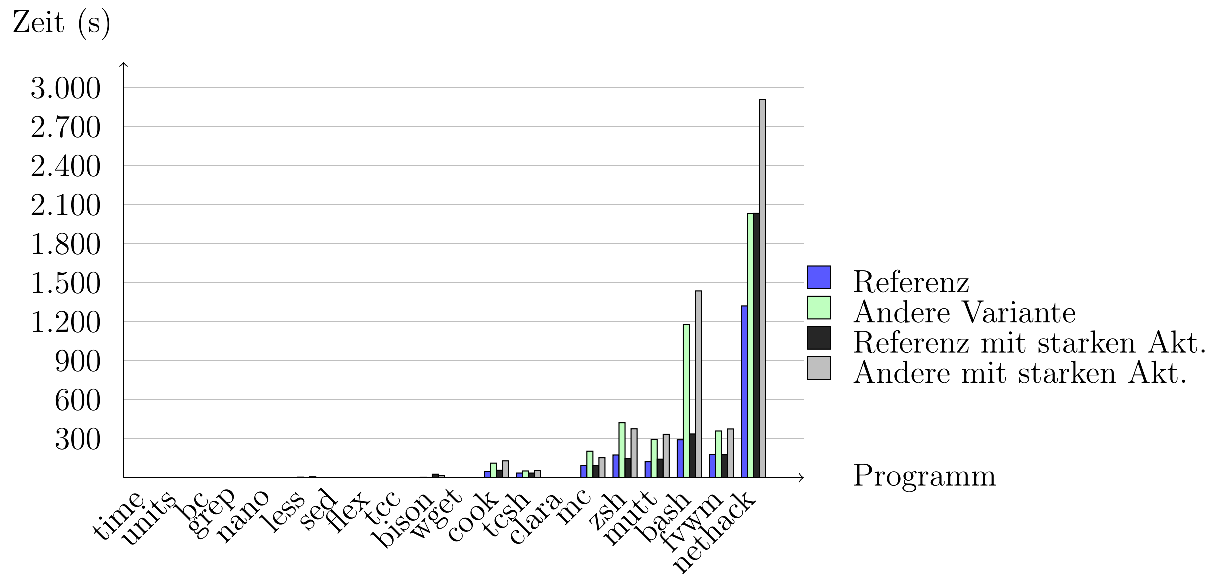


Abbildung 13.22: Laufzeit-Vergleich zur Propagierungsrichtung (feld-insensitiv)

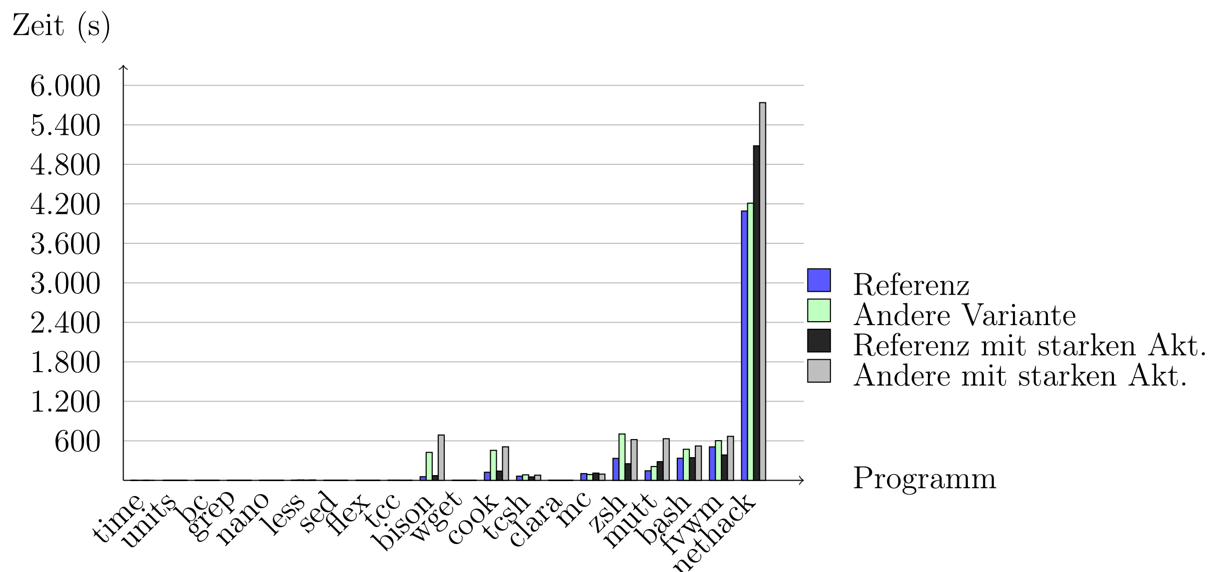


Abbildung 13.23: Laufzeit-Vergleich zur Propagierungsrichtung (feld-sensitiv)

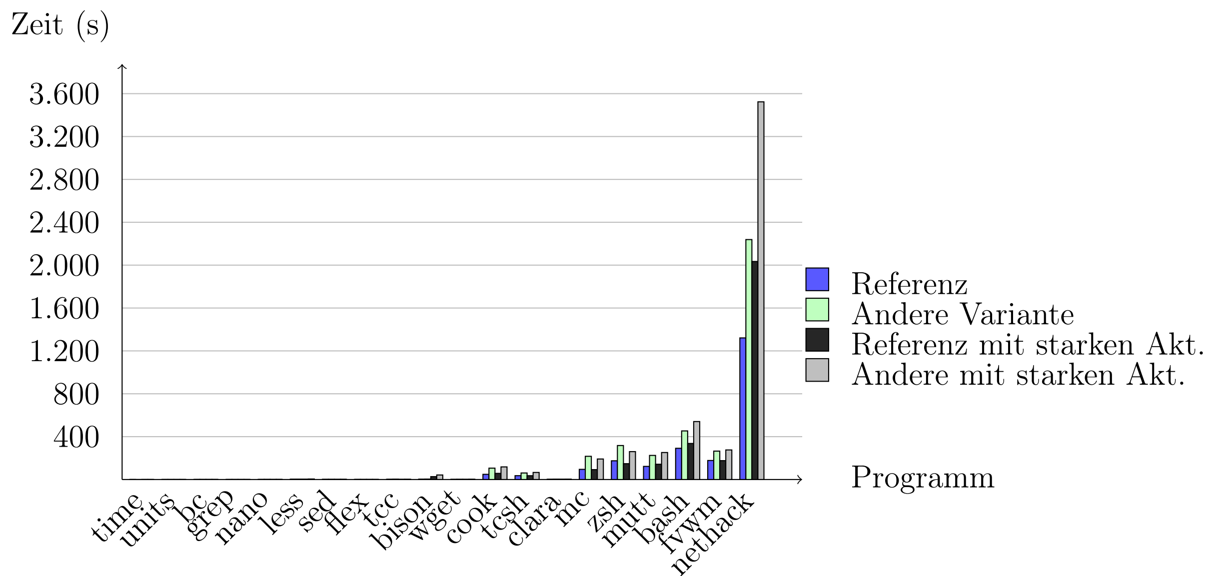


Abbildung 13.24: Laufzeit-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-insensitiv)

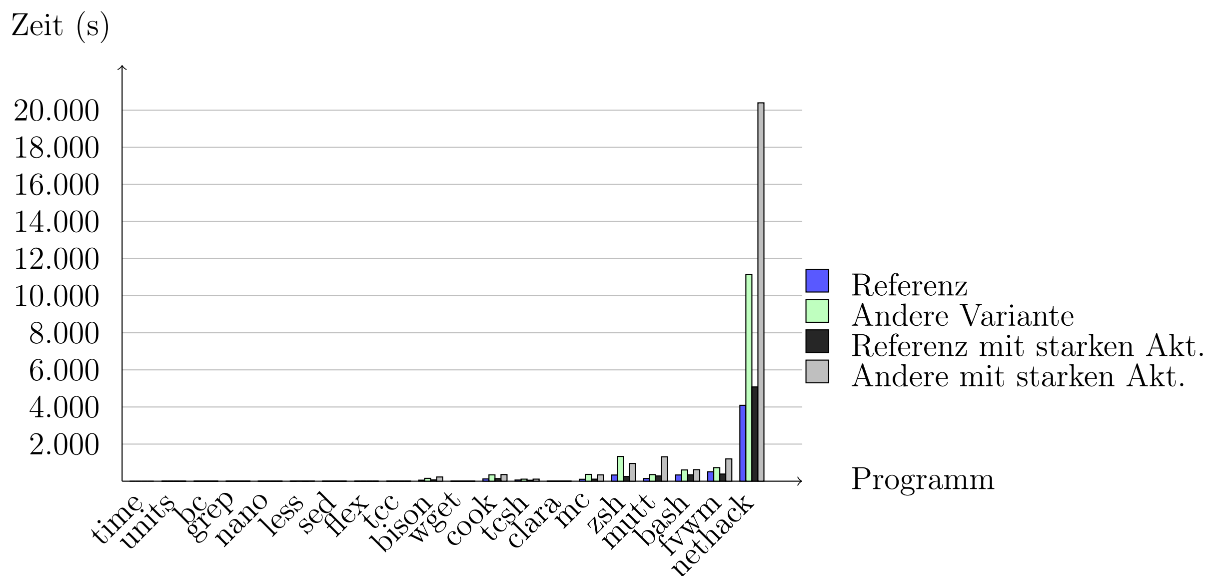


Abbildung 13.25: Laufzeit-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-sensitiv)

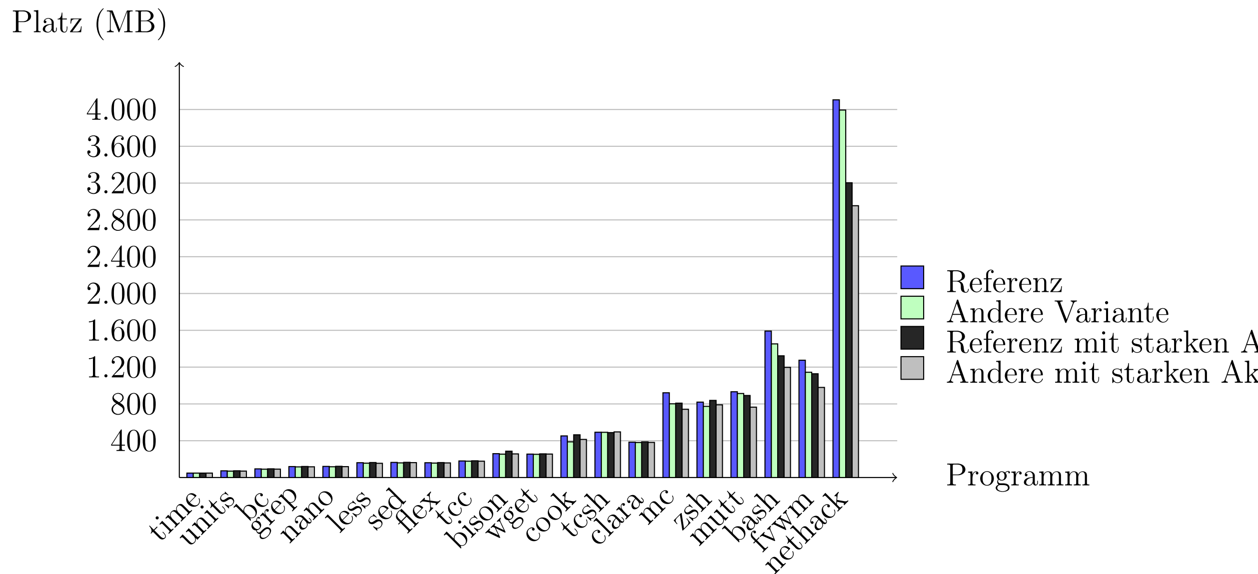


Abbildung 13.26: Speicherbedarf-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-insensitiv)

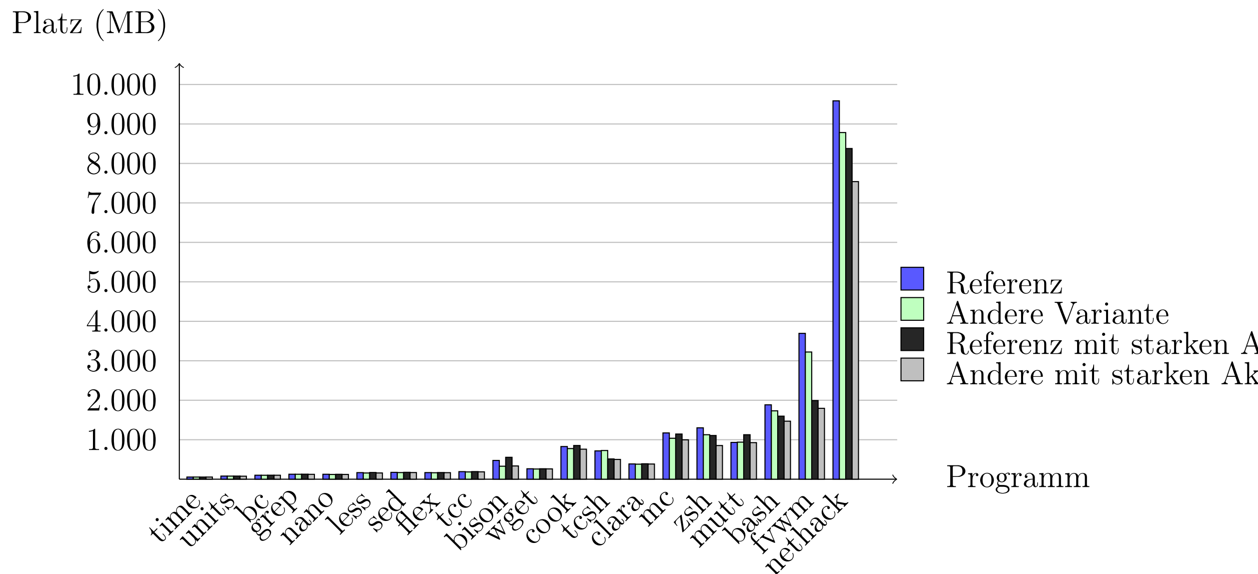


Abbildung 13.27: Speicherbedarf-Vergleich inkrementelle und erschöpfende Zyklenkontraktion (feld-sensitiv)

13.4.3 Inkrementell ohne Zyklenkontraktion

In diesem Unterabschnitt betrachten wir die Ausprägung der Analyse, für welche wir in der theoretischen Untersuchung die kubische Komplexität ermittelt haben. Es handelt sich dabei um die inkrementelle Propagierung ohne Zyklenkontraktion, welche sich zur Beschleunigung die Zeigerziele auch an den zwischendurch besuchten allgemeinen Definitionen abspeichert.

Die Abbildungen 13.28 und 13.29 vergleichen die Laufzeit dieser Analyse mit unserer effizientesten Version. Darin wird deutlich, dass die Zyklenkontraktion den Zeitbedarf um eine Größenordnung reduzieren kann (wohlgemerkt bei erschöpfender Propagierung auf dem SCC-DAG, d.h. mit inkrementeller Propagierung in Kombination mit inkrementeller Zyklenkontraktion könnte der Unterschied noch deutlicher ausfallen). Angesichts der im vorigen Kapitel präsentierten hohen Zahl an Knoten, die in den SCCs zusammengezogen werden, verwundert dies nicht.

Neben diesem Vergleich interessiert uns parallel zum theoretischen Resultat auch der empirische Trend bezüglich der Programmgröße. Diesen zeigen die vier Schaubilder 13.30 bis 13.33. Der Trend ist nur wenig schlechter als bei der effizientesten Variante. Die theoretisch ermittelte kubische Schranke scheint nicht ausgereizt zu werden.

