Institut für Softwaretechnologie
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2707

Statische Analyse von Programmen mit Bibliotheken

Minh Cuong Tran

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Erhard Plödereder
Betreuer:	Dipl.-Inf. Stefan Staiger
begonnen am:	02. Januar 2008
beendet am:	03. Juli 2008
CR-Klassifikation:	D.3.4, F.3.2

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Software-Reengineering beschäftigt sich mit der Frage, wie aus einem existierenden Quellcode wieder Wissen gewonnen werden kann. Das Wissen hilft einem Wartungsingenieur. Er versteht dadurch das ihm unbekannte System besser und kann Fehler beheben oder Änderungen durchführen. Nach einer empirischen Untersuchung von Arthur [Art88] liegt der Aufwand für die Wartung zwischen 60–80 % im gesamten Lebenszyklus einer Software, von der Erstentwicklung bis zur Ersetzung. Fjedstad und Hamlen [FH79] haben die Wartungsarbeiten empirisch untersucht und sind zu dem Ergebnis gekommen, dass ein Wartungsingenieur ca. 50 % seiner Zeit damit beschäftigt ist, das System zu analysieren und zu verstehen. Diese zwei Zahlen spiegeln wider, welch wichtige Rolle Software-Reengineering im Forschungsbereich spielt.

In der Softwareentwicklung werden Komponenten, die häufig wiederverwendet werden, in Bibliotheken ausgelagert. Benutzerprogramme[1] sprechen eine von der Bibliothek definierte Schnittstelle an und nutzen auf diese Weise Funktionen, die Bibliotheken anbieten.

Statische Programmanalyse beschäftigen sich mit der Methodik Programme automatisch und ohne ihre Ausführung zu analysieren.

Software-Reengineering benutzt statische Programmanalyse, um Wissen aus dem Quellcode zu gewinnen. Leider gehen viele statische Programmanalysen davon aus, dass das zu analysierende Programm samt den benutzten Bibliotheken oder ohne Benutzung einer Bibliothek als eine Einheit vorliegt. Diese Annahme ist oft falsch. Ein Beispiel und ein wenig Statistik sollen diese Annahme widerlegen. Die Statistik gibt auch die Größenverhältnisse zwischen Benutzerprogramm und Bibliothek und die Häufigkeit der Benutzung der Bibliotheken wieder, die als Motivation gesehen werden soll.

Als Beispiel wird der Archiv-Manager File Roller[2] in Abbildung 1.1 genannt. Wie zu sehen ist, ist File Roller ein Programm mit grafischer Benutzeroberfläche. File Roller unterstützt das Komprimieren und Dekomprimieren von verschiedenen Kompressionsarten. Für die grafische Benutzeroberfläche wird die Bibliothek gtk[3] benutzt, für das (De-)Komprimieren greift File Roller auf Bibliotheken für viele

[1] Englisch: client. Der Name wurde gewählt, da ein Programmierer als Benutzer einer Bibliothek Programme schreibt.

[2] http://fileroller.sourceforge.net/ Zuletzt besucht am 17. Juni 2008.

[3] http://www.gtk.org — Gimp Tool Kit ist eine Bibliothek für grafische Benutzeroberflächen in C. Zuletzt besucht am 17. Juni 2008.

verschiedene Kompressionsarten zurück. Ein Paket-Manager gibt an, dass File Roller insgesamt 30 Bibliotheken braucht, um laufen zu können.

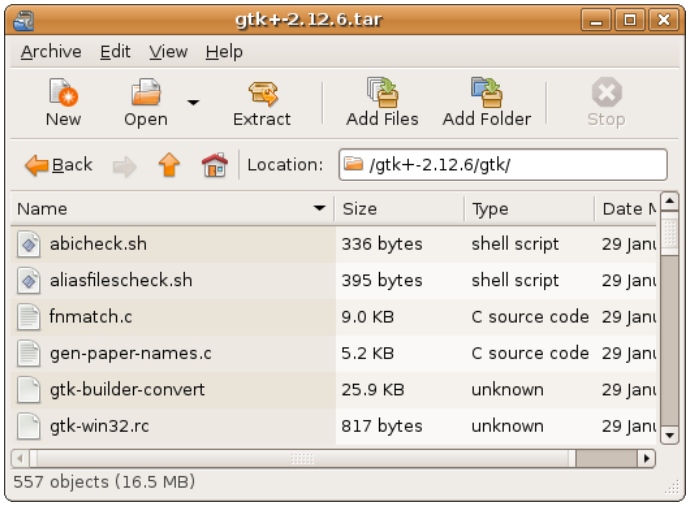

Abbildung 1.1: Screenshot vom Archiv-Manager File Roller

Tabelle 1.1 gibt eine kleine empirische Untersuchung über die Anzahl der installierten Bibliotheken auf Rechnern wieder, die im Umfeld des Autors eingesetzt werden. Selbst der mir bekannte Rechner svw.info als reiner Webserver ohne grafische Anwendungen besitzt 301 Bibliotheken, um seine Aufgabe zu erledigen.

Name	Anzahl	Betriebssystem
marvin	2862	Ubuntu 7.10
phileus	2477	Ubuntu 8.04
pslx1	2194	Debian
svw.info	301	Debian Webserver

Tabelle 1.1: Anzahl der Bibliotheken auf laufenden Systemen

Interessant ist auch das Größenverhältnis vom Benutzerprogramm zu der benutzten Bibliothek, welches sich in Tabelle 1.2 und 1.3 in Form der Anzahl von Quellcodezeilen (SLOC[4]) widerspiegelt. Die aufgelisteten Programme benutzen entweder gtk oder Qt[5] für die grafische Benutzeroberfläche. Der Anteil der Bibliotheken überwiegt in den meisten Fällen. Hinzu kommt die transitive Abhängigkeit von Bibliotheken, denn gtk benutzt als Bibliothek gdk.

Sowohl die Größenverhältnisse als auch die Häufigkeit der Benutzung einer Bibliothek, gekoppelt mit der Tatsache, dass viele statische Programmanalysen davon ausgehen, ein Programm als eine Einheit zu analysieren, geben Anlass und Mo-

[4]Englische Abkürzung für source line of code
[5]http://trolltech.com/products/qt/ — Qt ist eine Bibliothek für grafische Benutzeroberflächen in C++, die C++ und Java unterstützen kann. Zuletzt besucht am 17. Juni 2008.

Name	SLOC in k
gdk	87
gtk	328
Qt	274

Tabelle 1.2: Anzahl der Quellcodezeilen der Bibliotheken

gtk		Qt	
Name	SLOC in k	Name	SLOC in k
bluefish	44	JLRFractal	2,5
codebreaker	1,9	qtads	77
euler	24	roadmap	32
gimp	590	SGrViewer	36
gqview	54	spacehulk	27

Tabelle 1.3: Anzahl der Quellcodezeilen der Benutzerprogramme, die gtk und Qt benutzen

tivation genug, sich mit der statischen Programmanalyse mit Bibliotheken zu beschäftigen.

1.2 Überblick

In diesem Abschnitt wird die Aufgabenstellung erläutert. Die Gliederung gibt einen kurzen Überblick über die Arbeit. Anschließend werden die Konventionen angesprochen, unter denen die Arbeit geschrieben wurde.

1.2.1 Aufgabenstellung

Ziel dieser Diplomarbeit ist der Umgang mit statischen Analysen von Programmen, die an Bibliotheken angebunden sind. Die Arbeit gliedert sich in drei Teilaufgaben.

Theoretische Untersuchung

Folgende Strategien sollen (z. B. hinsichtlich Korrektheit und Genauigkeit, Aufwand für Anwender und Analysenprogrammierer, Zeit- und Speicherbedarf) verglichen werden:

- Ignorieren der (Effekte von) Funktionen aus der Bibliothek
- Ohne Voranalyse den schlimmsten Fall für Bibliotheksfunktionen annehmen

- Manuelle Erstellung von Attrappen für relevante Bibliotheksfunktionen
- Analyse mit gesamter dazu gebundener Bibliothek
- Fragment-Analyse nach Rountev [RR01] [Rou02]
- Nutzung der Rollenpropagierung nach Staiger [Sta07a]
- Component-Level-Analysis nach Rountev [RKM06]

Diese Strategien sollen speziell im Hinblick auf eine Nutzung für die folgenden Analysen verglichen werden: Aufrufgraphkonstruktion, Seiteneffektberechnung, GUI-Analyse[6], sowie optional auch für Zeigeranalysen und ISSA-Analyse[7]. Dazu sollen die Strategien jeweils generell sowie speziell im Kontext dieser Analysen beschrieben werden.

Implementierung

Nach der theoretischen Untersuchung soll eine geeignete Strategie ausgewählt und implementiert werden. Für geeignete Analysen soll die vorhandene Implementierung der Rollenpropagierung für die Voranalyse von Bibliotheken verwendet und erweitert werden, sodass die Rollen der Schnittstellenfunktionen einer Bibliothek erkannt werden. Das Ergebnis soll geeignet gespeichert und in Gesamtprogramm-Analysen verwendet werden können. Die Implementierung muss mit C- und C++-Programmen zurecht kommen[8].

Evaluation

Die Implementierung muss anschließend anhand der oben genannten Analysen evaluiert werden. Dazu sind eine Reihe von Testprogrammen auszuwählen (sowohl kleine Beispiele zur Überprüfung der Korrektheit als auch reale, größere Programme zur Überprüfung der Skalierbarkeit und zum Test in realen Szenarien. Weiterhin ist ein Vergleich mit der Analyse des gesamten Programms (inklusive Bibliothek) durchzuführen.

1.2.2 Gliederung der Arbeit

Die Diplomarbeit gliedert sich in sechs Hauptkapitel auf.

In Kapitel 2 werden die nötigen Grundlagen der statischen Programmanalyse im Zusammenhang mit dieser Diplomarbeit erläutert. Es wird kurz auf das Umfeld im Bauhausprojekt sowie die Zwischendarstellung IML eingegangen.

[6] Englisch: graphical user interface — grafische Benutzeroberfläche oder kurz GUI

[7] Interprocedural Static Single Assignment

[8] In Absprache mit dem Betreuer wurde aus Zeitgründen bei der Implementierung die Behandlung von C++-Programmen, speziell die Objektorientierung (mit Klassen als Schnittstellen von Bibliotheken) sowie Ausnahmebehandlung, beiseite gelassen. Die Beschreibung dieser Implementierung sowie die Diskussion bei auftretenden Schwierigkeiten finden sich jedoch in dieser Arbeit wieder.

Die theoretische Untersuchung als ein Teil der Aufgabe wird in Kapitel 3 besprochen. Es wird zuerst auf generelle Schwierigkeiten einer getrennten Analyse hingewiesen. Darauf aufbauend wird erklärt, warum einfache Ansätze unpräzise oder ungenau sind. Anschließend werden die Ansätze Fragment-Analyse und die Component-Level-Analysis ausführlich besprochen und verwandte Arbeiten angerissen. Eine Klassifikation sowie ein Vergleich der Ansätze werden vollzogen und die Entscheidung zur Fragment-Analyse wird anschließend begründet.

Die Erweiterung der Fragment-Analyse wird im Kapitel 4 angesprochen, sie erzielt eine höhere Genauigkeit. Besprochen wird ebenfalls ein spezieller Ansatz für C++ und Objektorientierung, in der Klassen als Schnittstellen von Bibliotheken angeboten werden.

Der Ablauf der Fragment-Analyse und darauf aufbauende Analysen werden in Kapitel 5 erläutert. Es wird kurz eine verwandte Arbeit vorgestellt. Hinzu kommen die Darstellung der Kompaktifizierung und ihre Verwendung.

Kapitel 6 bespricht im Detail die Umsetzung der Fragment-Analyse mit anschließender Kompaktifizierung. Datenstrukturen, Komplexität sowie Implementierungsdetails werden vorgestellt.

Die Evaluation des umgesetzten Ansatzes findet sich in Kapitel 7 wieder.

1.2.3 Konventionen

Früher wäre es nicht nötig gewesen, Konventionen beim Schreiben zu erwähnen, da Konventionen eine Allgemeingültigkeit besitzen. Im Zuge der (mehrmals korrigierten) Rechtschreibreform sowie der weit verbreiteten Anglizismen in der Informatik soll hier kurz erwähnt werden, dass diese Arbeit in der neuen Rechtschreibung geschrieben wurde. Englische Begriffe wurden, soweit möglich, durch existierende deutsche Begriffe ersetzt. Englische Titel wurden, falls es sinnvoll erschien, durch deutsche ersetzt. Beim erstmaligen Auftreten des deutschen Pendants wurde der englische Begriff/Titel angegeben. Falls dies nicht sinnvoll war, wurde der englische Begriff/Titel beibehalten.

Da die untersuchten Programme in C/C++ vorlagen und die Implementierung mit diesen zurechtkommen sollte, wurde bei der Angabe eines untersuchten Beispielprogramms stets die C/C++-Syntax vorgezogen. Die Implementierung erfolgte in Ada, Pseudocode-Angaben erfolgten in einer Ada-ähnlichen Syntax.

Der Leser sollte eine Informatik-Grundbildung besitzen und mit statischen Programmanalysen ein wenig vertraut sein, um die Arbeit umfassend verstehen und beurteilen zu können.

2 Grundlagen

In diesem Kapitel wird dem Leser ein Überblick über das Umfeld der Arbeit geboten. Es werden einige Grundlagen erklärt, die für die statische Programmanalyse benötigt werden und insbesondere im Rahmen dieser Diplomarbeit immer wieder auftauchen.

2.1 Bauhaus

Das Forschungsprojekt Bauhaus [RVP06] entstand 1996 und ist eine Zusammenarbeit der Abteilung Programmiersprachen und Übersetzerbau der Universität Stuttgart, der Arbeitsgruppe Softwaretechnik der Universität Bremen sowie der Axivion GmbH. Ziel der Zusammenarbeit ist die Erforschung und Entwicklung von Werkzeugen, welche die Software-Wartung, das Software-Reengineering sowie das Programmverstehen unterstützen sollen. Die Architekturerkennung im Großen, das Erkennen von Klonen und das Erstellen von Softwaremetriken, um nur einige zu nennen, sind Arbeitsfelder, bei denen Bauhaus den Analysenanwender mit Werkzeugen unterstützen kann.

Die Zwischendarstellung IML[1] ist der Kern des Bauhaus-Projekts. Sie wurde speziell für das Software-Reengineering entworfen [KG98]. Es existieren mächtige Frontends z. B. für die Sprachen Ada, C/C++ und Java, die den Quellcode in IML umwandeln können. Die Abstraktion von IML für die genannten Sprachen spielt eine große Rolle — IML ist somit sprachübergreifend. Für allgemeine Konstrukte gibt es Klassen, durch Vererbung können jedoch auch sprachspezifische Eigenheiten dargestellt werden. IML enthält einen Syntaxbaum, der ähnlich wie bei der herkömmlichen Kompilierung entsteht. Darüber hinaus enthält IML Kanten z. B. zwischen der Benutzung einer Variablen und ihrer Definition. IML ist also ein Graph. Zusätzlich speichern auf IML basierende Analysen ihre Ergebnisse ebenfalls in IML ab. IML versucht den Spagat zwischen Abstraktion und möglichst großer Quellcodenähe. Dieser Versuch gelingt, wie z. B. bei Makros in C/C++, nicht immer. Datenstrukturen für IML-Knoten wie Listen, Mengen, Hashtabellen etc., sowie Funktionen zur Traversierung des Syntaxbaumes oder zum Besuchen von bestimmten Knotentypen, existieren ebenfalls im Projekt Bauhaus.

Auf Basis von IML existieren in Bauhaus eine Reihe von Werkzeugen für Zeiger-, Daten- und Kontrollflussanalysen sowie Aufrufgraphkonstruktionen und die Erzeugung der SSA-/ISSA-Form. Auf diese Analysen wird im Folgenden kurz eingegangen.

[1] Abkürzung für InterMediate Language

2.2 Verschiedene Analysen

2.2.1 Zeigeranalysen

In C/C++ gibt es einen speziellen Datentyp Zeiger der Form `Datentyp *zeigervariable`, die auf `Datentyp` zeigen kann und die Adresse des Zieles enthält. Statische Zeigeranalysen beantworten die Frage, auf welche Objekte ein Zeiger zur Laufzeit zeigen kann. Bei Objekten handelt es sich im Allgemeinen um Daten, aber auch um Programmquellcode im Hauptspeicher. Für die Ziele kommen typischerweise Variablen auf dem Stack oder auf der Halde, aber auch Adressen von Funktionen in Frage. Letzteres ist speziell für den Aufrufgraphen interessant. In C/C++ gibt es Zeiger auf Funktionen und indirekte Aufrufe dieser Funktionen über Zeiger. Hierzu folgt ein kurzes Beispiel:

Beispiel

```
1  int zwei_x (int x) {
2      return 2*x; }
3
4  int x_hoch_zwei (int x) {
5      return pow(x, 2); }
6
7  int main () {
8      int a = 10;
9      int b = 20;
10     int *p_int;
11     int (*fp) (int);
12
13     if (...) {
14         p_int = &a;
15         fp = x_hoch_zwei;
16     } else {
17         p_int = &b;
18         fp = zwei_x; }
19
20     return (*fp) (*p_int);
21 }
```

Nach der `if-else`-Bedingung kann der Zeiger `p_int` auf `a` oder auf `b` zeigen, der Funktionszeiger `fp` auf die Funktionen `zwei_x` oder `x_hoch_zwei`.

Eine datenflusssensitive Analyse beachtet bei der Berechnung Zuweisungen zwischen Zeigern und deren Zielen, eine datenflussinsensitive Analyse dagegen nicht, sondern sie versucht beispielsweise über die Vererbungshierarchie die potenziellen Ziele zu berechnen.

Eine kontrollflussinsensitive Analyse unterscheidet nicht, welche Verzweigung genommen wurde und nimmt an, dass am Ende der Funktion `main` `p_int` sowohl auf `a` als auch auf `b` zeigen kann, eine kontrollflusssensitive Analyse dagegen unterscheidet die Verzweigungen.

Eine Funktion kann von mehreren Punkten aufgerufen werden. Falls eine Zeigeranalyse bei der Untersuchung einer Funktion zwischen den verschiedenen Aufrufkontexten und Aufrufpfaden der Funktion unterscheidet, so ist die Analyse kontextsensitiv, andernfalls kontextinsensitiv.

In Bauhaus existieren bereits Zeigeranalysen nach Andersen [And94, Fro06], Das [Das00], Steensgaard [Ste96] und Wilson [WL95]. Die Analysen von Andersen, Das und Steensgaard sind datenflusssensitiv, kontrollfluss- und kontextinsensitiv. Die Analyse von Wilson dagegen ist datenfluss-, kontrollfluss- und kontextsensitiv. Daher ist die Wilson-Analyse am präzisesten, absteigend gefolgt von Andersen, Das und Steensgaard [Fro06].

2.2.2 Aufrufgraph

Ein Aufrufgraph ist ein Graph mit Knoten als Funktionen im Quellcode. Eine Kante von Funktion `f` zu Funktion `g` existiert, falls `f` `g` aufrufen kann. Es handelt sich um einen Multigraphen, das heißt, es können zwischen zwei Knoten mehrere Kanten existieren, da `f` `g` an verschiedenen Stellen aufrufen kann. Die Abbildung 2.1 stellt den Aufrufgraphen aus dem Beispiel des vorherigen Abschnitts dar.

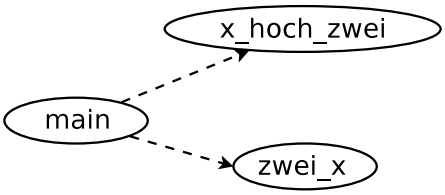

Abbildung 2.1: Aufrufgraph

Für die Konstruktion des Aufrufgraphen werden sowohl direkte Aufrufe, die zur Kompilierungszeit bereits feststehen, als auch indirekte Aufrufe über Funktionszeiger betrachtet (siehe vorheriges Beispiel), hinzu kommt die Auflösung polymorpher Aufrufe in der Objektorientierung. Um die potenziellen Ziele bei indirekten Aufrufen auflösen zu können, muss also vor der Aufrufgraphkonstruktion eine Zeigeranalyse die möglichen Ziele des indirekten Aufrufs berechnet haben. Für polymorphe Aufrufe kann unter anderem die Vererbungshierarchie betrachtet werden. Auch hierfür gibt es verschiedene Strategien.

Auf Basis des Aufrufgraphen erfolgt eine Menge unterschiedlicher Analysen, die hier nur eine kurze Erwähnung finden. Interprozedurale Datenflussanalysen greifen auf den Aufrufgraphen zurück. Die grafische Darstellung des Aufrufgraphen könnte bei der Erkennung einer unbekannten Architektur helfen. Die Dichte der Kanten in einem bestimmten Bereich des Graphen lassen Rückschlüsse auf einzelne Module eines Programms zu. Von `main` aus nicht erreichbare und somit nicht benutzte Funktionen (toter Quellcode) können erkannt und bei Bedarf für Optimierungen entfernt werden.

Weitere Metriken wie Größe, Tiefe und Breite des Aufrufgraphen, die Dichte der Kanten (in einem bestimmten Bereich), die Anzahl der Funktionen und Aufrufe

in Bezug auf die Anzahl der Quellcodezeilen etc. lassen einige Aussagen über die Architektur des vorliegenden Programms zu, wie z. B. Kopplung und Kohäsion.

2.2.3 Seiteneffekte

Bei Seiteneffekten wird zwischen lesenden und modifizierenden Seiteneffekten unterschieden. Eine Funktion hat einen lesenden Seiteneffekt, wenn sie eine Variable liest, die zuvor in einer anderen Funktion geändert wurde. Eine Funktion hat einen modifizierenden Seiteneffekt, wenn die Funktion eine Variable verändert, die nach der Beendigung der Funktion gelesen wird.

Im Vergleich zu funktionalen Programmiersprachen hat C/C++ durchaus Konstrukte, die einen Seiteneffekt hervorrufen können. Beispiele für diese Konstrukte sind Zeiger oder die Möglichkeit, eine globale Variable zu verändern.

Beispiel

```
1  int a = 4;
2  int b;
3
4  void f () {
5      a = b + 10;
6  }
7  int main () {
8      b = 2;
9      f();
10     ...
11 }
```

Der Aufruf von `f` in `main` hat einen lesenden Seiteneffekt auf `b` und einen modifizierenden Seiteneffekt auf `a`.

Das klassische Verfahren zur Berechnung der Seiteneffekte von Cooper und Kennedy [CK88] wurde in Bauhaus nicht implementiert. Es existiert ein einfacheres Verfahren in Bauhaus [SVKW07]. Im ersten Schritt wird die Berechnung intraprozedural durchgeführt, im zweiten Schritt werden Seiteneffekte im Aufrufgraphen postorder propagiert. Starke Zusammenhangskomponenten werden mittels Tarjans Algorithmus [Tar72] berechnet und erhalten dieselbe Menge für die Seiteneffektberechnung.

Als Ergebnis der Seiteneffektberechnung hat jede Funktion drei Mengen von Variablen. Die Menge may_use gibt die potenziellen Variablen an, die gelesen werden können (lesende Seiteneffekte). Bei modifizierenden Seiteneffekten wird unterschieden zwischen entweder must_def (Variablen, die geändert werden) oder may_def (Variablen, die potenziell geändert werden). Erhöht sich die Relation must_def zu may_def z. B. durch vorangegangene Analysen, so wird eine bessere Genauigkeit erzielt.

Die Seiteneffektberechnung ist nötig für die Berechnung der ISSA-Form.

2.2.4 ISSA-Analyse

Die Static Single Assignment Form kurz SSA-Form wurde von Cytron et al. 1991 entwickelt [CFR+91]. In Bauhaus dagegen existiert eine erweiterte Form, auch bekannt interprozedurale SSA-Form (ISSA) [SVKW07].

Die SSA-Form repräsentiert eindeutig die Datenabhängigkeit und reduziert die Anzahl der Datenabhängigkeitskanten durch das Hinzufügen von künstlichen ϕ-Knoten an Stellen, an denen Kontrollflüsse zusammenfließen (wie z. B. am Ende von `if`-Verzweigungen oder `while`-Schleifen).

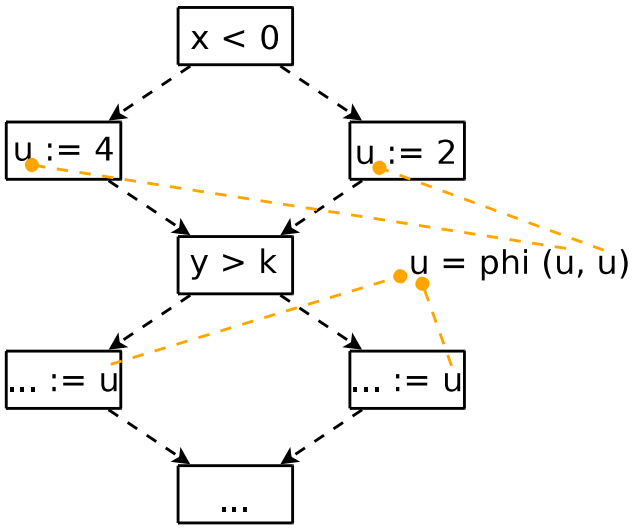

Abbildung 2.2: SSA-Form mit künstlichen ϕ-Knoten

Abbildung 2.2 stellt den Kontrollfluss von zwei `if-else`-Anweisungen dar. Nach der ersten `if-else`-Anweisung wurde ein künstlicher ϕ-Knoten $u = \phi(u, u)$ für `u` hinzugefügt. Die zwei Uses (Verwendungen) von `u` im zweiten `if-else`-Zweig zeigen jeweils mit einer Kante auf den künstlichen ϕ-Knoten, anstatt wie ursprünglich mit zwei Kanten auf die zwei Definitionen (Zuweisungen) an `u`.

Es ist nicht sinnvoll an jedem Zusammenfluss für jede Variable einen ϕ-Knoten zu erzeugen, denn diese Überschätzung würde zu unnötigen ϕ-Knoten führen. Ohne tiefer auf diese Problematik einzugehen, wird auf die allgemeine SSA-Form in Bauhaus verwiesen, welche die Arbeit von Cytron et al. umsetzt.

Die ISSA berücksichtigt neben Datenflüssen innerhalb von Funktionen auch interprozedurale Datenflüsse. Um dies umsetzen zu können, waren einige Umstellungen nötigt. Lokatoren abstrahieren lokale Variablen und sorgen für Alias-Freiheit. Funktionen können in verschiedenen Kontexten aufgerufen werden. Mittels einer Abbildung (Mapping) werden Lokatoren im jeweiligen Kontext vom Aufrufer zum Aufgerufenen abgebildet. Die Zuweisungen von Lokatoren zu Lokatoren werden am Anfang und am Ende einer Funktion mittels Pre Call Links und Post

Call Links simuliert. Sie ähneln Zuweisungen von aktuellen zu formalen Parametern. Die Seiteneffektberechnung kommt beim Setzen der ϕ-Knoten nach einem Funktionsaufruf ins Spiel, da bei einem Funktionsaufruf globale Variablen oder über Referenzparameter Variablen verändert worden sein könnten. Hier wurde in aller Kürze versucht, die SSA- und die ISSA-Form zu erklären. Für ausführlichere Beschreibungen wird auf die Originalarbeiten in [CFR+91] und [SVKW07] hingewiesen.

Die ISSA-Form findet zahlreiche Anwendungen, z. B. im Program Slicing [Wei84]. Ein Backward Slicing beantwortet für eine bestimmte Stelle die Frage, wie es zu einem Wert an dieser Stelle kommt. Ein Forward Slicing berechnet für eine Anweisung die Frage, was durch diese Anweisung beeinflusst wird. Weitere Anwendungen der ISSA-Form sind die später in 3.5 beschriebene Rollenpropagierung sowie die GUI-Analyse.

2.2.5 GUI-Analyse

Die Analyse der grafischen Benutzeroberfläche wurde von Staiger in [Sta07b] und [Sta07a] vorgestellt. Dabei geht es um das Erkennen von grafischen Benutzeroberflächen aus dem Quellcode. Die Struktur in Form von Widgets und die Verbindungen zwischen den Widgets werden erkannt. Eine Widget-Hierarchie bildet ein Fenster. Auf Basis von Widget-Hierarchien werden Interaktionen zwischen verschiedenen Fenstern identifiziert, so entsteht ein Fenstergraph. Die GUI-Analyse findet Anwendung zum Verständnis des vorliegenden Programms, denn ein Fenstergraph spiegelt oft die Logik der einzelnen Komponenten eines Programms wieder.

Auf Basis von Zeigeranalyse, Daten- und Kontrollflussinformation und der ISSA-Analyse wird die später in 3.5 beschriebene Rollenpropagierung benutzt, um Rollen wie „Einbetter“, „Konstruktor“ oder „Setzer“ zu identifizieren. Für gtk und Qt werden am Anfang Initialrollen mittels einer Konfigurationsdatei angegeben. Die initial angelegten Rollen werden entlang der Kanten des Aufrufgraphen rückwärts propagiert.

Bei der Untersuchung sollte berücksichtigt werden, wie grafische Bibliotheken analysiert werden können, sodass die Initialrollen für Schnittstellenfunktionen erkannt werden.