Das Programm zsh erscheint feld-insensitiv als deutlicher negativer Ausreißer mit über 9000 Sekunden Laufzeit. Hier wurden auch mehr als 2^{31} Tupel (allgemeine Definition, Zeigerziel) angelegt, d.h. ohne Zyklenkontraktion müssen hier deutlich mehr Schritte in der Propagierung erfolgen.

Die Speicherkosten dieser Variante sind nicht zu vernachlässigen. Die Diagramme 13.34 und 13.35 präsentieren dazu den Vergleich des Platzbedarfs zu unserer Referenz-Variante. Die Mehrkosten, die wir daraus ablesen, entstehen durch die Tupel (allgemeine Definition, Zeigerziel), welche für die inkrementelle Realisierung angelegt werden. Deren hohe Zahl sorgt dafür, dass sich der Platzbedarf bei den großen Programmen verdoppelt und feld-insensitiv bei bash und nethack, feld-sensitiv bei den beiden größten Programmen sogar noch höher ausfällt. Diese hohen Werte dürften in der Praxis die Anwendung des inkrementellen Ansatzes begrenzen.

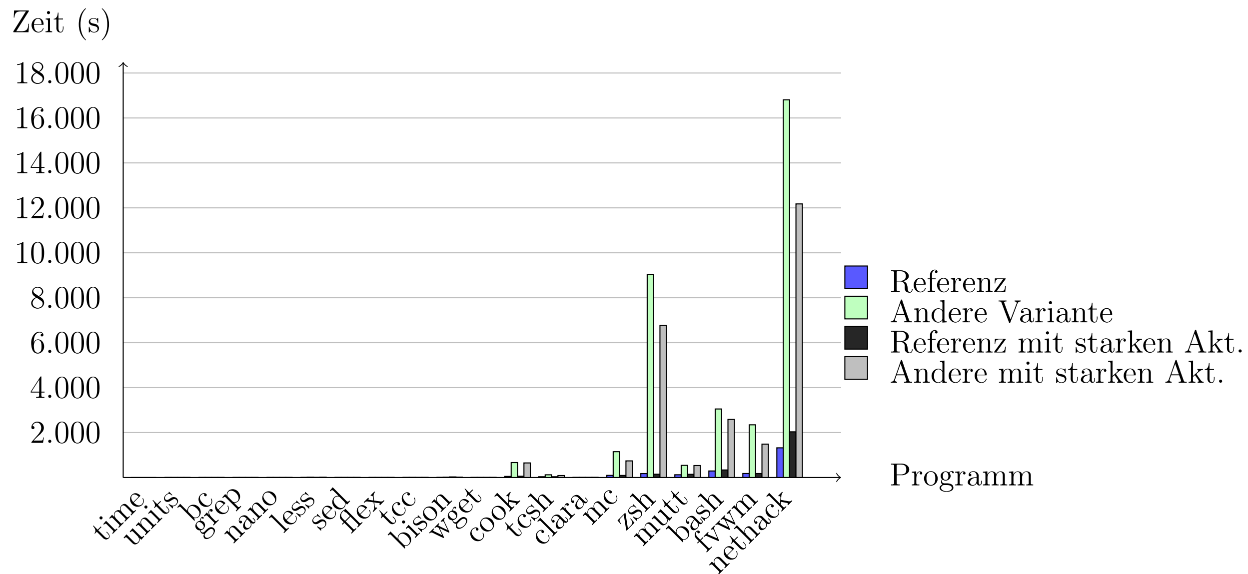


Abbildung 13.28: Laufzeit-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-insensitiv)

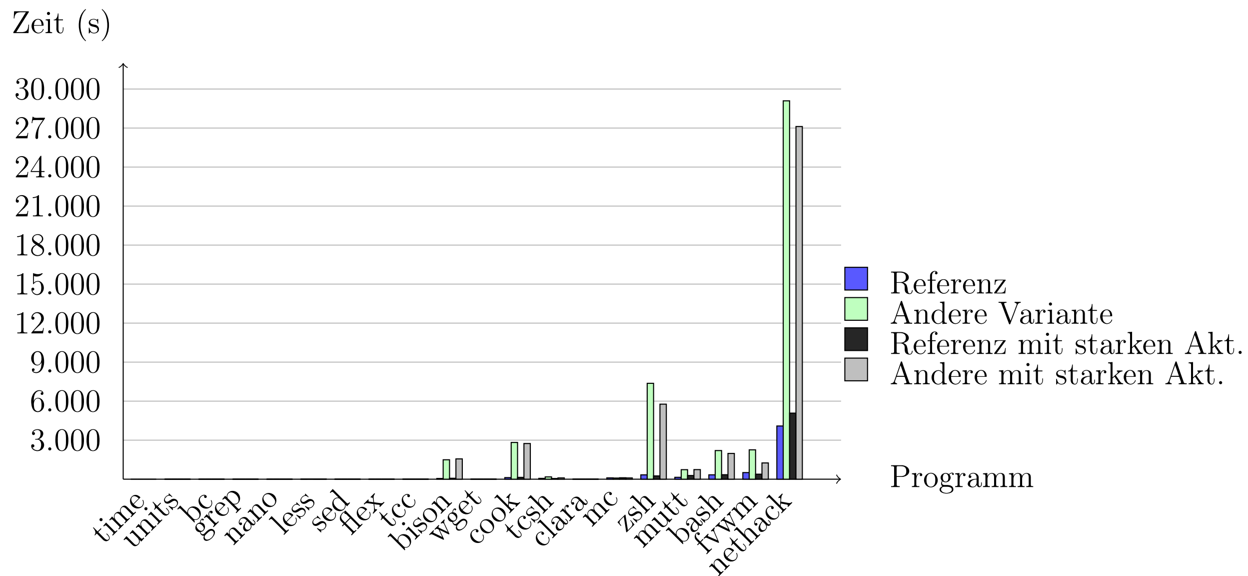


Abbildung 13.29: Laufzeit-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-sensitiv)

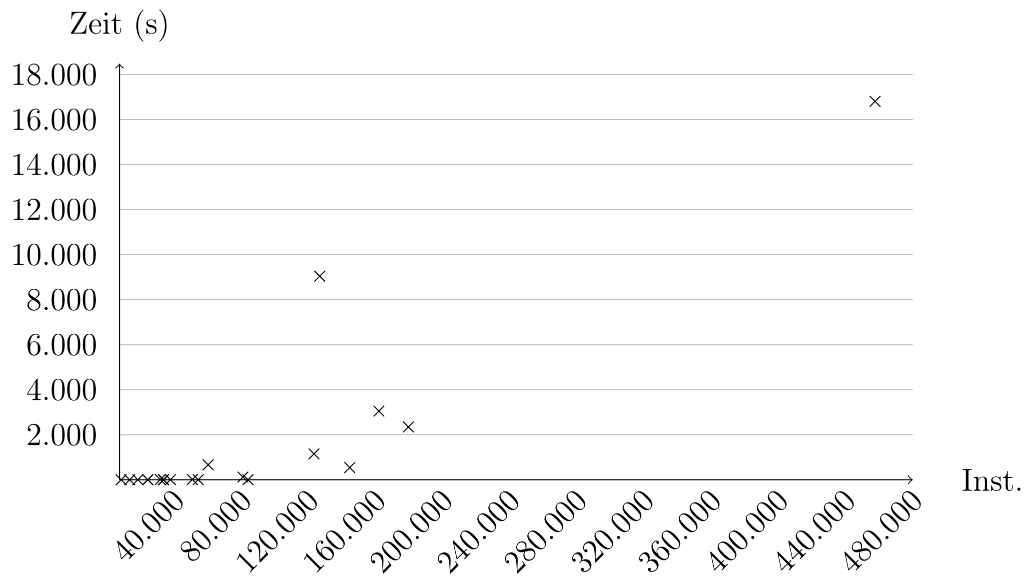


Abbildung 13.30: Laufzeit-Trend (feld-insensitiv, inkrementell ohne Zyklen)

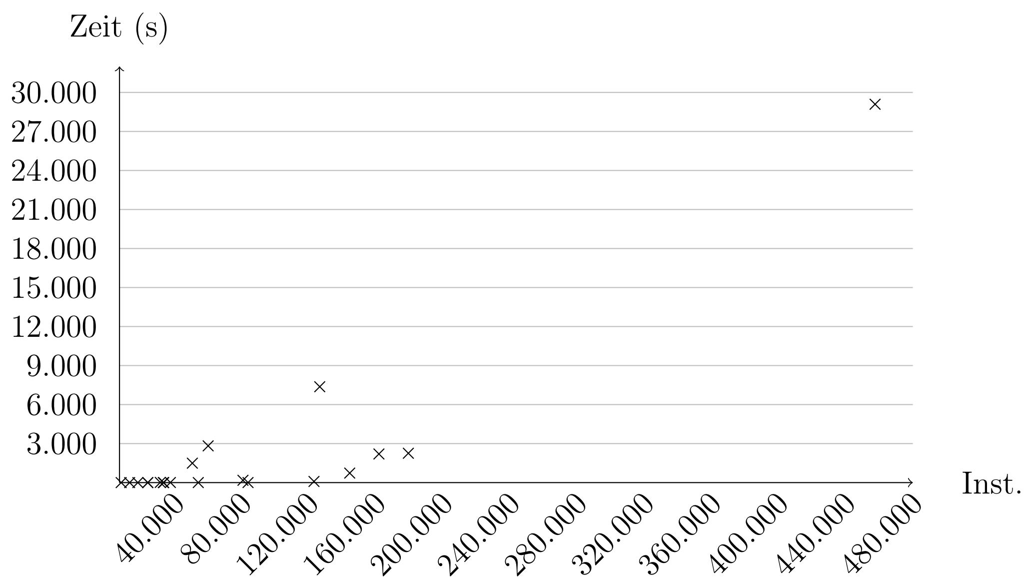


Abbildung 13.31: Laufzeit-Trend (feld-sensitiv, inkrementell ohne Zyklen)

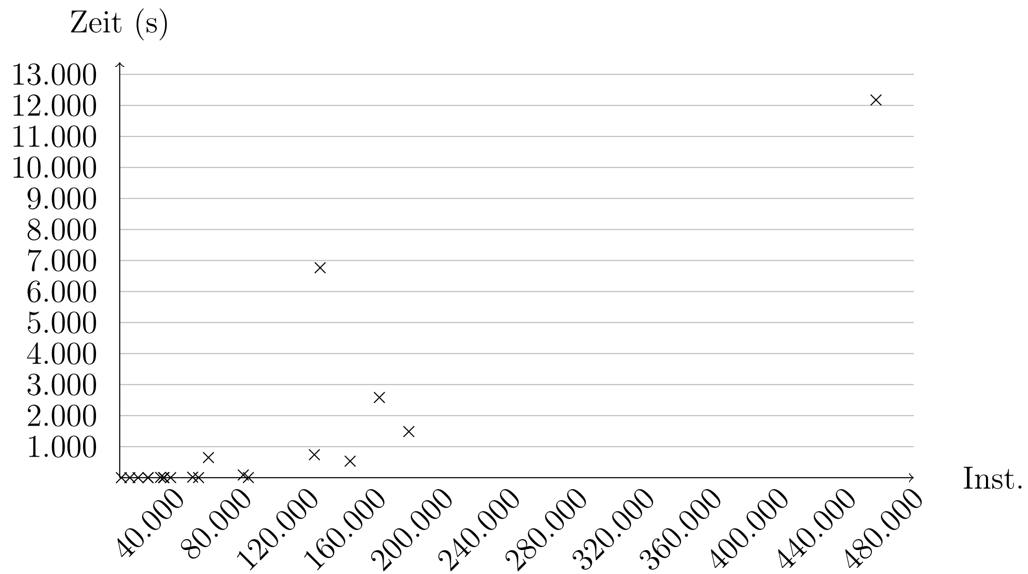


Abbildung 13.32: Laufzeit-Trend (feld-insensitiv mit starken Akt., inkrementell ohne Zyklen)

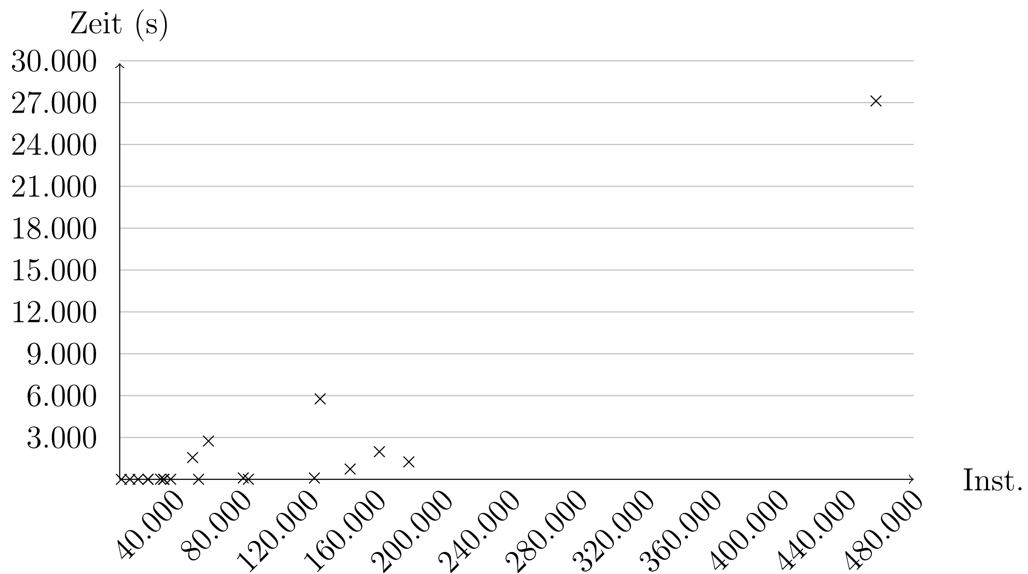


Abbildung 13.33: Laufzeit-Trend (feld-sensitiv mit starken Akt., inkrementell ohne Zyklen)

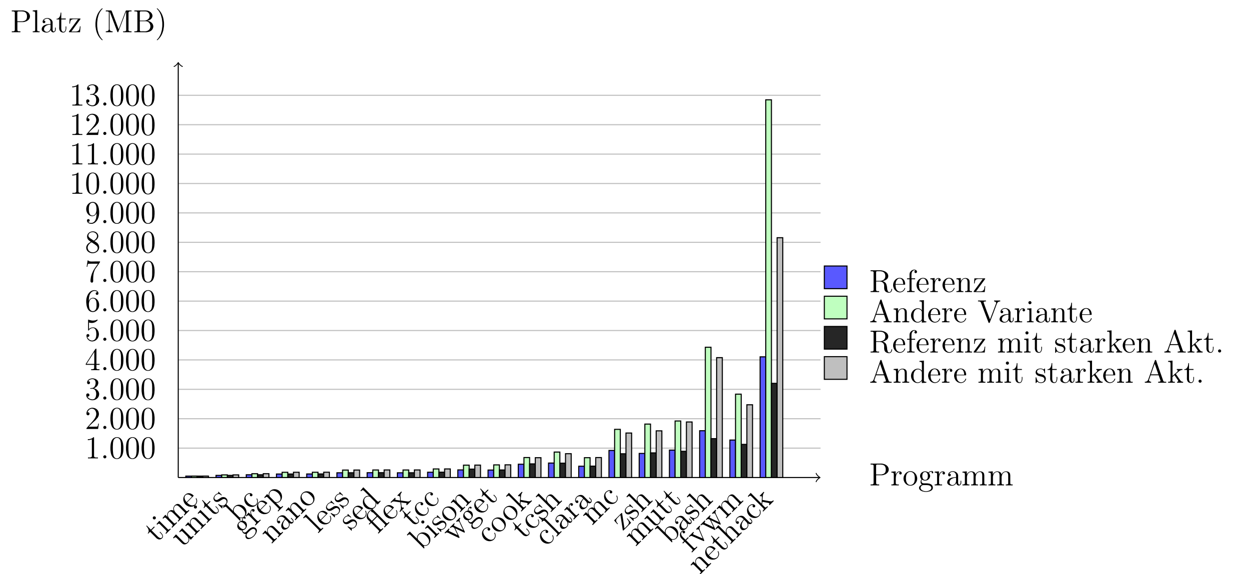


Abbildung 13.34: Speicherbedarf-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-insensitiv)

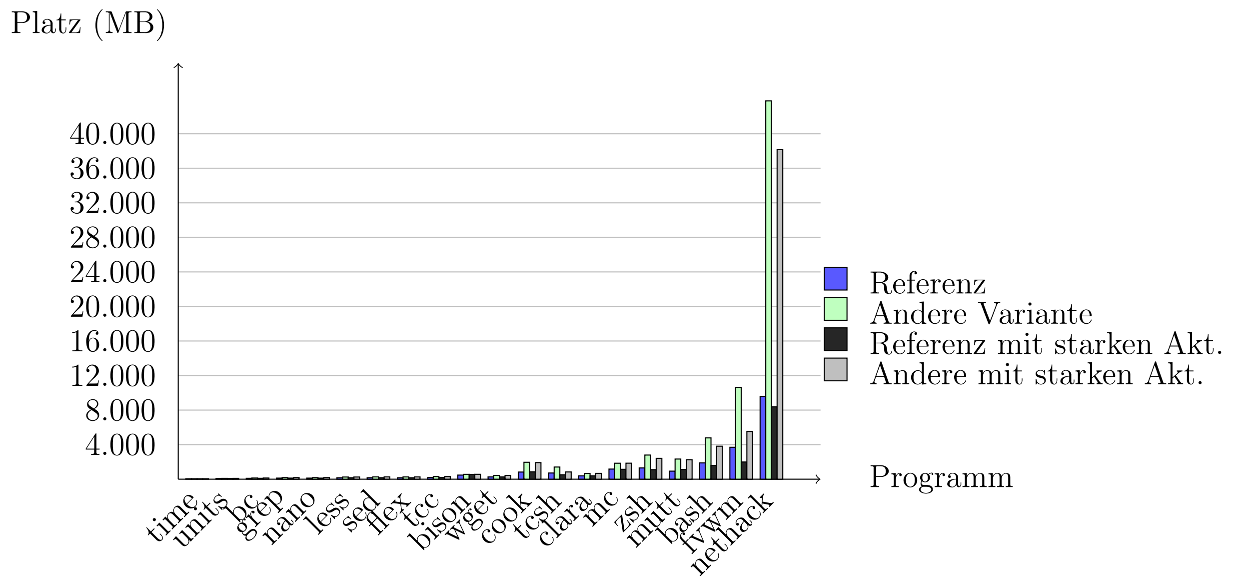


Abbildung 13.35: Speicherbedarf-Vergleich zur inkrementellen Propagierung ohne Zyklenkontraktion (feld-sensitiv)

13.4.4 Erschöpfende versus inkrementelle Propagierung

Schließlich wollen wir noch herausarbeiten, welchen Nutzen die inkrementelle Propagierung gegenüber der erschöpfenden Propagierung hat. Diesen Vergleich haben wir ohne eine Zyklenkontraktion und mit Vorwärts-Propagierung durchgeführt. Die Resultate für die Laufzeit sind in den Diagrammen 13.36 und 13.37 zu sehen.

Darin erkennen wir, dass die erschöpfende Propagierung noch einmal eine Größenordnung langsamer als die inkrementelle Propagierung ist. Bei nethack waren die beiden Varianten mit indirekten starken Aktualisierungen so langsam, dass wir sie abgebrochen haben, nachdem sie bereits deutlich länger liefen als die entsprechenden Varianten ohne indirekte starke Aktualisierungen. Die Diagramme zeigen dies mit „n/a“ an. Gegenüber unserer effizientesten Variante ist die hier zu erkennende Laufzeit um ein Vielfaches langsamer. Das gilt auch im Vergleich zur Vorwärts-Propagierung anstelle der Rückwärts-Propagierung auf dem inkrementell errichteten SCC-DAG. Das bedeutet, dass die Zyklenkontraktion derzeit das wesentliche Element dafür ist, eine akzeptable Laufzeit der kombinierten Analyse zu erreichen. Im Vergleich mit dem vorigen Abschnitt stellen wir weiterhin fest, dass auch das inkrementelle Vorgehen deutliche Gewinne bringt. Je nach Programm und Zahl der Iterationen (als besonders deutlichem Einfluss-Faktor für die Laufzeit der erschöpfenden Propagierung) fallen diese unterschiedlich hoch aus. Das lässt hoffen, dass die von uns noch nicht umgesetzte inkrementelle Propagierung auf dem inkrementell konstruierten SCC-DAG die Bestzeiten der Analyse noch einmal verbessert.

Die Diagramme 13.38 und 13.39 schließen den Vergleich ab mit der Betrachtung des Speicherbedarfs. (Auch hier gilt wieder das oben Gesagte bezüglich der letzten beiden Varianten bei nethack.) Hier ist klar, dass der Verzicht auf den inkrementell konstruierten SCC-DAG zu einer Platzreduktion führt. Wie wir sehen können, ist dieser jedoch vernachlässigbar gering, so dass die Zyklenkontraktion auch diesbezüglich keinen wirklichen Nachteil besitzt.

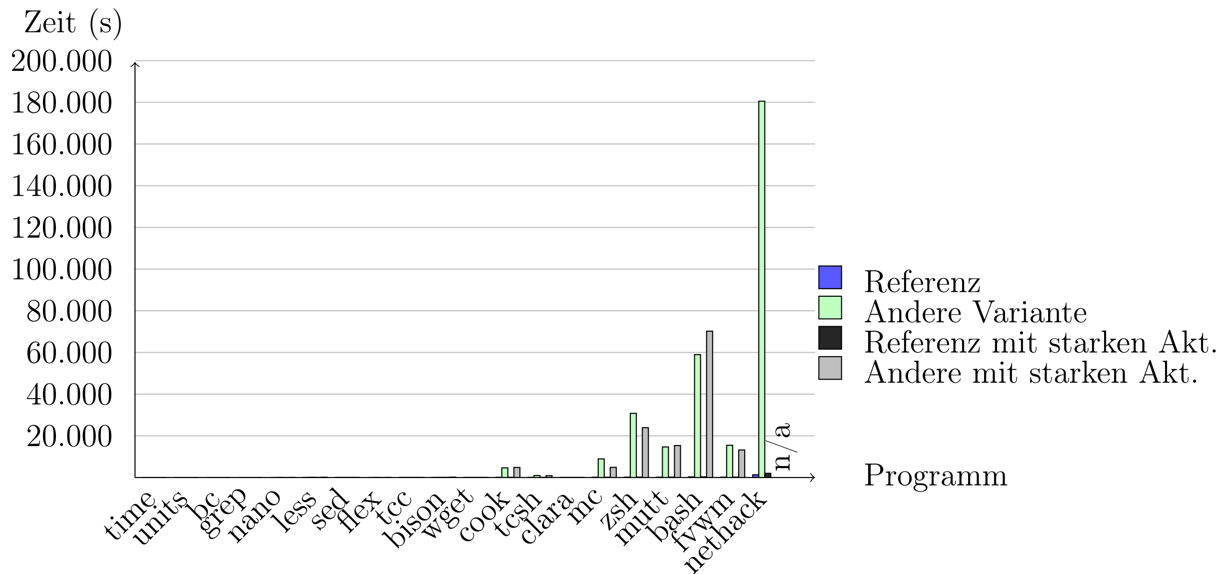


Abbildung 13.36: Laufzeit-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-insensitiv)

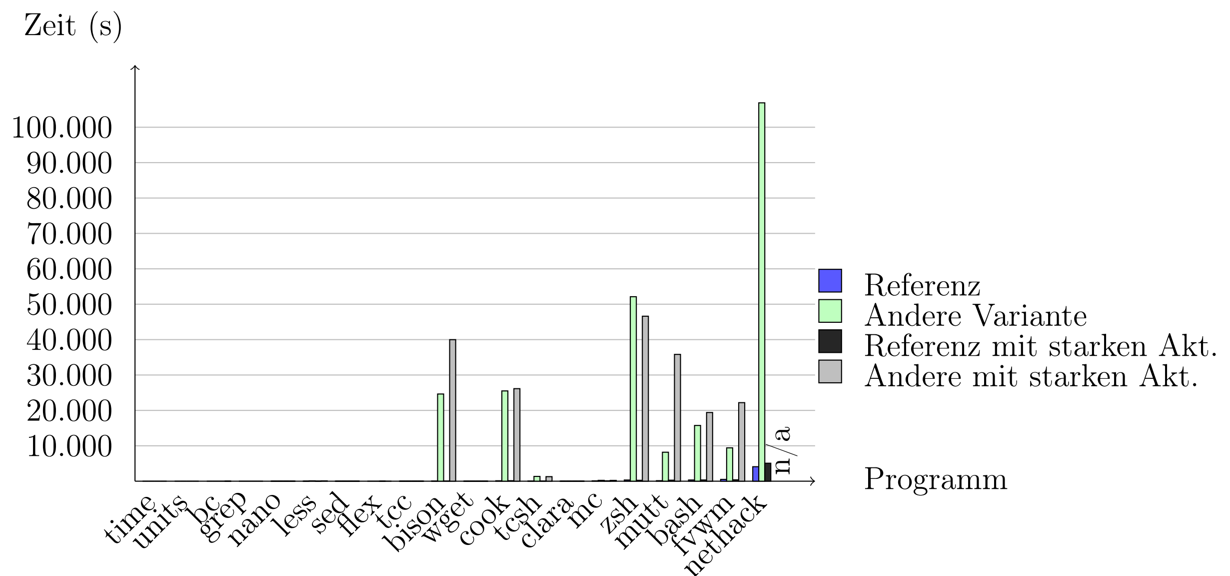


Abbildung 13.37: Laufzeit-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-sensitiv)

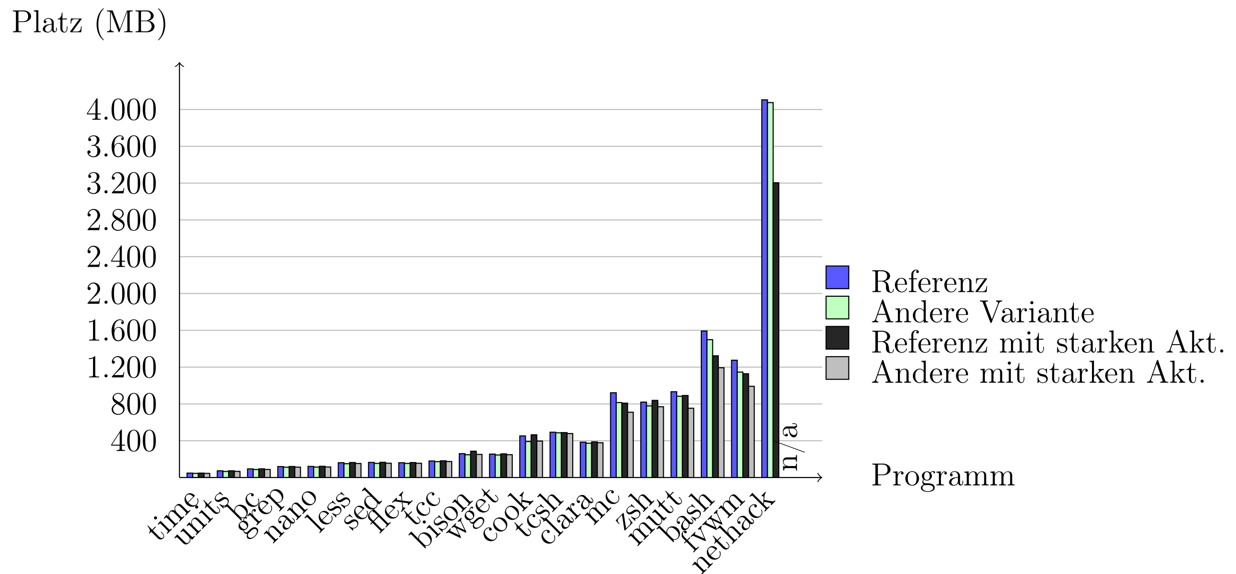


Abbildung 13.38: Speicherbedarf-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-insensitiv)

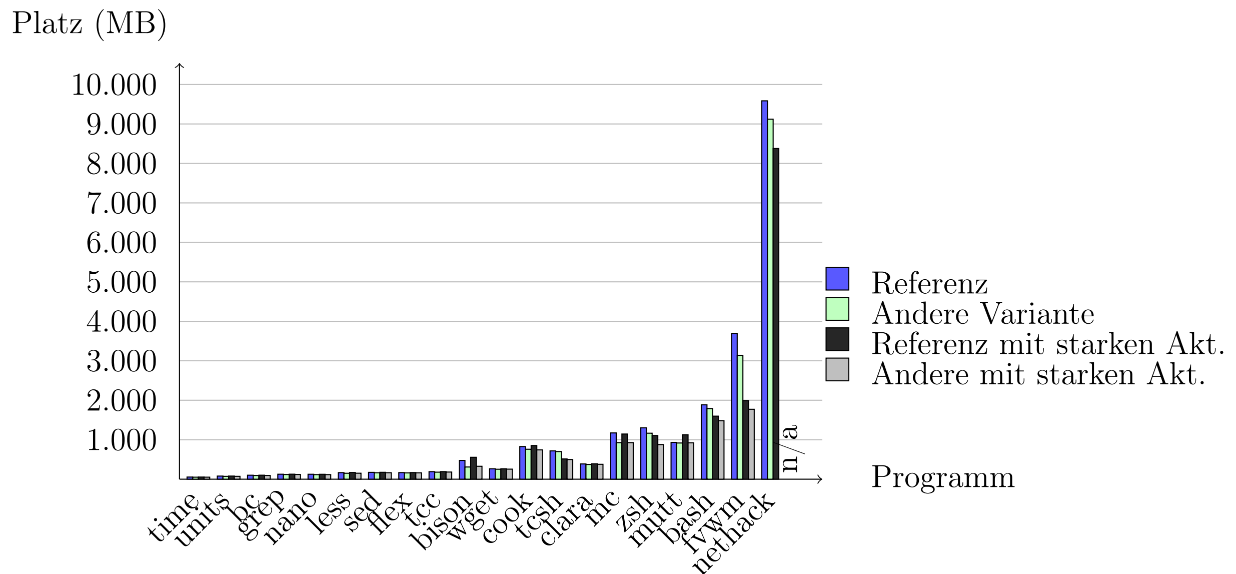


Abbildung 13.39: Speicherbedarf-Vergleich zur erschöpfenden Propagierung ohne Zyklenkontraktion (feld-sensitiv)

13.5 Zusammenfassung der Evaluation

Nachdem wir nun sowohl die Resultate als auch die Effizienz verschiedener Ausprägungen der Analyse untersucht haben, fassen wir die Beobachtungen in diesem Abschnitt kurz zusammen.

Die *Fluss-Sensitivität* kann die Präzision der Resultate gegenüber einer fluss-insensitiven Analyse mäßig verbessern. In manchen Fällen (z.B. bei mc oder zsh) sind auch deutliche Steigerungen zu beobachten. Will man nur die Zeigerziele bestimmen, so ist Andersens Analyse trotz ebenfalls kubischer Komplexität aufgrund des viel kleineren Propagierungsgraphen deutlich schneller. Das Bild dreht sich jedoch, wenn die Datenfluss-Darstellung mit Hilfe der Zeigerziele aufgebaut werden soll. Dann macht sich fluss-insensitiv die höhere Zahl an Zeigerzielen in einer schlechteren Laufzeit bemerkbar.