2.3 Mathematische Grundlagen für interprozedurale Datenflussanalyse

Die später vorgestellten interprozeduralen Datenflussanalysen stützen sich auf ein mathematisches Modell, welches hier vorgestellt wird. Es wird IDE[2] oder Interprozedurales Datenfluss-Framework genannt. Dieses Modell stammt von

[2]Englische Abkürzung für Interprocedural Distributive Environment

Sharir und Pnueli und legt bereits früh die theoretischen Grundlagen für die Datenflussanalyse [SP81].

Ein IDE ist ein 5-Tupel $\langle G, L, F, M, \eta \rangle$ mit folgenden Eigenschaften:

- $G = (N, E)$ ist ein interprozeduraler Kontrollflussgraph (ICFG) mit Knoten $n \in N$ als Anweisungen und intraprozeduralen Kanten $e \in E$ zwischen den Anweisungen. Ebenfalls als Kanten gelten interprozedurale Aufrufkanten und Rückkehrkanten von Funktionsaufrufen. Die Pfade im ICFG stellen jeden möglichen Programmfluss dar mit der Annahme, dass zu der zuletzt benutzten Aufrufkante auch eine entsprechende Rückkehrkante existiert und die Pfade „legal“ sind.
- L ist eine endliche Menge mit der Ordnungsrelation $\leq$ und dem größten Element $\top$ und ist ein konfluenter Halbverband[3]. $\wedge$ wird als ein binärer Konfluenzoperator[4] mit folgenden Eigenschaften definiert: Idempotenz, Assoziativität und Kommutativität. $x \wedge y$ ergibt immer ein eindeutiges Element m, welches $\leq x$ und y ist. Für alle $x, y \in L$ gibt es ein m.
- $F \subseteq \{f \mid f : L \to L\}$ ist eine monotone[5] Funktionsmenge, die abgeschlossen unter Komposition $\circ$ und Konfluenz $\wedge$ ist. Die Funktion $f(x)$ drückt aus, wie die Information x, die an einem Knoten n bekannt ist, in eine Information y an einem Knoten m umgewandelt wird, unter der Annahme, dass mindestens ein Pfad von n nach m führt.
- $M : E \to F$ ist eine Abbildung aller Kanten $e \in E$ an eine Funktion $f_e \in F$ mit $f_e = M(e)$ und drückt aus, welche Auswirkung das Ausführen einer Anweisung hat.
- $\eta \in L$ drückt die Anfangsinformation aus, die an einem Startknoten existiert. Es existiert genau ein Startknoten n_0, dieser ist Beginn der *main* Funktion.

Ein realisierbarer Pfad besteht aus Knoten $n_0 \cdots n_n$, die mit Kanten verbunden sind, und stellt einen statisch möglichen Ablauf des Programms dar.

Eine zusammenfassende Funktion ϕ_n an einem Knoten n_n drückt den Effekt vom Startknoten n_0 bis n_n aus. Werden die Knoten sequenziell ausgeführt, so ist z. B. $\phi_2 = f_1 \circ f_0$. Kommen die Pfade dagegen zusammen wie z. B. bei `if`, so gilt $\phi_2 = f_0 \wedge f_1$.

Für ein bestimmtes Problem wird eine Instanz vom angegebenen IDE als 5-Tupel abgeleitet und L, F, M und η werden entsprechend angepasst und angegeben. Auf nähere Details, wie ein Problem aussehen und gelöst werden kann, wird z. B. in den Abschnitten 3.3 und 3.6.2 sowie in [NNH99] eingegangen.

[3] Englisch: meet semilattice. Bei einem Halbverband gelten Assoziativität, neutrales Element, Kommutativität und Idempotenz.
[4] Englisch: meet
[5] Aus $x \leq y$ folgt $f(x) \leq f(y)$ für alle $f \in F$ und $x, y \in L$

3 Problematik der Analyse von Programmen mit Bibliotheken und Ansätze

Dieser Kapitel erläutert Schwierigkeiten, die bei der statischen Analyse von Bibliotheken auftreten. Es gibt einen Überblick über bekannte Ansätze für die Analyse von Bibliotheken. Diese werden klassifiziert. Einige nah verwandte Arbeiten mit Bibliotheken werden betrachtet. Das Hauptaugenmerk richtet sich auf die Component-Level-Analysis[1] [RKM06] und die Fragment-Analyse [RR01]. Anschließend werden die Ansätze hinsichtlich der Umsetzbarkeit, der Präzision und der Abdeckung der angestrebten Analysen verglichen.

3.1 Besondere Schwierigkeiten

In diesem Abschnitt werden Schwierigkeiten im Bezug auf die statische Analyse mit Bibliotheken aufgezeigt. Beim Vergleich (Abschnitt 3.7) wird überprüft, wie auf die Schwierigkeiten eingegangen wird und ob die vorgestellten Ansätze die dargestellten Schwierigkeiten befriedigend gelöst haben.

Funktionszeiger und Rückruffunktion

Funktionszeiger sind ein beliebtes Mittel in C/C++ und können mittels Parameterübergabe Rückrufe[2] zum Benutzerprogramm realisieren. Da Bibliotheken adaptiv zu jedem Benutzerprogramm passen müssen und zur Erstellungszeit der Bibliothek das Benutzerprogramm noch nicht vorhanden ist, kann es keinen direkten Aufruf von der Bibliothek zum Benutzerprogramm geben. Aufrufe von Bibliotheken zum Benutzerprogramm sind aber trotzdem notwendig und können über Funktionszeiger realisiert werden. Abbildung 3.1 stellt das Benutzerprogramm (`main`) und die Bibliothek (`bib`) mit einer Rückruffunktion dar.

Anhand eines Beispiels unter Benutzung einer grafischen Bibliothek wird gezeigt, wie die Wechselwirkung mit Rückruffunktionen stattfindet. Das Benutzerprogramm verwendet eine grafische Bibliothek zur Erzeugung eines Fensters (Abbildung 3.2). Darin enthalten sind drei Auswahlmöglichkeiten, die zur Verfügung stehen,

[1] Der englische Titel wird durchgehend beibehalten. Es findet sich hierfür kein befriedigender deutscher Titel.
[2] Englisch: callbacks

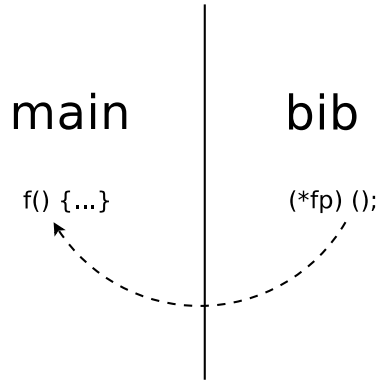

Abbildung 3.1: Rückruffunktion

diese sind grafisch mit drei Schaltflächen verbunden. Um nicht aktiv warten[3] zu müssen, hinterlegt das Benutzerprogramm für jede Schaltfläche eine Funktion, die aufgerufen werden soll, wenn die zugehörige Schaltfläche angeklickt wurde.

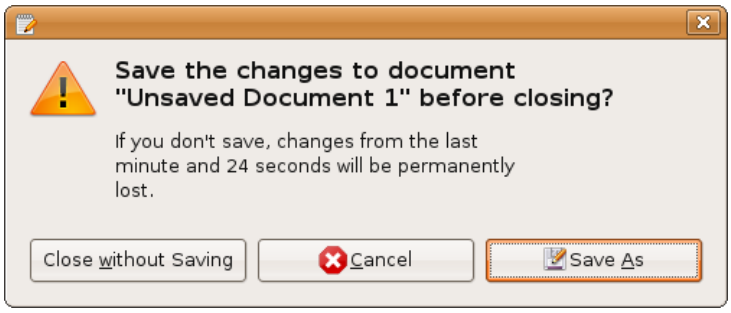

Abbildung 3.2: Abfrage mit Rückruffunktion

Die Schwierigkeit bei der Analyse ist also die Tatsache, dass von einem beliebigen Benutzerprogramm ausgegangen wird. Insbesondere kann für indirekte Aufrufe nicht bestimmt werden, welche Ziele diese Aufrufe haben. Das Fehlen von Zielen bei indirekten Aufrufen bedeutet im Allgemeinen das Fehlen von Datenflüssen und führt dazu, dass weiterführende Analysen unpräzise oder falsch werden.

Datenflüsse

Eine Bibliothek bietet als Schnittstelle zwar Funktionen oder Klassen an, die benutzt werden können, im Allgemeinen sind aber die Aufrufe dieser Funktionen oder Klassen nicht vorhanden, sondern im Benutzerprogramm zu finden, welches noch unbestimmt ist. Daher fehlen auch Datenflüsse von direkten Aufrufen.

Als einfaches Beispiel wird die Stackverwaltung dargestellt:

```
1  void push (List **top, Element *elem);
2  void pop (List **top);
```

[3] Englisch: busy waiting

Die Funktionen `push` und `pop` koalieren offensichtlich stark miteinander, um das Hinzufügen und Entfernen eines Elements zu einem Stack anzubieten. Eine direkte Analyse von `push` und `pop` ohne einen konkreten Stack und Aufrufe von `push` und `pop` würde dazu führen, dass beide Parameter `top` nicht ein Stack modifizieren, sondern diese zwei unabhängige Parameter sein könnten. Fehlende Aufrufe werden in der Abbildung 3.3 als gestrichelte Linien dargestellt.

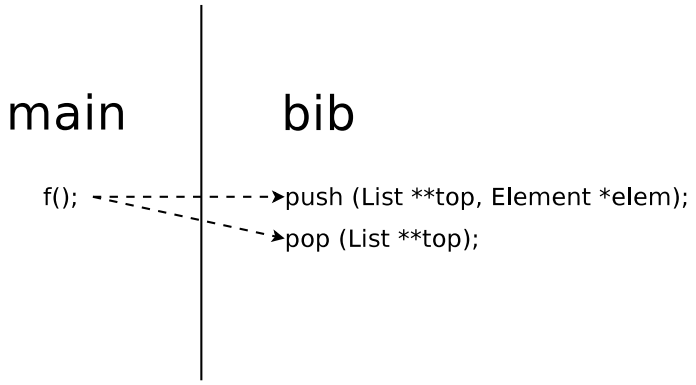

Abbildung 3.3: Fehlende Datenflüsse

Vererbung

Mit der Vererbung treten weitere Schwierigkeiten auf. In Abbildung 3.4 bietet die Bibliothek eine Klasse `A` an. `B`, `C` und `D` sind Klassen im Benutzerprogramm und erben von `A`. Beim Aufruf der Methode `m()` über `a->m()` im Benutzerprogramm kommen drei Methoden in Frage, da `m()` virtuell ist und erst zur Laufzeit entschieden wird, welches `m()` tatsächlich gemeint ist. Die beiden neuen Methoden `m()` von `B` und von `D` können ein anderes Verhalten aufweisen als `m()` von `A` und damit andere Datenflüsse erzeugen. Sie sind zudem zur Zeit der getrennten Bibliotheksanalyse noch nicht bekannt.

Falls eine getrennte Bibliotheksanalyse angestrebt wird, treten weitere Probleme auf. Zusätzliche Methoden `c->nm()`, die `A` erweitern, oder die Implementierung von abstrakten Klassen/Methoden können ebenfalls nicht berücksichtigt werden, da sie noch nicht bekannt sind.

Existierende Analysen

Zahlreiche existierende Analysen wie „gültige Definitionen“ (Kapitel 3.3.1), „verfügbare Ausdrücke“ oder „aktive Variablen“ [NNH99] — um hier nur einige Klassische zu nennen — gehen davon aus, dass **ganze Programme vorhanden** sind. Bei einer Bibliothek handelt es sich allerdings um ein unvollständiges Programm — existierende Gesamtprogramm-Analysen (in Bauhaus) können also nicht direkt auf Bibliotheken angewendet werden.

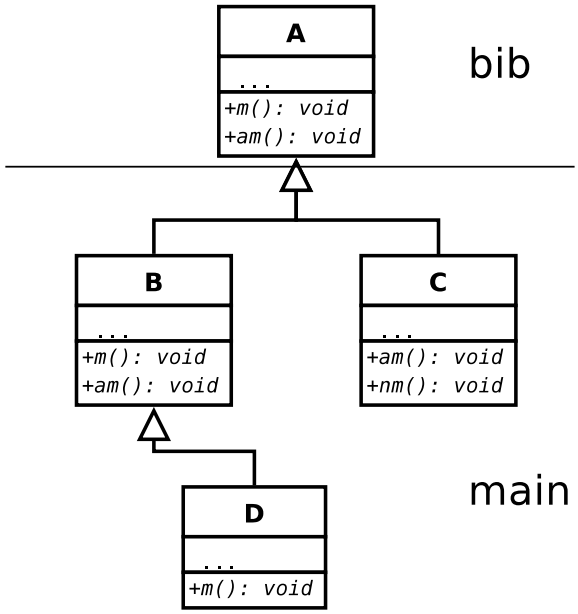

Abbildung 3.4: Vererbung von Bibliotheksklassen

Wie von der Aufgabenstellung gefordert sollen Zeigeranalysen, Aufrufgraphkonstruktion, Seiteneffektberechnung, ISSA-Analyse sowie GUI-Analysen untersucht werden. Diese Analysen existieren als Gesamtprogramm-Analysen bereits in Bauhaus. Sie können aber nicht direkt auf Bibliotheken angewendet werden, da Bibliotheken keine vollständigen Programme sind.

3.2 Einfache Ansätze

Nun werden einige einfache Ansätze betrachtet. Anhand dieser einfachen Überlegungen wird deutlich gemacht, welche generellen Schwierigkeiten auftreten können und was ein guter Ansatz mit sich bringen muss, um diese zu bewältigen.

3.2.1 Ignorieren der Effekte

Angenommen eine Bibliothek wird für die Verwaltung von Stacks benutzt. Die Funktion

```
1              void push (List **top, Element *elem);
```

fügt dem Stack ein Element hinzu. Falls nun das Benutzerprogramm die Funktion `push` benutzt, ist davon auszugehen, dass das Element `top` sich verändert hat. Würde diese Tatsache einfach ignoriert werden und angenommen werden,

dass sich nach dem Aufruf `top` nichts geändert hat, so ist es falsch und **nicht konservativ**[4]. Außerdem können Funktionsaufrufe die Werte von globalen Variablen geändert haben. Noch schlimmer verhält es sich, wenn Funktionen als Parameter Funktionszeiger haben. Über Funktionszeiger kann eine Kette von Funktionen im Benutzerprogramm oder in der Bibliothek angestoßen worden sein. Die aufgezählten Tatsachen zu ignorieren würde zu einer falschen Programmanalyse führen. Um es auf den Punkt zu bringen: Daten- und Kontrollflüsse sowie Seiteneffekte würden komplett fehlen.

3.2.2 Annahme des schlimmsten Falles

Um wie erwähnt konservativ zu bleiben, muss bei einem Funktionsaufruf wie `push` davon ausgegangen werden, dass sich alle Parameter geändert haben. Es ist außerdem davon auszugehen, dass `push` sämtliche globalen Variablen der Bibliothek geändert hat. Darüber hinaus ruft `push` alle Funktionen auf, die einen Funktionszeiger (als Parameter von `push`) als Ziel haben, mit der Konsequenz, dass wiederum Funktionen vom Benutzerprogramm angestoßen werden. Im schlimmsten Fall kann `push` alle Bibliotheksfunktionen aufrufen. Es bleibt zwar konservativ, mit der Methode kann aber keine Programmanalyse betrieben werden, da durch einen Funktionsaufruf der Bibliothek sämtliche Aussagen sehr **unpräzise** werden. Dies würde zu einer drastischen Überschätzung führen, welche die Aussagegenauigkeit verschlechtert. Speziell im Bereich Programmverstehen ist das nicht tragbar, da hier eine hohe Genauigkeit wichtig ist.

3.2.3 Manuelle Erstellung von Attrappen

Es ist vorstellbar, für die Funktion `push` folgenden Rumpf künstlich zu erstellen:

```
1  void push (List **top, Element *elem) {
2      *top = ...;
3      ... = elem;
4  }
```

Mit diesem Rumpf wird simuliert, dass `top` verändert und `elem` gelesen wurden. Auf den ersten Blick klingt die Idee plausibel, jedoch kann auch diese Lösung nicht befriedigend sein, da der Analysenanwender die Semantik aller Bibliotheksfunktionen kennen muss, um künstliche Rümpfe dafür zu erstellen. Dies ist nicht gegeben und scheitert am **Zeitaufwand**, für die Aneignung des Wissens und der Erstellung der Rümpfe. Größere Bibliotheken mit mehreren hunderttausend Programmzeilen machen diesen Ansatz unbrauchbar. Bei unzureichenden Kenntnissen führen falsch erstellte Rümpfe zu Fehlern in der Analyse.

[4] Statische Programmanalyse führt ein Programm P nicht aus. Konservativ zu bleiben bedeutet die Annahme, dass das Programm P auf jede Eingabe anders reagieren kann und alle Reaktionen (z. B. alle Ausführungspfade) von P abgedeckt werden muss.

3.2.4 Analyse mit gesamter dazu gebundener Bibliothek

Der letzte einfache Ansatz nimmt alle im Benutzerprogramm benutzten Bibliotheken und versucht das Benutzerprogramm und die benutzten Bibliotheken als eine Einheit zu betrachten und diese zu analysieren. Es ist vorstellbar, z. B. eine einfache Zeigeranalyse auf diese Einheit anzuwenden und den Aufrufgraphen zu erstellen. Der Tote Code könnte nach der Zeigeranalyse entfernt werden. Weitere darauf aufbauende Analysen arbeiten nur noch mit „lebendigem“ Code.

Hierzu sollte beachtet werden, dass Bibliotheken wiederum Bibliotheken benutzen können und somit alle weiteren Abhängigkeiten ebenfalls betrachtet werden müssen. Die besagte Einheit könnte also im Verhältnis zum Benutzerprogramm **überproportional** groß werden. Einfache Zeigeranalysen, wie die Steensgaard-[Ste96] oder Das-Analyse [Das00], sind nicht rechen- und zeitintensiv und können in der Praxis hinnehmbare Laufzeiten liefern — sie sind jedoch **ungenau** und führen dazu, dass zu wenig Toter Code entfernt wird. Genauere Zeigeranalysen, wie die von Andersen [And94] oder Wilson [WL95], sind für große Bibliotheken nicht praktikabel, da die Laufzeit im Fall von Andersen in $\mathcal{O}(n^3)$ ist.

3.3 Component-Level-Analysis

Die Component-Level-Analysis [RKM06] wurde von Rountev et al. auf der International Conference on Compiler Construction 2006 vorgestellt. Anhand eines Beispiels aus der Veröffentlichung wird veranschaulicht, welche Idee dahinter steckt. Anschließend ist zu diskutieren, welche weiteren Analysen der Ansatz abdeckt und ob dieser für die Zielanalysen dieser Diplomarbeit geeignet ist.

3.3.1 Gültige Definitionen

Dieses Beispiel behandelt das klassische Problem „gültige Definitionen“[5] [Muc97].

Die Definition einer Variablen ist eine Terminologie aus dem Backend und bedeutet eine Zuweisung eines Wertes. Die Definition D ist an einem Punkt P gültig, falls es einen Pfad von D nach P gibt und nicht auf allen Pfaden von D nach P eine Zuweisung an die Variable V erfolgt, Beispiel:

```
1  V = 1;
2  if (Cond)
3    X = 2;
4  else
5    X = 3;
6  X = V + X;
```

[5] Englisch: reaching definitions

Zwischen der vorletzten und letzten Zeile besitzt V die gültige Definition $\{1\}$, X die gültigen Definitionen $\{2,3\}$.

Das Lösen des Problems gültige Definitionen ist eine typische Vorwärts-Datenflussberechnung. Die Lösung wird mittels Erzeuge- und Lösche-Menge berechnet. Bezüglich weiterer Einzelheiten zur Lösung wird an dieser Stelle auf [Muc97] oder [NNH99] verwiesen.

3.3.2 Beispiel

Das folgende Beispielprogramm bedarf einer näheren Betrachtung. Das Programm hat fünf Funktionen, wobei `main` und `ext` zum Benutzerprogramm gehören, das eine Bibliothek mit den Funktionen `p1`, `p2` und `p3` benutzt. Darüber hinaus besitzt die Bibliothek vier globale Variablen `k`, `l`, `m` und `*fp`, die ebenfalls vom Benutzerprogramm verwendet werden. Der Einfachheit halber werden nur Literale an die globalen `int`-Variablen zugewiesen, der `void`-Zeiger `*fp` bekommt die Funktion `ext` zugewiesen.

```
void main() {
  k = 0;
  fp = &ext;
  p1();
}
void ext() {
  k = 9; }
// Bibliothek
public int k, l, m;
public void *fp;
public void p1() {
  if (...)
    k = 2;
  else
    k = 3;
  p2();
}
```

```
private void p2() {
  if (...) {
    (*fp)();
    if (...)
      l = 4;
    else
      l = 5;
  }
  else {
    l = 6;
    p3();
  }
}
private void p3() {
  k = 7;
  m = 8;
}
```

Das Beispielprogramm wird in Abbildung 3.5 als Flussgraph dargestellt. Knoten stellen Anweisungen dar, durchgezogene Linien sind intraprozedurale Flusskanten, gestrichelte Linien interprozedurale Flusskanten. Farbig markierte Funktionen gehören zur Bibliothek.

3.3.3 Transferfunktion

Die intraprozeduralen Flusskanten gehören zu einer jeweiligen Anweisung als Knoten, von der sie starten. Sie besitzen eine Transferfunktion $f(x)$, falls eine Anweisung eine Zuweisung ist, andernfalls ist die Transferfunktion die Identität *id*. Die Transferfunktion von Knoten zwei zu Knoten drei hat die Form:

$$f_0(x) = (x - D_k) \cup \{d_0\}$$

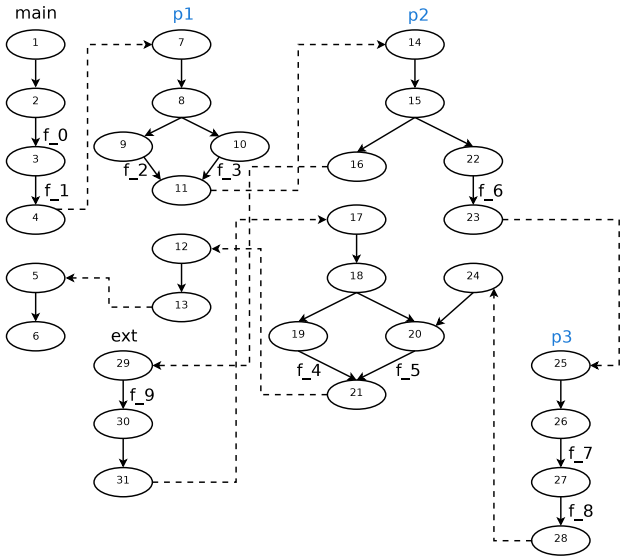

Abbildung 3.5: Beispielprogramm als Flussgraph

Sie drückt aus, dass alle gültigen Definitionen der Variablen `k` gelöscht werden. Hinzu kommt die gültige Definition d_0 für die Variable `k`. Gleiches gilt für andere Transferfunktionen:

$$
\begin{aligned}
f_1(x) &= (x - D_{fp}) \cup \{d_1\} & f_2(x) &= (x - D_k) \cup \{d_2\} \\
f_3(x) &= (x - D_k) \cup \{d_3\} & f_4(x) &= (x - D_l) \cup \{d_4\} \\
f_5(x) &= (x - D_l) \cup \{d_5\} & f_6(x) &= (x - D_l) \cup \{d_6\} \\
f_7(x) &= (x - D_k) \cup \{d_7\} & f_8(x) &= (x - D_m) \cup \{d_8\} \\
f_9(x) &= (x - D_k) \cup \{d_9\}
\end{aligned}
$$

3.3.4 Zusammenfassung und Benutzung der Transferfunktionen

Interessant ist nun die Zusammenfassung der Transferfunktionen für komplette interprozedurale Pfade, die lokal innerhalb der Bibliothek bleiben. Der Pfad von Knoten sieben bis Knoten elf wird betrachtet.

$$\psi_{11}^{7}(x) = (x - D_k) \cup \{d_2, d_3\}$$

Der Wert der Variablen `k` wurde sowohl im `if`- als auch im `else`-Zweig verändert. Das heißt, alle gültigen Definitionen von `k` werden gelöscht, hinzu kommen die gültigen Definitionen d_2 und d_3. Gleiches gilt für andere innerhalb der Bibliothek bleibenden intra- bzw. interprozeduralen Pfade:

$$
\begin{aligned}
\psi_{13}^{12}(x) &= x & \psi_{16}^{14}(x) &= x \\
\psi_{21}^{14}(x) &= (x - D_k - D_l - D_m) \cup \{d_6, d_7, d_8\} & \psi_{21}^{17}(x) &= (x - D_l) \cup \{d_4, d_5\}
\end{aligned}
$$

Anstatt einen Abstrakten Syntaxbaum oder einen Flussgraphen als Zwischendarstellung von Bibliotheksfunktionen zu haben, werden für jede Bibliotheksfunktion Teile ihrer intra- bzw. interprozeduralen Pfade in Form von Transferfunktionen gespeichert. Für die Funktion `p1` gilt:

$$\Psi_{p1} = \{\psi_{11}^{7}, \psi_{13}^{12}\}$$

Für `p2` wird Ψ_{p2} benutzt. Da Funktion `p3` weder explizit vom Benutzerprogramm benutzt wird noch interprozedurale Kanten hat (außer beim Ein- und Austritt der Funktion), wird `p3` direkt in `p2` eingebettet, da `p2` `p3` aufruft.

$$\Psi_{p2} = \{\psi_{21}^{14}, \psi_{16}^{14}, \psi_{21}^{17}\}$$

Auf der linken Seite von Abbildung 3.6 wird die Bibliothek getrennt als Flussgraph dargestellt, rechts ist die Zusammenfassung zu sehen. Wie auch hier deutlich zu erkennen ist, werden durch die Zusammenfassung Knoten eingespart und das Ergebnis ist eine kompakte Darstellung in Form von zusammengefassten Transferfunktionen Ψ.

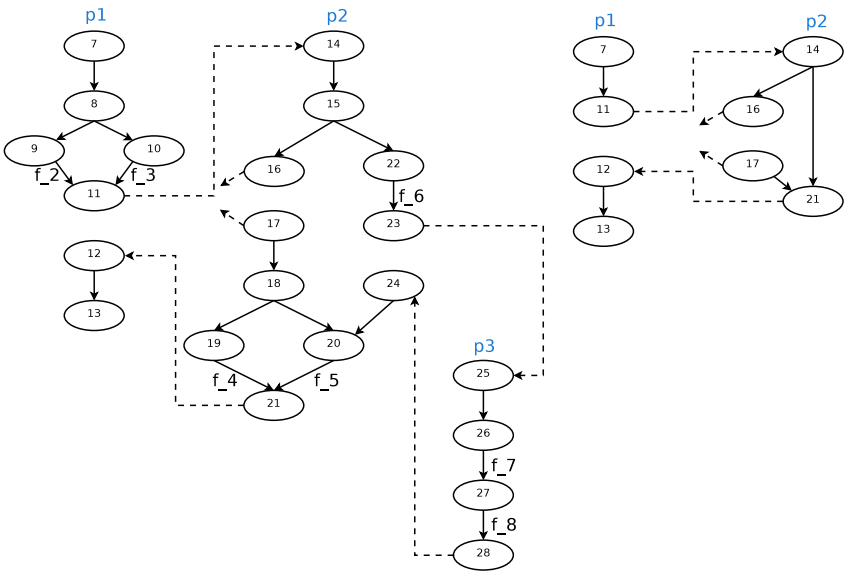

Abbildung 3.6: Bibliothek und die Zusammenfassung

Nach der Zusammenfassung der Bibliotheksfunktionen ist das Problem der gültigen Definitionen noch nicht gelöst, denn die Component-Level-Analysis soll nur eine Vorarbeit speziell für Bibliotheken liefern. Bei der Analyse der gültigen Definitionen werden an den Aufrufstellen zu den Bibliotheksfunktionen die zusammengefassten Transferfunktionen benutzt. Angenommen wird ein beliebiges Benutzerprogramm mit dem zugehörigen interprozeduralen Kontrollflussgraphen, welches die Bibliothek benutzt. Nun wird an den Aufrufstellen der Kontrollflussgraph mit dem kompakten Kontrollflussgraphen der Bibliothek (rechte Seite von

Abbildung 3.6) verbunden. Diese Verbindung ermöglicht die Analyse eines vollständigen Programms/Graphen. Dadurch ergibt sich Zeit- und Aufwandsersparnis für die eigentliche Analyse des Benutzerprogramms.