Die *Unterstützung für indirekte starke Aktualisierungen* kann den ISSA-Graphen gegenüber der fluss-sensitiven Analyse mit lediglich schwachen indirekten Aktualisierungen teilweise gut verkleinern. Die Zahl der Zeigerziele reduziert sich jedoch kaum. Die Relaxation als Strategie zum Umgang mit potentiellen Programmierfehlern treibt die Zahl der Iterationen in einigen Fällen hoch, was insbesondere bei einer erschöpfenden Propagierung spürbar ist. Auch ohne Relaxation ist die Zahl der Iterationen in dieser Analysen-Ausprägung tendenziell höher, was sich auf die Laufzeit auswirkt. Der zusätzliche Platzbedarf für den Blockade-Graphen und die blockierten Kanten wird mehr als aufgewogen durch die Reduktionen an anderer Stelle dank der höheren Präzision.

Die *Laufzeit* der kombinierten Analyse folgt mit inkrementeller Zyklenkontraktion einem moderat quadratischen Verlauf bezüglich der Programmgröße in Instruktionen. Auch ohne Zyklenkontraktion wird die theoretisch ermittelte kubische Schranke bei inkrementeller Propagierung nicht sichtbar. Die absoluten Laufzeiten sind mit Zyklenkontraktion angenehm.

Der *Speicherverbrauch* ist in absoluten Zahlen relativ hoch. Der Bedarf wächst jedoch nur linear im Verhältnis zur Zahl der Instruktionen. Erfolgt fluss-insensitiv auch eine ISSA-Konstruktion, so ist deren Platzbedarf vergleichbar mit dem Bedarf der kombinierten Analyse.

Die *Zyklenkontraktion* hat sich in der empirischen Untersuchung als wichtigstes Mittel erwiesen, um die Laufzeit der Analyse bei den größeren Programmen zu reduzieren. Ihre ohnehin geringen Kosten verschwinden angesichts der deutlichen Vorteile für die Effizienz. In Bezug auf Andersens flussinsensitive Analyse zeichnen einige Publikationen das gleiche Bild, d.h. auch diesbezüglich haben wir wieder eine Parallele der kombinierten Analyse zu Andersens Verfahren.

Die *Rückwärts-Propagierung* erweist sich überraschend deutlich schneller als die naheliegendere Vorwärts-Propagierung. Die *inkrementelle Propagierung* hat einen hohen Platzbedarf, reduziert die Laufzeit gegenüber der erschöpfenden Variante jedoch in der Regel spürbar.

Die Tabelle 13.4 zeigt zusammenfassend für die feld-insensitive Analyse, welche Faktoren in der Laufzeit wir durch die einzelnen Maßnahmen gewinnen konnten:

- *Inkrementell:* Diese Spalte enthält den Quotienten

$$\frac{\text{Laufzeit erschöpfend}}{\text{Laufzeit inkrementell}}$$

  (jeweils mit Vorwärts-Propagierung ohne Zyklenkontraktion)

- *Zyklen:* Diese Spalte enthält den Quotienten

$$\frac{\text{Laufzeit inkrementell ohne Zyklenkontraktion}}{\text{Laufzeit erschöpfend mit Zyklenkontraktion}}$$

  (jeweils mit Vorwärts-Propagierung). Wir nutzen hierbei die inkrementelle Propagierung als Vergleichswert, da die Zyklenkontraktion inkrementell erfolgt.

- *Rückwärts:* Diese Spalte enthält den Quotienten

$$\frac{\text{Laufzeit mit Vorwärts-Propagierung}}{\text{Laufzeit mit Rückwärts-Propagierung}}$$

  (jeweils mit inkrementeller Zyklenkontraktion)

	Fluss-sensitiv			Mit starken Akt.		
Programm	Inkrementell	Zyklen	Rückwärts	Inkrementell	Zyklen	Rückwärts
time	0,33	3	0,33	0,33	1,50	0,50
units	0,67	1,75	0,71	0,67	1,62	0,72
bc	1,26	1,50	0,72	1,19	1,49	0,75
grep	0,48	2,17	0,73	0,51	2,21	0,71
nano	2,84	1,50	0,75	2,88	1,49	0,74
less	5,05	2,40	1,57	10,92	1,50	2,18
sed	4,40	1,28	1,06	4,25	1,28	1,08
flex	1,47	1,62	0,89	1,46	1,61	0,89
tcc	1,94	1,81	0,87	1,64	1,63	0,86
bison	1,60	4,06	0,76	9,95	1,10	0,55
wget	2,68	1,49	1,01	2,90	1,41	1
cook	6,98	5,94	2,31	7,45	5,01	2,25
tcsh	7,79	2,35	1,44	9,84	1,61	1,51
clara	0,73	1,86	0,81	0,80	1,74	0,79
mc	7,77	5,64	2,15	6,59	4,83	1,66
zsh	3,41	21,41	2,42	3,53	18	2,54
mutt	27,14	1,84	2,40	28,85	1,59	2,34
bash	19,34	2,59	4,05	27,16	1,80	4,27
fvwm	6,59	6,53	2,02	8,90	3,96	2,13
nethack	10,74	8,27	1,54	>14,88	4,19	1,43

Tabelle 13.4: Laufzeitgewinne als Faktoren (feld-insensitiv)

	Fluss-sensitiv			Mit starken Akt.		
Programm	Inkrementell	Zyklen	Rückwärts	Inkrementell	Zyklen	Rückwärts
time	0,60	1,25	0,67	0,80	1,67	0,50
units	0,64	1,56	0,73	0,69	1,63	0,70
bc	1,22	1,43	0,76	1,31	1,40	0,74
grep	0,47	2,04	0,81	0,60	1,57	0,70
nano	1,24	1,23	0,81	1,11	1,55	0,76
less	7,44	1,96	1,58	7,22	1,96	1,54
sed	1,74	1,40	0,89	2,12	1,20	0,88
flex	0,67	1,67	0,79	0,66	1,60	0,84
tcc	2,77	1,83	0,99	2,53	1,79	0,99
bison	16,49	3,52	7,66	25,62	2,27	9,57
wget	1,14	1,66	0,85	1,32	1,51	0,86
cook	9,03	6,20	3,71	9,53	5,39	3,62
tcsh	7,65	2,10	1,32	12,96	1,26	1,49
clara	0,79	1,79	0,86	0,80	1,70	0,88
mc	2,67	1	0,86	2,35	1,01	0,86
zsh	7,07	10,47	2,11	8,08	9,32	2,45
mutt	11,19	3,49	1,44	48,58	1,17	2,23
bash	7,15	4,66	1,41	9,80	3,79	1,51
fvwm	4,17	3,75	1,18	17,75	1,87	1,73
nethack	3,67	6,91	1,03	>3,94	4,73	1,13

Tabelle 13.5: Laufzeitgewinne als Faktoren (feld-sensitiv)

Diese Faktoren wurden getrennt für die fluss-sensitive Analyse und die Ausprägung mit indirekten starken Aktualisierungen erhoben. Tabelle 13.5 enthält die gleichen Daten für die feld-sensitiven Varianten dieser Analysen. Bei den abgebrochenen nethack-Läufen können wir immerhin noch die Zeiten ohne indirekte starke Aktualisierungen als untere Schranke verwenden.

Ein Faktor größer als Eins bedeutet in den Tabellen eine Beschleunigung durch die jeweilige Maßnahme. Wir sehen, dass die Zyklenkontraktion durchweg eine zum Teil deutliche Beschleunigung darstellt, wohingegen die anderen beiden Strategien bei einigen kleineren Programmen durchaus auch zu einer Verlangsamung führen können. Hier ist jeweils die Gesamtlaufzeit so kurz, dass der Zusatzaufwand der Maßnahmen den Gewinn übersteigt. Dies gilt auch für das etwas größere Programm clara, dessen Analyse durch zwei der drei Strategien ebenfalls noch eine Verlangsamung erfährt, aber dennoch in weniger als drei Sekunden abläuft. Bei nethack muss man aufgrund der fehlenden exakten Messwerte für die erschöpfende Propagierung ohne Zyklenkontraktion in der Variante mit indirekten starken Aktualisierungen vorsichtig sein: Da die real benötigte Zeit höher als die für die Daten angesetzte Zeit ist, liegt der tatsächliche Faktor noch einmal höher als der (auch schon beachtliche) Faktor in der Tabelle.

Abschließend haben wir in Tabelle 13.6 noch das Produkt der drei Faktoren angeführt, um den Gesamtgewinn darzustellen. Für eine höhere Genauigkeit haben wir diese Zahlen als Quotient

$$\frac{\text{Laufzeit vorwärts erschöpfend ohne Zyklenkontraktion}}{\text{Laufzeit rückwärts auf inkrementellem SCC-DAG}}$$

berechnet.

Den höchsten Gewinn erlebt dabei die feld-sensitive Analyse von bison, hauptsächlich durch das inkrementelle Vorgehen, welches die erhöhte Zahl an Iterationen ausgleicht. Generell demonstriert diese Übersicht, dass unsere Strategien vor allem die Analyse der größeren Programme deutlich beschleunigen. Wir haben mit unseren Maßnahmen Beschleunigungsfaktoren größer als 100 erreicht, womit der praktische Einsatz der Analyse spürbar angenehmer ist.

	Feld-insensitiv		Feld-sensitiv	
Programm	Fluss-sensitiv	FS mit starken Akt.	Fluss-sensitiv	FS mit starken Akt.
time	0,33	0,25	0,50	0,67
units	0,82	0,78	0,73	0,78
bc	1,36	1,33	1,33	1,35
grep	0,76	0,79	0,78	0,66
nano	3,18	3,20	1,24	1,31
less	19,08	35,74	23,03	21,88
sed	5,97	5,84	2,16	2,22
flex	2,13	2,07	0,88	0,89
tcc	3,04	2,29	5,04	4,47
bison	4,94	6,08	444,32	555,84
wget	4,02	4,10	1,61	1,73
cook	95,70	84,10	207,41	185,98
tcsh	26,35	23,97	21,20	24,35
clara	1,09	1,10	1,23	1,21
mc	94,19	52,82	2,29	2,03
zsh	176,24	161,82	156,15	184,42
mutt	119,77	107,58	56,29	126,26
bash	202,38	208,66	46,86	56,28
fvwm	87,13	75,01	18,52	57,52
nethack	136,65	>89,08	26,13	>21,04

Tabelle 13.6: Gesamtfaktoren der Laufzeitgewinne

Teil III

Schluss

Kapitel 14

Ansätze für weitere Arbeiten

Wir haben in dieser Arbeit zentrale Aufgaben für statische, Datenfluss-basierte Analysen thematisiert: Kontrollfluss, Datenfluss, und Zeigerziele. Aufgrund der zentralen Bedeutung dieser Themen gibt es viele Anwendungen, die auch weiterhin schnellere und präzisere Lösungen fordern werden. So werden noch einige Anstrengungen nötig sein, um Programme in der Größenordnung von 1 MLoC und mehr zu beherrschen, die keineswegs selten sind. Daher skizziert dieses Kapitel Ideen und Ansätze für weitere Arbeiten, die auf den hier präsentierten Lösungen aufbauen können. Diese Überlegungen haben wir unterteilt in Maßnahmen zur Steigerung der Präzision, zur Verbesserung der Skalierbarkeit sowie zur Erweiterung der Fähigkeiten der Analyse.

14.1 Präzisionssteigerungen

Unser Beitrag hat die Präzision von Andersens Analyse auf fluss-sensitiv erhöht. Weitere Präzisionssteigerungen konnten wir durch die Unterstützung starker Aktualisierungen sowie die MOVP-Lösung anstelle der MOP-Lösung erreichen. Hier streifen wir nun weitere Optionen zur Präzisionsverbesserung.

14.1.1 Kontext-abhängiger Datenfluss

Eine direkte Fortsetzung der Ideen aus Kapitel 9 zur kontext-abhängigen Zeigerziel-Propagierung ist es, auch den Datenfluss kontext-abhängig zu repräsentieren. Dazu führen wir neben den bisherigen IDFG-Knoten, welche konkrete Objekte definieren oder verwenden, *abstrakte* IDFG-Knoten mit Bezug zu einem abstrakten, kontext-abhängigen Objekt ein. Abschnitt 9.6 hat bereits dargelegt, dass wir den ISSA-Eingang, über den kontext-abhängige Ziele an eine Dereferenzierung gelangen, als Stellvertreter für alle über ihn ankommenden derartigen Zeigerziele einsetzen können. Der Eingang dient dann als das abstrakte, kontext-abhängige Objekt in den zugehörigen IDFG-Knoten. Damit gelingt wiederum eine Unterscheidung in kontext-unabhängige und kontext-abhängige Knoten. Für die kontext-abhängigen Knoten ergibt sich eine Platzersparnis gegenüber dem bisherigen Ansatz, da die seither unterschiedenen Fälle der einzelnen kontext-abhängigen Ziele durch die Abstraktion zusammengefasst werden. Außerdem erlaubt diese Darstellung, Seiteneffekte für kontext-abhängige Knoten (fast) nur noch zu den auslösenden Aufrufern zu propagieren und somit die Präzision auch im Datenfluss-Teil in Bezug auf die Beachtung der Aufrufsemantik zu steigern.

Wir treffen hier jedoch auf das bekannte Aliasing-Problem: Können zwei Eingänge e_1, e_2 als abstrakte Objekte für das gleiche Objekt x stehen, so dürfen wir die resultierenden beiden abstrakten Datenfluss-Graphen mit den abstrakten Knoten zu e_1 und e_2 nicht isoliert betrachten. Ein solches Vorgehen würde potentielle Datenflüsse für x unterschlagen. Zuerst müssen wir daher Aliasing an den Eingängen erkennen. Weiterhin müssen wir uns für eine Lösung des Aliasing-Problems entscheiden. Nachdem Wilsons PTFs zu teuer sind und eine Zusammenlegung der betroffenen abstrakten Datenfluss-Graphen wohl zu ungenau wird, vermuten wir eine gute Lösung darin, zusätzlich zu den isolierten abstrakten DFGs noch einen vereinenden konkreten DFG für x anzulegen. Damit behalten wir für die übrigen, nicht vom Aliasing betroffenen Ziele von e_1 und e_2 die Vorteile. Um Seiteneffekte für x präziser als bisher zu behandeln, vermerken wir zusätzlich am nun entstehenden Eingang für x die Eingänge, über die x als Ziel in die Funktion kam.

Besonders für das häufige Konstrukt des Referenzparameters erwarten wir eine deutliche Präzisionsverbesserung durch diese Maßnahmen. Damit einher geht aber auch die Hoffnung auf eine verbesserte Skalierbarkeit durch die kompaktere und präzisere IDFG-Darstellung.

Gegenüber einer vollständig kontext-sensitiven Darstellung, welche die Effekte der verschiedenen Aufrufer echt voneinander trennt, halten wir diese polynomielle Lösung für praktikabler.

14.1.2 Relationale Probleme

Unsere Analyse betrachtete bislang jeden Datenfluss-Pfad isoliert von den übrigen. Eine höhere Genauigkeit ist durch eine relationale Betrachtung möglich. Betrachten wir dazu folgendes Beispiel aus einer GUI-Applikation:

```
void set_message (Dialog* dialog, Box* mbox)
{
    dialog->message_box = mbox;
}

void f()
{
    ...
    set_message (ok_dialog, ok_mbox);
    ...
}

void g()
{
    ...
    set_message (yesno_dialog, yesno_mbox);
    ...
}
```

In *set_message* ist *dialog* eine Variable, die auf ein Widget für ein Dialogfenster verweist. Die Zuweisung bestimmt, welche Meldung innerhalb des

Dialogfenster angezeigt wird, indem dessen Attribut hierfür belegt wird. In diesem Fall ist die Meldung jedoch nicht direkt gegeben, sondern ihrerseits über den Zeiger *mbox* designiert.

Unsere Analyse – und viele andere auch – würde hier als Zeigerziele ermitteln:

$$PointsTo(dialog) = \{ok_dialog, yesno_dialog\},$$

$$PointsTo(mbox) = \{ok_mbox, yesno_mbox\}.$$

Für ein genaueres Resultat wäre es jedoch wünschenswert, dass die beiden Parameter *dialog* und *mbox* nicht derart isoliert betrachtet werden. Dieses Ergebnis ließe nämlich auch zu, dass *mbox* auf *yesno_mbox* zeigt, wenn es sich um einen *ok_dialog* handelt. Ziel einer präziseren Analyse wäre daher die gemeinsame Betrachtung der beiden Zeiger.

Im Rahmen einer neuen statischen Analyse zur Untersuchung graphischer Oberflächen wurde vom Autor hierzu ein Verfahren entwickelt [Staiger 2007], welches gemeinsam zu betrachtende Variablen als eine Einheit kapseln kann und somit die gewünschte erste Lösung liefert. Dieses Verfahren sucht nach den interprozedural gültigen transitiven Definitionen über eine Aufwärts-Propagierung der gekapselten Einheiten im Aufrufgraphen. Dabei kann es die Summary-Kanten nutzen, welche unsere MOVP-Propagierung von Zeigerzielen bestimmt, um nicht in kopierende Funktionen hinabsteigen zu müssen (die Summary-Kanten erlauben eine lokale Verfolgung der Datenfluss-Beziehungen zurück zu den Definitionen).

14.2 Maßnahmen für die Skalierbarkeit

Unsere neue Analyse ist bereits für eine fluss-sensitive Zeigeranalyse im Vergleich zu anderen Verfahren akzeptabel bis gut skalierbar, sowohl theoretisch als auch in der Praxis. Der Wunsch nach noch deutlich besserer Skalierbarkeit ist jedoch stets vorhanden, und darum präsentieren wir hier ein paar Ideen, um diesem Ziel näher zu kommen. Zusätzlich sei auf die Idee der adaptiven Analyse in Abschnitt 14.4 verwiesen, die ebenfalls die Skalierbarkeit verbessern kann.

14.2.1 BDDs

BDDs zeigen sich in aktuellen Publikationen als geeignete Datenstrukturen, um die Skalierbarkeit präziser Analysen zu verbessern. Daher wäre es eine interessante Untersuchung, den Einsatz von BDDs für unsere neue Analyse zu betrachten. BDDs können jedoch auch exponentiell teuer werden, und die Publikationen berichten von aufwändigen Adjustierungen der Variablenordnung usw. nach dem trial-and-error Prinzip. Für große Programme, die mit hoher Genauigkeit untersucht werden sollen, kann dieser Aufwand lohnenswert sein, da konventionellere Strukturen bislang keine Chancen bieten.

14.2.2 Einbeziehung der Zielanalyse

In vielen Fällen ist es gar nicht nötig, für alle Programmstellen eine konservative Lösung zu berechnen. Eine Zielanalyse interessiert sich möglicherweise nur für einen Ausschnitt des Programmes. Auch für dieses Szenario wurden bereits Ansätze in der Literatur diskutiert: zu nennen sind hier z.B. die Fragment-Analyse von Rountev u. a. [1999], sowie ein bedarfsgesteuertes (demand-driven) oder klientengetriebenes (client-driven) Vorgehen. Solche Ideen können mit unserer neuen Analyse kombiniert werden, um das gewünschte Ziel zu erreichen. Hierfür eignet sich insbesondere die Rückwärtspropagierung: Die Zielanalyse benennt in einem solchen Szenario die sie interessierenden Dereferenzierungen, und (nur) diese werden dann propagiert. Auf dem Weg angetroffene weitere Dereferenzierungen, die einen Einfluss auf den jeweiligen Zeiger haben könnten, müssen dabei zusätzlich ermittelt und propagiert werden. Ein Konstrukt ähnlich zum Blockade-Graphen kann vermutlich dabei helfen, diese zusätzlichen Dereferenzierungen aufzuspüren.

14.2.3 Parallelisierung

Immer mehr Rechner haben mehr als einen Rechenkern, weswegen Überlegungen zur Parallelisierung der Analyse lohnenswert sein können. Das reduziert zwar nicht die insgesamt zu erledigende Arbeit, erreicht aber durch eine gleichzeitige Abarbeitung verschiedener Teilschritte eine kürzere Laufzeit.

Betrachten wir auf der Suche nach Parallelisierungsmöglichkeiten zunächst den Graphaufbau. In diesem können wir natürlich versuchen, die Menge der neuen Knoten parallel zu verarbeiten. Restriktionen bezüglich der Reihenfolge gibt es da im Wesentlichen nur in Bezug auf die Korrektur der Objektgraphen: Das Verschieben derjenigen Nachfolger des gefundenen Dominators, die vom neuen Knoten n dominiert werden, geht davon aus, dass selbiges Verschieben für den Dominator bereits erfolgt ist. Denn andernfalls, wenn der Graph auf diese Weise zuerst für n, dann für den Dominator korrigiert wird, gelangen eigentlich von n dominierte Knoten nur bis zum Dominator. Es sollte bei einer Parallelisierung daher bedacht werden, dass in einem solchen Szenario, in dem sowohl n als auch dessen Dominator neu sind, nicht durch eine parallele Bearbeitung der beiden diese Problematik auftritt.

Eine Möglichkeit, dies zu umgehen, besteht darin, dass die parallel laufenden Einheiten jeweils Knoten verschiedener Objekte oder verschiedener Unterprogramme behandeln. Das umgeht das genannte Problem. Jedoch muss auch in diesem Fall immer noch beim Verbinden von Objektgraphen darauf geachtet werden, dass es keine Konflikte durch gleichzeitigen Zugriff von den beiden zu verbindenden Seiten gibt.

Die (erschöpfende) Propagierung lässt sich vermutlich ebenfalls mit Gewinn parallelisieren. So könnte man die parallel laufenden Einheiten mit der Propagierung jeweils unterschiedlicher Propagierungselemente beauftragen, was nur beim Eintragen eines neuen Ziels an einer Dereferenzierung einen Schutz vor gleichzeitigem gemeinsamem Durchführen dieser Aktion bedarf. Um auch den noch zu vermeiden, kann man hoffen, dass der ISSA-Graph in genügend viele Zusammenhangskomponenten zerfällt, so dass jeder Prozess oder Thread auf einem anderen Teil des Graphen arbeitet. Dann bleibt nur noch die Menge New_Nodes über ein Lock zu schützen.

14.2.4 Andere Beschleunigungen

Die einfachste Zeigeranalyse besteht darin, die potentiellen Ziele durch Ausschluss derjenigen Variablen zu reduzieren, deren Adresse nie genommen wird. Eine Fortsetzung dieser Idee besteht darin, zu ermitteln, welche Varia-

blen insgesamt oder speziell für eine gegebene Dereferenzierung nicht mehr als Ziel in Frage kommen. Neben der Zeigeranalyse findet sodann eine komplementäre No-Target-Analyse statt, die untersucht, ob die Adresse einer Variable bereits vollständig verfolgt wurde. Damit grenzen wir die Zeigerziel-Mengen von beiden Seiten ein: von unten durch die Zeigeranalyse, von oben durch die No-Target-Analyse. Möglicherweise gelingt dadurch eine schnellere Konvergenz oder die frühzeitige Anwendung von Optimierungen, die nur bei Kenntnis bereits aller Ziele an einem Zeiger anwendbar sind.

14.2.5 Speicherreduktion für große Programme

Für große Programme stellt neben dem Zeitbedarf auch der Platzbedarf eine signifikante Hürde dar. Daher sind Strategien gefragt, mit denen die Analyse zu jedem Zeitpunkt nur einen kleinen Teil des zu analysierenden Programmes und der bisherigen Resultate dazu im Speicher halten muss. Die erschöpfende Propagierung als Realisierung des Propagierungs-Schrittes bietet sich hier an, allerdings mit den entsprechenden Laufzeiten.

Chatterjee u. a. [1999] erwähnt, dass seine Analyse auch unter diesem Aspekt entworfen wurde, zu jedem Zeitpunkt nur geringe Teile im Speicher zu benötigen. Andere Quellen für derartige Ansätze sind übliche Linker, die nur Teile einer Objektdatei für das eigentliche Linken einlesen (Symboltabellen), und danach Datei für Datei vorgehen können. Ein solcher Linker wurde vom Autor auch für das Projekt Bauhaus entwickelt und zeigte sich interessanterweise auch deutlich effizienter als ein vorausgehender Linker, welcher die Zwischendarstellung jeweils komplett einlas und die Dateien im Speicher vereinte.

Die Kernanalyse kann für diese Aufgabe von der Freiheit profitieren, in jedem der beiden zentralen Schritte die einzelnen Aktionen in fast beliebiger Reihenfolge ausführen zu können. So können wir für den Graphaufbau-Schritt gezielt Funktion für Funktion vorgehen, aufgrund der Seiteneffekte dabei am besten im Rahmen einer Postfix-Durchquerung des Aufrufgraphen. Dann muss dieser Schritt jeweils nur die für die aktuelle Funktion relevanten Daten im Speicher vorhalten und kann den Rest auslagern. Die Aufteilung nach

Funktionen ist dabei praktisch, weil eine Funktion vollständig in einer Datei steht: somit ist kein Linken vorab erforderlich, die Analyse betrachtet jeweils eine (Funktion aus einer) Datei.

Die Propagierung überschreitet dagegen die Funktionsgrenzen häufiger. Eine Gruppierung der Einzelschritte nach Funktionen wird hier aufwändiger, aber nicht unmöglich.

14.3 Integration von Analysen und Transformationen

Eine andere künftige Arbeit kann darin bestehen, die hier präsentierte Analyse zu einem Datenfluss-Framework auszubauen. Es ist klar, dass nach Abschluss der Analyse bekannte Frameworks zur Lösung von Datenfluss-Problemen genutzt werden können, so dass die Lösung auch auf Zeigerziele und Datenfluss-Relationen zugreifen kann. Spannend ist jedoch die Frage, ob auch ein Framework gelingt, das die Probleme während der Kernanalyse löst. Dabei würde der Propagierungs-Schritt nun allgemeiner zum Lösungsschritt, in dem jeweils weitere Teile der anliegenden Datenfluss-Probleme – inklusive der Zeigerpropagierung – berechnet werden. Kapitel 9 hat hierzu bereits gezeigt, wie das IFDS-Framework auf den ISSA-Graphen übertragen werden kann. Dabei haben wir am Beispiel der Propagierung auch gesehen, dass im Vergleich zu einer Kontrollfluss-basierten Problemspezifikation die Transferfunktionen einfacher werden.

Die allgemeine Hürde besteht vermutlich darin, zu einem gegebenen Datenfluss-Problem eine inkrementelle Lösung zu finden, die damit zurecht kommt, dass vor dem Fixpunkt noch nicht konservative Lösungen aller parallel bearbeiteten Probleme vorliegen. Ein Nutzen der Integration sehen wir darin, dass damit die bereits angesprochenen Ideen von bedarfs- oder klientengetriebenen Gesamtansätzen umgesetzt werden können. Weiterhin kann eine Integration von Transformationen auf Basis von Analyse-Resultaten erfolgen. So könnte z.B. eine Konstanten-Propagierung erfolgen und tote Verzweigungsäste eliminieren etc.

14.4 Adaptive Analyse

Traditionelle Programmanalysen sind wenig intelligent: Sie berechnen nach einem einheitlichen Muster ihre Ergebnisse, wie sie sie für praktisch jedes Programm herleiten können. *Adaptive* Analysen dagegen würden sich der Analyse-Situation anpassen: Dort, wo das zu untersuchende Programm einfach ist, werden Analysevarianten eingesetzt, die diese Einfachheit im Sinne einer besseren Laufzeit und Genauigkeit ausnutzen können. Beispielsweise sind viele Unterprogramme relativ einfach gehalten (ohne gotos etc.) und können daher mit speziellen Analysen untersucht werden. Auch andere Anpassungen an die Situation sind möglich: So kann man sich an die Anfrage des Benutzers anpassen, die vielleicht nur für einen Programmteil ein präzises Ergebnis verlangt und im restlichen Programm mit ungenaueren Ergebnissen zufrieden ist. Ebenso können die bisherigen (Zwischen-)Resultate als Quelle für Anpassungen benutzt werden. Während ein adaptives Vorgehen in anderen Wissenschaftsbereichen, z.B. der Numerik von Differentialgleichungen, bereits üblich ist, betreten wir im Gebiet der Programmanalysen damit noch weitgehend Neuland. Andere Ähnlichkeiten zur Numerik sind dagegen bereits auszumachen (Fixpunkt-Iteration, Konvergenz(geschwindigkeit), Fehlerschätzung) und z.B. im Rahmen der abstrakten Interpretation auch als einheitliches Prinzip erkannt worden.

Eine adaptive Steuerung für die Kernanalyse könnte beispielsweise wie ein Regelkreis nach jedem Einzelschritt anhand gewisser Metriken zu den Zwischenresultaten entscheiden, für welche Objekte im nachfolgenden Schritt eine präzisere Propagierung oder feinere Modellierung sinnvoll ist.

14.5 Unterstützung weiterer Sprachkonstrukte

In diesem Abschnitt wollen wir kurz ein paar Ansätze skizzieren, wie die in dieser Arbeit für C-Programme beschriebene Analyse so erweitert werden kann, dass sie weitere Sprachkonstrukte beherrscht.

14.5.1 Objektorientierung

Für die Unterstützung objektorientierter Programme muss die Analyse zusätzlich mit virtuellen Aufrufen zurecht kommen. Das sollte jedoch keine größeren Schwierigkeiten bereiten, da die Analyse bereits die Zeigerziele (und damit die möglichen dynamischen Typen) der Objekte bestimmt, über welche die Aufrufe erfolgen. Es fehlt dann lediglich eine Abbildung von den dynamischen Typen und der statisch bekannten Deklaration der gerufenen Methode auf die damit potentiell aufgerufenen Implementierungen, was anhand der Vererbungshierarchie und den darin anzutreffenden Redefinitionen zu bewerkstelligen ist. Mit einer entsprechenden Unterstützung sollte die Analyse damit auch auf Programme in objektorientierten Sprachen anwendbar sein.

14.5.2 Exceptions

Ausnahmen (engl. *exceptions*) bewirken eine Abweichung vom normalen Kontrollfluss. Im Unterschied zu einem *goto*-Sprung ist dabei das Ziel des Sprunges nicht bekannt, den eine Ausnahme auslöst. Die möglichen Ziele sind Exception-Handler und, für den unbehandelten Fall, das Programmende. Das konkrete Ziel hängt dabei vom Typ des geworfenen Objekts ab. Das Ziel einer Ausnahme kann noch im gleichen Unterprogramm liegen oder irgendwo entlang der Aufrufkette.

Neben indirekten Aufrufen sind Ausnahmen damit das zweite Konstrukt, welches einen erst im Laufe der Analyse erkennbaren Einfluss auf den Kontrollfluss hat. Wie in Abschnitt 2.1 beschrieben, haben wir sie in dieser Arbeit nicht näher betrachtet, weil hier auch die gewünschte Modellierung nicht so klar ist. Eine Möglichkeit ist eine Darstellung ähnlich zur Darstellung der Seiteneffekte in der ISSA-Form. Dabei hält man die interprozeduralen Übergänge des Sprunges explizit fest und propagiert die Ausnahme sozusagen in kleinen Schritten.