3.3.5 Technische Umsetzung

Die Berechnung der zusammengefassten Transferfunktionen wird näher betrachtet. Dazu werden erst einige Begriffe definiert. Ein feststehender Aufruf [6] ist ein Aufruf, dessen Ziel immer dieselbe Prozedur in der Bibliothek ist. In C ist jeder Aufruf, der nicht durch einen Funktionszeiger erfolgt, feststehend. In C++ sind alle Aufrufe zu Memberfunktionen, die durch das Attribut `private` markiert sind, feststehend. Hinzu kommen alle Aufrufe von Nicht-Memberfunktionen, die nicht durch einen Funktionszeiger erfolgen. Rekursiv wird eine Prozedur p als feststehend[7] definiert, wenn p entweder keine Aufrufe enthält oder p nur feststehende Aufrufe zu feststehenden Prozeduren hat. Alle übrigen Prozeduren sind nicht feststehend[8]. In dem genannten Beispiel ist die Prozedur `p3` feststehend, `p1` und `p2` nicht. Aufrufe bei Knoten 11/12 und Knoten 23/24 sind feststehend, 16/17 dagegen nicht.

Als Erstes erfolgt die Berechnung der feststehenden Funktionen und Aufrufe gemäß der oben angegebene Definition. Anschließend werden die Anfangswerte für die gültigen Definitionen initialisiert. Es erfolgt ein Fixpunktansatz mit Propagierung der Werte, bis sie konstant sind. Hierbei können die Werte für feststehende Aufrufe und Funktionen vollständig berechnet werden. Nicht feststehende Aufrufe bleiben offen in Form von kompakten Transferfunktionen und werden erst zur Analysezeit des Benutzerprogramms vollständig berechnet. Für die genaue Beschreibung des Algorithmus sei auf die Originalarbeit [RKM06] verwiesen.

Rountev schlägt für die Speicherung der Transferfunktionen zwei Mengen vor. Die erste Menge K enthält Variablen, deren Definition nicht mehr gilt. Die zweite Menge G enthält Definitionen, die jeweils für eine Variable hinzukommt. ψ_{11}^7 wird also auch dargestellt als:

$$\psi_{11}^7 = \{\{k\}, \{d_2^k, d_3^k\}\}$$

Hier die konkrete Umsetzung der Speicherung der Transferfunktionen als Mengen, speziell für gültige Definitionen:

$$f = (K, G)$$

Dabei wird die Hintereinanderausführung zweier Knoten als Mengenoperationen wie folgt berechnet:

$$f_2 \circ f_1 = (K_1 \cup K_2, (G_1 - K_2) \cup G_2)$$

[6] Englisch: fixed call
[7] Englisch: fixed procedure
[8] Englisch: non-fixed procedure

Für das Zusammentreffen zweier Pfade mit dem Konfluenzoperator werden ebenfalls Mengenoperationen angewandt:

$$f_1 \wedge f_2 = (K_1 \cap K_2, G_1 \cup G_2)$$

Diese zwei Regeln reichen aus, um die Berechnung der gültigen Definitionen durchzuführen.

3.3.6 Anpassung für Zeigeranalysen

Das vorgestellte Beispiel behandelt die Analyse der gültigen Definitionen, um die Idee der Kompaktifizierung der Transferfunktionen zu vermitteln. Die Component-Level-Analysis beschränkt sich jedoch keinesfalls auf gültige Definitionen, sondern deckt zwei Kategorien von Problemen ab: „Interprocedural Finite Distributive Subset“ (IFDS) [RHS95] und „Interprocedural Distributive Environment“ (IDE) [SRH96] (Abschnitt 2.3). Es muss jedoch für jedes Problem überlegt werden, wie der Aufbau von Transferfunktionen jeweils aussieht. Bei genauer Betrachtung ist zu erkennen, dass für klassische Datenflussberechnungen wie „verfügbare Ausdrücke“[9] oder „aktive Variablen“[10] [NNH99] die Component-Level-Analysis leicht angepasst und angewandt werden kann. Auf die Details der aufgeführten Kategorien und Probleme wird nicht näher eingegangen, da sie den Rahmen dieser Arbeit sprengen würden.

Für die angestrebten Zeigeranalysen könnten die Transferfunktionen nach eigener Überlegung wie folgt aussehen. Sei P_v die Menge der Variablen, auf die v an einem bestimmten Knoten zeigen kann. Die Menge P_v ist zur Analysezeit der Bibliothek noch nicht bekannt, da die Ziele im unbekannten Benutzerprogramm sein könnten. Es existiert daher nur Transferfunktionen an Kanten des Datenflussgraphen. Es wird nun für die möglichen Zeigerzuweisungen die entsprechende Transferfunktion mit Erklärung angegeben:

$$(1) \qquad \texttt{a = b;} \iff f(x) = (x - P_a) \cup P_b$$

Die Menge der Variablen, auf die `a` zeigt, wird gelöscht. `a` hat nach der Anweisung die Menge der Variablen, auf die `b` bis dahin gezeigt hat.

$$(2) \qquad \texttt{a = \&b;} \iff f(x) = (x - P_a) \cup \{b\}$$

Die Menge der Variablen, auf die `a` zeigt, wird gelöscht. `a` zeigt auf eine einelementige Menge mit dem Inhalt `b`.

[9] Englisch: available expressions
[10] Englisch: live variables

$$(3) \qquad \texttt{a = *b;} \iff f(x) = (x - P_a) \bigcup_{u \in P_b} P_u$$

Die Menge der Variablen, auf die `a` zeigt, wird gelöscht. `a` zeigt auf die Vereinigung aller Ziele von `u`, die Element von P_b sind.

$$(4) \qquad \texttt{*a = b;} \iff \forall v \in P_a : f(x) = (x - \emptyset) \cup P_b$$

`v` ist ein Element der Ziele von `a`. Zu den Zielen der verschiedenen `v` kommt die Menge P_b hinzu.

In C/C++ kommt jedoch noch die beidseitige Dereferenzierung `*a = *b;` sowie die mehrfache Dereferenzierung `**a = **b;` vor. Hier werden eindeutige bis dahin nicht benutzte Hilfsvariablen eingesetzt und die Anweisungen werden auf die Formen (3) oder (4) zurückgeführt [Froo6].

```
*a = *b;   <=>   (3) tmp_i = *b;
                 (4) *a    = tmp_i;
```

```
**a = **b;   <=>   (3) tmp_j   = *a;
                   (3) tmp_k   = *b;
                   (3) tmp_l   = *tmp_k;
                   (4) *tmp_j  = tmp_l;
```

Hintereinanderausführung

```
1  v1 = x;
2  v2 = y;
```

$$f_2 \circ f_1 = (K_1 \cup K_2, G_1 \cup G_2)$$

Für die Hintereinanderausführung zweier Knoten wird die verkettete Anwendung von zwei Funktionen am jeweiligen Knoten mit der Semantik benutzt, dass beide Zeigermengen der Variablen `v1` und `v2` gelöscht werden. `v1` bekommt die neue Zeigermenge G_1 von `x`, `v2` die Zeigermenge G_2 von `y`.

Zusammenführung

```
1  if (Cond)
2    v1 = ...;
3  else
4    v2 = ...;
5  ...
```

$$f_1 \wedge f_2 = (K_1 \cap K_2, G_1 \cup G_2)$$

Für zusammenführende Datenflüsse wird dieser Konfluenzoperator verwendet, wenn `v1` und `v2` gleich sind und auf beiden Wegen gesetzt wurden. Nur dann wird die Zeigermenge gelöscht. Für `v1` kommt die Zeigermenge G_1, für `v2` die Zeigermenge G_2 hinzu.

An folgenden Beispielen wird die Überlegung veranschaulicht. Mit diesen wird klar, dass die Component-Level-Analysis datenflusssensitive und kontrollflussinsensitive Zeigeranalysen unterstützen kann. Die Reihenfolge der Anweisungen wird nicht beachtet.

Beispiel 1

```
1  a = b;
2  a = c;
```

Es wird angenommen, dass vor der Ausführung der beiden Anweisungen `b` die Zeigermenge P_b und `c` die Zeigermenge P_c hat.

$$P_b = \{c, d\}$$
$$P_c = \{d, e\}$$

Laut der Form (1) gilt $f_1(x)$ für Zeile eins und $f_2(x)$ für Zeile zwei:

$$f_1(x) = (x - P_a) \cup P_b$$
$$f_2(x) = (x - P_a) \cup P_c$$

Die Hintereinanderausführung der beiden $f_2(x) \circ f_1(x) = f_2^1(x)$ ergibt

$$f_2^1 = (P_a \cup P_a, P_b \cup P_c)$$
$$f_2^1 = (P_a, \{c, d, e\}).$$

Die Aussage von f_2^1 lautet also: Die Zeigermenge von `a` wird gelöscht, `a` zeigt nun auf die neue Menge $\{c, d, e\}$. Würde die Component-Level-Analysis die Reihenfolge an Anweisungen beachten, so hätte `a` nur die Zeigermenge $P_c = \{d, e\}$ von `c`.

Beispiel 2

```
1              if (Cond)
2                a = b;
3              else
4                a = c;
5              ...
```

Es wird mit denselben Bedingungen wie im vorherigen Beispiel gestartet, `b` und `c` haben dieselben Zeigermengen.

$$P_b = \{c, d\}$$
$$P_c = \{d, e\}$$

$f_2(x)$ und $f_4(x)$ sind wie oben. $f_5^1 = f_2 \wedge f_4$ lautet:

$$f_2(x) = (x - P_a) \cup P_b$$
$$f_4(x) = (x - P_a) \cup P_c$$

$$f_5^1 = (P_a \cap P_a, P_b \cup P_c) = (P_a, \{c, d, e\})$$

Somit zeigt `a` unmittelbar vor Zeile fünf auf die Menge $\{c, d, e\}$, die ursprüngliche Zeigermenge von `a` wurde komplett gelöscht. Die Component-Level-Analysis berücksichtigt, wie gerade am Beispiel gezeigt, den Kontrollfluss für Zeigeranalysen nicht, denn die Mengen P_b und P_c werden nicht getrennt gehalten, sondern am Ende der `if`-Verzweigung vereinigt.

Die Anpassung der Component-Level-Analysis für Zeigeranalysen ist rein konzeptioneller Natur. In welche Richtung die Überlegung geht, haben die zwei Beispiele gezeigt. Die Angabe der Vorgehensweise erhebt nicht den Anspruch der Vollständigkeit für sich, sondern will vielmehr zeigen, wie schwierig die Vorgehensweise ist. Als Beispiel seien die unbekannten Datenflüsse genannt, die bei der Analyse der Bibliothek mit unbekanntem Benutzerprogramm vorkommen. Daraus resultiert die Tatsache, dass zur Analysezeit der Bibliothek ein Knoten nicht unbedingt alle Zeigerziele einer Variablen kennt, da das Ziel vom Benutzerprogramm abhängen kann.

3.3.7 Vor- und Nachteile

Die Zweckmäßigkeit mit den Vor- und Nachteilen der Component-Level-Analysis wird dargestellt im Hinblick auf die angestrebten Analysen dieser Arbeit.

Vorteile

Das Benutzerprogramm kann analysiert werden, ohne dass der Quellcode der Bibliothek vorliegen muss. Hersteller der Bibliothek und Hersteller des Benutzerprogramms können zwei unterschiedliche Parteien sein. Möchte nun der Besitzer des Benutzerprogramms sein Programm analysieren, so kann der Bibliotheksanbieter ihm die kompakte Darstellung der Transferfunktionen zur Verfügung stellen, ohne ihm den eigentlichen Quellcode der Bibliothek überlassen zu müssen.

Die **Kosten** in Bezug auf Laufzeit und Speicherverbrauch können für das Benutzerprogramm **reduziert** werden, da die Bibliothek bereits analysiert wurde und in kompakter Form vorliegt.

Eine Kosteneinsparung liegt ebenfalls vor, da eine Bibliothek für viele verschiedene Benutzerprogramme wiederverwendet werden kann — **analysiert** wurde die Bibliothek aber nur **einmal**.

Im Entwicklungszyklus eines Benutzerprogramms kann dieses an sich durch die Programmierung immer wieder geändert werden, während die Bibliothek im Idealfall konstant bleibt und nicht geändert wird. Durch die einmalige Voranalyse der Bibliothek wird mehrmalig an Analysezeit gespart, auch wenn der Quellcode des Benutzerprogramms verändert wird — **nur** das Benutzerprogramm an sich muss **erneut analysiert** werden.

Die kompakte Darstellung ist ein wesentlicher Vorteil. Die Component-Level-Analysis kann sowohl für Vorwärts- als auch für Rückwärts-Datenflussberechnungen verwendet werden. Sie hängt auch nicht vom Konfluenzoperator $f_1 \wedge f_2$ der jeweiligen Analyse ab.

Nachteile

Auf der anderen Seite sollten die Nachteile nicht außer Acht gelassen werden. Für klassische Datenflussberechnungen wie „gültige Definitionen“, „verfügbare Ausdrücke“ oder „aktive Variablen“ etc. [NNH99] muss **jeweils** ein Algorithmus existieren, der die kompakte Darstellung der Transferfunktionen berechnet. Je nachdem, welches Problem vorliegt, kommt ein anderer Konfluenzoperator $f_1 \wedge f_2$ zum Einsatz. Die Hintereinanderausführung $f_2 \circ f_1$ sieht ebenfalls für jedes Problem anders aus. Die jeweiligen Algorithmen müssen separat entwickelt werden. Die Speicherung erfolgt dann ebenfalls separat.

Sämtliche Werkzeuge in Bauhaus müssen **angepasst** werden, denn übliche Programmanalysen gehen davon aus, dass das komplette Programm samt Bibliothek vorhanden ist. Stößt nun eine Analyse auf einen Aufruf zu einer Bibliotheksfunktion, so muss diese Analyse „überredet“ werden, nicht weiter im Rumpf der Bibliotheksfunktion interprozedural zu analysieren, sondern die zusammenfassenden Transferfunktionen zu benutzen. Kompliziert wird es, da wie im Beispiel die Funktion `p1` und `p2` aus mehreren Teilpfaden ψ_i^j bestehen und diese erst zusammengefügt werden müssen.

Ein entscheidender Nachteil ist, dass **nicht für jede Datenflussanalyse** auch eine **Transferfunktion** nach Rountev existiert. Es konnte keine Lösung gefunden werden, wie eine Transferfunktion für die Seiteneffektberechnung und Aufrufgraphkonstruktion aussehen könnte. Für die Zeigeranalysen könnte die Transferfunktion wie in 3.3.6 beschrieben aussehen. Leider deckt somit die Component-Level-Analysis nur eine angestrebte Analyse ab und erfüllt nicht die geforderten Bedingungen eingangs dieser Arbeit.

3.4 Fragment-Analyse

Die zentrale Arbeit, auf die sich diese Diplomarbeit stützt, ist die Fragment-Analyse[11] von Rountev. 1999 wurde sie das erste Mal veröffentlicht [RRL99]. Weitere Veröffentlichungen erschienen 2001 auf der International Conference on Compiler Construction [RR01] sowie 2003 und 2004 [RMR03, RMR04]. Teile dieser Veröffentlichungen wurden vorher in Rountevs Dissertation ausführlich dargestellt [Rou02].

In der ersten Veröffentlichung [RRL99] wurde die Anpassung der Zeigeranalysen von Landi und Ryder [LR92] sowie eine 2-Phasen-Strategie der Fragment-Analyse behandelt. Zusätzlich gab Rountev einen formalen Korrektheitsbeweis der Fragment-Analyse an.

In [RR01] wurde die Thematik der Seiteneffektberechnung und der Zeigeranalyse von Andersen [And94] für Bibliotheken oder im Allgemeinen für Programmfragmente behandelt. Danach versuchte Rountev anhand der Zeigerinformation die Bibliothek zu kompaktifizieren. Dies geschah mittels Variablensubstitution und Anweisungseliminierung [RC00].

Bis dahin waren die Ansätze auf die Programmiersprache C zugeschnitten. In [RMR03, RMR04] wurde die Fragment-Analyse für die Programmiersprache Java, mit Fokus auf Objektorientierung und Klassen, entwickelt. Die Fragment-Analyse für Klassen[12] kann mit der Menge von Klassenanalysen umgehen, die ein ganzes Programm voraussetzt. Das Ergebnis dieser Methode bietet dem Analysenanwender z. B. die Möglichkeit, ein Programmfragment oder eine Bibliothek an ein Quellcode-Überdeckungswerkzeug[13] zu übergeben und diese(s) auf Polymorphismus testen zu lassen.

3.4.1 Idee

In diesem Abschnitt wird vorerst ohne Formalismus die Idee der Fragment-Analyse veranschaulicht. Wie in Abbildung 3.7 dargestellt, ist eine Bibliothek eine Ansammlung von Funktionen und globalen Variablen. Auf der linken Seite der Abbildung ist

[11] Englisch: Fragment Analysis
[12] Englisch: Fragment Class Analysis
[13] Englisch: code coverage tool

die Programmierschnittstelle[14] der Bibliothek dargestellt. Die Programmierschnittstelle ist in der typischen Form von Funktionen und Variablen vorhanden und bietet dem Benutzerprogramm die Möglichkeit, diese zu nutzen und zu verändern.

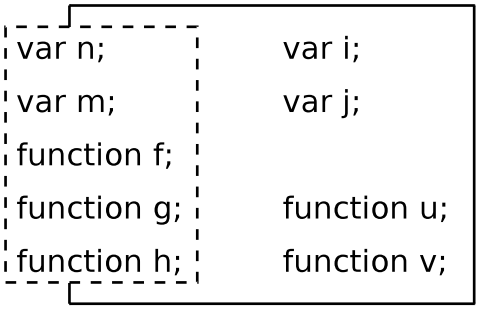

Abbildung 3.7: Bibliothek ohne Platzhalter

Die Fragment-Analyse ergänzt die Bibliothek mit einem oder mehreren Platzhaltern `ph` wie in Abbildung 3.8 dargestellt und vervollständigt die Bibliothek zu einem kompletten Programm. Durch die Vervollständigung können existierende Gesamtprogramm-Analysen, die in Bauhaus vorhanden sind, auf die Bibliothek angewendet werden.

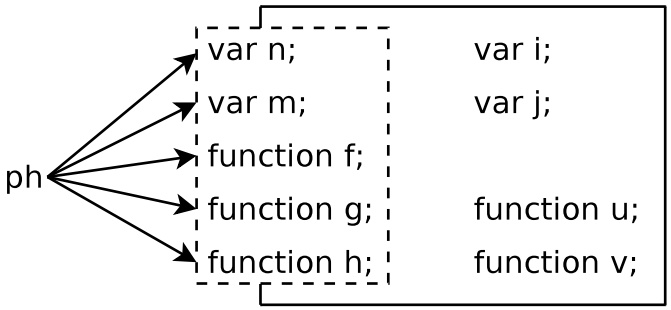

Abbildung 3.8: Bibliothek mit Platzhaltern

3.4.2 Platzhalter ph_*

Die Idee der Platzhalter wird als Quellcode konkreter veranschaulicht. Sie ergänzen die Bibliothek [Rou02, Kapitel 3.2.1] (Listing 3.1).

Sichtbare Variablen und Funktionen sind solche, die in der Programmierschnittstelle angeboten werden und die dafür vorgesehen sind, vom Benutzerprogramm benutzt und verändert zu werden. In Abbildung 3.8 sind die sichtbaren Variablen `n` und `m`, die sichtbaren Funktionen `f`, `g` und `h`.

Ergänzt wird die Bibliothek hier um eine globale Variable `ph_var` als Platzhalter (Zeile 1) und eine Funktion `ph_proc` (Zeile 2–22) als Platzhalterfunktion. Es wird ein Benutzerprogramm im ungünstigsten Fall simuliert und die nötigen Datenflüsse

[14] Englisch: application programming interface oder kurz API

```
1  global ph_var
2  procedure ph_proc (f1, ... ,fn) return ph_proc_ret {
3
4      ph_var = fi // fuer i von 1 bis n
5
6      ph_var = &v // fuer alle v, die nach aussen sichtbar sind
7
8      ph_var = &ph_var
9      ph_var = *ph_var
10     *ph_var = ph_var
11
12     ph_var = &ph_proc
13
14     ph_var = (*ph_var)(ph_var, ... ,ph_var) // mit m Parametern
15
16     // Alle Funktionen, deren Adressen irgendwo genommen wurden,
17     // werden in ph_proc aufgerufen.
18
19     ph_proc_ret = ph_var
20 }
21
22 // Alle indirekten Aufrufe rufen ph_proc auf
```

Listing 3.1: Die Ergänzung der Bibliothek

werden angestoßen. Daher existiert nur eine globale Variable `ph_var` und eine Funktion `ph_proc`. Die Variable `ph_var` hat keinen bestimmten Typ und könnte in der Implementierung für C/C++ den Typ `void*` haben.

Die Funktion `ph_proc` hat als Signatur `n` Parameter und den Rückgabewert `ph_proc_ret`. Der Typ der Parameter und des Rückgabewerts spielt keine Rolle. Die Anzahl der Parameter `n` von `ph_proc` berechnet sich aus der maximalen Anzahl der Parameter aller indirekten Aufrufe innerhalb der Bibliothek.

Für den nötigen Datenfluss sorgt Zeile 4. Alle `n` Parameter von `ph_proc` werden `ph_var` zugewiesen.

Adressen von sichtbaren Funktionen und Variablen werden `ph_var` zugewiesen. Hier geht es vorrangig um Adressen von sichtbaren Funktionen. Diese werden weitergereicht und gelten innerhalb der Bibliothek als Ziele für indirekte Aufrufe.

Die Zuweisungen in Zeile 8–10 simulieren den schlimmsten Fall für Zeigeranalysen. Dies geschieht dann, wenn der Adressoperator, Referenzierung und Dereferenzierung auf eine Variable `ph_var` erfolgen. Flussinsensitive Zeigeranalysen wie die von Andersen, Das und Steensgaard [And94, Das00, Ste96] achten nicht auf die Reihenfolge der Anweisungen — die drei Anweisungen „tricksen“ elegant die Zeigeranalysen aus.

Die Adresse von `ph_proc` selbst wird ebenfalls genommen (Zeile 12). Durchaus üblich ist die Benutzung von Rückruffunktionen im Benutzerprogramm. Konkret wird eine Funktion im Benutzerprogramm geschrieben und dessen Adresse genommen. Bei einer Bibliotheksfunktion wird die Adresse dieser Rückruffunktion

übergeben. Diese Zeile simuliert dementsprechend diesen Effekt und reduziert Rückruffunktionen auf eine einzige Rückruffunktion `ph_proc`.

Aufrufe zu sichtbaren Funktionen werden mittels eines indirekten Aufrufs über `ph_var` durch Zeile 14 sichergestellt, da `ph_var` durch Zeile 6 auf alle sichtbaren Funktionen zeigen kann. Die Variable `ph_var` selbst wird für alle Parameter verwendet. Der Datenfluss als Rückgabewert selbst fließt wieder an `ph_var` zurück. Es gibt bei diesem indirekten Aufruf `m` Parameter. `m` wird aus der maximalen Anzahl der formalen Parameter aller sichtbaren Funktionen berechnet. In dieser Zeile werden alle sichtbaren Funktionen aufgerufen, die nötigen fehlenden Datenflüsse werden ergänzt. Seiteneffekte und Rückgabewerte wirken sich auf `ph_var` aus.

Alle Funktionen, deren Adressen irgendwann genommen wurden, werden von `ph_proc` über einen direkten Aufruf aufgerufen (Zeile 16–17). Der Aufbau des Aufrufs unterscheidet sich aber ansonsten nicht von Zeile 14. Dieser künstliche Aufruf hat wiederum das Ziel, für den nötigen fehlenden Datenfluss zu sorgen. Er wird gebraucht, da es sein kann, dass die genommenen Adressen zum Benutzerprogramm propagiert werden und dort aufgerufen werden können.

Der Rückgabewert von `ph_proc` ist `ph_var` selbst (Zeile 19).

Zusätzlich zu der Erzeugung von `ph_proc` und `ph_var` wird `ph_proc` als Ziel für indirekte Aufrufe genommen. Hintergrund ist auch wieder der fehlende Datenfluss, für den `ph_proc` erzeugt wurde.

3.4.3 Beispiel

Für etwas mehr Klarheit soll das Beispiel [Rou02, Kapitel 3.2.2] sorgen, welches das im vorherigen Abschnitt Beschriebene verdeutlicht.

```
1  // Bibliothek
2  // g und exec sind sichtbar
3  global g
4  procedure exec (p, fp) {
5      local s, u, q, t
6      s = 3
7      u = 4
8      t = p
9      (*fp)(g,t)
10     q = &u
11     neg (q)
12     q = &s
13     neg (q)
14     *t = u
15     g = t
16 }
17 procedure neg (r) {
18     local i, j
19     i = *r
20     j = -i
21     *r = j
22 }
```

Die gegebene Bibliothek hat eine sichtbare Funktion `exec` und eine globale Variable `g`. Die Anzahl der Parameter `n` für `ph_proc` ist `2`, weil der indirekte Aufruf `(*fp)(g,t)` zwei Parameter hat. Der indirekte Aufruf in `ph_proc` mit `ph_var = (*ph_var) (ph_var, ph_var)` hat zwei Parameter (`m = 2`), da die sichtbare Funktion `exec` zwei Parameter hat.

```
1  // Ergaenzung der Bibliothek
2  global ph_var
3  procedure ph_proc (f1, f2) return ph_proc_ret { // n = 2
4      ph_var = f1
5      ph_var = f2
6      ph_var = &g
7      ph_var = &exec
8
9      ph_var = &ph_var
10     ph_var = *ph_var
11     *ph_var = ph_var
12
13     ph_var = &ph_proc
14
15     ph_var = (*ph_var) (ph_var, ph_var) // m = 2
16
17     ph_proc_ret = ph_var
18 }
```

3.4.4 Vor- und Nachteile

Nachteile

Ausgehend von der Ergänzung der Bibliothek zu einem vollständigen Programm wie im vorangegangenen Abschnitt erläutert, ist die Ergänzung eine **deutliche Überschätzung**. Realistische Benutzerprogramme, welche die Bibliothek verwenden, haben mehrere Variablen und Funktionen. Die Überschätzung liegt darin, dass mehrere Variablen auf eine Variable `ph_var` und mehrere Funktionen auf eine Funktion `ph_proc` fallen. Es wäre nach der Implementierung zu untersuchen, wie stark sich diese Überschätzung in der Praxis auf die Resultate der danach folgenden Analysen auswirkt. Dagegen kann argumentiert werden, dass zwar eine Überschätzung im Benutzerprogramm vorliegt, aber keine oder nur eine indirekte Überschätzung in der Bibliothek existiert. Es spielt außerdem eine Rolle, wie groß die Kopplung zwischen der Bibliothek und dem Benutzerprogramm ist — ausgedrückt in einer Metrik kann die Kopplung z. B. als das Verhältnis von sichtbaren zu unsichtbaren Funktionen sein.

An der Ergänzung ist zu sehen, dass `ph_var` **keinen speziellen Typ** besitzt. Der indirekte Aufruf `ph_var = (*ph_var) (ph_var, ... , ph_var)` achtet nicht auf die Übereinstimmung des Typs bei den Parametern und des Rückgabewerts. Es kommt auf die Sprache an, inwieweit Typsicherheit gewährleistet ist. C/C++ erlaubt als Beispiel `void`-Zeiger `void*`, die auf beliebige Elemente zeigen können. Daher benötigt die Ergänzung für die Programmiersprache C/C++ nur eine `ph_var`. In anderen Programmiersprachen mit Typsicherheit kann die Ergänzung unter

Beachtung des Typs und mehreren Platzhaltern `ph_var` möglicherweise genauere Resultate liefern.

Zu der Überschätzung trägt ebenfalls die Tatsache bei, dass kein Kontrollfluss (wie `if-else`, `for`, `while`, `switch-case` etc.) vorhanden ist. Im schlimmsten Fall kann das Benutzerprogramm tatsächlich keinen Kontrollfluss besitzen.

Die **Reihenfolge der Anweisungen**, wie deutlich an Zeile 8–10 (Listing 3.1) zu erkennen ist, spielt keine Rolle. Sollte eine auf die Fragment-Analyse aufbauende Analyse Wert auf die Flusssensitivität legen, so könnten die künstlichen Anweisungen in verschachtelten `if-else`-Zweigen platziert werden.

Zeile 6 proklamiert das Nehmen der **Adresse von allen sichtbaren Funktionen**. Dies ist gerade für die Aufrufgraphkonstruktion, die nach der Zeigeranalyse folgt, eine signifikante Verschlechterung, da die Adressen in die Bibliothek propagiert werden und sich die Ziele der indirekten Aufrufe in der Bibliothek um die sichtbaren Funktionen erhöhen. Als Lösung für dieses Problem könnte das Wissen des Analysenanwenders zur Geltung kommen, der feingranular einstellen kann, von welchen sichtbaren Funktionen die Adresse genommen werden soll. In Zeile 14 werden jedoch trotzdem alle sichtbaren Funktionen direkt anstatt indirekt aufgerufen.

Zusammengesetzte Datentypen, wie z. B. `struct` in C, finden bei der Ergänzung keine Beachtung. Sollten auf die Fragment-Analyse aufbauende Analysen zusammengesetzte Datentypen beachten, so sollte die Fragment-Analyse entsprechend weitere künstliche Anweisungen erzeugen.