Die Analyse müsste für die Unterstützung der Ausnahmen die zusätzlichen lokalen Kontrollflüsse sowie die interprozedurale Propagierung realisieren. Außerdem muss an den erreichten Exception-Handlern überprüft werden können, ob dieser die ankommende Ausnahme fängt oder nicht. Für

die nun ggf. erst dynamisch bekannten lokalen (zusätzlichen) Kontrollflüsse muss entsprechend die Dominanzinformation für den korrekten Ablauf des Graphaufbaus angepasst werden. Die Änderungen an der Analyse betreffen also primär den Graphaufbau.

Die neben den angesprochenen expliziten Ausnahmen noch existierenden impliziten Ausnahmen benötigen zusätzlich die Erkennung, an welchen Stellen eine solche Ausnahme erhoben werden könnte. Die kombinierte Analyse kann leicht Stellen aufspüren, an denen potentiell eine Ausnahme aufgrund der Dereferenzierung eines Null-Zeigers stattfindet; andere implizte Ausnahmen, z.B. bei Zugriffen auf nicht vorhandene Array-Elemente oder bei der Division durch Null, erfordern jedoch Informationen, die die Analyse bislang nicht hat. Sie müsste dazu z.B. die möglichen Werte diverser Integer-Variablen zusätzlich zu den Zeigerzielen propagieren. Es scheint derzeit jedoch allgemein noch üblich, die impliziten Ausnahmen im Rahmen einer solchen Analyse zu ignorieren, da ansonsten die Präzision und das Mengengerüst enorm leiden.

14.5.3 Parallelität

Programme, die Threads benutzen, werden mit der bislang beschriebenen Analyse nicht unbedingt konservativ analysiert. Das Problem hierbei sind Datenflüsse zwischen den Threads (ob vom Programmierer gewollt oder nicht), welche die Analyse noch nicht erkennt.

Ein Ansatz, diese zusätzlichen Datenflüsse zu erkennen, ist folgender: Im Fixpunkt der Analyse bestimmt man sozusagen als besonderen Graphaufbau-Schritt diese Datenflüsse zwischen Threads. Hierzu kann man mit Techniken, auf die wir hier nicht näher eingehen, eine *may-happen-in-parallel*-Information etablieren, sowie die Objekte identifizieren, welche potentiell in verschiedenen zugleich aktiven Threads definiert oder verwendet werden, und damit zusätzliche potentielle Datenfluss-Kanten entdecken. Über die Betrachtung, welche Locks jeweils gehalten werden (wozu in der Regel Zeigerinformationen wichtig sind und mit unserer Analyse zur Verfügung stehen), kann die Menge dabei noch reduziert werden.

Nach diesem besonderen Graphaufbau-Schritt setzt man dann die Analyse normal fort, da sich über die neuen Datenfluss-Kanten möglicherweise wieder neue Zeigerziele ergeben. Dies kann im nächsten Fixpunkt auch wieder zur Entdeckung neuer Datenflüsse zwischen den Threads führen, so dass wir insgesamt um die Analyse herum eine weitere Fixpunkt-Iteration erhalten (ähnlich zur Relaxation bei indirekten starken Aktualisierungen), die in jedem inneren Fixpunkt den besonderen Graphaufbau-Schritt ausführt. Da die Menge an potentiell zugleich aktiven Threads, der jeweils gehaltenen Locks, sowie der definierten und verwendeten Objekte dabei monoton wächst, nähert sich dieses Vorgehen sukzessive einer konservativen Lösung im endgültigen Fixpunkt.

Kapitel 15

Zusammenfassung und Abgrenzung

In diesem Kapitel positionieren wir die neue Analyse im Kontext zahlreicher bestehender Verfahren. Damit zeigen wir Unterschiede und Gemeinsamkeiten zu anderen Lösungen und stellen den Beitrag der Dissertation noch einmal zusammen. Zuvor fassen wir die Arbeit insgesamt zusammen. Damit schließt das Kapitel das Werk ab.

15.1 Zusammenfassung und Fazit

Die wohl einflussreichste Zeigeranalyse bislang stammt von Andersen [1994]. Es handelt sich dabei um eine fluss- und kontext-insensitive Zeigeranalyse, die vielfach in nachfolgenden Publikationen aufgegriffen wurde. Diese Weiterführungen der Analyse betrachteten Optimierungen – vor allem eine effiziente Zyklenerkennung –, übertrugen die Analyse auf andere Sprachen und schufen kontext-sensitive Erweiterungen.

In dieser Dissertation haben wir nun eine neue fluss-sensitive Zeigeranalyse beschrieben. Wir haben geschildert, dass wir diese durchaus als fluss-sensitive Ausprägung von Andersens Analyse verstehen können. Anstelle des Constraint-Graphen tritt dabei der Datenfluss-Graph, bei uns in Form des ISSA-Graphen. Wie beim Original ist auch der Ansatz der neuen Analy-

se relativ einfach: Das abwechselnde Ermitteln von Datenfluss-Beziehungen und Propagieren von Zeigerzielen anhand dieser löst iterativ das Problem. Wir haben gezeigt, dass diese Variante trotz der höheren Präzision noch die gleiche kubische Komplexität besitzt.

Gegenüber anderen fluss-sensitiven Zeigeranalysen haben wir mit der Analyse einen neuen Ansatz geschaffen, der insbesondere die Zeiger- und die Datenfluss-Analyse in Kombination betrachtet. Diese Kombination erfüllt dabei auch gleich Aufgaben aus dem Bereich der Kontrollfluss-Analyse, nämlich die Auflösung indirekter Aufrufe. Während bekannte FSCI-Verfahren die Zeigeranalyse als klassisches Datenfluss-Problem auf dem ICFG betrachten, haben wir stattdessen dank der kombinierten Lösung die Datenfluss-Darstellung selbst zugrunde legen können. Wie wir mit unserem Korrektheitsbeweis nachgewiesen haben, ist dabei keine Vorabüberschätzung von Zeigerzielen oder Datenfluss-Beziehungen notwendig.

Gegenüber den ICFG-basierten Verfahren vereinfacht sich die Zeigeranalyse bei uns zu einem Grapherreichbarkeitsproblem, nämlich dem Problem der dynamischen transitiven Hülle. Dies ist das gleiche zentrale Problem wie bei Andersens Originalanalyse; wie im Original dominiert es auch die Kosten der fluss-sensitiven Version.

Ferner haben wir zwei Erweiterungen der neuen Analyse vorgestellt: die Unterstützung starker Aktualisierungen und eine kontext-sensitive Erweiterung im Sinne der MOVP-Problematik. Beide Erweiterungen erhöhen die Präzision und lassen sich auch in Kombination einsetzen. Die Erweiterung für indirekte starke Aktualisierungen benutzte dabei den sogenannten Blockade-Graphen. Dieser erlaubte uns auch die Behandlung zusammengesetzter starker Aktualisierungen ohne asymptotische Mehrkosten.

Um statt der typischen MOP-Lösung die präzisere MOVP-Lösung zu erhalten, haben wir die Propagierung der Zeigerziele angepasst. Unsere Anpassungen beruhten dabei auf dem bekannten IFDS-Framework, können aber kontext-unabhängige und kontext-abhängige Ziele unterscheiden. Mit ein paar Verbesserungen haben wir auf diese Weise schließlich eine fluss-sensitive Zeigeranalyse beschrieben, welche starke Aktualisierungen beherrscht, die MOVP-Präzision erreicht und dabei (nur) die Komplexität $\mathcal{O}(n^4)$ aufweist.

Die Dissertation hat neben der Vorstellung der neuen Analyse, ihrer Erweiterungen und zugehöriger theoretischer Untersuchungen auch über eine empirische Evaluation im Rahmen des Bauhaus-Projekts berichtet. Diese hilft, die praktische Anwendbarkeit sowie den Nutzen der höheren Präzision einschätzen zu können. Neben der theoretischen Komplexität haben wir dadurch gesehen, dass auch die praktische Skalierbarkeit gut ist. So konnten wir z.B. das Programm fvwm mit um die 200 KLoC feld-insensitiv in weniger als fünf Minuten analysieren.

Wir können daher mit dem Fazit schließen, dass fluss-sensitive Verfahren in Zukunft konkurrenzfähiger sein dürften. Als weiteres Fazit wollen wir erwähnen, dass die ganzheitliche Sicht auf die drei zentralen Probleme von Kontrollfluss, Datenfluss und Zeigerzielen zu einer relativ effizienten Lösung geführt hat: der Blick für das große Ganze hilft, auch wenn die Auftrennung in kleinere Teilprobleme natürlich eine beliebte und sinnvolle Strategie bleibt.

15.2 Die neue Analyse im Vergleich

In den einzelnen Kapiteln haben wir bereits die Verwandtschaft sowie die Unterschiede zu bekannten Verfahren diskutiert. Hier wollen wir vor allem die Unterschiede noch einmal übersichtlich zusammenstellen. Dieser Abschnitt erfüllt damit zwei wichtige Aufgaben:

1. Er legt durch einen Vergleich mit der Literatur dar, dass die neue Analyse trotz ihrer einfachen Grundidee tatsächlich neu ist. Dazu zeigt er im Detail die Unterschiede zu bekannten Verfahren und stellt somit den Beitrag der Dissertation heraus.

2. Er rechtfertigt die Beschäftigung mit der Zeiger-Problematik angesichts der Fülle schon existierender Lösungen.

15.2.1 Vergleich zu fluss-insensitiven Verfahren

Gegenüber fluss-insensitiven Zeigeranalysen verfügt unsere Analyse über eine höhere Präzision. Wie wir gezeigt haben, erreichen wir diese bei gleicher

asymptotischer Komplexität, die auch allen gerichteten FICI-Verfahren innewohnt: im schlechtesten Fall kubisch in der Eingabegröße. Die Ursache ist dabei ebenfalls die gleiche, nämlich das Problem der dynamischen transitiven Hülle. Fortschritte für dieses Problem helfen daher beiden Welten.

Wir haben in Abschnitt 4.4 bereits gezeigt, dass unser Ansatz für die kombinierte Analyse als Verallgemeinerung und fluss-sensitive Spezialisierung von Andersens Verfahren verstanden werden kann. Unter den fluss-insensitiven Verfahren ist Andersens Analyse daher unserem Verfahren am nächsten. [Andersen 1994, S. 147f] hat tatsächlich auch bereits kurz betrachtet, ob sich Heintzes Analysetechnik mit Teilmengenbeziehungen dafür eignet, seinen Ansatz fluss-sensitiv auszuprägen. Dies scheiterte jedoch an einigen Problemen. So konnte die sich ergebende Analyse, wie Andersen feststellt, nicht mit mehrstufigen Zeigern (d.h. Zeigern auf Zeiger) umgehen.

Im Gegensatz dazu präsentierte die vorliegende Dissertation eine vollständige fluss-sensitive Ausprägung. Außerdem vermeiden wir es, an jedem Programmpunkt die Wertemenge aller Variablen abzuspeichern, wie es Heintzes Ansatz vornimmt. Stattdessen betrachten wir an jedem Programmpunkt nur die dort tatsächlich gelesenen oder modifizierten Variablen.

Auch der Ansatz von Hasti und Horwitz [1998], durch wiederholte Konstruktion der SSA-Form im Wechsel mit einer fluss-insensitiven Zeigeranalyse letztlich auf FSCI-Präzision zuzustreben, unterscheidet sich deutlich von unserem Vorgehen. Unser Verfahren geht konträr vor: Wir beginnen nicht mit einer Überschätzung der Ziele, deren Ungenauigkeiten wir womöglich nie wieder ausbessern können. Stattdessen arbeiten wir mit Unterschätzungen (im Sinne von: wir ignorieren die Tatsache, dass noch weitere Resultate hinzukommen könnten) und bessern diese bis zu einer konservativen Lösung aus. Das Resultat ist dabei garantiert von fluss-sensitiver Genauigkeit. Hasti und Horwitz haben außerdem keine empirische Daten zu ihrem Ansatz geliefert, und auch die theoretische Auswertung ist äußerst dürftig. Ihr Ansatz scheint zudem nicht in der Lage zu sein, indirekte starke Aktualisierungen oder Kontext-Abhängigkeiten zu beherrschen.

Orthogonal zu unserer Analyse ist die Idee von Kahlon [2008]. Er nutzt eine Kaskade von immer präziser werdenden (fluss-insensitiven) Zeigerana-

lysen, um mit den Resultaten der ungenaueren Analyse die Arbeit der jeweils nächstgenaueren Stufe auf relevante Objekte zu fokussieren und so das Mengengerüst zu reduzieren. Es sollte möglich sein, anstelle der bei Kahlon abschließend genutzten Analyse unsere Analyse (mit Modifikationen) einzusetzen und dank des reduzierten Mengengerüsts z.B. die Zahl der bei inkrementeller Propagierung an einem Knoten potentiell zu speichernden Ziele deutlicher zu begrenzen.

15.2.2 Nutzung des IDFG anstelle des ICFG

Durch die kombinierte Betrachtung und Lösung der Zeiger- und Datenfluss-Analyse entsteht als ein großer Unterschied der Kernanalyse zu anderen fluss-sensitiven Zeigeranalysen, dass der IDFG als Basis benutzt wird, um die Zeigeranalyse darauf als einfaches Datenfluss-Problem zu betrachten. Die bisherigen fluss-sensitiven Ansätze nutzen dagegen – wie für andere Datenfluss-Probleme üblich – den ICFG als Grundlage. Damit leiden sie jedoch unter folgenden Nachteilen im Vergleich zu unserem Vorgehen:

1. Durch starke Aktualisierungen ist eine fluss-sensitive Zeigeranalyse kein distributives Problem. Die klassischen Datenfluss-Frameworks sowie die dahinter stehende Theorie können daher nicht direkt zum Einsatz kommen.

   Im Gegensatz dazu ist die Zeigeranalyse auf dem IDFG in jeder Iteration ein distributives Problem, also zugänglich für Standardverfahren. Mehr noch, die Zeigeranalyse entspricht dem einfachsten Datenfluss-Problem, der Grapherreichbarkeit (d.h. mit der Identität als Transferfunktion). Das hat uns z.B. geholfen, die IFDS-Ideen anzuwenden.

2. Analysen wie [Hind und Pioli 1998] führen die vollständige Menge der an einem ICFG-Knoten gültigen Points-To-Relationen mit. Dies bedeutet eine aufwändigere Datenstruktur, ein großes Mengengerüst und unnötige Operationen (da die meisten Elemente bei der Propagierung zum nächsten ICFG-Knoten unverändert bleiben). Zudem muss an Prozedurgrenzen ausgefiltert werden, was definitiv nicht im Aufrufer bzw.

Aufgerufenen benötigt wird (bei Vorwärtspropagierung; Rückwärtspropagierung scheint in der Literatur generell seltener).

Im Gegensatz dazu interessiert an einem IDFG-Knoten nur die Zeigerzielmenge des Objekts an diesem Knoten. Die Ziele anderer Objekte sind irrelevant und müssen nicht mitgeführt werden. Jeder Knoten und jede Kante des IDFG transportieren nur die tatsächlich relevanten Informationen. Tatsächlich erlaubt die SSA-Eigenschaft überdies, dass wir die Ziele ausschließlich an allgemeinen Definitionen speichern. Im Falle der erschöpfenden Propagierung ist sogar keine Speicherung notwendig, sodass der Speicherbedarf deutlich gesenkt werden kann.

Ein zusätzlicher Vorteil für uns ist es, dass wir den IDFG als weiteres Resultat neben der Zeigeranalyse aufbauen, während ICFG-basierte Zeigeranalysen dies erst noch im Nachhinein erledigen müssen.

Abschnitt 2.3.3.3 hat bereits einige Publikationen aufgeführt, welche vom ICFG als Grundlage abweichen und stattdessen auf einer kompakteren Darstellung operieren [Choi u. a. 1993; Gutzmann u. a. 2007; Lundberg und Löwe 2007; Ruf 1995; Whaley und Lam 2002]. Diese wird jedoch stets im Voraus und daher mit äußerst konservativen Abschätzungen errichtet. Im Gegensatz dazu operieren wir auf dem tatsächlichen, nicht weiter überschätzten IDFG und treiben dessen Bestimmung im Wechsel mit der Zeigeranalyse voran.

Von diesen Verfahren sehen wir Ruf [1995] als den nächsten Verwandten zu unserer Kernanalyse, da er ebenfalls starke Aktualisierungen beherrscht. Außerdem ist dies die einzige uns bekannte frühere Publikation, welche eine kubische Komplexität für eine FSCI-Zeigeranalyse nennt. Ruf bietet jedoch keine Beweise zur Korrektheit und Laufzeitabschätzung; tatsächlich muss er für sein Resultat eine konstante Länge der verwendeten Zugriffspfade annehmen.

Im Lichte dieser Betrachtung zeigt sich, dass unser Vorgehen gegenüber dem Stand der Dinge zwei neue Ansätze umfasst:

1. Nutzung des IDFG anstelle des ICFG als Basis für die Zeigeranalyse.

2. Aufbau des IDFG im Wechsel mit der Zeigeranalyse, so dass keine konservative Vorabschätzung nötig ist.

Während die beiden Aufgaben Zeigeranalyse und IDFG-Konstruktion für sich betrachtet jeweils die konservative Lösung des anderen Problems voraussetzen, erlaubt eine kombinierte Betrachtung die Vermeidung der konservativen Annahmen. Der Korrektheitsbeweis ist dabei zentral für dieses überraschende Resultat. Der Beitrag der Dissertation besteht daher nicht nur aus der Angabe von Beschleunigungs- und Präzisionsverbesserungs-Ideen zu bekannten Ansätzen; vielmehr ist die Verzahnung von IDFG-Konstruktion und Zeigeranalyse ein echter neuer Ansatz.

15.2.3 Nutzung der ISSA-Darstellung

SSA und die interprozedurale Erweiterung ISSA stellen keinen eigenen Beitrag der Dissertation dar, sondern sind bekannte Datenfluss-Darstellungen [Cytron u. a. 1991; Staiger u. a. 2007]. Die Dissertation beschreibt jedoch, wie ein effizienter und präziser inkrementeller Aufbau der ISSA-Form für Programme mit Zeigern geschehen kann.

Zwar wurden SSA-Ideen in der Literatur fluss-sensitiver Zeigeranalysen schon zuvor eingesetzt, jedoch nur als Mittel zur Beschleunigung der Zeigeranalyse an sich, die weiterhin als Datenfluss-Problem auf dem ICFG betrachtet wurde [Chase u. a. 1990; Tok u. a. 2006; Wilson 1997]. Die Beschleunigung resultiert daraus, dass dank der Datenabhängigkeiten die Zahl der Besuche der Grundblöcke reduziert werden kann. Zwar reduziert dies die Laufzeit einer ICFG-basierten Zeigeranalyse, verliert aber die oben genannten Vorteile des IDFG-basierten Ansatzes.

Cytron und Gershbein [1993] beschreiben einen anderen als den von uns genutzten Ansatz, wie eine gegebene Alias-Analyse genutzt werden kann, um inkrementell eine SSA-Form aufzubauen. Sie beschreiben jedoch keine eigene Alias- oder Zeigeranalyse. Unser Vorgehen bei der inkrementellen SSA-Konstruktion ähnelt mehr dem Artikel von Tok u. a. [2006]. Im Unterschied dazu präsentierte die Dissertation jedoch ausführliche Details zu den Aktionen für neue Knoten und behandelt nicht initialisiert erscheinende Variablen effizienter.

15.2.4 Behandlung von starken Aktualisierungen

Fluss-Sensitivität erlaubt im Unterschied zur Fluss-Insensitivität die Erkennung von starken Aktualisierungen. Eine Unterstützung starker Aktualisierungen hilft, die Zeigerziel-Mengen zu verkleinern. Die Erkennung starker Aktualisierungen ist damit ein Beitrag zur Präzisionsverbesserung und kann die Laufzeit und den Platzbedarf reduzieren.

Die Dissertation hat gezeigt, dass die Kernanalyse um die Unterstützung direkter und indirekter starker Aktualisierungen erweitert werden kann. Dies hebt die Analyse bereits von einigen anderen FS-Zeigeranalysen ab [Gutzmann u. a. 2007; Lundberg und Löwe 2007; Zhu 2005]. Die übrigen Analysen mit Unterstützung starker Aktualisierungen realisieren dies über Sonderbehandlungen (aufgrund des Distributivitätsproblems, wenn man auf dem ICFG arbeitet), deren Ziel die verzögerte Bearbeitung indirekter Definitionen ist. Die vorliegende Dissertation hat zu diesem Zweck den Blockade-Graphen eingeführt.

15.2.5 Realisierung der Kontext-Beachtung

Die Dissertation präsentiert als erste Arbeit eine fluss-sensitive Zeigeranalyse mit MOVP-Präzision (in der Propagierung) bei $\mathcal{O}(n^4)$-Komplexität. Landis Analyse hierfür benötigte noch $\mathcal{O}(n^6)$ [Landi und Ryder 1992]. Gutzmann u. a. [2007]; Lundberg und Löwe [2007] erreichen die MOVP-Präzision nicht überall und verlieren an anderen Stellen weiter an Präzision. Die übrigen Publikationen sind exponentiell, z.B. die kontext-sensitive Version aus Ruf [1995].

Unser Vorgehen orientierte sich dabei am bekannten IFDS-Framework [Reps u. a. 1995] und wendet Ideen dieser Technik auf eine fluss-sensitive Zeigeranalyse an. Dies wurde zuvor von anderen auch für Andersens fluss-insensitive Analyse erreicht [Reps 1998, Kapitel 4.4]. Wie [Guyer und Lin 2005, S. 4] jedoch schreibt, setzt IFDS die Distributivität voraus, was mit der ICFG-basierten Sichtweise bei starken Aktualisierungen nicht mehr gegeben ist; dadurch scheint bislang keine Anwendung dieses Frameworks auf eine fluss-sensitive Zeigeranalyse erfolgt zu sein. Wir haben dieses Problem

durch die Nutzung der Datenfluss-Darstellung als Basis umgangen, welche ihrerseits wiederum dank der Kombination der Datenfluss- und Zeigeranalyse zur Verfügung steht.

Wilsons *extended parameters* ähneln unserer am Ende von Kapitel 9 und in Abschnitt 14.1.1 angeführten Idee, einen Eingang als Platzhalter für alle über ihn ankommenden kontext-abhängigen Ziele einzusetzen [Wilson 1997]. Er benutzt jedoch pro Objekt und Funktion höchstens einen extended parameter, während wir pro Eingang einen solchen haben.

15.2.6 Skalierbarkeit

Unsere Analyse ist eine BDD-freie Zeigeranalyse, welche rund 200 KLoC mit fluss-sensitiver Präzision kontext- und feld-insensitiv in weniger als fünf Minuten analysieren kann. Für andere Analysen dieser Präzision wurden bislang nur kleinere Programme erwähnt (Hind und Pioli erreichen nach Hardekopf und Lin [2009b] 30 KLoC, Tok [2007] 70 KLoC in einer halben Stunde.) Darüber hinaus konnten wir die Skalierbarkeit auch theoretisch evaluieren und haben z.B. im Unterschied zu BDD-basierten Verfahren die Garantie, schlechtestenfalls kubisch (kontext-insensitiv) bzw. biquadratisch (mit MOVP-Präzision) zu sein.

Die empirische Auswertung zeigte, dass die kontext-insensitive Analyse in der Praxis eine moderat quadratisch wirkende Komplexität aufweist. Durch verschiedene Propagierungsstrategien sowie Zyklenkontraktion und Fokussierung auf allgemeine Definitionen gelang dabei eine deutliche Beschleunigung. Die erzielte Beschleunigung erreichte teilweise Faktoren größer als 100 und wurde besonders bei größeren Programmen wirksam. Unsere Maßnahmen wirken sich damit erfreulich auf die Skalierbarkeit aus.

Ganz wie es in den letzten Jahren für Andersens Analyse der Fall war, haben wir mit dieser Arbeit als Nebenresultat gezeigt, dass auch im fluss-sensitiven Fall die Zyklenkontraktion einen wichtigen Baustein für eine gute Skalierbarkeit darstellt.

Wir schließen insgesamt, dass sich die kombinierte Sichtweise lohnt: Bei akzeptabler praktischer Skalierbarkeit sowie im Vergleich guter theoretischer

Komplexität haben wir eine im Ansatz einfache Lösung für die miteinander verwobenen Problematiken der Ermittlung von Zeigerzielen, Kontroll- und Datenfluss geschaffen.

Literatur

Die folgenden Werke waren nützliche Quellen für diese Arbeit und enthalten weitergehende Informationen:

[Aho u. a. 1986] AHO, Alfred V. ; SETHI, Ravi ; ULLMANN, Jeffrey D.: *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986

[Allen 1970] ALLEN, Frances E.: Control Flow Analysis. In: *Proceedings of a Symposium on Compiler Optimization*, 1970, S. 1–19

[Andersen 1994] ANDERSEN, Lars O.: *Program Analysis and Specialization for the C Programming Language*, DIKU, University of Copenhagen, Dissertation, Mai 1994

[Bacon und Sweeney 1996] BACON, David F. ; SWEENEY, Peter F.: Fast Static Analysis of C++ Virtual Function Calls. In: *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications.* New York, NY, USA : ACM, 1996, S. 324–341

[Berndl u. a. 2003] BERNDL, Marc ; LHOTÁK, Ondrej ; QIAN, Feng ; HENDREN, Laurie ; UMANEE, Navindra: Points-to Analysis using BDDs. In: *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation.* New York, NY, USA : ACM, 2003, S. 103–114

[Burke u. a. 1994] BURKE, Michael G. ; CARINI, Paul R. ; CHOI, Jong-Deok ; HIND, Michael: Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers. In: *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Springer, 1994, S. 234–250. – Lecture Notes in Computer Science Bd. 892

[Chakaravarthy 2003] CHAKARAVARTHY, Venkatesan T.: New Results on the Computability and Complexity of Points-to Analysis. In: *POPL '03:*

Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA : ACM, 2003, S. 115–125

[Chase u. a. 1990] CHASE, David R. ; WEGMAN, Mark ; ZADECK, F. K.: Analysis of Pointers and Structures. In: *PLDI '90: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 1990, S. 296–310

[Chatterjee 2000] CHATTERJEE, Ramkrishna: *Modular Data-Flow Analyis of Statically Typed Object-Oriented Programming Languages.* New Brunswick, NJ, USA, Rutgers University (State University of New Jersey), Dissertation, Januar 2000

[Chatterjee u. a. 1999] CHATTERJEE, Ramkrishna ; RYDER, Barbara G. ; LANDI, William A.: Relevant Context Inference. In: *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA : ACM Press, 1999, S. 133–146

[Choi u. a. 1993] CHOI, Jong-Deok ; BURKE, Michael ; CARINI, Paul: Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In: *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA, 1993, S. 232–245

[Cooper und Kennedy 1989] COOPER, Keith D. ; KENNEDY, Ken: Fast Interprocedural Alias Analysis. In: *POPL '89: Conference Record of the 16th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* New York, NY, USA : ACM Press, Januar 1989, S. 49 – 59

[Cousot und Cousot 1977] COUSOT, Patrick ; COUSOT, Radhia: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* New York, NY, USA : ACM, 1977, S. 238–252

[Cytron u. a. 1991] CYTRON, Ron ; FERRANTE, Jeanne ; ROSEN, Barry K. ; WEGMAN, Mark N. ; ZADECK, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Oktober, Nr. 4, S. 451–490

[Cytron und Gershbein 1993] CYTRON, Ron ; GERSHBEIN, Reid: Efficient Accommodation of May-Alias Information in SSA Form. In: *PLDI '93: Proceedings of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ACM, 1993, S. 36–45

[Das 2000] DAS, Manuvir: Unification-based pointer analysis with directional assignments. In: *PLDI '00: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM Press, 2000, S. 35–46

[Dean u. a. 1995] DEAN, Jeffrey ; GROVE, David ; CHAMBERS, Craig: Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In: *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*. London, UK : Springer-Verlag, 1995, S. 77–101

[Emami u. a. 1994] EMAMI, Maryam ; GHIYA, Rakesh ; HENDREN, Laurie J.: Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In: *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 1994, S. 242–256

[Erdös und Rényi 1961] ERDÖS, P. ; RÉNYI, A.: On the evolution of random graphs. In: *Bulletin of the International Statistical Institute* 38 (1961), Nr. 4, S. 343–347

[Fähndrich u. a. 1998] FÄHNDRICH, Manuel ; FOSTER, Jeffrey S. ; SU, Zhendong ; AIKEN, Alexander: Partial Online Cycle Elimination in Inclusion Constraint Graphs. In: *SIGPLAN Notices* 33 (1998), Nr. 5, S. 85–96

[Georgiadis 2005] GEORGIADIS, Loukas: *Linear-Time Algorithms for Dominators and Related Problems*, Princeton University, Dissertation, November 2005

[Grove und Chambers 2001] GROVE, David ; CHAMBERS, Craig: A Framework for Call Graph Construction Algorithms. In: *ACM Transactions on Programming Languages and Systems* 23 (2001), Nr. 6, S. 685–746

[Gutzmann u. a. 2007] GUTZMANN, Tobias ; LUNDBERG, Jonas ; LÖWE, Welf: Towards Path-Sensitive Points-to Analysis. In: *SCAM '07: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation*. Los Alamitos, CA, USA, 2007, S. 59–68

[Guyer und Lin 2005] Guyer, Samuel Z. ; Lin, Calvin: Error Checking with Client-Driven Pointer Analysis. In: *Science of Computer Programming* 58 (2005), Nr. 1-2, S. 83–114

[Hardekopf und Lin 2007] Hardekopf, Ben ; Lin, Calvin: The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In: *PLDI '07: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA, 2007, S. 290–299

[Hardekopf und Lin 2009a] Hardekopf, Ben ; Lin, Calvin: Analyzing Millions of Lines of Code with Sparse Flow-Sensitive Pointer Analysis. In: *Papier wurde nur auf der Homepage von B. Hardekopf publiziert*, 2009

[Hardekopf und Lin 2009b] Hardekopf, Ben ; Lin, Calvin: Semi-Sparse Flow-Sensitive Pointer Analysis. In: *POPL '09: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 2009, S. 226–238

[Hasti und Horwitz 1998] Hasti, Rebecca ; Horwitz, Susan: Using Static Single Assignment Form to Improve Flow-insensitive Pointer Analysis. In: *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 1998, S. 97–105

[Hecht 1977] Hecht, Matthew S.: *Flow Analysis of Computer Programs*. New York, NY, USA : Elsevier Science Inc., 1977. – ISBN 0444002162

[Heintze und Tardieu 2001] Heintze, Nevin ; Tardieu, Olivier: Ultra-fast Aliasing Analysis Using CLA: a Million Lines of C Code in a Second. In: *PLDI '01: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2001, S. 254–263

[Hind 2001] Hind, Michael: Pointer Analysis: Haven't We Solved This Problem Yet? In: *PASTE '01: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Snowbird, UT, USA : ACM Press, 2001, S. 54–61

[Hind u. a. 1999] Hind, Michael ; Burke, Michael ; Carini, Paul ; Choi, Jong-Deok: Interprocedural Pointer Alias Analysis. In: *ACM Transactions on Programming Languages and Systems* 21 (1999), Nr. 4, S. 848–894

[Hind und Pioli 1998] Hind, Michael ; Pioli, Anthony: Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In: *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*. London, UK : Springer-Verlag, 1998, S. 57–81. – Erweiterte Fassung des SAS-Beitrags aus Lecture Notes in Computer Science Bd. 1503