Vorteile

Durch die **einfache** Ergänzung wurde es geschafft, die Bibliothek zu einem ganzen Programm zu vervollständigen. Resultate von Analysen, die auf die Fragment-Analyse aufbauen, gelten für die Bibliothek immer, egal welches Benutzerprogramm die Bibliothek später benutzt. **Aussagen über die Bibliothek** sind möglich, z. B.: „Es gibt Toten Code in der Bibliothek.“

Software ist in der Regel modular in Form von Bibliotheken aufgebaut. Herkömmliche statische Analysen gehen aber von einem kompletten Programm aus und leiden nicht selten an Laufzeit- und Speicherproblemen. Ein Benutzerprogramm benutzt oft nicht nur eine Bibliothek, sondern mehrere. Bibliotheken verwenden wiederum andere Bibliotheken für das Erledigen bestimmter Aufgaben. Die **Skalierbarkeit** der Fragment-Analyse ist deutlich besser, da ein Programm zu Einheiten (in Form von Bibliotheken) heruntergebrochen werden kann. Angenommen ein Algorithmus läuft in $\mathcal{O}(n^3)$ und die Eingabe kann in zwei gleiche Einheiten heruntergebrochen werden, dann wurden unter Einbußen der Genauigkeit Dreiviertel der Zeit gespart: $(\frac{1}{2}n)^3 + (\frac{1}{2}n)^3 = \frac{1}{4}n^3$.

Ein Bibliotheksanbieter kann ohne die Fragment-Analyse keine Analysen auf seine Bibliothek anwenden, da seine Bibliothek zu einem beliebigen Benutzerprogramm

passen muss. Durch die modulare Analyse der Bibliothek ist es nun auch möglich, z. B. interprozedurale Compiler-Optimierungen für die Bibliothek zu benutzen.

Es gibt zwei gegensätzliche Analysetypen: modulare Analysen [CC02] und herkömmliche Gesamtprogramm-Analysen. Modulare Analysen berücksichtigen, dass das Programm (oder besser gesagt die Bibliothek) nicht vollständig ist. Die Aufgabenstellung fordert die Untersuchung für Zeigeranalysen, Aufrufgraphkonstruktion, Seiteneffektberechnung, ISSA-Analyse und GUI-Analyse. Diese Analysen existieren alle bereits in Bauhaus als Gesamtprogramm-Analysen. Pragmatisch gesehen würde eine modulare Neuimplementierung all dieser Analysen den Rahmen der Diplomarbeit weit übersteigen. Die Fragment-Analyse schafft hier eine Brücke und vervollständigt eine Bibliothek zu einem Programm. Der Vorteil liegt auf der Hand, (genannte) **existierende Analysen und Werkzeuge** können benutzt und auf eine Bibliothek angewendet werden.

Die **Wiederverwendung** der getrennten Analyse bringt den Vorteil, dass die Bibliothek zuvor kompaktifiziert werden kann. Rountev löscht redundante Variablen und Anweisungen und beschleunigt dadurch die späteren Analysen auf das Benutzerprogramm und die Bibliothek als eine Einheit [RR01]. Die **Kompaktifizierung** wird unter einem anderen Schwerpunkt im Auge behalten. Dadurch soll ebenfalls Laufzeit- und Speicherersparnis erzielt werden.

3.5 Rollenpropagierung

Die Rollenpropagierung wurde von Staiger in [Sta07a] vorgestellt. Sie wurde benutzt, um für die GUI-Analyse „Einbetter“, „Konstruktor“ oder „Setzer“ zu identifizieren. „Konstruktoren“ erzeugen Widgets, „Setzer“ setzen Attribute für das vom „Konstruktor“ erzeugte Widget und „Einbetter“ setzen mehrere Widgets, die ineinander verschachtelt sind, zu einem Fenster zusammen.

Am Anfang der Berechnung müssen Initialrollen festgelegt werden. Diese werden interprozedural entlang des Aufrufgraphen und intraprozedural entlang von Definitionen (Zuweisungen) und Uses (Verwendungen) über ϕ-Knoten der ISSA-Analyse rückwärts propagiert. Interprozedural hängen die Rollen direkt an einem Parameter einer Prozedur. Am Ende der Berechnung hängt beispielsweise die Rolle „Konstruktor“ am zweiten Parameter der Funktion `f`. Dies bedeutet, dass, wenn die Funktion `f` aufgerufen wird, über den zweiten Parameter ein Widget erzeugt wird.

Die Rollenpropagierung existiert als generisches Paket in Bauhaus und kann verwendet werden. Die am Anfang festzulegenden Rollentypen sollen vom Benutzer des generischen Pakets bestimmt werden.

Um die Rollenpropagierung zu veranschaulichen, wird nun anhand von Aufrufgraph und Rückruffunktion die Arbeitsweise der Rollenpropagierung an Beispielen näher erläutert.

3.5.1 Aufrufgraph

Beispiel 1

```
1  int f (int (*param_fp) (int)) {
2
3      int x = (*param_fp)(4);
4      return x + 2;
5  }
```

Das erste einfache Beispiel hat nur eine Funktion `f` mit einem Funktionszeiger `param_fp` als Parameter. Über diesen Parameter kann beim Aufruf der Funktion `f` eine Adresse für eine Rückruffunktion übergeben werden. Die Rückruffunktion wird indirekt in `f` in Zeile drei aufgerufen.

Die Initialrollen werden an jedem indirekten Aufruf erzeugt. In diesem Beispiel wird eine Rolle in Zeile drei erzeugt, sie hängt am ersten Parameter der Funktion `f`.

Natürlich kommt es auch vor, dass der Parameter nicht sofort in `f`, sondern erst später aufgerufen wird. `param_fp` wird also an Funktionen weitergereicht und erst in einer anderen Funktion aufgerufen.

Beispiel 2

```
1  int f (int (*param_fp) (int)) {
2      int x = (*param_fp)(20);
3      return x;
4  }
5
6  int g (int (*param_fp) (int)) {
7      if (...) {
8          return 1;
9      } else {
10         return f (param_fp);
11     }
12 }
13
14 int h (int (*param_fp) (int)) {
15     if (...) {
16         return g (param_fp);
17     } else {
18         return 0;
19     }
20 }
```

Beispiel 2 verdeutlicht das Weiterreichen von Parametern. Angenommen die Funktion `h` wird als Schnittstellenfunktion einer Bibliothek aufgerufen, `h` ruft wiederum `g` auf und gibt `param_fp` weiter. `g` ruft `f` auf und gibt `param_fp` ebenfalls weiter. Erst in `f` erfolgt ein indirekter Aufruf auf den zweimal weitergereichten Parameter `param_fp`.

Abbildung 3.9 stellt den Rollenpropagierungsgraphen für das eben genannte Beispiel dar. Ganz unten in der Abbildung wird für `f` eine Initialrolle (Zeile 2 des

angegebenen Quellcodes) erzeugt, diese wird rückwärts über `g` bis `h` weiterpropagiert.

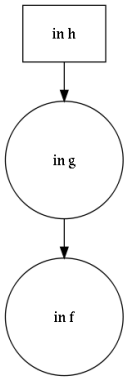

Abbildung 3.9: Rollenpropagierungsgraph für Beispiel 2

Für die Aufrufgraphkonstruktion lässt sich für eine Schnittstellenfunktion der Bibliothek also feststellen, dass über den *i*-ten Parameter der Funktion `h` irgendwann ein indirekter Aufruf erfolgen wird.

3.5.2 GUI-Analyse

Bei GUI-Bibliotheken soll analysiert werden, wie mittels der Rollenpropagierung Initialrollen („Einbetter“, „Konstruktor“ oder „Setzer“) automatisch erkannt werden können. Diese werden im Moment ohne eine Analyse der Bibliothek im Benutzerprogramm über eine Konfigurationsdatei händisch vom Analysenanwender gesetzt. Die Konfigurationsdatei wird einmal pro GUI-Bibliothek benötigt und sieht z. B. vor, für alle gtk-Funktionen, die den Namen `gtk_*_new` tragen, eine Initialrolle für „Konstruktor“ zu erzeugen.

Für die Erkennung eines „Konstruktors“ ist es vorstellbar, Initialrollen an speicherallokierten Stellen[15] zu erzeugen und diese zu den Schnittstellenfunktionen zu propagieren. Leider ist dies für gtk und Qt nicht einfach, da z. B. gtk `g_malloc` und Qt `qMalloc` für das Allokieren von Speicher benutzen, was nicht einheitlich ist. Das Speicherallokieren wird außerdem nicht für das Erzeugen von Widgets alleine benötigt, sondern auch für andere Datenstrukturen, die innerhalb der Bibliothek benötigt werden. Es läuft darauf hinaus, wieder eine fein eingestellte Konfigurationsdatei pro GUI-Bibliothek zu haben, um andere Fälle, die keine Erzeugung eines Widgets sind, aber Speicher allokieren, zu filtern. Diese Aufgabe erscheint dem Autor als nicht sinnvoll und wurde offen gelassen. Hinzu kommt das Problem des Speichermangels für gtk. Die in 3.5.1 beschriebene Benutzung der Rollenpropagierung für die Rückruffunktion musste aufgrund von Speichermangel abgebrochen werden.

[15] in C ist das z. B. die Benutzung der Funktion `malloc` aus der Standardbibliothek

3.6 Verwandte Arbeiten

Einen guten Überblick über statische modulare Programmanalyse geben P. Cousot und R. Cousot in [CC02] im Jahr 2002. Ziel dieses Abschnitts ist es nicht, einen umfassenden Überblick zu geben. Er ist vielmehr eine Ergänzung zu Cousot und Cousots Überblick durch eine neuere Arbeit von Gopan und Reps [GR07] und von Sharp et al. [RSX08, Sha07]. Es wird versucht diese zwei Arbeiten sowie die Component-Level-Analysis und die Fragment-Analyse den vier aufgestellten Kategorien von Cousot und Cousot zuzuordnen.

3.6.1 Low-Level-Analysis

Gopan und Reps stellen in ihrer Arbeit [GR07] einen interessanten Ansatz vor. Ausgehend von der Tatsache, dass der Quellcode nicht vorhanden ist, z. B. bei der Programmierung in proprietären Umgebungen oder mit kommerziellen Bibliotheken, untersuchten sie binären Assembler-Code von Bibliotheken für die x86-Architektur. Zentraler Punkt ihrer Arbeit war dabei, Assembler-Code wieder in Zuweisungen, Befehle und einfache Konstrukte wie `if` oder `goto` etc. umzuwandeln. Diese lagen als Zwischendarstellung vor. Durch die Umwandlung konnte die Information über Daten- und Kontrollfluss gewonnen werden. Somit war statische Programmanalyse (eingeschränkt) möglich.

Dabei traten eine Reihe von Problemen auf, die Gopan und Reps (teilweise) gelöst haben. Hier wird ein Abriss ohne die konkret vorgeschlagenen Lösungen der Arbeit vorgestellt:

- Es existiert auf der hardwarenahen Ebene so gut wie kein Typsystem und keine Typsicherheit.
- Ein Register wird zeitabhängig für verschiedene Variablen benutzt.
- Nutzung von prozessorspezifischen Operationen, wie die `shift`-Operation für Modulo- und Divisionsberechnung oder `alignment`, um mit einer Operation gleichzeitig z. B. vier Addition durchzuführen.
- Bestimmte Muster an Operationen treten auf, um z. B. den Offset von mehrdimensionalen Arrays zu berechnen.
- `jump`-Anweisungen werden benutzt, um `for` und `while`-Schleifen zu simulieren.

Der Vorteil dieser Vorgehensweise ist der mögliche Umgang mit fremden Bibliotheken, deren Quellcode dem Analysenanwender nicht vorliegt.

Erst auf den zweiten Blick eröffnen sich einige Nachteile. Das Vertauschen der Instruktionen aufgrund von Compiler- und hardwarespezifischen Optimierungen wurde bei der Vorgehensweise außer Acht gelassen und würde die Umwandlung in die Zwischendarstellung erheblich erschweren. Die Vorgehensweise ist sehr hardwarenah, sie bleibt eng an den Eigenheiten der x86-Architektur. Geschilderte

Probleme aus 3.1 wurden nicht beachtet, es erfolgt keine Beachtung von Benutzerprogrammen (die fehlen), globalen Variablen, oder interprozeduralen Analysen. Dadurch wird die Betrachtung intraprozedural und lokal auf den Datenfluss einer Funktion beschränkt.

3.6.2 IDE-Dataflow-Analysis

In [RSX08] und [Sha07, Kapitel 3] stellten Sharp et al. eine veränderte Definition von 2.3 vor. Sie setzten ihre Ideen für Summary Edges [HRB90] und Typanalyse um.

Ausgehend von der Grundlage in 2.3 formulierten sie eine andere Definition. Sei D die Menge aller lokalen Variablen und formalen Parameter und L die Potenzmenge über die formalen Parameter mit $\supseteq$ als Ordnungsrelation und $\cup$ als Konfluenzoperator.

Für die Berechnung der Summary Edges drückt $env(d) \in Env(D, L)$ die (transitive) Abhängigkeit von d zu L aus. Für eine Zuweisung eines Ausdrucks, abhängig von $d_1 \dots d_k$ an d mit $d := expr\{d_1, \dots, d_k\}$, gilt $env[d \mapsto \bigcup_i env(d_i)]$. Das heißt, alle bisherigen Abhängigkeiten von d_i werden auf d übertragen. Enthält die rechte Seite keine Variable, z. B. bei Zuweisung einer Konstanten $d := 4$, so gilt $env[d \mapsto \emptyset]$. Ein Aufruf $d := m(d_1, \dots, d_k)$ kann als eine Folge von Zuweisungen von aktuellen zu formalen Parametern betrachtet werden, gefolgt von einer zusammenfassenden Funktion sowie Zuweisung des Rückgabewertes von m an d. Interessant ist nun die Darstellung der Transferfunktion als ein bipartiter Graph mit $2(|D| + 1)$ Knoten. In jeder Partition gibt es $|D|$ mit d_i beschriftete Knoten und einen Knoten Λ. Die Kanten des bipartiten Graphen drücken aus, wie die Abhängigkeiten nach einer Anweisung transformiert werden. Der Vorteil dieser Darstellung ist bei einer Sequenz von Anweisungen die Umwandlung von mehreren bipartiten Graphen zu einem bipartiten Graphen als eine kompakte Repräsentation. Beim Zusammenfluss können zwei bipartite Graphen zu einem verschmolzen werden. Hierzu zwei Beispiele:

Beispiel 1

```
1  b = 5;
2  a = b + c;
3  d = 2 * a;
```

Auf der linken Seite der Abbildung 3.10 werden drei bipartite Graphen für die Anweisungen 1–3 dargestellt. Auf der rechten oberen Seite wird für zwei bipartite Graphen der Anweisungen 1–2 ein neuer Graph als kompakte Darstellung erstellt, rechts unten für alle drei Anweisungen. Wie auf der linken Seite zu erkennen ist, hängt `d` von `a` ab, `a` wiederum hängt von `c` ab. Dadurch entsteht eine transitive

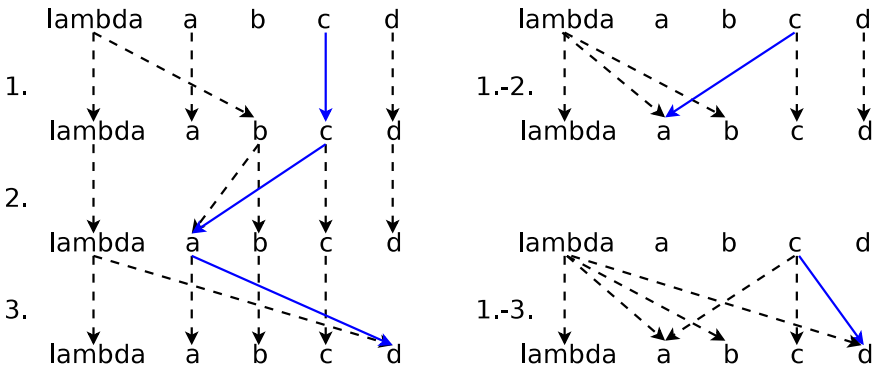

Abbildung 3.10: Bipartite Graphen für die Hintereinanderausführung

Abhängigkeit von `c` zu `d`, die links visuell und rechts unten kompakt dargestellt ist.

Beispiel 2

```
1  if (...) {
2      a = b + c;
3      c = d;
4  } else {
5      a = d;
6      c = d;
7      b = c;
8  }
```

Abbildung 3.11 stellt den Zusammenfluss des Beispiels 2 dar. Beide Graphen der `if`- und else-Zweige werden verschmolzen. Knoten bleiben identisch, neue Kanten werden übernommen, es gibt keine Doppelkanten.

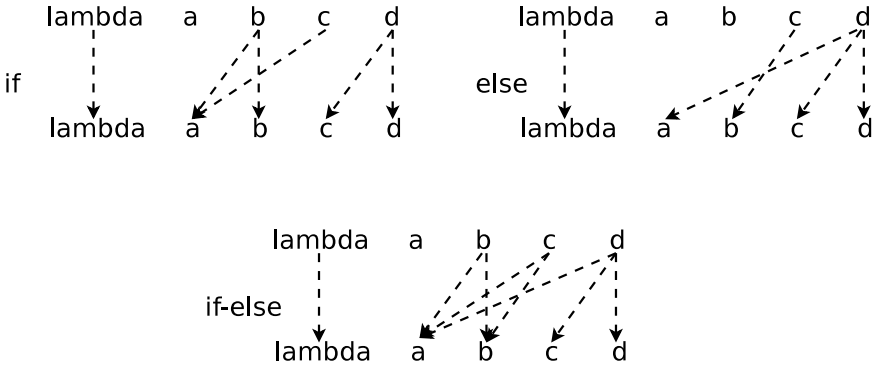

Abbildung 3.11: Bipartiter Graph für den Zusammenfluss

Ein weiterer Aspekt der Kompaktifizierung sind lokale Variablen, die zwar während der intraprozeduralen Berechnung benötigt werden, aber interprozedural keine Rolle spielen. Nach der intraprozeduralen Berechnung wird der bipartite Graph,

der eine Funktion `m` repräsentiert, auf formale Parameter, den Rückgabewert und deren Abhängigkeiten reduziert.

Um auf die eigentlich in 3.1 geschilderten Schwierigkeiten der Analysen mit Bibliotheken zurückzukommen, benutzten Sharp et al. die gleiche Methode wie die Component-Level-Analysis. Intraprozedural erfolgt die Berechnung für Teilpfade wie in den zwei Beispielen geschildert. Es werden, soweit bekannt, Teilpfade in bipartite Graphen kompaktifiziert. Für die interprozedurale Berechnung trägt ein Aufruf einer Methode die Eigenschaft **fest**, wenn zur Analysezeit der Bibliothek feststeht, welche Funktionen aufgerufen werden. Feste Aufrufe können mittels bipartiter Graphen ersetzt werden, falls sie rekursiv auch wieder fest sind. Nicht feste Aufrufe, z. B. durch Funktionszeiger oder polymorphe Aufrufe in der Objektorientierung, werden offen gelassen. Es handelt sich bei diesem Ansatz also auch wieder um eine Voranalyse. Zur Analysezeit des Benutzerprogramms werden voranalysierte Ergebnisse in Form von bipartiten Graphen benutzt, um die eigentliche Analyse zu beschleunigen. Die Voranalyse ist, im Vergleich zu der Fragment-Analyse, keine Annahme des schlimmsten Falls, sondern bleibt präzise.

Für die Typanalyse wird auf die Originalarbeit verwiesen. Die Vorgehensweise ist ähnlich wie bei den eben geschilderten Summary Edges.

Bis auf die Darstellung der Transferfunktion als kompaktem bipartitem Graph ähnelt dieser Ansatz stark der Vorgehensweise der Component-Level-Analysis von Rountev aus 3.3. Bezüglich der Vor- und Nachteile wird daher auf 3.3.7 verwiesen.

3.7 Vergleich der Ansätze und Entscheidung

3.7.1 Klassifikation der Ansätze

Um die vorgestellten Analysen im groben Kontext einordnen zu können, bedarf es einer Klassifikation, die hier vorgestellt wird.

Vorhandensein von Quellcode

Eine erste Klassifikation ist die Frage nach dem Vorhandensein von Quellcode von Bibliotheken oder Programmen. Ist der Quellcode nicht vorhanden, so können nur in Binärform Analysen betrieben werden. Bei den vorgestellten Arbeiten gehen nur Gopan und Reps (3.6.1) vom Fehlen des Quellcodes aus.

Gesamtprogramm-Analyse oder modulare Analyse

Wie eingangs in der Motivation erwähnt, gehen klassische Verfahren von einem ganzen Programm samt Bibliotheken als eine Einheit aus. Modulare Analysen gehen nicht von dieser Annahme aus und können auch auf Teile angewendet

werden. Alle in dieser Arbeit vorgestellten Verfahren, insbesondere die Component-Level-Analysis (3.3) und die Fragment-Analyse (3.4), sind modulare Analysen. Klassische Verfahren, wie beispielsweise die Berechnung der gültigen Definitionen, sowie Analysen in Bauhaus sind Gesamtprogramm-Analysen.

Modulare Analysen nach Cousot und Cousot

Cousot und Cousot haben 2002 modulare Analysen in vier Kategorien[16] eingeteilt [CC02]. Diese werden genannt, die in dieser Arbeit vorgestellten Ansätze gehören zu zwei der vier Kategorien, alle werden kurz erläutert:

1. **Simplification-based Separate Analysis**
2. **Worst-Case Separate Analysis**
3. **Separate Analysis with (User-Provided) Interfaces**
4. **Symbolic Relational Separate Analysis**

(1.) Eine Voranalyse wird durchgeführt. Zur Analysezeit schon feststehende Pfade können vereinfacht oder kompaktifiziert werden. Das Ergebnis der Voranalyse wird benutzt und zu der eigentlichen Analyse verbunden, um Speicher- und Laufzeiteinsparungen bei der eigentlichen Analyse zu erzielen. Zu dieser Kategorie gehören Rountevs Component-Level-Analysis (3.3) und Sharps Arbeit (3.6.2).

(2.) Die separate Analyse nimmt den schlechtesten Fall für den unbekannten Teil der Software an und analysiert auf dieser Basis den bekannten Teil. Rountevs Fragment-Analyse (3.4) gehört zu dieser Kategorie.

(3.) Die separate Analyse verlangt vom Analysenanwender Informationen oder Zusicherungen für den unbekannten Teil, um das Ergebnis der Analyse für den bekannten Teil zu verbessern.

(4.) Mittels der Abstrakten Interpretation[17] werden unbekannte Teilpfade abstrahiert. Die Abstraktion reicht aus, um Berechnungen im bekannten Teil durchzuführen oder die Berechnung auf später verschieben zu können.

Tabelle 3.1 gibt die erläuterten Klassifikationen noch einmal wieder, die vorgestellten Ansätze werden zugeordnet. CC-Kat ist die Abkürzung für die vorgeschlagenen Kategorien der modularen Analyse nach Cousot und Cousot [CC02].

[16] Die englischen Titel wurden der Einfachheit halber beibehalten

[17] Das geschickte Weglassen von Informationen, um auf grobgranularen Aussagen Berechnungen durchzuführen. Als Beispiel wird `x := 4 + 2` in `x := int + int` umgewandelt, um den Typ von `x` zu berechnen. Für weitere Erklärungen wird auf einschlägige Literatur verwiesen.

Ansatz	Quellcode	modular	CC-Kat 1	CC-Kat 2
Ignorieren (3.2.1)	–	–	–	–
Schlimmster Fall (3.2.2)	–	–	–	–
Attrappen (3.2.3)	–	–	–	–
Analyse mit Bibl. (3.2.4)	√	–	–	–
Component-Level-A. (3.3)	√	√	√	–
Fragment-Analyse (3.4)	√	√	–	√
Rollenpropagierung (3.5)	–	√	–	–
Low-Level-A. (3.6.1)	–	√	–	–
IDE-Dataflow-A. (3.6.2)	√	√	√	–

Tabelle 3.1: Klassifikation der Ansätze im Überblick

3.7.2 Entscheidung

Um zu einer Entscheidung zu kommen, welcher Ansatz der Geeignetste ist, wurden alle Ansätze in diesem Kapitel ausführlich diskutiert. In der Diskussion wurden die besonderen Stärken und Schwächen hervorgehoben. Die Tabelle 3.2 fasst alle Argumente für alle Ansätze noch einmal zusammen und gibt eine Übersicht in kompakter Form.

Die ersten drei Spalten gehen auf die in 3.1 genannten Schwierigkeiten hinsichtlich Funktionszeigern, fehlenden (interprozeduralen) Datenflüssen und Vererbung ein. Zusätzlich wurden die Ergebnisse des jeweiligen Ansatzes nach Qualität bewertet. Die Bewertung des Aufwands bezieht sich sowohl auf den Analysenprogrammierer als auch auf den Analysenanwender. Die Spalte für Speicher und Laufzeit bezieht sich auf die Laufzeit der Analyse sowie das Abspeichern der Ergebnisse oder den Speicherverbrauch während der Berechnung. Zusätzlich wurde noch betrachtet, wie leicht existierende Gesamtprogramm-Analysen in Bauhaus angepasst werden müssten, um die Ansätze umsetzen zu können.

Nach einer ausführlichen Betrachtung verschiedener Ansätze mit anschließender Diskussion und des Abwägens der Argumente wurde festgestellt, dass die Fragment-Analyse am geeignetsten ist. Die in dieser Arbeit umgesetzte Implementierung ist die Fragment-Analyse. Sie wird im nächsten Kapitel erweitert.

Die Rollenpropagierung wird nach der Fragment-Analyse anschließend benutzt, um die Aufrufgraphkonstruktion im Benutzerprogramm zu verbessern.

Ansatz	FktZeiger	Fehl. Datenfl.	Vererbung	Qualität	Aufwand	Speicher/Laufzeit	Exist. Analysen
Ignorieren (3.2.1)	−	−	−	−−	++	++	+
Schlimmster Fall (3.2.2)	√	√	√	−−	++	++	+
Attrappen (3.2.3)	√	√	√	−	−−	+	++
Analyse mit Bibl. (3.2.4)	√	√	√	++	++	−−	++
Component-Level-A. (3.3)	√	√	√	++	−	+	−−
Fragment-Analyse (3.4)	√	√	√[1]	◯[2]	◯	+	++
Rollenpropagierung (3.5)	−	−	−	+	−	+	+
Low-Level-A. (3.6.1)	−	−	−	−	−	◯	+
IDE-Dataflow-A. (3.6.2)	√	√	√	++	−	+	−−

− nicht erfüllt; √ erfüllt

−− sehr schlecht; − schlecht; ◯ neutral; + gut; ++ sehr gut

[1] Die ursprüngliche Fragment-Analyse behandelt nur C. Die Erweiterung der Fragment-Analyse für C++ erfolgt in 4.2.
[2] Die Frage nach der Qualität der Fragment-Analyse wird in Kapitel 7 beantwortet.

Tabelle 3.2: Vergleich und Bewertung der Ansätze

4 Erweiterung der Fragment-Analyse

Abschnitt 3.4 beschreibt die Fragment-Analyse allgemein. Rountev setzte die Idee der Fragment-Analyse nur für die Sprache C um [RR01]. In [RMR03, RMR04] wurde die Sprache Java behandelt, jedoch ohne die Annahme des schlimmsten Falles. In dieser Arbeit wurden vielmehr fehlende Datenflüsse ergänzt, um Tests für Quellcode-Überdeckungswerkzeuge[1] zu generieren. Auch die Nachteile der Fragment-Analyse wurden im Abschnitt 3.4.4 ausführlich diskutiert. In diesem Abschnitt wird eine Verbesserung vorgestellt, die versucht, die Nachteile der Fragment-Analyse zu beheben. Außerdem wird ein Ansatz für C++ bzw. Objektorientierung angegeben, dementsprechend ist der angebende Quellcode an C/C++ angepasst.

4.1 Typsensitivität

Der Hauptnachteil der Fragment-Analyse ist, wie im Kapitel 3.4.4 bereits angesprochen, die deutliche Überschätzung durch die Ergänzung **einer** Platzhalterfunktion `ph_proc` und **einer** Platzhaltervariablen `ph_var`. Hier werden zuerst einige Statistiken präsentiert, um die Verbesserung zu diskutieren und zu untermauern.

Statistik über Parameter und Variablen

Name	Sichtb. Fkt & Var	Anz. Var	Anz. unterschiedl. Typen
glpk	137 + 0	499	14
jpeg	48 + 0	1940	24
ncurses	154 + 30	453	33
sqlite	108 + 2	619	35
xvid	3 + 0	5775	29

Tabelle 4.1: Anzahl der unterschiedlichen benutzten Typen

In der Tabelle 4.1 werden die betrachteten Bibliotheken dargestellt. Spalte zwei gibt Auskunft über die Anzahl der unterschiedlichen sichtbaren Funktionen und Variablen als Schnittstelle der jeweiligen Bibliothek. Spalte drei zählt alle sichtbaren Variablen, alle formalen Parameter von sichtbaren Funktionen und alle formalen Parameter von indirekten Aufrufen der Bibliothek auf. Die in Spalte drei betrachteten Variablen werden auf ihre unterschiedlichen Typen untersucht, für die Bibliothek

[1] Englisch: code coverage tool

`glpk` ergeben sich beispielsweise für 499 Variablen oder Parameter insgesamt 14 unterschiedliche Typen.

Im Hinblick auf Funktionszeiger werden Signaturen von sichtbaren Funktionen gesondert betrachtet. Ähnlich wie 4.1 untersucht Tabelle 4.2 (Spalte drei) die Anzahl der unterschiedlichen Signaturen für sichtbare Funktionen.

Statistik über Funktionen

Name	Sichtb. Fkt	Anz. unterschiedl. Typen
glpk	137	29
jpeg	48	25
ncurses	154	87
sqlite	108	56
xvid	3	2

Tabelle 4.2: Anzahl der unterschiedlichen benutzten Typen für sichtbare Funktionen

Tabelle 4.3 gibt noch einmal ein Gesamtbild wieder. Betrachtet werden **alle relevanten Objekte** mit ihren Typen. Diese sind alle sichtbaren Funktionen und deren formale Parameter, sowie sichtbare Variablen, indirekte Aufrufe und deren formale Parameter. Hierzu zählen auch Funktionen mit den dazugehörenden formalen Parametern, deren Adressen genommen wurden.

Statistik aller vorkommenden Typen

Name	Anz. Var	Anz. unterschiedl. Typen
glpk	750	53
jpeg	2639	77
ncurses	646	121
sqlite	1330	153
xvid	6849	40

Tabelle 4.3: Anzahl aller vorkommenden relevanten Typen

Angesichts der dargestellten Zahlen wäre es ungenau zu sagen, dass es nur einen Platzhalter `ph_var` gibt. Wie die Tabelle 4.3 verdeutlicht, gibt es beispielsweise für `sqlite` schon 153 unterschiedlichen Typen. Die Idee ist also **für jeden unterschiedlichen Typ** `t` **einen Platzhalter** `ph_var_t` zu erzeugen, um auf diese Weise eine höhere Genauigkeit bei nachfolgenden Analysen zu erhalten.

Der Ansatz hat jedoch einen Nachteil und es wird nicht der schlimmste Fall simuliert, da insbesondere die Programmiersprache C keine strenge Typsicherheit

bietet [KR88, Kapitel 2.7, 5. und 6.4]. Hier sind ein paar Beispiele aufgezählt, die in C möglich sind, selbst bei Benutzung des gcc-Compilers mit Warnoption `-W`.

```
1  /* Beispiel 1 */
2  int a;
3  float f;
4
5  a = f;
6  f = a;
```

Beispiel 1. Durch implizite Konvertierung kann problemlos eine Variable vom Typ Float einer Variablen vom Typ Integer zugewiesen werden.

```
1  /* Beispiel 2 */
2  void *v;
3  int *pint;
4  int i;
5
6  v = pint;
7  pint = v;
8  v = &pint;
9  v = &i;
```

Beispiel 2. Typenlose Zeiger `void*` können auf beliebige Datenstrukturen zeigen.

```
1  /* Beispiel 3 */
2  struct struct_a *pa;
3  struct struct_b *pb;
4
5  pa = (struct struct_a*)pb;
6  pb = (struct struct_b*)pa;
7
8  v = pa;
9  pb = v;
```

Beispiel 3. Mittels einer C-Cast-Operation lassen sich Variablen zuweisen, die inkompatibel sind und somit gefährlich sein können.

Wenn vom schlimmsten Fall ausgegangen wird, so dürfte nur eine `ph_var` erzeugt werden, da Datenflüsse zwischen unterschiedlichen Typen existieren können. Die Erzeugung von verschiedenen `ph_var` ist also **nicht konservativ**.

Andererseits sind die drei konstruierten Beispiele nicht sehr praxisnah, da sie sehr leicht zu Fehler führen und selten im realen Quellcode wiederzufinden sind.

Als Kompromiss wurden in der Implementierung beide Versionen umgesetzt. Der Analysenanwender darf bei der Analyse entscheiden, wie konservativ er analysieren will.

Im Folgenden wird von der **typinsensitiven** Rountev-Version gesprochen, die Verbesserung dagegen ist **typsensitiv**.

Die nächsten Schritte sind erforderlich, um die Originalversion von Rountev in eine typsensitive Version umzuwandeln.

Originalversion von Rountev mit C-Syntax

```
1  int ph_var;
2  int ph_proc (int f_1, ... , int f_n) {
3
4      ph_var = f_i; // Fuer i von 1 bis n
5
6      ph_var = &v; // Fuer alle nach aussen sichtbaren Variablen
7      ph_var = &func_i; // Fuer alle nach aussen sichtbaren Funktionen
8
9      ph_var = &ph_var;
10     ph_var = *ph_var;
11     *ph_var = ph_var;
12
13     ph_var = &ph_proc;
14
15     // Alle nach aussen sichtbaren Funktionen und alle Funktionen
16     // deren Adresse genommen wurde.
17     for (;;) {
18         // Fuer Funktionen mit Rueckgabewert void
19         func_i (ph_var, ... , ph_var);
20         // Fuer Funktionen mit Rueckgabewert nicht void
21         ph_var = func_i (ph_var, ... , ph_var);
22     }
23
24     return ph_var;
25 }
26
27 // Alle indirekten Aufrufe in der Bibliothek rufen ph_proc auf:
28 (*fp1) (10, 'c');
29 int i = (*fp2) (2.4);
```

Angegeben ist der aus Abschnitt 3.4.2 bekannte Pseudocode, umgewandelt in C-Quellcode. Der einzige Unterschied liegt in Zeile 17–21. Aus den ursprünglich indirekten Aufrufen werden direkte Aufrufe zu sichtbaren Funktionen und Funktionen erzeugt, deren Adressen genommen wurden. Für Funktionen ohne Rückgabewert erfolgt keine Zuweisung zu `ph_var`.

Leichte Veränderung mit ph_wrapper

```
1  int ph_var;
2
3  void ph_proc () {
4
5      ph_var = &v; // Fuer alle nach aussen sichtbaren Variablen
6      ph_var = &func_i; // Fuer alle nach aussen sichtbaren Funktionen
7
8      ph_var = &ph_var;
9      ph_var = *ph_var;
10     *ph_var = ph_var;
11     ph_var = &ph_proc;
12
13     // Alle nach aussen sichtbaren Funktionen und alle Funktionen
14     // deren Adresse genommen wurde
15     for (;;) {
16         // Fuer Funktionen mit Rueckgabewert void
```

```
17          func_i (ph_var, ... , ph_var);
18          // Fuer Funktionen mit Rueckgabewert nicht void
19          ph_var = func_i (ph_var, ... , ph_var);
20      }
21  }
22
23  int ph_wrapper (f_1, ... , f_n) {
24      ph_var = f_i; // fuer i von 1 bis n
25
26      return ph_var;
27  }
28
29  // Alle indirekten Aufrufe in der Bibliothek rufen ph_wrapper auf
30  (*fp1) (10, 'c');
31  int i = (*fp2) (2.4);
```

Als nächster Zwischenschritt wird eine weitere künstliche Funktion `ph_wrapper` als Datenflussbrücke hinzugefügt. Alle indirekten Aufrufe rufen `ph_wrapper` anstatt `ph_proc` auf. Dagegen hat sich die Signatur von `ph_proc` geändert, `ph_proc` hat keine formalen Parameter mehr.

Veränderung für typsensitive Analyse

```
1   // Betrachtet werden
2   // alle sichtbaren Variablen;
3   // alle Parameter und der Rueckgabewert von allen sichtbaren Funktionen;
4   // alle Parameter und der Rueckgabewert von allen indirekten Aufrufen;
5   // alle Parameter und der Rueckgabewert von Funktionen,
6   //     deren Adresse genommen wurde;
7   // alle gerade aufgezaehlten Funktionen an sich.
8   // Es existiert fuer jeden eindeutigen Typ type eine Variable ph_var_t.
9   type ph_var_t;
10
11  // Alle Zuweisungen und Aufrufe in ph_proc haben ebenfalls passende Typen.
12  void ph_proc () {
13
14      ph_var_t = &v; // Fuer alle nach aussen sichtbaren Variablen
15      ph_var_t = &func_i; // Fuer alle nach aussen sichtbaren Funktionen
16
17      // Fuer jeden Typ t
18      ph_var_t = &ph_var_t;
19      ph_var_t = *ph_var_t;
20      *ph_var_t = ph_var_t;
21      ph_var_void = &ph_proc;
22
23      // Alle nach aussen sichtbaren Funktionen und alle Funktionen
24      // deren Adresse genommen wurde
25      for (;;) {
26          // Fuer Funktionen mit Rueckgabewert void
27          func_i (ph_var_t, ... , ph_var_t);
28          // Fuer Funktionen mit Rueckgabewert nicht void
29          ph_var_t = func_i (ph_var_t, ... , ph_var_t);
30      }
31  }
32
33  // Fuer jeden indirekten Aufruf existiert eine Funktion
34  // ph_wrapper mit dem passenden Typ.
```

```
35 void ph_wrapper_1 (int f_1, char f_2) {
36     ph_var_int = f_1;
37     ph_var_char = f_2;
38 }
39 int ph_wrapper_2 (double f_1) {
40     ph_var_double = f1;
41     return ph_var_int;
42 }
43
44 // Alle indirekten Aufrufe in der Bibliothek rufen den
45 // passenden ph_wrapper auf
46 (*fp1) (10, 'c');
47 int i = (*fp2) (2.4);
```

Als letzter Umwandlungsschritt wird, passend zu den indirekten Aufrufen, jeweils eine Funktion `ph_wrapper` erzeugt. Zeile 9 erzeugt nicht eine `ph_var` wie bisher, sondern für jeden benötigten Typ `type` eine `ph_var_t`. Alle Zuweisungen werden dem Typ entsprechend zugewiesen, ebenso alle Funktionen mit ihren Parametern und Rückgabewerten.

Wie am angegebenen Quellcode zu erkennen ist, ist es durch die künstlichen Datenflussbrücken `ph_wrapper` und verschiedene `ph_var_t` gelungen, jede künstlich erzeugte Anweisung inklusive Funktionsaufruf zum Typ passend zu erzeugen.

Wie die Tabelle 4.3 darstellt, werden für die untersuchten Bibliotheken zwischen 40 und 153 verschiedene `ph_var_t` erzeugt. Dadurch wird eine höhere Genauigkeit als die Erzeugung einer `ph_var` erwartet.

Weiterhin ungenau bleibt die Möglichkeit, dass im Normalfall verschiedene Variablen eines Typs benutzt werden. Bei der Ergänzung der verschiedenen `ph_var_t` verschmelzen verschiedene Variablen eines Typs zu einer Variablen `ph_var_t`.

Nachteil

Es sollte noch kurz erwähnt werden, dass die typsensitive Fragment-Analyse auch einige Nachteile mit sich bringt. Es existieren mehr globale Variablen `ph_var_t`. Daraus ergeben sich mehr ϕ-Knoten und damit mehr Berechnungsaufwand und Speicherverbrauch.

Feldsensitivität

Sicherlich wäre noch zu überdenken beispielsweise bei Strukturen `struct`:

```
1 struct name {
2     char vname[20];
3     char nname[20];
4 };
```

auch Zuweisungen zu den Feldern, hier `vname` und `nname`, zu haben. Zum Zeitpunkt der Implementierung berücksichtigen die auf die Fragment-Analyse aufbauende Zeigeranalysen in Bauhaus die Feldsensitivität ebenfalls nicht. Daher wurde die

Feldsensitivität nicht beachtet. Für eine mögliche Umsetzung der Feldsensitivität wird hier auf dem folgenden Abschnitt verwiesen.

4.2 Ansatz für C++ und Objektorientierung

Die ursprüngliche Fragment-Analyse wird nun für C++ und die Objektorientierung erweitert. Der Ansatz lehnt sich an die Arbeit von Rountev [RMR03] [RMR04]. Jedoch sind die Zielsprachen unterschiedlich. Rountev hat das Problem für Java gelöst, um Bibliotheksquellcode mit einem Werkzeug auf Quellcode-Überdeckung überprüfen zu können. Bei dieser Arbeit liegt der Fokus auf C++. In Kapitel 3.4 und 3.6 wurde bereits auf die Fragment-Analyse eingegangen. Die einfache Idee bleibt dieselbe, die Bibliothek wird zu einem Benutzerprogramm erweitert. Die fehlenden Datenflüsse der Bibliothek werden durch die Ergänzung der Bibliothek mit den Platzhaltern `ph_proc` und `ph_var` behoben. Darauf aufbauend kann dann eine Folge von Analysen durchlaufen, siehe Abbildung 5.1. Die Platzhalter `ph_*`, welche die Zwischendarstellung ergänzen, werden für ein besseres Verständnis in Form von Quellcode dargestellt. Die Darstellung ist als Ergänzung zu Kapitel 3.4 und 4.1 zu verstehen und behandelt nur die für C++ hinzugekommenen Eigenschaften.

```
1  // Betrachtet werden zusaetzlich
2  // alle Parameter und die Rueckgabewerte von oeffentlichen Methoden der
       oeffentlichen Klassen;
3  // alle oeffentlichen Variablen der oeffentlichen Klassen;
4  // alle Ausnahmen, die geworfen werden koennen;
5  // Es existiert fuer jeden eindeutigen Typ type_t eine Variable ph_var_t.
6  type_t *ph_var_t;
7
8  void ph_proc () {
9
10     try {
11         // Anstossen der Konstruktoren fuer jede ph_var_t.
12         ph_var_t = new type_t();
13
14         // Aufrufen von allen sichtbaren und oeffentlichen Funktionen der Klasse mit
15         // den dazu passenden Typen ph_var_t.
16         ph_var_t->f(*ph_var_t, *ph_var_t);
17         ph_var_t = ph_var_t->g(ph_var_t);
18
19         // Fuer alle oeffentlichen Variablen m der sichtbaren Klasse type_t
20         // erfolgt eine Zuweisung mit passendem Typ.
21         ph_var_t = ph_var_t->m;
22         ph_var_t->m = ph_var_t; // wenn m nicht const ist und veraendert werden kann
23
24         // fuer statische Variablen ebenfalls
25         ph_var_t = ph_var_t->m;
26         ph_var_t->m = ph_var_t;
27
28         // In der Vererbungshierarchie moegliche und "harmlose" Zuweisungen,
29         // z. B. wenn Klasse Y von Klasse X erbt:
30         ph_var_x = dynamic_cast<X>ph_var_y;
31
32         // Unaere Operatoren wie z. B. + - ! und
```

```
33          // binaere Operatoren wie z. B. && ||
34          // die in der Bibliothek ueberladen sind werden entsprechend
35          // zum Typ benutzt und zugewiesen
36          ph_var_t = !ph_var_t;
37          ph_var_t = ph_var_t && ph_var_t;
38
39          // alle Destruktoren von ph_var_t aufrufen:
40          ph_var_t->~type_t();
41
42      } catch (type_t ph_ex_var_t) {
43          // Alle Ausnahmen werden gefangen und der entsprechenden
44          // ph_var_t zugewiesen
45          ph_var_t = ph_ex_var_t;
46      }
47  }
```

Begonnen wird mit der Annahme, dass eine objektorientierte Bibliothek Schnittstellen in Form von Klassen zur Verfügung stellt. Diese Klassen werden **öffentliche Klassen** genannt. Die Schnittstellen der öffentlichen Klassen werden dem Benutzerprogramm in Form von Signaturen in einer Header-Datei *.h angeboten. Die eigentliche Implementierung wird versteckt. Wie in C wird in C++ ebenfalls die Header-Datei *.h der Bibliothek mittels `#include` eingebunden. Entweder werden die angebotenen Klassen instanziiert und benutzt oder das Benutzerprogramm leitet von den angebotenen Klassen ab, überschreibt Methoden und Variablen, instanziiert die so neu definierten Klassen und benutzt diese. `public` und `protected` Methoden werden dem Benutzerprogramm also direkt (durch Benutzung) oder indirekt (durch Vererbung) angeboten und benutzt. In diesem Zusammenhang werden daher **öffentliche Funktionen und Variablen** einer Klasse als solche definiert, die bei der Deklaration mit `public` oder `protected` gekennzeichnet sind. Äquivalent zum öffentlichen Teil wird auch der Ausdruck **sichtbar** verwendet, das heißt sichtbar vom Benutzerprogramm aus betrachtet.

Betrachtet werden hier zusätzlich zu der bisherigen Fragment-Analyse alle Parameter und die Rückgabewerte von öffentlichen Methoden und alle öffentlichen Variablen. Bibliotheksintern werden alle Ausnahmen herangezogen. Auch der Datentyp ist bei der Betrachtung von Interesse. Für jeden Typ, der so benutzt wird, wird ein **Platzhalter** erzeugt, also eine globale Variable `*ph_var_t` vom Typ `type_t` (siehe Zeile 6 im angegebenen Beispielquellcode). Falls ein Typ öfter vorkommt, wird trotzdem nur ein Platzhalter verwendet.

Für alle erzeugten Platzhalter `ph_var` von sichtbaren Klassen wird der **Konstruktor aufgerufen**. Falls mehrere Konstruktoren existieren, so werden sie alle nacheinander aufgerufen. Für die Realisierung des Entwurfsmusters Singleton [GHJV95] z. B. könnte der Konstruktor `private` sein. Da bereits die Zwischendarstellung vorliegt, kann die Überprüfung der Analysephase[2] umgangen werden und der `private` Konstruktor wird für Analysezwecke aufgerufen, auch wenn die Sprache dies nicht erlaubt. Ähnliches gilt für `private` Destruktoren.

Mit Zeile 16–17 werden alle **sichtbaren Funktionen aufgerufen** und es wird für den fehlenden Datenfluss gesorgt. Dabei wird darauf geachtet, dass immer der

[2] Englisch: frontend

zum Typ entsprechende Platzhalter `ph_var_t` für den jeweiligen Parameter benutzt wird. Statische Methoden (`static`) werden ohne eine Instanz aufgerufen.

Die Zeilen 21–22 und 25–26 sorgen für Datenfluss zwischen Variablen durch **Zuweisungen** in beide Richtungen. Betrachtet werden alle sichtbaren Variablen, auch wenn sie statisch (`static`) sind. Falls sie konstant (`const`) sind und nicht verändert werden können, gilt nur eine Richtung.

Zeile 30 simuliert die **Zuweisung** einer abgeleiteten Klasse `Y` zu der Elternklasse `X`, wobei `X` in der Vererbungshierarchie nicht direkter Vorgänger von `Y` sein muss. Es geht um „legale“, in der Sprache vorgesehene Zuweisungen.

Syntaktischer Zucker, wie **überladene** unäre **Operatoren**

```
1               + - * & ~ ! ++ -- -> ->*
```

und binäre Operatoren [Stroo, Kapitel 11.2]

```
1               +  -  *  /  %  ^  &  |  << >>
2               += -= *= /= %= ^= &= |= <<= >>=
3               <  <= >  >=  ==  !=  &&  ||  []  ()
4               new  new[]  delete  delete[]
```

werden ebenfalls mit dem passenden Typ benutzt und zugewiesen.

Am Ende der Liste der künstlichen Anweisungen wird der **Destruktor** des jeweiligen `ph_var_t` aufgerufen. `private` Destruktoren werden, wie bereits bei den Konstruktoren erwähnt, ebenfalls verwendet.

Datenflüsse, die durch **Ausnahmen** verursacht werden, werden behandelt. Ein `try`-Block umschließt die bisher aufgelisteten Anweisungen. Anschließend folgt eine Menge von `catch`-Blöcken. Für jeden möglichen Datentyp, der in der Bibliothek geworfen werden kann, wird ein `catch`-Block erzeugt. Dieser wird dem Platzhalter `ph_var_t` zum Typ passend zugewiesen (Zeile 42–45). Das Umschließen der erzeugten künstlichen Anweisung ist nötig, da insbesondere Aufrufe von Methoden Ausnahmen erzeugen können. Die Ausnahmen werden nicht behandelt, sondern weitergegeben und es wird für ausreichenden Datenfluss gesorgt. Die Ausnahmen werden also dem Typ entsprechend weiterpropagiert.

In C++ kann es vorkommen, dass ein `catch`-Block nie erreichbar ist. Wenn beispielsweise

```
1               catch (MathError me) { ph_var_me = me; }
2               catch (DivNull dn) { ph_var_dn = dn; }
```

und `DivNull` eine abgeleitete Klasse von `MathError` ist, so kann statisch durch die Vererbungshierarchie bestimmt werden, dass der zweite `catch`-Block nie erreichbar ist [Stroo, Kapitel 11]. Bei der Erzeugung der `catch`-Blöcke wird die Reihenfolge der `catch`-Blöcke gemäß der Vererbungshierarchie von unten nach oben beachtet:

```
1               catch (DivNull dn) { ph_var_dn = dn; }
2               catch (MathError me) { ph_var_me = me; }
```

Datenflüsse durch polymorphe Aufrufe (Dispatching) werden berücksichtigt, denn in Zeile 16–17 erfolgen alle Aufrufe von öffentlichen Methoden und in Zeile 30 wird ein Platzhalter `ph_var_t` von Typ `type_t` einem in der Vererbungshierarchie höher liegenden Platzhalter zugewiesen.

4.2.1 Offene Punkte

In diesem Abschnitt werden Eigenschaften von C++ angesprochen, die bei der Erstellung von `ph_proc` nicht berücksichtigt wurden bzw. offen lassen, wie die Implementierung tatsächlich aussieht.

In C++ gibt es **Mehrfachvererbung** [Stroo, Kapitel 15]. Dies führt dazu, dass das Benutzerprogramm mehrere sichtbare Klassen aus der Bibliothek zu einer Klasse „vereinigen“ kann. Diese Tatsache nicht zu berücksichtigen würde dazu führen, dass die Absicht, fehlende Datenflüsse zu ergänzen, verfehlt wird. Um dieser Tatsache entgegen zu wirken, bleibt nur die Möglichkeit, eine einzige `ph_var` zu erzeugen. Offen bleibt, ob `ph_var` ganz typenlos (`void*`) benutzt wird oder ob eine Klasse `ph_class` erzeugt werden soll, die von allen öffentlichen Klassen der Bibliothek erbt.

Einerseits sollte vom schlimmsten Fall ausgegangen werden, andererseits kommt es der Praxis nicht nahe, eine einzige Variable `ph_var` zu benutzen, die von allen Klassen der Bibliothek erbt. Hier sollte in der Implementierung mittels Parameterübergabe der Analysenanwender entscheiden können, wie konservativ er die Analyse anwenden will. Feingranulare Einstellungen wie „nur Klasse X und Y werden im Benutzerprogramm mehrfach vererbt“ sollen die Präzision der Analyse erhöhen.

Quellcode für `ph_class`:

```
1  // Eine einzige Klasse, die von allen oeffentlichen Klassen erbt:
2  class ph_class : public X, ... , public Y {};
3
4  // Eine Instanz dieser Klasse:
5  ph_class *ph_var;
6
7  // wie vorher ab Zeile 8
8  ...
```

Eine zweite Eigenschaft von C++, die in der Überlegung nicht berücksichtigt wird, ist `friend` [Stroo, Kapitel 11.5]. Das `friend`**-Konzept** ermöglicht einer Klasse `A` die Deklaration von Funktionen oder Klassen, die als „Freund“ betrachtet werden. Diese „Freunde“ können dann auf `private` Variablen oder Funktionen von `A` zugreifen.

Die Möglichkeit, dass eine Bibliothek die Namen der potenziellen „Freunde“ deklariert und sie dann offen lässt, wird für nicht sehr wahrscheinlich gehalten. Jedes Benutzerprogramm hat dann die Möglichkeit, diesen Namen als Funktionsnamen oder Klassennamen zu benutzen, die so definierte Funktion oder Klasse avanciert zum „Freund“ der entsprechenden Klasse in der Bibliothek. Dieses merkwürdig

konstruierte Beispiel verstößt jedoch z. B. gegen Software-Engineering-Methoden wie Verkapselung. Hinzu kommt die Tatsache, dass das `friend`-Konzept nicht sehr häufig benutzt wird. Daher wird davon ausgegangen, dass deklarierte „Freunde“ ebenfalls in der Bibliothek sind. Diese Berücksichtigung erfordert keine Änderung am oben angegebenen Quellcode für C++.

Neben den im Kapitel 3.4 erwähnten Möglichkeiten in C für implizite und explizite Konvertierung kann mittels `reinterpret_cast` [Stroo, Kapitel 6] die gebotene **Typsicherheit** in C++ umgangen werden:

```
1               X x;
2
3               int *xp = reinterpret_cast<int*>(&x);
```

Die Benutzung dieses „gefährlichen“ Merkmals führt dazu, dass zusätzlich Datenflüsse zwischen den unterschiedlichen Datentypen entstehen. Die Konsequenz ist, ähnlich wie bei der Mehrfachvererbung, ein Platzhalter `ph_var` anstatt verschiedener Platzhalter `ph_var_t` mit verschiedenen Typen.

4.2.2 Beispiel

Nach ausführlichen Erklärungen im vorherigen Abschnitt wird ein Beispiel betrachtet, das diese Idee verdeutlicht. Die Bibliothek bietet sichtbar nach außen einem Benutzerprogramm die drei Klassen `Link`, `Station` und `Factory` an.

```
1  /* ----- sichtbar fuer Benutzerprogramm ----- */
2  class Link {
3    public:
4      void transmit (int m);
5  };
6
7  class Station {
8    private:
9      Link *link;
10     int msg_id;
11   public:
12     Station();
13     void sendMessage(int m);
14     void report (Link *l);
15 };
16
17 class Factory {
18   private:
19     bool secure;
20   public :
21     Factory ();
22     Link* getLink ();
23     void makeSecure ();
24 };
```

In der Implementierung kommen zusätzlich noch drei Klassen hinzu (`NormalLink`, `PriorityLink` und `SecureLink`). Diese erben von `Link`. Die Klassen `Error` und `TooManyMessages` werden bei Ausnahmen geworfen.

```
/* ----- nicht sichtbar fuer Benutzerprogramm als Implementierung ----- */
void Link::transmit (int m) { /* ... */}

class NormalLink : public Link { /* ... */ };
class PriorityLink : public Link { /* ... */ };
class SecureLink : public Link { /* ... */ };

class Error { /* ... */ };
class TooManyMessages : public Error { /* ... */};

Station::Station () {
    this->link = new NormalLink ();
    this->msg_id = 0;
}
void Station::sendMessage (int m) {
    this->msg_id++;
    this->link->transmit (m);
    if (msg_id == 10)
        this->link = new PriorityLink ();
    if (msg_id == 100)
        throw new TooManyMessages ();
}
void Station::report (Link *l) {
        l->transmit (this->msg_id);
}
Factory::Factory () {
    secure = false;
}
Link* Factory::getLink () {
    if (this->secure)
        return new SecureLink();
    else
        return new NormalLink();
}
void Factory::makeSecure () {
    this->secure = true;
}
```

Die Ergänzung hat von den sichtbaren Klassen `Station`, `Factory` und `Link` jeweils eine Instanz. Zusätzlich gibt es pro Parametertyp einen Platzhalter, in diesem Fall nur `ph_int`. Die Implementierung könnte in Ausnahmefällen zwei unterschiedliche Typen (`Error` und `TooManyMessages`) werfen. Platzhalter werden hierfür ebenfalls erzeugt. Die künstlichen Anweisungen sind umschlossen von einem `try`-Block, beide Ausnahmentypen werden gefangen und weitergereicht.

```
/* ----- Ergaenzung ----- */
Station *ph_Station;
Factory *ph_Factory;
Link *ph_Link;
int ph_int;
TooManyMessages *ph_TooManyMessages;
Error *ph_Error;
void ph_proc () {
    try {
        ph_Station = new Station ();
        ph_Factory = new Factory ();
        ph_Station->sendMessage (ph_int);
```

```
13          ph_Station->report (ph_Link);
14          ph_Link = ph_Factory->getLink();
15          ph_Factory->makeSecure ();
16          ph_Link->transmit (ph_int);
17      } catch (TooManyMessages *e) {
18          ph_TooManyMessages = e;
19      } catch (Error *e) {
20          ph_Error = e;
21      }
22  }
```

5 Ablauf und Kompaktifizierung

5.1 Ablauf der Analysen von Programmen mit Bibliotheken

In Abbildung 5.1 wird der zeitliche Ablauf der Analysen dargestellt. Die Schritte sind nummeriert, dick umrandete Schritte wurden in dieser Diplomarbeit vollzogen.

Gestartet wird mit der Bibliothek nach der Kompilierung in IML-Format. Die Fragment-Analyse ergänzt die Bibliothek mit fehlenden Datenflüssen. Danach bietet Bauhaus die Möglichkeit, sich für eine der Zeigeranalysen entscheiden zu können, nämlich die von Andersen [And94] [Fro06], Das [Das00] oder Steensgaard [Ste96]. Nach der Zeigeranalyse ist die Aufrufgraphkonstruktion möglich. Anschließend wird der Kontrollfluss intra- und interprozedural aufgebaut. Danach folgt die ISSA-Analyse [SVKW07]. Mittels den vorangegangenen Analysen ist es möglich, die Bibliothek zu kompaktifizieren. Hierbei wurden bisher die Aufrufgraphkonstruktion und die Seiteneffekte berücksichtigt. Die neu entstehende Bibliothek ist somit „kleiner“ oder kompakter. Benötigte Bibliotheken werden anschließend zum Benutzerprogramm hinzugelinkt. Am Ende der Berechnungskette steht das Benutzerprogramm bereit für Analysen, die in Bauhaus angeboten werden.