[Hind und Pioli 2000] Hind, Michael ; Pioli, Anthony: Which Pointer Analysis Should I Use? In: *ISSTA '00: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA : ACM, 2000, S. 113–123

[Horwitz 1997] Horwitz, Susan: Precise Flow-Insensitive May-Alias Analysis is NP-hard. In: *ACM Transactions on Programming Languages and Systems* 19 (1997), Nr. 1, S. 1–6

[Horwitz u. a. 1988] Horwitz, Susan ; Reps, Thomas ; Binkley, David: Interprocedural Slicing Using Dependence Graphs. In: *PLDI '88: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 1988, S. 35–46

[Kahlon 2008] Kahlon, Vineet: Bootstrapping: a Technique for Scalable Flow and Context-Sensitive Pointer Alias Analysis. In: *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2008, S. 249–259

[Kam und Ullman 1977] Kam, John B. ; Ullman, Jeffrey D.: Monotone Data Flow Analysis Frameworks. In: *Acta Informatica* 7 (1977), S. 305–317

[Kildall 1973] Kildall, Gary A.: A Unified Approach to Global Program Optimization. In: *POPL '73: Conference Record of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA : ACM Press, Oktober 1973, S. 194 – 206

[Knoop u. a. 1998] Knoop, Jens ; Koschützki, Dirk ; Steffen, Bernhard: Basic-Block Graphs: Living Dinosaurs? In: *CC '98: Proceedings of the 7th International Conference on Compiler Construction*. London, UK : Springer-Verlag, 1998, S. 65–79

[Knoop und Steffen 1992] Knoop, Jens ; Steffen, Bernhard: The Interprocedural Coincidence Theorem. In: *CC '92: Proceedings of the 4th International Conference on Compiler Construction*. London, UK : Springer-Verlag, 1992, S. 125–140

[Koschke 2008] Koschke, Rainer: Zehn Jahre WSR - Zwölf Jahre Bauhaus. In: *Proceedings of the Workshop Software Reengineering*. Bad Honnef : GI, 2008, S. 51–65. – LNI Proceedings Bd. P-126

[Landi und Ryder 1992] Landi, William ; Ryder, Barbara G.: A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In: *PLDI '92: Proceedings of the Conference on Programming Language Design and Implementation*. New York, NY : ACM Press, 1992, S. 235–248

[Landi 1992] Landi, William A.: *Interprocedural Aliasing in the Presence of Pointers*. New Brunswick, NJ, USA, Rutgers University (State University of New Jersey), Dissertation, Januar 1992

[Lattner u. a. 2007] Lattner, Chris ; Lenharth, Andrew ; Adve, Vikram: Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In: *PLDI '07: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, California, Juni 2007

[Lhotak und Hendren 2006] Lhotak, Ondrej ; Hendren, Laurie: Context-Sensitive Points-to Analysis: Is It Worth It? In: *CC '06: Proceedings of the Conference on Compiler Construction*, Springer, 2006, S. 47–64. – LNCS Bd. 3923

[Liao 2000] Liao, Shih-Wei: *SUIF Explorer: An Interactive and Interprocedural Parallelizer*, Stanford University, Dissertation, 2000

[Lindenmaier u. a. 2005] Lindenmaier, Götz ; Beck, Michael ; Boesler, Boris ; Geiss, Rubino: Firm, an Intermediate Language for Compiler Research. University of Karlsruhe, Mai 2005 (2005-8). – Forschungsbericht. – 19 S

[Lundberg und Löwe 2007] Lundberg, Jonas ; Löwe, Welf: A Scalable Flow-Sensitive Points-to Analysis. In: *Compiler Construction - Advances and Applications, Festschrift on the occasion of the retirement of Prof. Dr. Dr. h.c. Gerhard Goos*, Springer Verlag, 2007. – URL http://w3.msi.vxu.se/~jonasl/goos07.pdf. – Bislang nur elektronisch verfügbar

[Marlowe und Ryder 1990] Marlowe, T. J. ; Ryder, B. G.: Properties of Data Flow Frameworks: a Unified Model. In: *Acta Informatica* 28 (1990), Nr. 2, S. 121–163

[Melski und Reps 2000] MELSKI, David ; REPS, Thomas: Interconvertibility of a Class of Set Constraints and Context-Free-Language Reachability. In: *Theoretical Computer Science* 248 (2000), Nr. 1-2, S. 29–98

[Milanova u. a. 2004] MILANOVA, Ana ; ROUNTEV, Atanas ; RYDER, Barbara G.: Precise Call Graphs for C Programs with Function Pointers. In: *Automated Software Engineering* 11 (2004), Nr. 1, S. 7–26

[Milanova u. a. 2005] MILANOVA, Ana ; ROUNTEV, Atanas ; RYDER, Barbara G.: Parameterized Object Sensitivity for Points-to Analysis for Java. In: *ACM Transactions on Software Engineering and Methodology* 14 (2005), Nr. 1, S. 1–41

[Muchnick 1997] MUCHNICK, Steven S.: *Advanced Compiler Design and Implementation*. San Francisco : Morgan Kaufmann Publishers, 1997

[Nielson u. a. 1999] NIELSON, Flemming ; NIELSON, Hanne R. ; HANKIN, Chris: *Principles of Program Analysis*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1999. – ISBN 3540654100

[Pearce 2005] PEARCE, David J.: *Some Directed Graph Algorithms and Their Application to Pointer Analysis*, University of London, Dissertation, Februar 2005

[Pearce u. a. 2007] PEARCE, David J. ; KELLY, Paul H. ; HANKIN, Chris: Efficient Field-Sensitive Pointer Analysis of C. In: *ACM Transactions on Programming Languages and Systems* 30 (2007), Nr. 1. – Artikel 4

[Plödereder 1980] PLÖDEREDER, Erhard Otto J.: *A Semantic Model for the Analysis and Verification of Programs in General, Higher-Level Languages*. Cambridge, MA, USA, Harvard University, Dissertation, Januar 1980

[Ramalingam 1994] RAMALINGAM, G.: The Undecidability of Aliasing. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), Nr. 5, S. 1467–1471

[Raza u. a. 2006] RAZA, Aoun ; VOGEL, Gunther ; PLÖDEREDER, Erhard.: Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In: *Reliable Software Technologies, Ada Europe 2006 (LNCS 4006)* (2006), S. 71–82

[Reps 1998] REPS, Thomas: Program Analysis via Graph Reachability. In: *Information and Software Technology* 40 (1998), Dezember, Nr. 11-12, S. 701–726

[Reps u. a. 1995] Reps, Thomas ; Horwitz, Susan ; Sagiv, Mooly: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1995, S. 49–61

[Reps u. a. 1994] Reps, Thomas ; Sagiv, Mooly ; Horwitz, Susan: Interprocedural Dataflow Analysis via Graph Reachability. University of Copenhagen : University of Copenhagen, April 1994 (TR 94-14). – Forschungsbericht. – 51 S

[Rice 1953] Rice, Henry G.: Classes of Recursively Enumerable Sets and Their Decision Problems. In: *Transactions of the American Mathematical Society* 74 (1953), März, Nr. 2, S. 358–366

[Rosen 1979] Rosen, Barry K.: Data Flow Analysis for Procedural Languages. In: *Journal of the ACM* 26 (1979), Nr. 2, S. 322–344

[Rountev u. a. 2001] Rountev, Atanas ; Milanova, Ana ; Ryder, Barbara G.: Points-to Analysis for Java Using Annotated Constraints. In: *SIGPLAN Notices* 36 (2001), Nr. 11, S. 43–55

[Rountev u. a. 1999] Rountev, Atanas ; Ryder, Barbara G. ; Landi, William: Data-Flow Analysis of Program Fragments. In: *Foundations of Software Engineering* (1999), S. 235 – 252

[Ruf 1995] Ruf, Erik: Context-insensitive Alias Analysis Reconsidered. In: *PLDI '95: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY, USA : ACM, 1995, S. 13–22

[Ryder 2003] Ryder, Barbara G.: Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages. In: *CC '03: Proceedings of the 12th International Conference on Compiler Construction*, Springer, 2003, S. 126–137. – Lecture Notes in Computer Science Bd. 2622

[Sagiv u. a. 1996] Sagiv, Mooly ; Reps, Thomas ; Horwitz, Susan: Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In: *TAPSOFT '95: Selected papers from the 6th International Joint Conference on Theory and Practice of Software Development*, Elsevier Science Publishers B. V., 1996, S. 131–170

[Shapiro und Horwitz 1997] Shapiro, Marc ; Horwitz, Susan: Fast and Accurate Flow-Insensitive Points-to Analysis. In: *POPL '97: Proceedings*

of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA : ACM, 1997, S. 1–14

[Sharir und Pnueli 1981] SHARIR, M. ; PNUELI, A.: Two Approaches to Interprocedural Data Flow Analysis. In: MUCHNICK, S. (Hrsg.) ; JONES, N. (Hrsg.): *Program Flow Analysis: Theory and Applications.* Prentice Hall, 1981, S. 189–233

[Shivers 1991] SHIVERS, Olin G.: *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*, Carnegie Mellon University, Dissertation, Mai 1991

[Singer 2005] SINGER, Jeremy: *Static Program Analysis based on Virtual Register Renaming*, University of Cambridge, Dissertation, März 2005

[Sridharan und Bodík 2006] SRIDHARAN, Manu ; BODÍK, Rastislav: Refinement-based Context-Sensitive Points-to Analysis for Java. In: *SIGPLAN Notices* 41 (2006), Nr. 6, S. 387–400

[Staiger 2007] STAIGER, Stefan: Reverse Engineering of Graphical User Interfaces Using Static Analyses. In: *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, IEEE Computer Society, Oktober 2007, S. 189–198

[Staiger u. a. 2007] STAIGER, Stefan ; VOGEL, Gunther ; KEUL, Steffen ; WIEBE, Eduard: Interprocedural Static Single Assignment Form. In: *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering.* Los Alamitos, CA, USA : IEEE Computer Society, Oktober 2007, S. 1 – 10

[Staiger-Stöhr 2009] STAIGER-STÖHR, Stefan: *Messwerte zur Dissertation.* 2009. – URL http://www.informatik.uni-stuttgart.de/iste/ps/bauhaus/papers/index.html

[Steensgaard 1996] STEENSGAARD, Bjarne: Points-to Analysis in Almost Linear Time. In: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA : ACM Press, 1996, S. 32–41

[Stocks u. a. 1998] STOCKS, Philip A. ; RYDER, Barbara G. ; LANDI, William A. ; ZHANG, Sean: Comparing Flow and Context Sensitivity on the Modification-Side-Effects Problem. In: *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis.* New York, NY, USA : ACM, 1998, S. 21–31

[Tip und Palsberg 2000] TIP, Frank ; PALSBERG, Jens: Scalable Propagation-Based Call Graph Construction Algorithms. In: *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA : ACM, 2000, S. 281–293

[Tok 2007] TOK, Teck B.: *Removing Unimportant Computations in Interprocedural Program Analysis*, University of Texas, Dissertation, August 2007

[Tok u. a. 2006] TOK, Teck B. ; GUYER, Samuel Z. ; LIN, Calvin: Efficient Flow-Sensitive Interprocedural Data-flow Analysis in the Presence of Pointers. In: *CC '06: Proceedings of the 15th International Conference on Compiler Construction*, Springer, 2006, S. 17–31

[Whaley 2007] WHALEY, John: *Context-Sensitive Pointer Analysis Using Binary Decision Diagrams*. Stanford, CA, USA, Stanford University, Dissertation, März 2007

[Whaley und Lam 2002] WHALEY, John ; LAM, Monica S.: An Efficient Inclusion-Based Points-to Analysis for Strictly-Typed Languages. In: *SAS '02: Proceedings of the International Symposium on Static Analysis*, 2002, S. 180–195. – LNCS Bd. 2477

[Whaley und Lam 2004] WHALEY, John ; LAM, Monica S.: Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In: *PLDI '04: Proceedings of the Conference on Programming Language Design and Implementation*, ACM Press, Juni 2004, S. 131 – 144

[Whaley und Rinard 1999] WHALEY, John ; RINARD, Martin: Compositional Pointer and Escape Analysis for Java Programs. In: *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA : ACM, 1999, S. 187–206

[Wilson und Lam 1995] WILSON, Robert P. ; LAM, Monica S.: Efficient Context-Sensitive Pointer Analysis for C Programs. In: *PLDI '95: Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM Press, 1995, S. 1–12

[Wilson 1997] WILSON, Robert P.: *Efficient, Context-Sensitive Pointer Analysis for C Programs*, Stanford University, Dissertation, Dezember 1997

[Zhu 2002] ZHU, Jianwen: Symbolic Pointer Analysis. In: *ICCAD '02: Proceedings of the IEEE/ACM International Conference on Computer-aided Design.* New York, NY, USA : ACM, 2002, S. 150–157

[Zhu 2005] ZHU, Jianwen: Towards Scalable Flow and Context Sensitive Pointer Analysis. In: *DAC '05: Proceedings of the 42nd Conference on Design Automation.* New York, NY, USA : ACM, 2005, S. 831–836

[Zhu und Calman 2004] ZHU, Jianwen ; CALMAN, Silvian: Symbolic Pointer Analysis Revisited. In: *PLDI '04: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY, USA : ACM, 2004, S. 145–157

Stichwortverzeichnis

0-Definition, 176
1-Definition, 176

Andersen, 22, 46, 105
Aufrufgraph, 32
Ausgang, 86

BDDs, 58
Blockade-Graph, 184

CFG, 32
CI-Graph, 234
Constraint-Graph, 46
CS-Graph, 235
CSU, 175

Datenfluss-Problem
- distributives, 37
- monotones, 37
- Zeigeranalyse als, 38, 49, 223

Definition, 35
- allgemeine, 90, 135

Dominanz, 41, 122

Eingang, 86
Exceptions, 33, 394

Feld-Sensitivität, 44, 67
Fluss-insensitive Analyse, 45, 166
Fluss-Sensitivität, 43, 48, 107, 111

Gültige Definition
- Suche, 127

Graphaufbau, 114

Heap, 66, 82, 242, 329

Hilfsvariable, 67

ICFG, 32
IDFG, 35
IFDS, 209
Instruktionen, 78
ISSA, 42, 83
- erweiterter Graph, 219

Kernanalyse, 21, 95
Kombination, 21, 26, 96
Konfluenzoperator, 37

LHS, 35

MOP, 38
MOVP, 38, 57, 208

Objektarten, 69
Objekte, 65
- nicht initialisierte, 82

Objektgraph, 93, 119, 122
Objektgruppe, 68

Parallele Programme, 395
Pfad
- balancierter, 207
- gültiger, 207
- kontext-abhängiger, 207
- kontext-unabhängiger, 207

Propagierung, 144
- erschöpfende, 139, 144
- inkrementelle, 139, 145, 237

Propagierungsgraph, 135
Propagierungsrichtung, 138

Pruning, 42, 157, 333

Rekursive Unterprogramme, 161
RHS, 35

Same-Level-Pfad, 213
SSA, 41
SSU, 174
starke Aktualisierung, 35, 325
  einfache, 174
  und Distributivität, 49, 226
  zusammengesetzte, 175

Tabulationsverfahren, 213, 223
These 1: Kubische fluss-sensitive Andersen-Analyse, 23, 265
These 2: Fluss-sensitive Analyse mit MOVP-Präzision, 24, 272
These 3: Indirekte starke Aktualisierungen, 24, 267
These 4: Kombination von Zeiger- und Datenfluss-Analyse, 26, 257
These 5: Zeigeranalyse auf Datenfluss-Darstellung, 26, 265
Transferfunktion, 37, 223

Unterausgang, 86
Untereingang, 86

Verwendung, 35
  allgemeine, 90

Zeigerliteral, 71
Zeigerziel
  aktuelles, 71
Zeigerziel-Quelle, 71
Zyklenkontraktion, 135
  Algorithmus, 141