Wie bei der Kompaktifizierung vorgegangen werden kann, zeigen Rountev und Chandra im folgenden Abschnitt.

5.2 Verwandte Arbeit

Rountev und Chandra haben sich in [RC00] ebenfalls mit Bibliotheken beschäftigt, ausführlicher wird die Arbeit im Technical Report [RR00] beschrieben. Rountev et al. beschäftigen sich speziell mit einer Voranalyse, um die Zeigeranalyse von Andersen zu beschleunigen. Die Idee dabei ist, äquivalente Mengen von Variablen durch eine Repräsentation zu substituieren. Für die Andersen-Analyse bleibt die Substitution präzise. Im Schnitt wird dadurch eine Laufzeiteinsparung von 53 % erreicht, die Größe der Binärdatei einer Bibliothek sinkt im Durchschnitt um 36 %. Diese Vorgehensweise wird direkt an einem Beispiel aus [RC00] erklärt.

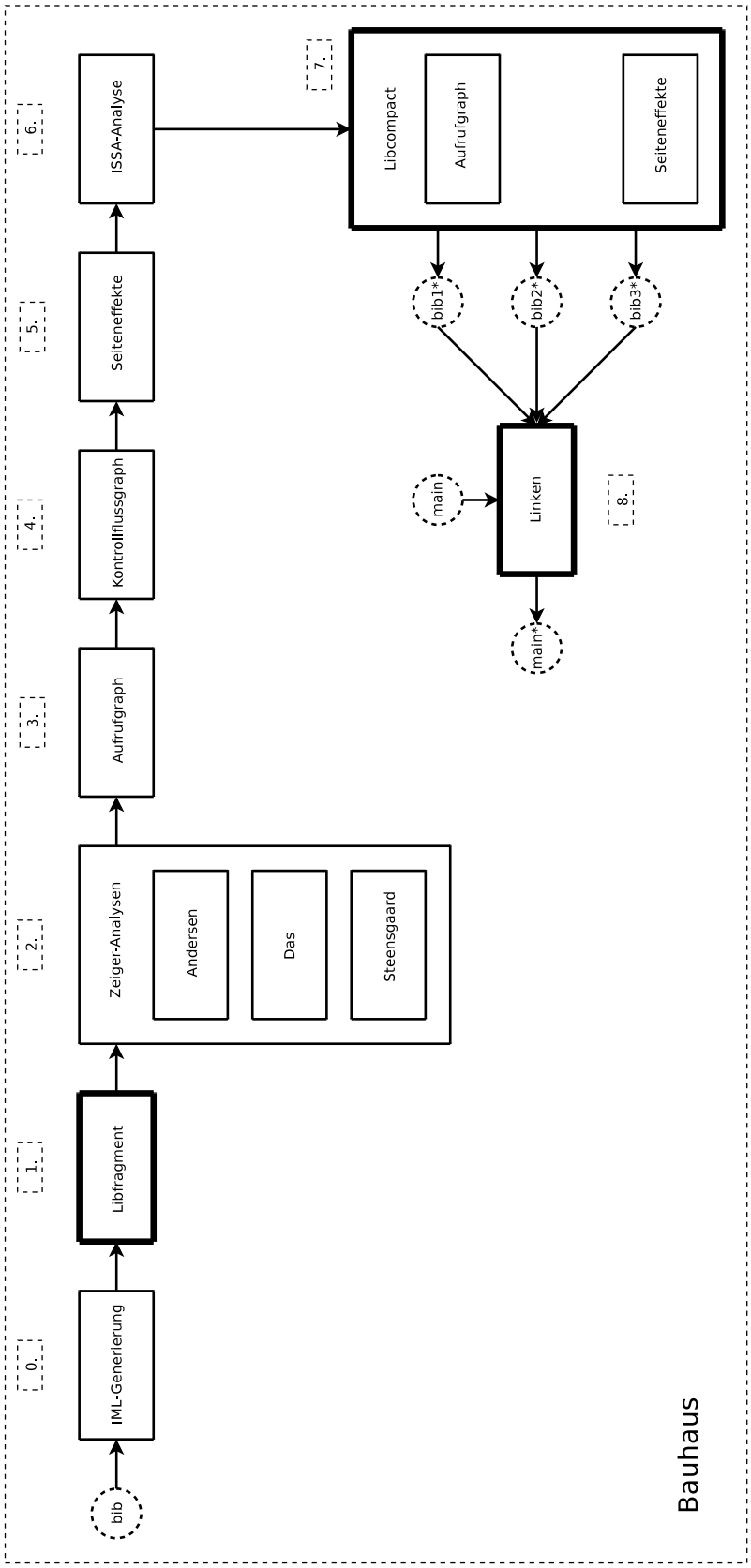

Abbildung 5.1: Ablauf der Analysen

5.2.1 Beispiel

Sei folgendes Programm gegeben, es existiert kein Kontrollfluss und die Reihenfolge der Anweisungen spielt keine Rolle, da die Andersen-Analyse diese ebenfalls nicht beachtet:

```
x = &a;
x = *p;
y = x;
*p = y;
s = *p;
t = *p;
```

```
a = *y;
q = &s;
z = *q;
m = i;
m = j;
// sichtbar p und m
```

Es wird nun ein Graph wie in Abbildung 5.2 für die Untermengenbeziehungen der points-to-Menge[1] $Pt(v)$ von Variable `v` aufgebaut. Jede Kante (n_1, n_2) sagt aus, dass die Zeigermenge von n_1 in n_2 enthalten ist: $Pt(n_1) \subseteq Pt(n_2)$. Für Zuweisungen werden Kanten gezogen. Dabei gelten die Regeln:

1. Für jede Zuweisung `p = &x` gibt es zwei Kanten $(\&x, p)$ und $(x, *p)$.
2. Für jede Zuweisung `p = q` gibt es zwei Kanten (q, p) und $(*q, *p)$.

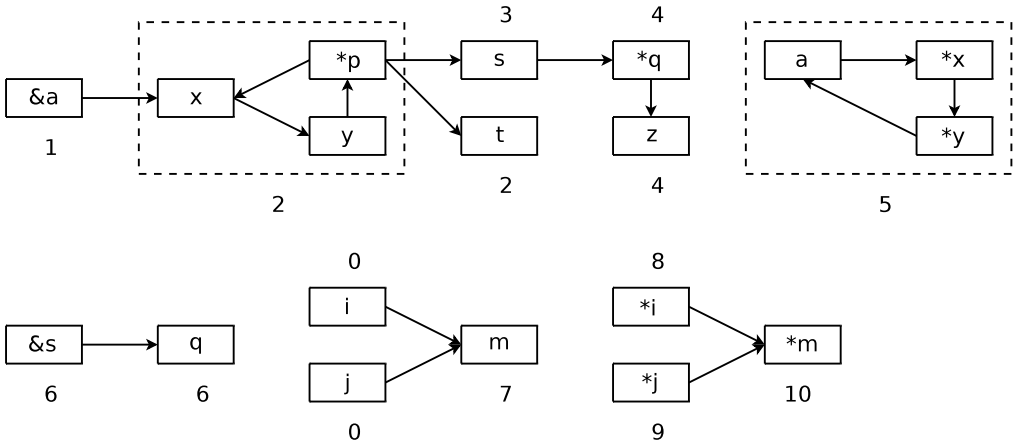

Abbildung 5.2: Graph als Repräsentation für die points-to-Mengen

Starke Zusammenhangskomponenten und einige Details, auf die hier nicht eingegangen wird[2], werden berechnet. Es entstehen Äquivalenzklassen, die in Abbildung 5.2 nummeriert sind. Insbesondere an der Äquivalenzklasse Nr. 2 ist ersichtlich, dass `x` und `y` durch eine neue, bis dahin noch nicht benutzte Variable `r2`, ersetzt werden können.

[1] Der englische Name wird beibehalten. Die points-to-Menge $Pt(v)$ für eine Variable `v` gibt an, auf welche Objekte `v` potenziell zeigen kann.

[2] Beispielsweise die Behandlung der sichtbaren Variablen `p`, da zur Analysezeit die points-to-Menge von `p` noch nicht vollständig ist etc.

Funktion für die Substitution

Aus den Äquivalenzklassen lässt sich die Funktion für die Substitution ablesen:

$\sigma(x) = r2$
$\sigma(y) = r2$
$\sigma(p) = p$
$\sigma(s) = r3$
$\sigma(t) = r2$
$\sigma(z) = r4$
$\sigma(a) = r5$
$\sigma(q) = r6$
$\sigma(i) = r0$
$\sigma(j) = r0$
$\sigma(m) = m$

Nach der Substitution

Das Beispielprogramm sieht nach der Substitution wie folgt aus:

```
r2 = &r5;
r2 = *p;
r2 = r2;
*p = r2;
r3 = *p;
r2 = *p;
```

```
r5 = *r2;
r6 = &r3;
r4 = *r6;
m = r0;
m = r0;
```

Endergebnis

Doppelte Zuweisungen, redundante Zuweisungen und Zuweisungen von Nicht-Zeigervariablen können entfernt werden:

```
r2 = &r5;
r2 = *p;
*p = r2;
r3 = *p;
```

```
r5 = *r2;
r6 = &r3;
r4 = r3;
```

Wie deutlich am Endergebnis zu sehen ist, wurde die Anzahl der Zuweisungen von elf auf sieben reduziert.

Ähnliche Strategien können verfolgt werden, um durch eine Kompaktifizierung redundante oder nicht relevante Anweisungen zu löschen.

5.3 Generelle Strategie und Weiterverwendung

Wie eben in 5.2 gezeigt, ist nun die erste Strategie Anweisungen in eine äquivalente Form zu bringen und dadurch eine Reduzierung der Anweisungen und somit eine Reduzierung der Laufzeit zu erreichen. Die zweite Möglichkeit ist das Löschen von allen Anweisungen, falls sie nicht mehr benötigt werden und die darauf folgende Erstellung von künstlichen Anweisungen. Diese sollen später nach dem Linken mit dem Benutzerprogramm das Verhalten des Resultats der Voranalyse

simulieren. Für dieses Verhalten wird IML als Speicherformat benötigt. Abschnitt 5.3.2 realisiert die zweite Strategie für die Aufrufgraphkonstruktion mittels der Rollenpropagierung.

5.3.1 IML als Speicherformat

Für die Speicherung und Weiterverwendung der Ergebnisse wurde IML als Speicherformat gewählt. IML bringt einige Vorteile mit sich, denn die Infrastruktur in Bauhaus ist bereits vorhanden. Das Laden, Speichern und Modifizieren von Knoten wird von der Infrastruktur angeboten. Hinzu kommen viele Analysen, die IML als Ein- und Ausgabeformat besitzen. Darüber hinaus können Benutzerprogramme nach der IML-Generierung mit der analysierten Bibliothek gelinkt werden.

5.3.2 Aufrufgraph

In 3.5 wurde bereits erklärt, wie die Rollenpropagierung für den Aufrufgraphen realisiert wurde. Die Abbildung 5.3 stellt einen Ausschnitt des Rollenpropagierungsgraphen für ncurses dar. Initialrollen werden (überwiegend an runden Knoten) erzeugt und nach oben propagiert.

Die Information, die mittels der Rollenpropagierung an einem formalen Parameter `fp` einer Schnittstellenfunktion `f` ankommt, ist, dass über diesen Parameter mindestens ein Aufruf erfolgen wird. Der Aufruf oder die Aufrufe können mittelbar in `f` erfolgen. Sie können aber auch von `f` weitergereicht werden, die Aufrufe erfolgen dann erst später. Um diese durch die Rollenpropagierung gewonnene Information in IML speichern zu können, wird in `f` ein indirekter Aufruf über Parameter `fp` künstlich zu `f` hinzugefügt.

Alle Anweisungen in `f`, speziell für die Aufrufgraphkonstruktion, werden nicht mehr benötigt und werden entfernt. Speziell für Rückrufe und die Ziele, die erst im Benutzerprogramm festgelegt werden, ist diese Vorgehensweise interessant. Nach der Kompaktifizierung und nach dem Linken mit dem Benutzerprogramm laufen Analysen „ins Leere“, denn die Rümpfe von allen Bibliotheksfunktionen enthalten keine Anweisungen mehr. Sie enthalten lediglich Aufrufe von Parametern als Ergebnis der Rollenpropagierung. Hier ein Beispiel:

Vorher

```
1 void f (int (*fp) (), int length, double depth) {
2     // viele Anweisungen und Aufrufe
3 }
```

Nachher

```
1 void f (int (*fp) (), int length, double depth) {
2     (*fp)();
3 }
```

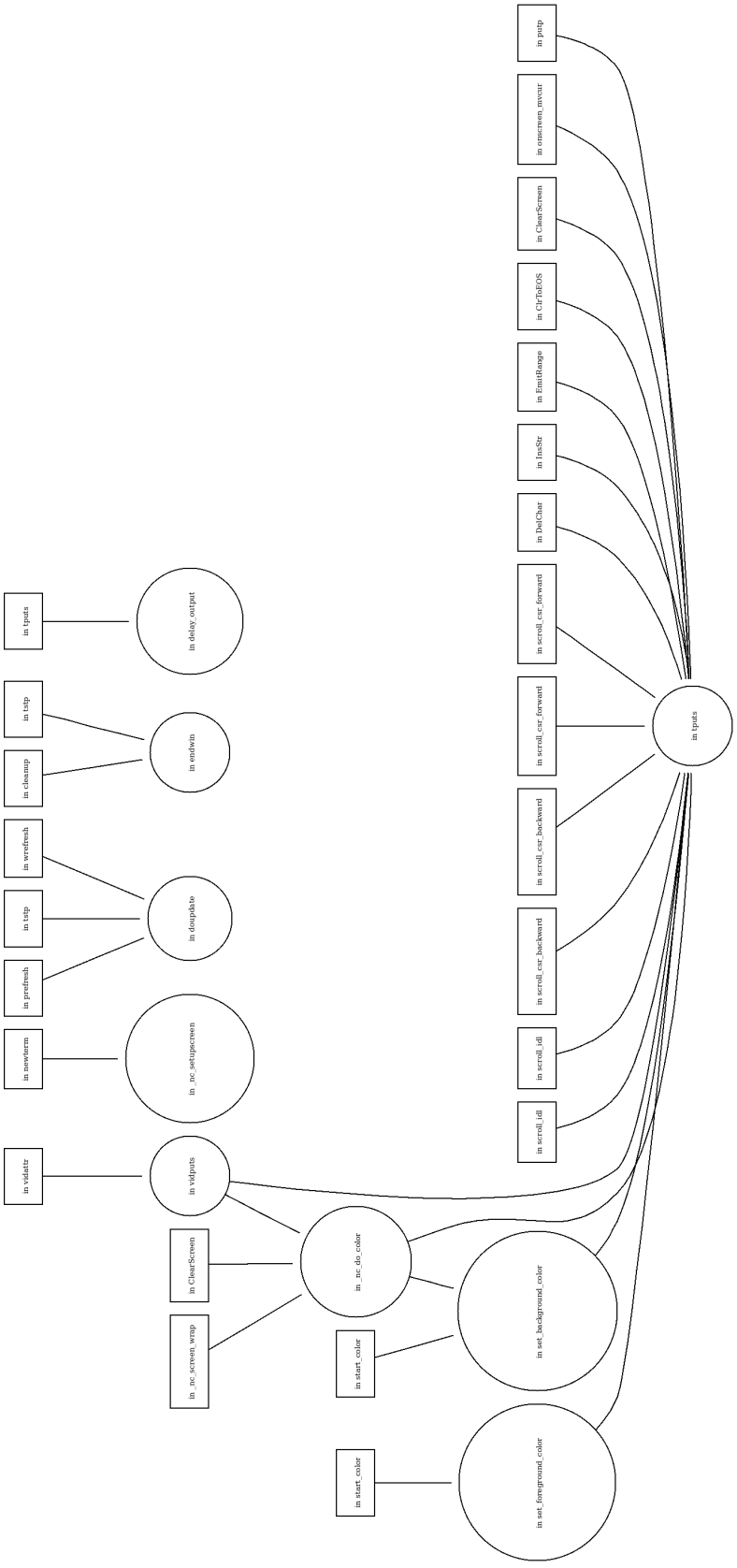

Abbildung 5.3: Ausschnitt des Rollenpropagierungsgraphen für ncurses

Bei der Vorgehensweise ist zu beachten, dass nicht mehr bestimmt werden kann, in welchem Kontext der ursprüngliche Aufruf von `fp` erfolgt ist. Die Übergabe eines Parameters an `fp` bleibt ebenfalls unbestimmt. Erhalten bleibt die Information: Es wird auf jeden Fall einen Aufruf auf diesem Parameter erfolgen. Als Rückruffunktion sind die Ziele nach der Zeigeranalyse im Benutzerprogramm zu finden.

5.3.3 Seiteneffektberechnung

Das Ergebnis der Seiteneffektberechnung liegt bereits in Form von drei Mengen may_def, must_def und may_use vor. Für die Implementierung sei hier auf 6.2.2 verwiesen. Zu beachten ist hierbei, dass in den drei genannten Mengen `ph_var` als Platzhalter noch enthalten ist. Nach dem Linken des Benutzerprogramms folgt wieder die Kette der Analysen aus der Abbildung 5.1. Bei der zweiten Ausführung der Seiteneffektberechnung sollte noch folgende Anpassung nötig sein: Anstelle der Variablen `ph_var` kommt nun der aktuelle Parameter des Aufrufs der Funktion, dieser ersetzt den Platzhalter `ph_var`.

6 Implementierung

In diesem Kapitel wird im Detail auf die Implementierung eingegangen. Das Wesentliche wird als Pseudocode wiedergegeben. Der „Bauplan“ für die Fragment-Analyse wurde bereits in 4.1 angegeben. Für die Benutzung der Rollenpropagierung in Zusammenhang mit der Aufrufgraphkonstruktion sei auf 3.5.1 und 5.3.2 verwiesen.

6.1 Fragment-Analyse

Das Hauptprogramm `Libfragment` in Listing 6.1 liest die Konfigurationsdatei ein und stellt fest, welche Funktionen sichtbar sind. Die benötigten verschiedenen Typen werden berechnet. Für jeden Typ wird eine Variable `ph_var` erzeugt. Daraufhin werden `ph_wrapper` und `ph_proc` erstellt.

```
1  procedure Libfragment is
2     Type_List_PH_Var : List;
3  begin
4     Read_Config;
5
6     Type_List_PH_Var := Calculate_Different_Types_Of_PH_Var;
7
8     Create_PH_Var_Diff_Types (Type_List_PH_Var);
9
10    Create_PH_Wrappers;
11
12    Create_PH_Proc;
13 end;
```

Listing 6.1: Hauptprogramm Libfragment

Für die Berechnung der verschiedenen Typen (Listing 6.3) werden alle relevanten Signaturen betrachtet: Signaturen von sichtbaren Funktionen, Funktionen, deren Adressen genommen wurden, Signaturen von indirekten Aufrufen und Typen von sichtbaren Variablen. Für jeden so berechneten Typ wird eine `ph_var` angelegt (Listing 6.4). Ist die Analyse typinsensitiv, so wird nur eine `ph_var` erzeugt.

```
1  typedef int wochentag;
2  wochentag t;
3  int x;
```

Listing 6.2: Zwei äquivalente Typen

`In_Type_List` gibt einen Wahrheitswert zurück, je nachdem, ob ein Typ in der Resultatsliste schon enthalten ist oder nicht. Hier sei angemerkt, dass das Vergleichen zweier Typen mit der IML-Klasse `T_Node` und aller abgeleiteten Klassen nicht trivial ist. So sind beispielsweise in Listing 6.2 die Variablen `x` und `t` typenäquivalent, sie werden aber in IML unterschiedlich dargestellt. Weitere IML-Klassen, die beim Vergleich betrachtet werden, sind `T_Volatile_Qualifier`, `T_Const_Qualifier` oder `T_Routine` etc., um hier nur einige zu nennen.

```
1  function Calculate_Different_Types_Of_PH_Var return List is
2     Result_List : List;
3  begin
4     if Is_Type_Insensitive then
5        Add_Type_Void (Result_List);
6        return Result_List;
7     else
8
9        -- BEGIN [1]
10       for I in Visible_Routines'Range loop
11          for J in Visible_Routines (I).Parameters'Range loop
12             if not In_Type_List (Result_List, Type_Of (Visible_Routines
                    (I).Parameters (J))) then
13                Add_Type (Type_Of (Visible_Routines (I).Parameters (J)), Result_List);
14             end if;
15          end loop;
16
17          if not In_Type_List (Result_List, Type_Of (Visible_Routines (I))) then
18             Add_Type (Type_Of (Visible_Routines (I)), Result_List);
19          end if;
20
21          if not In_Type_List (Result_List, Type_Of (Visible_Routines
                 (I).Return_Type)) then
22             Add_Type (Type_Of (Visible_Routines (I).Return_Type), Result_List);
23          end if;
24       end loop;
25       -- END [1]
26
27       -- Aehnlich wie [1] fuer Funktionen, deren Adressen genommen wurden
28       -- Aehnlich wie [1] fuer indirekte Aufrufe
29       -- Aehnlich wie [1] fuer sichtbare Variablen
30
31       return Result_List;
32    end if;
33 end;
```

Listing 6.3: Berechnung der verschiedenen Typen für `ph_var`

```
1  procedure Create_PH_Var_Diff_Types (List_Var_Diff_Types : List) is
2  begin
3     for I in List_Var_Diff_Types'Range loop
4        Create_PH_Var;
5     end loop;
6  end;
```

Listing 6.4: Anlegen von `ph_var`

Handelt es sich um die typinsensitive Fragment-Analyse, so existiert nur eine Prozedur ph_wrapper. *m* ist die maximale Anzahl der Parameter aller indirekten Aufrufe. ph_wrapper hat eine Signatur mit *m* formalen Parametern. ph_wrapper wird zur Callee-Menge eines indirekten Aufrufs hinzugefügt.

Bei der typsensitiven Fragment-Analyse hat jeder indirekte Aufruf eine ph_wrapper. Die Anzahl der Parameter von ph_wrapper hängt vom jeweiligen indirekten Aufruf ab.

Der Rumpf von ph_wrapper enthält Zuweisungen, um den Datenfluss zu ph_var zu ergänzen. Bei der typsensitive Fragment-Analyse wird die zum Typ passende Variable ph_var gesucht, bei der typinsensitiven Fragment-Analyse existiert nur eine ph_var.

```
1  procedure Create_PH_Wrappers is
2  begin
3     if Is_Type_Insensitive then
4        Create_PH_Wrapper;
5     else
6        for each indirect call loop
7           Create_PH_Wrapper;
8        end loop;
9     end if;
10 end;
```

Listing 6.5: Anlegen von ph_wrapper

Entsprechend wird in Listing 6.6 das Anlegen von ph_proc angegeben:

- Zuweisungen von sichtbaren Variablen
- Das Nehmen der Adressen von sichtbaren Funktionen
- Das Nehmen der Adresse von ph_proc sowie das „Austricksen“ der Zeigeranalysen
- Direkte Aufrufe zu sichtbaren Funktionen
- Direkte Aufrufe zu Funktionen, deren Adressen genommen wurden

Alle Anweisungen benutzen ph_var entsprechend. Bei der typsensitive Fragment-Analyse wird die zum Typ passende ph_var gesucht und verwendet.

6.1.1 Konfigurationsdatei

Variablen und Funktionen als Schnittstellen von Bibliotheken werden üblicherweise in Form einer *.h Datei angeboten. Diese Information ist informell und für jede Bibliothek verschieden. Sie geht außerdem nach der Generierung einer IML-Datei für eine Bibliothek verloren. Der Analysenanwender der Fragment-Analyse muss daher über eine Konfigurationsdatei festlegen, welche Variablen und Funktionen sichtbar sind. In der Konfigurationsdatei sind hierfür folgende Einstellungen vorgesehen:

```
1  procedure Create_PH_Proc is
2
3  begin
4       Create_Assigns_Visible_Vars;
5
6       Create_Assigns_Visible_Funcs;
7
8       Create_Assigns_PH_Vars;
9
10      Create_Call_Visible_Functions;
11
12      Create_Call_Functions_Address_Taken;
13 end;
```

Listing 6.6: Anlegen von `ph_proc` mit Zuweisungen und direkten Aufrufen

```
[visible variables]

[visible functions]
[visible files]
[visible prefixes]
```

Um die Namen der Variablen und Funktionen zu finden, muss der Analysenanwender die *.h Dateien betrachten und sie aus dem Kontext der jeweiligen Datei filtern. Hierzu können Linux-Werkzeuge wie awk und grep benutzt werden. Als Beispiel hier ein Ausschnitt aus xvid.h:

```
1 extern int xvid_global(void *handle, int opt, void *param1, void *param2);
2 extern int xvid_decore(void *handle, int opt, void *param1, void *param2);
3 extern int xvid_encore(void *handle, int opt, void *param1, void *param2);
```

Ein Linux-Befehl bringt das entsprechende Resultat. Für mehr Details wird auf die Referenzseite des jeweiligen Werkzeugs verwiesen.

```
grep xvid xvid.h | awk '{print $3}' | awk -F"(" '{print $1}'

xvid_decore
xvid_encore
xvid_global
```

Um den Aufwand zu minimieren, kann der Analysenanwender auch direkt die Datei angeben, in der sich sichtbare Variablen und Funktionen befinden. Alle Routine- und O_Variable-Objekte, die über `SLocs.Get_Filename` der Konfiguration entsprechen, werden als sichtbar betrachtet. Aufgrund des fehlenden Namensraums[1] in C haben viele Bibliotheken einen künstlichen Namensraum durch einheitliche Schreibweise geschaffen, wie beispielsweise `gtk_*` für sichtbare Funktionen und Variablen. Ein solches Präfix kann auch angegeben werden, um sichtbare Variablen und Funktionen festzulegen.

[1] Englisch: namespace

Der in Zeile 6 von 3.4.2 angegebene Quellcode (Listing 3.1) und die in 3.4.4 besprochenen Nachteile der Fragment-Analyse bezüglich des Nehmens der Adresse aller sichtbaren Funktionen, welches signifikant zur Verschlechterung der Konstruktion des Aufrufgraphen führt, wird mit einer möglichst feingranularen Konfiguration und dem Wissen des Analysenanwenders behoben. Diese Einstellungen ähneln der Angabe von sichtbaren Funktionen:

```
[visible functions address taken]
[visible files address taken]
[visible prefixes address taken]
```

Wenn der Analysenanwender keinerlei Informationen hat oder keinen Aufwand betreiben will, so sollten die Einstellungen gleich der Angabe der sichtbaren Funktionen sein. Zu beachten ist, dass dann Folgendes angenommen wird: Adressen aller sichtbaren Funktionen werden genommen und in die Bibliothek propagiert. Dies ist weder sinnvoll noch realistisch.

6.1.2 Weitere Implementierungsdetails

Ellipse

In C/C++ gibt es die Möglichkeit, die Anzahl der formalen Parameter bei einer Signatur mittels „...“ offen zu lassen. Als Beispiel ist klassisch `printf` zu nennen:

```
1   int printf ( const char * format, ... );
```

Aus dieser Signatur lässt sich auch der Typ nicht feststellen. Es stellt sich bei der Erzeugung von künstlichen Aufrufen die Frage, welche `ph_var` bei der typsensitive Analyse genommen werden soll und wie oft `ph_var` benutzt werden soll.

Eine empirische Untersuchung zeigt, dass dieses Merkmal selten verwendet wird. Außerdem handelt es sich in der Regel um eine „Datenflusssenke“. Dies bedeutet, dass der Datenfluss zwar hineingeht, um beispielsweise bei `printf` eine Ausgabe zu erzeugen, aber dass es selten Auswirkungen auf dem Parameter an sich hat. Daher wurde für die typsensitive Analyse entschieden, immer die Variable `ph_var` vom Typ `void` zu benutzen. Um zu zeigen, dass bei diesem Merkmal Datenflüsse existieren, wurde `ph_var` repräsentativ einmal eingesetzt.

6.1.3 Datenstrukturen und Komplexität

In der Fragment-Analyse wurde die IML ergänzt. In ihr gibt es im Wesentlichen keine schwierigen Algorithmen. Daher beruhen Datenstrukturen überwiegend auf dem generischen Listenpaket, welches bereits in Bauhaus existiert.

Sei n die Anzahl der Funktionen und indirekten Aufrufe, die betrachtet werden müssen. Sei m_i die Anzahl der Parameter der Funktion `f_i` und $m =$

$max(m_1, \ldots, m_n)$. Sei n_c die Anzahl der Funktionen, die in der Konfigurationsdatei angegeben sind und n_{max} die Gesamtanzahl aller Funktionen in der IML-Datei.

Dann hat `Read_Config` die Komplexität $\mathcal{O}(n_{max} \cdot n_c)$, da jeder Name in der Konfigurationsdatei mit allen Funktionen verglichen werden muss, um die sichtbaren Funktionen zu identifizieren.

`Calculate_Different_Types_Of_PH_Var` ist in $\mathcal{O}((n \cdot m)^2)$, da alle Parameter miteinander verglichen werden müssen, um die unterschiedlichen Typen zu finden.

`Create_PH_Var_Diff_Types` ist in $\mathcal{O}(n)$. Für jedes Element der erzeugten Typenliste wird jeweils ein Objekt angelegt.

`Create_PH_Wrappers` ist in $\mathcal{O}(n \cdot m)$. Für jeden indirekten Aufruf wird eine `ph_wrapper` erzeugt.

`Create_PH_Proc` ist in $\mathcal{O}(n \cdot m)$. Für jede Funktion erfolgt ein Aufruf. Zuweisungen sind ebenfalls in linearer Zeit.

Insgesamt ergibt sich für die typsensitive Fragment-Analyse eine Komplexität von $\mathcal{O}(n_{max} \cdot n_c + (n \cdot m)^2 + n + 2 \cdot (n \cdot m)) \in \mathcal{O}(n_{max} \cdot n_c + (n \cdot m)^2)$. Für die typinsensitive Fragment-Analyse fällt die Berechnung der verschiedenen Typen weg: $\mathcal{O}(n_{max} \cdot n_c + n + 2 \cdot (n \cdot m)) \in \mathcal{O}(n_{max} \cdot n_c + 3 \cdot (n \cdot m))$ für $m \geq 1$

6.2 Kompaktifizierung und Weiterverwendung

6.2.1 Aufrufgraph

Die Benutzung der Rollenpropagierung für die Aufrufgraphkonstruktion sowie die anschließende Kompaktifizierung wurden bereits in 3.5.1 und 5.3.2 beschrieben. Es existiert ein Rollentyp für indirekte Aufrufe und das generische Paket der Rollenpropagierung wird instanziiert:

```
1              -- instantiation of general role graph
2              type Role_Types is (Caller_Role);
3
4              package CG_Roles is new Role_Graphs (Role_Types);
```

An jeder Anweisung, Zuweisung und `return`-Anweisung von Typ indirekter Aufruf `Indirect_Calls.Indirect_Call_Class` wird eine Rolle erzeugt. Das generische Paket für Rollenpropagierung berechnet anhand der Initialrollen den Rollenpropagierungsgraphen (Abbildung 5.3). Dieser Graph wird traversiert und jeder Knoten wird einmal besucht. Jedoch kommen nur Knoten in Betracht, die sichtbar vom Benutzerprogramm sind. Für jede Rolle und die dazu gehörigen Funktion `f` und Parameter `p_i` wird ein indirekter Aufruf im Rumpf der Funktion `f` erzeugt, auch wenn das Paar (`f`, `p_i`) mehrfach auftreten kann. Zuvor wird der Rumpf der Funktion `f` gelöscht, `f` enthält vor der künstlichen Erzeugung von indirekten Aufrufen keine Anweisungen mehr.

6.2.2 Linker

Das Ergebnis der Seiteneffektberechnung liegt bereits vor und muss nicht weiter angepasst werden, sondern lediglich mit dem Benutzerprogramm verbunden werden.

Der Linker wurde bisher benutzt, um die Bibliothek als IML mit dem Benutzerprogramm zu verbinden. Das Verbinden geschah bisher direkt nach der jeweiligen Generierung der IML-Datei von Benutzerprogramm und Bibliothek. Nun wurde die Bibliothek mit weiteren Ergebnissen einer Reihe von Analysen angereichert. Hierfür muss der Linker imllink3 angepasst werden, um mit der Anreicherung zurechtzukommen.

Es sind im Wesentlichen folgende Knotentypen und alle damit verbundenen Knotentypen, die als interne Datenstruktur für die Aufgezählten benutzt werden:

- Locator_Table_Rep
- Definition_Table
- Phi_Table
- Simple_DF_Pattern

Die zeitaufwendige Arbeit beschränkt sich auf das Laden und Speichern der aufgezählten Datenstrukturen, sowie das Anpassen für Zeigerziele, da die neu erzeugte IML-Datei aus zwei oder mehr IML-Dateien zusammengefügt wird und die alten Zeigerziele nicht mehr stimmen.

7 Test, Messungen und Evaluation

Laufzeitumgebung

Alle Tests wurden auf einem Rechner mit der Hardwareausstattung: 4 Intel® Xeon™, CPU-Leistung 3.06 GHz, 4 GB Arbeitsspeicher durchgeführt. Das Betriebssystem war Debian Linux mit dem Kernel 2.16.18. Da gerade bei Laufzeiten Schwankungen von etwa 10 % festzustellen sind, wurden — so weit möglich[1] — alle Tests drei Mal durchgeführt und das arithmetische Mittel gebildet.

7.1 Betrachtete Bibliotheken

Um die Implementierung auf ihre Tauglichkeit zu prüfen, wurden zu Testzwecken fünf Bibliotheken ausgewählt. Sie wurden bereits am Anfang der Diplomarbeit ausgewählt und sollten zeigen, dass die Implementierung durchaus praxistauglich ist. Es existieren für Bauhaus eine Reihe von unterschiedlichen Programmen zu Testzwecken. Sie unterscheiden sich ihrer Größe und in ihrem Verhalten. Leider konnte nicht auf sie zurückgegriffen werden, da es sich nicht um Bibliotheken handelt[2]. In diesem Abschnitt wird darauf eingegangen, welche Kriterien bei der Auswahl eine Rolle spielten. Es wird kurz angegeben, was die Bibliotheken leisten. Einige Metriken werden erwähnt, um eine Vorstellung zu haben, um welche Größenordnungen es sich handelt.

Auswahl

Das erste Kriterium lautet kostenloser Zugang zum Quellcode. Deswegen wurde auf Bibliotheken aus der Open-Source-Gemeinde zurückgegriffen. Alle fünf ausgewählte Bibliotheken besitzen GPL[3] oder GPL-ähnliche Lizenzen. Für genauere lizenzrechtliche Fragen wird direkt auf die Webseite der jeweiligen Bibliothek verwiesen.

Nach dem zweiten Kriterium sollen die Bibliotheken ein möglichst breites Spektrum an Funktionalität bzw. Einsatzbereichen abdecken. Die Auswahl deckt von

[1] Bei zu langen Laufzeiten wie bei der Andersen-Analyse wurde aus Zeitgründen auf die dreifache Messung verzichtet.

[2] Die Suche gestaltete sich als schwierig und zeitaufwendig, da der Kompiliervorgang für jede Bibliothek eine andere Anforderung an die Laufzeitumgebung stellte und es nicht immer möglich war (sofort) die Zwischendarstellung IML aus dem Quellcode zu erzeugen.

[3] http://www.gnu.org/licenses/gpl-2.0.html Zuletzt besucht am 17. Juni 2008.

Video-Kompression über Datenbanksystem bis hin zum Lösen von linearen Gleichungssystemen alles ab.

Bezüglich der Größe wurde ein Mittelmaß gewählt. Zu große Bibliotheken führen zu langen Wartezeiten bei der Implementierung bzw. bei (Regressions-)Tests und wirken kontraproduktiv. Genauere Zeigeranalysen wie die von Andersen [And94] sowie die Berechnung der ISSA-Form [SVKW07] können beispielsweise mehrere Stunden bis zu einigen Tagen laufen. Zu kleine Bibliotheken sind nicht aussagekräftig genug und decken nicht alle Sonderfälle ab.

Da die Implementierung der Fragment-Analyse nur mit C Quellcode umgehen kann, sind alle fünf ausgewählten Bibliotheken in C.

Metriken

Tabelle 7.1 stellt die ausgewählten Bibliotheken dar. Spalte zwei präsentiert die Anzahl der Quellcodezeilen. Spalte drei gibt die Anzahl der sichtbaren Funktionen an, welche die Bibliothek als Programmierschnittstelle zur Verfügung stellt. Spalte vier veranschaulicht die Anzahl der global sichtbaren Variablen, die ebenfalls als Programmierschnittstelle zur Verfügung stehen.

Name	SLOC in k	Sichtb. Fkt	Sichtb. Var
glpk	41	137	0
jpeg	20	48	0
ncurses	27	154	30
sqlite	54	108	2
xvid	25	3	0

Tabelle 7.1: Die fünf ausgewählten Bibliotheken

Tabelle 7.2 gibt die Größe der IML-Dateien wieder, die direkt nach dem Kompilieren zu *.so Dateien entstanden sind. Es gibt offensichtlich keinen direkten Zusammenhang zwischen Anzahl der Quellcodezeilen (SLOC) und Größe der IML-Dateien. Da die Laufzeit in der Regel direkt von der Größe der IML-Datei abhängt, ist die Angabe in 7.2 wichtiger als SLOC.

Name	Größe in MB
glpk	14,5
jpeg	5,0
ncurses	5,3
sqlite	9,8
xvid	18,7

Tabelle 7.2: Die Größe der IML-Dateien

Die Abkürzung glpk[4] steht für GNU Linear Programming Kit. Sie bietet in Form von Programmierschnittstellen unter anderem die Möglichkeit, die Lineare Optimierung und die Gemischt-Ganzzahlige Programmierung[5] durchzuführen.

Die Kompressions- und Dekompressionsverfahren für Bilder werden von der Bibliothek jpeg[6] unterstützt. Sie findet weite Verbreitung in vielen Anwendungen, die das jpeg-Format unterstützen.

Reine Konsolen-Anwendungen wie das Dateiverwaltungsprogramm Midnight Commander[7] (Abbildung 7.1) benutzen ncurses[8], um Text oder grafische Elemente in Form von „Fenstern“ darzustellen.

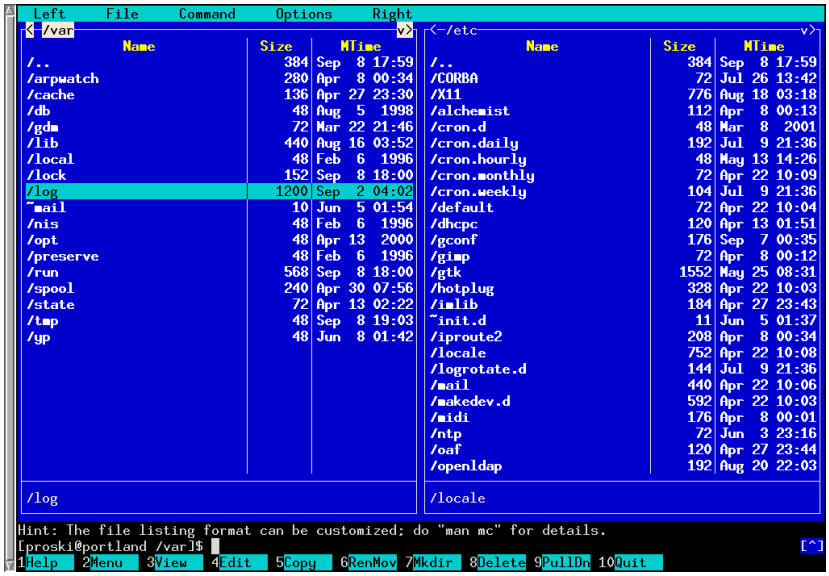

Abbildung 7.1: Screenshot von Midnight Commander mit ncurses

Das relationale Datenbanksystem sqlite[9] zeichnet sich durch seine Leichtgewichtigkeit aus und erfüllt die ACID-Eigenschaften[10]. sqlite wird z. B. im aktuellen Browser Firefox 3[11] verwendet.

xvid[12] kann Video-Bilder komprimieren und dekomprimieren und unterstützt das Format MPEG-4[13].

Ähnlich wie gewöhnliche grafische Bibliotheken wie Qt oder gtk interagiert ncurses sehr stark mit dem Benutzerprogramm, während xvid lediglich komprimiert und

[4]http://www.gnu.org/software/glpk/ Zuletzt besucht am 17. Juni 2008.
[5]Englisch: mixed integer programming
[6]http://www.ijg.org/ Zuletzt besucht am 17. Juni 2008.
[7]http://www.ibiblio.org/mc/ Zuletzt besucht am 17. Juni 2008.
[8]http://www.gnu.org/software/ncurses/ncurses.html Zuletzt besucht am 17. Juni 2008.
[9]http://www.sqlite.org/ Zuletzt besucht am 17. Juni 2008.
[10]Englische Abkürzung und steht für Atomarität, Konsistenz, Isoliertheit und Dauerhaftigkeit.
[11]http://www.mozilla.org/ Zuletzt besucht am 17. Juni 2008.
[12]http://www.xvid.org/
[13]http://www.chiariglione.org/mpeg/ Zuletzt besucht am 17. Juni 2008.

dekomprimiert. Diese zwei unterschiedlichen Verhalten erklären die Anzahl der sichtbaren Funktionen, die als Programmierschnittstelle angeboten werden.

7.1.1 Laufzeiten

Die vorgestellten Bibliotheken wurden einer Reihe von Tests unterzogen und auf die Tauglichkeit in der Praxis geprüft. Die Tabellen 7.3, 7.4, und Abbildung 7.2 stellen die Laufzeiten der typsensitiven und typinsensitiven Fragment-Analyse in Sekunden dar. Das Ausführen der Analyse auf allen Bibliotheken blieb unter fünf Sekunden. Selbst die typsensitive Fragment-Analyse angewendet auf die hier nicht aufgeführte Bibliothek gtk brauchte nur 17 Sekunden. Die Laufzeit der Fragment-Analyse machte sich also kaum bemerkbar. Die typsensitive Analyse dauerte länger, da sie aufwendiger ist und mehr Ergänzungen an der Zwischendarstellung IML vollzieht.

Name	Zeit in s
glpk	3.42
jpeg	1.34
ncurses	1.31
sqlite	2.14
xvid	4.07

Tabelle 7.3: Laufzeiten der typsensitiven Fragment-Analyse

Name	Zeit in s
glpk	2.93
jpeg	1.07
ncurses	1.17
sqlite	1.92
xvid	3.34

Tabelle 7.4: Laufzeiten der typinsensitiven Fragment-Analyse

Tabelle 7.5 gibt sehr detailliert Auskunft über die Laufzeiten der Analysen der untersuchten Bibliotheken. Alle Angaben sind in Sekunden. Die Tabelle ist in drei Untertabellen mit den jeweiligen Zeigeranalysen von Steensgaard, Das und Andersen aufgegliedert. Bei der Andersen-Analyse fehlen an manchen Stellen Zeiten, da die Analyse wahrscheinlich aufgrund von Arbeitsspeichermangel nicht durchlaufen konnte und abgebrochen worden ist[14]. Auf die Andersen-Analyse aufbauende Analysen konnten daher leider nicht durchgeführt werden und sind

[14]Die Analysen wurden nach mehr als zehn Stunden Laufzeit vom System mit Signal 11 beendet, `man signal` gibt folgende Auskunft: `SIGSEGV 11 Core Invalid memory reference`

ecr	glpk tsen	glpk tins	jpeg tsen	jpeg tins	ncurses tsen	ncurses tins	sqlite tsen	sqlite tins	xvid tsen	xvid tins
ecr	21.38	21.59	3.4	5.27	3	3.37	19.27	18.91	14.89	11.24
cg	22.34	22.41	3.57	6.04	3.01	3.49	19.33	19.93	14.83	10.78
cfg	5.91	5.89	2.25	2.05	2.2	2.17	4.22	4.16	6.56	5.91
loc	25.98	25.96	6.16	9.57	4.88	5.7	31.63	30.36	18.94	16.74
issa	157.36	107.28	480.22	305.78	69.41	46.25	3114.31	1856.37	111.85	119.94

das	glpk tsen	glpk tins	jpeg tsen	jpeg tins	ncurses tsen	ncurses tins	sqlite tsen	sqlite tins	xvid tsen	xvid tins
das	41.12	39.67	16.77	11.47	3.82	3.36	60.24	54.37	18.48	15.33
cg	4.16	4.08	9.99	6.31	1.43	1.36	4.64	4.48	5.58	5.13
cfg	6.79	6.68	3.41	2.7	2.4	2.33	5.32	5.25	8.07	6.98
loc	211.9	209.1	165.53	92.05	15.6	13.52	235.93	222.01	491.58	186.21
issa	365.13	323.23	2193.37	826.34	102.14	72.69	4202.06	3129.81	624.81	257.89

and	glpk tsen	glpk tins	jpeg tsen	jpeg tins	ncurses tsen	ncurses tins	sqlite tsen	sqlite tins	xvid tsen	xvid tins
and	33132.73	42805.88	70874.82	4074.45	6.87	1409.71	69593.21	4119.38	12.08	39.83
cg	71.2	61.97	–	–	1.43	–	–	32.14	4.34	4.13
cfg	74.32	64.55	–	–	2.5	–	–	79.68	6.86	284.04
loc	70.69	64.43	–	–	2.49	–	–	1490.45	5.94	64.53
issa	245.44	115.34	–	–	40.96	–	–	–	36.7	179.38

Tabelle 7.5: Übersicht der Laufzeiten in Sekunden

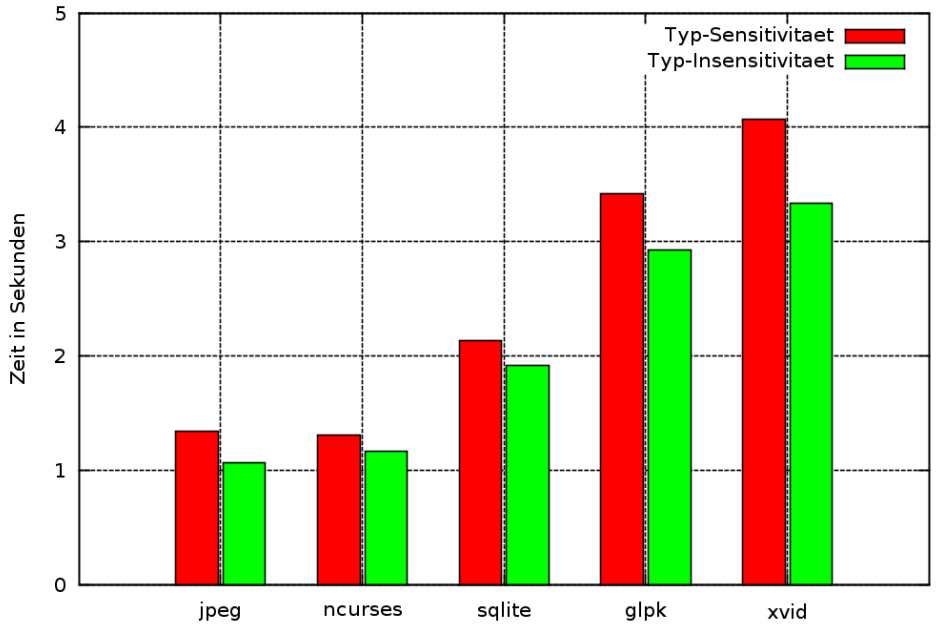

Abbildung 7.2: Laufzeit der Fragment-Analyse sortiert nach Größe der IML-Datei

mit einem „−“ versehen. Die typsensitive Fragment-Analyse ist mit *tsen*, die typinsensitive mit *tins* abgekürzt.

Auffällig an den Zeiten ist die Tatsache, dass je genauer die Zeigeranalyse ist, desto länger ist die Laufzeit. Auf die Zeigeranalyse aufbauende Analysen (Aufrufgraph cg, Kontrollflussgraph cfg, Seiteneffektberechnung und Erzeugung von Lokatoren loc, Interprozedurale SSA-Analyse issa) dauern ebenfalls länger.

Die typsensitive Fragment-Analyse erzeugt mehr Variablen `ph_var_t` und mehr Funktionen `ph_wrapper` im Vergleich zur typinsensitive Fragment-Analyse und ist dementsprechend langsamer. Auf die typsensitive Fragment-Analyse aufbauende Analysen sind dementsprechend ebenfalls langsamer. Weitere Analysen der Ergebnisse sollen zeigen, ob diese Investition eine höhere Genauigkeit erzielt.

7.1.2 Aufrufgraph

Tabelle 7.6 stellt die Ergebnisse der Aufrufgraphkonstruktion dar. Hierbei geht es um die **durchschnittliche Anzahl der Ziele** für indirekte Aufrufe. Die Zahlen stammen jeweils aus den Zeigeranalysen von Steensgaard, Das und Andersen (Abkürzung ECR, Das und And) und spiegeln die zwei unterschiedlichen Versionen typsensitive und typinsensitive Fragment-Analyse wider (Abkürzung tsen und tins). Die Zahlen in Klammern bedeuten, dass nicht für jeden indirekten Aufruf mindestens ein Ziel gefunden wurde, diese sind also mit Vorsicht zu betrachten[15].

[15] Auch wenn nicht für jeden Aufruf mindestens ein Ziel gefunden wurde, kann nicht direkt gefolgert werden, dass die Implementierung falsch ist. Im Fachpraktikum wurde vom Autor ein Modelchecker umgesetzt. Dabei wurden zwei generische Pakete (Liste und Hash) als Datenstrukturen

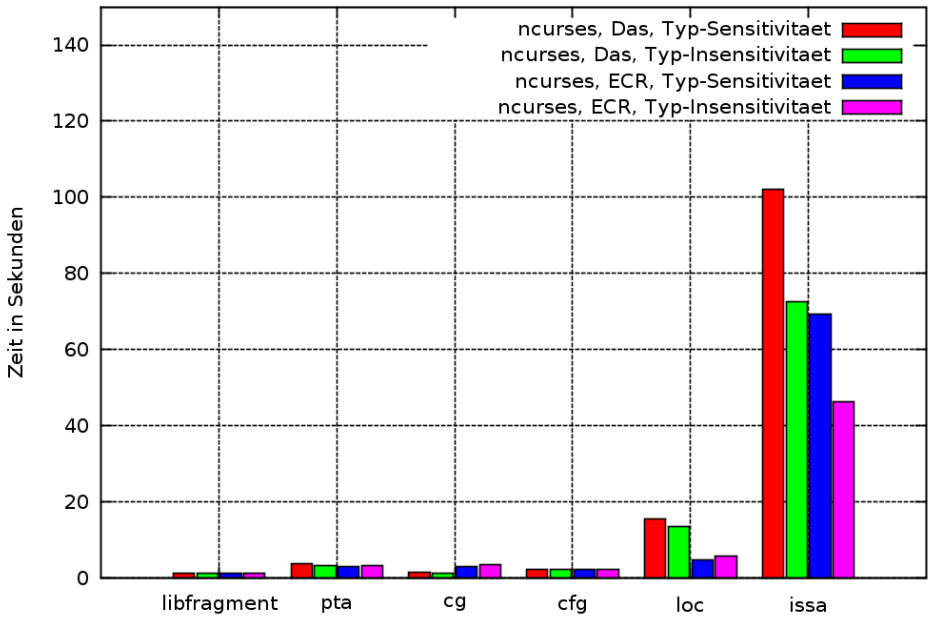

Abbildung 7.3: Übersicht der Laufzeiten in Sekunden für ncurses

Bei der Andersen-Analyse fehlen an manchen Stellen Ergebnisse, da die Analyse wahrscheinlich aufgrund von Arbeitsspeichermangel auch hier nicht durchlaufen konnte und abgestürzte[16]. AT steht für „Address Taken“ und ist der ungünstigste Fall. Dieser tritt ein, wenn für die Ziele alle Funktionen genommen werden, deren Adresse (irgendwo) genommen wurde. Die Spalte Das# ist das Ergebnis nach der Das-Analyse ohne die Fragment-Analyse — diese Spalte soll nur ein Richtwert sein, da z. B. Datenflüsse zu Schnittstellenfunktionen fehlen. Die letzte Spalte gibt die Anzahl der indirekten Aufrufe an, die in der jeweiligen Bibliothek vorkommen.

Die Tabelle 7.6 bestätigt die erste Vermutung: Je genauer[17] die Zeigeranalyse, desto präziser[18] ist das Ergebnis der Aufrufgraphkonstruktion.

Ebenso lässt die Tabelle die Vermutung zu, dass die typsensitive Analyse mit mehreren `ph_var_t` genauere Resultate liefert. Allerdings lässt sich diese Vermutung nicht mit Sicherheit bestätigen, da die Andersen-Analyse nicht bei allen Bibliotheken einwandfrei durchlief. Indirekte Aufrufe, deren Ziele nicht gefunden wurden, könnten das Ergebnis verfälscht haben[19].

geschrieben. Innerhalb dieser Pakete gibt es indirekte Aufrufe, die Ziele wurden ebenfalls nicht gefunden. Da jedoch der Durchschnitt gebildet wird, können nicht gefundene Ziele die Aussage verfälschen.

[16] Die Analysen wurden nach mehr als zehn Stunden Laufzeit vom System mit Signal 11 beendet, `man signal` gibt folgende Auskunft: `SIGSEGV 11 Core Invalid memory reference`

[17] siehe 2.2.1

[18] Je kleiner die Zahl, desto weniger Ziele wurden gefunden und folglich ist das Ergebnis präziser.

[19] Es wäre die logische Konsequenz noch mehr Bibliotheken zu betrachten. Leider blieb dazu zu wenig Zeit übrig, da die Analysen an sich viel Rechenzeit in Anspruch nehmen. Hinzu kommt das Suchen nach geeigneten Bibliotheken, die Kompilierung sowie die manuelle Erkennung der sichtbaren Variablen und Funktionen.

Name	ECR tsen ø	ECR tins ø	Das tsen ø	Das tins ø
glpk	26	25	11	11
jpeg	(99)	178	(166)	178
ncurses	(13)	(12)	7	14
sqlite	173	172	137	137
xvid	(14)	(16)	(7)	(7)

Name	And tsen ø	And tins ø	Das# ø	AT	Anz. i. Aufr.
glpk	11	11	3	35	27
jpeg	–	–	163	178	641
ncurses	2	–	(7)	19	15
sqlite	–	(12)	137	178	92
xvid	(1)	(1)	(7)	158	1085

Tabelle 7.6: Durchschnittliche Ziele für indirekte Aufrufe in Bibliotheken

7.1.3 Seiteneffektberechnung

Es wurden Messungen für die Seiteneffektberechnung der fünf untersuchten Bibliotheken durchgeführt. Dabei wurde die typsensitive mit der typinsensitiven Fragment-Analyse verglichen (Tabelle 7.7 und 7.8). Es wurden alle Funktionen betrachtet und jeweils der Durchschnitt der Mächtigkeit der drei Mengen must_def, may_def und may_use gebildet. Wie an must_def zu erkennen, ist die typsensitive Analyse marginal besser. Bei may_def und may_use ist es schwierig eine genaue Aussage zu treffen, da die künstlichen Variablen `ph_var` sich in der may_def und may_use wiederfinden und die typsensitive Fragment-Analyse im Vergleich zu der typinsensitiven Fragment-Analyse mehrere Variablen `ph_var` anstatt nur einer hat.

7.1.4 ISSA

Nun werden einige Metriken der ISSA-Analyse präsentiert. Die typsensitive wird der typinsensitiven Fragment-Analyse gegenüber gestellt (Tabelle 7.9, 7.10 und 7.11[20]). Die Anzahl der Zuweisungen hat sich erhöht, da bei der typsensitiven Analyse mehrere Funktionen `ph_wrapper` benutzt werden und innerhalb von `ph_wrapper` Zuweisungen existieren. Dementsprechend erhöhen sich auch die Lesezugriffe, die Pre Call Links, Post Call Links und die Anzahl der Lokatoren in den ϕ-Knoten. Mit Pruned markierte Zeilen stellen die Anzahl der Lokatoren dar, die nachträglich wieder entfernt werden können, da sie unnötig sind. Je höher die Zahl, desto besser ist die Analyse. Die typsensitive Fragment-Analyse konnte hierbei deutlich besser

[20]Die Zahlen der Bibliothek xvid sind mit Vorsicht zu betrachten, da bereits nach der IML-Generierung entdeckt wurde, dass xvid für dieselbe Funktion manchmal drei Routine-Knoten hat.

Name	must_def ø	may_def ø	may_use ø	Anz. `ph_var`
glpk	0.03	67.1	73.97	53
jpeg	0.08	115.58	128.64	77
ncurses	0.11	144.68	163.79	121
sqlite	0.14	342.63	392.84	153
xvid	0	23.68	30.04	40

Tabelle 7.7: Seiteneffektberechnung für Bibliotheken, typsensitiv

Name	must_def ø	may_def ø	may_use ø
glpk	0.02	32.5	39.43
jpeg	0.08	22.79	32.93
ncurses	0.1	59.59	81.09
sqlite	0.14	136.57	186.89
xvid	0	29.51	36.79

Tabelle 7.8: Seiteneffektberechnung für Bibliotheken, typinsensitiv

abschneiden. Auch hier ist jedoch die Aussage mit Vorsicht zu genießen, da es mehrere globale Variablen `ph_var` sind, die mitgezählt werden. Das Verhältnis der may_def- zu must_def-Lokatoren hat sich bei der typsensitiven Fragment-Analyse verbessert, es sind mehr must_def und weniger may_def. Dies deutet auf eine höhere Genauigkeit der typsensitiven Fragment-Analyse hin.

7.1.5 Rollenpropagierung für Aufrufgraph

Zuletzt wird nun das Ergebnis der Rollenpropagierung erörtert (Tabelle 7.12). Angegeben werden die Anzahl der sichtbaren Funktionen, die Laufzeit in Sekunden, die Anzahl der erzeugten indirekten Aufrufe für Rollen, die an sichtbaren Funktionen hängen und die Anzahl der Rollen insgesamt (also auch solche, die an unsichtbaren Funktionen hängen). Für jpeg und sqlite fehlen die Ergebnisse, hier wird ein Fehler vermutet, der noch nicht zu identifizieren war[21].

Wie in Abbildung 5.3 dargestellt, konnte festgestellt werden, dass die Tiefe des Rollenpropagierungsgraphen häufig eins ist. Funktionszeiger als Parameter von

[21] Die Analyse beider Bibliotheken brach aufgrund von Arbeitsspeichermangel an zwei unterschiedlichen Stellen nach mehr als zehn Stunden ab. Sie brach im Bereich des generischen Pakets für Rollenpropagierung ab. Vermutlich hängt dies mit der neulich umgestellten Modellierung der Sequenzen von Values zusammen. Die Rollenpropagierung folgt nach der ISSA-Analyse, davor liegt eine ganze Kette von Analysen. Es könnte auch ein Fehler in einer der Analysen zuvor weitergereicht worden sein. Eine aufwendige Fehlersuche konnte bisher noch nicht durchgeführt werden.

Anzahl der	glpk tsen	glpk tins	jpeg tsen	jpeg tins
Zuweisungen	14834	14589	7184	5780
Leere Zuw.	12321	12228	1353	8
Lesezugriffe	65135	64743	21954	18913
Leere Lesezugr.	53999	53789	3632	484
ϕ-Tabellen	1297	1289	1406	1459
Pre Call Links	11553	11527	1900	1340
Leere Pre Call L.	9750	9787	293	192
Post Call Links	11553	11527	1900	1340
Leere Post Call L.	9758	9796	303	202
may_def Lok.	35265	37165	32567	42412
must_def Lok.	1582	1476	3681	3527
Pr. Def. Lok	-277	-300	-60	-50
Lok. bei Lesezugr.	139724	146954	87663	113271
Lok. in ϕ-Knoten	165134	52727	153234	33782
Pr. Lok. in ϕ-Knoten	-59591	-931	-66419	-2637
Lok. in Pre Call L.	273362	82296	185598	42180
Pr. Lok. in Pre Call L.	-99048	-318	-79260	-552
Lok. In Post Call L.	268962	78696	167842	29206
M. Lok. In Post Call L.	736	617	217	74
Pr. Lok. In Post Call L.	-99148	-488	-74028	-635

Lok. = Lokatoren, Pr. = Pruned, L. = Links, Def. = Definitionen, M. = Must

Tabelle 7.9: ISSA-Metriken für Bibliotheken I

Schnittstellenfunktionen werden also häufig direkt in derselben Schnittstellenfunktion aufgerufen, der sie übergeben wurde. Der längste Pfad des Rollenpropagierungsgraphen von betrachteten Bibliotheken hat die Länge neun.

Da es sehr wenige Rollen indirekte Aufrufe sind, wurden sie manuell nachgeprüft. Sie scheinen korrekt zu sein. Als Beispiel sei hier xvid mit den drei sichtbaren Funktionen für Komprimieren und Dekomprimieren genannt:

```
1 extern int xvid_global(void *handle, int opt, void *param1, void *param2);
2 extern int xvid_decore(void *handle, int opt, void *param1, void *param2);
3 extern int xvid_encore(void *handle, int opt, void *param1, void *param2);
```

Für die Parameter `void *handle` der Funktionen `xvid_global` und `xvid_encore` konnte identifiziert werden, dass sie weitergereicht werden und irgendwo in der Bibliothek indirekt aufgerufen werden. Zu erwarten war dies auch bei `xvid_decore`. Hier wird noch manuell nachgeforscht, warum eine Rolle an `xvid_decore` nicht angekommen ist.

Für glpk scheint das Ergebnis richtig zu sein. Sichtbare Funktionen von glpk haben als Parameter keine Funktionszeiger oder `void*`-Zeiger.

Anzahl der	ncurses tsen	ncurses tins	sqlite tsen	sqlite tins
Zuweisungen	4622	4229	11140	10395
Leere Zuw.	817	778	311	22
Lesezugriffe	18795	18258	43690	42505
Leere Lesezugr.	4623	4477	978	386
ϕ-Tabellen	1677	1685	3922	3923
Pre Call Links	2699	2685	6544	6453
Leere Pre Call L.	1323	1303	593	593
Post Call Links	2699	2685	6544	6453
Leere Post Call L.	1543	1528	1056	1012
may_def Lok.	53851	61853	347237	348493
must_def Lok.	2696	2449	7270	6951
Pr. Def. Lok	-131	-110	-232	-196
Lok. bei Lesezugr.	251413	299274	1247617	1255478
Lok. in ϕ-Knoten	180741	87074	1107169	470198
Pr. Lok. in ϕ-Knoten	-50510	-3512	-348560	-22357
Lok. in Pre Call L.	199614	99416	2370751	1126336
Pr. Lok. in Pre Call L.	-53820	-2740	-661359	-26761
Lok. In Post Call L.	173371	69995	2089406	844348
M. Lok. In Post Call L.	537	295	1355	1059
Pr. Lok. In Post Call L.	-53143	-2791	-662151	-27400

Lok. = Lokatoren, Pr. = Pruned, L. = Links, Def. = Definitionen, M. = Must

Tabelle 7.10: ISSA-Metriken für Bibliotheken II

ncurses als grafische Bibliothek für die Konsole bekommt dagegen des Öfteren Funktionszeiger als Parameter, um über diese einen Rückruf auszuführen.

7.2 Betrachtete Benutzerprogramme

Für den Ansatz der Fragment-Analyse als modularer Analyse, die den schlimmsten Fall annimmt, ist ein Vergleich zur Gesamtprogramm-Analyse wichtig. In diesem Abschnitt wird ein Vergleich gezogen. Gewählt wurde die Bibliothek gtk mit fünf verschiedene Benutzerprogrammen unterschiedlicher Größe[22]. Auf der einen Seite werden die Benutzerprogramme mit gtk gelinkt, auf der anderen Seite wird gtk getrennt analysiert. Beide Tests durchlaufen die in Abbildung 5.1 dargestellte Kette von Analysen bis zur ISSA-Analyse. Als Zeigeranalyse wurde die Das-Implementierung gewählt und die typinsensitive Version der Fragment-Analyse benutzt. Tabelle 7.13 stellt die Benutzerprogramme kurz mit der Größe der IML-Datei vor dem Linken und nach dem Linken dar. Die Bibliothek gtk hat als IML-

[22]Hierbei konnte auf in Bauhaus existierende Testprogramme zurückgegriffen werden

Anzahl der	xvid tsen	xvid tins
Zuweisungen	18321	13439
Leere Zuw.	6579	1807
Lesezugriffe	95741	85075
Leere Lesezugr.	22818	12285
ϕ-Tabellen	2563	2617
Pre Call Links	4789	3705
Leere Pre Call L.	2119	1735
Post Call Links	4789	3705
Leere Post Call L.	2372	1899
may_def Lok.	66993	81488
must_def Lok.	7443	7364
Pr. Def. Lok	-460	-486
Lok. bei Lesezugr.	251634	299842
Lok. in ϕ-Knoten	37166	45004
Pr. Lok. in ϕ-Knoten	-2876	-2943
Lok. in Pre Call L.	49485	61073
Pr. Lok. in Pre Call L.	-226	-621
Lok. In Post Call L.	38761	48629
M. Lok. In Post Call L.	190	111
Pr. Lok. In Post Call L.	-664	-644

Lok. = Lokatoren, Pr. = Pruned, L. = Links, Def. = Definitionen, M. = Must

Tabelle 7.11: ISSA-Metriken für Bibliotheken III

Datei eine Größe von 43 MB. Die Unverhältnismäßigkeit der Größen vor und nach dem Linken spiegelt sich gerade in kleinen Benutzerprogrammen wie codebreaker wieder.

7.2.1 Laufzeiten

Nun werden die Laufzeiten der verschiedenen Analysen für die Benutzerprogramme betrachtet, eine Gegenüberstellung mit gtk erfolgt ebenfalls (Tabelle 7.14). Das große Programm gimp gelinkt mit gtk stellt die größte Herausforderung an die Analysen. Die ISSA-Analyse konnte für gimp (Größe der IML-Datei gelinkt mit gtk 145 MB) nicht ausgeführt werden,der benötigte Arbeitsspeicher ist zu groß. Hier lohnt es sich, eine getrennte Analyse durchzuführen. Für die ISSA-Analyse musste der Analysenanwender bei den anderen Programmen jeweils 6–12 Stunden warten. Die getrennte gtk-Analyse musste nur einmal ausgeführt werden und dauerte 17,5 Stunden.

Name	Sichtb. Fkt	Laufzeit in s	Anz. erzeugte Aufr.	Anz. Rollen insg.
glpk	137	208	0	24
jpeg	48	–	–	–
ncurses	154	275	7	18
sqlite	108	–	–	–
xvid	3	89	2	137

Tabelle 7.12: Ergebnisse der Rollenpropagierung

Name	SLOC in k	IML v. d. Linken in MB	IML n. d. Linken in MB
bluefish	44	14	55
codebreaker	1,9	1	42
euler	24	10	51
gimp	590	100	145
gqview	54	17	59
gtk	109	44	–

Tabelle 7.13: Benutzerprogramme gelinkt mit gtk

7.2.2 Aufrufgraph

Bei der Aufrufgraphkonstruktion gibt die Anzahl der Ziele für indirekte Aufrufe die Qualität der Analyse wieder (Tabelle 7.15). In Spalte zwei wurde für alle indirekten Aufrufe der Durchschnitt der Ziele gebildet. Spalte drei gibt den schlimmsten Fall an, nämlich die Anzahl der Funktionen, deren Adressen überhaupt genommen wurden. Spalte vier zeigt insgesamt die Anzahl der indirekten Aufrufe.

Im Vergleich zu den Benutzerprogrammen ist bei der getrennten Analyse von gtk eine deutliche Verschlechterung zu erkennen. Die Ziele wurden im Durchschnitt im Vergleich zu den Benutzerprogrammen mehr als verdoppelt. Zu beachten ist,

Name	das	cg	cfg	loc	issa
bluefish	134	24	89	2790	41859
codebreaker	91	19	57	2257	24287
euler	145	24	74	2762	37559
gimp	993	78	691	12552	–
gqview	203	28	81	2995	44445
gtk	183	28	1544	2673	63056

Tabelle 7.14: Laufzeit in Sekunden von Benutzerprogrammen gelinkt mit gtk und gtk getrennt

dass gerade kleine Benutzerprogramme wie codebreaker wahrscheinlich nicht alle Schnittstellenfunktionen von gtk benutzen und z. B. im Vergleich zu gimp weniger Datenflüsse existieren.

Name	Das ø	AT	Anz. i. Aufr.
bluefish	340	787	493
codebreaker	303	680	475
euler	335	778	551
gimp	339	1243	1455
gqview	375	802	523
gtk	826	1052	475

Tabelle 7.15: Durchschnitt der Ziele für indirekte Aufrufe in Benutzerprogrammen

7.2.3 Seiteneffektberechnung

Die Seiteneffektberechnung liefert als Ergebnis für jede Funktion drei Mengen: must_def, may_def und may_use (Tabelle 7.16). In den Spalten zwei, drei und vier wurde jeweils der Durchschnitt der Größe der Mengen für jede Bibliotheksfunktion von gtk gebildet. Der Durchschnitt von may_def und may_use für die getrennte Analyse ist niedriger als bei der gemeinsamen Analyse, da es nur eine `ph_var` gibt, die alle Variablen im Benutzerprogramm repräsentierte.

Wie bereits erwähnt, konnte gimp für die ISSA-Analyse nicht durchgeführt werden. Die Zahlen für gimp stammen aus der Seiteneffekt-Berechnung vor der ISSA-Analyse, während die Zahlen der anderen Programme nach der ISSA-Analyse abgelesen wurden. Mittels der ISSA-Form wird die Seiteneffekt-Berechnung verbessert.

Erfreulich ist die Tatsache, dass die Zahl sehr nah an die tatsächliche Analyse herankommt, ohne sie in diesem Fall fünf Mal analysieren zu müssen. Außerdem kann eine Tendenz festgestellt werden: Je größer das Benutzerprogramm, desto ungenauer wird die getrennte Analyse. Dies ist nicht verwunderlich, da die Fragment-Analyse von einem kleinstmöglichen Benutzerprogramm als Ergänzung ausgeht.

7.2.4 ISSA

Nun werden einige Metriken der ISSA-Analyse betrachtet (Tabelle 7.17). Die getrennte Analyse weist hier Schwächen auf. So konnten nachträglich weniger Lokatoren entfernt werden (Pruned). Das Verhältnis von may_def zu must_def Lokatoren in der getrennten Analyse hat sich im Vergleich zur gemeinsamen Analyse mit Benutzerprogrammen verschlechtert. Es sind weniger must_def und mehr may_def geworden. Der Vergleich ist hier etwas schwierig, da bei den Benutzerprogrammen

Name	must_def ø	may_def ø	may_use ø
bluefish	0.04	557.47	695.21
codebreaker	0.04	452.56	524.39
euler	0.04	493.63	576.99
gimp*	0	1530.41	1682.67
gqview	0.04	492.16	639.28
gtk	0.09	428.66	536.2

* gimp gelinkt mit gtk konnte wegen Arbeitsspeichermangel für die ISSA-Analyse nicht durchgeführt werden.

Tabelle 7.16: Seiteneffektberechnung für Benutzerprogramme

noch der IML-Teil des Benutzerprogramms hinzugekommen ist. Nichtsdestotrotz überwiegt in allen vier Fällen der Anteil von gtk (Tabelle 7.13), sodass die Aussagekraft nicht zu sehr abgeschwächt wird.

Anzahl der	bluefish	codebreaker	euler	gqview	gtk
Zuweisungen	26988	21036	31501	31130	23469
Leere Zuw.	4508	4113	4035	4801	339
Lesezugriffe	176748	140551	177274	193632	156375
Leere Lesezugr.	40370	33407	34379	41456	14824
ϕ-Tabellen	12323	10342	13621	14055	11853
Pre Call Links	38875	23674	30120	40864	29749
Leere Pre Call L.	20605	12197	13696	22565	11259
Post Call Links	38875	23674	30120	40864	29749
Leere Post Call L.	25007	16317	18716	27250	14394
may_def Lok.	2492631	1620657	2390224	2427826	2794763
must_def Lok.	15335	11125	19534	18385	14960
Pr. Def. Lok	-67196	-43299	-57694	-62170	-2727
Lok. bei Lesezugr.	21541634	15032520	18624815	19021737	19313925
Lok. in ϕ-Knoten	5821143	4145911	5706347	5672553	3409840
Pr. Lok. in ϕ-Knoten	-729831	-523607	-777455	-841469	-18876
Lok. in Pre Call L.	11475987	5176891	8602787	10434503	9042269
Pr. Lok. in Pre Call L.	-645601	-86354	-171749	-348052	-18806
Lok. In Post Call L.	8340016	3664631	6191095	7288215	6193660
M. Lok. In Post Call L.	3639	2784	3283	3611	3433
Pr. Lok. In Post Call L.	-635771	-331956	-794776	-872713	-22256

Lok. = Lokatoren, Pr. = Pruned, L. = Links, Def. = Definitionen, M. = Must

Tabelle 7.17: ISSA-Metriken für Benutzerprogramme

8 Zusammenfassung und Ausblick

Bei der statischen Analyse von Bibliotheken sollte beachtet werden, dass Datenflüsse fehlen. Das Fehlen von Datenflüssen von direkten Aufrufen ist darauf zurückzuführen, dass es kein Benutzerprogramm gibt, welches die Bibliothek benutzt. Datenflüsse von indirekten Aufrufen sind nicht vorhanden, da die eigentlichen Ziele im Benutzerprogramm zu finden sind. Benutzerprogramme können die angebotenen Klassen einer Bibliothek ableiten. Für virtuelle polymorphe Aufrufe können die Ziele zur Analysezeit der Bibliothek ebenfalls nicht bestimmt werden.

Einfache Ansätze weisen Schwächen auf. Sie sind nicht konservativ, unpräzise oder zeitaufwendig. Die Größe einer Bibliothek könnte im Vergleich zu einem Benutzerprogramm überproportional sein. Hinzu kommt die transitive Abhängigkeit von Bibliotheken.

Der vorgestellte Ansatz Component-Level-Analysis ist ersichtlich für die Berechnung der gültigen Definitionen. Die zentrale Idee ist die Zusammenfassung von Transferfunktionen. Die Component-Level-Analysis konnte konzeptionell für die Zeigeranalyse angepasst werden. Wie sie und die entsprechende Transferfunktion aussehen, z. B. für Seiteneffektberechnung oder Aufrufgraphkonstruktion, bleibt offen. Zudem ist der Ansatz aufwendig in der Implementierung für Analysenprogrammierer.

Die Fragment-Analyse nimmt den schlechtesten Fall an und ergänzt eine Bibliothek mit Platzhaltern `ph_*` zu einem vollständigen Programm. Die Schwächen der Fragment-Analyse wurden in dieser Diplomarbeit diskutiert. Die neu eingeführte Typsensitivität der Fragment-Analyse gilt als Verbesserung des ursprünglichen Ansatzes. Sie ist jedoch in einer typschwachen Programmiersprache wie C nicht konservativ. Eine einfach zu konfigurierende Datei sollte bei der Aufrufgraphkonstruktion eine weitere Verbesserung bringen. Das Konzept für C++ wurde entwickelt und ausführlich dargelegt. Es findet sich jedoch aus Zeitgründen in der Implementierung nicht wieder.

Als Speicherformat für die Weiterverwendung wurde IML gewählt. In Bauhaus existiert bereits eine Infrastruktur für das Laden, Speichern und Modifizieren von IML-Knoten. So liegt beispielsweise das Ergebnis der Seiteneffektberechnung nach der ISSA-Analyse bereits vor. Dies wird als IML mit dem Benutzerprogramm gelinkt. Hierzu musste der Linker angepasst werden, der die neuen IML-Knoten bisher noch nicht behandeln konnte.

Die Rollenpropagierung wurde für die Aufrufgraphkonstruktion benutzt. Mit dem Ergebnis der Rollenpropagierung konnte bestimmt werden, welcher Parameter einer Schnittstellenfunktion als Rückruf agiert. Das Ergebnis der Rollenpropagierung

wird in Form von künstlichen indirekten Aufrufen gespeichert und zu der IML hinzugefügt.

Bei Messungen und Evaluationen konnte festgestellt werden, dass die Laufzeit der Fragment-Analyse annehmbar ist. Selbst für gtk blieb die Analyse unter 17 Sekunden. Dies ist nicht verwunderlich, da die typsensitive Fragment-Analyse in $\mathcal{O}(n^2)$ ist, mit n als Anzahl der zu betrachtenden Funktionen der Bibliothek — n blieb in der Regel klein.

Die Qualität der Aufrufgraphkonstruktion hängt von der Genauigkeit der Zeigeranalyse ab. Tendenziell lässt sich sagen, dass die typsensitive Fragment-Analyse bessere Resultate erzielt.

Bei der Seiteneffektberechnung ist ein Vergleich zwischen der typsensitiven und der typinsensitiven Fragment-Analyse schwierig, da die verschiedenen `ph_var` in den Mengen must_def, may_def und may_use enthalten sind und das Ergebnis trüben könnten.

Die Metriken der ISSA-Analyse dagegen weisen auf eine Verbesserung der Fragment-Analyse mit der Typsensitivität.

Das Ergebnis der Rollenpropagierung war überschaubar. An dieser Stelle konnten die Ergebnisse manuell überprüft werden. In der Regel werden Funktionszeiger nicht weitergereicht, sondern direkt in der aufgerufenen Funktion aufgerufen. Der längste Pfad des Rollenpropagierungsgraphen von betrachteten Bibliotheken hat die Länge neun.

Bei der Evaluation von Benutzerprogrammen war bemerkenswert, dass ein kleines Programm wie codebreaker vor dem Linken mit gtk eine IML-Größe von 1 MB besitzt, nach dem Linken aber auf 42 MB gewachsen ist. Es wurde auch an die Grenzen der Analysen gestoßen, denn gimp mit gtk verbunden besitzt eine IML-Größe von 145 MB. Die ISSA-Analyse von gimp verbunden mit gtk konnte aufgrund von Arbeitsspeichermangel nicht mehr ausgeführt werden. Die Laufzeit der auf der Fragment-Analyse aufbauenden Analysen ist etwas schlechter als die Analyse eines Benutzerprogramms mit gtk. Jedoch müssen die Analysen im Vergleich zu den Analysen der Benutzerprogrammen nur einmal ausgeführt werden.

Das Ergebnis der Aufrufgraphkonstruktion mit der Fragment-Analyse ist deutlich schlechter als das Ergebnis der Gesamtprogramm-Analyse.

Das Resultat der Seiteneffektberechnung ist annähernd gleich, da bei der typinsensitiven Fragment-Analyse nur eine Variable `ph_var` in den Mengen must_def, may_def und may_use enthalten ist.

Dagegen verschlechtert sich das Ergebnis der ISSA-Analyse.

Als Fazit kann gesagt werden, dass — trotz Verbesserung — die Fragment-Analyse spürbar schlechtere Ergebnisse liefert als die Gesamtprogramm-Analyse. Aufgrund der Vorteile der modularen Vorgehensweise sind die schlechteren Ergebnisse aber tragbar.

Weitere Arbeiten, die an diese Arbeit anschließen, können das ausgearbeitete Konzept für C++ umsetzen und verfeinern. Es bedarf außerdem genauere Untersuchungen der Kompaktifizierung, um die Ergebnisse im Benutzerprogramm vollständig nutzen und Analysen im Benutzerprogramm beschleunigen oder verbessern zu können.

8.1 Danksagungen

Bei der Erstellung dieser Diplomarbeit gab es Hilfestellungen von verschiedenen Personen. Einen besonderen Dank möchte ich allen aussprechen. Namentlich sollen hier nur einige genannt werden:

- Prof. Dr. Erhard Plödereder für die Übergabe eines interessanten Themas sowie die zahlreichen Anregungen und Ratschläge während des Zwischenvortrags.
- Stefan Staiger für die sehr gute Betreuung, hilfreiche Denkanstöße und vor allem hohe Diskussionsbereitschaft sowohl bei technischen als auch theoretischen Problemen.
- Allen Mitarbeitern der Abteilung Programmiersprachen und Übersetzerbau, die immer hilfsbereit sind.
- Klemens Krause für das einwandfreie Funktionieren der Server und Terminals, die bei der Diplomarbeit zum Einsatz kamen.
- Carolin Christow, Caroline Hande, Alexander Maier, Aline Sagrabelny, Bernhard Schmitz und Holger Schröck für ihre Hilfe beim Korrekturlesen.
- Ein ganz großer Dank gilt meinen Eltern. Nur durch sie war mein Studium möglich. Sie haben mich stets mit Rat und Tat unterstützt.
- Zum Schluss noch einen Dank auch an den Fußballgott für die tolle EM kurz vor der Abgabe der Diplomarbeit ;-)

Literaturverzeichnis

[And94] Andersen, L. O.: *Program Analysis and Specialization for the C Programming Language*, DIKU, University of Copenhagen, Diss., Mai 1994. – (DIKU report 94/19)

[Art88] Arthur, Lowell J.: *Software Evolution: The Software Maintenance Challenge*. New York, NY, USA: Wiley-Interscience, 1988. – ISBN 0–471–62871–9

[CC02] Cousot, Patrick; Cousot, Radhia: Modular Static Program Analysis. In: *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, 2002, S. 159–178

[CFR+91] 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 (TOPLAS)* 13 (1991), Nr. 4, S. 451–490

[CK88] Cooper, Keith D.; Kennedy, Ken: Interprocedural Side-Effect Analysis in Linear Time. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1988, S. 57–66

[Das00] Das, Manuvir: Unification-based pointer analysis with directional assignments. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, S. 35–46

[FH79] Fjelstad, R.K; Hamlen, W.T.: Application program maintenance study report to our respondents. In: *Proceedings of the GUIDE 48*, 1979, S. 1284–1288

[Fro06] Frohn, Simon: *Konzeption und Implementierung einer Zeigeranalyse für C und C++*, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Diplomarbeit, Januar 2006. – 107 S.

[GHJV95] Gamma, Erich; Helm, Richard; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995

[GR07] Gopan, Denis; Reps, Thomas W.: Low-Level Library Analysis and Summarization. In: *Computer Aided Verification*, 2007, S. 68–81

[HRB90] Horwitz, Susan; Reps, Thomas W.; Binkley, David: Interprocedural Slicing Using Dependence Graphs. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (1990), Nr. 1, S. 26–60

[KG98] Koschke, Rainer; Girard, Jean-Francois: An Intermediate Representation for Reverse Engineering Analyses. In: *WCRE*, 1998, S. 241–250

[KR88] Kernighan, Brian W.; Ritchie, Dennis: *The C Programming Language, Second Edition*. Prentice-Hall, 1988. – ISBN 0–13–110370–9

[LR92] Landi, William; Ryder, Barbara G.: A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992, S. 235–248

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. – ISBN 1–55860–320–4

[NNH99] Nielson, Flemming; Nielson, Hanne R.; Hankin, Chris: *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. – ISBN 3540654100

[RC00] Rountev, Atanas; Chandra, Satish: Off-line Variable Substitution for Scaling Points-to Analysis. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, S. 47–56

[RHS95] Reps, Thomas W.; Horwitz, Susan; Sagiv, Shmuel: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: *Symposium on Principles of Programming Languages*, 1995, S. 49–61

[RKM06] Rountev, Atanas; Kagan, Scott; Marlowe, Thomas J.: Interprocedural Dataflow Analysis in the Presence of Large Libraries. In: *International Conference on Compiler Construction*, 2006 (LNCS 3923), S. 2–16

[RMR03] Rountev, Atanas; Milanova, Ana; Ryder, Barbara G.: Fragment Class Analysis for Testing of Polymorphism in Java Software. In: *International Conference on Software Engineering*, 2003, S. 210–220

[RMR04] Rountev, Atanas; Milanova, Ana; Ryder, Barbara G.: Fragment Class Analysis for Testing of Polymorphism in Java Software. In: *IEEE Transactions on Software Engineering* 30 (2004), Nr. 6, S. 372–387

[Rou02] Rountev, Atanas: *Dataflow Analysis of Software Fragments*, Rutgers University, Diss., August 2002

[RR00] Rountev, Atanas; Ryder, Barbara G.: Practical Points-to Analysis for Programs Built with Libraries / Department of Computer Science, Rutgers University. Version: Februar 2000. `ftp://athos.rutgers.edu/pub/technical-reports/dcs-tr-410.ps.Z`. 2000 (DCS-TR-410). – Forschungsbericht

[RR01] Rountev, Atanas; Ryder, Barbara G.: Points-to and Side-effect Analyses for Programs Built with Precompiled Libraries. In: *International Conference on Compiler Construction*, 2001 (LNCS 2027), S. 20–36

[RRL99] Rountev, Atanas; Ryder, Barbara G.; Landi, William: Data-Flow Analysis of Program Fragments. In: *ESEC / SIGSOFT FSE*, 1999, S. 235–252

[RSX08] Rountev, Atanas; Sharp, Mariana; Xu, Guoqing: IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In: *International Conference on Compiler Construction*, 2008 (LNCS 4959), S. 53–68

[RVP06] Raza, Aoun; Vogel, Gunther; Plödereder, Erhard: Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering. In: *Ada-Europe*, 2006, S. 71–82

[Sha07] Sharp, Mariana: *Static Analyses for Java in the Presence of Distributed Components and Large Libraries*, The Ohio State University, Diss., 2007

[SP81] 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

[SRH96] Sagiv, Shmuel; Reps, Thomas W.; Horwitz, Susan: Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. In: *Theor. Comput. Sci.* 167 (1996), Nr. 1&2, S. 131–170

[Sta07a] Staiger, Stefan: Reverse Engineering of Graphical User Interfaces Using Static Analyses. In: *WCRE*, 2007, S. 189–198

[Sta07b] Staiger, Stefan: Static Analysis of Programs with Graphical User Interface. In: *CSMR*, 2007, S. 252–264

[Ste96] Steensgaard, Bjarne: Points-to Analysis in Almost Linear Time. In: *Symposium on Principles of Programming Languages*, 1996, S. 32–41

[Str00] Stroustrup, Bjarne: *Die C++-Programmiersprache*. Addison-Wesley (Deutschland) Gmbh, 2000

[SVKW07] Staiger, Stefan; Vogel, Gunther; Keul, Steffen ; Wiebe, Eduard: Interprocedural Static Single Assignment Form. In: *WCRE*, 2007, S. 1–10

[Tar72] Tarjan, Robert E.: Depth-First Search and Linear Graph Algorithms. In: *SIAM J. Comput.* 1 (1972), Nr. 2, S. 146–160

[Wei84] Weiser, Mark: Program Slicing. In: *IEEE Transactions on Software Engineering* 10 (1984), Nr. 4, S. 352–357

[WL95] Wilson, Robert P.; Lam, Monica S.: Efficient Context-Sensitive Pointer Analysis for C Programs. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995, S. 1–12

Tabellenverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Minh Cuong Tran)